


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

PRÁCTICA N° 3

HEAPS

INFORMACIÓN BÁSICA					
ASIGNATURA:	ESTRUCTURA DE DATOS Y ALGORITMOS				
TÍTULO DEL TRABAJO:	Heaps				
NÚMERO DE TRABAJO:	03	AÑO LECTIVO:	2023-A	NRO. SEMESTRE:	III
FECHA DE PRESENTACIÓN	17/06/2023	HORA DE PRESENTACIÓN			
INTEGRANTE (s) Hidalgo Chinchay, Paulo Andre Chino Pari Joel Antonio				NOTA (0-20)	
DOCENTE(s): Dra. Karim Guevara Puente de la Vega					

INTRODUCCIÓN
<p>En este informe, se presenta la implementación del TAD (Tipo Abstracto de Datos) Heap genérico, utilizando una estructura de datos ArrayList. El objetivo es crear un Heap máximo, que permita realizar operaciones de inserción y eliminación de elementos. Además, se implementa la clase PriorityQueueHeap, que utiliza el Heap desarrollado previamente para simular una cola de prioridad. Esta clase proporciona las operaciones típicas de una cola, como Enqueue, Dequeue, Front y Back.</p>
MARCO CONCEPTUAL
<p>Un heap es una estructura de datos jerárquica que se basa en un árbol binario completo o casi completo. Se utiliza comúnmente para mantener y gestionar elementos con prioridades, donde se necesita un acceso rápido al elemento de máxima o mínima prioridad.</p> <p>La estructura y las propiedades de un heap permiten realizar operaciones eficientes, como la inserción y eliminación de elementos. Al insertar un nuevo elemento en un heap, se coloca al final del arreglo o se agrega como un nuevo nodo en el árbol, y luego se realiza un ajuste ascendente (up-heap) para asegurarse de que se cumpla la propiedad del heap correspondiente. En la eliminación de un elemento, generalmente se elimina el elemento de la</p>

raíz, que es el de máxima o mínima prioridad, y luego se reorganiza el heap mediante un ajuste descendente (down-heap) para mantener las propiedades del heap.

Existen dos tipos principales de heaps:

Un **max-heap** es un tipo de heap en el cual el valor almacenado en cada nodo es mayor o igual que los valores de sus nodos hijos. Esto significa que el elemento de máxima prioridad se encuentra en la raíz del heap. A medida que se desciende en el árbol, los valores disminuyen gradualmente.

Un **min-heap** es un tipo de heap en el cual el valor almacenado en cada nodo es menor o igual que los valores de sus nodos hijos. En este caso, el elemento de mínima prioridad se encuentra en la raíz del heap, y a medida que se desciende en el árbol, los valores aumentan gradualmente.

Ambos tipos de heaps comparten la propiedad de que se puede acceder rápidamente al elemento de máxima o mínima prioridad en tiempo constante. Sin embargo, el ordenamiento de los elementos y la posición del elemento prioritario varían entre ellos.

SOLUCIONES Y PRUEBAS

Para lograr la resolución del ejercicio se partió por crear el TDAHeap que era un interfaz llamada Heap y que contenía los métodos insert, donde su parámetro era un Elemento genérico "E" y no retornaba nada, el método remove, el cual elimina el último elemento y lo retornaba y por último el isEmpty que retorna un booleano si es que estaba o no vacío el ArrayList.

```
1 public interface Heap<E> {
2     void insert(E e);
3     E remove();
4     boolean isEmpty();
5 }
6
```

Con esto se creo la clase MaxHeap que implementaba la interfaz ya antes mencionada, la cual tenia 2 atributos: a)ArrayList de elementos genéricos, b)int size que era el tamaño del ArraList.

Método insert():

Se crea el método insertar que usa un método de apoyo llamado ascend() o flotar, lo que hace el método de apoyo es añadir el elemento “e” al final de la cola, para luego mediante un bucle reubicarlo en la posición que le corresponda, como se empieza de atrás hacia adelante, se resta uno y se divide entre 2 para encontrar al nodo una vez encontrado se compara ambos elementos y mediante la condición del bucle se decide si cambiarlo o no.

```
1  @Override
2  public void insert(E e) {
3      heap.add(e);
4      size++;
5      ascend(size-1);
6      System.out.println(toString());
7  }
8
9  private void ascend(int i) {
10     int parent = (i - 1) / 2;
11     while (i>0 && (heap.get(i).compareTo(heap.get(parent))>0)) {
12         swap(i, parent);
13         i = parent;
14         parent = (i - 1) / 2;
15     }
16 }
```

Método remove():

Este método es un poco más complejo que el método anterior, por lo que elimina el nodo raíz y se busca a un sucesor, también se usa un método de apoyo llamado descend, se intercambia la posición del nodo padre con el ultimo hijo del último nivel.

Lo que hace el método descend() es ubicar el nodo intercambiado en la posición que le corresponde, se define la posición de los hijos mediante posleft y posright y mediante un bucle se compara cada posición, caso contrario ninguno hijo sea mayor al padre se detiene el bucle o cuando ya no se encuentren más hijos. Se intercambia con el hijo mayor empezando primero por la izquierda.

```
1  @Override
2  public E remove() {
3      if (isEmpty()) throw new NullPointerException("El heap está vacío");
4      if (size == 1) return heap.remove(0);
5      E temp = heap.get(0);
6      heap.set(0, heap.remove(size-1));
7      descend(0);
8      return temp;
9  }
10
11 private void descend (int parent) {
12     int left, right;
13     int posLeft = (parent*2)+1;
14     int posRight = (parent*2)+2;
15     while (posLeft<size) {
16         left = heap.get(posLeft).compareTo(heap.get(parent));
17         right = heap.get(posRight).compareTo(heap.get(parent));
18         if (left>0) { // caso cuando el hijo de la izquierda es mayor
19             swap(parent, posLeft);
20             parent = posLeft;
21         }
22         else if (right>0) { //caso cuando el hijo de la derecha es mayor
23             swap(right, posRight);
24             parent = posRight;
25         }else //caso cuando ambos hijos son mayores al padre
26             break;
27         posLeft = (parent*2)+1;
28         posRight = (parent*2)+2;
29     }
30 }
```

Método end()

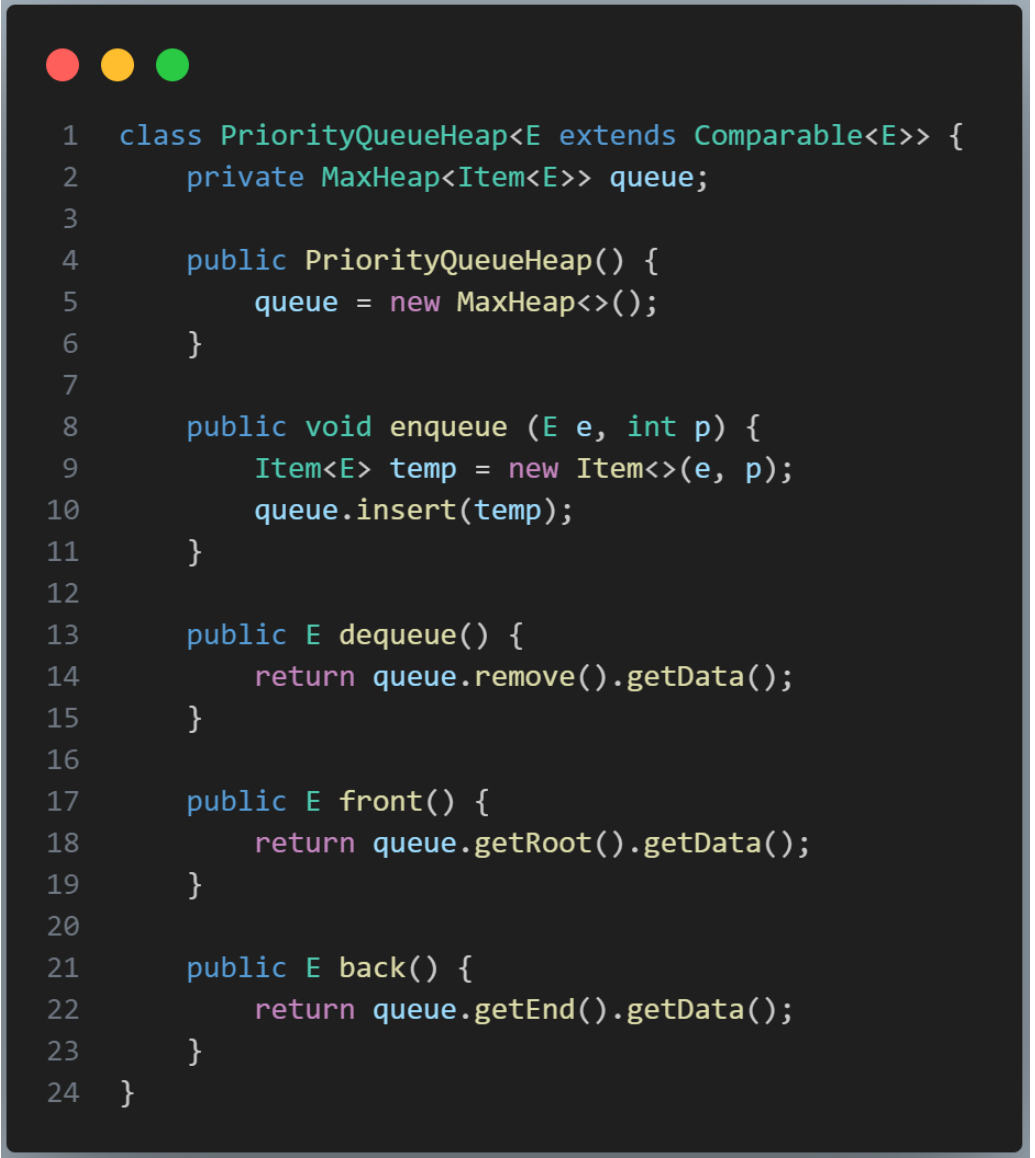
Este método está validado en que la cola no esté vacía, lo que hace es buscar mediante un bucle al elemento de menor prioridad, sea que esté al final o un medio de la cola lineal.

```
1 public E getEnd() { //menor valor de prioridad (1...5) ->1
2     if (isEmpty()) {
3         throw new NullPointerException("El heap está vacío");
4     }
5     E aux = heap.get(0);
6     for (int i = 0; i < this.size-1; i++) {
7         if (!(aux.compareTo(heap.get(i+1))<0)) aux =heap.get(i+1);
8     }
9     return aux;
10 }
```

La clase Item era genérica e implementa Comparable, tenía 2 atributos, su data y su prioridad, esta última era usada para el compareTo donde se retornaba la resta del su prioridad con la prioridad del argumento.


```
1  public class Item<E> implements Comparable<Item<E>> {
2
3      private E data;
4      private int priority;
5
6      public Item (E data, int p) {
7          this.data = data;
8          this.priority = p;
9      }
10
11     public E getData () {
12         return this.data;
13     }
14
15     public int getPriority() {
16         return this.priority;
17     }
18
19     @Override
20     public int compareTo(Item<E> e) {
21         if (e == null) {
22             throw new NullPointerException("El argumento no puede ser nulo");
23         }
24         return this.priority-e.getPriority();
25     }
26
27     @Override
28     public String toString() {
29         return this.data.toString()+"-p:"+this.priority;
30     }
31 }
```

Por último tenemos la clase PriorityQueueHeap que tenía solo 1 atributo: MaxHeap<Item<E>> denominado queue, con el cual se podían hacer operaciones de: a) insertar, representada por enqueue que recibía un elemento y su prioridad, el cual se transforma en un Item que se almacenaba en la estructura MaxHeap, b) remove, dequeue que elimina el elemento con mayor prioridad, c) front, retornaba el elemento de mayor prioridad y d) back, devuelve el elemento de menor prioridad.



```
1  class PriorityQueueHeap<E extends Comparable<E>> {
2      private MaxHeap<Item<E>> queue;
3
4      public PriorityQueueHeap() {
5          queue = new MaxHeap<>();
6      }
7
8      public void enqueue (E e, int p) {
9          Item<E> temp = new Item<>(e, p);
10         queue.insert(temp);
11     }
12
13     public E dequeue() {
14         return queue.remove().getData();
15     }
16
17     public E front() {
18         return queue.getRoot().getData();
19     }
20
21     public E back() {
22         return queue.getEnd().getData();
23     }
24 }
```

Se crea la clase persona con dos atributos un entero llamado edad y una cadena llamada nombre, con sus respectivos getters y setters y se sobrescribe el método toString.





```
1  public class Persona {
2      private int edad;
3      private String nombre;
4
5      public Persona(int edad, String nombre) {
6          this.edad = edad;
7          this.nombre = nombre;
8      }
9
10     public int getPriority() {
11         return edad;
12     }
13
14     @Override
15     public String toString() {
16         return nombre+"-"+edad+" años";
17     }
18 }
```

Para probar estos métodos se usó un main en la clase Test Head donde se crearon 2 PriorityQueueHeap uno de Integer y otro de personas, donde al ejecutar funcionó satisfactoriamente.

Se añaden instancias de la clase Persona y enteros a cada cola, probando cada método e imprimiendo correctamente.


```
● ● ●

1  public class TestHeap {
2      public static void main(String[] args) {
3          PriorityQueueHeap<Integer> priorityQueue = new PriorityQueueHeap<>();
4          priorityQueue.enqueue(10,1);
5          ultimo(priorityQueue);
6          primero(priorityQueue);
7
8          priorityQueue.enqueue(1,3);
9          ultimo(priorityQueue);
10         primero(priorityQueue);
11
12         priorityQueue.enqueue(13,2);
13         ultimo(priorityQueue);
14         primero(priorityQueue);
15
16         priorityQueue.enqueue(12,5);
17         ultimo(priorityQueue);
18         primero(priorityQueue);
19
20         priorityQueue.enqueue(11,5);
21         ultimo(priorityQueue);
22         primero(priorityQueue);
23         priorityQueue.enqueue(100,1);
24         ultimo(priorityQueue);
25         primero(priorityQueue);
26         PriorityQueueHeap<Persona> p = new PriorityQueueHeap<>();
27         Persona[] per = {new Persona(11,"Juan"),new Persona(19,"Paulo"),
28             new Persona(18,"Andre"),new Persona(91,"Joel")};
29         p.enqueue(per[0],per[0].getPriority());
30         System.out.println("Menor prioridad: "+p.back());
31         System.out.println("Mayor prioridad: "+p.front());
32         p.enqueue(per[1],per[1].getPriority());
33         System.out.println("Menor prioridad: "+p.back());
34         System.out.println("Mayor prioridad: "+p.front());
35         p.enqueue(per[2],per[2].getPriority());
36         System.out.println("Menor prioridad: "+p.back());
37         System.out.println("Mayor prioridad: "+p.front());
38         p.enqueue(per[3],per[3].getPriority());
39         System.out.println("Menor prioridad: "+p.back());
40         System.out.println("Mayor prioridad: "+p.front());
41
42     }
43     private static void ultimo(PriorityQueueHeap<Integer> p){
44         System.out.println("Menor prioridad: "+p.back());
45     }
46     private static void primero(PriorityQueueHeap<Integer> p){
47         System.out.println("Mayor prioridad: "+p.front());
48     }
49 }
```

	UNIVERSIDAD NACIONAL DE SAN AGUSTIN FACULTAD DE INGENIERÍA DE PRODUCCIÓN Y SERVICIOS ESCUELA PROFESIONAL DE INGENIERÍA DE SISTEMA	
Formato: Guía de Práctica de Laboratorio / Talleres / Centros de Simulación		
Aprobación: 2022/03/01	Código: GUIA-PRLE-001	Página: 10

EJECUCIÓN

```

"C:\Program Files\Java\jdk-20\bin\java.exe" "-javaagent:C:\Program Files\JetBrains\
[10-p:1]
Menor prioridad: 10
Mayor prioridad: 10
[1-p:3, 10-p:1]
Menor prioridad: 10
Mayor prioridad: 1
[1-p:3, 10-p:1, 13-p:2]
Menor prioridad: 10
Mayor prioridad: 1
[12-p:5, 1-p:3, 13-p:2, 10-p:1]
Menor prioridad: 10
Mayor prioridad: 12
[12-p:5, 11-p:5, 13-p:2, 10-p:1, 1-p:3]
Menor prioridad: 10
Mayor prioridad: 12
[12-p:5, 11-p:5, 13-p:2, 10-p:1, 1-p:3, 100-p:1]
Menor prioridad: 100
Mayor prioridad: 12
[Juan-11 años-p:11]
Menor prioridad: Juan-11 años
Mayor prioridad: Juan-11 años
[Paulo-19 años-p:19, Juan-11 años-p:11]
Menor prioridad: Juan-11 años
Mayor prioridad: Paulo-19 años
[Paulo-19 años-p:19, Juan-11 años-p:11, Andre-18 años-p:18]
Menor prioridad: Juan-11 años
Mayor prioridad: Paulo-19 años
[Joel-91 años-p:91, Paulo-19 años-p:19, Andre-18 años-p:18, Juan-11 años-p:11]
Menor prioridad: Juan-11 años
Mayor prioridad: Joel-91 años

Process finished with exit code 0

```

	UNIVERSIDAD NACIONAL DE SAN AGUSTIN FACULTAD DE INGENIERÍA DE PRODUCCIÓN Y SERVICIOS ESCUELA PROFESIONAL DE INGENIERÍA DE SISTEMA	
Formato: Guía de Práctica de Laboratorio / Talleres / Centros de Simulación		
Aprobación: 2022/03/01	Código: GUIA-PRLE-001	Página: 11

LECCIONES APRENDIDAS Y CONCLUSIONES

Los Heaps son útiles para gestionar elementos por su prioridad, ya que permiten mantener los elementos ordenados de forma eficiente. Utilizar estructuras de datos genéricas en Java proporciona flexibilidad y reutilización de código al permitir trabajar con diferentes tipos de elementos. La implementación de una cola de prioridad basada en un Heap es una forma eficiente de gestionar elementos con diferentes niveles de prioridad.

REFERENCIAS Y BIBLIOGRAFÍA

[1]Universitat Politècnica de València - UPV. El Montículo Binario | | UPV. (21 de septiembre de 2011). Accedido el 17 de junio de 2023. [Video en línea]. Disponible: https://www.youtube.com/watch?v=AD_J4ZUheik