# INSTITUTO INFNET

## Faculdade de Engenharia e Desenvolvimento de Software


# Velocidade e Qualidade com Estruturas de Dados e Algoritmos

## AT

**Aluno: JEAN MICHAEL ESTEVEZ ALVAREZ**

**E-mail: jean.alvarez@al.infnet.edu.br**

**Matrícula:**

**Professor: Flávio da Silva Neves**

**Distrito Federal**

**Dezembro, 2024**

**SUMÁRIO**

# 1 EXERCÍCIOS

## 1.1 QUESTÃO 1: VANTAGENS DA ÁRVORE DE BUSCA BINARIA

Organizar dados em uma árvore de busca binária (ABB) melhora significativamente a eficiência em operações de busca e inserção em comparação a uma lista desordenada, especialmente para grandes volumes de dados, no exemplo citado de 1 milhão de produtos em um e-commerce. Uma ABB balanceada, a busca e a inserção têm complexidade (O(log n)), enquanto em uma lista desordenada essas operações são (O(n)), devido à necessidade de percorrer todos os elementos.

A estrutura hierárquica da ABB permite eliminar metade dos elementos em cada comparação, otimizando o acesso. Além disso, árvores balanceadas, como AVL ou rubro-negras que foram apresentadas em sala de aula, garantem eficiência consistente ao evitar desbalanceamento. A ABB também facilita buscas condicionais e intervalares, permitindo ignorar subárvores inteiras fora do escopo da consulta.

Assim, ao oferecer eficiência, escalabilidade e flexibilidade, a ABB é uma escolha superior para gerenciar grandes conjuntos de dados em sistemas de e-commerce.

Para demostrar a eficiência desta solução, implementei 2 programas o primeiro gera os registros “fake” de um e-commerce e o segundo é a ABB para realizar as buscas destes registros, os programas estão disponibilizados no github o link esta no capitulo 2.

## 1.2 FILA OU PILHA?

Na gestão de pacotes para entregas, a escolha entre usar uma fila ou uma pilha depende diretamente da ordem desejada para as operações de inserção e remoção, dado que

essas estruturas possuem comportamentos distintos. Uma fila segue o princípio FIFO (First In, First Out), onde o primeiro pacote inserido é o primeiro a ser removido. Já a pilha opera segundo o princípio LIFO (Last In, First Out), permitindo que o último pacote inserido seja o primeiro a sair.

Em um cenário logístico típico, onde os pacotes precisam ser entregues na mesma ordem em que foram registrados, a fila é a estrutura mais adequada. Isso ocorre porque ela mantém a ordem natural das operações: o primeiro pacote inserido no sistema será o primeiro a ser processado ou entregue, refletindo uma sequência cronológica lógica e eficiente.

Por outro lado, uma pilha seria apropriada em situações específicas onde os pacotes precisam ser manipulados de forma reversa, como em cenários de empilhamento físico em que o último item adicionado deve ser o primeiro acessado, mas isso não é comum na gestão de entregas regulares.

Portanto, considerando que a ordem de entrega geralmente respeita a sequência de registro, uma fila oferece o comportamento mais intuitivo e eficiente, alinhado às exigências operacionais da empresa.

## 1.3 BUSCA LINEAR SIMPLES

Para implementar o programa sugerido, separei o problema em dois programas. O primeiro programa gera os dados fictícios dos contatos, armazenando-os em um arquivo no formato "nome,telefone". O segundo programa carrega esses dados e realiza a busca linear para localizar o número de telefone associado a um nome específico.

O programa responsável pela geração utiliza a biblioteca Faker para criar nomes e números de telefone aleatórios. A função gerar_registros_aleatorios recebe o caminho do arquivo e a quantidade de registros desejada, garantindo a existência do diretório de destino e salvando os contatos no arquivo registrosContatos.txt. Cada linha contém um par nome-telefone, separados por vírgula. Essa etapa assegura a criação de dados consistentes e em formato padronizado, simulando um cenário realista para o aplicativo.

O programa de busca, por sua vez, utiliza a função carregar_contatos_de_arquivo para ler os dados gerados no arquivo e organizar cada linha como um dicionário contendo as chaves "nome" e "telefone". A estrutura de lista de dicionários permite um acesso

programático mais claro e eficiente durante a busca. O algoritmo de busca linear, implementado na função buscar_contato_por_nome, percorre cada registro da lista, comparando o nome armazenado com o nome procurado. Caso uma correspondência seja encontrada, o número de telefone é imediatamente retornado; caso contrário, a função retorna None.

No exemplo, o sistema simula a busca por um nome existente na posição 10 da lista carregada, e o número de telefone correspondente é exibido. Essa abordagem é eficaz para listas pequenas ou moderadas e demonstra a integração prática entre geração, armazenamento e busca de dados. A modularidade dos programas reflete boas práticas de engenharia de software, separando responsabilidades e garantindo um fluxo claro e adaptável para os requisitos do aplicativo de mensagens.

## 1.4 BUSCA BINARIA VS BUSCA LINEAR

Para resolução deste exercício implementamos 2 programas, o primeiro gerou os registros e no segundo foi implementado a busca binaria e a busca linear e feita uma simples comparação de tempo entre as duas.

A busca binária é claramente superior em cenários onde a lista está previamente ordenada, como neste caso. Seu desempenho exponencialmente mais rápido se destaca especialmente em conjuntos de dados grandes, como os 100 mil livros usados aqui. Por outro lado, a busca linear, embora simples e adequada para listas pequenas ou não ordenadas, torna-se significativamente menos eficiente à medida que o tamanho da lista aumenta. Portanto, para uma biblioteca digital com listas ordenadas de livros, a busca binária é a abordagem ideal, equilibrando eficiência e escalabilidade.

## 1.5 O(N) E O(2N) EQUIVALÊNCIA NA DERIVADA

Em notação Big O, tanto um algoritmo que percorre uma lista uma vez O(N) quanto outro que a percorre duas vezes O(2N) são classificados como O(N), porque a notação Big O

considera apenas o comportamento assintótico, ou seja, como o algoritmo escala à medida que o tamanho da entrada (N) aumenta. Na análise assintótica, constantes e fatores menores são descartados, pois não afetam o crescimento da complexidade em relação ao tamanho da entrada. Assim, o termo 2N é simplificado para N, já que a constante multiplicativa 2 tem impacto desprezível quando N cresce muito.

Na prática, entretanto, a diferença entre percorrer a lista uma vez ou duas vezes pode ser relevante para entradas de tamanho pequeno ou médio, pois o algoritmo O(2N) realizará efetivamente o dobro de operações. Essa diferença, no entanto, torna-se proporcionalmente menos significativa conforme o tamanho da entrada aumenta, dado que o número de operações de ambos os algoritmos cresce linearmente. Portanto, embora a notação Big O seja útil para comparar o comportamento em larga escala, ela não captura as nuances de desempenho em cenários onde fatores constantes podem ser relevantes, especialmente em sistemas sensíveis à latência ou que processam dados de tamanho moderado.

Dessa forma, enquanto ambos os algoritmos são equivalentes em termos de Big O, a escolha do mais eficiente em cenários práticos deve considerar outros fatores, como otimização constante, limites de hardware e impacto acumulativo, além da análise teórica.

## 1.6 BUBBLE SORT

O impacto no desempenho do Bubble Sort com o aumento da quantidade de dados é diretamente relacionado à sua complexidade computacional. O algoritmo possui uma complexidade de O(N^2), o que significa que o número de comparações e trocas cresce quadraticamente com o tamanho da lista.

Para uma entrada de 1.000 elementos, o algoritmo realizou aproximadamente $1000^2$ operações, resultando em um tempo de execução de aproximadamente 0,14 segundos. Já para 10.000 elementos, o número de operações aumenta para $10000^2$, ampliando o tempo de execução para 20 segundos, uma diferença que reflete o crescimento exponencial das operações.

Na prática, isso ocorre porque o Bubble Sort percorre repetidamente a lista, comparando pares adjacentes e realizando trocas sempre que necessário. À medida que o tamanho da lista aumenta, o número de iterações e comparações cresce de forma

desproporcional. Esse comportamento torna o algoritmo inadequado para grandes conjuntos de dados, onde algoritmos mais eficientes, como Merge Sort ou Quick Sort O(N.log(N)), são preferíveis.

O resultado evidencia como o aumento no volume de dados amplifica drasticamente o tempo de execução de algoritmos com complexidade quadrática, ressaltando a importância de considerar a escalabilidade ao escolher algoritmos para problemas computacionais em larga escala.

## 1.7 OTIMIZAÇÃO DE COMPLEXIDADE

No código original, com complexidade $O(N^2)$, provavelmente utilizava comparações aninhadas para verificar duplicatas, onde cada elemento era comparado com todos os outros, resultando em desempenho impraticável para listas maiores. No código otimizado, a escolha de uma Hashtable (implementada como um set em Python) reduz o tempo de execução ao custo O(1) por operação de inserção e consulta, graças ao uso de hashing.

Durante a execução, cada elemento da lista é verificado quanto à presença no conjunto de elementos já vistos. Se o elemento já estiver no conjunto, há uma duplicata e o algoritmo retorna True imediatamente, encerrando a execução. Caso contrário, o elemento é adicionado ao conjunto, e o algoritmo continua. Essa abordagem garante que cada elemento é processado uma única vez, resultando em complexidade O(N), onde N é o tamanho da lista.

Essa otimização é justificada pelo comportamento linear esperado das operações de inserção e consulta em hashtables. Além disso, a estrutura do algoritmo permanece clara, legível e fácil de manter, alinhando-se aos princípios de Clean Code e eficiência computacional. Em listas grandes, a redução da complexidade de $O(N^2)$ para O(N) é essencial para garantir a escalabilidade do código.

## 1.8 SELECT SORT ORGANIZANDO JOGADORES

Primeiro passo foi criar um gerador de registrosA função gerar_registros_jogadores cria registros fictícios de jogadores, atribuindo nomes sequenciais e pontuações aleatórias entre 0 e 10.000. Esses registros são salvos no arquivo registros_jogadores.txt no formato "nome,pontuacao". Essa etapa simula um conjunto inicial de dados a ser organizado.

Carregamento dos registros: A função carregar_registros lê os dados do arquivo, separando o nome e a pontuação de cada jogador. Os dados são convertidos para uma lista de dicionários, permitindo manipulação mais intuitiva durante a ordenação.

Ordenação com Selection Sort: O algoritmo Selection Sort percorre a lista de jogadores iterativamente. Em cada iteração:

1. Encontra o índice do jogador com a menor pontuação no subarray restante.
2. Troca o jogador da posição atual pelo jogador com a menor pontuação.
3. Repete até que a lista esteja ordenada.

Esse processo garante que a lista seja ordenada em ordem crescente de pontuação, com complexidade $O(N^2)$.

Realizando uma análise geral da complexidade, podemos observar que tanto no caso da lista já está ordenada lidaremos com $O(N^2)$ pois ele percorrera elemento a elemento verificando o maior, sendo assim podemos sintetizar os seguintes tópicos principais

- Melhor caso: $O(N^2)$– Mesmo que a lista esteja ordenada, o algoritmo sempre percorre a lista para encontrar o menor elemento.
- Pior caso: $O(N^2)$ – Quando a lista está em ordem inversa, o algoritmo realiza o mesmo número de comparações e trocas.
- Espaço: O(1) – O algoritmo é in-place, ou seja, não utiliza memória adicional significativa.

## 1.9 HASH TABLE EM REDE SOCIAL

A abordagem baseada em hashtable para recuperar perfis em uma rede social é significativamente mais eficiente do que percorrer uma lista sequencialmente devido à sua complexidade computacional. Em uma lista sequencial, a busca por um elemento exige a verificação de cada registro, resultando em uma complexidade de tempo O(n), onde N é o número total de registros. No pior caso, todos os elementos da lista precisam ser examinados

antes de localizar o desejado ou concluir que ele não existe, o que torna essa abordagem ineficiente em cenários de grande escala.

Por outro lado, a hashtable utiliza uma função de hash para mapear diretamente uma chave, como o ID ou o nome do usuário, a um índice na estrutura subjacente, permitindo uma recuperação de tempo constante O(1) em média. Essa eficiência é alcançada porque a busca não depende do tamanho da tabela, mas sim da qualidade da função de hash e da dispersão uniforme das chaves. Mesmo em casos de colisões, onde múltiplas chaves são mapeadas para o mesmo índice, o uso de listas encadeadas para tratar colisões mantém o impacto controlado, preservando uma complexidade média de O(1).

Na prática, essa diferença de complexidade é crítica para redes sociais, onde os dados frequentemente envolvem milhões de usuários. Enquanto a abordagem sequencial se degrada linearmente com o aumento do número de registros, a hashtable mantém a recuperação eficiente, tornando-a a escolha ideal para sistemas que exigem operações de busca rápidas e frequentes. Essa otimização contribui para uma experiência de usuário fluida, reduzindo a latência e melhorando a escalabilidade do sistema, no entanto a busca por nome, principalmente pelo fato de houver nomes iguais são problemáticos e devem ser bem estruturados para ter um bom funcionando, como exemplo implementei ambos os casos a busca por nome a busca por id.

## 1.10 PILHA DE NAVEGAÇÃO

A pilha implementada simula a funcionalidade de navegação de um navegador, utilizando duas estruturas de dados baseadas em pilhas: uma para o histórico de páginas visitadas e outra para armazenar as páginas futuras. Ao visitar uma nova página, ela é adicionada ao topo da pilha de histórico, enquanto a pilha de páginas futuras é limpa para refletir o fluxo linear da navegação.

A operação de "voltar" remove a página atual do topo da pilha de histórico, move-a para o topo da pilha de páginas futuras e retorna a página anterior no histórico. A operação de "avançar" faz o inverso, movendo uma página do topo da pilha de páginas futuras de volta para o histórico. Essa abordagem garante que o estado de navegação seja consistente e permite alternar eficientemente entre páginas previamente visitadas.

A complexidade de todas as operações “visitar, voltar e avançar” é O(1), uma vez que envolvem apenas inserções ou remoções no topo da pilha. Essa implementação reflete um design modular e eficiente, alinhado a boas práticas de engenharia de software, facilitando a escalabilidade e manutenção do sistema.

## 1.11 CHAMADOS EM FILA

A fila implementada para gerenciar chamados utiliza a estrutura de dados `deque`, que oferece inserções e remoções eficientes em ambas as extremidades, garantindo desempenho consistente. Os chamados são organizados na ordem de chegada, seguindo o princípio FIFO (First In, First Out), onde os elementos adicionados primeiro são os primeiros a serem removidos.

A inserção de novos chamados ocorre na extremidade posterior da fila, simulando o registro de novos pedidos no sistema, enquanto a remoção é realizada na extremidade anterior, representando o atendimento dos chamados mais antigos. Essa abordagem assegura uma gestão justa e previsível, respeitando a sequência temporal dos registros.

A operação de organização pela data de abertura utiliza uma ordenação eficiente para rearranjar os elementos em ordem cronológica crescente, garantindo que os chamados mais antigos sejam priorizados. A combinação dessas funcionalidades reflete um design simples e eficaz, alinhado às boas práticas de engenharia de software, otimizando a experiência do usuário e facilitando a escalabilidade do sistema em cenários de maior demanda.

## 1.12 EXPLORAR DIRETÓRIOS ANINHADOS

Para realizar a exploração de diretórios aninhados a recursão é uma técnica fundamental, pois com ela conseguimos percorrer estruturas hierárquicas, como diretórios de arquivos. Em cada chamada da função, o algoritmo verifica os itens do diretório atual. Se o item for um arquivo, ele é adicionado à lista de resultados. Se for um subdiretório, a função é chamada recursivamente com o subdiretório como argumento. Isso permite que o algoritmo

explore cada nível de aninhamento sem precisar conhecer previamente a profundidade da estrutura. A recursão segue até que não haja mais subdiretórios a serem processados, momento em que a função retorna os arquivos encontrados.

Essa abordagem é eficiente para navegar por estruturas hierárquicas, pois mantém o foco no processamento de um único nível por vez e delega a exploração de níveis mais profundos para as chamadas subsequentes. Além disso, a recursão elimina a necessidade de gerenciar manualmente pilhas ou filas para rastrear o estado da navegação, simplificando o código e tornando-o mais legível.

## 1.13 PROGRAMAÇÃO DINÂMICA NO PROBLEMA DA MOCHILA

A programação dinâmica melhora significativamente a eficiência na resolução do problema da mochila ao evitar cálculos redundantes, característica inerente à abordagem recursiva tradicional. Na solução recursiva, o mesmo subproblema é resolvido repetidamente para diferentes combinações de itens, resultando em uma complexidade exponencial O(2^N). Isso ocorre porque a recursão explora todas as possibilidades, recalculando subproblemas já analisados.

Com a programação dinâmica, o problema é dividido em subproblemas menores cujos resultados são armazenados em uma tabela bidimensional. Essa tabela permite reutilizar os resultados previamente calculados, eliminando a necessidade de resolver subproblemas múltiplas vezes. A abordagem utiliza uma matriz onde cada entrada dp[$i$][$w$] representa o valor máximo que pode ser obtido utilizando os primeiros $i$ itens com uma capacidade de $w$. A tabela é preenchida iterativamente, comparando o valor da inclusão ou exclusão de um item, garantindo que cada subproblema seja resolvido apenas uma vez.

Essa otimização reduz a complexidade do algoritmo para O(N.$w$), onde N é o número de itens e $W$ é a capacidade da mochila. Essa redução de complexidade é essencial em problemas de larga escala, onde a abordagem recursiva seria computacionalmente inviável. Além disso, a programação dinâmica oferece maior clareza e previsibilidade, permitindo que o problema seja resolvido de forma eficiente e escalável, sem sacrificar a exatidão do resultado.

## 1.14 ARVORE DE BUSCA BINARIA

A árvore binária de busca (BST) foi implementada e  após inserir os preços [100, 50, 150, 30, 70, 130, 170], a busca pelo preço 70 retornou True, indicando que ele está disponível na árvore.

```
75
76  # Implementação da BST
77  precos = [100, 50, 150, 30, 70, 130, 170]
78  bst = BST()
79
80  # Inserir preços na árvore
81  for preco in precos:
82      bst.inserir(preco)
83
84  # Verificar se o preço 70 está disponível
85  preco_disponivel = bst.buscar(70)
86  print(preco_disponivel)
87
```

```
PROBLEMS  OUTPUT  DEBUG CONSOLE  TERMINAL  PORTS

PS D:\AFaculdade\0_QuintoSemestre\AlgoSpeedQuality\AT> ^C
PS D:\AFaculdade\0_QuintoSemestre\AlgoSpeedQuality\AT> & d:/A
e/AlgoSpeedQuality/AT/AT_ex14_ABB.py
True
PS D:\AFaculdade\0_QuintoSemestre\AlgoSpeedQuality\AT>
```

## 1.15 GERENCIAMENTO DE NOTAS

A árvore binária de busca (BST) foi implementada para armazenar as notas finais da turma [85, 70, 95, 60, 75, 90, 100]. Após executar as funções para localizar os valores mínimo e máximo, os resultados são:

Nota mínima: 60

Nota máxima: 100

Essas funções percorrem a árvore começando pela raiz e movendo-se à esquerda para encontrar o valor mínimo ou à direita para localizar o máximo, garantindo eficiência O(H), onde H é a altura da árvore.

```
PS D:\AFaculdade\0_QuintoSemestre\AlgoSpeedQ
e/AlgoSpeedQuality/AT/AT_ex15_GerNotas.py
Nota Minima:  60  Nota Maxima:  100
```

## 1.16 SISTEMA DE INVENTÁRIO

Foi implementado uma árvore binária de busca para gerenciar os códigos dos produtos no inventário. Após realizar as remoções especificadas, os resultados das exibições em ordem crescente foram os seguintes:

1. Após remover 20 (nó folha): [25, 30, 45, 60, 65, 70]
2. Após remover 25 (nó com um filho): [30, 45, 60, 65, 70]
3. Após remover 45 (nó com dois filhos): [30, 60, 65, 70]

Explicação das operações:

- Remoção do nó folha (20): Simplesmente remove o nó, pois ele não possui filhos.
- Remoção do nó com um filho (25): O nó é substituído por seu único filho (30), preservando a propriedade da árvore binária de busca.
- Remoção do nó com dois filhos (45): O menor valor da subárvore à direita (60) substitui o nó removido, garantindo que a ordem da árvore seja mantida.

## 2 LINK GITHUB

https://github.com/EstevezCodando/AT_AlgSpeedQual