Question 1. [16 MARKS]

Briefly answer the following questions.

Part (a) [2 MARKS]

What properties must a tree satisfy for it to be a min-heap?

It must be a complete binary tree with the property that the value stored on each node is not greater than the values stored on its children (if they exist).

Part (b) [2 MARKS]

What are the **worst case** runtimes of the insertion and removal methods for a binary search tree and a min-heap, which initially contain n elements? Use $\mathcal{O}$ notation.

For a min-heap, insertion and removal are both $\mathcal{O}(\log n)$. For a binary search tree, both are $\mathcal{O}(n)$.

Part (c) [2 MARKS]

One way to sort a list of n distinct numbers is to insert them into a binary search tree, and then perform an inorder traversal. Give two reasons why heap sort is a better sorting algorithm.

First, heap sort can be done in-place; second, the worst case runtime of the BST sorting algorithm is $\mathcal{O}(n^2)$, whereas heap sort is $\mathcal{O}(n \log n)$.

Part (d) [3 MARKS]

Examine the follow Python program, and determine what it will output. If you are unaware of what an exception will exactly output or its exact name, a short description of the exception will suffice.

```
def trouble(x):
    '''A troublesome function, up to no good.'''
    try:
        # Integer division.
        print 3 / x
        trouble(x - 1)
    except IndexError, e:
        print 'Error in the index.'
    else:
        print x
    finally:
        print 'finally!'

if __name__ == '__main__':
    try:
        trouble(2)
    except Exception, e:
        print e
    else:
        print 'That was not so bad...'
    finally:
        print 'The end.'
```

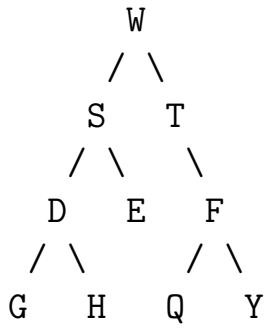
Output:

```
1
3
finally!
finally!
finally!
integer division or modulo by zero
The end.
```

Part (e) [3 MARKS]

Let T be a binary tree. The inorder traversal of T is `GDHSEWTQFY`. The postorder traversal of T is `GHDESQYFTW`. Can T exist? If not, explain why; if yes, construct T.

There is a unique solution for T:



Part (f) [2 MARKS]

Suppose that you are given a list `L` of length n such that each element of `L` is either a 0, 1 or 2. Describe an algorithm to sort `L` with worst case runtime $\mathcal{O}(n)$. You do not need to write any code for this.

Have three counters, count how many times 0, 1 and 2 appear in `L` (say a, b, c) and then set the first a elements of `L` to be 0, the next b elements to be 1, and the next c elements to be 2.

Part (g) [2 MARKS]

Circle True or False for each of the following statements (recall that $X \subseteq Y$ is read "X is a subset of, or equal, to Y"):

[**True** — False] $\mathcal{O}(n^2 \log n) \subseteq \mathcal{O}(3n^3)$

[**True** — False] $\mathcal{O}(n^{100} + 3n^2 + \frac{1}{n^5}) \subseteq \mathcal{O}(2^n)$

[True — **False**] $\mathcal{O}(n^5) \subseteq \mathcal{O}(\sqrt{n^9})$

[**True** — False] $\mathcal{O}(n(n-1)(n-2)) \subseteq \mathcal{O}(\frac{1}{1000}n^3 - 200n^2 - n + 2)$

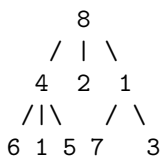
Question 2. [6 MARKS]

Here is a class to represent a node in a tree:

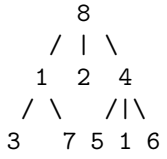
```
class TreeNode(object):

    '''A node in a tree.'''

    def __init__(self, v):
        '''A new Node with value v and an (initially empty) ordered list of children
        (which are Nodes themselves).'''
        self.value = v
        self.children = []
```

The *mirror image* of a tree is defined to be the tree where the ordering of the children at every node is reversed.

For example, if this is the initial tree:



Then the mirror image of it would be:



Fill in the method `mirror` which, given the root node of a tree as input, transforms it into its mirror image. Note that you should not construct a second tree.

```
def mirror(root):
    '''Transform root into its mirror image.'''
```

```
    # First reverse the children of this node.
    root.children.reverse()
    # Now recursively reverse all of the children.
    for child in root.children:
        mirror(child)
```

Continue your answer to the mirror tree question here.

Question 3. [9 MARKS]

A *palindrome* is a word spelt the same way fowards or backwards. Some examples are mom, dad, rotator, sees, and aibohphobia.

Part (a) [6 MARKS]

Write a recursive method to check if a given string is a palindrome; that is whether the string is equal to the string reversed.

```
def palindrome(s):
    '''Returns True if and only if s is a palindrome.'''
```

```
    if len(s) < 2:
       return True
    if s[0] == s[-1]:
        return palindrome(s[1:-1])
```

Part (b) [3 MARKS]

Give six test cases you would use to test `palindrome`, specify what `palindrome` should return, and briefly state why that case is "interesting". Each case should test something different.

- `s = "a"`: Testing the base case (True).
- `s = "QabbaQ"`: Even length palindrome success (True).
- `s = "qQ"`: Make sure that it is case insensitive (False).
- `s = ""`: Empty string (True).
- `s = "QabZbaQ"`: Odd length palindrome success (True).
- `s = "aaaaaaaaabaaaaaaaa"`: Almost a palindrome (False).

Question 4. [10 MARKS]

A common technique for shuffling a deck of playing cards is to first split the deck into two (not necessarily equal sized) piles, and interleaving them; alternating between taking from the first pile and second pile, and placing them on the bottom of our new shuffled pile. This is called a *riffle shuffle*.

For our purposes, each card will be represented by an integer in the range 1 to 52, and we may not be playing with a full deck.

For example, if we start with the following two piles (representing cards as integers in the range 1 to 52):

```
Pile 1: (top) 1 2 3 4 5 6 7 (bottom)
Pile 2: (top) 8 9 10 11 (bottom)
```

Then to shuffle these two piles, we construct a new pile by first taking the top card from Pile 1 and put it on the bottom of the shuffled pile, then from Pile 2, then from Pile 1 again, then from Pile 2, etc., always moving a card from one of the two piles to the bottom of the shuffled pile. Repeat this until all the cards are used up. If one of the piles runs out of cards, we continue moving cards from the other pile.

Performing this algorithm on the two piles described above, we get:

```
Shuffled Pile: (top) 1 8 2 9 3 10 4 11 5 6 7 (bottom)
```

The two piles will be represented as singly linked lists. You will be given references to the tops of both the piles and will construct (and return) a **new** linked list representing the shuffled pile. **The linked lists representing the two piles should remain unaltered.** Complete the method `shuffle` as described above.

```
class Node(object):

    '''A node in a linked list.'''

    def __init__(self, card):
        '''Create a new node containing card.'''
        self.card = card
        self.next = None

def shuffle(pile1, pile2):
    '''Shuffle the two piles, and return the new shuffled pile.
    pile1 and pile2 should not be altered.'''
```

Continue your answer to the card shuffle question here.

```
    head = None
    tail = None
    # Move the first card to the top of the new pile.
    if pile1:
        head = Node(pile1.card)
        tail = head
        pile1 = pile1.next
    elif pile2:
        head = Node(pile2.card)
        tail = head
        pile2 = pile2.next
    else:
        # Both the piles are empty.
        return None
    first_pile = False # Should we take from the first or second pile
    while pile1 or pile2:
        if first_pile and pile1:
            # Take from pile 1
            tail.next = Node(pile1.card)
            tail = tail.next
            pile1 = pile1.next
        elif (not first_pile) and pile2:
            # Take from pile 2
            tail.next = Node(pile2.card)
            tail = tail.next
            pile2 = pile2.next
        first_pile = not first_pile
    return head
```

Question 5. [8 MARKS]

In lecture, we implemented the Queue ADT using a Python list, and in Assignment 1, you implemented it using a linked list. In this question, you will to partially implement the Queue ADT using two stacks.

Consider the following definition of a queue:

```
from stack import Stack

class Queue(object):

    '''An implementation of the Queue ADT using two stacks.'''

    def __init__(self):
        '''Initialze two empty stacks.'''
        stack1 = Stack()
        stack2 = Stack()
        # No more attributes may be declared.

    def enqueue(self, x):
        '''Add x to the end of the queue.'''
        pass

    def dequeue(self):
        '''Remove (and return) the element from the front of
        the queue.'''
        pass

    def front(self):
        '''Return the element at the front of the queue.'''
        pass

    def empty(self):
        '''Return True if and only if the queue is empty.'''
        pass

    def size(self):
        '''Return how many elements are in the queue.'''
        pass
```

You are not allowed to use any new variables. Doing so will result in a mark of zero. You are allowed to use any stack methods as you see fit (push, pop, size, peek, and empty). You are not required to do any error checking or raise any exceptions. The efficiency of your solution is important. You may assume that all stack operations have runtime $\mathcal{O}(1)$.

Part (a) [6 MARKS]

Fill in the code for `enqueue`, `dequeue`, `front` and `size`.

```
def enqueue(self, x):
    '''Add x to the end of the queue.'''
```

```
    while not self.stack1.empty():
        self.stack2.push(self.stack1.pop())
    self.stack1.push(x)
    while not self.stack2.empty():
        self.stack1.push(self.stack2.pop())
```

```
def dequeue(self):
    '''Remove (and return) the element from the front of
    the queue.'''
```

```
    return self.stack1.pop()
```

```
def front(self):
    '''Return the element at the front of the queue.'''
```

```
    return self.stack1.peek()
```

```
def size(self):
    '''Return how many elements are in the queue.'''
```

```
    return self.stack1.size()
```

Part (b) [2 MARKS]

If your queue contains n elements, what are the runtimes of `enqueue`, `dequeue`, `front`, and `size`? Use $\mathcal{O}$ notation and justify your answer.

`size`, `front`, and `dequeue` are clearly $\mathcal{O}(1)$ (single stack operations).

`enqueue` moves every element to `stack2` and back, which has runtime $\mathcal{O}(n)$.

Note that there is an alternative solution that does all the work in `dequeue` instead of `enqueue`. This would result in linear `peek`.

Question 6. [12 MARKS]

Part (a) [2 MARKS]

Suppose that the values [0,1,2,3,4,5,6,7,8,9] are inserted into a min-heap in the following order: [5,4,7,1,6,0,2,8,3,9]. Draw the list representation of this heap.

```
+---+---+---+---+---+---+---+---+---+---+
| 0 | 3 | 1 | 4 | 6 | 7 | 2 | 8 | 5 | 9 |
+---+---+---+---+---+---+---+---+---+---+
```

Part (b) [1 MARK]

What would the heap look like after a single call of `extract_min`?

```
+---+---+---+---+---+---+---+---+---+---+
| 1 | 3 | 2 | 4 | 6 | 7 | 9 | 8 | 5 |   |
+---+---+---+---+---+---+---+---+---+---+
```

Part (c) [8 MARKS]

We would like to define a new operation we can perform on our min-heap called `update_key`. This new operation will take the list representation of the heap, the index of a node, and the new value that should be stored on that node. Fill in the code for `update_key` below.

You may find it useful to recall that in the list representation of a binary tree, the node at index `i` has its parent at `(i - 1) / 2`, left child at `2 * i + 1` and right child at `2 * i + 2`. A solution which does not have runtime $\mathcal{O}(\log n)$, where n is the number of values initially in the heap will receive a maximum of 3 marks. You are free to write any helper methods you see fit, but your answer must be self-contained; you may not call methods written in lecture.

```
def update_key(heap, i, new_value):
    '''Takes the list representation of a min-heap (heap) and updates
    the value on node i (heap[i]) to be new_value, and then fixes heap.'''
```

Continue your answer to the update key question here.

```
heap[i] = new_value
while i > 0 and heap[i] < heap[(i-1)/2]:
    heap[i], heap[(i-1)/2], i = heap[(i-1)/2], heap[i], (i-1)/2
while True:
    left = 2 * i + 1
    right = 2 * i + 2
    smallest = i
    if left < len(heap) and heap[smallest] > heap[left]:
        smallest = left
    if right < len(heap) and heap[smallest] > heap[right]:
        smallest = right
    if smallest == i:
        break
    else:
        heap[i], heap[smallest], i = heap[smallest], heap[i], smallest
```

Part (d) [1 MARK]

State the runtime of your `update_key` method (using $\mathcal{O}$ notation) and justify your answer. A runtime with no justification (or an incorrect one) will receive zero marks.

The first loop goes from the node to the root in the worst case; the second goes from the node to a leaf. That fact that these two loops are mutually exclusive is irrelevant. In the worst case, node `i` travels the height of the tree, which for an almost complete binary tree is $\mathcal{O}(\log n)$.

Question 7. [10 MARKS]

You have a pocket full of coins and want to know if it is possible to pay for an item with exact change. Since you are a computer scientist, you have decided to write a Python program to determine this.

You will write a recursive method which takes 3 parameters:

- `value`: A list where `value[i]` indicates how much a coin of type `i` is worth.
- `amount`: A list where `amount[i]` indicates how many of coin type `i` you have in your pocket.
- `target`: The amount you are trying to make.

It is guaranteed that `len(value) == len(amount)`, and every element of the two lists and `target` will be non-negative. Fill in the method `change` below.

```
def change(value, amount, target):
    '''Returns True if and only if it is possible to reach target using
    only the coins available.'''
```

```
    if target == 0:
        return True
    if value == []:
        return False
    for i in range(amount[0] + 1):
        if change(value[1:], amount[1:], target - i * value[0]):
            return True
    return False
```

Short Python function/method descriptions:

```
__builtins__:
  len(x) -> integer
    Return the length of the list, tuple, dict, or string x.
  max(L) -> value
    Return the largest value in L.
  min(L) -> value
    Return the smallest value in L.
  range([start], stop, [step]) -> list of integers
    Return a list containing the integers starting with start and ending with
    stop - 1 with step specifying the amount to increment (or decrement).
    If start is not specified, the list starts at 0.  If step is not specified,
    the values are incremented by 1.
int:
  int(x) -> integer
    Convert a string or number to an integer, if possible.  A floating point
    argument will be truncated towards zero.
list:
  L.append(x)
    Append x to the end of the list L.
  L.index(value) -> integer
    Returns the lowest index of value in L.
  L.insert(index, x)
    Insert x at position index.
  L.pop([index])
    Remove the item at the given position in the list, and return it.  If no
    index is given, removes the last element.
  L.remove(value)
    Removes the first occurrence of value from L.
  L.reverse()
    Reverse the order of the list.
str:
  str(x) -> string
    Convert an object into its string representation, if possible.
  S.find(sub[,i]) -> integer
    Return the lowest index in S (starting at S[i], if i is given) where the
    string sub is found or -1 if sub does not occur in S.
  S.index(sub) -> integer
    Like find but raises an exception if sub does not occur in S.
  S.isdigit() -> boolean
    Return True if all characters in S are digits and False otherwise.
  S.replace(old, new) -> string
    Return a copy of string S with all occurrences of the string old replaced
    with the string new.
  S.rstrip([chars]) -> string
    Return a copy of the string S with trailing whitespace removed.
    If chars is given and not None, remove characters in chars instead.
  S.split([sep]) -> list of strings
    Return a list of the words in S, using string sep as the separator and
    any whitespace string if sep is not specified.
  S.strip() -> string
    Return a copy of S with leading and trailing whitespace removed.
```