Questões Avançadas sobre Árvores Binárias de Pesquisa

Sessão 1: Implementação de Funções Não Clássicas da Árvore (10 questões)

Questão 1: Implemente uma função membro `findKthSmallest(int k)` que retorna o k-ésimo menor elemento de uma ABP. Considere que k é 1-indexado. Analise a complexidade temporal e espacial de sua implementação quando a árvore não possui informações de tamanho armazenadas em cada nó.

Questão 2: Desenvolva uma função `findLCA(int val1, int val2)` que encontra o Menor Ancestral Comum de dois valores em uma ABP. Sua implementação deve funcionar mesmo quando um ou ambos os valores não existem na árvore. Explique como seu algoritmo se comporta nesses casos.

Questão 3: Implemente uma função `convertToDoublyLinkedList()` que converte uma ABP em uma lista duplamente ligada ordenada in-place, reutilizando os ponteiros left e right como prev e next. O ponteiro left deve apontar para o nó anterior e right para o próximo.

Questão 4: Crie uma função `findRangeSum(int low, int high)` que calcula a soma de todos os valores dentro de um intervalo [low, high] em uma ABP. Otimize sua solução para evitar visitar nós desnecessários.

Questão 5: Implemente uma função `serializeToString()` que serializa uma ABP para uma string, e sua correspondente `deserializeFromString(string s)` que reconstrói a árvore. Considere valores negativos e garanta que a deserialização seja única.

Questão 6: Desenvolva uma função `findPath(int target)` que retorna o caminho da raiz até um nó com valor target como um vetor de strings ("L" para esquerda, "R" para direita). Se o valor não existir, retorne o caminho até onde seria inserido.

Questão 7: Implemente uma função `mirror()` que espelha uma ABP horizontalmente. Após a operação, todos os nós à esquerda ficarão à direita e vice-versa. Analise se a propriedade ABP é mantida após esta operação.

Questão 8: Crie uma função `findSuccessorPredecessor(int val)` que retorna um par contendo o sucessor e predecessor de um valor em uma ABP. Considere os casos onde o valor não existe na árvore e quando não há sucessor ou predecessor.

Questão 9: Implemente uma função `trimToBST(int minVal, int maxVal)` que remove todos os nós fora do intervalo [minVal, maxVal] mantendo a propriedade da ABP. Otimize para minimizar o número de operações de remoção.

Questão 10: Desenvolva uma função `findModeValues()` que retorna todos os valores que aparecem com maior frequência em uma ABP (considerando que valores podem se repetir). Implemente uma versão que funciona em O(n) tempo e O(h) espaço.

Sessão 2: Implementação de Funções Não Membros com Iteradores (10 questões)

Questão 11: Implemente uma função template `bool isValidBST(Iterator begin, Iterator end)` que verifica se uma sequência de valores forma uma ABP válida quando inserida em ordem. Considere que os iteradores podem ser de qualquer tipo de container.

Questão 12: Crie uma função `mergeTwoBSTs(BST& tree1, BST& tree2)` que mescla duas ABPs usando apenas iteradores, sem acessar diretamente os nós. A função deve retornar uma nova ABP balanceada.

Questão 13: Implemente uma função `findIntersection(BST& tree1, BST& tree2)` que encontra todos os elementos comuns entre duas ABPs usando iteradores. Otimize para O(m + n) onde m e n são os tamanhos das árvores.

Questão 14: Desenvolva uma função `validateBSTIterator(Iterator it)` que verifica se um iterador customizado de ABP está implementado corretamente, testando propriedades como ordem, incremento e dereferenciamento.

Questão 15: Crie uma função `countNodesInRange(Iterator begin, Iterator end, int low, int high)` que conta quantos nós estão dentro de um intervalo usando apenas operações de iterador, sem conhecer a estrutura interna da árvore.

Questão 16: Implemente uma função `detectCycle(Iterator begin, Iterator end)` que detecta se há um ciclo na estrutura de dados usando apenas iteradores. Considere que os iteradores podem não seguir a ordem esperada se houver ciclo.

Questão 17: Desenvolva uma função `transformBST(Iterator begin, Iterator end, Function f)` que aplica uma transformação a todos os elementos de uma ABP mantendo a propriedade de ordenação, usando apenas iteradores.

Questão 18: Crie uma função `findMedian(Iterator begin, Iterator end)` que encontra a mediana de uma ABP usando apenas iteradores bidirecionais, sem converter para array ou usar memória adicional O(n).

Questão 19: Implemente uma função `zipTwoBSTs(Iterator it1, Iterator it2)` que percorre duas ABPs simultaneamente e retorna pares de valores correspondentes, parando quando uma das árvores termina.

Questão 20: Desenvolva uma função `reverseIterator(Iterator it)` que implementa um iterador reverso para ABP usando apenas um iterador forward, mantendo complexidade O(1) para operações de incremento.

Sessão 3: Implementações no Main com Cenários Específicos (10 questões)

Questão 21: Dado um arquivo de texto com milhões de palavras, implemente no main um sistema que: (1) constrói uma ABP de frequências, (2) encontra as 10 palavras mais frequentes, (3) calcula a mediana das frequências. Considere limitações de memória.

Questão 22: Implemente no main um sistema de cache LRU usando uma ABP onde cada nó armazena tanto a chave quanto o timestamp do último acesso. O sistema deve permitir inserção, busca e remoção de elementos expirados.

Questão 23: Crie no main um programa que lê coordenadas 2D e constrói uma ABP baseada na distância euclidiana da origem. Implemente funções para encontrar todos os pontos dentro de um raio específico.

Questão 24: Desenvolva no main um simulador de sistema bancário onde cada conta é um nó na ABP (ordenado por ID). Implemente operações de transferência que podem falhar e precisam ser revertidas, mantendo a consistência da árvore.

Questão 25: Implemente no main um sistema de indexação de documentos onde cada palavra é um nó na ABP contendo uma lista de documentos onde aparece. Crie funções para busca booleana (AND, OR, NOT).

Questão 26: Crie no main um programa que simula um sistema de reservas de assentos em um cinema, usando ABP para gerenciar disponibilidade. Implemente algoritmos para encontrar o melhor conjunto de assentos consecutivos.

Questão 27: Desenvolva no main um sistema de árvore genealógica usando ABP onde cada pessoa é um nó. Implemente funções para encontrar ancestrais comuns, descendentes e calcular graus de parentesco.

Questão 28: Implemente no main um sistema de versionamento de código onde cada commit é um nó na ABP. Crie funções para merge de branches, detecção de conflitos e rollback para versões anteriores.

Questão 29: Crie no main um programa de análise de logs de servidor onde cada entrada é um nó na ABP ordenado por timestamp. Implemente detecção de padrões anômalos e geração de relatórios por período.

Questão 30: Desenvolva no main um sistema de recomendação usando ABP onde cada usuário é um nó com preferências. Implemente algoritmos para encontrar usuários similares e gerar recomendações colaborativas.

Sessão 4: Fatores de Balanceamento e Análise Passo a Passo (10 questões)

Questão 31: Dada a sequência de inserções [50, 30, 70, 20, 40, 60, 80, 10, 25, 35, 45] em uma árvore AVL inicialmente vazia, mostre passo a passo: (1) o estado da árvore após cada inserção, (2) os fatores de balanceamento de todos os nós, (3) as rotações necessárias.

Questão 32: Considere uma árvore AVL com a seguinte estrutura inicial: raiz=50, filhos de 50: 30 e 70, filhos de 30: 20 e 40, filhos de 70: 60 e 80. Mostre passo a passo o que acontece ao remover os nós 20, 40

e 30 nesta ordem, incluindo todos os fatores de balanceamento e rotações.

Questão 33: Analise uma árvore AVL onde cada nó armazena também o tamanho da subárvore. Mostre como os fatores de balanceamento e tamanhos são atualizados durante uma rotação dupla direita-esquerda, detalhando cada passo.

Questão 34: Dada uma ABP degenerada (lista ligada) com elementos [1,2,3,4,5,6,7,8,9,10], transforme-a em uma árvore AVL balanceada. Mostre o processo passo a passo, incluindo o cálculo dos fatores de balanceamento após cada operação.

Questão 35: Considere uma árvore AVL onde cada nó tem um campo adicional para armazenar a altura da subárvore. Mostre como implementar a operação de inserção com cálculo automático de fatores de balanceamento, detalhando a propagação das mudanças.

Questão 36: Analise o comportamento dos fatores de balanceamento durante a operação de merge de duas árvores AVL. Mostre um exemplo concreto com árvores de alturas diferentes e detalhe todas as rotações necessárias.

Questão 37: Dada uma sequência de remoções [45, 30, 60, 25, 75] em uma árvore AVL, mostre como os fatores de balanceamento são recalculados e quais rotações são necessárias. Considere que a árvore inicial tem altura 4.

Questão 38: Implemente uma função que detecta se uma árvore binária pode ser transformada em AVL apenas com rotações (sem inserções/remoções). Mostre o algoritmo passo a passo para um exemplo específico.

Questão 39: Analise uma árvore AVL com threading (ponteiros para sucessor/predecessor). Mostre como os fatores de balanceamento são afetados e como as rotações devem ser adaptadas para manter os threads.

Questão 40: Considere uma árvore AVL persistente (immutable). Mostre passo a passo como uma inserção cria uma nova versão da árvore, incluindo o cálculo de fatores de balanceamento e quais nós precisam ser copiados.

Sessão 5: Questões Teóricas Complexas sobre Complexidade (10 questões)

Questão 41: Prove que o número máximo de rotações necessárias para rebalancear uma árvore AVL após uma inserção é constante, mas demonstre que para uma sequência de n inserções, o número total de rotações pode ser $\Theta(n \log n)$ no pior caso.

Questão 42: Analise a complexidade de espaço de uma árvore binária de pesquisa quando implementada com: (1) ponteiros parent explícitos, (2) threading, (3) path compression. Compare o overhead de memória e o impacto na complexidade temporal das operações.

Questão 43: Considere uma modificação da árvore AVL onde cada nó armazena o rank (posição na ordem). Analise como isso afeta a complexidade das operações de insert, delete, find e select(k). Prove

que select(k) pode ser implementada em O(log n).

Questão 44: Demonstre que uma árvore binária de pesquisa com n nós tem pelo menos $\lceil \log_2(n+1) \rceil$ de altura no melhor caso, e prove que o número de comparações para encontrar um elemento é limitado por essa altura.

Questão 45: Analise a complexidade amortizada de uma sequência de operações em uma árvore splay. Prove que m operações em uma árvore com n nós têm complexidade O(m log n) no total, mesmo que algumas operações individuais sejam O(n).

Questão 46: Compare a complexidade de cache (cache misses) entre árvores binárias de pesquisa e árvores B. Demonstre por que árvores B são preferíveis para armazenamento em disco, considerando o modelo de memória hierárquica.

Questão 47: Analise a complexidade probabilística de operações em uma treap (árvore binária randomizada). Prove que a altura esperada é O(log n) e que operações básicas têm complexidade esperada O(log n).

Questão 48: Considere uma árvore binária de pesquisa que suporta operações de range update (atualizar todos os valores em um intervalo). Analise a complexidade de implementações usando: (1) lazy propagation, (2) segment trees, (3) árvores de intervalos.

Questão 49: Prove que o número de árvores binárias de pesquisa estruturalmente diferentes com n nós distintos é o n-ésimo número de Catalan. Derive a fórmula e analise as implicações para algoritmos de enumeração.

Questão 50: Analise a complexidade de paralelização de operações em árvores binárias de pesquisa. Demonstre que algumas operações podem ser paralelizadas eficientemente enquanto outras são inerentemente sequenciais.

Sessão 6: Análise de Código com Pegadinhas (10 questões)

Questão 51: Analise o seguinte código e determine a saída ou se há erro:

cpp

```cpp
BST tree;
auto it = tree.begin();
*it = 10;
tree.insert(5);
tree.insert(15);
cout << *tree.find(10) << endl;
++it;
tree.erase(tree.find(5));
cout << *it << endl;
```

Questão 52: Determine o comportamento do código:

cpp

```cpp
BST tree{1, 2, 3, 4, 5};
auto it1 = tree.lower_bound(3);
auto it2 = tree.upper_bound(3);
tree.erase(it1, it2);
for(auto it = tree.begin(); it != tree.end(); ++it) {
    if(*it == 3) tree.erase(it);
}
```

Questão 53: Analise este código com iteradores:

cpp

```cpp
BST tree{10, 5, 15, 3, 7, 12, 18};
auto it = tree.find(7);
++it;
tree.insert(8);
tree.insert(6);
cout << *it << " ";
--it; --it;
cout << *it << endl;
```

Questão 54: Determine a saída ou erro:

cpp

```cpp
BST tree;
tree.insert(50);
auto it = tree.find(50);
tree.erase(it);
tree.insert(50);
cout << (it == tree.find(50)) << endl;
cout << *it << endl;
```

Questão 55: Analise este código complexo:

cpp

```cpp
BST tree{4, 2, 6, 1, 3, 5, 7};
auto it = tree.begin();
advance(it, 3);
int val = *it;
tree.erase(tree.find(val));
tree.insert(val + 1);
cout << distance(tree.begin(), tree.find(val + 1)) << endl;
```

Questão 56: Determine o comportamento:

cpp

```cpp
BST tree{10, 5, 15};
auto it1 = tree.begin();
auto it2 = tree.end();
--it2;
tree.clear();
tree.insert(20);
cout << *it1 << " " << *it2 << endl;
```

Questão 57: Analise este código com modificação durante iteração:

cpp

```cpp
BST tree{1, 3, 5, 7, 9, 11};
for(auto it = tree.begin(); it != tree.end(); ++it) {
    if(*it % 2 == 1) {
        tree.insert(*it + 1);
        if(*it > 5) tree.erase(it);
    }
}
```

Questão 58: Determine a saída:

cpp

```cpp
BST tree{100, 50, 150, 25, 75, 125, 175};
auto it = tree.find(75);
tree.erase(tree.find(50));
cout << *it << " ";
++it;
cout << *it << " ";
tree.insert(80);
--it;
cout << *it << endl;
```

Questão 59: Analise este código com ponteiros:

cpp

```cpp
BST tree{10, 5, 15, 3, 7, 12, 18};
auto it = tree.find(10);
Node* ptr = &(*it);
tree.erase(it);
tree.insert(10);
cout << ptr->data << endl;
cout << (*tree.find(10)) << endl;
```

Questão 60: Determine o comportamento final:

cpp

```cpp
BST tree1{1, 2, 3};
BST tree2{4, 5, 6};
auto it1 = tree1.begin();
auto it2 = tree2.begin();
swap(tree1, tree2);
cout << *it1 << " " << *it2 << endl;
tree1.erase(it2);
cout << tree1.size() << endl;
```

Instruções Gerais:

- Todas as questões assumem uma implementação padrão de BST com iteradores bidirecionais
- Considere que BST é uma classe template com operações padrão (insert, erase, find, begin, end)
- Analise cuidadosamente invalidação de iteradores e comportamentos indefinidos
- Para questões de análise de código, considere possíveis warnings de compilação
- Assuma que todas as inclusões necessárias (#include) estão presentes

- Questões marcadas como "erro" devem especificar o tipo de erro (compilação, runtime, comportamento indefinido)