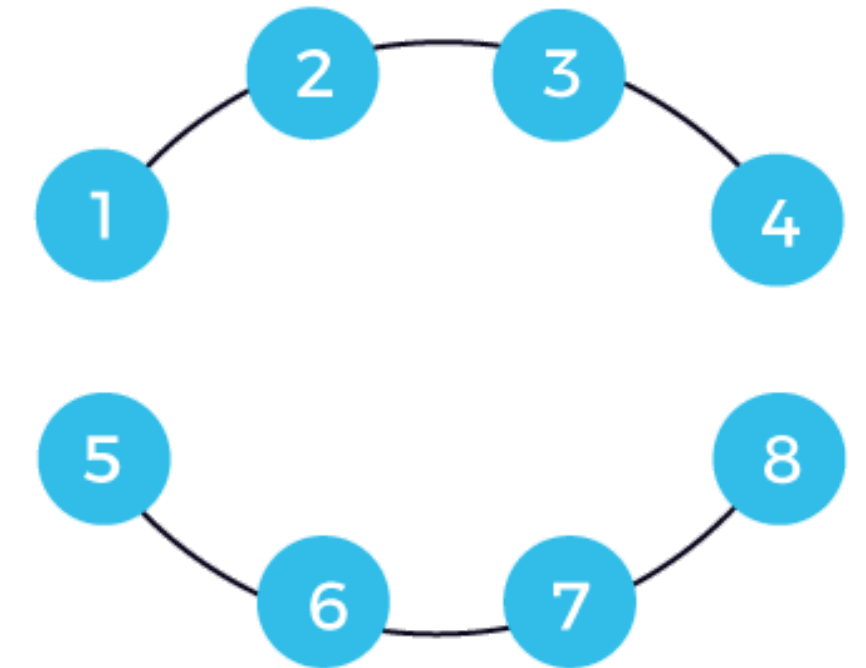
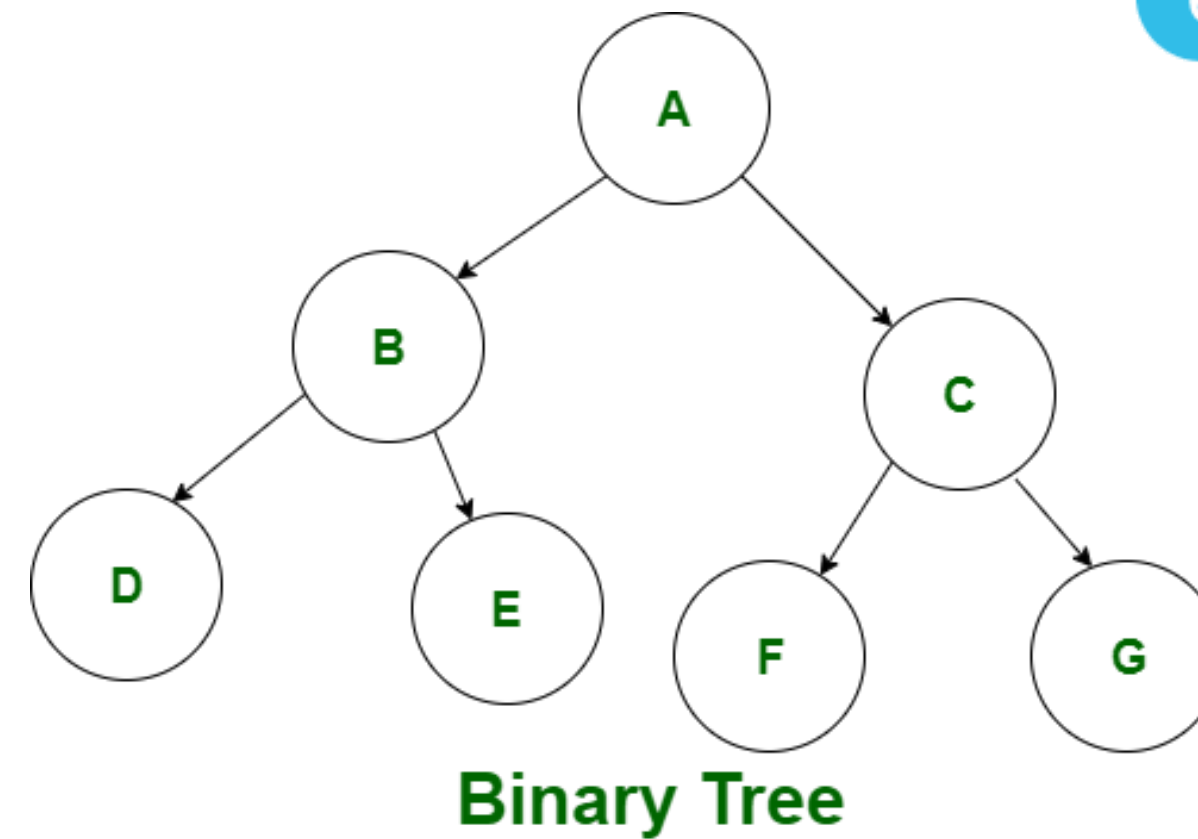


ESTRUCTURAS DE DATOS

Integrantes

- Deivid Farid Ardila Herrera
- Diego Alejandro Arévalo Arias
- Ángel David Beltrán García
- Cristofer Damián Camilo Ordoñez Osa



INTRODUCCIÓN



Bocu, Bogotá cultural, busca dar a los usuarios y a los expositores un espacio donde dar a conocer sus eventos, talleres, actividades, etc. Todo centrado en una misma app donde se podrá descubrir y buscar diferentes eventos en la ciudad de una manera intuitiva y de fácil uso.



01

BST Vs AVL

02

Montículos n-arios

03

Conjuntos Disjuntos

04

Implementación en la app

BST Y AVL



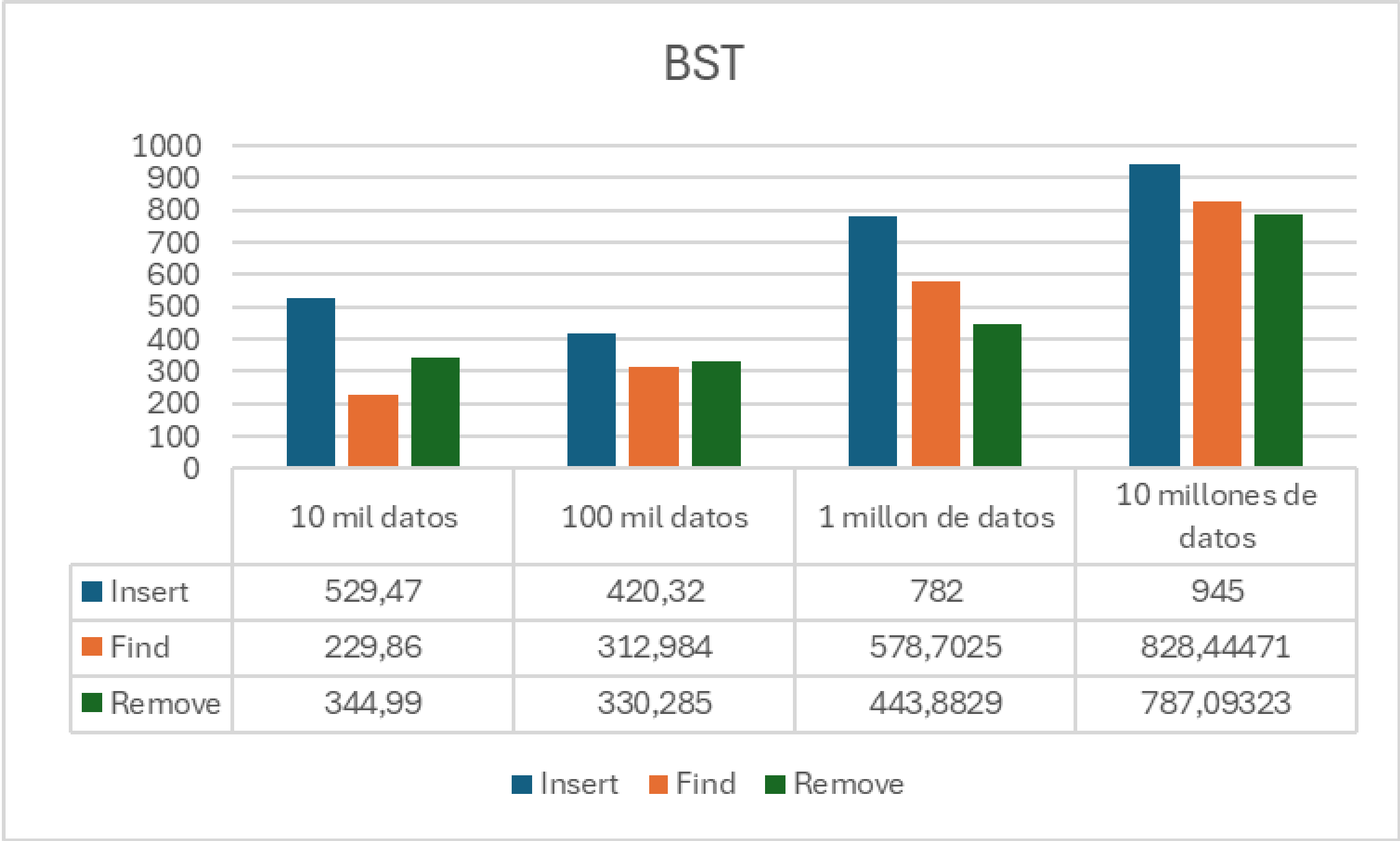
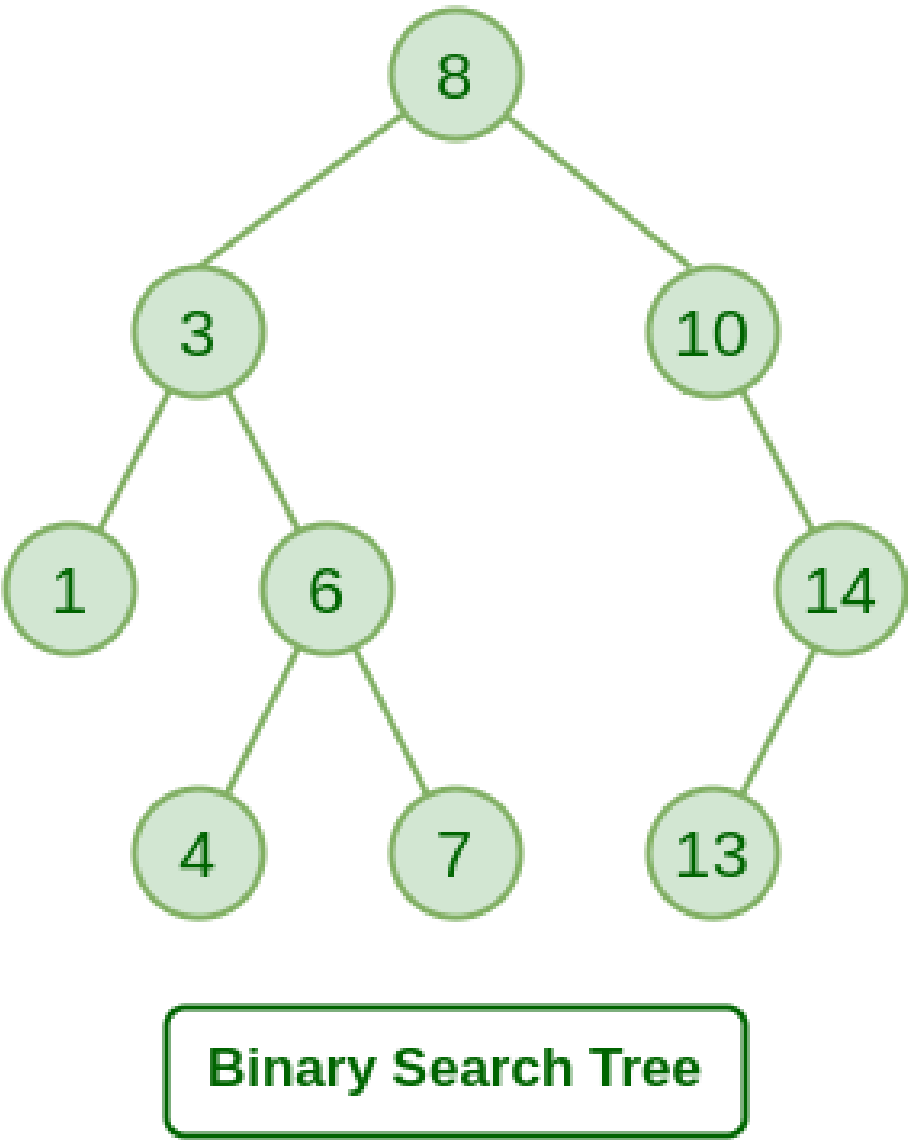
Se analizaron los métodos:

- Insert
- Find
- Remove

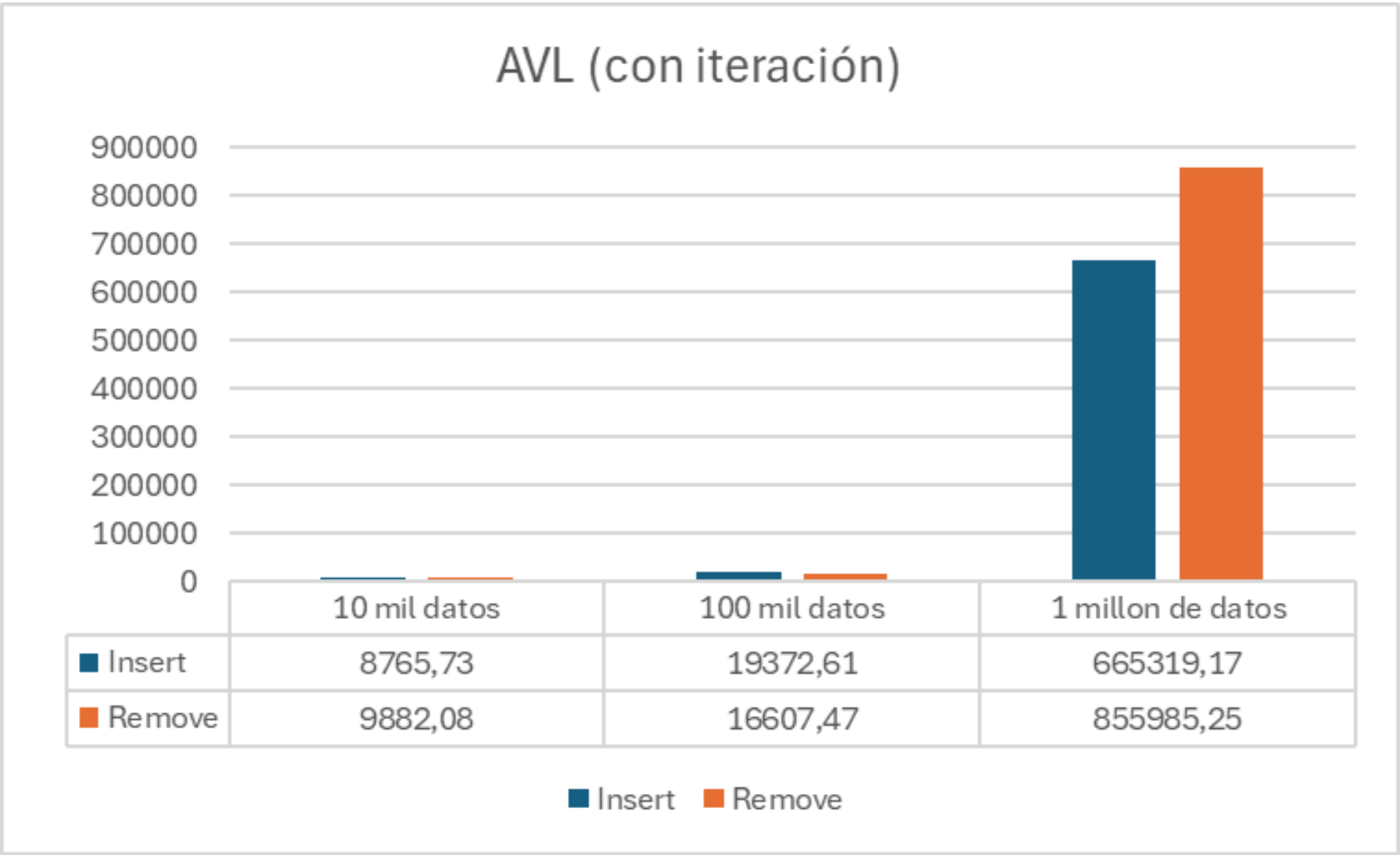
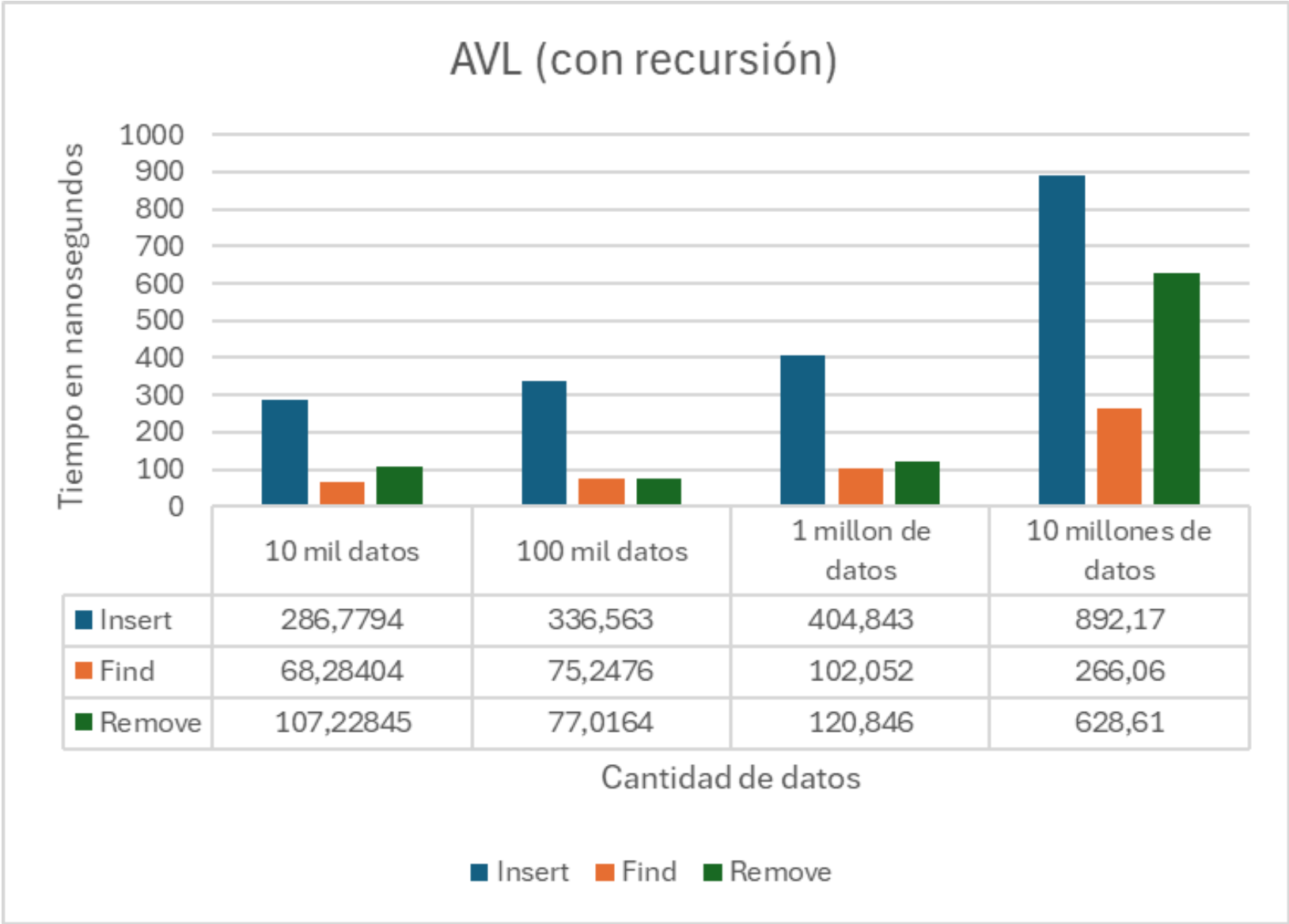
Se espera que el BST tenga un peor rendimiento con respecto al AVL

	AVL Tree		Binary Search Tree	
	Average	Worst	Average	Worst
Insertion	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
Deletion	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
Searching	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$

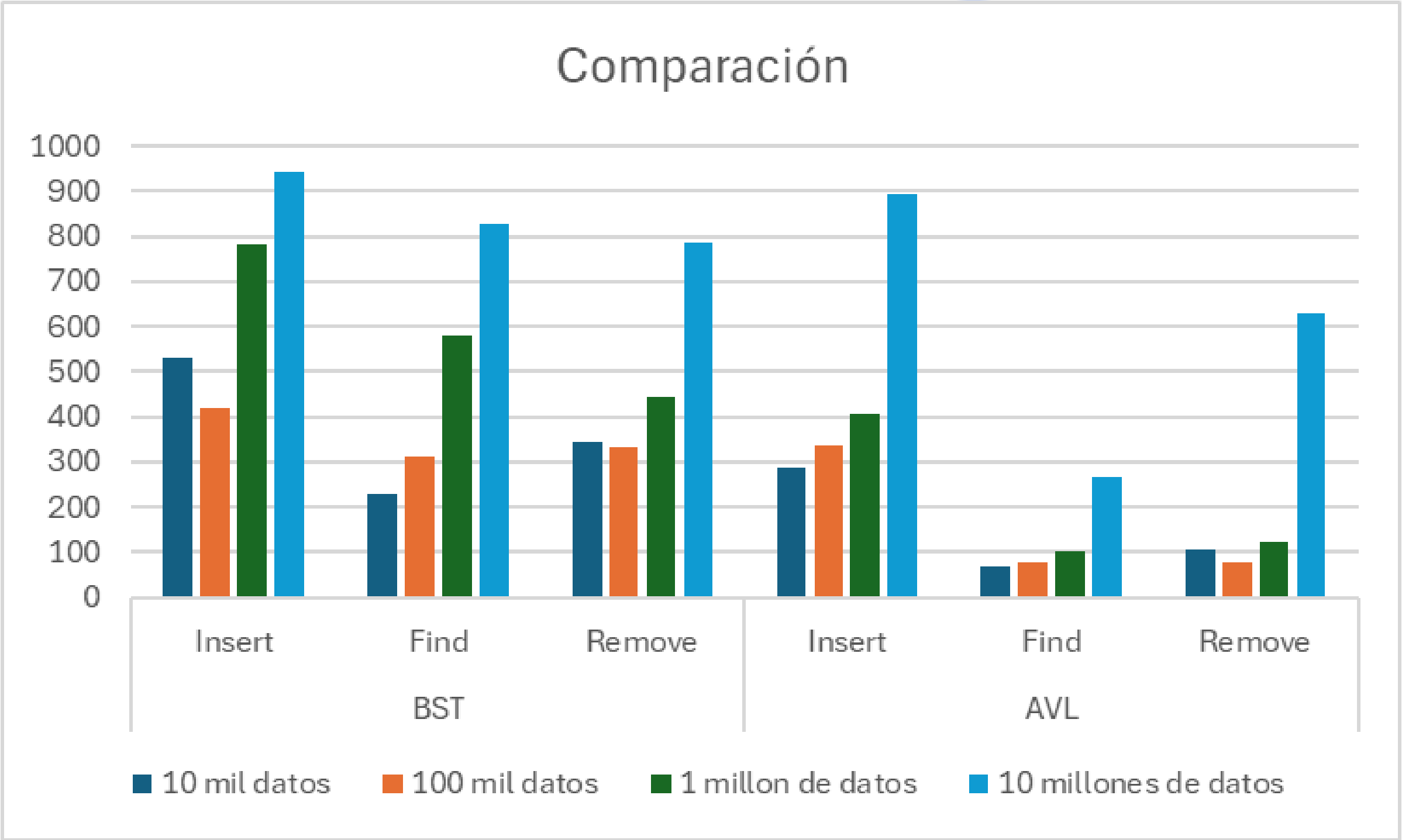
PRUEBAS Y RESULTADOS BST



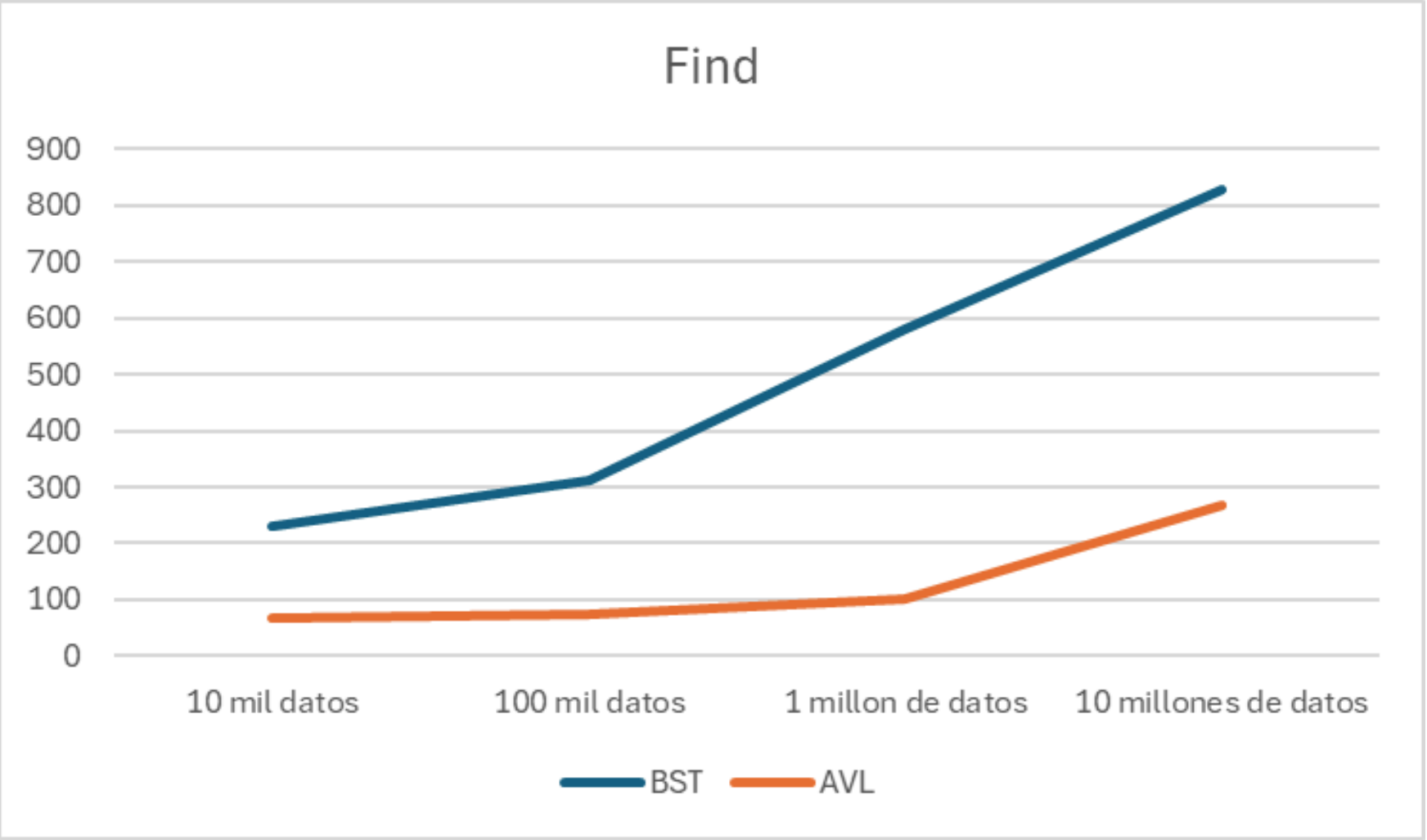
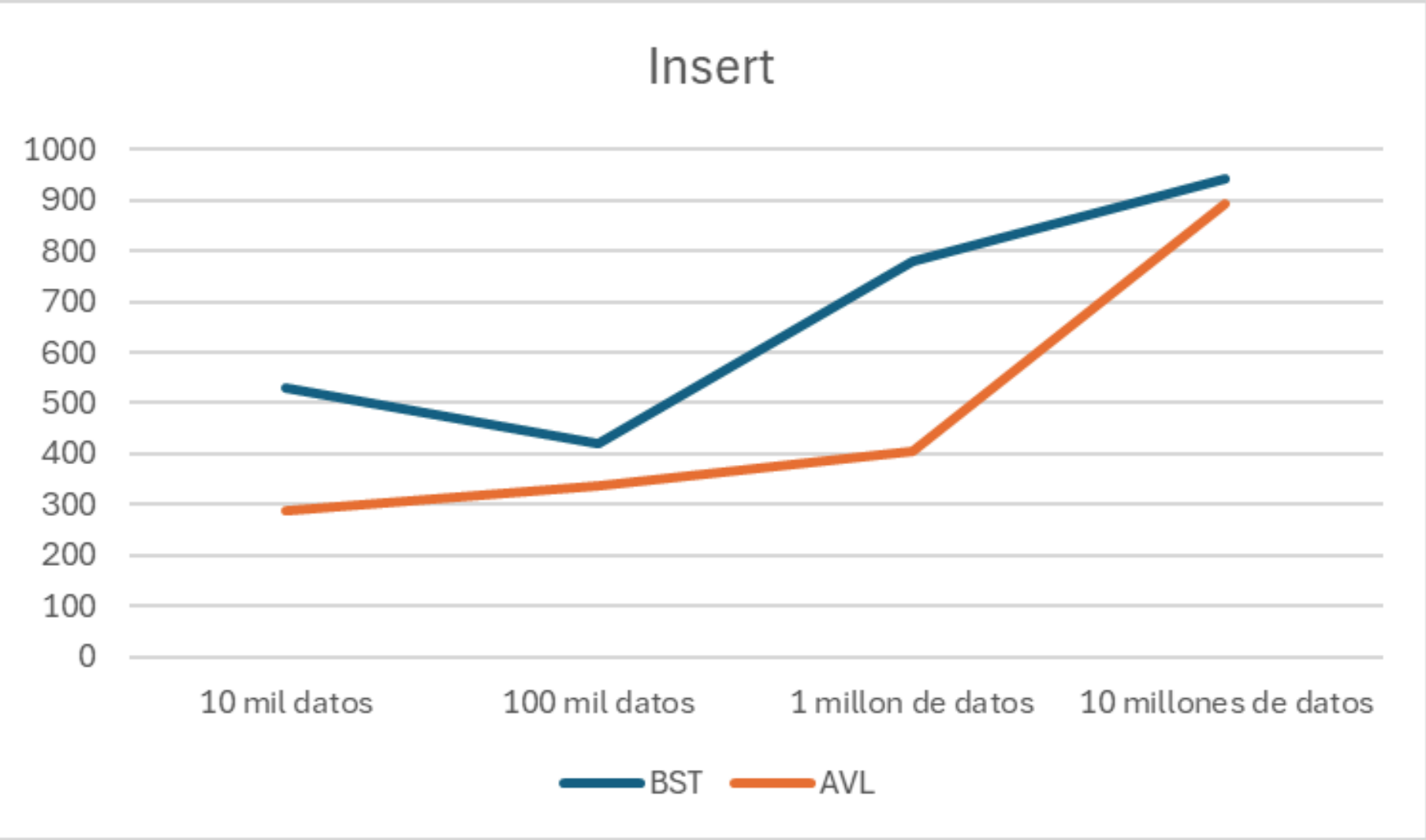
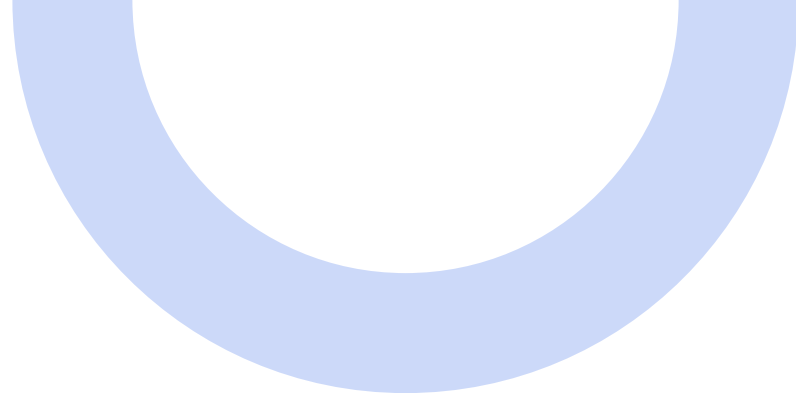
PRUEBAS Y RESULTADOS AVL



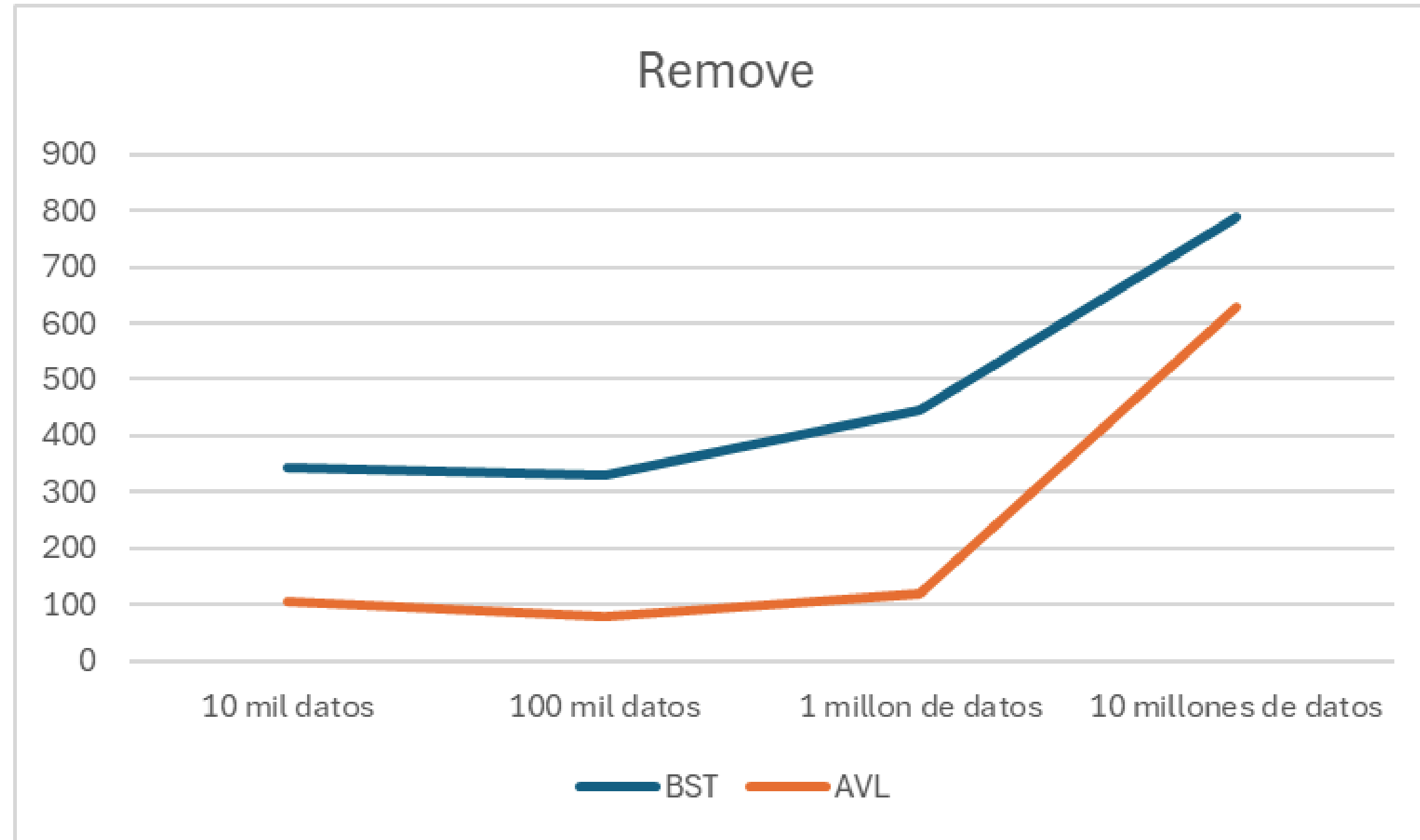
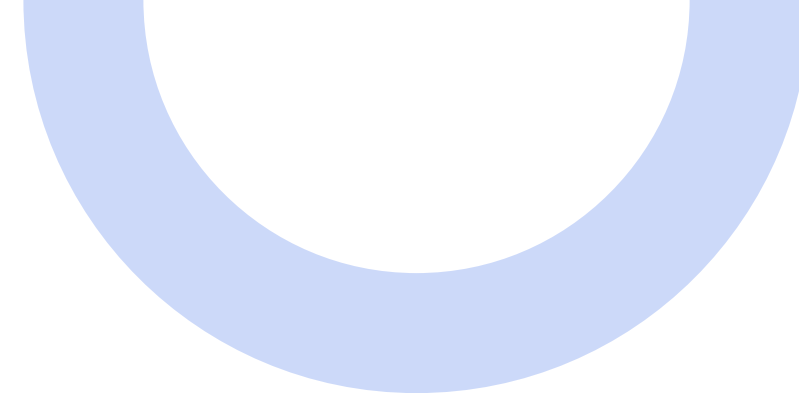
COMPARACIÓN



COMPARACIÓN



COMPARACIÓN



HEAP: MAXHEAP Y MINHEAP

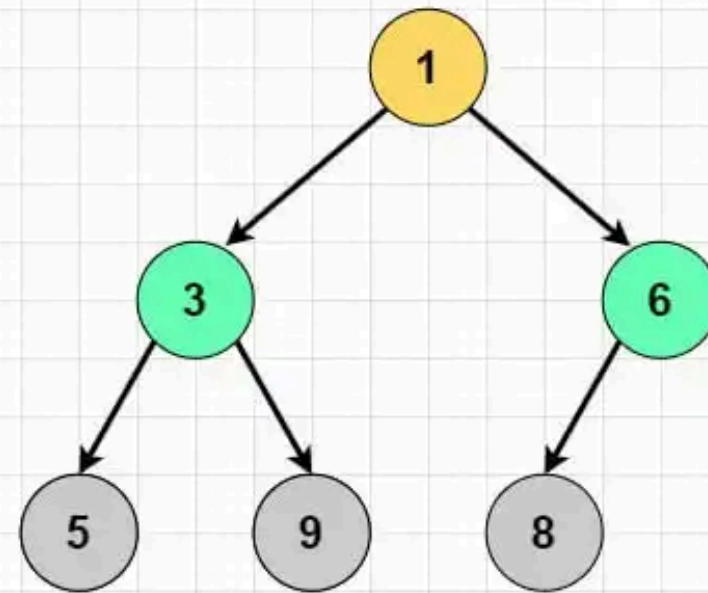


Se analizaron los métodos:

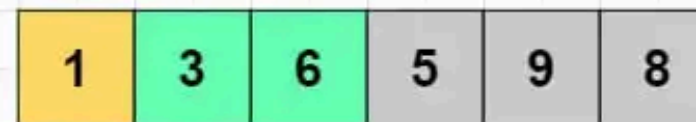
- Insert
- Remove
- ExtractMax - ExtractMin
- HeapSort

Binary Heap

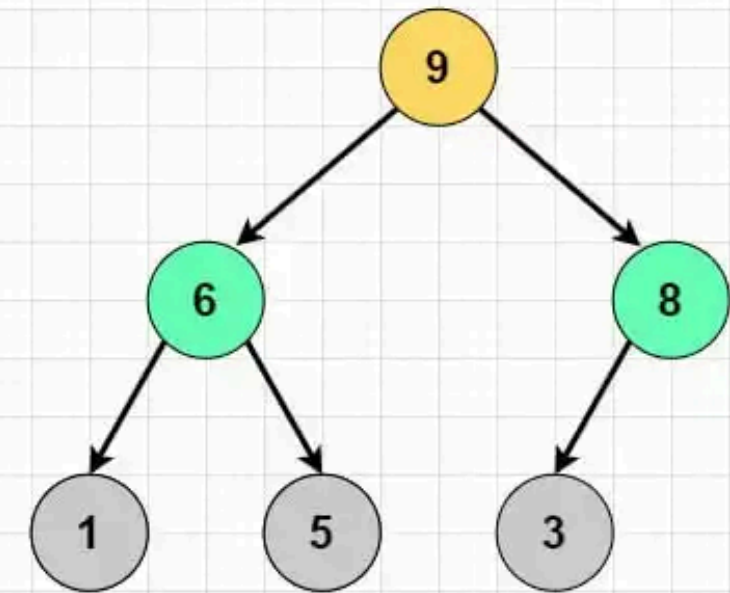
Min-Heap



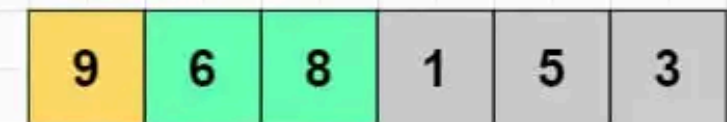
Array representation:



Max-Heap

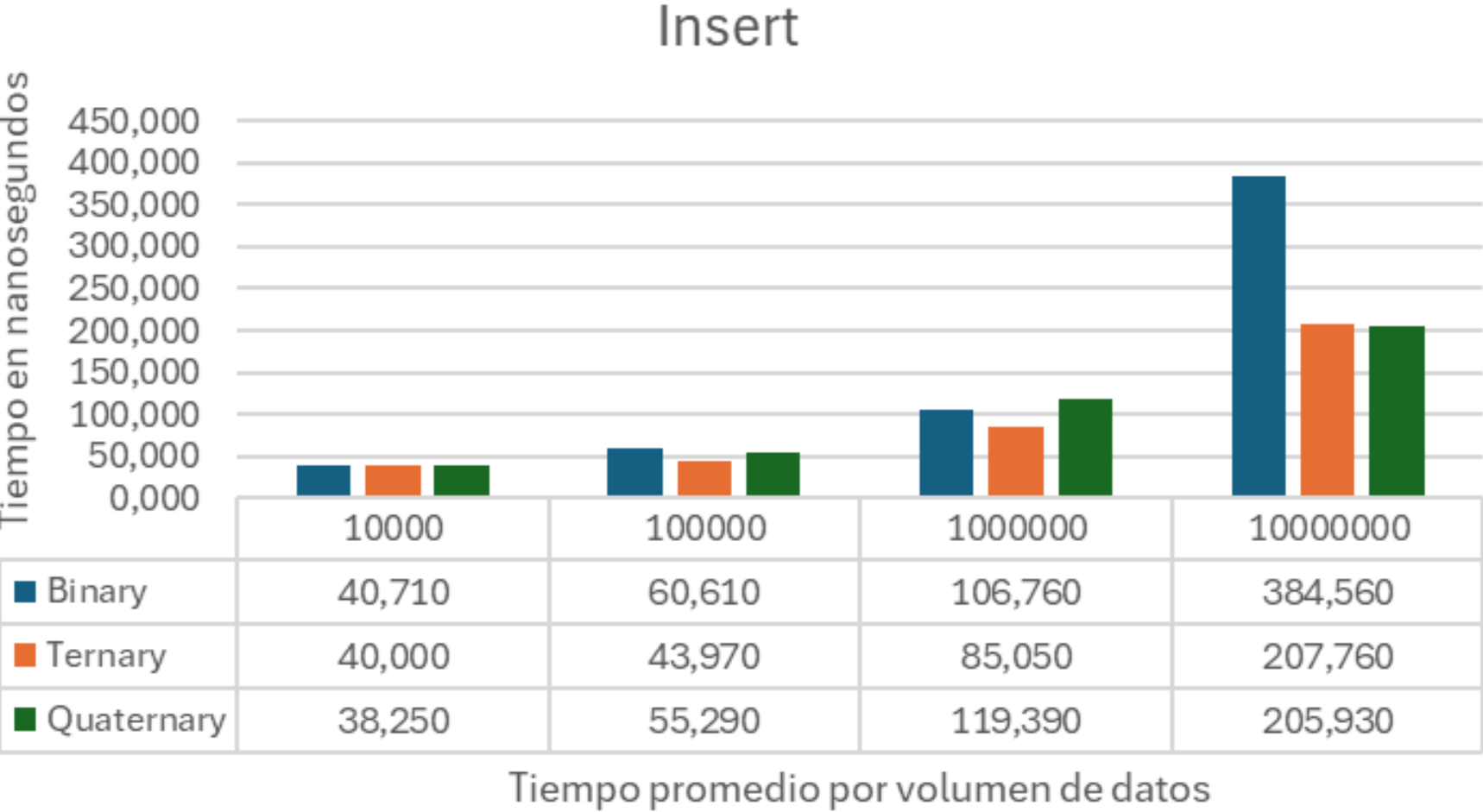


Array representation:

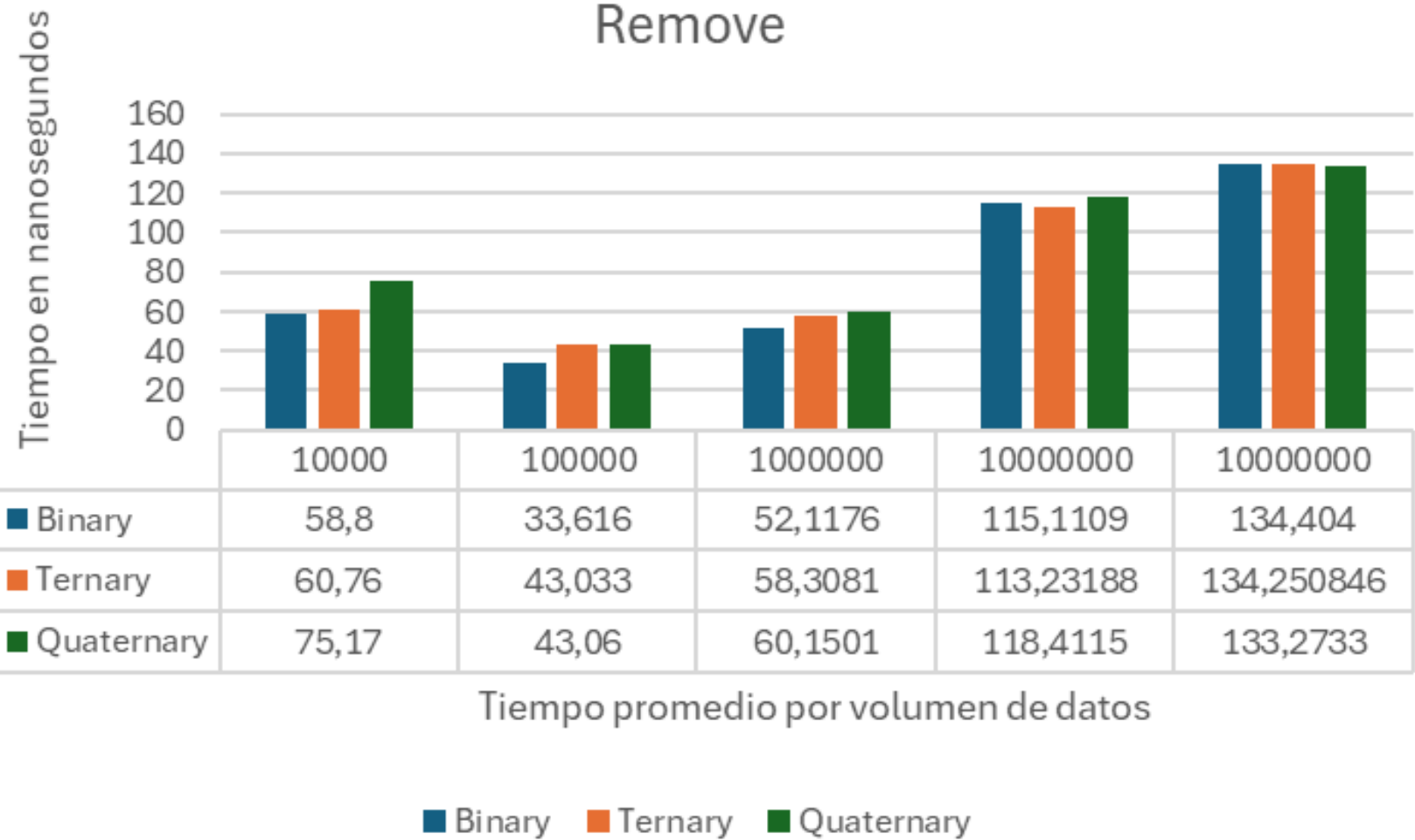


PRUEBAS Y RESULTADOS HEAP

INSERT

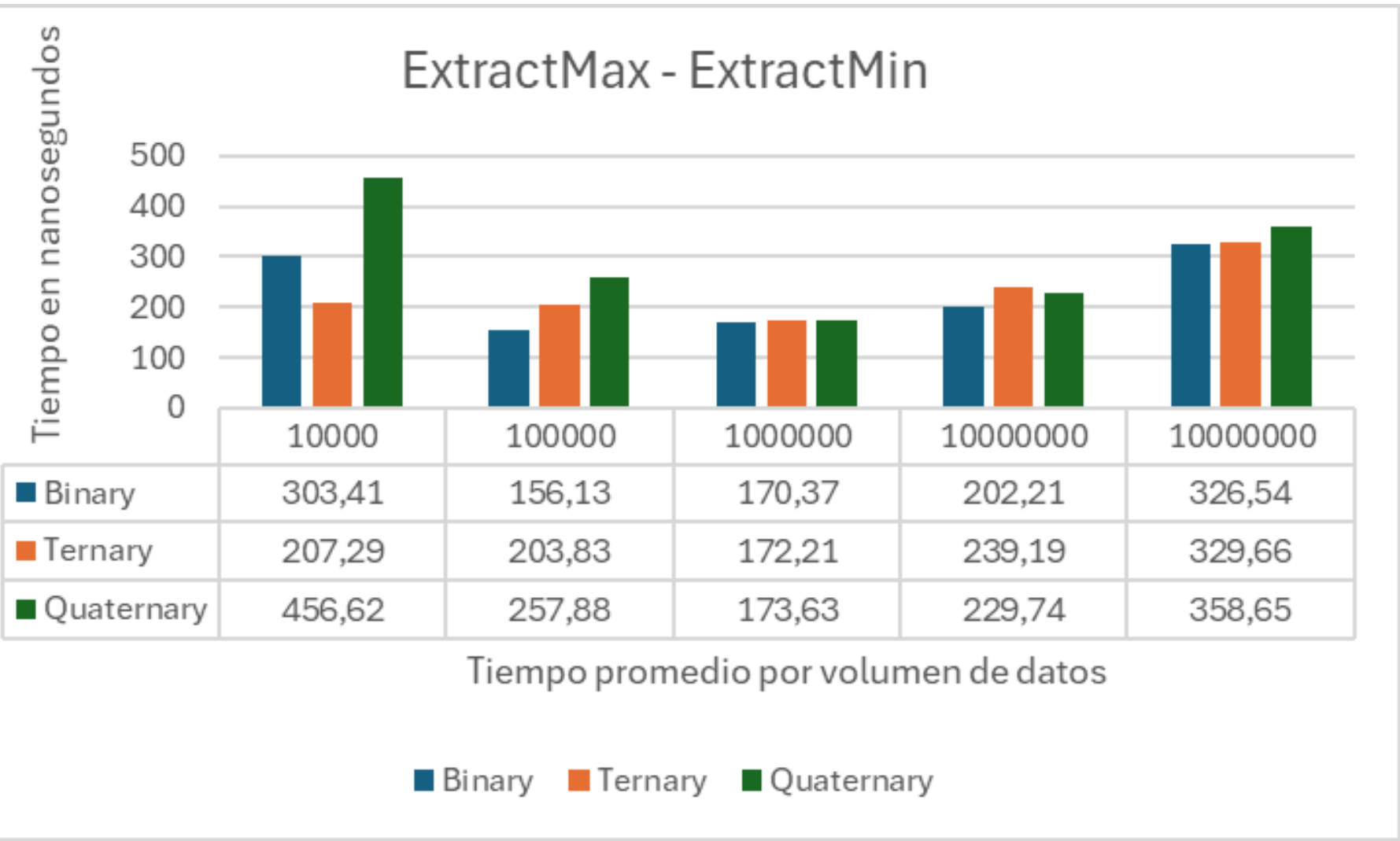


REMOVE

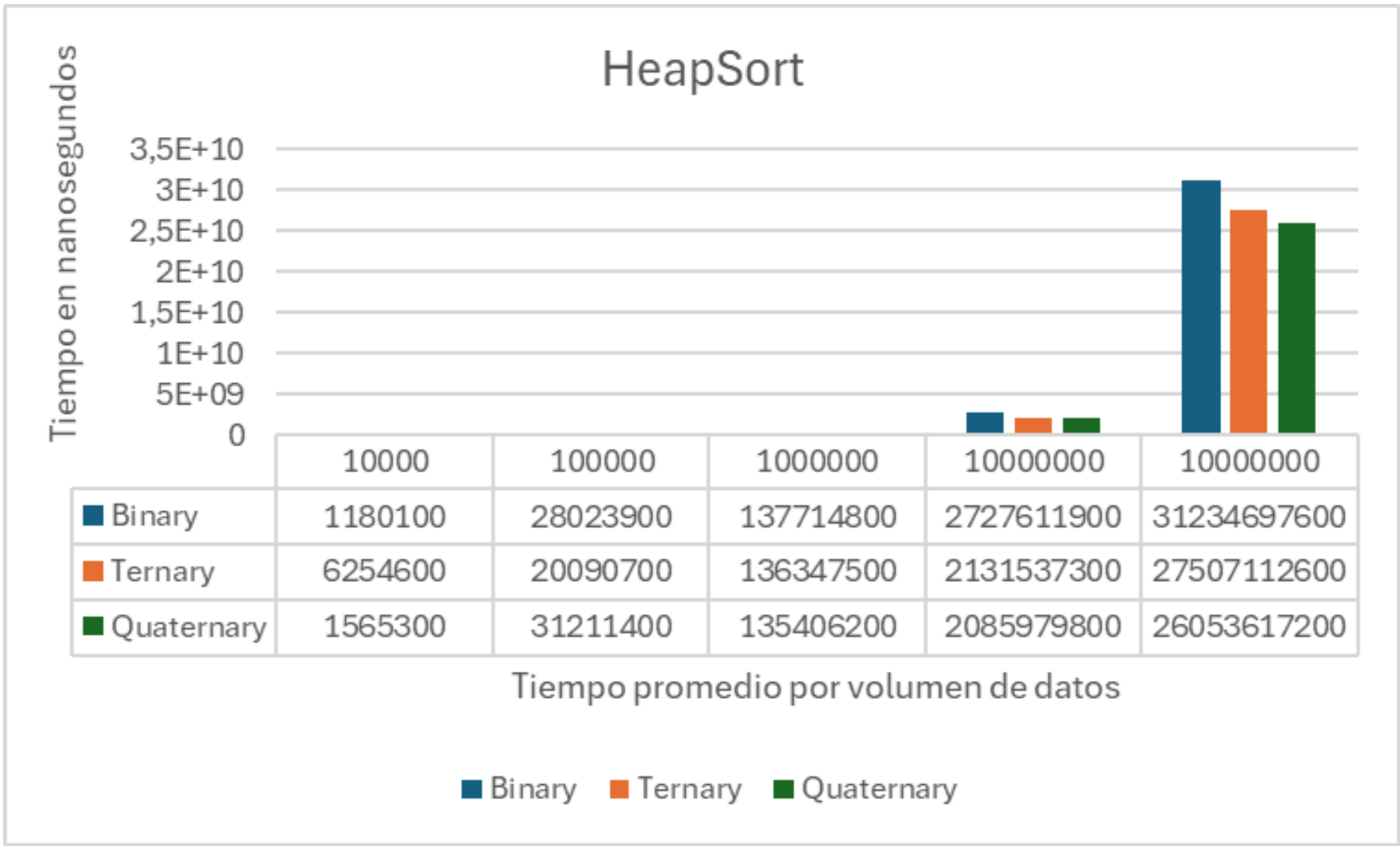


PRUEBAS Y RESULTADOS HEAP

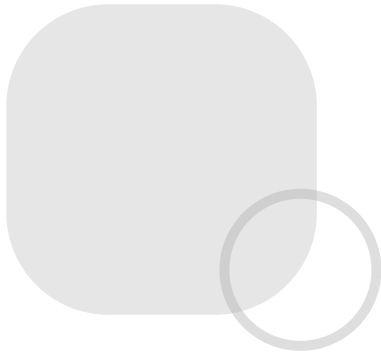
EXTRACTMAX - EXTRACTMIN



HEAPSORT



BIG(O)



	Insert	Remove	ExtractMax - ExtractMin	HeapSort
Binary	$O(\log_2 n)$	$O(2 \log_2 n)$	$O(2 \log_2 n)$	$O(2n \log_2 n)$
Ternary	$O(\log_3 n)$	$O(3 \log_3 n)$	$O(3 \log_3 n)$	$O(3n \log_3 n)$
Quaternary	$O(\log_4 n)$	$O(4 \log_4 n)$	$O(4 \log_4 n)$	$O(4n \log_4 n)$

$$\log_b(a) = \frac{\log_c(a)}{\log_c(b)}$$

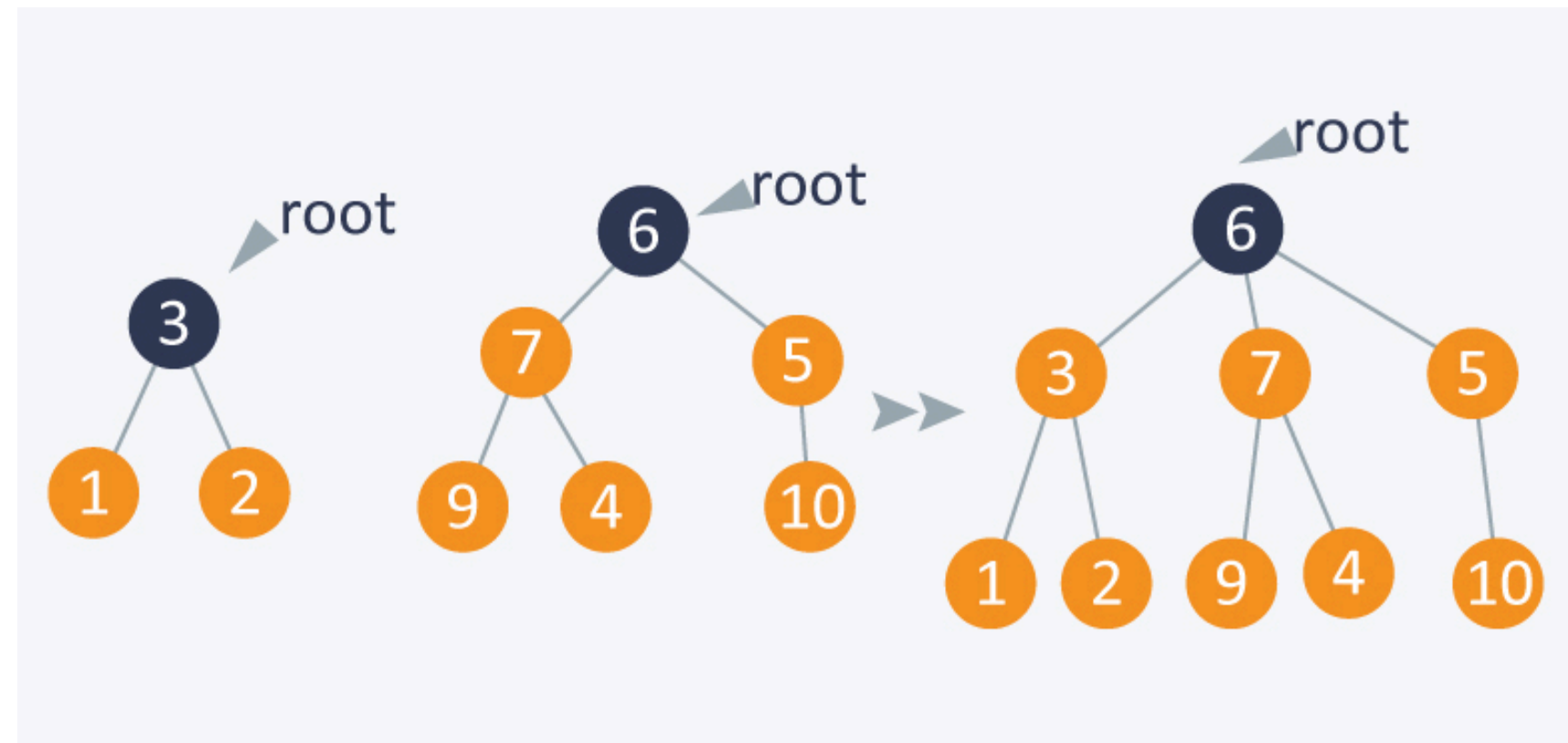
Funcionalidad	Big(O)
Insert	$O(\log n)$
Remove	$O(\log n)$
ExtractMax – ExtractMin	$O(\log n)$
HeapSort	$O(n \log n)$

CONJUNTOS DISJUNTOS



Se analizaron los métodos:

- Union
- Find
- Connected



Compresión de ruta y unión por rango

Resultados esperados:

Union, Find y Connected: Se espera que tengan una complejidad casi constante.

Comparativamente

Se espera que la implementación que tiene compresión de ruta y unión por rango tenga mejores resultados en términos de tiempo por operación

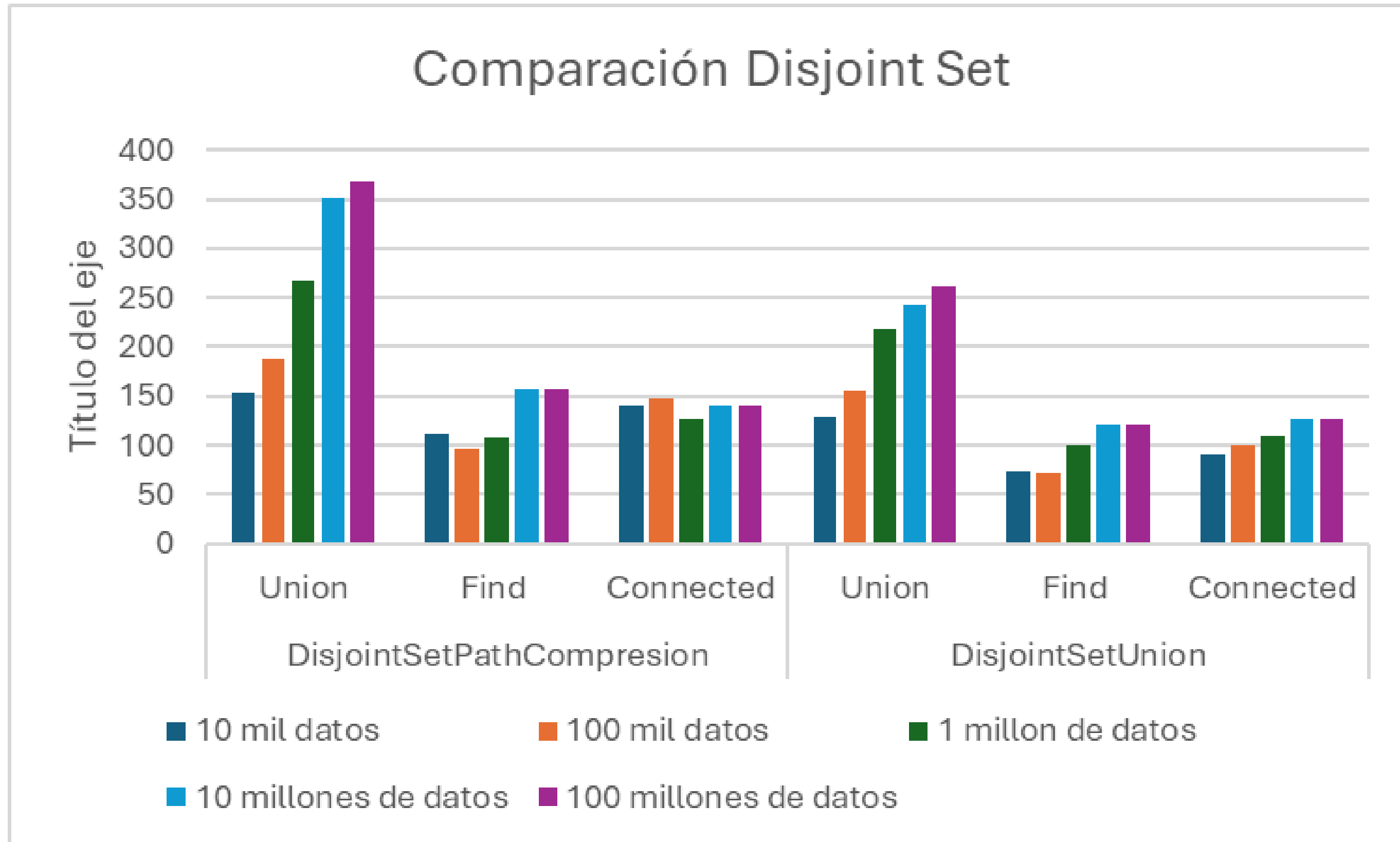
Compresión de ruta

Resultados esperados:

Union: En el peor de los casos puede ser $O(n)$ pues no está optimizado.

Find y Connected: Se espera que tenga una complejidad mayor a $O(1)$ y menor a $O(\log(n))$.

RESULTADOS



ANÁLISIS



Las hipotesis concuerdan con los resultados y el analisis, exceptuando la union con compresión de ruta, pues no se tomó en cuenta que disminuir el tamaño de las ramas tambien facilitaba la unión.

Se concluye que la mejor implementación en términos de tiempo es la union por rango y compresión de ruta.

$O(\alpha(n))$ Es la funcion de ackerman inversa, su resultado es inferior a 4 para practicamente cualquier numero.

$O(\log^*(n))$ es el logaritmo iterado o el número de veces que es necesario aplicar logaritmo para obtener un valor de uno, o menor.

Según el libro Algorithm Design de J. Kleinberg y E. Tardos, las compljidades son las siguientes

Método	Unión por rango y compresión de ruta	Compresión de ruta
Union	$O(\alpha(n))$	$O(\log^*(n))$
Find	$O(\alpha(n))$	$O(\log^*(n))$
Connected	$O(\alpha(n))$	$O(\log^*(n))$

$$O(1) < O(\alpha(n)) < O(\log^*(n)) < O(\log(n)) < O(n)$$

¿QUÉ IMPLEMENTAMOS?

¡HEAPS! (Montículos)

- Se han implementado montículos mínimos y máximos.
- En total se crearon seis clases.

© MaxHeapAlfabeticoEventos
© MaxHeapCostoEventos
© MaxHeapFechaEventos

© MinHeapAlfabeticoEventos
© MinHeapCostoEventos
© MinHeapFechaEventos

- Se crearon filtros de ordenamiento para los atributos costo, nombre y fecha.
- El método **heapSort**, esencial para un ordenamiento eficiente.

```
public MinHeapAlfabeticoEventos(DynamicUnsortedList<Evento> arr){  @cris...
    heap = arr;
    size = arr.size();

    for(int i = (arr.size() - 1) / 2; i > -1; i--){
        heapifyDown(i);
    }

    public DynamicUnsortedList<Evento> heapSort(){  @cris...

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

            swap(0, i);
            size--;
            heapifyDown(index: 0);
        }

        return heap;
    }
```


¿HAY DIFERENCIAS?



Los métodos *siftDown* y *siftUp* realizan comparaciones distintas en cada implementación.

MaxHeapAlfabeticoEventos

```
private void siftDown(int index) { 4 usages  👤 Diego *  
    int largest = index;  
    while (true) {  
        int leftChildIndex = 2 * largest + 1;  
        int rightChildIndex = 2 * largest + 2;  
  
        if (leftChildIndex < size &&  
            heap.get(leftChildIndex).getNombreEvento().compareToIgnoreCase(heap.get(largest).getNombreEvento()) > 0) {  
            largest = leftChildIndex;  
        }  
  
        if (rightChildIndex < size &&  
            heap.get(rightChildIndex).getNombreEvento().compareToIgnoreCase(heap.get(largest).getNombreEvento()) > 0) {  
            largest = rightChildIndex;  
        }  
  
        if (largest != index) {  
            swap(index, largest);  
            index = largest;  
        } else {  
            break;  
        }  
    }  
}
```

```
private void siftUp(int index) { 2 usages  👤 Diego +1 *  
    int parentIndex = (index - 1) / 2;  
    while (index > 0 && heap.get(index).getNombreEvento().compareToIgnoreCase(heap.get(parentIndex).getNombreEvento()) > 0) {  
        swap(index, parentIndex);  
        index = parentIndex;  
        parentIndex = (index - 1) / 2;  
    }  
}
```

MinHeapAlfabeticoEventos

```
private void siftUp(int index) { 2 usages  👤 Diego *  
    while (hasParent(index) &&  
        heap.get(index).getNombreEvento().compareToIgnoreCase(heap.get(getParentIndex(index)).getNombreEvento()) < 0) {  
        swap(index, getParentIndex(index));  
        index = getParentIndex(index);  
    }  
}
```

```
private void siftDown(int index) { 4 usages  👤 Diego +1 *  
    while (hasLeftChild(index)) {  
        int smallerChildIndex = getLeftChildIndex(index);  
        if (hasRightChild(index) &&  
            heap.get(getRightChildIndex(index)).getNombreEvento().compareToIgnoreCase(heap.get(smallerChildIndex).getNombreEvento()) < 0) {  
            smallerChildIndex = getRightChildIndex(index);  
        }  
  
        if (heap.get(index).getNombreEvento().compareToIgnoreCase(heap.get(smallerChildIndex).getNombreEvento()) < 0) {  
            break;  
        } else {  
            swap(index, smallerChildIndex);  
        }  
  
        index = smallerChildIndex;  
    }  
}
```

MaxHeapCostoEventos


```
private void siftDown(int index) { 4 usages  cristoferOrdonez *
    int largest = index;
    while (true) {
        int leftChildIndex = 2 * largest + 1;
        int rightChildIndex = 2 * largest + 2;


        if (leftChildIndex < size && heap.get(leftChildIndex).getCostoEvento() > heap.get(largest).getCostoEvento()) {
            largest = leftChildIndex;
        }

        if (rightChildIndex < size && heap.get(rightChildIndex).getCostoEvento() > heap.get(largest).getCostoEvento()) {
            largest = rightChildIndex;
        }

        if (largest != index) {
            swap(index, largest);
            index = largest;
        } else {
            break;
        }
    }
}
```

```
private void siftUp(int index) { 2 usages  cristoferOrdonez *
    int parentIndex = (index - 1) / 2;
    while (index > 0 && heap.get(index).getCostoEvento() > heap.get(parentIndex).getCostoEvento()) {
        swap(index, parentIndex);
        index = parentIndex;
        parentIndex = (index - 1) / 2;
    }
}
```

```
private void siftUp(int index) { 1 usage   cristoferOrdonez *
    while (hasParent(index) && heap.get(index).getCostoEvento() < heap.get(getParentIndex(index)).getCostoEvento()) {
        swap(index, getParentIndex(index));
        index = getParentIndex(index);
    }
}
```

```
private void siftDown(int index) { 4 usages   cristoferOrdonez *
    while (hasLeftChild(index)) {
        int smallerChildIndex = getLeftChildIndex(index);
        if (hasRightChild(index) && heap.get(getRightChildIndex(index)).getCostoEvento() < heap.get(smallerChildIndex).getCostoEvento()) {
            smallerChildIndex = getRightChildIndex(index);
        }

        if (heap.get(index).getCostoEvento() < heap.get(smallerChildIndex).getCostoEvento()) {
            break;
        } else {
            swap(index, smallerChildIndex);
        }

        index = smallerChildIndex;
    }
}
```

MinHeapCostoEventos

```
private void siftUp(int index) { 2 usages  👤 farid +1 *
    int parentIndex = (index - 1) / 2;
    while (index > 0 && heap.get(index).getFechaEvento().after(heap.get(parentIndex).getFechaEvento())) {
        swap(index, parentIndex);
        index = parentIndex;
        parentIndex = (index - 1) / 2;
    }
}
```

MaxHeapFechaEventos

```
private void siftDown(int index) { 4 usages  👤 farid +1 *
    int largest = index;
    while (true) {
        int leftChildIndex = 2 * largest + 1;
        int rightChildIndex = 2 * largest + 2;

        if (leftChildIndex < size && heap.get(leftChildIndex).getFechaEvento().after(heap.get(largest).getFechaEvento())) {
            largest = leftChildIndex;
        }

        if (rightChildIndex < size && heap.get(rightChildIndex).getFechaEvento().after(heap.get(largest).getFechaEvento())) {
            largest = rightChildIndex;
        }

        if (largest != index) {
            swap(index, largest);
            index = largest;
        } else {
            break;
        }
    }
}
```

```

private void siftUp(int index) { 2 usages  👤 farid +1 *
    int parentIndex = (index - 1) / 2;
    while (index > 0 && heap.get(index).getFechaEvento().before(heap.get(parentIndex).getFechaEvento())) {
        swap(index, parentIndex);
        index = parentIndex;
        parentIndex = (index - 1) / 2;
    }
}

```

```

private void siftDown(int index) { 4 usages  👤 farid +1 *
    int largest = index;
    while (true) {
        int leftChildIndex = 2 * largest + 1;
        int rightChildIndex = 2 * largest + 2;

        if (leftChildIndex < size && heap.get(leftChildIndex).getFechaEvento().before(heap.get(largest).getFechaEvento())) {
            largest = leftChildIndex;
        }

        if (rightChildIndex < size && heap.get(rightChildIndex).getFechaEvento().before(heap.get(largest).getFechaEvento())){
            largest = rightChildIndex;
        }

        if (largest != index) {
            swap(index, largest);
            index = largest;
        } else {
            break;
        }
    }
}

```

MinHeapFechaEventos

¡OBSERVAMOS LA APP EN ACCIÓN!

TIGO 35 % 3:58

Descubrir

Posible nombre del evento

Fecha

Localidad

\$ Costo mínimo 0 \$ Costo máximo

Tipo de evento

APLICAR FILTROS

TIGO 35 % 3:58

Ordenar Eventos

a - Presencial
2/9/2024 · 3:55a.m. - 8:55a.m.
Lugar: Cl. 12c #71C-30
Costo: \$ 50.000,00 COP
Tipo: Musica

b - Virtual
1/9/2024 · 3:56a.m. - 10:56a.m.
Plataforma: Teams
Costo: \$ 25.000,00 COP
Tipo: Talleres

c - Presencial
3/9/2024 · 3:57a.m. - 9:57a.m.
Lugar: Cra. 88c #54c 29 sur
Costo: \$ 10.000,00 COP

TIGO 35 % 3:58

Ordenar E

A_Z
Z_A
Fecha más próximos
Fecha menos próximos
Mayor costo
Menor costo

a - Presencial
2/9/2024 · 3:55a.m. - 8:55a.m.
Lugar: Cl. 12c #71C-30
Costo: \$ 50.000,00 COP
Tipo: Musica

b - Virtual
1/9/2024 · 3:56a.m. - 10:56a.m.
Plataforma: Teams
Costo: \$ 25.000,00 COP
Tipo: Talleres

c - Presencial
3/9/2024 · 3:57a.m. - 9:57a.m.
Lugar: Cra. 88c #54c 29 sur
Costo: \$ 10.000,00 COP

