

	<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>

INFORME DE TRABAJO PRÁCTICO

INFORMACIÓN BÁSICA					
ASIGNATURA:	Estructura de Datos y Algoritmos				
TITULO DEL TRABAJO:	Heaps				
NÚMERO DE PRÁCTICA:	03	AÑO LECTIVO:	2023	NRO. SEMESTRE:	III
FECHA DE PRESENTACIÓN	17/06/2023	HORA DE PRESENTACIÓN	12.00 PM		
INTEGRANTE (s) <ul style="list-style-type: none"> Condorios Yllapuma Jorge Enrique Mamani Uscamayta Agustin David 				NOTA (0-20)	
DOCENTE(s): <ul style="list-style-type: none"> Dra. Karim Guevara Puente de la Vega 					

MARCO CONCEPTUAL

Un heap es una estructura de datos especializada que se utiliza comúnmente para implementar colas de prioridad. Un heap es un árbol binario completo en el que cada nodo cumple una propiedad de ordenación específica, que puede ser "heap máximo" o "heap mínimo". En un heap máximo, cada nodo padre tiene un valor mayor o igual que los valores de sus nodos hijos, mientras que en un heap mínimo, cada nodo padre tiene un valor menor o igual que los valores de sus nodos hijos.

SOLUCIONES Y PRUEBAS

Clase Heap.java

```
import java.util.ArrayList;
import java.util.NoSuchElementException;

public class Heap <T extends Comparable<T>>{
    private ArrayList<T> heap;
    public Heap() {
        heap = new ArrayList<>();
    }

    public ArrayList<T> getHeap() {
```

```
        return heap;
    }

    public void insertar(T item) {
        heap.add(item);
        insertar(heap.size()-1);
    }

    private void insertar(int index) {
        int padre = (index - 1) / 2; // Calcula el índice del padre del
elemento en la posición index
        // Intercambia el elemento en la posición index con su padre si es
mayor
        while (index > 0 && heap.get(index).compareTo(heap.get(padre)) > 0)
        {
            intercambio(index, padre);
            index = padre;
            padre = (index - 1) / 2;
        }
    }

    public T eliminar() {
        if (isEmpty()) {
            throw new NoSuchElementException("El heap está vacío");
        }

        T root = heap.get(0);
        T ultimoItem = heap.remove(heap.size() - 1); // Remueve el último
elemento del ArrayList
        if (!isEmpty()) {
            heap.set(0, ultimoItem); // Reemplaza la raíz con el último
elemento del heap
            eliminar(0);
        }
        return root; // Retorna el elemento raíz removido
    }

    private void eliminar(int index) {
        int leftHijo = 2 * index + 1;
        int rightHijo = 2 * index + 2;
```

```
int mayor = index;
// Actualiza el índice del mayor elemento si el hijo izquierdo es
mayor
        if (leftHijo < heap.size() &&
heap.get(leftHijo).compareTo(heap.get(mayor)) > 0) {
    mayor = leftHijo;
}

        if (rightHijo < heap.size() &&
heap.get(rightHijo).compareTo(heap.get(mayor)) > 0) {
    mayor = rightHijo; // Actualiza el índice del mayor elemento si
el hijo derecho es mayor
}

    if (mayor != index) {
        intercambio(index, mayor);
        eliminar(mayor); //
    }
}

private void intercambio(int i, int j) {
    T temp = heap.get(i);
    heap.set(i, heap.get(j));
    heap.set(j, temp);
}

//retorna la raiz del heap
public T raiz() {
    if (isEmpty()) {
        throw new NoSuchElementException("El heap está vacío");
    }
    return heap.get(0);
}

public boolean isEmpty() {
    return heap.isEmpty();
}
}
```

- La clase **Heap** utiliza un ArrayList llamado heap para almacenar los elementos del heap.
- El método **insertar(T item)** permite insertar un elemento en el heap. Agrega el elemento al final del ArrayList y luego realiza el proceso de "sift up" (intercambio con el padre) para mantener la propiedad del heap.
- El método privado **insertar(int index)** realiza el proceso de "sift up" para asegurarse de que el elemento insertado se coloque en la posición correcta del heap.
- El método **eliminar()** elimina y devuelve el elemento raíz del heap. Reemplaza la raíz con el último elemento del ArrayList y luego realiza el proceso de "sift down" (intercambio con los hijos) para mantener la propiedad del heap.
- El método privado **eliminar(int index)** realiza el proceso de "sift down" para asegurarse de que el elemento reemplazado se coloque en la posición correcta del heap.
- El método privado **intercambio(int i, int j)** intercambia dos elementos en el ArrayList.
- El método **raiz()** devuelve la raíz del heap, que es el elemento en la posición 0 del ArrayList.
- El método **isEmpty()** verifica si el heap está vacío.

Clase PrioridadItem



```
public class PrioridadItem <E extends Comparable<E>> implements
Comparable<PrioridadItem<E>>{
    private E item;
    private int prioridad;

    public PrioridadItem(E item, int prioridad) {
        this.item = item;
        this.prioridad = prioridad;
    }

    public E getItem() {
        return item;
    }

    public int getPrioridad() {
        return prioridad;
    }

    @Override
    public int compareTo(PrioridadItem<E> pItem) {
        return Integer.compare(pItem.getPrioridad(), prioridad);
    }
}
```

	<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: 5</p>

- La clase **PrioridadItem** utiliza un tipo genérico E que debe ser comparable (extends Comparable<E>).
- Los atributos de la clase son item, que representa el elemento en sí, y prioridad, que indica la prioridad del elemento.
- El constructor **PrioridadItem(E item, int prioridad)** recibe un elemento y su prioridad y los asigna a los atributos correspondientes.
- Los métodos getItem() y getPrioridad() permiten acceder a los atributos item y prioridad, respectivamente.
- La clase implementa la interfaz **Comparable<PrioridadItem<E>>** y, por lo tanto, debe implementar el método **compareTo()**. Este método compara el objeto actual (this) con otro objeto pitem de tipo **PrioridadItem<E>**. La comparación se realiza comparando las prioridades de los elementos y devolviendo el resultado de la comparación.
- El método **compareTo()** utiliza **Integer.compare()** para realizar la comparación de las prioridades de manera conveniente.

En resumen, la clase PrioridadItem representa un elemento con su respectiva prioridad y proporciona métodos para acceder a los atributos y comparar objetos de tipo PrioridadItem<E> en función de sus prioridades. Esta clase se utiliza en el contexto de una cola de prioridad para asignar prioridades a los elementos y ordenarlos en consecuencia.

Clase PriorityQueueHeap

```
import java.util.ArrayList;
import java.util.NoSuchElementException;

public class PriorityQueueHeap<T extends Comparable<T>>{
    private Heap<PrioridadItem<T>> heap;
    public PriorityQueueHeap() {
        heap = new Heap<>();
    }

    public void enqueue(T item, int prioridad) {
        PrioridadItem<T> pItem = new PrioridadItem<>(item, prioridad);
        heap.insertar(pItem);
    }

    public boolean isEmpty() {
        return heap.isEmpty();
    }

    public T front() {
        if (isEmpty()) {
            throw new NoSuchElementException("La cola de prioridad está vacía");
        }
    }
}
```

```
        return heap.raiz().getItem();
    }

    public T back() {
        if (isEmpty()) {
            throw new NoSuchElementException("La cola de prioridad está vacía");
        }
        ArrayList<PrioridadItem<T>> items = heap.getHeap();
        return items.get(items.size() - 1).getItem();
    }

    public T dequeue() {
        if (isEmpty()) {
            throw new NoSuchElementException("La cola de prioridad está vacía");
        }
        return heap.eliminar().getItem();
    }
}
```

- La clase **PriorityQueueHeap** utiliza un Heap de elementos de tipo **PrioridadItem<T>**, donde T es un tipo genérico que debe ser comparable (extends **Comparable<T>**).
- El constructor **PriorityQueueHeap()** inicializa el Heap.
- El método **enqueue(T item, int prioridad)** recibe un elemento y su prioridad, crea un nuevo **PrioridadItem** con los parámetros recibidos y lo inserta en el Heap utilizando el método **insertar()**.
- El método **isEmpty()** verifica si la cola de prioridad está vacía verificando si el Heap está vacío.
- El método **front()** devuelve el elemento de mayor prioridad en la cola de prioridad. Si la cola está vacía, se lanza una excepción.
- El método **back()** devuelve el elemento de menor prioridad en la cola de prioridad. Si la cola está vacía, se lanza una excepción. Obtiene el **ArrayList** subyacente del Heap y accede al último elemento para obtener el elemento de menor prioridad.
- El método **dequeue()** elimina y devuelve el elemento de mayor prioridad en la cola de prioridad. Si la cola está vacía, se lanza una excepción.

En resumen, la clase **PriorityQueueHeap** proporciona una interfaz para interactuar con una cola de prioridad utilizando un heap como estructura de datos subyacente. Permite insertar elementos, eliminar el elemento de mayor prioridad y acceder al elemento de mayor y menor prioridad en la cola.

Clase Main

	<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: 7</p>

```

public class Main {
    public static void main(String[] args) {
        PriorityQueueHeap<String> lista = new PriorityQueueHeap<>();
        lista.enqueue("Agustin", 2);
        lista.enqueue("Jorge", 1);
        lista.enqueue("Joel", 3);

        System.out.println("El elemento de mas alta prioridad es: "+
lista.front());

        System.out.println("El elemento de mas baja prioridad es: "+
lista.back());

        // Eliminar elementos de la cola de prioridad y mostrarlos
        System.out.println("Elementos eliminados de la cola de
prioridad:");

        while (!lista.isEmpty()) {
            String item = lista.dequeue();
            System.out.println(item);
        }
    }
}

```

LECCIONES APRENDIDAS Y CONCLUSIONES

Durante el desarrollo de este trabajo práctico, se adquirieron diversas lecciones y se llegaron a las siguientes conclusiones:

1. Los heaps son estructuras de datos eficientes para implementar colas de prioridad: Los heaps proporcionan un acceso rápido al elemento de mayor o menor prioridad, lo que los hace ideales para implementar colas de prioridad. La estructura de heap permite mantener los elementos ordenados en función de su prioridad y realizar operaciones de inserción y eliminación eficientes.
2. La implementación de un heap puede variar: En el código proporcionado, se muestra una implementación de heap utilizando un ArrayList y una clase personalizada. Sin embargo, existen diferentes enfoques para implementar heaps, como el uso de arrays, listas enlazadas u otras estructuras de datos internas. La elección de la implementación depende de los requisitos y las características específicas del problema.
3. Las operaciones en el heap tienen una complejidad de tiempo eficiente: Las operaciones fundamentales en un heap, como la inserción, la eliminación y el acceso al elemento de mayor o menor prioridad, tienen una complejidad de tiempo eficiente en comparación con otras estructuras de datos. En un heap binario, estas operaciones tienen una complejidad de tiempo de $O(\log n)$, donde n es el número de elementos en el heap.

	<p style="text-align: center;">UNIVERSIDAD NACIONAL DE SAN AGUSTIN FACULTAD DE INGENIERÍA DE PRODUCCIÓN Y SERVICIOS ESCUELA PROFESIONAL DE INGENIERÍA DE SISTEMA</p>	
<p style="text-align: center;">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: 8</p>

4. Los heaps pueden ser utilizados en una variedad de aplicaciones: Debido a su capacidad para gestionar elementos con prioridades, los heaps tienen aplicaciones en diversos campos, como algoritmos de búsqueda y clasificación, programación dinámica, algoritmos de compresión de datos, programación de redes y más.

En resumen, los heaps son estructuras de datos poderosas para gestionar elementos con prioridades, y su implementación adecuada puede mejorar la eficiencia y el rendimiento de ciertas operaciones. El código proporcionado nos muestra una implementación básica de una cola de prioridad utilizando un heap, lo que demuestra su utilidad en la práctica.

REFERENCIAS Y BIBLIOGRAFÍA

- [1]"Heap Space y Stack Memory en Java - Refactorizando". Refactorizando. <https://refactorizando.com/heap-space-stack-memory-java/#:~:text=El%20espacio%20Heap%20es%20usado,almacenados%20en%20la%20memory%20stack>. (accedido el 17 de junio de 2023).