Trabalho Prático 3 - Consultas ao Sistema Logístico

Nome: Miguel Bertussi Carneiro Moreira
Matrícula: [redacted]

Departamento de Ciência da Computação - Universidade Federal de Minas Gerais

(UFMG) Belo Horizonte - MG - Brazil

1. Introdução

A empresa vietnamita Armazéns Hanoi, após ter o seu novo simulador de sistema logístico e ter tido resultados positivos com ele, decidiu desenvolver um sistema de consultas ao simulador. Partindo desta ideia, eles decidiram por implementar dois tipos de consultas principais, histórico de um pacote e o histórico de pacotes por cliente.

Seguindo esta ideia, já podemos inferir que será necessário um sistema robusto, rápido e eficiente, e devemos visar armazenar o número de eventos seguindo esta ideologia. Pensando neste mesmo eixo, uma estrutura de dados como uma **árvore binária balanceada** pode vir a ser útil para garantir mais eficiência nas buscas. Quando for necessário utilizar de *arrays*, vamos optar por um **vector** dinâmico, que nos permite ter um controle melhor da memória.

A coordenação entre estas estruturas e algoritmos possibilitará a otimização do sistema de consultas, que terá suas funcionalidades ocorrendo ao mesmo tempo que a leitura dos dados.

2. Implementação

O código foi inteiramente projetado e implementado na linguagem C++, visando principalmente a modularização através das classes e o aspecto multi-uso do **template**. Nesta seção passaremos por cada uma das **classes** e **structs** implementadas a fim de resolver o problema proposto. (**Structs** e **classes** que utilizam de templates foram inteiramente implementadas no .hpp).

2.1 Vector

A classe **vector** é uma implementação própria de um vetor dinâmico genérico, que pode armazenar elementos de qualquer tipo **T**. Ela é semelhante a **std::vector**, permitindo inserção dinâmica de elementos, acesso por índice e cópia profunda entre vetores.

Atributos

- **T* data**: *array* simples do tipo **T**.
- **int _capacity**: capacity atual alocada do vetor.
- **int _size**: quantidade atual de elementos armazenados no vetor.

Construtores e Destrutor

- **vector()**: inicializa o **vector** com **_capacity** igual a um e **_size** zero.
- **vector(int capacity)**: inicializa o **vector** com a **capacity** desejada e **_size** zero.
- **vector(const vector& other)**: construtor de cópia, aloca nova memória e copia os elementos.

- **~vector()**: destrutor responsável por desalocar a memória utilizada pelo **vector**.

Métodos e Operadores Sobrecarregados

- **void push_back(const T& element)**: insere elemento no final do **vector**. Caso o **vector** esteja cheio, **expande** sua **_capacity** antes da inserção.
- **void clear()**: limpa o **vector**, que mantém a memória alocada mas perde os elementos.
- **int size() const**: retorna quanto elementos estão no **vector**, **_size**.
- **void expand()**: dobra a **_capacity** e copia os elementos para o novo **vector**.
- **T& operator[](int index)**: retorna uma referência ao elemento da posição **index**.
- **const T& operator[](int index) const**: versão constante que permite apenas leitura.
- **vector<T>& operator=(const vector<T>& outro)**: realiza cópia profunda dos elementos do **vector outro**, substituindo os dados do **vector** atual.

2.2.1 AVL Tree

A classe **tree** é a implementação de uma **árvore AVL** genérica e flexível a qualquer tipo de dados, permitindo o armazenamento de dados de forma mais inteligente, colocando elementos maiores à direita e menores à esquerda e se balanceando a cada inserção.

Atributos

- **node<T>* root**: raiz da árvore, um **node** de qualquer tipo.

Construtor e Destrutor

- **tree()**: construtor que apenas inicializa a raiz com valor **nullptr**.
- **~tree()**: destrutor padrão que irá chamar o método **destroyRecursive()**.

Métodos

- **void destroyRecursive(node<T>* _node)**: é o destrutor propriamente dito, irá percorrer os **nodes** da árvore recursivamente e desalocar a memória de cada um deles.
- **node<T>* insertRecursive(node<T>* _node**, **const T& _data)**: função recursiva que percorre a árvore, insere o novo elemento e executa o balanceamento local, retornando o novo **node** raiz da subárvore.
- **int height(node<T>* _node)**: retorna a altura de um **node**, zero se for **nullptr.**
- i**nt getBalance(node<T>* _node)**: retorna o fator de balanceamento do **node**.
- **node<T>* rotateRight(node<T>* _node)**: realiza rotação simples à direita, usada nos casos de desbalanceamento à esquerda.
- **node<T>* rotateLeft(node<T>* _node)**: realiza rotação simples à esquerda, usada nos casos de desbalanceamento à direita.
- **node<T>* find(const T& _key)**: é o método de busca na árvore, irá fazer a checagem procurando por **_key** de forma inteligente e, caso encontre, retorna o **node** desta **_key**.
- **void insert(const T& _data)**: faz a chamada do **insertRecursive()**.

2.2.2 Node

A **struct node** é a implementação genérica de um **node**, possuindo um apontador para o elemento à esquerda um para o elemento à direita, além de seu conteúdo.

Atributos

- **T data**: conteúdo que pode ser de qualquer tipo de dados.
- **node<T>* left**: apontador para o **node** que está à esquerda.
- **node<T>* right**: apontador para o **node** que está à direita.
- **Int height**: armazena a altura de um **node** na árvore.

Construtores

- **node()**: não inicializa **data**, inicializa os ponteiros com **nullptr** e a **height** com **um**.
- **node(const T& _data)**: inicializa **data,** os ponteiros com **nullptr** e **height** com **um**.

2.3.1 Package Index

A **struct packageIndex** define os índices de pacote que serão utilizados na busca e no armazenamento dos eventos, já que todo evento tem um pacote atrelado a ele.

Atributos

- **int packageID**: armazena o ID de uma pacote.
- **vector<int> eventIndexes**: armazena os índices dos eventos que estão em **allEvents**.

Construtores e Operadores Sobrecarregados

- **packageIndex()**: construtor padrão para permitir que uma **árvore** deste **index** seja criada.
- **packageIndex(int _packageID)**: construtor parametrizado que inicializa **packageID**.
- **bool operator <, ==, >(const packageIndex& other) const**: fazem comparações quanto ao **packageID** entre duas instâncias de **packageIndex**.

2.3.2 Client Index

Similar a **struct packageIndex**, a **clientIndex** é utilizada para criação de uma **árvore** de **clientes**, permitindo o acesso rápido aos eventos relacionados ao cliente.

Atributos

- **std::string name**: armazena o nome do cliente.
- **vector<int> relatedPackageIDs**: armazena o ID dos pacotes relacionados ao cliente.

Construtores e Operadores Sobrecarregados:

- **clientIndex()**: construtor padrão para permitir a criação de uma **árvore** de índices.
- **clientIndex(std::string _name)**: construtor parametrizado que inicializa **name**.
- **bool operator <.>,==(const clientIndex& other) const**: comparam o atributo **name** entre duas instâncias de **clientIndex**.

2.4 Algorithms

Arquivo com implementação de funções que venham a ser úteis para o algoritmo.

Funções

- **void merge(vector<T>& array**, **int left**, **int mid**, **int right)**: é a função responsável por fazer com que o ***Merge Sort*** funcione, estabelece quais são os os **subvetores**, comparando seus elementos e povoando o **vector** principal com estes ordenadamente.
- **void mergeSort(vector<T>& array**, **int left**, **int right)**: é a função responsável por chamar a **merge** e dividir o vetor na metade para fazer as chamadas recursivas para cada uma destas.

- **T max(T a**, **T b)**: retorna o maior dos elementos entre **a** e **b**.

2.5 Event

A classe **event** define o principal tipo de dados do problema, e ela é responsável por definir e construir estes dados, além de prover informações sobre eles. **Obs.:** a classe foi feita pensando em generalismo, então, nem todos os eventos utilizarão de todos os atributos.

Atributos

- **int dateTime**, **packageID**, **originWarehouse**, **destinationWarehouse**, **destinationSession**: define a data de chegada, o ID do pacote, o armazém de origem e de destino e a seção de destino.
- **std::string type**, **sender**, **receiver**: define o tipo de evento, o remetente e o destinatário.

Construtores

- **event()**: construtor básico para viabilizar a construção de estruturas de eventos.
- **event(int _dateTime**, **std::string _type**, **int _packageID**, **std::string _sender**, **std::string _receiver**, **int _originWarehouse**, **int _destinationWarehouse)**: construtor do evento do tipo **RG**, define os atributos de nome semelhante às variáveis recebidas.
- **event(int _dateTime**, **std::string _type**, **int _packageID**, **int _destinationORoriginWarehouse**, **int _destinationSessionORWarehouse)**: construtor dos eventos dos tipos **AR**, **RM**, **UR** e **TR** (idem ao construtor definido anteriormente).
- **event(int _dateTime**, **std::string _type**, **int _packageID**, **int _destinationWarehouse)**: construtor de eventos do tipo **EN** (idem).

Métodos e Operadores Sobrecarregados

- **void printEvent()**: identifica o tipo do evento e faz o *print* de suas informações.
- **std::string getSender()**: retorna o remetente (apenas para eventos do tipo **RG**).
- **std::getReceiver()**: retorna o destinatário (apenas para eventos do tipo **RG**).
- **int getPackageID()**: retornam os atributos que dão nome aos métodos.
- **bool operator**<,>,<=,>=**(const event& other)**: viabilizam a comparação entre eventos, primeiro testam o **dateTime**, depois o **packageID** (todos eventos possuem estes dois atributos).

2.6 Consultation

É outra classe cerne, definirá o tipo de dados **consultation**, ou seja, as consultas. Irá construí-las e prover informações importantes sobre elas.

Atributos

- **int dateTime**, **packageID**:define a data da consulta e o ID do pacote a ser consultado. **Obs.:** as consultas não necessariamente utilizarão os dois, o mesmo vale para as **strings**.
- **std::string**: **type**, **name**: define o tipo de consulta e o nome do cliente a ser consultado.

Construtores

- **consultation()**: construtor básico para viabilizar a construção de estruturas de consultas.
- **consultation(int _dateTime**, **std::string _type**, **std::string _name)**: construtor para consultas do tipo **CL**.
- **consultation(int _dateTime**, **std::string _type**, **int _packageID)**: construtor para consultas do tipo **PC**.

Métodos

- **void printConsultation()**: identifica o tipo de consulta e faz o *print* de suas informações.
- **std::string getName()**: método getter para eventos do tipo **CL** que retorna **name**.
- **int getPackageID()**: método getter para eventos do tipo **PC** que retorna **packageID**.

2.7 Engine

Classe principal do sistema, responsável por armazenar todas as informações coletadas e realizar as consultas. Tudo de maneira eficiente e rápida.

Atributos

- **tree<packageIndex> packagesTree**: árvore que irá armazenar os pacotes (seus IDs serão as chaves) e os índices dos eventos associados a cada um deles.
- **tree<clientIndex> clientsTree**: árvore que irá armazenar os clientes (seus nomes como chaves) e os índices dos pacotes.
- **vector<event> allEvents**: armazena todos os eventos em um local só.

Construtor e Destrutor

- **engine()**: construtor padrão, delega aos construtores das árvores e dos índices.
- ~**engine()**: destrutor padrão, assim como o construtor, delega aos outros destrutores.

Métodos

- **void processLine(const std::string& line)**: recebe uma linha lida na *main* e julga se é um evento ou uma consulta, mandando para as respectivas funções de processamento.
- **void processConsultation(std::stringstream& ss)**: processa consultas. Instancia um objeto consultation do tipo **CL** ou **PC** e envia para os métodos correspondentes.
- **void processEvent(std::stringstream& ss**, **int dateTime)**: processa eventos. Identifica o tipo do evento (**RG**, **EN**, **AR**, **RM**, **UR**, **TR**) e instancia o objeto **event** correspondente, enviando-o para o método de tratamento adequado.
- **void handleClientConsultation(consultation& clientConsultation)**: trata as consultas do tipo **CL** (clientes). Verifica se o cliente está cadastrado e imprime os **primeiros** e **últimos** eventos dos pacotes relacionados a ele.
- **void handlePackageConsultation(consultation& packageConsultation)**: trata as consultas do tipo PC (pacotes). Verifica se o pacote existe e imprime todos os eventos associados a ele, ordenados por data.
- **void handleRegistrationEvent(event& regEvent)**: trata eventos do tipo **RG**, que registram o remetente e o destinatário do pacote. Atualiza ambas as árvores (**clientsTree** e **packagesTree**) com os dados.

- **void handleOtherEvent(event& otrEvent)**: trata os eventos **EN**, **AR**, **RM**, **UR**, e **TR**. Apenas adiciona informações do pacote correspondente na **packagesTree**.

3. Análise de Complexidade

Nesta seção, iremos definir a complexidade de cada um dos algoritmos implementados da forma mais precisa possível, finalizando com a complexidade geral.

3.1 Vector

Aqui, temos métodos mais custosos, que alocam memória dinamicamente, passam elementos de um **vector** ao outro etc. A complexidade dos métodos: **vector()**, **vector(const vector& other)**, **expand()**, **operator=(const vector<T>& other)**, será sempre O(n) tanto de espaço, quanto de tempo, porque devem alocar '**n**' espaços na memória e atribuir '**n**' elementos. As demais operações são sempre O(1), tanto de tempo, quanto de espaço, pois apenas retornam valores já armazenados. Já o **push_back()**, tem o pior caso com O(n), tanto de complexidade como de espaço, porque neste caso ele deve chamar o **expand()**, mas como o crescimento da capacidade é exponencial, este custo é amortizado e é, em média, O(1), pois a memória já estará alocada e apenas insere-se o elemento ao fim do vetor, o que é custo constante.

3.2 AVL Tree & Node

Começando pelo **node**, como ele não manipula memória e apenas atribui valores aos atributos, a sua complexidade é O(1) espacial e temporalmente. Já a árvore **AVL**, ela tem complexidade espacial O(n), sendo **n** o número de **nodes**, já que todos eles devem ser alocados. A complexidade temporal para inserir e achar é O(log(n)), já que, como é balanceada, a altura da árvore se manterá proporcional a log(n), o que evita o caso degenerado de árvores binárias padrão, que levaria a O(n) temporalmente. Portanto, temos: O(n) espacialmente e O(log(n)) no quesito temporal.

3.3 Index

Os índices, assim como o **node**, tem apenas operações de atribuição, retorno e *print* de atributos, o que são operações constantes. Como não há memória extra alocada, é O(1) temporal e espacialmente.

3.4 Algorithms

A função de ordenação adicionada ***Merge Sort*** funciona dividindo o vetor em subvetores à direita e à esquerda do meio até que estes possam ser comparados em pequenas partições, ao fim deste processo, o **merge** é chamado para juntar essas partes e montá-las ordenadamente no **vector** principal. Por esta característica de ser dividido na metade até que sejam partições mínimas e por precisar chamar o **merge** para cada subdivisão, temos que a complexidade temporal é O(n.log(n)), já a espacial, é necessário alocar uma quantidade de memória proporcional ao **vector** original, portanto, é O(n). A função **max** apenas retorna o maior entre dois elementos, então o seu custo é constante espacial e temporalmente.

3.5 Event

Os eventos, também possuem apenas métodos de atribuição, retorno e *print* de atributos. Também, não alocam memória extra, logo o custo é O(1) tanto para tempo quanto para espaço.

3.6 Consultation

Assim como o **event**, possui apenas métodos de atribuição, retorno e *print* de atributos. Não aloca memória extra, então o custo é O(1) espacial e temporalmente.

3.7 Engine

Os métodos aqui são mais caros, já que sempre operam sobre as árvores e **vector** de eventos. O método **processLine()** tem custo constante em tempo e espaço, pois apenas interpreta a linha e repassa informações para, ou **processEvent()** que, em si, também tem custo constante (de tempo e de espaço) já que continua apenas interpretando a linha e gera um evento que é delegado a outro método, ou **processConsultation()**, que também apenas interpreta a linha e delega a consulta gerada a outro método (logo, o custo aqui também é constante).

Os métodos que podem ser chamados em **processEvent()**, **handleOtherEvent()** e **handleRegistrationEvent()**, são mais custosos. O segundo, mais custoso, é responsável por registrar e procurar informações na árvore de pacotes e de clientes, então, sendo **n** o número de pacotes e **m** o número de clientes, temos que, como as árvores são balanceadas, teremos complexidade temporal O(log(n) + log(m)), pois faremos a inserção (1) e a busca depois (2), então seria O(2(log(n) + log(m)) = O(log(n) + log(m)). Espacialmente, este método é O(1), já que ele insere apenas um **node** na árvore de pacotes e **no máximo** dois na árvore de clientes. Já o primeiro método, opera apenas na árvore de pacotes, então o custo temporal é O(log(n)), além de ser é O(1) espacialmente. Cumpre destacar que, ambos métodos inserem elementos no **vector** central de eventos, mas o custo é amortizado O(1), como explicado na seção **3.1**.

Agora, para os métodos que pode ser chamado em **processConsultation()**, temos **handleClientConsultation()** e **handlePackageConsultation()**. O primeiro, realiza uma busca na árvore de clientes e faz o *print* de todos os eventos relacionados aos pacotes que estão ligados ao cliente (por meio do **vector** de índices), além de armazenar os últimos eventos de cada pacote em **lastEvents**, fazendo um ***Merge Sort*** neste **vector**. Então, temos, O(log(m) + O(n) + O(n.log(n))) = O(log(m) + n.log(n)). Espacialmente, teremos O(n), já que vamos iterar e armazenar eventos de todos os pacotes relacionados ao cliente. Para o segundo método, vamos apenas achar o pacote na árvore e printar todos os eventos associados a ele, então, sendo **e** o número de eventos associados a um pacote, teremos temporalmente O(log(n) + e), enquanto espacialmente, o custo é constante, já que nada é alocado.

3.8 Complexidade Geral

Como podemos observar, o principal fator da complexidade reside na classe **engine**, mas ainda faltam considerações gerais sobre a quantidade de vezes que os métodos são chamados. Portanto, a complexidade temporal total e espacial, respectivamente, serão:

- $O(E_{RG} \cdot (\log(n) + \log(m)) + E_{OUTRO} \cdot \log(n) + C_{PC} \cdot (\log(n) + e) + C_{CL} \cdot (\log(m) + p_2.\log(p_2))$, onde E_{RG} é o número de eventos de registo, E_{OUTRO} o número de eventos que não são de registro, C_{PC} o número de consultas de pacote, C_{CL} o número de consultas de cliente e p_2 o número de pacotes associados a um determinado cliente.
- $O(n + m + E)$, onde E é o número de eventos.

4. Análise de Robustez

Todos os códigos que envolvem alocação de memória dinâmica foram reforçados com estruturas ***try-catch*** genéricas para lançar mensagens de erro durante a execução, mas não interrompê-la, porque no contexto deste código, a estabilidade é preferível, já que em um caso real, mesmo que o sistema fique lento no primeiro momento, o precisamos processar a consulta, então precisamos manter a execução mesmo que signifique vazar memória. O sistema implementado dos ***try-catch*** não irá tratar a exceção, apenas sinalizar onde ocorreu o erro e qual foi ele, por meio do ***fprintf(stderr, ...)***.

Os blocos ***try-catch*** também foram inseridos em métodos para sinalizar erros distintos de *leaks*, como algum acesso a uma posição inexistente ou a um **node** nulo, por exemplo. Quanto à robustez do **vector**, na alocação de memória sua memória, no caso de não possuir size inicial, é iniciado com **_capacity** igual a um, o mínimo para ter memória alocada, para evitar *leaks*. Para a árvore **AVL**, os construtores garantem a inicialização com nulos e valores padrões (o mesmo vale para os **nodes**). Após rodar o teste do *Valgrind* para achar *leaks*, usando os três testes iniciais como exemplo, nenhum foi achado.

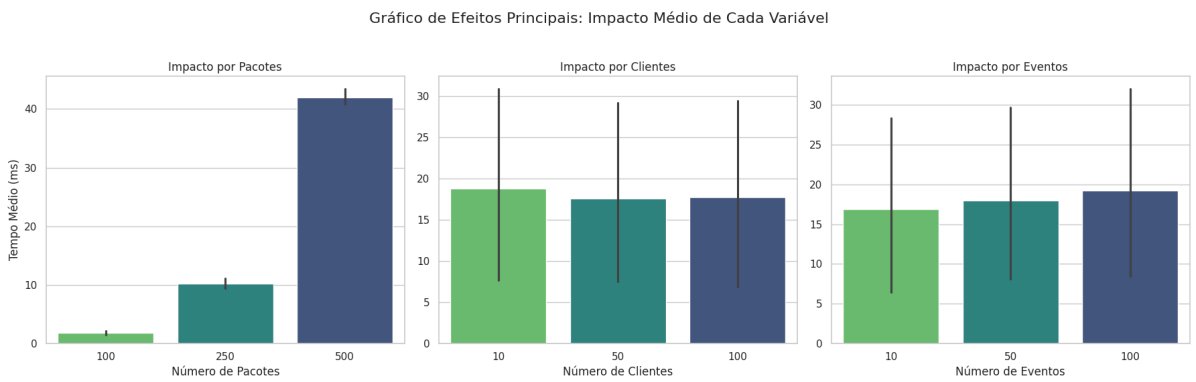
5. Análise Experimental

5.1 Elaboração

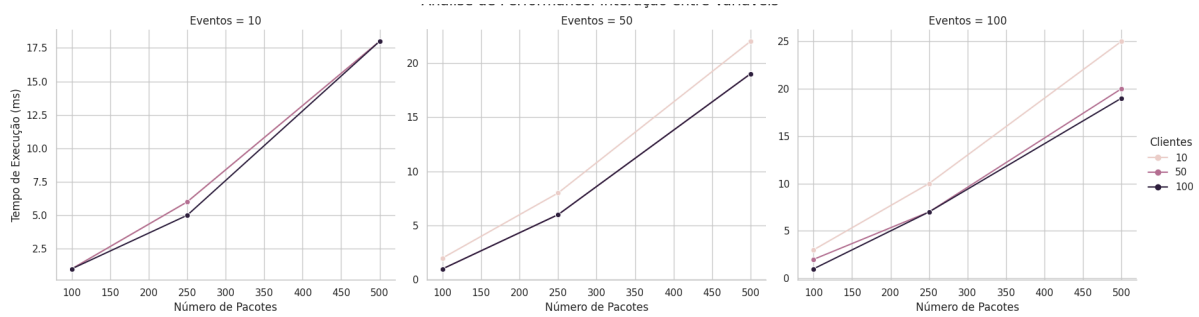
A fim de testar a robustez e escalabilidade do algoritmo, vamos variar três dimensões, o número de **pacotes** = {100, 250, 500}, **clientes** = {10, 50, 100} e **eventos** = {10, 50, 100}, e também analisar o comportamento tanto para a árvore AVL quanto para uma árvore binária tradicional não balanceada. O intuito será mostrar a escalabilidade e superioridade de uma árvore binária balanceada para a resolução de problemas maiores, além de visualizar o impacto destas variáveis no tempo de execução do algoritmo. **Obs.:** o algoritmo foi alterado para não processar os dados e consultas junto à leitura, como foi indicado no **pdf**, então a marcação do tempo está sendo feita após a leitura de todas as linhas do arquivo.

5.2 Resultados

Para o teste com a árvore binária não balanceada, temos:

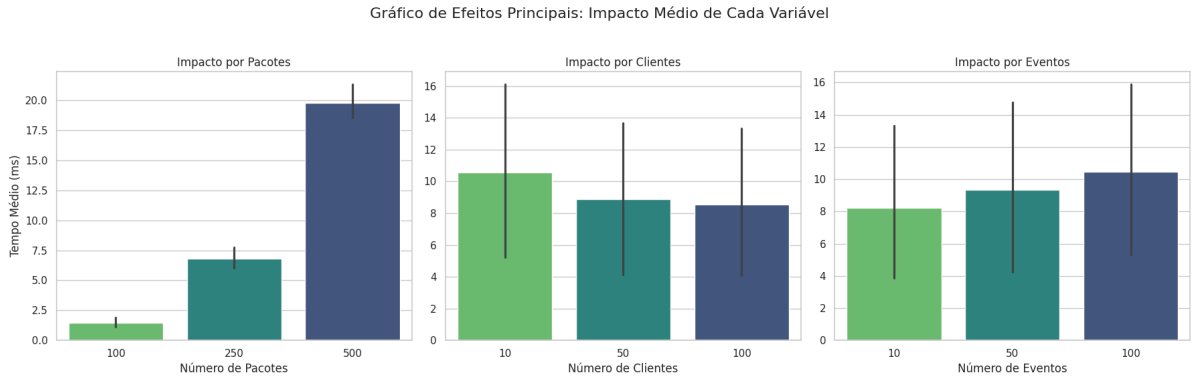


A partir destes gráficos, podemos observar que o tempo de execução deste algoritmo depende principalmente da quantidade de pacotes, o que é válido, já que a árvore principal do código opera sobre os pacotes.

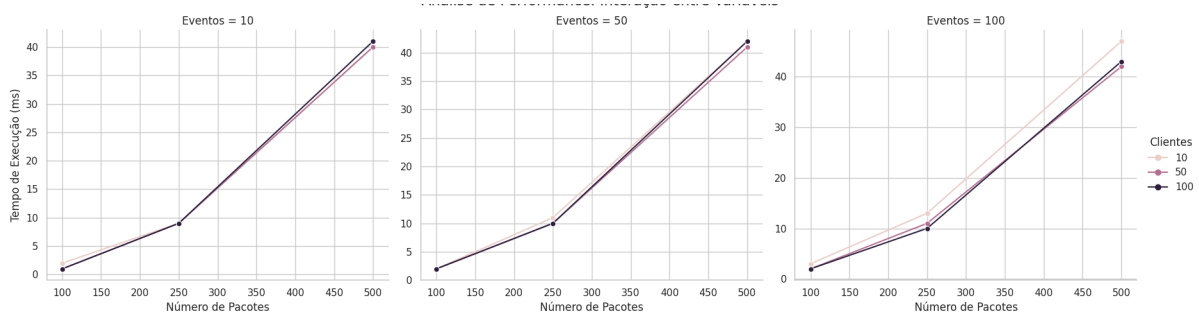


Estes outros gráficos mostram de forma ainda mais clara o impacto dos pacotes no tempo de execução do algoritmo, os grandes pulos no eixo do tempo ocorrem nas mudanças do número de pacotes, enquanto o número de clientes e eventos pouco alteram o tempo. Vemos que o tempo de execução é, em média, ~40 ms.

Agora, para a árvore AVL, temos:



Estes gráficos nos mostram, inicialmente, informações semelhantes, ao primeiro gráfico mostrado, o impacto principal no tempo de execução ocorre devido aos pacotes, pela mesma razão citada anteriormente.



Já estes gráficos, eles nos mostram a real diferença de usar uma árvore balanceada, podemos começar vendo, similar ao resultado passado, que os pacotes fazem a maior diferença, mas a real mudança agora está no eixo do tempo, podemos observar uma diminuição significativa no tempo de execução, que agora está perto de ~20 ms, ou seja, reduzimos pela metade o tempo que o algoritmo leva para processar as consultas.

5.3 Análise dos Resultados

Como foi observado pelos resultados, a implementação de uma árvore AVL ao invés de uma árvore binária não balanceada é fator **chave** para o tempo de execução do algoritmo. Isso ocorre, porque, com uma árvore balanceada como a AVL, evitamos completamente o caso degenerado citado na seção **3.2**, que ocorre em árvores binárias não balanceadas quando os elementos inseridos na árvore estão ordenados ou inversamente ordenados, o que resulta em todos os **nodes** em apenas um dos lados, tornando a árvore uma lista encadeada, que tem, para inserção e busca, complexidade O(n) de tempo.

Outro ponto que já foi citado anteriormente é de como os pacotes têm maior influência do que as outras variáveis, e isso se deve ao fato de que a estrutura do programa foi montada com os pacotes como elemento central, a árvore de pacotes é a responsável pelos eventos e a árvore de clientes depende da árvore de pacotes, então este resultado condiz com o esperado.

6. Conclusão

O problema tratado neste trabalho foi o de criação e otimização de um sistema de consultas para a empresa Armazéns Hanoi. A implementação de uma árvore binária balanceada como a AVL se mostrou extremamente eficiente para a resolução, evitando um pior caso linear e garantindo um custo logarítmico. Os resultados obtidos na análise mostraram a escalabilidade do sistema, o que evidenciou ainda mais a necessidade de estruturas de dados eficientes e índices para acelerar as buscas.

Em geral, o trabalho proveu uma solução eficiente ao problema proposto e grande entendimento sobre árvores balanceadas e sua importância na otimização das buscas e armazenamento e de dados.

7. Bibliografia

Lacerda, A. and Meira JR, W.(2024). Slides da disciplina de estruturas de dados, Aula 06 - Ordenação: MergeSort e Shellsort, Aula 11 - Árvores, Aula 19 - Árvores Balanceadas.
Disponibilizado via moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

8. Documentação Extra

8.1 Event

Somente o método **int getDateTime()** foi adicionado, que retorna a data-hora de um evento.

8.2 Consultation

Atributos

Foram adicionados os atributos **int start** e **int end**, que serão os atributos principais para a nova consulta a ser implementada.

Métodos

- **consultation(int _dateTime**, **std::string _type**, **int _start**, **int _end)**: construtor para novas consultas do tipo **IT** (intervalo de tempo).
- **void printConsultation()**: foi alterado para fazer o *print* de consultas do tipo **IT**.
- **int getStart()**, **getEnd()**: métodos getters para retornar **start** e **end**.

8.3 Engine

Métodos

- **void processConsultation(std::stringstream& ss**, **int dateTime**, **const std::string& consultationType)**: adaptado para leitura de consultas do tipo **IT**.
- **void handleTimeIntervalConsultation(consultation& timeIntervalConsultation)**: método responsável pelo processamento de uma consulta do tipo **IT**. É responsável por printar todos os eventos no determinado intervalo de tempo fornecido.
- i**nt LBbinarySearch(vector<event>& array**, **int target)**: método para fazer uma busca binária *lower bound*, ou seja, encontrar o menor índice de **target** no **array**.

9.3 Complexidades Extras

Os métodos adicionados a **Event** e a **Consultation** são apenas atribuições, *prints* e retornos de atributos, portanto, são O(1) tanto espacial quanto temporalmente. Já os métodos adicionados para a consulta **IT**, temos a adaptação do **processConsultation()**, que se mantém O(1) como mostrado na seção **3.7** e o **handleTimeIntervalConsultation()**, que tem complexidade temporal linear (O(n)), sendo **n** o número de eventos do **array**, e espacial constante (O(1)), pois ele apenas itera sobre os eventos do **vector allEvents**, atributo da **engine**. O método **LBbinarySearch()**, por ser uma busca binária, tem complexidade temporal O(log(n)) e espacial constante (O(1)), pois não aloca nova memória.

10. Considerações Finais

Esta nova consulta foi testada e será funcional apenas se o limite superior for menor ou igual ao último evento registrado. O *print* foi formatado da mesma maneira que as consultas padrão, o número de elementos a serem printados seguido destes elementos. Para utilizá-la, basta inserir uma linha no arquivo de entrada na forma **(int IT int int)**, em que ambos **int's** depois de **IT** existam como **dateTime** de algum evento, o algoritmo fará o *parsing* correto a partir disto.