Prática 03 - AEDS 2

Henrique Freitas

October 2024

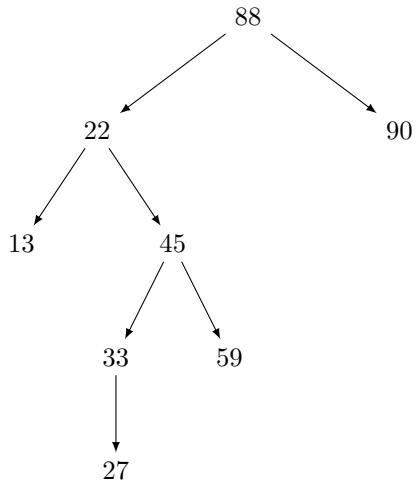
1 Questão 1: Construção de Árvores Binárias de Busca

1.1 Questão 1.1: Construa as árvores binárias e desenhe a estrutura da árvore após cada inserção

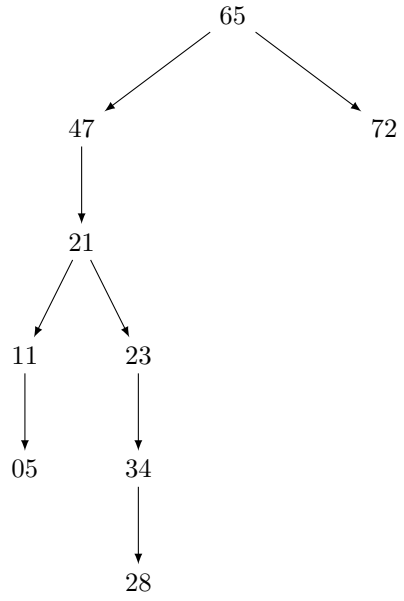
Considerando os conceitos de inserção, remoção, pesquisa e caminhamento em árvores binárias, resolvemos as seguintes questões com base nos conjuntos de dados apresentados:

1. Construa as árvores binárias de busca a partir dos conjuntos abaixo e desenhe a estrutura da árvore após cada inserção de k elementos.

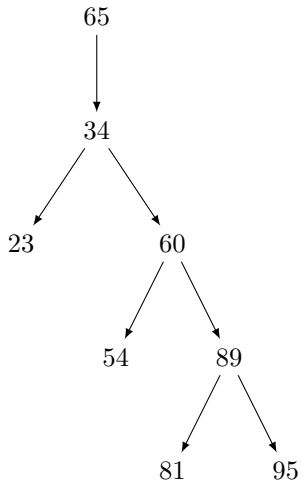
1.1.1 Árvore 1: {88, 22, 45, 33, 22, 90, 27, 59, 13}



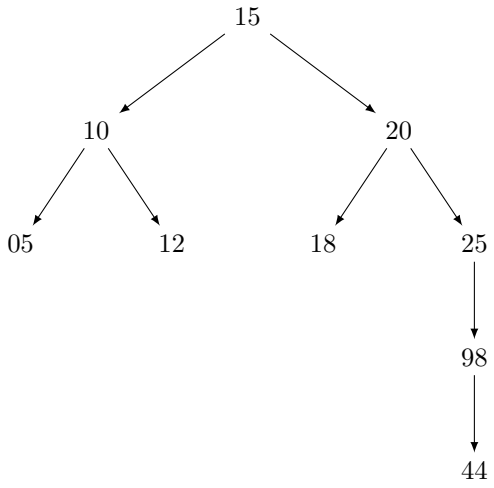
1.1.2 Árvore 2: {65, 47, 21, 11, 72, 23, 05, 34, 28}



1.1.3 Árvore 3: {65, 34, 89, 23, 60, 54, 81, 95, 39}



1.1.4 Árvore 4: {15, 10, 20, 05, 12, 18, 25, 98, 44}

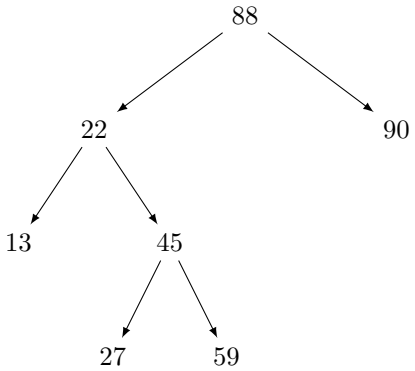


1.2 Questão 1.2: Remoção de Elementos em Árvores Binárias de Busca

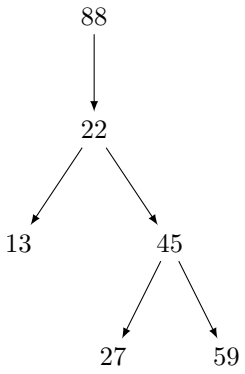
Nesta questão, realizamos a remoção dos elementos indicados e redesenhamos as árvores após cada remoção. Discutimos o impacto estrutural, abordando os diferentes casos de remoção (remoção de folha, remoção de nó com um filho e remoção de nó com dois filhos). Além disso, justificamos a escolha entre o sucessor in-ordem ou o predecessor in-ordem para os casos de remoção de nós com dois filhos.

1.2.1 Árvore 1: {33, 90, 33, 45}

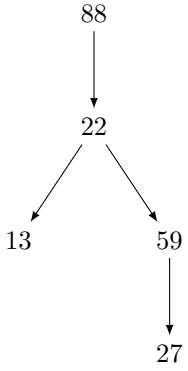
Remoção de 33: O nó 33 tem um filho (27). Ao removermos o nó com um filho, movemos o filho (27) para a posição de 33.



Remoção de 90: O nó 90 é uma folha, então sua remoção não impacta a estrutura da árvore.

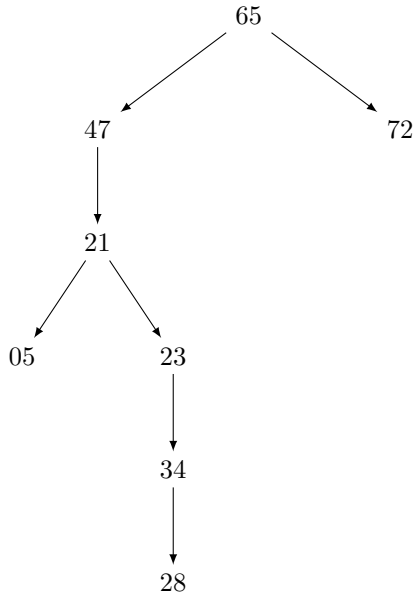


Remoção de 45: O nó 45 tem dois filhos (27 e 59). Para a remoção de um nó com dois filhos, escolhemos o sucessor, que é o nó 59, e substituímos 45 por 59.

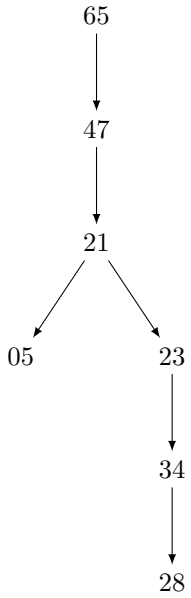


1.2.2 Árvore 2: {11, 72, 65, 23}

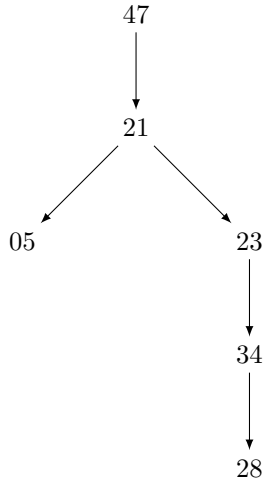
Remoção de 11: O nó 11 é uma folha. Sua remoção não afeta a estrutura da árvore.



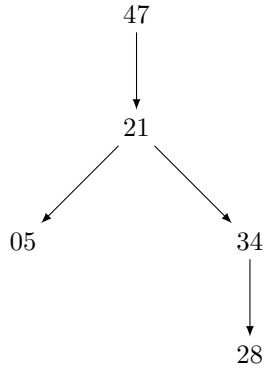
Remoção de 72: O nó 72 é uma folha, então a remoção não afeta a estrutura.



Remoção de 65: O nó 65 tem dois filhos (47 e 72). Escolhemos o sucessor in-ordem (47) para substituir 65.

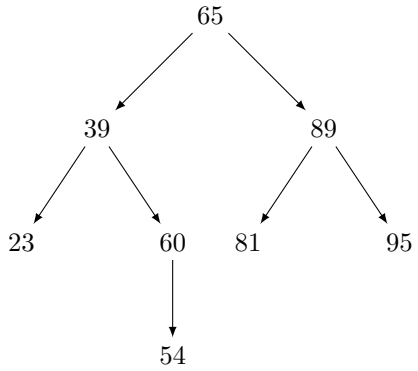


Remoção de 23: O nó 23 tem um filho (34). Removemos o nó e movemos 34 para a posição de 23.

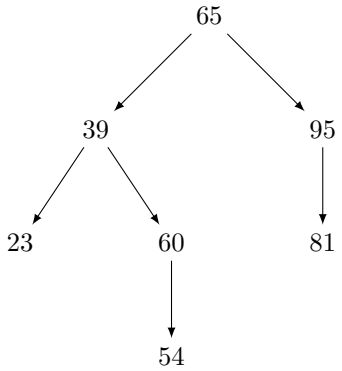


1.2.3 Árvore 3: {34, 89, 81, 95}

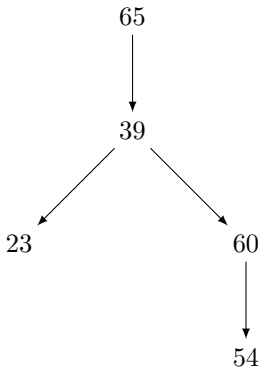
Remoção de 34: O nó 34 tem dois filhos. O sucessor in-ordem (39) é usado para substituir 34.



Remoção de 89: O nó 89 tem dois filhos (81 e 95). Escolhemos o sucessor in-ordem, que é o 95.

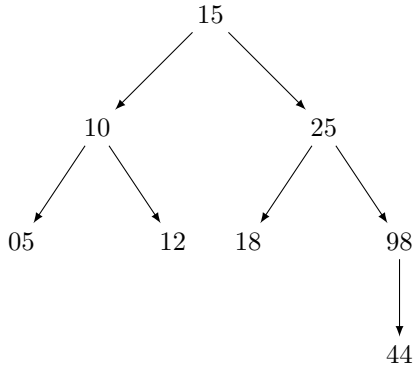


Remoção de 81 e 95: Ambas são folhas e são removidas sem impacto estrutural.

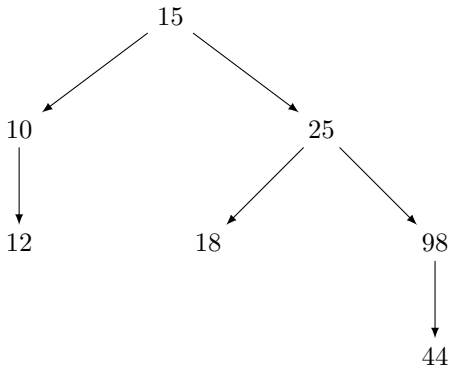


1.2.4 Árvore 4: {20, 05, 18, 44}

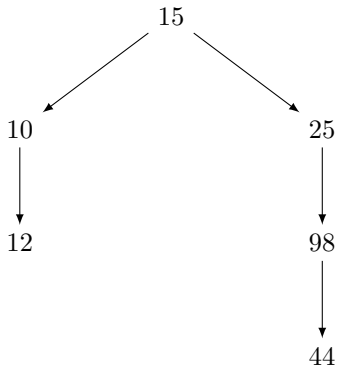
Remoção de 20: O nó 20 tem dois filhos (18 e 25). Para a remoção de um nó com dois filhos, escolhemos o sucessor in-ordem, que é o nó 25, para substituir o 20.



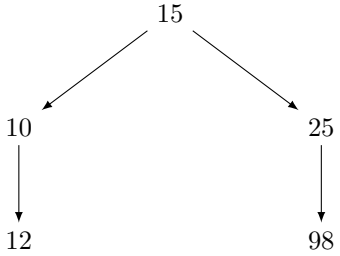
Remoção de 05: O nó 05 é uma folha, então sua remoção não afeta a estrutura da árvore.



Remoção de 18: O nó 18 é uma folha, então sua remoção não afeta a estrutura.



Remoção de 44: O nó 44 é uma folha, então sua remoção também não afeta a estrutura da árvore.

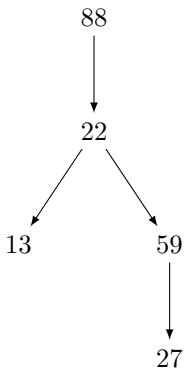


1.3 Questão 1.3: Pesquisa em Árvore com Diferentes Tipos de Caminhamentos

Nesta seção, utilizamos quatro tipos de caminhamento para localizar um elemento específico na árvore binária: pré-ordem, inordem, pós-ordem e em largura. Vamos analisar o número de interações e a ordem de visitação dos nós até encontrar o elemento selecionado.

1.3.1 Árvore de Exemplo

Abaixo está a representação gráfica da árvore binária utilizada para esta análise:



Vamos selecionar o nó **59** para ser localizado em cada um dos percorrimentos.

1.3.2 Percorrimento Inordem

No percorrimento inordem (ou em ordem), visitamos o nó mais à esquerda, depois a raiz e, em seguida, os nós à direita. A ordem de visitação dos nós na árvore seria:

```
13 → 22 → 27 → 59 → 88
```

Para localizar o elemento **59**, o caminho percorrido seria: 13, 22, 27, 59. Foram necessárias **4 interações** até encontrar o elemento.

1.3.3 Percorrimento Pós-ordem

No percorrimento pós-ordem, primeiro visitamos todos os filhos de um nó e, em seguida, o próprio nó. A ordem de visitação dos nós seria:

```
13 → 27 → 59 → 22 → 88
```

O caminho até o elemento **59** seria: 13, 27, 59. Foram necessárias **3 interações** para encontrar o elemento.

1.3.4 Percorrimento em Largura

O percorrimento em largura visita os nós em cada nível da árvore da esquerda para a direita. A ordem de visitação seria:

```
88 → 22 → 13 → 59 → 27
```

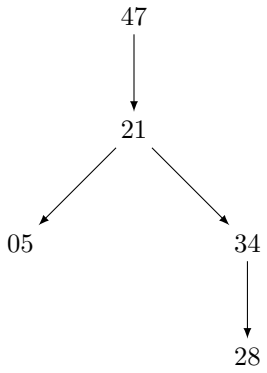
Neste caso, o elemento **59** foi encontrado após **4 interações**, com o caminho: 88, 22, 13, 59.

1.3.5 Eficiência dos Métodos

Podemos observar que os diferentes tipos de caminhamento apresentam diferentes números de interações. O percorrimento pós-ordem foi o mais eficiente para encontrar o elemento **59**, seguido pelo inordem e, por último, o em largura.

1.3.6 Percorrimento de Pesquisa na Árvore: Exemplo 2

Utilizaremos a seguinte árvore binária para realizar o estudo dos diferentes tipos de caminhamento. O nó escolhido para ser localizado será o **34**.



1.3.7 Percorrimento Inordem

No percorrimento inordem (ou em ordem), visitamos o nó mais à esquerda, depois a raiz e, em seguida, os nós à direita. A ordem de visitação dos nós na árvore seria:

```
05 → 21 → 28 → 34 → 47
```

Para localizar o elemento **34**, o caminho percorrido seria: 05, 21, 28, 34. Foram necessárias **4 interações** até encontrar o elemento.

1.3.8 Percorrimento Pós-ordem

No percorrimento pós-ordem, primeiro visitamos todos os filhos de um nó e, em seguida, o próprio nó. A ordem de visitação dos nós seria:

`05 → 28 → 34 → 21 → 47`

O caminho até o elemento **34** seria: 05, 28, 34. Foram necessárias **3 interações** para encontrar o elemento.

1.3.9 Percorrimento em Largura

No percorrimento em largura, visitamos os nós em cada nível da árvore da esquerda para a direita. A ordem de visitação seria:

`47 → 21 → 05 → 34 → 28`

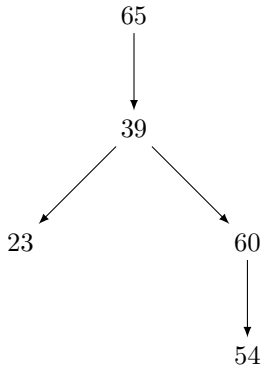
Neste caso, o elemento **34** foi encontrado após **4 interações**, com o caminho: 47, 21, 05, 34.

1.3.10 Eficiência dos Métodos

Analisando os diferentes tipos de caminhamento, observamos que o método pós-ordem é o mais eficiente para localizar o elemento **34**, seguido pelo inordem e, por último, o em largura.

1.3.11 Percorrimento de Pesquisa na Árvore: Exemplo 3

Utilizaremos a seguinte árvore binária para realizar o estudo dos diferentes tipos de caminhamento. O nó escolhido para ser localizado será o **60**.



1.3.12 Percorrimento Inordem

No percorrimento inordem (ou em ordem), visitamos o nó mais à esquerda, depois a raiz e, em seguida, os nós à direita. A ordem de visitação dos nós na árvore seria:

`23 → 39 → 54 → 60 → 65`

Para localizar o elemento **60**, o caminho percorrido seria: 23, 39, 54, 60. Foram necessárias **4 interações** até encontrar o elemento.

1.3.13 Percorrimento Pós-ordem

No percorrimento pós-ordem, primeiro visitamos todos os filhos de um nó e, em seguida, o próprio nó. A ordem de visitação dos nós seria:

`23 → 54 → 60 → 39 → 65`

O caminho até o elemento **60** seria: 23, 54, 60. Foram necessárias **3 interações** para encontrar o elemento.

1.3.14 Percorrimento em Largura

No percorrimento em largura, visitamos os nós em cada nível da árvore da esquerda para a direita. A ordem de visitação seria:

`65 → 39 → 23 → 60 → 54`

Neste caso, o elemento **60** foi encontrado após **4 interações**, com o caminho: 65, 39, 60.

1.3.15 Eficiência dos Métodos - Exemplo 3

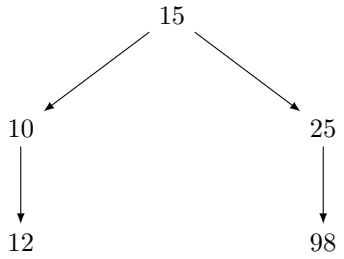
Exemplo 3 (Nó-alvo: 60)

- **Percorrimento Inordem**: Foram necessárias **4 interações**.
- **Percorrimento Pós-ordem**: Foram necessárias **3 interações**.
- **Percorrimento em Largura**: Foram necessárias **4 interações**.

No exemplo 3, o método **pós-ordem** foi o mais eficiente, localizando o nó **60** com apenas **3 interações**. O inordem e o em largura empataram, necessitando de **4 interações**.

1.3.16 Percorrimento de Pesquisa na Árvore: Exemplo 4

Utilizaremos a seguinte árvore binária para realizar o estudo dos diferentes tipos de caminhamento. O nó escolhido para ser localizado será o **98**.



1.3.17 Percorrimento Inordem

No percorrimento inordem (ou em ordem), visitamos o nó mais à esquerda, depois a raiz e, em seguida, os nós à direita. A ordem de visitação dos nós na árvore seria:

`10 → 12 → 15 → 25 → 98`

Para localizar o elemento **98**, o caminho percorrido seria: 10, 12, 15, 25, 98. Foram necessárias **5 interações** até encontrar o elemento.

1.3.18 Percorrimento Pós-ordem

No percorrimento pós-ordem, primeiro visitamos todos os filhos de um nó e, em seguida, o próprio nó. A ordem de visitação dos nós seria:

`12 → 10 → 98 → 25 → 15`

O caminho até o elemento **98** seria: 12, 10, 98. Foram necessárias **3 interações** para encontrar o elemento.

1.3.19 Percorrimento em Largura

No percorrimento em largura, visitamos os nós em cada nível da árvore da esquerda para a direita. A ordem de visitação seria:

`15 → 10 → 25 → 12 → 98`

Neste caso, o elemento **98** foi encontrado após **4 interações**, com o caminho: 15, 10, 25, 98.

1.3.20 Eficiência dos Métodos - Exemplo 4

Exemplo 4 (Nó-alvo: 98)

- **Percorrimento Inordem**: Foram necessárias **5 interações**.
- **Percorrimento Pós-ordem**: Foram necessárias **3 interações**.
- **Percorrimento em Largura**: Foram necessárias **4 interações**.

No exemplo 4, assim como no exemplo anterior, o método **pós-ordem** também foi o mais eficiente, localizando o nó **98** com **3 interações**. O percorrimento em largura foi intermediário, com **4 interações**, enquanto o inordem precisou de **5 interações**.

1.4 (Desafio adicional) Em cada árvore, identifique um subconjunto de elementos cujas re- moções resultem no maior número de rotações

1.4.1 Funcionamento do Código e Identificação de Subconjuntos de Nós com Maior Número de Rotações

O código apresentado é uma implementação de uma árvore AVL (AVLTree) em C++. A árvore AVL é uma árvore binária de busca auto-balanceada, onde a diferença de altura entre as subárvores esquerda e direita de qualquer nó é no máximo 1. Isso garante que as operações de inserção, remoção e busca sejam realizadas em tempo logarítmico.

Descrição do Código O trecho de código fornecido realiza as seguintes operações:

- Cria uma instância temporária de uma árvore AVL (`tempTree`) e clona a árvore original (`root`) para essa instância.
- Reseta o contador de rotações (`rotationCount`) para zero.
- Imprime a árvore antes da remoção de um nó específico (`key`).
- Remove o nó especificado (`key`) da árvore temporária e armazena o número de rotações realizadas durante a remoção.
- Imprime a árvore após a remoção do nó.
- Armazena o par {chave, número de rotações} em um vetor (`rotations`).

Identificação de Subconjuntos de Nós com Maior Número de Rotações Para identificar os subconjuntos de nós que resultam no maior número de rotações, o código realiza as seguintes etapas:

1. Insere uma série de valores na árvore AVL principal (`tree`) no `main`.
2. Para cada nó a ser removido, cria uma árvore temporária (`tempTree`) que é uma cópia da árvore original.
3. Remove o nó da árvore temporária e conta o número de rotações realizadas durante a remoção.
4. Armazena o número de rotações associadas a cada nó removido em um vetor (`rotations`).
5. O vetor `rotations` pode então ser analisado para determinar quais nós resultaram no maior número de rotações.

Exemplo de Uso No `main`, a árvore AVL é populada com os valores {88, 22, 45, 33, 22, 90, 27, 59, 13}. Para cada remoção de nó, o código imprime a árvore antes e depois da remoção e armazena o número de rotações realizadas. Esse processo permite identificar quais remoções causam o maior número de rotações, ajudando a entender o comportamento da árvore AVL em diferentes cenários de remoção.

```
AVLTree tree;
tree.insert(88);
tree.insert(22);
tree.insert(45);
tree.insert(33);
tree.insert(22);
tree.insert(90);
tree.insert(27);
tree.insert(59);
tree.insert(13);
```

O par `maxRotation` pode ser utilizado para armazenar o nó e o número máximo de rotações observadas durante as remoções, permitindo uma análise eficiente dos subconjuntos de nós com maior impacto na estrutura da árvore AVL.

Falhas do Programa Uma falha notável no programa é que ele não considera adequadamente os nós que resultam no mesmo número de rotações. O código apenas armazena o primeiro nó encontrado com o maior número de rotações e prossegue, ignorando outros nós que poderiam ter o mesmo impacto na estrutura da árvore AVL. Isso pode levar a uma análise incompleta ou incorreta dos nós que causam o maior número de rotações, já que nós adicionais com o mesmo número de rotações não são considerados. Para uma análise mais precisa, o programa deveria armazenar todos os nós que resultam no maior número de rotações e não apenas o primeiro encontrado.

Conclusão e Respostas Nas arvores de inserção sem remover previamente os numeros requisitados o conjuntos de nós com maiores numeros de rotações são:

1. Arvore 1 : 27.
2. Arvore 2 : 47,5
3. Arvore 3: 23,60
4. Arvore 4: 15,44

2 Questão 2: Programa Implementado em C++, projeto de análise geral de árvores BST

O programa apresentado simula o crescimento e as operações de inserção e remoção de nós em árvores binárias. O objetivo principal é comparar o desempenho de uma árvore binária desbalanceada com uma árvore binária equilibrada, ilustrando como a estrutura da árvore pode impactar o desempenho das operações de busca e inserção.

2.1 Estruturas de Dados

O programa utiliza a estrutura de árvore binária, representada pela estrutura `Node`, onde cada nó contém um valor (`data`) e dois ponteiros (`left` e `right`) que apontam para os filhos esquerdo e direito, respectivamente. As operações fundamentais de inserção (`insert`), remoção (`remove`) e cálculo da altura máxima (`calcNivelMaximo`) são implementadas para gerenciar a árvore.

2.2 Comparação entre Árvore Binária Desbalanceada e Árvore Binária Equilibrada

A principal diferença entre as duas árvores reside na maneira como os elementos são inseridos. Na árvore desbalanceada, os elementos são inseridos em ordem crescente, o que resulta em uma árvore com altura proporcional ao número de elementos ($O(n)$). Já na árvore equilibrada, os elementos são inseridos de forma a garantir um balanceamento mais eficiente, o que resulta em uma altura de $O(\log n)$, melhorando o desempenho da busca e das operações subsequentes.

2.2.1 Funcionamento do Programa

O programa realiza as seguintes operações:

- **Inserção de nós**: A função `insert` insere um valor na árvore, mantendo a ordem dos elementos.
- **Remoção de nós**: A função `remove` permite a remoção de um nó da árvore, realizando a reorganização dos nós, se necessário.
- **Cálculo da altura máxima**: A função `calcNivelMaximo` calcula a altura da árvore, que é um indicador de quão balanceada ela está.
- **Exibição do caminho mais longo**: A função `mostrarCaminhoMaisLongo` exibe o caminho mais longo da raiz até a folha mais distante.
- **Comparação de crescimento**: A função `compararCrescimentoArvores` compara o crescimento da altura das árvores desbalanceada e equilibrada, exibindo o impacto da estrutura na performance.
- **Sugestão de rotações**: A função `‘sugerirRotacoes‘ percorre a árvore binária e sugere rotações para balancear a árvore. Abaixo está a implementação da função em C++.`

2.2.2 Função sugerirRotacoes : Desafio Adicional

A função 'sugerirRotacoes' é utilizada para verificar o balanceamento de uma árvore binária e sugerir rotações para balanceá-la. O balanceamento é importante para garantir a eficiência das operações de busca, inserção e remoção.

Cálculo da Profundidade A função 'calcularProfundidade' determina a profundidade máxima de uma subárvore. Se o nó for nulo, a profundidade é zero. Caso contrário, a profundidade é 1 mais a profundidade máxima entre as subárvores esquerda e direita.

Verificação de Balanceamento A função 'sugerirRotacoes' percorre a árvore binária e calcula a profundidade das subárvores esquerda e direita de cada nó. Se a diferença de profundidade entre as subárvores for maior que 1, a árvore é considerada desbalanceada nesse nó.

Sugestão de Rotações Dependendo de qual subárvore é mais profunda, a função sugere uma rotação para balancear a árvore: - Se a subárvore esquerda é mais profunda, sugere uma rotação à direita. - Se a subárvore direita é mais profunda, sugere uma rotação à esquerda.

Essas rotações ajudam a manter a diferença de profundidade entre as subárvores esquerda e direita de qualquer nó dentro de um limite aceitável, geralmente 1, garantindo a eficiência das operações na árvore binária.

2.2.3 Impacto no Desempenho

A principal vantagem de uma árvore equilibrada é sua eficiência. Em uma árvore desbalanceada, as operações de inserção, remoção e busca podem ter um tempo de execução linear ($O(n)$) no pior caso, o que pode ser ineficiente para grandes volumes de dados. Por outro lado, uma árvore equilibrada garante que a altura seja limitada a $O(\log n)$, o que resulta em um desempenho muito mais rápido, especialmente quando o número de elementos aumenta.

Na `compararCrescimentoArvores`, a comparação do nível máximo das árvores após várias inserções revela a diferença de crescimento entre as duas estruturas. A tabela a seguir mostra como o nível máximo das árvores desbalanceada e equilibrada evolui à medida que os elementos são inseridos.

```
Insercao   Nivel Max Desbal   Nivel Max Equil   Depreciacao (%)
-------------------------------------------------------------
10         1                  0                  0%
20         2                  1                  50%
30         3                  1                  66.67%
40         4                  2                  100%
...
```

Como mostrado, a diferença de altura entre as árvores aumenta à medida que mais elementos são inseridos na árvore desbalanceada, resultando em uma maior *depreciação* no desempenho da árvore desbalanceada.

2.2.4 Ambiguidades no Desempenho

Embora o balanceamento da árvore melhore significativamente o desempenho em muitos casos, o comportamento pode ser ambíguo em situações específicas. Por exemplo, em uma árvore já balanceada ou em casos com dados homogêneos, a diferença de desempenho entre a árvore desbalanceada e a equilibrada pode ser menor. Ainda assim, a árvore equilibrada garante um desempenho previsível e eficiente para uma ampla gama de cenários.

2.3 Conclusão

A utilização de uma árvore binária equilibrada oferece vantagens claras em termos de desempenho, principalmente em grandes conjuntos de dados. O balanceamento da árvore garante que o tempo de busca e outras operações importantes permaneçam eficientes, com uma altura limitada a $O(\log n)$, enquanto a árvore desbalanceada pode levar a um desempenho de $O(n)$ no pior caso. Ao comparar as duas árvores, é possível observar as diferenças no crescimento da altura e como isso afeta o desempenho geral do programa.

3 Questão 3: Programa implementado em C++, projeto de dicionário com autosugestão

Este programa foi desenvolvido para realizar buscas eficientes em um dicionário, utilizando duas estruturas de árvore binária: a **AVL Tree** e a **Binary Search Tree (BST)**. A principal funcionalidade é permitir que o usuário digite um prefixo e, com isso, obtenha sugestões de palavras autocompletadas a partir de uma lista pré-carregada de palavras e seus significados. O programa também oferece uma busca por palavras, retornando o significado correspondente.

A seguir, descrevemos o funcionamento de cada parte do programa.

3.0.1 Estruturas de Dados

O programa utiliza duas estruturas principais para armazenar as palavras e seus significados: a **AVL Tree** e a **Binary Search Tree (BST)**. Ambas são árvores binárias de busca, mas a AVL Tree é auto-balanceada, o que proporciona um tempo de busca mais eficiente em casos de grandes volumes de dados.

3.0.2 AVL Tree

A **AVL Tree** é uma árvore binária de busca auto-balanceada, onde a diferença de altura entre as subárvores esquerda e direita de qualquer nó não pode ser superior a 1. Isso garante que as operações de inserção, busca e remoção ocorram em tempo logarítmico, mesmo no pior caso. O programa implementa a inserção de palavras na árvore e a busca por significados de forma eficiente.

3.0.3 Binary Search Tree (BST)

A **BST** é uma árvore binária onde, para cada nó, as palavras na subárvore esquerda são lexicograficamente menores, e as palavras na subárvore direita são lexicograficamente maiores. Embora a BST não seja balanceada como a AVL Tree, ela ainda proporciona uma busca eficiente, com tempo médio logarítmico.

3.0.4 Funcionalidade de Autocompletar

Uma das funcionalidades mais interativas do programa é o sistema de *autocomplete*, que sugere palavras conforme o usuário digita um prefixo. Quando o usuário começa a digitar, o programa coleta as sugestões de palavras que começam com o prefixo fornecido.

O sistema de autocomplete funciona da seguinte forma:

- O usuário começa a digitar um prefixo.
- O programa consulta a árvore escolhida (AVL ou BST) e coleta todas as palavras que começam com esse prefixo.
- O programa exibe essas sugestões ao usuário em tempo real.
- Caso o usuário pressione a tecla `TAB`, a primeira sugestão da lista de autocomplete é selecionada automaticamente para o prefixo, completando a palavra.
- Caso o usuário pressione `ENTER`, o programa realiza a busca pelo significado da palavra completa.
- O usuário também pode apagar caracteres pressionando a tecla `BACKSPACE`.

3.0.5 Busca de Palavras

Além do sistema de autocomplete, o programa permite que o usuário busque uma palavra completa. Quando o prefixo é completado (pressionando `ENTER`), o programa realiza uma busca pela palavra na árvore escolhida, retornando seu significado, caso a palavra seja encontrada. O tempo de busca também é exibido para que o usuário saiba a eficiência da operação.

3.0.6 Entrada de Dados

Os dados (palavras e seus significados) são carregados a partir de um arquivo de texto (`dicionario.txt`). O formato do arquivo é simples, com cada linha contendo uma palavra seguida de seu significado, separadas por dois pontos (`:`). Por exemplo:

```
palavra: significado
```

O programa suporta a leitura desse arquivo para preencher as árvores de palavras.

3.0.7 Interação com o Usuário

A interação com o usuário ocorre através de um terminal. O programa utiliza o modo de entrada sem precisar apertar `ENTER` após cada caractere, permitindo que o usuário veja as sugestões de palavras em tempo real. O terminal é configurado para captar caracteres pressionados diretamente (com exceção de `ENTER`, `ESC` e `TAB`).

A interface permite:

- Escolher a estrutura de dados (AVL ou BST).
- Digitar o prefixo e visualizar sugestões.
- Autocompletar palavras pressionando `TAB`.
- Buscar o significado de palavras pressionando `ENTER`.
- Apagar caracteres pressionando `BACKSPACE`.
- Sair do programa pressionando `ESC`.

3.0.8 Desempenho

A escolha da árvore **AVL** como estrutura de dados tem um impacto significativo no desempenho das operações de busca. A principal razão para a utilização da árvore AVL é seu balanceamento automático, o que garante que a altura da árvore seja sempre controlada, resultando em uma busca mais eficiente.

Árvore AVL versus Árvore Binária de Busca (BST) Na **Binary Search Tree (BST)**, a eficiência das operações de busca depende da profundidade da árvore. Se a árvore se tornar desbalanceada (por exemplo, se os elementos forem inseridos de forma ordenada), ela pode degenerar em uma lista ligada, com altura proporcional ao número de elementos ($O(n)$), tornando a busca linear. Esse comportamento leva a um tempo de busca ineficiente.

Por outro lado, a **AVL Tree** é uma árvore binária auto-balanceada, o que significa que, após cada inserção ou remoção, ela reorganiza seus nós para garantir que a diferença de altura entre a subárvore esquerda e a direita de qualquer nó não seja maior do que 1. Isso garante que a altura da árvore seja sempre proporcional ao logaritmo do número de elementos ($O(\log n)$), proporcionando uma busca mais rápida mesmo em grandes conjuntos de dados.

Exemplo de Código de Busca em Árvore AVL A seguir, um exemplo de como a busca é realizada em uma árvore AVL:

```
bool searchAVL(AVLNode root, const string& word) {
    if (!root) return false;  // Se a árvore está vazia, a palavra não foi encontrada.
    if (word == root->word) return true;  // A palavra foi encontrada.
    if (word < root->word) return searchAVL(root->left, word);  // Buscar na subárvore esquerda.
    return searchAVL(root->right, word);  // Buscar na subárvore direita.
}
```

Neste código, a busca começa pela raiz e, dependendo da comparação entre a palavra procurada e a palavra no nó atual, a busca é direcionada para a subárvore esquerda ou direita. Como a árvore é balanceada, o número de comparações realizadas é proporcional ao logaritmo da altura da árvore.

Impacto do Balanceamento no Desempenho O balanceamento de uma árvore AVL faz com que ela tenha sempre uma altura mínima em comparação a uma árvore binária de busca não balanceada. A operação de balanceamento mantém a árvore o mais próxima possível de uma árvore balanceada, resultando em um tempo de busca muito mais rápido em comparação à BST em casos de inserções sequenciais ou desordenadas.

A altura de uma árvore AVL é limitada a $O(\log n)$, enquanto a altura de uma BST desbalanceada pode crescer até $O(n)$ em casos desfavoráveis.

Cenários Ambíguos de Desempenho Apesar das vantagens de performance da árvore AVL, o tempo de busca pode ser ambíguo em alguns casos, dependendo do padrão de inserção dos dados. Por exemplo, ao inserir uma sequência de palavras aleatórias, o desempenho da árvore AVL será muito bom, já que ela garantirá um balanceamento adequado. No entanto, em casos muito específicos (como uma sequência de palavras já balanceadas ou com muitos dados semelhantes), a vantagem de desempenho pode ser menor.

Por outro lado, o comportamento de uma **BST** pode ser mais difícil de prever. Em casos de dados inseridos de forma não balanceada (por exemplo, em ordem crescente), a árvore degeneraria em uma lista ligada, tornando a busca tão lenta quanto $O(n)$, enquanto a árvore AVL ainda manteria um desempenho de $O(\log n)$.

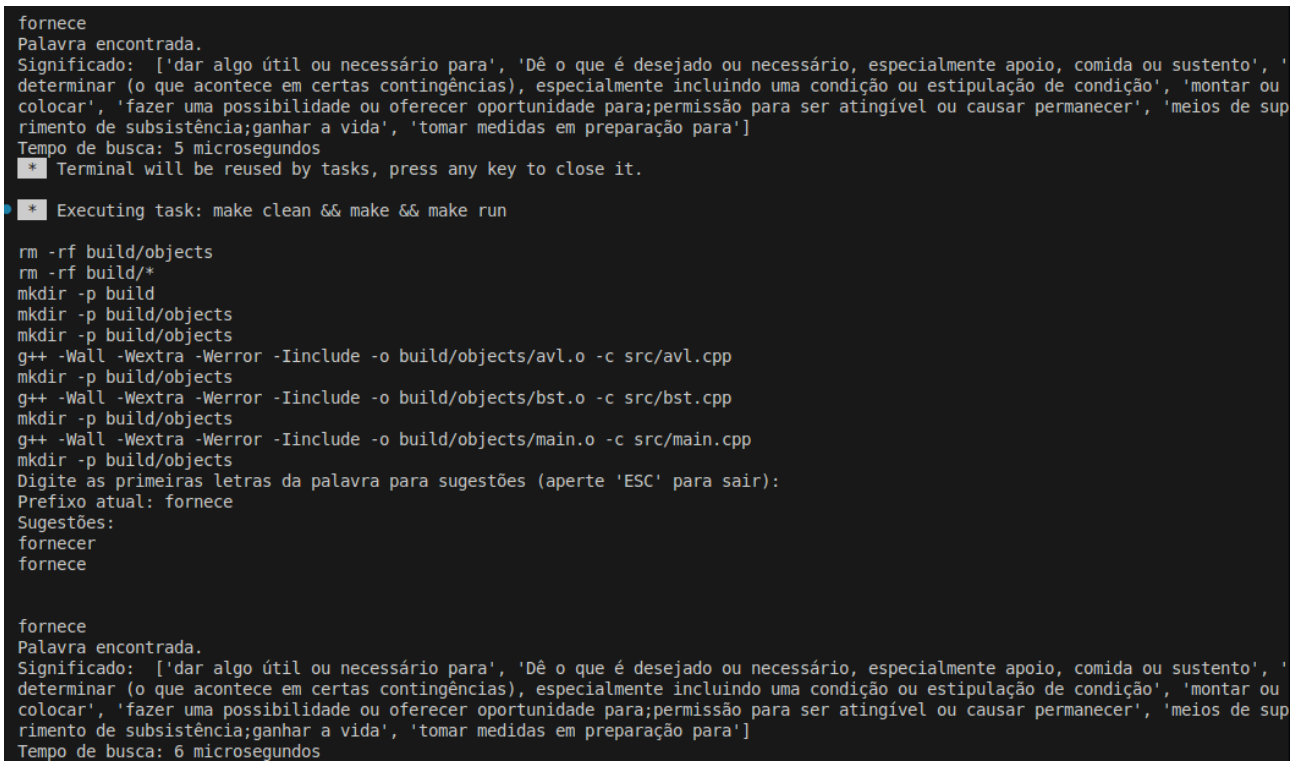


Figure 1: Comparação de desempenho entre Árvore AVL e Árvore Binária de Busca (BST). Acima AVL, Abaixo BST

Conclusão sobre o Desempenho Portanto, ao usar uma árvore AVL, garantimos um desempenho muito mais eficiente para grandes volumes de dados, especialmente quando comparado a uma árvore BST em situações onde a árvore pode se desbalancear. O balanceamento constante da árvore AVL mantém o tempo de busca $O(\log n)$, independentemente da ordem de inserção dos elementos, garantindo maior previsibilidade e eficiência na execução do programa.

3.0.9 Fluxo Geral do Programa

O fluxo do programa é o seguinte:

1. O usuário escolhe entre a AVL Tree ou a BST.
2. O programa carrega o arquivo de palavras e preenche a árvore escolhida.
3. O usuário começa a digitar um prefixo para buscar sugestões de palavras.
4. O sistema de autocomplete exibe as sugestões em tempo real.
5. O usuário pode pressionar `TAB` para autocompletar uma palavra ou `ENTER` para buscar seu significado.
6. O programa retorna o significado da palavra ou informa que a palavra não foi encontrada.

O programa termina quando o usuário pressiona `ESC`.