Relatório da Segunda Avaliação: Desenvolvimento de Aplicações com Lista Duplamente Encadeada, Árvores Vermelho-Preta e Árvores 2-3 na Disciplina de Estrutura de Dados II

Hermeson Alves de Oliveira
hermeson.de@ufpi.edu.br
Estruturas de Dados II - Juliana Oliveira de Carvalho

Abstract

Este trabalho, desenvolvido para a disciplina Estrutura de Dados II, implementa um sistema para gerenciar informações de estados brasileiros, suas cidades, CEPs e pessoas, utilizando uma lista duplamente encadeada para estados e estruturas de árvores específicas para diferentes finalidades. A primeira versão emprega árvores rubro-negras para organizar cidades, CEPs e pessoas, enquanto a segunda utiliza árvores 2-3 para as mesmas entidades. A terceira versão introduz árvores 4-5 para gerenciar blocos de memória em um sistema operacional, permitindo alocação e liberação de blocos livres e ocupados. O sistema suporta operações como cadastro, remoção e consultas específicas, incluindo identificar o estado mais populoso e contabilizar pessoas que não residem na cidade natal, além de funcionalidades de gerenciamento de memória. Cada implementação explora as características únicas de sua respectiva estrutura, contribuindo para o estudo prático de estruturas de dados complexas em aplicações que requerem organização hierárquica e gerenciamento de recursos.

Palavras-chave: Lista Duplamente Encadeada, Árvore Rubro-Negra, Árvore 2-3, Árvore 4-5, Gerenciamento de Dados, Gerenciamento de Memória

Introdução

Este trabalho, desenvolvido para a disciplina Estrutura de Dados II, implementa um sistema que utiliza uma lista duplamente encadeada para gerenciar os estados brasileiros, combinado com árvores balanceadas do tipo rubro-negra (no primeiro exercício) e 2-3 (no segundo exercício) para organizar cidades, CEPs e pessoas. Essas estruturas garantem ordenação e acesso eficiente aos dados, permitindo operações como cadastro, remoção e consultas específicas, como identificar o estado mais populoso. A escolha das árvores rubro-negras e 2-3 justifica-se por sua capacidade de manter o balanceamento, assegurando complexidade logarítmica em tarefas essenciais. Além disso, o trabalho incorpora uma árvore 4-5 (no terceiro exercício) para gerenciar blocos lógicos de 1 Mbyte a partir do bloco 0 em um sistema operacional, utilizando nós com status (livre ou ocupado), números de blocos inicial e final, e endereços inicial e final, otimizando a alocação e liberação de memória.

A implementação foi realizada em linguagem C, seguindo o paradigma estruturado, com o Visual Studio Code como ambiente de desenvolvimento, aproveitando suas ferramentas de depuração. O sistema simula um backend voltado à manipulação de dados de estados, municípios, códigos postais, indivíduos e blocos de memória, com foco em funcionalidades como registro, consulta e gestão de recursos, sem interface gráfica. O objetivo principal é explorar e analisar, de forma prática, as características das estruturas mencionadas em cenários reais de organização e alocação de dados.

Este relatório está organizado em quatro seções: a Seção 2 apresenta os fundamentos teóricos das estruturas utilizadas; a Seção 3 detalha os aspectos técnicos da implementação; a Seção 4 compila as conclusões e limitações observadas; e a Seção 5 oferece considerações adicionais. O projeto visa aprofundar o entendimento aplicado de estruturas de dados avançadas, com foco em situações que demandam hierarquização e gestão eficiente de recursos.

Especificações e Estruturação do Projeto

Nesta seção são detalhadas as especificações do sistema, incluindo sua arquitetura lógica, a organização dos arquivos e a aplicação das estruturas estudadas na disciplina. O projeto foi estruturado de forma modular, utilizando uma lista duplamente encadeada para representar os estados, árvores rubro-negras (primeiro exercício) e 2-3 (segundo exercício) para cidades, CEPs e pessoas, e uma árvore 4-5 (terceiro exercício) para gerenciar blocos de memória em um sistema operacional. Essa abordagem permite o reaproveitamento de código e a análise das características de cada estrutura em diferentes contextos. A organização dos arquivos, ilustrada na Figura 1, é compartilhada entre as implementações, com variações apenas nas funções específicas de balanceamento e gerenciamento de cada tipo de árvore.

Especificações dos Programas

O programa abrange três funcionalidades principais. Nos dois primeiros exercícios, organiza informações de estados brasileiros em uma lista duplamente encadeada, ordenada pelo nome do estado. Cada nó da lista contém dados do estado (nome, capital, quantidade de cidades, população) e uma árvore rubro-negra (ou 2-3, dependendo do exercício) de cidades. Cada nó da árvore de cidades inclui o nome da cidade, sua população e uma árvore de CEPs, que armazena apenas os códigos postais. Uma árvore separada de pessoas contém CPF, nome, CEPs de nascimento e moradia, e data de nascimento validada, permitindo operações como cadastro, remoção e consultas específicas, como identificar o estado mais populoso ou contabilizar pessoas que não residem na cidade natal. No terceiro exercício, o programa implementa um gerenciador de memória que utiliza uma árvore 4-5 para organizar blocos lógicos de 1 Mbyte, começando pelo bloco 0. Cada nó da árvore 4-5 armazena o status (livre ou ocupado), os números dos blocos inicial e final, e os endereços inicial e final, suportando operações de alocação e liberação de blocos de memória.

Especificações de Uso

O programa utiliza funções da biblioteca <string.h> para manipulação de strings e outras operações, como limpeza de buffer. É recomendado executá-lo em ambiente Windows para evitar problemas de compatibilidade, embora seja funcional em Linux e Mac com ajustes.

Nota: Para executar, compile o arquivo principal (compilador_main.c) com o comando gcc fornecido no código.

Hardware Utilizado

O sistema foi desenvolvido e testado em um notebook Dell Inspiron i15-i120K-A10P, conforme especificações na Tabela 1.

Especificação	Descrição
Processador	Intel Core i3-1215U (6 núcleos, 8 threads, 10 MB cache)
Geração	12ª geração
RAM	8 GB DDR4, 2666 MHz
Sistema Operacional	Windows 11 Home

Table 1: Especificações do Dell Inspiron i15-i120K-A10P.

Organização de Arquivos

O projeto está estruturado em diretórios principais denominados "Rubro_negro", "2-3" e "4-5", indicando claramente qual estrutura de dados está sendo empregada. As duas primeiras estruturas compartilham a mesma organização interna, diferenciando-se apenas pelo nome do diretório principal.

Internamente, há subdiretórios chamados "includes", "tads", "src" e "testes".

- O diretório "includes" armazena os arquivos de cabeçalho (.h), que incluem as definições das estruturas e os protótipos das funções.
- O diretório "tads" contém os arquivos de implementação (.c), abrangendo o programa principal.
- A subpasta "testes" reúne os testes unitários das funções desenvolvidas.

Dentro dos diretórios "includes" e "tads", os arquivos estão organizados em subpastas que categorizam suas respectivas funcionalidades, seguindo a ordem: Estruturas, Interatividade, Objetos e Utilitários.

- A subpasta "estruturas" inclui os arquivos responsáveis pela implementação das estruturas de dados.
- A subpasta "interatividade" abriga os arquivos que implementam as funções de interação com o usuário e as funções do sistema.
- A subpasta "objetos" contém os arquivos que implementam os objetos utilizados pelo sistema.
- A subpasta "utilitarios" armazena os arquivos com funções úteis e frequentemente repetidas, como verificação de alocação de memória e entrada de strings.

A organização detalhada dos arquivos pode ser visualizada na Figura 1.

Estruturação do Projeto

```
RUBRO_NEGRO
|-- includes
|   |-- Estruturas
|   |   |-- lista_dupla.h
|   |   |-- arvore_vermelho_preta.h
|   |   |-- dados.h
|   |-- Interatividade
|   |   |-- interatividade.h
|   |   |-- func_interatividade.h
|   |-- Objetos
|   |   |-- estado.h
|   |   |-- CEP.h
|   |   |-- pessoa.h
|   |   |-- cidade.h
|   |   |-- CPF.h
|   |   |-- data.h
|   |-- Utilitarios
|   |   |-- funcoes_sistema.h
|-- src
|   |-- output
|   |-- compilar_main.c
|   |-- main.c
|   Tads
|   |-- Estruturas
|   |   |-- lista_dupla.c
|   |   |-- arvore_vermelho_preta.c
|   |   |-- arvore_2_3.c
|   |-- Interatividade
|   |   |-- interatividade.c
|   |   |-- func_interatividade.c
|   |-- Objetos
|   |   |-- estado.c
|   |   |-- pessoa.c
|   |   |-- CEP.c
|   |   |-- Cidade.c
|   |   |-- CPF.c
|   |   |-- Data.c
|   |-- Utilitarios
|   |   |-- funcoes_sistema.c
|-- Testes
```

Figure 1: Estrutura das Pastas

OBS.: Foi-se usado a mesma arquitetura para a árvore 2-3 no quesito de separação por pastas.

Estruturas de Dados

As estruturas de dados foram projetadas para atender aos requisitos do sistema, eliminando o uso de ponteiros diretos para dados, conforme especificado:

- **Lista Duplamente Encadeada** [2]: Armazena estados, ordenados por nome.
- **Árvore Vermelho-Preta / 2-3** [5, 6]: Usadas para cidades, CEPs e pessoas, com balanceamento automático.
- **Estado** [7]: Contém nome, capital, quantidade de cidades, população e raiz da árvore de cidades.
- **Cidade** [8]: Contém nome, população e raiz da árvore de CEPs.
- **CEP** [5]: Armazena apenas o código CEP.
- **Pessoa** [10]: Contém CPF, nome, CEPs de nascimento e moradia, e data de nascimento.

Definição das Estruturas

```
typedef struct LISTA_DUPLAMENTE {
    ESTADO estado;
    struct LISTA_DUPLAMENTE *ant;
    struct LISTA_DUPLAMENTE *prox;
} LISTA_DUPLAMENTE;
```

Figure 2: Nó da lista duplamente encadeada.

```
typedef union DADOS {
    CIDADE cidade;
    PESSOA pessoa;
    char *CEP;
} DADOS;
```

Figure 3: União de dados.

```
typedef enum COR {
    PRETO,
    VERMELHO
} COR;
```

Figure 4: Enumeração de cores para a árvore rubro-negra.

```
typedef struct RUBRO_NEGRO {
    DADOS info;
    struct RUBRO_NEGRO *esquerda;
    struct RUBRO_NEGRO *direita;
    COR cor;
} RUBRO_NEGRO;
```

Figure 5: Nó da árvore rubro-negra.

```
typedef struct AVR_23 {
    DADOS info1, info2;
    short int nInfo;
    struct AVR_23 *esq, *cen, *dir;
} AVR_23;
```

Figure 6: Nó da árvore 2-3.

```
typedef struct ESTADO {
    char *nome_estado;
    char *nome_capital;
    short int quantidade_cidade;
    int quantidade_populacao;
    void *raiz_arvore_cidade;
} ESTADO;
```

Figure 7: Dados de estados.

```
typedef struct CIDADE {
    char *nome;
    int quantidade_populacao;
    void *raiz_arvore_CEPs;
} CIDADE;
```

Figure 8: Dados de cidades.

```
typedef struct DATA {
    short int dia;
    short int mes;
    short int ano;
} DATA;
```

Figure 9: Dados de data.

```
typedef struct PESSOA {
    char *CPF;
    char *nome;
    char *CEP_natal;
    char *CEP_atual;
    DATA data_nascimento;
} PESSOA;
```

Figure 10: Dados de pessoas.

Funções Específicas

1. **Lista Duplamente Encadeada**

   - **Função:** LISTA_DUPLAMENTE *inserir_ordernado_duplamente();
     Descrição: Insere um novo nó contendo um estado na lista duplamente encadeada, mantendo a ordem alfabética pelo nome do estado. A função ajusta os ponteiros ant e prox para preservar a integridade da lista.
     Parâmetros:
     - raiz: Ponteiro para o ponteiro da lista (permite modificar a raiz).
     - info: Estrutura ESTADO com os dados do estado.

     Retorno: Ponteiro para o nó inserido ou NULL se o estado já existe ou a inserção falhar.
     Exemplo: Para a lista [Bahia → São Paulo], inserir "Minas Gerais" posiciona o nó entre "Bahia" e "São Paulo", ajustando os ponteiros: [Bahia → Minas Gerais → São Paulo].
   - **Função:** LISTA_DUPLAMENTE *remover_duplamente();
     Descrição: Remove um nó da lista duplamente encadeada com base no nome do estado, ajustando os ponteiros ant e prox dos nós adjacentes.
     Parâmetros:
     - raiz: Ponteiro para o ponteiro da lista.
     - info: Estrutura ESTADO com o nome do estado a ser removido.

     Retorno: Ponteiro para o nó removido ou NULL se o estado não for encontrado.

     Exemplo: Para a lista [Bahia → Minas Gerais → São Paulo], remover "Minas Gerais" ajusta os ponteiros: Bahia−>prox = So Paulo e So Paulo−>ant = Bahia, resultando em [Bahia → São Paulo].

2. **Árvore Vermelho-Preta**

   - **Função:** RUBRO_NEGRO *inserir_rubro_negro();
     Descrição: Insere um novo nó com o dado info na árvore rubro-negra, mantendo as propriedades de balanceamento (raiz preta, nós vermelhos sem filhos vermelhos consecutivos, mesmo número de nós pretos em todos os caminhos). A inserção é recursiva, seguida de ajustes de rotação e troca de cores.
     Parâmetros:
     - raiz: Ponteiro para o ponteiro da raiz da árvore.
     - info: Dado a ser inserido (cidade, CEP ou pessoa).
     - comparar: Função de comparação para ordenação dos dados.

     Retorno: Ponteiro para o nó inserido ou NULL se a inserção falhar.
     Exemplo: Inserir uma cidade "Recife" em uma árvore com raiz "Salvador" (maior que "Recife") leva à inserção à esquerda. Se o nó pai for vermelho, rotações e trocas de cores são aplicadas para manter o balanceamento.

- **Função:** RUBRO_NEGRO *remover_rubro_negro();
  Descrição: Remove um nó da árvore rubro-negra com base no dado aux, mantendo as propriedades rubro-negras. A remoção envolve buscar o nó, substituí-lo pelo menor elemento da subárvore direita (se necessário) e rebalancear a árvore com rotações e ajustes de cores.
  Parâmetros:
  - raiz: Ponteiro para o ponteiro da raiz da árvore.
  - aux: Dado a ser removido.
  - comparar: Função de comparação para localizar o nó.

  Retorno: Ponteiro para o nó removido ou NULL se o nó não for encontrado.
  Exemplo: Remover "Salvador" de uma árvore com [Recife → Salvador → São Paulo] substitui "Salvador" pelo menor da subárvore direita ("São Paulo"), ajustando cores e ponteiros. Se necessário, rotações são aplicadas para manter o balanceamento.

3. **Árvore 2-3**

- **Função:** short int inserir_23();
  Descrição: Insere um novo valor na árvore 2-3, mantendo as propriedades de balanceamento. A inserção é recursiva, usando adiciona_infos para nós com 1 chave e quebra_no para nós com 2 chaves, promovendo uma chave ao pai. Se a raiz é dividida, uma nova raiz é criada.
  Parâmetros:
  - raiz: Ponteiro para o ponteiro da raiz da árvore.
  - valor: Dado a ser inserido (cidade, CEP ou pessoa).
  - comparar: Função de comparação para ordenação dos dados.

  Retorno: 1 se a inserção for bem-sucedida, 0 caso o valor já exista ou a alocação falhar.
  Exemplo: Inserir "Recife" em um nó [Salvador] forma [Recife, Salvador]. Inserir "Natal" em [Recife, Salvador] divide o nó, promovendo "Recife" como nova raiz e criando filhos [Natal] e [Salvador].
- **Função:** AVR_23 *quebra_no();
  Descrição: Divide um nó com 2 chaves (3-nó temporário) ao inserir um novo valor, promovendo a chave do meio ao nó pai e criando um novo nó com a maior chave. Ajusta os ponteiros dos filhos para manter a estrutura da árvore.
  Parâmetros:
  - no: Ponteiro para o nó a ser dividido.
  - info: Novo valor a ser inserido.
  - sobe: Ponteiro para armazenar a chave promovida.
  - F_dir: Filho direito associado ao novo valor.
  - comparar: Função de comparação para ordenação.

  Retorno: Ponteiro para o novo nó criado com a maior chave.
  Exemplo: Para um nó [Recife, Salvador] com inserção de "Natal", se "Natal" é menor que "Recife", promove "Recife", mantém "Natal" no nó original e cria um novo nó com "Salvador".
- **Função:** void adiciona_infos();
  Descrição: Adiciona um novo valor a um nó com 1 chave (2-nó), transformando-o em um 3-nó. Ajusta as chaves e os ponteiros dos filhos para manter a ordem correta.
  Parâmetros:
  - no: Ponteiro para o nó onde o valor será inserido.

- info: Novo valor a ser inserido.
- Sub_Arv_Info: Filho associado ao novo valor (se houver).
- comparar: Função de comparação para ordenação.

Retorno: Nenhum (modifica o nó diretamente).
Exemplo: Para um nó [Salvador], inserir "Recife" (menor que "Salvador") forma [Recife, Salvador], ajustando os ponteiros cen e dir.

Testes de Desempenho

Esta seção apresenta uma análise comparativa do desempenho das árvores balanceadas estudadas neste relatório: 2-3, 4-5 e Rubro-Negra. Os testes avaliam o tempo médio de inserção, busca, remoção e liberação, utilizando 100.000 números inteiros em três cenários distintos: crescente, decrescente e aleatório. Para garantir precisão, os tempos foram registrados em milissegundos e armazenados em arquivos, com as buscas sendo tão rápidas que, em muitos casos, não foram mensuráveis. Ressalta-se que os testes utilizam números inteiros para simplificar a implementação e padronizar as comparações.

Testes Crescentes

Os testes crescentes avaliam o desempenho das árvores ao inserir e remover números em ordem ascendente, de 1 a 100.000. Este cenário é particularmente desafiador, pois tende a gerar árvores desbalanceadas em estruturas menos robustas, destacando a eficiência dos mecanismos de balanceamento. Os resultados, apresentados na Tabela 2, mostram tempos médios de inserção e remoção, com buscas extremamente rápidas, indicando a eficácia das árvores balanceadas.

Métrica	**Árvore 2-3 (ms)**	**Árvore 4-5 (ms)**	**Árvore Rubro-Negra (ms)**
Tempo Médio de Inserção	0.33	0.33	0.56
Tempo Médio de Busca	0.00	0.00	0.00
Tempo Médio de Remoção	0.28	0.22	0.93
Tempo Médio de Liberação	0.00	0.00	0.00

Table 2: Tempos médios no teste Crescente.

Testes Decrescentes

Nos testes decrescentes, os números foram inseridos e removidos em ordem descendente, de 100.000 a 1. Este cenário testa a capacidade das árvores de manter o balanceamento em um padrão oposto ao crescente, podendo revelar diferenças na eficiência dos algoritmos de reestruturação. A Tabela 3 exibe os tempos médios, com destaque para a árvore Rubro-Negra, que apresentou tempos de remoção significativamente maiores, sugerindo maior sobrecarga em operações de rebalanceamento.

Métrica	**Árvore 2-3 (ms)**	**Árvore 4-5 (ms)**	**Árvore Rubro-Negra (ms)**
Tempo Médio de Inserção	0.34	0.46	0.72
Tempo Médio de Busca	0.00	0.00	0.00
Tempo Médio de Remoção	0.38	0.32	446.29
Tempo Médio de Liberação	0.00	0.00	0.00

Table 3: Tempos médios no teste Decrescente.

Testes Aleatórios

Os testes aleatórios envolvem a inserção e remoção de 100.000 números em ordem não determinística, simulando um caso de uso mais realista. Este cenário avalia a robustez das árvores em condições imprevisíveis, onde o balanceamento é constantemente desafiado. Conforme mostrado na Tabela 4, as árvores 2-3 e 4-5 mantêm desempenho estável, enquanto a Rubro-Negra exibe tempos de remoção elevados, possivelmente devido à complexidade de suas operações de rebalanceamento.

Métrica	**Árvore 2-3 (ms)**	**Árvore 4-5 (ms)**	**Árvore Rubro-Negra (ms)**
Tempo Médio de Inserção	0.46	0.43	0.77
Tempo Médio de Busca	0.00	0.00	0.00
Tempo Médio de Remoção	0.50	0.81	533.33
Tempo Médio de Liberação	0.00	0.00	0.00

Table 4: Tempos médios no teste Aleatório.

Conclusão

Este relatório apresentou a implementação de um sistema para gerenciar informações de estados brasileiros, cidades, CEPs e pessoas, utilizando uma lista duplamente encadeada para estados e árvores balanceadas (rubro-negras e 2-3) para organizar os dados de forma eficiente. A análise comparativa de desempenho revelou que a árvore 2-3 se destaca pela maior eficiência em operações de inserção e remoção, especialmente nos cenários de teste crescente, decrescente e aleatório, com tempos médios significativamente menores que os da árvore rubro-negra, que apresentou maior sobrecarga no rebalanceamento, particularmente em remoções. A estrutura 2-3 demonstrou robustez e estabilidade, garantindo complexidade logarítmica com menor custo computacional, enquanto a rubro-negra, embora eficaz, mostrou maior variabilidade nos tempos de remoção, como observado nas Tabelas 2, 3 e 4.

Devido a restrições de tempo, a implementação e os testes da árvore 4-5 não foram finalizados. A estruturação do projeto, com modularidade e organização clara dos arquivos, facilitou o desenvolvimento e a manutenção do código, mas a conclusão da árvore 4-5, destinada ao gerenciamento de blocos de memória, não foi concluída no prazo estipulado. Apesar disso, o trabalho reforça a importância das árvores balanceadas em aplicações que exigem organização hierárquica e acesso eficiente, contribuindo para o aprendizado prático de estruturas de dados avançadas na disciplina de Estrutura de Dados II.