Centro Federal de Educação Tecnológica de Minas Gerais
Engenharia de Computação
Algoritmos e Estruturas de Dados II

Lista 1:
Atividade sobre árvores

Guilherme Alvarenga de Azevedo

Divinópolis - MG
Novembro de 2024

Sumário

Questão 1 2

Questão 2 7

Questão 3 9

Questão 1

Pergunta

Considerando os conceitos de inserção, remoção, pesquisa e caminhamento em árvores binárias, resolva 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.

    - 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}

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.

    - Árvore 1: {33, 90, 33, 45}
    - Árvore 2: {11, 72, 65, 23}
    - Árvore 3: {34, 89, 81, 95}
    - Árvore 4: {20, 05, 18, 44}

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.

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

As respostas dos itens 1, 2 e 3 estão no PDF no git que pode ser acessado em: <https://github.com/alvarengazv/trabalhosAEDSII.git>. No diretório "lista1-aeds2-ex1/exercicio1-arvores.pdf".

Para o item 3, sobre os tipos de caminhamento:

1. Caminhamento Pré-Ordem

Definição: O nó atual é visitado primeiro, seguido pelo caminhamento na subárvore esquerda e, por fim, na subárvore direita. A ordem pode ser descrita como:

$$\text{Visita a raiz} \rightarrow \text{Subárvore esquerda} \rightarrow \text{Subárvore direita.}$$

Exemplo: Para uma árvore com a estrutura:

$$\begin{array}{c} A \\ /\backslash \\ B \quad C \end{array}$$

A pré-ordem seria: $A \rightarrow B \rightarrow C$.

Eficiência:

- Útil para copiar uma árvore ou gerar expressões prefixadas.
- A eficiência depende do formato da árvore:
  - Árvores balanceadas: mais eficiente na varredura completa.
  - Árvores degeneradas (parecidas com listas): menos eficiente.

2. Caminhamento Central (In-ordem)

Definição: Primeiro, visita-se a subárvore esquerda, depois o nó atual, e, por fim, a subárvore direita. A ordem pode ser descrita como:

$$\text{Subárvore esquerda} \rightarrow \text{Visita ao nó} \rightarrow \text{Subárvore direita.}$$

Exemplo: Para a mesma árvore:

$$\begin{array}{c} A \\ /\backslash \\ B \quad C \end{array}$$

O central seria: $B \rightarrow A \rightarrow C$.

Eficiência:

- Ideal para árvores binárias de busca (*Binary Search Trees - BSTs*), pois retorna os elementos em ordem crescente ou decrescente.
- Se a árvore não estiver balanceada, pode ter desempenho semelhante a uma busca linear em uma lista encadeada.

3. Caminhamento Pós-Ordem

Definição: Visita-se primeiro a subárvore esquerda, depois a direita, e só então o nó atual. A ordem pode ser descrita como:

$$\text{Subárvore esquerda} \rightarrow \text{Subárvore direita} \rightarrow \text{Visita ao nó.}$$

Exemplo: Para a mesma árvore:

$$\begin{array}{c} A \\ /\backslash \\ B \;\; C \end{array}$$

O pós-ordem seria: $B \rightarrow C \rightarrow A$.

Eficiência:

- Útil para aplicações como cálculo de expressões pós-fixadas (notação polonesa reversa).
- O desempenho segue as mesmas características das pré-ordem e central: depende do balanceamento da árvore.

4. Busca em Largura

Definição: Os nós são visitados nível por nível, começando pela raiz. Os nós no mesmo nível são processados antes de descer para o próximo nível.

Exemplo: Para a mesma árvore:

$$\begin{array}{c} A \\ /\backslash \\ B \;\; C \end{array}$$

O BFS seria: $A \rightarrow B \rightarrow C$.

Eficiência:

- Mais adequada para localizar o nó mais próximo da raiz que atenda a uma condição específica.
- Pode ser menos eficiente em termos de memória, já que exige uma fila para armazenar os nós a serem visitados, especialmente em árvores muito grandes.

Comparação de Eficiência

A eficiência de cada estratégia depende de fatores como:

1. **Estrutura da Árvore**:
   - Árvores balanceadas permitem buscas mais rápidas e caminhamentos mais eficientes.
   - Árvores degeneradas têm comportamento próximo ao de uma lista encadeada, tornando o processamento mais lento.
2. **Objetivo do Caminhamento**:
   - Para *identificar um elemento específico*:
     - BFS é mais eficiente se o elemento está próximo da raiz.
     - Pré-ordem, central e pós-ordem podem ser ineficientes, pois visitam nós desnecessários antes de atingir o desejado.
   - Para *varreduras completas* ou *geração de ordem específica*:
     - Central é ideal para árvores binárias de busca.
     - Pós-ordem é eficiente em cálculos acumulativos ou para deletar nós de uma árvore.
3. **Complexidade de Tempo**:
   - Para todos os métodos, no pior caso, a complexidade é $O(n)$, onde n é o número de nós na árvore, já que todos os nós podem ser visitados.
4. **Complexidade de Espaço**:
   - Pré-ordem, Central, Pós-ordem utilizam memória proporcional à altura da árvore ($O(h)$, onde h é a altura).
   - Busca em Largura pode consumir mais memória, $O(w)$, onde w é o número máximo de nós em um nível.

Conclusão

A escolha do caminhamento depende das necessidades do problema e da estrutura da árvore. Em árvores bem balanceadas, qualquer método pode ser eficiente para buscas completas. Para encontrar elementos específicos, a busca em largura pode ser preferível em certos cenários, como quando se necessita descobrir a altura da árvore. Já o central, ou in-ordem, por exemplo, é vantajoso em casos em que é necessário caminhar pelos elementos em ordem crescente.

Já para o desafio adicional, deveria se considerar a árvore AVL desde as inserções para se ter uma melhor ideia. Porém, para quaisquer uma das árvores, independentemente da altura total ou balanceamentos anteriores, qualquer conjunto de valores que se encontram "mais acima" na árvore e que tenham filhos que também possuam filhos, "netos", etc. Considerando as 4 árvores no estado inicial, do estado 1, podemos realizar as remoções dos seguintes valores:

1. Árvore 1: 45, 22, 33 e/ou a raiz 88.
2. Árvore 2: 23, 21, 47 e/ou a raiz 65.
3. Árvore 3: 60, 34, 89 e/ou a raiz 65.
4. Árvore 4: 25, 20, 10 e/ou a raiz 15.

Questão 2

Pergunta

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

O código para essa questão também se encontra no repositório <https://github.com/alvarengazv/trabalhosAEDSII.git>. No diretório "lista1-aeds2-ex2".

Função solicitada:

- **(Cálculo do nível máximo)**:

  int BinaryTree::calculateMaxLevel(TreeNode* node);

- **(Visualização Interativa:)**:

  void BinaryTree::removeNode(TreeNode **t, int value);

  TreeNode* BinaryTree::insertNode(TreeNode* node, int value);

- **(Análise de Crescimento)**:

  A própria função main possui essa explicação. Mas o exemplo utilizado, para a árvore desbalanceada, insere-se os elementos em ordem crescente, deixando-a desproporcional para o lado direito, com nível 5, proporcional à quantidade de elementos. Já para a árvore praticamente balanceada, insere-se os elementos de forma a deixar a árvore com seus nós possuindo filhos em ambos os lados até o nível mais baixo. O que resulta numa altura de 3 para 7 elementos inseridos.

  Para os testes realizados, com poucos elementos, a árvore praticamente balanceada teve uma eficiência de busca pouco superior à árvore desbalanceada. Porém, para grandes massas de dados, essa diferença pode se tornar mais significativa.

- **(Caminho mais longo)**:

  void BinaryTree::findLongestPath(TreeNode* node, vector<int>& path, vector<int>& longestPath);

- **Desafio adicional**:

  void BinaryTree::suggestRotationsDetailed();

  void BinaryTree::suggestSpecificRotations(TreeNode* node, vector<string>& suggestions, string path);

  Na própria função, considera-se como desbalanceado, o nó cujas alturas das árvores filhas se diferenciam por 1 ou mais nós.

Questão 3

Pergunta

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

O código como resposta desse problema se encontra no repositório git: <https://github.com/alvarengazv/trabalhosAEDSII.git>. No diretório "lista1-aeds2-ex3".

O código todo se baseia no enunciado. Para o item "Desempenho e Otimizações", o dicionário também foi implementado lançando mão de uma árvore AVL. Assim, para grandes quantidades de palavras, espera-se a busca e a função de autocompletar, por consequência, se tornam mais rápidas, pelo do balanceamento automático da estrutura. Para trocar de estrutura, basta modificar a constante "avl"de 1 para 0 ou de 0 para 1.