Informe de experimentos

Estructura de Datos Grupo 3

Presentado por:

Manuel Alejandro Verjan

Codigo 1088339441

German Andres Charfuelan

Codigo 1088652189

Presentado a:

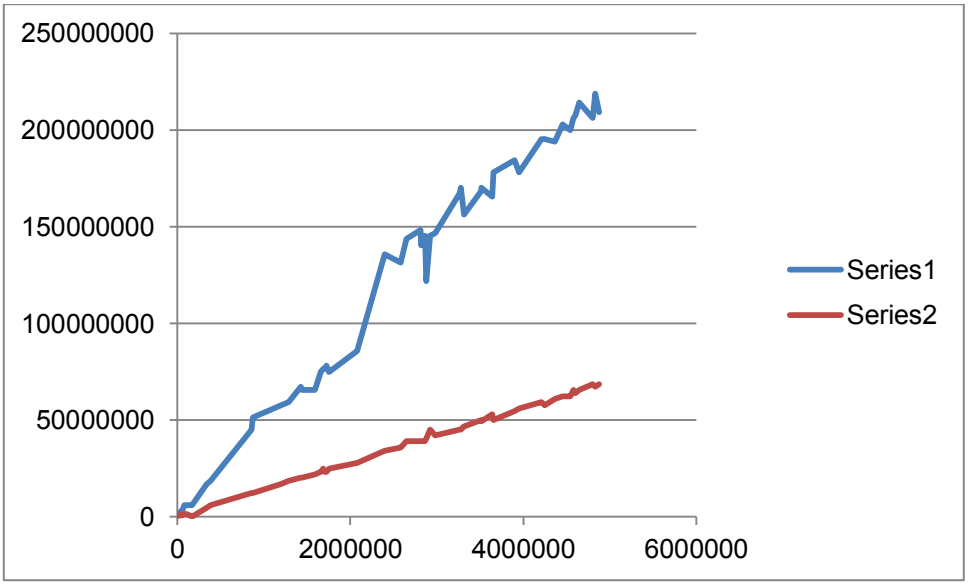
Gustavo Adolfo Gutierrez Sabogal

1. Se genero un vector con una cantidad importa de números aleatorios, los cuales, iban a ser usados como tamaño de un array y un vector para tomar medidas de su tiempo de ejecución. Para que los datos fueran más exactos, se tomó 10 veces el tiempo para cada tamaño y luego se sacó el promedio.

Despues de haber dado 50 tamaños diferentes, en el rango de 10000 y 5000000 elementos, al array y al vector, se obtuvo que:

- El array es alrededor de 3 veces más rápido que el Vector. Esto se debe a que el Array toma el tamaño necesario para almacenar la cantidad total de datos, desde un principio. Por el contra, el Vector tiene un tamaño estándar de 4, y por tanto, debe hacer resizes cada que se llena. Esto conlleva a mayor gasto temporal. Cabe resaltar que para los resize, se utilizó la expresión de capacity *3, para calcular el nuevo tamaño del Vector. Como veremos mas adelante, esto influye de manera directa en el tiempo que se toma Vector para almacenar los elementos.

- Como se puede observar en la siguiente grafica:



El tiempo de ejecución del array, crece de forma más lenta que el del vector. Por otro lado, el Vector crece de forma mucho más rápida, así como presenta más altibajos. Esto puede deberse también, al constante cambio de tamaño al que está sujeto el Vector para poder ingresar todos los elementos que se le pide.

Ahora, se evaluara el impacto en el tiempo de cada expresión para calcular el nuevo tamaño.

Para ello, se utilizó de nuevo los tamaños obtenidos mediante el vector de números aleatorios.

Al igual que en el caso anterior, se tomó el tiempo 10 veces con cada expresión, para cada tamaño, para así obtener mayor precisión.

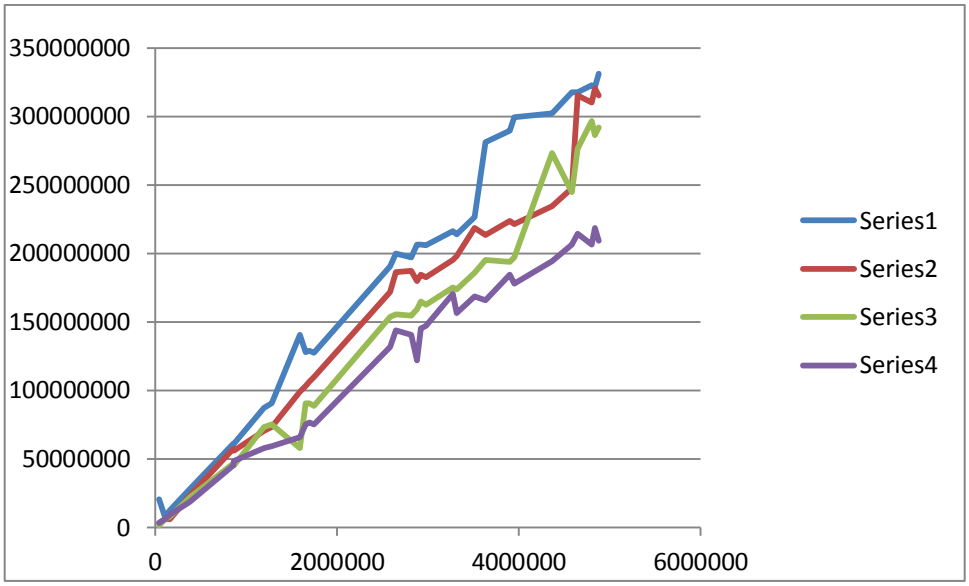
Tomadas las medidas se obtuvo que:

- Entre menor sea la expresión para calcular el nuevo tamaño del vector, mayor es el tiempo. Esto se debe a que, al ser menor el aumento en el vector, más veces se debe realizar es resize, y esto, como ya se había comentado anteriormente, va fuertemente ligado al gasto temporal. Cabe resaltar que la expresión “Capacity +1” es nefasta a nivel temporal, sobretodo, cuando se trabaja con los tamaños que se trabajó en el experimento.

- Mientras mayor sea el tamaño del Vector, mayor se va a haciendo la diferencia en tiempo con una expresión y otra. En tamaños pequeños, la relación de aumento de tiempo, entre una expresión y otra, no es tan marcada. Incluso, en algunos casos, una expresión menor llegaba a tener un tiempo ligeramente mejor que una mayor (Esto se puede deber a muchos factores). Pero, conforme los tamaños iban aumentado, la diferencia se iba notando, llegando a ser la expresión “Capacity *3” , 1,58 veces mas rápido que “Capacity *1.5”, 1,503 veces mas rápido que “Capacity * 1.8” y 1,39 veces mas rápido que “Capacity *2”.

En conclusión, el valor para calcular el nuevo tamaño es indirectamente proporcional al tiempo. Ahora, mas adelante veremos si esto se compensa con una mejora en el gasto de memoria.

Se adjunta grafica del experimento:



Morado: Capacity *3
Verde: Capacity *2
Rojo: Capacity *1.8
Azul: Capacity *1.5

2. En este apartado, se medirá la efectiva de la función add de la clase vector cuando se ingresan elementos en cualquier posición y no únicamente al final. Para ello, se creó un vector de 5000000 datos, y con la función produce, se generaron números aleatorios con un rango de números que puedan sean posición validas (0 – 4999999) en el vector de datos anteriormente creado.

   El experimento se dividió en dos. Primero, se tomó el tiempo agrupado, es decir, cuánto tarda en agregar, 100, 500, 1000….. datos, para comparar estos con los datos obtenidos en el primer punto. Luego, se evaluó el tiempo individualmente, para así, ver si estos tiempos están acordes con la complejidad teórica.

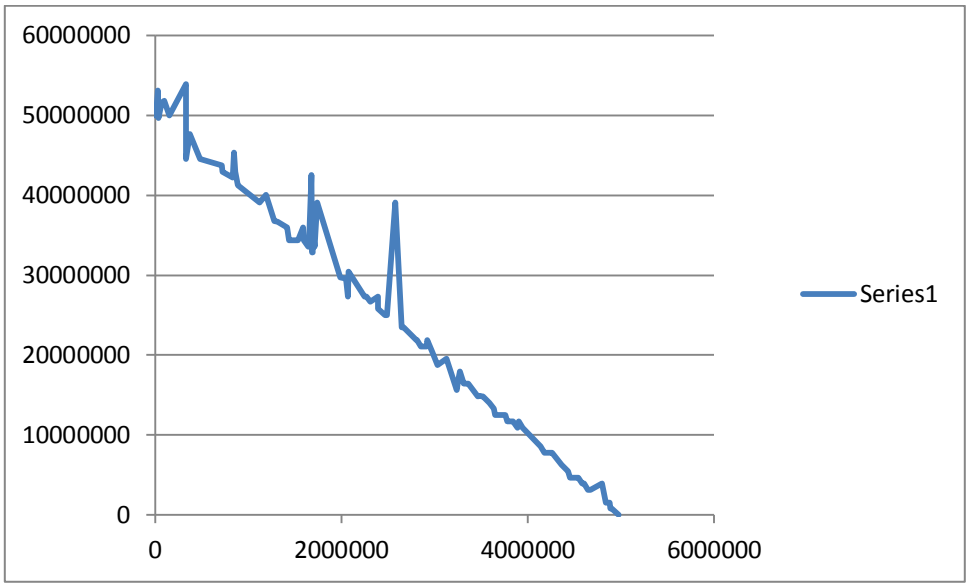
   En la primer parte, como era de esperarse, se comprobó que la complejidad al añadir elementos en posición diferentes al final, es mucho mayor, y por tanto, el tiempo aumenta considerablemente. Se tuvo por ejemplo que, añadir 10000 elementos en posiciones aleatorias, es alrededor de 2000 veces más demorado que ingresar 15000 al final del Vector.

Ahora, se examinaron los resultados obtenidos al evaluar los tiempos individualmente y queda patente la complejidad real de la función add (O(size-i)).

Conforme mayor era la posición, y por tanto, más cerca al final del Vector estaba, menor era el tiempo que tomaba agregar el dato. Esto debido a que se deben desplazar menos elementos para así abrirle espacio al nuevo elemento.

Se puede concluir entonces, que la posición es indirectamente proporcional al tiempo que toma agregar el dato.

Se adjunta gráfica:



3. Ahora, se evaluara el rendimiento de la función RemoveNaive, la cual elimina el elemento que hay en una posición dada.

Para ello, se siguió el mismo procedimiento que en los anteriores casos. Se generó un vector con una cantidad de 5000000 datos, y luego uno que contuviera valores aleatorios que fueran posición validas en el vector anteriormente creado.

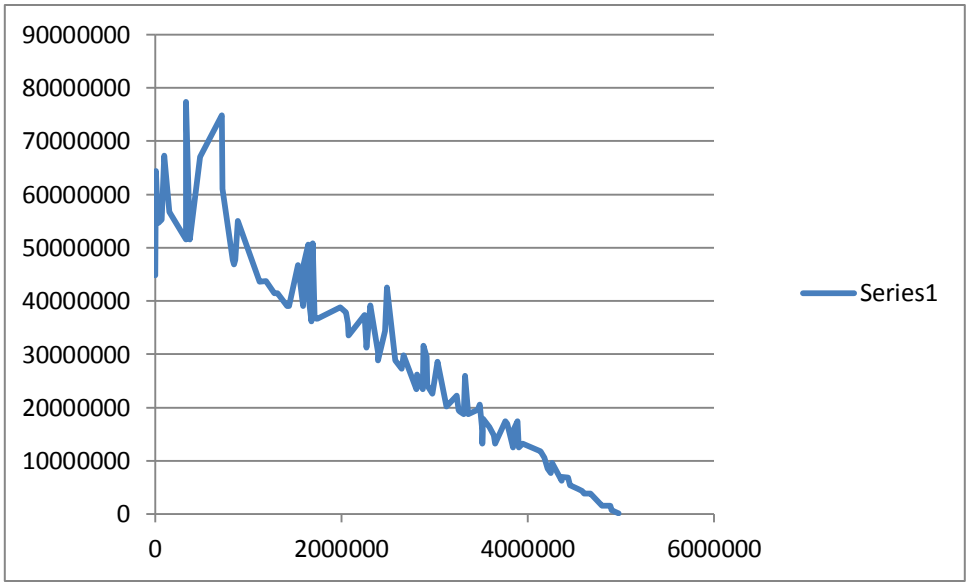
Para mayor precisión, se tomaron 10 tiempos en cada posición.

Una vez tomados los datos se observó que:

Al igual que para agregar, el valor de la posición, es inversamente proporcional al tiempo que toma hacer la operación, y por tanto, los resultado obtenidos son acordes a la complejidad de O(size-i). Esto se debe a que, entre menor era el tamaño, más elementos había que desplazar hacia la izquierda para así lograr que el elemento fuera eliminado.

La conclusión a la que se puede llegar, después de haber realizado los últimos dos puntos es que, tanto para agregar, como para borrar, lo más complicado es cuando se trata de modificar posiciones al principio del Vector.

Se adjunta gráfico del experimento:



4. En este punto, se evaluó la eficiencia en memoria de la estructura de dato Vector. Para ello, se hizo uso de los métodos waste y resizes implementados en la clase Vector, para ver así, cuanto gasto temporal y cuanto desperdicio de memoria se obtiene.

   El procedimiento a seguir, fue el siguiente. Se crearon vectores con 30 tamaños diferentes, los cuales empezaban en 1000 y se iban incrementando en 2000 por cada iteración. Cada uno de estos 30 tamaños, fue evaluado con las 5 expresiones, para así, poder llegar a una conclusión.

   Para evaluar cual expresión era la más eficiente y cual tenía la mejor relación desperdicio/resize, lo que se hizo fue obtener un coeficiente para cada tamaño con cada expresión.

   Este coeficiente estaba dado por:

   C = (waste*0.5) + (resizes*0.5)

   Una vez obtenido los coeficientes para cada tamaño con cada expresión, lo que se hacía era sacar un promedio de los coeficientes obtenidos para cada expresión y así, ver cuál era el que tenía la mejor relación entre gasto y resizes.

   En ese orden de ideas, se obtuvo el siguiente ranking, el cual va de mejor a peor.

   1. Capacity*1.5
   2. Capacity *1.8
   3. Capacity *2
   4. Capacity *3
   5. Capacity +1

   Con estos resultados, llegamos a la conclusión de que lo mejor, es que haya un buen balance entre gasto y numero de resizes. Ya que, por ejemplo, el caso de “Capacity +1” era bueno a nivel de gasto, pero muy malo a nivel de la cantidad de resizes que debía hacer el vector. Algo similar pasaba con “Capacity *3”, el mejor a nivel de resizes, pero mal en el apartado de gasto, pues desperdiciaba mucha memoria. “Capacity *1.5” no

era el mejor en ninguno de los dos casos pero tampoco era nefasto en alguno de ellos. Mantenía una buena relación, y por tanto, se convierte en la mejor opción.

Por último, y para culminar con el experimento, se busco verificar la efectividad de las funciones de RemoveNaive y RemoveWaste.

Para ello, lo que se hizo es, tras llenar un vector con X cantidad de elementos, utilizando las diferentes expresiones para aumento de tamaño, se elimino el ultimo elemento con la función RemoveWaste. Con esto, lo que se buscaba era ver con que frecuencia nos es útil esta función, es decir, si de verdad el desperdicio cuando se llena un vector es tan grande, como para que se necesario vaciar el desperdicio. Allí, veremos en qué casos es necesario usar RemoveWaste y en cuales basta con un RemoveNaive.

Cabe resaltar que se omitió en el experimento la expresion “Capacity +1”, pues esta, como ya se sabe, no deja desperdicio.

En el caso de Capacity *1.5, en ninguno de los 30 tamaños dados, el desperdicio fue suficiente como para que RemoveWaste debiera vaciar el desperdicio. Pero a partir del caso de “Capacity *1.8”, las función Remove Waste empezó a ser de utilidad. En el Caso de Capacity “*1.8” la función realizo vaciado en 24 de 30 casos, en “Capacity *2”, en 26 de 30, y finalmente, en “Capacity *3”, fue necesario hacer vaciado en los 30 casos.

Con esto, podemos llegar a la conclusión de que, a partir de un aumento por 1,8 del tamaño del vector, es donde se marca la diferencia de eficiencia de memoria entre una función a otra, siendo favorable a favor de RemoveWaste. En el caso de *1,5, el efecto es el mismo puesto que el desperdicio no es lo suficientemente grande como para que RemoveWaste haga vaciado del desperdicio, y por tanto, no hay diferencia entre usar una función u otra.