UniEinstein
CENTRO UNIVERSITÁRIO | COLÉGIO

Listas Encadeadas Simples

Docente: Leandro Lopes Taveira
Coordenador: Pedro Ivo Garcia Nunes

Revisão - Arrays

Vantagens dos Arrays

✅ Acesso rápido: O(1) por índice

✅ Simplicidade: Fácil de usar

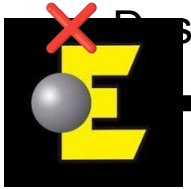
✅ Eficiência de cache: Elementos contíguos

Desvantagens dos Arrays

❌ Tamanho fixo: Não pode redimensionar

❌ Inserção/Remoção lenta: O(n) no meio

❌ Desperdício de memória: Pode alocar mais que necessário



Motivação para Listas Encadeadas

Problemas dos Arrays

- E se não soubermos o tamanho antecipadamente?
- E se precisarmos inserir/remover frequentemente?
- E se o tamanho variar muito durante a execução?

Solução: Listas Encadeadas

- Tamanho dinâmico
- Inserção/Remoção mais eficiente
- Uso flexível de memória

Conceito de Lista Encadeada

Definição

- **Coleção de elementos (nós)** conectados por ponteiros
- **Cada nó contém:**
  - **Dado:** Informação armazenada
  - **Ponteiro:** Referência para o próximo nó

Características

- **Não contíguos** na memória
- **Acesso sequencial** (do primeiro ao último)

Estrutura Visual

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

     Nó 1           Nó 2           Nó 3
```

Estrutura do Nó

```java
public class No {
    private int codigo;   // Dado armazenado
    private String nome;  // Dado armazenado

    No next;      // Referência para o próximo nó

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

    public int getCodigo(){
        return this.codigo;
    }
}
```

Utilizando o nó

```
No no1 = new No();
no1.preencher(10, "joaozinho");
No no2 = new No();
no2.preencher(20, "manoelzinho");
No no3 = new No();
no3.preencher(30, "mariazinha");

no1.next = no2;  // no1 aponta para no2
no2.next = no3;  // no2 aponta para no3
// no3.next já é null (fim da lista)
```

Estrutura da Lista

Classe base

Contém os controles da lista.

- **Inserção no Início**
- **Inserção no Fim**
- **Busca de Elemento**
- **Remoção do Início**
- **Remoção de Elemento Específico**

Classe que vai controlar a lista.

```
public class ListaEncadeadaSimples {
    private No head = null;  // Primeiro nó da lista

    public boolean isEmpty() {
        if(this.head == null){
            return true;
        }
        return false;
    }
}
```

Inserção no Início

Algoritmo

1. **Criar novo nó** com o dado
2. **Fazer o novo nó apontar** para o head atual
3. **Atualizar o head** para o novo nó

Visualização

```
Antes:  head -> [20|Next] -> [30|null]
Inserir 10:
 Depois: head -> [10|Next] -> [20|Next] -> [30|null]
```

Método: `public void addFirst(int codigo, String nome)`

Inserção no Início - Implementação - O(1)

```
public void addFirst(int codigo, String nome) {
    No novoNo = new No();
    novoNo.preencher(codigo, nome);
    novoNo.next = head;
    head = novoNo;
}
```

Inserção no Fim

Algoritmo

1. **Criar novo nó** com o dado
2. **Se lista vazia:** novo nó vira head
3. **Senão:** percorrer até o último nó e conectar

```
Método: public void addLast(int codigo, String nome)
```

Inserção no Fim - Implementação - O(n)

```
public void addLast(int codigo, String nome) {
    No novoNo = new No();
    novoNo.preencher(codigo, nome);
    if (isEmpty()) {
        head = novoNo;
        return;
    }

    No atual = head;
    while (atual.next != null) {
        atual = atual.next;
    }
    atual.next = novoNo;
}
```

Busca de Elemento

Algoritmo

1. **Começar do head**
2. **Percorrer a lista** comparando dados
3. **Retornar posição** (ou -1 se não encontrar)

```
Método: public int search(int codigo)
```

Busca de Elemento - Implementação - O(n)

```java
public int search(int codigo) {
    No atual = head;
    int posicao = 0;

    while (atual != null) {
        if (atual.getCodigo() == codigo) {
            return posicao;
        }
        atual = atual.next;
        posicao++;
    }

    return -1; // Não encontrado
}
```

Remoção do Início

Algoritmo

1. Verificar se lista não está vazia
2. Atualizar head para o próximo nó

Visualização

```
Antes:  head -> [10|Next] -> [20|Next] -> [30|null]

Depois: head -> [20|Next] -> [30|null]
```

Método: `public boolean removeFirst()`

Remoção do Início - Implementação - O(1)

```
public boolean removeFirst() {
    if (isEmpty()) {
        return false;
    }

    head = head.next;
    return true;
}
```

Remoção de Elemento Específico

Algoritmo

1. **Caso especial:** remoção do primeiro elemento
2. **Encontrar o nó anterior** ao que será removido
3. **Conectar o anterior** ao próximo do removido

`Método: public boolean remove(int codigo)`

Remoção de Elemento Específico - Implementação

```
public boolean remove(int codigo) {
    if (isEmpty()) {
        return false;
    }
    // Caso especial: remover o primeiro
    if (head.getCodigo == codigo) {
        head = head.next;
        return true;
    }
    No atual = head;
    while (atual.next != null && atual.next.getCodigo() != codigo) {
        atual = atual.next;
    }
    if (atual.next != null) {
        atual.next = atual.next.next;
        return true;
    }
    return false; // Não encontrado
}
```

Exercício Prático 1

Implemente um método size() que retorne o número de elementos na lista.

```
public int size() {
    // Seu código aqui
}
```

Dica: Percorra a lista contando os nós.

Exercício Prático 2

Implemente um método add(int elemento, int posicao) que insira um elemento em uma posição específica.

```
public boolean add(int elemento, int posicao) {

    // Seu código aqui

}
```

Considere:

- Posições inválidas (negativa ou maior que o tamanho)
- Posição 0 = início da lista

Exercício Prático 3

Implemente um método get(int posicao) que retorne o elemento em uma posição específica.

```
public Integer get(int posicao) {
    // Seu código aqui
    // Retorne null se posição inválida
}
```

Considere:

- Posições inválidas
- Lista vazia

Vantagens das Listas Encadeadas

✅ Tamanho Dinâmico

- **Cresce e diminui** conforme necessário
- **Não precisa definir** tamanho máximo

✅ Inserção/Remoção Eficiente

- **O(1) no início**
- **Sem deslocamento** de elementos

✅ Uso Eficiente de Memória

- **Aloca apenas** o que precisa
- **Sem desperdício** de espaço

Desvantagens das Listas Encadeadas

❌ Acesso Sequencial

- **O(n)** para acessar elemento específico
- **Não há acesso direto** por índice

❌ Maior Uso de Memória por Nó

- **Cada nó armazena:** dado + ponteiro
- **Overhead** de memória

❌ Complexidade de Implementação

- **Cuidado com ponteiros** (NullPointerException)
- **Mais complexo** que arrays

Comparação: Array vs Lista Encadeada

Operação	Array	Lista Encadeada
Acesso por índice	O(1)	O(n)
Busca	O(n)	O(n)
Inserção no início	O(n)	O(1)
Inserção no fim	O(1)*	O(n)
Remoção no início	O(n)	O(1)
Remoção no fim	O(1)	O(n)
Uso de memória	Menor	Maior
Tamanho	Fixo	Dinâmico

**Se houver espaço disponível*

Exercício: Menu Interativo de Manipulação de Lista

Crie um programa Java que utilize a classe ListaEncadeadaSimples (ou sua implementação) e apresente um menu interativo ao usuário. O usuário deverá ser capaz de:

1.Adicionar um novo elemento no início da lista (solicitando codigo e nome).

2.Adicionar um novo elemento no fim da lista (solicitando codigo e nome).

3.Remover o primeiro elemento da lista.

4.Remover um elemento específico da lista (solicitando o codigo do elemento a ser removido).

5.Buscar um elemento na lista (solicitando o codigo e informando a posição se encontrado).

6.Exibir todos os elementos da lista.

7.Sair do programa.

Utilize a classe Scanner para obter a entrada do usuário. Certifique-se de tratar casos de lista vazia e entradas inválidas.

Exercício: Contagem de Ocorrências

Implemente um método na classe ListaEncadeadaSimples chamado contarOcorrencias(String nome) que receba um nome como parâmetro e retorne o número de vezes que esse nome aparece na lista.

Em seguida, crie um programa principal que permita ao usuário inserir vários elementos na lista e, posteriormente, buscar e contar as ocorrências de um nome específico informado por ele.

Exercício: Inversão da Lista

Implemente um método na classe ListaEncadeadaSimples chamado inverterLista() que inverta a ordem dos elementos na lista encadeada.

O método não deve criar uma nova lista, mas sim reordenar os nós existentes.

Após implementar o método, crie um programa que permita ao usuário popular a lista e, em seguida, exiba a lista original e a lista invertida.

Exercício: União de Listas

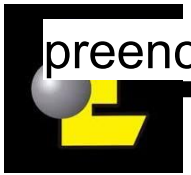
Crie um método estático em uma classe utilitária (ou na própria ListaEncadeadaSimples, se preferir) chamado

unirListas(ListaEncadeadaSimples lista1, ListaEncadeadaSimples lista2)

que receba duas listas encadeadas como parâmetro e retorne uma nova lista contendo todos os elementos da lista1 seguidos pelos elementos da lista2.

Os elementos duplicados devem ser mantidos.

Crie um programa principal que permita ao usuário criar duas listas, preenchê-las e, em seguida, exiba a lista resultante da união.



Exercício: Gerenciamento de Tarefas Simples

Crie um programa de gerenciamento de tarefas simples utilizando a classe ListaEncadeadaSimples. Cada nó da lista representará uma tarefa e deverá conter, além do codigo e nome, um campo boolean concluida. O programa deve permitir ao usuário:

1.Adicionar uma nova tarefa (solicitando codigo e nome).

2.Marcar uma tarefa como concluída (solicitando o codigo da tarefa).

3.Listar todas as tarefas (indicando se estão concluídas ou pendentes).

4.Remover tarefas concluídas.

5.Sair do programa.

Este exercício exige uma pequena modificação na classe No para incluir o campo concluída e métodos adicionais na ListaEncadeadaSimples para manipular o status das tarefas.