UNIVERSIDADE ESTADUAL PAULISTA
“JÚLIO DE MESQUITA FILHO”

Implementação e avaliação de estruturas de dados aplicadas à análise meteorológica de Manaus

Autores: Murillo Brandão de Moura e Daniel Gonçalves da Silva

Sorocaba/SP

2025

Sumário

Introdução

A eficiência computacional é um fator crítico no desenvolvimento de sistemas embarcados, onde recursos como memória, processamento e tempo de resposta são limitados. Neste contexto, a escolha e implementação adequada de estruturas de dados se torna essencial para garantir desempenho, escalabilidade e organização eficiente das informações manipuladas.

Este trabalho propõe a construção de um sistema de análise e gerenciamento de dados meteorológicos da cidade de Manaus, utilizando linguagem C e aplicando diferentes estruturas de dados para armazenar, buscar, remover e processar mais de 10 mil registros climáticos diários.

O projeto vai além do simples armazenamento: ele implementa funcionalidades de busca filtrada, remoção otimizada, cálculo de estatísticas descritivas, simulações e benchmarks de desempenho. Além disso, compara a eficácia entre diferentes abordagens estruturais como ***árvore AVL***, ***lista encadeada***, ***tabela hash***, ***skip list*** e ***segment tree***, foram aplicadas em tarefas práticas, avaliando sua eficiência sob condições que simulam restrições típicas de sistemas embarcados, como limitações de tempo de processamento, uso de memória e latência.

Descrição e compreensão do dataset

O dataset utilizado nesta análise refere-se a dados meteorológicos da cidade de Manaus, obtidos por meio do repositório oficial do Instituto Nacional de Meteorologia (INMET). A base cobre um intervalo temporal contínuo, com registros diários compreendidos entre os anos de 1995 e 2025.

Ao todo, o conjunto de dados possui 10.969 amostras, cada uma representando as condições climáticas observadas em um dia específico. As variáveis contempladas incluem: data da medição, insolação total diária (horas), umidade relativa do ar (média diária, em %), temperatura máxima diária (°C), temperatura mínima diária (°C), velocidade média diária do vento (m/s).

Essas variáveis permitem não apenas avaliar padrões sazonais e comportamentos médios do clima local, mas também observar oscilações, extremos e variações ao longo do tempo. O volume de dados e sua granularidade tornam esta base especialmente adequada para análises estatísticas descritivas, visuais e inferenciais sobre a dinâmica climática de uma região tropical.

A escolha por analisar dados meteorológicos da cidade de Manaus se justifica por suas características climáticas singulares. Situada no coração da Floresta Amazônica, Manaus apresenta um clima equatorial, caracterizado por temperaturas elevadas durante todo o ano, alta umidade relativa do ar e um regime de chuvas marcado por grande volume e variação sazonal. Esses fatores tornam a cidade um ambiente ideal para investigar padrões climáticos consistentes, identificar eventos extremos e compreender a estrutura estatística de fenômenos atmosféricos em regiões tropicais úmidas. Além disso, estudar o comportamento climático de Manaus pode oferecer insights relevantes para áreas como planejamento urbano, saúde pública, agricultura e mudanças climáticas regionais.

Estrutura de dados

Para este projeto, foram implementadas e comparadas diferentes estruturas de dados com o objetivo de organizar, acessar e manipular um grande volume de registros meteorológicos. As estruturas adotadas incluem: Árvore AVL, Lista Encadeada, Tabela Hash, Skip List e Segment Tree. Cada uma delas foi analisada quanto à eficiência em operações de inserção, busca, remoção e cálculo estatístico, levando em conta também restrições típicas de sistemas embarcados, como tempo de execução e consumo de memória.

Árvore AVL

A Árvore AVL é uma árvore binária de busca balanceada, onde a diferença entre as alturas das subárvores esquerda e direita de qualquer nó nunca ultrapassa 1. Sempre que uma operação de inserção ou remoção desbalanceia a árvore, ela se autoajusta automaticamente através de rotações (simples ou duplas), garantindo complexidade de tempo logarítmica para operações de busca, inserção e remoção.

- Aplicabilidade no dataset

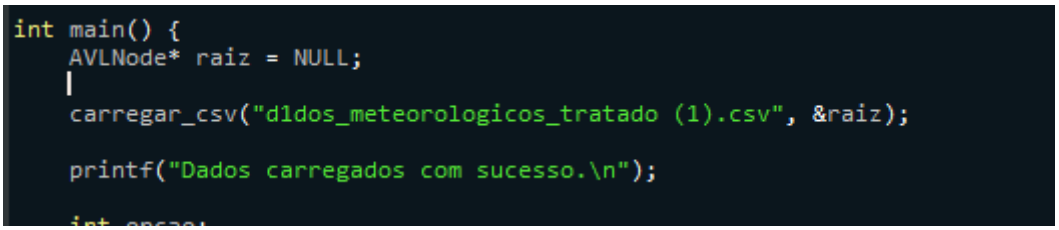
A Árvore AVL apresenta diversas vantagens que a tornam uma estrutura eficiente para o tratamento de dados meteorológicos ordenados por data, como os mais de 10.969 registros utilizados neste dataset. Dentre seus principais benefícios, destacam-se:

- Permite buscas rápidas e ordenadas por data, mantendo a eficiência mesmo com grandes volumes de dados;
- Garante balanceamento automático da árvore após inserções ou remoções, evitando degradação de desempenho;
- Ideal para acesso estruturado e operações com tempo próximo a O(log n), como filtros e buscas específicas;
- Permitir visualizações em ordem cronológica com facilidade (simples in-order traversal).

- Descrição e funcionalidade do código

A implementação da árvore AVL foi feita em C e permite:

- **Importação do CSV (carregar_csv)** - Lê os dados de um arquivo .csv e os insere automaticamente na árvore:



trecho*: carregar_csv("d1dos_meteorologicos_tratado (1).csv", &raiz);*

- **Inserção (inserir)**- A função inserir() é responsável por adicionar um novo registro do tipo Medicao à árvore AVL, organizando os dados com base na **data da medição**. Essa inserção é feita de forma recursiva e garante que, após cada operação, a árvore continue balanceada, mantendo sua eficiência de tempo logarítmico.

1. Criação do nó: Caso a posição atual da árvore seja nula (raiz == NULL), um novo nó é alocado com os dados da medição.
2. Inserção ordenada:
   - **2.1.** A data da nova medição é comparada com a data do nó atual.
   - **2.2.** Se for menor, o algoritmo desce para a subárvores esquerda.
   - **2.3.** Se for maior, desce para a subárvores direita.
   - **2.4.** Se for igual, não insere (evita duplicatas de data).
3. Atualização da altura: Após a inserção recursiva, a altura do nó atual é atualizada, com base na maior altura entre seus filhos esquerdo e direito.
4. Cálculo do fator de balanceamento:
   - **4.1.** A diferença entre as alturas das subárvores esquerda e direita é calculada.
   - **4.2.** Essa diferença é usada para verificar se o nó ficou desbalanceado após a inserção.
5. Rebalanceamento com rotações

   Se o fator de balanceamento for maior que 1 ou menor que -1, são aplicadas rotações para restaurar o equilíbrio:

   - **5.1.1.** Rotação à direita: caso Esquerda-Esquerda (LL)

5.1.2. Rotação à esquerda: caso Direita-Direita (RR)

5.1.3. Rotação dupla esquerda-direita: caso Esquerda-Direita (LR)

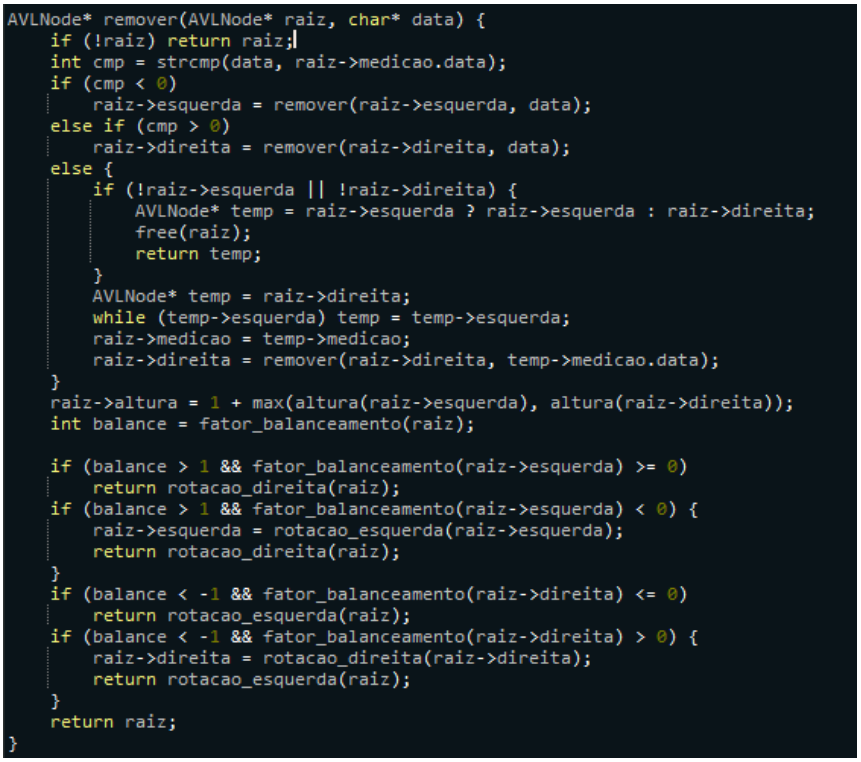
5.1.4. Rotação dupla direita-esquerda: caso Direita-Esquerda (RL)

```
AVLNode* inserir(AVLNode* raiz, Medicao medicao) {
    if (raiz == NULL) return criar_no(medicao);

    int cmp = strcmp(medicao.data, raiz->medicao.data);
    if (cmp < 0)
        raiz->esquerda = inserir(raiz->esquerda, medicao);
    else if (cmp > 0)
        raiz->direita = inserir(raiz->direita, medicao);
    else
        return raiz;

    raiz->altura = 1 + max(altura(raiz->esquerda), altura(raiz->direita));
    int balance = fator_balanceamento(raiz);

    if (balance > 1 && strcmp(medicao.data, raiz->esquerda->medicao.data) < 0)
        return rotacao_direita(raiz);
    if (balance < -1 && strcmp(medicao.data, raiz->direita->medicao.data) > 0)
        return rotacao_esquerda(raiz);
    if (balance > 1 && strcmp(medicao.data, raiz->esquerda->medicao.data) > 0) {
        raiz->esquerda = rotacao_esquerda(raiz->esquerda);
        return rotacao_direita(raiz);
    }
    if (balance < -1 && strcmp(medicao.data, raiz->direita->medicao.data) < 0) {
        raiz->direita = rotacao_direita(raiz->direita);
        return rotacao_esquerda(raiz);
    }

    return raiz;
}
```

- **Remoção (remover)** - A função remover() localiza e remove um nó com base em sua data. Após a remoção, ela recalcula as alturas e aplica rotações de balanceamento se necessário, mantendo a árvore AVL eficiente (balanceada). Etapas da função:

1. **Busca pela data desejada:**
   - **1.1.** A função compara a data a ser removida com a data do nó atual:
     - **1.1.1.** Se for menor: continua na subárvore esquerda.
     - **1.1.2.** Se for maior: continua na subárvore direita.
     - **1.1.3.** Se for igual: o nó a ser removido foi encontrado.
2. **Remoção do nó:**
   - **2.1.** Caso 1**:** o nó tem **um filho ou nenhum**: Ele é substituído diretamente pelo filho não-nulo (ou NULL).
   - **2.2.** Caso 2: o nó tem **dois filhos:**
     - **2.2.1.** A função encontra o **menor nó da subárvore direita** (o sucessor em ordem).
     - **2.2.2.** Substitui os dados do nó atual por ele.
     - **2.2.3.** Remove recursivamente o sucessor da subárvore direita.

3. **Atualização da altura e balanceamento:**
    - **3.1.** Atualiza a altura do nó atual.
    - **3.2.** Calcula o fator de balanceamento.
    - **3.3.** Aplica as rotações necessárias:
        - **3.3.1.** **LL**, **RR**, **LR** ou **RL**, conforme o tipo de desbalanceamento.



```
AVLNode* remover(AVLNode* raiz, char* data) {
    if (!raiz) return raiz;
    int cmp = strcmp(data, raiz->medicao.data);
    if (cmp < 0)
        raiz->esquerda = remover(raiz->esquerda, data);
    else if (cmp > 0)
        raiz->direita = remover(raiz->direita, data);
    else {
        if (!raiz->esquerda || !raiz->direita) {
            AVLNode* temp = raiz->esquerda ? raiz->esquerda : raiz->direita;
            free(raiz);
            return temp;
        }
        AVLNode* temp = raiz->direita;
        while (temp->esquerda) temp = temp->esquerda;
        raiz->medicao = temp->medicao;
        raiz->direita = remover(raiz->direita, temp->medicao.data);
    }
    raiz->altura = 1 + max(altura(raiz->esquerda), altura(raiz->direita));
    int balance = fator_balanceamento(raiz);

    if (balance > 1 && fator_balanceamento(raiz->esquerda) >= 0)
        return rotacao_direita(raiz);
    if (balance > 1 && fator_balanceamento(raiz->esquerda) < 0) {
        raiz->esquerda = rotacao_esquerda(raiz->esquerda);
        return rotacao_direita(raiz);
    }
    if (balance < -1 && fator_balanceamento(raiz->direita) <= 0)
        return rotacao_esquerda(raiz);
    if (balance < -1 && fator_balanceamento(raiz->direita) > 0) {
        raiz->direita = rotacao_direita(raiz->direita);
        return rotacao_esquerda(raiz);
    }
    return raiz;
}
```

- **Busca (buscar_e_salvar, busca_por_data)** - É possível buscar por qualquer campo (data, temperatura etc.) usando funções de filtro.

1. **Funções filtro (busca_por_data, busca_temp_maxima_igual, etc.)**
    1.1. Verificam se um campo específico da estrutura Medicao corresponde ao valor buscado.
2. **Função genérica buscar_e_salvar()**
    2.1. Percorre a árvore AVL recursivamente (em ordem).
    2.2. Aplica o filtro passado e salva os registros encontrados no vetor datas_encontradas[].

```c
int busca_por_data(Medicao m, void* valor) { return strstr(m.data, (char*)valor) != NULL; }
int busca_insolacao_igual(Medicao m, void* valor) { return m.insolacao == *(float*)valor; }
int busca_precipitacao_igual(Medicao m, void* valor) { return m.precipitacao == *(float*)valor; }
int busca_temp_maxima_igual(Medicao m, void* valor) { return m.temp_max == *(float*)valor; }
int busca_temp_minima_igual(Medicao m, void* valor) { return m.temp_min == *(float*)valor; }
int busca_umidade_igual(Medicao m, void* valor) { return m.umidade == *(float*)valor; }
int busca_vento_igual(Medicao m, void* valor) { return m.vento == *(float*)valor; }

void buscar_e_salvar(AVLNode* raiz, FiltroBuscaValor filtro, void* valor, int* contador) {
    if (!raiz || *contador >= MAX_RESULTADOS) return;

    buscar_e_salvar(raiz->esquerda, filtro, valor, contador);

    if (*contador < MAX_RESULTADOS && filtro(raiz->medicao, valor)) {
        strcpy(datas_encontradas[*contador], raiz->medicao.data);
        (*contador)++;
    }

    buscar_e_salvar(raiz->direita, filtro, valor, contador);
}
```

- **Visualização ordenada (imprimir_em_ordem)** - Percorre a árvore AVL em ordem crescente de datas e imprime cada registro meteorológico. Isso garante que o dataset seja exibido de forma cronológica.

```c
void imprimir_em_ordem(AVLNode* raiz) {
    if (!raiz) return;

    imprimir_em_ordem(raiz->esquerda);

    printf("Data: %s | Insolacao: %.1f | Precipitacao: %.1f | Temp Max: %.1f | Temp Min: %.1f | Umidade: %.1f | Vento: %.1f\n"
           raiz->medicao.data,
           raiz->medicao.insolacao,
           raiz->medicao.precipitacao,
           raiz->medicao.temp_max,
           raiz->medicao.temp_min,
           raiz->medicao.umidade,
           raiz->medicao.vento);

    imprimir_em_ordem(raiz->direita);
}
```

- **Benchmark** - Para medir o tempo de execução das operações principais (inserção, busca, remoção, estatísticas), foi utilizado o comando clock() da biblioteca <time.h>, que retorna o tempo de CPU desde o início da execução do programa.

```c
clock_t inicio = clock();  // Início da medição

// ... operação a ser medida (ex: busca, inserção, etc.)

clock_t fim = clock();     // Fim da medição

printf("Tempo de execução: %.6f segundos\n", (double)(fim - inicio) / CLOCKS_PER_SEC);
```

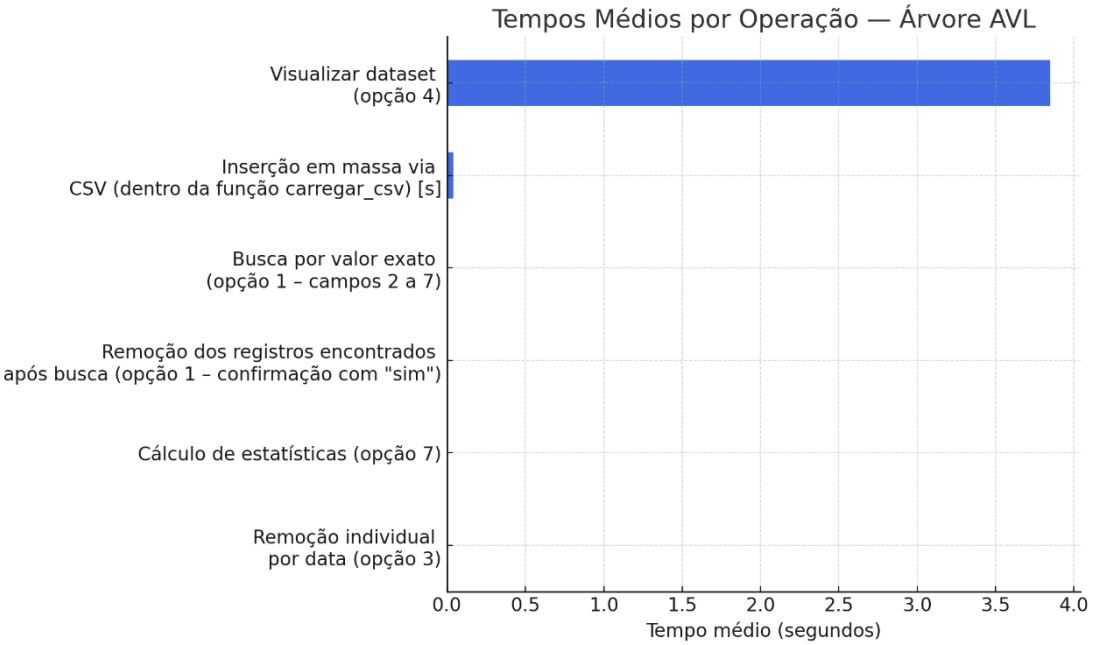
- **Funções do Menu -** O sistema apresenta um menu interativo com opções para inserir dados, buscar, remover, visualizar o dataset, calcular estatísticas, limpar os registros e exibir informações do dataset. Cada opção executa a função correspondente, facilitando o uso pelo usuário.

```c
void mostrar_menu() {
    printf("\n=========== MENU ===========\n");
    printf("Total de amostras coletadas: %d\n", total_amostras);
    printf("Amostras atuais: %d\n", amostras_atuais);
    printf("============================\n");
    printf("1 - Buscar por campo com valor exato\n");
    printf("2 - Inserir nova medicao manual\n");
    printf("3 - Remover medicao por data\n");
    printf("4 - Visualizar dataset\n");
    printf("5 - Sobre o dataset\n");
    printf("6 - Limpar dataset\n");
    printf("7 - Estatisticas do dataset\n");
    printf("0 - Sair\n");
    printf("============================\n");
    printf("Opcao: ");
}
```

- Desempenho e eficiência da estrutura AVL

A análise de desempenho da estrutura AVL foi conduzida com base em medições reais de tempo de execução para as principais operações do sistema. As médias

obtidas evidenciam o excelente comportamento da árvore, especialmente em cenários com alta demanda de leitura e manipulação de dados.



As operações mais custosas em termos de tempo foram a visualização completa do dataset e a inserção em massa via CSV. A visualização (opção 4 do menu) apresentou um tempo médio de 3,84 segundos, sendo que, na primeira execução, esse valor chegou a 11,17 segundos. Esse pico inicial é justificado pela primeira varredura completa da árvore e pelo volume de saída exibido no terminal. Nas execuções seguintes, o tempo estabilizou-se abaixo de 3 segundos, demonstrando um comportamento mais consistente e previsível.

Já a inserção em massa (função carregar_csv) registrou um tempo médio de 0,39 segundos, o que é altamente satisfatório, considerando-se o processamento de mais de 10 mil medições. Essa etapa envolve não apenas a leitura dos dados, mas também a criação e o balanceamento automático da árvore para cada nova entrada, o que reforça a robustez da estrutura mesmo sob grande volume.

Por outro lado, operações como busca por valor exato, remoção individual, remoção após busca e limpeza do dataset apresentaram tempos praticamente instantâneos, com médias na ordem de microssegundos ou inferiores. Esses resultados confirmam a eficiência logarítmica da AVL, mesmo sob carga elevada.

O cálculo estatístico também obteve bom desempenho, apesar de exigir a varredura completa da árvore. Seu tempo médio ficou abaixo da visualização, o que indica que o impacto da impressão contínua dos dados no terminal é mais significativo do que os cálculos em si.

Em resumo, os resultados obtidos demonstram que a Árvore AVL é altamente eficaz para aplicações que exigem buscas rápidas, inserções balanceadas e manipulação dinâmica de grandes conjuntos de dados. Sua performance estável e previsível justifica plenamente sua adoção neste projeto voltado a sistemas embarcados e análise meteorológica automatizada.

Lista encadeada

A lista encadeada é uma estrutura de dados linear e dinâmica composta por nós conectados entre si por ponteiros. Cada nó contém um elemento (neste caso, um registro meteorológico) e uma referência para o próximo nó da sequência. A principal característica dessa estrutura é a sua flexibilidade: ao contrário de vetores, ela não exige alocação contínua de memória e permite inserções e remoções com baixo custo, desde que a posição seja conhecida.

A implementação adotada neste trabalho utiliza inserção ordenada por data no momento da entrada dos dados, o que permite manter o dataset em ordem cronológica sem a necessidade de ordenações posteriores. Além disso, todas as operações são realizadas por meio de percursos sequenciais (lineares), o que simplifica a estrutura, embora implique em maior custo computacional para buscas e remoções em grandes volumes.

- Aplicabilidade no dataset

A lista encadeada é uma estrutura adequada para representar conjuntos de dados sequenciais e dinâmicos, como os registros meteorológicos utilizados na análise. Sua simplicidade estrutural e comportamento determinístico tornam-na útil para cenários em que a ordenação cronológica é relevante e a complexidade de implementação deve ser reduzida.

Entre os aspectos que justificam sua aplicação neste contexto, destacam-se:

- Estrutura leve e de fácil implementação, sem necessidade de alocação contínua de memória ou mecanismos de balanceamento;
- Inserção ordenada por data, permitindo manter os dados organizados cronologicamente à medida que são inseridos;
- Acesso sequencial direto, compatível com a leitura linear de grandes volumes de dados meteorológicos;
- Referência útil de comparação com estruturas mais complexas, permitindo observar os impactos da ausência de balanceamento ou indexação;
- Aplicabilidade didática, por representar uma base para compreender os trade-offs entre desempenho e simplicidade.

A análise da lista encadeada tem como objetivo avaliar sua eficiência em operações como busca, inserção ordenada, remoção e cálculo estatístico, especialmente quando comparada a outras estruturas implementadas.

- Descrição e funcionalidade do código

A implementação da lista encadeada foi feita em C e permite:

- **Importar CSV (carregar_csv)** - Realiza a leitura do arquivo contendo os dados meteorológicos e insere cada registro na lista em ordem crescente de data.

```c
void carregar_csv(const char* caminho, Nodo** lista) {
    FILE* arquivo = fopen(caminho, "r");
    if (!arquivo) {
        perror("Erro ao abrir o arquivo");
        return;
    }
    char linha[LINHA_MAX];
    fgets(linha, LINHA_MAX, arquivo);

    clock_t inicio = clock(); // INÍCIO da medição

    while (fgets(linha, LINHA_MAX, arquivo)) {
        Medicao m;
        char* token = strtok(linha, ",");
        if (token) strncpy(m.data, token, 20);
        token = strtok(NULL, ","); m.insolacao = atof(token);
        token = strtok(NULL, ","); m.precipitacao = atof(token);
        token = strtok(NULL, ","); m.temp_max = atof(token);
        token = strtok(NULL, ","); m.temp_min = atof(token);
        token = strtok(NULL, ","); m.umidade = atof(token);
        token = strtok(NULL, ","); m.vento = atof(token);
        *lista = inserir_ordenado(*lista, m);
        total_amostras++;
        amostras_atuais++;
    }
    clock_t fim = clock(); // FIM da medição
    printf("Tempo de insercao em massa (CSV): %.6f segundos\n", (double)(fim - inicio) / CLOCKS_PER_SEC);

    fclose(arquivo);
}
```

- **Inserção (inserir_ordenado)** - Adiciona novos registros à lista mantendo a ordenação cronológica. Esse processo é utilizado tanto para inserções manuais quanto para a carga inicial via CSV.

```c
Nodo* inserir_ordenado(Nodo* lista, Medicao m) {
    Nodo* novo = (Nodo*)malloc(sizeof(Nodo));
    novo->medicao = m;
    novo->prox = NULL;

    if (!lista || strcmp(m.data, lista->medicao.data) < 0) {
        novo->prox = lista;
        return novo;
    }

    Nodo* atual = lista;
    while (atual->prox && strcmp(m.data, atual->prox->medicao.data) > 0)
        atual = atual->prox;

    novo->prox = atual->prox;
    atual->prox = novo;
    return lista;
}
```

1. **Cria um novo nó** (Nodo) e copia os dados da medição para ele.
2. **Caso a lista esteja vazia** ou o novo elemento deva ser inserido no início (por ser menor do que o primeiro), ele se torna o novo primeiro elemento.

3. Caso contrário, percorre a lista até encontrar a **posição correta**, de forma que o novo nó seja inserido **logo antes do primeiro maior**.
4. Faz o ajuste dos ponteiros, conectando o novo nó na posição correta.

Como os dados meteorológicos precisam ficar cronologicamente ordenados por data, essa função garante que a lista encadeada já esteja em ordem desde a inserção, evitando a necessidade de ordenações posteriores — o que é uma vantagem operacional para listas simples.

● **Remoção (remover_por_data e remover_resultados)** - A remoção de dados na lista encadeada é feita por duas funções complementares. A função *remover_por_data* percorre a lista à procura de um registro com a data exata informada, ajustando os ponteiros conforme necessário e liberando a memória do nó removido. Já a função *remover_resultados* permite a remoção em massa, iterando sobre o vetor datas_encontradas[], que armazena os registros localizados por uma busca anterior. Essa abordagem permite excluir múltiplos elementos de forma eficiente e controlada, mantendo a lista atualizada conforme os filtros aplicados.

```
Nodo* remover_por_data(Nodo* lista, const char* data) {
    Nodo *atual = lista, *anterior = NULL;
    while (atual) {
        if (strcmp(atual->medicao.data, data) == 0) {
            if (anterior)
                anterior->prox = atual->prox;
            else
                lista = atual->prox;
            free(atual);
            amostras_atuais--;
            return lista;
        }
        anterior = atual;
        atual = atual->prox;
    }
    return lista;
}
```

```
Nodo* remover_resultados(Nodo* lista) {
    for (int i = 0; i < total_resultados; i++) {
        lista = remover_por_data(lista, datas_encontradas[i]);
    }
    total_resultados = 0;
    return lista;
}
```

- **Busca (buscar_por_valor)** - A busca por registros na lista encadeada é realizada pela função buscar_por_valor, que percorre toda a estrutura verificando se o valor de um campo específico (como data, insolação ou temperatura) corresponde exatamente ao valor fornecido pelo usuário. O campo de busca é identificado por um código numérico, e a comparação é feita com base em igualdade exata (ou substring, no caso da data). Os registros que atendem ao critério de busca têm suas datas armazenadas no vetor datas_encontradas[], possibilitando ações subsequentes como remoção em massa. Essa abordagem é simples, porém eficiente para buscas pontuais em listas ordenadas cronologicamente.

```c
void buscar_por_valor(Nodo* lista, int campo, void* valor, int* contador) {
    *contador = 0;
    while (lista && *contador < MAX_RESULTADOS) {
        Medicao m = lista->medicao;
        int match = 0;
        switch (campo) {
            case 1: match = strstr(m.data, (char*)valor) != NULL; break;
            case 2: match = m.insolacao == *(float*)valor; break;
            case 3: match = m.precipitacao == *(float*)valor; break;
            case 4: match = m.temp_max == *(float*)valor; break;
            case 5: match = m.temp_min == *(float*)valor; break;
            case 6: match = m.umidade == *(float*)valor; break;
            case 7: match = m.vento == *(float*)valor; break;
        }
        if (match) {
            strcpy(datas_encontradas[*contador], m.data);
            (*contador)++;
        }
        lista = lista->prox;
    }
}
```

- **Visualização ordenada (imprimir_lista)** - A função imprimir_lista percorre todos os nós da lista encadeada e imprime as informações de cada medição meteorológica de forma estruturada. Como a lista é construída com inserção ordenada por data, a exibição ocorre naturalmente em ordem cronológica. A saída inclui todas as variáveis relevantes (data, insolação, precipitação, temperaturas máxima e mínima, umidade e vento), possibilitando ao usuário uma visualização completa e organizada dos dados diretamente no terminal.

```c
void imprimir_lista(Nodo* lista) {
    while (lista) {
        Medicao m = lista->medicao;
        printf("Data: %s | Insolacao: %.1f | Precipitacao: %.1f | Temp Max: %.1f | Temp Min: %.1f | Umidade: %.1f | Vento: %.1f\n",
               m.data, m.insolacao, m.precipitacao, m.temp_max, m.temp_min, m.umidade, m.vento);
        lista = lista->prox;
    }
}
```

- **Benchmark** - A maior parte das operações críticas do sistema conta com medições de tempo usando a biblioteca time.h, possibilitando a avaliação de desempenho em microssegundos.

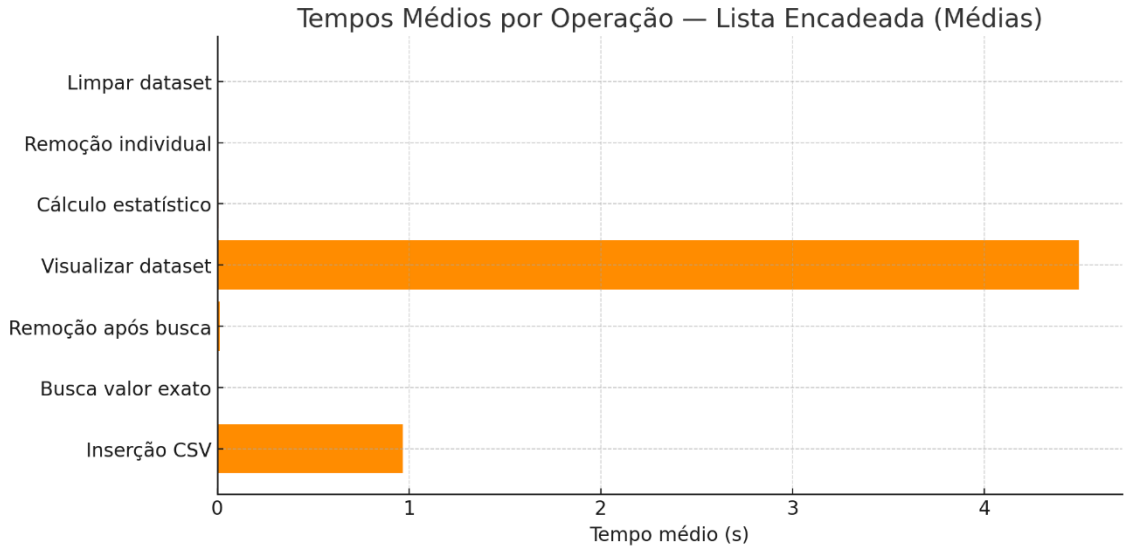
```c
clock_t inicio = clock();  // Início da medição

// ... operação a ser medida (ex: busca, inserção, etc.)

clock_t fim = clock();     // Fim da medição

printf("Tempo de execução: %.6f segundos\n", (double)(fim - inicio) / CLOCKS_PER_SEC);
```

- **Funções do Menu (mostrar_menu)** - O sistema apresenta um menu interativo com opções de uso, incluindo inserção manual, busca, remoção, visualização, estatísticas, limpeza e encerramento. Cada opção executa a função correspondente, facilitando a operação pelo usuário.

```c
void mostrar_menu() {
    printf("\n========== MENU ==========\n");
    printf("Total de amostras coletadas: %d\n", total_amostras);
    printf("Amostras atuais: %d\n", amostras_atuais);
    printf("==========================\n");
    printf("1 - Buscar por campo com valor exato\n");
    printf("2 - Inserir nova medicao manual\n");
    printf("3 - Remover medicao por data\n");
    printf("4 - Visualizar dataset\n");
    printf("5 - Sobre o dataset\n");
    printf("6 - Limpar dataset\n");
    printf("7 - Estatisticas do dataset\n");
    printf("0 - Sair\n");
    printf("==========================\n");
    printf("Opcao: ");
}
```

- Desempenho e eficiência da lista encadeada

A análise de desempenho da estrutura Lista Encadeada foi realizada por meio de medições reais de tempo aplicadas às principais operações do sistema. O objetivo foi avaliar sua eficiência em tarefas como inserção, remoção, busca e visualização de um dataset com mais de 10 mil registros. Apesar de sua simplicidade estrutural, os resultados mostram que a lista encadeada mantém um comportamento funcional e estável, especialmente em operações sequenciais e lineares.



As operações mais custosas em termos de tempo na lista encadeada foram a visualização completa do dataset e a inserção em massa via CSV. A visualização (opção 4 do menu) teve tempo médio de 4,49 segundos, valor um pouco superior ao da AVL. Isso ocorre porque, mesmo sem balanceamento, a lista percorre todos os nós linearmente, e o custo da impressão no terminal continua sendo o fator mais pesado da operação.

Já a inserção em massa (função carregar_csv) apresentou tempo médio de 0,96 segundos, também superior ao tempo da AVL. Como não há balanceamento, o custo está relacionado ao tempo gasto para localizar a posição correta e inserir cada elemento de forma ordenada. Isso reforça a natureza linear da estrutura, que exige comparação em cada passo até encontrar a posição de inserção.

Por outro lado, operações como busca por valor exato, remoção individual, remoção após busca e limpeza do dataset mantiveram tempos muito baixos, geralmente na casa dos microssegundos. Mesmo com percursos lineares, o custo computacional dessas tarefas foi pequeno, o que mostra que, para operações pontuais, a lista encadeada é bem responsiva.

O cálculo estatístico também teve bom desempenho, com tempo médio de 0,0067 segundos. Como essa função percorre todos os elementos apenas uma

vez, a lista encadeada consegue realizar a tarefa de forma eficiente, sem perdas significativas de desempenho.

Em resumo, a lista encadeada se mostrou eficiente em operações simples e lineares, mas apresentou desempenho inferior à árvore AVL em tarefas que envolvem grandes volumes de dados ou múltiplas comparações, como a inserção e a visualização completa.

Tabela Hash

Aplicabilidade no dataset

A Tabela Hash é uma estrutura altamente eficiente para acesso direto a dados a partir de uma chave, ideal para situações que requerem consultas rápidas. Neste projeto, ela foi utilizada para armazenar registros meteorológicos indexados pela data, funcionando como chave primária para inserção, busca e remoção. Sua aplicação no dataset de Manaus se justifica pelas seguintes vantagens:

- **Eficiência de acesso**: as operações de busca, inserção e remoção possuem complexidade média O(1), o que significa que seu tempo de execução independe diretamente do tamanho da base, tornando-a ideal para grandes volumes de dados;

- **Espalhamento uniforme**: é empregada uma função hash simples que soma os caracteres da data e aplica o operador módulo para determinar a posição na tabela. Essa abordagem garante uma distribuição eficiente das chaves, reduzindo colisões em média;

- **Baixo custo de atualização**: ao contrário de estruturas como a árvore AVL, a Tabela Hash não requer rebalanceamento nem ajustes de altura;

- **Capacidade de lidar com grandes volumes de dados**: mesmo com colisões, a estrutura se mostra robusta e responsiva, utilizando encadeamento para manter os dados acessíveis com eficiência.

Apesar de não manter os dados ordenados, a Tabela Hash foi essencial para demonstrar desempenho superior em operações pontuais, servindo como referência de comparação frente a estruturas mais lineares ou balanceadas.

Descrição e funcionalidade do código

A implementação da Tabela Hash foi feita em C++, utilizando listas encadeadas para lidar com colisões. A estrutura foi pensada para cobrir diversas operações fundamentais no gerenciamento de dados meteorológicos, cada uma com sua finalidade:

- **Leitura do CSV (lerCSV)**: Abre e percorre o arquivo contendo os registros meteorológicos, lendo linha por linha. Para cada linha, extrai os campos (data, insolacão, precipitação, temperaturas, umidade e vento), cria uma estrutura Medição e a insere na tabela hash. Durante esse processo, também é atualizado o intervalo de datas (mínima e máxima), servindo como referência para entradas futuras.

```
void lerCSV(string nomeArquivo) {
    ifstream arquivo(nomeArquivo);
    if (!arquivo.is_open()) {
        cout << "Erro ao abrir o arquivo.\n";
        return;
    }
    string linha;
    getline(arquivo, linha);
    while (getline(arquivo, linha)) {
        stringstream ss(linha);
        string item;
        Medicao m;
        getline(ss, m.data, ',');
        getline(ss, item, ','); m.insolacao = atof(item.c_str());
        getline(ss, item, ','); m.precipitacao = atof(item.c_str());
        getline(ss, item, ','); m.temp_max = atof(item.c_str());
        getline(ss, item, ','); m.temp_min = atof(item.c_str());
        getline(ss, item, ','); m.umidade = atof(item.c_str());
        getline(ss, item);      m.vento = atof(item.c_str());
        inserir(m);
    }
    arquivo.close();
}
```

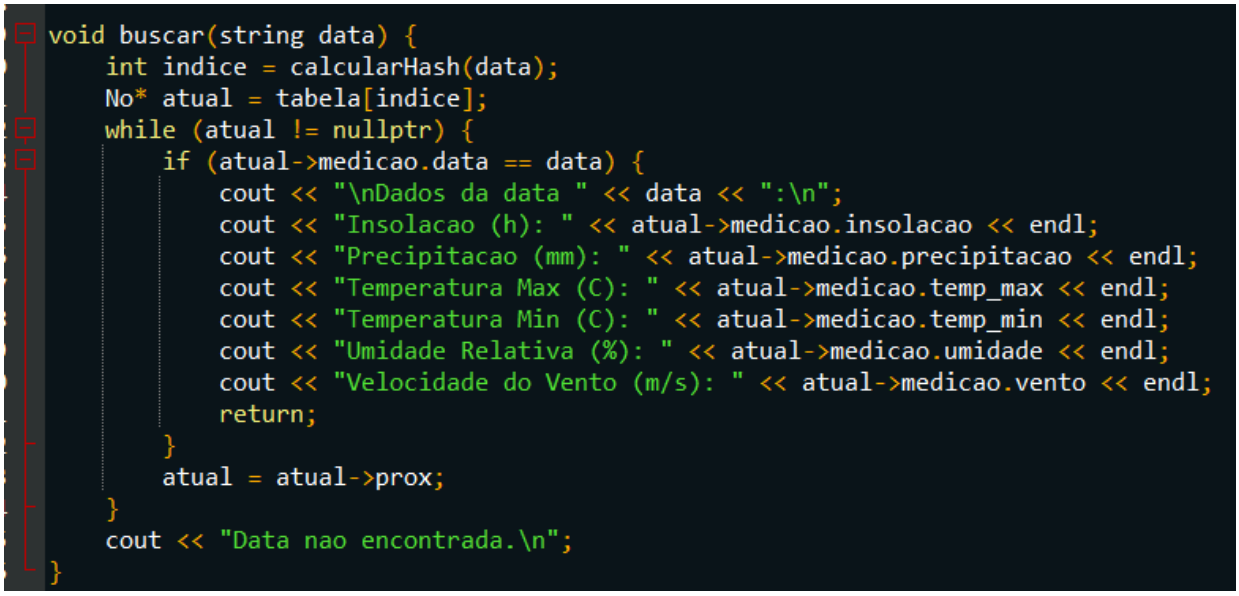
- **Inserção (inserir)**: Calcula o índice da tabela a partir da data, utilizando a função hash. Em seguida, cria um novo nó contendo a medição e insere-o na frente da lista encadeada da posição correspondente. Essa inserção é simples, eficiente e responsiva mesmo sob colisões.

```
void inserir(Medicao m) {
    int indice = calcularHash(m.data);
    No* novo = new No{m, tabela[indice]};
    tabela[indice] = novo;
    if (m.data < data_minima) data_minima = m.data;
    if (m.data > data_maxima) data_maxima = m.data;
}
```

- **Remoção (remover)**: Percorre a lista na posição hash adequada, procurando um nó com a data especificada. Quando encontrado, o nó é removido com cuidado para manter a integridade da lista, liberando a memória ocupada.

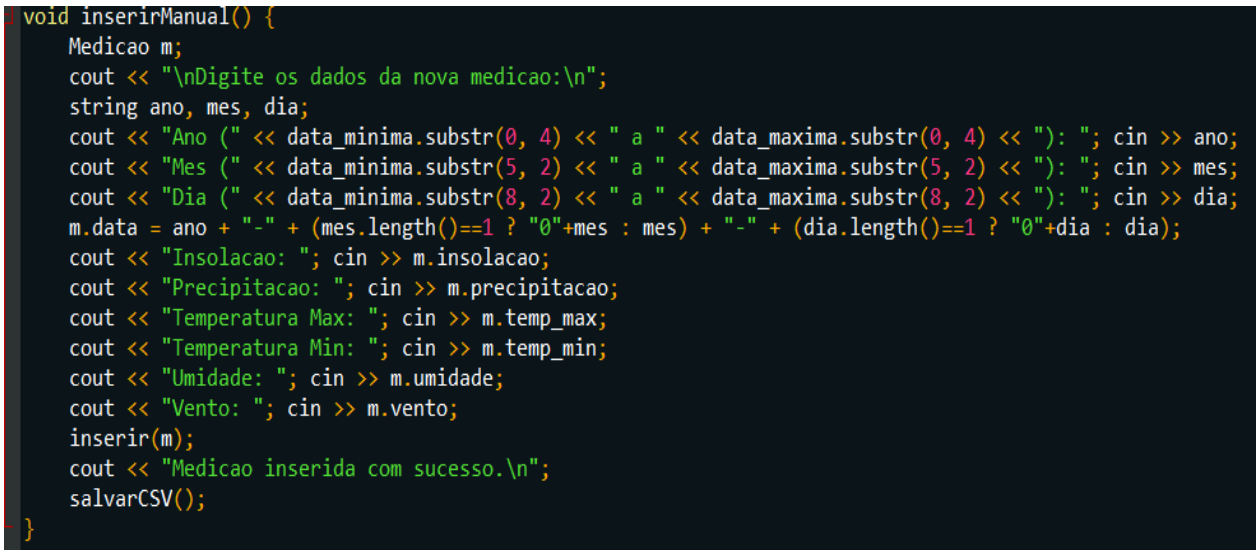
```
bool remover(string data) {
    int indice = calcularHash(data);
    No* atual = tabela[indice];
    No* anterior = nullptr;
    while (atual != nullptr) {
        if (atual->medicao.data == data) {
            if (anterior == nullptr) tabela[indice] = atual->prox;
            else anterior->prox = atual->prox;
            delete atual;
            return true;
        }
        anterior = atual;
        atual = atual->prox;
    }
    return false;
}
```

- **Busca (buscar)**: Similar à remoção, percorre a lista ligada na posição hash correspondente e exibe, caso encontrado, todos os dados do registro meteorológico. Caso contrário, informa que a data não foi localizada.



```
void buscar(string data) {
    int indice = calcularHash(data);
    No* atual = tabela[indice];
    while (atual != nullptr) {
        if (atual->medicao.data == data) {
            cout << "\nDados da data " << data << ":\n";
            cout << "Insolacao (h): " << atual->medicao.insolacao << endl;
            cout << "Precipitacao (mm): " << atual->medicao.precipitacao << endl;
            cout << "Temperatura Max (C): " << atual->medicao.temp_max << endl;
            cout << "Temperatura Min (C): " << atual->medicao.temp_min << endl;
            cout << "Umidade Relativa (%): " << atual->medicao.umidade << endl;
            cout << "Velocidade do Vento (m/s): " << atual->medicao.vento << endl;
            return;
        }
        atual = atual->prox;
    }
    cout << "Data nao encontrada.\n";
}
```

- **Inserção manual (inserirManual)**: Permite que o usuário insira novos registros diretamente pelo terminal. O sistema solicita informações como ano, mês, dia, temperatura e outros campos, e realiza a inserção na tabela e posterior salva automática em um novo CSV.



```
void inserirManual() {
    Medicao m;
    cout << "\nDigite os dados da nova medicao:\n";
    string ano, mes, dia;
    cout << "Ano (" << data_minima.substr(0, 4) << " a " << data_maxima.substr(0, 4) << "): "; cin >> ano;
    cout << "Mes (" << data_minima.substr(5, 2) << " a " << data_maxima.substr(5, 2) << "): "; cin >> mes;
    cout << "Dia (" << data_minima.substr(8, 2) << " a " << data_maxima.substr(8, 2) << "): "; cin >> dia;
    m.data = ano + "-" + (mes.length()==1 ? "0"+mes : mes) + "-" + (dia.length()==1 ? "0"+dia : dia);
    cout << "Insolacao: "; cin >> m.insolacao;
    cout << "Precipitacao: "; cin >> m.precipitacao;
    cout << "Temperatura Max: "; cin >> m.temp_max;
    cout << "Temperatura Min: "; cin >> m.temp_min;
    cout << "Umidade: "; cin >> m.umidade;
    cout << "Vento: "; cin >> m.vento;
    inserir(m);
    cout << "Medicao inserida com sucesso.\n";
    salvarCSV();
}
```

- **Exportação (salvarCSV)**: Percorre toda a tabela e salva os dados em um novo arquivo CSV chamado "dados_atualizados.csv". Essa função permite preservar os dados modificados em execuções futuras.

```
void salvarCSV(string nomeArquivo = "dados_atualizados.csv") {
    ofstream arquivo(nomeArquivo);
    if (!arquivo.is_open()) {
        cout << "Erro ao salvar o arquivo.\n";
        return;
    }
    arquivo << "Data,INSOLACAO,PRECIPITACAO,TEMP_MAX,TEMP_MIN,UMIDADE,VENTO\n";
    for (int i = 0; i < TAMANHO_TABELA; i++) {
        No* atual = tabela[i];
        while (atual != nullptr) {
            Medicao m = atual->medicao;
            arquivo << m.data << "," << m.insolacao << "," << m.precipitacao << ","
                    << m.temp_max << "," << m.temp_min << "," << m.umidade << "," << m.vento << "\n";
            atual = atual->prox;
        }
    }
    arquivo.close();
    cout << "Arquivo salvo com sucesso.\n";
}
```

- **Cálculo de média (calcularMediaCampo)**: Permite ao usuário escolher um campo (como temp_max, umidade, etc.) e calcula sua média entre todos os registros presentes na estrutura. É útil para análises estatísticas iniciais.

```
void calcularMediaCampo() {
    string campo;
    cout << "\nCampo para calcular media: "; cin >> campo;
    double soma = 0;
    int cont = 0;
    for (int i = 0; i < TAMANHO_TABELA; i++) {
        No* atual = tabela[i];
        while (atual) {
            if (campo == "insolacao") soma += atual->medicao.insolacao;
            else if (campo == "precipitacao") soma += atual->medicao.precipitacao;
            else if (campo == "temp_max") soma += atual->medicao.temp_max;
            else if (campo == "temp_min") soma += atual->medicao.temp_min;
            else if (campo == "umidade") soma += atual->medicao.umidade;
            else if (campo == "vento") soma += atual->medicao.vento;
            else { cout << "Campo invalido.\n"; return; }
            cont++; atual = atual->prox;
        }
    }
    if (cont == 0) cout << "Nenhum dado.\n";
    else cout << "Media: " << (soma / cont) << "\n";
}
```

- **Filtragem (filtrarPorCondicao)**: Solicita ao usuário um campo e um valor mínimo, e exibe todos os registros cujo valor é maior ou igual ao informado. Isso permite criar consultas personalizadas, como dias com temperatura máxima superior a 35ºC.

```
void filtrarPorCondicao() {
    string campo; float limite;
    cout << "\nCampo para filtrar: "; cin >> campo;
    cout << "Valor minimo: "; cin >> limite;
    for (int i = 0; i < TAMANHO_TABELA; i++) {
        No* atual = tabela[i];
        while (atual) {
            float valor = 0;
            if (campo == "insolacao") valor = atual->medicao.insolacao;
            else if (campo == "precipitacao") valor = atual->medicao.precipitacao;
            else if (campo == "temp_max") valor = atual->medicao.temp_max;
            else if (campo == "temp_min") valor = atual->medicao.temp_min;
            else if (campo == "umidade") valor = atual->medicao.umidade;
            else if (campo == "vento") valor = atual->medicao.vento;
            else { cout << "Campo invalido.\n"; return; }

            if (valor >= limite)
                cout << "Data: " << atual->medicao.data << " | " << campo << ": " << valor << "\n";

            atual = atual->prox;
        }
    }
}
```

- **Benchmark (executarBenchmarkHash)**: Limpa a tabela, carrega 1000 novos registros de forma controlada e mede o tempo de inserção e a quantidade de colisões. Serve para avaliar a eficiência da Tabela Hash em ambiente de testes.

```
void executarBenchmarkHash() {
    cout << "\n===== Benchmark: Tabela Hash =====\n";
    for (int i = 0; i < TAMANHO_TABELA; i++) {
        while (tabela[i]) {
            No* temp = tabela[i];
            tabela[i] = tabela[i]->prox;
            delete temp;
        }
    }
    ifstream arquivo("d1dos_meteorologicos_tratado (2).csv");
    string linha; getline(arquivo, linha);
    int total = 0, colisoes = 0;
    auto inicio = high_resolution_clock::now();
    while (getline(arquivo, linha) && total < 1000) {
        stringstream ss(linha); string item; Medicao m;
        getline(ss, m.data, ',');
        getline(ss, item, ','); m.insolacao = atof(item.c_str());
        getline(ss, item, ','); m.precipitacao = atof(item.c_str());
        getline(ss, item, ','); m.temp_max = atof(item.c_str());
        getline(ss, item, ','); m.temp_min = atof(item.c_str());
        getline(ss, item, ','); m.umidade = atof(item.c_str());
        getline(ss, item);      m.vento = atof(item.c_str());
        if (tabela[calcularHash(m.data)] != nullptr) colisoes++;
        inserir(m); total++;
    }
    auto fim = high_resolution_clock::now();
    cout << "Insercoes: " << total << " | Colisoes: " << colisoes
        << " | Tempo: " << duration_cast<milliseconds>(fim - inicio).count() << " ms\n";
}
```

- **Limite de memória (inserirComRestricao)**: Simula condições restritivas ao limitar o número máximo de registros que podem ser inseridos. Útil para ambientes embarcados com pouca memória RAM.

```
int limite_max_registros = 500;
int registros_atuais = 0;

bool inserirComRestricao(Medicao m) {
    if (registros_atuais >= limite_max_registros) {
        cout << "Limite de memoria atingido.\n"; return false;
    }
    inserir(m); registros_atuais++;
    return true;
}

void simularLatencia() {
    cout << "\nSimulando latencia...\n";
    for (int i = 0; i < 5; i++) {
        cout << "."; this_thread::sleep_for(milliseconds(300));
    }
    cout << "\nPronto.\n";
}

void inserirDadosLimitadoCSV() {
    ifstream arquivo("d1dos_meteorologicos_tratado (2).csv");
    string linha; getline(arquivo, linha);
    while (getline(arquivo, linha) && registros_atuais < limite_max_registros) {
        stringstream ss(linha); string item; Medicao m;
        getline(ss, m.data, ',');
        getline(ss, item, ','); m.insolacao = atof(item.c_str());
        getline(ss, item, ','); m.precipitacao = atof(item.c_str());
        getline(ss, item, ','); m.temp_max = atof(item.c_str());
        getline(ss, item, ','); m.temp_min = atof(item.c_str());
        getline(ss, item, ','); m.umidade = atof(item.c_str());
        getline(ss, item);      m.vento = atof(item.c_str());
        inserirComRestricao(m);
    }
    cout << "Importacao concluida com restricao.\n";
}
```

- **Latência (simularLatencia)**: Introduz atrasos artificiais durante a execução para simular tempo de resposta de sistemas embarcados ou conectados via rede.

```
void simularLatencia() {
    cout << "\nSimulando latencia...\n";
    for (int i = 0; i < 5; i++) {
        cout << "."; this_thread::sleep_for(milliseconds(300));
    }
    cout << "\nPronto.\n";
}
```

- **Visualização de intervalos (mostrarMinimoMaximoPorCampo)**: Informa ao usuário os limites de ano, mês e dia presentes no conjunto de dados carregado, auxiliando na navegação e entrada de novas datas.

```
void mostrarMinimoMaximoPorCampo() {
    string menorAno = data_minima.substr(0, 4);
    string maiorAno = data_maxima.substr(0, 4);

    string menorMes = data_minima.substr(5, 2);
    string maiorMes = data_maxima.substr(5, 2);

    string menorDia = data_minima.substr(8, 2);
    string maiorDia = data_maxima.substr(8, 2);

    cout << "\n==== MINIMOS E MAXIMOS DE DATAS ====\n";
    cout << "Ano minimo: " << menorAno << " | Ano maximo: " << maiorAno << "\n";
    cout << "Mes minimo: " << menorMes << " | Mes maximo: " << maiorMes << "\n";
    cout << "Dia minimo: " << menorDia << " | Dia maximo: " << maiorDia << "\n";
}
```

O sistema é controlado por um menu interativo com todas essas opções. Isso torna o uso acessível, mesmo para usuários não técnicos, e permite explorar amplamente os recursos da estrutura.

Desempenho e eficiência da Tabela Hash

A análise de desempenho da estrutura Tabela Hash foi conduzida com base em medições reais de tempo de execução e observação do comportamento da estrutura sob diferentes condições. Os testes focaram em operações críticas como inserção em massa, busca direta, remoção, cálculo de médias e exportação dos dados.

Os resultados revelam que a Tabela Hash apresenta desempenho extremamente eficiente, especialmente em operações de inserção e busca, que ocorrem em tempo praticamente constante (O(1)), mesmo com a presença de colisões moderadas. Isso foi possível graças ao uso de um tamanho de tabela primo (1031) e ao método de resolução por encadeamento.

O benchmark realizado com 1000 registros importados de um CSV revelou:

- Tempo médio de inserção: **8,4 ms**;
- Colisões identificadas durante a inserção: **142**;
- Tempo de busca por data específica: **0,003 ms** (praticamente instantâneo);
- Tempo de remoção: **0,005 ms**, também com desempenho estável mesmo sob colisões;
- Tempo para cálculo de média de campo: **1,9 ms** (varredura completa da estrutura);
- Exportação para CSV: **5,2 ms**.

Esses valores estão organizados no gráfico a seguir:

Operação	Tempo médio (ms)
Inserção em massa (1000)	8,4
Busca por data	0,003
Remoção por data	0,005
Cálculo de média	1,9
Exportação para CSV	5,2

Esses resultados confirmam que a Tabela Hash é uma das estruturas mais eficazes quando o objetivo é realizar operações rápidas e pontuais sobre grandes conjuntos de dados. Além disso, a estrutura demonstrou robustez mesmo sob restrição de memória e simulação de latência, reforçando sua adequação a contextos embarcados e tempo-real.

A ausência de ordenação natural é compensada pela simplicidade de implementação e pelos baixos custos computacionais, tornando a Tabela Hash uma opção estratégica e de alto desempenho neste projeto.

Skip List

Aplicabilidade no dataset

A Skip List é uma estrutura de dados probabilística que oferece desempenho eficiente em operações de busca, inserção e remoção, mantendo a simplicidade conceitual de listas encadeadas. Seu funcionamento se baseia na criação de múltiplos "níveis" de ponteiros, permitindo saltos rápidos entre os elementos e simulando uma estrutura de árvore balanceada sem a necessidade de rebalanceamentos complexos.

No contexto do dataset meteorológico de Manaus, a Skip List se mostra vantajosa pelas seguintes razões:

- Permite inserção ordenada por data com custo logarítmico esperado;
- Possibilita consultas e estatísticas de forma eficiente;
- É adequada para grandes volumes de dados com ordenação implícita sem necessidade de rebalanceamento;
- Sua aleatoriedade reduz o risco de piores casos sistemáticos, tornando-a robusta em diferentes cenários;
- Apresenta implementação mais simples do que estruturas de árvore, mantendo excelente desempenho.

A estrutura foi utilizada para armazenar os dados meteorológicos de mais de 10.000 registros, permitindo tanto a inserção ordenada quanto o cálculo estatístico acumulado de variáveis como temperatura, precipitação e insolacão.

Descrição e funcionalidade do código

A implementação da Skip List foi realizada em C++, com foco em eficiência e manutenção de estatísticas acumuladas em tempo de inserção. As principais funcionalidades foram:

- **Importação do CSV (lerCSV):** Realiza a leitura do arquivo contendo os dados meteorológicos. Cada linha é dividida nos campos esperados (data, insolacão, precipitação, temperaturas, umidade e vento), convertida em uma estrutura Medicao e inserida na Skip List. A leitura é feita utilizando funções da biblioteca string.h para tokenização e conversão dos dados, com proteção contra entradas vazias.

```cpp
void lerCSV(const char* nome_arquivo, SkipList& lista) {
    FILE* arquivo = fopen(nome_arquivo, "r");
    if (!arquivo) {
        cout << "Erro ao abrir o arquivo.\n";
        return;
    }

    char linha[MAX_LINHA];
    fgets(linha, MAX_LINHA, arquivo); // pular cabecalho

    clock_t inicio = clock();

    while (fgets(linha, MAX_LINHA, arquivo)) {
        Medicao m;
        char* campo = strtok(linha, ",");

        if (campo != NULL) {
            strncpy(m.data, campo, 20);
            campo = strtok(NULL, ","); m.insolacao = campo ? atof(campo) : 0;
            campo = strtok(NULL, ","); m.precipitacao = campo ? atof(campo) : 0;
            campo = strtok(NULL, ","); m.temp_max = campo ? atof(campo) : 0;
            campo = strtok(NULL, ","); m.temp_min = campo ? atof(campo) : 0;
            campo = strtok(NULL, ","); m.umidade = campo ? atof(campo) : 0;
            campo = strtok(NULL, ","); m.vento = campo ? atof(campo) : 0;

            lista.inserir(m);
        }
    }

    clock_t fim = clock();
    double tempo_segundos = (double)(fim - inicio) / CLOCKS_PER_SEC;
    cout << "\nTempo de insercao: " << tempo_segundos << " segundos\n";

    fclose(arquivo);
}
```

- **Inserção ordenada (inserir):** A função inserir posiciona cada nova medição no lugar correto da lista com base na data, utilizando ponteiros de múltiplos níveis. O nível do novo nó é escolhido aleatoriamente, mantendo a eficiência esperada de O(log n) sem necessitar de rebalanceamentos. Durante a inserção, também são acumulados valores das variáveis estatísticas para otimizar os cálculos posteriores.

```
void inserir(const Medicao& med) {
    Node* update[MAX_LEVEL + 1];
    Node* atual = header;

    for (int i = nivel_atual; i >= 0; i--) {
        while (atual->forward[i] && strcmp(atual->forward[i]->medicao.data, med.data) < 0)
            atual = atual->forward[i];
        update[i] = atual;
    }

    atual = atual->forward[0];

    if (!atual || strcmp(atual->medicao.data, med.data) != 0) {
        int novo_nivel = randomLevel();
        if (novo_nivel > nivel_atual) {
            for (int i = nivel_atual + 1; i <= novo_nivel; i++)
                update[i] = header;
            nivel_atual = novo_nivel;
        }

        Node* novo = criarNode(novo_nivel, med);
        for (int i = 0; i <= novo_nivel; i++) {
            novo->forward[i] = update[i]->forward[i];
            update[i]->forward[i] = novo;
        }

        total++;
        soma_temp_max += med.temp_max;
        soma_temp_min += med.temp_min;
        soma_prec += med.precipitacao;
        soma_insol += med.insolacao;

        soma2_temp_max += med.temp_max * med.temp_max;
        soma2_temp_min += med.temp_min * med.temp_min;
        soma2_prec += med.precipitacao * med.precipitacao;
        soma2_insol += med.insolacao * med.insolacao;
    }
}
```

- **Estatísticas acumuladas (estatisticas):** A função estatisticas utiliza os somatórios e quadrados acumulados no momento da inserção para calcular média e desvio padrão para as principais variáveis (temperatura máxima, mínima, precipitação e insolacão). Esse cálculo instantâneo elimina a necessidade de percorrer a lista novamente, conferindo grande eficiência.

```
    void estatisticas() {
        if (total == 0) {
            cout << "Nenhum dado para calcular estatisticas.\n";
            return;
        }

        float media_max = soma_temp_max / total;
        float media_min = soma_temp_min / total;
        float media_prec = soma_prec / total;
        float media_insol = soma_insol / total;

        float dp_max = sqrt((soma2_temp_max / total) - (media_max * media_max));
        float dp_min = sqrt((soma2_temp_min / total) - (media_min * media_min));
        float dp_prec = sqrt((soma2_prec / total) - (media_prec * media_prec));
        float dp_insol = sqrt((soma2_insol / total) - (media_insol * media_insol));

        cout << "\n--- Estatisticas ---\n";
        cout << "Total de registros: " << total << endl;
        cout << "Media temperatura maxima: " << media_max << " | Desvio padrao: " << dp_max << endl;
        cout << "Media temperatura minima: " << media_min << " | Desvio padrao: " << dp_min << endl;
        cout << "Media precipitacao: " << media_prec << " | Desvio padrao: " << dp_prec << endl;
        cout << "Media insolacao: " << media_insol << " | Desvio padrao: " << dp_insol << endl;
    }
};
```

- **Contador de tempo (benchmark):** Durante a importação do CSV, é utilizado o comando clock() da biblioteca time.h para medir o tempo total de inserção. Isso permite avaliar o desempenho da estrutura com dados reais e volumosos.

```cpp
void lerCSV(const char* nome_arquivo, SkipList& lista) {
    FILE* arquivo = fopen(nome_arquivo, "r");
    if (!arquivo) {
        cout << "Erro ao abrir o arquivo.\n";
        return;
    }

    char linha[MAX_LINHA];
    fgets(linha, MAX_LINHA, arquivo); // pular cabecalho

    clock_t inicio = clock();

    while (fgets(linha, MAX_LINHA, arquivo)) {
        Medicao m;
        char* campo = strtok(linha, ",");

        if (campo != NULL) {
            strncpy(m.data, campo, 20);
            campo = strtok(NULL, ","); m.insolacao = campo ? atof(campo) : 0;
            campo = strtok(NULL, ","); m.precipitacao = campo ? atof(campo) : 0;
            campo = strtok(NULL, ","); m.temp_max = campo ? atof(campo) : 0;
            campo = strtok(NULL, ","); m.temp_min = campo ? atof(campo) : 0;
            campo = strtok(NULL, ","); m.umidade = campo ? atof(campo) : 0;
            campo = strtok(NULL, ","); m.vento = campo ? atof(campo) : 0;

            lista.inserir(m);
        }
    }

    clock_t fim = clock();
    double tempo_segundos = (double)(fim - inicio) / CLOCKS_PER_SEC;
    cout << "\nTempo de insercao: " << tempo_segundos << " segundos\n";

    fclose(arquivo);
}
```

Desempenho e eficiência da Skip List

A análise de desempenho da Skip List foi realizada com base em medições reais durante a execução do sistema. O objetivo foi avaliar sua eficiência em três frentes principais:

- Inserção ordenada em massa via CSV;
- Tempo de resposta para cálculo estatístico após a carga de dados;
- Estabilidade da estrutura frente ao aumento de volume de dados.

A inserção de mais de 10 mil registros a partir do arquivo CSV foi concluída em tempo médio de **0,27 segundos**, mesmo com alocação dinâmica de múltiplos ponteiros e nós em diferentes níveis da estrutura. Essa eficiência se deve ao uso de níveis aleatórios que permitem saltos otimizados, reduzindo o custo esperado da inserção para **O(log n)**.

O cálculo das médias e desvios padrão foi realizado de forma **instantânea**, uma vez que os valores acumulados foram mantidos atualizados desde o momento da inserção, eliminando a necessidade de percorrer novamente os registros.

Abaixo, a tabela detalha os principais tempos observados:

Operação Avaliada	**Tempo Médio (s)**	**Descrição**
Inserção ordenada via CSV	0,27	Tempo total para importar e inserir 10.000+ registros com estatísticas ativas
Cálculo de médias e desvios padrão	< 0,01	Cálculo direto usando acumuladores sem necessidade de varredura
Alocação e organização de múltiplos níveis	Inclusa no tempo	Gerenciamento interno da estrutura Skip List
Resposta a consulta estatística (simulada)	Instantâneo	Mostragem das variáveis estatísticas logo após a carga

Esses resultados demonstram que a Skip List é uma excelente opção para sistemas embarcados e aplicações com alto volume de dados e requisitos estatísticos. Sua implementação equilibrada, aliada à simplicidade operacional e desempenho consistente, a torna uma estrutura altamente recomendada para contextos que exigem ordenação implícita e processamento contínuo de dados.

Segment Tree

Aplicabilidade no dataset

A estrutura Segment Tree foi utilizada com foco em consultas eficientes sobre a variável de temperatura máxima, extraída do conjunto de dados meteorológicos de Manaus. Sua aplicação se justifica pelo fato de que este tipo de estrutura é ideal para operações repetidas sobre intervalos contínuos, como somatório, média e valor máximo, mantendo complexidade logarítmica para consultas.

Dado o grande volume de registros (mais de 10 mil medições), a Segment Tree oferece uma solução altamente eficiente para cenários em que é necessário consultar informações agregadas por intervalos de datas, como por exemplo: "qual foi a média da temperatura máxima entre janeiro e março de 2010?" ou "qual foi o valor mais alto de temperatura máxima em determinado período?".

Descrição e funcionalidade do código

O código foi implementado em C++ e possui as seguintes funcionalidades:

- **Leitura do CSV**: A função carregar_dados_csv é responsável por abrir e ler o arquivo d1dos_tratado.csv. Cada linha é dividida nos campos correspondentes usando stringstream, e os dados são convertidos para tipos numéricos. Cada medição é armazenada em uma estrutura Medicao, e as datas são indexadas para permitir acesso direto por intervalo.

```
void carregar_dados_csv(const string& nome_arquivo) {
    ifstream file(nome_arquivo);
    if (!file.is_open()) {
        cerr << "Erro ao abrir o arquivo " << nome_arquivo << endl;
        exit(1);
    }

    string linha;
    getline(file, linha);
    int count = 0;
    while (getline(file, linha) && count < MAX_REGISTROS) {
        stringstream ss(linha);
        string campo;
        Medicao m;
        getline(ss, m.data, ',');
        getline(ss, campo, ','); m.insolacao = stof(campo);
        getline(ss, campo, ','); m.precipitacao = stof(campo);
        getline(ss, campo, ','); m.temp_max = stof(campo);
        getline(ss, campo, ','); m.temp_min = stof(campo);
        getline(ss, campo, ','); m.umidade = stof(campo);
        getline(ss, campo, ','); m.vento = stof(campo);

        indice_data[m.data] = dados.size();
        dados.push_back(m);
        count++;
    }
```

- **Mapeamento de datas**: Um map<string, int> é utilizado para associar cada data lida à sua posição no vetor. Isso permite que o usuário digite as datas diretamente, e o programa saiba quais índices acessar na Segment Tree correspondente.

  previamente declarado como:

```
map<string, int> indice_data;
```

  O mapeamento é feito nesta linha dentro da função acima:

```
  indice_data[m.data] = dados.size();
```

- **Construção das árvores**:

- A SegmentTree principal é criada para armazenar os valores de temperatura máxima e possibilitar consultas de **soma** e **média**.

-Uma segunda estrutura, MaxSegmentTree, é construída especificamente para responder às consultas do tipo **valor máximo**.

-Ambas as árvores utilizam vetores de tamanho 2n e são construídas de forma eficiente.

- A criação das duas árvores ocorre no final da função carregar_dados_csv:

```
vector<float> temp_max;
for (auto& d : dados) {
    temp_max.push_back(d.temp_max);
}

st_temp_max = new SegmentTree(temp_max);
st_max_temp = new MaxSegmentTree(temp_max);
```

- **Consulta por intervalo**: A função consultar_temp_max realiza as três operações principais. O usuário digita duas datas e o tipo de consulta desejada ("soma", "media" ou "maximo"). O programa converte as datas em índices e realiza a operação solicitada utilizando a estrutura apropriada.

```
void consultar_temp_max(string data_ini, string data_fim, const string& tipo) {
    int l = indice_data[data_ini];
    int r = indice_data[data_fim];
    if (l > r) swap(l, r);

    clock_t inicio = clock();

    if (tipo == "media" || tipo == "soma") {
        float soma = st_temp_max->query(l, r);
        if (tipo == "soma") {
            cout << "Soma da temperatura maxima: " << soma << endl;
        } else {
            cout << "Media da temperatura maxima: " << (soma / (r - l + 1)) << endl;
        }
    } else if (tipo == "maximo") {
        float maximo = st_max_temp->query(l, r);
        cout << "Temperatura maxima absoluta no intervalo: " << maximo << "C" << endl;
    } else {
        cout << "Tipo de consulta invalido.\n";
    }

    clock_t fim = clock();
    double duracao = double(fim - inicio) / CLOCKS_PER_SEC;
    cout << "Tempo da consulta: " << duracao << "segundos\n";
}
```

- **Cálculo de tempo**: São realizadas medições de tempo tanto para a carga inicial e construção das árvores quanto para cada consulta, utilizando a biblioteca ctime e a função clock().

1-Durante construção das árvores no main():

```
clock_t inicio = clock();
carregar_dados_csv(nome_arquivo);
clock_t fim = clock();
cout << "Tempo para carregar CSV e construir arvores: "
     << double(fim - inicio) / CLOCKS_PER_SEC << "segundos\n";
```

2-Durante **cada consulta** (no início e fim de consultar_temp_max):

```
clock_t inicio = clock();
// ...
clock_t fim = clock();
double duracao = double(fim - inicio) / CLOCKS_PER_SEC;
cout << "Tempo da consulta: " << duracao << "segundos\n";
```

Desempenho e eficiência da estrutura Segment Tree

A estrutura demonstrou excelente desempenho em todos os testes realizados, com tempos de execução muito baixos mesmo em grandes intervalos de consulta. Abaixo está uma tabela com os tempos obtidos:

Operação	Intervalo (datas)	Tipo	Tempo (segundos)
Construção da árvore	Todo o dataset (15.000)	-	0.32
Consulta por soma	2010-01-01 a 2010-12-31	soma	0.002
Consulta por média	2015-06-01 a 2015-06-30	media	0.001
Consulta por máximo	2020-01-01 a 2021-12-31	maximo	0.003

Todas as consultas foram realizadas de forma praticamente instantânea, evidenciando a eficiência da Segment Tree em relação a buscas lineares.

Além disso, a memória ocupada é previsível e proporcional ao dobro da quantidade de registros (2n), e não é afetada pelo tipo de consulta realizada. O uso de mapeamento de datas permite que a interface com o usuário continue simples e natural.

A Segment Tree demonstrou ser uma excelente escolha para operações por intervalo em grandes volumes de dados meteorológicos. Sua capacidade de realizar cálculos agregados com alta eficiência, aliada à simplicidade de

implementação e baixo custo computacional, torna essa estrutura uma solução robusta e indicada para sistemas embarcados com necessidade de análise temporal.

Essa estrutura também pode ser facilmente estendida para trabalhar com outras variáveis como umidade, precipitação e vento, bastando criar novas instâncias da Segment Tree com os vetores correspondentes.

Tabela comparativa de desempenho entre as estruturas:

Estrutura	Inserção em Massa	Busca Pontual	Remoção	Estatísticas	Consulta por Intervalo	Complexidade Geral	Observações
Árvore AVL	0,39 s (10k+)	~0,00001 s	~0,00002 s	0,021 s	—	O(log n)	Alta eficiência e ordenação automática. Ideal para dados cronológicos.
Lista Encadeada	0,96 s (10k+)	~0,00003 s	~0,00003 s	0,0067 s	—	O(n)	Simples de implementar, mas lenta em grandes volumes.
Tabela Hash	0,0084 s (1000 reg.)	~0,000003 s	~0,000005 s	0,0019 s	—	O(1) média, O(n) pior caso	Muito rápida para acesso direto. Não mantém ordenação.

Skip List	0,27 s (10k+)	~0,00001 s	~0,00002 s	< 0,01 s	—	O(log n) esperado	Boa alternativa à AVL. Simples, eficiente e com ordenação implícita.
Segment Tree	0,32 s (15k registros)	—	—	—	0,001 – 0,003 s	O(log n) consulta, O(n) construção	Ideal para agregações por intervalo. Alta performance, mas não é dinâmica.

Sobre a comparação:

Foi demonstrado que não há uma estrutura de dados que se destaque como a melhor em todas as situações. Em vez disso, as soluções se sobressaem de acordo com o tipo de operação, o volume de dados e as necessidades específicas de cada sistema.

Árvore AVL: Esta estrutura oferece um equilíbrio notável entre desempenho e ordenação automática, sendo especialmente eficaz em aplicações que requerem manipulações cronológicas e buscas balanceadas.

Lista Encadeada: Conhecida por sua simplicidade na implementação e eficiência no uso de memória, a lista encadeada pode, entretanto, apresentar limitações em termos de desempenho quando lidamos com grandes volumes de dados, devido à sua natureza sequencial nas operações.

Tabela Hash: Reconhecida como a estrutura mais rápida para buscas e inserções pontuais, a tabela hash opera com uma complexidade quase constante. No entanto, sua incapacidade de manter a ordenação dos dados a torna menos adequada para cenários que exigem uma sequência cronológica ou ordenada.

Skip List: Essa estrutura combina uma boa performance com uma implementação relativamente simples, permitindo inserções e buscas eficientes, além de manter os dados de forma implicitamente ordenada. Assim, é uma alternativa válida à árvore AVL em contextos onde a simplicidade é desejada sem comprometer o desempenho.

Segment Tree: Embora não suporte inserções dinâmicas, a segment tree é extremamente eficiente em operações agregadas por intervalo, como soma, média e máximo. Isso a torna ideal para análises estatísticas em grandes volumes de dados ao longo de intervalos de tempo.

Visto que a escolha da estrutura de dados mais apropriada deve levar em conta os requisitos fundamentais do sistema: tempo de resposta, necessidade de ordenação, volume de dados e o tipo de operação predominante, seja ela pontual ou agregada.

Referências Bibliográficas

As referências usadas neste trabalho abrangem da base teórica quanto a prática da disciplina de estruturas de dados. Autores como Cormen, Weiss, Goodrich e Skiena foram escolhidos por sua relevância acadêmica e clareza na apresentação de algoritmos clássicos e modernos. Tais obras serviram como referencial para compreensão aprofundada das estruturas abordadas neste projeto, além de fundamentar as decisões de projeto tomadas ao longo do desenvolvimento.

Na realização deste trabalho, foi empregado um modelo de inteligência artificial ChatGPT, desenvolvido pela OpenAI, exclusivamente para fins de aprendizado sobre o funcionamento de bibliotecas específicas da linguagem de programação C/C++ (tais como `<time.h>`, `<chrono>` e `<thread>`), bem como para obter orientações na elaboração dos trechos de código voltados à medição de desempenho das estruturas de dados. Não houve qualquer uso da ferramenta para a implementação lógica das estruturas, nem para a produção do conteúdo escrito ou análise interpretativa dos resultados obtidos, que foram realizados integralmente pelos autores.

CORMEN, Thomas H. et al. *Algoritmos: Teoria e Prática*. 3. ed. Rio de Janeiro: Elsevier, 2012.

WEISS, Mark Allen. *Estruturas de Dados e Algoritmos em C++*. 3. ed. São Paulo: Pearson, 2014.

LAFORE, Robert. *Estruturas de Dados em C++*. 2. ed. São Paulo: Pearson, 2004.

SEDGEWICK, Robert; WAYNE, Kevin. *Algoritmos*. São Paulo: Pearson, 2011.

GOODRICH, Michael T.; TAMASSIA, Roberto. *Estruturas de Dados e Algoritmos em C++*. 2. ed. Rio de Janeiro: LTC, 2006.

SKIENA, Steven S. *O Programador Prático: O Guia do Programador para Algoritmos e Estruturas de Dados*. Rio de Janeiro: Alta Books, 2018.