1. Write the method `public boolean hasDuplicate(int [] arr)`. The method returns true if `arr` contains duplicates. The numbers in the array must be between 1 and the size of the array. No nested loop is allowed to enhance efficiency.

2. `sortedList` is of type `ArrayList<Comparable>`. It contains some elements and they are sorted in order. Write the fragment of codes such that `uniqueList` (of the same type as `sortedList`) contains only the unique elements in `sortedList`. You can assume the appropriate method has been defined for you to compare two objects. `uniqueList` should reference the objects referenced by `sortedList`. For objects that are the same, you can reference either one of them.
   For example, if `sortedList` = `[4, 4, 4, 5, 6, 7, 7, 8]`, `uniqueList` is `[4, 5, 6, 7, 8]`.

3. A Priority Queue (PQ) is a list of items where each item is associated with a priority. The list is sorted according to the descending order of priority. That is, the item with the highest priority is the head of the queue, while the last item is of the lowest priority. Two items can have the same priority. To enhance efficiency in removing the head of queue, when a new item is inserted in a PQ, it should be inserted according to the priority order. Implement the abstract data type `MyPQ` that contains the following methods. You can assume the items are all `Comparables` according to the order of their priorities. Note that you should define instance variables if necessary, but the `toString` method is NOT necessary.

   `public boolean isEmpty()`
   This method returns `true` if the PQ is currently empty

   `public void insert(Comparable obj)`
   This method inserts `obj` to the PQ. Note that it should be inserted in the right position in the PQ.
   For example, suppose the objects in the PQ are `Integers`.

```
insert(4)
insert(3)
insert(8)
```

   will produce PQ [8, 4, 3].

   `public Comparable deleteMax()`
   This method returns the object with the highest priority. The object is removed from the PQ afterwards.

   `public int size()`
   This method returns the size of the PQ

`public void remove(Comparable item)`
This method removes the item from the PQ. Note that ALL items in the list that share the same priority with the parameter should be removed. `item` may or may not be in the PQ.

4. Class `Polynomial` represents polynomials. Instance variable `int[] coeff` keeps the coefficients of the polynomial.

```
class Polynomial {
    private int[] coeff;
}
```

Define necessary methods in class `Polynomial` so that class `TestPoly` produces the given output. Note that you cannot declare other instance variables in `Polynomial`. Doing so will not receive any credit.

```
public class TestPoly {
  public static void main(String[] args) {

    int[] f1coeff = {6, 5, 3, 0, 0, 1};
    int[] f2coeff = {5, -1, 0, 4, 2};

    Polynomial f1 = new Polynomial(f1coeff);
    Polynomial f2 = new Polynomial(f2coeff);

    System.out.println("f1 = " + f1);
    System.out.println("f2 = " + f2);
    System.out.println("f1 + f2 = " + f1.add(f2));
        // both f1 and f2 are not changed by method add

    f1.differentiate();
        // f1 is changed to its derivative
    System.out.println("f1 = " + f1);
  }
}
```

Output:

```
f1 = 6x^5+5x^4+3x^3+1x^0
f2 = 5x^4-1x^3+4x^1+2x^0
f1 + f2 = 6x^5+10x^4+2x^3+4x^1+3x^0
f1 = 30x^4+20x^3+9x^2
```

5. A *ring* is circular linked list that the last node points to the first node as shown in the figure.

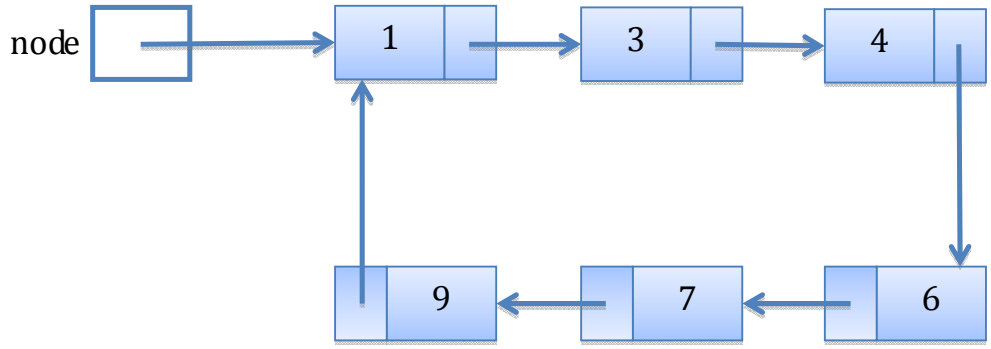

Figure 1: An example of the `Ring` structure

The following codes define classes `MyRing` and `RingNode` for the ring structure. You cannot define other instance variables.

```
public class MyRing {
  private RingNode node;

  public MyRing() {
    node = null;
  }


public class RingNode {
  private int info;
  private RingNode neighbor;

  public RingNode(int info) {
    this.info = info;
  }

  public RingNode getNeigh() {
    return neighbor;
  }

  public void setNeigh(RingNode node) {
    neighbor = node;
  }

  public int getInfo() {
    return info;
  }
}
```

a) Write a constructor for `MyRing` such that the following statements can be executed. You can assume the integer (`int`) parameters are arranged in ascending order. The nodes in the created ring are arranged in ascending order as well, with reference `node` points to the smallest node.

```
MyRing ring1 = new MyRing(1, 5, 9, 11);
MyRing ring2 = new MyRing(1, 3, 4, 6, 7, 9);
       // ring2 is the ring shown in Figure 1
```

b) Define method `public int size()` in class `MyRing` that returns the number of nodes in the ring.

c) Write method `public MyRing mergeWith(MyRing r)` in class `MyRing` that returns a new `MyRing` object that contains the numbers in `r` and the object calling this method. The nodes in the ring should be arranged in ascending order. Both `r` and the object calling the method should not be changed. The runtime complexity of your method must be linear to the total size of the rings.
For example, the numbers in `ring3` where
`MyRing ring3 = ring1.mergeWith(ring2)` are:
1, 1, 3, 4, 5, 6, 7, 9, 9, 11

d) Write method `public void removeDuplicates()` in class `MyRing` that removes duplicated numbers in the object calling the method.
For example, the numbers in `ring3` after `ring3.removeDuplicates()` are:
1, 3, 4, 5, 6, 7, 9, 11
You can assume the numbers in the ring are sorted. A number in the ring may be duplicated for more than two times.

6. The following defines a class representing nodes in a binary tree.

```
public class BinTreeNode {
    protected Comparable obj;
    protected BinTreeNode leftChild, rightChild, parent;

    public BinTreeNode(Comparable item) {
        obj = item;
        leftChild = rightChild = parent = null;
    }

  public BinTreeNode left() {
        return leftChild;
    }

    public BinTreeNode right() {
        return rightChild;
    }

    public BinTreeNode father() {
        return parent;
    }

    public Comparable getObj() {
        return obj;
    }

}
```

Class `ColorTreeNode` is extended from `BinTreeNode` for representing tree nodes that have color. `Color` is an enumerated type. `ColorBinTree` is the class for the trees where nodes have color.

```
public class ColorTreeNode extends BinTreeNode {
   private Color color;
}

public class ColorBinTree {
    private ColorTreeNode root;

    public ColorBinTree() {
        root = null;
    }
}
```

You may define new methods (but not new instance/class variables) in `ColorTreeNode`, `BinTreeNode` or `ColorBinTree` for (a) – (e).

a) Provide the definition of constructor
```
public ColorTreeNode(Comparable item, Color c).
```

b) Write method `public boolean isLeaf()` in `BinTreeNode`. The method returns true if the node invoking the method is a leaf node on the tree.

c) Implement method `public int countColor(Color c)` in class `ColorBinTree` that counts how many nodes on a tree with the color specified in the parameter. You may define methods in `ColorTreeNode` or `BinTreeNode.`

d) The mechanism of adding a tree node with color `Color.RED` is as follows:

- If the tree is empty, insert the new node as the root
- If the tree is not empty, start from the root:
  - If the node does not have a left child, insert the node as the left child, DONE
  - If the node does not have a right child, insert the node as the right child, DONE
  - If the node have both children, continue to search on the subtree that has fewer RED nodes

An example is shown in Figure 2 (The shaded nodes are RED nodes). Write method `public void addRed(ColorTreeNode node)` in `ColorBinTree`.

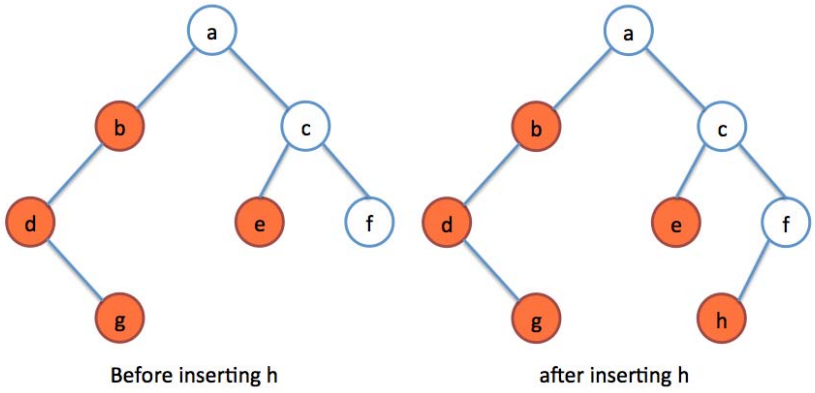

Figure 2: Insert a RED node

e) When a color tree node changes its color from other color to `Color.BLUE`, it will move down the tree until all of its children (if any) are BLUE. It will move down when there is at least one non-BLUE child. When both children are non-BLUE, move down on the left subtree.

Figure 3 shows the final tree when Node `a` turns BLUE. In the figure, shaded nodes are BLUE. As both `b` and `c` are non-BLUE, `a` moves down to the left subtree by swapping with `b`. Then, it has a single child `d` and it is also non-BLUE. `a` further moves down. Now, its only child is `g` and it is BLUE. The procedure stops.

Write method `public void turnBlue()` in `ColorTreeNode`.

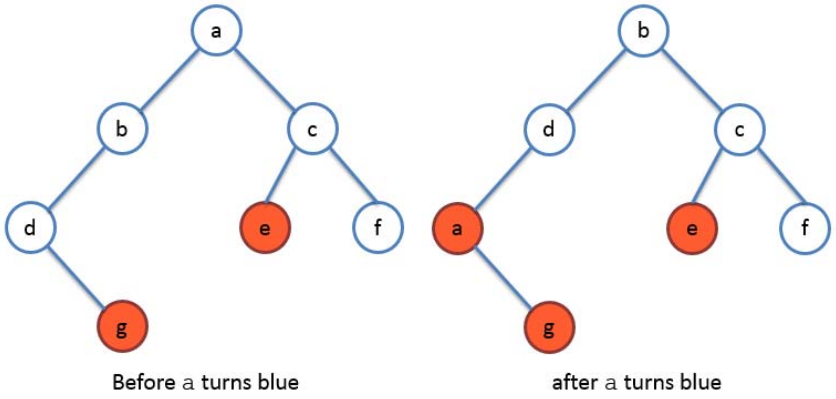

Figure 3: Turn a node to BLUE