# Algoritmos e Complexidade

## Aula 04: Estruturas de Dados Avançadas - Árvores, Hash Tables e Grafos

**Prof. Vagner Cordeiro**
**Sistemas de Informação**
**Universidade - 2024**

## Agenda da Aula

1. **Árvores Binárias: Fundamentos e Implementação**
2. **Árvores Binárias de Busca (BST)**
3. **Árvores Balanceadas: AVL e Red-Black**
4. **Hash Tables: Teoria e Implementação**
5. **Funções de Hash e Tratamento de Colisões**
6. **Grafos: Representação e Algoritmos Básicos**
7. **Algoritmos de Busca: DFS e BFS**
8. **Aplicações Práticas e Análise de Performance**

## Objetivos de Aprendizagem

### Ao final desta aula, o estudante será capaz de:

**Estruturas Hierárquicas:**

- **Implementar** árvores binárias e suas operações fundamentais
- **Analisar** complexidade de operações em árvores balanceadas e não balanceadas
- **Aplicar** algoritmos de balanceamento em estruturas críticas

**Hash Tables:**

- **Projetar** funções de hash eficientes para diferentes tipos de dados
- **Implementar** estratégias de resolução de colisões
- **Otimizar** performance através de análise de fator de carga

**Grafos:**

- **Representar** grafos usando listas de adjacência e matrizes
- **Implementar** algoritmos fundamentais de busca e travessia
- **Resolver** problemas práticos usando teoria de grafos

# 1. Árvores Binárias: Fundamentos Matemáticos

## Definição Formal

Uma **árvore binária** é uma estrutura hierárquica onde cada nó tem no máximo dois filhos:

$$T = (V, E)$$

Onde:

- $V$ = Conjunto de vértices (nós)
- $E$ = Conjunto de arestas (conexões pai-filho)
- Para cada nó $v \in V$: $|\text{filhos}(v)| \leq 2$

## Propriedades Matemáticas Fundamentais

**Altura e Nós:**

- **Altura máxima:** $h_{max} = n - 1$ (árvore degenerada)
- **Altura mínima:** $h_{min} = \lfloor \log_2 n \rfloor$ (árvore completa)
- **Número máximo de nós no nível** $i$: $2^i$
- **Número máximo de nós com altura** $h$: $2^{h+1} - 1$

**Relações Importantes:**

$$n_{folhas} = n_{internos} + 1$$

$$n_{total} = 2 \times n_{internos} + 1$$

# Implementação de Árvore Binária em C

## Estrutura Básica e Operações

```c
#include <stdio.h>
#include <stdlib.h>
#include <stdbool.h>

typedef struct NoArvore {
    int dados;
    struct NoArvore *esquerda;
    struct NoArvore *direita;
    int altura;  // Para árvores balanceadas
} NoArvore;

typedef struct {
    NoArvore *raiz;
    int tamanho;
    int altura_maxima;
} ArvoreBinaria;

// Criação de nó
NoArvore* criar_no(int valor) {
    NoArvore *novo = malloc(sizeof(NoArvore));
    if (novo != NULL) {
        novo->dados = valor;
        novo->esquerda = NULL;
        novo->direita = NULL;
        novo->altura = 0;
    }
    return novo;
}

// Inicialização da árvore
ArvoreBinaria* inicializar_arvore() {
    ArvoreBinaria *arvore = malloc(sizeof(ArvoreBinaria));
    if (arvore != NULL) {
        arvore->raiz = NULL;
        arvore->tamanho = 0;
        arvore->altura_maxima = 0;
    }
    return arvore;
}
```

# 2. Árvores Binárias de Busca (BST)

## Propriedade Fundamental

Para todo nó $x$ em uma BST:

$$\forall y \in \text{subárvore\_esquerda}(x) : y.valor < x.valor$$

$$\forall z \in \text{subárvore\_direita}(x) : z.valor > x.valor$$

## Operações Fundamentais

```c
// Busca em BST
NoArvore* buscar_bst(NoArvore *raiz, int valor) {
    // Caso base: árvore vazia ou valor encontrado
    if (raiz == NULL || raiz->dados == valor)
        return raiz;

    // Valor menor: busca na subárvore esquerda
    if (valor < raiz->dados)
        return buscar_bst(raiz->esquerda, valor);

    // Valor maior: busca na subárvore direita
    return buscar_bst(raiz->direita, valor);
}

// Inserção em BST
NoArvore* inserir_bst(NoArvore *raiz, int valor) {
    // Caso base: posição de inserção encontrada
    if (raiz == NULL)
        return criar_no(valor);

    // Determina direção da inserção
    if (valor < raiz->dados) {
        raiz->esquerda = inserir_bst(raiz->esquerda, valor);
    } else if (valor > raiz->dados) {
        raiz->direita = inserir_bst(raiz->direita, valor);
    }
    // Valor igual: não insere duplicatas

    return raiz;
}

// Encontra o menor valor (mais à esquerda)
NoArvore* encontrar_minimo(NoArvore *raiz) {
    while (raiz->esquerda != NULL)
        raiz = raiz->esquerda;
    return raiz;
}

// Remoção em BST
NoArvore* remover_bst(NoArvore *raiz, int valor) {
    if (raiz == NULL)
        return raiz;

    // Localiza o nó a ser removido
    if (valor < raiz->dados) {
        raiz->esquerda = remover_bst(raiz->esquerda, valor);
    } else if (valor > raiz->dados) {
        raiz->direita = remover_bst(raiz->direita, valor);
    } else {
        // Nó encontrado - casos de remoção

        // Caso 1: Nó folha ou com apenas um filho
```

# 3. Árvores AVL: Auto-Balanceamento

## Propriedade AVL

Para todo nó $x$:

$$|\text{altura}(\text{esquerda}(x)) - \text{altura}(\text{direita}(x))| \leq 1$$

## Rotações para Balanceamento

```c
// Calcula altura do nó
int altura_no(NoArvore *no) {
    if (no == NULL)
        return -1;
    return no->altura;
}

// Calcula fator de balanceamento
int fator_balanceamento(NoArvore *no) {
    if (no == NULL)
        return 0;
    return altura_no(no->esquerda) - altura_no(no->direita);
}

// Atualiza altura do nó
void atualizar_altura(NoArvore *no) {
    if (no != NULL) {
        int altura_esq = altura_no(no->esquerda);
        int altura_dir = altura_no(no->direita);
        no->altura = 1 + ((altura_esq > altura_dir) ? altura_esq : altura_dir);
    }
}

// Rotação à direita
NoArvore* rotacao_direita(NoArvore *y) {
    NoArvore *x = y->esquerda;
    NoArvore *T2 = x->direita;

    // Executa rotação
    x->direita = y;
    y->esquerda = T2;

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

    // Nova raiz
    return x;
}

// Rotação à esquerda
NoArvore* rotacao_esquerda(NoArvore *x) {
    NoArvore *y = x->direita;
    NoArvore *T2 = y->esquerda;

    // Executa rotação
    y->esquerda = x;
    x->direita = T2;

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

    // Nova raiz
    return y;
}

// Inserção AVL com balanceamento
NoArvore* inserir_avl(NoArvore *raiz, int valor) {
    // 1. Inserção normal de BST
    if (raiz == NULL)
        return criar_no(valor);

    if (valor < raiz->dados)
        raiz->esquerda = inserir_avl(raiz->esquerda, valor);
    else if (valor > raiz->dados)
        raiz->direita = inserir_avl(raiz->direita, valor);
    else  // Valores iguais não são permitidos
        return raiz;

    // 2. Atualiza altura do nó atual
    atualizar_altura(raiz);

    // 3. Obtém fator de balanceamento
    int balance = fator_balanceamento(raiz);

    // 4. Casos de rotação
```

# 4. Hash Tables: Fundamentos Teóricos

## Definição e Princípios

Uma **hash table** mapeia chaves para valores usando uma função de hash:

$$h : K \to \{0, 1, 2, \ldots, m-1\}$$

Onde:

- $K$ = Espaço de chaves (potencialmente infinito)
- $m$ = Tamanho da tabela (finito)
- $h(k)$ = Índice da tabela para a chave $k$

## Implementação Básica

```c
#define TAMANHO_TABELA 1009  // Número primo para melhor distribuição

typedef struct EntradaHash {
    char *chave;
    int valor;
    struct EntradaHash *proximo;  // Para encadeamento
} EntradaHash;

typedef struct {
    EntradaHash **tabela;
    int tamanho;
    int elementos;
    double fator_carga_maximo;
} TabelaHash;

// Inicialização da tabela hash
TabelaHash* criar_tabela_hash(int tamanho) {
    TabelaHash *tabela = malloc(sizeof(TabelaHash));
    if (tabela == NULL) return NULL;

    tabela->tamanho = tamanho;
    tabela->elementos = 0;
    tabela->fator_carga_maximo = 0.75;
```

# 5. Funções de Hash Avançadas

## Hash para Diferentes Tipos de Dados

```c
// Hash para inteiros (multiplicação)
unsigned int hash_inteiro(int chave, int tamanho_tabela) {
    const double A = 0.6180339887;  // (√5 - 1) / 2
    double produto = chave * A;
    double fracao = produto - (int)produto;
    return (int)(tamanho_tabela * fracao);
}

// Hash para strings (polinomial rolling hash)
unsigned long hash_polinomial(const char *str, int tamanho_tabela) {
    const int p = 31;  // Número primo
    const int m = 1e9 + 9;  // Módulo grande
    long long hash_value = 0;
    long long p_pow = 1;

    for (const char *c = str; *c; c++) {
        hash_value = (hash_value + (*c - 'a' + 1) * p_pow) % m;
        p_pow = (p_pow * p) % m;
    }

    return hash_value % tamanho_tabela;
}

// Hash para estruturas complexas
typedef struct {
    int id;
    char nome[50];
    double salario;
} Funcionario;

unsigned int hash_funcionario(const Funcionario *func, int tamanho_tabela) {
    unsigned int hash = 0;

    // Combina hash de diferentes campos
    hash ^= hash_inteiro(func->id, tamanho_tabela);
    hash ^= hash_polinomial(func->nome, tamanho_tabela) << 1;
    hash ^= hash_inteiro((int)(func->salario * 100), tamanho_tabela) << 2;

    return hash % tamanho_tabela;
}
```

# 6. Grafos: Representação e Algoritmos Fundamentais

## Definições Matemáticas

Um **grafo** é uma tupla $G = (V, E)$ onde:

- $V$ = Conjunto finito de vértices
- $E \subseteq V \times V$ = Conjunto de arestas

### Classificações:

- **Direcionado vs Não-direcionado**
- **Ponderado vs Não-ponderado**
- **Conectado vs Desconectado**
- **Cíclico vs Acíclico**

## Representações de Grafos

```c
#define MAX_VERTICES 100

// 1. Matriz de Adjacência
typedef struct {
    int matriz[MAX_VERTICES][MAX_VERTICES];
    int num_vertices;
    bool direcionado;
} GrafoMatriz;

// Inicialização matriz de adjacência
GrafoMatriz* criar_grafo_matriz(int vertices, bool direcionado) {
    GrafoMatriz *grafo = malloc(sizeof(GrafoMatriz));
    grafo->num_vertices = vertices;
    grafo->direcionado = direcionado;

    // Inicializa matriz com zeros
    for (int i = 0; i < vertices; i++) {
        for (int j = 0; j < vertices; j++) {
            grafo->matriz[i][j] = 0;
        }
    }

    return grafo;
}

// Adiciona aresta na matriz
void adicionar_aresta_matriz(GrafoMatriz *grafo, int origem, int destino) {
    if (origem >= 0 && origem < grafo->num_vertices &&
        destino >= 0 && destino < grafo->num_vertices) {

        grafo->matriz[origem][destino] = 1;
```

# 7. Algoritmos de Busca em Grafos

## Busca em Profundidade (DFS)

```c
void dfs_recursivo(GrafoLista *grafo, int vertice, bool visitado[]) {
    visitado[vertice] = true;
    printf("%d ", vertice);

    NoAdjacencia *adj = grafo->listas[vertice];
    while (adj != NULL) {
        if (!visitado[adj->vertice]) {
            dfs_recursivo(grafo, adj->vertice, visitado);
        }
        adj = adj->proximo;
    }
}

// DFS iterativo usando pilha
void dfs_iterativo(GrafoLista *grafo, int inicio) {
    bool visitado[MAX_VERTICES] = {false};
    int pilha[MAX_VERTICES];
    int topo = -1;

    // Empilha vértice inicial
    pilha[++topo] = inicio;

    while (topo >= 0) {
        int vertice = pilha[topo--];

        if (!visitado[vertice]) {
            visitado[vertice] = true;
            printf("%d ", vertice);

            // Adiciona vizinhos não visitados à pilha
            NoAdjacencia *adj = grafo->listas[vertice];
            while (adj != NULL) {
                if (!visitado[adj->vertice]) {
                    pilha[++topo] = adj->vertice;
                }
                adj = adj->proximo;
            }
        }
    }
}

// Aplicação: Detecção de ciclos
bool tem_ciclo_dfs(GrafoLista *grafo) {
    bool visitado[MAX_VERTICES] = {false};
    bool pilha_recursao[MAX_VERTICES] = {false};

    for (int i = 0; i < grafo->num_vertices; i++) {
        if (!visitado[i]) {
            if (dfs_ciclo_util(grafo, i, visitado, pilha_recursao)) {
                return true;
            }
        }
    }
    return false;
}

bool dfs_ciclo_util(GrafoLista *grafo, int vertice,
                    bool visitado[], bool pilha_recursao[]) {
    visitado[vertice] = true;
    pilha_recursao[vertice] = true;

    NoAdjacencia *adj = grafo->listas[vertice];
    while (adj != NULL) {
        if (!visitado[adj->vertice]) {
            if (dfs_ciclo_util(grafo, adj->vertice, visitado, pilha_recursao)) {
                return true;
            }
        } else if (pilha_recursao[adj->vertice]) {
            return true;  // Ciclo encontrado
        }
        adj = adj->proximo;
    }

    pilha_recursao[vertice] = false;
    return false;
}
```

# 8. Aplicações Práticas e Sistemas Reais

## Sistema de Recomendação usando Grafos

```c
typedef struct {
    int usuario_id;
    char nome[100];
    int interesses[10];  // IDs de categorias de interesse
    int num_interesses;
} Usuario;

typedef struct {
    Usuario usuarios[1000];
    GrafoLista *grafo_amizades;
    TabelaHash *tabela_usuarios;
    int num_usuarios;
} RedeSocial;

// Calcula similaridade entre usuários
double calcular_similaridade(Usuario *u1, Usuario *u2) {
    int interesses_comuns = 0;
    int total_interesses = u1->num_interesses + u2->num_interesses;

    for (int i = 0; i < u1->num_interesses; i++) {
        for (int j = 0; j < u2->num_interesses; j++) {
            if (u1->interesses[i] == u2->interesses[j]) {
                interesses_comuns++;
                break;
            }
        }
    }

    if (total_interesses == 0) return 0.0;
    return (2.0 * interesses_comuns) / total_interesses;
}

// Recomenda amigos baseado em amigos de amigos
void recomendar_amigos(RedeSocial *rede, int usuario_id) {
    bool visitado[MAX_VERTICES] = {false};
    int candidatos[MAX_VERTICES];
    double scores[MAX_VERTICES];
    int num_candidatos = 0;

    // BFS de profundidade 2 para encontrar amigos de amigos
    NoAdjacencia *amigos = rede->grafo_amizades->listas[usuario_id];

    while (amigos != NULL) {
        int amigo_id = amigos->vertice;
        NoAdjacencia *amigos_do_amigo = rede->grafo_amizades->listas[amigo_id];

        while (amigos_do_amigo != NULL) {
            int candidato_id = amigos_do_amigo->vertice;

            if (candidato_id != usuario_id && !visitado[candidato_id]) {
                visitado[candidato_id] = true;
                candidatos[num_candidatos] = candidato_id;

                // Calcula score baseado em similaridade
                scores[num_candidatos] = calcular_similaridade(
                    &rede->usuarios[usuario_id],
                    &rede->usuarios[candidato_id]
                );

                num_candidatos++;
            }
            amigos_do_amigo = amigos_do_amigo->proximo;
        }
        amigos = amigos->proximo;
    }

    // Ordena candidatos por score
    for (int i = 0; i < num_candidatos - 1; i++) {
        for (int j = 0; j < num_candidatos - i - 1; j++) {
            if (scores[j] < scores[j + 1]) {
                // Troca scores
                double temp_score = scores[j];
                scores[j] = scores[j + 1];
                scores[j + 1] = temp_score;

                // Troca candidatos
                int temp_id = candidatos[j];
                candidatos[j] = candidatos[j + 1];
                candidatos[j + 1] = temp_id;
            }
        }
    }

    // Mostra top 5 recomendações
    printf("Recomendações de amizade para %s:\n",
           rede->usuarios[usuario_id].nome);
    int limite = (num_candidatos < 5) ? num_candidatos : 5;
    for (int i = 0; i < limite; i++) {
        printf("%d. %s (score: %.2f)\n",
               i + 1,
               rede->usuarios[candidatos[i]].nome,
               scores[i]);
    }
```

## 9. Conclusões e Próximos Passos

### Resumo de Estruturas de Dados

| Estrutura | Busca | Inserção | Remoção | Uso Ideal |
|---|---|---|---|---|
| Array | $O(n)$ | $O(n)$ | $O(n)$ | Acesso sequencial |
| BST | $O(\log n)$ | $O(\log n)$ | $O(\log n)$ | Dados ordenados |
| AVL | $O(\log n)$ | $O(\log n)$ | $O(\log n)$ | Busca frequente |
| Hash Table | $O(1)$ | $O(1)$ | $O(1)$ | Acesso aleatório |
| Grafo (Lista) | $O(V + E)$ | $O(1)$ | $O(V)$ | Relações complexas |

### Preparação para Aulas Futuras

**Aula 05:** Algoritmos de Grafos Avançados

- Algoritmos de caminho mínimo (Dijkstra, Floyd-Warshall)
- Árvores de espalhamento mínimo (Kruskal, Prim)
- Ordenação topológica e componentes fortemente conexos

**Aula 06:** Programação Dinâmica

- Princípios de otimalidade e subestrutura
- Memoização vs tabulação
- Problemas clássicos: fibonacci, knapsack, LCS

## Bibliografia e Recursos Avançados

### Livros Especializados

- **Weiss, M. A.** *Data Structures and Algorithm Analysis in C*
- **Sedgewick, R.** *Algorithms in C++*, Parts 1-5
- **Skiena, S. S.** *The Algorithm Design Manual*

### Ferramentas de Visualização

- **VisuAlgo:** Animações interativas de estruturas de dados
- **Data Structure Visualizations:** USF Computer Science
- **Algorithm Visualizer:** Implementações visuais

### Implementações de Referência

- **GNU C Library:** Implementações otimizadas
- **OpenJDK Collections:** Java standard library
- **Boost C++ Libraries:** Estruturas avançadas

# Encerramento da Aula

## Algoritmos e Complexidade - Aula 04

*Estruturas de Dados Avançadas - Árvores, Hash Tables e Grafos*

**Próxima Aula:** Algoritmos de Grafos Avançados e Caminhos Mínimos
**Projeto Prático:** Implementar sistema de cache LRU completo

### Exercícios Recomendados

1. Implementar AVL tree com todas as operações
2. Criar hash table com redimensionamento dinâmico
3. Desenvolver sistema de navegação usando BFS/DFS