COMPSCI 105 Summer School 2017
Assignment Two

Worth: 9% of your final grade
Deadline: 5:00pm, Friday 17th 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

For this assignment, you will be creating a **single source file** called **Asst2.py**. At the top of the Asst2.py source file, you will import the Node, Stack, Queue and BinaryTree classes. The source files for these four classes are provided to you. You should not modify these classes, as you will not submit them. You will only submit your source file Asst2.py for marking.

Asst2.py

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

def evaluate_expression(infix):
    .....
    .....

def generate_chain(start, n):
    .....
    .....

def move_node_to_end(values, value):
    .....
    .....

.....
.....
.....
```

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

All required functions must be present in the file, however if you have not implemented a function you may use the keyword "pass" as its definition. This will ensure that your code still executes successfully.

The general structure of the "Asst2.py" file that you will submit is shown in the diagram on the left.

A simple test program has been provided to help you, however you should test your code thoroughly with examples of your own.

You must submit this single source file, named "Asst2.py", to the Assignment Drop Box:

https://adb.auckland.ac.nz

prior to 5:00pm on Friday 17th February.

Paul Denny
January 2017

Question One: *Evaluate a mathematical expression* **(18 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 modify the order in which the operators are evaluated. An example of an expression in infix notation is the following:

```
2 * ( 30 + 7 )
```

which evaluates to:

```
74
```

In this question, you must write a function called **evaluate_expression()** that evaluates infix expressions. Fortunately, you can assume that the infix expression provided as input to the function is valid (although as an extension exercise, feel free to check for expression validity).

The input to the function will be a string. Each token in the string (whether an operator, an operand or a parenthesis) will have a *single space character* on either side of it (with the only exceptions being the start and the end of the string which do not include spaces). You may find the Python split() method useful for converting an input string into a list of tokens.

For this question, all operands must be integers (this also means that division should be calculated using the // operator in Python, and not /). There are 7 operators that may appear in the input expression that you must support. In addition to the four usual operators (+, -, *, /) you should also support the following three operators:

Operator	Description	Examples
^	this is the exponentiation operator, which raises a base to a power	2 ^ 10 = 1024 5 ^ 2 = 25
<	this is the "minimum" operator which evaluates to the *smaller* of the two operands	10 < 20 = 10 20 < 10 = 10 15 < 15 = 15
>	this is the "maximum" operator which evaluates to the *larger* of the two operands	10 > 20 = 20 20 > 10 = 20 15 > 15 = 15

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:

`1 + 100 > 200 = 200`	`40 > 15 < 35 + 10 = 40`
`1 + ( 100 > 200 ) = 201`	`( 40 > 15 < 35 ) + 10 = 45`
`2 ^ ( 1 + 3 ^ 2 ) = 1024`	`2 * ( ( 4 < 2 + 3 ) + 3 * 4 ) = 32`

You can assume that the infix expression, provided as input to the function, will be valid. That is, the brackets will be balanced and there will be the correct number of operators and operands. You do not need to check this!

As an example, the following code:

```
expressionA = '2 ^ ( 1 + 3 ^ 2 )'
expressionB = '( 3 * 5 ) - ( 1 > 2 > 3 < 4 )'
expressionC = '4 ^ ( 10 < 2 ) / 5 + 100'

print('Result A =', evaluate_expression(expressionA))
print('Result B =', evaluate_expression(expressionB))
print('Result C =', evaluate_expression(expressionC))
```

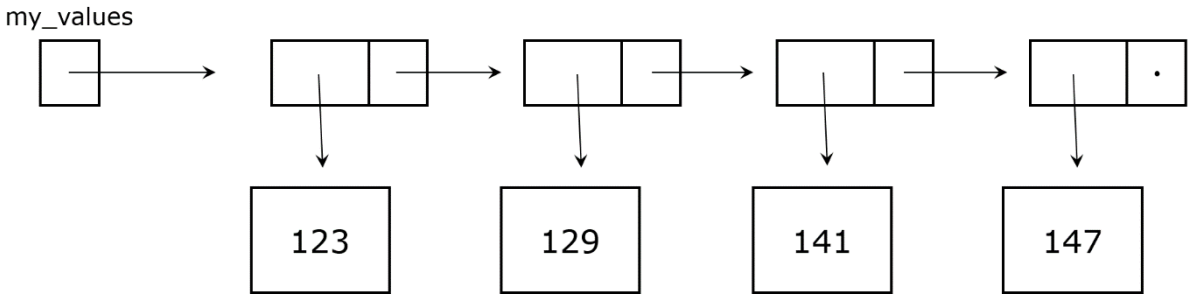
should produce the output:

```
Result A = 1024
Result B = 12
Result C = 103
```

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

Question Two: *Create a chain* **(13 marks)**

For this question you will be *creating* 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 Node class for more detail.



a chain of length 4, with a starting value of 123

Define a function called **generate_chain()** which takes two integer inputs. The first input, *start*, specifies the starting value of the first node in the chain. The second input, *n*, indicates how many nodes in total should be in the chain. The value of each node in the chain is calculated based on the value of the previous node. Each value is calculated as the previous value *plus* the sum of the digits of the previous value.

Once the chain has been constructed correctly, the function should return the chain.

For example, the following code:

```
my_values = generate_chain(409, 5)

# Print the chain
current = my_values
while current != None:
    print(current.get_data())
    current = current.get_next()
```

should produce the output:

```
409
422
430
437
451
```

Question Three: *Modify a chain* **(13 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 **move_node_to_end()** which is passed two inputs. The first input is the reference to the first node in a chain of nodes. The second input is a data value to search for in the chain. If the data value is found in the chain, then the corresponding Node must be moved to the very end of the chain.

For example, if the chain contains the following nodes:

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

and this chain is passed to the function along with the input value of 3, then after the function is called, the nodes should be in the following order:

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

Please note, you must not modify the *data* value of any node. Instead, you must only update the *next* references. Each node object should continue to point to the same data value. The following example illustrates this point, as it prints the id of each node, along with its value.

```
def print_chain_and_ids(chain):
    current = chain
    while current != None:
        print(id(current), current.get_data())
        current = current.get_next()

a = Node('first')
b = Node('middle')
c = Node('last')
a.set_next(b)
b.set_next(c)

print_chain_and_ids(a)

move_node_to_end(a, 'middle')

print_chain_and_ids(a)
```

The output here is (notice that although the order of the nodes in the chain has changed, each node object continues to point to the same data value as before):

```
49167760 first
49384112 middle
49385072 last
49167760 first
49385072 last
49384112 middle
```

You can assume that the node to move will <u>not</u> be the very <u>first</u> node in the chain, and the chain will contain at least two nodes. If the value to search for (i.e. the second input to the function) is *not* found in the chain, then you should *not* modify the chain. You can also assume that all Node values are distinct (in other words, there are no repeated values in the chain).

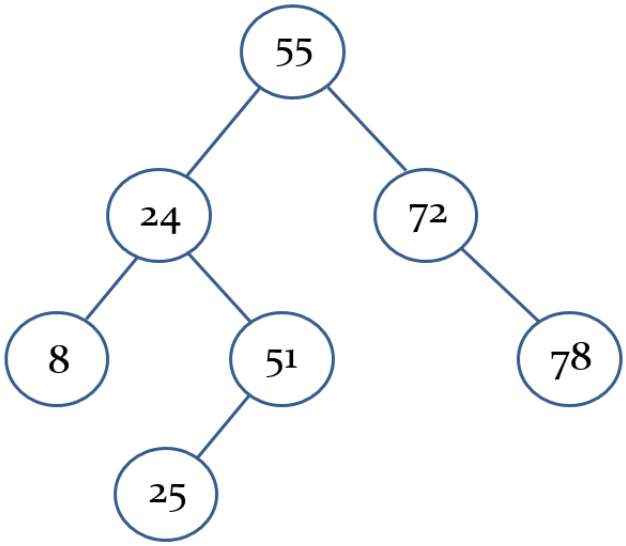
Question Four: *Creating trees* **(13 marks)**

Carefully examine the minimal BinaryTree class provided. Each instance of this class has three attributes: *data*, *left* and *right*. The *data* attribute stores the data value for a given node in the tree. The *left* and *right* attributes refer to subtrees that are valid BinaryTree objects, and which represent the left and right subtrees of the node respectively. If a node does not have a subtree, then the corresponding *left* or *right* attribute will have the value *None*.

The BinaryTree 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 in the diagram on the right. 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.

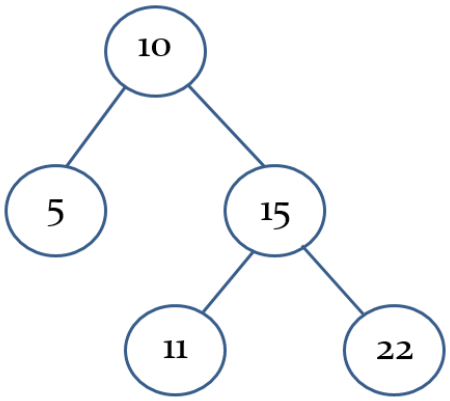
For example, the following code:

```
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
```

as the generated tree would have the structure 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)
```

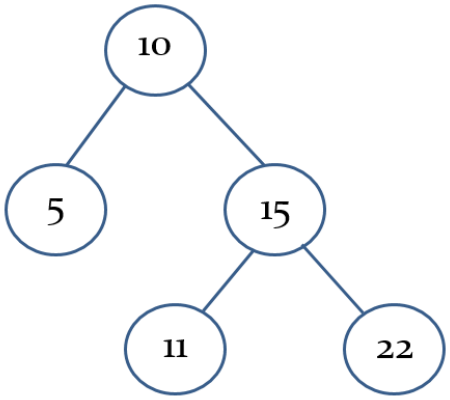
Define the two functions **create_tree_from_nested_list()** and **create_tree_from_flat_list()**.

Question Five: *Breadth-first traversal* **(13 marks)**

Now, assume that a binary tree has been created (perhaps by one of the functions you just defined in the previous question) and you would like to visit all of the nodes in the tree, one at a time.

Define a function called **breadth_first()** which takes a binary tree as input, and which returns a list of the data values in the tree in *breadth-first* order (i.e. as read from top to bottom, and left to right).

For example, consider the following binary tree:



The code below creates this tree, and then calls the breadth_first() function:

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

print("Breadth first =", breadth_first(my_tree))
```

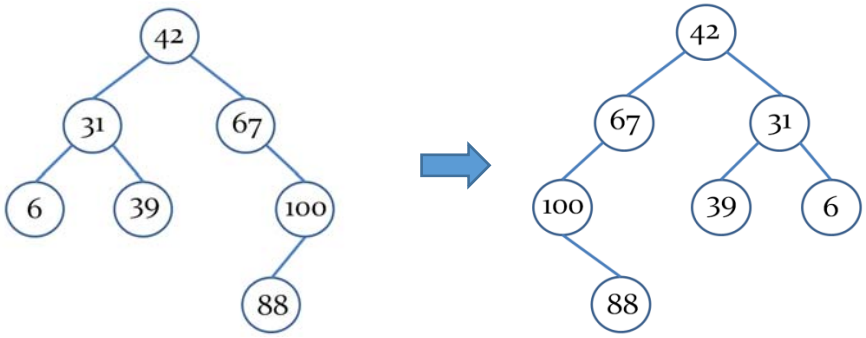
This should produce the output:

```
Breadth first = [10, 5, 15, 11, 22]
```

To perform the breadth-first traversal, you should make use of the Queue class that is provided to you.

Question Six: *Reflection* **(13 marks)**

Define a function called **mirror()** which takes a binary tree as input, and which *modifies that tree* by *mirroring* its structure. After mirroring, the tree should look like it is being viewed in a mirror.



before calling mirror() *after calling mirror()*

For example, the following code:

```
flat_list = [None, 42, 31, 67, 6, 39, None, 100, None, None,
                                          None, None, None, None, 88]
my_tree = create_tree_from_flat_list(flat_list)

print('Original =')
print(my_tree)

mirror(my_tree)

print('Reflected =')
print(my_tree)
```

would generate the output:

```
Original =
42
(l)    31
(l)        6
(r)        39
(r)    67
(r)        100
(l)            88
Reflected =
42
(l)    67
(l)        100
(r)            88
(r)    31
(l)        39
```

Question Seven: *Collision resolution* **(13 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 **hashing()** 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 keys are shown in their positions and the value *None* is used to represent unused positions.

The **hashing()** function will be passed four 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. the word 'linear', 'quadratic' or 'double' to indicate the kind of probing to be performed following a collision
4. if the third input is 'double', this fourth input represents the value, *q*, that will be used in the second hash function during double-hashing. This second hash function will be: h'(key) = *q* – (key % *q*). Otherwise, if linear or quadratic probing is performed, this fourth input will be *None*.

You should then simulate the keys being inserted in the order given in the list, into a hash table of the given size. To resolve collisions, you should use linear probing, quadratic probing or double-hashing as specified by the third input.

The output of the function should be a *list* illustrating the used and unused positions of the hashtable (used positions display key values, unused positions display None). For example, the code:

```
values = [26, 54, 94, 17, 31, 77, 44, 51]

linear = hashing(values, 13, 'linear', None)
quadratic = hashing(values, 13, 'quadratic', None)
double = hashing(values, 13, 'double', 5)

print('Linear =\n', linear)
print('Quadratic =\n', quadratic)
print('Double =\n', double)
```

would produce the output:

```
Linear =
 [26, 51, 54, 94, 17, 31, 44, None, None, None, None, None, 77]
Quadratic =
 [26, None, 54, 94, 17, 31, 44, None, 51, None, None, None, 77]
Double =
 [26, None, 54, 94, 17, 31, 44, 51, None, None, None, None, 77]
```

Please note:

The following two questions are bonus questions, and each is worth just 2% (2 out of 100) of the marks for this assignment.

Question Eight: *Balanced binary search tree* **(bonus - 2 marks)**

Define a function called **create_binary_search_tree()** which takes a list of unique integers as input, and which creates and returns a valid binary search tree that contains all of the integers and has *minimum depth*.

For example, the following code:

```
print('Tree A')
my_values = [1, 2, 3, 4, 5, 6, 7]
result = create_binary_search_tree(my_values)
print(result)

print('\nTree B')
my_values = [1000, 100, 10, 1]
result = create_binary_search_tree(my_values)
print(result)
```

would produce the output:

```
Tree A
4
(l)    2
(l)        1
(r)        3
(r)    6
(l)        5
(r)        7

Tree B
10
(l)    1
(r)    100
(r)        1000
```

Question Nine: *Wasting time* **(bonus - 2 marks)**

As soon as summer school is over, you are going on holiday. You are having a difficult time trying to plan how you will spend your time. So many activities, so little time!

You have a list of various activities that you would like to do, and each activity takes a certain amount of time. However, you also have a fixed amount of time available, and you can't start a new activity if you don't have the time to finish it. You don't mind which activities you do, but you don't want to waste time.

Essentially, you want to pick a specific set of activities where the total time required to complete those activities is as close as possible to the time you have available.

For example, let's say you have 700 minutes available, and there are 8 activities you are considering, each taking the following times:

155, 183, 281, 192, 111, 267, 154, 134

Let's say you choose these four activities: 183, 111, 267 and 134. Choosing these four activities means you will only waste 5 minutes (as 700 – (183 + 111 + 267 + 134) = 5). In fact, this is the *best that you can do* – there are no other combinations of activities you can take that will waste less time.

Define a function called **min_waste()** which is passed two inputs:

1. the total amount of time you have available
2. a list of the times needed for the various activities

Your function should calculate and return the *smallest* amount of time that will be wasted on your holiday if you select the right combination of activities.

For example, the following function call:

```
wasted = min_waste(700, [155, 183, 281, 192, 111, 267, 154, 134])
print('Time left =', wasted)
```

will produce the output:

```
Time left = 5
```

and the following function call:

```
wasted = min_waste(100, [50, 20, 10, 39])
print('Time left =', wasted)
```

will produce the output:

```
Time left = 1
```

Getting started with Assignment 2

To get started, you should download the resource files from Canvas:

- Asst2.py
- TestProgram.py
- BinaryTree.py
- Node.py
- Queue.py
- Stack.py

The Asst2.py source file is a skeleton version of the file you must submit for marking. You should implement the required functions in this file (don't remove any of the required functions, as this will prevent the marking program from running). If you don't attempt a particular question, that is fine, but remember to leave the function definition in place (you can use the word "pass" to leave an empty implementation for a function).

As you implement "Asst2.py", you should add appropriate tests to "TestProgram.py" to test your functions.

IMPORTANT

Your own work

This is an individual assignment. You must write the code for your classes 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 you must write all of the code you submit yourself.

This will be enforced, so please don't try to copy and make superficial changes to make your code "appear" different to the code you copied. It is simple to detect such copying and modifications, and all submissions for this assignment will be checked.

Submitting your assignment

When you have finished, submit your source file Asst1.py to the Assignment Drop box:

https://adb.auckland.ac.nz/

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

Have fun!