OBSERVACIONES DE LA PRÁCTICA

Luis Alejandro Rodríguez Arenas Cod 202321287
Santiago Rojas Cod 202420928

Ambientes de pruebas

	Máquina 1	Máquina 2	Máquina 3
Procesadores	AMD Ryzen 7 7735HS	AMD Ryzen 5 4600 H	N/A
Memoria RAM (GB)	16	8	N/A
Sistema Operativo	Windows 11	Windows 10	N/A

Tabla 1. Especificaciones de las máquinas para ejecutar las pruebas de rendimiento.

Máquina 1

Resultados para Queue con Array List

Porcentaje de la muestra	enqueue (Array List)	dequeue (Array List)	peek (Array List)
0.50%	0.0368	0.0472	0.0020
5.00%	0.1824	0.2477	0.0012
10.00%	0.5523	0.8540	0.0013
20.00%	0.8764	1.4495	0.0008
30.00%	1.7119	3.0647	0.0013
50.00%	1.8810	5.8165	0.0011
80.00%	3.4504	10.9684	0.0009
100.00%	3.7304	13.5330	0.0013

Resultados para Stack con Array List

Porcentaje de la muestra	push (Array List)	pop (Array List)	top (Array List)
0.50%	0.0284	0.0239	0.0022
5.00%	0.2978	0.2249	0.0013
10.00%	1.0501	0.4535	0.0010
20.00%	3.4431	1.2896	0.0017
30.00%	7.6463	1.9895	0.0017
50.00%	12.3908	4.3894	0.0017
80.00%	32.7458	9.6111	0.0026
100.00%	47.8129	13.5085	0.0018

Resultados para Queue con Linked List

Porcentaje de la muestra	enqueue (Linked List)	dequeue (Linked List)	peek Linked List)
0.50%	0.0477	0.0470	0.0028
5.00%	0.3690	0.3677	0.0015
10.00%	0.6401	0.7094	0.0010
20.00%	1.3532	1.4929	0.0010
30.00%	2.3883	2.2578	0.0022
50.00%	3.7735	6.8759	0.0013
80.00%	6.9614	6.1346	0.0017
100.00%	30.3864	7.8320	0.0021

Resultados para Stack con Linked List

Porcentaje de la muestra	push (Linked List)	pop (Linked List)	top(Linked List)
0.50%	0.0288	0.0381	0.0019
5.00%	0.2441	0.3572	0.0016
10.00%	0.5111	0.6989	0.0009
20.00%	1.6358	1.7344	0.0020
30.00%	1.9544	2.2412	0.0026
50.00%	5.6850	4.1248	0.0042
80.00%	4.9105	6.5219	0.0022
100.00%	7.6838	7.1742	0.0027

Máquina 2

Resultados para Queue con Array List

Porcentaje de la muestra	enqueue (Array List)	dequeue (Array List)	peek (Array List)
0.50%	0,2732	0,0242	0,0017
5.00%	6,4271	0,2287	0,0030
10.00%	32,4831	0,4624	0,0016
20.00%	102,3069	0,9590	0,0017
30.00%	257,6296	1,6218	0,0012
50.00%	713,1702	3,2218	0,0016
80.00%	1.832,7877	6,8016	0,0014
100.00%	2.822,0943	13,1044	0,0015

Resultados para Stack con Array List

Porcentaje de la muestra	push (Array List)	pop (Array List)	top(Array List)
0.50%	0,0812	0,0184	0,0014
5.00%	6,6167	0,2110	0,0030
10.00%	26,3217	0,5415	0,0016
20.00%	109,0285	0,9375	0,0015
30.00%	247,1421	1,5603	0,0015
50.00%	704,3876	4,6179	0,0025
80.00%	1871,5896	7,3275	0,0020
100.00%	3080,5145	9,2425	0,0015

Resultados para Queue con Linked List

Porcentaje de la muestra	enqueue (Linked List)	dequeue (Linked List)	peek Linked List)
0.50%	6,2801	0,1030	0,0048
5.00%	0,2506	0,3545	0,0013
10.00%	0,5213	0,6957	0,0021
20.00%	1,7475	1,3345	0,0014
30.00%	1,4713	1,8240	0,0013
50.00%	2,7343	3,0199	0,0011
80.00%	5,0654	5,3486	0,0012
100.00%	28,0564	6,7705	0,0016

Resultados para Stack con Linked List

Porcentaje de la muestra	push (Linked List)	pop (Linked List)	top(Linked List)
0.50%	0,0597	0,0890	0,0034
5.00%	0,2004	0,3069	0,0026
10.00%	0,3930	0,6224	0,0010
20.00%	0,8339	1,1813	0,0013
30.00%	1,4313	2,0059	0,0019
50.00%	2,0611	2,9496	0,0011
80.00%	4,1326	5,5368	0,0017
100.00%	5,1713	7,0506	0,0021

Preguntas de análisis

1. ¿Se observan diferencias significativas entre las implementaciones con ArrayList y LinkedList para las funciones de Queue y Stack? ¿Cuál es más eficiente en cada operación? ¿Por qué una implementación es más rápida en ciertos casos?

Se observan varias diferencias entre las implementaciones con **ArrayList** y **LinkedList** para ambas funciones.

Es claro que la implementación de **enqueue** con **ArrayList** es más rápida que **enqueue** con **LinkedList**, en este caso se puede atribuir a que, aunque en teoría **enqueue Arraylist** tiene complejidad temporal **O(1)*** mientras que **enqueue LinkedList es O(1)**, esta última probablemente es más lenta debido al manejo de CG en Python, pues durante las pruebas fue necesario aumentar la memoria alocada a este mismo antes de ejecutar una limpieza, pues de lo contrario las pruebas tardaban demasiado. De hecho en el código se implementó una función adicional para poder hacer todas las pruebas de corrido, y esta diferencia de velocidad fue particularmente notoria al usar esta opción, pues el CG de Python no se limpiaba en medio de las ejecuciones.

Otro ejemplo de diferencia fue **dequeue** donde claramente **LinkedList** es mucho más favorable para usar, debido principal mente a que la implementación con **LinkedList** es **O(1)**, pues solo implica eliminar el primer nodo, mientras que en **ArrayList** este implica correr todos los datos, volviéndola **O(n)**.

En el caso de **push** y **pop** el ganador vuelve a ser **LinkedList**, donde de nuevo es O(1), mientras que la implementación con **ArrayList** en el caso de push es **O(1)** y en el caso de pop en **O(n),** aunque cabe resultar que los resultados prácticos de **push** en **ArrayList** sugieren que este es del orden **O(n)** en realidad, debido a la diferencia entre los valores más altos de prueba.

En el caso de las funciones **peek** y **top**, para ambos casos son **O(1)** además de tener tiempo extremadamente constantes, por lo que no hubo diferencia.

2. ¿Cuándo es preferible usar ArrayList o LinkedList? Si insertamos y eliminamos con frecuencia, ¿qué estructura conviene más? Si accedemos aleatoriamente a elementos, ¿cuál es más eficiente?

Definitivamente si queremos acceso aleatorio rápido la mejor implementación es la de **ArrayList**, pues va a ser **O(1)**, ya que esta nos permite acceder de forma directa un índice en específico, a diferencia de **SingleLinked** donde tenemos que acceder a cada nodo hasta llegar al que queremos leer.

Por otro lado para inserciones y eliminaciones frecuentes en extremos **LinkedList** es **O(1)**, por lo que es mucho más beneficioso en el caso de querer insertar y eliminar elementos con frecuencia.

3. Durante la ejecución de las pruebas ¿Se presentan anomalías en los tiempos de ejecución que no se explican con la teoría?

Sí, durante las pruebas se observaron anomalías en los tiempos, principalmente con la operación **enqueue en LinkedList** y **push en ArrayList**.
En el caso de enqueue con LinkedList, aunque en teoría debería ser O(1) y muy competitivo, en la práctica resultó considerablemente más lento que en ArrayList. Esto puede explicarse por detalles de

implementación en Python, específicamente el manejo del **garbage collector (GC)** y la asignación dinámica de memoria. Durante las pruebas, fue necesario ajustar el manejo de memoria para evitar que las ejecuciones se volvieran demasiado lentas, ya que el GC no se limpiaba entre corridas. Esto afectó más a LinkedList porque cada inserción requiere crear un nodo en memoria, lo cual genera más trabajo para el recolector.

Por otro lado, los tiempos de **push en ArrayList** sugieren un comportamiento más cercano a **O(n)** en lugar de O(1) amortizado. Esto se debe a que nuestra pila está implementada con el tope en el primer elemento del ArrayList, lo que implica desplazar todos los elementos en cada inserción o eliminación. En la teoría clásica, si el tope fuera al final, push sería O(1), pero en nuestra implementación los resultados prácticos reflejan el costo de corrimiento de elementos.

En contraste, operaciones como **peek y top** mostraron tiempos extremadamente bajos y constantes, lo cual sí se ajusta perfectamente a la teoría (O(1)), confirmando que las anomalías aparecen sobre todo en las operaciones donde el detalle de implementación importa

4. Complete la siguiente tabla de acuerdo con qué operación es más eficiente en cada implementación (marque con una x la que es más eficiente). Adicionalmente, explique si este comportamiento es acorde con lo enunciado en la teoría. Justifique las respuestas.

		Array List	Linked List	Justificación
QUEUE	**Enqueue()**	x		ArrayList más rápido en la práctica; en teoría ambos son O(1). No se ajusta a la teoría.
	Dequeue()		x	LinkedList es O(1), ArrayList es O(n) por corrimiento de elementos. Se ajusta a la teoría
	Peek()	x	x	Ambos O(1), tiempos casi iguales. Se ajusta a la teoría
STACK	**Push()**		x	LinkedList es O(1); en la implementación ArrayList es O(n) porque inserta al inicio. Se ajusta a la teoría
	Pop()		x	LinkedList O(1); ArrayList O(n) al remover del inicio. Se ajusta a la teoría
	Top()	x	x	Ambos O(1), tiempos iguales. Se ajusta a la teoría.