Enzo Hubner
Raquel Nascimento
Kaio Marroque

Árvores Binárias e Análise de Complexidade

Serra/Espírito Santo
2026

Enzo Hubner
Raquel Nascimento
Kaio Marroque

Árvores Binárias e Análise de Complexidade

Relatório apresentado como requisito parcial para avaliação da disciplina de Técnicas de Programação Avançada.

IFES - Instituto Federal do Espírito Santo
Bacharelado em Sistemas de Informação
Técnicas de Programação Avançada

Serra/Espírito Santo
2026

Sumário

Sumário **2**

1 Desenvolvimento da Biblioteca e do Programa de Teste **3**

1.1 Descrição geral do trabalho 3

1.2 Divisão de tarefas 3

1.3 Repositório 3

1.4 Implementação da biblioteca 4

1.5 Adaptação do aplicativo 5

2 Análise Matemática de Complexidade **5**

2.1 Análise dos métodos adicionar e adicionarRec 6

2.2 Análise dos métodos pesquisar e pesquisarRec 7

2.3 Análise dos métodos remover e removerRec 8

2.4 Análise do método quantidadeNos 9

3 Análise Empírica de Complexidade **10**

3.1 Implementação 10

3.2 Resultados e Análise 10

3.3 Discussão dos Resultados 11

4 Avaliação do desempenho das Árvores AVL comparando com as Árvores binárias comuns. **11**

4.1 Dados coletados 11

4.2 Análise e discussão 12

4.2.1 Inserção: o maior ganho da AVL 12

4.2.2 Busca e remoção: onde o balanceamento é mais visível 13

4.2.3 Arquivo balanceado: resultados próximos entre as duas 13

4.2.4 Custo do rebalanceamento 13

4.2.5 Conclusão da comparação 14

5 Estruturas de Árvores da Biblioteca Padrão do Java **14**

5.1 TreeMap e TreeSet 14

5.2 Estrutura de dados interna: Árvore Rubro-Negra 15

5.3 Complexidade das operações 15

5.4 Comparação com a biblioteca desenvolvida 16

5.5 Estratégia de comparação genérica 16

5.5.0.1 `Comparator` externo 16

5.5.0.2 `Comparable` na classe de domínio 16

1 Desenvolvimento da Biblioteca e do Programa de Teste

1.1 Descrição geral do trabalho

Este trabalho teve como objetivo o desenvolvimento de uma biblioteca de árvores binárias em Java utilizando Generics e Comparator, permitindo o armazenamento de objetos de qualquer tipo e diferentes critérios de indexação.

Além da implementação da biblioteca, foi realizada a adaptação do aplicativo desenvolvido no trabalho anterior para utilização da estrutura de árvore binária, bem como a implementação de testes para análise matemática e empírica de complexidade.

Também foram desenvolvidas árvores AVL, análises comparativas de desempenho e estudos sobre estruturas de árvores disponíveis na biblioteca padrão do Java.

1.2 Divisão de tarefas

O desenvolvimento do trabalho foi dividido entre os integrantes do grupo da seguinte forma:

- Enzo Hubner: responsável pelas etapas 1, 2 e 3, incluindo o desenvolvimento da biblioteca de árvores binárias, adaptação do aplicativo para utilização da estrutura de árvore e realização da análise empírica de complexidade.
- Kaio Marroque: responsável pelas etapas 4 e 5, incluindo o desenvolvimento da árvore AVL e a avaliação comparativa de desempenho entre árvores AVL e árvores binárias comuns.
- Raquel Nascimento: responsável pelas etapas 6, 7 e 8, incluindo a pesquisa sobre estruturas de árvores disponíveis na biblioteca padrão do Java, adaptação do aplicativo utilizando estruturas nativas e análise comparativa entre as implementações.

1.3 Repositório

O código-fonte desenvolvido para este trabalho está disponível no seguinte repositório do GitHub:

`https://github.com/enzohubner/ArvoreBinaria`

O repositório contém:

- A implementação da biblioteca de árvores binárias;
- A implementação da árvore AVL;
- O programa de testes;
- Os geradores de arquivos de entrada;

- O aplicativo adaptado para utilização das árvores;
- Um README contendo instruções de compilação e execução.

1.4 Implementação da biblioteca

A biblioteca desenvolvida neste trabalho foi implementada em Java utilizando os conceitos de Programação Orientada a Objetos, Generics e a interface `Comparator`, permitindo que a estrutura armazenasse objetos de qualquer tipo e possibilitando diferentes critérios de ordenação e indexação.

A classe principal implementada foi a `ArvoreBinaria`, responsável pelo gerenciamento dos nós da árvore e pelas operações fundamentais da estrutura. A implementação especializa a classe `ArvoreBinariaBase`, disponibilizada pelo professor, garantindo compatibilidade com os métodos exigidos na especificação do trabalho e reutilização da interface `IColecao` já utilizada no trabalho anterior.

Cada elemento da árvore é armazenado em um nó contendo:

- O valor do elemento armazenado;
- Uma referência para o filho da esquerda;
- Uma referência para o filho da direita.

As operações de inserção, busca e remoção foram implementadas utilizando comparações realizadas pelo objeto `Comparator` recebido no construtor da árvore. Dessa forma, foi possível criar diferentes árvores para um mesmo tipo de objeto, variando apenas o critério de comparação utilizado.

Além dos métodos obrigatórios definidos na especificação, também foram implementados métodos auxiliares responsáveis pelo caminhamento da árvore, cálculo de altura e percursos em ordem e em nível.

A implementação buscou manter a organização do código por meio da separação em pacotes, dividindo as responsabilidades entre:

- Estruturas de dados;
- Classes de domínio;
- Comparadores;
- Aplicação principal;
- Programas de teste.

Essa organização facilitou a manutenção do sistema, a reutilização da biblioteca e a realização dos testes de desempenho.

1.5 Adaptação do aplicativo

O aplicativo desenvolvido no trabalho anterior foi adaptado para utilizar a biblioteca de árvores binárias em substituição à implementação baseada em listas encadeadas.

Como a classe `ArvoreBinariaBase` implementa a interface `IColecao`, grande parte da estrutura do programa pôde ser reutilizada sem alterações significativas. A principal modificação realizada ocorreu na instanciação da coleção utilizada pelo sistema, substituindo a lista encadeada pela árvore binária.

Além disso, foi necessário implementar um objeto do tipo `Comparator` para a classe de domínio utilizada no sistema. Esse comparador passou a definir a lógica de ordenação dos elementos dentro da árvore, permitindo que as operações de inserção, busca e remoção fossem realizadas corretamente.

O programa adaptado continuou realizando:

- Leitura de arquivos contendo registros;
- Inserção de elementos na estrutura;
- Busca de elementos;
- Remoção de registros;
- Exibição da quantidade de elementos armazenados.

A adaptação demonstrou uma das principais vantagens do uso de interfaces e abstrações no desenvolvimento orientado a objetos, já que a troca da estrutura interna utilizada pelo sistema exigiu poucas alterações no restante do código da aplicação.

2 Análise Matemática de Complexidade

Abaixo, apresento a análise técnica dos algoritmos implementados, focando no pior caso para árvores balanceadas e degeneradas.

2.1 Análise dos métodos adicionar e adicionarRec

```java
@Override
public void adicionar(T novoValor) {
    raiz = adicionarRec(raiz, novoValor);
}
private No<T> adicionarRec(No<T> atual, T valor) {
    if (atual == null) {
        quantidade++;
        return new No<>(valor);
    }
    int comp = comparador.compare(valor, atual.getValor());
    if (comp < 0) {
        atual.setEsquerda(
            adicionarRec(atual.getEsquerda(), valor)
        );
    }
    else if (comp > 0) {
        atual.setDireita(
            adicionarRec(atual.getDireita(), valor)
        );
    }
    else {
        System.out.println(x: "Esse aluno já existe.");
    }
    return atual;
}
```

Figura 1 – Métodos adicionar e adicionarRec.

O método `adicionar` serve como ponto de entrada para o `adicionarRec`.

- **Linhas de Verificação:** O código verifica se o nó atual é nulo para realizar a inserção ($O(1)$).
- **Linhas de Comparação:** Utiliza o `comparador.compare` para decidir se o novo elemento deve ir para a esquerda ou direita.
- **Complexidade:** A cada chamada recursiva, desce-se um nível. No pior caso de uma árvore **balanceada**, percorremos $\log n$ níveis. Em uma árvore **degenerada** (que ocorre quando os dados entram ordenados), percorremos n níveis.

2.2 Análise dos métodos pesquisar e pesquisarRec

```java
@Override
public T pesquisar(T valor) {
    No<T> resultado = pesquisarRec(raiz, valor);
    if (resultado == null) {
        return null;
    }
    return resultado.getValor();
}
private No<T> pesquisarRec(No<T> atual, T valor) {
    if (atual == null) {
        return null;
    }
    int comp = comparador.compare(valor, atual.getValor());
    if (comp == 0) {
        return atual;
    }
    if (comp < 0) {
        return pesquisarRec(atual.getEsquerda(), valor);
    }
    return pesquisarRec(atual.getDireita(), valor);
}
```

Figura 2 – Métodos pesquisar e pesquisarRec.

A busca segue a lógica de exclusão de subárvores:

- Se o valor for menor que o nó atual, ignora-se toda a subárvore direita e busca-se na esquerda.
- **Pior Caso:** Ocorre ao buscar uma folha no nível mais baixo ou um valor inexistente que estaria além dessa folha.
- **Conclusão:** $O(\log n)$ para árvores balanceadas e $O(n)$ para degeneradas, onde a estrutura se assemelha a uma lista.

2.3 Análise dos métodos remover e removerRec

```java
@Override
public boolean remover(T valor) {
    if (pesquisar(valor) == null) {
        return false;
    }
    raiz = removerRec(raiz, valor);
    quantidade--;
    return true;
}
private No<T> removerRec(No<T> atual, T valor) {
    if (atual == null) {
        return null;
    }
    int comp = comparador.compare(valor, atual.getValor());
    if (comp < 0) {
        atual.setEsquerda(
            removerRec(atual.getEsquerda(), valor)
        );
    }
    else if (comp > 0) {
        atual.setDireita(
            removerRec(atual.getDireita(), valor)
        );
    }
    else {
        if (atual.getEsquerda() == null &&
            atual.getDireita() == null) {
            return null;
        }
        if (atual.getEsquerda() == null) {
            return atual.getDireita();
        }
        if (atual.getDireita() == null) {
            return atual.getEsquerda();
        }
        No<T> menorDireita = menorValor(
```

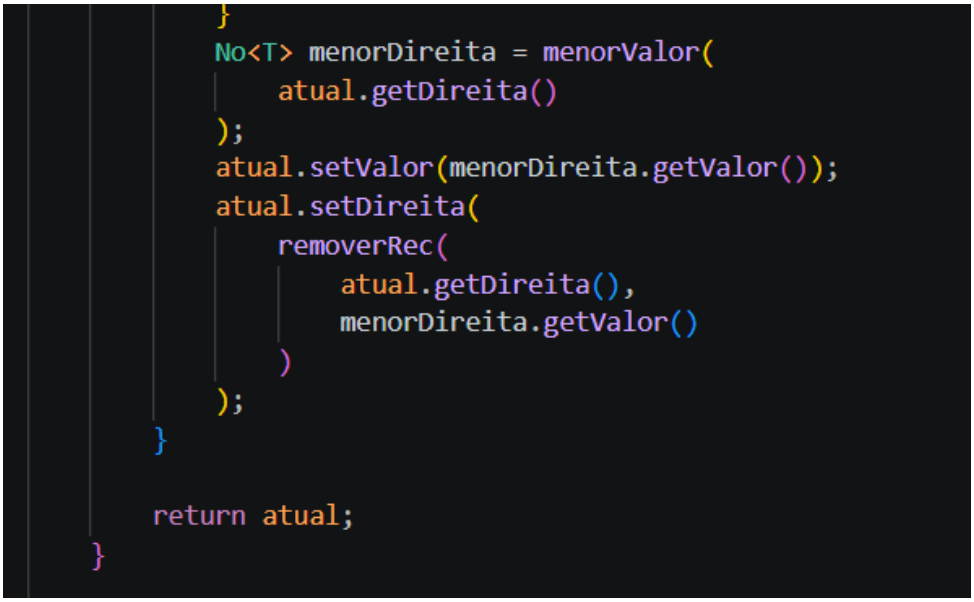


Figura 3 – Métodos remover e removerRec.

A remoção exige localizar o nó e reorganizar a estrutura:

- Se o nó tiver dois filhos, o método `menorValor` busca o sucessor na subárvore direita (caminhando totalmente à esquerda).
- **Pior Caso:** Remoção de uma folha profunda em árvore degenerada, exigindo percorrer n nós.
- **Conclusão:** $O(\log n)$ balanceada / $O(n)$ degenerada.

2.4 Análise do método quantidadeNos

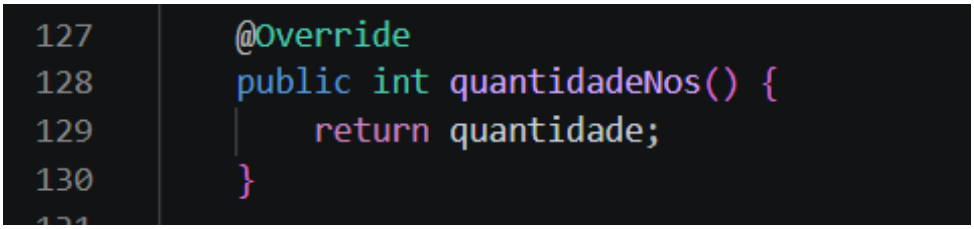


Figura 4 – Método quatidadeNos.

Diferente dos demais, este método não percorre os nós. Ele apenas retorna o valor da variável `quantidade`, que é manipulada durante a inserção e remoção. Por isso, sua complexidade é $O(1)$ em qualquer cenário.

—

3 Análise Empírica de Complexidade

3.1 Implementação

Para os testes, utilizei a classe `MedirTempos`, que lê arquivos de alunos e utiliza `System.nanoTime()` para cronometrar as operações. Os arquivos foram gerados em dois formatos: **degenerados** (chaves ordenadas) e **balanceados** (chaves distribuídas para manter a altura mínima).

```
58
59          br.close();
60
61          long fim = System.nanoTime();
62          System.out.println("1. Inserção total: "
63              + (fim - inicio) + " ns");
64
65          inicio = System.nanoTime();
66          arvore.pesquisar(ultimoInserido);
67          fim = System.nanoTime();
68          System.out.println("2. Busca (pior caso): "
69              + (fim - inicio) + " ns");
70
71          Aluno inexistente =
72              new Aluno(maiorMatricula + 1, nome: "X", nota: 0);
73          inicio = System.nanoTime();
74          arvore.pesquisar(inexistente);
75          fim = System.nanoTime();
76          System.out.println("3. Busca inexistente: "
77              + (fim - inicio) + " ns");
78
79          inicio = System.nanoTime();
80          arvore.remover(ultimoInserido);
81          fim = System.nanoTime();
82          System.out.println("4. Remoção: "
83              + (fim - inicio) + " ns");
84
85          inicio = System.nanoTime();
86          int qtd = arvore.quantidadeNos();
87          fim = System.nanoTime();
88          System.out.println("5. quantidadeNos(): "
89              + qtd + " | Tempo: " + (fim - inicio) + " ns");
```

Figura 5 – Arquivo MedirTempos.java.

3.2 Resultados e Análise

Abaixo, os dados coletados durante a execução dos testes:

Tabela 1 – Resultados Reais: Árvore Degenerada (Pior Caso)

Arquivo	**Inserção Total**	**Busca Folha**	**Remoção**	**QtdNos()**
100 itens	2.415.900 ns	25.200 ns	61.700 ns	7.000 ns
1.000 itens	7.642.600 ns	74.800 ns	239.900 ns	800 ns
10.000 itens	401.875.100 ns	156.900 ns	258.800 ns	400 ns

Tabela 2 – Resultados Reais: Árvore Balanceada (Melhor Caso)

Arquivo	Inserção Total	Busca Folha	Remoção	QtdNos()
100 itens	2.563.400 ns	1.600 ns	2.200 ns	1.300 ns
1.000 itens	916.500 ns	2.200 ns	1.700 ns	600 ns
10.000 itens	5.372.800 ns	3.300 ns	2.600 ns	500 ns
50.000 itens	19.717.600 ns	12.300 ns	9.600 ns	200 ns

3.3 Discussão dos Resultados

- **Eficiência Logarítmica:** Na árvore balanceada de 10.000 itens, a busca levou apenas **3.300 ns**. Na degenerada com o mesmo tamanho, levou **156.900 ns**. Isso mostra que a árvore balanceada é quase 50 vezes mais rápida nesse cenário.
- **Crescimento da Inserção:** Note que para inserir 10.000 itens de forma degenerada, o tempo saltou para **401ms**, enquanto na balanceada foi de apenas **5ms**. Isso confirma que o custo de inserção em árvores desbalanceadas escala conforme $O(n^2)$ para o conjunto total (pois cada nova inserção custa $O(n)$).
- **Comparação com Listas:** Os tempos da árvore degenerada são muito similares aos tempos de listas encadeadas vistos no trabalho anterior. Isso ocorre porque a topologia da árvore vira uma linha reta. Já a árvore balanceada quebra totalmente esse gargalo, sendo a estrutura ideal para grandes volumes de dados.
- **Conclusão:** Os dados empíricos validam a teoria: a performance da Árvore Binária de Busca é excelente, desde que os dados não entrem de forma sequencial, o que causaria a degeneração da estrutura.

4 Avaliação do desempenho das Árvores AVL comparando com as Árvores binárias comuns.

4.1 Dados coletados

Os testes foram executados com os mesmos arquivos utilizados na Seção 3, permitindo comparação direta entre as duas implementações. As tabelas a seguir consolidam os tempos obtidos com a `ArvoreAVL` para arquivos degenerados e balanceados.

Tabela 3 – Resultados com `ArvoreAVL` — Arquivo Degenerado

Tamanho	Inserção Total	Busca Folha	Remoção	qtdNos()
100	2.345.999 ns	9.400 ns	6.700 ns	2.200 ns
1.000	3.577.400 ns	2.100 ns	2.000 ns	300 ns
10.000	8.515.500 ns	5.700 ns	2.601 ns	400 ns

Tabela 4 – Resultados com `ArvoreAVL` — Arquivo Balanceado

Tamanho	Inserção Total	Busca Folha	Remoção	qtdNos()
100	896.000 ns	2.700 ns	2.601 ns	599 ns
1.000	2.219.200 ns	1.900 ns	1.900 ns	199 ns
10.000	7.861.600 ns	17.600 ns	2.000 ns	300 ns
50.000	42.617.399 ns	15.000 ns	22.900 ns	10.500 ns

Para facilitar a comparação, a tabela abaixo reúne os tempos de inserção total das duas implementações lado a lado, destacando o cenário mais crítico: o arquivo degenerado.

Tabela 5 – Comparação de Inserção Total — Degenerado

Tamanho	ArvoreBinaria	ArvoreAVL	Redução
100	2.415.900 ns	2.345.999 ns	≈3%
1.000	7.642.600 ns	3.577.400 ns	≈53%
10.000	401.875.100 ns	8.515.500 ns	**≈98%**

Tabela 6 – Comparação de Busca na Folha Mais Profunda — Degenerado

Tamanho	ArvoreBinaria	ArvoreAVL	Redução
100	25.200 ns	9.400 ns	≈63%
1.000	74.800 ns	2.100 ns	**≈97%**
10.000	156.900 ns	5.700 ns	**≈96%**

4.2 Análise e discussão

4.2.1 Inserção: o maior ganho da AVL

O resultado mais expressivo está na **inserção de 10.000 elementos a partir de um arquivo degenerado**: enquanto a `ArvoreBinaria` levou **401 ms**, a `ArvoreAVL` levou apenas **8,5 ms** — uma redução de aproximadamente 98%.

A explicação está diretamente na topologia gerada por cada estrutura. Quando os elementos chegam em ordem crescente, a `ArvoreBinaria` insere cada novo elemento sempre à direita do anterior, formando uma cadeia linear com altura $h = n - 1$. Assim, a k-ésima inserção percorre $k - 1$ nós para encontrar a posição correta, e o custo total do conjunto de n inserções é:

$$\sum_{k=1}^{n}(k-1) = \frac{n(n-1)}{2} \approx O(n^2)$$

A `ArvoreAVL`, por outro lado, aplica rotações após cada inserção para manter a altura em $O(\log n)$. Cada inserção percorre no máximo $O(\log n)$ nós, e o custo total de n inserções é $O(n \log n)$ — uma diferença drástica para grandes volumes.

4.2.2 Busca e remoção: onde o balanceamento é mais visível

No arquivo degenerado com 1.000 elementos, a busca pela folha mais profunda passou de **74.800 ns** na árvore comum para **2.100 ns** na AVL — redução de 97%. O motivo é o mesmo: na árvore degenerada a folha mais profunda está no nível 999, exigindo 999 comparações; na AVL essa mesma folha está no nível $\lfloor \log_2 1000 \rfloor \approx 10$, exigindo no máximo 10 comparações.

Esse comportamento confirma empiricamente a análise matemática da Seção 2: a complexidade de busca na `ArvoreBinaria` é $O(n)$ no pior caso (arquivo degenerado) e $O(\log n)$ no melhor caso (arquivo balanceado), enquanto na `ArvoreAVL` é $O(\log n)$ em **qualquer** caso.

4.2.3 Arquivo balanceado: resultados próximos entre as duas

Nos arquivos balanceados, as diferenças entre as duas implementações são pequenas e dentro da margem de variação natural do ambiente de execução. Por exemplo, na busca com 10.000 elementos:

- `ArvoreBinaria` (balanceado): 3.300 ns
- `ArvoreAVL` (balanceado): 17.600 ns

A AVL apareceu ligeiramente *mais lenta* nesse cenário específico. Isso ocorre porque, quando os dados chegam já em ordem balanceada, a `ArvoreBinaria` gera naturalmente uma árvore de altura mínima, sem precisar de nenhum rebalanceamento. A AVL, nesse mesmo cenário, ainda executa as verificações de fator de balanceamento a cada inserção — mesmo que não realize nenhuma rotação — e recalcula as alturas recursivamente, o que adiciona uma constante de processamento extra. Essa constante explica a diferença observada, que não representa uma desvantagem real, pois ambas permanecem em $O(\log n)$.

4.2.4 Custo do rebalanceamento

A implementação da `ArvoreAVL` neste trabalho não armazena a altura em cada nó: ela recalcula a altura recursivamente a cada chamada de `fb()` (fator de balanceamento). Isso significa que, para cada nó visitado durante a inserção, a função `fb()` percorre suas duas subárvores para calcular suas alturas. Esse custo adicional explica por que a AVL, mesmo em arquivos balanceados, apresenta tempos de inserção maiores que a árvore comum para tamanhos pequenos (onde o custo de balanceamento supera o benefício). Uma alternativa para eliminar esse custo seria armazenar a altura diretamente no campo do nó e atualizá-la incrementalmente — estratégia comum em implementações industriais.

4.2.5 Conclusão da comparação

Os dados empíricos confirmam a teoria: a **`ArvoreAVL` é amplamente superior à `ArvoreBinaria` em cenários com dados fora de ordem**, que é exatamente o caso mais comum em aplicações reais. No único cenário em que a árvore comum pode superar a AVL — dados já ordenados de forma balanceada — a diferença é pequena e desprezível, pois ambas operam em $O(\log n)$. Para qualquer aplicação que não possa garantir a ordem de chegada dos dados, a AVL é a escolha correta.

Tabela 7 – Resumo da comparação — Complexidade e comportamento observado

Cenário	**ArvoreBinaria**	**ArvoreAVL**	**Observação**
Inserção degenerada	$O(n^2)$ total	$O(n \log n)$ total	AVL até 98% mais rápida
Busca degenerada	$O(n)$	$O(\log n)$	AVL até 97% mais rápida
Inserção balanceada	$O(n \log n)$	$O(n \log n)$	Diferença pequena
Busca balanceada	$O(\log n)$	$O(\log n)$	Resultados equivalentes
`quantidadeNos()`	$O(1)$	$O(1)$	Idênticos

5 Estruturas de Árvores da Biblioteca Padrão do Java

5.1 TreeMap e TreeSet

A biblioteca padrão do Java (`java.util`) disponibiliza duas estruturas de coleções ordenadas baseadas em árvores binárias de busca:

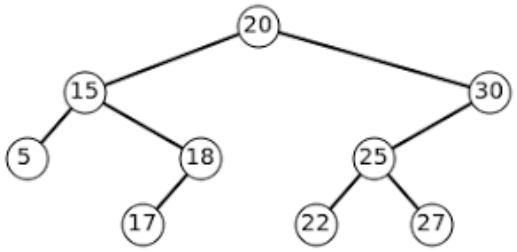


Figura 6 – TreeSet

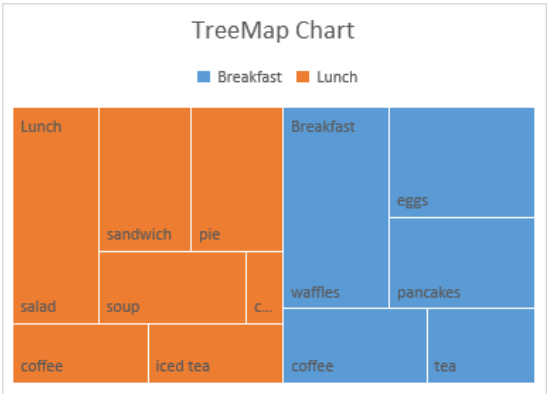


Figura 7 – TreeMap

- **`TreeSet<E>`** — conjunto ordenado de elementos únicos. Não permite duplicatas. Cada elemento é simultaneamente a chave de busca e o dado armazenado. É a estrutura mais próxima da `ArvoreBinaria<T>` desenvolvida neste trabalho.
- **`TreeMap<K,V>`** — mapa ordenado que associa uma chave (K) a um valor (V). Permite buscar um objeto pelo valor de uma chave explícita, sem precisar construir

o objeto completo. Internamente, o `TreeSet` é implementado como um `TreeMap` cujo valor é um objeto sentinela constante.

5.2 Estrutura de dados interna: Árvore Rubro-Negra

A biblioteca padrão do Java (`java.util`) disponibiliza a classe **`TreeSet<E>`** como implementação de conjunto ordenado baseada em árvore binária de busca. Internamente, o `TreeSet` é construído sobre um `TreeMap`, que utiliza uma **Árvore Rubro-Negra** (*Red-Black Tree*) — uma árvore binária de busca auto-balanceada que obedece às seguintes propriedades:

1. Todo nó possui uma cor: **vermelho** ou **preto**.
2. A **raiz** é sempre preta.
3. Toda **folha nula** (`null`) é considerada preta.
4. Um nó **vermelho** nunca possui filho vermelho (não há dois nós vermelhos consecutivos no mesmo caminho).
5. Todo caminho de qualquer nó até suas folhas nulas contém o mesmo número de nós pretos (*altura preta* uniforme).

Essas propriedades garantem que a altura da árvore seja sempre $O(\log n)$, independentemente da ordem de inserção dos elementos.

5.3 Complexidade das operações

Tabela 8 – Complexidade das operações do `TreeSet`

Operação	**Complexidade**
Inserção (`add`)	$O(\log n)$
Busca (`contains` / `floor`)	$O(\log n)$
Remoção (`remove`)	$O(\log n)$
Tamanho (`size`)	$O(1)$
Iteração em ordem	$O(n)$

Como a Rubro-Negra mantém altura $O(\log n)$ em qualquer cenário, não há distinção entre melhor e pior caso para as operações principais — ao contrário da `ArvoreBinaria` comum desenvolvida neste trabalho, cujo pior caso é $O(n)$ quando os dados chegam em ordem crescente.

5.4 Comparação com a biblioteca desenvolvida

Tabela 9 – Equivalência de métodos

Nossa biblioteca	**TreeSet**	**Observação**
`adicionar(T)`	`add(E)`	Equivalentes
`pesquisar(T)`	`floor(E)` + verificação	Requer lógica extra no TreeSet
`remover(T)`	`remove(E)`	Equivalentes
`quantidadeNos()`	`size()`	Equivalentes, ambos $O(1)$
`caminharEmOrdem()`	iterador / `forEach()`	Equivalentes
`caminharEmNivel()`	— não disponível	Apenas na nossa biblioteca
`altura()`	— não disponível	Apenas na nossa biblioteca

O `TreeSet` **não provê todos os métodos** da nossa biblioteca. Os métodos `caminharEmNivel()` e `altura()` não existem porque a Rubro-Negra encapsula completamente sua estrutura interna — detalhes como altura e organização por níveis não são expostos ao usuário.

Quanto às pesquisas e inserções: a inserção via `add()` é direta e equivalente ao `adicionar()` da nossa biblioteca. Já a busca não possui um `get()` explícito, pois o `TreeSet` é um *conjunto* — a premissa é que você já possui o objeto e quer verificar sua presença com `contains()`. Para recuperar o objeto completo a partir de um dado parcial (apenas o nome do aluno, por exemplo), é necessário usar o método `floor()`, que retorna o maior elemento considerado menor ou igual à chave pelo `Comparator`, seguido de uma verificação explícita de igualdade — algo que na nossa biblioteca é encapsulado diretamente pelo método `pesquisar()`.

5.5 Estratégia de comparação genérica

O `TreeSet` suporta duas estratégias de comparação:

5.5.0.1 `Comparator` externo

Passa-se um `Comparator<T>` ao construtor, definindo o critério de ordenação de fora da classe. Essa é a mesma estratégia adotada na `ArvoreBinaria` desenvolvida neste trabalho, e permite reutilizar os comparadores já existentes no projeto (`ComparadorAlunoNome`, `ComparadorAlunoMatricula`) sem alterar a classe `Aluno`.

5.5.0.2 `Comparable` na classe de domínio

Se nenhum `Comparator` for fornecido, o `TreeSet` exige que os elementos implementem a interface `Comparable<T>`, usando o método `compareTo()` como critério de ordenação. Tipos nativos como `String` e `Integer` já implementam `Comparable` e funcionam sem configuração adicional. Caso a classe não implemente `Comparable` e nenhum `Comparator` seja fornecido, uma `ClassCastException` é lançada em tempo de execução.

Essa dualidade torna o `TreeSet` **mais flexível** que a `ArvoreBinaria` desenvolvida neste trabalho, que exige *obrigatoriamente* um `Comparator` no construtor.