Trabalho Prático 2
Sistema de Escalonamento Logístico

Lucas Pimenta Braga - 2023034552

[1]Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brasil

lucaspimentabraga@gmail.com

1. Introdução

Este trabalho tem como objetivo implementar e analisar um **sistema logístico para o transporte de pacotes entre armazéns**, utilizando a técnica de **simulação de eventos discretos**. A simulação automatiza o processo de movimentação e armazenamento de pacotes em uma **rede de armazéns interligados**, modelada como um **grafo não direcionado**, em que os **vértices representam os armazéns** e as **arestas indicam rotas de transporte** sujeitas a **restrições de capacidade e latência**. O modelo incorpora estruturas **LIFO (Last-In, First-Out)** para gerenciamento do armazenamento temporário de pacotes, conforme a lógica dos **Armazéns de Hanoi**. O sistema foi desenvolvido com foco em **modularidade, eficiência computacional e escalabilidade**.

A simulação é conduzida por meio de **eventos discretos** que representam mudanças de estado do sistema — como **chegada, armazenamento, remoção, transporte e entrega de pacotes** — e cuja execução ocorre em **ordem cronológica**, controlada por uma **fila de prioridade**. Cada pacote possui uma **rota pré-calculada** entre origem e destino, determinada via **algoritmo de busca em largura (BFS)** sobre a matriz de adjacência. Ao final, realiza-se uma **análise experimental de desempenho computacional**, na qual são variáveis os parâmetros de entrada, como **número de pacotes, número de armazéns e volume de rearmazenamentos**, com o objetivo de avaliar a **robustez estrutural e a capacidade de otimização do sistema sob diferentes cenários operacionais**.

2. Método

O sistema foi estruturado para ser **eficiente e escalável**, com foco na **avaliação de desempenho sob diferentes cenários de carga**. A implementação foi realizada em **C++**, utilizando **estruturas de dados estáticas** (como arrays e matrizes) e executada em ambiente **Linux**, com uso exclusivo das bibliotecas padrão da linguagem.

A arquitetura do sistema baseia-se em quatro **Tipos Abstratos de Dados (TADs)** principais, cada um com responsabilidades específicas no contexto da simulação. O **TAD Pacote** armazena dados como **origem, destino, estado atual**, bem como a **rota calculada via algoritmo BFS** e estatísticas de tempo. O **TAD Armazém** gerencia o **armazenamento temporário de pacotes** em seções entre armazéns, utilizando **pilhas LIFO** que seguem a lógica dos **Armazéns de Hanoi**. O **TAD Transporte** define os procedimentos para **remoção, reordenação e envio de pacotes**, embora o transporte em si ocorra indiretamente por meio de **eventos programados**. Já o **TAD Escalonador** implementa uma **fila de**

prioridade sobre estrutura sequencial de tamanho fixo, organizada por **chaves compostas (tempo, ID e tipo de evento)**, garantindo a execução **cronológica, determinística e estável** dos eventos.

A simulação segue o paradigma de **eventos discretos**, que modelam o comportamento dinâmico do sistema ao longo do tempo. O fluxo principal envolve:

- **Leitura dos parâmetros de simulação** a partir de arquivo de entrada;
- **Processamento dos pacotes**, incluindo roteamento, armazenamento, transporte e entrega;
- **Gerenciamento da fila de eventos**, que controla a ordem de execução com base em prioridade temporal.

2.1. Estruturas de Dados Utilizadas

2.1.1. Matriz de Adjacência (adj):

A matriz adj, **de tamanho** $MAX\_ARMAZENS \times MAX\_ARMAZENS$ representa a conectividade entre os armazéns. Cada célula $adj[i][j]$ vale 1 se houver rota entre os armazéns i e j , e 0 caso contrário. Como o grafo é **não direcionado**, vale $adj[i][j] = adj[j][i]$. Essa estrutura é essencial para o cálculo de rotas, utilizando o algoritmo de **Busca em Largura (BFS)**.

2.1.2. Pilhas de Pacotes ($secoes$):

Cada par de armazéns possui uma pilha **LIFO (Last-In, First-Out)** associada para armazenar temporariamente os pacotes. A estrutura Pilha contém um **array fixo** de pacotes e um índice de topo, que aponta para o último elemento inserido. O desempilhamento respeita a ordem inversa de chegada, garantindo a lógica LIFO conforme o modelo dos **Armazéns de Hanoi**.
Cada seção de armazém é tratada como uma pilha independente, permitindo que o último pacote colocado seja o primeiro a ser removido para transporte. Quando os pacotes estão prontos para seguir para o próximo armazém, são desempilhados respeitando essa ordem.

2.1.3. Pacotes:

Cada pacote é definido por uma estrutura que armazena: id (identificador único), $origem$ e $destino$ (armazéns de partida e chegada), $estado$ (como "em trânsito", "armazenado" ou "entregue"), $tempo\_chegada$ (instante de entrada na simulação), $rota[100]$ (sequência de armazéns definida por BFS), $tamanho\_rota$ (comprimento da rota) e pos_rota (posição atual do pacote na rota). Durante a simulação, os pacotes percorrem essa rota passando por operações de armazenamento, remoção e transporte até a entrega final.

2.1.4. Fila de Eventos (FilaEventos):

A fila de eventos controla a execução cronológica da simulação de eventos discretos. É implementada como um **array fixo de eventos**, com capacidade limitada (até 1000 elementos). Cada evento representa uma mudança de estado no sistema e possui campos como tipo, tempo, origem, destino e ID do pacote. Os eventos são ordenados por prioridade utilizando uma chave baseada no tempo e outros atributos, garantindo execução correta e determinística.

Tipos de evento tratados:

- **Armazenamento**: Chegada de pacote a um armazém.
- **Remoção**: Retirada do pacote da seção de armazenamento.
- **Transporte**: Deslocamento do pacote entre armazéns.
- **Rearmazenamento**: Pacote que não pôde ser transportado e é retornado ao armazenamento.
- **Entrega**: Finalização do transporte, com o pacote entregue ao destino.

2.2. Funções Implementadas

2.2.1. Função $executar\_simulacao$:

Responsável por iniciar e coordenar a execução da simulação. Suas principais etapas são:

- **Leitura dos dados de entrada**: Carrega parâmetros como capacidade de transporte, latência, intervalo entre transportes, custo de remoção e a matriz de adjacência que define a conectividade entre armazéns.
- **Inicialização das estruturas**: Configura as pilhas ($secoes$), os pacotes ($pacotes$[]) e a fila de eventos ($fila$).
- **Processamento inicial dos pacotes**: Calcula a rota de cada pacote via BFS e insere na fila um evento inicial de armazenamento.
- **Agendamento dos eventos de transporte**: Com base na matriz de adjacência, são programados eventos periódicos de transporte para todas as conexões válidas.
- **Execução da simulação**: Enquanto houver eventos na fila e pacotes não entregues, extrai-se o próximo evento e executa-se a ação correspondente (armazenamento ou transporte).

2.2.2. Função $processar\_transporte$:

Lida com o envio de pacotes de um armazém a outro. Etapas principais:

- **Remoção de pacotes da pilha ($secoes[origem][destino]$)**: Todos os pacotes armazenados na seção são desempilhados, respeitando a ordem LIFO.
- **Reordenação dos pacotes removidos**: Os pacotes são invertidos em memória para que os mais antigos sejam transportados primeiro, conforme exigência do modelo.
- **Transporte dos pacotes**: Até o limite de capacidade, os pacotes são transportados. Um novo evento de chegada é agendado para cada um.
- **Rearmazenamento dos excedentes**: Pacotes não transportados retornam à pilha.
- **Agendamento do próximo transporte**: Um novo evento de transporte entre os mesmos armazéns é inserido na fila para ocorrer após o intervalo definido.

2.2.3. Função $calcular\_rota$:

Determina a rota que o pacote seguirá, do armazém de origem ao destino, utilizando o algoritmo de **Busca em Largura (BFS)**:

- **Busca no grafo**: A matriz de adjacência é percorrida para encontrar o caminho mais curto entre os armazéns.
- **Reconstrução do caminho**: O caminho encontrado é armazenado no array fixo $rota[]$ do pacote.
- **Atualização dos campos**: Os atributos $tamanho\_rota$ e pos_rota são definidos para orientar o avanço do pacote na simulação.

2.2.4. Funções de Impressão:

Cada tipo de evento executado na simulação gera uma saída descritiva formatada, facilitando a análise da execução.
São utilizadas funções específicas para cada tipo de evento:

- $\boldsymbol{imprimir\_evento\_armazenado}$: Pacote armazenado em uma seção.
- $\boldsymbol{imprimir\_evento\_transporte}$: Pacote em trânsito entre armazéns.
- $\boldsymbol{imprimir\_evento\_removido}$: Pacote removido da pilha para transporte.
- $\boldsymbol{imprimir\_evento\_rearmazenado}$: Pacote que retornou ao armazenamento.
- $\boldsymbol{imprimir\_evento\_entregue}$: Pacote entregue ao armazém de destino.

Essas funções utilizam printf com formatação padronizada (zeros à esquerda, alinhamento), conforme exigido no projeto.

2.2.5. Funções Auxiliares:

Suporte à operação correta das estruturas:

- Inserção de eventos na fila**: Garante ordenação cronológica por meio de deslocamento estável no array fixo de eventos.**
- Manipulação das pilhas**: Funções como $\boldsymbol{empilha}$, $\boldsymbol{desempilha}$ e $\boldsymbol{topo}$ controlam a estrutura LIFO das seções.**
- Comparação de eventos ($\boldsymbol{evento\_menor}$)**: Geração de chave de prioridade e comparação para ordenar os eventos conforme tempo, ID e tipo.**

3. Análise de Complexidade

Esta seção avalia o desempenho dos algoritmos implementados no sistema, considerando tempo de execução e uso de memória. A seguir, são analisadas as principais funções da simulação dos **Armazéns Hanoi**.

3.1. Função $\boldsymbol{executar\_simulacao}$:

Responsável por carregar os dados, inicializar estruturas e gerenciar a execução da simulação.

- **Complexidade de Tempo**: Itera sobre n pacotes e insere eventos na fila, que é processada cronologicamente. A manipulação da fila tem complexidade $O(mlogm)$, onde $m \approx n$. Assim, temos $O(nlogn)$.

- **Complexidade de Espaço**: São alocados os arrays de pacotes, eventos e estruturas auxiliares, resultando em $O(n+m)$.

3.2. Função $processar\_transporte$:

Remove pacotes da pilha, transporta até o limite de capacidade e rearmazena o restante.

- **Complexidade de Tempo**: Remove e processa até n pacotes, resultando em $O(n)$.
- **Complexidade de Espaço**: Utiliza um array auxiliar de tamanho n, com complexidade $O(n)$.

3.3. Função $calcular\_rota$:

Usa Busca em Largura (BFS) sobre a matriz de adjacência para definir a rota do pacote.

- Complexidade de Tempo**:** A BFS percorre $\boldsymbol{n}$ vértices e até $\boldsymbol{n^2}$ arestas (em um grafo completo), resultando em $O(n^2)$**.**
- Complexidade de Espaço: Utiliza arrays auxiliares de tamanho $\boldsymbol{n}$, com complexidade $\boldsymbol{O(n)}$**.**

3.4. Funções de Impressão:

Imprimem eventos como armazenamento, transporte, remoção e entrega.

- **Complexidade de Tempo:** Cada evento gera uma chamada de impressão. Para $\boldsymbol{m}$ eventos, temos $O(m)$.
- **Complexidade de Espaço:** Não alocam estruturas auxiliares significativas: $O(1)$.

3.6. Funções Auxiliares:

Incluem inserção na fila de eventos, comparação de chaves e operações com pilhas.

- **Complexidade de Tempo**:
    - **Inserção Ordenado na fila**: A fila é ordenada a cada inserção: $O(mlog\ m)$.
    - **Empilhar/Desempilhar**: $O(1)$ por operação, pois as pilhas são implementadas com acesso direto ao topo.
    - **Comparação de eventos**: A comparação entre dois eventos tem complexidade $\boldsymbol{O(1)}$, pois envolve apenas a comparação de duas chaves de prioridade.
- **Complexidade de Espaço**: Fila de eventos e pilhas armazenam, juntas, até $n\ +\ m$ elementos: $O(n+m)$.

4. Estratégias de Robustez

A robustez de um sistema se refere à sua capacidade de funcionar corretamente e manter a estabilidade mesmo em situações inesperadas ou adversas, como entradas inválidas, falhas de memória ou eventos de execução não previstos. No desenvolvimento deste sistema de simulação de eventos discretos para os Armazéns Hanoi, foram implementadas diversas estratégias de programação defensiva e tolerância a falhas para garantir que o sistema opere de maneira confiável e eficiente.

4.1. Inicialização Adequada e Gerenciamento de Memória

A correta inicialização das variáveis e estruturas é essencial, especialmente em um sistema baseado em **arrays fixos** e pilhas manuais. Evita-se o uso de memória não inicializada, acessos inválidos e possíveis **falhas de segmentação (segfaults)**.

Como foi implementado:

- Pilhas de pacotes são inicializadas com $topo = -1$;
- Arrays e contadores são zerados antes do uso;
- A manipulação de índices respeita os limites superiores e inferiores dos arrays.

4.2. Verificação de Limites e Acesso Seguro à Memória

Para garantir acesso seguro aos dados, o sistema valida previamente os índices usados em arrays e pilhas. A maioria das falhas críticas em C++ decorre de acessos fora dos limites dos arrays ou desempilhamento de estruturas vazias.

Aplicações no código:

- Antes de desempilhar, verifica-se se $topo >= 0$;
- Ao acessar a matriz de adjacência $adj[i][j]$, os índices i e j são verificados;
- O algoritmo de roteamento (BFS) não continua se a rota não puder ser construída.

4.3. Tratamento de Erros em Arquivos de Entrada

O sistema realiza validações no momento da leitura do arquivo de entrada, assegurando que ele esteja acessível e em formato válido. Evita-se que a simulação opere com dados inconsistentes ou cause falhas por entradas malformadas.

Detalhes técnicos:

- O código checa se $entrada.is\_open()$ retorna verdadeiro;
- O programa encerra com mensagem clara se o arquivo não for encontrado;
- Os dados são lidos com cuidado quanto ao tipo e ordem esperados (ex: tempo, id, origem, destino).

4.4. Validação de Eventos e Garantia de Ordem

A simulação depende da ordem correta de execução dos eventos, que são processados cronologicamente por meio da fila de eventos. Eventos fora de ordem comprometeriam a integridade da simulação (ex: um pacote ser transportado antes de ser armazenado).

Como é garantido:

- A função $inserir\_evento$ insere os eventos na fila em ordem estável de prioridade (baseada em tempo, tipo e ID);

- $extrair\_proximo$ sempre retorna o evento mais antigo (com menor tempo), garantindo o fluxo correto.

4.5. Tolerância a Falhas e Monitoramento de Estado

O sistema foi projetado para continuar operando corretamente mesmo quando ocorrem restrições logísticas ou falhas parciais. Tendo como objetivo manter a simulação funcional, mesmo sob limitações, garantindo a entrega gradual dos pacotes.

Cenários tratados:

- Se a **capacidade de transporte** for atingida, os pacotes excedentes são **rearmazenados** automaticamente;
- O transporte é reavaliado em ciclos, com **novos eventos agendados** conforme o sistema permite;
- Em falhas de roteamento, o pacote é descartado e o erro é informado sem interromper a simulação.

Essas estratégias tornam o sistema mais confiável e preparado para lidar com cenários adversos. Ao combinar verificação preventiva, tratamento explícito de erros e reescalonamento inteligente, o modelo simula com precisão e segurança o comportamento logístico proposto.

5. Análise experimental

Esta seção apresenta os experimentos realizados para avaliar o desempenho do sistema logístico dos Armazéns Hanoi, com foco no impacto de diferentes parâmetros no tempo de execução.

5.1. Plano Experimental

Os testes foram conduzidos variando **um parâmetro por vez**, com os demais mantidos fixos. Os parâmetros controlados foram:

- **Tamanho do grafo (número de armazéns)**: 10-150.
- **Número de pacotes**: 10-850.
- **Número de rearmazenamentos**: 10-10000.

Parâmetros mantidos constantes durante os testes:

- **Capacidade de transporte**: 2 pacotes por ciclo.
- **Latência de transporte**: 30 segundos.
- **Intervalo entre transportes**: 120 segundos.
- **Custo de remoção**: 4 segundos.
- **Número de armazéns**: 8 (exceto nos testes que o variam).
- **Número de pacotes**: 15 (exceto nos testes que o variam).
- **Matriz de adjacência**: fixa e conectada.

5.2. Procedimento Experimental

1. **Geração dos arquivos de entrada**: script automatizada, com base na combinação de parâmetros.
2. **Execução da simulação**: o sistema foi executado até a entrega de todos os pacotes, e o tempo total foi registrado.
3. **Análise dos resultados**: o tempo de execução foi avaliado conforme a variável controlada.
4. **Geração de gráficos**: foram produzidos gráficos comparativos para ilustrar o impacto dos parâmetros no desempenho do sistema.

5.3. Resultados e Discussão

Como dito anteriormente, foram conduzidos experimentos variando individualmente três parâmetros principais — número de armazéns, número de pacotes e número de rearmazenamentos — enquanto os demais se mantiveram constantes. O objetivo foi avaliar o impacto de cada um no **tempo de execução** da simulação.

a. Variação do Número de Armazéns (Tamanho do Grafo)

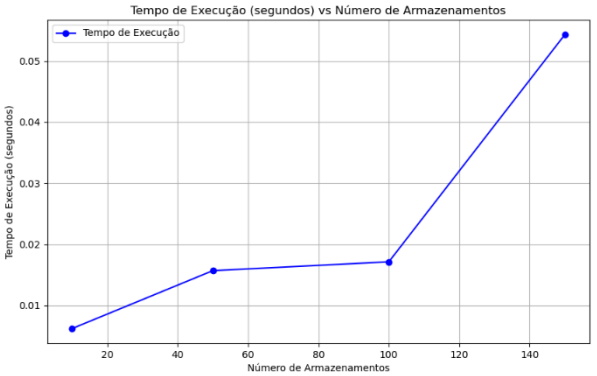
Com 15 pacotes e 100 rearmazenamentos fixos, variou-se o número de armazéns entre 10, 50, 100 e 150. Observou-se que o **tempo de execução diminuiu** com o aumento da conectividade da rede, devido à maior disponibilidade de rotas, que resultou em transporte mais eficiente e menor congestionamento.

Ressalta-se, porém, que **não foi possível testar valores acima de 150 armazéns**, pois o sistema passou a apresentar **falhas de segmentação ou travamentos**, devido à limitação da matriz de adjacência estática ($adj[MAX\_ARMAZENS][MAX\_ARMAZENS]$), definida em tempo de compilação.

Tabela 1 – Tempo de execução da simulação para diferentes quantidades de armazéns

Número de Armazéns	Tempo de Execução (s)
10	0.00619591
50	0.0156803
100	0.017127
150	0.054323

Gráfico 1 – Tempo de execução vs. Número de Armazéns



b. Variação do Número de Pacotes

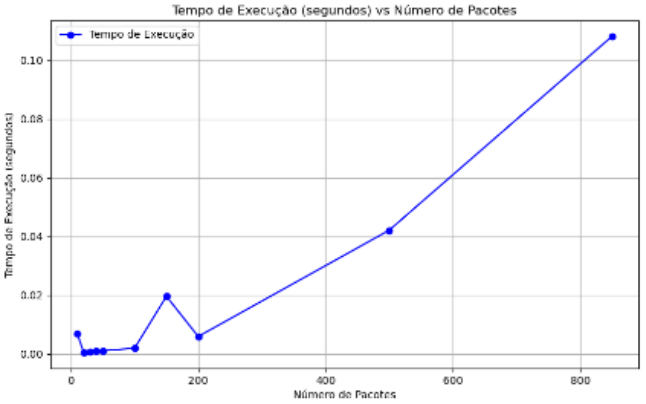
Com 8 armazéns e 100 rearmazenamentos, variou-se o número de pacotes entre 10 e 850. Os resultados mostraram um **aumento não linear no tempo de execução**, especialmente após 200 pacotes. Acima de 850 pacotes, o sistema passou a apresentar instabilidade, como travamentos

ou falhas de segmentação (*segfaults*), provocadas pelo estouro do array estático de pacotes e pelo excesso de uso da pilha (*stack overflow*), característicos de estruturas alocadas estaticamente na memória de execução.

Tabela 2 – Tempo de execução da simulação para diferentes quantidades de pacotes

Número de Pacotes	Tempo de Execução (s)
10	0.00682813
20	0.000386083
30	0.000535618
40	0.000866631
50	0.000971853
100	0.00185618
150	0.0195547
200	0.00585417
500	0.0420954
850	0.108214

Gráfico 2 – Tempo de execução vs. Número de Pacotes



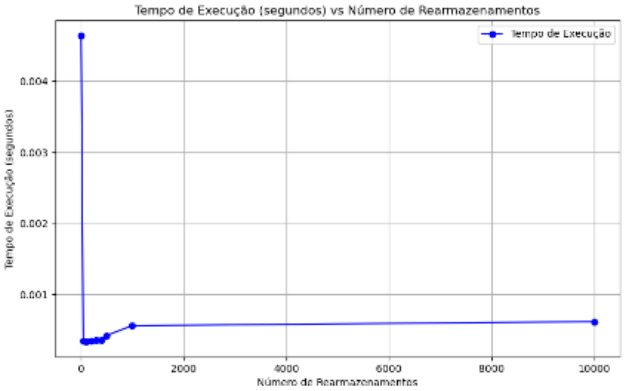
c. Variação do Número de Rearmazenamentos

Com 8 armazéns e 15 pacotes fixos, foram testados valores entre 10 e 10.000 rearmazenamentos. Os resultados mostraram **impacto mínimo** no tempo de execução, evidenciando a **eficiência do sistema em lidar com rearmazenamentos**, mesmo em volumes elevados.

Tabela 3 – Tempo de execução da simulação para diferentes quantidades de rearmazenamentos.

Número de Rearmazenamentos	Tempo de Execução (s)
10	0.00463921
50	0.000349641
100	0.000337101
200	0.000350296
300	0.000353533
400	0.000359461
500	0.000421532
1000	0.000560595
10000	0.00061806

Gráfico 3 – Tempo de execução vs. Número de Rearmazenamentos



5.4. Discussão e Conclusões Experimentais

A comparação entre os resultados experimentais e a análise de complexidade teórica confirmou a consistência do modelo simulado, evidenciando os seguintes pontos técnicos:

- **Número de Pacotes:** Com complexidade $O(n \log n)$, associada à ordenação e ao gerenciamento da fila de eventos, o número de pacotes mostrou-se o principal fator de impacto no desempenho. A partir de 200 pacotes, observou-se crescimento acentuado no tempo de execução, e acima de 850, ocorreram falhas de segmentação ou travamentos, causados por estouro da stack ou saturação de arrays fixos.
- **Número de Armazéns:** Apesar do custo teórico $O(n^2)$ no cálculo de rotas via BFS, o aumento no número de armazéns contribuiu para reduzir o tempo de execução, graças à maior conectividade do grafo e rotas mais curtas. A verificação de conectividade em tempo constante (via matriz estática) favoreceu o desempenho. Entretanto, acima de 150 armazéns, a simulação se tornou instável devido ao alto consumo de memória causado pela matriz $adj[MAX][MAX]$.
- **Número de Rearmazenamentos:** Com operações de custo $O(1)$, esse parâmetro teve impacto mínimo, mesmo com 10.000 ocorrências. Os pacotes foram bem distribuídos ao longo da simulação, sem sobrecarga das pilhas, e a estrutura LIFO provou-se eficiente para lidar com altos volumes.

Esses resultados reforçam os limites e eficiências do modelo em diferentes cenários, indicando onde estão os principais gargalos e como as escolhas estruturais impactam a escalabilidade do sistema

6. Conclusões

Este trabalho apresentou o desenvolvimento de um sistema de simulação de eventos discretos voltado à logística dos Armazéns Hanoi, implementado em C++ com estruturas estáticas. A abordagem orientada a eventos e o uso de grafos, pilhas e filas de prioridade permitiram modelar com precisão o transporte e armazenamento de pacotes.

A implementação mostrou-se funcional e coerente com a análise de complexidade, além de ilustrar os efeitos práticos de diferentes estratégias de estruturação e controle. As limitações impostas pela alocação estática, observadas em cenários mais exigentes, reforçaram a importância de escolhas arquiteturais bem fundamentadas para garantir desempenho e escalabilidade.

Além de cumprir seu propósito técnico, o projeto proporcionou aprendizado prático sobre estruturas de dados, controle de fluxo e simulação, consolidando fundamentos essenciais da disciplina.

7. Bibliografia

1. ANISIO MENDES LACERDA; WAGNER MEIRA JÚNIOR. *Slides virtuais da disciplina de Estruturas de Dados – 2024*. Disponibilizado via Moodle. Departamento de Ciência da Computação, Universidade Federal de Minas Gerais, Belo Horizonte, 2024.

2. GEEKSFORGEEKS. *Difference between static and dynamic memory allocation in C*. GeeksforGeeks, 18 jan. 2023. Disponível em: https://www.geeksforgeeks.org/difference-between-static-and-dynamic-memory-allocation-in-c/. Acesso em: 18 jun. 2025.

3. TANENBAUM, Andrew S.; LANGSAM, Yedidyah; AUGENSTEIN, Moshe J. *Estruturas de dados usando C*. 2. ed. São Paulo: Pearson, 2009.