UNIVERSIDADE FEDERAL DE MINAS GERAIS
Faculdade de Engenharia de Produção

Trabalho Prático 3 - Busca de Passagens Aéreas

Tasso A. T. Pimmenta
2021072198
TassoossaT842@gmail.com

3 de fevereiro de 2025

Sumário

Resumo

A busca de passagens aéreas consiste em busca os melhores voos disponiveis de acordo com os criterios de escolha, a organização da busca se deu por meio de arvores binarias balanceadas, um por cada critério de escolha, onde os elementos eram organizados de modo em que apenas os indices de cada item ficavam nos nós correspondentes. O codigo retorna a lista voos disponiveis de acordo com os criterios de escolha.

1 Introdução

Esta documentação lida com o problema de busca eficiente de passagens aéreas para a empresa Xulambs Tour. O objetivo é desenvolver um sistema que permita aos usuários realizarem buscas personalizadas através de expressões lógicas e definirem critérios específicos de ordenação dos resultados.

Para resolver o problema citado, foi seguida uma abordagem baseada em árvores binárias balanceadas, que permite realizar buscas eficientes considerando múltiplos critérios de ordenação. A solução processa arquivos de entrada contendo listas de voos e consultas, retornando os resultados filtrados e ordenados conforme as especificações de cada consulta.

A seção 2 descreve as principais classes implementadas e suas funções. Já na seção 3, é apresentada a análise de complexidade dos principais procedimentos implementados. A seção 4 trata das estratégias de robustez adotadas para garantir a confiabilidade do sistema. Por fim, a seção 5 apresenta a análise experimental dos resultados obtidos e a seção 6 traz as conclusões do trabalho.

2 Implementação

Nesta seção, apresentamos as decisões tomadas para a resolução do problema proposto.

O código está organizado em várias classes, cada uma responsável por uma parte específica do sistema. As principais classes são:

- **Flight**: Representa um voo, contendo informações como origem, destino, preço, assentos disponíveis, horários de partida e chegada, e número de paradas.
- **Date**: Manipula datas e horários, permitindo operações como comparação e cálculo de diferenças de tempo.
- **SequentialList**: Implementa uma lista sequencial para armazenar objetos do tipo Flight.
- **BalancedBinaryTree**: Implementa uma árvore binária balanceada para armazenar e buscar índices de voos com base em diferentes critérios (origem, destino, preço, etc.). Existe uma árvore para cada índice, permitindo buscas eficientes e rápidas. Cada nó da árvore contém um valor e uma lista de índices dos voos que possuem esse valor.

- **QuickSort**: Implementa o algoritmo de ordenação QuickSort para ordenar os voos de acordo com os critérios especificados.
- **DynamicArray**: Implementa um array dinâmico para armazenar índices de voos resultantes das consultas. Ele pode redimensionar automaticamente sua capacidade conforme necessário e suporta operações de união e interseção de conjuntos.
- **Expression**: Avalia expressões lógicas para filtrar os voos com base nos critérios fornecidos. Esta classe é responsável por analisar e avaliar expressões lógicas complexas, combinando resultados de múltiplas árvores binárias balanceadas para retornar os índices dos voos que atendem aos critérios especificados.

O programa principal lê os arquivos de entrada contendo listas de voos e consultas, processa as consultas utilizando as classes mencionadas e retorna os resultados filtrados e ordenados conforme as especificações.

A configuração utilizada para testar o programa foi a seguinte:

- Sistema Operacional: Windows 11
- Linguagem de Programação: C++ (C++11 standard)
- Compilador: GCC 9.3.0
- Processador: Ryzen 5-5500U
- Memória RAM: 8 GB

3 Análise de Complexidade

Nesta seção, analisamos a complexidade de tempo e espaço dos principais procedimentos implementados, utilizando a notação assintótica.

3.1 BalancedBinaryTree

A inserção em uma árvore binária balanceada tem complexidade de tempo $O(\log n)$, onde n é o número de elementos na árvore. A busca por índices também tem complexidade $O(\log n)$. O espaço utilizado pela árvore é $O(n)$.

3.2 QuickSort

O algoritmo QuickSort tem uma complexidade de tempo média de $O(n \log n)$ e uma complexidade de tempo no pior caso de $O(n^2)$, onde n é o número de elementos a serem ordenados. No entanto, com uma boa escolha de pivô, a complexidade média é $O(n \log n)$. O espaço utilizado é $O(\log n)$ devido à recursão.

3.3 DynamicArray

A inserção em um array dinâmico tem complexidade de tempo amortizada $O(1)$. As operações de união e interseção têm complexidade $O(n \cdot m)$, onde n e m são os tamanhos dos arrays envolvidos. O espaço utilizado é $O(n)$.

3.4 Expression

A avaliação de expressões lógicas depende da complexidade das operações de busca nas árvores binárias balanceadas. Cada operação de busca tem complexidade $O(\log n)$, e a avaliação de uma expressão lógica complexa pode envolver múltiplas operações de busca, resultando em uma complexidade total de $O(k \log n)$, onde k é o número de operações de busca.

3.5 SequentialList

A inserção em uma lista sequencial tem complexidade de tempo $O(1)$. O acesso a elementos por índice também tem complexidade $O(1)$. O espaço utilizado é $O(n)$.

4 Estratégias de Robustez

Nesta seção, descrevemos os mecanismos de programação defensiva e tolerância a falhas implementados para garantir a robustez do sistema.

4.1 Validação de Entrada

No início do programa, validamos o número de argumentos passados pela linha de comando e verificamos se o arquivo de entrada foi aberto corretamente. Caso contrário, exibimos mensagens de erro apropriadas e encerramos a execução do programa.

4.2 Tratamento de Exceções

Utilizamos exceções para lidar com situações inesperadas, como tentativas de acesso a índices fora do intervalo válido em listas sequenciais e erros de alocação de memória. As exceções são capturadas e tratadas de forma a garantir que o programa não falhe de maneira catastrófica.

4.3 Verificação de Limites

Em várias partes do código, verificamos se os índices e tamanhos estão dentro dos limites esperados antes de realizar operações. Por exemplo, ao inserir elementos em arrays dinâmicos ou listas sequenciais, garantimos que a capacidade não seja excedida.

4.4 Gerenciamento de Memória

Implementamos mecanismos para garantir que a memória alocada dinamicamente seja liberada corretamente, evitando vazamentos de memória. Por exemplo, no destrutor da classe `QuickSort`, liberamos a memória alocada para o array dinâmico.

4.5 Redimensionamento Dinâmico

Para estruturas de dados como `DynamicArray`, implementamos redimensionamento dinâmico para garantir que a capacidade seja ajustada conforme necessário, evitando estouros de capacidade e garantindo a eficiência das operações de inserção.

4.6 Mensagens de Erro Informativas

Sempre que ocorre um erro, exibimos mensagens de erro informativas que ajudam a identificar a causa do problema. Isso facilita a depuração e a correção de erros.

4.7 Testes e Validação

Realizamos testes extensivos para validar o comportamento do sistema em diferentes cenários, incluindo casos de borda e situações de erro. Isso nos permite identificar e corrigir problemas antes que o sistema seja utilizado em produção.

5 Análise Experimental

Os experimentos foram muito positivos, o codigo foi capaz de retornar os voos disponiveis de acordo com os criterios de escolha, os dez itens de input foram testados e todos retornaram os resultados esperados. E os dez itens do VPL também deram os resultados esperados. O codigo também aceita varias opções de operações logicas de busca além da &&, o operador || também é aceito.

6 Conclusão

Este trabalho lidou com o problema de busca eficiente de passagens aéreas para a empresa Xulambs Tour, na qual a abordagem utilizada para sua resolução foi baseada em árvores binárias balanceadas.

Com a solução adotada, pode-se verificar que o sistema é capaz de realizar buscas eficientes e personalizadas, retornando os voos disponíveis de acordo com os critérios de escolha especificados pelos usuários. A implementação das classes e estruturas de dados permitiu uma organização eficiente e um processamento rápido das consultas.

Por meio da resolução desse trabalho, foi possível praticar os conceitos relacionados a estruturas de dados avançadas, como árvores binárias balanceadas, listas sequenciais, arrays dinâmicos e algoritmos de ordenação. Além disso, a implementação de expressões lógicas complexas para filtragem de dados proporcionou um entendimento mais profundo sobre a avaliação de expressões e operações lógicas.

Durante a implementação da solução para o problema, houveram importantes desafios a serem superados, por exemplo, a correta manipulação de datas e horários, a otimização das operações de busca e ordenação, e a garantia de robustez e eficiência do sistema. No entanto, esses desafios foram superados com sucesso, resultando em um sistema funcional e eficiente.