Java 数据结构和算法

授课老师：韩国庆

目录

1. 数据结构和算法概述

1.1 为什么要学习数据结构和算法

- 误区

1、认为学习数据结构和算法需要很扎实的数学和编程功底，学起来并不容易；

2、数据结构和算法在平时开发过程中的应用并不多，而且都有现成的类库接口让我们调用，没必要花太多时间去研究。

- 疑问

为什么数据结构和算法使用率这么低，这些大公司还必问呢？这是因为数据结构和算法真的很重要，而且他们的使用率并不低，所有的程序中都有数据结构和算法的身影，他们默默的为程序服务，只是你没重视他们罢了。

- 列举我们为什么要学好数据结构和算法原因
  - 提升代码性能，结省空间复杂度和时间复杂度；
  - 算法锻炼自己的逻辑思维；
  - 你会不自觉的考虑你写的代码处理的数据量级是多少，会考虑你的代码是否能处理大量数据的情况
  - 更好的理解应用软件和框架，很多知名软件和框架中都大量用了数据结构算法，比如 mysql 的索引用了 b+树，redis 的 list 底层用了跳跃表，理解这些数据结构能更好的帮助我们理解使用这些软件。
  - 掌握了数据结构与算法，你看待问题的深度，解决问题的角度就会完全不一样。
  - 一流的程序员搞算法，二流的程序员搞架构，三流的程序员搞业务；

1.2 数据结构和算法介绍

- 算法

案例 1：

有一个背包，背包容量是 M=150。有 7 个物品，物品可以分割成任意大小。要求尽可能让装入背包中的物品总价值最大，但不能超过总容量。

案例 2：

有一对兔子,从出生后第 3 个月起每个月都生一对兔子,小兔子长到第 4 个月后,每个月又生一对兔子,假如兔子不死,问每个月的兔子总数为多少?

是指解题方案的准确而完整的描述，是一系列解决问题的清晰指令，算法代表着用系统的方法描述解决问题的策略机制。也就是说，能够对一定规范的输入，在有限时间内获得所要求的输出。如果一个算法有缺陷，或不适合于某个问题，执行这个算法将不会解决这个问题。不同的算法可能用不同的时间、空间或效率来完成同样的任务。一个算法的优劣可以用空间复杂度与时间复杂度来衡量。

算法是独立存在的一种解决问题的方法和思想。
对于算法而言，实现的语言并不重要，重要的是思想。
算法可以有不同的语言描述实现版本（如 C 描述、C++描述、Python 描述等）

- 数据结构

数据结构就是把数据组织起来，为了更方便地使用数据我们为了解决问题，需要将数据保存下来，然后根据数据的存储方式来设计算法实现进行处理，那么数据的存储方式不同就会导致需要不同的算法进行处理。我们希望算法解决问题的效率越快越好，于是我们就需要考虑数据究竟如何保存的问题，这就是数据结构。

数据结构是计算机存储、组织数据的方式。数据结构是指相互之间存在一种或多种特定关系的数据元素的集合。

1.3 数据结构和算法关系

程序 = 数据结构 + 算法

数据结构是算法的基础。

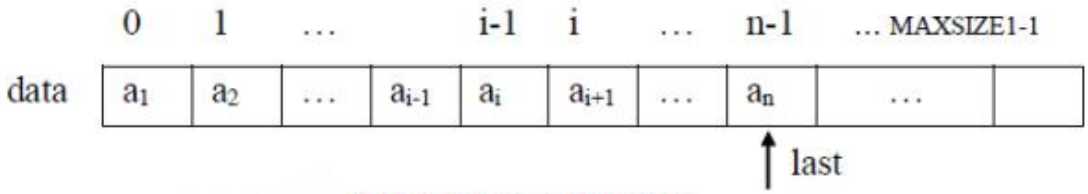
图书馆储藏书籍你肯定见过吧？为了方便查找，图书管理员一般会将书籍分门别类进行“存储”。按照一定规律编号，就是书籍这种“数据”的存储结构。

那我们如何来查找一本书呢？有很多种办法，你当然可以一本一本地找，也可以先根据书籍类别的编号，是人文，还是科学、计算机，来定位书架，然后再依次查找。笼统地说，这些查找方法都是算法。

数据结构和算法是相辅相成的。数据结构是为算法服务的，算法要作用在特定的数据结构之上。 因此，我们无法孤立数据结构来讲算法，也无法孤立算法来讲数据结构。

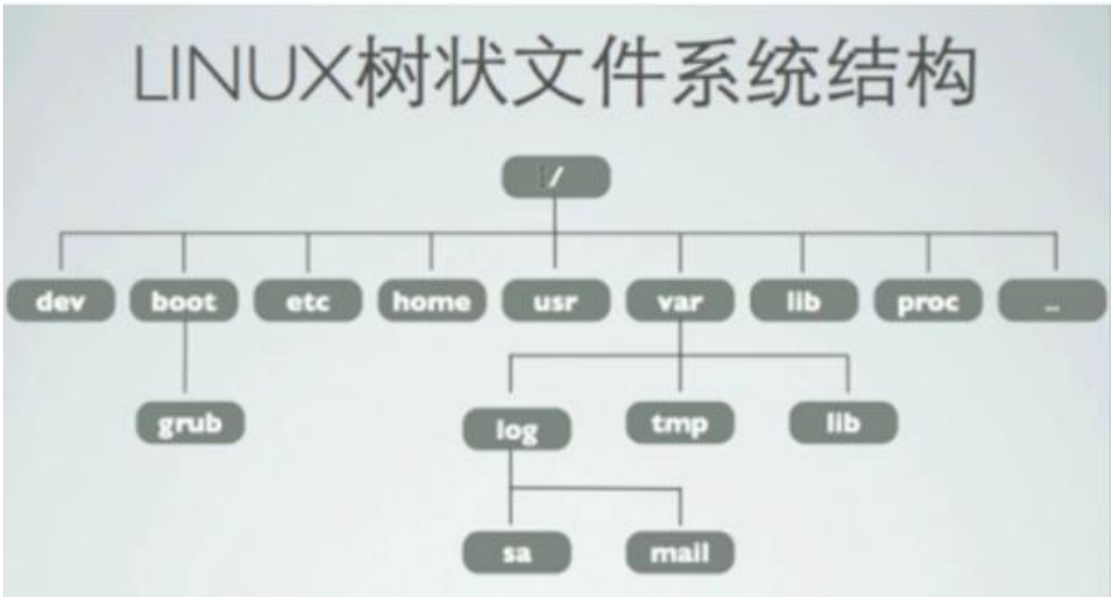
1.4 线性结构和非线性结构

- 线性结构
  - 线性结构作为最常用的数据结构，其特点是数据元素之间存在一对一的线性关系。
  - 线性结构有两种不同的存储结构，即顺序存储结构(数组)和链式存储结构(链表)。顺序存储的线性表称为顺序表，顺序表中的存储元素是连续的。
  - 链式存储的线性表称为链表，链表中的存储元素不一定是连续的，元素结点存放数据元素以及相邻元素的地址信息。
  - 线性结构常见的有：数组、队列、链表和栈。



线性表的顺序存储示意图

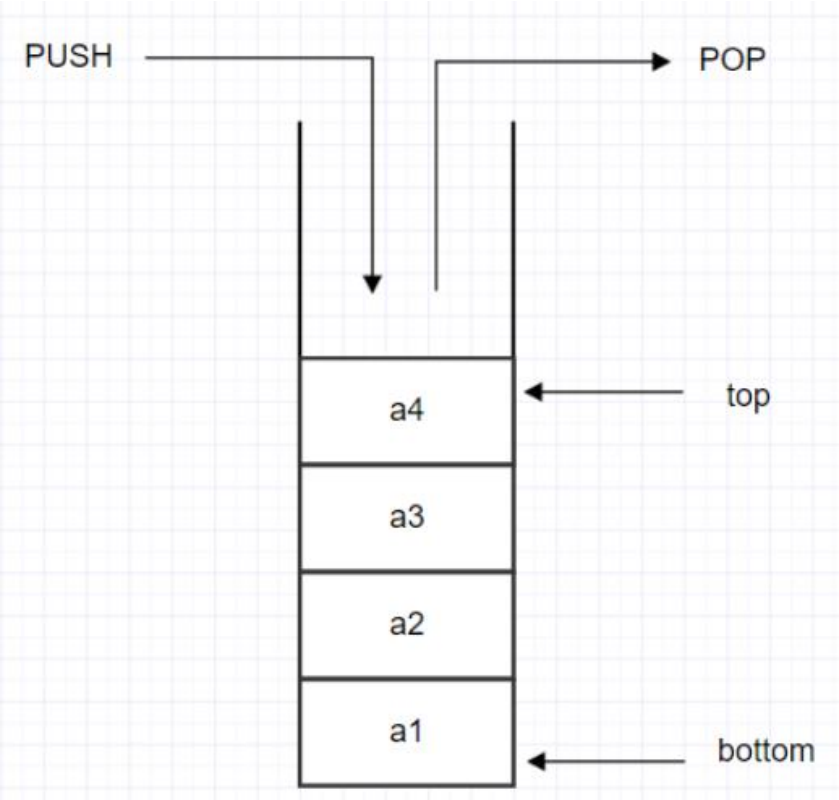
- 非线性结构
  - 二维数组，多维数组，广义表，树结构，图结构。



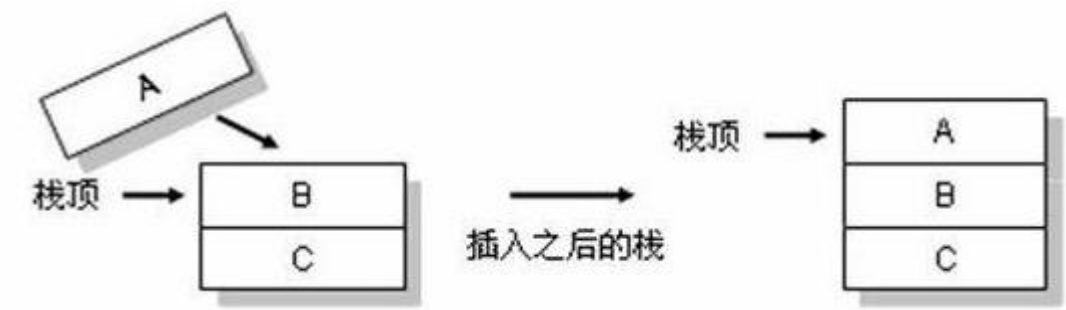
2. 栈

2.1 栈的介绍

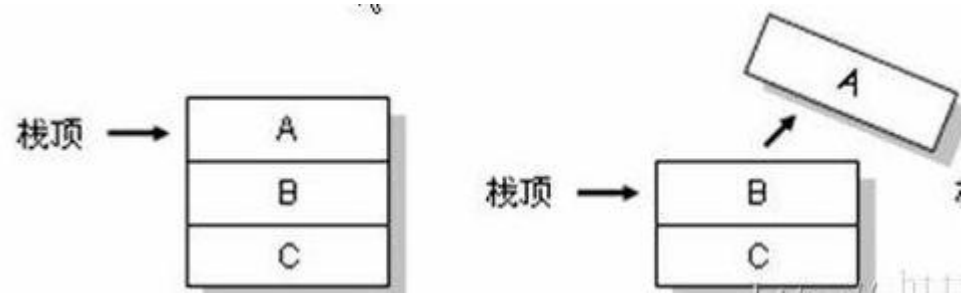
栈是限制插入和删除只能在一个位置上进行的线性表。其中，允许插入和删除的一端位于表的末端，叫做栈顶（top），不允许插入和删除的另一端叫做栈底（bottom）。对栈的基本操作有 PUSH（压栈）和 POP （出栈），前者相当于表的插入操作（向栈顶插入一个元素），后者则是删除操作（删除一个栈顶元素）。栈是一种**后进先出（LIFO）**的数据结构，最先被删除的是最近压栈的元素。


PUSH
POP
a4
top
a3
a2
a1
bottom

压栈：


A
栈顶
B
C
插入之后的栈
栈顶
A
B
C

弹栈：


栈顶
A
B
C
A
栈顶
B
C

栈实现

由于栈是一个表，因此任何实现表的方法都可以用来实现栈。主要有两种方式，链表实现和数组实现。

链表实现栈

可以使用单链表来实现栈。通过在表顶端插入一个元素来实现 PUSH，通过删除表顶端元素来实现 POP。使用链表方式实现的栈又叫**动态栈**。动态栈有链表的部分特性，即元素与元素之间在物理存储上可以不连续，但是功能有些受限制，动态栈只能在栈顶处进行插入和删除操作，不能在栈尾或栈中间进行插入和删除操作

数组实现栈

栈也可以用数组来实现。使用数组方式实现的栈叫**静态栈**。

2.2 栈的应用场景

子程序的调用：在跳往子程序前，会先将下个指令的地址存到堆栈中，直到子程序执行完后再将地址取出，以回到原来的程序中。

处理递归调用：和子程序的调用类似，只是除了储存下一个指令的地址外，也将参数、区域变量等数据存入堆栈中。

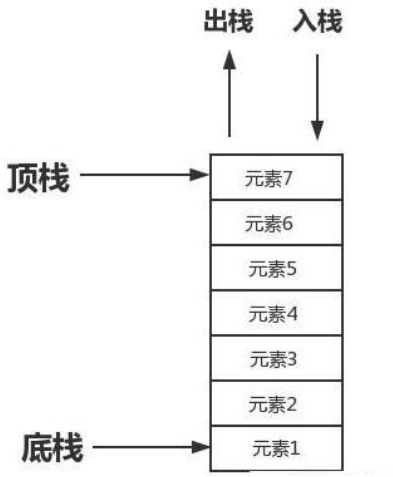
表达式的转换[中缀表达式转后缀表达式]与求值(实际解决)。

二叉树的遍历。

图形的深度优先(depth 一 first)搜索法。

2.3 栈的快速入门

习题 1:使用数组来模拟栈的使用




```
/**
 * author:韩国庆
 * date:2021/1/18 14:16
 * version:1.0
 */
public class ArrayStack {
    /**
        栈的大小
     */
    private int maxStack;

    /**
        数组用来模拟栈
     */
    private int[] stack;

    /**
     * 表示栈顶，默认值为-1
     */
```

```
    private int top = -1;

    /**
     *
     * @param maxStack 初始化栈的大小
     */
    public ArrayStack(int maxStack){
        this.maxStack = maxStack;
        this.stack = new int[this.maxStack];
    }

    /**
     *  判断是否已经栈满
     * @return
     */
    public boolean isFull(){
        return this.top==maxStack-1;
    }

    /**
     * 是否空栈
     */
    public boolean isEmpty(){
        return this.top==-1;
    }

    /**
     * 入栈
     */
    public void push(int value){
```

```
        if (isFull()){
            throw new RuntimeException("栈已满...");
        }

        top++;
        stack[top] = value;
    }

    /**
     *出栈
     */
    public int pop(){
        if (isEmpty()){
            throw new RuntimeException("空栈，没有数据");
        }
        int value = stack[top];
        top--;
        return value;
    }

    /**
     * 查看栈中数据
     */
    public void list(){
        if (isEmpty()){
            throw new RuntimeException("空栈，没有数据");
        }
        for (int i=0;i<stack.length;i++){
            System.out.printf("stack[%d]=%d\n",i,stack[i]);
        }
```

```
    }


}
```

习题 2：回文数

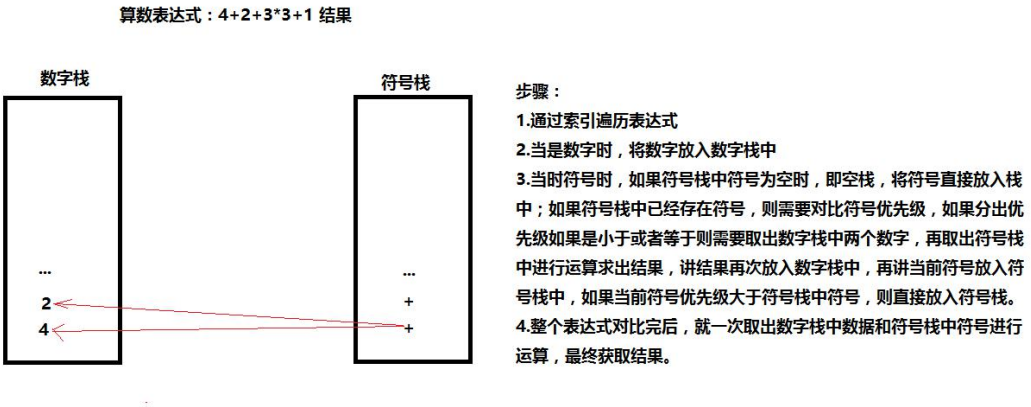
回文： 一个单词、短语或数字，从前往后写和从后往前写都是一样的。比如，单词“dad”、“racecar”就是回文；如果忽略空格和 标点符号，下面这个句子也是回文，“A man, a plan, a canal: Panama”， 数字 1001 也是回文。

思路：使用栈，可以轻松判断一个字符串是否是回文。我们将拿到的字符串的每个字符按从左至右的顺序压入栈。当字符串中的字符都入栈后，栈内就保存了一个反转后的字符串，最后的字符在栈顶，第一个字符在栈底

```
public static boolean detection(String words){
        ArrayStack arrayStack = new ArrayStack(10);
        int length = words.length();
        for (int i=0;i<length;i++){
            arrayStack.push(words.charAt(i));
        }
        String newValue = "";
        for (int i=0;i<arrayStack.length();i++){
            if (!arrayStack.isEmpty()){
                Object value = arrayStack.pop();
                newValue = newValue+value;
            }
        }
        if (words.equals(newValue)){
            return true;
        }
        return false;
    }
```

2.4 栈实现计算器

思维分析结果图：



2.4.1 ArrayStack

```
public class ArrayStack {
    /**
        栈的大小
     */
    private int maxStack;

    /**
        数组用来模拟栈
     */
    private int[] stack;

    /**
     * 表示栈顶，默认值为-1
```

```
     */
    private int top = -1;

    /**
     *
     * @param maxStack 初始化栈的大小
     */
    public ArrayStack(int maxStack){
        this.maxStack = maxStack;
        this.stack = new int[this.maxStack];
    }

    /**
     *  判断是否已经栈满
     * @return
     */
    public boolean isFull(){
        return this.top==maxStack-1;
    }

    /**
     * 是否空栈
     */
    public boolean isEmpty(){
        return this.top==-1;
    }

    /**
     * 入栈
     */
```

```
    public void push(int value){
        if (isFull()){
            throw new RuntimeException("栈已满...");
        }

        top++;
        stack[top] = value;
    }

    /**
     *出栈
     */
    public int pop(){
        if (isEmpty()){
            throw new RuntimeException("空栈，没有数据");
        }
        int value = stack[top];
        top--;
        return value;
    }

    /**
     * 查看栈中数据
     */
    public void list(){
        if (isEmpty()){
            throw new RuntimeException("空栈，没有数据");
        }
        for (int i=0;i<stack.length;i++){
            System.out.printf("stack[%d]=%d\n",i,stack[i]);
```

```
        }
    }

    /**
     * 查看栈大小
     * @return
     */
    public int length(){
        return stack.length;
    }

    /**
     * 查看栈顶
     * @return
     */
    public int peek(){
        return stack[top];
    }

    /**
     * 运算符优先级
     * 数字越大优先级越高
     */
    public int priority(int oper){
        if (oper=='*' || oper=='/'){
            return 1;
        }else if (oper=='+' || oper=='-'){
            return 0;
        }else {
            return -1;
```

```
        }

    }
    /**
     * 判断是否是一个运算符
     */
    public boolean isOper(char v){
        return v=='+' || v=='-'||v=='*'||v=='/';
    }

    /**
     * 计算
     */
    public int calculate(int num1,int num2,int oper){
        int res = 0;
        switch (oper){
            case '+':
                res=num1+num2;
                break;
            case '-':
                res=num2-num1;
                break;
            case '*':
                res=num1*num2;
                break;
            case '/':
                res=num2/num1;
                break;

                default:
```

```
                    break;

            }
            return res;
        }
```

2.4.2 main

```
public static void main(String[] args) {



        String str = "3+2*3-1-1";

        ArrayStack numStack = new ArrayStack(10);
        ArrayStack symbolStack = new ArrayStack(10);
        int length = str.length();
        int temp1 = 0;
        int temp2 = 0;
        int symbolChar = 0;
        int result = 0;
        String values = "";
        for (int i=0;i<length;i++) {
            //获取每一个字符
            char c = str.charAt(i);
            //判断是否是一个字符
            if (symbolStack.isOper(c)){
                //如果字符栈中不是空栈
                if (!symbolStack.isEmpty()){
                    //如果当前字符优先级小于等于字符栈中存在的
                    if
```

```
(symbolStack.priority(c)<=symbolStack.priority(symbolStack.peek())){
                        /**
                         * 即从数字栈中 pop 出来两个数字，再从符号栈中 pop 出来一个符号进行计算，然后把结果
                         * 然后把结果再次存入栈中
                         */
                        temp1 = numStack.pop();
                        temp2 = numStack.pop();
                        symbolChar = symbolStack.pop();
                        result = numStack.calculate(temp1,temp2,symbolChar);
                        //把计算结果再次放入栈中
                        numStack.push(result);
                        //把当前符号放入栈中
                        symbolStack.push(c);
                    }else {
                        //如果当前符号优先级大于符号栈中的符号，直接放入符号栈中
                        symbolStack.push(c);
                    }
                }else {
                    //如果符号栈中是空，则也直接如符号栈
                    symbolStack.push(c);
                }
            }else {
                //如果扫描的是数字。数字很能存在多位，比如 33,22,100，如何能保证这多位数字
                //判断当前字符后一位是否是一个字符，如果是字符表示当前数字结束直接存入数字栈，如果不是字符，表示
                //这是一个多位数字
```



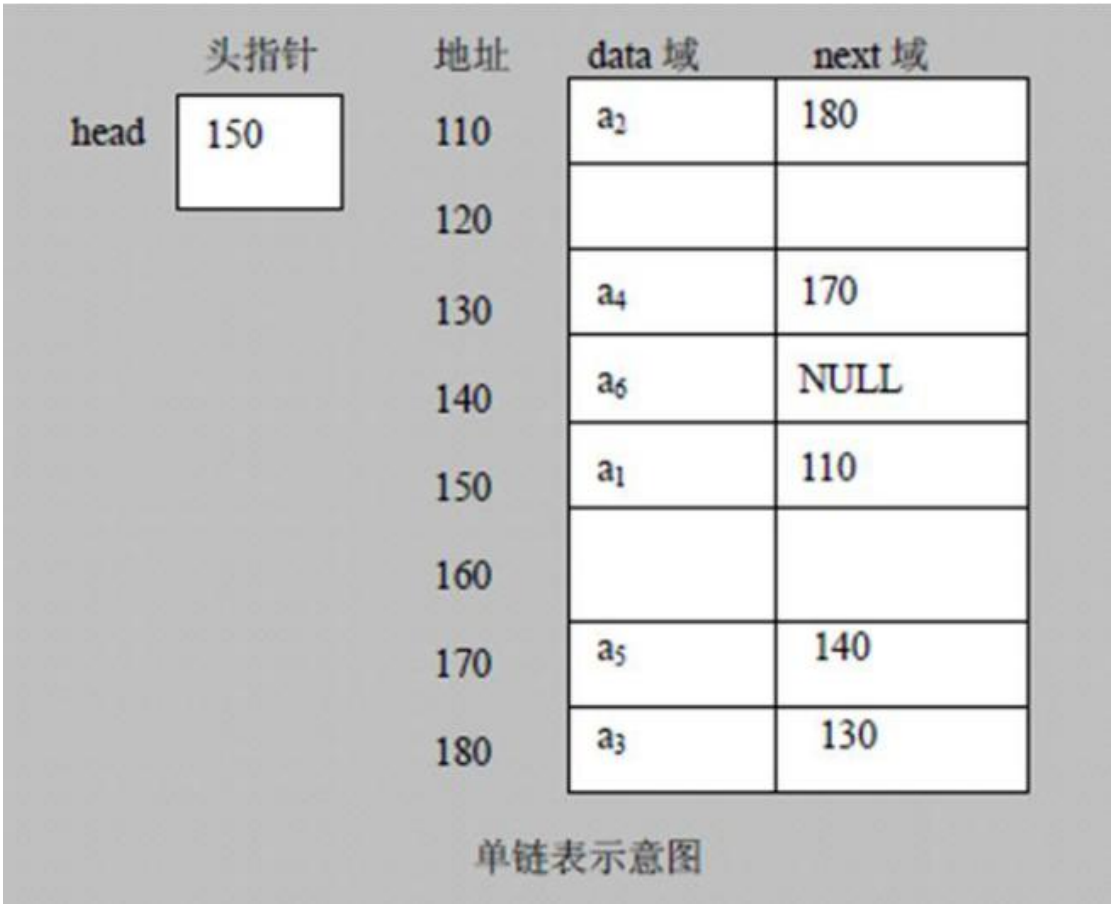
```
                values+=c;
                if (i==length-1){//表示是最后一个数字，可以直接存放在
数字栈中
                    numStack.push(Integer.parseInt(values));
                }else {
                    char data = str.substring(i+1, i + 2).charAt(0);
                    if (symbolStack.isOper(data)){//如果是符号，则把数
字存入数字栈中，并清空 values。
                        numStack.push(Integer.parseInt(values));
                        values="";
                    }
                }

            }
        }

        while (true){
            if (symbolStack.isEmpty()){
                break;
            }
            temp1 = numStack.pop();
            temp2 = numStack.pop();
            symbolChar = symbolStack.pop();
            result = numStack.calculate(temp1,temp2,symbolChar);
            numStack.push(result);
        }
        int res = numStack.pop();
        System.out.println("结果是:"+res);
```

```
    }
```

3. 链表

3.1 链表(Linked List)介绍

链表是一种物理存储单元上非连续，非顺序的存储结构，数据元素的逻辑顺序是通过链表中的指针链接实现的。

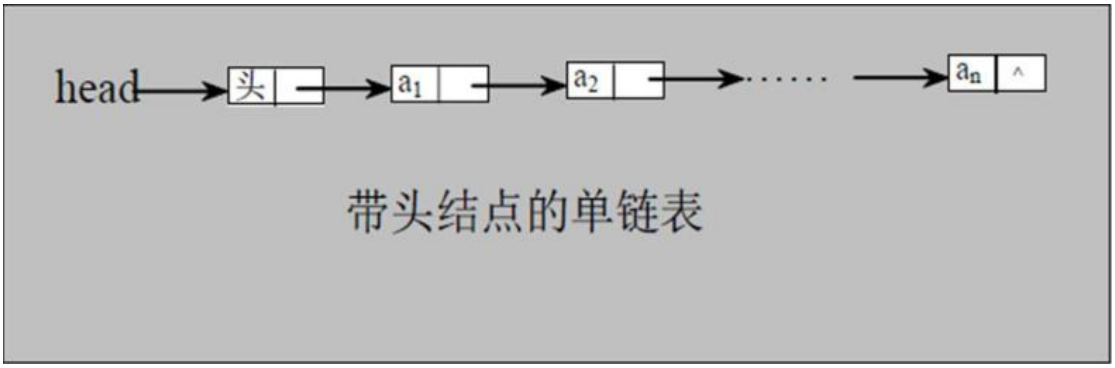


特点：

1. 链表是以结点形式存储的，是链式存储

2. 每个结点包含 data 区域和 next 区域
3. 如上图各个结点并不是连续存储的
4. 链表分带头结点链表和没有带头结点链表，根据实际的需求来确定

带头结点链表逻辑结构



需求：

根据带有头部的单链表，实现商品增删改查，并且也可以针对商品已编号进行排序，完成排行榜

3.2 单链表应用

```
/**
 * author:韩国庆
 * date:2021/1/23 14:50
 * version:1.0
 */
public class GoodsNode {

    public int gId;
```

```
    public String gName;
    public double gProce;

    //指针指向下一个结点
    public GoodsNode next;

    public GoodsNode(int gId, String gName, double gProce) {
        this.gId = gId;
        this.gName = gName;
        this.gProce = gProce;
    }


}
```

```
/**
 * author:韩国庆
 * date:2021/1/23 15:23
 * version:1.0
 */
public class DLLinkedList {

    private GoodsNode node = new GoodsNode(0,"",0.0);

    /**
     * 添加结点
     * @param goodsNode
     */
    public void add(GoodsNode goodsNode){
```

```
        GoodsNode temp = node;
        while (true){
            if (temp.next == null){
                break;
            }
            temp = temp.next;
        }

        temp.next = goodsNode;
    }

    /**
     * 插入结点按照 id 编号插入
     */
    public void addByOrder(GoodsNode goodsNode){
        GoodsNode temp = node;
        boolean flag = false;
        while (true){
            if (temp.next==null){
                break;
            }
            if (temp.next.gId>goodsNode.gId){
                break;
            }else if (temp.next.gId==goodsNode.gId){//说明已经
                flag = true;
                break;
            }
            temp = temp.next;
        }
```



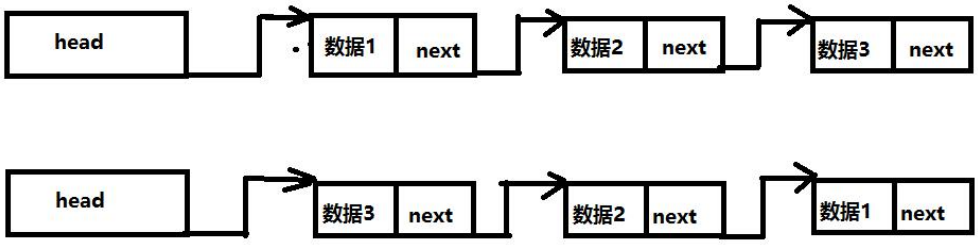
```
        if (flag){
            System.out.println("不能添加该商品，已经存在了...");
        }else {
            goodsNode.next = temp.next;
            temp.next = goodsNode;
        }
    }

    /**
     * 修改结点
     * @param goodsNode
     */
    public void updateNode(GoodsNode goodsNode){
        if (node.next==null){
            System.out.println("链表为空...");
            return;
        }
        GoodsNode temp = node.next;
        boolean flag = false;
        while (true){
            if (temp == null){
                break;
            }
            if (temp.gId == goodsNode.gId){
                flag = true;
                break;
            }
            temp = temp.next;
        }
```

```
        if (flag){
            temp.gName = goodsNode.gName;
            temp.gProce = goodsNode.gProce;
        }else {
            System.out.println("没有找到该结点");
        }
    }

    /**
     * 删除结点
     *
     */
    public void delNode(int gId){
        GoodsNode temp = node;
        boolean flg = false;
        while(true){
            if (temp.next==null){
                break;
            }
            if (temp.next.gId==gId){
                flg = true;
                break;
            }
            temp = temp.next;
        }

        if (flg){
            temp.next = temp.next.next;
        }else {
            System.out.println("要删除的信息不存在");
```

```
        }
    }

}
```

3.2.1 常见面试题

给定一个链表，实现倒转链表




```
public void reverse(GoodsNode goodsNode){
        if (goodsNode.next==null || goodsNode.next.next == null){
            return;
        }

        GoodsNode point = goodsNode.next;
        GoodsNode next = null;
        GoodsNode reverseNode = new GoodsNode(0,"",0.0);

        while (point !=null){
            next = point.next;
            point.next = reverseNode.next;
            reverseNode.next = point;
            point = next;
```

```
        }

        goodsNode.next = reverseNode.next;
    }
```

3.3 双向链表应用场景

双向链表也叫双链表，是链表的一种，它的每个数据结点中都有两个指针，分别指向直接后继和直接前驱。所以，从双向链表中的任意一个结点开始，都可以很方便地访问它的前驱结点和后继结点。

图：单链表示意图

图：双向链表示意图


```
/**
 * author:韩国庆
 * date:2021/1/25 16:45
 * version:1.0
 */
public class GoodsNode2 {
    public int gId;
    public String gName;
    public double gProce;

    //指针指向下一个结点
```

```
    public GoodsNode2 next;
    //指针指向上一个结点
    public GoodsNode2 pre;

    public GoodsNode2(int gId, String gName, double gProce) {
        this.gId = gId;
        this.gName = gName;
        this.gProce = gProce;
    }
}
```

```
/**
 * author:韩国庆
 * date:2021/1/25 16:47
 * version:1.0
 */
public class TwoLinkedList {

    private GoodsNode2 node = new GoodsNode2(0,"",0.0);

    /**
     *
     * @return  返回头结点
     */
    public GoodsNode2 returnHeadNode(){
        return node;
    }

    /**
```

```
     * 添加结点到链表最后面
     */
    public void addLast(GoodsNode2 goodsNode2){
        GoodsNode2 temp = node;

       while (true){
            if (temp.next == null){
                break;
            }

            temp = temp.next;
       }
       temp.next = goodsNode2;
       goodsNode2.pre = temp;

    }

    /**
     * 修改结点
     */
    public void updateNode(GoodsNode2 newNode){
        if (node.next==null){
            System.out.println("链表为空");
            return;
        }
        GoodsNode2 temp = node.next;
        boolean flg = false;
        while (flg){
            if (temp == null){
                break;
```

```
            }
            if (temp.gId == newNode.gId){
                flg = true;
                break;
            }
            temp = temp.next;
        }
        if (flg){
            temp.gName = newNode.gName;
            temp.gProce = newNode.gProce;
        }else {
            System.out.println("没有找到...");
        }

    }

    /**
     * 双向链表删除
     */
    public void delNode(int gId){
        if (node.next == null){
            System.out.println("链表为空,无法删除");
            return;
        }
        GoodsNode2 temp = node.next;
        boolean flag = false;
        while (true){
            if (temp == null){
                break;
            }
```

```
            if (temp.gId == gId){
                flag = true;
                break;
            }
            temp = temp.next;
        }
        if (flag){
            temp.pre.next = temp.next;
            if (temp.next !=null){
                temp.next.pre = temp.pre;
            }
        }else {
            System.out.println("要删除的结点不存在");
        }

    }
}
```

3.4 单向环形链表应用场景

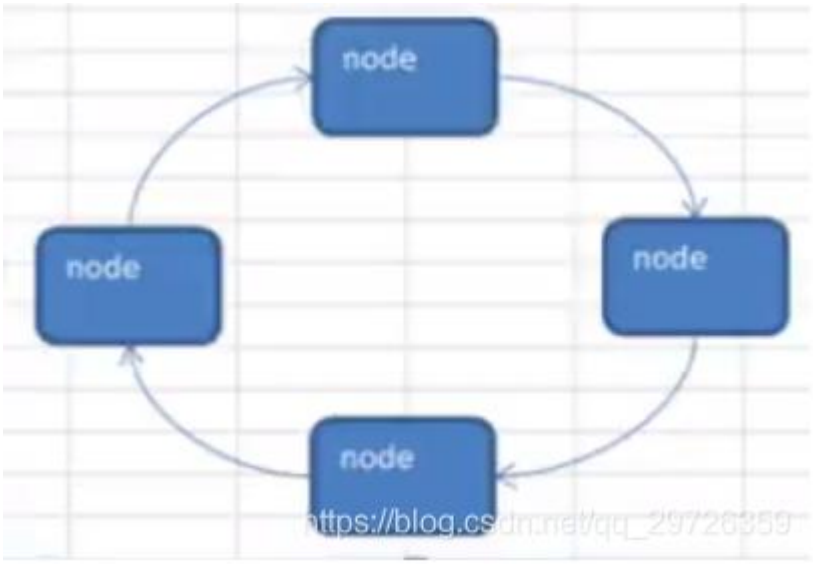
约瑟夫问题（约瑟夫环）

设编号为 1,2…，n 的 n 个人围坐一圈，约定编号为 k(1<=k<=n)的人从 1 开始报数，数到 m 的那个人出列，它的下一位又从 1 开始报数，数到 m 的那个人出列，它的下一位又从 1 开始报数，数到 m 的那个人又出列，依次类推，直到所有人出列为止，由此产生一个出队编号的序列。

3.5 单向环形链表介绍

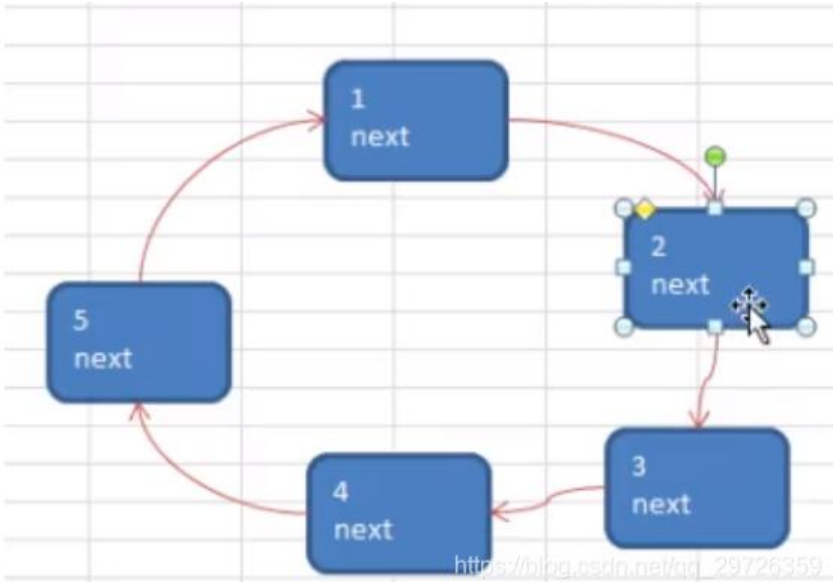
采用一个单项环形链表来处理（可以带表头也可以不带表头）,具体使用头结点还是不带头

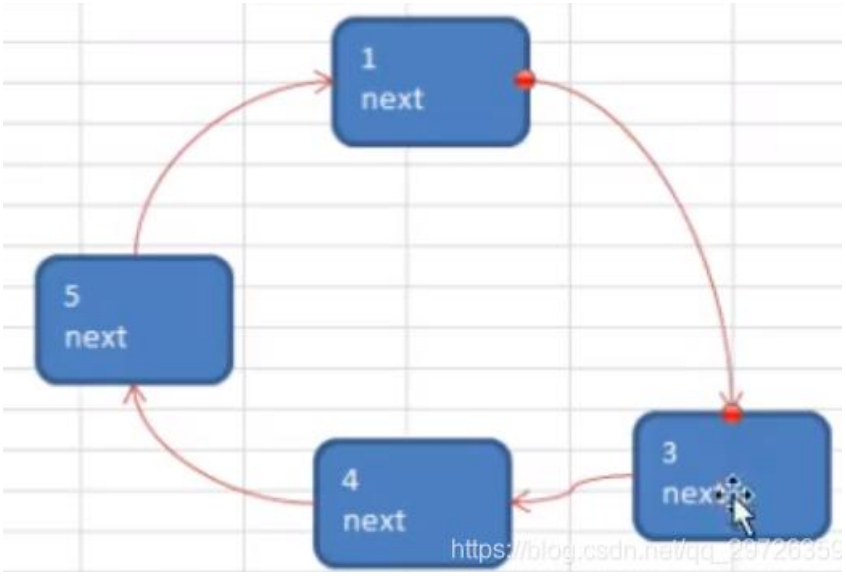
结点，后面会具体分析。

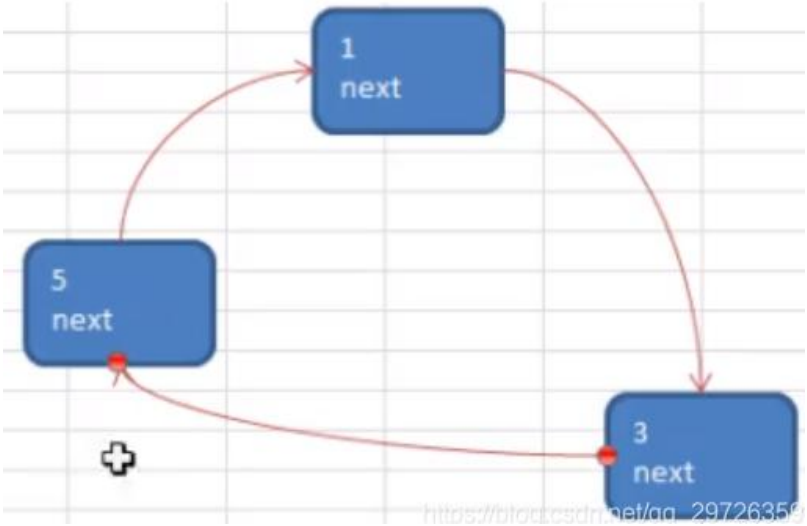


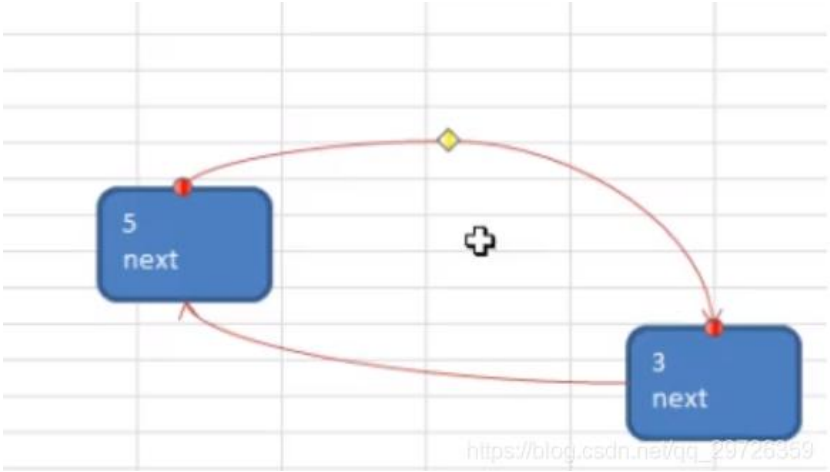
约瑟问题示意图

- n=5,即有 5 个人
- k=1，从第一个人开始报数
- m=2，数 2 下


1
next
2
next
3
next
4
next
5
next


1
next
3
next
4
next
5
next


1
next
5
next
3
next


5
next
3
next


3
next

解决约瑟夫问题分为两个步骤

第一步：做环形链表

第二步：解决约瑟夫问题

```
/**
 * @company: 北京动力节点
 * @author:韩国庆
 */
public class Boy {

    private int no;//编号

    private Boy next;//指向下一个节点，默认依旧为空


    public Boy(int no){
        this.no = no;
    }
    public void setNo(int no) {
        this.no = no;
    }
    public int getNo() {
        return no;
    }
    public Boy getNext() {
        return next;
    }
    public void setNext(Boy next) {
        this.next = next;
    }

}
```

```
/**
 * @company: 北京动力节点
 * @author:韩国庆
 */
public class CircleSingleLinkedList {

    //创建一个first 节点，当前没有编号
    private Boy first = new Boy(-1);

    //加入小孩节点，构建成环形链表
    public void addBoy(int nums){
        //判断数据真实性，nums 做数据校验
        if (nums < 1 ){
            System.out.println("nums 的值不正确");
            return;
        }
        Boy curBoy = null;//辅助指针，帮助构建环形链表
        //使用for 循环来创建环形链表
        for (int i = 1;i <= nums;i ++){
            //根据编号，创建小孩节点
            Boy boy = new Boy(i);
            //如果是第一个小孩
            if (i == 1){
                first = boy;
                first.setNext(first);//构成环状
```

```
            curBoy = first;//让 curBoy 指向第一个小孩
        }else {
            curBoy.setNext(boy);//前面的指向新来的小孩
            boy.setNext(first);//新添加的小孩指向开头，构成环形
            curBoy = boy;//后移
        }
    }
}
//遍历当前的环形链表
public void showBoy(){
    //判断链表是否为空
    if (first == null){
        System.out.println("链表为空~~");
        return;
    }
    //因为first 不能动，因此需要辅助指针，完成遍历
    Boy curBoy = first;
    while (true){
        System.out.printf("小孩的编号%d\n",curBoy.getNo());
        if (curBoy.getNext() == first){//说明遍历完毕
            break;
        }
        curBoy = curBoy.getNext();//让 curBoy 后移
    }
}
//根据用户的输入，计算出小孩节点出圈的顺序
```

```
    /*
    startNo 表示从第几个小孩开始数数
    countNum 表示数几下
    nums 表示 最初有多少小孩在圈
     */
    public void countBoy(int startNo,int countNum,int nums){
        //先对数据进行校验
        if (first == null || startNo < 1 || startNo > nums){
            System.out.println("参数输入有误，请重新输入！");
            return;
        }
        //创建一个辅助指针。帮助我们完成小孩出圈
        Boy helper = first;
        //需要创建一个辅助指针 helper，事先应该指向环形链表的最后的这个
节点。
        while (true){
            if (helper.getNext() == first){//说明小孩指向最后节点
                break;
            }
            helper = helper.getNext();
        }
        // 报数之前，先让 first 和 helper 移动 k-1 次
        for (int j = 0;j < startNo - 1; j ++){
            first = first.getNext();
            helper = helper.getNext();
        }
        //当小孩报数时，让 first 和 helper 指针同时的移动 m-1 次
```



```
        //这里是一个循环的操作，直到圈中只有一个节点
        while (true){
            if (helper == first){ //说明圈中只有一个节点
                break;
            }
            //否则让first 与helper 移动 countNum-1 次
            for (int j = 0;j < countNum - 1;j ++){
                first = first.getNext();
                helper = helper.getNext();
            }
            //这时first 指向的节点，就是要出圈小孩的节点

            System.out.printf("小孩%d 出圈\n",first.getNo());

            //这时将first 指向的小孩节点出圈
            first = first.getNext();
            helper.setNext(first);//
        }
        System.out.printf("最后留在圈中的小孩编号
是%d\n",first.getNo());
    }
}
```

```
public class TestApp {

    public static void main(String[] args) {
        //测试，构建环形与遍历是否可行
        CircleSingleLinkedList circleSingleLinkedList = new
CircleSingleLinkedList();
```

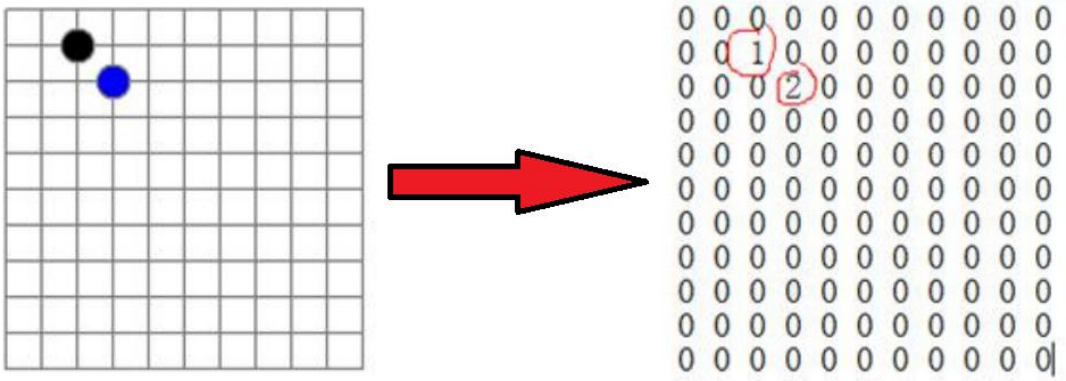
```
        circleSingleLinkedList.addBoy(5);//添加五个小孩节点
        circleSingleLinkedList.showBoy();
        //测试小孩出圈是否正确
        circleSingleLinkedList.countBoy(1,2,5);//从第一个开始，每两个
小孩出去一次，一共有五个小孩
    }
}
```

4. 稀疏数组和队列

4.1 稀疏数组

4.1.1 需求案例

常见五子棋程序中，有黑子和蓝子棋子，使用二维数组来表示



0 表示默认值，可以发现的是记录了很多没有意义的数据。那么我们思考，如何能够把数据记录优化至最小呢。

4.1.2 介绍

压缩条件：

1、 原数组中存在大量的无效数据，占据了大量的存储空间，真正有用的数据很少

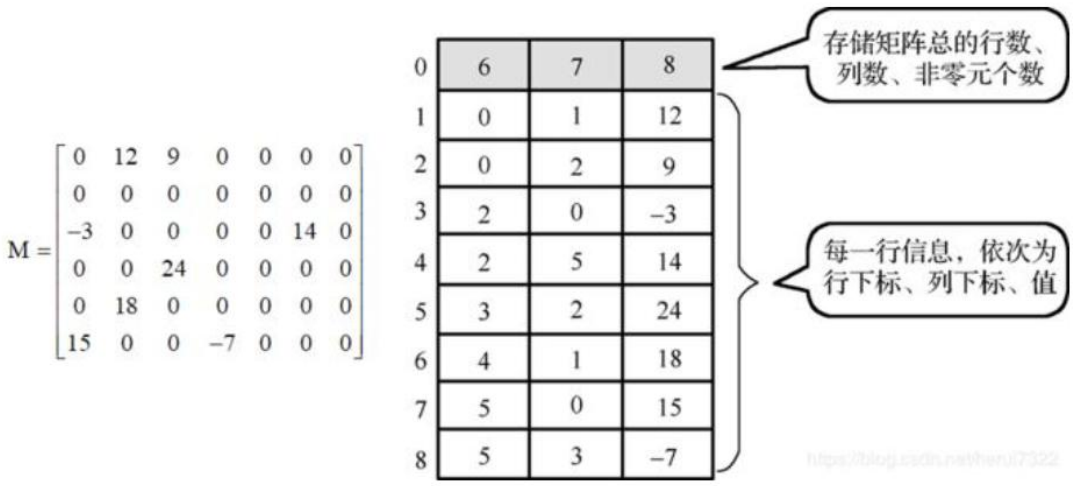
2、 压缩存储可以节省储存空间，避免资源的不必要的浪费，在数据序列化到磁盘时，压缩存储可以提高 IO 效率。

稀疏数组处理方法：

1、 记录数组一共有几行几列，有多少个不同的值

2、 把具有不同值元素的行列及值记录在一个小规模的数组中，从而缩小程序的规模。

4.1.3 实现步骤



4.1.4 练习

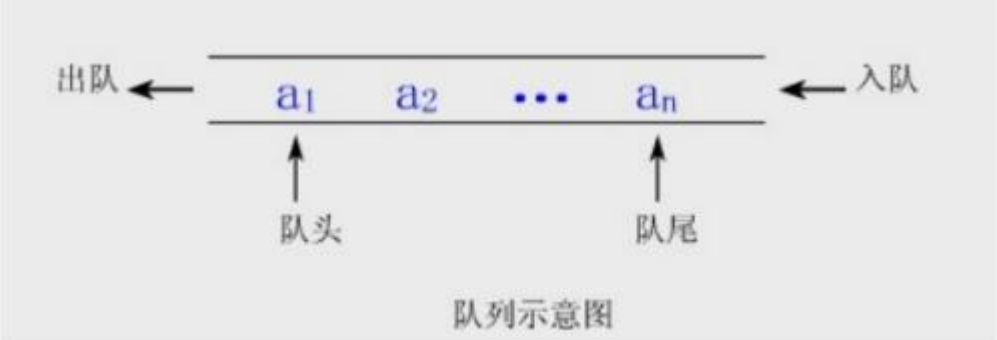
1、 使用稀疏数组来保留如上二维数组（棋盘）

2、 把稀疏数组存盘可以重新恢复原来的二维数组

4.2 队列

4.2.1 队列介绍

队列是一种特殊的线性表，特殊之处在于它只允许在表的前端（front）进行删除操作，而在表的后端（rear）进行插入操作，和栈一样，队列是一种操作受限制的线性表。进行插入操作的端称为队尾，进行删除操作的端称为队头

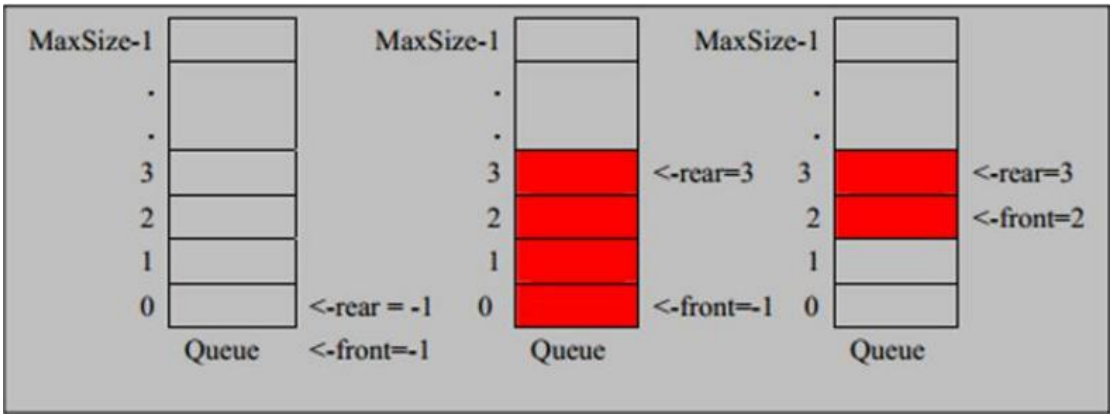


队列示意图

1、 队列是一个有序列表，可以用数　　组和链表来实现

2、 队列先进先出，即是先入队列的数据最先被取出，后存入的数据后取出。

4.2.2 数组模拟队列场景




```
/**
 * author:韩国庆
 * date:2021/1/28 16:38
 * version:1.0
 */
public class ArrayQueue {
    private int maxSize;
    private int headPoint;
    private int lastPoint;
    private int[] array;

    public ArrayQueue(int maxSize){
        this.maxSize = maxSize;
        array = new int[maxSize];
        headPoint = -1;
        lastPoint = -1;
    }

    public boolean isFull(){
        return lastPoint == maxSize-1;
```

```
    }

    public boolean isEmpty(){
        return lastPoint == headPoint;
    }

    public void addQueue(int n){
        if (isFull()){
            System.out.println("队列已满");
            return;
        }
        lastPoint++;
        array[lastPoint] = n;

    }

    public int getQueue(){
        if (isEmpty()){
            throw new RuntimeException("空队列");
        }
        headPoint++;
        return array[headPoint];
    }

    public void findQueue(){
        if (isEmpty()){
            throw new RuntimeException("空队列");
        }
        for (int i=0;i<array.length;i++){
            System.out.printf("array[%d]=%d\n",i,array[i]);
```

```
        }
    }
    /**
        查看队列头数据，不是取出来数据
     */
    public int headQueue(){
        if (isEmpty()){
            throw new RuntimeException("空队列");
        }
        return array[headPoint+1];
    }

}
```

5. 递归

5.1 递归概念

递归就是方法自己去调用自己，每次调用时传入的参数是不同的，递归有助于解决程序中复杂的问题，同时可以让代码更为简洁。

5.2 递归解决的问题

1、 可以解决各种数学问题，汉若塔，迷宫问题， 8 皇后问题等等

2、 各种算法的递归，如快排，归并排序，二分查找，分治算法等

5.3 递归的规则

1、执行一个方法时，就创建一个新的受保护的独立栈空间

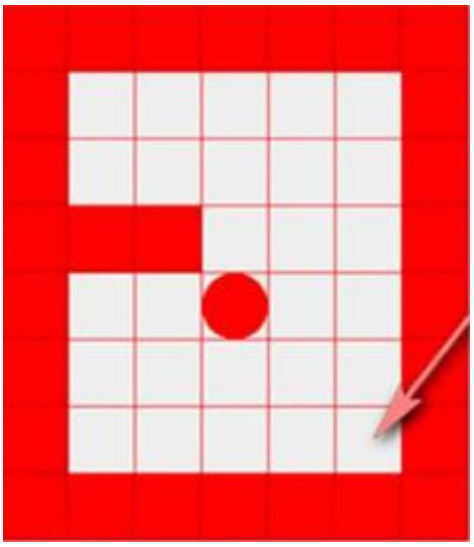
2、方法的局部变量是独立的，不会相互影响。

3、如果方法中使用的是引用型类型变量，比如数组，则会共享引用型的数据

4、递归必须向退出递归的条件接近，否则就是无线递归，会出现 StackOverflowError。

5、一个方法执行完毕后，或者遇到 return，会就返回数据，遵守谁调用就将结果返回给谁，同时当方法执行完毕或者返回时，该方法也就执行完毕。

5.4 递归（迷宫问题）



```
/**
 * author:韩国庆
 * date:2021/2/27 16:46
 * version:1.0
 */
public class Maze {
```



```
public static void main(String[] args) {

    int[][] map = new int[8][7];

    /**
     * 使用编号 1 表示上下左右都是墙
     */
    for (int i=0;i<7;i++){
        map[0][i] = 1;
        map[7][i] = 1;
    }

    for (int i=0;i<8;i++){
        map[i][0] = 1;
        map[i][6] = 1;
    }

    //设置阻挡墙
    map[3][1] = 1;
    map[3][2] = 1;

    System.out.println("输出地图情况");
    for (int i=0;i<8;i++){
        for (int j=0;j<7;j++){
            System.out.print(map[i][j]+" ");
        }
        System.out.println();
    }

    getRoute(map,1,1);
```



```
        //getRoute2(map,1,1);

        System.out.println("小球标识走过的地图情况");
        for (int i=0;i<8;i++){
            for (int j=0;j<7;j++){
                System.out.print(map[i][j]+" ");
            }
            System.out.println();
        }


    }

    /**
     * 回溯思想
     * 1、map 对象表示地图
     * 2、i 和 j 表示从那个位置出发 （1,1）
     * 3、如果小球能到 map[6][5]位置，则说明通路已经找到
     * 4、当 map[i][j]为 0 时，表示该点没有走过，当为 1 时表示墙，当时 2 时表示通过可以走，当时 3 时该点已经走过，
     * 但是走不通。
     * 5、走迷宫时，需要确定一个策略方法，下,右，上，左，如果该点走不通再回溯。
     *
     */
    public static boolean getRoute(int[][] map,int i,int j){
        if (map[6][5] == 2){
            return true;
        }else {
            if (map[i][j] == 0){//如果没有走过该点
```

```
            //按照策略走，下，右，上，左
            map[i][j] = 2;
            if (getRoute(map,i+1,j)){ //往下走
                return true;
            }else if (getRoute(map,i,j+1)){ //往右走
                return true;
            }else if (getRoute(map,i-1,j)){
                return true;
            }else if (getRoute(map,i,j-1)){
                return true;
            }else {
                //说明该路是死路，走不通
                map[i][j] = 3;
                return false;
            }

        }else {//如果 map[i][j]不是 0，而是 1,2,3 呢
            return false;
        }
    }

}

public static boolean getRoute2(int[][]map,int i,int j){
    if (map[6][5] == 2){
        return true;
    }else{
        if (map[i][j] == 0){//表示该路没有走过
            map[i][j] = 2;//假设该点是可以通走的
            if (getRoute(map,i-1,j)){//上
```



```
                return true;
            }else if (getRoute(map,i,j+1)){//右
                return true;
            }else if (getRoute(map,i+1,j)){//下
                return true;
            }else if (getRoute(map,i,j-1)){//左
                return true;
            }else {
                //说明该路是死路，走不通过
                map[i][j] = 3;
                return false;
            }

        }else {
            //如果 map[i][j] 不是 0，而是 1,2,3
            return false;
        }
    }

}
```


6. 排序算法

6.1 排序算法介绍

排序也称之为排序算法（Sort Algorithm），是讲一组数据以指定的顺序进行排列的过程。

6.1.1 排序分类

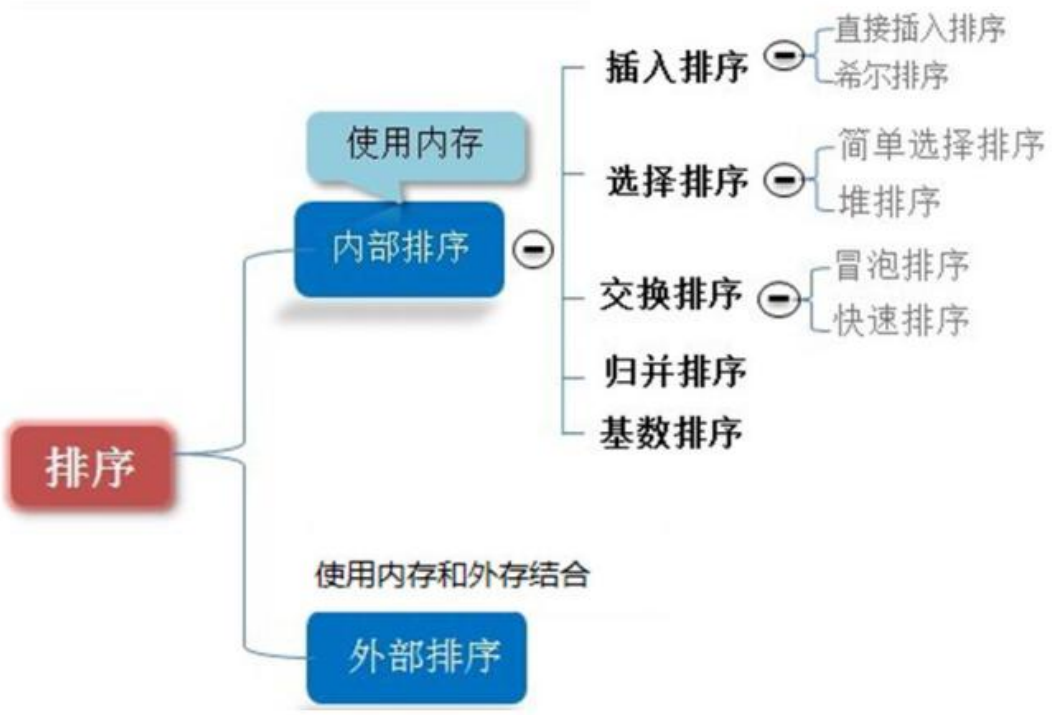
内部排序：

将所有的数据都加载到内部存储器上进行排序

外部排序：

数据量多大，无法全部加载到内存中，需要借助外部存储（文档）进行排序

算法分类图：



6.2 算法时间效率

6.2.1 度量一个程序执行时间两种方法

事后统计方法

这种方法可行，但是存在于两个问题：一是要想对设计的算法运行性能进行评测就需要实际去运行该程序，而是所得时间统计量依赖于计算机硬件、软件等因素，这种方式要在同一台计算机的相同状态下运行，才能比较出哪一个算法速度更快，更好。

事前估算方法

通过分析某一个算法的时间复杂度来判断哪个算法更优，更好。

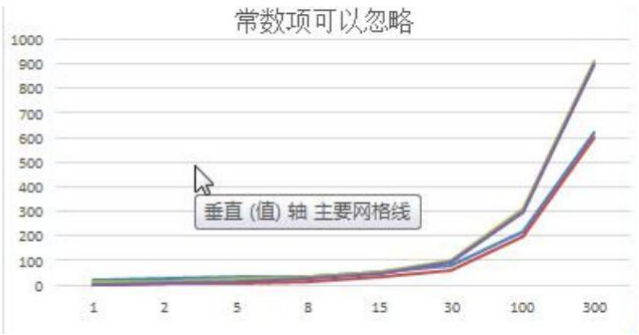
6.2.2 时间频度

时间频度：一个算法花费的时间与算法中语句的执行次数成正比，哪一个算法中语句执行次数多，那么他所花费的时间就会多。 **一个算法中语句执行次数称之为语句频度或时间频度**。记为 T(n)

```
int sum = 0;
for(int i=1;i<=n;i++){
  sum+=i;
}
n = 100;
T(n) = n+1;
n*2/100
T(n)  =1;
```

忽略常数项：

	T(n)=2n+20	T(n)=2*n	T(3n+10)	T(3n)
1	22	2	13	3
2	24	4	16	6
5	30	10	25	15
8	36	16	34	24
15	50	30	55	45
30	80	60	100	90
100	220	200	310	300
300	620	600	910	900



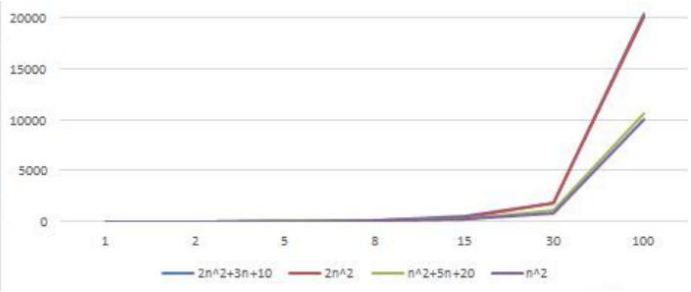
结论：

1、 2n+20 和 2n 随着 n 变大，执行曲线无线接近，20 可以忽略

2、 3n+10 和 3n 随着 n 变大，执行曲线无限接近，10 可以忽略

忽略低次项：

	T(n)=2n^2+3n+10	T(2n^2)	T(n^2+5n+20)	T(n^2)
1	15	2	26	1
2	24	8	34	4
5	75	50	70	25
8	162	128	124	64
15	505	450	320	225
30	1900	1800	1070	900
100	20310	20000	10520	10000



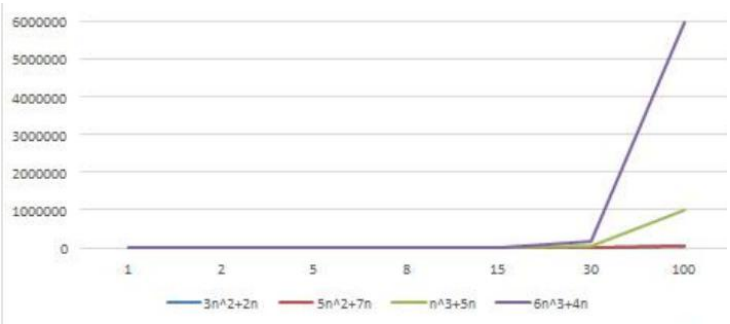
结论：

1、2n^2+3n+10 和 2n^2 随着 n 变大，执行曲线无线接近，可以忽略 3n+10

2、n^2+5n+20 和 n^2 随着 n 变大，执行曲线无线接近，可以忽略 5n+20

忽略系数：

	T(3n^2+2n)	T(5n^2+7n)	T(n^3+5n)	T(6n^3+4n)
1	5	12	6	10
2	16	34	18	56
5	85	160	150	770
8	208	376	552	3104
15	705	1230	3450	20310
30	2760	4710	27150	162120
100	30200	50700	1000500	6000400



结论：

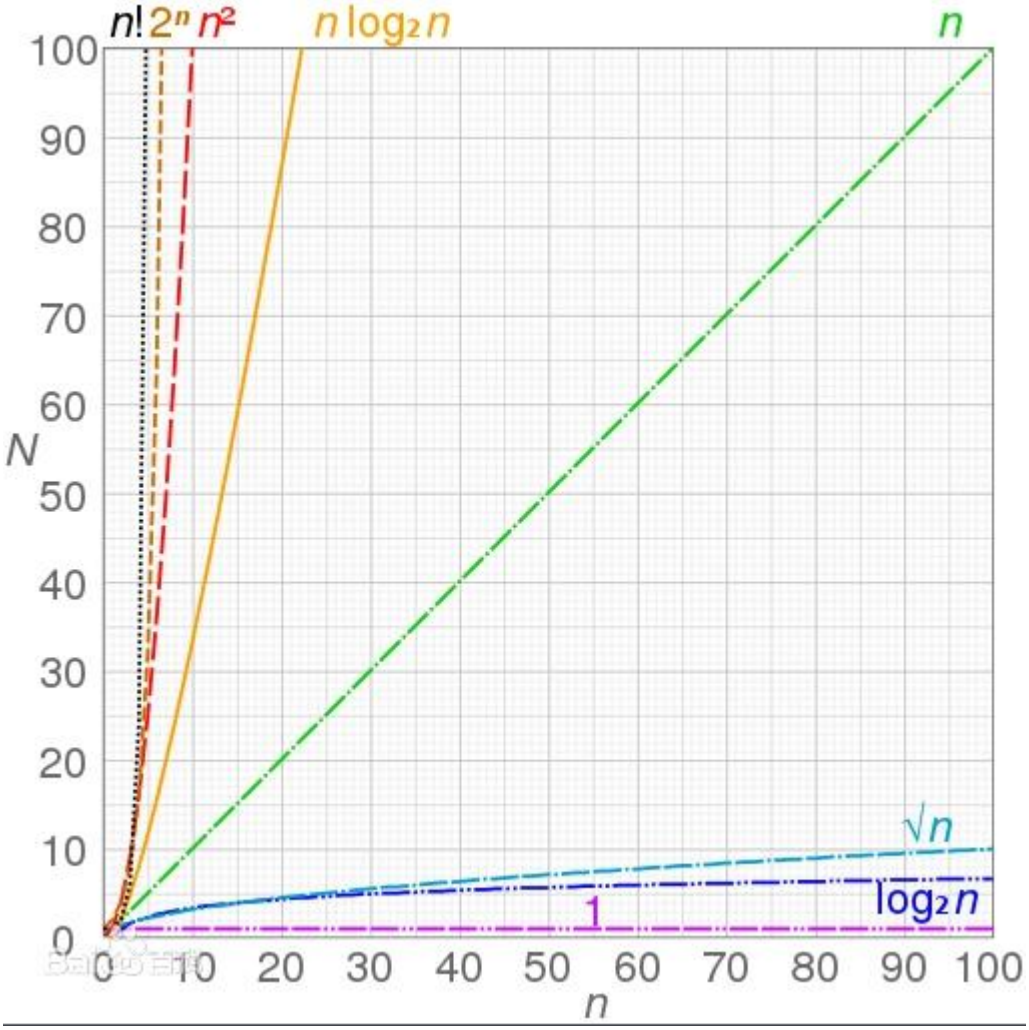
1、 随着 n 值变大，5n^2+7n 和 3n^2+2n 执行区间重合，说明这种情况下 5 和 3 可以忽略

2、 而 n^3+5n 和 6n^3+4n 执行区间分离，说明多少次方式关键

6.2.3 时间复杂度

在计算机科学中，时间复杂性，又称时间复杂度，算法的时间复杂度是一个函数，它定性描述该算法的运行时间。这是一个代表算法输入值的字符串的长度的函数。时间复杂度常用大

0 符号表述，不包括这个函数的低阶项和首项系数。使用这种方式时，时间复杂度可被称为是渐近的，渐近时间复杂度又称之为时间复杂度。



6.2.4 常见的时间复杂度

1、 常数时间

若对于一个算法，的上界与输入大小无关，则称其具有**常数时间**，记作 O(1)时间

T(n) =1

2、 对数时间

若算法的 $T(n)$ =O(log$n$)，则称其具有**对数时间**。

```
int I = 1;
while(i<n){
 i=i*2;
```

```
}
x=log2^n    O(log2n)
```

3、 幂对数时间

对于某个常数 *k*，若算法的 *T*(*n*) = O((log*n*))，则称其具有**幂对数时间**

4、 次线性时间

对于一个算法，若其匹配 *T*(*n*) = o(*n*)，则其时间复杂度为**次线性时间**（sub-linear time 或 sublinear time）。

5、 线性时间

如果一个算法的时间复杂度为 O(*n*)，则称这个算法具有线性时间，或 O(*n*)时间。

```
for(i=1;i<=n;++i){
  j=I;
j++;
}
```

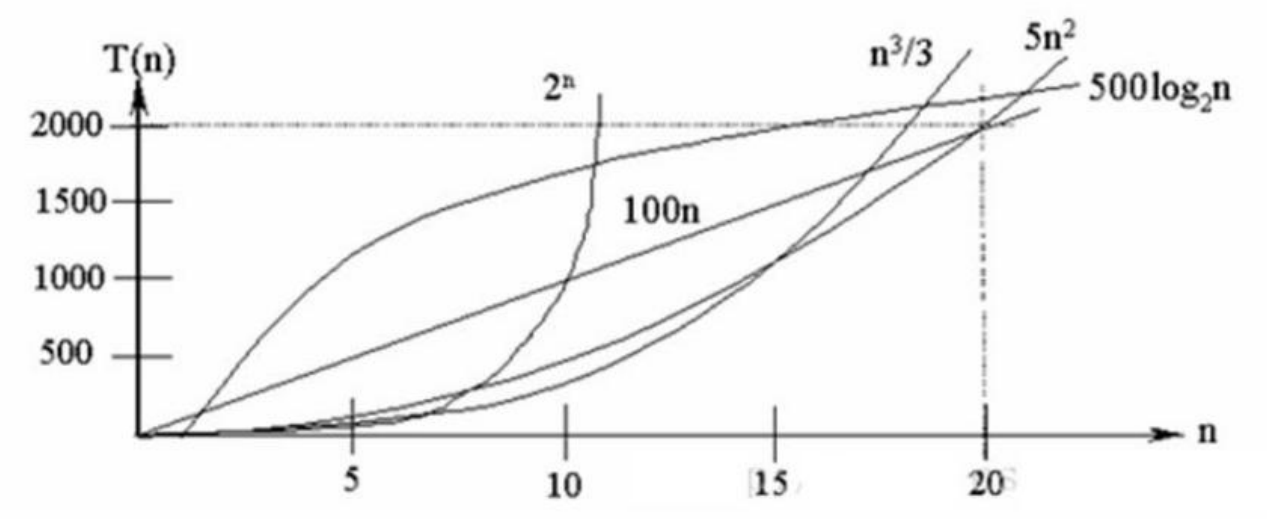
6、 线性对数时间

若一个算法时间复杂度 T(n) = O(nlog n)，则称这个算法具有线性对数时间。

7、 指数时间

若 *T*(*n*) 是以 2 为上界，其中 poly(*n*) 是 *n* 的多项式，则算法被称为**指数时间**

常见的时间复杂度对应图：



结论：

1、常见的算法时间复杂度由小到大依次为：
O(1)<O(log2n)<O(n)<O(nlog2n)<O(n2)<O(n3)<O(nk)<O(2n)

2、尽可能的避免使用指数阶的算法

6.2.5 平均和最坏时间复杂度

平均时间复杂度是指所有可能的输入实例均以等概率的出现情况下得到算法的运行时间

最坏时间复杂度，一般讨论的时间复杂度均是最坏情况下的时间复杂度，这样做的原因是最坏情况下的时间复杂度是算法在任何输入实例上运行的界限，这就保证了算法的运行时间不会比最坏情况更长。

平均时间复杂度和最坏时间复杂度是否一样，这就需要根据算法不同而不同了。

排序算法	平均时间复杂度	最坏时间复杂度	空间复杂度	是否稳定
冒泡排序	$O(n^2)$	$O(n^2)$	$O(1)$	是
选择排序	$O(n^2)$	$O(n^2)$	$O(1)$	不是
直接插入排序	$O(n^2)$	$O(n^2)$	$O(1)$	是
归并排序	$O(nlogn)$	$O(nlogn)$	$O(n)$	是
快速排序	$O(nlogn)$	$O(n^2)$	$O(logn)$	不是
堆排序	$O(nlogn)$	$O(nlogn)$	$O(1)$	不是
希尔排序	$O(nlogn)$	$O(n^s)$	$O(1)$	不是
计数排序	$O(n+k)$	$O(n+k)$	$O(n+k)$	是
基数排序	$O(N*M)$	$O(N*M)$	$O(M)$	是

6.3 基数排序

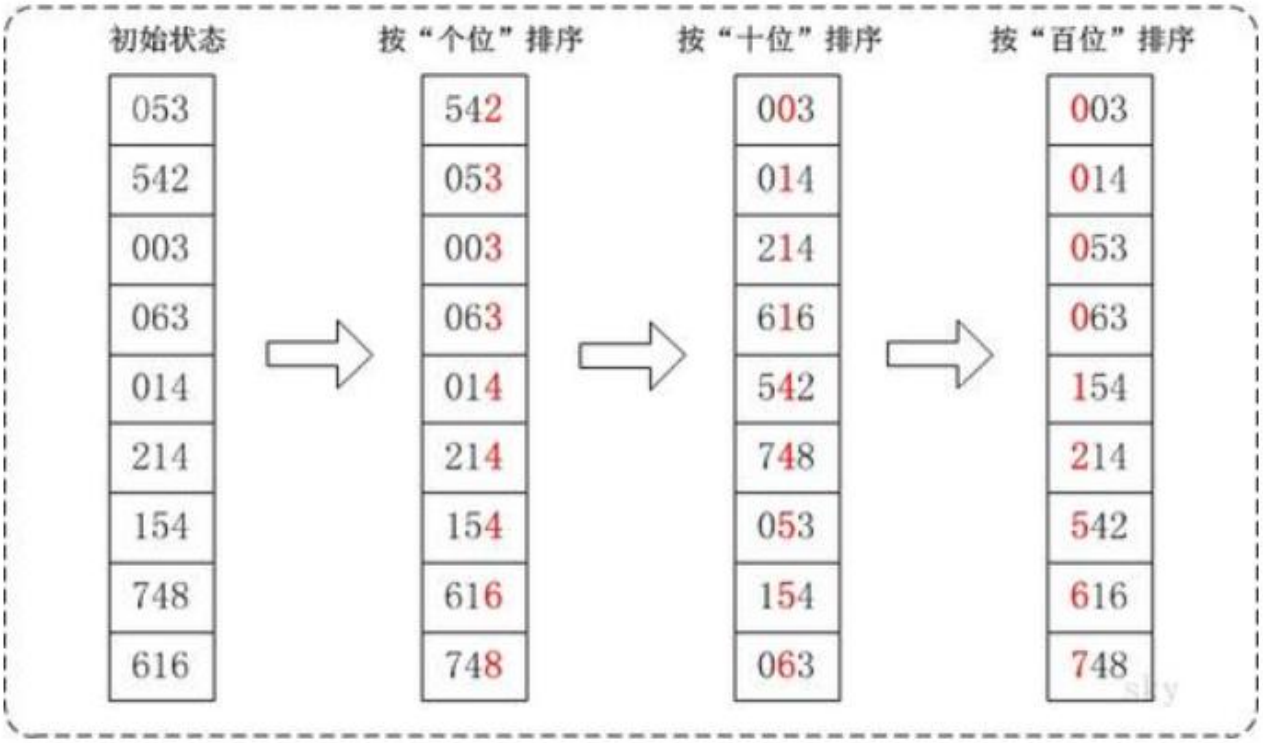
6.3.1 介绍

基数排序（radix sort）属于“分配式排序”（distribution sort），又称“桶子法”（bucket sort）或 bin sort，顾名思义，它是透过键值的部份资讯，将要排序的元素分配至某些“桶”中，藉以达到排序的作用，**基数排序法是属于稳定性的排序**，其时间复杂度为 O(nlog(r)m)

基数排序是 1887 年赫尔曼、何乐礼发明的。思想是讲整数按位数切割成不同的数字，然后按每个位数分别比较。

6.3.2 思想

讲所有的待比较数值统一设置为同样的数位长度，位数比较短的数前面补零，然后从最地位开始依次进行一次排序，这样从最低位排序一直到最高位排序完成以后，数列就变成一个有序序列。



6.3.3 基数排序代码实现

讲数组 53,542,3,63,14,214,154,748,616 进行排序

```
/**
 * author:韩国庆
 * date:2021/3/8 16:37
 * version:1.0
 */
public class BasicSort {

    public static void main(String[] args) {

        int[] arrays = new int[]{53, 542, 3, 63, 14, 214, 154, 748, 616};
        Date before = new Date();
        SimpleDateFormat  simpleDateFormat  =  new  SimpleDateFormat("yyyy-MM-dd
```

```
HH:mm:ss:SS");
        System.out.println("排序前：" + simpleDateFormat.format(before));


        sort(arrays);


        Date after = new Date();
        System.out.println("排序后：" + simpleDateFormat.format(after));

    }

    public static void sort(int[] arrays) {

        /**
         * 得到数组中最大位数
         */
        //假设第一个位就是最大数
        int max = arrays[0];
        for (int i = 0; i < arrays.length; i++) {
            if (arrays[i] > max) {
                max = arrays[i];
            }
        }

        //得到最大位数的位数
        int maxLength = (max + "").length();

        /**
         * 定义一个二维数组，大小是 10，表示 10 个桶，每一个桶则是一个数组
```

```
         * 二维数组包含 10 个一维数组
         * 为了防止在放入数据时数据溢出， 则把每个桶设置为 arrays.length 的大小。
         * 基数排序则是典型的空间换时间的算法
         */
        int[][] bucket = new int[10][arrays.length];

        /**
         * 为了记录每个桶中实际存放了多少个数据，我们定义一个一维数组来记录各个桶的每次放入的数据个数
         * 比如：bucketElementCount[0] 记录着 bucket[0]桶中的元素个数
         */
        int[] bucketElementCount = new int[10];


        for (int i = 0, n = 1; i < maxLength; i++, n *= 10) {

            /**
             * 循环取出每一位，比如先去个位，十位，百位...
             */
            for (int j = 0; j < arrays.length; j++) {
                /**
                 * 取出每一个元素对应的位数，
                 */
                int locationElement = arrays[j] / n % 10;

                /**
                 * 放入对用的桶中
                 */
                bucket[locationElement][bucketElementCount[locationElement]]    =
arrays[j];
```

```
            bucketElementCount[locationElement]++;

            /**
             * 根据一维数组的下标依次取出数据，放入原数组中
             */
            int index = 0;
            /**
             * 遍历每一个桶，将数组放入原数组中
             */
            for (int k = 0; k < bucketElementCount.length; k++) {
                if (bucketElementCount[k] != 0) {

                    //循环第 k 个桶（第 k 个一维数组）
                    for (int l = 0; l < bucketElementCount[k]; l++) {
                        arrays[index++] = bucket[k][l];
                    }
                }

                //取出后需要将每一个 bucketElementCount[k]置为 0
                bucketElementCount[k] = 0;

            }

        }

    }

    /**
     * 第一轮比较  个位数比较
     */
```

```
        for (int j=0;j<arrays.length;j++){
            int locationElement = arrays[j]/1%10;
            bucket[locationElement][bucketElementCount[locationElement]]        =
arrays[j];
            bucketElementCount[locationElement]++;
        }

        /**
         * 按照这个顺序依次取出放入原数组
         */
        int index = 0;
        for (int k=0;k<bucketElementCount.length;k++){
            if (bucketElementCount[k] !=0){
                for (int l = 0;l<bucketElementCount[k];l++){
                    arrays[index++] = bucket[k][l];
                }
            }
            //将 bucketElementCount[k] 置为 0
            bucketElementCount[k] = 0;
        }

        System.out.println("第一轮：对个位排序处理  "+ Arrays.toString(arrays));

        /**
         * 第二轮：针对十位进行排序
         */
        for (int j=0;j<arrays.length;j++){
            int locationElement = arrays[j]/10%10;
            bucket[locationElement][bucketElementCount[locationElement]]        =
arrays[j];
```

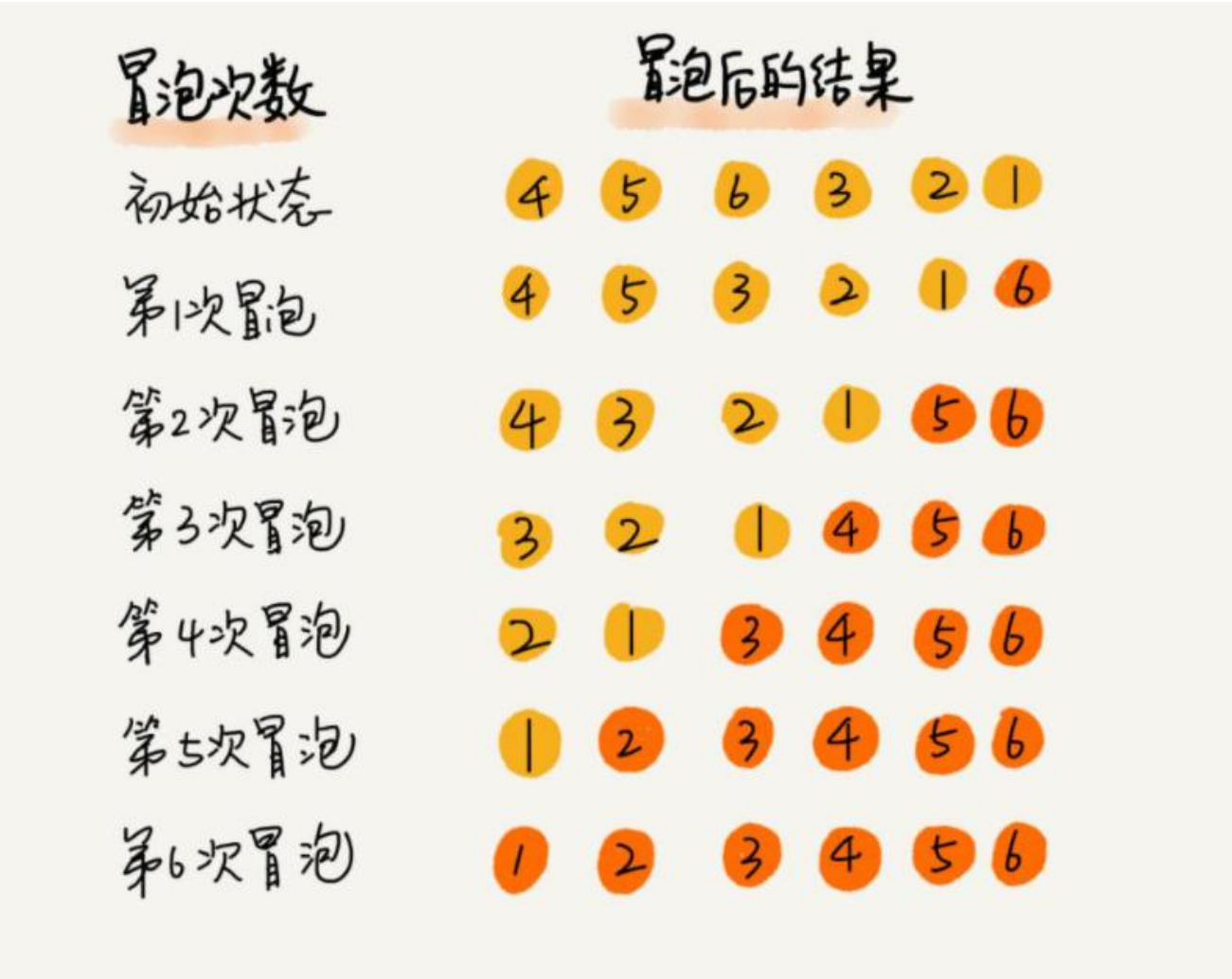
```
            bucketElementCount[locationElement]++;
        }

        index = 0;
        for (int k=0;k<bucketElementCount.length;k++){
            if (bucketElementCount[k] !=0){
                for (int l = 0;l<bucketElementCount[k];l++){
                    arrays[index++] = bucket[k][l];
                }
            }
            //将 bucketElementCount[k] 置为 0
            bucketElementCount[k] = 0;
        }
        System.out.println("第二轮：对十位排序处理  "+ Arrays.toString(arrays));

        /**
         * 第三轮：针对百位进行排序
         */
        for (int j=0;j<arrays.length;j++){
            int locationElement = arrays[j]/100%10;
            bucket[locationElement][bucketElementCount[locationElement]]        =
arrays[j];
            bucketElementCount[locationElement]++;
        }

        index = 0;
        for (int k=0;k<bucketElementCount.length;k++){
            if (bucketElementCount[k] !=0){
                for (int l = 0;l<bucketElementCount[k];l++){
                    arrays[index++] = bucket[k][l];
```

```
                }
            }
            //将 bucketElementCount[k] 置为 0
            bucketElementCount[k] = 0;
        }
        System.out.println("第三轮：对百位排序处理  "+ Arrays.toString(arrays));
    }
}
```

6.4 冒泡排序

6.4.1 介绍

冒泡排序的思想是通过对待排序序列从前往后依次比较相邻元素值，若发现逆序则交换，使值较大的元素从前逐步移向后面，就想水中气泡。



特点：

1、 需要循环 array.length-1 次 外层循环

2、 每次排序的次数逐步递减

3、 也可能存在本次排序没有发生变化

6.4.2 冒泡排序实际应用

```
/**
 * author:韩国庆
 * date:2021/3/9 16:04
```

```
 * version:1.0
 */
public class BubblingSort {

    public static void main(String[] args) {

        int[] arrays = new int[]{2,4,1,3,8,5,9,6,7};


        int temp = 0;
        boolean flag = false;
        for (int i=0;i<arrays.length-1;i++){
            for (int j=0;j<arrays.length-1-i;j++){
                if (arrays[j]>arrays[j+1]){
                    flag=true;
                    temp = arrays[j];
                    arrays[j] = arrays[j+1];
                    arrays[j+1] = temp;

                }
            }
            if (!flag){//在排序过程中，没有发生一次交换
                break;
            }else {
                flag = false;
            }

        }

        System.out.println(Arrays.toString(arrays));
```

```
    }

}
```

6.5 快速排序

6.5.1 介绍

快速排序是对冒泡排序的一种改进。通过一趟排序将要排序的数据分割成独立的两部分，其中一部分的所有数据都比另一部分所有的数据都要小，然后再按此方法对这两部分数据分别进行快速排序，整个排序过程可以递归进行，以此达到整个数据变成有序序列

6.5.2 示意图

代码实现

```
/**
 * author:韩国庆
 * date:2021/3/9 16:53
 * version:1.0
 */
public class QuickSort {

    public static void main(String[] args) {
        int[] arrays = new int[]{2,9,4,7,3,1,6,5};
        sort(arrays,0,arrays.length-1);
        System.out.println(Arrays.toString(arrays));

    }
    public static void sort(int[] arrays,int left,int right){
        int l = left;//左下标
        int r = right;//右下标

        int pivot = arrays[(left+right)/2];//中间值
        int temp = 0;
        /**
         * 循环的目的是让比 pivot 小的值放在左边，比 pivot 大的值放在右边
         */
        while (l<r){
            //在 pivot 左边找到大于等于 pivot 值，再退出
            while (arrays[l]<pivot){
                l+=1;
            }
            /**
             * 在 pivot 右边找小于等于 pivot 值，再退出
```

```
        */
        while (arrays[r]>pivot){
            r-=1;
        }

        /**
         * 如果 l>=r 说明左边全部是小于 pivot 的值，右边全部是大于 pivot 的值
         */
        if (l>=r){
            break;
        }
        /**
         * 交换位置
         */
        temp = arrays[l];
        arrays[l] = arrays[r];
        arrays[r] = temp;
        /**
         * 交换后发现 arrays[l] == piovt 值，则 r--;
         */
        if (arrays[l]==pivot){
            r-=1;
        }
        /**
         * 交换后发现 arrays[r] == piovt 值，则 l++;
         */
        if (arrays[r] == pivot){
            l+=1;
        }
```

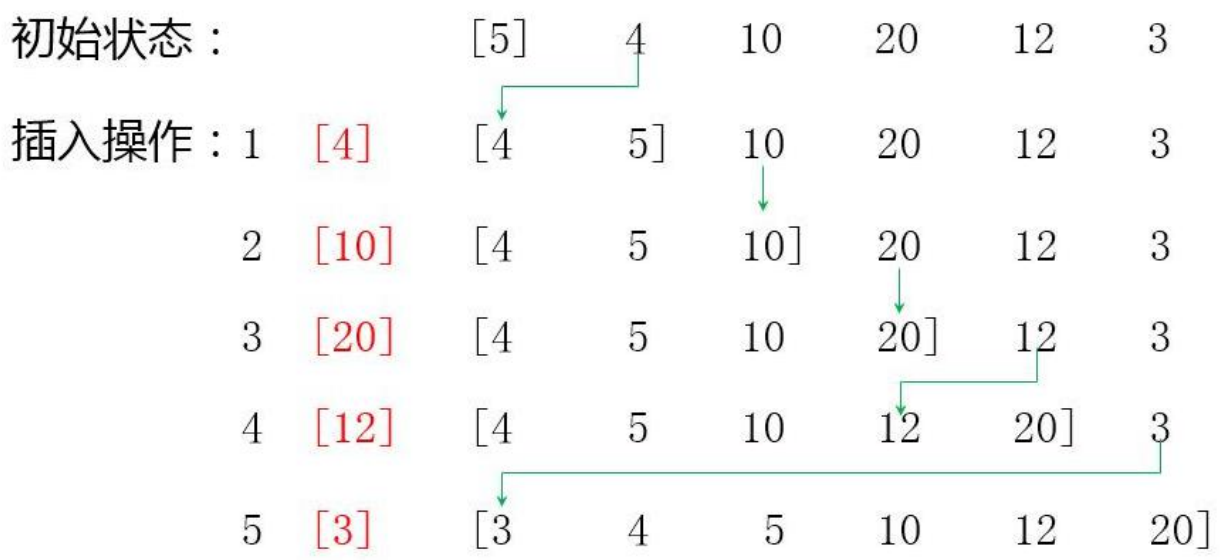
```
            /**
             * 如果 l==r 必须 l++，r--,否则出现溢栈
             */
            if (l==r){
                l+=1;
                r-=1;
            }

            if (left<r){
                sort(arrays,left,r);
            }
            if (right>l){
                sort(arrays,l,right);
            }
        }

    }
}
```

6.6 插入排序

6.6.1 介绍

插入排序属于内部排序，是对于排序的元素以插入的方式寻找该元素的适当位置，已达到排序的目的。

6.6.2 示意图



代码实现

```
/**
 * author:韩国庆
 * date:2021/3/12 14:44
 * version:1.0
 */
public class InsertSort {

    public static void main(String[] args) {

        int[] arrays = new int[]{2,6,4,1,3,7,5};
        sort(arrays);

        System.out.println(Arrays.toString(arrays));


    }
```

```
    public static void sort(int[] arrays){

        int[] arr = arrays;
        int temp;
        for (int i=1;i<arr.length;i++){
            System.out.println("第  "+i+"  次");
            for (int j=i;j>=1;j--){
                if (arr[j]<arr[j-1]){
                    temp = arr[j];
                    arr[j] = arr[j-1];
                    arr[j-1] = temp;
                }else {
                    break;
                }
            }
        }


    }

}
```

6.7 选择排序

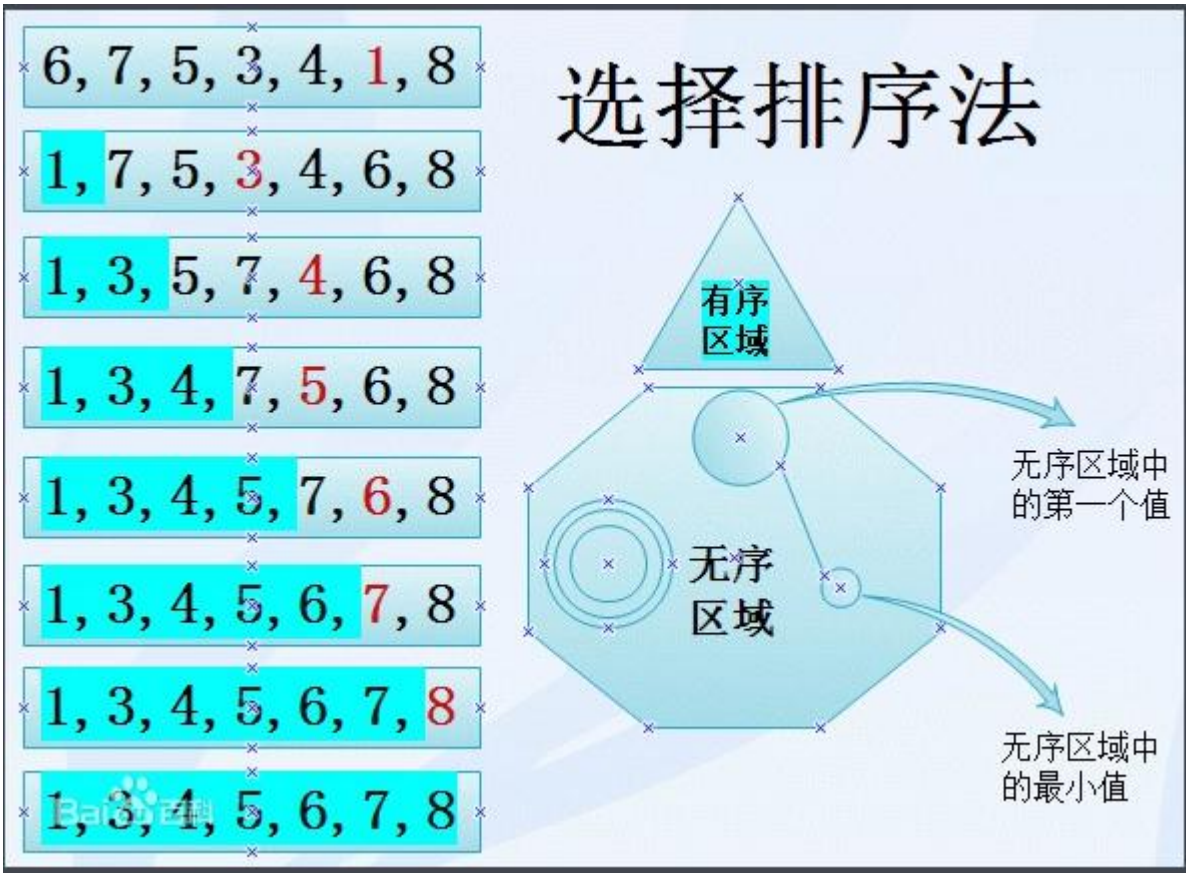
6.7.1 介绍

选择排序也属于内部排序法，是从欲排序的数据中按指定的规则选择某一个元素，再以规则交换位置后达到的排序目的。

它的工作原理是：

第一次从待排序的数据元素中选出最小（或最大）的一个元素，存放在序列的起始位置，然后再从剩余的未排序元素中寻找到最小（大）元素，然后放到已排序的序列的末尾。以此类推，直到全部待排序的数据元素的个数为零。选择排序是不稳定的排序方法。

6.7.2 示意图



6.7.3 源码实现

```
/**
 * author:韩国庆
 * date:2021/3/12 15:02
```

```
 * version:1.0
 */
public class SelectSort {

    public static void main(String[] args) {

        int[] arrays = new int[]{2,6,4,1,3,7,5};
        sort(arrays);

        System.out.println(Arrays.toString(arrays));

    }

    public static void sort(int[] arrays){
        for (int i=0;i<arrays.length;i++){
            System.out.println("      "+i);
            int index = i;
            for (int j=arrays.length-1;j>i;j--){
                if (arrays[j]<arrays[index]){
                    index = j;
                    int temp = 0;
                    temp = arrays[j];
                    arrays[j] = arrays[i];
                    arrays[i] = temp;
                }
            }
        }
    }
}
```

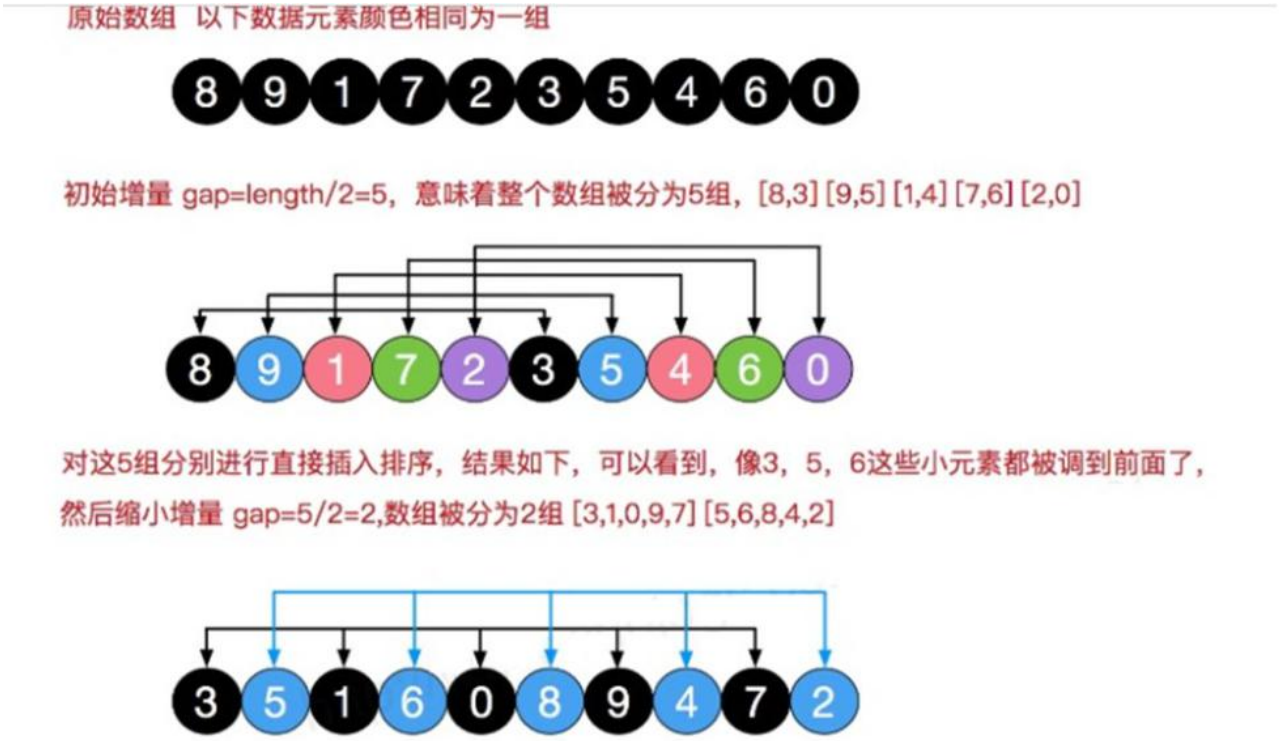
6.8 希尔排序

6.8.1 介绍

希尔排序(Shell's Sort)是插入排序的一种又称“缩小增量排序”（Diminishing Increment Sort），是插入排序算法的一种更高效的改进版本。希尔排序是非稳定排序算法。该方法因D.L.Shell 于 1959 年提出而得名。

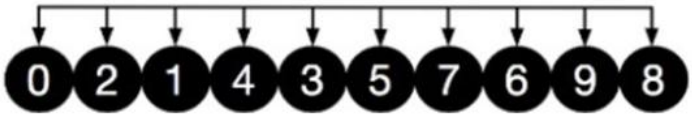
希尔排序是把记录按下标的一定增量分组，对每组使用直接插入排序算法排序；随着增量逐渐减少，每组包含的关键词越来越多，当增量减至 1 时，整个文件恰被分成一组，算法便终止。

6.8.2 示意图：



对以上2组再分别进行直接插入排序，结果如下，可以看到，此时整个数组的有序程度更进一步啦。

再缩小增量gap=2/2=1，此时，整个数组为1组[0,2,1,4,3,5,7,6,9,8]，如下



经过上面的“宏观调控”，整个数组的有序化程度成果喜人。

此时，仅仅需要对以上数列简单微调，无需大量移动操作即可完成整个数组的排序。

6.8.3 源码实现

```
/**
 * author:韩国庆
 * date:2021/3/13 15:12
 * version:1.0
 */
public class ShellSort {

    public static void main(String[] args) {
        int[] array = {7,3,2,5,9,4,8,1,6};
       // shellSort(array); //交换法
        shellSort2(array);//移动法
    }
```

```
/**
 * 交换方式
 * @param arrays
 */
public static void shellSort(int[] arrays){
    int temp = 0;
    int count = 0;

    for (int gap = arrays.length/2;gap>0;gap/=2){
        for (int i=gap;i<arrays.length;i++){
            for (int j=i-gap;j>=0;j-=gap){
                if (arrays[j]>arrays[j+gap]){
                    temp=arrays[j];
                    arrays[j] = arrays[j+gap];
                    arrays[j+gap] = temp;
                }
            }
        }
        System.out.println("第"+(++count)+"轮    =    "+ Arrays.toString(arrays));
    }
}

/**
 * 移位法
 */
public static void shellSort2(int[] arrays){
    for (int gap=arrays.length/2;gap>0;gap/=2){
        for (int i=gap;i<arrays.length;i++){
            int j=i;
```

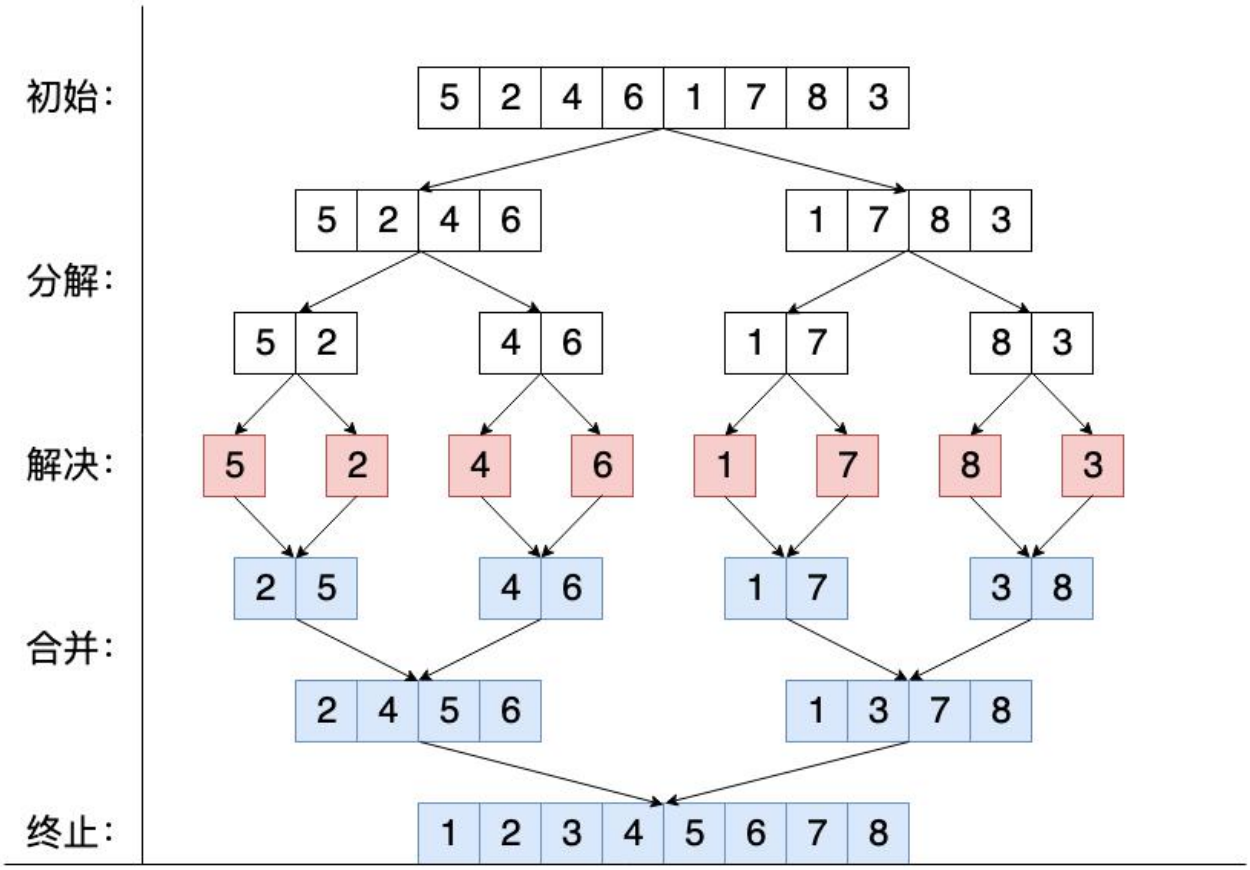
```
                int temp = arrays[j];
                if (arrays[j]<arrays[j-gap]){
                    while (j-gap>=0 && temp<arrays[j-gap]){
                        //移动
                        arrays[j] = arrays[j-gap];
                        j-=gap;
                    }
                    arrays[j] = temp;
                }
            }
        }

        System.out.println(Arrays.toString(arrays));
    }
}
```

6.9 归并排序

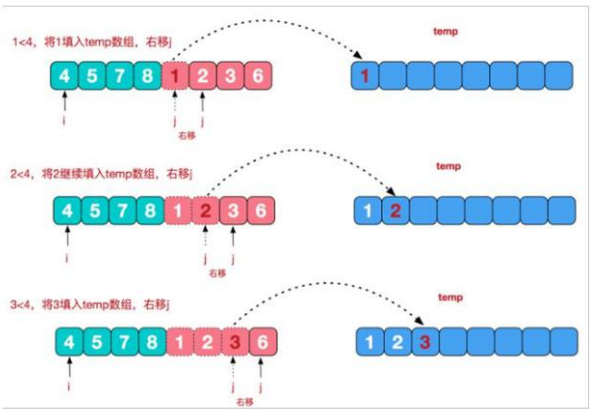
6.9.1 介绍

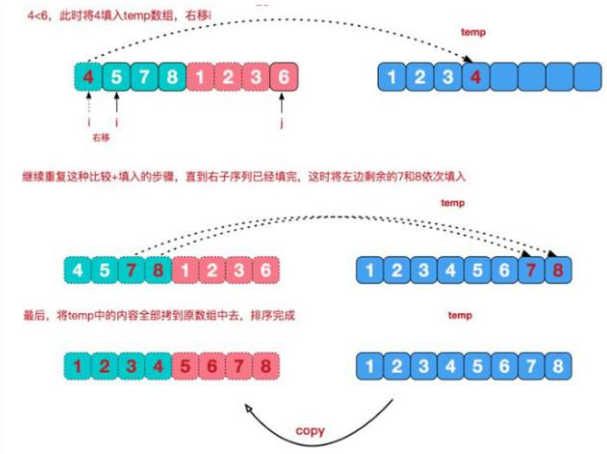
归并排序（Merge Sort）是建立在归并操作上的一种有效，稳定的排序算法，该算法是采用分治法（Divide and Conquer）的一个非常典型的应用。将已有序的子序列合并，得到完全有序的序列；即先使每个子序列有序，再使子序列段间有序。若将两个有序表合并成一个有序表，称为二路归并。

6.9.2 示意图



我们需要将两个已经有序的子序列合并成一个有序序列，比如上图最后一次合并，将[2,4,5,6]和[1,3,7,8]已经有序的子序列合并最终序列[1,2,3,4,5,6,7,8]





6.9.3 源码实现

```
/**
 * author:韩国庆
 * date:2021/3/15 14:29
 * version:1.0
 */
public class MergetSort {

    public static void main(String[] args) {
        int[] arrays = {8,4,7,2,3,6,1,5};

        /**
         * 定义一个临时数组
         */
        int[] temp = new int[arrays.length];
        mSort(arrays,0,arrays.length-1,temp);
        System.out.println(Arrays.toString(arrays));
    }

    public static void mSort(int[] array,int left,int right,int[] temp){
        if (left<right){
            int mid = (left+right)/2;

            /**
             * 向左进行分解
             */
            mSort(array,left,mid,temp);
```

```
            /**
             * 向右分解
             */
            mSort(array,mid+1,right,temp);

            /**
             * 合并
             */
            sort(array,left,right,mid,temp);
        }
    }

    public static void sort(int[] arrays,int left,int right,int mid,int[] temp){
        /**
         * 初始化 i，指向左边序列的初始索引值
         */
        int i = left;
        /**
         * 初始化 j，指向左边序列初始值
         */
        int j = mid+1;

        /**
         * 初始化  t，指向数组的当前索引
         */
        int t = 0;

        /**
         * 先把左右两边的数据按照有序序列方式填充到 temp 数组中
         * 直至左右两边的有序序列，有一边处理完毕为止
```

```
        */
        while (i<=mid && j<=right){

            /**
             * 如果左边的有序序列的当前元素小于等于右边有序序列的当前元素，即让左边的当前元素填充到 temp 数组中
             然后  t++,i++
             */
            if (arrays[i]<=arrays[j]){
                temp[t] = arrays[i];
                t+=1;
                i+=1;
            }else {
                temp[t] = arrays[j];
                t+=1;
                j+=1;
            }
        }
        /**
         * 把剩余数据的一边的数据一次全部填充到 temp；
         */
        while (i<=mid){
            temp[t] = arrays[i];
            t+=1;
            i+=1;
        }

        while (j<=right){
            temp[t] = arrays[j];
            t+=1;
```

```
            j+=1;
        }

        /**
         * 将 temp 数组中的元素拷贝至 arrays 数组中。这里不是每次拷贝所有的元素
         */
        t=0;
        int tempIndex = left;
        while (tempIndex<=right){
            arrays[tempIndex] = temp[t];
            t+=1;
            tempIndex +=1;
        }
    }
}
```

7. 查找算法

在 java 程序中一般常用到四种查找方式。**顺序查找（线性），二分查找，插值查找，斐波那契查找**

7.1 线性查找算法

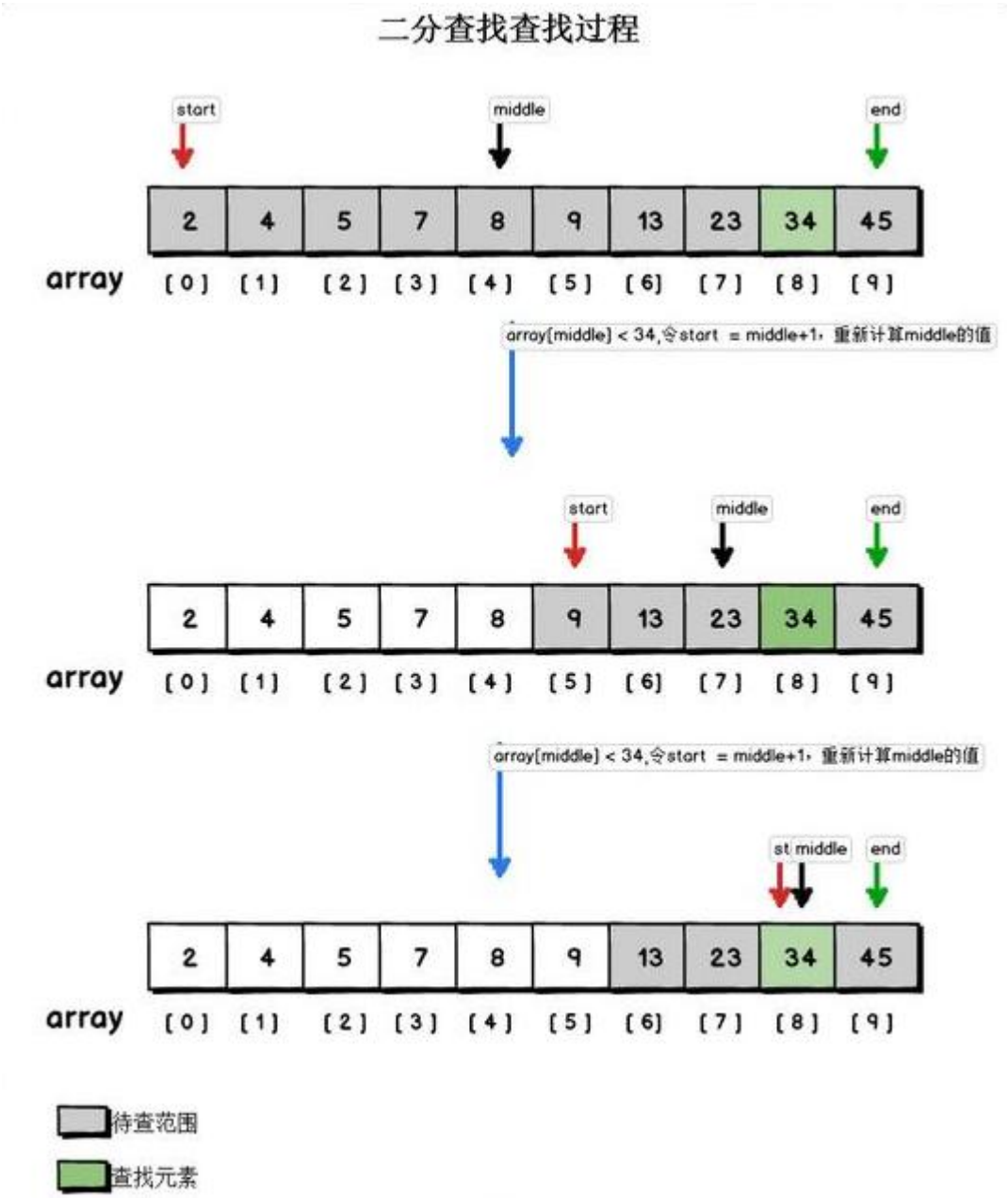
线性查找又称顺序查找，是一种最简单的查找方法，它的基本思想是从第一个记录开始，逐个比较记录的关键字，直到和给定的 K 值相等，则查找成功；

```
public static void main(String[] args) {
    int[] arrays = {1,4,6,8,9,10,20};
    search(arrays, value: 10);

}


public static int  search(int[] array,int value){
    for (int i=0;i<array.length;i++){
        if (array[i]==value){
            return i;
        }
    }
    return -1;
}
```

7.2 二分查找算法

二分查找也称折半查找（Binary Search），它是一种效率较高的查找方法。但是，折半查找要求线性表必须采用顺序存储结构，而且表中元素按关键字有序排列

7.2.1 二分查找算法原理



7.2.2 二分查找代码

```
/**
```

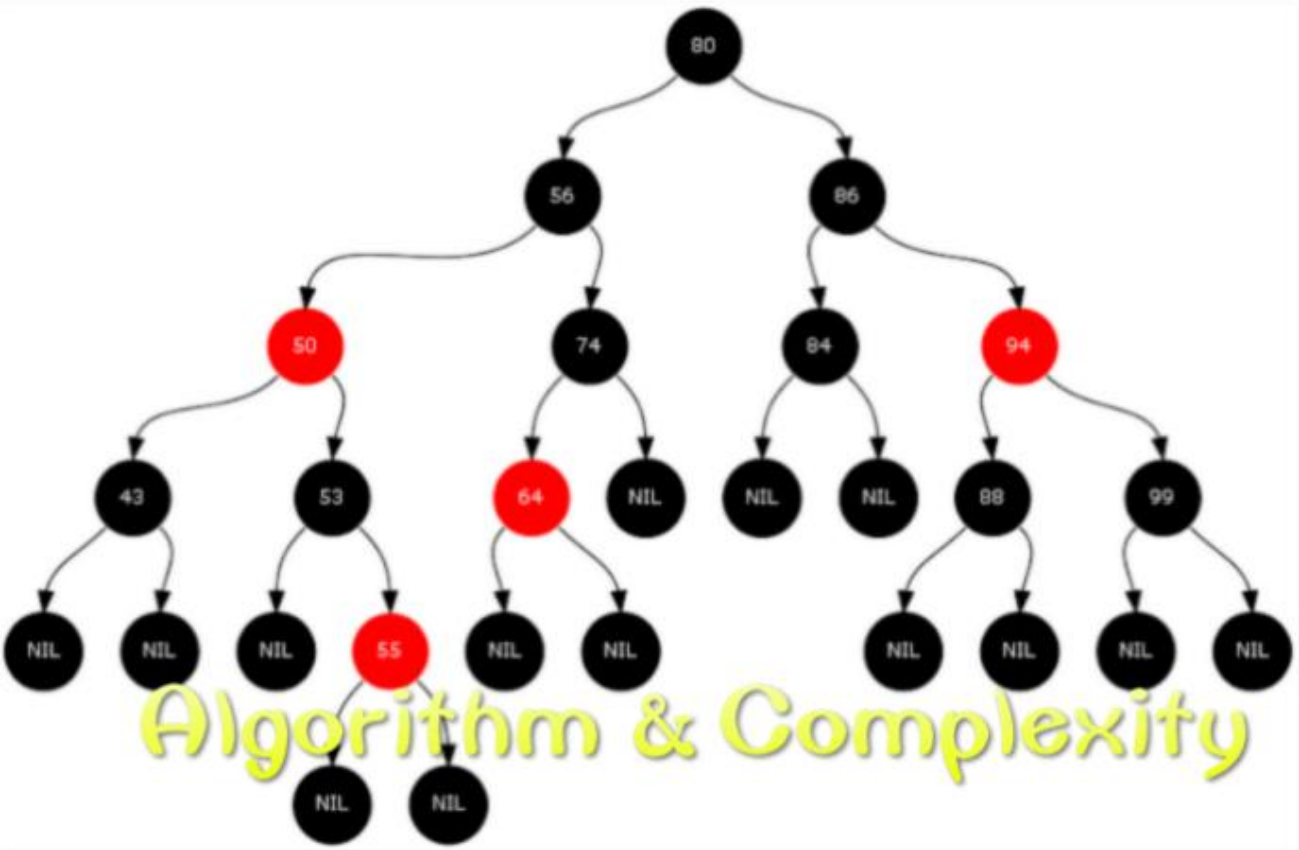


```
 * author:韩国庆
 * date:2021/3/15 15:33
 * version:1.0
 */
public class Sort {

    public static void main(String[] args) {
        int[] arrays = {1,4,6,8,9,10,20};
        int startIndex = 0;
        int endIndex = arrays.length-1;
        System.out.println(search(arrays,startIndex,endIndex,8));;

    }

    public static int search(int[] arrays,int start,int end,int val){
        if (start>end){
            return -1;
        }
        int mid = (start+end)/2;
        int midVal = arrays[mid];
        if (val>midVal){
            return search(arrays,mid+1,end,val);
        }else if (val<midVal){
            return search(arrays,start,mid-1,val);
        }else {
            return mid;
        }

    }
}
```

7.3 插值查找算法

插值查找，有序表的一种查找方式。插值查找是根据查找关键字与查找表中最大最小记录关键字比较后的查找方法。插值查找基于二分查找，将查找点的选择改进为自适应选择，提高查找效率。

7.3.1 插值查找算法原理



当我们从字典中查找 “algorithm” 这个单词的时候，我们肯定不会傻傻地像二分查找一样首先从中间开始。相反，我们会从首字母为 a 的地方开始查找，然后根据第二个字母在字母表中的位置，找到相应的位置再继续查找，这样重复这个过程，直到我们查找到这个单词。

7.3.2 插值查找代码

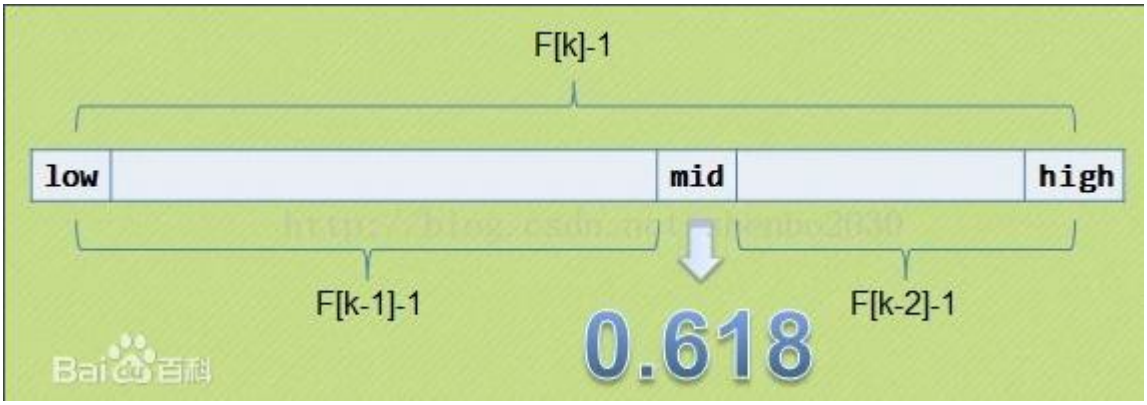
```
/**
 * author:韩国庆
 * date:2021/3/15 15:33
 * version:1.0
 */
public class Sort {

    public static void main(String[] args) {
        int[] arrays = {1,4,6,8,9,10,20};
        int startIndex = 0;
        int endIndex = arrays.length-1;
        System.out.println(insertSearch(arrays,startIndex,endIndex,8));;

    }

    /**
     * 插值查找
     */
    public static int insertSearch(int[] arr,int left,int right,int searchVal){

        /**
         * 判断，防止越界，searchVal<array[0] ,searchVal>array[arr.length-1]
         */
        if (left>right || searchVal<arr[0] || searchVal>arr[arr.length-1]){
            return -1;
        }
```

```
        int mid = left+(right-left)*(searchVal-arr[left])/(arr[right]-arr[left]);
        int midVal = arr[mid];
        if (searchVal>midVal){
            return insertSearch(arr,mid+1,right,searchVal);
        }else if (searchVal<midVal){
            return insertSearch(arr,left,mid-1,searchVal);
        }else {
            return mid;
        }


    }
    }
```

7.4 黄金分割法算法（斐波那契算法）

7.4.1 介绍

在介绍斐波那契查找算法之前，先介绍一下很它紧密相连的一个概念——黄金分割。黄金比例又称黄金分割，是指事物各部分间一定的数学比例关系，即将整体一分为二，较大部分与较小部分之比等于整体与较大部分之比，其比值约为 1:0.618 或 1.618:1。因此被称为黄金分割。斐波那契数列：1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89…….（从第三个数开始，后边每一个数都是前两个数的和）。然后我们会发现，随着斐波那契数列的递增，前后两个数的比值会越来越接近0.618，利用这个特性，我们就可以将黄金比例运用到查找技术中。



7.4.2 黄金分割法代码

```
package com.bjpowernode.sort;

import java.time.Year;
import java.util.Arrays;

/**
 * author:韩国庆
 * date:2021/3/19 14:09
 * version:1.0
 */
public class FibonacciSearch {

    public static void main(String[] args) {
        int[] arrays = {1, 4, 6, 8, 9, 10, 20};
        System.out.println(sort(arrays, 20));

    }
```

```
    /**
     * 黄金分割算法
     */
    public static int sort(int[] arrays, int key) {

        int low = 0;
        int hight = arrays.length - 1;

        /**
         * 表示斐波那契分割数值下标
         */
        int k = 0;

        /**
         * 存放 mid 值的
         */
        int mid = 0;

        int[] fib = fbnq();

        while (hight > fib[k] - 1) {
            k++;
        }

        /**
         * 因为 fib[k]值可能大于 arrays 的长度，因此我们需要使用 Arrays 类构造一个新的数组 temp，并且指向它。没有值的地方
         * 使用 0 填充
         */
        int[] temp = Arrays.copyOf(arrays, fib[k]);
```

```
for (int i = hight + 1; i < temp.length; i++) {
        temp[i] = arrays[hight];
}

/**
 * 使用 while 循环处理，找到我们的 key
 */
while (low <= hight) {
        mid = low + fib[k - 1] - 1;
        if (key < temp[mid]) {
                hight = mid - 1;

                /**
                 * 前面元素+后面的元素 = 全部元素
                 * f[k-1] + f[k-2] = f[k]
                 */
                k--;
        } else if (key > temp[mid]) {
                low = mid + 1;

                k -= 2;
        } else {//找到的
                if (mid <= hight) {
                        return mid;
                } else {
                        return hight;
                }
        }
```

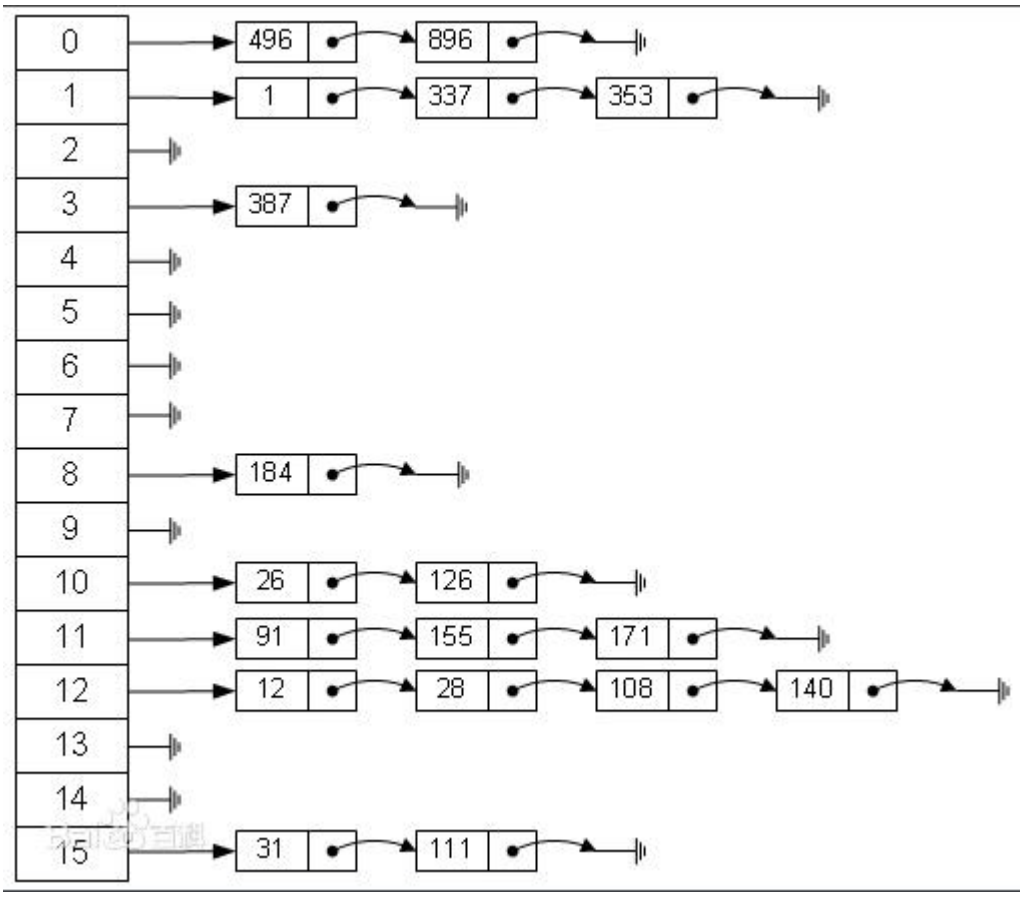
```
            }

            return -1;
        }

        public static int[] fbnq() {
            int[] fib = new int[20];
            fib[0] = 1;
            fib[1] = 1;
            for (int i = 2; i < fib.length; i++) {
                fib[i] = fib[i - 1] + fib[i - 2];
            }
            return fib;
        }

}
```

8. 哈希表

8.1 哈希表基本介绍

散列表（Hash table，也叫**哈希表**），是根据关键码值(Key value)而直接进行访问的数据结构。也就是说，它通过把关键码值映射到表中一个位置来访问记录，以加快查找的速度。

8.2 哈希表原理



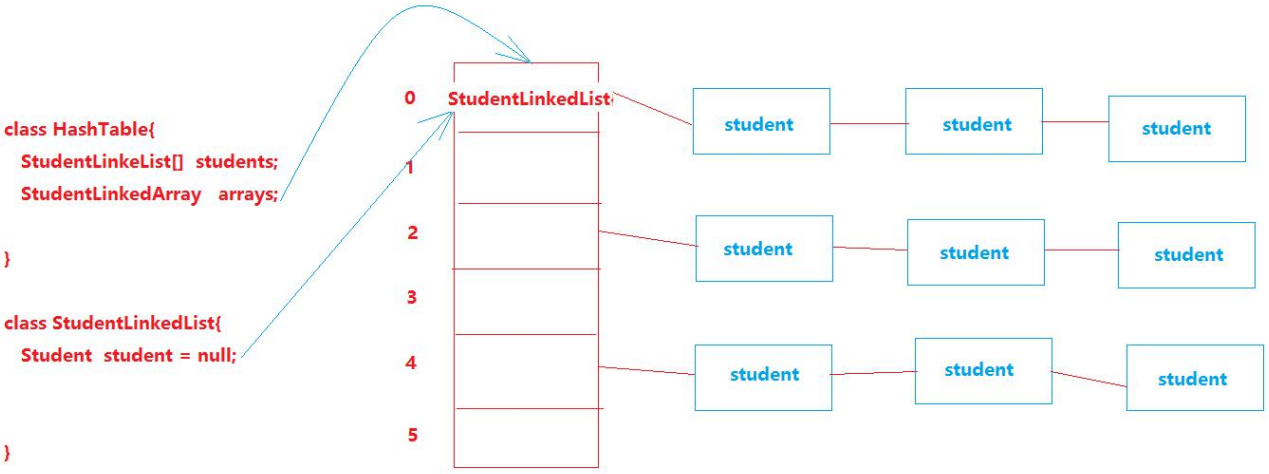
思路分析：

比如使用添加一个学生到校园系统中去，输入学生学号可以查找出该学生所有信息。

要求：

1、 添加学生编号时按照从低到高的顺序

2、 使用链表来实现哈希表，该链表不带表头



8.3 哈希表应用案例

8.3.1.1 student

```
/**
 * author:韩国庆
 * date:2021/3/20 15:55
 * version:1.0
 */
public class Student {

    public int id;
    public String name;
    public Student next;

    public Student(int id, String name) {
        this.id = id;
        this.name = name;
```

```
    }
}
```

8.3.1.2 StudentLinkedList

```
/**
 * author:韩国庆
 * date:2021/3/20 15:58
 * version:1.0
 */
public class StudentLinkedList {

    private Student head;



    public void add(Student newStudent){
        /**
         * 如果添加时是第一个学生对象。则使用一个指针，帮助找到最后一个
         */
        if (head ==null){
            head = newStudent;
            return;
        }

        /**
         * 如果添加时不是第一个学生对象
         */
        Student tempStudent = head;
        while (true){
```

```
            /**
             *
             */
            if (tempStudent.next==null){
                break;
            }
            /**
             * 否则后移一个
             */
            tempStudent = tempStudent.next;

        }
        /**
         * 循环结束后，将新的学生加入到链表中
         */
        tempStudent.next = newStudent;
    }

    /**
     * 遍历链表的学生信息
     */
    public void list(int no){

        /**
         * 说明为 null 时
         */
        if (head == null){
            System.out.println("第"+(no+1)+"链表为空");
            return;
```

```
        }
        System.out.println("第"+(no+1)+"链表信息为");

        Student tempStudent = head;
        while (true){
            System.out.printf("id=%d    name=%s\t",tempStudent.id,tempStudent.name);

            /**
             * 说明已经是最后一个结点了
             */
            if (tempStudent.next == null){
                break;
            }
            tempStudent = tempStudent.next;

        }

        System.out.println();
    }

    public Student findByStudentId(int id){
        if (head == null){
            System.out.println("链表为空");
            return null;
        }

        /**
         * 辅助指针
         */
        Student pointStudent = head;
```

```
            while (true){
                /**
                 * 找到 id 值
                 */
                if (pointStudent.id == id){
                    break;
                }

                /**
                 * 说明当前链表没有该学员
                 */
                if (pointStudent.next == null){
                    pointStudent = null;
                    break;
                }
                pointStudent = pointStudent.next;
            }
            return pointStudent;
        }
}
```

8.3.1.3 HashTable

```
/**
 * author:韩国庆
 * date:2021/3/20 16:58
 * version:1.0
 */
public class HashTable {
```

```
    private StudentLinkedList[] studentLinkedLists;

    private int size;

    public HashTable(int size){
        this.size = size;
        studentLinkedLists = new StudentLinkedList[size];

        for (int i=0;i<size;i++){
            studentLinkedLists[i] = new StudentLinkedList();
        }
    }

    /**
     * 添加学员
     */
    public void add(Student student){
        /**
         * 根据学员的 id，得到该学员应当添加到哪一条链表上
         */
        int linkedListNode = hashValue(student.id);

        /**
         * 讲学生对象添加到链表中去
         */
        studentLinkedLists[linkedListNode].add(student);
    }

    /**
```

```
     * 遍历所有的链表
     */
    public void list(){
        for (int i=0;i<size;i++){
            studentLinkedLists[i].list(i);
        }
    }

    /**
     * 根据输入的编号查询学员
     */
    public void findByStudentId(int id){
        int linkedListNode = hashValue(id);
        Student students = studentLinkedLists[linkedListNode].findByStudentId(id);
        /**
         * 找到指定的对象
         */
        if (students !=null){
            System.out.printf(" 在 第 %d 条 链 表 中 找 到 学 员 ， 编 号 是：%d\n",(linkedListNode+1),id);
        }else {
            System.out.println("整个哈希表中没有找到该学员");
        }

    }


    /**
     *编写一个散列函数，使用一个简单取模办法
     */
```

```
    public int hashValue(int id){
        return id%size;
    }


}
```

8.3.1.4 Test

```
/**
 * author:韩国庆
 * date:2021/3/20 15:55
 * version:1.0
 */
public class Test {

    public static void main(String[] args) {

        HashTable hashTable = new HashTable(10);

        Scanner scanner = new Scanner(System.in);
        while (true){
            System.out.println("add:添加学员");
            System.out.println("list:显示学员");
            System.out.println("find:查找学员");
            System.out.println("exit:退出系统");
            String next = scanner.next();
            switch (next){
                case "add":
```

```
                    System.out.println("请输入 id");
                    int id = scanner.nextInt();
                    System.out.println("请输入名字");
                    String name = scanner.next();
                    Student student = new Student(id,name);
                    hashTable.add(student);
                    break;
                case "list":
                    hashTable.list();
                    break;
                case "find":
                    System.out.println("请输入要查找的 id");
                    id = scanner.nextInt();
                    hashTable.findByStudentId(id);
                    break;
                    case "exit":
                        scanner.close();
                        System.exit(0);
                        break;
                        default:
                            break;

            }

        }
    }
}
```

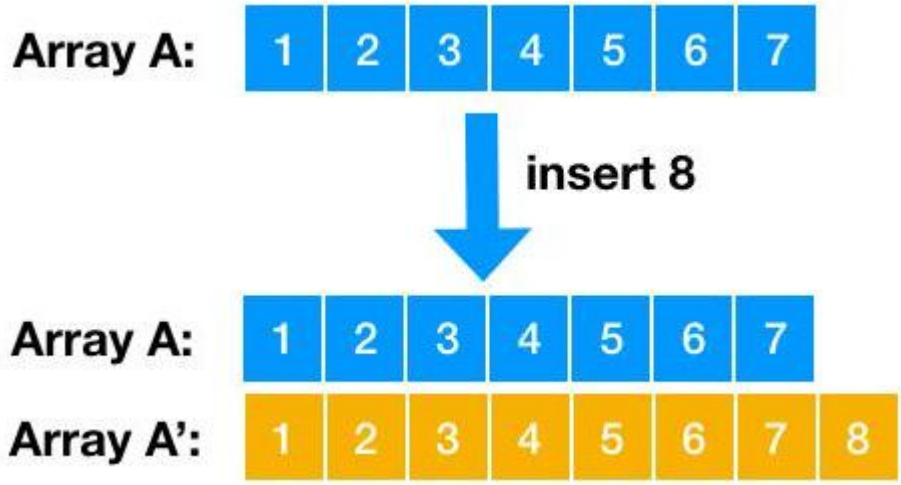
9. 树

为什么需要树中结构呢？

数组存储方式的解析：

1、 优点：通过下标方式访问元素，速度快，对于有序数组还可以使用二分查找提高检索速度。

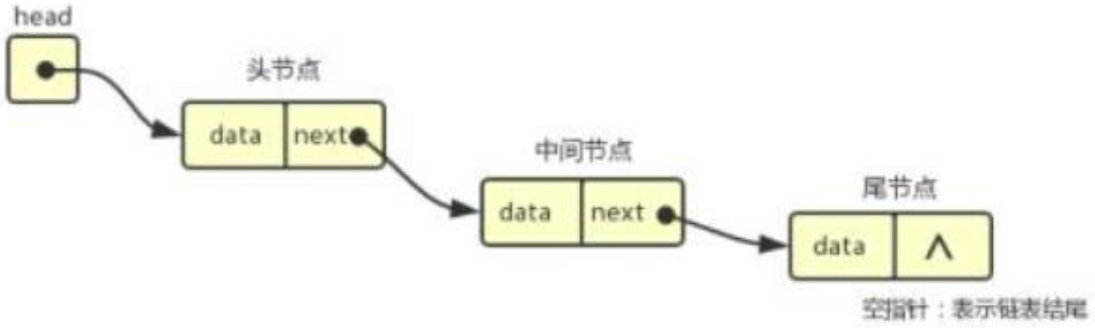
2、 缺点：如果要检索具体某个值，或者插入值会整理移动，效率较低



链式存储方式分析：

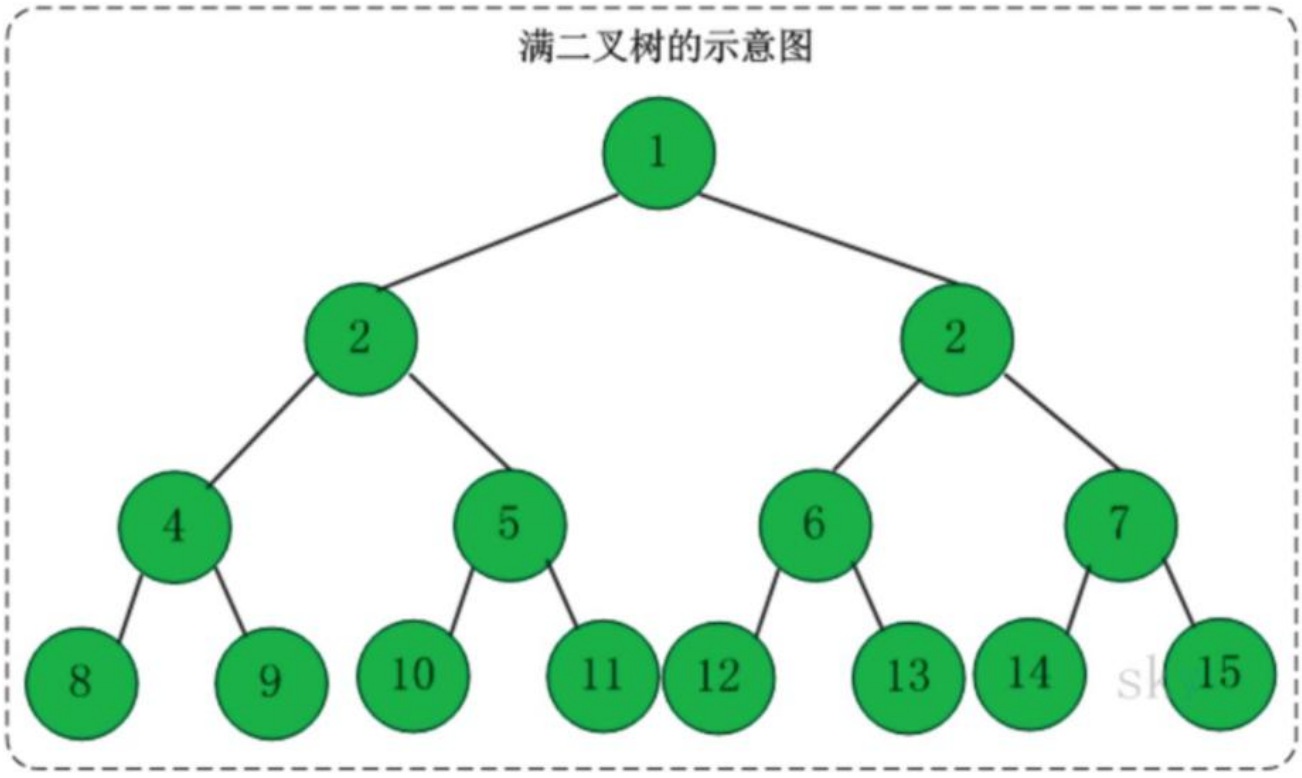
优点：在一定程序上对数组存储方式有优化，比如插入一个数值时，只需要讲插入点接到链表中即可，删除效率也是同理效果好。

缺点：在进行检索时，效率仍然很低，检索某一个值时，需要从链表头一直做检索。



树存储方式分析：

能提高数据存储，读取的效率，比如可以使用二叉树既可以保证数据检索速度，同时也可以保证数据的插入，删除，修改的速度。

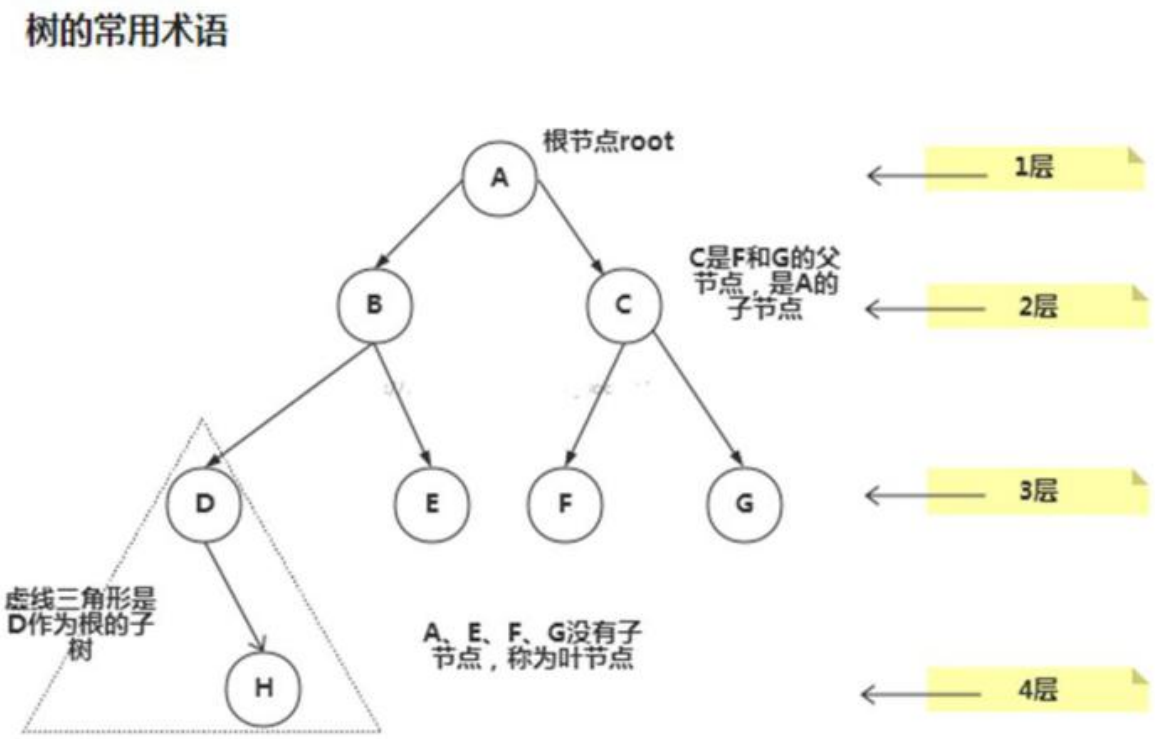


9.1 二叉树

9.1.1 二叉树概念

二叉树（Binary tree）是树形结构的一个重要类型。许多实际问题抽象出来的数据结构往往是二叉树形式，即使是一般的树也能简单地转换为二叉树，而且二叉树的存储结构及其算法都较为简单，因此二叉树显得特别重要。二叉树特点是每个结点最多只能有两棵子树，且有左右之分

9.1.2 树示意图



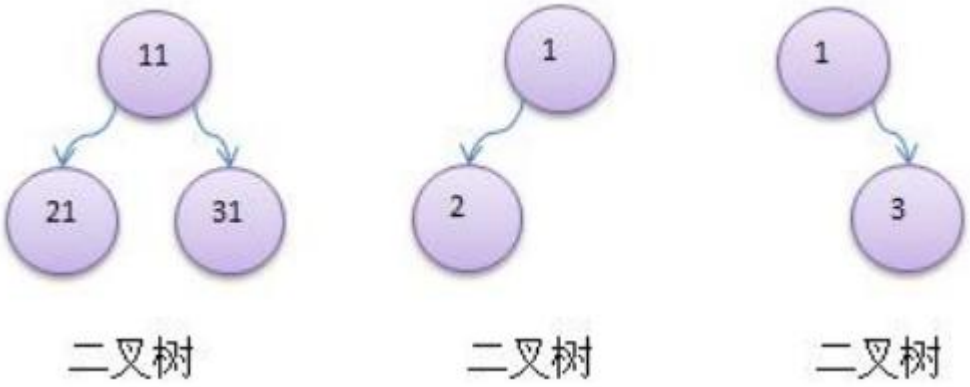
常用术语：

1、 结点

2、 根结点
3、 父结点
4、 子结点
5、 叶子结点（没有子结点的结点）
6、 结点的权（结点值）
7、 路径（从 root 结点找到目标结点的线路）
8、 层
9、 子树
10、 树的高度（最大层数）
11、 森林（多颗子树构成森林）

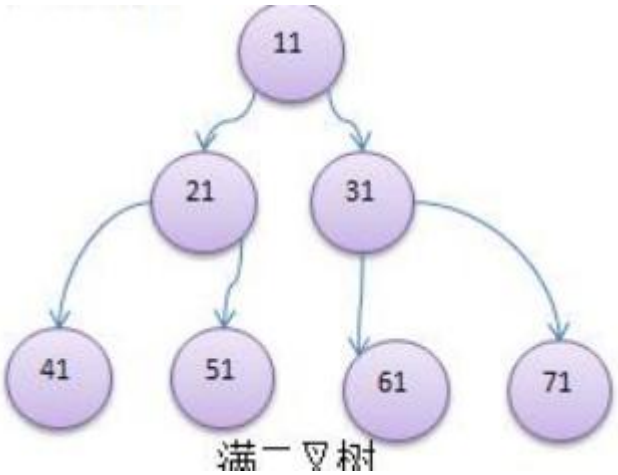
9.1.3 二叉树介绍

1、树有很多种，每个结点最多只能有两个子结点的一种形式称之为二叉树
二叉树分为左结点和右结点



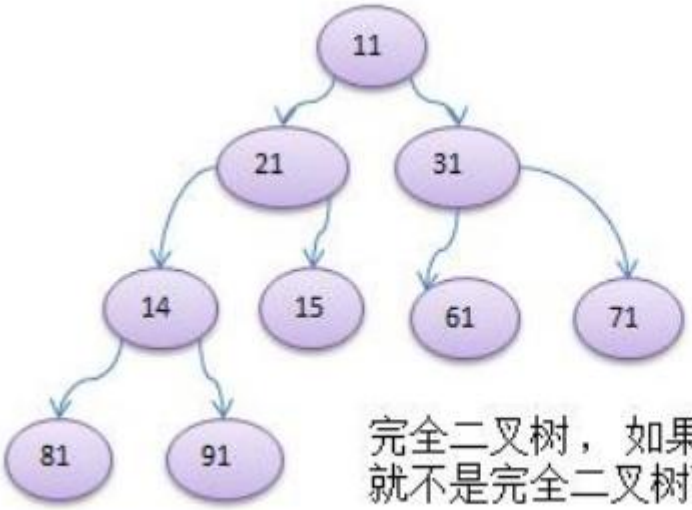
二叉树　　二叉树　　二叉树

3、 如果该二叉树的所有叶子结点都在最后一层，并且结点总数是 2^n-1，n 是层数，则我们称之为满二叉树



满二叉树

4、 如果该二叉树的所有叶子结点都在最后一层或者倒数第二层，而且最后一层的叶子结点在左边连续，倒数第二层的叶子结点在右边连续，我们称之为全完二叉树。



完全二叉树， 如果把(61)节点删除，
就不是完全二叉树了,因为叶子节点不连续了

9.1.4 二叉树应用案例

可以使用前序，中序，后序对下面的二叉树进行遍历

1、 前序遍历：先输出父结点，在遍历左子树和右子树

2、 中序遍历：先遍历左子树，在遍历父结点，再遍历右子树

3、 后序遍历：先遍历左子树，再遍历右子树，最后遍历父结点

结论：看父结点输出顺序即是某序遍历

9.1.4.1 Node

```
/**
 * author:韩国庆
 * date:2021/3/22 16:55
 * version:1.0
 */
public class Node {

    private int no;
    private String name;
    private Node left;
    private Node right;

    public Node(int no,String name){
        this.no = no;
        this.name = name;
    }

    public int getNo(){
```

```
        return no;
    }

    public void setNo(int no){
        this.no = no;
    }

    public String getName(){
        return name;
    }

    public void setName(String name){
        this.name = name;
    }

    public Node getLeft() {
        return left;
    }

    public void setLeft(Node left) {
        this.left = left;
    }

    public Node getRight() {
        return right;
    }

    public void setRight(Node right) {
        this.right = right;
    }
```



```
@Override
public String toString() {
        return "Node{" +
                "no=" + no +
                ", name='" + name + '\'' +
                ", left=" + left +
                ", right=" + right +
                '}';
}

/**
 * 前序遍历
 */
public void preOrder(){
        //先输出父结点
        System.out.println(this);
        if (this.left !=null){
                this.left.preOrder();
        }
        if (this.right !=null){
                this.right.preOrder();
        }
}

/**
 * 中序遍历
 */
public void infixOrder(){
        //递归向左子树遍历
```

```
            if (this.left !=null){
                this.left.infixOrder();
            }
            //输出父结点
            System.out.println(this);

            if (this.right !=null){
                this.right.infixOrder();
            }
        }

        /**
         * 后序遍历
         */
        public void postOrder(){
            if (this.left !=null){
                this.left.postOrder();
            }
            if (this.right !=null){
                this.right.postOrder();
            }
            //输出父结点
            System.out.println(this);
        }
}
```

9.1.4.2 BinaryTree

```
/**
 * author:韩国庆
 * date:2021/3/22 16:47
 * version:1.0
 */

public class BinaryTree {

    private Node root;

    public void setRoot(Node root) {
        this.root = root;
    }

    /**
     * 前序遍历
     */
    public void preOrder(){
        if (this.root !=null){
            this.root.preOrder();
        }else {
            System.out.println("二叉树为空，无法遍历");
        }
    }

    public void   infixOrder(){
        if (this.root !=null){
            this.root.infixOrder();
        }else {
            System.out.println("二叉树为空，无法遍历");
```

```
        }
    }

    public void postOrder(){
        if (this.root !=null){
            this.root.postOrder();
        }else {
            System.out.println("二叉树为空，无法遍历");
        }
    }
}
```

9.1.4.3 Test

```
/**
 * author:韩国庆
 * date:2021/3/23 14:23
 * version:1.0
 */
public class Test {

    public static void main(String[] args) {

        BinaryTree binaryTree = new BinaryTree();

        Node root = new Node(1,"孙尚香");
        Node node2 = new Node(2,"夏侯惇");
        Node node3 = new Node(3,"貂蝉");
        Node node4 = new Node(4,"吕布");
```

```
        Node node5 = new Node(5,"虞姬");
        Node node6 = new Node(6,"王昭君");

        root.setLeft(node2);
        node2.setLeft(node3);
        root.setRight(node4);
        node4.setRight(node5);
        node2.setLeft(node6);


        binaryTree.setRoot(root);
        /**
         * 前序
         */
        binaryTree.preOrder();

        System.out.println("----------------------------------------------");

        binaryTree.infixOrder();
        System.out.println("----------------------------------------------");
        binaryTree.postOrder();


    }
}
```

9.1.5 二叉树 查询结点

根据后序，中序，前序来进行查找

Node

```
package com.bjpowernode.binary;

/**
 * author:韩国庆
 * date:2021/3/22 16:55
 * version:1.0
 */
public class Node {

    private int no;
    private String name;
    private Node left;
    private Node right;

    public Node(int no,String name){
        this.no = no;
        this.name = name;
    }

    public int getNo(){
        return no;
    }

    public void setNo(int no){
        this.no = no;
    }

    public String getName(){
        return name;
```

```
    }

    public void setName(String name){
        this.name = name;
    }

    public Node getLeft() {
        return left;
    }

    public void setLeft(Node left) {
        this.left = left;
    }

    public Node getRight() {
        return right;
    }

    public void setRight(Node right) {
        this.right = right;
    }

    @Override
    public String toString() {
        return "Node{" +
                "no=" + no +
                ", name='" + name + '\'' +
                ", left=" + left +
                ", right=" + right +
                '}';
```

```
    }

    /**
     * 前序遍历
     */
    public void preOrder(){
        //先输出父结点
        System.out.println(this);
        if (this.left !=null){
            this.left.preOrder();
        }
        if (this.right !=null){
            this.right.preOrder();
        }
    }

    /**
     * 中序遍历
     */
    public void infixOrder(){
        //递归向左子树遍历
        if (this.left !=null){
            this.left.infixOrder();
        }
        //输出父结点
        System.out.println(this);

        if (this.right !=null){
            this.right.infixOrder();
        }
```

```
}

/**
 * 后序遍历
 */
public void postOrder(){
    if (this.left !=null){
        this.left.postOrder();
    }
    if (this.right !=null){
        this.right.postOrder();
    }
    //输出父结点
    System.out.println(this);
}


/**
 * 前序遍历查找
 */
public Node preOrderSearch(int no){

    /**
     * 判断是否是当前结点
     */
    if (this.no == no){
        return this;
    }

    /**
```

```
        * 判断当前结点的左子结点是否为空，如果不是空，则递归前序查找
            如果左递归前序查找，找到结点，则返回
        */
        Node lNode = null;
        if (this.left !=null){
            lNode = this.left.preOrderSearch(no);
        }

        if (lNode !=null){
            return lNode;
        }

        /**
        * 左递归前序查找，找到结点，则返回否则继续判断
        *
        * 当前的结点的右子结点是否为空，如果不空，则继续像右递归前序查找
        */
        if (this.right !=null){
            lNode = this.right.preOrderSearch(no);
        }
        return lNode;

    }

    /**
     * 中序遍历查找
     */
    public Node infixOrderSearch(int no){
        /**
        * 判断当前结点左子结点是否为空，如果不为空，则递归中序查找
```

```
         */
        Node node = null;

        if (this.left !=null){
            node = this.left.infixOrderSearch(no);
        }
        if (node !=null){
            return node;
        }

        if (this.no == no){
            return this;
        }

        if (this.right !=null){
            node = this.right.infixOrderSearch(no);
        }
        return node;
    }

    /**
     * 后序遍历查找
     */
    public Node postOrderSearch(int no){
        Node node = null;
        if (this.left !=null){
            node = this.left.postOrderSearch(no);
        }

        if (node !=null){
```

```
            return node;
        }

        /**
         * 如果左边子树没有找到，则向右边子树递归进行后序遍历查找
         */
        if (this.right !=null){
            node = this.right.postOrderSearch(no);
        }
        if (node !=null){
            return node;
        }

        /**
         * 如果左右子树都没有找到，那么判断当前结点是否是呢
         */
        if (this.no==no){
            return this;
        }
        return node;
    }

}
```

BinaryTree

```
/**
 * author:韩国庆
 * date:2021/3/22 16:47
 * version:1.0
```

```
 */

public class BinaryTree {

    private Node root;

    public void setRoot(Node root) {
        this.root = root;
    }

    /**
     * 前序遍历
     */
    public void preOrder(){
        if (this.root !=null){
            this.root.preOrder();
        }else {
            System.out.println("二叉树为空，无法遍历");
        }
    }

    public void    infixOrder(){
        if (this.root !=null){
            this.root.infixOrder();
        }else {
            System.out.println("二叉树为空，无法遍历");
        }
    }

    public void postOrder(){
```

```
        if (this.root !=null){
            this.root.postOrder();
        }else {
            System.out.println("二叉树为空，无法遍历");
        }
    }


    /**
     * 前序
     */
    public Node preOrderNode(int no){
        if (root !=null){
            return root.preOrderSearch(no);
        }else {
            return null;
        }
    }

    /**
     * 中序
     */
    public Node infixOrderNode(int no){
        if (root !=null){
            return root.infixOrderSearch(no);
        }else {
            return null;
        }
    }
```

```
    /**
     * 后序
     */
    public Node postOrderNode(int no){
        if (root !=null){
            return root.postOrderSearch(no);
        }else {
            return null;
        }
    }



}
```

test

```
/**
 * author:韩国庆
 * date:2021/3/23 14:23
 * version:1.0
 */
public class Test {

    public static void main(String[] args) {
```

```
BinaryTree binaryTree = new BinaryTree();

Node root = new Node(1,"孙尚香");
Node node2 = new Node(2,"夏侯惇");
Node node3 = new Node(3,"貂蝉");
Node node4 = new Node(4,"吕布");
Node node5 = new Node(5,"虞姬");
Node node6 = new Node(6,"王昭君");

root.setLeft(node2);
node2.setLeft(node3);
root.setRight(node4);
node4.setRight(node5);
node2.setLeft(node6);



binaryTree.setRoot(root);
/**
 * 前序
 */
/*binaryTree.preOrder();

System.out.println("----------------------------------------------");

binaryTree.infixOrder();
System.out.println("----------------------------------------------");
binaryTree.postOrder();*/


/**
```

```
        * 根据 id   前序查找
        */
        Node node = binaryTree.preOrderNode(5);
        if (node !=null){
            System.out.printf("信息为：id=%d name%s",node.getNo(),node.getName());
        }else {
            System.out.println("没有找到");
        }

        /**
        * 后序查找
        */
        Node infix = binaryTree.infixOrderNode(5);
        if (infix !=null){
            System.out.printf("信息为：id=%d name%s",infix.getNo(),infix.getName());
        }else {
            System.out.println("没有找到");
        }


        Node post = binaryTree.postOrderNode(5);
        if (post !=null){
            System.out.printf("信息为：id=%d name%s",post.getNo(),post.getName());
        }else {
            System.out.println("没有找到");
        }


    }
```

```
}
```

9.1.6 二叉树 删除结点

删除结点可划分为两种情况：

1、 删除的结点是叶子结点，那么删除当前几点即可。

2、 删除的结点是非叶子结点，那么需要删除该子树。

删除需求：

1、 删除 5 号结点

Node 类增加删除方法

```
public void delNode(int no){

        /**
         * 注意：
         * 1、二叉树是单向的，所以我们是判断当前结点的子结点是否需要删除，而不能去判断当前这个结点是不是需要删除结点
         * 2、如果当前结点左子结点不为空，并且左子结点就是需要删除的结点，则将 this.left = null,返回即可
         * 3、如果当前结点右子结点不为空，并且右子结点就是需要删除的结点，就将 this.right = null,返回即可
         * 4、如果 2,3 步没有执行，那么需要向左子树进行递归删除
         * 5、如果第 4 步也没有删除结点，则向右子树进行递归删除
         */

        /**
         * 第 2 步操作
         */
```

```
if (this.left !=null &&this.left.no==no){
        this.left = null;
        return;
}

/**
 * 第 3 步操作
 */
if (this.right !=null && this.right.no==no){
        this.right = null;
        return;
}

/**
 * 第 4 步：向左子树递归删除
 */
if (this.left !=null){
        this.left.delNode(no);
}

/**
 * 第 5 步：向右子树递归删除
 */
if (this.right !=null){
        this.right.delNode(no);
}
```

```
    }
```

BinaryTree 类增加删除方法

```
public void delNode(int no){

        if (root !=null){
            if (root.getNo()==no){
                root = null;
            }else {
                root.delNode(no);
            }
        }else {
            System.out.println("树为空，不能操作删除");
        }
    }
```

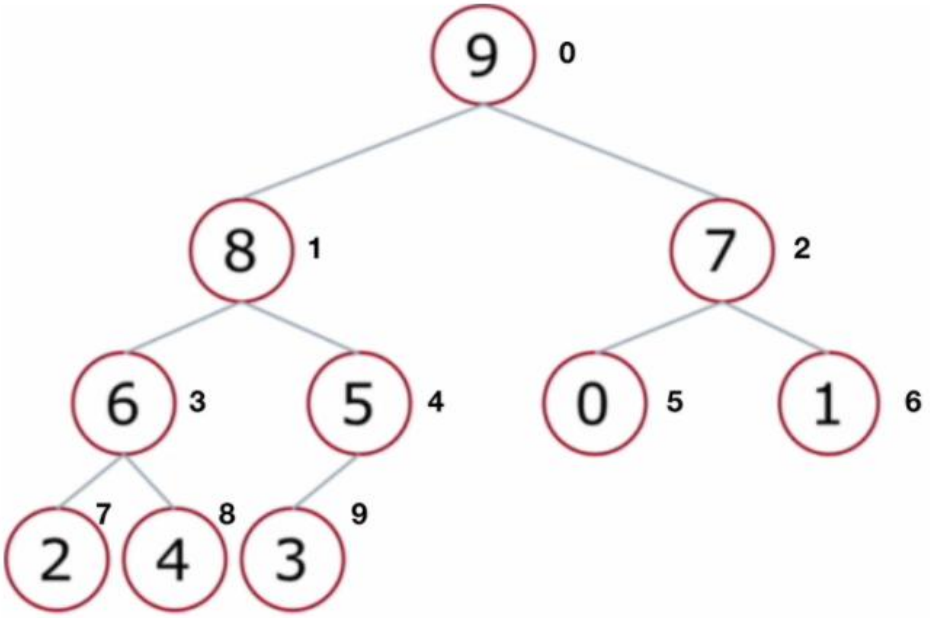
测试

```
Node node2 = new Node( no: 2, name: "夏侯惇");
Node node3 = new Node( no: 3, name: "貂蝉");
Node node4 = new Node( no: 4, name: "吕布");
Node node5 = new Node( no: 5, name: "虞姬");
Node node6 = new Node( no: 6, name: "王昭君");

root.setLeft(node2);
node2.setLeft(node3);
root.setRight(node4);
node4.setRight(node5);
node2.setLeft(node6);


binaryTree.setRoot(root);
/**
 * 前序
 */
binaryTree.preOrder();
binaryTree.delNode(5);
binaryTree.preOrder();
```


9.2 顺序存储二叉树

9.2.1 顺序存储二叉树概念

二叉树的顺序存储就是用一组连续的存储单元（数组）存放二叉树中的结点元素，一般按照二叉树结点自上向下、自左向右的顺序存储。



顺序存储二叉树特点：

1、 顺序二叉树通常只考虑完全二叉树

2、 第 n 个元素的左子结点为 2*n+1

3、 第 n 个元素的右子结点为 2*n+2

4、 第 n 个元素的父结点为（n-1）/2

5、 n 表示二叉树中第几个元素，按编号从 0 开始

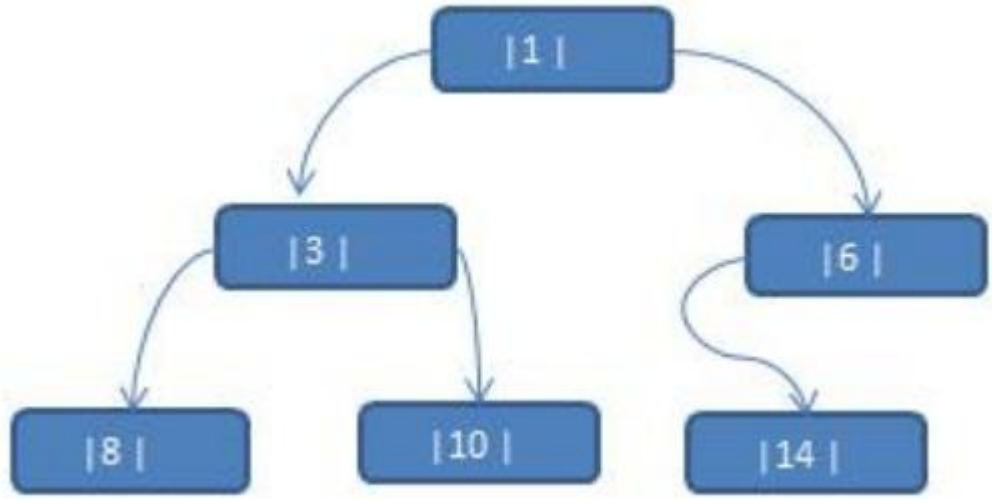
9.2.2 顺序存储二叉树遍历

```
/**
 * author:韩国庆
 * date:2021/3/26 9:21
 * version:1.0
 */
public class ArraysBinaryTree {

    private int[] arrays;

    public ArraysBinaryTree(int[] arrays){
        this.arrays = arrays;
    }

    /**
     * 顺序存储二叉树前序遍历
     * @param index
     */
    public void preOrder(int index){
        if (this.arrays == null || arrays.length == 0){
            System.out.println("数组为空，不能遍历");
        }

        /**
         * 向左递归
         */
```

```
            if ((index *2+1)<arrays.length){
                preOrder(index*2+1);
            }

            /**
             * 向右递归
             */
            if ((index*2+2)<arrays.length){
                preOrder(index*2+2);
            }



    }
}
```

9.3 线索化二叉树

将数列[1,3,6,8,10,14] 构建成一颗二叉树



问题分析：

1、 当我们对上述二叉树进行中序遍历时，数列为[8,3,10,1,6,14]

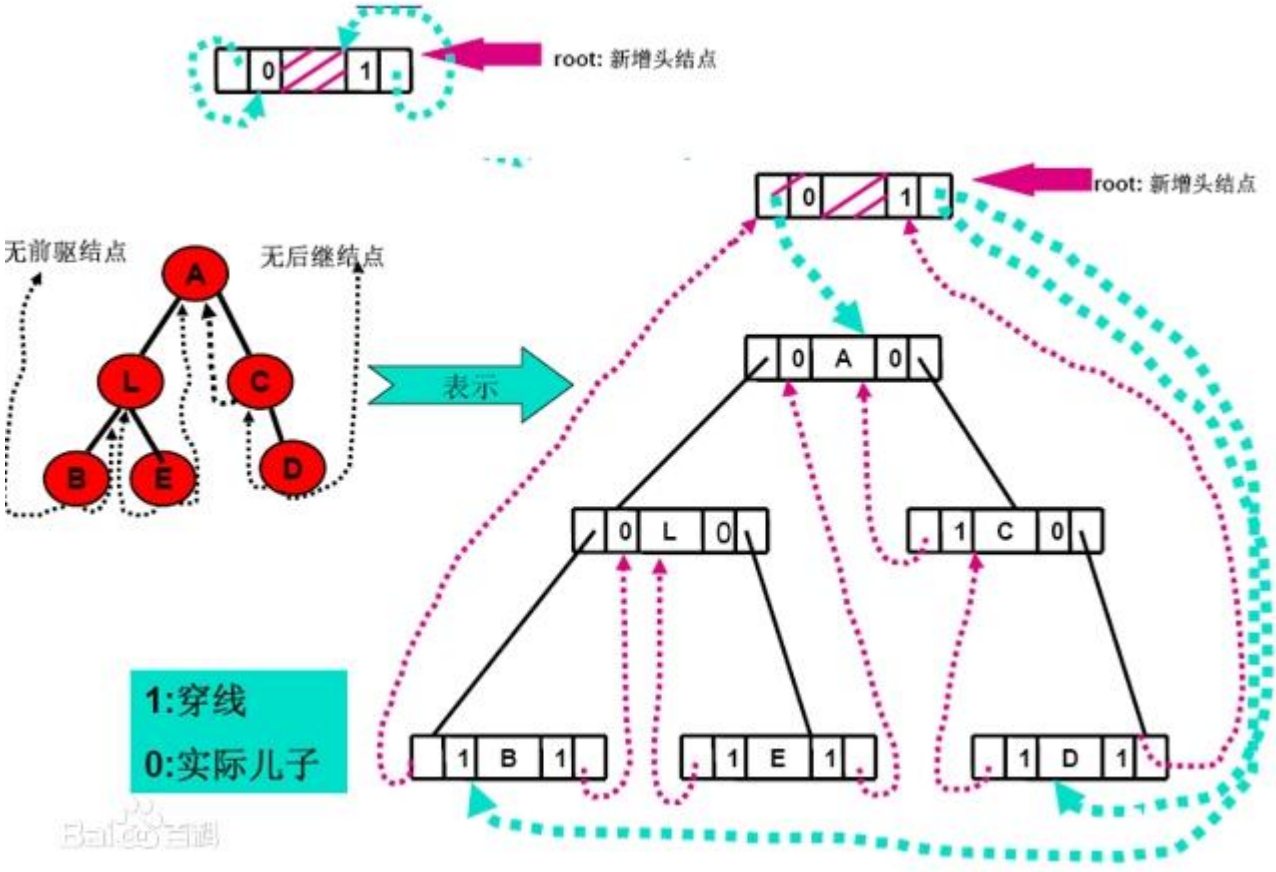
2、 6,8,10,14 这几个结点左右指针并没有很好的利用上。

3、 如果考虑能够让每个结点充分利用左右指针指向自己的前后结点怎么去解决呢

这时只能选择使用线索二叉树

9.3.1 线索二叉树介绍

在二叉树的结点上加上线索的二叉树称为线索二叉树，对二叉树以某种遍历方式（如先序、中序、后序等）进行遍历，使其变为线索二叉树的过程称为对二叉树进行线索化



对于 n 个结点的二叉树，在二叉链存储结构中有 n+1 个空链域，利用这些空链域存放在某种遍历次序下该结点的前驱结点和后继结点的指针，这些指针称为线索，加上线索的二叉树称为线索二叉树。

这种加上了线索的二叉链表称为线索链表，相应的二叉树称为线索二叉树(Threaded BinaryTree)。根据线索性质的不同，线索二叉树可分为前序线索二叉树、中序线索二叉树和后序线索二叉树三种。

一个结点的前一个结点，称之为前驱结点，一个结点后一个结点称之为后继结点。

11.3 B 树、B+树、B*树

11.3.1 B 树介绍

11.3.2 B+树介绍

11.3.3 B*树介绍

12. 图

12.1 图基本介绍

12.1.1 为什么要有图

12.1.2 图的举例说明

12.1.3 图的常用概念

12.2 图的表示方式

12.2.1 邻接矩阵

12.2.2 邻接表

13. 常用 10 中算法

13.1 二分查找算法

13.1.1 二分查找算法介绍

13.1.2 二分查找算法实现

13.2 分治算法

13.2.1 分治算法介绍

13.2.2 分治算法步骤

13.2.3 汉诺塔

13.3 动态规划算法

13.3.1 动态规划算法介绍

13.3.2 动态规划算法实现

13.4 KMP 算法

13.4.1 实践应用场景

13.4.2 暴力匹配算法

13.4.3 KMP 算法介绍

13.4.4 KMP 算法应用

13.5 贪心算法

13.6 普里姆算法

13.7 克鲁斯卡尔算法

13.7.1 应用场景

13.7.2 克鲁斯卡尔算法介绍

13.7.3 克鲁斯卡尔算法示意图

13.7.4 克鲁斯卡尔算法算法分析

13.7.5 克鲁斯卡尔算法是否构成回路

13.7.6 克鲁斯卡尔算法实现

13.8 迪杰斯特拉算法

13.8.1 应用场景

13.8.2 迪杰斯特拉算法介绍

13.8.3 迪杰斯特拉算法实现

13.9 佛洛伊德算法

13.9.1 应用场景

13.9.2 佛洛伊德算法介绍

13.9.3 佛洛伊德算法图解

13.9.4 佛洛伊德算法实现

13.10 马踏棋盘算法

13.10.1 马踏棋盘算法介绍

13.10.2 马踏棋盘算法应用实现

