```
/**********************************************************
 * EECS2011: Fundamentals of Data Structures, Fall 2016
 * Assignment 3, a3sol.pdf
 * Student Name: Khushal Patel
 * Student CSE account: york18
 * Student ID number: 214037618
 **********************************************************/
```

Note: I am using following classes from textbook for question 2 and 3 to implement a binary tree:

- AbstractBinaryTree.java
- AbstractTree.java
- BinaryTree.java
- LinkedBinaryTree.java
- Position.java
- Tree.java

Question 1: CardShuffle.java

Basic Idea: Split the given LinkedList into two halves from middle. Now we are interested in 4 nodes of this list. First one being the first node. Then the second one being the last node of the first half of the list, then the one that is the first node of the second half is the third and fourth node is one next to the third node.

Now node1's next should point to node3. Node3's next should point to the initial node1's next and node2's next should point to node4.

After this step, Node1 becomes Node2, Node2 becomes Node1's next and Node 4 becomes Node3's next and Node 3 remains unchanged.

Repeat this as long as i < j where i is 0 at the beginning of the loop and j is list.size – 1.

Space and Time Complexity: So the running time of the program is O(n). In terms of space, I am using 3 int variables and 4 Node object which remains constant throughout the method. So, space complexity is O(1).

Output: (1) **{1, 6, 2, 7, 3, 8, 4, 9, 5, 10}** //for input {1,2,3,4,5,6,7,8,9,10}

(2) **{1, 16, 2, 17, 3, 18, 4, 19, 5, 20, 6, 21, 7, 22, 8, 23, 9, 24, 10, 25, 11, 26, 12, 27, 13, 28, 14, 29, 15, 30}** // for input {1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 25, 26, 27, 28, 29, 30}

Question 2: BalanceFactor.java

Basic Idea: Recursively find the height of a Binary Tree. Start with the root position. Base case of the recursive method would be to check if it's an external node. If it is an external node stop. Else, recursively call itself but now for it's left and right subtree and also print the current node element along with it's calculate balance factor.

Time Complexity: Since this recursive method visits every node in the tree once, it's running time would be O(n).

Output: `(1,0)`
`(5,0)`
`(2,1)`
`(0,1)`

Question 3: PrioritySearchTree.java

Basic Idea: Given a Binary Tree, to convert it into Priority Search Tree, first get all the External 2-D points of the tree and store it in an ArrayList. Then sort that ArrayList such that the maximal y-coordinate is the first element of the array and rest of them are in decreasing order of their y-coordinates. Then fill the Internal nodes of the tree recursively using the sorted ArrayList. Also, when you're done adding a point in the node of a tree, remove that element (i.e., removing at index 0)

While adding the nodes check if the current nodes children are external. If they are external and the next element in our array is not either of it's children's value then don't add that element at that node (i.e., leave that node null).

The code for that is:

```
if(t.isInternal(p) && array.size() > 0)

{

      if(t.isExternal(t.right(p))&&
      !(t.right(p).element().equals(array.get(0))||
      t.left(p).element().equals(array.get(0))))

      {

            // don't do anything since the node's value has to either one of
its external nodes

      }

      else

      {

            t.set(p, array.get(0));
```

```
            array.remove(0); //remove(int index) takes O(n) time in the worst
case but takes O(1) time in the best case and removing at index 0 is always
O(1)

                    fillNodes(array, t, t.right(p));

                    fillNodes(array, t, t.left(p));

      }

}
```

Time Complexity: The program run in O(n logn) since sorting the array takes O(n logn). Other methods run in O(n) time.

Output:

```
(5,9) //root
(4,8) //right child of the root
(9,7) //right right
(9,7) // the external point right right right
(7,1) // the external point right right left
null  // right left
(5,9) // right left right
(4,8) // right left left
(-1,6) // left child of the root
(2,4) // left right
(2,4) // the external node left right right
(-1,6 // the external node left right left
(-8,3 // left left
(-7,1) // the external node left left right
(-8,3) // the external node left left left
```