学习情况表

|  |  |  |  |
| --- | --- | --- | --- |
| **姓名** | 崔田雨 | **学号** | 2021901365 |
| **学院** | 信息工程学院 | **专业** | 计算机科学与技术（卓越工程师） |
| 学习情况简述 | | | |
| 根据视频理解练习代码  IMG_20220821_195203IMG_20220821_195207IMG_20220821_195211IMG_20220821_195213IMG_20220821_195219 | | | |
| 本周练习过的代码1 | | | |
| 代码1：递归算法的测试  public class Solution3 {  public ListNode removeElements(ListNode head,int val,int depth){  String depthString=generateDepthString(depth);  System.*out*.print(depthString);  System.*out*.println("Call:remove "+val+" in "+head);  if (head == null) {  System.*out*.print(depthString);  System.*out*.println("Return " + head);  return null;  }   ListNode res = removeElements(head.next,val,depth+1);  System.*out*.print(depthString);  System.*out*.println("After remove "+val+":"+res);  ListNode ret;  if (head.val == val)  ret=res;  else {  head.next = res;  ret=head;  }  System.*out*.print(depthString);  System.*out*.println("Return:"+ret);  return ret;  }  private String generateDepthString(int depth){  StringBuilder res=new StringBuilder();  for(int i=0;i<depth;i++)  res.append("--");  return res.toString();  }    public static void main(String[] args) {  int[] nums = {1,2,6,3,4,5,6};  ListNode head = new ListNode (nums);//用数组生成的链表用一个head引用来表示，head指向头节点  System.*out*.println(head);   ListNode res = (new Solution3()).removeElements(head,6,0);  System.*out*.println(res);  } } | | | |
| 本周练习过的代码2 | | | |
| 链表功能的实现（添加元素通过递归的方式）  public class LinkedList <E>{  //组成链表节点的类  private class Node{  public E e;  public Node next;   public Node (E e,Node next){  this.e=e;  this.next=next;  }   public Node(E e){  this(e,null);  }   public Node(){  this(null,null);  }  @Override  public String toString(){  return e.toString();   }  }  private Node dummyHead;  private int size;//让用户不能从外部直接修改这个变量   public LinkedList(){  dummyHead=new Node(null,null);  size=0;  }   //获取链表中的元素个数  public int getSize(){  return size;  }   //返回链表是否为空  public boolean isEmpty(){  return size==0;  }     //在索引Index处添加元素e，注意1：索引为0处；注意2：指向顺序很重要，先是添加的元素指向后面的，然后是前面的指向添加的元素  //在实际应用中不常用，因为链表不考虑索引，练习常用！  public void add(int index,E e){  if(index<0||index>size)  throw new IllegalArgumentException("Add failed.Illegal index.");   Node prev=dummyHead;  for(int i=0;i<index;i++)  prev =prev.next;  /\*Node node=new Node(e);  node.next=prev.next;  prev.next=node;\*/  prev.next=new Node(e,prev.next);  size++;    }  //在链表头添加新的元素e  public void addFirst(E e){  /\* Node node=new Node(e);  node.next=head;  head=node;  head=new Node(e,head);//先创建节点存放e，然后连接head,最后将节点赋值给head   size++;\*/  add(0,e);  }  //在链表末尾添加新的元素e  public void addLast(E e){  add(size,e);  }   //获得链表中第Index个位置的元素  //在链表中不常用  public E get(int index){  if(index<0||index>=size)  throw new IllegalArgumentException("Get failed.Illegal index.");  Node cur=dummyHead.next;//和add中的区别  for(int i=0;i<index;i++)  cur =cur.next;  return cur.e;  }   //获得链表的第一个元素  public E getFirst(){  return get(0);  }  //获得链表中最后一个元素  public E getLast(){  return get(size-1);  }  //修改链表中第index个位置的元素e  public void set(int index,E e){  if(index<0||index>size)  throw new IllegalArgumentException("Set failed.Illegal index.");  Node cur=dummyHead.next;  for(int i=0;i<index;i++)  cur =cur.next;  cur.e=e;  }  //查找链表中是否有元素e;  public boolean contains(E e){  Node cur=dummyHead.next;  while(cur!=null){//判断是不是遍历完一遍  if(cur.e.equals(e))  return true;  cur=cur.next;  }  return false;  }  //从链表中删除index（0-based）位置的元素，返回删除的元素  //在链表中不常用  public E remove(int index){  if(index<0||index>=size)  throw new IllegalArgumentException("Remove failed.Index is illegal.");  Node prev=dummyHead;  for(int i=0;i<index;i++)  prev=prev.next;   Node retNode=prev.next;  prev.next=retNode.next;  retNode.next=null;  size--;  return retNode.e;  }   //从链表中删除第一个元素  public E removeFirst(){  return remove(0);  }  //删除最后一个  public E removeLast(){  return remove(size-1);  }   @Override  public String toString(){  StringBuilder res=new StringBuilder();   /\* Node cur=dummyHead.next;  while(cur!=null){  res.append(cur+"->");  cur=cur.next;  }\*/  for(Node cur=dummyHead.next;cur!=null;cur=cur.next)  res.append(cur+"->");    res.append("NULL");  return res.toString();  } } | | | |
| 本周练习过的代码3 | | | |
| 代码1：反转链表非递归实现  public class ListNode {  int val;  ListNode next;  ListNode(int x){  val=x;  } }  public class Solution {//反转链表的非递归实现  public ListNode reverseList(ListNode head){  ListNode pre=null;  ListNode cur=head;  while(cur!=null){  ListNode next=cur.next;  cur.next=pre;  pre=cur;  cur=next;  }  return pre;  }  }  代码2：用递归实现链表反转  public class Solution2 {//反转链表的递归实现  public ListNode reverseList(ListNode head) {  if (head == null || head.next == null)  return head;  ListNode rev = reverseList(head.next);  head.next.next = head;  head.next = null;  return rev;  }  } | | | |
| 本周练习过的代码4 | | | |
| 代码1：双链表基本功能的实现  public class DoubleLinkedList<T> {  private static int *size*;  private Node<T> first;  private Node<T> last;   private static class Node<T> {  private T data;  private Node<T> pri;  private Node<T> next;   public Node(T data) {  this.data = data;  }   public T getData() {  return data;  }   public void setData(T data) {  this.data = data;  }   public Node<T> getPri() {  return pri;  }   public void setPri(Node<T> pri) {  this.pri = pri;  }   public Node<T> getNext() {  return next;  }   public void setNext(Node<T> next) {  this.next = next;  }   }   public static int getSize() {  return *size*;  }   //添加节点到头部   public boolean addFirst(T value){  Node<T> node = new Node<>(value);  if (*size* == 0) {  first = node;  last = first;  }else {  node.next = first;  first.pri = node;  first = node;  }  *size*++;  return true;  }  //添加节点到尾部   public boolean addLast(T value){  if (*size* == 0){  return addFirst(value);  }else {  Node<T> node = new Node<>(value);  last.next = node;  node.pri = last;  last = node;  }  *size*++;  return true;  }  //元素添加到指定位置   public boolean add(int index,T value){  if (index < 0 || index > *size*){  throw new IndexOutOfBoundsException("数据下标越界 Index:" + index + "\tsize:" + *size*);  } else if(index == 0){  return addFirst(value);  }else if (index == *size*){  return addLast(value);  }else {  Node<T> node = new Node<>(value);  Node<T> head = first;  for (int i = 0; i < index-1; i++) {  head = head.getNext();  }  node.next = head.next;  head.next = node;  node.pri = head;  node.next.pri = node;  }  *size*++;  return true;  }   //删除头结点  public T removeFirst(){  if (*size* == 0){  throw new RuntimeException("链表为空！");  }  T data = first.getData();  Node<T> node = first.next;  node.setPri(null);  first = node;  return data;  }   // 删除尾节点  public T removeLast(){  if (*size* == 0){  throw new RuntimeException("链表为空！");  }  T data = last.getData();  Node<T> node = last.pri;  node.setNext(null);  last = node;  return data;  }   // 删除指定下标结点  public T remove(int index){  if (*size* == 0){  throw new RuntimeException("链表为空！");  }  //注意添加的时候，下标取不到size，但是添加的位置可以是size，但是删除的时候不行  if (index < 0 || index > *size*-1){  throw new IndexOutOfBoundsException("数据下标越界 Index:" + index + "\tsize:" + *size*);  } else if(index == 0){  return removeFirst();  }else {  Node<T> node = first;  for (int i = 0; i < index - 1; i++) {  node = node.next;  }  Node<T> temp = node.next;  if (temp != last){  T data = temp.getData();  node.next = temp.next;  temp.next.pri = node;  temp.setNext(null);  return data;  }else {  return removeLast();  }  }  }   //获取对应下标数据  public T getData(int index){  if (index<0 || index>*size*-1){  throw new IndexOutOfBoundsException("数据下标越界 Index:" + index + "\tsize:" + *size*);  }else if (*size* == 0){  throw new RuntimeException("链表为空");  }else if (*size* == 1){  return first.data;  }else {  Node<T> node = first;  for (int i = 0; i < index; i++) {  node = node.next;  }  return node.data;  }  }   // 清空链表  public void clear(){  first.next = null;  last = first;  }   //遍历输出当前所有数据  public void print(){  if (*size* == 0) {  System.*out*.println("该链表为空!");  }  Node<T> node = first;  while (node != null) {  System.*out*.print(node.getData() + "\t");  node = node.next;  }  System.*out*.println();  }   //详细遍历输出:前驱节点值,当前节点值,后继节点值  public void detailPrint(){  if (*size* == 0) {  System.*out*.println("该链表为空!");  }  Node<T> node = first;  while (node != null) {  System.*out*.print("前驱节点值：");  System.*out*.printf("%-5s",node.pri == null ? "null\t" : node.pri.getData()+"\t");  System.*out*.print("当前节点值：");  System.*out*.printf("%-6s",node.getData() + "\t");  System.*out*.print("后继节点值：");  System.*out*.printf("%-5s",node.next == null ? "null\t" : node.next.getData()+"\t");  System.*out*.println();  node = node.getNext();  }  System.*out*.println();  } } | | | |