COMPSCI 105 Summer School 2016
Assignment Two

Worth: 9% of your final grade
Deadline: 9:00pm, Friday 12th February

Please submit what you have completed before the deadline
Only submit code that you have written, debugged and tested yourself
Zero-tolerance for plagiarism - course failure and disciplinary action will result
No extensions, unless *exceptional* circumstances apply
(discuss with Paul as soon as possible)

What you need to do

You must create a single source file called Asst2.py. In this source file, you will import the Node, Stack and BinaryTree classes that are provided to you (however you should not modify these classes as you will not submit them - you will only submit the Asst2.py source file).

Asst2.py

```
from Node import Node
from Stack import Stack
from BinaryTree import BinaryTree

def generate_postfix(infix):
    ...
    ...

def generate_prime_chain(n):
    ...
    ...

def back_to_front_chain(head):
    ...
    ...


...
...
```

Within the Asst2.py source file, you will need to define the functions described in this document.

All functions must be defined, however if you have not implemented a function you may use the keyword "pass" as its definition.

The general structure of the file will be as shown in the diagram on the left.

When you download the project resources, please run the program "TestProgramAsst2.py" for further instructions.

You must submit only one file - **Asst2.py** - to the Assignment Drop Box:

https://adb.auckland.ac.nz

prior to 9:00pm on Friday 12th February.

Paul Denny
February 2016

Question One: Postfix generator (12 marks)

A mathematical expression in *infix* notation has the operators placed *between* the operands. With such a notation, it is necessary to use parentheses to change the order in which the operators are evaluated. With *postfix* notation the operators are re-ordered, and appear to the right of the operands, in such a way that the order or evaluation is made explicit.

For example, the following infix expression:

```
2 * (3 + 7)
```

can be represented in postfix notation as follows:

```
2 3 7 + *
```

In this question, you must write a function called **generate_postfix()** that converts an infix expression to postfix. The function must also check that the infix expression is valid.

The input to the function will be a string. Each token in the string (whether an operator, an operand or a bracket) will have a single space character on either side of it (with the exception of the start and end of the string). You will find the Python split() method useful to convert the input string to a list of tokens.

An infix expression is *valid* if the following three conditions hold:

1. every token must be either an open parenthesis, a close parenthesis, a valid mathematical operator, or a string representing an integer
2. the brackets (i.e. the opening and closing parentheses) are balanced
3. there is exactly one more operand than there are operators

For this question, all operands must be integers. Also, there are exactly 7 operators that are valid. In addition to the four usual operators (+, -, *, /) you should also allow the following three operators:

Operator	Description	Example
^	this is the exponentiation operator, and raises a base to a power	2 ^ 10
<	this is the “minimum” operator and evaluates to the smaller of the two operands	10 < 20
>	this is the “maximum” operator and evaluates to the larger of the two operands	10 > 20

Relative to the four standard operators, the exponentiation operator (^) has *higher* precedence, and the minimum and maximum operators (<, >) have *lower* precedence. The following table summarises the precedence of all of the operators:

Highest	2nd	3rd	4th	Lowest
()	^	* /	+ -	< >

In expressions consisting of several operators of the same precedence, the evaluation order will be *left to right*. A few examples are shown below:

Infix expression	**Corresponding postfix expression**
7 * 2 ^ (8 + 4 / 2 > 1) - 15 / 4	7 2 8 4 2 / + 1 > ^ * 15 4 / -
6 > 7 + 6	6 7 6 + >
6 - 7 + 6	6 7 - 6 +
2 ^ (4 < 6) - 3 / 3 * 1	2 4 6 < ^ 3 3 / 1 * -

If the input infix expression is valid, then the function should return a string representing the corresponding postfix expression.

However, if the input infix expression is not valid, there are four error messages that may also be returned:

1. 'imbalanced brackets' (if the parentheses are not balanced)
2. 'invalid symbol' (if any token is not a valid operator, integer, or bracket)
3. 'too many operators' (if the number of operators is greater than or equal to the number of operands)
4. 'too few operators' (if the number of operators is more than one less than the number of operands)

Examples of each of these are shown below.

The following code:

```
print(generate_postfix('2 ^ ( 4 < 6 ) - 3 / 3 * 1'))
print(generate_postfix('( 2 ^ ( 4 < 6 ) - 3 / 3 * 1'))
print(generate_postfix('2 ^ ( 4 < 6.6 ) - 3 / 3 * 1'))
print(generate_postfix('2 ^ + ( 4 < 6 ) - 3 / 3 * 1'))
print(generate_postfix('2 ^ ( 4 < 6 ) - 3 3 * 1'))
```

should produce the output:

```
2 4 6 < ^ 3 3 / 1 * -
imbalanced brackets
invalid symbol
too many operators
too few operators
```

You should make use of the Stack class that is provided to you.

Question Two: Prime chain (11 marks)

For this question you will *create* a chain of nodes using the Node class provided. The Node class consists of a *data* attribute and a *next* attribute. Please refer to the provided class.

Define a function called **generate_prime_chain()** which takes one integer as input, *n*. The function should construct a chain of Node objects, where the data attributes are prime numbers, representing the first *n* primes. Once this chain is constructed, the function should return the chain.

For example, the following code:

```
my_primes = generate_prime_chain(3)

print(my_primes.get_data(), end = ' ')
print(my_primes.get_next().get_data(), end = ' ')
print(my_primes.get_next().get_next().get_data(), end = ' ')
print(my_primes.get_next().get_next().get_next())
```

should produce the output:

```
2 3 5 None
```

Question Three: Back to front (11 marks)

For this question you will *modify* an existing chain of nodes. Once again, please refer to the provided Node class.

Define a function called **back_to_front_chain()** which is passed a reference to the first Node in a chain of nodes. You must then *modify* the chain by moving the last node in the chain to the second to front position. Please note, you should not return anything from this function. You should simply modify the existing chain that is passed as input to the function.

For example, if the list contains the following nodes: 1 -> 2 -> 3 -> 4 -> 5 then after the function is called, the nodes should be in the following order:

```
1 -> 5 -> 2 -> 3 -> 4
```

As another example, consider the code below:

```
first_node = Node('ann')
second_node = Node('bob')
third_node = Node('cat')
first_node.set_next(second_node)
second_node.set_next(third_node)

back_to_front_chain(first_node)

print(first_node.get_data(), end = ' ')
print(first_node.get_next().get_data(), end = ' ')
print(first_node.get_next().get_next().get_data(), end = ' ')
```

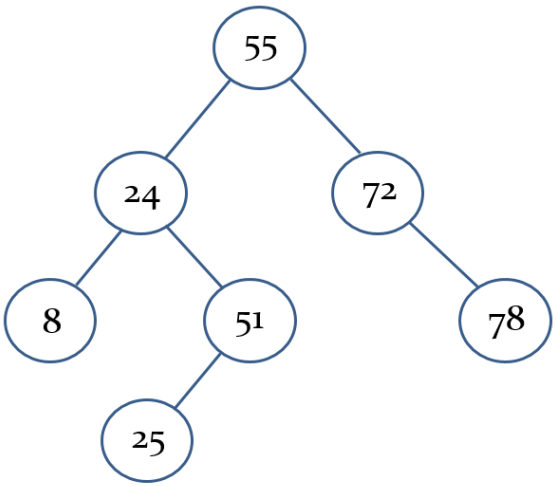
The output generated by this code should be: `ann cat bob`

Obviously, if the original chain consists of just 1 or 2 nodes, then this function will have no effect on the nodes.

Question Four: Creating trees (11 marks)

Carefully examine the minimal BinaryTree class provided. Each instance of this class has three attributes: *data*, *left* and *right*. The *data* attribute stores the relevant data for a given node in the tree, and the *left* and *right* attributes represent subtrees that are either None or refer to valid BinaryTree objects. The class provides only basic accessor/mutator methods (i.e. getters and setters) and a method for generating a string representation of a BinaryTree object.

In this question you will be defining two functions for generating trees from lists. Consider the binary tree shown below:



This binary tree could be represented by two lists that use quite different approaches for encoding the tree structure:

A nested list

```
[55, [24, [8, None, None], [51, [25, None, None], None]],
                                  [72, None, [78, None, None]]]
```

The *nested list format* always uses a list of length three to represent a binary tree. The first item in the list is the data value of the root, the second item in the list is the left subtree (this may be None if the left subtree is empty, or it may be a nested list) and the third item in the list is the right subtree (this may be None if the right subtree is empty, or it may be a nested list)

A flat list

```
[None, 55, 24, 72, 8, 51, None, 78, None, None, 25]
```

The *flat list format* always begins with the value None, so that the data value of the root is stored in index position 1. For any node at index position **i**, the left child is stored at index position **2*i**, and the right child is stored at index position **2*i+1**. A value of None in the list means there is no child at the corresponding index position.

Define a function called **create_tree_from_nested_list()** that takes a list of values in the nested list format and returns the corresponding BinaryTree object.

Define a function called **create_tree_from_flat_list()** that takes a list of values in the flat list format and returns the corresponding BinaryTree object.

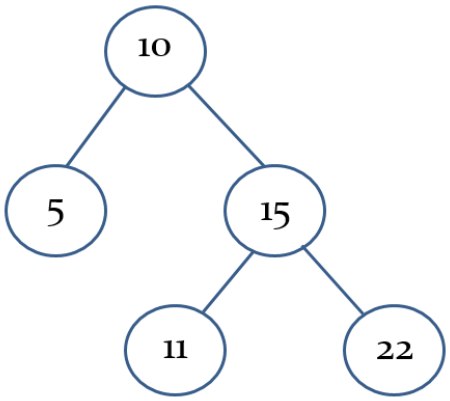
For example, the following code:

```
nested_list = [10, [5, None, None], [15, [11, None, None],
                                                [22, None, None]]]
my_tree = create_tree_from_nested_list(nested_list)
print(my_tree)
```

would produce the output:

```
10
(l)    5
(r)    15
(l)        11
(r)        22
```

which represents the tree illustrated below:



This exact binary tree could also be generated by the code:

```
flat_list = [None, 10, 5, 15, None, None, 11, 22]
my_tree = create_tree_from_flat_list(flat_list)
print(my_tree)
```

Define the two functions **create_tree_from_nested_list()** and **create_tree_from_flat_list()**.

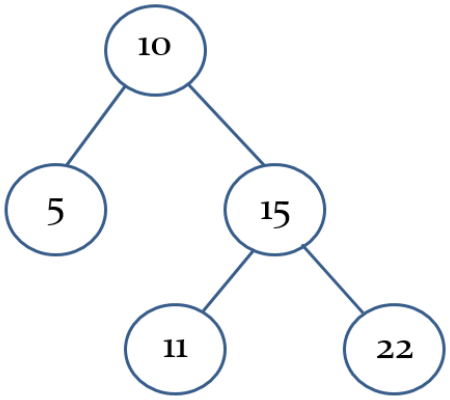
Question Five: Tree sum (11 marks)

If all of the data values in the nodes of a tree are numeric, then we may want to sum those values.

Define a function called **sum_tree()** which is passed a tree as input and which returns the sum of all of the data values in the nodes of the tree.

You can assume that all data values will be numeric.

As an example, consider the tree below:



The sum of all data values in this tree is: 63

This could be calculated using the following code:

```
flat_list = [None, 10, 5, 15, None, None, 11, 22]
my_tree = create_tree_from_flat_list(flat_list)
print('Sum of tree values =', sum_tree(my_tree))
```

which would produce the output:

```
Sum of tree values = 63
```

Question Six: Tree Traversals (11 marks)

Define the following three functions:

1. **pre_order()**
2. **in_order**
3. **post_order()**

Each function should take a binary tree as input and should return a list of the data values in that tree. The lists should be ordered in the same way that the tree elements would be visited in a pre-order, in-order and post-order traversal, respectively.

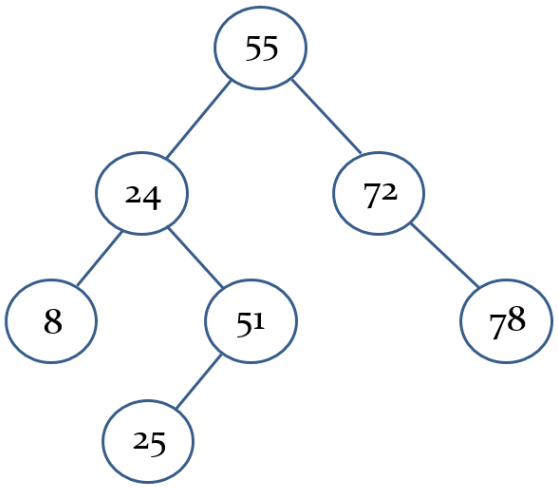
For example, the following code:

```
flat_node_list = [None, 55, 24, 72, 8, 51, None, 78, None, None, 25]
bst = create_tree_from_flat_list(flat_node_list)

pre_order_nodes = pre_order(bst)
in_order_nodes = in_order(bst)
post_order_nodes = post_order(bst)

print('Pre =', pre_order_nodes)
print('In =', in_order_nodes)
print('Post =', post_order_nodes)
```

which generates the tree:

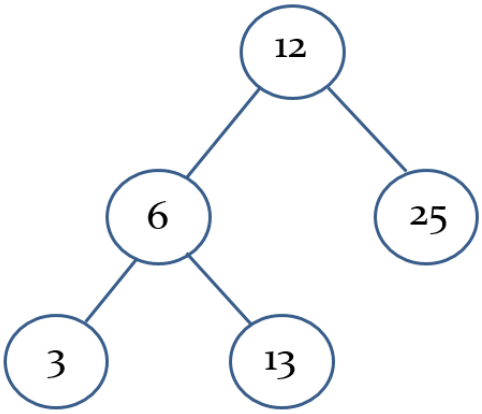


should produce the output:

```
Pre = [55, 24, 8, 51, 25, 72, 78]
In = [8, 24, 25, 51, 55, 72, 78]
Post = [8, 25, 51, 24, 78, 72, 55]
```

Question Seven: Is it a binary search tree? (11 marks)

Consider the following tree, which is *not* a binary search tree:



This could have been created in either of the following ways:

```
nested_node_list = [12, [6, [3, None, None],
                            [13, None, None]], [25, None, None]]
flat_node_list = [None, 12, 6, 25, 3, 13]

bst1 = create_tree_from_nested_list(nested_node_list)
bst2 = create_tree_from_flat_list(flat_node_list)
```

Define a function called **is_binary_search_tree()** that is passed a tree as input, and returns True only if the tree is a binary search tree. If the tree is not a binary search tree, the function should return False.

For example, the following code:

```
flat_node_list_1 = [None, 12, 6, 25, 3, 13]
bst1 = create_tree_from_flat_list(flat_node_list_1)
print(is_binary_search_tree(bst1))

flat_node_list_2 = [None, 12, 6, 25, 3, 10]
bst2 = create_tree_from_flat_list(flat_node_list_2)
print(is_binary_search_tree(bst2))
```

would produce the output:

```
False
True
```

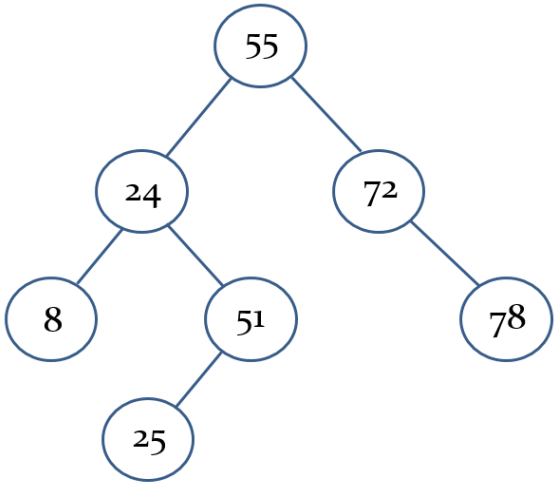
Question Eight: Insert position (11 marks)

If we have a *binary search tree*, then inserting a new node into the tree will result in the new node becoming either the left child or right child of an existing node in the tree.

For example, consider the tree created as follows:

```
flat_node_list = [None, 55, 24, 72, 8, 51, None, 78, None, None, 25]
bst = create_tree_from_flat_list(flat_node_list)
```

We could visualise this tree as shown below:



Now consider inserting the node 54. This would end up becoming the right child of node 51.

Or, if we inserted the node 49, this would become the right child of node 25.
Or, if we inserted the node 75, this would become the left child of node 78.

Define a function called **insert_position_in_bst()** that takes two inputs: a tree (representing a binary search tree) and a value to be inserted into the binary search tree.

The function should not modify the tree - rather it should simply return a string indicating where the node *would* be inserted in that tree. This string should indicate the parent of the new node, and whether the node would be a left or a right child of its parent. For example, using the code above, the following:

```
print('Insert (54) =', insert_position_in_bst(bst, 54))
print('Insert (49) =', insert_position_in_bst(bst, 49))
print('Insert (75) =', insert_position_in_bst(bst, 75))
```

should produce the output:

```
Insert (54) = right of 51
Insert (49) = right of 25
Insert (75) = left of 78
```

Question Nine: Linear and quadratic probing (11 marks)

A hashtable has been created, and is using the following simple hash function:

$$h(\text{key}) = \text{key} \% \textit{size}$$

where *size* is the capacity of the hashtable.

You need to define a function called **hash_with_probing()** which simulates a list of keys being inserted, in the order given, into the hashtable. The function should return a representation of the hashtable as a list - where the value *None* is used to represent unused positions.

The **hash_with_probing()** function will be passed three inputs:

1. the list of keys to be inserted (you can assume there will be no duplicate keys in the list)
2. the size of the hashtable (you can assume this will be a prime number)
3. either the word 'linear' or the word 'quadratic' to indicate the kind of probing being performed following a collision

You should then simulate the keys being inserted in the order given in the list, and using either linear or quadratic probing as specified.

The output of the function should be a *list* illustrating the used and unused positions of the hashtable.

For example, the code:

```
values = [26, 54, 94, 17, 31, 77, 44, 51]
linear = hash_with_probing(values, 13, 'linear')
print('A =', linear)

values = [26, 54, 94, 17, 31, 77, 43, 25]
quadratic = hash_with_probing(values, 13, 'quadratic')
print('B =', quadratic)
```

would produce the output:

```
A = [26, 51, 54, 94, 17, 31, 44, None, None, None, None, None, 77]
B = [26, None, 54, 94, 17, 31, None, None, 43, None, None, 25, 77]
```

IMPORTANT

Your own work

This is an individual assignment. You must write and debug all source code on your own. Under no circumstances should you copy source code from anyone else, or allow your work to be copied. Discussing ideas or algorithms in general is acceptable, but source code cannot be shared or copied at all. It was very disappointing to see seven students penalised for Asst 1.

There will be absolutely zero tolerance for plagiarism for this assignment. Course failure will result.

If you copy source code, under any circumstances for any part of this assignment, you are running an extremely big risk.

Copying code and making superficial changes to it is not the same thing as writing, debugging and testing the code yourself.

Submitting your assignment

When you have finished, you must submit Asst2.py to the Assignment Drop Box:

https://adb.auckland.ac.nz

prior to 9:00pm on Friday 12th February.

Your submitted functions will be marked automatically, so make sure that you adhere to the formats exactly as described in this document.

Good luck and enjoy!