# Análise Temporal - Projeto 2

**Docente:** João Dias

**Discente:** Diogo Fonseca nº79858

## Problema A – ArrayQueue

**Função:** enqueue()

**Complexidade temporal assintótica:** O(1)

**Testes:**

| n | Tempo Médio de Execução (ms) | Razão |
|---|---|---|
| **250** | 0.0 | 0.0 |
| **500** | 0.0 | 0.0 |
| **1000** | 0.0 | 0.0 |
| **2000** | 0.0 | 0.0 |
| **4000** | 0.0 | 0.0 |
| **8000** | 0.0 | 0.0 |
| **16000** | 0.0 | 0.0 |
| **32000** | 0.0 | 0.0 |
| **64000** | 0.0 | 0.0 |
| **128000** | 0.0 | 0.0 |
| **256000** | 0.0 | 0.0 |
| **512000** | 0.0 | 0.0 |
| **1024000** | 0.0 | 0.0 |
| **2048000** | 0.0 | 0.0 |
| **4096000** | 0.0 | 0.0 |
| **8192000** | 0.0 | 0.0 |

Tabela 1. Ensaios de razão dobrada da função enqueue()

Para cada entrada na tabela 1, n sendo o tamanho da queue, foram realizados 30 testes ao qual o tempo médio de execução é a média geométrica dos tempos dos testes.

A complexidade temporal assintótica da função enqueue() é constante já que o número de operações não cresce (nem diminui) com o tamanho do array. Isto pode ser facilmente verificado olhando para a Tabela 1. O tempo médio de execução é sempre 0 já que esta operação para além de constante, é extremamente eficiente, demorando menos de 1 nanossegundo para terminar a sua execução (de acordo com o relógio interno do java). Isto causa então a razão a ser definida como 0 porque a divisão por 0 é impossível.

Os exemplos gerados para os testes são arrays de tamanho n – 1, ao qual se vai adicionar dar enqueue no último elemento. Os valores dentro dos arrays não interessam, mas estão a ser inteiros gerados aleatoriamente. Conforme o n cresce, o método enqueue() põe o elemento na fila à frente de cada vez mais elementos (à frente de n – 1 elementos).

Estes dados evidenciam que independentemente do tamanho da queue (n), o tempo de execução de adicionar um elemento à queue é constante.

**Função:** dequeue()

**Complexidade temporal assintótica:** O(1)

**Testes:**

| n | Tempo Médio de Execução (ms) | Razão |
|---|---|---|
| **250** | 0.0 | 0.0 |
| **500** | 0.0 | 0.0 |
| **1000** | 0.0 | 0.0 |
| **2000** | 0.0 | 0.0 |
| **4000** | 0.0 | 0.0 |
| **8000** | 0.0 | 0.0 |
| **16000** | 0.0 | 0.0 |
| **32000** | 0.0 | 0.0 |
| **64000** | 0.0 | 0.0 |
| **128000** | 0.0 | 0.0 |
| **256000** | 0.0 | 0.0 |
| **512000** | 0.0 | 0.0 |
| **1024000** | 0.0 | 0.0 |
| **2048000** | 0.0 | 0.0 |
| **4096000** | 0.0 | 0.0 |
| **8192000** | 0.0 | 0.0 |

Tabela 2. Ensaios de razão dobrada da função dequeue()

A complexidade temporal assintótica da função dequeue() é também constante, já que esta funciona simetricamente à função enqueue().

Os exemplos gerados são também idênticos aos da função enqueue() com a ligeira diferença de preencher a queue com n elementos em vez de n – 1.

Os dados da tabela 2, em conjunto com a lógica apresentada evidenciam que o tempo de remover um elemento da queue é constante e independente do número de elementos nela.

# Problema B – Lista Sovina

**Função:** get(index)

**Complexidade temporal assintótica:** O(n)

**Testes:**

| n | Tempo Médio de Execução (ms) | Tempo Médio de Execução (ns) | Razão |
|---|---|---|---|
| **250** | 0.0 | 0 | 0.0 |
| **500** | 0.0 | 0 | 0.0 |
| **1000** | 0.0 | 0 | 0.0 |
| **2000** | 0.0 | 0 | 0.0 |
| **4000** | 0.0 | 0 | 0.0 |
| **8000** | 0.0 | 0 | 0.0 |
| **16000** | 0.0 | 15625 | 0.0 |
| **32000** | 0.0 | 31250 | 2.0 |
| **64000** | 0.1 | 62500 | 2.0 |
| **128000** | 0.1 | 125000 | 2.0 |
| **256000** | 0.3 | 328125 | 2.625 |
| **512000** | 0.7 | 656250 | 2.0 |
| **1024000** | 1.9 | 1859375 | 2.833 |
| **2048000** | 4.4 | 4406250 | 2.370 |
| **4096000** | 9.1 | 9125000 | 2.071 |
| **8192000** | 19.2 | 19171875 | 2.101 |

Tabela 3. Ensaios de razão dobrada da função get(index)

Para cada entrada na tabela 3, n sendo o tamanho da queue, foram realizados 1000 testes ao qual o tempo médio de execução é a média geométrica dos tempos dos testes.

A complexidade temporal assintótica da função get(index) é linear, isto pode ser observado nos testes empíricos na tabela 3: O tempo médio de execução duplica (a razão tende para 2) quando o n duplica. Isto faz o facto da complexidade obvia, mas também podemos fazer o logaritmo de base 2 da razão obtendo 1, ou, linear ($n^1$).

Apesar do tempo de execução médio do get(index) escalar linearmente com o tamanho da lista, a sua complexidade temporal pode ser aproximada a $\sim \frac{n}{4}$, já que este método "corta" a lista em 2, e a complexidade temporal média de ir até ao index seria $\sim \frac{n}{2}$ para ambos os extremos da lista. Por outras palavras, sendo n o tamanho da lista, em média o algoritmo percorre $\frac{n}{4}$ elementos até chegar ao elemento do index recebido.

Os exemplos gerados são listas completamente preenchidas de números aleatórios (o número não vai afetar o tempo de execução). Cada teste vai chamar a função get(index) para um index aleatório.

**Função:** getSlow(index)

**Complexidade temporal assintótica:** O(n)

**Testes:**

| n | Tempo Médio de Execução (ms) | Tempo Médio de Execução (ns) | Razão |
|---|---|---|---|
| **250** | 0.0 | 0 | 0.0 |
| **500** | 0.0 | 0 | 0.0 |
| **1000** | 0.0 | 0 | 0.0 |
| **2000** | 0.0 | 0 | 0.0 |
| **4000** | 0.0 | 0 | 0.0 |
| **8000** | 0.0 | 15625 | 0.0 |
| **16000** | 0.0 | 31250 | 2.0 |
| **32000** | 0.0 | 62500 | 2.0 |
| **64000** | 0.1 | 125000 | 2.0 |
| **128000** | 0.1 | 265625 | 2.125 |
| **256000** | 0.3 | 593750 | 2.235 |
| **512000** | 0.7 | 1421875 | 2.395 |
| **1024000** | 1.9 | 3812500 | 2.681 |
| **2048000** | 4.4 | 8890625 | 2.332 |
| **4096000** | 9.1 | 17906250 | 2.014 |
| **8192000** | 19.2 | 35984375 | 2.010 |

Tabela 4. Ensaios de razão dobrada da função getSlow(index)

Para cada entrada na tabela 4, n sendo o tamanho da queue, foram realizados 1000 testes ao qual o tempo médio de execução é a média geométrica dos tempos dos testes.

A complexidade temporal assintótica da função getSlow(index) é linear, isto pode ser observado nos testes empíricos na tabela 4: O tempo médio de execução duplica (a razão tende para 2) quando o n duplica. Tal como na função get(index), estes dados evidenciam uma complexidade temporal assintótica de O(n).

Tal como a função get(index), apesar da complexidade temporal assintótica ser O(n), a complexidade temporal do caso médio é mais baixa, neste caso é $\sim \frac{n}{2}$ que é o valor médio do index. Em comparação com o get(index), este método em caso médio é teoricamente 2 vezes mais lento. Isto pode ser evidenciado pelos valores obtidos de ambos os métodos: para o mesmo n, podemos observar que o getSlow(index) tem ~2x o valor de get(index).

Os exemplos gerados são iguais aos do get(index). O método de execução do getSlow(index) também é idêntico, gerando um index aleatório cada vez.

**Função:** reverse()

**Complexidade temporal assintótica:** O(1)

**Testes:**

| n | Tempo Médio de Execução (ms) | Razão |
|---|---|---|
| **250** | 0.0 | 0.0 |
| **500** | 0.0 | 0.0 |
| **1000** | 0.0 | 0.0 |
| **2000** | 0.0 | 0.0 |
| **4000** | 0.0 | 0.0 |
| **8000** | 0.0 | 0.0 |
| **16000** | 0.0 | 0.0 |
| **32000** | 0.0 | 0.0 |
| **64000** | 0.0 | 0.0 |
| **128000** | 0.0 | 0.0 |
| **256000** | 0.0 | 0.0 |
| **512000** | 0.0 | 0.0 |
| **1024000** | 0.0 | 0.0 |
| **2048000** | 0.0 | 0.0 |
| **4096000** | 0.0 | 0.0 |
| **8192000** | 0.0 | 0.0 |

Tabela 5. Ensaios de razão dobrada da função reverse ()

Para cada entrada na tabela 5, n sendo o tamanho da queue, foram realizados 1000 testes ao qual o tempo médio de execução é a média geométrica dos tempos dos testes.

A complexidade temporal assintótica da função reverse() é linear, isto é facilmente mostrado pela razão de 0 e pelo tempo médio de execução constante independentemente de n (independentemente de n, a função reverse() faz sempre o mesmo número de operações, sendo só preciso trocar o first e last node numa double linked list).

Os exemplos gerados são mais uma vez uma lista de dimensão n com valores inteiros aleatórios. O método reverse() é chamado uma vez por teste para a lista.