LinkedList Assignment

20BD1A1290-Harika Jakkanagari

Program 1:

Given the head of a linked list, remove the n th node from the end of the list and return its head.

Solution:

import java.util.\*;

class Node {

int data;

Node next;

Node(int value)

{

this.data = value;

this.next = null;

}

}

class LinkedList {

static int length(Node head)

{

Node temp = head;

int count = 0;

while (temp != null) {

count++;

temp = temp.next;

}

return count;

}

static LinkedList insert(LinkedList ll,int val){

Node nn=new Node(val);

if(ll.head==null){

ll.head=nn;

}

else{

Node last=ll.head;

while(last.next!=null){

last=last.next;

}

last.next=nn;

}

return ll;

}

static void printList(Node head)

{

Node ptr = head;

while (ptr != null) {

System.out.print(ptr.data + " ");

ptr = ptr.next;

}

System.out.println();

}

static Node delete(Node head, int n)

{

int Length = length(head);

int nodeFromBeginning = Length - n + 1;

Node prev = null;

Node temp = head;

for (int i = 1; i < nodeFromBeginning; i++) {

prev = temp;

temp = temp.next;

}

if (prev == null) {

head = head.next;

return head;

}

else {

prev.next = prev.next.next;

return head;

}

}

static Node head;

public static void main(String[] args)

{

Scanner sc=new Scanner(System.in);

int n=sc.nextInt();

LinkedList ll=new LinkedList();

for(int i=0;i<n;i++){

ll=ll.insert(ll,sc.nextInt());

}

int b=sc.nextInt();

head = delete(head,b);

printList(head);

}

}

Example 1: Input: head = [1,2,3,4,5], n = 2 Output: [1,2,3,5]

Program 2:

You are given the heads of two sorted linked lists list1 and list2. Merge the two lists in a one sorted list. The list should be made by splicing together the nodes of the first two lists. Return the head of the merged linked list.

Solution:

import java.util.\*;

class Node{

int val;

Node next;

Node(int d){

val=d;

next=null;

}

}

class LinkedList{

Node head;

public void insert(Node n){

if(head==null){

head=n;

}

else{

Node temp=head;

while(temp.next!=null){

temp=temp.next;

}

temp.next=n;

}

}

void display(){

Node temp=head;

while(temp!=null){

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

temp=temp.next;

}

}

public static void main(String[] args){

Scanner sc=new Scanner(System.in);

int n=sc.nextInt();

LinkedList ll=new LinkedList();

LinkedList ll1=new LinkedList();

for(int i=0;i<n;i++){

ll.insert(new Node(sc.nextInt()));

}

int n1=sc.nextInt();

//LinkedList ll1=new LinkedList();

for(int i=0;i<n1;i++){

ll1.insert(new Node(sc.nextInt()));

}

ll.head=mergingsortedl(ll.head,ll1.head);

ll.display();

}

public static Node mergingsortedl(Node a1,Node a2){

if(a1==null){

return a2;

}

if(a2==null){

return a1;

}

if(a1.val<a2.val){

a1.next=mergingsortedl(a1.next,a2);

return a1;

}

else{

a2.next=mergingsortedl(a1,a2.next);

return a2;

}

}

}

Example 1: Input: list1 = [1,2,4], list2 = [1,3,4] Output: [1,1,2,3,4,4]

Program 3:

Given a linked list, swap every two adjacent nodes and return its head. You must solve the problem without modifying the values in the list's nodes (i.e., only nodes themselves may be changed.)

Solution:

import java.util.\*;

class Node {

int data;

Node next;

Node(int value)

{

this.data = value;

this.next = null;

}

}

class LinkedList {

static LinkedList insert(LinkedList ll,int val){

Node nn=new Node(val);

if(ll.head==null){

ll.head=nn;

}

else{

Node last=ll.head;

while(last.next!=null){

last=last.next;

}

last.next=nn;

}

return ll;

}

static void printList(Node head)

{

Node ptr = head;

while (ptr != null) {

System.out.print(ptr.data + " ");

ptr = ptr.next;

}

System.out.println();

}

static Node head;

public static void main(String[] args)

{

Scanner sc=new Scanner(System.in);

int n=sc.nextInt();

LinkedList ll=new LinkedList();

for(int i=0;i<n;i++){

ll=ll.insert(ll,sc.nextInt());

}

Swapingpairwise(ll.head);

printList(ll.head);

}

public static void Swapingpairwise(Node head)

{

if (head != null && head.next != null) {

int temp = head.data;

head.data = head.next.data;

head.next.data = temp;

Swapingpairwise(head.next.next);

}

}

}

Example : Input: head = [1,2,3,4] Output: [2,1,4,3]

Program 4:

Given the head of a sorted linked list, delete all duplicates such that each element appears only once. Return the linked list sorted as well.

Solution:

import java.util.\*;

class Node {

int data;

Node next;

Node(int value)

{

this.data = value;

this.next = null;

}

}

class LinkedList {

static LinkedList insert(LinkedList ll,int val){

Node nn=new Node(val);

if(ll.head==null){

ll.head=nn;

}

else{

Node last=ll.head;

while(last.next!=null){

last=last.next;

}

last.next=nn;

}

return ll;

}

static void printList(Node head)

{

Node ptr = head;

while (ptr != null) {

System.out.print(ptr.data + " ");

ptr = ptr.next;

}

System.out.println();

}

static Node head;

public static void main(String[] args)

{

Scanner sc=new Scanner(System.in);

int n=sc.nextInt();

LinkedList ll=new LinkedList();

for(int i=0;i<n;i++){

ll=ll.insert(ll,sc.nextInt());

}

head=removingdup(ll.head);

printList(head);

}

public static Node removingdup(Node head)

{

Node temp;

if(head==null){

return null;

}

if(head.next!=null){

if(head.data==head.next.data){

temp=head.next;

head.next=head.next.next;

removingdup(head);

}

else{

removingdup(head.next);

}

}

return head;

}

}  
Example:

Input: head = [1,1,2]Output: [1,2]

Program 5:  
Given the head of a sorted linked list, delete all nodes that have duplicate numbers, leaving only distinct numbers from the original list. Return the linked list sorted as well.

Solution:

import java.util.\*;

class Node {

int data;

Node next;

Node(int value)

{

this.data = value;

this.next = null;

}

}

class LinkedList {

static LinkedList insert(LinkedList ll,int val){

Node nn=new Node(val);

if(ll.head==null){

ll.head=nn;

}

else{

Node last=ll.head;

while(last.next!=null){

last=last.next;

}

last.next=nn;

}

return ll;

}

static void printList(Node head)

{

Node ptr = head;

while (ptr != null) {

System.out.print(ptr.data + " ");

ptr = ptr.next;

}

System.out.println();

}

static Node head;

public static void main(String[] args)

{

Scanner sc=new Scanner(System.in);

int n=sc.nextInt();

LinkedList ll=new LinkedList();

for(int i=0;i<n;i++){

ll=ll.insert(ll,sc.nextInt());

}

head=removingdup(ll.head);

printList(head);

}

public static Node removingdup(Node head)

{

Node d=new Node(0);

d.next=head;

Node prev=d;

Node cur=head;

while(cur!=null){

while(cur.next!=null && prev.next.data==cur.next.data){

cur=cur.next;

}

if(prev.next==cur){

prev=prev.next;

}

else{

prev.next=cur.next;

}

cur=cur.next;

}

head=d.next;

return head;

}

}

Example:

Input: head = [1,2,3,3,4,4,5]Output: [1,2,5]

Program 6:

Design a data structure that follows the constraints of a Least Recently Used (LRU) cache. Implement the LRUCache class: ● ● ● LRUCache(int capacity) Initialize the LRU cache with positive size capacity. int get(int key) Return the value of the key if the key exists, otherwise return -1. void put(int key, int value) Update the value of the key if the key exists. Otherwise, add the key-value pair to the cache. If the number of keys exceeds the capacity from this operation, evict the least recently used key. The functions get and put must each run in O(1) average time complexity.  
Solution:

import java.util.HashMap;

import java.util.Map;

class LRUCache {

private int capacity;

private Map<Integer, Node> cache;

private Node head;

private Node tail;

class Node {

int key;

int value;

Node prev;

Node next;

Node(int key, int value) {

this.key = key;

this.value = value;

}

}

public LRUCache(int capacity) {

this.capacity = capacity;

cache = new HashMap<>();

head = new Node(0, 0);

tail = new Node(0, 0);

head.next = tail;

tail.prev = head;

}

public int get(int key) {

if (cache.containsKey(key)) {

Node node = cache.get(key);

removeNode(node);

addToHead(node);

return node.value;

}

return -1;

}

public void put(int key, int value) {

if (cache.containsKey(key)) {

Node node = cache.get(key);

node.value = value;

removeNode(node);

addToHead(node);

} else {

if (cache.size() == capacity) {

cache.remove(tail.prev.key);

removeNode(tail.prev);

}

Node newNode = new Node(key, value);

cache.put(key, newNode);

addToHead(newNode);

}

}

private void removeNode(Node node) {

node.prev.next = node.next;

node.next.prev = node.prev;

}

private void addToHead(Node node) {

node.next = head.next;

node.prev = head;

head.next.prev = node;

head.next = node;

}

}

public class Main {

public static void main(String[] args) {

LRUCache lRUCache = new LRUCache(2);

lRUCache.put(1, 1);

lRUCache.put(2, 2);

System.out.println(lRUCache.get(1));

lRUCache.put(3, 3);

System.out.println(lRUCache.get(2));

lRUCache.put(4, 4);

System.out.println(lRUCache.get(1));

System.out.println(lRUCache.get(3));

System.out.println(lRUCache.get(4));

}

}

Example :

Input ["LRUCache", "put", "put", "get", "put", "get", "put", "get", "get", "get"] [[2], [1, 1], [2, 2], [1], [3, 3], [2], [4, 4], [1], [3], [4]]

Output [null, null, null, 1, null, -1, null, -1, 3, 4] Explanation LRUCache lRUCache = new LRUCache(2); lRUCache.put(1, 1); // cache is {1=1} lRUCache.put(2, 2); // cache is {1=1, 2=2} lRUCache.get(1);    // return 1 lRUCache.put(3, 3); // LRU key was 2, evicts key 2, cache is {1=1, 3=3} lRUCache.get(2);    // returns -1 (not found) lRUCache.put(4, 4); // LRU key was 1, evicts key 1, cache is {4=4, 3=3} lRUCache.get(1);    // return -1 (not found) lRUCache.get(3);    // return 3 lRUCache.get(4);    // return 4

Program 7:

Design your implementation of the circular double-ended queue (deque). Implement the MyCircularDeque class: MyCircularDeque(int k) Initializes the deque with a maximum size of k. boolean insertFront() Adds an item at the front of Deque. Returns true if the operation is successful, or false otherwise. boolean insertLast() Adds an item at the rear of Deque. Returns true if the operation is successful, or false otherwise. boolean deleteFront() Deletes an item from the front of Deque. Returns true if the operation is successful, or falseotherwise. boolean deleteLast() Deletes an item from the rear of Deque. Returns true if the operation is successful, or falseotherwise. int getFront() Returns the front item from the Deque. Returns -1 if the deque is empty. int getRear() Returns the last item from Deque. Returns -1 if the deque is empty. boolean isEmpty() Returns true if the deque is empty, or false otherwise. boolean isFull() Returns true if the deque is full, or false otherwise. Solution:

class MyCircularDeque {

private int[] deque;

private int front;

private int rear;

private int size;

private int capacity;

public MyCircularDeque(int k) {

capacity = k;

deque = new int[k];

front = -1;

rear = -1;

size = 0;

}

public boolean insertFront(int value) {

if (isFull()) {

return false;

}

if (isEmpty()) {

front = 0;

rear = 0;

} else {

front = (front - 1 + capacity) % capacity;

}

deque[front] = value;

size++;

return true;

}

public boolean insertLast(int value) {

if (isFull()) {

return false;

}

if (isEmpty()) {

front = 0;

rear = 0;

} else {

rear = (rear + 1) % capacity;

}

deque[rear] = value;

size++;

return true;

}

public boolean deleteFront() {

if (isEmpty()) {

return false;

}

if (front == rear) {

front = -1;

rear = -1;

} else {

front = (front + 1) % capacity;

}

size--;

return true;

}

public boolean deleteLast() {

if (isEmpty()) {

return false;

}

if (front == rear) {

front = -1;

rear = -1;

} else {

rear = (rear - 1 + capacity) % capacity;

}

size--;

return true;

}

public int getFront() {

if (isEmpty()) {

return -1;

}

return deque[front];

}

public int getRear() {

if (isEmpty()) {

return -1;

}

return deque[rear];

}

public boolean isEmpty() {

return size == 0;

}

public boolean isFull() {

return size == capacity;

}

}

public class Main {

public static void main(String[] args) {

MyCircularDeque circularDeque = new MyCircularDeque(3);

System.out.println(circularDeque.insertLast(1));

System.out.println(circularDeque.insertLast(2));

System.out.println(circularDeque.insertFront(3));

System.out.println(circularDeque.insertFront(4));

System.out.println(circularDeque.getRear());

System.out.println(circularDeque.isFull());

System.out.println(circularDeque.deleteLast());

System.out.println(circularDeque.insertFront(4));

System.out.println(circularDeque.getFront());

}

}

Example:

Input ["MyCircularDeque", "insertLast", "insertLast", "insertFront", "insertFront", "getRear", "isFull", "deleteLast", "insertFront", "getFront"]

[[3], [1], [2], [3], [4], [], [], [], [4], []]

Output [null, true, true, true, false, 2, true, true, true, 4]

Explanation MyCircularDeque myCircularDeque = new MyCircularDeque(3); myCircularDeque.insertLast(1);  // return True myCircularDeque.insertLast(2);  // return True myCircularDeque.insertFront(3); // return True myCircularDeque.insertFront(4); // return False, the queue is full. myCircularDeque.getRear();      // return 2 myCircularDeque.isFull();       // return True myCircularDeque.deleteLast();   // return True myCircularDeque.insertFront(4); // return True myCircularDeque.getFront();     // return 4

Program 8:

You are given two linked lists: list1 and list2 of sizes n and m respectively. Remove list1's nodes from the a th node to the b th node, and put list2 in their place. The blue edges and nodes in the following figure indicate the result: Build the result list and return its head.

Solution:

import java.util.Scanner;

class ListNode {

int val;

ListNode next;

ListNode(int val) {

this.val = val;

this.next = null;

}

}

class LinkedList {

ListNode head;

public LinkedList() {

this.head = null;

}

public void add(int val) {

ListNode newNode = new ListNode(val);

if (head == null) {

head = newNode;

} else {

ListNode current = head;

while (current.next != null) {

current = current.next;

}

current.next = newNode;

}

}

public void mergeLists(int a, int b, LinkedList list2) {

ListNode current = head;

int index = 0;

ListNode prev = null;

while (current != null && index < a) {

prev = current;

current = current.next;

index++;

}

if (prev != null) {

prev.next = list2.head;

} else {

head = list2.head;

}

ListNode list2LastNode = list2.head;

while (list2LastNode.next != null) {

list2LastNode = list2LastNode.next;

}

if (current != null) {

list2LastNode.next = current.next;

}

current.next = null;

}

public void display() {

ListNode current = head;

while (current != null) {

System.out.print(current.val + " ");

current = current.next;

}

System.out.println();

}

}

public class Main {

public static void main(String[] args) {

Scanner sc = new Scanner(System.in);

int n = sc.nextInt();

LinkedList ll = new LinkedList();

for (int i = 0; i < n; i++) {

ll.add(sc.nextInt());

}

int n1 = sc.nextInt();

LinkedList ll1 = new LinkedList();

for (int i = 0; i < n1; i++) {

ll1.add(sc.nextInt());

}

int a=sc.nextInt();

int b=sc.nextInt();

ll.mergeLists(a, b, ll1);

ll.display();

}

}

Example:

 list1 = [0,1,2,3,4,5], a = 3, b = 4, list2 = [1000000,1000001,1000002] Output: [0,1,2,1000000,1000001,1000002,5]