# Estruturas Fundamentais e Listas

## Usando arranjos e encadeamento

Prof. Marcelo de Souza

45RPE – Resolução de Problemas com Estruturas de Dados

Universidade do Estado de Santa Catarina

# Listas dinâmicas

## Ideia geral

Uma **lista dinâmica** é uma estrutura que armazena elementos de forma sequencial e permite operações (inserção e remoção) em posições arbitrárias.

# Listas dinâmicas

## Ideia geral

Uma **lista dinâmica** é uma estrutura que armazena elementos de forma sequencial e permite operações (inserção e remoção) em posições arbitrárias.

As listas são usadas em diversas situações e servem de **base** para a construção de estruturas de dados mais sofisticadas.

# Listas dinâmicas

## Ideia geral

Uma **lista dinâmica** é uma estrutura que armazena elementos de forma sequencial e permite operações (inserção e remoção) em posições arbitrárias.

As listas são usadas em diversas situações e servem de **base** para a construção de estruturas de dados mais sofisticadas.

**Algumas operações:**

- Consultar elemento de uma posição.
- Atualizar o elemento de uma posição.
- Inserir elemento no início.
- Inserir elemento no final.
- Inserir elemento em posição arbitrária.
- Remover elemento do início.
- Remover elemento do final.
- Remover elemento de posição arbitrária.

# Listas dinâmicas

## Implementação

O Python nos fornece uma implementação de listas usando **arranjos** (*arrays*, vetores). Alternativamente, podemos implementar listas usando **encadeamento**.

# Listas dinâmicas

## Implementação

O Python nos fornece uma implementação de listas usando **arranjos** (*arrays*, vetores). Alternativamente, podemos implementar listas usando **encadeamento**.

A principal característica dos **arranjos** é a **alocação contígua** em memória.

- ✓ Fácil de usar.
- ✓ Acesso rápido (tempo constante).
- ✗ Tamanho estático (aumentar implica copiar elementos; desperdício de memória).

# Listas dinâmicas

## Implementação

O Python nos fornece uma implementação de listas usando **arranjos** (*arrays*, vetores). Alternativamente, podemos implementar listas usando **encadeamento**.

A principal característica dos **arranjos** é a **alocação contígua** em memória.

- ✓ Fácil de usar.
- ✓ Acesso rápido (tempo constante).
- ✗ Tamanho estático (aumentar implica copiar elementos; desperdício de memória).

A principal característica do **encadeamento** é a **alocação dinâmica** de memória.

- ✓ Tamanho dinâmico (consome somente o necessário de memória).
- ✗ Acesso lento (tempo linear).

# Arranjos

## Listas usando arranjos

Arranjos são **estruturas de dados sequenciais**, armazenando sequências finitas e ordenadas de valores, geralmente de um mesmo tipo.

# Arranjos

## Listas usando arranjos

Arranjos são **estruturas de dados sequenciais**, armazenando sequências finitas e ordenadas de valores, geralmente de um mesmo tipo.

**Algumas operações:**

- Consultar elemento de uma posição: `L[i]`
- Atualizar o elemento de uma posição: `L[i] = e`
- Inserir elemento no início: `L.insert(0, e)`
- Inserir elemento no final: `L.append(e)`
- Inserir elemento em posição arbitrária: `L.insert(i, e)`
- Remover elemento do início: `L.pop(0)`
- Remover elemento do final: `L.pop(-1)`
- Remover elemento de posição arbitrária: `L.pop(i)`

# Arranjos

## Listas usando arranjos

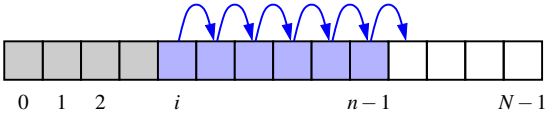
**Problema 1:** realocação de elementos nas operações.

# Arranjos

## Listas usando arranjos

**Problema 1:** realocação de elementos nas operações.

- Na **inserção**, os elementos seguintes são movidos pra **trás**.

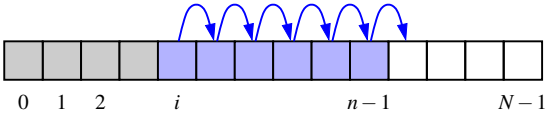


# Arranjos

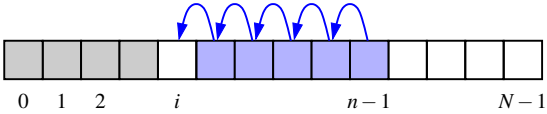
## Listas usando arranjos

**Problema 1:** realocação de elementos nas operações.

- Na **inserção**, os elementos seguintes são movidos pra **trás**.




- Na **remoção**, os elementos seguintes são movidos pra **frente**.



# Arranjos

## Listas usando arranjos

**Problema 2:** capacidade limitada devido à estrutura fixa do arranjo.

# Arranjos

## Listas usando arranjos

**Problema 2:** capacidade limitada devido à estrutura fixa do arranjo.

**Solução:**

- Mantém uma lista de **tamanho fixo**.
- Quando precisar de mais espaço, **dobra o tamanho** da lista.
  - Cria um novo arranjo com o dobro de capacidade.
  - Copia os elementos para o novo arranjo.
  - Atualiza a referência.

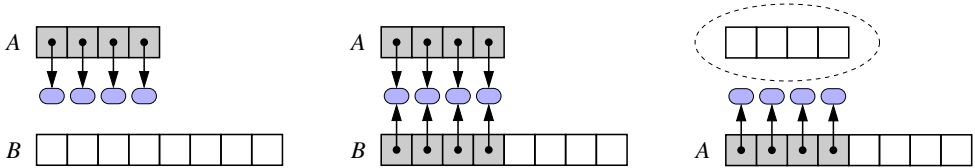
# Arranjos

## Listas usando arranjos

**Problema 2:** capacidade limitada devido à estrutura fixa do arranjo.

**Solução:**

- Mantém uma lista de **tamanho fixo**.
- Quando precisar de mais espaço, **dobra o tamanho** da lista.
  - Cria um novo arranjo com o dobro de capacidade.
  - Copia os elementos para o novo arranjo.
  - Atualiza a referência.



# Arranjos

## Análise de complexidade

| Operação | Complexidade |
|---|---|
| Consultar elemento de uma posição | |
| Atualizar elemento de uma posição | |
| Inserir elemento no início | |
| Inserir elemento no final | |
| Inserir elemento em posição arbitrária | |
| Remover elemento do início | |
| Remover elemento do final | |
| Remover elemento de posição arbitrária | |

# Arranjos

## Análise de complexidade

| Operação | Complexidade |
|---|---|
| Consultar elemento de uma posição | $\mathcal{O}(1)$ |
| Atualizar elemento de uma posição | $\mathcal{O}(1)$ |
| Inserir elemento no início | $\mathcal{O}(n)$ |
| Inserir elemento no final | $\mathcal{O}(1)^*$ |
| Inserir elemento em posição arbitrária | $\mathcal{O}(n)$ |
| Remover elemento do início | $\mathcal{O}(n)$ |
| Remover elemento do final | $\mathcal{O}(1)$ |
| Remover elemento de posição arbitrária | $\mathcal{O}(n)$ |

# Listas simplesmente encadeadas

## Encadeamento

Queremos uma estrutura de dados dinâmica que permita **expandir** e **contrair** com eficiência.

# Listas simplesmente encadeadas

## Encadeamento

Queremos uma estrutura de dados dinâmica que permita **expandir** e **contrair** com eficiência.

**Lista encadeada**: coleção de nodos formados em uma sequência linear.

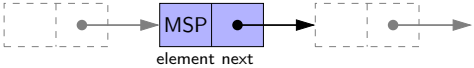
# Listas simplesmente encadeadas

## Encadeamento

Queremos uma estrutura de dados dinâmica que permita **expandir** e **contrair** com eficiência.

**Lista encadeada**: coleção de nodos formados em uma sequência linear.

**Lista simplesmente encadeada**: cada nodo armazena os dados do elemento e uma referência ao próximo nodo. A alocação em memória **não é contígua**.



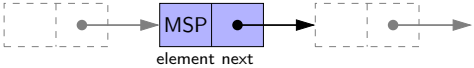
# Listas simplesmente encadeadas

## Encadeamento

Queremos uma estrutura de dados dinâmica que permita **expandir** e **contrair** com eficiência.

**Lista encadeada**: coleção de nodos formados em uma sequência linear.

**Lista simplesmente encadeada**: cada nodo armazena os dados do elemento e uma referência ao próximo nodo. A alocação em memória **não é contígua**.



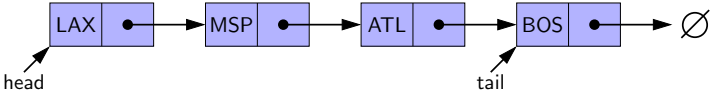

**Benefícios:**

- Tamanho dinâmico;
- Consumo de memória dinâmico;
- Fácil inserção e remoção de elementos.

# Listas simplesmente encadeadas

## Encadeamento

**Exemplo:** uma lista simplesmente encadeada para armazenar aeroportos dos EUA.




**Elementos:**

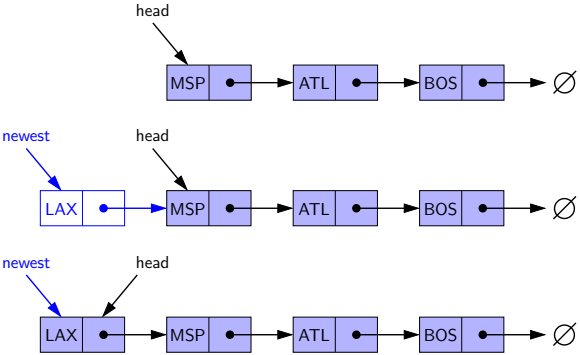
- `head`: referência ao primeiro elemento da lista;
- `tail`: referência ao último elemento da lista;
- O próximo elemento do último elemento aponta para `null`.

# Listas simplesmente encadeadas

## Operações

**Inserção no início**

```
newest.next ← head
head ← newest
```



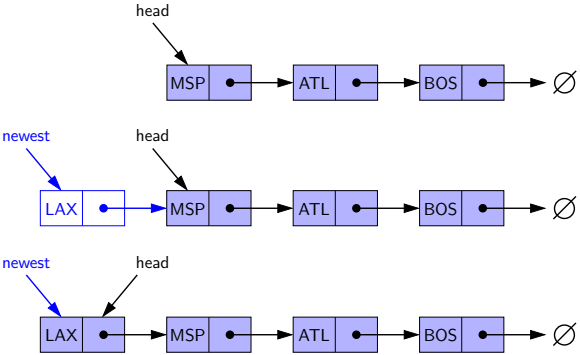
# Listas simplesmente encadeadas

## Operações

**Inserção no início**

```
newest.next ← head
head ← newest
```

Operação em $\mathcal{O}(1)$.

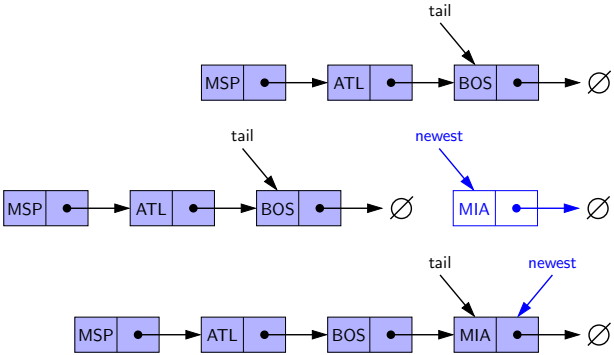


# Listas simplesmente encadeadas

## Operações

**Inserção no final**

```
tail.next ← newest
tail ← newest
```



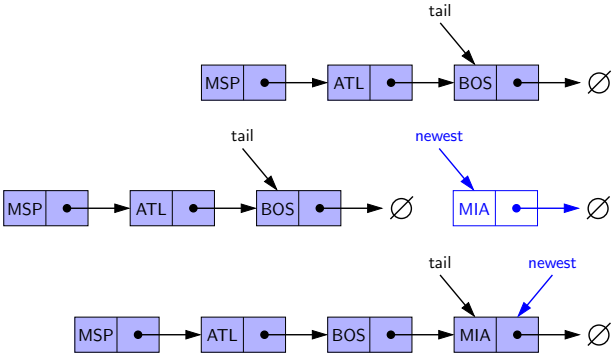
# Listas simplesmente encadeadas

## Operações

**Inserção no final**

```
tail.next ← newest
tail ← newest
```

Operação em $\mathcal{O}(1)$.

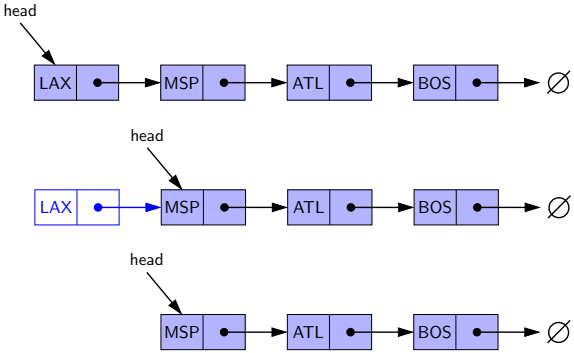


# Listas simplesmente encadeadas

## Operações

**Remoção do início**

`head` ← `head.next`



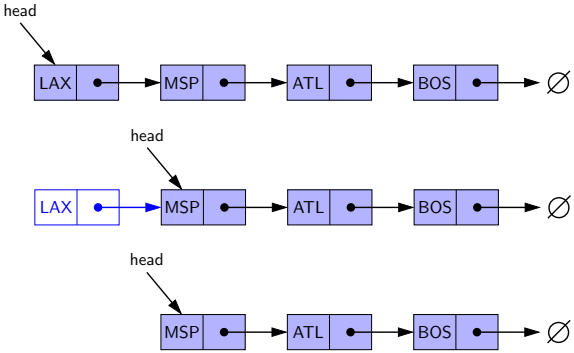
# Listas simplesmente encadeadas

## Operações

**Remoção do início**

`head` ← `head.next`

Operação em $\mathcal{O}(1)$.



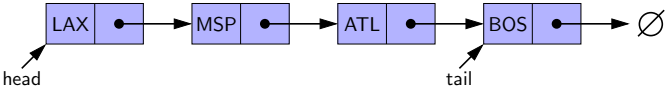
# Listas simplesmente encadeadas

## Operações

E se quisermos **inserir** ou **remover** em/de uma **posição arbitrária**?

1. Percorre a lista, até chegar no nodo da posição desejada.
2. Faz a inserção ou remoção, atualizando as referências.

Operação em $O(n)$, por conta da necessidade de percurso.



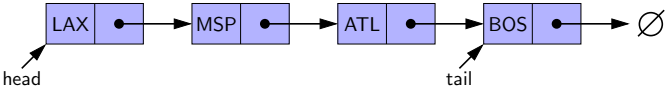
# Listas simplesmente encadeadas

## Operações

E se quisermos **inserir** ou **remover** em/de uma **posição arbitrária**?

1. Percorre a lista, até chegar no nodo da posição desejada.
2. Faz a inserção ou remoção, atualizando as referências.

Operação em $O(n)$, por conta da necessidade de percurso.




E se quisermos **remover** do **final** da lista?

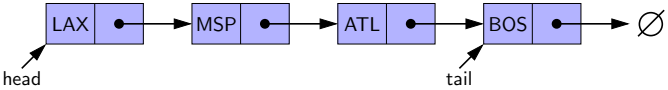
# Listas simplesmente encadeadas

## Operações

E se quisermos **inserir** ou **remover** em/de uma **posição arbitrária**?

1. Percorre a lista, até chegar no nodo da posição desejada.
2. Faz a inserção ou remoção, atualizando as referências.

Operação em $O(n)$, por conta da necessidade de percurso.




E se quisermos **remover** do **final** da lista?

1. Percorre a lista até o penúltimo nodo.
2. Atualiza sua referência `next` para `null` e a referência do `tail`.

Operação em $O(n)$, por conta da necessidade de percurso.

# Listas duplamente encadeadas

## Conceito e motivação

Problemas do encadeamento simples:

- Não conseguimos **remover o último nodo** de forma eficiente.
  - Precisamos atualizar a referência `next` do nodo anterior.
  - Para chegar no penúltimo, precisamos **percorrer a lista**.
- Remover um nodo (que não seja o primeiro) tendo apenas a sua referência é custoso.

# Listas duplamente encadeadas

## Conceito e motivação

Problemas do encadeamento simples:

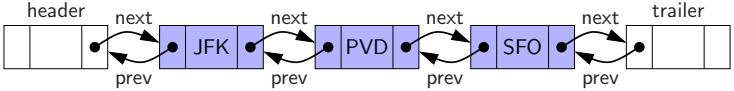
- Não conseguimos **remover o último nodo** de forma eficiente.
  - Precisamos atualizar a referência `next` do nodo anterior.
  - Para chegar no penúltimo, precisamos **percorrer a lista**.
- Remover um nodo (que não seja o primeiro) tendo apenas a sua referência é custoso.

**Lista duplamente encadeada**: cada nodo mantém a referência do anterior (`prev`) e do próximo (`next`) nodos.

# Listas duplamente encadeadas

## Sentinelas

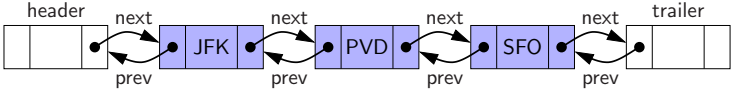
Na implementação dessas listas, usamos uma técnica muito útil: uso de **nodos sentinelas**. Trata-se de nodos vazios ("fictícios") no início (`header`) e no fim (`trailer`) da lista.



# Listas duplamente encadeadas

## Sentinelas

Na implementação dessas listas, usamos uma técnica muito útil: uso de **nodos sentinelas**. Trata-se de nodos vazios ("fictícios") no início (`header`) e no fim (`trailer`) da lista.




Facilidades (na implementação):

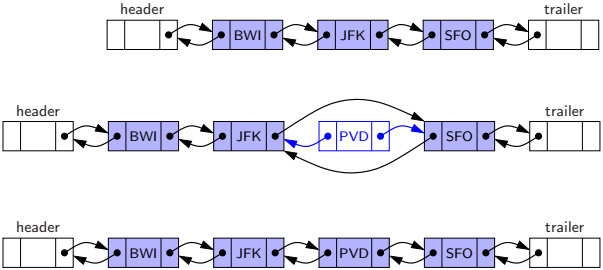
- Certeza de que cada nodo possui dois vizinhos;
- Toda inserção será entre dois nodos.
  - Nunca inserimos no verdadeiro início ou fim;
  - Casos excepcionais (lista vazia ou com um nodo) não acontecem.

# Listas duplamente encadeadas

## Operações

**Inserção arbitrária** (entre `A` ↔ `B`)

```
newest.prev ← A
newest.next ← B
A.next ← newest
B.prev ← newest
```

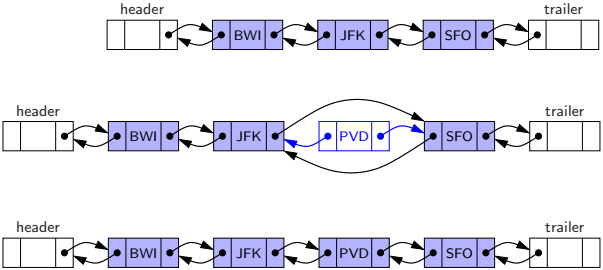


# Listas duplamente encadeadas

## Operações

**Inserção arbitrária** (entre `A` ↔ `B`)

```
newest.prev ← A
newest.next ← B
A.next ← newest
B.prev ← newest
```




A inserção tendo as **referências** dos nodos vizinhos exige $\mathcal{O}(1)$.

A inserção nas **extremidades** exige $\mathcal{O}(1)$.

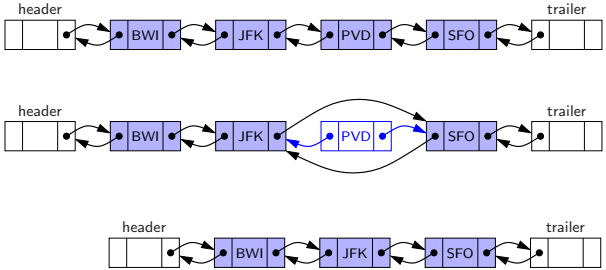
A inserção tendo somente a **posição** desejada exige $\mathcal{O}(n)$, dada a necessidade de percurso.

# Listas duplamente encadeadas

## Operações

**Remoção arbitrária** (do nodo `X`)

```
X.prev.next ← X.next
X.next.prev ← X.prev
```

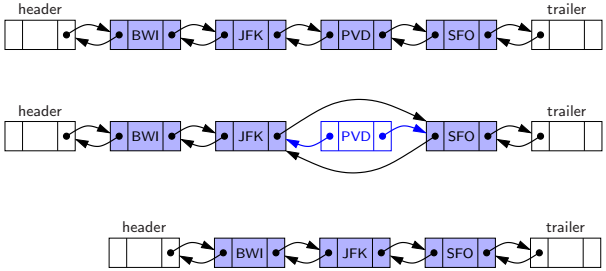


# Listas duplamente encadeadas

## Operações

**Remoção arbitrária** (do nodo `X`)

```
X.prev.next ← X.next
X.next.prev ← X.prev
```



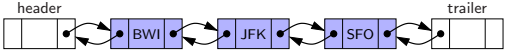

A remoção tendo a **referência** do nodo a ser removido exige $\mathcal{O}(1)$.

A remoção das **extremidades** exige $\mathcal{O}(1)$.

A remoção tendo somente a **posição** desejada exige $\mathcal{O}(n)$, dada a necessidade de percurso.

# Listas encadeadas

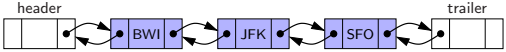
## Análise de complexidade




| Operação | Arranjos | Encadeamento |
|---|---|---|
| Consultar elemento de uma posição | $\mathcal{O}(1)$ | |
| Atualizar elemento de uma posição | $\mathcal{O}(1)$ | |
| Inserir elemento no início | $\mathcal{O}(n)$ | |
| Inserir elemento no final | $\mathcal{O}(1)^*$ | |
| Inserir elemento em posição arbitrária | $\mathcal{O}(n)$ | |
| Remover elemento do início | $\mathcal{O}(n)$ | |
| Remover elemento do final | $\mathcal{O}(1)$ | |
| Remover elemento de posição arbitrária | $\mathcal{O}(n)$ | |

# Listas encadeadas

## Análise de complexidade




| Operação | Arranjos | Encadeamento |
|---|---|---|
| Consultar elemento de uma posição | $\mathcal{O}(1)$ | $\mathcal{O}(n)$ |
| Atualizar elemento de uma posição | $\mathcal{O}(1)$ | $\mathcal{O}(n)$ |
| Inserir elemento no início | $\mathcal{O}(n)$ | $\mathcal{O}(1)$ |
| Inserir elemento no final | $\mathcal{O}(1)^*$ | $\mathcal{O}(1)$ |
| Inserir elemento em posição arbitrária | $\mathcal{O}(n)$ | $\mathcal{O}(n)$ |
| Remover elemento do início | $\mathcal{O}(n)$ | $\mathcal{O}(1)$ |
| Remover elemento do final | $\mathcal{O}(1)$ | $\mathcal{O}(1)$ |
| Remover elemento de posição arbitrária | $\mathcal{O}(n)$ | $\mathcal{O}(n)$ |

45RPE – Resolução de Problemas com Estruturas de Dados
Prof. Marcelo de Souza