

Question 1. [6 MARKS]

Define each of the following terms.

Part (a) [2 MARKS] Abstract Data Type

The definition of a data structure defined only in terms of the operations which can be performed on it without regard to implementation.

Part (b) [2 MARKS] Stack

An abstract data type with 5 operations defined on it: push, pop, peek, size, and empty.

Items are added/removed (by push and pop respectively) in a Last In, First Out order (LIFO).

Part (c) [2 MARKS] Queue

An abstract data type with 5 operations defined on it: enqueue, dequeue, front, size, and empty.

Items are added/removed (by enqueue and dequeue respectively) in a First In, First Out order (FIFO).

Question 2. [8 MARKS]

In lecture, we wrote a method which tested if a string of opening and closing parentheses was balanced. Write a method `count_balanced(n)` which returns how many strings of length n composed only of the two characters '(' and ')' are balanced.

For example, there are 5 such strings of length 6:

`()()()`, `()(())`, `((()))`, `((()()))`, `((()()))`

Hint: All (non-empty) such strings are of the form (A)B where A and B are both (potentially empty) strings of properly balanced parentheses.

```
def count_balanced(n):
    '''Return how many properly balanced strings of ( and ) are of
    length n.'''

    if n % 2 == 1:
        # No solutions if n is odd.
        return 0
    if n == 0:
        # Only the empty string.
        return 1
    ret = 0
    for i in range(n-1):
        # Using the hint, (A)B, |A| = i, |B| = n - 2 - i
        ret += count_balanced(i) * count_balanced(n - 2 - i)
    return ret
```

Question 3. [12 MARKS]

A Dictionary is an ADT which stores a collection of key/value pairs. We would like to implement a Dictionary using a linked list which contains the key/value pairs. Consider the following definition of this class:

```
class Node(object):
    def __init__(self, key, value):
        '''Create a new node representing the pair (key,value).'''
        self.key = key
        self.value = value
        self.next = None

    def __str__(self):
        '''Return a string in of the form (key,value).'''
        return '(' + str(key) + ', ' + str(value) + ')'

class Dictionary(object):
    def __init__(self):
        '''Create an empty Dictionary.'''
        self._head = None

    def get(self, key):
        '''Return the value associated with key in the Dictionary, or None if
        there is no association.'''
        pass

    def set(self, key, value):
        '''Set key to be associated with value.'''
        pass
```

Part (a) [4 MARKS] Fill in the code for the `get(self, key)` method.

```
def get(self, key):

    current = self._head
    while current is not None:
        if current.key == key:
            return current.value
        current = current.next
    return None
```

Part (b) [6 MARKS] Fill in the code for the `set(self, key, value)` method.

```
def set(self, key, value):  
  
    current = self._head  
    while current is not None:  
        if current.key == key:  
            # Update value associated with key.  
            current.value = value  
            return  
        current = current.next  
    # key not found; insert to front.  
    new_node = Node(key, value)  
    new_node.next = self._head  
    self._head = new_node
```

Part (c) [2 MARKS] What is the worst case runtime of calling `set` n times on an initially empty dictionary? What is the best case runtime? Use \mathcal{O} notation, and briefly justify your answer.

The best case runtime is $\mathcal{O}(n)$; this occurs when calling `set(1,2)` n times, each call of which runs in time $\mathcal{O}(1)$.

The worst case runtime is $\mathcal{O}(n^2)$; this occurs when calling `set` with any n distinct keys, resulting in a quadratic runtime.

Question 4. [8 MARKS]

For part 3 of Assignment 1, you converted a string of opening tags, closing tags, and words into a more structured format of nested lists.

For example, given the following input string:

```
'Here are some <tag><CAPSLOCK>TAGS</CAPSLOCK></tag><bold> example </bold>'
```

You produced the nested list structure (line breaks for display purposes only):

```
['Here', 'are', 'some',
 ['<tag>', ['<CAPSLOCK>', 'TAGS', '</CAPSLOCK>'], '</tag>'],
 ['<bold>', 'example', '</bold>']]
```

We would like to write a method that converts the nested list structure back into a single string of space separated tags/words. Continuing with the example, we would like to produce:

```
'Here are some <tag> <CAPSLOCK> TAGS </CAPSLOCK> </tag> <bold> example </bold>'
```

Note the lack of trailing/leading spaces.

Write a recursive method `flatten(parsed_document)` which takes as input a nested list structure, and returns a flattened string representation of its contents. You are not required to check the validity of `parsed_document`.

You may find the `isinstance` method useful. In particular, you can test if a variable `x` is a reference to a list by checking `isinstance(x, list)` and test if `x` is a string by checking `isinstance(x, str)`.

```
def flatten(parsed_document):
    '''Return a string representation of parsed_document.'''

    flat_string = ''
    for x in parsed_document:
        if isinstance(x, str):
            # Add to the end of the string.
            flat_string += ' ' + x
        else:
            # Must be a list.
            flat_string += ' ' + flatten(x)
    # Return the string with leading spaces removed.
    # Could also have checked len(flat_string) != 0
    # before adding leading space earlier.
    return flat_string.strip()
```

