UNIVERSIDADE DO ESTADO DE SANTA CATARINA - UDESC JOINVILLE
CENTRO DE CIÊNCIAS TECNOLÓGICAS
TECNOLOGIA EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS

HELLEN FIGUEIREDO RAMOS
LIZE ANA ZABOTE
VITOR MAURÍCIO DOS SANTOS ROSA

RELATÓRIO DA ANÁLISE DE COMPLEXIDADE ALGORÍTMICA DAS ESTRUTURAS DE ORGANIZAÇÃO DE DADOS: ÁRVORES

JOINVILLE
2025

RESUMO

Nessa atividade foi realizada uma análise para comparação do nível de complexidade algorítmica das operações de adição e remoção de nós nas estruturas de Árvores estudadas na disciplina, são elas: *árvores AVL, rubro-negra e B*. Para realizar esta análise, foi medido o esforço computacional de cada uma delas, considerando operações de navegação e balanceamento de nós. Os dados utilizados foram gerados aleatoriamente com valores inteiros. Ao final da execução, verificou-se que o esforço computacional para as operações de adição e remoção, todas as estruturas de árvores avaliadas apresentaram uma complexidade de ordem logarítmica. A Árvore Rubro-Negra demonstrou o desempenho mais eficiente, seguida pelas árvores B com um esforço intermediário, enquanto a Árvore AVL apresentou um esforço computacional maior que as demais.

1. INTRODUÇÃO

O objetivo do trabalho é analisar e avaliar a complexidade dos algoritmos de organização de dados de forma hierárquica, comparando os algorítmos de árvore B, com parâmetro de ordem igual à 1, 5 e 10, também como os de árvores AVL e Rubro-Negra.

A *Árvore AVL* é uma árvore binária de busca auto balanceada, ela garante que a diferença entre as alturas entre as subárvores esquerda e direita de qualquer nó nunca seja maior que 1, realizando esta organização através de rotações após inserções ou remoções.

A *Árvore Rubro-Negra* também é uma árvore binária de busca auto balanceada, porém ao invés de balancear pela altura, utiliza um esquema de colaboração de nós (vermelho ou preto) e regras específicas para manter o balanceamento, com o objetivo de garantir que o caminho mais longo de qualquer nó até uma folha não seja mais que o dobro do caminho mais curto.

A *Árvore B* é uma estrutura de dados de árvores muito utilizada em bancos de dados e sistemas de arquivos, caracterizada por ter múltiplos nós filhos e manter todos os nós folhas no mesmo nível, o que minimiza o número de acessos.

A atividade nos ajuda a entender a eficiência de métodos de organização hierárquica, permitindo identificar em quais situações práticas cada um deles pode ser mais vantajoso e adequado.

2. CÓDIGO E MÉTODOS

O código emprega a função `rand` para gerar conjuntos de números inteiros aleatórios, com tamanhos que variam de 1 a 10.000 (caso médio). Posteriormente, são executadas operações de inserção e remoção de dados. Para assegurar maior validade estatística e aproximar o teste do comportamento típico do algoritmo, utilizam-se 10 conjuntos de amostra, calculando a média dos resultados obtidos.

Para cada árvore foi implementado um contador que é incrementado a cada operação de comparação e/ou balanceamento (como rotações ou divisão de nós).

O código utilizado para a coleta de dados está disponível no *Anexo 1,* ao final deste documento.

Também disponível em: https://github.com/lize-zabote/comparador-arvores-binarias/.

3. RESULTADOS E DISCUSSÕES

Segue abaixo os gráficos obtidos através da coleta dos dados na execução do código.

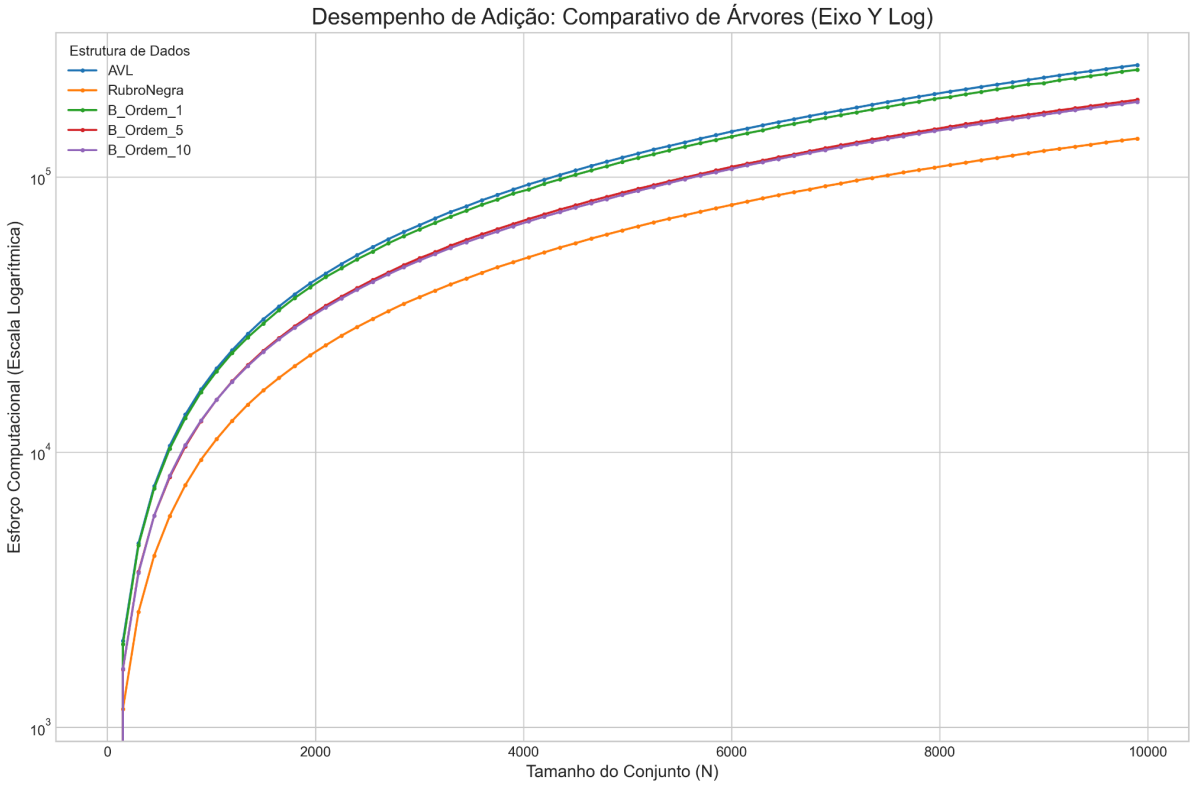

Figura 1. Gráfico de desempenho de adição

O gráfico de desempenho de adição mostra os diferentes comportamentos entre as estruturas, de acordo com o aumento do conjunto.

A *Árvore AVL*, representada pela linha azul, apresenta o maior esforço computacional para as inserções. Este comportamento é esperado devido ao rebalanceamento constante exigido após cada adição, visando manter a árvore perfeitamente balanceada. A AVL é eficiente para manter a ordem, mas à medida que o volume de inserções cresce, a mesma torna-se cada vez mais custosa.

A *Árvore B de ordem 1*, representada pela linha verde, se assemelha ao comportamento de uma árvore binária comum, tem um desempenho intermediário, possui um custo de inserção maior que as árvores B de ordens maiores, mas ainda assim consegue apresentar um esforço computacional menor que o da AVL, por não necessitar a balanceamento de toda a estrutura completa a cada nova inserção.

As *Árvores B de ordens 5 e 10*, representadas pelas linhas vermelha e roxa respectivamente, possuem um desempenho bastante eficiente para inserções. A árvore de ordem 5 apresenta um esforço um pouco maior do que a árvore de ordem 10, tendo em vista que: quanto maior a ordem, menor a altura da árvore, menos acessos e modificações, ótima para cenários com grande volumes de dados e inserções frequentes.

A *Rubro-Negra*, representada pela linha amarela, mostra o menor esforço computacional entre todas para a inserção. Isso se deve ao balanceamento mais flexível, reduzindo a quantidade de rotações, isso a torna bastante eficiente em inserções, especialmente em cenários de caso médio, onde os números são inseridos de forma aleatória, como o caso utilizado no estudo, através da função *rand()*. A *Rubro-Negra* combina um bom desempenho com uma estrutura simples de manter.

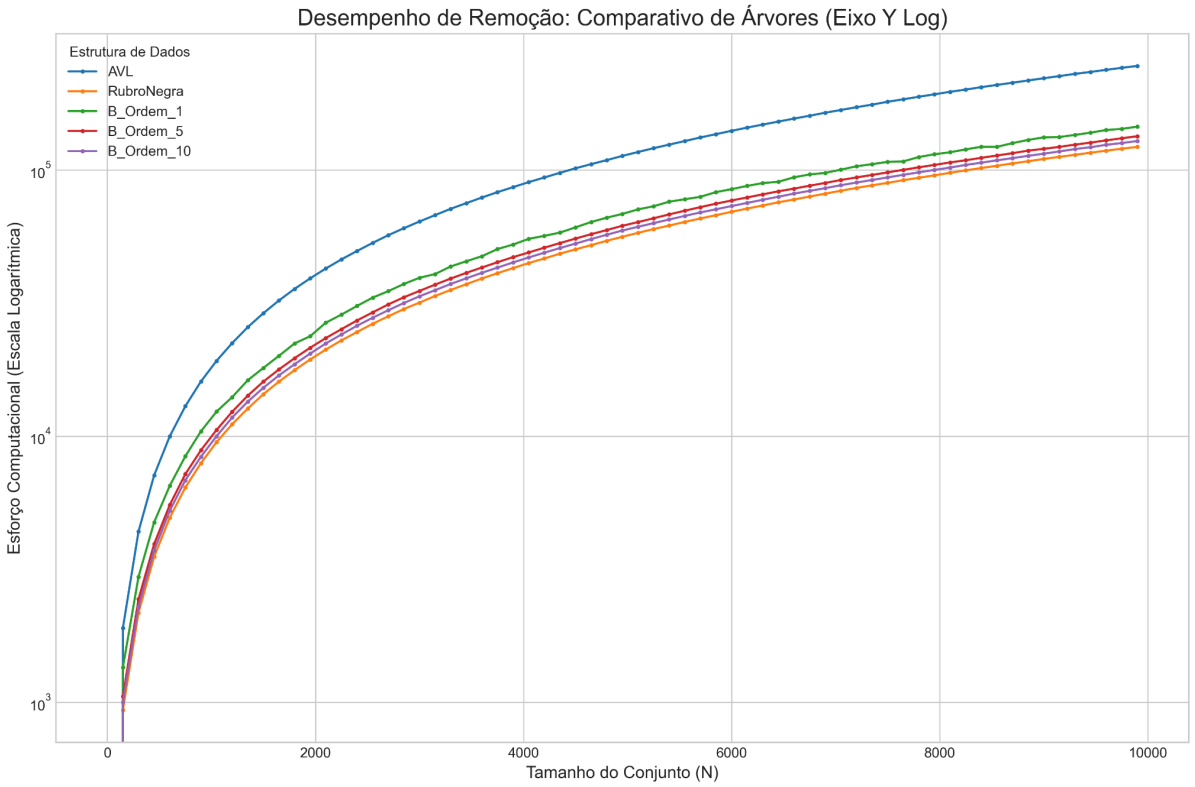

Figura 2. Gráfico de desempenho de remoção

No gráfico de remoção, assim como no de adição, a *Árvore AVL*, linha azul, apresenta o maior custo computacional. Como a operação de remoção demanda diversos ajustes e rotações devido ao balanceamento estrito, a árvore se torna menos eficiente, a distância das demais já é percebida desde os primeiros conjuntos, mas ao decorrer dos testes, com os conjuntos maiores é possível perceber essa diferença.

A *Árvore B de ordem 1*, linha verde, assim como na operação de adição mostra um desempenho razoável, melhor que o da AVL, mas não tão eficiente quanto as árvores B de ordem maior ou a Rubro-Negra.

As *Árvores B de ordens 5 e 10*, linhas vermelha e roxa respectivamente, apresentam um menor desempenho do que as árvores anteriores, devido aos seus nós que armazenam vários elementos e sua menor profundidade, elas exigem menos operações de rebalanceamento após uma exclusão. A Árvore B de ordem 10, mantém o melhor desempenho geral entre as de ordem menor, sendo assim a mais indicada para sistemas que precisem de remoções frequentes e de alta performance.

Por fim, a *Rubro-Negra*, linha amarela, continua mostrando o melhor desempenho entre todas as analisadas, devido ao balanceamento mais flexível. Sendo ideal também para sistemas que realizam remoções dinâmicas.

4. CONCLUSÃO

A análise comparativa entre as estruturas abordadas (AVL, Rubro-Negra e B de diferentes ordens) evidenciou o impacto dos balanceamentos e organização estrutural no desempenho computacional das operações de adição e remoção. Pois embora todas as estruturas garantam um desempenho logarítmico, conseguimos observar um comportamento ligeiramente distinto entre elas.

A Árvore AVL, oferece um ótimo desempenho para operações de busca, mas devido a sua necessidade de balanceamentos constante, apresenta o maior custo computacional, tanto em operações de adição, quanto nas de remoção, isso se destaca à medida que o tamanho do conjunto de dados cresce.

As Árvores B, especialmente com ordens maiores, apresentam um bom desempenho, devido a menor profundidade, a árvore tem menos acessos e modificações, o que a torna eficaz para lidar com muitos dados e inserções constantes. Apesar de não ser o caso utilizado no estudo, a Árvore B se tornaria muito mais eficiente em operações de classificação externa.

A Árvore Rubro-Negra mostrou um desempenho significativamente superior, com o esforço computacional reduzido, devido ao seu balanceamento flexível. Isso a torna menos eficiente para operações de busca em comparação à AVL, mas tornando-se a mais eficiente para operações de inserção e remoção de dados, conforme mostrado no estudo.

REFERÊNCIAS

LEITE, Allan Rodrigo. **Árvores AVL**. Joinville: UDESC, [s.d.]. Disponível em: https://moodle.joinville.udesc.br/pluginfile.php/372640/mod_resource/content/1/08%20-%20Árvores%20AVL.pdf. Acesso em: 19 jun. 2025.

LEITE, Allan Rodrigo. **Árvores rubro-negra**. Joinville: UDESC, [s.d.]. Disponível em: https://moodle.joinville.udesc.br/pluginfile.php/372643/mod_resource/content/1/09%20-%20Árvores%20rubro-negra.pdf. Acesso em: 19 jun. 2025.

LEITE, Allan Rodrigo. **Árvores B**. Joinville: UDESC, [s.d.]. Disponível em: https://moodle.joinville.udesc.br/pluginfile.php/372646/mod_resource/content/1/10%20-%20Árvores%20B.pdf. Acesso em: 19 jun. 2025.

GOMES, Rafael Beserra. **Algoritmos e Estruturas de Dados 2: Árvores Binárias**. Universidade Federal do Rio Grande do Norte, 2017. Disponível em: https://dimap.ufrn.br/~rafaelbg/material/aed/arvoresbinarias.pdf. Acesso em: 20 jun. 2025.

DIGIAMPIETRI, Luciano Antônio. **Árvores**. Universidade de São Paulo – USP, 2016. Disponível em: https://www.each.usp.br/digiampietri/SIN5013/16-arvores.pdf. Acesso em: 20 jun. 2025.

MANZANO, Guilherme. **Tudo o que você precisa saber sobre Algoritmos e Estrutura de Dados**. DIO, 06 out. 2023. Disponível em: https://www.dio.me/articles/tudo-o-que-voce-precisa-saber-sobre-algoritmos-e-estrutura-de-dados. Acesso em: 30 jun. 2025.

ANEXO 1

Código C utilizado para a coleta de dados:

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

void embaralhar(int *array, size_t n) {
    if (n > 1) {
        for (size_t i = n - 1; i > 0; i--) {
            size_t j = rand() % (i + 1);
            int temp = array[j];
            array[j] = array[i];
            array[i] = temp;
        }
    }
}

/* --- MÓDULO: ÁRVORE AVL --- */

int contAVL = 0;

typedef struct noAVL {
    struct noAVL* pai;
    struct noAVL* esquerda;
    struct noAVL* direita;
    int valor;
    int altura;
```

```c
} NoAVL;

typedef struct arvoreAVL {
    struct noAVL* raiz;
} ArvoreAVL;

ArvoreAVL* avl_criar();
void avl_destruir(ArvoreAVL* arvore);
void avl_adicionar(ArvoreAVL* arvore, int valor);
void avl_remover(ArvoreAVL* arvore, int valor);
int avl_altura(NoAVL* no);
int avl_fb(NoAVL* no);
void avl_percorrer(NoAVL* no, void (*callback)(int));
int avl_maximo(int a, int b);
void avl_balanceamento(ArvoreAVL* arvore, NoAVL* no);
NoAVL* avl_rsd(ArvoreAVL* arvore, NoAVL* no);
NoAVL* avl_rse(ArvoreAVL* arvore, NoAVL* no);
NoAVL* avl_rdd(ArvoreAVL* arvore, NoAVL* no);
NoAVL* avl_rde(ArvoreAVL* arvore, NoAVL* no);
NoAVL* avl_localizar(NoAVL* no, int valor);
void avl_removerNo(ArvoreAVL* arvore, NoAVL* noRemover);
NoAVL* avl_minimo(NoAVL* no);
void avl_destruir_no(NoAVL* no);

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

ArvoreAVL* avl_criar() {
    ArvoreAVL *arvore = malloc(sizeof(ArvoreAVL));
```

```
    arvore->raiz = NULL;
    return arvore;
}

int avl_altura(NoAVL* no) {
    return no != NULL ? no->altura : 0;
}

int avl_fb(NoAVL* no) {
    if (no == NULL) return 0;
    return avl_altura(no->esquerda) - avl_altura(no->direita);
}

void avl_adicionar(ArvoreAVL* arvore, int valor) {
    NoAVL* pai = NULL;
    NoAVL* atual = arvore->raiz;

    while (atual != NULL) {
        contAVL++;
        pai = atual;
        if (valor > atual->valor) {
            atual = atual->direita;
        } else {
            atual = atual->esquerda;
        }
    }

    NoAVL* novo = malloc(sizeof(NoAVL));
    novo->valor = valor;
    novo->pai = pai;
```

```
    novo->esquerda = NULL;
    novo->direita = NULL;
    novo->altura = 1;

    if (pai == NULL) {
        arvore->raiz = novo;
    } else {
        if (valor > pai->valor) {
            pai->direita = novo;
        } else {
            pai->esquerda = novo;
        }
        avl_balanceamento(arvore, pai);
    }
}

void avl_balanceamento(ArvoreAVL* arvore, NoAVL* no) {
    while (no != NULL) {
        contAVL++;
                no->altura = 1 + avl_maximo(avl_altura(no->esquerda),
avl_altura(no->direita));
        int fator = avl_fb(no);

        if (fator > 1) {
            if (avl_fb(no->esquerda) >= 0) {
                avl_rsd(arvore, no);
            } else {
                avl_rdd(arvore, no);
            }
        } else if (fator < -1) {
```

```
            if (avl_fb(no->direita) <= 0) {
                avl_rse(arvore, no);
            } else {
                avl_rde(arvore, no);
            }
        }

        no = no->pai;
    }
}

NoAVL* avl_rse(ArvoreAVL* arvore, NoAVL* no) {
    contAVL++;
    NoAVL* pai = no->pai;
    NoAVL* direita = no->direita;

    if (direita->esquerda != NULL) direita->esquerda->pai = no;
    no->direita = direita->esquerda;
    no->pai = direita;
    direita->esquerda = no;
    direita->pai = pai;

    if (pai == NULL) {
        arvore->raiz = direita;
    } else {
        if (pai->esquerda == no) pai->esquerda = direita;
        else pai->direita = direita;
    }
```

```
            no->altura   =   1   +   avl_maximo(avl_altura(no->esquerda),
avl_altura(no->direita));
      direita->altura  =  1  +  avl_maximo(avl_altura(direita->esquerda),
avl_altura(direita->direita));
    return direita;
}

NoAVL* avl_rsd(ArvoreAVL* arvore, NoAVL* no) {
    contAVL++;
    NoAVL* pai = no->pai;
    NoAVL* esquerda = no->esquerda;

    if (esquerda->direita != NULL) esquerda->direita->pai = no;
    no->esquerda = esquerda->direita;
    no->pai = esquerda;
    esquerda->direita = no;
    esquerda->pai = pai;

    if (pai == NULL) {
        arvore->raiz = esquerda;
    } else {
        if (pai->esquerda == no) pai->esquerda = esquerda;
        else pai->direita = esquerda;
    }

            no->altura   =   1   +   avl_maximo(avl_altura(no->esquerda),
avl_altura(no->direita));
     esquerda->altura  =  1  +  avl_maximo(avl_altura(esquerda->esquerda),
avl_altura(esquerda->direita));
    return esquerda;
}
```

```
NoAVL* avl_rde(ArvoreAVL* arvore, NoAVL* no) {
    contAVL+=2;
    no->direita = avl_rsd(arvore, no->direita);
    return avl_rse(arvore, no);
}

NoAVL* avl_rdd(ArvoreAVL* arvore, NoAVL* no) {
    contAVL+=2;
    no->esquerda = avl_rse(arvore, no->esquerda);
    return avl_rsd(arvore, no);
}

NoAVL* avl_localizar(NoAVL* no, int valor) {
    while (no != NULL) {
        if (no->valor == valor) return no;
        contAVL++;
        no = valor < no->valor ? no->esquerda : no->direita;
    }
    return NULL;
}

NoAVL* avl_minimo(NoAVL* no) {
    while (no != NULL && no->esquerda != NULL) {
        contAVL++;
        no = no->esquerda;
    }
    return no;
}
```

```c
void avl_removerNo(ArvoreAVL* arvore, NoAVL* noRemover) {
    contAVL++;
    NoAVL* noPai = noRemover->pai;
     NoAVL* filho = (noRemover->esquerda != NULL) ? noRemover->esquerda
: noRemover->direita;

    if (noRemover->esquerda == NULL || noRemover->direita == NULL) {
        if (noPai == NULL) {
            arvore->raiz = filho;
        } else if (noPai->esquerda == noRemover) {
            noPai->esquerda = filho;
        } else {
            noPai->direita = filho;
        }

        if (filho != NULL) filho->pai = noPai;
        free(noRemover);

    } else {
        NoAVL* sucessor = avl_minimo(noRemover->direita);
        noRemover->valor = sucessor->valor;
        avl_removerNo(arvore, sucessor);
        return;
    }

    if (noPai != NULL) {
        avl_balanceamento(arvore, noPai);
    }
}
```

```c
void avl_remover(ArvoreAVL* arvore, int valor) {
    contAVL++;
    NoAVL* noRemover = avl_localizar(arvore->raiz, valor);
    if (noRemover != NULL) {
        avl_removerNo(arvore, noRemover);
    }
}

void avl_percorrer(NoAVL* no, void (*callback)(int)) {
    if (no != NULL) {
        avl_percorrer(no->esquerda, callback);
        callback(no->valor);
        avl_percorrer(no->direita, callback);
    }
}

void avl_destruir_no(NoAVL* no) {
    if (no == NULL) return;
    avl_destruir_no(no->esquerda);
    avl_destruir_no(no->direita);
    free(no);
}

void avl_destruir(ArvoreAVL* arvore) {
    if (arvore == NULL) return;
    avl_destruir_no(arvore->raiz);
    free(arvore);
}

/* --- MÓDULO: ÁRVORE B --- */
```

```c
int contB = 0;

typedef struct noB {
    int total;
    int* chaves;
    struct noB** filhos;
    struct noB* pai;
} NoB;

typedef struct arvoreB {
    NoB* raiz;
    int ordem;
} ArvoreB;

ArvoreB* btree_criar(int ordem);
void btree_destruir(ArvoreB* arvore);
void btree_adicionar(ArvoreB* arvore, int chave);
void btree_remover(ArvoreB* arvore, int chave);
void btree_percorrer(NoB* no);
NoB* btree_criaNo(ArvoreB* arvore);
int btree_pesquisaBinaria(NoB* no, int chave);
NoB* btree_localizaNo(ArvoreB* arvore, int chave);
void btree_adicionar_recursivo(ArvoreB* arvore, NoB* no, NoB* novo, int
chave);
void btree_adicionaChaveNo(NoB* no, NoB* novo, int chave);
NoB* btree_divideNo(ArvoreB* arvore, NoB* no);
int btree_transbordo(ArvoreB* arvore, NoB* no);
void btree_remover_interno(ArvoreB* arvore, NoB* no, int chave);
void remover_de_folha(NoB* no, int idx);
```

```c
void remover_de_nao_folha(ArvoreB* arvore, NoB* no, int idx);
void preencher_filho(ArvoreB* arvore, NoB* no, int idx_filho);
void emprestar_do_anterior(NoB* no, int idx);
void emprestar_do_proximo(NoB* no, int idx);
void fundir_com_proximo(ArvoreB* arvore, NoB* no, int idx);
NoB* btree_getPredecessor(ArvoreB* arvore, NoB* no, int pos);
void btree_destruir_no(NoB* no);

ArvoreB* btree_criar(int ordem) {
    ArvoreB* a = malloc(sizeof(ArvoreB));
    a->ordem = ordem;
    a->raiz = btree_criaNo(a);
    return a;
}

NoB* btree_criaNo(ArvoreB* arvore) {
    int max = arvore->ordem * 2;
    NoB* no = malloc(sizeof(NoB));
    no->pai = NULL;
    no->chaves = malloc(sizeof(int) * (max + 1));
    no->filhos = malloc(sizeof(NoB*) * (max + 2));
    no->total = 0;
    for (int i = 0; i < max + 2; i++) no->filhos[i] = NULL;
    return no;
}

void btree_adicionar(ArvoreB* arvore, int chave) {
    NoB* no = btree_localizaNo(arvore, chave);
    btree_adicionar_recursivo(arvore, no, NULL, chave);
}
```

```c
void btree_remover(ArvoreB* arvore, int chave) {
    contB++;
    if (!arvore->raiz) return;

    btree_remover_interno(arvore, arvore->raiz, chave);

    if (arvore->raiz->total == 0 && arvore->raiz->filhos[0] != NULL) {
        NoB* tmp = arvore->raiz;
        arvore->raiz = arvore->raiz->filhos[0];
        arvore->raiz->pai = NULL;
        free(tmp->chaves);
        free(tmp->filhos);
        free(tmp);
    }
}

int btree_pesquisaBinaria(NoB* no, int chave) {

    int inicio = 0, fim = no->total - 1, meio;
    while (inicio <= fim) {
        contB++;
        meio = inicio + (fim - inicio) / 2;
        if (no->chaves[meio] == chave) return meio;
        if (no->chaves[meio] > chave) fim = meio - 1;
        else inicio = meio + 1;
    }
    return inicio;
}
```

```c
NoB* btree_localizaNo(ArvoreB* arvore, int chave) {
    NoB* no = arvore->raiz;
    while (no != NULL) {
        int i = btree_pesquisaBinaria(no, chave);
        if (no->filhos[i] == NULL) return no;
        contB++;
        no = no->filhos[i];
    }
    return NULL;
}

void btree_adicionar_recursivo(ArvoreB* arvore, NoB* no, NoB* novo, int chave) {
    btree_adicionaChaveNo(no, novo, chave);

    if (btree_transbordo(arvore, no)) {
        int promovido = no->chaves[arvore->ordem];
        NoB* novoNo = btree_divideNo(arvore, no);

        if (no->pai == NULL) {
            NoB* pai = btree_criaNo(arvore);
            pai->filhos[0] = no;
            btree_adicionaChaveNo(pai, novoNo, promovido);
            no->pai = pai;
            novoNo->pai = pai;
            arvore->raiz = pai;
            contB++;
        } else {
            contB++;
```

```
                    btree_adicionar_recursivo(arvore, no->pai, novoNo,
promovido);
        }
    }
}

void btree_adicionaChaveNo(NoB* no, NoB* novo, int chave) {
    int i = btree_pesquisaBinaria(no, chave);

    for (int j = no->total - 1; j >= i; j--) {
        no->chaves[j + 1] = no->chaves[j];
        no->filhos[j + 2] = no->filhos[j + 1];
    }

    no->chaves[i] = chave;
    no->filhos[i + 1] = novo;
    if (novo != NULL) novo->pai = no;
    no->total++;
}

NoB* btree_divideNo(ArvoreB* arvore, NoB* no) {
    int meio = no->total / 2;
    NoB* novo = btree_criaNo(arvore);
    novo->pai = no->pai;

    contB++;

    for (int i = meio + 1; i < no->total; i++) {
        novo->filhos[novo->total] = no->filhos[i];
```

```
                            if    (novo->filhos[novo->total]    !=    NULL)
novo->filhos[novo->total]->pai = novo;
        novo->chaves[novo->total] = no->chaves[i];
        novo->total++;
    }
    novo->filhos[novo->total] = no->filhos[no->total];
                        if      (novo->filhos[novo->total]      !=      NULL)
novo->filhos[novo->total]->pai = novo;

    no->total = meio;
    return novo;
}

int btree_transbordo(ArvoreB* arvore, NoB* no) {
    return no->total > arvore->ordem * 2;
}

void btree_remover_interno(ArvoreB* arvore, NoB* no, int chave) {
    int idx = btree_pesquisaBinaria(no, chave);

    if (idx < no->total && no->chaves[idx] == chave) {
        if (no->filhos[0] == NULL)
            remover_de_folha(no, idx);
        else
            remover_de_nao_folha(arvore, no, idx);
    } else {
        if (no->filhos[0] == NULL) return;

        bool ultimo_filho = (idx == no->total);
        if (no->filhos[idx]->total < arvore->ordem) {
```

```
            preencher_filho(arvore, no, idx);
        }

        if (ultimo_filho && idx > no->total) {
            //contB++;
            btree_remover_interno(arvore, no->filhos[idx - 1], chave);
        } else {
            //contB++;
            btree_remover_interno(arvore, no->filhos[idx], chave);
        }
    }
}

void remover_de_folha(NoB* no, int idx) {
    for (int i = idx + 1; i < no->total; ++i) {
        no->chaves[i - 1] = no->chaves[i];
    }
    no->total--;
}

void remover_de_nao_folha(ArvoreB* arvore, NoB* no, int idx) {
    int chave = no->chaves[idx];

    if (no->filhos[idx]->total >= arvore->ordem) {
        NoB* pred_node = btree_getPredecessor(arvore, no, idx);
        int predecessor = pred_node->chaves[pred_node->total - 1];
        no->chaves[idx] = predecessor;
        btree_remover_interno(arvore, no->filhos[idx], predecessor);
    } else if (no->filhos[idx + 1]->total >= arvore->ordem) {
        NoB* suc_node = no->filhos[idx + 1];
```

```
                while (suc_node->filhos[0] != NULL) suc_node =
suc_node->filhos[0];
        int sucessor = suc_node->chaves[0];
        no->chaves[idx] = sucessor;
        btree_remover_interno(arvore, no->filhos[idx + 1], sucessor);
    } else {
        fundir_com_proximo(arvore, no, idx);
        btree_remover_interno(arvore, no->filhos[idx], chave);
    }
}

void preencher_filho(ArvoreB* arvore, NoB* no, int idx_filho) {
        if (idx_filho != 0 && no->filhos[idx_filho - 1]->total >=
arvore->ordem)
        emprestar_do_anterior(no, idx_filho);
    else if (idx_filho != no->total && no->filhos[idx_filho + 1]->total
>= arvore->ordem)
        emprestar_do_proximo(no, idx_filho);
    else {
        if (idx_filho != no->total)
            fundir_com_proximo(arvore, no, idx_filho);
        else
            fundir_com_proximo(arvore, no, idx_filho - 1);
    }
}

void emprestar_do_anterior(NoB* no, int idx) {
    contB++; //add
    NoB* filho = no->filhos[idx];
    NoB* irmao = no->filhos[idx - 1];
```

```
    for (int i = filho->total - 1; i >= 0; --i) filho->chaves[i + 1] =
filho->chaves[i];
    if (filho->filhos[0] != NULL) {
        for (int i = filho->total; i >= 0; --i) filho->filhos[i + 1] =
filho->filhos[i];
    }

    filho->chaves[0] = no->chaves[idx - 1];
    if (filho->filhos[0] != NULL) {
        filho->filhos[0] = irmao->filhos[irmao->total];
        if (filho->filhos[0]) filho->filhos[0]->pai = filho;
    }

    no->chaves[idx - 1] = irmao->chaves[irmao->total - 1];
    filho->total++;
    irmao->total--;
}

void emprestar_do_proximo(NoB* no, int idx) {
    contB++; //add
    NoB* filho = no->filhos[idx];
    NoB* irmao = no->filhos[idx + 1];

    filho->chaves[filho->total] = no->chaves[idx];
    if (filho->filhos[0] != NULL) {
        filho->filhos[filho->total + 1] = irmao->filhos[0];
        if (filho->filhos[filho->total + 1]) filho->filhos[filho->total
+ 1]->pai = filho;
    }

    no->chaves[idx] = irmao->chaves[0];
```

```
      for (int i = 1; i < irmao->total; ++i) irmao->chaves[i - 1] =
irmao->chaves[i];
    if (irmao->filhos[0] != NULL) {
         for (int i = 1; i <= irmao->total; ++i) irmao->filhos[i - 1] =
irmao->filhos[i];
    }

    filho->total++;
    irmao->total--;
}

void fundir_com_proximo(ArvoreB* arvore, NoB* no, int idx) {
    contB++; //add
    NoB* filho = no->filhos[idx];
    NoB* irmao = no->filhos[idx + 1];

    filho->chaves[arvore->ordem - 1] = no->chaves[idx];

    for (int i = 0; i < irmao->total; ++i) {
        filho->chaves[i + arvore->ordem] = irmao->chaves[i];
    }
    if (filho->filhos[0] != NULL) {
        for (int i = 0; i <= irmao->total; ++i) {
            filho->filhos[i + arvore->ordem] = irmao->filhos[i];
                if(filho->filhos[i + arvore->ordem]) filho->filhos[i +
arvore->ordem]->pai = filho;
        }
    }

      for (int i = idx + 1; i < no->total; ++i) no->chaves[i - 1] =
no->chaves[i];
```

```
      for (int i = idx + 2; i <= no->total; ++i) no->filhos[i - 1] =
no->filhos[i];


    filho->total += irmao->total + 1;
    no->total--;


    free(irmao->chaves);
    free(irmao->filhos);
    free(irmao);
}


NoB* btree_getPredecessor(ArvoreB* arvore, NoB* no, int pos) {
    contB++;
    NoB* atual = no->filhos[pos];
    while (atual->filhos[atual->total] != NULL) {
        contB++;
        atual = atual->filhos[atual->total];
    }
    return atual;
}


void btree_percorrer(NoB* no) {
    if (no != NULL) {
        for (int i = 0; i < no->total; i++) {
            btree_percorrer(no->filhos[i]);
            printf("%d ", no->chaves[i]);
        }
        btree_percorrer(no->filhos[no->total]);
    }
}
```

```c
void btree_destruir_no(NoB* no) {
    if (no == NULL) return;
    for (int i = 0; i <= no->total; i++) {
        btree_destruir_no(no->filhos[i]);
    }
    free(no->chaves);
    free(no->filhos);
    free(no);
}

void btree_destruir(ArvoreB* arvore) {
    if (arvore == NULL) return;
    btree_destruir_no(arvore->raiz);
    free(arvore);
}

/* --- MÓDULO: ÁRVORE RUBRO-NEGRA --- */

int contRubro = 0;

typedef enum { Vermelho, Preto } Cor;

typedef struct noRubro {
    struct noRubro* pai;
    struct noRubro* esquerda;
    struct noRubro* direita;
    Cor cor;
    int valor;
} NoRubro;
```

```c
typedef struct arvoreRubro {
    struct noRubro* raiz;
    struct noRubro* nulo;
} ArvoreRubro;

ArvoreRubro* rb_criar();
void rb_destruir(ArvoreRubro* arvore);
void rb_adicionar(ArvoreRubro* arvore, int valor);
void rb_remover(ArvoreRubro* arvore, int valor);
void rb_percorrer(ArvoreRubro* arvore, NoRubro* no, void (*callback)(int));
bool rb_vazia(ArvoreRubro* arvore);
NoRubro* rb_criarNo(ArvoreRubro* arvore, NoRubro* pai, int valor);
NoRubro* rb_adicionarNo(ArvoreRubro* arvore, NoRubro* no, int valor);
void rb_balancear_insercao(ArvoreRubro* arvore, NoRubro* no);
void rb_removerNo(ArvoreRubro* arvore, NoRubro* noRemover);
void rb_balancear_remocao(ArvoreRubro* arvore, NoRubro* no);
void rb_transplantar(ArvoreRubro* arvore, NoRubro* u, NoRubro* v);
NoRubro* rb_minimo(ArvoreRubro* arvore, NoRubro* no);
NoRubro* rb_localizar(ArvoreRubro* arvore, int valor);
void rb_rotacionarEsquerda(ArvoreRubro* arvore, NoRubro* no);
void rb_rotacionarDireita(ArvoreRubro* arvore, NoRubro* no);
void rb_destruir_no(ArvoreRubro* arvore, NoRubro* no);

ArvoreRubro* rb_criar() {
    ArvoreRubro *arvore = malloc(sizeof(ArvoreRubro));
    arvore->nulo = malloc(sizeof(NoRubro));
    arvore->nulo->cor = Preto;
    arvore->nulo->valor = 0;
```

```
    arvore->nulo->pai = NULL;
    arvore->nulo->esquerda = NULL;
    arvore->nulo->direita = NULL;
    arvore->raiz = arvore->nulo;
    return arvore;
}

bool rb_vazia(ArvoreRubro* arvore) {
    return arvore->raiz == arvore->nulo;
}

NoRubro* rb_criarNo(ArvoreRubro* arvore, NoRubro* pai, int valor) {
    NoRubro* no = malloc(sizeof(NoRubro));
    no->pai = pai;
    no->valor = valor;
    no->direita = arvore->nulo;
    no->esquerda = arvore->nulo;
    no->cor = Vermelho;
    return no;
}

void rb_adicionar(ArvoreRubro* arvore, int valor) {
    if (rb_vazia(arvore)) {
        arvore->raiz = rb_criarNo(arvore, arvore->nulo, valor);
        arvore->raiz->cor = Preto;
    } else {
        NoRubro* novoNo = rb_adicionarNo(arvore, arvore->raiz, valor);
        rb_balancear_insercao(arvore, novoNo);
    }
}
```

```c
NoRubro* rb_adicionarNo(ArvoreRubro* arvore, NoRubro* no, int valor) {
    contRubro++;
    if (valor >= no->valor) {
        if (no->direita == arvore->nulo) {
            no->direita = rb_criarNo(arvore, no, valor);
            return no->direita;
        }
        return rb_adicionarNo(arvore, no->direita, valor);
    } else {
        if (no->esquerda == arvore->nulo) {
            no->esquerda = rb_criarNo(arvore, no, valor);
            return no->esquerda;
        }
        return rb_adicionarNo(arvore, no->esquerda, valor);
    }
}

void rb_balancear_insercao(ArvoreRubro* arvore, NoRubro* no) {
    while (no != arvore->raiz && no->pai->cor == Vermelho) {
        NoRubro* pai = no->pai;
        NoRubro* avo = pai->pai;
        if (pai == avo->esquerda) {
            NoRubro *tio = avo->direita;
            if (tio->cor == Vermelho) {
                tio->cor = Preto;
                pai->cor = Preto;
                avo->cor = Vermelho;
                no = avo;
            } else {
```

```
                if (no == pai->direita) {
                    no = pai;
                    rb_rotacionarEsquerda(arvore, no);
                }
                no->pai->cor = Preto;
                no->pai->pai->cor = Vermelho;
                rb_rotacionarDireita(arvore, no->pai->pai);
            }
        } else {
            NoRubro *tio = avo->esquerda;
            if (tio->cor == Vermelho) {
                tio->cor = Preto;
                pai->cor = Preto;
                avo->cor = Vermelho;
                no = avo;
            } else {
                if (no == pai->esquerda) {
                    no = pai;
                    rb_rotacionarDireita(arvore, no);
                }
                no->pai->cor = Preto;
                no->pai->pai->cor = Vermelho;
                rb_rotacionarEsquerda(arvore, no->pai->pai);
            }
        }
        contRubro+=2;
    }
    arvore->raiz->cor = Preto;
}
```

```
void rb_remover(ArvoreRubro* arvore, int valor) {
    contRubro++;
    NoRubro* noRemover = rb_localizar(arvore, valor);
    if (noRemover != arvore->nulo) {
        rb_removerNo(arvore, noRemover);
    }
}

void rb_removerNo(ArvoreRubro* arvore, NoRubro* noRemover) {
    NoRubro *y = noRemover, *x;
    Cor corOriginalY = y->cor;

    if (noRemover->esquerda == arvore->nulo) {
        x = noRemover->direita;
        rb_transplantar(arvore, noRemover, noRemover->direita);
    } else if (noRemover->direita == arvore->nulo) {
        x = noRemover->esquerda;
        rb_transplantar(arvore, noRemover, noRemover->esquerda);
    } else {
        y = rb_minimo(arvore, noRemover->direita);
        corOriginalY = y->cor;
        x = y->direita;
        if (y->pai == noRemover) {
            x->pai = y;
        } else {
            rb_transplantar(arvore, y, y->direita);
            y->direita = noRemover->direita;
            y->direita->pai = y;
        }
        rb_transplantar(arvore, noRemover, y);
```

```
        y->esquerda = noRemover->esquerda;
        y->esquerda->pai = y;
        y->cor = noRemover->cor;
    }
    free(noRemover);
    if (corOriginalY == Preto) {
        rb_balancear_remocao(arvore, x);
    }
}

void rb_balancear_remocao(ArvoreRubro* arvore, NoRubro* no) {
    while (no != arvore->raiz && no->cor == Preto) {
        contRubro++;
        if (no == no->pai->esquerda) {
            NoRubro* irmao = no->pai->direita;
            if (irmao->cor == Vermelho) {
                irmao->cor = Preto;
                no->pai->cor = Vermelho;
                rb_rotacionarEsquerda(arvore, no->pai);
                irmao = no->pai->direita;
            }
             if (irmao->esquerda->cor == Preto && irmao->direita->cor ==
Preto) {
                irmao->cor = Vermelho;
                no = no->pai;
            } else {
                if (irmao->direita->cor == Preto) {
                    irmao->esquerda->cor = Preto;
                    irmao->cor = Vermelho;
                    rb_rotacionarDireita(arvore, irmao);
```

```
                    irmao = no->pai->direita;
                }
                irmao->cor = no->pai->cor;
                no->pai->cor = Preto;
                irmao->direita->cor = Preto;
                rb_rotacionarEsquerda(arvore, no->pai);
                no = arvore->raiz;
            }
        } else {
            NoRubro* irmao = no->pai->esquerda;
            if (irmao->cor == Vermelho) {
                irmao->cor = Preto;
                no->pai->cor = Vermelho;
                rb_rotacionarDireita(arvore, no->pai);
                irmao = no->pai->esquerda;
            }
             if (irmao->direita->cor == Preto && irmao->esquerda->cor ==
Preto) {
                irmao->cor = Vermelho;
                no = no->pai;
            } else {
                if (irmao->esquerda->cor == Preto) {
                    irmao->direita->cor = Preto;
                    irmao->cor = Vermelho;
                    rb_rotacionarEsquerda(arvore, irmao);
                    irmao = no->pai->esquerda;
                }
                irmao->cor = no->pai->cor;
                no->pai->cor = Preto;
                irmao->esquerda->cor = Preto;
```

```
                rb_rotacionarDireita(arvore, no->pai);
                no = arvore->raiz;
            }
        }
    }
    no->cor = Preto;
}

void rb_transplantar(ArvoreRubro* arvore, NoRubro* u, NoRubro* v) {
    if (u->pai == arvore->nulo) arvore->raiz = v;
    else if (u == u->pai->esquerda) u->pai->esquerda = v;
    else u->pai->direita = v;
    v->pai = u->pai;
}

NoRubro* rb_minimo(ArvoreRubro* arvore, NoRubro* no) {
    while (no->esquerda != arvore->nulo) {
        contRubro++;
        no = no->esquerda;
    }
    return no;
}

NoRubro* rb_localizar(ArvoreRubro* arvore, int valor) {
    NoRubro* no = arvore->raiz;
    while (no != arvore->nulo) {
        if (no->valor == valor) return no;
        contRubro++;
        no = valor < no->valor ? no->esquerda : no->direita;
    }
```

```
    return arvore->nulo;
}

void rb_rotacionarEsquerda(ArvoreRubro* arvore, NoRubro* no) {
    NoRubro* direita = no->direita;
    no->direita = direita->esquerda;
    if (direita->esquerda != arvore->nulo) direita->esquerda->pai = no;
    direita->pai = no->pai;

    if (no->pai == arvore->nulo) arvore->raiz = direita;
    else if (no == no->pai->esquerda) no->pai->esquerda = direita;
    else no->pai->direita = direita;

    direita->esquerda = no;
    no->pai = direita;
}

void rb_rotacionarDireita(ArvoreRubro* arvore, NoRubro* no) {
    NoRubro* esquerda = no->esquerda;
    no->esquerda = esquerda->direita;
    if (esquerda->direita != arvore->nulo) esquerda->direita->pai = no;
    esquerda->pai = no->pai;

    if (no->pai == arvore->nulo) arvore->raiz = esquerda;
    else if (no == no->pai->esquerda) no->pai->esquerda = esquerda;
    else no->pai->direita = esquerda;

    esquerda->direita = no;
    no->pai = esquerda;
}
```

```c
void rb_percorrer(ArvoreRubro* arvore, NoRubro* no, void (*callback)(int)) {
    if (no != arvore->nulo) {
        rb_percorrer(arvore, no->esquerda, callback);
        callback(no->valor);
        rb_percorrer(arvore, no->direita, callback);
    }
}

void rb_destruir_no(ArvoreRubro* arvore, NoRubro* no) {
    if (no == NULL || no == arvore->nulo) return;
    rb_destruir_no(arvore, no->esquerda);
    rb_destruir_no(arvore, no->direita);
    free(no);
}

void rb_destruir(ArvoreRubro* arvore) {
    if (arvore == NULL) return;
    rb_destruir_no(arvore, arvore->raiz);
    free(arvore->nulo);
    free(arvore);
}

/* --- FUNÇÃO PRINCIPAL --- */

int main() {
    const int MAX_N = 10000;
    const int PASSO = 1;
    const int NUM_AMOSTRAS = 10;
```

```
    const int MAX_VALOR_CHAVE = 100000;

    srand(time(NULL));

    FILE* f_adicao = fopen("resultados_adicao.csv", "w");
    FILE* f_remocao = fopen("resultados_remocao.csv", "w");

    if (f_adicao == NULL || f_remocao == NULL) {
        perror("Erro ao criar arquivos de resultado");
        return 1;
    }

                                                    fprintf(f_adicao,
"Tamanho,AVL,RubroNegra,B_Ordem_1,B_Ordem_5,B_Ordem_10\n");
                                                   fprintf(f_remocao,
"Tamanho,AVL,RubroNegra,B_Ordem_1,B_Ordem_5,B_Ordem_10\n");

    printf("Iniciando experimento de comparacao de arvores...\n");
    printf("Gerando resultados para N de %d a %d (passo de %d), com %d
amostras cada.\n", PASSO, MAX_N, PASSO, NUM_AMOSTRAS);

    for (int N = PASSO; N <= MAX_N; N += PASSO) {
        long long soma_adicao_avl = 0, soma_remocao_avl = 0;
        long long soma_adicao_rb = 0, soma_remocao_rb = 0;
        long long soma_adicao_b1 = 0, soma_remocao_b1 = 0;
        long long soma_adicao_b5 = 0, soma_remocao_b5 = 0;
        long long soma_adicao_b10 = 0, soma_remocao_b10 = 0;

        for (int amostra = 0; amostra < NUM_AMOSTRAS; amostra++) {

            int* chaves = malloc(sizeof(int) * N);
```

```
                    bool* numeros_usados = calloc(MAX_VALOR_CHAVE,
sizeof(bool));
            if (chaves == NULL || numeros_usados == NULL) {
                perror("Erro ao alocar memoria para as chaves");
                return 1;
            }

            for (int i = 0; i < N; i++) {
                int nova_chave;
                do {
                    nova_chave = rand() % MAX_VALOR_CHAVE;
                } while (numeros_usados[nova_chave]);
                chaves[i] = nova_chave;
                numeros_usados[nova_chave] = true;
            }
            free(numeros_usados);

            ArvoreAVL* avl = avl_criar();
            contAVL = 0;
            for(int i = 0; i < N; i++) avl_adicionar(avl, chaves[i]);
            soma_adicao_avl += contAVL;

            embaralhar(chaves, N);
            contAVL = 0;
            for(int i = 0; i < N; i++) avl_remover(avl, chaves[i]);
            soma_remocao_avl += contAVL;
            avl_destruir(avl);

            ArvoreRubro* rb = rb_criar();
            contRubro = 0;
```

```
        for(int i = 0; i < N; i++) rb_adicionar(rb, chaves[i]);
        soma_adicao_rb += contRubro;

        embaralhar(chaves, N);
        contRubro = 0;
        for(int i = 0; i < N; i++) rb_remover(rb, chaves[i]);
        soma_remocao_rb += contRubro;
        rb_destruir(rb);

        ArvoreB* b1 = btree_criar(1);
        contB = 0;
        for(int i = 0; i < N; i++) btree_adicionar(b1, chaves[i]);
        soma_adicao_b1 += contB;

        embaralhar(chaves, N);
        contB = 0;
        for(int i = 0; i < N; i++) btree_remover(b1, chaves[i]);
        soma_remocao_b1 += contB;
        btree_destruir(b1);

        ArvoreB* b5 = btree_criar(5);
        contB = 0;
        for(int i = 0; i < N; i++) btree_adicionar(b5, chaves[i]);
        soma_adicao_b5 += contB;

        embaralhar(chaves, N);
        contB = 0;
        for(int i = 0; i < N; i++) btree_remover(b5, chaves[i]);
        soma_remocao_b5 += contB;
        btree_destruir(b5);
```

```
            ArvoreB* b10 = btree_criar(10);
            contB = 0;
            for(int i = 0; i < N; i++) btree_adicionar(b10, chaves[i]);
            soma_adicao_b10 += contB;

            embaralhar(chaves, N);
            contB = 0;
            for(int i = 0; i < N; i++) btree_remover(b10, chaves[i]);
            soma_remocao_b10 += contB;
            btree_destruir(b10);

            free(chaves);
        }

        fprintf(f_adicao, "%d,%lld,%lld,%lld,%lld,%lld\n", N,
                        soma_adicao_avl / NUM_AMOSTRAS, soma_adicao_rb /
NUM_AMOSTRAS,
                         soma_adicao_b1 / NUM_AMOSTRAS, soma_adicao_b5 /
NUM_AMOSTRAS,
                soma_adicao_b10 / NUM_AMOSTRAS);
        fflush(f_adicao);

        fprintf(f_remocao, "%d,%lld,%lld,%lld,%lld,%lld\n", N,
                       soma_remocao_avl / NUM_AMOSTRAS, soma_remocao_rb /
NUM_AMOSTRAS,
                        soma_remocao_b1 / NUM_AMOSTRAS, soma_remocao_b5 /
NUM_AMOSTRAS,
                soma_remocao_b10 / NUM_AMOSTRAS);
        fflush(f_remocao);
```

```
        printf("Concluido para N = %d\n", N);
    }

    fclose(f_adicao);
    fclose(f_remocao);

    printf("\nExperimento finalizado com sucesso!\n");
           printf("Resultados salvos em 'resultados_adicao.csv' e
'resultados_remocao.csv'.\n");

    return 0;
}
```

ANEXO II

Código *Python* utilizado para a geração de gráficos.

```
import pandas as pd
import matplotlib.pyplot as plt


def   gerar_grafico(caminho_csv,   titulo_grafico,   nome_arquivo_saida,
escala_y_log=False):
    try:
        df = pd.read_csv(caminho_csv)


        plt.style.use('seaborn-v0_8-whitegrid')
        plt.figure(figsize=(12, 8))


        for coluna in df.columns[1:]:
            if df[coluna].sum() > 0:
                    plt.plot(df['Tamanho'], df[coluna], label=coluna,
marker='o', markersize=2, linestyle='-')


        plt.title(titulo_grafico, fontsize=16)
        plt.xlabel("Tamanho do Conjunto (N)", fontsize=12)
               plt.ylabel("Esforço  Computacional  (N°  de  Operações)",
fontsize=12)


        if escala_y_log:
            plt.yscale('log')
               plt.ylabel("Esforço Computacional (Escala Logarítmica)",
fontsize=12)


        plt.legend(title="Estrutura de Dados", fontsize=10)
        plt.ticklabel_format(style='plain', axis='x')
```

```
        if not escala_y_log:
            plt.ticklabel_format(style='plain', axis='y')

        plt.tight_layout()
        plt.savefig(nome_arquivo_saida, dpi=300)
        print(f"Gráfico '{nome_arquivo_saida}' gerado com sucesso!")

        plt.show()

    except FileNotFoundError:
        print(f"Erro: O arquivo '{caminho_csv}' não foi encontrado.")
    except Exception as e:
        print(f"Ocorreu um erro inesperado: {e}")

if __name__ == "__main__":
    gerar_grafico(
        caminho_csv='resultados_adicao.csv',
          titulo_grafico='Desempenho de Adição: Comparativo de Árvores
(Eixo Y Log)',
        nome_arquivo_saida='grafico_adicao_log.png',
        escala_y_log=True
    )

    print("-" * 30)

    gerar_grafico(
        caminho_csv='resultados_remocao.csv',
         titulo_grafico='Desempenho de Remoção: Comparativo de Árvores
(Eixo Y Log)',
```

```
        nome_arquivo_saida='grafico_remocao_log.png',
        escala_y_log=True
    )
```