Aula: Pilhas, Listas e Filas em Java (Aprox. 2h10min)

Introdução

Bem-vindos à nossa aula sobre Pilhas, Listas e Filas em Java. Hoje, vamos expandir o conhecimento que vocês já possuem sobre estruturas de dados, especificamente sobre a classe Node e ListaLigada, para explorar estruturas mais avançadas e suas aplicações práticas.

As estruturas de dados são fundamentais na ciência da computação e no desenvolvimento de software, pois permitem o armazenamento e manipulação eficiente de dados em memória. Compreender profundamente essas estruturas e suas implementações é essencial para desenvolver soluções eficientes e escaláveis.

Antes de mergulharmos nos novos conceitos, vamos fazer uma breve revisão dos conceitos fundamentais que vocês já conhecem. A classe Node representa um nó em uma estrutura de dados encadeada, contendo um valor e uma referência para o próximo nó. A ListaLigada, por sua vez, é uma coleção de nós conectados, onde cada nó aponta para o próximo, formando uma sequência.

Durante esta aula, vamos explorar três estruturas de dados principais: Pilhas, Listas avançadas e Filas. Veremos tanto os aspectos teóricos quanto as implementações práticas em Java, além de discutir aplicações reais dessas estruturas.

Pilhas (Stacks)

Conceito e Definição

Uma pilha é uma estrutura de dados linear que segue o princípio LIFO (Last In, First Out), ou seja, o último elemento inserido é o primeiro a ser removido. Podemos visualizar uma pilha como uma pilha de pratos: só podemos adicionar ou remover pratos do topo da pilha.

O princípio LIFO governa todas as operações em uma pilha, tornando-a uma estrutura de dados com acesso restrito, onde apenas o elemento do topo está acessível a qualquer momento. Esta característica torna as pilhas particularmente úteis em cenários onde a ordem de processamento inversa é necessária.

Operações Básicas

As pilhas possuem quatro operações fundamentais:

1. **Push**: Adiciona um elemento ao topo da pilha. Esta operação aumenta o tamanho da pilha em uma unidade.
2. **Pop**: Remove o elemento do topo da pilha e o retorna. Esta operação diminui o tamanho da pilha em uma unidade. Se a pilha estiver vazia, uma exceção é geralmente lançada.
3. **Peek** (ou Top): Retorna o elemento do topo da pilha sem removê-lo. A pilha permanece inalterada após esta operação.
4. **isEmpty**: Verifica se a pilha está vazia. Retorna verdadeiro se a pilha não contiver elementos, e falso caso contrário.

Aplicações Práticas

As pilhas são amplamente utilizadas em diversos contextos computacionais:

- **Avaliação de expressões**: Pilhas são usadas para avaliar expressões matemáticas, especialmente aquelas em notação polonesa reversa.
- **Gerenciamento de chamadas de função**: O sistema operacional utiliza uma pilha de chamadas para rastrear as funções em execução e seus contextos.
- **Desfazer/Refazer operações**: Editores de texto e aplicativos gráficos usam pilhas para implementar funcionalidades de desfazer e refazer.
- **Verificação de parênteses balanceados**: Algoritmos que verificam se expressões têm parênteses, colchetes e chaves balanceados utilizam pilhas.
- **Algoritmos de busca em profundidade**: Em grafos, a busca em profundidade (DFS) é implementada usando pilhas para rastrear os nós a serem visitados.

Complexidade das Operações

As operações em uma pilha têm complexidade de tempo constante O(1), independentemente do tamanho da pilha, quando implementadas corretamente. Isso significa que push, pop, peek e isEmpty são operações muito eficientes.

Implementação de Pilha usando Array

Vamos agora examinar como implementar uma pilha usando um array em Java. Esta é uma das abordagens mais comuns e eficientes para implementar pilhas.

```
/**
 * Implementação de uma Pilha usando array em Java
 */
public class PilhaArray<T> {
    private static final int CAPACIDADE_INICIAL = 10;
    private Object[] elementos;
    private int topo;

    public PilhaArray() {
        this(CAPACIDADE_INICIAL);
    }

    public PilhaArray(int capacidade) {
        elementos = new Object[capacidade];
        topo = -1;
    }

    public boolean isEmpty() {
        return topo == -1;
    }

    public boolean isFull() {
        return topo == elementos.length - 1;
    }

    public int size() {
        return topo + 1;
    }

    public void push(T elemento) {
        if (isFull()) {
            redimensionar();
        }
        elementos[++topo] = elemento;
    }

    @SuppressWarnings("unchecked")
    public T pop() {
        if (isEmpty()) {
            throw new IllegalStateException("A pilha está
vazia");
        }
        T elemento = (T) elementos[topo];
        elementos[topo--] = null; // Ajuda o garbage collector
        return elemento;
```

```
    }

    @SuppressWarnings("unchecked")
    public T peek() {
        if (isEmpty()) {
            throw new IllegalStateException("A pilha está
vazia");
        }
        return (T) elementos[topo];
    }

    private void redimensionar() {
        int novaCapacidade = elementos.length * 2;
        Object[] novoArray = new Object[novaCapacidade];
        System.arraycopy(elementos, 0, novoArray, 0,
elementos.length);
        elementos = novoArray;
    }

    public void clear() {
        for (int i = 0; i <= topo; i++) {
            elementos[i] = null;
        }
        topo = -1;
    }

    @Override
    public String toString() {
        if (isEmpty()) {
            return "[]";
        }
        StringBuilder sb = new StringBuilder("[");
        for (int i = 0; i <= topo; i++) {
            sb.append(elementos[i]);
            if (i < topo) {
                sb.append(", ");
            }
        }
        sb.append("]");
        return sb.toString();
    }
}
```

Nesta implementação, utilizamos um array para armazenar os elementos da pilha e um índice `topo` para rastrear a posição do elemento no topo da pilha. A pilha começa vazia, com `topo = -1`. Quando adicionamos um elemento, incrementamos `topo` e armazenamos o elemento nessa posição. Quando removemos um elemento, retornamos o elemento na posição `topo` e decrementamos `topo`.

Uma característica importante desta implementação é o método `redimensionar()`, que dobra o tamanho do array quando ele está cheio. Isso permite que a pilha cresça dinamicamente conforme necessário, evitando a limitação de tamanho fixo.

Implementação de Pilha usando ListaLigada

Agora, vamos examinar como implementar uma pilha usando uma lista ligada. Esta abordagem oferece algumas vantagens em relação à implementação com array, como não precisar se preocupar com redimensionamento.

```java
/**
 * Implementação de uma Pilha usando ListaLigada em Java
 */
public class PilhaListaLigada<T> {
    private class Node {
        T data;
        Node next;

        public Node(T data) {
            this.data = data;
            this.next = null;
        }
    }

    private Node topo;
    private int tamanho;

    public PilhaListaLigada() {
        topo = null;
        tamanho = 0;
    }

    public boolean isEmpty() {
        return topo == null;
    }

    public int size() {
        return tamanho;
    }

    public void push(T elemento) {
        Node novoNode = new Node(elemento);
        novoNode.next = topo;
        topo = novoNode;
        tamanho++;
    }

    public T pop() {
        if (isEmpty()) {
```

```
            throw new IllegalStateException("A pilha está
vazia");
        }
        T elemento = topo.data;
        topo = topo.next;
        tamanho--;
        return elemento;
    }

    public T peek() {
        if (isEmpty()) {
            throw new IllegalStateException("A pilha está
vazia");
        }
        return topo.data;
    }

    public void clear() {
        topo = null;
        tamanho = 0;
    }

    @Override
    public String toString() {
        if (isEmpty()) {
            return "[]";
        }
        StringBuilder sb = new StringBuilder("[");
        Node atual = topo;
        while (atual != null) {
            sb.append(atual.data);
            if (atual.next != null) {
                sb.append(", ");
            }
            atual = atual.next;
        }
        sb.append("]");
        return sb.toString();
    }
}
```

Nesta implementação, cada elemento da pilha é representado por um nó que contém o dado e uma referência para o próximo nó. O topo da pilha é representado pela variável `topo`, que aponta para o primeiro nó da lista. Quando adicionamos um elemento, criamos um novo nó, fazemos ele apontar para o nó atual do topo e atualizamos o topo para apontar para o novo nó. Quando removemos um elemento, retornamos o dado do nó do topo e atualizamos o topo para apontar para o próximo nó.

Comparação entre as Implementações de Pilha

Ambas as implementações têm vantagens e desvantagens:

Pilha com Array: - **Vantagens**: Acesso direto aos elementos, menor overhead de memória por elemento (sem ponteiros). - **Desvantagens**: Necessidade de redimensionamento (que pode ser custoso), possível desperdício de espaço se a capacidade for muito maior que o uso.

Pilha com Lista Ligada: - **Vantagens**: Tamanho dinâmico sem necessidade de redimensionamento, uso eficiente de memória (aloca apenas o necessário). - **Desvantagens**: Maior overhead por nó devido às referências, acesso sequencial aos elementos (embora não seja um problema para operações de pilha).

A escolha entre essas implementações depende do contexto específico da aplicação, considerando fatores como frequência de operações, restrições de memória e previsibilidade do tamanho máximo da pilha.

Listas Avançadas

Agora que exploramos as pilhas, vamos avançar para tipos mais complexos de listas encadeadas. Vocês já estão familiarizados com listas simplesmente encadeadas, onde cada nó contém um dado e uma referência para o próximo nó. Vamos explorar estruturas mais avançadas: listas duplamente encadeadas e listas circulares.

Tipos de Listas Encadeadas

Além da lista simplesmente encadeada que vocês já conhecem, existem outros tipos importantes de listas:

1. **Lista Duplamente Encadeada**: Cada nó contém referências tanto para o próximo nó quanto para o nó anterior. Isso permite a navegação em ambas as direções, facilitando operações como remoção de elementos e navegação reversa.

2. **Lista Circular**: O último nó da lista aponta de volta para o primeiro, formando um ciclo. Isso elimina a necessidade de verificações de fim de lista em muitos algoritmos e permite a implementação eficiente de estruturas como buffers circulares.

3. **Lista Circular Duplamente Encadeada**: Combina as características das listas duplamente encadeadas e circulares, permitindo navegação bidirecional e cíclica.

Operações Básicas em Listas Avançadas

As listas encadeadas suportam várias operações fundamentais:

1. **Inserção**: Adicionar elementos no início, no final ou em uma posição específica da lista.
2. **Remoção**: Remover elementos do início, do final ou de uma posição específica da lista.
3. **Busca**: Localizar um elemento específico na lista.
4. **Travessia**: Percorrer todos os elementos da lista, geralmente para realizar alguma operação em cada um deles.

Vantagens e Desvantagens em Relação a Arrays

Vantagens das Listas Encadeadas: - Tamanho dinâmico: Podem crescer ou diminuir conforme necessário, sem necessidade de realocação. - Inserção e remoção eficientes: Especialmente no início da lista ou em posições conhecidas (O(1) se a referência for conhecida). - Não requerem espaço contíguo em memória.

Desvantagens das Listas Encadeadas: - Acesso aleatório ineficiente: Para acessar o n-ésimo elemento, é necessário percorrer a lista desde o início (O(n)). - Maior consumo de memória: Cada elemento requer espaço adicional para armazenar referências. - Não aproveitam a localidade de cache, o que pode resultar em desempenho inferior em alguns cenários comparado a arrays.

Complexidade das Operações em Listas

A complexidade das operações em listas encadeadas varia:

- **Acesso (get)**: O(n) no pior caso.
- **Busca (contains)**: O(n) no pior caso.
- **Inserção/Remoção no início**: O(1) para listas simplesmente e duplamente encadeadas.
- **Inserção/Remoção no final**: O(n) para listas simplesmente encadeadas (sem tail), O(1) para listas duplamente encadeadas (com tail) e listas circulares (com referência ao último).
- **Inserção/Remoção em posição arbitrária (dado o índice)**: O(n) no pior caso (para encontrar a posição).
- **Inserção/Remoção em posição arbitrária (dada a referência ao nó)**: O(1) para listas duplamente encadeadas, O(n) para listas simplesmente encadeadas (para encontrar o anterior).

Implementação de Lista Duplamente Encadeada

Vamos examinar como implementar uma lista duplamente encadeada em Java:

```java
/**
 * Implementação de uma Lista Duplamente Encadeada em Java
 */
public class ListaDuplamenteEncadeada<T> {
    private class Node {
        T data;
        Node next;
        Node prev;

        public Node(T data) {
            this.data = data;
            this.next = null;
            this.prev = null;
        }
    }

    private Node head; // Referência para o primeiro nó
    private Node tail; // Referência para o último nó
    private int tamanho;

    public ListaDuplamenteEncadeada() {
        head = null;
        tail = null;
        tamanho = 0;
    }

    public boolean isEmpty() {
        return head == null;
    }

    public int size() {
        return tamanho;
    }

    public void addFirst(T elemento) {
        Node novoNode = new Node(elemento);
        if (isEmpty()) {
            head = novoNode;
            tail = novoNode;
        } else {
            novoNode.next = head;
            head.prev = novoNode;
            head = novoNode;
        }
        tamanho++;
    }
```

```
    public void addLast(T elemento) {
        Node novoNode = new Node(elemento);
        if (isEmpty()) {
            head = novoNode;
            tail = novoNode;
        } else {
            tail.next = novoNode;
            novoNode.prev = tail;
            tail = novoNode;
        }
        tamanho++;
    }

    public void add(T elemento, int indice) {
        if (indice < 0 || indice > tamanho) {
            throw new IndexOutOfBoundsException("Índice
inválido: " + indice);
        }
        if (indice == 0) {
            addFirst(elemento);
            return;
        }
        if (indice == tamanho) {
            addLast(elemento);
            return;
        }
        Node atual = getNodeAt(indice);
        Node novoNode = new Node(elemento);
        novoNode.next = atual;
        novoNode.prev = atual.prev;
        atual.prev.next = novoNode;
        atual.prev = novoNode;
        tamanho++;
    }

    public T removeFirst() {
        if (isEmpty()) {
            throw new IllegalStateException("A lista está
vazia");
        }
        T elemento = head.data;
        if (head == tail) {
            head = null;
            tail = null;
        } else {
            head = head.next;
            head.prev = null;
        }
        tamanho--;
        return elemento;
    }
```

```
    public T removeLast() {
        if (isEmpty()) {
            throw new IllegalStateException("A lista está
vazia");
        }
        T elemento = tail.data;
        if (head == tail) {
            head = null;
            tail = null;
        } else {
            tail = tail.prev;
            tail.next = null;
        }
        tamanho--;
        return elemento;
    }

    public T remove(int indice) {
        if (indice < 0 || indice >= tamanho) {
            throw new IndexOutOfBoundsException("Índice
inválido: " + indice);
        }
        if (indice == 0) {
            return removeFirst();
        }
        if (indice == tamanho - 1) {
            return removeLast();
        }
        Node atual = getNodeAt(indice);
        atual.prev.next = atual.next;
        atual.next.prev = atual.prev;
        tamanho--;
        return atual.data;
    }

    public T get(int indice) {
        if (indice < 0 || indice >= tamanho) {
            throw new IndexOutOfBoundsException("Índice
inválido: " + indice);
        }
        return getNodeAt(indice).data;
    }

    private Node getNodeAt(int indice) {
        if (indice < tamanho / 2) {
            Node atual = head;
            for (int i = 0; i < indice; i++) {
                atual = atual.next;
            }
            return atual;
        } else {
            Node atual = tail;
```

```
            for (int i = tamanho - 1; i > indice; i--) {
                atual = atual.prev;
            }
            return atual;
        }
    }

    public boolean contains(T elemento) {
        Node atual = head;
        while (atual != null) {
            if ((elemento == null && atual.data == null) ||
                (elemento != null &&
elemento.equals(atual.data))) {
                return true;
            }
            atual = atual.next;
        }
        return false;
    }

    public void clear() {
        head = null;
        tail = null;
        tamanho = 0;
    }

    @Override
    public String toString() {
        if (isEmpty()) {
            return "[]";
        }
        StringBuilder sb = new StringBuilder("[");
        Node atual = head;
        while (atual != null) {
            sb.append(atual.data);
            if (atual.next != null) {
                sb.append(", ");
            }
            atual = atual.next;
        }
        sb.append("]");
        return sb.toString();
    }

    public String toStringReverse() {
        if (isEmpty()) {
            return "[]";
        }
        StringBuilder sb = new StringBuilder("[");
        Node atual = tail;
        while (atual != null) {
            sb.append(atual.data);
```

```
            if (atual.prev != null) {
                sb.append(", ");
            }
            atual = atual.prev;
        }
        sb.append("]");
        return sb.toString();
    }
}
```

A principal diferença entre uma lista duplamente encadeada e uma lista simplesmente encadeada é que cada nó contém referências tanto para o próximo nó quanto para o nó anterior. Isso permite a navegação em ambas as direções, o que é particularmente útil para operações como remoção de elementos (que não requer mais a busca pelo nó anterior) e navegação reversa.

Além disso, mantemos uma referência para o último nó da lista (`tail`), o que permite operações eficientes no final da lista, como adicionar ou remover o último elemento em tempo constante O(1).

Uma otimização importante nesta implementação é o método `getNodeAt()`, que decide se percorre a lista a partir do início ou do fim, dependendo da posição do elemento desejado. Isso reduz o tempo médio de acesso aos elementos da lista.

Implementação de Lista Circular

Agora, vamos examinar como implementar uma lista circular em Java:

```
/**
 * Implementação de uma Lista Circular em Java
 */
public class ListaCircular<T> {
    private class Node {
        T data;
        Node next;

        public Node(T data) {
            this.data = data;
            this.next = null;
        }
    }

    private Node ultimo;  // Referência para o último nó
    private int tamanho;

    public ListaCircular() {
        ultimo = null;
```

```
        tamanho = 0;
    }

    public boolean isEmpty() {
        return ultimo == null;
    }

    public int size() {
        return tamanho;
    }

    public void addFirst(T elemento) {
        Node novoNode = new Node(elemento);
        if (isEmpty()) {
            ultimo = novoNode;
            ultimo.next = ultimo; // Aponta para si mesmo
        } else {
            novoNode.next = ultimo.next;
            ultimo.next = novoNode;
        }
        tamanho++;
    }

    public void addLast(T elemento) {
        addFirst(elemento);
        if (tamanho > 1) { // Só precisa atualizar o último se
houver mais de um nó
            ultimo =
ultimo.next; // Move a referência do último para o novo nó
        }
    }

    public void add(T elemento, int indice) {
        if (indice < 0 || indice > tamanho) {
            throw new IndexOutOfBoundsException("Índice
inválido: " + indice);
        }
        if (indice == 0) {
            addFirst(elemento);
            return;
        }
        if (indice == tamanho) {
            addLast(elemento);
            return;
        }
        Node anterior = getNodeAt(indice - 1);
        Node novoNode = new Node(elemento);
        novoNode.next = anterior.next;
        anterior.next = novoNode;
        tamanho++;
    }
```

```
    public T removeFirst() {
        if (isEmpty()) {
            throw new IllegalStateException("A lista está
vazia");
        }
        T elemento = ultimo.next.data;
        if (ultimo.next == ultimo) {
            ultimo = null;
        } else {
            ultimo.next = ultimo.next.next;
        }
        tamanho--;
        return elemento;
    }

    public T removeLast() {
        if (isEmpty()) {
            throw new IllegalStateException("A lista está
vazia");
        }
        T elemento = ultimo.data;
        if (ultimo.next == ultimo) {
            ultimo = null;
        } else {
            Node penultimo = getNodeAt(tamanho - 2);
            penultimo.next = ultimo.next;
            ultimo = penultimo;
        }
        tamanho--;
        return elemento;
    }

    public T remove(int indice) {
        if (indice < 0 || indice >= tamanho) {
            throw new IndexOutOfBoundsException("Índice
inválido: " + indice);
        }
        if (indice == 0) {
            return removeFirst();
        }
        if (indice == tamanho - 1) {
            return removeLast();
        }
        Node anterior = getNodeAt(indice - 1);
        T elemento = anterior.next.data;
        anterior.next = anterior.next.next;
        tamanho--;
        return elemento;
    }

    public T get(int indice) {
        if (indice < 0 || indice >= tamanho) {
```

```
            throw new IndexOutOfBoundsException("Índice
inválido: " + indice);
        }
        return getNodeAt(indice).data;
    }

    private Node getNodeAt(int indice) {
        if (isEmpty() || indice < 0 || indice >= tamanho) {
             throw new IndexOutOfBoundsException("Índice
inválido ou lista vazia");
        }
        Node atual = ultimo.next; // Começa do primeiro nó
        for (int i = 0; i < indice; i++) {
            atual = atual.next;
        }
        return atual;
    }

    public boolean contains(T elemento) {
        if (isEmpty()) {
            return false;
        }
        Node atual = ultimo.next; // Começa do primeiro nó
        do {
            if ((elemento == null && atual.data == null) ||
                (elemento != null &&
elemento.equals(atual.data))) {
                return true;
            }
            atual = atual.next;
        } while (atual != ultimo.next); // Continua até voltar
ao início
        return false;
    }

    public void rotate() {
        if (!isEmpty() && tamanho > 1) {
            ultimo = ultimo.next;
        }
    }

    public void clear() {
        ultimo = null;
        tamanho = 0;
    }

    @Override
    public String toString() {
        if (isEmpty()) {
            return "[]";
        }
        StringBuilder sb = new StringBuilder("[");
```

```
        Node atual = ultimo.next; // Começa do primeiro nó
        do {
            sb.append(atual.data);
            atual = atual.next;
            if (atual != ultimo.next) {
                sb.append(", ");
            }
        } while (atual != ultimo.next); // Continua até voltar
ao início
        sb.append("]");
        return sb.toString();
    }
}
```

A principal característica de uma lista circular é que o último nó aponta de volta para o primeiro, formando um ciclo. Isso elimina a necessidade de verificações de fim de lista em muitos algoritmos e permite a implementação eficiente de estruturas como buffers circulares.

Nesta implementação, mantemos apenas uma referência para o último nó da lista (`ultimo`), e o primeiro nó é acessado através de `ultimo.next`. Isso permite operações eficientes tanto no início quanto no final da lista.

Uma operação interessante em listas circulares é a rotação, implementada pelo método `rotate()`, que move o primeiro elemento para o final da lista (ou, equivalentemente, move a "cabeça" da lista para o próximo elemento). Esta operação é muito eficiente em listas circulares, requerendo apenas a atualização da referência `ultimo`.

Ao percorrer uma lista circular, é importante usar um loop do-while em vez de um loop while convencional, para garantir que o primeiro nó seja processado antes de verificar se voltamos ao início.

Comparação entre os Tipos de Listas

Cada tipo de lista tem suas próprias vantagens e desvantagens:

Lista Simplesmente Encadeada: - **Vantagens**: Implementação simples, menor overhead de memória. - **Desvantagens**: Navegação apenas em uma direção, remoção ineficiente (requer acesso ao nó anterior).

Lista Duplamente Encadeada: - **Vantagens**: Navegação em ambas as direções, remoção eficiente de qualquer nó (dada a referência). - **Desvantagens**: Maior overhead de memória devido à referência adicional, implementação mais complexa.

Lista Circular: - **Vantagens**: Acesso eficiente ao início e fim da lista (com referência ao último), facilita implementação de algoritmos cíclicos (ex: round-robin). - **Desvantagens**: Requer cuidado para evitar loops infinitos, implementação pode ser um pouco mais complexa que a simples.

A escolha entre esses tipos de listas depende das necessidades específicas da aplicação, considerando fatores como padrões de acesso, frequência de inserções e remoções, e restrições de memória.

Filas (Queues)

Conceito e Definição

Após explorarmos pilhas e listas, vamos agora nos aprofundar em outra estrutura de dados linear fundamental: a fila (queue). Diferente da pilha, que segue o princípio LIFO, a fila opera sob o princípio FIFO (First In, First Out), ou seja, o primeiro elemento inserido é o primeiro a ser removido. Podemos visualizar uma fila como uma fila de pessoas esperando para serem atendidas: a primeira pessoa a chegar é a primeira a ser atendida.

O princípio FIFO governa todas as operações em uma fila. Elementos são adicionados no final da fila (enfileirados) e removidos do início da fila (desenfileirados). Essa característica torna as filas ideais para cenários onde a ordem de chegada é importante e o processamento deve ocorrer na mesma sequência.

Operações Básicas

As filas possuem quatro operações fundamentais:

1. **Enqueue** (ou Offer, Add): Adiciona um elemento ao final da fila. Esta operação aumenta o tamanho da fila em uma unidade.

2. **Dequeue** (ou Poll, Remove): Remove o elemento do início da fila e o retorna. Esta operação diminui o tamanho da fila em uma unidade. Se a fila estiver vazia, uma exceção ou um valor nulo é geralmente retornado.

3. **Peek** (ou Element): Retorna o elemento do início da fila sem removê-lo. A fila permanece inalterada após esta operação. Se a fila estiver vazia, uma exceção ou um valor nulo é geralmente retornado.

4. **isEmpty**: Verifica se a fila está vazia. Retorna verdadeiro se a fila não contiver elementos, e falso caso contrário.

Aplicações Práticas

As filas são amplamente utilizadas em diversos contextos computacionais:

- **Gerenciamento de processos**: Sistemas operacionais utilizam filas para escalonar processos que aguardam acesso à CPU ou outros recursos.
- **Buffers de dados**: Em comunicação de dados e I/O, filas são usadas como buffers para armazenar dados temporariamente enquanto são transferidos entre processos ou dispositivos com velocidades diferentes.
- **Simulações**: Filas são essenciais em modelos de simulação para representar entidades esperando por serviço (por exemplo, clientes em um banco, carros em um pedágio).
- **Algoritmos de busca em largura**: Em grafos, a busca em largura (BFS) é implementada usando filas para rastrear os nós a serem visitados, garantindo a exploração nível por nível.
- **Sistemas de mensagens**: Filas de mensagens (Message Queues) são usadas para comunicação assíncrona entre diferentes partes de um sistema distribuído.
- **Impressão em rede**: Spoolers de impressão utilizam filas para gerenciar os trabalhos de impressão enviados por múltiplos usuários.

Complexidade das Operações em Filas

Quando implementadas corretamente (por exemplo, usando uma lista ligada com referências para o início e o fim, ou um array circular), as operações básicas em uma fila (enqueue, dequeue, peek, isEmpty) têm complexidade de tempo constante O(1). Isso as torna estruturas de dados muito eficientes para cenários que exigem processamento FIFO.

Implementação de Fila usando ListaLigada

Uma forma comum e eficiente de implementar uma fila é usando uma lista ligada, mantendo referências para o início e o fim da lista.

```
/**
 * Implementação de uma Fila (Queue) usando ListaLigada em Java
 */
public class FilaListaLigada<T> {
    private class Node {
        T data;
        Node next;
```

```
        public Node(T data) {
            this.data = data;
            this.next = null;
        }
    }

    private Node inicio; // Referência para o início da fila
(primeiro elemento)
    private Node fim;    // Referência para o fim da fila
(último elemento)
    private int tamanho;

    public FilaListaLigada() {
        inicio = null;
        fim = null;
        tamanho = 0;
    }

    public boolean isEmpty() {
        return inicio == null;
    }

    public int size() {
        return tamanho;
    }

    public void enqueue(T elemento) {
        Node novoNode = new Node(elemento);
        if (isEmpty()) {
            inicio = novoNode;
            fim = novoNode;
        } else {
            fim.next = novoNode;
            fim = novoNode;
        }
        tamanho++;
    }

    public T dequeue() {
        if (isEmpty()) {
            throw new IllegalStateException("A fila está
vazia");
        }
        T elemento = inicio.data;
        inicio = inicio.next;
        tamanho--;
        if (isEmpty()) {
            fim = null;
        }
        return elemento;
    }
```

```
    public T peek() {
        if (isEmpty()) {
            throw new IllegalStateException("A fila está
vazia");
        }
        return inicio.data;
    }

    public void clear() {
        inicio = null;
        fim = null;
        tamanho = 0;
    }

    @Override
    public String toString() {
        if (isEmpty()) {
            return "[]";
        }
        StringBuilder sb = new StringBuilder("[");
        Node atual = inicio;
        while (atual != null) {
            sb.append(atual.data);
            if (atual.next != null) {
                sb.append(", ");
            }
            atual = atual.next;
        }
        sb.append("]");
        return sb.toString();
    }
}
```

Nesta implementação, mantemos duas referências: `inicio` para o primeiro nó (de onde os elementos são removidos) e `fim` para o último nó (onde os elementos são adicionados). Isso permite que as operações `enqueue` e `dequeue` sejam realizadas em tempo constante O(1).

Implementação de Fila usando Array Circular

Outra abordagem eficiente é usar um array circular. Isso evita o overhead de memória dos ponteiros da lista ligada, mas requer gerenciamento dos índices e redimensionamento.

```
/**
 * Implementação de uma Fila (Queue) usando array circular em
Java
```

```java
 */
public class FilaArray<T> {
    private static final int CAPACIDADE_INICIAL = 10;
    private Object[] elementos;
    private int inicio; // Índice do primeiro elemento
    private int fim;    // Índice da próxima posição livre
    private int tamanho; // Número de elementos na fila

    public FilaArray() {
        this(CAPACIDADE_INICIAL);
    }

    public FilaArray(int capacidade) {
        elementos = new Object[capacidade];
        inicio = 0;
        fim = 0;
        tamanho = 0;
    }

    public boolean isEmpty() {
        return tamanho == 0;
    }

    public boolean isFull() {
        return tamanho == elementos.length;
    }

    public int size() {
        return tamanho;
    }

    public void enqueue(T elemento) {
        if (isFull()) {
            redimensionar();
        }
        elementos[fim] = elemento;
        fim = (fim + 1) % elementos.length; // Avança o fim
circularmente
        tamanho++;
    }

    @SuppressWarnings("unchecked")
    public T dequeue() {
        if (isEmpty()) {
            throw new IllegalStateException("A fila está
vazia");
        }
        T elemento = (T) elementos[inicio];
        elementos[inicio] = null; // Ajuda o garbage collector
        inicio = (inicio + 1) % elementos.length; // Avança o
início circularmente
        tamanho--;
```

```
        return elemento;
    }

    @SuppressWarnings("unchecked")
    public T peek() {
        if (isEmpty()) {
            throw new IllegalStateException("A fila está
vazia");
        }
        return (T) elementos[inicio];
    }

    private void redimensionar() {
        int novaCapacidade = elementos.length * 2;
        Object[] novoArray = new Object[novaCapacidade];
        for (int i = 0; i < tamanho; i++) {
            int indiceAntigo = (inicio + i) % elementos.length;
            novoArray[i] = elementos[indiceAntigo];
        }
        elementos = novoArray;
        inicio = 0;
        fim = tamanho;
    }

    public void clear() {
        for (int i = 0; i < tamanho; i++) {
            int indice = (inicio + i) % elementos.length;
            elementos[indice] = null;
        }
        inicio = 0;
        fim = 0;
        tamanho = 0;
    }

    @Override
    public String toString() {
        if (isEmpty()) {
            return "[]";
        }
        StringBuilder sb = new StringBuilder("[");
        for (int i = 0; i < tamanho; i++) {
            int indice = (inicio + i) % elementos.length;
            sb.append(elementos[indice]);
            if (i < tamanho - 1) {
                sb.append(", ");
            }
        }
        sb.append("]");
        return sb.toString();
    }
}
```

Nesta implementação, usamos índices `inicio` e `fim` que se movem circularmente dentro do array usando o operador módulo (`%`). Isso permite reutilizar o espaço no início do array quando elementos são removidos, tornando o uso da memória mais eficiente do que um array simples. O redimensionamento é necessário quando o array fica cheio.

Comparação entre as Implementações de Fila

Fila com Lista Ligada: - **Vantagens**: Tamanho dinâmico sem redimensionamento, implementação relativamente simples das operações O(1). - **Desvantagens**: Overhead de memória dos ponteiros.

Fila com Array Circular: - **Vantagens**: Menor overhead de memória por elemento, boa localidade de cache. - **Desvantagens**: Necessidade de redimensionamento, gerenciamento de índices mais complexo.

A escolha depende das mesmas considerações feitas para pilhas: frequência de operações, restrições de memória e previsibilidade do tamanho.

Aplicações Práticas

Pilhas em Ação

Exemplo: Verificação de Parênteses Balanceados

Um problema clássico que pode ser resolvido eficientemente usando pilhas é verificar se uma expressão tem parênteses, colchetes e chaves balanceados. Por exemplo, a expressão `{[()]}` está balanceada, mas `{[(])}` não está.

```
// (Código da função verificaParentesesBalanceados já
apresentado anteriormente)
public static boolean verificaParentesesBalanceados(String
expressao) {
    PilhaListaLigada<Character> pilha = new
PilhaListaLigada<>();
    for (char c : expressao.toCharArray()) {
        if (c == '(' || c == '[' || c == '{') {
            pilha.push(c);
        } else if (c == ')' || c == ']' || c == '}') {
            if (pilha.isEmpty()) return false;
            char topo = pilha.pop();
            if ((c == ')' && topo != '(') || (c == ']' && topo !
= '[') || (c == '}' && topo != '{')) {
                return false;
            }
```

```
        }
    }
    return pilha.isEmpty();
}
```

Listas em Ação

Exemplo: Histórico de Navegação (Simplificado)

Um histórico de navegação pode ser implementado usando uma lista duplamente encadeada para permitir avançar e voltar entre as páginas visitadas.

```
public class HistoricoNavegacao {
    private ListaDuplamenteEncadeada<String> historico = new
ListaDuplamenteEncadeada<>();
    private ListaDuplamenteEncadeada<String>.Node paginaAtual;

    public void visitar(String url) {
        // Se estivermos no meio do histórico, remove as páginas
futuras
        while (paginaAtual != null && paginaAtual.next != null)
{
            historico.removeLast(); // Simplificado - ideal
seria remover a partir do nó atual
        }

        historico.addLast(url);
        paginaAtual = historico.tail; // Atualiza para a última
página visitada
        System.out.println("Visitando: " + url);
    }

    public String voltar() {
        if (paginaAtual != null && paginaAtual.prev != null) {
            paginaAtual = paginaAtual.prev;
            System.out.println("Voltando para: " +
paginaAtual.data);
            return paginaAtual.data;
        } else {
            System.out.println("Não há página anterior.");
            return null;
        }
    }

    public String avancar() {
        if (paginaAtual != null && paginaAtual.next != null) {
            paginaAtual = paginaAtual.next;
            System.out.println("Avançando para: " +
paginaAtual.data);
```

```
            return paginaAtual.data;
        } else {
            System.out.println("Não há página seguinte.");
            return null;
        }
    }
    // Nota: Esta é uma implementação simplificada. Uma real
precisaria de acesso
    // direto aos nós internos da lista, o que exigiria
modificar a classe ListaDuplamenteEncadeada
    // ou usar a implementação LinkedList nativa do Java.
}
```

Filas em Ação

Exemplo: Simulação de Fila de Atendimento

Podemos simular uma fila de atendimento onde clientes chegam e são atendidos na ordem de chegada.

```
public static void simularFilaAtendimento() {
    FilaListaLigada<String> filaClientes = new
FilaListaLigada<>();
    int tempoSimulacao = 10; // Simular por 10 unidades de tempo
    double probChegada = 0.6; // Probabilidade de um cliente
chegar a cada unidade de tempo
    int tempoAtendimento = 3; // Tempo para atender um cliente
    int tempoRestanteAtendimento = 0;
    String clienteAtual = null;

    System.out.println("--- Iniciando Simulação de Fila ---");

    for (int tempo = 1; tempo <= tempoSimulacao; tempo++) {
        System.out.println("\nTempo: " + tempo);

        // Chegada de cliente?
        if (Math.random() < probChegada) {
            String novoCliente = "Cliente_" + tempo;
            filaClientes.enqueue(novoCliente);
            System.out.println("  -> " + novoCliente +
" chegou. Fila: " + filaClientes);
        }

        // Atendimento
        if (clienteAtual == null && !filaClientes.isEmpty()) {
            clienteAtual = filaClientes.dequeue();
            tempoRestanteAtendimento = tempoAtendimento;
            System.out.println("  <- Atendendo " + clienteAtual
+ ". Fila: " + filaClientes);
```

```
        }

        if (clienteAtual != null) {
            tempoRestanteAtendimento--;
            if (tempoRestanteAtendimento == 0) {
                System.out.println("  ** " + clienteAtual + "
terminou o atendimento.");
                clienteAtual = null;
            } else {
                System.out.println("     Atendendo " +
clienteAtual + " (" + tempoRestanteAtendimento + " restante)");
            }
        }

        if (clienteAtual == null && filaClientes.isEmpty()) {
             System.out.println("     Guichê livre e fila
vazia.");
        }
    }

    System.out.println("\n--- Fim da Simulação ---");
    if (clienteAtual != null) {
        System.out.println("Cliente em atendimento: " +
clienteAtual);
    }
    System.out.println("Clientes restantes na fila: " +
filaClientes);
}
```

Considerações de Desempenho

Ao escolher entre pilhas, listas e filas, e suas respectivas implementações, é crucial considerar os requisitos específicos da aplicação:

- **Padrão de Acesso**: Se a ordem LIFO é necessária, use uma Pilha. Se a ordem FIFO é necessária, use uma Fila. Se o acesso aleatório ou inserções/remoções em qualquer posição são frequentes, uma Lista (ou um `ArrayList` / `LinkedList` nativo do Java) pode ser mais apropriada.
- **Frequência de Operações**: Pilhas e Filas oferecem operações de adição/remoção em O(1) (nas extremidades apropriadas). Listas encadeadas oferecem O(1) para adição/remoção nas extremidades (com as referências corretas), mas O(n) para acesso/busca/remoção por índice. Arrays (ou `ArrayList`) oferecem acesso O(1) por índice, mas adição/remoção no meio pode ser O(n).

- **Memória**: Listas ligadas (usadas em Pilhas, Filas ou Listas) têm overhead de ponteiros. Arrays (usados em Pilhas, Filas ou `ArrayList`) podem ter desperdício de espaço ou custo de redimensionamento.
- **Implementação**: Usar as classes nativas do Java (`Stack`, `LinkedList` como `Queue` ou `Deque`, `ArrayList`, `LinkedList`) é geralmente recomendado em produção, pois são otimizadas e robustas. Implementar do zero é valioso para aprendizado.

A escolha da estrutura de dados correta pode ter um impacto significativo no desempenho e na complexidade do código, tornando-se um aspecto crucial do design de software eficiente.

Resumo e Conclusão

Nesta aula, exploramos três estruturas de dados lineares fundamentais: pilhas, listas e filas. Vimos como as pilhas seguem o princípio LIFO, as filas seguem o princípio FIFO, e as listas oferecem flexibilidade para inserção, remoção e acesso em diferentes posições. Analisamos suas operações básicas, complexidades, implementações comuns (usando arrays e listas ligadas) e diversas aplicações práticas.

Compreender essas estruturas de dados é essencial para desenvolver soluções eficientes e escaláveis. Cada estrutura tem suas próprias vantagens e desvantagens, e a escolha entre elas, bem como a implementação específica, depende dos requisitos da aplicação.

Espero que esta aula tenha fornecido uma compreensão sólida dessas estruturas de dados e suas implementações em Java. Continuem praticando e explorando essas estruturas em seus próprios projetos para consolidar o conhecimento.

Referências e Recursos Adicionais

1. Cormen, T. H., Leiserson, C. E., Rivest, R. L., & Stein, C. (2009). Introduction to Algorithms (3rd ed.). MIT Press.
2. Sedgewick, R., & Wayne, K. (2011). Algorithms (4th ed.). Addison-Wesley Professional.
3. Java Documentation: Collections Framework (Inclui `Stack`, `Queue`, `Deque`, `List`, `LinkedList`, `ArrayList`)

4. Goodrich, M. T., Tamassia, R., & Goldwasser, M. H. (2014). Data Structures and Algorithms in Java (6th ed.). Wiley.

5. Weiss, M. A. (2011). Data Structures and Algorithm Analysis in Java (3rd ed.). Pearson.

6. Visualização de Estruturas de Dados: VisuAlgo

7. Exercícios Práticos: LeetCode, LeetCode, LeetCode, HackerRank