**Pilha:**

* Complexidade de tempo:
  + **push(int element)**: O(n)
  + **pop()**: O(n)
* Complexidade de espaço:
  + **push(int element)**: O(1)
  + **pop()**: O(1)

**push(int element)**:

Complexidade de tempo: A primeira verificação if (quantityElementsPilha == capacity) é uma operação de tempo constante. Em seguida, há um loop que percorre os elementos do array elements, movendo-os uma posição para a direita. Isso requer um número máximo de iterações igual a quantityElementsPilha. Portanto, a complexidade de tempo desse loop é O(quantityElementsPilha). A atribuição elements[0] = element e o incremento de quantityElementsPilha são operações de tempo constante. No geral, a complexidade de tempo é O(quantityElementsPilha).

Complexidade de espaço: Não há uso de espaço adicional além das variáveis locais, portanto, a complexidade de espaço é constante, O(1).

**pop()**:

Complexidade de tempo: A primeira verificação if (quantityElementsPilha == 0) é uma operação de tempo constante. Em seguida, há um loop que percorre os elementos do array elements, movendo-os uma posição para a esquerda. Isso requer um número máximo de iterações igual a quantityElementsPilha - 1. Portanto, a complexidade de tempo desse loop é O(quantityElementsPilha). O decremento de quantityElementsPilha é uma operação de tempo constante. A segunda verificação if(quantityElementsPilha > 0 && quantityElementsPilha == capacity/4) também é uma operação de tempo constante. No geral, a complexidade de tempo é O(quantityElementsPilha).

Complexidade de espaço: Não há uso de espaço adicional além das variáveis locais, portanto, a complexidade de espaço é constante, O(1).

**Fila:**

* **Complexidade de tempo:**
  + enqueue(int element): O(n)
  + dequeue(): O(1)
  + rear(): O(1)
  + front(): O(1)
* **Complexidade de espaço:**
  + enqueue(int element): O(1)
  + dequeue(): O(1)
  + rear(): O(1)
  + front(): O(1)

**Método enqueue(int element):**

Complexidade de tempo: A primeira verificação if (quantityElements == capacity) é uma operação de tempo constante. Em seguida, há um loop que percorre os elementos do array elements, movendo-os uma posição para a direita. Isso requer um número máximo de iterações igual a quantityElements. Portanto, a complexidade de tempo desse loop é O(quantityElements). A atribuição elements[0] = element e o incremento de quantityElements são operações de tempo constante. No geral, a complexidade de tempo é O(quantityElements).

Complexidade de espaço: Não há uso de espaço adicional além das variáveis locais, portanto, a complexidade de espaço é constante, O(1).

**Método dequeue():**

Complexidade de tempo: A primeira verificação if (quantityElements == 0) é uma operação de tempo constante. A atribuição elements[quantityElements - 1] = 0 e o decremento de quantityElements são operações de tempo constante. A segunda verificação if(quantityElements > 0 && quantityElements <= capacity / 4) também é uma operação de tempo constante. No geral, a complexidade de tempo é O(1).

Complexidade de espaço: Não há uso de espaço adicional além das variáveis locais, portanto, a complexidade de espaço é constante, O(1).

**Método rear():**

Complexidade de tempo: Acessar o elemento elements[0] é uma operação de tempo constante. Portanto, a complexidade de tempo é O(1).

Complexidade de espaço: Não há uso de espaço adicional além das variáveis locais, portanto, a complexidade de espaço é constante, O(1).

**Método front():**

Complexidade de tempo: Acessar o elemento elements[quantityElements-1] é uma operação de tempo constante. Portanto, a complexidade de tempo é O(1).

Complexidade de espaço: Não há uso de espaço adicional além das variáveis locais, portanto, a complexidade de espaço é constante, O(1).

**LinkedList:**

* Complexidade de tempo:
  + **push(Node node)**: O(n)
  + **pop()**: O(n)
  + **insert(int index, Node node)**: O(n)
  + **remove(int index)**: O(n)
  + **elementAt(int index)**: O(n)
* Complexidade de espaço: O(1) para todos os métodos, pois não há uso de espaço adicional além das variáveis locais.

**Método push(Node node):**

Complexidade de tempo: Se a lista estiver vazia (head == null), a atribuição head = node é uma operação de tempo constante, O(1). Caso contrário, o método percorre a lista até o último nó, que requer um número máximo de iterações igual ao tamanho da lista (size). Portanto, a complexidade de tempo desse loop é O(n). No geral, a complexidade de tempo é O(n).

Complexidade de espaço: Não há uso de espaço adicional além das variáveis locais, portanto, a complexidade de espaço é constante, O(1).

**Método pop():**

Complexidade de tempo: O método começa verificando se a lista está vazia (head == null), o que é uma operação de tempo constante. Se a lista tiver apenas um nó (head.getNext() == null), as operações de atribuição e remoção do nó ocorrem em tempo constante. Caso contrário, o método percorre a lista até o penúltimo nó, que requer um número máximo de iterações igual ao tamanho da lista (size). Portanto, a complexidade de tempo desse loop é O(n). No geral, a complexidade de tempo é O(n).

Complexidade de espaço: Não há uso de espaço adicional além das variáveis locais, portanto, a complexidade de espaço é constante, O(1).

**Método insert(int index, Node node):**

Complexidade de tempo: O método começa verificando se o índice está fora dos limites (index < 0 || index > size), o que é uma operação de tempo constante. Se o índice for 0, a atribuição e a inserção do nó ocorrem em tempo constante. Caso contrário, o método percorre a lista até o nó anterior ao índice especificado, que requer um número máximo de iterações igual ao índice (index). Portanto, a complexidade de tempo desse loop é O(n). No geral, a complexidade de tempo é O(n).

Complexidade de espaço: Não há uso de espaço adicional além das variáveis locais, portanto, a complexidade de espaço é constante, O(1).

**Método remove(int index):**

Complexidade de tempo: O método começa verificando se o índice está fora dos limites (index < 0 || index >= size), o que é uma operação de tempo constante. Se o índice for 0, a remoção do primeiro nó ocorre em tempo constante. Caso contrário, o método percorre a lista até o nó anterior ao nó a ser removido, que requer um número máximo de iterações igual ao índice (n). Portanto, a complexidade de tempo desse loop é O(n). No geral, a complexidade de tempo é O(n).

Complexidade de espaço: Não há uso de espaço adicional além das variáveis locais, portanto, a complexidade de espaço é constante, O(1).

**Método elementAt(int index):**

Complexidade de tempo: O método começa verificando se o índice está fora dos limites (index < 0 || index >= size), o que é uma operação de tempo constante. Em seguida, o método percorre a lista até o nó no índice especificado, que requer um número máximo de iterações igual ao índice (index). Portanto, a complexidade de tempo desse loop é O(n). No geral, a complexidade de tempo é O(n).

Complexidade de espaço: Não há uso de espaço adicional além das variáveis locais, portanto, a complexidade de espaço é constante, O(1).