数据结构与算法（四）

1：今日课程内容介绍(了解)

在今天的课程中我们主要学习三个模块的相关知识点，第一个模块是堆，其中包括堆的定义和实现以及堆的应用；其次是关于图，介绍图的定义及相关概念，图的存储，以及两种搜索算法 BFS 和 DFS；最后介绍字符串匹配算法，分别是 BF,RK

2：今日课程目标(了解)

1：理解堆的定义及存储结构

2：掌握堆的实现

3：掌握堆排序算法

4：理解图的定义及概念

5：理解图的存储方法

6：掌握并实现 BFS 和 DFS 搜索算法

7：掌握并实现单模式字符串查找算法 BF,RK 算法

3：堆

我们今天讲另外一种特殊的树：“堆”（ Heap）。堆这种数据结构的应用场景非常多，你可能听过经典的堆排序，当然了我们在本章内容中也会讲到堆排序是一种原地的、时间复杂度为$O(n*logn)$的排序算法。接下来我们先来学习堆这种树结构。

3.1.堆的定义

堆其实是一棵树，并且是一种特殊的树。我们现在就来看看，什么样的树才是堆。其实只要满足以下两个要求它就是一个堆。

1. 堆是一个完全二叉树；
2. 堆中每一个节点的值都必须大于等于（或小于等于）其子树中每个节点的值

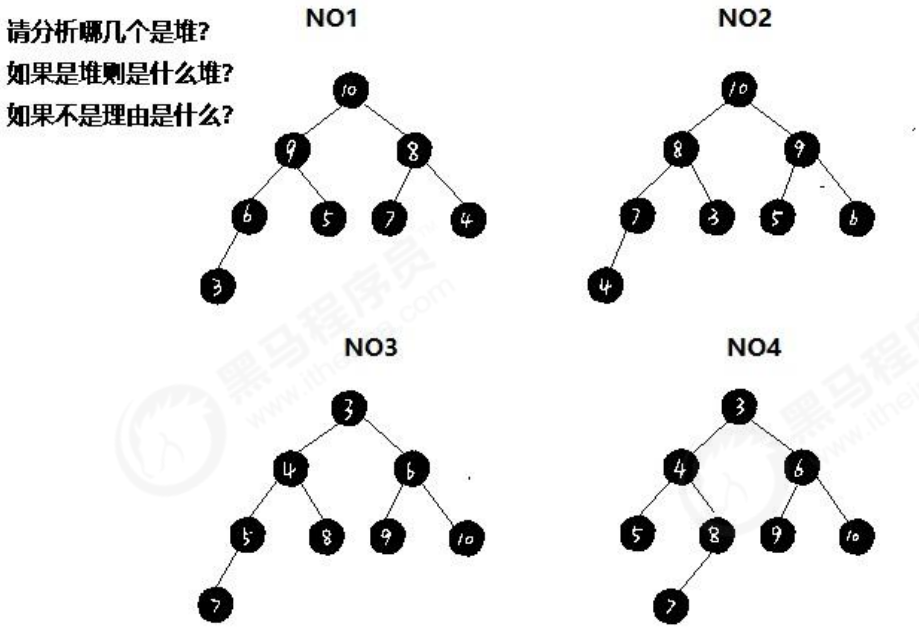
对于这两点要求，我们依次解释如下：

第一点要求堆是一个完全二叉树，对于完全二叉树除了最后一层，其他层的节点个数都是满的，最后一层的节点都靠左排列，当然想到完全二叉树我们立马能能够想到我们之前所讲到的，**完全二叉树可以用数组来进行存储**，之前的树每个节点不仅要存储数据还要存储左右节点的指针，虽然比较符合我们的认知但毕竟消耗一些

存储空间，但是完全二叉树就不一样了，比较适合用数组来存储，是非常节省存储空间的，单纯的通过数组的下标我们就可以找到一个节点的左子节点和右子节点。

第二点要求堆中每一个节点的值都必须大于等于（或小于等于）其子树中每个节点的值，或者我们可以换一种表述方式：堆中每个节点的值都大于等于（或者小于等于）其左右子节点的值。对于每个节点的值都大于等于子树中每个节点值的堆，我们叫作“**大顶堆**”。对于每个节点的值都小于等于子树中每个节点值的堆，我们叫作“**小顶堆**”。

下面我们通过一副图来描述一下堆这种数据结构，请你分析一下图中各自是否是堆？如果是都是什么堆？

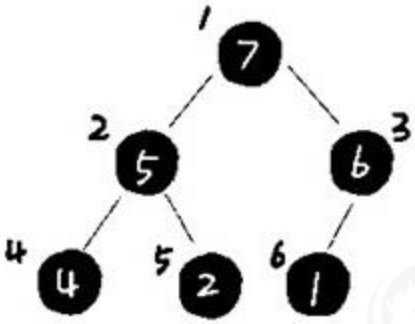


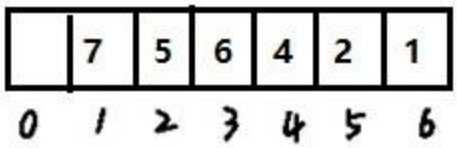
其中第 1 个和第 2 个是大顶堆，第 3 个是小顶堆，第 4 个不是堆。

3.2.堆的存储结构

我们知道堆是一个完全二叉树，我们可以用数组来存储，用数组来存储完全二叉树是非常节省存储空间的。因为我们不需要存储左右子节点的指针，单纯地通过数组的下标，就可以找到一个节点的左右子节点和父节点。

具体如何存储我们接下来用一副图的形式来解释清楚堆的存储结构





数组中下标为 k 的节点的左子节点，就是下标为 2*k 的节点，右子节点就是下标为 2 * k +1 的节点，父节点就是下标为 k/2 的节点，其中 k>=1。另外我们也发现，数组下标为 0 的位置并未存储数据，这是因为为了方便于在程序中计算，所以会浪费掉数组的一个存储空间。

3.3.堆的实现

掌握了堆的存储结构之后我们可以来手动实现一个

3.3.1 创建堆

堆的类结构定义如下：

```
/**
 * 堆:
 *   利用数组实现的大顶堆
 */
public class Heap{

    // 存储堆中元素数据的数组
    private int[]  data;
    // 堆中可存数据的最大个数
    private int size;
    // 堆中已存元素的个数
    private int count;
```

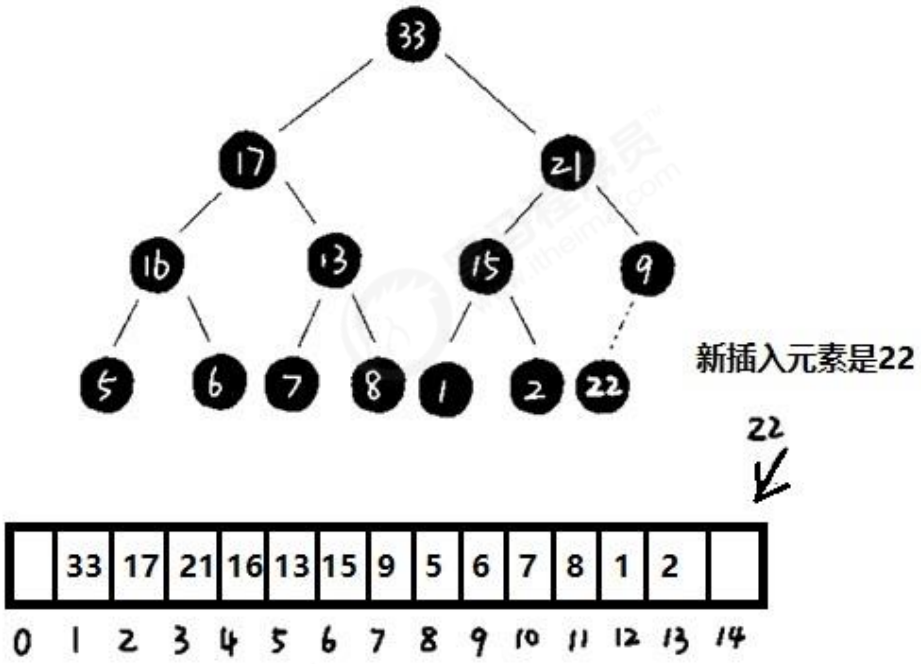
```
    /**
     * 利用初始容量构造存储堆元素的数组
     * @param capacity
     */
    public Heap(int capacity){
        // 利用数组存储堆,从数组下标为1 的地方开始存储,方便计算
        this.data = new int[capacity+1];
        this.size = capacity;
        this.count = 0;
    }
@Override
    public String toString() {
        return "Heap{" +
                "data=" + Arrays.toString(data) +
                ", size=" + size +
                ", count=" + count +
                '}';
    }
}
```

接下来我们分析如何向堆中添加一个元素

3.3.2 向堆中插入元素

我们往堆中插入元素就是向数组的数据末位添加值，也就是将新来的值放在堆的最后，前面我们已经知道了堆的特性，所以添加完成后要继续满足堆的两个特性，就需要继续进行调整，这个调整的过程我们叫做**堆化**。

堆化实际上有两种，从下往上和从上往下。这里我先讲从下往上的堆化方法。



其实自下而上的堆化特别简单，就是顺着节点所在的路径，向上对比，然后交换。又由于堆是一棵完全二叉树，数组存储的时候下标是从 1 开始存储的，所以要找某个节点的父节点非常的容易，假设节点的下标为 k，那么 k/2 就是其父节点的下标。

所以我们实现堆的插入方法如下：

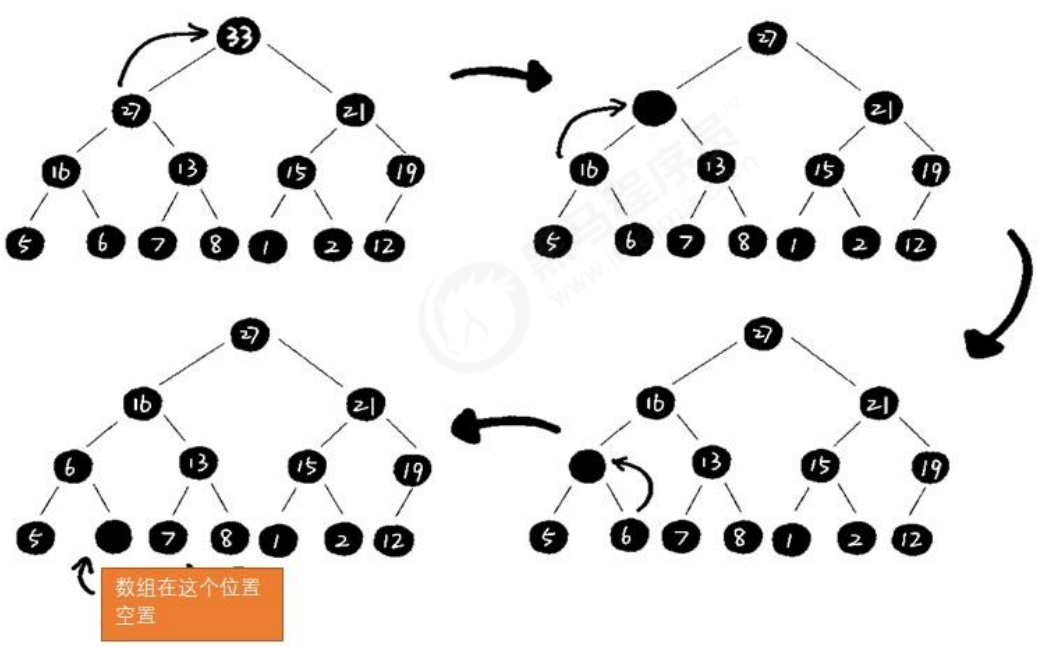
```
/**
 * 将数据存入堆中,
 * @param data
 */
public void insert(int data){
    if(count >= size ){
        return;// 堆已满
    }
    // 将数据存入
    this.data[++count] = data;
    // 堆化：自下而上堆化
    heapifyFromBottom2Top(this.data,count);
}
```

```
/**
 * 将数组 data 堆化
 * @param data
 * @param end   从 end 位置自下而上堆化
 */
private void heapifyFromBottom2Top(int[] data,int end){
    int i = end;
    while ( i/2 > 0 && this.data[i/2] < this.data[i]){
        swap(this.data,i/2,i);
        i /=2;
    }
}
/**
 * 交换数组下标 i,j 的元素
 * @param array
 * @param i
 * @param j
 */
public void swap(int[] array,int i,int j){
    int temp = array[i];
    array[i] = array[j];
    array[j] = temp;
}
```

3.3.3.删除堆顶元素

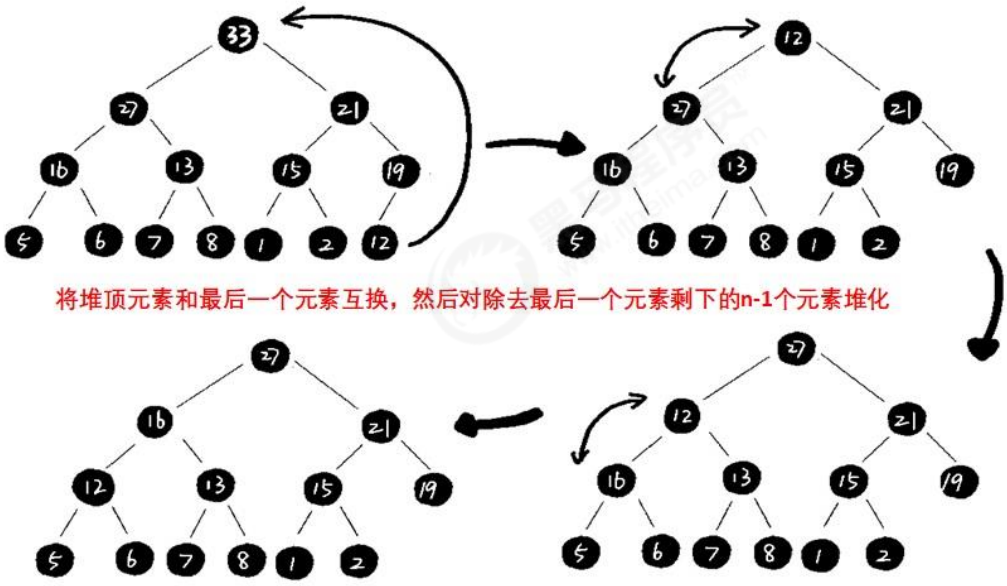
从堆的定义中我们知道：任何节点的值都大于等于（或小于等于）子树节点的值，那也就是说堆顶元素存储的就是堆中数据的最大值或者最小值。

假设我们构造的是大顶堆，堆顶元素就是最大的元素。当我们删除堆顶元素之后，就需要把第二大的元素放到堆顶，那第二大元素肯定会出现在左右子节点中。然后我们再迭代地删除第二大节点，以此类推，直到叶子节点被删除。不过这种方法有点问题，就是最后堆化出来的堆并不满足完全二叉树的特性。如下图：



我们可以换一个思路来考虑这个问题：我们把最后一个节点放到堆顶，然后利用同样的父子节点对比方法。对于不满足父子节点大小关系的，互换两个节点，并且重复进行这个过程，直到父子节点之间满足大小关系为止。这就是**从上往下的堆化**方法。

因为我们移除的是数组中的最后一个元素，而在堆化的过程中，都是交换操作，不会出现数组中的“空洞”，所以这种方法堆化之后的结果，肯定满足完全二叉树的特性



我们将刚刚所讲的代码实现如下：

```
/**
     * 移除堆顶元素
     * @return
     */
    public int removeMax(){
        // 如果是大顶堆,数组下标为1 的元素就是最大值
        int max = data[1];
        //移除完成后要保证堆的完整，需要找第二大的元素放到堆顶
        //1: 将最后一个元素直接放在堆顶,并减少数量
        data[1] = data[count--];
        //2: 堆化让其继续成为一个合格的堆---->这个时候要自上而下堆化
        headifyFromTop2Button(data,1,count);
        return max;
    }

    /**
     * 将数组 data 堆化
     * @param data
     * @param begin 从该位置自上而下堆化
     * @param end    堆化截止位置
     */
    private void headifyFromTop2Button(int[] data,int begin,int end){
        while (true){
```

```
            // 定义最大值的下标
            int maxPos = begin;
            // 比较当前节点与其左子节点,右子节点的大小关系,找出最大值
            if( 2*begin <= end && data[maxPos] < data[2*begin]){
                maxPos = 2*begin;
            }
            if(2*begin+1 <=end && data[maxPos] < data[2*begin +1]){
                maxPos = 2*begin +1;
            }
            if(begin == maxPos){
                break;
            }
            swap(data,begin,maxPos);
            begin = maxPos;
        }
    }
```

3.3.4 时间复杂度分析

通过之前的学习我们知道，一个包含 n 个节点的完全二叉树，树的高度不会超过 $log_2 n$ 。堆化的过程是顺着节点所在路径比较交换的，所以堆化的时间复杂度跟树的高度成正比，也就是 $O(logn)$。插入数据和删除堆顶元素的主要逻辑就是堆化，所以，往堆中插入一个元素和删除堆顶元素的时间复杂度都是 $O(logn)$。

3.4.堆的应用

3.4.1：堆排序

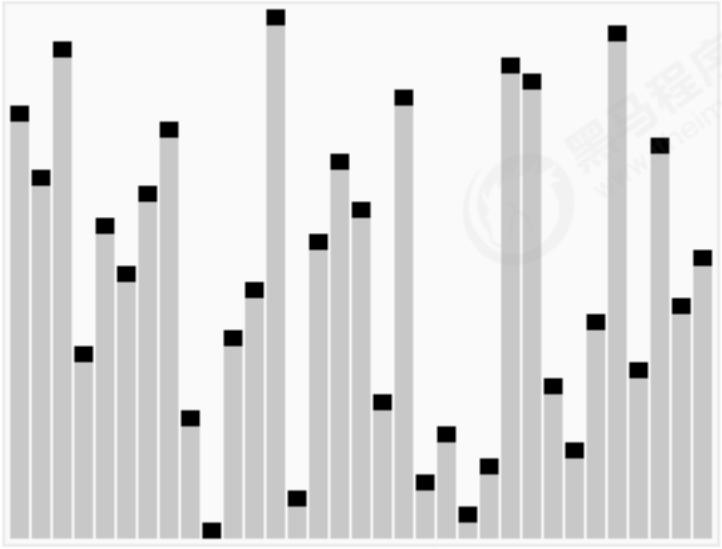
借助于堆这种数据结构实现的排序算法，就叫作堆排序。这种排序方法的时间复杂度非常稳定，是 $O(n*logn)$，并且它还是原地排序算法。我们可以把堆排序的过程大致分解成两个大的步骤，**建堆和排序**。

算法描述如下：

- 将初始待排序关键字序列(R1,R2….Rn)构建成大顶堆，此堆为初始的无序区；
- 将堆顶元素 R[1]与最后一个元素 R[n]交换，此时得到新的无序区(R1,R2,……Rn-1)和新的有序区(Rn),且满足 R[1,2…n-1]<=R[n]；
- 由于交换后新的堆顶 R[1]可能违反堆的性质，因此需要对当前无序区(R1,R2,……Rn-1)调整为新堆，然后再次将 R[1]与无序区最后一个元素交换，得到新的无序区(R1,R2….Rn-2)和新的有序区(Rn-1,Rn)。不断重复此过程直到有序区的元素个数为 n-1，则整个排序过程完成。

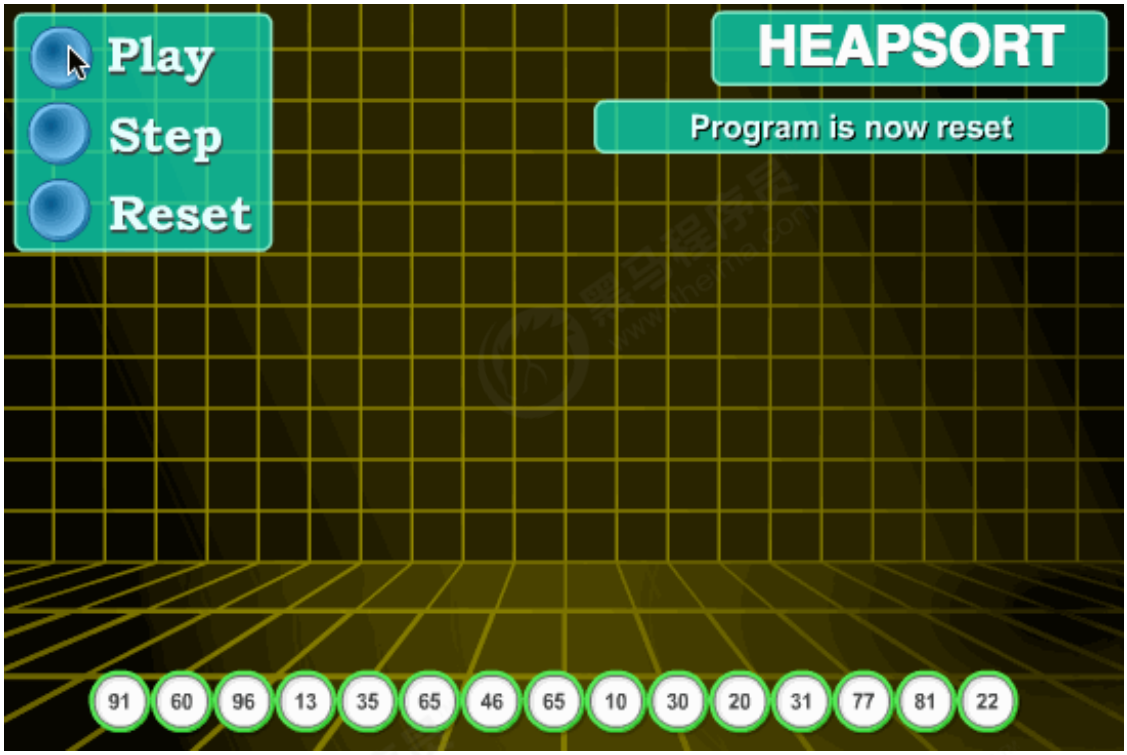
动图链接如下：

https://mmbiz.qpic.cn/mmbiz*gif/XaklVibwUKn4hqBOkhUQouzu2FDmmen4X6lF2xtnS438yPH47Cicuc8CSvAM5ORZ2iaV8REOKjavmGpL0uEPblXGg/640?tp=webp&wxfrom=5&wx*lazy=1



当然了下面这副图可能更好理解

https://mmbiz.qpic.cn/mmbiz*gif/QCu849YTaIOOdfiakqsTRHKk9icjqQZJYuZBYy12wXibrm87xCQ9hvnjtD1MXv7qmAIAQDgk1zUhkiaeoCRg6giaZ4Q/640?tp=webp&wxfrom=5&wx*lazy=1



可能看完效果之后还是不太明确，那我们就更加详细的从堆排序的两个操作上入手来进行讲解。

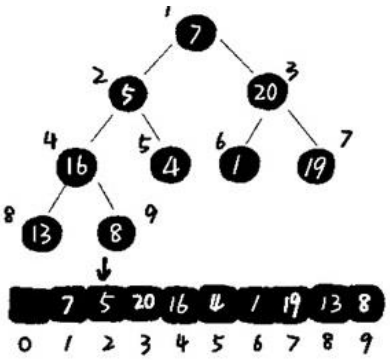
（1）、先建堆

如果给定一个数组，我们要把这个数组中的数据变成符合堆的数据存储，意味着将原始数据变成堆，并且我们是在原地将原始数据变成堆，所谓的原地操作也就是空间复杂度为 O(1)的操作，无需额外的内存空间即可完成。那如何建堆呢？

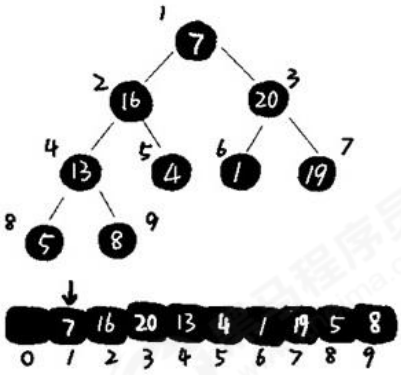
第一种是借助我们前面讲的，在堆中插入一个元素的思路。尽管数组中包含 n 个数据，但是我们可以假设，起初堆中只包含一个数据，就是下标为 1 的数据。然后，我们调用前面讲的插入操作，将下标从 2 到 n 的数据依次插入到堆中。这样我们就将包含 n 个数据的数组，组织成了堆，这里面的堆化操作是属于自下而上的堆化操作。

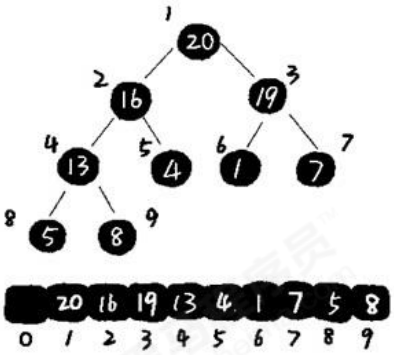
第二种实现思路，跟第一种截然相反，第一种建堆思路的处理过程是从前往后处理数组数据，并且每个数据插入堆中时，都是从下往上堆化。而第二种实现思路，是从后往前处理数组，并且每个数据都是从上往下堆化。

下面以图示的形式解释一下第二种实现思路。因为叶子节点往下堆化只能自己跟自己比较，所以我们直接从第一个非叶子节点开始，依次堆化就行了，那第一个非叶子节点如何找呢？别忘记堆是一个完全二叉树，堆中最后一个元素下标除以 2 取整就是第一个非叶子节点的下标位置。



从第一个非叶子节点开始，依次堆化，如何找第一个非叶子节点，因为堆是一个完全二叉树，堆中最后一个元素下标除以2取整就是第一个非叶子节点的下标位置





代码实现如下：

```
/**
     * 建堆：建大顶堆,堆化是自上而下堆化
     * @param array    待建堆数组
     * @param n        数组最后一个元素的下标---堆中最后一个元素下标
     */
    private void buildHeap(int[] array,int n) {
        // 从后往前处理数组数据,堆化的时候是自上而下堆化
        for(int i= n/2;i>0; i--){
            headifyFromTop2Button(array,i,n);
        }
    }

    /**
     * 将数组 data 堆化
     * @param data
     * @param begin 从该位置自上而下堆化
     * @param end    堆化截止位置
```

```
     */
    private void headifyFromTop2Button(int[] data,int begin,int end){
        while (true){
            // 定义最大值的下标
            int maxPos = begin;
            // 比较当前节点与其左子节点,右子节点的大小关系,找出最大值
            if( 2*begin <= end && data[maxPos] < data[2*begin]){
                maxPos = 2*begin;
            }
            if(2*begin+1 <=end && data[maxPos] < data[2*begin +1]){
                maxPos = 2*begin +1;
            }
            if(begin == maxPos){
                break;
            }
            swap(data,begin,maxPos);
            begin = maxPos;
        }
    }

    /**
     * 交换数组下标 i,j 的元素
     * @param array
     * @param i
     * @param j
     */
    public void swap(int[] array,int i,int j){
        int temp = array[i];
        array[i] = array[j];
        array[j] = temp;
    }
```

从代码的实现逻辑上我们发现，我们对下标从 k/2 开始到 1 的数据进行堆化，下标是 k/2+1 到 k 的节点是叶子节点，我们不需要堆化。实际上，对于完全二叉树来说，下标从 k/2+1 到 k 的节点都是叶子节点。

那我们建堆操作的时间复杂度是多少呢？我们可以大致分析一下：

首先每个节点堆化的复杂度是：$O(logn)$，跟我们堆化了多少个节点呢？大约是 n/2 +1 个节点，所以这么看时间复杂度是$O(n*logn)$，当然了这么推导出来的也没有错误，只是不够精确，实际上建堆操作的时间复杂度是 O(n)，由于推导的过程涉及很多数学知识在此我们就不做过多推导，你暂时理解就行。

（2）、后排序

建堆结束之后，数组中的数据已经是按照大顶堆的特性来组织的。数组中的第一个元素就是堆顶，也就是最大的元素。我们把它跟最后一个元素交换，那最大元素就放到了下标为 n 的位置，将剩下的 n-1 个元素重新构建成堆。堆化完成之后，我

们再取堆顶的元素，放到下标是 n-1 的位置，一直重复这个过程，直到最后堆中只剩
下标为 的一个元素，排序工作就完成了，这个过程有点类似上面讲的“删除堆顶元素”的操作

排序代码如下：

```
/**
 * 堆排序：分为两步
 *  1：建堆
 *  2：排序
 *  注意：这里数组中的数据是从下标为 1 开始
 * @param array
 */
public void heapSort(int[] array){
    //1:建堆
    buildHeap(array,array.length-1);
    // 排序
    sort(array,array.length-1);
}

/**
 * 排序：
 *  我们把堆顶元素跟最后一个元素互换，然后对 1~n-1 区间的元素再堆化，然后在将堆顶元素跟最后一个元素互换继续操作，
 *  有点类型删除堆顶元素
 * @param array
 */
private void sort(int[] array,int n){
    while (n > 1){
        swap(array,1,n);
        headifyFromTop2Button(array,1,--n);
    }
}
```

加上之前建堆的代码整个堆排序的代码就完成了，接下来编写测试代码进行测试：

```
@Test
public void testHeapSort(){
    //准备一个 int 数组
    int[] array = new int[7];
    array[1] = 2;
    array[2] = 6;
    array[3] = 9;
    array[4] = 0;
    array[5] = 3;
    array[6] = 5;
    //进行排序
```

```
        System.out.println(Arrays.toString(array));
        heapSort(array);
        //输出排序结果
        System.out.println(Arrays.toString(array));
    }
```

（3）、复杂度分析

整个堆排序的过程，都只需要极个别临时存储空间，所以堆排序是原地排序算法。堆排序包括建堆和排序两个操作，建堆过程的时间复杂度是 O(n)，排序过程的时间复杂度是$O(n*logn)$，所以，堆排序整体的时间复杂度是 $O(n*logn)$。堆排序不是稳定的排序算法，因为在排序的过程，存在将堆的最后一个节点跟堆顶节点互换的操作，所以就有可能改变值相同数据的原始相对顺序。

堆排序和快排等相同复杂度的同类排序算法的比较：

1：堆排序数据访问的方式没有快速排序友好。对于快速排序来说，数据是顺序访问的。而对于堆排序来说，数据是跳着访问的。 比如，堆排序中，最重要的一个操作就是数据的堆化。比如下面这个例子，对堆顶节点进行堆化，会依次访问数组下标是 1，2，4，81，2，4，8 的元素，而不是像快速排序那样，局部顺序访问。

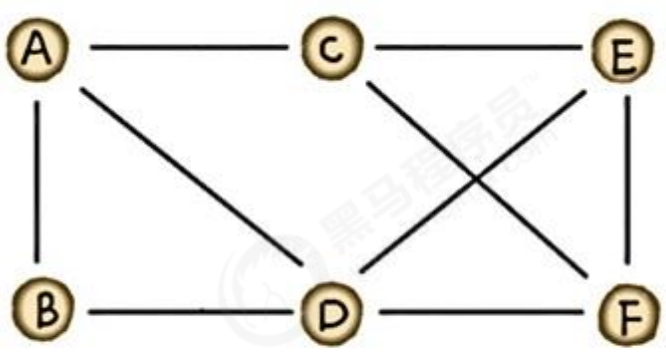
2：对于同样的数据，在排序过程中，堆排序算法的数据交换次数要多于快速排序。

3：从实际应用来说我们一般使用快排使用的多。

4：图

4.1.图的定义及相关概念

在前面的章节中我们学习了树这种非线性表数据结构，那么在本章节中我们要讲另一种非线性表数据结构，图（Graph）。图和树比起来，这是一种更加复杂的非线性表结构，那到底什么是图呢？直接说定义可能不是很好理解，那我们直接看下方图示：



以前学习树的时候，树中的元素我们称之为节点 Node，现在在**图中的每一个元素我们称之为顶点（Vertex）**，并且**图中的一个顶点可以与其他任意顶点建立连接关系，我们把这种建立的关系叫做边(Edge)**，

我们阐述了图的概念，也对图有了一个直观的认识，那我们想一想生活中有没有跟图相同或者类似的结构呢？

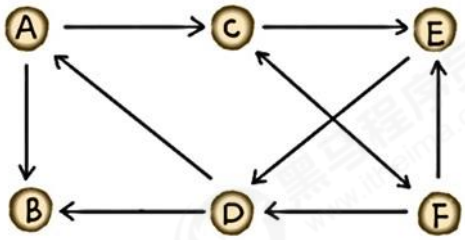
实际上，生活中的社交网络就是一个非常典型的图的结构，比如微博，微信，qq，每个人都有好友，可以互相关注，等等。

我们就拿微信举例子吧。我们可以把每个用户看作一个顶点。如果两个用户之间互加好友，那就在两者之间建立一条边。所以，整个微信的好友关系就可以用一张图来表示。其中，每个用户有多少个好友，对应到图中，就叫作顶点的**度（degree）**，就是跟顶点相连接的边的条数。

实际上，微博的社交关系跟微信还有点不一样，或者说更加复杂一点。微博允许单向关注，也就是说，用户 A 关注了用户 B，但用户 B 可以不关注用户 A。那我们如何用图来表示这种单向的社交关系呢？

我们可以把刚刚讲的图结构稍微改造一下，引入边的“方向”的概念。如果用户 A 关注了用户 B，我们就在图中画一条从 A 到 B 的带箭头的边，来表示边的方向。如果用户 A 和用户 B 互相关注了，那我们就画一条从 A 指向 B 的边，再画一条从 B 指向 A 的边。我们把这种边有方向的图叫作“有向图”。以此类推，我们把边没有方向的图就叫作“无向图”。

有向图

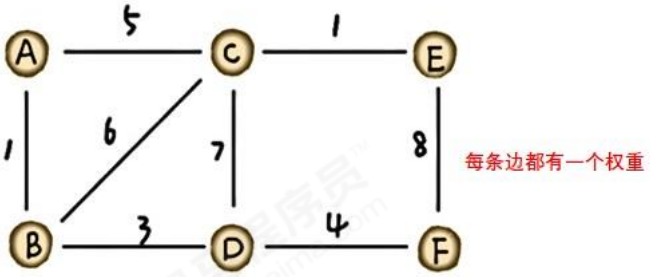


我们刚刚讲过，无向图中有“度”这个概念，表示一个顶点有多少条边。在有向图中，我们把度分为**入度（In-degree）和出度（Out-degree）**。

顶点的入度，表示有多少条边指向这个顶点；顶点的出度，表示有多少条边是以这个顶点为起点指向其他顶点。对应到微博的例子，入度就表示有多少粉丝，出度就表示关注了多少人。前面讲到了微信、微博、无向图、有向图，现在我们再来看另一种社交软件：QQ。QQ 中的社交关系要更复杂的一点。不知道你有没有留意过 QQ 亲密度这样一个功能。QQ 不仅记录了用户之间的好友关系，还记录了两个用户之间的亲密度，如果两个用户经常往来，那亲密度就比较高；如果不经常往来，亲密度就比较低。如何在图中记录这种好友关系的亲密度呢？

这里就要用到另一种图，**带权图（weighted graph）**。在带权图中，每条边都有一个**权重（weight）**，我们可以通过这个权重来表示 QQ 好友间的亲密度。

带权图

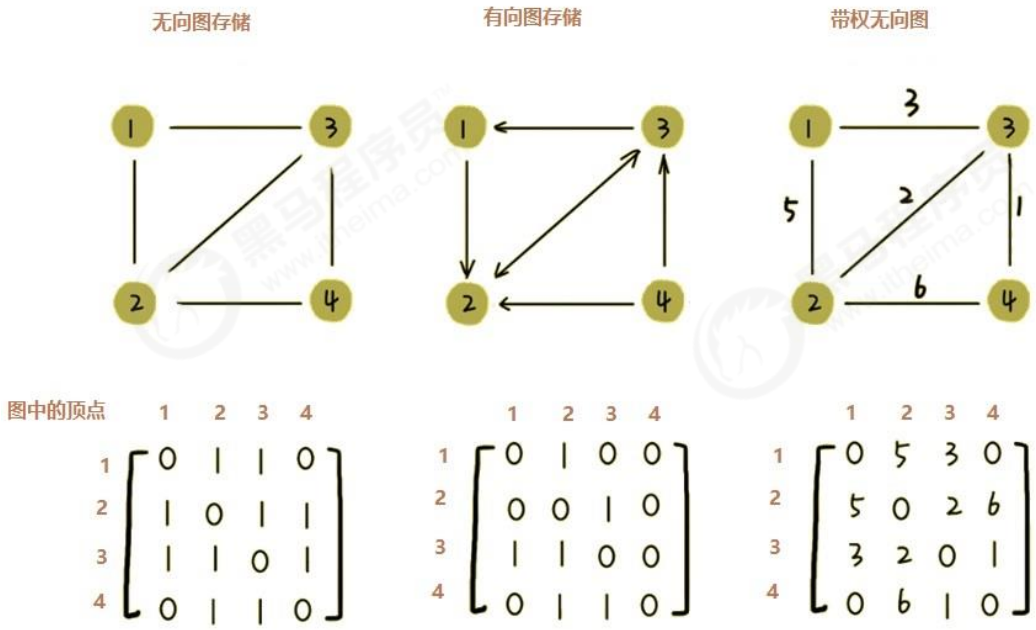


4.2.图的存储方式

理解了图的概念之后，那我们想如何在内存中存储一个图呢？其实图有很多中存储形式包括：邻接矩阵，邻接表，十字链表，邻接多重表，边集数组等等，在今天的课程中我们主要讲解其中的两种存储：邻接矩阵和邻接表

（1）、邻接矩阵存储图

邻接矩阵存储图底层依赖一个二维数组，对于无向图来说，如果顶点 i 与顶点 j 之间有边，我们就将 A[i][j] 和 A[j][i]标记为 1；对于有向图来说，如果顶点 i 到顶点 j 之间，有一条箭头从顶点 i 指向顶点 j 的边，那我们就将 A[i][j]标记为 1。同理，如果有一条箭头从顶点 j 指向顶点 i 的边，我们就将 A[j][i]标记为 1。对于带权图，数组中就存储相应的权重。

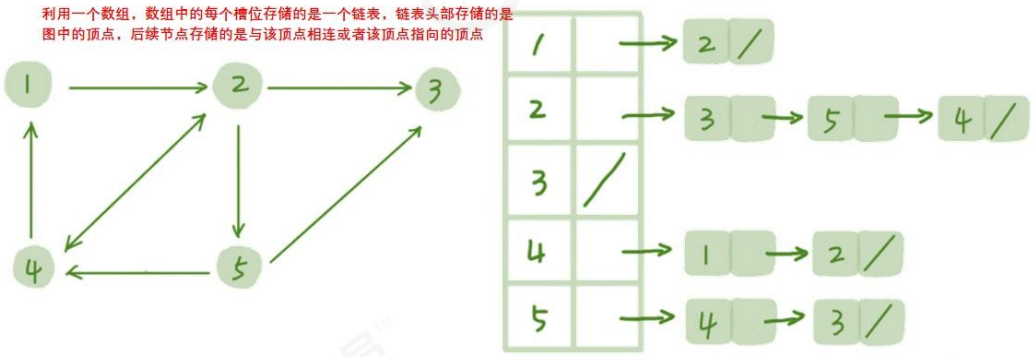


对于使用邻接矩阵来存储图的特点是：简单，直观。但是也有一定的缺点就是浪费存储空间，比如对于上面图示的情况来看第一种无向图的存储，A[i][j]等于 1 那么 A[j][i]也等于 1，在那个二维数组中我们沿着对角线划分为上下两部分，两部分其实是对称的，其实我们只需要一半的存储空间就够了，另一半算是浪费了，比如微信用户好几亿，但是每个用户的好友并不会很多，基本在三五百左右，这种情况下如果邻接矩阵来存储，很大一部分存储空间都浪费，所以我们可以总结出来邻接矩阵存储的适用场景。

适用场景：使用于稠密的图，可以快速定位到指定的边，但是如果是稀疏的图，会比较浪费空间。

（2）、邻接表存储图

针对上面邻接矩阵比较浪费内存空间的问题，我们来看另外一种图的存储方法，**邻接表（Adjacency List）**。下面用一幅图示来描述一下邻接表存储图



初步一看邻接表是不是有点像散列表？每个顶点对应一条链表，链表中存储的是与这个顶点相连接的其他顶点。另外图中画的是一个有向图的邻接表存储方式，每个顶点对应的链表里面，存储的是指向的顶点。对于无向图来说，也是类似的，不过，每个顶点的链表中存储的，是跟这个顶点有边相连的顶点。

当然了上面使用邻接矩阵存储图的好处是直观简单方便，但是缺点是浪费存储空间，相反的邻接表存储图的好处就是比较节省存储空间，但是缺点就是时间成本教高。

就像图中的例子，如果我们要确定，是否存在一条从顶点 2 到顶点 4 的边，那我们就要遍历顶点 2 对应的那条链表，看链表中是否存在顶点 4。在散列表那几节里，我讲到，在基于链表法解决冲突的散列表中，如果链过长，为了提高查找效率，我们可以将链表换成其他更加高效的数据结构，比如平衡二叉查找树等。我们刚刚也讲到，邻接表长得很像散列。所以，我们也可以将邻接表同散列表一样进行“改进升级”。

我们可以将邻接表中的链表改成平衡二叉查找树。实际开发中，我们可以选择用红黑树。这样，我们就可以更加快速地查找两个顶点之间是否存在边了。当然，这里的二叉查找树可以换成其他动态数据结构，比如跳表、散列表等。除此之外，我们还可以将链表改成有序动态数组，可以通过二分查找的方法来快速定位两个顶点之间否是存在边。

（3）、图存储的案例

前面我们讲到社交关系我们可以用图来进行存储，像微博社交关系我们可以使用有向图进行存储，微信社交关系可以使用无向图进行存储；我们课程第一天就讲到数据结构是为算法服务的，我们如何存储取决于我们要做什么操作，现假设有如下一个案例：

在微博社交中需要满足如下操作：

1：需要能够判断 A 用户是否关注了 B 用户

2：需要能够判断 C 用户是否是 D 用户的粉丝

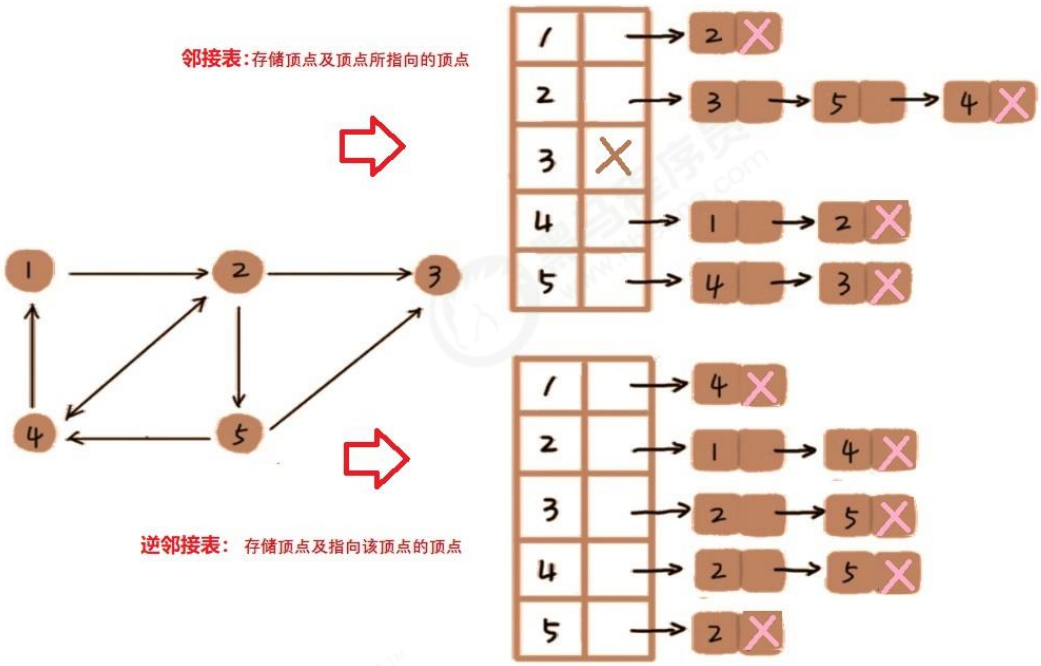
3：需要能够满足用户 E 去关注用户 F

4：需要能够根据用户名称排序获取该用户的粉丝列表

5：需要能够根据用户名称排序获取该用户所关注的用户列表

要想满足以上要求，首先从节省存储空间的角度来考虑我们选用邻接表来进行存储，但是只用邻接表的话我们能够满足的是判断 A 用户是否关注了 B 用户，我们只需要在 A 用户所在的链表上去查找是否存在 B 用户即可，存在则表明关注了不存在则表明没有关注；用户 E 去关注用户 F 需要进行的操作就是在用户 E 的链表中添加用户 F。

不过，用一个邻接表来存储这种有向图是不够的。我们去查找某个用户关注了哪些用户非常容易，但是如果要想知道某个用户都被哪些用户关注了，也就是用户的粉丝列表，是非常困难的。基于此，我们需要一个**逆邻接表**。邻接表中存储了用户的关注关系，逆邻接表中存储的是用户的被关注关系。对应到图上，邻接表中，每个顶点的链表中，存储的就是这个顶点指向的顶点，逆邻接表中，每个顶点的链表中，存储的是指向这个顶点的顶点。如果要查找某个用户关注了哪些用户，我们可以在邻接表中查找；如果要查找某个用户被哪些用户关注了，我们从逆邻接表中查找。



4.3.图的应用-搜索算法

我们知道，算法是作用于具体数据结构之上的，深度优先搜索算法和广度优先搜索算法都是基于“图”这种数据结构的，因为图这种数据结构的表达能力很强，大部分涉及搜索的场景都可以抽象成“图"。当然了这两种搜索算法不仅可以用在图上，也可以用在树上。

图上的搜索算法，最直接的理解就是，在图中找出从一个顶点出发，到另一个顶点的路径。具体方法有很多，比如今天要讲的两种最简单、最“暴力”的深度优先、广度优先搜索，还有 A*、IDA *等启发式搜索算法。接下来我们就来讲解一下基于图的两种搜索算法：广度优先搜索和深度优先搜索，关注这两种算法的实现我们使用的是邻接表来存储图，而且图采用的是无向图。所以我们先定义出来图的结构

```
/**
 * 基于邻接表实现的无向图
 *       该类中还实现了基于图的 BFS 和 DFS
 */
public class UndiGraph {
    // 图中顶点的个数
    private int points;

    // 邻接表：有点类似散列表,数组每个槽位链表头节点存储的是图中的顶点,链表其他节点存储的是与该顶点相连的顶点
```

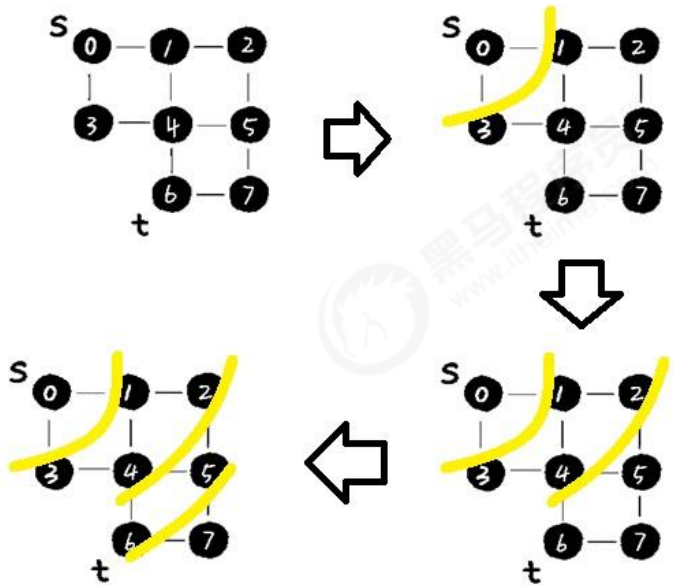
```
    private LinkedList<Integer> adjacencyList[];

    public UndiGraph(int points){
        this.points = points;
        //初始化数组
        adjacencyList = new LinkedList[this.points];
        // 初始化数组每个槽位上的链表
        for(int i=0;i < this.points;i++){
            adjacencyList[i] = new LinkedList<>();
        }
    }

    /**
     * 向图中添加顶点(边)
     * @param s
     * @param t
     */
    public void addEdge(int s,int t){
        // 无向图,一条边存储两回;注意：在该案例中,无向图中存储的顶点的值和数组的下标值保持对应
        adjacencyList[s].add(t);
        adjacencyList[t].add(s);
    }
}
```

4.3.1.广度优先搜索 BFS

广度优先搜索算法（Breadth-First-Search），是一种图形搜索算法，直观地讲，它其实就是一种“地毯式”层层推进的搜索策略，即先查找离起始顶点最近的，然后是次近的，依次往外搜索。理解起来并不难，所以我画了一张示意图，你可以看下



简单的说，BFS 是从根节点开始，沿着树(图)的宽度遍历树(图)的节点。如果所有节点均被访问，则算法中止,一般用队列数据结构来辅助实现 BFS 算法。

我们假设要从图中源顶点 S 找到目标顶点 T,我们搜索出一条从 S 到 T 的路径，则步骤如下：

算法步骤：

1. 首先将源顶点放入队列中。
2. 从队列中取出第一个顶点，并检验它是否为目标。如果找到目标，则结束搜寻并回传结果。否则将它所有尚未检验过的直接子顶点加入队列中。
3. 若队列为空，表示整张图都检查过了——亦即图中没有欲搜寻的目标。结束搜寻并回传“找不到目标”。
4. 重复步骤 2。

实际上我们这样求解出来的路径就是从 S 到 T 的最短路径。算法实现如下：

```
/**
     * 从源顶点到目标顶点的广度优先搜索 BFS
     * @param source
     * @param target
     */
    public void bfs(int source,int target){
        if(source == target){
            return;
        }
```

```
        // 定义一个boolean 数组记录顶点是否被访问过
        boolean[] visited = new boolean[this.points];
        visited[source] = true;
        // 定义一个队列,BFS 算法借助了一个队列
        Queue<Integer> queue = new LinkedList();
        // 将源顶点加入队列
        queue.add(source);
        // 定义一个数组，记录从源顶点到目标顶点之间的线路
        int[] prev = new int[this.points];
        for (int i = 0; i < prev.length; ++i) {
            prev[i] = -1;
        }
        //
        while (!queue.isEmpty()){
            // 取出队列中顶点元素
            Integer p = queue.poll();
            // 从邻接表中取出跟该顶点相连的顶点链表
            for(int j=0;j< adjacencyList[p].size();j++){
                // 依次取出跟顶点 p 相连的顶点 p
                Integer p_connect = adjacencyList[p].get(j);
                if(!visited[p_connect]){
                    //记录 p_connect 之前的顶点是 p
                    prev[p_connect] = p;
                    // 判断跟顶点 p 相连的顶点 p_connect 是否是目标顶点
                    if(p_connect == target){
                        //打印从源顶点到目标顶点之间的线路
                        print(prev,source,target);
                        return;
                    }
                    // 标识顶点 p 已被访问
                    visited[p] = true;
                    // 将顶点 p 相连的顶点 p_connect 加入队列
                    queue.add(p_connect);
                }
            }
        }
    }

    /**
     * 递归打印从 s 到 t 之间的线路
     * @param prev
     * @param s
     * @param t
     */
    private void print(int[] prev,int s,int t){
        if(prev[t]!=-1 && s != t){
            print(prev,s,prev[t]);
        }
```

```
        System.out.println(t+"->");
    }
```

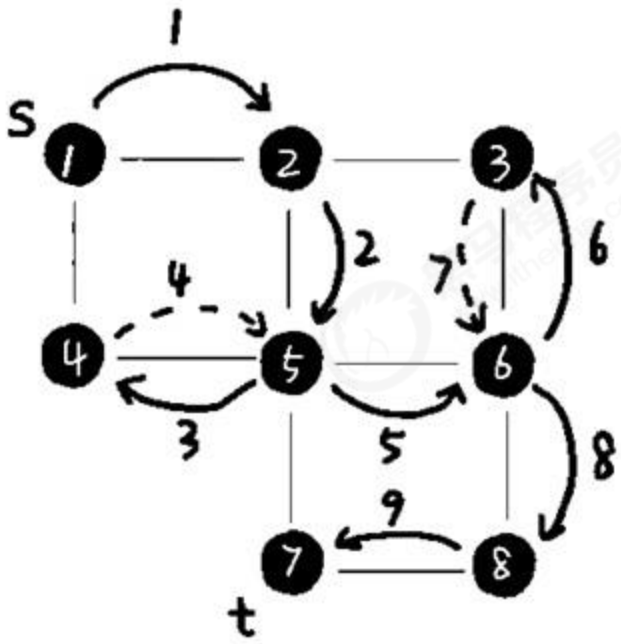
掌握了广优先搜索算法的原理，我们来看下，广度优先搜索的时间、空间复杂度是多少呢？最坏情况下，终止顶点 t 离起始顶点 s 很远，需要遍历完整个图才能找到。这个时候，每个顶点都要进出一遍队列，每个边也都会被访问一次，所以，广度优先搜索的时间复杂度是 O(V+E)，其中，V 表示顶点的个数，E 表示边的个数。当然，对于一个连通图来说，也就是说一个图中的所有顶点都是连通的，E 肯定要大于等于 V-1，所以，**广度优先搜索的时间复杂度也可以简写为 O(E)。**

广度优先搜索的空间消耗主要在几个辅助变量 visited 数组、queue 队列、prev 数组上。这三个存储空间的大小都不会超过顶点的个数，所以**空间复杂度是 O(V)**

4.3.2.深度优先搜索 DFS

深度优先搜索（Depth-First-Search），简称 DFS。最直观的例子就是“走迷宫”。假设你站在迷宫的某个岔路口，然后想找到出口。你随意选择一个岔路口来走，走着走着发现走不通的时候，你就回退到上一个岔路口，重新选择一条路继续走，直到最终找到出口。这种走法就是一种深度优先搜索策略。

走迷宫的例子很容易能看懂，我们现在再来看下，如何在图中应用深度优先搜索，来找某个顶点到
另一个顶点的路径。你可以看我画的这幅图。搜索的起始顶点是 s，终止顶点是 t，我们希望在图中寻找一条从顶点 s 到顶点 t 的路径。如果映射到迷宫那个例子，s 就是你起始所在的位置，t 就是出口。我用深度递归算法，把整个搜索的路径标记出来了。这里面实线箭头表示遍历，虚线箭头表示回退。从图中我们可以看出，深度优先搜索找出来的路径，并不是顶点 s 到顶点 t 的最短路径。



实际上，深度优先搜索用的是一种比较著名的算法思想，回溯思想。这种思想解决问题的过程，非常适合用递归来实现。回溯思想我们后面会学习到，我们现将 DFS 实现如下：

```
/**
     * 从源顶点到目标顶点的深度优先搜索 DFS
     *       需要借助一个全局的变量来记录是否找到目标顶点
     * @param source
     * @param target
     */
    private boolean found = false;

    public void dfs(int source,int target){
        if(source == target){
            return;
        }
        // 定义一个boolean 数组记录顶点是否被访问过
        boolean[] visited = new boolean[this.points];
        visited[source] = true;
        // 定义一个数组，记录从源顶点到目标顶点之间的线路
        int[] prev = new int[this.points];
        for (int i = 0; i < prev.length; ++i) {
            prev[i] = -1;
        }
```

```
        //递归调用
        recurDFS(source,target,visited,prev);
        //打印线路
        print(prev,source,target);
    }

    /**
     * 递归的查找顶点 p 到目标顶点之间的线路图
     * @param point
     * @param target
     * @param visited
     * @param prev
     */
    private void recurDFS(int point,int target,boolean[] visited,int[]
prev){
        if(found){
            return;
        }
        // 标记当前顶点已被访问
        visited[point] = true;
        // 如果当前顶点就是目标顶点
        if(point == target){
            found = true;
            return;
        }
        //获取与当前顶点相连接的所有顶点
        for(int j=0;j<adjacencyList[point].size();j++){
            // 获取与顶点 P 相连的顶点
            Integer p_connect = adjacencyList[point].get(j);
            if(!visited[p_connect]){
                //记录 p_connect 之前的顶点是 p
                prev[p_connect] = point;
                //递归的去找与 p_connect 相连的顶点
                recurDFS(p_connect,target,visited,prev);
            }
        }
    }
```

从我前面画的图可以看出，每条边最多会被访问两次，一次是遍历，一次是回退。所以，图上的**深度优先搜索算法的时间复杂度是 O(E)，E 表示边的个数**。深度优先搜索算法的消耗内存主要是 visited、prev 数组和递归调用栈。visited、prev 数组的大小跟顶点的个数 V 成正比，递归调用栈的最大深度不会超过顶点的个数，所以总的**空间复杂度就是 O(V)**。

5：字符串匹配/查找算法

在本章中我们要来学习字符串匹配算法，对于字符串匹配，每个工程师在实际的企业开发中都经常会用到，比如在 java 中提供的 indexOf()，starWith()，endWith()，这些方法的底层就依赖了字符串匹配算法。

当然了字符串匹配算法有很多，总体来说我们选择其中比较经典的几种算法来讲解：BF 算法，RK 算法，BM 算法，KMP 算法；以上这些算法都是单模式的字符串匹配算法，即一个串跟一个串进行匹配，除了单模式匹配算法我们还会讲解两种多模式字符串匹配，也就是在一个串中同时查找多个串，即 Trie 树和 AC 自动机。

5.1.BF 算法

原理

BF 算法，即**暴风(Brute Force)算法**，也叫**朴素匹配算法**，是普通的模式匹配算法，这种算法的字符串匹配方式很“暴力”，当然也就会比较简单、好懂，但相应的性能也不高，所以 BF 算法也是一种蛮力算法。

在正式讲解之前我们要先来明确几个概念，方便后续课程的讲解：模式匹配，主串和模式串

所谓模式匹配：即子串 P(模式串)在主串 T(目标串)中的定位运算，也称串匹配。假设我们有两个字符串：**T(Target, 目标串，主串)和 P(Pattern, 模式串，子串)**；在主串 T 中查找模式串 P 的定位过程，称为**模式匹配**。

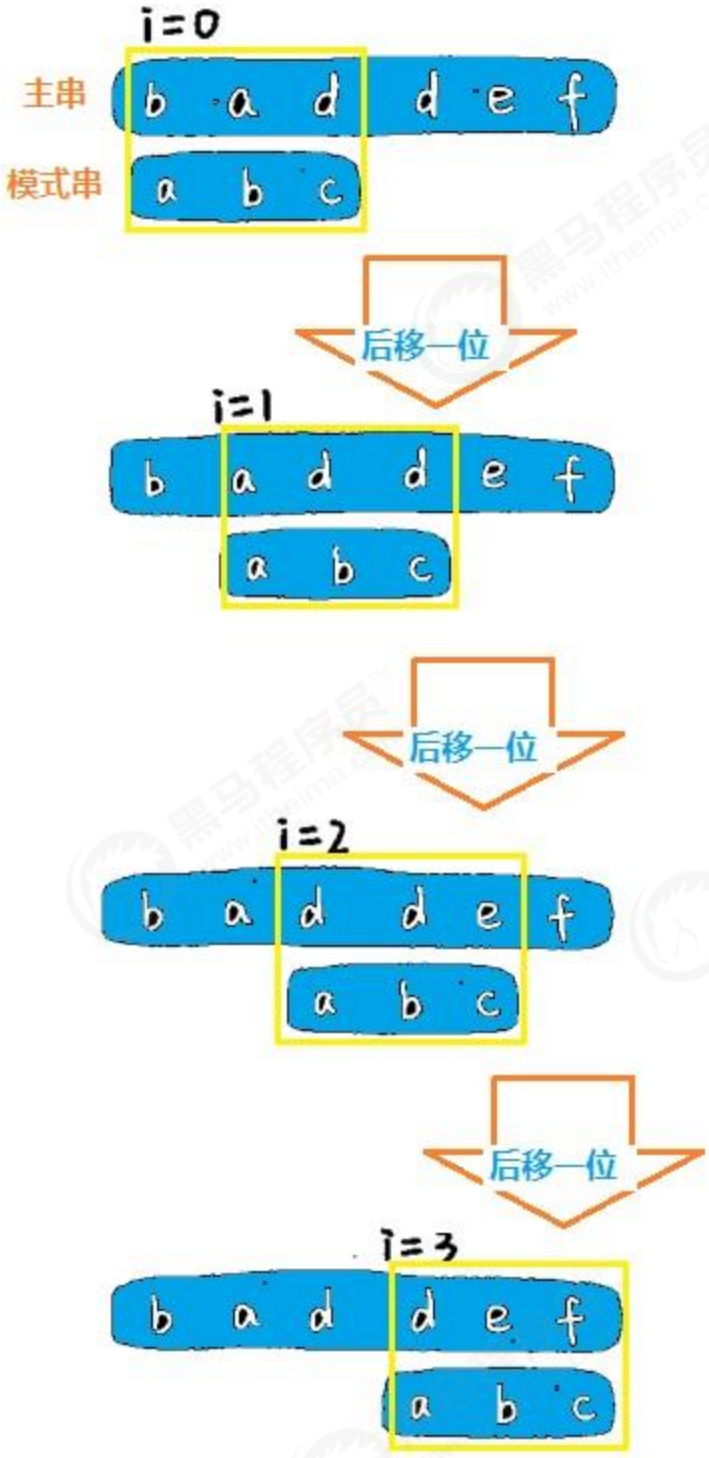
比方说，我们在字符串 A 中查找字符串 B，那字符串 A 就是主串(目标串 T)，字符串 B 就是模式串(字串 P)。我们把主串的长度记作 n，模式串的长度记作 m。因为我们是在主串中查找模式串，所以 n>m。

模式匹配有两种结果：

- 主串中找到模式为 P 的子串，返回 P 在 T 中的起始位置下标值；
- 未成功匹配，返回-1

BF 算法的思路：

我们在主串中，检查起始位置分别是 0、1、2...n-m 且长度为 m 的 n-m+1 个子串，看有没有跟
模式串匹配的，思路图如下：


i=0
主串
b a d d e f
模式串
a b c
后移一位
i=1
b a d d e f
a b c
后移一位
i=2
b a d d e f
a b c
后移一位
i=3
b a d d e f
a b c

实现

```
/**
 * BF 算法实现
 */
public class BF {

    /**
     * 使用 bf 算法在主串 t 中匹配模式串 p
     * @param t    主串/目标串
     * @param p    模式串/字串
     * @return     能匹配上则返回 p 在 t 中匹配的下标,匹配不上则返回-1
     */
    public int bf(String t,String p){
        if(t==null|| t.length()==0|| p==null|| p.length()==0|| p.length
()> t.length()){
            return -1;
        }
        // 将单一字符串转换成数组
        char[] t_array = t.toCharArray();
        char[] p_array = p.toCharArray();

        return match(t_array,p_array);
    }
    private int match(char[] t,char[] p){
        int i =0;// 主串下标
        int j=0;// 模式串下标
        int posi =0;//匹配到的位置

        while(i< t.length && j < p.length){

            if(t[i] == p[j]){ //如果匹配则目标串和模式串进行下一个字符的匹
配
                i++;
                j++;
            }else {//如果匹配不成功，则从目标字符串的下一个位置开始从新匹配
                i = i-j+1;
                j=0;
            }
        }
        if(i<=t.length){
            posi = i-p.length;
        }else {
            posi = -1;
        }
        return posi;
    }
    /*private int match(char[] t,char[] p){
```

```
        for(int i=0;i<t.length-p.length+1;i++){
            boolean isMatch = true;
            for(int j=0 ;j < p.length;j++){

                if(p[j] != t[j+i]){
                    isMatch = false;
                    break;
                }
            }
            if(isMatch){
                return i;
            }else {
                continue;
            }
        }
        return -1;
    }*/


    @Test
    public void test(){
        String a = "abccbadef";
        String b = "cba";
        System.out.println("匹配下标为:"+bf(a,b));
    }
}
```

从上面的实现情况可以看出：比如主串是“aaaaa...aaaaaa”（省略号表示有很多重复的字符 a），模式串是“aaaaab”。我们每次都比对 m 个字符，要比对 n-m+1 次，所以，这种算法的最坏情况时间复杂度是 O(n*m*)。
*尽管理论上，**BF 算法的时间复杂度很高，是 O(*n*m)，n 是主串的长度，m 是模式串的长度**，但在实际的开发中，它却是一个比较常用的字符串匹配算法。为什么这么说呢？原因有两点。

第一，实际的软件开发中，大部分情况下，模式串和主串的长度都不会太长。而且每次模式串与主串中的子串匹配的时候，当中途遇到不能匹配的字符的时候，就可以就停止了，不需要把 m 个字符都比对一下。所以，尽管理论上的最坏情况时间复杂度是 O(n*m)，但是，统计意义上，大部分情况下，算法执行效率要比这个高很多。

第二，朴素字符串匹配算法思想简单，代码实现也非常简单。简单意味着不容易出错，如果有 bug 也容易暴露和修复。在工程中，在满足性能要求的前提下，简单是首选。这也是我们常说的 KISS（Keep it Simple and Stupid）设计原则。

所以，在实际的软件开发中，绝大部分情况下，朴素的字符串匹配算法就够用了。

5.2.RK 算法

原理

RK 算法的全称叫 Rabin-Karp 算法，是由它的两位发明者 Rabin 和 Karp 的名字来命名的。它是 BF 算法的升级版，主要在这个算法中引入了哈希算法。

BF 算法中如果模式串长度为 m，主串长度为 n，那在主串中，就会有 n-m+1 个长度为 m 的子串，我们只需要暴力地对比这 n-m+1 个子串与模式串，就可以找出主串与模式串匹配的子串。但是，每次检查主串与子串是否匹配，需要依次比对每个字符，所以 BF 算法的时间复杂度就比较高，是 O(n*m)。我们对朴素的字符串匹配算法稍加改造，引入哈希算法，时间复杂度立刻就会降低；那如何引入哈希算法呢？

我们通过哈希算法对主串中的 n-m+1 个子串分别求哈希值，然后逐个与模式串的哈希值比较大小。如果某个子串的哈希值与模式串不相等，那这个字串和模式串必定不等，如果某个字串的哈希值与模式串的哈希值相等，则该字串与模式串不一定相等，因为有哈希冲突的存在，如果出现了哈希冲突后我们可以采用 BF 算法思想对该子串和模式串进行暴力匹配；因为哈希值是一个数字，数字之间比较是否相等是非常快速的，所以模式串和子串比较的效率就提高了。

不过，通过哈希算法计算子串的哈希值的时候，我们需要遍历子串中的每个字符。尽管模式串与子串比较的效率提高了，但是，算法整体的效率并没有提高。有没有方法可以提高哈希算法计算子串哈希值的效率呢？

这就需要我们设计良好的哈希函数，我们假设要匹配的字符串的字符集中只包含 R 个字符，我们可以用一个 R 进制数来表示一个子串，这个 R 进制数转化成十进制数，作为子串的哈希值。举个例子：

比如要处理的字符串只包含 a～z 这 26 个小写字母，那我们就用二十六进制来表示一个字符串。我们把 a～z 这 26 个字符映射到 0～25 这 26 个数字，a 就表示 0，b 就表示 1，以此类推，z 表示 25。在十进制的表示法中，一个数字的值是通过下面的方式计算出来的。对应到二十六进制，一个包含 a 到 z 这 26 个字符的字符串，计算哈希的时候，我们只需要把进位从 10 改成 26 就可以。

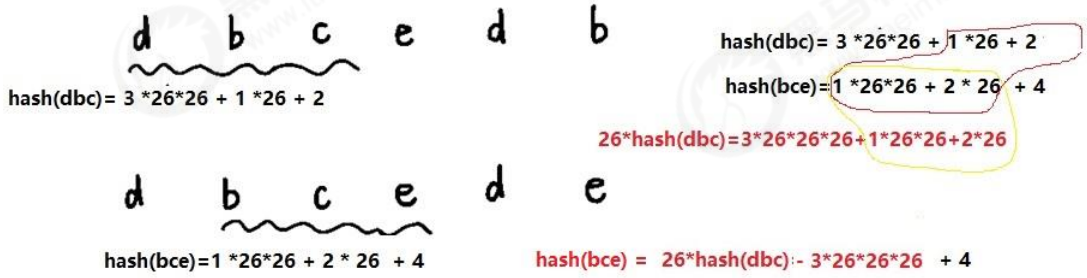
十进制 "657" = 6 * 10 * 10 + 5 * 10 + 7 * 1

二十六进制 "cba" = "c" * 26 * 26 + "b" * 26 + "a" * 1

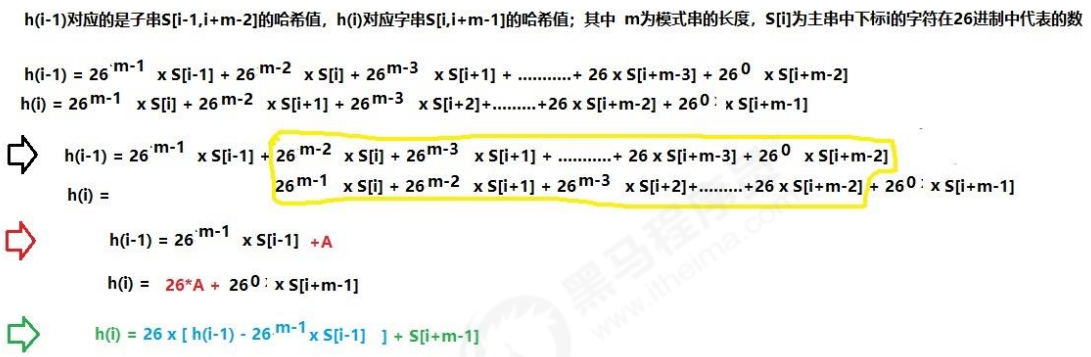
= 2 * 26 * 26 + 1 * 26 + 0 * 1

= 1353

这种哈希算法有一个特点，在主串中，相邻两个子串的哈希值的计算公式有一定关系，如下图



相邻两个子串 s[i-1] 和 s[i]（i 表示子串在主串中的起始位置，子串的长度都为 m），对应的哈希值计算公式有交集，也就是说，我们可以使用 s[i-1] 的哈希值很快的计算出 s[i] 的哈希值。如果用公式表示的话，就是下面这个样子



不过，这里有一个小细节需要注意，那就是 26^(m-1) 这部分的计算，我们可以通过查表的方法来提高效率。我们事先计算好 26^0 26^1 26^2.....26^m-1，并且存储在一个长度为 m 的数组中，公式中的“次方”就对应数组的下标。当我们需要计算 26 的 x 次方的时候，就可以从数组的下标为 x 的位置取值，直接使用，省去了计算的时间。

那如何来计算一个串的哈希值呢？

通过上面的分析我们可以将一个字符串看成 R=26 进制的数，这样在比较的时候比较其换算成十进制后的数字，但是可能一个 26 进制的数换算成 10 进制后太大，甚至有可能超过了 java 中 int 类型的范围，那这时该如何处理呢？我们还可以借助另一个哈希函数，取模运算。

取模运算：“%”, hash = key % M，其中 M 一般是素数，否则可能无法利用 key 中包含的所有信息。如 key 是十进制数而 M 是 10 的 k 次方，那么只能利用 key 的后 K 位，不均匀，增加碰撞概率。

%运算也适用于字符串，将字符串看成一个若干位 R 进制的整数，不同的是 R 并不一定非得大于其字符集的个数，可以适当进行调整，常常选取质数；通过我们刚刚分析的哈希值的求解过程可写出 hash 算法的大致实现如下：

```
public int hash(String s,int M){
    int hash = 0;
    int R = 101;     //进制
    for(int i=0;i<s.length();i++){
        hash = R * hash + s.charAt(i);
    }
    return hash % M;
}
```

这种计算方式当 charAt 方法返回的值过大或 R 太大时可能造成溢出，所以我们可以调整如下：

```
public int hash(String s,int M){
    int hash = 0;
    int R = 101;     //进制
    for(int i=0;i<s.length();i++){
        hash = (R * hash + s.charAt(i)) % M;
    }
    return hash % M;
}
```

这利用了%运算的基本性质，对于两个正整数 a、b，有：

```
(a+b)%n = (a%n + b%n) %n;
 (a-b)%n = (a%n - b%n + n) %n;
 (a*b)%n = (a%n * b%n) %n;
```

实现

```
/**
 * RK 算法实现
 */
public class RK {


    /**
     * 使用 bf 算法在主串 t 中匹配模式串 p
     * @param t     主串/目标串
     * @param p     模式串/字串
     * @return      能匹配上则返回 p 在 t 中匹配的下标,匹配不上则返回-1
     */
    public int rk(String t,String p){
        if(t==null|| t.length()==0|| p==null|| p.length()==0|| p.length
()> t.length()){
            return -1;
        }
        int pHash = hash(p,26,31,0,p.length());
        /*int tHash = hash(t,26,31,0,t.length());
        if(pHash == tHash && match(t,p,0)){
            return 0;
        }*/
        for(int i=0;i<t.length() - p.length()+1;i++){
            if(hash(t,26,31,i,p.length()) == pHash && match(t,p,i)){
                return i;
            }
        }
        return -1;
    }

    /**
     * 求一个字符串的哈希值
     * @param str         字符串
```

```
     * @param R          对应的进制
     * @param K          将字符串映射到 K 的范围内
     * @param start      从 str 串的 start 位置开始
     * @param len        模式串的长度 len
     * @return
     */
    private int hash(String str,int R,int K,int start,int len){
        int hash = 0;
        for(int i=start;i<start+len;i++){
            hash = (R * hash + str.charAt(i)) % K;
        }
        return hash % K;
    }

    /**
     * 当连个字符串的 hash 一样时则表明哈希冲突了,hash 值一样两个串不一定一样,
需要再依次比较一下。
     * @param t    主串
     * @param p    模式串
     * @param i    从主串下标为 i 的地方开始比较
     * @return
     */
    private boolean match(String t,String p,int i){
        for(int j=0;j<p.length();j++){
            if(p.charAt(j) != t.charAt(j+i)){
                return false;
            }
        }
        return true;
    }


    @Test
    public void test(){
        String a = "abcedfg";
        String b = "xyz";
        System.out.println("匹配下标为:"+rk(a,b));
    }
}
```

整个 RK 算法包含两部分，计算子串哈希值和模式串哈希值与子串哈希值之间的比较。第一部分，我们前面也分析了，可以通过设计特殊的哈希算法，只需要扫描一遍主串就能计算出所有子串的哈希值了，所以这部分的时间复杂度是 O(n)。模式串哈希值与每个子串哈希值之间的比较的时间复杂度是 O(1)，总共需要比较 n-m+1 个子串的
哈希值，所以，这部分的时间复杂度也是 O(n)。所以，RK 算法整体的时间复杂度就是 O(n)。跟 BF 算法相比，效率提高了很多。不过这样的效率取决于哈希算法的

设计方法，如果存在冲突的情况下，时间复杂度可能会退化。极端情况下，哈希算法大量冲突，时间复杂度就退化为 O(n*m)。

6：今日内容总结与作业安排

6.1.今日总结

堆：堆其实是一个满足特殊条件的树，其条件有：

1. 堆是一个完全二叉树；
2. 堆中每一个节点的值都必须大于等于（或小于等于）其子树中每个节点的值

堆的存储使用数组来存储，由于堆的特点使得用数组来存储堆访问起来更快。

堆排序：分为两步；第一步先建堆，第二步排序。

图的存储有两种形式：邻接矩阵和邻接表存储

图的两种应用：BFS 搜索算法和 DFS 搜索算法

单模式的字符串匹配算法：BF,RK

6.2.作业安排

1：实现一个大顶堆

2：实现一个小顶堆

3：实现堆排序算法

4：实现 BFS 算法

5：实现 DFS 算法

6：实现 BF 算法

7：实现 RK 算法