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 tree of height 3 with a branching factor (arity) of 3.

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

sample solution

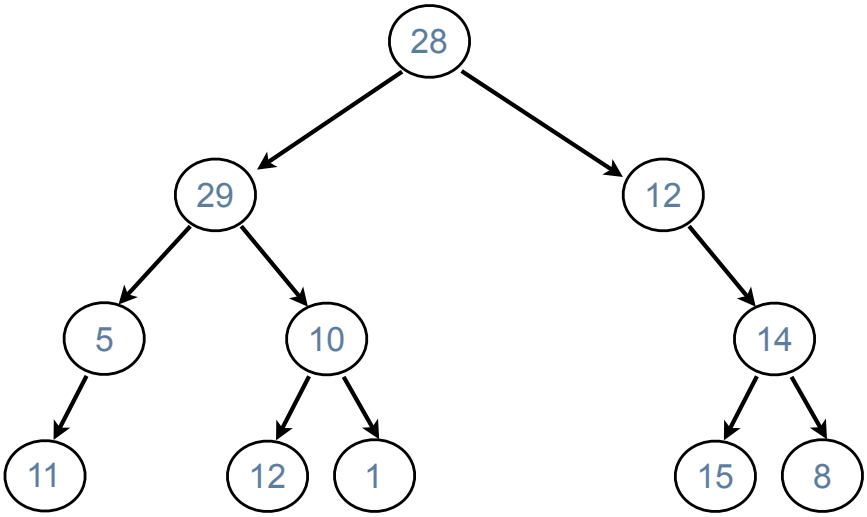
The greatest number of nodes is 13: 1 at level zero, 3 at level one, and 9 at level three.

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

The least number of nodes is 3: the root, with a right child that also has a right child.

Part (b) [4 MARKS]

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



1

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

sample solution:

Here are the values in order:

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

Question 2. [10 MARKS]

Read over the declaration of class **BTNode** and the docstring for function **list_leaves_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_leaves_between(t, start, stop):
    ''' (BTNode, int, int) -> list

    Return a list of data in leaves of the tree rooted at t that
    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_leaves_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_leaves_between(t3, 3, 8)
    >>> L.sort()
    >>> L
    [5, 8]
    '''
```

On the next page, implement (write the body for) **list_leaves_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_leaves_between(t.left, start, stop)
                 if t.data > start
                 else [])
    right_list = (list_leaves_between(t.right, start, stop)
                  if t.data < stop
                  else [])
    mid_list = ([t.data]
                if (start <= t.data <= stop) and not t.left and not t.right
                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 **split_back**. 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 split_back(lnk):
    ''' (LinkedList) -> NoneType

    Insert a new node before lnk.back with
    value (lnk.back.value // 2), and replace lnk.back.value
    with (lnk.back.value - (lnk.back.value // 2)).  If there
    is no lnk.back node, leave lnk unchanged.

    >>> lnk = LinkedList()
    >>> lnk.append(7)
    >>> split_back(lnk)
    >>> print(lnk.front)
    3 -> 4 ->|
    >>> split_back(lnk)
    >>> print(lnk.front)
    3 -> 2 -> 2 ->|
    '''
```

sample solution

```
    if lnk.front:
        prev_node, cur_node = None, lnk.front
        while cur_node.nxt:
```

```
            prev_node = cur_node
            cur_node = cur_node.nxt
        new_node = LLNode(cur_node.value // 2, cur_node)
        cur_node.value = cur_node.value - new_node.value
        if prev_node:
            prev_node.nxt = new_node
        else:
            lnk.front = new_node
        lnk.size += 1
    else:
        pass
```

Now implement (write the body) of **split_back** You should probably **draw pictures!** (not required).