UNIVERSIDADE ESTADUAL PAULISTA “JÚLIO DE MESQUITA FILHO” - UNESP
INSTITUTO DE CIÊNCIA E TECNOLOGIA DE SOROCABA
BACHARELADO EM ENGENHARIA DE CONTROLE E AUTOMAÇÃO
ESTRUTURA DE DADOS

GUILHERME ROCHA RODRIGUES 221271112
ÍTALO AQUINO CASTRO 241270201

TRABALHO EM GRUPO ESTRUTURA DE DADOS

Dataset: Machine Failure.

Sorocaba-SP
2025

GUILHERME ROCHA RODRIGUES
ÍTALO AQUINO CASTRO

TRABALHO EM GRUPO ESTRUTURA DE DADOS

Dataset: Machine Failure.

Trabalho apresentado ao curso de engenharia de controle e automação da Universidade Estadual Paulista, campus de Sorocaba, como requisito parcial para obtenção de nota na disciplina de Estrutura de dados.

Orientador: Prof. Dr. Leopoldo André Dutra Lusquino Filho.

SOROCABA - SP
2025

SUMÁRIO

1. INTRODUÇÃO

O presente trabalho utiliza do dataset Predictive Maintenance e auxílio de inteligência artificial em 5 estruturas de dados, 3 vistas na matéria e 2 fora da emenda de aula, com análise dos dados de 10.000 amostras de processos de uma fresa(equipamento utilizado para realizar o processo de usinagem mecânica), sendo esses dados usados no desenvolvimento das estruturas em funções de inserção, remoção, busca, estatística, filtragem e classificação. Sendo essas funções e organização dos dados realizadas de diferentes maneiras para cada estrutura, devido à própria característica de cada uma.

Por fim, a tarefa no qual todos os códigos realizam é a de encontrar padrões de falha pelos dados existentes no dataset a fim de em um ambiente simulado seja possível identificar por meio das variações dos dados a tendência de falha em uma fresa.

Por fim, a tarefa no qual todos os códigos realizam é a de encontrar padrões de falha pelos dados existentes no dataset a fim de em um ambiente simulado seja possível identificar por meio das variações dos dados a tendência de falha em uma fresa.

2. OBJETIVOS

2.1. OBJETIVO GERAL

Implementar 5 estruturas de dados das quais 3 da emenda do curso e 2 extras a fim de analisar e comparar suas implementações e capacidades.

2.2. OBJETIVOS ESPECÍFICOS

- Implementar as funções básicas de cada estrutura;
- Adicionar funções complementares;
- Testar as capacidades de cada estrutura por uso de benchmark;
- Testar as estruturas com determinadas restrições;
- Realização de determinada tarefa e simulação.

3. METODOLOGIA

Foram implementadas 5 estruturas de dados, sendo 3 da emenda do curso e 2 fora, além da otimização da fila em fila circular, às implementação do código para o trabalho foi realizada com auxílio de inteligência artificial(todos os códigos e .exe podem ser vistos no tópico 9 deste trabalho):

- Lista duplamente encadeada;
- Fila Circular;
- Árvore AVL;
- Skip List;
- Segment tree moderna.

Foram utilizadas de suas funções básicas de inserção, remoção e busca, além de funções de filtragem, estatística e classificação.

Sendo as estruturas avaliadas pelas métricas de benchmark:

- Tempo de Inserção;
- Tempo de Remoção;.
- Tempo de Busca;.
- Uso de Memória.
- Tempo Médio de Acesso;.
- Escalabilidade;.

- Latência Média.

As estruturas também foram testadas com as seguintes restrições:

- R2: Restrição no tamanho máximo das estruturas de dados;
- R10: Execução forçada com interrupções a cada 100 ms para simular concorrência com outros processos.;
- R13: Inserção artificial de processamento em lote, forçando atrasos antes da execução;
- R18: Simulação de sensores defeituosos, retornando valores anômalos em 10% das medições;
- R24: Uso de algoritmos menos otimizados.

Todos esses tópicos seguiram as seguintes etapas:

- Implementação:
    - Foi utilizado de linguagem C no software Dev C++ para a implementação do código;
    - Cada estrutura foi feita em um arquivo .cpp separadamente;
    - As funções implementadas foram acessadas por meio de um menu interativo por meio da função switch case;
- Testes e Validação:
    - Após a implementação, foram realizados testes em ambiente simulado com geração de novos dados aleatórios para teste de benchmarks;
    - Para teste das restrições também foi implementado por meio de simulação com geração de dados aleatórios.
- Tarefa:
    - Para a realização da tarefa foram implementadas duas funções;
    - Inicialmente com base no dataset o código entende certos padrões de situações de quando a fresa apresenta falha, como temperatura, torque, entre outras…
    - Após aprender os padrões, é possível gerar uma simulação de novas dados em que o código detectar possíveis casos de falha nos dados simulados;

4. IMPLEMENTAÇÃO DAS ESTRUTURAS

Todas as estruturas possuem as mesmas opções, sendo implementadas as funções básicas de exibição, busca (por ID, por tipo ou por falha) inserção ou remoção manual das amostras.

Ademais, as estruturas foram implementadas para cálculos estatísticos das variáveis, classificar as falhas por tipo de máquina e calcular suas ocorrências, realizar uma filtragem para pesquisas específicas das amostras baseadas nas variáveis, além de funções para testar benchmarks, restrições e simulação de tarefas.

4.1. Lista Duplamente Encadeada

A lista duplamente encadeada conta com ponteiros apontando para a cabeça (primeiro elemento) e para calda (último elemento), contando com nós que apontam para o elemento posterior e anterior.

Sendo a lista iniciada vazia por initList, na função append é puxado uma amostra do dataset e adicionado na cauda (tail) aumentando a o tamanho da lista, e por fim há uma função para limpar a lista.

4.2. Fila Circular

A fila circular, como observado na figura 6, possui estrutura muito parecida com a fila circular comum, com sua remoção normalmente na cabeça (front) e adição na cauda (tail).

4.3. Árvore AVL

Na árvore Avl(figura 8) é criada uma árvore (AVLTree) que aponta para os nós (AVLNode) que apontam para os filhos a direita e esquerda usando de recursividade para implementar isso, para definir o chave de cada nó foi usado da variável UDI, adicionado a esquerda UDI menores e a direita UDI maiores.

Para determinar a altura dos filhos de um nó, foi implementado max, servindo com a função getBalance para verificar se o nó está balanceado, se a diferença entre alturas esquerda e direita for menor ou igual a 1.

Posteriormente, houve a criação de um novo nó do tipo AVLNode, inicialmente vazio. Conforme os itens vão sendo inseridos, é observada a necessidade de balancear. Se sim, há rotação a esquerda ou direita.

4.4. Skip List

A SkipList funciona como uma lista: iniciando na cabeça, com cada nó apontando para o nó no próximo nível (forward) utilizando da variável UDI para ordenação.

A implementação de um novo nó ocorre por meio de “atalhos”, na SkipList, usando srand() e randomLevel para que a implementação de níveis para os nós seja aleatória, com todos os nós tendo nível 0. Porém, menores níveis têm alta chance de implementação e maiores têm baixa chance.

4.5. Segment Tree Moderna

A segment tree é usada para manipular ou acessar dados segmentados. Sua estrutura se organiza em variáveis de máximo, mínimo e soma, otimizando o código para permitir que os cálculos estatísticos sejam mais rápidos.

Utilizar Potência de 2 é útil para transformar uma Segment Tree Moderna em uma Árvore Binária, permitindo preencher completamente a árvore.

Quando a árvore é iniciada pela função nextPower Of Two, o tamanho inicial da árvore é realocado de modo para um tamanho de potência de 2 mais próximo do original, exemplo se o tamanho original for 5, a potência de 2 mais próxima é 8.

Quando o tamanho máximo da árvore é preenchido, é realocado para potência de 2 mais próxima em resizeSegmentTree. Já em updateNode, os índices dos nós são atualizados com base nos filhos a esquerda (2 * pos) e a direita (2 * pos +1) definindo:

- Máximo: O maior valor entre o máximo do filho esquerdo e o máximo do filho direito.
- Mínimo: O menor valor entre o mínimo do filho esquerdo e o mínimo do filho direito.
- Soma: A soma dos valores de soma do filho esquerdo e do filho direito.

A inicialização de um novo nó leva em conta com base na capacidade (capacity) e tamanho (size) da árvore como observado na função:
int pos = st->capacity + size.

5. ANÁLISE DE DESEMPENHO

Os testes de benchmarks foram realizados, em simulações, nas mesmas condições, sendo essas simulações feitas com dados aleatórios a partir dos dados do dataset.

5.1. Tempo de inserção

O tempo de inserção foi testado de duas maneiras, uma com 1000 amostras aleatórias e outra com 10000, sendo o primeiro apresentando um tempo médio de 0.656 ms (milisegundos), já o segundo teve média de 6.236 ms. A estrutura Fila Circular foi a que levou menos tempo para inserir as amostras, tanto com 1000 ou 10000, provavelmente devido a não ter que ficar alocando e realocando itens, como nas árvores. Porém, SkipList e Lista Duplamente Encadeada, para o aumento de 10000 amostras, apresentaram uma maior lentidão, provavelmente devido a fatores como o cálculo de nível, em SkipList, atrasando um pouco a inserção.

5.2. Tempo de remoção

As remoções foram testadas com 1000 ops(operações) apresentando um tempo médio de 228.233 ms (o que é uma média alta), mas, visto um desvio 422,72 ms, demonstra como os dados estão altamente dispersos. Um destaque a ser dado é para a Fila Circular, tempo de remoção de 0.008 e velocidade de remoção de 131578.9 ops/ms (demonstrando uma resposta de remoção quase instantânea, provavelmente devido a simplesmente ir retirando itens da cabeça sem necessidade de balanceamento por exemplo). Mas, a Segment Tree Moderna é algo que chama muita atenção e demonstra o porquê da alta dispersão entre os dados, com tempo de 1069.085 ms e 0.9 ops/ms levando mais de um segundo para executar 1000 ops, devido a característica da árvore de reconstruir o nó sempre que um dado é removido, mas, se comparar com Árvore AVL, há uma diferença de tempo de mais de um segundo devido a própria AVL ser optimizada para inserção, busca e

remoção, diferente da Segment Tree.

5.3. Tempo de busca

Para simular as buscas, foram realizada simulações de 10000 ops, em cada estrutura, o que não necessariamente todas as simulações apresentaram encontrar os itens simulados, mas analisando apenas os dados concretos a um tempo médio de 388.194 ms e desvio padrão de 316.492 ms evidenciado por dois extremos, enquanto Lista Duplamente Encadeada, Fila Circular e Segment Tree apresentaram ser mais lentas com tempos superior a 600 ms. SkipList e Árvore AVL, que são duas estruturas otimizadas para busca, apresentaram uma alta velocidade quase instantânea, isso se deve a como essas estruturas são feitas, já que a SkipList possui “atalhos” de nível e AVL organiza seus nós de modo que diminua o caminho a se percorrer para achar algo.

5.4. Uso de memória

Para analisar o uso de memória de cada estrutura, foi analisado o quanto consomem para armazenar um limite de 10000 itens, analisando apenas o quanto cada item consome de memória. Esse consumo varia com média de 88.6 bytes por elemento armazenado e desvio padrão de 46.702 bytes, além do valor consumido para o total de 10000 amostras com média de 866000 e desvio padrão de 467016.06. Os diferentes consumos de memória podem ser fatores de como a estrutura se organiza, por exemplo a SkipList, em que, além dos dados, cada nó armazena dados de posição e a informação das camadas em que está presente. Embora essa organização seja para deixar a estrutura mais rápida, ela apresenta um consumo de memória relativamente alto, em comparação a Fila Circular, que tem um baixo consumo, de apenas 44 bytes, devido a cada espaço na fila ser dedicada a apenas os itens do elemento, além da característica da própria Fila Circular que consegue reutilizar espaços vazios devido a sua circularidade, o que diminui a memória utilizada.

5.5. Tempo médio de acesso

Nessa simulação, foi medido o tempo médio de acesso de 10000 amostras aleatórias, o que demonstrou dois extremos, por um lado Lista Duplamente Encadeada e SkipList tiveram um tempo de acesso elevado de 188.608 ms e 189.022 ms, respectivamente, o que se deve a característica linear de uma lista tendo que iniciar na cabeça e ir acessando os dados a qual eles apontam, enquanto as outras são otimizadas para serem mais rápidas como as árvores estruturadas em uma rede de nós interligados diminuindo distâncias, e Fila Circular que, mesmo sendo linear, sua circularidade permite conectar a cabeça a cauda, evitando deslocamentos dos itens acelerando os acessos.

5.6. Escalabilidade

Para testar a escalabilidade, foram simuladas inserções de 1000, 5000, 10000, 20000 e 50000, comparando como o tempo varia ao inserir mais dados, sendo que a maioria das estruturas apresentaram uma proporcionalidade em relação ao tempo médio de inserção. Apenas a Árvore AVL demonstrou ficar mais lenta com mais inserções, devido a ter que rebalancear toda vez que um novo elemento é inserido.

5.7. Latência Média

Para testar a latência, foram simuladas 1000 ops de inserção, remoção e busca, de forma combinada, em que os dados mostraram estar bem dispersos com 3 grupos, um grupo baixo de Árvore AVL e SkipList, um grupo médio com Lista Duplamente Encadeada e Fila Circular, e a Segment Tree com alta latência de 19.779 ms. O primeiro grupo é devido a serem otimizados para essas funções, como já discutido anteriormente, por terem uma espécie de “atalho” para essas operações. Lista Duplamente Encadeada e Fila Circular apresentam operações lineares, já a Segment Tree é bem mais lenta, com tempo de 19.779 ms, isso se deve a ela ser focada em cálculos estatísticos, mas não apresentar otimização para busca, inserção e remoção.

6. ANÁLISE DAS RESTRIÇÕES

6.1. Lista Duplamente Encadeada

A análise de desempenho da Lista Duplamente Encadeada, operando sob condições simuladas, demonstrou eficiência variável entre as operações, com maior impacto na busca e acesso aleatório.

Resultados Principais:

- Tempo Total e Elementos: O tempo total para geração de dados, com 4 restrições aplicadas, foi de 728.771 ms, com a lista atingindo sua capacidade máxima de 500 elementos (R2). A restrição R24 (ordenação ineficiente) foi aplicada.
- Desempenho por Operação:
    - Busca (10.000 ops): Concluída em 41.018 ms (243.8 ops/ms). O baixo número de elementos encontrados (148/10.000) é consistente com a simulação de dados anômalos (R18). O tempo reflete a característica O(N) da busca sequencial em listas encadeadas.
    - Remoção (1.000 ops): Realizada em 4.035 ms (247.8 ops/ms). A remoção, em uma lista duplamente encadeada, é O(1), se o nó a ser removido já for conhecido, mas se a busca precede a remoção, o tempo total inclui a busca (O(N)). O tempo observado sugere que a operação de remoção, em si, é eficiente, mas pode incluir o tempo para localizar o elemento se ele não for o primeiro ou o último.
    - Acesso Aleatório (10.000 acessos): Efetuado em 4.342 ms (2303.2 acessos/ms). O acesso aleatório, em uma lista encadeada, exige travessia sequencial, a partir de uma das extremidades, resultando em desempenho O(N). Este tempo é significativamente maior do que o acesso O(1) em estruturas baseadas em array, como a Fila Circular.
- Uso de Memória: A memória total, estimada para 500 nós, é de 27.83 kB. Cada nó ocupa 57 bytes (estimativa), enquanto sizeof(Node) é 64 bytes. O padding do compilador é uma consideração para o tamanho real. A estrutura MachineData ocupa 44 bytes.

6.2. Fila Circular

A análise de desempenho da Fila Circular, sob condições simuladas, revelou eficiência particular nas operações de fila, com distinções no desempenho da busca.
Resultados Principais:

- Tempo Total e Elementos: O tempo total para geração de dados foi de 739.955 ms, com a fila atingindo sua capacidade máxima de 500 elementos (R2). A restrição R24 (ordenação ineficiente) foi explicitamente aplicada.
- Desempenho por Operação:
    - Busca (10.000 ops): Concluída em 53.529 ms (186.8 ops/ms). O baixo número de encontrados (148/10.000) é consistente com a simulação de dados anômalos (R18). O tempo reflete a característica O(N) da busca sequencial em filas.
    - Dequeue (250 ops): Extremamente rápida, com 0.003 ms (80645.2 ops/ms), confirmando a eficiência O(1) da remoção.
    - Acesso Aleatório (10.000 acessos): Efetuado em 0.130 ms (77041.6 acessos/ms), indicando acesso O(1) eficiente a elementos indexados.
- Uso de Memória: A memória total, estimada para 500 elementos, é de 21.51 kB, com cada item de dado ocupando 44 bytes e a estrutura da fila 24 bytes (sizeof(CircularQueue)). O padding do compilador é uma consideração para o tamanho real.

6.3. Árvore AVL

A análise de desempenho da Árvore AVL, sob condições(interrupções, atrasos forçados e dados anômalos), demonstrou notável eficiência.
Resultados Principais:

- Desempenho: As operações de busca, remoção e acesso aleatório foram concluídas em tempos na casa dos milissegundos (ex: 10.000 buscas em 1.095 ms), evidenciando a eficiência da AVL.
- Restrições e Impacto:
    - Concorrência (R10) e Atrasos (R13): Estas foram as restrições mais impactantes, elevando o tempo total de execução para 739.627 ms e influenciando a pureza de benchmarks.

    - Dados Anômalos (R18): Justifica o baixo número de elementos encontrados em buscas (301 de 10.000), simulando falhas em sensores.
    - Tamanho Limitado (R2): A árvore foi testada no seu limite de 500 elementos.
- Uso de Memória: A memória total estimada para 500 nós é de 31.75 KB, com cada nó ocupando 65-72 bytes (incluindo padding do compilador).
- Limitações do Cenário: Embora a AVL seja otimizada, a restrição de "algoritmos menos otimizados (R24)", se aplicada em outras partes do sistema, poderia impactar o desempenho geral.

6.4. Skip List

A análise de desempenho da Skip List, sob condições simuladas, demonstrou alta eficiência em operações de busca e remoção, com desempenho variável para acesso aleatório.

Resultados Principais:

- Tempo Total e Elementos: O tempo total para geração de dados com 4 restrições aplicadas foi de 716.344 ms, com a lista contendo o número máximo de 500 elementos (R2).
- Desempenho por Operação:
    - Busca (10.000 ops): Concluída em 0.692 ms (14450.9 ops/ms). O baixo número de elementos encontrados (135/10.000) é consistente com a simulação de dados anômalos (R18). Este tempo de busca é notavelmente baixo, comparável ao de uma AVL (O(logN), em média), destacando a eficiência da Skip List para esta operação.
    - Remoção (1.000 ops): Realizada em 0.379 ms (2637.1 ops/ms). A remoção na Skip List também é eficiente, com complexidade média O(logN), e o tempo registrado reflete essa característica, sendo muito rápido para a quantidade de operações.
    - Acesso Aleatório (10.000 acessos): Efetuado em 6.691 ms (1494.6 acessos/ms). Embora a Skip List seja eficiente para busca, o "acesso aleatório" puro (por índice, por exemplo) não é sua operação primária e

pode ter que percorrer múltiplos níveis, resultando em um tempo maior que o de busca e remoção para este tipo específico de acesso.

- Uso de Memória: A memória total estimada para 500 nós é de 84.67 kB. Cada nó ocupa 173 bytes (estimativa), enquanto sizeof(SkipNode) é 184 bytes. A estrutura MachineData ocupa 44 bytes. O padding/alignment pelo compilador é uma consideração para o tamanho real dos nós, e o tamanho maior de SkipNode se deve aos múltiplos ponteiros de "níveis" da Skip List.

6.5. Segment Tree Moderna

A análise de desempenho da Segment Tree Moderna, sob condições simuladas de estresse, demonstrou eficiência notável em operações de acesso aleatório, porém com desempenho variável em busca e remoção.

Resultados Principais:

- Tempo Total e Elementos: O tempo total para geração de dados com 4 restrições aplicadas foi de 725.186 ms, com a estrutura contendo 500 elementos finais de uma capacidade de 1024. A restrição R24 (ordenação ineficiente) foi aplicada.
- Desempenho por Operação:
    - Busca (10.000 ops): Concluída em 46.901 ms (213.2 ops/ms). O baixo número de elementos encontrados (145/10.000) é consistente com a simulação de dados anômalos (R18). Embora a Segment Tree seja otimizada para consultas de intervalo, uma "busca" por elemento individual pode exigir a travessia de partes da árvore, impactando o tempo.
    - Remoção (1.000 ops): Realizada em 37.615 ms (26.6 ops/ms). A remoção direta em uma Segment Tree pode ser complexa e não é uma operação primária ou naturalmente eficiente como em outras estruturas, potencialmente envolvendo atualizações em múltiplos nós ancestrais, o que se reflete no tempo e no baixo número de operações por milissegundo.
    - Acesso Aleatório (10.000 acessos): Efetuado em 0.152 ms (65789.5 acessos/ms). Este é um resultado incrível, indicando que o acesso a

posições ou valores específicos (como uma consulta de ponto) é extremamente rápido na Segment Tree, característica de sua natureza baseada em array e acessos logarítmicos ou constantes para certas operações.

- Uso de Memória: A memória total, estimada para 2048 nós (capacidade de 1024 elementos), é de 208.00 kB. Cada nó ocupa 104 bytes, enquanto sizeof(MachineData) é 44 bytes. O padding/alignment pelo compilador é uma consideração para o tamanho real dos nós.

7. TAREFA E SIMULAÇÃO

A tarefa proposta, em todos os códigos, é identificar padrões de falha, com base nos padrões do dataset(função 12 do menu), e identificar possíveis falhas em uma simulação(função 12), sendo os dados de falha obtidos através de cálculo e identificações das temperaturas, torque, rotação, entre outros. Quando as amostras apresentaram falhas, assim identificando em simulação de inserções de dados aleatórios uma possível falha, todas apresentaram os mesmos resultados, sem demonstrar grandes alterações.

8. CONCLUSÃO

O trabalho analisou, de diferentes formas, que as estruturas de dados, escolhidas, apresentaram eficiências e problemas diferentes, com a escolha para a melhor estrutura dependendo da tarefa ou necessidade. Abordando de forma geral, Árvore AVL e SkipList são melhores para remoções, inserções e buscas de forma rápida, mas SkipList, por exemplo, tem um alto consumo de memória e Árvore AVL, em escalabilidade, demonstra ficar mais tempo com aumento de inserções, devido ao rebalanceamento, Fila Circular mostrou ser boa para ter um baixo consumo de memória, porém não otimizada para buscas, Segment Tree é focada em cálculos estatísticos, mas não otimizada para buscas, inserções ou remoções, e Lista Duplamente Encadeada boa para remoção e lenta para acessos. Para a tarefa proposta, a melhor estrutura depende da ideia, ou seja, para cálculos estatísticos de aprender falhas, Segment Tree aparenta ser melhor, porém lenta para buscas e identificação de falhas, enquanto SkipList e Árvore AVL são mais rápidas para as identificações.

9. REFERÊNCIAS E MATERIAL COMPLEMENTAR

- **IAs generativas:**
  - GEMINI. Google. Disponível em: https://gemini.google.com/app?hl=pt-BR. Acesso em: 9 jun. 2025.
  - OPENAI. Disponível em: https://openai.com/. Acesso em: 9 jun. 2025.
  - MANUS. Disponível em: https://manus.im/app. Acesso em: 9 jun. 2025.
  - DEEPSEEK. Disponível em: https://www.deepseek.com/. Acesso em: 9 jun. 2025.

- **Material contendo os códigos e os executáveis:**
  - https://drive.google.com/drive/folders/1lq0Phi9rY7wNMRills92n7il1rl1yCqt?usp=drive_link

- **Dataset:**
  - https://drive.google.com/file/d/1oNbKsZ_LQGT9w9fxObhyzHrdTXzjeAh4/view?usp=drive_link