Estrategias de Programación y Estructuras de Datos
PEC - 2021

Centro Asociado de la UNED en Bizkaia

Martín Romera Sobrado
Bilbao

3 de junio de 2021

1. Preguntas teóricas

1.1. Bucket Queue

1.1.1. Pregunta 1 : Secuencia adecuada

¿Qué tipo de secuencia sería la adecuada para realizar esta implementación? ¿Por qué? ¿Qué consecuencias tendría el uso de de otro tipo de secuencias?

La opción más viable sería implementarlo sobre una lista, ya que el orden de la secuencia `SamePriorityQueue` no está por el orden de entrada de estas en la `BucketQueue`. Las pilas y colas tienen un orden definido por el orden de entrada, lo cual las hace inconvenientes, sin embargo en la lista podemos alterar su orden a nuestro gusto, en este caso por la propiedad `priority` de las propias `SamePriorityQueue`.

1.1.2. Pregunta 2 : Funcionamiento de operaciones

Describe el funcionamiento de las operaciones `getFirst`, `enqueue` y `dequeue` con esta estructura de datos.

- `getFirst` : Se encargará de obtener el primer elemento en la cola de prioridad, de forma que accedera a la priemra `SamePriorityQueue` de la lista y obtendrá su primer elemento mediante `getFirst`.
- `enqueue` : Se encargará de encolar el elemento en su `SamePriorityQueue` correspondiente, de forma que irá recorriendo la lista hasta que encuentre una `SamePriorityQueue` con mismo o menor valor de prioridad. Si encuentra una `SamePriorityQueue` con misma prioridad que el elemento entonces lo encola en esa misma cola. Si la cola que encuentra tiene menor prioridad, entonces crea una `SamePriorityQueue` nueva con el elemento y la inserta en la lista antes de la cola con menor prioridad que hemos encontrado en la busqueda. Si no encuentra ninguna `SamePriorityQueue` con menor o igual prioridad entonces, crea una nueva cola con el elemento y la inserta al final de la cola.
- `dequeue` : Se encargará de extraer el elemento en la cabeza de la cola de prioridad. Para eso extraerá el primer elemento de la `SamePriorityQueue` de la lista. Si con esa acción la `SamePriorityQueue` se queda vacía (`isEmpty()`) entonces la eliminará de la lista dejando ahora en primera posición a la siguiente cola con menor prioridad.

1.1.3. Pregunta 3 : Funcionamiento del iterador

Describe el funcionamiento del iterador de la cola con piroridad en esta implementación.

El iterador iterará sobre cada elemento del `BucketQueue` para ello podemos hacer uso de los iteradores de la implementación de `List` (`listIterator`) y de la implementación de `SamePriorityQueue` (`subIterator`). `listIterator` iterará sobre las `SamePriorityQueue` de la lista y `subIterator` iterará sobre los elementos de las `SamePriorityQueue` a las que apunta `listIterator`. Devolveremos el valor al que apunta en cada momento el `subIterator`, y actualizaremos la `SamePriorityQueue` sobre la que itera cada vez que se quede sin elementos en la anterior (`!hasNext()`) con ayuda de `listIterator`.

1.2. Árbol Binario de Busqueda

1.2.1. Pregunta 1 : Funcionamiento de las operaciones

Describe el funcionamiento de las operaciones `getFirst`, `enqueue` y `dequeue` con esta estructura de datos.

- `getFirst` : Esta operación se aprovecha del iterador, creando uno para el objeto `BSTPriorityQueue` y obteniendo el primer elemento con `getNext`.

- `enqueue` : También se aprovecha del iterador de la clase `BSTPriorityQueue`. Recorre con el iterador todo el árbol comprobando si hay algún nodo con una `SamePriorityQueue` con la misma prioridad que el elemento en la que encolarlo. Si lo encuentra lo encola en esa misma cola y si no, crea una nueva `SamePriorityQueue`, encola el elemento y lo añade al árbol con el método nativo `add` de la clase `BSTree` que lo añade en su posición correcta del árbol.

- `dequeue` : De nuevo nos aprovechamos del iterador para apuntar al elemento con mayor prioridad llamando una sola vez a `getNext` y hacemos un `dequeue` del resultado.

1.2.2. Pregunta 2 : Funcionamiento del iterador

Describe el funcionamiento de la cola con prioridad en esta implementación

El iterador hará uso de dos iteradores. Uno de ellos (llamemoslo `treeIterator`) iterará sobre los diferentes nodos del árbol apuntando en cada iteración a una nueva `SamePriorityQueue`. El otro iterador (llamemoslo `subIterator`) iterará sobre los elementos contenidos por la `SamePriorityQueue` a la que apunta `treeIterator`. Cuando `subIterator` no tenga nada más que recorrer, actualizamos `treeIterator` al siguiente nodo y hacemos que `subIterator` sea el iterador de esta nueva `SamePriorityQueue`.

1.3. Estudio empírico del coste

1.3.1. Cálculo del coste asintótico temporal

Calcule el coste asintótico temporal en el caso peor de las operaciones `enqueue` *y* `dequeue` *para ambas implementaciones.*

Para la implementación de `BucketQueue` la operación `enqueue` funciona como si encolaramos en una `Queue` contenida en la n-sima posición de una `List`. Esta operación tiene un coste lineal $O(n)$. Por otra parte la función `dequeue` tendrá siempre un coste constante, ya que para desencolar el elemento de mayor prioridad este se encontrará en la cabeza de la cola en la primera posición siempre. Acceder al elemento n de una lista tiene un coste de $O(n)$, como $n = 1$ es una constante, entonces el tiempo de ejecución también $O(1)$ y desencolar elementos de una cola también tiene un tiempo de ejecución constante $O(1)$ de lo que observamos que esta operación tiene un tiempo de ejecuión constante.

Para la implementación de `BSTPriorityQueue` sin embargo las operaciones tendrán un coste acorde a la profundiad del árbol que contenga las colas (h a partir de ahora). Esto tiene sus ventajas y desventajas. La ventaja es que un árbol binario de busqueda equilibrado tendrá una proundidad logarítmica respecto al número de elementos (n), es decir $h = \log_2 n$. Sin embargo si el árbol no está balanceado h estará en un valor entre $\log_2 n$ y n. En el peor caso h será n, haciendo que la estructura tenga semejanza a una lista como la que usa `BucketQueue` y por ende los costes temporales de sus operaciones serán iguales, $O(n)$ para `enqueue` y $O(1)$ para `dequeue`. Por supuesto `enqueue` no tendrá siempre un coste lineal, a que su coste dependerá de h, de forma que realmente el coste de esta operación realmente es $O(h)$.

1.3.2. Comparación entre coste asintótico y costes empíricos

Compare el coste asintótico temporal obtenido en la pregunta anterior con los costes empíricos obtenidos. ¿Coincide el coste calculado con el coste real?

Para probar los algoritmos se realizan 6 pruebas sobre cada función, cada una estructura de 1000, 2000, 3000, 4000, 5000 o 6000 nodos con colas de un solo elemento, cada test se ejecutará 100 veces. Las colas solo tienen un elemento ya que no nos interesa evaluar el `enqueue` o `dequeue` de `SamePriorityQueue` si no el de las estructuras `BucketQueue` y `BSTPriorityQueue`.

Para `BucketQueue` la operación `enqueue` los tests nos dan los resultados de la figura 1.

El resultado del test empírico no es completamente lineal, esto puede deberse principalmente a el margen de error del tiempo de ejecución (σ_t), que vemos que también incrementa con el tiempo (exceptuando el 4^{o} caso que por algún motivo tiene un margen de error de tiempo mayor, tal vez por las cosas que había en ejecución en ese momento en mi ordenador). De todas formas la forma que adopta el resultado del test es casí lineal.
La operación `dequeue` dijimos que tenía un tiempo de ejecución constante, y el resultado del test lo muestra más o menos, como podemos ver en la figura 2.

Para este test lo que se ha hecho es restar el resultado de tiempo de este test, con el resultado del caso correspondiente en el test anterior (para restar el tiempo de encolar todos los elementos). Como puede observarse, la tendencia de tiempo no es creciente en ningún momento, fluctua entre los tests cerca de los 0 ms lo que denota una tendencia de tiempo constante.

Vamos ahora con `BSTPriorityQueue`, analizamos primero la operación `enqueue` y obtenemos los resultados de la figura 3.
De nuevo, el resultado no es completamente lineal, pero vemos que tiene una tendencia similar al test de la operación `enqueue` de la anterior implementación. Esto es porque esta estructura en el caso peor tiene la misma forma que una lista enlazada, de forma que el tiempo de ejecución de `enqueue` es lineal (o casí lineal prácticamente) y el tiempo de `dequeue` constante como podemos ver en la tendencia que muestra 4.

Número de nodos	σ_t	t
1000	2,927029	22,35
2000	4,635256	55,12
3000	5,282755	107,65
4000	17,580384	192,51
5000	8,428968	294,45
6000	13,587877	407,36

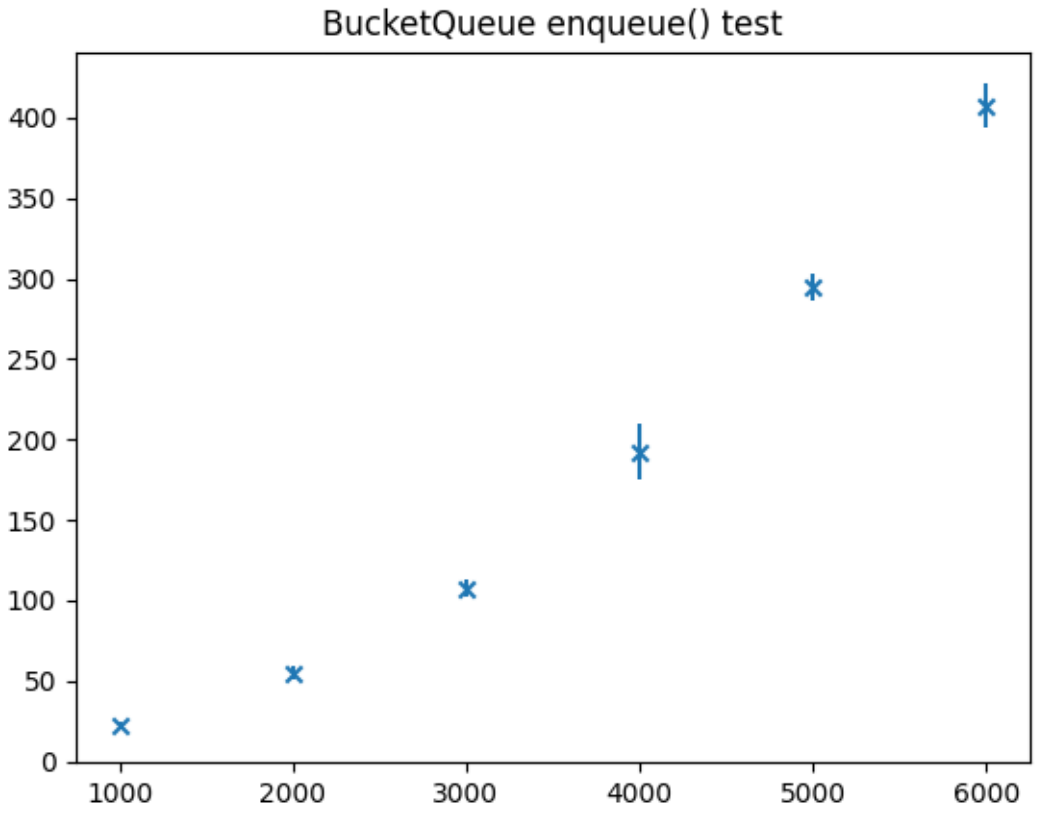

Figura 1: Resultados del test de coste empírico para `enqueue` de `BucketQueue`

Número de nodos	σ'	$\lvert\sigma\rvert$	t'	t
1000	1,905020	$1,022009$	21,47	$-0,88$
2000	3,364209	1,271047	55,89	0,86
3000	3,602999	1,679756	116,28	8,63
4000	8,867604	8,71278	189,16	$-3,35$
5000	8,098173	0,330795	292,14	$-2,31$
6000	14,510341	0,922464	407,70	$-0,34$

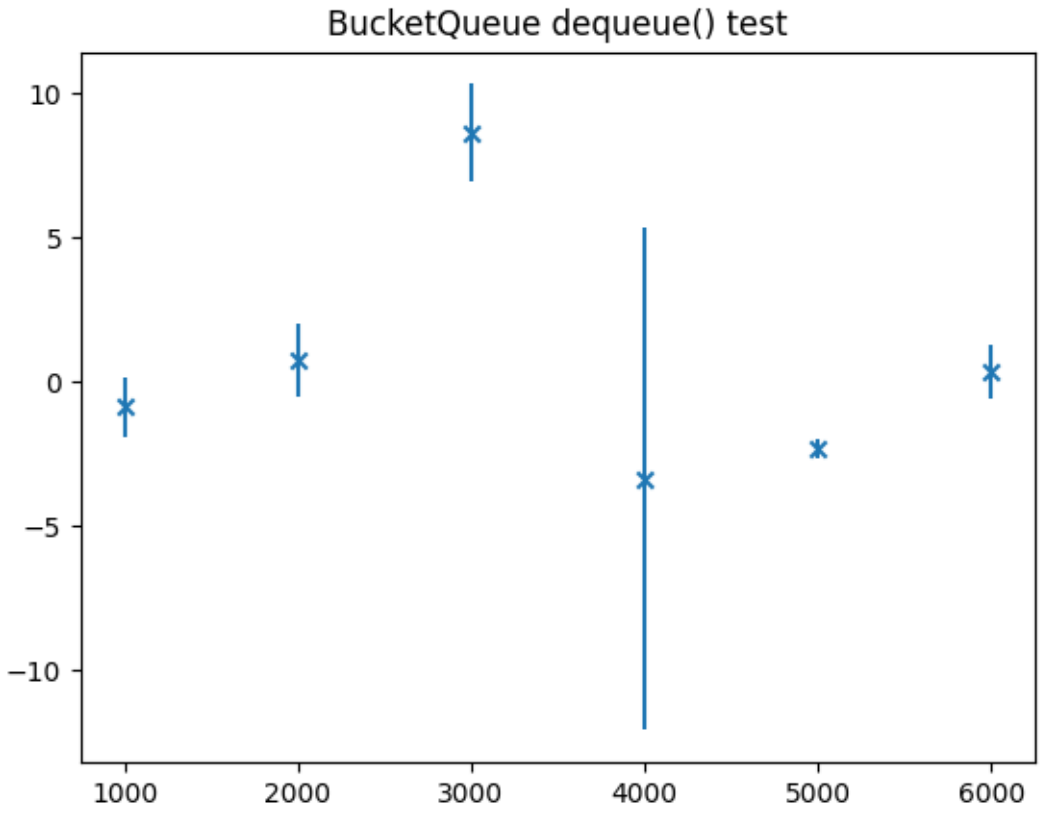

Figura 2: Resultados del test de coste empírico para `dequeue` de `BucketQueue`

Número de nodos	σ_t	t
1000	3,594704	21,91
2000	3,403219	56,41
3000	4,908146	118,01
4000	9,459783	189,95
5000	9,056401	295,04
6000	15,672090	408,16

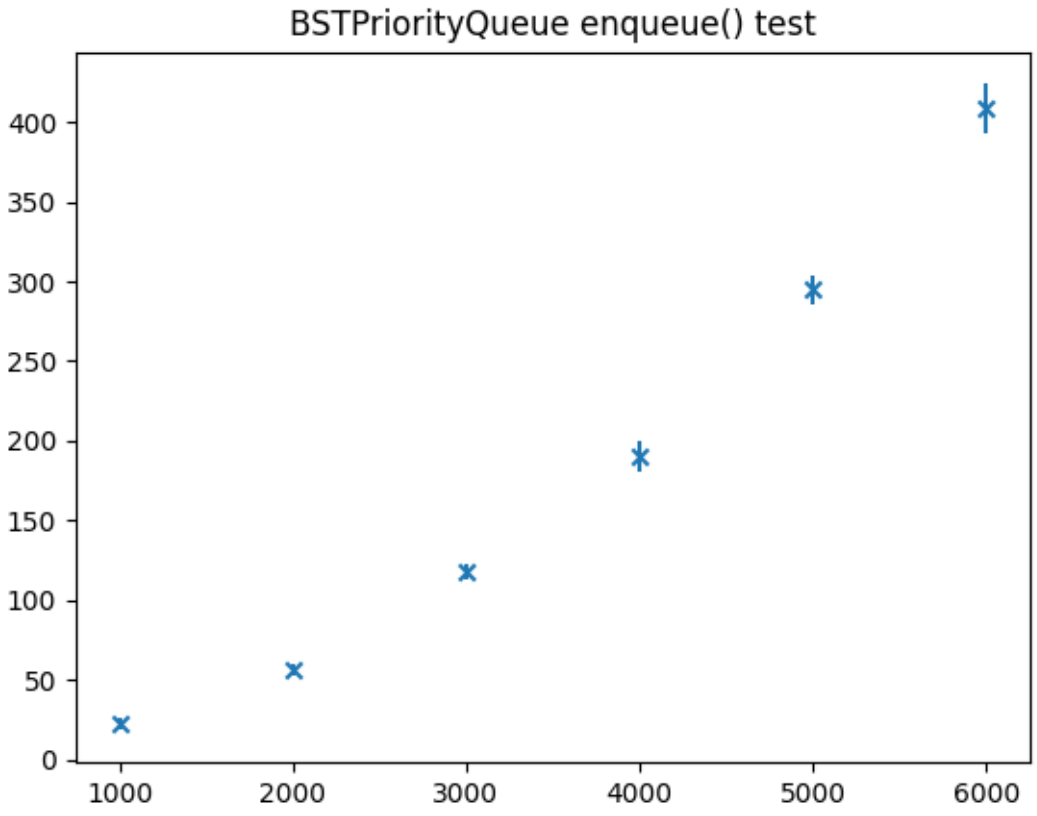

Figura 3: Resultados del test de coste empírico para `enqueue` de `BSTPriorityQueue`

Número de nodos	σ'	$\lvert\sigma\rvert$	t'	t
1000	1,894624	1, 70008	21,48	0,43
2000	2,946184	0,457035	57,00	−0,59
3000	3,255530	1,652616	116,94	−1,07
4000	9,026073	0,43371	189,70	−0,25
5000	6,277866	2,778535	292,22	−2,82
6000	26,366454	10,694364	412,01	3,85

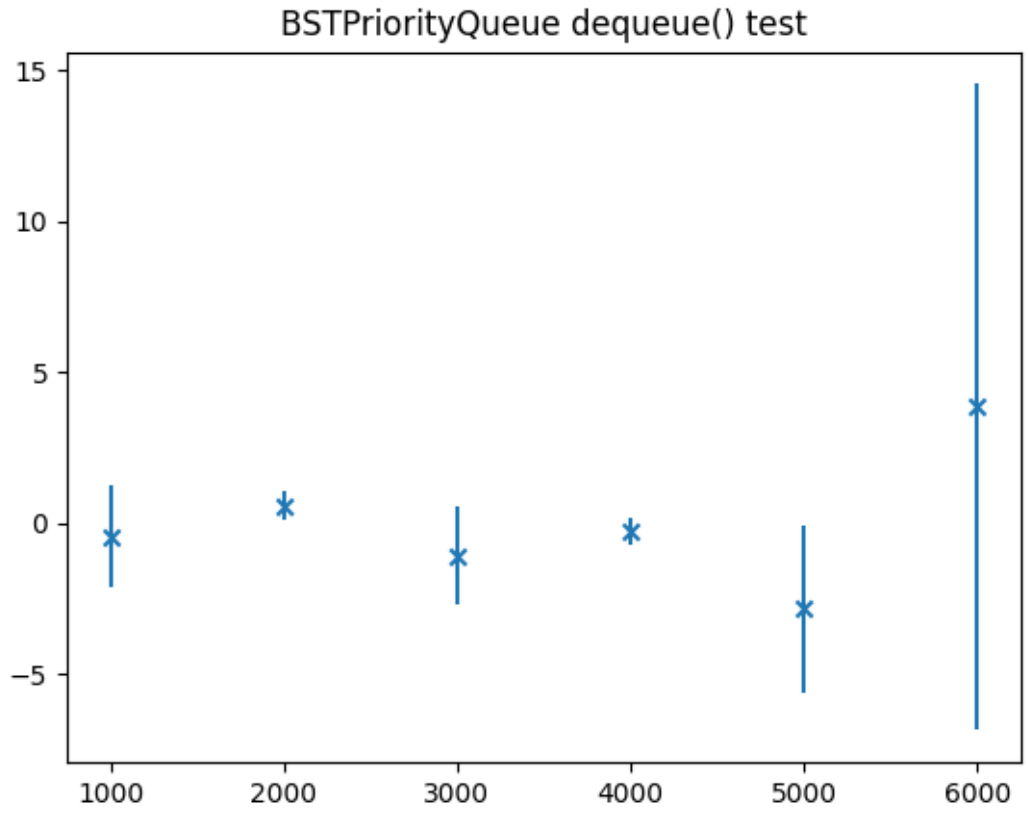

Figura 4: Resultados del test de coste empírico para `dequeue` de `BSTPriorityQueue`