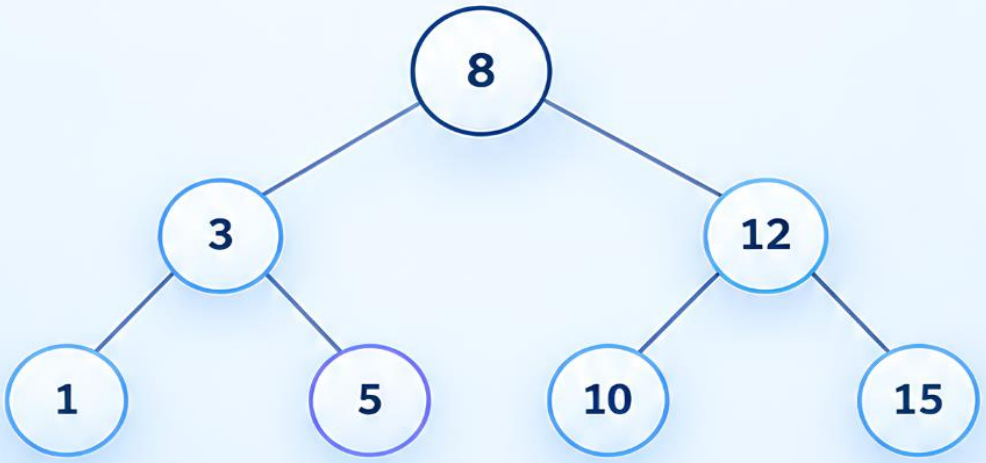
Jose Augusto dos Santos Silva

Estruturas de Dados em Python:

Fundamentos, Implementação e Aplicações




```
lista = []
pilha = []
fila = []
arvore = {}
grafo = {}
```

```
inserir()
remover()
buscar()
percorrer()
```

thesis editora científica

Estruturas de Dados em Python

Fundamentos, Implementação e Aplicações

Jose Augusto dos Santos Silva

Engenheiro de Computação pelo Instituto de Computação da Universidade Federal de Alagoas
Mestre em Informática pelo Programa de Pós-Graduação em Informática do Instituto de Computação da Universidade Federal de Alagoas

Livro didático de Ciência da Computação

2026

2026 - Thesis Editora Científica

Editor Chefe: Felipe Cardoso Rodrigues Vieira

Diagramação, Projeto Gráfico e Design da Capa: Thesis Ed. Cien. e Jose Augusto dos Santos Silva

Revisão: Jose Augusto dos Santos Silva

O manuscrito foi previamente submetido à avaliação cega pelos pares (*blind peer review*), membros do Conselho Editorial desta Editora, tendo sido aprovado para a publicação com base em critérios de neutralidade e imparcialidade acadêmica.

ISBN: 978-65-83199-79-9

Thesis Editora Científica

Teresina – PI – Brasil

contato@thesiseditora.com.br

www.thesiseditora.com.br

Conselho editorial

2026 - Thesis Editora Científica

Editor Chefe: Felipe Cardoso Rodrigues Vieira

Diagramação, Projeto Gráfico e Design da Capa: Thesis Ed. Cien. e Jose Augusto dos Santos Silva

Revisão: Jose Augusto dos Santos Silva

Catalogação na Publicação

Elaborada por Bibliotecária Janaina Ramos – CRB-8/9166

S586e

Silva, Jose Augusto dos Santos

Estruturas de dados em Python: fundamentos, implementação e aplicações / Jose Augusto dos Santos Silva. – Teresina-PI: Thesis Editora Científica, 2026.

Livro em PDF

ISBN 978-65-83199-79-9

1. Estruturas de dados (Computação). 2. Python (Linguagem de programação). I. Silva, Jose Augusto dos Santos. II. Título.

CDD 005.73

Índice para catálogo sistemático

I. Estruturas de dados (Computação)

Thesis Editora Científica

Teresina – PI – Brasil

contato@thesiseditora.com.br

www.thesiseditora.com.br

Dados editoriais

Esta obra foi organizada como livro didático para estudantes de Ciência da Computação, com explicações graduais, exemplos executáveis, atividades de fixação e referências técnicas.

Jose Augusto dos Santos Silva é Engenheiro de Computação pelo Instituto de Computação da Universidade Federal de Alagoas. Também é Mestre em Informática pelo Programa de Pós-Graduação em Informática do Instituto de Computação da Universidade Federal de Alagoas.

Os códigos foram escritos para fins didáticos. O leitor deve executá-los em ambiente Python compatível e adaptar exemplos, nomes e dados ao contexto de seus próprios estudos.

Nota de responsabilidade. Conceitos, exemplos e exercícios devem ser lidos como material educacional. A execução de programas em sistemas reais exige avaliação de segurança, desempenho e manutenção.

Apresentação

Aprender Computação envolve mais do que aprender uma linguagem. É necessário representar problemas, separar decisões, escolher estruturas adequadas, testar hipóteses e explicar por que uma solução funciona. Este livro foi escrito para acompanhar esse percurso em passos curtos, com exemplos que podem ser lidos, executados, modificados e discutidos.

Python é utilizada como ferramenta de expressão porque permite concentrar a atenção na lógica. A linguagem não elimina a necessidade de raciocínio: ela apenas reduz o ruído sintático para que o estudante observe dados, estados, condições, repetições e funções.

O texto alterna explicações, tabelas, diagramas, pequenos programas e exercícios. A recomendação é não tratar os exemplos como respostas para copiar. Primeiro leia o problema, formule uma previsão, escreva uma solução mínima e só então compare o resultado com o programa apresentado.

Como usar este livro

- Leia a explicação antes de executar o código; identifique entradas, processamento e saídas.
- Digite ou reescreva os exemplos, pois a formulação manual revela detalhes que a leitura passiva esconde.
- Altere um elemento por vez e observe como a execução muda.
- Resolva os exercícios sem consultar a resposta orientativa; use a resposta para revisar o raciocínio.
- Mantenha um diário de erros com a mensagem recebida, a hipótese testada e a correção realizada.

Sumário

Os títulos abaixo possuem navegação interna no arquivo. Clique em qualquer item para ir diretamente à seção.

Lista de figuras

As figuras são numeradas na ordem em que aparecem e possuem navegação interna no arquivo.

Lista de tabelas

As tabelas são numeradas na ordem em que aparecem e possuem navegação interna no arquivo.

CAPÍTULO 1

Por que estruturas de dados

A forma de organizar os dados determina quais perguntas podem ser respondidas com eficiência.

Dados, operações e representação

Objetivos de aprendizagem. Ao concluir esta parte, você deverá conseguir:

- relacionar operações do problema à representação dos dados;
- distinguir uma estrutura de dados de sua implementação concreta;
- reconhecer por que uma mesma informação pode pedir representações diferentes.

Uma estrutura de dados é uma forma organizada de armazenar informação para que determinadas operações sejam possíveis. A lista de tarefas, a fila de atendimento e a rede de cidades são exemplos de situações em que o significado dos dados não pode ser separado da forma de acesso.

O mesmo conjunto de valores pode ser representado de maneiras diferentes. Uma sequência ordenada favorece acesso por posição; uma pilha favorece o último item inserido; uma fila favorece o primeiro; uma árvore favorece relações hierárquicas; um grafo favorece conexões. A representação não é neutra: ela torna algumas operações naturais e outras mais caras.

Antes de escolher uma classe ou uma biblioteca, descreva as operações necessárias. É preciso buscar por posição, inserir no início, remover o item mais antigo, descobrir vizinhos ou recuperar um valor pela chave? A pergunta principal orienta a estrutura mais adequada.

A biblioteca padrão de Python oferece implementações prontas, como list, deque, heapq, set e dict. Usá-las é uma decisão profissional quando o comportamento atende ao problema. Estudar as estruturas manualmente, porém, permite compreender custos, invariantes e limites das abstrações.

Tabela 1 — Estrutura e operação predominante

Situação	Operação principal	Representação candidata
acesso por posição	ler o item de um índice	sequência indexada
desfazer ações	retirar o último item	pilha
atender por ordem	retirar o primeiro item	fila
relações hierárquicas	visitar descendentes	árvore
conexões entre entidades	visitar vizinhos	grafo
localizar por identificador	buscar uma chave	tabela de espalhamento

Síntese do capítulo

- Estrutura de dados expressa uma relação entre informação e operações.
- A escolha deve começar pelas operações necessárias.
- Bibliotecas prontas e implementações didáticas cumprem papéis diferentes.

Tipos abstratos de dados

Um tipo abstrato de dados descreve o comportamento esperado sem obrigar uma representação interna específica. Uma pilha, por exemplo, pode ser implementada com uma lista, uma sequência encadeada ou outra estrutura. O contrato continua sendo inserir no topo, consultar o topo e retirar do topo.

Separar interface e representação é uma ideia central. O usuário de uma pilha precisa conhecer operações como push e pop, mas não deveria depender de como os itens são armazenados. Essa separação permite trocar a implementação quando o contexto muda.

O contrato também informa pré-condições e pós-condições. Pop em uma pilha vazia pode produzir erro ou retornar um valor especial; a escolha precisa ser documentada. Depois de uma operação, o invariante da estrutura deve continuar válido.

Em Python, uma classe pode expressar o tipo abstrato por meio de métodos. O encapsulamento não torna os dados mágicos: ele apenas concentra regras e reduz o número de lugares que precisam conhecer detalhes internos.

Interface mínima de uma pilha. Leia o código com uma entrada pequena e tente antecipar a saída.

```
class Pilha:
    def __init__(self):
        self._itens = []

    def push(self, item):
        self._itens.append(item)

    def pop(self):
        if not self._itens:
            raise IndexError("pilha vazia")
        return self._itens.pop()
```

Saída esperada

```
A representação fica protegida pelos métodos da classe.
```

Síntese do capítulo

- Tipo abstrato descreve comportamento, não armazenamento.
- Contratos devem informar casos de estrutura vazia.
- Encapsulamento concentra invariantes e operações.

Custo e crescimento

A análise de complexidade estuda como o custo de uma operação cresce quando o tamanho da entrada aumenta. O foco não é medir segundos em uma máquina específica, mas comparar o comportamento relativo de soluções. Uma operação que parece instantânea em uma lista pequena pode se tornar o gargalo em uma lista grande.

A notação O grande descreve um limite assintótico aproximado. Acesso por índice em uma lista costuma ser $O(1)$; percorrer todos os itens é $O(n)$; dois percursos aninhados podem ser $O(n^2)$. Essas expressões não são cronômetros: são modelos para raciocinar sobre crescimento.

Constantes também importam em aplicações reais, mas a análise assintótica permite enxergar a tendência principal. Uma solução $O(n)$ pode possuir uma constante maior que uma solução $O(n^2)$ para entradas muito pequenas e ainda assim ser a escolha mais sustentável quando o volume cresce.

Complexidade de espaço acompanha a memória extra utilizada. Uma transformação que cria uma nova lista pode manter o tempo linear, mas usar memória proporcional a n. A decisão deve considerar tempo, espaço, simplicidade e os requisitos do problema.

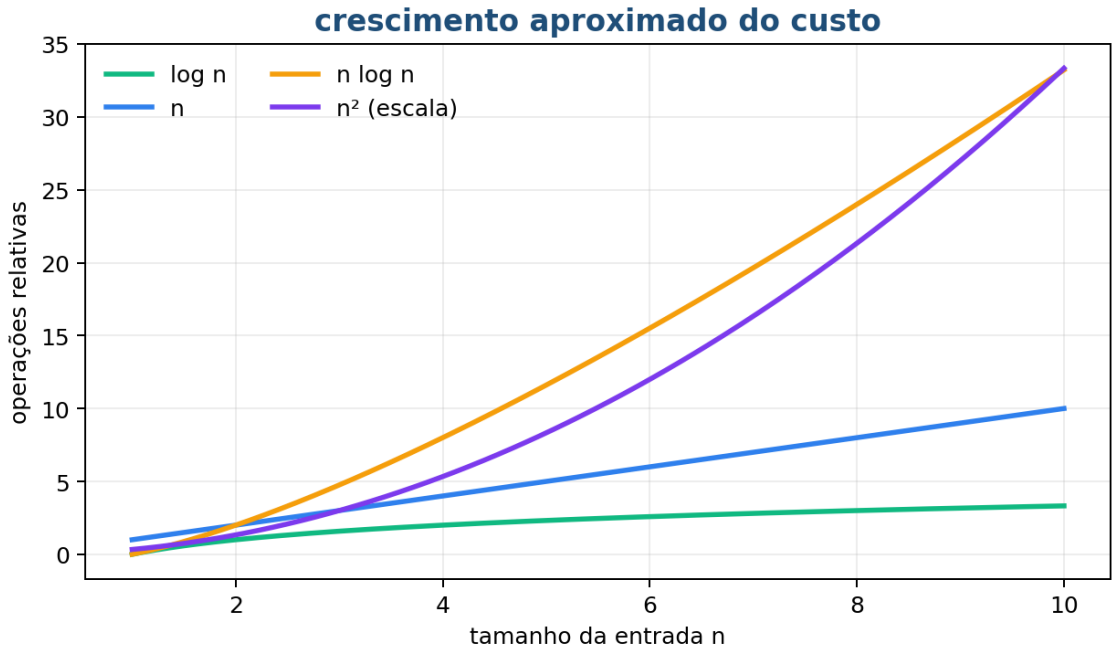


Figura 1 — Crescimento aproximado de funções de custo

Tabela 2 — Leitura inicial de ordens de crescimento

Ordem	Interpretação	Exemplo recorrente
O(1)	custo não depende do tamanho	acesso por índice
O(log n)	reduz o espaço de busca por fator	busca binária
O(n)	visita cada item uma vez	varredura linear
O(n log n)	divide e combina em níveis	ordenação eficiente
O(n²)	compara pares ou usa dois níveis	laços aninhados

Síntese do capítulo

- Complexidade modela crescimento, não um tempo absoluto.
- Tempo e espaço devem ser analisados juntos.
- A entrada pequena não revela necessariamente o comportamento assintótico.

Invariantes e correção

Um invariante é uma propriedade que deve permanecer verdadeira em determinados pontos da execução. Em uma fila, a ordem dos elementos precisa refletir a ordem de chegada. Em uma árvore de busca, os valores da subárvore esquerda devem respeitar a relação definida com o nó. O invariante torna a implementação verificável.

Ao implementar uma operação, descreva o estado antes, a mudança realizada e o estado depois. Essa descrição evita alterar uma referência sem atualizar outra ou remover um item sem ajustar o contador. O código passa a ser uma tradução de uma pequena prova.

Casos vazios, unitários e cheios são especialmente importantes. Uma estrutura pode funcionar para três elementos e falhar quando possui um único nó. Testar a transição entre vazio e não vazio costuma revelar erros de inicialização.

A correção não significa ausência de escolhas. Uma estrutura pode rejeitar operações inválidas, retornar None ou usar um objeto de resultado. O importante é que o comportamento seja consistente com o contrato e esteja documentado para quem chama.

Síntese do capítulo

- Invariantes explicam o que a estrutura nunca deve perder.
- Operações podem ser revisadas como transições de estado.
- Estruturas vazias e unitárias merecem testes próprios.

Exercícios de escolha

Para cada situação, escreva as operações exigidas e escolha uma representação. Não implemente ainda. A justificativa deve mencionar pelo menos uma operação que ficará simples e uma operação que poderá ser mais custosa.

Diagnóstico de representação

Escolha a estrutura principal e justifique a decisão em um parágrafo.

1. Histórico de páginas visitadas com botão voltar.
2. Clientes aguardando atendimento em ordem de chegada.
3. Pastas e arquivos organizados em níveis.
4. Mapa de rotas entre cidades.
5. Cadastro que localiza rapidamente um estudante pelo identificador.

Como conferir. Antes de executar, calcule o resultado para uma entrada pequena.

Síntese do capítulo

- A escolha começa pela lista de operações.
- Toda representação envolve vantagens e custos.
- A justificativa deve mencionar o que a estrutura favorece.

CAPÍTULO 2

Sequências e listas

Uma sequência oferece ordem; uma lista Python acrescenta flexibilidade.

Listas como abstração

A lista é uma sequência ordenada de elementos acessíveis por posição. Em Python, list é mutável: itens podem ser acrescentados, substituídos e removidos. Essa combinação torna a estrutura versátil para programas introdutórios e aplicações gerais.

O índice é uma posição inteira entre zero e o tamanho da lista menos um. Acesso por posição permite ler ou substituir um item, mas não deve ser confundido com busca por valor. Para localizar um item, o programa pode precisar percorrer a sequência até encontrá-lo.

Fatias produzem novas sequências a partir de um intervalo. Elas são úteis para copiar trechos, mas podem consumir memória proporcional ao trecho copiado. A notação compacta não elimina o custo da operação.

Uma lista vazia é uma situação válida, não uma exceção conceitual. Funções que recebem listas devem decidir se retornam um valor neutro, se informam ausência ou se levantam uma exceção. O contrato deve tornar essa escolha visível.

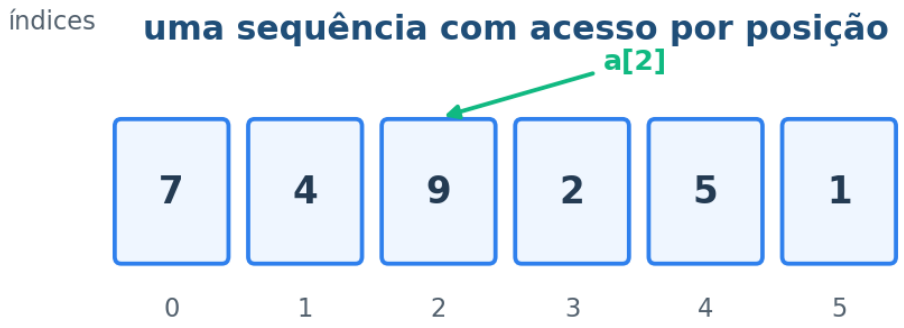


Figura 2 — Acesso por posição em uma lista

Índices, fatias e substituição. Antes de executar, acompanhe a mudança dos valores nas linhas principais.

```
valores = [7, 4, 9, 2, 5, 1]
segundo = valores[1]
trecho = valores[2:5]
valores[0] = 8

print(segundo)
print(trecho)
print(valores)
```

Saída esperada

```
4
[9, 2, 5]
[8, 4, 9, 2, 5, 1]
```

Síntese do capítulo

- Lista combina ordem, acesso por índice e mutabilidade.
- Acesso por posição e busca por valor são operações diferentes.
- Listas vazias precisam de comportamento definido.

Percurso, busca e atualização

Percorrer uma lista visita os itens em uma ordem. A busca linear compara cada elemento até encontrar o alvo ou chegar ao fim. No pior caso, todos os itens são examinados, por isso o custo é O(n).

Atualizar por índice é uma operação direta quando a posição é conhecida. Atualizar pelo valor exige primeiro localizar o item. Se houver valores repetidos, o contrato deve dizer se a alteração ocorre na primeira ocorrência, em todas ou em uma posição indicada.

Remover um item do meio de uma lista exige reorganizar as posições seguintes. A biblioteca cuida dessa movimentação, mas a consequência de custo continua existindo. Inserções e remoções frequentes no início podem indicar que outra estrutura é mais adequada.

Ordenar uma lista modifica sua ordem quando se usa sort e cria uma nova lista quando se usa sorted. Essa diferença é importante quando outras partes do programa compartilham a mesma lista. Mutabilidade é uma ferramenta e também uma responsabilidade.

Busca linear. Teste primeiro o caso apresentado; depois altere uma entrada e observe o que muda.

```
def localizar(valores, alvo):
    for indice, valor in enumerate(valores):
        if valor == alvo:
            return indice
    return -1

dados = [11, 7, 3, 7]
print(localizar(dados, 7))
print(localizar(dados, 5))
```

Saída esperada

```
0
-1
```

Síntese do capítulo

- Busca linear pode examinar toda a sequência.
- Inserir e remover no meio envolvem deslocamento de posições.
- sort e sorted diferem quanto à mutação da lista.

Implementação conceitual de vetor dinâmico

Uma lista dinâmica pode ser entendida como um vetor que reserva espaço para elementos. Enquanto houver capacidade, acrescentar no final é uma operação simples. Quando o espaço termina, a implementação reserva uma área maior e copia os elementos. Essa expansão é custosa em uma operação isolada, mas não ocorre a cada inserção.

A análise amortizada considera o custo médio de uma sequência de operações. Por isso, append é descrito como O(1) amortizado, embora uma expansão ocasional custe O(n). A distinção evita a conclusão errada de que nenhuma inserção jamais pode exigir uma cópia.

O tamanho lógico é a quantidade de elementos válidos; a capacidade é o espaço reservado. Separar os dois conceitos ajuda a entender por que uma lista pode crescer sem realocar em toda operação.

A implementação interna da lista real possui detalhes de baixo nível que não precisam ser reproduzidos em uma aplicação. O objetivo didático é observar a relação entre representação, operação e custo. A classe a seguir é uma versão simplificada para estudo.

Expansão simplificada. Use o trecho para relacionar a regra explicada ao comportamento do programa.

```
class VetorDinamico:
  def __init__(self):
    self._dados = [None] * 2
    self._tamanho = 0

  def append(self, valor):
    if self._tamanho == len(self._dados):
      self._dados += [None] * len(self._dados)
    self._dados[self._tamanho] = valor
    self._tamanho += 1
```

Saída esperada

```
A capacidade dobra quando o vetor fica cheio.
```

Síntese do capítulo

- Tamanho lógico e capacidade são conceitos diferentes.
- Custos ocasionais podem ser avaliados por análise amortizada.
- Implementações didáticas revelam princípios sem substituir a biblioteca padrão.

Exercícios com listas

Registre a complexidade aproximada da operação principal de cada solução. Em seguida, teste listas vazias, listas com um item, valores repetidos e uma entrada maior.

Laboratório de sequências

Implemente as operações sem usar a função pronta que resolve diretamente o objetivo.

6. Encontre a segunda maior posição de uma sequência sem ordenar a lista.
7. Remova repetições preservando a primeira ocorrência de cada valor.
8. Intercale duas listas de mesmo tamanho.
9. Divida uma lista em blocos de tamanho informado.
10. Compare o resultado de uma busca linear com a busca em uma lista ordenada.

Verificação. Compare uma simulação manual com a execução e registre qualquer diferença.

Síntese do capítulo

- Listas favorecem acesso por posição e percurso.
- Valores repetidos precisam de uma regra explícita.
- Complexidade deve acompanhar a implementação dos exercícios.

CAPÍTULO 3

Pilhas e filas

A ordem de entrada e saída define o comportamento da estrutura.

Pilha e disciplina LIFO

Uma pilha segue a disciplina LIFO: o último item inserido é o primeiro a ser retirado. A imagem de uma pilha de livros ajuda a visualizar o comportamento, mas o conceito aparece em chamadas de funções, histórico de edição, desfazer ações e avaliação de expressões.

As operações fundamentais são push, para inserir, pop, para retirar, e peek ou top, para consultar o elemento do topo sem removê-lo. Uma pilha pode informar erro quando está vazia ou oferecer uma operação segura que retorne ausência.

Usar o final de uma lista como topo costuma evitar deslocamentos. Inserir e remover no final são adequados para o contrato da pilha. Usar o início da lista pode funcionar semanticamente, mas pode exigir mover todos os itens.

A pilha também serve para transformar uma solução recursiva em iterativa. Em vez de deixar chamadas pendentes na memória de execução, o programa pode armazenar explicitamente estados que ainda precisam ser processados.

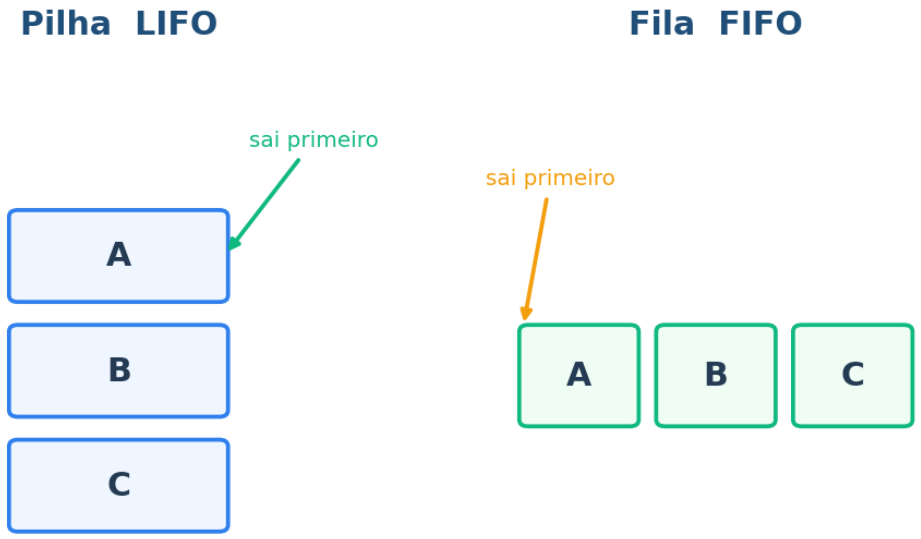


Figura 3 — Pilha e fila em contraste

Desfazer a última ação. Faça uma previsão, execute e compare os dois resultados.

```
pilha = []
pilha.append("abrir")
pilha.append("editar")
pilha.append("salvar")

ultima = pilha.pop()
print(ultima)
```

```
print(pilha)
```

Saída esperada

```
salvar
['abrir', 'editar']
```

Síntese do capítulo

- Pilha retira o item mais recentemente inserido.
- Escolher o final da lista costuma produzir operações adequadas.
- A pilha representa trabalho pendente em muitos algoritmos.

Fila e disciplina FIFO

Uma fila segue a disciplina FIFO: o primeiro item inserido é o primeiro a ser retirado. Filas representam atendimento, tarefas de impressão, mensagens e níveis de uma busca. A ordem de chegada é uma parte do significado.

Em Python, collections.deque oferece operações eficientes nas duas extremidades. A operação append acrescenta ao final; popleft retira o primeiro elemento. Usar pop(0) em uma lista pode produzir deslocamentos repetidos e degradar o desempenho.

Uma fila pode ser limitada. Quando a capacidade está cheia, o produtor pode esperar, rejeitar a entrada ou substituir uma mensagem antiga. O tipo abstrato não decide sozinho qual política é correta; a aplicação precisa documentar a escolha.

Filas de prioridade mudam a regra: o próximo item não é necessariamente o mais antigo, mas o de maior prioridade. Essa estrutura será discutida no capítulo de heaps. O nome fila, portanto, deve vir acompanhado da disciplina efetivamente usada.

Atendimento em ordem de chegada. Observe como a entrada passa por cada etapa até chegar à saída.

```
from collections import deque

fila = deque(["Ana", "Bia"])
fila.append("Caio")

while fila:
    pessoa = fila.popleft()
    print(f"Atendendo {pessoa}")
```

Saída esperada

```
Ana
Bia
Caio
```

Síntese do capítulo

- Fila comum preserva a ordem de chegada.
- deque é adequada para retirar elementos do início.
- Fila de prioridade é uma abstração diferente da fila FIFO.

Aplicações combinadas

Uma pilha pode verificar se delimitadores de uma expressão estão balanceados. Ao encontrar um abre parêntese, o programa empilha; ao encontrar um fecha, compara com o topo. A estrutura torna o histórico relevante para a próxima decisão.

Uma fila pode organizar uma busca em largura. O programa visita um vértice, coloca seus vizinhos ainda não vistos no final da fila e continua retirando pela frente. A ordem da fila produz camadas de distância.

Pilha e fila não são apenas contêineres de itens. Elas codificam políticas de processamento. Em um sistema, a escolha de uma política afeta justiça, latência, memória e previsibilidade.

Quando uma aplicação precisa inserir e retirar em pontos diferentes, escreva o contrato antes da classe. Isso impede que uma lista seja usada com operações incompatíveis com a intenção e facilita substituir a implementação.

Pilha para delimitadores. Acompanhe o fluxo linha a linha e anote o valor que mais lhe interessa.

```
def delimitadores_balanceados(texto):
    pares = {")": "(", "]": "[", "}": "{"}
    abertos = []
    for caractere in texto:
        if caractere in "([{":
            abertos.append(caractere)
        elif caractere in pares:
            if not abertos or abertos.pop() != pares[caractere]:
                return False
    return not abertos

print(delimitadores_balanceados("(a + [b])"))
```

Saída esperada

```
True
```

Síntese do capítulo

- A estrutura escolhida determina a ordem de processamento.
- Pilha é útil quando o histórico mais recente é o primeiro relevante.
- Fila é útil quando o processamento deve avançar por camadas ou chegada.

Exercícios de pilhas e filas

Para cada exercício, indique se a regra é LIFO, FIFO ou prioridade. Depois implemente uma versão simples e teste a transição entre estrutura vazia, estrutura com um item e estrutura com vários itens.

Prática de políticas de acesso

Explique a política da estrutura antes de escrever os métodos.

11. Implemente um histórico com desfazer e refazer limitado.
12. Simule uma fila de senhas de atendimento.
13. Verifique expressões com três tipos de delimitadores.
14. Implemente uma fila circular com capacidade fixa.
15. Organize tarefas por prioridade e compare o resultado com a ordem de chegada.

Depois do teste. Inclua um caso comum e um caso de fronteira na sua verificação.

Síntese do capítulo

- LIFO, FIFO e prioridade são políticas diferentes.
- Casos vazios e unitários revelam erros de fronteira.
- O contrato deve indicar o comportamento de capacidade e ausência.

CAPÍTULO 4

Listas encadeadas

Em uma lista encadeada, a ordem é construída pelas referências entre nós.

Nós e referências

Uma lista encadeada é composta por nós. Cada nó guarda um valor e uma referência para o próximo nó. A estrutura não precisa reservar posições contíguas; a ordem é determinada seguindo referências a partir de um ponto inicial chamado cabeça.

Essa representação facilita inserir um nó depois de uma referência conhecida, porque basta ajustar algumas ligações. Em contrapartida, acessar o item de uma posição exige percorrer os nós desde a cabeça. A estrutura troca acesso direto por flexibilidade de encadeamento.

O último nó aponta para None, indicando o fim. Uma lista vazia possui cabeça igual a None. Esses dois estados são invariantes básicos e devem ser preservados por cada operação.

Referências são valores que apontam para objetos. Ao alterar uma ligação, a ordem das atribuições importa. Guardar a referência do próximo nó antes de substituir o ponteiro evita perder o restante da lista.

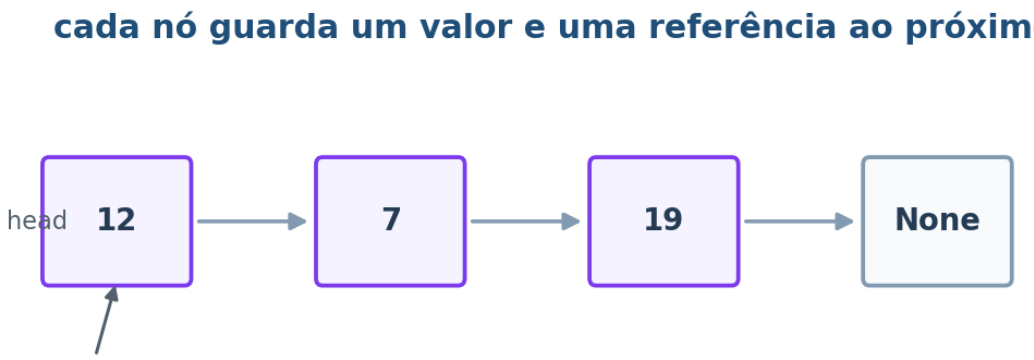


Figura 4 — Encadeamento de nós

Construção de uma cadeia. O caso é pequeno para que a execução possa ser conferida sem pressa.

```
class No:
    def __init__(self, valor, proximo=None):
        self.valor = valor
        self.proximo = proximo

terceiro = No(19)
segundo = No(7, terceiro)
primeiro = No(12, segundo)
```

Saída esperada

```
primeiro aponta para segundo, que aponta para terceiro.
```

Síntese do capítulo

- A ordem encadeada é determinada pelas referências.
- Lista vazia e fim da lista são representados por None.
- Guardar referências antes de alterá-las evita perder nós.

Percurso e inserção

Percorrer uma lista encadeada começa na cabeça e avança enquanto o nó atual não for None. O percurso é linear porque cada passo conhece apenas o próximo nó. Não há acesso direto ao nó de índice cinco sem visitar os anteriores.

Para inserir no início, o novo nó aponta para a cabeça atual e depois a cabeça passa a ser o novo nó. Para inserir depois de um nó conhecido, o novo nó recebe o antigo próximo e o nó anterior passa a apontar para o novo.

A operação de inserção é curta, mas a busca do ponto de inserção pode ser longa. A análise deve separar o custo de localizar a referência do custo de ajustar os ponteiros.

Uma classe de lista deve decidir se expõe nós ou se oferece operações por posição e valor. Expor nós é útil em uma implementação didática; ocultá-los protege a representação em uma biblioteca de aplicação.

Inserção e percurso. Depois de entender a versão básica, experimente uma entrada diferente.

```
def inserir_inicio(cabeca, valor):
  return No(valor, cabeca)

def percorrer(cabeca):
  atual = cabeca
  while atual is not None:
    yield atual.valor
    atual = atual.proximo
```

Saída esperada

```
A função de percurso permite visitar os valores sem copiar a cadeia.
```

Síntese do capítulo

- Acesso por posição em lista encadeada custa um percurso.
- Inserir no início exige ajustar uma referência e a cabeça.
- O custo de localizar e o custo de ligar nós são partes diferentes.

Remoção e listas duplamente encadeadas

Remover um nó exige fazer o nó anterior apontar para o próximo, ignorando o nó removido. Se o alvo for a cabeça, a cabeça precisa avançar. Se o alvo não existir, a operação deve informar ausência sem danificar a cadeia.

Uma lista duplamente encadeada mantém referências para o próximo e para o anterior. Isso facilita voltar no percurso e remover um nó quando a referência ao próprio nó está disponível. O preço é manter mais ligações consistentes.

Cada inserção ou remoção em uma lista dupla pode exigir atualizar duas ou quatro referências. O invariante é que, quando um nó possui vizinho anterior, o vizinho anterior deve apontar de volta para ele. Testes devem verificar os dois sentidos.

A estrutura encadeada pode ser útil quando há muitas inserções e remoções e pouca necessidade de acesso aleatório. Ela não é automaticamente mais rápida: se a aplicação busca constantemente por índice, o percurso repetido pode dominar o custo.

Tabela 3 — Comparação entre sequências

Aspecto	Lista dinâmica	Lista encadeada
acesso por índice	direto, geralmente O(1)	percurso, O(n)
inserção no início	desloca itens	ajusta referências
memória	área de elementos	elemento mais referências
localidade	boa em memória	depende dos nós
uso natural	acesso e percurso	encadeamento e remoção

Síntese do capítulo

- Remoção deve preservar as ligações da cadeia.
- Lista dupla oferece navegação nos dois sentidos com maior custo de manutenção.
- A estrutura só é adequada em relação às operações da aplicação.

Exercícios de listas encadeadas

Desenhe a cadeia antes e depois de cada operação. O desenho deve mostrar cabeça, valores e referências. Depois implemente e use uma função que converta a lista para uma sequência comum apenas para conferir o resultado.

Laboratório de referências

Implemente e teste invariantes após cada operação.

16. Insira um valor no fim de uma lista encadeada.
17. Remova a primeira ocorrência de um valor.
18. Inverta uma lista encadeada sem criar novos nós.
19. Conte os nós e retorne o último valor.
20. Implemente uma lista duplamente encadeada com percurso nos dois sentidos.

Para conferir. Altere os valores do exemplo e confirme se a regra continua válida.

Síntese do capítulo

- Desenhos tornam referências abstratas visíveis.
- Cada operação deve conservar cabeça e fim.
- Lista encadeada troca acesso direto por flexibilidade de ligação.

CAPÍTULO 5

Árvores e recursão

Árvores organizam relações hierárquicas; a recursão descreve uma estrutura usando versões menores dela mesma.

Recursão como estratégia

Uma função recursiva chama a si própria para resolver um problema menor. Para ser correta, ela precisa de um caso-base, que pode ser respondido diretamente, e de um passo recursivo que aproxime a entrada desse caso. Sem essas duas partes, a execução pode não terminar.

A pilha de chamadas guarda as versões ainda não concluídas. Quando uma chamada chega ao caso-base, os resultados retornam na ordem inversa em que as chamadas foram feitas. Esse comportamento explica por que a recursão combina naturalmente com estruturas aninhadas e percursos de árvores.

Recursão não garante eficiência. Cada chamada tem custo de memória, e algumas definições recursivas repetem trabalho. A escolha deve considerar legibilidade, tamanho da entrada, profundidade possível e existência de uma versão iterativa simples.

Antes de escrever uma função recursiva, descreva a redução: qual parte do problema será resolvida agora e qual parte ficará para a chamada seguinte? Em seguida, prove informalmente que a medida escolhida diminui. Pode ser o tamanho de uma lista, a altura restante ou o número de itens ainda não visitados.

Caso-base e redução. Preste atenção ao ponto em que o programa decide, repete ou transforma os dados.

```
def fatorial(n):
    if n < 0:
        raise ValueError('n deve ser não negativo')
    if n in (0, 1):
        return 1
    return n * fatorial(n - 1)

print(fatorial(5))
```

Saída esperada

```
120
```

Síntese do capítulo

- Toda recursão precisa de caso-base e redução.
- A pilha de chamadas explica a ordem de retorno.
- Legibilidade, profundidade e custo devem orientar a escolha.

Vocabulário das árvores

Uma árvore é formada por nós ligados por relações de pai e filho, sem ciclos no modelo usual. O nó superior é a raiz. Nós sem filhos são folhas; nós que possuem filhos são internos. A profundidade mede a distância até a raiz, enquanto a altura indica a maior distância até uma folha.

Uma árvore binária possui no máximo dois filhos por nó, geralmente chamados esquerdo e direito. A forma da árvore importa: duas árvores podem conter os mesmos valores e apresentar alturas muito diferentes. Como a altura influencia os percursos, também influencia o custo de operações que descem pela árvore.

Os três percursos clássicos são pré-ordem, em-ordem e pós-ordem. Na pré-ordem, visita-se o nó antes dos filhos; na em-ordem, entre o filho esquerdo e o direito; na pós-ordem, depois dos dois. O percurso por níveis usa uma fila e visita a árvore camada a camada.

A representação deve tornar explícitos os casos vazios. Uma árvore sem nós pode ser representada por None. Uma função de percurso deve retornar imediatamente nesse caso, porque não há valor nem filho a visitar.

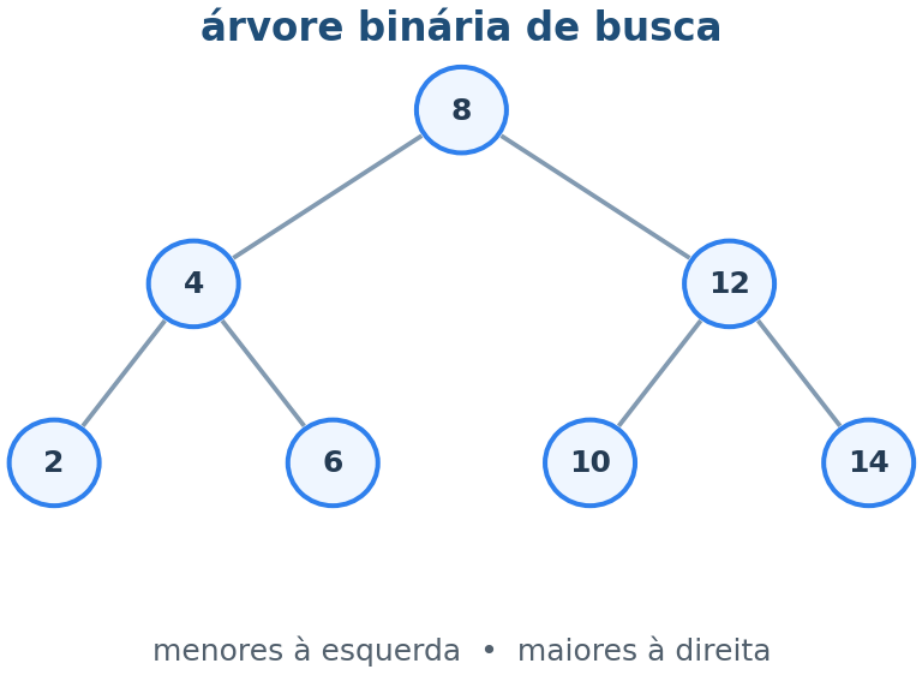


Figura 5 — Representação de uma árvore binária

Percurso em ordem. Rode o exemplo uma vez e descreva, com suas palavras, o que aconteceu.

```
class NoArvore:
  def __init__(self, valor, esquerda=None, direita=None):
    self.valor = valor
    self.esquerda = esquerda
    self.direita = direita

def em_ordem(no):
  if no is not None:
    yield from em_ordem(no.esquerda)
    yield no.valor
    yield from em_ordem(no.direita)
```

Saída esperada

```
Em uma árvore de busca, os valores aparecem ordenados.
```

Síntese do capítulo

- Raiz, folha, profundidade e altura descrevem papéis e distâncias.
- Percursos diferentes respondem a perguntas diferentes.
- O caso de árvore vazia é parte do contrato.

Árvore binária de busca

Uma árvore binária de busca mantém um invariante: valores menores que o nó ficam no lado esquerdo e valores maiores, no lado direito. Se valores repetidos forem permitidos, a política deve ser declarada, como guardar contagem no próprio nó ou escolher um lado fixo.

A busca compara o alvo com o nó atual e descarta metade da direção possível. A vantagem depende da altura. Em uma árvore equilibrada, a altura cresce aproximadamente como o logaritmo do número de nós; em uma árvore inclinada, a busca pode se comportar como uma lista.

A inserção segue o mesmo caminho da busca até encontrar uma posição vazia. A remoção tem três situações: nó folha, nó com um filho e nó com dois filhos. No último caso, é possível substituir o valor pelo sucessor em-ordem e remover o nó que forneceu esse sucessor.

A árvore de busca é uma estrutura com regra semântica, não apenas uma forma geométrica. Depois de cada operação, um teste de percurso em ordem pode verificar se a sequência permanece ordenada. Verificar o invariante é mais forte do que conferir apenas um exemplo de busca.

Tabela 4 — Operações em uma árvore de busca

Operação	Caminho típico	Custo dependente da altura
buscar	comparar e escolher um filho	O(h)
inserir	descer até uma posição vazia	O(h)
remover	ajustar um ou dois caminhos	O(h)
percorrer	visitar todos os nós	O(n)

Busca iterativa em árvore binária. Compare a saída com o contrato descrito no texto anterior.

```
def buscar(no, alvo):
  while no is not None:
    if alvo == no.valor:
      return no
    no = no.esquerda if alvo < no.valor else no.direita
  return None
```

Saída esperada

```
Retorna o nó encontrado ou None.
```

Síntese do capítulo

- A regra de ordenação permite descartar subárvores.
- O custo é O(h), e a altura pode variar com a forma.
- Percurso em ordem é uma verificação simples do invariante.

Exercícios de árvores

Represente a árvore com um desenho antes de executar o código. Em seguida, escreva a sequência produzida por cada percurso. Por fim, identifique o invariante que precisa permanecer verdadeiro após a operação.

Laboratório de hierarquias

Use árvores pequenas, com valores repetidos e com diferentes formas.

21. Implemente uma função que retorne a altura de uma árvore.
22. Conte as folhas de uma árvore binária usando recursão.
23. Implemente pré-ordem, em-ordem e pós-ordem.
24. Insira valores em uma árvore de busca e compare alturas.
25. Implemente a busca e trate a árvore vazia.
26. Descreva como remover um nó com dois filhos.

Como conferir. Releia o enunciado e verifique se cada condição foi contemplada.

Síntese do capítulo

- Recursão acompanha a definição de uma árvore.
- A forma da árvore afeta a altura e o custo.
- Invariantes tornam operações complexas verificáveis.

CAPÍTULO 6

Heaps e filas de prioridade

Uma fila de prioridade retira o item mais importante segundo uma regra explícita.

Prioridade e heap

Uma fila de prioridade oferece inserção e remoção do item de maior ou menor prioridade. Ela aparece em escalonamento, simulação de eventos, algoritmos de grafos e seleção dos próximos trabalhos. Diferentemente de uma fila FIFO, o tempo de chegada não é o único critério.

Um heap mínimo é uma árvore quase completa na qual cada pai tem valor menor ou igual aos seus filhos. O menor elemento fica na raiz, mas os demais não estão totalmente ordenados. Essa distinção é importante: um heap permite encontrar a prioridade extrema, não percorrer todos os valores em ordem sem trabalho adicional.

A forma quase completa permite armazenar o heap em uma lista. Para um índice i iniciado em zero, o filho esquerdo está em 2i + 1 e o direito em 2i + 2. O pai de um índice positivo pode ser obtido por (i - 1) // 2. A representação evita referências explícitas.

Após inserir no fim, o novo item pode violar a relação com o pai. O procedimento de subir troca o item com o pai enquanto necessário. Após remover a raiz, o último item ocupa o topo e pode precisar descer para restaurar o invariante.

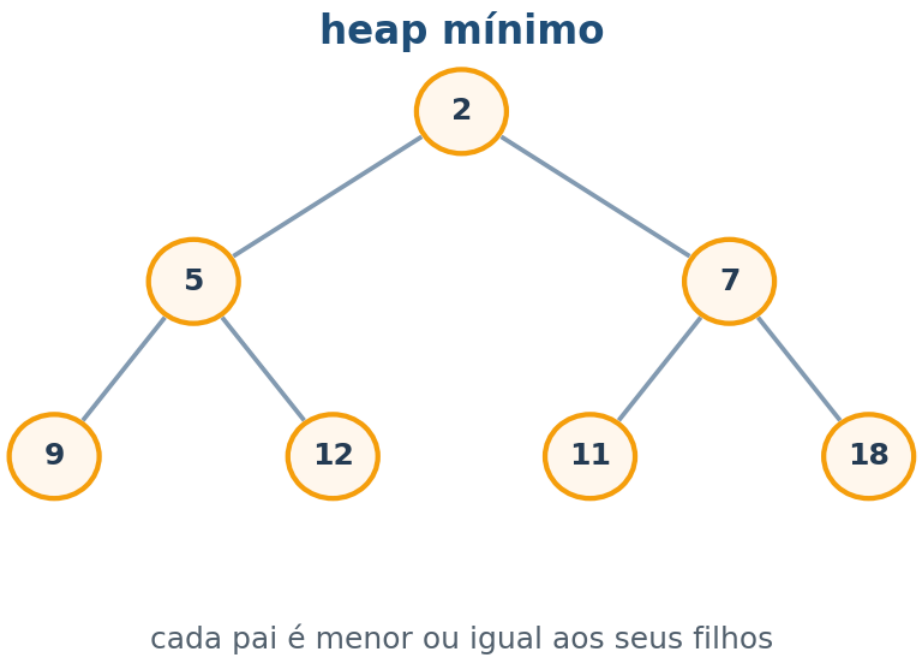


Figura 6 — Heap mínimo como árvore e como lista

Fila de prioridade com heapq. Leia o código com uma entrada pequena e tente antecipar a saída.

```
import heapq

tarefas = []
heapq.heappush(tarefas, (2, 'revisar'))
```

```
heapq.heappush(tarefas, (1, 'corrigir'))
prioridade, tarefa = heapq.heappop(tarefas)
print(prioridade, tarefa)
```

Saída esperada

```
1 corrigir
```

Síntese do capítulo

- Heap mantém uma prioridade extrema acessível.
- A árvore quase completa pode ser representada por uma lista.
- A ordem total dos itens não é um requisito do heap.

Operações e análise

Em um heap, consultar o menor item é O(1), pois ele está na posição inicial. Inserir e remover o menor item percorrem no máximo a altura da árvore, portanto custam O(log n). Construir um heap a partir de uma coleção pode ser feito em O(n) com heapify, embora inserir cada item um a um custe O(n log n).

Empates exigem atenção. Se a prioridade for apenas um número, dois itens com o mesmo valor podem não ser comparáveis entre si. Uma tupla com prioridade e contador de chegada pode oferecer desempate estável. O contrato precisa indicar se estabilidade é necessária.

A biblioteca heapq implementa um heap mínimo sobre uma lista. Para simular um heap máximo, uma estratégia comum é armazenar prioridades negativas quando os valores são numéricos. Para objetos complexos, use uma chave comparável e mantenha o objeto associado.

Não confunda a estrutura interna com a política de negócio. Um sistema de chamados pode usar menor número como maior urgência, ou pode usar prazo, impacto e ordem de chegada. A estrutura só consegue aplicar a chave que a aplicação define.

Tabela 5 — Custos típicos de uma fila de prioridade

Operação	Heap binário	Observação
consultar mínimo	O(1)	não remove o item
inserir	O(log n)	subida pelo caminho
remover mínimo	O(log n)	descida pelo caminho
construir com heapify	O(n)	a partir de coleção
percorrer ordenado	O(n log n)	retirando repetidamente

Processamento por instante. Antes de executar, acompanhe a mudança dos valores nas linhas principais.

```
import heapq

eventos = [(30, 'fechar'), (10, 'abrir'), (20, 'processar')]
heapq.heapify(eventos)
while eventos:
    instante, acao = heapq.heappop(eventos)
    print(instante, acao)
```

Saída esperada

```
10 abrir
20 processar
```

```
30 fechar
```

Síntese do capítulo

- O custo de inserção e remoção acompanha a altura do heap.
- Empates e estabilidade precisam de uma política explícita.
- heapq oferece uma implementação prática de heap mínimo.

Exercícios de heaps

Escreva primeiro qual item deve sair em cada situação. Depois observe a lista interna apenas para compreender o invariante; não use a aparência da lista como se fosse uma sequência ordenada.

Laboratório de prioridades

Compare a regra de prioridade com uma fila FIFO.

27. Implemente uma agenda de eventos ordenada por instante.
28. Crie um desempate por ordem de chegada.
29. Use heapq para obter os três menores valores de uma coleção.
30. Simule o atendimento de chamados por urgência.
31. Explique por que a lista de um heap não aparece completamente ordenada.

Verificação. Acompanhe o contador ou acumulador em uma tabela curta.

Síntese do capítulo

- A prioridade deve ser uma chave comparável e documentada.
- A lista interna de um heap não é uma lista ordenada.
- A escolha entre FIFO e prioridade muda o comportamento do sistema.

CAPÍTULO 7

Grafos e redes

Grafos representam relações; seus algoritmos exploram caminhos, vizinhanças e conectividade.

Vocabulário e modelagem

Um grafo é formado por vértices e arestas. Vértices podem representar pessoas, páginas, cidades ou estados; arestas representam relações entre eles. O mesmo conjunto de pontos pode modelar amizades, rotas de transporte, dependências ou conexões de uma rede.

Em um grafo não direcionado, uma aresta entre A e B pode ser percorrida nos dois sentidos. Em um grafo direcionado, a seta importa. Arestas podem possuir pesos, como distância, duração ou custo. A modelagem correta depende de qual pergunta o programa precisa responder.

Um caminho é uma sequência de vértices conectados. Um ciclo retorna ao ponto de partida. O grau de um vértice indica quantas arestas incidem nele, com convenções próprias para grafos direcionados. Esses termos permitem descrever propriedades antes de escolher um algoritmo.

Modelar é decidir o que será um vértice e o que será uma aresta. Se a decisão for inadequada, o algoritmo pode estar correto para o modelo e ainda assim responder a uma pergunta diferente da desejada. Registre exemplos pequenos e desenhe o grafo antes de implementar.

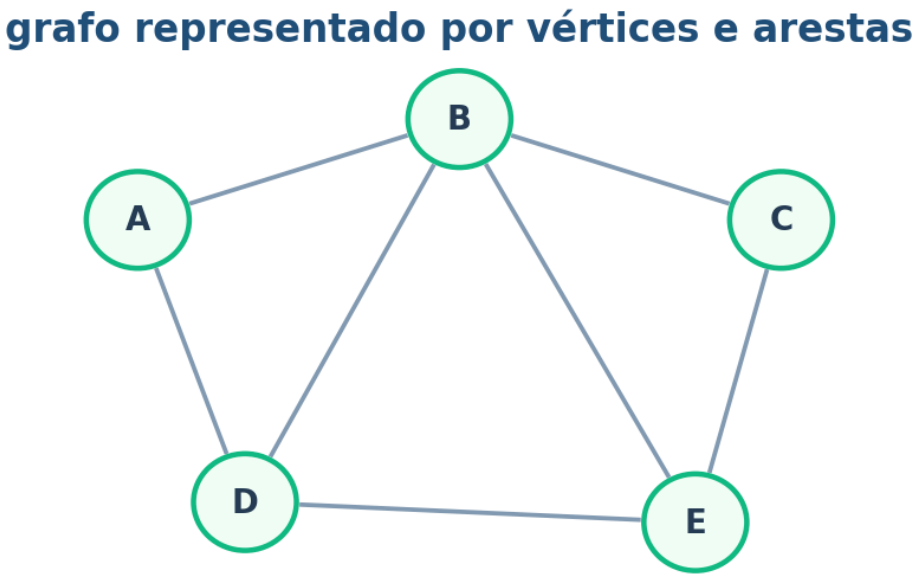


Figura 7 — Exemplo de grafo

Síntese do capítulo

- A semântica dos vértices e das arestas vem do problema.
- Direção e peso mudam os algoritmos aplicáveis.

- Desenhos pequenos ajudam a validar a modelagem.

Representações

A lista de adjacência associa cada vértice aos vizinhos. Ela usa memória proporcional ao número de vértices mais arestas e é natural quando o algoritmo precisa percorrer os vizinhos de um ponto. Um dicionário de listas ou conjuntos é uma implementação comum em Python.

A matriz de adjacência usa uma tabela em que a posição linha, coluna indica a existência ou o peso de uma aresta. Consultar diretamente se há uma aresta é simples, mas a matriz ocupa espaço proporcional ao quadrado do número de vértices, mesmo quando poucas relações existem.

A representação deve ser escolhida pelo padrão de uso. Grafos esparsos, com poucas arestas em relação ao máximo possível, favorecem listas de adjacência. Grafos densos ou operações frequentes de consulta direta podem justificar uma matriz. O custo de converter entre representações também pode importar.

Em um grafo com rótulos, use chaves legíveis e mantenha uma política para vértices desconhecidos. Decida se adicionar uma aresta cria automaticamente seus vértices ou se exige cadastro prévio. Uma política explícita evita estados parcialmente válidos.

Tabela 6 — Lista e matriz de adjacência

Aspecto	Lista de adjacência	Matriz de adjacência
memória	O(V + E)	O(V²)
listar vizinhos	eficiente	percorre uma linha
consultar aresta	depende do grau	O(1)
grafos esparsos	boa escolha	desperdiça posições
pesos	valor junto ao vizinho	valor na célula

Lista de adjacência com conjuntos. Teste primeiro o caso apresentado; depois altere uma entrada e observe o que muda.

```
grafo = {
   'A': {'B', 'D'},
   'B': {'A', 'C'},
   'C': {'B', 'D'},
   'D': {'A', 'C'},
}
```

Saída esperada

```
Os conjuntos evitam repetir o mesmo vizinho.
```

Síntese do capítulo

- Lista de adjacência economiza espaço em grafos esparsos.
- Matriz oferece consulta direta em troca de memória.
- O contrato deve definir como vértices e arestas são criados.

Busca em largura e profundidade

A busca em largura, conhecida como BFS, visita primeiro os vértices mais próximos da origem. Ela usa uma fila e um conjunto de visitados. Em grafos não ponderados, a primeira vez que um vértice é descoberto fornece um caminho com o menor número de arestas.

A busca em profundidade, ou DFS, avança por um caminho antes de retornar para explorar outras alternativas. Pode ser implementada com recursão ou com uma pilha explícita. DFS é útil para componentes, detecção de ciclos e ordenações que dependem de precedências.

Marcar um vértice quando ele é descoberto, e não apenas quando é retirado da fila, evita que vários vizinhos o coloquem repetidamente na fronteira. A decisão parece pequena, mas muda a quantidade de trabalho e simplifica a reconstrução do caminho.

Em grafos desconectados, uma busca iniciada em um único vértice não alcança tudo. Para visitar todos os componentes, percorra os vértices e inicie uma nova busca quando encontrar um ainda não visitado. O objetivo da busca precisa ser declarado: alcançar um destino não é o mesmo que visitar o grafo inteiro.

Distâncias em um grafo não ponderado. Use o trecho para relacionar a regra explicada ao comportamento do programa.

```
from collections import deque

def distancias(grafo, origem):
    dist = {origem: 0}
    fila = deque([origem])
    while fila:
        atual = fila.popleft()
        for vizinho in grafo.get(atual, set()):
            if vizinho not in dist:
                dist[vizinho] = dist[atual] + 1
                fila.append(vizinho)
    return dist
```

Saída esperada

```
distancias retorna o menor número de arestas até cada vértice alcançável.
```

Síntese do capítulo

- BFS usa fila e encontra distâncias mínimas em grafos não ponderados.
- DFS pode usar recursão ou pilha.
- Vértices visitados impedem trabalho repetido e ciclos infinitos.

Caminhos e pesos

Quando as arestas têm pesos não negativos, contar arestas não basta. O caminho com menos saltos pode ser mais caro que outro com mais trechos. O algoritmo de Dijkstra mantém a melhor distância conhecida e explora primeiro o vértice com menor custo provisório, normalmente usando uma fila de prioridade.

A escolha do algoritmo depende das propriedades do grafo. BFS é suficiente para pesos iguais; Dijkstra requer pesos não negativos; outros cenários, como pesos negativos ou todos os pares de distâncias, pedem técnicas diferentes. Usar o algoritmo errado pode produzir uma resposta plausível, mas incorreta.

Para reconstruir o caminho, guarde o predecessor de cada vértice quando uma distância é melhorada. Depois, parta do destino e volte pelos predecessores até a origem, invertendo a sequência. Separar cálculo de distâncias e montagem da resposta torna o código mais testável.

Em aplicações reais, também é preciso definir o que acontece quando o destino não é alcançável, quando há múltiplos caminhos de mesmo custo e quando um peso está ausente ou é inválido. Esses casos fazem parte do contrato da função.

Reconstrução por predecessores. Faça uma previsão, execute e compare os dois resultados.

```
def reconstruir(predecessor, origem, destino):
    caminho = []
    atual = destino
    while atual is not None:
        caminho.append(atual)
        if atual == origem:
            return list(reversed(caminho))
        atual = predecessor.get(atual)
    return None
```

Saída esperada

```
Retorna a sequência da origem ao destino ou None.
```

Síntese do capítulo

- Pesos alteram a noção de melhor caminho.
- O algoritmo precisa respeitar as propriedades dos dados.
- Predecessores permitem devolver o caminho completo, não apenas seu custo.

Exercícios de grafos

Modele cada situação com um pequeno desenho. Identifique se o grafo é direcionado, se possui pesos e se pode haver ciclos. Em seguida, escolha a representação e justifique a busca utilizada.

Laboratório de redes

Teste os algoritmos com grafos vazios, desconectados e com ciclos.

32. Implemente BFS para retornar a ordem de visita.
33. Retorne um caminho entre duas pessoas em uma rede.
34. Conte os componentes conexos de um grafo não direcionado.
35. Implemente DFS iterativa com uma pilha.
36. Compare BFS e Dijkstra em um grafo com pesos iguais.
37. Explique por que Dijkstra não deve ser usado com pesos negativos.

Depois do teste. Teste a entrada mínima, vazia ou ausente quando ela fizer sentido.

Síntese do capítulo

- Uma boa modelagem vem antes da busca.
- Representação e propriedades do grafo determinam os custos.
- Teste conectividade, ciclos e ausência de caminho.

CAPÍTULO 8

Tabelas de espalhamento

Tabelas de espalhamento aproximam a busca direta ao transformar chaves em posições.

Chaves, funções hash e colisões

Uma tabela de espalhamento associa uma chave a um valor. A função hash transforma a chave em um número; uma redução, como o resto da divisão pelo tamanho da tabela, produz um índice. O valor armazenado pode ser recuperado quando a mesma chave é apresentada.

Uma colisão acontece quando chaves diferentes produzem o mesmo índice. Colisões não são uma falha excepcional: uma tabela finita recebe potencialmente muitas chaves. A estrutura precisa de uma estratégia para continuar armazenando e encontrando os itens.

Na resolução por encadeamento, cada posição guarda uma pequena coleção de pares chave-valor. Na endereçagem aberta, a tabela procura outra posição conforme uma sequência definida, como sondagem linear. Cada técnica tem custos e requisitos de ocupação próprios.

O dicionário de Python oferece uma tabela de espalhamento para as operações usuais. Conhecer o modelo ajuda a interpretar o custo médio esperado e a escolher uma chave apropriada. Não se deve presumir que qualquer objeto é uma boa chave: ela precisa ser hashable e manter igualdade consistente.

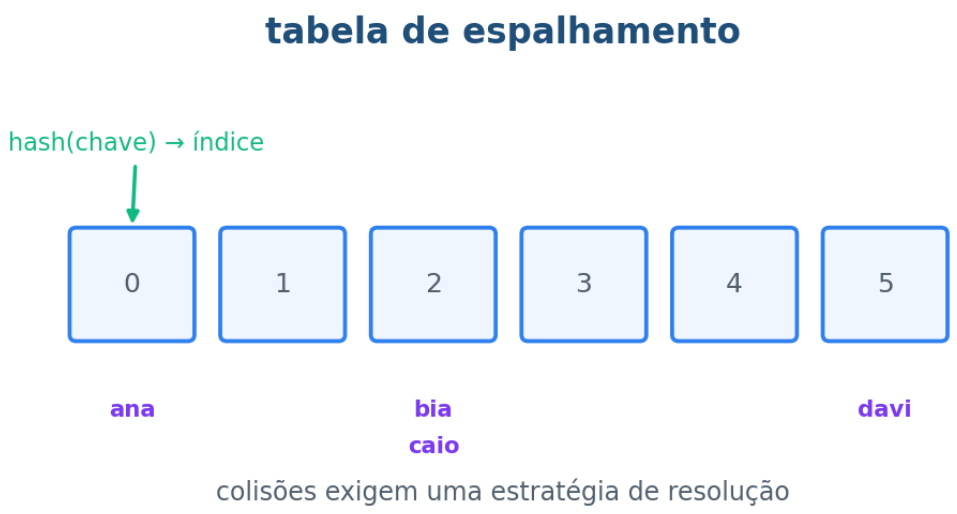


Figura 8 — Ideia de uma tabela de espalhamento

Acesso por chave. Observe como a entrada passa por cada etapa até chegar à saída.

```
contatos = {'ana': '5551', 'bia': '5552'}
contatos['caio'] = '5553'
print(contatos.get('bia', 'não encontrado'))
```

Saída esperada

```
5552
```

Síntese do capítulo

- Hash transforma chaves em posições candidatas.

- Colisões exigem encadeamento ou sondagem.
- dict oferece uma implementação prática de associação chave-valor.

Carga e implementação didática

O fator de carga relaciona o número de itens ao número de posições. À medida que a tabela fica cheia, aumentam as colisões e as buscas podem percorrer mais candidatos. Implementações de biblioteca redimensionam a estrutura para preservar um custo médio adequado.

Uma implementação didática por encadeamento precisa de operações para inserir, buscar e remover. Ao inserir uma chave já existente, a política usual é atualizar o valor, e não criar uma segunda associação indistinguível. Remover uma chave ausente deve ser uma operação definida, como retornar False.

O custo médio esperado de busca, inserção e remoção é O(1) sob uma distribuição razoável de hashes e uma carga controlada. No pior caso, muitas chaves podem cair no mesmo balde e o custo chegar a O(n). A análise deve declarar a hipótese; notação assintótica não elimina o comportamento dos dados.

A qualidade da chave também importa. Uma classe que redefine __eq__ deve manter compatibilidade com __hash__, ou pode produzir resultados surpreendentes. Para estruturas próprias, escreva testes com chaves iguais, chaves diferentes que colidem e tabela pequena.

Tabela 7 — Cenários de acesso por chave

Situação	Custo médio esperado	Risco
buscar chave existente	O(1)	colisões e carga
inserir nova chave	O(1)	redimensionamento
remover chave	O(1)	marcação ou encadeamento
pior caso	O(n)	muitos itens no mesmo índice

Busca didática com encadeamento. Acompanhe o fluxo linha a linha e anote o valor que mais lhe interessa.

```
class TabelaHash:
  def __init__(self, tamanho=8):
    self.baldes = [[] for _ in range(tamanho)]

  def _balde(self, chave):
    return self.baldes[hash(chave) % len(self.baldes)]

  def buscar(self, chave):
    for k, valor in self._balde(chave):
      if k == chave:
        return valor
    return None
```

Saída esperada

```
A lista do balde é percorrida somente quando necessário.
```

Síntese do capítulo

- Fator de carga influencia colisões e custo.
- Atualização e ausência devem fazer parte do contrato.
- O custo O(1) é uma expectativa média sob hipóteses.

Exercícios de espalhamento

Use uma tabela pequena para forçar colisões e registre a posição calculada para cada chave. O objetivo é observar a estratégia de resolução, não reproduzir a implementação interna do dicionário de Python.

Laboratório de chaves

Compare o comportamento da tabela com uma lista de pares.

38. Implemente inserir, buscar e remover em uma tabela encadeada.
39. Conte colisões durante as inserções.
40. Crie uma função de frequência de palavras usando dict.
41. Explique por que listas não são boas para busca repetida por chave.
42. Teste chaves iguais, chaves ausentes e colisões forçadas.

Para conferir. Explique em uma frase por que o resultado atende ao enunciado.

Síntese do capítulo

- Colisões são parte normal do modelo.
- Uma tabela deve preservar a associação entre chave e valor.
- Testes com tabela pequena tornam o mecanismo observável.

CAPÍTULO 9

Ordenação e busca

Ordenar é reorganizar; buscar é explorar uma organização para responder mais depressa.

Busca linear e busca binária

A busca linear percorre os itens até encontrar o alvo ou chegar ao fim. Ela funciona em qualquer sequência e pode parar cedo quando o alvo aparece. No pior caso, compara todos os n itens, portanto seu custo é O(n).

A busca binária exige uma sequência ordenada. Ela compara o alvo com o elemento central e descarta metade do intervalo possível. Repetindo a operação, encontra o item ou conclui que ele não está presente em O(log n) comparações.

A precondição de ordenação não é um detalhe interno: é parte da especificação. Se a sequência mudar depois de ordenada, os resultados da busca binária deixam de ser confiáveis. Em uma aplicação, pode ser melhor manter uma coleção ordenada ou ordenar uma cópia, dependendo do padrão de atualização.

Ao buscar valores repetidos, a pergunta precisa ser refinada. Queremos qualquer ocorrência, a primeira, a última ou a posição de inserção? A biblioteca bisect oferece operações úteis para limites em sequências ordenadas, mas o chamador ainda precisa definir o significado da posição.

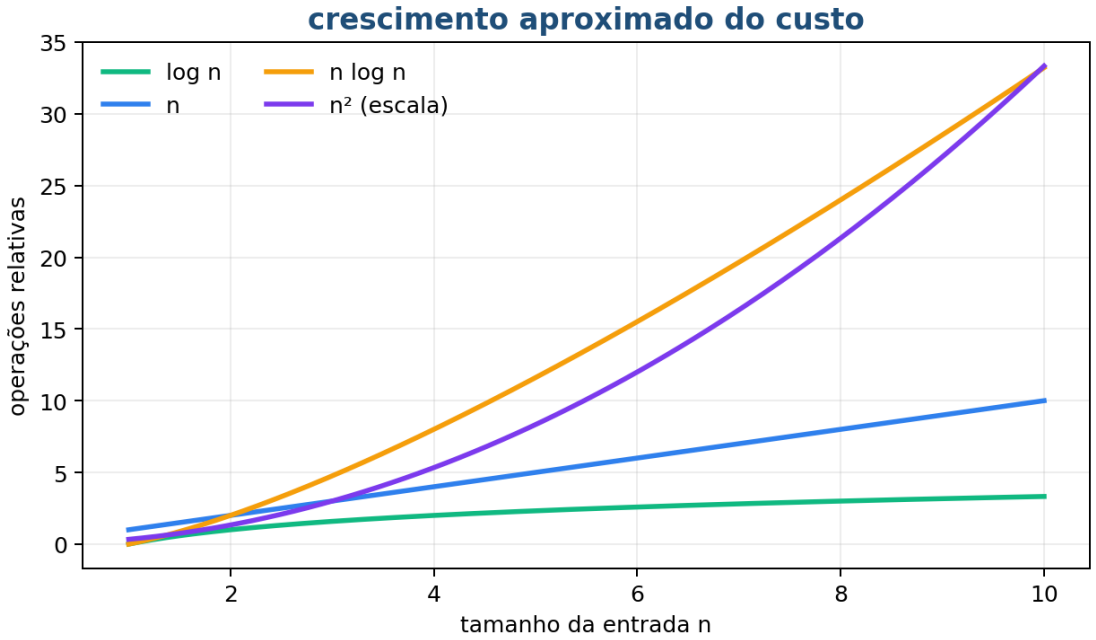


Figura 9 — Crescimento de custos de busca e ordenação

Busca binária iterativa. O caso é pequeno para que a execução possa ser conferida sem pressa.

```
def busca_binaria(valores, alvo):
    inicio, fim = 0, len(valores) - 1
    while inicio <= fim:
        meio = (inicio + fim) // 2
        if valores[meio] == alvo:
            return meio
        if valores[meio] < alvo:
            inicio = meio + 1
        else:
```

```
            fim = meio - 1
    return None
```

Saída esperada

```
Retorna um índice ou None; valores deve estar ordenado.
```

Síntese do capítulo

- Busca linear não exige ordenação e custa O(n) no pior caso.
- Busca binária exige ordenação e reduz o intervalo pela metade.
- A pergunta sobre ocorrências repetidas deve ser explícita.

Ordenação por inserção

O algoritmo de inserção mantém um prefixo ordenado. Para cada novo elemento, desloca os maiores para a direita até abrir uma posição adequada. Ele é simples de acompanhar e costuma funcionar bem em coleções pequenas ou quase ordenadas.

A versão in-place usa pouca memória adicional, mas modifica a sequência. Ela pode ser estável quando desloca apenas elementos estritamente maiores, preservando a ordem relativa de itens equivalentes. Estabilidade é relevante quando uma ordenação é feita em etapas.

Em uma entrada já ordenada, o algoritmo faz poucas comparações e deslocamentos, aproximando-se de O(n). Em uma entrada inversamente ordenada, pode realizar uma quantidade quadrática de trabalho. O melhor e o pior caso contam histórias diferentes sobre a mesma implementação.

Mesmo algoritmos simples devem ter um critério de teste. Depois da ordenação, verifique se cada par consecutivo está na ordem e se a quantidade de elementos foi preservada. Para registros, verifique também se a chave de ordenação foi aplicada corretamente.

Ordenação por inserção. Depois de entender a versão básica, experimente uma entrada diferente.

```
def insercao(valores):
    resultado = list(valores)
    for i in range(1, len(resultado)):
        atual = resultado[i]
        j = i - 1
        while j >= 0 and resultado[j] > atual:
            resultado[j + 1] = resultado[j]
            j -= 1
        resultado[j + 1] = atual
    return resultado
```

Saída esperada

```
insercao([4, 2, 3]) retorna [2, 3, 4] e preserva a entrada.
```

Síntese do capítulo

- O prefixo ordenado é o invariante central.
- A implementação pode ser simples e estável.
- Entrada quase ordenada muda o desempenho observado.

Ordenação por intercalação

A ordenação por intercalação divide a sequência, ordena cada metade e intercala duas partes já ordenadas. A intercalação percorre cada parte uma vez. A divisão recorrente cria aproximadamente log n níveis, resultando em custo O(n log n).

A intercalação compara os primeiros elementos ainda não usados de cada parte. O menor é copiado para a saída, e o ponteiro daquela parte avança. Quando uma parte termina, o restante da outra pode ser anexado sem novas comparações.

A técnica normalmente usa memória auxiliar para construir as partes e a saída. O custo de memória é uma troca por previsibilidade de tempo. Em Python, sorted e list.sort implementam algoritmos altamente otimizados; estudar a intercalação ajuda a compreender a ideia, não a substituir a biblioteca.

Comparar algoritmos exige controlar o experimento. Use entradas do mesmo tamanho, repita medições, separe tempo de preparação e observe o tipo de dado. Um único cronômetro para um caso pequeno não prova que uma ordem de crescimento é melhor em todos os cenários.

Tabela 8 — Características de algoritmos de ordenação

Algoritmo	Tempo típico	Memória adicional	Estável?
inserção	$O(n^2)$ pior caso	baixa	sim, com comparação estrita
intercalação	O(n log n)	moderada	sim, se a intercalação preservar empates
sorted do Python	depende da entrada	implementação	sim

Núcleo da intercalação. Preste atenção ao ponto em que o programa decide, repete ou transforma os dados.

```
def intercalar(esquerda, direita):
    saida = []
    i = j = 0
    while i < len(esquerda) and j < len(direita):
        if esquerda[i] <= direita[j]:
            saida.append(esquerda[i])
            i += 1
        else:
            saida.append(direita[j])
            j += 1
    return saida + esquerda[i:] + direita[j:]
```

Saída esperada

```
Recebe duas sequências ordenadas e devolve uma sequência ordenada.
```

Síntese do capítulo

- Intercalação combina partes ordenadas com uma passagem linear.
- O custo O(n log n) vem da divisão em níveis.
- Medições precisam de entradas e procedimentos controlados.

Exercícios de ordenação e busca

Para cada exercício, registre a precondição, a saída esperada e o custo assintótico. Inclua entradas vazias, de um item, já ordenadas e com valores repetidos.

Laboratório de ordenação

Compare a solução própria com sorted apenas para validar o resultado.

43. Implemente busca linear que retorne todas as posições.
44. Adapte a busca binária para retornar a primeira ocorrência.

45. Ordene registros por nome e, em empate, por nota.
46. Conte comparações realizadas pela ordenação por inserção.
47. Implemente a divisão e a intercalação completas.
48. Explique quando ordenar uma cópia é preferível a alterar a entrada.

Como conferir. Escolha um valor no limite da regra e confira o comportamento.

Síntese do capítulo

- Busca e ordenação devem ser analisadas junto com suas precondições.
- Estabilidade e uso de memória são requisitos possíveis.
- Testes devem verificar ordem, conteúdo e casos de fronteira.

CAPÍTULO 10

Escolha de estruturas e projetos

A melhor estrutura é a que expressa as operações importantes do problema com custo e comportamento aceitáveis.

Uma matriz de decisão

Não há uma estrutura universalmente melhor. Uma lista é uma escolha natural para acesso por índice e percurso; deque para operações nas duas extremidades; pilha para histórico e retorno; fila para chegada; heap para prioridade; árvore para hierarquias ou busca ordenada; grafo para relações; dicionário para acesso por chave.

A escolha começa pelas operações mais frequentes. Pergunte quais itens serão inseridos, removidos, encontrados e percorridos; se a ordem possui significado; se os dados têm relações; e se a memória precisa ser contígua. A resposta reduz o espaço de alternativas antes de considerar micro-otimizações.

Uma análise responsável também considera manutenção. A implementação mais rápida pode ser inadequada se o contrato ficar difícil de entender ou se os invariantes não puderem ser testados. Em Python, prefira estruturas da biblioteca padrão quando elas atendem ao problema e documente a razão da escolha.

O custo assintótico é um mapa, não uma medição completa. Constantes, localidade de memória, tamanho real da entrada, frequência de operações e custo de conversão influenciam o resultado. Use análise para levantar hipóteses e experimentos para investigar o comportamento.

Tabela 9 — Estrutura e operação natural

Estrutura	Pergunta que responde bem	Operação característica
lista	qual item está no índice i?	acesso por posição
deque	quem está em cada extremidade?	append e popleft
pilha	qual foi o item mais recente?	push e pop
heap	qual tem menor prioridade?	heappop
árvore de busca	onde está uma chave ordenada?	buscar por comparação
grafo	quais objetos se relacionam?	visitar vizinhos
dict	qual valor corresponde à chave?	acesso associativo

Síntese do capítulo

- Comece pelo contrato e pelas operações frequentes.
- Estruturas da biblioteca padrão reduzem código acidental.
- Análise e experimento se complementam.

Projeto um: gerenciador de tarefas

Um gerenciador de tarefas precisa localizar uma tarefa por identificador e escolher a próxima tarefa por prioridade. Um dicionário atende à primeira operação; um heap atende à segunda. Manter as duas estruturas significa aceitar que elas podem ficar temporariamente com registros antigos, desde que a remoção valide a versão atual.

O projeto ilustra uma técnica comum: usar uma estrutura principal para consulta e uma estrutura auxiliar para ordenar candidatos. Quando uma tarefa é atualizada, inserir uma nova entrada no heap é mais simples que procurar e alterar uma entrada existente. Ao retirar, descarte entradas obsoletas.

O domínio precisa definir a ordem das prioridades, o comportamento para identificador desconhecido e o que significa concluir uma tarefa. Essas decisões devem aparecer em funções pequenas, com nomes que indiquem a regra. O programa abaixo é um núcleo didático, não uma aplicação pronta para produção.

Dicionário mais heap em um projeto. Rode o exemplo uma vez e descreva, com suas palavras, o que aconteceu.

```
import heapq
from itertools import count

class Tarefas:
    def __init__(self):
        self.por_id = {}
        self.fila = []
        self.versoes = count()

    def adicionar(self, identificador, prioridade, texto):
        versao = next(self.versoes)
        registro = (prioridade, versao, identificador, texto)
        self.por_id[identificador] = registro
        heapq.heappush(self.fila, registro)

    def proxima(self):
        while self.fila:
            registro = heapq.heappop(self.fila)
            if self.por_id.get(registro[2]) == registro:
                del self.por_id[registro[2]]
                return registro
        return None
```

Saída esperada

```
A validação evita que uma versão antiga seja entregue.
```

Síntese do capítulo

- Uma aplicação pode combinar estruturas com responsabilidades diferentes.
- Versões ajudam a invalidar entradas auxiliares antigas.
- Regras do domínio precisam ser visíveis no código.

Projeto dois: caminhos em uma rede

Um mapa pequeno pode ser representado por um dicionário de adjacência. Para descobrir se dois pontos estão conectados, uma BFS usa uma fila e um conjunto de visitados. Para devolver o caminho, o programa guarda o predecessor de cada ponto descoberto.

A implementação precisa evitar um erro comum: marcar apenas o destino quando ele for retirado da fila. Em um grafo com vários caminhos, o mesmo vértice poderia entrar muitas vezes. Marcar no momento da descoberta mantém uma única fronteira por vértice.

O projeto pode ser estendido com pesos, restrições de horário e pontos bloqueados. Cada extensão altera o modelo ou o algoritmo. Se os trechos receberem tempos diferentes, a BFS deixa de ser suficiente para minimizar duração e uma fila de prioridade passa a ser necessária.

Caminho mínimo em número de trechos. Compare a saída com o contrato descrito no texto anterior.

```
from collections import deque

def caminho(grafo, origem, destino):
    anteriores = {origem: None}
    fila = deque([origem])
    while fila:
        atual = fila.popleft()
        if atual == destino:
            break
        for vizinho in grafo.get(atual, []):
            if vizinho not in anteriores:
                anteriores[vizinho] = atual
                fila.append(vizinho)
    if destino not in anteriores:
        return None
    resultado = []
    atual = destino
    while atual is not None:
        resultado.append(atual)
        atual = anteriores[atual]
    return resultado[::-1]
```

Saída esperada

```
BFS devolve um caminho curto em um grafo não ponderado.
```

Síntese do capítulo

- Fila, conjunto e dicionário cooperam na BFS.
- Predecessores transformam descoberta em resposta explicável.
- Adicionar pesos muda a pergunta e pode mudar o algoritmo.

Projeto três: frequência de palavras

Contar palavras é uma aplicação direta de acesso por chave. O texto é percorrido, cada palavra é normalizada e o dicionário acumula uma contagem. Depois, uma lista de pares pode ser ordenada para produzir um ranking. O projeto combina strings, dicionários e ordenação.

Normalizar não significa apagar informação sem pensar. Converter para minúsculas, remover pontuação, tratar acentos e decidir como lidar com números são escolhas do domínio. O resultado precisa ser definido antes de comparar implementações.

A contagem é aproximadamente linear no tamanho do texto sob o custo médio do dicionário. A ordenação do ranking depende do número de palavras distintas. Relatar esses dois custos ajuda a explicar por que a estrutura escolhida é adequada.

Contagem e ranking. Leia o código com uma entrada pequena e tente antecipar a saída.

```
import re

def frequencias(texto):
    contagem = {}
    palavras = re.findall(r'[A-Za-zÀ-ÿ]+', texto.lower())
    for palavra in palavras:
        contagem[palavra] = contagem.get(palavra, 0) + 1
    return sorted(contagem.items(), key=lambda par: (-par[1], par[0]))
```

```
print(frequencias('Dados e dados: código!'))
```

Saída esperada

```
[('dados', 2), ('código', 1), ('e', 1)]
```

Síntese do capítulo

- Dicionário é natural para acumular valores por palavra.
- Normalização é parte da especificação.
- O ranking acrescenta uma etapa de ordenação.

Checklist de projeto

Um projeto de estruturas de dados começa com uma pergunta e termina com evidências. Antes de codificar, escreva o contrato. Durante a implementação, registre os invariantes. Depois, meça ou argumente sobre custos e teste casos que poderiam quebrar a representação.

A documentação deve explicar por que uma estrutura foi escolhida, e não apenas como os métodos funcionam. Um leitor precisa conseguir trocar a implementação sem alterar o código que usa o contrato. Essa separação é uma das ideias centrais de tipos abstratos de dados.

Para apresentar um projeto, mostre um exemplo pequeno que possa ser acompanhado manualmente, uma implementação legível, testes automatizados e uma discussão de limites. Resultados honestos, incluindo situações em que a solução não serve, tornam o material mais útil.

Tabela 10 — Revisão antes de entregar

Pergunta	Evidência esperada
O contrato é claro?	entradas, saídas e erros documentados
A estrutura preserva seu invariante?	testes após operações
O custo é conhecido?	análise de operações relevantes
Casos de fronteira foram testados?	vazio, um item, repetição e ausência
A escolha é justificável?	comparação com alternativas
O código é reutilizável?	funções pequenas e responsabilidades separadas

Síntese do capítulo

- Projetos conectam abstração, implementação, teste e análise.
- Invariantes e contratos orientam a manutenção.
- A justificativa fica mais forte quando registra limites e alternativas.

Exercícios integradores

Escolha um projeto e produza uma pequena entrega: problema, modelo, estruturas escolhidas, operações, implementação, testes, análise de custo e reflexão sobre alternativas.

Desafio final

Faça uma apresentação curta em que outra pessoa consiga executar e revisar a solução.

49. Implemente uma agenda que combine dict e heap.
50. Modele uma rede de disciplinas e encontre pré-requisitos.
51. Crie um índice de palavras para buscar linhas de um texto.
52. Compare uma lista, um deque e uma lista encadeada em operações de extremidade.
53. Use uma árvore de busca para manter nomes ordenados.
54. Escreva uma tabela justificando cada estrutura escolhida.

Verificação. Procure uma inicialização que, se estivesse errada, alteraria o resultado.

Síntese do capítulo

- Escolha da estrutura é uma decisão de projeto.
- A solução precisa ser explicável por operações e invariantes.
- Testes e análise dão sustentação à implementação.

CAPÍTULO 11

Laboratórios de estruturas de dados

Cada laboratório relaciona uma operação, um invariante, uma implementação e uma forma de medir.

Laboratório 1: comparar operações

Comece por uma tabela de operações, não por uma classe. Para cada estrutura, escreva como inserir, buscar, remover e percorrer. A comparação revela que uma estrutura pode ser excelente para uma operação e inadequada para outra.

Use listas pequenas para acompanhar o estado depois de cada operação. O objetivo é distinguir o comportamento abstrato da representação concreta. Uma lista Python, por exemplo, esconde uma área de armazenamento redimensionável.

A análise assintótica descreve crescimento, mas não substitui o contrato. Se uma operação precisa preservar a ordem, um custo semelhante pode esconder comportamentos diferentes. Registre também se a operação modifica ou copia os dados.

Conclua com uma recomendação para um caso de uso. Uma justificativa curta, apoiada por operações frequentes, é mais valiosa do que afirmar que uma estrutura é simplesmente rápida.

Tabela 11 — Mapa inicial

Necessidade	Estrutura candidata	Pergunta
acesso por índice	lista	a posição é importante?
retirar dos dois lados	deque	há operações nas extremidades?
último a entrar primeiro	pilha	o histórico recente domina?
menor custo primeiro	heap	há prioridade?
associação por chave	dict	a chave identifica o item?

Ficha de comparação

Escolha a operação dominante antes de escolher o tipo.

55. Compare lista e deque para remover do início.
56. Compare lista e conjunto para testar pertencimento.
57. Indique uma estrutura para tarefas urgentes.
58. Escreva uma justificativa com custo e semântica.

Depois do teste. Compare uma versão simples com um caso um pouco maior.

Síntese do capítulo

- A operação dominante guia a escolha.
- Comportamento e custo devem ser analisados juntos.
- Uma justificativa deve mencionar alternativas.

Laboratório 2: vetor dinâmico

Uma lista dinâmica mantém elementos em posições consecutivas e aumenta a capacidade quando o espaço acaba. A expansão pode copiar os itens para uma área maior. Embora uma inserção individual possa custar O(n), o custo médio de acrescentar ao final pode ser amortizado.

O laboratório é conceitual: use uma classe pequena para observar tamanho e capacidade. A proposta não é substituir list, mas perceber que uma operação abstrata pode esconder passos internos.

Escolha uma política de crescimento e documente-a. Crescer uma posição por vez copia muitas vezes; crescer por fator reserva espaço e usa memória adicional. Não existe número mágico independente do padrão de uso.

Teste sequências que cruzam a capacidade. Verifique se nenhum valor desaparece, se o tamanho representa a quantidade lógica e se a capacidade nunca fica menor que o tamanho.

Crescimento por fator. Antes de executar, acompanhe a mudança dos valores nas linhas principais.

```
class VetorDinamico:
    def __init__(self):
        self.dados = [None] * 2
        self.tamanho = 0

    def append(self, valor):
        if self.tamanho == len(self.dados):
            self.dados.extend([None] * len(self.dados))
        self.dados[self.tamanho] = valor
        self.tamanho += 1
```

Saída esperada

```
A capacidade aumenta quando o vetor fica cheio.
```

Ficha de vetor

Observe tamanho lógico e capacidade física separadamente.

59. Implemente acesso por índice com validação.
60. Implemente remoção do fim.
61. Registre quantas expansões ocorreram.
62. Explique por que o custo amortizado não é O(1) em toda operação.

Para conferir. Registre a parte da solução que pode ser reutilizada em outro exercício.

Síntese do capítulo

- Representação física e tamanho lógico são conceitos diferentes.
- Expansão troca cópia ocasional por espaço reservado.
- Custo amortizado descreve uma sequência de operações.

Laboratório 3: pilha e histórico

Um editor simples pode armazenar ações em uma pilha para implementar desfazer. Cada ação precisa conter informação suficiente para reverter a mudança. Uma segunda pilha pode armazenar ações desfeitas e permitir refazer.

O contrato deve indicar o que acontece quando não há ação para desfazer. Retornar False, devolver None ou lançar uma exceção são escolhas possíveis, mas o chamador precisa saber qual delas foi adotada.

Limitar o tamanho do histórico é uma política de memória. Quando a capacidade é atingida, a ação mais antiga pode ser descartada. Essa decisão muda a experiência do usuário e deve ser testada.

O exemplo abaixo usa strings como ações. Um sistema real poderia usar objetos ou funções, mas o padrão de acesso continua sendo LIFO.

Duas pilhas para histórico. Teste primeiro o caso apresentado; depois altere uma entrada e observe o que muda.

```
historico = []
refazer = []

def registrar(acao):
    historico.append(acao)
    refazer.clear()

def desfazer():
    if not historico:
        return None
    acao = historico.pop()
    refazer.append(acao)
    return acao
```

Saída esperada

```
Registrar uma ação nova limpa o caminho de refazer.
```

Ficha de pilhas

Desenhe as duas pilhas após cada chamada.

63. Implemente refazer.
64. Defina um limite de histórico.
65. Retorne uma mensagem para pilha vazia.
66. Teste registrar, desfazer e registrar novamente.

Como conferir. Antes de executar, calcule o resultado para uma entrada pequena.

Síntese do capítulo

- Histórico recente é uma aplicação de LIFO.
- Desfazer e refazer possuem estados coordenados.
- Limites de capacidade são parte da política.

Laboratório 4: fila e simulação

Uma simulação de atendimento representa clientes em uma fila. A cada passo, um cliente pode chegar, ser atendido ou abandonar a espera. A fila mantém a ordem dos clientes que ainda aguardam.

O relógio da simulação é um estado adicional. Não basta retirar nomes: registre instante de chegada, início do atendimento e tempo de espera. A estrutura da fila resolve a ordem; o modelo resolve o significado dos tempos.

Use deque para retirar do início sem deslocar todos os elementos. Se o atendimento tiver prioridades, substitua a fila por uma fila de prioridade e compare os resultados. A troca da estrutura deve alterar uma política identificável.

Casos de fronteira incluem nenhum cliente, um único cliente e vários clientes chegando no mesmo instante. Especifique como desempatar antes de implementar.

Simulação de atendimento. Use o trecho para relacionar a regra explicada ao comportamento do programa.

```
from collections import deque

eventos = deque([('Ana', 2), ('Bia', 5), ('Caio', 7)])
relogio = 0
while eventos:
```

```
    nome, chegada = eventos.popleft()
    relogio = max(relogio, chegada)
    print(nome, 'esperou', relogio - chegada)
    relogio += 1
```

Saída esperada

```
A fila preserva a ordem de chegada.
```

Ficha de filas

Registre relógio, chegada e saída em uma tabela.

67. Calcule o tempo médio de espera.
68. Modele dois atendentes.
69. Adicione abandonos depois de um limite.
70. Compare FIFO com prioridade.

Verificação. Compare uma simulação manual com a execução e registre qualquer diferença.

Síntese do capítulo

- Fila representa uma política de espera.
- Tempo e ordem são estados diferentes.
- Simulações exigem regras de empate e abandono.

Laboratório 5: encadeamento

Implemente uma lista encadeada começando por uma cadeia de três nós desenhada à mão. O desenho torna visível qual referência deve ser alterada em cada inserção. Depois transforme a operação em função.

Na inserção depois de um nó, salve o próximo antigo antes de escrever o novo encadeamento. Essa ordem protege os nós restantes. Faça o mesmo raciocínio para remoção: o nó anterior precisa saltar o alvo.

Uma lista encadeada não fornece acesso aleatório barato. Se o programa busca repetidamente pela posição, o custo de percorrer a cadeia pode superar a vantagem de inserir sem deslocar. A análise precisa refletir o uso real.

Teste cabeça, meio, fim e ausência. Esses casos exercitam referências diferentes e revelam se a estrutura preserva o fim None.

Inserção após referência conhecida. Faça uma previsão, execute e compare os dois resultados.

```
def inserir_depois(no, valor):
    novo = No(valor, no.proximo)
    no.proximo = novo

def valores(cabeca):
    atual = cabeca
    while atual is not None:
        yield atual.valor
        atual = atual.proximo
```

Saída esperada

```
O novo nó assume o próximo antigo antes de a ligação ser atualizada.
```

Ficha de encadeamento

Faça um desenho para cada mutação.

71. Insira no início e no fim.

72. Remova a primeira ocorrência.
73. Inverta os ponteiros.
74. Conte acessos necessários para buscar a posição i.

Depois do teste. Inclua um caso comum e um caso de fronteira na sua verificação.

Síntese do capítulo

- Referências precisam ser atualizadas em ordem segura.
- Casos de cabeça, meio e fim são distintos.
- O custo de localização pode dominar o custo de ligação.

Laboratório 6: recursão e árvore

Use uma árvore pequena para observar que uma operação em um nó se repete em suas subárvores. Uma função de contagem pode somar um para o nó atual e delegar o restante aos filhos. O caso None encerra cada ramo.

Escreva a versão recursiva e depois descreva uma versão iterativa. A pilha explícita armazena nós que ainda precisam ser visitados. Comparar as versões esclarece o que a linguagem guarda implicitamente.

A altura é uma medida de estrutura. Uma árvore muito inclinada pode ter n nós e altura próxima de n; uma árvore balanceada pode ter altura próxima de log n. O mesmo algoritmo pode ter comportamentos diferentes conforme a forma.

Use percurso em ordem para validar árvores de busca e percurso por níveis para observar a largura. Não misture o objetivo de visitar com o objetivo de ordenar.

Duas medidas recursivas. Observe como a entrada passa por cada etapa até chegar à saída.

```
def contar_nos(no):
    if no is None:
        return 0
    return 1 + contar_nos(no.esquerda) + contar_nos(no.direita)

def altura(no):
    if no is None:
        return 0
    return 1 + max(altura(no.esquerda), altura(no.direita))
```

Saída esperada

```
O caso vazio define a convenção para a altura.
```

Ficha de árvores

Declare se a altura conta nós ou arestas.

75. Conte folhas.
76. Calcule a soma dos valores.
77. Escreva o percurso por níveis.
78. Compare uma árvore equilibrada com uma inclinada.

Para conferir. Altere os valores do exemplo e confirme se a regra continua válida.

Síntese do capítulo

- Recursão divide a árvore em subárvores.
- Altura influencia o custo de busca.
- A convenção de medida deve ser documentada.

Laboratório 7: árvore de busca

Construa uma árvore de busca inserindo valores em uma ordem escolhida. Insira a mesma coleção em outra ordem e compare os desenhos. A diferença demonstra que uma árvore comum não se equilibra automaticamente.

Use o percurso em ordem como teste de invariável. Se os valores não aparecem em ordem, alguma ligação ou regra de inserção está incorreta. O teste percorre a estrutura inteira, não apenas o último elemento.

A remoção de uma folha é simples; a remoção de um nó com um filho exige ligar o pai ao neto; dois filhos exigem escolher um sucessor ou predecessor. Desenhe antes de alterar o código.

Para entradas ordenadas, a estrutura pode degenerar em uma cadeia. Essa limitação é uma boa oportunidade para pesquisar árvores balanceadas depois de dominar o caso básico.

Inserção recursiva. Acompanhe o fluxo linha a linha e anote o valor que mais lhe interessa.

```
def inserir(no, valor):
    if no is None:
        return NoArvore(valor)
    if valor < no.valor:
        no.esquerda = inserir(no.esquerda, valor)
    else:
        no.direita = inserir(no.direita, valor)
    return no
```

Saída esperada

```
Retornar no ao final preserva a raiz de cada subárvore.
```

Ficha de busca

Teste ordens de inserção diferentes.

79. Implemente buscar.
80. Implemente o menor valor.
81. Verifique a ordenação em ordem.
82. Explique o pior caso para dados já ordenados.

Como conferir. Releia o enunciado e verifique se cada condição foi contemplada.

Síntese do capítulo

- A ordem de inserção influencia a forma.
- Percurso em ordem verifica a regra de busca.
- Árvore básica pode degenerar sem balanceamento.

Laboratório 8: heap em uma agenda

Uma agenda de eventos precisa retirar sempre o menor instante. Uma lista ordenada pode resolver o caso pequeno, mas cada inserção pode deslocar itens. O heap representa diretamente a operação de menor prioridade e mantém o custo de inserção proporcional à altura.

A prioridade deve vir acompanhada do evento. Quando duas tarefas possuem o mesmo instante, acrescente um contador de chegada se a ordem de entrada for importante. Não dependa de comparar objetos que não possuem uma ordem natural.

Use a lista interna apenas para observar a estrutura. Depois de cada operação, confira a propriedade pai-menor-que-filhos. A aparência da lista não deve ser usada como teste de ordenação completa.

Meça uma sequência de inserções e remoções, mas não tire conclusões de um único tamanho. A pergunta é como o trabalho cresce quando n aumenta.

Desempate com contador. O caso é pequeno para que a execução possa ser conferida sem pressa.

```
import heapq
from itertools import count

contador = count()
agenda = []

def agendar(instante, nome):
  heapq.heappush(agenda, (instante, next(contador), nome))

agendar(30, 'fechar')
agendar(10, 'abrir')
print(heapq.heappop(agenda))
```

Saída esperada

```
(10, 1, 'abrir')
```

Ficha de heap

Registre prioridade, desempate e item.

83. Implemente cancelar por identificador.
84. Obtenha os três menores eventos.
85. Compare heap com lista ordenada.
86. Explique o efeito de muitos empates.

Verificação. Acompanhe o contador ou acumulador em uma tabela curta.

Síntese do capítulo

- Heap expressa a operação de menor prioridade.
- Contador pode tornar o desempate estável.
- A propriedade do heap é local, não uma ordenação total.

Laboratório 9: BFS em um mapa

Modele um mapa como grafo não ponderado. Cada local é vértice e cada passagem é aresta. A BFS parte da origem e visita camadas; por isso o primeiro caminho encontrado minimiza o número de passagens.

O dicionário de predecessores registra de onde cada vértice veio. Se o destino não aparecer, não há caminho na componente alcançada. Se o grafo for desconectado, iniciar de outro vértice produz outra componente.

Uma fila de deque evita retirar do início de uma lista. O conjunto de visitados impede ciclos e trabalho repetido. Essas duas escolhas são parte da implementação do algoritmo.

Faça uma tabela com fila, visitados e predecessores depois de cada passo. A tabela é uma ferramenta de aprendizagem e também uma forma de explicar o resultado.

BFS com predecessores. Depois de entender a versão básica, experimente uma entrada diferente.

```
from collections import deque

def caminho_sem_peso(grafo, origem, destino):
  anterior = {origem: None}
  fila = deque([origem])
  while fila:
    atual = fila.popleft()
```

```
        for vizinho in grafo[atual]:
            if vizinho not in anterior:
                anterior[vizinho] = atual
                fila.append(vizinho)
    if destino not in anterior:
        return None
    caminho = []
    while destino is not None:
        caminho.append(destino)
        destino = anterior[destino]
    return caminho[::-1]
```

Saída esperada

```
A resposta contém vértices, não apenas a distância.
```

Ficha de BFS

Desenhe as camadas de distância.

87. Retorne apenas a distância até o destino.
88. Conte componentes conexas.
89. Trate origem igual ao destino.
90. Teste um destino inalcançável.

Depois do teste. Teste a entrada mínima, vazia ou ausente quando ela fizer sentido.

Síntese do capítulo

- BFS constrói camadas a partir da origem.
- Predecessores permitem reconstruir o caminho.
- Fila e visitados evitam caminhos repetidos.

Laboratório 10: DFS e componentes

A busca em profundidade explora um vizinho e continua até não haver caminho novo. Use uma pilha explícita para acompanhar a fronteira. A ordem dos vizinhos pode mudar a ordem de visita, mas não deve mudar o conjunto alcançado.

Para contar componentes, percorra todos os vértices e inicie uma DFS quando o vértice ainda não estiver visitado. A busca marca toda a componente; o laço externo encontra a próxima.

A recursão é uma descrição natural, mas um grafo muito profundo pode ultrapassar o limite de chamadas. A versão iterativa torna o espaço explícito. Em ambos os casos, a ideia é controlar uma fronteira de vértices descobertos.

Teste ciclos, laços e vértices isolados. Um vértice sem arestas forma uma componente sozinho em um grafo não direcionado.

Componentes com DFS iterativa. Preste atenção ao ponto em que o programa decide, repete ou transforma os dados.

```
def componentes(grafo):
    visitados = set()
    total = 0
    for inicio in grafo:
        if inicio in visitados:
            continue
        total += 1
        pilha = [inicio]
        while pilha:
            atual = pilha.pop()
```

```
        if atual in visitados:
            continue
        visitados.add(atual)
        pilha.extend(grafo[atual])
    return total
```

Saída esperada

```
A contagem inclui vértices isolados.
```

Ficha de DFS

Compare ordens de vizinhos diferentes.

91. Retorne a ordem de visita.
92. Detecte se há um ciclo simples.
93. Conte componentes de um grafo desconectado.
94. Explique a diferença entre pilha e fila no percurso.

Para conferir. Explique em uma frase por que o resultado atende ao enunciado.

Síntese do capítulo

- DFS percorre um caminho antes de voltar.
- Componente inclui vértices alcançáveis entre si.
- A ordem pode variar; o conjunto visitado deve ser consistente.

Laboratório 11: pesos e Dijkstra

Quando uma rede possui distâncias diferentes, o caminho com menos arestas pode não ser o mais barato. Dijkstra mantém uma melhor distância conhecida para cada vértice e explora primeiro a alternativa de menor custo provisório.

A fila de prioridade pode conter registros antigos. Quando um registro sai, compare seu custo com a melhor distância atual e descarte-o se estiver obsoleto. Essa técnica evita procurar e alterar itens no meio do heap.

Os pesos precisam ser não negativos para a justificativa usual do algoritmo. Se aparecer um peso negativo, interrompa a implementação e revise o modelo. Um resultado numérico não é prova de correção.

Guarde predecessores sempre que uma distância melhorar. Ao final, reconstrua o caminho e devolva custo e sequência, ou uma indicação clara de que o destino não é alcançável.

Dijkstra com entradas obsoletas. Rode o exemplo uma vez e descreva, com suas palavras, o que aconteceu.

```
import heapq

def distancias_peso(grafo, origem):
    melhor = {origem: 0}
    fila = [(0, origem)]
    while fila:
        custo, atual = heapq.heappop(fila)
        if custo != melhor[atual]:
            continue
        for vizinho, peso in grafo.get(atual, []):
            novo = custo + peso
            if novo < melhor.get(vizinho, float('inf')):
                melhor[vizinho] = novo
                heapq.heappush(fila, (novo, vizinho))
    return melhor
```

Saída esperada

```
A distância final é a melhor conhecida para cada vértice alcançado.
```

Ficha de caminhos ponderados

Liste os custos candidatos antes de escolher o próximo.

95. Adicione predecessores ao algoritmo.
96. Reconstrua um caminho mínimo.
97. Teste pesos iguais e compare com BFS.
98. Explique por que peso negativo quebra a hipótese.

Como conferir. Escolha um valor no limite da regra e confira o comportamento.

Síntese do capítulo

- Pesos mudam a definição de melhor caminho.
- Heap pode conter registros que ficaram obsoletos.
- Hipóteses do algoritmo devem ser verificadas.

Laboratório 12: tabela hash didática

Construa uma tabela de encadeamento com poucos baldes para produzir colisões visíveis. Cada balde será uma lista de pares. Ao buscar, calcule o índice e percorra apenas o balde correspondente.

Escolha o comportamento para chave repetida. Atualizar o valor costuma ser o que um mapa espera. Se o domínio permite várias ocorrências, o valor pode ser uma lista, mas essa decisão muda o contrato.

Conte colisões e observe como o fator de carga afeta o percurso. A implementação didática não precisa redimensionar na primeira versão; o objetivo é compreender por que bibliotecas fazem isso.

Teste objetos que podem ser chaves e objetos que não podem. A igualdade precisa ser coerente com o hash para que a associação não desapareça entre inserção e busca.

Inserção com atualização. Compare a saída com o contrato descrito no texto anterior.

```
class MapaSimples:
    def __init__(self, quantidade=4):
        self.baldes = [[] for _ in range(quantidade)]

    def _lista(self, chave):
        return self.baldes[hash(chave) % len(self.baldes)]

    def colocar(self, chave, valor):
        balde = self._lista(chave)
        for i, (k, _) in enumerate(balde):
            if k == chave:
                balde[i] = (chave, valor)
                return
        balde.append((chave, valor))
```

Saída esperada

```
Uma chave repetida atualiza seu valor no mesmo balde.
```

Ficha de hash

Use quatro baldes e registre cada colisão.

99. Implemente buscar e remover.
100. Conte colisões por lote.
101. Defina uma política de redimensionamento.
102. Compare com uma lista de pares.

Verificação. Procure uma inicialização que, se estivesse errada, alteraria o resultado.

Síntese do capítulo

- Encadeamento mantém colisões em uma coleção local.
- Chave repetida e valor múltiplo são políticas diferentes.
- Fator de carga orienta redimensionamento.

Laboratório 13: busca e ordenação

Use uma sequência pequena para comparar busca linear e binária. Registre o intervalo considerado pela busca binária depois de cada comparação. O desenho mostra por que a redução pela metade depende da ordenação.

Implemente inserção e verifique o invariante de prefixo ordenado. Em seguida, use sorted apenas para conferir o resultado. A biblioteca é referência de comportamento, não um substituto para a análise.

Para registros, escolha uma chave e documente os empates. Uma ordenação estável pode preservar a ordem original; se isso for importante, crie um teste que o verifique.

Meça com entradas crescentes e separe geração de dados do tempo do algoritmo. Medição não substitui prova, mas pode revelar quando o custo teórico começa a aparecer.

Busca e propriedade de ordenação. Leia o código com uma entrada pequena e tente antecipar a saída.

```
def busca_linear(valores, alvo):
    for indice, valor in enumerate(valores):
        if valor == alvo:
            return indice
    return None

def esta_ordenada(valores):
    return all(a <= b for a, b in zip(valores, valores[1:]))
```

Saída esperada

```
A função esta_ordenada verifica pares consecutivos.
```

Ficha de algoritmos

Registre comparações e tamanho da entrada.

103. Adapte a busca binária para limites.
104. Conte comparações na busca linear.
105. Teste estabilidade com registros.
106. Compare entrada ordenada e inversa.

Depois do teste. Compare uma versão simples com um caso um pouco maior.

Síntese do capítulo

- Busca binária depende de uma propriedade global da entrada.
- Invariantes podem ser expressos por funções de verificação.
- Medição precisa separar preparação e processamento.

Laboratório 14: diagnóstico de desempenho

Escolha uma operação e duas estruturas candidatas. Defina um conjunto de entradas e uma métrica, como tempo, comparações, memória aproximada ou número de deslocamentos. Sem esse planejamento, a comparação fica sujeita a impressões.

Faça várias execuções e observe a tendência, não apenas o menor valor. Sistema operacional, aquecimento e ruído alteram medições pequenas. Um relatório didático informa ambiente, tamanho e método.

Relacione o resultado à análise assintótica. Se uma lista parece suficiente em n pequeno, isso não contradiz que uma busca por chave cresça linearmente. A conclusão precisa indicar para qual faixa e padrão de uso a recomendação vale.

Inclua uma ameaça à validade. Pode ser entrada pouco representativa, implementação simplificada, custo de conversão ou ausência de concorrência. Reconhecer limites melhora a qualidade da análise.

Tabela 12 — Plano de experimento

Item	Registro
pergunta	qual operação está sendo comparada?
entradas	tamanho, ordem e repetição
implementações	estruturas e funções equivalentes
métrica	tempo, comparações ou memória
repetições	quantas medições e como resumir
limitações	o que o experimento não prova

Ficha de desempenho

Escreva o plano antes de medir.

107. Compare busca em lista e conjunto.
108. Compare append e inserção no início.
109. Compare fila com lista e deque.
110. Relacione resultado observado e custo esperado.

Para conferir. Registre a parte da solução que pode ser reutilizada em outro exercício.

Síntese do capítulo

- Experimento precisa de pergunta, entradas e métrica.
- Tendências são mais úteis que uma medição isolada.
- Toda conclusão possui condições e limitações.

Laboratório 15: projeto integrado

Escolha um problema que combine pelo menos duas estruturas. Uma agenda pode usar dicionário e heap; uma rede pode usar grafo, fila e dicionário; um índice textual pode usar dicionário e ordenação. O critério é que cada estrutura tenha uma responsabilidade clara.

Escreva o contrato das operações públicas e liste os invariantes internos. Depois crie testes para inserir, consultar, atualizar, remover e lidar com ausência. A implementação fica mais segura quando os casos de fronteira são planejados antes.

Apresente uma alternativa. Explique por que uma lista simples, embora mais fácil, poderia ficar lenta ou expressar mal a política. Também explique em que tamanho ou cenário a alternativa seria aceitável.

Finalize com um pequeno relatório: modelo, estrutura, algoritmo, custos, testes, exemplo executado e limitações. A documentação é parte da solução de Computação.

Tabela 13 — Entregável do projeto

Seção	Conteúdo mínimo
modelo	entidades, relações e operações
representação	estrutura escolhida e invariante
implementação	funções ou classe com contrato
verificação	testes e resultados esperados
análise	custos e hipóteses
reflexão	alternativa e limitações

Projeto final

Faça uma entrega que outra pessoa consiga executar e revisar.

111. Inclua um exemplo pequeno acompanhado passo a passo.
112. Inclua um caso vazio e um caso de ausência.
113. Mostre uma operação cujo custo justifique a estrutura.
114. Escreva uma melhoria possível para próxima versão.

Como conferir. Antes de executar, calcule o resultado para uma entrada pequena.

Síntese do capítulo

- Estruturas colaboram quando cada uma atende uma operação.
- Contrato e invariante orientam implementação e testes.
- Análise inclui alternativas, hipóteses e limitações.

CAPÍTULO 12

Roteiros de revisão e desafios

Revisar é transformar uma implementação em uma explicação que pode ser conferida.

Desafios de seleção

Para cada situação, escolha uma estrutura e justifique. Não responda apenas com o nome do tipo: indique a operação dominante, o comportamento esperado e o custo que importa. Em caso de empate, compare duas alternativas.

Se a situação permitir mais de uma resposta, declare suas hipóteses. Uma fila de atendimento pode ser FIFO, prioridade ou múltiplos servidores. A estrutura correta depende da política desejada.

Desenhe um exemplo antes de implementar. O desenho deve mostrar o estado inicial, uma operação e o estado final. Em grafos e árvores, inclua relações; em tabelas hash, inclua índices e colisões.

Use as respostas orientativas apenas depois de tentar. A meta é explicar a escolha, não encontrar uma palavra-chave.

Desafios de projeto

Em cada item, escreva estrutura, operação, custo e justificativa.

115. Manter o histórico recente de um editor.
116. Retirar sempre o evento mais urgente.
117. Encontrar amigos a dois passos de uma pessoa.
118. Consultar telefone por nome.
119. Manter valores em ordem e encontrar sucessor.
120. Percorrer dependências de uma disciplina.
121. Retornar os três menores valores.
122. Remover duplicatas preservando ordem.
123. Encontrar caminho mínimo em mapa ponderado.
124. Ordenar registros por duas chaves.

Verificação. Compare uma simulação manual com a execução e registre qualquer diferença.

Síntese do capítulo

- Escolha precisa ser relacionada à operação.
- Hipóteses transformam uma resposta vaga em um contrato.
- Desenhos pequenos tornam a estrutura observável.

Rubrica de domínio

Use a rubrica para acompanhar o aprendizado. Em cada critério, marque 0, 1 ou 2. A pontuação não mede o valor da pessoa; indica qual aspecto precisa de uma nova tentativa.

Se o problema está na escolha, volte ao contrato e às operações. Se está na implementação, desenhe o estado e revise o invariante. Se está na análise, conte operações em uma entrada pequena e depois generalize.

Uma estrutura de dados deve ser explicável em três níveis: comportamento abstrato, representação concreta e custo das operações. O domínio aparece quando os três níveis podem ser relacionados sem confusão.

Finalize cada estudo com uma pergunta aberta. Por exemplo: o que aconteceria se os dados fossem dez vezes maiores? Se a ordem mudasse? Se houvesse empate? Perguntas desse tipo preparam o próximo assunto.

Tabela 14 — Rubrica de domínio

Critério	0	1	2
contrato	ausente	parcial	completo
invariante	não identifica	reconhece	verifica em testes
implementação	não executa	casos comuns	casos de fronteira
complexidade	não analisa	cita notação	relaciona operação e hipótese
escolha	sem justificativa	justifica uma opção	compara alternativas
comunicação	difícil de seguir	descreve passos	explica decisões

Síntese do capítulo

- Domínio conecta contrato, representação e custo.
- Erros indicam o próximo experimento de aprendizagem.
- Justificar alternativas é parte do projeto.

CAPÍTULO 13

Guia de estudo e revisão

O domínio aparece quando a escolha, a implementação e o custo podem ser explicados no mesmo exemplo.

Do tipo abstrato à implementação

Um tipo abstrato de dados descreve operações e comportamento observável. A implementação pode usar lista, nós, árvore ou outra representação. Separar os níveis permite trocar a estrutura sem reescrever o código que depende do contrato.

Comece escrevendo o que uma operação deve fazer para entrada válida, ausência, duplicação e estrutura vazia. Depois escolha uma representação que torne as operações importantes simples. O código deve preservar as propriedades que o contrato pressupõe.

A mesma interface pode esconder custos muito diferentes. Uma operação buscar pode custar O(n) em uma lista, O(log n) em uma árvore equilibrada e O(1) em média em um dicionário. A documentação precisa indicar essas expectativas.

Em uma revisão, pergunte se o usuário da estrutura precisa conhecer nós, baldes ou capacidade. Se a resposta for não, mantenha detalhes internos privados e exponha operações com nomes orientados ao domínio.

Tabela 15 — Dois níveis de descrição

Nível	Pergunta	Exemplo
abstrato	qual comportamento é oferecido?	inserir, remover, buscar
concreto	como o estado é armazenado?	lista, nós ou baldes
análise	como o custo cresce?	O(1), O(log n) ou O(n)

Oficina de abstração

Descreva o contrato sem mencionar a representação.

125. Defina uma fila com operação retirar.
126. Defina um mapa com colocar e buscar.
127. Indique um invariante de árvore.
128. Compare duas implementações possíveis.

Depois do teste. Inclua um caso comum e um caso de fronteira na sua verificação.

Síntese do capítulo

- Contrato e representação são níveis diferentes.
- Custos precisam acompanhar a interface.
- Detalhes internos devem ser protegidos quando possível.

Contratos de operações

Um contrato descreve pré-condições, pós-condições e efeitos. A pré-condição informa o que a operação espera; a pós-condição diz o que será verdadeiro depois; o efeito indica se a estrutura foi modificada. Esse vocabulário reduz ambiguidades.

Para pop, por exemplo, é preciso decidir o comportamento quando a pilha está vazia. Para inserir em uma árvore, é preciso decidir como chaves repetidas serão tratadas. Para buscar em um mapa, é preciso definir ausência.

O contrato também orienta testes. Cada frase pode gerar um caso: entrada válida, vazio, repetição, ausência ou erro. Se uma frase não puder ser testada, talvez ainda esteja vaga.

Documentar contratos no código facilita o uso por outra pessoa. Uma docstring curta, exemplos e tipos aproximados já podem evitar interpretações incompatíveis.

Contrato simples de fila. Antes de executar, acompanhe a mudança dos valores nas linhas principais.

```
class Fila:
  def __init__(self):
    self._dados = deque()

  def entrar(self, valor):
    self._dados.append(valor)

  def sair(self):
    if not self._dados:
      return None
    return self._dados.popleft()
```

Saída esperada

```
sair retorna None quando não há item e mantém a fila consistente.
```

Oficina de contratos

Escreva comportamento de ausência e repetição.

129. Defina contrato para heap mínimo.
130. Defina contrato para remover chave.
131. Defina contrato para caminho inexistente.
132. Crie um teste para cada pós-condição.

Para conferir. Altere os valores do exemplo e confirme se a regra continua válida.

Síntese do capítulo

- Contrato organiza pré-condições e pós-condições.
- Ausência e repetição precisam de política.
- Testes podem ser derivados das frases do contrato.

Verificar invariantes

Estruturas mutáveis acumulam risco porque uma operação incorreta pode danificar operações futuras. Um invariante torna o estado esperado explícito. Na fila, itens estão na ordem de chegada; na lista encadeada, o último próximo é None; no heap, pai respeita prioridade.

Crie funções auxiliares de verificação durante o desenvolvimento. Elas podem percorrer toda a estrutura depois de cada operação. O custo extra é aceitável em testes, mesmo que a verificação não faça parte do caminho de produção.

Quando um teste falhar, descubra qual operação foi a primeira a quebrar o invariante. Isso é mais informativo do que observar somente a resposta final. Um estado inválido cedo pode produzir erros muito depois.

Invariantes não são apenas ferramentas de depuração. Eles ajudam a explicar por que um algoritmo funciona e a revisar se uma otimização preserva o comportamento.

Verificador de heap mínimo. Teste primeiro o caso apresentado; depois altere uma entrada e observe o que muda.

```
def heap_valido(heap):
    for i, valor in enumerate(heap):
        esquerdo = 2 * i + 1
        direito = 2 * i + 2
        if esquerdo < len(heap) and valor > heap[esquerdo]:
            return False
        if direito < len(heap) and valor > heap[direito]:
            return False
    return True
```

Saída esperada

```
A função examina somente relações pai-filho.
```

Oficina de invariantes

Escreva a propriedade antes da função.

133. Verifique uma lista encadeada sem ciclo.
134. Verifique ordem em árvore de busca.
135. Verifique que uma fila preserva os itens.
136. Introduza um defeito e observe o primeiro teste que falha.

Como conferir. Releia o enunciado e verifique se cada condição foi contemplada.

Síntese do capítulo

- Invariante descreve estado válido.
- Verificadores tornam defeitos localizáveis.
- Correção e otimização devem preservar propriedades.

Custo, memória e localidade

Duas estruturas com a mesma ordem assintótica podem apresentar comportamentos práticos diferentes. Uma lista usa posições consecutivas e tende a aproveitar localidade de memória; nós encadeados podem estar espalhados e exigem referências adicionais.

Memória auxiliar também é um recurso. Ordenar uma cópia protege a entrada, mas ocupa espaço; modificar no lugar economiza memória, mas altera o estado recebido. A melhor escolha depende do contrato e da possibilidade de compartilhar dados.

Custo de alocação, conversão e comparação pode dominar em entradas pequenas. Ao medir, declare se esses custos fazem parte da operação. Ao analisar, explique o que foi abstraído.

Quando a entrada cresce, a ordem de crescimento se torna mais informativa. Ainda assim, a conclusão precisa indicar o padrão de acesso. Uma estrutura que é excelente para leitura pode ser ruim para atualizações frequentes.

Tabela 16 — Dimensões da análise

Dimensão	Pergunta

Dimensão	Pergunta
tempo	quantas operações crescem com n?
memória	quantos valores e referências são mantidos?
localidade	os dados ficam próximos na memória?
mutabilidade	a operação altera a entrada?
implementação	qual custo é escondido pela biblioteca?

Oficina de análise

Dê uma conclusão com hipótese e limite.

137. Compare lista e lista encadeada.
138. Compare heap e sequência ordenada.
139. Explique memória de uma tabela hash.
140. Separe custo de conversão do custo da consulta.

Verificação. Acompanhe o contador ou acumulador em uma tabela curta.

Síntese do capítulo

- Tempo não é a única dimensão do custo.
- Localidade e memória influenciam o comportamento observado.
- Conclusões devem declarar hipóteses.

Escolher por padrão de uso

Uma coleção que recebe muitas consultas por chave pede uma estrutura associativa. Uma coleção que recebe muitas operações nas extremidades pede deque. Uma coleção que sempre retira o menor custo pede heap. A pergunta é mais importante que o nome do objeto.

Se o programa precisa percorrer em ordem e procurar por comparação, uma árvore ordenada pode ser natural. Se precisa representar relações arbitrárias, um grafo é mais fiel. Se precisa de histórico recente, pilha comunica a política.

Observe também o que não acontece. Uma lista encadeada não resolve automaticamente uma busca por valor; um heap não permite obter o segundo menor em O(1); um conjunto não preserva a multiplicidade da lista.

Produza uma tabela de decisão para seu próprio projeto. Escreva a alternativa que você descartou e a razão. Esse registro será útil quando o tamanho, a prioridade ou a regra de negócio mudar.

Tabela 17 — Padrões e escolhas

Padrão dominante	Escolha inicial	Armadilha
consultar por chave	dict	assumir ordem total
retirar menor prioridade	heap	esperar sequência ordenada
percorrer relações	grafo	ignorar direção ou peso

Padrão dominante	Escolha inicial	Armadilha
manter histórico recente	pilha	misturar FIFO
acesso por índice	lista	inserir sempre no início

Oficina de escolha

Justifique com operação e invariante.

141. Modele uma fila de impressão.
142. Modele pré-requisitos de disciplinas.
143. Modele catálogo por código.
144. Proponha uma alternativa e seu limite.

Depois do teste. Teste a entrada mínima, vazia ou ausente quando ela fizer sentido.

Síntese do capítulo

- Padrão de uso orienta a estrutura.
- Toda escolha exclui ou dificulta alguma operação.
- Armadilhas fazem parte da justificativa.

Testar estruturas mutáveis

Testar uma estrutura mutável exige olhar para sequências de operações. Uma operação isolada pode funcionar, enquanto duas operações em conjunto expõem uma ligação perdida ou um estado que não foi atualizado.

Comece pelo ciclo vazio, um item e vários itens. Depois teste inserir-remover, inserir-buscar, remover-reinserir e atualização de uma chave. Esses pares exercitam transições comuns.

Use uma implementação de referência simples para validar resultados quando possível. Uma lista comum pode servir para conferir uma fila didática; sorted pode conferir ordenação; um dicionário pode conferir contagem.

Não use a mesma implementação complexa para produzir o resultado esperado. O teste de referência deve ter uma lógica independente, ainda que seja mais lenta.

Teste de transição vazio-um-vazio. Use o trecho para relacionar a regra explicada ao comportamento do programa.

```
fila = deque()
assert not fila
fila.append('A')
assert fila.popleft() == 'A'
assert not fila
```

Saída esperada

```
A sequência verifica o estado antes e depois.
```

Oficina de testes

Escreva testes orientados a sequências.

145. Teste duas inserções e duas remoções.
146. Teste uma chave atualizada.
147. Teste uma lista encadeada com um nó.
148. Compare saída com uma referência simples.

Para conferir. Explique em uma frase por que o resultado atende ao enunciado.

Síntese do capítulo

- Estado mutável deve ser testado em sequência.
- Casos vazios e unitários são essenciais.
- Referência independente reduz falsos testes.

Documentar decisões

Uma documentação técnica deve responder por que a estrutura foi escolhida, quais operações são oferecidas e quais limites existem. Descrever apenas a sintaxe de heappush ou popleft não explica o projeto.

Inclua um exemplo mínimo que mostre a política. Para uma fila, mostre ordem de chegada; para um heap, mostre prioridade; para um grafo, mostre direção ou peso; para uma tabela hash, mostre uma colisão.

Registre versões de bibliotecas apenas quando elas alterarem o contrato ou o resultado. Em material didático, prefira APIs estáveis da biblioteca padrão e indique a documentação oficial como referência.

Documentar limitações evita que uma função seja usada fora do cenário para o qual foi analisada. Isso é especialmente importante em busca binária, Dijkstra e estruturas com custo médio.

Oficina de documentação

Escreva uma página para um projeto do capítulo 10.

149. Inclua uma frase de escolha.
150. Inclua um exemplo executável.
151. Inclua custos e hipóteses.
152. Inclua um caso que não é atendido.

Como conferir. Escolha um valor no limite da regra e confira o comportamento.

Síntese do capítulo

- Documentação deve explicar decisões e limites.
- Exemplos mostram a política em ação.
- Hipóteses protegem o uso correto.

Próximos assuntos

Depois de dominar estruturas básicas, o estudo pode avançar para árvores balanceadas, grafos ponderados, algoritmos de fluxo, análise formal, programação dinâmica e técnicas de projeto de algoritmos. Cada tema acrescenta uma maneira de organizar decisões e custos.

Orientação a objetos pode encapsular invariantes e operações em classes. Banco de dados oferece índices e estruturas persistentes. Sistemas operacionais e redes mostram filas, buffers, tabelas e grafos em contextos reais.

Engenharia de software acrescenta versionamento, revisão, testes automatizados e documentação de interfaces. Esses temas não substituem os fundamentos; eles organizam a colaboração em torno deles.

Leve para qualquer novo assunto o mesmo ciclo: delimitar a pergunta, identificar dados, escolher representação, escrever contrato, testar casos, analisar custo e revisar a explicação.

Tabela 18 — Trilhas de continuidade

Tema	Relação com este livro
algoritmos	prova, projeto e análise de custo
orientação a objetos	encapsulamento de estado e invariantes
bancos de dados	índices, consultas e persistência
sistemas	buffers, filas, memória e escalonamento
engenharia	testes, revisão e manutenção

Plano de continuidade

Escolha uma trilha e formule uma pergunta de estudo.

153. Defina um conceito que deseja aprofundar.
154. Escolha uma implementação para experimentar.
155. Liste uma métrica ou propriedade a verificar.
156. Escreva uma referência técnica para consultar.

Verificação. Procure uma inicialização que, se estivesse errada, alteraria o resultado.

Síntese do capítulo

- Estruturas básicas conectam várias áreas da Computação.
- Novos temas mantêm o ciclo de modelagem e verificação.
- Um plano concreto transforma curiosidade em estudo.

CAPÍTULO 14

Projeto guiado índice de biblioteca

Estruturas diferentes cooperam quando cada uma responde a uma pergunta bem definida.

Pergunta, dados e operações

Objetivos de aprendizagem. Ao concluir esta parte, você deverá conseguir:

- modelar registros e relações entre entidades
- combinar dicionário, conjunto, lista, fila e heap
- especificar operações e invariantes
- testar uma solução integrada e analisar seus limites

O projeto deste capítulo é um pequeno índice de biblioteca. O sistema recebe livros, autores e palavras de busca; depois permite localizar títulos, sugerir itens por prioridade e representar relações simples entre obras. O objetivo é praticar a escolha de estruturas, não construir um catálogo completo.

Antes da implementação, escreva as operações que o usuário realmente precisa: cadastrar uma obra, buscar por palavra, listar por autor e selecionar o próximo item de uma fila de leitura. Cada operação favorece uma representação diferente. A arquitetura nasce desse conjunto de perguntas.

O projeto também delimita o que não será resolvido. Não haverá persistência em banco de dados, busca semântica ou recomendação baseada em histórico real. Limitar o domínio permite observar as estruturas com clareza e medir cada decisão.

Tabela 19 — Operações do índice

Pergunta	Estrutura inicial	Motivo
qual obra possui esta palavra?	dict de conjuntos	acesso por chave
quais obras pertencem ao autor?	dict para lista	agrupamento por chave
qual leitura vem depois?	deque	ordem de chegada
qual item tem maior urgência?	heap	prioridade explícita
quais obras se relacionam?	grafo	vizinhanças

Síntese do capítulo

- Operações frequentes orientam a representação.
- O índice combina estruturas com papéis diferentes.
- Limites explícitos tornam o projeto analisável.

Modelar um índice invertido

Um índice invertido associa cada palavra ao conjunto de obras em que ela aparece. A chave é a palavra normalizada; o valor é um conjunto de identificadores. O conjunto elimina duplicações quando uma palavra aparece várias vezes na mesma obra.

A normalização precisa ser uma etapa do contrato. Converter para minúsculas e remover pontuação pode ser suficiente para o exemplo, mas outras aplicações precisam tratar acentos, hífens e diferentes formas de uma palavra. Não esconda essa decisão dentro da busca.

O dicionário não guarda necessariamente o texto completo. Separar identificadores de registros permite alterar a forma de exibição sem refazer o índice. Essa separação é uma versão pequena da ideia de índice usada em sistemas maiores.

Índice invertido. Faça uma previsão, execute e compare os dois resultados.

```
indice = {}

def adicionar(indice, obra_id, palavras):
    for palavra in palavras:
        indice.setdefault(palavra.lower(), set()).add(obra_id)

adicionar(indice, 1, ["dados", "algoritmos", "dados"])
print(sorted(indice["dados"]))
```

Saída esperada

```
[1]
```

Ficha de indexação

Escreva qual informação deve ser preservada em cada chave.

157. Indexe duas obras com uma palavra em comum.
158. Teste uma palavra ausente.
159. Explique por que o valor é um conjunto.
160. Defina uma política para palavras vazias.

Depois do teste. Compare uma versão simples com um caso um pouco maior.

Síntese do capítulo

- A chave do índice responde à busca.
- Conjuntos evitam repetir a mesma obra.
- Normalização faz parte do resultado esperado.

Agrupar e ordenar resultados

Buscar por palavra devolve identificadores, mas uma aplicação precisa apresentar registros compreensíveis. Um segundo dicionário pode localizar o título e o autor a partir do identificador. A consulta então combina acesso por chave e uma ordenação de saída.

Ordenar o resultado não muda o índice. Essa distinção é importante: o índice atende localização; a lista ordenada atende apresentação. Se a aplicação fizer muitas consultas, a preparação pode ser reaproveitada.

Uma ordenação estável e um critério de desempate tornam a saída previsível. Previsibilidade simplifica testes e reduz a sensação de que o sistema mudou sem motivo.

Resultado ordenado. Observe como a entrada passa por cada etapa até chegar à saída.

```
obras = {
    1: {"titulo": "Algoritmos", "autor": "Ana"},
    2: {"titulo": "Dados", "autor": "Bia"},
}

def detalhes(ids):
    return sorted(
```

```
    (obras[obra_id] for obra_id in ids),
    key=lambda obra: (obra["titulo"], obra["autor"]),
  )

print(detalhes({2, 1}))
```

Saída esperada

```
[{'titulo': 'Algoritmos', 'autor': 'Ana'}, {'titulo': 'Dados', 'autor': 'Bia'}]
```

Ficha de consulta

Separe busca, recuperação do registro e apresentação.

161. Busque por autor.
162. Ordene títulos ignorando maiúsculas.
163. Trate um identificador inexistente.
164. Compare saída ordenada e ordem de cadastro.

Para conferir. Registre a parte da solução que pode ser reutilizada em outro exercício.

Síntese do capítulo

- Índice e apresentação são responsabilidades distintas.
- Critérios de desempate tornam a saída determinística.
- Preparação pode ser reaproveitada em muitas consultas.

Fila de leitura e heap de prioridade

O catálogo pode manter uma fila de leitura para itens adicionados pelo usuário. A política FIFO responde à pergunta: qual foi o primeiro item indicado? Uma fila de prioridade responde a outra: qual item deve ser atendido antes segundo uma regra? Não misture as duas semânticas.

No heap, armazene a prioridade junto com o identificador. A tupla precisa ordenar de modo coerente; se prioridades empatarem, inclua um segundo critério que não dependa de comparar objetos incompatíveis.

A implementação deve documentar se retirar um item o remove da fila, se um item pode ser atualizado e o que acontece quando não existe item. Esses detalhes são o contrato da aplicação.

Tabela 20 — Políticas de retirada

Estrutura	Pergunta	Estado vazio
deque	qual chegou primeiro?	retornar None
heap	qual tem menor prioridade?	retornar None
lista ordenada	qual vem na ordem?	lista vazia

Prioridade com desempate. Acompanhe o fluxo linha a linha e anote o valor que mais lhe interessa.

```
import heapq

proximas = []
heapq.heappush(proximas, (1, 2, "Dados"))
heapq.heappush(proximas, (2, 1, "Algoritmos"))
prioridade, obra_id, titulo = heapq.heappop(proximas)
print(prioridade, titulo)
```

Saída esperada

```
1 Dados
```

Ficha de filas

Escreva a política antes de chamar a operação.

165. Adicione três obras a uma deque.
166. Retire até a estrutura ficar vazia.
167. Crie prioridades empatadas.
168. Explique qual estrutura seria mais simples para cada caso.

Como conferir. Antes de executar, calcule o resultado para uma entrada pequena.

Síntese do capítulo

- FIFO e prioridade respondem a perguntas diferentes.
- O heap precisa de uma chave comparável e documentada.
- Ausência é parte do contrato de retirada.

Representar relações como grafo

Uma relação entre obras pode indicar que dois assuntos são próximos ou que uma leitura é pré-requisito de outra. Uma lista de adjacência representa cada obra e seus vizinhos; ela é econômica quando poucas relações existem.

O grafo deve declarar se as relações são direcionadas. Uma recomendação simétrica pode ser uma aresta em ambos os sentidos; uma dependência possui direção. O algoritmo de busca precisa respeitar essa escolha.

Ao percorrer o grafo, marque os vértices assim que forem descobertos. Essa marca evita ciclos e torna possível reconstruir um caminho por predecessores. A resposta também precisa dizer o que acontece quando o destino não é alcançável.

Busca na rede. O caso é pequeno para que a execução possa ser conferida sem pressa.

```
from collections import deque

rede = {
   1: [2, 3],
   2: [1, 4],
   3: [1],
   4: [2],
}

fila = deque([1])
visitados = {1}
while fila:
   atual = fila.popleft()
   for vizinho in rede[atual]:
      if vizinho not in visitados:
         visitados.add(vizinho)
         fila.append(vizinho)
print(sorted(visitados))
```

Saída esperada

```
[1, 2, 3, 4]
```

Ficha de relações

Desenhe a rede antes de executar a busca.

169. Inclua uma obra isolada.
170. Conte componentes desconectadas.
171. Reconstrua um caminho até um destino.
172. Compare relações direcionadas e não direcionadas.

Verificação. Compare uma simulação manual com a execução e registre qualquer diferença.

Síntese do capítulo

- Lista de adjacência representa vizinhanças com clareza.
- Direção e peso pertencem ao modelo do problema.
- Visitados evitam repetição e ciclos durante a busca.

Testar, medir e explicar

Um projeto integrado deve ser testado por operação e por sequência. Comece com um catálogo vazio, um registro, duplicação e uma consulta sem resultado. Depois combine indexação, busca, fila, prioridade e relações.

Meça apenas depois de definir o que será comparado. O tempo de construir o índice é diferente do tempo de consultar uma palavra. A quantidade de obras, palavras distintas e relações precisa aparecer no relatório.

A explicação final deve ligar escolha, invariante e custo. O dicionário reduz a busca por chave em condições médias; o heap retira a prioridade extrema; a BFS percorre relações alcançáveis. Cada afirmação depende de hipóteses que devem ser registradas.

Tabela 21 — Plano de verificação

Parte	Caso mínimo	Evidência
índice	uma palavra em duas obras	conjunto sem duplicação
consulta	palavra ausente	resultado vazio
fila	um item	ordem de retirada
heap	prioridades empatadas	desempate estável
grafo	vértice isolado	componentes corretas

Entrega do projeto

Prepare uma demonstração de cinco minutos.

173. Mostre o modelo das obras.
174. Execute uma consulta comum e uma vazia.
175. Explique por que cada estrutura foi usada.
176. Informe um custo e uma hipótese.
177. Proponha uma extensão que exigiria outra decisão.

Depois do teste. Inclua um caso comum e um caso de fronteira na sua verificação.

Síntese do capítulo

- Testes integrados devem atravessar operações e estados.

- Medições precisam separar preparação de consulta.
- Uma explicação consistente explicita hipóteses e limites.

Respostas orientativas

As respostas desta seção não substituem o processo de raciocínio. Elas indicam uma direção possível, mas uma solução equivalente pode ser melhor quando apresenta clareza, testes e uma justificativa adequada.

Capítulo 1 — Um tipo abstrato de dados define operações e comportamento; não obriga uma representação específica.

Capítulo 2 — A lista dinâmica favorece acesso por índice; inserções no meio podem deslocar elementos.

Capítulo 3 — A pilha atende LIFO; a fila atende FIFO. Os casos vazios devem ter comportamento definido.

Capítulo 4 — Na remoção, preserve a referência do próximo antes de substituir a ligação do nó anterior.

Capítulo 5 — O percurso em ordem de uma árvore binária de busca deve produzir valores não decrescentes quando repetição é permitida.

Capítulo 6 — Heap mantém apenas a relação entre pai e filhos; por isso a lista interna não é totalmente ordenada.

Capítulo 7 — BFS é adequada para menor número de arestas em grafo não ponderado; pesos podem exigir Dijkstra.

Capítulo 8 — O custo médio O(1) depende de uma função hash razoável e de fator de carga controlado.

Capítulo 9 — Busca binária só é correta enquanto a sequência permanece ordenada.

Capítulo 10 — A estrutura deve ser justificada pelas operações, custos, invariantes e requisitos do projeto.

Glossário essencial

Aresta. relação que conecta dois vértices de um grafo

Árvore. estrutura hierárquica acíclica formada por nós e relações pai-filho

Busca binária. busca que descarta metade de um intervalo ordenado a cada comparação

Cabeça. referência ao primeiro nó de uma lista encadeada

Colisão. situação em que chaves diferentes produzem a mesma posição de hash

Complexidade. medida do crescimento de recursos em função do tamanho da entrada

Deque. fila de duas extremidades com operações eficientes em ambos os lados

Fator de carga. relação entre a quantidade de itens e a capacidade de uma tabela hash

Fila. estrutura cujo acesso típico segue FIFO

Fila de prioridade. estrutura que retira o item de acordo com uma prioridade

Grafo. conjunto de vértices ligados por arestas

Hash. valor calculado a partir de uma chave para apoiar localização

Heap. árvore quase completa que preserva uma relação de prioridade entre pai e filhos

Invariante. propriedade que deve permanecer verdadeira durante as operações

LIFO. política em que o último item inserido é o primeiro a sair

Lista encadeada. sequência formada por nós ligados por referências

Nó. objeto que armazena um valor e, conforme a estrutura, ligações

Recursão. estratégia em que uma definição resolve uma versão menor do próprio problema

Vértice. entidade ou ponto que participa de um grafo

Referências

As referências abaixo foram selecionadas por sua relevância para a formação em Computação e para a documentação da linguagem.

[1] Python Software Foundation. Python 3 Documentation. Acesso

[2] Goodrich, Michael T.; Tamassia, Roberto; Goldwasser, Michael H. Data Structures and Algorithms in Python. Wiley, 2013. Acesso

[3] Cormen, Thomas H.; Leiserson, Charles E.; Rivest, Ronald L.; Stein, Clifford. Introduction to Algorithms. 4. ed. MIT Press, 2022. Acesso

[4] Knuth, Donald E. The Art of Computer Programming, Volume 1: Fundamental Algorithms. 3. ed. Addison-Wesley, 1997. Acesso

[5] Sedgewick, Robert; Wayne, Kevin. Algorithms. 4. ed. Addison-Wesley, 2011. Acesso

[6] Python Software Foundation. collections — Container datatypes. Acesso

[7] Python Software Foundation. heapq — Heap queue algorithm. Acesso

Sobre o autor

Jose Augusto dos Santos Silva é Engenheiro de Computação pelo Instituto de Computação da Universidade Federal de Alagoas. Também é Mestre em Informática pelo Programa de Pós-Graduação em Informática do mesmo Instituto. Sua pesquisa de mestrado concentrou-se na construção e na avaliação de funções racionais candidatas a kernel para Máquinas de Vetor de Suporte, considerando desempenho preditivo, complexidade dos modelos e positividade semidefinida.

Neste livro, essa experiência aparece na ligação entre representação, operações, invariantes e custo, conduzindo o leitor das estruturas básicas a árvores, heaps, grafos e tabelas de espalhamento.

9786583199799ISBN 978-65-83199-79-9

thesis editora científica