Universidade Federal de Minas Gerais (UFMG) -
Departamento de Ciência da Computação

Trabalho Prático 3

Estudante: Sarah Menks Sperber

Matrícula: 2023001824

Introdução:

Este trabalho tem como objetivo a implementação de um sistema de busca de passagens aéreas eficiente, permitindo a filtragem baseada em expressões lógicas e a ordenação dos resultados de acordo com critérios predefinidos. A entrada consiste em uma lista de voos, seguida de um conjunto de consultas estruturadas. Assim, no decorrer deste relatório, vamos fazer uma análise do código implementado, discutindo pontos como metodologias usadas e complexidade assintótica.

Métodos:

O código usado é bem sucinto. Foram usados dois TADs: AVL.hpp e Flight.hpp que são responsáveis por montar o esqueleto do código.
O arquivo AVL.cpp possui a declaração da maior parte das funções usadas,
Por fim, a main.cpp é aquela que executa o código, já que é ela que abre o arquivo e chama as funções.

1. Flight.hpp:

O TAD Flight.hpp é responsável apenas por criar a struct Flight, que armazena as informações dos voos (origem, destino, preço, assentos disponíveis, horário de saída, horário de chegada, quantidade de paradas e duração). Além disso, a estrutura armazena os tempos de saída e chegada do avião em segundos, informações auxiliares que apenas nos dão uma informação adicional.

2. AVL.hpp:

Cria a struct Node, que, além de ponteiros para os filhos do nó, contém a chave de comparação e um vetor de índices, que faz referência aos índices do vetor de voos, declarado no arquivo main.

O arquivo também possui as declarações de algumas funções, como as funções de inserção, remoção e balanceamento da árvore, que são as funções principais do programa. Além disso, o TAD possui outras funções auxiliares, como métodos de “Get” e “Find” que auxiliam no processo de consulta e na filtragem dos dados.

3. Main.cpp:

Este é o arquivo principal do programa, onde se localizam as declarações das variáveis (vetor de Voos e as árvores de consulta), a leitura da entrada e o manejo das funções implementadas.
A Main.cpp também possui uma pequena função auxiliar, que transforma o horário de entrada em segundos, para facilitar o cálculo do tempo de duração do voo. Além disso, para um código mais seguro, o arquivo possui boa parte das estratégias de robustez (como as checagens de parâmetro).

Análise de Complexidade:

Como dito anteriormente, foram implementados três TADs, sendo o AVL.cpp o arquivo de maior relevância, uma vez ele possui a execução da grande maioria das funções. Os casos de teste se deram, principalmente, por 10 arquivos de diferentes tamanhos, com entradas que variam entre 5 e 100 linhas.

1. Flight.hpp:

Como foi dito anteriormente, o TAD Flight não possui nenhum método, uma vez que seu objetivo é apenas armazenar os atributos dos voos. Sendo assim, não é possível realizar a análise de complexidade desta estrutura. Entretanto, se levarmos em conta as atribuições feitas na struct, percebemos que sua complexidade, no pior caso, é O(1).

2. Main:

Mesmo que sua função seja trivial em um trabalho como esse, para fins didáticos, sua função também deve ser analisada. O trabalho da main, além de receber a entrada, é integrar todo o código. O loop principal do arquivo possui as declarações do vetor de voos, assim como a inserção nas árvores de consulta.
Complexidade assintótica:

- Ler arquivo e inserir no vetor de voos: O(n)
- Inserir nas árvores: O(n)
- Imprimir os valores: O(n)
- Complexidade geral: O(3n) = O(n) - onde n é a quantidade de linhas do arquivo

3. Árvore AVL:

Como foi dito anteriormente, a árvore é a peça central do trabalho, e é nela que se encontram a maior parte das funções. A complexidade assintótica da maioria das funções depende da altura da árvore, comportamento característico das árvores. Como a árvore usada é uma árvore balanceada (árvore AVL), os algoritmos de busca, de inserção e de remoção serão log(n), que é outro comportamento característico de árvores balanceadas. Abaixo, está listado a complexidade assintótica de cada uma das funções que correspondem ao TAD AVL.hpp:

- GetBalance: O(1). Apenas calcula o valor do balanceamento, usando o balanceamento dos filhos
- GetHeight: O(1). Mais uma vez, apenas retorna o valor da altura
- RightRotate e LeftRotate: O(1). Apenas troca valores de ponteiros
- FindKey: O(log n). Usa das propriedades de árvore para procurar o nó correspondente à chave
- Insert e DeleteNode: O(log n). Usa a mesma lógica do algoritmo FindKey para encontrar a posição correta de inserção e remoção.
- FindAndInsert: O(log n). Chama o método Find para inserir um valor ao vetor de voos de um nó
- PostOrder: O(n). Percorre todos os nós da árvore para a impressão

Estratégias de Robustez:

Para garantir um código seguro e sem falhas, algumas estratégias de segurança foram implementadas, tais como:

1. Função parse_args que checa os parâmetros:

Para evitar problemas com entradas inválidas, uma função de checagem dos parâmetros foi adicionada no arquivo main.cpp. Nela, nós checamos a quantidade de parâmetros passados para o programa, e caso o usuário não tenha colocado a quantidade de parâmetros adequada, o código não funcionará.

2. Verificação de arquivo válido:

Assim como a função parse_args checa os parâmetros, a função main possui um teste de checagem do arquivo. Se o usuário não passar a entrada desejada por parâmetro (um arquivo válido), o código é encerrado e uma mensagem de texto é exibida.

3. Uso de try catch:

Além das funções mencionadas acima, o código se utiliza de blocos try catch para lidar com exceções no programa, evitando que ele encerre abruptamente ao encontrar um erro. Esse mecanismo permite capturar erros, tratá-los e continuar a execução do código de forma controlada.

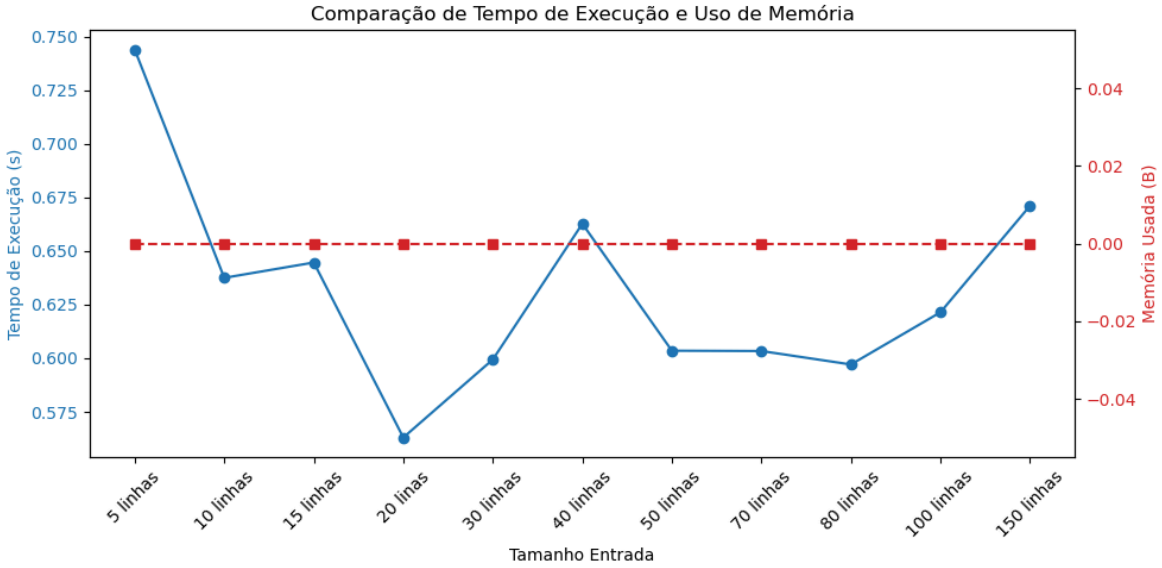
Análise Experimental:

Para a análise temporal, usei a biblioteca chromo para medir o tempo de execução do código. Dessa forma, podemos entender melhor a relação entre o tamanho da entrada e o desempenho do código

Arquivo	**Tamanho da entrada**	**Tempo (segundos)**
0	5 linhas	0.7439625263214111
1	10 linhas	0.6375153064727783
2	15 linhas	0.6446638107299805
3	20 linhas	0.5630221366882324
4	30 linhas	0.5994265079498291
5	40 linhas	0.6627576351165771
6	50 linhas	0.6035230159759521
7	70 linhas	0.6033754348754883
8	80 linhas	0.5971033573150635
9	100 linhas	0.6214601993560791
10	150 linhas	0.6711087226867676

Através dessas informações, vemos que, mesmo com uma certa variância dos valores, não foi possível identificar um aumento do tempo gasto proporcional ao tamanho do arquivo.

Agora, usando o valgrind, podemos olhar a forma como o código usa a memória do computador, e como as duas informações estão correlacionadas. Neste primeiro gráfico, como a unidade de medida foi truncada, não é possível perceber uma grande variação no uso de memória



Porém, quando examinamos o uso de memória mais profundamente, vemos que, dentro do mesmo caso, existe um certo crescimento no uso do heap.

n	time(i)	total(B)	useful-heap(B)	extra-heap(B)
0	0	0	0	0
1	1,988,858	73,736	73,728	8
2	2,005,991	90,608	90,560	48
3	2,026,601	90,776	90,694	82
4	2,037,311	98,968	98,684	284
5	2,061,747	99,024	98,715	309
6	2,077,427	105,416	104,912	504
7	2,090,911	105,472	104,943	529
8	2,107,281	112,760	112,012	748
9	2,120,732	112,816	112,043	773
10	2,137,310	120,104	119,112	992

n	time(i)	total(B)	useful-heap(B)	extra-heap(B)
63	2,914,538	277,760	271,104	6,656
64	2,940,915	191,704	186,238	5,466

Legenda tabela:

n: número da linha;
time(i): Medição em períodos de tempo;
Total(B): Quantidade de memória usada naquele ponto;
Useful-heap(B): Bytes alocados no heap, que são necessários para o programa;
Extra-heap(B): Memória extra alocada, como um processo auxiliar;

Conclusão:

Neste trabalho, foi proposto o desenvolvimento de um sistema para a busca de passagens aéreas que permitiria a filtragem dos voos baseada em expressões lógicas e na ordenação dos resultados conforme critérios específicos. A implementação seguiu a estrutura proposta, armazenando os voos em memória usando árvores balanceadas. No entanto, alguns requisitos não foram completamente atendidos, como a execução de consultas e a avaliação completa das expressões lógicas utilizando uma árvore sintática.

Entretanto, durante a implementação, obtive mais conhecimento em estruturas de dados, mais especificamente em arrays e árvores balanceadas. A experiência evidenciou a importância da escolha correta nos algoritmos, e como isso impacta diretamente o desempenho do sistema, especialmente ao lidar com grandes volumes de informações. Apesar das limitações da implementação atual, o projeto serviu como base para um entendimento mais profundo sobre estratégias de armazenamento e processamento de consultas, abrindo espaço para futuras melhorias e refinamentos.

Bibliografia:

MATPLOTLIB. Matplotlib 3.0.0 documentation. Disponível em: https://matplotlib.org/stable/. Acesso em: 2 dez. 2024.

C++ REFERENCE. chrono - C++ Reference. Disponível em: https://cplusplus.com/reference/chrono/. Acesso em: 2 dez. 2024.

VALGRIND. Valgrind User Manual. Disponível em: https://valgrind.org/docs/manual/ms-manual.html. Acesso em: 2 dez. 2024.
6

LACERDA, Anísio; MEIRA JR., Wagner. Slides virtuais da disciplina de estruturas de dados. Belo Horizonte: Universidade Federal de Minas Gerais, 2024.