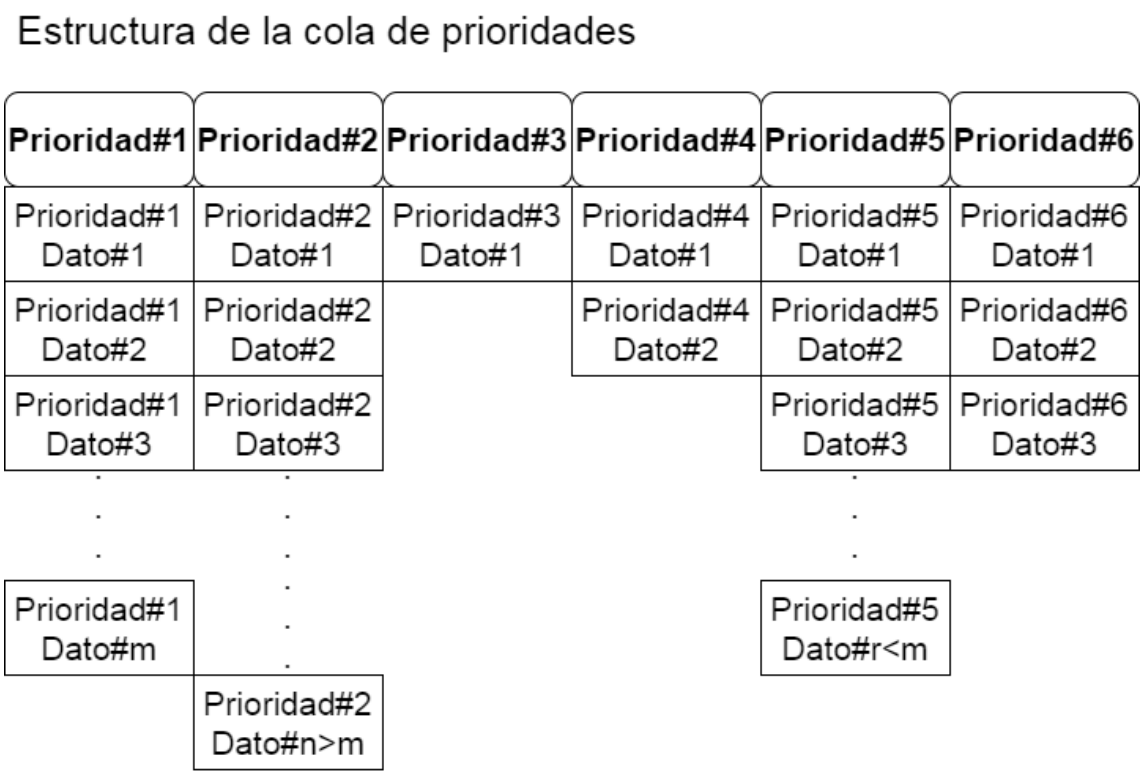
Preguntas teóricas de la sección 2.2

1.- Elección del tipo de secuencia para realizar la implementación.

La forma de la estructura de datos con la implementación `BucketQueue` es:



`SamePriorityQueue` es una estructura de datos en la que los datos se encolan en las prioridades siguiendo un método FIFO; según su orden de llegada. Así el dato#1 de cualquier prioridad es el primero que llego con esa prioridad y el último dato de cada prioridad es el último dato que llegó a esa prioridad.

No existen `SamePriorityQueue` sin datos. `SamePriorityQueue` se crea en la secuencia si llega un dato de una determinada prioridad.

La cola de prioridades es una secuencia de `SamePriorityQueue` que ordena las `SamePriorityQueue` de forma orgánica según las exigencias de los datos que van llegando. Esta estructura de datos accede a todas las `SamePriorityQueue` para confirmar si existe alguna que pueda encolar el dato recién llegado y, si no existe, al encolar el dato deber ser capaz de crear y colocar la `SamePriorityQueue` en el orden que le corresponde enlazándola con las demás `SamePriorityQueue` de la cola.

1.1.- ¿Qué tipo de secuencia sería la adecuada para realizar esta implementación?

La secuencia adecuada para realizar esta implementación es la lista.

1.2.- ¿Por qué?

Aunque tanto la cola como la pila pueden permitir que se examinen los datos que no se encuentran en primer ni en último lugar mediante el uso de iteradores, no tiene métodos públicos que permitan la inserción datos en posiciones intermedias.

La lista puede insertar métodos en posiciones intermedias usando el método `public void insert(int pos, E elem)`

El método `enqueue` de la cola de prioridad examinará las `SamePriorityQueue` que componen la lista y, en caso de que la prioridad del dato a encolar no se encuentre entre las `SamePriorityQueue` de la lista, actuará del siguiente modo:

1. Creará la `SamePriorityQueue` que corresponde a la prioridad del dato.
2. *Insertará en el lugar de la lista que le corresponda.*

De entre las secuencias, la estructura de datos que mejor puede cumplir con el segundo paso usando el método `public void insert(int pos, E elem)` es la lista enlazada.

1.3.- ¿Qué consecuencias tendría el uso de otro tipo de secuencias?

Si en lugar de usar una lista enlazada, se usaran una cola o una pila se debería recorrer toda la estructura de datos para comprobar si la prioridad del dato se corresponde con alguna de las prioridades ya encoladas, en caso de:

I. Encontrase entre la prioridad entre las prioridades de los datos ya encolados, se encolará el dato la prioridad que le corresponda.
II. No encontrarse la prioridad entre las prioridades de los datos ya encolados, se deberá
    1. Crear la prioridad que corresponde a la prioridad del dato.
    2. *Crear una estructura auxiliar que conserve las prioridades que se encuentran en lugares anteriores de la estructura de datos a la prioridad que se creó para el dato a encolar.*
    3. *Encolar o apilar la nueva prioridad que se creó para el dato a encolar.*
    4. *Encolar o apilar las prioridades de la estructura auxiliar en la cola de prioridades con la nueva prioridad ya apilada o encolada hasta recuperar la estructura primitiva con la modificación de la nueva prioridad en el lugar que le corresponde.*

Usar otro tipo de secuencias aumentará innecesariamente la complejidad del algoritmo que implemente el método enqueue.

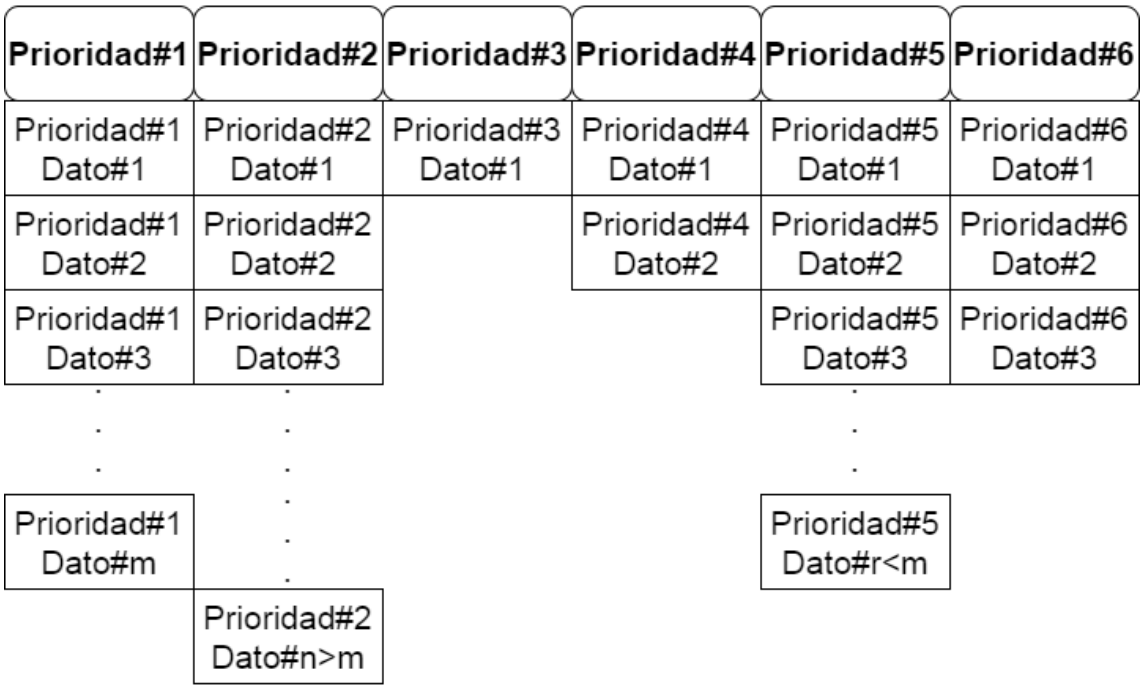
2.- Descripción del funcionamiento de las operaciones.

2.1.- `getFirst`

El método `getFirst` de la cola de prioridad debe devolver el dato de mayor prioridad que llegó en primer lugar de los datos que en ese momento tenga la cola de prioridades.

Según el esquema, el dato: Prioridad#1, Dato#

Estructura de la cola de prioridades



Si la cola de prioridad es una lista enlazada de colas modificadas `SamePriorityQueue` el método `getFirst` de las colas de prioridad,

1. Usando el método `list.get(int pos)` se accede a la `SamePriorityQueue` más prioritaria.
2. El método `SamePriorityQueue.getFirst()` accede al dato que llegó en primer lugar de los encolados con más alta prioridad.

2.2.- `enqueue`

Si la cola de prioridad está vacía:

1. Se crea una estructura de datos `SamePriorityQueue` con la prioridad del dato.
2. Se encola el dato en la estructura `SamePriorityQueue` recién creada.
3. Se inserta la estructura `SamePriorityQueue` recién creada en la primera posición de la lista enlazada.

Si la cola de prioridad no está vacía:

1. Se crea una estructura de datos `SamePriorityQueue` con la prioridad del dato.
2. Se recorre la lista enlazada por medio de un iterador buscando una `SamePriorityQueue` que tenga la misma prioridad que la recién creada.
   a) Si se encuentra un `SamePriorityQueue` con la misma prioridad, el dato se encola `SamePriorityQueue.enqueue(E elem)` en esa `SamePriorityQueue` ya enlazada.
   b) En caso contrario:
      i) Se recorre la lista enlazada buscando una `SamePriorityQueue` con prioridad menor a del dato a encolar y la cola que se creó con la prioridad del dato se inserta en la lista enlazada `list.insert(int pos, E elem)` en el lugar de la `SamePriorityQueue` que tenga la prioridad menor.

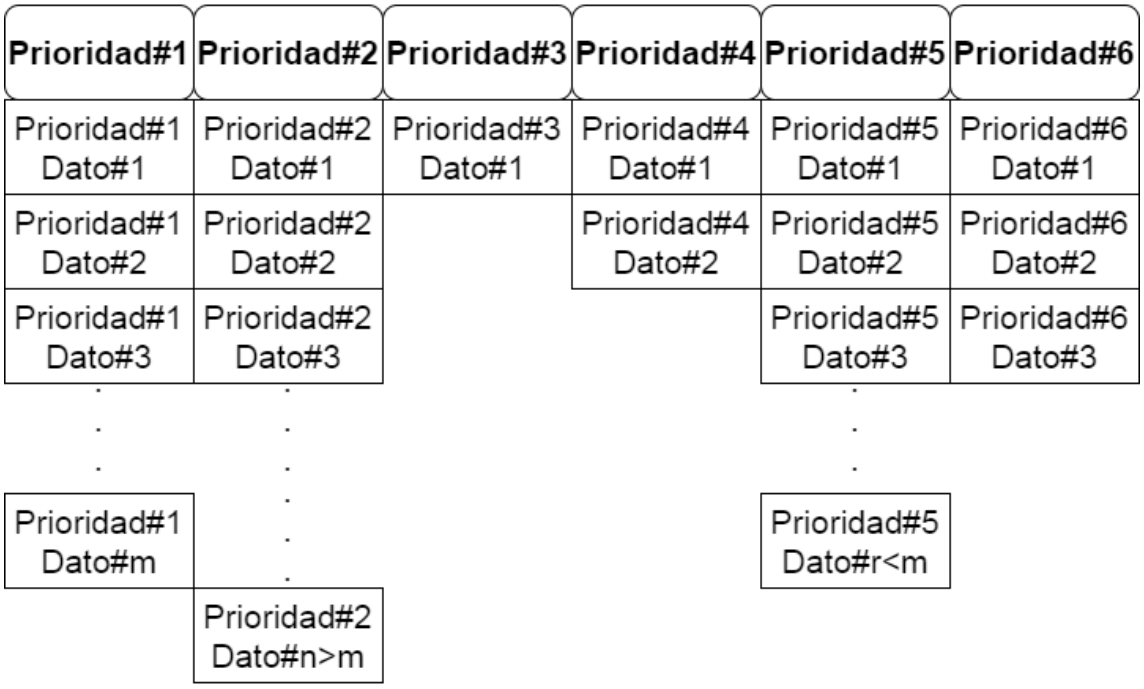
ii) Si no existe ninguna `SamePriorityQueue` ya enlazada con prioridad menor al dato a encolar, se inserta `list.insert(int pos, E elem)`la `SamePriorityQueue` recién creada con la prioridad del dato a encolar en el último lugar de la lista enlazada.

2.3.- enqueue

El método enqueu de la cola de prioridad debe desencolar el dato de mayor prioridad que llegó en primer lugar de los datos que en ese momento tenga la cola de prioridades.

Según el esquema, el dato: Prioridad#1, Dato#

Estructura de la cola de prioridades



Si la cola de prioridad es una lista enlazada de colas modificadas `SamePriorityQueue` el método `enqueue` de las colas de prioridad,

1. Usando el método `list.get(int pos)` se accede a la `SamePriorityQueue` más prioritaria.
2. El método `SamePriorityQueue.enqueue()` desencola el dato que llegó en primer lugar de los encolados con más alta prioridad.
3. Si tras desencolar el dato la `SamePriorityQueue` queda vacía, el eliminada de la lista enlazada `list.remove(1)`.

3.- Descripción del funcionamiento del iterador en esta implementación.

Lista de colas de prioridad				Cola Auxliar
Dato 1, Prioridad 4	Dato 1, Prioridad 3	Dato 1, Prioridad 2	Dato 1, Prioridad 1	14
Dato 2, Prioridad 4		Dato 2, Prioridad 2	Dato 2, Prioridad 1	24
Dato 3, Prioridad 4		Dato 3, Prioridad 2		34
Dato 4, Prioridad 4				44
				13
				12
				22
				32
				11
				21

```java
/* Clase privada que implementa un iterador para la *
 * cola con prioridad basada en secuencia.          */
public class PriorityQueueIterator implements IteratorIF<E> {

  Queue<E> auxQueue;
  IteratorIF<E> itAuxQueue;

  /*Constructor por defecto*/
  protected PriorityQueueIterator(){
      auxQueue = new Queue<>();
      IteratorIF<SamePriorityQueue<E>> itList = list.iterator();
      while(itList.hasNext()) {
          itAuxQueue = itList.getNext().iterator();
          while (itAuxQueue.hasNext()) {
              auxQueue.enqueue(itAuxQueue.getNext());
          }
      }
      itAuxQueue = auxQueue.iterator();
  }
```

`PriorityQueueIterator` crea una cola de datos auxiliar y su iterador.

Para crearla, recorre la lista enlazada de `SamePriorityQueue` encolando los datos en según su orden de llegada y prioridad.

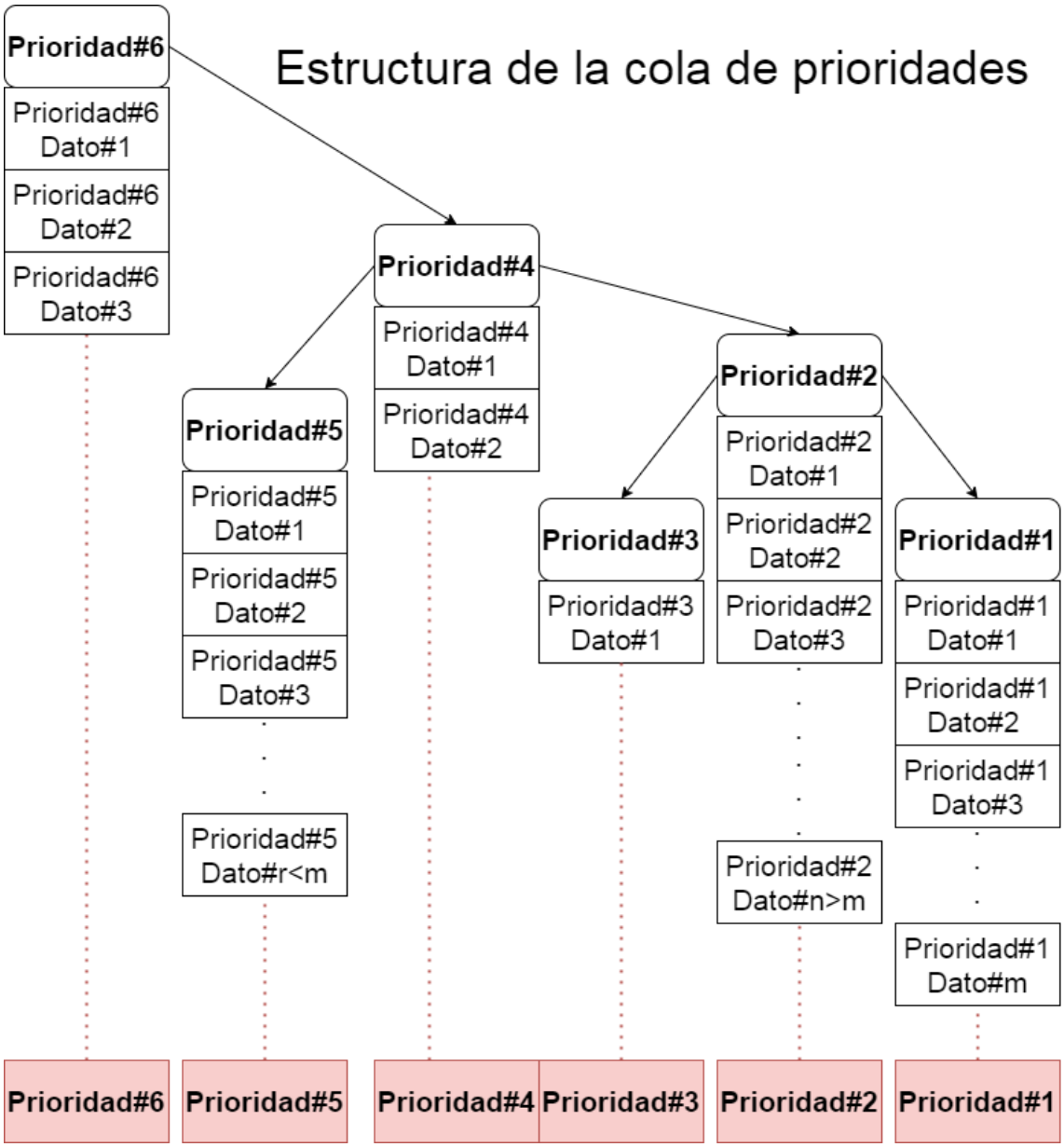
Las operaciones PriorityQueueIterator.`hasNext()`, PriorityQueueIterator.`getNext()`y PriorityQueueIterator.`reset` están delegadas en el iterador de la cola auxiliar.

Aunque el constructor del iterador `BucketQueue.iterator()` recorre todos los elementos de `BucketQueue` para la construcción de la cola auxiliar, el coste asintótico del las operaciones del iterador será lineales O(n) .

Preguntas teóricas de la sección 2.3

1.- Descripción de las operaciones con la estructura de datos.

La estructura de los datos con la implementación `BSTPriorityQueue` es:



`SamePriorityQueue` es una estructura de datos en la que los datos se encolan en las prioridades siguiendo un método FIFO; según su orden de llegada. Así el dato#1 de cualquier prioridad es el primero que llego con esa prioridad y el último dato de cada prioridad es el último dato que llegó a esa prioridad.

No existen `SamePriorityQueue` sin datos. `SamePriorityQueue` se crea en `BSTree` si llega un dato de una determinada prioridad.

La cola de prioridades es un árbol de `SamePriorityQueue` que ordena las `SamePriorityQueue` de forma orgánica según las exigencias de los datos que van llegando. Esta estructura de datos accede a todas las `SamePriorityQueue` para confirmar si existe alguna que pueda encolar el dato recién llegado y, si no existe, al encolar el dato deber ser capaz de crear y colocar la `SamePriorityQueue` en el orden que le corresponde en el contexto del árbol.

1.- Descripción del funcionamiento de las operaciones.

2.1.- `getFirst`

La estructura de datos `BSTPriorityQueue` está ordenada de forma que las `SamePriorityQueue` que se corresponden a prioridades más altas quedan a la izquierda en `BSTree`.

Creando un iterador `BSTree.iterator(IteratorModes.REVERSEORDER)` el primer elemento que devuelva el iterador será el objeto de la clase `SamePriorityQueue` de mayor prioridad y su primer dato el dato que devuelve el método `getFirst`.

2.2.- `enqueue`

El dato crea un objeto `SamePriorityQueue` de su misma prioridad.

Un objeto `BSTree.iterator()` recorre el árbol en su totalidad comprobando si alguno de los objetos que lo componen es de la misma prioridad que el objeto `SamePriorityQueue` que creó el dato.

Si existe algún dato de las misma prioridad, el dato se enconlará en ese objeto.

En caso contrario, el objeto `SamePriorityQueue` que creó el dato será añadido al árbol con el método `BSTree.add(SamePriorityQueue)`.

2.2.- `dequeue`

La estructura de datos `BSTPriorityQueue` está ordenada de forma que las `SamePriorityQueue` que se corresponden a prioridades más altas quedan a la izquierda en `BSTree`.

Creando un iterador `BSTree.iterator(IteratorModes.REVERSEORDER)` el primer elemento que devuelva el iterador será el objeto de la clase `SamePriorityQueue` de mayor prioridad. La cola de prioridades desencolará desencolando la `SamePriorityQueue.dequeue()` que devolvió el iterador.

2.- Descripción del funcionamiento del iterador en esta implementación.

```
/* Clase privada que implementa un iterador para la *
 * cola con prioridad basada en secuencia.          */
public class PriorityQueueIterator implements IteratorIF<E> {

  Queue<E> auxQueue;
  IteratorIF<E> itAuxQueue;;

  /*Constructor por defecto*/
  protected PriorityQueueIterator(){
      IteratorIF<SamePriorityQueue<E>> itTree = tree.iterator(IteratorModes.REVERSEORDER);
      auxQueue = new Queue<>();
      while(itTree.hasNext()) {
          itAuxQueue = itTree.getNext().iterator();
          while(itAuxQueue.hasNext()) {
              auxQueue.enqueue(itAuxQueue.getNext());
          }
      }
      itAuxQueue = auxQueue.iterator();
  }
```

`PriorityQueueIterator` crea una cola de datos auxiliar y su iterador.

Para crearla, como los objetos `SamePriorityQueue` de mayor prioridad quedan a la izquierda del árbol, `BSTree.iterator(IteratorModes.REVERSEORDER)`recorree los objetos `SamePriorityQueue` encolando los datos en según su orden de llegada y prioridad.

Las operaciones PriorityQueueIterator.`hasNext()`, PriorityQueueIterator.`getNext()`y PriorityQueueIterator.`reset` están delegadas en el iterador de la cola auxiliar.

Aunque el constructor del iterador `BSTreePriorityQueue.iterator()` recorre todos los elementos de `BSTree` para la construcción de la cola auxiliar, el coste asintótico de la operaciones del iterador será lineales O(n) .

6.- Estudio empírico del coste.

1.- Coste asintótico temporal de las operaciones enqueue y dequeue.

BucketQueue.enqueue() O(n), n ≡ Número de prioridades.

```
/*Añade un elemento a la cola de acuerdo a su prioridad
 *y su orden de llegada
 */
public void enqueue(E elem, int prior) {
    boolean insert = false;
    if (list.isEmpty()) {
        SamePriorityQueue<E> auxQueueElem = new SamePriorityQueue<>(prior);
        auxQueueElem.enqueue(elem);
        list.insert(1, auxQueueElem);
        insert = true;
    } else {
        SamePriorityQueue<E> auxQueueElem = new SamePriorityQueue<>(prior);
        auxQueueElem.enqueue(elem);
        IteratorIF<SamePriorityQueue<E>> itList = list.iterator();
        SamePriorityQueue<E> auxQueueList;
        int counter = 1;
        while(itList.hasNext()) {
            auxQueueList = itList.getNext();
            if (auxQueueList.compareTo(auxQueueElem)==0 && !insert) {
                auxQueueList.enqueue(elem);
                insert = true;
            } else {
                if(auxQueueList.compareTo(auxQueueElem)==-1 && !insert) {
                    list.insert(counter, auxQueueElem);
                    insert = true;
                }
            }
            counter++;
        }
        if(!insert) {
            list.insert(list.size()+1, auxQueueElem);
            insert = true;
        }
    }
}
```

```
enqueue (E elem, int prior) {
	if {
		O(1)
	} else {
		O(1)
		while {
```

O(n) n ≡ `list.size()` = Número de objetos `SamePriorityQueue` en la lista enlazada.

```
		}
	}
	if {
```

O(n) n ≡ `list.size()` = Número de objetos `SamePriorityQueue` en la lista enlazada.

```
	}
}
```

`BucketQueue.enqueue(E elem, int prio)` O(n)

n ≡ `list.size()` = Número de objetos `SamePriorityQueue` en la lista enlazada.

Por la estructura de las colas de prioridad con la implementación `BucketQueue`, el coste del método no está en función del número de pacientes a encolar.

BucketQueue.dequeue() O(1)

```java
/*Elimina el elemento más prioritario y que llegó a la cola
 *en primer lugar
 * @Pre !isEmpty()
 */
public void dequeue() {
    list.get(1).dequeue();
    if (list.get(1).isEmpty()) {
        list.remove(1);
    }
}
```

```
public void dequeue(){
      O(1)
      if {
            O(1)
      }
}
```

BucketQueue.dequeue() O(1)

Por la estructura de las colas de prioridad con la implementación `BucketQueue`, el coste del método no está en función de del número de objetos `SamePriorityQueue` enlazados en la lista de prioridad.

BSTreePriorityQueue.enqueue() O(n), n ≡ Número de prioridades.

```java
/*Añade un elemento a la cola de acuerdo a su prioridad
 *y su orden de llegada
 */
public void enqueue(E elem, int prior) {
    SamePriorityQueue<E> auxQueueOutside = new SamePriorityQueue<>(prior);
    IteratorIF<SamePriorityQueue<E>> itTree = tree.iterator(IteratorModes.REVERSEORDER);
    auxQueueOutside.enqueue(elem);
    boolean added = false;
    if(tree.isEmpty()) {
        tree.add(auxQueueOutside);
    } else {
        while (itTree.hasNext() && !added) {
            SamePriorityQueue<E> AuxQueueInside = itTree.getNext();
            if(auxQueueOutside.compareTo(AuxQueueInside)==0){
                AuxQueueInside.enqueue(elem);
                added = true;
            }
        }
    }
    if(!added) {
        tree.add(auxQueueOutside);
    }
}
```

`BSTree.add(E elem)` en su caso más desfavorable, la inserción de un elemento en la última posición de un árbol degenerado el lista enlazada, tiene un coste asintótico O(n).
n≡`BSTree.size()` = número de elementos del árbol.

`enqueue (E elem, int prior) {`

`tree.iterator(IteratorModes.REVERSEORDER)` **O(n), n ≡ BSTree.size(), número de objetos SamePriorityQueue que forman el árbol.**

`if {`

O(1)

`} else {`

`while {`

O(n), n ≡ BSTree.size(), número de objetos SamePriorityQueue que forman el árbol.

`if {`

O(1)

`}`

`}`

`}`

`if {`

O(n), n ≡ BSTree.size(), número de objetos SamePriorityQueue que forman el árbol.

`}`

`}`

`BSTreePriorityQueue.enqueue(E elem, int prio)` O(n)

n ≡ `list.size()` = número de objetos `SamePriorityQueue` que forman el árbol.

Por la estructura de las colas de prioridad con la implementación `BSTreePriorityQueue`, el coste del método no está en función del número de pacientes a encolar.

`BSTPriorityQueue.dequeue()` O(n), n ≡ Número de prioridades.

```java
/*Elimina el elemento más prioritario y que llegó a la cola
 *en primer lugar
 * @Pre !isEmpty()
 */
public void dequeue() {
    if (!tree.isEmpty()) {
        IteratorIF<SamePriorityQueue<E>> itTree = tree.iterator(IteratorModes.REVERSEORDER);
        SamePriorityQueue<E> auxQueue = itTree.getNext();
        auxQueue.dequeue();
        if(auxQueue.isEmpty()) {
            tree.remove(auxQueue);
        }
    }
}
```

`public void dequeue(){`

`tree.iterator(IteratorModes.REVERSEORDER)` **O(n), n ≡ BSTree.size(), número de objetos `SamePriorityQueue` que forman el árbol.**

`}`

`BucketQueue.dequeue()` O(n)

n ≡ `list.size()` = número de objetos `SamePriorityQueue` que forman el árbol.

Por la estructura de las colas de prioridad con la implementación `BSTreePriorityQueue`, el coste del método no estás en función del número de pacientes a encolar.

2.- Coste asintótico temporal de las operaciones enqueue y dequeue con los costes empíricos obtenidos.

BucketQueue.enqueue()

Para generar los tiempos se emplea el algoritmo:

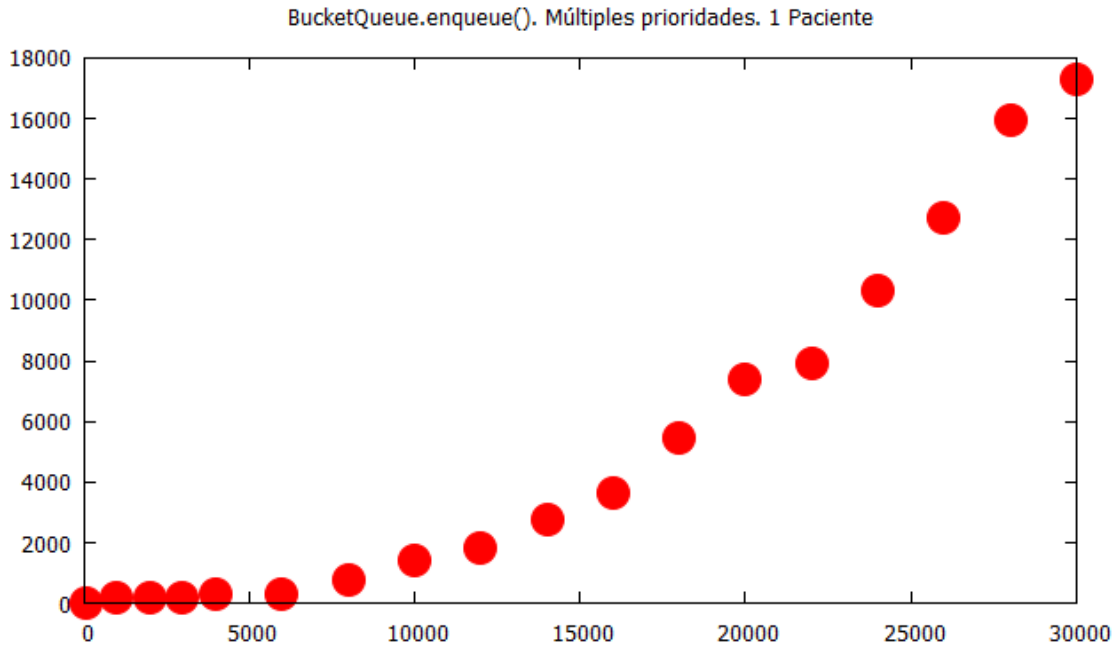
```
for (int i=1; i<=num_prioridades; i++) {
        Bucketqueue.enqueue(paciente1, i);
}
```

O(n) x Coste BucketQueue.enqueue() = Coste del algoritmo de generación de tiempos;

Coste BucketQueue.enqueue() = Coste algoritmo generación / O(n)

Tiempos generados

Número de prioridades	Tiempos generados (ms)
100	19
1000	160
2000	193
3000	178
4000	284
6000	310
8000	782
10000	1417
12000	1795
14000	2756
16000	3628
18000	5433
20000	7408
22000	7922
24000	10306
26000	12723
28000	15951
30000	17279



De la observación empírica: Coste generación = $O(n^2)$. Cada llamada al método tiene un coste lineal: $1+2+3+\ldots+n = (n^2+n)/2 \rightarrow O(n^2)$.

Coste BusketQueue.enqueue() = $O(n^2/n)$ = **O(n)**

La observación empírica confirma la hipótesis teórica.

Se adjuntan los ficheros de datos para la obtención de los tiempos en: Graficas\BQ\Enqueue

BucketQueue.dequeue()

Para generar los tiempos se emplea el algoritmo:

```
for (int i=1; i<=num_pacientes; i++) {
        Bucketqueue.enqueue(paciente_i, prioridad1);
}
for (int i=1; i<=num_pacientes; i++) {
        Bucketqueue.dequeue();
}
```

Coste generación = O(n) x Coste BucketQueue.enqueue() + O(n) x Coste BucketQueue.dequeue();

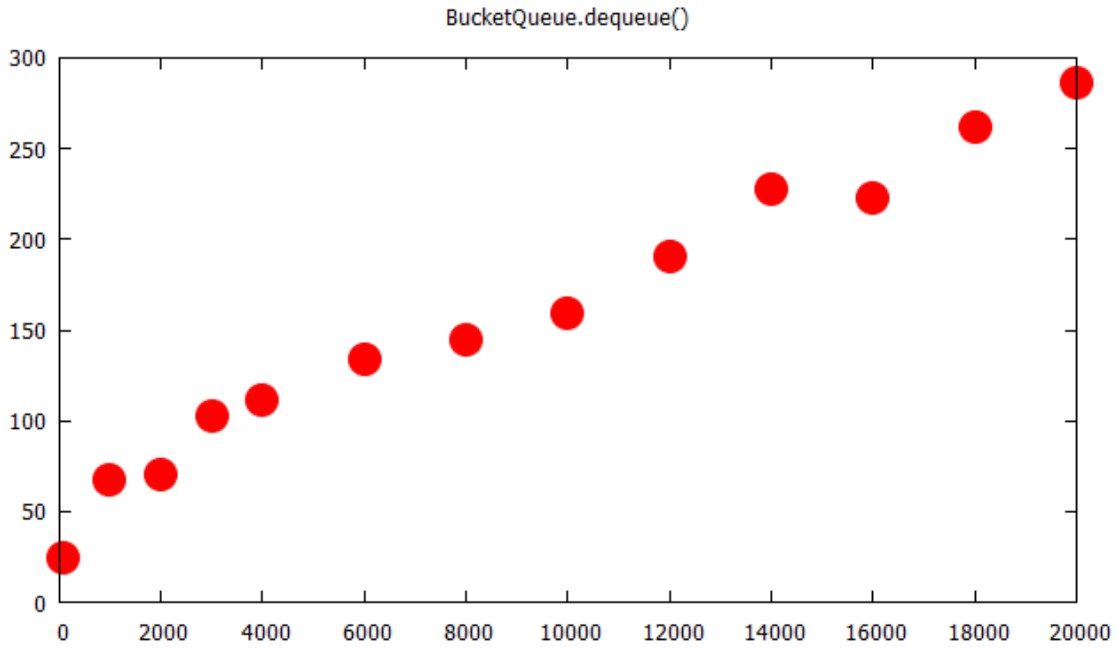
Coste generación = O(n) x O(1) + O(n) x Coste BucketQueue.dequeue();

Coste generación = O(n) x Coste BucketQueue.dequeue();

Coste BucketQueue.dequeue() = Coste generación / O(n)

Tiempos generados

Número de pacientes	Tiempos generados (ms)
100	25
1000	67
2000	70
3000	103
4000	111
6000	134
8000	145
10000	159
12000	191
14000	228
16000	223
18000	262
20000	286



De la observación empírica: Coste generación = O(n)

Coste BusketQueue.dequeue() = O(n) / O(n) = **O(1)**

La observación empírica confirma la hipótesis teórica.

Se adjuntan los ficheros de datos para la obtención de los tiempos en: Graficas\BQ\Dequeue

BSTreePriorityQueue.enqueue()

Para generar los tiempos se emplea el algoritmo:

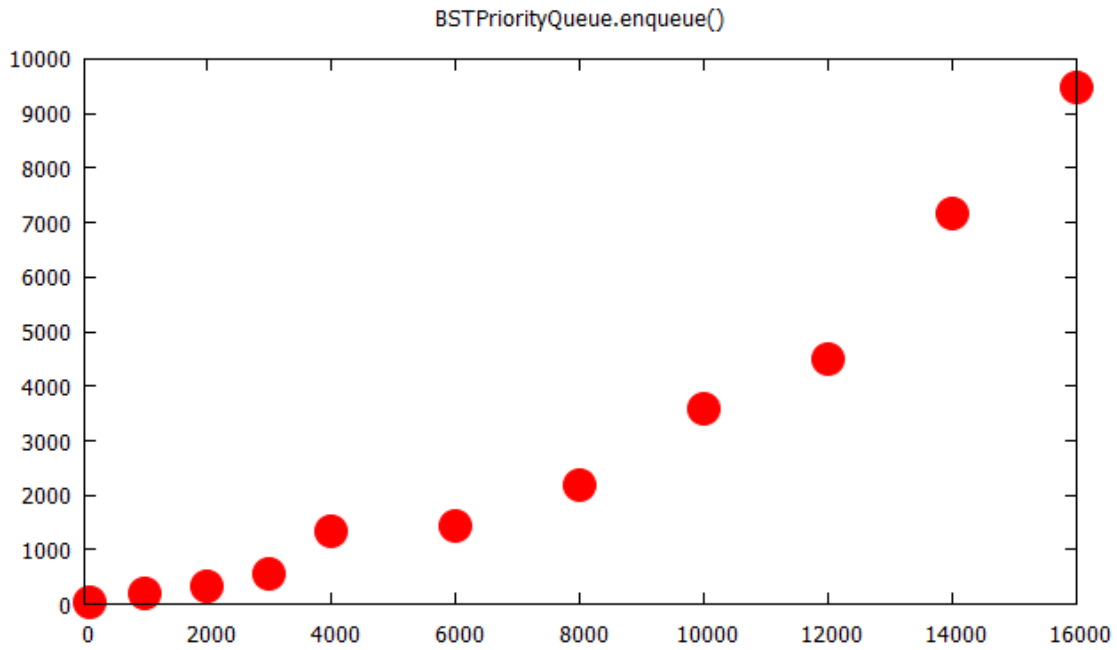
```
for (int i=1; i<=num_prioridades; i++) {
        BSTreePriorityQueue.enqueue(paciente1, i);
}
```

O(n) x BSTreePriorityQueue.enqueue() = Coste del algoritmo de generación de tiempos;

Coste BSTreePriorityQueue.enqueue() = Coste algoritmo generación / O(n)

Tiempos generados

Número de prioridades	Tiempos generados (ms)
100	46
1000	185
2000	322
3000	547
4000	1332
6000	1439
8000	2200
10000	3589
12000	4507
14000	7169
16000	9471



De la observación empírica: Coste generación = $O(n^2)$. Cada llamada al método tiene un coste lineal: $1+2+3+\ldots+n = (n^2+n)/2 \rightarrow O(n^2)$.

Coste BSTPriorityQueue.enqueue() = $O(n^2/n)$ = **O(n)**

La observación empírica confirma la hipótesis teórica.

Se adjuntan los ficheros de datos para la obtención de los tiempos en: Graficas\BST\Enqueue

BSTreePriorityQueue.dequeue()

Para generar los tiempos se emplea el algoritmo:

```
for (int i=1; i<=num_prioridades; i++) {
        BSTreePriorityQueue.enqueue(paciente1, i);
}
for (int i=1; i<=num_prioridades; i++) {
        Bucketqueue.dequeue();
}
```

Al ser costes en tiempo, el coste asintótico del método `BSTreePriorityQueue.dequeue()` se evaluará en función de = tiempo total – tiempo inserción, siendo el tiempo de inserción el tiempo empleado en:

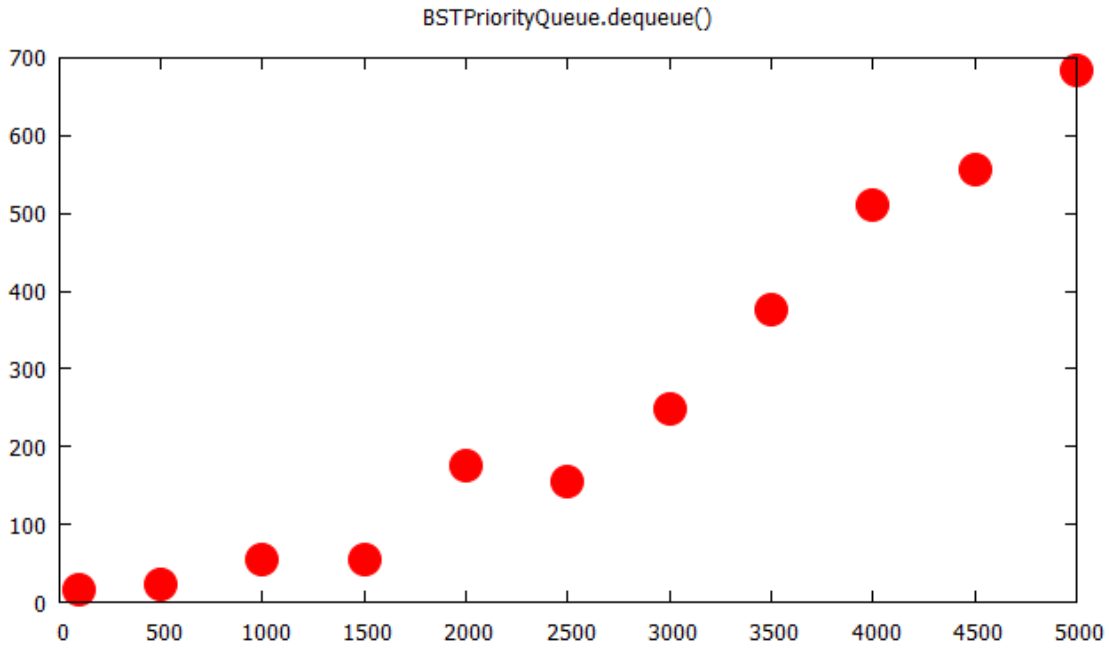
```
for (int i=1; i<=num_prioridades; i++) {
        BSTreePriorityQueue.enqueue(paciente1, i);
}
```

O(n) x BSTreePriorityQueue.dequeue() = Coste del algoritmo de generación de tiempos;

Coste BSTreePriorityQueue.dequeue() = Coste algoritmo generación / O(n)

Tiempos generados

Número de Prioridades	Tiempo Total	Tiempo inserción	Tiempo dequeue
100	40	23	17
500	109	85	24
1000	224	169	55
1500	252	198	54
2000	473	297	176
2500	570	415	155
3000	770	521	249
3500	988	612	376
4000	1220	710	510
4500	1359	802	557
5000	1864	1180	684



De la observación empírica: Coste generación = $O(n^2)$ Cada llamada al método tiene un coste lineal: $1+2+3+\ldots+n = (n^2+n)/2 \rightarrow O(n^2)$.

Coste BSTreePriorityQueue.dequeue() = $O(n^2/n)$ = **O(n)**

La observación empírica confirma la hipótesis teórica.

Se adjuntan los ficheros de datos para la obtención de los tiempos en: Graficas\BST\Dequeue