EAD

EDUCAÇÃO A DISTÂNCIA

UNIDADE III

Algoritmos e Estrutura de Dados em Python

Prof. MSc. Tarcísio Peres

Conteúdo da Unidade III

- Algoritmos de Pesquisa
- Pesquisa linear e binária: funcionamento e eficiência
- Pesquisa em árvores: busca binária e introdução a árvores balanceadas (AVL e Red-Black)
- Estruturas de Dados Avançadas
- Hash Tables: conceito, funções hash e resolução de colisões
- Algoritmos de Ordenação
- Heaps: heaps binários e introdução ao Heap Sort

Fundamentos da pesquisa linear

- Pesquisa linear examina cada elemento de uma coleção sequencialmente até localizar o alvo ou esgotar os registros disponíveis.
- Cada comparação é feita individualmente, resultando em tempo proporcional ao número total de itens verificados.
- Em cenários desfavoráveis, o elemento procurado pode estar na última posição, levando à análise de todos os registros.
- A complexidade temporal da pesquisa linear é O(n), refletindo o custo total em função do tamanho do conjunto.
    - O consumo de memória permanece estável durante a execução, já que são utilizadas apenas variáveis de controle simples.
    - Estruturas não ordenadas ou coleções pequenas favorecem o uso da pesquisa linear devido à sua simplicidade operacional.

Características e aplicações da pesquisa linear

- O crescimento do tempo de resposta se torna notável conforme o volume de dados aumenta, limitando a pesquisa linear a conjuntos menores.
- A simplicidade do método torna-o atraente em situações nas quais a ordenação não é viável ou a coleção muda frequentemente.
- Aplicações práticas incluem buscas em listas não classificadas ou verificação rápida de pequenos catálogos de produtos.
- Como não depende de estrutura prévia, pode ser implementada em qualquer linguagem e contexto computacional.
    - O método não exige preparação do conjunto antes da consulta, diferentemente de abordagens que exploram ordenação.
    - A previsibilidade do desempenho é uma vantagem, pois o custo é sempre proporcional ao tamanho da coleção analisada.

Fundamentos da pesquisa binária

- Pesquisa binária só pode ser utilizada em coleções previamente ordenadas, aproveitando o conhecimento da disposição dos elementos.
- O algoritmo seleciona repetidamente o elemento central e elimina metade dos casos restantes a cada iteração de busca.
- Cada comparação permite descartar uma grande quantidade de elementos, tornando a busca significativamente mais rápida.
- O processo pode ser implementado tanto de maneira recursiva quanto iterativa, conforme as necessidades do sistema.
    - O número de comparações cresce de maneira logarítmica em relação ao número total de registros avaliados.
    - A complexidade temporal $O(\log n)$ caracteriza a eficiência da pesquisa binária em grandes coleções de dados ordenados.

Eficiência e requisitos da pesquisa binária

- Para operar, a pesquisa binária exige um custo inicial: a ordenação da coleção, etapa que pode ser computacionalmente custosa.
- Em sistemas nos quais diversas buscas subsequentes ocorrem, o investimento inicial na ordenação é rapidamente compensado.
- O consumo de memória permanece baixo, pois o algoritmo utiliza apenas variáveis de controle para delimitar os extremos ativos.
- A pesquisa binária é amplamente empregada em estruturas como arrays ordenados, bases de dados e índices de pesquisa.
    - O processo elimina, a cada iteração, metade dos elementos remanescentes, reduzindo drasticamente o número de verificações.
    - A confiabilidade e velocidade em consultas repetidas destacam a pesquisa binária como padrão para buscas em grandes catálogos.

Comparação entre pesquisas linear e binária

- A pesquisa linear apresenta desempenho consistente, independentemente da organização dos elementos no conjunto analisado.
- Em contrapartida, a pesquisa binária requer que a coleção seja previamente ordenada, mas entrega resposta muito mais rápida.
- Em ambientes de dados que sofrem constantes alterações, o custo de ordenação pode reduzir o benefício do método binário.
- Quando o conjunto permanece estático e há muitas consultas, a pesquisa binária torna-se claramente preferível.
    - A escolha do algoritmo ideal depende dos padrões de acesso, frequência de inserções e tamanho do conjunto de dados.
    - Avaliar a dinâmica do conjunto e os requisitos de desempenho orienta a decisão entre varredura sequencial ou busca por divisão.

E-commerce: contexto prático para buscas

- O comércio eletrônico expandiu o varejo global ao operar catálogos com milhões de produtos acessíveis a qualquer momento.
- Plataformas gerenciam descrições detalhadas de itens, incluindo preço, estoque, avaliações, cores e metadados logísticos.
- O crescimento de Big Data intensificou a necessidade de localizar rapidamente itens em meio a grandes volumes de registros.
- A experiência do usuário e as taxas de conversão dependem de respostas ágeis na apresentação de produtos e recomendações.
    - Sistemas eficientes de busca tornam-se críticos para exibir resultados em tempo real e alimentar sugestões personalizadas.
    - A performance das operações de busca impacta diretamente a satisfação do cliente e a competitividade da empresa.

Busca linear e binária no contexto do comércio eletrônico

- Busca linear examina todos os registros até localizar o produto, sendo útil em listas pequenas ou ainda não classificadas.
- Busca binária, utilizada em catálogos ordenados, reduz pela metade o espaço de busca a cada comparação realizada.
- Em grandes bases de dados, a pesquisa binária alcança o item em poucas operações, acelerando o atendimento ao usuário.
- Manter listas ordenadas agiliza buscas, mas pode encarecer operações de inserção e atualização de produtos.
    - O padrão de acesso e o acordo de nível de serviço (SLA) determinam qual abordagem será priorizada em produção.
    - Ambientes de Big Data valorizam soluções que conciliam rapidez de busca com viabilidade de atualização dos registros.

Exercício e-commerce

```
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
```

Exercício e-commerce

```
@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)]
```

Exercício e-commerce

```
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
```

Exercício e-commerce

```
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}
```

Exercício e-commerce

```
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}
```

Exercício e-commerce

```
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)]
```

Exercício e-commerce

```
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
```

Exercício e-commerce

```
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]
```

Exercício e-commerce

```
# 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)
```

Interatividade

Considerando os algoritmos de pesquisa linear e binária, assinale a alternativa correta a respeito das suas características e aplicação em sistemas de comércio eletrônico.

a) A pesquisa linear apresenta complexidade logarítmica e é preferida para grandes catálogos ordenados devido ao baixo custo de inserção de novos elementos.

b) A pesquisa binária pode ser aplicada a qualquer coleção, independentemente de sua ordenação, e exige mais memória que a pesquisa linear.

c) A pesquisa linear examina todos os elementos sequencialmente, mantendo o consumo de memória constante e sendo mais indicada para listas pequenas e não ordenadas.

d) A pesquisa binária elimina a necessidade de ordenação do conjunto, tornando-se vantajosa em contextos de dados frequentemente atualizados.

e) Em ambientes de e-commerce, a combinação dos métodos de busca é inviável, pois aumenta excessivamente o tempo de resposta para o usuário.

Resposta

Considerando os algoritmos de pesquisa linear e binária, assinale a alternativa correta a respeito das suas características e aplicação em sistemas de comércio eletrônico.

a) A pesquisa linear apresenta complexidade logarítmica e é preferida para grandes catálogos ordenados devido ao baixo custo de inserção de novos elementos.

b) A pesquisa binária pode ser aplicada a qualquer coleção, independentemente de sua ordenação, e exige mais memória que a pesquisa linear.

c) A pesquisa linear examina todos os elementos sequencialmente, mantendo o consumo de memória constante e sendo mais indicada para listas pequenas e não ordenadas.

d) A pesquisa binária elimina a necessidade de ordenação do conjunto, tornando-se vantajosa em contextos de dados frequentemente atualizados.

e) Em ambientes de e-commerce, a combinação dos métodos de busca é inviável, pois aumenta excessivamente o tempo de resposta para o usuário.

Introdução à busca em árvores

- A busca em árvores explora a estrutura hierárquica dos dados, reduzindo o número de comparações necessárias para localizar um valor em coleções grandes.
- Uma árvore binária de busca organiza cada nó de modo que os valores à esquerda sejam sempre menores e os à direita, maiores do que o nó central.
- O processo de pesquisa começa na raiz e segue por comparações sucessivas, avançando para a subárvore esquerda ou direita conforme o valor procurado.
- A eficiência desse método depende da altura da árvore: em árvores equilibradas, o número máximo de comparações é proporcional ao logaritmo do número de nós.
    - Inserções ordenadas podem degenerar a árvore binária de busca em uma estrutura semelhante a uma lista encadeada, reduzindo a eficiência para O(n).
    - O uso de árvores balanceadas torna-se fundamental para garantir eficiência logarítmica em operações, mesmo após muitas inserções ou remoções.

Funcionamento da árvore binária de busca

- Cada nó de uma árvore binária de busca mantém uma relação de ordem com seus filhos, organizando os dados de forma sistemática para facilitar consultas.
- A busca em árvores binárias segue o mesmo princípio da divisão binária, mas aplica a cada nó a escolha de caminho, tornando o processo dinâmico e eficiente.
- Em condições ideais, quando os ramos possuem altura semelhante, a árvore binária de busca garante operações rápidas e eficientes.
- O desempenho, porém, pode ser comprometido se a estrutura for alimentada apenas por inserções ordenadas, causando desequilíbrio.
    - Nessas situações, a árvore pode perder sua principal vantagem de busca rápida, aumentando o tempo necessário para localizar elementos.
    - Para evitar a degeneração, técnicas de balanceamento automático foram desenvolvidas, aprimorando a robustez das operações de busca.

Motivação para árvores balanceadas

- Árvores balanceadas foram criadas para manter alturas similares entre diferentes ramos, garantindo eficiência mesmo com inserções e remoções frequentes.
- O balanceamento previne que o tempo de busca cresça de modo linear, sustentando operações dentro de limites logarítmicos.
- A manutenção desse equilíbrio exige monitoramento constante do fator de balanceamento a cada alteração na árvore.
- Nas árvores balanceadas, rotações são aplicadas automaticamente sempre que a diferença de altura entre subárvores excede o permitido.
    - Essa manutenção rigorosa evita o surgimento de “galhos” longos, que comprometeriam a eficiência do acesso e atualização dos dados.
    - O conceito de balanceamento automático revolucionou o uso de árvores para indexação em grandes bancos de dados e sistemas críticos.

Árvores AVL: conceito e funcionamento

- As árvores AVL são o primeiro tipo amplamente utilizado de árvore binária de busca balanceada, surgidas em 1962 por Adelson-Velsky e Landis.
- Cada nó da árvore AVL armazena o fator de balanceamento, definido pela diferença entre as alturas das subárvores esquerda e direita.
- Quando essa diferença ultrapassa uma unidade, a árvore executa rotações simples ou duplas para restabelecer o equilíbrio.
- O rigor no controle de alturas mantém a eficiência logarítmica em operações como inserção, remoção e busca, mesmo em grandes conjuntos.
    - A rigidez no balanceamento garante árvores de altura mínima próxima ao ideal, evitando degradação do desempenho ao longo do tempo.
    - O custo adicional das rotações é geralmente compensado pelo ganho em previsibilidade e estabilidade das operações.

Importância do fator de balanceamento nas AVL

- O fator de balanceamento em árvores AVL limita a diferença de altura entre subárvores a no máximo um, garantindo simetria estrutural.
- Esse mecanismo permite que a árvore responda rapidamente a operações de inserção e remoção sem perder desempenho.
- Sempre que o fator de balanceamento sai do intervalo permitido, são realizadas rotações que corrigem o desequilíbrio imediatamente.
- Manter esse equilíbrio exige atualizações constantes durante operações que alteram a estrutura da árvore.
    - Apesar do custo das rotações, a operação permanece eficiente em grandes volumes de dados, sendo amplamente utilizada em bancos de dados.
    - O uso do fator de balanceamento diferencia as árvores AVL de outras soluções menos rigorosas de organização.

Rotações em árvores AVL

- Rotações são operações locais que reorganizam os nós para restabelecer o equilíbrio estrutural sempre que o fator de balanceamento é violado.
- Existem rotações simples (direita ou esquerda) e rotações duplas (direita-esquerda ou esquerda-direita), dependendo do caso de desequilíbrio detectado.
- O ajuste da árvore ocorre de modo eficiente, afetando apenas uma pequena parte da estrutura, sem necessidade de reconstrução total.
- O número de rotações é proporcional ao grau de desequilíbrio, sendo geralmente baixo em cenários reais de uso.
    - O código das árvores AVL implementa funções específicas para detectar desequilíbrio e acionar a rotação apropriada.
    - O controle rigoroso das rotações faz das AVL uma referência para aplicações que exigem alta estabilidade e desempenho previsível.

Red-Black Tree: fundamentos e vantagens

- A árvore Red-Black surgiu para reduzir o número de rotações necessárias ao manter o equilíbrio, especialmente em ambientes com atualizações intensas.
- Cada nó recebe uma cor (vermelho ou preto) e a estrutura impõe restrições específicas quanto à distribuição das cores ao longo dos caminhos.
- Entre as regras: cada caminho da raiz até as folhas tem o mesmo número de nós negros e nenhum nó vermelho possui filho vermelho.
- Essas regras garantem que a altura máxima nunca ultrapasse duas vezes o valor mínimo possível, mantendo a complexidade O(log n).
    - A flexibilidade estrutural reduz a necessidade de correções frequentes, tornando a Red-Black ideal para sistemas com fluxo intenso de alterações.
    - Por sacrificar simetria perfeita em troca de menos rotações, a Red-Black Tree atinge desempenho superior em bancos de dados e sistemas de arquivos.

Comparação entre AVL e Red-Black

- AVL privilegia rigor simétrico, mantendo alturas quase ideais, enquanto Red-Black aceita mais assimetria para reduzir rotações.
- Em aplicações com grandes volumes e fluxos intensos de atualização, Red-Black costuma ser preferida pelo menor custo de manutenção.
- Ambas garantem complexidade O(log n) para buscas, inserções e remoções, mas diferem no comportamento frente a cargas variáveis.
- AVL é indicada para cenários em que operações de leitura são predominantes e previsibilidade é requisito central.
    - Red-Black apresenta melhor desempenho em ambientes dinâmicos, nos quais inserções e exclusões ocorrem com alta frequência.
    - A escolha entre elas depende do perfil de uso e das necessidades específicas de cada aplicação ou serviço.

Aplicações práticas de árvores balanceadas

- Bancos de dados modernos utilizam árvores balanceadas para indexar e recuperar dados rapidamente, assegurando baixa latência em consultas.
- Em sistemas de nuvem, a eficiência das buscas é vital para manter SLAs rigorosos e evitar gargalos em consultas a grandes tabelas.
- Estruturas como AVL e Red-Black são fundamentais em workloads multitenant, onde dados de múltiplos usuários competem por acesso eficiente.
- Ao prevenir degradação para O(n), essas árvores tornam possível escalar serviços que lidam com trilhões de registros simultaneamente.
    - A manutenção do equilíbrio é automatizada, sendo disparada sempre que inserções ou remoções tendem a desbalancear a estrutura.
    - Profissionais que compreendem o funcionamento dessas árvores conseguem diagnosticar saltos de latência e ajustar políticas de índice.

Relação com B-trees e LSM-trees em bancos de dados

- B-trees e LSM-trees são alternativas especializadas para indexação em disco, otimizando operações de leitura e escrita em sistemas de larga escala.
- B-trees mantêm múltiplas chaves por nó, minimizando acessos ao disco e acelerando consultas em bancos relacionais como MySQL e PostgreSQL.
- LSM-trees são projetadas para cargas de escrita intensiva, armazenando dados inicialmente em memória e depois consolidando-os em camadas no disco.
- Em ambas, árvores balanceadas na memória são usadas como estrutura temporária antes de persistir os dados em camadas secundárias.
    - O entendimento do comportamento de árvores AVL auxilia na configuração e otimização de bancos que empregam B-trees ou LSM-trees.
    - Estruturas de indexação eficientes são vitais para evitar varreduras completas e garantir consultas rápidas mesmo sob altos volumes.

Desafios em ambientes de nuvem e indexação

- Em serviços em nuvem, instâncias podem ser reiniciadas ou migradas, exigindo que estruturas de indexação sejam reconstruídas de maneira eficiente.
- Árvores balanceadas na memória RAM precisam se reerguer rapidamente a partir de logs ou caches “quentes” após eventos de falha ou reinicialização.
- A performance das consultas permanece dependente do equilíbrio estrutural, mesmo em ambientes distribuídos e virtualizados.
- Implementar árvores AVL em Python permite simular o impacto de inserções, buscas e remoções em condições semelhantes às de produção.
- Métricas como profundidade máxima e tempo médio de busca ilustram o ganho de desempenho proporcionado pelo balanceamento.

Exemplo: Bancos de Dados e Serviços em Nuvem

```
from __future__ import annotations
import random, time
from dataclasses import dataclass
from typing import Optional, Tuple, Any
                            @dataclass
                            class NoAVL:
                                chave: int
                                valor: Any
                                esquerda: Optional['NoAVL'] = None
                                direita: Optional['NoAVL'] = None
                                altura: int = 1
```

Exemplo: Bancos de Dados e Serviços em Nuvem

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

Exemplo: Bancos de Dados e Serviços em Nuvem

```
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
```

Exemplo: Bancos de Dados e Serviços em Nuvem

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

Exemplo: Bancos de Dados e Serviços em Nuvem

```
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
```

Interatividade

Considere as características das árvores AVL e Red-Black. Em relação às diferenças práticas entre essas duas estruturas de dados, qual das afirmações a seguir está correta?

a) Árvores AVL são menos rigorosas no balanceamento e, por isso, demandam menos rotações que as árvores Red-Black em cargas de atualização intensas.

b) Árvores Red-Black garantem alturas absolutamente mínimas, mesmo após milhares de inserções e remoções sequenciais, superando as AVL em simetria.

c) Árvores AVL mantêm alturas quase ideais, oferecendo buscas previsíveis, porém exigem mais rotações que as Red-Black para manter o equilíbrio.

d) Árvores AVL e Red-Black apresentam comportamento idêntico em todas as operações, não existindo diferenças de desempenho conforme o fluxo de atualizações.

e) Árvores Red-Black não utilizam qualquer critério de balanceamento, apenas regras para ordenação dos elementos, sendo inferiores às árvores binárias simples.

Resposta

Considere as características das árvores AVL e Red-Black. Em relação às diferenças práticas entre essas duas estruturas de dados, qual das afirmações a seguir está correta?

a) Árvores AVL são menos rigorosas no balanceamento e, por isso, demandam menos rotações que as árvores Red-Black em cargas de atualização intensas.

b) Árvores Red-Black garantem alturas absolutamente mínimas, mesmo após milhares de inserções e remoções sequenciais, superando as AVL em simetria.

c) Árvores AVL mantêm alturas quase ideais, oferecendo buscas previsíveis, porém exigem mais rotações que as Red-Black para manter o equilíbrio.

d) Árvores AVL e Red-Black apresentam comportamento idêntico em todas as operações, não existindo diferenças de desempenho conforme o fluxo de atualizações.

e) Árvores Red-Black não utilizam qualquer critério de balanceamento, apenas regras para ordenação dos elementos, sendo inferiores às árvores binárias simples.

Estrutura de uma tabela hash em Python

- A tabela hash em Python organiza pares chave-valor em um vetor de slots, com cada chave transformada em número inteiro pela função hash embutida.
- Essa transformação gera um índice que determina onde o dado será armazenado, permitindo consultas em tempo médio constante.
- O vetor é dimensionado em potências de dois, acelerando cálculos com máscaras de bits durante inserções e buscas.
- Cada elemento é armazenado em posição baseada no resultado numérico da função hash aplicada à chave original.

Estrutura de uma tabela hash em Python

- O tamanho do vetor sendo potência de dois facilita o uso de operações bit a bit para indexação eficiente.
- Funções hash precisam ser determinísticas: chaves iguais produzem sempre o mesmo valor hash.
- Imutabilidade lógica das chaves é obrigatória, evitando erros durante buscas subsequentes. A função hash de objetos definidos pelo usuário deve respeitar a coerência entre igualdade e valor hash.
- Aleatoriedade interna na função hash do Python protege contra ataques baseados em colisões artificiais.

Sondagem aberta e resolução de colisões

- Quando diferentes chaves produzem o mesmo valor hash, a tabela hash em Python utiliza endereçamento aberto com sondagem aritmética. Esse método desloca a busca por incrementos calculados, a partir do hash original e um fator de perturbação, até encontrar uma posição livre ou a chave procurada.
- A sondagem evita o uso de listas encadeadas adicionais, reduzindo o consumo de memória.
- Os incrementos utilizados na sondagem variam conforme os bits do hash, alterando o caminho de busca.
- Esse método melhora a localidade de memória, otimizando o desempenho em relação ao cache da CPU.
    - A busca prossegue até encontrar um slot disponível ou o par chave-valor desejado, mantendo eficiência.
    - O algoritmo de sondagem preserva a integridade dos dados mesmo com ocupação elevada.
    - A ausência de listas auxiliares torna a estrutura mais compacta e ágil para leituras e inserções.

Redimensionamento e reinserção de elementos

- Se a taxa de ocupação da tabela hash ultrapassa cerca de setenta por cento, ocorre redimensionamento automático.
- Uma nova tabela maior é alocada, e todos os pares chave-valor são reinseridos com base nos hashes previamente calculados.
- O redimensionamento distribui o custo elevado da operação ao longo de diversas inserções, mantendo eficiência média.
- Cada chave é realocada conforme seu hash na nova tabela, evitando colisões antigas.
- Essa abordagem garante que a complexidade amortizada das inserções permaneça baixa mesmo após expansão.
    - O redimensionamento é transparente para o usuário, preservando a experiência de uso do dicionário Python.
    - A capacidade do vetor principal cresce em potências de dois, acompanhando o volume de dados armazenados.
    - O procedimento reduz a probabilidade de múltiplas colisões sucessivas, melhorando o desempenho global.

Marcação de slots liberados e busca eficiente

- Ao remover um elemento, a tabela hash marca o slot correspondente como "dummy", sem apagá-lo imediatamente. Esse marcador especial assegura que futuras buscas não sejam interrompidas prematuramente, mantendo a sequência da sondagem.
- O uso de marcadores evita buracos que poderiam interromper a busca antes de alcançar a posição correta.
- A presença de "dummy" permite remoções eficientes sem reestruturação imediata da tabela.
- Slots marcados são reciclados somente após novo redimensionamento, liberando espaço para reutilização.
    - A manutenção de marcadores preserva a consistência da estrutura mesmo após múltiplas exclusões.
    - Esse detalhe técnico impede degradação de desempenho ao longo do tempo com remoções frequentes.
    - A busca permanece confiável e eficiente, mesmo diante de múltiplos slots excluídos recentemente.

Função hash e segurança contra ataques

- O Python utiliza uma chave secreta (sal) ao inicializar o interpretador, técnica chamada hash randomization. Essa medida impede ataques de colisão direcionados, dificultando a antecipação de padrões por invasores.
- O sal garante que o valor hash de uma mesma chave varie entre execuções diferentes do programa.
- Essa abordagem protege aplicações web contra inserção de grandes volumes de chaves maliciosas.
- Ataques por colisão ficam inviáveis, pois o atacante não consegue prever o índice da tabela hash.
    - Cada inicialização do interpretador gera um novo sal, tornando os hashes imprevisíveis.
    - O recurso fortalece a segurança sem comprometer o desempenho das operações.
    - O uso de hash randomization tornou-se padrão após incidentes envolvendo negação de serviço em aplicações web.

Hash tables e cibersegurança na prática

- Tabelas hash desempenham papel central em operações de cibersegurança, como verificação de autenticidade de arquivos e detecção de padrões anômalos. Sua eficiência permite monitorar milhões de registros em tempo real.
- Ataques de ransomware e fraudes exigem mecanismos rápidos para autenticação e resposta imediata.
- Hash tables indexam resumos criptográficos de arquivos, acelerando verificações de integridade.
- Centros de operação de segurança dependem dessas estruturas para rastrear comportamentos suspeitos.
    - Sistemas de monitoramento usam hash tables para identificar logs inéditos ou alterações não autorizadas.
    - A velocidade de acesso das tabelas hash é fundamental para conter ameaças em frações de segundo.
    - A resiliência das hash tables as torna base para prototipagem de novos detectores de ataques.

Hash tables em arquiteturas de microserviços e IoT

- A proliferação de microsserviços e dispositivos IoT elevou a importância das hash tables, permitindo replicação de índices de assinaturas diretamente na memória dos serviços.
- Reduzir a latência de ida e volta a bancos centrais é vital para a proteção em tempo real.
- Cada serviço pode manter cópias de tabelas hash de assinaturas críticas para tomada de decisão rápida.
- Hash tables escritas em Python aceleram o desenvolvimento e prototipagem antes da migração para C ou Rust.
- Replicação local evita sobrecarga de consultas e torna sistemas mais resilientes a falhas de comunicação.
    - Estruturas em memória são especialmente úteis em dispositivos restritos e aplicações distribuídas.
    - O entendimento detalhado da implementação favorece ajustes finos de desempenho em ambientes produtivos.

Exemplo: Cibersegurança

```
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: Cibersegurança

```
@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()
```

Exemplo: Cibersegurança

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

Exemplo: Cibersegurança

```
def principal():
    verificador = TabelaAssinaturas()
    seguro = b"documento_legitimo"
    malicioso = b"macro_malware"
    # registra arquivo confiável conhecido
    verificador.marcar_confiavel(hashlib.sha256(seguro).hexdigest())
```

Interatividade

No contexto do funcionamento interno da tabela hash do Python, assinale a alternativa correta:

a) A tabela hash em Python utiliza listas encadeadas para resolver colisões entre chaves, permitindo acesso constante mesmo em alta ocupação do vetor.

b) O valor hash de uma chave é recalculado a cada consulta, garantindo distribuição dinâmica e impedindo duplicação de pares chave-valor.

c) Para garantir desempenho e segurança, a tabela hash só aceita chaves que sejam números inteiros, evitando colisões causadas por strings ou objetos personalizados.

d) Todas as versões do Python mantêm a ordem de inserção dos elementos em dicionários desde o início da linguagem, independentemente de implementações internas.

e) Durante remoções, o slot correspondente é marcado como “dummy” para evitar que buscas posteriores sejam interrompidas antes de encontrar o elemento correto.

Resposta

No contexto do funcionamento interno da tabela hash do Python, assinale a alternativa correta:

a) A tabela hash em Python utiliza listas encadeadas para resolver colisões entre chaves, permitindo acesso constante mesmo em alta ocupação do vetor.

b) O valor hash de uma chave é recalculado a cada consulta, garantindo distribuição dinâmica e impedindo duplicação de pares chave-valor.

c) Para garantir desempenho e segurança, a tabela hash só aceita chaves que sejam números inteiros, evitando colisões causadas por strings ou objetos personalizados.

d) Todas as versões do Python mantêm a ordem de inserção dos elementos em dicionários desde o início da linguagem, independentemente de implementações internas.

e) Durante remoções, o slot correspondente é marcado como “dummy” para evitar que buscas posteriores sejam interrompidas antes de encontrar o elemento correto.

Estrutura Conceitual do Heap

- É uma estrutura de dados baseada em árvore, organizada de modo que o menor ou maior elemento esteja sempre acessível na raiz, permitindo operações prioritárias rápidas.
- Os nós seguem uma relação de dominância, com o nó pai menor (em heaps mínimos) ou maior (em heaps máximos) que seus filhos diretos, o que facilita remoções e inserções eficientes.
- Essa estrutura é mantida de maneira compacta em um vetor, economizando memória e otimizando o acesso sequencial.
- Inserções no heap envolvem anexar o novo valor ao final do vetor, realizando uma subida até restaurar a propriedade de dominância.
    - Remover o elemento prioritário consiste em substituir a raiz pelo último item e efetuar descidas, preservando a ordem do heap.
    - Por não depender de ponteiros, heaps aproveitam melhor a localidade de cache da CPU, sendo especialmente úteis em operações que exigem acesso constante ao elemento de maior ou menor prioridade.

Propriedade de Dominância e Representação Vetorial

- O heap binário é representado por uma árvore completa armazenada em vetor, onde cada nó obedece à propriedade de dominância: em heaps mínimos, o valor do pai é sempre menor ou igual ao dos filhos; nos máximos, ocorre o inverso.
- Os índices dos filhos de cada nó são calculados aritmeticamente, dispensando ponteiros, o que reduz consumo de memória e simplifica manipulação.
- Inserir um elemento exige anexá-lo ao fim do vetor e realizar operações de subida, em que o valor troca de posição com seus antecessores até encontrar a posição correta.
- Esse procedimento garante tempo $O(\log n)$ devido ao crescimento logarítmico da altura da árvore.
    - A remoção do elemento prioritário, situado na raiz, é realizada trocando-o pelo último elemento e promovendo descidas sucessivas para restaurar a propriedade do heap.
    - O heap binário é amplamente utilizado em filas de prioridade pela sua eficiência de acesso.

Funções Heap no Python e Operações Essenciais

- O módulo heapq da biblioteca padrão Python implementa um min-heap utilizando listas comuns, oferecendo funções como heappush (inserção), heappop (remoção) e heapreplace (substituição direta do topo).
- Qualquer sequência pode ser convertida em heap por meio da função heapify, que realiza descidas a partir da metade inferior do vetor em tempo linear O(n).
- A manutenção do heap é feita inteiramente dentro do próprio vetor, dispensando estruturas auxiliares, o que favorece a eficiência em ambientes restritos em memória.
- A inserção de novos elementos, a remoção do menor valor e a substituição direta do topo são operações fundamentais para a gerência de prioridades.
    - O heapq proporciona ainda desempenho consistente para operações frequentes em contextos que exigem respostas rápidas.

Heap Sort: Princípio e Funcionamento

- O Heap Sort explora a propriedade do heap máximo para ordenar vetores de forma eficiente. Primeiramente, todo o vetor é transformado em heap, colocando o maior valor na raiz.
- O algoritmo então troca a raiz pelo último elemento do vetor não ordenado, reduz o intervalo e executa uma descida para restaurar a propriedade do heap.
- Esse processo repete-se até que todo o vetor esteja ordenado.
- A construção inicial do heap consome tempo $O(n)$ e cada remoção exige $O(\log n)$, totalizando $O(n \log n)$ para toda a ordenação.
- O Heap Sort opera in-place, exigindo apenas memória adicional constante além do vetor original, o que o torna adequado para cenários de memória restrita.
    - Apesar de não ser estável, é amplamente utilizado para dados sem necessidade de preservação da ordem original.

Eficiência Prática do Heap Sort

- O Heap Sort oferece desempenho robusto graças ao padrão de acesso quase sequencial, minimizando realocações volumosas e mantendo o vetor sempre parcialmente ordenado.
- Apesar de possuir complexidade $O(n \log n)$ em todos os casos, suas constantes de tempo podem superar as do Quick Sort em cenários médios devido à maior incidência de trocas.
- Por outro lado, a ausência de requisitos de memória adicional relevante e o desempenho previsível tornam o Heap Sort uma opção atrativa em ambientes nos quais a estabilidade não é uma exigência.
- Essa característica é especialmente útil em aplicações embarcadas e sistemas operacionais, onde previsibilidade e baixo consumo de recursos são prioritários.

Heap como Fila de Prioridade

- O heap binário é amplamente utilizado como estrutura base para filas de prioridade e oferece operações eficientes para acesso, inserção e remoção do elemento mais relevante.
- Essas filas permitem agendar eventos, processar tarefas em ordem de prioridade e realizar buscas pelos k menores ou maiores elementos de um conjunto com custo $O(n + k \log n)$, superando abordagens tradicionais que exigiriam a ordenação completa dos dados.
- O uso de heaps simplifica a manutenção do conjunto em tempo real, com inserções e remoções sendo tratadas de forma ágil e previsível.
- A localidade de memória favorecida pelo armazenamento vetorial potencializa o desempenho em ambientes de alta demanda.

Aplicação em Sistemas Financeiros de Alta Frequência

- Em sistemas financeiros modernos, heaps são essenciais para acompanhar variações de preços em ativos negociados em alta frequência, permitindo identificar oportunidades de arbitragem e sinais de liquidez em frações de segundo.
- A estrutura do heap mantém os ativos mais relevantes acessíveis, com inserções e remoções rápidas mesmo sob fluxos contínuos de dados.
- Algoritmos de trading e mesas de tesouraria dependem do heap para reagir imediatamente a anomalias de preço, otimizando a execução de ordens e mitigando riscos.
- O acesso constante ao maior ou menor valor é vital para preservar a competitividade e evitar prejuízos por atraso na análise.

Filas de Prioridade e Latência Operacional

- A organização dos dados de mercado em filas de prioridade baseadas em heaps traz benefícios práticos para estratégias de execução.
- O heap permite identificar rapidamente o ativo com maior variação percentual, disparando ordens sem necessidade de varredura completa.
- Simultaneamente, valores obsoletos podem ser removidos à medida que novas cotações chegam, mantendo a fila sempre atualizada com informações pertinentes ao momento.
- Essa agilidade é fundamental em ambientes de alta volatilidade, nos quais a demora de milissegundos pode impactar diretamente resultados financeiros e exposição a riscos.

Exemplo: Sistemas Financeiros

```
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
```

Exemplo: Sistemas Financeiros

```python
@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
```

Exemplo: Sistemas Financeiros

```
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
```

Interatividade

Considere as características e operações dos heaps binários. Qual das afirmações a seguir representa corretamente um aspecto fundamental da estrutura de dados heap e de sua aplicação no Heap Sort em Python?

a) O heap binário exige ponteiros explícitos para os filhos, tornando o acesso mais lento, embora preserve a propriedade de dominância.

b) O Heap Sort é um algoritmo estável, pois sempre preserva a ordem relativa de elementos equivalentes ao realizar as operações de descida.

c) O heap binário permite inserir elementos em tempo linear, pois cada operação de subida requer percorrer toda a estrutura.

d) O heap binário, armazenado em vetor, mantém o menor ou maior elemento sempre acessível na raiz, e o Heap Sort utiliza essa propriedade para ordenar vetores in-place com complexidade $O(n \log n)$.

e) A conversão de uma lista comum em heap pelo método heapify consome tempo exponencial, tornando o uso do heap impraticável para grandes conjuntos.

Resposta

Considere as características e operações dos heaps binários. Qual das afirmações a seguir representa corretamente um aspecto fundamental da estrutura de dados heap e de sua aplicação no Heap Sort em Python?

a) O heap binário exige ponteiros explícitos para os filhos, tornando o acesso mais lento, embora preserve a propriedade de dominância.

b) O Heap Sort é um algoritmo estável, pois sempre preserva a ordem relativa de elementos equivalentes ao realizar as operações de descida.

c) O heap binário permite inserir elementos em tempo linear, pois cada operação de subida requer percorrer toda a estrutura.

d) O heap binário, armazenado em vetor, mantém o menor ou maior elemento sempre acessível na raiz, e o Heap Sort utiliza essa propriedade para ordenar vetores in-place com complexidade $O(n \log n)$.

e) A conversão de uma lista comum em heap pelo método heapify consome tempo exponencial, tornando o uso do heap impraticável para grandes conjuntos.

Referências

- AHO, A. V.; HOPCROFT, J. ; ULLMAN, J. *Estruturas de dados e algoritmos*. Porto Alegre: Bookman, 2002.
- ARAÚJO, Gustavo. *Automatize Tarefas Maçantes com Python*. São Paulo: Novatec, 2020.
- BEAZLEY, D.; JONES, B. K. *Python Cookbook*: Recipes for Mastering Python 3. 3. ed. O'Reilly Media, 2013.
- CORMEN, H. *et al*. *Algoritmos*: Teoria e Prática. 4. ed. LTC, 2024.
- FURTADO, André Luiz. Python: Programação para Leigos. Rio de Janeiro: Alta Books, 2021.
- KAMAL, M.; RAWAT, K. *Dynamic Programming for Coding Interviews*. Notion Press, 2017.
- KLEINBERG, J.; TARDOS, ; TARDOS, I. *Algorithm Design*. Addison-Wesley Professional, 2005.

Referências

- KNUTH, D. *The Art of Computer Programming*. 3. ed. Addison-Wesley Professional, v. 2, 1997.
- LUTZ, M. *Learning Python*. 5. ed. O'Reilly Media, 2013.
- MILLER, B. N.; RANUM , L. *Problem Solving with Algorithms and Data Structures Using Python*. Franklin Beedle & Associates.
- NILO, Luiz Eduardo. *Introdução à Programação com Python*. São Paulo: Novatec, 2019.

ATÉ A PRÓXIMA!