Prática № 1

Maíra Beatriz de Almeida Lacerda

Outubro 2024

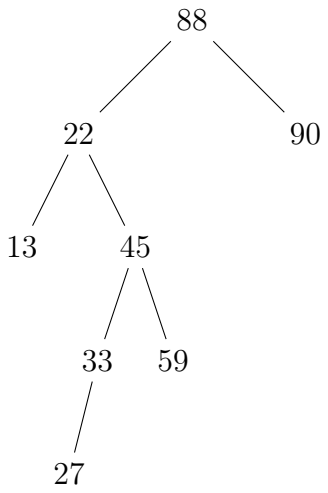
Centro Federal de Educação Tecnológica de Minas Gerais - Campus V
Engenharia da Computação

Disciplina: Algoritmos e Estruturas de Dados II
Professor: Michel Pires Dias

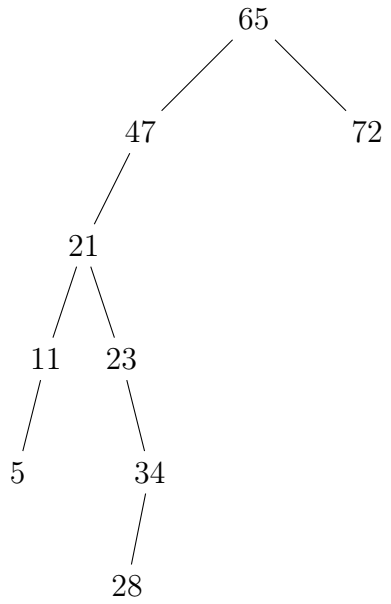
1 PROBLEMA 1

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

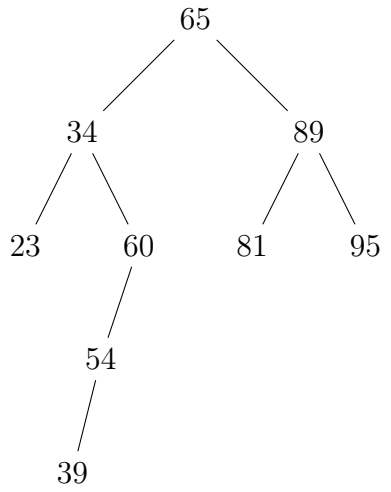
- Árvore 1: 88, 22, 45, 33, 22, 90, 27, 59, 13



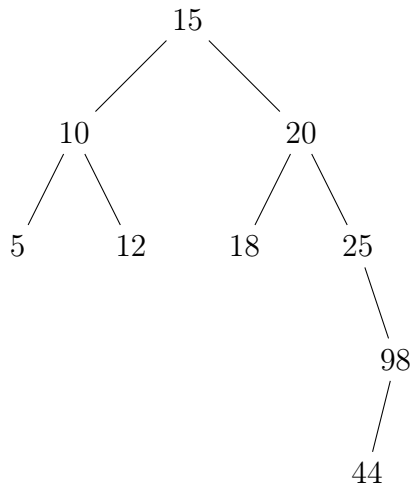
- Árvore 2: 65, 47, 21, 11, 72, 23, 05, 34, 28



- Árvore 3: 65, 34, 89, 23, 60, 54, 81, 95, 39



- Árvore 4: 15, 10, 20, 05, 12, 18, 25, 98, 44



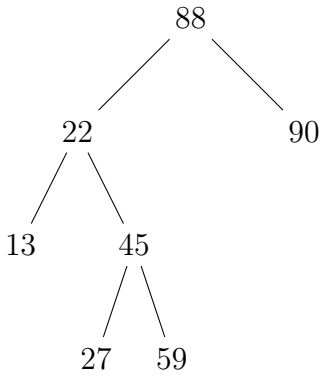
1.2 Realize a remoção dos elementos a seguir, redesenhando a árvore após cada remoção. Para cada remoção, discuta o impacto estrutural na árvore, abordando os diferentes casos de remoção (remoção de folha, nó com um filho e nó com dois filhos). Além disso, ao remover os nós com dois filhos, determine e justifique a escolha entre o sucessor in-ordem ou o predecessor in-ordem.

RESPOSTA

- Árvore 1: 33, 90, 33, 45

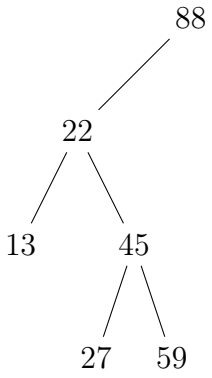
1. Remover 33

   - O nó 33 tem um único filho (27), então 27 é colocado no lugar do 33.



2. Remover 90

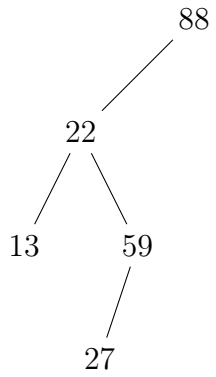
   - O nó 90 é um nó folha, então apenas remove ele.



3. Remover 33 (já foi removido anteriormente, então nada muda).

4. Remover 45

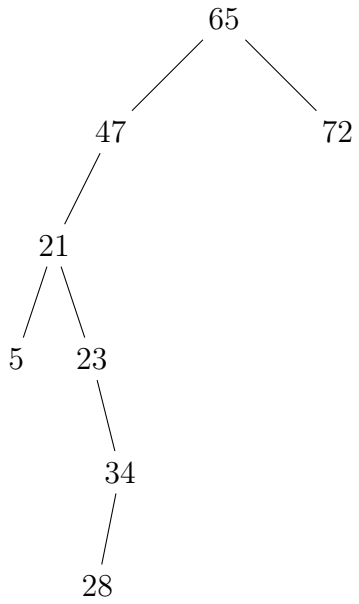
   - O nó 45 tem dois filhos (27 e 59). Para remover um nó com dois filhos, pode substituir 45 por seu sucessor in-ordem, que neste caso é 59 (o menor elemento na subárvore direita de 45).



- Árvore 2: 11, 72, 65, 23

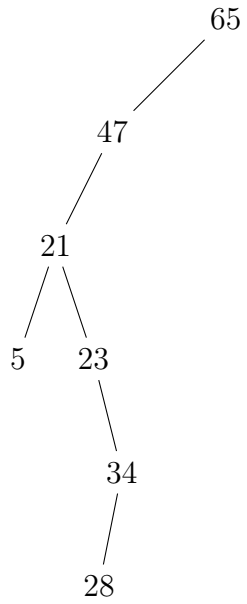
1. Remover 11

   - O nó 11 tem um único filho (5), então 5 é elevado para o lugar de 11.



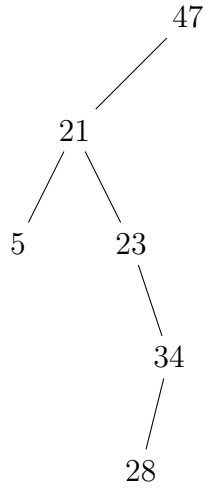
2. Remover 72

   - O nó 72 é uma folha, então apenas o remove.



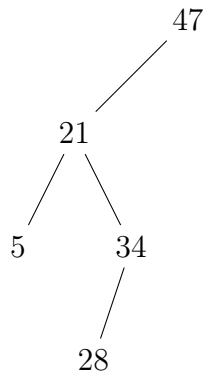
3. Remover 65

   - O nó 65 tem dois filhos (47 e 72, sendo que 72 já foi removido). Substitui o 65 pelo seu sucessor in-ordem, que é o menor elemento na subárvore direita de 65. Nesse caso, o sucessor in-ordem é 47.



4. Remover 23

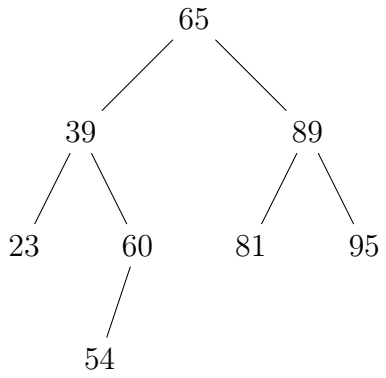
   - O nó 23 tem um único filho (34), então 34 é elevado para o lugar de 23.



- Árvore 3: 34, 89, 81, 95

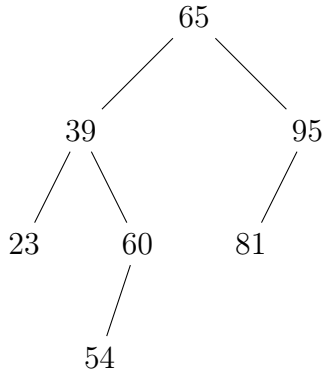
1. Remover 34

   - O nó 34 tem dois filhos. Para removê-lo, pode substituí-lo pelo sucessor in-ordem (39).



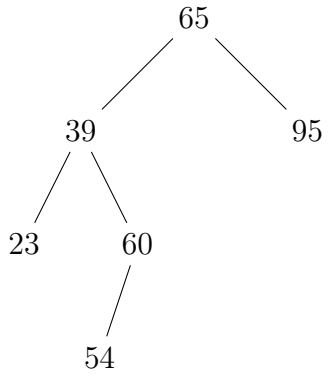
2. Remover 89

   - O nó 89 tem dois filhos. Substitui o 89 pelo sucessor in-ordem, que é 95 (o menor elemento da subárvore direita).



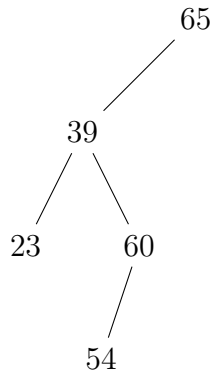
3. Remover 81

   - O nó 81 é uma folha, então apenas remove.



4. Remover 95

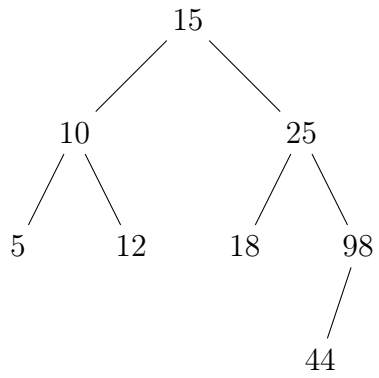
   - O nó 95 é uma folha, então apenas remove.



- Árvore 4: 20, 05, 18, 44

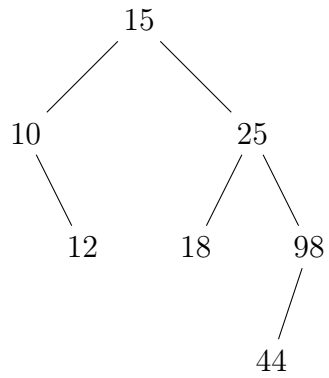
1. Remover 20

   - O nó 20 tem dois filhos. Substitui 20 pelo sucessor in-ordem (25).



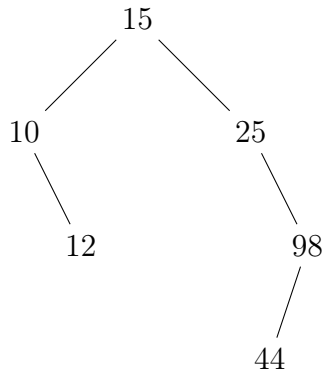
2. Remover 05

   - O nó 05 é uma folha, então apenas o remove.



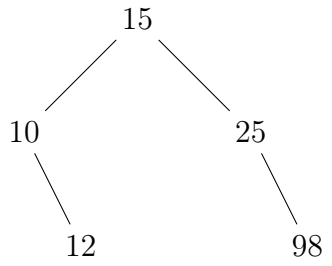
3. Remover 18

   - O nó 18 é uma folha, então apenas o remove.



4. Remover 44

   - O nó 44 é uma folha, então apenas o remove.



1.3 Após realizar todas as inserções e remoções, selecione um elemento específico em cada uma das árvores e utilize os quatro tipos de caminhamento apresentados em sala de aula como métodos de pesquisa para localizar o elemento escolhido. Para cada tipo de caminhamento, determine

- O número de interações necessárias com a estrutura da árvore para encontrar o elemento selecionado.

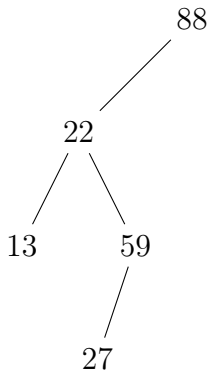
- A ordem de visitação dos nós até localizar o elemento, destacando o caminho percorrido.

- A eficiência de cada estratégia ao decorrer do processamento necessário para identificar o elemento desejado.

RESPOSTA

- Árvore 1: Elemento escolhido: 27

  Estrutura da árvore:



1. **Pré-ordem (Raiz → Esquerda → Direita)**
   - Número de interações: 5
   - Caminho percorrido: 88 → 22 → 59 → 27
   - Ordem de visitação (completa): 88 → 22 → 13 → 59 → 27
   - Eficiência: Média, pois o 27 foi encontrado após percorrer 4 nós, passando primeiro pela subárvore esquerda antes de encontrar o nó na subárvore direita.

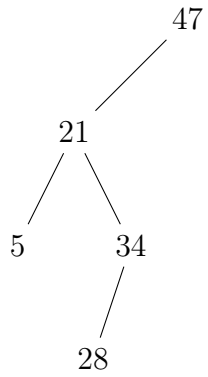
2. **In-ordem (Esquerda → Raiz → Direita)**
   - Número de interações: 5
   - Caminho percorrido: 13 → 22 → 27
   - Ordem de visitação (completa): 13 → 22 → 27 → 59 → 88
   - Eficiência: Alta, pois o 27 foi encontrado de forma eficiente, uma vez que o caminhamento em in-ordem visita os nós na ordem crescente, ideal para árvores binárias de busca.

3. **Pós-ordem (Esquerda → Direita → Raiz)**
   - Número de interações: 5
   - Caminho percorrido: 13 → 27
   - Ordem de visitação (completa): 13 → 27 → 59 → 22 → 88
   - Eficiência: Média, pois o 27 foi encontrado antes de passar pela raiz, mas a subárvore esquerda e direita foram totalmente percorridas.

4. **Em nível (Nível por nível)**
   - Número de interações: 5
   - Caminho percorrido: 88 → 22 → 59 → 27
   - Ordem de visitação (completa): 88 → 22 → 59 → 13 → 27
   - Eficiência: Média, pois o 27 foi encontrado no mesmo nível que outros nós, mas após visitar todos os nós do mesmo nível.

- Árvore 2: Elemento escolhido: 34

  Estrutura da árvore:



1. **Pré-ordem (Raiz → Esquerda → Direita)**
   - Número de interações: 4
   - Caminho percorrido: 47 → 21 → 5 → 34
   - Ordem de visitação (completa): 47 → 21 → 5 → 34 → 28
   - Eficiência: Moderada, pois o caminhamento em pré-ordem, é necessário visitar a subárvore esquerda completamente antes de localizar o nó 34, que está à direita de 21.

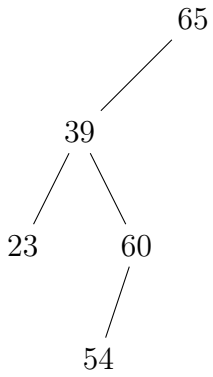
2. **- In-ordem (Esquerda → Raiz → Direita)**
   - Número de interações: 4
   - Caminho percorrido: 5 → 21 → 28 → 34
   - Ordem de visitação: 5 → 21 → 28 → 34 → 47
   - Eficiência: Alta, pois o caminhamento em in-ordem visita os nós na ordem crescente, o que é ideal para localizar elementos em uma árvore binária de busca. Neste caso, o nó 34 é localizado de forma eficiente após percorrer os nós menores.

3. **Pós-ordem (Esquerda → Direita → Raiz)**
   - Número de interações: 3
   - Caminho percorrido:5 → 28 → 34
   - Ordem de visitação: 5 → 28 → 34 → 21 → 47
   - Eficiência: Moderada, pois o nó 34 é encontrado antes de visitar a raiz (47), mas a subárvore esquerda e direita precisam ser completamente percorridas.

4. **Em nível (Nível por nível)**
   - Número de interações: 4
   - Caminho percorrido: 47 → 21 → 5 → 34
   - Ordem de visitação (completa): 47 → 21 → 5 → 34 → 28
   - Eficiência: Moderada, pois o nó 34 foi encontrado após visitar todos os nós no mesmo nível. A visitação por nível não prioriza a subárvore esquerda ou direita, o que pode tornar o processo menos eficiente em árvores balanceadas ou com nós mais profundos.

- Árvore 3: Elemento escolhido: 54

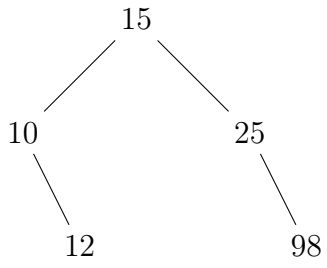
  Estrutura da árvore:



1. **Pré-ordem (Raiz → Esquerda → Direita)**
   - Número de interações: 5
   - Caminho percorrido: 65 → 39 → 23 → 60 → 54
   - Ordem de visitação: 65 → 39 → 23 → 60 → 54
   - Eficiência: Moderada, pois o caminhamento pré-ordem visita a raiz e as subárvores em sequência, o que resulta em passar por todos os nós antes de chegar ao nó 54.
2. **- In-ordem (Esquerda → Raiz → Direita)**
   - Número de interações: 3
   - Caminho percorrido: 23 → 39 → 54
   - Ordem de visitação (completa): 23 → 39 → 54 → 60 → 65
   - Eficiência: Alta, pois o caminhamento em in-ordem localiza o nó 54 rapidamente, ele é visitado de acordo com a ordem crescente dos nós, o que é ideal para árvores binárias de busca.
3. **Pós-ordem (Esquerda → Direita → Raiz)**
   - Número de interações: 2
   - Caminho percorrido: 23 → 54
   - Ordem de visitação (completa): 23 → 54 → 60 → 39 → 65
   - Eficiência: Alta, pois no pós-ordem, o nó 54 é encontrado antes de visitar as raízes superiores, o nó é localizado diretamente na subárvore direita de 39.
4. **Em nível (Nível por nível)**
   - Número de interações: 5
   - Caminho percorrido: 65 → 39 → 23 → 60 → 54
   - Ordem de visitação: 65 → 39 → 23 → 60 → 54
   - Eficiência: Moderada, pois o caminhamento em nível não prioriza a subárvore esquerda ou direita e visita todos os nós do mesmo nível antes de localizar o nó 54, resultando em um número maior de interações.

- Árvore 4: Elemento escolhido: 25

  Estrutura da árvore:



1. **Pré-ordem (Raiz → Esquerda → Direita)**
   - Número de interações: 4
   - Caminho percorrido: 15 → 10 → 12 → 25
   - Ordem de visitação (completa): 15 → 10 → 12 → 25 → 98
   - Eficiência: Moderada, pois o caminhamento pré-ordem requer percorrer a subárvore esquerda antes de visitar o nó 25 na subárvore direita.

2. **- In-ordem (Esquerda → Raiz → Direita)**
   - Número de interações: 4
   - Caminho percorrido: 10 → 12 → 15 → 25
   - Ordem de visitação (completa): 10 → 12 → 15 → 25 → 98
   - Eficiência: Alta, pois o caminhamento em in-ordem localiza o nó 25 de forma eficiente, o nó é visitado na ordem crescente de valor, adequada para árvores binárias de busca.

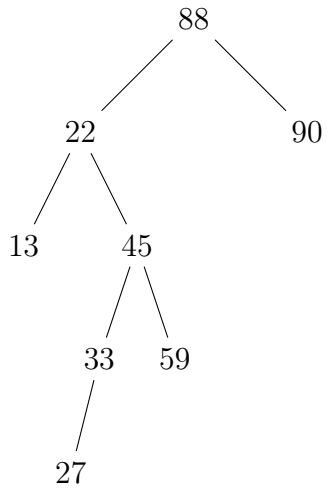
3. **Pós-ordem (Esquerda → Direita → Raiz)**
   - Número de interações: 4
   - Caminho percorrido: 12 → 10 → 98 → 25
   - Ordem de visitação (completa): 12 → 10 → 98 → 25 → 15
   - Eficiência: Moderada, pois o caminhamento pós-ordem requer percorrer completamente a subárvore esquerda e direita antes de chegar ao nó 25.

4. **Em nível (Nível por nível)**
   - Número de interações: 3
   - Caminho percorrido: 15 → 10 → 25
   - Ordem de visitação (completa): 15 → 10 → 25 → 12 → 98
   - Eficiência: Alta, pois o caminhamento em nível é eficiente, pois o nó 25 é encontrado rapidamente no segundo nível, após percorrer a raiz e a subárvore esquerda.

1.4 (Desafio adicional) Em cada árvore, identifique um subconjunto de elementos cujas remoções resultem no maior número de rotações (rebalanceamentos) da árvore. Esse desafio deve considerar os conceitos a serem apresentados em sala sobre árvores AVL.

RESPOSTA

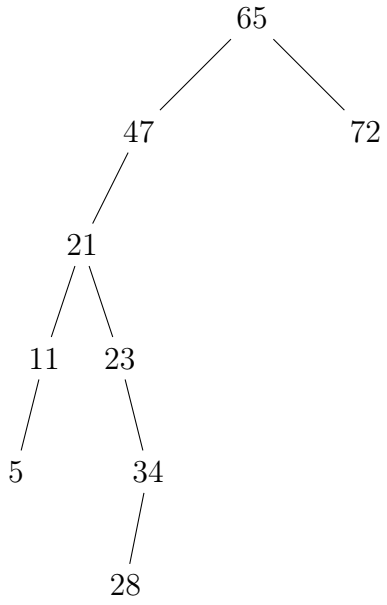
- Árvore 1: Remoções de 33, 45 e 59



Estrutura da árvore:

1. Remover 33: o nó 33 tem um filho (27). Após a remoção, 27 sobe, causando um leve desbalanceamento, mas sem rotações.
2. Remover 45: desbalanceia o nó 22. A remoção exige uma rotação à esquerda no nó 22.
3. Remover 22: após a remoção de 22, 45 (ou 27, dependendo da estratégia) ocupa sua posição, e o nó 88 exige uma rotação à direita.

- Árvore 2: Remoções de 23, 34 e 47.

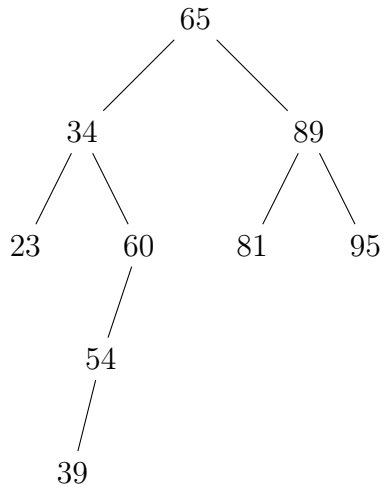


Estrutura da árvore:

1. Remover 23: causa o desbalanceamento em 21. Necessita de uma rotação à esquerda.
2. Remover 34: desbalanceia o nó 47, resultando em uma rotação dupla direita-esquerda.
3. Remover 47: após sua remoção, o nó 65 requer uma rotação à direita

- Árvore 3: Remoções de 39, 54 e 89.

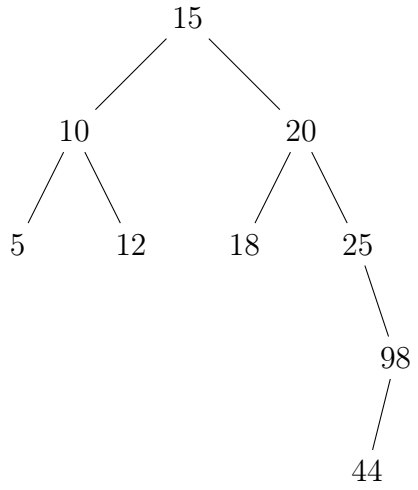
  Estrutura da árvore:



1. Remover 54: desbalanceia o nó 60, necessitando de uma rotação à esquerda.
2. Remover 39: desbalanceia o nó 60 novamente, necessitando de uma rotação direita-esquerda.
3. Remover 89: desbalanceia o nó 65, exigindo uma rotação à esquerda.

- Árvore 4: Remoções de 25, 44 e 98.

  Estrutura da árvore:



1. Remover 20: o sucessor 25 substitui 20, mas causa o desbalanceamento em 15, exigindo uma rotação à direita.
2. Remover 44: o nó 44 é removido sem impacto imediato na estrutura.
3. Remover 98: desbalanceia o nó 25, necessitando de uma rotação à esquerda.

2 PROBLEMA 2

Em árvores binárias, o nível máximo é frequentemente utilizado para compreender a profundidade da estrutura e o tempo necessário para percorrer a árvore em diferentes operações. O nível máximo, também chamado de altura da árvore, é definido como a distância (em termos de número de elementos ou nós) da raiz até a folha mais distante.

Neste exercício, você deverá elaborar uma função que não apenas calcule o nível máximo da árvore, mas que também apresente os seguintes desafios:

1. **Cálculo do Nível Máximo:** Implemente uma função que calcule o nível máximo de uma árvore binária sem balanceamento. A função deve percorrer toda a estrutura e identificar o nível da folha mais distante da raiz, retornando esse valor ao usuário.

2. **Visualização Interativa:** A cada nova inserção ou remoção de um nó, atualize e exiba o nível máximo da árvore, permitindo que o usuário visualize como a profundidade da árvore é impactada pela desbalanceamento natural da estrutura.

3. **Análise de Crescimento:** Considere dois conjuntos de inserções, um que gere uma árvore "torta"(mais desbalanceada) e outro que resulte em uma árvore mais equilibrada (embora sem ser balanceada automaticamente). Calcule e compare os níveis máximos dessas duas árvores ao longo de cada inserção, explicando por que certas inserções resultam em maiores níveis do que outras. Além disso, tente observar, se possível, se a prerrogativa de custo de 39% de depreciação de fato ocorre em uma árvore não balanceada em comparação com aquela que se mostra mais organizada.

4. **Caminho mais longo:** Após calcular o nível máximo da árvore, identifique e mostre ao usuário o caminho completo da raiz até a folha que define esse nível. Discuta como o desbalanceamento da árvore afeta o comprimento desse caminho em comparação com uma árvore idealmente balanceada.

- **Desafio adicional:** Implemente uma função que, dado o nível máximo calculado, sugira possíveis rotações ou reordenações que poderiam ser realizadas (manual ou hipoteticamente) para diminuir a profundidade da árvore, sem torná-la balanceada automaticamente. Discuta por que essas rotações são eficazes ou ineficazes dependendo da estrutura da árvore.

RESPOSTA

1. Parte 1: Cálculo do Nível Máximo (Altura da Árvore)

```
1
2  //Node.h
3  struct Node {
4      int val;
5      Node* left;
6      Node* right;
7      Node(int value) : val(value), left(nullptr), right(nullptr) {}
8  };
9
10 //Tree.h
11 int alturaArvore(Node* raiz) {
12     if (raiz == nullptr) {
13         return 0; // Arvore vazia tem altura 0
14     }
15     int alturaEsquerda = alturaArvore(raiz->left);
16     int alturaDireita = alturaArvore(raiz->right);
17     return 1 + std::max(alturaEsquerda, alturaDireita);
18 }
```

Listing 1: Cálculo da Altura Máxima

2. Parte 2.1: Inserção e Remoção com Atualização da Altura

```
1
2  //Tree.h
3  Node* inserir(Node* raiz, int valor) {
4      if (raiz == nullptr) {
5          return new Node(valor);
6      }
7      if (valor < raiz->val) {
8          raiz->left = inserir(raiz->left, valor);
9      } else {
10         raiz->right = inserir(raiz->right, valor);
11     }
12     return raiz;
13 }
14
15 Node* encontrarMinimo(Node* raiz) {
16     while (raiz->left != nullptr) {
17         raiz = raiz->left;
18     }
19     return raiz;
20 }
21
22 Node* remover(Node* raiz, int valor) {
23     if (raiz == nullptr) {
24         return raiz;
25     }
26     if (valor < raiz->val) {
27         raiz->left = remover(raiz->left, valor);
28     } else if (valor > raiz->val) {
29         raiz->right = remover(raiz->right, valor);
30     } else {
31     //No com um ou nenhum filho
32         if (raiz->left == nullptr) {
33             Node* temp = raiz->right;
34             delete raiz;
35             return temp;
36         } else if (raiz->right == nullptr) {
37             Node* temp = raiz->left;
38             delete raiz;
39             return temp;
40         }
41         //No com dois filhos
42         Node* temp = encontrarMinimo(raiz->right);
43         raiz->val = temp->val;
44         raiz->right = remover(raiz->right, temp->val);
45     }
46     return raiz;
47 }
```

Listing 2: Funções de Inserção e Remoção

- Visualização Interativa

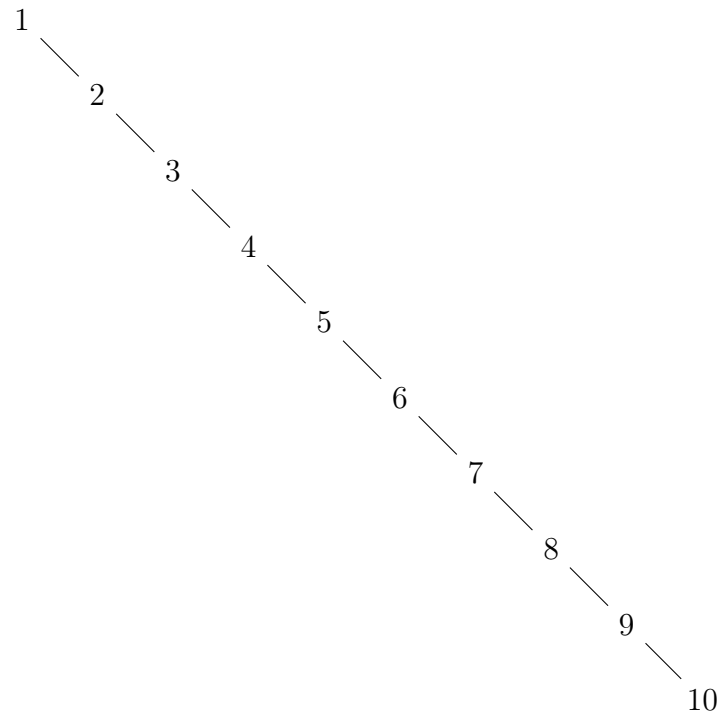
```
1  //main.cpp
2  void atualizarAltura(Node* raiz, const std::string& operacao, int
       valor) {
3      if (operacao == "inserir") {
4          raiz = inserir(raiz, valor);
```

```
    } else if (operacao == "remover") {
        raiz = remover(raiz, valor);
    }
    std::cout << "Altura da  rvore  apps " << operacao << " " <<
        valor << ": " << alturaArvore(raiz) << std::endl;
}

int main() {
    Node* raiz = nullptr;

    raiz = inserir(raiz, 15);
    raiz = inserir(raiz, 10);
    raiz = inserir(raiz, 20);
    raiz = inserir(raiz, 5);
    raiz = inserir(raiz, 12);
    raiz = inserir(raiz, 18);
    raiz = inserir(raiz, 25);
    raiz = inserir(raiz, 98);
    raiz = inserir(raiz, 44);

    atualizarAltura(raiz, "inserir", 8);
    atualizarAltura(raiz, "remover", 44);
    atualizarAltura(raiz, "remover", 25);

    return 0;
}
```

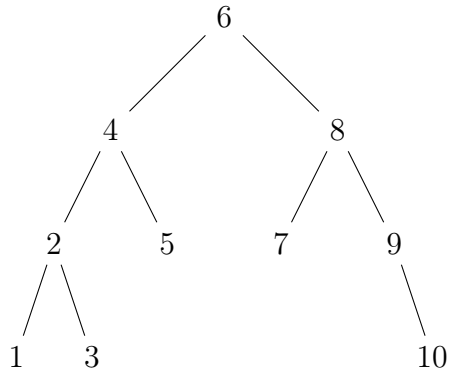
Listing 3: Visualização Interativa

3. Análise de Crescimento:

- Conjunto 1: árvore "torta"(mais desbalanceada)
  - [1, 2, 3, 4, 5, 6, 7, 8, 9, 10]



- Conjunto 2: árvore mais equilibrada
  - [6, 4, 8, 2, 5, 7, 9, 1, 3, 10]



Etapa de Inserção	Altura da Árvore Desbalanceada	Altura da Árvore Balanceada
1	1	1
2	2	2
3	3	2
4	4	3
5	5	3
6	6	3
7	7	4
8	8	4
9	9	4
10	10	4

Tabela 1: Comparação das Alturas das Árvores em Cada Etapa de Inserção

- Análise de Altura e Depreciação

Etapa de Inserção	Altura da Árvore Desbalanceada	Altura da Árvore Balanceada	Depreciação (%)
1	1	1	0.00
2	2	2	0.00
3	3	2	33.33
4	4	3	25.00
5	5	3	40.00
6	6	3	50.00
7	7	4	42.86
8	8	4	50.00
9	9	4	55.56
10	10	4	60.00

Tabela 2: Comparação das Alturas das Árvores e Depreciação Percentual

Observação sobre a Depreciação de 39%:

– Em alguns passos de inserção, como o 5º passo, a depreciação se aproxima de 39% (atingindo exatamente 40%).

- A depreciação é diretamente proporcional à diferença nas alturas das árvores e indica o impacto negativo de um desbalanceamento na eficiência.
- O custo de operações como busca, inserção e remoção em árvores desbalanceadas é significativamente maior devido à altura excessiva.

Conclusão
A análise demonstra que em árvores não balanceadas, o custo de operações cresce muito mais rapidamente, com depreciação percentual em relação à árvore balanceada superando 39% em diversos momentos. Isso reforça a importância de manter as árvores relativamente balanceadas para maior eficiência.

4. Parte 4: Caminho Mais Longo

```cpp
#include <vector>
//main.cpp

 std::vector<int> caminho, melhorCaminho;
    encontrarCaminhoMaisLongo(raiz, caminho, melhorCaminho);
    std::cout << "Caminho mais longo: ";
    for (int val : melhorCaminho) {
        std::cout << val << " ";
    }
    std::cout << std::endl;

    return 0;
}
```

Listing 4: Caminho Mais Longo

3 PROBLEMA 3

Imagine que você está desenvolvendo um dicionário eletrônico que permite aos usuários pesquisar rapidamente definições de palavras em um idioma específico. O desafio é projetar uma estrutura de dados eficiente, que permita buscas rápidas e economize espaço de armazenamento. Nesse contexto, elabore uma solução baseada em uma árvore binária de busca, com as seguintes características e requisitos adicionais:

- **Autocompletar e Sugestões Inteligentes:** Implemente recursos de autocompletar que, conforme o usuário digita as primeiras letras de uma palavra, sugira automaticamente termos correspondentes. A estrutura da árvore deve ser otimizada para permitir buscas rápidas e dinâmicas, retornando sugestões em tempo real. Discuta a eficiência do autocompletar utilizando a árvore binária e apresente uma análise comparativa em termos de tempo de busca para diferentes tamanhos de dicionário.

- **Desempenho e Otimizações:** Embora a árvore binária de busca ofereça vantagens de eficiência, ela pode se tornar desbalanceada à medida que mais palavras são inseridas. Discuta técnicas de otimização, como a utilização de árvores balanceadas (ex.: AVL, Red-Black) para garantir que a estrutura mantenha sua eficiência, mesmo com grandes volumes de dados. Proponha métodos de compactação ou armazenamento que minimizem o uso de memória sem comprometer o desempenho.

Além disso, elabore um conjunto de testes que simulem o uso do dicionário, com inserções e buscas de palavras, e avalie o tempo de resposta para diferentes volumes de dados. Utilize essas métricas para justificar as escolhas feitas na estrutura e nas otimizações aplicadas.

RESPOSTA

1. *Informações do Repositório*

   O código-fonte completo desta implementação pode ser acessado no seguinte repositório:

   GitHub: Pratica1-AEDSII

   - O dicionário começa com uma lista fixa de pelo menos 100 palavras reais em português. Essas palavras são utilizadas para garantir representatividade e diversidade no autocompletar, especialmente para tamanhos menores de dicionário.
   - Quando o tamanho do dicionário solicitado excede 100 palavras, o restante é preenchido com palavras geradas aleatoriamente, cada uma com 5 caracteres. Essa abordagem permite criar dicionários maiores mantendo eficiência no preenchimento.
   - O dicionário é capaz de lidar com tamanhos significativos, como 10^3 (1.000 palavras) ou 10^5 (100.000 palavras). Nesses casos, o dicionário é composto por palavras reais e aleatórias, garantindo eficiência e diversidade para suportar operações de autocompletar em diferentes cenários.

2. O autocompletar baseado em árvore binária de busca (BST) depende fortemente da estrutura da árvore e da ordem em que as palavras são inseridas. A eficiência da busca por palavras com prefixos específicos pode ser resumida como:

   Formato da Árvore:

   - **Caso Ideal:** Em uma árvore balanceada, a busca é realizada em $O(\log n)$, onde n é o número de palavras no dicionário.
   - **Caso Pior:** Em uma árvore desbalanceada, a busca pode degradar para $O(n)$, especialmente se a árvore for linearizada (exemplo: inserções já ordenadas).

3. Eficiência por Tamanho do Dicionário

   - **Dicionário Pequeno ($n = 100$):** O impacto do balanceamento é pequeno, pois mesmo no caso pior, a profundidade da árvore é baixa.
   - **Dicionário Médio ($n = 10^3$):** A eficiência começa a depender do balanceamento. Em um cenário ideal, a busca permanece eficiente ($O(\log n)$).
   - **Dicionário Grande ($n = 10^5$):** O desbalanceamento afeta significativamente a performance, degradando o tempo de busca para $O(n)$ em uma árvore desbalanceada. Estruturas balanceadas mantêm eficiência ($O(\log n)$).

Tamanho do Dicionário (n)	**Estrutura**	**Tempo de Inserção (O)**	**Tempo de Busca (O)**	**Observação**
100	Árvore Binária	$O(\log n)$	$O(\log n)$	Desempenho aceitável.
1.000	Árvore Binária	$O(n)$	$O(n)$	Desbalanceamento perceptível.
1.000	AVL	$O(\log n)$	$O(\log n)$	Mantém desempenho eficiente.
100.000	Árvore Binária	$O(n)$	$O(n)$	Inserções e buscas muito lentas.
100.000	Red-Black	$O(\log n)$	$O(\log n)$	Escalabilidade eficiente.

Tabela 3: Comparação de tempo para diferentes tamanhos de dicionário

4. Análise Comparativa de Tempo de Busca

5. Técnicas de Otimização

   - Utilização de Árvores Balanceadas
     Para garantir eficiência em grandes volumes de dados, árvores balanceadas como AVL e Red-Black podem ser usadas:
     - **Árvores AVL:** Mantêm o balanceamento após cada operação e são mais eficientes para cenários onde a busca é mais frequente que a inserção.
     - **Árvores Red-Black:** Oferecem inserções e remoções mais rápidas em comparação às AVL, sendo mais indicadas para cenários com alta taxa de atualização.
   - Compactação e Minimização de Memória
     - **Tries (Árvores de Prefixo):** Estruturas otimizadas para armazenar palavras compartilhando prefixos, reduzindo redundâncias.
     - **Alocação Dinâmica:** Gerenciamento eficiente da memória liberando nós desnecessários.
     - **Representação em Disco:** Serializar a árvore para economizar memória e carregar em memória conforme necessário.

6. Conjunto de Testes Os testes foram realizados com inserção e busca nos seguintes tamanhos de dicionários:

   - Pequeno ($n = 100$).
   - Médio ($n = 10^3$).
   - Grande ($n = 10^5$).

7. Justificativa das Escolhas

   - A árvore binária de busca é uma boa solução para dicionários pequenos e médios. No entanto, para grandes volumes de dados ($n > 10^3$), a utilização de árvores balanceadas é essencial para manter a eficiência.
   - Árvores AVL são indicadas para buscas mais frequentes, enquanto Red-Black são melhores em cenários de alta taxa de inserção.
   - Implementar tries pode ser uma alternativa eficiente para operações de auto-completar, pois compartilham prefixos e reduzem o espaço necessário.