CS-UY 1134: Homework 3
Summer 2019

Due August 5, 11:55 PM

Programming Part (100 Points)

Submission Instructions

Write your solution to each programming problem in a separate ".py" file. Name your files "P1.py", "P2.py", etc. There should be 6 files total. In each file, include your implementation of the given functions. Make sure the name of your function is *exactly* the same as the the name we provide in the problem. If a problem has sub-parts, include your solutions to all parts in the same file.

Submit all of your ".py" files (there should be 5 of them, named "P1.py", "P2.py", etc.) to the Gradescope assignment "Assignment 3 – Programming."

Problem 1 (20 points)

Write a class, `LinkedQueue`, which implements the Queue ADT using a Doubly Linked List. Use the `DoublyLinkedList` class provided, which was presented in lecture. Your implementation will use an instance of the `DoublyLinkedList` class as a data member.

Implementation Requirements

1. Data Members: A `DoublyLinkedList` object. You may use $\Theta(1)$ space for additional data members if desired (but you may not use any other non-constant size data structures, such as arrays or trees).

2. Running Time: All operations must run in *worst case* $\Theta(1)$ time.

As a reminder, since you are implementing the *Queue ADT*, you must implement all of the Queue operations. The documentation for your class will look like:

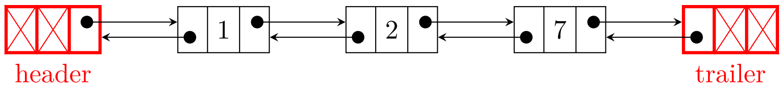
```python
class LinkedQueue:
    def __init__(self):
        '''Initialize an empty queue'''

    def __len__(self):
        '''Return the number of elements in the stack''''

    def is_empty(self):
        '''Returns True if the stack is empty'''

    def enqueue(self, elem):
        '''Add elem to the queue (at the back)'''

    def dequeue(self):
        '''Remove and return the item at the front of the queue
        or raise an Exception if the queue is empty'''

    def first(self):
        '''Return the item at the front of the queue without returning it
        or raise an Exception if the queue is empty'''
```

Problem 2 (20 points)

The most common way numbers are represented in a computer is using a fixed number of bytes (a common size is 4 bytes, which are 32-bit integers; 64-bit integers, using 8 bytes, are also sometimes used). The bytes are stored directly in memory. This makes arithmetic (adding, multiplying, etc.) with the numbers very fast, as the processor can perform these calculations directly as a single instruction. However, this limits the number of possible integers that can be stored (for example, with 4 bytes, only 2^{32} different possible values can be represented).

To allow for arbitrarily large numbers, some languages represent numbers in special numeric objects that can represent numbers of arbitrary size. This adds more memory overhead and makes the operations slower, but allows for arbitrarily large integers.

In this problem, you will suggest a data structure for representing arbitrarily large integers. To do this, we will represent an integer value as a linked list of its digits. For example, the number 127 will be represented as a length-3 linked list, containing the values 1, 2, and 7 (in that order):



Write the class `Integer` which represents numbers in this way. You should overload the addition operator to support adding to numbers. Do this by completing the class definition below:

```
class Integer:
    def __init__(self, num_str):
        '''Initializes the Integer object to represent the number
        given by the string num_str'''

    def __add__(self, other):
        '''Create and return an Integer object that represents the sum
        of self and other, both of which are also Integer objects'''

    def __repr__(self):
        '''Returns the string representation of the Integer object self'''
```

Note that the addition adds two objects of type `Integer` (this is *not* Python's built-in `int`, but the class you are defining!) and returns a new `Integer` object.

As an example, your class should follow this behavior:

```
>>> num1 = Integer('127')
>>> num2 = Integer('974')
>>> num3 = num1 + num2
>>> num3
1101
```

As noted above, the addition operator should return an `Integer` object. This means in the example above, `num3` should have type `Integer`.

Note that you should use the "grade school" technique for adding numbers. **DO NOT** convert the `Integer` objects to built-in `int`s, do addition with those, and then convert the result back into an `Integer` object. This defeats the point of the problem; **you will lose ALL the points for the problem if you do this**.

Problem 3 (20 points)

Implement a function that merges two sorted linked lists. Write the function as:

```
def merge_linked_lists(srt_lnk_lst1, srt_lnk_lst2)
```

The function should take two doubly linked lists (that is, objects of type `DoublyLinkedList`. The elements in each list are in sorted (increasing) order.

When called, the function should create and return a *new* doubly linked list object that contains all the elements of both lists. The returned list should be sorted in increasing order. The function **should not** modify either of the two given lists. It should only return a new list.

For example, if `srt_lnk_lst1` is `[1<-->3<-->6]` and `srt_lnk_lst2` is `[2<-->3<-->5<-->10]`, calling `merge_linked_lists` should return: `[1<-->2<-->3<-->3<-->5<-->6<-->10]`.

You should implement this using recursion. The function `merge_linked_lists` will not be recursive itself, but it will define and call a recursive helper function, `merge_sublists`. Below is a skeleton of what your implementation should look like:

```
def merge_linked_lists(srt_lnk_lst1, srt_lnk_lst2):
    def merge_sublists(____________________):
        # Your code for merge_sublists goes here
        # You will have to choose the parameter(s) for merge_sublists as well
        # Your helper function should merge "sublists" of the original lists

    # You will have to determine what the initial call to the helper function should be
    return merge_sublists(____________________)
```

Implementation Requirements

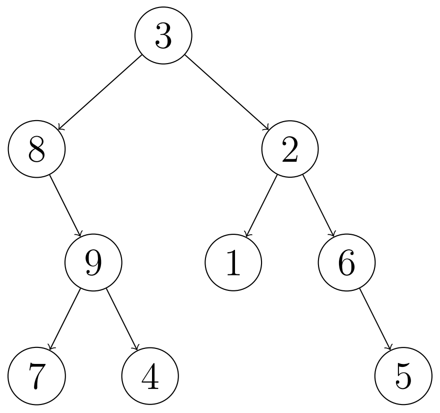
1. `merge_sublists` **must be recursive**. You will have to determine what parameters it should take and the initial call made to this function by `merge_linked_lists`.
2. Your solution should run in **linear** time. That is, if the two input lists have sizes n_1 and n_2, the merge would run in time $\Theta(n_1 + n_2)$.

Problem 4 (20 points)

Define the function:

```
def min_max(bin_tree)
```

which takes a binary tree (a `LinkedBinaryTree` object containing numerical data in all the nodes (you may assume in your implementation that this is the case), and returns the minimum value and maximum value of the tree as a tuple. For example, given that `bin_tree` is the following binary tree:



the function call `min_max(bin_tree)` should return `(1,9)` because 1 is the minimum value in the tree, and 9 is the maximum. Note the order of the tuple is (min, max) — make sure you return the values in the correct order!

Note: The given tree `bin_tree` is not necessarily a binary search tree!

Implementation Requirements

1. Use **recursion**. Define a helper function:

   ```
   def subtree_min_max(subtree_root)
   ```

   that takes as a parameter a `Node` object representing the root of the subtree to consider, and return the (min, max) tuple containing the minimum and maximum values in that subtree.

2. You are **not allowed** to use any of the methods that are already defined in the `LinkedBinaryTree` class. Specifically, you may not use *any* of the traversals to iterate over the tree.

3. Your function should run in *linear time*.

4. The minimum and maximum are not defined for an empty tree. If the function `min_max` is called on an empty tree, you should raise an Exception (seriously; you will lose points if you don't do this)

Think about (but you don't have to submit your response): How would this problem change if `bin_tree` were a binary search tree? Would the problem be easier or harder? Think about the running time of finding the minimum and maximum in a binary search tree. How does it compare to the above (which should be linear time)?

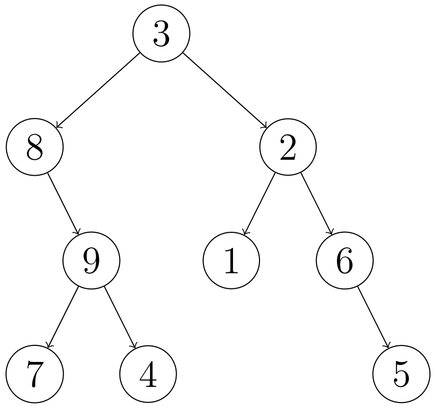
Problem 5 (20 points)

Implement the function:

```
def iterative_inorder(bin_tree)
```

which takes as a parameter a binary tree object `bin_tree` and produces a generator that yields the *values* in the tree in the "in-order" order.

For example, consider the tree `bin_tree` from Problem 4, shown again below:



Suppose we run the code below:

```
for val in iterative_inorders(bin_tree):
    print(val, end=' ')
print()
```

The output of this code should be: `"8 7 9 4 3 1 2 6 5"`.

Implementation Requirements

1. Your solution should be purely iterative. Recursion is **not allowed**. Seriously — you will get **zero** points on this problem if you use recursion.
2. Do not call any helper functions or methods. You should not call any methods in the `LinkedBinaryTree` class. All the work should be done directly in the `iterative_inorder` method.
3. Your implementation must run in **linear time**.

Hint: Use a Stack. You may import and use the `ArrayStack` class from lecture if you want (or simply use, e.g., `append` and `pop` with a list). [It is *possible* to do with only $O(1)$ extra memory (i.e., without a Stack or any other data structure), but it is much harder and it is not a requirement to do so. Using a Stack will be much easier.]

Extra Credit

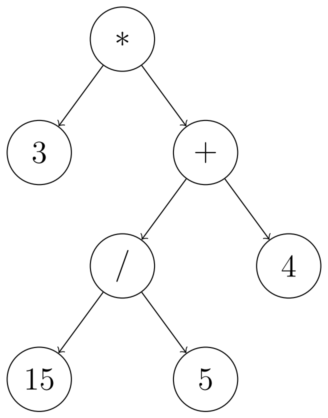
Problem 6

In this problem, we will explore "pretix" notation for arithmetic. Recall the postfix notation we discussed in lecture, in which each operator is written *after* both of its operands. In prefix notation, we do the opposite: operators are written *before* their operands. That is, in prefix notation, we first write the "left" operand, then the "right" operand, and finally the operator.

This problem will also work with "expression tress", another way of representing arithmetic expressions. In an expression tree, an arithmetic expression is represented as a binary tree, where a node is either a number or an operator; the two children of an operator node represent the operands of that operation.

The prefix notation and expression trees are described in more detail in the written part. It is srongly recommended that you do the written part first to familiarize yourself more with prefix notation and expression trees, which you will work with in this problem.

For the following problems, we will use the tree below as an example:



(a) Write a function

```
def prefix_expression(expr_tree)
```

which takes as an argument an expression tree (an instance of the `LinkedBinaryTree` class), and outputs a string representing the expression in *prefix* notation. Tokens should be separated by a single space. For example, when given the tree above, this function should return `"* 3 + / 15 5 4"`. Assume that all nodes representing operators store the operators as `string`s, all operands are positive integers, and all nodes representing operands store the operands as `int`s. You may further assume the tree is a valid expression tree.

(b) Write a function

```
def postfix_expression(expr_tree)
```

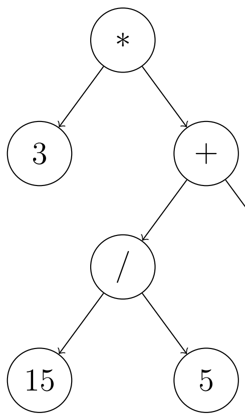
which takes as an argument an expression tree (an instance of the `LinkedBinaryTree` class), and outputs a string representing the expression in *postfix* notation. Tokens should be separated by a single space. For example, when given the tree above, this function should return `"3 15 5 / 4 + *"`. Assume that all nodes representing operators store the operators as `string`s, all operands are positive integers, and all nodes representing operands store the operands as `int`s. You may further assume the tree is a valid expression tree.

(c) Implement the function

```
def create_expr_tree(prefix_expr_str)
```

which takes a string representing an arithmetic expression in **prefix notation**. The function should construct and return a `LinkedBinaryTree` object which stores the expression tree representing the given arithmetic expression.

For example, calling `create_expr_tree("* 3 + / 15 5 4")` should produce the following tree:



Note: Assume the following about the format of `prefix_expr_str`

1. All operators will be one of `"+"`, `"-"`, `"*"`, or `"/"`
2. All operands will be positive integers
3. The string will not contain any tokens that are not either a valid operator or operand.
4. The tokens in the expression will be separated by a space

Implementation Requirements

1. Nodes containing operators should have the operators as a `string`, and nodes containing operands should store the operands as an `int`.
2. The runtime for creating the expression tree should be **linear** (in the length of the prefix expression string)
3. You are **not allowed** to use a Stack.

Hint: Notice that the first token in a prefix expression is an operator, which will be the root of the expression tree. Some subexpression representing its left operand immediately follows the operator. Following the subexpression for the left operand is a subexpression representing the right operand.

Approach this problem with recursion. While there is more than one way to solve the problem, we will briefly describe one way of approaching it. First, use `split` on the expression string to get a list of tokens, and then call a helper function. The helper function is:

```
def create_expr_tree_helper(prefix_expr, start)
```

which takes the *list* of tokens that make up a prefix expression, as well as a `start` index indicating the start position of the current subexpression being considered. The helper function will **recursively** create a subtree representing the subexpression being considered (which start at position `start`). The helper function will return two values (as a tuple): the root node to the subtree that was created, and the size of the subtree. The size of the subtree helps to determine how long the subexpression that made up the constructed tree was, allowing you to know where the next subexpression starts.

Written Part (Extra Credit)

Submission Instructions

Answer the following questions and submit your solutions to Gradescope, under "Assignment 3 – Written (Extra Credit)." Typed answers are preferred, but you may handwrite your answers and submit a PDF image of your answers *as long as you write neatly* and your answers are readable. **You may be marked wrong if we can't read your answer.**

Overview of Prefix Notation and Expression Trees

Recall the postfix notation of arithmetic we discussed in lecture. Specifically, recall that in the postfix notation, we write the operator *after* both of its operands (writing the "left" operand first, then the "right", and finally the operator).

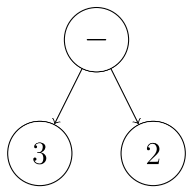
As a brief example for the sake of reminder, the expression `"3 2 -"` is the postfix form of $3 - 2$, and so `"3 2 -"` evaluates to 1. A more complex example might be: `"7 2 + 4 1 - /"`, which evaluates to 3, since it is equivalent to the "infix" expression $(7 + 2)/(4 - 1)$.

In this problem we will explore *prefix* notation, in which operators are written *before* their operands. In this notation, we write the operator first, then its "left" operand, and then the "right" operand. For example, `"- 3 2"` is the prefix expression that is equivalent to the first example above: `"3 2 -"` in postfix, $3 - 2$ in infix, and the expression evaluates to 1. As a slightly more complex example, the expression `"/ + 7 2 - 4 1"` would be the pretix expression for the second example above: `"7 2 + 4 1 - /"` in postfix, $(7 + 2)/(4 - 1)$ in infix, and the expression evaluates to 3.

In this problem we will create an arithmetic "expression tree". In an expression tree, each leaf node represents a value in the expression, and each internal (i.e., non-leaf) node represents an operator. The two children of an operator represent its two operands.

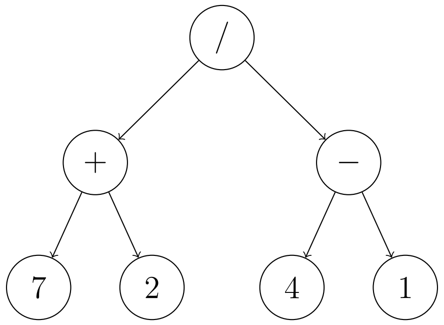
The expression tree is not specific to any notation. Rather, it gives an alternate way of representing an expression (namely, as a tree), which complements the different notations for writing the expression.

For example, our first prefix expression was `"- 3 2"` (or, equivalently, `"3 2 -"` in postfix). Represented as an expression tree, this is:



Notice how this is not specific to any expression. Rather it is an alternate way to represent both the operator, subtraction, and the two operands, 3 and 2, without using a specific notation.

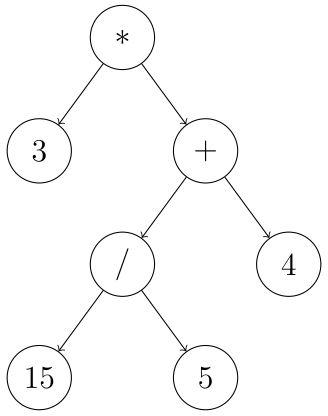
Here is another, more complicated example:



This tree has three operators. The `+` operator has operands 7 and 2, and so it represents the addition of 7 and 2 (which is, of course, 9). The `-` operator has operands 4 (on the left) and 1 (on the right), representing a subtraction of 1 *from* 4 (that is, $4-1$ in infix, the result of which is 3). Finally, the operator `/` has the `+` and `-` operators as children, representing that the "left" operand of the division is 9, the result of the evaluation of the `+` operator (in more words, this is the result of evaluating the expression represented by the subtree rooted at the `+` node). Similarly, the `/` has as its "right" operand the result of the `-` operator, which was 3. So, this root node represents the operation of dividing 9 (the left operand) by 3 (the right operand).

You may have noticed that this last example is in fact the expression tree for the prefix expression `"/ + 7 2 - 4 1"`, the same example from before.

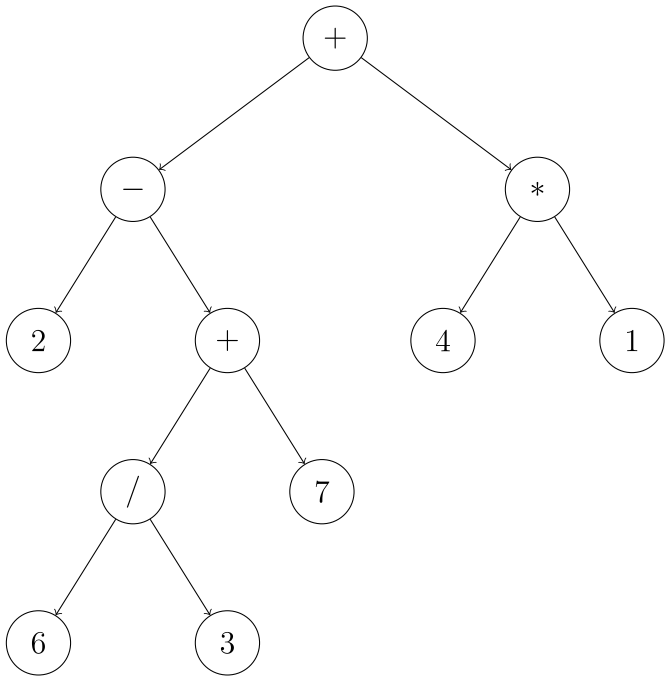
As a further example, the expression tree for the prefix expression `"* 3 + / 15 5 4"` is the following tree:



Another thing to note is that the expression tree is defined somewhat recursively. That is, its two children are the roots of some subtree that is also an expression tree. The values that the expressions represented by these two subtrees evaluate to are the operands. In the example above, the "right operand" of the $*$ operator is its right child, which is actually an operator itself: so the right operand of the $*$ is actually the result of evaluating the expression represented by the subtree rooted at the $+$ (that is, the right child of $*$).

Problems

1. Consider the following expression in infix notation: $((21+3)/(6*(4-2)))$.
   (a) Draw the expression tree for this expression.
   (b) Give the prefix notation for the same expression.
   (c) Give the postfix notation for the same expression.
2. For the expression tree below, answer the following questions:
   (a) Write the expression represented by this tree in infix notation (you may need to include parentheses to eliminate ambiguity).
   (b) Write the expression in postfix notation
   (c) Write the expression in prefix notation


+
−
*
2
+
4
1
/
7
6
3