Questão 1  
  
Eu escolhi a Árvore de Huffman, que é muito usada para compressão de dados, como em arquivos ZIP ou imagens JPEG. Ela funciona dando códigos menores para os símbolos mais frequentes, o que ajuda a reduzir o tamanho dos dados.

A construção é simples: sempre junta os dois símbolos com menor frequência até formar uma única árvore. O caminho até cada símbolo vira seu código binário.

Diferente das árvores binárias comuns, ela não se preocupa com ordenação nem busca rápida, e sim em otimizar o espaço. É um exemplo de como um algoritmo simples pode resolver um problema prático de forma eficiente.

Questão 2  
  
Primeiro, a função eh\_arvore\_binaria recebe uma lista de arestas, ou seja, conexões entre pares de nós.  
Ela então cria três listas principais:

* filhos: para guardar os filhos de cada nó.
* todos\_os\_nos: para registrar todos os nós que aparecem.
* pais: para saber quem já foi filho de alguém.

A primeira verificação que ela faz é se algum nó tem mais de dois filhos, o que não pode acontecer numa árvore binária.  
Se acontecer, a função já retorna False.

Depois, ela vê se um mesmo nó aparece como filho de mais de um pai.  
Isso cria conflito, e também não é permitido em árvores binárias.

Aí vem a parte de verificar a raiz:  
Ela verifica se existe apenas um único nó que nunca foi filho de ninguém.  
Esse nó é considerado a raiz da árvore.  
Se tiver mais de uma raiz ou nenhuma, a estrutura já não é válida.

Por fim, ela faz uma busca em largura:  
Começa pela raiz e vai visitando todos os nós conectados.  
Se algum nó ficar de fora, significa que a estrutura não está conectada por completo — então não é uma árvore.

No final, se todas essas condições forem respeitadas, a função retorna True, dizendo que a estrutura parece sim uma árvore binária!  
Caso contrário, retorna False.

Na parte de entrada, a gente digita os pares de arestas (pai filho), e digita “fim” quando quiser parar.  
A função é chamada com esses dados e já mostra o resultado no terminal."  
  
exemplo de execução  
1 2

1 3

2 4

2 5

Questão 3  
  
Uma árvore binária de busca (ou ABB) é uma estrutura onde cada nó tem no máximo dois filhos, e os valores são organizados da seguinte forma: à esquerda ficam os valores menores e à direita, os maiores. Isso facilita a busca, inserção e remoção de dados de forma eficiente  
  
Eu criei uma classe chamada Node, que representa cada nó da árvore. Ele guarda o valor e tem ponteiros para os filhos esquerdo e direito."

🗣️ "Depois, temos a classe ArvoreBinariaBusca, que implementa os métodos de inserção e exclusão. A inserção segue a regra da árvore binária de busca: se o valor for menor, vai para a esquerda; se for maior, vai para a direita. A exclusão trata os três casos principais: quando o nó é folha, quando tem um filho, e quando tem dois filhos."

🗣️ "Também criei o método em\_ordem, que percorre a árvore e retorna os valores em ordem crescente.  
  
Na parte visual, eu utilizei o Tkinter para construir uma interface simples. O usuário pode digitar um valor, clicar em 'Inserir' ou 'Excluir', e ver o resultado tanto em forma de texto quanto em forma gráfica."

🗣️ "Para isso, eu usei o matplotlib junto com o networkx, que permite gerar dinamicamente o grafo da árvore. A cada operação, a árvore é redesenhada.

Por exemplo, se eu inserir os valores 50, 30 e 70, a árvore começa a se formar com o 50 como raiz, 30 à esquerda e 70 à direita. Se eu clicar em 'Mostrar em Ordem', o sistema exibe: 30, 50, 70."

🗣️ "Se eu excluir o 30, ele é removido e a árvore se reorganiza automaticamente, e isso já aparece na visualização.

Questão 4  
  
As árvores AVL são um tipo especial de árvore binária de busca que se mantém balanceada automaticamente. Isso é importante porque se a árvore ficar muito desbalanceada, as operações como inserção, busca ou remoção podem ficar lentas.

Então, sempre que eu insiro um novo valor, o código calcula o fator de balanceamento — que é a diferença de altura entre o lado esquerdo e o direito do nó. Se essa diferença for maior que 1 ou menor que -1, a árvore faz uma rotação.

Existem quatro tipos de rotação que eu implementei no código:

1. Rotação simples à direita, quando o valor é inserido na esquerda da subárvore esquerda.
2. Rotação simples à esquerda, quando o valor vai pra direita da subárvore direita.
3. Rotação dupla à direita, que acontece quando o valor é inserido à direita do filho da esquerda.
4. Rotação dupla à esquerda, que é quando o valor vai pra esquerda do filho da direita.

No código, cada vez que eu insiro um valor, ele verifica automaticamente se precisa fazer alguma dessas rotações. A rotação reorganiza os nós da árvore mantendo ela sempre equilibrada.

Eu testei esse código com a sequência: 30, 10, 20, 50, 70, 15, 12, 56. O próprio código já aplica as rotações necessárias e no final eu uso uma função em\_ordem que mostra os elementos em ordem crescente, confirmando que a árvore está organizada corretamente.

Esse balanceamento automático é o que garante que a árvore AVL continue com um desempenho eficiente, mesmo com muitos dados sendo inseridos.