Algoritmos e Estrutura de Dados II

Aula 17

Tabela Hash ou Tabela de Dispersão

Prof. Dr. Dilermando Piva Jr

2º Semestre - CDN

Tabela Hash ou Tabela de Dispersão

- As **tabelas hash** são a base do armazenamento e recuperação eficiente de dados no desenvolvimento de software. Ao fornecer acesso rápido aos dados por meio de chaves exclusivas, as tabelas hash permitem pesquisas, inserções e exclusões em alta velocidade, tornando-as indispensáveis em cenários onde o desempenho é crítico, como indexação de banco de dados e soluções de cache.
- A essência de uma tabela hash está em seu mecanismo de hash, **que converte uma chave em um índice de array usando uma função hash**. Este índice escolhido determina onde o valor correspondente é armazenado no array.
- Ao garantir que esta função distribua as chaves uniformemente pelo array e ao empregar técnicas avançadas de resolução de colisões, as tabelas hash podem minimizar colisões e otimizar os tempos de recuperação de dados.

Tabela Hash ou Tabela de Dispersão

As tabelas de dispersão ou tabelas hashing, consistem no armazenamento de cada elemento em um determinado endereço calculado a partir da aplicação de uma função sobre a chave de busca. Matematicamente teríamos o seguinte:

$$e = f(c)$$

Onde, e é o endereço; c é a chave de busca e $f(c)$ é a função que tem como entrada a chave de busca.

Tabela Hash ou Tabela de Dispersão

Dessa forma, o processo de pesquisa sobre elementos organizados dessa forma é similar a um acesso direto ao elemento pesquisado.

Exemplo:

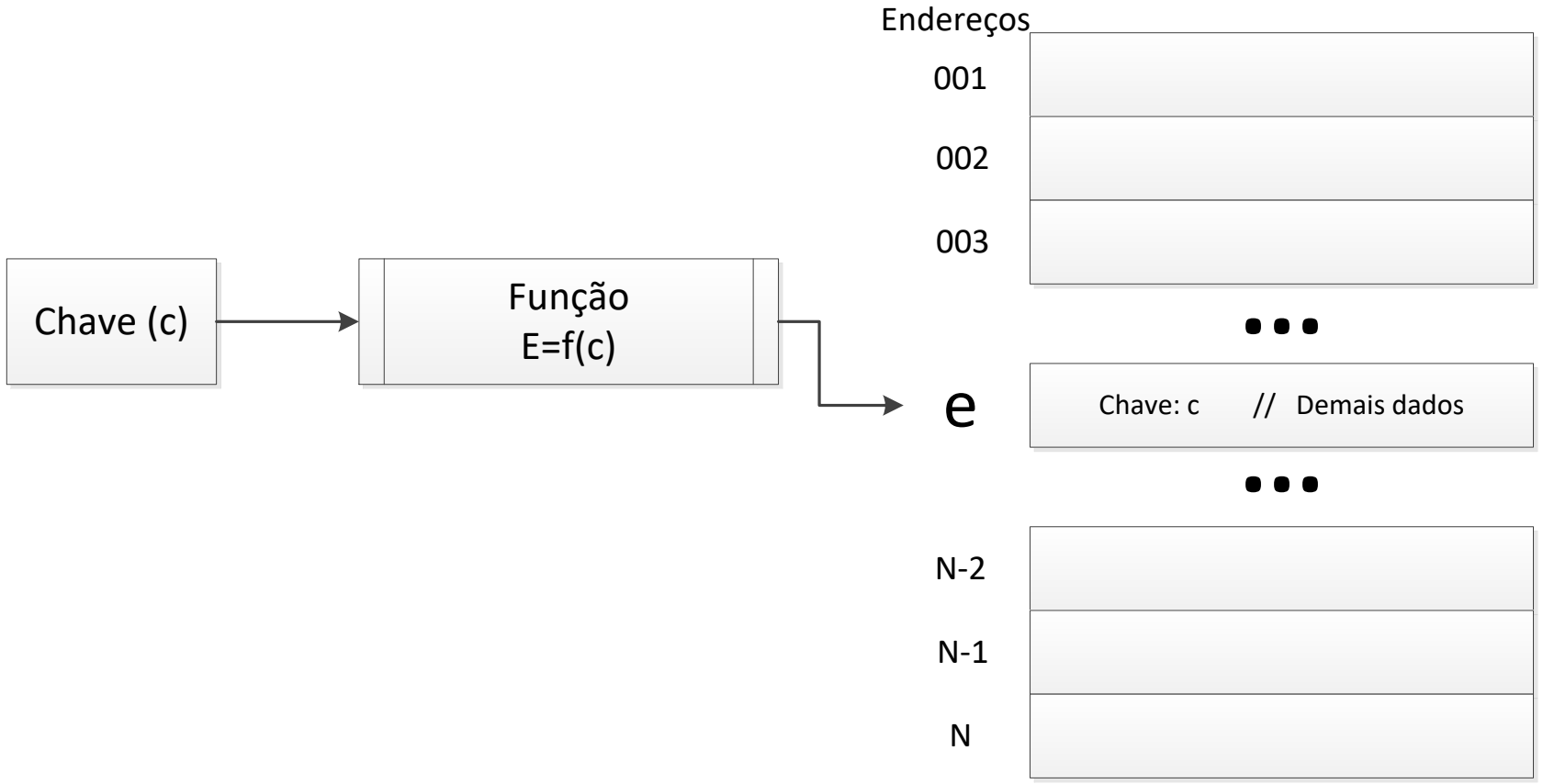


Tabela Hash ou Tabela de Dispersão

- A eficiência da pesquisa neste tipo de organização dos dados depende, indubitavelmente, da função de cálculo do endereço ($f(c)$).
- A função ideal seria aquela que pudesse gerar um endereço diferente para cada elemento da tabela (chave de busca).
- Entretanto, isso é praticamente inviável, principalmente pela dinâmica de atualização dos dados e crescimento da quantidade de elementos na tabela.

Tabela Hash ou Tabela de Dispersão

- Vamos entender esse conceito trabalhando com algo concreto.
- Supondo os dados constantes no vetor abaixo, acrescido da posição de cada elemento, teríamos:

Endereço →	0	1	2	3	4	5	6	7	8	9
Valores →	30	11	72	83	44	65	86	17	58	9

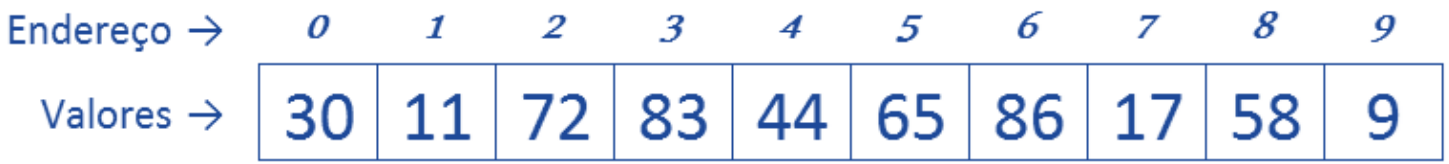
- Pensando nessa tabela e organização dos dados, poderíamos sugerir uma função de dispersão que tivesse como entrada o valor do elemento (chave de busca) e a função retornaria o endereço no vetor. Na amostra de dados, uma função possível seria:

$$f(c) = (c \bmod 10)$$

Onde **mod** corresponde ao resto da divisão inteira da chave **c** por **10** (número de elementos da tabela).

Tabela Hash ou Tabela de Dispersão

- Podemos verificar se a função de dispersão proposta é eficiente, testando alguns valores, por exemplo:



Chave (c)	c mod 10	f	Acesso direto?
30	30 mod 10	0	Ok! → v[0] = 30
44	44 mod 10	4	Ok! → v[4] = 44
9	9 mod 10	9	Ok! → v[9] = 9
72	72 mod 10	2	Ok! → v[2] = 72

Tabela Hash - COLISÃO

- O problema acontece quando existe uma possibilidade de atualização de valores.

Endereço →	0	1	2	3	4	5	6	7	8	9
Valores →	30	11	72	83	44	65	86	17	58	9

- Por exemplo, vamos imaginar que queiramos mudar o valor da posição 0 (de 30 para 91). A função agora não é mais eficiente, pois se aplicarmos a função utilizando como chave o valor 91, teremos o endereço igual a 1.
- Como pode observar, o endereço 1 já está ocupado pelo valor 11.
- A esse fenômeno (dois valores distintos resultarem e um mesmo endereço quando aplicados a uma função de dispersão) chamamos de **colisão**.

Tabela Hash - COLISÃO

- A colisão é um fenômeno comum, e pode ser tratado de diversas formas.
- Apenas para você poder entender melhor o processo de colisão no mundo real, se você possui um smartphone, certamente você tem um aplicativo de gerenciamento de contatos.
- Nesse aplicativo, existe várias maneiras de acessar os contatos. Uma delas é escolhendo a letra inicial do nome.
- Quando você escolhe uma determinada letra, todos os nomes que começam com aquela letra em específico são exibidos.
- Essa é uma maneira de entendermos uma tabela de dispersão.
- A letra inicial do nome é a entrada de uma função, que separa todos os nomes que iniciam com aquela letra em específico.
- Depois disso, a busca pelo nome desejado, fica bem mais rápida. A figura ao lado ilustra essa implementação e o controle das colisões.

Tabela Hash - COLISÃO

Existem várias formas de trabalhar as colisões.

Uma das mais utilizadas é a implementação de **listas encadeadas** a partir do endereço base.

Exemplo:

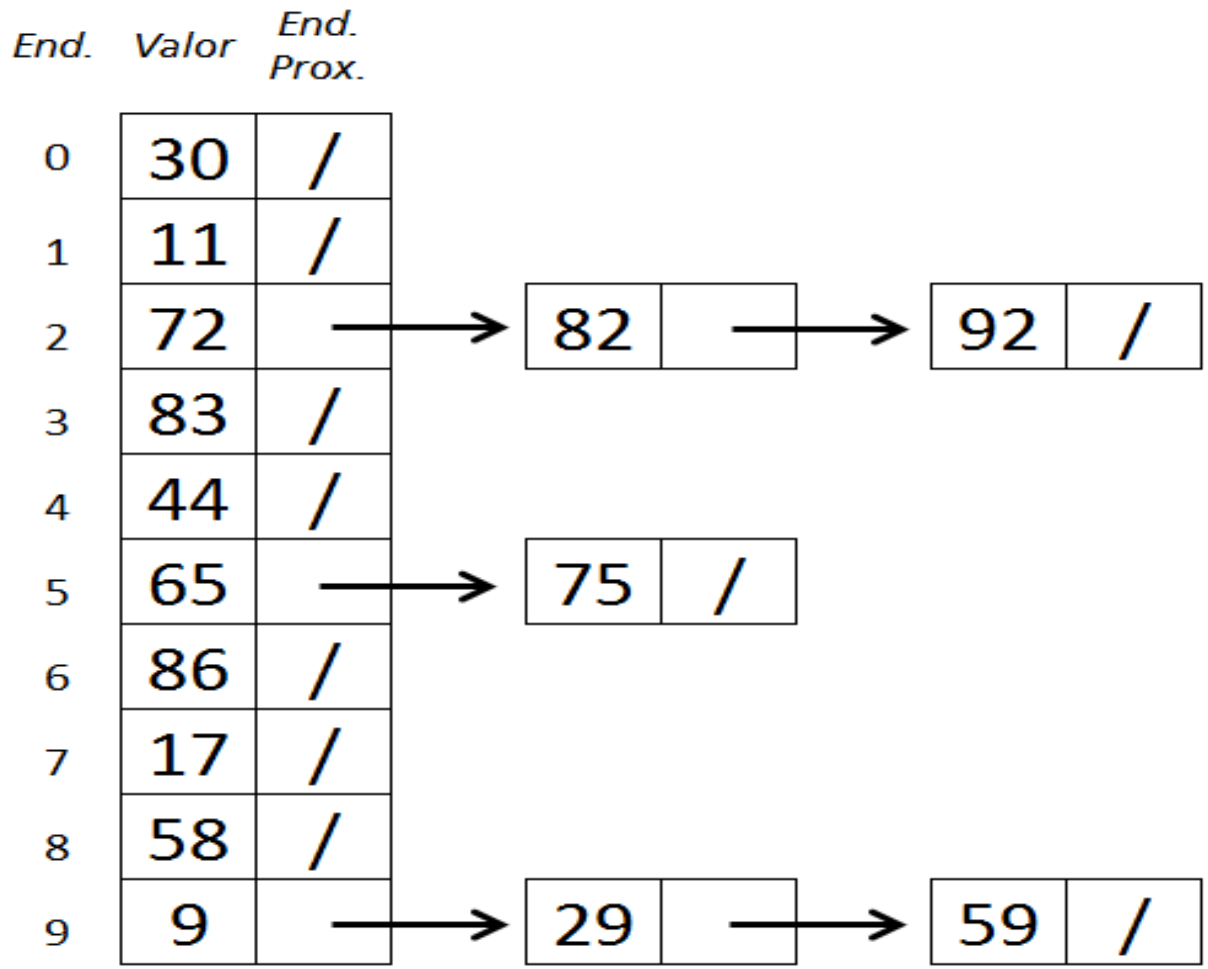


Tabela Hash - PRINCIPAIS APLICAÇÕES

- **Indexação de banco de dados**: as tabelas hash fornecem recuperação rápida de dados, o que é essencial para o desempenho dos sistemas de indexação de banco de dados.
- **Armazenamento em cache**: as tabelas hash são ideais para aplicativos de armazenamento em cache onde a pesquisa rápida de dados armazenados em cache é crucial. Eles permitem inserções, pesquisas e exclusões eficientes.
- **Desduplicação de dados**: em cenários onde a redundância de dados deve ser minimizada, as tabelas hash podem ajudar a identificar rapidamente dados duplicados.
- **Matrizes Associativas**: Muitas linguagens de programação usam tabelas hash para implementar matrizes associativas (também conhecidas como mapas ou dicionários), que podem recuperar e armazenar dados com base em chaves definidas pelo usuário.
- **Representação de dados exclusivos**: tabelas hash são úteis para manter conjuntos de itens exclusivos e são amplamente utilizadas em implementações que exigem verificações contra repetição.

Tabela Hash - IMPLEMENTAÇÃO EM PYTHON

- Em Python, a função hash pode ser chamada em qualquer objeto para retornar um valor inteiro, que chamamos de código hash ou valor hash. Vejamos alguns exemplos:

```
print(hash(“abc”))      # Exemplo de string
print(hash("123"))      # Exemplo de string numérica
print(hash(45))         # Exemplo de inteiro
print(hash(45.0))       # Exemplo de número de ponto flutuante
print(hash(45.3))       # Exemplo de outro número de ponto flutuante
print(hash(True))       # Exemplo de booleano True
print(hash(False))      # Exemplo de booleano False

try:
    print(hash([1, 2, 3]))  # Exemplo de lista (imutável)
except TypeError as e:
    print(e)  # Exibe o erro de tipo
```

VAMOS PARA A PRÁTICA ?!!!

Tabela Hash - IMPLEMENTAÇÃO EM PYTHON

Construindo a Classe HashSet

- Para implementar uma tabela hash em Python, vamos começar criando uma classe chamada **HashSet**.
- Esta classe irá conter uma lista (**items**) para armazenar os elementos e um contador (**numItems**) para acompanhar o número de itens na lista.
- Inicialmente, a lista será preenchida com valores **None**, que indicarão posições vazias.
- Primeiro, precisamos entender que a função hash em Python retorna um valor inteiro para um objeto, chamado de código hash.
- Este valor será usado para calcular o índice na lista onde o item será armazenado.
- Para garantir que o índice esteja dentro dos limites da lista, usaremos o operador **% (mod)**, que retorna o resto da divisão do código hash pelo comprimento da lista.

Tabela Hash - IMPLEMENTAÇÃO EM PYTHON

Definindo a Classe HashSet:

- A classe terá um construtor (__init__) que inicializa a lista de itens e o contador de itens.
- O construtor também permite inicializar o conjunto com um conteúdo opcional.

Tabela Hash - IMPLEMENTAÇÃO EM PYTHON

```
class HashSet:
    class __Placeholder:
        def __init__(self):
            pass

        def __eq__(self, other):
            return False

    def __init__(self, contents=[]):
        """
        Inicializa a tabela hash.

        :param contents: lista opcional de itens para inicializar o HashSet
        """
        self.items = [None] * 10  # Inicializa a lista com 10 posições, todas definidas como None
        self.numItems = 0  # Inicializa o contador de itens com 0
        for item in contents:
            self.add(item)  # Adiciona os itens iniciais ao conjunto
```

Tabela Hash - IMPLEMENTAÇÃO EM PYTHON

```
def add(self, item):
    """
    Adiciona um item à tabela hash.

    :param item: item a ser adicionado
    """
    if self.__add(item, self.items):
        self.numItems += 1
        load = self.numItems / len(self.items)
        if load >= 0.75:
            self.items = self.__rehash(self.items, [None] * 2 * len(self.items))
```

Tabela Hash - IMPLEMENTAÇÃO EM PYTHON

- O método add insere um item na tabela hash. Primeiro, calcula-se o índice na lista usando o código hash do item. Se a posição está vazia (None), o item é inserido diretamente. Se não, é necessário lidar com colisões.
- Aqui, o método add chama a função __add para inserir o item. Se o item é inserido com sucesso, incrementamos o contador numItems. Se a taxa de ocupação (*load factor*) exceder 75%, refazemos o hash da lista para o dobro do tamanho atual.
-
- **Função Estática de Adição (add)**
- A função estática __add insere um item em uma lista de itens, lidando com colisões usando *linear probing*.

Tabela Hash - IMPLEMENTAÇÃO EM PYTHON

```
@staticmethod
def __add(item, items):
    """
    Função estática para adicionar um item a uma lista de itens com tratamento de colisão.

    :param item: item a ser adicionado
    :param items: lista de itens
    :return: True se o item foi adicionado, False se o item já estava na lista
    """
    idx = hash(item) % len(items)  # Calcula o índice usando o código hash e o operador mod
    loc = -1  # Inicializa a variável loc para rastrear posições de placeholders
    while items[idx] is not None:
        if items[idx] == item:
            return False  # Item já está no conjunto
        if loc < 0 and isinstance(items[idx], HashSet.__Placeholder):
            loc = idx  # Marca a posição do placeholder
        idx = (idx + 1) % len(items)  # Avança para o próximo índice (linear probing)

    if loc < 0:
        loc = idx  # Define a posição de inserção como o índice atual se nenhum placeholder foi encontrado
    items[loc] = item  # Insere o item na posição determinada
    return True
```

Tabela Hash - IMPLEMENTAÇÃO EM PYTHON

ATENÇÃO!

- O **@staticmethod** é um decorador em Python que indica que um método dentro de uma classe é um método estático. Métodos estáticos não dependem do estado da instância da classe (ou seja, não usam self) e não podem acessar ou modificar o estado da instância ou da classe. Eles são como funções normais que são definidas dentro de uma classe por questões de organização e estrutura do código.
- Vantagens dos Métodos Estáticos:
  - **Organização**: Agrupam funções que têm uma relação lógica com a classe, mas não dependem do estado da instância;
  - **Encapsulamento**: Mantêm funções relacionadas dentro da definição da classe;
  - **Facilidade de Uso**: Podem ser chamados diretamente na classe sem a necessidade de criar uma instância da classe.

Tabela Hash - IMPLEMENTAÇÃO EM PYTHON

Função Estática de Rehash (rehash)

Se a taxa de ocupação exceder 75%, precisamos aumentar o tamanho da lista e redistribuir os itens. A função **__rehash** percorre a lista antiga e refaz o hash de cada item não-nulo e que não seja um placeholder na nova lista.

```
@staticmethod
def __rehash(oldList, newList):
    """
    Re-hash dos itens de uma lista antiga para uma nova lista.

    :param oldList: lista antiga de itens
    :param newList: nova lista de itens
    :return: nova lista com itens re-hashados
    """
    for x in oldList:
        if x is not None and not isinstance(x, HashSet.__Placeholder):
            HashSet.__add(x, newList)  # Adiciona os itens não-nulos e não-placeholders à nova lista
    return newList
```

Tabela Hash - IMPLEMENTAÇÃO EM PYTHON

Excluindo um Item

Para excluir um item, precisamos encontrar sua posição na lista. Se o item estiver no meio de uma cadeia, substituímos pelo objeto **__Placeholder**.

```
    def remove(self, item):
        """
        Remove um item da tabela hash.

        :param item: item a ser removido
        :raises KeyError: se o item não estiver na tabela
        """
        if HashSet.__remove(item, self.items):
            self.numItems -= 1
            load = max(self.numItems, 10) / len(self.items)
            if load <= 0.25:
                self.items = HashSet.__rehash(self.items, [None] * int(len(self.items) / 2))
        else:
            raise KeyError("Item não está no HashSet")
```

Tabela Hash - IMPLEMENTAÇÃO EM PYTHON

Excluindo um Item

Para excluir um item, precisamos encontrar sua posição na lista. Se o item estiver no meio de uma cadeia, substituímos pelo objeto **__Placeholder**.

```
    @staticmethod
    def __remove(item, items):
        """
        Função estática para remover um item de uma lista de itens.

        :param item: item a ser removido
        :param items: lista de itens
        :return: True se o item foi removido, False se o item não estava na lista
        """
        idx = hash(item) % len(items)
        while items[idx] is not None:
            if items[idx] == item:
                nextIdx = (idx + 1) % len(items)
                if items[nextIdx] is None:
                    items[idx] = None
                else:
                    items[idx] = HashSet.__Placeholder()
                return True
            idx = (idx + 1) % len(items)
        return False
```

Tabela Hash - IMPLEMENTAÇÃO EM PYTHON

Encontrando um Item

Para encontrar um item, usamos o método **__contains__** que verifica se o item está na lista.

```
def __contains__(self, item):
    """
    Verifica se o item está na tabela hash.

    :param item: item a ser verificado
    :return: True se o item está na tabela, False caso contrário
    """
    idx = hash(item) % len(self.items)
    while self.items[idx] is not None:
        if self.items[idx] == item:
            return True
        idx = (idx + 1) % len(self.items)
    return False
```

Tabela Hash - IMPLEMENTAÇÃO EM PYTHON

Iterando Sobre os Itens

Para iterar sobre os itens do conjunto, definimos o método **__iter__** que percorre a lista de itens.

```
    def __iter__(self):
        """
        Itera sobre os itens na tabela hash.

        :yield: próximo item na tabela
        """
        for i in range(len(self.items)):
            if self.items[i] is not None and not isinstance(self.items[i], HashSet.__Placeholder):
                yield self.items[i]
```

Tabela Hash - EXEMPLO DE USO...

Exemplo de Uso
Vamos ver um exemplo de uso da classe HashSet:

```
# Criando um HashSet com alguns itens iniciais
hash_set = HashSet([1, 2, 3])

# Adicionando itens ao HashSet
hash_set.add(4)
hash_set.add(5)

# Exibindo os itens do HashSet
print(list(hash_set))  # Saída: [1, 2, 3, 4, 5]

# Tentando adicionar um item duplicado
hash_set.add(3)  # Não adiciona, pois 3 já está no conjunto

# Exibindo o número de itens no HashSet
print(hash_set.numItems)  # Saída: 5

# Removendo um item
hash_set.remove(3)
print(list(hash_set))  # Saída: [1, 2, 4, 5]

# Verificando se um item está no HashSet
print(2 in hash_set)  # Saída: True
print(3 in hash_set)  # Saída: False

# Iterando sobre os itens do HashSet
for item in hash_set:
    print(item)  # Saída: 1 2 4 5
```

VAMOS PARA A PRÁTICA ?!!!

Tabela Hash - IMPLEMENTAÇÃO EM PYTHON

- A implementação de uma tabela hash em Python envolve a criação de uma função hash eficiente e métodos para inserção, busca e exclusão de elementos.
- O tratamento de colisões é essencial para garantir que a tabela funcione corretamente em casos onde múltiplas chaves geram o mesmo índice.
- Embora existam várias técnicas avançadas para tratamento de colisões e otimização do desempenho da tabela hash, a implementação básica apresentada aqui é suficiente para fornecer uma noção inicial sobre o funcionamento dessas estruturas.