45EST – Algoritmos e Estruturas de Dados

Estruturas de dados fundamentais

Prof. Marcelo de Souza

UDESC Ibirama
Bacharelado em Engenharia de Software

marcelo.desouza@udesc.br
Versão compilada em 12 de setembro de 2018

Leitura obrigatória:

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

Leitura complementar:

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

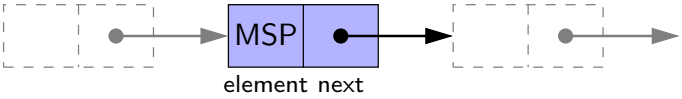
Listas simplesmente encadeadas

Problemas com o uso de arranjos:

- Capacidade fixa.
- Inserção interna dificultada.
- Remoção interna dificultada.

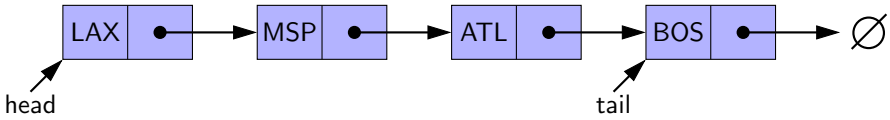
Lista encadeada: coleção de nodos formados em uma sequência linear.

Lista simplesmente encadeada: cada nodo armazena os dados do elemento e uma referência ao próximo nodo.



Elementos:

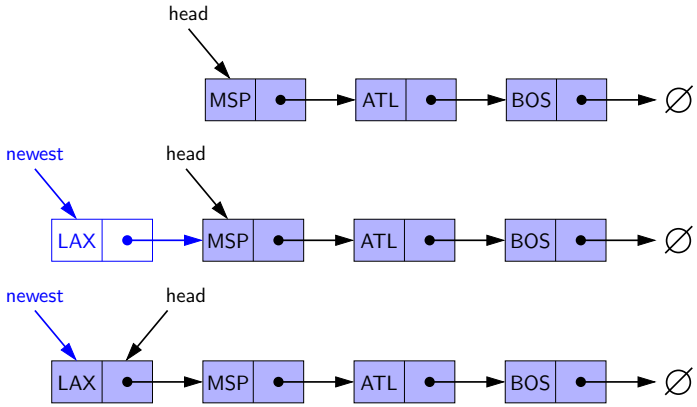
- `head`: aponta para o primeiro elemento da lista.
- `tail`: aponta para o último elemento da lista.
- Último elemento aponta para `null`.



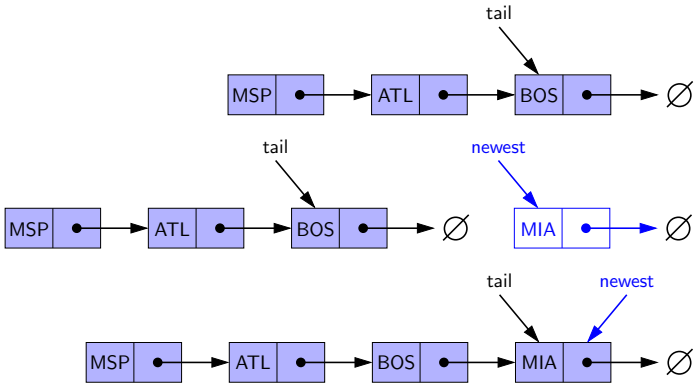
Benefícios:

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

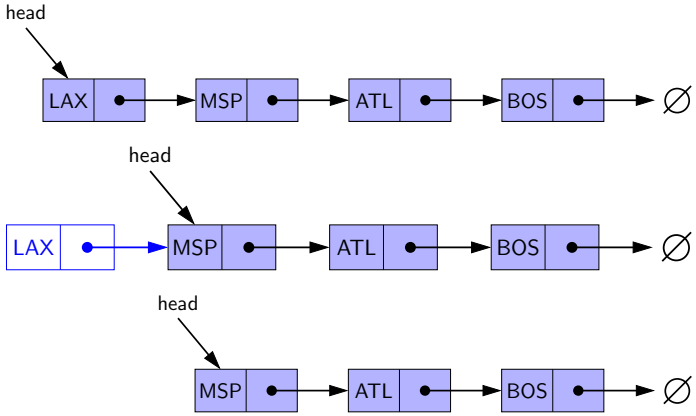
Inserção de elemento no início



Inserção de elemento no final



Remoção de elemento



Problemas na remoção do final:

- Precisamos acessar o penúltimo elemento.
- Toda a lista tem que ser percorrida $\rightarrow$ custoso.

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;

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

   public E first() {
      if (isEmpty()) return null;
      return head.getElement();
   }

   public E last() {
      if (isEmpty()) return null;
      return tail.getElement();
   }

   // Métodos addFirst, addLast e removeFirst...
}
```

Comentários:

- Classes externas não têm acesso à estrutura de nodos.
- Outros métodos podem ser implementados para manutenção da lista.

Método `addFirst(E e)`:

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

Método `addLast(E e)`:

```
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  }
```

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)
7        tail = null;
8     return answer;
9  }
```

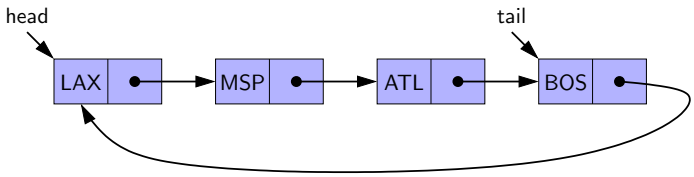
Exercícios:

1. Implemente uma lista encadeada e use a estrutura para armazenar os dados dos alunos inscritos no SEMESO 2018 (nome, matrícula e fase). Crie métodos para inserção, remoção, consulta e listagem de inscritos.

Listas encadeadas circulares

Conceitos básicos:

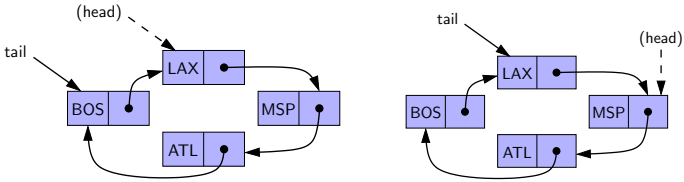
- Trata-se de uma lista encadeada com ordenação cíclica.
- Exemplos de uso:
  - Jogadores de cartas.
  - Escalonamento de processos.
- "Último" elemento aponta para o "primeiro".



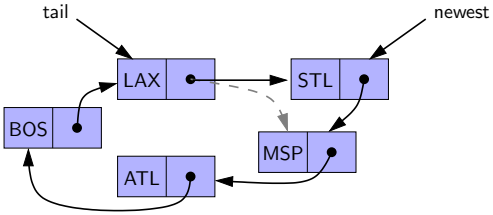
Novidades:

- Não é necessária a referência para `head` (`tail.getNext()`).
- Novo método `rotate`, que atualiza a referência `tail`.

Método `rotate`:



Método `addFirst`:



Método `addLast`:

- Chama o método `addFirst` e atualiza a referência `tail`.

Método `removeFirst`:

- Basta atualizar a referência `next` do elemento `tail`.

Implementação

```
1  public class CircularlyLinkedList<E> {
2
3     //Definição da classe interna Node
4
5     private Node<E> tail = null;
6     private int size = 0;
7
8     public int size() { return size; }
9     public boolean isEmpty() { return size == 0; }
10
11    public E first() {
12       if (isEmpty()) return null;
13       return tail.getNext().getElement();
14    }
15
16    public E last() {
17       if (isEmpty()) return null;
18       return tail.getElement();
19    }
20
21    public void rotate() {
22       if (tail != null)
23          tail = tail.getNext();
24    }
25
26    public void addFirst(E e) {
27       if (size == 0) {
28          tail = new Node<>(e, null);
29          tail.setNext(tail);
30       } else {
31          Node<E> newest = new Node<>(e, tail.getNext());
32          tail.setNext(newest);
33       }
34       size++;
35    }
36
```

```
37      public void addLast(E e) {
38          addFirst(e);
39          tail = tail.getNext();
40      }
41
42      public E removeFirst() {
43          if (isEmpty()) return null;
44          Node<E> head = tail.getNext();
45          if (head == tail) tail = null;
46          else tail.setNext(head.getNext());
47          size--;
48          return head.getElement();
49      }
50
51      public String toString() {
52          if (tail == null) return "()";
53          StringBuilder sb = new StringBuilder("(");
54          Node<E> walk = tail;
55          do {
56              walk = walk.getNext();
57              sb.append(walk.getElement());
58              if (walk != tail)
59                  sb.append(", ");
60          } while (walk != tail);
61          sb.append(")");
62          return sb.toString();
63      }
64  }
```

Exercícios:

1. Implemente uma lista encadeada circular e use a estrutura para armazenar os dados dos jogos da Copa do Mundo 2018 (equipes, local e horário). Crie métodos para inserção, remoção, consulta e listagem de jogos.

Listas duplamente encadeadas

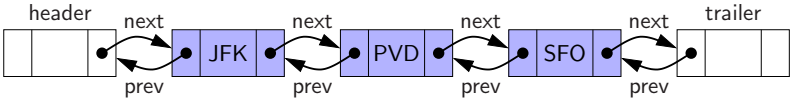
Problemas do encadeamento simples:

- Deletar o último nodo. ← como acessar o penúltimo elemento?
- Deletar nodo tendo apenas sua referência?

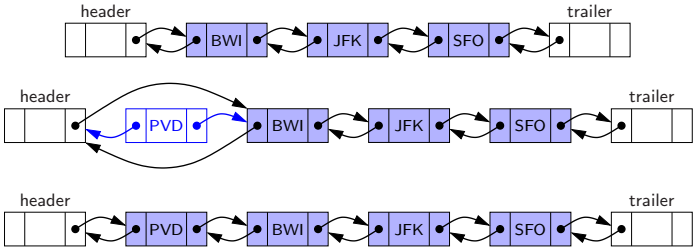
Lista duplamente encadeada: cada nodo mantém a referência do *anterior* (`prev`) e do *próximo* (`next`) nodos.

Sentinelas:

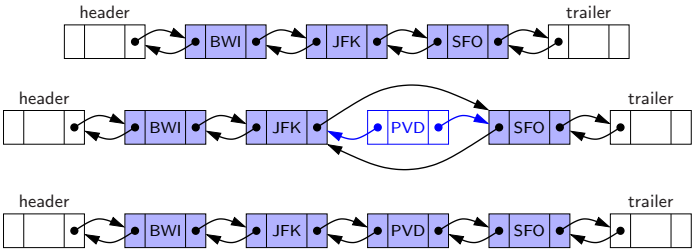
- Nodos "vazios" usados para o início (`header`) e fim (`trailer`) da lista.
- Facilidade: certeza de que cada elemento possui dois vizinhos.



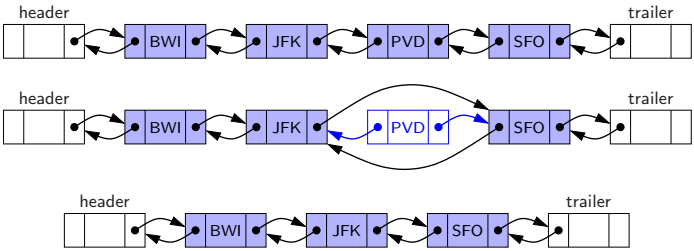
Inserção de elemento no início



Inserção arbitrária de elemento



Remoção de elemento



Implementação

```
1  public class DoublyLinkedList<E> {
2
3     private static class Node<E> {
4        private E element;
5        private Node<E> prev;
6        private Node<E> next;
7
8        public Node(E e, Node<E> p, Node<E> n) {
9           element = e;
10          prev = p;
11          next = n;
```

```
12          }
13
14          public E getElement() { return element; }
15          public Node<E> getPrev() { return prev; }
16          public Node<E> getNext() { return next; }
17
18          public void setPrev(Node<E> p) { prev = p; }
19          public void setNext(Node<E> n) { next = n; }
20      }
21
22      private Node<E> header;
23      private Node<E> trailer;
24      private int size = 0;
25
26      public DoublyLinkedList() {
27          header = new Node<>(null, null, null);
28          trailer = new Node<>(null, header, null);
29          header.setNext(trailer);
30      }
31
32      public int size() { return size; }
33      public boolean isEmpty() { return size == 0; }
34
35      public E first() {
36          if (isEmpty()) return null;
37          return header.getNext().getElement();
38      }
39
40      public E last() {
41          if (isEmpty()) return null;
42          return trailer.getPrev().getElement();
43      }
44
45      public void addFirst(E e) {
46          addBetween(e, header, header.getNext());
47      }
48
49      public void addLast(E e) {
```

```
50          addBetween(e, trailer.getPrev(), trailer);
51      }
52
53      public E removeFirst() {
54          if (isEmpty()) return null;
55          return remove(header.getNext());
56      }
57
58      public E removeLast() {
59          if (isEmpty()) return null;
60          return remove(trailer.getPrev());
61      }
62
63      private void addBetween(E e, Node<E> pred, Node<E> succ) {
64          Node<E> newest = new Node<>(e, pred, succ);
65          pred.setNext(newest);
66          succ.setPrev(newest);
67          size++;
68      }
69
70      private E remove(Node<E> node) {
71          Node<E> predecessor = node.getPrev();
72          Node<E> successor = node.getNext();
73          predecessor.setNext(successor);
74          successor.setPrev(predecessor);
75          size--;
76          return node.getElement();
77      }
78
79      public String toString() {
80          StringBuilder sb = new StringBuilder("(");
81          Node<E> walk = header.getNext();
82          while (walk != trailer) {
83              sb.append(walk.getElement());
84              walk = walk.getNext();
85              if (walk != trailer)
86              sb.append(", ");
87          }
```

```
88          sb.append(")");
89          return sb.toString();
90      }
91  }
```

Exercícios:

1. Implemente uma lista duplamente encadeada e use a estrutura para armazenar os dados dos pratos de um restaurante (nome e valor). Crie métodos para inserção, remoção, consulta e listagem de pratos.

Comparando estruturas de dados

Ideia geral:

- Comparação de objetos compara os ponteiros (espaço de memória).
- Sobrescrever o método `equals`, comparando os elementos.
- Cuidados: referência nula, classes distintas, tamanho da lista.

Implementação:

```
1   public boolean equals(Object o) {
2       if (o == null) return false;
3       if (getClass() != o.getClass()) return false;
4       SinglyLinkedList other = (SinglyLinkedList) o;
5       if (size != other.size) return false;
6       Node walkA = head;
7       Node walkB = other.head;
8       while (walkA != null) {
9           if (!walkA.getElement().equals(walkB.getElement()))
10              return false;
11          walkA = walkA.getNext();
12          walkB = walkB.getNext();
13      }
14      return true;
```

```
15  }
```

Clonando estruturas de dados

Importante: na tentativa de copiar um objeto em Java, a cópia aponta para a mesma posição em memória (referência).

- Solução: implementar a cópia no método `clone`.

A estrutura de dados deve implementar a interface `Cloneable`:

```
1  public class SinglyLinkedList<E> implements Cloneable {
```

O método `clone` deve ser sobrescrito:

- O programador decide o que manter a referência e o que copiar.

```
1  public SinglyLinkedList<E> clone()
2         throws CloneNotSupportedException {
3
4     SinglyLinkedList<E> other = (SinglyLinkedList<E>) super.clone();
5     if (size > 0) {
6        other.head = new Node<>(head.getElement(), null);
7        Node<E> walk = head.getNext();
8        Node<E> otherTail = other.head;
9        while (walk != null) {
10          Node<E> newest = new Node<>(walk.getElement(), null);
11          otherTail.setNext(newest);
12          otherTail = newest;
13          walk = walk.getNext();
14       }
15    }
16    return other;
17 }
```

Atividades

1. Leia sobre as estratégias de geração de números aleatórios detalhadas em Goodrich and Tamassia [2013].
2. Faça os seguintes exercícios de reforço de Goodrich et al. [2014].

R-3.5: O método `removeFirst` da classe `SinglyLinkedList` inclui um caso especial para redefinir o campo `tail` para `null` na remoção do último elemento da lista. Quais são as consequências de remover essas linhas de código? Explique por que a classe não funcionaria com esta modificação.

R-3.6: Proponha um algoritmo para encontrar o penúltimo nodo em uma lista simplesmente encadeada na qual o último nodo possui uma referência nula no campo `next`.

R-3.7: Considere o método `addFirst` da classe `CircularlyLinkedList`. O corpo do `else` depende de uma variável local `newest`. Projete um novo código para esta cláusula sem o uso de nenhuma variável local.

R-3.8: Descreva um método para encontrar o nodo central de uma lista duplamente encadeada com nodos sentinelas, sem o uso de informações sobre o tamanho da lista. No caso de um número par de nodos, o método deve devolver o nodo à esquerda do ponto central. Qual a complexidade deste algoritmo?

R-3.9: Forneça uma implementação para o método `size()` da classe `SinglyLinkedList`, considerando que a mesma não mantenha o tamanho armazenado em uma variável.

R-3.10: Forneça uma implementação para o método `size()` da classe `CircularlyLinkedList`, considerando que a mesma não mantenha o tamanho armazenado em uma variável.

R-3.11: Forneça uma implementação para o método `size()` da classe `DoublyLinkedList`, considerando que a mesma não mantenha o tamanho armazenado em uma variável.

R-3.15: Implemente o método `equals()` para a classe `CircularlyLinkedList`, assumindo que duas listas são iguais se elas possuem a mesma sequência de elementos, com os elementos correspondentes no início da lista.

R-3.16: Implemente o método `equals()` para a classe `DoublyLinkedList`.

3. Faça os seguintes exercícios de criatividade de Goodrich et al. [2014].

C-3.25: Descreva um algoritmo para concatenar duas listas simplesmente encadeadas L e M, em uma lista única L' que contém todos os nodos de L seguido de todos os nodos de M.

C-3.26: Descreva um algoritmo para concatenar duas listas duplamente encadeadas L e M com sentinelas, em uma lista única L'.

C-3.27: Descreva em detalhes como trocar dois nodos x e y de posição (não apenas seu conteúdo) em uma lista simplesmente encadeada L, dadas as referências para x e y somente. Repita este exercício para o caso em que L é uma lista duplamente encadeada. Qual algoritmo possui maior complexidade?

C-3.28: Descreva em detalhes um algoritmo para reverter uma lista simplesmente encadeada L usando somente uma quantidade constante de espaço adicional.

C-3.31: Nossa implementação de uma lista duplamente encadeada depende de dois nodos sentinelas, `header` e `trailer`, mas um único nodo sentinela para ambos início e fim seria suficiente. Reimplemente a classe `DoublyLinkedList` usando apenas um nodo sentinela.

Referências

Goodrich, M. T. and Tamassia, R. (2013). *Estruturas de Dados & Algoritmos em Java*. Bookman Editora, 5th edition.

Goodrich, M. T., Tamassia, R., and Goldwasser, M. H. (2014). *Data structures and algorithms in Java*. John Wiley & Sons, 6th edition.

Pereira, S. d. L. (2008). Estruturas de dados fundamentais: Conceitos e aplicações.

Preiss, B. R. (2001). *Estruturas de dados e algoritmos: padrões de projetos orientados a objetos com Java*. Campus.