Collections LinkedList

Derick Josue Falkowski

Jean Felipe Moreira

Kayc Matheus Chauchuti

Matheus Feliphe Dias dos Santos

UNIVERSIDADE POSITIVO
ADS
Prof. Rhafael Freitas da Costa

DESCRIÇÃO DO TIPO DE COLLECTION

LinkedList é uma classe em Java que implementa a interface List e utiliza uma estrutura de dados de lista duplamente encadeada.

Cada elemento na lista é armazenado em um nó que possui referências para o próximo e o anterior na lista.

PRINCIPAIS CARACTERÍSTICAS E USO ADEQUADO

- Inserção rápida e remoção de elementos no início e no fim da lista.
- Acesso lento a elementos em posições arbitrárias, pois requer percorrer a lista do início até a posição desejada.
- Baixa eficiência em operações que exigem acesso aleatório, como `get(int index)` e `remove(int index)`.
- O uso adequado da LinkedList é quando a inserção e remoção frequente de elementos ocorre no início ou no fim da lista e quando a ordem dos elementos é importante.

IMPLEMENTAÇÕES DISPONÍVEIS

Quando dizemos que a LinkedList faz parte da biblioteca padrão de Java, queremos dizer que não é necessário baixar ou instalar nada extra para utilizá-la. Ela já está incluída no conjunto de bibliotecas que são fornecidas automaticamente quando você instala ou usa a JDK (Java Development Kit) ou a JRE (Java Runtime Environment).

OPERAÇÕES COMUNS E COMPLEXIDADE TEMPORAL

Inserção no Início (addFirst):

- Complexidade Temporal: O(1)
- Adicionar um elemento no início da lista encadeada é uma operação de tempo constante, pois envolve apenas a criação de um novo nó e ajuste dos ponteiros.

Inserção no Final (addLast ou add sem índice específico):

- Complexidade Temporal: O(1)
- Adicionar um elemento no final da lista é uma operação de tempo constante, pois a LinkedList mantém uma referência direta para o último nó.

OPERAÇÕES COMUNS E COMPLEXIDADE TEMPORAL

Inserção em uma Posição Específica (add com índice específico):

- Complexidade Temporal: O(n)
- Adicionar um elemento em uma posição específica na lista requer percorrer os nós até encontrar o local correto. Portanto, a complexidade temporal é linear, dependendo do tamanho da lista.

Remoção no Início (removeFirst):

- Complexidade Temporal: O(1)
- Remover o primeiro elemento da lista é uma operação de tempo constante, pois envolve apenas a atualização dos ponteiros.

OPERAÇÕES COMUNS E COMPLEXIDADE TEMPORAL

Remoção no Final (removeLast):

- Complexidade Temporal: O(1)
- Remover o último elemento da lista também é uma operação de tempo constante, pois a LinkedList mantém uma referência direta para o último nó.

Remoção em uma Posição Específica (remove com índice específico):

- Complexidade Temporal: O(n)
- Remover um elemento em uma posição específica requer percorrer os nós até encontrar o elemento desejado. Portanto, a complexidade temporal é linear, dependendo do tamanho da lista.

OPERAÇÕES COMUNS E COMPLEXIDADE TEMPORAL

Acesso a um Elemento por Índice (get):

- Complexidade Temporal: O(n)
- Acessar um elemento por índice requer percorrer os nós até chegar ao índice desejado. Portanto, a complexidade temporal é linear, dependendo do tamanho da lista.

Verificação de Presença de um Elemento (contains):

- Complexidade Temporal: O(n)
- Verificar se um elemento está presente na lista requer percorrer todos os nós até encontrar o elemento desejado. Portanto, a complexidade temporal é linear, dependendo do tamanho da lista.

EXEMPLOS DE CÓDIGO DEMONSTRANDO

Main.java

```java
import java.util.LinkedList;

public class Main {
    public static void main(String[] args) {
        // Criando uma LinkedList
        LinkedList<String> linkedList = new LinkedList<>();

        // Adicionando elementos
        linkedList.add("Maçã");
        linkedList.add("Banana");
        linkedList.add("Laranja");

        // Acessando elementos
        System.out.println("Primeiro elemento: " + linkedList.getFirst());
        System.out.println("Último elemento: " + linkedList.getLast());

        // Removendo o primeiro e o último elemento
        linkedList.removeFirst();
        linkedList.removeLast();

        // Iterando sobre os elementos
        for (String fruit : linkedList) {
            System.out.println(fruit);
        }
    }
}
```

BOAS PRÁTICAS E DICAS RELEVANTES

- Evite acessar elementos aleatórios em uma LinkedList devido à sua baixa eficiência nesse tipo de operação.
- Considere o uso de LinkedList quando a ordem dos elementos é importante e quando há inserção e remoção frequente no início ou no fim da lista.
- Lembre-se de que LinkedList consome mais memória que ArrayList devido à necessidade de armazenar referências para o próximo e o anterior em cada nó.

LinkedList

Preparação de Exercícios:

1. Escreva um método para inverter uma LinkedList.

1. Crie um método para encontrar o k-ésimo elemento a partir do final em uma LinkedList.

1. Implemente uma função para mesclar duas LinkedLists ordenadas em uma única LinkedList ordenada.