Aula 13: Roteiro de Laboratório

Objetivos

- Aplicar conceitos teóricos em implementações práticas
- Desenvolver habilidades de programação em C
- Analisar complexidade de algoritmos implementados
- Resolver problemas complexos usando estruturas de dados

Laboratório 1: Implementação e Análise de Algoritmos de Ordenação

Parte A: Implementação Comparativa

Implemente os seguintes algoritmos de ordenação:

```c
// Arquivo: lab_ordenacao.c
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <string.h>

// Bubble Sort
void bubble_sort(int arr[], int n) {
    for (int i = 0; i < n-1; i++) {
        for (int j = 0; j < n-i-1; j++) {
            if (arr[j] > arr[j+1]) {
                int temp = arr[j];
                arr[j] = arr[j+1];
                arr[j+1] = temp;
            }
        }
    }
}

// Quick Sort
void quick_sort(int arr[], int low, int high) {
    if (low < high) {
        int pi = partition(arr, low, high);
        quick_sort(arr, low, pi - 1);
        quick_sort(arr, pi + 1, high);
    }
}

int partition(int arr[], int low, int high) {
    int pivot = arr[high];
    int i = (low - 1);

    for (int j = low; j <= high - 1; j++) {
        if (arr[j] < pivot) {
            i++;
            int temp = arr[i];
            arr[i] = arr[j];
            arr[j] = temp;
        }
    }
    int temp = arr[i + 1];
    arr[i + 1] = arr[high];
    arr[high] = temp;
    return (i + 1);
}

// Merge Sort
void merge_sort(int arr[], int l, int r) {
    if (l < r) {
        int m = l + (r - l) / 2;
        merge_sort(arr, l, m);
        merge_sort(arr, m + 1, r);
        merge(arr, l, m, r);
    }
}

void merge(int arr[], int l, int m, int r) {
    int n1 = m - l + 1;
    int n2 = r - m;

    int *L = malloc(n1 * sizeof(int));
    int *R = malloc(n2 * sizeof(int));

    for (int i = 0; i < n1; i++)
        L[i] = arr[l + i];
    for (int j = 0; j < n2; j++)
        R[j] = arr[m + 1 + j];

    int i = 0, j = 0, k = l;
```

Laboratório 2: Árvore Binária de Busca Completa

Implementação Completa

```c
// Arquivo: lab_bst.c
#include <stdio.h>
#include <stdlib.h>

typedef struct No {
    int data;
    struct No *esquerda, *direita;
    int altura;  // Para futuro uso em AVL
} No;

// Funções básicas implementadas anteriormente...

// Função adicional: Validar se é BST
int eh_bst(No *raiz, int min_val, int max_val) {
    if (raiz == NULL) return 1;

    if (raiz->data <= min_val || raiz->data >= max_val)
        return 0;

    return eh_bst(raiz->esquerda, min_val, raiz->data) &&
           eh_bst(raiz->direita, raiz->data, max_val);
}

// Função para encontrar k-ésimo menor elemento
void kth_smallest(No *raiz, int k, int *counter, int *result) {
    if (raiz == NULL || *counter >= k) return;

    kth_smallest(raiz->esquerda, k, counter, result);

    (*counter)++;
    if (*counter == k) {
        *result = raiz->data;
        return;
    }

    kth_smallest(raiz->direita, k, counter, result);
}

// Converter BST para array ordenado
void bst_to_array(No *raiz, int arr[], int *index) {
    if (raiz != NULL) {
        bst_to_array(raiz->esquerda, arr, index);
```

Laboratório 3: Sistema de Grafos com Algoritmos

Sistema Completo de Grafos

```c
// Arquivo: lab_grafos.c
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>

// Estruturas definidas anteriormente...

// Algoritmo de Dijkstra
void dijkstra(int grafo[][MAX_VERTICES], int origem, int vertices) {
    int dist[MAX_VERTICES];
    int visitado[MAX_VERTICES];

    for (int i = 0; i < vertices; i++) {
        dist[i] = INT_MAX;
        visitado[i] = 0;
    }

    dist[origem] = 0;

    for (int count = 0; count < vertices - 1; count++) {
        int u = encontrar_minimo_dist(dist, visitado, vertices);
        visitado[u] = 1;

        for (int v = 0; v < vertices; v++) {
            if (!visitado[v] && grafo[u][v] &&
                dist[u] != INT_MAX &&
                dist[u] + grafo[u][v] < dist[v]) {
                dist[v] = dist[u] + grafo[u][v];
            }
        }
    }

    imprimir_distancias(dist, vertices, origem);
}

// Detecção de ciclo usando DFS
int tem_ciclo_util(GrafoLista *g, int v, int *visitado, int *rec_stack) {
    visitado[v] = 1;
    rec_stack[v] = 1;

    NoLista *atual = g->listas[v];
    while (atual) {
        if (!visitado[atual->vertice]) {
            if (tem_ciclo_util(g, atual->vertice, visitado, rec_stack))
                return 1;
        } else if (rec_stack[atual->vertice]) {
            return 1;
        }
        atual = atual->proximo;
    }

    rec_stack[v] = 0;
    return 0;
}

int tem_ciclo(GrafoLista *g) {
    int *visitado = calloc(g->num_vertices, sizeof(int));
    int *rec_stack = calloc(g->num_vertices, sizeof(int));

    for (int i = 0; i < g->num_vertices; i++) {
        if (!visitado[i]) {
            if (tem_ciclo_util(g, i, visitado, rec_stack)) {
                free(visitado);
                free(rec_stack);
                return 1;
            }
        }
```

Laboratório 4: Projeto Integrador - Sistema de Gerenciamento

Especificação do Sistema

Desenvolva um sistema que integre múltiplas estruturas de dados:

Sistema de Biblioteca Digital:

- **Árvore BST:** Índice de livros por ID
- **Hash Table:** Busca rápida por título
- **Lista Ligada:** Fila de reservas
- **Grafo:** Recomendações baseadas em similaridade

Estrutura Base

```c
// Arquivo: sistema_biblioteca.c
typedef struct Livro {
    int id;
    char titulo[100];
```

Laboratório 5: Análise de Performance e Otimização

Framework de Benchmark

```c
// Arquivo: benchmark.c
#include <time.h>
#include <sys/time.h>

typedef struct {
    char nome[50];
    double tempo_ms;
    long memoria_bytes;
} ResultadoBenchmark;

double medir_tempo(void (*funcao)(int*, int), int *dados, int tamanho) {
    struct timeval inicio, fim;
    gettimeofday(&inicio, NULL);

    funcao(dados, tamanho);

    gettimeofday(&fim, NULL);

    double tempo = (fim.tv_sec - inicio.tv_sec) * 1000.0;
    tempo += (fim.tv_usec - inicio.tv_usec) / 1000.0;

    return tempo;
}

void benchmark_ordenacao() {
    int tamanhos[] = {1000, 5000, 10000, 50000, 100000};
    int num_tamanhos = sizeof(tamanhos) / sizeof(int);

    printf("Tamanho\tBubble\tQuick\tMerge\tHeap\n");

    for (int i = 0; i < num_tamanhos; i++) {
        int *dados1 = gerar_dados_aleatorios(tamanhos[i]);
        int *dados2 = copiar_array(dados1, tamanhos[i]);
        int *dados3 = copiar_array(dados1, tamanhos[i]);
        int *dados4 = copiar_array(dados1, tamanhos[i]);

        double t1 = medir_tempo(bubble_sort, dados1, tamanhos[i]);
        double t2 = medir_tempo_quick(dados2, 0, tamanhos[i]-1);
        double t3 = medir_tempo_merge(dados3, 0, tamanhos[i]-1);
        double t4 = medir_tempo(heap_sort, dados4, tamanhos[i]);

        printf("%d\t%.2f\t%.2f\t%.2f\t%.2f\n",
               tamanhos[i], t1, t2, t3, t4);
```

Laboratório 6: Problemas Avançados e Competição

Problemas Desafiadores

Problema 1: Merge de K Arrays Ordenados

```
typedef struct {
    int valor;
    int array_index;
    int element_index;
} HeapNode;

int* merge_k_arrays(int **arrays, int *tamanhos, int k, int *resultado_tamanho) {
    // Implementar usando heap mínimo
    // Complexidade: O(N log k) onde N é total de elementos
}
```

Problema 2: Árvore de Segmentos

```
typedef struct {
```

Entrega e Avaliação

Estrutura de Entrega:

```
laboratorio_[numero]/
├── src/
│   ├── main.c
│   ├── estruturas.c
│   ├── algoritmos.c
│   └── utils.c
├── docs/
│   ├── relatorio.md
│   └── analise_complexidade.md
├── testes/
│   ├── dados_teste/
│   └── resultados/
└── Makefile
```

Critérios de Avaliação:

1. Implementação (40%): Corretude e completude

Recursos de Apoio

Ferramentas Recomendadas:

- **Compilador:** GCC com flags de debug (-g -Wall -Wextra)
- **Debugger:** GDB para depuração
- **Profiler:** Valgrind para análise de memória
- **IDE:** Code::Blocks, Dev-C++, ou VSCode

Bibliografia de Apoio:

- Cormen et al. - Introduction to Algorithms
- Sedgewick - Algorithms in C
- Kernighan & Ritchie - The C Programming Language

Contato: