学习情况表

|  |  |  |  |
| --- | --- | --- | --- |
| **姓名** | 杜金瑞 | **学号** | 2020905073 |
| **学院** | 信息工程学院 | **专业** | 计算机科学与技术 |

|  |
| --- |
| 学习情况简述 |
| C:\Users\29043\Documents\Tencent Files\2904326062\Image\C2C\985CC3A32D4879D91A01EFC88099CEC7.jpg  C:\Users\29043\Documents\Tencent Files\2904326062\Image\C2C\8982F7F830375E9B61D593F256EE179B.jpg |
| 本周练习过的代码（例） |
| import java.util.LinkedList; import java.util.Queue; import java.util.Stack;  public class BST <E extends Comparable <E>>{   public class Node{   public E e;  public Node left,right;   public Node(E e){  this.e=e;  left=null;  right=null;  }  }   private Node root;  private int size;   public BST(){  root=null;  size=0;  }   public int getSize(){  return size;  }   public boolean isEmpty(){  return size==0;  }   public void add(E e){   if(root == null){  root = new Node(e);  size++;  }  else  add(root,e);  }   private void add(Node node,E e){   if(e.equals(node.e))  return;  else  if(e.compareTo(node.e)<0 && node.left==null){  node.left = new Node(e);  size++;  return;  }  else  if(e.compareTo(node.e)>0 && node.right==null){  node.right = new Node(e);  size++;  return;  }   if(e.compareTo(node.e)<0)  add(node.left,e);  else  add(node.right,e);  }   public void addR(E e){  root=addR(root,e);  }  private Node addR(Node node,E e){   if(node==null){  size++;  return new Node(e);  }   if(e.compareTo(node.e)<0)  node.left=addR(node.left,e);  else if(e.compareTo(node.e)>0)  node.right=addR(node.right,e);   return node;  }   public boolean contains(E e){  return contains(root,e);  }   //查找以node 为根的节点中是否包含e  private boolean contains(Node node,E e){   if(node == null)  return false;   if(e.compareTo(node.e) == 0)  return true;  else if(e.compareTo(node.e) < 0)  return contains(node.left,e);  else return contains(node.right,e);  }    public void preOrderNR(){   Stack<Node> stack=new Stack<>(); //stack中装的Node  stack.push(root);  while(!stack.isEmpty()){   Node cur = stack.pop();  System.*out*.println(cur.e);   if(cur.right!=null)  stack.push(cur.right);  if(cur.left!=null)  stack.push(cur.left);  }  }  public void preOrder(){  preOrder(root);  }   private void preOrder(Node node){   if(node == null)  return;   System.*out*.println(node.e);  preOrder(node.left);  preOrder(node.right);  }   public void inOrder(){  inOrder(root);  }   private void inOrder(Node node){   if(node == null)  return;   inOrder(node.left);  System.*out*.println(node.e);  inOrder(node.right);  }   public void postOrder(){  postOrder(root);  }   private void postOrder(Node node){   if(node == null)  return;   postOrder(node.left);  postOrder(node.right);  System.*out*.println(node.e);  }   public void levelOrder(){  Queue<Node> q=new LinkedList<>(); //Queue为接口，链表是底层数据结构  q.add(root);  while(!q.isEmpty()){  Node cur = q.remove();  System.*out*.println(cur.e);   if(cur.right!=null)  q.add(cur.right);  if(cur.left!=null)  q.add(cur.left);   }  }   public E minimum(){  if(size==0)  throw new IllegalArgumentException("Is Empty!");   return minimum(root).e;  }   //返回以Node为根的二分搜索树的最小值所在节点  public Node minimum(Node node){  if(node.left==null)  return node;   return minimum(node.left);  }   public E maximum(){  if(size==0)  throw new IllegalArgumentException("Is Empty!");   return maximum(root).e;  }   //返回以Node为根的二分搜索树的最大值所在节点  public Node maximum(Node node){  if(node.right==null)  return node;   return maximum(node.right);  }   public E removeMin(){  E ret = minimum();  root= removeMin(root);  return ret;  }   //删除以node 为根的二分搜索树中最小节点  //返回删除节点后新的二分搜索树的根  public Node removeMin(Node node){  if(node.left==null){  Node right = node.right;  node.right=null;  size--;  return right;  }   node.left = removeMin(node.left);  return node;  }   public E removeMax(){  E ret = maximum();  root= removeMax(root);  return ret;  }   //删除以node 为根的二分搜索树中最大节点  //返回删除节点后新的二分搜索树的根  public Node removeMax(Node node){  if(node.right==null){  Node left = node.left;  node.left=null;  size--;  return left;  }   node.right = removeMin(node.right);  return node;  }   public void remove(E e){  root= remove(root,e); //将删除元素后树的根节点返回  }   public Node remove(Node node,E e){  if(node==null)  return null;   if(e.compareTo(node.e)<0) {  node.left = remove(node.left, e);  return node;  }  else if(e.compareTo(node.e)>0) {  node.right = remove(node.right, e);  return node;  }  else{ //e.compareTo(node.e)==0  //待删除节点右子树为空  if(node.right==null){  Node left=node.left;  node.left=null;  size--;  return left;  }   //待删除节点左子树为空  if(node.left==null){  Node right = node.right;  node.right=null;  size--;  return right;  }   //待删除节点左右子树都不为空  Node successor=minimum(node.right);  successor.right=removeMin(node.right);  successor.left=node.left;   node.left=node.right=null;  return successor;  }  }  @Override  public String toString(){   StringBuilder res = new StringBuilder();  generateBSTString(root,0,res);  return res.toString();  }   private void generateBSTString(Node node,int depth,StringBuilder res){   if(node == null){  res.append(generateDepthString(depth) +"null\n" );  return;  }   res.append(generateDepthString(depth) + node.e + "\n");  generateBSTString(node.left,depth + 1 , res);  generateBSTString(node.right,depth + 1, res);  }   private String generateDepthString(int depth){   StringBuilder res = new StringBuilder();  for(int i=0 ; i < depth ; i++)  res.append("--");  return res.toString();  } }  public class BSTMap <K extends Comparable<K>,V> implements Map<K,V>{ public class Node {   public K key;  public V value;   public Node left, right;   public Node(K key, V value) {  this.key = key;  this.value = value;  left = null;  right = null;  } }   private Node root;  private int size;   public BSTMap(){  root =null;  size=0;  }   public int getsize(){  return size;  }   @Override  public boolean isempty(){  return size==0;  }   public void addd(K key,V value){  root =addd(root,key,value);  }  private Node addd(Node node,K key,V value ){   if(node==null){  size++;  return new Node(key,value);  }   if(key.compareTo(node.key)<0)  node.left=addd(node.left,key,value);  else if(key.compareTo(node.key)>0)  node.right=addd(node.right,key,value);  else  node.value=value;   return node;  }   private Node getnode(Node node,K key){   if(node==null)  return null;   if(node.key.compareTo(key)==0)  return node;  else if(node.key.compareTo(key)<0)  return getnode(node.right,key);  else  return getnode(node.left,key);   }   @Override  public boolean contain(K key){  return getnode(root,key)==null;  }   @Override  public V gett(K key){  Node node= getnode(root,key);  return node==null? null:node.value;  }   @Override  public void set(K key,V newValue){  Node node=getnode(root,key);   if(node==null)  throw new IllegalArgumentException("is Empty");   node.value=newValue;   }   public Node minimum(Node node){  if(node.left==null)  return node;   return minimum(node.left);  }   public Node removeMin(Node node){  if(node.left==null){  Node right = node.right;  node.right=null;  size--;  return right;  }   node.left = removeMin(node.left);  return node;  }   @Override  public V removee(K key){  Node node = getnode(root,key);  if(node!=null){  root=removee(root,key);  return node.value;  }  return null;  }  public Node removee(Node node,K key){  if(node==null)  return null;   if(key.compareTo(node.key)<0) {  node.left = removee(node.left, key);  return node;  }  else if(key.compareTo(node.key)>0) {  node.right = removee(node.right, key);  return node;  }  else{ //e.compareTo(node.e)==0  //待删除节点右子树为空  if(node.right==null){  Node left=node.left;  node.left=null;  size--;  return left;  }   //待删除节点左子树为空  if(node.left==null){  Node right = node.right;  node.right=null;  size--;  return right;  }   //待删除节点左右子树都不为空  Node successor=minimum(node.right);  successor.right=removeMin(node.right);  successor.left=node.left;   node.left=node.right=null;  return successor;  }  } }  public class BSTSet <E extends Comparable<E>> implements Set{   private BST<E> bst;   public BSTSet(){  bst = new BST<>();  }   @Override  public int getSize(){  return bst.getSize();  }   @Override  public boolean isEmpty(){  return bst.isEmpty();  }   @Override  public void add(E e){  bst.add(e);  }   @Override  public boolean contains(E e){  return bst.contains(e);  }   @Override  public void remove(E e){  bst.remove(e);  }  }  public class LinkedListMap <K,V> implements Map <K,V>{   public class Node{   public K key;  public V value;   public Node next;   public Node(K key,V value,Node next){  this.key=key;  this.value=value;  this.next=next;  }   public Node(K key){this(key,null,null)};   public Node(){this(null,null,null)};   @Override  public String toString(){return key.toString()+":"+value.toString()};   }   private Node dummyHead;  private int size;   public LinkedListMap(){  dummyHead=new Node();  size=0;  }   public int getsize(){  return size;  }   @Override  public boolean isempty(){  return size==0;  }   private Node getNode(K key){  Node cur=dummyHead.next;  while(cur!=null){  if(cur.key.equals(key))  return cur;  cur=cur.next;   }  return null;  }   @Override  public boolean contain(K key){  return getNode(key)!=null;  }   @Override  public V gett(K key){  Node node=getNode(key);  return node==null? null: node.value;  }   @Override  public void addd(K key,V value){  Node node=getNode(key);  if(node==null){  dummyHead.next=new Node(key,value,dummyHead.next);  size++;  }  else  node.value=value;  }   @Override  public void set(K key,V newValue){  Node node=getNode(key);  if(node==null)  throw new IllegalArgumentException(key +"isn't exist");   node.value=newValue;  }   @Override  public V removee(K key){   Node prev=dummyHead;  while(prev.next!=null){  if(prev.next.key.equals(key))  break;  prev=prev.next;  }   if(prev.next!=null){  Node delNode=prev.next;  prev.next=delNode.next;  delNode.next=null;  size--;  return delNode.value;  }   return null;  } }  public class LinkedListSet <E>{   private LinkedList<E> list;   public LinkedListSet(){  list=new LinkedList<>();  }   @Override  public int getSize(){  return list.getSize();  }   @Override  public boolean isEmpty(){  return list.isEmpty();  }   @Override  public boolean contains(E e){  return list.contains(e);  }   @Override  public void add(E e){  if(!list.contains(e))  list.addFirst(e);  }   }  public interface Map <K,V>{   void addd(K key,V value);  V removee(K key);  boolean contain(K key);  V gett(K key);  void set(K key,V newValue);  boolean isempty();   }  public interface Set <E>{  void add(E e);  void remove(E e);  boolean contains(E e);  int getSize();  boolean isEmpty(); }  import java.util.TreeSet; class Solution {  public int uniqueMorseRepresentations(String[] words) {   String[] codes={".-","-...","-.-.","-..",".","..-.","--.","....","..",".---","-.-",".-..","--","-.","---",".--.","--.-",".-.","...","-","..-","...-",".--","-..-","-.--","--.."};   TreeSet<String> Tree=new TreeSet<>();   for(String word:words){   StringBuilder res=new StringBuilder();  for(int i =0;i<word.length();i++)  res.append( codes[word.charAt(i)-'a'] );   Tree.add(res.toString());   }   return Tree.size();  } } |