ADLER GUILHERME FURTADO FARIA
2023028501

TP 3 ESTRUTURA DE DADOS

Busca de Passagens Aéreas

UFMG Belo Horizonte, 2024

INTRODUÇÃO

Este trabalho prático se baseia na implementação e otimização de um sistema de busca de passagens aéreas complexo que deve funcionar para encontrar quais as passagens que melhor se adequam às necessidades de quem quer comprar as passagens.
A princípio, o programa deve receber uma entrada que contém os principais dados dos voos, como: a origem, o destino, o preço da passagem, o número de paradas, assentos disponíveis, o horário de saída e o de chegada. Além disso, o programa também recebe as preferências do cliente, ou seja, ele pode filtrar esses voos a partir de seu gosto, para ver apenas os voos que atendem os seus requisitos, através de uma ordenação com base em trigramas e especificações de atributos.
A partir disso, será feita uma análise e manipulação dos dados em vista a fazer com que isso seja disponibilizado para o cliente, através do uso de árvores balanceadas de pesquisa para a ordenação do trabalho. Para isso utilizei várias árvores, aplicando os filtros em uma árvore binária de busca para atingir eficiência nos critérios de filtragem.

MÉTODO

O código possui 4 arquivos principais, devidamente modularizados: árvore, filtro, voo e main, sendo cada um deles formado por structs e funções que interagem entre si. Assim, para a explicação do funcionamento do código irei explicar, primeiramente, cada arquivo de inclusão e seguir explicando através do fluxo da main.

A árvore é um arquivo base que serve para implementar uma árvore balanceada de buscas, através da struct Nodo (Nó) e funções de inserção, ordenação e remoção, que trabalham bastante em conjunto com o arquivo filtro.

O filtro, então, guarda todas as informações acerca das linhas de especificação dos filtros lidos na entrada. Ele foi implementado de dividido em duas structs: Filtros e o Operador; essa foi uma decisão de implementação minha, pois nas entradas lidas possuímos uma linha que vai especificar todas as restrições que um voo deve ter para ser exibido, então implementei a struct Filtros para guardar as informações dessa linha, como o número de resultados e qual o trigrama relacionado à pesquisa, e, como uma linha pode possuir mais de um atributo de restrição, decidi implementar o Operador como um atributo de de Filtros, o qual irá guardar todos os dados relacionados às restrições, como os operadores “==”, “<=”, etc. e os números de comparação. Para isso ele irá armazenar todos os dados dentro de de um objeto filtros primeiro e depois a partir de *char condicao_completa[100]* e das funções de verificação (a qual irá analisar todos os filtros e armazenar em seus devidos lugares), serão guardados os dados dentro dos objetos alocados de Operador.

Assim, também temos o arquivo voo que vai ser responsável por guardar todas as informações do voo a serem comparadas para atender os filtros e restrições feitas.

Inicialmente, minha main começa lendo o arquivo e salvando os dados contidos neles, salvando primeiramente o int contido logo no início do arquivo, através da função

calcula_voos, a qual irá interpretar esse número inteiro como o total de voos contidos no arquivo, a partir disso é inicializado uma struct tipo Voo junto com a função *aloca_voos*, a qual irá ler os dados de todos os voos do arquivo e armazená-los em um vetor de struct Voo.

Após isso, para finalizar a etapa de leitura do arquivo, é lido as últimas linhas, as quais contêm dados que são relacionados aos filtros e restrições que são armazenados em uma struct do tipo Filtros.

Após isso são inicializados os nós iniciais para todas as 8 árvores do código, uma para cada índice, onde são chamadas as funções de inserção em cada uma delas, formando árvores binárias.

Logo após é inicializado um nó raiz, que vai servir início para uma árvore filtrada, processando os voos e aplicando uma série de filtros para exibir resultados de acordo com as especificações do usuário. Cada filtro é aplicado em uma árvore binária, e os resultados são impressos de forma ordenada, respeitando o número máximo de resultados.

Para isso são utilizadas as funções de inserir, a qual insere um voo na árvore de forma ordenada, garantindo que a árvore se mantenha balanceada e ordenando, de acordo com o critério especificado pelo trigrama (por exemplo, preço, número de paradas e duração). Ela compara o voo atual com o nó da árvore e decide se o voo deve ser inserido na subárvore esquerda ou direita, dependendo do resultado da comparação. Já a função *em_ordem* realiza uma travessia em ordem na árvore, imprimindo os voos de acordo com o número de resultados solicitado pelo filtro. Ela percorre a árvore da esquerda para a direita, imprimindo os voos até o limite de resultados definidos e utilizando a função *mostrar_voo* para exibir as informações do voo. Ambos os métodos utilizam recursão para navegar pela árvore e garantir a correta ordenação e exibição dos voos filtrados. Assim, o código é finalizado com a devida desalocação da memória utilizada durante o processo, garantindo a liberação eficiente dos recursos.

ANÁLISE DE COMPLEXIDADE

As funções *calcula_voos*, *calcula_filtros* são exemplos de funções no meu código que realizam tarefas simples, as quais não dependem de alocação de memória e possuem complexidade temporal e espacial de O(1).

As funções *aloca_voos* e *le_filtros* são exemplos de funções que leem os dados e realizam ações alocando um vetor de tamanho n. Portanto possuem complexidade temporal e espacial de O(n) e O(m), sendo n o número de voos e m o número de filtros.

Já a função *inserir* insere um voo em uma árvore binária de acordo com o tipo de filtro, e como é uma árvore binária balanceada a complexidade temporal é de O(log n), onde n é o número de elementos na árvore e a complexidade espacial de O(n), onde n é o número de voos inseridos na árvore.

Em contrapartida as funções de *desalocar_arvore* possui complexidade temporal de

O(n) pois percorre toda a árvore pra desalocar o nó mas sem alocar memória o que faz com que a complexidade de espaço seja de O(1).
Agora, olhando para o código como geral, ele utiliza O(n log n) para a inserção e organização dos voos e O(n + m) para a alocação de memória. A principal operação que afeta a complexidade de tempo é a inserção dos voos nas árvores binárias, que é bem gerenciada com um O(log n) por operação devido ao uso de árvores balanceadas.

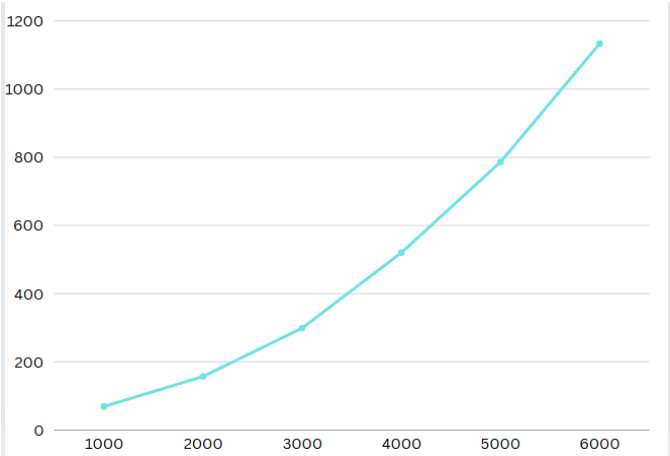
ESTRATÉGIAS DE ROBUSTEZ

Compilei e utilizei o Valgrind durante a produção do trabalho, ele me ajudou a apontar os erros em diversos momentos o que facilitou muito o desenvolvimento.
Durante a produção também, utilizei de muitos prints em cada etapa do processo, usando para visualizar se as saídas do código estão saindo como esperadas, como por exemplo, eu imprimia na tela uma lista dos voos após alocar, após adicionar na árvore, etc. Isso foi bastante presente durante todo o processo, porém se manteve no código final para não atrapalhar no resultado dos vpls.
Agora em relação a estratégias de defesa, diversas foram implementadas para garantir que o sistema opere de maneira confiável e eficaz, como o uso de verificações de erros durante a leitura de arquivos. Funções como *calcula_voos* e *calcula_filtros* incluem verificações de retorno, garantindo que a leitura do número de voos e filtros seja realizada corretamente. Caso contrário, mensagens de erro são exibidas, e a execução é interrompida, evitando que o sistema opere com dados corrompidos ou incompletos. Além disso, o código também cuida da alocação de memória, utilizando a função malloc para alocar dinamicamente os vetores de voos e filtros. Se a alocação falhar, o código exibe uma mensagem de erro e interrompe a execução, prevenindo o uso de memória inválida. Além disso, todas as alocações de memória para voos e filtros são seguidas por liberações explícitas no final do programa, com o uso das funções de desalocação, o que ajuda a evitar vazamentos de memória. Finalmente, a robustez no tratamento de árvores binárias é garantida através de uma inserção cuidadosa de nós nas árvores, assegurando que a estrutura de dados seja mantida balanceada e eficiente, e ao limpar as árvores após a execução para liberar recursos corretamente. Essas estratégias em conjunto formam uma base sólida para que o código seja capaz de lidar com diferentes cenários e falhas de maneira controlada e eficiente.

ANÁLISE EXPERIMENTAL

Para a análise experimental utilizei um arquivo de entrada input.txt e a biblioteca chrono.
Fiz minha análise através dos dados retirados da análise de complexidade, analisando principalmente a influência do número de voos e número de filtros no código, visto que são as partes mais custosas e que mais interagem com o resto do código, monitorando o tempo desde o início do código até o seu fim.
Minha primeira métrica de análise foi o número de voos, para isso mantive fixo o número de filtros em 2 e aumentei a quantidade de voos de forma linear.
Esse foi o seguinte gráfico gerado:

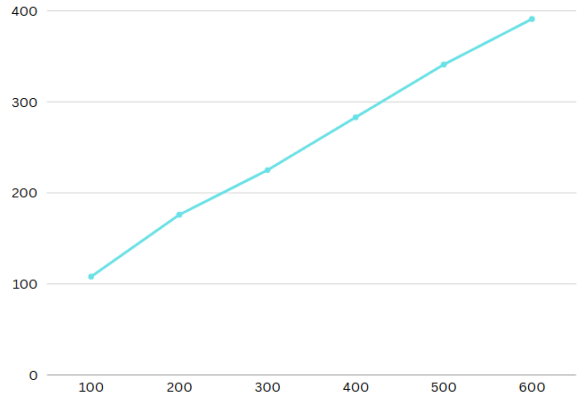
Número de voos x tempo (ms)
1000 = 76
2000 = 157
3000 = 299
4000 = 520
5000 = 786
6000 = 1133



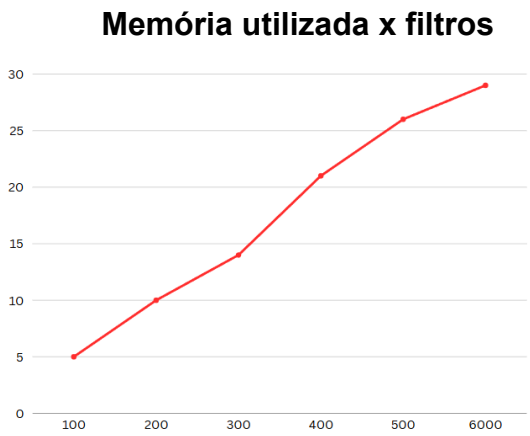
Nesse gráfico o eixo Y corresponde ao tempo de execução em milissegundos e o eixo x ao número de voos, pode ver que ao aumentar linearmente o número de voos, o eixo do tempo aumenta mais rapidamente, reforçando a conclusão retirada da análise de complexidade a qual O(n log n) foi associada à inserção de voos em uma árvore binária balanceada.

Já uma segunda métrica analisada foi o número de filtros, para isso mantive fixo o número de voos em 1000 e aumentei a quantidade de filtros gradualmente de forma linear.
Pode-se perceber que em, relação ao número de voos, os números de filtros tiveram um crescimento mais linear:

Número de filtros x tempo (ms)
100 = 108
200 = 176
300 = 225
400 = 283
500 = 341
600 = 391



Essa tendência à linearidade também faz sentido, visto que a aplicação de filtros está ocorrendo em uma fase em que a complexidade não é tão alta quanto a da inserção de voos nas árvores. Isso reflete a complexidade O(m) para leitura e aplicação dos filtros, onde m é o número de filtros, o que leva a um aumento proporcionalmente menor no tempo de execução.
A diferença entre o comportamento não linear para o número de voos e o linear para o número de filtros reforça a ideia de que a inserção em árvores balanceadas é a operação mais custosa em termos de tempo de execução do meu código.
Já em questão de memória alocada, ela aumenta de forma proporcional ao número de voos, uma vez que cada voo inserido nas árvores binárias exige alocação de memória para os nós, crescendo linearmente para tanto os casos de variação de voo quanto os casos de variação de filtros.
Pode-se ver nos gráficos a seguir (à esquerda memória por voos, e à direita memória por filtros) que a relação de memória utilizada por número de voos foi bem parecida com a relação de memória utilizada por número de filtros, reforçando o que foi supracitado e o que foi analisado na seção de análise de complexidade.



Para completar a análise, também utilizei as ferramentos do valgrind e kcachegrind que me forneceram os seguintes dados para a análise de uso da CPU:

Incl.	Self	Distance	Calling	Callee
99.86	0.00	2	1	(below main) (libc-2.31.so: libc-start.c)
99.86	0.00	1	1	_start (tp3.out)
99.85	1.15	3	1	main (tp3.out: main.cpp)
89.86	1.37	4	609 000	inserir(Nodo*&, Voo const&, char const*) (tp3.out: arvore.cpp)
86.45	22.08	5	608 391	inserir(Nodo*&, Voo const&, char const*)'2 (tp3.out: arvore.cpp)
63.14	63.14	5-6 (6)	9 970 278	comparar_voos(Voo const&, Voo const&, char const*) (tp3.out: arvore.cpp)
5.45	0.00	4	609	desalocar_arvore(Nodo*) (tp3.out: arvore.cpp)
5.45	1.27	5	1 218	desalocar_arvore(Nodo*)'2 (tp3.out: arvore.cpp)
4.18	0.07	6	609 000	0x0000000000109260
4.10	0.07	7	608 390	operator delete(void*, unsigned long) (libstdc++.so.6.0.28)
4.03	0.07	8	608 390	0x00000000048df100

Incl.	Self	Distance	Calling	Callee
89.99	1.37	1	609 000	inserir(Nodo*&, Voo const&, char const*) (tp3.out: arvore.cpp)
86.58	22.11	2	608 391	inserir(Nodo*&, Voo const&, char const*)'2 (tp3.out: arvore.cpp)
63.23	63.23	2-3 (3)	9 970 278	comparar_voos(Voo const&, Voo const&, char const*) (tp3.out: arvore.cpp)
5.46	0.00	1	609	desalocar_arvore(Nodo*) (tp3.out: arvore.cpp)
5.45	1.27	2	1 218	desalocar_arvore(Nodo*)'2 (tp3.out: arvore.cpp)
4.18	0.07	3	609 000	0x0000000000109260
4.11	0.07	4	608 390	operator delete(void*, unsigned long) (libstdc++.so.6.0.28)
4.03	0.07	5	608 390	0x00000000048df100
3.96	0.07	6	607 248	operator delete(void*) (libstdc++.so.6.0.28)
3.88	0.07	7	607 249	0x00000000048df960
3.83	0.79	8-9 (8)	619 621	free (libc-2.31.so: malloc.c)

Esses dados explicitam com a ideia de que a função *inserir* é a função mais custosa do código, o que faz sentido devido à complexidade O(n log n) associada à inserção de voos em árvores binárias balanceadas. A função inserir, que organiza os voos dentro da árvore, realiza buscas e reequilíbrios da estrutura, o que justifica o alto custo de CPU observado e o aumento logarítmico do gráfico *número de voos x tempo (ms)*.
Em resumo, a análise experimental e os dados coletados corroboram as conclusões retiradas da análise de complexidade.

CONCLUSÃO

O trabalho desenvolvido teve como objetivo a criação de um sistema eficiente para gerenciar e filtrar informações de voos, utilizando árvores binárias, com filtros específicos para organizar os dados conforme diferentes critérios. O sistema foi projetado para garantir alta performance, permitindo a inserção e organização rápida dos voos em árvores balanceadas, assegurando que o desempenho se mantivesse estável mesmo com o aumento no volume de dados.

A implementação de múltiplas árvores para diferentes critérios de filtro proporcionou grande flexibilidade e permitiu que o código atendesse às diversas necessidades de consulta e ordenação dos voos. Embora o sistema não tenha sido aprovado em todos os testes de VPLs, acredito que consegui desenvolver o trabalho de forma sólida e satisfatória, adquirindo uma compreensão profunda sobre a utilização de árvores binárias como mecanismo de filtragem e organização de dados.

Embora existam abordagens mais simples, como o uso de vetores e filas, eu percebi que as árvores balanceadas realmente fazem a diferença quando o assunto é eficiência. Durante o desenvolvimento deste trabalho, aprendi bastante sobre como a escolha certa da estrutura de dados pode impactar no desempenho de um sistema, e como, mesmo que algo pareça mais complexo à primeira vista, isso pode trazer benefícios a longo prazo. Além disso, essa experiência me ajudou a entender melhor como implementar soluções que sejam não só eficientes, mas também bem estruturadas.

BIBLIOGRAFIA

PF, Paulo. Algoritmos e Estruturas de Dados: Árvores Binárias de Busca. 2023. Disponível em: https://www.ime.usp.br/~pf/algoritmos/aulas/binst.html. Acesso em: 2 fev. 2025.

Slides da disciplina.