INSTITUTO FED. DE EDUCAÇÃO, CIÊNC. E TEC. DE PERNAMBUCO
CURSO: TEC. EM ANÁLISE E DESENVOLVIMENTO DE SISTEMAS
DISCIPLINA: ALGORITMOS E ESTRUTURAS DE DADOS
PROFESSOR: RAMIDE DANTAS
ASSUNTO: ÁRVORE DE BUSCA BINÁRIA E AVL

Prática 07

OBS.: Veja o código que acompanha a prática e estude-o.

Parte 1: Implementando BST e AVL (arquivo de teste: main.cpp)

Problema 1: Em **bst.cpp,** implementar as funções privadas `_insert()` e `_search()`.

A implementação mais direta da inserção é recursiva. A função é chamada passando um nó da árvore (inicialmente a raiz, depois sub-árvore esquerda ou direita recursivamente), e o retorno permite atualizar os ponteiros do nó pai (ou a raiz, quando for a primeira inserção). Depois de inserir o novo elemento, a altura do nó `_root` deve ser atualizada: chame `updateH()` para esse nó ao final da inserção (isso será necessário o funcionamento correto da AVL). **Importante:** caso o valor já exista na árvore, ignore-o; duplicações quebram o funcionamento da AVL.

Para a função de busca, siga o material de aula (solução recursiva) ou pense em uma forma iterativa.

Desafio (Opcional): Em **bst.cpp**, implementar `_successor()`, dado `_predecessor()`.

Em uma árvore binária de busca, o predecessor de um valor X presente na árvore é o maior valor Y também presente na árvore que é menor que X. Isto é, se pegássemos os elementos da árvore em ordem, Y seria o valor que viria imediatamente antes de X (se houver). O método `_predecessor()` realiza essa busca.

Simetricamente, o sucessor de X é o valor W que viria logo após X na sequência de elementos presentes na árvore. Implemente o método `_successor()` se baseando na implementação de `_predecessor()`. Estude esse método para fazer os modificações necessárias.

Para testar, será necessário ativar o flag `test_succ_pred` em **main.cpp**.

Problema 2: Em **avl.cpp**, implementar `rotateRight()`, dada `rotateLeft()`.

O restante do código da AVL já está implementado. Estudo-o para entender seu funcionamento.

Veja que ao final da rotação, é preciso atualizar as alturas dos nós, para recalcular o fator de balanceamento.

Desafio (Opcional): Resolver o problema LC108 no LeetCode.

Esse problema pede para gerar uma árvore binária de busca de altura mínima dado um array ordenado de inteiros.

Parte 2: Usando BST e AVL

Problema 1: Usando a BST para ordenar os elementos (teste no arquivo **tree_sort.cpp**).

É possível usar uma árvore binária de busca para ordenar os elementos de um vetor. Implemente a função `_sort()` em **bst.cpp**, tomando como base a função `_show()`. Veja que essa função recebe uma referência para `vector<>`, inicialmente vazio, que onde ficará o resultado da ordenação. Implemente a função `_sort()` dando `push_back()` dos elementos no vetor na ordem correta (isto é, percurso em-ordem da árvore). Porém, fica o alerta que o uso do `push_back()` nessa solução é bastante ineficiente pois força o `vector<>` a realocar o array interno a cada inserção.

Desafio 1 (Opcional): Melhore o código de `_sort()` evitando usar `push_back()` no vetor.

Tente inicializar o vetor com o tamanho certo. Você pode precisar mudar a assinatura de `_sort()` e a implementação da função pública `sort()` caso precise passar um índice como parâmetro adicional.

Desafio 2 (Opcional): Resolver o problema *Kth Smallest Element* no LeetCode (LC230).

Nesse problema é pedido para você retornar o K-ésimo menor elemento na árvore. Veja que você recebe como parâmetro a árvore pronta, seguindo a estrutura dada no código, logo você não pode usar diretamente o código desenvolvido na Parte 1, somente a lógica básica.

Problema 2: Resolver o problema *Two Sum* (simplificado, ver desafio) (arquivo **two_sum.cpp**).

Nesse problema, é dado um conjunto de inteiros não ordenado e um valor alvo (*target*). Sua tarefa é encontrar (se possível) um par de inteiros do conjunto que somados dão o valor alvo. A solução ingênua (força bruta) desse problema testa todas as somas possíveis de pares, sendo portanto $O(N^2)$. É possível resolver o problema de forma mais eficiente usando a BST/AVL que desenvolvemos ou a estrutura `std::set<>` (conjunto) da STL. Nesse caso, em vez de testar todas a somas, para cada inteiro X do conjunto, veja se seu complemento para o alvo T (isto é $T - X$) existe no conjunto. A estrutura `std::set<>` é implementada tipicamente usando uma BST auto-balanceada (AVL ou *Red-Black*), logo a consulta tem custo $O(logN)$, e a solução final tem complexidade total $O(NlogN)$, em vez de $O(N^2)$ (para referência, é a mesma diferença de desempenho entre o *Quick Sort* e o *Bubble Sort*). **Atenção:** a soma deve ser entre valores diferentes; isto é, somar duas vezes o mesmo valor não é uma solução válida.

Desafio (Opcional): Resolver o problema *Two Sum* no LeetCode (LC001).

Mesma ideia da versão anterior, só que nessa versão deve-se retornar os índices dos valores que somam o *target* (em vez dos valores em si, como na versão simplificada). Nesse caso é recomendado usar um `std::map<int, int>` em vez de `std::set<int>`. O `std::map<int, int>` deve mapear cada valor no seu respectivo índice. O `std::map<int, int>` também é implementado com uma árvore binária de busca auto-balanceada, portanto a complexidade é a mesma da versão anterior.