Trabalho Prático 3 de Estruturas de Dados - Consultas ao Sistema Logístico

Lara Amélia Maia de Freitas

2024005912

Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG) - Belo Horizonte – MG – Brazil

laraameliamf@gmail.com

1. Introdução

O problema proposto foi a implementação de um mecanismo para realização de consultas sobre o sistema logístico da empresa "Armazéns Hanói". Assim, o principal objetivo foi armazenar corretamente as informações de eventos ocorridos no sistema da empresa e realizar, sobre esse registro de eventos, pesquisas acerca de pacotes ou clientes envolvidos. Para tanto, foram modeladas e empregadas diversas estruturas de dados e índices de pesquisa, cujos papéis vão desde representantes de elementos do mundo real (como as classes Pacote e Evento) até índices específicos para favorecer a realização de certas consultas sobre os dados existentes.

Esta documentação tem como fim descrever a implementação do sistema (o que será feito nas seções 2, "Método" e 4, "estratégias de robustez"), além de analisar o seu funcionamento e custo (o que será feito nas seções 3, "análise de complexidade", 5, "análise experimental" e 6, "conclusões"). A seção 7 conta com a bibliografia utilizada no desenvolvimento do trabalho.

2. Método

O sistema foi inteiramente desenvolvido em linguagem C++ e compilado com o compilador g++ da GNU Compiler Collection, especificamente na versão c++11. O código está organizado em, além do Makefile, mais 12 arquivos, sendo 6 deles do tipo .hpp (headers dos TAD's utilizados na solução e do template criado para a árvore AVL) e 6 do tipo .cpp (implementações dos headers e da main, que define o fluxo principal do programa). Para solucionar o problema, foram definidas 6 classes próprias do sistema.

A classe arvoreAVL, como o nome sugere, implementa uma árvore AVL, e é utilizada pelos 3 índices desenvolvidos no sistema com o fim de garantir que as pesquisas sejam realizadas sobre uma árvore balanceada. Para ser utilizada por cada um dos índices sem haver necessidade de reimplementação e repetição de código, a classe foi desenvolvida como um template. Assim, os nodes da árvore contém um campo chave e um campo valor que podem variar, e operam de acordo com esses tipos. A classe realiza as funções básicas de uma árvore AVL (inserção, checagem de altura e fatores de balanceamento, rotações, dentre outros), e contém também métodos adaptados a demandas específicas do sistema, como o percurso in-order pela árvore coletando os dados dos nodes visitados. Não foram implementados métodos para remoção na árvore, já que eles não são necessários no sistema.

A classe Pacote representa um Pacote do mundo real e é utilizada, basicamente, para facilitar a atualização de ponteiros fim no índice de clientes. Um objeto pacote armazena o ID do pacote e os seus atributos remetente e destinatário, de forma que, quando é registrado um evento para um pacote, para saber em quais clientes deve ser feita a atualização do ponteiro fim, basta obter o remetente e destinatário associados a ele. Os métodos da classe são na maioria getters, setters e construtores, mas também há o método encontraPacote, que acha o índice de um pacote com um certo ID no vetor de Pacotes existente na main.

A classe Evento representa os eventos que ocorrem no mundo real, são registrados no sistema logístico e devem ser devidamente processados para a realização de consultas posteriores. Os atributos que cada evento possui dependem do tipo de evento e são especificados pelo arquivo de entrada. À medida que as linhas da entrada são processadas, os eventos são criados e armazenados no vetor de eventos da main, que é o único que de fato conta com objetos Evento. Ao longo do sistema, os eventos armazenados no vetor são referenciados via sua posição nesse vetor (é o que ocorre no índice de clientes) ou por ponteiros para os eventos (é o que ocorre no índice pacoteTempo). Os principais métodos da classe são os construtores (adaptados para os atributos que cada tipo de evento armazena) e métodos getters (permitem o acesso a dados do evento). Além disso, há o método geraResposta, que faz a impressão de informações do evento no formato especificado, e o método ordenaPorTempoId, que ordena os eventos em um vetor de acordo com o seu tempo de ocorrência e com o id do pacote associado.

A classe indiceCli implementa o índice a ser utilizado para facilitar consultas de histórico de pacote de clientes. A classe é utilizada como campo de valor para uma árvore AVL que é instanciada na main e representa o índice de clientes em si. Seus atributos são o nome do cliente, o número de

pacotes associado a ele e uma segunda árvore AVL aninhada. Essa árvore, por sua vez, tem ID de pacotes como chave (int) e, como valor, um objeto da classe infoPacotes, que conta com os "ponteiros" para os eventos de início e para o evento mais recente de um certo pacote ("fim"). Nesse caso, foram utilizados ponteiros "indiretos" para os eventos - os campos início e fim armazenam os índices dos respectivos eventos no vetor de eventos principal do sistema, instanciado e criado na main. Assim, sempre que é feita uma busca, obtemos esses índices e a extração de dados é realizada sobre o vetor principal em si. Os métodos das classes são basicamente construtores, destrutores, getters e setters. Em indiceCli, também temos o método addPacote, que percorre a árvore de valor indiceCli em busca de um certo cliente e, se o encontra, insere o pacote na árvore aninhada do cliente. O método mudaPacote busca um pacote e, se ele existir, muda seu ponteiro para fim. Por último, o método atualizaFim engloba a atualização de fim a ser realizada na main sempre que um novo evento é registrado para um pacote.

A classe indicePacs implementa o índice de pacotes, que apenas armazena o id de um pacote e o momento no tempo em que ele foi registrado. Assim, a classe conta, basicamente, com uma árvore AVL que tem esses dados como chave e valor, respectivamente. A principal função desse índice é permitir a construção da chave que será utilizada para busca no índice pacoteTempo, indicando em qual node desse índice a busca deve se iniciar. Os métodos da classe são, portanto, para registrar um novo pacote (registraPacote), checar a existência de um pacote (existePac) e produzir a chave a ser empregada no índice pacoteTempo (geraChaveParaBuscaTempo).

A classe indicePacTempo implementa o índice pacoteTempo. Também é, essencialmente, uma árvore AVL, tendo uma string como chave e ponteiros para eventos (que referenciam o vetor principal de eventos na main) como valor. A string-chave utilizada é construída a partir do id do pacote, do tempo do evento associado e do tipo do evento, que é utilizado como critério de desempate para casos em que há mais de um evento associado a um pacote em um mesmo instante de tempo. Os métodos da classe são utilizados para geração de chaves (geraChave), inserção de nodes na árvore (registraEvento, que constrói a chave e insere o node correspondente na AVL), busca por nodes na árvore/índice (buscaEvento) e coleta de eventos a partir de uma determinada chave (eventosDoPacoteAteTempo).

O funcionamento do sistema de pesquisas é definido em main.cpp. Temos a instanciação de cada um dos índices, de um vetor de pacotes e de um vetor de eventos, que é o vetor principal referenciado por todo o sistema. As linhas do arquivo de entrada são todas lidas e armazenadas em um vetor de strings, com o fim de evitar a concomitância entre ações do sistema e leitura do arquivo de entrada. Após a leitura e armazenamento, processamos cada linha (na ordem em que foram dadas no arquivo de entrada em si) e, com isso, processamos também os eventos e consultas à medida que ocorrem. Dessa forma, não é necessário se preocupar com os eventos que ocorram em um momento "futuro" à pesquisa, já que, a qualquer momento, os índices e os vetores do sistema contém somente as informações dadas até aquele instante. Caso a linha a ser processada indique um evento, criamos o novo evento e realizamos as ações necessárias (inserção no índice pacoteTempo, criação de clientes e inserção no índice de clientes, atualização de ponteiros para fim no índice de clientes, dentre outros) e, caso a linha seja de algum tipo de consulta, são chamados os métodos que realizam as consultas em cada índice. Para o índice de clientes, percorremos a árvore do índice de clientes até encontrar o cliente correto. Em seguida, percorremos a árvore aninhada de pacotes associados coletando as informações sobre eventos de início e fim de cada um dos pacotes. Os dados são, então, ordenados pelo tempo de ocorrência e pelo id dos pacotes e, por fim, é realizada a impressão das informações com o método geraRespostas. Para consultas de pacotes, primeiro percorremos o índice de Pacotes em busca do node que tem o id desejado como chave. A partir das informações nesse node, é construída a chave que indicará em qual node deve se iniciar a busca no índice pacoteTempo. Com essa informação, é feito um percurso in-order a partir do node associado no índice pacoteTempo, armazenando os valores obtidos no decorrer do percurso. Assim, já são fornecidos todos os dados de eventos do pacote pesquisado. Eles são ordenados por tempo de ocorrência e id do pacote e, por fim, têm seus dados impressos utilizando o método geraRespostas.

A carga de trabalho recebida pelo sistema consiste em um arquivo de entrada contendo as informações sobre os eventos que devem estar presentes e sobre as consultas que devem ser realizadas

sobre eles. Nesse sentido, há 3 tipos de linhas possíveis: linhas “EV”, que fornecem informações sobre um evento específico, linhas “CL”, que indicam consultas sobre clientes, e linhas “PC”, que indicam consultas sobre histórico de um pacote específico. O tamanho da entrada pode variar e o sistema foi projetado com o fim de possibilitar que ele opere para valores consideráveis, tanto em termos de quantidade de linhas na entrada quanto em termos de número de pacotes tratados pelo sistema. Além disso, o sistema lida com os vários formatos possíveis para as linhas da entrada, processando e tratando os dados obtidos em cada uma delas.

2.1. Instruções para compilação e execução

Antes de tudo, é necessário que já esteja instalado e disponível um compilador para linguagem C++. Recomenda-se o compilador g++ da GNU Compiler Collection, especificamente na versão c++11, que foi utilizado em todo o desenvolvimento do projeto.

1. Abra o terminal e vá até o diretório raiz do projeto, com cd + <nome do diretório>
2. Digite o comando ‘make’ ou ‘make all’, que compila os arquivos e gera o executável ‘tp1.out’, armazenado na pasta bin
3. Após a compilação, para executar o programa, digite ‘./bin/tp1.out <nome do arquivo de entrada>’. Caso o arquivo esteja armazenado em um diretório específico, inclua o caminho para ele antes do nome, como ‘diretorio/<nome do arquivo>’.
4. Para limpar os arquivos armazenados nas pastas bin e obj, use o comando ‘make clean’.
5. **Exemplo de entrada para execução:** ./bin/tp3.out input1.txt

3. Análise de Complexidade

Será realizada uma análise organizada pelas classes implementadas. Além disso, apresenta-se a complexidade de métodos “generalistas” do sistema, como getters e setters, bem como uma análise geral do sistema.

1. Métodos getters e setters: métodos desse tipo estão presentes em praticamente todas as classes, com o fim de permitir o acesso a dados armazenados por elas ou de alterar esses dados. Como apenas realizam operações de atribuição (para setters) ou de return (para getters), a complexidade de tempo é $O(1)$. Além disso, como também operam somente sobre atributos já existentes e não realizam alocação de memória, a complexidade de tempo também é $O(1)$.

2. Construtores e Destrutores: também estão presentes na grande maioria das classes. Assim como os getters e setters, executam somente operações de custo constante (como atribuições, alocação e desalocação de memória) sobre objetos já existentes. Logo, as complexidades tanto de espaço quanto de tempo são $O(1)$.

3. Classe Pacote: além dos métodos “gerais”, a classe conta com com o método encontraPacote, que percorre o vetor principal de pacotes da main em busca de um pacote com um id específico. Se o número de pacotes total registrado no sistema é p, a complexidade de tempo é $O(p)$, uma vez que, no pior caso, temos de percorrer todo o vetor para encontrar o elemento desejado. Como o método recebe o vetor de pacotes por referência e não realiza alocação de memória, a complexidade de espaço é $O(1)$.

4. Classe Evento: o método geraResposta realiza somente uma quantidade constante de operações de comparação e de impressão de dados, apresentando complexidade de tempo $O(1)$. Como recebe um ponteiro para Evento como objeto para realização de impressões, não há alocação de memória/realização de cópias e, portanto, a complexidade de espaço também é $O(1)$. Similarmente, o método defineTipo realiza somente quantidades constantes de comparações e atribuição de valores, além de operar sobre uma string previamente definida na main, de forma que as complexidades de tempo e de espaço são ambas $O(1)$. O método ordenaPorTempoId é uma adaptação do algoritmo de insertion sort para os fins do sistema.

Sabemos que os vetores que são recebidos por esse método são, em geral, quase ordenados (já que são obtidos pelo percurso in-order em árvore balanceadas), e a principal função do método é tratar empates em casos de eventos com mesmo valor de tempo, de forma que a complexidade de tempo pode ser considerada $O(n)$, em que n é o número total de eventos registrados no sistema (no pior caso, temos de ordenar um vetor contendo todos os eventos do sistema, o que pode ocorrer se todos os eventos na entrada são associados a um mesmo pacote). Como o método opera diretamente sobre o vetor de eventos a ser ordenado, sem utilizar memória extra, temos complexidade de espaço $O(1)$.

5. Classe arvoreAVL: como especificado, a classe arvoreAVL é um template que implementa uma árvore balanceada AVL. Dessa forma, ela pode tratar de nodes com diferentes tipos para os campos de chave e de valor, sendo empregada nos 3 índices desenvolvidos. Os métodos max_val, altura, updateAltura, getBF, rotacaoDir, rotacaoEsq e vazia, como realizam apenas operações de custo constante (como atribuições, operações aritméticas, manipulação de ponteiros ou returns) e não realizam alocações de memória (operam sempre sobre estruturas já existentes), têm complexidade tanto de tempo quanto de espaço de $O(1)$. Pelas estratégias utilizadas na sua construção, sabemos que a árvore AVL é balanceada e sempre possui altura da ordem de, no máximo, logk, sendo k a quantidade de nós presente na árvore. Assim, os métodos insereRec, insere, buscaRec, busca, buscaNode e existe, como têm todos de percorrer a árvore, possuem complexidade de $O(logk)$. Além disso, têm complexidade de espaço de $O(logk)$ devido à pilha de recursão gerada ao longo do percurso pela árvore. Já os métodos que percorrem a árvore in-order, passando por cada de um de seus nós ou pelos nós a partir de uma certa posição na árvore (métodos buscaInOrder, que realiza uma travessia in-order pela árvore; coletaAPartirDe, que pecorre a árvore a partir de um nó com características específicas; inOrderTraversal, que é o método público para realizar o percurso in-order, utilizando o método privado buscaInOrder; e coletaEventosDoPacoteAteTempo), possuem complexidade de tempo $O(k)$. O uso de uma instância da árvore AVL gera uma complexidade de espaço geral de $O(k)$, uma vez que temos k nodes que são criados e armazenados nela. Para as 3 árvores utilizadas no sistema, temos p como o número de pacotes, n como o número de eventos registrados no sistema e c como o número de clientes. Para o índice de pacotes, temos complexidade de espaço $O(p)$, $O(n)$ para o índice pacoteTempo e $O(c + p)$ para o índice de clientes (há c clientes e, no pior caso, um cliente tem p pacotes associados em sua árvore aninhada). A mesma lógica pode ser utilizada para definir a complexidade de tempo em termos de cada um dos índices que utilizam a árvore.

6. Classe indiceCli: o método addPacote utiliza os métodos existe e insere da arvoreAVL, além de métodos setters para definição de valores de atributos, o que gera uma complexidade de tempo de $O(logp)$, já que, no pior caso, a árvore aninhada de um cliente terá todos os pacotes existentes no sistema. Similarmente, o método mudaPacote utiliza o método busca da arvoreAVL e métodos setters, gerando uma complexidade de tempo também de $O(logp)$. Ambos alocam um único objeto infoPacotes, com complexidade de espaço $O(1)$. No método atualizaFim, primeiro percorremos a árvore/índice de clientes em busca de um cliente com um nome específico, gerando complexidade de espaço de $O(logc)$, e depois percorremos a árvore aninhada desse cliente (caso ele seja, de fato, encontrado na árvore) para encontrar e atualizar as informações do pacote, com complexidade de tempo de $O(logp)$. Assim, tem-se uma complexidade de tempo total de $O(logc + logp)$. Não há alocação de estruturas, o que gera complexidade de espaço de $O(1)$. O destrutor da classe, como percorre as árvores aninhadas, têm complexidade de tempo de $O(p)$, uma vez que passa por todos os nós de uma árvore aninhada que pode ter, no máximo, p pacotes. Além disso, utiliza um vetor de mesmo tamanho como memória auxiliar, com complexidade de espaço $O(p)$.

7. Classe indicePacs: os métodos da classe basicamente chamam métodos de arvoreAVL para operar sobre o atributo desse tipo que é parte da classe. Assim, os métodos registraPacote (que chama insere de arvoreAVL), existePac (que chama existe de arvoreAVL) e tempoRegistro

(que chama busca de arvoreAVL), possuem complexidade de tempo $O(logp)$, em que p é o número de pacotes presentes no sistema. A complexidade de espaço geral da classe é de $O(p)$, uma vez que, ao longo da execução, sempre temos até p nodes na árvore da classe. O método geraChaveParaBuscaTempo utiliza o método buscaNode de arvoreAVL e realiza outras operações apenas para a construção de strings, gerando complexidade de tempo de $O(logp)$ e de espaço de $O(1)$.

8. Classe indicePacTempo: o método geraChave apenas manipula strings, com complexidade tanto de tempo quanto de espaço de $O(1)$. Similarmente à classe indicePacs, os métodos da classe basicamente chamam métodos de arvoreAVL para operar sobre o atributo desse tipo que é parte da classe. Assim, os métodos registraEvento (que chame insere de arvoreAVL e geraChave), buscaEvento (que chama busca de arvoreAVL) e eventosDoPacote (que chama coletaEventosDoPacote de arvoreAVL) possuem complexidade $O(n)$, em que n é o número de eventos registrado no sistema. A complexidade de espaço geral para o uso da classe, devido à presença de uma árvore AVL como atributo, é de $O(n)$.

9. Complexidade geral do sistema: temos n como o número de eventos no sistema, p como o número de pacotes, c como o número de clientes e l como o número de linhas no arquivo de entrada. O índice de clientes têm complexidade de espaço $O(c + p)$, o de pacotes $O(p)$ e o de pacoteTempo $O(n)$. Além disso, é instanciado na main um vetor de pacotes, um vetor de eventos e um vetor de linhas, com complexidades de espaço de $O(p)$, $O(n)$ e $O(l)$, respectivamente. Assim, a complexidade de espaço total do sistema é $O(c + p + n + l)$. Para a complexidade de tempo, temos: $O(l)$ para a leitura e processamento de cada uma das linhas do arquivo de entrada; para cada linha de evento, complexidade de tempo de $O(logc + logp + logn)$, uma vez que, no pior caso (evento de tipo registro), temos de fazer inserções em cada um dos 3 índices do sistema, gerando complexidade total de $O(nlogc + nlogp + nlogn)$ para processamento de n eventos; para consultas de clientes, complexidade de $O(logc + logp)$ para processamento das consultas, $O(p)$ para ordenação dos resultados obtidos (já que utilizamos um insertion sort adaptado, o vetor pode ser considerado quase ordenado e um cliente pode ter, no pior caso, cada um dos p pacotes associado) e $O(p)$ para impressão das respostas da consulta, gerando uma complecidade total de $O(logc + logp + p)$; para consultas de pacotes, complexidade de $O(logp + logn + n)$ para processamento das consultas (percorrer ambos os índices em busca dos valores desejados e, posteriormente, fazer a busca in-order pela árvore de pacoteTempo), complexidade $O(n)$ para impressão das respostas (já que, no pior caso, um pacote pode ter todos os eventos associados a ele), gerando uma complexidade total de $O(logp + logn + n)$. Com isso, a complexidade de tempo total do sistema pode ser dada por $O(nlogc + nlogp + nlogn + logp + logc + logn + n + p + l)$, que pode ser simplificada para $O(nlogc + nlogp + nlogn + l)$, já que os valores de logp, logc, logn são necessariamente menores que nlogc, nlogp, nlogn, e os valores de n e p sempre são necessariamente menores ou iguais ao valor de l. A complexidade pode ser, por fim, simplificada para $O(max(nlogc, nlogp, nlogn, l))$.

4. Estratégias de Robustez

O principal mecanismo utilizado, em termos de estratégias de robustez, foi o tratamento de exceções com o mecanismo de try-throw-catch disponibilizado pela linguagem C++. Em especial, foi empregado em main.cpp para indicar se ocorreram erros ao ler dados do arquivo de entrada ou mesmo ao abri-lo para a leitura. Assim, torna-se possível identificar possíveis erros e evitar a sua propagação ao longo do sistema, que poderia gerar comportamentos inesperados ou errôneos. Além disso, o tratamento de exceções também foi empregado em métodos que poderiam gerar acesso indevido a posições de memória, em especial nos métodos de busca em vetores ou busca em árvores, para indicar que um elemento

de interesse não foi encontrado. Assim, são evitados problemas como acessos indevidos caso os métodos retornassem um ponteiro nulo ao invés de lançar a exceção.

Por fim, também foi feita a modularização das classes, mantendo os seus atributos privados e, portanto, inacessíveis a alterações diretas e possivelmente maliciosas por parte dos usuários.

5. Análise Experimental

5.1 Método e Resultados

Para realização da análise experimental, primeiro foram fixadas algumas configurações padrões do sistema, e depois variados os parâmetros de interesse um a um. Os fatores principais de cada uma das análises são registrados nas tabelas, e as demais configurações padrões, quando omitidas, são: custo de transporte = 20; capacidade de transporte = 2; intervalo entre transportes = 100; custo de retirada = 1. Como diferentes execuções geram diferentes tempos, foi utilizada a média de 5 execuções do sistema para o tempo de execução. Além disso, os tempos de leitura e de ações do sistema em si (tempo leitura e tempo sistema, respectivamente) foram medidos e registrados separadamente As entradas para realização da análise foram obtidas utilizando o código para geração de carga de trabalho fornecido pelos professores. Consideramos, para os gráficos, que o tempo de execução corresponde ao tempo gasto com as ações do sistema em si, tomadas após a leitura de todas as linhas da entrada e seu armazenamento em memória auxiliar. Para cada um dos gráficos, temos o tempo (de leitura ou de execução/sistema) como o eixo y e o outro fator analisado como o eixo x.

O primeiro fator analisado foi o número de clientes no sistema. Na Tabela 1, estão representados os dados obtidos com a variação desse parâmetro. Para facilitar a sua visualização e interpretação, também incluímos os Gráficos 1 e 2.

nro. clientes	nro. pacotes	nro. armazéns	tempo sistema	tempo leitura
50	100	20	0,00920867	0,0058533268
100	100	20	0,011237138	0,0101087916
150	100	20	0,012018892	0,0037400602
200	100	20	0,012737238	0,0043988832
250	100	20	0,013582	0,0146647214
300	100	20	0,014930408	0,00872202
350	100	20	0,015575514	0,011125385
400	100	20	0,01597548	0,0054397752

Tabela 1. Resultados da variação do nro. de clientes.

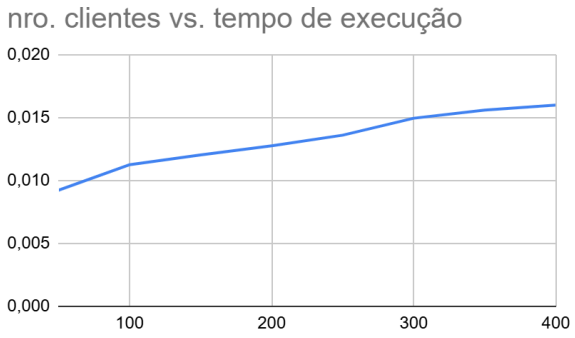


Gráfico 1. nro. de pacotes e tempo sistema.

Gráfico 2. nro. de pacotes e tempo de leitura.

Em seguida, foi observado o comportamento do sistema em relação ao número de pacotes nele registrados. Os resultados obtidos são mostrados na Tabela 2, que acompanha os gráficos 3 e 4, representativos da relação desse fator com os tempos de execução do sistema e leitura do arquivo de entrada.

nro. pacotes	nro. clientes	nro. armazéns	tempo sistema	tempo leitura
50	50	20	0,005954172	0,012812085
100	50	20	0,008622382	0,0157132336
150	50	20	0,033521082	0,0045189738
200	50	20	0,03663822	0,0076222436
250	50	20	0,04614728	0,0216686782
300	50	20	0,05959492	0,0153568264
350	50	20	0,06372416	0,011046535
400	50	20	0,06695172	0,0168026036
450	50	20	0,0672415	0,00533593
500	50	20	0,0703715	0,01243948

Tabela 2. Resultados obtidos com a variação do número de pacotes no sistema.

Gráfico 3. nro. pacotes e tempo sistema.

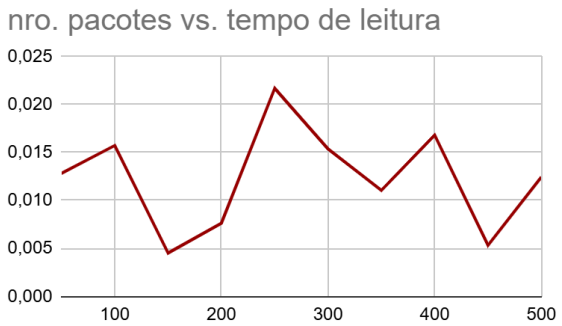
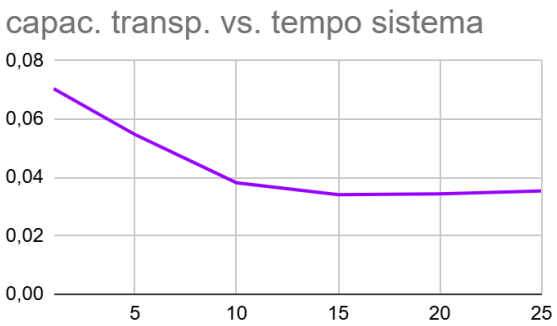


Gráfico 4. nro. pacotes e tempo de leitura.

O último parâmetro analisado foi o número de eventos, cujas variações foram geradas alterando o valor do parâmetro capacidade de transporte. Nota-se que, quanto menor esse valor, mais eventos são gerados, uma vez que são necessários mais eventos de rearmazenamento e, posteriormente de transporte entre armazéns. Os resultados obtidos são apresentados na Tabela 3 e nos Gráficos 4 e 5.

capac. transp.	nro. armazéns	nro. pacotes	nro. clientes	tempo sistema	tempo leitura	nro. eventos
1	10	500	50	0,07046784	0,0095262784	44975
5	10	500	50	0,05471314	0,0078968736	6315
10	10	500	50	0,03815814	0,006370538	4957
15	10	500	50	0,03407846	0,004071932	4957
20	10	500	50	0,03437456	0,0008485406	4957
25	10	500	50	0,03532544	0,000619722	4957

Tabela 3. Resultados obtidos com a variação do número de eventos no sistema.



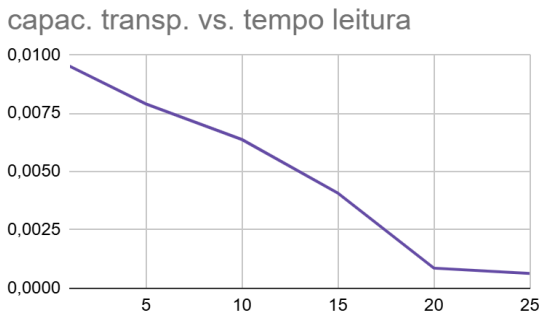


Gráfico 5. Eventos e tempo sistema.

Gráfico 6. Eventos e tempo de leitura.

5.2 Conclusões

Temos que o aumento no número de clientes gera um aumento no tempo de execução. Esse resultado é esperado uma vez que, quanto maior o número de clientes, maior é a árvore representativa do índice de clientes, o que torna operações como a atualização de ponteiros fim, buscas e inserções na árvore mais complexas e, portanto, mais demoradas. No entanto, nota-se que o impacto desse fator sobre o tempo de execução do sistema é mais discreto que o impacto do número de pacotes ou de eventos, por exemplo - o que também faz sentido, uma vez que esse fator impacta diretamente somente as operações que envolvem o índice de clientes em si.

Para o número de pacotes, também temos que, quanto maior é esse valor, maior é o tempo de execução do sistema. Esse resultado também faz sentido, uma vez que, com mais pacotes, são gerados no sistema mais eventos de todos os tipos. Além disso, o impacto desse fator no tempo de execução é mais pronunciado, o que também é esperado - um número maior de pacotes e, consequentemente, de eventos, faz com que seja necessário o processamento de uma quantidade maior de linhas, e a presença de mais pacotes impacta diretamente cada um dos 3 índices de pesquisa: precisamos atualizar mais vezes os ponteiros de fim no índice de clientes (gerando mais buscas na árvore de clientes e nas árvores aninhadas de pacotes), inserir mais nodes no índice de pacotes (já que esse índice é diretamente dependente dessa valor) e mais eventos associados a pacotes no índice pacoteTempo. Assim, as árvores representativas de cada um dos índices se tornam todas maiores e mais complexas à medida que esse valor cresce, o que aumenta os custos das operações sobre ela no geral e, consequentemente, os custos de pesquisas e inserções de novos itens.

Para a capacidade de transporte notamos que, quanto menor esse valor, maior é o tempo gasto pelo sistema. Como a redução desse valor acompanha aumentos significativos no número de eventos produzidos (já que, se há mais retenções, há muitos eventos de rearmazenamento e transporte entre armazéns), esse resultado também é esperado - se há mais eventos, os impactos são diretos sobre o índice de clientes (são necessárias mais atualizações de ponteiros fim para eventos de pacote) e sobre o índice de pacoteTempo (logicamente, temos de inserir mais eventos na árvore do índice), o que torna as operações de registro de eventos e, posteriormente, de consultas, mais complexas e mais caras, aumentando também o tempo de execução. Observa-se, no entanto, que, para esse fator, há um certo "ponto de saturação" a partir do qual o aumento da capacidade de transporte deixa de afetar o tempo de execução e o número de eventos gerado, o que pode ocorrer se esse valor se torna superior ao número de pacotes em cada uma das seções de armazenamento por vez. Pontua-se também que o valor 1 para esse parâmetro gera a quantidade máxima de eventos mantendo os demais parâmetros, já que sempre transporta-se um único pacote por vez e, portanto, são gerados eventos de rearmazenamento para cada um dos outros pacotes em uma mesma seção.

Um fato notável na análise é a falta de regularidade no tempo de leitura dos arquivos de entrada para os 2 primeiros fatores analisados. Ao executar o programa com cada um dos arquivos, foi possível notar que, no primeiro acesso a eles (e, em especial, quando não os tínhamos acessado recentemente), os tempos de leitura sempre foram consideravelmente maiores, o que pode ser explicado pela interação com a memória secundária e a necessidade de "resgatar" esses arquivos sempre que eles não estavam mais disponíveis em memórias de nível superior. Nota-se que a quantidade de linhas dos arquivos de entrada em cada um dos casos era praticamente a mesma, de forma que esse fator também pode explicar os "picos" obtidos nos tempos de leitura. Já para a leitura no caso do número de eventos, apesar de o comportamento nos primeiros acessos também ser observado, a quantidade de linhas de cada arquivo apresentava diferenças expressivas, o que pode explicar o comportamento mais regular do tempo de leitura, sendo aproximadamente proporcional à quantidade de linhas no arquivo.

6. Conclusões

Este trabalho lidou com o problema de implementar um mecanismo de consultas a eventos para o Sistema Logístico da empresa Armazéns Hanói. Os principais objetivos do sistema implementado foram processar eventos corretamente e realizar consultas sobre os eventos existentes de forma eficiente. Com a solução adotada, o sistema desenvolvido não só atende aos requisitos iniciais, como o faz de forma eficiente em termos de memória e tempo de execução.

Durante o desenvolvimento do trabalho, foi possível exercitar o conhecimento sobre diversas estruturas de dados, bem como métodos de pesquisa. A principal estrutura de dados empregada foi a árvore AVL, que foi utilizada com o fim de garantir que as pesquisas ocorressem em estruturas balanceadas, evitando casos degenerados que poderiam degradar os índices definidos e, com isso, comprometer a sua eficiência. Foi possível praticar a implementação desse tipo de estrutura e compreender melhor em quais cenários do mundo real ela poderia ser utilizada. Nesse sentido, também foi possível exercitar habilidades relacionadas à modelagem de estruturas por meio da implementação dos índices de pesquisa, que estão envolvidos com representantes de elementos reais do sistema logístico.

No decorrer do trabalho, os principais desafios enfrentados foram relacionados à modelagem dos índices, buscando formas de representação que transmitissem suas relações com elementos do sistema logístico real e, ao mesmo tempo, pudessem trazer contribuições reais para a eficiência da realização de consultas. Além disso,

7. Bibliografia

1. Cormen , T., Leiserson, C, Rivest R., Stein, C. Introduction to Algorithms, Third Edition, MIT Press, 2009.

2. Meira Jr, Wagner. e Lacerda, Anisio. (2025). Aulas e slides da disciplina Estruturas de Dados. DCC, UFMG.

3. Especificação do Trabalho Prático 3, Consultas Sobre o Sistema Logístico. DCC, UFMG.

4. Ziviani, N. (2006). Projetos de Algoritmos com Implementaçõeses em Java e C++. Editora Cengage.