1.

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode() {}

\* ListNode(int val) { this.val = val; }

\* ListNode(int val, ListNode next) { this.val = val; this.next = next; }

\* }

\*/

class Solution {

public ListNode reverseList(ListNode head) {

ListNode p = null; //prev

ListNode c = head; //curr

ListNode n = head; //head

while( n != null){

//speial change

n = n.next;

c.next = p;

p = c;

c = n;

}

return p;

}

}

2.

/\*\*

\* Definition for singly-linked list.

\* public class ListNode {

\* int val;

\* ListNode next;

\* ListNode() {}

\* ListNode(int val) { this.val = val; }

\* ListNode(int val, ListNode next) { this.val = val; this.next = next; }

\* }

\*/

class Solution {

public ListNode middleNode(ListNode head) {

ListNode s = head;

ListNode f = head;

while( f != null && f.next != null){

s = s.next;

f = f.next.next;

}

return s;

}

}

4.

class Solution {

public int detectLoop(Node head) {

// Add code here

Node slow = head;

Node fast = head;

while(true){

if( fast == null || fast.next == null){

return 0;

}

slow = slow.next;

fast = fast.next.next;

if( slow == fast){

return 1;

}

}

// return 0;

}

}

5.

class Clone {

Node copyList(Node head) {

// your code here

// Node temp = new Node(head.data,head.next);

if(head==null) return null;

Node temp = head;

//code of linkedList very easy

//looks tough if we read code separately

// make code yourself,instead of reading code

//just concentrate on dry-run only

//not on code

while( temp != null){

//insert

Node nn = new Node( temp.data);

nn.next = temp.next;

temp.next = nn;

temp = temp.next.next;

}

// temp = head;

// while( temp!= null){

// System.out.print( temp.val + " ");

// temp = temp.next;

// }

temp = head;

Node Head2 = head.next;

Node newhead = head.next;

while( temp != null){

Node store = temp.arb;

//Node temp\_ = temp.next;

temp = temp.next.next;

if( store != null)

newhead.arb = store.next;

else

newhead.arb = null;

// newhead.next = newhead.next.next;

if( newhead.next != null)

newhead = newhead.next.next;

else

newhead = null;

}

temp = head;

newhead = temp.next;

while( temp != null){

temp.next = temp.next.next;

if(newhead.next==null){

newhead.next = null;

}else{

newhead.next = newhead.next.next;

}

// if( newhead.next != null)

// else

// newhead = null;

temp = temp.next;

newhead = newhead.next;

}

return Head2;

}

}

6. LRU cache

class LRUCache {

LinkedList<Integer> ll = new LinkedList<>();

HashMap<Integer,Integer> map = new HashMap<>();

int defaultSize = 2;

LRUCache(int capacity) {

defaultSize = capacity;

}

public void put(int key, int val){

if( map.containsKey(key)){

map.put(key, val);

Integer b = 2;

Integer a = new Integer(key);

ll.remove(a);

ll.addFirst(key);

}

else{

if( ll.size() == defaultSize){

int val2 = ll.removeLast();

map.remove(val2);

}

map.put(key, val);

ll.addFirst(key);

}

}

public int get(int key){

if( !map.containsKey(key)){

return -1;

}

else{

Integer a = new Integer(key);

ll.remove(a);

ll.addFirst(key);

return map.get(key);

}

}

}

7.

public class Solution {

public ListNode getIntersectionNode(ListNode headA, ListNode headB) {

ListNode p1 = headA, p2 = headB;

int size1 = 0 , size2 = 0;

while( p1 != null){

size1++;

p1 = p1.next;

}

while( p2 != null){

size2++;

p2 = p2.next;

}

if( size1 > size2){

return f( headA, headB, size1 - size2);

}

else{

return f( headB, headA, size2 - size1);

}

}

static ListNode f( ListNode p1, ListNode p2 , int num){

int i = 0 ;

while( i != num){

p1 = p1.next;

i++;

}

while( p1 != null && p2 != null ){

if( p1==p2){

return p1

;}

p1 = p1.next;

p2 = p2.next;

}

p1 = null;

return p1;

}

}