THE UNIVERSITY OF THE WEST INDIES
Department of Computing
COMP1127–Introduction to Computing II

Lab 5

Access and Complete the following Lab Exercise on Hackerrank.

Opening Date: **Friday, November 18, 2022**

Lab Date (Week 4): **Monday, November 21 – Saturday, November 26, 2022**
Due Date: **11:45 pm, Sunday, November 27, 2022 (on Hackerrank)**

A binary tree ADT is provided for this exercise. The binary tree is a list which contains a tag 'btree', root, left tree, right tree. If the binary tree is empty, the list contains only the tag 'btree'. The following functions have been provided in the hackerrank program:

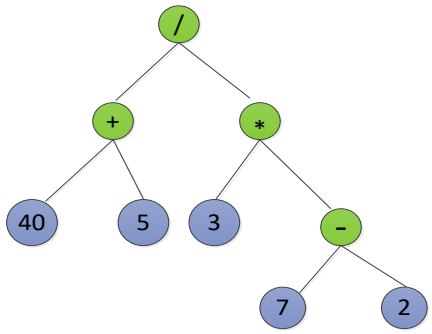
Type	Name	Description
Constructor	`makeTree()`	Creates a binary tree as a list with a tag 'btree', root value, left subtree, right subtree.
Constructor	`make_empty_tree()`	Creates an empty binary tree as a list with a tag 'btree'.
Selector	`root()`	Returns the root value of a given binary tree.
Selector	`left_subtree()`	Returns left subtree of a given binary tree.
Selector	`right_subtree()`	Returns right subtree of a given binary tree.
Predicate	`is_btree()`	Returns whether a given object is a binary tree.
Predicate	`is_empty_tree()`	Returns whether a given binary tree is empty.
Predicate	`is_leaf_tree()`	Returns whether a given binary tree is a leaf i.e. the left and right subtrees are empty trees.

In the hackerrank file, the following are also provided:

`preorder` - A function which takes a tree and returns a list of the root values where parent nodes are listed before those of the left and right subtrees

`inorder` - A function which takes a tree and returns a list of the root values where parent nodes are listed between those of the left and right subtrees

`tree_ex` - A binary tree with 4 operators and 5 operands.



Details of the binary tree `tree_ex`, and the functions `preorder` and `inorder`

```
tree_ex = makeTree ('/' ,  \
            makeTree ('+',  \
               makeTree(40, make_empty_tree(), make_empty_tree()),
               makeTree(5, make_empty_tree(), make_empty_tree())),  \
            makeTree('*', \
               makeTree(3, make_empty_tree(),make_empty_tree()),  \
               makeTree('-',  \
                  makeTree(7, make_empty_tree(), make_empty_tree()),
                  makeTree(2, make_empty_tree(), make_empty_tree()))))

def preorder(tree):
    if is_empty_tree(tree):
        return []
    else:
        return [root(tree)] + \
               preorder(left_subtree(tree))  +  preorder(right_subtree(tree))

def inorder(tree):
    if is_empty_tree(tree):
        return []
    else:
        return inorder(left_subtree(tree)) + [root(tree)] +  \
               inorder(right_subtree(tree))
```

When using an Interactive Development Environment (IDE) such as IDLE, the functions for the ADT may be put in a separate file (such as `tree.py`) and imported in the main program (such as `lab5.py`) by using the following command:

```
from tree import *
```

Problem 1

Write a function `postorder()` which takes a tree and flattens the tree into a list using the post order traversal, i.e. left subtree, right subtree, root of tree.
For example, given the expression tree example `tree_ex`, flattening this tree into a list using the post order traversal would give [40, 5, '+', 3, 7, 2, '-', '*', '/']

```
>>> postorder(tree_ex)
[40, 5, '+', 3, 7, 2, '-', '*', '/']
```

Hint: `preorder` and `inorder` are provided as guides to `postorder`. Pre order traversal processes (1) root of tree, (2) left subtree, (3) right subtree. In order traversal processes (1) left subtree, (2) root of tree, (3) right subtree.

Problem 2

Implement the following functions:

Type	Name	Description
Constructor	`stack()`	Creates a stack as a tuple, where first part of the tuple is a tag "stack" and second part of the tuple is a list.
Selector	`contents()`	Takes a stack and returns the structure where the elements in the stack are stored.
Selector	`top()`	Takes a stack and returns the element that is on top of the stack.
Predicate	`is_stack()`	Takes an object as input and returns True if the object is a tuple and the first part of the structure is a tag "stack"
Predicate	`stack_empty()`	Takes a stack and returns True if it is empty.
Mutator	`push()`	Takes a stack and an element and modifies the stack such that the element is added to the back (or end of the list).
Mutator	`pop()`	Takes a stack and removes the top element from the stack.

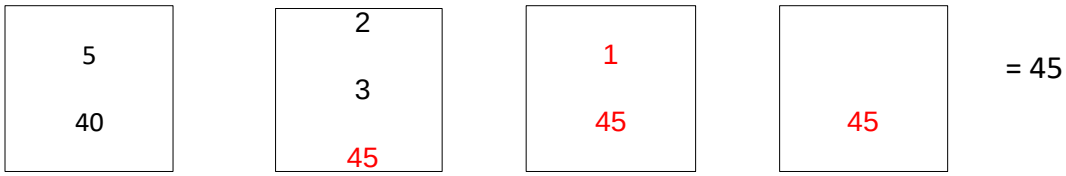
Example:

```
>>> st = stack()
>>> st
('stack', [])
>>> push(st, 5)
>>> push(st, 6)
>>> top(st)
6
>>> st
('stack', [5, 6])
>>> pop(st)
>>> top(st)
5
>>> st
('stack', [5])
```

Problem 3

The `postorder()` method of flattening the example expression tree `tree_ex` would give [40, 5, '+', 3, 7, 2, '-', '*', '/']. A stack is often used to evaluate such an expression. While traversing through this list if an element is as operand then it is pushed onto a stack, and when an operator is found, then the stack is popped twice and the operator is evaluated with the popped value and its result is pushed back onto the stack.

```
>>> evalPostfix_helper([40,5, '+',3,2, '-', '*'])
45
```



40+5 = 45 3-2 = 1 45 * 1 = 45

3.1 Write a function `is_operator()` which returns True if the argument is an operator (i.e. "+", "-", "*" or "/") and False otherwise.

```
>>> is_operator("+")
True
>>> is_operator("=")
False
```

3.2 Write a function `apply_operator()` which takes three arguments, the operator, the second popped element and the first popped element and returns the result of applying the operator to the second and first popped elements. For example, `apply_operator('+',40,5)` would evaluate 40+5 and return 45.

```
>>> apply_operator("+", 40, 5)
45
>>> apply_operator("-", 40, 5)
35
```

3.3 Write a function called `evalPostfix()` which takes an expression tree as an argument. It converts the expression tree to a list using a `postorder()` function. If the element in the list is not an operator then it is added to the stack. If the element is an operator then it pops the stack twice and calls the function `apply_operator()` with three arguments the operator, the second popped value and the first popped value. This function returns a value which is pushed on the stack. When all the elements of the list have been processed then return the top of stack which would give you the result of the expression being calculated.

```
>>> evalPostfix(tree_ex)
3.0
```