Estruturas de Dados Fundamentais

Arranjos e listas encadeadas

Prof. Marcelo de Souza

45EST – Algoritmos e Estruturas de Dados
Universidade do Estado de Santa Catarina

UDESC

Material de apoio

Leitura principal:

- Capítulo 3 de Goodrich et al. (2014)[1] – Estruturas de dados fundamentais.

Leitura complementar:

- Capítulo 4 de Preiss (2001)[2] – Estruturas de dados fundamentais.
- Capítulo 2 de Pereira (2008)[3] – Listas lineares.

[1] Michael T Goodrich et al. (2014). *Data structures and algorithms in Java*. 6ª ed. John Wiley & Sons.

[2] Bruno R Preiss (2001). *Estruturas de dados e algoritmos: padrões de projetos orientados a objetos com Java*. Campus.

[3] Silvio do Lago Pereira (2008). *Estruturas de Dados Fundamentais: Conceitos e Aplicações*.

Arranjos

Ou seja, *arrays*/vetores

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

- **Números:** 1, 2, 4, 5, 7, 8, ...
- **Strings:** "Brasil", "Alemanha", "Croácia", ...
- **Veículos:** ("Corcel", 1977), ("Fusca", 1968), ("Passat", 1984), ...

Arranjos

Ou seja, *arrays*/vetores

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

- **Números:** 1, 2, 4, 5, 7, 8, ...
- **Strings:** "Brasil", "Alemanha", "Croácia", ...
- **Veículos:** ("Corcel", 1977), ("Fusca", 1968), ("Passat", 1984), ...

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

- **Vantagens:**
  - Fácil de usar;
  - Acesso rápido (tempo constante).
- **Desvantagens:**
  - Tamanho fixo (aumentar implica copiar elementos);
  - Inserção e remoção [interna] custosas (*shift* de elementos).

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.

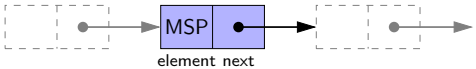
Listas simplesmente encadeadas

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**.



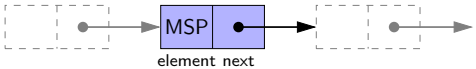
Listas simplesmente encadeadas

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**.



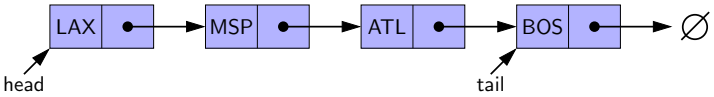
Benefícios:

- 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.



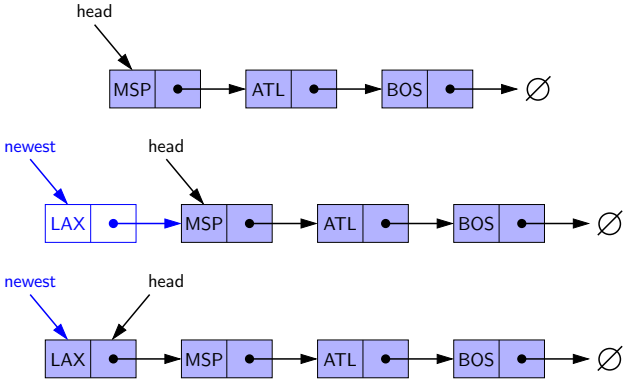
Elementos:

- `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

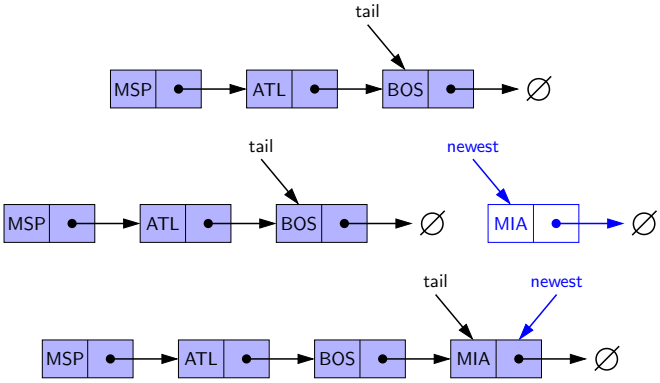
Inserção de elemento no início



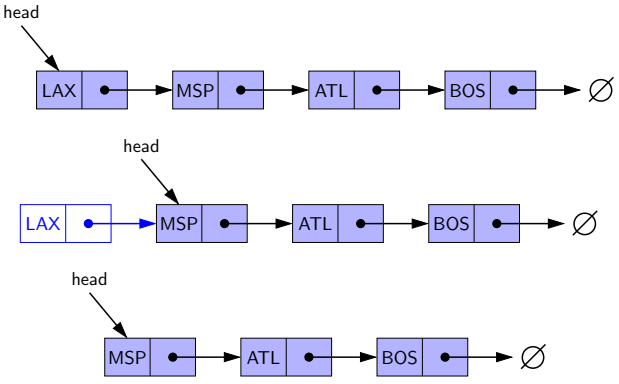
Listas simplesmente encadeadas

Operações

Inserção de elemento no final



Remoção de elemento do início



Listas simplesmente encadeadas

Implementação

```java
public class SinglyLinkedList<E> {

  private static class Node<E> {
    private E element;
    private Node<E> next;

    public Node(E e, Node<E> n) {
      element = e;
      next = n;
    }

    public E getElement() { return element; }
    public Node<E> getNext() { return next; }
    public void setNext(Node<E> n) { next = n; }
  }

  private Node<E> head = null;
  private Node<E> tail = null;
  private int size = 0;

  // ...
}
```

Listas simplesmente encadeadas

Implementação

```
1  public class SinglyLinkedList<E> {
2
3    private static class Node<E> {
4      private E element;
5      private Node<E> next;
6
7      public Node(E e, Node<E> n) {
8        element = e;
9        next = n;
10     }
11
12     public E getElement() { return element; }
13     public Node<E> getNext() { return next; }
14     public void setNext(Node<E> n) { next = n; }
15   }
16
17   private Node<E> head = null;
18   private Node<E> tail = null;
19   private int size = 0;
20
21   // ...
22 }
```

Detalhes:

- Usamos **genéricos** (`<E>`) para suportar qualquer tipo de dados.
  - e.g., podemos ter uma lista de inteiros, Strings, veículos, ...
- A classe `Node` define um nodo, que contém um elemento e uma referência ao próximo nodo da lista.
  - `Node` é uma **nested class**, pois queremos encapsular o nodo.
- A lista contém referências para o primeiro e último nodos (`head` e `tail`) e o seu tamanho (número de nodos), inicialmente zero.

Listas simplesmente encadeadas

Implementação

Métodos `size` e `isEmpty`:

```
1  public int size() { return size; }
2  public boolean isEmpty() { return size == 0; }
```

Listas simplesmente encadeadas

Implementação

Métodos `size` e `isEmpty`:

```
1  public int size() { return size; }
2  public boolean isEmpty() { return size == 0; }
```

Métodos `first` e `last`:

```
1  public E first() {
2    if (isEmpty()) return null;
3    return head.getElement();
4  }
5
6  public E last() {
7    if (isEmpty()) return null;
8    return tail.getElement();
9  }
```

Note que:

- O método retorna o elemento armazenado pelo primeiro (ou último) nodo.
- A estrutura da lista (`Node`) é transparente (encapsulada).

Listas simplesmente encadeadas

Implementação

Método `addFirst`:

```
1  public void addFirst(E e) {
2    head = new Node<>(e, head);
3    if (size == 0)
4      tail = head;
5    size++;
6  }
```

- Criamos um novo nodo para armazenar o elemento, que passa a ser o novo `head` e aponta para o antigo `head`.
- Caso a lista esteja vazia, ele é o novo `tail`.

Listas simplesmente encadeadas

Implementação

Método `addFirst`:

```
1  public void addFirst(E e) {
2    head = new Node<>(e, head);
3    if (size == 0)
4      tail = head;
5    size++;
6  }
```

- Criamos um novo nodo para armazenar o elemento, que passa a ser o novo `head` e aponta para o antigo `head`.
- Caso a lista esteja vazia, ele é o novo `tail`.

Método `addLast`:

```
1  public void addLast(E e) {
2    Node<E> newest = new Node<>(e, null);
3    if (isEmpty())
4      head = newest;
5    else
6      tail.setNext(newest);
7    tail = newest;
8    size++;
9  }
```

- O novo nodo (`newest`) passa a ser o próximo nodo do atual `tail`, antes de assumir a posição do `tail`.
- Caso a lista esteja vazia, o novo nodo passa a ser o `head` e o `tail`.

Listas simplesmente encadeadas

Implementação

Método `removeFirst`:

```
1  public E removeFirst() {
2    if (isEmpty()) return null;
3    E answer = head.getElement();
4    head = head.getNext();
5    size--;
6    if (size == 0) tail = null;
7    return answer;
8  }
```

- Retorna o elemento do nodo removido.
- O nodo `head` passa a ser o próximo nodo do `head` atual.
- Se a remoção deixa a lista vazia, o `tail` passa a ser `null`.

Listas simplesmente encadeadas

Implementação

Método `removeFirst`:

```
1  public E removeFirst() {
2    if (isEmpty()) return null;
3    E answer = head.getElement();
4    head = head.getNext();
5    size--;
6    if (size == 0) tail = null;
7    return answer;
8  }
```

- Retorna o elemento do nodo removido.
- O nodo `head` passa a ser o próximo nodo do `head` atual.
- Se a remoção deixa a lista vazia, o `tail` passa a ser `null`.

Método `toString`:

```
1  public String toString() {
2    StringBuilder sb = new StringBuilder("(");
3    Node<E> walk = head;
4    while (walk != null) {
5      sb.append(walk.getElement());
6      if (walk != tail) sb.append(", ");
7      walk = walk.getNext();
8    }
9    sb.append(")");
10   return sb.toString();
11 }
```

- Percorremos a estrutura com um `while`, visitando um nodo por vez.
- Para cada nodo, recuperamos seu elemento e o adicionamos na string de resultado.
- O próximo nodo do `tail` é `null`, o que viola a condição do `while` e interrompe a execução do laço.

45EST – Algoritmos e Estruturas de Dados
Prof. Marcelo de Souza