UNIVERSIDAD CATÓLICA DE SANTA MARÍA
ESCUELA PROFESIONAL DE INGENIERÍA DE SISTEMAS

LABORATORIO 06:

STACK & QUEUE

I
OBJETIVOS

- Utilizar los conceptos de genericidad para implementar los TAD pila y cola genéricas.
- Definir la interface genérica de una pila y de una cola
- Implementar el TAD pila y el TAD cola a través de: arreglos y de listas dinámicas
- Definir e implementar de diversas formas una cola de prioridad.
- Utilizar los TAD pila, cola y cola de prioridad en la resolución de diversos problemas.

II
TEMAS A TRATAR

- Definición y operaciones de una Pila (*Stack*)
- Definición y operaciones de una Cola (*Queue*)
- Implementaciones del TAD pila y del TAD cola
- Definición y operaciones de una Cola de prioridad (*PriorityQueue*)
- Implementación de la cola de prioridad

III
DURACIÓN DE LA PRÁCTICA

❖ Dos sesiones (4 horas)

IV
RECURSOS

❖ Equipos de cómputo.
❖ Lenguaje de programación Java y pseudocódigo.
❖ IDE de desarrollo. Eclipse, VCode.
❖ Herramientas de seguimiento de versiones: Git & GitHub.

V
MARCO TEORICO

Definición y operaciones de una Pila (*Stack*)

Una pila es una lista lineal de elementos homogéneos, en donde las inserciones, eliminaciones y, generalmente, todos los accesos son realizados por un extremo de la lista, llamado el **tope** o **cima** de la pila. Los elementos son colocados uno sobre otro, por ejemplo, en una pila de platos, solo puede sacarse el segundo si es que el primero (el del **tope**) ha sido sacado primero. El elemento

insertado menos recientemente está en el fondo de la pila. Las pilas admiten dos operaciones principales:

- **push(x)** que agrega un elemento x a la colección
- **pop()** que elimina el último elemento que se agregó

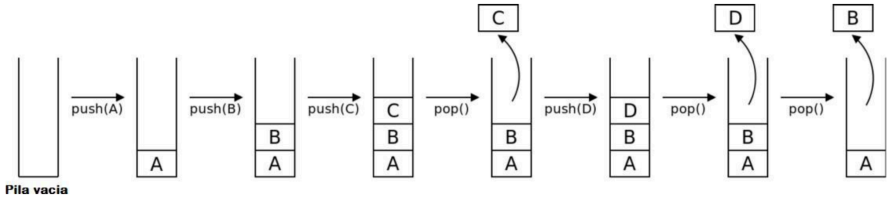


Figura 6.1. Operaciones fundamentales de un *Stack*

Debido a que, la eliminación e inserción de elementos se produce solamente en uno de los extremos, la pila también es llamada **estructura de datos o listas LIFO** (*Last In First Out*).

a) Donde se aplican los stacks?

Aplicaciones en estructuras anidadas:

- Cuando es necesario explorar en conjuntos de datos y guardar una lista de cosas a realizar posteriormente.
- El control de secuencias de llamadas a subprogramas.
- La sintaxis de expresiones aritméticas.

Las pilas ocurren en estructuras de naturaleza recursiva (como árboles). Ellas son utilizadas para implementar la recursividad.

b) Operaciones en los stacks:

Las operaciones deben obedecer a la manera como este tipo de lista accede a los datos (LIFO). Entre las principals operaciones están las siguientes:

- push(x) : inserta en la pila el elemento 'x'. (*apilar*)
- pop(): elimina de la pila el elemento que está en la cima o el tope de la pila. (*desapilar*)
- top(): recupera el elemento del tope o cima de la pila. (*tope*)
- destroyStack(): elimina todos los elementos, dejando la pila vacía.
- isEmpty(): verifica si la pila está vacía.
- isFull(): verifica si la pila está llena. Se usa cuando la pila esta implementa sobre una estructura de datos estática.

Definición y operaciones de una Cola (Queue)

Una cola es una lista lineal en la que los elementos son agregados en uno de los extremos de la lista llamado la **cola** (*rear / back / last*) y eliminados por el otro extremo de la lista llamado el **frente** (*front / first*) de la cola. El modelo intuitivo de una cola es de una fila de espera de personas, por ello estas estructuras se les llaman **listas FIFO** (*First In First Out)*. Las colas admiten dos operaciones principales:

- enqueue(x) que agrega un elemento x a la colección
- dequeue() que elimina el elemento que se agregó por más tiempo

Figura 6.2. Operaciones fundamentales de un *Queue*

a) ¿Cómo se utilizan las queues?

Son utilizadas cuando deseamos procesar items de acuerdo con el orden "***primero que llega, primero en ser atendido***".

Los sistemas operativos, utilizan colas para regular el orden en el cual las tareas deben recibir procesamiento y para ver que recursos deben ser atribuidos a los procesos.

b) Operaciones en las queues

Las operaciones que se aplican a las colas obedecen a la forma como esta lista accede y/o manipula la información almacenada en ella, las cuales son:

- enqueue(x): agrega el elemento 'x' al final de la cola. La cola debe existir y no estar llena. (*encolar*).
- dequeue(): elimina el elemento que se encuentra al frente (*front*) de la cola. La cola debe existir y no estar vacía. (*desencolar*).
- destroyQueue(): elimina los elementos de la cola dejándola vacía.
- isEmpty(): verifica si la cola está o no vacía.
- isFull(): verifica si la cola está llena o no. Se usa cuando la cola está implementada sobre una estructura estática.
- front(): retorna el elemento que se encuentra al frente de la cola (primer elemento)
- back(): retorna el último elemento de la cola.

Implementaciones de los TAD pila y cola

Dado que tanto la pila como la cola son estructuras cuyos elementos se organizan de forma lineal, pueden estar implementadas usando un arreglo o una lista enlazada como estructura de datos que almacene los elementos que éstas guardarán.

a) TAD Pila - *Stack*

En el caso de una pila su implementación obedecerá al siguiente diagrama:

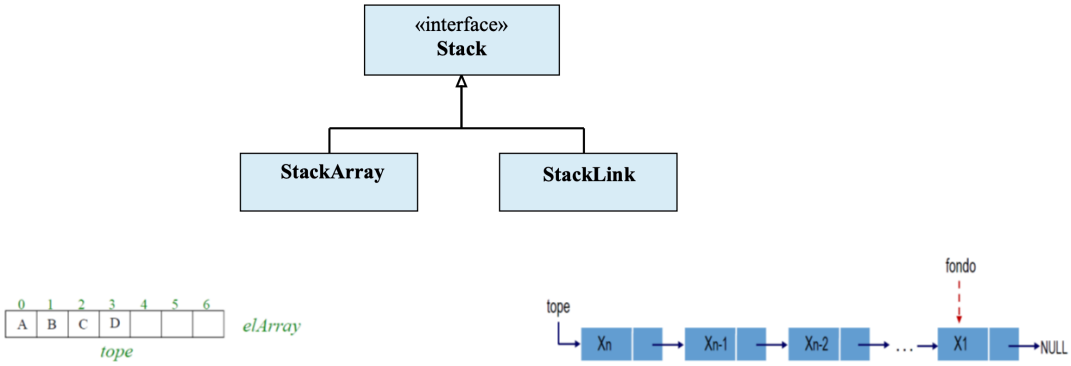


Figura 6.3. Implementaciones de un *Stack*

b) TAD Cola – *Queue*

Y en el caso de una cola, también se puede usar un arreglo o una lista enlazada en su implementación tal como se observa en el siguiente diagrama:

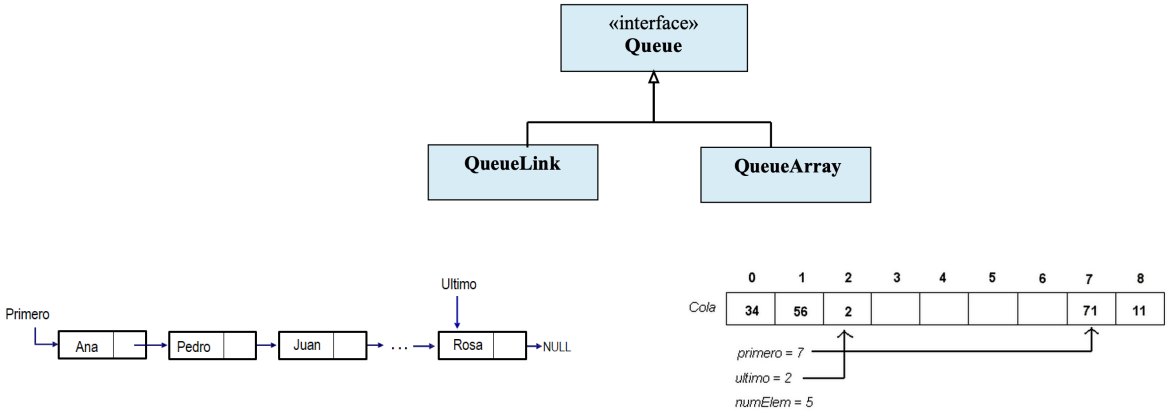


Figura 6.4. Implementaciones de una *Queue*

Definición y operaciones de la Cola de prioridad

Una **Cola de Prioridad** es una cola en donde cada elemento tiene asociado cierta información denominada ***prioridad*** que determina el orden de acceso a cada uno de los elementos. Una vez insertados los elementos a una cola de prioridad, la salida o procesamiento de éstos no se basa necesariamente en el orden de llegada, sino en el orden definido por la prioridad que tiene cada uno de los elementos. En una cola, por el contrario, se acceden o procesar los datos según el orden estricto en el que se incorporan a la cola.

Las operaciones que se consideran para una cola de prioridad son las mismas que para una cola FIFO, con la única diferencia que, el orden en el que se sacan los elementos (se procesan los elementos) está en función de la prioridad de cada uno de éstos.

Dependiendo de la estructura de datos que se utilice para su implementación, la realización de estas operaciones requerirá o no mantener **ordenada** la colección de elementos en la cola.

a) Operaciones en las colas de prioridad

Las operaciones que se aplican a las colas de prioridad se diferencian con las de una cola principalmente por que ahora los elementos que se encuentran en la cola se acceden en función de su prioridad. La forma como estás operaciones serán implementadas dependen esencialmente de la estructura de datos que se este utilizando. En general, las operaciones son:

- enqueue(x, p): agrega el elemento ‘x’ con prioridad ‘p’.
- dequeue(): elimina el elemento de mayor prioridad. La cola debe existir y no estar vacía.
- destroyQueue(): elimina los elementos de la cola dejándola vacía.
- isEmpty(): verifica si la cola está o no vacía.
- isFull(): verifica si la cola está llena o no. Se usa cuando la cola está implementada sobre una estructura estática.
- front(): retorna el elemento que tiene más alta prioridad.
- back(): retorna el elemento de menor prioridad.

Implementación de colas de prioridad

a) Por medio de una única lista

Al implementar la cola de prioridad por medio de listas (pueden ser arreglos o lista enlazadas), cada elemento tiene además la prioridad. Por ejemplo, si fuera una lista enlazada cada nodo tendría la estructura que se muestra en la figura 6.5.

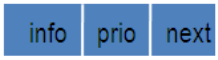


Figura 6.5. Estructura del nodo de una cola de prioridad

Cada elemento de la cola de prioridad forma un nodo de la lista enlazada, y ésta podría estar ordenada o no por el campo que determina la prioridad.

Si la lista se mantiene ordenada por la prioridad (ver figura 6.6), la operación de inserción de un nuevo elemento hay que hacerla siguiendo este criterio:

- La posición de inserción es tal que la lista debe seguir ordenada.
- A igualdad de prioridad, el elemento se añade como último en el grupo de nodos de igual prioridad.

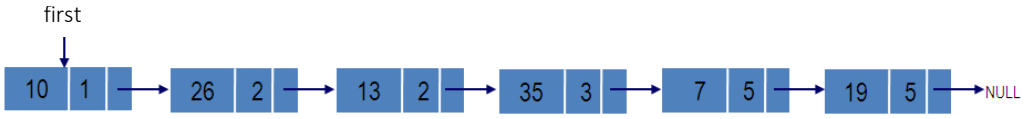


Figura 6.6. Una cola de prioridad en una única lista ordenada por la prioridad

Esta forma de mantener la lista presenta como principal ventaja que es inmediato determinar el siguiente nodo a procesar: **siempre será el primero de la lista**.

Pero si la lista no está ordenada por el campo de prioridad, entonces la inserción puede ser una operación muy eficiente, sin embargo, el procesamiento de un elemento implicaría el buscar el elemento que tenga la mayor prioridad en la lista, y esto puede ser un costo alto computacional. Ver figura 6.7.

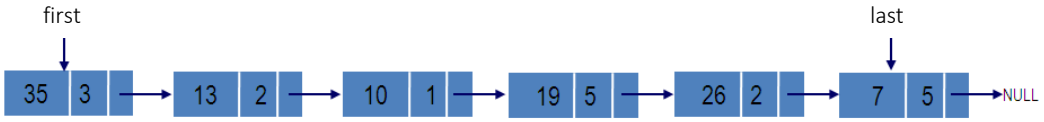


Figura 6.7. Una cola de prioridad en una única lista de tipo FIFO

b) Por medio de varias lista

En este caso, se tiene una lista para cada valor diferente de prioridad. Se requiere de otra estructura que permita el acceso a las diferentes colas.

Los elementos de la misma prioridad se encolan considerando el tiempo de llegada.

El acceso o procesamiento de los elementos se realiza teniendo en cuenta lo siguiente:

- Primero se atienden los elementos de la cola de prioridad mayor.
- Si en la cola de prioridad mayor no hay elementos, se atiende a los elementos de la cola de la siguiente prioridad, y así sucesivamente.
- Cada lista que implementa una cola de una determinada prioridad tiene un comportamiento igual al de una cola normal.

Observe la figura 6.8 donde se muestra un arreglo que tiene como información los objetos cola. Una para cada prioridad, que está relacionada con el índice del arreglo (en este caso las colas son listas enlazadas).

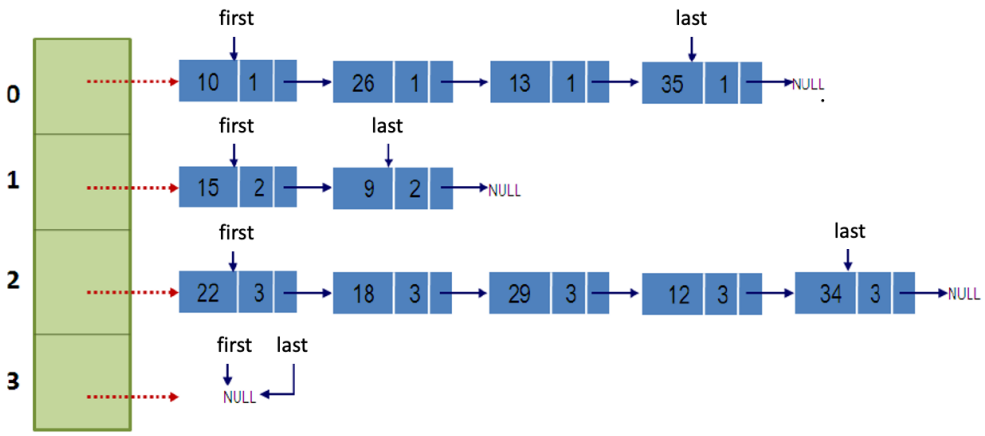


Figura 6.8. Una cola de prioridad en varias lista de tipo FIFO

VI ACTIVIDADES

Para las actividades tenga en cuenta el diagrama de la sección anterior.

Actividad 1: Implementando una Pila a través de un arreglo

01. En un nuevo proyecto, cree el paquete actividad1 y en este paquete, defina la interface genérica **Stack** en la que declare las operaciones de una pila. Como la operación *isFull*() está condicionada a utilizar una estructura estática, no la incluya en la interface.

```
public interface Stack<E> {
   void push(E x);
   E pop() throws ExceptionIsEmpty;
   E top() throws ExceptionIsEmpty;
   boolean isEmpty();
}
```

02. Observe que en la definición de algunos métodos se hace referencia a la excepción ExceptionIsEmpty, por consiguiente, debe crear la clase que implemente esta excepción.

03. En el paquete *actividad1*, agregue la clase StackArray que implemente la interface *Stack* y por ende los métodos de ésta.

```
class StackArray<E> implements Stack<E> {
   private E[] array;
   private int tope;

   public StackArray(int n){
      this.array = (E[])new Object[n];
      tope = -1;
   }

   public void push(E x){
      // include here your code
   }
```

```
    public E pop() throws ExceptionIsEmpty {
       // include here your code
    }

    public E top() throws ExceptionIsEmpty {
       // include here your code
    }

    public boolean isEmpty() {
       return this.tope == -1;
    }

    public boolean isFull (){
       // include here your code
    }

    //The elements must be included in the chain from the one at the top
    //to the one at the bottom of the stack.
    public String toString(){
       // include here your code
    }
}
```

04. En el paquete por defecto del proyecto, agregue la clase Test donde se harán las pruebas de lo realizado. Para esto cree objetos del tipo *StackArray* en variables del tipo *Stack*, que manipule datos de diversos tipos en los cuales aplique las diferentes operaciones y verifique los resultados obtenidos.

Actividad 2: Implementando una cola a través de una lista enlazada

05. En el mismo proyecto, cree el paquete actividad2 y en este paquete, defina la interface genérica **Queue** en la que declare las operaciones de una cola. Como la operación *isFull*() está condicionada a utilizar una estructura estática, no la incluya en la interface.

```
public interface Queue<E> {
   void enqueue(E x);
   E dequeue() throws ExceptionIsEmpty;
   E front() throws ExceptionIsEmpty;
   E back() throws ExceptionIsEmpty;
   boolean isEmpty();
}
```

Observe que en la definición de algunos métodos se hace referencia a la misma excepción ExceptionIsEmpty, por tanto, debe importar el paquete donde se encuentra definida la clase correspondiente.

06. En el paquete *actividad2*, agregue la clase QueueLink que implemente la interface *Queue* y por ende los métodos de ésta.
Además, debe copiar la clase Node (con la que trabajo la lista enlazada en el laboratorio 06 – *Node.java*) al paquete *actividad2*.

```
class QueueLink<E> implements Queue<E> {
   private Node<E> first;
   private Node<E> last;

   public QueueLink(){
      this.first = null;
      this.last = null;
   }

   public void enqueue(E x){
      Node<E> aux = new Node<E>(x);
      if (this.isEmpty()) {
         this.first = aux;
      else
         this.last.setNext(aux);
```

```
        this.last = aux;
    }

    public E dequeue() throws ExceptionIsEmpty {
        // include here your code
    }

    public E back() throws ExceptionIsEmpty {
        // include here your code
    }

    public E front() throws ExceptionIsEmpty {
        // include here your code }

    public boolean isEmpty() {
        // include here your code
    }

    //The elements must be included in the chain from the one at the front
    //to the one at the back of the queue.
    public String toString(){
        // include here your code
    }
}
```

07. En el paquete por defecto del proyecto, agregue la clase Test donde se harán las pruebas de lo realizado. Para esto cree objetos del tipo *QueueLink* en variables del tipo *Queue*, que manipule datos de diversos tipos en los cuales aplique las diferentes operaciones y verifique los resultados obtenidos.

Actividad 3: Implementando una cola de prioridad a través de una lista enlazada ordenada por la prioridad

08. En el mismo proyecto, cree el paquete actividad3 y en este paquete, defina la interface genérica **PriorityQueue** en la que declare las operaciones de una cola de prioridad. Recuerde que son las mismas de una cola, con la diferencia que los elementos que están en la cola se procesan en función de su prioridad.

```
public interface PriorityQueue<E, N> {
    void enqueue(E x, N pr);
    E dequeue() throws ExceptionIsEmpty;
    E front() throws ExceptionIsEmpty;
    E back() throws ExceptionIsEmpty;
    boolean isEmpty();
}
```

Puesto que también se hace referencia a ExceptionIsEmpty, importe el paquete donde se encuentra definida la clase correspondiente.

09. En el paquete *actividad3*, agregue la clase PriorityQueueLinkSort que implemente la interface *PriorityQueue* y todos los métodos definidos en ésta.
Además, debe copiar la clase Node nuevamente al paquete *actividad3*. Como vamos a utilizar la clase *Node* que tiene un solo atributo data que encapsula la información del elemento que se esta almacenando en un nodo determinado de la lista, y como ahora cada elemento tiene además una prioridad, se necesita encapsular el *dato* y la *prioridad* en un solo objeto que se almacenará en un *Node* de la lista. Para esto se creará la clase EntryNode en la clase *PrioriryQueueLinkSort*.

```
class PriorityQueueLinkSort<E,N> implements PriorityQueue<E,N> {

   class EntryNode{
      E data;
      N priority;
      EntryNode(E data, N priority){
         this.data = data;
         this.priority = priority;
      }
   }
   private Node<EntryNode> first;
   private Node<EntryNode> last;

   public PriorityQueueLinkSort (){
      this.first = null;
      this.last = null;
   }

   public void enqueue(E x, N pr){
      // The list must be ordered by the priority of the elements.
      // The higher the priority, the element is further to the front.
   }

   public E dequeue() throws ExceptionIsEmpty {
      if (isEmpty())
         throw new ExceptionIsEmpty("Cannot remove from an empty queue");
      E aux = this.first.getData().data;
      this.first = this.first.getNext();
      if (this.first == null)
         this.last = null;
      return aux;
   }

   public E back() throws ExceptionIsEmpty {
      // include here your code
   }

   public E front() throws ExceptionIsEmpty {
      // include here your code }

   public boolean isEmpty() {
      // include here your code
   }

   // Elements must be included in the string as they are located in the list
   public String toString(){
      // include here your code
   }
}
```

10. En el paquete por defecto del proyecto, agregue la clase Test donde se harán las pruebas de lo realizado. Para esto cree objetos del tipo *PriorityQueueLinkSort* en variables del tipo *PriorityQueue*, que manipule datos de diversos tipos en los cuales aplique las diferentes operaciones y verifique los resultados obtenidos.

VII
EJERCICIOS

01. Implementación de una pila utilizando una lista enlazada.

a) En el proyecto cree el paquete ejercicio1, y defina la clase **StackLink** que implemente la interface *Stack* y los métodos que en está fueron declarados (en la actividad 1), pero ahora tenga en cuenta que está usted usando una lista enlazada como estructura de datos. Incorpore la clase *Node* de la lista enlace o importe el paquete donde tiene dicha clase. Haga lo propio con la clase donde esta *ExcepcionIsEmpty*.

b) En la clase *Test* del paquete por defecto del proyecto, cree objetos del tipo *StackLink* en variables del tipo *Stack*, que manipule datos de diversos tipos en los cuales aplique las diferentes operaciones y verifique los resultados obtenidos.
Observe que el hecho de cambiar la implementación, el usuario no debería de verse afectado puesto que él está usando la misma interface.

02. Implementación de una cola utilizando un arreglo.

a) En el proyecto cree el paquete ejercicio2, y defina la clase **QueueArray** que implemente la interface *Queue* y los métodos que en está fueron declarados (en la actividad 2), pero ahora tenga en cuenta que está usted usando un arreglo de un tamaño determinado. Cuando se instancie esta clase, el constructor recibirá la cantidad de elementos que tendrá como máximo el arreglo y en función de este valor se le asignará el tamaño correspondiente.
Incorpore la clase donde esta *ExcepcionIsEmpty*.
Así mismo, dado que es un arreglo lo que se está manipulando, los atributos *first* y *last* ahora son posiciones. Estos atributos deben ser actualizados en las operaciones de inserción y eliminación de la cola a través de la aritmética modular.

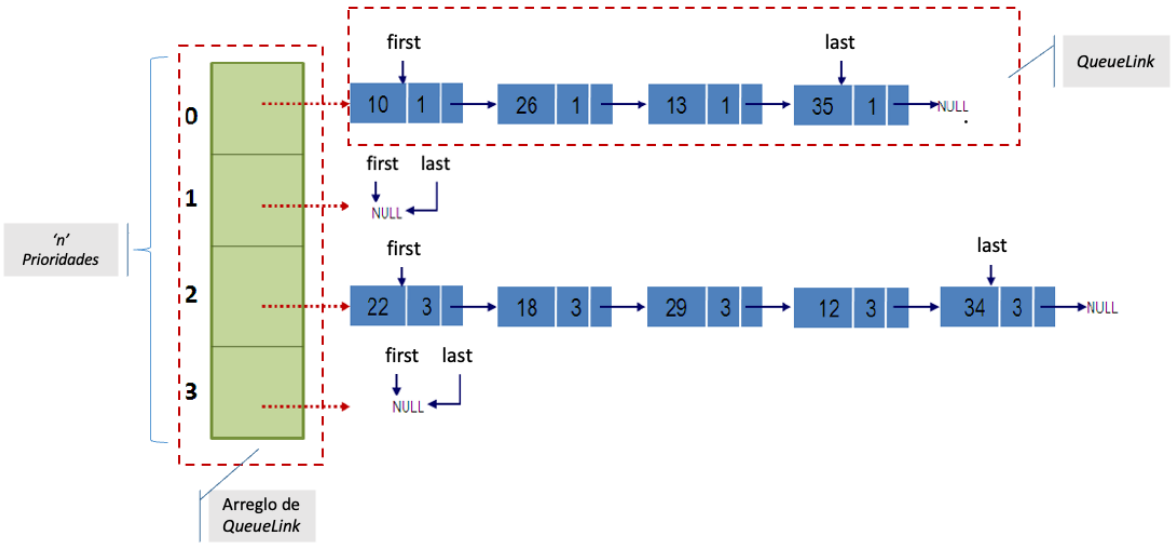
b) En la clase *Test* del paquete por defecto del proyecto, cree objetos del tipo *QueueArray* en variables del tipo *Queue*, que manipule datos de diversos tipos en los cuales aplique las diferentes operaciones y verifique los resultados obtenidos.
Observe que el hecho de cambiar la implementación de la cola, el usuario no debería de verse afectado puesto que él está usando la misma interface.

03. Implementación de una cola de prioridad utilizando varias colas basadas en listas enlazadas.

a) En el proyecto cree el paquete ejercicio3, y defina la clase **PriorityQueueLinked** que implemente la interface *PriorityQueue* y los métodos que en está fueron declarados (en la actividad 3), pero ahora tenga en cuenta que está usted usando varias colas (una por cada prioridad) vinculas a un arreglo de un tamaño dado, el mismo que representa la cantidad de prioridad con las que se trabajará. Cuando se instancie esta clase, el constructor recibirá la cantidad de prioridades y en función de este valor se le asignará el tamaño correspondiente al arreglo de colas.

Incorpore la clase donde esta *ExcepcionIsEmpty*.

Las colas son del tipo *QueueLink* que ya fue implementada en el ejercicio 2, y no debería de requerir ningún ajuste, puesto que en su implementación se ha considerado la característica genérica del dato que almacenan los elementos de la cola, por tanto, debería de adaptarse a cualquier tipo de dato. Entonces, debe enfocarse en la implementación propiamente dicha de los métodos de la clase *PriorityQueueLinked***.** El siguiente esquema le muestra los tipos de datos de los diferentes objetos a considerar en la solución.



b) En la clase *Test* del paquete por defecto del proyecto, cree objetos del tipo *PriorityQueueLinked* en variables del tipo *PriorityQueue*, que manipule datos de diversos tipos en los cuales aplique las diferentes operaciones y verifique los resultados obtenidos. Observe que el hecho de cambiar la implementación de la cola de prioridad, el usuario no debería de verse afectado puesto que él está usando la misma interface.

04. Aplicación: Problema de los corchetes

El problema de los corchetes anidados es un problema que determina si una secuencia de corchetes está correctamente anidada. Una secuencia de corchetes "S" se considera anidada correctamente si se cumple alguna de las siguientes condiciones:

- "S" está vacío
- "S" tiene la forma (U) o [U] o {U} donde U es una cadena correctamente anidada
- "S" tiene la forma VW donde V y W son cadenas correctamente anidadas

Por ejemplo, la cadena "()()[()]" está correctamente anidada pero "[(()]" no.

La función llamada "*symbol balancing*" toma como entrada una cadena S que es una secuencia de corchetes y devuelve verdadero si S está anidado y falso en caso contrario. Esta función es un método estático de la clase *Aplication.*

Entrada	Salida
"()()()[()]()"	true
"((()))[]"	true
"([])[]("	false
"([{}]}"	false
"["	false
"[][][]{{{}}}"	true

IMPORTANTE: Es obligatorio hacer uso de la clase " **StackLink**".

VIII
PARA INVESTIGAR

1. Investigue que es una bicola.
2. ¿Qué formas de implementación se puede tener para una bicola?
3. ¿Cómo se usan las pilas con la notación infija y posfija?

IX
BIBLIOGRAFÍA Y REFERENCIAS

- Weiss M., *Data Structures & Problem Solving Using Java*, Addison-Wesley, 2010. Chapter 16.
- Cutajar J. *Beginning Java Data Structures and Algorithms*, Packt Publishing, 2018. Chapter 2.
- Goodrich M., Tamassia R. and Goldwasser M., *Data Structures & Algorithms*, Wiley, 2014. Chapter 6
- The JavaTM Tutorials - https://docs.oracle.com/javase/tutorial/
- https://www.tutorialspoint.com/data_structures_algorithms/linked_list_algorith ms.htm

CALIFICACIÓN DEL LABORATORIO

CRITERIOS PARA EVALUAR		Nivel				
		A	B	C	D	NP
R1	Uso de estándares y buenas prácticas de programación	2.0	1.6	1.2	0.7	0
R2	Solución de las actividades a través de la definición y uso adecuado de los TAD Stack y Queue, lo que se evidencia en el registro oportuno de los commits en los repositorios correspondientes, así como de las pruebas realizadas para su verificación.	4.0	3.2	2.4	1.4	0
R3	Solución de los ejercicios a través de la definición y uso adecuado de los TAD Stack y Queue, lo que se evidencia en el registro oportuno de los commits en los repositorios correspondientes, así como de las pruebas realizadas para su verificación.	4.0	3.2	2.4	1.4	0
R4	Informe bien redactado, ordenado, bien detallado de las actividades y ejercicios resueltos, y de acuerdo con las normas establecidas. Mostrar evidencia suficiente de lo efectuado.	4.0	3.2	2.4	1.4	0
R5	Participación y contribución en el desarrollo del laboratorio del estudiante.	4.0	3.2	2.4	1.4	0
R6	Entrega oportuna de las actividades, ejercicios e informe.	2.0	1.6	1.2	0.7	0
	Total	**20**				

A: Excelente (100%)
B: Bueno (80%)
C: Regular (60%)
D: Deficiente (35%)
NP: Muy deficiente o No presenta (0%)