Relatório do Trabalho 1 de Estruturas de Dados II: Comparação de Desempenho em Árvores Binárias e AVL

João Marcos Sousa Rufino Leal
jsousarufinoleal@ufpi.edu.br
Estruturas de Dados II- Juliana Oliveira de Carvalho

Abstract

Este relatório fundamenta-se na resolução do Trabalho 1 da primeira avaliação da disciplina Estruturas de Dados II, visando abordar os aspectos essenciais para a compreensão do problema e sua solução. As questões foram implementadas utilizando a linguagem de programação C, empregando estruturas de dados hierárquicas, especificamente árvores binárias de busca e árvores AVL, que oferecem balanceamento automático. Os testes realizados com diferentes conjuntos de dados, incluindo cenários com dados ordenados, avaliam as operações de inserção e busca, confirmando a adequação de ambas as estruturas para sistemas que requerem otimização no gerenciamento de dados.

Palavras-chave: Estruturas de Dados, Árvore Binária, Árvore AVL, Balanceamento, Operações de Inserção e Busca

1 Introdução

Este relatório apresenta uma análise comparativa do desempenho entre Árvores Binárias de Busca (ABB) e Árvores AVL, estruturas de dados hierárquicas amplamente empregadas para a organização e manipulação eficiente de informações. As estruturas foram implementadas na linguagem C e avaliadas com ênfase nas operações de inserção e busca, considerando diferentes cenários de teste que variaram o volume de dados (1000, 2000 e 3000 artistas, com respectivos álbuns e músicas associados) e o padrão de ordenação (crescente, decrescente e aleatório).

O objetivo principal da análise foi investigar o impacto do balanceamento automático da Árvore AVL em comparação ao comportamento natural da ABB, particularmente em contextos de grandes volumes de dados e entradas altamente ordenadas, nas quais a ABB tende a degenerar, assumindo comportamento linear e comprometendo sua eficiência. Para a avaliação, foram implementadas funções específicas para as operações básicas de ambas as estruturas, assegurando a fidelidade dos testes de desempenho.

A metodologia consistiu na criação de conjuntos de dados estruturados, simulando um ambiente real de gerenciamento de artistas, álbuns e músicas. Em cada cenário, foram realizados múltiplos ciclos de inserções e buscas, permitindo uma coleta abrangente de tempos de execução, a partir da qual foram analisadas médias, variações e comportamento em casos extremos. Essa abordagem possibilitou identificar de maneira precisa as diferenças de desempenho entre as duas estruturas, considerando tanto a eficiência média quanto a estabilidade sob diferentes padrões de dados.

Foram avaliados aspectos como o crescimento da árvore em relação ao número de inserções, a eficiência das operações de busca e o impacto do balanceamento dinâmico da AVL em cenários variados. Em situações de dados ordenados, observou-se que a ABB perde seu

comportamento eficiente típico (O(log n)) e aproxima-se de uma complexidade O(n), enquanto a AVL mantém-se balanceada, preservando a complexidade logarítmica das operações. Em contrapartida, em cenários aleatórios, a ABB pode formar árvores razoavelmente balanceadas de forma natural, reduzindo a diferença de desempenho em relação à AVL.

O relatório está organizado em seções que descrevem o ambiente de hardware utilizado para os testes, as estruturas de dados e as funções implementadas, a metodologia aplicada para a realização dos experimentos, os resultados obtidos com análises comparativas entre ABB e AVL, e, por fim, as conclusões sobre o desempenho e a adequação de cada estrutura para diferentes padrões de dados e volumes de informação.

2 Hardware tilizado

Todos os testes demonstrados neste relatório foram feitos utilizando o mesmo hardware com as seguintes especificações.

Table 1: Especificações de Hardwere

Componente	**Especificação**
Marca/Modelo	Samsung Galaxy Book 2
Processador	Intel Core i5-1235U (12ª geração)
Núcleos / Threads	10 núcleos (2 P-cores + 8 E-cores) / 12 threads
Memória RAM	16 GB LPDDR4x 3200 MHz
Armazenamento	256 GB SSD NVMe
Sistema Operacional	Windows 11 Home

3 Seções Específicas

As seções a seguir descrevem as estruturas utilizadas, o funcionamento das principais funções implementadas e os testes realizados para avaliação de desempenho.

3.1 Estruturas de Dados Utilizadas

Foram utilizadas duas estruturas de dados não lineares: a árvore binária de busca e a árvore AVL.

- **Árvore Binária**
  - Estrutura Artista: Contem os campos de nome do artista, tipo, estilo musical e numero de álbuns além dos ponteiros esquerda e direita e por fim um ponteiro para a árvore de álbuns do artista.
  - Estrutura Álbum: Representa um álbum de um artista. Contém informações como título, ano de lançamento, quantidade de músicas e ponteiros para organizar álbuns na árvore binária. Também tem um ponteiro para a árvore de músicas dentro desse álbum.
  - Estrutura Música: Contém informações sobre uma música, como título e duração. Também possui ponteiros para as músicas à esquerda e à direita, para organizar em uma árvore binária de busca.
  - Estrutura Playlist: Representa uma playlist de músicas. Contém o título da playlist, a quantidade de músicas e um ponteiro para organizar playlists na árvore binária de busca. Também possui um ponteiro para a árvore de músicas dentro da playlist.

– Estrutura Música Playlist: Representa uma música dentro de uma playlist. Contém informações sobre o título da música, sua duração, o artista e o álbum ao qual pertence. Também possui ponteiros para organizar músicas na árvore binária de busca dentro da playlist.

- **AVL**

  – A árvore AVL utiliza as mesmas estruturas de Artista, Álbum, Música, Playlist e Música da Playlist que são usadas na árvore binária, com a adição de um campo chamado altura, que indica a altura de cada nó e possibilita o balanceamento automático da árvore.

3.1.1 Implementação das Funções da Árvore Binária

Esta seção apresentará as funções utilizadas em cada árvore (Artista, Álbum, Música, Playlist e Música Playlist).

1. **Artista**

   - `aloca_no_artista`: Essa função cria um no do tipo artista, passando os parâmetros nome, tipo, e estilo musical (ponteiros para string). Também define a quantidade de álbuns como 0 e o ponteiro da árvore de álbuns como NULL (assim como os ponteiros esq e dir). Por fim retorna o nó no artista criado.
   - `existe_artista`: Essa função recebe como parâmetro o ponteiro para a raiz da árvore de artistas e o nome do artista (que é um ponteiro também), percorre a árvore de artista procurando um artista com o mesmo nome, se for achado retorna o nó do artista encontrado se não for encontrado retorna NULL.
   - `cadastrar_artista`: Coleta os dados do artista (nome, tipo, estilo), chama a função `aloca_no_artista` passando os dados coletados como parâmetro, por fim retorna o artista com os dados cadastrados (ou NULL se der errado).
   - `inserir_artista` Essa função recebe como parâmetro a raiz da árvore de artistas e o nó do artista a ser inserido, percorre a árvore até encontrar NULL (nesse caso adiciona o artista a árvore) ou um artista do mesmo nome (nesse caso ele não insere o artista na árvore e limbera o nó não inserido), por fim retorna 1 (inserido) ou 0 (não inserido).
   - `imprimir_todos_os_dados_dos_artistas`: Essa função recebe a raiz da árvore de artistas, verifica se é NULL para percorrer a árvore recursivamente imprimindo todos os dados do artista. Retorna 1 (imprimiu) ou 0 (não imprimiu).
   - `imprimir_artistas_tipo`: função recebe a raiz da árvore de artistas, verifica se é NULL para percorrer a árvore recursivamente imprimindo todos os artistas (só o nome) de um determinado tipo. Retorna 1 (imprimiu) ou 0 (não imprimiu).
   - `imprimir_artistas_estilo`: Essa função recebe a raiz da árvore de artistas, verifica se é NULL para percorrer a árvore recursivamente imprimindo todos os artistas de um determinado estilo musical (só o nome) de um determinado tipo. Retorna 1 (imprimiu) ou 0 (não imprimiu).
   - `imprime_artista_estilo_e_tipo`: Recebe a raiz da árvore de artistas, verifica se é NULL para percorrer a árvore recursivamente imprimindo todos os artistas de um determinado tipo e estilo musical (só o nome) de um determinado tipo. Retorna 1 (imprimiu) ou 0 (não imprimiu).

- `limpar_no_artista`: Recebe um nó do tipo artista e libera as strings apontadas pelos ponteiros de strings da estrutura artista, setando os mesmos como NULL. Não retorna nada, pois modifica o nó diretamente.
- `liberar_arv_artista`: Recebe a raiz da árvore de artista, verifica se é NULL, verifica se existe árvore de álbuns (se existir chama a função `liberar_arv_album`), percorre a árvore de artistas recursivamente até as folhas e volta limpando e liberando os nós.

2. **Álbum**

As funções `aloca_no_album`, `cadastrar_album`, `existe_album` e `insere_album` seguem a mesma lógica das funções equivalentes para artistas. A principal diferença está nos parâmetros utilizados, que agora são o título do álbum e o ano de lançamento.

- `aloca_no_album`: aloca e inicializa um novo nó de álbum.
- `cadastrar_album`: verifica duplicidade e insere o álbum na árvore.
- `existe_album`: busca um álbum pelo título.
- `insere_album`: insere o nó na posição correta da árvore binária.
- `mostrar_dados_album`: Recebe um nó de álbum e imprime todos os dados dele (titulo, ano de lançamento e quantidade de músicas)
- `imprimir_todos_albuns`: Recebe a raiz para a árvore de álbuns de um artista e imprime todos os álbuns retornando 1 (imprimiu) e 0 (não imprimiu).
- `imprimir_albuns_ano`: Recebe a raiz da árvore de álbuns e o ano de lançamento, percorrendo a árvore de álbuns e imprimindo os álbuns do ano buscado. Retorna 1 (imprimiu) e 0 (não imprimiu).
- `mostrar_todos_artistas_album_ano`: Receba a raiz da árvore de artistas e o ano de lançamento, percorre a árvore de artista imprimindo todos os álbuns lançados no ano buscado, de cada artista. Retorna 1 (imprimiu) e 0 (não imprimiu).
- As funções `limpar_no_album` e `liberar_arv_album` seguem a mesma logica de suas equivalências em Artista. Com a `liberar_arv_album` verificando a existência de uma árvore de música e chamando a `liberar_arv_musica`.

3. **Música**

- As funções `aloca_no_musica`, `cadastrar_musica`, `existe_musica` e `insere_musica` seguem a mesma estrutura das utilizadas para artistas e álbuns. Os parâmetros tratados agora são o título da música e a duração.
- `imprimir_todas_as_musicas`: Recebe a raiz de uma árvore de música a percorrendo e imprimindo às músicas em ordem alfabética. Retorna 1 (imprimiu) e 0 (não imprimiu).
- As funções `imprime_dados_da_musica_album_arista`, `imprime_dados_da_musica_album` e `imprime_dados_da_musica_buscada` trabalham em conjunto para buscar uma música somente pelo nome, elas adentram as camadas Artista, Álbum e Música respectivamente. Caso a camada de música encontre a música, ela a imprime e retorna 1 (achou) para a camada de álbuns que imprime o álbum e retorna o 1 (achou) para a camada de Artista que imprime o nome do artista e essa camada por sua vez retorna 1 (achou) ou 0 (não achou) para a interface possibilitado aferir se a música foi achada ou não.

- As funções `limpa_no_musica` e `liberar_arv_musica` funcionam igualmente as de Artista e Álbum, com a única diferença que a função `liberar_arv_musica` não precisa verificar a existência de nenhuma subárvore.
- `eh_folha_musica`: Verifica se o nó da música é uma folha, ou seja, não possui filhos.
- `so_um_filho_musica`: Retorna o filho existente de um nó que possui apenas um filho (esquerdo ou direito). Se tiver dois filhos ou nenhum, retorna NULL.
- `dois_filhos_musica`: Retorna 1 se o nó possui dois filhos, caso contrário, retorna 0.
- `menor_no_musica`: Localiza e retorna o nó com o menor valor (mais à esquerda) na subárvore passada como parâmetro.
- `remove_musica`: Remove uma música da árvore com base no título. Trata os três casos clássicos:
  - Nó folha: simplesmente limpa o nó e remove.
  - Nó com um filho: liga o filho ao pai do nó removido.
  - Nó com dois filhos: substitui o conteúdo pelo menor nó da subárvore direita e remove esse menor nó recursivamente.

4. **Playlist**

- `aloca_no_playlist`: Aloca e inicializa um novo nó de playlist com o título passado por parâmetro. Inicializa a quantidade de músicas como 0 e os ponteiros `arv_musicas_playlist`, esq e dir como NULL.
- `existe_playlist`: Percorre a árvore de playlists buscando por uma com o título especificado. Se encontrada, retorna o ponteiro para o nó correspondente; caso contrário, retorna NULL.
- `cadastrar_playlist`: Solicita ao usuário o nome da playlist, chama `aloca_no_playlist` para criar o nó e retorna esse nó. Caso ocorra erro na alocação, retorna NULL
- `inserir_playlist`: Insere a nova playlist na árvore de forma ordenada (por título). Se já existir uma playlist com o mesmo nome, libera a memória do novo nó e retorna 0. Caso contrário, insere corretamente e retorna 1.
- `percorre_todas_as_playlists_buscando_uma_musica`: Percorre toda a árvore de playlists e, para cada playlist, verifica se a música informada (por título, álbum e artista) está presente. Retorna 1 se encontrar a música em alguma playlist; senão, retorna 0.
- `imprimir_todas_as_playlists`: Percorre a árvore de playlists e imprime os títulos de todas em ordem alfabética. Retorna 1 se imprimiu algo e 0 se a árvore estiver vazia.
- `imprime_dados_de_uma_playlist`: Imprime o nome da playlist, a quantidade de músicas e chama a função que imprime todas as músicas da playlist. Retorna 1 se os dados foram impressos com sucesso.
- `limpar_no_playlist`: Libera a string do título da playlist e seta o ponteiro como NULL. Não libera o nó em si.
- `liberar_arv_playlist`: Libera toda a árvore de playlists. Antes de liberar o nó da playlist, libera também sua árvore de músicas, chamando `liberar_arv_musica_playlist`.

- `eh_folha_playlist`: Verifica se o nó da playlist é uma folha, ou seja, não tem filhos à esquerda nem à direita.
- `so_um_filho_playlist`: Verifica se o nó da playlist possui apenas um filho (à esquerda ou à direita) e retorna esse filho. Se tiver dois filhos ou nenhum, retorna NULL
- `menor_no_playlist`: Retorna o nó com o menor valor (mais à esquerda) da subárvore passada como parâmetro.
- `remove_playlist`: Remove uma playlist da árvore juntamente com sua árvore de músicas da playlist com base no título. Trata os três casos:
  - Nó folha: simplesmente limpa o nó e remove.
  - Nó com um filho: liga o filho ao pai do nó removido.
  - Nó com dois filhos: substitui o conteúdo pelo menor nó da subárvore direita e remove esse menor nó recursivamente.

5. **Música Playlist**

- `alocar_musica_playlist` Equivalente a `aloca_no_album`, mas específica para músicas em playlists. Aloca e inicializa uma struct `MUSICA_PLAYLIST` com título, duração, artista e álbum.
- **inserir_musica_playlist** Lógica semelhante à de `inserir_album` e `inserir_artista`, mas com chave composta: título da música, título do álbum e nome do artista. Se a música já estiver na playlist (todos os campos iguais), não insere.
- `verifica_se_musica_esta_na_playlist` Percorre a árvore binária da playlist recursivamente para verificar se uma música com título, artista e álbum específicos está presente. Retorna 1 se encontrar, 0 caso contrário.
- A função `cadastrar_musica_playlist`, equivalente a `cadastrar_album` ou `cadastrar_musica`, realiza o cadastro de uma música em uma playlist seguindo os passos: solicita o nome do artista e busca com `existe_artista`; solicita o título do álbum e busca com `existe_album`; solicita o título da música e busca com `existe_musica`. Caso todos os dados sejam encontrados, a música é alocada com `alocar_musica_playlist` e inserida na árvore da playlist com `inserir_musica_playlist`. Os retornos possíveis são: 1 para sucesso, 2 se a música não for encontrada, 3 se o álbum não for encontrado, 4 se o artista não for encontrado, e 5 em caso de erro na alocação.
- `imprime_todas_as_musicas_da_playlist` Idêntica em lógica a `mostrar_todas_as_musicas_de_um_album`. Percorre in-order a árvore de músicas da playlist e imprime os títulos.
- `eh_folha_musica_playlist`, `so_um_filho_musica_playlist`, `menor_musica_playlist`. Todas seguem a lógica já descrita em funções equivalentes como `eh_folha_musica` ou `menor_no_musica`. São funções auxiliares para remoção.
- `remove_musica_playlist` Idêntica em estrutura à `remove_album` ou `remove_musica`. Compara a chave composta (título, álbum, artista) para localizar e remover a música na árvore da playlist.
- `liberar_arv_musica_playlist` Libera a árvore de músicas da playlist recursivamente. Mesma lógica de `liberar_arv_album` e `liberar_arv_musica`.

3.1.2 Implementação das Funções da Árvore AVL

AVL é um tipo de árvore binária de busca que se destaca por manter um equilíbrio entre seus nós. Por esse motivo, as funções já discutidas anteriormente não serão repetidas aqui. A principal distinção está nos procedimentos de inserção e remoção, nos quais são adicionadas funções específicas para manter o balanceamento da árvore e para atualizar a altura dos nós. Além disso, apenas as funções relacionadas à estrutura de Artista serão apresentadas, uma vez que todas seguem a mesma lógica. O que as diferencia é apenas o tipo de parâmetro utilizado em cada uma.

- `pegar_altura_artista`: Retorna a altura do nó fornecido. Se o nó for nulo, retorna -1.
- `atualizar_altura_artista`: Atualiza o campo `altura_artista` de um nó com base nas alturas dos filhos esquerdo e direito, utilizando a maior altura entre eles somada de 1.
- `fator_balanceamento_artista`: Calcula e retorna o fator de balanceamento do nó, que é a diferença entre a altura da subárvore esquerda e da direita.
- `rotacao_esq_artista`: Executa uma rotação simples à esquerda. A raiz passa a ser o filho direito, e a antiga raiz vira o filho esquerdo do novo nó raiz. Após a rotação, atualiza a altura dos nós envolvidos.
- `rotacao_dir_artista`: Executa uma rotação simples à direita. A raiz passa a ser o filho esquerdo, e a antiga raiz vira o filho direito do novo nó raiz. Após a rotação, atualiza a altura dos nós envolvidos.
- `balanceamento_artista`: Verifica o fator de balanceamento do nó raiz e aplica rotações simples ou duplas conforme necessário para manter a propriedade de balanceamento AVL da árvore de artistas.

4 Resultados da Execução do Programa

Esta seção apresenta os resultados obtidos a partir da comparação dos tempos de execução das operações de inserção e busca nas estruturas de árvore binária e árvore AVL. Os tempos foram registrados tendo como unidade de medida os segundos.

4.1 Testes

Os testes foram realizados com diferentes quantidades de dados, a fim de avaliar distintos cenários e analisar o comportamento das árvores em diversos níveis de profundidade. Para cada cenário, foram realizadas 10 execuções. Nas operações de busca, cada uma dessas 10 execuções incluiu 1000 buscas.

A primeira rodada de testes foi realizada com 1000 artistas, cada um contendo 10 álbuns, e cada álbum com 10 músicas. A segunda rodada utilizou 2000 artistas, com 20 álbuns por artista e 20 músicas por álbum. Por fim, a terceira rodada envolveu 3000 artistas, com 30 álbuns por artista e 30 músicas por álbum.

Essas três configurações foram testadas com dados ordenados de forma crescente, decrescente e de maneira aleatória, permitindo uma análise abrangente do desempenho das estruturas de árvore binária e AVL em diferentes cenários e com variados volumes de dados.

Table 2: Resultados de Desempenho da Árvore Binária de Busca (1000 10 10)

Operação	Cenário	Quantidade Total	Média (10 execuções)	Mínimo (s)	Máximo (s)
Inserção de Artistas	Crescente	10 000	0.020 103 5	0.006 835	0.047 588
	Decrescente	10 000	0.011 523 9	0.000 000	0.023 024
	Aleatória	10 000	0.002 693 4	0.000 000	0.015 992
Inserção de Álbuns	Crescente	100 000	0.174 145 3	0.158 613	0.192 674
	Decrescente	100 000	0.178 535 4	0.173 002	0.189 588
	Aleatória	100 000	0.020 237 5	0.010 221	0.034 750
Inserção de Músicas	Crescente	1 000 000	2.993 679 1	1.837 494	4.706 519
	Decrescente	1 000 000	2.439 242 2	1.863 807	4.646 720
	Aleatória	1 000 000	0.247 953 8	0.233 655	0.268 957
Busca de Músicas	Crescente	10 000	0.000 034 895 7	0.000 022 326	0.000 048 069
	Decrescente	10 000	0.000 036 306 4	0.000 025 864	0.000 053 812
	Aleatória	10 000	0.000 019 048 8	0.000 015 326	0.000 032 341

Table 3: Resultados de Desempenho da Árvore AVL (1000 10 10)

Operação	Cenário	Quantidade Total	Média (10 execuções)	Mínimo (s)	Máximo (s)
Inserção de Artistas	Crescente	10 000	0.003 192 5	0.000 000	0.016 069
	Decrescente	10 000	0.000 000 0	0.000 000	0.000 000
	Aleatória	10 000	0.001 580 2	0.000 000	0.008 012
Inserção de Álbuns	Crescente	100 000	0.016 280 1	0.007 806	0.024 556
	Decrescente	100 000	0.016 171 9	0.006 596	0.023 830
	Aleatória	100 000	0.015 938 3	0.010 292	0.021 545
Inserção de Músicas	Crescente	1 000 000	0.200 809 9	0.189 293	0.208 891
	Decrescente	1 000 000	0.206 715 4	0.200 671	0.212 146
	Aleatória	1 000 000	0.234 157 1	0.219 230	0.252 472
Busca de Músicas	Crescente	10 000	0.000 020 223 3	0.000 009 628	0.000 031 941
	Decrescente	10 000	0.000 018 802 4	0.000 011 300	0.000 034 051
	Aleatória	10 000	0.000 020 696 6	0.000 011 654	0.000 027 823

4.2 Análise Comparativa: Árvore Binária vs. Árvore AVL (1000 10 10)

1. **Inserção de Artistas (10.000)**

   - **Crescente**: AVL é 6,3x mais rápida (0.003 192 5 s vs. 0.020 103 5 s) devido ao balanceamento ($\mathcal{O}(\log n)$ vs. $\mathcal{O}(n)$ da Binária degenerada).
   - **Decrescente**: AVL é muito mais rápida (0.000 000 0 s vs. 0.011 523 9 s), pois evita degeneração.
   - **Aleatória**: AVL é 1,7x mais rápida (0.001 580 2 s vs. 0.002 693 4 s) e mais consistente.

2. **Inserção de Álbuns (100.000)**

   - **Crescente**: AVL é 10,7x mais rápida (0.016 280 1 s vs. 0.174 145 3 s) devido à altura logarítmica.
   - **Decrescente**: AVL é 11,0x mais rápida (0.016 171 9 s vs. 0.178 535 4 s), mantendo eficiência.
   - **Aleatória**: AVL é 1,3x mais rápida (0.015 938 3 s vs. 0.020 237 5 s), com menor variabilidade.

3. **Inserção de Músicas (1.000.000)**

   - **Crescente**: AVL é 14,9x mais rápida (0.200 809 9 s vs. 2.993 679 1 s), beneficiada pelo volume maior.
   - **Decrescente**: AVL é 11,8x mais rápida (0.206 715 4 s vs. 2.439 242 2 s), evitando degeneração.
   - **Aleatória**: AVL é 1,06x mais rápida (0.234 157 1 s vs. 0.247 953 8 s), com desempenhos próximos.

4. **Busca de Músicas (10.000)**

   - **Crescente**: AVL é 1,7x mais rápida (0.000 020 223 3 s vs. 0.000 034 895 7 s) devido a $\mathcal{O}(\log n)$.
   - **Decrescente**: AVL é 1,9x mais rápida (0.000 018 802 4 s vs. 0.000 036 306 4 s), mantendo eficiência.
   - **Aleatória**: Binária é 0,92x mais rápida (0.000 019 048 8 s vs. 0.000 020 696 6 s), mas a diferença é mínima.

Table 4: Resultados de Desempenho da Árvore Binária de Busca (2000 20 20)

Operação	Cenário	Quantidade Total	Média (10 execuções)	Mínimo (s)	Máximo (s)
Inserção de Artistas	Crescente	20 000	0.067 852 5	0.057 756	0.081 765
	Decrescente	20 000	0.036 413 7	0.022 353	0.057 291
	Aleatória	20 000	0.005 080 3	0.000 000	0.022 186
Inserção de Álbuns	Crescente	400 000	2.020 074 3	1.332 677	3.438 786
	Decrescente	400 000	1.937 958 0	1.352 225	2.856 489
	Aleatória	400 000	0.076 268 4	0.065 727	0.088 874
Inserção de Músicas	Crescente	8 000 000	39.617 541 2	36.864 946	42.563 074
	Decrescente	8 000 000	40.232 401 7	35.344 883	42.228 747
	Aleatória	8 000 000	3.259 755 0	2.114 497	4.981 516
Busca de Músicas	Crescente	10 000	0.000 047 532 5	0.000 047 221	0.000 047 763
	Decrescente	10 000	0.000 091 870 1	0.000 078 778	0.000 099 554
	Aleatória	10 000	0.000 033 666 0	0.000 019 682	0.000 047 234

Table 5: Resultados de Desempenho da Árvore AVL (2000 20 20)

Operação	Cenário	Quantidade Total	Média (10 execuções)	Mínimo (s)	Máximo (s)
Inserção de Artistas	Crescente	20 000	0.004 995 3	0.000 000	0.033 499
	Decrescente	20 000	0.003 794 8	0.000 000	0.014 242
	Aleatória	20 000	0.005 140 0	0.000 000	0.025 938
Inserção de Álbuns	Crescente	400 000	0.062 657 2	0.047 226	0.075 340
	Decrescente	400 000	0.064 647 1	0.058 707	0.074 506
	Aleatória	400 000	0.070 162 0	0.061 212	0.084 141
Inserção de Músicas	Crescente	8 000 000	1.725 190 2	1.664 885	1.892 087
	Decrescente	8 000 000	2.303 578 2	1.699 238	4.611 410
	Aleatória	8 000 000	2.408 994 6	2.065 196	3.743 765
Busca de Músicas	Crescente	10 000	0.000 036 979 7	0.000 029 333	0.000 044 941
	Decrescente	10 000	0.000 035 857 7	0.000 024 091	0.000 047 381
	Aleatória	10 000	0.000 085 418 1	0.000 024 523	0.000 126 499

4.3 Análise Comparativa: Árvore Binária vs. Árvore AVL (2000 20 20)

1. **Inserção de Artistas (20.000)**
   - **Crescente**: AVL é 13,6x mais rápida (0.004 995 3 s vs. 0.067 852 5 s) devido ao balanceamento ($\mathcal{O}(\log n)$ vs. $\mathcal{O}(n)$ da Binária degenerada).
   - **Decrescente**: AVL é 9,6x mais rápida (0.003 794 8 s vs. 0.036 413 7 s), mantendo eficiência.
   - **Aleatória**: Binária é 1,01x mais rápida (0.005 080 3 s vs. 0.005 140 0 s), beneficiada por dados aleatórios.
2. **Inserção de Álbuns (400.000)**
   - **Crescente**: AVL é 32,2x mais rápida (0.062 657 2 s vs. 2.020 074 3 s), beneficiada pelo balanceamento.
   - **Decrescente**: AVL é 30,0x mais rápida (0.064 647 1 s vs. 1.937 958 0 s), evitando degeneração.
   - **Aleatória**: Binária é 1,1x mais rápida (0.076 268 4 s vs. 0.070 162 0 s), devido a dados aleatórios balanceados.
3. **Inserção de Músicas (8.000.000)**
   - **Crescente**: AVL é 23,0x mais rápida (1.725 190 2 s vs. 39.617 541 2 s), destacando sua escalabilidade.
   - **Decrescente**: AVL é 17,5x mais rápida (2.303 578 2 s vs. 40.232 401 7 s), com balanceamento superior.
   - **Aleatória**: Binária é 1,4x mais rápida (3.259 755 0 s vs. 2.408 994 6 s), favorecida por dados balanceados.
4. **Busca de Músicas (10.000)**
   - **Crescente**: AVL é 1,3x mais rápida (0.000 036 979 7 s vs. 0.000 047 532 5 s) devido a $\mathcal{O}(\log n)$.
   - **Decrescente**: AVL é 2,6x mais rápida (0.000 035 857 7 s vs. 0.000 091 870 1 s), com maior eficiência.
   - **Aleatória**: Binária é 2,5x mais rápida (0.000 033 666 0 s vs. 0.000 085 418 1 s), com menor overhead em árvore balanceada.

Table 6: Resultados de Desempenho da Árvore Binária de Busca (3000 30 30)

Operação	Cenário	Quantidade Total	Média (10 execuções)	Mínimo (s)	Máximo (s)
Inserção de Artistas	Crescente	30 000	0.149 567 9	0.137 235	0.176 960
	Decrescente	30 000	0.083 914 3	0.078 745	0.111 892
	Aleatória	30 000	0.006 371 4	0.000 000	0.025 874
Inserção de Álbuns	Crescente	900 000	6.118 587 0	4.423 566	7.653 144
	Decrescente	900 000	6.514 058 9	4.571 481	7.386 928
	Aleatória	900 000	0.183 745 5	0.170 759	0.198 278
Inserção de Músicas	Crescente	27 000 000	199.857 856 6	193.836 239	207.856 344
	Decrescente	27 000 000	198.341 305 0	192.424 037	202.774 875
	Aleatória	27 000 000	10.291 183 8	7.840 103	13.335 954
Busca de Músicas	Crescente	10 000	0.000 062 201 3	0.000 050 403	0.000 068 838
	Decrescente	10 000	0.000 155 814 4	0.000 142 400	0.000 164 056
	Aleatória	10 000	0.000 049 902 4	0.000 038 139	0.000 060 511

Table 7: Resultados de Desempenho da Árvore AVL (3000 30 30)

Operação	Cenário	Quantidade Total	Média (10 execuções)	Mínimo (s)	Máximo (s)
Inserção de Artistas	Crescente	30 000	0.004 345 9	0.000 000	0.015 964
	Decrescente	30 000	0.013 724 8	0.007 267	0.016 949
	Aleatória	30 000	0.005 779 0	0.000 000	0.015 766
Inserção de Álbuns	Crescente	900 000	0.138 572 1	0.127 316	0.146 986
	Decrescente	900 000	0.238 859 0	0.127 666	0.447 123
	Aleatória	900 000	0.163 827 3	0.158 357	0.175 423
Inserção de Músicas	Crescente	27 000 000	7.969 770 2	5.911 991	9.587 183
	Decrescente	27 000 000	8.856 552 2	8.703 884	9.491 938
	Aleatória	27 000 000	12.315 810 8	7.796 382	15.066 050
Busca de Músicas	Crescente	10 000	0.000 163 295 3	0.000 158 614	0.000 176 087
	Decrescente	10 000	0.000 159 123 5	0.000 087 791	0.000 177 887
	Aleatória	10 000	0.000 166 432 9	0.000 157 603	0.000 177 300

4.4 Análise Comparativa: Árvore Binária vs. Árvore AVL (3000 30 30)

1. **Inserção de Artistas (30.000)**

   - **Crescente**: AVL é 34,4x mais rápida (0.004 345 9 s vs. 0.149 567 9 s) devido ao balanceamento ($\mathcal{O}(\log n)$ vs. $\mathcal{O}(n)$ da Binária degenerada).
   - **Decrescente**: AVL é 6,1x mais rápida (0.013 724 8 s vs. 0.083 914 3 s), mantendo eficiência.
   - **Aleatória**: AVL é 1,1x mais rápida (0.005 779 0 s vs. 0.006 371 4 s), com leve vantagem.

2. **Inserção de Álbuns (900.000)**

   - **Crescente**: AVL é 44,2x mais rápida (0.138 572 1 s vs. 6.118 587 0 s), beneficiada pelo balanceamento.
   - **Decrescente**: AVL é 27,3x mais rápida (0.238 859 0 s vs. 6.514 058 9 s), evitando degeneração.
   - **Aleatória**: AVL é 1,1x mais rápida (0.163 827 3 s vs. 0.183 745 5 s), com melhor consistência.

3. **Inserção de Músicas (27.000.000)**

   - **Crescente**: AVL é 25,1x mais rápida (7.969 770 2 s vs. 199.857 856 6 s), destacando sua escalabilidade.
   - **Decrescente**: AVL é 22,4x mais rápida (8.856 552 2 s vs. 198.341 305 0 s), com balanceamento superior.
   - **Aleatória**: Binária é 1,2x mais rápida (10.291 183 8 s vs. 12.315 810 8 s), favorecida por dados balanceados.

4. **Busca de Músicas (10.000)**

   - **Crescente**: Binária é 2,6x mais rápida (0.000 062 201 3 s vs. 0.000 163 295 3 s), com menor overhead.
   - **Decrescente**: Binária é 1,0x mais rápida (0.000 155 814 4 s vs. 0.000 159 123 5 s), com desempenho similar.
   - **Aleatória**: Binária é 3,3x mais rápida (0.000 049 902 4 s vs. 0.000 166 432 9 s), beneficiada por árvore balanceada.

5 Conclusão

O experimento realizou uma comparação de desempenho entre Árvores Binárias de Busca (ABB) e Árvores AVL em operações de inserção e busca, ambas implementadas na linguagem C. Foram considerados três cenários distintos, variando a quantidade de dados (1000 artistas/10 álbuns/10 músicas, 2000/20/20 e 3000/30/30), utilizando conjuntos de dados ordenados (em ordem crescente e decrescente) e aleatórios.

Os resultados indicaram que a Árvore AVL apresentou desempenho significativamente superior nas inserções de dados ordenados, sendo até 44 vezes mais rápida (por exemplo, 7,9697702 segundos contra 199,8578566 segundos para 27 milhões de músicas no cenário de 3000/30/30 com dados crescentes). Essa vantagem decorre do balanceamento automático da AVL, que garante complexidade O(log n) mesmo em situações de ordenação extrema. Em contraste, a Árvore Binária de Busca sofre degeneração em dados ordenados, assumindo uma estrutura similar à de uma lista encadeada, resultando em complexidade O(n) e desempenho inferior.

Nos cenários com dados aleatórios, a ABB mostrou-se competitiva, superando a AVL em operações de busca (até 3,3 vezes mais rápida, como no caso de 0,0000499024 segundos contra 0,0001664329 segundos no cenário 3000/30/30). Também obteve desempenho superior em algumas inserções aleatórias, sendo 1,01 vezes mais rápida na inserção de artistas (0,0050803 segundos contra 0,0051400 segundos) e 1,1 vezes na inserção de álbuns (0,0762684 segundos contra 0,0701620 segundos) no cenário 2000/20/20, além de 1,4 vezes na inserção de músicas no mesmo cenário (3,2597550 segundos contra 2,4089946 segundos) e 1,2 vezes no cenário 3000/30/30 (10,2911838 segundos contra 12,3158108 segundos). Esses resultados são atribuídos à formação natural de árvores relativamente balanceadas com dados aleatórios e ao menor overhead estrutural da ABB. Apesar disso, a Árvore AVL demonstrou maior estabilidade e escalabilidade para grandes volumes de dados.

O propósito deste experimento foi comparar o desempenho entre Árvores Binárias de Busca (ABB) e Árvores AVL nas operações de inserção e busca, analisando o impacto do balanceamento automático em diferentes cenários de dados. O objetivo foi alcançado, uma vez que o experimento evidenciou as vantagens da AVL em conjuntos de dados ordenados e a competitividade da ABB em dados aleatórios.

Os principais resultados mostraram que a Árvore AVL foi até 44 vezes mais rápida em inserções de dados ordenados. Por outro lado, a Árvore Binária de Busca foi até 3,3 vezes mais rápida em buscas aleatórias e também apresentou melhor desempenho em algumas inserções aleatórias.

Verificou-se que o balanceamento automático da AVL é fundamental para a manutenção de alta performance em grandes conjuntos de dados ordenados. Em contrapartida, a ABB mostrou-se eficiente em cenários aleatórios, onde há menor necessidade de operações estruturais complexas.

Com isso constatamos que escolha entre utilizar uma Árvore AVL ou uma Árvore Binária de Busca simples deve considerar o padrão dos dados e o tipo de operações predominantes. A AVL é mais indicada para sistemas críticos que lidam com grandes volumes de dados, enquanto a ABB representa uma solução mais simples e eficaz para cenários aleatórios com menor necessidade de balanceamento.