数据结构的 Java 表示

1．链表 2
1.1 线性链表 2
1.2 链式链表 4
1.3 双端链表 8
2.栈 10
2.1 线性栈 10
2.2 链式栈 12
3.队列 14
3.1 线性单端队列 14
3.2 链式单端队列 16
3.3 线性循环队列 19
3.4 线性双端队列 22
4.二叉树（二叉链表的实现） 26
5.图(邻接链表表示法)和最短路径 Dijkstra 算法 30
6.排序算法 36
7.常见考题（更新中…） 41
7.1 单链表反转 41

详情请访问个人 github 主页: https://github.com/HuiT2015/MyDocuments

1. 链表

1.1 线性链表

```
（1）  package com.ex.list;
（2）
（3）  import java.io.BufferedReader;
（4）  import java.io.FileNotFoundException;
（5）  import java.io.FileReader;
（6）  import java.io.IOException;
（7）
（8）  public class MySeqList<M> {
（9）
（10）   // 设置线性链表的最大容量
（11）   private static final int MAX_SIZE = 100;
（12）
（13）   // 创建线性链表的数据结构
（14）   private static class SeqList<T> {
（15）         T[] list = (T[]) new Object[MAX_SIZE];
（16）         int length;
（17）   }
（18）
（19）   // 定义一个线性链表
（20）   private SeqList<M> seqList;
（21）
（22）   // 初始化线性链表
（23）   private void initSeqList() {
（24）         if (seqList == null) {
（25）               seqList = new SeqList<M>();
（26）         }
（27）   }
（28）
（29）   // 获取位置为 i 的链表节点
（30）   private M getElem(int i) {
（31）         if (i < 1 || i > seqList.length)
（32）               return null;
（33）         return seqList.list[i - 1];
（34）   }
（35）
```

```
（36）    // 在地 i 个位置插入节点 e
（37）    private void insert(int i, M e) {
（38）        if (i < 1 || i > seqList.length + 1) {
（39）            System.out.println("插入位置非法！");
（40）            return;
（41）        } else if (seqList.length >= MAX_SIZE) {
（42）            System.out.println("链表已满员，无法进行插入操作！");
（43）            seqList.list = (M[]) new Object[2 * MAX_SIZE];
（44）            return;
（45）        } else {
（46）            for (int j = seqList.length; j >= i; j++) {
（47）                seqList.list[j] = seqList.list[j - 1];
（48）            }
（49）            seqList.list[i - 1] = e;
（50）            seqList.length += 1;
（51）        }
（52）    }
（53）
（54）    // 删除第 i 个节点的值
（55）    private M deleteAt(int i) {
（56）        M result = null;
（57）        if (i < 1 || i > seqList.length) {
（58）            System.out.println("删除位置非法！");
（59）            return result;
（60）        }
（61）        result = seqList.list[i - 1];
（62）        for (int j = i - 1; j < seqList.length - 1; j++) {
（63）            seqList.list[j] = seqList.list[j + 1];
（64）        }
（65）        return result;
（66）    }
（67）
（68）    // 从文件中创建链表
（69）    private void createSeqList() {
（70）        try {
（71）            BufferedReader bufferedReader = new BufferedReader(new
    FileReader("list.txt"));
（72）            try {
（73）                String[] strArr = bufferedReader.readLine().split(" ");
（74）                int nodeNum = Integer.parseInt(strArr[0]);
（75）                for (int i = 0; i < nodeNum; i++) {
（76）                    Character elem = Character.valueOf(strArr[i + 1].charAt(0));
（77）                    insert(i + 1, (M) elem);
（78）                }
```

```
（79）                    bufferedReader.close();
（80）                } catch (IOException e) {
（81）                    // TODO Auto-generated catch block
（82）                    e.printStackTrace();
（83）                }
（84）            } catch (FileNotFoundException e) {
（85）                // TODO Auto-generated catch block
（86）                e.printStackTrace();
（87）            }
（88）    }
（89）
（90）    //  打印线性链表
（91）    private void printSeqList() {
（92）        for (int i = 0; i < seqList.length; i++) {
（93）            System.out.print(seqList.list[i] + " ");
（94）        }
（95）    }
（96）
（97）    /**
（98）     * @param args
（99）     */
（100）        public static void main(String[] args) {
（101）            // TODO Auto-generated method stub
（102）            MySeqList<Character> mySeqList = new MySeqList<Character>();
（103）            mySeqList.initSeqList();
（104）            System.out.println("创建线性链表... ");
（105）            mySeqList.createSeqList();
（106）            System.out.println("打印线性链表...");
（107）            mySeqList.printSeqList();
（108）        }
（109）
（110）    }
```

1.2 链式链表

```
（1）    package com.ex.list;
（2）    import java.io.BufferedReader;
（3）    import java.io.FileNotFoundException;
（4）    import java.io.FileReader;
（5）    import java.io.IOException;
（6）    import java.util.Scanner;
（7）
（8）    public class MyLinkedList<M> {
（9）
```

```
（10）  // 定义单链表的数据结构
（11）  private static class LinkedNode<T> {
（12）      T data;
（13）      LinkedNode<T> next;
（14）
（15）      public LinkedNode() {
（16）          this(null, null);
（17）      }
（18）
（19）      public LinkedNode(T data) {
（20）          this(data, null);
（21）      }
（22）
（23）      public LinkedNode(T data, LinkedNode<T> next) {
（24）          this.data = data;
（25）          this.next = next;
（26）      }
（27）  }
（28）
（29）  // 定义一个单链表的头指针
（30）  private LinkedNode<M> head;
（31）
（32）  /**
（33）   * 初始化链表
（34）   */
（35）  private void initList() {
（36）      head = null;
（37）  }
（38）
（39）  /**
（40）   * 创建链表
（41）   */
（42）  private void createLinkedList() {
（43）      if (head == null) {
（44）          head = new LinkedNode<M>();
（45）      }
（46）      BufferedReader bufferedReader;
（47）      String[] strArr = null;
（48）      try {
（49）          bufferedReader = new BufferedReader(new FileReader("list.txt"));
（50）          try {
（51）              strArr = bufferedReader.readLine().split(" ");
（52）              bufferedReader.close();
（53）          } catch (IOException e) {
```

```
（54）                    // TODO Auto-generated catch block
（55）                    e.printStackTrace();
（56）                }
（57）            } catch (FileNotFoundException e) {
（58）                // TODO Auto-generated catch block
（59）                e.printStackTrace();
（60）            }
（61）            int nodeNum = Integer.parseInt(strArr[0]);
（62）            LinkedNode<M> newNode = null;
（63）            for (int i = 0; i < nodeNum; i++) {
（64）                Character inputCh = strArr[i + 1].charAt(0);
（65）                newNode = new LinkedNode<M>((M) inputCh);
（66）                newNode.next = head.next;
（67）                head.next = newNode;
（68）            }
（69）        }
（70）
（71）        /**
（72）         * 打印链表
（73）         */
（74）        private void printLinkedList() {
（75）            LinkedNode<M> p = head.next;
（76）            while (p != null) {
（77）                System.out.print(p.data + "\t");
（78）                p = p.next;
（79）            }
（80）        }
（81）
（82）        /**
（83）         * n 指第几个节点（1,2,3...),nFlag={-1,0,1},分别返回前、中、后节点
（84）         *
（85）         * @param n
（86）         * @param nFlag
（87）         * @return
（88）         */
（89）        private LinkedNode<M> locateNode(int n, int nFlag) {
（90）            LinkedNode<M> p = head;
（91）            int nCnt = 0;
（92）            while (p.next != null && nCnt < n + nFlag) {
（93）                p = p.next;
（94）                nCnt++;
（95）            }
（96）            return p;
（97）        }
```

```
（98）
（99）    /**
（100）         * 插入节点操作
（101）         *
（102）         * @param e
（103）         * @param i
（104）         */
（105）        private void insertNode(M e, int i) {
（106）              LinkedNode<M> pre = locateNode(i, -1);
（107）              LinkedNode<M> newNode = new LinkedNode<M>(e);
（108）              newNode.next = pre.next;
（109）              pre.next = newNode;
（110）        }
（111）
（112）        /**
（113）         * 删除节点操作
（114）         *
（115）         * @param i
（116）         */
（117）        private void eraseNode(int i) {
（118）              LinkedNode<M> pre = locateNode(i, -1);
（119）              LinkedNode<M> cur = locateNode(i, 0);
（120）              pre.next = cur.next;
（121）              cur.next = null;
（122）              cur.data = null;
（123）              cur = null;
（124）        }
（125）
（126）        /**
（127）         * 主函数
（128）         *
（129）         * @param args
（130）         */
（131）        public static void main(String[] args) {
（132）              MyLinkedList<Character>          myLinkedList          =          new
        MyLinkedList<Character>();
（133）              myLinkedList.initList();
（134）              System.out.println("建立单链表！");
（135）              myLinkedList.createLinkedList();
（136）              System.out.println("打印单链表！");
（137）              myLinkedList.printLinkedList();
（138）        }
（139）  }
```

1.3 双端链表

```
（1）  package com.ex.list;
（2）  public class DBLinkedList<T> {
（3）
（4）   /**
（5）    * 定义双端链表的数据结构
（6）    *
（7）    * @author lenovo
（8）    * @param <M>
（9）    */
（10）  static final class DBLinkNode<M> {
（11）        M data;
（12）        DBLinkNode<M> prior;
（13）        DBLinkNode<M> next;
（14）
（15）        public DBLinkNode(M data, DBLinkNode<M> prior, DBLinkNode<M> next) {
（16）              this.data = data;
（17）              this.prior = prior;
（18）              this.next = next;
（19）        }
（20）
（21）  }
（22）
（23）  // 定义一个双端链表头指针
（24）  private DBLinkNode<T> head;
（25）
（26）  /**
（27）   * 双端链表初始化函数
（28）   */
（29）  private void init() {
（30）        head = new DBLinkNode<T>(null, null, null);
（31）        head.prior = head;
（32）        head.next = head;
（33）  }
（34）
（35）  /**
（36）   * 在指定位置插入元素
（37）   *
（38）   * @param e
（39）   * @param nPos
（40）   */
（41）  private void insertElement(T e, int nPos) {
（42）        DBLinkNode<T> cur = getLocation(nPos);
```

```
（43）          DBLinkNode<T> newNode = new DBLinkNode<T>(e, cur.prior, cur);
（44）          cur.prior.next = newNode;
（45）          cur.prior = newNode;
（46）      }
（47）
（48）      /**
（49）       * 获取指定位置的元素
（50）       *
（51）       * @param nPos
（52）       * @return
（53）       */
（54）      private DBLinkNode<T> getLocation(int nPos) {
（55）          int i = 0;
（56）          DBLinkNode<T> cur = head.next;
（57）          while (i < nPos && cur != head) {
（58）              cur = cur.next;
（59）              i++;
（60）          }
（61）          return cur;
（62）      }
（63）
（64）      /**
（65）       * 删除指定位置的元素，并返回它
（66）       *
（67）       * @param nPos
（68）       * @return
（69）       */
（70）      private T deleteElement(int nPos) {
（71）          DBLinkNode<T> cur = getLocation(nPos);
（72）          cur.next.prior = cur.prior;
（73）          cur.prior.next = cur.next;
（74）          return cur.data;
（75）      }
（76）
（77）      /**
（78）       * 打印双端链表元素
（79）       */
（80）      private void printDBLinkedList() {
（81）          DBLinkNode<T> cur = head.next;
（82）          int i = 0;
（83）          while (cur != head) {
（84）              System.out.println("第" + (++i) + "个元素为:" + cur.data);
（85）              cur = cur.next;
（86）          }
```

```
（87）    }
（88）
（89）    /**
（90）     * 主函数
（91）     *
（92）     * @param args
（93）     */
（94）    public static void main(String[] args) {
（95）        // TODO Auto-generated method stub
（96）        DBLinkedList<Integer> dLinkedList = new DBLinkedList<Integer>();
（97）        dLinkedList.init();
（98）        for (int i = 0; i < 10; i++) {
（99）            dLinkedList.insertElement(2 * i + 1, i);
（100）           }
（101）           System.out.println("初始化时双端链表中的元素为:");
（102）           dLinkedList.printDBLinkedList();
（103）           System.out.println("删除第 4 个元素" + dLinkedList.deleteElement(3) + "
        后的双端链表为:");
（104）           dLinkedList.printDBLinkedList();
（105）       }
（106）   }
```

2.栈

2.1 线性栈

```
(1)  package com.ex.stack;
(2)  import java.io.BufferedReader;
(3)  import java.io.FileNotFoundException;
(4)  import java.io.FileReader;
(5)  import java.io.IOException;
(6)
(7)  public class MySeqStack<M> {
(8)
(9)      // 定义一个线性栈的容量常值
(10)     private static int STACK_SIZE = 100;
(11)
(12)     // 定义线性栈的数据结构
(13)     private static class SeqStack<T> {
(14)         T[] stack = (T[]) new Object[STACK_SIZE];
(15)         int top;
(16)     }
```

```
(17)
(18)    // 定义一个线性栈
(19)    private SeqStack<M> seqStack;
(20)
(21)    // 初始化线性栈
(22)    private void initSeqStack() {
(23)        if (seqStack == null) {
(24)            seqStack = new SeqStack<M>();
(25)        }
(26)    }
(27)
(28)    /**
(29)     * 进栈操作
(30)     *
(31)     * @param e
(32)     */
(33)    private void push(M e) {
(34)        if (seqStack.top == STACK_SIZE) {
(35)            System.out.println("线性栈已满员，需要扩容!");
(36)            STACK_SIZE *= 2;
(37)            seqStack.stack = (M[]) new Object[STACK_SIZE];
(38)        }
(39)        seqStack.stack[seqStack.top++] = e;
(40)    }
(41)
(42)    /**
(43)     * 栈顶元素出栈
(44)     *
(45)     * @return
(46)     */
(47)    private M pop() {
(48)        return seqStack.stack[--seqStack.top];
(49)    }
(50)
(51)    /**
(52)     * 主函数
(53)     *
(54)     * @param args
(55)     */
(56)    public static void main(String[] args) {
(57)        MySeqStack<Integer> mySeqStack = new MySeqStack<Integer>();
(58)        System.out.println("建立线性栈...");
(59)        mySeqStack.initSeqStack();
(60)        int stackCapacity = -1;
```

```
(61)            try {
(62)                BufferedReader bufferedReader = new BufferedReader(new
    FileReader("stack.txt"));
(63)                try {
(64)                    String[] strArr = bufferedReader.readLine().split(" ");
(65)                    stackCapacity = Integer.parseInt(strArr[0]);
(66)                    for (int i = 0; i < stackCapacity; i++) {
(67)                        mySeqStack.push(Integer.parseInt(strArr[i + 1]));
(68)                    }
(69)                    bufferedReader.close();
(70)                } catch (IOException e) {
(71)                    // TODO Auto-generated catch block
(72)                    e.printStackTrace();
(73)                }
(74)
(75)            } catch (FileNotFoundException e) {
(76)                // TODO Auto-generated catch block
(77)                e.printStackTrace();
(78)            }
(79)            System.out.println("打印线性栈...");
(80)            for (int i = 0; i < stackCapacity; i++) {
(81)                System.out.print(mySeqStack.pop() + " ");
(82)            }
(83)    }
(84)}
```

2.2 链式栈

```
（1）  package com.ex.stack;
（1）
（2）  public class LinkedStack<T> {
（3）
（4）   /**
（5）    * 定义一个链式栈的数据结构
（6）    * @author lenovo
（7）    *
（8）    * @param <T>
（9）    */
（10）  static class LinkedNode<T>
（11）  {
（12）      T data;
（13）      LinkedNode<T> next;
（14）      public LinkedNode(T data,LinkedNode<T> next) {
（15）          this.data = data;
```

```
（16）                this.next = next;
（17）          }
（18）
（19）    }
（20）
（21）    //顶一个栈顶指针
（22）    public LinkedNode<T> top=null;
（23）
（24）    /**
（25）     * 进栈
（26）     * @param e
（27）     */
（28）    public void push(T e) {
（29）          if (top==null) {
（30）                top=new LinkedNode<T>(null, null);
（31）          }
（32）          LinkedNode<T> newNode=new LinkedNode<T>(e, top.next);
（33）          top.next=newNode;
（34）    }
（35）
（36）    /**
（37）     * 出栈
（38）     * @return
（39）     */
（40）    public T pop() {
（41）          T result=null;
（42）          LinkedNode<T> p=top.next;
（43）          if (p!=null) {
（44）                top.next=p.next;
（45）                result=p.data;
（46）          }
（47）          return result;
（48）    }
（49）
（50）    /**
（51）     * 返回栈顶元素而不删除
（52）     * @return
（53）     */
（54）    public T peekTop() {
（55）          return top.next.data;
（56）    }
（57）
（58）    /**
（59）     * 清空栈中的元素
```

```
（60）     */
（61）    public void clear() {
（62）          LinkedNode<T> p=top;
（63）          LinkedNode<T> q=null;
（64）          while (p!=null) {
（65）                q=p;
（66）                p.data=null;
（67）                p.next=null;
（68）                p=null;
（69）                p=q.next;
（70）          }
（71）    }
（72）
（73）    /**
（74）     *  判断栈是否为空
（75）     * @return
（76）     */
（77）    public boolean isEmpty() {
（78）          return top.next==null;
（79）    }
（80）  }
```

3.队列

3.1 线性单端队列

```
（1）    package com.ex.queue;
（2）
（3）    import java.io.BufferedReader;
（4）    import java.io.FileNotFoundException;
（5）    import java.io.FileReader;
（6）    import java.io.IOException;
（7）
（8）    public class MySeqQueue<M> {
（9）
（10）    // 定义队列的容量大小
（11）    private static final int QUEUE_SIZE = 100;
（12）
（13）    // 定义单端队列的数据结构
（14）    private static class SeqQueue<T> {
（15）          T[] queue = (T[]) new Object[QUEUE_SIZE];
（16）          int front, rear;
```

```
（17）  }
（18）
（19）  // 定义单端队列
（20）  private SeqQueue<M> seqQueue;
（21）
（22）  /**
（23）   * 初始化单端队列
（24）   */
（25）  private void initSeqQueue() {
（26）      if (seqQueue == null) {
（27）          seqQueue = new SeqQueue<M>();
（28）      }
（29）      seqQueue.front = seqQueue.rear = 0;
（30）  }
（31）
（32）  /**
（33）   * 向队尾添加元素
（34）   *
（35）   * @param e
（36）   */
（37）  private void enterQueue(M e) {
（38）      if (seqQueue.rear == QUEUE_SIZE) {
（39）          System.out.println("队列满员！");
（40）          return;
（41）      }
（42）      seqQueue.queue[seqQueue.rear++] = e;
（43）  }
（44）
（45）  /**
（46）   * 从对头删除元素
（47）   *
（48）   * @return
（49）   */
（50）  private M deleteQueue() {
（51）      if (seqQueue.front == seqQueue.rear) {
（52）          System.out.println("队列已经为空！");
（53）          return null;
（54）      }
（55）      return seqQueue.queue[seqQueue.front++];
（56）  }
（57）
（58）  /**
（59）   * 从文件中创建队列
（60）   */
```

```
（61）    private void createSeqQueue() {
（62）        BufferedReader bufferedReader;
（63）        String[] strArr = null;
（64）        try {
（65）            bufferedReader = new BufferedReader(new FileReader("queue.txt"));
（66）            try {
（67）                strArr = bufferedReader.readLine().split(" ");
（68）                bufferedReader.close();
（69）            } catch (IOException e) {
（70）                // TODO Auto-generated catch block
（71）                e.printStackTrace();
（72）            }
（73）        } catch (FileNotFoundException e) {
（74）            // TODO Auto-generated catch block
（75）            e.printStackTrace();
（76）        }
（77）        int queueSize = Integer.parseInt(strArr[0]);
（78）        for (int i = 0; i < queueSize; i++) {
（79）            Object elem = (Object) Integer.parseInt(strArr[i + 1]);
（80）            enterQueue((M) elem);
（81）        }
（82）    }
（83）
（84）    /**
（85）     * 主函数
（86）     *
（87）     * @param args
（88）     */
（89）    public static void main(String[] args) {
（90）        MySeqQueue<Integer> mySeqQueue = new MySeqQueue<Integer>();
（91）        mySeqQueue.initSeqQueue();
（92）        System.out.println("建立线性队列！");
（93）        mySeqQueue.createSeqQueue();
（94）        System.out.println("打印线性队列！");
（95）        for (int i = 0; i < mySeqQueue.seqQueue.rear; i++) {
（96）            System.out.print(mySeqQueue.deleteQueue() + " ");
（97）        }
（98）    }
（99）}
```

3.2 链式单端队列

```
（1）    package com.ex.queue;
```

```
（2）
（3）  import java.io.BufferedReader;
（4）  import java.io.FileNotFoundException;
（5）  import java.io.FileReader;
（6）  import java.io.IOException;
（7）
（8）  public class MyLinkedQueue<M> {
（9）
（10）   /**
（11）    * 定义链式队列结点数据结构
（12）    *
（13）    * @author lenovo
（14）    *
（15）    * @param <T>
（16）    */
（17）   private final static class QueueNode<T> {
（18）        T data;
（19）        QueueNode<T> next;
（20）
（21）        public QueueNode() {
（22）             this(null, null);
（23）        }
（24）
（25）        public QueueNode(T data) {
（26）             this(data, null);
（27）        }
（28）
（29）        public QueueNode(T data, QueueNode<T> next) {
（30）             this.data = data;
（31）             this.next = next;
（32）        }
（33）   }
（34）
（35）   /**
（36）    * 定义链式队列数据结构
（37）    *
（38）    * @author lenovo
（39）    *
（40）    * @param <M>
（41）    */
（42）   private final static class LinkQueue<M> {
（43）        QueueNode<M> front;
（44）        QueueNode<M> rear;
（45）   }
```

```
（46）
（47）  /**
（48）   * 定义链式队列
（49）   */
（50）  private LinkQueue<Integer> LQ;
（51）
（52）  /**
（53）   * 初始化队列操作
（54）   */
（55）  private void initLinkedQueue() {
（56）      if (LQ == null) {
（57）          LQ = new LinkQueue<Integer>();
（58）      }
（59）      LQ.front = LQ.rear = new QueueNode<Integer>();
（60）  }
（61）
（62）  /**
（63）   * 入队列操作
（64）   *
（65）   * @param elem
（66）   */
（67）  private void enterQueue(int elem) {
（68）      QueueNode<Integer> newNode = new QueueNode<Integer>(elem);
（69）      LQ.rear.next = newNode;
（70）      LQ.rear = newNode;
（71）  }
（72）
（73）  /**
（74）   * 出队列操作
（75）   *
（76）   * @return
（77）   */
（78）  private int deleteQueue() {
（79）      if (LQ.front == LQ.rear) {
（80）          System.out.println("队列为空！");
（81）          return -1;
（82）      }
（83）      QueueNode<Integer> newNode = LQ.front.next;
（84）      int result = newNode.data;
（85）      LQ.front.next = newNode.next;
（86）      if (LQ.rear == newNode) {
（87）          LQ.rear = LQ.front;
（88）      }
（89）      return result;
```

```
（90）    }
（91）
（92）    /**
（93）     * 测试
（94）     *
（95）     * @param args
（96）     */
（97）    public static void main(String[] args) {
（98）        MyLinkedQueue<Integer> myLinkedQueuey = new MyLinkedQueue<Integer>();
（99）        System.out.println("建立链式队列...");
（100）            myLinkedQueuey.initLinkedQueue();
（101）            int queueSize = -1;
（102）            try {
（103）                BufferedReader bufferedReader = new BufferedReader(new FileReader("queue.txt"));
（104）                try {
（105）                    String[] strArr = bufferedReader.readLine().split(" ");
（106）                    queueSize = Integer.parseInt(strArr[0]);
（107）                    for (int i = 0; i < queueSize; i++) {
（108）                        myLinkedQueuey.enterQueue(Integer.parseInt(strArr[i + 1]));
（109）                    }
（110）                    bufferedReader.close();
（111）                } catch (IOException e) {
（112）                    // TODO Auto-generated catch block
（113）                    e.printStackTrace();
（114）                }
（115）
（116）            } catch (FileNotFoundException e) {
（117）                // TODO Auto-generated catch block
（118）                e.printStackTrace();
（119）            }
（120）            System.out.println("打印链式队列...");
（121）            while (myLinkedQueuey.LQ.front != myLinkedQueuey.LQ.rear) {
（122）                System.out.print(myLinkedQueuey.deleteQueue() + " ");
（123）            }
（124）        }
（125）    }
```

3.3 线性循环队列

（1） package com.ex.queue;

```
（2）  public class MySCQueue<T> {
（3）
（4）   // 顺序循环队列大小，可以自动扩容
（5）   private static int nQueueSize = 10;
（6）
（7）   /**
（8）    * 定义顺序循环队列的数据结构
（9）    *
（10）   * @author lenovo
（11）   *
（12）   * @param <M>
（13）   */
（14）  private static final class SCQueue<M> {
（15）      int front;
（16）      int rear;
（17）      M[] queue;
（18）
（19）      public SCQueue() {
（20）          // TODO Auto-generated constructor stub
（21）          front = rear = 0;
（22）          queue = (M[]) new Object[nQueueSize];
（23）      }
（24）  }
（25）
（26）  /**
（27）   * 定义一个顺序循环队列的对象
（28）   */
（29）  private SCQueue<T> SQ;
（30）
（31）  /**
（32）   * 初始化顺序循环队列 SQ
（33）   */
（34）  private void initQueue() {
（35）      SQ = new SCQueue<T>();
（36）      SQ.rear = SQ.front = 0;
（37）  }
（38）
（39）  /**
（40）   * 向顺序循环队列中插入元素，如果队列已满，队列将会自动扩容两倍
（41）   *
（42）   * @param e
（43）   */
（44）  private void EnterQueue(T e) {
（45）      if (SQ.front == (SQ.rear + 1) % nQueueSize) {
```

```
（46）              System.out.println("当前队列已满，队列将扩容为原来的两倍");
（47）              enlargeQueue();
（48）          }
（49）          SQ.queue[SQ.rear] = e;
（50）          SQ.rear = (SQ.rear + 1) % nQueueSize;
（51）      }
（52）
（53）      /**
（54）       * 扩大队列的容量为原来的两倍
（55）       */
（56）      private void enlargeQueue() {
（57）          T[] tmpQueue = (T[]) new Object[(SQ.rear + nQueueSize - SQ.front) %
        nQueueSize];
（58）          for (int i = 0; i < tmpQueue.length; i++) {
（59）              tmpQueue[i] = DeleteQueue();
（60）          }
（61）          SQ.front = SQ.rear = 0;
（62）          nQueueSize *= 2;
（63）          SQ.queue = (T[]) new Object[nQueueSize];
（64）          for (int i = 0; i < tmpQueue.length; i++) {
（65）              EnterQueue(tmpQueue[i]);
（66）          }
（67）      }
（68）
（69）      /**
（70）       * 删除队列的对头元素并返回对头元素
（71）       *
（72）       * @return
（73）       */
（74）      private T DeleteQueue() {
（75）          if (SQ.front == SQ.rear) {
（76）              System.out.println("队列中已经为空:");
（77）              return null;
（78）          }
（79）          T e = SQ.queue[SQ.front];
（80）          SQ.front = (SQ.front + 1) % nQueueSize;
（81）          return e;
（82）      }
（83）
（84）      /**
（85）       * 打印队列元素
（86）       */
（87）      private void printSCQueue() {
（88）          int i = SQ.front, j = 0;
```

```
（89）          while (i != SQ.rear) {
（90）              System.out.println("队列的第" + (++j) + "个的元素为:" + SQ.queue[i]);
（91）              i = (i + 1) % nQueueSize;
（92）          }
（93）      }
（94）
（95）      /**
（96）       * 主函数
（97）       *
（98）       * @param args
（99）       */
（100）         public static void main(String[] args) {
（101）             // TODO Auto-generated method stub
（102）             MySCQueue<Character> queue = new MySCQueue<Character>();
（103）             queue.initQueue();
（104）             for (int i = 0; i < 26; i++) {
（105）                 queue.EnterQueue((char) (i + 65));
（106）             }
（107）             System.out.println("出队列前队列中的元素为:");
（108）             queue.printSCQueue();
（109）             queue.DeleteQueue();
（110）             System.out.println("对头元素出队列后队列中的元素为:");
（111）             queue.printSCQueue();
（112）         }
（113）
（114）     }
```

3.4 线性双端队列

```
（1）   package com.ex.queue;
（2）
（3）   import java.util.Random;
（4）
（5）   public class MyDQueue<T> {
（6）
（7）    // 定义双端队列的大小
（8）    private static int nQueueSize = 100;
（9）
（10）   private static final class DQueue<M> {
（11）       int end1;
（12）       int end2;
（13）       M[] queue;
（14）   }
```

```
（15）
（16）  /**
（17）   * 定义一个枚举结构来表示左右队列
（18）   *
（19）   * @author lenovo
（20）   *
（21）   */
（22）  static enum QType {
（23）      Left {
（24）          void tellDirection(boolean nFlag) {
（25）              if (nFlag == true) {
（26）                  System.out.println("元素从左边入队列:");
（27）              } else {
（28）                  System.out.println("元素从左边出队列： ");
（29）              }
（30）
（31）          }
（32）      },
（33）      Right {
（34）          void tellDirection(boolean nFlag) {
（35）              if (nFlag == true) {
（36）                  System.out.println("元素从右边入队列:");
（37）              } else {
（38）                  System.out.println("元素从右边出队列： ");
（39）              }
（40）
（41）          }
（42）      };
（43）      abstract void tellDirection(boolean nFlag);
（44）  }
（45）
（46）  // 定义一个双端队列对象
（47）  private DQueue<T> DQ;
（48）
（49）  private void initDQ() {
（50）      DQ = new DQueue<T>();
（51）      DQ.end2 = nQueueSize / 2;
（52）      DQ.end1 = DQ.end2 - 1;
（53）      DQ.queue = (T[]) new Object[nQueueSize];
（54）  }
（55）
（56）  /**
（57）   * 元素入队
（58）   *
```

```
（59）     * @param e
（60）     * @param nFlag
（61）     */
（62）    private void enterQueue(T e, QType eFlag) {
（63）        eFlag.tellDirection(true);
（64）        switch (eFlag) {
（65）        case Left:
（66）            if (DQ.end1 != DQ.end2) {
（67）                DQ.queue[DQ.end1] = e;
（68）                DQ.end1 = (DQ.end1 - 1) % nQueueSize;
（69）            }
（70）            break;
（71）        case Right:
（72）            if (DQ.end1 != DQ.end2) {
（73）                DQ.queue[DQ.end2] = e;
（74）                DQ.end2 = (DQ.end2 + 1) % nQueueSize;
（75）            }
（76）            break;
（77）        default:
（78）            System.out.println("非法进入队列！");
（79）            break;
（80）        }
（81）    }
（82）
（83）    /**
（84）     *  出队列
（85）     *
（86）     * @param nFlag
（87）     * @return
（88）     */
（89）    private T deleteQueue(QType eFlag) {
（90）        T ret = null;
（91）        eFlag.tellDirection(false);
（92）        switch (eFlag) {
（93）        case Left:
（94）            if (DQ.end1 + 1 != DQ.end2) {
（95）
（96）                DQ.end1 = (DQ.end1 + 1) % nQueueSize;
（97）                ret = DQ.queue[DQ.end1];
（98）            }
（99）            break;
（100）           case Right:
（101）               if (DQ.end1 + 1 != DQ.end2) {
（102）                   DQ.end2 = (DQ.end2 - 1) % nQueueSize;
```

```
（103）                    ret = DQ.queue[DQ.end2];
（104）                }
（105）                break;
（106）            default:
（107）                System.out.println("出队列非法！ ");
（108）                break;
（109）            }
（110）            return ret;
（111）        }
（112）
（113）        /**
（114）         * 打印双端队列
（115）         */
（116）        private void printDQ() {
（117）            int i = 0;
（118）            int nEnd1 = DQ.end1, nEnd2 = DQ.end2;
（119）            while (DQ.end1 + 1 != DQ.end2) {
（120）                DQ.end1 = (DQ.end1 + 1) % nQueueSize;
（121）                System.out.println("左边对列中的第" + (++i) + "个元素为:" +
    DQ.queue[DQ.end1]);
（122）                DQ.end2 = (DQ.end2 - 1) % nQueueSize;
（123）                System.out.println("右边对列中的第" + (++i) + "个元素为:" +
    DQ.queue[DQ.end2]);
（124）            }
（125）            DQ.end1 = nEnd1;
（126）            DQ.end2 = nEnd2;
（127）        }
（128）
（129）        /**
（130）         * 主函数
（131）         *
（132）         * @param args
（133）         */
（134）        public static void main(String[] args) {
（135）            // TODO Auto-generated method stub
（136）            MyDQueue<Double> dQueue = new MyDQueue<Double>();
（137）            dQueue.initDQ();
（138）            Random random = new Random();
（139）            for (int i = 0; i < 12; i++) {
（140）                dQueue.enterQueue(i * 1.0,
    QType.values()[random.nextInt(QType.values().length)]);
（141）            }
（142）            System.out.println("出队列前队列中的元素为:");
（143）            dQueue.printDQ();
```

```
（144）                System.out.println("左边队列对头元素出队列，右边队列队尾元素出
        队列后的队列为:");
（145）                dQueue.deleteQueue(QType.Left);
（146）                dQueue.deleteQueue(QType.Right);
（147）                dQueue.printDQ();
（148）        }
（149）
（150）  }
（151）
```

4.二叉树（二叉链表的实现）

```
（1）   package com.ex.tree;
（2）
（3）   import java.io.BufferedReader;
（4）   import java.io.FileNotFoundException;
（5）   import java.io.FileReader;
（6）   import java.io.IOException;
（7）   import java.util.Scanner;
（8）   import com.ex.graph.LinkedStack;
（9）
（10）  public class BinaryTree {
（11）
（12）   // 定义根节点
（13）   private BinaryNode<Character> root;
（14）   // 定义一个链式栈用来存储访问过的节点
（15）   private LinkedStack<BinaryNode<Character>> linkedStack = null;
（16）
（17）   /**
（18）    * 初始化
（19）    */
（20）   private void initBinaryTree() {
（21）         this.root = null;
（22）         linkedStack = new LinkedStack<BinaryNode<Character>>();
（23）   }
（24）
（25）   /**
（26）    * 创建二叉树
（27）    *
（28）    * @param treeNode
（29）    * @return
（30）    */
```

```
（31）   private BinaryNode<Character> createBinaryTree(BinaryNode<Character>
      treeNode, BufferedReader bufReader)
（32）               throws IOException {
（33）         Character inputCh = bufReader.readLine().charAt(0);
（34）         if (inputCh.equals('#'))
（35）               return null;
（36）         else {
（37）               if (treeNode == null)
（38）                     treeNode = new BinaryNode<Character>(inputCh);
（39）               treeNode.left = createBinaryTree(treeNode.left, bufReader);
（40）               treeNode.right = createBinaryTree(treeNode.right, bufReader);
（41）               return treeNode;
（42）         }
（43）   }
（44）
（45）   /**
（46）    * 插入左子树
（47）    *
（48）    * @param p
（49）    * @param newEle
（50）    */
（51）   private void insertLeftChild(BinaryNode<Character> p, Character newEle) {
（52）         if (p != null) {
（53）               BinaryNode<Character> newNode = new
      BinaryNode<Character>(newEle);
（54）               newNode.right = p.left;
（55）               p.left = newNode;
（56）         }
（57）   }
（58）
（59）   /**
（60）    * 插入右子树
（61）    *
（62）    * @param p
（63）    * @param newEle
（64）    */
（65）   private void insertRightChild(BinaryNode<Character> p, Character newEle) {
（66）         if (p != null) {
（67）               BinaryNode<Character> newNode = new
      BinaryNode<Character>(newEle);
（68）               newNode.right = p.right;
（69）               p.right = newNode;
（70）         }
（71）   }
```

```
（72）
（73）    /**
（74）     * 前序遍历
（75）     */
（76）    private void preOrderTraverse() {
（77）        linkedStack.clear();
（78）        BinaryNode<Character> p = root;
（79）        while (p != null || linkedStack.isEmpty() == false) {
（80）            while (p != null) {
（81）                System.out.print(p.element + " ");
（82）                linkedStack.push(p);
（83）                p = p.left;
（84）            }
（85）            if (linkedStack.isEmpty() != true) {
（86）                p = linkedStack.pop();
（87）                p = p.right;
（88）            }
（89）        }
（90）        System.out.print("\n");
（91）    }
（92）
（93）    /**
（94）     * 中序遍历
（95）     */
（96）    private void inOrderTraverse() {
（97）        linkedStack.clear();
（98）        BinaryNode<Character> p = root;
（99）        while (p != null || linkedStack.isEmpty() == false) {
（100）            while (p != null) {
（101）                linkedStack.push(p);
（102）                p = p.left;
（103）            }
（104）            if (linkedStack.isEmpty() != true) {
（105）                p = linkedStack.pop();
（106）                System.out.print(p.element + " ");
（107）                p = p.right;
（108）            }
（109）        }
（110）        System.out.print("\n");
（111）    }
（112）
（113）    /**
（114）     * 后序遍历
（115）     */
```

```
（116）        private void postOrderTraverse() {
（117）              linkedStack.clear();
（118）              BinaryNode<Character> p = root;
（119）              BinaryNode<Character> q = null;
（120）              while (p != null || linkedStack.isEmpty() == false) {
（121）                    while (p != null) {
（122）                          linkedStack.push(p);
（123）                          p = p.left;
（124）                    }
（125）                    if (linkedStack.isEmpty() != true) {
（126）                          p = linkedStack.peekTop();
（127）                          if (p.right == null || p.right == q) {
（128）                                System.out.print(p.element + " ");
（129）                                q = p;
（130）                                p = null;
（131）                                linkedStack.pop();
（132）                          } else {
（133）                                p = p.right;
（134）                          }
（135）                    }
（136）              }
（137）              System.out.print("\n");
（138）        }
（139）
（140）        /**
（141）          * @param args
（142）          */
（143）        public static void main(String[] args) {
（144）              // TODO Auto-generated method stub
（145）              BinaryTree binaryTree = new BinaryTree();
（146）              binaryTree.initBinaryTree();
（147）              System.out.println("创建二叉树...");
（148）              try {
（149）                    BufferedReader bufferedReader = new BufferedReader(new
      FileReader("tree.txt"));
（150）                    try {
（151）                          binaryTree.root =
      binaryTree.createBinaryTree(binaryTree.root, bufferedReader);
（152）                          bufferedReader.close();
（153）                    } catch (IOException e) {
（154）                          // TODO Auto-generated catch block
（155）                          e.printStackTrace();
（156）                    }
（157）
```

```
（158）            } catch (FileNotFoundException e) {
（159）                  // TODO Auto-generated catch block
（160）                  e.printStackTrace();
（161）            }
（162）            System.out.println("二叉树前序遍历为...");
（163）            binaryTree.preOrderTraverse();
（164）            System.out.println("二叉树中序遍历为...");
（165）            binaryTree.inOrderTraverse();
（166）            System.out.println("二叉树后序遍历为...");
（167）            binaryTree.postOrderTraverse();
（168）            System.out.println("二叉树示例完毕!");
（169）      }
（170）
（171） }
```

5.图(邻接链表表示法)和最短路径 Dijkstra 算法

```
（1）  package com.ex.graph;
（2）
（3）  /**
（4）   *
（5）   * @author lenovo  定义图的邻接链表数据结构实现
（6）   * @param <T>
（7）   */
（8）  public class AdjGraph<T> {
（9）   private static final int MAX_SIZE = 1000;
（10）  VNode<T>[] vertexNodes;
（11）  int vexNum, arcNum;
（12）  GraphKind graphKind;
（13）
（14）  @SuppressWarnings("unchecked")
（15）  public AdjGraph() {
（16）        vertexNodes = (VNode<T>[]) new VNode[MAX_SIZE];
（17）        vexNum = arcNum = 0;
（18）        graphKind = GraphKind.DG;
（19）  }
（20）
（21） }
（22）
（23） /**
（24）  * 定义弧节点
（25）  */
（26） class ArcNode {
```

```
（27）    int adjvex;
（28）    ArcNode nextArcNode;
（29）    int weight;
（30）
（31）    public ArcNode(int adjvex, ArcNode nextArcNode, int weight) {
（32）        this.adjvex = adjvex;
（33）        this.nextArcNode = nextArcNode;
（34）        this.weight = weight;
（35）    }
（36）  }
（37）
（38）  /**
（39）    * 定义顶点结点
（40）    */
（41）  class VNode<T> {
（42）    T data;
（43）    ArcNode firstArcNode;
（44）
（45）    boolean know;
（46）    int dist;
（47）    VNode<T> nearNode;
（48）  }
（49）
（50）  /**
（51）    * 定义图 类型
（52）    */
（53）  enum GraphKind {
（54）    DG, DN, UG, UN
（55）  }
（56）  package com.ex.graph;
（57）
（58）  import java.io.BufferedReader;
（59）  import java.io.File;
（60）  import java.io.FileInputStream;
（61）  import java.io.FileNotFoundException;
（62）  import java.io.IOException;
（63）  import java.io.InputStream;
（64）  import java.io.InputStreamReader;
（65）  import java.util.Scanner;
（66）
（67）  public class DGraph {
（68）
（69）    // 定义无穷大值作为路径不可达的标志
（70）    private static final int INFINITY = (int) 1e9;
```

```
（71）  // 定义一个图的对象
（72）  private AdjGraph<Character> graph;
（73）
（74）  /**
（75）   * 创建图
（76）   */
（77）  private void createGraph() {
（78）      if (graph == null)
（79）          graph = new AdjGraph<Character>();
（80）      graph.graphKind = GraphKind.DG;
（81）      BufferedReader bufferedReader = null;
（82）      try {
（83）          bufferedReader = new BufferedReader(
（84）                  new InputStreamReader(new FileInputStream(new
      File("data.txt").getAbsoluteFile())));
（85）          try {
（86）              if (bufferedReader.ready() == true) {
（87）                  String[] graphInfoArr = bufferedReader.readLine().split(" ");
（88）                  graph.vexNum = Integer.parseInt(graphInfoArr[0]);
（89）                  graph.arcNum = Integer.parseInt(graphInfoArr[1]);
（90）                  String[] vertexArr = bufferedReader.readLine().split(" ");
（91）                  for (int i = 0; i < graph.vexNum; i++) {
（92）                      graph.vertexNodes[i] = new VNode<Character>();
（93）                      graph.vertexNodes[i].data =
      Character.valueOf(vertexArr[i].charAt(0));
（94）                      graph.vertexNodes[i].firstArcNode = null;
（95）                      graph.vertexNodes[i].know = false;
（96）                      graph.vertexNodes[i].dist = INFINITY;
（97）                      graph.vertexNodes[i].nearNode = null;
（98）                  }
（99）                  Character ch1, ch2;
（100）                     ArcNode arcNode = null;
（101）                     for (int i = 0; i < graph.arcNum; i++) {
（102）                         String[] arcInfoArr =
      bufferedReader.readLine().split(" ");
（103）                         ch1 = Character.valueOf(arcInfoArr[0].charAt(0));
（104）                         ch2 = Character.valueOf(arcInfoArr[1].charAt(0));
（105）                         int u = locateVertex(ch1);
（106）                         int v = locateVertex(ch2);
（107）                         arcNode = new ArcNode(v,
      graph.vertexNodes[u].firstArcNode, Integer.parseInt(arcInfoArr[2]));
（108）                         graph.vertexNodes[u].firstArcNode = arcNode;
（109）                     }
（110）                 }
```

```
（111）                    bufferedReader.close();
（112）                } catch (IOException e) {
（113）                    // TODO: handle exception
（114）                    e.printStackTrace();
（115）                }
（116）            } catch (FileNotFoundException e) {
（117）                // TODO Auto-generated catch block
（118）                e.printStackTrace();
（119）            }
（120）        }
（121）
（122）        /**
（123）         * 获取某个节点在图中的位置
（124）         *
（125）         * @param v
（126）         * @return
（127）         */
（128）        private int locateVertex(Character v) {
（129）            for (int i = 0; i < graph.vexNum; i++) {
（130）                if (graph.vertexNodes[i].data.equals(v)) {
（131）                    return i;
（132）                }
（133）            }
（134）            return 0;
（135）        }
（136）
（137）        /**
（138）         * 打印图
（139）         */
（140）        private void displayGraph() {
（141）            ArcNode p;
（142）            System.out.printf("总共有%d个顶点!\n", graph.vexNum);
（143）            for (int i = 0; i < graph.vexNum; i++) {
（144）                System.out.println(graph.vertexNodes[i].data);
（145）            }
（146）            System.out.printf("总共有%d条边!\n", graph.arcNum);
（147）            for (int i = 0; i < graph.vexNum; i++) {
（148）                p = graph.vertexNodes[i].firstArcNode;
（149）                while (p != null) {
（150）                    System.out.printf("%s->%s\t", graph.vertexNodes[i].data,
    graph.vertexNodes[p.adjvex].data);
（151）                    p = p.nextArcNode;
（152）                }
（153）                System.out.println();
```

```
（154）            }
（155）        }
（156）
（157）        /**
（158）         * 主函数
（159）         *
（160）         * @param args
（161）         */
（162）        public static void main(String[] args) {
（163）            DGraph dGraph = new DGraph();
（164）            dGraph.createGraph();
（165）            dGraph.displayGraph();
（166）            System.out.println("请输入起点节点...");
（167）            Scanner scanner = new Scanner(System.in);
（168）            Character s = Character.valueOf(scanner.next().charAt(0));
（169）            int s0 = dGraph.locateVertex(s);
（170）            dGraph.dijkstra(s0);
（171）            for (int i = 0; i < dGraph.graph.vexNum; i++) {
（172）                if (i != s0) {
（173）                    System.out.printf("%s->%s的最短路径为:%d\t",
        dGraph.graph.vertexNodes[s0].data,
（174）                            dGraph.graph.vertexNodes[i].data,
        dGraph.graph.vertexNodes[i].dist);
（175）                    dGraph.showPath(s0, i);
（176）                }
（177）            }
（178）        }
（179）
（180）        /**
（181）         * 单源最短路径
（182）         *
（183）         * @param s
（184）         */
（185）        private void dijkstra(int s0) {
（186）            ArcNode arcNode = null;
（187）            // 初始化各个节点距离初始源节点的路径长度
（188）            for (int i = 0; i < graph.vexNum; i++) {
（189）                arcNode = graph.vertexNodes[s0].firstArcNode;
（190）                while (arcNode != null) {
（191）                    if (arcNode.adjvex == i) {
（192）                        graph.vertexNodes[i].dist = arcNode.weight;
（193）                        graph.vertexNodes[i].nearNode = graph.vertexNodes[s0];
（194）                        break;
（195）                    }
```

```
（196）                    arcNode = arcNode.nextArcNode;
（197）                }
（198）                if (arcNode == null) {
（199）                    graph.vertexNodes[i].dist = INFINITY;
（200）                    graph.vertexNodes[i].nearNode = null;
（201）                }
（202）            }
（203）            graph.vertexNodes[s0].dist = 0;
（204）            for (int i = 0; i < graph.vexNum; i++) {
（205）                int m = -1;
（206）                int min_dist = INFINITY;
（207）                // 寻找出还未访问过的最短路径点
（208）                for (int j = 0; j < graph.vexNum; j++) {
（209）                    if (graph.vertexNodes[j].know == false &&
    graph.vertexNodes[j].dist < min_dist) {
（210）                        min_dist = graph.vertexNodes[j].dist;
（211）                        m = j;
（212）                    }
（213）                }
（214）                if (m == -1)
（215）                    continue;
（216）                graph.vertexNodes[m].know = true;
（217）                // 根据已找出最短路径的节点修正
（218）                for (int j = 0; j < graph.vexNum; j++) {
（219）                    arcNode = graph.vertexNodes[m].firstArcNode;
（220）                    if (graph.vertexNodes[j].know == false) {
（221）                        while (arcNode != null) {
（222）                            if (arcNode.adjvex == j) {
（223）                                if (arcNode.weight > 0
（224）                                        && graph.vertexNodes[j].dist >
    graph.vertexNodes[m].dist + arcNode.weight) {
（225）                                    graph.vertexNodes[j].dist =
    graph.vertexNodes[m].dist + arcNode.weight;
（226）                                    graph.vertexNodes[j].nearNode =
    graph.vertexNodes[m];
（227）                                }
（228）                            }
（229）                            arcNode = arcNode.nextArcNode;
（230）                        }
（231）                    }
（232）                }
（233）            }
（234）        }
（235）
```

```
（236）        /**
（237）         * 显示最短路径
（238）         *
（239）         * @param v1
（240）         * @param v2
（241）         */
（242）        private void showPath(int v0, int v) {
（243）            LinkedStack<Character> linkedStack = new LinkedStack<Character>();
（244）            boolean bCanReach = true;
（245）            while (v != v0) {
（246）                linkedStack.push(graph.vertexNodes[v].data);
（247）                if (graph.vertexNodes[v].dist == INFINITY) {
（248）                    bCanReach = false;
（249）                    break;
（250）                }
（251）                if (graph.vertexNodes[v].nearNode != null)
（252）                    v = locateVertex(graph.vertexNodes[v].nearNode.data);
（253）            }
（254）            linkedStack.push(graph.vertexNodes[v0].data);
（255）            Character e = null;
（256）            while ((e = linkedStack.pop()) != null) {
（257）                System.out.print(e + " ");
（258）            }
（259）            if (bCanReach == false)
（260）                System.out.print("终点不可达！");
（261）            System.out.println();
（262）        }
（263）    }
```

6.排序算法

```
（1）    package com.ex.sort;
（2）
（3）    import java.util.Arrays;
（4）    import java.util.Scanner;
（5）
（6）    /**
（7）     * 排序算法汇总，默认按照升序进行排序 待排序数组为：{2,16,9,8,11,33,5,4}
         具体排序算法见注释部分
（8）     *
（9）     * @author lenovo write:Sep 3,2015 21:05 in School of Remote Sensing and
（10）    *              Information Technology in Wuhan University
（11）    */
```

```
（12）
（13） public class SortAlgorithms {
（14）
（15）  static enum ESort {
（16）        DI {
（17）              void print(int[] nArray) {
（18）                    System.out.println("直接插入排序的结果为:" +
      Arrays.toString(nArray));
（19）              }
（20）        },
（21）        HF {
（22）              void print(int[] nArray) {
（23）                    System.out.println("折半查找插入排序的结果为:" +
      Arrays.toString(nArray));
（24）              }
（25）        },
（26）        SH {
（27）              void print(int[] nArray) {
（28）                    System.out.println("希尔排序的结果为:" +
      Arrays.toString(nArray));
（29）              }
（30）        },
（31）        SS {
（32）              void print(int[] nArray) {
（33）                    System.out.println("简单选择排序的结果为:" +
      Arrays.toString(nArray));
（34）              }
（35）        },
（36）        BS {
（37）              void print(int[] nArray) {
（38）                    System.out.println("冒泡排序的结果为:" +
      Arrays.toString(nArray));
（39）              }
（40）        };
（41）        abstract void print(int[] nArray);
（42）  }
（43）
（44）  /**
（45）    * 待排序数组
（46）    */
（47）  private static int[] nArr = { 2, 16, 9, 8, 11, 33, 5, 4 };
（48）
（49）  /**
（50）    * 入口函数
```

```
（51）     *
（52）     * @param args
（53）     */
（54）    public static void main(String[] args) {
（55）          // TODO Auto-generated method stub
（56）          System.out.println("请输入排序方式:DI表示直接插入排序，HF表示折半插
          入排序，SH表示希尔排序，SS表示简单选择排序，BS表示冒泡排序.");
（57）          ESort sortType;
（58）          Scanner scanner = new Scanner(System.in);
（59）          sortType = ESort.valueOf(scanner.next());
（60）          switch (sortType) {
（61）          case DI:
（62）                insertSort();
（63）                break;
（64）          case HF:
（65）                halfFindSort();
（66）                break;
（67）          case SH:
（68）                shellInsertSort();
（69）                break;
（70）          case SS:
（71）                simpleSelectSort();
（72）                break;
（73）          case BS:
（74）                bubbleSort();
（75）                break;
（76）          default:
（77）                break;
（78）          }
（79）          sortType.print(nArr);
（80）    }
（81）
（82）    /**
（83）     *  直接插入排序
（84）     */
（85）    private static void insertSort() {
（86）          int nTemp = 0;
（87）          for (int i = 0; i < nArr.length - 1; i++) {
（88）                nTemp = nArr[i + 1];
（89）                int j = i;
（90）                while (j > -1 && nTemp < nArr[j]) {
（91）                      nArr[j + 1] = nArr[j];
（92）                      j--;
（93）                }
```

```
（94）                nArr[j + 1] = nTemp;
（95）            }
（96）    }
（97）
（98）    /**
（99）     * 折半插入排序
（100）     */
（101）    private static void halfFindSort() {
（102）        int temp, low, high, mid;
（103）        int i, j;
（104）        for (i = 0; i < nArr.length - 1; i++) {
（105）            temp = nArr[i + 1];
（106）            low = 0;
（107）            high = i;
（108）            while (low <= high) {
（109）                mid = (low + high) / 2;
（110）                if (nArr[mid] > temp) {
（111）                    high = mid - 1;
（112）                } else {
（113）                    low = mid + 1;
（114）                }
（115）            }
（116）            for (j = i; j >= low; j--) {
（117）                nArr[j + 1] = nArr[j];
（118）            }
（119）            nArr[low] = temp;
（120）        }
（121）
（122）    }
（123）
（124）    /**
（125）     * 希尔排序
（126）     */
（127）    private static void shellInsertSort() {
（128）        for (int i = nArr.length / 2 - 1; i >= 0; i--) {
（129）            shellInsert(2 * i + 1);
（130）        }
（131）    }
（132）
（133）    private static void shellInsert(int delta) {
（134）        int nTemp, j;
（135）        for (int i = delta; i < nArr.length; i++) {
（136）            if (i == 1) {
（137）                System.out.println("最后一趟排序！ ");
```

```
（138）                    }
（139）                    if (nArr[i] < nArr[i - delta]) {
（140）                          nTemp = nArr[i];
（141）                          for (j = i - delta; j > 0 && nArr[j] > nTemp; j -= delta) {
（142）                                nArr[j + delta] = nArr[j];
（143）                          }
（144）                          nArr[j + delta] = nTemp;
（145）                    }
（146）              }
（147）        }
（148）
（149）        /**
（150）         * 简单选择排序
（151）         */
（152）        private static void simpleSelectSort() {
（153）              int nTemp = 0;
（154）              for (int i = 0; i < nArr.length - 1; i++) {
（155）                    for (int j = i + 1; j < nArr.length; j++) {
（156）                          if (nArr[i] > nArr[j]) {
（157）                                nTemp = nArr[i];
（158）                                nArr[i] = nArr[j];
（159）                                nArr[j] = nTemp;
（160）                          }
（161）                    }
（162）              }
（163）        }
（164）
（165）        /**
（166）         * 冒泡排序
（167）         */
（168）        private static void bubbleSort() {
（169）              int nTemp = 0;
（170）              for (int i = 0; i < nArr.length - 1; i++) {
（171）                    for (int j = 0; j < nArr.length - i - 1; j++) {
（172）                          if (nArr[j] > nArr[j + 1]) {
（173）                                nTemp = nArr[j];
（174）                                nArr[j] = nArr[j + 1];
（175）                                nArr[j + 1] = nTemp;
（176）                          }
（177）                    }
（178）              }
（179）        }
（180）
（181）  }
```

7.常见考题（更新中…）

7.1 单链表反转

```
（1）  package com.ex.problems;
（2）
（3）  public class LinkedListReverse {
（4）
（5）    /**
（6）     * 单链表的数据结构
（7）     *
（8）     * @author lenovo
（9）     *
（10）    * @param <T>
（11）    */
（12）   static final class LNode<T> {
（13）        T data;
（14）        LNode<T> next;
（15）
（16）        /**
（17）         * 注明，构造函数不需要带额外的类型参数，不管是C++还是Java，注意
      啦，带额外参数的是类类型
（18）         *
（19）         * @param data
（20）         * @param next
（21）         */
（22）        public LNode(T data, LNode<T> next) {
（23）            this.data = data;
（24）            this.next = next;
（25）        }
（26）
（27）   }
（28）
（29）   // 定义单链表的头指针
（30）   private static LNode<Integer> head;
（31）
（32）   // 初始化单链表
（33）   private static void initLinkedList() {
（34）        head = new LNode<Integer>(null, null);
（35）   }
（36）
（37）   /**
```

```
（38）     * 创建单链表
（39）     *
（40）     * @param nArr
（41）     */
（42）    private static void createLinkedList(int... nArr) {
（43）        LNode<Integer> prev = head, cur;
（44）        for (int i = 0; i < nArr.length; i++) {
（45）            cur = new LNode<Integer>(nArr[i], null);
（46）            prev.next = cur;
（47）            prev = cur;
（48）        }
（49）
（50）    }
（51）
（52）    /**
（53）     * 打印单链表
（54）     */
（55）    public static void printLinkedList() {
（56）        LNode<Integer> cur = head;
（57）        while (cur.next != null) {
（58）            System.out.println(cur.next.data);
（59）            cur = cur.next;
（60）        }
（61）    }
（62）
（63）    /**
（64）     * 反转单链表
（65）     */
（66）    private static void reverseLinkedList() {
（67）        if (head == null) {
（68）            return;
（69）        }
（70）        LNode<Integer> prev, cur, next;
（71）        prev = head;
（72）        cur = prev.next;
（73）        while (cur != null) {
（74）            next = cur.next;
（75）            if (prev != head) //  当时收个节点时候直接将next域置为空
（76）                cur.next = prev;
（77）            else
（78）                cur.next = null;
（79）            prev = cur;
（80）            cur = next;
（81）        }
```

```
（82）          head.next = prev;
（83）    }
（84）
（85）    /**
（86）     *  主函数
（87）     *
（88）     * @param args
（89）     */
（90）    public static void main(String[] args) {
（91）          // TODO Auto-generated method stub
（92）          initLinkedList();
（93）          createLinkedList(1, 3, 5, 6, 7);
（94）          System.out.println("反正前链表为:");
（95）          printLinkedList();
（96）          System.out.println("反转后链表为:");
（97）          reverseLinkedList();
（98）          printLinkedList();
（99）    }
（100）
（101）  }
```