

Code : 1

```
package Topic_13_Linked_List;

import java.io.*;
import java.util.*;

public class LinkedListAsQueue {

    public static class LLToQueueAdapter {
        LinkedList<Integer> list;

        public LLToQueueAdapter() {
            list = new LinkedList<>();
        }

        int size() {
            return list.size();
        }

        boolean isEmpty() {
            return list.size() == 0;
        }

        void add(int val) {
            list.addLast(val);
        }

        int remove() {
            if (isEmpty()) {
                System.out.println("Queue underflow");
                return -1;
            } else {
                return list.removeFirst();
            }
        }

        int peek() {
            if (isEmpty()) {
                System.out.println("Queue underflow");
                return -1;
            } else {
                return list.getFirst();
            }
        }
    }

    public static void main(String[] args) throws Exception {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        LLToQueueAdapter qu = new LLToQueueAdapter();

        String str = br.readLine();
        while (str.equals("quit") == false) {
            if (str.startsWith("add")) {
                int val = Integer.parseInt(str.split(" ")[1]);
                qu.add(val);
            } else if (str.startsWith("remove")) {
```

```
        int val = qu.remove();
        if (val != -1) {
            System.out.println(val);
        }
    } else if (str.startsWith("peek")) {
        int val = qu.peek();
        if (val != -1) {
            System.out.println(val);
        }
    } else if (str.startsWith("size")) {
        System.out.println(qu.size());
    }
    str = br.readLine();
}
}
```

Code : 2

```
package Topic_13_Linked_List;

import java.io.BufferedReader;
import java.io.InputStreamReader;
import java.util.LinkedList;

public class LinkedListAsStack {

    public static class LLToStackAdapter {
        LinkedList<Integer> list;

        public LLToStackAdapter() {
            list = new LinkedList<>();
        }

        int size() {
            return list.size();
        }

        void push(int val) {
            list.addFirst(val);
        }

        int pop() {
            if (size() == 0) {
                System.out.println("Stack underflow");
                return -1;
            } else {
                int val = list.getFirst();
                list.removeFirst();
                return val;
            }
        }

        int top() {
            if (size() == 0) {
                System.out.println("Stack underflow");
                return -1;
            } else {
                int val = list.getFirst();
                return val;
            }
        }
    }

    public static void main(String[] args) throws Exception {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        LLToStackAdapter st = new LLToStackAdapter();

        String str = br.readLine();
        while (str.equals("quit") == false) {
            if (str.startsWith("push")) {
                int val = Integer.parseInt(str.split(" ")[1]);
                st.push(val);
            } else if (str.startsWith("pop")) {
```

```
        int val = st.pop();
        if (val != -1) {
            System.out.println(val);
        }
    } else if (str.startsWith("top")) {
        int val = st.top();
        if (val != -1) {
            System.out.println(val);
        }
    } else if (str.startsWith("size")) {
        System.out.println(st.size());
    }
    str = br.readLine();
}
}
```

Code : 3

```
package Topic_13_Linked_List;
```

```
import java.util.*;
```

```
public class PepLinkedList {
```

```
    public static class Node {
```

```
        int data;
```

```
        Node next;
```

```
        Node(int data) {
```

```
            this.data = data;
```

```
            this.next = null;
```

```
        }
```

```
    }
```

```
    public static class LinkedList {
```

```
        Node head;
```

```
        Node tail;
```

```
        int size;
```

```
        public int size() {
```

```
            return size;
```

```
        }
```

```
        // O(n)
```

```
        // 2nd PG
```

```
        public void display() {
```

```
            Node temp = head;
```

```
            while (temp != null) {
```

```
                System.out.print(temp.data + " ");
```

```
                temp = temp.next;
```

```
            }
```

```
            System.out.println();
```

```
            // for (Node temp = head; temp != null; temp = temp.next) {
```

```
            // System.out.print(temp.data + " ");
```

```
            // }
```

```
            // System.out.print(temp.data + " ");
```

```
        }
```

```
        // O(1)
```

```
        // 4th PG
```

```
        public int getFirst() {
```

```
            if (size == 0) {
```

```
                System.out.println("List is empty");
```

```
                return -1;
```

```
            } else {
```

```
                return head.data;
```

```
            }
```

```
        }
```

```
        // O(1)
```

```

// 4th PG
public int getLast() {
    if (size == 0) {
        System.out.println("List is empty");
        return -1;
    } else {
        return tail.data;
    }
}

```

```

// O(n)
// 4th PG
public int getAt(int idx) {
    if (size == 0) {
        System.out.println("List is empty");
        return -1;
    } else if (idx < 0 || idx >= size) {
        System.out.println("Invalid arguments");
        return -1;
    } else {
        Node temp = head;
        for (int i = 0; i < idx; i++) {
            temp = temp.next;
        }
        return temp.data;
    }
}

```

```

// O(n)
public Node getAt2(int idx) {
    Node temp = head;
    for (int i = 0; i < idx; i++) {
        temp = temp.next;
    }
    return temp;
}

```

```

// O(1)
// 5th PG
public void addFirst(int val) {
    Node node = new Node(val);
    if (size == 0) {
        head = tail = node;
    } else {
        node.next = head;
        head = node;
    }

    size++;
}

```

```

// O(1)
// 1st PG
public void addLast(int val) {
    Node node = new Node(val);
    if (size == 0) {

```

```

        head = tail = node;
    } else {
        tail.next = node;
        tail = node;
    }

    size++;
}

// O(n)
// 6th PG
public void addAt(int idx, int val) {
    if (idx < 0 || idx > size) {
        System.out.println("Invalid arguments");
    } else if (idx == 0) {
        addFirst(val);
    } else if (idx == size) {
        addLast(val);
    } else {
        Node node = new Node(val);
        Node temp = head;
        for (int i = 0; i < idx - 1; i++) {
            temp = temp.next;
        }
        node.next = temp.next;
        temp.next = node;
        size++;
    }
}

```

```

// O(1)
// 3rd PG
public void removeFirst() {
    if (size == 0) {
        System.out.println("List is empty");
    } else if (size == 1) {
        head = tail = null;
    } else {
        head = head.next;
    }
    size--;
}

```

```

// O(n)
public void removeLast() {
    if (size == 0) {
        System.out.println("Empty List");
    } else if (size == 1) {
        head = tail = null;
    } else {
        Node temp = head;
        while (temp.next != tail) {
            temp = temp.next;
        }
        tail = temp;
        temp.next = null;
    }
}

```

```

    }
    size--;
}

// O(n)
public void removeAt(int idx) {
    if (size == 0) {
        System.out.println("Empty List");
    } else if (idx == 0) {
        removeFirst();
    } else if (idx == size - 1) {
        removeLast();
    } else if (idx < 0 || idx >= size) {
        System.out.println("Invalid Arguments");
    } else {
        Node temp = head;
        for (int i = 0; i < idx - 1; i++) {
            temp = temp.next;
        }

        temp.next = temp.next.next;
        size--;
    }
}

```

```

// O(n)
public int kthFromLast(int k) {
    Node fast = head;
    Node slow = head;

    for (int i = 0; i < k; i++) {
        fast = fast.next;
    }

    while (fast.next != null) {
        fast = fast.next;
        slow = slow.next;
    }

    return slow.data;
}

```

```

// O(n)
// middle of linked list
public int getMid() {
    Node fast = head;
    Node slow = head;

    // while (fast != tail && fast.next != tail) {
    while (fast.next != null && fast.next.next != null) {
        fast = fast.next.next;
        slow = slow.next;
    }

    return slow.data;
}

```



```

// O(n^2)
public void reverseIteratively() {
    int l = 0;
    int h = size - 1;
    while (l < h) {
        Node first = getAt2(l);
        Node second = getAt2(h);
        int temp = first.data;
        first.data = second.data;
        second.data = temp;
        l++;
        h--;
    }
}

```

```

// O(n)
public void reversePointerIteratively() {
    Node pre = null;
    Node curr = head;
    Node next = null;
    while (curr != null) {
        next = curr.next;
        curr.next = pre;
        pre = curr;
        curr = next;
    }

    // swapping head and tail
    Node temp = head;
    head = tail;
    tail = temp;
}

```

```

public LinkedList mergeTwoSortedList(LinkedList l1, LinkedList l2) {
    LinkedList list = new LinkedList();
    Node one = l1.head;
    Node two = l2.head;

    while (one != null && two != null) {
        if (one.data < two.data) {
            list.addLast(one.data);
            one = one.next;
        } else {
            list.addLast(two.data);
            two = two.next;
        }
    }

    while (one != null) {
        list.addLast(one.data);
        one = one.next;
    }

    while (two != null) {
        list.addLast(two.data);
    }
}

```

```

        two = two.next;
    }

    return list;
}

public LinkedList mergeSort(Node head, Node tail) {
    if (head == tail) {
        LinkedList list = new LinkedList();
        list.addLast(head.data);
        return list;
    }

    Node mid = midNode(head, tail);
    Node midNext = mid.next;

    LinkedList fsl = mergeSort(head, mid);
    LinkedList ssl = mergeSort(mid.next, tail);

    LinkedList sl = mergeTwoSortedList(fsl, ssl);
    return sl;
}

public Node midNode(Node head, Node tail) {
    Node fast = head;
    Node slow = head;
    while (fast != tail && fast.next != tail) {
        fast = fast.next.next;
        slow = slow.next;
    }
    return slow;
}

// you have sorted list, remove duplicates from list.
// O(n)
public void removeDuplicates() {
    // write your code here
    LinkedList res = new LinkedList();
    while (this.size > 0) {
        int val = this.getFirst();
        this.removeFirst();
        if (res.size == 0 || res.tail.data != val) {
            res.addLast(val);
        }
    }

    this.head = res.head;
    this.tail = res.tail;
    this.size = res.size;
}

public void removeDuplicates2() {
    if (size == 1) {
        return;
    }
}

```

```

Node temp = head;
while (temp.next != null) {
    if (temp.data == temp.next.data) {
        temp.next = temp.next.next;
        size--;
    } else {
        temp = temp.next;
    }
}

tail = temp;
}

```

// arrange all odd elements first then even.

```

public void oddEven() {
    Node i = head;
    Node j = head;

    while (i != null) {
        if (i.data % 2 == 0) {
            i = i.next;
        } else {
            int temp = i.data;
            i.data = j.data;
            j.data = temp;
            i = i.next;
            j = j.next;
        }
    }
}

```

// O(n)

```

public void oddEven2() {
    LinkedList odd = new LinkedList();
    LinkedList even = new LinkedList();

    while (this.size > 0) {
        int data = this.getFirst();
        this.removeFirst();
        if (data % 2 == 0) {
            even.addLast(data);
        } else {
            odd.addLast(data);
        }
    }

    if (odd.size > 0 && even.size > 0) {
        odd.tail.next = even.head;
        this.head = odd.head;
        this.tail = even.tail;
        this.size = odd.size + even.size;
    } else if (odd.size > 0) {
        this.head = odd.head;
        this.tail = odd.tail;
        this.size = odd.size;
    } else if (even.size > 0) {

```

```

        this.head = even.head;
        this.tail = even.tail;
        this.size = even.size;
    }
}

// O(n)
public void kReverse(int k) {
    LinkedList pre = null;
    while (this.size > 0) {
        LinkedList curr = new LinkedList();

        if (this.size >= k) {
            for (int i = 0; i < k; i++) {
                int data = this.getFirst();
                this.removeFirst();
                curr.addFirst(data);
            }
        } else {
            int os = this.size;
            for (int i = 0; i < os; i++) {
                int data = this.getFirst();
                this.removeFirst();
                curr.addLast(data);
            }
        }

        if (pre == null) {
            pre = curr;
        } else {
            pre.tail.next = curr.head;
            pre.tail = curr.tail;
            pre.size += curr.size;
        }
    }
    this.head = pre.head;
    this.tail = pre.tail;
    this.size = pre.size;
}

// you have to display list in reverse without changing the data.
public void displayReverse() {
    displayReverseHelper(head);
    System.out.println();
}

private void displayReverseHelper(Node node) {
    if (node == null) {
        return;
    }

    displayReverseHelper(node.next);
    System.out.print(node.data + " ");
}

// reverse using different approach.(data recursively)

```

```

Node rleft;

public void reverse4() {
    rleft = head;
    reverse4(head, 0);
}

private void reverse4(Node right, int floor) {
    if (right == null) {
        return;
    }

    reverse4(right.next, floor + 1);

    if (floor >= size / 2) {
        int temp = right.data;
        right.data = rleft.data;
        rleft.data = temp;

        rleft = rleft.next;
    }
}

// reverse list(pointer recursively)
public void reversePR() {
    reversePRHelper(head);
    Node temp = head;
    head = tail;
    tail = temp;
    tail.next = null;
}

private void reversePRHelper(Node node) {
    // write your code here

    if (node.next == null) {
        return;
    }

    reversePRHelper(node.next);

    node.next.next = node;
}

public static int findIntersection(LinkedList one, LinkedList two) {
    // write your code here

    Node t1 = one.head;
    Node t2 = two.head;

    int diff = Math.abs(one.size - two.size);

    if (one.size > two.size) {
        for (int i = 0; i < diff; i++) {
            t1 = t1.next;
        }
    }
}

```

```

        } else if (two.size > one.size) {
            for (int i = 0; i < diff; i++) {
                t2 = t2.next;
            }
        }

        while (t1 != t2) {
            t1 = t1.next;
            t2 = t2.next;
        }

        return t1.data;
    }

    // find list is palindrome or not.
    Node left = null;

    public boolean IsPalindrome() {
        // write your code here
        left = head;
        return process(head);
    }

    public boolean process(Node head) {
        if (head == null) {
            return true;
        }

        boolean res = process(head.next);

        if (res == false) {
            return false;
        } else if (left.data != head.data) {
            return false;
        } else {
            left = left.next;
            return true;
        }
    }
}

// fold of linkedlist
// Example 1
// 1->2->3->4->5
// will fold as
// 1->5->2->4->3
// O(n)
Node leftNode = null;

public void fold() {
    leftNode = head;
    foldHelper(head, 0);
}

public void foldHelper(Node node, int level) {
    if (node == null) {
        return;
    }

```

```

    }

    foldHelper(node.next, level + 1);
    if (level > size / 2) {
        Node nextNode = leftNode.next;
        leftNode.next = node;
        node.next = nextNode;
        leftNode = nextNode;
    } else if (level == size / 2) {
        tail = node;
        tail.next = null;
    }
}

// 1. Time complexity -> O(n)
// 2. Space complexity -> Recursion space, O(n)
public static LinkedList addTwoLists(LinkedList one, LinkedList two) {
    // write your code here
    LinkedList res = new LinkedList();
    int oc = addTwoListsHelper(one.head, one.size, two.head, two.size, res);
    if (oc > 0) {
        res.addFirst(oc);
    }
    return res;
}

public static int addTwoListsHelper(Node one, int pv1, Node two, int pv2, LinkedList res) {
    if (one == null && two == null) {
        return 0;
    }

    if (pv1 > pv2) {
        int oc = addTwoListsHelper(one.next, pv1 - 1, two, pv2, res);
        int data = one.data + oc;
        int newData = data % 10;
        int newCarry = data / 10;
        res.addFirst(newData);
        return newCarry;
    } else if (pv2 > pv1) {
        int oc = addTwoListsHelper(one, pv1, two.next, pv2 - 1, res);
        int data = two.data + oc;
        int newData = data % 10;
        int newCarry = data / 10;
        res.addFirst(newData);
        return newCarry;
    } else { // if both place value is equal
        int oc = addTwoListsHelper(one.next, pv1 - 1, two.next, pv2 - 1, res);
        int data = one.data + two.data + oc;
        int newData = data % 10;
        int newCarry = data / 10;
        res.addFirst(newData);
        return newCarry;
    }
}

// delete node without head pointer.

```

```

public static void deleteNode(Node node) {
    // you have given the node ,not the head you have to delete that node from
    // list.

    // we will copy the data of next node into the given node.
    // then we will delete the next node.

    node.data = node.next.data;
    node.next = node.next.next;
}

public static Node pairwiseSwap(Node head) {
    Node temp = head;

    while (temp != null && temp.next != null) {
        int tm = temp.data;
        temp.data = temp.next.data;
        temp.next.data = tm;

        temp = temp.next.next;
    }

    return head;
}

// multiply each node by 3.
public void multiplyBy3() {
    // Node rr = head;

    int Oldcarry = multiplyBy3(head);
    if (Oldcarry > 0) {
        Node node = new Node(Oldcarry);
        node.next = head;
        head = node;
    }
    // this.head = rr;
}

public int multiplyBy3(Node head) {
    if (head == null) {
        return 0;
    }

    int carry = multiplyBy3(head.next);
    int t = head.data * 3;
    head.data = (carry + t) % 10;
    carry = (carry + t) / 10;
    return carry;
}

}

public static void main(String[] args) {
    LinkedList list = new LinkedList();
    list.addFirst(10);
    list.addFirst(20);
}

```



```
list.display();
```

```
list.addLast(30);  
list.addLast(40);  
list.display();
```

```
list.addAt(50, 2);  
list.addAt(60, 5);  
list.display();
```

```
System.out.println(list.kthFromLast(2));  
System.out.println("Middle of list is: " + list.getMid());
```

```
list.reverseIteratively();  
list.display();
```

```
System.out.println(list.getFirst());  
System.out.println(list.getLast());  
System.out.println(list.getAt(3));
```

```
list.removeFirst();  
list.display();
```

```
list.removeLast();  
list.display();
```

```
list.removeAt(2);  
list.display();
```

```
LinkedList newList = list.mergeSort(list.head, list.tail);  
newList.display();
```

```
}
```

```
}
```

Code : 4

```
package Topic_13_Linked_List;
```

```
import java.io.*;
```

```
public class Practice_Linked_List {
```

```
    public static class Node {
```

```
        int data;
```

```
        Node next;
```

```
    }
```

```
    public static class LinkedList {
```

```
        Node head;
```

```
        Node tail;
```

```
        int size;
```

```
        public int size() {
```

```
            return size;
```

```
        }
```

```
        // 1st Program
```

```
        void addLast(int val) {
```

```
            // Write your code here
```

```
            Node node = new Node();
```

```
            node.data = val;
```

```
            node.next = null;
```

```
            if (size == 0) {
```

```
                head = tail = node;
```

```
            } else {
```

```
                tail.next = node;
```

```
                tail = node;
```

```
            }
```

```
            size++;
```

```
        }
```

```
        public void display() {
```

```
            Node temp = head;
```

```
            while (temp != null) {
```

```
                System.out.print(temp.data + " ");
```

```
                temp = temp.next;
```

```
            }
```

```
        }
```

```
        public void removeFirst() {
```

```
            if (size == 0) {
```

```
                System.out.println("List is empty");
```

```
            } else if (size == 1) {
```

```
                head = tail = null;
```

```
            } else {
```

```
                head = head.next;
```

```
            }
```

```
            size--;
```

```
        }
```

```

public int getFirst() {
    if (size == 0) {
        System.out.println("List is empty");
        return -1;
    }
    return head.data;
}

public int getLast() {
    if (size == 0) {
        System.out.println("List is empty");
        return -1;
    }
    return tail.data;
}

public int getAt(int idx) {
    if (size == 0) {
        System.out.println("List is empty");
        return -1;
    } else if (idx < 0 || idx >= size) {
        System.out.println("Invalid arguments");
        return -1;
    } else {
        int index = 0;
        int data = -1;
        Node temp = head;
        while (temp != null) {
            if (index == idx) {
                data = temp.data;
                break;
            }
            index++;
            temp = temp.next;
        }
        return data;
    }
}

public void addFirst(int val) {
    Node node = new Node();
    node.data = val;
    if (size == 0) {
        head = tail = node;
    } else {
        node.next = head;
        head = node;
    }
    size++;
}

public void addAt(int idx, int val) {
    if (idx < 0 || idx >= size) {
        System.out.println("Invalid arguments");
    } else if (idx == 0) {

```

```

        addFirst(val);
    } else if (idx == size) {
        addLast(val);
    } else {
        Node node = new Node();
        node.data = val;
        Node temp = head;
        for (int i = 0; i < idx - 1; i++) {
            temp = temp.next;
        }
        node.next = temp.next;
        temp.next = node;
        size++;
    }
}

public void removeLast() {
    if (size == 0) {
        System.out.println("List is empty");
    } else if (size == 1) {
        head = tail = null;
    } else {
        Node temp = head;
        while (temp != null) {
            if (temp.next.data == tail.data) {
                break;
            }
            temp = temp.next;
        }
        temp.next = null;
        tail = temp;
        size--;
    }
}

public void removeAt(int idx) {
    if (size == 0)
        System.out.println("List is empty");
    else if (idx < 0 || idx >= size) {
        System.out.println("Invalid arguments");
    } else if (idx == 0) {
        removeFirst();
    } else if (idx == size - 1) {
        removeLast();
    } else {
        Node temp = head;
        for (int i = 0; i < idx - 1; i++) {
            temp = temp.next;
        }
        Node current = temp.next;
        temp.next = temp.next.next;
        current.next = null;
        size--;
    }
}

```

```

private Node getNodeAt(int idx) {
    Node temp = head;
    for (int i = 0; i < idx; i++) {
        temp = temp.next;
    }
    return temp;
}

public void reverseDI() {
    int li = 0;
    int ri = size - 1;

    while (li < ri) {
        Node left = getNodeAt(li);
        Node right = getNodeAt(ri);

        int temp = left.data;
        left.data = right.data;
        right.data = temp;

        li++;
        ri--;
    }
}

public void reversePI() {
    Node c = head;
    Node p = null;
    while (c != null) {
        Node temp = c.next;
        c.next = p;
        p = c;
        c = temp;
    }
    Node temp = tail;
    tail = head;
    head = temp;
}

public int kthFromLast(int k) {
    Node f = head;
    Node s = head;
    for (int i = 0; i < k; i++) {
        f = f.next;
    }
    while (f != tail) {
        s = s.next;
        f = f.next;
    }
    return s.data;
}

public int mid() {
    Node f = head;
    Node s = head;
    while (f.next != null && f.next.next != null) {

```

```

        s = s.next;
        f = f.next.next;
    }
    return s.data;
}

public static LinkedList mergeTwoSortedLists(LinkedList l1, LinkedList l2) {
    LinkedList temp = new LinkedList();
    Node pt1 = l1.head;
    Node pt2 = l2.head;

    while (pt1 != null && pt2 != null) {
        if (pt1.data <= pt2.data) {
            temp.addLast(pt1.data);
            pt1 = pt1.next;
        } else if (pt2.data < pt1.data) {
            temp.addLast(pt2.data);
            pt2 = pt2.next;
        }
    }

    while (pt2 != null) {
        temp.addLast(pt2.data);
        pt2 = pt2.next;
    }

    while (pt1 != null) {
        temp.addLast(pt1.data);
        pt1 = pt1.next;
    }

    return temp;
}

public static Node mid(Node head, Node tail) {
    Node f = head;
    Node s = head;
    while (f.next != tail && f.next.next != tail) {
        s = s.next;
        f = f.next.next;
    }
    return s;
}

public static LinkedList mergeSort(Node head, Node tail) {
    if (head == tail) {
        LinkedList list = new LinkedList();
        list.addLast(head.data);
        return list;
    }
    Node mid = mid(head, tail);
    LinkedList firstSortedHalf = mergeSort(head, mid);
    LinkedList secondSortedHalf = mergeSort(mid.next, tail);
    LinkedList sortedList = mergeTwoSortedLists(
        firstSortedHalf,
        secondSortedHalf);
}

```

```

        return sortedList;
    }

    public Node removeFirstAndGet() {
        Node deleteNode = head;
        if (size == 0) {
            System.out.println("List is empty");
        } else if (size == 1) {
            head = tail = null;
        } else {
            head = head.next;
        }
        return deleteNode;
    }

    public void removeDuplicates() {
        // write your code here
        LinkedList res = new LinkedList();
        while (this.size > 0) {
            int val = this.getFirst();
            this.removeFirst();
            if (res.size == 0 || res.tail.data != val) {
                res.addLast(val);
            }
        }

        this.head = res.head;
        this.tail = res.tail;
        this.size = res.size;
    }

    public void oddEven() {
        LinkedList odd = new LinkedList();
        LinkedList even = new LinkedList();

        while (this.size > 0) {
            int data = this.getFirst();
            this.removeFirst();
            if (data % 2 == 0) {
                even.addLast(data);
            } else {
                odd.addLast(data);
            }
        }

        if (odd.size > 0 && even.size > 0) {
            odd.tail.next = even.head;
            this.head = odd.head;
            this.tail = even.tail;
            this.size = odd.size + even.size;
        } else if (odd.size > 0) {
            this.head = odd.head;
            this.tail = odd.tail;
            this.size = odd.size;
        } else if (even.size > 0) {
            this.head = even.head;

```

```

        this.tail = even.tail;
        this.size = even.size;
    }
}

public void kreverse(int k) {
    LinkedList p = new LinkedList();
    LinkedList c = new LinkedList();
    while (size != 0 && k <= size) {
        if (p.size == 0) {
            for (int i = 0; i < k; i++) {
                int data = getFirst();
                removeFirst();
                c.addFirst(data);
            }
            p = c;
        } else {
            for (int i = 0; i < k; i++) {
                int data = getFirst();
                removeFirst();
                c.addFirst(data);
            }
            p.tail.next = c.head;
            p.tail = c.tail;
            p.size = p.size + c.size;
        }
        c = new LinkedList();
    }
    int sz = size();
    while (sz != 0) {
        int data = getFirst();
        removeFirst();
        p.addLast(data);
        sz--;
    }
    this.head = p.head;
    this.tail = p.tail;
    this.size = p.size;
}

private void displayReverseHelper(Node node) {
    // write your code here
    if (node == null) {
        return;
    }
    displayReverseHelper(node.next);
    System.out.print(node.data + " ");
}

public void displayReverse() {
    displayReverseHelper(head);
    System.out.println();
}

private void reversePRHelper(Node node) {
    // write your code here

```



```

        if (node == null) {
            return;
        }
        reversePRHelper(node.next);
        if (node == tail) {
            return;
        } else {
            Node temp = node.next;
            temp.next = node;
        }
    }

    public void reversePR() {
        // write your code here
        reversePRHelper(head);
        head.next = null;
        Node temp = head;
        head = tail;
        tail = temp;
    }

    Node left;

    public boolean IsPalindrome() {
        // write your code here
        left = head;
        return IsPalindromeHelper(head);
    }

    private boolean IsPalindromeHelper(Node node) {
        if (node == null) {
            return true;
        }
        boolean res = IsPalindromeHelper(node.next);
        if (res == false) {
            return false;
        }
        if (node.data == left.data) {
            left = left.next;
            return true;
        } else {
            return false;
        }
    }

    Node leftNode = null;

    public void fold() {
        leftNode = head;
        foldHelper(head, 0);
    }

    private void foldHelper(Node node, int level) {
        if (node == null) {
            return;
        }

```

```

        foldHelper(node.next, level + 1);
    if (level > size / 2) {
        Node nextNode = leftNode.next;
        leftNode.next = node;
        node.next = nextNode;
        leftNode = nextNode;
    } else if (level == size / 2) {
        tail = node;
        tail.next = null;
    }
}

}

public static LinkedList addTwoLists(LinkedList one, LinkedList two) {
    // write your code here
    LinkedList res = new LinkedList();
    int oc = addTwoListsHelper(one.head, one.size, two.head, two.size, res);
    if (oc > 0) {
        res.addFirst(oc);
    }
    return res;
}

public static int addTwoListsHelper(Node one, int pv1, Node two, int pv2, LinkedList res) {
    if (one == null && two == null) {
        return 0;
    }

    int data = 0;
    if (pv1 > pv2) {
        int oc = addTwoListsHelper(one.next, pv1 - 1, two, pv2, res);
        data = one.data + oc;
    } else if (pv2 > pv1) {
        int oc = addTwoListsHelper(one, pv1, two.next, pv2 - 1, res);
        data = two.data + oc;
    } else { // if both place value is equal
        int oc = addTwoListsHelper(one.next, pv1 - 1, two.next, pv2 - 1, res);
        data = one.data + two.data + oc;
    }

    int newData = data % 10;
    int newCarry = data / 10;
    res.addFirst(newData);
    return newCarry;
}

public static int findIntersection(LinkedList one, LinkedList two) {
    // write your code here
    Node head1 = one.head, head2 = two.head;
    int absSize = Math.abs(one.size - two.size);
    if (one.size() > two.size()) {
        for (int i = 0; i < absSize; i++) {
            head1 = head1.next;
        }
    } else {
        for (int i = 0; i < absSize; i++) {
            head2 = head2.next;
        }
    }
}

```

```

        }
    }
    while (head1 != head2) {
        head1 = head1.next;
        head2 = head2.next;
    }
    return head1.data;
}
}

```

```

public static void main(String[] args) throws Exception {
    LinkedList l1 = new LinkedList();
    String[] values = "1 2 3 4 5 6 7 8 9 10 11".split(" ");
    for (int i = 0; i < values.length; i++) {
        int d = Integer.parseInt(values[i]);
        l1.addLast(d);
    }
    // System.out.println("Before");
    // l1.display();
    l1.displayReverse();
    // System.out.println("\nAfter");
    // l1.display();
}

```

```

public static void main2(String[] args) throws Exception {
    BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
    LinkedList list = new LinkedList();

    String str = br.readLine();
    while (str.equals("quit") == false) {
        if (str.startsWith("addLast")) {
            int val = Integer.parseInt(str.split(" ")[1]);
            list.addLast(val);
        } else if (str.startsWith("size")) {
            System.out.println(list.size());
        } else if (str.startsWith("display")) {
            list.display();
        } else if (str.startsWith("removeFirst")) {
            list.removeFirst();
        } else if (str.startsWith("getFirst")) {
            int val = list.getFirst();
            if (val != -1) {
                System.out.println(val);
            }
        } else if (str.startsWith("getLast")) {
            int val = list.getLast();
            if (val != -1) {
                System.out.println(val);
            }
        } else if (str.startsWith("getAt")) {
            int idx = Integer.parseInt(str.split(" ")[1]);
            int val = list.getAt(idx);
            if (val != -1) {
                System.out.println(val);
            }
        } else if (str.startsWith("addFirst")) {

```

```
        int val = Integer.parseInt(str.split(" ")[1]);
        list.addFirst(val);
    } else if (str.startsWith("addAt")) {
        int idx = Integer.parseInt(str.split(" ")[1]);
        int val = Integer.parseInt(str.split(" ")[2]);
        list.addAt(idx, val);
    }
    str = br.readLine();
}
}
```

