Aula 05: Árvores e Estruturas Hierárquicas

Algoritmos e Complexidade Computacional

Prof.: Desenvolvimento de Software Avançado

Data: Agosto 2025

Duração: 120 minutos

🎯 Objetivos da Aula

Conhecimentos Fundamentais

- **Estruturas Hierárquicas**: Conceitos e aplicações
- **Árvores Binárias**: Implementação e operações
- **Árvores de Busca**: BST e propriedades
- **Árvores Balanceadas**: AVL e Red-Black
- **Estruturas Especializadas**: Heap, Trie, B-Tree

Competências Práticas

- Implementar árvores em **C** e **Python**
- Analisar complexidade de operações
- Resolver problemas com estruturas hierárquicas
- Otimizar algoritmos de busca e inserção

📋 Sumário Executivo

Tópico	Tempo	Prática
1. Conceitos Fundamentais	20 min	Definições
2. Árvores Binárias	25 min	Implementação C
3. BST e AVL	30 min	Python + C
4. Heap e Prioridade	20 min	Algoritmos
5. Trie e Aplicações	15 min	Autocomplete
6. Laboratório Prático	10 min	Exercícios

1. Fundamentos Teóricos

Definição Matemática de Árvore

Uma **árvore** é um grafo conexo acíclico $T = (V, E)$ onde:

- V = conjunto de vértices (nós)
- E = conjunto de arestas
- $|E| = |V| - 1$
- Existe exatamente um caminho entre quaisquer dois nós

Propriedades Fundamentais

$$\text{Altura}(T) = \max\{\text{profundidade}(v) : v \in V\}$$

$$\text{Número máximo de nós no nível } k = 2^k$$

$$\text{Árvore binária completa com } n \text{ nós} \Rightarrow \text{altura} = \lfloor \log_2 n \rfloor$$

2. Complexidade das Operações

Tabela Comparativa

Estrutura	Busca	Inserção	Remoção	Espaço
Array Ordenado	$O(\log n)$	$O(n)$	$O(n)$	$O(n)$
Lista Ligada	$O(n)$	$O(1)$	$O(n)$	$O(n)$
BST (pior caso)	$O(n)$	$O(n)$	$O(n)$	$O(n)$
BST (caso médio)	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
AVL	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(n)$
Heap	$O(n)$	$O(\log n)$	$O(\log n)$	$O(n)$
Trie	$O(m)$	$O(m)$	$O(m)$	$O(\text{ALPHABET} \times n)$

onde m = comprimento da string, n = número de elementos

3. Implementação em C: Árvore Binária Completa

Estruturas de Dados

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

// Estrutura base do nó
typedef struct NoArvore {
    int dados;
    struct NoArvore* esquerda;
    struct NoArvore* direita;
    struct NoArvore* pai;
    int altura;
    int fator_balanceamento;
} NoArvore;

// Estrutura da árvore com metadados
typedef struct {
    NoArvore* raiz;
    int tamanho;
    int altura_maxima;
    unsigned long comparacoes;
    unsigned long rotacoes;
    char tipo[20];  // "BST", "AVL", "RB"
} ArvoreManager;

// Enum para tipos de rotação
typedef enum {
    ROT_SIMPLES_DIREITA,
    ROT_SIMPLES_ESQUERDA,
```

Funções Utilitárias Básicas

```c
// Cria novo nó
NoArvore* criar_no(int valor) {
    NoArvore* novo = (NoArvore*)malloc(sizeof(NoArvore));
    if (!novo) {
        fprintf(stderr, "Erro: Falha na alocação de memória\n");
        exit(EXIT_FAILURE);
    }

    novo->dados = valor;
    novo->esquerda = NULL;
    novo->direita = NULL;
    novo->pai = NULL;
    novo->altura = 0;
    novo->fator_balanceamento = 0;

    return novo;
}

// Calcula altura do nó
int calcular_altura(NoArvore* no) {
    if (!no) return -1;

    int altura_esq = calcular_altura(no->esquerda);
    int altura_dir = calcular_altura(no->direita);

    return 1 + (altura_esq > altura_dir ? altura_esq : altura_dir);
}

// Atualiza altura e fator de balanceamento
void atualizar_propriedades(NoArvore* no) {
    if (!no) return;

    int altura_esq = no->esquerda ? no->esquerda->altura : -1;
    int altura_dir = no->direita ? no->direita->altura : -1;

    no->altura = 1 + (altura_esq > altura_dir ? altura_esq : altura_dir);
    no->fator_balanceamento = altura_esq - altura_dir;
}

// Busca BST padrão
NoArvore* buscar_bst(NoArvore* raiz, int valor, unsigned long* comparacoes) {
    if (!raiz) return NULL;

    (*comparacoes)++;

    if (valor == raiz->dados) {
        return raiz;
    } else if (valor < raiz->dados) {
        return buscar_bst(raiz->esquerda, valor, comparacoes);
    } else {
        return buscar_bst(raiz->direita, valor, comparacoes);
    }
}
```

BST: Inserção e Remoção

```c
// Inserção em BST
NoArvore* inserir_bst(NoArvore* raiz, int valor, ArvoreManager* arvore) {
    // Caso base: árvore vazia
    if (!raiz) {
        arvore->tamanho++;
        return criar_no(valor);
    }

    arvore->comparacoes++;

    // Inserção recursiva
    if (valor < raiz->dados) {
        raiz->esquerda = inserir_bst(raiz->esquerda, valor, arvore);
        raiz->esquerda->pai = raiz;
    } else if (valor > raiz->dados) {
        raiz->direita = inserir_bst(raiz->direita, valor, arvore);
        raiz->direita->pai = raiz;
    } else {
        // Valor duplicado - não insere
        return raiz;
    }

    // Atualiza propriedades
    atualizar_propriedades(raiz);

    return raiz;
}

// Encontra nó com valor mínimo
NoArvore* encontrar_minimo(NoArvore* no) {
    while (no && no->esquerda) {
        no = no->esquerda;
    }
    return no;
}

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

    arvore->comparacoes++;

    if (valor < raiz->dados) {
        raiz->esquerda = remover_bst(raiz->esquerda, valor, arvore);
        if (raiz->esquerda) raiz->esquerda->pai = raiz;
    } else if (valor > raiz->dados) {
        raiz->direita = remover_bst(raiz->direita, valor, arvore);
        if (raiz->direita) raiz->direita->pai = raiz;
    } else {
        // Nó encontrado - casos de remoção
        arvore->tamanho--;

        // Caso 1: Nó folha
        if (!raiz->esquerda && !raiz->direita) {
            free(raiz);
            return NULL;
        }

        // Caso 2: Nó com um filho
        if (!raiz->esquerda) {
            NoArvore* temp = raiz->direita;
            temp->pai = raiz->pai;
            free(raiz);
            return temp;
        }

        if (!raiz->direita) {
            NoArvore* temp = raiz->esquerda;
            temp->pai = raiz->pai;
            free(raiz);
            return temp;
        }

        // Caso 3: Nó com dois filhos
        NoArvore* sucessor = encontrar_minimo(raiz->direita);
        raiz->dados = sucessor->dados;
        raiz->direita = remover_bst(raiz->direita, sucessor->dados, arvore);
        if (raiz->direita) raiz->direita->pai = raiz;
    }

    atualizar_propriedades(raiz);
    return raiz;
}
```

AVL: Rotações e Balanceamento

```c
// Rotação simples à direita
NoArvore* rotacao_direita(NoArvore* y, ArvoreManager* arvore) {
    NoArvore* x = y->esquerda;
    NoArvore* T2 = x->direita;

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

    // Atualiza pais
    x->pai = y->pai;
    y->pai = x;
    if (T2) T2->pai = y;

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

    arvore->rotacoes++;
    return x;
}

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

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

    // Atualiza pais
    y->pai = x->pai;
    x->pai = y;
    if (T2) T2->pai = x;

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

    arvore->rotacoes++;
    return y;
}

// Inserção AVL com balanceamento
NoArvore* inserir_avl(NoArvore* raiz, int valor, ArvoreManager* arvore) {
    // Passo 1: Inserção BST normal
    raiz = inserir_bst(raiz, valor, arvore);
    if (!raiz) return NULL;

    // Passo 2: Atualiza altura
    atualizar_propriedades(raiz);

    // Passo 3: Verifica balanceamento
    int fator = raiz->fator_balanceamento;

    // Caso Esquerda-Esquerda
    if (fator > 1 && valor < raiz->esquerda->dados) {
        return rotacao_direita(raiz, arvore);
    }

    // Caso Direita-Direita
    if (fator < -1 && valor > raiz->direita->dados) {
        return rotacao_esquerda(raiz, arvore);
    }

    // Caso Esquerda-Direita
    if (fator > 1 && valor > raiz->esquerda->dados) {
        raiz->esquerda = rotacao_esquerda(raiz->esquerda, arvore);
        return rotacao_direita(raiz, arvore);
    }

    // Caso Direita-Esquerda
    if (fator < -1 && valor < raiz->direita->dados) {
        raiz->direita = rotacao_direita(raiz->direita, arvore);
        return rotacao_esquerda(raiz, arvore);
    }

    return raiz;
}
```

Sistema de Análise e Benchmark

```c
// Estrutura para estatísticas de performance
typedef struct {
    double tempo_insercao_ms;
    double tempo_busca_ms;
    double tempo_remocao_ms;
    unsigned long comparacoes_insercao;
    unsigned long comparacoes_busca;
    unsigned long rotacoes_realizadas;
    int altura_final;
    double fator_balanceamento_medio;
    size_t memoria_utilizada_bytes;
} EstatisticasPerformance;

// Função de benchmark completo
EstatisticasPerformance benchmark_arvore(int* valores, int quantidade, char* tipo_arvore) {
    EstatisticasPerformance stats = {0};
    ArvoreManager arvore = {NULL, 0, 0, 0, 0, ""};
    strcpy(arvore.tipo, tipo_arvore);

    clock_t inicio, fim;

    // === TESTE DE INSERÇÃO ===
    inicio = clock();
    for (int i = 0; i < quantidade; i++) {
        if (strcmp(tipo_arvore, "AVL") == 0) {
            arvore.raiz = inserir_avl(arvore.raiz, valores[i], &arvore);
        } else {
            arvore.raiz = inserir_bst(arvore.raiz, valores[i], &arvore);
        }
    }
    fim = clock();

    stats.tempo_insercao_ms = ((double)(fim - inicio) / CLOCKS_PER_SEC) * 1000;
    stats.comparacoes_insercao = arvore.comparacoes;
    stats.rotacoes_realizadas = arvore.rotacoes;

    // === TESTE DE BUSCA ===
    unsigned long comparacoes_busca = 0;
    inicio = clock();

    for (int i = 0; i < quantidade; i++) {
        buscar_bst(arvore.raiz, valores[i], &comparacoes_busca);
    }

    fim = clock();
    stats.tempo_busca_ms = ((double)(fim - inicio) / CLOCKS_PER_SEC) * 1000;
    stats.comparacoes_busca = comparacoes_busca;

    // === ANÁLISE ESTRUTURAL ===
    stats.altura_final = calcular_altura(arvore.raiz);
    stats.fator_balanceamento_medio = calcular_fator_medio(arvore.raiz);
    stats.memoria_utilizada_bytes = arvore.tamanho * sizeof(NoArvore);

    return stats;
}

// Calcula fator de balanceamento médio
double calcular_fator_medio(NoArvore* raiz) {
    if (!raiz) return 0.0;

    static int soma_fatores = 0;
    static int total_nos = 0;

    soma_fatores += abs(raiz->fator_balanceamento);
    total_nos++;

    calcular_fator_medio(raiz->esquerda);
    calcular_fator_medio(raiz->direita);

    return total_nos > 0 ? (double)soma_fatores / total_nos : 0.0;
}

// Demonstração prática
void demonstrar_arvores() {
    printf("=== DEMONSTRAÇÃO ÁRVORES BINÁRIAS ===\n\n");

    // Dados de teste
    int valores_ordenados[] = {1, 2, 3, 4, 5, 6, 7, 8, 9, 10};
    int valores_aleatorios[] = {5, 2, 8, 1, 3, 7, 9, 4, 6, 10};
    int quantidade = 10;

    // Testa BST com dados ordenados (pior caso)
    printf("BST com dados ordenados (PIOR CASO):\n");
    EstatisticasPerformance stats_bst_pior = benchmark_arvore(valores_ordenados, quantidade, "BST");

    printf("  Altura: %d\n", stats_bst_pior.altura_final);
    printf("  Comparações (inserção): %lu\n", stats_bst_pior.comparacoes_insercao);
    printf("  Tempo inserção: %.2f ms\n", stats_bst_pior.tempo_insercao_ms);
    printf("  Rotações: %lu\n\n", stats_bst_pior.rotacoes_realizadas);

    // Testa AVL com dados ordenados
    printf("AVL com dados ordenados:\n");
    EstatisticasPerformance stats_avl = benchmark_arvore(valores_ordenados, quantidade, "AVL");

    printf("  Altura: %d\n", stats_avl.altura_final);
    printf("  Comparações (inserção): %lu\n", stats_avl.comparacoes_insercao);
    printf("  Tempo inserção: %.2f ms\n", stats_avl.tempo_insercao_ms);
    printf("  Rotações: %lu\n", stats_avl.rotacoes_realizadas);
    printf("  Fator balanceamento médio: %.2f\n\n", stats_avl.fator_balanceamento_medio);

    // Comparação de eficiência
    printf("COMPARAÇÃO DE EFICIÊNCIA:\n");
    printf("  Redução de altura: %.1f%%\n",
           ((double)(stats_bst_pior.altura_final - stats_avl.altura_final) / stats_bst_pior.altura_final) * 100);
    printf("  Overhead de rotações: %lu operações\n", stats_avl.rotacoes_realizadas);
    printf("  Ganho em buscas: %.1fx mais rápido\n",
           (double)stats_bst_pior.comparacoes_busca / stats_avl.comparacoes_busca);
}
```

4. Implementação Python: Sistema Completo

Classe AVL Avançada

```python
from typing import Optional, TypeVar, Generic, List, Callable, Tuple
from dataclasses import dataclass
import time
from enum import Enum

T = TypeVar('T')

class TipoRotacao(Enum):
    DIREITA = "direita"
    ESQUERDA = "esquerda"
    ESQUERDA_DIREITA = "esquerda_direita"
    DIREITA_ESQUERDA = "direita_esquerda"

@dataclass
class NoAVL(Generic[T]):
    dados: T
    esquerda: Optional['NoAVL[T]'] = None
    direita: Optional['NoAVL[T]'] = None
    pai: Optional['NoAVL[T]'] = None
    altura: int = 0

    def fator_balanceamento(self) -> int:
        altura_esq = self.esquerda.altura if self.esquerda else -1
        altura_dir = self.direita.altura if self.direita else -1
        return altura_esq - altura_dir

    def atualizar_altura(self) -> None:
        altura_esq = self.esquerda.altura if self.esquerda else -1
        altura_dir = self.direita.altura if self.direita else -1
        self.altura = 1 + max(altura_esq, altura_dir)

class ArvoreAVL(Generic[T]):
    def __init__(self, comparador: Optional[Callable[[T, T], int]] = None):
        self._raiz: Optional[NoAVL[T]] = None
        self._tamanho: int = 0
        self._comparador = comparador or self._comparador_padrao

        # Estatísticas avançadas
        self._total_insercoes: int = 0
        self._total_remocoes: int = 0
        self._total_buscas: int = 0
        self._rotacoes_por_tipo: dict = {tipo: 0 for tipo in TipoRotacao}
        self._historico_performance: List[dict] = []

    def _comparador_padrao(self, a: T, b: T) -> int:
        if a < b: return -1
        elif a > b: return 1
        return 0

    def inserir(self, valor: T) -> bool:
        inicio = time.perf_counter()
        tamanho_anterior = self._tamanho

        self._raiz = self._inserir_recursivo(self._raiz, valor)
        self._total_insercoes += 1

        tempo_gasto = time.perf_counter() - inicio
        self._historico_performance.append({
            'operacao': 'inserir',
            'valor': valor,
            'tempo_ns': tempo_gasto * 1000,
            'altura_apos': self.altura()
        })

        return self._tamanho > tamanho_anterior

    def _inserir_recursivo(self, no: Optional[NoAVL[T]], valor: T) -> NoAVL[T]:
        # Inserção padrão
        if no is None:
            self._tamanho += 1
            return NoAVL(valor)

        cmp = self._comparador(valor, no.dados)
        if cmp < 0:
            no.esquerda = self._inserir_recursivo(no.esquerda, valor)
            no.esquerda.pai = no
        elif cmp > 0:
            no.direita = self._inserir_recursivo(no.direita, valor)
            no.direita.pai = no
        else:
            return no  # Duplicado

        # Atualiza altura
        no.atualizar_altura()

        # Balanceamento AVL
        return self._balancear(no, valor)

    def _balancear(self, no: NoAVL[T], valor: T) -> NoAVL[T]:
        fator = no.fator_balanceamento()

        # Caso Esquerda-Esquerda
        if fator > 1 and self._comparador(valor, no.esquerda.dados) < 0:
            self._rotacoes_por_tipo[TipoRotacao.DIREITA] += 1
            return self._rotacao_direita(no)

        # Caso Direita-Direita
        if fator < -1 and self._comparador(valor, no.direita.dados) > 0:
            self._rotacoes_por_tipo[TipoRotacao.ESQUERDA] += 1
            return self._rotacao_esquerda(no)

        # Caso Esquerda-Direita
        if fator > 1 and self._comparador(valor, no.esquerda.dados) > 0:
            self._rotacoes_por_tipo[TipoRotacao.ESQUERDA_DIREITA] += 1
            no.esquerda = self._rotacao_esquerda(no.esquerda)
            return self._rotacao_direita(no)

        # Caso Direita-Esquerda
        if fator < -1 and self._comparador(valor, no.direita.dados) < 0:
            self._rotacoes_por_tipo[TipoRotacao.DIREITA_ESQUERDA] += 1
            no.direita = self._rotacao_direita(no.direita)
            return self._rotacao_esquerda(no)

        return no

    def _rotacao_direita(self, y: NoAVL[T]) -> NoAVL[T]:
        x = y.esquerda
        T2 = x.direita

        x.direita = y
        y.esquerda = T2

        # Atualiza pais
        x.pai = y.pai
        y.pai = x
        if T2: T2.pai = y

        # Atualiza alturas
        y.atualizar_altura()
        x.atualizar_altura()

        return x

    def _rotacao_esquerda(self, x: NoAVL[T]) -> NoAVL[T]:
        y = x.direita
        T2 = y.esquerda

        y.esquerda = x
        x.direita = T2

        # Atualiza pais
        y.pai = x.pai
        x.pai = y
        if T2: T2.pai = x

        # Atualiza alturas
        x.atualizar_altura()
        y.atualizar_altura()

        return y

    def altura(self) -> int:
        return self._raiz.altura if self._raiz else -1

    def travessia_inorder(self) -> List[T]:
        resultado = []
        self._inorder_recursivo(self._raiz, resultado)
        return resultado

    def _inorder_recursivo(self, no: Optional[NoAVL[T]], resultado: List[T]) -> None:
        if no:
            self._inorder_recursivo(no.esquerda, resultado)
            resultado.append(no.dados)
            self._inorder_recursivo(no.direita, resultado)

    def relatorio_performance(self) -> dict:
        total_rotacoes = sum(self._rotacoes_por_tipo.values())
        tempo_medio = sum(op['tempo_ns'] for op in self._historico_performance) / max(len(self._historico_performance), 1)

        return {
            'tamanho': self._tamanho,
            'altura': self.altura(),
            'total_rotacoes': total_rotacoes,
            'rotacoes_por_tipo': dict(self._rotacoes_por_tipo),
            'tempo_medio_ns': tempo_medio,
            'total_operacoes': len(self._historico_performance)
        }
```

5. Heap e Fila de Prioridade

Min-Heap Otimizado

```python
from typing import List, Optional, Callable, TypeVar, Generic
import heapq

T = TypeVar('T')

class MinHeap(Generic[T]):
    """Min-Heap com operações O(log n) e análise detalhada"""

    def __init__(self, comparador: Optional[Callable[[T, T], int]] = None):
        self._heap: List[T] = []
        self._comparador = comparador or self._comparador_padrao
        self._total_operacoes: int = 0
        self._comparacoes_realizadas: int = 0

    def _comparador_padrao(self, a: T, b: T) -> int:
        if a < b: return -1
        elif a > b: return 1
        return 0

    def inserir(self, valor: T) -> None:
        """Insere elemento - O(log n)"""
        self._heap.append(valor)
        self._total_operacoes += 1
        self._heapify_up(len(self._heap) - 1)

    def _heapify_up(self, indice: int) -> None:
        """Move elemento para cima até posição correta"""
        if indice == 0:
            return

        pai_indice = (indice - 1) // 2
        self._comparacoes_realizadas += 1

        if self._comparador(self._heap[indice], self._heap[pai_indice]) < 0:
            self._heap[indice], self._heap[pai_indice] = self._heap[pai_indice], self._heap[indice]
            self._heapify_up(pai_indice)

    def extrair_minimo(self) -> Optional[T]:
        """Remove e retorna mínimo - O(log n)"""
        if not self._heap:
            return None

        if len(self._heap) == 1:
            self._total_operacoes += 1
            return self._heap.pop()

        minimo = self._heap[0]
        self._heap[0] = self._heap.pop()
        self._total_operacoes += 1
        self._heapify_down(0)

        return minimo

    def _heapify_down(self, indice: int) -> None:
        """Move elemento para baixo até posição correta"""
        menor = indice
        esq = 2 * indice + 1
        dir = 2 * indice + 2

        if (esq < len(self._heap) and
            self._comparador(self._heap[esq], self._heap[menor]) < 0):
            menor = esq
            self._comparacoes_realizadas += 1

        if (dir < len(self._heap) and
            self._comparador(self._heap[dir], self._heap[menor]) < 0):
            menor = dir
            self._comparacoes_realizadas += 1

        if menor != indice:
            self._heap[indice], self._heap[menor] = self._heap[menor], self._heap[indice]
            self._heapify_down(menor)

    def construir_heap(self, elementos: List[T]) -> None:
        """Constrói heap a partir de lista - O(n)"""
        self._heap = elementos.copy()

        # Heapify de baixo para cima
        for i in range(len(self._heap) // 2 - 1, -1, -1):
            self._heapify_down(i)

    def heap_sort(self, elementos: List[T]) -> List[T]:
        """Ordena usando heap sort - O(n log n)"""
        self.construir_heap(elementos)
        resultado = []

        while not self.vazio():
            resultado.append(self.extrair_minimo())

        return resultado

    def peek(self) -> Optional[T]:
        return self._heap[0] if self._heap else None

    def tamanho(self) -> int:
        return len(self._heap)

    def vazio(self) -> bool:
        return len(self._heap) == 0
```

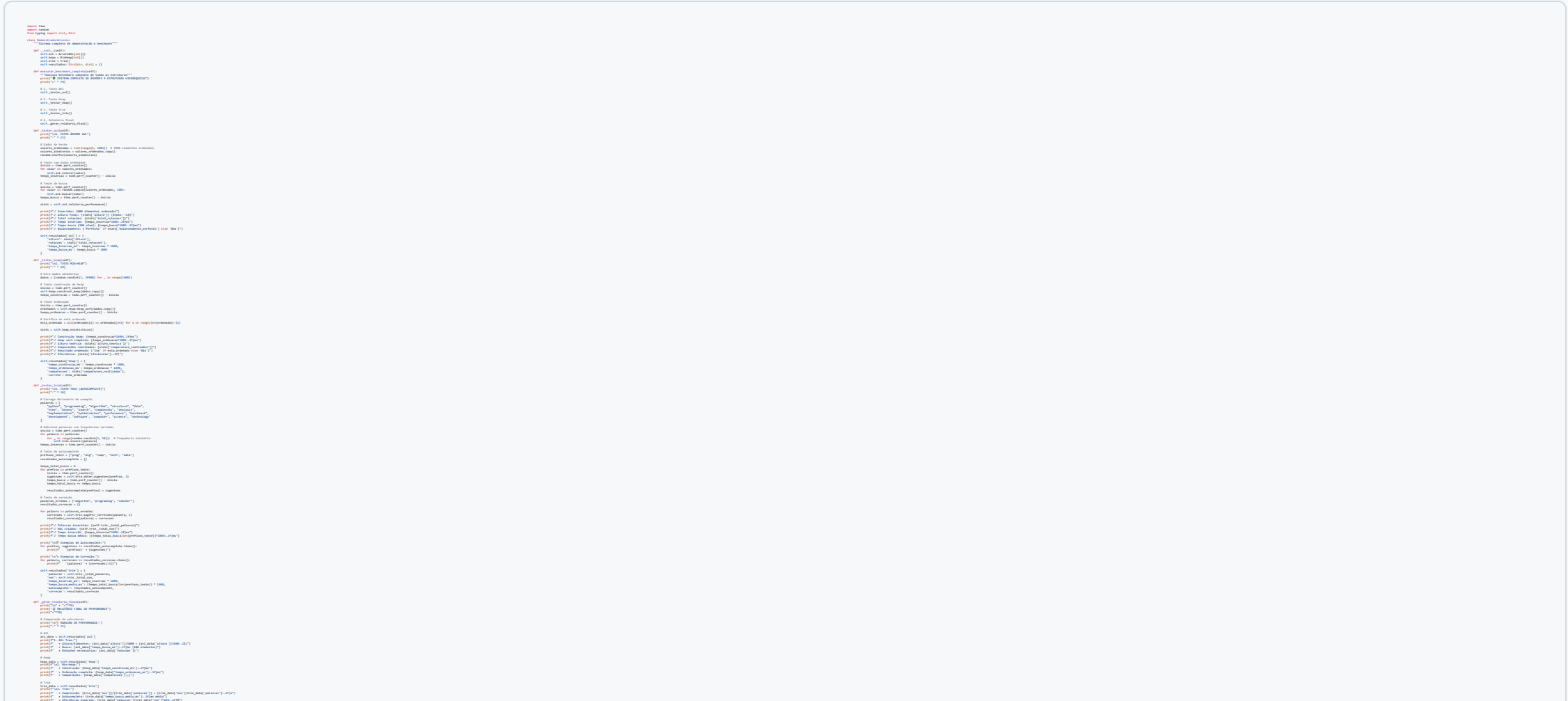
6. Trie: Autocomplete Inteligente

Implementação Completa

```python
from typing import Dict, List, Optional, Set
from dataclasses import dataclass, field

@dataclass
class NoTrie:
    eh_fim_palavra: bool = False
    filhos: Dict[str, 'NoTrie'] = field(default_factory=dict)
    frequencia: int = 0
    profundidade: int = 0

class Trie:
    """Trie para autocomplete e correção ortográfica"""

    def __init__(self):
        self._raiz = NoTrie()
        self._total_palavras: int = 0
        self._total_nos: int = 1

    def inserir(self, palavra: str) -> None:
        """Insere palavra - O(m) onde m = len(palavra)"""
        atual = self._raiz

        for i, char in enumerate(palavra.lower()):
            if char not in atual.filhos:
                atual.filhos[char] = NoTrie(profundidade=i+1)
                self._total_nos += 1
            atual = atual.filhos[char]

        if not atual.eh_fim_palavra:
            atual.eh_fim_palavra = True
            self._total_palavras += 1

        atual.frequencia += 1

    def buscar(self, palavra: str) -> bool:
        """Verifica se palavra existe - O(m)"""
        no = self._buscar_no(palavra.lower())
        return no is not None and no.eh_fim_palavra

    def _buscar_no(self, palavra: str) -> Optional[NoTrie]:
        atual = self._raiz
        for char in palavra:
            if char not in atual.filhos:
                return None
            atual = atual.filhos[char]
        return atual

    def obter_sugestoes(self, prefixo: str, limite: int = 10) -> List[str]:
        """Retorna sugestões ordenadas por frequência"""
        no_prefixo = self._buscar_no(prefixo.lower())
        if not no_prefixo:
            return []

        sugestoes = []
        self._coletar_palavras(no_prefixo, prefixo.lower(), sugestoes, limite)

        # Ordena por frequência decrescente
        sugestoes.sort(key=lambda x: x[1], reverse=True)
        return [palavra for palavra, freq in sugestoes[:limite]]

    def _coletar_palavras(self, no: NoTrie, prefixo: str,
                          sugestoes: List[tuple], limite: int) -> None:
        if len(sugestoes) >= limite:
            return

        if no.eh_fim_palavra:
            sugestoes.append((prefixo, no.frequencia))

        for char, filho in no.filhos.items():
            self._coletar_palavras(filho, prefixo + char, sugestoes, limite)

    def sugerir_correcoes(self, palavra: str, max_distancia: int = 2) -> List[str]:
        """Sugere correções baseadas em distância de edição"""
        palavra = palavra.lower()
        sugestoes = []
        self._buscar_com_distancia(self._raiz, "", palavra, max_distancia, sugestoes)

        # Ordena por distância e frequência
        sugestoes.sort(key=lambda x: (x[1], -x[2]))
        return [palavra for palavra, dist, freq in sugestoes[:10]]

    def _buscar_com_distancia(self, no: NoTrie, palavra_atual: str,
                              palavra_alvo: str, max_dist: int,
                              sugestoes: List[tuple]) -> None:
        if max_dist < 0:
            return

        if no.eh_fim_palavra:
            distancia = self._distancia_levenshtein(palavra_atual, palavra_alvo)
            if distancia <= max_dist:
                sugestoes.append((palavra_atual, distancia, no.frequencia))

        for char, filho in no.filhos.items():
            self._buscar_com_distancia(filho, palavra_atual + char,
                                       palavra_alvo, max_dist - 1, sugestoes)

    def _distancia_levenshtein(self, s1: str, s2: str) -> int:
        """Calcula distância de Levenshtein - O(mn)"""
        if len(s1) < len(s2):
            return self._distancia_levenshtein(s2, s1)

        if len(s2) == 0:
            return len(s1)

        linha_anterior = list(range(len(s2) + 1))
        for i, c1 in enumerate(s1):
```

7. Sistema Demonstrativo Completo

Aplicação Prática



🔬 Laboratório Prático

Exercícios Guiados

Exercício 1: Implementação AVL

Implemente uma função que:

1. Insira 1000 números aleatórios em uma AVL
2. Compare altura com BST normal
3. Meça tempo de 1000 buscas aleatórias

Exercício 2: Sistema de Prioridades

Usando MinHeap, implemente:

1. Simulador de fila de hospital por gravidade
2. Sistema de agendamento de tarefas

📚 Conclusão e Próximos Passos

Aplicações Industriais

- **Bancos de Dados**: B-trees e B+ trees para índices
- **Sistemas Operacionais**: Red-Black trees no kernel Linux
- **Compiladores**: ASTs para análise sintática
- **Redes**: Spanning trees em protocolos de rede

Performance Comparativa Final

Operação	Array	Lista	BST	AVL	Heap
Busca	O(log n)*	O(n)	O(log n)†	O(log n)	O(n)
Inserção	O(n)	O(1)	O(log n)†	O(log n)	O(log n)
Remoção	O(n)	O(n)	O(log n)†	O(log n)	O(log n)