Question 1. [8 MARKS]

Part (a) [4 MARKS]

Recall that we defined the height of a tree in such a way that a tree consisting of just the root has a height of 1. Suppose we have a binary tree with 20 nodes.

(i) The greatest height this tree could have is: _____. Use a diagram to justify your answer:

sample solution

The greatest height is 20. If each interior node has only a right child, then the 20 nodes form a path from root to leaf.

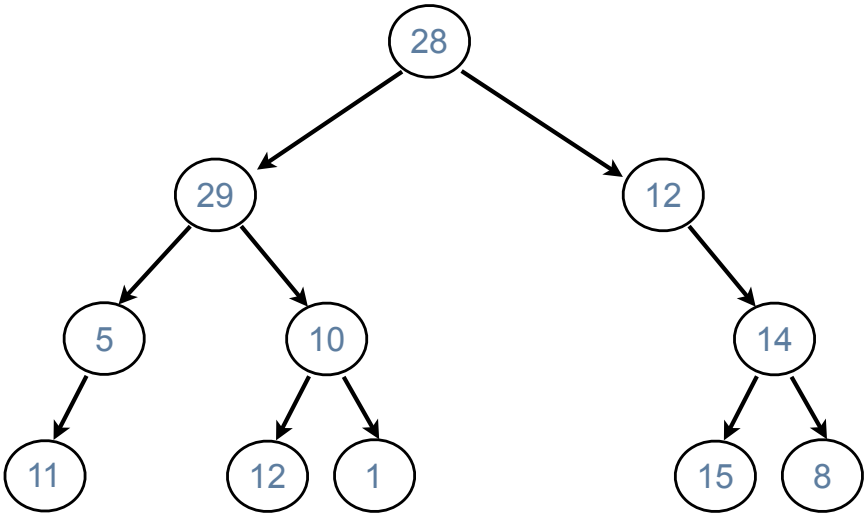
(ii) The least height this tree could have is: _____. Use a diagram to justify your answer:

The least height is 5. We obtain this by filling each level as much as possible:

- one node (root) at level 0
- two nodes at level 1
- four nodes at level 2
- eight nodes at level 3
- four nodes at level 4 (we could fit sixteen, but don't need to)

Part (b) [4 MARKS]

Here is a tree. Notice that it is *not* a binary search tree.



1

If we traverse the tree using pre-order traversal, in what order would the nodes be visited? Write the values below in the correct order.

sample solution:

The node would be visited in this order (by their values):

`28, 29, 5, 11, 10, 12, 1, 12, 14, 15, 8`

Question 2. [10 MARKS]

Read over the declaration of class **BTNode** and the docstring for function **list_even_between**:

```
class BTNode:
    '''Binary Tree node.'''

    def __init__(self, data, left=None, right=None):
        ''' (BTNode, object, BTNode, BTNode) -> NoneType

        Create BTNode (self) with data and children left and right.
        '''
        self.data, self.left, self.right = data, left, right


def list_even_between(t, start, stop):
    ''' (BTNode, int, int) -> list

    Return a list of data in the tree rooted at t that
    (a) are even and (b) are between start and stop, inclusive.

    Assume that t is the root of a (possibly empty) binary
    search tree with integer elements. That is, assume:
        -- every non-empty node has integer data
        -- all data in any left sub-tree is less than the
           data in the root node
        -- all data in any right sub-tree is more than the
           data in the root node.

    >>> t = None
    >>> list_even_between(t, 5, 9)
    []
    >>> t1 = BTNode(4, BTNode(2), BTNode(5))
    >>> t2 = BTNode(9, BTNode(8), BTNode(10))
    >>> t3 = BTNode(7, t1, t2)
    >>> L = list_even_between(t3, 3, 8)
    >>> L.sort()
    >>> L
    [4, 8]
    '''
```

On the next page, implement (write the body for) **list_even_between**. For maximum credit, your implementation should use the binary search tree property to avoid visiting unnecessary nodes.

sample solution

```
    if t is None:
        return []
    else:
```

```
    left_list = (list_even_between(t.left, start, stop)
                 if t.data > start
                 else [])
    right_list = (list_even_between(t.right, start, stop)
                  if t.data < stop
                  else [])
    mid_list = ([t.data]
                if (start <= t.data <= stop) and (t.data % 2 == 0)
                else [])
    return left_list + mid_list + right_list
```

Question 3. [9 MARKS]

Read over the initializers below for classes **LLNode** and **LinkedList**, as well as the docstring for function **absorb_value**. You may assume that appropriate **LLNode._str_** and **LinkedList.append** methods have been defined.

```
class LLNode:
    '''Node to be used in linked list

    nxt: LLNode -- next node
                   None iff we're at end of list
    value: object --- data for current node
    '''

    def __init__(self, value, nxt=None):
        ''' (LLNode, object, LLNode) -> NoneType

        Create LLNode (self) with data value and
        successor nxt.
        '''
        self.value, self.nxt = value, nxt
```

```
class LinkedList:
    '''Collection of LLNodes organized into a
    linked list.

    front: LLNode -- front of list
    back:  LLNode -- back of list
    size:  int    -- size of list
    '''

    def __init__(self):
        ''' (LinkedList) -> NoneType

        Create an empty linked list.
        '''
        self.front, self.back = None, None
        self.size = 0
```

```
def absorb_value(lnk, value):
    ''' (LinkedList, int) -> NoneType

    Remove from lnk the node with the first occurrence of value, and add that
    value to the node that follows. If there is no node in lnk containing
    value, or if it occurs only in the back node, leave lnk unchanged.

    Assume that lnk contains only integer values.

    >>> lnk = LinkedList()
    >>> lnk.append(6)
    >>> lnk.append(7)
    >>> lnk.append(8)
    >>> lnk.append(7)
    >>> print(lnk.front)
    6 -> 7 -> 8 -> 7 ->|
    >>> absorb_value(lnk, 7)
    >>> print(lnk.front)
    6 -> 15 -> 7 ->|
    >>> absorb_value(lnk, 6)
    >>> print(lnk.front)
    21 -> 7 ->|
    '''
```

sample solution

```
    if lnk.front:
        prev_node, cur_node = None, lnk.front
        while cur_node and not cur_node.value == value:
            prev_node = cur_node
            cur_node = cur_node.nxt
        if cur_node and cur_node.nxt:
            cur_node.nxt.value += value
            if prev_node:
                prev_node.nxt = cur_node.nxt
            else:
                lnk.front = cur_node.nxt
            lnk.size -= 1
        else: # didn't find value
            pass
    else: # empty list
        pass
```

Now implement (write the body) of **absorb_value** You should probably **draw pictures!**