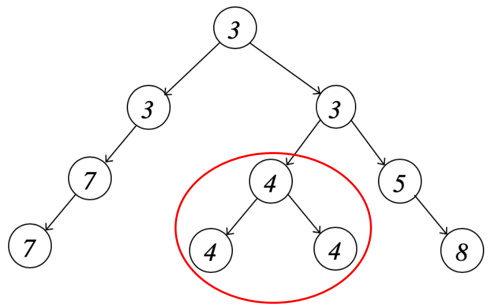
Consider the following definition of a *mono-data subtree*:
Let *T* be a binary tree containing integers as data in its nodes, and let *T'* be a subtree of *T*. We say that *T'* is a mono-data subtree, if all the nodes of *T'* have the same data.

For example, in the following tree, the subtree that is circled in red is a mono-data subtree, as all its nodes have the same data (the common data is 4).



Note: Each leaf is a mono-data subtree (these are the smallest mono-data subtrees possible).

In this question, we will implement the following function:

```
def count_mono_data_subtrees(bin_tree)
```

The function is given `bin_tree`, a non-empty `LinkedBinaryTree` object, it will return the number of mono-data subtrees in `bin_tree`.

For example, if called with the tree above, it should return 6, as these are the six mono-data subtrees:

Complete the implementation (given in the next page) for the function `count_mono_data_subtrees`.
In the implementation you should define a nested **recursive** helper function:

```
def count_mono_data_subtrees_helper(root)
```

This function is given `root`, a reference to a `LinkedBinaryTree.Node`, that indicates the root of the subtree that this function operates on.

```python
def count_mono_data_subtrees(bin_tree):
    def count_mono_data_subtrees_helper(root):
        …
        …
        …

    ___________ = count_mono_data_subtrees_helper(bin_tree.root)
    return ________
```

Implementation requirements:

1. Your implementation has to run in **linear time**. That is, if there are n nodes in the tree, your function should run in $\theta(n)$ worst-case.
2. Your implementation for the helper function must be **recursive**.
3. You are **not allowed** to:
    - Define any other helper function.
    - Add parameters to the function's header lines.
    - Set default values to any parameter.
    - Use global variables.

Hint:
To meet the runtime requirement, you may want `count_mono_data_subtrees_helper` to return more than one value (multiple values could be collected as a tuple).

We say that a sequence of numbers is a *palindrome* if it is read the same backward or forward.
For example, the sequence: *6, 15, 6, 3, 47, 3, 6, 15, 6* is a palindrome.

Implement the following function:

```
def construct_a_longest_palindrome(numbers_bank)
```

When given `numbers_bank`, a non-empty list of integers, it will create and return a list containing a longest possible palindrome made only with numbers from `numbers_bank`.

Notes:

1. The longest palindrome might NOT contain all of the numbers in the sequence.
2. If no multi-number palindromes can be constructed, the function may return just one number (as a single number, alone, is a palindrome).
3. If there is more than one possible longest palindrome, your function can return any one of them.

For example, if `numbers_bank=[3, 47, 6, 6, 5, 6, 15, 3, 22, 1, 6, 15]`,
Then the call `construct_a_longest_palindrome(numbers_bank)` could return:
`[6, 15, 6, 3, 47, 3, 6, 15, 6]` (Which is a palindrome of length 9, and there is no palindrome made only with numbers from `numbers_bank` that is longer than 9).

Implementation requirements:

1. **You may use one `ArrayQueue`, one `ArrayStack`, and one `ChaniningHashTableMap`.**
2. Your function has to run in **expected (average) linear time**. That is, if `numbers_bank` is a list with n numbers, your function should **run in $\Theta(n)$ average case**.
3. Besides the queue, stack, hash table, and the list that is created and returned, you may use only **constant additional space**. That is, besides the queue, stack, hash table, and the returned list, you may use variables to store an integer, a double, etc. However, you may **not** use an additional data structure (such as another list, stack, queue, etc.) to store non-constant number of elements.

Recall the *Minimum-Priority-Queue ADT* we introduced in class. A minimum priority queue is a collection of (priority, value) items, that come out in an increasing order of priorities.

A *Minimum-Priority-Queue* supports the following operations:

- ***p = PriorityQueue()***: Creates an empty priority queue.
- ***len(p)***: Returns the number of items in ***p***.
- ***p.is_empty()***: Returns *True* if ***p*** is empty, or *False* otherwise.
- ***p.insert(pri, val)***: Inserts an item with priority ***pri*** and value ***val*** to ***p***.
- ***p.min()***: Returns the *Item (pri, val)* with the lowest priority in ***p***, or raises an *Exception*, if ***p*** is empty.
- ***p.delete_min()*** : Removes and returns the *Item (pri, val)* with the lowest priority in ***p***, or raises an *Exception*, if ***p*** is empty.

Complete the definition below of the `LinkedMinHeap` class, implementing the *Minimum-Priority-Queue ADT*. In this implementation, you should represent the heap using node objects and references to form a tree structure (a "linked representation" of the tree). That is, you should construct Node objects with references to their "children" and "parent".

<u>**Note**</u>: In class (when we implemented the `ArrayMinHeap` class) we represented the heap using an "array representation" of the tree.

```
class LinkedMinHeap:
   class Node:
       def __init__(self, item):
           self.item = item
           self.parent = None
           self.left = None
           self.right = None

   class Item:
       def __init__(self, priority, value=None):
           self.priority = priority
           self.value = value

       def __lt__(self, other):
           return self.priority < other.priority

   def __init__(self):
       self.root = None
       self.size = 0

   def __len__(self):
       …

   def is_empty(self):
       …

   def min(self):
       …

   def insert(self, priority, value=None):
       …

   def delete_min(self):
       …
```

Notes:

1. In the `LinkedMinHeap` class that you would need to complete (given in the previous page), we already implemented two nested classes:
   - `Node` – should be used for each node object of the linked binary tree.
   - `Item` – should be used to store the (priority, value) item of the Priority Queue.
2. We also implemented the `__init__` method of the `LinkedMinHeap` class.
   Each `LinkedMinHeap` object would maintain two data members:
   - `self.root` – A reference to the heap's root node. Initially set to **`None`** (indicating an empty tree)
   - `self.size` – Indicating the number of nodes in the heap. Initially set to 0.

Implementation requirements:

1. You are **not** allowed to add data members to the `LinkedMinHeap` object.
   That is, you **can't** edit the `__init__` method, that initializes `root` and `size` as the **only** data member.
2. Runtime requirements:
   - Each one of the `insert` and `delete_min` operations should run in $\theta(\log(n))$ **worst case** (where n is the number of elements in the priority queue).
   - Each one of the `len`, `is_empty`, and `min` operations should run in $\theta(1)$ worst case.
3. You may define additional helper methods.