Algoritmos e Estruturas de Dados 2
Tabelas de hash, árvores de busca e grafos

Márcio Moretto Ribeiro

12 de novembro de 2025

Conteúdo

Capítulo 1

Introdução

Estruturas de dados e algoritmos são os pilares fundamentais da ciência da computação. Enquanto algoritmos são sequências finitas de instruções bem definidas para resolver um problema ou realizar uma tarefa, estruturas de dados são formas de organizar, armazenar e acessar informações de maneira eficiente. Juntos, eles permitem que possamos resolver problemas complexos de forma sistemática, com clareza, precisão e eficiência. O mesmo problema pode ser abordado por diferentes algoritmos, e a escolha da estrutura de dados adequada muitas vezes determina o desempenho final da solução.

Compreender estruturas de dados e algoritmos é essencial para escrever programas corretos, legíveis e eficientes. Saber quando usar uma tabela de dispersão em vez de uma lista, ou uma árvore balanceada em vez de um vetor ordenado, é o que distingue uma solução ingênua de uma solução robusta e escalável. Além disso, a análise de algoritmos – tanto no pior caso quanto em média – fornece ferramentas para comparar alternativas e justificar decisões de projeto. Estudar esses temas é, portanto, um exercício de precisão lógica, mas também de criatividade na construção de soluções.

Estas notas foram elaboradas para acompanhar a disciplina e pressupõem que os alunos já tenham familiaridade com os fundamentos da análise de algoritmos – como notação assintótica, complexidade de tempo e espaço, e noções básicas de recursão –, conteúdos geralmente abordados na disciplina *Introdução à Análise de Algoritmos*. Também se espera conhecimento prévio sobre estruturas de dados elementares, como vetores, listas ligadas, pilhas e filas, que são introduzidas em *Algoritmos e Estruturas de Dados 1*.

Nesta disciplina, o foco será o estudo de três classes centrais de estruturas de dados: tabelas de dispersão (hashes), árvores de busca e grafos. Esses temas serão tratados do ponto de vista conceitual e prático, com ênfase na análise de desempenho e nas aplicações típicas de cada estrutura. O objetivo é proporcionar uma base sólida que permita aos alunos projetar, implementar

e analisar algoritmos eficientes para problemas que envolvem organização e busca de informação.

Nesta apostila, as estruturas de dados serão apresentadas inicialmente de forma abstrata, por meio de especificações em termos de Tipos Abstratos de Dados (TADs), seguidas por implementações em pseudocódigo e em linguagem C. A descrição abstrata permite compreender a funcionalidade esperada de cada estrutura sem se comprometer com detalhes de implementação, promovendo uma visão mais clara e geral. O pseudocódigo, por sua vez, serve como ponte entre a abstração conceitual e a codificação concreta, permitindo expressar os algoritmos de maneira legível e independente de linguagem. As implementações em C buscam fornecer uma visão prática e eficiente, permitindo ao aluno experimentar diretamente com os conceitos em exercícios e projetos.

Nesta apostila, os conteúdos são organizados em doze capítulos, acompanhando a sequência didática da disciplina. Iniciamos com uma introdução ao conceito de Tipos Abstratos de Dados (TADs), com ênfase no TAD dicionário e em suas duas variações principais: o dicionário simples e o dicionário ordenado. Em seguida, tratamos das tabelas de dispersão e funções de hash, abordando o conceito de função de dispersão, o fator de carga e aplicações como verificação de duplicatas e contagem de frequência.

Os capítulos seguintes se dedicam a técnicas de tratamento de colisões em tabelas de hash, incluindo encadeamento e sondagem linear, com menções a estratégias mais avançadas como sondagem quadrática e duplo hash. Também discutimos o desempenho dessas estruturas, analisando casos médios e piores, e revisitamos listas ligadas como suporte para algumas implementações.

Na segunda parte do curso, o foco se volta para árvores. Iniciamos pelas árvores binárias de busca, estudando sua implementação e uso como dicionários ordenados. Em seguida, apresentamos três variantes de árvores balanceadas: as árvores AVL, as árvores B (adequadas para acesso em disco) e as árvores vermelho-preto, que realizam balanceamento automático por meio de marcações nos nós.

A última parte da apostila é dedicada aos grafos. Apresentamos os conceitos fundamentais e representações possíveis, com exemplos de aplicações reais. Estudamos algoritmos de busca em grafos, como busca em largura (BFS) e busca em profundidade (DFS), e algoritmos para cálculo de caminhos mínimos, incluindo Dijkstra e Bellman-Ford. Encerramos com os algoritmos de Kruskal e Prim para obtenção de árvores geradoras mínimas, fundamentados no teorema do corte.

Objetivos

O principal objetivo desta disciplina é aprofundar o conhecimento dos alunos sobre estruturas de dados e algoritmos, com foco na escolha e implementação de representações eficientes para problemas computacionais complexos. Pretende-se desenvolver a capacidade de raciocínio algorítmico, aliando clareza conceitual à análise crítica de desempenho.

Especificamente, ao final da disciplina, espera-se que o aluno seja capaz de:

- Compreender e implementar estruturas de dados fundamentais como tabelas de dispersão, árvores de busca e grafos;
- Avaliar a eficiência de algoritmos em diferentes contextos, utilizando ferramentas de análise assintótica;
- Reconhecer padrões de problemas e selecionar estruturas e algoritmos adequados para resolvê-los;
- Aplicar os conceitos estudados em implementações práticas, com ênfase na correção, clareza e desempenho do código.

Bibliografia

- Goodrich & Tamassia. *Estruturas de Dados e Algoritmos em Java* (Bookman, 2007).
- Sedgewick & Wayne. *Algorithms, 4th Edition.*

Capítulo 2

Tipos Abstratos de Dados (TADs)

Quando programamos, é comum precisarmos agrupar dados e definir operações que possam ser realizadas sobre eles. Por exemplo, ao trabalhar com uma pilha de objetos, queremos adicionar ou remover um item do topo – e, nesse contexto, o que nos interessa não é como os dados estão armazenados internamente, mas sim o comportamento da estrutura. Essa distinção entre o que uma estrutura faz e como ela faz é o ponto de partida para o conceito de Tipo Abstrato de Dado (TAD).

Um TAD é uma especificação formal de um conjunto de dados e das operações que podem ser realizadas sobre ele, independentemente da forma como essas operações são implementadas. Ao utilizar um TAD, o foco está no comportamento da estrutura – nas operações disponíveis e em suas propriedades –, e não nos detalhes técnicos de código ou armazenamento. Isso permite que o programador raciocine de forma mais clara sobre o problema que está resolvendo, sem se distrair com questões de implementação.

Essa separação traz diversas vantagens. A principal delas é a modularidade: podemos dividir um programa em partes independentes, em que cada módulo (ou TAD) tem uma responsabilidade bem definida. Isso torna o código mais organizado, mais fácil de entender, de testar e de manter. Outra vantagem importante é a reutilização: um TAD bem projetado pode ser empregado em diferentes programas e contextos, sem a necessidade de reescrever sua lógica.

A essência de um TAD está na separação entre a definição lógica da estrutura e sua implementação concreta. Em vez de manipularmos diretamente a forma como os dados estão organizados na memória — como ocorre na programação estruturada tradicional —, interagimos apenas com um conjunto de funções que compõem a interface do TAD. Essas funções são as únicas autorizadas a acessar e modificar a estrutura interna dos dados. Essa restrição é fundamental: garante que todo acesso e modificação passem por um canal

controlado, que pode impor regras e preservar invariantes. Essa abordagem lembra a forma como lidamos com funções em matemática: sabemos o que elas fazem, quais são suas entradas e saídas, mas não precisamos conhecer os detalhes internos de seu funcionamento para utilizá-las corretamente.

Ao longo desta apostila, utilizaremos TADs para representar estruturas como listas, dicionários, filas, árvores e grafos. Em cada caso, começaremos pela especificação abstrata – ou seja, pela definição das operações disponíveis e de como elas devem se comportar – e, em seguida, discutiremos possíveis implementações concretas. Essa metodologia permite entender a estrutura conceitualmente antes de mergulhar nos detalhes técnicos de sua construção.

2.1 Dicionário Simples

O TAD Dicionário representa uma coleção de pares chave–valor. Seu objetivo é permitir o armazenamento e a recuperação eficiente de informações associadas a uma chave. No caso do dicionário simples, as chaves são distintas – ou seja, não há repetições –, e o conjunto de operações é focado na manipulação desse mapeamento básico.

Do ponto de vista abstrato, um dicionário simples oferece as seguintes operações fundamentais:

- `criar()`: cria um dicionário vazio.
- `inserir(d, k, v)`: insere no dicionário `d` o par formado pela chave `k` e o valor `v`. Se a chave `k` já estiver presente, seu valor é substituído por `v`.
- `remover(d, k)`: remove do dicionário `d` o par associado à chave `k`. Se a chave não estiver presente, nenhuma alteração é feita.
- `buscar(d, k)`: retorna o valor associado à chave `k`, caso ela esteja presente no dicionário. Se a chave não existir, a operação indica ausência (por exemplo, retornando `null` ou um valor especial).
- `tamanho(d)`: retorna o número de pares armazenados no dicionário.
- `vazio(d)`: indica se o dicionário está vazio.

Essas operações definem o comportamento esperado da estrutura, sem especificar como os dados são armazenados internamente. A implementação pode variar – por exemplo, pode-se usar listas, tabelas de dispersão (hash) ou árvores de busca –, mas o comportamento das funções deve seguir essa

especificação. Essa separação permite ao programador utilizar o dicionário como uma ferramenta, sem precisar conhecer sua estrutura interna.

Nas próximas seções, estudaremos diferentes formas de implementar esse TAD, analisando os custos de cada operação e as vantagens de cada abordagem conforme o contexto de uso.

2.1.1 Interface em C

A seguir, apresentamos a interface do TAD Dicionário, definida em C como um arquivo de cabeçalho (dicionario.h). Essa interface descreve o conjunto de operações disponíveis para quem deseja utilizar um dicionário simples, sem revelar os detalhes da sua implementação. O uso de tipos opacos e ponteiros para funções permite que diferentes implementações (por exemplo, usando listas, tabelas de dispersão ou árvores) possam ser utilizadas de forma intercambiável, desde que respeitem o mesmo contrato.

```c
#ifndef dic_H
#define dic_H

typedef const char* Chave;
typedef void* Valor;

// Tipo opaco para o dicionário
typedef struct Dicionario Dicionario;

// Interface do TAD
Dicionario* dic_criar();
void dic_inserir(Dicionario* d, Chave k, Valor v);
void dic_remover(Dicionario* d, Chave k);
Valor dic_buscar(Dicionario* d, Chave k);
int dic_tamanho(Dicionario* d);
int dic_vazio(Dicionario* d);
void dic_destruir(Dicionario* d);

#endif
```

Programa 2.1: Interface do TAD Dicionário

Vamos analisar os principais elementos dessa interface:

- `typedef const char* Chave;` define o tipo das chaves como cadeias de caracteres constantes. Em implementações reais, poderíamos generalizar esse tipo com ponteiros para funções de comparação ou usar um tipo genérico.

- `typedef void* Valor;` indica que os valores armazenados são genéricos, acessados como ponteiros opacos. Isso torna o dicionário flexível, podendo armazenar qualquer tipo de dado, desde que convertido para `void*`.

- `typedef struct Dicionario Dicionario;` declara um tipo opaco: o conteúdo interno da estrutura `Dicionario` não é revelado ao usuário do TAD. Isso reforça a ideia de encapsulamento – o usuário interage com o dicionário apenas por meio das funções declaradas.

- `Dicionario* dic_criar();` aloca e retorna um novo dicionário vazio.

- `void dic_inserir(...);` insere um par chave–valor. Se a chave já existir, o valor anterior é substituído.

- `void dic_remover(...);` remove do dicionário o par associado à chave fornecida.

- `Valor dic_buscar(...);` retorna o valor associado a uma chave, ou `NULL` caso a chave não esteja presente.

- `int dic_tamanho(...);` retorna o número de elementos armazenados.

- `int dic_vazio(...);` retorna 1 se o dicionário estiver vazio, ou 0 caso contrário.

- `void dic_destruir(...);` libera toda a memória associada ao dicionário.

Essa interface oferece um conjunto consistente e mínimo de operações para manipular dicionários, mantendo a separação entre a especificação e a implementação. A estrutura interna pode ser alterada sem afetar o código dos usuários do TAD, desde que as funções mantenham o comportamento descrito.

2.1.2 Exemplo de uso

A seguir, mostramos um exemplo simples de como utilizar o TAD Dicionário. O programa cria um dicionário, insere dois pares chave–valor, busca um dos valores, remove uma das chaves e verifica se a remoção foi bem-sucedida. Ao final, a memória alocada é liberada com a operação de destruição.

```c
#include <stdio.h>
#include <stdlib.h>
#include "dicionario.h"

int main() {
    Dicionario* d = dic_criar();

    dic_inserir(d, "chave1", "valor1");
    dic_inserir(d, "chave2", "valor2");

    printf("%s\n", (char*) dic_buscar(d, "chave1")); // imprime valor1

    dic_remover(d, "chave1");

    if (dic_buscar(d, "chave1") == NULL)
        printf("chave1 removida\n");

    dic_destruir(d);
    return 0;
}
```

Programa 2.2: Exemplo de uso do TAD Dicionário

Vamos destacar alguns pontos importantes sobre esse código:

- A função `dic_criar()` aloca dinamicamente um novo dicionário e retorna um ponteiro para ele.
- As strings passadas como chave e valor neste exemplo são literais de string (constantes). Como o tipo `Valor` é genérico (`void*`), precisamos fazer `cast` para `char*` ao imprimir.
- A função `dic_buscar()` retorna o valor associado à chave ou `NULL` se a chave não estiver presente. Isso é usado para verificar se a remoção foi bem-sucedida.
- Por fim, `dic_destruir()` libera toda a memória alocada pelo dicionário.

Esse exemplo mostra como a interface permite manipular o dicionário de forma intuitiva, sem que o usuário precise conhecer a estrutura interna usada para armazenar os dados. A mesma interface pode ser usada com diferentes implementações, o que demonstra na prática a vantagem da abstração promovida pelo TAD.

2.1.3 Implementação com Lista Ligada

Como primeira forma concreta de implementar o TAD Dicionário, utilizamos uma lista ligada simples. Cada elemento da lista armazena um par chave–valor, bem como um ponteiro para o próximo elemento. Essa é uma estratégia simples e direta, adequada para conjuntos pequenos de dados ou para fins didáticos.

Nesta implementação com lista ligada, cada operação é realizada da seguinte forma:

- A inserção de um novo par chave–valor é feita sempre no início da lista.
- Se a chave já existir, o valor correspondente é atualizado e a lista permanece inalterada em estrutura.
- A busca percorre os nós da lista, do primeiro até encontrar a chave desejada ou chegar ao final.
- A remoção também percorre a lista até localizar o nó com a chave desejada, ajustando os ponteiros para removê-lo.
- A estrutura mantém um contador de elementos, permitindo obter o tamanho atual do dicionário de forma direta.
- A verificação de dicionário vazio é feita consultando esse contador.

Essa organização permite uma implementação simples e clara, ideal como primeira aproximação do TAD Dicionário. A estrutura interna permanece escondida do usuário, que interage apenas por meio das funções da interface.

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "dicionario.h"

typedef struct No {
    Chave chave;
    Valor valor;
    struct No* prox;
} No;

struct Dicionario {
    No* primeiro;
    int tamanho;
};
```

```
16
17 Dicionario* dic_criar() {
18     Dicionario* d = malloc(sizeof(Dicionario));
19     d->primeiro = NULL;
20     d->tamanho = 0;
21     return d;
22 }
23
24 void dic_inserir(Dicionario* d, Chave k, Valor v) {
25     No* atual = d->primeiro;
26     while (atual != NULL) {
27         if (strcmp(atual->chave, k) == 0) {
28             atual->valor = v;
29             return;
30         }
31         atual = atual->prox;
32     }
33     No* novo = malloc(sizeof(No));
34     novo->chave = k;
35     novo->valor = v;
36     novo->prox = d->primeiro;
37     d->primeiro = novo;
38     d->tamanho++;
39 }
40
41 void dic_remover(Dicionario* d, Chave k) {
42     No** pp = &d->primeiro;
43     while (*pp != NULL) {
44         if (strcmp((pp)->chave, k) == 0) {
45             No temp = *pp;
46             *pp = temp->prox;
47             free(temp);
48             d->tamanho--;
49             return;
50         }
51         pp = &(*pp)->prox;
52     }
53 }
54
55 Valor dic_buscar(Dicionario* d, Chave k) {
56     No* atual = d->primeiro;
57     while (atual != NULL) {
58         if (strcmp(atual->chave, k) == 0) {
```

```
59              return atual->valor;
60          }
61          atual = atual->prox;
62      }
63      return NULL;
64  }
65
66  int dic_tamanho(Dicionario* d) {
67      return d->tamanho;
68  }
69
70  int dic_vazio(Dicionario* d) {
71      return d->tamanho == 0;
72  }
73
74  void dic_destruir(Dicionario* d) {
75      No* atual = d->primeiro;
76      while (atual != NULL) {
77          No* temp = atual;
78          atual = atual->prox;
79          free(temp);
80      }
81      free(d);
82  }
```

Programa 2.3: Implementação do TAD Dicionário com lista ligada

Essa implementação satisfaz todos os requisitos do TAD Dicionário: encapsula os dados, respeita a interface pública definida no arquivo `dicionario.h`, e pode ser substituída por outra (como uma baseada em tabela de dispersão ou árvore) sem alterar o código dos programas que usam o TAD. Essa separação entre interface e implementação é o que torna os TADs uma ferramenta poderosa na construção de software modular e reutilizável.

2.1.4 Análise de Eficiência da Implementação com Lista Ligada

A implementação do TAD Dicionário com lista ligada é funcional e simples, mas sua eficiência depende diretamente do número de elementos armazenados. A seguir, avaliamos cada operação em termos do custo computacional no pior caso, considerando n como o número de pares armazenados no dicionário.

- `criar()` A criação do dicionário consiste na alocação de uma estrutura inicial com ponteiro nulo e contador zero. **Custo:** $O(1)$

- `inserir(d, k, v)` Para inserir, é necessário primeiro percorrer a lista para verificar se a chave já existe. Se encontrar, o valor é atualizado. Caso contrário, um novo nó é inserido no início. **Custo:** $O(n)$ no pior caso (quando a chave não está presente)

- `remover(d, k)` A remoção percorre a lista até encontrar a chave. Ao localizar o nó, ajusta os ponteiros e libera a memória. **Custo:** $O(n)$ no pior caso

- `buscar(d, k)` A busca percorre os nós da lista, comparando cada chave até encontrar a desejada ou atingir o final. **Custo:** $O(n)$ no pior caso

- `tamanho(d)` O tamanho é mantido por um contador atualizado a cada inserção ou remoção, portanto não exige varredura da lista. **Custo:** $O(1)$

- `vazio(d)` A verificação se o dicionário está vazio consulta apenas o valor do contador. **Custo:** $O(1)$

- `destruir(d)` A destruição percorre toda a lista e libera a memória de cada nó. **Custo:** $O(n)$

A principal limitação dessa abordagem está nas operações que dependem da busca por uma chave (inserção, remoção e busca propriamente dita), que exigem varredura linear da lista. Mais adiante, exploraremos implementações que oferecem desempenho mais eficiente para essas operações, como tabelas de dispersão e árvores de busca.

2.2 Dicionário Ordenado

O TAD Dicionário Ordenado estende o TAD Dicionário Simples com operações que exploram a ordenação das chaves. Assume-se que as chaves pertencem a um conjunto em que existe uma relação de ordem total – isto é, qualquer par de chaves pode ser comparado usando operadores como "menor que", "maior que", ou "igual".

O comportamento esperado das operações é o seguinte:

- `menor(d)`: retorna a menor chave armazenada no dicionário `d`, ou indica ausência se o dicionário estiver vazio.

- `maior(d)`: retorna a maior chave armazenada no dicionário `d`, ou indica ausência se o dicionário estiver vazio.
- `anterior(d, k)`: retorna a maior chave armazenada que seja estritamente menor que `k`, ou indica ausência caso tal chave não exista.
- `posterior(d, k)`: retorna a menor chave armazenada que seja estritamente maior que `k`, ou indica ausência caso tal chave não exista.
- `visitar_em_ordem(d, f)`: percorre o dicionário `d` em ordem crescente de chave e aplica a função `f(k, v)` para cada par chave–valor armazenado.

Essas operações permitem, por exemplo, percorrer os dados em ordem crescente, implementar buscas intervalares (como "todas as chaves entre a e b") e construir algoritmos que dependem de vizinhanças ordenadas. A implementação dessas operações exige estruturas de dados que mantenham a ordem, como árvores binárias de busca.

Interface em C

A interface do TAD Dicionário Ordenado é uma extensão da interface do dicionário simples. Se estivéssemos utilizando uma linguagem orientada a objetos, como Java ou C++, poderíamos definir o dicionário ordenado como uma subclasse do dicionário simples, herdando todas as operações básicas e adicionando apenas as operações específicas de ordenação.

Entretanto, como estamos implementando o TAD em C – uma linguagem que não possui herança nem mecanismos nativos de orientação a objetos – optamos por declarar explicitamente todas as operações, mesmo que algumas já existam na interface do dicionário simples. Essa repetição facilita a organização e deixa clara a assinatura completa da interface para quem vai utilizá-la.

Abaixo está uma possível definição da interface para o TAD Dicionário Ordenado:

```
1 #ifndef dic_ord_H
2 #define dic_ord_H
3
4 typedef const char* Chave;
5 typedef void* Valor;
6
7 typedef struct DicionarioOrdenado DicionarioOrdenado;
8
```

```
// Interface herdada do dicionário simples
DicionarioOrdenado* dic_ord_criar();
void dic_ord_inserir(DicionarioOrdenado* d, Chave k,
    Valor v);
void dic_ord_remover(DicionarioOrdenado* d, Chave k);
Valor dic_ord_buscar(DicionarioOrdenado* d, Chave k);
int dic_ord_tamanho(DicionarioOrdenado* d);
int dic_ord_vazio(DicionarioOrdenado* d);
void dic_ord_destruir(DicionarioOrdenado* d);

// Novas operações específicas da ordenação
Chave dic_ord_menor(DicionarioOrdenado* d);
Chave dic_ord_maior(DicionarioOrdenado* d);
Chave dic_ord_anterior(DicionarioOrdenado* d, Chave k);
Chave dic_ord_posterior(DicionarioOrdenado* d, Chave k);
void dic_ord_visitar_em_ordem(DicionarioOrdenado* d, void
    (*visita)(Chave, Valor));

#endif
```

Programa 2.4: Interface do TAD Dicionário Ordenado

Com essa interface, podemos implementar as funções do TAD Dicionário Ordenado usando diferentes estruturas de dados que mantêm os elementos ordenados — como árvores binárias de busca (que veremos a seguir). O usuário do TAD continua protegido da implementação interna, utilizando apenas os nomes das funções definidas na interface.

2.2.1 Exemplo de uso

A seguir, mostramos um exemplo básico de uso do TAD Dicionário Ordenado. O programa insere alguns pares chave–valor, exibe o menor e o maior elemento, e percorre o dicionário em ordem crescente.

```
#include <stdio.h>
#include "dic_ord.h"

void imprimir(Chave k, Valor v) {
    printf("%s -> %s\n", k, (char*) v);
}

int main() {
    DicionarioOrdenado* d = dic_ord_criar();

```

```
11      dic_ord_inserir(d, "joao", "Aluno");
12      dic_ord_inserir(d, "ana", "Professora");
13      dic_ord_inserir(d, "maria", "Coordenadora");
14
15      printf("Menor chave: %s\n", dic_ord_menor(d));
16      printf("Maior chave: %s\n", dic_ord_maior(d));
17
18      Chave a = "maria";
19      Chave ant = dic_ord_anterior(d, a);
20      Chave pos = dic_ord_posterior(d, a);
21      if (ant) printf("Anterior a %s: %s\n", a, ant);
22      if (pos) printf("Posterior a %s: %s\n", a, pos);
23
24      printf("\nElementos em ordem:\n");
25      dic_ord_visitar_em_ordem(d, imprimir);
26
27      dic_ord_destruir(d);
28      return 0;
29  }
```

Programa 2.5: Exemplo de uso do Dicionário Ordenado

Saída esperada:

```
Menor chave: ana
Maior chave: maria
Anterior a maria: joao
Posterior a maria: (nada)

Elementos em ordem:
ana -> Professora
joao -> Aluno
maria -> Coordenadora
```

Esse exemplo mostra como o TAD Dicionário Ordenado permite não apenas armazenar e recuperar valores por chave, mas também explorar relações de ordem entre as chaves, algo que não seria possível com a versão simples do TAD.

Parte I

Hash

Capítulo 3

Tabela de Dispersão

No capítulo anterior, apresentamos o TAD dicionário, uma estrutura abstrata que associa chaves a valores por meio de operações como inserção, busca e remoção. Implementações simples usando listas ou vetores são funcionais, mas pouco eficientes: no pior caso, é necessário percorrer toda a estrutura para localizar uma chave, resultando em tempo linear por operação.

Tabelas de dispersão – ou tabelas hash – oferecem uma alternativa muito mais eficiente. Utilizando uma função de hash para mapear chaves diretamente a posições de uma tabela, é possível realizar essas operações em tempo constante no caso médio, desde que a função seja bem projetada e o fator de carga esteja controlado.

Essa eficiência torna as tabelas hash ideais para aplicações como:

- verificação de duplicatas (por exemplo, ao filtrar elementos repetidos);
- contagem de frequência (como contar palavras em um texto);
- acesso rápido a dados indexados por identificadores (como nomes de usuário ou códigos de produto).

Tabelas hash estão presentes em praticamente todas as linguagens de programação modernas e são fundamentais para o desenvolvimento de sistemas rápidos e escaláveis. Nesta aula, daremos os primeiros passos para entendê-las.

Uma tabela de dispersão, ou *hash table*, é uma estrutura de dados usada para associar chaves a valores. A ideia central é utilizar uma função de hash para transformar a chave (por exemplo, uma palavra, um número ou um identificador) em um índice de um vetor. Esse índice indica a posição onde o valor correspondente deve ser armazenado ou recuperado.

A estrutura básica consiste em:

- uma *função de hash*, que transforma uma chave em um número inteiro;
- um *vetor* (ou tabela) onde os valores são armazenados em posições indicadas pela função de hash.

Por exemplo, ao registrar a frequência de palavras em um texto, podemos usar a palavra como chave e sua contagem como valor. A função de hash transforma a palavra em um número que indica uma posição na tabela, onde a contagem será atualizada.

Esse mecanismo permite acessar os dados diretamente a partir da chave, sem a necessidade de percorrer toda a estrutura. A ideia é que chaves diferentes sejam distribuídas (ou "dispersadas") ao longo da tabela, ocupando posições distintas.

No entanto, pode acontecer de duas chaves diferentes serem transformadas pela função de hash no mesmo índice. Isso é chamado de **colisão**. Por exemplo, as palavras "`casa`" e "`amor`" podem acabar sendo associadas à mesma posição da tabela. A forma como lidar com colisões será discutida mais para frente no capítulo.

3.1 Dicionário e Tabelas de Hash

Como vimos no capítulo anterior, o ponto de vista abstrato de um dicionário simples envolve operações como `criar()`, `inserir()`, `remover()`, `buscar()`, `tamanho()` e `vazio()`. A ideia é manipular pares `(chave, valor)` de maneira eficiente, sem nos preocupar com os detalhes da implementação.

As tabelas de dispersão oferecem uma maneira concreta e eficiente de implementar esse TAD. A função de hash permite transformar uma chave `k` em um índice de um vetor, e esse índice determina a posição onde o par `(k, v)` será armazenado. Com isso, as operações fundamentais do dicionário podem ser implementadas da seguinte forma:

- `criar()`: aloca um vetor (tabela) inicialmente vazio.
- `inserir(d, k, v)`: aplica a função de hash à chave `k` para determinar a posição na tabela, e insere o par `(k, v)` nessa posição.
- `remover(d, k)`: aplica a função de hash à chave `k` para localizar sua posição na tabela, e remove o par armazenado ali (caso exista).
- `buscar(d, k)`: aplica a função de hash à chave `k` e retorna o valor armazenado na posição correspondente.

As demais operações, como `tamanho()` e `vazio()`, podem ser implementadas com contadores auxiliares mantidos junto à estrutura.

Assim, uma tabela de hash concretiza o TAD dicionário a partir da ideia de associar chaves a posições calculadas por uma função de hash. Essa associação direta evita buscas lineares, tornando as operações mais ágeis, como veremos em detalhes nas próximas seções.

3.2 Funções de Hash

O elemento central de uma tabela de dispersão é a *função de hash*. Ela é responsável por transformar uma chave arbitrária (como um número, uma palavra ou uma estrutura mais complexa) em um número inteiro que servirá como índice da tabela.

Formalmente, uma função de hash é uma função `h(k)` que recebe uma chave `k` e retorna um número inteiro entre `0` e `N - 1`, onde `N` é o tamanho da tabela.

Para que a tabela funcione bem, é importante que a função de hash atenda a alguns requisitos:

- **Determinismo:** a mesma chave deve sempre produzir o mesmo índice. Isso é essencial para que seja possível recuperar os dados corretamente.
- **Distribuição uniforme:** as chaves devem ser espalhadas de forma razoavelmente uniforme pelos índices da tabela, evitando que muitas chaves sejam mapeadas para a mesma posição.
- **Simplicidade computacional:** a função deve ser fácil de calcular, já que será chamada com frequência nas operações de inserção, busca e remoção.

Exemplos simples

- **Hash para inteiros:** uma função simples e comum é usar o operador módulo:

  $$h(x) = x \bmod N$$

  onde `N` é o tamanho da tabela. Por exemplo, se `N = 10` e `x = 37`, temos `h(37) = 7`.

- **Hash para strings:** uma estratégia básica é usar o valor numérico dos caracteres da string (por exemplo, o código ASCII) com pesos:

  $$h(s) = (c_0 + 31 \cdot c_1 + 31^2 \cdot c_2 + \cdots + 31^{n-1} \cdot c_{n-1}) \bmod N$$

onde `c_i` é o código do i-ésimo caractere da string. O número 31 é uma base comumente usada por ser um número primo pequeno.

Esses exemplos ilustram a ideia geral de funções de hash. O tratamento de colisões – quando duas chaves diferentes resultam no mesmo índice – será discutido nas próximas seções.

3.3 Fator de Carga

Ao construir uma tabela de dispersão, é importante controlar a quantidade de elementos que estamos armazenando em relação ao tamanho da tabela. Esse controle é feito por meio de um valor chamado *fator de carga*.

Definição

O fator de carga de uma tabela de hash é definido pela razão:

$$\alpha = \frac{n}{m}$$

onde:

- `n` é o número de elementos armazenados na tabela;
- `m` é o número total de posições disponíveis (ou seja, o tamanho do vetor).

Esse valor nos dá uma ideia de quão "cheia" está a tabela.

Impacto sobre o desempenho

Quanto maior o fator de carga, maior a chance de ocorrerem colisões – ou seja, de duas ou mais chaves serem mapeadas para a mesma posição da tabela. Quando muitas colisões ocorrem, as operações de busca, inserção e remoção tendem a ficar mais lentas, pois passam a depender de mecanismos extras para resolver conflitos.

Por outro lado, manter a tabela com muitas posições vazias (isto é, um fator de carga muito pequeno) pode desperdiçar memória.

Estratégias para manter um bom fator de carga

Para manter o desempenho da tabela em níveis aceitáveis, costuma-se adotar estratégias como:

- **Escolher um tamanho inicial adequado** para a tabela, levando em conta a estimativa de número de elementos.
- **Monitorar o fator de carga** durante as operações, e realizar um *redimensionamento* (ou *rehashing*) quando ele ultrapassa um certo limite.
- **Utilizar tamanhos de tabela primos** pode ajudar a reduzir padrões indesejados na distribuição das chaves.

Nos próximos capítulos, veremos como essas estratégias se aplicam na prática e como diferentes métodos de resolução de colisão se comportam com diferentes fatores de carga.

3.4 Exemplos de Funções de Hash

Para entender na prática como diferentes funções de hash se comportam, realizamos um experimento simples: escolhemos um conjunto realista de chaves e aplicamos diversas funções sobre ele, observando a distribuição resultante em uma tabela de dispersão com tamanho fixo.

O conjunto de chaves: palavras de *O Arquipélago Gulag*

Utilizamos como base o livro *O Arquipélago Gulag*, de Aleksandr Soljenítsin, uma das principais obras do século XX sobre o sistema de campos de trabalho forçado da União Soviética. O livro combina testemunhos pessoais, documentos históricos e análise política, e se tornou um símbolo da denúncia dos regimes totalitários.

A partir do texto completo, extraímos todas as palavras distintas – ao todo, **22.077 palavras únicas**. Esse conjunto é representativo por conter vocabulário natural, com repetições, prefixos, sufixos e estruturas linguísticas recorrentes.

O experimento

Para cada função de hash analisada, aplicamos a seguinte metodologia:

1. Cada palavra foi convertida em um número inteiro usando a função de hash em questão.

2. Esse número foi reduzido ao intervalo $[0, N-1]$ com uma operação de módulo, usando $N = 1000$ (tabela pequena) e $N = 50000$ (tabela grande).

3. Contamos quantas palavras foram mapeadas para cada índice da tabela.

4. Analisamos a distribuição: número de colisões, posições vazias, e o maior acúmulo de palavras em um único índice.

Objetivo

O objetivo do experimento é ilustrar de forma empírica como diferentes funções de hash afetam a distribuição das chaves na tabela. Algumas funções aparentemente simples podem gerar distribuições desbalanceadas, com muitas colisões e uso desigual das posições. Outras, mesmo com implementações simples, apresentam desempenho muito melhor.

Nos exemplos seguintes, comparamos duas funções:

- uma função baseada na **soma dos caracteres** (muito simples, e com desempenho ruim);
- a função **polinomial com base 31**, usada em muitas bibliotecas padrão;

Essa comparação ajudará a visualizar, com dados reais, os impactos concretos de uma boa ou má escolha de função de hash.

Função baseada na soma dos caracteres

A primeira função que analisamos é extremamente simples: ela calcula a soma dos valores numéricos dos caracteres da string, e então aplica o operador módulo com o tamanho da tabela:

$$h(s) = \left(\sum_{i=0}^{n-1} \texttt{ord}(s_i) \right) \bmod N$$

Essa função é fácil de implementar e computacionalmente barata. No entanto, ela apresenta sérios problemas quando aplicada a dados reais como palavras de um texto.

Exemplo 3.4.1. *Considere a palavra* `programador`*. Vamos calcular o valor de hash utilizando a função baseada na soma dos códigos dos caracteres.*

Os valores ASCII dos caracteres são:

Letra	*Código*
p	*112*
r	*114*
o	*111*
g	*103*
r	*114*
a	*97*
m	*109*
a	*97*
d	*100*
o	*111*
r	*114*

Somando os valores:

$$112 + 114 + 111 + 103 + 114 + 97 + 109 + 97 + 100 + 111 + 114 = 1.182$$

Se estivermos usando uma tabela com $N = 1000$ *posições, o valor da função será:*

$$h(\texttt{programador}) = 1182 \mod 1000 = 182$$

Assim, a palavra `programador` *seria armazenada na posição 182 da tabela.*

Esse exemplo mostra que, mesmo com palavras um pouco maiores, a soma raramente ultrapassa 1500 — o que explica por que funções baseadas apenas na soma tendem a ocupar uma faixa muito estreita da tabela.

Resultados com $N = 1000$

Aplicamos essa função às 22.077 palavras distintas de *O Arquipélago Gulag*, usando uma tabela de tamanho $N = 1000$. Os resultados mostraram uma distribuição bastante desigual:

- **17 posições** da tabela ficaram completamente vazias;
- Algumas posições armazenaram mais de **60 palavras diferentes**;
- O número total de colisões foi elevado em comparação com outras funções;
- Muitas palavras distintas foram mapeadas para o mesmo índice.

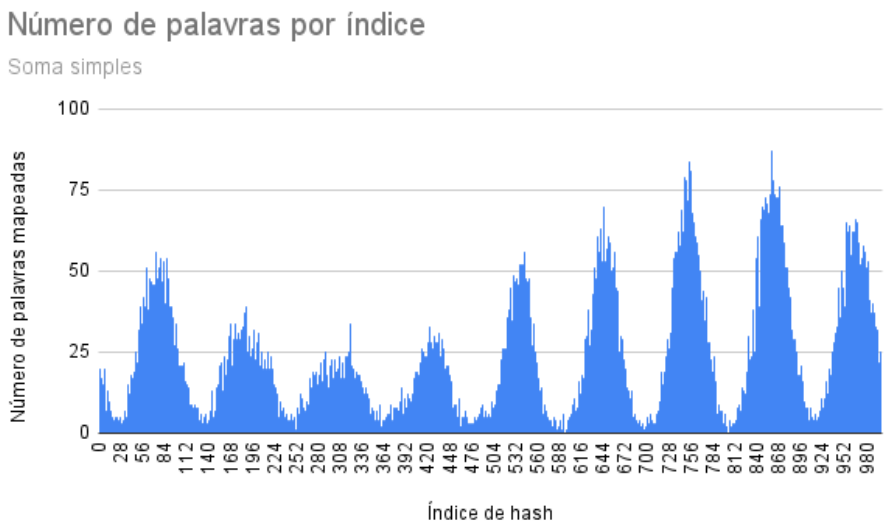


Figura 3.1: Distribuição de palavras distintas por índice da tabela usando a função de hash baseada na soma dos caracteres.

Resultados com $N = 50000$

Repetimos o experimento com a função baseada na soma dos caracteres, agora utilizando uma tabela com $N = 50000$ posições. Apesar do aumento expressivo no tamanho da tabela, os resultados permaneceram insatisfatórios:

- **48.600 posições** da tabela ficaram completamente vazias;
- A maior carga observada em um único índice foi novamente de **87 palavras distintas**;
- O número total de colisões foi de **20.677**, praticamente igual ao observado com $N = 1000$;
- Apenas uma pequena faixa inicial da tabela foi efetivamente utilizada – a maioria dos índices jamais recebeu qualquer palavra.

Esse comportamento ocorre porque a função de hash baseada na soma dos caracteres gera valores relativamente baixos: como a maioria das palavras tem entre 4 e 15 letras e utiliza caracteres com códigos entre 97 e 122 (no caso de letras minúsculas), a soma total raramente ultrapassa 1500. Com isso, todos os valores de hash produzidos se concentram nos primeiros mil ou dois mil índices da tabela. Os demais índices – mais de 90% da tabela

– simplesmente não são utilizados. Isso evidencia uma das principais falhas dessa função: sua incapacidade de aproveitar o espaço disponível, mesmo quando a tabela é grande.

Por que essa função falha?

Apesar de ser válida como função no sentido técnico (é determinística e retorna um valor inteiro), essa função é inadequada para uso prático em tabelas de dispersão. Os principais problemas são:

- **Ignora a ordem dos caracteres:** palavras como `amor` e `roma` têm o mesmo valor de hash.
- **Alta redundância linguística:** palavras com estruturas semelhantes (prefixos, sufixos, letras de mesmo valor) tendem a cair nas mesmas posições.
- **Pouca difusão:** pequenas mudanças na entrada resultam em mudanças pequenas no hash.

Esse exemplo mostra que a escolha da função de hash tem impacto direto sobre a eficiência da tabela: funções simples demais podem gerar um número elevado de colisões, comprometendo o desempenho das operações básicas.

Função polinomial com base 31

A segunda função que analisamos é baseada em um cálculo polinomial clássico, que atribui pesos crescentes aos caracteres da string conforme sua posição:

$$h(s) = (c_0 + 31 \cdot c_1 + 31^2 \cdot c_2 + \cdots + 31^{n-1} \cdot c_{n-1}) \bmod N$$

onde c_i representa o valor numérico do i-ésimo caractere. Essa função leva em conta a ordem dos caracteres, o que a torna mais sensível a variações entre palavras diferentes.

Exemplo 3.4.2. Exemplo ilustrativo

Vamos agora calcular o valor da função de hash polinomial com base 31 para a palavra `programador`*. Essa função considera a ordem dos caracteres e aplica pesos exponenciais:*

$$h(s) = (c_0 + 31 \cdot c_1 + 31^2 \cdot c_2 + \cdots + 31^{n-1} \cdot c_{n-1}) \bmod N$$

Para `programador`*, com* $N = 1000$*, temos os seguintes valores ASCII:*

Letra	*Código*	*Peso*
p	*112*	$31^0 = 1$
r	*114*	$31^1 = 31$
o	*111*	$31^2 = 961$
g	*103*	$31^3 = 29.791$
r	*114*	$31^4 = 923.521$
a	*97*	$31^5 = 28.628.151$
m	*109*	$31^6 = 887.472.681$
a	*97*	$31^7 = 27.511.653.111$
d	*100*	$31^8 = 852.861.246.441$
o	*111*	$31^9 = 26.438.698.639.671$
r	*114*	$31^{10} = 820.599.658.829.801$

Multiplicamos cada caractere por seu respectivo peso e somamos todos os produtos:

$$h(\texttt{programador}) = (112{\cdot}1{+}114{\cdot}31{+}111{\cdot}961{+}\cdots{+}114{\cdot}820.599.658.829.801) \bmod 1000$$

O valor intermediário fica enorme, mas ao aplicar o módulo 1000 em cada etapa (ou no final), obtemos:

$$h(\texttt{programador}) \equiv 722 \mod 1000$$

Esse exemplo mostra como a função polinomial leva em conta tanto os valores dos caracteres quanto sua posição. Palavras com os mesmos caracteres em ordens diferentes terão valores distintos, e o efeito exponencial ajuda a dispersar melhor os dados pela tabela.

Resultados com $N = 1000$

Aplicamos essa função ao mesmo conjunto de 22.077 palavras distintas, utilizando uma tabela de tamanho $N = 1000$. Os resultados foram significativamente melhores do que na função baseada na soma:

- **Nenhuma posição** da tabela ficou vazia;
- A maior carga observada em um único índice foi de **38 palavras distintas**;
- O número total de colisões foi de **21.077**, substancialmente menor do que o número total de entradas (o que é esperado, dado que $n > N$);
- A distribuição dos valores foi mais uniforme, com menos aglomerações extremas.

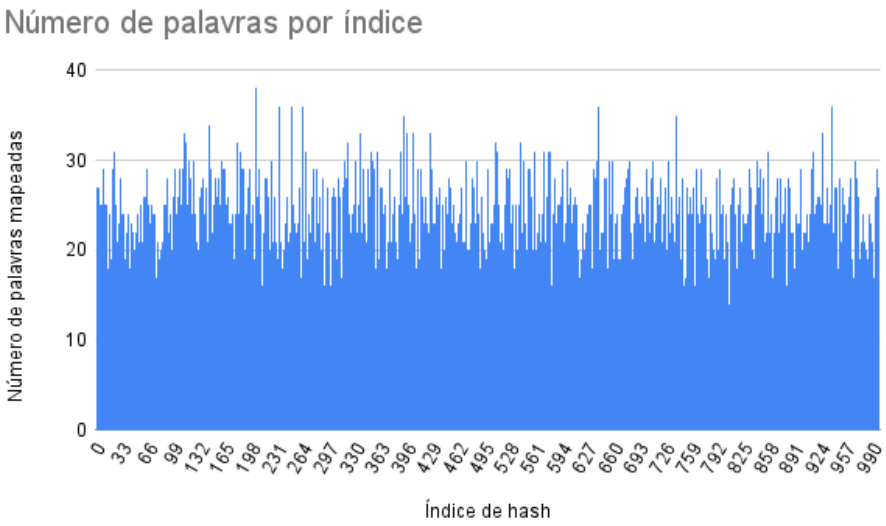


Figura 3.2: Distribuição de palavras distintas por índice da tabela usando a função de hash polinomial com base 31.

Por que essa função funciona melhor?

Comparada à função da soma, a função polinomial apresenta várias vantagens:

- **Considera a ordem dos caracteres:** isso evita que anagramas gerem o mesmo valor de hash.
- **Maior variação:** o uso de potências introduz um efeito de amplificação, mesmo para pequenas mudanças na string.
- **Distribuição mais equilibrada:** o número de posições ocupadas e a dispersão geral das chaves foram significativamente melhores.

Esse experimento mostra que, mesmo com um número de colisões inevitável quando $n \gg N$, uma função de hash bem projetada pode reduzir o impacto dessas colisões ao espalhar melhor os dados pela tabela.

Resultados com $N = 50000$

Repetimos o experimento com a função polinomial base 31 utilizando uma tabela significativamente maior: $N = 50000$.

- O número total de colisões foi de apenas **4.236**;

- A maior carga observada em um único índice foi de **5 palavras distintas**.

Esses resultados mostram como o aumento do tamanho da tabela reduz drasticamente o número de colisões. Embora muitas posições da tabela fiquem vazias – o que é natural quando N é muito maior que o número de elementos – as posições ocupadas recebem em geral apenas uma ou duas palavras. A distribuição resultante é bastante equilibrada, com poucas colisões e sem pontos de concentração.

Capítulo 4

Resolução de Colisões

Como vimos anteriormente, uma função de hash associa uma chave a uma posição em uma tabela, idealmente de forma uniforme e eficiente. No entanto, mesmo com boas funções de hash e tabelas com fator de carga controlado, colisões são inevitáveis: duas ou mais chaves diferentes podem ser mapeadas para o mesmo índice.

Essas colisões não indicam erro ou falha do algoritmo – são uma consequência natural do mapeamento de um conjunto potencialmente muito grande de chaves em um espaço fixo de posições. A ocorrência de colisões exige que a implementação da tabela de dispersão adote estratégias específicas para resolvê-las. O objetivo destas estratégias é garantir que operações como inserção, busca e remoção continuem eficientes, mesmo quando múltiplas chaves disputam o mesmo espaço.

Neste capítulo, vamos estudar as principais abordagens para lidar com colisões: encadeamento (*chaining*), que utiliza listas para armazenar múltiplos elementos em uma mesma posição, e endereçamento aberto (*probing*), que busca novas posições disponíveis na própria tabela. Analisaremos o funcionamento de cada técnica, suas vantagens, limitações e os impactos que causam no desempenho da tabela.

4.1 Encadeamento (Chaining)

Uma das formas mais comuns de lidar com colisões em tabelas de dispersão é por meio do encadeamento. A ideia central dessa abordagem é permitir que múltiplos elementos ocupem a mesma posição da tabela, armazenando-os em uma estrutura auxiliar associada a cada índice – geralmente, uma lista.

Em vez de restringir cada posição da tabela a conter no máximo um único elemento, o encadeamento permite que cada posição funcione como

um "balde", onde várias chaves com o mesmo valor de hash podem ser inseridas. Assim, quando ocorre uma colisão, o novo elemento simplesmente é adicionado à lista existente naquele índice.

Por exemplo, se duas chaves diferentes forem mapeadas para a mesma posição pela função de hash, ambas serão armazenadas na lista associada àquela posição. A operação de busca percorre essa lista em busca da chave desejada; a inserção adiciona um novo elemento à lista; e a remoção procura pela chave e a elimina, se presente.

Essa abordagem é especialmente eficaz quando a função de hash distribui bem as chaves e o fator de carga da tabela permanece controlado. No entanto, se muitas colisões ocorrerem em um mesmo índice, a lista associada pode crescer, e o desempenho se degradar.

A seguir, veremos como essa estratégia é implementada e analisaremos seu impacto sobre o desempenho das operações básicas.

Estrutura de dados

Na estratégia de encadeamento, cada posição da tabela de dispersão aponta para o início de uma lista ligada. Cada elemento dessa lista armazena um **par chave-valor**, além de um ponteiro para o próximo elemento.

Para isso, usamos duas estruturas em `C`:

```c
typedef struct No {
    char* chave;
    int valor;
    struct No* prox;
} No;
```

Programa 4.1: Estrutura de um nó da lista encadeada

Essa estrutura representa um elemento armazenado em uma das listas. Ela contém:

- `chave`: a chave do par (usamos `char*` para permitir chaves do tipo string),
- `valor`: o valor associado à chave (neste exemplo, um inteiro),
- `prox`: ponteiro para o próximo elemento da lista.

A tabela em si é representada por um vetor de ponteiros para `No`, encapsulado em uma estrutura `Dicionario`:

```c
typedef struct {
    No* tabela[TAM];
} Dicionario;
```

Programa 4.2: Estrutura do dicionário com tabela de dispersão

Aqui, `TAM` é o tamanho da tabela (escolhido previamente). Cada entrada da tabela aponta para o início de uma lista de colisões – ou é `NULL` se estiver vazia.

Função `criar`

A função `criar` aloca dinamicamente uma estrutura do tipo `Dicionario` e inicializa todas as posições da tabela com o valor `NULL`, indicando que nenhuma chave foi inserida ainda. Cada posição da tabela está pronta para receber uma lista de colisões quando necessário.

```c
Dicionario* criar() {
    Dicionario* d = malloc(sizeof(Dicionario));
    for (int i = 0; i < TAM; i++)
        d->tabela[i] = NULL;
    return d;
}
```

Programa 4.3: Criação de um dicionário vazio

Essa função:

- Aloca memória para um novo dicionário;
- Inicializa todas as entradas da tabela com `NULL`, ou seja, listas vazias;
- Retorna um ponteiro para a estrutura recém-criada.

A tabela resultante está pronta para que as operações de inserção, remoção e busca sejam aplicadas. O tamanho da tabela, definido pela constante `TAM`, determina o número de "baldes" disponíveis para armazenar listas de colisões.

Função `inserir`

A função `inserir(d, k, v)` insere o par `(k, v)` no dicionário `d`. Para isso, aplica a função de hash à chave `k` para determinar a posição na tabela, e então insere o par na lista ligada correspondente.

Se a chave `k` já estiver presente na lista, seu valor é atualizado.

```c
void inserir(Dicionario* d, const char* k, int v) {
    unsigned int h = hash(k);
    No* atual = d->tabela[h];

    while (atual != NULL) {
        if (strcmp(atual->chave, k) == 0) {
            atual->valor = v;  // substitui o valor existente
            return;
        }
        atual = atual->prox;
    }

    // cria novo nó e insere no início da lista
    No* novo = malloc(sizeof(No));
    novo->chave = strdup(k);
    novo->valor = v;
    novo->prox = d->tabela[h];
    d->tabela[h] = novo;
}
```

Programa 4.4: Inserção de um par (k, v) na tabela

Essa função realiza os seguintes passos:

- Aplica a função de hash à chave `k` para obter a posição `h` na tabela;
- Percorre a lista encadeada na posição `h` em busca de uma chave igual;
- Se encontrar, substitui o valor antigo por `v`;
- Caso contrário, cria um novo nó e o insere no início da lista.

Inserir no início da lista é eficiente e não afeta a corretude, já que a ordem dos elementos na lista não é relevante para a tabela de dispersão.

Função `buscar`

A função `buscar(d, k)` procura pela chave `k` no dicionário `d`. Aplica-se a função de hash à chave para localizar a lista correspondente e percorre-se essa lista à procura de um nó cuja chave seja igual a `k`. Se encontrado, retorna-se um ponteiro para o valor associado; caso contrário, retorna-se `NULL`.

```c
int* buscar(Dicionario* d, const char* k) {
    unsigned int h = hash(k);
    No* atual = d->tabela[h];

    while (atual != NULL) {
        if (strcmp(atual->chave, k) == 0)
            return &atual->valor;
        atual = atual->prox;
    }
    return NULL;
}
```

Programa 4.5: Busca de uma chave na tabela

Essa função:

- Aplica a função de hash à chave `k` para obter a posição `h`;
- Percorre a lista encadeada na posição `h`;
- Se encontrar um nó com chave igual a `k`, retorna o endereço do campo `valor`;
- Se não encontrar, retorna `NULL`.

Como o valor retornado é um ponteiro, o usuário da função pode tanto ler quanto modificar diretamente o valor associado à chave.

Função `remover`

A função `remover(d, k)` remove do dicionário `d` o par cuja chave é `k`, caso ele exista. Para isso, aplica-se a função de hash à chave para localizar a lista apropriada e percorre-se essa lista à procura do nó correspondente. Ao encontrar o nó, ele é retirado da lista e sua memória é liberada.

```c
void remover(Dicionario* d, const char* k) {
    unsigned int h = hash(k);
    No* atual = d->tabela[h];
    No* anterior = NULL;

    while (atual != NULL) {
        if (strcmp(atual->chave, k) == 0) {
            if (anterior == NULL)
                d->tabela[h] = atual->prox;
```

```
10              else
11                  anterior->prox = atual->prox;
12              free(atual->chave);
13              free(atual);
14              return;
15          }
16          anterior = atual;
17          atual = atual->prox;
18      }
19  }
```

Programa 4.6: Remoção de uma chave da tabela

Essa função realiza os seguintes passos:

- Aplica a função de hash à chave `k` para determinar a posição `h`;
- Percorre a lista encadeada na posição `h`, mantendo um ponteiro para o nó atual e outro para o anterior;
- Se encontrar a chave, remove o nó da lista (ajustando os ponteiros) e libera a memória alocada para a chave e para o nó;
- Se a chave não for encontrada, a função não faz nada.

A presença do ponteiro `anterior` é essencial para que seja possível remover nós que não estejam na primeira posição da lista.

4.2 Endereçamento Aberto (Probing)

A segunda abordagem clássica para lidar com colisões em tabelas de dispersão é o **endereçamento aberto** (ou *probing*). Diferentemente do encadeamento, onde cada posição da tabela pode armazenar uma lista de elementos, no endereçamento aberto toda a informação é armazenada diretamente no vetor da tabela, sem o uso de listas auxiliares.

Quando ocorre uma colisão – isto é, quando a posição indicada pela função de hash já está ocupada – o algoritmo busca uma **outra posição livre** dentro da própria tabela, de acordo com uma sequência determinada por uma regra de sondagem (o *probe*).

A ideia é simples: em vez de manter múltiplos elementos por posição, o algoritmo tenta encontrar uma nova posição onde o par chave-valor possa ser inserido. Essa busca por uma nova posição segue uma ordem bem definida, que depende da técnica de probing utilizada (como sondagem linear, quadrática ou duplo hashing).

O mesmo procedimento é adotado para realizar buscas e remoções: a função de hash fornece a posição inicial, e a sequência de probing é seguida até encontrar a chave desejada ou uma posição vazia que indique sua ausência.

Essa estratégia evita a alocação dinâmica de memória e torna a estrutura mais compacta, mas depende de que a tabela tenha **espaço disponível**, ou seja, seu fator de carga deve ser mantido abaixo de um certo limite para garantir bom desempenho.

É importante observar que, no endereçamento aberto, o fator de carga da tabela deve ser estritamente menor que 1. Como não há listas auxiliares, todos os elementos precisam estar contidos diretamente na tabela. Se a tabela estiver completamente cheia, não é possível inserir novos elementos, independentemente da estratégia de sondagem utilizada.

Essa restrição não existe no caso do encadeamento, onde cada posição da tabela pode armazenar múltiplos elementos por meio de listas. Lá, o fator de carga pode inclusive ultrapassar 1 – embora, nesse caso, o desempenho possa piorar progressivamente.

Por isso, ao utilizar probing, é fundamental manter o fator de carga sob controle – muitas implementações realocam (*rehash*) a tabela automaticamente ao atingir, por exemplo, 70% de ocupação.

A forma mais simples de sondagem é a chamada **sondagem linear** (*linear probing*). Nesse método, ao detectar uma colisão na posição h, verifica-se sequencialmente as posições $h + 1$, $h + 2$, $h + 3$, e assim por diante (com aritmética modular), até encontrar uma posição vazia ou a chave procurada. Essa estratégia é fácil de implementar e eficiente com fator de carga baixo, mas pode causar aglomeração de elementos (*clustering*), o que degrada o desempenho das operações.

Além da sondagem linear, existem outras estratégias, como a **sondagem quadrática** e o **duplo hashing**, que serão apresentadas brevemente ao final desta seção.

Função `criar`

A função `criar` aloca uma estrutura de tabela de dispersão usando endereçamento aberto com sondagem linear. Como não há listas auxiliares, a tabela é simplesmente um vetor de ponteiros para pares chave-valor. Cada posição começa como `NULL`, indicando que está livre.

A estrutura usada é levemente diferente da versão com encadeamento: agora, armazenamos diretamente os pares no vetor. Para isso, usamos a seguinte definição:

```
1 typedef struct {
```

```
    char* chave;
    int valor;
} Entrada;

typedef struct {
    Entrada* tabela[TAM];
} Dicionario;
```

Programa 4.7: Estrutura do par chave-valor com marcador de ocupação

Note que o vetor contém ponteiros para `Entrada`. Uma posição `NULL` indica que está livre. Para marcação de remoções (discutida depois), pode-se usar um valor especial (como um ponteiro constante) para distinguir entre posição nunca usada e posição já removida.

A função de criação inicializa todas as posições com `NULL`:

```
Dicionario* criar() {
    Dicionario* d = malloc(sizeof(Dicionario));
    for (int i = 0; i < TAM; i++)
        d->tabela[i] = NULL;
    return d;
}
```

Programa 4.8: Criação de tabela com sondagem linear

Essa estrutura prepara o dicionário para operar com sondagem linear: cada inserção buscará uma posição livre, e cada busca percorrerá posições sucessivas até encontrar a chave ou uma posição vazia.

Função `inserir`

A função `inserir(d, k, v)` aplica a função de hash à chave `k` e, em caso de colisão, procura a próxima posição livre na tabela, avançando sequencialmente até encontrar uma posição disponível ou uma chave já existente (caso em que o valor é substituído).

```
void inserir(Dicionario* d, const char* k, int v) {
    unsigned int h = hash(k);
    for (int i = 0; i < TAM; i++) {
        int pos = (h + i) % TAM;
        Entrada* e = d->tabela[pos];

        if (e == NULL) {
            // posição livre: inserir nova entrada
            e = malloc(sizeof(Entrada));
```

```
            e->chave = strdup(k);
            e->valor = v;
            d->tabela[pos] = e;
            return;
        }

        if (strcmp(e->chave, k) == 0) {
            // chave já presente: substituir valor
            e->valor = v;
            return;
        }
    }

    // tabela cheia (fator de carga >= 1)
    printf("Erro: tabela cheia.\n");
}
```

Programa 4.9: Inserção com sondagem linear

Essa função:

- Calcula a posição inicial com a função de hash;
- Verifica sequencialmente as posições seguintes (com aritmética modular);
- Insere o par na primeira posição livre encontrada;
- Se a chave já existir, substitui o valor antigo;
- Se nenhuma posição estiver livre, exibe uma mensagem de erro.

Função `buscar`

A função `buscar(d, k)` procura pela chave `k` na tabela usando sondagem linear. A partir da posição fornecida pela função de hash, percorre-se sequencialmente a tabela até encontrar a chave ou até chegar a uma posição vazia – o que indica que a chave não está presente.

```
int* buscar(Dicionario* d, const char* k) {
    unsigned int h = hash(k);
    for (int i = 0; i < TAM; i++) {
        int pos = (h + i) % TAM;
        Entrada* e = d->tabela[pos];

```

```
7           if (e == NULL)
8               return NULL; // posição nunca usada: chave nã
                    o está na tabela
9
10          if (strcmp(e->chave, k) == 0)
11              return &e->valor; // chave encontrada
12      }
13
14      return NULL; // chave não encontrada após varrer toda
            a tabela
15  }
```

Programa 4.10: Busca com sondagem linear

Essa função:

- Calcula a posição inicial com a função de hash;
- Percorre posições consecutivas, segundo a regra da sondagem linear;
- Interrompe se encontrar a chave procurada ou uma posição nunca usada;
- Retorna um ponteiro para o valor correspondente, ou `NULL` se a chave não for encontrada.

A verificação de posição vazia é essencial para determinar que a chave não está na tabela. Isso é possível porque posições nunca utilizadas são inicializadas como `NULL` na criação do dicionário.

Função `remover`

A operação de remoção em tabelas com sondagem linear exige cuidado: ao remover uma entrada, **não podemos simplesmente colocar** `NULL` na posição, pois isso interromperia buscas por chaves que foram inseridas após uma colisão e deslocadas para posições seguintes.

Para contornar esse problema, usamos um marcador especial para indicar que a posição foi ocupada no passado mas está **atualmente removida**. Um valor comum para isso é um ponteiro constante, que nunca será confundido com uma entrada válida nem com `NULL`:

```
1  #define REMOVIDO ((Entrada*) -1)
```

Programa 4.11: Marcador especial para posição removida

Com isso, a função de remoção pode ser implementada da seguinte forma:

```
void remover(Dicionario* d, const char* k) {
    unsigned int h = hash(k);
    for (int i = 0; i < TAM; i++) {
        int pos = (h + i) % TAM;
        Entrada* e = d->tabela[pos];

        if (e == NULL)
            return; // posição nunca usada: chave não está na tabela

        if (e != REMOVIDO && strcmp(e->chave, k) == 0) {
            free(e->chave);
            free(e);
            d->tabela[pos] = REMOVIDO;
            return;
        }
    }
}
```

Programa 4.12: Remoção com sondagem linear

Essa função:

- Aplica a função de hash à chave;
- Percorre a tabela segundo a sondagem linear;
- Se encontrar a chave, libera sua memória e marca a posição como `REMOVIDO`;
- Se encontrar uma posição `NULL`, termina a busca (a chave não está presente).

Ao marcar a posição como `REMOVIDO`, mantemos o funcionamento correto das buscas futuras, que devem continuar além dessa posição ao procurar por chaves deslocadas por colisão.

Na prática, também é possível reaproveitar posições marcadas como `REMOVIDO` durante inserções – o que melhora o uso do espaço –, mas essa otimização não está incluída aqui para manter o código simples.

Outras formas de sondagem

A sondagem linear é simples e eficiente enquanto o fator de carga da tabela permanece baixo. No entanto, ela sofre de um problema conhecido como **clustering primário**. Esse fenômeno ocorre quando várias chaves colidem na mesma região da tabela e passam a ocupar posições consecutivas.

Como a sondagem linear sempre verifica posições adjacentes em caso de colisão, uma sequência de elementos tende a crescer como um bloco contínuo. Quando novas colisões ocorrem, é mais provável que elas aconteçam dentro desse bloco – o que aumenta o número médio de tentativas necessárias para inserção e busca. Esse acúmulo progressivo de elementos em regiões contíguas da tabela pode degradar o desempenho da estrutura significativamente, mesmo quando a tabela ainda não está cheia.

Para reduzir esse efeito, foram propostas estratégias de sondagem que espalham melhor as colisões pela tabela. As duas mais conhecidas são a **sondagem quadrática** e o **duplo hashing**.

Na **sondagem quadrática**, as posições alternativas são calculadas com base em deslocamentos quadráticos em relação à posição original. Ou seja, se a função de hash retorna a posição h, as próximas posições a serem testadas são:

$$h + 1^2, \quad h + 2^2, \quad h + 3^2, \quad \ldots$$

Em termos práticos, a i-ésima tentativa verifica a posição $(h+i^2) \bmod m$, onde m é o tamanho da tabela. Essa estratégia evita a formação de *clusters* lineares como ocorre na sondagem linear, pois espaça mais as tentativas e distribui os elementos de maneira mais uniforme.

No entanto, a sondagem quadrática apresenta duas limitações importantes:

- Nem sempre garante que todas as posições da tabela serão visitadas, o que pode impedir a inserção de novos elementos mesmo antes de a tabela estar completamente cheia.
- Requer que o tamanho da tabela obedeça a certas propriedades (como ser um número primo ou uma potência de dois com ajustes) para garantir boa cobertura.

O **duplo hashing** utiliza uma segunda função de hash para definir o deslocamento entre uma tentativa e outra. A sequência de posições testadas é dada por:

$$h + i \cdot h_2(k) \bmod m$$

onde h é o valor da primeira função de hash, $h_2(k)$ é a segunda função de hash aplicada à chave k, e i é o número da tentativa.

A principal vantagem do duplo hashing é a sua capacidade de distribuir as colisões de forma menos previsível e mais uniforme pela tabela, reduzindo significativamente o problema do *clustering*. Além disso, com uma boa escolha da segunda função de hash (por exemplo, garantindo que ela nunca retorne zero), o método pode percorrer toda a tabela antes de repetir posições.

A desvantagem é que o cálculo da segunda função de hash pode aumentar o custo computacional das operações, além de exigir maior cuidado no projeto das funções para evitar ciclos ou repetições.

Essas técnicas são especialmente úteis quando se deseja melhorar o desempenho de tabelas com fator de carga mais alto, desde que a tabela seja redimensionada adequadamente antes de atingir a saturação.

Clustering

Para ilustrar o problema do *clustering* e comparar as estratégias de sondagem, realizamos um experimento empírico utilizando as palavras distintas extraídas do livro *O Arquipélago Gulag*, de Aleksandr Soljenítsin. O total de palavras distintas foi de 22.077, e elas foram inseridas em uma tabela de dispersão de tamanho $m = 50.000$ utilizando três métodos distintos: sondagem linear, sondagem quadrática e duplo hashing.

Usamos a função de hash polinomial com base 31 para calcular a posição inicial de cada palavra. No caso do duplo hashing, uma segunda função de hash foi obtida aplicando novamente a função polinomial à chave original.

Após a inserção de todas as palavras, analisamos a distribuição das posições ocupadas na tabela. Em particular, medimos o número de blocos consecutivos de posições ocupadas com comprimento superior a certos limiares (10, 20, 30, 40 e 50 posições). Os resultados obtidos estão resumidos na tabela a seguir:

Método	**>10**	**>20**	**>30**	**>40**	**>50**
Sondagem Linear	140	41	18	12	8
Sondagem Quadrática	138	27	7	6	4
Duplo Hashing	85	3	0	0	0

Os resultados demonstram de forma clara o impacto do *clustering primário* na sondagem linear. Mesmo com fator de carga abaixo de 50%, esse método produziu diversos blocos com mais de 50 posições consecutivas ocupadas, tornando a tabela densamente preenchida em certas regiões.

A sondagem quadrática melhora essa situação, reduzindo significativamente o número de blocos longos, embora ainda apresente alguma aglomeração local. Já o duplo hashing praticamente elimina o fenômeno: não se observa nenhum bloco com mais de 30 posições consecutivas ocupadas, indicando uma distribuição muito mais uniforme e eficiente.

A estratégia adotada para lidar com colisões varia entre linguagens de programação e suas bibliotecas padrão. No caso da linguagem **Java**, a classe `HashMap` utiliza por padrão a técnica de **encadeamento** (*chaining*), armazenando listas de pares chave-valor em cada posição da tabela. Desde o Java 8, quando uma lista encadeada cresce além de um certo limite (por padrão, 8 elementos), ela é automaticamente convertida em uma árvore balanceada (veremos isso nos capítulos adjacentes), melhorando o desempenho no pior caso[1].

Em contraste, a linguagem **Python** implementa seus dicionários (`dict`) e conjuntos (`set`) com base em **endereçamento aberto** (*open addressing*), resolvendo colisões por meio de **sondagem linear modificada**. A partir da versão 3.6, os dicionários passaram a preservar a ordem de inserção dos pares chave-valor, sem alterar o mecanismo fundamental de resolução de colisões.

A implementação do `dict` em CPython utiliza deslocamentos calculados a partir do hash da chave, com ajustes dinâmicos e política de inserção que favorece a ocupação de posições mais próximas à posição ideal. Essa abordagem é inspirada em técnicas como o *robin hood hashing*, nas quais elementos com maior "idade de inserção" podem ser mantidos em posições melhores em detrimento de elementos mais recentes. Essa estratégia melhora a distribuição das colisões, reduz o *clustering* primário e favorece a localidade de memória, contribuindo para o desempenho eficiente mesmo sob fatores de carga elevados.

[1] `https://www.javapedia.net/Map-and-its-implementations/2606`

Capítulo 5

Análise de Desempenho dos Métodos de Hash

Neste capítulo, analisaremos o desempenho das principais estratégias de implementação de tabelas de dispersão, com ênfase nas operações de inserção e busca. Consideraremos tanto o **pior caso** quanto o **caso médio**, comparando os métodos de encadeamento e de endereçamento aberto sob diferentes condições de uso.

A análise será feita utilizando a **notação assintótica** Θ, que descreve a ordem de crescimento do tempo de execução das operações em função do número de elementos armazenados. Quando dizemos que uma operação tem custo $\Theta(n)$, queremos dizer que o tempo de execução cresce proporcionalmente a n, tanto como limite superior quanto inferior. De forma análoga, $\Theta(1)$ indica que o tempo de execução permanece constante, independentemente do tamanho da entrada.

Hipótese da função de hash uniforme: Para todas as análises de caso médio apresentadas neste capítulo, assumiremos que a função de hash utilizada distribui as chaves de forma **uniforme** e **aleatória** entre as posições da tabela.

Seja n o número total de elementos armazenados na tabela e m o tamanho da tabela (isto é, o número de posições disponíveis). O **fator de carga** é definido por:

$$\alpha = \frac{n}{m}$$

Esse fator expressa a média de elementos por posição da tabela. No caso de encadeamento, representa o comprimento médio das listas associadas a cada posição. No caso de endereçamento aberto, indica a proporção da tabela que está ocupada.

5.1 Encadeamento

No encadeamento, cada posição da tabela aponta para uma lista encadeada contendo os pares chave–valor cujas chaves foram mapeadas para aquela posição pela função de hash. Essa abordagem é simples, flexível e lida bem com colisões, desde que o fator de carga seja mantido sob controle.

Custo de busca

A seguir, analisamos o custo da operação de busca, tanto no caso médio quanto no pior caso. Para essa análise, usamos a notação Θ para indicar a ordem exata de crescimento do tempo de execução em função do número de elementos armazenados.

Sob a hipótese de que a função de hash distribui as chaves uniformemente entre as m posições da tabela, temos:

- **Busca bem-sucedida:** a chave está presente e espera-se que esteja distribuída uniformemente. Nesse caso, o custo médio é

$$\Theta\left(1 + \frac{\alpha}{2}\right)$$

  pois percorremos, em média, metade da lista associada à posição. Em particular, se garantimos que $n = O(m)$, ou seja, o número de elementos cresce no máximo linearmente com o tamanho da tabela, o custo médio torna-se

$$\Theta(1)$$

- **Busca mal-sucedida:** a chave não está presente e percorremos toda a lista daquela posição para verificar sua ausência. O custo médio é

$$\Theta(\alpha)$$

No pior caso, todas as n chaves colidem na mesma posição da tabela, formando uma única lista encadeada. A busca, nesse cenário, precisa percorrer a lista inteira:

$$\Theta(n)$$

5.1.1 Custo de inserção

A inserção em tabelas com encadeamento consiste em adicionar um novo par chave–valor na lista associada à posição calculada pela função de hash. Se a chave já estiver presente, o valor é atualizado; caso contrário, um novo nó é adicionado.

Assumindo novamente uma distribuição uniforme das chaves, espera-se que o comprimento médio de cada lista seja proporcional ao fator de carga α. Como a inserção é realizada no início da lista (sem necessidade de percorrê-la), o custo da operação é constante:

$$\Theta(1)$$

No pior caso, todas as chaves colidem na mesma posição da tabela, formando uma única lista encadeada de comprimento n. Se for necessário verificar a existência prévia da chave antes de inserir (como em um dicionário), é preciso percorrer toda a lista. O custo, portanto, é:

$$\Theta(n)$$

5.2 Endereçamento Aberto

No endereçamento aberto, todos os elementos são armazenados diretamente na tabela. Quando ocorre uma colisão, a função de hash é ajustada de forma sistemática para sondar outras posições disponíveis. Neste modelo, não há listas auxiliares: todas as chaves ocupam posições dentro do vetor principal.

No *linear probing*, em caso de colisão, procuramos sequencialmente pela próxima posição livre na tabela.

Caso médio

Assumindo que as chaves são distribuídas de forma uniforme e aleatória e que a tabela ainda não está muito cheia (isto é, $\alpha < 1$), temos os seguintes custos médios para busca no *linear probing*, conforme a análise clássica de Knuth[1]:

- **Busca mal-sucedida:**

$$\Theta\left(\frac{1}{2}\left(1 + \frac{1}{(1-\alpha)^2}\right)\right)$$

[1]Donald E. Knuth. *The Art of Computer Programming*, Volume 3: *Sorting and Searching*, 2nd ed., Addison-Wesley, 1998. Seção 6.4.

- **Busca bem-sucedida:**

$$\Theta\left(\frac{1}{2}\left(1+\frac{1}{1-\alpha}\right)\right)$$

Não desenvolveremos aqui a demonstração dessas expressões, mas elas ilustram como o custo médio de busca cresce rapidamente à medida que o fator de carga α se aproxima de 1. Em particular, o desempenho se mantém eficiente apenas enquanto α for significativamente menor do que 1.

5.3 Comparação dos métodos

As Tabelas a seguir resumem os custos médios e de pior caso das operações de busca e inserção nas principais estruturas discutidas. A lista ligada é apresentada como referência: embora tenha inserção eficiente no início da lista, sua busca é linear no número de elementos. O encadeamento oferece inserção rápida e desempenho de busca que depende do fator de carga $\alpha = n/m$; quando esse fator é mantido constante, tanto a busca quanto a inserção apresentam custo constante no caso médio. No endereçamento aberto com sondagem linear, o custo médio das operações cresce rapidamente à medida que α se aproxima de 1, conforme indicado pelas expressões assintóticas. Em todos os métodos, o pior caso pode chegar a $\Theta(n)$, o que reforça a importância de uma boa função de hash e do controle rigoroso do fator de carga.

Tabela 5.1: Desempenho médio das operações

Estrutura	**Busca (média)**	**Inserção (média)**
Lista ligada	$\Theta(n)$	$\Theta(1)$
Encadeamento	$\Theta(1+\alpha)$	$\Theta(1)$
Sondagem linear	$\Theta\left(\frac{1}{1-\alpha}\right)$	$\Theta\left(\frac{1}{1-\alpha}\right)$

Tabela 5.2: Desempenho no pior caso

Estrutura	**Busca (pior)**	**Inserção (pior)**
Lista ligada	$\Theta(n)$	$\Theta(1)$
Encadeamento	$\Theta(n)$	$\Theta(n)$
Sondagem linear	$\Theta(n)$	$\Theta(n)$

Capítulo 6

Exercícios

Exercício 6.1. *Considere uma tabela de dispersão com $m = 13$ posições e $n = 9$ elementos.*

1. *Calcule o fator de carga α.*
2. *Supondo uso de encadeamento, qual é o comprimento médio das listas?*
3. *O que aconteceria com α e com o desempenho das operações se inserirmos mais 10 elementos sem aumentar o tamanho da tabela?*

Exercício 6.2. *Explique o papel da função de hash no desempenho das tabelas de dispersão.*

1. *O que significa dizer que uma função de hash é "uniforme"?*
2. *Dê um exemplo simples de função de hash não uniforme e explique por que ela pode causar degradação de desempenho.*
3. *Qual a relação entre a qualidade da função de hash e o pior caso de busca?*

Exercício 6.3. *Compare as estratégias de resolução de colisão.*

1. *Descreva a diferença entre encadeamento e endereçamento aberto com sondagem linear.*
2. *Por que o encadeamento é considerado mais robusto em cenários de alto fator de carga?*
3. *O que é* clustering primário *e como ele afeta a sondagem linear?*

Exercício 6.4. *Considere um sistema com sondagem linear e fator de carga $\alpha = 0{,}75$.*

1. *Estime o custo médio de uma busca bem-sucedida e uma mal-sucedida com base nas fórmulas de Knuth (não é necessário justificar a fórmula).*

2. *E se* $\alpha = 0{,}95$*?*

3. *Por que o controle de* α *é mais crítico em sondagem linear do que em encadeamento?*

Exercício 6.5. *Analise os seguintes cenários de desempenho:*

1. *Qual é o custo médio da busca bem-sucedida em encadeamento com* $\alpha = 3$*?*

2. *Qual é o pior caso para busca usando qualquer método de dispersão?*

3. *Explique como o uso de rehashing pode garantir que o custo médio de busca permaneça constante.*

Parte II

Árvores de Busca

Capítulo 7

Árvores Binárias de Busca

Uma **árvore binária** é uma estrutura de dados hierárquica em que cada elemento, chamado de *nó*, possui no máximo dois filhos: um à esquerda e outro à direita. Como nas tabelas de dispersão, cada nó pode conter uma chave (e eventualmente um valor associado), além de ponteiros para seus filhos esquerdo e direito. A estrutura é naturalmente recursiva: cada subárvore também é uma árvore binária.

As chaves em uma ABB devem ser **comparáveis** – ou seja, é necessário poder determinar se uma chave é menor, igual ou maior que outra. Essa exigência é fundamental porque a estrutura da ABB depende de comparações para decidir a posição de cada chave.

Propriedade fundamental das ABBs: Para cada nó com chave k, todas as chaves da *subárvore esquerda* devem ser *estritamente menores* que k, e todas as chaves da *subárvore direita* devem ser *estritamente maiores* que k.

Essa organização garante que um percurso em ordem (esquerda $\rightarrow$ raiz $\rightarrow$ direita) produza as chaves em ordem crescente, o que permite a implementação eficiente de operações como busca, inserção, remoção e iteração ordenada. O fato de que as chaves são comparáveis torna possível usar ABBs para implementar um dicionário ordenado, em que é possível não apenas recuperar valores associados a chaves específicas, mas também navegar pelas chaves em ordem, encontrar o menor ou maior elemento, ou consultar faixas de valores.

As ABBs são utilizadas em contextos onde a ordenação é importante ou onde há necessidade de navegação eficiente por faixas de valores. Exemplos comuns incluem:

- Implementação de **dicionários ordenados**

- Sistemas de **banco de dados** que requerem ordenação ou busca por intervalo
- **Sistemas de arquivos**, para indexação de nomes ou blocos
- Estruturas auxiliares em algoritmos de compressão, compiladores e engines de jogos

7.1 Estrutura

Uma árvore binária de busca é composta por nós que armazenam três informações principais:

- uma **chave**, usada para determinar a posição do nó na árvore com base em comparações;
- um **valor** associado à chave, como em um dicionário;
- dois ponteiros, para os **filhos esquerdo e direito**.

Em C, essa estrutura pode ser representada por um `struct` da seguinte forma:

```
1 typedef struct no {
2     int chave;
3     void* valor;
4     struct no* esq;
5     struct no* dir;
6 } No;
```

Programa 7.1: Estrutura de um nó de ABB

O campo `chave` é usado para comparações, o campo `valor` pode apontar para qualquer tipo de dado associado, e os campos `esq` e `dir` representam, respectivamente, os filhos à esquerda e à direita do nó.

O ponteiro para a raiz da árvore representa a ABB como um todo. Todas as operações – como busca, inserção e remoção – são executadas a partir desse ponteiro, percorrendo a árvore com base na comparação das chaves.

Essa estrutura recursiva permite representar árvores de qualquer tamanho, e sua organização garante que operações fundamentais possam ser implementadas de forma eficiente.

7.2 Função `buscar`

Dada uma árvore binária de busca e uma chave, a operação de `buscar` percorre a árvore a partir da raiz comparando a chave desejada com a chave do nó atual:

- Se a chave for igual à do nó atual, o valor associado é retornado.
- Se for menor, a busca continua na subárvore esquerda.
- Se for maior, a busca continua na subárvore direita.

A seguir apresentamos a **versão recursiva** do algoritmo. Também é possível implementar a operação de forma iterativa, com uma estrutura de repetição.

```
1  void* buscar(No* raiz, int chave) {
2      if (raiz == NULL)
3          return NULL;
4      if (chave == raiz->chave)
5          return raiz->valor;
6      if (chave < raiz->chave)
7          return buscar(raiz->esq, chave);
8      else
9          return buscar(raiz->dir, chave);
10 }
```

Programa 7.2: Busca recursiva em ABB

O tempo de execução da operação de busca em uma árvore binária de busca é proporcional à altura da árvore. Em outras palavras, a busca sempre toma tempo $\Theta(h)$, onde h representa a **altura da árvore**.

A altura de uma árvore é definida como o número de arestas no caminho mais longo entre a raiz e uma folha. Em uma árvore com apenas um único nó (a raiz), a altura é zero. Se a árvore tem dois níveis, com a raiz e dois filhos diretos, sua altura é 1. Intuitivamente, quanto mais "profunda" for a árvore, maior será a quantidade de comparações necessárias em uma busca.

No **pior caso**, a altura h pode ser igual a $n - 1$, quando a árvore está completamente desbalanceada – por exemplo, quando todas as chaves são inseridas em ordem crescente ou decrescente. Nessa situação, a árvore degenera em uma lista encadeada, e a busca tem complexidade $\Theta(n)$.

No **caso médio**, assumimos – para fins de análise teórica - que as chaves são inseridas em ordem aleatória e com distribuição aproximadamente

uniforme. Sob essa hipótese, a árvore resultante tende a ficar razoavelmente equilibrada.

Em uma árvore binária perfeitamente equilibrada, cada nível da árvore contém o dobro de nós do nível anterior:

- nível 0 (a raiz): 1 nó
- nível 1: até 2 nós
- nível 2: até 4 nós
- nível 3: até 8 nós
- ...
- nível h: até 2^h nós

Se somarmos o número total de nós da árvore até o nível h, temos:

$$1 + 2 + 4 + \cdots + 2^h = 2^{h+1} - 1$$

Isso significa que uma árvore com altura h pode conter até $2^{h+1} - 1$ nós. Invertendo essa relação, o número de níveis necessário para acomodar n nós é aproximadamente $\log_2 n$. Assim, dizemos que uma árvore binária bem equilibrada tem altura $h = \Theta(\lg n)$.

Ou seja, se assumirmos que a ordem de iserção de chaves em um ABB foi feita de maneira aleatória – algo difícil de garantir na prática – sua altura é $\Theta(lg(n))$ e, consequentemente, a operação de busca é $\Theta(lg(n))$.

Portanto, embora a análise do caso médio sob inserção aleatória nos leve a uma altura logarítmica, essa não é uma suposição realista para aplicações em que os dados têm estrutura ou ordenação previsível. Nessas situações, o desempenho pode se degradar para o pior caso, com altura $\Theta(n)$. Para garantir bom desempenho independentemente da ordem de inserção, é necessário usar árvores com balanceamento automático que veremos mais a frente na apostila.

7.3 Função `inserir`

A operação de inserção em uma árvore binária de busca segue a mesma lógica da busca: percorremos a árvore comparando a nova chave com as chaves já presentes, até encontrar uma posição vazia (isto é, um ponteiro nulo) onde o novo nó deve ser inserido.

Se a chave já existe na árvore, podemos optar por atualizar o valor associado. Caso contrário, criamos um novo nó com a chave e o valor fornecidos, e o inserimos como filho de um nó existente.

A seguir apresentamos a versão recursiva da inserção:

```
No* inserir(No* raiz, int chave, void* valor) {
    if (raiz == NULL) {
        No* novo = malloc(sizeof(No));
        novo->chave = chave;
        novo->valor = valor;
        novo->esq = NULL;
        novo->dir = NULL;
        return novo;
    }

    if (chave == raiz->chave)
        raiz->valor = valor; // atualiza
    else if (chave < raiz->chave)
        raiz->esq = inserir(raiz->esq, chave, valor);
    else
        raiz->dir = inserir(raiz->dir, chave, valor);

    return raiz;
}
```

Programa 7.3: Inserção recursiva em ABB

Assim como na busca, a complexidade da inserção depende da altura da árvore. Em geral, a operação percorre o caminho da raiz até uma folha, realizando comparações até encontrar a posição correta.

7.4 Função `remover`

A operação de remoção em uma árvore binária de busca é mais complexa do que busca ou inserção, pois precisamos manter a propriedade de ordenação da árvore mesmo após retirar um nó.

Dado um nó a ser removido, três situações podem ocorrer:

1. O nó não tem filhos: basta removê-lo.

2. O nó tem apenas um filho: removemos o nó e conectamos seu filho diretamente ao pai.

3. O nó tem dois filhos: substituímos o conteúdo do nó pela menor chave da subárvore direita (ou, alternativamente, pela maior da subárvore esquerda), e então removemos essa chave auxiliar da subárvore correspondente.

A seguir mostramos uma implementação recursiva usando o menor elemento da subárvore direita como substituto:

```
No* remover(No* raiz, int chave) {
    if (raiz == NULL)
        return NULL;

    if (chave < raiz->chave)
        raiz->esq = remover(raiz->esq, chave);
    else if (chave > raiz->chave)
        raiz->dir = remover(raiz->dir, chave);
    else {
        // caso 1 e 2: zero ou um filho
        if (raiz->esq == NULL) {
            No* temp = raiz->dir;
            free(raiz);
            return temp;
        }
        if (raiz->dir == NULL) {
            No* temp = raiz->esq;
            free(raiz);
            return temp;
        }

        // caso 3: dois filhos
        No* sucessor = raiz->dir;
        while (sucessor->esq != NULL)
            sucessor = sucessor->esq;

        raiz->chave = sucessor->chave;
        raiz->valor = sucessor->valor;
        raiz->dir = remover(raiz->dir, sucessor->chave);
    }

    return raiz;
}
```

Programa 7.4: Remoção recursiva em ABB

Vale destacar que, embora a remoção envolva lógica mais elaborada, seu custo assintótico é o mesmo das demais operações: ele depende unicamente da altura da árvore.

7.5 Menor e maior chave

Em um dicionário ordenado implementado com uma árvore binária de busca, é comum precisar encontrar a menor ou a maior chave presente na estrutura. Essas operações são diretas graças à propriedade fundamental da ABB.

Menor chave: a menor chave está no nó mais à esquerda da árvore. Basta seguir os ponteiros para o filho esquerdo até encontrar um nó que não tenha filho à esquerda.

Maior chave: a maior chave está no nó mais à direita da árvore. O procedimento é simétrico ao anterior: seguimos os ponteiros para o filho direito até encontrar um nó sem filho à direita.

```
No* minimo(No* raiz) {
    if (raiz == NULL)
        return NULL;
    while (raiz->esq != NULL)
        raiz = raiz->esq;
    return raiz;
}
```

Programa 7.5: Busca da menor chave em ABB

```
No* maximo(No* raiz) {
    if (raiz == NULL)
        return NULL;
    while (raiz->dir != NULL)
        raiz = raiz->dir;
    return raiz;
}
```

Programa 7.6: Busca da maior chave em ABB

Essas operações percorrem apenas um caminho da raiz até uma folha, o que leva tempo $\Theta(h)$, onde h é a altura da árvore.

7.6 Anterior e posterior

Em um dicionário ordenado, é útil localizar a chave imediatamente anterior ou posterior a uma dada chave. Essas operações são chamadas de **prede-**

cessor (anterior) e **sucessor** (posterior).

Dado um nó com chave k, temos duas situações para cada uma dessas operações:

Anterior (predecessor):

- Se o nó tem subárvore esquerda, o predecessor é o maior nó dessa subárvore (ou seja, o mais à direita da subárvore esquerda).
- Se o nó não tem subárvore esquerda, o predecessor está entre os ancestrais: é o último ancestral pelo qual passamos **descendo para a direita**.

Posterior (sucessor):

- Se o nó tem subárvore direita, o sucessor é o menor nó dessa subárvore (isto é, o mais à esquerda da subárvore direita).
- Se o nó não tem subárvore direita, o sucessor está entre os ancestrais: é o último ancestral pelo qual passamos **descendo para a esquerda**.

Observação: essas operações dependem do acesso direto ao nó cuja chave k queremos usar como referência. Se partimos apenas da raiz e conhecemos a chave k, precisamos primeiro localizar o nó correspondente com uma busca.

A seguir, apresentamos uma versão recursiva que, a partir da raiz da árvore e de uma chave k, encontra o sucessor (posterior):

```
1  No* sucessor(No* raiz, int chave) {
2      No* atual = raiz;
3      No* candidato = NULL;
4
5      while (atual != NULL) {
6          if (chave < atual->chave) {
7              candidato = atual;
8              atual = atual->esq;
9          } else {
10             atual = atual->dir;
11         }
12     }
13
14     return candidato;
15 }
```

Programa 7.7: Busca do sucessor em ABB

O algoritmo para o `predecessor` é análogo, trocando a lógica de comparação e os lados esquerdo/direito.

Como todas as operações anteriores, essas operações percorrem um único caminho da raiz até uma folha, tomando tempo $\Theta(h)$, onde h é a altura da árvore.

7.7 Visita em ordem

Uma das principais vantagens de usar uma árvore binária de busca é que ela permite percorrer todas as chaves em ordem crescente de maneira simples e eficiente.

O **percurso em ordem** (ou *in-order traversal*) segue o seguinte padrão recursivo:

1. Visita a subárvore esquerda;
2. Visita o nó atual;
3. Visita a subárvore direita.

Esse padrão garante que as chaves sejam visitadas em ordem crescente, respeitando a propriedade estrutural da árvore binária de busca.

A seguir, apresentamos uma função que percorre a árvore e aplica uma função passada como parâmetro a cada chave e valor:

```
1 void visitar_em_ordem(No* raiz, void (*visitar)(int, void
    *)) {
2     if (raiz == NULL)
3         return;
4
5     visitar_em_ordem(raiz->esq, visitar);
6     visitar(raiz->chave, raiz->valor);
7     visitar_em_ordem(raiz->dir, visitar);
8 }
```

Programa 7.8: Percurso em ordem em ABB

Essa função é útil para exibir os elementos do dicionário ordenado, salvar os pares chave-valor em um vetor, ou realizar qualquer outra operação sequencial.

A visita percorre todos os nós da árvore exatamente uma vez. Assim, sua complexidade é sempre $\Theta(n)$, independentemente da forma da árvore.

Capítulo 8

Árvores AVL

Árvores binárias de busca (ABBs) são estruturas eficientes para armazenar dados ordenados, permitindo operações como busca, inserção e remoção em tempo proporcional à altura da árvore. Quando essa altura é pequena – idealmente $\Theta(\lg(n))$ – essas operações são rápidas. No entanto, a eficiência das ABBs depende diretamente de sua forma estrutural.

Em muitas situações práticas, a árvore pode crescer de forma desbalanceada. Isso ocorre, por exemplo, quando os dados são inseridos em ordem crescente ou decrescente – casos extremos em que a ABB assume a forma de uma lista encadeada. Mas o problema não se limita a esses cenários: mesmo inserções em ordens menos regulares podem causar ramificações assimétricas, resultando em árvores com altura muito maior do que o necessário.

À medida que a altura cresce, o custo das operações também cresce. Uma ABB desbalanceada pode atingir altura linear no pior caso, tornando todas as operações proporcionalmente mais lentas – de tempo logarítmico para tempo linear.

Para evitar essa degradação, surgem as chamadas *árvores balanceadas*, que reorganizam sua estrutura a cada inserção ou remoção para manter a altura sob controle. O balanceamento visa preservar o desempenho logarítmico, limitando o crescimento desproporcional dos ramos. A árvore AVL, uma das primeiras estruturas desse tipo, impõe regras rígidas sobre o equilíbrio entre os ramos esquerdo e direito de cada nó, garantindo uma altura sempre próxima ao ideal.

Uma *árvore AVL* é uma árvore binária de busca que mantém uma propriedade de balanceamento estrito: para todo nó da árvore, a diferença entre as alturas das subárvores esquerda e direita deve ser, no máximo, 1. Essa diferença é chamada de **fator de balanceamento** do nó.

Formalmente, seja h_e a altura da subárvore esquerda de um nó e h_d a altura da subárvore direita. A árvore é dita balanceada nesse nó se:

$$|h_e - h_d| \leq 1$$

Se essa condição for violada após uma operação de inserção ou remoção, a árvore executa uma ou mais rotações locais para restaurar o equilíbrio.

Esse critério local garante que a altura total da árvore permaneça $\Theta(\lg n)$, o que assegura que as operações de busca, inserção e remoção mantenham complexidade eficiente, mesmo no pior caso.

A estrutura foi proposta em 1962 pelos matemáticos soviéticos **Georgy Adelson-Velsky** e **Evgenii Landis**, cujos sobrenomes deram origem à sigla **AVL**. Foi a primeira árvore binária de busca com balanceamento automático proposta na literatura, e estabeleceu as bases para várias outras estruturas balanceadas utilizadas em algoritmos e sistemas de dados.

O **fator de balanceamento** de um nó em uma árvore AVL é definido como a diferença entre as alturas das subárvores esquerda e direita. Seja h_e a altura da subárvore esquerda e h_d a altura da subárvore direita de um nó. O fator de balanceamento f é dado por:

$$f = h_e - h_d$$

Um nó está balanceado se $|f| \leq 1$. Se essa condição for violada após uma operação de inserção ou remoção – isto é, se $f < -1$ ou $f > 1$ – o nó está desbalanceado e a árvore precisa ser reestruturada.

O valor de f é utilizado para determinar qual tipo de rotação é necessária para restaurar o equilíbrio da árvore. Os casos mais comuns são:

- $f = 2$ e o fator do filho esquerdo > 0: rotação simples à direita.
- $f = 2$ e o fator do filho esquerdo < 0: rotação dupla esquerda-direita.
- $f = -2$ e o fator do filho direito < 0: rotação simples à esquerda.
- $f = -2$ e o fator do filho direito > 0: rotação dupla direita-esquerda.

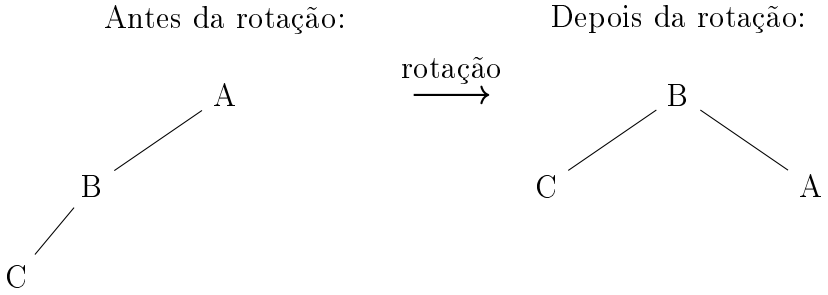
Essas rotações locais garantem que a altura da árvore permaneça próxima de $\log n$, mesmo após modificações que causariam desequilíbrio.

8.1 Rotações

Quando uma inserção ou remoção causa desequilíbrio em um nó — isto é, quando $|h_e - h_d| > 1$ — a árvore AVL realiza rotações locais para restaurar o equilíbrio. As rotações ajustam a estrutura da árvore sem violar a propriedade de ordem da árvore binária de busca.

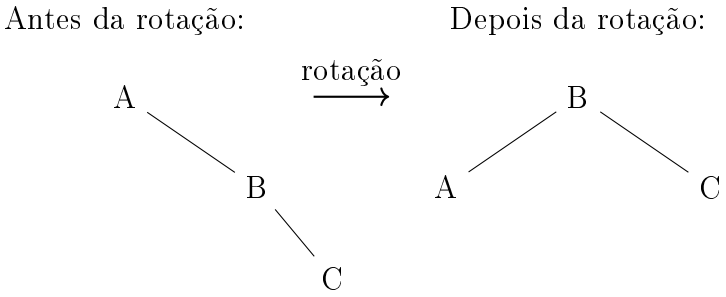
Rotação simples à direita

Esse tipo de rotação é usado quando o desequilíbrio ocorre na subárvore esquerda de um nó, e essa subárvore também tem seu maior peso à esquerda. Isso ocorre, por exemplo, após a inserção em uma subárvore esquerda-esquerda.



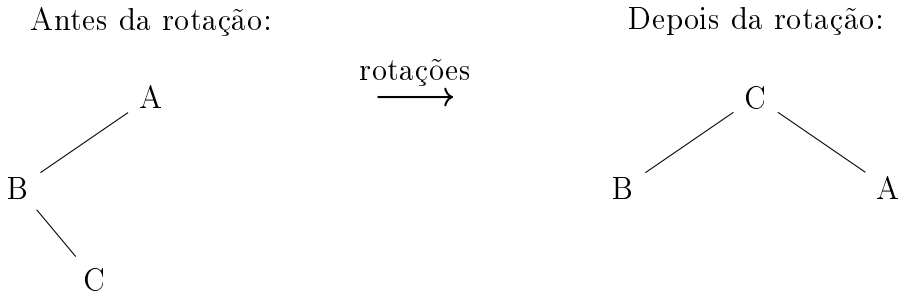
Rotação simples à esquerda

A rotação simples à esquerda é simétrica à rotação simples à direita, e ocorre quando o desequilíbrio está à direita de um nó, com o ramo mais pesado também à direita (caso direita-direita).



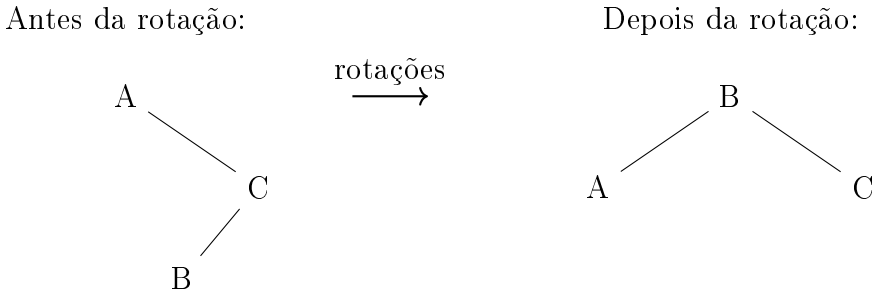
Rotação dupla esquerda-direita

Esse tipo de rotação é necessário quando o desequilíbrio ocorre na subárvore esquerda de um nó, mas o maior peso está à direita dessa subárvore. É o caso típico após inserção em uma subárvore esquerda-direita. A correção envolve duas rotações: uma rotação simples à esquerda no filho esquerdo, seguida de uma rotação simples à direita no nó desbalanceado.



Rotação dupla direita-esquerda

Esse tipo de rotação ocorre quando o desequilíbrio está na subárvore direita de um nó, mas o maior peso está à esquerda dessa subárvore. É o caso típico após inserção em uma subárvore direita-esquerda. A correção envolve duas rotações: uma rotação simples à direita no filho direito, seguida de uma rotação simples à esquerda no nó desbalanceado.



Função `inserir`

Antes de apresentar a inserção em uma árvore AVL, definimos três funções auxiliares essenciais para manter a propriedade de balanceamento da estrutura.

A função `altura` retorna a altura de um nó, considerando que um nó nulo tem altura -1:

```
1 int altura(Node* n) {
2     if (n == NULL)
3         return -1;
4     return n->altura;
5 }
```

A função `fatorBalanceamento` calcula a diferença entre as alturas das subárvores esquerda e direita de um nó:

```
int fatorBalanceamento(Node* n) {
    if (n == NULL)
        return 0;
    return altura(n->esq) - altura(n->dir);
}
```

A função `max` retorna o maior entre dois inteiros:

```
int max(int a, int b) {
    return (a > b) ? a : b;
}
```

A função `atualizarAltura` deve ser chamada sempre que a estrutura da árvore for modificada. Ela atualiza a altura de um nó com base nas alturas de seus filhos:

```
void atualizarAltura(Node* n) {
    n->altura = 1 + max(altura(n->esq), altura(n->dir));
}
```

As rotações são operações locais que reestruturam a árvore para restaurar o balanceamento, mantendo a propriedade de ordem da árvore binária de busca.

```
Node* rotacaoDireita(Node* y) {
    Node* x = y->esq;
    Node* T2 = x->dir;

    // Rotação
    x->dir = y;
    y->esq = T2;

    // Atualiza alturas
    atualizarAltura(y);
    atualizarAltura(x);

    return x;
}
```

```
Node* rotacaoEsquerda(Node* x) {
    Node* y = x->dir;
    Node* T2 = y->esq;

    // Rotação
    y->esq = x;
```

```
    x->dir = T2;

    // Atualiza alturas
    atualizarAltura(x);
    atualizarAltura(y);

    return y;
}
```

A operação de inserção em uma árvore AVL segue o mesmo princípio da inserção em uma árvore binária de busca, mas com um passo adicional: após a inserção, é necessário verificar se houve desequilíbrio e aplicar rotações se necessário. A função abaixo insere uma nova chave e reequilibra a árvore:

```
Node* inserir(Node* raiz, int chave) {
    if (raiz == NULL)
        return novoNo(chave);

    if (chave < raiz->chave)
        raiz->esq = inserir(raiz->esq, chave);
    else if (chave > raiz->chave)
        raiz->dir = inserir(raiz->dir, chave);
    else
        return raiz; // chaves duplicadas não são
           permitidas

    atualizarAltura(raiz);

    int fb = fatorBalanceamento(raiz);

    // Caso esquerda-esquerda
    if (fb > 1 && chave < raiz->esq->chave)
        return rotacaoDireita(raiz);

    // Caso direita-direita
    if (fb < -1 && chave > raiz->dir->chave)
        return rotacaoEsquerda(raiz);

    // Caso esquerda-direita
    if (fb > 1 && chave > raiz->esq->chave) {
        raiz->esq = rotacaoEsquerda(raiz->esq);
        return rotacaoDireita(raiz);
    }

```

```
30      // Caso direita-esquerda
31      if (fb < -1 && chave < raiz->dir->chave) {
32          raiz->dir = rotacaoDireita(raiz->dir);
33          return rotacaoEsquerda(raiz);
34      }
35
36      return raiz;
37  }
```

Função `remover`

A operação de remoção em uma árvore AVL começa como em uma árvore binária de busca: localiza-se o nó com a chave a ser removida, e aplica-se uma das três regras usuais:

- Se o nó é uma folha, ele é simplesmente removido.
- Se tem apenas um filho, substitui-se o nó pelo filho.
- Se tem dois filhos, substitui-se o valor pela menor chave da subárvore direita (ou maior da esquerda), e remove-se esse sucessor recursivamente.

Após a remoção, a árvore pode ficar desbalanceada em vários nós no caminho de volta da recursão. Para cada nó, atualizamos a altura e aplicamos rotações se necessário.

```
1   Node* remover(Node* raiz, int chave) {
2       if (raiz == NULL)
3           return NULL;
4
5       if (chave < raiz->chave)
6           raiz->esq = remover(raiz->esq, chave);
7       else if (chave > raiz->chave)
8           raiz->dir = remover(raiz->dir, chave);
9       else {
10          // Caso com 0 ou 1 filho
11          if (raiz->esq == NULL || raiz->dir == NULL) {
12              Node* temp = raiz->esq ? raiz->esq : raiz->
                    dir;
13              free(raiz);
14              return temp;
```

```
15          }
16
17          // Caso com 2 filhos: substitui pelo menor da
                direita
18          Node* temp = minimo(raiz->dir);
19          raiz->chave = temp->chave;
20          raiz->dir = remover(raiz->dir, temp->chave);
21      }
22
23      atualizarAltura(raiz);
24      int fb = fatorBalanceamento(raiz);
25
26      // Rebalanceamento
27
28      // Caso esquerda-esquerda
29      if (fb > 1 && fatorBalanceamento(raiz->esq) >= 0)
30          return rotacaoDireita(raiz);
31
32      // Caso esquerda-direita
33      if (fb > 1 && fatorBalanceamento(raiz->esq) < 0) {
34          raiz->esq = rotacaoEsquerda(raiz->esq);
35          return rotacaoDireita(raiz);
36      }
37
38      // Caso direita-direita
39      if (fb < -1 && fatorBalanceamento(raiz->dir) <= 0)
40          return rotacaoEsquerda(raiz);
41
42      // Caso direita-esquerda
43      if (fb < -1 && fatorBalanceamento(raiz->dir) > 0) {
44          raiz->dir = rotacaoDireita(raiz->dir);
45          return rotacaoEsquerda(raiz);
46      }
47
48      return raiz;
49  }
```

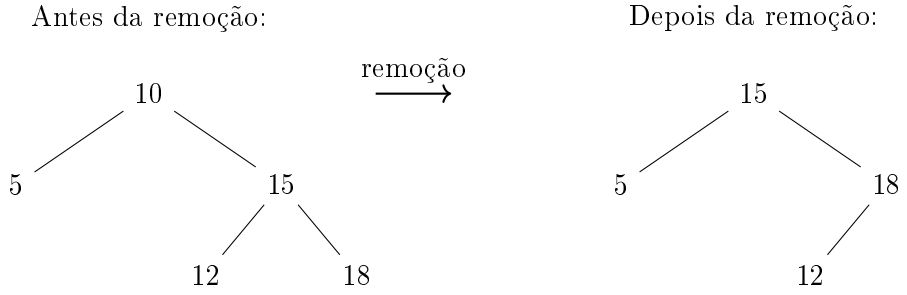
A função `minimo` retorna o ponteiro para o nó com a menor chave em uma árvore, ou seja, o mais à esquerda.

```
1   Node* minimo(Node* n) {
2       while (n->esq != NULL)
3           n = n->esq;
4       return n;
```

```
}
```

Mesmo com o custo adicional de rotações, a remoção em uma árvore AVL continua com complexidade $O(\log n)$ no pior caso.

Como exemplo ilustrativo, considere a remoção do valor `10` em uma árvore AVL. O nó a ser removido possui dois filhos, o que exige que seja substituído por seu sucessor – isto é, o menor elemento da subárvore direita, neste caso o nó `15`. Após a substituição, o nó `15` original é removido, e a árvore pode ficar desbalanceada, exigindo atualizações de altura e, se necessário, rotações. O diagrama a seguir mostra a árvore antes e depois da operação de remoção:



8.2 Análise de Complexidade

As árvores AVL mantêm a altura controlada por meio de um critério estrito de balanceamento, garantindo complexidade eficiente para todas as operações fundamentais.

Uma árvore AVL com n nós tem altura $\Theta(\log n)$. Isso se deve ao fato de que, a cada nó, a diferença de altura entre as subárvores esquerda e direita é no máximo 1. Pode-se demonstrar, por indução, que a altura $h(n)$ de uma árvore AVL com n nós satisfaz a desigualdade:

$$h(n) \leq c \cdot \log_2(n+1)$$

para alguma constante c. Assim, as operações de busca, inserção e remoção têm complexidade assintótica proporcional à altura da árvore, isto é, $O(\log n)$.

As rotações são operações locais, que envolvem um número constante de atualizações de ponteiros e alturas. Cada rotação simples (à esquerda ou à direita) custa $O(1)$, e o mesmo vale para rotações duplas (compostas por duas rotações simples consecutivas). Assim, mesmo quando ocorrem, as rotações não aumentam a complexidade assintótica das operações principais.

A operação de inserção em uma árvore AVL realiza, no pior caso, um número proporcional a $\log n$ de chamadas recursivas, correspondente à altura da árvore. No entanto, o rebalanceamento necessário após a inserção afeta apenas o primeiro nó do caminho de volta cuja propriedade de balanceamento foi violada. Após aplicar a rotação apropriada nesse ponto – seja simples ou dupla – o equilíbrio é restaurado localmente, e os demais nós ancestrais não precisam de novas rotações, apenas de atualizações de altura.

Essa característica é uma consequência direta da estrutura da árvore AVL: como a inserção só aumenta a altura de uma subárvore em no máximo 1, e a rotação corrige esse crescimento excessivo, não é necessário propagar rotações por todo o caminho até a raiz. Portanto, embora o caminho de subida possa envolver até $O(\log n)$ atualizações de altura, apenas uma rotação (ou, no máximo, uma reestruturação composta por duas rotações simples) é realizada.

Assim, o custo total da inserção permanece $O(\log n)$ no pior caso, e o custo amortizado das rotações é constante.

A operação de remoção, no entanto, pode provocar desequilíbrios em múltiplos nós até a raiz. Ainda assim, o número total de rotações é limitado por $O(\log n)$, o que mantém o custo amortizado da remoção dentro da mesma ordem de complexidade.

Capítulo 9

Árvores B

Árvores B são estruturas de dados projetadas especificamente para contextos em que o custo do acesso à **memória externa** – como discos rígidos ou SSDs – é o principal gargalo de desempenho. Isso ocorre, por exemplo, em sistemas de arquivos, bancos de dados e outras aplicações que manipulam grandes volumes de dados que não cabem inteiramente na memória principal. Nesses casos, é essencial reduzir ao máximo a quantidade de acessos a disco, que são várias ordens de magnitude mais lentos do que os acessos à RAM.

Estruturas tradicionais como árvores binárias de busca balanceadas oferecem bom desempenho em memória, com altura proporcional a $\log n$, mas ainda exigem múltiplos acessos a diferentes blocos de disco: um para cada nível da árvore. Árvores B foram concebidas para esse cenário. Ao permitir que cada nó armazene várias chaves e tenha muitos filhos, elas reduzem a altura da árvore de $\log_2 n$ para $\log_t n$, com $t \gg 2$, o que diminui drasticamente o número de acessos à memória secundária necessários para localizar uma chave.

As **árvores B** são estruturas balanceadas que mantêm os dados ordenados e permitem operações eficientes de busca, inserção e remoção, mesmo em situações em que os dados estão distribuídos por centenas ou milhares de blocos de disco. Uma das características mais importantes dessas árvores é que cada nó pode conter várias chaves e ter múltiplos filhos, o que permite manter a altura da árvore pequena mesmo com um número muito grande de elementos.

Enquanto uma árvore binária de busca (ABB) tem no máximo dois filhos por nó e mantém uma única chave em cada nó, uma árvore B pode ter dezenas ou centenas de filhos por nó e armazenar múltiplas chaves. Isso permite que:

- A altura da árvore seja muito menor
- A quantidade de acessos à memória externa seja reduzida

- A inserção e remoção sejam feitas de forma eficiente mesmo em grandes volumes de dados

Além disso, ao contrário de árvores como as AVL, que focam em manter o tempo de operação em $O(\log n)$ com reestruturações locais, as árvores B são projetadas para alinhar sua estrutura à lógica de acesso por blocos, típica de discos e SSDs.

A definição de uma árvore B depende de um parâmetro chamado *grau mínimo*, denotado por t, com $t \geq 2$. Esse parâmetro determina os limites inferior e superior para o número de chaves que um nó pode armazenar. Cada nó da árvore (exceto a raiz) deve conter no mínimo $t-1$ e no máximo $2t-1$ chaves. Consequentemente, o número de filhos de um nó com k chaves é $k+1$, o que significa que cada nó interno pode ter entre t e $2t$ filhos. A raiz é uma exceção: ela pode ter menos de $t-1$ chaves (inclusive zero, no caso de uma árvore vazia).

As chaves em cada nó estão armazenadas em ordem crescente. Cada filho intercalado entre as chaves define um intervalo: o primeiro filho contém apenas chaves menores que a primeira chave do nó, o segundo filho contém chaves entre a primeira e a segunda chave, e assim por diante, até o último filho, que contém apenas chaves maiores que a última chave do nó.

Há dois tipos de nós em uma árvore B: os nós internos e as folhas. Os *nós internos* possuem filhos e organizam a estrutura da árvore; *as folhas*, por sua vez, não possuem filhos e armazenam apenas as chaves. Uma propriedade essencial das árvores B é que todas as folhas estão no mesmo nível, o que garante que a árvore seja perfeitamente balanceada – todos os caminhos da raiz até as folhas têm o mesmo comprimento.

Essas regras garantem que a altura da árvore B cresça de forma lenta mesmo com um número muito grande de elementos. Como consequência, operações de busca, inserção e remoção mantêm complexidade logarítmica, mas com uma base maior no logaritmo, o que resulta em menos acessos a disco e melhor desempenho em sistemas com memória externa.

9.1 Estrutura

Cada nó em uma árvore B armazena um conjunto de chaves ordenadas e ponteiros para seus filhos. A estrutura de um nó com k chaves é composta por:

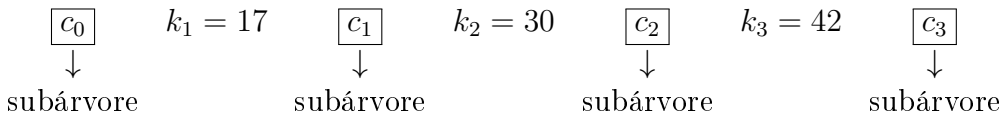
- Um vetor de chaves ordenadas: $k_1 < k_2 < \cdots < k_k$
- Um vetor de ponteiros para filhos: $c_0, c_1, \ldots, c_k$

Cada ponteiro c_i aponta para uma subárvore que contém apenas chaves dentro de um intervalo bem definido:

Todas as chaves em $c_0 < k_1$, todas as chaves em $c_1 \in (k_1, k_2)$, ..., todas em $c_k > k_k$

O número de filhos de um nó é sempre uma unidade a mais do que o número de chaves. Assim, um nó pode ter entre $t-1$ e $2t-1$ chaves e entre t e $2t$ filhos (exceto a raiz).

Exemplo com $t = 3$:
Neste caso, cada nó pode ter entre 2 e 5 chaves e entre 3 e 6 filhos. Abaixo está uma representação textual e visual de um nó que contém 3 chaves e, portanto, 4 filhos: Visualmente, esse nó pode ser representado assim:



Nesse exemplo, o nó possui três chaves ordenadas $k_1 = 17$, $k_2 = 30$, $k_3 = 42$, e quatro ponteiros c_0, c_1, c_2, c_3, c_4, intercalados entre as chaves. Cada ponteiro aponta para uma subárvore que cobre o intervalo correspondente:

- c_0: chaves menores que 17
- c_1: chaves entre 17 e 30
- c_2: chaves entre 30 e 42
- c_3: chaves maiores que 42

Essa estrutura é especialmente eficiente para leitura em disco: o nó inteiro pode ser armazenado em um único bloco, e ao acessá-lo, o sistema já carrega diversas chaves e todos os ponteiros relevantes para decidir o próximo passo da busca, da inserção ou da remoção.

A seguir está o código em C que define a estrutura de uma árvore B com grau mínimo genérico t, representado simbolicamente pela constante `T`. Cada nó pode armazenar até $2t-1$ chaves e até $2t$ ponteiros para filhos. No trecho abaixo, o valor de `T` está fixado em 3, mas ele pode ser ajustado conforme a aplicação.

```
1 // Grau mínimo da árvore B
2 #define T 3
3
```

```c
// Número máximo de chaves e filhos por nó
#define MAX_KEYS (2 * T - 1)
#define MAX_CHILDREN (2 * T)

// Estrutura de um nó da árvore B
typedef struct BTreeNode {
    int n;                              // Número de chaves atualmente no nó
    int is_leaf;                        // 1 se for folha, 0 caso contrário
    int keys[MAX_KEYS];                 // Vetor de chaves ordenadas
    struct BTreeNode* children[MAX_CHILDREN];  // Ponteiros para os filhos
} BTreeNode;

// Estrutura da árvore B (mantém referência à raiz)
typedef struct BTree {
    BTreeNode* root;
} BTree;
```

A seguir, definimos funções para criar um novo nó e uma nova árvore B vazia com a raiz sendo uma folha:

```c
#include <stdlib.h>

// Cria um novo nó da árvore B
BTreeNode* create_node(int is_leaf) {
    BTreeNode* node = (BTreeNode*) malloc(sizeof(BTreeNode));
    node->n = 0;
    node->is_leaf = is_leaf;
    for (int i = 0; i < MAX_CHILDREN; i++) {
        node->children[i] = NULL;
    }
    return node;
}

// Cria uma nova árvore B com raiz vazia
BTree* create_btree() {
    BTree* tree = (BTree*) malloc(sizeof(BTree));
    tree->root = create_node(1);  // começa com a raiz sendo uma folha
    return tree;
```

```
}
```

9.2 Função `buscar`

A operação de busca em uma árvore B segue o mesmo princípio das árvores de busca binária, porém adaptada para nós que contêm múltiplas chaves. A cada nível da árvore, a busca ocorre em dois passos:

1. Percorremos sequencialmente o vetor de chaves do nó atual até encontrar a chave desejada ou uma chave maior.
2. Se encontrarmos a chave, a busca termina com sucesso. Caso contrário, seguimos para o filho correspondente, cujo intervalo contém a chave procurada.

Como cada nó possui múltiplas chaves, conseguimos descartar grandes intervalos do espaço de busca em cada passo, mantendo a profundidade da árvore pequena (proporcional a $\log_t n$). Isso torna a busca eficiente, com poucos acessos a disco em aplicações de memória externa.

A busca é naturalmente recursiva, mas pode ser implementada iterativamente.

```
// Busca uma chave em um nó da árvore B (recursivamente)
BTreeNode* btree_search_node(BTreeNode* node, int key,
    int* index_out) {
    int i = 0;
    while (i < node->n && key > node->keys[i]) {
        i++;
    }

    if (i < node->n && key == node->keys[i]) {
        if (index_out) *index_out = i;
        return node;
    }

    if (node->is_leaf) return NULL;

    return btree_search_node(node->children[i], key,
        index_out);
}

// Busca uma chave na árvore B
```

```
19 BTreeNode* btree_search(BTree* tree, int key, int*
       index_out) {
20     if (!tree || !tree->root) return NULL;
21     return btree_search_node(tree->root, key, index_out);
22 }
```

A complexidade é $O(\log n)$ no pior caso, tanto em número de comparações quanto em profundidade da árvore.

9.3 Função `inserir`

O processo de inserção em árvores B mantém a árvore balanceada e todas as suas propriedades. O objetivo é inserir uma nova chave mantendo as chaves em ordem e garantindo que nenhum nó ultrapasse o número máximo de $2t-1$ chaves.

A inserção pode ocorrer em duas situações:

- **Inserção em uma folha que ainda possui espaço:** a chave é inserida diretamente no vetor de chaves do nó, mantendo a ordenação.
- **Inserção em uma folha cheia:** o nó precisa ser dividido. A chave mediana sobe para o pai, e o nó é dividido em dois nós com $t-1$ chaves cada.

Considere a inserção da chave 35 em um nó que já contém as chaves 17, 24, 30, 42 e 50. Com $t = 3$, o nó já está com o número máximo de chaves ($2t - 1 = 5$) e precisa ser dividido.

O processo é o seguinte:

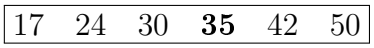
1. As chaves do nó são reorganizadas com a nova chave: 17, 24, 30, **35**, 42, 50.
2. A chave mediana (35) é promovida ao nó pai.
3. As chaves menores que 35 permanecem no nó original.
4. As chaves maiores que 35 vão para um novo nó à direita.

O diagrama abaixo mostra a estrutura antes e depois da divisão:

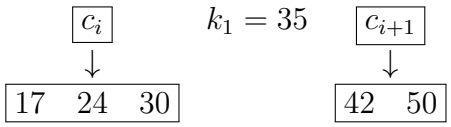
Antes da inserção:

17	24	30	42	50

Após tentativa de inserir 35 (antes da divisão):



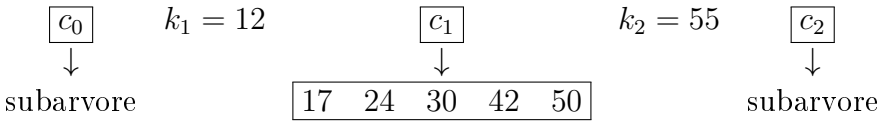
Após a divisão:



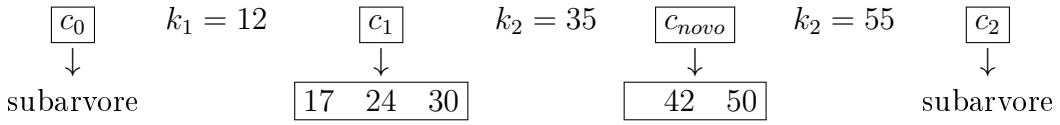
Se o nó pai também estiver cheio, o processo se repete recursivamente até (se necessário) a raiz ser dividida e a altura da árvore aumentar em 1.

Quando um nó folha está cheio e uma nova chave é inserida, ele é dividido e a chave mediana é promovida ao nó pai. Essa chave deve ser inserida em ordem no vetor de chaves do pai, e o novo ponteiro (referente ao nó criado à direita da divisão) deve ser inserido imediatamente à direita da nova chave.

Por exemplo, suponha que o nó pai inicialmente contenha:



Após a promoção da chave 35, ela deve ser inserida entre 12 e 55, e um novo ponteiro c_{novo} deve ser inserido à sua direita. O resultado será:



A estrutura do nó pai continua com chaves ordenadas e ponteiros intercalados, mantendo as propriedades da árvore B.

A inserção em uma árvore B é feita de forma recursiva, sempre garantindo que a chave seja inserida em um nó que ainda não está cheio. Quando um nó está cheio, ele é dividido antes da inserção. A seguir, explicamos as principais funções envolvidas no processo.

Essa é a função principal de inserção. Ela verifica se a raiz da árvore está cheia. Se estiver, cria uma nova raiz, divide a antiga e então chama a função auxiliar `btree_insert_nonfull`.

```
1 void btree_insert(BTree* tree, int key) {
2     BTreeNode* root = tree->root;
3
```

```
    if (root->n == MAX_KEYS) {
        BTreeNode* new_root = create_node(0);
        new_root->children[0] = root;
        btree_split_child(new_root, 0);
        tree->root = new_root;
        btree_insert_nonfull(new_root, key);
    } else {
        btree_insert_nonfull(root, key);
    }
}
```

Vamos primeiro considerar o caso em que o nó que *não está cheio*. Se o nó for uma folha, a chave é inserida diretamente no vetor de chaves. Se o nó for interno, a função localiza o filho adequado e, caso ele esteja cheio, o divide antes de continuar.

```
void btree_insert_nonfull(BTreeNode* node, int key) {
    int i = node->n - 1;

    if (node->is_leaf) {
        while (i >= 0 && key < node->keys[i]) {
            node->keys[i + 1] = node->keys[i];
            i--;
        }
        node->keys[i + 1] = key;
        node->n++;
    } else {
        while (i >= 0 && key < node->keys[i]) {
            i--;
        }
        i++;
        if (node->children[i]->n == MAX_KEYS) {
            btree_split_child(node, i);
            if (key > node->keys[i]) {
                i++;
            }
        }
        btree_insert_nonfull(node->children[i], key);
    }
}
```

Caso o nó esteja cheio essa função divide o filho `children[i]`. A chave mediana é promovida para `node`, e um novo nó é criado com as chaves maiores. As subárvores também são realocadas, se necessário.

```c
void btree_split_child(BTreeNode* node, int i) {
    BTreeNode* y = node->children[i];
    BTreeNode* z = create_node(y->is_leaf);
    z->n = T - 1;

    for (int j = 0; j < T - 1; j++) {
        z->keys[j] = y->keys[j + T];
    }

    if (!y->is_leaf) {
        for (int j = 0; j < T; j++) {
            z->children[j] = y->children[j + T];
        }
    }

    y->n = T - 1;

    for (int j = node->n; j >= i + 1; j--) {
        node->children[j + 1] = node->children[j];
    }
    node->children[i + 1] = z;

    for (int j = node->n - 1; j >= i; j--) {
        node->keys[j + 1] = node->keys[j];
    }
    node->keys[i] = y->keys[T - 1];
    node->n++;
}
```

9.4 Função `remover`

A operação de remoção em árvores B é mais complexa do que a inserção, pois deve garantir que todos os nós (exceto a raiz) mantenham pelo menos $t - 1$ chaves após a remoção. Isso exige, em alguns casos, realizar fusões de nós ou redistribuições de chaves entre irmãos.

O processo de remoção envolve múltiplos casos, dependendo de onde está a chave a ser removida e das condições dos nós envolvidos. A seguir, apresentamos os casos possíveis:

Caso 1: a chave está em um nó folha

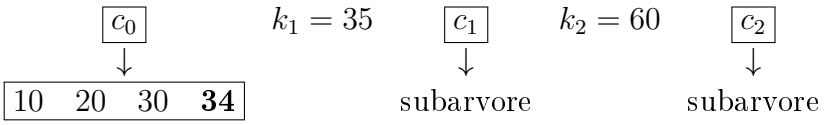
Este é o caso mais simples. Basta remover a chave diretamente do vetor de chaves do nó.

Caso 2: a chave está em um nó interno

Nesse caso, temos três subcasos possíveis:

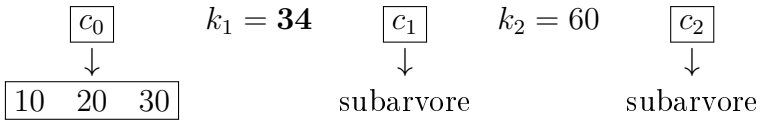
- **2a. O filho anterior (c_i) tem pelo menos t chaves:** substituímos a chave a ser removida pela maior chave da subárvore esquerda e removemos recursivamente essa chave.

  Considere o seguinte nó:



  Desejamos remover a chave 35, que está no nó interno. O filho à esquerda, c_0, contém pelo menos $t = 3$ chaves, então aplicamos a substituição pelo **predecessor** da chave – neste caso, a maior chave em c_0, que é **34**.

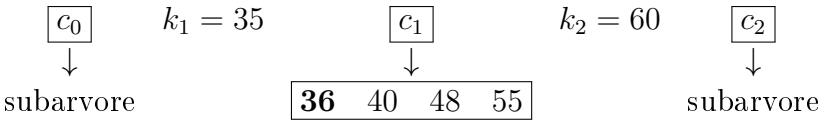
  A nova configuração do nó será:



  Agora, a chave 35 foi substituída por 34, e a chave 34 será removida recursivamente do nó c_0.

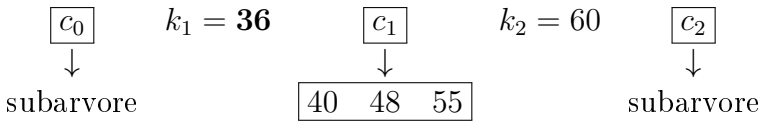
- **2b. O filho seguinte (c_{i+1}) tem pelo menos t chaves:** substituímos a chave pela menor chave da subárvore direita e removemos recursivamente essa chave.

  Considere o seguinte nó:



Desejamos remover a chave 35, que está no nó interno. O filho à direita, c_1, contém pelo menos $t = 3$ chaves. Nesse caso, substituímos a chave 35 por seu **sucessor**, a menor chave da subárvore direita — neste exemplo, **36**.

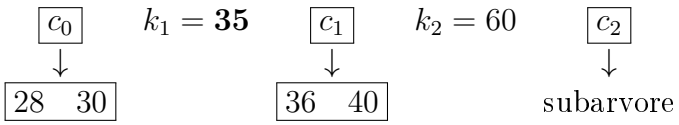
A nova configuração do nó será:



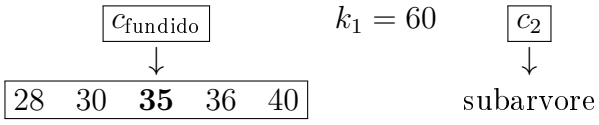
Agora, a chave 35 foi substituída por 36, e a chave 36 será removida recursivamente do nó c_1.

- **2c. Ambos os filhos têm exatamente $t-1$ chaves:** unimos os dois filhos e a chave entre eles em um único nó e continuamos a remoção recursivamente.

  Considere o seguinte nó:



  Queremos remover a chave 35, que está no nó interno. Ambos os filhos adjacentes, c_0 e c_1, têm exatamente $t-1 = 2$ chaves (assumindo $t = 3$). Nesse caso, fundimos os dois filhos e a chave 35 em um único nó com 5 chaves:



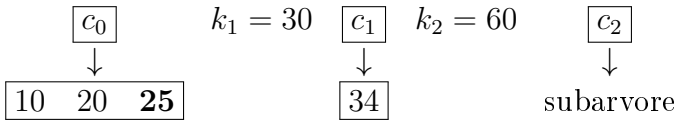
  Agora o nó pai tem uma chave a menos, e a árvore continua válida. A remoção da chave 35 continua recursivamente no novo nó fundido.

Caso 3: a chave não está no nó e a descida ocorre para um filho com $t-1$ chaves

Se ao descer a árvore encontramos um filho com o número mínimo de chaves, precisamos garantir que ele tenha pelo menos t chaves antes de prosseguir. Isso pode ser feito de duas formas:

- **3a. O irmão à esquerda ou à direita tem pelo menos t chaves:** realizamos uma rotação, movendo uma chave do pai para o filho e uma chave do irmão para o pai.

  Considere o seguinte nó pai:

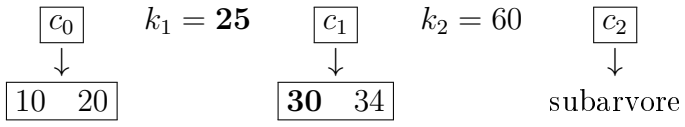


  O filho c_1 tem apenas uma chave $(t-1)$, e precisamos descer por ele para remover algum valor. Seu irmão à esquerda, c_0, tem 3 chaves (assumindo $t = 3$) e pode ceder uma chave.

  Executamos a rotação:

  - A chave 30 (do pai) desce para c_1;
  - A maior chave de c_0, que é 25, sobe para o pai;
  - Se houver ponteiros, o último de c_0 é transferido para c_1.

  Depois da rotação:

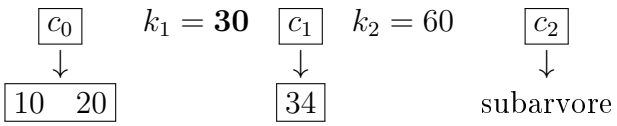


  Agora c_1 possui 2 chaves e a remoção pode prosseguir. As propriedades da árvore B estão preservadas: todos os nós (exceto possivelmente a raiz) mantêm pelo menos $t-1$ chaves.

- **3b. Ambos os irmãos têm $t-1$ chaves:** fundimos o filho com um dos irmãos, reduzindo o número de filhos do nó pai.

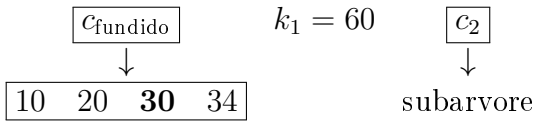
Caso 3b: Fusão quando todos os irmãos têm $t-1$ chaves

Considere o seguinte nó pai:



Suponha que desejamos descer por c_1, que tem apenas 1 chave ($t-1$), e seu irmão à esquerda c_0 também tem apenas $t-1=2$ chaves (assumindo $t=3$). Como nenhum dos irmãos pode ceder uma chave, realizamos uma *fusão* de c_0, a chave 30 e c_1 em um único nó.

Após a fusão:



Agora:

- A chave 30 foi removida do pai e fundida com os dois filhos adjacentes;
- O número de filhos do pai foi reduzido;
- A remoção prossegue recursivamente no nó fundido.

Se a fusão esvaziar a raiz, a altura da árvore diminui em 1, pois a nova raiz será o nó fundido.

A cada etapa da remoção, tomamos cuidado para que as propriedades fundamentais da árvore B sejam preservadas:

- **Todos os nós (exceto a raiz) possuem pelo menos $t-1$ chaves:** Antes de descer por um filho com exatamente $t-1$ chaves, garantimos que ele receba uma chave extra, seja por rotação com um irmão (caso 3a) ou por fusão com um irmão e uma chave do pai (caso 3b). Assim, evitamos que o nó atinja um número de chaves inferior ao permitido.
- **As chaves permanecem ordenadas:** Tanto as substituições por predecessores ou sucessores (casos 2a e 2b), quanto as rotações e fusões, preservam a ordem das chaves nos nós. A árvore mantém sua propriedade de busca, ou seja, todas as chaves em um filho à esquerda de uma chave k são menores que k, e todas as chaves no filho à direita são maiores.
- **A altura da árvore pode diminuir:** Em situações em que a raiz perde sua última chave – por exemplo, após uma fusão entre seus dois únicos filhos –, ela deixa de ser necessária e a árvore diminui de altura. Isso é permitido e faz parte do funcionamento da estrutura: a árvore B se adapta dinamicamente ao número de elementos, mantendo-se balanceada com altura proporcional a $\log_t n$.

A função `btree_delete` é o ponto de entrada para a remoção de uma chave na árvore B. Ela realiza as seguintes etapas:

- Verifica se a raiz da árvore está vazia. Caso esteja, não há o que remover.
- Chama a função auxiliar `delete_from_node`, que implementa a lógica recursiva da remoção.
- Após a remoção, verifica se a raiz ficou sem nenhuma chave. Se isso acontecer:
  - Se a raiz for uma folha, a árvore se torna vazia.
  - Se a raiz não for uma folha (ou seja, tinha filhos), a altura da árvore diminui: o primeiro filho da raiz se torna a nova raiz.

Esse procedimento é necessário porque, em árvores B, a raiz é o único nó autorizado a ter menos de $t-1$ chaves. Quando ela se esvazia completamente e ainda possui filhos, isso indica que o conteúdo real da árvore está em um nível inferior e a estrutura pode ser simplificada.

A remoção nunca quebra as propriedades da árvore B: após essa verificação e ajuste, a árvore continua válida, balanceada e com todas as restrições de número de chaves por nó satisfeitas.

```
1  // Função principal: remove uma chave da árvore B
2  void btree_delete(BTree* tree, int key) {
3      if (!tree->root) return;
4
5      delete_from_node(tree->root, key);
6
7      // Se a raiz ficou sem chaves e tem filhos, diminui a
           altura
8      if (tree->root->n == 0) {
9          BTreeNode* old_root = tree->root;
10         if (old_root->is_leaf) {
11             // árvore vazia
12             free(old_root);
13             tree->root = NULL;
14         } else {
15             tree->root = old_root->children[0];
16             free(old_root);
17         }
18     }
19 }
```

A função `delete_from_node` remove uma chave de um nó da árvore B, tratando todos os casos previstos:

- Se a chave está no nó e ele é uma folha, a chave é simplesmente removida.
- Se a chave está no nó e ele é interno:
  - Se o filho anterior tem pelo menos t chaves, a chave é substituída pelo seu predecessor.
  - Caso contrário, se o filho seguinte tem pelo menos t chaves, a chave é substituída pelo sucessor.
  - Se ambos os filhos têm $t - 1$ chaves, realiza-se uma fusão e a remoção continua no nó resultante.
- Se a chave não está no nó, é necessário descer para o filho apropriado. Antes disso, se esse filho tiver apenas $t - 1$ chaves, é preciso garantir que ele tenha pelo menos t – usando rotação (caso 3a) ou fusão (caso 3b).

```
1  // Remove uma chave de um nó da árvore B
2  void delete_from_node(BTreeNode* node, int key) {
3      int idx = 0;
4      while (idx < node->n && key > node->keys[idx]) {
5          idx++;
6      }
7
8      // Caso 1 e 2: chave encontrada no nó
9      if (idx < node->n && node->keys[idx] == key) {
10         if (node->is_leaf) {
11             // Caso 1: nó folha
12             for (int i = idx + 1; i < node->n; i++) {
13                 node->keys[i - 1] = node->keys[i];
14             }
15             node->n--;
16         } else {
17             // Caso 2: nó interno
18             BTreeNode* left = node->children[idx];
19             BTreeNode* right = node->children[idx + 1];
20
21             if (left->n >= T) {
22                 int pred = get_predecessor(left);
```

```
                node->keys[idx] = pred;
                delete_from_node(left, pred);
            } else if (right->n >= T) {
                int succ = get_successor(right);
                node->keys[idx] = succ;
                delete_from_node(right, succ);
            } else {
                merge_children(node, idx);
                delete_from_node(left, key); // após fusão, left contém tudo
            }
        }
    } else {
        // Caso 3: chave não está no nó
        if (node->is_leaf) return; // chave não encontrada

        BTreeNode* child = node->children[idx];

        // Se o filho tem apenas t-1 chaves, prepare-o
        if (child->n == T - 1) {
            fill(node, idx);
        }

        // Após ajuste, reobter o filho (pode ter sido fundido)
        if (idx > node->n) {
            delete_from_node(node->children[idx - 1], key);
        } else {
            delete_from_node(node->children[idx], key);
        }
    }
}
```

A seguinte função percorre o filho mais à direita recursivamente até encontrar uma folha e retorna sua última chave.

```
// Retorna a maior chave da subárvore esquerda
int get_predecessor(BTreeNode* node) {
    while (!node->is_leaf) {
        node = node->children[node->n];
    }
    return node->keys[node->n - 1];
```

```
}
```

Simétrica à anterior, a função a seguir percorre o filho mais à esquerda até uma folha e retorna sua primeira chave.

```
// Retorna a menor chave da subárvore direita
int get_successor(BTreeNode* node) {
    while (!node->is_leaf) {
        node = node->children[0];
    }
    return node->keys[0];
}
```

A próxima função funde dois filhos e a chave que os separa, reduzindo o número de filhos e reorganizando o nó pai.

```
// Funde os filhos children[idx] e children[idx+1] com a
   chave entre eles
void merge_children(BTreeNode* node, int idx) {
    BTreeNode* left = node->children[idx];
    BTreeNode* right = node->children[idx + 1];

    // Move a chave do pai para o meio de left
    left->keys[T - 1] = node->keys[idx];

    // Copia chaves de right para left
    for (int i = 0; i < right->n; i++) {
        left->keys[i + T] = right->keys[i];
    }

    // Copia filhos de right (se não for folha)
    if (!left->is_leaf) {
        for (int i = 0; i <= right->n; i++) {
            left->children[i + T] = right->children[i];
        }
    }

    left->n += right->n + 1;

    // Remove chave e ponteiro do pai
    for (int i = idx + 1; i < node->n; i++) {
        node->keys[i - 1] = node->keys[i];
        node->children[i] = node->children[i + 1];
    }
    node->n--;
```

```
29
30      free(right);
31  }
```

Chamada quando um filho com $t-1$ chaves precisa ser ajustado antes de continuar a descida chamamos a seguinte função que tenta rotação com um irmão; se não for possível, realiza fusão.

```
1   // Garante que children[idx] terá pelo menos T chaves
2   void fill(BTreeNode* node, int idx) {
3       if (idx > 0 && node->children[idx - 1]->n >= T) {
4           borrow_from_prev(node, idx);
5       } else if (idx < node->n && node->children[idx + 1]->
            n >= T) {
6           borrow_from_next(node, idx);
7       } else {
8           if (idx < node->n) {
9               merge_children(node, idx);
10          } else {
11              merge_children(node, idx - 1);
12          }
13      }
14  }
```

A função `borrow_from_prev` realiza uma rotação à direita, transferindo uma chave do irmão esquerdo para o filho e ajustando o pai para manter as propriedades da árvore B.

```
1   // Rotação: empresta do irmão anterior
2   void borrow_from_prev(BTreeNode* node, int idx) {
3       BTreeNode* child = node->children[idx];
4       BTreeNode* sibling = node->children[idx - 1];
5
6       // Desloca chaves e filhos de child para a direita
7       for (int i = child->n - 1; i >= 0; i--) {
8           child->keys[i + 1] = child->keys[i];
9       }
10      if (!child->is_leaf) {
11          for (int i = child->n; i >= 0; i--) {
12              child->children[i + 1] = child->children[i];
13          }
14      }
15
16      // Move chave do pai para child
17      child->keys[0] = node->keys[idx - 1];
```

```
18
19      // Move último filho do irmão para child
20      if (!child->is_leaf) {
21          child->children[0] = sibling->children[sibling->n
                ];
22      }
23
24      // Move chave do irmão para o pai
25      node->keys[idx - 1] = sibling->keys[sibling->n - 1];
26
27      child->n++;
28      sibling->n--;
29  }
```

A função `borrow_from_next` é análoga a anterior:

```
1   // Rotação: empresta do irmão seguinte
2   void borrow_from_next(BTreeNode* node, int idx) {
3       BTreeNode* child = node->children[idx];
4       BTreeNode* sibling = node->children[idx + 1];
5
6       // Move chave do pai para child
7       child->keys[child->n] = node->keys[idx];
8
9       // Move primeiro filho do irmão, se necessário
10      if (!child->is_leaf) {
11          child->children[child->n + 1] = sibling->children
                [0];
12      }
13
14      // Move chave do irmão para o pai
15      node->keys[idx] = sibling->keys[0];
16
17      // Ajusta chaves e filhos do irmão
18      for (int i = 1; i < sibling->n; i++) {
19          sibling->keys[i - 1] = sibling->keys[i];
20      }
21      if (!sibling->is_leaf) {
22          for (int i = 1; i <= sibling->n; i++) {
23              sibling->children[i - 1] = sibling->children[
                    i];
24          }
25      }
26
```

```
27      child->n++;
28      sibling->n--;
29  }
```

9.5 Árvores B+

Uma árvore B+ é uma variação da árvore B projetada para otimizar o desempenho de buscas em grandes volumes de dados, especialmente em sistemas de banco de dados e arquivos.

A principal diferença em relação à árvore B tradicional é que:

- Todas as chaves e valores **são armazenados apenas nas folhas**.
- Os nós internos armazenam apenas chaves de direcionamento, usadas para navegação.
- As folhas são **encadeadas entre si**, formando uma lista ligada ordenada, o que permite percorrer intervalos de chaves com muita eficiência.

Essa estrutura permite que:

- A busca por um único valor continue eficiente, com complexidade $O(\log n)$.
- A leitura sequencial de registros ordenados (como em operações de varredura ou range scan) seja extremamente rápida.

Devido a essas características, árvores B+ são amplamente utilizadas em sistemas de gerenciamento de banco de dados, onde operações com intervalos de chaves são frequentes.

Árvores B e B+ são amplamente utilizadas em sistemas que exigem acesso eficiente a grandes volumes de dados armazenados em memória secundária. A seguir, destacamos algumas implementações conhecidas:

- **Sistemas de arquivos:**
  - **NTFS (Windows)** usa uma variação de árvore B+ chamada *Master File Table (MFT)* para indexar metadados de arquivos.
  - **ext4 (Linux)** utiliza uma estrutura baseada em árvore B para gerenciar blocos de dados e diretórios grandes.
  - **HFS+ e APFS (macOS)** também empregam variações de árvores B+ para indexação de arquivos.

- **Bancos de dados relacionais:**
  - **PostgreSQL** usa árvores B+ para seus índices padrão (tipo `btree`), otimizando buscas e operações de ordenação.
  - **MySQL (InnoDB)** também utiliza árvores B+ para a implementação de índices primários e secundários.
  - **SQLite** implementa uma variação compacta de árvore B para representar suas páginas de índice e dados.

Esses exemplos demonstram a robustez e a versatilidade das árvores B e B+, especialmente em contextos onde a eficiência de acesso e atualização em disco é crítica.

Capítulo 10

Árvores Vermelho-Preto

Árvores balanceadas são estruturas fundamentais para garantir eficiência em operações de busca, inserção e remoção em tempo logarítmico. No capítulo anterior, estudamos as árvores B, voltadas principalmente para sistemas de armazenamento externo, como bancos de dados e sistemas de arquivos. Antes delas, vimos as árvores AVL, que mantêm um balanceamento rigoroso por meio de diferenças de altura entre subárvores.

As árvores vermelho-preto oferecem uma abordagem diferente: relaxam o critério de balanceamento das AVL em troca de algoritmos mais simples e eficientes para inserção e remoção. Em vez de controlar diferenças de altura, usam uma coloração (vermelha ou preta) em cada nó, junto a regras que garantem que a altura da árvore permaneça proporcional a $\log n$.

Neste capítulo, estudaremos as propriedades que definem as árvores vermelho-preto, os algoritmos de inserção e remoção que mantêm essas propriedades.

Uma **árvore vermelho-preto** é uma árvore binária de busca em que cada nó contém, além da chave, uma **cor**: vermelha ou preta. O balanceamento da árvore é garantido por um conjunto de propriedades que limitam a forma como os nós vermelhos e pretos podem ser organizados. Essas regras asseguram que a altura da árvore permaneça proporcional a $\log n$.

As cinco propriedades fundamentais de uma árvore vermelho-preto são:

1. **Cada nó é vermelho ou preto.**
   A cor é um atributo armazenado em cada nó da árvore.

2. **A raiz é sempre preta.**
   Essa regra é aplicada após cada operação que altera a árvore.

3. **Todas as folhas (nós nulos) são pretas.**
   Consideramos que todas as referências nulas – ou seja, os ponteiros para filhos inexistentes – são nós pretos.

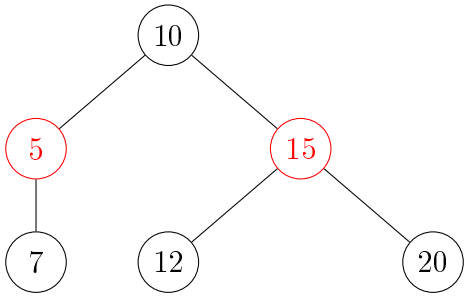
4. **Um nó vermelho não pode ter filhos vermelhos.**
   Ou seja, dois nós vermelhos nunca aparecem consecutivamente em um caminho.

5. **Todos os caminhos de um nó até qualquer folha descendente contêm o mesmo número de nós pretos.**
   Esse número é chamado de *altura preta* do nó.

Essas propriedades impõem uma estrutura que impede o crescimento exagerado da altura, mesmo sem exigir um balanceamento rígido como nas árvores AVL.

Exemplo

A figura abaixo ilustra uma árvore vermelho-preto que satisfaz todas as propriedades:



As propriedades das árvores vermelho-preto impõem restrições que evitam o crescimento descontrolado da altura da árvore, mesmo sem exigir um balanceamento tão rigoroso quanto o das árvores AVL. O segredo está em como as cores limitam a estrutura dos caminhos da raiz até as folhas.

A chave da análise está na **altura preta**, definida como o número de nós pretos em qualquer caminho da raiz até uma folha. Pela propriedade (5), todos esses caminhos têm a mesma altura preta, digamos $bh(x)$ para um nó x. Isso significa que os caminhos não podem variar muito em comprimento: se um caminho tivesse muitos nós vermelhos consecutivos, violaria a propriedade (4), que proíbe dois vermelhos seguidos. Assim, o número de nós vermelhos em qualquer caminho é no máximo igual ao número de nós pretos.

Logo, o **comprimento máximo de qualquer caminho** da raiz até uma folha é no máximo o dobro da altura preta. Isso implica:

$$\text{altura da árvore} \ \leq 2 \cdot bh(\text{raiz})$$

Além disso, o número mínimo de nós em uma árvore com altura preta bh ocorre quando todos os nós pretos estão intercalados com vermelhos, ou seja, com altura total $2 \cdot bh$. Mostra-se por indução que uma árvore vermelho-preto com n nós tem altura no máximo:

$$\text{altura} \leq 2 \log(n + 1)$$

Isso garante que todas as operações de inserção, remoção e busca, que percorrem a árvore do topo até uma folha, têm custo assintótico de $O(\log n)$ no pior caso, mantendo a eficiência da estrutura mesmo após várias atualizações.

10.1 Relação com as Árvores 2-3

Uma maneira elegante de compreender as árvores vermelho-preto é enxergá-las como uma representação binária das **árvores 2-3**. Essa correspondência, proposta por Robert Sedgewick, revela que as propriedades das árvores vermelho-preto são equivalentes às das árvores 2-3 e explica por que elas mantêm o balanceamento de forma natural.

Árvores 2-3

Uma árvore 2-3 é uma árvore de busca balanceada em que cada nó pode conter uma ou duas chaves:

- Um **2-nó** contém uma única chave e dois filhos (à esquerda e à direita).
- Um **3-nó** contém duas chaves e três filhos (à esquerda, ao centro e à direita).

Todas as folhas de uma árvore 2-3 estão no mesmo nível, o que garante que sua altura seja sempre proporcional a $\log n$. A ideia central das árvores vermelho-preto é simular essa mesma estrutura usando apenas ponteiros binários.

Correspondência entre 2-3 e Vermelho-Preto

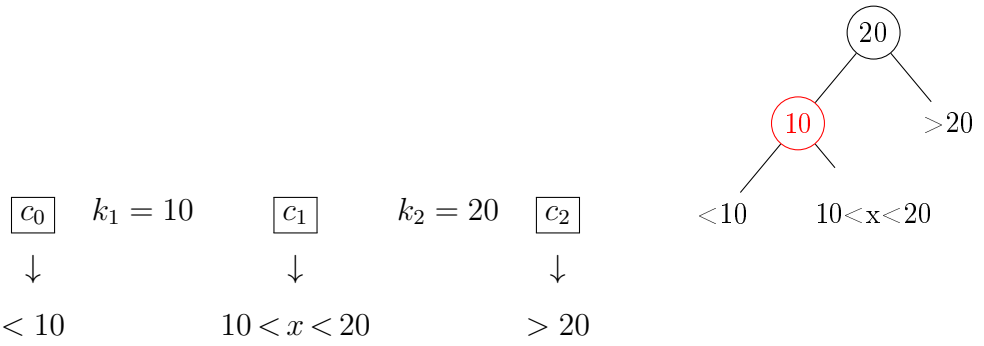
A correspondência entre as duas estruturas é simples:

- Cada **2-nó** de uma árvore 2-3 é representado por um nó **preto** isolado em uma árvore vermelho-preto.

- Cada **3-nó** é representado por dois nós conectados por um **link vermelho inclinado à esquerda**.

Em outras palavras, o **link vermelho** une duas chaves que pertenciam ao mesmo nó da árvore 2-3. Os **links pretos** representam conexões entre nós distintos da árvore 2-3.

Árvore 2-3 **Árvore vermelho-preto**



Nessa codificação, cada **link vermelho à esquerda** representa a fusão de dois nós consecutivos da árvore binária em um único 3-nó da árvore 2-3. Por convenção, as árvores vermelho-preto são implementadas com links vermelhos sempre inclinados à esquerda — isso evita ambiguidade e simplifica os algoritmos.

Equivalência estrutural

Essa correspondência garante que:

- Não existem dois links vermelhos consecutivos — o que equivaleria a um nó 4 em uma árvore 2-3, proibido por definição.

- Todo caminho da raiz até uma folha contém o mesmo número de links pretos — o que corresponde à propriedade de que todas as folhas de uma árvore 2-3 estão na mesma profundidade.

Assim, cada árvore vermelho-preto é uma árvore binária de busca que mantém exatamente as mesmas restrições estruturais de uma árvore 2-3, mas representadas por cores em vez de múltiplas chaves por nó.

Implicações práticas

Ver as árvores vermelho-preto como codificações de árvores 2-3 tem duas vantagens conceituais importantes:

1. Permite compreender suas operações (inserção e remoção) como transformações locais que preservam a equivalência com a árvore 2-3.

2. Explica por que a altura das árvores vermelho-preto está limitada por $2\log(n+1)$: é a altura de uma árvore 2-3 "expandida" em forma binária.

Em outras palavras, as árvores vermelho-preto combinam a **estrutura balanceada** das árvores 2-3 com a **simplicidade** das árvores binárias de busca. Essa visão servirá de base para entender as operações de inserção e remoção que estudaremos a seguir.

10.2 Função `inserir`

10.3 Inserção

A inserção em uma árvore vermelho-preto segue duas etapas principais:

1. Inserir o novo nó como em uma árvore binária de busca (ABB) comum.

2. Colorir o novo nó como **vermelho** e, em seguida, executar uma sequência de **recolorações** e possivelmente **rotações** para restaurar as propriedades vermelho-preto.

Colorir o novo nó de vermelho ajuda a manter a propriedade (5) (altura preta), mas pode violar a propriedade (4) (nenhum nó vermelho pode ter filho vermelho). Para resolver essa violação, aplicamos um procedimento de correção que depende da cor do *tio* do novo nó.

Leitura da inserção via árvores 2-3

Como vimos, uma árvore vermelho-preto pode ser interpretada como uma codificação binária de uma árvore 2-3 (links vermelhos representam a fusão local que forma um 3-nó). Nessa visão, cada passo da inserção RB corresponde a uma operação natural em 2-3:

1. **Inserir como ABB $\Rightarrow$ descer até a folha na 2-3** e tentar acomodar a nova chave no nó corrente (um 2-nó vira 3-nó; um 3-nó pode "estourar").

2. **Colorir de vermelho ⇒ juntar a chave ao nó 2-3 corrente**, isto é, representar um 3-nó por um link vermelho.

3. **Recolorar/rotacionar ⇒ corrigir estouros (split) e promover a mediana**, quando um nó temporário com três chaves (nó 4) aparece.

Tradução caso a caso. Ao tratar a violação "pai vermelho", inspecionamos a cor do tio (na RB); na 2-3, isso indica se há *split* ou apenas recodificação local:

Caso RB	Ação na RB	Leitura em 2-3
1. Tio vermelho	Flip de cores (pai/tio → preto; avô → vermelho)	*Split* de nó 4: mediana sobe; pode propagar para cima.
2. Tio preto, zig-zag	Rotação simples para "endireitar"	Só recodificação: 3-nó permanece o mesmo, forma canônica.
3. Tio preto, em linha	Rotação + flip de cores	*Split* local: mediana torna-se pai; filhos viram 2-nós.

Mnemônico.

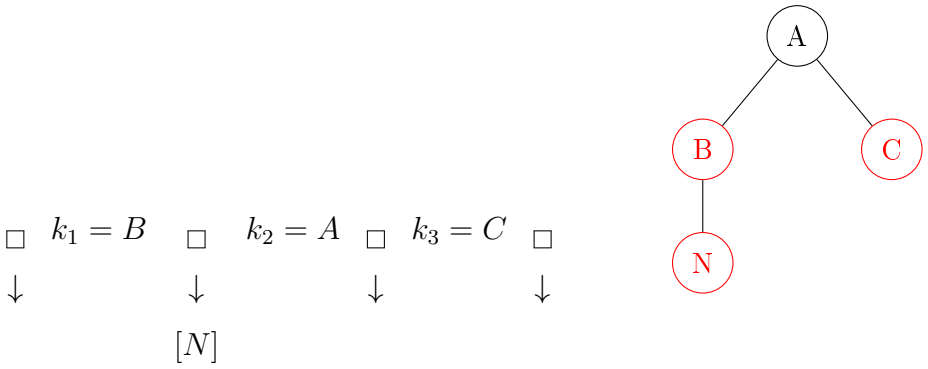
$$\underbrace{\text{flip de cores}}_{\text{RB}} \iff \underbrace{\text{split de nó 4}}_{\text{2-3}} \qquad \text{e} \qquad \underbrace{\text{rotação}}_{\text{RB}} \iff \underbrace{\text{reorientar / elevar a mediana}}_{\text{2-3}}.$$

Os principais casos de rebalanceamento são:

- **Caso 1 (Tio vermelho):** recolorimos pai, tio e avô.

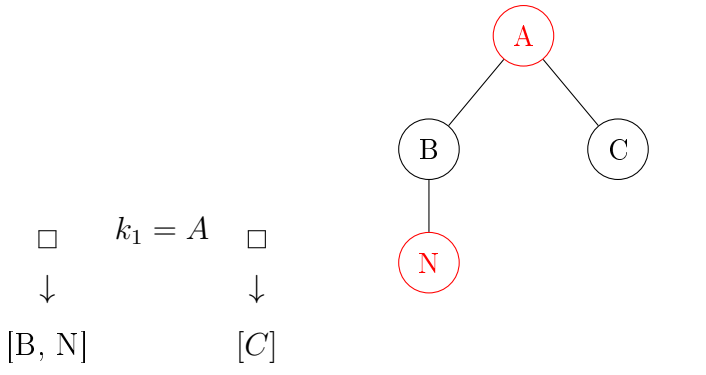
  Esse caso corresponde, na árvore 2-3, à situação em que a inserção gera temporariamente um **nó 4**, ou seja, um nó com três chaves `(B, A, C)`. A recoloração na árvore vermelho-preto equivale a dividir esse nó 4 em três nós 2 separados, promovendo a chave do meio (`A`) para o nível acima.

Árvore 2-3 **Árvore vermelho-preto**



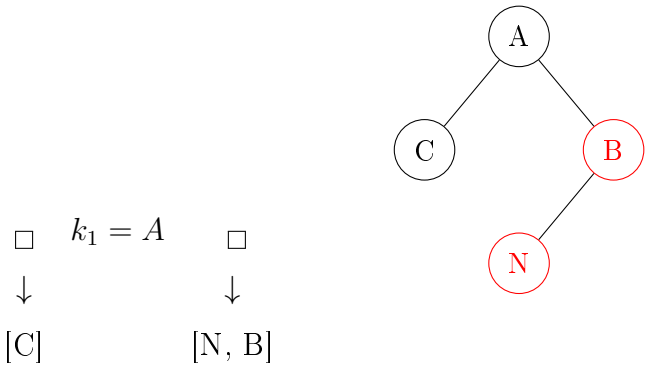
Após a recoloração, a chave `A` é promovida e as chaves `B` e `C` se tornam nós pretos independentes. Na árvore 2-3, isso corresponde à divisão do nó 4 em três nós 2, mantendo o balanceamento da altura.

Árvore 2-3 **Árvore vermelho-preto**



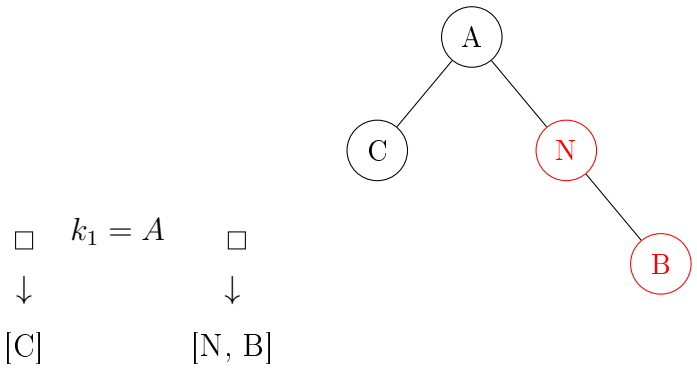
- **Caso 2 (Tio preto e filho em zig-zag):** rotação simples para transformar em Caso 3.

Árvore 2-3 **Árvore vermelho-preto**



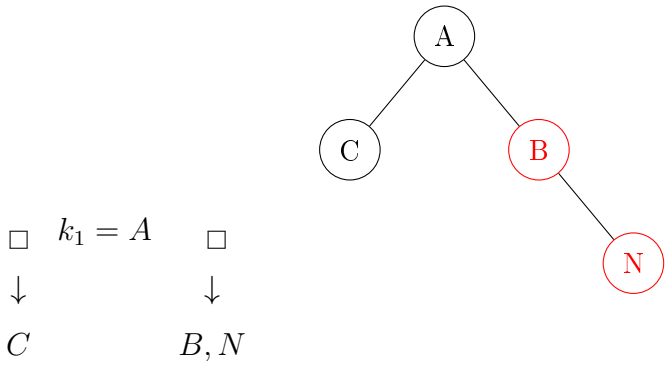
Interpretação 2-3: o filho direito de [A] é um **3-nó** [N, B]. Na vermelho-preto, porém, esse 3-nó está codificado com um **link vermelho torto** (direita-depois-esquerda): é o "zig-zag".

Árvore 2-3 **Árvore vermelho-preto**



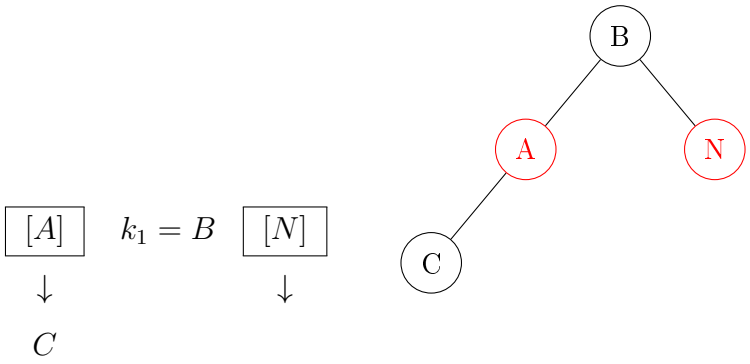
- **Caso 3 (Tio preto e filho em linha):** rotação e recoloração.

Árvore 2-3 **Árvore vermelho-preto**



Leitura 2-3: o filho direito de $[A]$ é o **3-nó** $[B, N]$ (com $B < N$). Na vermelho-preto, corrigimos a codificação (link vermelho à direita e vermelho duplo) com **rotação à esquerda em** A seguida de **troca de cores**.

Árvore 2-3 **Árvore vermelho-preto**



A seguir, um esboço da função de inserção com chamada à função de correção:

```
1  typedef enum { VERMELHO, PRETO } Cor;
2
3  typedef struct No {
4      int chave;
5      Cor cor;
```

```
    struct No *esq, *dir, *pai;
} No;

void inserir(No **raiz, int chave) {
    No *novo = criar_no(chave, VERMELHO);
    inserir_abb(raiz, novo);
    corrigir_insercao(raiz, novo);
}
```

Programa 10.1: Inserção em árvore vermelho-preto (esboço)

A função `corrigir_insercao` é responsável por garantir que as propriedades da árvore vermelho-preto sejam restauradas após a inserção.

```
void corrigir_insercao(No **raiz, No *n) {
    while (n != *raiz && n->pai->cor == VERMELHO) {
        No *pai = n->pai;
        No *avo = pai->pai;

        if (pai == avo->esq) {
            No *tio = avo->dir;

            if (tio && tio->cor == VERMELHO) {
                // Caso 1: tio vermelho
                pai->cor = PRETO;
                tio->cor = PRETO;
                avo->cor = VERMELHO;
                n = avo; // sobe para verificar novo
                    conflito
            } else {
                if (n == pai->dir) {
                    // Caso 2: zig-zag
                    n = pai;
                    rotacao_esquerda(raiz, n);
                }
                // Caso 3: linha reta
                pai = n->pai;
                avo = pai->pai;
                pai->cor = PRETO;
                avo->cor = VERMELHO;
                rotacao_direita(raiz, avo);
            }
        } else {
            // simétrico: pai está à direita
            No *tio = avo->esq;
```

```
31
32              if (tio && tio->cor == VERMELHO) {
33                  // Caso 1 (simétrico)
34                  pai->cor = PRETO;
35                  tio->cor = PRETO;
36                  avo->cor = VERMELHO;
37                  n = avo;
38              } else {
39                  if (n == pai->esq) {
40                      // Caso 2 (simétrico)
41                      n = pai;
42                      rotacao_direita(raiz, n);
43                  }
44                  // Caso 3 (simétrico)
45                  pai = n->pai;
46                  avo = pai->pai;
47                  pai->cor = PRETO;
48                  avo->cor = VERMELHO;
49                  rotacao_esquerda(raiz, avo);
50              }
51          }
52      }
53
54      (*raiz)->cor = PRETO;
55  }
```

Programa 10.2: Correção após inserção em árvore vermelho-preto

10.4 Função `remover`

A operação de remoção em árvores vermelho-preto segue a mesma ideia geral da remoção em árvores binárias de busca (ABB):

1. Localizamos o nó a ser removido.

2. Se ele tiver dois filhos, substituímos seu valor pelo de seu sucessor imediato (ou antecessor) e então removemos esse sucessor, que terá no máximo um filho.

3. O nó removido de fato terá, portanto, no máximo um filho (como em uma ABB).

No entanto, como as cores dos nós afetam diretamente as propriedades da árvore, a remoção pode gerar violações nas propriedades vermelho-preto, principalmente a da altura preta. Isso acontece especialmente quando se remove um nó preto.

Para lidar com essas violações, introduzimos o conceito de *nó duplamente preto*, que aparece temporariamente durante o processo de correção. Ele representa um déficit de cor e será tratado com rotações e recolorações, até que todas as propriedades sejam restauradas.

Abaixo está um esboço da função de remoção, com chamada a uma função de correção:

```
void remover(No **raiz, int chave) {
    No *n = buscar(*raiz, chave);
    if (n == NULL) return;

    No *substituto = n;
    Cor cor_original = substituto->cor;

    No *x; // nó que sobe (pode ser nulo)

    if (n->esq == NULL) {
        x = n->dir;
        transplantar(raiz, n, n->dir);
    } else if (n->dir == NULL) {
        x = n->esq;
        transplantar(raiz, n, n->esq);
    } else {
        substituto = minimo(n->dir);
        cor_original = substituto->cor;
        x = substituto->dir;

        if (substituto->pai == n) {
            if (x) x->pai = substituto;
        } else {
            transplantar(raiz, substituto, substituto->
               dir);
            substituto->dir = n->dir;
            substituto->dir->pai = substituto;
        }

        transplantar(raiz, n, substituto);
        substituto->esq = n->esq;
        substituto->esq->pai = substituto;
```

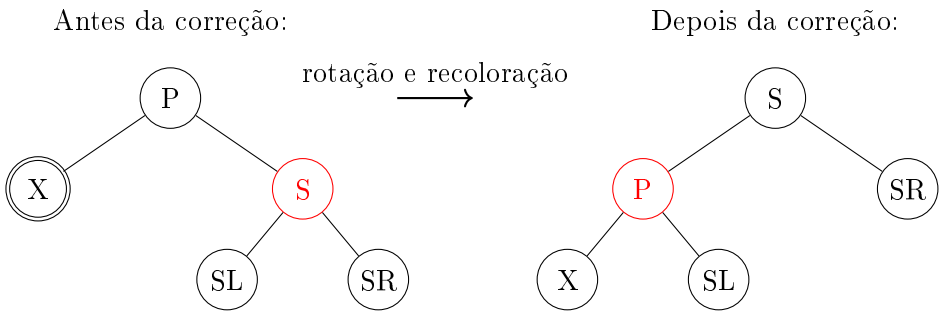
```
32          substituto->cor = n->cor;
33      }
34
35      if (cor_original == PRETO)
36          corrigir_remocao(raiz, x);
37  }
```

Programa 10.3: Remoção em árvore vermelho-preto

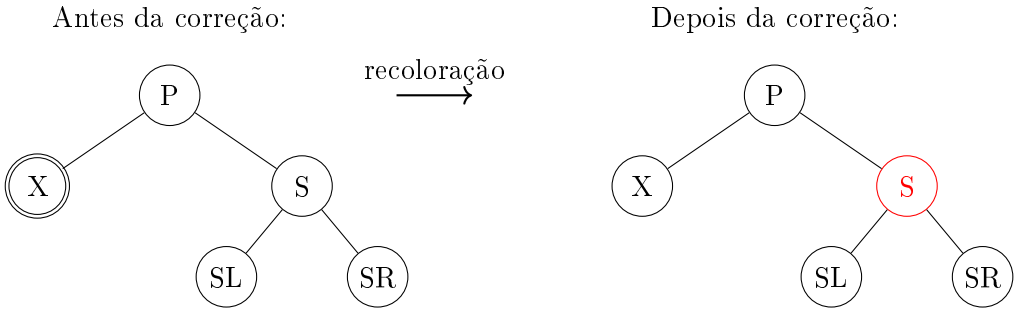
A função auxiliar `corrigir_remocao` é responsável por eliminar o nó duplamente preto que pode surgir após a remoção de um nó preto.

Durante a correção, tratamos uma série de casos que dependem da cor do irmão do nó duplamente preto e de seus filhos:

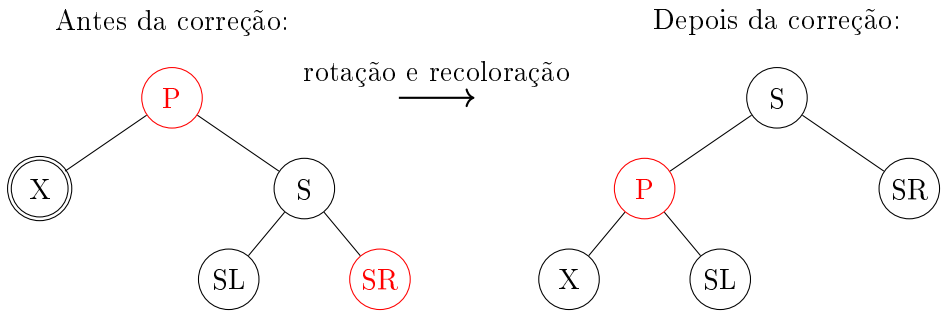
- **Caso 1:** irmão vermelho



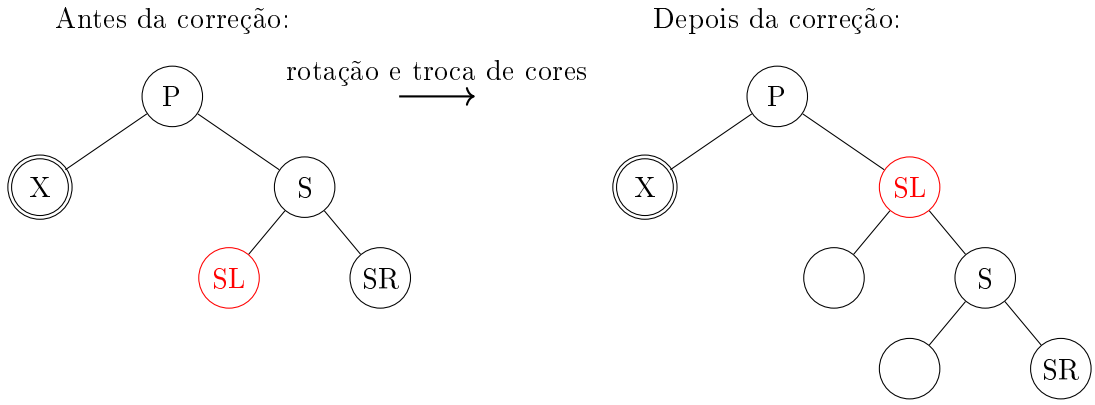
- **Caso 2:** irmão preto com filhos pretos



- **Caso 3:** irmão preto com filho vermelho longe



- **Caso 4:** irmão preto com filho vermelho próximo



Cada um desses casos requer uma combinação específica de rotações e recolorações. A complexidade da remoção é maior do que a da inserção, mas a altura da árvore continua limitada por $2\log(n+1)$, e todas as operações mantêm custo assintótico de $O(\log n)$.

Parte III

Grafos

Capítulo 11

Introdução a Grafos

Grafos são uma das abstrações mais versáteis da Computação. Sempre que precisamos modelar entidades e suas relações — pessoas e amizades, cidades e estradas, páginas e links, tarefas e dependências — estamos, essencialmente, lidando com grafos. Nesta aula, apresentaremos a ideia central, os conceitos básicos, os principais tipos, algumas propriedades elementares, o problema clássico das pontes de Königsberg, aplicações, e as representações mais usadas em código (matriz e lista de adjacência).

11.1 O que são grafos

Um **grafo** é formado por um conjunto de **vértices** (ou *nós*) e um conjunto de **arestas** que conectam pares de vértices. Intuitivamente, vértices representam objetos e arestas representam relações entre esses objetos.

Exemplos cotidianos incluem:

- **Rede social:** pessoas são vértices; amizades são arestas.
- **Mapa:** cidades são vértices; estradas são arestas.
- **Dependências:** tarefas são vértices; uma aresta indica que uma tarefa depende de outra.

11.2 Conceitos fundamentais

- **Vértice (nó):** entidade básica que queremos modelar.
- **Aresta (ligação):** relação entre dois vértices.

- **Grau:** número de arestas incidentes a um vértice (em dígrafos, distinguimos *grau de entrada* e *grau de saída*).
- **Caminho:** sequência de vértices conectados por arestas.
- **Ciclo:** caminho que começa e termina no mesmo vértice.
- **Conectividade:** dois vértices estão conectados se existe um caminho entre eles; um grafo é *conexo* se qualquer vértice alcança qualquer outro.
- **Componente conexa:** subgrafo conexo maximal.

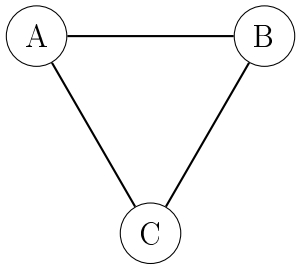
11.3 Tipos de grafos

Dependendo da natureza da relação, usamos variantes de grafos:

Tipo	Descrição	Exemplo prático
Não direcionado	Arestas sem orientação (relação bidirecional).	Amizade em redes sociais.
Direcionado (dígrafo)	Arestas orientadas ($u \to v$).	Seguidores no Twitter, fluxo de dados.
Ponderado	Arestas com pesos (custo, tempo, distância).	Rotas de transporte.
Multigrafo	Arestas paralelas entre o mesmo par de vértices.	Linhas distintas entre duas cidades.
Com laços	Aresta que liga um vértice a si próprio.	Dependência recursiva.

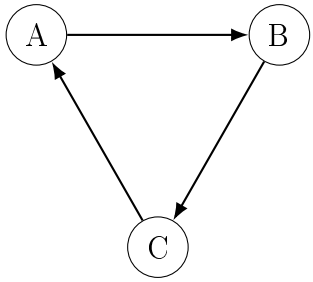
Diagramas ilustrativos

Grafo não direcionado



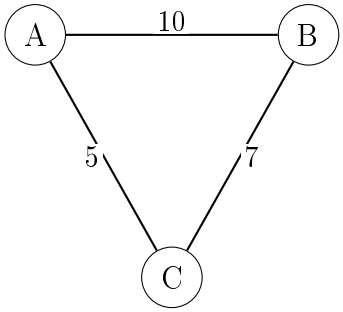
Arestas sem orientação.

Grafo direcionado (dígrafo)



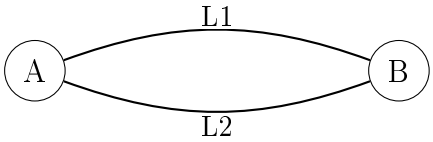
Arestas com sentido ($u \to v$).

Grafo ponderado



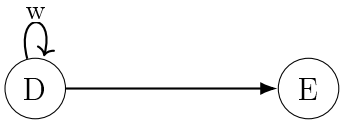
Cada aresta carrega um custo ou distância.

Multigrafo (arestas paralelas)



Mais de uma aresta entre o mesmo par de vértices.

Grafo com laço



Aresta que sai e retorna ao mesmo vértice.

11.4 Propriedades básicas

- **Ordem** de um grafo: número de vértices ($|V|$).
- **Tamanho** de um grafo: número de arestas ($|E|$).
- **Soma dos graus (não direcionado):** $\sum_{v \in V} \deg(v) = 2|E|$.

- **Em dígrafos:** $\sum \deg^+(v) = \sum \deg^-(v) = |E|$.
- **Limite superior de arestas (sem laços):**
  - *Grafo não direcionado simples:* $|E|_{\max} = \dfrac{|V|(|V|-1)}{2}$.
  - *Grafo direcionado (dígrafo) simples:* $|E|_{\max} = |V|(|V|-1)$.
- **Limite inferior de arestas em grafos conexos.**
  - *Grfao não direcionado simples:* se o grafo é conexo, então

$$|E| \geq |V| - 1,$$

Essas relações ajudam a checar consistência e servem de base para algoritmos (por exemplo, muitos percorrem vizinhos de um vértice, logo custos dependem do grau).

11.5 O problema das pontes de Königsberg

Em 1736, Euler modelou o problema de atravessar todas as pontes da cidade de Königsberg sem repetir nenhuma. Abstraindo regiões como vértices e pontes como arestas, a pergunta torna-se: *existe um caminho que use cada aresta exatamente uma vez?* (caminho euleriano).

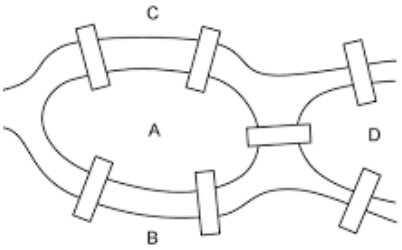
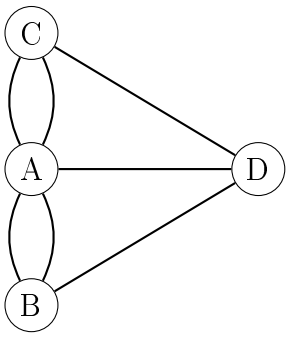


Figura 11.1: As 7 pontos de Konigsberg.

Euler mostrou que para que exista um caminho (euleriano), o grafo deve ser conexo e ter exatamente dois vértices de grau ímpar. Essa condição aparece porque, ao percorrer as arestas, todo vértice intermediário é visitado um número par de vezes: a cada vez que o caminho entra por uma aresta, ele precisa sair por outra, e assim o número total de arestas incidentes a esses vértices é par. Já os vértices inicial e final se comportam de modo diferente: no vértice de partida há uma saída a mais que entrada, e no vértice de chegada há uma entrada a mais que saída, o que faz com que ambos tenham grau

ímpar. Se houvesse mais de dois vértices ímpares, seria necessário começar ou terminar o percurso em mais de dois pontos, o que é impossível em um único trajeto.

Abstração de Königsberg



11.6 Aplicações

Grafos aparecem em:

- **Redes sociais e informação:** comunidade, influência, recomendações.
- **Rotas e logística:** caminhos mínimos, cobertura, árvores geradoras.
- **Dependências e planejamento:** ordenação topológica em DAGs.
- **Computação e redes:** roteamento, conectividade, tolerância a falhas.
- **Ciências naturais:** interações biológicas, redes metabólicas, ecologia.

11.7 Tipo Abstrato de Dados (TAD) Grafos

Nos capítulos anteriores, estudamos dois tipos de dicionário: o **dicionário simples**, baseado em tabelas de dispersão, e o **dicionário ordenado**, implementado com árvores binárias de busca. Essas estruturas tinham como foco associar chaves e valores de forma eficiente, seja pelo acesso direto (hash) ou pela ordenação (árvores). Agora, no estudo de grafos, ampliamos esse mesmo princípio de abstração para um tipo de dado mais geral, no qual os elementos não estão apenas associados a valores, mas conectados entre si.

Assim como nos dicionários, a noção de *Tipo Abstrato de Dados* (TAD) continua essencial. O **TAD Grafo** define um conjunto de operações que permitem manipular grafos sem se comprometer com a forma como eles são

representados em memória. Essa separação entre o nível abstrato e o nível de implementação é fundamental porque existem várias formas possíveis de representar um grafo, cada uma com vantagens e limitações específicas.

Do ponto de vista abstrato, um TAD Grafo deve permitir, entre outras, as seguintes operações:

- `criarGrafo()`: cria um grafo vazio.
- `inserirVertice(v)` e `inserirAresta(u, v)`: adicionam vértices e arestas.
- `removerVertice(v)` e `removerAresta(u, v)`: removem elementos existentes.
- `adjacentes(u)`: devolve os vértices conectados a u.
- `grau(v)`: retorna o grau de um vértice.
- `numVertices()` e `numArestas()`: retornam as quantidades de vértices e arestas.

Essas operações definem o comportamento lógico do grafo, isto é, o que ele deve ser capaz de fazer, sem indicar como faz. Dessa forma, o mesmo TAD pode ter múltiplas implementações compatíveis com os algoritmos que o utilizam — exatamente como vimos nos TADs anteriores. Essa abordagem é adequada porque permite estudar e comparar diferentes representações, trocando a estrutura interna sem alterar a interface pública.

As duas representações mais tradicionais de um grafo são as seguintes:

- **Matriz de adjacência:** consiste em uma matriz $|V| \times |V|$ na qual a célula (i, j) indica a presença (ou o peso) de uma aresta entre os vértices i e j. Essa representação é simples e permite verificar a existência de uma aresta em tempo constante $O(1)$, mas tem custo de memória proporcional ao quadrado do número de vértices:

  $$\text{Espaço ocupado} = O(|V|^2)$$

  Mesmo que o grafo tenha poucas arestas (isto é, seja esparso), a matriz reserva espaço para todas as combinações possíveis de vértices.

- **Lista de adjacência:** para cada vértice, mantém-se uma lista com seus vizinhos. Essa estrutura ocupa espaço proporcional ao número de vértices mais o número de arestas:

$$\text{Espaço ocupado} = O(|V| + |E|)$$

  Por isso, é muito mais eficiente para grafos grandes e esparsos, nos quais $|E| \ll |V|^2$.

Em resumo, a matriz de adjacência é vantajosa em grafos *densos*, com muitas conexões, pois o custo de $O(|V|^2)$ é justificado pela rapidez de acesso. Já a lista de adjacência é superior em grafos *esparsos*, que têm poucas arestas, economizando memória e tornando as operações de iteração sobre vizinhos mais eficientes.

O uso do TAD Grafo nos permite abstrair essas diferenças: podemos implementar algoritmos de busca, caminhos mínimos ou árvores geradoras sem nos preocupar com o formato interno da representação. Se precisarmos otimizar para memória ou velocidade, basta trocar a implementação do TAD, mantendo o mesmo conjunto de operações.

11.8 TAD Grafo com Matriz de Adjacência

A seguir implementamos um grafo não-direcionado, simples e não-ponderado usando **matriz de adjacência**. Essa escolha facilita o teste de existência de aresta em tempo $O(1)$, ao custo de memória $O(|V|^2)$. As seis operações expostas aqui compõem uma interface mínima e suficiente para algoritmos clássicos (BFS, DFS, etc.).

Políticas de implementação

- **Índices de vértices**: inteiros de 0 a $nV - 1$.
- **Inserção de vértice**: cria-se um novo índice no final (incrementa nV).
- **Remoção de vértice**: *compactação*: a última linha/coluna é movida para a posição do vértice removido, e nV é decrementado.
- **Inserção/remoção de aresta**: atualizam nE (# de arestas) e mantêm a simetria da matriz.
- **Adjacentes(u)**: retorna, em um vetor externo, os vizinhos de u.

Cabeçalhos e estrutura

```c
// grafo_matriz.h
#ifndef GRAFO_MATRIZ_H
#define GRAFO_MATRIZ_H

typedef struct {
    int maxV;     // capacidade máxima
    int nV;       // número atual de vértices
    int nE;       // número atual de arestas
    unsigned char *M; // matriz nV x nV (alocada com maxV x maxV)
} Graph;

// Acesso M[u][v]
#define MAT(G,u,v) ((G)->M[(u)*(G)->maxV + (v)])

// Criação/Destruição
Graph* grafo_criar(int maxV);
void    grafo_destruir(Graph *G);

// Operações pedidas (6)
int grafo_inserir_vertice(Graph *G);           // retorna índice do novo vértice ou -1
int grafo_remover_vertice(Graph *G, int v);    // retorna 0 ok, -1 erro
int grafo_inserir_aresta(Graph *G, int u, int v);   // 0 ok, -1 erro
int grafo_remover_aresta(Graph *G, int u, int v);   // 0 ok, -1 erro
int grafo_adjacentes(const Graph *G, int u, int *out, int max_out); // retorna k

// Utilitários
int grafo_num_vertices(const Graph *G);
int grafo_num_arestas(const Graph *G);

#endif // GRAFO_MATRIZ_H
```

Implementação

```c
// grafo_matriz.c
```

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "grafo_matriz.h"

static int vertice_valido(const Graph *G, int v) {
    return (G && v >= 0 && v < G->nV);
}

Graph* grafo_criar(int maxV) {
    if (maxV <= 0) return NULL;
    Graph *G = (Graph*) malloc(sizeof(Graph));
    if (!G) return NULL;
    G->maxV = maxV;
    G->nV = 0;
    G->nE = 0;
    G->M = (unsigned char*) calloc((size_t)maxV * (size_t)maxV, sizeof(unsigned char));
    if (!G->M) { free(G); return NULL; }
    return G;
}

void grafo_destruir(Graph *G) {
    if (!G) return;
    free(G->M);
    free(G);
}

int grafo_num_vertices(const Graph *G) { return G ? G->nV : 0; }
int grafo_num_arestas (const Graph *G) { return G ? G->nE : 0; }

int grafo_inserir_vertice(Graph *G) {
    if (!G) return -1;
    if (G->nV == G->maxV) return -1; // capacidade esgotada
    int v = G->nV++;
    // zera linha e coluna do novo vértice dentro do bloco atual nV x nV
    for (int i = 0; i < G->nV; ++i) {
        MAT(G, v, i) = 0;
        MAT(G, i, v) = 0;
```

```c
    }
    return v;
}

static int soma_linha(const Graph *G, int v) {
    int deg = 0;
    for (int j = 0; j < G->nV; ++j) deg += MAT(G, v, j);
    return deg;
}

int grafo_remover_vertice(Graph *G, int v) {
    if (!vertice_valido(G, v)) return -1;
    // remover arestas incidentes (atualiza nE)
    int deg_v = soma_linha(G, v);
    G->nE -= deg_v; // cada aresta é contada 1x na linha v (grafo não-direcionado, simetria garantida)

    int last = G->nV - 1;
    if (v != last) {
        // move a linha 'last' para 'v'
        for (int j = 0; j < G->nV; ++j) {
            MAT(G, v, j) = MAT(G, last, j);
        }
        // move a coluna 'last' para 'v'
        for (int i = 0; i < G->nV; ++i) {
            MAT(G, i, v) = MAT(G, i, last);
        }
        // zera a linha/coluna 'last' (não obrigatório, mas ajuda a depurar)
        for (int j = 0; j < G->nV; ++j) {
            MAT(G, last, j) = 0;
            MAT(G, j, last) = 0;
        }
        /* Ajuste de nE: já descontamos as arestas de v; mas ao mover 'last' para 'v',
           preservamos o número de arestas restantes pois copiamos simetricamente.
           Não há dupla contagem aqui pois as incidências de 'last' continuam existindo,
           agora rotuladas como 'v'. */
    } else {
        // apenas zera última linha/coluna
        for (int j = 0; j < G->nV; ++j) {
```

```
78              MAT(G, last, j) = 0;
79              MAT(G, j, last) = 0;
80          }
81      }
82      G->nV--;
83      return 0;
84  }
85
86  int grafo_inserir_aresta(Graph *G, int u, int v) {
87      if (!vertice_valido(G, u) || !vertice_valido(G, v))
           return -1;
88      if (u == v) return -1; // sem laços neste TAD simples
89      if (MAT(G, u, v)) return 0; // já existe; idempotente
90      MAT(G, u, v) = 1;
91      MAT(G, v, u) = 1;
92      G->nE++;
93      return 0;
94  }
95
96  int grafo_remover_aresta(Graph *G, int u, int v) {
97      if (!vertice_valido(G, u) || !vertice_valido(G, v))
           return -1;
98      if (u == v) return -1;
99      if (!MAT(G, u, v)) return 0; // não existe;
           idempotente
100     MAT(G, u, v) = 0;
101     MAT(G, v, u) = 0;
102     G->nE--;
103     return 0;
104 }
105
106 int grafo_adjacentes(const Graph *G, int u, int *out, int
    max_out) {
107     if (!vertice_valido(G, u) || !out || max_out <= 0)
           return -1;
108     int k = 0;
109     for (int v = 0; v < G->nV; ++v) {
110         if (MAT(G, u, v)) {
111             if (k < max_out) out[k] = v;
112             k++;
113         }
114     }
```

```
115     // retorna o número total de vizinhos (mesmo se >
            max_out)
116     return k;
117 }
```

Observação importante (IDs de vértices). Como adotamos *compactação* em `removerVertice`, os índices de vértices podem mudar: o último vértice é movido para a posição removida. Essa política mantém a matriz densa e as operações simples, mas significa que estruturas externas que guardam IDs de vértices precisam ser atualizadas após remoções. Em muitos projetos, quando é necessário manter IDs estáveis, prefere-se *marcar* vértices como inativos em vez de compactar (ao custo de lidar com "buracos").

11.9 TAD Grafo com Listas de Adjacência

Nesta seção implementamos o mesmo TAD Grafo anterior, mas usando **listas de adjacência** em vez de matriz. Essa representação é mais eficiente em termos de memória para grafos esparsos, pois armazena apenas as arestas existentes. Enquanto a matriz exige espaço $O(|V|^2)$ independentemente da densidade do grafo, as listas de adjacência utilizam apenas $O(|V| + |E|)$.

A ideia é simples: cada vértice mantém uma lista encadeada contendo os vértices a ele adjacentes. Inserir uma aresta corresponde a inserir dois nós nessas listas (um em cada direção), e remover uma aresta significa remover ambos os nós. Para operações de vizinhança, basta percorrer a lista correspondente ao vértice.

Cabeçalhos e estrutura

```
1  // grafo_lista.h
2  #ifndef GRAFO_LISTA_H
3  #define GRAFO_LISTA_H
4
5  typedef struct no {
6      int v;              // vértice destino
7      struct no *prox;    // próximo vizinho
8  } No;
9
10 typedef struct {
11     int maxV;     // capacidade máxima
```

```
    int nV;       // número atual de vértices
    int nE;       // número atual de arestas
    No **adj;     // vetor de listas (tamanho maxV)
} Graph;

// Criação/Destruição
Graph* grafo_criar(int maxV);
void   grafo_destruir(Graph *G);

// Operações principais (6)
int grafo_inserir_vertice(Graph *G);
int grafo_remover_vertice(Graph *G, int v);
int grafo_inserir_aresta(Graph *G, int u, int v);
int grafo_remover_aresta(Graph *G, int u, int v);
int grafo_adjacentes(const Graph *G, int u, int *out, int
    max_out);

// Consultas auxiliares
int grafo_num_vertices(const Graph *G);
int grafo_num_arestas(const Graph *G);

#endif // GRAFO_LISTA_H
```

Implementação

```
// grafo_lista.c
#include <stdio.h>
#include <stdlib.h>
#include "grafo_lista.h"

static int vertice_valido(const Graph *G, int v) {
    return (G && v >= 0 && v < G->nV);
}

// cria grafo com listas inicialmente vazias
Graph* grafo_criar(int maxV) {
    if (maxV <= 0) return NULL;
    Graph *G = (Graph*) malloc(sizeof(Graph));
    if (!G) return NULL;
    G->maxV = maxV;
    G->nV = 0;
    G->nE = 0;
```

```
18      G->adj = (No**) calloc(maxV, sizeof(No*));
19      if (!G->adj) { free(G); return NULL; }
20      return G;
21  }
22
23  void grafo_destruir(Graph *G) {
24      if (!G) return;
25      for (int i = 0; i < G->nV; ++i) {
26          No *p = G->adj[i];
27          while (p) {
28              No *tmp = p->prox;
29              free(p);
30              p = tmp;
31          }
32      }
33      free(G->adj);
34      free(G);
35  }
36
37  int grafo_num_vertices(const Graph *G) { return G ? G->nV
        : 0; }
38  int grafo_num_arestas (const Graph *G) { return G ? G->nE
        : 0; }
39
40  // insere novo vértice no final
41  int grafo_inserir_vertice(Graph *G) {
42      if (!G) return -1;
43      if (G->nV == G->maxV) return -1;
44      G->adj[G->nV] = NULL;
45      return G->nV++;
46  }
47
48  // remove todas as arestas incidentes e compacta vetor
49  static void remover_todas_arestas(Graph *G, int v) {
50      // remove arestas que partem de v
51      No *p = G->adj[v];
52      while (p) {
53          grafo_remover_aresta(G, v, p->v);
54          p = G->adj[v];
55      }
56  }
57
58  // remove vértice (com compactação)
```

```c
int grafo_remover_vertice(Graph *G, int v) {
    if (!vertice_valido(G, v)) return -1;
    remover_todas_arestas(G, v);
    int last = G->nV - 1;
    if (v != last) {
        // mover lista do último vértice para a posição v
        G->adj[v] = G->adj[last];
        // ajustar todos os ponteiros que apontavam para 'last'
        for (int i = 0; i < last; ++i) {
            No *p = G->adj[i];
            while (p) {
                if (p->v == last) p->v = v;
                p = p->prox;
            }
        }
    }
    G->nV--;
    return 0;
}

static No* criar_no(int v, No *prox) {
    No *n = (No*) malloc(sizeof(No));
    n->v = v;
    n->prox = prox;
    return n;
}

static int contem(const No *p, int v) {
    while (p) {
        if (p->v == v) return 1;
        p = p->prox;
    }
    return 0;
}

int grafo_inserir_aresta(Graph *G, int u, int v) {
    if (!vertice_valido(G, u) || !vertice_valido(G, v))
       return -1;
    if (u == v) return -1; // sem laços
    if (contem(G->adj[u], v)) return 0; // já existe
    G->adj[u] = criar_no(v, G->adj[u]);
    G->adj[v] = criar_no(u, G->adj[v]);
```

```c
    G->nE++;
    return 0;
}

int grafo_remover_aresta(Graph *G, int u, int v) {
    if (!vertice_valido(G, u) || !vertice_valido(G, v))
       return -1;
    No **pp = &G->adj[u];
    while (*pp && (*pp)->v != v) pp = &(*pp)->prox;
    if (*pp) {
        No *tmp = *pp;
        *pp = tmp->prox;
        free(tmp);
    } else return 0;
    // simétrico
    pp = &G->adj[v];
    while (*pp && (*pp)->v != u) pp = &(*pp)->prox;
    if (*pp) {
        No *tmp = *pp;
        *pp = tmp->prox;
        free(tmp);
    }
    G->nE--;
    return 0;
}

int grafo_adjacentes(const Graph *G, int u, int *out, int
    max_out) {
    if (!vertice_valido(G, u) || !out || max_out <= 0)
       return -1;
    int k = 0;
    No *p = G->adj[u];
    while (p) {
        if (k < max_out) out[k] = p->v;
        k++;
        p = p->prox;
    }
    return k;
}
```

Exemplo de uso do TAD Grafo

Um ponto importante do uso do TAD é que o programa que utiliza o grafo não precisa conhecer sua forma de representação interna. O mesmo código de aplicação funciona tanto para a versão baseada em matriz de adjacência quanto para a versão com listas de adjacência — basta alterar o arquivo de cabeçalho incluído no início do programa.

```c
#include <stdio.h>
// Altere aqui para trocar a implementação:
// #include "grafo_matriz.h"
#include "grafo_lista.h"

int main(void) {
    // cria um grafo com capacidade máxima de 8 vértices
    Graph *G = grafo_criar(8);

    // insere três vértices (índices 0, 1 e 2)
    int a = grafo_inserir_vertice(G);
    int b = grafo_inserir_vertice(G);
    int c = grafo_inserir_vertice(G);

    // adiciona arestas (a-b) e (b-c)
    grafo_inserir_aresta(G, a, b);
    grafo_inserir_aresta(G, b, c);

    // imprime vizinhos do vértice b
    int viz[8];
    int k = grafo_adjacentes(G, b, viz, 8);
    printf("Vizinhos de b: ");
    for (int i = 0; i < k && i < 8; ++i)
        printf("%d ", viz[i]);
    printf("\n|V|=%d |E|=%d\n",
           grafo_num_vertices(G),
           grafo_num_arestas(G));

    // remove o vértice b (a estrutura pode compactar os
       índices)
    grafo_remover_vertice(G, b);
    printf("Depois de remover b: |V|=%d |E|=%d\n",
           grafo_num_vertices(G),
           grafo_num_arestas(G));

    grafo_destruir(G);
```

```
36      return 0;
37  }
```

O programa acima compila e executa de forma idêntica para ambas as implementações. A única diferença está na linha de inclusão do cabeçalho: basta escolher `grafo_matriz.h` ou `grafo_lista.h`. Esse exemplo ilustra a principal vantagem do TAD: o código cliente depende apenas da *interface*, e não da *estrutura interna*. Assim, podemos trocar a implementação — buscando mais eficiência em tempo ou em espaço – sem modificar os algoritmos que utilizam o grafo.

Capítulo 12

Busca em Grafos

Os algoritmos de busca em grafos são ferramentas fundamentais para explorar, percorrer e descobrir propriedades estruturais de um grafo. Eles permitem visitar todos os vértices e arestas de forma sistemática, servindo de base para diversos problemas clássicos, como o cálculo de *caminhos mínimos*, a identificação de *componentes conexas* e a *ordenação topológica* de vértices em grafos direcionados acíclicos.

12.1 Introdução

Muitos problemas em ciência da computação, redes, engenharia e biologia podem ser modelados como grafos. Em todos esses contextos, surge a necessidade de percorrer as conexões entre vértices: verificar se há caminho entre dois pontos, medir distâncias, detectar ciclos, ou simplesmente visitar todos os vértices acessíveis a partir de uma origem.

Para resolver esses problemas, é necessário um método que percorra o grafo de maneira sistemática, garantindo que todos os vértices e arestas sejam eventualmente considerados.

12.1.1 Visitar e Processar Vértices

Durante a execução de uma busca, distinguimos duas ações fundamentais:

- **Visitar um vértice**: marcá-lo como descoberto, reconhecendo que ele existe e que será analisado.
- **Processar um vértice**: executar alguma operação sobre ele, como imprimir, acumular informações ou verificar suas adjacências.

Em geral, cada vértice é visitado exatamente uma vez e processado conforme as necessidades do algoritmo.

12.1.2 Percorrendo Grafos Sistematicamente

O percurso sistemático de um grafo significa escolher uma estratégia que defina a ordem em que os vértices serão visitados. As duas abordagens clássicas são:

- **Busca em Largura (BFS)**: explora primeiro os vértices mais próximos da origem, camada por camada.
- **Busca em Profundidade (DFS)**: segue um caminho o mais fundo possível antes de retroceder.

Ambas as estratégias produzem uma *árvore de busca* que descreve o caminho percorrido e podem ser adaptadas para diversas finalidades.

12.1.3 Contexto e Aplicações

Os algoritmos de busca constituem o ponto de partida para uma ampla classe de problemas e técnicas:

- Encontrar caminhos mínimos em grafos não ponderados.
- Determinar componentes conexas em grafos não direcionados.
- Detectar ciclos e identificar subestruturas críticas.
- Realizar ordenação topológica de grafos direcionados acíclicos (DAGs).

A partir dessas ideias básicas, podemos construir algoritmos mais sofisticados, como os de Dijkstra, Bellman-Ford, Kruskal e Prim, todos dependentes da noção de busca sistemática.

12.1.4 Estruturas de Apoio

Aqui reunimos apenas as estruturas auxiliares necessárias às buscas. As representações de grafos (matriz e lista de adjacência) já foram tratadas no capítulo anterior. Deste ponto em diante, supomos disponível um operador `adjacentes(u)` que devolve (ou itera) os vizinhos de u. Com essa suposição, nossa implementação interage com o grafo **via interface**, de modo funcional, como com um **TAD (Tipo Abstrato de Dados)**: os algoritmos de busca dependem apenas das operações expostas pelo TAD, e não dos detalhes de sua representação interna.

Fila (para Busca em Largura — BFS)

A **fila** segue a política *FIFO* (*first in, first out*): o primeiro elemento a entrar é o primeiro a sair. Na BFS, a fila preserva a exploração *por camadas* a partir de uma origem s: primeiro, visitam-se os vizinhos de s; depois, os vizinhos desses vizinhos, e assim por diante.

Operações típicas (tempo $O(1)$ amortizado):

- `enqueue(x)`: insere x no fim da fila;
- `dequeue()`: remove e retorna o elemento do início;
- `empty()`: indica se a fila está vazia.

Na prática, uma BFS mantém:

- um vetor booleano `visitado[v]` para evitar reprocessar vértices;
- (opcional) um vetor `dist[v]` com a *distância em arestas* a partir de s;
- (opcional) um vetor `pai[v]` para reconstruir o caminho.

Pilha (para Busca em Profundidade — DFS)

A **pilha** segue a política *LIFO* (*last in, first out*): o último a entrar é o primeiro a sair. Na DFS, a pilha garante a exploração *em profundidade*, seguindo um caminho até não haver mais vizinhos não visitados e então retrocedendo.

Há duas formas equivalentes de usar pilha na DFS:

1. **Recursiva** (pilha implícita): cada chamada recursiva empilha o vértice atual e desempilha no retorno.
2. **Iterativa** (pilha explícita): usamos uma estrutura `stack` com `push`/`pop`.

Operações típicas (tempo $O(1)$):

- `push(x)`: empilha x;
- `pop()`: desempilha e retorna o topo;
- `top()`: consulta o topo sem remover;
- `empty()`: indica se a pilha está vazia.

Assim como na BFS, a DFS usa um vetor `visitado[v]` para impedir revisitas. Em versões mais ricas, podemos usar *cores* (`branco/cinzento/preto`) para distinguir estados:

- **branco**: não descoberto;
- **cinzento**: descoberto, ainda em processamento (na pilha);
- **preto**: totalmente processado.

Essa convenção facilita a detecção de ciclos em dígrafos (arestas de retorno para vértices cinzentos).

12.1.5 Conjunto de Vértices Visitados (Marcação Booleana)

A marcação de visitados é essencial para garantir que cada vértice seja processado no máximo uma vez, evitando laços infinitos em presença de ciclos. As duas formas mais comuns são:

- **Vetor booleano** `visitado[0..|V|-1]`: acesso $O(1)$, baixo overhead;
- **Conjunto/hash set** `Visitados`: útil quando os identificadores não são inteiros densos, ou quando os vértices são objetos.

Padrão de uso. Ao descobrir um vizinho v de u, marcamos `visitado[v] = true` *no momento em que o enfileiramos/empilhamos*, não quando o removemos da fila/pilha. Isso evita múltiplas inserções do mesmo vértice na estrutura de controle.

Com essas três estruturas (fila, pilha e marcação de visitados), estamos prontos para formalizar os algoritmos de **Busca em Largura (BFS)** e **Busca em Profundidade (DFS)** nas próximas seções, com suas garantias e aplicações.

12.2 Busca em Largura (BFS)

A **Busca em Largura** explora o grafo em *camadas* a partir de uma origem s. Primeiro visitamos todos os vizinhos diretos de s (distância 1), depois os vizinhos desses vizinhos (distância 2), e assim sucessivamente. A analogia mais comum é a de *ondas de propagação*: uma frente de exploração avança uniformemente para fora do vértice inicial.

Como consequência, a BFS computa a **distância mínima em número de arestas** de s a qualquer vértice alcançável (em grafos não ponderados). Além disso, o conjunto das arestas usadas na descoberta forma uma *árvore de níveis* (ou floresta, se iniciarmos de múltiplas origens).

12.2.1 Implementação

A seguir apresentamos uma implementação direta da BFS em C, usando o TAD `Grafo` e uma fila simples baseada em vetor circular. Assumimos disponíveis as funções `grafo_num_vertices(G)` e `grafo_adjacentes(G, u, viz, max)`, que devolve os vizinhos de u em um vetor.

```c
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include "grafo_lista.h"   // ou grafo_matriz.h
#include "fila.h"

void bfs(Graph *G, int s) {
    int n = grafo_num_vertices(G);
    int *visitado = calloc(n, sizeof(int));
    int *dist = malloc(sizeof(int) * n);
    int *pai = malloc(sizeof(int) * n);

    for (int i = 0; i < n; ++i) {
        dist[i] = INT_MAX;
        pai[i] = -1;
    }

    Fila *Q = fila_criar(n);
    visitado[s] = 1;
    dist[s] = 0;
    pai[s] = s;
    fila_enfileirar(Q, s);

    while (!fila_vazia(Q)) {
        int u = fila_desenfileirar(Q);
        int viz[64]; // limite arbitrário
        int k = grafo_adjacentes(G, u, viz, 64);

        for (int i = 0; i < k; ++i) {
            int v = viz[i];
            if (!visitado[v]) {
```

```
32                  visitado[v] = 1;
33                  dist[v] = dist[u] + 1;
34                  pai[v] = u;
35                  fila_enfileirar(Q, v);
36              }
37          }
38      }
39
40      // Exemplo de saída
41      for (int v = 0; v < n; ++v) {
42          if (dist[v] < INT_MAX)
43              printf("dist[%d] = %d (pai=%d)\n", v, dist[v
                    ], pai[v]);
44          else
45              printf("dist[%d] = INF\n", v);
46      }
47
48      free(visitado);
49      free(dist);
50      free(pai);
51      fila_destruir(Q);
52  }
```

Programa 12.1: Busca em Largura (BFS)

Explicação da fila. A fila mantém a ordem *por camadas*:

1. Inicialmente, s é enfileirado com `dist[s] = 0`.
2. Ao desenfileirar s, examinamos seus vizinhos; todo vizinho ainda não visitado recebe `dist = 1`, `pai = s` e é enfileirado.
3. Em seguida, processamos os vértices com `dist = 1`; ao visitar seus vizinhos ainda não descobertos, eles passam a ter `dist = 2`, e assim por diante.

O ato de *marcar* `visitado[v] := verdadeiro` *no momento em que* v *é enfileirado* evita inserções duplicadas do mesmo vértice.

Reconstrução de caminhos. Após a BFS, se `visitado[t]` for verdadeiro, um caminho mínimo de s até t pode ser reconstruído recuando pelos ponteiros `pai`: $t, \texttt{pai}[t], \texttt{pai}[\texttt{pai}[t]], \ldots, s$ (e depois invertendo a ordem).

12.2.2 Complexidade

Com representação por **listas de adjacência**, cada vértice entra e sai da fila no máximo uma vez, e cada aresta é examinada no máximo uma vez. Assim, o tempo é:

$$T(|V|, |E|) = O(|V| + |E|).$$

O espaço auxiliar é $O(|V|)$ para `visitado`, `dist`, `pai` e a fila.

Em grafos **densos**, quando se usa **matriz de adjacência**, percorrer todos os vizinhos de um vértice custa $O(|V|)$, logo o custo total pode chegar a $O(|V|^2)$, mesmo que $|E|$ seja grande. Em contrapartida, a matriz permite testar adjacência em $O(1)$. Para buscas, listas de adjacência costumam ser preferíveis.

12.2.3 Aplicações

- **Distância mínima (não ponderado):** `dist[v]` fornece o menor número de arestas de s a v.
- **Conectividade:** iniciar BFS em s e verificar quais vértices ficaram visitados.
- **Bipartição (grafo bipartido):** colorir níveis alternadamente durante a BFS; um conflito de cores indica ímpar-ciclo.
- **Árvore de níveis:** as arestas `(pai[v], v)` formam uma árvore (ou floresta) que estratifica V por distância a s.

Observação prática (múltiplas origens). Para múltiplas fontes $S \subseteq V$, basta enfileirar todas as origens com `dist=0` no início. A BFS calculará, para cada vértice, a distância até a *origem mais próxima* em S.

12.3 Busca em Profundidade (DFS)

A **Busca em Profundidade** explora o grafo seguindo um caminho o mais longe possível antes de retroceder (*backtracking*). A analogia clássica é a de um *labirinto*: avançamos pelas bifurcações até não haver saída; então voltamos e tentamos outro caminho. Diferentemente da BFS (camadas), a DFS prioriza a *profundidade*.

12.3.1 Implementação

Versão recursiva. Usa a pilha implícita de chamadas. Mantemos `visitado`, `pai` e (opcionalmente) tempos de descoberta/fecho.

```c
#include <stdio.h>
#include <stdlib.h>
#include "grafo_lista.h"    // ou grafo_matriz.h
#include "pilha.h"

static void dfs_visita(Graph *G, int u, int *visitado,
   int *pai) {
    visitado[u] = 1;
    int viz[64];
    int k = grafo_adjacentes(G, u, viz, 64);
    for (int i = 0; i < k; ++i) {
        int v = viz[i];
        if (!visitado[v]) {
            pai[v] = u;
            dfs_visita(G, v, visitado, pai);
        }
    }
}

void dfs(Graph *G) {
    int n = grafo_num_vertices(G);
    int *visitado = calloc(n, sizeof(int));
    int *pai = malloc(sizeof(int) * n);
    for (int i = 0; i < n; ++i) pai[i] = -1;

    for (int s = 0; s < n; ++s)
        if (!visitado[s]) {
            pai[s] = s;
            dfs_visita(G, s, visitado, pai);
        }

    // Exemplo de saída
    for (int v = 0; v < n; ++v)
        printf("pai[%d] = %d\n", v, pai[v]);

    free(visitado);
    free(pai);
}
```

Programa 12.2: DFS recursiva genérica em C

Versão iterativa (pilha explícita). Equivale à recursiva, mas controla a pilha manualmente. Para simular o *estado* da iteração sobre os vizinhos de cada u, podemos guardar o índice do próximo vizinho a explorar.

```c
typedef struct { int u, idx; } Frame; // u = vértice atual; idx = próximo vizinho a explorar

void dfs_iterativa(Graph *G) {
    int n = grafo_num_vertices(G);
    int *visitado = calloc(n, sizeof(int));
    int *pai = malloc(sizeof(int) * n);

    for (int s = 0; s < n; ++s) if (!visitado[s]) {
        Pilha *P = pilha_criar(n);
        pai[s] = s;
        visitado[s] = 1;
        pilha_push(P, (Frame){s, 0});

        while (!pilha_vazia(P)) {
            Frame *fr = pilha_top(P);
            int u = fr->u;

            int viz[64], k = grafo_adjacentes(G, u, viz, 64);

            if (fr->idx >= k) { // acabou vizinhos de u
                pilha_pop(P);
                continue;
            }

            int v = viz[fr->idx++];
            if (!visitado[v]) {
                visitado[v] = 1;
                pai[v] = u;
                pilha_push(P, (Frame){v, 0});
            }
        }
        pilha_destruir(P);
    }

```

```
35      for (int v = 0; v < n; ++v)
36          printf("pai[%d] = %d\n", v, pai[v]);
37
38      free(visitado);
39      free(pai);
40  }
```

Programa 12.3: DFS iterativa com pilha explícita

Controle de visitados e cores. Além do booleano `visitado`, é comum usar *cores* para estados: `branco` (não descoberto), `cinzento` (descoberto/em processamento), `preto` (finalizado). Em dígrafos, encontrar aresta para um vértice `cinzento` indica *ciclo*.

12.3.2 Complexidade

Com listas de adjacência, cada vértice é processado uma vez e cada aresta é examinada no máximo uma vez. Portanto,

$$T(|V|, |E|) = O(|V| + |E|),$$

e o espaço auxiliar é $O(|V|)$ (para `visitado`, `pai` e pilha). Com matriz de adjacência, a DFS custa $O(|V|^2)$.

12.3.3 Aplicações

- **Detecção de ciclos:** em dígrafos, detectar arestas para vértices `cinzentos`; em grafos não direcionados, detectar arestas para vértices `visitados` que não sejam o `pai`.
- **Componentes conexas (não direcionado):** iniciar DFS de todos os vértices não visitados; cada árvore é uma componente.
- **Ordenação topológica (DAG):** empilhar o vértice ao terminar sua exploração; ao final, desempilhar em ordem fornece o topo.
- **Pontes e articulações (não direcionado):** usar tempos de descoberta $disc[v]$ e o menor alcance $low[v]$ (algoritmo de Tarjan): $low[v] = \min(disc[v], low[\text{filhos}], disc[\text{retornos}])$. Uma aresta (u, v) é *ponte* se $low[v] > disc[u]$; u é *articulação* por condições análogas.

Topos práticos. Para grafos grandes e profundos, prefira a versão iterativa para evitar `stack overflow`. Se os IDs de vértices não forem inteiros densos, substitua `visitado[]` por um `HashSet`.