FACOM32402 – Estrutura de Dados II

Prof. Renato Pimentel

2025/2

Sumário

1 Introdução a grafos **6**
1.1 Definições básicas 6
1.2 Representações 13
1.3 Um tipo abstrato de dados (TAD) para grafo 15
1.4 Exercícios 20

2 Algoritmos de busca em grafos **22**
2.1 Introdução 22
2.2 Busca em profundidade 23
2.3 Busca em largura 28
2.4 Busca pelo menor caminho 32
2.5 Exercícios 38

3 Árvore geradora mínima em grafos **40**
3.1 Introdução 40
3.2 Algoritmo de Prim 43
3.3 Algoritmo de Kruskal 47
3.4 Exercícios 53

4 Introdução a árvores **54**
4.1 Definições básicas 54
4.2 Árvores binárias 60
4.3 Representações 62
4.4 Um tipo abstrato de dados (TAD) para árvore binária 63
4.5 Exercícios 68

5 Árvore binária de busca **69**
5.1 Definição 69
5.2 Inserção na ABB 70
5.3 Busca na ABB 72
5.4 Remoção na ABB 74
5.5 Exercícios 77

6 Percursos: pré-ordem, em-ordem e pós-ordem **79**
6.1 Introdução . . . 79
6.2 Percurso pré-ordem . . . 80
6.3 Percurso em-ordem . . . 81
6.4 Percurso pós-ordem . . . 82
6.5 Exercícios . . . 83

7 Balanceamento de ABB – Árvores AVL **84**
7.1 Introdução . . . 84
7.2 Árvore AVL . . . 85
7.3 TAD árvore AVL . . . 86
7.4 Rotações . . . 88
7.4.1 Rotação LL (simples à direita) . . . 89
7.4.2 Rotação RR (simples à esquerda) . . . 90
7.4.3 Rotação LR (dupla à direita) . . . 92
7.4.4 Rotação RL (dupla à esquerda) . . . 95
7.4.5 Uso . . . 97
7.5 Inserção de nó na AVL . . . 97
7.6 Remoção de nó na AVL . . . 100
7.7 Exercícios . . . 104

8 Tabelas *hash* **105**
8.1 Definições básicas e aplicações . . . 105
8.2 Funções de *hashing* . . . 108
8.2.1 Método da divisão . . . 110
8.2.2 Método da multiplicação . . . 110
8.2.3 Método da dobra . . . 111
8.2.4 Cadeias de caracteres como chaves . . . 113
8.3 Um tipo abstrato de dados (TAD) para tabela *hash* . . . 113
8.4 Inserção e busca sem tratamento de colisões . . . 116
8.5 *Hashing* universal, perfeito e imperfeito . . . 118
8.6 Tratamento de colisões . . . 120
8.7 Exercícios . . . 128

9 Armazenamento secundário **130**
9.1 Conceitos básicos . . . 130
9.2 Índices ordenados . . . 131
9.3 Árvore B^+ . . . 135
9.4 Exercícios . . . 148

Apresentação

Objetivos e Ementa

Objetivos

- **Geral:** Capacitar ao aluno a aplicar algoritmos e estrutura de dados adequados à solução eficiente de problemas.

- **Específicos:** O aluno será capaz de compreender e implementar estruturas de dados envolvendo grafos, árvores e tabelas *hash*.

Ementa do curso

- Grafos;
- Árvores;
- Compressão de dados;
- Espalhamento (*hashing*);
- Estruturas de armazenamento secundário.

Bibliografia

Bibliografia básica

- AHO, A. V.; HOPCROFT, J. E.; ULLMAM, J. D. **Data structures and algorithms**. 3. ed. Menlo Park: Addison-Wesley, 1983.
- CORMEN, T. H.; et al. **Algoritmos: Teoria e Prática**. 3. ed. Rio de Janeiro: Campus, 2012.
  - 4. ed. (2023): https://integrada.minhabiblioteca.com.br/books/9788595159914
- SEDGEWICK, R.; WAYNE, K. **Algorithms**. 4. ed. Upper Saddle River: Addison-Wesley 2011.

Bibliografia complementar

- *BACKES, A. R.* ***Estrutura de Dados Descomplicada – em Linguagem C****. 1. ed. São Paulo: LTC, 2016.*
  - novo livro: https://integrada.minhabiblioteca.com.br/books/9788521638315

- CELES, W.; CERQUEIRA, R.; RANGEL, J. L. **Introdução a estruturas de dados**. 2. ed. Rio de Janeiro : Campus, 2004. KNUTH, D. E. **The art of computer programming** v. 1: fundamental algorithm. 3. ed. Reading : Addison Wesley, 1997.
- *SILBERSCHATZ, A.; KORTH, H. F.; SUDARSHAN, S.* ***Sistema de Banco de Dados****. 7. ed. Rio de Janeiro: GEN LTC, 2020*: https://integrada.minhabiblioteca.com.br/reader/books/9788595157552/
- SZWARCFITER, J. L.; MARKENZON, L. **Estrutura de dados e seus algoritmos**. 3. ed. Rio de Janeiro: LTC, 2010.
- TENENBAUM, A.M.; LANGSAM, Y.; AUGENSTEIN, M. J. **Estrutura de Dados Usando C**. São Paulo: Makron Books, 1995.
- ZIVIANI, N. **Projeto de Algoritmos:** com implementações em Pascal e C. 3. ed. São Paulo: Cengage Learning, 2011.

Conteúdo *previsto*

1. Grafos: definições básicas e representações; TAD; percursos em busca e profundidade; menor caminho; árvore geradora mínima.
2. Árvores: definições básicas e tipos; ABB e TAD; percursos; balanceamento; algoritmo de Huffman (compressão).
3. Tabelas *hash*: definição, aplicações; funções de *hashing*; tratamentos de colisões;
4. Armazenamento secundário: motivação, indexação; árvores B.

Avaliação: aproveitamento e frequência

- 2 provas teóricas individuais:
  - 04/12/2025 ($P1$) – 40 pontos
  - 10/03/2026 ($P2$) – 40 pontos
- Trabalho em grupo (a ser implementado em C). Entrega:
  - 19/02/2026 – 20 pontos
- Nota final (aproveitamento):

$$NF = P1 + P2 + T$$

- TDEs – trabalhos discentes efetivos: complementações de carga horária, a serem feitas ao longo do curso, na forma de listas de exercícios.

Observação importante
Atividades práticas completa ou parcialmente copiadas, receberão nota *zero*, *tanto de quem copiou quanto quem forneceu o material*. Mesmo nos casos em que o aluno assuma que copiou ou forneceu, a nota será mantida. Somente trabalhe por si ou com os integrantes de seu grupo – no caso do projeto – e não forneça seu material aos demais.

O mesmo se aplica a materiais copiados, mesmo que parcialmente, de outras fontes, como Internet, **ou gerados por meio de LLM**.

Avaliação substitutiva

- Alunos que não atingiram a nota 60 *(somente)* terão direito a uma prova substitutiva (SUB).
- Data: 17/03/2026
- O conteúdo da prova será o visto ao longo de todo o semestre.
- A prova vale 40 pontos, e substituirá a nota da prova de menor nota, *caso maior que a mesma*.
- Caso prevaleça a nota da SUB, a NF neste caso será dada por

$$NF = \begin{cases} 60, \text{ se } \max(P1, P2) + SUB + T \geq 60 \\ \max(P1, P2) + SUB + T, \text{ caso contrário,} \end{cases}$$

  ou seja, o aluno que ficou de *SUB terá nota máxima 60*, caso atinja na mesma a pontuação necessária para ser aprovado.

Frequência

- O aluno que tiver frequência inferior a 75% é reprovado por faltas.
- A assiduidade será computada através da chamada em sala durante as aulas, em um horário aleatório após 10 minutos do início de cada encontro diário. O professor poderá adotar, a seu critério, caso haja demasiada desistência de continuidade em sala, uma segunda chamada ao final do segundo horário de aula.
- *Falta em dia de prova*: o aluno somente terá direito a fazer prova em nova data caso apresente justificativa prevista pelas Normas de Graduação.
- É responsabilidade do aluno controlar sua frequência, de modo a evitar reprovação por falta.
- O professor não corrigirá atividades de alunos que já tenham sido reprovados por falta. Inclusive, os mesmos não terão direito à SUB (Normas da Graduação).

Logística

Aulas

- Terças-feiras: 19:00 até 20:40 – *Sala 5R-A 316*;
- Quintas-feiras: 20:50 até 22:30 – *Sala 5R-A 316*.

Comunicação e entrega das atividades práticas

- *Teams:* Equipe **FACOM32402 – 2025/2**, código de inscrição *bwamdgd*.

Atendimento e outras informações

- Professor: Renato Pimentel
  - Página: http://www.facom.ufu.br/∼rpimentel
  - E-mail: rpimentel @ ufu . br
  - Sala 1B139
- Atendimento (agendar previamente através de e-mail):
  - Quartas-feiras, 14:00 até 15:40, sala 1B139
  - Quintas-feiras, 19:00 até 20:40, sala 1B139
- Material da disciplina:
  - http://www.facom.ufu.br/∼rpimentel > Ensino > 2025/2 > FACOM32402 – Estrutura de Dados II

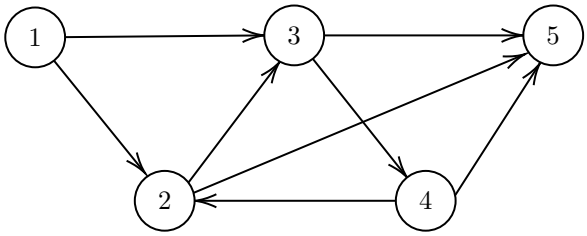
1 Introdução a grafos

1.1 Definições básicas

Grafo

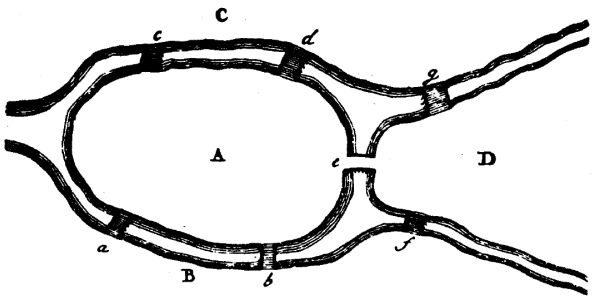
Um *grafo* ou *rede* é um modelo matemático que representa *relações* entre objetos – *nós* ou *vértices* – de um determinado conjunto. Tais relações, no modelo, são representadas por meio de *arestas* (também chamadas *arcos* ou *ramos*) que interconectam tais vértices.

- *Notação*: $G(V, A)$: V é o conjunto – não-vazio – de *vértices*; e A é o conjunto de *arestas*.
- *Exemplo:* $V = \{1, 2, 3, 4, 5\}$,
  $A = \{(1, 2), (1, 3), (2, 3), (2, 5), (3, 4), (3, 5), (4, 2), (4, 5)\}$



Exemplo clássico

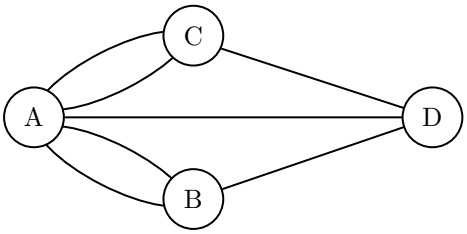
Pontes de Königsberg, Prússia
Visitar todas as quatro partes A, B, C e D da cidade (atual Kaliningrado, na Rússia), voltando ao ponto de partida e passando apenas *uma vez* por cada uma das sete pontes (a, b, . . . , g) sobre o Rio Prególia.



Fonte: Leonhard Euler - The World of Mathematics, Volume 1, Public Domain, https://commons.wikimedia.org/w/index.php?curid=33336521

- Leonhard Euler (séc. XVIII): não há solução possível.

Representação do problema como um grafo:

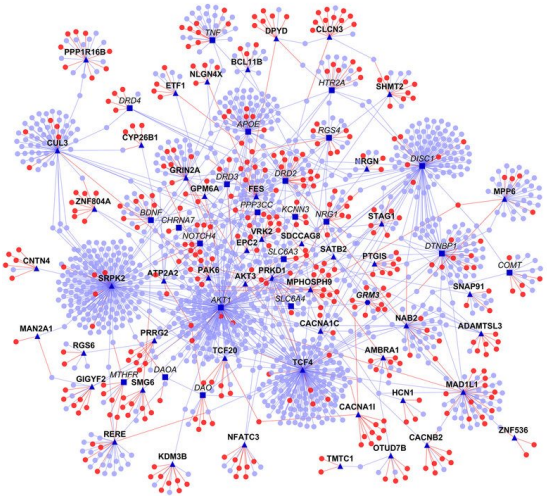


- Vértices são as partes de terra e arestas representam as pontes.
- A partir de qualquer vértice, não é possível encontrar um caminho que passe por todos os outros lugares sem que se passe mais de uma vez por pelo menos uma mesma aresta.

Exemplos de aplicações

1. *Redes sociais*: modelagem de redes como Facebook, Instagram, etc. (vértices são os perfis de usuários; arestas, as relações);
2. *Navegação*: modelagem de sistemas de navegação, como aplicações de GPS e Google Maps;
3. *Biologia*: pesquisas em redes neurais (reais), redes de proteínas, e afins;
4. *Comunicações*: telefonia e fibras óticas, redes de computadores, etc.

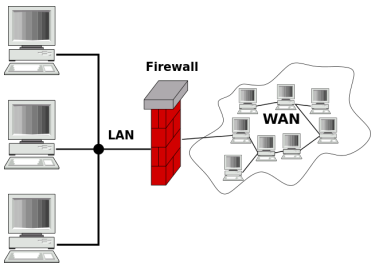
Jardim
Vila Sônia
Campo Limpo Paulista
Butantã
Francisco Morato
Baltazar Fidélis
Franco da Rocha
Caieiras
Perus
Vila Aurora
Jaraguá
Vila Clarice
Pirituba
Piqueri
Lapa
Água Branca
Palmeiras-Barra Funda
Júlio Prestes
Luz
Brás
Vila Galvão
Tucuruvi
Parada Inglesa
Jardim São Paulo-Ayrton Senna
Santana
Carandiru
Portuguesa-Tietê
Arménia
Tiradentes
Tabuão
Aeroporto-Guarulhos
Parada Rodoviária
Guarulhos-Cecap
Cecap
Engº Goulart
Vila Ipiranga
Tamanduateí
São Caetano do Sul
Utinga
Prefeito Saladino
Santo André
Capuava
Mauá
Guapituba
Ribeirão Pires
Rio Grande da Serra
Sônia Maria
Ipiranga
Sacomã
Alto do Ipiranga
Santos-Imigrantes
Chácara Klabin
Paraíso
Ana Rosa
Vila Mariana
Santa Cruz
Praça da Árvore
Saúde
São Judas
Conceição
Jabaquara
Diadema
Piraporinha
São Bernardo
Ferrazópolis
Eucaliptos
Campo Belo
Moema
AACD-Servidor
Hospital São Paulo
Santa Cecília
República
Mal. Deodoro
Higienópolis-Mackenzie
Anhangabaú
Consolação
Trianon-Masp
Brigadeira
Vergueiro
São Joaquim
Japão-Liberdade
Sé
Pedro II
São Bento
Juventude-Mooca
Belém
Brás
Tatuapé
Carrão
Penha
Vila Matilde
Guilhermina-Esperança
Artur Alvim
Cidade Tiradentes
Itaquera
Corinthians-Itaquera
Dom Pedro II
Ana Rosa
Paulista
Faria Lima
Fradique Coutinho
Oscar Freire
Clínicas
Sumaré
Vila Madalena
Pinheiros
Butantã
Hebraica-Rebouças
Cidade Jardim
Vila Olímpia
Berrini
Morumbi
Granja Julieta
João Dias
Santo Amaro
Socorro
Jurubatuba
Autódromo
Primavera-Interlagos
Grajaú
Mendes-Vila Natal
Campo Limpo
Capão Redondo
Vila das Belezas
Giovanni Gronchi
Vila Sônia
São Paulo-Morumbi
Pres. Altino
Osasco
Jd. Piratininga
Jd. Silveira
Jd. Mutinga
Jd. Rosa
Jd. Ângela
Jd. Herculano
Jd. Maria Duarte
Jd. Maria Helena
Jd. Ângela
Jd. Ângela



Fonte: Madhavicmu - Own work, CC BY-SA 4.0,
https://commons.wikimedia.org/w/index.php?curid=48447204

Conceitos

- *Vértice* é cada item do conjunto em estudo representado no grafo.
- A representação do vértice depende do problema estudado:
  - Um indivíduo (perfil) numa rede social;
  - Um lugar em um mapa;
  - Um computador ou servidor numa rede, etc.

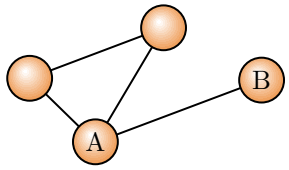


Fonte: Harald Mühlböck - selbst erstellt mithilfe von xfig und der xfig-libaries., CC BY-SA 3.0,
https://commons.wikimedia.org/w/index.php?curid=815692

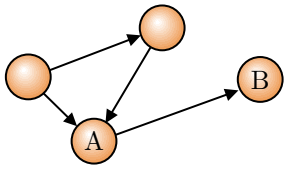
- Cada *aresta* representa a ligação (conexão) existente entre dois vértices.
  - Modela a *relação* entre ambos;
  - Dois vértices de um grafo são *adjacentes* – ou *vizinhos* – se existe uma aresta interligando-os:

    * Contatos ou seguidores numa rede social;
    * Uma interação biológica entre duas proteínas;
    * Uma ferrovia interligando duas estações em um mapa, etc.

- A aresta é *dirigida* ou *orientada* se a relação entre dois elementos ocorre somente numa direção.
  - Depende da aplicação sendo modelada.
- O grafo é um *grafo dirigido* (*dígrafo*) se todas as suas arestas forem dirigidas.
  - Num dígrafo, é possível haver uma conexão a partir de um vértice A para outro B, sem existir outra conexão no sentido oposto.

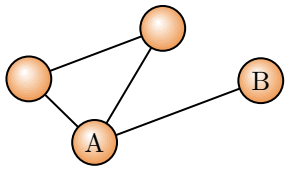


grafo

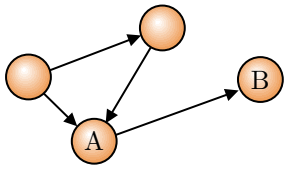


dígrafo

- *Grau*: o número de vizinhos que um vértice possui.
  - Quantidade de arestas que chegam ao vértice, ou partem do mesmo.
- No caso de um *dígrafo*, há dois tipos:
  - *Grau de entrada*: número de arestas que chegam ao vértice;
  - *Grau de saída*: número de arestas que partem do vértice;

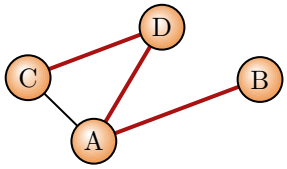


$G(A) = 3$
$G(B) = 1$

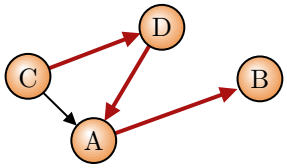


$G_E(A) = 2;\ G_S(A) = 1$
$G_E(B) = 1;\ G_S(B) = 0$

- *Caminho*: sequência de vértices do grafo, onde cada vértice está conectado ao seguinte por meio de uma aresta.
  - *Comprimento*: quantidade de vértices percorridos ao longo do caminho.

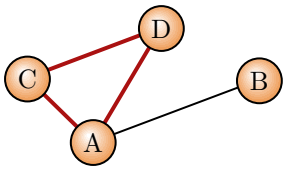


Caminhos: C-D-A-B ou B-A-D-C(ambos comprimento: 4)



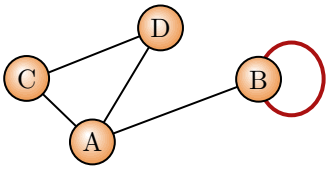
Caminho: C-D-A-B(comprimento: 4)

- O primeiro vértice – o *vértice inicial* – está *conectado* ao último vértice da sequência – o *vértice final* – se existe um caminho interligando-os.
  - Ex.: no dígrafo do exemplo acima, C é conectado a B.
- No exemplo, todos são *caminhos simples*, pois nenhum vértice aparece mais de uma vez nos mesmos.
- *Ciclo*: caminho fechado, em que o vértice inicial e o final são o mesmo.
  - O *comprimento do ciclo* é o número de vértices percorridos do inicial até o final, sem contar novamente este último.



Ciclo: C-D-A
(comprimento: 3)

- *Grafo acíclico*: grafo que não possui *ciclos simples* – nos quais cada vértice aparece uma única vez.
- *Laço*: aresta que conecta um vértice a si próprio, i.e. o vértice de origem e o de destino são o mesmo.
  - todo laço é um ciclo de comprimento 1.

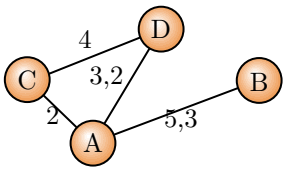


- *Multigrafo*: grafo que permite *arestas múltiplas* – quando pares de vértices adjacentes possuem mais de uma aresta os relacionando.

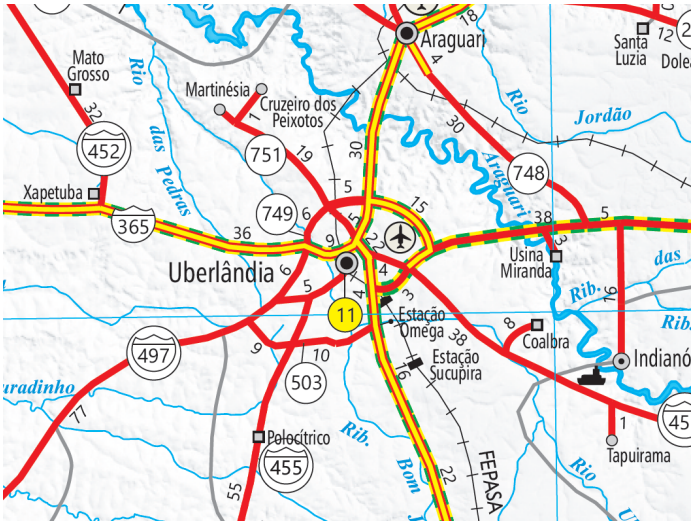
– Ex.: O grafo das pontes de Königsberg visto.

- Arestas podem conter um *peso* (valor numérico) associado, representando uma grandeza associada ao relacionamento dos vértices que interliga (ex.: a *distância* entre os mesmos).

- Um grafo é dito *ponderado* quando isto ocorre.



- Exemplo prático de grafo ponderado: um mapa rodoviário.

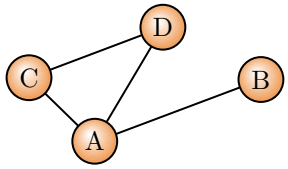


Fonte: DER-MG - https://www.der.mg.gov.br/files/1950/Mapa-Rodoviario/22325/Mapa-Rodoviario-2021.pdf

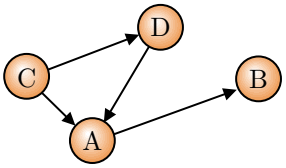
- O grafo é dito *conexo* se, para *quaisquer* dois vértices u e v, é possível encontrar um caminho de u para v *ou* vice-versa.

  – o grafo é *fortemente conexo* se para qualquer par u e v, há *ambos os caminhos* – de u para v e de v para u.

  Todo grafo não-direcionado conexo é, na verdade, *fortemente conexo*.



grafo conexo(fortemente conexo)

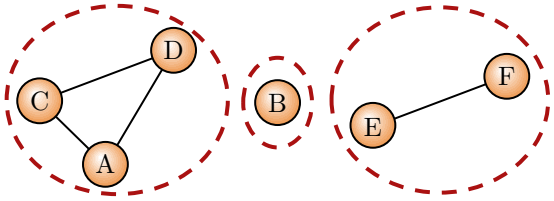


dígrafo fracamente conexo

- O dígrafo acima é *fracamente conexo*: por exemplo, dado o par A e D, existe um caminho de D até A, mas não de A até D.

- Se um grafo ou dígrafo não é conexo, então o mesmo é *desconexo*.

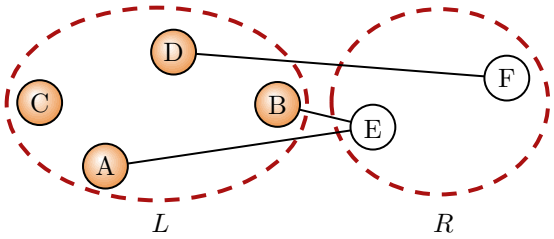
  – Cada subgrafo[1] de um grafo desconexo que é, por si, conexo, é chamado de *componente conexa*.



  – Há pelo menos duas componentes conexas num grafo desconexo.

- Um grafo $G(V, A)$ é *bipartido* se seu conjunto de vértices V puder ser dividido em dois subconjuntos *disjuntos* – sem intersecção – L e R tais que toda aresta em E somente conecte vértices de subconjuntos diferentes.

  – i.e. se $(u, v) \in E$, então necessariamente $u \in L$ e $v \in R$, ou vice-versa, onde $L \cup R = V$.



[1] $G_s(V_s, A_s)$ é *subgrafo* de um grafo $G(V, A)$ se $V_s \subseteq V$ e $A_s \subseteq A$

1.2 Representações

Representação computacional

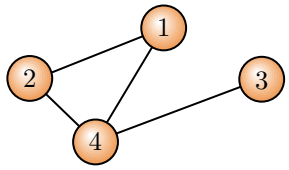
- Duas abordagens principais:
  - *Matriz de adjacências*
  - *Lista de adjacências*
- A melhor representação, dentre as duas, depende da aplicação. Não existe uma ideal para todos os casos.

Matriz de adjacências

- Um grafo $G(V, A)$ contendo n vértices, i.e. $|V| \equiv n$, é representado numa matriz $\mathbf{A} = \{a_{ij}\}$, quadrada $n \times n$, tal que:

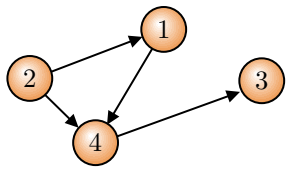
$$a_{ij} = \begin{cases} 1 & \text{se } (i,j) \in A\,; \\ 0 & \text{caso contrário.} \end{cases}$$

- $n \times n$: custo computacional é $\Theta(n^2)$ – independe do número de arestas $|A|$ do grafo.
  - Não indicada para grafos com n alto, mas poucas arestas (grafo pouco denso).
- Matriz de adjacências de um grafo (não-direcionado):



	1	2	3	4
1	0	1	0	1
2	1	0	0	1
3	0	0	0	1
4	1	1	1	0

  - A matriz é sempre *simétrica*.
- Matriz de adjacências de um dígrafo:

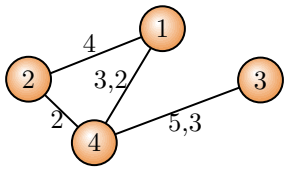


	1	2	3	4
1	0	0	0	1
2	1	0	0	1
3	0	0	0	0
4	0	0	1	0

- A simetria ocorre somente se para cada aresta $(i, j) \in A$ houver outra correspondente $(j, i) \in A$, para todo i, $j \in V$.

- Se o grafo ou dígrafo é ponderado: representa-se o *peso* de cada aresta diretamente na matriz $\mathbf{A}$, i.e.:

$$a_{ij} = \begin{cases} \omega_{ij} & \text{se } (i,j) \in A \text{ e tem peso } \omega_{ij}\,; \\ 0 & \text{caso contrário.} \end{cases}$$



	1	2	3	4
1	0	4	0	3,2
2	4	0	0	2
3	0	0	0	5,3
4	3,2	2	5,3	0

Lista de adjacências

- Um grafo $G(V, A)$ contendo n vértices e m arestas, i.e. $|V| \equiv n$ e $|A| \equiv m$, é representado por um *array* (vetor) Adj com n listas, uma para cada vértice de V.

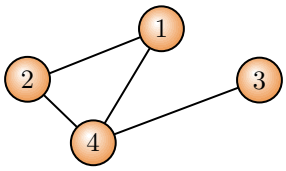
- Para cada vértice $u \in V$, a lista $Adj[u]$ contém todos os vértices adjacentes a u em G (representados por *índices* ou *ponteiros*).

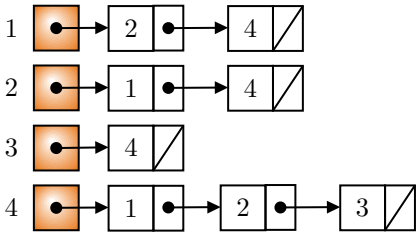
  - Ou seja, para cada vértice $u \in V$, $Adj[u]$ contém todos os vértices v tais que a aresta (u, v) pertença a A.

- O custo computacional é $\Theta(n + m)$ – em geral menor que o custo no caso de matriz de adjacências.

  - Indicada para grafos com n alto, mas poucas arestas.

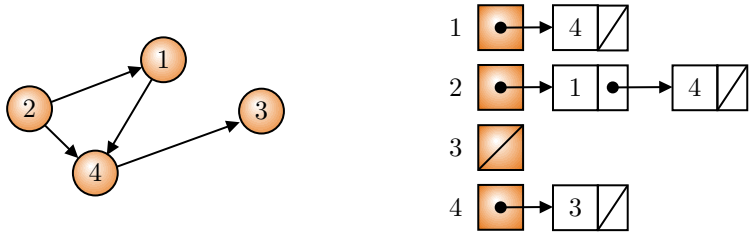
- Lista de adjacências de um grafo (não-direcionado):





- Cada aresta $(u, v) \in A$ é representada duas vezes:
  uma na lista de adjacências do vértice u, e outra na lista do vértice v.
- A soma dos comprimentos das listas de adjacências dos vértices corresponde a $2m \equiv 2|A|$.

- Lista de adjacências de um dígrafo:



- Cada aresta $(u, v) \in A$ é representada uma única vez.
- A soma dos comprimentos das listas de adjacências dos vértices corresponde a $m \equiv |A|$.

1.3 Um tipo abstrato de dados (TAD) para grafo

- Consideramos um tipo abstrato de dados *opaco* em Linguagem C para representação por *lista de adjacências*.

  Tipo opaco

  - A ED será referenciada mediante um *ponteiro* para a mesma, ao invés de uma variável do tipo `Grafo` diretamente.
  - É como ocorre, por exemplo, com o tipo `FILE` em C.
  - Isto permite ocultar a implementação, permitindo sua alteração sem interferir no funcionamento de qualquer código que o utilize – desde que a *interface* seja respeitada.

- Modularização do TAD:
  - *grafo.h*: a *interface* em si: definição do tipo opaco; funções disponíveis para manipular o grafo (protótipos).
  - *grafo.c*: o que deve ficar oculto do usuário, além da implementação das funções definidas pela interface. Inclui a definição do *tipo grafo*.

- *Tipo grafo*: estrutura que contém:
  - `eh_ponderado`:se as arestas têm peso ou não (i.e. grafo ponderado?);

- `nro_vert`: número de vértices que o grafo – tamanho do conjunto V;
- `grau_max`: número máximo de arestas a qual um vértice poderá se conectar;
- `arestas`: ponteiro onde é alocada a matriz de arestas do grafo;
- `pesos`: ponteiro onde é alocada a matriz de pesos das arestas do grafo – caso o mesmo seja ponderado;
- `grau`: ponteiro onde é alocado um vetor que armazena o número de arestas já associadas a um vértice.

Interface: grafo.h

grafo.h

```
typedef struct grafo Grafo;
Grafo* cria_Grafo(int nro_vertices, int grau_max, int
    eh_ponderado);
void libera_Grafo(Grafo* gr);
int insereAresta(Grafo* gr, int orig, int dest, int
    eh_digrafo, float peso);
int removeAresta(Grafo* gr, int orig, int dest, int
    eh_digrafo);
```

Implementação do TAD opaco Grafo: grafo.c

grafo.c: bibliotecas usadas e definição do tipo

```
#include <stdio.h>
#include <stdlib.h>
#include "Grafo.h" //inclui os protótipos

//Definição do tipo Grafo
struct grafo{
  int eh_ponderado;
  int nro_vertices;
  int grau_max;  // tamanho das listas
  int** arestas; // array de arestas
  float** pesos;
  int* grau; // quantidade de elementos em cada lista
};
```

Criando um grafo

Exemplo de uso

```
Grafo *gr;
gr = cria_Grafo(10, 7, 0);
```

grafo.c: criação do grafo

```c
Grafo* cria_Grafo(int nro_vertices, int grau_max, int eh_ponderado){
  Grafo *gr;
  gr = (Grafo*) malloc(sizeof(struct grafo));
  if(gr != NULL){
    int i;
    gr->nro_vertices = nro_vertices;
    gr->grau_max = grau_max;
    gr->eh_ponderado = (eh_ponderado != 0)?1:0;
    gr->grau = (int*) calloc(nro_vertices,sizeof(int));

    gr->arestas = (int**) malloc(nro_vertices * sizeof(int*));
    for(i=0; i<nro_vertices; i++)
      gr->arestas[i] = (int*) malloc(grau_max * sizeof(int));

    if(gr->eh_ponderado){
      gr->pesos = (float**) malloc(nro_vertices * sizeof(float*));
      for(i=0; i<nro_vertices; i++)
        gr->pesos[i] = (float*) malloc(grau_max * sizeof(float));
    }
  }
  return gr;
}
```

`gr = cria_Grafo(10, 7, 0);`

- Cria um grafo com 10 vértices;
- Cada um pode se conectar com até outros 7;
- Matriz 10×7 para as arestas;
- Vetor `grau` indica o número de conexões para cada vértice

arestas

	0	1	2	3	4	5	6
0							
1							
2							
3							
4							
5							
6							
7							
8							
9							

grau

0	0
1	0
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0

Liberando o grafo da memória

Exemplo de uso

```c
Grafo *gr;
gr = cria_Grafo(10, 7, 0);
.
```

```
.
libera_Grafo(gr);
```

grafo.c: liberando o grafo

```
void libera_Grafo(Grafo* gr){
  if(gr != NULL){
    int i;

    // libera a matriz de arestas
    for(i=0; i<gr->nro_vertices; i++)
      free(gr->arestas[i]);
    free(gr->arestas);

    // libera a matriz de pesos
    if(gr->eh_ponderado){
      for(i=0; i<gr->nro_vertices; i++)
        free(gr->pesos[i]);
      free(gr->pesos);
    }
    free(gr->grau);
    free(gr);
  }
}
```

Inserindo uma aresta

Exemplo de uso

```
Grafo *gr;
gr = cria_Grafo(10, 7, 0);
insereAresta(gr, 0, 1, 0, 0);
insereAresta(gr, 1, 3, 0, 0);
```

grafo.c: inserindo uma aresta no grafo

```
int insereAresta(Grafo* gr, int orig, int dest, int
    eh_digrafo, float peso){
  if(gr == NULL)
    return 0;

  // verifica a existência dos vértices de origem e destino
  if(orig < 0 || orig >= gr->nro_vertices)
    return 0;
  if(dest < 0 || dest >= gr->nro_vertices)
    return 0;

  // insere aresta ao final da linha
```

```
  gr->arestas[orig][gr->grau[orig]] = dest;
  if(gr->eh_ponderado)
    gr->pesos[orig][gr->grau[orig]] = peso;
  gr->grau[orig]++;

  // insere a outra aresta se não for dígrafo
  if (eh_digrafo == 0)
    insereAresta(gr,dest,orig,1,peso);
  return 1;
}
```

`insereAresta(gr, 0, 1, 0, 0);`

arestas

	0	1	2	3	4	5	6
0	1						
1	0						
2							
3							
4							
5							
6							
7							
8							
9							

grau

0	1
1	1
2	0
3	0
4	0
5	0
6	0
7	0
8	0
9	0

`insereAresta(gr, 1, 3, 0, 0);`

arestas

	0	1	2	3	4	5	6
0	1						
1	0	3					
2							
3	1						
4							
5							
6							
7							
8							
9							

grau

0	1
1	2
2	0
3	1
4	0
5	0
6	0
7	0
8	0
9	0

Removendo uma aresta

Exemplo de uso

```
Grafo *gr;
gr = cria_Grafo(10, 7, 0);
insereAresta(gr, 0, 1, 0, 0);
insereAresta(gr, 1, 3, 0, 0);
.
.
removeAresta(gr, 0, 1, 0, 0);
```

- Caso o ponteiro para grafo não seja nulo, é preciso *buscar* pela aresta a ser removida.
- A implementação não é encadeada: é necessário, nas listas de adjacência dos vértices envolvidos, ocupar as respectivas posições sendo disponibilizadas pela exclusão.
- Como `insereAresta()`, retorna 1 se exclusão for bem-sucedida, e 0 caso contrário.

grafo.c: removendo uma aresta do grafo

```c
int removeAresta(Grafo* gr, int orig, int dest, int
    eh_digrafo){
  if(gr == NULL)
    return 0;
  if(orig < 0 || orig >= gr->nro_vertices)
    return 0;
  if(dest < 0 || dest >= gr->nro_vertices)
    return 0;

  // procura pela aresta
  int i = 0;
  while(i < gr->grau[orig] && gr->arestas[orig][i] != dest)
    i++;
  if(i == gr->grau[orig]) // elemento não encontrado
    return 0;
  gr->grau[orig]--;
  gr->arestas[orig][i] = gr->arestas[orig][gr->grau[orig]];
  if(gr->eh_ponderado)
    gr->pesos[orig][i] = gr->pesos[orig][gr->grau[orig]];
  if(eh_digrafo == 0)
    removeAresta(gr,dest,orig,1);
  return 1;
}
```

```c
insereAresta(gr, 0, 1, 0, 0);
insereAresta(gr, 1, 3, 0, 0);
```

arestas

	0	1	2	3	4	5	6
0	1						
1	0	3					
2							
3	1						
4							
5							
6							
7							
8							
9							

grau

0	1
1	2
2	0
3	1
4	0
5	0
6	0
7	0
8	0
9	0

```c
removeAresta(gr, 0, 1, 0, 0);
```

arestas

	0	1	2	3	4	5	6
0	~~1~~						
1	3 ←						
2							
3	1						
4							
5							
6							
7							
8							
9							

grau

0	0
1	1
2	0
3	1
4	0
5	0
6	0
7	0
8	0
9	0

1.4 Exercícios

1. Qual é o custo computacional da exclusão (ou mesmo a consulta) de uma aresta no código visto?

2. Este custo seria o mesmo para a implementação por matriz de adjacência? Justifique.

3. A partir da implementação vista, adapte o código para a versão com lista encadeada.

Referências

1. BACKES, A. R. **Algoritmos e Estruturas de Dados em Linguagem C**. 1. ed. Rio de Janeiro: LTC, 2023. Disponível em: https://www.sistemas.ufu.br/biblioteca-gateway/minhabiblioteca/9788521638315.

2. CORMEN, T. H.; et al. **Algoritmos**. 4. ed. Rio de Janeiro: GEN LTC, 2024. Disponível em: https://www.sistemas.ufu.br/biblioteca-gateway/minhabiblioteca/9788595159914.

Os materiais de parte desta seção foram gentilmente cedidos por André R. Backes (UFSCar); Bruno Travençolo e Christiane Brasil (FACOM/UFU).

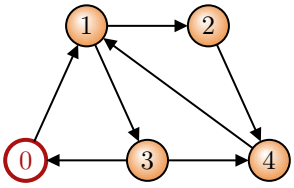
TAD grafo: disponível em: https://www.facom.ufu.br/~backes/wordpress/ProjGrafo.zip. Acesso em: 22/jan/2025.

Adaptações: Renato Pimentel, FACOM/UFU

2 Algoritmos de busca em grafos

2.1 Introdução

- O objetivo das *buscas* em grafos é explorar os mesmos, de uma maneira bem determinada.
  - I.e. correspondem a processos sistemáticos de como *caminhar* por seus vértices e arestas.
- As três principais formas de busca:
  1. Busca em profundidade
  2. Busca em largura
  3. Busca pelo menor caminho
- De modo geral, as operações de busca dependem do *vértice inicial*.
  - O ponto de partida é um aspecto bastante importante da própria busca.
  - Por exemplo, em uma busca pelo menor caminho, temos que saber qual é o ponto de partida desse caminho.

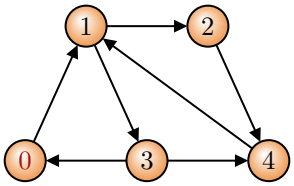


Fonte: adaptado de BACKES, A. R. Algoritmos e Estruturas de Dados em Linguagem C. https://www.sistemas.ufu.br/biblioteca-gateway/minhabiblioteca/9788521638315

- A busca pode precisar visitar todos ou apenas um subconjunto dos vértices.
- Vários problemas em grafos podem ser resolvidos efetuando uma busca.
  - Encontrar o caminho entre dois vértices conectados do grafo;
  - Determinar o menor caminho entre dois vértices do grafo;
  - Resolução de quebra-cabeças (labirintos);
  - Determinar componentes conexas (fortemente ou não), etc.

Grafo para teste

A fim de demonstrar os algoritmos de busca, utilizamos o seguinte dígrafo:



A criação do mesmo usa a seguinte implementação, considerando-se um máximo de 5 conexões por vértice:

main.c – grafo de teste

```
#include <stdio.h>
#include <stdlib.h>
#include "Grafo.h"
int main(){
  int eh_digrafo = 1;
  Grafo* gr = cria_Grafo(5, 5, 0);

  insereAresta(gr, 0, 1, eh_digrafo, 0);
  insereAresta(gr, 1, 3, eh_digrafo, 0);
  insereAresta(gr, 1, 2, eh_digrafo, 0);
  insereAresta(gr, 2, 4, eh_digrafo, 0);
  insereAresta(gr, 3, 0, eh_digrafo, 0);
  insereAresta(gr, 3, 4, eh_digrafo, 0);
  insereAresta(gr, 4, 1, eh_digrafo, 0);

  // realizar a busca aqui

  libera_Grafo(gr);
  system("pause");
  return 0;
}
```

2.2 Busca em profundidade

- O funcionamento da busca em profundidade pode ser assim descrito:

  Partindo de um vértice inicial, a busca explora o máximo possível cada um dos vizinhos de um vértice antes de retroceder (*backtracking*).

- Em inglês: *depth first search* (*DFS*).

- Esse tipo de busca se inicia em um vértice e se aprofunda nos vértices vizinhos deste até encontrar um dos dois casos:

  - o alvo da busca seja encontrado; ou
  - um vértice sem vizinhos ainda não visitados.

- Retrocesso ou *backtracking*:
  - O grafo é percorrido de maneira sistemática até que a busca falhe, ou se encontre um vértice sem vizinhos.
    * Nesse momento, entra em funcionamento o mecanismo de *backtracking*: a busca retorna pelo mesmo caminho percorrido, visando encontrar um caminho alternativo.

- A função que inicializa a busca – `buscaProfundidade_Grafo()` – recebe três parâmetros:
  1. o grafo
  2. o vértice inicial da busca
  3. um *array* que marcará a ordem que cada vértice será visitado – correspondendo ao *resultado* da busca.

- Uma segunda função, `buscaProfundidade()`, recebe, além dos mesmos 3 parâmetros acima, um *contador de visitação*. Trabalha de forma recursiva.
  - A recursão termina quando não houver mais vizinhos a serem visitados.

main.c – busca em profundidade

```
  .
  .
  insereAresta(gr, 4, 1, eh_digrafo, 0);

  int vis[5];
  buscaProfundidade_Grafo(gr, 0, vis);
  .
```

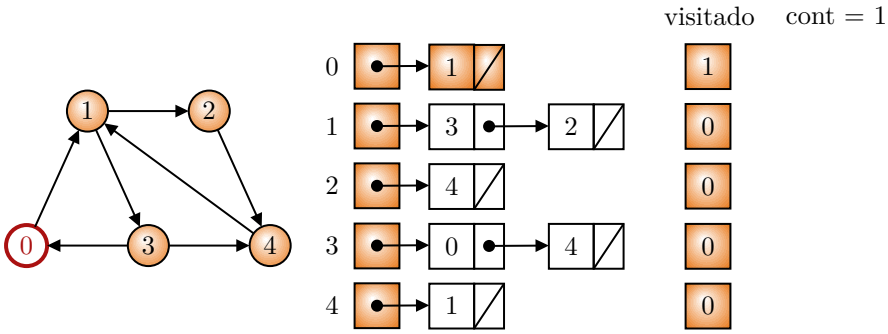
grafo.c – busca em profundidade

```
void buscaProfundidade(Grafo *gr, int ini, int *visitado, int *cont){
  int i;

  // marca o vértice como visitado; visita os vizinhos ainda não visitados.
  visitado[ini] = *cont;
  (*cont)++;
  for(i=0; i<gr->grau[ini]; i++){
    if(!visitado[gr->arestas[ini][i]])
      buscaProfundidade(gr,gr->arestas[ini][i],visitado,cont);
  }
}
```

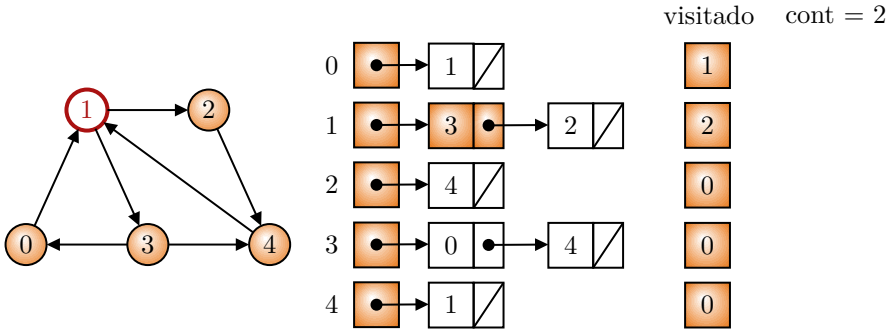
grafo.c – busca em profundidade (inicial, chamada pelo programa principal)

```c
void buscaProfundidade_Grafo(Grafo *gr, int ini, int *
    visitado){
  int i, cont = 1;

  // marca os vértices como não-visitados.
  for(i=0; i<gr->nro_vertices; i++)
    visitado[i] = 0;
  buscaProfundidade(gr,ini,visitado,&cont);
}
```

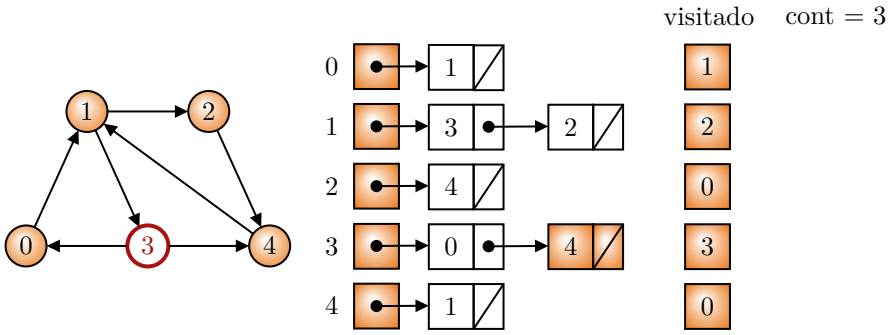
Passo a passo



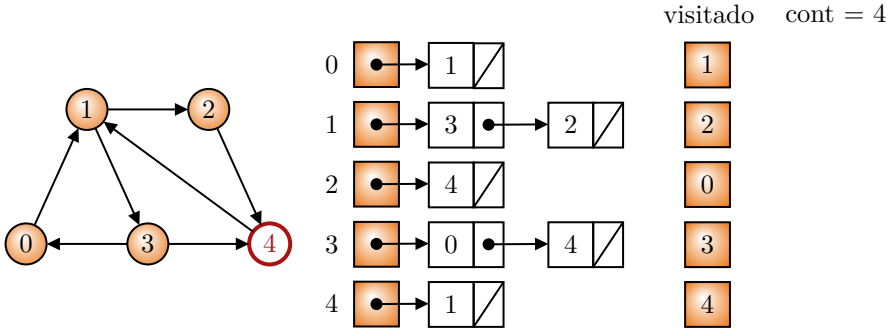
Inicia a busca com o vértice 0. Marca o vértice 0 como visitado e executa a busca para o vértice adjacente (1)



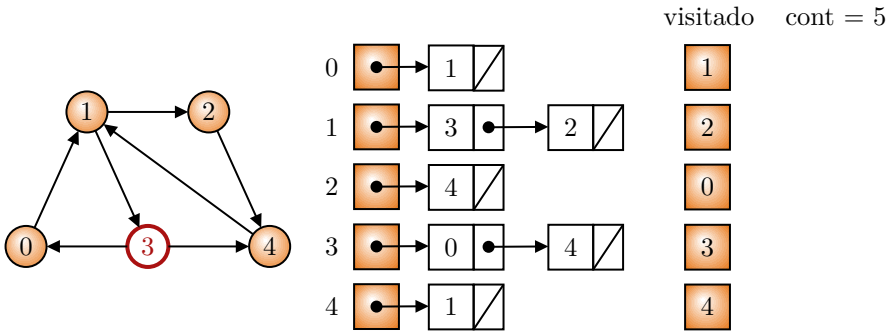
Marca o vértice 1 como visitado e
executa a busca para o primeiro vértice adjacente (3)



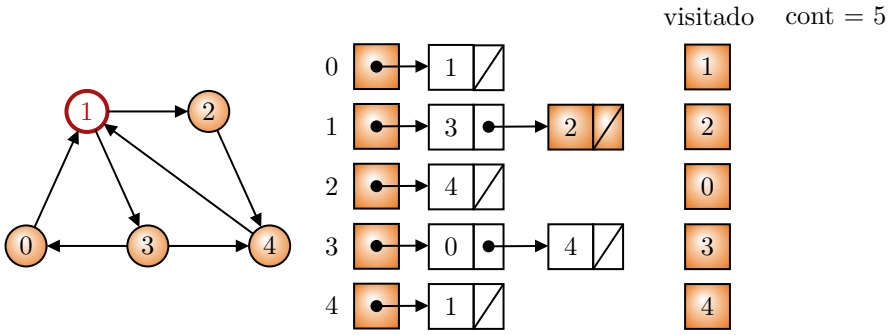
Marca o vértice 3 como visitado e
executa a busca para o primeiro vértice adjacente não visitado (4)



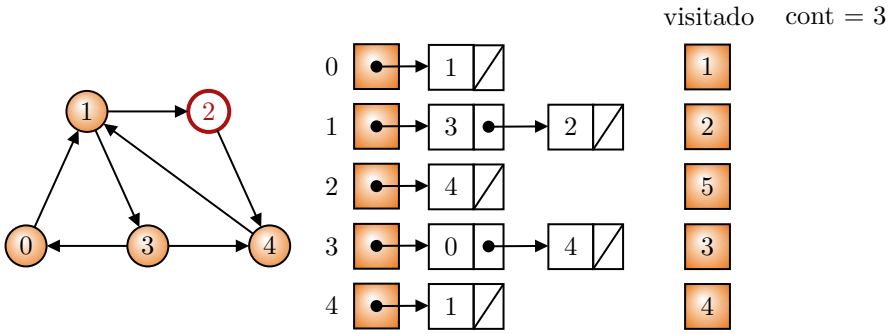
Marca o vértice 4 como visitado.
Todos os vértices adjacentes já foram visitados. Volta para o vértice 3



Todos os vértices adjacentes ao vértice 3 já foram visitados.
Volta para o vértice 1



Executa a busca para o segundo vértice adjacente (2).



Marca o vértice 2 como visitado. A partir desse ponto o algoritmo apenas volta na recursão (todos os vértices já foram visitados) e finaliza a busca.

Complexidade

- Considerando um grafo $G(V, A)$, onde $|V|$ é o número de vértices e $|A|$ é o número de arestas, a complexidade no pior caso é:
  - Custo de ir para cada vértice: $O(|V|)$;
  - Custo de transitar em cada aresta: $O(|A|)$;
  - Complexidade da busca no pior caso: $O(|V| + |A|)$.

Aplicações

- Encontrar componentes conexas e fortemente conexas;
- Ordenação topológica de um grafo;
- Procurar a saída de um labirinto;

- Verificar se um grafo é completamente conexo:
  - por exemplo, se uma rede de computadores está funcionando corretamente ou não;
- implementar a *ferramenta de preenchimento de cor* em editores de imagens.

2.3 Busca em largura

- O funcionamento da busca em largura pode ser assim descrito:

  Partindo de um vértice inicial, a busca explora todos os vizinhos de um vértice. Em seguida, para cada vizinho, repete-se esse processo, visitando os vértices ainda inexplorados.
- Em inglês: *breadth first search* (*BFS*).
- Esse tipo de busca se inicia em um vértice e então visita todos os seus vizinhos antes de se aprofundar na busca. Esse processo continua até que:
  - o alvo da busca seja encontrado; ou
  - não existam mais vértices a serem visitados.
- O algoritmo usa o conceito de *fila*:
  - O grafo é percorrido de maneira sistemática, primeiro marcando como "visitados" todos os vizinhos de um vértice e em seguida começa a visitar os vizinhos de cada vértice na ordem em que eles foram marcados.
  - Para realizar essa tarefa, uma fila é utilizada para administrar a visitação dos vértices
    - O primeiro vértice marcado (ou marcado a mais tempo) é o primeiro a ser visitado.
- A função que realiza a busca recebe três parâmetros:
  1. o grafo
  2. o vértice inicial da busca
  3. um *array* que marcará a ordem que cada vértice será visitado – correspondendo ao *resultado* da busca.

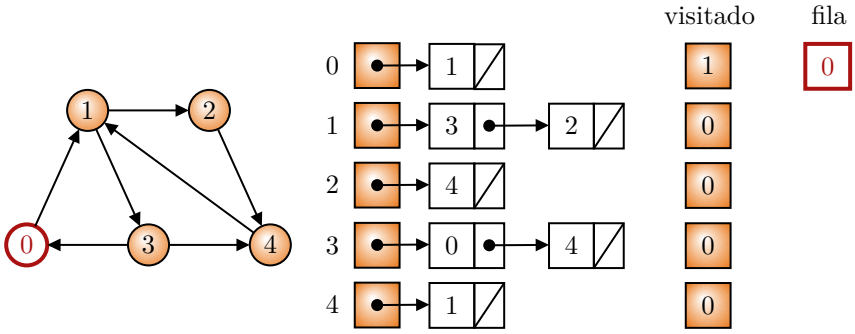
main.c – busca em largura

```
  .
  .
  insereAresta(gr, 4, 1, eh_digrafo, 0);

  int vis[5];
  buscaLargura_Grafo(gr, 0, vis);
  .
  .
```

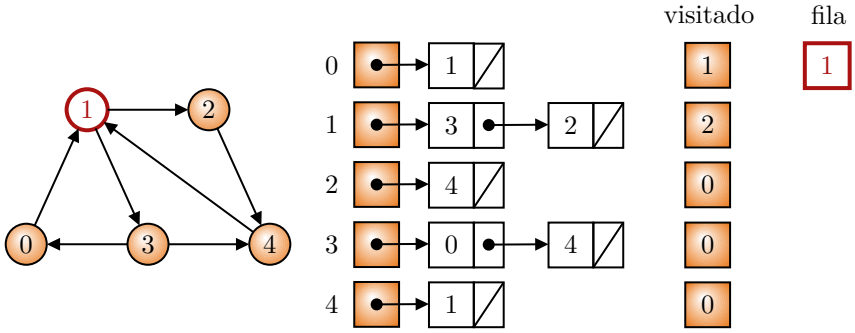
grafo.c – busca em largura

```
void buscaLargura_Grafo(Grafo *gr, int ini, int *visitado){
  int i, vert, NV, cont = 1;
  int *fila, IF = 0, FF = 0; // IF/FF: início/final da fila

  // marca os vértices como não-visitados
  for(i=0; i<gr->nro_vertices; i++)
    visitado[i] = 0;

  // Cria a fila; visita e insere "ini" na mesma.
  NV = gr->nro_vertices;
  fila = (int*) malloc(NV * sizeof(int));
  FF++;
  fila[FF] = ini;
  visitado[ini] = cont;
  while(IF != FF){
    // toma o primeiro da fila
    IF = (IF + 1) % NV;
    vert = fila[IF];

    //cont++;

    // visita os vizinhos ainda não visitados, colocando-os
    na fila
    for(i=0; i<gr->grau[vert]; i++){
      if(!visitado[gr->arestas[vert][i]]){
        FF = (FF + 1) % NV;
        fila[FF] = gr->arestas[vert][i];
        visitado[gr->arestas[vert][i]] = ++cont;
      }
    }
  }
  free(fila);
}
```

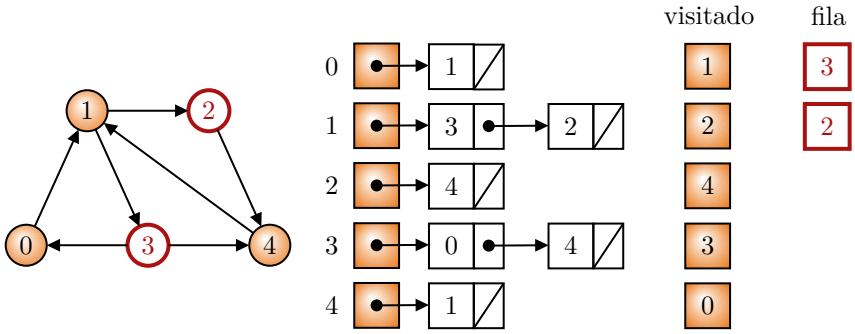
Passo a passo



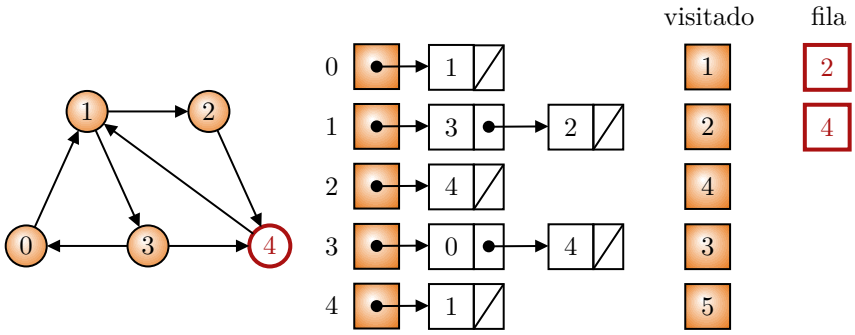
Inicializa a busca em largura.
Visita e insere na fila o vértice 0.



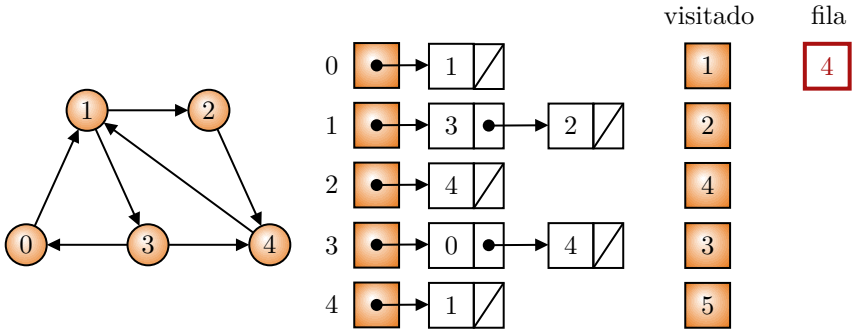
Remove vértice 0 da fila.
Insere na fila o vértice adjacente (1) que não foi visitado.



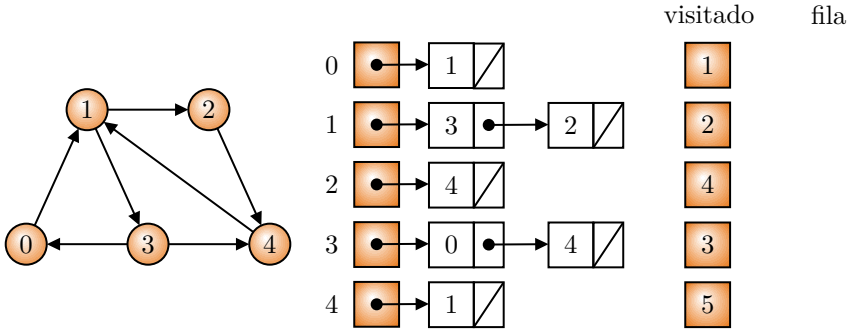
Remove vértice 1 da fila.
Insere na fila os vértices adjacentes (3 e 2) que não foram visitados.



Remove vértice 3 da fila.
Insere na fila o vértice adjacente (4) que não foi visitado.



Remove vértice 2 da fila.
Vértices adjacentes já foram visitados.



Remove vértice 4 da fila. Vértices adjacentes já foram visitados.
Fila vazia: fim da busca!

Complexidade

- Considerando um grafo $G(V, A)$, onde $|V|$ é o número de vértices e $|A|$ é o número de arestas, a complexidade no pior caso é:

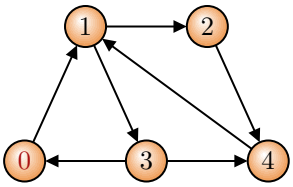
- Custo de inserção e remoção em fila: constante;
- Custo de enfileirar e remover todos os vértices uma vez: $O(|V|)$;
- Custo de utilizar todas as arestas: $O(|A|)$;
- Complexidade da busca no pior caso: $O(|V| + |A|)$.

Aplicações

- Encontrar todos os vértices conectados a apenas uma componente conexa;
- Encontrar o *menor caminho* entre dois vértices;
- Testar se um grafo é bipartido;
- Roteamento: encontrar um número mínimo de *hops* em uma rede;
  - Os *hops* são os vértices intermediários no caminho correspondente à conexão.
- Encontrar o número mínimo de intermediários entre 2 pessoas.

2.4 Busca pelo menor caminho

- Relembrando:
  - Dado um grafo $G(V, A)$, um vértice $u \in V$ está *conectado* a outro vértice, $v \in V$ se existe um caminho de u para v.
- Quando u e v são adjacentes (vizinhos), o menor caminho de u para v é a aresta que os conecta.
- No entanto, é muito comum não existir uma aresta conectando u e v diretamente.
  - Por exemplo, os vértices 0 e 4 não são adjacentes.



- Quando u e v não são adjacentes, mas conectados, é possível representar o(s) *caminho(s)* de u para v em termos das arestas que compõem o(s) mesmo(s);

- No dígrafo sendo usado, um possível caminho de 1 para 4 é dado por $(0, 1)$, $(1, 2)$, $(2, 4)$.
  Outra possibilidade: $(0, 1)$, $(1, 3)$, $(3, 4)$.

Menor caminho

O *menor caminho* (ou *caminho geodésico*) entre dois vértices é o que representa o *menor comprimento*[2] dentre todos os caminhos possíveis entre ambos.

- Ambos os caminhos acima têm comprimento 3.
- em *grafos ponderados*, o comprimento do caminho corresponde à *soma dos pesos* das arestas que o define.

Algoritmo de Dijkstra

- Uma das maneiras de achar o menor caminho é utilizando o *algoritmo de Dijkstra*:
  - Talvez o mais conhecido algoritmo;
  - Trabalha com grafos e dígrafos, ponderados ou não.
  - No caso de um grafo ponderado, as arestas *não* podem ter pesos negativos;
- Como funciona o algoritmo de Dijkstra:

  Partindo de um vértice inicial, o algoritmo de Dijkstra calcula a menor distância deste vértice a todos os demais a ele conectados.

- A função que inicializa a busca – `menorCaminho_Grafo()` – recebe quatro parâmetros:
  1. o grafo
  2. o vértice inicial
  3. um *array*, `ant`, para armazenar o antecessor dos vértices no caminho
  4. outro *array*, `dist`, para a distância a partir do vértice inicial.
- Outra função, `procuraMenorDistancia()`, recebe três parâmetros:
  - o *array* `dist`
  - um segundo *array*, para representar os vértices visitados; e
  - o número de vértices do grafo.

[2] Neste caso, definido em termos da quantidade de arestas do caminho.

main.c – busca pelo menor caminho

```
  .
  .
  insereAresta(gr, 4, 1, eh_digrafo, 0);

  int ant[5];
  float dist[5];
  menorCaminho_Grafo(gr,0, ant, dist);
  .
```

grafo.c – menor caminho

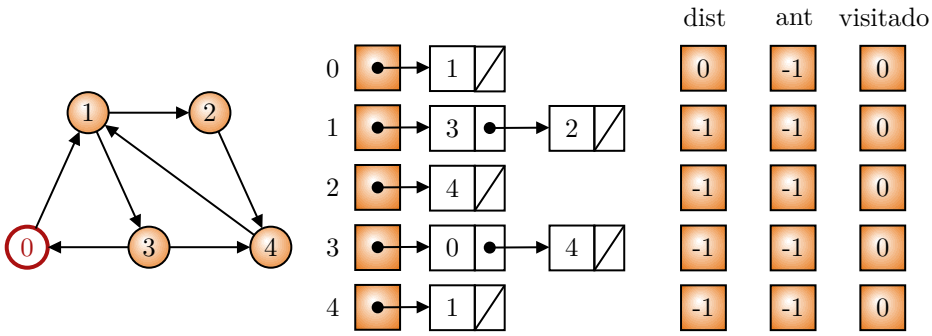
```
int procuraMenorDistancia(float *dist, int *visitado, int NV)
    {
  int i, menor = -1, primeiro = 1;
  for (i=0; i < NV; i++){
    // procura vértice com menor distância e que não tenha
    sido visitado
    if (dist[i] >= 0 && visitado[i] == 0){
      if(primeiro){
        menor = i;
        primeiro = 0;
      } else {
        if (dist[menor] > dist[i])
          menor = i;
      }
    }
  }
  return menor;
}
```

grafo.c – menor caminho

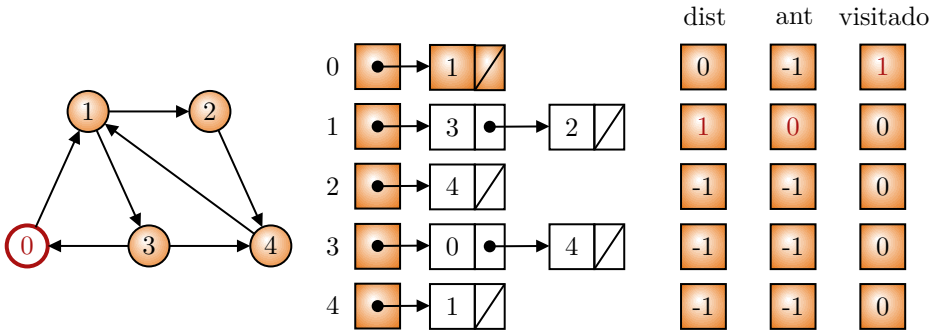
```
void menorCaminho_Grafo(Grafo *gr, int ini, int *ant, float *
    dist){
  int i, cont, NV, ind, *visitado, vert;
  cont = NV = gr->nro_vertices;

  // cria vetor auxiliar. Inicializa distâncias e anteriores
  visitado = (int*) malloc(NV * sizeof(int));
  for(i=0; i < NV; i++){
    ant[i] = -1;
    dist[i] = -1;
    visitado[i] = 0;
  }
  dist[ini] = 0;

  // procura vértice com menor distância e o marca como
    visitado
```

```
  while(cont > 0){
    vert = procuraMenorDistancia(dist, visitado, NV);
    //printf("u = %d\n",u);
    if(vert == -1)
      break;

    visitado[vert] = 1;
    cont--;

    for(i=0; i<gr->grau[vert]; i++){ // para cada vértice vizinho
      ind = gr->arestas[vert][i];
      // atualiza distâncias dos vizinhos
      if(dist[ind] < 0){
         dist[ind] = dist[vert] + 1;//ou peso da aresta
         ant[ind] = vert;
      }else{
        if(dist[ind] > dist[vert] + 1){
          dist[ind] = dist[vert] + 1;//ou peso da aresta
          ant[ind] = vert;
        }
      }
    }
  }

  free(visitado);
}
```

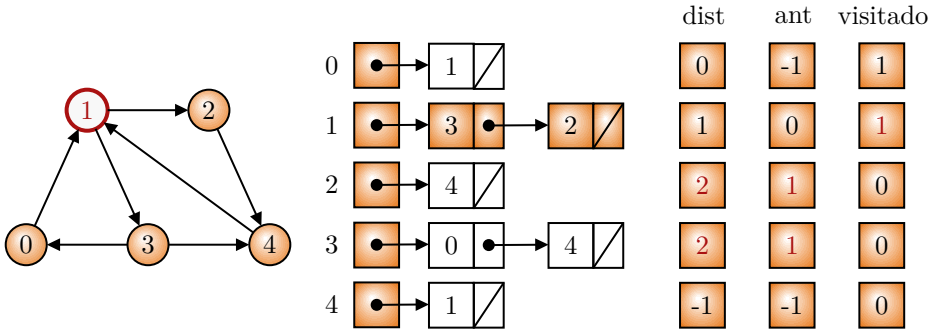
Passo a passo



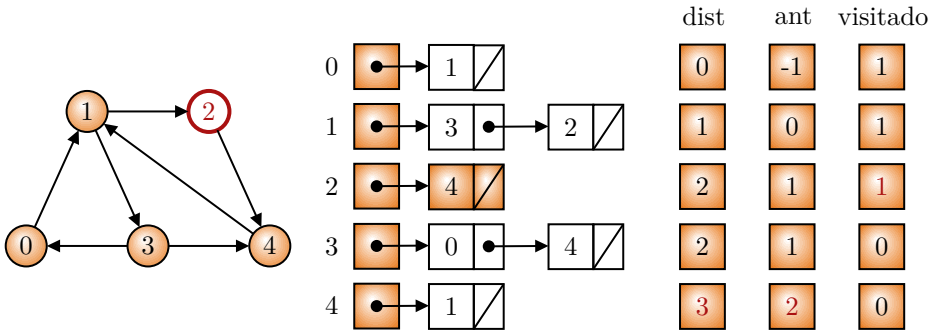
Inicia o cálculo com o vértice 0.
Atribui distância ZERO a ele (início).
O restante dos vértices recebem distância -1



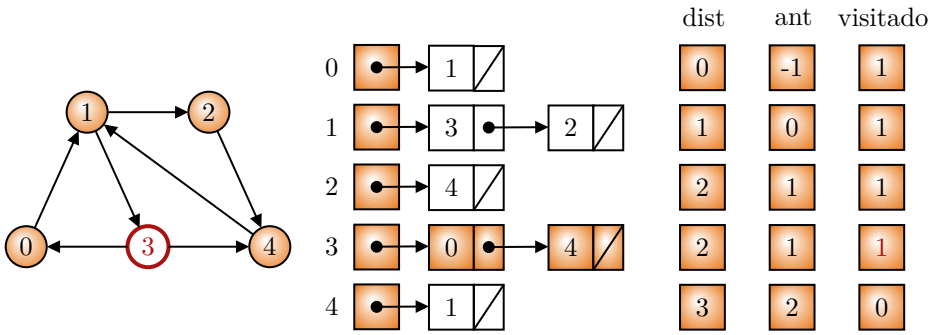
Recupera vértice com menor distância ainda não visitado
e o marca como visitado: vértice 0.
Verifica e atualiza (se necessário) dist e ant do vértice adjacente (1)



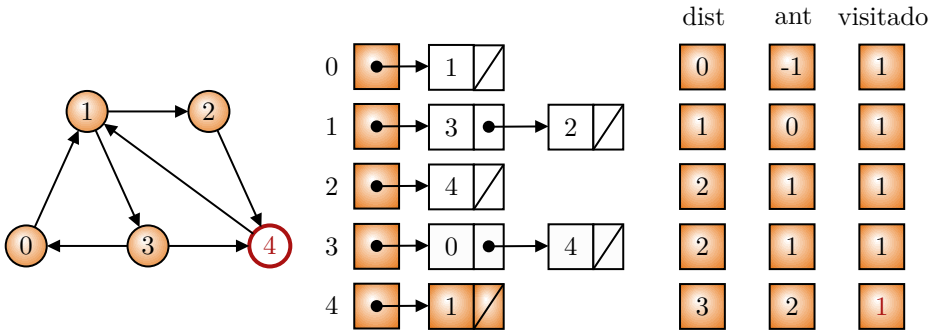
Recupera vértice com menor distância ainda não visitado
e o marca como visitado: vértice 1.
Verifica e atualiza (se necessário) dist e ant dos vértices adjacentes (2 e 3)



Recupera vértice com menor distância ainda não visitado
e o marca como visitado: vértice 2.
Verifica e atualiza (se necessário) dist e ant do vértice adjacente (4)



Recupera vértice com menor distância ainda não visitado
e o marca como visitado: vértice 3.
Verifica e atualiza (se necessário) dist e ant dos vértices adjacentes (0 e 4)



Recupera vértice com menor distância ainda não visitado
e o marca como visitado: vértice 4. Verifica e atualiza (se necessário)
dist e ant do vértice adjacente (1). **Todos visitados: fim**

Aplicações

- Encontrar o grau de separação entre duas pessoas em uma rede social;
- Encontrar um trajeto num mapa rodoviário;
- Programar robôs para explorar áreas;
- Algoritmos de roteamento.

2.5 Exercícios

1. Escreva um algoritmo para verificar se um grafo é acíclico, usando busca em profundidade.
2. Escreva um algoritmo para determinar as componentes conexas de um grafo, usando busca em profundidade.
3. Escreva uma versão não recursiva do algoritmo de busca em profundidade. Dica: o algoritmo de busca em largura usa uma fila. No caso da DFS, qual ED auxiliar seria mais apropriada para evitar a recursão?
4. Adapte o algoritmo de Dijkstra visto para trabalhar com grafos ponderados, e em seguida, explique: por que não é possível trabalhar com pesos menores que zero?
5. Adapte os algoritmos das buscas em profundidade e largura para se trabalhar com matriz de adjacências.

Referências

1. BACKES, A. R. **Algoritmos e Estruturas de Dados em Linguagem C**. 1. ed. Rio de Janeiro: LTC, 2023. Disponível em: https://www.sistemas.ufu.br/biblioteca-gateway/minhabiblioteca/9788521638315.

2. CORMEN, T. H.; et al. **Algoritmos**. 4. ed. Rio de Janeiro: GEN LTC, 2024. Disponível em: https://www.sistemas.ufu.br/biblioteca-gateway/minhabiblioteca/9788595159914.

Os materiais de parte desta seção foram gentilmente cedidos por André R. Backes (UFSCar); Bruno Travençolo e Christiane Brasil (FACOM/UFU).

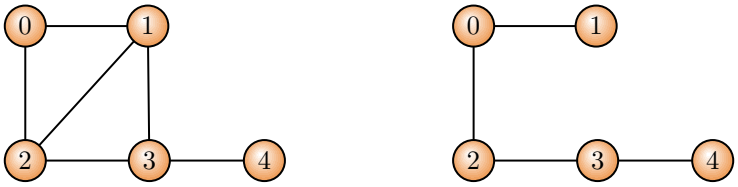
TAD grafo: disponível em: https://www.facom.ufu.br/~backes/wordpress/ProjGrafo.zip. Acesso em: 22/jan/2025.

Adaptações: Renato Pimentel, FACOM/UFU

3 Árvore geradora mínima em grafos

3.1 Introdução

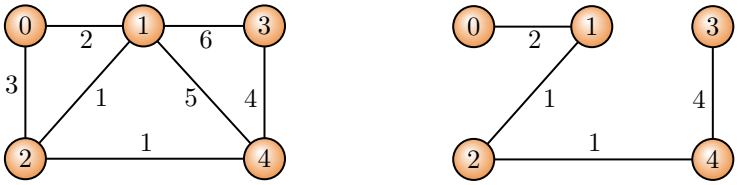
- Dado um grafo $G(V, A)$, com $n \equiv |V|$ vértices, uma *árvore geradora* (em inglês, *spanning tree*) é um subgrafo $T(V, A_T)$ contendo todos os n vértices do grafo original, além de um subconjunto $A_T \subseteq A$ com $|A_T| = n - 1$ arestas que permitam a conexão entre tais vértices.
  - O grafo G deve ser *conexo*;
  - Tal subgrafo T é *acíclico* – daí o termo *árvore*.



Um grafo (esq.) e uma possível árvore geradora (dir.)

Se o grafo, além de conexo, é ponderado, sua *árvore geradora mínima (AGM)* – em inglês, *minimum spanning tree* – é a árvore geradora cujo *custo* (em termos da soma dos pesos das arestas) é o menor possível.

- Além do grafo ser conexo e ponderado, deve ser não-direcionado (i.e. não pode ser dígrafo)
  - Para quaisquer dois vértices na AG, existe um caminho que os une – situação em geral possível somente sem direcionamento.
- Como todos os vértices estão conectados entre si, o cálculo da árvore geradora independe do vértice inicial.



Um grafo ponderado (esq.) e sua árvore geradora mínima (dir.)

Exemplos de uso

- Minimizar custos em redes de comunicação, eletricidade, transporte, etc.
  - Garante a interligação de todos os pontos com o menor custo possível em termos de circuitos integrados.
- Otimização de Infraestrutura em redes de computadores, rodovias, tubulações e distribuição de energia;
- Análise de *clusters* (agrupamentos) em aprendizado de máquina;
- Segmentação de imagens;
- Aplicações em finanças (correlações entre diferentes ativos); etc.

Resolução

- O problema da árvore geradora pode ser resolvido usando uma estratégia que constrói a árvore incrementalmente;
  1. A partir de um grafo $G(V, A)$, começa-se com uma árvore T, com um vértice e nenhuma aresta (*grafo trivial*);
  2. Define-se a *franja* de T como o conjunto de todas as arestas (u, v) de A tais que $u \in T$ mas $v \notin T$ (i.e. arestas conectando um vértice na árvore com outro fora).

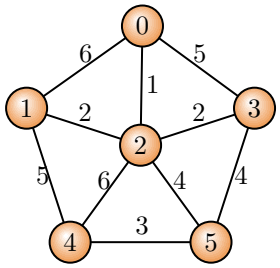
     Passo geral

     enquanto a franja não estiver vazia:

     tome uma aresta (u, v) da franja e troque T por $T + (u, v)$
  3. Ao final, processo termina com franja vazia.
     - Se $|V_T| = |V|$, o grafo é conexo. Caso contrário, V_T é isolado e encontrou-se uma componente conexa.
- Existem dois algoritmos clássicos para obter soluções ótimas para a *árvore geradora mínima* – quando a aresta da franja é a de menor peso:
  - Algoritmo de Prim
  - Algoritmo de Kruskal
- Ambos são *gulosos*: a cada etapa, fazem a escolha melhor possível no momento, dentre as várias possibilidades.
- Diferença entre eles está na regra usada para encontrar as arestas que farão parte da árvore.

Grafo ponderado para teste

A fim de demonstrar os algoritmos de árvore geradora mínima, utilizamos o seguinte grafo ponderado conexo:



Fonte: adaptado de BACKES, A. R. Algoritmos e Estruturas de Dados em Linguagem C. https://www.sistemas.ufu.br/biblioteca-gateway/minhabiblioteca/9788521638315

A criação do mesmo usa a seguinte implementação, considerando-se um máximo de 5 conexões por vértice:

main.c – grafo de teste

```c
#include <stdio.h>
#include <stdlib.h>
#include "Grafo.h"
int main(){
  int eh_digrafo = 0;
  Grafo* gr = cria_Grafo(6, 6, 1);

  insereAresta(gr, 0, 1, eh_digrafo, 6);
  insereAresta(gr, 0, 2, eh_digrafo, 1);
  insereAresta(gr, 0, 3, eh_digrafo, 5);
  insereAresta(gr, 1, 2, eh_digrafo, 2);
  insereAresta(gr, 1, 4, eh_digrafo, 5);
  insereAresta(gr, 2, 3, eh_digrafo, 2);
  insereAresta(gr, 2, 4, eh_digrafo, 6);
  insereAresta(gr, 2, 5, eh_digrafo, 4);
  insereAresta(gr, 3, 5, eh_digrafo, 4);
  insereAresta(gr, 4, 5, eh_digrafo, 3);

  // buscar pela AGM aqui

  libera_Grafo(gr);
  system("pause");
  return 0;
}
```

3.2 Algoritmo de Prim

- Parte de um vértice inicial;
- A cada iteração, busca a aresta de *menor peso* que conecte um vértice da árvore a outro que ainda não esteja na mesma;
- O novo vértice é adicionado a árvore, e repete-se o processo;
- O algoritmo prossegue até que:
  - Todos os vértices façam parte da árvore;
  - Não se pode encontrar uma aresta que satisfaça essa condição (*grafo desconexo*).
- A função que obtém a árvore recebe três parâmetros:
  1. o grafo
  2. o vértice inicial da busca
  3. um *array* que indicará, para cada vértice, qual o seu *pai*, ou seja, o vértice anterior na AGM – correspondendo ao *resultado* da busca.
     - o vértice do *array* cujo pai é seu próprio índice representa a *raiz* da árvore.

main.c – algoritmo de Prim

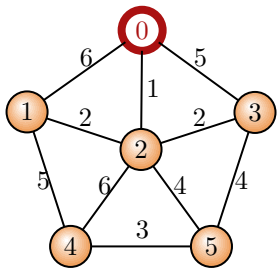
```
  .
  .
  insereAresta(gr, 4, 5, eh_digrafo, 3);

  int pai[6];
  arvoreGeradoraMinimaPrim_Grafo(gr, 0, pai);
  .
  .
```

grafo.c – algoritmo de Prim

```
void arvoreGeradoraMinimaPrim_Grafo (Grafo *gr, int orig, int
    *pai) {
  int i, j, dest, primeiro, NV = gr->nro_vertices;
  double menorPeso;

  // vértices inicialmente não têm pai (exceto orig)
  for(i=0; i < NV; i++)
    pai[i] = -1; // sem pai
  pai[orig] = orig; // vértice inicial: raiz da AGM

```

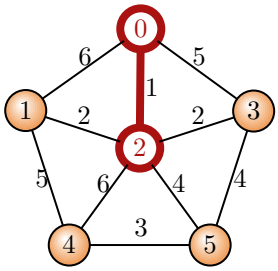
```
  while (1){
    primeiro = 1;
    // percorre todos os vértices, e
    // procura aresta de menor peso ligando um vértice
    // já na árvore a outro ainda não inserido
    for(i=0; i < NV; i++) {
      if (pai [i] != -1){ // vértice já inserido:
        // percorre-se seus vizinhos
        for(j=0; j<gr->grau[i]; j++) {
          int vizinho = gr->arestas[i][j];
          if (pai[vizinho] == -1){ // não inserido
            // busca aresta com menor peso
            if (primeiro) {
              menorPeso = gr->pesos[i][j];
              orig = i;
              dest = vizinho;
              primeiro = 0;
            } else {
              if (menorPeso > gr->pesos[i][j]){
                menorPeso = gr->pesos[i][j];
                orig = i;
                dest = vizinho;
              }
            }
          }
        }
      }
    }
    if (primeiro == 1)
      break;

    pai[dest] = orig;
  }
}
```

Passo a passo



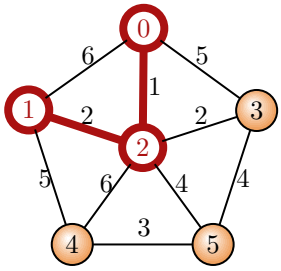
	pai
0	0
1	-1
2	-1
3	-1
4	-1
5	-1

Inicia o cálculo com o vértice 0.
Atribui seu próprio índice como pai.
O restante dos vértices recebem pai igual a -1 (sem pai)



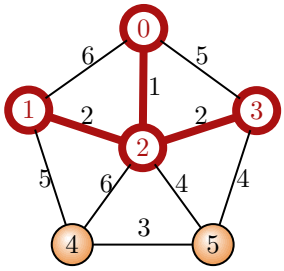
	pai
0	0
1	-1
2	0
3	-1
4	-1
5	-1

Procura nos vértices com pai por um vizinho sem pai e com menor peso: vértice 2.
Atribui vértice 0 como pai do vértice 2.



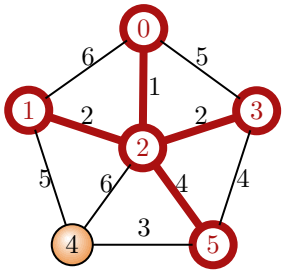
	pai
0	0
1	2
2	0
3	-1
4	-1
5	-1

Procura nos vértices com pai por um vizinho sem pai e com menor peso: vértice 1.
Atribui vértice 2 como pai do vértice 1.



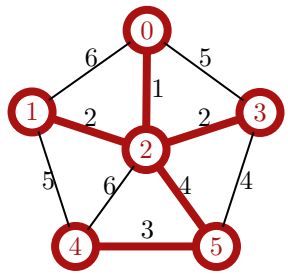
	pai
0	0
1	2
2	0
3	2
4	-1
5	-1

Procura nos vértices com pai por um vizinho sem pai e com menor peso: vértice 3.
Atribui vértice 2 como pai do vértice 3.



	pai
0	0
1	2
2	0
3	2
4	-1
5	2

Procura nos vértices com pai por um vizinho sem pai e com menor peso: vértice 5.
Atribui vértice 2 como pai do vértice 5.



	pai
0	0
1	2
2	0
3	2
4	5
5	2

Procura nos vértices com pai por um vizinho sem pai e com menor peso: vértice 4.
Atribui vértice 5 como pai do vértice 4. **Fim do cálculo.**

Complexidade

- Considerando um grafo $G(V, A)$, onde $|V|$ é o número de vértices e $|A|$ é o número de arestas, a complexidade no pior caso é $O(|V| \times |A|)$.
  - Como $|A|$ é proporcional a $|V|^2$, o custo é $O(|V|^3)$.
- A eficiência depende da forma usada para procurar a aresta de menor peso.
  - Usando uma *fila de prioridades* para as arestas, o custo pode ser reduzido para $O(|A| log |V|)$. *Sugestão de leitura:* https://www.ime.usp.br/~pf/analise_de_algoritmos/aulas/MST-prim.html#sec:prim3

3.3 Algoritmo de Kruskal

- Considera cada vértice como uma árvore independente;
- A cada iteração, o algoritmo procura a aresta de *menor peso* que conecta duas árvores diferentes; Os vértices das árvores selecionadas passam a fazer parte de uma mesma árvore;
- O algoritmo prossegue até que:
  - Todos os vértices façam parte da árvore;
  - Não se pode encontrar uma aresta que satisfaça essa condição (*grafo desconexo*).

Diferença entre ambos os algoritmos

- O algoritmo de Prim se inicia com um vértice e cresce uma única árvore a partir dele;
- O algoritmo de Kruskal constrói uma *floresta* (várias árvores) ao longo do tempo, e são unidas ao final do processo.

- A função que obtém a árvore recebe três parâmetros:
  1. o grafo
  2. o vértice inicial da busca
  3. um *array* que indicará, para cada vértice, qual o seu *pai*, ou seja, o vértice anterior na AGM – correspondendo ao *resultado* da busca.
     - o vértice do *array* cujo pai é seu próprio índice representa a *raiz* da árvore.

main.c – algoritmo de Kruskal

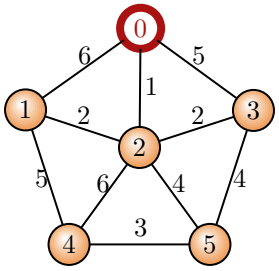
```
  .
  .
  insereAresta(gr, 4, 5, eh_digrafo, 3);

  int pai[6];
  arvoreGeradoraMinimaKruskal_Grafo(gr, 0, pai);
  .
  .
```

grafo.c – algoritmo de Kruskal

```
void arvoreGeradoraMinimaKruskal_Grafo (Grafo *gr, int orig,
    int *pai) {
  int i, j, dest, primeiro, NV = gr->nro_vertices;
  double menorPeso;
  int *arv = (int*) malloc(NV*sizeof(int));

  // cada vértice inicialmente é uma árvore sem pai
  for(i=0; i < NV; i++) {
    pai[i] = -1; // sem pai
    arv[i] = i;
  }
  pai[orig] = orig; // vértice inicial: raiz da AGM

  while (1){
    primeiro = 1;
    // percorre todos os vértices, e
    // Procura aresta de menor peso ligando
    // árvores diferentes
```

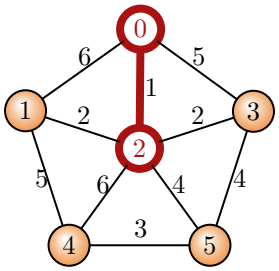
```
    for(i=0; i < NV; i++) {
      for(j=0; j<gr->grau[i]; j++) {
        int vizinho = gr->arestas[i][j];
        if (arv[i] != arv[vizinho]){ // não inserido
          // busca aresta com menor peso
          if (primeiro) {
            menorPeso = gr->pesos[i][j];
            orig = i;
            dest = vizinho;
            primeiro = 0;
          } else {
            if (menorPeso > gr->pesos[i][j]){
              menorPeso = gr->pesos[i][j];
              orig = i;
              dest = vizinho;
            }
          }
        }
      }
    }
    if (primeiro == 1)
      break;

    if (pai[orig] == -1)
      pai[orig] = dest;
    else
      pai[dest] = orig;

    // une as duas árvores da floresta selecionada
    for(i=0; i < NV; i++)
      if (arv[i] == arv[dest])
        arv[i] = arv[orig];
  }
  free(arv);
}
```

Passo a passo



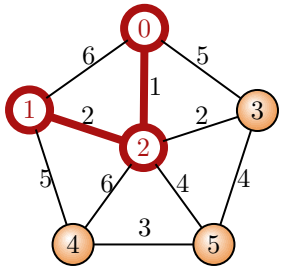
	pai	arv
0	0	0
1	-1	1
2	-1	2
3	-1	3
4	-1	4
5	-1	5

Inicia o cálculo com o vértice 0.
Atribui seu próprio índice como pai.
O restante dos vértices recebem pai igual a -1 (sem pai)
Inicializa a árvore com o índice do vértice.



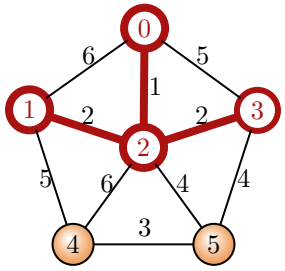
	pai	arv
0	0	0
1	-1	1
2	0	0
3	-1	3
4	-1	4
5	-1	5

Procura a aresta com menor peso conectando vértices com árvores diferentes: vértices 0 e 2.
Atribui vértice 0 como pai do vértice 2.
Todos que possuem árvore igual ao vértice 2 passam a ter árvore igual ao vértice 0.



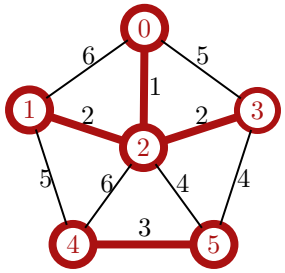
	pai	arv
0	0	1
1	2	1
2	0	1
3	-1	3
4	-1	4
5	-1	5

Procura a aresta com menor peso conectando vértices com árvores diferentes: vértices 1 e 2.
Atribui vértice 2 como pai do vértice 1.
Todos que possuem árvore igual ao vértice 2 passam a ter árvore igual ao vértice 1.



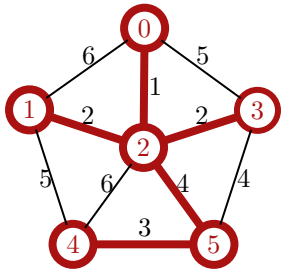
	pai	arv
0	0	1
1	2	1
2	0	1
3	2	1
4	-1	4
5	-1	5

Procura a aresta com menor peso conectando vértices com árvores diferentes: vértices 2 e 3.
Atribui vértice 2 como pai do vértice 3.
Todos que possuem árvore igual ao vértice 3 passam a ter árvore igual ao vértice 1.



	pai	arv
0	0	1
1	2	1
2	0	1
3	2	1
4	5	4
5	-1	4

Procura a aresta com menor peso conectando vértices com árvores diferentes: vértices 4 e 5.
Atribui vértice 2 como pai do vértice 3.
Todos que possuem árvore igual ao vértice 3 passam a ter árvore igual ao vértice 1.



	pai	arv
0	0	1
1	2	1
2	0	1
3	2	1
4	5	1
5	2	1

Procura a aresta com menor peso conectando vértices com árvores diferentes: vértices 2 e 5.
Atribui vértice 2 como pai do vértice 5.
Todos que possuem árvore igual ao vértice 4 passam a ter árvore igual ao vértice 1. **Fim do cálculo.**

Complexidade

- Considerando um grafo $G(V, A)$, onde $|V|$ é o número de vértices e $|A|$ é o número de arestas, a complexidade no pior caso é $O(|V| \times |A|)$.
  - Como $|A|$ é proporcional a $|V|^2$, o custo é $O(|V|^3)$.
- A eficiência depende da forma usada para procurar a aresta de menor peso.
  - Usando uma estrutura de dados união-busca (*Union & Find*), o custo pode ser reduzido para $O(|A| log|V|)$ – veja https://www.ime.us

p.br/~pf/analise_de_algoritmos/aulas/MST-kruskal.html#sec:implementation.

3.4 Exercícios

1. (CORMEN, 2024) Forneça uma implementação simples do algoritmo de Prim, que seja executada no tempo $O(|V|^2)$, quando o grafo $G(V, E)$ é representado como uma matriz de adjacências.

2. Mostre que um grafo G tem uma árvore geradora se, e somente se, G é conexo. (Sugestão: Use o algoritmo da árvore geradora.)

3. Escreva um algoritmo que conte o número de componentes de um grafo. (Sugestão: Use o algoritmo da árvore geradora.)

Referências

1. BACKES, A. R. **Algoritmos e Estruturas de Dados em Linguagem C**. 1. ed. Rio de Janeiro: LTC, 2023. Disponível em: https://www.sistemas.ufu.br/biblioteca-gateway/minhabiblioteca/9788521638315.

2. CORMEN, T. H.; et al. **Algoritmos**. 4. ed. Rio de Janeiro: GEN LTC, 2024. Disponível em: https://www.sistemas.ufu.br/biblioteca-gateway/minhabiblioteca/9788595159914.

Os materiais de parte desta seção foram gentilmente cedidos por André R. Backes (UFSCar); Bruno Travençolo e Christiane Brasil (FACOM/UFU).

TAD grafo: disponível em: https://www.facom.ufu.br/~backes/wordpress/ProjGrafo.zip. Acesso em: 22/jan/2025.

Adaptações: Renato Pimentel, FACOM/UFU

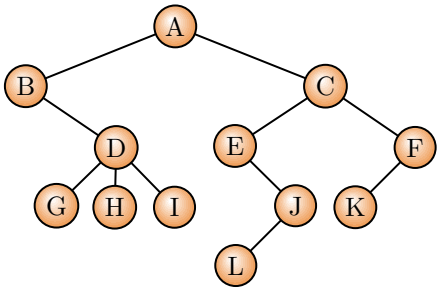
4 Introdução a árvores

4.1 Definições básicas

Árvore

Uma *árvore* é uma abstração matemática usada para representar estruturas hierárquicas não lineares dos objetos modelados.

- É um tipo especial de grafo:
  - Definida usando um conjunto de nós (ou vértices) e arestas
  - Qualquer par de vértices está conectado a apenas uma aresta
    - Grafo não direcionado, conexo e *acíclico* (sem ciclos).



Exemplos de aplicações

1. *Genealogia*: Relações de descendência (pai, filho, etc.)
2. *Gestão/administração*: diagrama hierárquico de uma organização; campeonatos de modalidades desportivas;
3. *Biologia*: taxonomia.
4. *Computação*:
   - *Árvores trie* (ou árvores de prefixo): usadas em mecanismos de busca e corretores ortográficos.
   - Representação do *espaço de soluções* de um problema.
     - Ex.: jogo de xadrez – cada elemento da árvore correspondente a uma determinada posição das peças no tabuleiro;
   - Modelagem de algoritmos:
     - Algoritmos e busca em largura (BFS) e profundidade (DFS), árvores de decisão (IA), etc.

Árvore de decisão

Devemos jogar tênis hoje ou não?

```
                (Como está o tempo?)
              /         |         \
       Ensolarado    Nublado    Chuvoso
          /             |           \
  (Umidade?)          Sim       (Vento?)
    /       \                    /      \
Alta      Normal            Fraco     Forte
 |           |                |          |
Não        Sim              Sim        Não
```

Fonte: adaptado de HEMMAT, HAFID (2016).
https://arxiv.org/abs/1611.10338

Estrutura de pastas e arquivos do S. O.

Fonte: FrancoGG - Obra do próprio, CC BY-SA 3.0,
https://commons.wikimedia.org/w/index.php?curid=766938

Conceitos

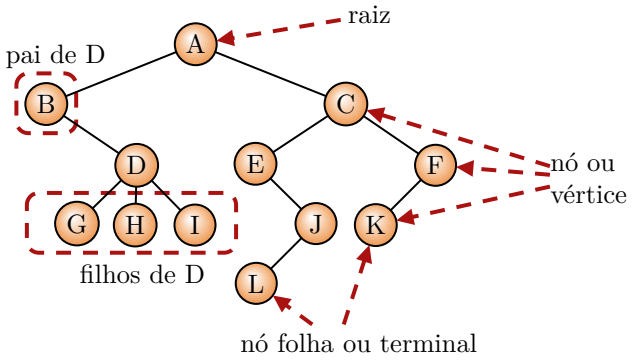
- *Vértice*, ou *nó*, é cada uma das entidades representadas na árvore.
- A representação do vértice depende do problema estudado:
  - Organização do sistema operacional, onde cada vértice representa justamente uma pasta ou diretório do sistema.
  - Árvore *trie*: cada vértice pode representar uma letra ou prefixo.

```
    (C)
   /   \
 (A)   (Ã)
```

```
      /    \      \
    (S)    (R)    (O)
    /        |
  (A)        (R)
   |          |
  ( )        (O)
```

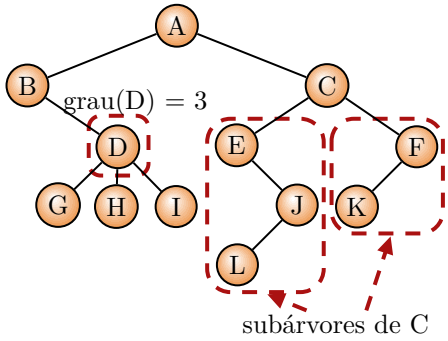
Fonte: https://chatgpt.com

- Assim como em grafos, cada *aresta* representa a ligação (conexão) existente entre dois vértices.
- *Raiz*: vértice mais alto na árvore, o único que não possui pai.
- *Pai* ou *ancestral*: vértice antecessor imediato de outro vértice.
- *Filho*: o vértice sucessor imediato de outro vértice.
- *Nó folha* ou *terminal*: qualquer vértice que não possui filhos.
- *Nó interno* ou *não-terminal*: vértice que possui ao menos um filho.
- *Caminho*: sequência de nós, de modo que sempre existe uma aresta conectando o nó anterior ao seguinte.
  - Existe exatamente um caminho da *raiz* até qualquer um dos vértices.



Subárvores e grau

- Dado um nó, cada filho é considerado a raiz de uma nova *subárvore*;
- Qualquer nó é raiz de uma subárvore que consiste do próprio e dos nós abaixo dele (conceito recursivo).



- O *grau* de um nó é o número de subárvores (ou filhos) que possui.
  - Nós folhas possuem grau zero.

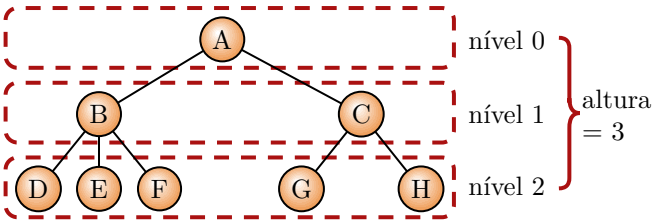
Nível e altura

Nível

- É dado pelo número de nós no caminho entre o nó em questão e a raiz (*nível 0*).
- Nós são classificados em diferentes níveis.

Altura

- Ou *profundidade*, corresponde ao total de níveis de uma árvore;
- Comprimento do *caminho mais longo* da raiz até uma de suas folhas.



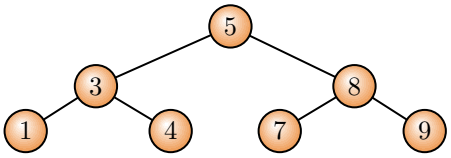
Tipos de árvores

- Na computação, assim como na natureza, existem vários tipos diferentes de árvores.
- Cada uma delas foi desenvolvida pensando-se em distintas possibilidades de aplicações:
  1. árvore binária de busca;

2. árvore AVL;
3. árvore rubro-negra;
4. árvore B, B+ e B*;
5. árvore 2-3;
6. árvore 2-3-4;
7. quadtree;
8. octree.

Árvore binária de busca

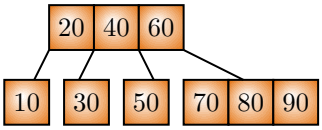
- *Árvore binária* onde, para cada nó, seus descendentes da subárvore da esquerda (direita) possuem valores menores (maiores) que o mesmo.



- Útil em diversas aplicações, como bancos de dados (eficiência de índices de pesquisa), compiladores (análise de símbolos), IA (tomadas de decisão), dicionários, filtragem de informações (*firewall*), etc.
- Dependendo das operações de inserção/remoção, árvore pode ficar *desbalanceada*, prejudicando eficiência. Duas variantes sanam o problema:
  1. Árvore AVL;
  2. Árvore rubro-negra (ou *red-black*).

Árvore B e variantes

- A árvore B e suas variantes (B+ e B*) são EDs balanceadas usadas principalmente para armazenar e organizar grandes volumes de dados em sistemas que requerem acesso eficiente, como bancos de dados e sistemas de arquivos.



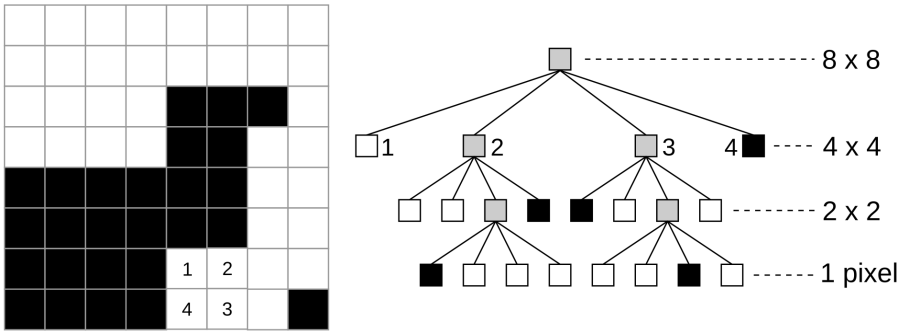
- Evoluções das árvores binárias de busca, otimizadas para minimizar acessos a disco e maximizar eficiência em leitura e escrita.

- *Árvore B+*: dados armazenados somente nas folhas;
- *Árvore B**: otimização para muitas inserções/remoções (política de divisão);
- *Árvores 2-3* e *2-3-4*: Árvores B onde cada nó pode ter 1 ou 2 chaves (e 2 ou 3 filhos); ou 1 a 3 chaves e 2 a 4 filhos, respectivamente.

Quadtree e octree

- *Quadtree*: ED usada para armazenar e organizar informações espaciais de forma eficiente. Particiona recursivamente um espaço 2D em quatro regiões menores, facilitando operações como busca, inserção e remoção de elementos.
- Os dados armazenados em cada folha dependem da aplicação. Mais comuns: computação gráfica (armazenamento, compressão e renderização de imagens), Sistemas de Informações Geográficas (GIS), simulações físicas e jogos (colisões), indexação espacial (buscas), visão computacional.
- *Octree*: equivalente 3D da quadtree.

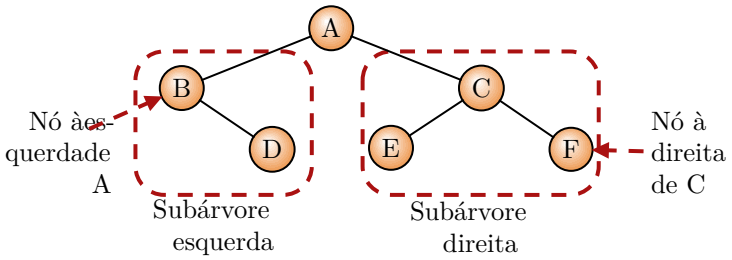
Fonte: Lkjhsdfljsd - Own work, CC BY-SA 4.0,
https://commons.wikimedia.org/w/index.php?curid=47469171



Fonte: Wojciech Muła - Own work, Public Domain,
https://commons.wikimedia.org/w/index.php?curid=4878903

4.2 Árvores binárias

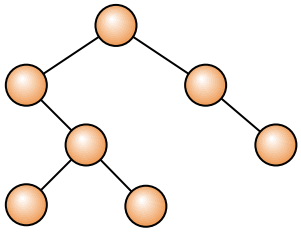
- Árvores onde cada nó pode possuir nenhuma, uma ou *no máximo duas subárvores*:
  - Subárvore da *esquerda* e a da *direita*.
- Úteis em situações onde a cada passo é preciso tomar uma decisão entre duas direções.
  - Ex.: bancos de dados, compiladores, compressão de dados (*algoritmo de Huffman*), representação de expressões aritméticas, etc.



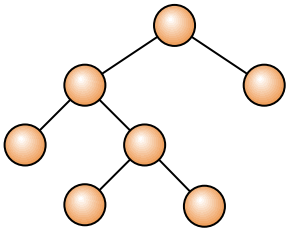
- Existem três tipos de árvores binárias:
  1. Estritamente binária;
  2. Quase completa;
  3. Cheia.

Árvore estritamente binária

- Cada nó possui sempre ou zero (no caso de nó folha) ou duas subárvores;
- Nenhum nó tem *filho único*.



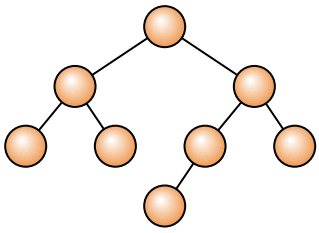
Árvore binária



Árvore estritamente binária

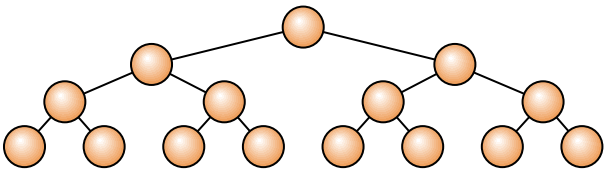
Árvore binária quase completa

- A diferença de altura entre as subárvores de qualquer nó é no máximo 1.
- Se a altura da árvore é H, então cada nó folha está no nível $H-1$ (último) ou $H-2$.



Árvore binária cheia ou completa

- Árvore estritamente binária onde todos os nós folhas estão no mesmo nível.



- Características:
  - Há exatamente 2^n nós no nível n;
  - Uma árvore de altura H possui $2^H - 1$ nós.

4.3 Representações

Representação computacional

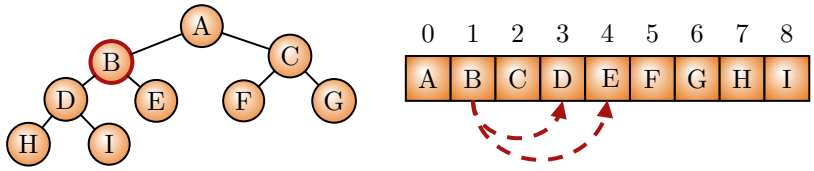
- Duas abordagens principais:
  - Usando um *array* – *Alocação estática*;
  - Usando uma lista encadeada – *Alocação dinâmica*.
- Assim como em grafos, a melhor representação, dentre as duas, depende da aplicação. Não existe uma ideal para todos os casos.
- Independentemente do tipo de alocação usado na implementação da árvore, as seguintes operações básicas são possíveis:
  1. Criação da árvore;
  2. inserção de um elemento;
  3. remoção de um elemento;
  4. busca por um elemento;
  5. destruição da árvore;
  6. informações como número de nós, altura ou se está vazia.

Usando um *array* (*heap*)

- *Alocação estática*: há um máximo de nós – tamanho do *array*.
- A *heap* permite uma árvore binária completa ou quase completa.
- Usa 2 funções para a posição dos filhos à esquerda e à direita:

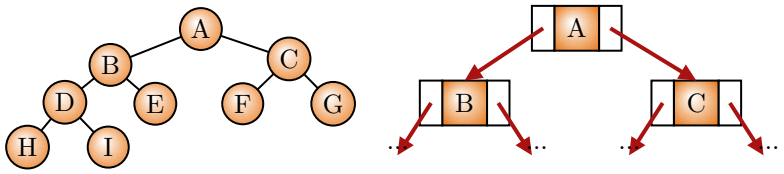
```
FILHO_ESQ(PAI) = 2 * PAI + 1
FILHO_DIR(PAI) = 2 * PAI + 2
```

- Uso: quando todos os elementos e a estrutura da árvore são conhecidos *a priori*. Mais adequado para *buscas* frequentes.



Usando lista encadeada (alocação dinâmica)

- Espaço de memória alocado em tempo de execução;
- Cresce à medida que novos elementos são armazenados, e diminui à medida que elementos são removidos.
- Uso: não necessita que se conheça previamente todos os elementos a serem inseridos na árvore. A árvore a ser representada não precisa ser uma árvore binária completa ou quase completa.
  - Mais adequado na maioria das situações.



4.4 Um tipo abstrato de dados (TAD) para árvore binária

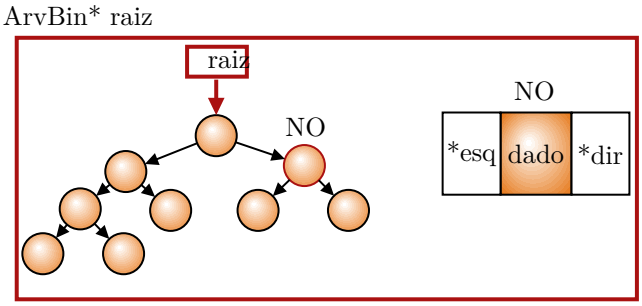
- Consideramos um tipo abstrato de dados *opaco* em Linguagem C para representação *encadeada*.

  Relembrando: Tipo opaco

  - A ED será referenciada mediante um *ponteiro* para a mesma, ao invés de uma variável do tipo "árvore binária" diretamente.
  - Isto permite ocultar a implementação, permitindo sua alteração sem interferir no funcionamento de qualquer código que o utilize – desde que a *interface* seja respeitada.

Alocação dinâmica

- Para guardar o primeiro nó da árvore utilizamos *um ponteiro para ponteiro*;
  - armazena-se o *endereço de um ponteiro*.
- Possibilita mudar quem é a *raiz* da árvore (caso necessário).
  - Mudança não fica "visível" para o usuário da ED.



- Modularização do TAD:
  - *ArvoreBinaria.h*: a *interface* em si: definição do tipo opaco; funções disponíveis para manipular a árvore binária (protótipos).
  - *ArvoreBinaria.c*: o que deve ficar oculto do usuário, além da implementação das funções definidas pela interface. Inclui a definição do *tipo nó*.
- *Tipo nó*: estrutura que contém:
  - `info`: Representa a *informação*, ou seja, os dados correspondentes ao elemento (nó/vértice) em questão;
    - Para fins de simplicidade, será no TAD ilustrado um `int`.
  - `*esq` e `*dir`: ponteiros para os nós filhos à esquerda e à direita do nó.

Interface: ArvoreBinaria.h

ArvoreBinaria.h

```
typedef struct NO* ArvBin;

ArvBin* cria_ArvBin();
void libera_ArvBin(ArvBin *raiz);
int estaVazia_ArvBin(ArvBin *raiz);
int altura_ArvBin(ArvBin *raiz);
int totalNO_ArvBin(ArvBin *raiz);
```

Outras funções serão vistas posteriormente.

Definição do nó: ArvoreBinaria.c

ArvoreBinaria.c: bibliotecas usadas e definição do tipo

```
include <stdio.h>
#include <stdlib.h>
#include "ArvoreBinaria.h" //inclui os protótipos

struct NO{
```

```
  int info;
  struct NO *esq;
  struct NO *dir;
};
```

Criando uma árvore binária

Exemplo de uso

```
ArvBin* raiz = cria_ArvBin();
```

ArvoreBinaria.c: criação da árvore binária

```
ArvBin* cria_ArvBin(){
  ArvBin* raiz = (ArvBin*) malloc(sizeof(ArvBin));
  if(raiz != NULL)
    *raiz = NULL;
  return raiz;
}
```

Liberando a árvore binária da memória

Uso de duas funções:

uma auxiliar (não disponível na interface) percorre e libera os nós; outra – a que o usuário chama – trata a raiz

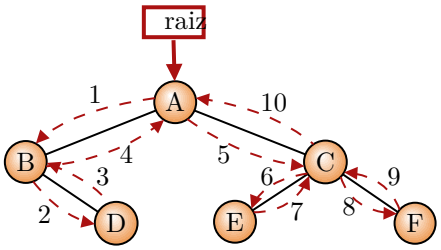
Exemplo de uso

```
ArvBin* raiz = cria_ArvBin();
.
.
libera_ArvBin(raiz);
```

ArvoreBinaria.c: liberando a árvore binária

```
void libera_NO(struct NO* no){
  if(no == NULL)
    return;
  libera_NO(no->esq);
  libera_NO(no->dir);
  free(no);
  no = NULL;
}

void libera_ArvBin(ArvBin* raiz){
  if(raiz == NULL)
    return;
  libera_NO(*raiz); // libera cada nó
  free(raiz); // libera a raiz
}
```

Passo-a-passo



	1	visita B
	2	visita D
×	3	libera D, volta p/ B
×	4	libera B, volta p/ A
	5	visita C
	6	visita E
×	7	libera E, volta p/ C
	8	visita F
×	9	libera F, volta p/ C
×	10	libera C, volta p/ A e
×		libera A
		libera raiz

Informações básicas sobre a árvore binária

Algumas informações sobre a árvore estão inclusas no TAD, como:

- A árvore binária está vazia?
- Quantos elementos (nós) possui a árvore?
- Qual a *altura* da árvore?

Verificando se a árvore binária está vazia

Exemplo de uso

```
ArvBin* raiz = cria_ArvBin();
.
.
if (estaVazia_ArvBin(raiz))
  printf("A árvore binária está vazia\n");
```

ArvoreBinaria.c: árvore está vazia?

```
34 int estaVazia_ArvBin(ArvBin *raiz){
35   if(raiz == NULL)
36     return 1;
37   if(*raiz == NULL)
38     return 1;
39   return 0;
40 }
```

- Primeiro teste verifica se ponteiro `ArvBin *raiz` é `NULL` – *problema na criação da árvore*;
- Segundo teste verifica se existe uma raiz (árvore criada com sucesso, mas não possui nenhum nó).

Quantidade de elementos na árvore

Exemplo de uso

```
ArvBin* raiz = cria_ArvBin();
.
.
printf("A árvore possui %d nós\n", totalNO_ArvBin(raiz));
```

ArvoreBinaria.c: quantidade de nós

```
42 int totalNO_ArvBin(ArvBin *raiz){
43   if (raiz == NULL)
44     return 0;
45   if (*raiz == NULL)
46     return 0;
47   int totalNO_esq = totalNO_ArvBin(&((*raiz)->esq));
48   int totalNO_dir = totalNO_ArvBin(&((*raiz)->dir));
49   return (totalNO_esq + totalNO_dir + 1);
50 }
```

- `totalNO_ArvBin` recebe como parâmetro um *ponteiro para ponteiro* – `ArvBin*` ou `struct NO**`;
- Como os nós são ponteiros simples (`struct NO *`), é preciso passar os *endereços* destes nas chamadas recursivas.

Altura da árvore

Exemplo de uso

```
ArvBin* raiz = cria_ArvBin();
.
.
printf("A árvore tem altura %d\n", altura_ArvBin(raiz));
```

ArvoreBinaria.c: altura

```
52 int altura_ArvBin(ArvBin *raiz){
53   if (raiz == NULL)
54     return 0;
55   if (*raiz == NULL)
56     return 0;
57   int alt_esq = altura_ArvBin(&((*raiz)->esq));
58   int alt_dir = altura_ArvBin(&((*raiz)->dir));
59   if (alt_esq > alt_dir)
60     return (alt_esq + 1);
61   else
62     return(alt_dir + 1);
63 }
```

- Como no caso anterior, `altura_ArvBin` recebe como parâmetro um *ponteiro para ponteiro* – `ArvBin*` ou `struct NO**`;
- Como os nós são ponteiros simples (`struct NO *`), é preciso passar os *endereços* destes nas chamadas recursivas.

4.5 Exercícios

1. Escreva uma função que percorra uma árvore binária e conte o seu número de nós.
2. Escreva uma função que percorra uma árvore binária e conte o seu número de nós não-folhas.

Referências

1. BACKES, A. R. **Algoritmos e Estruturas de Dados em Linguagem C**. 1. ed. Rio de Janeiro: LTC, 2023. Disponível em: https://www.sistemas.ufu.br/biblioteca-gateway/minhabiblioteca/9788521638315.
2. CORMEN, T. H.; et al. **Algoritmos**. 4. ed. Rio de Janeiro: GEN LTC, 2024. Disponível em: https://www.sistemas.ufu.br/biblioteca-gateway/minhabiblioteca/9788595159914.

Os materiais de parte desta seção foram gentilmente cedidos por André R. Backes (UFSCar); Bruno Travençolo e Christiane Brasil (FACOM/UFU).

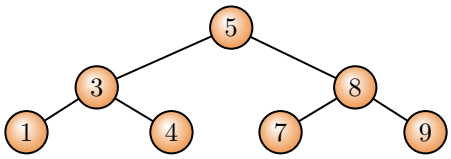
TAD árvore binária de busca: disponível em: https://www.facom.ufu.br/~backes/wordpress/ProjArvoreBinaria.zip. Acesso em: 24/fev/2025.

Adaptações: Renato Pimentel, FACOM/UFU

5 Árvore binária de busca

5.1 Definição

- *Árvore binária* – cada nó possui 0, 1 ou 2 filhos.
- Comumente referenciada pela sigla *BST* – *binary search tree*.
- Todo nó possui um *valor* (*chave*) associado.
  - Não existem chaves repetidas.
- Chave determina a *posição* do nó na ABB. Para cada nó pai:
  - todos os valores da *subárvore da esquerda* são *menores* do que o nó pai
  - todos os valores da *subárvore da direita* são *maiores* do que o nó pai;
- Inserção e remoção devem ser realizadas respeitando tal regra.



- Ótima alternativa para operações de busca binária.
  - Possui a vantagem de ser uma estrutura dinâmica em comparação ao *array* (*heap*);
  - Mais fácil inserir valores na árvore do que num *array* ordenado:
    - *Array*: envolve deslocamento de elementos.

Custo computacional assintótico

- Custo para as principais operações em uma árvore binária de busca contendo n nós.
  - O pior caso ocorre quando a ABB não está *balanceada*.

	Melhor Caso	**Pior Caso**
Inserção	$O(\log_2 n)$	$O(n)$
Busca	$O(\log_2 n)$	$O(n)$
Remoção	$O(\log_2 n)$	$O(n)$

No TAD Árvore binária:

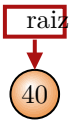
ArvoreBinaria.h

```
typedef struct NO* ArvBin;

ArvBin* cria_ArvBin();
void libera_ArvBin(ArvBin *raiz);
int estaVazia_ArvBin(ArvBin *raiz);
int altura_ArvBin(ArvBin *raiz);
int totalNO_ArvBin(ArvBin *raiz);
int insere_ArvBin(ArvBin* raiz, int valor);
int consulta_ArvBin(ArvBin *raiz, int valor);
int remove_ArvBin(ArvBin *raiz, int valor);
```

5.2 Inserção na ABB

- Para inserir um valor V na ABB:
  1. Se a raiz é igual a `NULL`, insira o nó;
  2. Se V é *menor que* a raiz: siga para a *subárvore da esquerda*;
  3. Se V é *maior que* a raiz: siga para a *subárvore da direita*;
  4. Aplique o método recursivamente – também é possível sem recursão.
- Percorre-se os nós da ABB até o *nó folha* que se tornará *pai* do novo nó.
- Também deve se considerar inserção numa árvore vazia;
- Neste caso, novo elemento será a *raiz* da ABB.

Insere valor 40
em uma árvore vazia



*raiz = novo;

Fonte: adaptado de BACKES (2023).

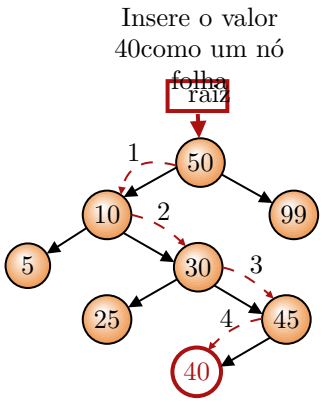
Exemplo de uso

```
int N = 8, dados[8] = {50,100,30,20,40,45,35,37};
ArvBin* raiz = cria_ArvBin();
int i;
for(i=0; i<N; i++)
  insere_ArvBin(raiz, dados[i]);
```

ArvoreBinaria.c: inserção de nó

```
65 int insere_ArvBin(ArvBin* raiz, int valor){
66   if(raiz == NULL)
67     return 0;
68   struct NO* novo;
69   novo = (struct NO*) malloc(sizeof(struct NO));
70   if(novo == NULL)
71     return 0;
72   novo->info = valor;
73   novo->dir = NULL;
74   novo->esq = NULL;
75
76   if(*raiz == NULL)
77     *raiz = novo;
78   else{
79     struct NO* atual = *raiz;
80     struct NO* ant = NULL;
81
82     // Navega nos nós da árvore até chegar em um nó folha
83     while(atual != NULL){
84       ant = atual;
85       if(valor == atual->info){
86         free(novo);
87         return 0; // elemento já existe
88       }
89
90       if(valor > atual->info)
91         atual = atual->dir;
92       else
93         atual = atual->esq;
94     }
95
96     // Insere como filho desse nó folha
97     if(valor > ant->info)
98       ant->dir = novo;
99     else
100      ant->esq = novo;
101  }
102  return 1;
103 }
```

Passo-a-passo



1	valor é menor do que 50: visita filho da esquerda
2	valor é maior do que 10: visita filho da direita
3	valor é maior do que 30: visita filho da direita
4	valor é menor do que 45: visita filho da esquerda
fim	Não existe filho da esquerda. Valor passa a ser o filho da esquerda de 45

Fonte: BACKES (2023).

5.3 Busca na ABB

- Consultar se um determinado nó V existe em uma ABB é similar a operação de inserção:
  1. Inicialmente, compare o valor buscado com a *raiz*;
  2. Se V é *menor que* a raiz: siga para a *subárvore da esquerda*;
  3. Se V é *maior que* a raiz: siga para a *subárvore da direita*;
  4. Aplique o método recursivamente até que a raiz seja igual ao valor buscado, ou `NULL`– também é possível sem recursão.

Exemplo de uso

```
int N = 8, dados[8] = {50,100,30,20,40,45,35,37};
ArvBin* raiz = cria_ArvBin();
int i;
for(i=0; i<N; i++)
  insere_ArvBin(raiz, dados[i]);
.
.
if (consulta_ArvBin(raiz, 20))
  printf("20 está na árvore\n");
```

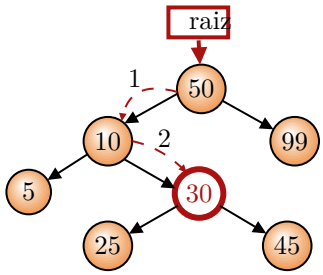
ArvoreBinaria.c: busca por nó

```
105 int consulta_ArvBin(ArvBin *raiz, int valor){
106   if (raiz == NULL)
```

```
    return 0;
  struct NO* atual = *raiz;
  while (atual != NULL){
    if (valor == atual->info){
      return 1;
    }
    if (valor > atual->info)
      atual = atual->dir;
    else
      atual = atual->esq;
  }
  return 0;
}
```

Passo-a-passo

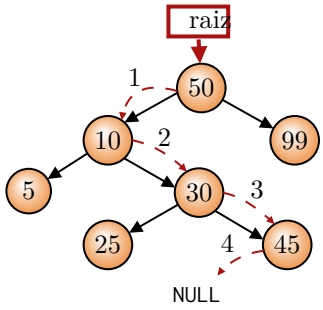
Valor procurado: 30



1	valor é menor do que 50: visita filho da esquerda
2	valor é maior do que 10: visita filho da direita
fim	valor procurado é igual ao do nó: retornar dados do nó (ou 1)

Fonte: BACKES (2023).

Valor procurado: 38



1	valor é menor do que 50: visita filho da esquerda
2	valor é maior do que 10: visita filho da direita
3	valor é maior do que 30: visita filho da direita
4	valor é menor do que 45: visita filho da esquerda
fim	filho da esquerda de 45 não existe: busca falhou

Fonte: BACKES (2023).

5.4 Remoção na ABB

- Remover um nó de uma árvore binária de busca não é uma tarefa tão simples quanto a inserção.
- Isso ocorre porque precisamos procurar o nó a ser removido da ABB, o qual pode ser um
  - Nó folha;
  - *Nó interno* – inclusive a raiz – com um ou dois filhos.
- Se for um nó interno:
  - Reorganizar a árvore para que ela continue sendo uma ABB.

Uso de duas funções:

- `remove_ArvBin()` – faz a busca pelo nó – disponível na interface com o usuário;
- `remove_atual()`, auxiliar, que trata os 3 tipos de remoção – com 0, 1 ou 2 filhos – reorganizando a ABB.
  - Recebe o endereço atual do nó a ser removido, e
  - retorna qual deverá ser seu substituto na ABB.

Exemplo de uso

```
int N = 8, dados[8] = {50,100,30,20,40,45,35,37};
ArvBin* raiz = cria_ArvBin();
int i;
for(i=0; i<N; i++)
  insere_ArvBin(raiz, dados[i]);
.
.
if (remove_ArvBin(raiz, 20))
  printf("20 foi excluido\n");
```

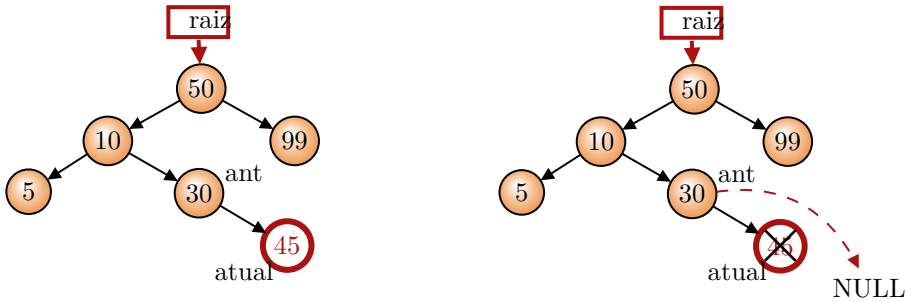
ArvoreBinaria.c: remoção de nó

```
121 struct NO* remove_atual(struct NO* atual) {
122   struct NO *no1, *no2;
123
124   // Sem filho da esquerda. Apontar para o filho da direita (
      trata nó folha e nó com 1 filho).
125   if (atual->esq == NULL){
126     no2 = atual->dir;
127     free(atual);
```

```
    return no2;
  }

  // Procura filho mais à direita na subárvore da esquerda.
  no1 = atual;
  no2 = atual->esq;
  while (no2->dir != NULL){
    no1 = no2;
    no2 = no2->dir;
  }

  // Copia o filho mais à direita na subárvore da esquerda
    para o lugar do nó removido.
  if (no1 != atual){
    no1->dir = no2->esq;
    no2->esq = atual->esq;
  }
  no2->dir = atual->dir;

  free(atual);
  return no2;
}

int remove_ArvBin(ArvBin *raiz, int valor){
  if (raiz == NULL)
    return 0;
  struct NO* ant = NULL;
  struct NO* atual = *raiz;
  while (atual != NULL){
    //Achou o nó a ser removido. Tratar o tipo de remoção.
    if (valor == atual->info){
      if (atual == *raiz)
        *raiz = remove_atual(atual);
      else{
        if (ant->dir == atual)
          ant->dir = remove_atual(atual);
        else
          ant->esq = remove_atual(atual);
      }
      return 1;
    }
    // Continua buscando na árvore nó a ser removido
    ant = atual;
    if (valor > atual->info)
      atual = atual->dir;
    else
      atual = atual->esq;
  }
  return 0;
}
```

Removendo um nó folha

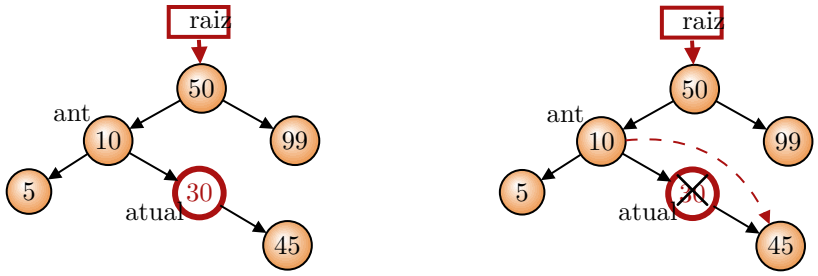
```
if (atual->esq == NULL){
  no2 = atual->dir;
  free(atual);
  return no2;
}
```



Fonte: BACKES (2023).

Removendo um nó sem filho à esquerda

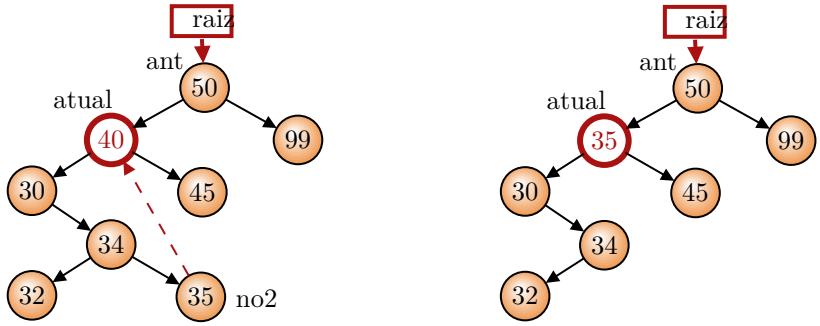
```
if (atual->esq == NULL){
  no2 = atual->dir;
  free(atual);
  return no2;
}
```



Fonte: BACKES (2023).

Removendo um nó com dois filhos

- Remove-se o nó "atual" e o substitui pelo nó mais à direita de sua subárvore da esquerda.



Fonte: BACKES (2023).

5.5 Exercícios

1. Implemente uma versão não-recursiva do método de inserção de nó em árvore binária de busca.
2. Escreva uma função que retorne se uma árvore é binária de busca ou não.
3. Escreva uma função que encontre o maior valor existente em uma árvore binária de busca.
4. Escreva uma função que encontre o menor valor existente em uma árvore binária de busca.
5. Desenhe a ABB resultante da inserção dos seguintes elementos (na ordem dada):
   (a) 60, 22, 64, 33, 67, 11, 75, 13, 8, 65.
   (b) 8, 11, 13, 22, 33, 60, 64, 65, 67, 75.

Referências

1. BACKES, A. R. **Algoritmos e Estruturas de Dados em Linguagem C**. 1. ed. Rio de Janeiro: LTC, 2023. Disponível em: https://www.sistemas.ufu.br/biblioteca-gateway/minhabiblioteca/9788521638315.
2. CORMEN, T. H.; et al. **Algoritmos**. 4. ed. Rio de Janeiro: GEN LTC, 2024. Disponível em: https://www.sistemas.ufu.br/biblioteca-gateway/minhabiblioteca/9788595159914.

Os materiais de parte desta seção foram gentilmente cedidos por André R. Backes (UFSCar); Bruno Travençolo e Christiane Brasil (FACOM/UFU).

TAD árvore binária de busca: disponível em: https://www.facom.ufu.br/~backes/wordpress/ProjArvoreBinaria.zip. Acesso em: 24/fev/2025.

Adaptações: Renato Pimentel, FACOM/UFU

6 Percursos: pré-ordem, em-ordem e pós-ordem

6.1 Introdução

- Percorrer todos os nós é uma operação comum em árvores binárias.
- Motivações: mostrar (imprimir) o valor do nó, modificar tal valor, etc.
  - Neste caso, cada nó é visitado uma *única vez*.
  - Isso gera uma sequência de nós, cuja ordem depende de como a árvore foi percorrida.
- Três formas mais comuns:
  1. Percurso *pré-ordem*: visita a raiz, a seguir o filho da esquerda, e depois o filho da direita;
  2. Percurso *em-ordem*: visita o filho da esquerda, a seguir a raiz e depois o filho da direita;
  3. Percurso *pós-ordem*: visita o filho da esquerda, a seguir o filho da direita e depois a raiz.

Árvore binária de busca
No caso do percurso *em-ordem*, se a árvore é ABB, a sequência resultante dos valores (chaves) visitados é *ordenada*.

No TAD Árvore binária:

ArvoreBinaria.h

```
typedef struct NO* ArvBin;

ArvBin* cria_ArvBin();
void libera_ArvBin(ArvBin *raiz);
int estaVazia_ArvBin(ArvBin *raiz);
int altura_ArvBin(ArvBin *raiz);
int totalNO_ArvBin(ArvBin *raiz);
int insere_ArvBin(ArvBin* raiz, int valor);
int consulta_ArvBin(ArvBin *raiz, int valor);
int remove_ArvBin(ArvBin *raiz, int valor);
void preOrdem_ArvBin(ArvBin *raiz);
void emOrdem_ArvBin(ArvBin *raiz);
void posOrdem_ArvBin(ArvBin *raiz);
```

6.2 Percurso pré-ordem

- Ordem de visitação da árvore/subárvores:
  1. Raiz;
  2. Filho/subárvore da esquerda;
  3. Filho/subárvore da direita.

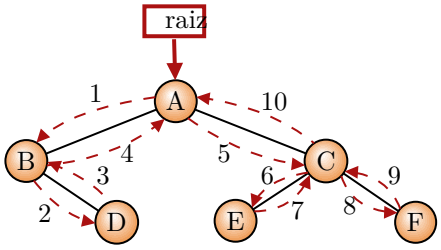
Exemplo de uso

```
int N = 8, dados[8] = {50,100,30,20,40,45,35,37};
ArvBin* raiz = cria_ArvBin();
int i;
for(i=0; i<N; i++)
  insere_ArvBin(raiz, dados[i]);
.
.
preOrdem_ArvBin(raiz);
```

ArvoreBinaria.c: percurso pré-ordem

```
178 void preOrdem_ArvBin(ArvBin *raiz){
179   if (raiz == NULL)
180     return;
181   if (*raiz != NULL){
182     printf("%d\n",(*raiz)->info);
183     preOrdem_ArvBin(&((*raiz)->esq));
184     preOrdem_ArvBin(&((*raiz)->dir));
185   }
186 }
```

Passo-a-passo



Resultado:
ABDCEF

	Inicia-se em A
1	imprime A, visita B
2	imprime B, visita D
3	imprime D, volta para B
4	volta para A
5	visita C
6	imprime C, visita E
7	imprime E, volta para C
8	visita F
9	imprime F, volta para C
10	volta para A

6.3 Percurso em-ordem

- Ordem de visitação da árvore/subárvores:
  1. Filho/subárvore da esquerda;
  2. Raiz;
  3. Filho/subárvore da direita.

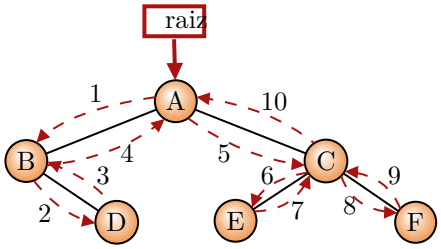
Exemplo de uso

```
int N = 8, dados[8] = {50,100,30,20,40,45,35,37};
ArvBin* raiz = cria_ArvBin();
int i;
for(i=0; i<N; i++)
  insere_ArvBin(raiz, dados[i]);
.
.
emOrdem_ArvBin(raiz);
```

ArvoreBinaria.c: percurso em-ordem

```
188 void emOrdem_ArvBin(ArvBin *raiz){
189   if (raiz == NULL)
190     return;
191   if (*raiz != NULL){
192     emOrdem_ArvBin(&((*raiz)->esq));
193     printf("%d\n",(*raiz)->info);
194     emOrdem_ArvBin(&((*raiz)->dir));
195   }
196 }
```

Passo-a-passo



Resultado:
BDAECF

	Inicia-se em A
1	visita B
2	imprime B, visita D
3	imprime D, volta para B
4	volta para A
5	imprime A, visita C
6	visita E
7	imprime E, volta para C
8	imprime C, visita F
9	imprime F, volta para C
10	volta para A

6.4 Percurso pós-ordem

- Ordem de visitação da árvore/subárvores:
  1. Filho/subárvore da esquerda;
  2. Filho/subárvore da direita;
  3. Raiz.

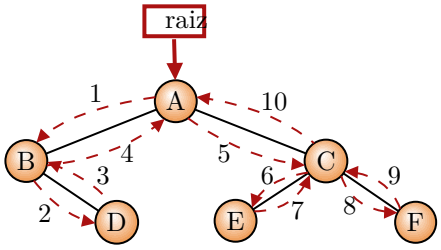
Exemplo de uso

```
int N = 8, dados[8] = {50,100,30,20,40,45,35,37};
ArvBin* raiz = cria_ArvBin();
int i;
for(i=0; i<N; i++)
  insere_ArvBin(raiz, dados[i]);
.
.
posOrdem_ArvBin(raiz);
```

ArvoreBinaria.c: percurso pós-ordem

```
198 void posOrdem_ArvBin(ArvBin *raiz){
199   if (raiz == NULL)
200     return;
201   if (*raiz != NULL){
202     posOrdem_ArvBin(&((*raiz)->esq));
203     posOrdem_ArvBin(&((*raiz)->dir));
204     printf("%d\n",(*raiz)->info);
205   }
206 }
```

Passo-a-passo



Resultado:
DBEFCA

	Inicia-se em A
1	visita B
2	visita D
3	imprime D, volta para B
4	imprime B, volta para A
5	visita C
6	visita E
7	imprime E, volta para C
8	visita F
9	imprime F, volta para C
10	imprime C, volta para A
11	imprime A

6.5 Exercícios

1. (CORMEN, 2024) Dê um algoritmo não recursivo que execute um percurso de árvore em ordem. (Sugestão: uma solução fácil usa uma pilha como estrutura de dados auxiliar. Uma solução mais complicada, porém elegante, não emprega nenhuma pilha, mas considera que é possível testar a igualdade entre dois ponteiros.)

2. Crie uma ABB a partir da inserção dos seguintes valores, nesta ordem:

   55, 42, 28, 74, 63, 12, 48, 54, 17, 9, 2, 99.

   Agora, liste os valores segundo os percursos pré-ordem, em-ordem e pós-ordem.

Referências

1. BACKES, A. R. **Algoritmos e Estruturas de Dados em Linguagem C**. 1. ed. Rio de Janeiro: LTC, 2023. Disponível em: https://www.sistemas.ufu.br/biblioteca-gateway/minhabiblioteca/9788521638315.

2. CORMEN, T. H.; et al. **Algoritmos**. 4. ed. Rio de Janeiro: GEN LTC, 2024. Disponível em: https://www.sistemas.ufu.br/biblioteca-gateway/minhabiblioteca/9788595159914.

Os materiais de parte desta seção foram gentilmente cedidos por André R. Backes (UFSCar); Bruno Travençolo e Christiane Brasil (FACOM/UFU).

TAD árvore binária de busca: disponível em: https://www.facom.ufu.br/~backes/wordpress/ProjArvoreBinaria.zip. Acesso em: 24/fev/2025.

Adaptações: Renato Pimentel, FACOM/UFU

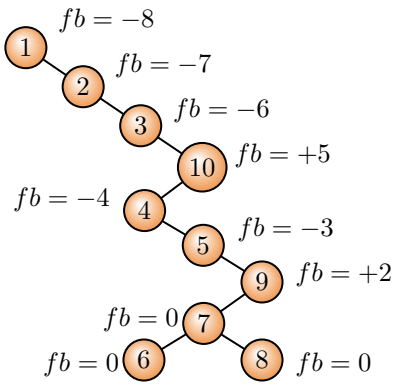
7 Balanceamento de ABB – Árvores AVL

7.1 Introdução

- A eficiência da busca em uma árvore binária é proporcional a sua *altura* e depende do seu *balanceamento*.
- Numa árvore com n nós:
  - *Melhor caso*: $O(\log_2 n)$, se a árvore está idealmente balanceada;
  - *Pior caso*: $O(n)$, se a árvore está totalmente desbalanceada.
- Os algoritmos de inserção e remoção em ABB vistos não garantem que a árvore gerada a cada passo esteja balanceada;
- Dependendo da ordem em que os dados são inseridos na árvore, podemos criar uma ABB bastante ou completamente desbalanceada – situação em que os nós internos possuem uma única subárvore.

Exemplo

Inserção dos valores:{1, 2, 3, 10, 4, 5, 9, 7, 8, 6}



Solução para o problema do balanceamento

- Modificar os algoritmos de inserção e remoção de modo a balancear a árvore a cada operação.
- Garantir que a *diferença de altura* das subárvores da esquerda e da direita de cada nó seja de no máximo uma unidade – *árvores quase completas*.
- Exemplos de implementação de árvores balanceadas:
  - Árvore AVL;
  - Árvore 2-3-4;
  - Árvore Rubro-Negra.

7.2 Árvore AVL

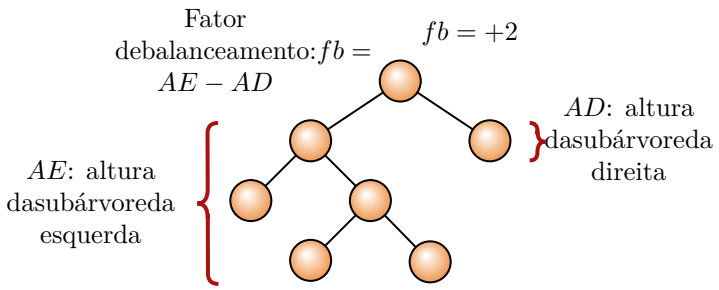
- Tipo de ABB balanceada com relação à altura das suas subárvores;
- Criada pelos matemáticos soviéticos Adelson-Velsky (1922-2014) e Evgenii Landis (1921-1997), de onde recebeu a sua nomenclatura (**A**delson-**V**elsky **L**andis), em 1962.
  - https://zhjwpku.com/assets/pdf/AED2-10-avl-paper.pdf

Aspectos

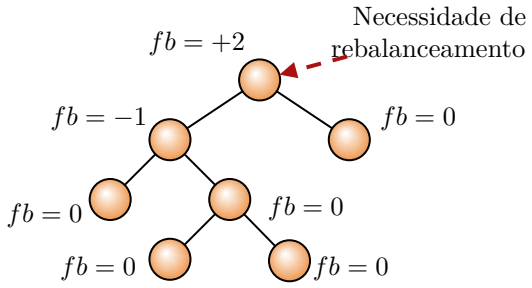
- Permite o rebalanceamento *local* da árvore – apenas a parte afetada pela inserção ou remoção é rebalanceada;
- Usa *rotações simples* ou *duplas* na etapa de rebalanceamento, executadas a cada inserção ou remoção;
- As rotações buscam manter a árvore como uma ABB quase completa;
  -
  - Custo máximo de qualquer algoritmo é $O(log_2 n)$

Fator de balanceamento

- Objetivo das rotações: corrigir o *fator de balanceamento* (fb).
  - $fb =$ Diferença entre as alturas das subárvores de um nó.
- Caso uma das subárvores não exista, então a altura dessa será igual a -1.

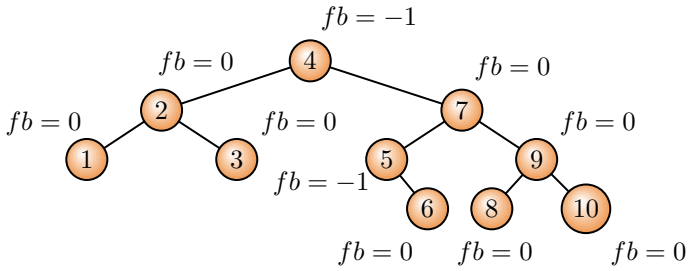


- Espera-se que as alturas das subárvores de cada nó difiram de no máximo uma unidade:
  - O fator de balanceamento deve ser $+1$, 0 ou -1.
- Se $fb > 1$ ou $fb < -1$: a árvore deve ser balanceada naquele nó.



Exemplo anterior – após balanceamento

Inserção dos
valores:{1, 2, 3, 10, 4, 5, 9, 7, 8, 6}



7.3 TAD árvore AVL

- Consideramos um tipo abstrato de dados *opaco* em Linguagem C para representação a árvore AVL, baseado na implementação vista para ABB.
- As considerações feitas com relação a tipo opaco, alocação dinâmica e acesso à estrutura a partir de sua raiz se mantém.
- Modularização do TAD:
  - *ArvoreAVL.h*: a *interface* em si: definição do tipo opaco; funções disponíveis para manipular a árvore AVL (protótipos).
  - *ArvoreAVL.c*: o que deve ficar oculto do usuário, além da implementação das funções definidas pela interface. Inclui a definição do *tipo nó*.
- *Tipo nó*: estrutura que contém:
  - `info`: Representa a *informação*, ou seja, os dados correspondentes ao elemento (nó/vértice) em questão;

– `altura`: altura da subárvore cujo nó é sua raiz.

– `*esq` e `*dir`: ponteiros para os nós filhos à esquerda e à direita do nó.

Interface: ArvoreAVL.h

ArvoreAVL.h

```
typedef struct NO* ArvAVL;

ArvAVL* cria_ArvAVL();
void libera_ArvAVL(ArvAVL *raiz);
int insere_ArvAVL(ArvAVL *raiz, int data);
int remove_ArvAVL(ArvAVL *raiz, int valor);
int estaVazia_ArvAVL(ArvAVL *raiz);
int altura_ArvAVL(ArvAVL *raiz);
int totalNO_ArvAVL(ArvAVL *raiz);
int consulta_ArvAVL(ArvAVL *raiz, int valor);
void preOrdem_ArvAVL(ArvAVL *raiz);
void emOrdem_ArvAVL(ArvAVL *raiz);
void posOrdem_ArvAVL(ArvAVL *raiz);
```

Definição do nó: ArvoreAVL.c

ArvoreAVL.c: bibliotecas usadas e definição do tipo

```
#include <stdio.h>
#include <stdlib.h>
#include "ArvoreAVL.h" //inclui os Protótipos

struct NO
{
  int info;
  int altura;
  struct NO *esq;
  struct NO *dir;
};
```

Cálculo da altura e fb

- As funções abaixo são auxiliares, e não estão visíveis ao usuário do TAD – portanto, não são listadas em ArvoreAVL.h;
- A função que calcula o fb retorna seu módulo, mas isto não interfere nas operações de inserção e remoção a serem mostradas adiante.
- A função `maior()` será usada para o comparativo das alturas das subárvores da esquerda e da direita de um nó.

ArvoreAVL.c

```c
int altura_NO(struct NO* no)
{
  if(no == NULL)
    return -1;
  else
    return no->altura;
}

int fatorBalanceamento_NO(struct NO* no)
{
  return labs(altura_NO(no->esq) - altura_NO(no->dir));
}

int maior(int x, int y)
{
  if(x > y)
    return x;
  else
    return y;
}
```

7.4 Rotações

Objetivo
Corrigir o fator de balanceamento (fb) de cada nó:
Operação básica para balancear uma árvore AVL.

- Existem dois tipos de rotação:
  - Rotação simples;
  - Rotação dupla.
- As rotações diferem entre si pelo sentido da inclinação entre o nó pai e filho.
- *Rotação simples*: o nó desbalanceado (pai), seu filho e seu neto estão todos no mesmo sentido de inclinação.
- *Rotação dupla*: o nó desbalanceado (pai) e seu filho estão inclinados em sentido inverso ao de seu neto
  - Equivale a duas rotações simples.

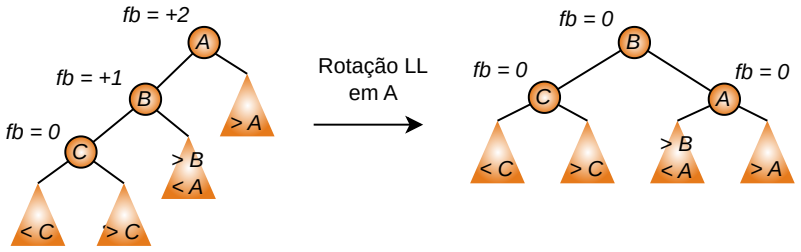
Tipos de rotações

- Rotação *simples à direita* ou *rotação LL*;
- Rotação *simples à esquerda* ou *rotação RR*;
- Rotação *dupla à direita* ou *rotação LR*;
- Rotação *dupla à esquerda* ou *Rotação RL*.
- Rotações são aplicadas no ancestral mais próximo do nó inserido cujo fator de balanceamento passa a ser $+2$ ou -2;
  - Após uma inserção ou remoção, deve-se voltar pelo caminho feito e recalcular o fb de cada nó;
  - Para o nó onde $|fb| = 2$, uma rotação deverá ser aplicada.

7.4.1 Rotação LL (simples à direita)

- Quando um novo nó é inserido na subárvore da esquerda (*left*) do filho da esquerda (*left*), B, de A – daí o LL.
  - A é o nó desbalanceado.
- É necessário fazer uma *rotação à direita*, de modo que o nó intermediário B ocupe o lugar de A, e A se torne filho da direita de B.

Exemplo



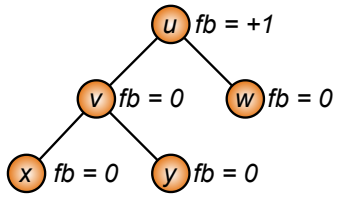
No TAD árvore AVL

ArvoreAVL.c

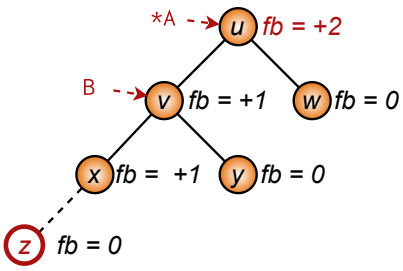
```
153 void RotacaoLL(ArvAVL *A)
154 {
155   struct NO *B;
156   B = (*A)->esq;
157   (*A)->esq = B->dir;
```

```
158   B->dir = *A;
159   (*A)->altura = maior(altura_NO((*A)->esq),altura_NO((*A)->
        dir)) + 1;
160   B->altura = maior(altura_NO(B->esq),(*A)->altura) + 1;
161   *A = B;
162 }
```

Passo a passo

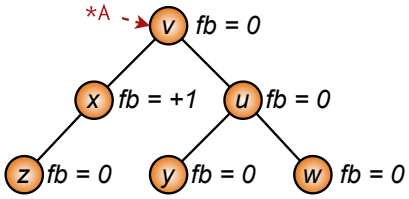


Árvore AVL e fator de balanceamento de cada nó.



Inserção de nó na árvore:
Árvore fica desbalanceada em `A`.
Aplica-se Rotação LL em `A`.

```
B = (*A)->esq;
(*A)->esq = B->dir;
B->dir = *A;
*A = B;
```



Árvore balanceada.

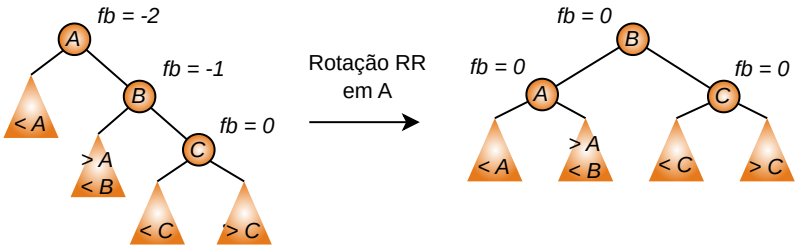
7.4.2 Rotação RR (simples à esquerda)

- Quando um novo nó é inserido na subárvore da direita (*right*) do filho da direita (*right*), B, de A – daí o RR.

– A é o nó desbalanceado.

- É necessário fazer uma *rotação à esquerda*, de modo que o nó intermediário B ocupe o lugar de A, e A se torne filho da esquerda de B.

Exemplo

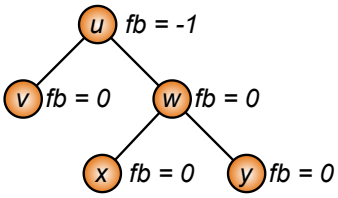


No TAD árvore AVL

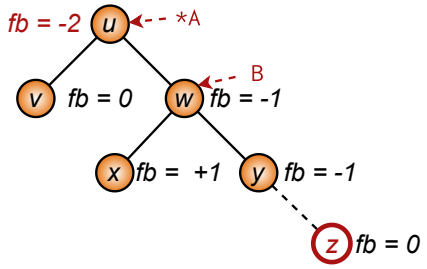
ArvoreAVL.c

```
void RotacaoRR(ArvAVL *A)
{
  struct NO *B;
  B = (*A)->dir;
  (*A)->dir = B->esq;
  B->esq = (*A);
  (*A)->altura = maior(altura_NO((*A)->esq),altura_NO((*A)->
    dir)) + 1;
  B->altura = maior(altura_NO(B->dir),(*A)->altura) + 1;
  (*A) = B;
}
```

Passo a passo

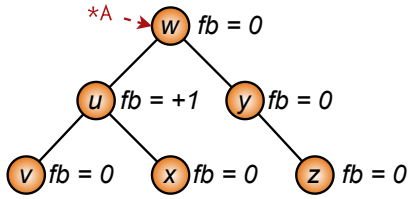


Árvore AVL e fator de balanceamento de cada nó.



Inserção de nó na árvore:
Árvore fica desbalanceada em `A`.
Aplica-se rotação RR em `A`.

```
B = (*A)->dir;
(*A)->dir = B->esq;
B->esq = (*A);
(*A) = B;
```

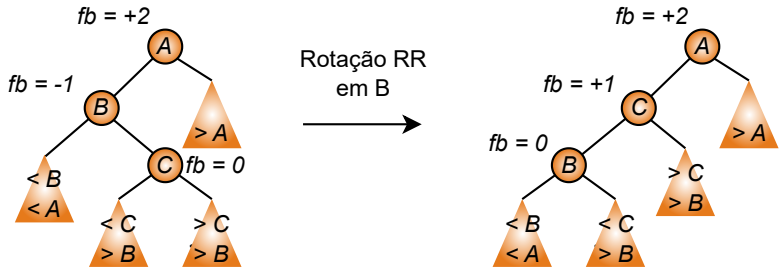


Árvore balanceada.

7.4.3 Rotação LR (dupla à direita)

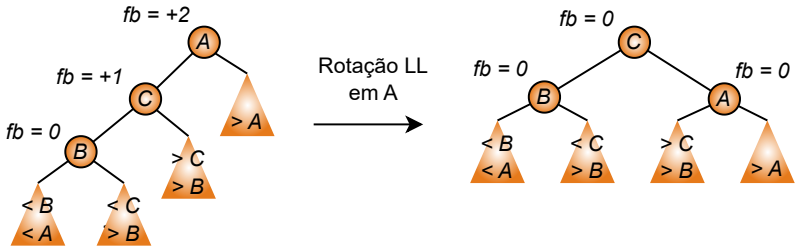
- Quando um novo nó é inserido na subárvore da direita (*right*) do filho da esquerda (*left*), B, de A – daí o LR.
  - A é o nó desbalanceado.
- É necessário fazer uma rotação dupla, de modo que o nó C – raiz da subárvore da direita de B – se torne o pai dos nós A (à sua direita) e B (à sua esquerda).
- Equivalente a duas rotações simples:
  1. Rotação RR em B;
  2. Rotação LL em A.

Exemplo: primeira rotação




```
RotacaoRR(&(*A)->esq);
RotacaoLL(A);
```

Exemplo: segunda rotação



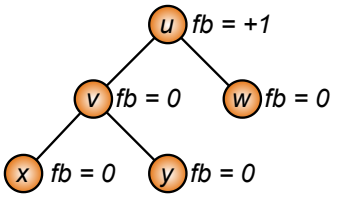

```
RotacaoRR(&(*A)->esq);
RotacaoLL(A);
```

No TAD árvore AVL

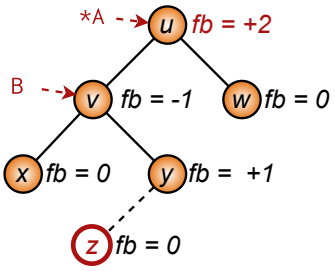
ArvoreAVL.c

```
175 void RotacaoLR(ArvAVL *A)
176 {
177   RotacaoRR(&(*A)->esq);
178   RotacaoLL(A);
179 }
```

Passo a passo

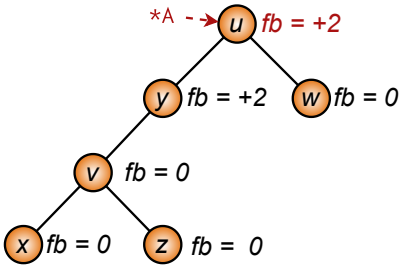


Árvore AVL e fator de balanceamento de cada nó.

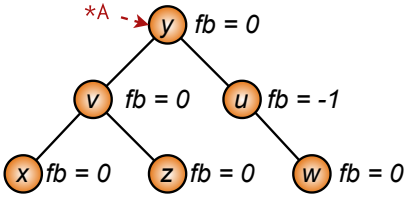


Inserção de nó na árvore:
Árvore fica desbalanceada em `A`.
Aplica-se Rotação LR em `A`. Isto equivale a:

- Aplicar Rotação RR em `B`;
- Aplicar Rotação LL em `A`.



Árvore após Rotação RR em `B`.

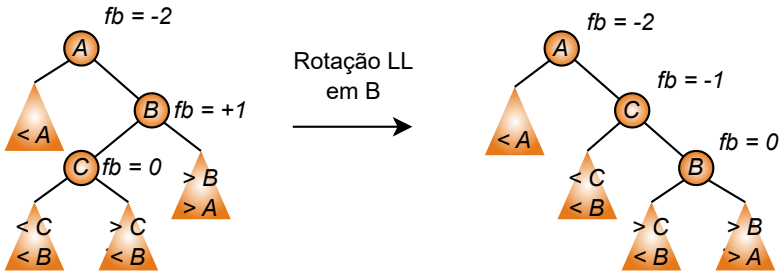


Árvore balanceada.

7.4.4 Rotação RL (dupla à esquerda)

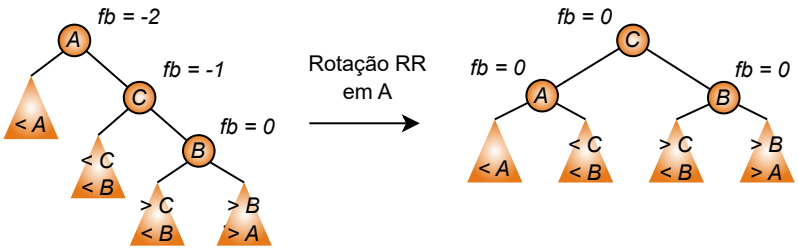
- Quando um novo nó é inserido na subárvore da esquerda (*left*) do filho da direita (*right*), B, de A – daí o RL.
  - A é o nó desbalanceado.
- É necessário fazer uma rotação dupla, de modo que o nó C – raiz da subárvore da esquerda de B – se torne o pai dos nós A (à sua esquerda) e B (à sua direita).
- Equivalente a duas rotações simples:
  1. Rotação LL em B;
  2. Rotação RR em A.

Exemplo: primeira rotação




```
RotacaoLL(&(*A)->dir);
RotacaoRR(A);
```

Exemplo: segunda rotação



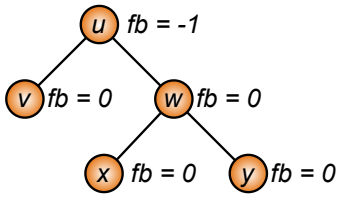
```
RotacaoLL(&(*A)->dir);
RotacaoRR(A);
```

No TAD árvore AVL

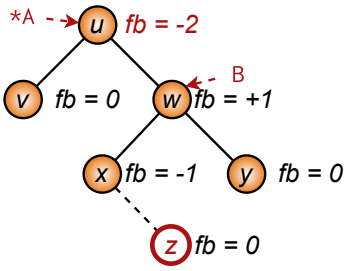
ArvoreAVL.c

```
181 void RotacaoRL(ArvAVL *A)
182 {
183   RotacaoLL(&(*A)->dir);
184   RotacaoRR(A);
185 }
```

Passo a passo

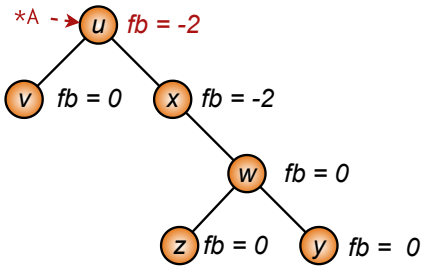


Árvore AVL e fator de balanceamento de cada nó.

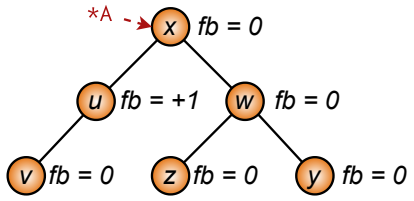


Inserção de nó na árvore:
Árvore fica desbalanceada em A.
Aplica-se Rotação RL em A. Isto equivale a:

- Aplicar Rotação LL em B;
- Aplicar Rotação RR em A.



Árvore após Rotação LL em B.



Árvore balanceada.

7.4.5 Uso

Quando usar cada tipo de rotação?

fb **de** A	fb **de** B	**Posições de** B **e** C **em relação a** A	**Rotação**
2	1	B é filho à esquerda de A; C é filho à esquerda de B.	LL
-2	-1	B é filho à direita de A; C é filho à direita de B.	RR
2	-1	B é filho à esquerda de A; C é filho à direita de B.	LR
-2	1	B é filho à de direita A; C é filho à esquerda de B.	RL

- *Sinais iguais*: rotações simples – $+$: à direita; $-$: à esquerda;
- *Sinais diferentes*: rotações duplas – $fb(A) > 0$: à direita (LR); $fb(A) < 0$: à esquerda (RL).

7.5 Inserção de nó na AVL

- Para inserir um valor V na AVL:
  1. Se a raiz é igual a `NULL`, insira o nó;
  2. Se V é *menor que* a raiz: siga para a *subárvore da esquerda*;
  3. Se V é *maior que* a raiz: siga para a *subárvore da direita*;
  4. Aplique o método recursivamente – também é possível sem recursão.
- Percorre-se os nós da AVL até o *nó folha* que se tornará *pai* do novo nó.
- Após cada inserção de um nó na AVL:
  - Volta-se pelo caminho percorrido, recalculando o fator de balanceamento de cada um dos nós visitados;

– Aplica-se a rotação necessária em um nó onde $|fb| > 1$, restabelecendo o balanceamento da árvore.

main.c: Exemplo de uso

```
int res,i;
int N = 10, dados[10] = {1,2,3,10,4,5,9,7,8,6};
ArvAVL* avl = cria_ArvAVL();
for(i=0; i<N; i++)
  res = insere_ArvAVL(avl,dados[i]);
```

ArvoreAVL.c: inserção de nó

```
187 int insere_ArvAVL(ArvAVL *raiz, int valor) {
188   if (raiz == NULL)
189     return 0;
190
191   int res;
192   if (*raiz == NULL) { // árvore vazia ou nó folha
193     struct NO *novo;
194     novo = (struct NO*)malloc(sizeof(struct NO));
195     if (novo == NULL)
196       return 0;
197
198     novo->info = valor;
199     novo->altura = 0;
200     novo->esq = NULL;
201     novo->dir = NULL;
202     *raiz = novo;
203     return 1;
204   }
205
206   struct NO *atual = *raiz;
207   if (valor < atual->info){
208     if ((res = insere_ArvAVL(&(atual->esq), valor)) == 1) {
209       if (fatorBalanceamento_NO(atual) >= 2) {
210         if (valor < (*raiz)->esq->info)
211           RotacaoLL(raiz); // inserção na subárvore da
     esquerda do filho da esquerda de atual
212         else
213           RotacaoLR(raiz); // inserção na subárvore da
     direita do filho da esquerda de atual
214       }
215     }
216   } else {
217     if (valor > atual->info) {
218       if ((res = insere_ArvAVL(&(atual->dir), valor)) == 1){
219         if (fatorBalanceamento_NO(atual) >= 2){
220           if ((*raiz)->dir->info < valor)
```

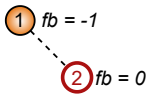
```
            RotacaoRR(raiz); // inserção na subárvore da direita do filho da direita de atual
          else
            RotacaoRL(raiz); // inserção na subárvore da esquerda do filho da direita de atual
        }
      }
    } else {
      printf("Valor duplicado!!\n");
      return 0;
    }
  }
  atual->altura = maior(altura_NO(atual->esq),altura_NO(atual->dir)) + 1;
  return res;
}
```

Passo a passo

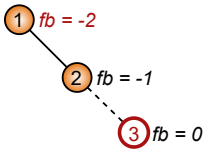
Inserção dos valores:
$\{1,\ 2,\ 3,\ 10,\ 4,\ 5,\ 9,\ 7,\ 8,\ 6\}$

Insere valor: 1

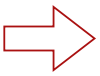
Insere valor: 2

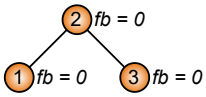


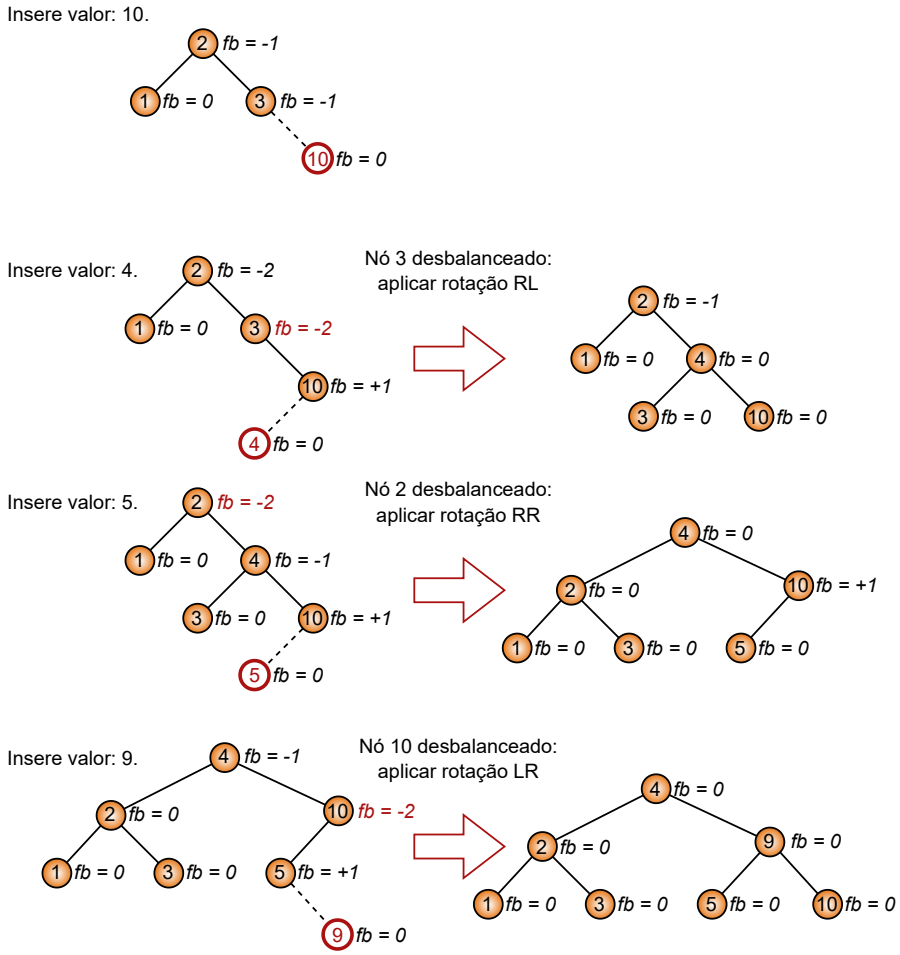
Insere valor: 3.



Nó 1 desbalanceado:
aplicar rotação RR







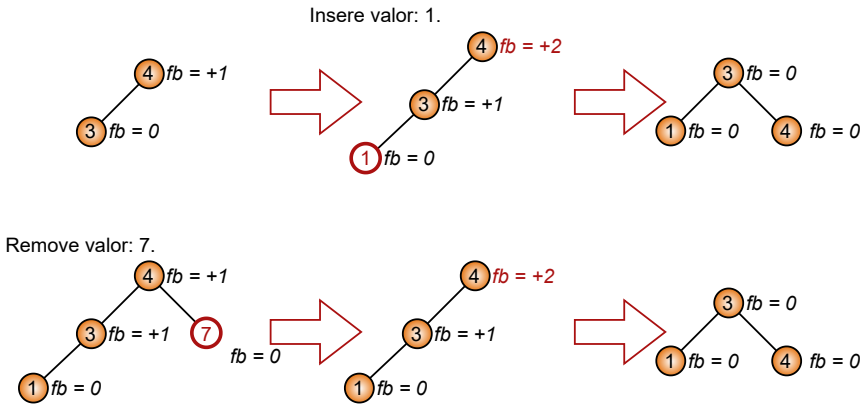
Exercício: inserir na AVL, mostrando o passo a passo, os valores faltantes: 7, 8 e 6.

7.6 Remoção de nó na AVL

- Percorre-se um conjunto de nós da árvore até chegar ao nó que será removido.
- Existem 3 tipos de remoção:
  1. Nó folha (sem filhos);
  2. Nó com um filho;
  3. Nó com dois filhos.

- Após cada remoção de um nó na AVL:
  - Volta-se pelo caminho percorrido, recalculando o fator de balanceamento de cada um dos nós visitados;
  - Aplica-se a rotação necessária em um nó onde $|fb| > 1$, restabelecendo o balanceamento da árvore.

Remover um nó da subárvore da *direita* equivale a *inserir* um nó na subárvore da *esquerda* (e vice-versa).



Uso de duas funções:

- `remove_ArvAVL()` – faz a busca pelo nó e sua remoção, além do rebalanceamento, quando necessário – disponível na interface com o usuário;
- `procuraMenor()`, auxiliar, procura o nó mais à esquerda da subárvore da esquerda do nó atual.

main.c: Exemplo de uso

```
int res,i;
int N = 10, dados[10] = {1,2,3,10,4,5,9,7,8,6};
ArvAVL* avl = cria_ArvAVL();
for(i=0; i<N; i++)
  res = insere_ArvAVL(avl,dados[i]);
.
.
if (remove_ArvAVL(raiz, 20))
  printf("20 foi excluido\n");
```

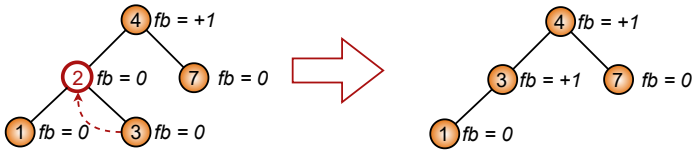
ArvoreAVL.c: remoção de nó

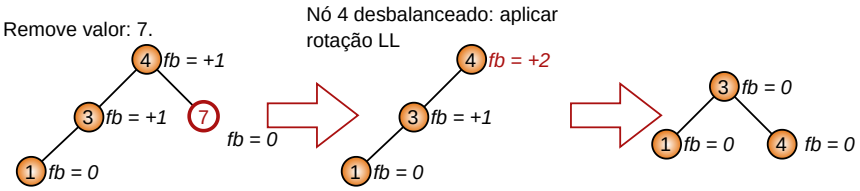
```
235 struct NO* procuraMenor(struct NO* atual) {
236   struct NO *no1 = atual;
237   struct NO *no2 = atual->esq;
238   while (no2 != NULL) {
239     no1 = no2;
240     no2 = no2->esq;
241   }
242   return no1;
243 }
244
245 int remove_ArvAVL(ArvAVL *raiz, int valor) {
246   if (raiz == NULL)
247     return 0;
248
249   int res;
250   if (*raiz == NULL) { // valor não existe
251     printf("valor não existe!!\n");
252     return 0;
253   }
254
255   if (valor < (*raiz)->info) {
256     if ((res = remove_ArvAVL(&(*raiz)->esq,valor)) == 1) { //
      removeu nó da subárvore da esquerda
257       if (fatorBalanceamento_NO(*raiz) >= 2) { // subárvore
      da direita muito mais alta
258         if (altura_NO((*raiz)->dir->esq) <= altura_NO((*raiz)
      ->dir->dir))
259           RotacaoRR(raiz); // remoção correspondeu a inserção
      na subárvore da direita do filho da direita de atual
260         else
261           RotacaoRL(raiz); // remoção correspondeu a inserção
      na subárvore da esquerda do filho da direita de atual
262       }
263     }
264   }
265   if ((*raiz)->info < valor) {
266     if ((res = remove_ArvAVL(&(*raiz)->dir, valor)) == 1) {
      // removeu nó da subárvore da direita
267       if(fatorBalanceamento_NO(*raiz) >= 2) { // subárvore da
      esquerda muito mais alta
268         if(altura_NO((*raiz)->esq->dir) <= altura_NO((*raiz)
      ->esq->esq) )
269           RotacaoLL(raiz); // remoção correspondeu a inserção
      na subárvore da esquerda do filho da esquerda de atual
270         else
271           RotacaoLR(raiz); // remoção correspondeu a inserção
      na subárvore da direita do filho da esquerda de atual
272       }
```

```
    }
  }
  if ((*raiz)->info == valor) {
    if (((*raiz)->esq == NULL || (*raiz)->dir == NULL)) {// nó tem somente um ou nenhum filho
      struct NO *oldNode = (*raiz);
      if ((*raiz)->esq != NULL)
        *raiz = (*raiz)->esq;
      else
        *raiz = (*raiz)->dir;
      free(oldNode);
    } else {  // nó tem dois filhos: substitui-o pelo nó mais à esquerda da subárvore da direita
      struct NO* temp = procuraMenor((*raiz)->dir);
      (*raiz)->info = temp->info;
      remove_ArvAVL(&(*raiz)->dir, (*raiz)->info);
      if (fatorBalanceamento_NO(*raiz) >= 2) {
        if(altura_NO((*raiz)->esq->dir) <= altura_NO((*raiz)->esq->esq))
          RotacaoLL(raiz);
        else
          RotacaoLR(raiz);
      }
    }
    if (*raiz != NULL)
      (*raiz)->altura = maior(altura_NO((*raiz)->esq), altura_NO((*raiz)->dir)) + 1; // atualiza a altura da raiz
    return 1;
  }
  (*raiz)->altura = maior(altura_NO((*raiz)->esq),altura_NO((*raiz)->dir)) + 1; // atualiza a altura da raiz
  return res;
}
```

Passo a passo

Remove valor: 2.





7.7 Exercícios

1. Crie uma árvore AVL a partir da inserção dos seguintes valores, nesta ordem, mostrando o passo a passo a cada inserção:

   55, 42, 28, 74, 63, 12, 48, 54, 17, 9, 2, 99.

2. Agora, remova os nós da árvore, um a um, na ordem em que foram inseridos.

Referências

1. BACKES, A. R. **Algoritmos e Estruturas de Dados em Linguagem C**. 1. ed. Rio de Janeiro: LTC, 2023. Disponível em: https://www.sistemas.ufu.br/biblioteca-gateway/minhabiblioteca/9788521638315.

2. CORMEN, T. H.; et al. **Algoritmos**. 4. ed. Rio de Janeiro: GEN LTC, 2024. Disponível em: https://www.sistemas.ufu.br/biblioteca-gateway/minhabiblioteca/9788595159914.

Os materiais de parte desta seção foram gentilmente cedidos por André R. Backes (UFSCar); Bruno Travençolo e Christiane Brasil (FACOM/UFU).

TAD árvore AVL: disponível em: https://www.facom.ufu.br/~backes/wordpress/ProjArvoreAVL.zip. Acesso em: 18/mar/2025.

Adaptações: Renato Pimentel, FACOM/UFU

8 Tabelas *hash*

8.1 Definições básicas e aplicações

Motivação: busca

- Princípio de funcionamento dos métodos de busca:
  - Procurar a informação desejada com base na comparação de suas *chaves*, i.e. com base em algum valor que a compõe.
- *Problema*: algoritmos eficientes necessitam que os elementos estejam armazenados de forma *ordenada*:
  - Custo ordenação melhor caso é $O(n \log n)$;
  - Custo da busca melhor caso é $O(\log n)$.
    - Árvores binárias balanceadas: $O(\log_2 n)$.

Busca ideal: $O(1)$

- Custo da comparação de chaves é alto.
- O que seria uma operação de busca ideal?
  - Seria aquela que permitisse o acesso direto ao elemento procurado, sem *nenhuma* etapa de comparação de chaves.
  - Nesse caso, teríamos um custo $O(1)$ – tempo sempre *constante* de acesso.
- Uma saída é usar *arrays*:
  - Estruturas que utilizam *índices* para armazenar informações;
  - Permitem acesso a qualquer posição com custo $O(1)$.

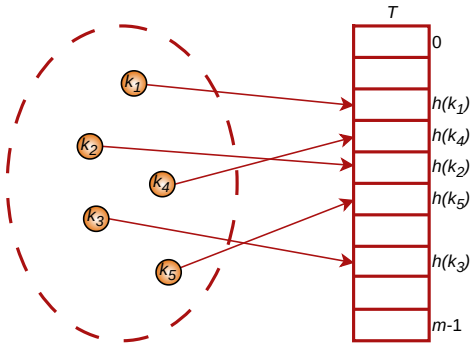
Problema
Arrays não possuem nenhum mecanismo que permita calcular a posição onde uma informação está armazenada – a operação de busca *não é* $O(1)$.

- Precisamos do tempo de acesso do *array*, em conjunto com a capacidade de busca qualquer elemento em tempo constante.
- *Solução*: usar uma tabela *hash*.

Tabelas *hash*

- Também conhecidas como *tabelas de indexação* ou *de espalhamento*[3].
  - São generalizações do conceito de *array*.
  - EDs especializadas nas *operações de dicionário* (busca, inserção e remoção) de maneira eficiente.
- Ideia central:
  - Utilizar uma função, chamada de *função de hashing*, para *espalhar* os elementos que se deseja armazenar na tabela.
  - Tal espalhamento faz com que os elementos fiquem dispersos de forma não ordenada dentro do *array* que define a tabela.

Exemplo



Benefícios do espalhamento

- A tabela permite a associar valores a chaves:
  - *Chave*: parte da informação que compõe o elemento a ser inserido ou buscado na tabela.
    - Ex.: Num cadastro de alunos, a *matrícula* pode ser chave – dois alunos não tem a mesma matrícula.
    - As demais informações de cada aluno, como nome, etc., são os *dados satélites*.
  - *Valor*: posição (índice) onde o elemento se encontra no *array* que define a tabela.

[3] Em inglês: *hashing*

- Dadas n chaves, e uma tabela de tamanho m – com $m = \Theta(n)$ – as posições da tabela se situam no intervalo $[0, m-1]$: cada posição pode armazenar um ou mais elementos.
- Logo, a partir de uma chave pode-se acessar de forma rápida – em geral, $O(1)$ – uma determinada posição do *array*.

Vantagens

- Alta eficiência na operação de busca
  - Caso médio é $O(1)$, enquanto o da busca linear é $O(n)$ e da busca binária é $O(log_2 n)$.
- Tempo de busca é praticamente independente do número de chaves armazenadas na tabela.
- Implementação simples.

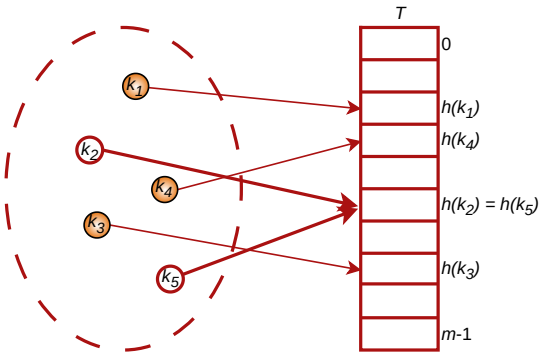
Desvantagens

- Alto custo para recuperar os elementos da tabela ordenados pela chave.
  - Nesse caso, é preciso ordenar a tabela;
  - O pior caso é $O(m)$, sendo m o tamanho da tabela.
- *Colisões*.

Colisão
Quando duas (ou mais) chaves diferentes tentam ocupar a mesma posição na tabela *hash*.

- Inevitável, mas indesejável, pois diminui o desempenho do sistema.

Exemplo de colisão



Principais aplicações

- Busca de elementos em bases de dados. Ex.: estruturas de dados em memória, bancos de dados e mecanismos de busca na Internet;
- verificação de integridade de dados e autenticação de mensagens;
  - os dados são enviados juntamente com o resultado da função de *hashing*;
  - O receptor dos dados recalcula a função de *hashing* usando os dados recebidos, e compara o resultado obtido com o que ele recebeu;
  - Resultados diferentes: *erro* de transmissão.
- Tabelas de símbolos em compiladores;
- Compressão de dados – para evitar armazenamentos duplicados;
- NAT (*Network Address Translation*) em redes de computadores;
- Criptografia – armazenamento de senhas com segurança.
  - MD5 (*Message-Digest Algorithm* 5);
  - SHA-1 (*Secure Hash Algorithm*).

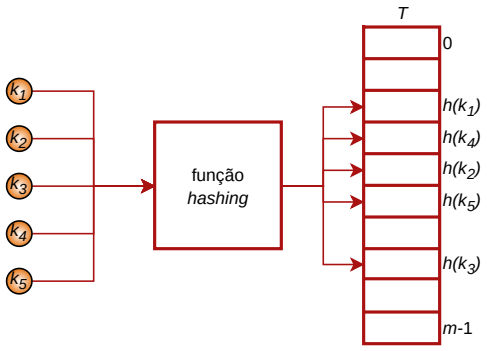
8.2 Funções de *hashing*

- Operações de dicionário: necessário calcular a *posição* (valor) dos dados dentro da tabela.
- A *função de hashing* h calcula a posição $h(k)$, $0 \leq h(k) < m$, para uma determinada chave k a partir dos dados.
  - Extremamente importante para o bom desempenho da tabela.
  - Responsável por distribuir as informações de forma equilibrada na ED – *espalhamento*.

Requisitos

- Idealmente fácil de computar e *uniforme* (distribuição equilibrada dos dados).
  - Cada posição da tabela tem a mesma chance de receber uma chave (máximo espalhamento).
- O ideal é que a função forneça índices únicos para o conjunto das chaves de entrada possíveis;
- Eventualmente, duas chaves diferentes podem ser mapeadas em valores iguais (colisão): $x \neq y$ tal que $h(x) = h(y)$.
- A implementação depende do conhecimento prévio da natureza e domínio da chave a ser utilizada.
- *Exemplo*: utilizar apenas três dígitos do número de telefone de uma pessoa para armazená-lo na tabela.
  - Neste caso, seria melhor usar os três últimos dígitos ao invés dos três primeiros, pois estes costumam se repetir com maior frequência, gerando posições iguais na tabela.
- Assim, o ideal é usar um cálculo diferente de espalhamento para cada tipo de chave.

Esquema de funcionamento



Funções de *hashing* comumente utilizadas

- Método da divisão;
- Método da multiplicação;
- Método da dobra.

8.2.1 Método da divisão

- Também conhecido como *método da congruência linear*.
- Consiste em calcular o *resto da divisão* do valor inteiro que representa o elemento pelo tamanho da tabela, m:

$$h(k) = k \bmod m\,.$$

- Simples e direta;
- *Implementação em C*: A *operação de* `E` *bit-a-bit* (`&`) com o valor `0x7FFFFFFF` elimina o *bit de sinal* e evita o risco de ocorrer um *overflow* e consequente obtenção de um número negativo.

TabelaHash.c: método da divisão

```
52 int chaveDivisao(int chave, int TABLE_SIZE) {
53   return (chave & 0x7FFFFFFF) % TABLE_SIZE;
54 }
```

Limitações

- *Resto da divisão*: valores diferentes podem resultar na mesma posição.
  - Ex.: O resto da divisão de 11 por 10 e de 21 por 10 são o mesmo valor de posição: 1
- Uma maneira de reduzir esse tipo de problema é utilizar como tamanho da tabela, m, um *número primo*.

8.2.2 Método da multiplicação

- Também conhecido como *método da congruência linear multiplicativo*.
- Usa uma *constante fracionária* A, $0 < A < 1$, para multiplicar o valor da chave que representa o elemento;
- Em seguida, a *parte fracionária* resultante é multiplicada pelo tamanho da tabela para calcular a posição do elemento:

$$h(k) = \lfloor m \cdot [(k \cdot A) \bmod 1] \rfloor$$

TabelaHash.c: método da multiplicação

```
71 int chaveMultiplicacao(int chave, int TABLE_SIZE) {
72   double A = 0.6180339887; // constante: 0 < A < 1
73   double val = chave * A;
74   val = val - (int) val;
75   return (int) (TABLE_SIZE * val);
76 }
```

Aspectos

- Escolha de m não é mais crítica;
- Porém, escolha de A depende dos dados.
  - Knuth sugere o negativo do *conjugado da razão áurea*:

$$A = -\frac{1-\sqrt{5}}{2} \approx 0,6180339887$$

Exemplo – com tabela de tamanho $m = 1024$

- Chave $k = 2112$ e $A = 0,8660254038$:
- Multiplicação: $k \cdot A = 1829,0456528256$;
- Parte fracionária: $(k \cdot A) \bmod 1 = 0,0456528256$;
- Produto da parte fracionária por m: $m \cdot [(k \cdot A) \bmod 1] = 1024 \cdot 0,0456528256 = 46,7484934144$;
- A parte inteira (piso) é a *posição* na tabela: $h(k) = \lfloor 46,74\ldots \rfloor = 46$.
- Repetindo-se o exemplo anterior para $A = 0,6180339887$ (Knuth): $h(k) = 294$.

8.2.3 Método da dobra

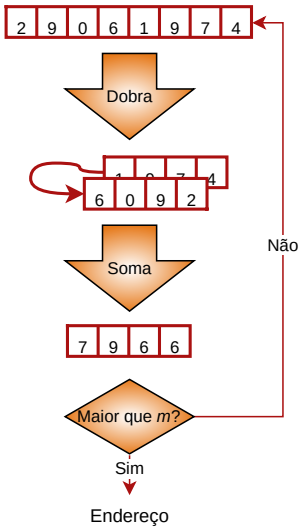
- Utiliza um esquema de *dobrar e somar* os dígitos da chave para calcular a sua posição.
- Considera o valor inteiro que representa o elemento como uma sequência de dígitos escritos num pedaço de papel.
  - Enquanto esse valor for maior que o tamanho da tabela, o papel é dobrado e os dígitos sobrepostos são somados, *desconsiderando-se as dezenas*.
- O processo se repete até que os dígitos formem um número menor que m, o tamanho da tabela.

Exemplo – com tabela de tamanho $m = 64$

$k = 29061974$

$$\begin{array}{r} 1974 \\ \underline{6092} \\ 7966 \end{array}$$

$$\begin{array}{r} 66 \\ \underline{97} \\ \mathbf{53} \end{array}$$



Implementação com operações bit-a-bit

- O método da dobra pode ser usado com *operações bit-a-bit*:
  - Utiliza a operação de *OU exclusivo* (XOR);
  - Não se usa as operações de *E* e *OU* bit-a-bit, pois estas produzem resultados menores e maiores, respectivamente, que os operandos envolvidos.

TabelaHash.c: método da dobra

```
int chaveDobra(int chave, int TABLE_SIZE) {
  int num_bits = 10;
  int parte1 = chave >> num_bits;
  int parte2 = chave & (TABLE_SIZE-1);
  return (parte1 ^ parte2);
}
```

- A dobra neste caso pode ser realizada de b em b bits, o que resulta em $f(k) \in [0, 2^b)$.
- Ex.: queremos calcular a posição do valor 71 (0001000111 em binário), usando $b = 5$:

$$\begin{aligned} \text{Posição} &= 00010 \text{ “OU exclusivo” } 00111 \\ &= 00101 \\ &= \mathbf{5}\,. \end{aligned}$$

8.2.4 Cadeias de caracteres como chaves

- Quando a chave é *alfanumérica*, i.e. uma *string*, pode-se optar por calcular um valor numérico a partir de seu conteúdo:
  - Esse valor pode ser facilmente calculado, por exemplo, somando-se os valores ASCII dos caracteres que compõem a *string* – porém, isto não é bom.
- O resultado pode então ser usado como parâmetro numa função de *hashing*.

TabelaHash.c: String como chave

```
41 int valorString(char *str) {
42   int i, valor = 7;
43   int tam = strlen(str);
44   for (i=0; i < tam; i++)
45     valor = 31 * valor + (int) str[i];
46   return (valor & 0x7FFFFFFF);
47 }
```

- Por que não devemos simplesmente somar os valores ASCII dos caracteres da *string*?
  - Porque palavras que correspondem a permutações uma da outra produzirão o mesmo valor e, consequentemente, uma *colisão*.
  - Exemplo: *rato* e *ator*.

$$\begin{aligned} \text{rato: } & 114 + 97 + 116 + 111 = 438 \\ \text{ator: } & 97 + 116 + 111 + 114 = 438\,. \end{aligned}$$

- A posição da letra *deve* ser considerada.
- No trecho de código, cada novo caractere é somado ao produto da soma dos caracteres anteriores pelo número 31 (número primo, assim reduzindo a probabilidade de se produzir valores iguais).

8.3 Um tipo abstrato de dados (TAD) para tabela *hash*

- Consideramos um tipo abstrato de dados *opaco* em Linguagem C

  Relembrando: Tipo opaco

  - A ED será referenciada mediante um *ponteiro* para a mesma, ao invés de uma variável do tipo "tabela *hash*" diretamente.
  - Isto permite ocultar a implementação, permitindo sua alteração sem interferir no funcionamento de qualquer código que o utilize – desde que a *interface* seja respeitada.

- Modularização do TAD:
  - *TabelaHash.h*: a *interface* em si: definição do tipo opaco; funções disponíveis para as operações na tabela (protótipos).
  - *TabelaHash.c*: o que deve ficar oculto do usuário, além da implementação das funções definidas pela interface. Inclui a definição do *tipo hash*.

- *Tipo hash*: estrutura que contém:
  - `TABLE_SIZE`: tamanho da tabela (m, a quantidade de entradas disponíveis) ;
  - `qtd`: quantidade de entradas ocupadas da tabela;
  - `**itens`: a ED em si, i.e. um *array* de endereços dos dados.

Observação

- Por questões de desempenho e de bom uso da memória, a tabela irá armazenar apenas o endereço para a estrutura que contém os dados, e não os dados em si.
- A alocação é feita à medida que os elementos são inseridos.

Interface: TabelaHash.h

TabelaHash.h

```
struct aluno{
  int matricula;
  char nome[30];
  float n1,n2,n3;
};

typedef struct hash Hash;

Hash* criaHash(int TABLE_SIZE);
void liberaHash(Hash* ha);
int valorString(char *str);
int insereHash_SemColisao(Hash* ha, struct aluno al);
int buscaHash_SemColisao(Hash* ha, int mat, struct aluno* al)
    ;
```

```
14 int insereHash_EnderAberto(Hash* ha, struct aluno al);
15 int buscaHash_EnderAberto(Hash* ha, int mat, struct aluno* al
    );
```

Definição da tabela: TabelaHash.c

TabelaHash.c: bibliotecas usadas e definição do tipo

```
1 #include <stdlib.h>
2 #include <string.h>
3 #include "TabelaHash.h" //inclui os protótipos
4
5 //Definição do tipo Hash
6 struct hash {
7   int qtd, TABLE_SIZE;
8   struct aluno **itens;
9 };
```

Criando uma tabela *hash*

main.c – Exemplo de uso

```
Hash *tabela = criaHash(tamanho);
```

TabelaHash.c: criação da tabela hash

```
11 Hash* criaHash(int TABLE_SIZE) {
12   Hash* ha = (Hash*) malloc(sizeof(Hash));
13   if (ha != NULL) {
14     int i;
15     ha->TABLE_SIZE = TABLE_SIZE;
16     ha->itens = (struct aluno**) malloc(TABLE_SIZE * sizeof(
       struct aluno*));
17     if (ha->itens == NULL) {
18       free(ha);
19       return NULL;
20     }
21     ha->qtd = 0;
22     for (i=0; i < ha->TABLE_SIZE; i++)
23       ha->itens[i] = NULL;
24   }
25   return ha;
26 }
```

Liberando a tabela *hash* da memória

main.c – Exemplo de uso

```
Hash *tabela = criaHash(tamanho);
.
.
liberaHash(tabela);
```

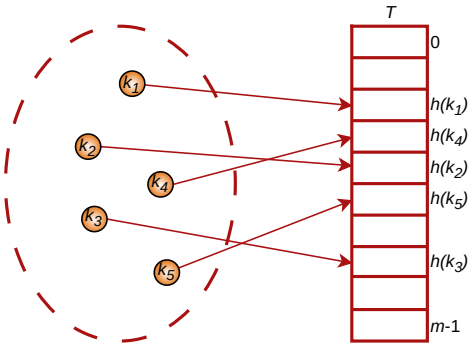
TabelaHash.c: liberando a tabela

```
28 void liberaHash(Hash* ha) {
29   if (ha != NULL) {
30     int i;
31     for (i=0; i < ha->TABLE_SIZE; i++) {
32         if (ha->itens[i] != NULL)
33             free(ha->itens[i]);
34     }
35     free(ha->itens);
36     free(ha);
37   }
38 }
```

8.4 Inserção e busca sem tratamento de colisões

Inserção

- Calcular a posição da chave (valor) no *array*.
- Alocar espaço para os dados.
- Armazenar os dados na posição calculada.



main.c – Exemplo de uso

```
Hash *tabela = criaHash(tamanho);
.
.
struct aluno alu = {1000, "Nicomedes Alves dos Santos", 8.8,
    7.4, 7.7};
insereHash_SemColisao(tabela,alu);
```

TabelaHash.c: inserção na tabela hash

```
int insereHash_SemColisao(Hash* ha, struct aluno al) {
  if (ha == NULL || ha->qtd == ha->TABLE_SIZE)
    return 0;

  int chave = al.matricula;
  //int chave = valorString(al.nome);

  int pos = chaveDivisao(chave,ha->TABLE_SIZE);
  struct aluno* novo;
  novo = (struct aluno*) malloc(sizeof(struct aluno));
  if (novo == NULL)
    return 0;
  *novo = al;
  ha->itens[pos] = novo;
  ha->qtd++;
  return 1;
}
```

Busca

- Calcular a posição da chave (valor) no *array*.
- Verificar se há dados na posição calculada.
- Retornar os dados.

main.c – Exemplo de uso

```
Hash *tabela = criaHash(tamanho);
.
.
struct aluno alu = {1000, "Nicomedes Alves dos Santos", 8.8,
    7.4, 7.7};
insereHash_SemColisao(tabela,alu);
.
.
buscaHash_SemColisao(tabela, 1000, &al);
printf("%s, %d\n",al.nome,al.matricula);
```

TabelaHash.c: busca na tabela hash

```
int buscaHash_SemColisao(Hash* ha, int mat, struct aluno* al) {
  if (ha == NULL)
    return 0;

  int pos = chaveDivisao(mat,ha->TABLE_SIZE);
  if (ha->itens[pos] == NULL)
    return 0;
  *al = *(ha->itens[pos]);
  return 1;
}
```

8.5 *Hashing* universal, perfeito e imperfeito

Hashing universal

- Função de *hashing* está sujeita ao problema de se gerar *colisões* – posições iguais para chaves diferentes;
- Por se tratar de uma função determinística, pode ser manipulada de forma indesejada.
  - Conhecendo-se a função de *hashing*, pode-se escolher as chaves de entrada de modo que todas colidam, diminuindo o desempenho da tabela na busca para $O(n)$.
- O *hashing universal* é uma estratégia que busca minimizar o problema de colisões.
- Escolhe-se aleatoriamente, *em tempo de execução*, a função de *hashing* que será utilizada.
- Para isso, constrói-se um conjunto (ou família) de funções de *hashing*.

Família de funções

- Existem diversas formas de se construir uma família de funções de *hashing*.
- Ex.: Uma família de funções pode ser facilmente obtida da seguinte forma:
  1. Escolhe-se um número primo p que seja maior do que qualquer chave k a ser inserida.
  2. p também deve ser maior do que o tamanho da tabela, m.
  3. Então, escolhe-se, aleatoriamente, dois números inteiros, a e b, tais que $0 < a < p$ e $0 \leq b < p$.

4. Dados p, a e b, define-se a *função de hashing universal* como sendo:

$$h_{ab}(k) = [(a \cdot k + b) \bmod p] \bmod m$$

- A função de *hashing* universal dada permite o tamanho da tabela, m, possa não ser necessariamente primo.
- Ainda, como existem ao todo $p - 1$ valores possíveis para a e p para b, é possível gerar $p(p - 1)$ funções de *hashing* diferentes.

Hashing imperfeito × *hashing* perfeito

- Dependendo do tamanho da tabela, m, e dos valores inseridos, uma função de *hashing* pode ser definida como:
  - *Hashing* imperfeito;
  - *Hashing* perfeito.

Hashing imperfeito

- Para duas chaves diferentes k_1 e k_2, a saída da função de *hashing* é a mesma, i.e. $h(k_1) = h(k_2)$: *colisão*.
- Como já explicado, algo indesejável, pois diminui o desempenho.
- De modo geral, muitas tabelas *hash* fazem uso de alguma outra ED para lidar com tal problema, como será visto adiante.

Hashing perfeito

- *Nunca* ocorre colisão:
  - Chaves diferentes sempre produzem posições diferentes.
- No pior caso, as operações são sempre executadas em tempo constante, $O(1)$.
- É utilizado onde a colisão não é tolerável.
- Trata-se de um tipo de aplicação muito específica, por exemplo, o conjunto de palavras reservadas de uma linguagem de programação. Nesse caso, se conhece previamente o conteúdo a ser armazenado.

8.6 Tratamento de colisões

Mundo ideal × mundo real

- No mundo ideal:
  - *Hashing* perfeito – a função de *hashing* irá sempre fornecer posições diferentes para cada uma das chaves inseridas.
- No mundo real, contudo:
  - Independente da função de *hashing* utilizada, a mesma fatalmente poderá retornar a mesma posição para duas chaves diferentes: *colisão*!

Técnicas para tratamento de colisões

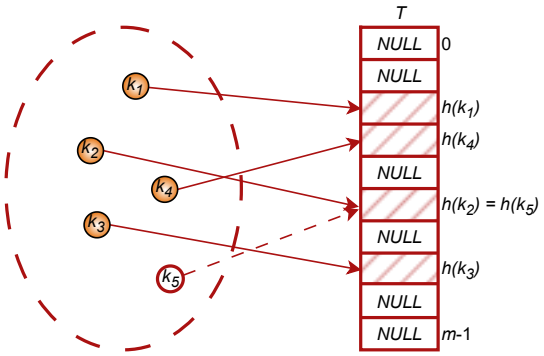
- Infelizmente, não se pode garantir que as funções de *hashing* possuam um bom potencial de espalhamento por que as colisões – teoricamente inevitáveis – também são uniformemente distribuídas.
- Por isso, é preciso ter uma abordagem para tratá-las.
- Duas técnicas muito comuns:
  - *Endereçamento aberto*;
  - *Encadeamento separado*.

Endereçamento aberto

- Em inglês, *open addressing*. Também conhecido por *rehash*.
- Em caso de colisão, percorre-se a tabela *hash* buscando por uma posição ainda não ocupada.
- Os elementos são armazenados na própria tabela.
  - Desta forma, evita-se o uso de listas encadeadas.

Exemplo

- k_5 é tal que $h(k_5) = h(k_2)$, com k_2 já inserido na tabela:
  - Percorre-se a tabela até encontrar uma posição vaga (`NULL`).



Vantagens

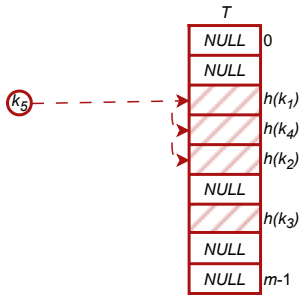
- Maior número de posições na tabela para a mesma quantidade de memória usada no *encadeamento separado*;
  - A memória utilizada para armazenar os ponteiros da lista encadeada no encadeamento separado pode ser aqui usada para aumentar o tamanho da tabela, diminuindo o número de colisões.
- Busca é realizada dentro da própria tabela – recuperação mais rápida de elementos;
- Voltada para aplicações com restrições de memória;
- Ao invés de acessar ponteiros extras, calcula-se a sequência de posições a serem armazenadas.

Desvantagens

- Custo maior no processamento no cálculo das posições
  - Se deve ao fato de que, quando uma colisão ocorre, deve-se calcular uma nova posição da tabela.
- Possibilidade de colisões sucessivas.

Colisões sucessivas: exemplo

- Ao simplesmente se percorrer a tabela, haverá colisões sucessivas.



Estratégias de endereçamento aberto

- Para a realização do cálculo da nova posição após a colisão, existem três estratégias muito utilizadas:
  - Sondagem linear;
  - Sondagem quadrática;
  - Duplo *hash*.

Sondagem linear

- Também conhecida como *tentativa linear*, *espalhamento linear* ou *rehash linear*.
- Tenta espalhar os elementos de forma sequencial, a partir da posição calculada utilizando a função de *hashing*:
  - Se já existe um elemento na posição, tenta-se o próximo, até que se encontre uma posição vazia, de forma circular.
- Funcionamento:
  - Primeiro elemento ($i = 0$) é colocado na posição obtida pela função de *hashing*: `pos`;
  - Segundo elemento (colisão) é colocado na posição `pos + 1` – caso a mesma esteja livre;
  - Terceiro elemento (nova colisão) é colocado na posição `pos + 2` – caso a mesma esteja livre, etc.;

TabelaHash.c: sondagem linear

```
118 int sondagemLinear(int pos, int i, int TABLE_SIZE) {
119   return ((pos + i) & 0x7FFFFFFF) % TABLE_SIZE;
120 }
```

Exemplo

Chave	$h(k_i)$	Inserção
k_1	2	Posição 2 vazia: insere o elemento.
k_2	6	Posição 6 vazia: insere o elemento.
k_3	2	Posição 2 ocupada, procura na próxima posição: 3. Posição 3 vazia. Insere o elemento.
k_4	9	Posição 9 vazia: insere o elemento.
k_5	9	Posição 9 ocupada, procura na próxima posição. Como 9 é a última, volta para o início: 0. Posição 0 vazia. Insere elemento.

k_5	0
NULL	1
k_1	2
k_3	3
NULL	4
NULL	5
k_2	6
NULL	7
NULL	8
k_4	9

Desvantagem: agrupamento primário

- À medida que a tabela *hash* fica mais cheia, o tempo para incluir ou buscar um elemento aumenta;
- À medida que os elementos são inseridos, surgem longas sequências de posições ocupadas;
- A ocorrência desses agrupamentos aumenta o tempo de pesquisa, diminuindo o desempenho.

Sondagem quadrática

- Também conhecida como *tentativa quadrática*, *espalhamento quadrático* ou *rehash quadrático*;
- Tenta espalhar os elementos com base numa equação do segundo grau.
- Exemplo: $pos + (c_1 \cdot i) + (c_2 \cdot i^2)$.

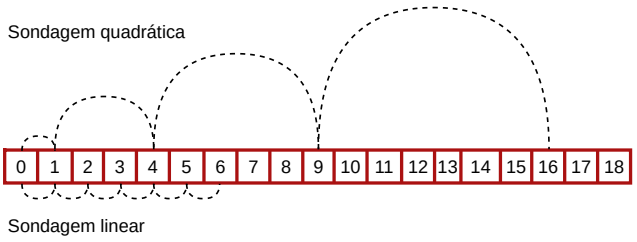
- *pos* é a posição obtida pela função de *hashing*;
- i é tentativa atual;
- c_1 e c_2 são os coeficientes da equação.

- Funcionamento:
  - Primeiro elemento ($i = 0$) é colocado na posição obtida pela função de *hashing*: `pos`;
  - Segundo elemento (colisão) é colocado na posição `pos + (c1*1) + (c2*1)` – caso a mesma esteja livre;
  - Terceiro elemento (nova colisão) é colocado na posição `pos + (c1*2) + (c2*2*2)` – caso a mesma esteja livre, etc.;

TabelaHash.c: sondagem quadrática

```
int sondagemQuadratica(int pos, int i, int TABLE_SIZE) {
  pos = pos + 2*i + 5*i*i; // hash + (c1 * i) + (c2 * i^2)
  return (pos & 0x7FFFFFFF) % TABLE_SIZE;
}
```

Comparativo



Aspectos

- Resolve o problema de agrupamento primário;
- Porém, gera outro problema, conhecido como *agrupamento secundário*:
  - Todas as chaves que produzirem a mesma posição inicial também produzirão as mesmas posições na sondagem quadrática.
- Felizmente, a degradação produzida pelos agrupamentos secundários é menor que a produzida pelos agrupamentos primários.

Duplo *hash*

- Também conhecida como *espalhamento duplo*;
- Tenta espalhar os elementos utilizando *duas* funções de *hashing*, necessariamente diferentes entre si:
  - a primeira função de *hashing*, h_1, é utilizada para calcular a *posição inicial* do elemento;
  - a segunda função de *hashing*, h_2, é utilizada para calcular os deslocamentos em relação à posição inicial, em caso de colisão.
- A posição de um novo elemento na tabela *hash* é obtida como sendo

$$h_1 + i \cdot h_2,$$

  onde i é tentativa atual de inserção do elemento.
- A segunda função de *hashing* não pode resultar em *zero*, pois neste caso não haveria deslocamento.
- Funcionamento:
  - Primeiro elemento ($i = 0$) é colocado na posição obtida pela função de *hashing* h_1;
  - Segundo elemento (colisão) é colocado na posição dada por $h_1 + h_2$ – caso a mesma esteja livre;
  - Terceiro elemento (nova colisão) é colocado na posição $h_1 + 2h_2$ – caso a mesma esteja livre, etc.;

TabelaHash.c: duplo hash

```
127 int duploHash(int H1, int chave, int i, int TABLE_SIZE) {
128   int H2 = chaveDivisao(chave,TABLE_SIZE-1) + 1;
129   return ((H1 + i*H2) & 0x7FFFFFFF) % TABLE_SIZE;
130 }
```

Inserção com tratamento de colisões

- Calcular a posição da chave (valor) no *array*.
- Recalcular a posição enquanto houver colisão (limitar o número de tentativas).
- Alocar espaço para os dados.
- Armazenar os dados na posição calculada.

main.c – Exemplo de uso

```
Hash *tabela = criaHash(tamanho);
.
.
struct aluno alu = {1000, "Nicomedes Alves dos Santos", 8.8,
    7.4, 7.7};
insereHash_EnderAberto(tabela,alu);
```

TabelaHash.c: inserção com tratamento de colisão

```
132 int insereHash_EnderAberto(Hash* ha, struct aluno al) {
133   if (ha == NULL || ha->qtd == ha->TABLE_SIZE)
134     return 0;
135
136   int chave = al.matricula;
137   //int chave = valorString(al.nome);
138
139   int i, pos, newPos;
140   pos = chaveDivisao(chave,ha->TABLE_SIZE);
141   for (i=0; i < ha->TABLE_SIZE; i++) {
142     newPos = sondagemLinear(pos,i,ha->TABLE_SIZE);
143     //newPos = sondagemQuadratica(pos,i,ha->TABLE_SIZE);
144     //newPos = duploHash(pos,chave,i,ha->TABLE_SIZE);
145     if (ha->itens[newPos] == NULL) {
146       struct aluno* novo;
147       novo = (struct aluno*) malloc(sizeof(struct aluno));
148       if (novo == NULL)
149           return 0;
150       *novo = al;
151       ha->itens[newPos] = novo;
152       ha->qtd++;
153       return 1;
154     }
155   }
156   return 0;
157 }
```

Busca com tratamento de colisões

- Calcular a posição da chave (valor) no *array*.
- Verificar se há dados na posição calculada, e se tais dados correspondem à chave.
- Recalcular a posição enquanto os dados forem diferentes da chave.
- Retornar os dados.

main.c – Exemplo de uso

```
Hash *tabela = criaHash(tamanho);
.
.
struct aluno alu = {1000, "Nicomedes Alves dos Santos", 8.8,
    7.4, 7.7};
insereHash_EnderAberto(tabela,alu);
.
.
int x = buscaHash_EnderAberto(tabela, 1000, &al);
```

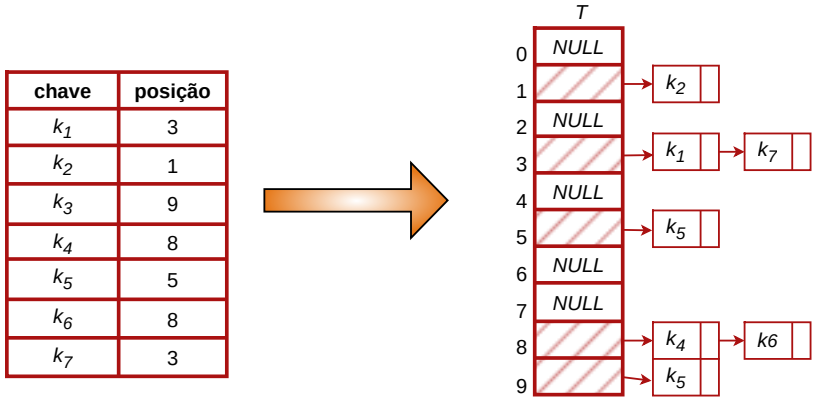
TabelaHash.c: busca com tratamento de colisão

```
159 int buscaHash_EnderAberto(Hash* ha, int mat, struct aluno* al
    ) {
160   if (ha == NULL)
161     return 0;
162
163   int i, pos, newPos;
164   pos = chaveDivisao(mat,ha->TABLE_SIZE);
165   for (i=0; i < ha->TABLE_SIZE; i++) {
166     newPos = sondagemLinear(pos,i,ha->TABLE_SIZE);
167     //newPos = sondagemQuadratica(pos,i,ha->TABLE_SIZE);
168     //newPos = duploHash(pos,mat,i,ha->TABLE_SIZE);
169     if (ha->itens[newPos] == NULL)
170       return 0;
171
172     if (ha->itens[newPos]->matricula == mat) {
173       *al = *(ha->itens[newPos]);
174       return 1;
175     }
176   }
177   return 0;
178 }
```

Encadeamento separado

- Em inglês, *separate chaining*.
- Não procura por posições vagas (valor `NULL`) dentro do *array* que define a tabela.
- Armazena em cada posição do *array* o início de uma *lista dinâmica encadeada*.
  - É dentro dessa lista que serão armazenadas as colisões (elementos com chaves iguais) para aquela posição do *array*.

Exemplo



Características

- A lista dinâmica encadeada mantida em cada posição da tabela pode ser ordenada ou não. Lista não ordenada
  - Inserção tem complexidade $O(1)$ no pior caso: basta inserir o elemento no *início da lista*.
  - Busca tem complexidade $O(m)$ no pior caso: *busca linear*.
- *Desvantagem*: quantidade de memória consumida – gasta-se mais memória para manter os ponteiros que ligam os diferentes elementos dentro de cada lista.

8.7 Exercícios

1. Descreva de que forma podemos detectar que todas as posições possíveis vagas da tabela *hash* foram acessadas durante o reespalhamento.
2. Considere uma tabela de *hash* de tamanho $m = 1000$ e a função de *hashing* usando o método da multiplicação com $A = (\sqrt{5} - 1)/2$. Calcule os valores de *hash* das chaves 61, 62, 63, 64 e 65.
3. Suponha um conjunto de n chaves, formado pelos n primeiros múltiplos do número 7. Quantas colisões seriam obtidas mediante a aplicação das funções de *hashing* seguintes?

(a) $h(k) = k \bmod 7$ (b) $h(k) = k \bmod 14$ (c) $h(k) = k \bmod 5$

4. Suponha uma tabela *hash* de tamanho $m = 10$ com endereçamento aberto para armazenar chaves no intervalo $[1, 999]$. Insira as seguintes chaves na tabela: 371, 173, 203, 11, 24, nesta ordem, considerando diferentes métodos de resolução de colisões:

   (a) Sondagem linear, função de *hashing* $h(k) = (k + i) \bmod m$.

   (b) Sondagem quadrática, função de *hashing* $h(k) = (k + i^2) \bmod m$.

   (c) Sondagem quadrática, função de *hashing* $h(k) = (k + 2i + i^2) \bmod m$.

   (d) *Hash* duplo, funções de *hashing*: $h_1(k) = k \bmod m$ e $h_2(k) = 7 - (k \bmod 7)$.

5. (CORMEN, 2024) Considere uma tabela de espalhamento com 9 posições inicialmente vazia e a função de *hash* $h(k) = k \bmod 9$. Demonstre o que acontece quando inserimos as chaves 5, 28, 19, 15, 20, 33, 12, 17, 10 com as colisões resolvidas por encadeamento.

Referências

1. BACKES, A. R. **Algoritmos e Estruturas de Dados em Linguagem C**. 1. ed. Rio de Janeiro: LTC, 2023. Disponível em: https://www.sistemas.ufu.br/biblioteca-gateway/minhabiblioteca/9788521638315.

2. CORMEN, T. H.; et al. **Algoritmos**. 4. ed. Rio de Janeiro: GEN LTC, 2024. Disponível em: https://www.sistemas.ufu.br/biblioteca-gateway/minhabiblioteca/9788595159914.

Os materiais de parte desta seção foram gentilmente cedidos por Ana Cláudia Martinez (FACOM/UFU) e André R. Backes (UFSCar).

TAD tabela *hash*: disponível em: http://www.facom.ufu.br/~backes/wordpress/ProjTabelaHash.zip. Acesso em: 31/mar/2025.

Adaptações: Renato Pimentel, FACOM/UFU

9 Armazenamento secundário

9.1 Conceitos básicos

- *Mecanismos de indexação* usados para acelerar o acesso aos dados desejados em BD.
  - Ex.: *catálogo de autores* na biblioteca.
- *Chave de busca* – atributo ou conjunto de atributos usado para procurar registros em um arquivo.
- Um *arquivo de índice* consiste em registros (chamados *entradas de índice*) com estrutura:

chave de busca	ponteiro

- Um arquivo de índice é normalmente muito menor que o arquivo original.
- Dois tipos básicos de índices:
  - *Índices ordenados*: as chaves de pesquisa são armazenadas em ordem classificada;
  - *Índices de hash*: as chaves de pesquisa são distribuídas uniformemente entre "buckets" (entradas) usando uma função de *hashing*.

Métricas de avaliação de índice

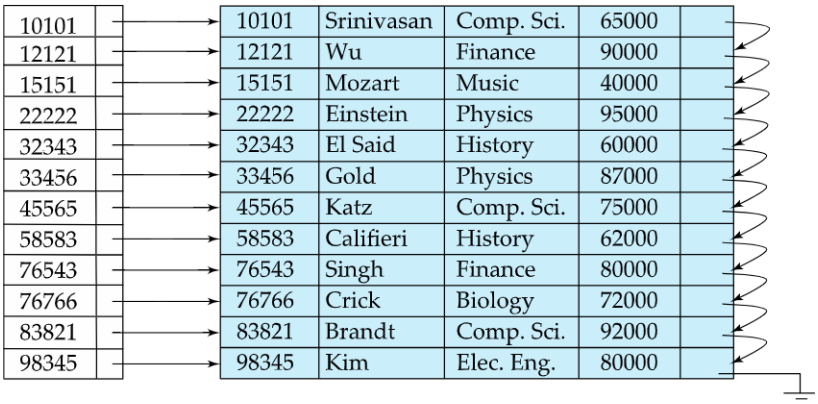
- Tipos de acesso suportados de forma eficiente. Exemplos:
  - Registros com um valor específico de um atributo.
  - Registros com um valor de atributo que se enquadra em um intervalo especificado de valores.
- Tempo de acesso;
- Tempo de inserção;
- Tempo de exclusão;
- Sobrecarga de espaço de armazenamento.

9.2 Índices ordenados

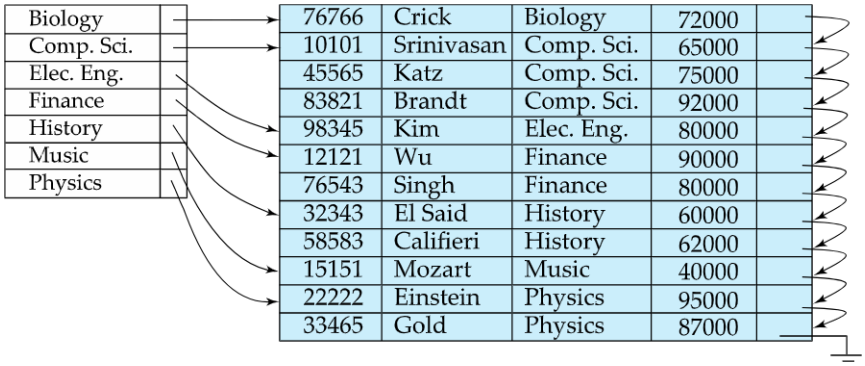
- Em um índice ordenado, as entradas são armazenadas classificadas pelo valor da *chave de busca*:
  - Números, *data/hora*, *strings*.
- *Índice agrupado* (*clustering index*): num arquivo ordenado sequencialmente, o índice cuja chave de busca especifica a ordem sequencial do arquivo.
  - Também chamado de *índice primário*.
  - A chave de busca de um índice primário geralmente é, mas não necessariamente, a chave primária.
- *Índice secundário*: um índice cuja chave de busca especifica uma ordem diferente da ordem sequencial do arquivo. Também chamado de *índice não agrupado* (*nonclustering index*).
- *Arquivo sequencial indexado*: arquivo sequencial ordenado em uma chave de busca, com um índice agrupado na mesma.

Arquivos de índice denso

- *Índice denso* – Há um registro de índice para cada valor de chave de busca no arquivo.
- Por exemplo, índice no atributo `ID` da relação `instrutor`:

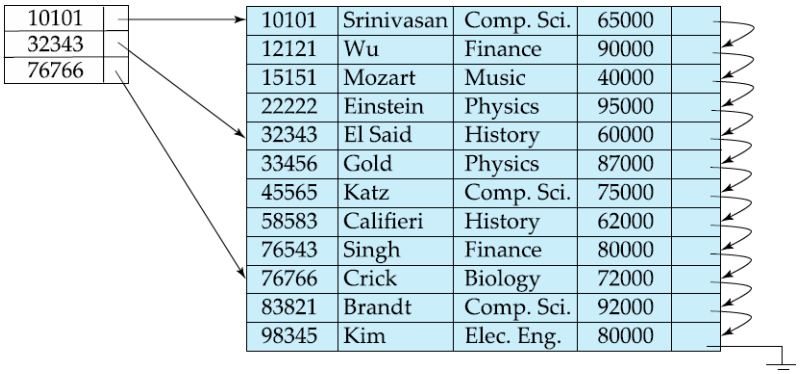


- Índice denso em `dept_name`, com arquivo de `instrutor` classificado em `dept_name`:

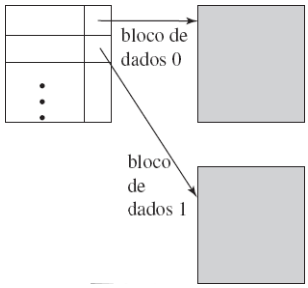


Arquivos de índice esparso

- *Índice esparso*: contém registros de índice apenas para alguns valores da chave de busca.
  - Aplicável quando os registros estão classificados pela chave de busca.
- Para localizar um registro com valor de chave de busca K deve-se:
  - Encontrar a entrada de índice com o maior valor de chave $< K$;
  - Pesquisar sequencialmente o arquivo a partir do registro apontado pela entrada de índice.



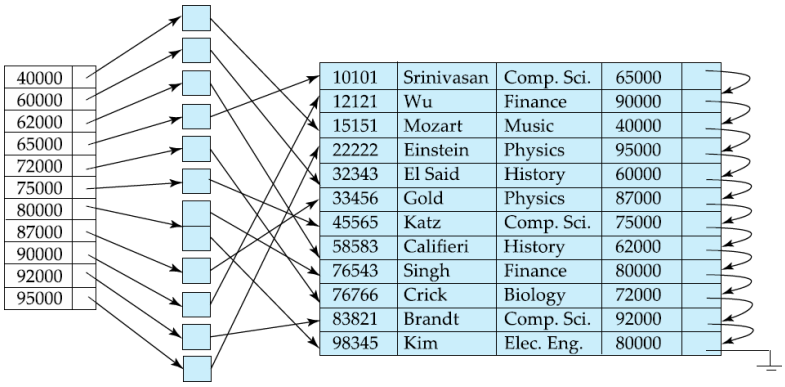
- Comparado com índices densos:
  - Menores espaço e sobrecarga de manutenção para inserções e exclusões.
  - Geralmente mais lento que o índice denso para localizar registros.
- Bom meio termo:
  - Para índice agrupado: índice esparso com uma entrada de índice por *bloco* do arquivo – menor valor da chave de busca no mesmo.



  - Para índice não agrupado: índice esparso sobre índice denso (*índice multinível*)

Exemplo de índice secundário

- Índice secundário no campo salarial de `instrutor`:



- O registro de índice aponta para um *bucket*, que contém ponteiros para todos os registros reais com aquele valor de chave de busca específico.
- Os índices secundários devem ser densos.

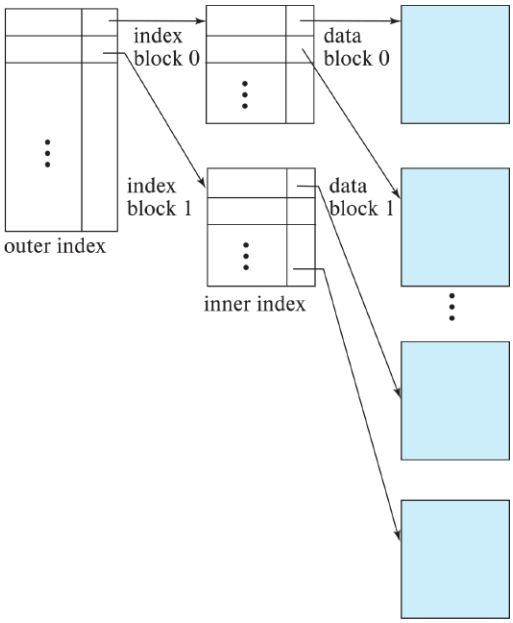
Índices agrupados × índices não agrupados

- Os índices oferecem benefícios substanciais na busca de registros.
- Entretanto, índices impõem sobrecarga na modificação do banco de dados:
  - Quando um registro é inserido ou excluído, cada índice na relação deve ser atualizado;
  - Quando um registro é atualizado, qualquer índice em um atributo atualizado deve ser atualizado.

- A varredura sequencial usando índice agrupado é eficiente, mas uma varredura sequencial usando um índice secundário (não agrupado) é cara em disco magnético:
  - Cada acesso ao registro pode buscar um novo bloco do disco;
  - Cada busca de bloco no disco magnético requer cerca de 5 a 10 milissegundos.

Índice multinível

- Se o índice não couber na memória, o acesso se torna caro.
- *Solução*: tratar o índice mantido no disco como um arquivo sequencial e construir um índice esparso nele.
  - *Índice externo* – um índice esparso do índice básico;
  - *Índice interno* – índice básico para o arquivo.
- Se mesmo o índice externo for grande demais para caber na memória principal, outro nível de índice poderá ser criado, e assim por diante.
- Índices em todos os níveis devem ser atualizados a cada inserção ou exclusão no arquivo.



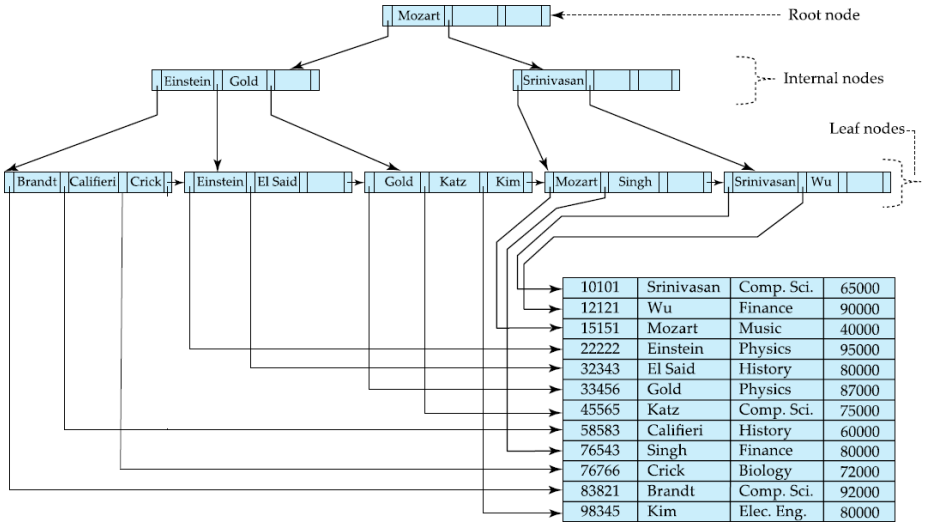
Índices em várias chaves

- *Chave de busca composta.*
  - Por exemplo, índice na relação `instrutor` para atributos `(nome, ID)`
  - Os valores são classificados lexicograficamente.
    - Ex.: `(João, 12121)` < `(João, 13514)` e `(João, 13514)` < `(Pedro, 11223)`.
  - Pode-se buscar apenas pelo `nome`, ou `(nome, ID)`.

9.3 Árvore B^+

- Desvantagem dos arquivos sequenciais indexados:
  - O desempenho diminui à medida que o arquivo cresce, pois muitos *blocos de estouro* (*overflow blocks*) são criados.
  - É necessária a reorganização periódica de todo o arquivo – não desejável.
- Vantagem dos arquivos de índice de árvore B^+:
  - Reorganiza-se automaticamente com pequenas alterações locais, em face de inserções e exclusões.
  - Não é necessária a reorganização de todo o arquivo para manter o desempenho.
- (Menor) desvantagem das árvores B^+:
  - Sobrecarga extra na inserção e exclusão, sobrecarga de espaço.
- Vantagens das árvores B^+ superam as desvantagens:
  - Árvores B^+ são amplamente utilizadas.

Exemplo de árvore B$^+$



Definição

- A árvore B$^+$ é uma árvore enraizada que satisfaz as seguintes propriedades:
  - Todos os caminhos da raiz até a folha têm a mesma altura;
  - Cada nó que não é a raiz ou uma folha tem entre $\lceil n/2 \rceil$ e n filhos.
  - Um nó folha tem entre $\lceil (n-1)/2 \rceil$ e $n-1$ valores.
  - Casos especiais:
    * Se a raiz não for uma folha, ela tem pelo menos 2 filhos.
    * Se a raiz for uma folha (i.e. se não houver outros nós na árvore), ela pode ter entre 0 e $n-1$ valores.

Estrutura de um nó da árvore B$^+$

- Nó típico:

P_1	K_1	P_2	$\ldots$	P_{n-1}	K_{n-1}	P_n

  - K_i são valores de chaves de busca;
  - P_i são ponteiros para filhos (nós internos), ou ponteiros para registros ou grupos de registros (nós folhas).

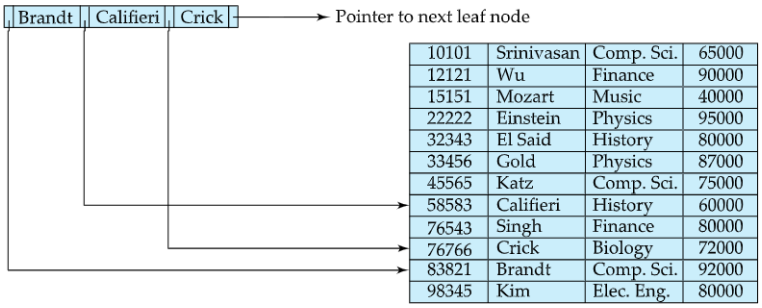
- As chaves de busca num nó são ordenadas:

$$K_1 < K_2 < K_3 < \cdots < K_{n-1}$$

(Inicialmente sem chaves duplicadas)

Nó folha de uma árvore B$^+$

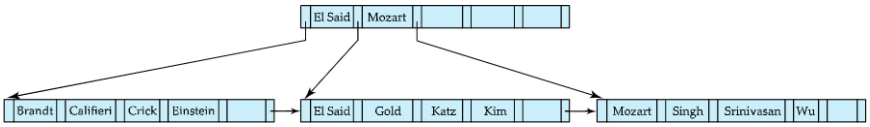
- Propriedades de um nó folha:
  - Para $i = 1, 2, \ldots, n-1$, o ponteiro P_i aponta para um registro de arquivo com valor de chave de busca K_i;
  - Se L_i e L_j são nós folhas e $i < j$, os valores das chaves de busca de L_i são menores ou iguais aos valores das chaves de busca de L_j;
  - P_n aponta para o próximo nó folha, seguindo a ordem das chaves de busca.



Nó interno de uma árvore B$^+$

- Nós internos formam um índice esparso multinível para os nós-folha. Para um nó interno com n ponteiros:
  - Todas as chaves de busca na subárvore para a qual P_1 aponta são menores que K_1;
  - Para $2 \leq i \leq n-1$, todas as chaves de busca na subárvore para a qual P_i aponta têm valores maiores ou iguais a K_{i-1} e menores que K_i;
  - Todas as chaves de busca na subárvore para a qual P_n aponta têm valores maiores ou iguais a K_{n-1}.

Exemplo de árvore B$^+$



Árvore B$^+$ para arquivo `instrutor` ($n = 6$).

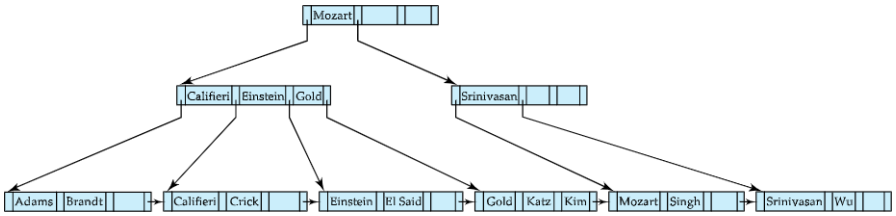
- Os nós folhas devem ter entre 3 e 5 valores ($\lceil (n-1)/2 \rceil$ e $n-1$, com $n = 6$).
- Nós internos (exceto a raiz) devem ter entre 3 e 6 filhos (entre $\lceil n/2 \rceil$ e n com $n = 6$).
- A raiz deve ter pelo menos 2 filhos.

Observações sobre árvores B$^+$

- Como as conexões entre nós são feitas por ponteiros, blocos "logicamente" próximos não precisam ser "fisicamente" próximos.
- Os níveis internos da árvore B$^+$ formam uma hierarquia de índices esparsos.
- A árvore B$^+$ contém um número relativamente pequeno de níveis:
  - ☐ O nível abaixo da raiz tem pelo menos $2 \cdot \lceil n/2 \rceil$ valores;
  - ☐ O próximo nível tem pelo menos $2 \cdot \lceil n/2 \rceil \cdot \lceil n/2 \rceil$ valores;
  - ☐ ... etc.
    - Se houver K valores de chave de busca no arquivo, a altura da árvore será no máximo $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$;
    - Assim as buscas podem ser conduzidas de forma eficiente.
- Inserções e exclusões no arquivo principal podem ser tratadas com eficiência, pois o índice pode ser reestruturado em tempo logarítmico.

Consultas em árvores B$^+$

```
function encontrar(v)
  C = nó raiz
  while (C não é um nó folha) begin
    seja i = menor número tal que v <= C.Ki
    if não houver tal número i then begin
      C = último ponteiro não nulo em C
    end
    else if ( v = C.Ki ) then C = Pi+1
    else C = C.Pi
  end
  if para algum i , Ki = v then
    retorne C.Pi
  else retorne null /* nenhum registro com valor v */
```



- *Consultas por intervalo* encontram todos os registros com valores de chave de busca num determinado intervalo.
  - Veja o livro do Silberschatz para detalhes da função `findRange(lb, ub)`, que retorna o conjunto de tais registros;
  - Implementações reais geralmente fornecem uma interface iteradora para buscar registros correspondentes um de cada vez, usando uma função `next()`.

Altura e desempenho da B$^+$

- Se houver K registros no arquivo, a altura da árvore não será maior que $\lceil \log_{\lceil n/2 \rceil}(K) \rceil$.
- Um nó tem geralmente o mesmo tamanho de um *bloco de disco*, normalmente 4 kilobytes.
  - e um valor típico de n é em torno de 100 (40 bytes por entrada de índice)
- Com 1 milhão de valores de chave de busca e $n = 100$:
  - no máximo $\log_{50}(1.000.000) = 4$ nós são acessados numa travessia de consulta da raiz até uma folha.
- Compare isso com uma árvore binária balanceada com 1 milhão de valores da chave de busca – cerca de 20 nós serão acessados em uma consulta.
  - A diferença acima é significativa, pois cada acesso ao nó pode precisar de uma operação de E/S de disco, custando cerca de *10 milissegundos*!
  - Em memória *flash*, 10 a 100 microssegundos – ainda significativo!

Chaves não-exclusivas

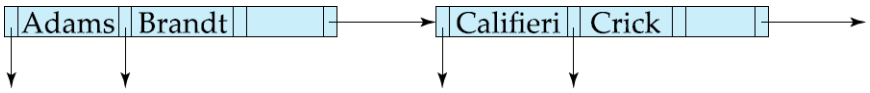
- Se uma chave de busca a_i não é chave candidata – i.e. pode ter duplicatas – cria-se em um índice em uma *chave composta* (a_i, A_p), que é única.
  - A_p pode ser uma chave primária, ID de registro, ou qualquer outro atributo que garanta exclusividade.
- A busca por $a_i = v$ pode ser implementada por uma *consulta por intervalo* na chave composta, com intervalo $(v, -\infty)$ a $(v, +\infty)$.
- Porém mais operações de E/S são necessárias para buscar os registros reais
  - Se o índice estiver agrupado, todos os acessos serão sequenciais.
  - Se o índice não for agrupado, cada acesso ao registro pode precisar de uma operação de E/S.

Inserção na árvore B$^+$

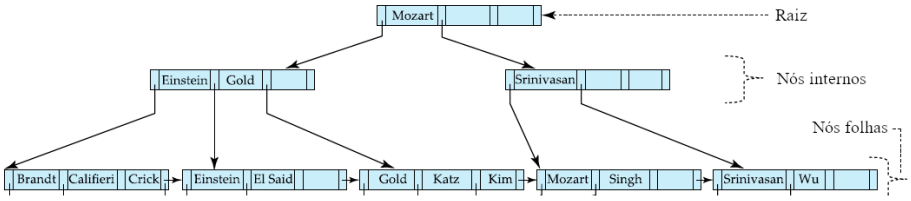
- Suponha que o registro já foi adicionado ao arquivo. Seja:
  - pr seja um ponteiro para o registro, e
  - v o valor da chave de busca do registro.
- Encontre o nó folha no qual o valor da chave de busca apareceria.
  1. Se houver espaço no nó folha, insira o par (v, pr) no nó folha;

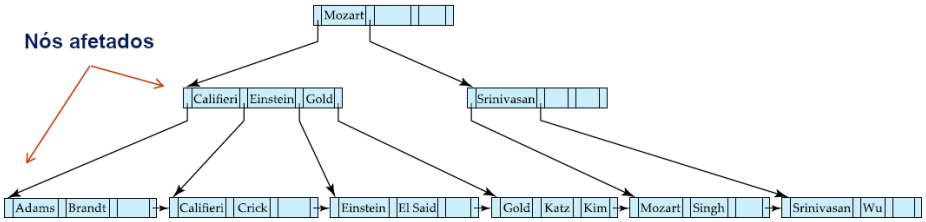
2. Caso contrário, divida o nó – junto com a nova entrada (v, pr) – conforme discutido a seguir e propague as atualizações para os nós pais.

- Dividindo um nó folha:
  - Tomam-se os n pares (valor da chave de busca, ponteiro) (incluindo o que está sendo inserido) em ordem. Colocam-se os primeiros $\lceil n/2 \rceil$ no nó original, e o restante em um novo nó.
  - Seja p o novo nó, e k o menor valor da chave em p. Insira (k, p) no pai do nó que está sendo dividido.
  - Se o pai estiver completo, divida-o e *propague* a divisão para cima.
- A divisão dos nós prossegue para cima até que um nó que não esteja cheio seja encontrado.
  - No pior caso, o nó raiz pode ser dividido, aumentando a altura da árvore em 1.

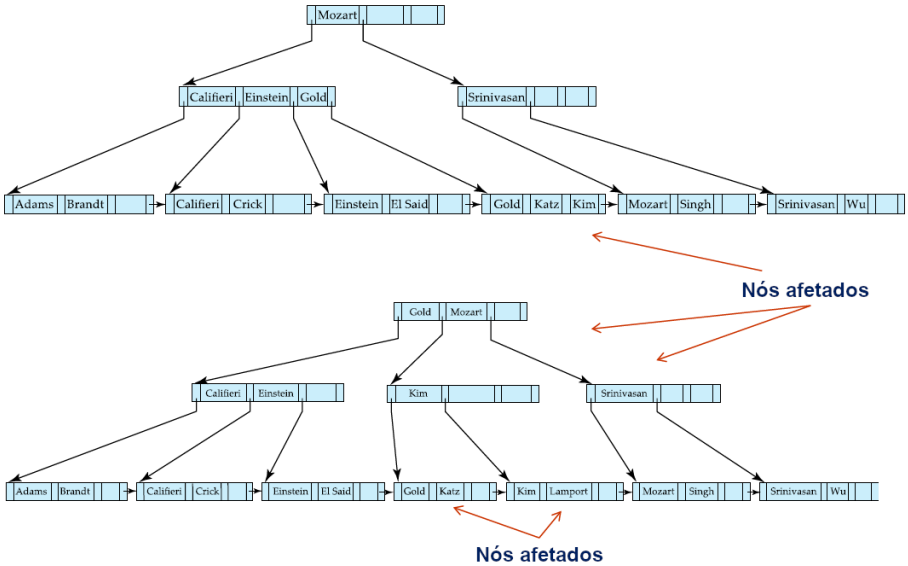


Resultado da divisão do nó contendo Brandt, Califieri e Crick na inserção de Adams. Próximo passo: inserir entrada com (Califieri, ponteiro-para-novo-nó) no pai.



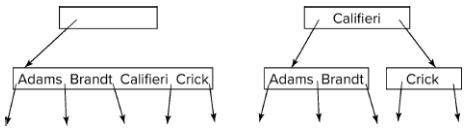


Árvore B^+ antes e após a inserção de Adams.



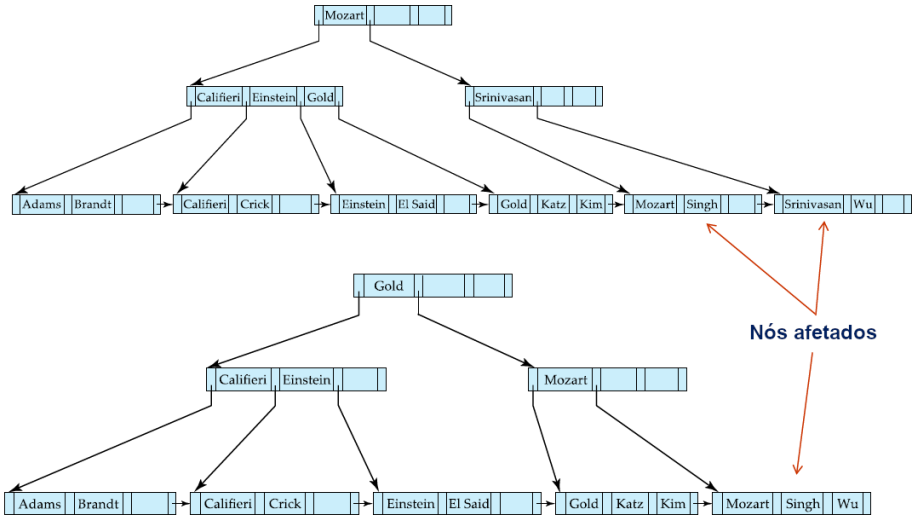
Árvore B$^+$ antes e após a inserção de Lamport.

- Dividindo um nó interno: ao inserir (k, p) em um nó interno N já completo:
  - Copie N para uma área de memória M com espaço para $n+1$ ponteiros e n chaves;
  - Insira (k, p) em M;
  - Copie $P_1, K_1, \ldots, K_{\lceil n/2 \rceil - 1}, P_{\lceil n/2 \rceil}$ de M de volta para o nó N;
  - Copie $P_{\lceil n/2 \rceil + 1}, K_{\lceil n/2 \rceil + 1}, \ldots, K_n, P_{n+1}$ de M para um nó recém-alocado N'
  - Insira $(K_{\lceil n/2 \rceil}, N')$ no pai de N.
- Exemplo:

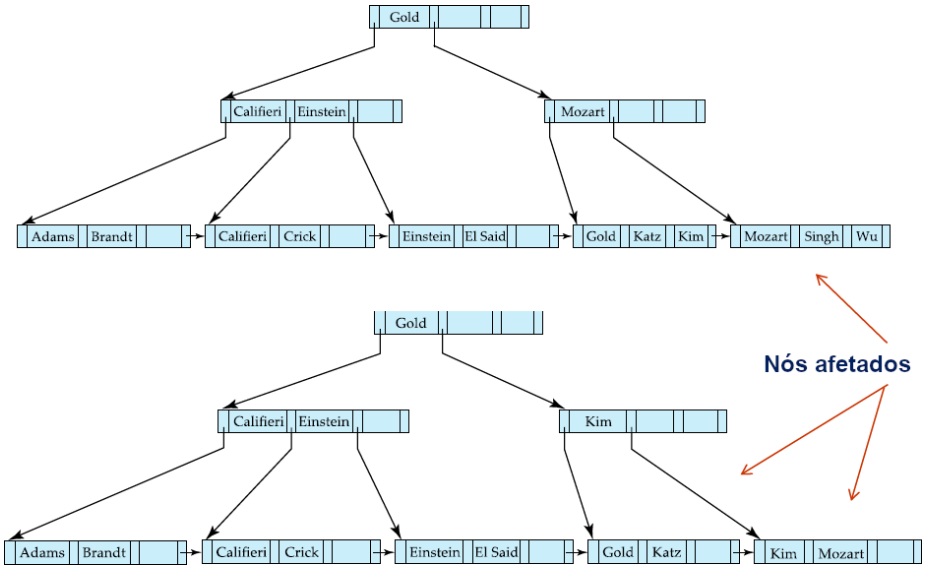


- Pseudocódigo: livro do Silberschatz.

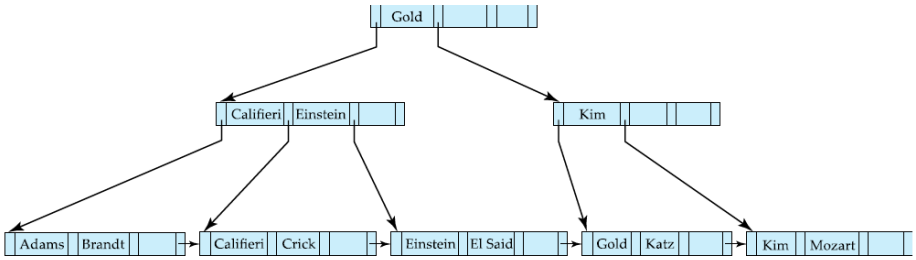
Exemplos de exclusão na árvore B$^+$

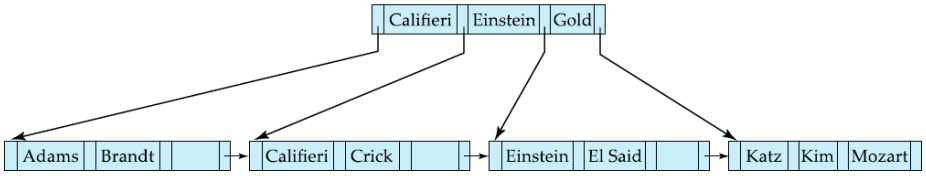


Antes e depois de excluir Srinivasan.
A exclusão causa a *fusão* das folhas não totalmente cheias abaixo



- Antes e depois de excluir Singh e Wu: a folha contendo Singh e Wu ficou com menos de $\lceil n/2 \rceil$ valores; *toma-se emprestado* Kim de seu irmão esq.;
- O valor da chave de busca no pai muda como resultado.





- Antes e depois de excluir Gold: nó com Gold e Katz ficou incompleto e foi mesclado com seu irmão; o mesmo ocorre com o nó pai.
  - O valor que separa os dois nós (no pai) *desce* na mesclagem.
- A raiz então tem apenas um filho e é excluída.

Exclusão na árvore B^+

- Suponha que o registro já foi excluído do arquivo. Seja v o valor de chave de busca do registro e pr o ponteiro para o registro.
- Remover (v, pr) do nó folha;
- Se o nó tiver poucas entradas devido à remoção, e se suas entradas e as de um irmão couberem em um único nó, então:
  - Insira todos os valores da chave de busca nos dois nós em um único nó (o da esquerda), e exclua o outro nó.
  - Exclua de seu pai o par (K_{i-1}, P_i), onde P_i é o ponteiro para o nó excluído, usando recursivamente o procedimento acima.
- Caso contrário, se o nó tiver poucas entradas devido à remoção, mas as entradas no nó e em um irmão não couberem em um único nó, *redistribua os ponteiros*:
  - Redistribua os ponteiros entre o nó e um irmão de modo que ambos tenham mais do que o número mínimo de entradas.
  - Atualize o valor da chave de busca correspondente no pai do nó.
- As exclusões de nós podem ocorrer em cascata, até que um nó que tenha $\lceil n/2 \rceil$ ou mais ponteiros seja encontrado;
- Se o nó raiz tiver apenas um ponteiro após a exclusão, ele será excluído e o único filho passa a ser a nova raiz.

Complexidade das atualizações

- Custo (em termos de número de operações de E/S) de inserção e exclusão de uma única entrada é proporcional à *altura* da árvore:
  - Com K entradas e um número máximo de n filhos, a pior complexidade de caso de inserção/exclusão de uma entrada é $O(\lceil \log_{\lceil n/2 \rceil}(K) \rceil)$
- Na prática, o número de operações de E/S é menor:
  - Os nós internos tendem a estar em *buffer* de memória;
  - Divisões/mesclas são raras: a maioria das operações de inserção/exclusão afetam apenas um nó folha.
- A ocupação média dos nós depende da ordem de inserção:
  - 2/3 ($\approx 67\%$) se aleatoriamente, 50% com inserção ordenada.

Chaves não-exclusivas

- Se lista de ponteiros de tupla com cada chave:
  - Código extra para lidar com listas longas;
  - A exclusão de uma tupla pode ser cara se houver muitas duplicatas na chave de busca
    - A pior complexidade pode ser linear!
  - Baixo *overhead* de espaço, sem custo extra para consultas.
- Torne a chave de busca única adicionando um identificador de registro (ex.: adicionando a chave primária)
  - Espaço extra de armazenamento para chaves;
  - Código mais simples para inserção/exclusão;
  - Amplamente utilizado.

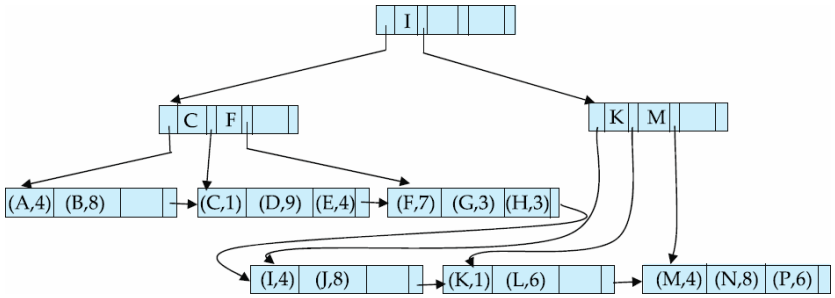
Organização de arquivos por árvore B^+

- Organização de arquivos por árvore B^+:
  - Os nós folhas em uma organização de arquivos por árvore B^+ armazenam registros, em vez de ponteiros.
  - Ajuda a manter os registros de dados agrupados mesmo quando há inserções/exclusões/atualizações.
- Os nós folhas ainda precisam estar meio cheios:

  - Como os registros são maiores que os ponteiros, o número máximo de registros que podem ser armazenados em um nó folha é menor que o número de ponteiros em um nó interno.

- A inserção e a exclusão são tratadas da mesma forma que a inserção e a exclusão de entradas em um índice por árvore B^+.

- Exemplo de organização de arquivos de árvore B^+ (*Index-Organized Tables*)



- A boa utilização do espaço é importante, pois os registros usam mais espaço do que os ponteiros.

- Para melhorar a utilização do espaço, envolva mais nós irmãos na redistribuição durante divisões e fusões.

  - Envolver dois irmãos na redistribuição (para evitar divisão/fusão sempre que possível) resulta em cada nó tendo pelo menos $\lfloor 2n/3 \rfloor$ entradas.

Índices secundários e realocação de registros

- Se um registro for movido (por exemplo, por uma atualização num registro com uma *string* armazenada em um `VARCHAR`), todos os índices secundários que armazenam ponteiros de registro devem ser atualizados.

- As divisões de nós em organizações de arquivos por B^+ tornam-se muito caras.

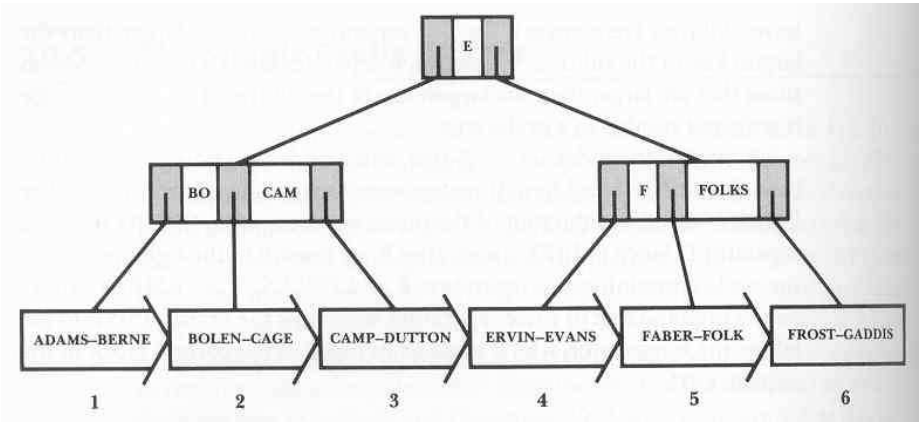
- *Solução*: use a *chave de busca* (chave primária) ao invés do ponteiro de registro no índice secundário:

  - Adicione uma id de registro se a chave de pesquisa da organização de arquivo B^+ não for exclusiva;
  - Travessia extra da organização do arquivo para localizar o registro;
  - Custo mais alto para consultas, mas as divisões dos nós são baratas.

Indexando *strings*

- Strings de comprimento variável como chaves
  - *Fanout* variável – *fanout* é o número de ponteiros do nó (quantidade de filhos);
  - Considerar a utilização do espaço como critério para divisão, e não o número de ponteiros.
- Compressão de prefixo:
  - Os valores-chave nos nós internos podem ser prefixos da chave completa.
    - Mantenha caracteres suficientes para distinguir entradas nas subárvores separadas pelo valor da chave.
    - Ex.: Silas e Silberschatz podem ser separados por Silb.
  - As chaves no nó folha podem ser compactadas compartilhando prefixos comuns.

Indexando *strings*: árvore B$^+$ de prefixo simples



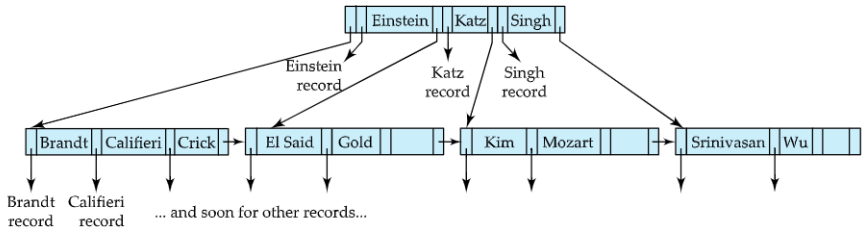
Carregamento em massa (*bulkloading*) econstrução de baixo para cima (*bottom-up*)

- Inserir entradas uma a uma numa árvore B$^+$ requer diversas operações de E/S (no melhor caso, 1; no pior, proporcional à altura da mesma).
  - Assumindo-se que o nível da folha não cabe na memória:
  - pode ser muito ineficiente inserir inúmeras entradas de uma vez (*carregamento em massa*)
- *Alternativa eficiente 1*:

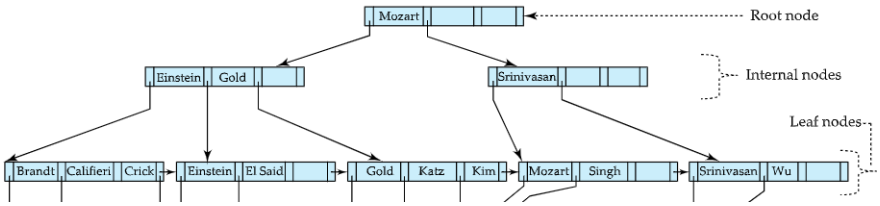
  - ordenar as entradas primeiro (usando algoritmos eficientes para ordenação em memória externa – Sec. 15.4 livro-texto Silberschatz)
  - inserir em ordem:
    * A inserção será na página existente (ou causará uma divisão)
    * Desempenho de IO significativamente melhorado, porém com maioria dos nós folhas apenas meio cheios.

- *Alternativa eficiente 2: construção de árvore B^+ de baixo para cima*
  - Como antes, ordene as entradas;
  - E então, crie a árvore camada por camada, começando pelo nível das folhas;
  - Implementado como parte do utilitário de carregamento em massa pela maioria dos sistemas de banco de dados.

Exemplo de arquivo de índice de árvore B



Árvore B (acima) e B+ (abaixo) para os mesmos dados



Indexação em *flash*

- Custo de E/S aleatório muito menor no flash
  - 20 a 100 microssegundos para leitura/gravação;
- As reescritas não são feitas no mesmo local e (eventualmente) exigem uma exclusão mais cara;
- Tamanho de página ideal, portanto muito menor;
- O carregamento em massa ainda é útil, pois minimiza a eliminação de páginas;

- Estruturas de árvore otimizadas para gravação foram adaptadas para minimizar gravações de página para árvores de busca otimizadas para *flash*.

Indexação na memória principal

- Acesso aleatório na memória:
  - Muito mais barato do que em disco/*flash*;
  - Mas ainda caro comparado à leitura em cache.
    - Estruturas de dados que fazem melhor uso de cache são preferíveis;
  - A busca binária por um valor de chave em um grande nó de árvore B^+ resulta em muitas falhas de cache.
- B^+ com pequenos nós que caibam em cache são preferíveis para reduzir tais falhas;
- *Ideia principal*: usar um tamanho de nó grande para otimizar o acesso ao disco, mas estruturar os dados em um nó usando uma árvore com tamanho de nó pequeno, em vez de usar um *array*.

9.4 Exercícios

1. Qual é o número máximo de acessos a disco para encontrar uma chave em uma árvore B^+ de ordem n no pior caso? (considere a ordem como o número máximo de filhos de um nó).
2. Explique as árvores B^+ de prefixo simples.
3. Considere uma árvore B^+ com $n = 4$ filhos em que todos os elementos estão presentes em páginas folha. Considere o algoritmo de divisão de páginas padrão. Apresente graficamente as seguintes operações de inserção na árvore: 13, 28, 36, 45, 52, 3, 4, 8, 17, 20, 10, 14, 25, 40, 29, 37, 50, 43, 30, 7, 16, 6. Apresente graficamente a remoção dos elementos 28, 10, 40, 45, 3, 14.

Referências

1. SILBERSCHATZ, A. **Sistema de Banco de Dados**. 7. ed. Rio de Janeiro: GEN LTC, 2020. Disponível em: https://integrada.minhabiblioteca.com.br/reader/books/9788595157552.

   Recomendação de leitura: capítulo 14 do livro (até Sec. 14.4, inclusive)

Os materiais desta seção foram gentilmente cedidos por Humberto L. Razente (FACOM/UFU).

Adaptações: Renato Pimentel, FACOM/UFU