1. Pilha (Lista Encadeada)

Na definição padrão de uma pilha (ou stack), espera-se que os elementos sejam adicionados e removidos no mesmo lado (topo), o que garantiria complexidade constante O(1) tanto para push quanto para pop. No entanto, na minha implementação, utilizei uma lista encadeada onde a remoção de um elemento (pop) exige o percurso da lista até o penúltimo elemento.

Justificativa do push() (O(1)): O push na implementação segue a definição padrão da pilha, pois adicionar um elemento no final da lista encadeada é eficiente, uma vez que mantenho uma referência direta ao último nó da lista. Isso garante complexidade O(1).

Justificativa do pop() (O(n)): Diferentemente da definição padrão, onde o pop é tipicamente O(1) em uma pilha, na minha implementação, a remoção do último elemento da lista exige um percurso da lista até o penúltimo nó, o que tem complexidade O(n). Isso ocorre porque, para redefinir o ponteiro do penúltimo nó como o novo último, é necessário percorrer a lista.

Essa escolha de design se justifica pelo uso de uma lista encadeada simples. Uma implementação mais eficiente de pilha poderia usar uma lista duplamente encadeada ou inserir/remover no início da lista, o que garantiria O(1) para push e pop.

2. Fila (Lista Encadeada)

Na definição padrão de uma fila (ou queue), os elementos são adicionados no final (enqueue) e removidos do início (dequeue), com a expectativa de que ambas as operações sejam feitas em tempo constante O(1). Minha implementação da fila se aproxima dessa definição.

Justificativa do enqueue() (O(1)): A operação de adicionar um elemento no final da fila é O(1) porque mantenho uma referência direta ao último nó da lista encadeada (tail). Isso está em conformidade com a definição padrão da fila.

Justificativa do dequeue() (O(1)): A operação de remover o primeiro elemento também é O(1), pois tenho uma referência direta ao primeiro nó (head), e a remoção não exige a necessidade de percorrer a lista.

Justificativa de rear() e front() (O(1)): Ambos os métodos têm complexidade constante porque uso referências diretas tanto ao início quanto ao fim da lista. Isso está de acordo com a definição padrão de uma fila.

3. Lista Encadeada

Na minha implementação da lista encadeada, as operações são geralmente mais lentas quando comparadas a outras estruturas, como arrays, porque a lista precisa ser percorrida em muitas situações.

Justificativa do push() (O(1)): A adição de um nó ao final da lista encadeada é O(1), uma vez que mantenho uma referência ao último nó (tail). Isso é eficiente e condiz com a definição padrão de uma lista encadeada.

Justificativa do pop() (O(n)): A remoção de um nó do final da lista exige a iteração até o penúltimo nó, resultando em uma complexidade O(n), uma vez que a lista encadeada simples não possui uma referência ao penúltimo nó. Isso difere de implementações mais sofisticadas, como listas duplamente encadeadas, que poderiam reduzir essa operação para O(1).

Justificativa do insert() (O(n)): Inserir em uma posição arbitrária na lista encadeada requer o percurso até o índice correto, o que resulta em complexidade O(n). Isso é típico em listas encadeadas, já que, ao contrário de arrays, não há acesso direto por índice.

Justificativa do remove() (O(n)): A remoção de um nó em um índice arbitrário segue a mesma lógica do insert(), onde é necessário percorrer a lista até o nó anterior ao que será removido, resultando em O(n).

Justificativa do elementAt() (O(n)): A obtenção de um elemento por índice em uma lista encadeada simples também tem complexidade O(n), já que, diferentemente de arrays, a lista precisa ser percorrida sequencialmente até o índice desejado.

Justificativa da Complexidade de Espaço

Em todas as implementações, a complexidade de espaço é O(n), onde n é o número de nós na estrutura. Isso ocorre porque cada nó precisa armazenar um valor inteiro e uma referência para o próximo nó (no caso de listas encadeadas simples). Essa é uma característica padrão das listas encadeadas e não difere da implementação tradicional.

Conclusão Geral

A principal diferença entre minhas implementações e as definições padrão das estruturas de dados (especialmente no caso da pilha) está relacionada à eficiência de remoção de elementos. Enquanto as implementações padrão de pilha e fila têm operações de remoção eficientes, o uso de uma lista encadeada simples impacta a complexidade de tempo de algumas operações, como pop() na pilha e a manipulação de nós no final da lista encadeada. Uma otimização seria usar uma lista duplamente encadeada, ou no caso da pilha, ajustar a estrutura para que as operações de inserção e remoção ocorram no mesmo lado (início ou fim).