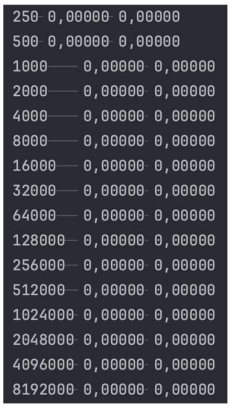
Projeto 2 AED

Octavian Popuiac (nº79911)
José Ferrás (nº67990)
Algoritmos e Estruturas de Dados – Licenciatura Engenharia Informática
Faculdade de Ciência e Tecnologia – Universidade do Algarve
Docente: João Miguel de Sousa de Assis Dias

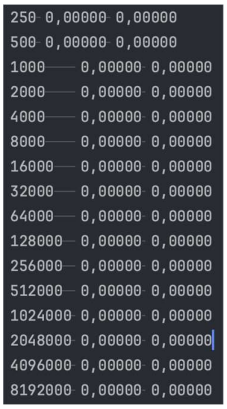
Testes realizados para o problema A (QueueArray)

Foram realizados testes de razão dobrada, para determinar o tempo de execução médio e a ordem de crescimento temporal dos métodos enqueue e dequeue. Os testes em ambos os métodos são executados 30 vezes para cada tamanho individual do array, começam por criar uma fila de qualquer tamanho entre 0 e n-1(sendo n o tamanho do array), sendo realizado um teste para cada lista diferente com o método adequado. Após os testes, é retornado o tempo de execução médio. Depois é calculada a razão dobrada e assim vai se repetindo duplicando sempre o tamanho do array, n.



250	0,00000	0,00000
500	0,00000	0,00000
1000	0,00000	0,00000
2000	0,00000	0,00000
4000	0,00000	0,00000
8000	0,00000	0,00000
16000	0,00000	0,00000
32000	0,00000	0,00000
64000	0,00000	0,00000
128000	0,00000	0,00000
256000	0,00000	0,00000
512000	0,00000	0,00000
1024000	0,00000	0,00000
2048000	0,00000	0,00000
4096000	0,00000	0,00000
8192000	0,00000	0,00000

Exemplo 1



250	0,00000	0,00000
500	0,00000	0,00000
1000	0,00000	0,00000
2000	0,00000	0,00000
4000	0,00000	0,00000
8000	0,00000	0,00000
16000	0,00000	0,00000
32000	0,00000	0,00000
64000	0,00000	0,00000
128000	0,00000	0,00000
256000	0,00000	0,00000
512000	0,00000	0,00000
1024000	0,00000	0,00000
2048000	0,00000	0,00000
4096000	0,00000	0,00000
8192000	0,00000	0,00000

Exemplo 2

- ### Teste para Enqueue:

O Exemplo 1 apresenta os dados dos testes, incluindo os valores de execução médio e da razão dobrada do método enqueue().
Com os valores de tempo de execução médio obtidos, podemos ver que são todos iguais a zero, indicando que o método enqueue() leva sempre o mesmo tempo para ser executado, independentemente do tamanho do array (é constante).
Com base nos resultados dos testes, podemos concluir que a complexidade assintótica do método enqueue() é O(1), pois independentemente do número de dados existentes na fila, o custo de execução será sempre o mesmo.

- ### Teste para Dequeue:

O Exemplo 2 apresenta os dados dos testes, incluindo os valores de execução médio e da razão dobrada do método dequeue().
Com os valores de tempo de execução médio obtidos, podemos ver que são todos iguais a zero, indicando que o método dequeue() leva sempre o mesmo tempo para ser executado, independentemente do tamanho do array (é constante).
Isto faz com que a complexidade assintótica do método dequeue() seja igual à do enqueue(), (O)1.

Testes realizados para o problema B (StingyList)

Para o problema B (Lista Sovina) foram realizados testes de razão dobrada, tal como para o problema A. Estes testes foram realizados para determinar o tempo de execução médio e a ordem de crescimento temporal dos métodos getSlow(), ge()t e reverse().
Os testes de razão dobrada foram realizados da mesma forma que tinham sido para o problema A.

```
250 0.03333333333333333 0.0
500 0.0 0.0
1000    0.03333333333333333 0.0
2000    0.0 0.0
4000    0.0 0.0
8000    0.03333333333333333 0.0
16000   0.03333333333333333 1.0
32000   0.13333333333333333 4.0
64000   0.2 1.5
128000  0.5666666666666667  2.833333333333333
256000  1.0 1.7647058823529411
512000  1.5 1.5
1024000 3.5 2.3333333333333335
2048000 6.5 1.8571428571428572
4096000 9.466666666666667   1.4564102564102563
8192000 24.066666666666666  2.5422535211267605
16384000    47.43333333333333   1.9709141274238227
```

Exemplo 3

```
250 0.0 0.0
500 0.0 0.0
1000    0.03333333333333333 0.0
2000    0.0 0.0
4000    0.0 0.0
8000    0.03333333333333333 0.0
16000   0.0 0.0
32000   0.06666666666666667 0.0
64000   0.1 1.5
128000  0.16666666666666666 1.6666666666666665
256000  0.36666666666666664 2.2
512000  0.7333333333333333  2.0
1024000 1.5 2.0454545454545454
2048000 3.433333333333333   2.2888888888888888
4096000 6.433333333333334   1.8737864077669906
8192000 12.333333333333334  1.9170984455958548
16384000    28.366666666666667  2.3
```

Exemplo 4

- ## Teste para getSlow:

O exemplo 3 apresenta os resultados dos testes, incluindo os valores de execução médio e da razão dobrada do método getSlow().
Podemos obervar que após estabilizar, o valor da razão dobrada dos testes será por volta de 2^1, o que fará com T(n) = n.
Concluimos então que a complexidade assintótica do método getSlow() é de O(n), já que o tempo de execução cresce linearmente com o tamanho da lista.

- ## Teste para get:

O exemplo 4 apresenta os resultados dos testes, incluindo os valores de execução médio e da razão dobrada do método get().
Podemos observar que o valor da razão dobrada dos testes será parecido ao do exemplo anterior (getSlow()), no entanto, se observarmos o tempo de execução médio e comparamos um com o outro, reparamos que o tempo de execução do get() é metade do getSlow(), já que com o get() utilizamos o ultimo nó da lista para percorrer a segunda metade e o primeiro para percorrer a primeira, enquanto que no getSlow() isso não acontece.
Tendo isto em conta, a complexidade assintótica do método get() é de O(n/2) o que na prática será igual a O(n).

```
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
```

Exemplo 5

- Teste para reverse:

O Exemplo 5 apresenta os dados dos testes, incluindo os valores de execução médio e da razão dobrada do método reverse().
Com os valores de tempo de execução médio obtidos, podemos ver que são todos iguais a zero, indicando que o método reverse() leva sempre o mesmo tempo para ser executado, independentemente do tamanho do array (é constante).
Tendo em conta que este método é constante, a complexidade assintótica do mesmo é O(1).

```java
public static void main(String args[]){

    int n = 125;
    double previousTime = calculateAverageExecutionTimeEnqueue(n);  //test enqueue
    //double previousTime = calculateAverageExecutionTimeDequeue(n);    //test dequeue
    double newTime;
    double doublingRatio;
    for(int i = 250; true; i*=2)
    {
        newTime = calculateAverageExecutionTimeEnqueue(i);  //test enqueue
        //newTime = calculateAverageExecutionTimeDequeue(i);  //test dequeue
        if(previousTime > 0)
        {
            doublingRatio = newTime/previousTime;
        }
        else doublingRatio = 0;
        previousTime = newTime;
        System.out.printf("%d\t%.5f\t%.5f\n", i, newTime, doublingRatio);
    }
}
```

Funções testadas:

Problema A (enqueue e dequeue):

```java
public void enqueue(Item item){

    if(this.isFull()){
        throw new OutOfMemoryError( s: "OutOfMemoryError");
    }
    else if(item == null){
        throw new IllegalArgumentException("IllegalArgumentException");
    }
    else{
        this.items[this.available] = item;
        this.available++;
        if(this.available == this.items.length){
            this.available = 0;
        }
    }
}
```

```java
public static void main(String args[]){

public static double calculateAverageExecutionTimeEnqueue(int n)
{
    int trials = 30;
    double totalTime = 0;
    for(int i = 0; i < trials; i++)
    {
        QueueArray<Integer> queueArrayExample = generateQueueArrayExample(n);
        Random r = new Random();
        int x = r.nextInt();
        long time = System.currentTimeMillis();
        queueArrayExample.enqueue(x);
        totalTime += System.currentTimeMillis() - time;
    }
    return totalTime/trials;
}

}
```

```java
public Item dequeue(){
    if(this.isEmpty()){
        return null;
    }
    Item result = this.items[this.firstI];
    this.items[this.firstI] = null;
    this.firstI++;
    if(this.firstI == this.items.length){
        this.firstI = 0;
    }
    return result;
}
```

```java
public static double calculateAverageExecutionTimeDequeue(int n)
{
    int trials = 30;
    double totalTime = 0;
    for(int i = 0; i < trials; i++)
    {
        QueueArray<Integer> queueArrayExample = generateQueueArrayExample(n);
        long time = System.currentTimeMillis();
        queueArrayExample.dequeue();
        totalTime += System.currentTimeMillis() - time;
    }
    return totalTime/trials;
}
```

Problema B(getSlow, get e reverse):

```java
public void reverse()
{
    long tempAddr = this.first;
    this.first = this.last;
    this.last = tempAddr;
}
```

```java
public static double calculateAverageExecutionTimeReverse(int n){
    int trials = 30;
    double totalTime = 0;
    for(int i = 0; i < trials; i++){
        StingyList<Integer> exampleList = generateStingyListExample(n);
        long time = System.currentTimeMillis();
        exampleList.reverse();
        totalTime += System.currentTimeMillis() - time;
    }
    return totalTime/trials;
}
```

```java
public T getSlow(int i)
{
    //Caso em que i não é uma posição válida da lista
    if(i < 0 || i >= this.size){
        throw new IndexOutOfBoundsException();
    }
    long resultAddr = getAddrFirstHalf(i);
    return UNode.get_item(resultAddr);
}
```

```java
public static double calculateAverageExecutionTimeGetSlow(int n){
    int trials = 30;
    double totalTime = 0;
    for(int i = 0; i < trials; i++){
        StingyList<Integer> exampleList = generateStingyListExample(n);
        Random r = new Random();
        int randomI = r.nextInt( origin: 0, n);
        long time = System.currentTimeMillis();
        exampleList.getSlow(randomI);
        totalTime += System.currentTimeMillis() - time;
    }
    return totalTime/trials;
}
```

```java
public T get(int i)
{
    T result;
    //Caso em que i não é uma posição válida da lista
    if(i < 0 || i > this.size-1){
        throw new IndexOutOfBoundsException();
    }
    //Caso em que i se encontra na primeira metade da lista
    else if(i < (this.size/2)){
        result = this.getSlow(i);
    }
    //Caso em que i se encontra na segunda metade da lista
    else{
        long resultAddr = getAddrSecondHalf(i);
        result = UNode.get_item(resultAddr);
    }
    return result;
}
```

```java
public static double calculateAverageExecutionTimeGet(int n){
    int trials = 30;
    double totalTime = 0;
    for(int i = 0; i < trials; i++){
        StingyList<Integer> exampleList = generateStingyListExample(n);
        Random r = new Random();
        int randomI = r.nextInt( origin: 0, n);
        long time = System.currentTimeMillis();
        exampleList.get(randomI);
        totalTime += System.currentTimeMillis() - time;
    }
    return totalTime/trials;
}
```