Question 1. [8 MARKS]

Read the docstring below for method **remove_first_double**. You may assume that classes **LinkedListNode** and **LinkedList** from the API have been imported. Implement **remove_first_double**. **Note**: The only **LinkedList** and **LinkedListNode** methods provided are those in the API.

```
def remove_first_double(self):
    """
    Remove second of two adjacent nodes with duplicate values.
    If there is no such node, leave self as is.  No need
    to deal with subsequent adjacent duplicate values.

    @param LinkedList self: this linked list
    @rtype: None

    >>> list_ = LinkedList()
    >>> list_.append(3)
    >>> list_.append(2)
    >>> list_.append(2)
    >>> list_.append(3)
    >>> list_.append(3)
    >>> print(list_.front)
    3 -> 2 -> 2 -> 3 -> 3 ->|
    >>> list_.remove_first_double()
    >>> print(list_.front)
    3 -> 2 -> 3 -> 3 ->|
    """

    if self.size < 2:
        # no room for doubles
        return None
    else:
        current_node = self.front
        while (current_node.next_ and
               current_node.value != current_node.next_.value):
            current_node = current_node.next_
        if current_node.next_ is not None:
            current_node.next_ = current_node.next_.next_
            if current_node.next_ is None:
                self.back = current_node
            self.size -= 1
```

Question 2. [8 MARKS]

Read the docstring for function **contains_satisfier** below, and then implement it.

```
def contains_satisfier(list_, predicate):
    """
    Return whether possibly-nested list_ contains a non-list element
    that satisfies (returns True for) predicate.

    @param list list_: list to check for predicate satisfiers
    @param (object)->bool predicate: boolean function
    @rtype: bool

    >>> list_ = [5, [6, [7, 8]], 3]
    >>> def p(n): return n > 7
    >>> contains_satisfier(list_, p)
    True
    >>> def p(n): return n > 10
    >>> contains_satisfier(list_, p)
    False
    """

    return any([contains_satisfier(c, predicate)
                if isinstance(c, list) else predicate(c)
                for c in list_])
```

Question 3. [8 marks]

Read the docstring below for function **count_odd_above**, as well as the API for class **Tree**. You may assume that class **Tree** has been imported. Implement function **count_odd_above**. **Hint**: The depth of a node is 1 less than the depth of its children.

```
def count_odd_above(t, n):
    """
    Return the number of nodes with depth less than n that have odd values.

    Assume t's nodes have integer values.

    @param Tree t: tree to list values from
    @param int n: depth above which to list values
    @rtype: int

    >>> t1 = Tree(4)
    >>> t2 = Tree(3)
    >>> t3 = Tree(5, [t1, t2])
    >>> count_odd_above(t3, 1)
    1
    """
    if n <= 0:
        return 0
    else:
        return ((1 if t.value % 2 == 1 else 0) +
                sum([count_odd_above(c, n-1) for c in t.children]))
```

Question 4. [6 MARKS]

Draw a diagram of a binary search tree of minimum height containing the following integer values:

7, 1, 9, 13, 5, 3, 15, 11

One sample solution (without circles and edges)

```
       7

    3        11

  1   5    9    13
                    15
```