Trabalho Prático da disciplina de Estrutura de Dados

Leticia Ribeiro Miranda
Departamento de Ciência da Computação – Universidade Federal de Minas Gerais (UFMG)
Belo Horizonte – MG – Brazil
leticia-ribeiro98@hotmail.com

1. Introdução

O trabalho tem como objetivo a implementação de um sistema eficiente e flexível para buscar passagens aéreas. Com esse objetivo, o programa deve importar os dados das passagens a partir de um arquivo no formato .txt ou .csv, ler as consultas registradas e gerar a saída de acordo com os filtros e critérios de ordenação definidos na busca.

Com o propósito de resolver o problema citado, optou-se por criar nove classes, *'NoFloat', 'ArvoreAVLFloat'*, *'NoInt'*, *'ArvoreAVLInt'*, *'NoString'*, *'ArvoreAVLString'*, *'DadosViagem'*, *'HeapSort' e 'SistemaBusca'*, que realizaram toda a implementação da estrutura de dados necessária para o processamento eficiente das buscas e ordenações. As classes de árvores AVL (*ArvoreAVLFloat*, *ArvoreAVLInt*, *ArvoreAVLString*) garantiram a organização e recuperação dos dados de acordo com as buscas, enquanto as classes de 'No' representaram os nós para cada tipo de dado. A classe 'HeapSort' foi responsável pela ordenação das consultas conforme os critérios de preço (p), duração (d) e número de paradas (s), garantindo que os resultados fossem apresentados de acordo com as preferências do usuário. Por fim, a classe 'SistemaBusca' integrou todas as funcionalidades, coordenando a interação entre as estruturas de dados.

Por último, foi feita a análise experimental, que inclui medições de tempo de relógio e recurso e utilização de ferramentas do CacheGrind e do Valgrind.

2. Método

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

- Sistema operacional: Ubuntu 24.04.1 LTS
- Compilador: gcc (Ubuntu 13.2.0-23 ubuntu 4) 13.2.0
- Processador: AMD Ryzen 5 3500U with Radeon Vega Mobile Gfx 2.10 GHz
- RAM: 16Gb (utilizável: 13,9 GB)

2.1. Estrutura de Dados

Para resolver o problema proposto, a implementação foi estruturada em seis arquivos de cabeçalho: *'ArvoreAVLFloat.hpp'*, '*ArvoreAVLInt.hpp'*, '*ArvoreAVLString.hpp'*, *'DadosViagem.hpp'*, *'HeapSort.hpp'* e *'SistemaBusca.hpp'*, os quais contêm a definição das classes, incluindo seus atributos e métodos. Em cada um desses arquivos, a primeira parte é dedicada à definição dos atributos privados das classes, enquanto a segunda parte descreve os construtores e os métodos públicos. Além disso, o programa conta com sete arquivos de implementação: *'ArvoreAVLFloat.cpp'*, '*ArvoreAVLInt.cpp'*, '*ArvoreAVLString.cpp'*, *'DadosViagem.cpp'*, *'HeapSort.cpp'* e *'SistemaBusca.cpp'*, que

contém o desenvolvimento do comportamento das funções e métodos definidos nas classes correspondentes.

As três árvores de balanceamento (AVL) implementadas nas classes *'NoFloat', 'ArvoreAVLFloat'*, *'NoInt', 'ArvoreAVLInt'*, *'NoString'* e *'ArvoreAVLString'* são variações da mesma estrutura de dados, diferenciando-se apenas pelo tipo da chave armazenada:

- ArvoreAVLFloat: Usa float como chave, para armazenar o preço.
- ArvoreAVLInt: Usa int como chave, para a quantidade de assentos disponíveis, a duração dos voos e o número de conexões.
- ArvoreAVLString: Usa std::string como chave, para armazenar a origem e o destino do voo, a data e hora da chegada e partida de cada viagem.

As classes *'NoFloat'*, *'NoInt'* e *'NoString'* servem para armazenar as chaves, ou seja, os valores que identificam os voos de acordo com um critério e mantêm referências para os voos correspondentes.

Já as classes das árvores AVL servem para armazenar e organizar os dados de voos, permitindo buscas baseadas em diferentes parâmetros. Além disso, elas indexam os voos por um atributo específico e mantêm um equilíbrio automático após a inserção. Cada nó contém um array dinâmico de índices, responsável por armazenar os IDs dos voos que possuem a mesma chave. Dessa forma, um único nó pode armazenar múltiplos índices associados à ele, possibilitando a busca de voos com características semelhantes.

Por fim, os métodos *'buscarInOrdemMaior'*, *'buscarInOrdemMenor'* e *'buscarInOrdemIgual'* das *'ArvoreAVLFloat'* e *'ArvoreInt'* permitem retornar um array de índices correspondentes aos valores maior ou igual, menor ou igual e igual ao valor buscado, respectivamente. Já o método *'buscarInOrdem'* da classe *'ArvoreAVLString'* retorna os índices cujos valores são exatamente iguais aos solicitados nos filtros.

A classe *'DadosViagem'* representa um voo e armazena todas as suas informações. Ela inclui um método chamado *'calcularDuracaoSegundos'* que calcula a duração do voo em segundos e converte datas e horários ao formato adequado de manipulação. Além disso, a classe contém métodos para acessar dados privados.

Já a classe *'HeapSort'* implementa uma fila de prioridade baseada em um heap binário para ordenar objetos da classe *DadosViagem* segundo critérios de preço (p), duração (d) e paradas (s). Ela armazena e ordena voos buscados usando um minheap, ou seja, o voo com menor valor conforme os critérios de ordenação escolhidos será sempre o primeiro na estrutura. Usando os métodos *'heapifyUp'* e *'heapifyDown'*, o heap se mantém balanceado automaticamente após inserções e remoções, enquanto o método *'comparar'* determina qual voo tem maior prioridade.

A classe *'SistemaBusca'* gerencia a busca e a ordenação de voos armazenados no sistema. Ela permite buscar, filtrar e ordenar voos com base em diferentes atributos, usando árvores AVL para indexação e um minheap para ordenação dos resultados.

A parte central da classe *'SistemaBusca'* envolve a busca pela interseção de voos que atendem a múltiplos critérios de filtragem. Inicialmente, cada árvore AVL no sistema armazena os voos indexados por um atributo específico. Quando uma consulta é realizada, o programa verifica, para cada parâmetro fornecido, a árvore correspondente e extrai os

índices dos voos que atendem ao filtro. Essa operação é repetida para todos os critérios especificados na busca.

O ponto chave da interseção é quando o sistema compara os índices dos voos que atendem a cada filtro e encontra os voos que estão presentes em todos os filtros aplicados. Isso é feito comparando os índices das buscas realizadas nas diferentes árvores AVL e selecionando apenas aqueles que aparecem em todas as consultas. Após identificar os voos que satisfazem todos os filtros, eles são inseridos em um heap para ordenação. Por fim, os voos podem ser exibidos de forma organizada e formatada para o usuário.

O programa principal, 'main.cpp', utiliza as definições e funcionalidades contidas nas classes supracitadas. O algoritmo começa verificando se foi passado um argumento de entrada na linha de comando. Após isso, ele lê o número de voos que será processado no arquivo e cria um objeto sistema da classe *'SistemaBusca'*, recebendo a quantidade de voos lida no arquivo. Em seguida, o programa entra em um laço onde ele lê os dados dos voos do arquivo, cria um objeto *'DadosViagem'* com os dados lidos e adiciona ao *'SistemaBusca'*. O índice do voo é passado como parâmetro para associar corretamente o voo no sistema de busca.

O programa então lê o número de consultas que serão realizadas, a quantidade máxima de resultados a ser retornada e como será a prioridade de ordenação. A consulta completa é lida como uma string e então processada pelo método *'satisfazConsulta'* da classe *'SistemaBusca'*, que busca os voos que atendem aos critérios da consulta, ordenando-os de acordo com as prioridades. Por fim, para cada consulta, o programa imprime os parâmetros da busca juntamente com os voos correspondentes.

3. Análise de Complexidade

A análise de complexidade foi feita individualmente em cada classe, e, em seguida, foi analisada a interação entre elas no programa principal, determinando o impacto combinado das suas operações.

- **'NoFloat', 'ArvoreAVLFloat', 'NoInt', 'ArvoreAVLInt', 'NoString' e 'ArvoreAVLString'**

Como as árvores são variações da mesma estrutura de dados, diferenciando-se apenas pelo tipo da chave armazenada, vamos analisá-las de maneira generalizadas

As operações de inserção, remoção, e busca têm a complexidade de tempo O(log n), onde n é o número de nós na árvore. Isso se deve ao fato de que a árvore AVL é balanceada, o que garante que a altura da árvore nunca será maior que log n.

A inserção de um nó envolve a navegação até a posição correta para ele, o que leva O(log n), pois a árvore é balanceada e, em seguida, o balanceamento da árvore, que também ocorre em O(log n). No pior caso, pode ser necessário realizar uma rotação para balanceá-la, mas como há no máximo 2 rotações em qualquer inserção, o custo de tempo permanece O(log n).

A busca por uma chave específica, ou por chaves maiores, menores ou iguais, também ocorre em O(log n), já que, em uma árvore balanceada, a profundidade máxima será log n. Já o balanceamento da árvore após uma inserção ou remoção envolve no máximo 2

rotações, que têm custo O(1) cada. Como as rotações ocorrem durante a navegação nela, elas não afetam a complexidade geral, que permanece O(log n).

A complexidade de espaço da árvore AVL depende de dois fatores principais: a quantidade de memória usada pelos nós e o espaço necessário para armazenar os índices relacionados às chaves.

Cada nó da árvore AVL contém a chave que tem complexidade de espaço O(1), o ponteiro para os filhos ocupando O(2) do espaço e a altura do nó que é igual a O(1).

No caso das classes do No, existe também o array de índices dos voos que são armazenados dentro de cada nó. Para ele, o espaço necessário é O(m), onde m é o número de índices armazenados por nó. Portanto, a complexidade de espaço para cada nó é O(m).

Para n nós na árvore, o espaço total ocupado pela árvore é O(m*n), onde m é a quantidade de índices por nó e n é o número total de deles.

- **DadosViagem**

A complexidade de tempo de todas as operações, incluindo os construtores, os métodos auxiliares e os getters, é O(1), já que cada operação realizada é constante e não depende do tamanho da entrada. A complexidade de espaço por instância de DadosViagem é O(1), visto que a quantidade de memória alocada para os atributos não varia com a entrada e é fixa para cada objeto.

- **HeapSort**

O construtor e o método ‘inicializa()’ apenas inicializam o tamanho como zero, ambos têm complexidade de tempo O(1). A estrutura do heap já está alocada e ocupa espaço O(n), onde n é o número de elementos no heap.

Para inserir um voo ou retirá-lo, o HeapSort simula uma árvore como um vetor, isso garante que a profundidade das folhas irá diferir em no máximo um. Logo, a árvore estará sempre balanceada e a complexidade de tempo será O(log n), sendo ‘n’ o número de voos inseridos no momento. Além disso, nenhuma alocação adicional é feita, então o espaço é O(1).

Portanto, a complexidade de tempo para a classe ‘HeapSort’ é O(log n) e a complexidade de espaço é igual a O(n), devido à estrutura do heap.

- **SistemaBusca**

Inicialmente, o sistema de busca faz inserções e buscas nas árvores balanceadas, como citado anteriormente, essas operações têm complexidade de tempo igual a O(log n) e cada árvore tem complexidade de espaço igual O(n), onde n é a quantidade total de voos.

Os vetores utilizados na classe tem complexidade de tempo de acesso O(1), enquanto a complexidade de espaço é igual a O(n), também.

Após realizar as consultas, os resultados são inseridos no heap, que tem complexidade de tempo O(m log m) e de espaço igual a O(m), onde m é o número de voos encontrados. Como a quantidade de voos encontrados não será maior que o número total de voos, a

complexidade total de espaço será O(n). Além disso, a complexidade de tempo O(m log m) domina O(log n) quando m é de um tamanho significativo em comparação a n.

Portanto a complexidade de tempo total será O(m log m) e de espaço O(n), onde m é a quantidade de voos encontrados e n a quantidade total de voos.

- **Main**

O programa lê *'n'* linhas do arquivo (uma por voo). As informações das linhas são processadas individualmente, logo a complexidade de tempo é O(n).

Para cada voo, é feita a inserção nas árvores AVL, que tem complexidade O(log n), onde n é o número de elementos na árvore. Como estamos inserindo n voos, a complexidade total de inserção será O(n log n).

O código lê a quantidade de buscas e para cada uma delas, chama *'satisfazConsulta'* com a consulta, a ordem e o número máximo de resultados. A leitura da busca e processamento das variáveis tem complexidade de tempo O(1) por consulta.

No entanto, o custo real de tempo vem da função *'satisfazConsulta*'. Dentro dela, as buscas nas árvores AVL são feitas, então a complexidade tempo de cada uma será O(log n), como temos q quantidade de consultas, então o custo total será O(q * log n). O código chama *'HeapSort'*, que possui uma complexidade de tempo O(m log m), onde m é o número de resultados encontrados na consulta. O custo de inserção dos resultados no heap e a ordenação são O(m log m).

A complexidade total de tempo será, portanto, a soma de todas as partes: leitura dos voos - O(n), inserção nas árvores AVL - O(n log n), processamento das buscas - O(q log n) e HeapSort - O(m log m).

A complexidade de tempo será dominado por O(n log n) e O(m log m), onde m pode ser no máximo igual a n. Assim, a complexidade total de tempo é O(n log n + m log m), ou simplificadamente, O(n log n), considerando que m ≤ n.

Já para a complexidade de espaço, será considerado: o espaço utilizado pelas árvores AVL que é igual a O(n), pois cada voo é armazenado nas árvores, o espaço necessário para o heap que é O(m), onde m é o número de voos encontrados em uma consulta, e o armazenamento das variáveis temporárias, que ocupa O(1) por voo ou por consulta.

Portanto, o espaço total será dominado pelas árvores AVL e pelo heap, com complexidade de O(n) para as árvores AVL e O(m) para o heap. Como m ≤ n, a complexidade total de espaço será O(n).

4. Estratégias de Robustez

No desenvolvimento do código, foram implementadas estratégias de robustez para garantir que o programa lidasse com possíveis erros e condições inesperadas, são elas:

- Verificação de abertura de Arquivo: No main, o código verifica se o arquivo foi aberto corretamente, caso contrário, exibe uma mensagem de erro e interrompe a execução.

- Verificação de Limite: O método *'adicionarIndice'* garante que os índices só sejam adicionados enquanto houver espaço disponível no array, evitando um acesso indevido ao array.
- Busca com Limitação de Resultados: As funções de busca (*'buscarInOrdemMaior', 'buscarInOrdemMenor', 'buscarInOrdemIgual', 'buscarInOrdem'*) utilizam um array estático para armazenar os índices encontrados durante a consulta. Um contador é utilizado, garantindo que não sejam adicionados mais resultados do que o máximo permitido. Isso ajuda a evitar estouros de memória e garante que as buscas sejam feitas de forma controlada.
- Verificação de Resultado: Caso nenhum índice seja encontrado, a função imprime uma mensagem indicando que o índice não foi encontrado.
- Destruição dos Nós: O destruidor da árvore AVL é implementado recursivamente, garantindo que toda a memória alocada pelos nós e pelo array de índices seja corretamente liberada.
- Acesso Seguro aos Atributos: A classe *'DadosViagem'* utiliza métodos getter para acessar os atributos privados, garantindo que os dados da viagem possam ser acessados de forma controlada e não sejam alterados inadvertidamente.
- Alocação Segura de Memória: A alocação de memória para o heap é feita dinamicamente, permitindo que o número de elementos no heap seja flexível, de acordo com o valor de capacidade. No destruidor da classe *'HeapSort'*, a memória é liberada corretamente, evitando vazamentos.
- Proteção Contra Acesso Fora dos Limites: O código não permite inserções quando o número de elementos excede a capacidade do heap, evitando acesso a áreas de memória inválidas e problemas relacionados à alocação de memória excessiva.
- Evita Erros de Precisão nas Comparações de Preço: Quando a comparação de preços é realizada, há uma tolerância para erros de precisão de ponto flutuante, garantindo que pequenas imprecisões nos valores de ponto flutuante não resultem em comparações erradas.

No geral, essas práticas ajudaram a garantir que o código fosse mais coeso, livre de erros e eficiente.

5. Análise Experimental

Na etapa de análise experimental foram utilizadas três bibliotecas para medir o tempo de relógio e recursos do programa.

Para medir o tempo de relógio, foi usada a função '*high_resolution_clock::now()'* da biblioteca chrono, que registra o tempo inicial e o tempo final ao redor da execução da função alvo. A diferença entre esses dois instantes é calculada e armazenada em uma variável do tipo '*duration<double, std::milli>'*, que converte a duração para milissegundos.

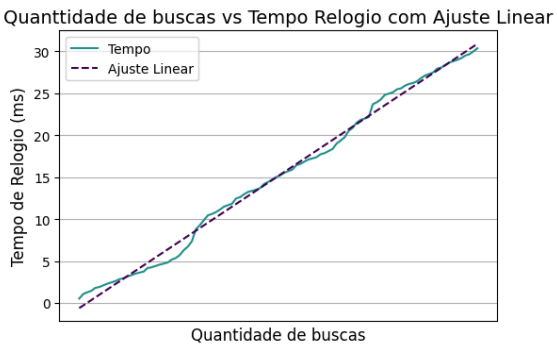
Já para os tempos de utilização de recursos (usuário e sistema), usou-se a função clock_gettime que pertence à biblioteca padrão do sistema no Linux, '*time.h'*. Calculou-se a diferença entre o início e fim da execução da função alvo para obter as durações.

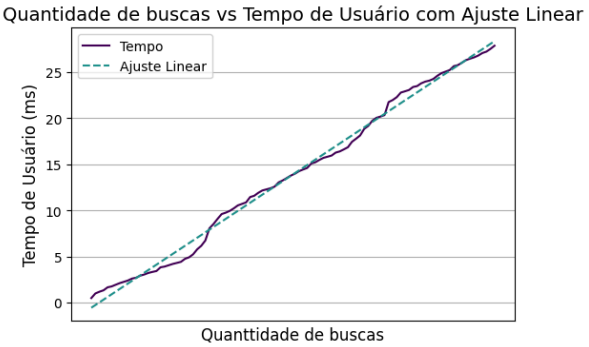
Além disso, foram utilizadas as ferramentas Cachegrind, que simula a hierarquia de memória e mede o desempenho de cache, e o Valgrind, que analisa as funções mais chamadas.

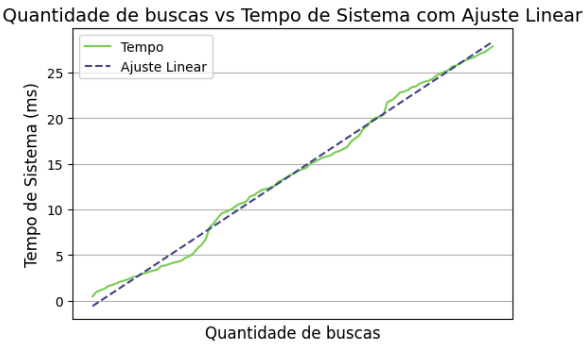
Por fim, em todas as análises, utilizaram-se uma base de dados com 1000 registros, totalizando 58.385 mil caracteres e 100 tipos de consultas distintas. Após esse limite, o programa apresentou um erro de segmentation fault, o que restringiu a quantidade de dados analisados.

Nos gráficos abaixo, observa-se uma tendência linear entre a quantidade de iterações e o tempo. À medida que o número de pacientes aumenta, consequentemente o número de iterações que o programa realiza também aumenta e o tempo de relógio, de usuário e de sistema cresce linearmente.

Com base na regressão linear, pode-se concluir que, para cada incremento de 1 unidade no eixo x, o tempo aumenta, em média, 0.3182 milissegundos (ms) para o de relógio e 0.2920 milissegundos (ms) para de usuário e de sistema. Isso sugere uma relação direta e previsível entre a quantidade de buscas e o tempo total registrado, embora o gráfico também revele pequenas flutuações ao redor da reta de ajuste.







Porém, essa análise não reflete a complexidade de tempo do programa, que é O(n log n), isso ocorre porque a curva logarítmica é muito semelhante à curva linear quando o número de dados é pequeno. Portanto, no gráfico gerado, o comportamento acaba se parecendo mais com uma curva linear.

Executando o código utilizando o Cachegrind, especificamente com a ferramenta cg_annotate, foram analisadas as principais funções que impactam no total de instruções executadas.

O programa executou o total de 82.330.750 instruções, sendo que 8.660.178 (10,52%) delas vêm dos arquivos referentes à implementação das árvores AVL, o que já era esperado, visto que ela é o fator principal da complexidade do tempo do algoritmo.

Dentro da classe *'ArvoreAVLFloat'* e *'ArvoreAVLInt'*, o método *'buscarInOrdemMaior'* é um dos maiores responsáveis pelo consumo de instruções, representando 17,39% (1.505.916) do total desse valor. Já o *'inserir'*, da classe *'ArvoreAVLStrinG'* também é bem significativo, consumindo 19,29% (1.671.330) de instruções. Vale destaque também para a classe *'DadosViagem'*, que ficou encarregado por 8,1% (6.667.863) das instruções totais.

A análise indica que as operações referente às árvores AVL são responsáveis por uma porção considerável do programa.

Por fim, através do comando *'kcachegrind callgrind.out.<pid>'*, analisou-se as funções mais chamadas e quanto tempo de CPU foi consumido por cada uma delas.

Na figura 1, observa-se que o método *'satisfazConsulta()'* consome 43,49% do tempo total de CPU. Isso ocorre porque ele é o componente central do programa, responsável pelas buscas nas árvores balanceadas, inserções e remoções no heap, além da exibição dos dados filtrados. Na segunda e terceira figura, percebe-se como os métodos do heap e das árvores consomem um cache significativo.

Na Figura 2, observa-se que o método *'inserirNoHeap'* consome 34,79% do tempo total de CPU. Já na Figura 3, o método *'adicionarVoo'*, responsável por acionar as oito árvores de balanceamento, representa 21,65% do tempo total de CPU.

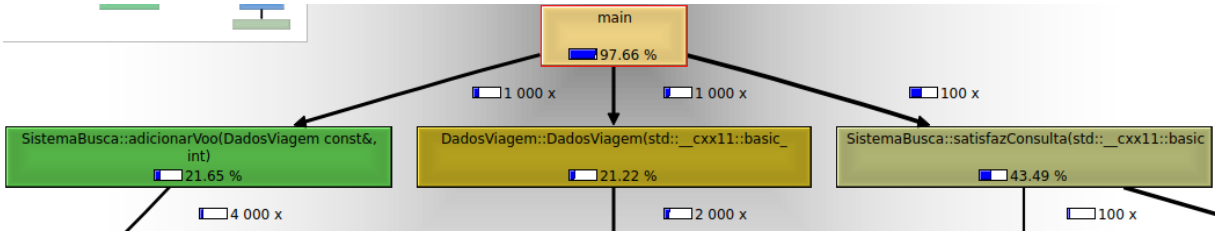


Figura 1: representa os tempos de execução, retirada do kcachegrind

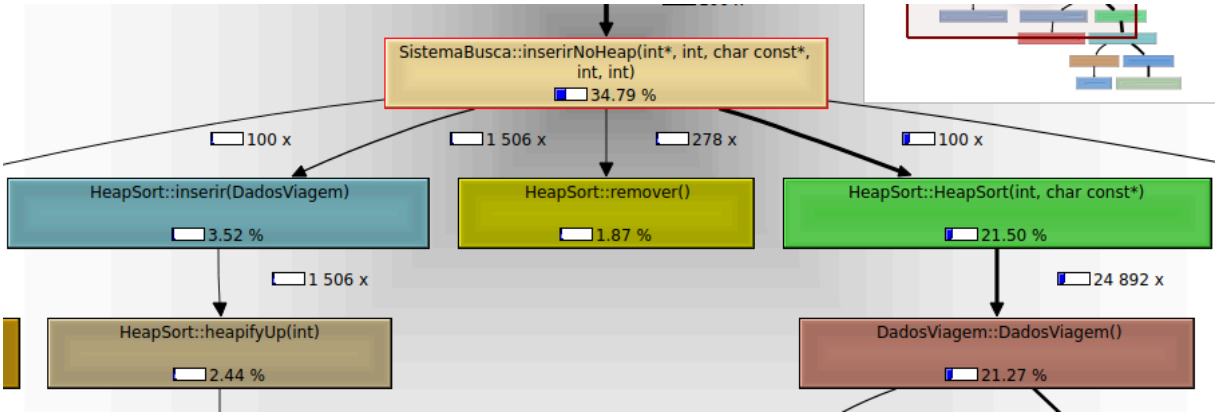


Figura 2: representa os tempos de execução, retirada do kcachegrind

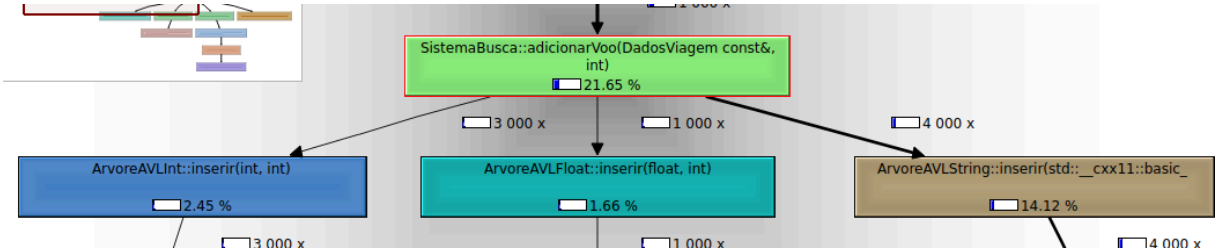


Figura 3: representa os tempos de execução, retirada do kcachegrind

Esse resultado já era esperado, pois, como destacado anteriormente, as árvores de balanceamento e a classe *'SistemaBusca'* desempenham um papel central na implementação de um sistema para buscar passagens aéreas.

6. Conclusão

Este trabalho lidou com o problema de implementação de um sistema eficiente e flexível para buscar passagens aéreas, na qual a abordagem utilizada na solução foi desenvolver oito árvores de balanceamento, um por critério de consulta, que servem para armazenar e organizar os dados de voos, permitindo buscas baseadas em diferentes filtros. Além disso, os dados foram organizados através de um heap binário, cujo a ordenação era feita com base nas prioridades solicitadas nas buscas. Também foi criado uma classe para armazenar os dados de cada viagem e uma classe central, que gerenciava e mostrava as buscas dos voos

Com a solução adotada, pode-se verificar que o sistema apresentou maior eficiência nas consultas dos voos. As árvores balanceadas e o método de ordenação baseado em um heap binário permitiu que as consultas fossem processadas de forma eficiente, garantindo que o resultado fosse de acordo com o solicitado. A divisão do programa em várias classes possibilitou uma estrutura de código organizada, facilitando a implementação de melhorias e permitindo a identificação e correção de problemas rapidamente.

Por meio da resolução deste trabalho, foi possível praticar os conceitos relacionados à árvores balanceadas e heap binário. Durante a implementação da solução houveram importantes desafios a serem superados, a principal delas foi selecionar os voos que atendesse a todos os filtros, ou seja, a interseção das buscas.

Este estudo reforça a importância de estruturar um código eficiente e bem organizado que garanta o funcionamento otimizado e confiável em um sistema de buscas.

7. Bibliografia

Wagner Meira Jr. (2023). Slides virtuais da disciplina de estruturas de dados. Disponibilizado via Moodle. Departamento de Ciência da Computação. Universidade Federal de Minas Gerais. Belo Horizonte.

8. Extra

O objetivo desse tópico é aumentar a flexibilidade e poder expressivo das consultas. Para isso, o programa aceitará consultas mais genéricas, permitindo os operadores lógicos de "ou" ou "e".

8.1. Implementação

Na classe 'SistemaBusca', o método 'inserirNoHeapUniao' foi implementado para verificar se pelo menos um dos filtros é atendido, em vez de exigir a interseção entre eles, ou seja, nem todos os filtros precisam ser satisfeitos simultaneamente, garantindo que aceitem consultas com operadores lógicos "ou". Esse método pode ser chamado dentro de *'SatisfazConsulta'*.

8.2. Análise de Complexidade

Em relação à complexidade de tempo e espaço, não houve alterações, pois o método não modifica a estrutura de dados do código, apenas deixa de verificar a interseção. Dessa forma, a complexidade temporal do programa continua sendo O(n log n) e a espacial O(n).

8.3. Exemplo de uso

Abaixo, segue um exemplo de implementação. Na busca, solicitou 9 resultados ordenados por preço, quantidade de paradas e duração. A seleção aplicada foi: o destino igual a DEN e quantidade de assentos disponíveis igual a zero.

Na primeira busca, aparece somente a interseção dos resultados: percebe-se que o resultado contém todas viagens com destino a DEN e com a quantidade de assentos igual a zero.

```
9 psd (((dst==DEN)&&(sea==0)))
Interseção
LAX DEN 100.58 0 2022-07-14T19:41:00 2022-07-15T02:47:00 1
OAK DEN 100.58 0 2022-07-25T19:00:00 2022-07-26T02:47:00 1
OAK DEN 113.58 0 2022-09-20T17:05:00 2022-09-20T20:55:00 1
LAX DEN 124.78 0 2022-05-19T04:30:00 2022-05-19T17:14:00 1
BOS DEN 125.78 0 2022-04-26T13:53:00 2022-04-27T04:35:00 1
OAK DEN 150.58 0 2022-11-14T13:10:00 2022-11-14T18:35:00 1
DTW DEN 161.58 0 2022-05-26T17:40:00 2022-05-27T00:37:00 1
EWR DEN 322.58 0 2022-04-17T18:17:00 2022-04-18T03:51:00 1
BOS DEN 382.58 0 2022-04-19T09:04:00 2022-04-19T19:58:00 1
```

Na segunda busca, com a implementação da conjunção 'ou', os resultados incluem tanto viagens com destino a DEN, quanto aquelas para outros destinos, desde que a quantidade de assentos seja zero. Além disso, podem ser exibidas viagens sem assentos disponíveis ou com assentos restantes, desde que tenham como destino DEN.

```
União
LAX DEN 100.58 0 2022-07-14T19:41:00 2022-07-15T02:47:00 1
OAK DEN 100.58 0 2022-07-25T19:00:00 2022-07-26T02:47:00 1
DTW DEN 109.59 4 2022-11-07T01:05:00 2022-11-07T15:50:00 1
EWR ATL 112.96 0 2022-05-23T13:42:00 2022-05-24T00:09:00 1
OAK DEN 113.58 0 2022-09-20T17:05:00 2022-09-20T20:55:00 1
DFW LAX 118.59 0 2022-07-06T19:36:00 2022-07-06T22:53:00 0
LAX DEN 124.78 0 2022-05-19T04:30:00 2022-05-19T17:14:00 1
ATL ORD 125.59 0 2022-09-08T20:35:00 2022-09-08T22:41:00 0
BOS DEN 125.78 0 2022-04-26T13:53:00 2022-04-27T04:35:00 1
```

8.4. Conclusão

A implementação da conjunção "ou" no sistema de busca aumentou a flexibilidade e a expressividade das consultas, permitindo ao usuário combinar diferentes critérios de forma mais abrangente. Através do método *‘inserirNoHeapUniao’*, o sistema pode retornar resultados que atendem a qualquer um dos filtros estabelecidos, ampliando as possibilidades de pesquisa sem impactar a eficiência da estrutura de dados existente.

Os testes demonstraram que a funcionalidade adicional permite consultas mais amplas, exibindo voos que atendem a múltiplas condições de forma independente. A manutenção da complexidade temporal O(n log n) e espacial O(n) comprova que a nova funcionalidade foi integrada sem comprometer o desempenho do sistema.