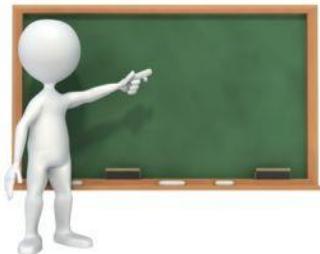




# LISTAS DUPLAMENTE DINAMICAMENTE ENCADEADAS

ESTRUTURA DE DADOS

CST em Desenvolvimento de Software Multiplataforma



PROF. Me. TIAGO A. SILVA



# LIVRO DE REFERÊNCIA DA DISCIPLINA

- **BIBLIOGRAFIA BÁSICA:**

- GRONER, Loiane. **Estrutura de dados e algoritmos com JavaScript**: escreva um código JavaScript complexo e eficaz usando a mais recente ECMAScript. São Paulo: Novatec Editora, 2019.



- **NESTA AULA:**

- Capítulo 6

# PARA SOBREVIVER AO JAVASCRIPT

Non-zero value



0



null

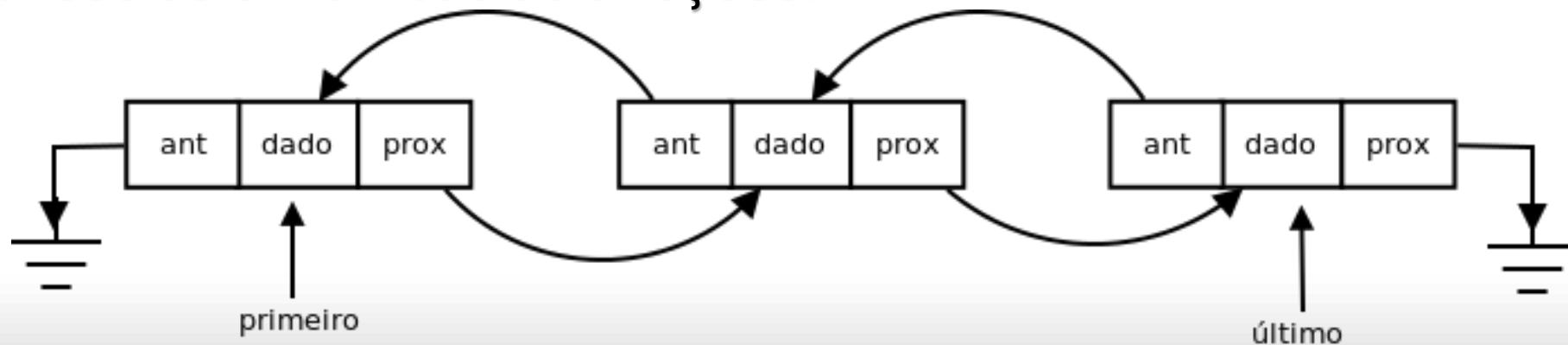


undefined



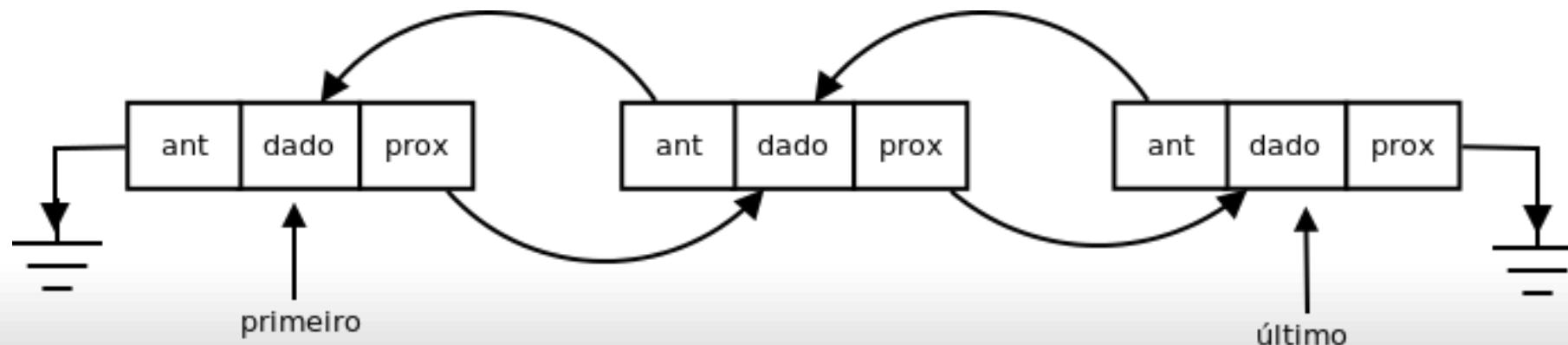
# O QUE SÃO LISTAS DUPLAMENTE ENCADEADAS?

- Uma lista duplamente encadeada é uma estrutura de dados onde cada nó possui um ponteiro que aponta para o próximo e para o nó anterior. Diferentemente das listas encadeadas simples, que só têm um ponteiro para o próximo nó, a lista duplamente encadeada permite percorrer tanto no sentido direto (do início ao fim) quanto no inverso (do fim ao início). Isso a torna bastante útil quando precisamos realizar inserções, exclusões ou travessias em ambas as direções.



# O QUE SÃO LISTAS DUPLAMENTE ENCADEADAS?

- Na lista duplamente encadeada, cada elemento ou nó possui três partes principais:
  - Valor do nó: o dado armazenado no nó.
  - Ponteiro para o próximo nó: uma referência ao nó seguinte.
  - Ponteiro para o nó anterior: uma referência ao nó anterior.



# ESTRUTURA DE UMA LISTA DUPLAMENTE ENCADEADA

- O primeiro passo para implementar uma lista duplamente encadeada é definir um nó. Cada nó contém um valor, um ponteiro para o próximo nó e um ponteiro para o nó anterior. Abaixo está um exemplo da estrutura de um nó:

```
1 class Node {  
2     constructor(value) {  
3         this.value = value;  
4         this.next = null; // Próximo nó  
5         this.prev = null; // Nó anterior  
6     }  
7 }
```

# ESTRUTURA DE UMA LISTA DUPLAMENTE ENCADEADA

- Neste exemplo, criamos a classe **Node** que representa um nó da lista.
- Quando um novo nó é criado, ele possui apenas um valor e os ponteiros **next** e **prev** são inicialmente definidos como **null**.

```
1 class Node {  
2     constructor(value) {  
3         this.value = value;  
4         this.next = null; // Próximo nó  
5         this.prev = null; // Nó anterior  
6     }  
7 }
```

# O QUE VAMOS IMPLEMENTAR?

```
10  class DoublyLinkedList {
11      constructor() {
12          this.head = null; // Primeiro nó (cabeça)
13          this.tail = null; // Último nó (cauda)
14          this.length = 0; // Tamanho da lista
15      }
16
17      // Adicionar um nó ao final da lista
18 >  append(value) { ...
31      }
32
33      // Adicionar um nó ao início da lista
34 >  prepend(value) { ...
47      }
48
49      // Remover o nó do final da lista
50 >  removeLast() { ...
64      }
65
66      // Remover o nó do início da lista
67 >  removeFirst() { ...
81      }
82
83      // Percorrer a lista do início ao fim
84 >  traverse() { ...
90      }
91
92      // Percorrer a lista do fim ao início
93 >  traverseReverse() { ...
99      }
100 }
```

# MÉTODO CONSTRUTOR

- Este método é o construtor da classe **DoublyLinkedList**. Ele inicializa os atributos principais da lista duplamente encadeada:
  - this.head**: o primeiro nó da lista (inicialmente **null**).
  - this.tail**: o último nó da lista (inicialmente **null**).
  - this.length**: o número de nós na lista (inicialmente 0).

```
10  class DoublyLinkedList {  
11      constructor() {  
12          this.head = null; // Primeiro nó (cabeça)  
13          this.tail = null; // Último nó (cauda)  
14          this.length = 0; // Tamanho da lista  
15      }  
16  }
```

# MÉTODO APPREND

- **Adiciona um novo nó com o valor fornecido ao final da lista:**
  - Cria um novo nó.
  - Se a lista estiver vazia, o novo nó será tanto a cabeça (**head**) quanto a cauda (**tail**).
  - Se a lista já tiver elementos, o novo nó é adicionado após a cauda atual, e o ponteiro **prev** do novo nó é configurado para apontar para a cauda antiga. A cauda é atualizada para o novo nó.
  - O tamanho da lista é incrementado.

```
17 // Adicionar um nó ao final da lista
18 append(value) {
19   const newNode = new Node(value);
20
21   if (!this.head) {
22     this.head = newNode;
23     this.tail = newNode;
24   } else {
25     this.tail.next = newNode;
26     newNode.prev = this.tail;
27     this.tail = newNode;
28   }
29
30   this.length++;
31 }
```

# MÉTODO PREPEND

- **Adiciona um novo nó com o valor fornecido ao início da lista:**
  - Cria um novo nó.
  - Se a lista estiver vazia, o novo nó será tanto a cabeça quanto a cauda.
  - Se a lista já tiver elementos, o novo nó é inserido antes da cabeça atual, e o ponteiro **next** do novo nó é configurado para apontar para a cabeça antiga. O ponteiro **prev** da antiga cabeça é configurado para apontar para o novo nó.
  - O tamanho da lista é incrementado.

```
33 // Adicionar um nó ao início da lista
34 prepend(value) {
35   const newNode = new Node(value);
36
37   if (!this.head) {
38     this.head = newNode;
39     this.tail = newNode;
40   } else {
41     newNode.next = this.head;
42     this.head.prev = newNode;
43     this.head = newNode;
44   }
45
46   this.length++;
47 }
```

# MÉTODO REMOVELAST

- **Remove o último nó da lista:**
  - Se a lista estiver vazia, não há nada para remover, e o método retorna **null**.
  - Se a lista contiver apenas um nó, tanto a cabeça quanto a cauda são configuradas como **null**.
  - Caso contrário, o nó anterior à cauda torna-se a nova cauda, e o ponteiro **next** da nova cauda é configurado como **null**.
  - O tamanho da lista é decrementado.
  - Retorna o valor do nó removido.

```
49 // Remover o nó do final da lista
50 removeLast() {
51     if (!this.tail) return null;
52
53     const removedNode = this.tail;
54     if (this.tail === this.head) {
55         this.head = null;
56         this.tail = null;
57     } else {
58         this.tail = this.tail.prev;
59         this.tail.next = null;
60     }
61
62     this.length--;
63     return removedNode.value;
64 }
```

# MÉTODO REMOVEFIRST

- **Remove o primeiro nó da lista:**
  - Se a lista estiver vazia, não há nada para remover, e o método retorna **null**.
  - Se a lista contiver apenas um nó, tanto a cabeça quanto a cauda são configuradas como **null**.
  - Caso contrário, o nó seguinte à cabeça torna-se a nova cabeça, e o ponteiro `prev` da nova cabeça é configurado como **null**.
  - O tamanho da lista é decrementado.
  - Retorna o valor do nó removido.

```
66 // Remover o nó do início da lista
67 removeFirst() {
68   if (!this.head) return null;
69
70   const removedNode = this.head;
71   if (this.head === this.tail) {
72     this.head = null;
73     this.tail = null;
74   } else {
75     this.head = this.head.next;
76     this.head.prev = null;
77   }
78
79   this.length--;
80   return removedNode.value;
81 }
```

# MÉTODO TRAVERSE

- **Percorre a lista do início ao fim, imprimindo o valor de cada nó.**
  - Começa pela cabeça e segue até o final (cauda), acessando os nós através do ponteiro **next**.

```
83     // Percorrer a lista do início ao fim
84     traverse() {
85         let current = this.head;
86         while (current) {
87             console.log(current.value);
88             current = current.next;
89         }
90     }
```

# MÉTODO TRAVERSEVERSE

- **Percorre a lista do fim ao início, imprimindo o valor de cada nó.**
  - Começa pela cauda e segue até o início (cabeça), acessando os nós através do ponteiro **prev**.

```
92      // Percorrer a lista do fim ao início
93      traverseReverse() {
94          let current = this.tail;
95          while (current) {
96              console.log(current.value);
97              current = current.prev;
98          }
99      }
```

## VANTAGENS

- **Travessia bidirecional:** Uma lista duplamente encadeada pode ser percorrida em ambas as direções, o que é uma vantagem em comparação com listas encadeadas simples.
- **Inserção e remoção eficientes:** Como cada nó contém uma referência ao nó anterior, as operações de inserção e remoção em qualquer posição são mais rápidas do que em listas encadeadas simples.

# DESVANTAGENS

- **Maior uso de memória:** Cada nó armazena dois ponteiros adicionais, o que aumenta o consumo de memória.
- **Maior complexidade de implementação:** O gerenciamento dos ponteiros `prev` e `next` exige mais cuidado para evitar erros.

# EXEMPLO DE USO

```
102 const list = new DoublyLinkedList();
103
104 list.append(10);
105 list.append(20);
106 list.append(30);
107
108 console.log("Percorrendo do início ao fim:");
109 list.traverse();
110
111 list.prepend(5);
112
113 console.log("Percorrendo do início ao fim após adição no início:");
114 list.traverse();
115
116 list.removeLast();
117 console.log("Percorrendo após remover o último nó:");
118 list.traverse();
119
120 list.removeFirst();
121 console.log("Percorrendo após remover o primeiro nó:");
122 list.traverse();
123
124 console.log("Percorrendo em ordem inversa:");
125 list.traverseReverse();
```

# CONCLUSÃO

- A lista duplamente encadeada é uma estrutura de dados versátil e poderosa.
- Suas principais vantagens são a capacidade de percorrer em ambas as direções e a eficiência nas operações de inserção e remoção.
- No entanto, ela requer mais memória e uma implementação mais complexa devido aos ponteiros adicionais.

# EXERCÍCIOS

- 1) Implemente uma função na classe **DoublyLinkedList** que insira um nó em uma posição específica, chamando **insertAt(value, index)**
- 2) Implemente um método **find(value)** na lista duplamente encadeada que retorne o índice do nó que contém o valor fornecido. Se o valor não for encontrado, retorne -1.
- 3) Modifique o método **remove** para que ele remova um nó em uma posição específica da lista duplamente encadeada., chame-o de **removeAt(index)**

**COMENTE TODO CÓDIGO DOS EXERCÍCIOS EXPLICANDO SEU FUNCIONAMENTO**

# OBRIGADO!

- Encontre este **material on-line** em:
  - Slides: Plataforma Microsoft Teams
- Em caso de **dúvidas**, entre em contato:
  - **Prof. Tiago:** [tiago.silva238@fatec.sp.gov.br](mailto:tiago.silva238@fatec.sp.gov.br)

