INFORME DE TRABAJO PRÁCTICO

INFORMACIÓN BÁSICA					
ASIGNATURA:	**Estructura de Datos y Algoritmos**				
TÍTULO DEL TRABAJO:	**HEAPS - MONTÍCULOS**				
NÚMERO DE TRABAJO:	**02**	**AÑO LECTIVO:**	**2023A**	**NRO. SEMESTRE:**	**III**
FECHA DE PRESENTACIÓN	**17/06/2023**	**HORA DE PRESENTACIÓN**	**11:15 am**		
INTEGRANTE (s) • **Cusilayme García, José Luis**				**NOTA (0-20)**	
DOCENTE(s): • **Dra. Karim Guevara Puente de la Vega**					

INTRODUCCIÓN

¿Qué es un Heap – Montículo?

Es un árbol binario completo, las hojas están en los dos últimos niveles, y en nivel más bajo se rellena de izquierda a derecha

Parcialmente ordenado, debido a que tiene todas y cada una de sus ramas totalmente ordenadas, ya sea de forma creciente o decreciente.

- HeapMax: raíz es mayor que sus descendientes
  - El máximo está en la raíz del árbol
- HeapMin: raíz es menor que sus descendientes
  - El mínimo está en la raíz del árbol

Heaps - Almacenamiento

- A través de los heaps, se consigue un mejor equilibrio en la búsqueda y eliminación de elementos.
- Se puede almacenar en un arreglo

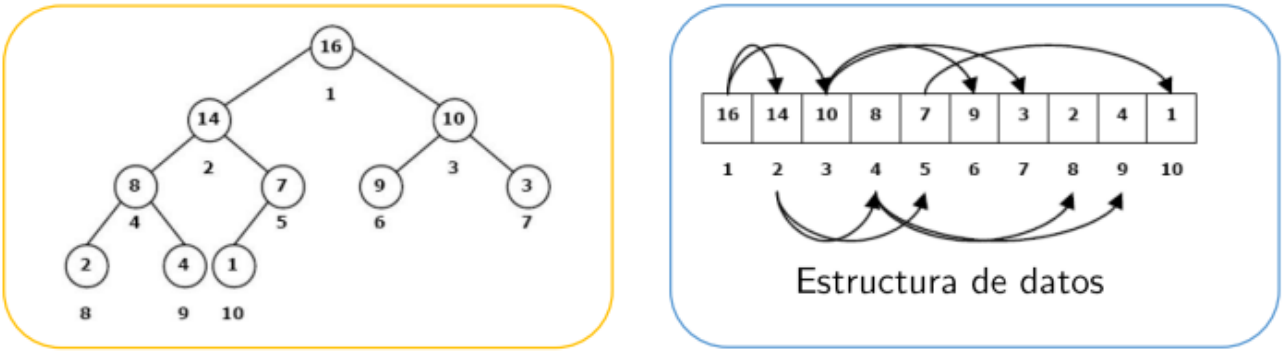

Fig. 1 Heap - Almacenamiento

Heap - Estructura

- La raíz del árbol representada por el heap es A[1] y, dado el índice i de un nodo, los índices del PADRE(i), del hijo izquierdo IZQUIERDA(i) y del hijo derecho DERECHA(i) son:
  - PADRE(i)
    - return ë i/2 û ;
  - IZQUIERDA(i)
    - return 2 * i;
  - DERECHA(i)
    - return (2 * i) + 1;

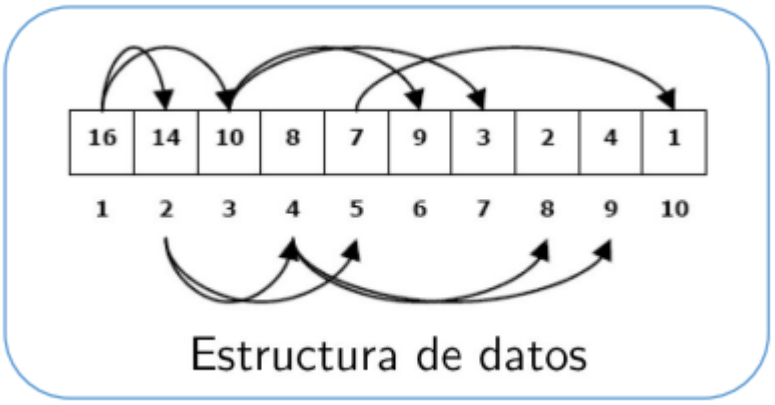

Fig. 2 Heap - Estructura

Para la resolución de este trabajo se tomará en cuenta los conceptos de head dados previamente y además de esto tomaremos los conceptos de eliminación del Heap, tanto el HEAPMAX como el Heap normal.

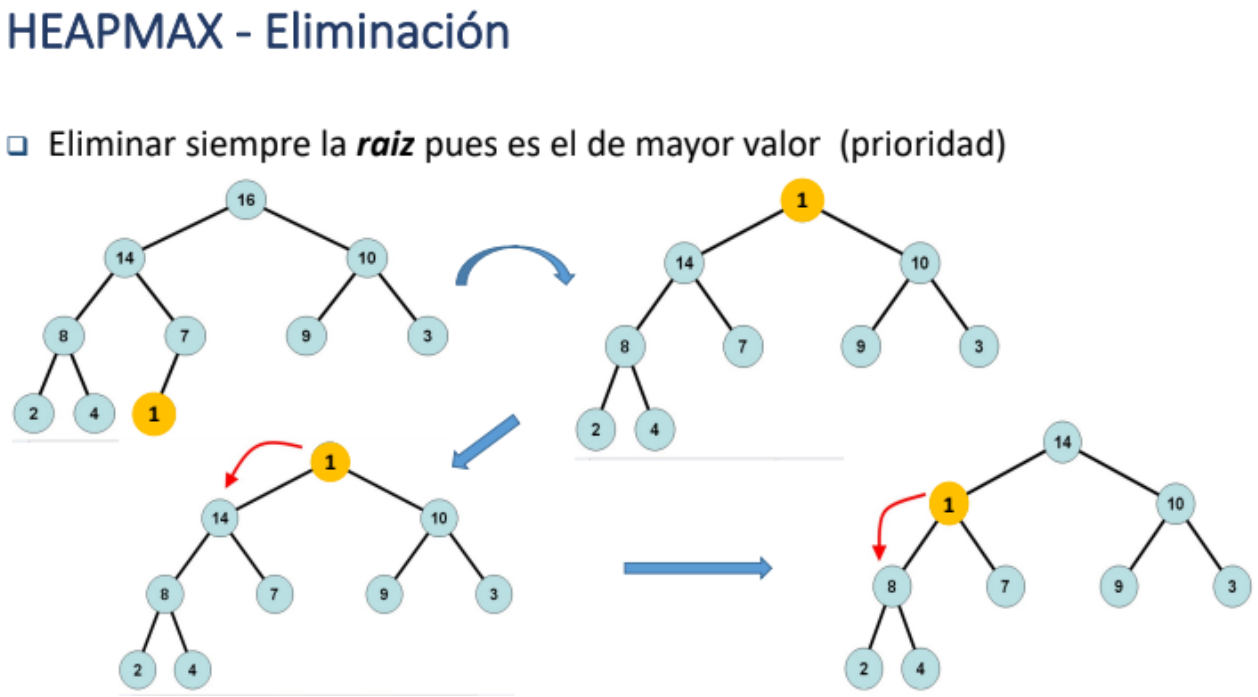

Fig. 3 HEAPMAX - Eliminación

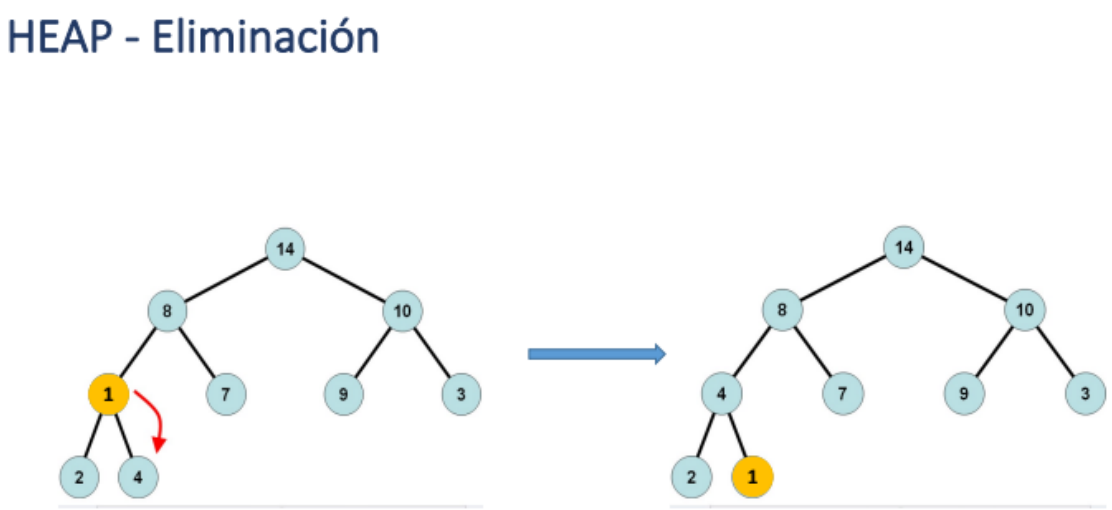

Fig. 4 HEAP - Eliminación

SOLUCIONES Y PRUEBAS

EJERCICIO:

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:

- Enqueue (x, p) : inserta un elemento a la cola ‘x’ de prioridad ‘p’ a la cola. Como la cola está sobre un heap, este deberá ser insertado en el heap-max y reubicado de acuerdo a su prioridad.
- 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.
- Front() : solo devuelve el elemento de mayor prioridad.
- 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.

RESOLUCIÓN:

La clase Heap<T> es una implementación genérica del TAD Heap. Utiliza un ArrayList para almacenar los elementos del heap. Los elementos deben ser comparables entre sí (T extends Comparable<T>).

El constructor de Heap inicializa el ArrayList vacío.

El método insert(T item) agrega un elemento al heap. Se añade al final del ArrayList y luego se reordena hacia arriba (siftUp) para mantener la propiedad del heap.

El método remove() elimina y devuelve el elemento raíz del heap. Se intercambia el elemento raíz con el último elemento del ArrayList, se elimina el último elemento y se reordena hacia abajo (siftDown) para restaurar la propiedad del heap.

El método peek() devuelve el elemento raíz del heap sin eliminarlo.

El método isEmpty() verifica si el heap está vacío.

El método size() devuelve el número de elementos en el heap.

El método get(int index) devuelve el elemento en la posición dada del heap.

Los métodos siftUp(int index) y siftDown(int index) son operaciones auxiliares utilizadas para mantener la propiedad del heap al insertar y eliminar elementos.

```java
import java.util.*;
class Heap<T extends Comparable<T>> {
    private ArrayList<T> heap;

    public Heap() {
        heap = new ArrayList<>();
    }

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

    public T remove() {
        if (isEmpty()) {
            throw new NoSuchElementException("El heap est\uFFFD vac\uFFFDo.");
        }

        T root = heap.get(0);
        T lastItem = heap.remove(heap.size() - 1);

        if (!isEmpty()) {
            heap.set(0, lastItem);
            siftDown(index:0);
        }

        return root;
    }

    public T peek() {
        if (isEmpty()) {
            throw new NoSuchElementException("El heap est\uFFFD vac\uFFFDo.");
        }

        return heap.get(0);
    }

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

```
44
45      public T get(int index) {
46          if (index < 0 || index >= heap.size()) {
47              throw new IndexOutOfBoundsException("ndice fuera de rango.");
48          }
49
50          return heap.get(index);
51      }
52      private void siftUp(int index) {
53          int parentIndex = (index - 1) / 2;
54          while (index > 0 && heap.get(index).compareTo(heap.get(parentIndex)) > 0) {
55              swap(index, parentIndex);
56              index = parentIndex;
57              parentIndex = (index - 1) / 2;
58          }
59      }
60
61      private void siftDown(int index) {
62          int leftChildIndex = 2 * index + 1;
63          int rightChildIndex = 2 * index + 2;
64          int largestIndex = index;
65
66          if (leftChildIndex < heap.size() && heap.get(leftChildIndex).compareTo(heap.get(largestIndex)) > 0) {
67              largestIndex = leftChildIndex;
68          }
69
70          if (rightChildIndex < heap.size() && heap.get(rightChildIndex).compareTo(heap.get(largestIndex)) > 0) {
71              largestIndex = rightChildIndex;
72          }
73
74          if (largestIndex != index) {
75              swap(index, largestIndex);
76              siftDown(largestIndex);
77          }
78      }
79
80      private void swap(int i, int j) {
81          T temp = heap.get(i);
82          heap.set(i, heap.get(j));
83          heap.set(j, temp);
84      }
85  }
```

Fig 5 Clase Heap

- La clase PriorityQueueHeap<T> implementa una cola de prioridad utilizando el heap anteriormente definido.
- El constructor de PriorityQueueHeap crea un nuevo objeto Heap<PriorityQueueNode<T>> para almacenar los nodos de la cola de prioridad.
- El método enqueue(T item, int priority) agrega un elemento a la cola de prioridad con su respectiva prioridad. Crea un nuevo PriorityQueueNode con el elemento y la prioridad proporcionados y lo inserta en el heap.

- El método dequeue() elimina y devuelve el elemento de mayor prioridad de la cola de prioridad. Utiliza el método remove() del heap para obtener el nodo de mayor prioridad y luego devuelve el elemento asociado a ese nodo.
- El método front() devuelve el elemento de mayor prioridad de la cola de prioridad sin eliminarlo. Utiliza el método peek() del heap para obtener el nodo de mayor prioridad y luego devuelve el elemento asociado a ese nodo.
- El método back() devuelve el elemento de menor prioridad de la cola de prioridad sin eliminarlo. Obtiene el último elemento del heap utilizando el método get(int index) del heap.
- El método isEmpty() verifica si la cola de prioridad está vacía.
- La clase PriorityQueueNode<T> es una clase interna utilizada para representar los nodos de la cola de prioridad. Cada nodo contiene un elemento y su respectiva prioridad. Implementa la interfaz Comparable para poder comparar los nodos en función de sus prioridades.

```
1
2   class PriorityQueueHeap<T extends Comparable<T>> {
3       private Heap<PriorityQueueNode<T>> heap;
4
5       public PriorityQueueHeap() {
6           heap = new Heap<>();
7       }
8
9       public void enqueue(T item, int priority) {
10          PriorityQueueNode<T> node = new PriorityQueueNode<>(item, priority);
11          heap.insert(node);
12      }
13
14      public T dequeue() {
15          PriorityQueueNode<T> node = heap.remove();
16          return node.getItem();
17      }
18
19      public T front() {
20          PriorityQueueNode<T> node = heap.peek();
21          return node.getItem();
22      }
23
24      public T back() {
25          int lastIndex = heap.size() - 1;
26          PriorityQueueNode<T> node = heap.get(lastIndex);
27          return node.getItem();
28      }
29
30      public boolean isEmpty() {
31          return heap.isEmpty();
32      }
33
```

Fig 6 Clase PriorityQueueHeap

```java
    // Clase interna para representar los nodos de la cola de prioridad
    private class PriorityQueueNode<T extends Comparable<T>> implements Comparable<PriorityQueueNode<T>> {
        private T item;
        private int priority;

        public PriorityQueueNode(T item, int priority) {
            this.item = item;
            this.priority = priority;
        }

        public T getItem() {
            return item;
        }

        public int getPriority() {
            return priority;
        }

        @Override
        public int compareTo(PriorityQueueNode<T> other) {
            return Integer.compare(other.getPriority(), priority);
        }
    }

}
```

Fig 7 Clase Interna PriorityQueueNode

En el ejemplo de uso en la clase Main, se muestra cómo utilizar la cola de prioridad. Se agregan varios elementos con diferentes prioridades, se obtiene el elemento de mayor y menor prioridad utilizando los métodos front() y back(), y se eliminan los elementos en orden de prioridad utilizando el método dequeue().

```java
public class Main {
    Run | Debug
    public static void main(String[] args) {
        PriorityQueueHeap<String> priorityQueue = new PriorityQueueHeap<>();
        priorityQueue.enqueue(item:"Elemento 1", priority:3);
        priorityQueue.enqueue(item:"Elemento 2", priority:1);
        priorityQueue.enqueue(item:"Elemento 3", priority:2);

        System.out.println("Elemento de mayor prioridad: " + priorityQueue.front());
        System.out.println("Elemento de menor prioridad: " + priorityQueue.back());

        while (!priorityQueue.isEmpty()) {
            String item = priorityQueue.dequeue();
            System.out.println("Elemento eliminado: " + item);
        }
    }
}
```

LECCIONES APRENDIDAS Y CONCLUSIONES

Lecciones aprendidas:

- La implementación de una cola de prioridad utilizando un heap proporciona un tiempo de ejecución eficiente para las operaciones de inserción y eliminación de elementos con mayor prioridad.
- El uso de estructuras de datos genéricas y programación orientada a objetos permite crear implementaciones flexibles y reutilizables.
- El diseño de clases y métodos bien encapsulados promueve un código limpio y mantenible.
- La comprensión de las propiedades y operaciones del heap es fundamental para implementar correctamente una cola de prioridad.

Conclusión:

En este ejercicio, se ha implementado una cola de prioridad utilizando un heap como estructura de datos subyacente. La implementación del heap permite realizar las operaciones de inserción, eliminación y acceso a los elementos con mayor y menor prioridad de manera eficiente. El uso de una clase genérica y la implementación de la interfaz Comparable proporcionan flexibilidad para trabajar con diferentes tipos de elementos y prioridades.

La implementación de una cola de prioridad basada en un heap es una estrategia poderosa y eficiente para resolver problemas que implican la gestión de elementos con diferentes niveles de prioridad. Al comprender los conceptos y técnicas presentadas en este ejercicio, se puede aplicar este conocimiento a una amplia gama de problemas en los que se requiere un manejo eficiente de la prioridad.

REFERENCIAS Y BIBLIOGRAFÍA

[1] Molto, G. El montículo Binario. [Online] Recuperado de https://aulavirtual.unsa.edu.pe/2023A/mod/page/view.php?id=44926

[2]De la Vega K. Heaps Montículos. [Online] Recuperado de https://drive.google.com/file/d/1Ma9WERoSytydCbQc9S3XOsiUbim1XXQq/view.