Trabalho Prático 3 – Busca de Passagens Aéreas

RAFAEL NEUBANER DE ALMEIDA – 2023001638

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

Belo Horizonte – MG – Brazil

neubanerrafael@gmail.com

1-Introdução

Esse código foi feito para ler um arquivo de passagens de avião e armazenalos em arvores, para após todo o armazenamento fazer a leitura de linhas de comando com “filtros” que seriam aplicados nas passagens para retornar um numero X de passagens que cumpram com as características do filtro: valor, paradas, assentos livres, origem e destino. Após a filtragem as passagens devem ser impressas ordenadas por 3 fatores, preço, numero de paradas e duração, sendo os critérios primário secundário e terciário de ordenação passados por um trigrama no inicio de cada linha de pesquisa. Ex: PDS ordena como critério primário por preço, secundário duração e terciário stops (paradas).

2-Método

O programa foi desenvolvido na linguagem C++, compilado pelo compilador GCC da GNU Compiler Collection. O Computador utilizado tem as seguintes especificaç̃oes:

• Sistema Operacional: Linux Ubuntu 22.04

• Processador: 12th Gen Intel(R) Core(TM) i7-12650h @ 2.30GHz -

• RAM: 32,00 GB (utilizavel: 31,80 GB)

2.1 Organização do código

O código segue os princípios da programação orientada a objetos, utilizando varias classes para implementação das estruturas de dados. Essas classes são utilizadas pelo arquivo main do programa em cima de um arquivo txt lido que pega os dados de cada passagem e as armazena em AVLs, após a leitura de

todas as passagens são lidas e executadas todas as pesquisas recebidas imprimindo seu resultado na saída padrão.

2.2 Funcionamento

Após ler e armazenar todos os dados do arquivo o programa lê o numero de pesquisas que serão realizadas e para cada pesquisa entra num loop que roda N vezes onde N é o numero de filtros que serão aplicados, cada filtro salva em um vetor temporário os índices aos quais as passagens que foram filtradas correspondem, e, após a filtragem esses índices são chocados para se descobrir a interseção dos filtros e inserir as passagens correspondentes em um minheap ordenado de acordo com o trigrama recebido. Após a inserção e ordenação por meio do minheap o programa entra num loop de 0 ate N onde N agora é o numero de passagens solicitadas recebido pela linha de comando de filtragem, onde N remoções do minheap são realizadas e impressas na tela.

3-instruções de compilação

O programa pode ser executado utilizando o comando make run após passar como parâmetro para o comando “all” o nome do arquivo txt que sera lido e deverá estar localizado na pasta raiz juntamente com o makefile. Dessa forma ao rodar o comando make run o programa já irá printar na tela todas as pesquisas realizadas.

4-Análise de complexidade

Vamos analisar a complexidade de tempo e espaço das funções dos códigos fornecidos.

4.1 - Rafael.cpp

Função push
Complexidade de Tempo: O(n) - No pior caso, quando o array precisa ser redimensionado, copia todos os elementos para um novo array de tamanho dobrado.
Complexidade de Espaço: O(n) - No pior caso, aloca espaço adicional para um novo array de tamanho dobrado.
Função get(int index) const
Complexidade de Tempo: O(1) - Acessa um elemento em um índice específico.
Complexidade de Espaço: O(1) - Não utiliza espaço adicional.
Função pop()
Complexidade de Tempo: O(1) - Remove o último elemento.
Complexidade de Espaço: O(1) - Não utiliza espaço adicional.

Função size() const
Complexidade de Tempo: O(1) - Retorna o número de elementos.
Complexidade de Espaço: O(1) - Não utiliza espaço adicional.
Função getcapacity() const
Complexidade de Tempo: O(1) - Retorna a capacidade do array.
Complexidade de Espaço: O(1) - Não utiliza espaço adicional.
Função countElements() const
Complexidade de Tempo: O(1) - Retorna o número de elementos armazenados.
Complexidade de Espaço: O(1) - Não utiliza espaço adicional.
Função print() const
Complexidade de Tempo: O(n) - Percorre e imprime todos os elementos.
Complexidade de Espaço: O(1) - Não utiliza espaço adicional.
Funções begin() e end()
Complexidade de Tempo: O(1) - Retorna ponteiros para o início e o fim do array.
Complexidade de Espaço: O(1) - Não utiliza espaço adicional.
Operadores operator[]
Complexidade de Tempo: O(1) - Acessa um elemento em um índice específico.
Complexidade de Espaço: O(1) - Não utiliza espaço adicional.

4.2 - Passagens.cpp

Construtor Passagem()
Complexidade de Tempo: O(1) - Inicializa um objeto Passagem.
Complexidade de Espaço: O(1) - Não utiliza espaço adicional.
Destrutor ~Passagem()
Complexidade de Tempo: O(1) - Destrói um objeto Passagem.
Complexidade de Espaço: O(1) - Não utiliza espaço adicional.
Função Print()
Complexidade de Tempo: O(1) - Imprime os atributos de um objeto Passagem.
Complexidade de Espaço: O(1) - Não utiliza espaço adicional.
Função horatotal()
Complexidade de Tempo: O(1) - Calcula a diferença de tempo entre duas datas.
Complexidade de Espaço: O(1) - Não utiliza espaço adicional.

4.3 - MinHeap.cpp

Construtor MinHeap(int capacidade)
Complexidade de Tempo: O(1) - Inicializa um heap com capacidade especificada.
Complexidade de Espaço: O(n) - Aloca espaço para o array de Passagem.
Destrutor ~MinHeap()
Complexidade de Tempo: O(1) - Libera a memória alocada.

Complexidade de Espaço: O(1) - Não utiliza espaço adicional.
Funções pai(int i), filhoEsquerdo(int i), filhoDireito(int i)
Complexidade de Tempo: O(1) - Calcula os índices dos nós pai e filhos.
Complexidade de Espaço: O(1) - Não utiliza espaço adicional.
Função troca(int i, int j)
Complexidade de Tempo: O(1) - Troca dois elementos no heap.
Complexidade de Espaço: O(1) - Não utiliza espaço adicional.
Funções subirpds(int i), subirpsd(int i), subirdps(int i), subirdsp(int i), subirspd(int i), subirsdp(int i)
Complexidade de Tempo: O(log n) - No pior caso, percorre a altura do heap para ajustar a posição do elemento.
Complexidade de Espaço: O(1) - Não utiliza espaço adicional.
Funções descerpds(int i), descerdsp(int i), descerpsd(int i), descerdps(int i), descerspd(int i), descersdp(int i)
Complexidade de Tempo: O(log n) - No pior caso, percorre a altura do heap para ajustar a posição do elemento.
Complexidade de Espaço: O(1) - Não utiliza espaço adicional.
Funções pds(const Passagem &a, const Passagem &b), psd(const Passagem &a, const Passagem &b), dsp(const Passagem &p1, const Passagem &p2), dps(const Passagem &p1, const Passagem &p2), spd(const Passagem &p1, const Passagem &p2), sdp(const Passagem &p1, const Passagem &p2)
Complexidade de Tempo: O(1) - Compara dois objetos Passagem.
Complexidade de Espaço: O(1) - Não utiliza espaço adicional.
Função redimensionar()
Complexidade de Tempo: O(n) - No pior caso, copia todos os elementos para um novo array de tamanho aumentado.
Complexidade de Espaço: O(n) - No pior caso, aloca espaço adicional para um novo array de tamanho aumentado.
Função inserir(Passagem p, std::string ord)
Complexidade de Tempo: O(log n) - No pior caso, insere um elemento e ajusta a posição no heap.
Complexidade de Espaço: O(1) - Não utiliza espaço adicional além do necessário para redimensionar o heap.
Função remover(std::string ord)
Complexidade de Tempo: O(log n) - No pior caso, remove o elemento de maior prioridade e ajusta a posição no heap.
Complexidade de Espaço: O(1) - Não utiliza espaço adicional.
Função vazio() const
Complexidade de Tempo: O(1) - Verifica se o heap está vazio.
Complexidade de Espaço: O(1) - Não utiliza espaço adicional.

AVL.hpp

Construtor AVLTree()

Complexidade de Tempo: O(1) - Inicializa a árvore AVL.
Complexidade de Espaço: O(1) - Não utiliza espaço adicional.
Função height(AVLNode<T> *node)
Complexidade de Tempo: O(1) - Retorna a altura de um nó.
Complexidade de Espaço: O(1) - Não utiliza espaço adicional.
Função balanceFactor(AVLNode<T> *node)
Complexidade de Tempo: O(1) - Calcula o fator de balanceamento de um nó.
Complexidade de Espaço: O(1) - Não utiliza espaço adicional.
Função rightRotate(AVLNode<T> *y)
Complexidade de Tempo: O(1) - Realiza uma rotação à direita.
Complexidade de Espaço: O(1) - Não utiliza espaço adicional.
Função leftRotate(AVLNode<T> *x)
Complexidade de Tempo: O(1) - Realiza uma rotação à esquerda.
Complexidade de Espaço: O(1) - Não utiliza espaço adicional.
Função insert(AVLNode<T> *node, T key, int index)
Complexidade de Tempo: O(log n) - No pior caso, percorre a altura da árvore para inserir um novo nó e ajustar o balanceamento.
Complexidade de Espaço: O(log n) - No pior caso, utiliza espaço na pilha de chamadas proporcional à altura da árvore devido à recursão.
Função minValueNode(AVLNode<T> *node)
Complexidade de Tempo: O(log n) - No pior caso, percorre a altura da subárvore para encontrar o nó com o menor valor.
Complexidade de Espaço: O(1) - Não utiliza espaço adicional.
Função deleteNode(AVLNode<T> *root, T key)
Complexidade de Tempo: O(log n) - No pior caso, percorre a altura da árvore para deletar um nó e ajustar o balanceamento.
Complexidade de Espaço: O(log n) - No pior caso, utiliza espaço na pilha de chamadas proporcional à altura da árvore devido à recursão.
Função inorder(AVLNode<T> *root)
Complexidade de Tempo: O(n) - Percorre todos os nós da árvore em ordem.
Complexidade de Espaço: O(n) - No pior caso, utiliza espaço na pilha de chamadas proporcional ao número de nós na árvore devido à recursão.
Função search(AVLNode<T> *root, T key)
Complexidade de Tempo: O(log n) - No pior caso, percorre a altura da árvore para buscar um nó.
Complexidade de Espaço: O(log n) - No pior caso, utiliza espaço na pilha de chamadas proporcional à altura da árvore devido à recursão.
Função collectCheaperPassages(AVLNode<T> *node, Rafael<Passagem> &temp, T maxValue, Passagem *passagens)
Complexidade de Tempo: O(n) - No pior caso, percorre todos os nós da árvore para coletar passagens mais baratas.
Complexidade de Espaço: O(n) - No pior caso, utiliza espaço na pilha de chamadas proporcional ao número de nós na árvore devido à recursão.

Função collectMoreExpensivePassages(AVLNode<T> *node, Rafael<Passagem> &temp, T minValue, Passagem *passagens)
Complexidade de Tempo: O(n) - No pior caso, percorre todos os nós da árvore para coletar passagens mais caras.
Complexidade de Espaço: O(n) - No pior caso, utiliza espaço na pilha de chamadas proporcional ao número de nós na árvore devido à recursão.
Função insere(T key, int index)
Complexidade de Tempo: O(log n) - No pior caso, percorre a altura da árvore para inserir um novo nó e ajustar o balanceamento.
Complexidade de Espaço: O(log n) - No pior caso, utiliza espaço na pilha de chamadas proporcional à altura da árvore devido à recursão.
Função remove(T key)
Complexidade de Tempo: O(log n) - No pior caso, percorre a altura da árvore para deletar um nó e ajustar o balanceamento.
Complexidade de Espaço: O(log n) - No pior caso, utiliza espaço na pilha de chamadas proporcional à altura da árvore devido à recursão.
Função searche(T key)
Complexidade de Tempo: O(log n) - No pior caso, percorre a altura da árvore para buscar um nó.
Complexidade de Espaço: O(log n) - No pior caso, utiliza espaço na pilha de chamadas proporcional à altura da árvore devido à recursão.
Função getCheaperPassages(Rafael<Passagem> &temp, double maxValue, Passagem *passagens)
Complexidade de Tempo: O(n) - No pior caso, percorre todos os nós da árvore para coletar passagens mais baratas.
Complexidade de Espaço: O(n) - No pior caso, utiliza espaço na pilha de chamadas proporcional ao número de nós na árvore devido à recursão.
Função getMoreExpensivePassages(Rafael<Passagem> &temp, double minValue, Passagem *passagens)
Complexidade de Tempo: O(n) - No pior caso, percorre todos os nós da árvore para coletar passagens mais caras.
Complexidade de Espaço: O(n) - No pior caso, utiliza espaço na pilha de chamadas proporcional ao número de nós na árvore devido à recursão.
Função printInorder()
Complexidade de Tempo: O(n) - Percorre todos os nós da árvore em ordem.
Complexidade de Espaço: O(n) - No pior caso, utiliza espaço na pilha de chamadas proporcional ao número de nós na árvore devido à recursão.

Main.cpp

Função isOperatorChar(char c)
Complexidade de Tempo: O(1) - Verifica se um caractere é um operador binário.
Complexidade de Espaço: O(1) - Não utiliza espaço adicional.
Função parsePredicates(const std::string &input)

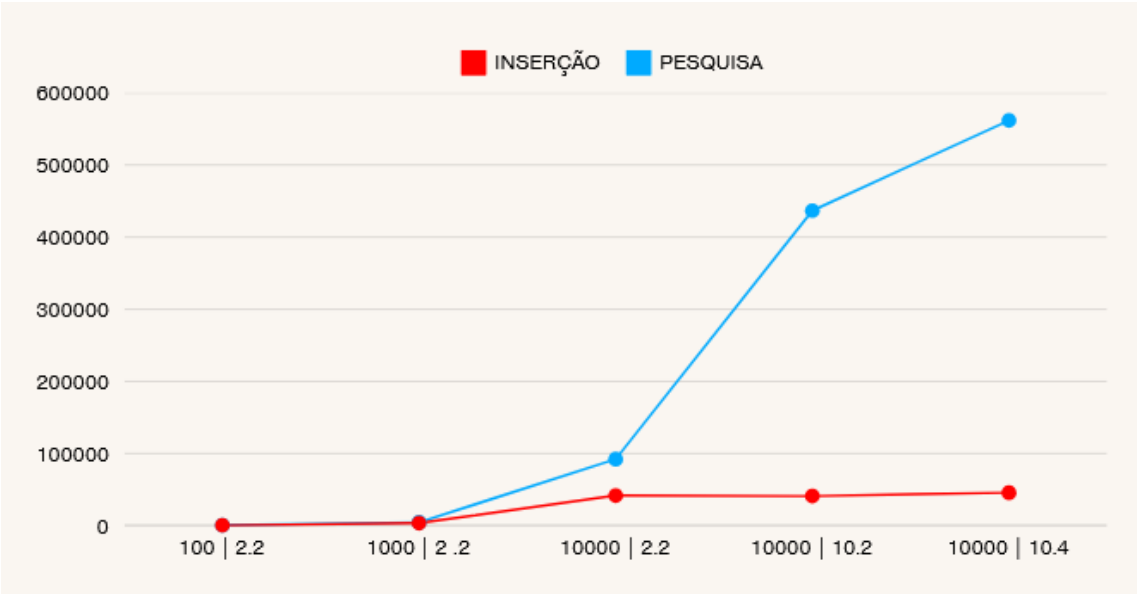
Complexidade de Tempo: O(m) - Onde m é o tamanho da string de entrada. Percorre a string para analisar os predicados.
Complexidade de Espaço: O(p) - Onde p é o número de predicados. Armazena os predicados em um vetor.
Função main(int argc, char const *argv[])
Complexidade de Tempo: O(n log n + k log k) - Onde n é o número de passagens e k é o número de consultas. Inserir n passagens na árvore AVL leva O(n log n). Para cada consulta, a complexidade é O(k log k) devido às operações de heap.
Complexidade de Espaço: O(n) - Armazena n passagens e as árvores AVL.

5-Estrategias de robustez

Foi feita uma condição para testar se o arquivo pode ser aberto para evitar que fossem criadas variáveis sem os parâmetros passados pelo arquivo e ocasionar consequentemente em possíveis erros.
O código foi comentado e identado para melhor leitura e entendimento.
Tambem foi feito uso da modularização do código para melhor organização das implementações.

6-Análise experimental

Utilizei a biblioteca chronus para medir os tempos de execução das funções do código e montar o gráfico a seguir:



O eixo X do gráfico representa o numero de passagens do arquivo de entrada | numero de pesquisas . numero de filtros por pesquisa. Enquanto o eixo Y representa o tempo em microssegundos que o código ficou rodando.
Graças à ordem de complexidade do código ser razoavelmente baixa, O(nlogn) podemos perceber que mesmo com cargas bem elevadas o código se executa em menos de um segundo.

Fica explicito no gráfico a seguir que a complexidade de inserção das passagens nas arvores se deu de acordo com a complexidade logn ao observar que teve pouco crescimento com o aumento entre as três cargas analisadas (100, 1000, 10000) e que as pesquisas possuem um aumento muito grande quando o numero de pesquisas aumenta, obviamente visto que o laço de filtragem será executado mais vezes, e também um aumento considerável quando se aumenta o numero de filtros analisados já que a filtragem é que representa o procedimento mais custoso do código (O(nlogn)).

Utilizei também as ferramentas do valgrind para testar os acessos a memória, cache e chamadas das funções. Após rodar o memcheck não foi identificada nenhum vazamento de memória nem acesso indevido a posições de memória durante a execução do programa.

7-Conclusao

Esse trabalho lidou com a problemática de filtragem de passagens de avião por meio de preferencias e comandos fornecidos. O objetivo dele era de através de um arquivo de passagens obtido, armazenar todos os voos em arvores de pesquisa balanceada ordenadas por índices, no caso em analise, origem, destino, numero de paradas, numero de assentos, data e hora de saída, data e hora de chegada, preço e duração, onde serão posteriormente realizadas pesquisas de filtragem para retornar as passagens que atendem a todos os critérios fornecidos ordenadas por parâmetros passados juntamente das pesquisas representados sempre por um trigrama composto pelas letras P (preço), D (duração), e S(stops).

Através do decorrer do trabalho foi importante os conhecimentos em arvores para se montar o TAD AVL que é o ponto critico e central de todo o código, de forma a modela-lo para ser balanceado e implementa-lo de forma a atender todos os possíveis tipos de chave. Alem também das pesquisas que a depender dos filtros passados devem retornar caminhamentos prefix infix ou posfix para percorrer todos os nós necessários para completar a filtragem. Fiquei mais confortável e consegui entender melhor na pratica como se da o funcionamento de uma AVL e seus métodos de pesquisa e caminhamento

8-Referências bibliográficas

Chaimowicks, L. and Prates, R. Slides virtuais da disciplina de estrutura de dados.
Disponibilizados via Moodle. Departamento de Ciencia da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

Lacerda, A. and Meira, W. Slides virtuais da disciplina de estrutura de dados. Disponibilizados via Moodle. Departamento de Ciencia da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.