COSC230 Assignment 4

This assignment consolidates material from *Practical 6* on "Binary Search Trees and Hash Tables". Please do not attempt Assignment 4 before working through this practical. Please submit all of your assignment files through `Turing` (see the link Assignment Submission via `Turing`). Before you submit your assignment please check your code compiles on `Turing` and does what you expect it to.

Question 1: [*30 marks*]

Download `Assignment_4_Code_files` from myLearn that includes `bst.h` for the binary search tree class from *Practical 6*. Make use of the binary search tree property and the recursive structure of a binary search tree to develop a recursive implementation of the `insert` function:

```
void Binary_Search_Tree<T>::insert(Node<T>* parent, Node<T>* q, Node<T>* p).
```

Think carefully about the Base and Recursion steps (the Base is different to that for `search`). Remember that each step of Recursion should get closer to Base. Test your function using the provided `Q1_test.cc`.

Question 2: [*10 marks*]

Download `Assignment_4_Code_files` from myLearn that includes `rbt.h` for the red-black tree class from *Practical 6*. Write a function, similar to `inorder_leaf_height` from *Practical 6*, to return the black-height of each leaf node in the tree:

```
void Red_Black_Tree<T>::inorder_leaf_black_height(Node<T>*, int).
```

Remember from lectures, the black-height is given by the number of black nodes along a simple path from, but not including, a node down to a leaf. Test your function using the provided `Q2_test.cc`.

Question 3: [*60 marks*]

Download `Assignment_4_Code_files` from myLearn that includes `expression_tree.h` for an *expression tree*. An expression tree provides an unambiguous representation of algebraic expressions without the use of parentheses or rules of operator precedence. The idea is very similar to that of a parse tree used to generate grammatically-correct sentences. An expression tree can be used to generate unambiguous algebraic expressions through the use of Polish notation. Constructing an expression tree corresponding to a particular algebraic expression, such as $(2 + (3 * (2 + 2))) + 5$, then traversal of this expression tree will lead to unambiguous expressions without parentheses in either prefix notation: $++2*3+225$ (for preorder traversal), or postfix notation: $2322+*+5+$ (for postorder traversal). Infix notation resulting from inorder traversal remains ambiguous. Now consider the following expression-tree class interface:

```
#include <stack>
#include <vector>
#include <iostream>

// A very simple expression tree class that converts a parenthetized input
// consisting of addition and multiplication operations, into an expression tree.
// Traversal of this expression tree generates various Polish notations.

class Expression_Tree {
public:
    Expression_Tree() { root = 0; }
    ~Expression_Tree() { postorder_delete(root); }
    bool empty() { return root == 0; }
    void build_expression_tree(std::vector<char>&);
    void postorder_delete() { postorder_delete(root); }
    void postorder_delete(Node<char>*);
    void preorder_walk() { preorder_walk(root); }
    void preorder_walk(Node<char>*);
    void inorder_walk() { inorder_walk(root); }
    void inorder_walk(Node<char>*);
    void postorder_walk() {postorder_walk(root); }
    void postorder_walk(Node<char>*);
private:
    Node<char>* root;
};
```

To use this class, you will need to implement the member function:

`void Expression_Tree::build_expression_tree(std::vector<char>&).`

To do this, convert the input into a stack of subtree root-nodes containing no parentheses. Parenthesis matching requires a stack data structure, which is why you are using a stack to store the subtrees. Parenthesis matching can be done as follows:

1. Read each input character into a new node by iterating through the input vector, and push them onto the stack until a right parenthesis is found.
2. If a right parenthesis is found, backtrack by popping three items (right operand, operator, and left operand) off the stack.
3. Construct a subtree from these three items with the operator (second item) as the root, and the left operand (third item) and right operand (first item) as the left and right children. Note: a parent pointer is not used in this expression tree.
4. Free and remove the matching left parenthesis from the stack.
5. Add the root-node of the subtree you just built to the stack.
6. Repeat 1–5 until all parentheses have been removed and you have reached the end of the input vector.

Due to your careful parenthesis matching, you should now have a stack of subtrees in the correct order. To build a tree from these subtrees requires a final pass through the stack. Pop three consecutive items off the stack, and repeat steps 3 and 5. Continue to do this until the stack contains just one single node: this will be the root-node of the completed expression tree. Finally, assign the root of the expression tree to this node. Test your function using the provided `Q3_test.cc`.

Submit only your `bst.h`, `rbt.h`, and `expression_tree.h` files.