
	<b>UNIVERSIDAD NACIONAL DE SAN AGUSTIN</b> <b>FACULTAD DE INGENIERÍA DE PRODUCCIÓN Y SERVICIOS</b> <b>ESCUELA PROFESIONAL DE INGENIERÍA DE SISTEMA</b>	
<b>Formato:</b> Guía de Práctica de Laboratorio / Talleres / Centros de Simulación		
<b>Aprobación:</b> 2022/03/01	<b>Código:</b> GUIA-PRLE-001	<b>Página:</b> 1

## INFORME DE TRABAJO PRÁCTICO

INFORMACIÓN BÁSICA					
<b>ASIGNATURA:</b>	<i>Estructura de Datos y Algoritmos</i>				
<b>TÍTULO DEL TRABAJO:</b>	<i>HEAPS</i>				
<b>NÚMERO DE TRABAJO:</b>	<i>N° 3</i>	<b>AÑO LECTIVO:</b>	<i>2023- A</i>	<b>NRO. SEMESTRE:</b>	<i>III</i>
<b>FECHA DE PRESENTACIÓN</b>	<i>17/06/2023</i>	<b>HORA DE PRESENTACIÓN</b>	<i>11:00 am</i>		
<b>INTEGRANTE (s) :</b> <ul style="list-style-type: none"> <li><i>Wilson Josue Turpo Huanca</i></li> <li><i>Fiorela Clariza Quispe Quispe</i></li> </ul>				<b>NOTA (0-20)</b>	
<b>DOCENTE(s):</b>  <i>KARIM GUEVARA PUENTE DE LA VEGA</i>					

## INTRODUCCIÓN

El propósito del trabajo es: aprender heaps en Java ya que este te proporcionará una herramienta poderosa para manejar elementos con prioridad y mejorar la eficiencia de tus algoritmos y estructuras de datos. Además, te brindará una base sólida para explorar y comprender otros conceptos y estructuras de datos más complejas.

## METODOLOGÍA



Para resolver este ejercicio se utilizó la clase `PriorityQueueHeap`: es una implementación específica de cola de prioridad utilizando el heap anteriormente mencionado. Permittiéndonos agregar elementos a la cola de prioridad mediante el método `enqueue`, donde se especifica el elemento y su prioridad.

Los elementos se insertan en el heap basándose en su prioridad. También proporciona métodos para eliminar el elemento de mayor prioridad (`dequeue`), obtener el elemento de mayor prioridad sin eliminarlo (`front`) y verificar si la cola de prioridad está vacía (`isEmpty`).

La clase `PriorityQueueNode` es una clase auxiliar utilizada por `PriorityQueueHeap` para representar un nodo de la cola de prioridad. Cada nodo contiene un elemento y su prioridad. La implementación de `Comparable` en `PriorityQueueNode` se utiliza para comparar nodos según su prioridad al insertar y eliminar elementos del heap.

En el `main` de la clase `TestHeaps`, se crea una instancia de `PriorityQueueHeap` y se agregan varios elementos a la cola de prioridad con diferentes prioridades. Luego se eliminan y muestran los elementos en orden de prioridad utilizando `dequeue`. Esto demuestra el funcionamiento de la cola de prioridad basada en el heap máximo.

## SOLUCIONE Y PRUEBAS

	<p>UNIVERSIDAD NACIONAL DE SAN AGUSTIN FACULTAD DE INGENIERÍA DE PRODUCCIÓN Y SERVICIOS ESCUELA PROFESIONAL DE INGENIERÍA DE SISTEMA</p>	
<p>Formato: Guía de Práctica de Laboratorio / Talleres / Centros de Simulación</p>		
<p>Aprobación: 2022/03/01</p>	<p>Código: GUIA-PRLE-001</p>	<p>Página: 1</p>

## CONSIDERACIONES PARA LA ENTREGA

- Además, deben de subir al aula virtual el archivo con el código realizado.
- El trabajo será desarrollado en parejas, durante las horas de práctica en aula.
- Las soluciones de los ejercicios deberán ser subidos a un repositorio en Github, que deben compartirlo con el profesor. Límite de plazo para cualquier actualización que se realice hasta las 12.00 del mediodía del sábado 17/06/2023.
- Además, un integrante del grupo debe subir al aula virtual el archivo con el código realizado, y el enlace del repositorio de trabajo. Límite de plazo hasta el término de la sesión del viernes 16 de junio de 2023.

## EJERCICIO 5: Construya una cola de prioridad que utilice un heap como estructura de datos. Para esto realice lo siguiente:

- Implemente el TAD Heap genérico que esté almacenado sobre un ArrayList con las operaciones de inserción y eliminación. Este TAD debe de ser un heap máximo.
- Implemente la clase PriorityQueueHeap genérica que utiliza como estructura de datos el heap desarrollado en el punto anterior. Esta clase debe tener las operaciones de una cola tales como:
  - a. Enqueue (x, p) : inserta un elemento a la cola 'x' de prioridad 'p' a la cola. Como la cola esta sobre un heap, este deberá ser insertado en el heap-max y reubicado de acuerdo a su prioridad.
  - b. Dequeue() : elimina el elemento de la mayor prioridad y lo devuelve. Nuevamente como la cola está sobre un heap-max, el elemento que debe ser eliminado es la raíz, por tanto, deberá sustituir este elemento por algún otro de modo que se cumpla las propiedades del heap-max.
  - c. Front() : sólo devuelve el elemento de mayor prioridad.
  - d. Back(): sólo devuelve el elemento de menor prioridad.

NOTA: tenga cuidado en no romper el encapsulamiento en el acceso a los atributos de las clases correspondientes.

REPOSITORIO GITHUB: <https://github.com/FiorelaClarz/PRACTICA-CLASE-16-6-23/tree/main/src>

REPOSITORIO GITHUB colaboracion: <https://github.com/wilsonjosue/TEO-EDA-C-03/tree/master>

## Implementación de la clase Heap

- El constructor Heap() crea un objeto Heap vacío inicializando el ArrayList heap.
- El método insert(T ítem) se utiliza para insertar un elemento en heap.
- Agrega el elemento al final del ArrayList y luego utiliza el método ajustarEstructura() para mantener en la estructura al Heap hacia.
- El método remove() se utiliza para eliminar y devolver el elemento raíz del montículo. Verifica si el montículo está vacío y, si no lo está, extrae el elemento raíz, lo reemplaza por el último elemento del ArrayList.

```
3  import java.util.*;
4  /**
5   * La clase Heap<T> es una implementación de un montículo binario utilizando un ArrayList
6   * para almacenar los elementos. El tipo genérico T se limita a aquellos tipos que
7   * implementan la interfaz Comparable<T>, lo que significa que los elementos pueden ser
8   * comparados entre sí.
9   *
10 * */
11 class Heap<T extends Comparable<T>> {
12     private List<T> heap;
13
14     public Heap() {
15         heap = new ArrayList<>();
16     }
17
18     public void insert(T item) {
19         heap.add(item);
20         ajustarEstructura(heap.size() - 1);
21     }
22
23     public T remove() {
24         if (isEmpty()) {
25             throw new IllegalStateException("Heap is empty");
26         }
27         T root = heap.get(0);
28         T lastItem = heap.remove(heap.size() - 1);
29         if (!isEmpty()) {
30             heap.set(0, lastItem);
31             verificarEstructura(0);
32         }
33         return root;
34     }
35 }
```

- El método `devolverElemento()` devuelve el elemento raíz del montículo sin eliminarlo. Si el montículo está vacío, se lanza una excepción.
- El método `isEmpty()` verifica si el montículo está vacío y devuelve un valor booleano en consecuencia.
- Los métodos privados `ajustarEstructura(int index)` y `verificarEstructura(int index)` se utilizan para reajustar el montículo hacia arriba o hacia abajo, respectivamente, después de insertar o eliminar un elemento. Estos métodos aseguran que las propiedades del montículo se mantengan

```
36 public T devolverElemento() {
37     if (isEmpty()) {
38         throw new IllegalStateException(s:"Heap is empty");
39     }
40     return heap.get(index:0);
41 }
42
43 public boolean isEmpty() {
44     return heap.isEmpty();
45 }
46
47 private void ajustarEstructura(int index) {
48     int parentIndex = (index - 1) / 2;
49     while (index > 0 && heap.get(index).compareTo(heap.get(parentIndex)) > 0) {
50         intercambio(index, parentIndex);
51         index = parentIndex;
52         parentIndex = (index - 1) / 2;
53     }
54 }
55
56 private void verificarEsctructura(int index) {
57     int leftChildIndex = 2 * index + 1;
58     int rightChildIndex = 2 * index + 2;
59     int largestIndex = index;
60
61     if (leftChildIndex < heap.size() && heap.get(leftChildIndex).compareTo(heap.get(largestIndex)) > 0) {
62         largestIndex = leftChildIndex;
63     }
64     if (rightChildIndex < heap.size() && heap.get(rightChildIndex).compareTo(heap.get(largestIndex)) > 0) {
65         largestIndex = rightChildIndex;
66     }
67     if (largestIndex != index) {
68         intercambio(index, largestIndex);
69         verificarEsctructura(largestIndex);
70     }
71 }
```

- Si largestIndex no es igual a index, significa que el elemento en index es menor (o mayor) que al menos uno de sus hijos. En ese caso, se realiza un intercambio entre el elemento en index y el hijo correspondiente (en la posición largestIndex).
- Después del intercambio, el método se llama recursivamente a sí mismo con largestIndex como el nuevo índice, para seguir verificando y ajustando la posición del elemento en el subárbol descendiente.
- El método mostrarDatosInsertados() nos muestra el como quedo el árbol binario

```
73     private void intercambio(int index1, int index2) {
74         T temp = heap.get(index1);
75         heap.set(index1, heap.get(index2));
76         heap.set(index2, temp);
77     }
78
79     public void mostrarDatosInsertados(Heap<?> heap) {
80         System.out.println(x:"Datos insertados en el Heap:");
81
82         while (!heap.isEmpty()) {
83             System.out.print "["+heap.remove()+"]");
84         }
85         System.out.println();
86     }
87
88 }
```

## Implementación de la clase PriorityQueueHeap

- La clase PriorityQueueHeap<T> implementa una cola de prioridad utilizando un Heap como estructura de datos subyacente. El tipo genérico T se limita a aquellos tipos que implementan la interfaz Comparable<T>.
- El constructor PriorityQueueHeap() crea un objeto PriorityQueueHeap inicializando el Heap vacío.

```
3 public class PriorityQueueHeap<T extends Comparable<T>> {  
4  
5     private Heap<PriorityQueueNode<T>> heap;  
6  
7     public PriorityQueueHeap() {  
8         heap = new Heap<>();  
9     }
```

- El método enqueue(T item, int priority) se utiliza para insertar un elemento con una prioridad en la cola de prioridad. Crea un nuevo nodo PriorityQueueNode<T> con el elemento y la prioridad especificados y lo inserta en el montículo utilizando el método insert() del Heap.

```
11 public void enqueue(T item, int priority) {  
12     PriorityQueueNode<T> node = new PriorityQueueNode<>(item, priority);  
13     heap.insert(node);  
14 }  
15
```

- El método dequeue() se utiliza para eliminar y devolver el elemento de mayor prioridad de la cola de prioridad. Utiliza el método remove() del montículo (Heap) para extraer el nodo de mayor prioridad y luego devuelve el elemento almacenado en ese nodo.

```
16 public T dequeue() {  
17     PriorityQueueNode<T> node = heap.remove();  
18     return node.getItem();  
19 }  
20
```

- El método `front()` se utiliza para obtener el elemento de mayor prioridad en la cola de prioridad sin eliminarlo. Utiliza el método `devolverElemento()` del Heap para obtener el nodo de mayor prioridad y luego devuelve el elemento almacenado en ese nodo.

```
21     public T front() {  
22         PriorityQueueNode<T> node = heap.devolverElemento();  
23         return node.getItem();  
24     }  
25
```

- El método `back()` se utiliza para obtener el elemento de menor prioridad en la cola de prioridad sin eliminarlo. Utiliza el método `devolverElemento()` del Heap para obtener el nodo de mayor prioridad (ya que es un montículo máximo) y luego devuelve el elemento almacenado en ese nodo.

```
26     public T back() {  
27         return heap.devolverElemento().getItem();  
28     }  
29
```

- El método `isEmpty()` verifica si la cola de prioridad está vacía y devuelve un valor booleano. Utiliza el método `isEmpty()` del Heap.

```
29  
30     public boolean isEmpty() {  
31         return heap.isEmpty();  
32     }
```

- El método `mostrarPrioritarios()` sólo nos muestra los datos prioritarios que fueron solicitados en el main

```
32     }  
33     public void mostrarPrioritarios(PriorityQueueHeap<?> priQueue) {  
34         System.out.println(x:"Datos insertados en el Heap:");  
35  
36         while (!priQueue.isEmpty()) {  
37             System.out.println(priQueue.dequeue());  
38         }  
39         System.out.println();  
40     }  
41 }
```



## Implementación de la clase PriorityQueueNode

- La línea 3 se declara la clase PriorityQueueNode como una clase genérica que acepta un tipo T que debe implementar la interfaz Comparable<T>. Esto significa que los elementos de la cola de prioridad deben ser comparables entre sí.
- Los campos private T item y private int priority representan el elemento y la prioridad del nodo respectivamente.
- El constructor public PriorityQueueNode(T item, int priority) se utiliza para crear un nuevo nodo de la cola de prioridad con un elemento y una prioridad especificados.

```
3 public class PriorityQueueNode<T extends Comparable<T>> implements Comparable<PriorityQueueNode<T>> {  
4  
5     private T item;  
6     private int priority;  
7  
8     public PriorityQueueNode(T item, int priority) {  
9         this.item = item;  
10        this.priority = priority;  
11    }  
}
```

- El método public T getItem() devuelve el elemento almacenado en el nodo.
- El método public int getPriority() devuelve la prioridad del nodo.

```
13 public T getItem() {  
14     return item;  
15 }  
16  
17 public int getPriority() {  
18     return priority;  
19 }  
20
```

- CompareTo se implementa de acuerdo con la interfaz Comparable<PriorityQueueNode<T>>. Compara el nodo actual con otro nodo (o) en función de sus prioridades. Utiliza Integer.compare() para comparar las prioridades y devuelve el resultado de la comparación. Permitiendo ordenar los nodos en una cola de prioridad según sus prioridades.

```
20  
21 @Override  
22 public int compareTo(PriorityQueueNode<T> o) {  
23     return Integer.compare(this.priority, o.priority);  
24 }  
25 }
```

## Implementación TestHeaps del main()

- Insertamos los datos de prueba y lo ordenamos de acuerdo a la mayor prioridad donde 7 es el mayor grado de prioridad en este ejemplo, por tanto el primero en ser atendido es el número 9, seguido por 4, 6 y 7.

```
Run | Debug
4 public static void main(String[] args) {
5     Heap<Integer> monticulo = new Heap<>();
6
7     PriorityQueueHeap<Integer> heapPrioritario = new PriorityQueueHeap<>();
8     // Agregar elementos a la cola de prioridad
9     heapPrioritario.enqueue(item:4,priority:6);
10    heapPrioritario.enqueue(item:6,priority:4);
11    heapPrioritario.enqueue(item:7,priority:3);
12    heapPrioritario.enqueue(item:9,priority:7);
13
14    // Obtener y mostrar el elemento de mayor prioridad (máximo)
15    System.out.println("Elemento de mayor prioridad: " + heapPrioritario.front()); // Salida: 2
16
17    // Eliminar y mostrar los elementos en orden de prioridad (máximo a mínimo)
18    System.out.println(x:"Eliminando elementos:");
19    heapPrioritario.mostrarPrioritarios(heapPrioritario);
20    // Salida:
21    // 9
22    // 4
23    // 6
24    // 7
25 }
26 }
```

## Resultados

```
Output - EDA-TEO-10 (run) ×
run:
Elemento de mayor prioridad: 9
Eliminando elementos:
Datos insertados en el Heap:
9
4
6
7

BUILD SUCCESSFUL (total time: 0 seconds)
```

Para los datos ingresamos (4,6),(6,4),(7,3),(9,7),(10,9),(2,2),(1,1)

```
12
13     heapPrioritario.enqueue(4, 6);
14     heapPrioritario.enqueue(6, 4);
15     heapPrioritario.enqueue(7, 3);
16     heapPrioritario.enqueue(9, 7);
17     heapPrioritario.enqueue(10, 9);
18     heapPrioritario.enqueue(2, 2);
19     heapPrioritario.enqueue(1, 1);
```

```
Output - EDA-TEO-10 (run) ×
run:
Elemento de mayor prioridad: 10
Eliminando elementos:
Datos insertados en el Heap:
10
9
4
6
7
2
1

BUILD SUCCESSFUL (total time: 0 seconds)
```

## LECCIONES APRENDIDAS Y CONCLUSIONES

- Un heap es una estructura que a primera vista nos da información de la raíz, sin embargo, muy poca información del resto de nodos. Nosotros usamos el heap cuando queremos interactuar siempre con el más pequeño o en algunos casos con el más grande de los elementos. De tal forma, cuando queremos eliminar, buscar o leer un elemento que se encuentra en el medio del heap no tiene mucho sentido, debido a que solamente podemos leer, buscar o eliminar el elemento raíz del heap. Esta característica es de suma importancia, ya que es la condición que nos permite tener la operación búsqueda de complejidad algorítmica  $O(1)$ . Es decir, tenemos acceso directo a la raíz, en todo momento; por lo tanto, leer su contenido es una operación con complejidad  $O(1)$ , la cual es una gran ventaja en comparación con la búsqueda de otros elementos y otras búsquedas en lista.

## REFERENCIAS Y BIBLIOGRAFÍA

**Colocare las referencias utilizadas para el desarrollo del trabajo en formato IEEE**

- [1] *Implementación de Min Heap y Max Heap en Java*  
<https://www.techiedelight.com/es/min-heap-max-heap-implementation-in-java/>
- [2] *Implementar min-heap en Java*  
<https://www.delftstack.com/es/howto/java/min-heap-java/>
- [3] *Implementing a Heap in Java - Part 2*  
<https://www.youtube.com/watch?v=W81Qzuz4qH0>
- [4] *Priority Queue o Cola de Prioridad en Java*  
<https://www.youtube.com/watch?v=uUpPQrHsr54>
- [5] *Heaps*  
[https://aulavirtual.unsa.edu.pe/2023A/pluginfile.php/295836/mod\\_page/content/7/El%20Monti%CC%81culo%20Binario%20%20%20%20UPV.mp4](https://aulavirtual.unsa.edu.pe/2023A/pluginfile.php/295836/mod_page/content/7/El%20Monti%CC%81culo%20Binario%20%20%20%20UPV.mp4)
- [6] *Heaps*  
<https://drive.google.com/file/d/1Ma9WERoSytYdCbQc9S3XOsiUbim1XXQq/view>
- [7] **W. Turpo, F. Quispe , Heap -PRACTICA No. 3:**  
<https://github.com/FiorelaClarz/PRACTICA-CLASE-16-6-23/tree/main/src>