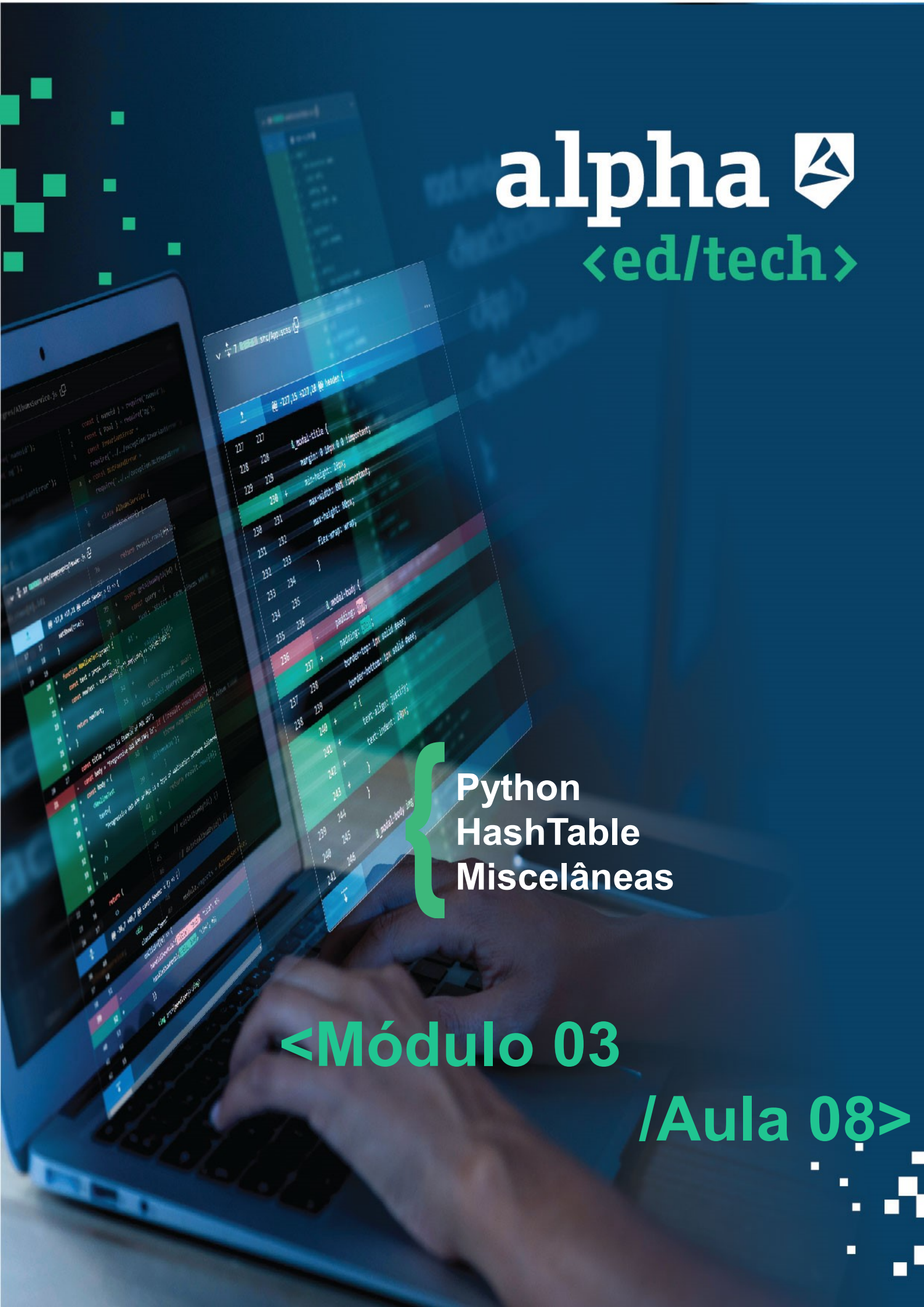

alpha
<ed/tech>
Python
HashTable
Miscelâneas
<Módulo 03
/Aula 08>

HashTable TDD & Miscelâneas

Estruturas de Dados

Estrutura de dados é uma maneira organizada de armazenar e manipular dados para que possamos realizar operações eficientes. É como a "caixa" ou "formato" que usamos para guardar informações, permitindo-nos acessá-las, modificá-las e realizar diferentes operações.

Imagine que você está resolvendo um quebra-cabeça. Se todas as peças estão bagunçadas aleatoriamente, levará mais tempo e esforço para encontrar a peça correta. No entanto, se as peças estão organizadas, com bordas e cores similares juntas, torna-se mais fácil resolver o quebra-cabeça. Da mesma forma, as estruturas de dados ajudam a organizar informações de maneira eficiente para facilitar a manipulação.

Tipos Comuns de Estruturas de Dados:

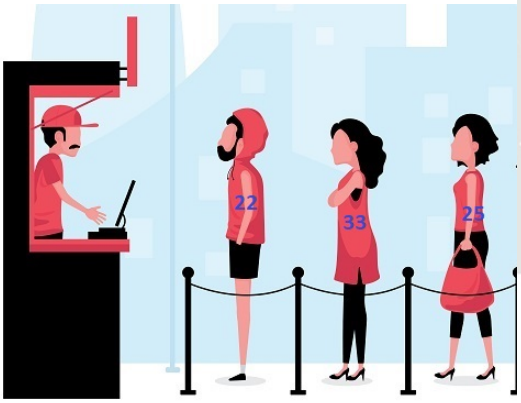
1. **Arrays**:
   - **O que são**: Uma coleção ordenada de elementos, onde cada elemento tem um índice.
   - **Para que servem**: Acesso rápido a elementos, especialmente se soubermos o índice.
   - **Analogia**: Uma prateleira de livros numerados
2. **Listas Ligadas**:
   - O que são: Uma sequência de elementos onde cada elemento possui uma referência ao próximo elemento na sequência.
   - Para que servem: Inserção e exclusão eficientes de elementos.
   - Analogia: Uma corrente onde cada elo aponta para o próximo.
3. **Pilhas**:
   - **O que são**: Uma coleção de elementos com operações de inserção (push) e remoção (pop) realizadas no topo.
   - **Para que servem**: Útil para operações LIFO (Last In, First Out), como desfazer ações em um software.
   - Analogia:** Pilha de pratos onde sempre adicionamos ou removemos do topo.
4. **Filas**:
   - **O que são**: Semelhantes a pilhas, mas operam em um princípio **FIFO (First In, First Out)**.
   - **Para que servem**: Útil para operações onde o primeiro elemento a entrar é o primeiro a sair.
   - **Analogia**: Uma fila no supermercado.
5. **Árvores**:
   - **O que são**: Estrutura hierárquica de elementos, com um elemento chamado raiz e outros agrupados em níveis.
   - **Para que servem**: Organizar dados hierarquicamente, como a estrutura de pastas em um computador.
   - **Analogia**: Uma árvore genealógica.

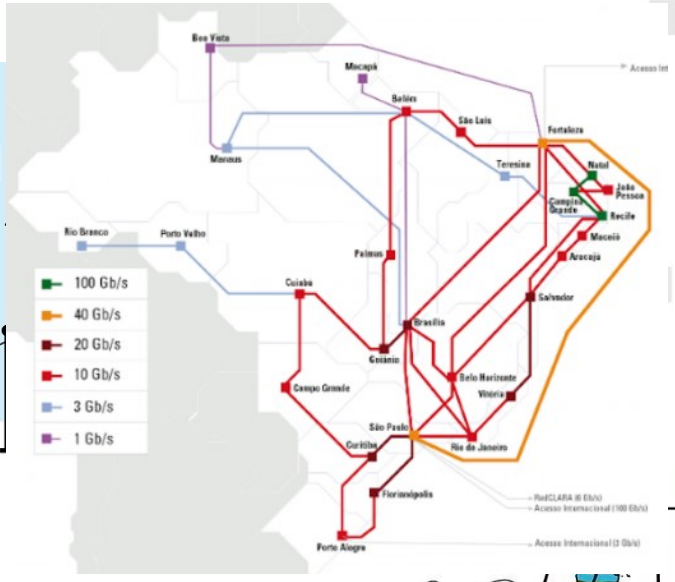
HashTable TDD & Miscelâneas

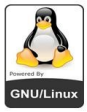
Estruturas de Dados

1. **Grafos**:
   - **O que são**: Conjunto de vértices (nós) conectados por arestas.
   - **Para que servem**: Modelar relações complexas entre diferentes elementos.
   - **Analogia**: Uma rede de estradas interconectadas.

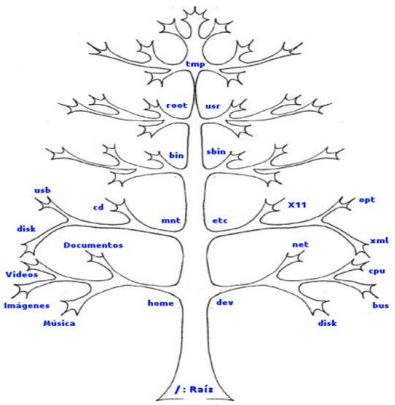
Entender estruturas de dados é fundamental para projetar algoritmos eficientes e resolver problemas de forma otimizada. Escolher a estrutura de dados certa para um determinado problema pode fazer uma grande diferença na eficiência do seu código.

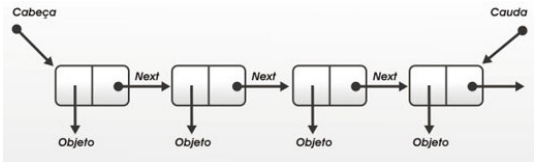


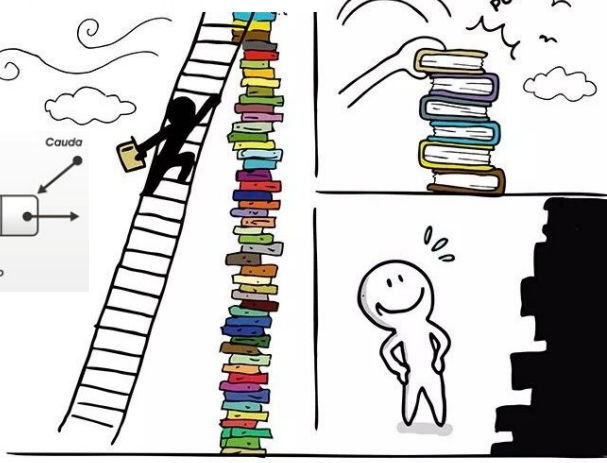


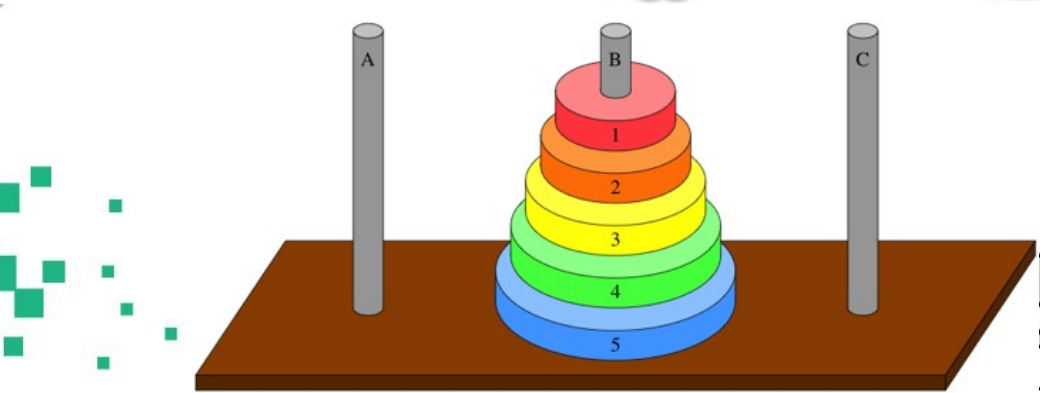


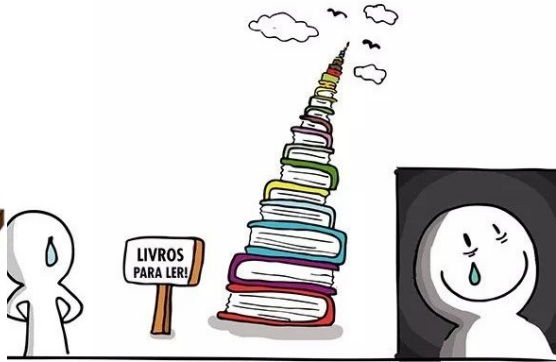
Arvore de Diretório











HashTable TDD & Miscelâneas

Engenharia de Software

Engenharia de Software é uma disciplina da ciência da computação que se dedica à concepção, construção, manutenção e evolução de sistemas de software de forma sistemática, controlada e eficiente. É a aplicação de princípios de engenharia para o desenvolvimento de software de alta qualidade.

Princípios Fundamentais da Engenharia de Software:

1. **Arrays**:
   - **O que é**: A Engenharia de Software segue um processo bem definido, que consiste em fases como análise, projeto, implementação, testes e manutenção.
   - **Analogia**: Assim como construir uma casa, prédio ou uma pirâmide segue uma sequência de etapas, o desenvolvimento de software também tem um processo.
2. **Controle e Organização:**
   - **O que é**: Envolve a aplicação de técnicas e métodos para organizar e controlar o desenvolvimento de software, garantindo eficiência e qualidade.
   - **Analogia**: Semelhante à gestão de um projeto de construção, onde há planejamento, monitoramento e controle.
3. **Qualidade e Eficiência:**
   - **O que é**: A busca pela qualidade no software, garantindo que ele atenda aos requisitos e seja eficiente em termos de desempenho e recursos.
   - **Analogia**: Como um fabricante de carros que se esforça para produzir veículos seguros, confiáveis e eficientes.

Atividades Principais da Engenharia de Software:

1. **Análise de Requisitos:**
   - **O que é**: Entender e documentar os requisitos do cliente para o software.
   - **Analogia**: Como um arquiteto que coleta informações sobre o que o cliente deseja em uma casa.
2. **Projeto de Software:**
   - **O que é**: Criar uma estrutura lógica para o software com base nos requisitos.
   - **Analogia**: Como um engenheiro que projeta os planos detalhados de uma construção.
3. **Implementação:**
   - **O que é**: Escrever o código-fonte do software com base no design.
   - **Analogia**: Como os engenheiros, projetistas e construtores que seguem o planejamento definido anteriormente para construir a casa.
4. **Testes:**
   - **O que é**: Verificar se o software funciona conforme esperado e identificar possíveis erros.
   - **Analogia**: Semelhante a inspecionar uma casa para garantir que tudo funcione corretamente. Exemplo, ao testar a entrada do carro na garagem foi esquecido que o motorista precisa descer do carro após estacionar.

HashTable TDD & Miscelâneas

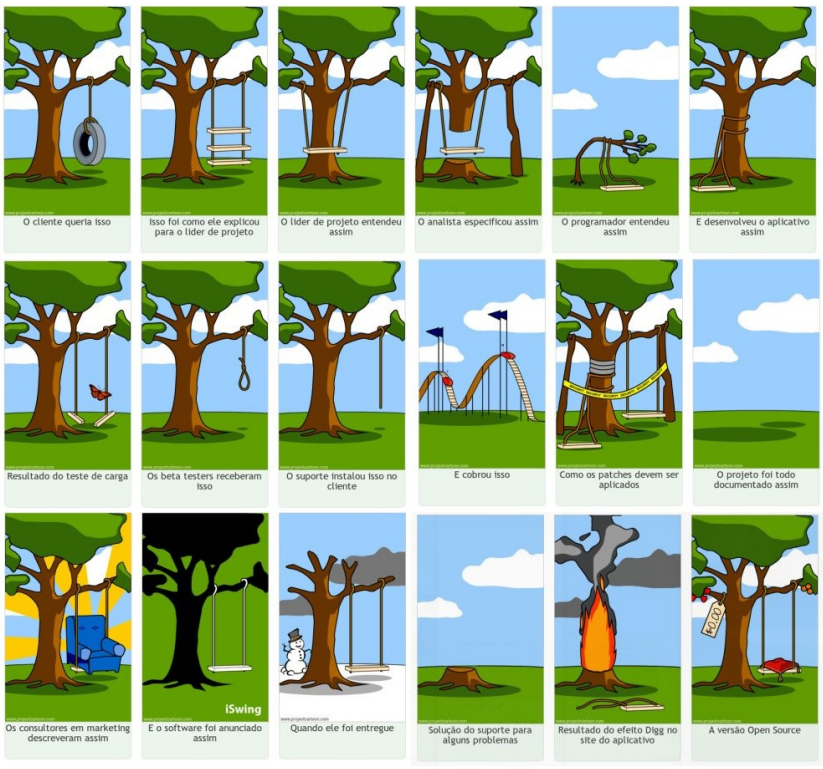
Engenharia de Software

5. **Manutenção**:
    - **O que é**: Realizar alterações no software para corrigir erros, adicionar novos recursos ou melhorar o desempenho.
    - **Analogia**: Como fazer correções antes da entrega da casa para atender às necessidades não previstas no projeto da casa. Exemplo, mesmo que a garagem estava correta para um carro de passageiro, o proprietário esqueceu de avisar que ele tinha um caminhão utilitário baú.
6. **Evolução:**
    - **O que é**: Realizar alterações substanciais no software mas não para corrigir erros, agora para adicionar novas funcionalidades, recursos ou melhorar o desempenho.
    - **Analogia**: Como fazer reformas e ampliações em uma casa para atender às novas necessidades. Exemplo, a casa funcional mas agora o pai está com uma idade avançada e precisa rampas para cadeiras de rodas, barras de apoio nos banheiros um novo quarto no andar térreo.

Benefícios da Engenharia de Software:

- **Eficiência**: Desenvolver software de forma mais rápida e com menos erros.
- **Qualidade**: Entregar produtos de software mais confiáveis e que atendam às expectativas.
- **Controle**: Gerenciar o desenvolvimento de software de forma organizada e controlada.

Engenharia de Software é a abordagem organizada e disciplinada para desenvolver software. Assim como em outras engenharias, ela visa criar soluções confiáveis e eficientes para atender às necessidades dos usuários. Essa disciplina é crucial para lidar com a crescente complexidade dos sistemas de software na atualidade.



HashTable TDD & Miscelâneas

TDD

TDD (**Test-Driven Development**) ou **Desenvolvimento Orientado a Testes**, é uma prática na engenharia de software em que você escreve testes automatizados antes de escrever o código de produção. O ciclo básico do TDD consiste em três etapas principais: escrever um teste, fazer o teste passar e, em seguida, refatorar (melhorar) o código escrito.

O Ciclo do TDD

1. **Escrever um Teste**:
   - **O que é**: Antes de começar a escrever o código real, você escreve um teste que descreve uma funcionalidade específica que deseja implementar.
   - **Analogia**: Se você estivesse construindo um carro, seria como escrever uma lista de verificações para garantir que cada parte do carro funcione corretamente.
2. **Fazer o Teste Passar**:
   - **O que é**: Escrever o código de produção mínimo necessário para fazer o teste passar. Atualmente a ideia do MVP (Mínimo Produto Viável, como implementar e aprimorar para gerar mais resultados antes de investir muito dinheiro) como usar esse conceito para validar uma ideia e crescer com o feedback do mercado.
   - **Analogia**: No contexto do carro, isso seria realmente construir a parte do carro correspondente ao que você listou na sua lista de verificações.
3. **Refatorar o Código**:
   - **O que é**: Melhorar o código sem alterar seu comportamento, mantendo os testes passando.
   - **Analogia**: Ajustar ou melhorar as peças do carro para torná-las mais eficientes, mantendo as funcionalidades.

Por que usar TDD?

1. **Feedback Rápido**: TDD fornece feedback rápido sobre a qualidade do seu código. Se um teste falhar, você saberá imediatamente que algo precisa ser corrigido.
2. **Código Mais Robusto**: O código resultante do TDD geralmente é mais robusto, pois é validado por testes automatizados.
3. **Facilita a Manutenção**: Testes automatizados tornam a manutenção do código mais fácil. Se você precisa fazer alterações no futuro, os testes ajudam a garantir que as alterações não quebram a funcionalidade existente.
4. **Design Melhorado**: O TDD muitas vezes leva a um design de código melhor, pois você precisa pensar nas interfaces e na estrutura do código antes de implementar.
5. **Confiança no Código**: Ter um conjunto abrangente de testes automatizados dá confiança de que o código está funcionando conforme o esperado.

HashTable TDD & Miscelâneas

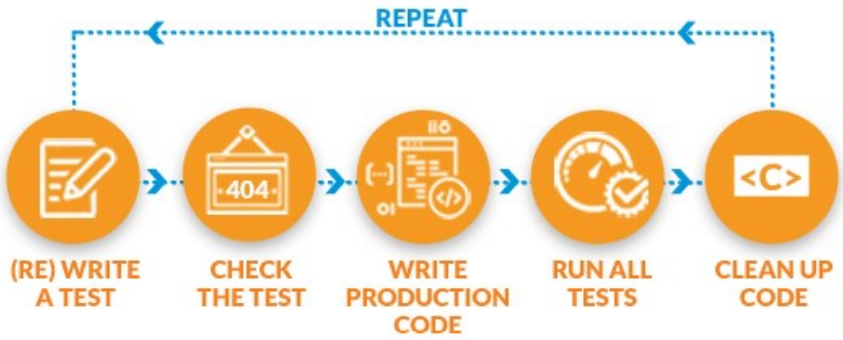
TDD – Exemplo Prático

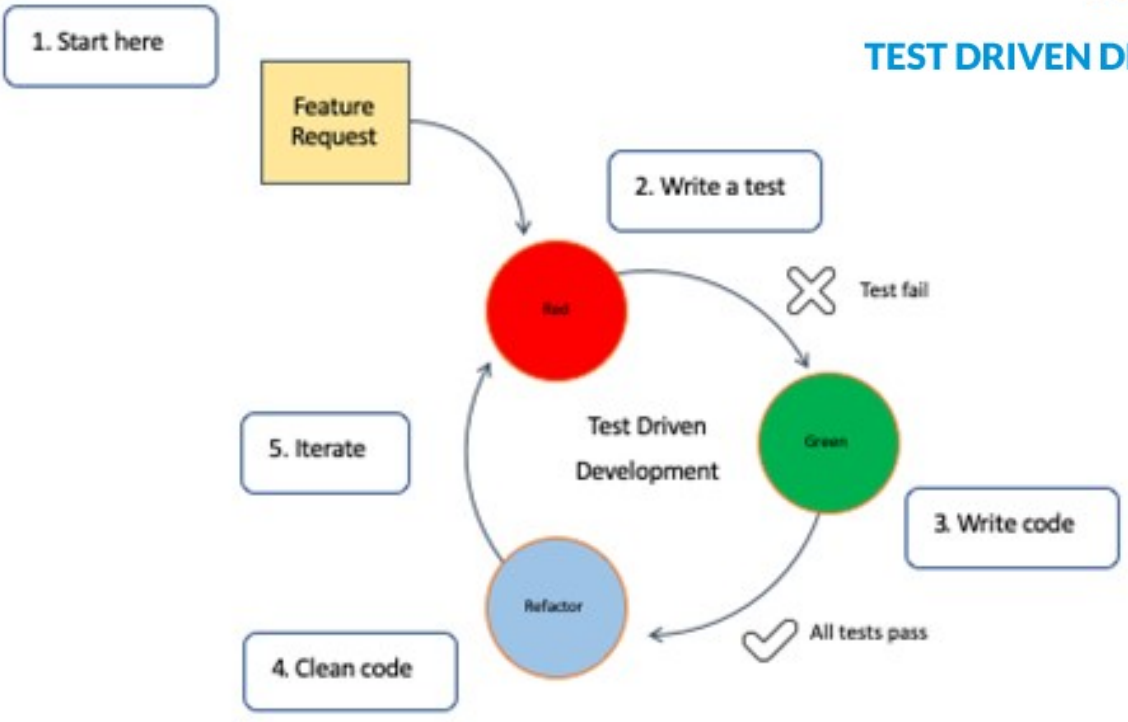
Vamos imaginar que você está desenvolvendo uma função simples que adiciona dois números. No TDD, o ciclo seria assim.

O Ciclo do TDD

1. **Escrever um Teste**: Escrever um teste que chama a função `adicionar(2, 3)` e verifica se o resultado é 5.
2. **Fazer o Teste Passar**: Implementar a função `adicionar` para retornar a soma de dois números.
3. **Refatorar o Código**: Se necessário, ajustar o código para torná-lo mais limpo ou eficiente, mantendo os testes passando.

O TDD é uma abordagem que visa criar software de alta qualidade, confiável e fácil de manter. Ele promove a escrita de código testável desde o início do desenvolvimento, garantindo que cada parte do sistema seja verificada automaticamente quanto à conformidade com os requisitos. Isso resulta em um processo de desenvolvimento mais seguro e eficiente.





HashTable TDD

HashTable ajuda a resolver muitos problemas da vida real, como indexar tabelas de banco de dados, armazenar valores calculados em cache ou implementar conjuntos.
Frequentemente, ela é abordada em entrevistas de emprego, e o Python utiliza tabelas hash em muitos lugares para tornar as consultas de nomes quase instantâneas.
Python tem sua própria tabela hash chamada "dict" (dicionário), necessário entender como as tabelas hash funcionam nos bastidores. Uma avaliação de codificação pode até desafiá-lo a construir uma.
Estrutura de Dados propiciará alguns desafios que introduzirão conceitos importantes e lhe darão uma ideia de por que as tabelas hash são tão rápidas.

Cuidado para a terminologia não ficar um pouco confusa. Coloquialmente, o termo "tabela hash" ou "hash map" é frequentemente usado de forma intercambiável com a palavra "dicionário". No entanto, há uma diferença sutil entre os dois conceitos, pois o primeiro é mais específico que o último esperado.

Tabela Hash vs. Dicionário: Na ciência da computação, um dicionário é um tipo de dado abstrato composto por chaves e valores organizados em pares.
Além disso, ele define as seguintes operações para esses elementos:
•Adicionar um par chave-valor
•Excluir um par chave-valor
•Atualizar um par chave-valor
•Encontrar um valor associado à chave fornecida

De certa forma, esse tipo de dado abstrato se assemelha a um dicionário bilíngue, onde as **chaves são palavras estrangeiras** e os **valores são suas definições ou traduções para outros idiomas**.

No entanto, nem sempre é necessário haver um sentido de equivalência entre chaves e valores.

Um catálogo telefônico é outro exemplo de um dicionário, que combina nomes com os respectivos números de telefone.

Sempre que você **mapeia** uma coisa para outra ou **associa** um valor a uma chave, você está essencialmente usando uma espécie de dicionário. É por isso que os dicionários também são conhecidos como mapas ou arrays associativos.

Dicionários têm algumas propriedades interessantes. Uma delas é que você pode pensar em um dicionário como uma função matemática que projeta um ou mais argumentos para exatamente um valor.

HashTable TDD

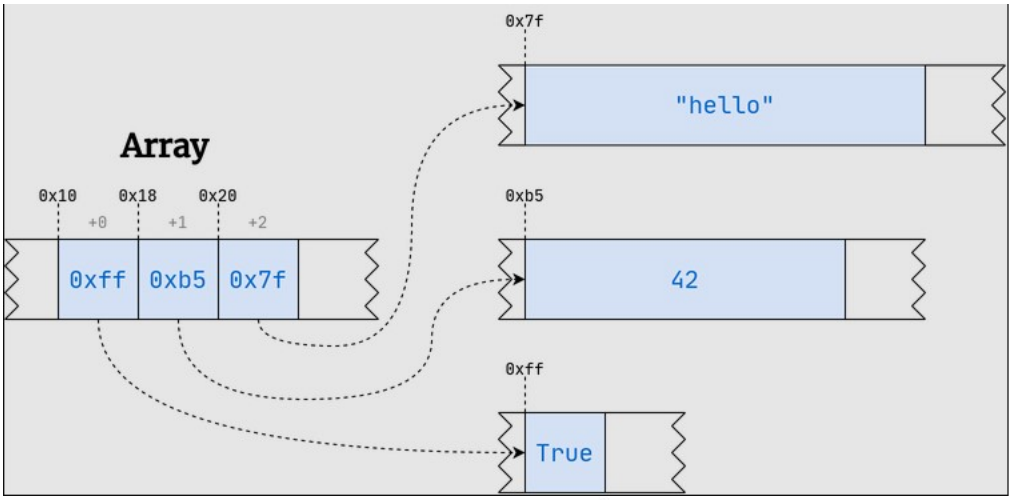
As consequências diretas desse fato são as seguintes:
•**Apenas Pares Chave-Valor**: Você não pode ter uma chave sem o valor ou vice-versa em um dicionário. Eles sempre vão juntos.
•**Chaves e Valores Arbitrários**: Chaves e valores podem pertencer a dois conjuntos disjuntos do mesmo tipo ou tipos diferentes. Tanto chaves quanto valores podem ser quase qualquer coisa, como números, palavras ou até mesmo imagens.
•**Pares Chave-Valor Não Ordenados**: Devido ao último ponto, os dicionários geralmente não definem nenhuma ordem para seus pares chave-valor. No entanto, isso pode depender da implementação específica.
•**Chaves Únicas**: Um dicionário não pode conter chaves duplicadas, pois isso violaria a definição de uma função.
•**Valores Não Únicos**: O mesmo valor pode ser associado a muitas chaves.

Existem conceitos relacionados que **estendem a ideia de um dicionário**. Por exemplo, um **multimapa** permite que você tenha **mais de um valor por chave**, enquanto um mapa bidirecional não apenas mapeia chaves para valores, mas também fornece mapeamento na direção oposta.

O **dict do Python** permite que você realize todas as operações de dicionário, com a sintaxe de colchetes (`[]`) usada em Listas do Python:
•Pode adicionar um novo par chave-valor a um dicionário.
•Pode atualizar o valor ou excluir um par existente identificado por uma chave.
•Pode procurar o valor associado à chave fornecida.
•Pode fazer uma pergunta diferente.

Como o dicionário built-in no Python realmente funciona?
Como ele mapeia chaves de tipos de dados arbitrários e como o faz tão rapidamente?
Encontrar uma implementação eficiente desse tipo de dado abstrato é conhecido como o **problema do dicionário**.
A solução do **dict do Python** aproveita a estrutura de dados da tabela hash.

No entanto, observe que essa não é a única maneira de implementar um dicionário em geral. Outra implementação popular é baseada em uma árvore rubro-negra.



HashTable TDD

Tabela Hash: Um Array Com uma Função de Hash

Acessar elementos de sequências em Python é tão rápido, independentemente do índice solicitado.

```
import string
texto = string.ascii_uppercase * 100000000

print(len(texto))

print(texto[300000000:300000050])
```

Há 2,6 bilhões de caracteres provenientes de letras ASCII repetidas na variável texto acima, que você pode contar com a função len() do Python. No entanto, obter o primeiro, do meio, último ou qualquer outro caractere desta string é igualmente rápido.

O mesmo é válido para **todos os tipos de sequências em Python**, como **listas** e **tuplas**.

O segredo para essa velocidade impressionante é que sequências em Python são respaldadas por **um array (estrutura de dados de acesso aleatório)**. Arrays segue dois princípios:

1. O array ocupa um bloco contínuo de memória.
2. Cada elemento no array tem um tamanho fixo conhecido antecipadamente.

Obs.: **Acesso Aleatório é um péssimo nome**, pois foi uma expressão criada da fita de acesso sequencial diferente do disco rígido, que **na verdade é um disco de acesso direto**. Como também, RAM Memória de Acesso Randômico, deveria ser acesso direto.

Quando você **conhece o endereço de memória de um array, chamado de offset**, então pode chegar ao elemento desejado no array instantaneamente, simplesmente sabendo o tamanho de cada elemento do array (como int ou float ou complex).

Observação: Ao contrário dos arrays, as listas do Python podem conter elementos heterogêneos de tamanhos variados. Para mitigar isso, o Python adiciona outro nível de indireção introduzindo um array de ponteiros para locais de memória, em vez de armazenar valores diretamente no array.

Ponteiros são apenas números inteiros, que sempre ocupam a mesma quantidade de espaço. É costume **denotar endereços de memória usando a notação hexadecimal. O Python e algumas outras linguagens prefixam tais números com 0x.**

Encontrar um elemento em um array é rápido, não importa onde esse elemento esteja fisicamente localizado.
Usar a mesma estratégia em um dicionário. Um array de endereços de memória, para cada elemento do dicionário.