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 BINÁRIA DE BUSCA E AVL

Aluno (a):			
Matrícula:		**Data:**	

Prática 08

Parte 0: Preparação

Passo 1: Crie um novo projeto chamado **Pratica8**.

Configure o projeto para utilizar C++11.
(Project Properties > C++ Build > Settings: no caso do Cygwin procure por "Dialect" e selecione "ISO C++11" em "Language standard")

Passo 2: Adicione os arquivos que acompanham a prática 8 ao projeto.

main.cpp: contém a função main. Cria a árvore e realiza inserções, buscas e remoções.

bst.h e **bst.cpp:** declaração e implementação da árvore binária de busca, respectivamente.

avl.h e **avl.cpp:** declaração e implementação da árvore AVL.

Passo 3: Compile e rode o código para verificar se não há erros.

Nesse ponto deve apenas compilar e rodar mas não produzir resultados; algumas funções não estão implementadas.

Passo 4: Estude o código para se familiarizar.

Veja o material de aula se necessário. Faça comentários nos trechos mais complicados. Refatore o código se achar que vai ajudar seu trabalho. Teste novamente antes de fazer modificações mais profundas.

Parte 1: Implementando Inserção na Árvore Binária de Busca (BST)

Passo 1: Implemente a função de inserção na BST.

A função a ser implementado é a privado, cujo corpo está em **bst.cpp**. A implementação mais direta é recursiva e se assemelha com a busca (`search()`). A função é chamada passado um nó da árvore (inicialmente a raiz, depois esquerdo ou direito), 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 da árvore deve ser atualizada (`updateH()`). **Importante:** caso a chave já exista na árvore, não faça nada. Duplicações na árvore quebram o funcionamento da árvore AVL.

Passo 2: Compile e teste a aplicação.

Verifique se os elementos inseridos foram encontrados pela busca. Veja também se a altura dos nós da árvore está correta. A altura é o segundo componente exibido quando se usa o método `show()`.

Parte 2: Implementando Sucessor

Passo 1: Implemente o método `successor()` em **bst.cpp**.

Em uma árvore 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.

Passo 2: Compile e teste a aplicação.

Rode a aplicação com poucos elementos no array, verificando se o valor retornado por `successor()` é o correto.

Parte 3: Implementando Rotações na Árvore AVL

Passo 1: Implemente as métodos `rotateLeft()` e `rotateRight()` em **avl.cpp**.

Esses métodos realizam as rotações simples à esquerda e à direita respectivamente na árvore AVL. Veja como esses métodos são usados dentro de `rebalance()`. Nesses métodos, após realizar a rotação em si (ajuste dos ponteiros), é preciso atualizar a altura dos nós envolvidos. Atenção que a ordem de atualização é importante.

Passo 2: Compile e teste a aplicação.

Verifique se a árvore permanece válida (`validate()`) e se os fatores de balanceamento e alturas estão corretos.

Parte 4: (Desafio) Modificando a estrutura da árvore

Passo 1: Modifique o nó da árvore de forma que ele contenha um ponteiro para o nó pai.

Passo 2: Ajuste as classes BinaryTree e AVLTree de forma a considerar esse ponteiro.

Reescreva as funções `predecessor()` e `sucessor()` de forma a usar essa informação. Modifique as funções que achar necessário.

Passo 3: Compile e teste aplicação.

Verifique se as mudanças alteraram o comportamento da estrutura.