### 

### 

### **O que é um Tipo Abstrato de Dados (TAD)?**

Um Tipo Abstrato de Dados (TAD) é um conceito da ciência da computação que define um tipo de dado pelo comportamento (operações) que ele oferece, sem especificar como essas operações são implementadas. Em outras palavras, é uma forma de encapsular dados e operações de forma que a implementação interna é escondida do usuário. O foco está no "o que" o TAD faz, e não no "como" ele faz.

### **Principais Características dos TADs**

1. **Encapsulamento**: Os detalhes internos (implementação) são escondidos dos usuários do TAD. Os usuários interagem com o TAD apenas através de sua interface pública.
2. **Modularidade**: Um TAD é um módulo que pode ser desenvolvido, testado e mantido independentemente do restante do programa.
3. **Reusabilidade**: Uma vez definido, um TAD pode ser reutilizado em diferentes partes de um programa ou em diferentes programas.
4. **Consistência**: O TAD garante que as operações mantenham o estado interno consistente.

### **Exemplos de TADs Comuns**

Alguns exemplos clássicos de TADs são:

* Listas
* Pilhas (Stacks)
* Filas (Queues)
* Árvores (Trees)
* Grafos (Graphs)

### **Implementando um TAD em C#**

Vamos criar um exemplo simples de um TAD em C#: uma Pilha (Stack). Uma pilha é uma estrutura de dados que segue o princípio LIFO (Last In, First Out), onde o último elemento inserido é o primeiro a ser removido.

#### **Interface da Pilha**

Primeiro, definimos a interface da nossa Pilha, que especifica as operações que ela deve suportar:

csharp

Copiar código

public interface IStack<T>

{

void Push(T item);

T Pop();

T Peek();

bool IsEmpty();

}

#### **Implementação da Pilha**

Em seguida, implementamos essa interface. A implementação específica (como armazenamos os dados, por exemplo) está escondida dos usuários da pilha.

csharp

Copiar código

using System;

using System.Collections.Generic;

public class Stack<T> : IStack<T>

{

private List<T> \_items = new List<T>();

public void Push(T item)

{

\_items.Add(item);

}

public T Pop()

{

if (IsEmpty())

{

throw new InvalidOperationException("The stack is empty.");

}

T item = \_items[\_items.Count - 1];

\_items.RemoveAt(\_items.Count - 1);

return item;

}

public T Peek()

{

if (IsEmpty())

{

throw new InvalidOperationException("The stack is empty.");

}

return \_items[\_items.Count - 1];

}

public bool IsEmpty()

{

return \_items.Count == 0;

}

}

### **Usando o TAD**

Finalmente, podemos usar a nossa pilha sem nos preocuparmos com como ela é implementada:

csharp

Copiar código

public class Program

{

public static void Main()

{

IStack<int> stack = new Stack<int>();

stack.Push(1);

stack.Push(2);

stack.Push(3);

Console.WriteLine(stack.Pop()); // Output: 3

Console.WriteLine(stack.Peek()); // Output: 2

Console.WriteLine(stack.Pop()); // Output: 2

Console.WriteLine(stack.Pop()); // Output: 1

Console.WriteLine(stack.IsEmpty()); // Output: True

}

}

### **Conclusão**

Um TAD permite que você defina tipos de dados e suas operações de uma maneira que a implementação interna fique oculta, facilitando a manutenção, reutilização e modularidade do código. Em C#, usamos interfaces para definir o comportamento do TAD e classes para implementar esse comportamento. Essa separação de interface e implementação é fundamental para criar sistemas flexíveis e robustos.

encad

## **Aula sobre Listas Encadeadas em C#**

### **Introdução**

Listas encadeadas são estruturas de dados dinâmicas que consistem em uma sequência de elementos chamados nós (nodes), onde cada nó contém um valor e uma referência (ponteiro) para o próximo nó na sequência. Existem dois tipos principais de listas encadeadas:

1. **Listas Simplesmente Encadeadas**
2. **Listas Duplamente Encadeadas**

### **Listas Simplesmente Encadeadas**

Uma **Lista Simplesmente Encadeada** é uma coleção de nós onde cada nó contém um valor e uma referência para o próximo nó na lista. A estrutura básica de um nó em uma lista simplesmente encadeada é a seguinte:

#### **Estrutura de um Nó**

csharp

Copiar código

public class Node

{

public int Value { get; set; }

public Node Next { get; set; }

public Node(int value)

{

Value = value;

Next = null;

}

}

#### **Estrutura da Lista Simplesmente Encadeada**

csharp

Copiar código

public class SinglyLinkedList

{

private Node head;

public SinglyLinkedList()

{

head = null;

}

// Método para adicionar um novo nó no início da lista

public void AddFirst(int value)

{

Node newNode = new Node(value);

newNode.Next = head;

head = newNode;

}

// Método para remover o primeiro nó da lista

public int RemoveFirst()

{

if (head == null)

{

throw new InvalidOperationException("A lista está vazia");

}

int value = head.Value;

head = head.Next;

return value;

}

// Método para imprimir todos os elementos da lista

public void PrintList()

{

Node current = head;

while (current != null)

{

Console.Write(current.Value + " -> ");

current = current.Next;

}

Console.WriteLine("null");

}

}

#### **Explicação**

* **AddFirst**: Adiciona um novo nó no início da lista. O novo nó aponta para o antigo nó da cabeça, e a cabeça é atualizada para o novo nó.
* **RemoveFirst**: Remove e retorna o valor do primeiro nó da lista. A cabeça é atualizada para o próximo nó.
* **PrintList**: Imprime todos os elementos da lista, do primeiro ao último.

#### **Exemplo de Uso**

csharp

Copiar código

SinglyLinkedList list = new SinglyLinkedList();

list.AddFirst(10);

list.AddFirst(20);

list.AddFirst(30);

list.PrintList(); // Saída: 30 -> 20 -> 10 -> null

Console.WriteLine(list.RemoveFirst()); // Saída: 30

list.PrintList(); // Saída: 20 -> 10 -> null

### **Listas Duplamente Encadeadas**

Uma **Lista Duplamente Encadeada** é semelhante a uma lista simplesmente encadeada, mas cada nó contém uma referência para o próximo nó e uma referência para o nó anterior. Isso permite a navegação bidirecional.

#### **Estrutura de um Nó**

csharp

Copiar código

public class DoublyNode

{

public int Value { get; set; }

public DoublyNode Next { get; set; }

public DoublyNode Previous { get; set; }

public DoublyNode(int value)

{

Value = value;

Next = null;

Previous = null;

}

}

#### **Estrutura da Lista Duplamente Encadeada**

csharp

Copiar código

public class DoublyLinkedList

{

private DoublyNode head;

private DoublyNode tail;

public DoublyLinkedList()

{

head = null;

tail = null;

}

// Método para adicionar um novo nó no início da lista

public void AddFirst(int value)

{

DoublyNode newNode = new DoublyNode(value);

if (head == null)

{

head = newNode;

tail = newNode;

}

else

{

newNode.Next = head;

head.Previous = newNode;

head = newNode;

}

}

// Método para remover o primeiro nó da lista

public int RemoveFirst()

{

if (head == null)

{

throw new InvalidOperationException("A lista está vazia");

}

int value = head.Value;

head = head.Next;

if (head != null)

{

head.Previous = null;

}

else

{

tail = null;

}

return value;

}

// Método para imprimir todos os elementos da lista

public void PrintList()

{

DoublyNode current = head;

while (current != null)

{

Console.Write(current.Value + " <-> ");

current = current.Next;

}

Console.WriteLine("null");

}

}

#### **Explicação**

* **AddFirst**: Adiciona um novo nó no início da lista. O novo nó aponta para a antiga cabeça, e a cabeça é atualizada para o novo nó. Se a lista estiver vazia, a cauda também é atualizada para o novo nó.
* **RemoveFirst**: Remove e retorna o valor do primeiro nó da lista. A cabeça é atualizada para o próximo nó. Se a nova cabeça não for nula, a referência anterior da cabeça é ajustada para nulo. Se a lista ficar vazia, a cauda também é ajustada para nulo.
* **PrintList**: Imprime todos os elementos da lista, do primeiro ao último, com setas duplas para indicar a ligação bidirecional.

#### **Exemplo de Uso**

csharp

Copiar código

DoublyLinkedList list = new DoublyLinkedList();

list.AddFirst(10);

list.AddFirst(20);

list.AddFirst(30);

list.PrintList(); // Saída: 30 <-> 20 <-> 10 <-> null

Console.WriteLine(list.RemoveFirst()); // Saída: 30

list.PrintList(); // Saída: 20 <-> 10 <-> null

### **Conclusão**

As listas encadeadas são estruturas de dados flexíveis que permitem a inserção e remoção eficiente de elementos. As listas simplesmente encadeadas são mais simples e usam menos memória, pois cada nó tem apenas uma referência. As listas duplamente encadeadas, por outro lado, permitem a navegação bidirecional, o que pode ser vantajoso em certas aplicações, embora usem mais memória devido à referência adicional.