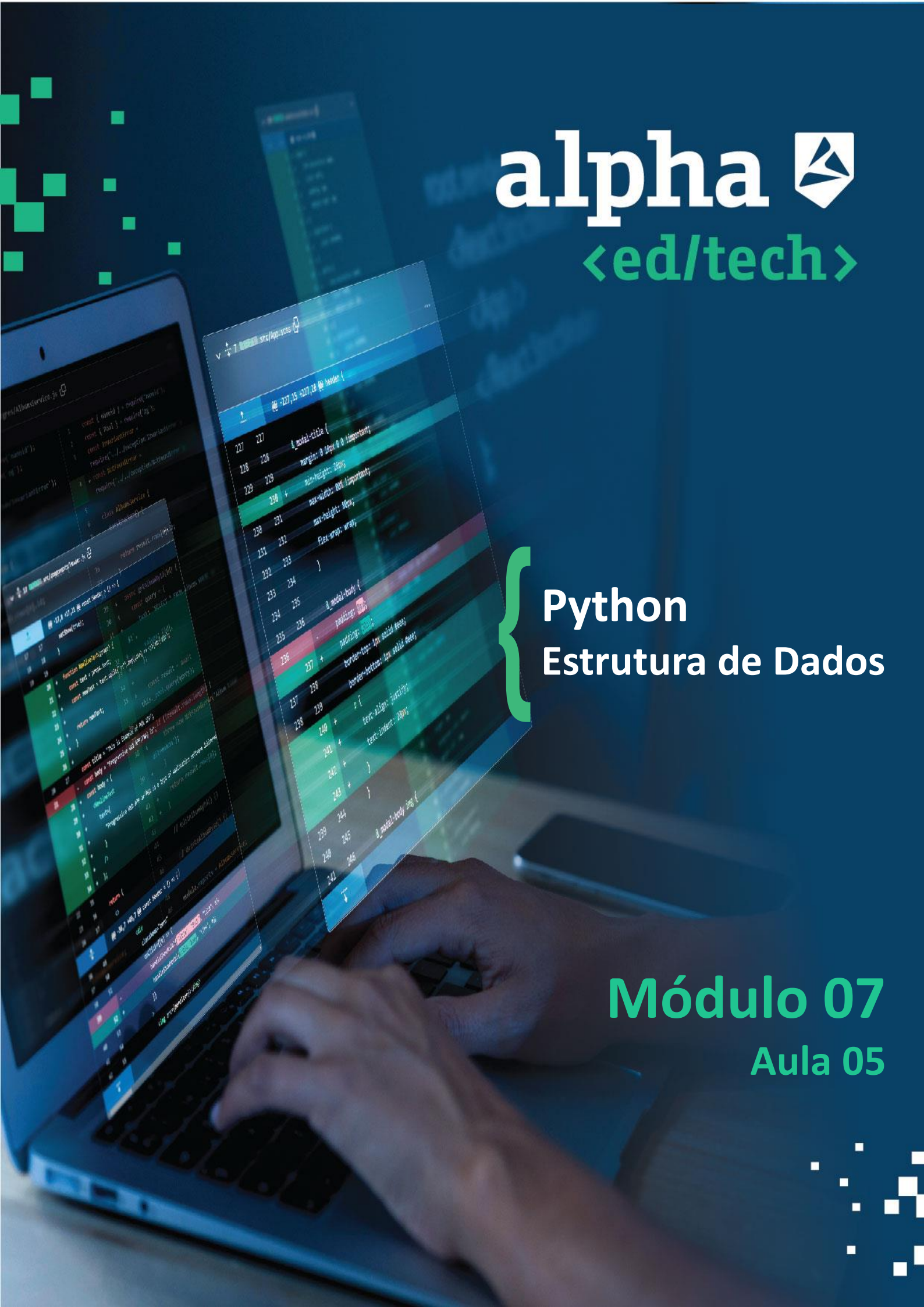


alpha
<ed/tech>

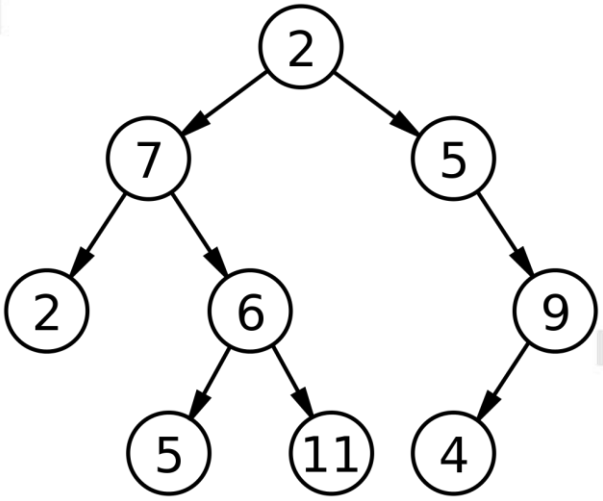
Python
Estrutura de Dados

Módulo 07
Aula 05

Estrutura de Dados

Árvores Binárias

As árvores binárias são estruturas de dados fundamentais na ciência da computação, caracterizadas por sua natureza hierárquica e organização específica. Consistem em nós interconectados, onde cada nó tem até dois filhos. Cada nó em uma árvore binária tem no máximo dois filhos, conhecidos como filho esquerdo e filho direito.



Estrutura de Uma Árvore Binária

- **Nó Raiz:** O topo da árvore é chamado de raiz, e é a partir dela que todos os outros nós são acessíveis.
- **Subárvores:** Cada filho de um nó pode ser a raiz de sua própria subárvore, replicando a estrutura binária.
- **Folhas:** Nós sem filhos são chamados de folhas ou nós terminais.

Propriedades Importantes

- **Altura da Árvore:** A distância do nó raiz até o nó mais distante.
- **Profundidade de um Nó:** O número de arestas do nó raiz até o nó em questão.
- **Nível de um Nó:** Profundidade do nó mais um.

Tipos de Árvores Binárias

- **Árvore Binária Completa:** Todos os níveis, exceto talvez o último, estão completamente preenchidos.
- **Árvore Binária Balanceada:** A diferença entre as alturas das subárvores de qualquer nó não é mais do que um.
- **Árvore Binária de Busca (BST):** Uma árvore especializada onde cada nó tem um valor, de modo que todos os valores na subárvore esquerda são menores que o valor do nó e todos os valores na subárvore direita são maiores.

Por Que Usar Árvores Binárias

- **Eficiência em Operações:** Busca, inserção e exclusão podem ser realizadas eficientemente em árvores binárias balanceadas.
- **Aplicações Diversas:** Desde a implementação de estruturas de dados mais complexas, como mapas e conjuntos, até o uso em processamento de linguagem natural e sistemas de banco de dados.

Percursos em Árvores Binárias

Os percursos em árvores binárias são métodos sistemáticos para visitar todos os nós de uma árvore. Cada método oferece uma sequência diferente de visitas, útil em diferentes aplicações. Os três percursos principais são: Pré Ordem, Em Ordem e Pós Ordem.

Percurso em Pré Ordem

No percurso em pré ordem, cada nó é processado antes de seus sub nós.

Sequência de Passos:

1. Visitar o nó atual
2. Percorrer a sub árvore esquerda em pré ordem.
3. Percorrer a sub árvore direita em pré ordem.

Utilizado para criar uma cópia da árvore, expressar a árvore em notação de parênteses, entre outros.

Percurso em Em Ordem

No percurso em em ordem, os nós são processados entre as visitas às subárvores esquerda e direita.

Sequência de Passos:

1. Percorrer a subárvore esquerda em inordem.
2. Visitar o nó atual.
3. Percorrer a subárvore direita em inordem.

Particularmente útil em Árvores Binárias de Busca (BSTs), onde o percurso em inordem resulta nos valores em ordem ascendente.

Percurso em Pós-ordem

Em pós-ordem, um nó é processado após as visitas às suas subárvores esquerda e direita.

Sequência de Passos:

1. Percorrer a subárvore esquerda em pós-ordem.
2. Percorrer a subárvore direita em pós-ordem.
3. Visitar o nó atual.

Útil para deletar a árvore, calcular o espaço ocupado por uma árvore em sistemas de arquivos, entre outros.

python Copy code

```python
class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key

# Funções para cada tipo de percurso
def print_preorder(root):
    if root:
        print(root.val, end=' ')
        print_preorder(root.left)
        print_preorder(root.right)

def print_inorder(root):
    if root:
        print_inorder(root.left)
        print(root.val, end=' ')
        print_inorder(root.right)

def print_postorder(root):
    if root:
        print_postorder(root.left)
        print_postorder(root.right)
        print(root.val, end=' ')
```

Estes percursos são ferramentas essenciais para interagir com árvores binárias, cada um adequado a diferentes necessidades e aplicações. Compreender esses métodos é fundamental para trabalhar eficazmente com árvores binárias em diversas situações práticas de programação.

Implementação de uma Árvore Binária de Busca (BST)

Uma Árvore Binária de Busca (BST) é uma estrutura de dados de árvore em que cada nó possui uma chave (ou valor) e dois possíveis subnós, seguindo regras específicas: todos os nós na subárvore esquerda de um nó têm chaves menores que a do nó, e todos os nós na subárvore direita têm chaves maiores.

Estrutura Básica de um Nó

python Copy code

```python
class Node:
    def __init__(self, key):
        self.left = None
        self.right = None
        self.val = key
```

Cada nó em uma BST contém um valor e ponteiros para seus subnós esquerdo e direito.

Inserção em uma BST

python Copy code

```python
def insert(root, key):
    if root is None:
        return Node(key)
    else:
        if root.val < key:
            root.right = insert(root.right, key)
        else:
            root.left = insert(root.left, key)
    return root
```

A inserção começa comparando o valor a ser inserido com o valor do nó. Se for maior, move-se para a subárvore direita; se menor, para a esquerda.

Busca em uma BST

python Copy code

```python
def search(root, key):
    if root is None or root.val == key:
        return root
    if root.val < key:
        return search(root.right, key)
    return search(root.left, key)
```

A busca segue um processo semelhante à inserção, movendo-se para esquerda ou direita dependendo do valor buscado.

```python
def delete(root, key):
    if root is None:
        return root

    if key < root.val:
        root.left = delete(root.left, key)
    elif(key > root.val):
        root.right = delete(root.right, key)
    else:
        # Nó com apenas um filho ou sem filho
        if root.left is None:
            return root.right
        elif root.right is None:
            return root.left

        # Nó com dois filhos: obter o sucessor (menor na subárvore direita)
        root.val = minValue(root.right)

        # Deletar o sucessor
        root.right = delete(root.right, root.val)

    return root

def minValue(node):
    current = node
    while current.left is not None:
        current = current.left
    return current.val

```

A deleção é mais complexa, envolvendo três casos: deletar um nó sem filhos, com um filho, ou com dois filhos.

python Copy code

```python
# Criando uma nova BST e inserindo valores
root = None
root = insert(root, 50)
insert(root, 30)
insert(root, 20)
insert(root, 40)
insert(root, 70)
insert(root, 60)
insert(root, 80)

# Buscando um valor
print("60 Encontrado" if search(root, 60) else "60 Não encor

# Deletando um valor
root = delete(root, 20)

# Verificando a árvore após a deleção
print("20 Encontrado" if search(root, 20) else "20 Não encor
```

A implementação de uma BST em Python é um excelente exercício para entender estruturas de dados hierárquicas e algoritmos de busca e manipulação. Compreender e aplicar corretamente essas operações é fundamental para resolver muitos problemas computacionais complexos de forma eficiente.