Flatten BT to LL

```
class Solution {
    public void flatten(TreeNode root) {
        if (root == null)
            return;
        Stack<TreeNode> s = new Stack<>();
        s.push(root);
        while (!s.isEmpty()) {
            TreeNode node = s.pop();
            if (node.right != null) {
                s.push(node.right);
            }
            if (node.left != null) {
                s.push(node.left);
            }
            if (!s.isEmpty()) {
                node.right = s.peek();
            }
            node.left = null;
        }
    }
}
```

Flatten 2D Vector

```
class Vector2D {

    private int[][] v;
    private int row;
    private int col;

    public Vector2D(int[][] v) {
        this.v = v;
        row = 0;
        col = 0;
    }

    public int next() {
        skipEmptyRows();
        if (!hasNext()) {
            throw new NoSuchElementException();
        }
        int next = v[row][col++];
        if (col == v[row].length) {
            row++;
            col = 0;
        }
        return next;
    }

    public boolean hasNext() {
        skipEmptyRows();
        return row < v.length - 1 || (row == v.length - 1 && col < v[row].length);
    }

    private void skipEmptyRows() {
        // Skip empty rows
        while (row < v.length && v[row].length == 0) {
            row++;
        }
    }
}
```

Flatten a Multilevel DLL

```
/*
class Node {
    public int val;
    public Node prev;
    public Node next;
    public Node child;

    public Node() {}

    public Node(int _val,Node _prev,Node _next,Node _child) {
        val = _val;
        prev = _prev;
        next = _next;
        child = _child;
    }
};
*/
class Solution {
    public Node flatten(Node head) {
        if (head == null)
            return head;
        Deque<Node> dq = new ArrayDeque<>();
        Node itr = head;
        while (itr != null) {
            if (itr.child != null) {
                if (itr.next != null) {
                    dq.offerLast(itr.next);
                }
                itr.next = itr.child;
                itr.child.prev = itr;
                itr.child = null;
            }
            if (itr.next == null && !dq.isEmpty()) {
                Node node = dq.pollLast();
                itr.next = node;
                node.prev = itr;
            }
            itr = itr.next;
        }
        return head;
    }
}
```

Flatten Nested List Iterator

```
/**
 * // This is the interface that allows for creating nested lists.
 * // You should not implement it, or speculate about its implementation
 * public interface NestedInteger {
 *
 *     // @return true if this NestedInteger holds a single integer, rather than a nested list.
 *     public boolean isInteger();
 *
 *     // @return the single integer that this NestedInteger holds, if it holds a single integer
 *     // Return null if this NestedInteger holds a nested list
 *     public Integer getInteger();
 *
 *     // @return the nested list that this NestedInteger holds, if it holds a nested list
 *     // Return null if this NestedInteger holds a single integer
 *     public List<NestedInteger> getList();
 * }
 */
public class NestedIterator implements Iterator<Integer> {

    Stack<ListIterator<NestedInteger>> stack;
    public NestedIterator(List<NestedInteger> nestedList) {
        stack = new Stack<>();
        stack.push(nestedList.listIterator());
    }

    @Override
    public Integer next() {
        hasNext();
        return stack.peek().next().getInteger();
    }

    @Override
    public boolean hasNext() {
        while (!stack.isEmpty()) {
            if (!stack.peek().hasNext()) {
                stack.pop();
                continue;
            }
            NestedInteger ni = stack.peek().next();
            if (ni.isInteger()) {
                stack.peek().previous();
                return true;
            } else {
                stack.push(ni.getList().listIterator());
            }
        }
        return false;
    }
}
```