Estrutura De Dados
Trabalho Prático 3

Bruno Lima Soares

Matrícula: 2022055785

3 de fevereiro de 2025

1 Introdução

O problema central desse trabalho prático, consiste em criar um sistema que permita aos usuários filtrar e ordenar voos de acordo com critérios personalizados, como preço, duração e número de paradas. Dado que, com um processo cada vez maior de globalização e conexão do mundo, unido à necessidade de economizar tempo, a aviação se tornou extremamente presente, formando um cenário, bastante real, visto que, no mundo atual, a busca por passagens aéreas exige respostas rápidas e precisas, impactando positivamente a experiência dos usuários e a eficiência operacional do setor.

A implementação do sistema demandou, primeiramente, uma abstração da lógica por trás das consultas – identificar como transformar expressões lógicas complexas em uma estrutura que pudesse ser processada pelo computador. A transição dessa lógica abstrata para código efetivo representou um dos maiores desafios do projeto. Além disso, a gestão da lógica das expressões, utilizando árvores de expressão, e a ordenação dos dados dos voos foram pontos críticos que exigiram atenção especial, ressaltando a importância de estruturas como as árvores AVL e algoritmos de ordenação como o quicksort.

Pude perceber através desse TP, e de forma geral, dos trabalhos, práticas e conversas ao longo da disciplina, como as soluções computacionais podem transformar desafios cotidianos em problemas manejáveis e, muitas vezes, resolver questões reais com eficiência. No projeto, voltado à busca de passagens aéreas, pude desenvolver não apenas uma implementação técnica, mas uma reflexão sobre como a escolha adequada de estruturas de dados e algoritmos pode impactar diretamente a otimização de processos do dia a dia.

O desenvolvimento e os testes ocorreram em uma máquina com as seguintes especificações:

- **Processador:** Ryzen 7 5700;
- **Memória RAM:** 32GB 3200MHz;
- **Sistema Operacional:** Sub-Sistema Linux (WSL);
- **Memória interna:** SSD 960GB NVME;

Essa configuração possibilitou a realização de testes e a validação do desempenho do sistema em cenários com grande volume de dados, sem sofrer tanto impacto em função de atividades secundárias e tarefas do sistema operacional.

2 Método

O sistema foi desenvolvido de forma modular, com cada módulo responsável por uma parte essencial da funcionalidade. A seguir, descrevemos os principais módulos e algoritmos implementados:

2.1 Módulo AVLTree

O módulo `AVLTree` implementa uma árvore binária de busca balanceada (AVL). Essa estrutura é utilizada para indexar os voos por diferentes atributos (por exemplo, preço, duração, número de paradas, datas) e permite operações de inserção e consulta com alta eficiência, tendo complexidade média de $O(\log n)$. Cada nó da árvore contém:

- Uma chave;
- Uma lista encadeada (`FlightListNode`) para armazenar voos com chaves duplicadas;
- Ponteiros para os filhos esquerdo e direito;
- A altura do nó, utilizada para manter o balanceamento da árvore.

As rotações à esquerda e à direita são aplicadas após cada inserção para garantir que a árvore permaneça balanceada, o que é crucial para a eficiência das consultas, especialmente em bases de dados com grande quantidade de registros.

2.2 Módulo Flight

Este módulo define a estrutura `Flight`, que contém os atributos relevantes de um voo, tais como origem, destino, preço, assentos disponíveis, horários (tanto em formato string quanto numérico) e número de paradas. Essa estrutura serve como a base para o armazenamento dos dados e para a execução de consultas e ordenação dos resultados.

2.3 Módulo Expression e Parser

Para processar as consultas, o sistema utiliza uma árvore de expressão lógica. O módulo `Expression` define:

- `Expr`: a classe abstrata que serve de base para todas as expressões;
- `BinaryExpr`: que lida com expressões lógicas binárias, como AND e OR;

- `NotExpr`: que implementa a negação (NOT);
- `PredicateExpr`: que representa predicados de comparação (por exemplo, `prc<=500`).

O módulo `Parser` é responsável por analisar a string da consulta e transformá-la em uma árvore de expressões. Esse parser utiliza técnicas de análise recursiva, possibilitando a interpretação correta de expressões complexas e aninhadas. A eficiência deste módulo é essencial, visto que uma consulta mal interpretada pode comprometer a filtragem dos dados.

2.4 Módulo DateTime

O módulo `DateTime` converte strings de data/hora para o formato `time_t` (UTC), permitindo a comparação e o cálculo da duração dos voos. Essa conversão é fundamental para que os voos sejam corretamente indexados e comparados com base em seus horários.

2.5 Módulo Sort

A ordenação dos voos filtrados é realizada pelo módulo `Sort`, que utiliza o algoritmo de quicksort. As funções principais deste módulo são:

- `compareFlightByCriteria`: Esta função compara dois voos de acordo com os critérios definidos – preço (`p`), duração (`d`) e número de paradas (`s`). A função avalia cada critério na ordem especificada, retornando -1, 0 ou 1 conforme a relação entre os atributos dos voos. Essa abordagem permite uma ordenação personalizada e hierárquica dos resultados.
- `quickSortFlights`: Esta função ordena um array de ponteiros para `Flight` utilizando o algoritmo de quicksort, com complexidade média de $O(n \log n)$. A função realiza a partição do array com base em um pivô escolhido (geralmente o último elemento do array) e, recursivamente, ordena as subpartições. Apesar de o pior caso ter complexidade $O(n^2)$, a escolha adequada do pivô e a distribuição dos dados normalmente garantem o desempenho esperado.

2.6 Integração e Gerenciamento dos Dados

O módulo principal (`main.cpp`) integra todos os componentes, realizando a leitura dos dados dos voos, a construção dos índices por meio das árvores AVL e o processamento das consultas. Um papel central nesse processo é desempenhado pela classe `FlightManager`,

cuja função é gerenciar o conjunto de voos e construir os índices necessários para a consulta.

A Classe `FlightManager`

A classe `FlightManager` atua como um gerenciador dos voos, sendo responsável por:

- **Armazenamento dos Dados:** Receber um array de voos e o número total de voos disponíveis. Essa estrutura permite o acesso direto aos dados durante o processamento das consultas.
- **Construção dos Índices:** Através do método `buildIndices()`, a classe instancia diversas árvores AVL, cada uma indexando um atributo específico dos voos, como:
  - `indexOrigin`: Indexa o campo de origem dos voos;
  - `indexDestination`: Indexa o campo de destino;
  - `indexPrice`: Indexa os preços dos voos;
  - `indexDuration`: Indexa a duração dos voos;
  - `indexStops`: Indexa o número de paradas;
  - `indexSeats`: Indexa a quantidade de assentos disponíveis;
  - `indexDeparture` e `indexArrival`: Indexam os horários de partida e chegada, respectivamente.
- **Inserção dos Voos nos Índices:** Para cada voo no array, o método `buildIndices()` insere os dados nos respectivos índices. Isso permite que, durante a execução das consultas, os voos sejam rapidamente localizados e filtrados de acordo com os critérios especificados.
- **Gerenciamento de Recursos:** No destrutor da classe, todos os índices criados são liberados da memória, garantindo que não ocorram vazamentos e que o sistema mantenha um uso eficiente dos recursos.

Essa abordagem centralizada simplifica o código presente na função `main`, pois o gerenciamento dos voos e a construção dos índices ficam encapsulados na classe `FlightManager`, permitindo que a função principal se concentre na leitura dos dados, no processamento das consultas e na exibição dos resultados.

3 Análise de Complexidade

A escolha de algoritmos e estruturas de dados foi crucial para a eficiência do sistema. A seguir, detalhamos a análise de complexidade dos principais procedimentos:

3.1 Árvore AVL: Inserção e Consulta

- **Inserção:** Cada inserção na árvore AVL possui complexidade média de $O(\log n)$ devido ao balanceamento automático da estrutura. Embora haja um custo adicional para atualizar listas encadeadas em casos de chaves duplicadas, esse custo é normalmente pequeno.
- **Consulta por Intervalo:** A operação de range query explora o balanceamento da árvore para realizar uma busca eficiente, com complexidade de $O(\log n + k)$, onde k é o número de elementos que satisfazem o critério.

3.2 Parsing e Avaliação de Expressões

O processo de parsing da expressão, realizado pelo módulo `Parser`, opera em tempo linear em relação ao tamanho da string da consulta, ou seja, $O(m)$, onde m representa o comprimento da expressão. A avaliação da árvore de expressão, que é feita para cada voo, pode ter um custo adicional dependendo da complexidade da expressão, mas em geral, essa operação é eficiente para a maioria dos casos práticos.

3.3 Ordenação com Quicksort

- **`compareFlightByCriteria`:** Esta função é chamada repetidamente durante o processo de ordenação. Ela compara os atributos de dois voos com base nos critérios definidos (preço, duração e paradas). Cada comparação individual tem custo constante $O(1)$, e como essa função é utilizada dentro do algoritmo de quicksort, sua eficiência é essencial para a ordenação global.
- **`quickSortFlights`:** O quicksort, na média, ordena os elementos com complexidade $O(n \log n)$. A função utiliza a técnica de divisão e conquista, particionando o array e ordenando as subpartições recursivamente. Embora o pior caso seja $O(n^2)$, a escolha do pivô e a distribuição dos dados garantem, na prática, o desempenho esperado.

3.4 Análise de Espaço

Os principais consumidores de memória incluem:

- As árvores AVL, que ocupam espaço proporcional a $O(n)$, considerando os nós e as listas encadeadas para chaves duplicadas.
- Arrays dinâmicos utilizados para armazenar resultados intermediários, que também apresentam complexidade espacial $O(n)$.

4 Estratégias de Robustez

A robustez do sistema foi assegurada por meio de diversas estratégias de programação defensiva e tolerância a falhas, detalhadas a seguir:

4.1 Validação de Entrada e Tratamento de Erros

Uma das prioridades durante o desenvolvimento foi garantir que os dados de entrada fossem rigorosamente validados:

- **Validação de Arquivos:** O sistema verifica se o arquivo de dados foi aberto corretamente, emitindo mensagens de erro claras caso contrário.
- **Consistência dos Dados:** Durante a leitura dos voos, são realizadas verificações para assegurar que os dados (como horários de partida e chegada) estejam consistentes. Por exemplo, o sistema identifica e trata casos em que o horário de chegada é anterior ao de partida.
- **Mensagens Informativas:** Em todas as fases de processamento (leitura, parsing, consulta e ordenação), o sistema emite mensagens detalhadas para auxiliar na identificação e resolução de problemas.

4.2 Gerenciamento de Memória

O uso intensivo de alocação dinâmica de memória torna crucial a implementação de mecanismos de gerenciamento para evitar vazamentos:

- **Destrutores Adequados:** Todas as classes que utilizam alocação dinâmica (como `AVLTree` e as classes do módulo `Expression`) implementam destrutores que liberam a memória utilizada.

- **Fallback para Consulta Completa:** Quando não há predicados indexáveis na consulta, o sistema recorre a uma varredura completa do array de voos. Essa estratégia garante que, mesmo em situações adversas, os dados sejam processados corretamente, embora com custo adicional de desempenho.

4.3 Verificação Sintática e Robustez no Parser

O parser foi implementado com um enfoque especial na robustez:

- **Checagem de Parênteses e Operadores:** A presença correta de parênteses e operadores é verificada, e qualquer inconsistência resulta em uma mensagem de erro detalhada, interrompendo o processamento da consulta para evitar resultados incorretos.
- **Fallback e Mensagens de Erro:** Caso a expressão seja mal-formada, o sistema fornece informações que auxiliam na correção do erro, garantindo uma experiência de usuário mais intuitiva e segura.

4.4 Testes e Validação Contínua

Além das estratégias de código, foram realizados testes para validar cada componente:

- **Scripts de Teste Automatizados:** Foram criados scripts que simulam diferentes cenários e entradas, desde 100 até 500.000 voos, garantindo a robustez e escalabilidade do sistema.
- **Benchmarking e Monitoramento:** Utilizando `high_resolution_clock::now()`, o sistema mede o desempenho das operações, permitindo ajustes finos e a identificação precoce de gargalos.

5 Análise Experimental

Para validar a eficiência das estruturas de dados e algoritmos implementados, foi desenvolvido um script de testes que gerou entradas aleatórias com diferentes quantidades de voos, variando de 100 a 500.000 voos. Foram realizadas duas análises principais:

5.1 Benchmarking da Inserção

Comparou-se o tempo de inserção dos voos em uma árvore AVL com a inserção linear tradicional. Utilizando a função `high_resolution_clock::now()` para medir os tempos

de operação, observou-se que a inserção na árvore AVL apresentou um desempenho significativamente melhor, especialmente conforme o volume de dados aumentava. A Figura 1 ilustra essa comparação.

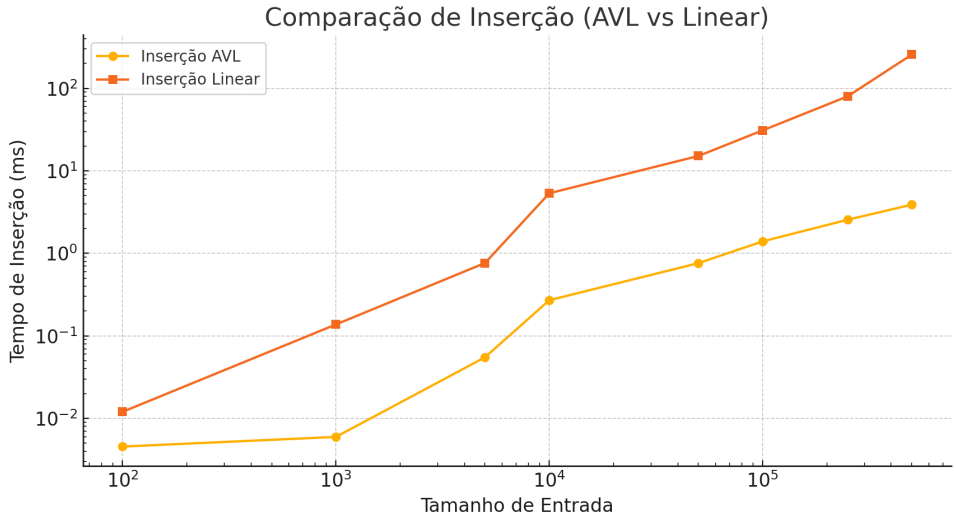

Figura 1: Benchmarking da Inserção: Comparação entre a inserção na árvore AVL e a inserção linear.

5.2 Benchmarking da Ordenação

A ordenação dos resultados filtrados foi avaliada utilizando o algoritmo de quicksort. Apesar do crescimento do número de voos, os tempos de ordenação foram pouco impactados, evidenciando a eficiência do quicksort, com seu comportamento médio de $O(n \log n)$. A Figura 2 apresenta o desempenho do quicksort conforme o tamanho dos dados aumenta.

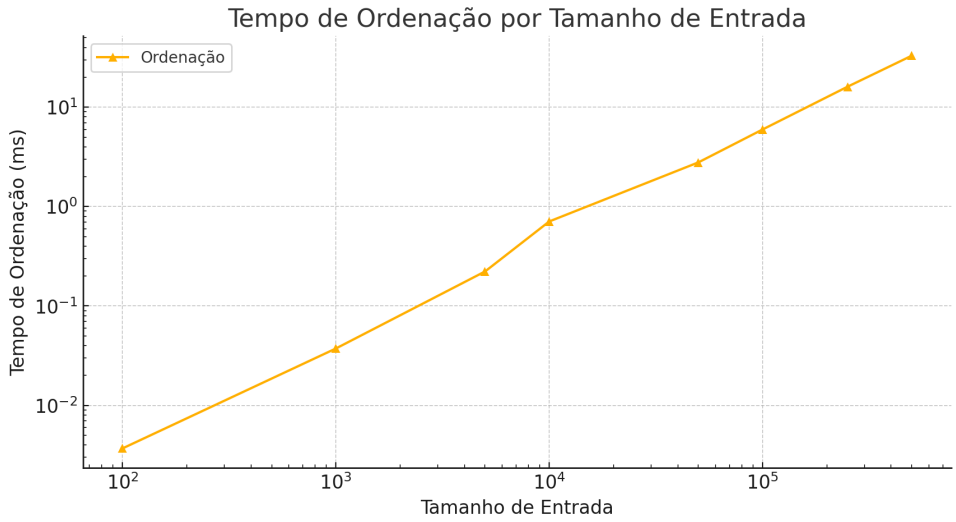

Figura 2: Benchmarking da Ordenação: Tempo de execução do quicksort para diferentes tamanhos de entrada.

Esses experimentos reforçam a importância da escolha adequada de estruturas de dados e algoritmos para a otimização de processos críticos, como foi o caso do TP com a busca e ordenação de passagens aéreas.

6 Conclusões

Neste trabalho, foi implementado um sistema de busca de passagens aéreas que integra técnicas avançadas de indexação, parsing de expressões lógicas e ordenação eficiente. A utilização de árvores AVL permitiu otimizar as operações de inserção e consulta, enquanto o algoritmo de quicksort garantiu a ordenação adequada dos resultados.

O desenvolvimento deste projeto possibilitou aprendizado sobre:

- **Estruturas de Dados:** A implementação e utilização de árvores AVL demonstrou como o balanceamento pode melhorar drasticamente o tempo de resposta em operações de busca. A abordagem mais confortável seria certamente uma inserção linear, e que em pequenos volumes de dados lidaria bem, mas que dado um contexto real, com empresas funcionando a nível macro, iria gerar gargalos. Esse processo demonstrou o quanto a aplicação de uma Estrutura de Dados correta e assertiva pode potencializar processos.
- **Parsing e Interpretação de Expressões:** A conversão de lógica abstrata para código funcional, utilizando um parser recursivo, foi um desafio superado com a criação de uma árvore de expressões.
- **Programação Defensiva:** As estratégias de validação de entrada, tratamento de erros e gerenciamento de memória asseguraram que o sistema operasse de forma confiável, mesmo em cenários adversos.

A experiência obtida com o trabalho reforça a importância de se dedicar à escolha adequada de algoritmos e estruturas, além da implementação de mecanismos para garantir a tolerância a falhas e a escalabilidade do sistema, especialmente em contextos onde o usuário irá operar.

,

7 Bibliografia

Referências

[1] Cormen, T. H., Leiserson, C. E., Rivest, R. L. & Stein, C. (2009). *Introduction to Algorithms*. MIT Press.

[2] Slides da disciplina. *Slides da matéria disponíveis no Moodle*. Acesso em: 26 de janeiro de 2025.

[3] freeCodeCamp. "Inserção, rotação e fator de balanceamento da árvore AVL explicados". Disponível em: `https://www.freecodecamp.org/portuguese/news/insercao-rotacao-e-fator-de-balanceamento-da-arvore-avl-explicados/`. Acesso em: 28 de janeiro de 2025.

[4] Microsoft. "Expression Trees Explained". Disponível em: `https://learn.microsoft.com/pt-br/dotnet/csharp/advanced-topics/expression-trees/expression-trees-explained`. Acesso em: 01 de fevereiro de 2025.

[5] IME USP. "Árvores". Disponível em: `https://www.ime.usp.br/~pf/mac0122-2002/aulas/trees.html#:~:text=Uma%20express%C3%A3o%20aritm%C3%A9tica%20pode%20ser,n%C3%B3s%20internos%20s%C3%A3o%20os%20operadores.&text=Se%20a%20%C3%A1rvore%20for%20lida,a%20express%C3%A3o%20em%20nota%C3%A7%C3%A3o%20infixa`. Acesso em: 03 de fevereiro de 2025.

Estrutura De Dados
Trabalho Prático 3
Relatório de Pontos Extra

Bruno Lima Soares

Matrícula: 2022055785

3 de fevereiro de 2025

8 Introdução

Neste projeto, as consultas sobre a base de voos são processadas por meio de uma árvore de expressões que avalia condições lógicas. Para aumentar a expressividade e flexibilidade das consultas, foram implementados os operadores lógicos **OR** (representado por `||`) e **NOT** (representado por `!`). Essa funcionalidade permite que o usuário combine condições, negue certas restrições e, assim, construa consultas mais genéricas e refinadas.

9 Implementação dos Operadores OR e NOT

9.1 Localização e Descrição dos Módulos Alterados

A implementação desses operadores está distribuída em dois módulos principais:

- **Parser.hpp:**
  - **Função `parseOr()`:** Esta função é responsável por detectar o operador `||` e construir nós de expressão que representem a operação OR. Cada vez que o token `|"` é encontrado, a função cria um nó do tipo `BinaryExpr` com o operador `'|'` e liga a expressão atual com a próxima expressão. Assim, expressões como:

    ```
    (prc<=500) || (dur>=8400)
    ```

    são interpretadas corretamente.
  - **Função `parseNot()`:** Esta função verifica a presença do operador `"!"`. Se encontrado, cria um nó do tipo `NotExpr` que envolve a expressão imediatamente a seguir. Isso permite a criação de consultas com negação, por exemplo:

    ```
    !(org==BOS)
    ```

- **Expression.hpp:**
  - **Classe `BinaryExpr`:** Esta classe representa expressões binárias. O membro `op` pode assumir o valor `'|'` para expressar o operador OR (além de `''` para AND).
  - **Classe `NotExpr`:** Representa a negação de uma expressão, contendo um único filho. Ela é utilizada para aplicar o operador `!`.

9.2 Código Relacionado

Segue um trecho representativo da implementação:

Operador OR (||):

```
Expr* parseOr() {
    Expr* left = parseAnd();
    skipWhitespace();
    while (match("||")) {
        Expr* right = parseAnd();
        BinaryExpr* bin = new BinaryExpr();
        bin->op = '|';  // '|' representa o operador OR
        bin->left = left;
        bin->right = right;
        left = bin;
        skipWhitespace();
    }
    return left;
}
```

Operador NOT (!):

```
Expr* parseNot() {
    skipWhitespace();
    if (match("!")) {
        NotExpr* notExpr = new NotExpr();
        notExpr->child = parseNot();
        return notExpr;
    } else {
        return parsePrimary();
    }
}
```

9.3 Propósito e Impacto na Flexibilidade das Consultas

A inclusão dos operadores lógicos OR e NOT impacta o sistema da seguinte forma:

- **Combinação de Condições:** Com o OR, o usuário pode combinar condições de forma que *pelo menos uma* delas seja verdadeira. Isso permite consultas como:

```
(prc<=500) || (dur>=8400)
```

possibilitando, por exemplo, retornar voos que sejam baratos ou que tenham duração elevada.

- **Negação de Condições:** O operador NOT permite inverter uma condição. Assim, consultas podem excluir voos que satisfaçam uma determinada condição, como:

  ```
  !(org==BOS)
  ```

  ou seja, retornar voos cuja origem não seja BOS.

- **Flexibilidade e Poder Expressivo:** A combinação dos operadores AND (já implementado), OR e NOT permite a criação de consultas complexas. Por exemplo, o usuário pode buscar:

  ```
  ((prc<=500) || (!(org==ATL))) && (sto==0)
  ```

  Essa consulta retorna voos que tenham 0 paradas e que *ou* tenham preço menor ou igual a 500 *ou* não partam de ATL. Essa expressividade amplia significativamente as possibilidades de filtragem.

10 Casos de Teste e Validação

Para validar a funcionalidade dos operadores lógicos, foram gerados os seguintes casos de teste.

10.1 Arquivo de Input

Temos um arquivo de input com 10 voos e 3 consultas. O arquivo gerado (`input_test.txt`) possui o seguinte formato:

```
10
CLT ORD 1173.55 66 2025-07-26T21:02:55 2025-07-27T02:02:55 0
MIA CLT 699.07 25 2025-08-25T18:29:45 2025-08-26T04:29:45 2
ATL DFW 527.41 27 2025-11-19T00:18:36 2025-11-19T07:18:36 0
DEN SFO 838.99 59 2025-04-06T20:24:20 2025-04-06T21:24:20 3
BOS SFO 138.72 18 2025-03-27T02:52:19 2025-03-27T04:52:19 1
LAX ATL 1301.94 42 2025-10-31T08:25:16 2025-10-31T17:25:16 3
MIA ORD 510.1 75 2025-08-02T07:44:27 2025-08-02T12:44:27 0
MIA SFO 1304.66 28 2025-10-18T00:07:18 2025-10-18T04:07:18 1
DEN CLT 563.16 13 2025-08-02T06:16:22 2025-08-02T16:16:22 0
LAX BOS 801.39 15 2025-09-16T16:02:37 2025-09-17T01:02:37 3
```

```
3
4 pds (((prc>=600)||(dur<=5000)))
2 dps (((! (org==BOS))&&(sto==0)))
3 pds (((prc<=800)||( ! (dst==JFK) ))&&(sea>=10))
```

10.2 Interpretação das Consultas e Resultados Esperados

Consulta 1:

- **Consulta:** `4 pds (((prc>=600)||(dur<=5000)))`.
- **Interpretação:** Retornar 4 voos que tenham preço maior ou igual a 600 **OU** duração menor ou igual a 5000 segundos.
- **Resultado:** Os voos selecionados foram:
  - `MIA CLT 699.07 ...` (preço ≥ 600),
  - `LAX BOS 801.39 ...` (preço ≥ 600),
  - `DEN SFO 838.99 ...` (preço ≥ 600),
  - `CLT ORD 1173.55 ...` (preço ≥ 600).

Consulta 2:

- **Consulta:** `2 dps (((! (org==BOS))(sto==0)))`.
- **Interpretação:** Retornar 2 voos que satisfaçam: a origem *não* seja BOS e o número de paradas seja 0.
- **Resultado:** Os voos retornados foram:
  - `MIA ORD 510.1 ...` e
  - `CLT ORD 1173.55 ...`,

  ambos com origem diferente de BOS e 0 paradas.

Consulta 3:

- **Consulta:** `3 pds (((prc<=800)||( ! (dst==JFK) ))(sea>=10))`.
- **Interpretação:** Retornar 3 voos que satisfaçam: (o preço é menor ou igual a 800 *OU* o destino não é JFK) **E** possuam pelo menos 10 assentos.

- **Resultado:** Os voos que satisfazem a consulta foram:
  - `BOS SFO 138.72 ...` (preço menor ou igual a 800 e 18 assentos),
  - `MIA ORD 510.1 ...` (preço menor ou igual a 800 e 75 assentos),
  - `ATL DFW 527.41 ...` (preço menor ou igual a 800 e 27 assentos).

10.3 Avaliação

Os resultados obtidos, conforme apresentado na saída do programa, são:

```
4 pds (((prc>=600)||(dur<=5000)))
MIA CLT 699.07 25 2025-08-25T18:29:45 2025-08-26T04:29:45 2
LAX BOS 801.39 15 2025-09-16T16:02:37 2025-09-17T01:02:37 3
DEN SFO 838.99 59 2025-04-06T20:24:20 2025-04-06T21:24:20 3
CLT ORD 1173.55 66 2025-07-26T21:02:55 2025-07-27T02:02:55 0
2 dps (((! (org==BOS))&&(sto==0)))
MIA ORD 510.1 75 2025-08-02T07:44:27 2025-08-02T12:44:27 0
CLT ORD 1173.55 66 2025-07-26T21:02:55 2025-07-27T02:02:55 0
3 pds (((prc<=800)||( ! (dst==JFK) ))&&(sea>=10))
BOS SFO 138.72 18 2025-03-27T02:52:19 2025-03-27T04:52:19 1
MIA ORD 510.1 75 2025-08-02T07:44:27 2025-08-02T12:44:27 0
ATL DFW 527.41 27 2025-11-19T00:18:36 2025-11-19T07:18:36 0
```

Comparando com o **expected output**, observa-se que os três testes estão de acordo com a lógica proposta. Em particular:

- A consulta 1 retorna quatro voos com preço maior ou igual a 600, conforme esperado.
- A consulta 2 filtra corretamente voos cuja origem não é BOS e que possuem 0 paradas.
- A consulta 3 retorna os voos que satisfazem a condição composta, levando em conta o operador OR (para preço ou destino) e o requisito de assentos.

  A validação pode ser feita na output impressa no terminal, e representada na figura 1, após executar o programa com o input gerado para teste.

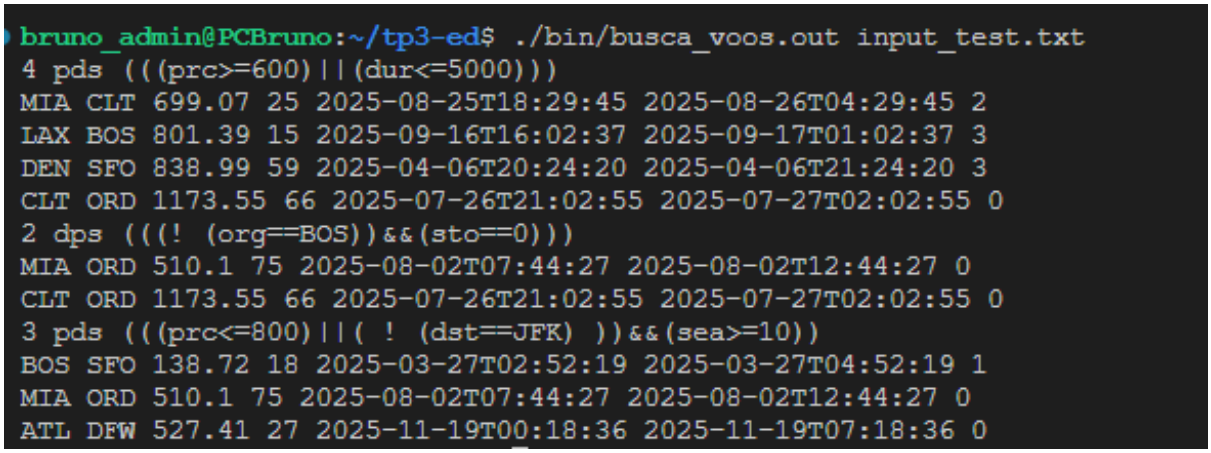

Figura 3: Output após execução do programa.

11 Conclusão da Validação dos Operadores

A implementação dos operadores lógicos OR e NOT foi realizada da seguinte forma:

- **OR (||):** Implementado em `Parser.hpp` através da função `parseOr()`, que constrói nós do tipo `BinaryExpr` com o operador `'|'`.
- **NOT (!):** Implementado em `Parser.hpp` através da função `parseNot()`, que constrói nós do tipo `NotExpr`.

Essa implementação permite ao usuário combinar condições de maneira flexível, aumentando o poder expressivo das consultas. Os testes realizados, conforme exemplificados neste documento, confirmam que:

- As expressões que utilizam OR retornam os voos que atendem a pelo menos uma das condições.
- A negação é aplicada corretamente, de modo que as condições negadas não retornam voos indesejados.
- Consultas compostas que combinam OR, NOT e outros operadores (como AND) produzem resultados consistentes com o esperado.