

Estruturas de Dados

A. G. Silva, A. von Wangenheim, J. E. Martina

14 de agosto de 2018

Conceito de Lista

Lista

Uma **Lista** é um conjunto de dados dispostos em uma sequência determinada, podendo ser consultados e manipulados em qualquer posição.

Definições de Lista

- Lista é um conjunto de dados em sequência definida de posições (primeiro elemento na posição lógica 0, segundo elemento na posição lógica 1, ...);
- Lista **pode ou não** ter uma ordem intrínseca, por exemplo, de ordenação do conteúdo de cada elemento;
- Este conjunto de dados **pode ou não** ocupar espaços de memória fisicamente consecutivos, espelhando a sua ordem;
- Se os dados estiverem dispersos fisicamente, para que este conjunto seja uma lista, deve possuir operações e informações adicionais que permitam que seja tratado como tal (Lista Encadeada).

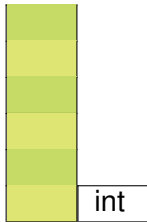
Lista

A Lista é uma estrutura de dados cujo funcionamento é inspirado no de uma lista “natural”.

- Uma lista pode ser ordenada ou não;
- Quando for ordenada, pode o ser por alguma característica intrínseca dos dados (ex: ordem alfabética);
- Pode também refletir a ordem cronológica (ordem de inserção) dos dados.

Listas usando vetores

- Vetores possuem um espaço limitado para armazenar dados;
- Precisamos definir um espaço grande o suficiente para a nossa lista;
- Precisamos de um indicador de qual elemento do vetor é o atual último da lista.



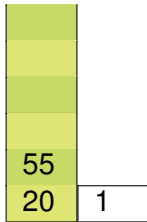
Listas usando vetores

- Vetores possuem um espaço limitado para armazenar dados;
- Precisamos definir um espaço grande o suficiente para a nossa lista;
- Precisamos de um indicador de qual elemento do vetor é o atual último da lista.
- Lista vazia!



Listas usando vetores

- Vetores possuem um espaço limitado para armazenar dados;
- Precisamos definir um espaço grande o suficiente para a nossa lista;
- Precisamos de um indicador de qual elemento do vetor é o atual último da lista.



55	
20	1

Listas usando vetores

- Vetores possuem um espaço limitado para armazenar dados;
- Precisamos definir um espaço grande o suficiente para a nossa lista;
- Precisamos de um indicador de qual elemento do vetor é o atual último da lista.

4	
55	
20	2

Listas usando vetores

- Vetores possuem um espaço limitado para armazenar dados;
- Precisamos definir um espaço grande o suficiente para a nossa lista;
- Precisamos de um indicador de qual elemento do vetor é o atual último da lista.

12	
4	
55	
20	3

Listas usando vetores

- Vetores possuem um espaço limitado para armazenar dados;
- Precisamos definir um espaço grande o suficiente para a nossa lista;
- Precisamos de um indicador de qual elemento do vetor é o atual último da lista.

89	
12	
4	
55	
20	4

Listas usando vetores

- Vetores possuem um espaço limitado para armazenar dados;
- Precisamos definir um espaço grande o suficiente para a nossa lista;
- Precisamos de um indicador de qual elemento do vetor é o atual último da lista.

24	
89	
12	
4	
55	
20	5

Listas usando vetores

- Vetores possuem um espaço limitado para armazenar dados;
- Precisamos definir um espaço grande o suficiente para a nossa lista;
- Precisamos de um indicador de qual elemento do vetor é o atual último da lista.
- Lista cheia!

24	
89	
12	
4	
55	
20	5

Modelagem da Lista em vetor

- Aspecto estrutural:
 - Necessitamos de um vetor para armazenar as informações;
 - Necessitamos de um indicador da posição atual do último elemento da lista.

```
constantes MAXLISTA ← 100;
```

```
classe Lista {  
    T _dados[MAXLISTA]; // vetor estatico  
    inteiro _ultimo;  
};
```

Modelagem da Lista em vetor

- Aspecto estrutural:
 - Necessitamos de um vetor para armazenar as informações;
 - Necessitamos de um indicador da posição atual do ultimo elemento da lista.

```
constantes MAXLISTA ← 100;
```

```
classe Lista {  
    T* _dados; // _dados ← new T[MAXLISTA];  
    inteiro _ultimo;  
    inteiro _tam ← MAXLISTA;  
};
```

Modelagem da Lista em vetor

- Aspecto funcional:
 - Temos que inserir e remover dados da lista;
 - Temos que testar se a lista está vazia ou cheia (dentro outros testes);
 - Temos que inicializar a lista e garantir a ordem de seus elementos.

Modelagem da Lista em vetor

- Inicializar ou limpar:
 - `Lista();`
 - `Lista(int tam);`
 - `limpaLista();`
 - `~Lista();`
- Testar se a lista está vazia ou cheia e outros testes:
 - `bool listaCheia();`
 - `bool listaVazia();`
 - `int posicao(dado)`
 - `bool contem(dado);`

Modelagem da Lista em vetor

- Inserir e remover dados da lista:
 - `adiciona(T dado);`
 - `adicionaNoInicio(T dado);`
 - `adicionaNaPosicao(T dado, int posicao);`
 - `adicionaEmOrdem(T dado);`
 - `T retira();`
 - `T retiraDoInicio();`
 - `T retiraDaPosicao(int posicao);`
 - `T retiraEspecifico(dado);`

Método Lista()

```
Lista()  
  inicio  
    _dados ← new T[_tam];  
    _ultimo ← -1;  
  fim
```

Método Lista(int tam)

```
Lista(int tam)
    inicio
        _tam ← tam;
        _dados ← new T[tam];
        ultimo ← -1;
    fim
```

Método limpaLista()

```
void limpaLista()  
    inicio  
        ultimo ← -1;  
    fim
```

Método Lista()

```
void ~Lista()  
    inicio  
        delete [] _dados;  
    fim
```

Observação:

Este método só é necessário quando há alocação dinâmica do vetor.

Método listaCheia()

```
bool listaCheia()  
    inicio  
        SE (_ultimo = _tam - 1) ENTAO  
            RETORNE(Verdadeiro);  
        SENA  
            RETORNE(Falso);  
    fim
```

Método listaVazia()

```
bool listaVazia()  
  inicio  
    SE (_ultimo = -1) ENTAO  
      RETORNE(Verdadeiro)  
    SENA  
      RETORNE(Falso);  
  fim
```


Método adiciona(T dado)

- Testamos se há espaço;
- Incrementamos o último;
- Adicionamos o novo dado na posição último.

12	
4	
55	
20	3

Método adiciona(T dado)

- Testamos se há espaço;
- Incrementamos o último;
- Adicionamos o novo dado na posição último.

12	
4	
55	
20	4

Método adiciona(T dado)

- Testamos se há espaço;
- Incrementamos o último;
- Adicionamos o novo dado na posição último.

89	
12	
4	
55	
20	4

Método adiciona(T dado)

```
adiciona(T dado)
  inicio
    SE (listaCheia) ENTAO
      THROW(ERROLISTACHEIA)
    SENA0
      _ultimo ← _ultimo + 1;
      _dados[_ultimo] ← dado;
    FIM SE
  fim
```

Método T_remove()

- Testamos se há elementos;
- Decrementamos o último;
- Devolvemos o último elemento.

89	
12	
4	
55	
20	4

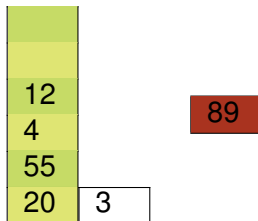
Método T retira()

- Testamos se há elementos;
- Decrementamos o último;
- Devolvemos o último elemento.

89	
12	
4	
55	
20	3

Método T_remove()

- Testamos se há elementos;
- Decrementamos o último;
- Devolvemos o último elemento.



Método T retira()

```
T retira()
    inicio
        SE (listaVazia) ENTAO
            THROW(ERROLISTAVAZIA)
        SENA0
            _ultimo ← _ultimo - 1;
            RETORNE(_dados[_ultimo + 1]);
        FIM SE
    fim
```


Método adicionaNoInicio(T dado)

- Testamos se há espaço;
- Incrementamos o último;
- Empurramos tudo para trás;
- Adicionamos o novo dado na primeira posição.

89	
12	
4	
55	
20	4

Método adicionaNoInicio(T dado)

- Testamos se há espaço;
- Incrementamos o último;
- Empurramos tudo para trás;
- Adicionamos o novo dado na primeira posição.

89	
12	
4	
55	
20	5

Método adicionaNoInicio(T dado)

- Testamos se há espaço;
- Incrementamos o último;
- Empurramos tudo para trás;
- Adicionamos o novo dado na primeira posição.

89	
12	
4	
55	
20	5

Método adicionaNoInicio(T dado)

- Testamos se há espaço;
- Incrementamos o último;
- Empurramos tudo para trás;
- Adicionamos o novo dado na primeira posição.

89	
12	
4	
55	
20	5

Método adicionaNoInicio(T dado)

- Testamos se há espaço;
- Incrementamos o último;
- Empurramos tudo para trás;
- Adicionamos o novo dado na primeira posição.

89	
12	
4	
55	
20	5

Método adicionaNoInicio(T dado)

- Testamos se há espaço;
- Incrementamos o último;
- Empurramos tudo para trás;
- Adicionamos o novo dado na primeira posição.

89	
12	
4	
55	
20	5

Método adicionaNoInicio(T dado)

- Testamos se há espaço;
- Incrementamos o último;
- Empurramos tudo para trás;
- Adicionamos o novo dado na primeira posição.

89	
12	
4	
55	
20	
	5

Método adicionaNoInicio(T dado)

- Testamos se há espaço;
- Incrementamos o último;
- Empurramos tudo para trás;
- Adicionamos o novo dado na primeira posição.

89	
12	
4	
55	
20	
42	5

Método adicionaNoInicio(T dado)

```
adicionaNoInicio(T dado)
    int posicao; // Var auxiliar para "caminhar"
    inicio
        SE (listaCheia) ENTAO
            THROW(ERROLISTACHEIA)
        SENAO
            _ultimo ← _ultimo + 1;
            posicao ← _ultimo;
            ENQUANTO (posicao > 0) FAÇA
                // Empurrar tudo para tras
                _dados[posicao] ← _dados[posicao - 1];
                posicao ← posicao - 1;
            FIM ENQUANTO
            _dados[0] ← dado;
        FIM SE
    fim
```

Método adicionaNoInicio(T dado)

```
adicionaNoInicio(T dado)
    int posicao; // Var auxiliar para "caminhar"
    inicio
        SE (listaCheia) ENTAO
            THROW(ERROLISTACHEIA)
        SENAO
            _ultimo ← _ultimo + 1;
            posicao ← _ultimo;
            ENQUANTO (posicao > 0) FAÇA
                // Empurrar tudo para tras
                _dados[posicao] ← _dados[posicao - 1];
                posicao ← posicao - 1;
            FIM ENQUANTO
            _dados[0] ← dado;
        FIM SE
    fim
```

Método adicionaNoInicio(T dado)

```
adicionaNoInicio(T dado)
    int posicao; // Var auxiliar para "caminhar"
    inicio
        SE (listaCheia) ENTAO
            THROW(ERROLISTACHEIA)
        SENAO
            _ultimo ← _ultimo + 1;
            posicao ← _ultimo;
            ENQUANTO (posicao > 0) FACA
                // Empurrar tudo para tras
                _dados[posicao] ← _dados[posicao - 1];
                posicao ← posicao - 1;
            FIM ENQUANTO
            _dados[0] ← dado;
        FIM SE
    fim
```

Método adicionaNoInicio(T dado)

```
adicionaNoInicio(T dado)
    int posicao; // Var auxiliar para "caminhar"
    inicio
        SE (listaCheia) ENTAO
            THROW(ERROLISTACHEIA)
        SENAO
            _ultimo ← _ultimo + 1;
            posicao ← _ultimo;
            ENQUANTO (posicao > 0) FAÇA
                // Empurrar tudo para tras
                _dados[posicao] ← _dados[posicao - 1];
                posicao ← posicao - 1;
            FIM ENQUANTO
            _dados[0] ← dado;
        FIM SE
    fim
```

Método T `retiraDoInicio()`

- Testamos se há elementos;
- Decrementamos o último;
- Salvamos o primeiro elemento;
- Empurramos tudo para a frente.

89	
12	
4	
55	
20	
42	5

Método T `retiraDoInicio()`

- Testamos se há elementos;
- Decrementamos o último;
- Salvamos o primeiro elemento;
- Empurramos tudo para a frente.

89	
12	
4	
55	
20	
42	4

Método T `retiraDoInicio()`

- Testamos se há elementos;
- Decrementamos o último;
- Salvamos o primeiro elemento;
- Empurramos tudo para a frente.

89	
12	
4	
55	
20	
42	4

42

Método T `retiraDoInicio()`

- Testamos se há elementos;
- Decrementamos o último;
- Salvamos o primeiro elemento;
- Empurramos tudo para a frente.

89	
12	
4	
55	
20	
20	4

42

Método T retiraDoInicio()

- Testamos se há elementos;
- Decrementamos o último;
- Salvamos o primeiro elemento;
- Empurramos tudo para a frente.

89	
12	
4	
55	
55	
20	4

42

Método T `retiraDoInicio()`

- Testamos se há elementos;
- Decrementamos o último;
- Salvamos o primeiro elemento;
- Empurramos tudo para a frente.

89	
12	
4	
4	
55	
20	4

42

Método T `retiraDoInicio()`

- Testamos se há elementos;
- Decrementamos o último;
- Salvamos o primeiro elemento;
- Empurramos tudo para a frente.

89	
12	
12	
4	
55	
20	4

42

Método T retiraDoInicio()

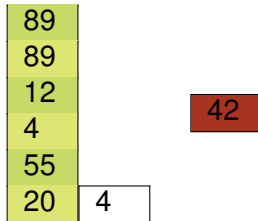
- Testamos se há elementos;
- Decrementamos o último;
- Salvamos o primeiro elemento;
- Empurramos tudo para a frente.

89	
89	
12	
4	
55	
20	4

42

Método T `retiraDoInicio()`

- Testamos se há elementos;
- Decrementamos o último;
- Salvamos o primeiro elemento;
- Empurramos tudo para a frente.
- O segundo “89” é desconsiderado, pois o `_ultimo` determina que a lista acaba antes dele!



Método T `retiraDoInicio()`

```
T retiraDoInicio()  
    int posicao;  
    T valor;  
    inicio  
        SE (listaVazia) ENTAO  
            THROW(ERROLISTAVAZIA)  
        SENAO  
            _ultimo  $\leftarrow$  _ultimo - 1;  
            valor  $\leftarrow$  _dados[0];  
            posicao  $\leftarrow$  0;  
            ENQUANTO (posicao  $\leq$  _ultimo) FACA  
                // Empurrar tudo para a frente  
                _dados[posicao]  $\leftarrow$  _dados[posicao + 1];  
                posicao  $\leftarrow$  posicao + 1;  
            FIM ENQUANTO  
            RETORNE(valor);  
        FIM SE  
fim
```

Método adicionaNaPosicao(T dado, int posicao)

- Testamos se há espaço e se a posição existe;
- Incrementamos o último;
- Empurramos tudo para trás a partir da posição;
- Adicionamos o novo dado na posição informada;
- **Exemplo:** inserir o **42** na posição 3.

89	
12	
4	
55	
20	4

Método adicionaNaPosicao(T dado, int posicao)

- Testamos se há espaço e se a posição existe;
- Incrementamos o último;
- Empurramos tudo para trás a partir da posição;
- Adicionamos o novo dado na posição informada;
- **Exemplo:** inserir o **42** na posição 3.

89	
12	
4	
55	
20	5

Método adicionaNaPosicao(T dado, int posicao)

- Testamos se há espaço e se a posição existe;
- Incrementamos o último;
- Empurramos tudo para trás a partir da posição;
- Adicionamos o novo dado na posição informada;
- **Exemplo:** inserir o **42** na posição 3.

89	
89	
12	
4	
55	
20	5

Método adicionaNaPosicao(T dado, int posicao)

- Testamos se há espaço e se a posição existe;
- Incrementamos o último;
- Empurramos tudo para trás a partir da posição;
- Adicionamos o novo dado na posição informada;
- **Exemplo:** inserir o **42** na posição 3.

89	
12	
12	
4	
55	
20	5

Método adicionaNaPosicao(T dado, int posicao)

- Testamos se há espaço e se a posição existe;
- Incrementamos o último;
- Empurramos tudo para trás a partir da posição;
- Adicionamos o novo dado na posição informada;
- **Exemplo:** inserir o **42** na posição 3.

89	
12	
42	
4	
55	
20	5

Método adicionaNaPosicao(T dado, int posicao)

```
adicionaNaPosicao(T dado, int posicao)
    int atual;
    inicio
        SE (listaCheia) ENTAO
            THROW(ERROLISTACHEIA)
        SENAO
            SE (posicao < 0 OU posicao > _ultimo+1) ENTAO
                THROW(ERROPOSICAO);
            FIM SE
            _ultimo ← _ultimo + 1;
            atual ← _ultimo;
            ENQUANTO (atual > posicao) FACA
                _dados[atual] ← _dados[atual - 1];
                atual ← atual - 1;
            FIM ENQUANTO
            _dados[posicao] ← dado;
        FIM SE
    fim
```

Equivalência de métodos

- `adicionaNoInicio(dado) \iff adicionaNaPosicao(dado,0)`
- `adiciona(dado) \iff adicionaNaPosicao(dado,ultimo+1)`

Método T `retiraDaPosicao(int posicao)`

- Testamos se há elementos e se a posição existe;
- Decrementamos o último;
- Salvamos elemento na posição;
- Empurramos tudo para frente até posição;
- **Exemplo:** remover da posição 3.

89	
12	
42	
4	
55	
20	5

Método T `retiraDaPosicao(int posicao)`

- Testamos se há elementos e se a posição existe;
- Decrementamos o último;
- Salvamos elemento na posição;
- Empurramos tudo para frente até posição;
- **Exemplo:** remover da posição 3.

89	
12	
42	
4	
55	
20	4

Método T `retiraDaPosicao(int posicao)`

- Testamos se há elementos e se a posição existe;
- Decrementamos o último;
- Salvamos elemento na posição;
- Empurramos tudo para frente até posição;
- **Exemplo:** remover da posição 3.

89	
12	
42	
4	
55	
20	4

42

Método T `retiraDaPosicao(int posicao)`

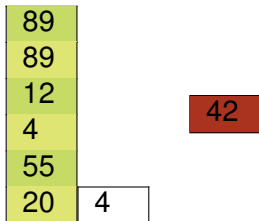
- Testamos se há elementos e se a posição existe;
- Decrementamos o último;
- Salvamos elemento na posição;
- Empurramos tudo para frente até posição;
- **Exemplo:** remover da posição 3.

89	
12	
12	
4	
55	
20	4

42

Método T `retiraDaPosicao(int posicao)`

- Testamos se há elementos e se a posição existe;
- Decrementamos o último;
- Salvamos elemento na posição;
- Empurramos tudo para frente até posição;
- **Exemplo:** remover da posição 3;
- O segundo “89” é desconsiderado, pois o `_ultimo` determina que a lista acaba antes dele!



Método T `retiraDaPosicao(int posicao)`

```
T retiraDaPosicao(int posicao)
    int atual;
    T valor;
    inicio
        SE (posicao < 0 OU posicao > _ultimo) ENTAO
            THROW(ERROPOSICAO)
        SENAO
            SE (listaVazia) ENTAO
                THROW(ERROLISTAVAZIA)
            SENAO
                _ultimo ← _ultimo - 1;
                valor ← _dados[posicao];
                atual ← posicao;
                ENQUANTO (atual ≤ _ultimo) FACA
                    _dados[atual] ← _dados[atual + 1];
                    atual ← atual + 1;
                FIM ENQUANTO
                RETORNE(valor);
            FIM SE
        FIM SE
    fim
```

Equivalência de métodos

- $T \text{ retira()} \iff \text{retiraDaPosicao}(\text{_ultimo})$
- $T \text{ retiraDoInicio()} \iff \text{retiraDaPosicao}(0)$

Método adicionaEmOrdem(T dado)

- Necessitamos de uma função para comparar os dados (próximo slide);
- Testamos se há espaço;
- Procuramos pela posição onde inserir comparando dados;
- Chamamos `adicionaNaPosicao()`.

89
55
42
12
4
4

Inserir 45

Método adicionaEmOrdem(T dado)

- Necessitamos de uma função para comparar os dados (próximo slide);
- Testamos se há espaço;
- Procuramos pela posição onde inserir comparando dados;
- Chamamos `adicionaNaPosicao()`.

89
55
42
12
4

4

Inserir 45

Método adicionaEmOrdem(T dado)

- Necessitamos de uma função para comparar os dados (próximo slide);
- Testamos se há espaço;
- Procuramos pela posição onde inserir comparando dados;
- Chamamos `adicionaNaPosicao()`.

89	
55	
42	
12	
4	4

Inserir 45

Método adicionaEmOrdem(T dado)

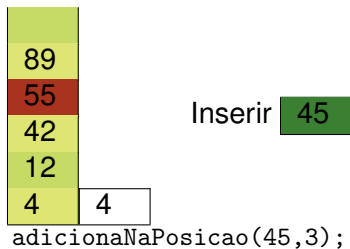
- Necessitamos de uma função para comparar os dados (próximo slide);
- Testamos se há espaço;
- Procuramos pela posição onde inserir comparando dados;
- Chamamos `adicionaNaPosicao()`.

89
55
42
12
4
4

Inserir 45

Método adicionaEmOrdem(T dado)

- Necessitamos de uma função para comparar os dados (próximo slide);
- Testamos se há espaço;
- Procuramos pela posição onde inserir comparando dados;
- Chamamos `adicionaNaPosicao()`.



Método `T::operator>(T t)`

- Quando o dado a ser armazenado em uma lista não for um tipo primitivo, este tem que implementar a sobrecarga do operador “>”;
- Para deixar os algoritmos de operações sobre lista independentes do tipo de dado específico armazenado na lista, usamos uma função do tipo `T::operator>(T t)`;
- Isso não deve ser implementado pelo programador da lista, mas pelo programador dos objetos que serão colocados na lista.

Método adicionaEmOrdem(T dado)

```
adicionaEmOrdem(T dado)
    int atual;
    inicio
        SE (listaCheia) ENTAO
            THROW(ERROLISTACHEIA)
        SENA0
            atual ← 0;
            ENQUANTO (atual ≤ _ultimo E
                      (dado > _dados[atual])) FACA
                // Encontrar posicao para inserir
                atual ← atual + 1;
            FIM ENQUANTO
            RETORNE(adicionaNaPosicao(dado, atual));
        FIM SE
    fim
```

- **Exercício:** melhorar essa implementação usando a técnica “dividir-para-conquistar”.

Método T `retiraEspecifico(dado)`

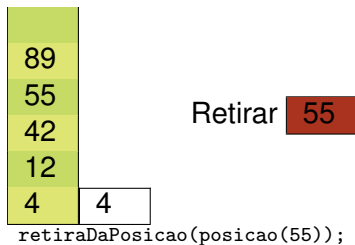
- Testamos se há elementos;
- Testamos se o dado existe e qual sua posição;

- Necessitamos de uma função

```
posicao(dado);
```

- Chamamos

```
retiraDaPosicao(posicao(dado));
```



Método int posicao(T dado)

```
posicao(T dado)
  int atual;
  inicio
    atual ← 0;
    ENQUANTO (atual ≤ _ultimo E dado ≠ _dados[atual]) FACA
      atual ← atual + 1;
    FIM ENQUANTO
    SE (atual > _ultimo) ENTAO
      THROW(ERROPOSICAO);
      // ou simplesmente 'RETORNA(atual);'
      // posicao invalida a ser verificada
      // (forma usada no 'find()' do VPL)
    SENA0
      RETORNE(atual);
    FIM SE
  fim
```

- Os seguintes métodos ficam por conta do aluno:
 - `T retiraEspecifico(T dado);`
 - `bool contem(dado);`
- Não esqueça da sobrecarga de operadores nos tipos não primitivos:
 - `T::operator=(T t);`
 - `T::operator<(T t);`
- **Exercício final:**
 - Sabe-se que a implementação de **Lista em vetor** pode ser feita **sem qualquer movimentação de dados**.
 - Projete e aplique essa ideia em sua implementação para tornar o código mais eficiente.

Trabalho Lista em vetor

- Implemente uma classe Lista com todas as operações vistas;
- Implemente a lista usando *templates*;
- Implemente a lista com um número de elementos variável definido na instanciação;
- Use as melhores práticas de orientação a objetos;
- Documente todas as classes, métodos e atributos;
- Aplique os testes unitários disponíveis no Moodle da disciplina para validar sua estrutura de dados;
- Entregue até a data definida no Moodle.

Perguntas?





Este trabalho está licenciado sob uma Licença Creative Commons Atribuição 4.0 Internacional. Para ver uma cópia desta licença, visite

<http://creativecommons.org/licenses/by/4.0/>.

