## 1. 顺序表

*///////////////////////////////////////  
// 文件名: LList.java  
// 描述: 线性表的抽象数据类型接口  
// 作者: 殷建  
///////////////////////////////////////*public interface LList<T> *//线性表接口，泛型参数 T 表示数据元素的数据类型*{  
 void clear() ; *//将线性表清空* boolean isEmpty(); *//判断线性表是否空* int size(); *//返回线性表长度* T get(int i); *//返回第 i(i≥0)个元素* T set(int i, T x); *//设置第 i 个元素值为 x* void add(T x); *// 添加元素到末尾* void add(int i, T x); *//插入 x 作为第 i 个元素* T remove(int i); *//删除第 i 个元素并返回被删除对象* public boolean remove(Object x); *// 删除指定元素（首次出现）* boolean contains(Object x);  
 int indexOf(Object x); *//查找,返回首次出现的关键字为 x 的元素*}

*///////////////////////////////////////  
// 文件名: SeqList.java  
// 描述: 顺序表的算法实现  
// 作者: 殷建  
///////////////////////////////////////*public class SeqList<T> implements LList<T>  
{  
 private Object[] elements; *// 存储元素的数组* private int size; *// 当前元素个数* private static final int *DEFAULT\_CAPACITY* = 10; *// 默认初始容量  
  
 // 构造方法* public SeqList()  
 {  
 this(*DEFAULT\_CAPACITY*);  
 }  
  
 public SeqList(int initialCapacity)  
 {  
 if (initialCapacity <= 0) {  
 throw new IllegalArgumentException("初始容量必须大于0");  
 }  
 this.elements = new Object[initialCapacity];  
 this.size = 0;  
 }  
  
 *// 清空顺序表* public void clear()  
 {  
 for (int i = 0; i < size; i++) {  
 elements[i] = null;  
 }  
 size = 0;  
 }  
  
 *// 判断线性表是否空* public boolean isEmpty()  
 {  
 return size == 0;  
 }  
  
 *// 获取元素数量* public int size()  
 {  
 return size;  
 }  
  
 *// //返回第 i(i≥0)个元素* @SuppressWarnings("unchecked")  
 public T get(int i)  
 {  
 if (i < 0 || i >= size) {  
 throw new IndexOutOfBoundsException("索引越界");  
 }  
 return (T) elements[i];  
 }  
  
 *// 设置第 i 个元素值为 x* @SuppressWarnings("unchecked")  
 public T set(int i, T x)  
 {  
 if (i < 0 || i >= size) {  
 throw new IndexOutOfBoundsException("索引越界");  
 }  
 T oldValue = (T) elements[i];  
 elements[i] = x;  
 return oldValue;  
 }  
  
 *// 添加元素到末尾* public void add(T x)  
 {  
 ensureCapacity(size + 1);  
 elements[size++] = x;  
 }  
  
 *// 插入 x 作为第 i 个元素* public void add(int i, T x)  
 {  
 if (i < 0 || i > size) {  
 throw new IndexOutOfBoundsException("索引越界");  
 }  
 ensureCapacity(size + 1);  
 System.*arraycopy*(elements, i, elements, i + 1, size - i);  
 elements[i] = x;  
 size++;  
 }  
  
 *// 删除指定位置的元素* @SuppressWarnings("unchecked")  
 public T remove(int i)  
 {  
 if (i < 0 || i >= size) {  
 throw new IndexOutOfBoundsException("索引越界");  
 }  
 T oldValue = (T) elements[i];  
 int numMoved = size - i - 1;  
 if (numMoved > 0) {  
 System.*arraycopy*(elements, i + 1, elements, i, numMoved);  
 }  
 elements[--size] = null; *// 清除引用，帮助GC* return oldValue;  
 }  
  
 *// 删除指定元素（首次出现）* public boolean remove(Object x) {  
 if (x == null) {  
 for (int i = 0; i < size; i++) {  
 if (elements[i] == null) {  
 remove(i);  
 return true;  
 }  
 }  
 } else {  
 for (int i = 0; i < size; i++) {  
 if (x.equals(elements[i])) {  
 remove(i);  
 return true;  
 }  
 }  
 }  
 return false;  
 }  
  
 *// 检查是否包含某元素* public boolean contains(Object x)  
 {  
 return indexOf(x) >= 0;  
 }  
  
 *// 查找元素的索引（首次出现）* public int indexOf(Object x)  
 {  
 if (x == null) {  
 for (int i = 0; i < size; i++) {  
 if (elements[i] == null) {  
 return i;  
 }  
 }  
 } else {  
 for (int i = 0; i < size; i++) {  
 if (x.equals(elements[i])) {  
 return i;  
 }  
 }  
 }  
 return -1;  
 }  
  
 *// 确保容量足够* private void ensureCapacity(int minCapacity) {  
 if (minCapacity > elements.length) {  
 int newCapacity = elements.length \* 2;  
 if (newCapacity < minCapacity) {  
 newCapacity = minCapacity;  
 }  
 Object[] newElements = new Object[newCapacity];  
 System.*arraycopy*(elements, 0, newElements, 0, size);  
 elements = newElements;  
 }  
 }  
  
 *// toString方法，方便打印* @Override  
 public String toString() {  
 if (size == 0) {  
 return "[]";  
 }  
 StringBuilder sb = new StringBuilder("[");  
 for (int i = 0; i < size; i++) {  
 sb.append(elements[i]);  
 if (i < size - 1) {  
 sb.append(", ");  
 }  
 }  
 sb.append("]");  
 return sb.toString();  
 }  
  
 *// 测试代码* public static void main(String[] args) {  
 SeqList<String> list = new SeqList<>();  
  
 *// 添加元素* list.add("Apple");  
 list.add("Banana");  
 list.add("Orange");  
 System.*out*.println("初始列表: " + list);  
  
 *// 在指定位置插入* list.add(1, "Grape");  
 System.*out*.println("插入后: " + list);  
  
 *// 获取元素* System.*out*.println("索引1的元素: " + list.get(1));  
  
 *// 设置元素* list.set(2, "Pear");  
 System.*out*.println("修改后: " + list);  
  
 *// 删除元素* list.remove(0);  
 System.*out*.println("删除后: " + list);  
  
 *// 查找元素* System.*out*.println("包含Pear吗? " + list.contains("Pear"));  
 System.*out*.println("Pear的索引: " + list.indexOf("Pear"));  
 }  
}

## 2. 单链表

*///////////////////////////////////////  
// Filename: LinearList.java  
// Description: The interface of linear list  
// Author: Jian Yin  
///////////////////////////////////////*public interface LinearList <T>  
{  
 void clear() ; *//将线性表清空* boolean isEmpty(); *//判断线性表是否空* int size(); *//返回线性表长度* T get(int i); *//返回第 i(i≥0)个元素* void set(int i, T x); *//设置第 i 个元素值为 x* void add(T x); *// 添加元素到末尾* void add(int i, T x); *//插入 x 作为第 i 个元素* T remove(int i); *//删除第 i 个元素并返回被删除对象* boolean remove(T x); *// 删除指定元素（首次出现）* boolean contains(T x);  
 int indexOf(T x); *//查找,返回首次出现的关键字为 x 的元素*}

*/// /////////////////////////////////////////////  
/// Filename: Node.java  
/// Description: The node of SingleLinedList  
/// Author: Jian Yin  
/// /////////////////////////////////////////////*public class Node<T>  
{  
 public T value;  
 public Node<T> next;  
  
 public Node(T data, Node<T> next)  
 {  
 *//构造结点，data 指定数据元素，next 指定后继结点* this.value = data;  
 this.next = next;  
 }  
  
 public Node()  
 {  
 this(null, null); *//调用 Node(T data, Node<T> next)构造函数* }  
  
 public Node(T data)  
 {  
 this(data, null); *//调用 Node(T data, Node<T> next)构造函数* }  
}

*/////////////////////////////////////////////////  
// Filename: SingleLinkedList.java  
// Description: The class of single linked list  
// Author: Jian Yin  
/////////////////////////////////////////////////*public class SingleLinkedList<T> implements LinearList<T>  
{  
 public Node<T> head; *//头指针，指向单链表的头结点  
  
 // 默认构造方法，构造空单链表* public SingleLinkedList()  
 {  
 this.head = new Node<T>(); *// 创建头结点，data 和 next 值均为 null* }  
  
 *// 将线性表清空* public void clear()  
 {  
 head.next = null;  
 }  
  
 *// 判断线性表是否空* public boolean isEmpty()  
 {  
 if(head.next==null)  
 return true;  
 else  
 return false;  
 }  
  
 *// 返回线性表长度* public int size()  
 {  
 int count = 0;  
 Node<T> p = this.head.next; *//head.next 表示单链表第一个结点* while(p != null) *//若单链表未结束* {  
 count++;  
 p = p.next; *//p 到达后继结点* }  
 return count;  
 }  
  
 *// 返回第 i(i≥0)个元素* @SuppressWarnings("unchecked")  
 public T get(int i)  
 {  
 Node<T> p = this.head.next;  
 for(int j=0; p!=null && j<i; j++)  
 p = p.next;  
 if (i>=0 && p!=null)  
 return p.value; *//p 指向第 i 个结点* else throw new IndexOutOfBoundsException("out of bounds");*//抛出序号越界异常* }  
  
 *// 设置第 i 个元素值为 x* @SuppressWarnings("unchecked")  
 public void set(int i, T x)  
 {  
 if (x==null)  
 return; *// 不能设置元素为空对象* Node<T> p = this.head.next;  
 *// p从单链表第一个元素开始查找第 i 个元素的位置* for (int j=0; p!=null && j<i; j++)  
 p = p.next;  
 if (i>=0 && p!=null)  
 p.value = x;  
 else throw new IndexOutOfBoundsException("out of bounds");*//抛出序号越界异常* }  
  
 *// 插入 x 作为第 i 个元素* public void add(int i, T x)  
 {  
 if (x==null)  
 return;  
 Node<T> p = this.head; *// p指向头结点* for (int j=0; p.next!=null && j<i; j++) *// 寻找插入位置* p = p.next; *// 循环停止时，p指向第 i-1 个结点或最后一个结点* Node<T> q = new Node<T>(x);  
 q.next = p.next;  
 p.next = q;  
 }  
  
 *// 添加元素到末尾* public void add(T x)  
 {  
 if (x==null)  
 return;  
 Node<T> p = this.head; *// p指向头结点* while(p.next!=null) *// 寻找插入位置* p = p.next; *// 循环停止时，p最后一个结点* Node<T> q = new Node<T>(x);  
 q.next = p.next;  
 p.next = q;  
 }  
  
 *// 删除第 i 个元素并返回被删除对象* @SuppressWarnings("unchecked")  
 public T remove(int i)  
 {  
 Node<T> p = this.head;  
 for(int j=0; p.next!=null && j<i; j++) *// p定位到待删除的前一个结点* p = p.next;  
 if(i>=0 && p!=null)  
 {  
 T old = p.next.value; *//获得原对象* p.next = p.next.next; *//删除 p 所指的后继结点* return old;  
 }  
 else throw new IndexOutOfBoundsException("out of bounds");*//抛出序号越界异常* }  
  
 *// 删除指定元素（首次出现）* public boolean remove(T x)  
 {  
 Node<T> p = this.head;  
 while(p.next!=null && p.next.value != x)  
 p = p.next;  
 if(p.next!=null)  
 {  
 p.next = p.next.next;  
 return true;  
 }  
 else  
 return false;  
 }  
  
 public boolean contains(T x)  
 {  
 Node<T> p = this.head;  
 while(p.next!=null && p.next.value != x)  
 p = p.next;  
 if(p.next!=null)  
 return true;  
 else  
 return false;  
 }  
  
 *// 查找,返回首次出现的关键字为 x 的元素* public int indexOf(T x)  
 {  
 int count = 0;  
 Node<T> p = this.head;  
 while(p.next!=null && p.next.value != x)  
 {  
 p = p.next;  
 count++;  
 }  
 if(p.next!=null)  
 return count;  
 else  
 return -1;  
 }  
  
 *// toString方法，方便打印* @Override  
 public String toString()  
 {  
 if(this.size() == 0) {  
 return "[]";  
 }  
 StringBuilder sb = new StringBuilder("[");  
 for(int i = 0; i < this.size(); i++)  
 {  
 sb.append(get(i));  
 if(i < this.size() - 1)  
 {  
 sb.append(", ");  
 }  
 }  
 sb.append("]");  
 return sb.toString();  
 }  
  
 *// 测试代码* public static void main(String[] args)  
 {  
 SingleLinkedList<String> list = new SingleLinkedList<>();  
  
 *// 添加元素* list.add("Apple");  
 list.add("Banana");  
 list.add("Orange");  
 System.*out*.println("初始列表: " + list);  
  
 *// 在指定位置插入* list.add(1, "Grape");  
 System.*out*.println("插入后: " + list);  
  
 *// 获取元素* System.*out*.println("索引1的元素: " + list.get(1));  
  
 *// 设置元素* list.set(2, "Pear");  
 System.*out*.println("修改后: " + list);  
  
 *// 删除元素* list.remove(0);  
 System.*out*.println("删除后: " + list);  
  
 *// 查找元素* System.*out*.println("包含Pear吗? " + list.contains("Pear"));  
 System.*out*.println("Pear的索引: " + list.indexOf("Pear"));  
 }  
}

## 3. 顺序树

*//seqTree.java  
//完全二叉树的顺序存储表示法  
//假设下标从0开始*import java.util.Vector;  
  
public class SeqTree<T> {  
 private Vector<T> vector;  
  
 *// 创建一棵顺序树* public SeqTree() {  
 vector = new Vector<>();  
 }  
  
 *// 左孩子结点编号* public int leftChild(int i)  
 {  
 return 2\*i + 1;  
 }  
  
 *// 右孩子结点编号* public int rightChild(int i)  
 {  
 return 2\*i + 2;  
 }  
  
 *// 双亲结点编号* public int parent(int i)  
 {  
 if(i>0)  
 return (i-1)/2; *// 向下取整* else  
 return -1;  
 }  
  
 *// 对一个顺序存储的完全二叉树赋值* public void assignTree(T[] p, int n)  
 {  
 for(int i=0; i<n; i++)  
 {  
 vector.add(p[i]);  
 }  
 }  
  
 *// 先序遍历* public void preOrderTraverse(int i)  
 {  
 if(i<vector.size())  
 {  
 System.*out*.print(vector.get(i)+" ");  
 preOrderTraverse(leftChild(i));  
 preOrderTraverse(rightChild(i));  
 }  
 }  
  
 *// 中序遍历* public void inOrderTraverse(int i)  
 {  
 if(i<vector.size())  
 {  
 System.*out*.print(vector.get(i)+" ");  
 inOrderTraverse(leftChild(i));  
 inOrderTraverse(rightChild(i));  
 }  
 }  
  
 *// 后序遍历* public void posOrderTraverse(int i)  
 {  
 if(i<vector.size())  
 {  
 posOrderTraverse(leftChild(i));  
 posOrderTraverse(rightChild(i));  
 System.*out*.print(vector.get(i)+" ");  
 }  
 }  
  
 *// 设计将树中序号最大的叶子结点的祖先结点全部打印输出的算法* void printAncestorsOfMaxLeaf()  
 {  
 int i;  
 int n = vector.size();  
 int maxLeafIndex = -1;  
 *// 寻找序号最大的叶子节点* for(i=n-1; i >= 0; --i)  
 {  
 if(leftChild(i)>=n && rightChild(i)>=n)  
 {  
 maxLeafIndex = i;  
 break;  
 }  
 }  
 if (maxLeafIndex == -1)  
 {  
 return; *// 没有叶子节点的情况* }  
 int parentIndex = maxLeafIndex;  
 while(parentIndex>-1)  
 {  
 System.*out*.print(vector.get(parentIndex)+" ");  
 parentIndex = parent(parentIndex);  
 }  
 }  
  
 public static void main(String[] args){  
 SeqTree<String> tree = new SeqTree<>();  
 String[] a = {"张钧富", "张梓睿", "张梓辉", "张沐江", "张沐海", "张泽宇", "张泽阳", "张炜斌", "张炜安", "张焕然"};  
 tree.assignTree(a, 10);  
 tree.preOrderTraverse(0);  
 System.*out*.println("\n");  
 tree.printAncestorsOfMaxLeaf();  
 }  
}

## 4. 二叉树

*/////////////////////////////////////////////////  
// Filename: BinTreeNode.java  
// Description: The node class of binary tree  
// Author: Jian Yin  
/////////////////////////////////////////////////*public class BinTreeNode<T>  
{  
 public T data;  
 public BinTreeNode<T> left;  
 public BinTreeNode<T> right;  
  
 public BinTreeNode(T value) {  
 data = value;  
 left = null;  
 right = null;  
 }  
  
 public BinTreeNode(T value, BinTreeNode<T> left, BinTreeNode<T> right)  
 {  
 data = value;  
 this.left = left;  
 this.right = right;  
 }  
  
 public BinTreeNode()  
 {  
 this(null, null, null);  
 }  
}

*/////////////////////////////////////////////////  
// Filename: BinTree.java  
// Description: The class of binary tree  
// Author: Jian Yin  
/////////////////////////////////////////////////*import java.util.Vector;  
  
public class BinTree <T>  
{  
 public BinTreeNode<T> root;  
  
 *// 创建一棵空二叉树* public BinTree()  
 {  
 root = null;  
 }  
  
 *// 判断是否空树* public boolean isEmpty()  
 {  
 return root == null;  
 }  
  
 *// 创建根结点* public boolean insertRoot(T data)  
 {  
 if(!isEmpty())  
 return false;  
 else  
 {  
 root = new BinTreeNode<>(data);  
 return true;  
 }  
 }  
  
 *// 返回以 p 结点为根的子树的结点个数* public int count(BinTreeNode<T> p)  
 {  
 if(p == null)  
 return 0;  
 else  
 {  
 return 1+count(p.left)+count(p.right);  
 }  
 }  
  
 *// 返回二叉树的结点个数* public int count()  
 {  
 return count(root);  
 }  
  
 *// 返回以 p 结点为根的子树高度* public int height(BinTreeNode<T> p)  
 {  
 if (p == null)  
 return 0;  
 int lh = height(p.left); *//返回左子树的高度* int rh = height(p.right); *//返回右子树的高度* return (lh>=rh) ? lh+1 : rh+1; *//当前子树高度为较高子树的高度加 1* }  
  
 *// 返回二叉树的高度* public int height()  
 {  
 return height(root);  
 }  
  
 *// 在以p为根的子树中查找并返回首次出现的关键字为 key 的元素结点* public BinTreeNode<T> searchNode(BinTreeNode<T> p, T key)  
 {  
 if (p==null || key==null)  
 return null;  
 if (p.data.equals(key))  
 return p; *//查找成功,返回找到结点* BinTreeNode<T> find=searchNode(p.left, key); *//在左子树中查找,递归调用* if (find==null) *//若在左子树中未找到* find=searchNode(p.right, key); *//则继续在右子树中查找,递归调用* return find; *//返回查找结果* }  
  
 *// 在指定的结点key中插入左孩子* public boolean insertLeft(T key, T e)  
 {  
 BinTreeNode<T> p = searchNode(root, key);  
 if(p==null || e==null || p.left!=null)  
 return false;  
 p.left = new BinTreeNode<>(e);  
 return true;  
 }  
  
 *// 在指定的结点key中插入右孩子* public boolean insertRight(T key, T e)  
 {  
 BinTreeNode<T> p = searchNode(root, key);  
 if(p==null || e==null || p.right!=null)  
 return false;  
 p.right = new BinTreeNode<>(e);  
 return true;  
 }  
  
 public void preOrderTraverse(BinTreeNode<T> p, Vector<T> vec)  
 {  
 if(p!=null)  
 {  
 vec.add(p.data);  
 preOrderTraverse(p.left, vec);  
 preOrderTraverse(p.right, vec);  
 }  
 }  
  
 public void preOrderTraverse(Vector<T> vec)  
 {  
 preOrderTraverse(root, vec);  
 }  
  
 public static void main(String[] args)  
 {  
 */\*  
 a  
 / \  
 b c  
 / \ \  
 d e f  
 / \ /  
 g h i  
 \*/* BinTree<Character> binTree = new BinTree<>();  
 binTree.insertRoot('a');  
 binTree.insertLeft('a', 'b');  
 binTree.insertRight('a', 'c');  
 binTree.insertLeft('b', 'd');  
 binTree.insertRight('b', 'e');  
 binTree.insertRight('c', 'f');  
 binTree.insertLeft('e', 'g');  
 binTree.insertRight('e', 'h');  
 binTree.insertLeft('f', 'i');  
  
 Vector<Character> vec = new Vector<>();*//保存遍历结果向量* binTree.preOrderTraverse(vec);*//先序遍历，结果保存在vec中* for(final Character v: vec)  
 System.*out*.print(v+" ");  
 vec.clear();  
 }  
}

## 哈夫曼树

*// 已知一个电文字符集中有8个字符{A，B，C，D，E，F，G，H}，  
// 它们使用的频率为{0.04, 0.21, 0.06, 0.07, 0.15, 0.18, 0.12, 0.03}，  
// 请用java设计一个哈夫曼编码程序。  
// 哈夫曼树的每个分支左分支设为0，右分支设为1，  
// 要求同层中叶子结点权值从左到右，从小到大*import java.util.\*;  
  
*// 哈夫曼树节点类*class HuffmanNode implements Comparable<HuffmanNode> {  
 char character;  
 double frequency;  
 HuffmanNode left;  
 HuffmanNode right;  
  
 public HuffmanNode(char character, double frequency) {  
 this.character = character;  
 this.frequency = frequency;  
 this.left = null;  
 this.right = null;  
 }  
  
 public HuffmanNode(double frequency, HuffmanNode left, HuffmanNode right) {  
 this.character = '\0';  
 this.frequency = frequency;  
 this.left = left;  
 this.right = right;  
 }  
  
 public boolean isLeaf() {  
 return left == null && right == null;  
 }  
  
 @Override  
 public int compareTo(HuffmanNode other) {  
 *// 首先比较频率，频率相同则比较字符（确保同层节点从左到右，从小到大）* if (this.frequency != other.frequency) {  
 return Double.*compare*(this.frequency, other.frequency);  
 } else {  
 return Character.*compare*(this.character, other.character);  
 }  
 }  
}  
  
public class HuffmanCoding {  
  
 *// 构建哈夫曼树* public static HuffmanNode buildHuffmanTree(Map<Character, Double> frequencyMap) {  
 PriorityQueue<HuffmanNode> priorityQueue = new PriorityQueue<>();  
  
 *// 为每个字符创建叶子节点并加入优先队列* for (Map.Entry<Character, Double> entry : frequencyMap.entrySet()) {  
 priorityQueue.add(new HuffmanNode(entry.getKey(), entry.getValue()));  
 }  
  
 *// 构建哈夫曼树* while (priorityQueue.size() > 1) {  
 *// 取出两个频率最小的节点* HuffmanNode left = priorityQueue.poll();  
 HuffmanNode right = priorityQueue.poll();  
  
 *// 创建新节点，频率为两个子节点频率之和* HuffmanNode parent = new HuffmanNode(left.frequency + right.frequency, left, right);  
  
 *// 将新节点加入队列* priorityQueue.add(parent);  
 }  
  
 *// 返回根节点* return priorityQueue.poll();  
 }  
  
 *// 生成哈夫曼编码表* public static Map<Character, String> generateCodes(HuffmanNode root) {  
 Map<Character, String> codeMap = new HashMap<>();  
 *buildCodeMap*(root, "", codeMap);  
 return codeMap;  
 }  
  
 private static void buildCodeMap(HuffmanNode node, String code, Map<Character, String> codeMap) {  
 if (node == null) return;  
  
 if (node.isLeaf()) {  
 codeMap.put(node.character, code);  
 return;  
 }  
  
 *// 左分支为0，右分支为1  
 buildCodeMap*(node.left, code + "0", codeMap);  
 *buildCodeMap*(node.right, code + "1", codeMap);  
 }  
  
 *// 打印哈夫曼编码表* public static void printHuffmanCodes(Map<Character, String> codeMap) {  
 System.*out*.println("字符\t频率\t哈夫曼编码");  
 List<Map.Entry<Character, String>> entries = new ArrayList<>(codeMap.entrySet());  
  
 *// 按字符顺序排序输出* entries.sort(Map.Entry.*comparingByKey*());  
  
 for (Map.Entry<Character, String> entry : entries) {  
 System.*out*.printf("%c\t%.2f\t%s\n", entry.getKey(), *getFrequency*(entry.getKey()), entry.getValue());  
 }  
 }  
  
 *// 辅助方法：获取字符频率（仅用于打印）* private static double getFrequency(char c) {  
 Map<Character, Double> frequencyMap = *getFrequencyMap*();  
 return frequencyMap.get(c);  
 }  
  
 *// 获取字符频率映射* private static Map<Character, Double> getFrequencyMap() {  
 Map<Character, Double> frequencyMap = new HashMap<>();  
 frequencyMap.put('A', 0.04);  
 frequencyMap.put('B', 0.21);  
 frequencyMap.put('C', 0.06);  
 frequencyMap.put('D', 0.07);  
 frequencyMap.put('E', 0.15);  
 frequencyMap.put('F', 0.18);  
 frequencyMap.put('G', 0.12);  
 frequencyMap.put('H', 0.03);  
 return frequencyMap;  
 }  
  
 public static void main(String[] args) {  
 *// 获取字符频率映射* Map<Character, Double> frequencyMap = *getFrequencyMap*();  
  
 *// 构建哈夫曼树* HuffmanNode root = *buildHuffmanTree*(frequencyMap);  
  
 *// 生成哈夫曼编码* Map<Character, String> huffmanCodes = *generateCodes*(root);  
  
 *// 打印哈夫曼编码表  
 printHuffmanCodes*(huffmanCodes);  
 }  
}