Unidade III

5 ALGORITMOS DE PESQUISA

Os algoritmos de pesquisa representam componentes essenciais na ciência da computação, sendo utilizados para localizar informações específicas dentro de conjuntos estruturados ou não estruturados. Em Python, essas técnicas assumem particular importância tanto para aplicações didáticas quanto profissionais. Dominar diferentes métodos de pesquisa permite que desenvolvedores otimizem sistemas para velocidade, eficiência e economia de recursos computacionais, influenciando diretamente na qualidade final das aplicações produzidas.

A forma mais elementar dessas técnicas é a pesquisa linear, cuja lógica reside em examinar sequencialmente cada elemento até localizar o item desejado ou concluir que ele não existe no conjunto. Em Python, sua implementação é trivial, frequentemente resumida a poucas linhas de código, como um laço que percorre todos os elementos até a condição de igualdade ser satisfeita. Apesar de sua simplicidade conceitual, a pesquisa linear revela-se pouco eficiente para conjuntos extensos ou situações nas quais consultas frequentes ocorrem, já que, no pior caso, demanda tempo proporcional ao número total de elementos (complexidade O(n)). Entretanto, sua facilidade de aplicação faz dessa abordagem uma solução válida para pequenos volumes de dados ou contextos em que nenhuma ordenação prévia tenha ocorrido, como listas aleatórias de valores, coleções dinâmicas ou ambientes com pouca complexidade estrutural.

Por outro lado, a pesquisa binária oferece uma alternativa significativamente mais eficiente, desde que aplicada a dados previamente ordenados. Nesse método, o conjunto é repetidamente dividido pela metade, comparando-se o elemento buscado ao elemento central da divisão atual. Se o valor central coincidir com o elemento buscado, a busca encerra-se com sucesso. Caso contrário, a metade correspondente ao valor buscado é selecionada e o processo repete-se até o elemento ser encontrado ou a sublista tornar-se vazia, indicando ausência. Implementações em Python aproveitam índices e cortes de lista (slices) para simplificar o código, mantendo legibilidade e eficiência. A vantagem crucial da pesquisa binária reside em sua complexidade temporal logarítmica, O(log n), que garante um desempenho muito superior ao da pesquisa linear em conjuntos extensos. Contudo, para que essa eficiência seja plenamente explorada, é necessária a manutenção rigorosa dos dados em ordem crescente ou decrescente, o que implica operações prévias de ordenação e custos adicionais relacionados a essa organização inicial.

Expandindo as abordagens para estruturas de dados não lineares, os algoritmos de pesquisa em árvores ganham destaque significativo, especialmente pela flexibilidade e capacidade de manter um equilíbrio eficiente entre tempo de busca e custo de manutenção. As árvores binárias de busca são uma representação hierárquica que armazenam elementos de maneira estruturada, permitindo buscas mais rápidas e eficientes do que listas ou vetores simples. Como vimos no tópico 3, a propriedade essencial dessas árvores é que cada nó possui valor maior do que qualquer elemento em sua subárvore esquerda e menor do que qualquer elemento em sua subárvore direita. Assim, realizar uma busca nessas árvores é intuitivamente semelhante

à pesquisa binária, seguindo-se um percurso lógico através de nós e subárvores, resultando também em complexidade média logarítmica O(log n). Implementações em Python normalmente empregam classes e métodos recursivos para alcançar uma implementação clara, concisa e de fácil manutenção.

No entanto, as árvores binárias convencionais podem apresentar um sério inconveniente, que é a possibilidade de degeneração. Caso inserções sucessivas ocorram em ordem estritamente crescente ou decrescente, a árvore binária pode degenerar em uma estrutura linear, perdendo sua eficiência e retornando para complexidade de pesquisa linear O(n). Para solucionar essa limitação surgem as árvores balanceadas, destacando-se as AVL e Red-Black. Elas são projetadas para manter automaticamente um equilíbrio estrutural que evita casos degenerativos, garantindo que os caminhos das raízes até as folhas permaneçam relativamente uniformes.

As árvores AVL realizam ajustes rígidos através de rotações simples ou duplas após cada inserção ou remoção, mantendo a altura das subárvores estritamente controlada. Por meio dessas rotações, as árvores AVL preservam uma diferença máxima de altura de uma unidade entre quaisquer duas subárvores de um mesmo nó. Essa propriedade garante tempos de busca consistentemente próximos à eficiência teórica máxima, O(log n). Apesar da eficiência garantida, as árvores AVL podem exigir maior número de rotações em certas operações, resultando em custo adicional significativo para inserções frequentes.

Já as árvores Red-Black introduzem uma abordagem menos rigorosa, porém mais eficiente em termos gerais, atribuindo cores (vermelho e preto) aos nós e estabelecendo regras específicas sobre a distribuição dessas cores ao longo da árvore. As inserções e as remoções desencadeiam processos de recoloração e eventuais rotações, que são menos frequentes e menos custosas que nas árvores AVL. Assim, as Red-Black preservam o balanceamento suficiente para garantir complexidade O(log n) com menor custo operacional nas modificações, o que as torna amplamente preferidas em aplicações reais, como bancos de dados e sistemas de arquivos. Em Python, embora existam implementações complexas dessas estruturas, elas frequentemente são encontradas em módulos e bibliotecas especializadas, simplificando significativamente sua adoção em cenários práticos.

Portanto, o domínio dessas técnicas de pesquisa é essencial para desenvolvedores Python que almejam otimizar eficiência computacional e desempenho em aplicações que manipulam grandes volumes de dados ou necessitam de respostas rápidas. A escolha entre pesquisa linear, pesquisa binária ou árvores balanceadas deve considerar criteriosamente características como volume de dados, frequência das operações, custo computacional de inserções e deleções, além dos requisitos específicos da aplicação. Com isso, desenvolvedores garantem não somente o desempenho ideal de suas aplicações, como maximizam a eficiência e a escalabilidade das soluções implementadas.

5.1 Pesquisas linear e binária: funcionamento e eficiência

A pesquisa linear percorre cada posição de uma coleção até encontrar o elemento desejado ou esgotar o conjunto. Durante esse processo, a comparação ocorre item a item, o que significa custo proporcional à quantidade total de registros. Quando o valor procurado ocupa a última posição, todas as verificações possíveis já terão sido realizadas, originando complexidade temporal O(n). Em contrapartida, o consumo de memória permanece constante, pois apenas índices e variáveis de controle precisam ser

mantidos. A simplicidade da técnica favorece sua adoção em estruturas não ordenadas ou em coleções muito pequenas, embora o crescimento do tempo de resposta torne-se perceptível conforme o volume de dados aumenta.

A pesquisa binária explora previamente a ordenação do conjunto. A cada iteração, o algoritmo seleciona o elemento central, confronta-o com o valor procurado e elimina imediatamente metade dos casos restantes, pois qualquer valor inferior reside na região esquerda e qualquer valor superior na região direita. A operação se repete de forma recursiva ou iterativa até restar um único candidato. Como o número de elementos avaliados diminui exponencialmente, a quantidade total de comparações situa-se em O(log n). A exigência de pré-ordenar a coleção pode introduzir custo adicional, contudo essa etapa costuma ocorrer apenas uma vez antes de inúmeras consultas posteriores, compensando o investimento inicial. A pesquisa binária mantém consumo de memória igualmente constante, pois utiliza apenas variáveis de controle para delimitar os limites esquerdo e direito da subseção ativa.

Ao comparar eficiência, observa-se que a pesquisa linear apresenta desempenho previsível independentemente da organização dos valores, enquanto a pesquisa binária oferece ganho expressivo quando trabalha sobre coleções já ordenadas. Em contextos nos quais atualizações frequentes reorganizam os dados, o tempo de ordenação pode atenuar os benefícios do método logarítmico; em cenários dominados por múltiplas consultas sobre conjuntos estáticos ou raramente modificados, a pesquisa binária destaca-se como escolha preferencial. A análise cuidadosa dos padrões de acesso, da frequência de inserções e do tamanho do conjunto orienta a seleção entre percorrer sequencialmente todas as posições ou restringir-se a divisões sucessivas que levam diretamente ao alvo.

Faremos um exercício prático. O comércio eletrônico (e-commerce) ampliou o alcance do varejo ao permitir que empresas comercializem produtos para consumidores em qualquer lugar do mundo, vinte e quatro horas por dia. Plataformas globais administram catálogos com dezenas de milhões de itens, cada qual descrito por atributos de preço, estoque, variação de cor, avaliações de usuários e metadados de logística. Paralelamente, a ascensão do Big Data trouxe a necessidade de armazenar, processar e analisar volumes maciços de informações gerados por cliques, pesquisas e transações em tempo real. Nessas circunstâncias, a eficiência na localização de itens – seja para exibir resultados de busca ao cliente, seja para alimentar recomendações personalizadas – torna-se crítica para a experiência do usuário e conversão de vendas.

Quando o sistema precisa retornar rapidamente o preço de um produto ou verificar se um SKU existe no estoque, duas estratégias clássicas entram em cena: pesquisa linear e pesquisa binária. A pesquisa linear percorre todos os registros até encontrar o alvo, revelando-se apropriada para listas curtas ou para catálogos ainda não classificados. Já a pesquisa binária, executada sobre coleções previamente ordenadas, divide repetidamente o espaço de busca pela metade, atingindo o item desejado em O(logn) operações. Em ambientes de Big Data, nos quais cada microssegundo conta, a escolha entre manter uma lista ordenada (acelera a busca, mas encarece inserções) ou varrer sequencialmente depende do padrão de acesso aos dados e do SLA da aplicação.

Observação

O SKU (Stock Keeping Unit ou Unidade de Manutenção de Estoque) é um código único atribuído a um produto específico para identificar e rastrear itens em um estoque. Ele ajuda a gerenciar inventário, vendas e logística, diferenciando produtos por características como modelo, tamanho, cor ou embalagem. Por exemplo, uma camiseta azul tamanho M pode ter um SKU diferente da mesma camiseta em vermelho tamanho G.

O SLA (Service Level Agreement ou Acordo de Nível de Serviço) da aplicação é um contrato ou documento que define os padrões de desempenho e disponibilidade esperados de uma aplicação ou serviço. Ele estabelece métricas específicas, como tempo de atividade (uptime), tempo de resposta, capacidade de processamento ou resolução de problemas, que o provedor do serviço deve cumprir. Por exemplo, um SLA pode garantir 99,9% de disponibilidade mensal ou um tempo de resposta inferior a 2 segundos para requisições. É usado para assegurar qualidade, confiabilidade e transparência entre o provedor e o cliente, com penalidades possíveis em caso de descumprimento.

Entender esses algoritmos simples continua essencial mesmo em sistemas que empregam bancos NoSQL, caches distribuídos e índices invertidos, pois em algum nível – seja na CPU, no SSD ou em estruturas intermediárias – comparações ordenadas ou sequenciais ocorrem. O exercício a seguir mostra como o Python pode simular as duas abordagens, fornecendo contadores de comparações para avaliar empiricamente o trade-off e instruindo o aluno a refletir sobre como grandes fornecedores gerenciam seus catálogos em produção.

Exemplo de aplicação

Exemplo 1 – E-commerce e Big Data

Uma loja virtual especializada em eletrônicos mantém um catálogo com 100.000 produtos. A aplicação front-end, porém, carrega em memória apenas os 5.000 itens mais populares para acelerar a navegação nas páginas de destaque.

As regras de negócio determinam que, ao pesquisar por SKU, o sistema deve:

- procurar primeiro nessa lista em memória;
- se o SKU estiver na lista quente, retornar seu preço imediatamente;
- caso contrário, verificar um arquivo ordenado em disco com todos os SKUs e preços, utilizando pesquisa binária;

- registrar o número de comparações executadas para posterior ajuste de cache.

Essas regras exigem a implementação de dois algoritmos distintos – linear em memória, binário em disco – além de instrumentação de métricas que ajudam a decidir, futuramente, quantos produtos devem compor a lista quente.

O exercício utiliza três recursos de Python que espelham soluções empregadas em sistemas de e-commerce. O primeiro é o módulo `random`, aplicado para gerar SKUs sintéticos e preços unitários de maneira reprodutível, imitando a ingestão de dados de um ERP. O segundo é `bisect`, biblioteca padrão que implementa busca binária sobre sequências ordenadas e, internamente, opera em linguagem C, oferecendo desempenho próximo ao limite da linguagem. O terceiro é o uso de dataclasses (via `@dataclass`) para modelar objetos Produto, favorecendo legibilidade e permitindo ordenação automática pelo atributo `sku` mediante o parâmetro `order=True`.

Na pesquisa linear, uma simples iteração `for` percorre a lista quente (com itens mais acessados), contando comparações até encontrar o SKU desejado ou atingir o fim. Embora aparentemente trivial, esse laço ilustra como coleções em cache (por exemplo, dicionários `Redis`) podem acelerar respostas quando o conjunto é pequeno e acessado com frequência. Já na pesquisa binária, a função `bisect_left` devolve a posição em que o SKU deveria estar na lista ordenada; comparando o valor encontrado com o procurado confirma-se a presença em O(logn). Para registrar estatísticas de execução, o programa encapsula cada algoritmo em um gerador, no qual a instrução yield entrega, a cada passagem pelo laço, um dicionário com as seguintes métricas: o número de comparações realizadas, a confirmação de êxito na busca, a posição obtida e a duração da operação; esse fluxo incremental preserva o desempenho da aplicação e reflete as práticas de observabilidade adotadas em arquiteturas de microsserviços.

Por fim, o exercício demonstra como concatenar os dois métodos: primeiro tentar localizar localmente (linear), depois recorrer ao arquivo completo (binário). Esse padrão híbrido de "cache + base de dados" reflete arquiteturas reais, nas quais dados quentes (mais usados) residem em memória volátil enquanto o acervo completo permanece em armazenamento persistente, evitando custo elevado de leitura aleatória em disco. A seguir, consta o código-fonte da solução:

```
from __future__ import annotations
from bisect import bisect_left
from dataclasses import dataclass
import random
import csv, os, time
from typing import List, Tuple, Dict

TAMANHO_QUENTE = 5_000    # cache em memória
TAMANHO_TOTAL = 100_000   # catálogo completo

@dataclass(order=True)
class Produto:
    sku: int
    preco: float
```

```
def gerar_catalogo(n: int) -> List[Produto]:
    random.seed(42)
    return [Produto(random.randint(1_000_000, 9_999_999),
                    round(random.uniform(10, 5000), 2))
            for _ in range(n)]

def salvar_ordenado(produtos: List[Produto], caminho: str):
    produtos_ordenados = sorted(produtos) # ordena por sku
    with open(caminho, "w", newline="") as arq:
        w = csv.writer(arq)
        for p in produtos_ordenados:
            w.writerow([p.sku, p.preco])
    return produtos_ordenados # devolve lista

def pesquisa_linear(lista: List[Produto], alvo: int) -> Dict:
    comparacoes = 0
    for p in lista:
        comparacoes += 1
        if p.sku == alvo:
            return {"encontrado": True, "preco": p.preco,
                    "comparacoes": comparacoes}
    return {"encontrado": False, "comparacoes": comparacoes}

def pesquisa_binaria(lista_ord: List[Produto], alvo: int) -> Dict:
    comparacoes = 1
    idx = bisect_left(lista_ord, Produto(alvo, 0.0))
    if idx < len(lista_ord) and lista_ord[idx].sku == alvo:
        return {"encontrado": True, "preco": lista_ord[idx].preco,
                "comparacoes": comparacoes}
    return {"encontrado": False, "comparacoes": comparacoes}

def carregar_catalogo_csv(caminho: str) -> List[Produto]:
    with open(caminho, newline="") as arq:
       return [Produto(int(row[0]), float(row[1])) for row in csv.reader(arq)]

def localizar_produto(sku_busca: int,
                      cache: List[Produto],
                      catalogo_ordenado: List[Produto]) -> Dict:
    t0 = time.perf_counter()
    res_cache = pesquisa_linear(cache, sku_busca)
    if res_cache["encontrado"]:
        res_cache["fonte"] = "cache"
        res_cache["tempo_ms"] = (time.perf_counter() - t0) * 1000
        return res_cache
    res_full = pesquisa_binaria(catalogo_ordenado, sku_busca)
    res_full["fonte"] = "catálogo"
    res_full["tempo_ms"] = (time.perf_counter() - t0) * 1000
    return res_full

# ---------- Demonstração ----------
def principal():
    # gera catálogo completo e salva em disco simulando banco
    todos = gerar_catalogo(TAMANHO_TOTAL)
    caminho_csv = "catalogo.csv"
    produtos_ordenados = salvar_ordenado(todos, caminho_csv)
```

```
    # cache quente: simplesmente as primeiras 5 000 posições
    cache_quente = produtos_ordenados[:TAMANHO_QUENTE]

    # Escolhe SKUs para teste: um no cache, um fora, um inexistente
    sku_cache = cache_quente[123].sku
    sku_disc = produtos_ordenados[80_000].sku
    sku_inexistente = 999_999_999

    for sku in (sku_cache, sku_disc, sku_inexistente):
        resultado = localizar_produto(sku, cache_quente, produtos_ordenados)
        print(f"Busca SKU {sku}: {resultado}")

    # limpeza
    os.remove(caminho_csv)

if __name__ == "__main__":
        principal()
```

O quadro 19 apresenta as funções do Python utilizadas no código-fonte e suas respectivas descrições.

Quadro 19 – Funções usadas no código-fonte do primeiro exercício prático

Função ou construção	Explicação de uso
from bisect import bisect_left	Importa a função `bisect_left`, que realiza busca binária eficiente para localizar a posição de inserção de um item em uma lista ordenada
@dataclass(order=True)	Gera automaticamente métodos como `__lt__`, `__le__` etc., permitindo comparação entre instâncias da classe. No código, isso permite ordenar e buscar objetos Produto por `sku`
sorted(lista)	Retorna uma nova lista com os elementos ordenados. Funciona com objetos que possuem operadores de comparação definidos (como Produto com `order=True`)
csv.writer(arquivo)	Cria um objeto para escrita de arquivos CSV. No código, é usado para salvar o catálogo de produtos ordenado por `sku`
csv.reader(arquivo)	Cria um iterador que lê linhas de um arquivo CSV, retornando listas de strings. Aqui, converte essas listas para instâncias de Produto
with open(caminho, modo)	Abre um arquivo e garante que ele será fechado corretamente após o bloco. É usado tanto para leitura quanto para escrita segura do catálogo em disco
time.perf_counter()	Função de alta resolução para medir intervalos curtos. Aqui, é usada para cronometrar a duração das buscas
os.remove(caminho)	Remove um arquivo do sistema de arquivos. Usado para limpar o CSV do catálogo após a demonstração
lista[:n]	Fatiamento de listas. Retorna uma sublista contendo os primeiros n elementos. No código, seleciona os produtos que formam o "cache quente"

O código começa com a importação de módulos e a definição de constantes. A instrução `from __future__ import annotations` habilita o uso de anotações de tipo postergadas, permitindo referências a tipos ainda não definidos. As constantes `TAMANHO_QUENTE` (5.000) and `TAMANHO_TOTAL` (100.000) definem o tamanho do cache em memória e do catálogo completo, respectivamente. O módulo `bisect` é usado para pesquisa binária, `dataclasses` para criar a classe `Produto`, `random` para geração de dados aleatórios, `csv` e `os` para manipulação de arquivos, `time` para medição de desempenho e `typing` para anotações de tipo.

A classe `Produto`, decorada com `@dataclass(order=True)`, define a estrutura de um produto com um SKU (inteiro) e um preço (float). O decorador `order=True` gera automaticamente métodos de comparação, permitindo ordenação dos produtos pelo SKU. A função `gerar_catalogo` cria uma lista de `n` produtos com SKUs aleatórios entre 1.000.000 e 9.999.999 e preços entre 10 e 5.000, usando a semente `42` para reprodutibilidade. A função `salvar_ordenado` ordena os produtos pelo SKU e os salva em um arquivo CSV, retornando a lista ordenada. O CSV armazena cada produto como uma linha com SKU e preço.

Para busca, o código implementa dois algoritmos: `pesquisa_linear` e `pesquisa_binaria`. A pesquisa linear itera sequencialmente pela lista, contando comparações até encontrar o SKU ou esgotar a lista, retornando um dicionário com o resultado (`encontrado`), o preço (se encontrado) e o número de comparações. A pesquisa binária utiliza a função `bisect_left` do módulo `bisect` para localizar a posição de inserção de um SKU em uma lista ordenada, aproveitando a eficiência logarítmica do algoritmo. Como a lista já está ordenada, a pesquisa binária é significativamente mais rápida para grandes catálogos, embora o código simplifique a contagem de comparações, retornando sempre 1.

A função `carregar_catalogo_csv` lê o arquivo CSV e recria a lista de objetos `Produto`. A função `localizar_produto` é o núcleo da lógica de busca, combinando cache e catálogo completo. Ela primeiro realiza uma pesquisa linear no cache (5.000 itens), que é mais rápido devido ao tamanho reduzido. Se o SKU não for encontrado, executa uma pesquisa binária no catálogo completo (100.000 itens). O tempo de execução é medido em milissegundos usando `time.perf_counter` e o resultado inclui a fonte da busca ("cache" ou "catálogo"), o status, o preço (se encontrado), o número de comparações e o tempo gasto.

A função `principal` demonstra o uso do sistema. Ela gera o catálogo completo, salva-o em um arquivo CSV, cria um cache com os primeiros 5.000 produtos ordenados e realiza três buscas de teste: um SKU presente no cache, um presente apenas no catálogo e um inexistente. Os resultados são impressos e o arquivo CSV é removido ao final. A estrutura modular do código, com funções bem definidas e uso de tipos anotados, facilita manutenção e extensibilidade. A escolha de pesquisa linear para o cache e binária para o catálogo reflete um equilíbrio entre simplicidade e desempenho, otimizando para cenários nos quais o cache tem alta probabilidade de acerto.

A seguir, apresentaremos algumas sugestões para que o leitor possa implementar no futuro, com o objetivo de sofisticar o código-fonte do exercício.

- Substituir a lista quente por um dicionário (`dict`) indexado pelo SKU, reduzindo comparações a uma operação O(1) por meio de hashing, embora ao custo de mais memória.

- Seria valioso adicionar um mecanismo LRU (leastrecentlyused) para atualizar dinamicamente o cache, com a ajuda do módulo `functools.lru_cache` ou de bibliotecas externas como `cachetools`.

- Para suportar prefixos de busca (autocompletar), construiria um trie ou índice invertido baseado em sufixos, acelerando consultas de texto.

- Finalmente, serializar o catálogo em formato Parquet e utilizar pandas para processamento em lote facilitaria análises de preço e estoque, integrando o exercício às práticas modernas de Data Engineering.

5.2 Pesquisa em árvores: busca binária e introdução a árvores balanceadas (AVL e Red-Black)

A busca em árvores constitui abordagem fundamental para recuperação eficiente de informação, pois permite explorar a estrutura hierárquica dos dados reduzindo drasticamente o conjunto de comparações necessárias até a localização de um valor. Na árvore binária de busca, cada nó guarda uma chave que mantém relação estrita com as subárvores filhas: todo elemento presente à esquerda apresenta valor inferior ao contido no nó central, enquanto todo elemento à direita apresenta valor superior. Tal propriedade conduz a um processo de pesquisa que se assemelha a uma versão dinâmica da divisão binária aplicada a listas ordenadas; inicia-se na raiz, compara-se a chave procurada com a chave do nó corrente e, dependendo do resultado, segue-se para a subárvore esquerda ou direita. A altura da árvore determina o número máximo de comparações, de forma que, em condições ideais nas quais as subárvores apresentam tamanhos equilibrados, alcança-se complexidade O(log n). Ocorre, entretanto, que inserções sucessivas em ordem crescente ou decrescente convertem a árvore em uma sequência degenerada comparável a uma lista encadeada, elevando a demora das pesquisas a O(n).

A fim de evitar degeneração surgem as árvores balanceadas, cuja lógica de manutenção garante alturas similares para ramos diferentes, preservando a eficiência logarítmica mesmo após longas séries de inserções e remoções. As árvores AVL constituem a primeira proposta amplamente adotada para alcançar tal equilíbrio. Cada nó armazena um fator de balanceamento equivalente à diferença entre as alturas das subárvores filhas. Quando essa diferença ultrapassa uma unidade, executa-se uma rotação simples ou dupla que redistribui os nós e restabelece o peso simétrico entre os lados da árvore. A rigidez dessas correções assegura alturas mínimas próximas ao limite teórico, de modo que pesquisas, inclusões e exclusões permanecem sempre dentro de O(log n). A contrapartida desse rigor reside no custo adicional introduzido pelas rotações, particularmente quando o fluxo de atualizações é intenso; ainda assim, o ganho em previsibilidade costuma compensar o esforço.

Observação

A sigla AVL (em árvores AVL) refere-se aos nomes dos criadores desse tipo de estrutura de dados: Adelson-Velsky e Landis. Eles propuseram, em 1962, a árvore AVL como sendo binária de busca balanceada, na qual a diferença de altura entre as subárvores esquerda e direita de cada nó (fator de balanceamento) é no máximo 1. Esse balanceamento garante que operações como inserção, remoção e busca tenham complexidade O(log n), tornando-as eficientes para grandes conjuntos de dados.

Com o propósito de reduzir a quantidade de ajustes, concebeu-se posteriormente a árvore Red-Black, estrutura que colore cada nó de vermelho ou preto e impõe regras específicas acerca da distribuição

dessas cores ao longo dos caminhos da raiz às folhas. Cada caminho contém a mesma quantidade de nós negros e nenhum nó vermelho possui filho vermelho. Tais restrições mantêm a altura no máximo em duas vezes o valor mínimo possível, o que preserva a complexidade O(log n) para todas as operações sem exigir tantas rotações quanto o esquema AVL. Ao custo de certo relaxamento na simetria perfeita obtém-se desempenho de inserção e remoção muitas vezes superior em cenários de grande carga, razão pela qual bancos de dados, sistemas de arquivos e bibliotecas de linguagem recorrem com frequência a esse formato.

Tanto nas árvores AVL quanto nas Red-Black, a pesquisa segue o mesmo percurso descendente efetuado em uma árvore binária tradicional, diferindo apenas na garantia de que o caminho percorrido jamais excederá o limite logarítmico em relação ao número de nós. Para desenvolvedores Python que desejam aproveitar tais benefícios, existem implementações prontas em bibliotecas especializadas, contudo a compreensão dos mecanismos internos, incluindo a lógica de rotação e o recolorimento, permanece indispensável ao ajuste fino de desempenho ou à depuração de sistemas complexos. Em última análise, entender a transição entre a simplicidade da busca binária convencional e a robustez oferecida por árvores balanceadas fornece recurso conceptual crucial para criar aplicações escaláveis e responsivas diante de grandes volumes de dados.

Saiba mais

O livro a seguir cobre estruturas de dados avançadas, incluindo árvores AVL e Red-Black, com implementações em Python. Ele explica os conceitos teóricos e fornece exemplos práticos de código.

GOODRICH, M. T.; TAMASSIA, R.; GOLDWASSER, M. *Data structures and algorithms in Python*. New Jersey: Wiley, 2021.

Outra ótima referência é a seguinte obra, que oferece uma introdução clara a árvores AVL e Red-Black, com foco em algoritmos e implementações em Python.

MILLER, B. N.; RANUM, L. *Problem solving with algorithms and data structures using Python.* [s.l.]: Franklin, Beedle & Associates, 2013.

Os bancos de dados modernos sustentam aplicações que variam de redes sociais globais a sistemas financeiros de alta frequência, armazenando e consultando volumes que ultrapassam trilhões de registros. A migração dessas cargas de trabalho para serviços em nuvem – ofertados por provedores como AWS, Azure e Google Cloud – intensificou a necessidade de consultas previsíveis, escaláveis e de baixa latência. Enquanto a elasticidade da nuvem facilita o provisionamento dinâmico de recursos de CPU e armazenamento, ela não elimina gargalos causados por varreduras completas (full-scan) em tabelas extensas. Assim, a eficiência de acesso continua dependente de estruturas de indexação que mantenham chaves em ordem e proporcionem busca O(logn).

Entre essas estruturas, árvores balanceadas – como AVL e Red-Black – destacam-se por garantir profundidade limitada mesmo sob inserções e remoções adversas, condição imprescindível para workloads multitenant que recebem dados de fontes imprevisíveis. Ao inserir ou remover registros, esses algoritmos executam rotações que preservam o equilíbrio, evitando degradação para O(n) comum a árvores binárias ingênuas. Nos bastidores de bancos relacionais e motores NoSQL, variações dessas árvores indexam dados em memória (buffer pool) antes de serem serializados em B-trees ou LSM-trees em disco. Compreender o funcionamento de uma AVL na memória ajuda profissionais a diagnosticar saltos de latência e a ajustar políticas de índice em serviços gerenciados.

Observação

B-trees e LSM-trees são estruturas de dados usadas para indexar e organizar dados em bancos de dados, otimizando operações de leitura e escrita.

- **B-trees**: são árvores balanceadas projetadas para sistemas de disco, em que cada nó pode conter múltiplas chaves e ponteiros para filhos. Elas minimizam acessos a disco ao manter uma estrutura hierárquica com poucos níveis, sendo ideais para bancos relacionais (como MySQL ou PostgreSQL), que exigem buscas rápidas e atualizações moderadas. São eficientes para leituras aleatórias e suportam operações como inserções, exclusões e consultas de intervalo.

- **LSM-trees (Log-Structured Merge-trees)**: são otimizadas para cenários de escrita intensiva, comuns em bancos NoSQL (como Cassandra ou RocksDB). Os dados são primeiro armazenados em memória (estruturas como árvores AVL ou similares) e, ao atingir um limite, são mesclados em camadas no disco em formato de arquivos ordenados. Isso reduz escritas em disco, mas pode aumentar a latência de leitura, exigindo compactação periódica para organizar os dados.

No contexto de serviços em nuvem, no qual instâncias podem ser reiniciadas ou migradas, estruturas de indexação em memória RAM precisam reconstruir-se rapidamente a partir de logs de compactação ou caches quentes. Implementar uma AVL em Python, com contadores que demonstram profundidade máxima e tempo de consulta, oferece visualização concreta de como esses índices preservam eficiência mesmo após milhões de operações. Esse exercício, portanto, serve como ponte entre teoria de algoritmos e práticas operacionais de bancos de dados distribuídos.

Exemplo de aplicação

Exemplo 2 – Bancos de dados e serviços em nuvem

Uma plataforma de streaming mantém um catálogo de cem mil filmes e séries, indexado pela combinação de `id_título` e timestamp de atualização. A cada novo dado de popularidade ou legenda, o serviço precisa localizar rapidamente o título para atualizar o registro.

As regras de negócio definem que:

- o índice em memória deve aceitar inserções em tempo real, mantendo profundidade não superior a ⌈log2n⌉+1\lceil \log_2 n \rceil + 1⌈log2 n⌉+1;
- buscas por id_título devem ocorrer em menos de 2 ms para listas de até 200.000 registros;
- toda remoção deve deixar a árvore balanceada para evitar buracos que desacelerem consultas de recomendação;
- ao final de cada hora, a aplicação precisará relatar profundidade máxima, média de rotações por inserção e tempo médio de busca, auxiliando a equipe SRE na definição de redimensionamento automático de instâncias.

Esses requisitos orientam a implementação a fornecer operações de inserção, busca e remoção balanceada, bem como métricas em execução.

O núcleo do exercício é a árvore AVL, cujos nós mantêm um fator de equilíbrio definido como a diferença entre alturas de subárvores esquerda e direita. Sempre que essa diferença sai do intervalo [–1, 1], rotações simples ou duplas restauram o equilíbrio. Em Python, a implementação requer:

- uma classe `NoAVL` com campos chave, valor, ponteiros esquerda e direita e altura atual;
- métodos auxiliares para calcular altura, fator de equilíbrio e rotações (`rotacao_esquerda`, `rotacao_direita`);
- funções recursivas de inserir, remover e buscar que devolvam o nó atualizado e contadores de rotações.

Python permite anotar tipos (`from __future__ import annotations`), aumentando legibilidade e habilitando ferramentas como `MyPy` em ambientes empresariais. `time.perf_counter_ns` mede nanossegundos por busca, aproximando-se do profiling real usado por observabilidade de bancos gerenciados. Uma classe `IndiceAVL` encapsulará a raiz da árvore e exporá métricas cumulativas de rotações e buscas, seguindo padrões de instrumentação usados em bibliotecas como `pymongo` ou `sqlalchemy`.

Ao final, geraremos 100.000 pares de (`id_título, timestamp`), inserindo-os na AVL, depois de consultar mil chaves escolhidas aleatoriamente. As métricas mostrarão que a profundidade cresce logaritmicamente e que o tempo médio de busca permanece pequeno, ilustrando porque bancos de dados confiam em estruturas balanceadas. A seguir, consta o código-fonte da solução:

```
from __future__ import annotations
import random, time
from dataclasses import dataclass
```

```
from typing import Optional, Tuple, Any

# ---------- Nó da árvore AVL ----------
@dataclass
class NoAVL:
    chave: int
    valor: Any
    esquerda: Optional['NoAVL'] = None
    direita: Optional['NoAVL'] = None
    altura: int = 1

# ---------- Classe do índice ----------
class IndiceAVL:
    def __init__(self):
        self.raiz: Optional[NoAVL] = None
        self.rotacoes = 0
        self.buscas = 0
        self.ns_total_busca = 0

    # ---------- Interface pública ----------
    def inserir(self, chave: int, valor: Any) -> None:
        self.raiz, rot = self._inserir(self.raiz, chave, valor)
        self.rotacoes += rot

    def buscar(self, chave: int) -> Optional[Any]:
        inicio = time.perf_counter_ns()
        no = self._buscar(self.raiz, chave)
        self.ns_total_busca += time.perf_counter_ns() - inicio
        self.buscas += 1
        return no.valor if no else None

    def remover(self, chave: int) -> None:
        self.raiz, rot = self._remover(self.raiz, chave)
        self.rotacoes += rot

    # ---------- Estatísticas ----------
    def profundidade(self) -> int:
        return self.raiz.altura if self.raiz else 0

    def media_rotacoes(self) -> float:
        return self.rotacoes / max(1, self.contagem())

    def tempo_medio_busca_ms(self) -> float:
        return (self.ns_total_busca / 1_000_000) / max(1, self.buscas)

    def contagem(self) -> int:
        def contar(no: Optional[NoAVL]) -> int:
            return 0 if not no else 1 + contar(no.esquerda) + contar(no.
direita)
        return contar(self.raiz)

    # ---------- Métodos internos ----------
    def _altura(self, no: Optional[NoAVL]) -> int:
        return no.altura if no else 0

    def _balancear(self, no: NoAVL) -> Tuple[NoAVL, int]:
        fator = self._altura(no.esquerda) - self._altura(no.direita)
        rot = 0
```

```
        # Rotação esquerda
        if fator > 1 and self._altura(no.esquerda.esquerda) >= self._altura(no.
esquerda.direita):
            no, rot = self._rot_dir(no), 1
        # Rotação esquerda-direita
        elif fator > 1:
            no.esquerda = self._rot_esq(no.esquerda)
            no, rot = self._rot_dir(no), 2
        # Rotação direita
        elif fator < -1 and self._altura(no.direita.direita) >= self._
altura(no.direita.esquerda):
            no, rot = self._rot_esq(no), 1
        # Rotação direita-esquerda
        elif fator < -1:
            no.direita = self._rot_dir(no.direita)
            no, rot = self._rot_esq(no), 2
        return no, rot

    def _atualizar_altura(self, no: NoAVL):
        no.altura = 1 + max(self._altura(no.esquerda), self._altura(no.
direita))

    # ---------- Inserção ----------
    def _inserir(self, no: Optional[NoAVL], chave: int, valor: Any) ->
Tuple[NoAVL, int]:
        if not no:
            return NoAVL(chave, valor), 0
        if chave < no.chave:
            no.esquerda, rot = self._inserir(no.esquerda, chave, valor)
        elif chave > no.chave:
            no.direita, rot = self._inserir(no.direita, chave, valor)
        else: # atualiza valor
            no.valor, rot = valor, 0
            return no, rot
        self._atualizar_altura(no)
        no, balance_rot = self._balancear(no)
        return no, rot + balance_rot

    # ---------- Busca ----------
    def _buscar(self, no: Optional[NoAVL], chave: int) -> Optional[NoAVL]:
        while no:
            if chave == no.chave:
                return no
            no = no.esquerda if chave < no.chave else no.direita
        return None

    # ---------- Remoção ----------
    def _remover(self, no: Optional[NoAVL], chave: int) ->
Tuple[Optional[NoAVL], int]:
        if not no:
            return None, 0
        if chave < no.chave:
            no.esquerda, rot = self._remover(no.esquerda, chave)
        elif chave > no.chave:
            no.direita, rot = self._remover(no.direita, chave)
        else: # chave encontrada
            if not no.esquerda or not no.direita:
                return (no.esquerda or no.direita), 0
```

```
                sucessor = self._minimo(no.direita)
                no.chave, no.valor = sucessor.chave, sucessor.valor
                no.direita, rot = self._remover(no.direita, sucessor.chave)
            if not no:
                return None, rot
            self._atualizar_altura(no)
            no, bal_rot = self._balancear(no)
            return no, rot + bal_rot

        def _minimo(self, no: NoAVL) -> NoAVL:
            while no.esquerda:
                no = no.esquerda
            return no

        # ---------- Rotações ----------
        def _rot_esq(self, z: NoAVL) -> NoAVL:
            y, z.direita = z.direita, z.direita.esquerda
            y.esquerda = z
            self._atualizar_altura(z)
            self._atualizar_altura(y)
            return y

        def _rot_dir(self, z: NoAVL) -> NoAVL:
            y, z.esquerda = z.esquerda, z.esquerda.direita
            y.direita = z
            self._atualizar_altura(z)
            self._atualizar_altura(y)
            return y

    # ---------- Demonstração ----------
    def principal():
        indice = IndiceAVL()
        total = 100_000
        random.seed(1)
        # Insere cem mil títulos
        for _ in range(total):
            chave = random.randint(1, 2_000_000_000)
            indice.inserir(chave, f"título_{chave}")

        print(f"Profundidade após {total:,} inserções: {indice.profundidade()}")
        print(f"Rotações médias por inserção: {indice.media_rotacoes():.3f}")

        # Executa mil buscas
        chaves_busca = random.sample(range(1, 2_000_000_000), 1_000)
        for k in chaves_busca:
            indice.buscar(k)
        print(f"Tempo médio de busca: {indice.tempo_medio_busca_ms():.3f} ms")

        # Remove mil chaves aleatórias
        for k in chaves_busca[:500]:
            indice.remover(k)
        print(f"Profundidade após remoções: {indice.profundidade()}")
        print(f"Total de rotações: {indice.rotacoes}")

    if __name__ == "__main__":
        principal()
```

O quadro 20 apresenta as funções do Python utilizadas no código-fonte e suas respectivas descrições.

Quadro 20 – Funções usadas no código-fonte do segundo exercício prático

Função ou construção	Explicação de uso
@dataclass com atributos padrão e tipados	Define uma estrutura de dados imutável com campos opcionais (Optional) e valor padrão (altura = 1). Simplifica a criação e a manutenção de nós da árvore
Optional[T]	Indica que um atributo pode ser do tipo T ou None. Utilizado para filhos esquerda e direita nos nós da árvore AVL
time.perf_counter_ns()	Mede o tempo com alta resolução e em nanossegundos, ideal para cronometrar operações curtas com maior precisão
Tuple[T1, T2]	Anotação de tipo que representa uma tupla com dois valores de tipos distintos. Utilizada como retorno das funções de inserção, remoção e balanceamento
Any (do typing)	Tipo genérico que aceita qualquer valor. No código, permite que os valores armazenados nos nós da árvore sejam de qualquer tipo
random.sample(população, k)	Gera uma amostra de k elementos únicos retirados da população. Aqui, seleciona chaves aleatórias para busca e remoção

No código apresentado, os módulos `random` e `time` são usados para gerar chaves aleatórias e medir o desempenho, enquanto `dataclasses` simplifica a criação da classe `NoAVL` e `typing` fornece tipos como `Optional` e `Tuple` para maior clareza nas assinaturas.

A classe `NoAVL`, definida com o decorador `@dataclass`, representa um nó da árvore AVL. Cada nó armazena uma chave inteira (chave), um valor de tipo genérico (valor), ponteiros para os filhos esquerdo e direito (esquerda e direita, ambos opcionais e do tipo `NoAVL`), e um campo `altura` inicializado como `1`, que rastreia a altura do nó na árvore. A utilização de `Optional['NoAVL']` nas definições dos filhos permite indicar que esses campos podem ser `None`, enquanto a string `NoAVL` é usada para referenciar a própria classe de forma recursiva, possibilitada pela importação de annotations.

A classe `IndiceAVL` encapsula a árvore AVL e fornece uma interface pública para manipulação e análise. No construtor `__init__`, inicializa-se a raiz como `None` e contadores para rotações (`rotacoes`), buscas realizadas (`buscas`) e tempo total de busca em nanossegundos (`ns_total_busca`). Esses contadores são usados para calcular estatísticas de desempenho.

Os métodos públicos da classe incluem `inserir, buscar e remover`. O método `inserir` adiciona um novo par chave-valor à árvore, delegando a lógica principal ao método interno `_inserir`, que retorna o nó atualizado e o número de rotações realizadas. O método `buscar` localiza um valor associado a uma chave, medindo o tempo de execução em nanossegundos com `time.perf_counter_ns` para estatísticas de desempenho. O método `remover` elimina um nó com a chave especificada, também rastreando rotações via `_remover`. Esses métodos são projetados para manter o balanceamento da árvore após cada operação.

A manutenção do balanceamento é central na implementação. O método `_altura` retorna a `altura` de um nó (ou `0` se o nó for `None`), enquanto `_atualizar_altura` recalcula a altura de um nó com base na altura máxima de seus filhos. O método `_balancear` verifica o fator de balanceamento, definido como a diferença entre as alturas dos filhos esquerdo e direito. Se o fator for maior que 1 ou

menor que -1, realiza-se uma rotação apropriada: rotação simples à direita ou esquerda para desequilíbrios diretos, ou rotações compostas (esquerda-direita ou direita-esquerda) para casos mais complexos. Cada rotação atualiza as alturas dos nós envolvidos e retorna o número de rotações realizadas (1 para rotações simples, 2 para compostas).

As rotações são implementadas em `_rot_esq` e `_rot_dir`. A rotação à esquerda, por exemplo, reposiciona o filho direito de um nó como o novo pai, ajustando os ponteiros e recalculando alturas. A rotação à direita opera de forma análoga. Essas operações garantem que a árvore permaneça balanceada, com diferença de altura entre subárvores esquerda e direita de no máximo 1.

Os métodos internos `_inserir`, `_buscar` e `_remover` implementam a lógica recursiva das operações principais. A inserção percorre a árvore comparando a chave com a chave do nó atual, inserindo recursivamente no filho apropriado ou atualizando o valor se a chave já existe. Após a inserção, a altura é atualizada e o balanceamento é verificado. A busca é iterativa, percorrendo a árvore até encontrar a chave ou atingir um nó `None`. A remoção é mais complexa: se o nó a ser removido tem apenas um filho, ele é substituído por esse filho; se tem dois filhos, o sucessor (menor nó da subárvore direita, obtido por `_minimo`) é usado para substituir a chave e o valor, e a remoção prossegue recursivamente. Em todos os casos, o balanceamento é mantido.

Os métodos de estatísticas, como profundidade, `media_rotacoes`, `tempo_medio_busca_ms` e `contagem`, fornecem métricas sobre a árvore. A `profundidade` retorna a `altura` da raiz, a média de rotações divide o total de rotações pelo número de nós (calculado recursivamente por `contagem`) e o tempo médio de busca converte o tempo acumulado em nanossegundos para milissegundos, dividido pelo número de buscas. Essas métricas são úteis para avaliar a eficiência da implementação.

A função `principal` demonstra o uso da árvore AVL. Ela cria uma instância de `IndiceAVL`, insere 100.000 chaves aleatórias geradas com `random.randint`, realiza 1.000 buscas e remove 500 chaves. Após cada etapa, exibe estatísticas como profundidade, média de rotações por inserção, tempo médio de busca e número total de rotações. A semente fixa `random.seed(1)` garante reprodutibilidade dos resultados.

Tecnicamente, o código é eficiente e bem estruturado. A árvore AVL mantém operações em O(log n) devido ao balanceamento. O uso de `dataclasses` e anotações de tipo melhora a legibilidade e a manutenibilidade. A medição de desempenho com `time.perf_counter_ns` permite análises precisas, enquanto o rastreamento de rotações oferece insights sobre o custo do balanceamento. A implementação é robusta, lidando corretamente com casos especiais, como nós sem filhos ou chaves duplicadas, e fornece uma interface clara para uso em aplicações que requerem buscas rápidas e atualizações dinâmicas.

A seguir, apresentaremos algumas sugestões para que o leitor possa implementar no futuro, com o objetivo de sofisticar o código-fonte do exercício.

- Uma expansão profissional integraria esse índice AVL ao módulo `shelve` ou `sqlite3`, persistindo nós em disco e medindo impacto de paginação.

- Poder-se-ia também implementar árvore RedBlack, comparando número de rotações e profundidade com a versão AVL.

- Para refletir sharding em nuvem, diferentes instâncias de IndiceAVL poderiam ser distribuídas em threads via `concurrent.futures`, reproduzindo splits de partição.

- Por fim, exportar métricas para o Prometheus e exibir gráficos de profundidade e rotações em tempo real aproximaria o exercício de observabilidade utilizada em bancos como Amazon Aurora e Cloud Spanner, fechando o ciclo entre teoria e operação.

6 ESTRUTURAS AVANÇADAS DE DADOS

As estruturas avançadas de dados desempenham papel crucial na otimização de desempenho e eficiência em aplicações computacionais, oferecendo soluções práticas para armazenar, organizar e recuperar informações rapidamente. Em Python, duas dessas estruturas destacam-se de forma expressiva: as tabelas hash (Hash Tables) e os heaps binários, cada qual com suas particularidades e aplicações próprias.

As tabelas hash representam uma poderosa estrutura baseada no conceito matemático de espalhamento uniforme de valores através de uma função específica, conhecida como função hash. O princípio fundamental dessas tabelas é a transformação de chaves em valores numéricos que, por sua vez, correspondem diretamente às posições em um vetor, permitindo recuperação extremamente rápida dos elementos. As funções hash são desenhadas especificamente para minimizar a ocorrência de colisões, ou seja, situações em que diferentes chaves geram o mesmo valor numérico, levando ao compartilhamento de posições e à necessidade de tratamento adicional para resolver o conflito. Apesar do esforço em criar funções que reduzam colisões ao mínimo, sua ocorrência é inevitável em cenários reais; por esse motivo, técnicas de resolução tornam-se necessárias.

Entre as estratégias comuns para tratamento de colisões estão o encadeamento separado e o endereçamento aberto. No encadeamento separado, cada posição do vetor armazena uma lista ou estrutura secundária, que mantém todos os elementos que compartilham aquela posição. Essa abordagem permite simplicidade de implementação e manutenção da eficiência geral das operações, mesmo que as colisões sejam frequentes. Por outro lado, o endereçamento aberto utiliza um sistema sequencial de sondagem para localizar um espaço vazio no próprio vetor principal, reduzindo consumo de memória adicional e mantendo desempenho eficiente enquanto houver baixa densidade de elementos. Ambas as abordagens encontram aplicações práticas na linguagem Python, sendo especialmente evidentes na implementação interna do tipo de dado `dict`, que utiliza uma estrutura hash otimizada, permitindo inserções, buscas e exclusões em tempo médio constante O(1).

Outras estruturas avançadas amplamente utilizadas são os heaps binários, tipos específicos de árvores binárias completas que seguem um critério de ordenação entre o nó pai e seus filhos. Nos heaps, cada nó possui valor maior ou igual (heap máximo) ou menor ou igual (heap mínimo) do que seus filhos imediatos. Essa propriedade estrutural garante acesso eficiente ao elemento de maior ou menor prioridade, que se encontra sempre no topo da árvore, sendo especialmente útil em filas de prioridade e algoritmos de escalonamento.

Os heaps binários permitem inserções e remoções em tempo logarítmico O(log n), mantendo o balanceamento necessário para eficiência constante. O Python fornece diretamente suporte a heaps através do módulo `heapq`, oferecendo funções simples e práticas para transformar listas comuns em heaps, inserir novos elementos e remover elementos prioritários rapidamente. Essa facilidade permite rápida adoção dessa estrutura em cenários práticos, como gerenciar eventos por ordem cronológica ou realizar operações constantes em filas de processamento.

Além de sua utilização direta, os heaps fornecem base conceitual para o algoritmo Heap Sort, método avançado de ordenação que explora as propriedades da estrutura heap para ordenar eficientemente grandes conjuntos de dados. O Heap Sort primeiramente organiza o vetor em forma de heap máximo, garantindo que o maior elemento fique sempre na posição inicial. Ao extrair repetidamente esse elemento e reorganizar o heap, o algoritmo gera, progressivamente, uma lista ordenada, com complexidade garantida O(n log n) em todos os casos, independentemente da disposição inicial dos dados. Sua vantagem adicional reside em operar diretamente sobre o vetor original, sem necessidade de memória extra proporcional ao tamanho do conjunto, conferindo ao Heap Sort especial relevância em ambientes com limitações severas de espaço ou necessidade de previsibilidade de desempenho.

6.1 Hash Tables: conceito, funções hash e resolução de colisões

A tabela hash em Python apresenta arquitetura baseada em vetor de slots que armazena pares chave-valor, fornecendo acesso médio em tempo constante devido ao mapeamento numérico gerado por funções hash. Cada chave passa por transformação determinística executada por `PyObject_Hash`, que devolve inteiro de largura adaptada à plataforma, incorporando aleatoriedade de execução para atenuar ataques de colisão intencional. O valor obtido após essa transformação indica posição inicial no vetor principal, construído com tamanho sempre em potência de dois a fim de permitir cálculo rápido de índices por operação de máscara bit a bit. Em objetos definidos pelo usuário, a coerência da função hash exige que chaves iguais retornem valores idênticos e que a propriedade de imutabilidade lógica seja respeitada, pois qualquer alteração posterior ao cálculo poderia inviabilizar a localização do elemento.

Quando duas ou mais chaves geram resultado idêntico, a estrutura recorre a endereçamento aberto com algoritmo de sondagem que emprega perturbação aritmética. O processo desloca o ponteiro de busca por incrementos dependentes dos bits altos do hash original junto a fator de perturbação reduzido a cada iteração, visitando posições subsequentes até encontrar slot livre ou chave correspondente. Esse modelo evita listas encadeadas adicionais, preservando a localidade de memória e diminuindo acessos cache adversos. Durante inserções, se a taxa de ocupação ultrapassa limite predefinido – setenta e poucos por cento do vetor – ocorre redimensionamento automático, no qual nova tabela com capacidade ampliada é alocada e todas as chaves sofrem reinserção utilizando os hashes previamente computados. Tal operação mantém complexidade amortizada aceitável, já que o custo elevado do rehash distribui-se ao longo de várias inserções.

Python adota ainda marcação especial para slots liberados a fim de evitar falhas na sequência de sondagem. Ao remover elemento, o slot recebe sinalizador dummy e permanece ocupando a posição até que o redimensionamento subsequente recicle o espaço, garantindo que buscas posteriores não cessem prematuramente. Esse detalhe técnico impede degradação de desempenho provocada por buracos na

sequência de sondagem, preservando consistência das operações de leitura. A partir da versão 3.6 da linguagem, a ordem de inserção passou a ser mantida devido à reorganização interna do vetor em dois arranjos: um para índices compactos e outro para pares efetivos, diminuindo realocação de objetos e favorecendo iteração previsível. Tal mudança não compromete princípios fundamentais de dispersão nem a eficiência de consulta, pois a lógica de sondagem permanece inalterada.

No contexto de segurança, a função `hash` global aplica chave secreta (sal) gerada a cada inicialização do interpretador, técnica conhecida como hash randomization. Esse sal impede que o invasor antecipe colisões em massa, preservando tempo de resposta de aplicações web expostas a entradas hostis. Desenvolvedores que definem suas próprias classes devem garantir compatibilidade entre `__hash__` e `__eq__`, pois igualdade verdadeira obriga hashes iguais, enquanto desigualdade não impõe qualquer relação específica. O descumprimento dessas regras compromete operações internas da tabela, resultando em falhas de busca ou duplicação indesejada de chaves.

A combinação de vetor de potência de dois, sondagem com perturbação, marcação de exclusão preguiçosa e redimensionamento dinâmico confere ao dicionário padrão robustez e desempenho consistente na maioria dos cenários. Quando cargas de trabalho apresentam perfil de chaves extensas ou probabilidades elevadas de colisão, recomenda-se avaliar hash personalizado que distribua melhor os valores. Em contrapartida, aplicações focadas em manipular grande quantidade de pares pequenos beneficiam-se da otimização de memória introduzida no layout compartilhado, no qual múltiplos objetos de mesmo tipo reutilizam tabela de índices. Pela atenção a esses detalhes, a infraestrutura de tabela hash em Python demonstra equilíbrio entre simplicidade de uso na superfície e engenhosidade de engenharia no núcleo, atendendo simultaneamente requisitos de velocidade, segurança e economia de recursos.

Nos últimos anos, a cibersegurança consolidou-se como disciplina central para a continuidade de operações corporativas e a proteção da privacidade individual. Ataques de ransomware, fraudes em sistemas de pagamento e espionagem industrial evidenciam que a informação digital tornou-se alvo permanente de agentes mal-intencionados. Verificar a autenticidade de arquivos por meio de assinaturas digitais e, ao mesmo tempo, detectar padrões anômalos em fluxos de dados são tarefas recorrentes em centros de operação de segurança. Em ambos os casos, a resposta precisa ocorrer em frações de segundo, pois qualquer atraso possibilita a propagação de ameaças ou a invalidação de evidências. Para alcançar essa rapidez, profissionais recorrem a estruturas de dados baseadas em hashing capazes de indexar milhões de registros na memória e recuperar itens em tempo O(1).

A indexação por tabela de dispersão permite que sumários criptográficos – representados por cadeias hexadecimais resultantes de SHA-256 ou algoritmos equivalentes – sejam armazenados como chaves e consultados sem varredura exaustiva. Quando um arquivo chega ao sistema, o motor calcula o seu hash; se o resumo já existir no índice e for associado a uma assinatura válida, a verificação conclui-se instantaneamente. Caso contrário, novos metadados são gravados e a assinatura é submetida à validação assíncrona. Paralelamente, contadores associados a cada chave revelam aumentos repentinos de ocorrência, sinalizando spikes de atividade que podem indicar distribuição de malware ou comportamento anômalo de usuários internos. O mesmo mecanismo serve a sistemas de monitoramento de logs, que armazenam impressões digitais de linhas de registro e acusam, por exclusão, a presença de entradas desconhecidas.

A importância desse tipo de indexação cresce à medida que arquiteturas de microsserviços e dispositivos IoT multiplicam a superfície de ataque. Manter uma única fonte de verdade em banco relacional não basta, é preciso replicar a lista de assinaturas em memória dentro dos próprios serviços, reduzindo latência de ida-e-volta. Hash tables escritas em linguagens gerenciadas, como Python, facilitam a prototipagem de novos detectores e esquemas de assinatura antes que versões otimizadas em linguagem C ou Rust entrem em produção. Portanto, compreender como construir, consultar e atualizar uma tabela de dispersão tornou-se competência fundamental para equipes de cibersegurança moderna.

Exemplo de aplicação

Exemplo 3 – Cibersegurança

Um provedor de email corporativo deseja bloquear anexos maliciosos sem degradar a experiência do usuário. O pipeline de filtragem recebe cerca de cem mil arquivos por hora, calcula o hash SHA-256 de cada anexo e consulta um índice de assinaturas aprovadas mantido em memória RAM.

A política de proteção estabelece que, se o hash estiver na tabela e possuir selo de confiança emitido por laboratório interno, o anexo será liberado de imediato. Caso o hash não exista, o sistema deverá inserir a nova entrada com carimbo temporal e encaminhar o arquivo para análise em sandbox.

Além disso, se um mesmo hash desconhecido aparecer mais de cinquenta vezes em quinze minutos, um alerta de possível campanha de spam deverá ser aberto. Essas exigências implicam atualizações frequentes da estrutura, contagem de ocorrências e expiração automática de registros antigos para que a memória não cresça indefinidamente.

A implementação utiliza o módulo `hashlib` para produzir resumos criptográficos de arquivos, prática cotidiana em verificação de integridade. O índice em si apoia-se no dicionário nativo do Python, que adota a técnica de hashing aberto com endereçamento direto; a distribuição uniforme das chaves depende da função `__hash__` das strings que representam os SHA-256. Para registrar a primeira aparição e contar reincidências, cada valor armazena uma tupla contendo timestamp de inclusão, número de ocorrências e um sinalizador de confiança. O módulo `time` fornece carimbo em segundos, permitindo cálculo de janelas móveis, enquanto `collections.deque` sustenta uma fila de hashes recentes, facilitando remoção de itens fora da janela sem percorrer toda a tabela. A decisão de alerta baseia-se em contagem simples, evitando processamento estatístico pesado que comprometeria a latência.

Embora o dicionário do Python resolva colisões internamente, o código ilustra manualmente como computar o índice de bucket usando `hash()` e módulo de capacidade para fins didáticos, mostrando ao aluno a ligação entre função de dispersão e localização da chave. Essa visualização reproduz raciocínio que engenheiros aplicam ao configurar tamanho inicial de hash tables em sistemas de logging de alto volume. Além disso, a demonstração de expiração periódica de registros simula garbage collection necessário em serviços de nuvem, nos quais recursos são pagos por uso efetivo e estruturas esquecidas podem encarecer a fatura. O exercício emprega dataclasses para encapsular os campos do valor associado a cada hash, conferindo clareza ao contrato de dados e compatibilidade com analisadores estáticos. A seguir, consta o código-fonte da solução:

```
from __future__ import annotations
import hashlib
import time
from dataclasses import dataclass
from collections import deque
from typing import Dict, Tuple

JANELA_ALERTA_SEG = 900        # quinze minutos
LIMITE_ALERTA = 50             # ocorrências para disparar alerta
TTL_REGISTRO = 3600            # expirar após uma hora
CAPACIDADE_BUCKETS = 131071    # primo próximo a 2^17 para exemplo

@dataclass
class Registro:
    timestamp: float
    ocorrencias: int
    confiavel: bool

class TabelaAssinaturas:
    def __init__(self):
        self.tabela: Dict[str, Registro] = {}
        self.fila_recent: deque[Tuple[str, float]] = deque()

    def _hash_indice(self, chave: str) -> int:
        return hash(chave) % CAPACIDADE_BUCKETS

    def _limpar_expirados(self, agora: float):
        while self.fila_recent and agora - self.fila_recent[0][1] > JANELA_ALERTA_
SEG:
            chave, t = self.fila_recent.popleft()
            reg = self.tabela.get(chave)
            if reg and agora - reg.timestamp > TTL_REGISTRO:
                del self.tabela[chave]

    def registrar_anexo(self, conteudo: bytes) -> Tuple[bool, bool]:
        agora = time.time()
        assinatura = hashlib.sha256(conteudo).hexdigest()
        indice = self._hash_indice(assinatura) # demonstração didática
        reg = self.tabela.get(assinatura)
        if reg:
            reg.ocorrencias += 1
            self.fila_recent.append((assinatura, agora))
            alerta = reg.ocorrencias >= LIMITE_ALERTA
            return reg.confiavel, alerta
        novo = Registro(timestamp=agora, ocorrencias=1, confiavel=False)
        self.tabela[assinatura] = novo
        self.fila_recent.append((assinatura, agora))
        self._limpar_expirados(agora)
        return False, False # primeiro contato, não confiável, sem alerta

    def marcar_confiavel(self, assinatura: str):
        reg = self.tabela.get(assinatura)
        if reg:
            reg.confiavel = True

# ---------- Demonstração ----------
def principal():
```

```
    verificador = TabelaAssinaturas()
    seguro = b"documento_legitimo"
    malicioso = b"macro_malware"
    # registra arquivo confiável conhecido

verificador.marcar_confiavel(hashlib.sha256(seguro).hexdigest())

    # simula fluxo de anexos
    for i in range(55):
        conf, alerta = verificador.registrar_anexo(seguro if i == 0 else
malicioso)
        if i in (0, 1, 49, 54): # imprime quatro amostras
          estado = "confiável" if conf else "desconhecido"
          print(f"Iteração {i:02d}: arquivo {estado}, alerta={alerta}")
    # imprime estatísticas finais
    total = len(verificador.tabela)
    print(f"\nRegistros na tabela após fluxo: {total}")
    profund = max((time.time() - r.timestamp) for r in verificador.tabela.values())
    print(f"Tempo máximo de permanência (s): {profund:.1f}")

if __name__ == "__main__":
        principal()
```

O quadro 21 apresenta as funções do Python utilizadas no código-fonte e suas respectivas descrições.

Quadro 21 – Funções usadas no código-fonte do terceiro exercício prático

Função ou construção	Explicação de uso
hashlib.sha256(dados).hexdigest()	Calcula o hash SHA-256 de uma sequência de bytes e retorna o resultado como string hexadecimal. Garante integridade e identidade dos conteúdos
hash(valor) % CAPACIDADE_BUCKETS	Aplica a função de hash embutida do Python e usa o operador módulo para mapear a chave para um índice dentro de uma tabela de tamanho fixo (técnica comum em tabelas hash)
deque com tuplas	Utiliza deque para manter ordem temporal de elementos com pares (assinatura, timestamp), permitindo remoções eficientes de registros antigos (estratégia FIFO)
agora – self.fila_recent[0][1] > JANELA_ALERTA_SEG	Verifica se o tempo atual excede a janela limite para considerar ocorrências recentes. Garante que apenas atividades recentes influenciem alertas
max(expressão for elemento in iterável)	Retorna o maior valor resultante de uma expressão aplicada a cada elemento. No código, calcula o tempo de permanência mais longo de registros ainda ativos

O código Python apresentado implementa um sistema para monitoramento de anexos com base em suas assinaturas criptográficas, utilizando uma abordagem orientada a objetos e estruturas de dados eficientes para gerenciar registros com expiração temporal. Ele emprega a biblioteca padrão do Python, incluindo módulos como `hashlib` para geração de assinaturas, `time` para manipulação de timestamps, `dataclasses` para definição de estruturas de dados, `collections.deque` para uma fila de registros recentes e `typing` para anotações de tipo.

Lembrete

A importação from__future__import annotations assegura compatibilidade com anotações de tipo avançadas, permitindo referências a tipos ainda não definidos.

O sistema define constantes globais que configuram seu comportamento: `JANELA_ALERTA_SEG` estabelece uma janela de 15 minutos (900 segundos) para monitoramento de alertas, `LIMITE_ALERTA` determina o número mínimo de ocorrências (`50`) para disparar um alerta, `TTL_REGISTRO` define o tempo de vida de um registro como 1 hora (3.600 segundos) e `CAPACIDADE_BUCKETS` especifica um número primo (`131071`) para dimensionar a tabela hash, otimizando a distribuição de chaves.

Observação

A escolha do valor `131071` reflete uma decisão técnica voltada para otimizar o desempenho da tabela hash, embora o método `_hash_indice` seja usado apenas para demonstração didática. O número `131071` é primo próximo a 2^{17} (131.072) e sua seleção está fundamentada em práticas comuns de design de tabelas hash.

Em sistemas que utilizam tabelas hash, o tamanho da tabela (ou o número de buckets) influencia diretamente a eficiência das operações de inserção, consulta e remoção. Um tamanho primo é frequentemente escolhido porque números primos minimizam colisões ao distribuir as chaves de forma mais uniforme. Quando uma função hash é aplicada a uma chave e o resultado é mapeado para um índice usando o operador módulo (como em `hash(chave)%CAPACIDADE_BUCKETS`), um número primo como divisor reduz a probabilidade de múltiplas chaves serem mapeadas para o mesmo índice, especialmente quando as chaves não seguem um padrão previsível.

O valor `131071` é particularmente conveniente por ser muito próximo de uma potência de 2, o que alinha bem com otimizações de memória e cálculos em sistemas computacionais, que frequentemente trabalham com potências de 2. Escolher um primo ligeiramente menor ou maior que uma potência de 2, como `131071`, combina os benefícios de um tamanho grande o suficiente para suportar muitas entradas sem desperdiçar memória e a propriedade de primalidade para distribuição uniforme.

Além disso, o comentário no código indica que `131071` foi selecionado para exemplo, sugerindo que o valor é ilustrativo. Em uma implementação real, o tamanho da tabela poderia ser ajustado com base em requisitos específicos, como o volume esperado de assinaturas ou restrições de memória.

A classe `Registro`, decorada com `@dataclass`, estrutura os dados de cada entrada, armazenando o timestamp da criação ou última atualização, o número de ocorrências e um indicador booleano de confiabilidade. A classe principal, `TabelaAssinaturas`, gerencia uma tabela hash (`self.tabela`) que mapeia assinaturas para objetos `Registro` e uma fila dupla (`self.fila_recent`) que rastreia a ordem de inserção para expiração de registros.

O método `_hash_indice` calcula um índice para uma chave (assinatura) usando a função `hash` do Python e o operador módulo com `CAPACIDADE_BUCKETS`. Embora o código utilize esse método para demonstração didática, ele não é diretamente aplicado na implementação, já que a tabela hash usa as assinaturas como chaves. O método `_limpar_expirados` remove registros expirados com base no tempo atual e na janela de alerta, iterando sobre `fila_recent` e deletando entradas da tabela cujo tempo de vida exceda `TTL_REGISTRO`.

O método `registrar_anexo` é o núcleo do programa. Ele recebe o conteúdo de um anexo como bytes, calcula sua assinatura SHA-256 usando `hashlib` e verifica se a assinatura já existe na tabela. Se existir, incrementa o contador de ocorrências, adiciona a entrada à fila recente e verifica se o número de ocorrências atingiu o limite de alerta. Caso a assinatura seja nova, cria um registro com uma ocorrência, marca-o como não confiável e o adiciona à tabela e à fila, chamando `_limpar_expirados` para manutenção. O método retorna uma tupla indicando se o anexo é confiável e se um alerta foi disparado. O método `marcar_confiavel` permite marcar uma assinatura existente como confiável, alterando o campo correspondente no registro.

A função `principal` demonstra o uso do sistema. Ela cria uma instância de `TabelaAssinaturas`, define dois anexos simulados (um "seguro" e um "malicioso"), marca o anexo seguro como confiável e simula um fluxo de 55 registros, sendo o primeiro seguro e os demais maliciosos. Durante a simulação, imprime o estado (confiável ou desconhecido) e a condição de alerta em quatro iterações específicas. Ao final, exibe o número total de registros na tabela e o tempo máximo de permanência de um registro.

A seguir, constam sugestões de melhoria para que você implemente no futuro.

- Uma evolução prática poderia integrar contadores probabilísticos, como HyperLogLog, para medir cardinalidade de hashes sem armazenar cada ocorrência, reduzindo consumo de memória em ambientes de altíssimo volume.

- Outra ampliação consistiria em substituir a deque por um heap de min-timestamp, permitindo expiração ordenada mesmo sob picos irregulares, embora exigindo manutenção de ponteiros cruzados.

- Em cenários distribuídos, replicar o índice via Redis Cluster ou DynamoDB com partição por hash prefixo garantiria escalabilidade horizontal, enquanto filtros Bloom reduziriam consultas remotas ao descartar imediatamente hashes nunca vistos.

- Para fortalecer a verificação de confiança, o exercício poderia incorporar assinatura digital ECDSA e armazenar a chave pública do laboratório no próprio índice, criando associação direta entre digest e certificado, aproximando-se de práticas empregadas em repositórios de amostras de malware corporativos.

6.2 Heaps: heaps binários e introdução ao Heap Sort

Um heap é uma estrutura de dados que organiza números (ou outros valores) de forma a sempre manter o menor (ou o maior) deles facilmente acessível, como se fosse uma lista especial com uma regra específica. Imagine uma árvore genealógica, mas em vez de pessoas, temos números, e cada pai (nó pai) tem que ser menor (heap mínimo) ou maior (heap máximo) que seus filhos. Essa organização é guardada em uma lista (vetor) de forma compacta, sem precisar de ponteiros, o que economiza memória.

Por exemplo, em um heap mínimo, o número na raiz (o topo) é sempre o menor de todos. Quando você quer adicionar um número novo, ele é colocado no final da lista e sobe, trocando de lugar com outros números até encontrar seu lugar certo, respeitando a regra de que o pai é menor que os filhos. Isso é rápido, leva um tempo proporcional à altura da árvore, que é pequena (cresce lentamente, como um logaritmo). Para remover o menor número, você pega a raiz, coloca o último número da lista no lugar dela e faz ele descer até que a regra do heap seja restaurada.

O heap binário representa variação específica de árvore completa armazenada em vetor contíguo, recurso que oferece acesso imediato à raiz sem sacrificar compacidade de memória. Cada nó obedece à propriedade de dominância: em um heap mínimo, o valor guardado no nó pai é menor ou igual ao contido em cada filho; no heap máximo, ocorre a relação inversa. A localização dos filhos deriva de aritmética simples sobre o índice do pai, fato que dispensa ponteiros explícitos e favorece localidade de cache. Inserir novo elemento exige anexá-lo ao final do vetor para preservar completude estrutural e, em seguida, realizar operação de subida na qual o valor move-se em direção à raiz enquanto violações da dominância persistirem. Essa etapa consome tempo O(log n) porque a altura da árvore cresce logaritmicamente em relação ao número total de nós.

Remover o elemento prioritário – sempre situado na raiz – envolve trocá-lo com o último item do vetor, reduzir o tamanho efetivo e executar descida que restaura a propriedade comparando o novo valor da raiz com os filhos e trocando-o repetidamente com o menor (ou maior) deles. O módulo `heapq` de Python implementa min-heap mediante lista comum, expondo funções como `heappush`, `heappop` e `heapreplace`, de modo que qualquer sequência pode converter-se em heap através de `heapify`, operação que executa descidas sucessivas a partir da metade inferior do vetor e conclui em tempo linear O(n).

Lembrete

O `heapq` já foi introduzido, de forma discreta, no terceiro exercício do tópico 4. Neste tópico, veremos com mais detalhes.

O Heap Sort surge diretamente da propriedade de dominância ao combinar construção de heap máximo com extrações sucessivas. Inicialmente todo o vetor torna-se heap, garantindo que o maior valor resida na posição zero. Em seguida, a raiz troca-se com o elemento da última posição ainda não ordenada, reduzindo o intervalo considerado e restaurando o heap por descida. Após n-1 repetições, o vetor encontra-se completamente ordenado em ordem crescente. A fase de construção consome O(n); cada remoção exige O(log n) e ocorre n-1 vezes, resultando em complexidade total O(n log n) independentemente da disposição inicial dos dados. Essa abordagem opera in-place, visto que o heap

permanece dentro do próprio vetor durante todo o processo, exigindo memória adicional constante além de variáveis auxiliares. Não se trata de algoritmo estável, pois elementos equivalentes podem trocar de posição durante rotações internas, característica irrelevante para dados sem chaves secundárias, porém restritiva em fluxos nos quais a preservação de ordem original importe.

A eficiência prática do Heap Sort deriva do padrão de acesso quase sequencial que evita realocações volumosas; contudo, as constantes de tempo frequentemente superam as observadas em Quick Sort em cenários médios devido à maior incidência de trocas. O heap, por sua vez, mantém relevância ímpar em filas de prioridade, agendadores de eventos e obtenção dos k menores ou maiores elementos, já que a construção linear seguida de k extrações entrega custo O(n + k log n), superando abordagens ingênuas que precisariam ordenar completamente o conjunto. Dominar internamente o arranjo vetorial, compreender a diferença entre subida e descida e reconhecer o impacto de operações de heap em estruturas de nível superior capacita profissionais Python a explorar a versatilidade do heap binário tanto na ordenação quanto na gerência de prioridades, alcançando soluções mais responsivas e parcimoniosas em recursos.

Saiba mais

Focado em algoritmos e estruturas de dados, o livro a seguir explora hash tables com ênfase em sua eficiência para busca e armazenamento, incluindo detalhes sobre funções de hash e tratamento de colisões. Para heaps, ele discute a implementação de filas de prioridade usando o módulo heapq e sua aplicação em algoritmos como o heapsort. O texto é acessível e inclui exemplos claros em Python.

HETLAND, M. L. *Python algorithms*: mastering basic algorithms in the Python language. 2. ed. New York: Apress, 2014.

Já a seguinte obra oferece uma introdução prática e acessível a estruturas de dados e algoritmos em Python, com foco em aplicações do mundo real. Para hash tables, explica como funcionam (incluindo funções de hash e resolução de colisões) e sua implementação em Python, com exemplos claros usando dicionários. Para heaps, cobre a estrutura de heap binário, sua implementação para filas de prioridade e o uso do módulo heapq para algoritmos como heapsort. Inclui exercícios práticos e explicações otimizadas para Python.

WENGROW, J. *A common-sense guide to data structures and algorithms in Python*. Volume 1. Raleigh: Pragmatic Bookshelf, 2024.

Vamos fazer um exemplo prático. Os sistemas financeiros modernos operam em um ambiente regido por fluxos de dados contínuos, nos quais preços de ações, contratos futuros, pares cambiais e criptomoedas sofrem variações em frações de segundo. Esse ecossistema, sustentado por bolsas globais e plataformas de negociação eletrônica, exige que instituições identifiquem oportunidades de arbitragem,

assimetrias de preço e sinais de liquidez quase instantaneamente. Para que algoritmos de trading, mesas de tesouraria e analistas de risco consigam reagir a oscilações repentinas, torna-se indispensável uma estrutura de dados que mantenha, em tempo real, os ativos mais relevantes, ordenados por métricas de interesse. Heaps revelam-se particularmente adequados a essa tarefa, pois armazenam elementos de tal forma que a extração do maior ou do menor valor ocorre em tempo O(logn), mesmo quando novos preços chegam à alta frequência.

A organização dos dados de mercado em uma fila de prioridade proporciona dois benefícios essenciais. Primeiro, permite que estratégias de execução busquem de forma imediata o instrumento com maior variação percentuais em determinado intervalo, disparando ordens de compra ou venda sem percorrer todo o universo de ativos. Segundo, possibilita remover valores obsoletos conforme novas cotações surgem, preservando apenas informações recentes que refletem o estado atual do mercado. Esse mecanismo de substituição contínua garante que o algoritmo mantenha foco nas oportunidades vigentes, reduzindo latência operacional. Em um cenário de alta volatilidade, deixar de responder a uma anomalia de preço por alguns milissegundos pode representar perda financeira ou risco de arbitragem inversa explorada por concorrentes.

A importância de organizar dados de mercado por meio de heaps ganhou destaque com a difusão de sistemas de negociação de baixa latência e do crescimento do volume de dados gerados por micro-ofertas em livros de ordens. Plataformas empregam heaps tanto dentro de cache-servers em memória, que filtram sinais para modelos de previsão, quanto em bancos de dados especializados, que guardam as cotações mais extremas para auditoria posterior. Ao entender o funcionamento de um heap binário e de seu parente, o Heap Sort, desenvolvedores podem projetar componentes que se integram a essas pipelines, garantindo que operações de inserção, atualização e retirada preservem a ordem de prioridade e não degradem o throughput do sistema.

Exemplo de aplicação

Exemplo 4 – Sistemas financeiros

Uma corretora eletrônica monitora 4.000 ativos listados em diferentes bolsas. O motor de análise interna deve manter, durante o pregão, um ranking dos 20 papéis com maior variação percentual de preço nos últimos quatro minutos.

A política de risco estabelece que, a cada 120 milissegundos, as estratégias de arbitragem consultem esse ranking para avaliar ordens imediatas. As regras operacionais fixam que, toda vez que o preço de um ativo chega, o sistema deve atualizar seu valor, recalcular a variação percentual em relação ao preço de abertura e, caso a ação figure entre as vinte maiores altas ou quedas, posicioná-la corretamente na fila de prioridade. Ao mesmo tempo, ativos que deixarem de pertencer a esse grupo devem ser descartados para que o heap mantenha tamanho fixo.

O código, portanto, precisa inserir e retirar elementos eficientemente, acessando o topo em tempo constante, atualizando posições em O(logn) e fornecendo snapshot ordenado sem interromper o fluxo de entrada.

O exercício usa o módulo `heapq`, biblioteca padrão do Python implementada em C, que oferece operações de empurrar (`heappush`), retirar (`heappop`) e substituir (`heappushpop`) em listas que representam heaps binários. Como esse módulo mantém a menor chave no topo, o exemplo emprega chaves negativas quando deseja priorizar maiores variações, convertendo o minheap em maxheap lógico. Cada elemento guardado no heap contém a variação percentual, um carimbo temporal e o ticker do ativo. A presença do timestamp impede que cotações atrasadas sobreponham valores recentes durante atualização.

Para garantir atualização sem percorrer todo o heap, a estrutura associa a cada ticker seu último registro em um dicionário, permitindo acesso direto ao elemento antigo. Como `heapq` não suporta redução de chave nativamente, o código adota a estratégia de marcação preguiçosa: ele insere o novo valor e mantém um campo booleano indicando se aquele registro está vivo. Quando um `heappop` encontra um registro vencido, simplesmente descarta-o e segue até o próximo válido. Esse padrão replica práticas de sistemas de prioridade em nuvem, em que remoções físicas imediatas podem ser caras em presença de concorrência.

A janela móvel de quatro minutos é controlada por uma `deque` que armazena tuplas (`timestamp`, `ticker`, `preço_abertura`). Ao chegar uma nova cotação, o algoritmo primeiramente remove, da frente da `deque`, registros cujo `timestamp` seja mais antigo que 240 segundos; quando isso ocorre, ele verifica se aquele ativo possuía referência viva no heap e, caso a variação tenha se alterado significativamente, marca-o como expirado, deixando que a estratégia de limpeza preguiçosa descarte o item na próxima extração. A contagem de registros vivos mantém-se assim limitada, garantindo consumo de memória previsível. A seguir, consta o código-fonte da solução:

```
from __future__ import annotations
import heapq, random, time
from collections import deque
from dataclasses import dataclass
from typing import Dict, Tuple, List

# -------- Parâmetros de janela e ranking --------
JANELA_SEG = 240        # quatro minutos
TAMANHO_RANKING = 20
INTERVALO_COTACAO_MS = 120

@dataclass
class RegistroHeap:
    variacao: float     # variação percentual
    ts: float       # timestamp da cotação
    ticker: str     # símbolo do ativo
    vivo: bool      # flag para marcação preguiçosa

class RankingMercado:
    def __init__(self):
        self.heap: List[Tuple[float, int, RegistroHeap]] = []
        self.mapa: Dict[str, RegistroHeap] = {}
        self.deque_precos: deque[Tuple[float, str, float]] = deque()
        self.contador = 0
```

```
    def _limpar_expirados(self, agora: float):
        while self.deque_precos and agora - self.deque_precos[0][0] > JANELA_
SEG:
            _, ticker, _ = self.deque_precos.popleft()
            antigo = self.mapa.get(ticker)
            if antigo:
                antigo.vivo = False

    def _topo_valido(self):
        while self.heap and not self.heap[0][2].vivo:
            heapq.heappop(self.heap)
        return self.heap[0][2] if self.heap else None

    def atualizar_cotacao(self, ticker: str, preco_abertura: float, preco_atual:
float):
        agora = time.time()
        self._limpar_expirados(agora)
        variacao = (preco_atual - preco_abertura) / preco_abertura * 100
        reg = RegistroHeap(variacao=variacao, ts=agora, ticker=ticker,
vivo=True)
        self.deque_precos.append((agora, ticker, preco_abertura))
        self.mapa[ticker] = reg
        if len(self.heap) < TAMANHO_RANKING:
            heapq.heappush(self.heap, (-variacao, self.contador, reg))
            self.contador += 1
        else:
            topo = self._topo_valido()
            if topo and variacao > topo.variacao:
                heapq.heappush(self.heap, (-variacao, self.contador, reg))
                self.contador += 1
                topo.vivo = False

    def snapshot(self) -> List[Tuple[str, float]]:
        válido = []
        copia = self.heap.copy()
        while copia and len(válido) < TAMANHO_RANKING:
            _, _, reg = heapq.heappop(copia)
            if reg.vivo:
                válido.append((reg.ticker, reg.variacao))
        válido.sort(key=lambda x: x[1], reverse=True)
        return válido

# --------- Demonstração simplificada ----------
def principal():
    random.seed(3)
    ranking = RankingMercado()
    tickers = [f"ACAO{i:04d}" for i in range(1, 4001)]
    precos_abertura = {t: random.uniform(10, 300) for t in tickers}
    inicio = time.time()

    for passo in range(1000):      # mil ciclos de cotações
        t = random.choice(tickers)
        preco_open = precos_abertura[t]
        preco_now = preco_open * random.uniform(0.95, 1.05)
        ranking.atualizar_cotacao(t, preco_open, preco_now)
        if passo % 200 == 0:       # imprime a cada 200 atualizações
```

```
                top10 = ranking.snapshot()[:10]
                print(f"\nSnapshot passo {passo} (top 10):")
                for sym, var in top10:
                    print(f"{sym}: {var:+.2f}%")
            time.sleep(INTERVALO_COTACAO_MS / 1000) # atraso artificial
    tempo_exec = time.time() - inicio
    print(f"\nExecução de demonstração concluída em {tempo_exec:.1f}s.")

if __name__ == "__main__":
        principal()
```

O quadro 22 apresenta as funções do Python utilizadas no código-fonte e suas respectivas descrições.

Quadro 22 – Funções usadas no código-fonte do quarto exercício prático

Função ou construção	Explicação de uso
import heapq	Módulo que fornece uma implementação de fila de prioridade (heap), usada aqui para manter os ativos com maiores variações em ordem eficiente
heapq.heappush(heap, elemento)	Insere um elemento no heap, mantendo a ordenação. Aqui, usa-se o valor negativo da variação para simular um max-heap
heapq.heappop(heap)	Remove e retorna o menor elemento do heap (no caso, o de menor variação negativa, ou seja, maior variação positiva)
heap com tuplas ordenadas (-variacao, contador, objeto)	A ordenação do heap é feita com base em tuplas, priorizando primeiro a maior variação (com sinal invertido) e depois o valor de contador como critério de desempate
flag vivo para invalidação preguiçosa	Em vez de remover entradas expiradas diretamente do heap (ineficiente), marca-se o registro como inativo e ignora-se na próxima leitura
cópia do heap com heap.copy() para leitura	Ao tirar o snapshot, evita modificar o heap original copiando-o para uma variável auxiliar antes de processar os dados visíveis
time.sleep(ms / 1000)	Converte milissegundos para segundos para aplicar um atraso artificial entre as cotações (simulando streaming de dados)

O código Python apresentado implementa um sistema de ranking de mercado financeiro que rastreia variações percentuais de preços de ativos em uma janela de tempo fixa, utilizando estruturas de dados otimizadas para eficiência. Ele emprega uma combinação de heap, `deque` e dicionário, com técnicas como marcação preguiçosa para gerenciar atualizações frequentes de cotações.

O programa começa com importações que habilitam anotações de tipo modernas e fornecem as ferramentas necessárias: `heapq` para manipulação de heaps, `random` para simulações, `time` para timestamps, `deque` para uma fila de dupla extremidade, `dataclasses` para criar classes de dados e `typing` para anotações de tipo. Três constantes globais definem parâmetros do sistema: `JANELA_SEG` (`240` segundos ou quatro minutos) determina o período de validade das cotações, `TAMANHO_RANKING` (`20`) limita o número de ativos no ranking e `INTERVALO_COTACAO_MS` (`120` milissegundos) controla a frequência das atualizações na demonstração.

A classe `RegistroHeap`, definida com `@dataclass`, encapsula os dados de um ativo: `variacao` (variação percentual do preço), `ts` (timestamp da cotação), `ticker` (símbolo do ativo) e `vivo` (flag booleana para marcação preguiçosa). Essa flag é usada para indicar se um registro ainda é válido, evitando remoções imediatas de elementos expirados no heap, o que seria custoso.

A classe principal, `RankingMercado`, gerencia o ranking dinâmico. No construtor, inicializa três estruturas: um `heap` (lista que armazena tuplas com variação negativa, contador e registro), um `mapa` (dicionário que mapeia de tickers a registros) e uma `deque_precos` (fila que armazena tuplas com timestamp, ticker e preço de abertura). O atributo contador garante unicidade nas entradas do heap, evitando ambiguidades em comparações.

O método `_limpar_expirados` remove cotações antigas da `deque_precos` quando o tempo decorrido excede `JANELA_SEG`. Para cada cotação removida, marca o registro correspondente no mapa como não vivo (`vivo = False`), adiando a remoção do heap para quando necessário, uma técnica de marcação preguiçosa que otimiza o desempenho. O método `_topo_valido` garante que o elemento no topo do heap seja válido, removendo elementos não vivos até encontrar um registro válido ou esvaziar o heap.

O método `atualizar_cotacao` é o núcleo do sistema. Ele recebe o ticker, preço de abertura e preço atual, calcula a variação percentual e cria um `RegistroHeap`. Após chamar `_limpar_expirados` para manter a janela temporal, adiciona a nova cotação à `deque_precos` e atualiza o mapa. Se o heap tiver menos de `TAMANHO_RANKING` elementos, insere o novo registro diretamente; caso contrário, compara a variação do novo registro com o topo válido do heap. Se a nova variação for maior, insere o registro e marca o topo como não vivo, mantendo o heap com os maiores valores de variação.

O método snapshot retorna uma lista ordenada dos registros válidos no heap, limitada a `TAMANHO_RANKING`. Ele cria uma cópia do heap, extrai registros vivos, ordena-os por variação em ordem decrescente e retorna uma lista de tuplas com ticker e variação. Essa abordagem evita modificar o heap original, garantindo consistência.

A função principal simula o sistema. Inicializa um `RankingMercado`, gera 4.000 tickers fictícios com preços de abertura aleatórios entre 10 e 300, e executa 1.000 ciclos de atualizações. Em cada ciclo, escolhe um ticker aleatoriamente, calcula um preço atual com variação de até ±5% e atualiza o ranking. A cada 200 ciclos, exibe os 10 primeiros do ranking via snapshot. Um atraso artificial de 120 milissegundos simula a frequência de cotações. Ao final, reporta o tempo total de execução.

O código é eficiente devido ao uso de um heap para manter as maiores variações, marcação preguiçosa para evitar remoções frequentes e `deque` para gerenciar a janela temporal. A escolha de estruturas como `deque` (O(1) para inserção e remoção) e `heapq` (O(log n) para inserção e remoção) reflete a preocupação com desempenho em um cenário de alta frequência. A simulação demonstra a capacidade do sistema de processar milhares de atualizações rapidamente, com o ranking refletindo as maiores variações em tempo real.

A seguir, constam sugestões de melhoria para que você implemente no futuro.

- O código poderia integrar-se a websockets que distribuem cotações em tempo real, substituindo o gerador aleatório.

- Uma evolução natural envolveria múltiplos heaps: um para maiores altas, outro para maiores quedas e ainda um min-heap por setor, permitindo diversificação de alertas.

- Em ambientes de múltiplas instâncias, seria pertinente compartilhar snapshots por meio de Redis ou Apache Kafka, empregando serialização binária com Protobuf para reduzir latência de rede. Também seria interessante substituir a deque por uma estrutura de anel em Cython, diminuindo overhead de gerenciamento de objetos Python. Por fim, adicionar cálculo de desvio padrão por ativo e emprego de filtros de Kalman aproximaria o exercício de sistemas de trading quantitativo que não apenas reagem, mas antecipam reversões de tendência.

Resumo

Nesta unidade, examinamos, de maneira abrangente, como técnicas clássicas e estruturas avançadas de dados são postas em prática com Python para tornar a busca, a ordenação e o armazenamento de informações mais eficientes. Partimos dos algoritmos de pesquisa mais elementares, explicando que a busca linear percorre sequencialmente os elementos de uma coleção e apresenta custo proporcional ao tamanho do conjunto, ao passo que a busca binária explora coleções previamente ordenadas e reduz o espaço de procura pela metade a cada passo, alcançando complexidade logarítmica. Essa discussão foi estendida às árvores binárias de busca, nas quais cada comparação conduz o percurso à subárvore adequada; contudo, vimos que inserções em ordem podem degenerar a árvore em um encadeamento linear, motivo pelo qual se introduzem as árvores balanceadas.

As AVL preservam o equilíbrio por meio de rotações rigorosas sempre que o fator de balanceamento excede uma unidade, ao passo que as Red-Black atenuam a rigidez e recorrem a recolorações e rotações menos frequentes, mantendo a altura no limite de duas vezes o mínimo necessário. Em ambos os casos, as operações de inserção, remoção e busca permanecem em tempo O(log n), garantindo desempenho estável, razão pela qual motores de banco de dados, sistemas de arquivos e bibliotecas empregam versões dessas árvores como índices internos.

Na sequência, aprendemos sobre estruturas de dados avançadas. As tabelas hash são apresentadas como vetores de slots nos quais uma função hash transforma chaves em índices, oferecendo inserções e pesquisas em tempo médio constante. São descritos detalhes da implementação do dicionário do Python, que adota endereçamento aberto com sondagem e redimensionamento dinâmico quando a ocupação ultrapassa um limiar, além de empregar randomização de hash para mitigar ataques de colisão. Em seguida, apresentamos os heaps binários, árvores completas armazenadas em arranjos contínuos que mantêm a propriedade de dominância entre pais e filhos, facilitando o acesso em tempo constante ao menor ou maior elemento. A partir desse conceito derivamos o Heap Sort, que constrói um heap máximo e efetua extrações sucessivas para ordenar o vetor in-place em O(n log n) independentemente da disposição inicial dos dados.

Cada técnica foi contextualizada em cenários reais: catálogos de e-commerce combinam busca linear em caches quentes com busca binária em acervos completos; sistemas de bancos de dados em nuvem dependem

de árvores balanceadas para preservar latência sob cargas imprevisíveis; centros de operação de segurança utilizam tabelas de dispersão para indexar assinaturas criptográficas e detectar anomalias; e plataformas de trading de alta frequência recorrem a heaps para priorizar ativos de maior variação em fluxos de mercado.

Exercícios

Questão 1. (Instituto Consulplan 2024, adaptada) Considere o vetor ordenado `V = [3, 8, 15, 19, 24, 30, 42]`. Usando o algoritmo de pesquisa linear, qual é o número de comparações realizadas para encontrar o elemento 24?

A) 3.

B) 4.

C) 5.

D) 6.

E) 7.

Resposta correta: alternativa C.

Análise da questão

A pesquisa linear percorre cada posição de uma coleção até encontrar o elemento desejado ou até esgotar o conjunto. Durante esse processo, a comparação ocorre item a item.

No vetor V do enunciado, o algoritmo percorrerá o vetor sequencialmente, a partir do índice 0, comparando cada elemento com o valor buscado. O processo se dará da forma a seguir.

1ª comparação: 3 == 24 (F)

2ª comparação: 8 == 24 (F)

3ª comparação: 15 == 24 (F)

4ª comparação: 19 == 24 (F)

5ª comparação: 24 == 24 (V)

Logo, foram necessárias cinco comparações para encontrar o elemento desejado.

Questão 2. (Iades 2024, adaptada) No que se refere ao uso de tabelas hash para armazenamento de informação, avalie as afirmativas a seguir.

I – A busca por um elemento em uma tabela hash tem complexidade de tempo O(log n).
II – É impossível garantir que uma tabela hash seja completamente livre de colisões de chaves.

III – Para maior eficiência, recomenda-se a inserção de elementos de maneira ordenada em uma tabela hash.

É correto o que se afirma em:

A) I, apenas.

B) II, apenas.

C) I e II, apenas.

D) II e III, apenas.

E) I, II e III.

Resposta correta: alternativa B.

Análise das afirmativas

I – Afirmativa incorreta.

Justificativa: em condições ideais, ou seja, na ausência de colisões, a busca em uma tabela hash tem complexidade de tempo O(1). Não há condição em que uma tabela hash apresente complexidade O(log n).

II – Afirmativa correta.

Justificativa: a menos que o conjunto de chaves seja fixo e conhecido e que uma função hash perfeita seja criada, colisões de chaves são inevitáveis. Mesmo com funções hash bem projetadas, colisões podem ocorrer.

III – Afirmativa incorreta.

Justificativa: a ordem de inserção não afeta a eficiência, pois a localização depende exclusivamente da função hash.