UniEinstein
CENTRO UNIVERSITÁRIO | COLÉGIO

Listas Encadeadas Duplas e Circulares e Pilhas

Docente: Leandro Lopes Taveira
Coordenador: Pedro Ivo Garcia Nunes

Revisão - Listas Encadeadas Simples

Operações

- `addFirst()`: O(1)
- `addLast()`: O(n)
- `removeFirst()`: O(1)
- `remove()`: O(n)
- `search()`: O(n)

```
[Dado|Next] -> [Dado|Next] -> null
```

Limitação

- Navegação apenas para frente.
- Remover o nó anterior a um dado nó é custoso.

Listas Encadeadas Duplas

Conceito

- Cada nó tem **dois ponteiros**:
  - next: para o próximo nó
  - prev: para o nó anterior

Visualização

```
null <- [Dado|Prev|Next] <-> [Dado|Prev|Next] <-> [Dado|Prev|Next] -> null
```

Implementação

```
public class NoDuplo {
    private int codigo;
    private String nome;
    NoDuplo next;
    NoDuplo prev;

    public void preecher(int codigo, String nome) {
        this.codigo = codigo;
        this.nome = nome;
        this.next = null;
        this.prev = null;
    }
}
```

Operações em Listas Duplas

Vantagens

- **Navegação bidirecional:** Para frente e para trás.
- **Remoção mais eficiente:** Acessar o nó anterior é O(1).

Operações em Listas Duplas

Inserção no Início (`addFirst`)

```
// Novo nó aponta para o head atual
// Head atual aponta para o novo nó (prev)
// Head vira o novo nó
```

Inserção no Fim (`addLast`)

```
// Novo nó aponta para o tail atual
// Tail atual aponta para o novo nó (next)
// Tail vira o novo nó
```

Listas Encadeadas Circulares

Conceito

- O **último nó aponta para o primeiro nó**.
- Forma um **círculo**.

Tipos

- **Simplesmente Circular:** Ponteiro `next` do último para o primeiro.
- **Duplamente Circular:** Ponteiros `next` e `prev` formam o círculo.

Listas Encadeadas Circulares

Visualização (Simplesmente Circular)

```
[Dado|Next] -> [Dado|Next] -> [Dado|Next]

   ^-------------------------------|
```

Vantagens

- Pode-se **percorrer a lista a partir de qualquer nó**.
- Útil para **filas de tarefas** ou **buffers circulares**.

Comparativo de Listas Encadeadas

Característica	Simples	Dupla	Circular
Navegação	Unidirecional	Bidirecional	Unidirecional/Bidirecional
Ponteiros por Nó	1 (`next`)	2 (`next`, `prev`)	1 ou 2
Overhead	Menor	Maior	Depende
Remoção de Anterior	O(n)	O(1)	O(n) ou O(1)
Acesso ao Último	O(n)	O(1)	O(1) (se ponteiro para último)

Introdução a Pilhas (Stack)

Conceito

- Estrutura de dados linear.
- Princípio **LIFO** (Last In, First Out).
    - O último elemento a entrar é o primeiro a sair.

Analogia

- Pilha de pratos: O último prato colocado é o primeiro a ser retirado.
- Pilha de livros, pilha de bandejas.

Operações Principais da Pilha

`push(elemento)`

- Adiciona um elemento ao **topo** da pilha.

`pop()`

- Remove e retorna o elemento do **topo** da pilha.

`peek()`

- Retorna o elemento do **topo** sem removê-lo.

`isEmpty()`

- Verifica se a pilha está **vazia**.

`size()`

- Retorna o **número de elementos** na pilha.

Implementação da Classe Pilha (Estrutura Básica)

Conceito

- Implementação usando Lista Encadeada
- **Vantagens:** Tamanho dinâmico, sem limite de elementos (memória).
- **Desvantagens:** Maior overhead de memória por nó.

Visualização

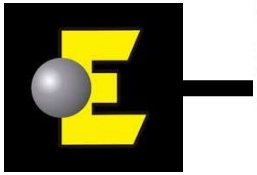
```
topo -> [Dado|Next] -> [Dado|Next] -> [Dado|Next] -> null
```

Vamos precisar de uma classe de Cliente

```java
public class Cliente {  8 usages
    private int codigo;  3 usages
    private String nome;  3 usages
    private String CPF;  3 usages

    public void preecher(int codigo, String nome, String CPF){  no usages
        this.codigo = codigo;
        this.nome = nome;
        this.CPF = CPF;
    }

    public int getCodigo() { return codigo; }
    public void setCodigo(int codigo) { this.codigo = codigo; }
    public String getNome() { return nome; }
    public void setNome(String nome) { this.nome = nome; }
    public String getCPF() { return CPF; }
    public void setCPF(String CPF) { this.CPF = CPF; }
}
```



Como fica então nosso Nó?

```
public class No {  2 usages
    Cliente cliente;  1 usage
    No next;  1 usage

    public void preencher(Cliente cliente) {  no usages
        this.cliente = cliente;
        this.next = null;
    }
}
```

Estrutura Básica da Pilha

Visualização

```
topo -> [Cliente|Next] -> [Cliente|Next] -> [Cliente|Next] -> null
```

```
public class Pilha {  no usages
    private No topo; // Equivalente ao head da lista  no usages

    // Métodos serão detalhados nos próximos slides
}
```

Operação: push(elemento)

Descrição

- Adiciona um novo cliente ao **topo** da pilha.
- Passos:
  1. Criar um novo nó com o cliente fornecido
  2. Fazer o novo nó apontar para o topo atual
  3. Atualizar o topo para ser o novo nó

Exemplo Visual

```
Antes do push(c1):

topo -> [Cliente(102)|•] -> [Cliente(103)|•] -> null

Depois do push(c1):

  topo -> [Cliente(101)|•] -> [Cliente(102)|•] ->
  [Cliente(103)|•] -> null
```

Método: `public void push(Cliente cliente) { /*code*/}`

Código do Método push()

```
public void push(Cliente cliente) {  no usages
    No novoNo = new No();
    novoNo.preencher(cliente);
    novoNo.next = topo;
    topo = novoNo;
}
```

Operação: isEmpty()

Descrição

- Verifica se a pilha está **vazia**.
- Passos:
  1. Verificar se o topo é nulo
  2. Retornar verdadeiro se for nulo, falso caso contrário

Exemplo Visual

```
Pilha vazia:

topo -> null


Pilha com clientes:

topo -> [Cliente(102)|•] ->
[Cliente(103)|•] -> null
```

Método: `public booblean isEmpty() { /*code*/}`

Código do Método isEmpty()

Observações

- Método simples mas essencial para evitar erros.
- Usado internamente pelos métodos pop() e peek().
- A operação é sempre de tempo constante.

```
public boolean isEmpty() {  1 usage
    if(topo == null){
        return true;
    }
    return false;
}
```

Operação: pop()

Descrição

- Remove e retorna o cliente do **topo** da pilha.
- Passos:
  1. Verificar se a pilha está vazia (Retornar `null` e estiver)
  2. Armazenar o cliente do topo
  3. Atualizar o topo para o próximo nó
  4. Retornar o cliente armazenado

Exemplo Visual

```
Antes do pop():

topo -> [Cliente(101)|•] ->
[Cliente(102)|•] -> [Cliente(103)|•] ->
null

Depois do pop():

  topo -> [Cliente(102)|•] ->
  [Cliente(103)|•] -> null
```

Método: `public Cliente pop() { /*code*/}`

Código do Método pop()

```
public Cliente pop() {  no usages
    if (isEmpty()) {
        return null;
    }
    Cliente cliente = topo.cliente;
    topo = topo.next;
    return cliente;
}
```

Operação: peek()

Descrição

- Retorna o cliente do **topo** da pilha sem removê-lo.
- Passos:
  1. Verificar se a pilha está vazia (Retornar `null` e estiver)
  2. Retornar o cliente do topo

Exemplo Visual

```
Antes do peek():

topo -> [Cliente(102)|•] ->
[Cliente(103)|•] -> null

Depois do peek():

  topo -> [Cliente(102)|•] ->
  [Cliente(103)|•] -> null  // Sem
  alteração na pilha
```

Método: `public Cliente peek() { /*code*/}`

Código do Método peek()

Observações

- Não altera a estrutura da pilha.
- Útil para verificar o próximo cliente a ser processado.
- A operação é sempre de tempo constante.

```
public Cliente peek() {  no usages
    if (isEmpty()) {
        return null;
    }
    return topo.cliente;
}
```

Implementação do Método size()

Descrição

- Retorna o **número de clientes** na pilha.
- Passos:
  1. Inicializar contador como 0
  2. Percorrer a pilha do topo até o final
  3. Incrementar contador para cada nó
  4. Retornar o contador

Exemplo Visual

```
Pilha:

topo -> [Cliente(101)|•] ->
[Cliente(102)|•] -> [Cliente(103)|•]
-> null

  size() retorna: 3
```

Método: `public int size() { /*code*/}`

Código do Método size()

```
public int size() {  no usages
    int count = 0;
    No atual = topo;

    while (atual != null) {
        count++;
        atual = atual.next;
    }

    return count;
}
```

Métodos de suporte ao desenvolvedor.

- Podemos criar métodos de suporte ao desenvolvimento.
- Devem ser removidos ao final do desenvolvimento.
- Para facilitar, mantenha sempre no início do fim da classe.

Implementação do método showStack()

Descrição

- Imprime todos os elementos da Pilha.
- Passos:
  1. Percorrer a pilha do topo até o final
  2. Imprime as informações de cada cliente na ordem.

Método: `public void showStack() { /*code*/}`

Operação: showStack()

```
public void showStack(){  3 usages
    System.out.println("---- Inicio Pilha ----");
    No atual = topo;
    while (atual != null) {
        System.out.println("["+atual.cliente.getCodigo() +" | "+atual.cliente.getNome()
                +" | "+ atual.cliente.getCPF()+" ]");
        atual = atual.next;
    }
    System.out.println("---- Fim Pilha ----");
}
```

Exercício: Adicionar Funcionalidade - Método clear()

Objetivo

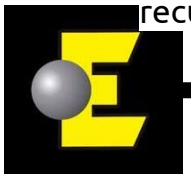
Implementar um método na classe `Pilha` que remova todos os elementos, deixando-a vazia.

Descrição

Crie um método `public void clear()` na sua classe `Pilha`. Este método deve esvaziar a pilha, ou seja, após sua execução, a pilha não deve conter nenhum `Cliente` e o método `isEmpty()` deve retornar `true`.

Dicas

- Pense em como o `topo` da pilha deve se comportar para indicar uma pilha vazia.
- Não é necessário iterar sobre os elementos para removê-los um por um, a menos que você queira liberar recursos explicitamente



Exercício: Funcionalidade - Método contains(Cliente cliente)

Objetivo

Implementar um método que verifique se um determinado `Cliente` está presente na pilha.

Descrição

Crie um método `public boolean contains(Cliente cliente)` na sua classe `Pilha`. Este método deve percorrer a pilha e retornar `true` se o `Cliente` passado como parâmetro for encontrado, e `false` caso contrário.

Dicas

- Você precisará iterar sobre os nós da pilha, começando pelo `topo`.
- Use o CPF que é um identificador único de cada cliente.

Conclusão: Próximos Passos e Desafios

Recapitulando

- A **Pilha** é uma estrutura de dados fundamental, seguindo o princípio **LIFO** (Last In, First Out).
- Sua implementação com **Listas Encadeadas** oferece flexibilidade e dinamismo.
- A capacidade de trabalhar com **objetos complexos** (como `Cliente`) a torna aplicável em cenários reais.

Desafios e Reflexões

- **Otimização:** Como você otimizaria o método `size()` para ter complexidade O(1)?
- **Outras Estruturas:** Em quais cenários uma **Fila** (FIFO) seria mais adequada que uma Pilha?
- **Tratamento de Erros:** Explore formas de tratamento de erros.


STACKS
Querem mais exercícios?
Push/Pop
LIFO Principle
Push/Pop

Exercício: Uso da Pilha - Inverter Ordem de Clientes

Objetivo

Criar uma função externa que inverte a ordem dos `Clientes` em uma `Pilha`.

Descrição

Escreva um método em uma NOVA classe que você vai criar. Este método recebe uma `Pilha` de `Clientes` como parâmetro e retorne uma **nova** `Pilha` com os clientes na ordem inversa. A pilha original não deve ser modificada.

Dicas

- Você pode precisar de uma ou mais pilhas auxiliares para realizar a inversão.
- Lembre-se das operações básicas da pilha: `push()`, `pop()`, `peek()`, `isEmpty()`.

Exercício: Uso da Pilha - Histórico de Navegação de Clientes

Objetivo

Simular um histórico de navegação de clientes em um sistema, utilizando a `Pilha`.

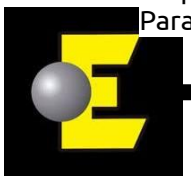
Descrição

Crie uma classe `HistoricoNavegacao` que utilize uma `Pilha` para armazenar os `Clientes` que acessaram determinadas páginas ou funcionalidades. Implemente os seguintes métodos:

- `acessarPagina(Cliente cliente)`: Adiciona o cliente ao histórico (topo da pilha).
- `voltarPagina()`: Remove o cliente atual do topo e retorna o cliente anterior (simulando o botão 'voltar').
- `paginaAtual()`: Retorna o cliente que está no topo do histórico (página atual).
- `exibirHistorico()`: Imprime todos os clientes no histórico, do mais recente para o mais antigo, sem remover da pilha.

Dicas

- A operação `voltarPagina()` deve lançar uma exceção ou retornar `null` se não houver páginas anteriores.
Para `exibirHistorico()`, você pode usar uma pilha auxiliar para manter a ordem original da pilha principal.



Exercício: Uso da Pilha - Processamento de Pedidos Prioritários

Objetivo: Simular um sistema de processamento de pedidos onde os pedidos de clientes são tratados com base em uma ordem específica, utilizando a `Pilha`.

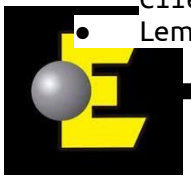
Descrição: Considere que você tem uma lista de pedidos e precisa processá-los. No entanto, alguns pedidos são prioritários e devem ser processados antes dos pedidos normais. Utilize duas pilhas: uma para pedidos normais e outra para pedidos prioritários.

Implemente uma classe `ProcessadorPedidos` com os seguintes métodos:

- `adicionarPedido(Cliente cliente, boolean prioritario)`: Adiciona um cliente à pilha apropriada.
- `processarProximoPedido()`: Remove e retorna o próximo cliente a ser processado. Deve sempre priorizar a pilha de pedidos prioritários. Se a pilha prioritária estiver vazia, então processa da pilha normal.
- `pedidosPendentes()`: Retorna o número total de pedidos aguardando processamento.

Dicas

- A classe `Cliente` pode ser usada para representar o pedido, ou você pode criar uma nova classe `Pedido` que contenha um `Cliente` e outras informações.
- Lembre-se de verificar se as pilhas estão vazias antes de tentar `pop()`.



Exercício Prático: Sistema de Atendimento com Pilha

Tarefa: Implemente um sistema simples de atendimento de clientes usando a pilha:

1. Crie uma classe `SistemaAtendimento` que use a `Pilha` para gerenciar clientes.
2. Implemente os métodos:
    - `adicionarCliente(Cliente c)`: Adiciona um cliente à pilha
    - `atenderProximo()`: Remove e retorna o próximo cliente a ser atendido
    - `consultarProximo()`: Mostra o próximo cliente sem removê-lo
    - `clientesEmEspera()`: Retorna o número de clientes aguardando
3. Teste o sistema com pelo menos 5 clientes diferentes.

Dica:

- Lembre-se que a pilha segue o princípio LIFO (Last In, First Out).
- Considere se a pilha é a estrutura mais adequada para um sistema de atendimento real.