"AÑO DE LA UNIDAD, LA PAZ Y EL DESARROLLO"

UNIVERSIDAD NACIONAL DE SAN AGUSTÍN DE AREQUIPA

FACULTAD DE INGENIERÍA DE PRODUCCIÓN Y SERVICIOS

ESCUELA PROFESIONAL DE INGENIERÍA DE SISTEMAS

TRABAJO GRUPAL

Tema

PRÁCTICA NUM. 3: HEAPS

ASIGNATURA:

ESTRUCTURA DE DATOS Y ALGORITMOS - GRUPO-A

DOCENTE:

Dra.Karim, Guevara Puente de la Vega

INTEGRANTES:

Condori Leon, Joel Isaias

Hancco Soncco, Vladimir Jaward

AREQUIPA 2023

Índice

1. ENUNCIADOS:

1.1. 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 este almacenado sobre un ArrayList con las operaciones de inserción y eliminación. Este TAD debe de ser un heap maximo.
- Implemente la clase PriorityQueueHeap generica que utilice como estructura de datos el heap desarrollado en el punto anterior. Esta clase debe tener las operaciones de una cola tales como:
  1. 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.
  2. 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.
  3. Front() : solo devuelve el elemento de mayor priorioridad.
  4. 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.

2. RESOLUCION DEL EJERCICIO :

- Implemente el TAD Heap genérico que este almacenado sobre un ArrayList con las operaciones de inserción y eliminación. Este TAD debe de ser un heap maximo.
- Creamos la clase Heap, la cual implementa un TAD(Tipo Abstracto de Datos) Heap genérico que almacena los elementos en un ArrayList. Incluye métodos para la inserción, eliminación, obtención del primer y último elemento, verificación de si está vacío, obtener el tamaño y limpiar el heap. Además, se ha sobrescrito el método toString() para mostrar el contenido del heap.

2.1. CLASE HEAP

Listing 1: clase Heap

```
1  import java.util.ArrayList;
2
3  import myExceptions.ExceptionIsEmpty;
4
5  public class Heap<T extends Comparable<T>>{
```

- En esta clase se hace uso de el atributo heap de tipo ArrayList, tal y como el enunciado solicita, es utilizado para almacenar los elementos del Heap y proporciona flexibilidad, acceso rápido y facilidad de manipulación de los elementos en la implementación del Heap máximo.

Listing 2: clase Heap

```
1  public class Heap<T extends Comparable<T>>{
2  private ArrayList<T> heap;
```

- El constructor crea un nuevo objeto Heap y establece un ArrayList vacío que se utilizará para almacenar los elementos del Heap. Este constructor proporciona la base para trabajar con instancias de la clase Heap y permite la posterior inserción y eliminación de elementos en el Heap.

Listing 3: clase Heap

```
1       public Heap() {
2           heap = new ArrayList<>();
3       }
```

- Metodo insert(T element) Este método se utiliza para insertar un elemento en el Heap. El elemento se agrega al inicio del ArrayList, lo que corresponde a la raiz del árbol, y luego se realiza un proceso de ajuste descendente (sift-down) para mantener la propiedad del Heap.
- Asegura que el nuevo elemento se inserte correctamente en el Heap máximo y se mantenga la propiedad del Heap. Esto garantiza que el elemento con el valor máximo esté en la posición raíz del Heap.

Listing 4: clase Heap

```
1           public void insert(T element) {
2           if (heap.isEmpty()) {
3               heap.add(element);
4               return;
5           }
6           heap.add(0, element);
7           int index;
8           for (int i = 0; i < 2; i++) {
9               index = 0;
10              while (index < heap.size()) {
11                  int leftChildIndex = 2 * index + 1;
12                  int rightChildIndex = 2 * index + 2;
13                  int maxChildIndex = -1;
14                  if (leftChildIndex < heap.size()) {
15                      if (rightChildIndex < heap.size()) {
16                          maxChildIndex = heap.get(leftChildIndex).compareTo
                                (heap.get(rightChildIndex)) > 0
17                                  ? leftChildIndex
18                                  : rightChildIndex;
19                      } else {
20                          maxChildIndex = leftChildIndex;
21                      }
22                      if (heap.get(index).compareTo(heap.get(maxChildIndex))
                            < 0) {
23                          T temp = heap.get(maxChildIndex);
24                          heap.set(maxChildIndex, heap.get(index));
25                          heap.set(index, temp);
26                          index = maxChildIndex;
27                      } else {
28                          break;
29                      }
30                  } else {
31                      break;
32                  }
33              }
34          }
35      }
```

- Se tiene el metodo T remove() se encarga de eliminar y devolver el elemento con la máxima prioridad del Heap.

  1. Verifica si el Heap está vacío y lanza una excepción si es el caso.
  2. Obtiene el elemento con máxima prioridad, que corresponde a la raíz del Heap.
  3. Elimina el último elemento del Heap y lo asigna como la nueva raíz, si el Heap contiene más de un elemento.
  4. Reajusta el Heap hacia abajo para mantener su propiedad de Heap máximo.
  5. Retorna el elemento con máxima prioridad.

Listing 5: clase Heap

```
1         public T remove() throws ExceptionIsEmpty {
2         if (heap.isEmpty())
3             throw new ExceptionIsEmpty("Heap is empty");
4         T root = heap.get(0);
5         T lastElement = heap.remove(heap.size() - 1);
6         if (!heap.isEmpty()) {
7             heap.set(0, lastElement);
8             int index = 0;
9             int size = heap.size();
10            while (index < size / 2) {
11                int leftChildIndex = 2 * index + 1;
12                int rightChildIndex = 2 * index + 2;
13                int maxIndex = leftChildIndex;
14                if (rightChildIndex < size && heap.get(rightChildIndex).
                      compareTo(heap.get(leftChildIndex)) > 0) {
15                    maxIndex = rightChildIndex;
16                }
17                if (heap.get(index).compareTo(heap.get(maxIndex)) < 0) {
18                    T temp = heap.get(index);
19                    heap.set(index, heap.get(maxIndex));
20                    heap.set(maxIndex, temp);
21                    index = maxIndex;
22                } else {
23                    break;
24                }
25            }
26        }
27        return root;
28    }
```

- El método T first() en la clase Heap devuelve el elemento con la máxima prioridad, es decir, el elemento que se encuentra en la raíz del Heap. En resumen, realiza las siguientes acciones:

  1. Verifica si el Heap está vacío y lanza una excepción si es el caso.
  2. Obtiene el elemento en la posición 0 del ArrayList, que corresponde a la raíz del Heap.
  3. Retorna el elemento con máxima prioridad.

Listing 6: clase Heap

```
1         public T first() throws ExceptionIsEmpty {
2         if (heap.isEmpty())
3             throw new ExceptionIsEmpty("Heap is empty");
4         T root = heap.get(0);
5         return root;
6         }
```

- El método last() en la clase Heap devuelve el último elemento del Heap, es decir, el elemento con la mínima prioridad. En resumen, realiza las siguientes acciones:

  1. Verifica si el Heap está vacío y lanza una excepción si es el caso.
  2. Obtiene el último elemento del ArrayList, que corresponde al último elemento del Heap.
  3. Retorna el elemento con mínima prioridad.

Listing 7: clase Heap

```
1        public T last() throws ExceptionIsEmpty {
2        if (heap.isEmpty())
3            throw new ExceptionIsEmpty("Heap is empty");
4        T last = heap.get(heap.size() - 1);
5        return last;
6        }
```

- El método toString() en la clase Heap devuelve una representación en forma de cadena de todos los elementos presentes en el Heap. En resumen, realiza las siguientes acciones:
- El método isEmpty() en la clase Heap verifica si el Heap está vacío y retorna un valor booleano indicando si está vacío o no. En resumen, realiza las siguientes acciones:

Listing 8: clase Heap

```
1        public String toString() {
2            String str = "[";
3            for (int i = 0; i < heap.size(); i++) {
4                str += heap.get(i);
5                if (i < heap.size() - 1)
6                    str += ", ";
7            }
8            str += "]";
9            return str;
10       }
11       public boolean isEmpty() {
12       return heap.isEmpty();
13       }
```

2.2. CLASE PriorityQueueHeap

- Se crea la clase PriorityQueueHeap que es es una implementación de una cola de prioridad que utiliza un Heap y una SkipList como estructuras de datos subyacentes. Está parametrizada para trabajar con elementos genéricos T que deben implementar la interfaz Comparable¡T¿.
- La clase tiene los atributos principales:

  1. heap: es un objeto de la clase Heap que almacena los elementos con sus prioridades y mantiene un orden de máxima prioridad.
  2. skipList: es un objeto de la clase SkipList que permite acceder y buscar los elementos de manera eficiente en función de su valor y prioridad.
  3. El constructor de la clase PriorityQueueHeap crea una nueva instancia de la clase. En este caso, inicializa los atributos heap y skipList utilizando sus respectivos constructores sin argumentos.

Listing 9: clase PriorityQueueHeap

```
1        import myExceptions.ExceptionIsEmpty;
2
3        public class PriorityQueueHeap<T extends Comparable<T>>  {
4            private Heap<PriorityElement<T>> heap;
5            private SkipList<PriorityElement<T>> skipList;
6         public PriorityQueueHeap() {
7         heap = new Heap<>();
8         skipList = new SkipList<>();
9        }
```

- El método enqueue(T element, int priority) se utiliza para agregar un elemento a la cola de prioridad con la prioridad especificada. Toma dos argumentos: element, que representa el elemento a agregar, y priority, que indica la prioridad del elemento.
- Dentro del método, se crea un nuevo objeto PriorityElement que encapsula el elemento y su prioridad. Luego, se inserta este objeto tanto en el Heap como en la SkipList utilizando los respectivos métodos de inserción.

Listing 10: clase PriorityQueueHeap

```
1    public void enqueue(T element, int priority) {
2        PriorityElement<T> priorityElement = new PriorityElement<>(element
             , priority);
3        heap.insert(priorityElement);
4        skipList.insert(priorityElement);
5    }
```

- El método dequeue() se utiliza para extraer y eliminar el elemento de mayor prioridad de la cola de prioridad. Retorna el elemento que fue removido.
- Dentro del método, se invoca el método remove() del Heap para obtener el elemento de mayor prioridad. Luego, se utiliza este elemento para realizar la eliminación correspondiente en la SkipList mediante el método remove(). Finalmente, se retorna el elemento extraído.

Listing 11: clase PriorityQueueHeap

```
1        public T dequeue() throws ExceptionIsEmpty {
2    if (heap.isEmpty()) {
3        throw new ExceptionIsEmpty("Priority queue is empty");
4    }
5    PriorityElement<T> priorityElement = heap.remove();
6    skipList.remove(priorityElement);
7    return priorityElement.getElement();
8    }
```

- El método front() se utiliza para obtener el elemento de mayor prioridad en la cola de prioridad sin eliminarlo. Retorna el elemento de mayor prioridad.
- Dentro del método, se invoca el método first() del Heap para obtener el elemento de mayor prioridad. Luego, se utiliza el método getElement() en el objeto PriorityElement retornado para obtener el elemento subyacente. Finalmente, se retorna dicho elemento.

Listing 12: clase PriorityQueueHeap

```
1       public T front() throws ExceptionIsEmpty {
2        return heap.first().getElement();
3    }
```

- El método back() se utiliza para obtener el elemento de menor prioridad en la cola de prioridad sin eliminarlo. Retorna el elemento de menor prioridad.
- Dentro del método, se invoca el método last() del Heap para obtener el elemento de menor prioridad. Luego, se utiliza el método getElement() en el objeto PriorityElement retornado para obtener el elemento subyacente. Finalmente, se retorna dicho elemento.

Listing 13: clase PriorityQueueHeap

```
1    public T back() throws ExceptionIsEmpty {
2        return heap.last().getElement();
3    }
```

- La clase PriorityElement es una clase interna privada en la clase PriorityQueueHeap. Representa un elemento en la cola de prioridad, que consta de un elemento genérico y su respectiva prioridad.
- La clase PriorityElement representa un elemento en la cola de prioridad, con su respectivo elemento genérico y prioridad.

Listing 14: clase PriorityQueueHeap

```
1        private class PriorityElement<E extends Comparable<E>> implements
             Comparable<PriorityElement<E>> {
2        private E element;
3        private int priority;
4
5        public PriorityElement(E element, int priority) {
6            this.element = element;
7            this.priority = priority;
8        }
9
10       public E getElement() {
11           return element;
12       }
13
14       public int getPriority() {
15           return priority;
16       }
17
18       public int compareTo(PriorityElement<E> other) {
19           return Integer.compare(this.priority, other.getPriority());
20       }
21
22       public String toString() {
23           return this.element.toString();
24       }
25   }
26
27
28   public String toString() {
29       return heap.toString();
30   }
```

2.3. CLASE SkipList

- La clase SkipList es una implementación de la estructura de datos Skip List. Esta estructura de datos es similar a una lista enlazada, pero permite un acceso rápido y eficiente a los elementos mediante el uso de múltiples niveles.

- en otras palabras es una implementación de la Skip List que utiliza nodos enlazados para almacenar los elementos. Proporciona una forma eficiente de buscar, insertar y eliminar elementos en la estructura de datos. Los niveles de los nodos se generan de manera aleatoria para mejorar el rendimiento de las operaciones en la Skip List.

Listing 15: clase SkipList

```
public class SkipList<T extends Comparable<T>> {
        private static final int MAX_LEVEL = 32; // N mero m ximo de
            niveles en la Skip List
        private Node<T> head; // Nodo de la cabeza de la Skip List
        private int level; // Nivel actual de la Skip List
        private Random random; // Generador de n meros aleatorios
```

- El constructor crea una Skip List vacía con un nodo de cabeza y establece el nivel actual en 0, preparándola para su uso posterior.

Listing 16: clase SkipList

```
 public SkipList() {
    head = new Node<>(null, MAX_LEVEL);
    level = 0;
    random = new Random();
}
```

- El método insert genera un nivel aleatorio para el nuevo nodo, encuentra el lugar adecuado para insertarlo en la lista y realiza las conexiones necesarias para mantener la estructura de la Skip List

Listing 17: clase SkipList

```
    public void insert(T value) {
    int newLevel = randomLevel();
    if (newLevel > level) {
        level = newLevel;
    }

    Node<T> newNode = new Node<>(value, newLevel);
    Node<T> current = head;

    for (int i = level; i >= 0; i--) {
        while (current.forward[i] != null && current.forward[i].value.
           compareTo(value) < 0) {
            current = current.forward[i];
        }

        if (i <= newLevel) {
            newNode.forward[i] = current.forward[i];
            current.forward[i] = newNode;
        }
    }
}
```

- El método remove busca y elimina el nodo que contiene el valor especificado, actualizando los enlaces de los nodos adyacentes para mantener la estructura de la Skip List.

Listing 18: clase SkipList

```
1         public boolean remove(T value) {
2         Node<T> current = head;
3         boolean removed = false;
4
5         for (int i = level; i >= 0; i--) {
6             while (current.forward[i] != null && current.forward[i].value.
                  compareTo(value) < 0) {
7                 current = current.forward[i];
8             }
9
10            if (current.forward[i] != null && current.forward[i].value.
                  equals(value)) {
11                current.forward[i] = current.forward[i].forward[i];
12                removed = true;
13            }
14        }
15
16        return removed;
17    }
```

- el método contains recorre la Skip List para verificar si un valor específico está presente o no.

Listing 19: clase SkipList

```
1         public boolean contains(T value) {
2         Node<T> current = head;
3
4         for (int i = level; i >= 0; i--) {
5             while (current.forward[i] != null && current.forward[i].value.
                  compareTo(value) < 0) {
6                 current = current.forward[i];
7             }
8
9             if (current.forward[i] != null && current.forward[i].value.
                  equals(value)) {
10                return true;
11            }
12        }
13
14        return false;
15    }
```

- El método randomLevel() genera un nivel aleatorio para un nuevo nodo basado en la probabilidad de 0.5. Esto se utiliza para determinar la altura del nodo en la estructura de la Skip List.

Listing 20: Clase SkipList

```
1         private int randomLevel() {
2         int level = 0;
3         while (random.nextDouble() < 0.5 && level < MAX_LEVEL) {
4             level++;
5         }
6         return level;
7     }
```

```
8
9       private static class Node<T extends Comparable<T>> {
10          private T value;
11          private Node<T>[] forward;
12
13          @SuppressWarnings("unchecked")
14          public Node(T value, int level) {
15              this.value = value;
16              forward = new Node[level + 1];
17          }
18      }
```

- El método toString() genera una representación en forma de cadena de la Skip List, mostrando los valores almacenados en cada nivel de la estructura. Esto facilita la visualización y depuración de la Skip List.

Listing 21: Clase SkipList

```
1           public String toString() {
2           String str = "SkipList{";
3           Node<T> current = head.forward[0];
4           while (current != null) {
5               str += "\n";
6               Node<T> levelStart = current;
7               str += "[ ";
8               while (current != null) {
9                   str += current.value + " ";
10                  current = current.forward[0];
11              }
12              str += "]";
13              current = levelStart.forward[0];
14          }
15          str += "\n}";
16          return str.toString();
17      }
```

2.4. CLASE TestPriorityQueueHeap

- Esta clase muestra cómo utilizar las estructuras de datos PriorityQueueHeap y SkipList para agregar, eliminar y obtener elementos de una cola de prioridad y una Skip List, respectivamente. También muestra cómo imprimir el estado actual de las estructuras utilizando el método toString().

Listing 22: Clase TestPriorityQueueHeap

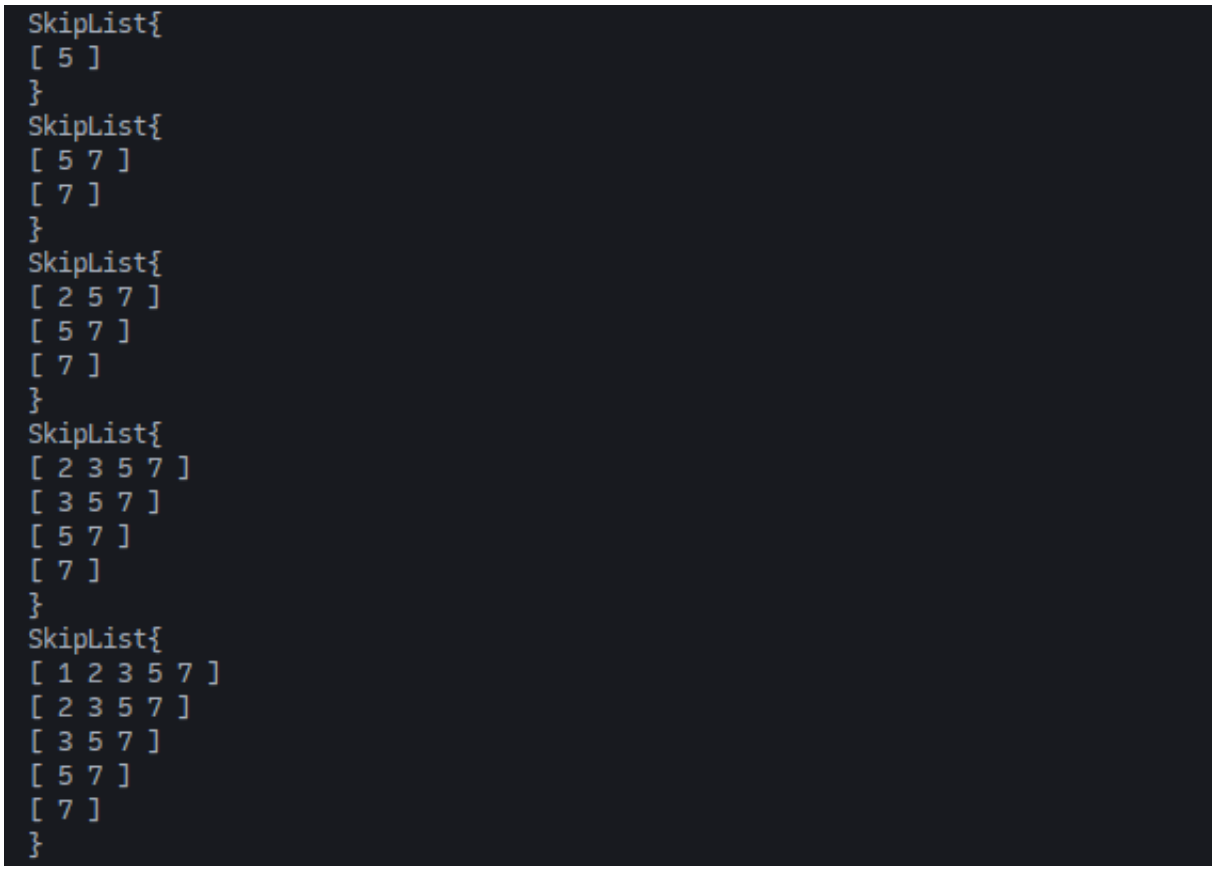
```
1       public class TestPriorityQueueHeap {
2       public static void main(String[] args) throws ExceptionIsEmpty {
3           PriorityQueueHeap<String> queue = new PriorityQueueHeap<String>();
4
5           // Agregamos los elementos y su prioridad
6           // 13, 14, 16, 24, 21, 19, 68, 65, 26, 32, 31
7           System.out.println("Agregando elementos");
8           queue.enqueue("13", 13);
9           System.out.println(queue);
10          queue.enqueue("14", 14);
11          System.out.println(queue);
```

```
12          queue.enqueue("16", 16);
13          System.out.println(queue);
14          queue.enqueue("24", 24);
15          System.out.println(queue);
16          queue.enqueue("21", 21);
17          System.out.println(queue);
18          queue.enqueue("19", 19);
19          System.out.println(queue);
20          queue.enqueue("68", 68);
21          System.out.println(queue);
22          queue.enqueue("65", 65);
23          System.out.println(queue);
24          queue.enqueue("26", 26);
25          System.out.println(queue);
26          queue.enqueue("32", 32);
27          System.out.println(queue);
28          queue.enqueue("31", 31);
29          System.out.println(queue);
30
31          // Prueba de elemento con m s y menos prioridad
32          System.out.println("\nMuestra el primer y  ltimo  elemento de la
                cola de prioridad");
33          System.out.println(queue.front() + ", " + queue.back() + "\n" );
34
35          // Quitamos los elementos seg n su prioridad
36          System.out.println("Quitando elementos de la cola de prioridad");
37          for (int i = 0; i < 11; i++) {
38              queue.dequeue();
39              System.out.println(queue);
40          }
41
42          SkipList<Integer> sl = new SkipList<Integer>();
43          sl.insert(5);
44          System.out.println(sl);
45          sl.insert(7);
46          System.out.println(sl);
47          sl.insert(2);
48          System.out.println(sl);
49          sl.insert(3);
50          System.out.println(sl);
51          sl.insert(1);
52          System.out.println(sl);
53
54          System.out.println("Quitando elementos de la cola de prioridad");
55          sl.remove(7);
56          System.out.println(sl);
57          sl.remove(5);
58          System.out.println(sl);
59          sl.remove(2);
60          System.out.println(sl);
61          sl.remove(3);
62          System.out.println(sl);
63          sl.remove(1);
64          System.out.println(sl);
65      }
```

```
66 }
```

2.5. EJECUCION:

```
Quitando elementos de la cola de prioridad
SkipList{
[ 1 2 3 5 ]
[ 2 3 5 ]
[ 3 5 ]
[ 5 ]
}
SkipList{
[ 1 2 3 ]
[ 2 3 ]
[ 3 ]
}
SkipList{
[ 1 3 ]
[ 3 ]
}
SkipList{
[ 1 ]
}
SkipList{
}
```



3. REFERENCIAS:

Enlace repositorio en GitHub: `https://github.com/YoelLeon/EDA_Practica_No_3_Heaps.git`