目录

大 0 表示法

为什么要有大 O 表示法

我们都知道，算法优劣程度的一个很重要的衡量标准就是算法的运行时间，所以我们需要一种衡量方式，来把算法的运行速度和数据项的个数关联在一起。大 0 表示法就是一种很好的表示方式。需要注意的一点是：大 0 表示法表示的时间复杂度均为最坏情况下的时间复杂度。关于最坏、最好情况会在下面介绍。

最好最坏情况说明

算法总的来说就是处理输入数据，得到输出数据，当输入数据不同时，同一个算法可能会得到不一样的运行时间，这是根据输入数据的情况来定的，我们把能够得到最小运行时间的输入情况叫做最好情况，把能够得到最大运行时间的输入情况叫做最坏情况。

大 O 计算时间复杂度例子

注意：这里我们只来研究时间复杂度，不谈代码。具体代码在后面的内容中都有。

1.向无序数组插入新数据

向无序数组中插入新数据时，总是插入到下一个有空的位置，所以每次插入新数据需要的时间都是一定的，这个时间只和自身电脑的一些硬件环境(微处理器、编译程序生成代码的效率)有关，所以我们可以认为每次插入新数据都需要相同的时间 k。k 为一个确定的常数。

2.线性查找

假定我们需要在某个数组里查找某个值，数组的长度为 n，一般思路就是从头遍历一遍，这样的话如果运气好，可能仅仅一次就可以找到，运气差的话则需要遍历整个数组（数组里无此值或者此值在最后一个）。由于在衡量算法效率时，我们考虑的都是最坏情况，所以若设每次查找操作花费的时间为 k，那么查找一个值所需要花费的时间就是 k*n。

3.二分查找

假定在排序好的数据中查找某个值，数组长度为 n。按照如下方式迭代(假设查找的值是 x，数组是 a)：

1. start=0,end=n-1,middle=(end+start)/2

2. 将 x 和 a[middle] 比较，若是 x<a[middle]，那么 end=middle-1，middle=(start+end)/2.若是大于，那么 start=middle+1，middle=(start+end/2).若是等于，那么直接退出。

3. 判断 start 是否大于 end，若是直接退出，否则执行 2

最坏情况下的执行次数怎么计算？

二分查找就是一种折半的查找，每次把数组的长度缩小一半，那么最坏的情况就是缩小到不能再缩才找到所需要查找的值。假设这时查找了 x 次，那么 2^x=n，所以 x=log(n)。

最坏情况要执行 log(n)次，那么需要的时间就是 k*log(n)。 注意：描述时间复杂度时 log 函数的默认底是 2 不是 10。

运行时间和大 O 的关系

大 O 表示法与上面的公式很类似，但是它省去了常数 K，因为我们比较算法的效率的时候，不在意微处理器芯片或者编译器，真正需要比较的是对应不同的 N 值，时间 T 是怎么变化的。由此，上述几种操作用大 O 表示法可以表示如下：

无序数组插入 O(1)

线性查找 O(N)

二分查找 O(log(N))

总之，大 O 表示法的参数实际就是算法执行的基本操作的次数所代表的数量级。

排序算法

排序问题引入

在我们的世界中充斥着各种的数据，不可避免的需要对这些数据进行排序，掌握有效的排序算法是很有必要的。

常见的排序算法：冒泡、选择、插入、桶、归并、基数、希尔、快速。

排序算法的稳定性

假定在待排序的记录中，存在着多个具有相同关键字的记录，若经过排序后，这些记录

的相对位置仍然不变，那么称这次排序是稳定的，否则是不稳定的。

排序算法稳定性的好处

主要体现在对多个属性的排序上，如果选择不稳定的排序算法，可能会导致错误的结果。

例子：比如学生有成绩和年龄两个属性，现在按照这两个属性来排序，先按照成绩升序来排，再按照年龄升序排，比如学生 A 成绩 90 年龄 18 ，学生 B 成绩 88 年龄 18 ，按照成绩排一遍，此时顺序是 BA，这时需要再按照年龄排一遍，如果选择不稳定的排序算法，可能顺序会变成 AB，这是错误的。错误的原因就是选择不稳定的排序算法可能会导致记录的相对位置改变。

排序算法的分类

根据排序过程中待排序文件存放位置的不同，可以把排序分为内部排序和外部排序两类。在排序过程中，所有需要排序的数都在内存，并在内存中调整它们的存储顺序，称为内排序；在排序过程中，只有部分数被调入内存，并借助内存调整数在外存中的存放顺序排序方法称为外排序。内部排序适用于记录个数不很多的较小待排序文件的排序；外部排序则适用于记录个数太多不能一次全部放入内存的较大待排序文件的排序。

在这里，我们暂时只讨论内部排序，外部排序会在海量数据处理部分介绍。

内部排序

冒泡排序

原理

假设有 n 个数，要对它们进行从小到大的排序，那么首先将最大的数移到最后一个位置，接着把第二大的数移到倒数第二个的位置，以此类推。那么经过 n-1 次操作之后，就完成了排序的过程。

考虑如何把最大的数移动到最后一个位置？

设置一个 index 变量，初始化为 0，假设待排序的数组是 a，每次比较 a[index]和 a[index+1]的大小，如果 a[index]<a[index+1]，那么不做任何操作，否则，交换 a[index]和 a[index+1]，每次操作完成之后 index++，这样当 index=a.length-2 时，就完成了比较的过程，最大的数就到了数组的最大索引处。类似操作都是这个原理。

代码

```
    /**
     * 冒泡排序  这里我们按照从小到大的顺序排列
     * @param a
     */
    private static void BubbleSort(int[] a) {

        int temp;
        for (int i = 0; i < a.length-1; i++) {
            for (int j = 0; j < a.length-i-1; j++) {
                if(a[j] > a[j+1]) {
                    temp = a[j];
                    a[j] = a[j+1];
                    a[j+1] = temp;
                }
            }
        }

    }
```

测试代码：

```
    /**
     * 测试
     * @param args
     */
    public static void main(String[] args) {
        int[] a = {21,25,55,44,11,22,36};
        BubbleSort(a);
        for (int i : a) {
            System.out.println(i);
        }
    }
```

时间复杂度

从代码可以看出有两层 for 循环，外层循环 n-1 次，内层循环次数递减，所以实际操作次数就大致可看成 1+2+3+…+n，数量级的角度来看，是 n^2，所以冒泡排序的时间复杂度是 O(N^2)。

空间

仅仅在交换变量的时候需要在内存中创建一个临时变量用于交换，所消耗的空间不随着数据项个数的增多而增多。所以空间复杂度是 O(1)。

稳定性

因为我们每次比较大小时都是用的<或者>号，所以两个数相等时它们的相对位置不会改变，所以冒泡排序是一种稳定的排序算法。

选择排序

原理

假设需要从小到大排列 n 个数，首先寻找最小的一个数，记录下他的索引，将他和第一个数交换，然后在剩下的 n-1 个数中寻找最小的，记录下索引，与第二个数交换，以此类推，经过 n-1 次操作，排序完成。

和冒泡排序相比的好处：减少了交换的次数，但是比较次数仍然没有减少。

代码

```
/**
 * 选择排序
 * @param a
 */
private static void selectSort(int[] a) {

    int min = Integer.MAX_VALUE;
    int minIndex = -1;
    int temp = 0;
    for (int i = 0; i < a.length-1; i++) {
        minIndex = i;
        min = a[i];
        for (int j = i+1; j < a.length; j++) {
            if(a[j] < min) {
                minIndex = j;
                min = a[j];
```

```
                }
            }
            //交换
            temp = a[i];
            a[i] = a[minIndex];
            a[minIndex] = temp;
        }

    }
```

测试代码：

```
    /**
     * 测试
     * @param args
     */
    public static void main(String[] args) {
        int[] a = {21,25,55,44,11,22,36};
        selectSort(a);
        for (int i : a) {
            System.out.print(i+" ");
        }
    }
```

时间复杂度

交换次数为 n，比较次数仍旧是 N^2 的数量级，所以时间复杂度是 O(N^2)。

空间

只需要常数量的额外内存保存 min、temp、minIndex 三个变量，所以需要的空间复杂度是 O(1)。

稳定性

看一个例子：5、8、5、2、9，第一趟交换第一个 5 和 2 会交换，那么两个 5 的相对位置就发生了改变，所以说选择排序是不稳定的排序算法。

插入排序

原理

按照如下的方式进行迭代(假设待排序的数组是 a[0...n-1])：
1.初始时，a[0]自成一个有序区，无序区为 a[1…n-1] 令 i=1
2.将 a[i]并入当前的有序区 a[0…i-1]形成 a[0…i]的有序区
3.i++并且重复第二步　直到 i==n-1

我们可以看出插入排序就是一种不断地增加有序区域的排序方法。直到有序区域等于待排序的整个数组，此时就完成了排序的过程。

考虑如何把一个元素插入到合适的位置？
首先，它前面的所有元素一定是排好序的，它需要首先找到比它大的数中最小的那一个，然后那个数到当前元素之间的所有元素都后移，然后把当前元素赋值到比它大的数中最小的那个数的位置上。这样就完成了元素插入到合适位置的过程。

代码

```
/**
 * 插入排序
 * @param a
 */
private static void insertSort(int[] a) {

    int index = 0;
    int temp = 0;
    for (int i = 1; i < a.length; i++) {
        if(a[i] > a[i-1] ) continue;
        index = i-1;
        while((index>=0) && a[index]>a[i]) {
            index--;
        }
        index++;
        temp = a[i];
        for (int j = i; j > index; j--) {
            a[j] = a[j-1];
        }
        a[index] = temp;
    }
```

```
    }
```

测试代码：

```
    /**
     * 测试
     * @param args
     */
    public static void main(String[] args) {
        int[] a = {21,25,55,44,11,22,36};
        insertSort(a);
        for (int i : a) {
            System.out.print(i+" ");
        }
    }
```

时间复杂度

外层循环是大约 n 次，内层最坏情况下长度递增，也是 1+2+…+n，也是 N^2，所以插入排序最坏情况下的时间复杂度是 O(N^2)。

空间

插入排序仅仅需要几个临时变量来辅助完成插入排序的过程，所以额外的内存消耗也是常量级的，所以插入排序的时间复杂度是 O(N^2)。

稳定性

稳定，一个数只有前面的数比它大，才会向前移动，所以不会改变两个相等元素的相对位置。

桶排序

原理

1.假设待排序的数据为 n 个，将 n 个数据放入 m 个桶中。
2.对每个桶中的数据进行排序
3，从头打印每个桶内的数据

总之，桶排序就是把待排序的数据分成若干组，每个组单独进行排序，最后汇总。具体分多少个桶，视具体的情况而定。

代码

```
/**
 * 桶排序
 * @param a
 * @param size 桶的大小
 * @param num  桶的数目
 */
private static void bucketSort(int[] a, int size, int num) {
    ArrayList<Integer>[] bucket = new ArrayList[num];
    //数据填到桶里
    for (int i = 0; i < a.length; i++) {
        int index = (a[i]%size==0)?a[i]/size-1:a[i]/size;
        if(bucket[index] == null) {
            bucket[index] = new ArrayList<Integer>();
        }
        bucket[index].add(a[i]);
    }
    for (int i = 0; i < bucket.length; i++) {
        if(bucket[i] != null) {
            Collections.sort(bucket[i]);
            System.out.print(bucket[i]);
        }
    }

}
```

测试代码：

```
/**
 * 测试
 * @param args
 */
public static void main(String[] args) {
    int[] a = {21,25,55,44,11,22,36};
    bucketSort(a, 10, 6);

}
```

时间复杂度

时间复杂度取决于两方面：桶映射函数和每个桶内数据的排序。

桶映射是很快的，一般是一个模运算，常量级。

N 个元素，那么分桶的时间为 O(N)，分桶之后我们需要进行的操作是排序，假设桶分的足够的均匀，各个数据分散的比较好，那么每个桶中基本不用再排序，此时的时间复杂度就是 O(N),，而若是分配的不均匀，大多数桶中都要进行排序，假设运行快速排序，O(log(N))，那么复杂度也达到了 O(N*log(N))。

空间

要开 m 个桶存储这 n 个元素，所以空间复杂度是 O(N)。

稳定性

稳定的排序算法，只要桶内部排序算法选择稳定的排序算法。

归并排序

原理

归并排序的核心便是分治法思想。

许多的算法在结构上是递归的，为了解决一个给定的问题。算法多次的调用自身，这些算法典型的遵循分治法的思想，将原问题分解为几个规模较小但是类似于原问题的子问题，递归的求解这些问题。然后合并这些子问题的解来建立原问题的解。

那么归并排序是如何应用分治法思想的呢？

假设待排序数组是 a，长度是 n，那么把 a 分成两等份(n 是奇数就有一份多一个)，分别对两份进行排序，排序完了之后再合并这两份，其中每一份在排序的过程中也应用归并排序。以此类推。

在排序的过程中就会不断地进行拆分，直到拆无可拆。然后开始合并。

归并排序需要两个方法：拆分方法(merge_sort)和合并(merge)方法。

代码

Merge_sort 方法：

```
/**
 *
 * @param a
 * @param p起始索引
```

```
     * @param r末尾的索引
     */
    private static void merge_sort(int[] a, int p, int r) {
        if(p < r) {
            int q = (p+r)/2;
            merge_sort(a, p, q);
            merge_sort(a, q+1, r);
            merge(a, p, q, r);
        }
    }
```

Merge 方法:

```
    /**
     * 完成合并的过程 这里假设两个子部分已经排序完成
     * @param a
     * @param p
     * @param q
     * @param r
     */

    public static void merge(int[] a, int p, int q, int r) {

        int[] tempArr = new int[r-p+1+p+1]; //缓存排序结果
        //注意这里的数组大小为什么不是r-p+1?为什么还要加上p+1
        //因为p有可能不是从0开始 所以开数组时必须加上p+1的大小 才不会让数组开的太小而导致索引越界

        int left = p;  //第一个子部分迭代到的位置
        int right = q + 1;    //第二个子部分迭代到的位置
        int index = p; //记录当前完成排序的截止位置的索引
        while (left <= q && right <= r) {
            // 从两个数组中取出最小的放入临时数组
            if (a[left] <= a[right]) {
                tempArr[index++] = a[left++];
            } else {
                tempArr[index++] = a[right++];
            }
        }
        // 剩余部分依次放入临时数组（实际上两个while只会执行其中一个）
        while (left <= q) {
            tempArr[index++] = a[left++];
        }
        while (right <= r) {
            tempArr[index++] = a[right++];
        }
```

```
        //排序结果赋值给a
        for (int i = p; i < r+1; i++) {
            a[i] = tempArr[i];
        }
    }
```

测试代码：

```
    /**
     * 测试
     * @param args
     */
    public static void main(String[] args) {
        int[] a = {21,25,55,44,11,22,36};
        merge_sort(a, 0, a.length-1);
        for (int i : a) {
            System.out.print(i+" ");
        }
    }
```

时间复杂度

时间复杂度是 O(N*log(N))。具体推导过程可见算法导论。

空间

需要开一个等于待排序数组大小的临时数组，所以空间复杂度是 O(N)。

稳定性

稳定的排序算法。因为在合并的过程中，有 L 和 R 两个数组，L 在先，R 在后，若是 L 和 R 中有元素大小相等或者 L 中两个元素大小相等或者是 R 中两个元素大小相等，他们一定会按照顺序添加到临时数组中，是绝对不会改变相等元素的相对位置的。

基数排序

原理

与桶排序类似，但是基数排序的桶的个数恒为 10 个，编号为 0-9。
例子如下：假设待排序数组是 62,14,59,88,16，首先按照个位入桶，则 62 进入桶 2，14

进入桶 1，……，以此类推，此时输出是 62,12,16,88,59，个位入桶结束后，按照 10 位入桶，若是 10 位相同，那么个位入桶在前的数先入桶，以此类推…，最多有几位数，就入几次桶，即可完成排序。

代码

```
    /**
     * 基数排序
     * @param a待排序的数组
     * @param distance最大数的位数
     */
    public static void radixSort(int[] a, int distance) {
        int len = a.length;
        int[] temp = new int[len];    //缓存数组
        int[] count = new int[10];
        /**
         * count数组十分的关键
         * count[i]存储的就是比指定位上比i小或者等于的值得个数
         * 这样在排序的时候  就按照count[i]的值来安排应该把值安排到什么位置
         * 每次安排完了count[i]--
         */
        int divide = 1;    //基数  开始时候是1 代表先按照个位入桶
        for (int i = 0; i < distance; i++) { //有多少位就入多少次桶
            //copy一个a数组副本
            System.arraycopy(a, 0, temp, 0, len);
            Arrays.fill(count, 0);

            //入桶
            for (int j = 0; j < a.length; j++) {
                int num = (temp[j]/divide)%10;   //得到相应的位数上的值
                                                 //divide为1时就是得到个位的值
                count[num]++;  //此时count[0]代表该位上值为0的个数
                               //count[1]代表该位上值为1的个数
            }
            for (int j = 1; j < 10; j++) {
                count[j]+=count[j-1]; //递推过程
                                      //count[j]代表该位上值小于等于j的值得
个数
            }

            /**
             * 这个过程十分的关键
```

```
             * 要保留上一次的结果 必须倒着遍历
             * 因为上一次的结果大的在后面
             * 并且count中存储的值在运行过程中会递减 为了避免错位
             */
            for (int j = a.length-1; j >= 0; j--) {
                int num = (temp[j]/divide)%10;   //得到相应的位数上的值
                count[num]--;
                a[count[num]] = temp[j];
            }
            divide*=10;    //基数乘10
        }
    }
```

测试代码：

```
/**
     * 测试
     * @param args
     */
    public static void main(String[] args) {
        int[] a = {21,25,55,44,11,22,36};
        radixSort(a, 2);
        for (int i : a) {
            System.out.print(i+" ");
        }
    }
```

时间复杂度

基数排序的时间复杂度是 O(d*N)。d 代表最大数的位数。

空间

假设 d 是最大数的位数，那么要入桶 d 次，每次都要申请一个原数组大小的临时数组，所以空间复杂度是 O(n*d)。

稳定性

稳定的排序算法，两个相等的数会按照顺序先后入同一个桶，出来的时候也会按照顺序出来，没有产生什么交换操作。

希尔排序

原理

先将整个待排序序列分割成若干个子序列(由相隔某个“增量”的元素组成)，分别进行插入排序，然后依次缩减增量再进行排序，当增量为 1 时，对整个序列来一次直接插入排序，即可完成排序。

代码

```
    /**
     * 希尔排序
     */
    public static void shellSort(int[] a) {
        int len = a.length;
        for (int i = len>>1; i > 0; i>>=1) { //每次都缩小一半 这里使用位操作符
            /**
             * 插入排序的过程
             */
            for (int j = i; j < a.length; j++) {
                int temp = a[j];
                int k = j;
                for (; k >=i && a[k-i] > temp ; k-=i) {
                    a[k] = a[k-i];
                }
                a[k] = temp;
            }
        }
    }
```

时间复杂度

平均 O（N^1.3）。这只是科学家根据实验得出的一个大体值。准确的复杂度无法计算。

空间

不需要额外的空间，O(1)。

稳定性

不稳定。因为按照增量来排序，很可能因为交换而改变两个相等的值的相对位置。

快速排序

原理

分治思想，给定待排序数组 a[0…n-1],以 a[0]为关键字，把数组分为 a[0…k]和 a[k+1…n-1],使得第一个数组中的数都小于 a[0]，第二个数组中的数都大于 a[0]，重复执行上述过程（递归），即可完成排序。

快速排序的实现和归并排序十分的类似。都是有两个方法完成，快速排序的两个方法一个是完成划分过程，将数组按照某个元素划分成两部分，左面的都小于这个元素，右面的都大于这个元素，一个是完成递归调用的过程。

代码

划分方法：

```
/**
 * 快速排序--辅助方法
 * 将原数组划分成为两个部分  使得左面的都小于某个数 右面的都大于某个数
 */
public static int partition(int[] a, int left, int right) {
    int num = a[left];//使用第一个元素作为划分依据元素
    while(left < right) {
        //完成划分过程 使得左面的都小于某个数 右面的都大于某个数
        while(a[right] > num) right--;
        a[left] = a[right];
        while((left<right)&&a[left]<num) left++;
        a[right] = a[left];
    }
    a[left] = num;
    return left;
}
```

递归调用过程：

```
/**
 * 快速排序
 */
public static void quickSort(int[] a, int left, int right) {
    if(left < right) {
```

```
            int n = partition(a, left, right);
            quickSort(a, left, n-1);
            quickSort(a, n+1, right);
        }
    }
```

测试代码：

```
    /**
     * 测试
     * @param args
     */
    public static void main(String[] args) {
        int[] a = {21,25,55,44,11,22,36};
        quickSort(a, 0, a.length-1);
        for (int i : a) {
            System.out.print(i+" ");
        }
    }
```

时间复杂度

最坏情况下，每次划分都只减少一个元素，那么快速排序就退化为了冒泡排序，时间复杂度就是 O(N)。一般情况下，冒泡排序的时间复杂度是 O(n*log(N))。快速排序是最快的排序算法。并不是说时间复杂度小的就一定快，时间复杂度只是一种对算法效率的量化，反映了算法运行效率的大体数量级。

空间

需要使用一个长度为 log2(N)的栈实现递归。所以时间复杂度就是 O(log(N))。

稳定性

快速排序是不稳定的。在中枢元素和 left 指向的元素或者 right 指向的元素交换时，可能会改变相同元素的相对位置。

各种排序算法的比较

算法名称	最坏时间	平均时间	空间	稳定性
冒泡	N^2	N^2	1	稳定
选择	N^2	N^2	1	不稳定

插入	N^2	N^2	1	稳定
桶	N^2	N	N	稳定
基数	D*N	D*N	D*N	稳定
希尔	N^2	N^1.3	1	不稳定
快速	N^2	N^LOG(N)	N*LOG(N)	不稳定
归并	N*LOG(N)	N*LOG(N)	N	稳定

数据结构

为什么不用数组表示一切？

为什么我们不使用数组来完成所有数据的存储呢？原因何在？数组的特点：

1.无序数组中可以快速的插入 O(1)

2.查找　遍历:O(N)　二分:O(log(N))

3.删除　O(N)

4.创建后大小不可改变

我们可以看出数组的插入是很快的，但是删除和查找是比较慢的，如果解决一些需要大量删除和查找操作的问题，只用数组来存储数据显然是不可以的。

没有最好的数据结构，只有最合适的数据结构。

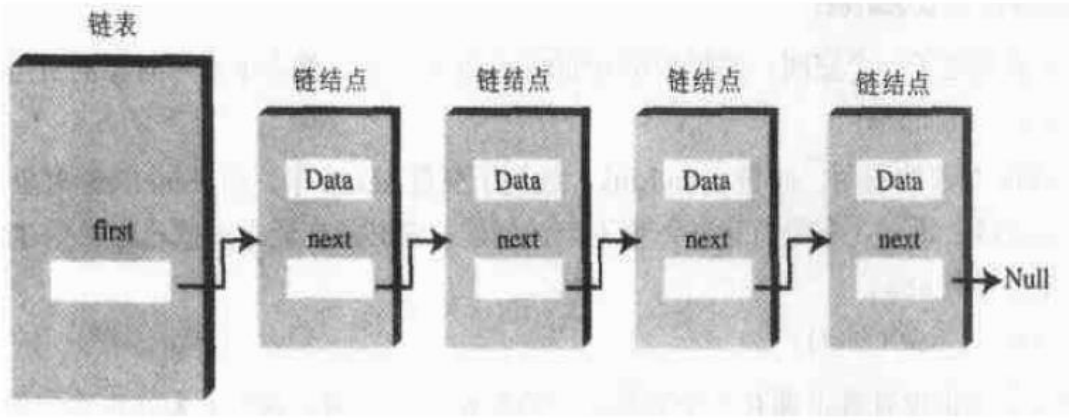
链表

链表可以说是弥补了数组的一些缺点。它的插入和删除操作很快，但是查找很慢，不能像数组那样按照索引下标来进行访问。

要了解链表这种数据结构，我们必须先了解以下几个概念。

链节点

在链表中，每个数据项都被包含在“链节点“中，一个链节点是某个类的对象，这个类可以叫做 Link，一个链表中有许多类似的链节点，每个 Link 对象都保存着对下一个链节点的引用，链表本身对象中有一个字段指向对第一个链节点的引用。关系如下图：



关系，而不是位置

在数组中，每一项占有一个特定的位置，这个位置可以使用一个下标号直接访问，它就像一排房子，你可以凭借地址找到其中特定的一间。

在链表中，寻找一个特定元素的唯一方法就是沿着这个元素的链一直向下寻找。

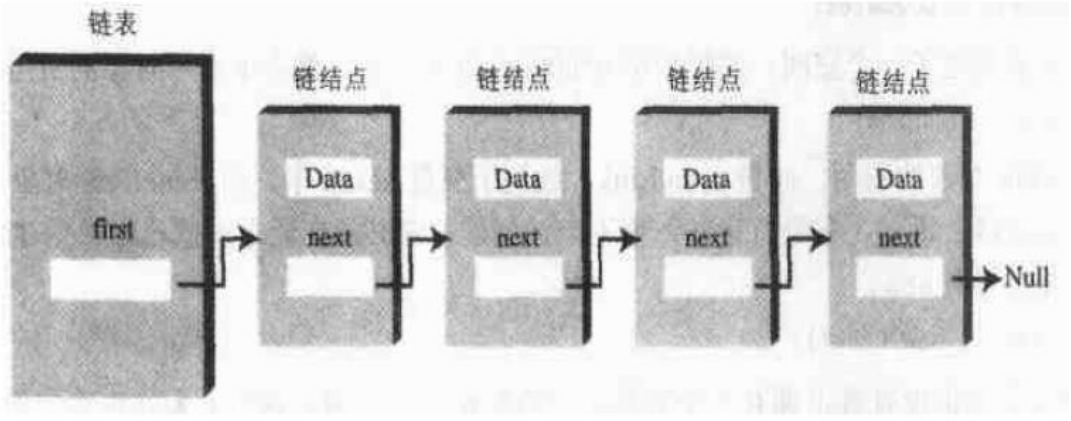
它强调的是链节点之间的关系，而不是链节点的位置。

链表的分类

常见的链表有四种，单向链表、双端链表、双向链表、有序链表，它们的实现略有差异，但是基本的构成元素都是上面提到的 Link。下面将一一介绍：

单向链表

单向链表是最普通、最常见的链表，每个链节点都只保存对下一个链节点的引用，并且遍历方向只有一个。如下图就是最常见的单向链表：



要实现单向链表，我们需要两个类，一个 Link 类，代表链节点，其中包含数据项和对下一个链节点的引用。一个 LinkList 类，代表链表类，它保存对第一个链节点的引用。还有一些常见的操作。

代码如下：

Link 类：

```
/**
 * 单向链表链节点的实现
 * @author yy
 *
 */
public class Link {

    //链节点包括两部分内容    数据项和指向下一个链节点的引用
    //这里假设我们的数据项是float类型的变量  即用链表来完成float数组所能完成的存储功能
    public float data;//数据项
    public Link next = null;  //指向下一个链节点的引用

    public Link(float data) {
        this.data = data;
    }

    @Override
    public String toString() {
        return "Link [data=" + data + "]";
    }

}
```

LinkList 类：

```
package com.algorithm.link;


/**
 * 单向链表的实现
 * @author yy
 *
 */
public class LinkList {

    //链表就是由若干个链节点完成的    链表对象中必须有一个字段指向对第一个链节点的引用
```

```java
private Link first;   //指向第一个链节点

public LinkList() {
    //完成对first变量的初始化  置为null
    first = null;
}

//判断链表是否为空 只要first有指向的节点 即first不为空 那么链表就不为空
public boolean isEmpty() {
    return first == null;
}

//在链表的首部添加一个节点
public void insertFirst(float data) {
    Link link = new Link(data);
    link.next = first;
    first = link;
}

//在链表的首部删除一个节点 返回被删除的节点
public Link deleteFirst() {
    Link temp = first;
    first = first.next;
    return temp;
}


//在链表的尾部添加一个节点
public void insertTail(float data) {
    Link link = new Link(data);
    if(first == null) {
        first = link;
        return;
    }
    Link current = first;
    while(current.next != null) {
        current = current.next;
    }
    current.next = link;
}

//在链表的尾部删除一个节点
public void deleteTail() {
    Link current = first;
```

```
    if(current == null) return;
    if(current.next==null) {current = null;return;};
    while(current.next.next != null) {
        current = current.next;
    }
    current.next = null;
}


//根据数据项查找某个链节点
public Link find(float key) {
    Link current = first;
    while(current != null && current.data != key) {
        if(current.next == null) return null;
        current = current.next;
    }
    return current;
}


//根据给定数据项删除某个链节点
public Link delete(float key) {
    Link previous = first;
    Link current = first;
    while(current.data != key) {
        if(current.next == null) return null;
        previous = current;
        current = current.next;
    }
    if(current == first) {
        first = first.next;
    }else {
        previous.next = current.next;
    }
    return current;
}


//遍历一遍链表 并且打印
@Override
public String toString() {

    StringBuffer buffer = new StringBuffer();
    Link current = first;
```

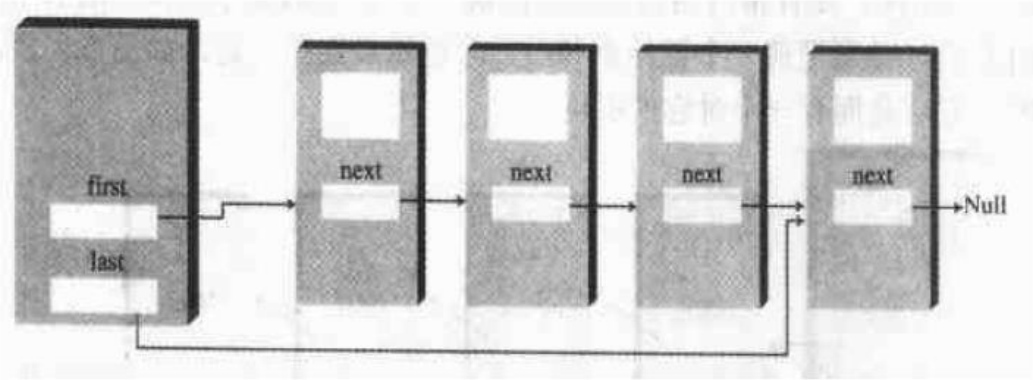
```
        while(current != null) {
            buffer.append(current.data+" ");
            current = current.next;
        }

        return buffer.toString();
    }

}
```

测试代码：

```
package com.algorithm.link;

public class Main {

    public static void main(String[] args) {
        LinkList linkList = new LinkList();
//      linkList.insertFirst(12);
//      linkList.insertFirst(2231);
//      linkList.insertFirst(232);
        linkList.insertTail(11);
        linkList.insertTail(222);
        linkList.insertTail(33);
        System.out.println(linkList);
//      linkList.deleteFirst();
//      linkList.deleteTail();
//      linkList.deleteTail();
//      System.out.println(linkList);
        System.out.println(linkList.delete(222).data);
        System.out.println(linkList);
    }
}
```

双端链表

双端链表和传统链表唯一不同的地方就是双端链表增加了对最后一个链节点的引用，如下图：



代码如下：

Link 类：

```
package com.algorithm.firstlastlink;

/**
 * 双端链表链节点   和普通链表的链节点相同
 * @author yy
 *
 */
public class Link {

    public float data;

    public Link next;

    public Link(float data) {
        this.data = data;
    }

    @Override
    public String toString() {
        return "Link [data=" + data + "]";
    }

}
```

FirstLastList 类：

```
package com.algorithm.firstlastlink;

/**
 * 双端链表
 * @author yy
```

```
 *
 */
public class FirstLastList {

    private Link first;   //指向第一个链节点
    private Link last;//指向最后一个链节点

    public FirstLastList() {
        //完成first和last变量的初始化
        first = null;
        last = null;
    }

    //判断链表是否为空
    public boolean isEmpty() {
        return first == null;
    }

    //开头插入一个数据
    public void insertFirst(float data) {
        Link link = new Link(data);
        if(isEmpty()) {
            last = link;
        }
        link.next = first;
        first = link;
    }

    //结尾插入一个数据
    public void insertLast(float data) {
        Link link = new Link(data);
        if(isEmpty()) first = link;
        else last.next = link;
        last = link;
    }

    //删除开头第一个数据
    public float deleteFirst() {
        float temp = first.data;
        if(first.next == null) {
            last = null;
        }
        first = first.next;
        return temp;
```

```
    }

}
```

测试代码：

```
package com.algorithm.firstlastlink;

/**
 * 测试双端链表
 * @author yy
 *
 */
public class Main {

    public static void main(String[] args) {
        FirstLastList firstLastList = new FirstLastList();
        firstLastList.insertFirst(11);
        firstLastList.insertFirst(22);
        firstLastList.insertLast(44);
        System.out.println(firstLastList.deleteFirst());
        System.out.println(firstLastList.deleteFirst());
        System.out.println(firstLastList.deleteFirst());
    }

}
```

有序链表

在某些链表中，需要保持对数据的有效性，具有这个特性的链表叫做”有序链表“。有序链表中，都是按照关键值有序排列的，

有序链表优于有序数组的地方：一是插入速度快，因为不需要元素的移动，二是链表可以扩展到全部有效的内存，而数组只能局限于一个固定的大小之中。

代码如下：

Link 类：

```
package com.algorithm.orderlink;


/**
 * 有序链表链节点
 * @author yy
 *
```

```
 */
public class Link {

    public int data;

    public Link next;

    public Link(int data) {
        this.data = data;
    }

    public int getKey() {
        return data;
    }


    @Override
    public String toString() {
        return "Link [data=" + data + "]";
    }



}
```

OrderList 类：

```
package com.algorithm.orderlink;

/**
 * 有序链表的实现
 * @author yy
 *
 */
public class OrderList {

    private Link first;

    public OrderList() {
        first = null;
    }

    public boolean isEmpty() {
        return first == null;
    }
```

```java
/**
 * 插入新值
 * @param key
 */
public void insert(int key) {
    Link newLink = new Link(key);
    Link previous = null;
    Link current = first;
    while(current!=null && current.data<key) {
        //如果当前值小于key，那么就继续向下遍历
        previous = current;
        current = current.next;
    }
    if(previous == null) {
        first = newLink;
    }else
        previous.next = newLink;
    newLink.next = current;
}

public Link find(int key) {
    Link current = first;
    while(current!=null && current.data!=key) {
        current = current.next;
    }
    return current;
}

public void delete(int key) {
    Link previous = null;
    Link current = first;

    while(current != null && key!=current.getKey()) {
        previous = current;
        current = current.next;
    }
    if(previous == null) {
        first = first.next;
    }else {
        first.next =current.next;
    }
}
```

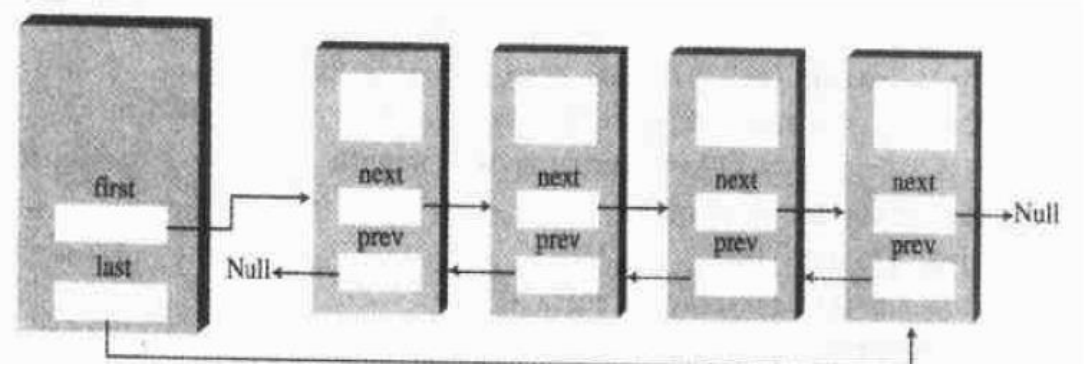
```
    public void display() {
        Link current = first;
        while(current != null) {
            System.out.print(current.data+" ");
            current = current.next;
        }
        System.out.println();
    }


}
```

测试代码：

```
package com.algorithm.orderlink;

public class Main {

    public static void main(String[] args) {
        OrderList list = new OrderList();
        list.insert(22);
        list.insert(223);
        list.insert(15);
        list.insert(1);
        list.display();
    }

}
```

双向链表

双向链表既可以从头往后遍历，也可以从后往前遍历。原因是双向链表的每个链节点都保留两个引用，一个指向下一个，一个指向上一个。结构图如下：



代码如下：

Link 类：

```
package com.algorithm.doublelink;


/**
 * 双向链表 链节点
 * @author yy
 *
 */
public class Link {

    public float data;
    public Link previous;
    public Link next;

    public Link(float data) {
        this.data = data;
    }

    @Override
    public String toString() {
        return "Link [data=" + data + "]";
    }

}
```

DoubleLinkedList 类：

```
package com.algorithm.doublelink;

public class DoubleLinkedList {

    private Link first;
    private Link last;//这里采用双端的方式

    public DoubleLinkedList() {
        first = null;
        last = null;
    }

    public boolean isEmpty() {
        return first == null;
```

```
    }

    //开头插入元素
    public void insertFirst(float data) {
        Link newLink = new Link(data);
        if(first == null) {
            last = newLink;
        }else {
            first.previous = newLink;
        }
        newLink.next = first;
        first = newLink;

    }

    //结尾插入元素
    public void insertLast(float data) {
        Link newLink = new Link(data);
        if(isEmpty()) {
            first = newLink;
        }else {
            last.next = newLink;
            newLink.previous = last;
        }
        last = newLink;
    }

    //开头删除元素
    public Link deleteFirst() {
        Link temp = first;
        if(first.next == null) {
            last = null;
        }else {
            first.next.previous = null;
        }
        first = first.next;
        return temp;
    }

}
```

测试代码：

```
package com.algorithm.doublelink;
```

```java
/**
 * 双向链表的测试
 * @author yy
 *
 */
public class Main {

    public static void main(String[] args) {
        DoubleLinkedList doubleLinkedList = new DoubleLinkedList();
        doubleLinkedList.insertFirst(11);
        doubleLinkedList.insertFirst(22);
        doubleLinkedList.insertLast(222);
        doubleLinkedList.insertLast(1);
        System.out.println(doubleLinkedList.deleteFirst());
        System.out.println(doubleLinkedList.deleteFirst());
        System.out.println(doubleLinkedList.deleteFirst());
        System.out.println(doubleLinkedList.deleteFirst());
    }

}
```

栈

原理

栈只允许访问一个数据项，即最后一个插入的元素。移除最后一个元素之后才能访问倒数第二个元素，以此类推。
栈是一种后进先出的数据结构。

栈的几种常见操作:

Push：插入一个元素到栈顶

Pop ：删除栈顶中的元素

Peek ：查看栈顶的元素

要实现一个栈，一般有两种方式，基于链表实现和基于数组实现。具体使用哪种方式，视情况而定，若是栈的最大值确定，那么使用数组实现开销最小，否则使用链表实现，因为链表适合实现无上限的栈。在这里，我们使用数组来实现一个栈，链表的可自行研究。

代码

StackX 类：

```
package com.algorithm.stack;

/**
 * 使用数组来实现栈
 * @author yy
 *
 */
public class StackX {

    private int maxSize;  //栈的大小 构造函数中完成初始化
    private char[] stackArray;    //存储栈中数据的数组
    private int top;  //栈顶指针  代表栈顶元素的索引

    public StackX(int s) {
        maxSize = s;
        stackArray = new char[maxSize];
        top = -1;  //初始值-1
    }

    //入栈操作
    public void push(char i) {
        stackArray[++top] = i;
    }

    //出栈操作
    public char pop() {
        return stackArray[top--];
    }


    //查看栈顶元素
    public char peek() {
        return stackArray[top];
    }


    //判断栈是否空
    public boolean isEmpty() {
        return top == -1;
    }

    //判断栈是否满
    public boolean isFull() {
```

```
        return top == maxSize-1;
    }


    /**
     * 栈测试
     * @param args
     */
    public static void main(String[] args) {
        StackX stackX = new StackX(10);
        stackX.push('1');
        stackX.push('3');
        stackX.push('4');
        stackX.push('5');
        stackX.push('6');

        while(!stackX.isEmpty()) {
            System.out.println(stackX.pop());
        }

    }


}
```

栈的应用：分隔符匹配

栈通常用于分隔符的匹配。分隔符匹配的问题就是在一段文本中'['和']'、'{'和'}'、'('和')'要成对的出现。

我们可以这么考虑：{、(、[一定是在]、)、}前面出现，那么我们遇到左面的分隔符就可以把它们压入栈中，碰到右面的分隔符就弹出栈顶元素并进行比较，如果不相等或者栈已经为空，那么就不匹配。当然，循环完了也要检查栈是否为空，如果不为空，那么说明左面的分隔符多了，也是不匹配的。

代码如下：

```
package com.algorithm.stack;


/**
 * 分隔符匹配问题
 * @author yy
 *
 */
```

```java
public class BracketChecker {

    private String input;

    public BracketChecker(String in) {
        input = in;
    }

    //check 分隔符是否匹配
    public void check() {
        int stackSize = input.length();
        StackX theStack = new StackX(stackSize);

        for (int i = 0; i < input.length(); i++) {
            char ch = input.charAt(i);
            switch (ch) {
            case '{':
            case '[':
            case '(':
                theStack.push(ch);
                break;
            case '}':
            case ']':
            case ')':
                if(!theStack.isEmpty()) {
                    char chx = theStack.pop();
                    if((ch=='}' && chx!='{' )||
                       (ch==')' && chx!='(')||
                       (ch==']' && chx!='[')) {
                        System.out.println("error char at "+i);
                    }
                }else {
                    System.out.println("error char at "+i);
                }
                break;
            default:
                break;
            }
        }

        if(!theStack.isEmpty()) {
            System.out.println("error missing right delimiter");
        }
```

```
    }


    public static void main(String[] args) {
        new BracketChecker("(sah[fdjs[([])])").check();
    }
}
```

队列

原理

队列是一种类似栈的数据结构，但是与栈不同的是，队列中第一个插入的元素会被第一个移出。

几种常见操作：

1.Insert　向队尾插入元素

2.Remove　　移出队首元素

3.Peek　查看队首元素

队列也可以基于数组实现或者链表实现，我们这里同样使用数组实现。

代码

```
package com.algorithm.queue;


/**
 * 队列的实现
 * @author yy
 *
 */
public class Queue {

    private int maxSize;  //队列大小 构造函数中传入
    private long[] queArray;  //保存队列数据的数组
    private int front;//队首指针
    private int rear;  //队尾指针
    private int nItems;   //表示当前元素的个数
```

```
public Queue(int s) {
    maxSize = s;
    queArray = new long[maxSize];
    front = 0;
    rear = -1;
    nItems = 0;
}


//插入一个元素
public void insert(long j) throws Exception {
    if(isFull()) {
        throw new Exception("队列元素已满");
    }
    if(rear == maxSize-1)
        rear = -1;
    queArray[++rear] = j;
    nItems++;
}

//删除一个元素
public long remove() {
    long temp = queArray[front++];
    if(front == maxSize)
        front = 0;
    nItems--;
    return temp;
}


//判断是否为空
public boolean isEmpty() {
    return nItems == 0;
}


//判断队列是否已满
public boolean isFull() {
    return nItems == maxSize;
}

//队列元素个数
public int size() {
    return nItems;
```

```
    }

    //查看队首元素
    public long peekFront() {
        return queArray[front];
    }


    public static void main(String[] args) throws Exception {
        Queue queue = new Queue(4);
        queue.insert(2);
        queue.insert(3);
        queue.insert(2);
        queue.insert(1);
        System.out.println(queue.remove());
        System.out.println(queue.remove());
        System.out.println(queue.remove());
        System.out.println(queue.remove());
    }
}
```

查找

普通数组的查找

假设我们现在需要在一堆数据里查找我们需要的数据，此时数据本身没有什么特征，我们要查找的数据可能在这一堆数据的任何位置，所以我们只有通过从头到尾遍历的方式才能找到我们需要查找的数据。

效率:O(N) 这里假定这一堆数据的长度是 N

有序数组的查找

假设数据排列的是很整齐的，即是排好序的，此时我们可以通过二分查找来会进行查找，不断折半，缩小查找范围。

假定在排序好的数据中查找某个值，数组长度为 n。按照如下方式迭代(假设查找的值是 x，数组是 a)：

1. start=0, end=n-1, middle=(end+start)/2

2. 将 x 和 a[middle] 比较，若是 x<a[middle], 那么 end=middle-1，middle=(start+end)/2. 若是大于，那么 start=middle+1，middle=(start+end/2). 若是等于，那么直接退出。

3. 判断 start 是否大于 end，若是直接退出，否则执行 2

效率:O(lg(N)) 这里假定这一堆数据的长度是 N

二分查找代码：

```
/**
 * 二分查找 返回索引 -1代表没有查找到
 */
public int binary_search(int[] arr, int length, int value) {
    if(arr == null || arr.length == 0)
        return -1;
    int start = 0;
    int end = length-1;

    while(start <= end) {
        int middle = (start+end)/2;
        if(arr[middle] == value) {
            return middle;
        }else if(arr[middle] > value) {
            end = middle-1;
        }else {
            start = middle+1;
        }
    }


    return -1;
}
```

哈希

哈希表

哈希表(Hash Table 也叫散列表),是根据关键码值(key value)来进行访问的数据结构。也就是说，他通过把关键码值映射到表中的一个位置来访问记录，来加快查找的速度，这个映射叫做散列函数，这个存放记录的数组叫做散列表。

哈希表特点

1.哈希表可以快速的插入、查找、删除（O(1)时间）

2.哈希表基于数组实现，一但确定了大小，不易维护

3.没有一种简易的方式实现遍历，因此若是需要这个功能，那么需要选择其他的数据结构。

哈希化过程

将关键字转换成为数组的下标，这一步是通过哈希函数来实现。如何正确的选择哈希函数，是哈希表构建过程中最关键的一步。下面我们通过几个例子看一下如何选择正确的哈希函数。

哈希化例子

雇员

假设现在一个公司有 1000 个雇员，我们需要写一个程序存储这个公司的雇员信息，计算机部门的领导要求尽可能快的存取每个雇员的信息。而且，每个雇员都有一个自己的编号，1 到 1000，这个雇员的号码就是存取的关键字，那么我们如何建立哈希函数把关键字转换为索引呢？

这个例子只能说十分的巧合，我们可以直接把关键字当做索引，所以此时的哈希函数我们可以看做是 y=x。(y 代表索引，x 代表关键字)。

但我们应该清楚的是，数据不可能总是这么有序。

字典

随便给定一个单词，我们如何得到它的数组索引呢？单词和数组索引之间有什么关系呢？我们如何选择哈希函数？

假定数据集现在是 5w。

字典中我们假设只有小写字母和空格，字母一共 26 个，加上空格是 27 个，那么我们可以使用 1-26 来代表小写字母 a-z，0 来代表空格。接下来的就是编码的问题，如何编码表示一个单词。

这里给出两种思路。

第一种：相加。如果每个单词 10 位，那么最小的和就是 0*10=0，最大是 26*10=260，那么存储单元一共有 260 个，用 260 个单元来存储 5000 多个单词，这显然不合适。因为没有把数据分开，会有太多数据集中在一个单元，这样的话查找效率还是很慢。

第二种：使用幂乘的方式，使得每个单词都可以占数组中一个单元，假设单词 cats，c 是 3，a 是 1，t 是 20，s 是 19，因为用 27 个数字来表示所有的字符，那么我们模拟十进制

的幂乘，cats 就可以表示为 3*27^3+1*27^2+20*27^1+19*27^0，得到 60337。这样的话确实可以为每个单词创建一个独一无二的整数，但是整数的范围会非常大，会溢出。

比较一下两种方案，第一种产生的范围太小，第二种产生的太多。那么怎么办呢？

答案是取模运算，50000 个单词的话，我们就用 100000 个存储单元来进行存储，所以只需要把第二种方式产生的整数%100000 即可。

哈希冲突

在将关键字转换成为数组下标的时候，有可能转成的数组下标上已经有值了，这便是哈希冲突。

解决方案

开放地址法

在开放地址法中，如果数据不能放在由哈希函数得到的数组下标所指的单元时，就要寻找数组的其他位置。有三种寻找位置的方法，下面将一一介绍。

线性探测

线性探测就是如果当前的位置冲突，它会按照数组的下标一步步的查找空白的单元。

缺点：当哈希表越来越满时，聚集会变得越来越严重，这会导致产生非常长的探测长度。意味着存储序列的最后单元会非常的耗时。

数组填的越慢，聚集越可能发生，数组有一半数据时，这通常不是什么问题，三分之二满时，情况也不会太快，但是超过这个界限，性能下降会非常严重。因此，设计哈希表的关键是确保它不会超过整个数组容量的一半。

代码：

数据项类

```
package com.algorithm.hash;


/**
 * 数据项类 这里存储的是int
 * @author yy
 *
 */
public class DataItem {

    private int iData;
```

```
    public DataItem(int i) {
        iData = i;
    }

    public int getKey() {
        return iData;
    }

}
```

哈希类

```
package com.algorithm.hash;

/**
 * 完成哈希存储和查找功能
 * 开放地址法之线性探测
 * @author yy
 *
 */
public class HashTable {

    private DataItem[] hashArray;
    private int arraySize;
    private DataItem nonItem;

    public HashTable(int size) {
        arraySize = size;
        hashArray = new DataItem[arraySize];
        nonItem = new DataItem(-1);
    }

    public void displayTable() {
        System.out.println("table: ");
        for (int i = 0; i < hashArray.length; i++) {
            if(hashArray[i] != null)
                System.out.println(hashArray[i].getKey()+" ");
            else
                System.out.println("** ");
        }

    }

    //hash映射函数
    public int hashFun(int key) {
```

```
    return key % arraySize;
}


//哈希插入
public void insert(DataItem item) {
    int key = item.getKey();
    int hashVal = hashFun(key);

    while(hashArray[hashVal] != null &&
            hashArray[hashVal].getKey() != -1) {
        ++hashVal;
        hashVal%=arraySize;
    }
    hashArray[hashVal] = item;
}


//哈希删除
public DataItem delete(int key) {
    int hashVal = hashFun(key);

    while(hashArray[hashVal] != null) {

        if(hashArray[hashVal].getKey() == key) {
            DataItem temp = hashArray[hashVal];
            hashArray[hashVal] = nonItem;
            return temp;
        }

        ++hashVal;
        hashVal %= arraySize;

    }

    return null;

}


//哈希查找
public DataItem find(int key) {
    int hashVal = hashFun(key);
```

```
        while(hashArray[hashVal] != null) {
            if(hashArray[hashVal].getKey() == key) {
                return hashArray[hashVal];
            }
            ++hashVal;
            hashVal%=arraySize;
        }
        return null;
    }

}
```

二分探测

二分探测是防止聚集产生的一种尝试，思想是探测较远的单元，步骤是步数的平方，1、4、9…以此类推。二分探测会产生二次聚集的问题，一般不使用，因为有更好的解决冲突的方案。

再哈希

为了消除原始聚集和二次聚集，可以使用另外的方法，再哈希法，二次聚集产生的原因就是因为二次探测的算法产生的探测序列步长总是固定的，1、4、9、16 以此类推。

现在需要的是一种依赖关键字的探测序列，而不是每个关键字都一样，那么不同的关键字即使映射到同一单元上，也会产生不同的探测步长。

方法是把关键字用不同的哈希函数再进行一次哈希化，用这个结果进行探测。

第二个哈希函数必须有如下特点：

1. 和第一个不一样
2. 不能输出 0，0 的话原地踏步 将进入死循环

专家发现如下的函数效果十分好：

Stepsize=constant-(key%constant)

Constant 是质数 并且小于数组的容量

代码：

```
package com.algorithm.hash;

/**
 * 开放地址之再哈希解决哈希冲突
 * 在开放地址法解决哈希冲突时  再哈希法是最常使用的
 * @author yy
 *
 */
public class HashDouble {

    private DataItem[] hashArray;
```

```
private int arraySize;
private DataItem nonItem;

public HashDouble(int size) {
    arraySize = size;
    hashArray = new DataItem[arraySize];
    nonItem = new DataItem(-1);
}

public void displayTable() {
    System.out.println("table: ");
    for (int i = 0; i < hashArray.length; i++) {
        if(hashArray[i] != null)
            System.out.println(hashArray[i].getKey()+" ");
        else
            System.out.println("** ");
    }

}

//hash映射 key--index
public int hashFun1(int key) {
    return key % arraySize;
}

//步数映射
public int hashFun2(int key) {
    return 5-key%5;
}

//哈希插入
public void insert(int key, DataItem item) {
    int hashVal = hashFun1(key);
    int stepSize = hashFun2(key);

    while(hashArray[hashVal] != null &&
            hashArray[hashVal].getKey() != -1) {
        hashVal += stepSize;
        hashVal%=arraySize;
    }
    hashArray[hashVal] = item;
}
```

```
    //哈希删除
    public DataItem delete(int key) {
        int hashVal = hashFun1(key);
        int stepSize = hashFun2(key);
        while(hashArray[hashVal] != null) {
            if(hashArray[hashVal].getKey() == key) {
                DataItem temp = hashArray[hashVal];
                hashArray[hashVal] = nonItem;
                return temp;
            }
            hashVal += stepSize;
            hashVal%=arraySize;
        }
        return null;
    }

    //哈希查找
    public DataItem find(int key) {
        int hashVal = hashFun1(key);
        int stepSize = hashFun2(key);

        while(hashArray[hashVal] != null) {
            if(hashArray[hashVal].getKey() == key) {
                return hashArray[hashVal];
            }
            hashVal+=stepSize;
            hashVal%=arraySize;
        }
        return null;

    }
}
```

拉链法

拉链法就是在哈希表的每个单元中设置链表，数据映射到某个单元上之后直接插入到链表中即可。不用再寻找空位。

代码：

```
package com.algorithm.hash;

import com.algorithm.orderlink.Link;
import com.algorithm.orderlink.OrderList;
```

```java
//拉链法解决哈希冲突
public class HashChain {

    private OrderList[] hashArray;
    private int arraySize;

    public HashChain(int size) {
        arraySize = size;
        hashArray = new OrderList[size];
        for (int i = 0; i < hashArray.length; i++) {
            hashArray[i] = new OrderList();
        }
    }

    //打印哈希表
    public void displayTable() {
        for (int i = 0; i < arraySize; i++) {
            System.out.print(i+". ");
            hashArray[i].display();
        }
    }

    public int hashFun(int key) {
        return key % arraySize;
    }


    //哈希插入
    public void insert(int key) {
        int hashVal = hashFun(key);
        hashArray[hashVal].insert(key);
    }

    //哈希删除
    public void delete(int key) {
        int hashVal = hashFun(key);
        hashArray[hashVal].delete(key);
    }

    //哈希查找
    public Link find(int key) {
        int hashVal = hashFun(key);
```

```
        Link theLink = hashArray[hashVal].find(key);
        return theLink;
    }

}
```

题目：出现次数超过一半的数

题目描述：数组中有一个数出现的次数超过了数组长度的一半，找出这个数。(数组是无序的)。

思路一：先排序。我们如果选择最快的排序算法，时间复杂度是 O(log(N))。排序完了之后，一般思路是遍历，然后统计次数，但其实没有必要这么做。因为如果某个数在数组中出现的次数超过了一半，那么在已经排好序的数组索引的 n/2 处就一定是要找的这个数。那么总的时间复杂度其实就是排序的时间复杂度，O(log(N))。

代码比较简单，不再展示，就是一个排序，然后输出。

思路二：散列表，以空间换取时间。把数组中的数当做散列表中的键，值为该键出现的次数。那么，利用散列表完成统计之后，直接遍历整个散列表，直接输出即可。

遍历一次需要的时间是 O(N)，而构造散列表需要 O(N)的空间开销，而且还要设计散列函数。有没有更好的方法呢？

思路三：每次删除两个不同的数。我们每次删除数组中两个不同的数，不管是不是我们要查找的那个数，那么在剩下的数中，我们要查找的那个数仍然会超过剩余总数的一半。通过不断重复这个过程，最终会找到那个出现次数超过一半的数。这样一算，时间复杂度是 O(N)，而空间复杂度是 O(1)。

那么我们如何能够最高效的删除两个数，完成遍历呢？

方法就是遍历数组的时候保存两个值，一个是 temp，用来保存数组中遍历到的某个数，另一个是 count，用来保存当前数的出现次数，初始化为 1。当遍历到下一个数的时候，可能有如下情况：

1.如果下一个数与之前的 temp 中保存的数相同 那么 count++

2.如果下一个数与之前的 temp 中保存的数不同 那么 count—

3.每当 count=0 时，用 temp 保存下一个数，并把 count 设置为 1。

4.直到遍历完所有的数组。

代码：

```
    /**
     * 通过不断地删除两个数 找到出现次数超过一半的数
     * 假设输入总是有效的
     * @throws Exception
     */
    private static int findOneNumberByDel(int[] a, int len) {
        int temp = 0;
        int count = 0; //用来计数
        for (int i = 0; i < a.length; i++) {
```

```
            if(count == 0) {   //等于0代表前面的所有数据恰好已经抵消了
                temp = a[i];   //需要重新选择一个元素
                count++;
            }else {
                if(temp == a[i]) {//碰到一样的
                    count++;
                }else {//碰到不一样的  需要抵消
                    count--;
                }
            }
        }
        return temp;   //返回结果
    }


    public static void main(String[] args) {
        int[] a = {0,1,2,1,1};
        System.out.println(findOneNumberByDel(a, a.length));
    }
```

字符串查找

问题引入

现在有这么一个问题，有一个文本串 S 和一个模式串 P，要查找 P 在 S 中的位置应该怎么做？

蛮力匹配方法

思路：假设现在 S 匹配到了 i 位置，P 匹配到了 j 位置，如果是蛮力搜索的思路，则有：

1. 如果当前字符匹配成功，那么 i++，j++，继续匹配下一个字符
2. 匹配失败的话，令 i=i-(j-1)，j=0，相当于每次匹配失败时，i 回溯，j 置为 0。

代码：

```
    /**
     * 查找P在S中出现的位置
     * 暴力匹配 不匹配返回-1
     * @param s
     * @param p
     */
```

```
private static int StringMatch(String s, String p) {
    int sLen = s.length();
    int pLen = p.length();
    int i = 0, j = 0;
    while(i < sLen && j < pLen) {
        if(s.charAt(i) == p.charAt(j)) {
            i++;
            j++;
        }else {
            i = i-j+1;
            j = 0;
        }
    }
    if(j == pLen) {
        return i-j;
    }else {
        return -1;
    }
}
```

KMP 算法

刚才介绍的蛮力匹配算法，效率是十分低的，每次 j 置 0，而 i 回溯。那么有没有一种算法，让 i 不往回退，而只需要移动 j 即可呢？答案是肯定的，这种算法就是 KMP。

KMP 算法中有一个很关键的数组，这个数组就是 next 数组，它是我们通过计算求得的。对于 next 数组，我们先不介绍它的具体求法，我们先来看一下 KMP 算法的运行过程。

假设现在 S 匹配到 i 位置，P 匹配到 j 位置：

1.如果 j=-1 或者当前字符匹配成功 都令 i++，j++。

2.如果 j!=-1 并且当前字符匹配失败，则令 i 不变，j=next[j]。

我们可以看出，匹配失败时，P 相对于文本串 S 向右移动了 j=next[j]位。

KMP 的过程还是比较简单，现在的关键就是 next 数组究竟是什么？

Next 数组就是当前字符之前的字符串中，有多大长度的相同前缀和后缀。

最大前缀后缀长度

看懂了下面的表格，最大前缀后缀长度就知道怎么求了。

模式串的各个子串	前缀	后缀	公共元素的最大长度

A	空	空	0
AB	A	B	0
ABC	A,AB	C,BC	0
ABCD	A,AB,ABC	D,CD,BCD	0
ABCDA	A,AB,ABC,ABCD	A,DA,CDA,BCDA	1
ABCDAB	A,AB,ABC,ABCD,ABCDA	B,AB,DAB,CDAB,BCDAB	2
ABCDABD	A,AB,ABC,ABCD,ABCDA,ABCDAB	D,BD,ABD,DABD,CDABD,BCDABD	0

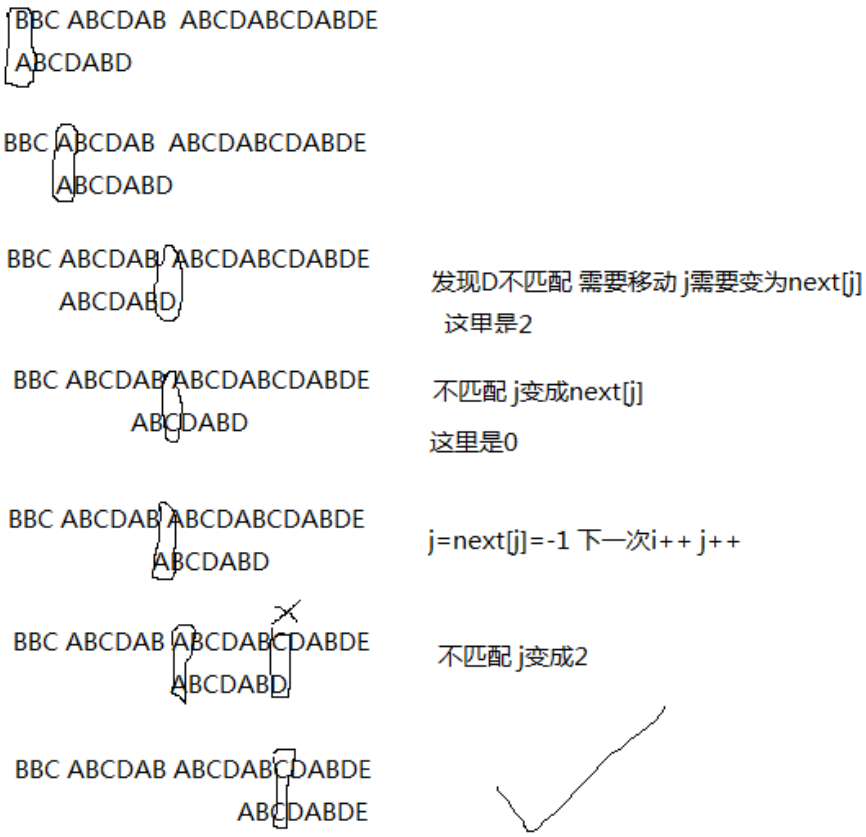
最大前后缀求 next 数组

知道了最大长度表，如何求解 next 数组呢？就是初值赋 1，整体右移。看下面的表格就明白了：

模式串	A	B	C	D	A	B	D
最大长度	0	0	0	0	1	2	0
Next 数组	-1	0	0	0	0	1	2

基于 next 数组的匹配过程

下面我们一起来看一个匹配过程。



代码递推求解 next 数组

现在最后一个问题就是如何通过代码来求解 next 数组？可以使用递推来解决这个问题。已知 next[0,…,j] 如何求出 next[j+1]？

1.若是 P[k]=P[j] 那么 next[j+1]=next[j]+1=k+1(假设 next[j]=k)

2.若是不等于 如果此时 P[next[k]]==P[j] next[j+1]=next[k]+1,否则递归 k=next[k];

关于这两个过程的解释：第一个过程还是比较容易理解的，就是说如果 next[j]=k，那么 P[0…1…k-1] 等于 P[j-k…j-k+1…j-1]，此时若是 P[k]==P[j]，那么最大前缀后缀长度自然就加 1，变成 k+1。第二个过程就是说如果不相等，就要寻找长度更小的相同前缀和后缀。这归结于 next 数组的定义，为了寻找长度相同的前缀和后缀，我们拿前缀 POPk-1Pk 去跟 Pj-kPj-1Pj，如果 Pj 和 Pk 匹配失败，下一步就是使用 P[next[k]]去跟 P[j]继续匹配，如果还是不匹配，就只能递归 k=next[k]继续寻找更小的前缀后缀长度。

Next 数组优化

考虑这种情况，如果 P[j]!=S[i]，此时 j=next[j]，所以下次必然是 P[next[j]]和 S[i]比较，如果 P[j]-P[next[j]]，必然会导致匹配失败，因为 P[j]和 S[i]已经匹配失败，还用跟 P[j]等同的 P[next[j]]去和 S[i]匹配，这一定会失败。

所以出现了就需要再次递归，令 next[j] = next[next[j]];

KMP 代码

```
package com.algorithm.kmp;

/**
 * KMP算法的实现
 *
 */
public class KMP {

    /**
     * 求 next数组
     * @param p
     * @param next
     */
    private static void getNext(char[] p, int[] next) {
        int len = p.length;
        next[0]  = -1;
        int k = -1;
        int j = 0;
        while(j < len-1) {
            if(k == -1 || p[j] == p[k]) {
                ++j;
                ++k;
                if(p[j] != p[k]) {
                    next[j] = k;
                }else {
                    next[j] = next[k];
                }
            }else {
                k = next[k];
            }
        }
    }


    public static int KmpSearch(char[] s, char[] p) {
        int i = 0;
        int j = 0;
        int[] next = new int[p.length];
        getNext(p, next);
```

```
        int sLen = s.length;
        int pLen = p.length;
        while(i<sLen && j<pLen) {
            if(j==-1 || s[i]==p[j]) {
                i++;
                j++;
            }else {
                j = next[j];
            }
        }

        if(j == pLen) {
            return i-j;
        }else {
            return -1;
        }
    }


    public static void main(String[] args) {
        int index =KmpSearch("BBC ABCDAB ABCDABCDABDE".toCharArray(),
"ABCDABD".toCharArray());
        System.out.println(index);
    }
}
```

树

树的定义

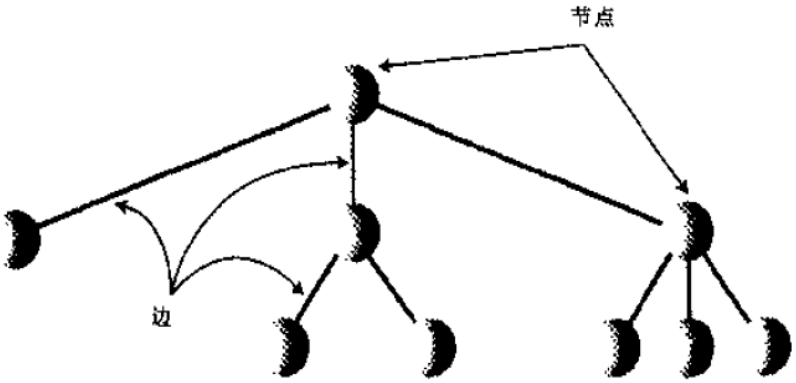
一棵树（tree）是由 n（n>0）个元素组成的有限集合，其中：

（1）每个元素称为结点（node）；

（2）有一个特定的结点，称为根结点或根（root）；

（3）除根结点外，其余结点被分成 m（m>=0）个互不相交的有限集合，而每个子集又都是一棵树（称为原树的子树）

如下图，就是一棵树：



常见术语

根

树顶端的节点就是根，一棵树只有一个根。

父节点

每个节点都恰好有一条边向上连接到另一个节点，上面的这个节点就称为下面这个节点的父节点。

子节点

每个节点都可能有一条边或者多条边向下连接到其他节点，下面的这些节点就叫做该节点额子节点。

叶节点

没有子节点的节点叫做叶子节点。

访问

当程序控制流程到达某个节点时，就称为访问这个节点。

遍历

遍历就是按照某种特定的顺序遍历树中的所有节点。

层

一个节点的层是从根开始到这个节点有多少“代”，假设根是第 0 代，它的子节点就是第一代，孙节点就是第二代。

路径

对于一棵子树中的任意两个不同的结点，如果从一个结点出发，按层次自上而下沿着一个个树枝能到达另一结点，称它们之间存在着一条路径。可用路径所经过的结点序列表示路径，路径的长度等于路径上的结点个数减 1.

森林

指若干棵互不相交的树的集合

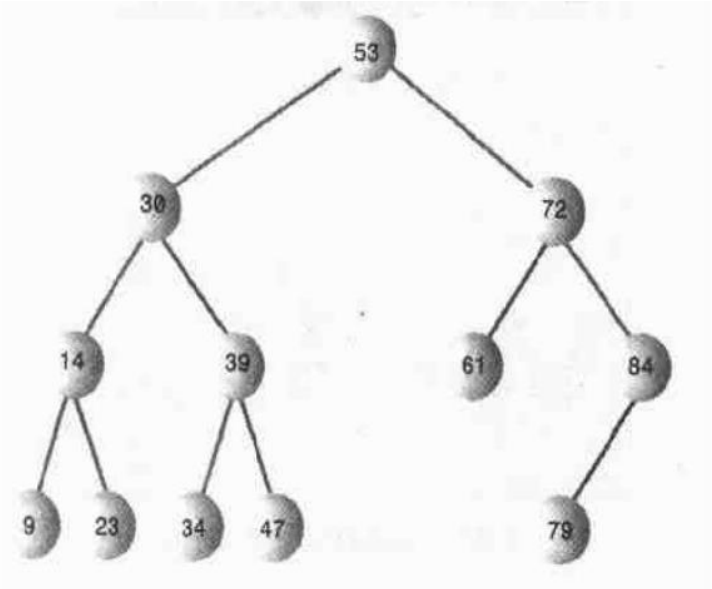
二叉树

二叉树定义

如果树中每个节点最多有两个子节点，这种树就叫做二叉树。

为什么要有二叉树的出现？因为数组查找快，但是插入、删除太慢，而链表插入、删除快，但是查找慢，二叉树就是对数组和链表优点的一种均衡。

下面我们要学习的二叉树学术上被称为二叉搜索树。如下图，就是一棵二叉树：



Java 实现二叉树常见操作

节点的表示

二叉树是由若干个节点组成的，首先我们需要创建一个节点类。那么节点类都需要哪些属性呢？

需要至少三个。首先是数据项，代表这个节点表示的数据，可以是一个属性或者多个，视具体的情况而定。然后就是子节点的表示，因为二叉树每个节点最多有两个子节点，所以我们这里仅仅需要两个引用，一个指向它的左孩子，一个指向它的右孩子即可。

节点 Node 类代码如下：(这里的数据项假设就是一个 int 类型的变量)

```
/**
 * 节点类
 * @author yy
 *
 */
class Node {
    public int data;  //节点数据项
    public Node leftChild;    //指向左孩子的引用
    public Node rightChild;   //指向右孩子的引用
}
```

二叉树的表示

节点我们使用 Node 类来进行表示，要想表示一颗二叉树，我们仅仅需要一个指向二叉树根节点的引用即可。为什么呢？

因为二叉树是一种层次的结构，每个节点都至多有两个子节点，而 Node 类包含了对两个子节点的引用，所以只要有了对根节点的引用，那么就相当于有了对二叉树所有节点的引用。

无任何操作的二叉树代码：

```
/**
 * Java实现二叉树的常见操作
 * @author yy
 *
 */
public class BinaryTree {
    private Node root;//根节点
}
```

已经有了一棵二叉树，我们现在来学习一下它的几种操作。

插入新节点

二叉树有了，那么我们必须提供一种方法使得它能够插入新的节点。

要插入节点，我们必须先找到插入的位置，这很像找一个不存在的节点的过程，从根开始查找一个节点，它将是新节点的父节点，父节点找到之后，新的节点就可以插入父节点的左面或者右面，这取决于它的值比父节点的值大还是小。

插入代码如下：

```
//插入操作
    public void insert(int data) {
        Node newNode = new Node();
        newNode.data = data;
        if(root == null) {
            root = newNode;
        }else {
            Node current = root;
            Node parent;
            while(true) {
                parent = current;
                if(data < current.data) {
                    current = current.leftChild;
                    if(current == null) {
```

```
                    parent.leftChild = newNode;
                    return;
                }
            }else {
                current = current.rightChild;
                if(current == null) {
                    parent.rightChild = newNode;
                    return;
                }
            }
        }
    }

}
```

查找节点

在二叉搜索树中，我们保证左孩子总是比自己的值小，右孩子总是比自己的值大。那么要查找某个值，我们就可以利用这一特性，从根节点开始，如果当前值大于要查找的值，那就去左子树查找，否则去右子树查找。

代码如下：

```
//查找操作
    public Node find(int key) {

        Node current = root;
        if(current == null) {
            return null;
        }
        while(current.data != key) {
            if(key < current.data) {
                current = current.leftChild;
            }else {
                current = current.rightChild;
            }
            if(current == null) {
                return null;
            }
        }

        return current;
    }
```

节点的删除

节点的删除是比较麻烦的，因为需要考虑的情况比较多。要删除的节点可能有三种情况：

1. 是叶子节点，没有子节点
2. 有一个子节点
3. 有两个子节点

每种情况处理的方式都是不同的。下面我们逐一讨论。

无子节点

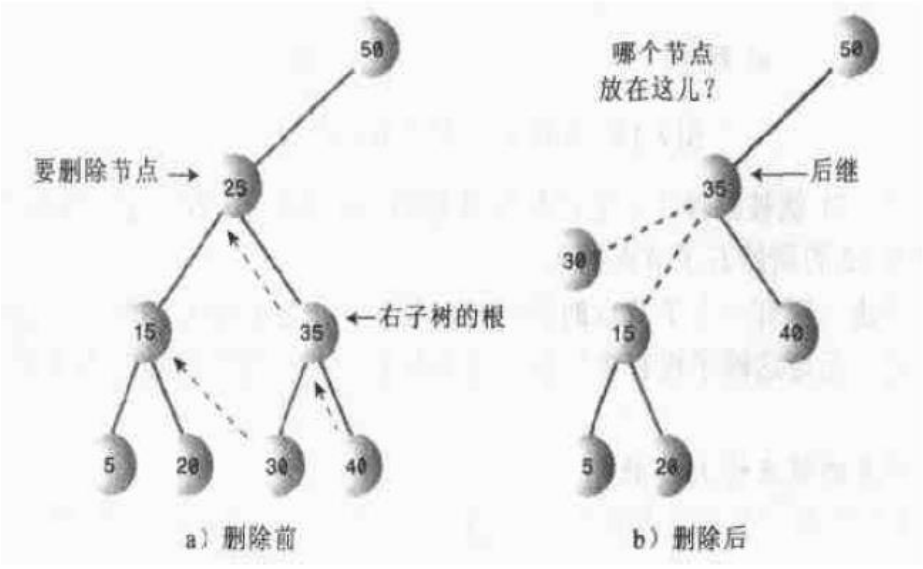
这种情况是最简单的，只需要把该节点对应的父节点的字段值设置为 null 即可，GC 会完成剩下的回收过程。要删除的节点可能在一段时间内仍然存在，但是它已经不是树的一部分了。

有一个子节点

这种情况也不是很难，要删除的节点有两个连接，连向父节点和它唯一的子节点，仅需要改变父节点的引用，指向要删除节点的子节点即可。

有两个子节点

这种情况是最复杂的。看一个例子：



假设我们需要删除的节点是 25，并且用根是 35 的右子树取代它，那么 35 的左子节点是谁呢？是原来的左子结点 30 还是删除节点的左子结点 15 亦或是其他的节点，我们必须寻找一种有效的方式来做出判断。

在二叉搜索树中，二叉搜索树的节点是按照升序的关键字值排列的，对于一个节点来说，比它的关键字值次高的是它的后继节点。就是比它大的值中最小的那一个，例如上面 25 的

后继节点就是 30。我们删除一个节点时，就可以使用这个节点的后继节点来代替该节点。

Java 实现二叉树删除的话，我们需要两个方法，一个是删除，另外一个就是求后继节点。

代码

求后继节点：

```
//查找某个节点的中序后继节点并完成一些处理
    private Node getSuccessor(Node delNode) {
        Node successorParent = delNode;  //最后保存的是后继节点的父节点
        Node successor = delNode; //最后保存的是后继节点
        Node current = delNode.rightChild;   //current用来向下查找
        while(current!=null) {
            successorParent = successor;
            successor = current;
            current = current.leftChild;
        }

        if(successor != delNode.rightChild) {
            successorParent.leftChild = successor.rightChild;  //后继节点
的父节点的左子孩子赋值为后继节点的右孩子
            successor.rightChild = delNode.rightChild;  //后继节点的右孩子
赋值为删除节点的右孩子
        }
        return successor;
    }
```

删除：

```
    //删除操作
    public boolean delete(int key) {
        Node current = root;
        Node parent = root;
        boolean isLeftChild = true;
        while(current.data != key) {
            parent = current;
            if(key < current.data) {
                isLeftChild = true;
                current = current.leftChild;
            }else {
                isLeftChild = false;
                current = current.rightChild;
            }
            if(current == null) return false;
```

```
}//上述步骤是找到要删除的节点
//下面将分情况来进行删除操作
//1.无孩子
if(current.leftChild == null && current.rightChild == null) {
    if(root == current) {
        root = null;
    }else if(isLeftChild) {
        parent.leftChild = null;
    }else {
        parent.rightChild = null;
    }
}else if(current.rightChild == null) {
    //2.有左孩子
    if(current == root) {
        root = current.leftChild;
    }else if(isLeftChild) {
        parent.leftChild = current.leftChild;
    }else {
        parent.rightChild = current.leftChild;
    }
}else if(current.leftChild == null) {
    //3.有右孩子
    if(current == root) {
        root = current.rightChild;
    }else if(isLeftChild) {
        parent.leftChild = current.rightChild;
    }else {
        parent.rightChild = current.rightChild;
    }
}else {
    //4.有两个孩子
    Node successor = getSuccessor(current);

    if(current == root) {
        root = successor;
    }else if(isLeftChild) {
        parent.leftChild = successor;
    }else {
        parent.rightChild = successor;
    }

    successor.leftChild = current.leftChild;
}
```

```
        return true;
    }
```

遍历

二叉树的遍历无非三种方式：中序、前序、后序。
前序就是先访问当前节点，再访问左子树，最后访问右子树。
中序就是先访问左子树，再访问当前节点，最后访问右子树。
后序就是先访问左子树，再访问右子树，最后访问当前节点。

代码比较容易，就是递归调用，不再展示。

求二叉树的宽度

什么是二叉树的宽度？二叉树的宽度定义为具有最多结点数的层中包含的结点数。
那么我们怎么来求解呢？
我们可以考虑使用队列来实现，队列是一种先进先出的数据结构，具体步骤如下：

1. 一开始让根节点入队，此时队列长度为 1，最大宽度也就是 1
2. 让队列中的元素一一出队，出队时看一下他们是否有子节点，如果有的话入队
3. 查看队列长度，跟最大宽度比较，重复第二个过程，直到队列长度为 0。

代码如下：

```
/**
 * 求二叉树宽度
 * @return
 */
public int getMaxWidth() {
    if(root == null) return 0;
    Queue<Node> queue = new ArrayDeque<Node>();
    int maxWidth = 1; //最大宽度
    queue.add(root);  //入队

    while(true) {
        int len = queue.size();
        if(len == 0) break;
        while(len > 0) {  //当前层还有节点
            Node n = queue.poll();
            len--;
            if(n.leftChild != null)
                queue.add(n.leftChild);;
            if(n.rightChild != null)
```

```
                queue.add(n.rightChild);
            }
            maxWidth = Math.max(maxWidth, queue.size());
        }

        return maxWidth;
    }
```

求二叉树最大距离

二叉树最大距离就是相距最远的两个叶子节点之间的距离。

思路：二叉树最大距离其实就是左子树最长距离+右子树最长距离，所以可以通过递归来做，递归的求左子树最大距离和右子树最大距离，然后判断，更新结果。

代码：

```
package com.algorithm.tree;


/**
 * 求一颗二叉树中的最大距离
 * @author yy
 *
 */
class TreeNode {
    TreeNode pLeft;       //左孩子
    TreeNode pRight;  //右孩子
    int nMaxLeft;  //左子树中的最长距离
    int nMaxRight; //右子树中的最长距离
    char chValue;  //该节点的值
}

public class Tree {

    public static int nMaxLen;

    //寻找树中的最长距离
    public static void findMaxLen(TreeNode pRoot) {
        //如果遍历到叶子节点 返回
        if(pRoot == null) return;
        //如果左子树空 那么该节点的左边最长距离为0
        if(pRoot.pLeft == null) {
            pRoot.nMaxLeft = 0;
        }
```

```
        //如果右子树是空 那么该节点的右边最常距离为0
        if(pRoot.pRight == null) {
            pRoot.nMaxRight = 0;
        }

        //左子树不为空 递归寻找左子树的最长距离
        if(pRoot.pLeft != null) {
            findMaxLen(pRoot.pLeft);
        }


        //计算左子树最长节点距离
        if(pRoot.pLeft != null) {
            int nTempMax = 0;
            if(pRoot.pLeft.nMaxLeft > pRoot.pLeft.nMaxRight) {
                nTempMax = pRoot.pLeft.nMaxLeft;
            }else {
                nTempMax = pRoot.pLeft.nMaxRight;
            }
            pRoot.nMaxLeft = nTempMax+1;
        }
        //计算右子树
        if(pRoot.pRight != null) {
            findMaxLen(pRoot.pRight);
        }

        //计算you子树最长节点距离
        if(pRoot.pRight != null) {
            int nTempMax = 0;
            if(pRoot.pRight.nMaxLeft > pRoot.pRight.nMaxRight) {
                nTempMax = pRoot.pRight.nMaxLeft;
            }else {
                nTempMax = pRoot.pRight.nMaxRight;
            }
            pRoot.nMaxRight = nTempMax+1;
        }

        //返回最长距离
        if(pRoot.nMaxLeft+pRoot.nMaxRight > nMaxLen) {
            nMaxLen = pRoot.nMaxLeft + pRoot.nMaxRight;
        }
    }

}
```

哈弗曼树

字符编码

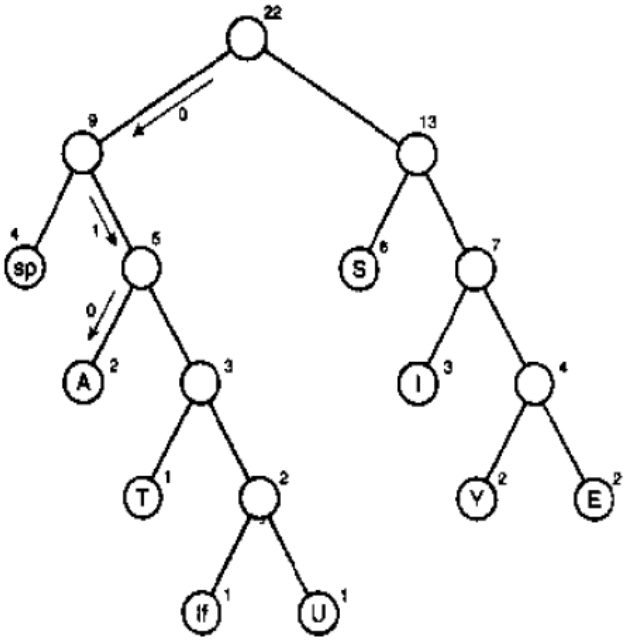
计算机里每个字符在没有压缩的文本中由一个字节或者两个字节表示，这些方案中，每个字符需要相同的位数。这种方式使得数据在网络中传输时效率不是很高，因为每个字符都用相同的位数，没有分别对待。

如何分别对待呢？

我们知道不同的字符使用的频率不一样，有的高，有的低，那么我们可以考虑使用较少的位数表示使用频率高的字符，使用较多的位数表示使用频率低的字符，这样的话数据在网络中的传输速度就会大大的加快。

哈弗曼编码

哈弗曼树就是为了解决这种数据编码问题而存在的，它也是一棵二叉树，看下图：



树中的每个节点表示一个字符，越是使用频率高的字符越在树的上方，使用频率低的字符则在树的下方。那么编码是如何进行的呢？往左记 0，往右记 1，比如 S，就是 10。

那么如何解码呢？比如 10，开始遇到 1，就向右走，然后是 0，就向左走，这样就找到了 S 字符。

构建哈弗曼树的过程

1. 为消息中的每个字符创建一个 Node 对象，每个节点有两个数据项，字符和字符出现的频率。
2. 为这些节点创建 Tree 对象，这些节点就是 Tree 的根。
3. 把这些树插入到一个优先级队列中，按照频率排序，频率最小的节点有最高的优先级。
4. 从优先级队列中删除两棵树，并把他们作为一个新节点的子节点。新节点的频率是两个子节点频率的和。
5. 把这个新的节点插入到优先级队列中。
6. 重复 4、5 步，当优先级队列中只剩下一棵树时，就是所创建的哈弗曼树。

代码

```
package com.algorithm.huffman;

import java.nio.charset.Charset;
import java.util.ArrayList;
import java.util.HashMap;
import java.util.LinkedList;
import java.util.List;
import java.util.Map;
import java.util.PriorityQueue;

/**
 * 哈弗曼树的实现
 *
 */
public class Main {

    /**
     * 树的实现 在构造哈弗曼树的过程中  不需要什么繁杂的删除 插入等操作
     * @author yy
     *
     */
    static class Tree {
        private Node root;

        public Node getRoot() {
            return root;
        }
```

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

    /**
     * 哈弗曼树节点的实现 实现Comparable接口  使得节点能够按照其所包含的字符串的使用频率来比较
     * @author yy
     *
     */
    static class Node implements Comparable<Node> {
        private String chars = "";
        private int frequence = 0;
        private Node parent;
        private Node leftNode;
        private Node rightNode;

        @Override
        public int compareTo(Node n) {
            return frequence - n.frequence;
        }

        public boolean isLeaf() {
            return chars.length() == 1;
        }

        public boolean isRoot() {
            return parent == null;
        }

        public boolean isLeftChild() {
            return parent != null && this == parent.leftNode;
        }

        public int getFrequence() {
            return frequence;
        }

        public void setFrequence(int frequence) {
            this.frequence = frequence;
        }

        public String getChars() {
```

```
        return chars;
    }

    public void setChars(String chars) {
        this.chars = chars;
    }

    public Node getParent() {
        return parent;
    }

    public void setParent(Node parent) {
        this.parent = parent;
    }

    public Node getLeftNode() {
        return leftNode;
    }

    public void setLeftNode(Node leftNode) {
        this.leftNode = leftNode;
    }

    public Node getRightNode() {
        return rightNode;
    }

    public void setRightNode(Node rightNode) {
        this.rightNode = rightNode;
    }
}

/**
 * 统计频率
 * 例如  a: 1 b:1 c:1 a:1   合并成为a:2 b:1 c:1
 * @param charArray
 * @return
 */
public static Map<Character, Integer> statistics(char[] charArray) {
    Map<Character, Integer> map = new HashMap<Character, Integer>();
    for (char c : charArray) {
        Character character = new Character(c);
        if (map.containsKey(character)) {
            map.put(character, map.get(character) + 1);
```

```
            } else {
                map.put(character, 1);
            }
        }

        return map;
    }

    /**
     * 构建哈弗曼树
     * 三步走
     * @param statistics
     * @param leafs
     * @return
     */
    private static Tree buildTree(Map<Character, Integer> statistics,
            List<Node> leafs) {
        Character[] keys = statistics.keySet().toArray(new
Character[0]);

        PriorityQueue<Node> priorityQueue = new PriorityQueue<Node>();
        for (Character character : keys) {
            Node node = new Node();
            node.chars = character.toString();
            node.frequence = statistics.get(character);
            priorityQueue.add(node);
            leafs.add(node);
        }

        int size = priorityQueue.size();
        for (int i = 1; i <= size - 1; i++) {
            Node node1 = priorityQueue.poll();
            Node node2 = priorityQueue.poll();

            Node sumNode = new Node();
            sumNode.chars = node1.chars + node2.chars;
            sumNode.frequence = node1.frequence + node2.frequence;

            sumNode.leftNode = node1;
            sumNode.rightNode = node2;

            node1.parent = sumNode;
            node2.parent = sumNode;
```

```
            priorityQueue.add(sumNode);
        }

        Tree tree = new Tree();
        tree.root = priorityQueue.poll();
        return tree;
    }

    /**
     * 给定一个字符串  得到它对应的二进制编码
     *
     * @param originalStr
     * @param statistics
     * @return
     */
    public static String encode(String originalStr,
            Map<Character, Integer> statistics) {
        if (originalStr == null || originalStr.equals("")) {
            return "";
        }

        char[] charArray = originalStr.toCharArray();
        List<Node> leafNodes = new ArrayList<Node>();
        buildTree(statistics,leafNodes);
        Map<Character, String> encodInfo = buildEncodingInfo(leafNodes);

        StringBuffer buffer = new StringBuffer();
        for (char c : charArray) {
            Character character = new Character(c);
            buffer.append(encodInfo.get(character));
        }

        return buffer.toString();
    }


    /**
     * 自下向顶遍历树 获得字符对应的二进制编码
     * @param leafNodes
     * @return
     */
    private static Map<Character, String> buildEncodingInfo(List<Node>
leafNodes) {
        Map<Character, String> codewords = new HashMap<Character,
```

```
String>();
        for (Node leafNode : leafNodes) {
            Character character = new
Character(leafNode.getChars().charAt(0));
            String codeword = "";
            Node currentNode = leafNode;

            do {
                if (currentNode.isLeftChild()) {
                    codeword = "0" + codeword;
                } else {
                    codeword = "1" + codeword;
                }

                currentNode = currentNode.parent;
            } while (currentNode.parent != null);

            codewords.put(character, codeword);
        }

        return codewords;
    }

    /**
     * 完成哈弗曼解码过程
     * @param binaryStr
     * @param statistics
     * @return
     */
    public static String decode(String binaryStr,
            Map<Character, Integer> statistics) {
        if (binaryStr == null || binaryStr.equals("")) {
            return "";
        }

        char[] binaryCharArray = binaryStr.toCharArray();
        LinkedList<Character> binaryList = new LinkedList<Character>();
        int size = binaryCharArray.length;
        for (int i = 0; i < size; i++) {
            binaryList.addLast(new Character(binaryCharArray[i]));
        }

        List<Node> leafNodes = new ArrayList<Node>();
        Tree tree = buildTree(statistics, leafNodes);
```

```
        StringBuffer buffer = new StringBuffer();

        while (binaryList.size() > 0) {
            Node node = tree.root;

            do {
                Character c = binaryList.removeFirst();
                if (c.charValue() == '0') {
                    node = node.leftNode;
                } else {
                    node = node.rightNode;
                }
            } while (!node.isLeaf());

            buffer.append(node.chars);
        }

        return buffer.toString();
    }

    public static void main(String[] args) {
        String oriStr = "Huffman codes compress data very effectively:
savings of 20% to 90% are typical, "
                + "depending on the characteristics of the data being
compressed. 中华崛起";
        Map<Character, Integer> statistics =
statistics(oriStr.toCharArray());
        String encodedBinariStr = encode(oriStr, statistics);
        String decodedStr = decode(encodedBinariStr, statistics);

        System.out.println("Original sstring: " + oriStr);
        System.out.println("Huffman encoed binary string: " +
encodedBinariStr);
        System.out.println("decoded string from binariy string: " +
decodedStr);
    }

}
```

Trie 树

Trie 树定义

Trie 树，即字典树，又称为单词查找树。常用于统计大量字符串等场景中，它的优点是最大限度的减少无谓的字符串比较，查询效率比较高。

Trie 树的核心思想是以空间换取时间，利用字符串的公共前缀来降低查询时间的开销。

它有如下三个特性：

1. 根节点不包含字符，除根节点外的其他节点都只包含一个字符
2. 从根节点到某一节点的路径上经过的字符串连接起来，即为该节点对应的字符串。
3. 每个节点的所有子节点包含的字符都不相同。

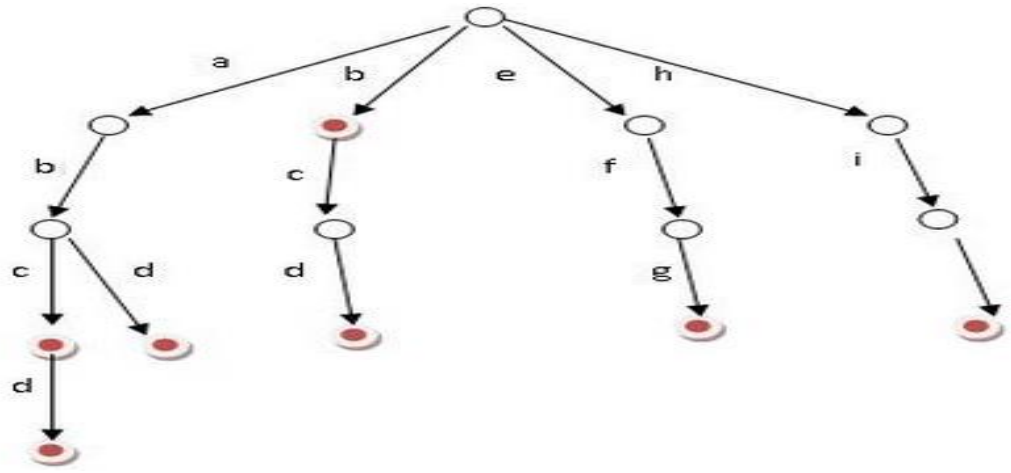
构建

先看一个问题，假如现在给定 10w 个长度不超过 10 个字母的单词，对于每一个单词，要判断它出没出现过，如果出现了，求第一次出现在哪个位置。如何解决？

最笨的方法，对每一个单词都去查找它前面的单词中是否有它，那么这个算法的时间复杂度就是 N^2，显然对于 10w 的数据量难以接受。

换个思路想：假设现在需要查询的单词是 abcd，那么在它前面的单词中，以 b、c、d 开头的单词就不用考虑了，而只要找以 a 开头的单词中是否有 abcd 就可以了。同样，在以 a 为开头的单词中，只要考虑以 b 作为第二个字母的，一次次缩小范围和针对性，这样一个树的模型就渐渐清晰了。

假设现在有 b、abc、abd、bcd、abcd、efg 和 hii 六个单词，就可以构成如下的字典树：



从根节点遍历到每一个节点的路径就是一个单词，如果某个节点被标记为红色，就表示这个单词存在，否则不存在。那么，对于一个单词，只要顺着它从根节点走到对应的节点，再看这个节点是否被标记为红色即可。这样的话，查询和插入操作的时间复杂度都是单词的长度。

Trie 树的搭建和查询

插入一个新的单词很简单，无非是逐一把每个单词的每个字母插入 Trie 树，插入前看看前缀是否存在，如果存在就共享，不存在就创建对应的边和节点。

查询也很简单，例如要查找 int，顺着路径 i->in->int 就找到了。

代码

为了能够清晰地理解 Trie 树的搭建，代码将一点点的展开。

Trie 树节点

Trie 树是由若干个节点构成的，所以我们必须首先创建一个类来表示节点。那么 Trie 树的节点需要什么属性呢？

对于任意一个节点而言，从根节点到当前节点的路径可能表示一个单词，也可能表示某个单词的前缀，所以说必须有一个 words 属性来表示单词的数量，prefixes 属性来表示前缀的数量，同时，每个节点是有若干个子节点的，所有必须有一个节点类型的数组，在这里，假设所有的字符都是小写字母，那么这个数组开到 26 即可。

Vertex 类：

```
// Trie树的节点
    protected class Vertex {
        protected int words;  //单词的个数
        protected int prefixes;   //前缀的个数
        protected Vertex[] edges;// 每个节点包含26个子节点(类型为自身)

        Vertex() {
            words = 0; //默认单词数是0
            prefixes = 0;  //默认前缀是0
            edges = new Vertex[26];   //保存子节点   这里开到26即可
            for (int i = 0; i < edges.length; i++) {
                edges[i] = null;
            }
        }
    }
```

Trie 树结构

Trie 树的根节点是不包含任何字符的，Trie 树结构中我们只需要一个对 Trie 树根节点的引用即可。

如下：

```
private Vertex root; // Trie树的根节点
```

插入新单词

在插入新单词的过程中，我们遍历整个字符串。对于每一个字符，都要判断它需要走哪一条路径，如果该路径上缺少节点，我们需要创建。如果到了最后一个字符，那么其对应节点的 words++，如果不是最后一个字符，那么其对应节点的 prefixes++。

代码如下，采取了递归的方式;

```
/**
 * 添加一个顶点
 * @param vertex   根节点
 * @param word 插入的单词
 */
private void addWord(Vertex vertex, String word) {
    if (word.length() == 0) { //如果所有的字符都被添加了
        vertex.words++;
    } else {
        vertex.prefixes++;
        char c = word.charAt(0);
        c = Character.toLowerCase(c);
        int index = c - 'a';  //求出该节点对应的索引
        if (vertex.edges[index] == null) { //如果该节点不存在 就创建
            vertex.edges[index] = new Vertex();
        }

        addWord(vertex.edges[index], word.substring(1));    //递归过程
    }
}
```

计算完全匹配单词个数

还是遍历字符串，根据每个字符进行选路。

代码：

```
/**
     * 计算完全匹配单词个数
     * @param vertex
     * @param wordSegment
     * @return
     */
    private int countWords(Vertex vertex, String wordSegment) {
        if (wordSegment.length() == 0) { //如果最后一个字符也选路结束了
            return vertex.words;
        }

        char c = wordSegment.charAt(0);
        int index = c - 'a';
        if (vertex.edges[index] == null) { // 单词不存在
            return 0;
        } else {
            return countWords(vertex.edges[index],
wordSegment.substring(1));
        }

    }
```

计算指定前缀单词个数

与计算完全匹配单词个数唯一不同的地方就是遍历完成后，返回的是最后节点的 prefixes 属性。

代码：

```
    /**
     * 计算指定前缀单词的个数
     * @param prefix
     * @return
     */
    public int countPrefixes(String prefix) {
        return countPrefixes(root, prefix);
    }

    private int countPrefixes(Vertex vertex, String prefixSegment) {
        if (prefixSegment.length() == 0) {
            return vertex.prefixes;
        }

        char c = prefixSegment.charAt(0);
        int index = c - 'a';
```

```
        if (vertex.edges[index] == null) { // the word does NOT exist
            return 0;
        } else {
            return countPrefixes(vertex.edges[index],
                    prefixSegment.substring(1));

        }

    }
```

红黑树

来源

红黑树本质上也是一棵二叉搜索树，就是前文提到的那种。那么既然有了二叉搜索树，为什么还要有红黑树呢？

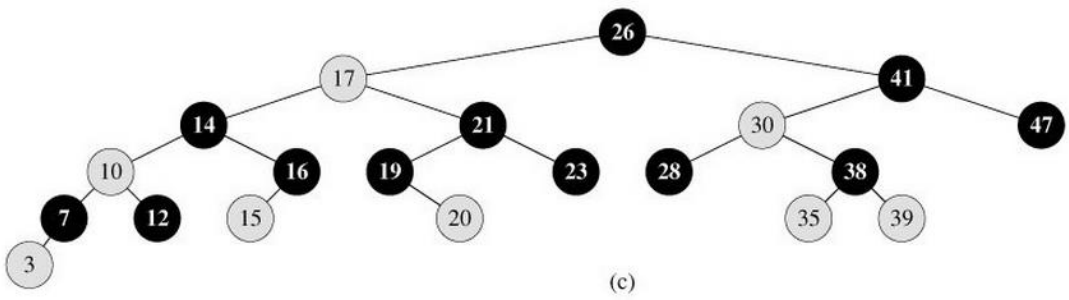
因为由 n 个节点构造二叉树具有随机性，假设给定的节点的数据项是依次递增的，那么二叉树就会退化成一棵具有 n 个节点的线性链。此时树中查找、删除等操作就会退化成 O(n)，而不是 O(log(n))。

红黑树就是为了均衡这种情况而诞生的，它使得不管输入如何，二叉树都不会退化成为线性链。它有自己的一些均衡手段，使得插入、删除、查找在最坏情况下的时间复杂度都能达到 O(log(N))。

特点

1.每个结点要么是红的，要么是黑的。
2.根结点是黑的。
3.每个叶结点是黑的。
4.如果一个结点是红的，那么它的俩个儿子都是黑的。
5.对每个结点，其到叶节点的每条路径上都包含相同数目的黑结点。

如下图，就是一棵红黑树(此图忽略了叶子节点)：

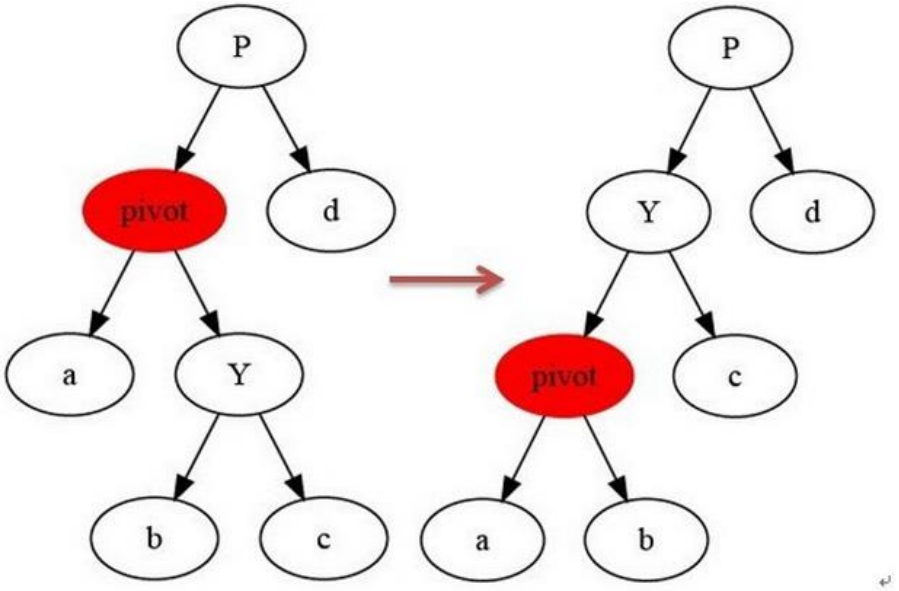


红黑树的旋转

在对红黑树进行插入和删除操作时，对树做了修改可能会破坏红黑树的性质，可以通过对树重新着色，或者对树进行旋转操作来使红黑树继续保持平衡。

树的旋转分为两种，左旋和右旋。

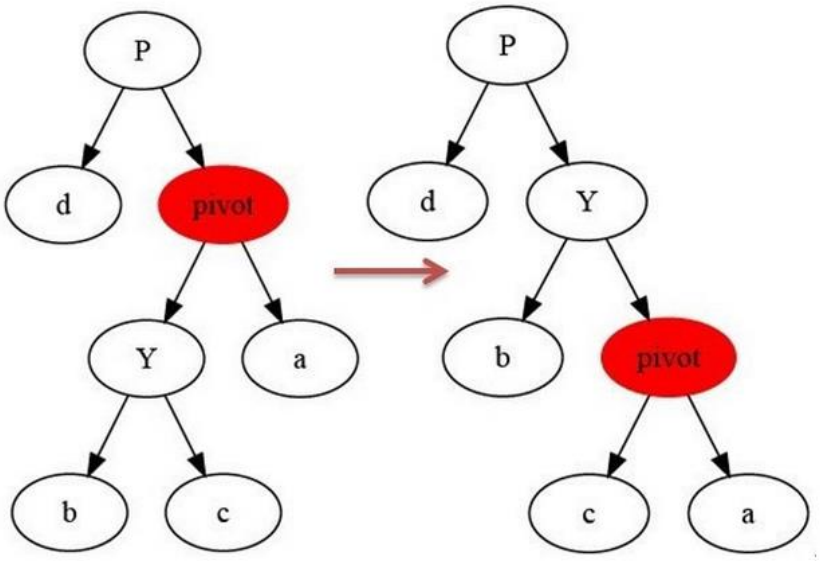
左旋：当在某个结点 pivot 上，做左旋操作时，我们假设它的右孩子 y 不是 NIL，pivot 可以为树内任意不是 NIL 的左子结点。。左旋以 pivot 到 y 之间的链为"支轴"进行，它使 y 成为该孩子树新的根，而 y 的左孩子 b 则成为 pivot 的右孩子。



算法导论对左旋的描述;

```
LEFT-ROTATE(T, x)
1  y ← right[x] ▹ Set y.
2  right[x] ← left[y]      ▹ Turn y's left subtree into x's right subtree.
3  p[left[y]] ← x
4  p[y] ← p[x]             ▹ Link x's parent to y.
5  if p[x] = nil[T]
6     then root[T] ← y
7     else if x = left[p[x]]
8             then left[p[x]] ← y
9             else right[p[x]] ← y
10 left[y] ← x             ▹ Put x on y's left.
11 p[x] ← y
```

右旋：和左旋类似，直接上图：



红黑树节点的插入

红黑树插入的操作相当于在二叉搜索树插入的基础上，为了重新恢复平衡，继续做了插入修复的操作。假设插入的节点是 z，算法导论中给出的伪代码如下：

RB-INSERT-FIXUP(T, z)，如下所示：

```
 1 while color[p[z]] = RED
 2     do if p[z] = left[p[p[z]]]
 3           then y ← right[p[p[z]]]
 4                if color[y] = RED
 5                   then color[p[z]] ← BLACK                ▹ Case 1
 6                        color[y] ← BLACK                   ▹ Case 1
 7                        color[p[p[z]]] ← RED               ▹ Case 1
 8                        z ← p[p[z]]                        ▹ Case 1
 9                   else if z = right[p[z]]
10                           then z ← p[z]                   ▹ Case 2
11                                LEFT-ROTATE(T, z)          ▹ Case 2
12                        color[p[z]] ← BLACK                ▹ Case 3
13                        color[p[p[z]]] ← RED               ▹ Case 3
14                        RIGHT-ROTATE(T, p[p[z]])           ▹ Case 3
15           else (same as then clause
                      with "right" and "left" exchanged)
```

16 color[root[T]] ← BLACK

先来重新回顾一下红黑树的性质：

1.每个结点要么是红的，要么是黑的。

2.根结点是黑的。

3.每个叶结点是黑的。

4.如果一个结点是红的，那么它的俩个儿子都是黑的。

5.对每个结点，其到叶节点的每条路径上都包含相同数目的黑结点。

在对红黑树进行插入操作时，我们一般总是插入红色的结点，因为这样可以在插入过程中尽量避免对树的调整。那么，我们插入一个结点后，可能会使原树的哪些性质改变列?由于，我们是按照二叉树的方式进行插入，因此元素的搜索性质不会改变。

如果插入的结点是根结点，性质 2 会被破坏，如果插入结点的父结点是红色，则会破坏性质 4。因此，总而言之，插入一个红色结点只会破坏性质 2 或性质 4。

如果是第一种情况，插入的节点是根节点，那么只会违反性质 2，只需要把它着色为黑色即可。

如果是第二种情况，插入的节点的父节点是黑色，此时红黑树的性质没有被破坏，不需要改动。

如果是第三种情况，插入节点的父节点是红色，那么我们需要穷举所有的可能性，之后把能归于同一类方法处理的归为同一类，不能直接处理的化归到下面的几种情况。

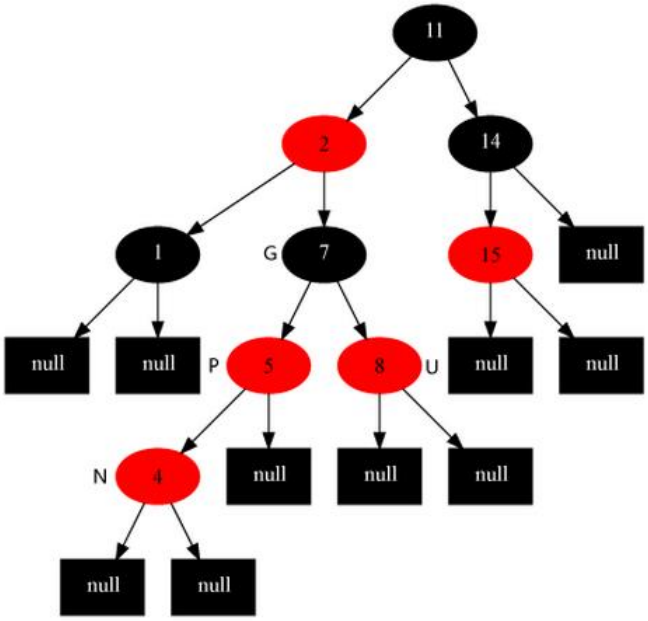
我们看一下第三种情况的分类：

1.当前结点的父结点是红色且祖父结点的另一个子结点（叔叔结点）是红色。

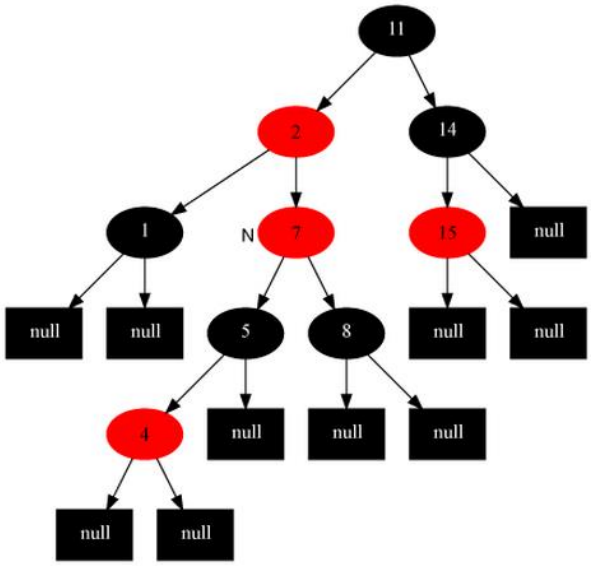
此时父结点的父结点一定存在，否则插入前就已不是红黑树。与此同时，又分为父结点是祖父结点的左子还是右子，对于对称性，我们只要解开一个方向就可以了。在此，我们只考虑父结点为祖父左子的情况。

同时，还可以分为当前结点是其父结点的左子还是右子，但是处理方式是一样的。我们将此归为同一类。对策：将当前节点的父节点和叔叔节点涂黑，祖父结点涂红，把当前结点指向祖父节点，从新的当前节点重新开始算法。

变化前(插入 4 节点)：



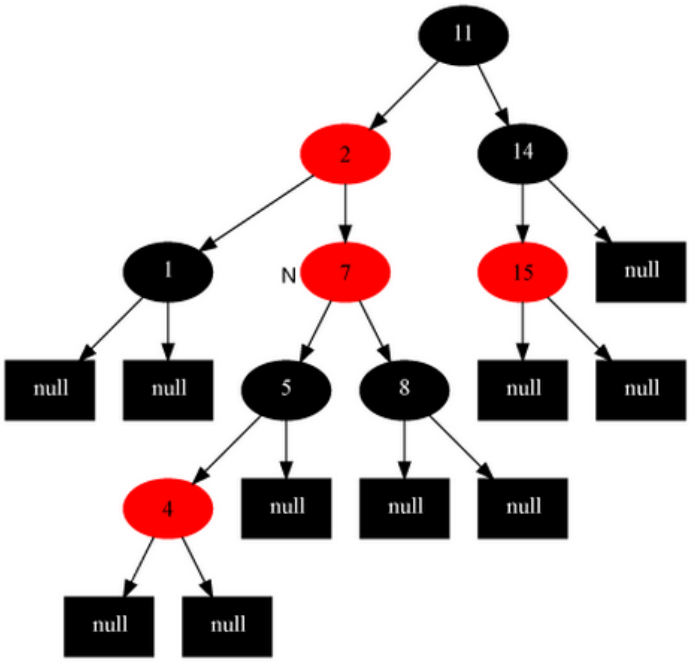
变化后



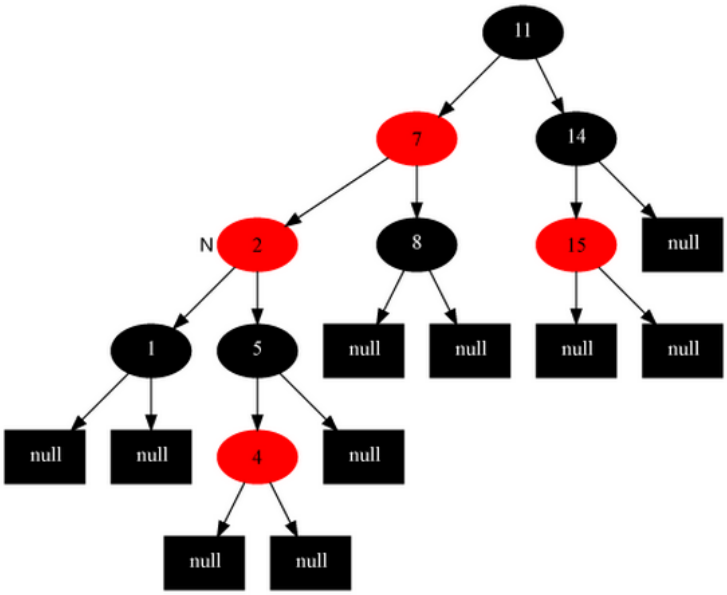
2.当前节点的父节点是红色,叔叔节点是黑色，当前节点是其父节点的右子

对策：当前节点的父节点做为新的当前节点，以新当前节点为支点左旋。

插入前(插入 7 节点)：



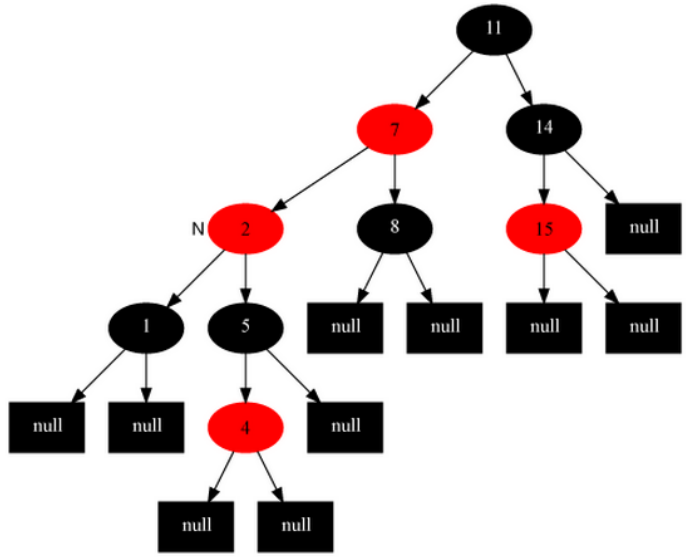
插入后：



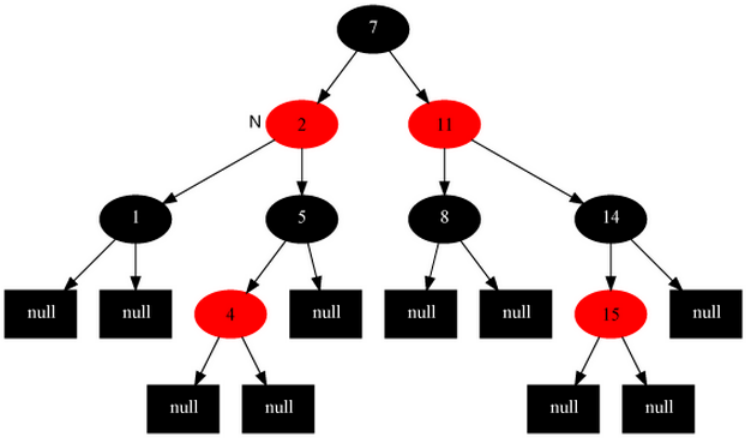
3. 当前节点的父节点是红色,叔叔节点是黑色，当前节点是其父节点的左子

解法：父节点变为黑色，祖父节点变为红色，在祖父节点为支点右旋

变化前(插入 2 节点)：



插入后：



这里只介绍插入操作。如果想了解删除或者更多的操作，可以看 July 的博客或者算法导论。旋转操作是红黑树的关键，其他的操作大多依赖于旋转。

代码

```
package com.algorithm.rbtree;

/**
 *  红黑树结构的实现
 */
public class RBTree {

    private final Node NIL = new Node(null,null,null,Color.BLACK,-1);
//初始化一个NIL节点 总是黑色
```

```
    private Node root;  //红黑树的根节点

    public RBTree() {
        root = NIL;  //初始化根节点=NIL
    }

    public RBTree(Node  root) {
        this.root = root;  //有参构造函数
    }




    /**
     * 向红黑树插入一个节点
     * @param node
     */
    public void rbInsert(Node node) {

        Node previous = NIL;
        Node temp = root;  //用作向下寻找

        while (temp != NIL) {
            previous = temp;
            if (temp.getValue() < node.getValue()) {
                temp = temp.getRight();
            } else {
                temp = temp.getLeft();
            }
        }  //while循环就是查找合适位置的过程    循环结束时previous指向的就是要插入节点的父节点
        node.setParent(previous);

        if (previous == NIL) {  //树是空树 插入的就是根
            root = node;
            root.setParent(NIL);
        } else  if (previous.getValue() > node.getValue()) {
            previous.setLeft(node);
        } else {
            previous.setRight(node);
        }

        node.setLeft(NIL);
```

```
        node.setRight(NIL);
        node.setColor(Color.RED);
        rb_Insert_Fixup(node);  //对树进行调整

    }

    //插入节点后的调整
    private void rb_Insert_Fixup(Node node) {

        while (node.getParent().getColor() == Color.RED) {

            if (node.getParent() ==
node.getParent().getParent().getLeft()) {
                //如果插入节点的父节点是其祖父节点的左孩子

                Node rightNuncle =
node.getParent().getParent().getRight();//得到叔叔节点

                if (rightNuncle.getColor() == Color.RED) { //叔叔节点是红
色
                    //将当前节点的父节点和叔叔节点涂黑，祖父结点涂红，把当前结
点指向祖父节点
                    rightNuncle.setColor(Color.BLACK);
                    node.getParent().setColor(Color.BLACK);
                    node.getParent().getParent().setColor(Color.RED);
                    node = node.getParent().getParent();

                } else if (node == node.getParent().getRight()) { // 叔叔
节点是黑色并且自己是父节点的右孩子
                    node = node.getParent();
                    leftRotate(node); //左旋

                } else {
                    // 叔叔节点是黑色并且自己是父节点的左孩子
                    node.getParent().setColor(Color.BLACK);
                    node.getParent().getParent().setColor(Color.RED);
                    //父节点变为黑色，祖父节点变为红色，在祖父节点为支点右旋
                    rightRotate(node.getParent().getParent());  //右旋
                }

            } else {
                //插入节点的父节点是其祖父节点的右孩子
                Node leftNuncle = node.getParent().getParent().getLeft();
//得到叔叔节点
```

```
            //有对称性 这里的操作和上面的大if里的操作相同
            if (leftNuncle.getColor() == Color.RED) {

                leftNuncle.setColor(Color.BLACK);
                node.getParent().setColor(Color.BLACK);
                node.getParent().getParent().setColor(Color.RED);
                node = node.getParent().getParent();

            } else if (node == node.getParent().getLeft()) {

                node = node.getParent();
                rightRotate(node);

            } else {

                node.getParent().setColor(Color.BLACK);
                node.getParent().getParent().setColor(Color.RED);
                leftRotate(node.getParent().getParent());

            }

        }


    }
    root.setColor(Color.BLACK);

}

//左转函数
private void leftRotate(Node node) {

    Node rightNode = node.getRight();

    node.setRight(rightNode.getLeft());
    if (rightNode.getLeft() != NIL) {
        rightNode.getLeft().setParent(node);
    }
    rightNode.setParent(node.getParent());

    if (node.getParent() == NIL) {
        rightNode = root;
    } else if (node == node.getParent().getLeft()) {
```

```
            node.getParent().setLeft(rightNode);
        } else {
            node.getParent().setRight(rightNode);
        }

        rightNode.setLeft(node);
        node.setParent(rightNode);


    }

    //右转函数
    private void rightRotate(Node node) {

        Node leftNode = node.getLeft();
        node.setLeft(leftNode.getRight());

        if (leftNode.getRight() != null) {
            leftNode.getRight().setParent(node);
        }

        leftNode.setParent(node.getParent());

        if (node.getParent() == NIL) {
            root = leftNode;
        } else if (node == node.getParent().getLeft()) {
            node.getParent().setLeft(leftNode);
        } else {
            node.getParent().setRight(leftNode);
        }

        leftNode.setRight(node);
        node.setParent(leftNode);

    }

    //中序遍历红黑树
    public void printTree() {
        inOrderTraverse(root);
    }

    private void inOrderTraverse(Node node) {

        if (node != NIL) {
```

```
            inOrderTraverse(node.getLeft());
            System.out.println(" 节点：“+node.getValue() + "的颜色为： " +
node.getColor());
            inOrderTraverse(node.getRight());
        }

    }


    public Node getNIL() {
        return NIL;
    }

}


/**
 *  红黑树节点
 */
class Node {
    private Node left;  //左孩子
    private Node right;  //右孩子
    private Node parent;  //父节点
    private Color color;  //颜色  枚举类型
    private int value;  //值
    public Node(Node left, Node right, Node parent, Color color, int
value) {  //构造函数初始化
        super();
        this.left = left;
        this.right = right;
        this.parent = parent;
        this.color = color;
        this.value = value;
    }

    public Node() {  //空构造函数
    }

    public Node(int value) {
        this(null,null,null,null,value);
    }

    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;
    }

    public Node getParent() {
        return parent;
    }

    public void setParent(Node parent) {
        this.parent = parent;
    }

    public Color getColor() {
        return color;
    }

    public void setColor(Color color) {
        this.color = color;
    }

    public int getValue() {
        return value;
    }

    public void setValue(int value) {
        this.value = value;
    }

}

/**
 * 枚举颜色
 */
enum Color {
```

```
    RED,BLACK
}
```

B 树

B 树索引

在大规模数据存储中实现索引查询这样一个实际背景下，树节点存储的元素是有限的，这样就导致二叉查找树的结构由于树的深度过大而造成磁盘读写过于频繁，进而导致查询效率低下。因此，应该想办法降低树的深度，进而减少磁盘查找和存储的次数。一个基本的想法就是多叉树，所以，B 树——平衡多路查找树应运而生。

B 树的各种操作能够使 B 树保持较低的高度，从而有效的避免过于频繁的查找和存取操作，进而达到有效提高查找效率的目的。

硬件相关知识

为了更好地了解为什么需要 B 树这种外存储器数据结构，我们必须先了解一下相关的硬件知识。

外存储器：硬盘

计算机存储设备一般分为内存储器和外存储器两种，内存储器的存取速度块，但容量小，价格昂贵，而且不能长期保存数据(断电情况下会消失)。外存储器是一种直接存取的存储设备，与内存储器相比，存储速度慢一些，但是容量大，价格便宜，且断电之后数据不会消失。

磁盘的构造

磁盘是一个扁平的圆盘(与电唱机的唱片类似)。盘面上有许多称为磁道的圆圈，数据就记录在这些磁道上。磁盘可以是单片的，也可以是由若干盘片组成的盘组，每一盘片上有两个面。如下图 11.3 中所示的 6 片盘组为例，除去最顶端和最底端的外侧面不存储数据之外，一共有 10 个面可以用来保存信息。

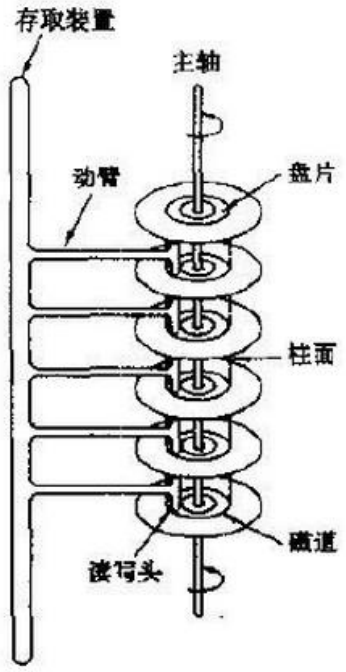


图 11.3　活动头盘示意图

盘片装在一个主轴上，并饶主轴高速旋转，当磁盘驱动器执行读写操作时，磁道在读写头(又称磁头)下通过，就可以进行数据的读写了。

一般磁盘分为固定头盘(磁头固定)和活动头盘。固定头盘的每一个磁道上都有独立的磁头，它是固定不动的，专门负责这一磁道上数据的读/写。

活动头盘　(如上图)的磁头是可移动的。每一个盘面上只有一个磁头(磁头是双向的，因此正反盘面都能读写)。它可以从该面的一个磁道移动到另一个磁道。所有磁头都装在同一个动臂上，因此不同盘面上的所有磁头都是同时移动的(行动整齐划一)。当盘片绕主轴旋转的时候，磁头与旋转的盘片形成一个圆柱体。各个盘面上半径相同的磁道组成了一个圆柱面，我们称为柱面　。因此，柱面的个数也就是盘面上的磁道数。

磁盘的读写效率和原理

磁盘上数据必须用一个三维地址唯一标示：柱面号、盘面号、块号(磁道上的盘块)。

读/写磁盘上某一指定数据需要下面 3 个步骤：

(1)　首先移动臂根据柱面号使磁头移动到所需要的柱面上，这一过程被称为定位或查找　。

(2)　如上图 11.3 中所示的 6 盘组示意图中，所有磁头都定位到了 10 个盘面的 10 条磁道上(磁头都是双向的)。这时根据盘面号来确定指定盘面上的磁道。

(3)　盘面确定以后，盘片开始旋转，将指定块号的磁道段移动至磁头下。

经过上面三个步骤，指定数据的存储位置就被找到。这时就可以开始读/写操作了。

访问某一具体信息，由 3 部分时间组成：

- 查找时间(seek time) Ts: 完成上述步骤(1)所需要的时间。这部分时间代价最高，最大可达到 0.1s 左右。
- 等待时间(latency time) Tl: 完成上述步骤(3)所需要的时间。由于盘片绕主轴旋转速度很快，一般为 7200 转/分(电脑硬盘的性能指标之一， 家用的普通硬盘的转速一般有 5400rpm(笔记本)、7200rpm 几种)。因此一般旋转一圈大约 0.0083s。
- 传输时间(transmission time) Tt: 数据通过系统总线传送到内存的时间，一般传输一个字节(byte)大概 0.02us=2*10^(-8)s

磁盘读取数据是以盘块(block)为基本单位的。位于同一盘块中的所有数据都能被一次性全部读取出来。而磁盘 IO 代价主要花费在查找时间 Ts 上。因此我们应该尽量将相关信息存放在同一盘块，同一磁道中。或者至少放在同一柱面或相邻柱面上，以求在读/写信息时尽量减少磁头来回移动的次数，避免过多的查找时间 Ts。

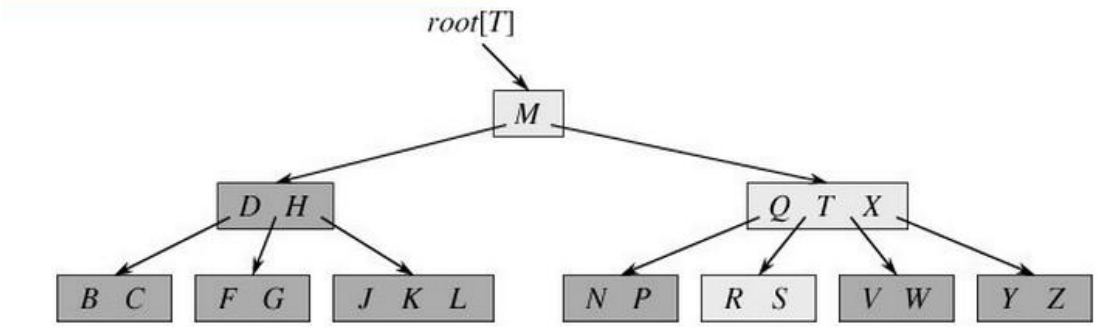
所以，在大规模数据存储方面，大量数据存储在外存磁盘中，而在外存磁盘中读取/写入块(block)中某数据时，首先需要定位到磁盘中的某块，如何有效地查找磁盘中的数据，需要一种合理高效的外存数据结构，就是下面所要重点阐述的 B-tree 结构，以及相关的变种结构：B^+-tree 结构。

什么是 B 树

B 树是为磁盘或者其他存储设备设计的一种多叉(相对于二叉)平衡查找树。降低磁盘 IO 方面 B 树表现的很好，很多数据库系统都使用 B 树或者 B 树的变种。

B 树和红黑树最大的不同就是 B 树的节点可以有多个孩子，从几个到几千个。不过，B 树和红黑树也有相同点，一棵含 n 个节点的 B 树的高度也为 O(log(n))，但可能比一棵红黑树的高度小很多，因为它的分支分子比较大。所以 B 树可以在 O(log(n))的时间复杂度内完成插入、删除等动态操作集合。

如下图所示，即是一棵 B 树，一棵关键字为英语中辅音字母的 B 树，现在要从树种查找字母 R（包含 n[x]个关键字的内结点 x，x 有 n[x]+1]个子女（也就是说，一个内结点 x 若含有 n[x]个关键字，那么 x 将含有 n[x]+1 个子女)。所有的叶结点都处于相同的深度，带阴影的结点为查找字母 R 时要检查的结点)。



相信，从上图你能轻易的看到，一个内结点 x 若含有 n[x]个关键字，那么 x 将含有 n[x]+1 个子女。如含有 2 个关键字 D H 的内结点有 3 个子女，而含有 3 个关键字 Q T X 的内结点有 4 个子女。

B 树定义

一棵 m 阶的 B 树定义如下：

1. 树中每个结点最多含有 m 个孩子（m>=2）；
2. 除根结点和叶子结点外，其它每个结点至少有[ceil(m / 2)]个孩子（其中 ceil(x)是一个取上限的函数）；
3. 根节点至少包含两棵子树(除非根节点)。
4. 所有叶子结点都出现在同一层，叶子结点不包含任何关键字信息。
5. 有 j 个孩子的非叶结点恰好有 j-1 个关键码，关键码按照递增顺序排列。

每个节点包含的关键字的个数有上界和下界，用一个被称为 B 树的最小度数的固定整数 t>=2 表示，每个节点的关键字个数都是大于等于 t-1，小于等于 2*t-1。

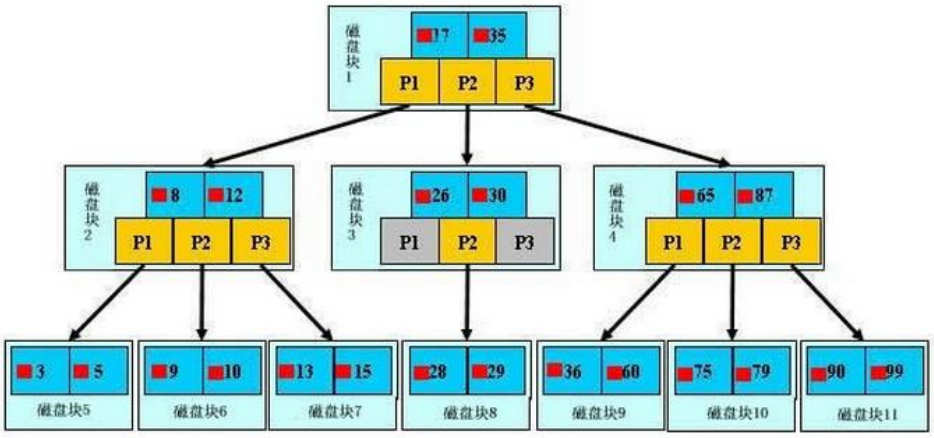
B 树中的每个节点可以根据实际情况包含大量的关键字信息和分支(当然不能超过磁盘块的大小，根据磁盘驱动器的不同，一般块的大小在 1KB 到 4KB)，这样树的深度降低了，这就意味着查找一个元素只需要很少的节点从外存储器中读入内存即可，从而可以很快的访问到要查找的数据。

文件的查找过程

B 树节点可以定义如下：

```
//BTree的节点
    private class BTNode {
        private int number = 0;  //文件数量
        private List<E> values = new ArrayList<E>();  //文件名
```

```
        private List<BTNode> children = new ArrayList<BTNode>();  //指向
子节点的引用
        private boolean isLeaf = false;  //是否是叶子节点

        E getKey(int i) {
            return values.get(i);
        }

        BTNode getChildren(int i) {
            return children.get(i);
        }

        void AddKey(int i, E element) {
            values.add(i, element);
        }

        void removeKey(int i) {
            values.remove(i);
        }

        void AddChildren(int i, BTNode c) {
            children.add(i, c);
        }

        void removeChildren(int i) {
            children.remove(i);
        }

        boolean isFull() {
            if (number == fullNum)
                return true;
            return false;
        }

        int getSize() {
            return values.size();
        }

        boolean isNull() {
            return (this == NullBTNode);
        }

    }
```



为了简单，这里用少量数据构造一棵 3 叉树的形式，实际应用中的 B 树结点中关键字很多的。上面的图中比如根结点，其中 17 表示一个磁盘文件的文件名；小红方块表示这个 17 文件内容在硬盘中的存储位置；p1 表示指向 17 左子树的指针。

假如每个盘块可以正好存放一个 B 树的结点（正好存放 2 个文件名）。那么一个 BTNODE 结点就代表一个盘块，而子树指针就是存放另外一个盘块的地址。

下面，咱们来模拟下查找文件 29 的过程：

1. 根据根结点指针找到文件目录的根磁盘块 1，将其中的信息导入内存。【磁盘 IO 操作 1 次】
2. 此时内存中有两个文件名 17、35 和三个存储其他磁盘页面地址的数据。根据算法我们发现：17<29<35，因此我们找到指针 p2。
3. 根据 p2 指针，我们定位到磁盘块 3，并将其中的信息导入内存。【磁盘 IO 操作 2 次】
4. 此时内存中有两个文件名 26，30 和三个存储其他磁盘页面地址的数据。根据算法我们发现：26<29<30，因此我们找到指针 p2。
5. 根据 p2 指针，我们定位到磁盘块 8，并将其中的信息导入内存。【磁盘 IO 操作 3 次】
6. 此时内存中有两个文件名 28，29。根据算法我们查找到文件名 29，并定位了该文件内存的磁盘地址。

分析上面的过程，发现需要 **3 次磁盘 IO 操作和 3 次内存查找**操作。关于内存中的文件名查找，由于是一个有序表结构，可以利用折半查找提高效率。至于 IO 操作是影响整个 B 树查找效率的决定因素。

当然，如果我们使用平衡二叉树的磁盘存储结构来进行查找，磁盘 4 次，最多 5 次，而且文件越多，B 树比平衡二叉树所用的磁盘 IO 操作次数将越少，效率也越高。

B 树高度

对于辅助存储做 IO 读的次数取决于 B 树的高度，B 树的高度怎么求呢？

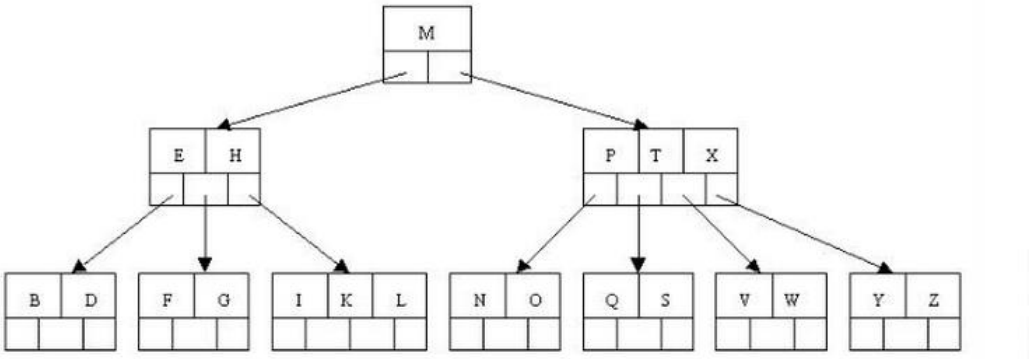
对于一棵含有 n 个关键字，m 阶的 B 树来说，当树的高度从 0 开始计数时，其高度 h 为：

h <= $\log_{ceil(m/2)}(n+1)/2$。

这个公式从侧面反映出 B 树的查找效率相当高。为什么呢？因为底 m/2 可以取相当大，m 可以达到上千，因此在关键数一定的情况下，m 越大 h 就越小，树的高度降低，磁盘 IO 存取的次数也会降低。

B 树的插入

看一个例子，以一棵五阶 B 树为例(任何一个节点最多含有四个关键字，5 个子树)，关键字为大写字母，顺序为字母升序。

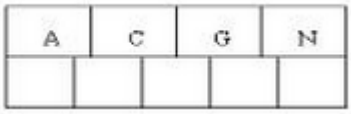


针对一棵高度为 h 的 m 阶 B 树，插入一个元素时，首先判断这个元素在 B 中是否存在，如果不存在，一般在叶节点中插入该元素，此时分为三种情况：

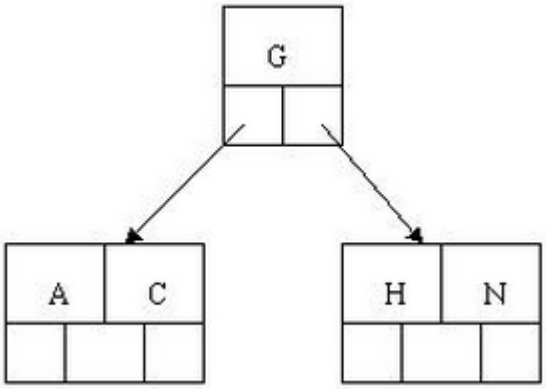
1. 如果叶节点空间足够，即该节点的关键字数小于 m-1，则直接插入在叶节点的左面或者右面。
2. 如果空间满了以致没有足够的空间去添加新元素，即该节点的关键字数已经有了 m 个，则需要将该节点进行分裂，将一半的关键字元素分裂到新的其相邻的右节点中，中间关键字元素上移到父节点中，而且当节点中关键字元素向右移动时，相关的指针也需要向右移动。
3. 此外，如果在上述的第二种情况中，由于中间关键字上移到父节点的过程中，导致根节点空间满了，那么根节点也要进行分裂操作，这样，原来的根节点中的中间关键字元素上移到新的根节点中，从而导致树的高度增加一层

下面通过一个插入实例来看一下，插入以下字符到一棵空 5 阶 B 树中，C N G A H E K Q M F W L T Z D P R X Y S。。注意一步步的过程。

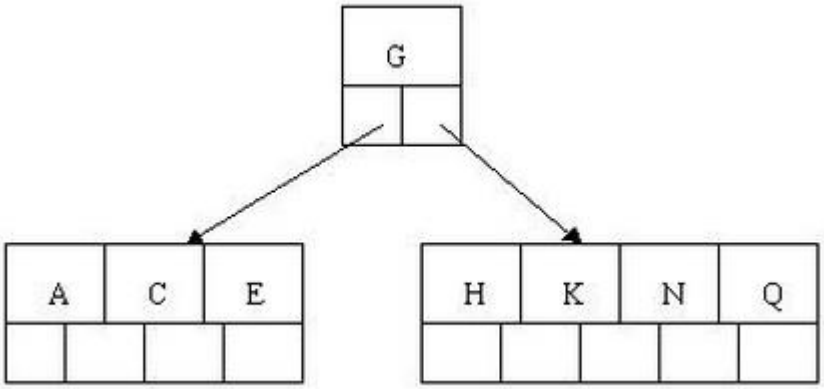
1. 首先，结点空间足够，4 个字母插入相同的结点中，如下图：



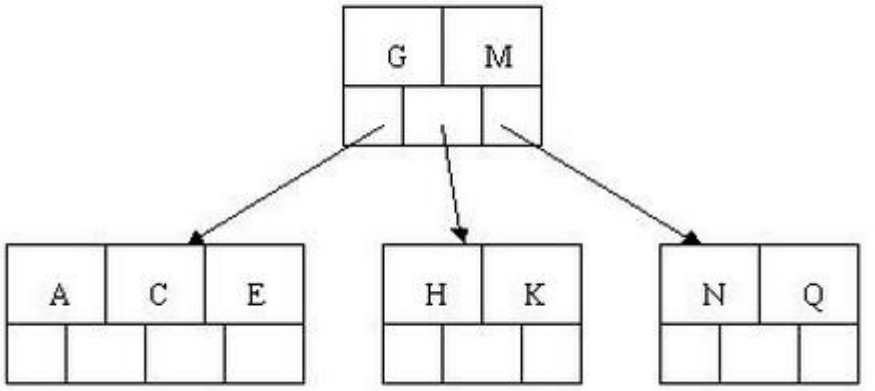
2. 当咱们试着插入 H 时，结点发现空间不够，以致将其分裂成 2 个结点，移动中间元素 G 上移到新的根结点中，在实现过程中，咱们把 A 和 C 留在当前结点中，而 H 和 N 放置新的其右邻居结点中。如下图：



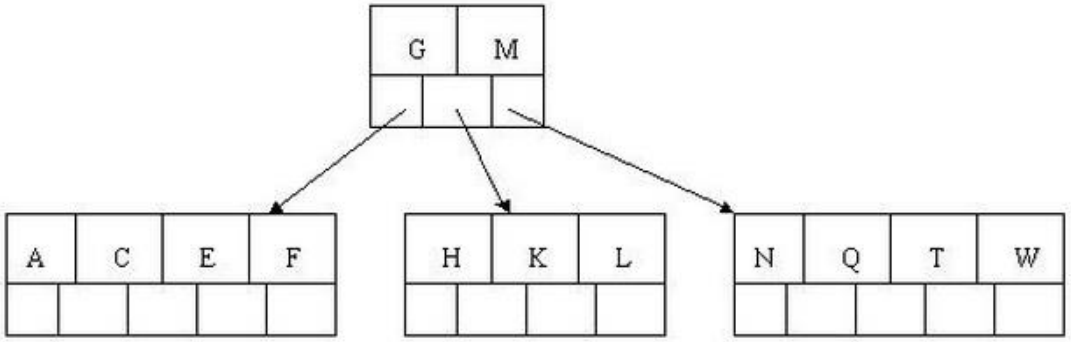
3. 当咱们插入 E,K,Q 时，不需要任何分裂操作



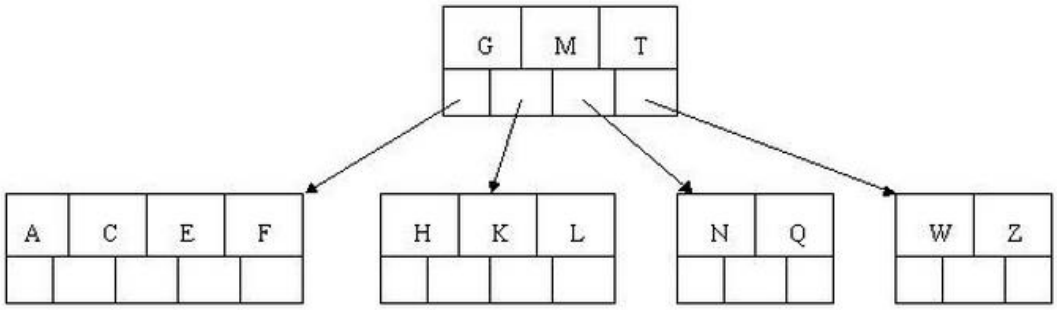
4. 插入 M 需要一次分裂，注意 M 恰好是中间关键字元素，以致向上移到父节点中



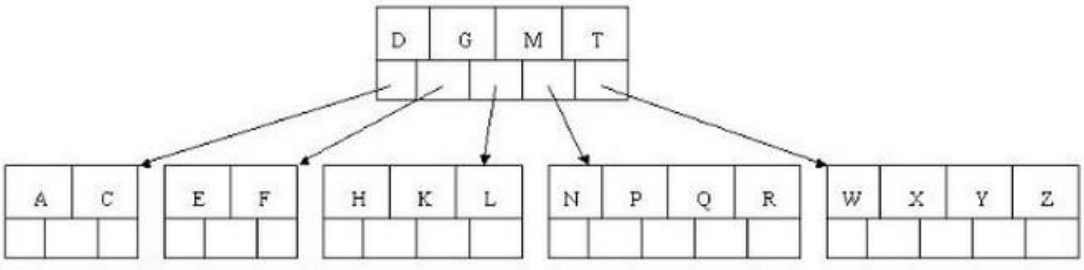
5. 插入 F,W,L,T 不需要任何分裂操作



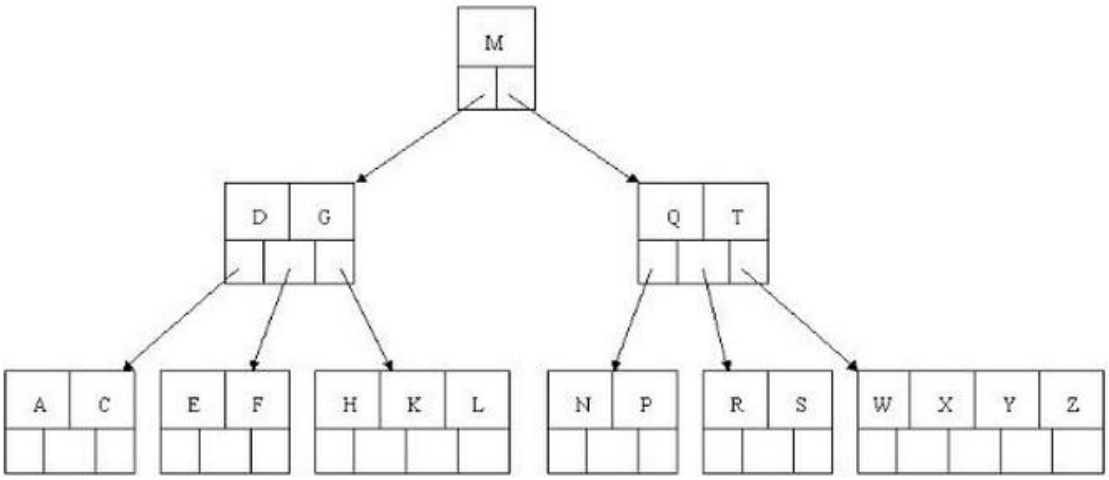
6. 插入 Z 时，最右的叶子结点空间满了，需要进行分裂操作，中间元素 T 上移到父节点中，注意通过上移中间元素，树最终还是保持平衡，分裂结果的结点存在 2 个关键字元素。



7. 插入 D 时，导致最左边的叶子结点被分裂，D 恰好也是中间元素，上移到父节点中，然后字母 P,R,X,Y 陆续插入不需要任何分裂操作（别忘了，树中至多 5 个孩子）



8. 最后，当插入 S 时，含有 N,P,Q,R 的结点需要分裂，把中间元素 Q 上移到父节点中，但是情况来了，父节点中空间已经满了，所以也要进行分裂，将父节点中的中间元素 M 上移到新形成的根结点中，注意以前在父节点中的第三个指针在修改后包括 D 和 G 节点中。这样具体插入操作的完成，下面介绍删除操作，删除操作相对于插入操作要考虑的情况多点。



看懂了这八步，就初步了解了 B 树的插入操作。插入操作的代码会在最后展示。

B 树的删除

B 树的删除操作相对于插入操作要麻烦一些。

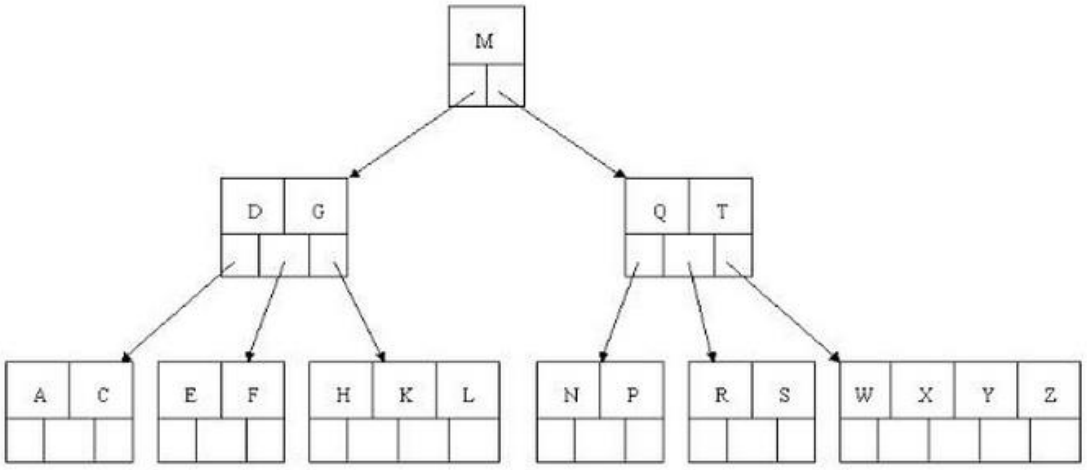
首先查找 B 树中需要删除的元素，如果该元素在 B 树中存在，则将该元素在其节点中进行删除，删除元素时，还要判断该元素有无左右孩子节点。

- 如果没有左右孩子节点，则上移孩子节点中的某相近元素(左孩子最右面的节点或者右孩子最左面的节点)到父节点中，然后移动相关元素。
- 如果没有左右孩子节点，直接删除，然后移动相应元素

再删除元素和移动相应元素之后，如果某节点中元素数小于 ceil(m/2)-1，则需要查看其某个相邻兄弟节点是否丰满(即节点中元素个数是否大于 ceil(m/2)-1)。

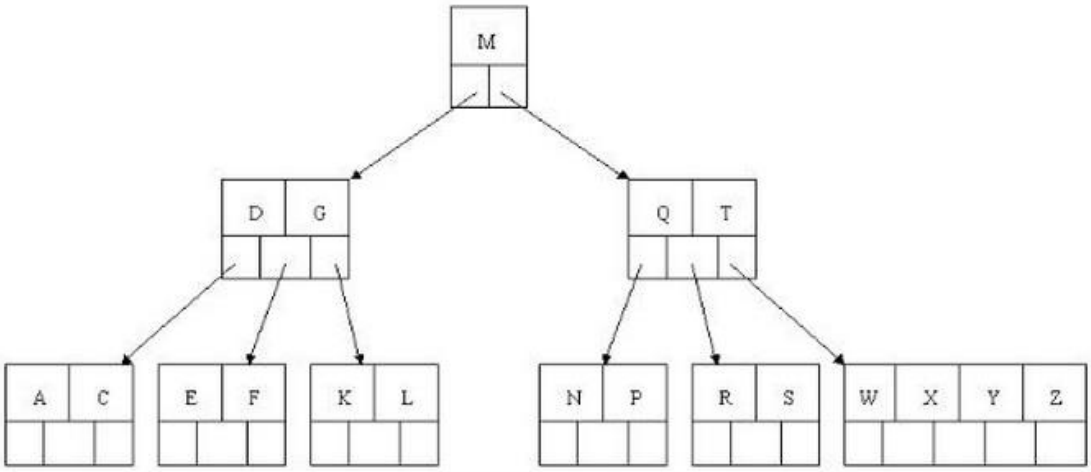
- 如果丰满，则向父节点借一个元素来满足条件
- 如果其相邻兄弟都刚脱贫，则该节点与其相邻节点的某一兄弟节点合并成一个节点，以此来满足条件。

还是来看一个具体的例子：

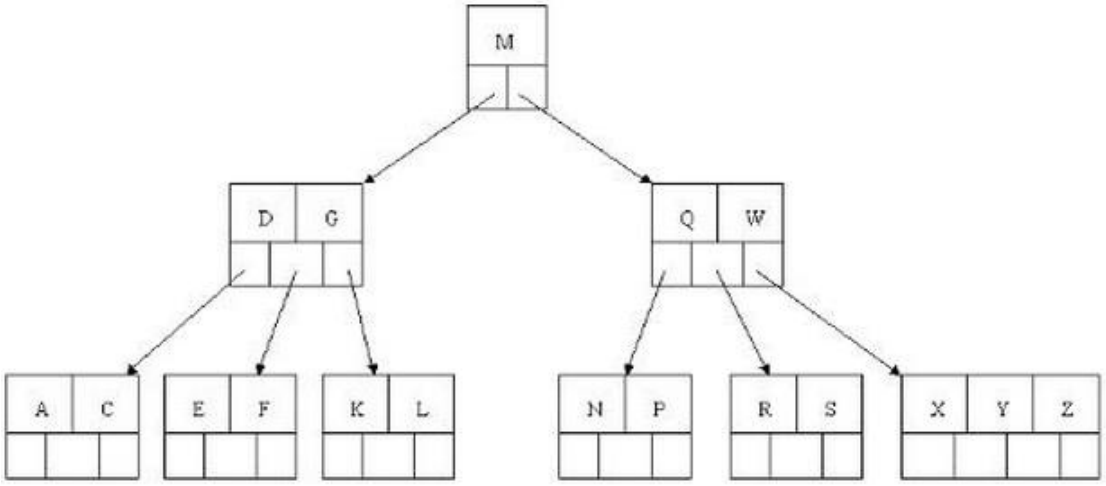


我们需要依次删除 H,T,R,E 节点，下面看一下具体的步骤。

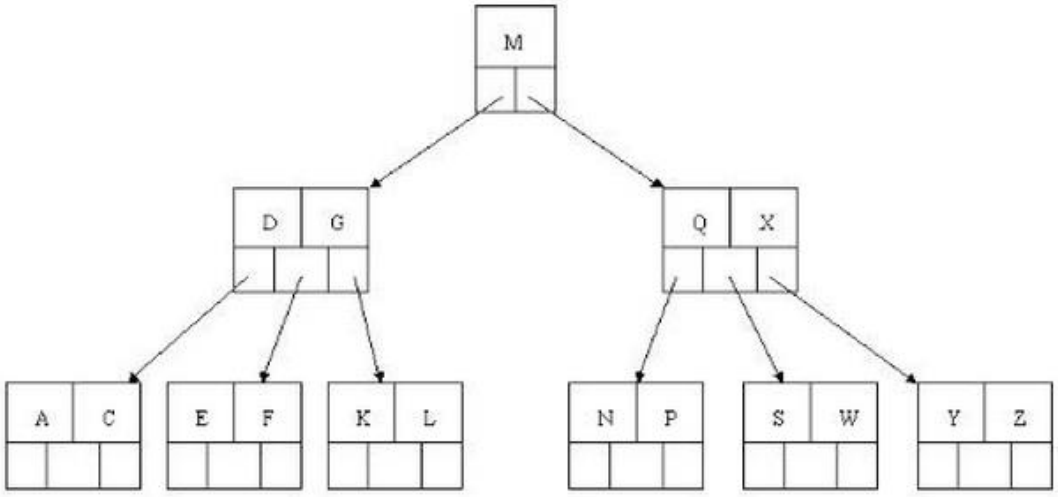
1. 首先删除元素 H，当然首先查找 H，H 在一个叶子结点中，且该叶子结点元素数目 3 大于最小元素数目 ceil(m/2)-1=2，则操作很简单，咱们只需要移动 K 至原来 H 的位置，移动 L 至 K 的位置（也就是结点中删除元素后面的元素向前移动）



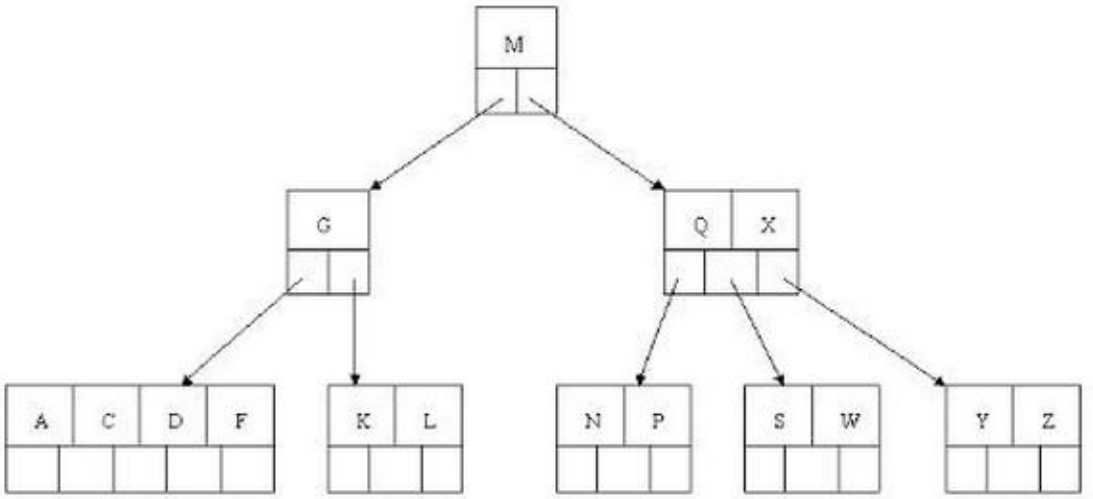
2. 下一步，删除 T,因为 T 没有在叶子结点中，而是在中间结点中找到，咱们发现他的继承者 W(字母升序的下个元素)，将 W 上移到 T 的位置，然后将原包含 W 的孩子结点中的 W 进行删除，这里恰好删除 W 后，该孩子结点中元素个数大于 2，无需进行合并操作。



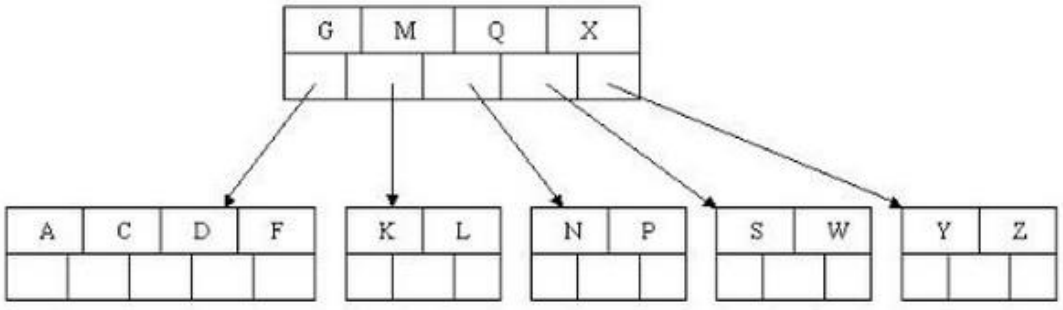
3. 下一步删除 R，R 在叶子结点中,但是该结点中元素数目为 2，删除导致只有 1 个元素，已经小于最小元素数目 ceil(5/2)-1=2,而由前面我们已经知道：如果其某个相邻兄弟结点中比较丰满（元素个数大于 ceil(5/2)-1=2），则可以向父结点借一个元素，然后将最丰满的相邻兄弟结点中上移最后或最前一个元素到父节点中（有没有看到红黑树中左旋操作的影子?），在这个实例中，右相邻兄弟结点中比较丰满（3 个元素大于 2），所以先向父节点借一个元素 W 下移到该叶子结点中，代替原来 S 的位置，S 前移；然后 X 在相邻右兄弟结点中上移到父结点中，最后在相邻右兄弟结点中删除 X，后面元素前移。



4. 最后一步删除 E，删除后会导致很多问题，因为 E 所在的结点数目刚好达标，刚好满足最小元素个数（ceil(5/2)-1=2）,而相邻的兄弟结点也是同样的情况，删除一个元素都不能满足条件，所以需要该节点与某相邻兄弟结点进行合并操作；首先移动父结点中的元素（该元素在两个需要合并的两个结点元素之间）下移到其子结点中，然后将这两个结点进行合并成一个结点。所以在该实例中，咱们首先将父节点中的元素 D 下移到已经删除 E 而只有 F 的结点中，然后将含有 D 和 F 的结点和含有 A,C 的相邻兄弟结点进行合并成一个结点。



5. 也许你认为这样删除操作已经结束了，其实不然，在看看上图，对于这种特殊情况，你立即会发现父节点只包含一个元素 G，没达标（因为非根节点包括叶子结点的关键字数 n 必须满足于 2=<n<=4，而此处的 n=1），这是不能够接受的。如果这个问题结点的相邻兄弟比较丰满，则可以向父结点借一个元素。假设这时右兄弟结点（含有 Q,X）有一个以上的元素（Q 右边还有元素），然后咱们将 M 下移到元素很少的子结点中，将 Q 上移到 M 的位置，这时，Q 的左子树将变成 M 的右子树，也就是含有 N，P 结点被依附在 M 的右指针上。所以在这个实例中，咱们没有办法去借一个元素，只能与兄弟结点进行合并成一个结点，而根结点中的唯一元素 M 下移到子结点，这样，树的高度减少一层。



至此，删除操作也完成了。下面我们来具体的看一下如何使用 Java 实现这些操作。

Java 实现 B 树

```
package com.algorithm.bTree;
import java.util.ArrayList;
import java.util.LinkedList;
import java.util.List;
import java.util.Queue;
import java.util.Random;
```

```java
/**
 * B树的实现
 *
 * @param <K>
 * @param <V>
 */
public class BTree<K, V>
{

    /**
     * 在B树节点中搜索给定键值的返回结果。
     * 该结果有两部分组成。第一部分表示此次查找是否成功，
     * 如果查找成功，第二部分表示给定键值在B树节点中的位置，
     * 如果查找失败，第二部分表示给定键值应该插入的位置。
     */
    private static class SearchResult
    {
        private boolean result;
        private int index;

        public SearchResult(boolean result, int index)
        {
            this.result = result;
            this.index = index;
        }

        public boolean getResult()
        {
            return result;
        }

        public int getIndex()
        {
            return index;
        }
    }

    /**
     * 简单起见  只支持整形的key
     * B树的节点
     */
    private static class BTreeNode
```

```
{
    private List<Integer> keys; //节点的关键字
    private List<BTreeNode> children;   //孩子节点引用
    private boolean leaf;   //是否是叶子节点

    public BTreeNode()
    {
        //完成初始化
        keys = new ArrayList<Integer>();
        children = new ArrayList<BTreeNode>();
        leaf = false;
    }

    public boolean isLeaf()
    {
        return leaf;
    }

    public void setLeaf(boolean leaf)
    {
        this.leaf = leaf;
    }

    //返回关键字的个数
    public int size()
    {
        return keys.size();
    }

    /**
     * 在节点中查找给定的key 返回SearchResult对象
     * 这是一个二分查找算法，可以保证时间复杂度为O(log(t))。
     *
     * @param key - 给定的键值
     * @return - 查找结果
     */
    public SearchResult searchKey(Integer key)
    {
        int l = 0;
        int h = keys.size() - 1;
        int mid = 0;
        while(l <= h)
        {
            mid = (l + h) / 2;
```

```
            if(keys.get(mid) == key)
                break;
            else if(keys.get(mid) > key)
                h = mid - 1;
            else
                l = mid + 1;
        }
        boolean result = false;
        int index = 0;
        if(l <= h) // 说明查找成功
        {
            result = true;
            index = mid; // index表示元素所在的位置
        }
        else
        {
            result = false;
            index = l; // index表示元素应该插入的位置
        }
        return new SearchResult(result, index);
    }

    /**
     * 将给定的key追加到节点的末尾，
     * 一定要确保调用该方法之后，节点中的关键字还是以非降序存放。
     * @param key - 给定的键值
     */
    public void addKey(Integer key)
    {
        keys.add(key);
    }

    /**
     * 删除给定索引的键值。
     * 你需要自己保证给定的索引是合法的。
     *
     * @param index - 给定的索引
     */
    public void removeKey(int index)
    {
        keys.remove(index);
    }

    /**
```

```
     * 得到节点中给定索引的键值。
     * 你需要自己保证给定的索引是合法的。
     *
     * @param index - 给定的索引
     * @return 节点中给定索引的键值
     */
    public Integer keyAt(int index)
    {
        return keys.get(index);
    }

    /**
     * 在该节点中插入给定的key，
     * 该方法保证插入之后，其键值还是以非降序存放。
     * 不过该方法的时间复杂度为O(t)。
     *
     * @param key - 给定的键值
     */
    public void insertKey(Integer key)
    {
        SearchResult result = searchKey(key);
        insertKey(key, result.getIndex());
    }

    /**
     * 在该节点中给定索引的位置插入给定的key，
     * 你需要自己保证key插入了正确的位置。
     *
     * @param key - 给定的键值
     * @param index - 给定的索引
     */
    public void insertKey(Integer key, int index)
    {
        List<Integer> newKeys = new ArrayList<Integer>();
        int i = 0;

        // index = 0或者index = keys.size()都没有问题
        for(; i < index; ++ i)
            newKeys.add(keys.get(i));
        newKeys.add(key);
        for(; i < keys.size(); ++ i)
            newKeys.add(keys.get(i));
        keys = newKeys;
    }
```

```
        /**
         * 返回节点中给定索引的子节点。
         * 你需要自己保证给定的索引是合法的。
         *
         * @param index - 给定的索引
         * @return 给定索引对应的子节点
         */
        public BTreeNode childAt(int index)
        {
            if(isLeaf())
                throw new UnsupportedOperationException("Leaf node doesn't have
children.");
            return children.get(index);
        }

        /**
         * 将给定的子节点追加到该节点的末尾。
         *
         * @param child - 给定的子节点
         */
        public void addChild(BTreeNode child)
        {
            children.add(child);
        }

        /**
         * 删除该节点中给定索引位置的子节点。
         * 你需要自己保证给定的索引是合法的。
         *
         * @param index - 给定的索引
         */
        public void removeChild(int index)
        {
            children.remove(index);
        }

        /**
         * 将给定的子节点插入到该节点中给定索引的位置。
         *
         * @param child - 给定的子节点
         * @param index - 子节点带插入的位置
         */
        public void insertChild(BTreeNode child, int index)
```

```
    {
        List<BTreeNode> newChildren = new ArrayList<BTreeNode>();
        int i = 0;
        for(; i < index; ++ i)
            newChildren.add(children.get(i));
        newChildren.add(child);
        for(; i < children.size(); ++ i)
            newChildren.add(children.get(i));
        children = newChildren;
    }
}

private static final int DEFAULT_T = 2;


private BTreeNode root;//B树的根节点
private int t = DEFAULT_T;  //B树的最小度数
private int minKeySize = t - 1; //非根节点中最小的关键值数
private int maxKeySize = 2*t - 1;   //非根节点中最大的关键值数

public BTree()
{
    root = new BTreeNode();
    root.setLeaf(true);
}

public BTree(int t)
{
    this();
    this.t = t;
    minKeySize = t - 1;
    maxKeySize = 2*t - 1;
}

/**
 * 搜索给定的key。
 * @param key - 给定的键值
 * @return
 */
public int search(Integer key)
{
    return search(root, key);
}
```

```
    /**
     * 在以给定节点为根的子树中，递归搜索给定的key
     *
     * @param node - 子树的根节点
     * @param key - 给定的键值
     * @return
     */
    private static int search(BTreeNode node, Integer key)
    {
        SearchResult result = node.searchKey(key);
        if(result.getResult())
            return result.getIndex();
        else
        {
            if(node.isLeaf())
                return -1;
            else
                search(node.childAt(result.getIndex()), key);

        }
        return -1;
    }

    /**
     * 分裂一个满子节点childNode。
     * 你需要自己保证给定的子节点是满节点。
     *
     * @param parentNode - 父节点
     * @param childNode - 满子节点
     * @param index - 满子节点在父节点中的索引
     */
    private void splitNode(BTreeNode parentNode, BTreeNode childNode, int
index)
    {

        BTreeNode siblingNode = new BTreeNode();
        siblingNode.setLeaf(childNode.isLeaf());
        // 将满子节点中索引为[t, 2t - 2]的(t - 1)个关键字插入新的节点中。
        for(int i = 0; i < minKeySize; ++ i)
            siblingNode.addKey(childNode.keyAt(t + i));
        // 提取满子节点中的中间关键字，其索引为(t - 1)
        Integer key = childNode.keyAt(t - 1);
        // 删除满子节点中索引为[t - 1, 2t - 2]的t个关键字
        for(int i = maxKeySize - 1; i >= t - 1; -- i)
```

```
            childNode.removeKey(i);
        if(!childNode.isLeaf()) // 如果满子节点不是叶节点，则还需要处理其子节点
        {
            // 将满子节点中索引为[t, 2t - 1]的t个子节点插入新的节点中
            for(int i = 0; i < minKeySize + 1; ++ i)
                siblingNode.addChild(childNode.childAt(t + i));
            // 删除满子节点中索引为[t, 2t - 1]的t个子节点
            for(int i = maxKeySize; i >= t; -- i)
                childNode.removeChild(i);
        }
        // 将key插入父节点
        parentNode.insertKey(key, index);
        // 将新节点插入父节点
        parentNode.insertChild(siblingNode, index + 1);
    }

    /**
     * 在一个非满节点中插入给定的key。
     *
     * @param node - 非满节点
     * @param key - 给定的键值
     */
    private void insertNotFull(BTreeNode node, Integer key)
    {
        assert node.size() < maxKeySize;

        if(node.isLeaf()) // 如果是叶子节点，直接插入
            node.insertKey(key);
        else
        {
            /* 找到key在给定节点应该插入的位置，那么key应该插入
             * 该位置对应的子树中
             */
            SearchResult result = node.searchKey(key);
            BTreeNode childNode = node.childAt(result.getIndex());
            if(childNode.size() == 2*t - 1) // 如果子节点是满节点
            {
                // 则先分裂
                splitNode(node, childNode, result.getIndex());
                /* 如果给定的key大于分裂之后新生成的键值，则需要插入该新键值的
右边，
                 * 否则左边。
                 */
                if(key > node.keyAt(result.getIndex()))
```

```
                childNode = node.childAt(result.getIndex() + 1);
            }
            insertNotFull(childNode, key);
        }
    }

    /**
     * 在B树中插入给定的key。
     *
     * @param key - 给定的键值
     */
    public void insert(Integer key)
    {
        if(root.size() == maxKeySize) // 如果根节点满了，则B树长高
        {
            BTreeNode newRoot = new BTreeNode();
            newRoot.setLeaf(false);
            newRoot.addChild(root);
            splitNode(newRoot, root, 0);
            root = newRoot;
        }
        insertNotFull(root, key);
    }

    /**
     * 从B树中删除一个给定的key。
     *
     * @param key - 给定的键值
     */
    public void delete(Integer key)
    {
        // root的情况还需要做一些特殊处理
        delete(root, key);
    }

    /**
     * 从以给定node为根的子树中删除指定的key。
     *
     * @param node - 给定的节点
     * @param key - 给定的键值
     */
    public void delete(BTreeNode node, Integer key)
    {
        // 该过程需要保证，对非根节点执行删除操作时，其关键字个数至少为t。
```

```
        assert node.size() >= t || node == root;

        SearchResult result = node.searchKey(key);
        /*
         * 因为这是查找成功的情况，0 <= result.getIndex() <= (node.size() -
1),
         * 因此(result.getIndex() + 1)不会溢出。
         */
        if(result.getResult())
        {
            // 1.如果关键字在节点node中，并且是叶节点，则直接删除。
            if(node.isLeaf())
                node.removeKey(result.getIndex());
            else
            {
                // 2.a 如果节点node中前于key的子节点包含至少t个关键字
                BTreeNode leftChildNode = node.childAt(result.getIndex());
                if(leftChildNode.size() >= t)
                {
                    // 使用leftChildNode中的最后一个键值代替node中的key
                    node.removeKey(result.getIndex());
                    node.insertKey(leftChildNode.keyAt(leftChildNode.size() -
1), result.getIndex());
                    delete(leftChildNode,
leftChildNode.keyAt(leftChildNode.size() - 1));
                    // node.
                }
                else
                {
                    // 2.b 如果节点node中后于key的子节点包含至少t个关键字
                    BTreeNode rightChildNode = node.childAt(result.getIndex() +
1);
                    if(rightChildNode.size() >= t)
                    {
                        // 使用rightChildNode中的第一个键值代替node中的key
                        node.removeKey(result.getIndex());
                        node.insertKey(rightChildNode.keyAt(0),
result.getIndex());
                        delete(rightChildNode, rightChildNode.keyAt(0));
                    }
                    else // 2.c 前于key和后于key的子节点都只包含t-1个关键字
                    {
                        node.removeKey(result.getIndex());
                        node.removeChild(result.getIndex() + 1);
```

```
                        // 将key和rightChildNode中的键值合并进leftChildNode
                        leftChildNode.addKey(key);
                        for(int i = 0; i < rightChildNode.size(); ++ i)
                            leftChildNode.addKey(rightChildNode.keyAt(i));
                        // 将rightChildNode中的子节点合并进leftChildNode，如果
有的话
                        if(!rightChildNode.isLeaf())
                        {
                            for(int i = 0; i <= rightChildNode.size(); ++ i)

leftChildNode.addChild(rightChildNode.childAt(i));
                        }
                        delete(leftChildNode, key);
                    }
                }
            }
        }
        else
        {
            /*
             * 因为这是查找失败的情况，0 <= result.getIndex() <= node.size(),
             * 因此(result.getIndex() + 1)会溢出。
             */
            if(node.isLeaf()) // 如果关键字不在节点node中，并且是叶节点，则什么
都不做，因为该关键字不在该B树中
            {
                return;
            }
            BTreeNode childNode = node.childAt(result.getIndex());
            if(childNode.size() >= t)
                delete(childNode, key); // 递归删除
            else // 3
            {
                // 先查找右边的兄弟节点
                BTreeNode siblingNode = null;
                int siblingIndex = -1;
                if(result.getIndex() < node.size()) // 存在右兄弟节点
                {
                    if(node.childAt(result.getIndex() + 1).size() >= t)
                    {
                        siblingNode = node.childAt(result.getIndex() + 1);
                        siblingIndex = result.getIndex() + 1;
                    }
                }
```

```
                // 如果右边的兄弟节点不符合条件，则试试左边的兄弟节点
                if(siblingNode == null)
                {
                    if(result.getIndex() > 0) // 存在左兄弟节点
                    {
                        if(node.childAt(result.getIndex() - 1).size() >= t)
                        {
                            siblingNode = node.childAt(result.getIndex() - 1);
                            siblingIndex = result.getIndex() - 1;
                        }
                    }
                }
                // 3.a 有一个相邻兄弟节点至少包含t个关键字
                if(siblingNode != null)
                {
                    if(siblingIndex < result.getIndex()) // 左兄弟节点满足条件
                    {
                        childNode.insertKey(node.keyAt(siblingIndex), 0);
                        node.removeKey(siblingIndex);
                        node.insertKey(siblingNode.keyAt(siblingNode.size() -
1), siblingIndex);
                        siblingNode.removeKey(siblingNode.size() - 1);
                        // 将左兄弟节点的最后一个孩子移到childNode
                        if(!siblingNode.isLeaf())
                        {

childNode.insertChild(siblingNode.childAt(siblingNode.size()), 0);
                            siblingNode.removeChild(siblingNode.size());
                        }
                    }
                    else // 右兄弟节点满足条件
                    {
                        childNode.insertKey(node.keyAt(result.getIndex()),
childNode.size() - 1);
                        node.removeKey(result.getIndex());
                        node.insertKey(siblingNode.keyAt(0),
result.getIndex());
                        siblingNode.removeKey(0);
                        // 将右兄弟节点的第一个孩子移到childNode
                        // childNode.insertChild(siblingNode.childAt(0),
childNode.size() + 1);
                        if(!siblingNode.isLeaf())
                        {
                            childNode.addChild(siblingNode.childAt(0));
```

```
                        siblingNode.removeChild(0);
                    }
                }
                delete(childNode, key);
            }
            else // 3.b 如果其相邻左右节点都包含t-1个关键字
            {
                if(result.getIndex() < node.size()) // 存在右兄弟
                {
                    BTreeNode rightSiblingNode =
node.childAt(result.getIndex() + 1);
                    childNode.addKey(node.keyAt(result.getIndex()));
                    node.removeKey(result.getIndex());
                    node.removeChild(result.getIndex() + 1);
                    for(int i = 0; i < rightSiblingNode.size(); ++ i)
                        childNode.addKey(rightSiblingNode.keyAt(i));
                    if(!rightSiblingNode.isLeaf())
                    {
                        for(int i = 0; i <= rightSiblingNode.size(); ++ i)

childNode.addChild(rightSiblingNode.childAt(i));
                    }
                }
                else // 存在左节点
                {
                    BTreeNode leftSiblingNode =
node.childAt(result.getIndex() - 1);
                    childNode.addKey(node.keyAt(result.getIndex() - 1));
                    node.removeKey(result.getIndex() - 1);
                    node.removeChild(result.getIndex() - 1);
                    for(int i = leftSiblingNode.size() - 1; i >= 0; -- i)
                        childNode.insertKey(leftSiblingNode.keyAt(i), 0);
                    if(!leftSiblingNode.isLeaf())
                    {
                        for(int i = leftSiblingNode.size(); i >= 0; -- i)

childNode.insertChild(leftSiblingNode.childAt(i), 0);
                    }
                }
                // 如果node是root并且node不包含任何关键字了
                if(node == root && node.size() == 0)
                    root = childNode;
                delete(childNode, key);
            }
```

```
            }
        }
    }

    /**
     * 一个简单的层次遍历B树实现，用于输出B树。
     */
    public void output()
    {
        Queue<BTreeNode> queue = new LinkedList<BTreeNode>();
        queue.offer(root);
        while(!queue.isEmpty())
        {
            BTreeNode node = queue.poll();
            for(int i = 0; i < node.size(); ++ i)
                System.out.print(node.keyAt(i) + " ");
            System.out.println();
            if(!node.isLeaf())
            {
                for(int i = 0; i <= node.size(); ++ i)
                    queue.offer(node.childAt(i));
            }
        }
    }

    public static void main(String[] args)
    {
        Random random = new Random();
        BTree<Integer, Byte[]> btree = new BTree<Integer, Byte[]>();
        for(int i = 0; i < 10; ++ i)
        {
            int r = random.nextInt(100);
            System.out.println(r);
            btree.insert(r);
        }
        System.out.println("----------------------");
        btree.output();
    }
}
```

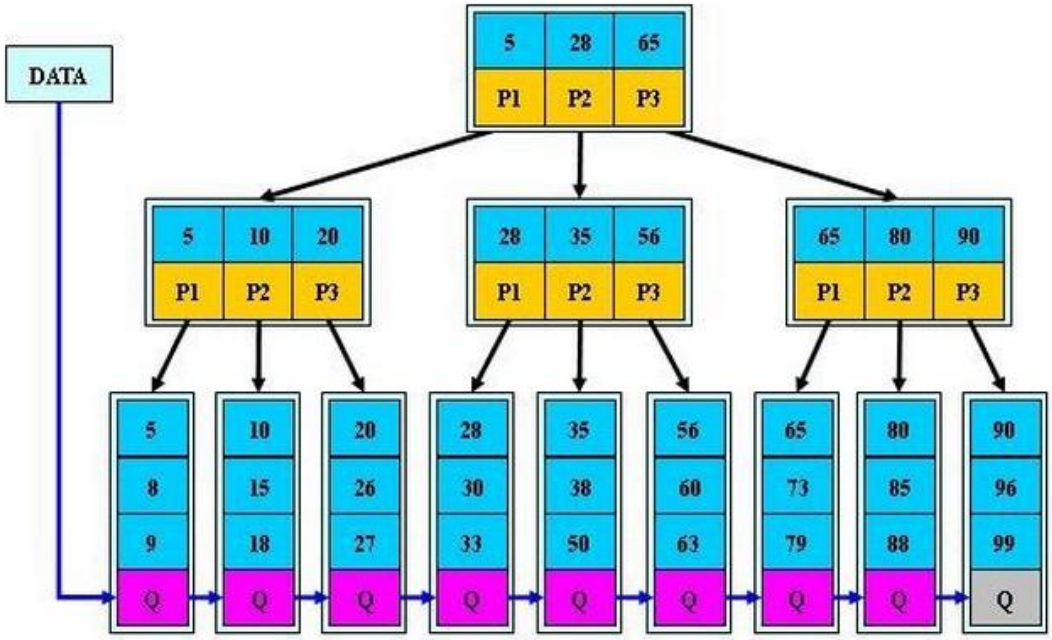
B 树的代码能看懂即可。

B+树

学习B+之前必须已经学习过前面的B树知识。

B+树是B树的变种，都是为了改善B树的效率而诞生的。

下图是一棵B+树：



先来看一下m阶B+树和m阶B树的不同之处：

1. 有n棵子树的节点含有n个关键字
2. 所有的叶节点中都包含了全部关键字的信息及指向含有这些关键字记录的指针，且叶节点本身按照关键字的大小自小而大的顺序链接。
3. 所有的非终端节点可以看做是索引部分。节点中仅含有其子树根节点中最大(或者最小)关键字，B树中的非终端节点也包含需要查找的有效信息。

为什么B+树比B树更适合在实际应用中操作系统的文件索引和数据库索引？

1. B+树的磁盘读写代价更低，因为B+树的内部节点并没有指向关键字具体信息的指针，所以其内部节点比B树更小。举个例子：假设磁盘中的一个盘块容纳16字节，而一个关键字2字节，一个关键字具体信息指针2字节，一棵9阶B树存储其内部节点需要两个盘块，而B+树只需要一个。
2. B+树的查询效率更加稳定。因为非终节点并不是最终指向文件内容的节点，而是叶节点中关键字的索引，所以任何关键字的查找都必须走一条从根节点到叶节点的路。所有关键字查询的路径长度相同，这样使得每一个数据查询的效率相当。

堆

什么是堆

堆是一种特殊的二叉树，其插入和删除最大(最小)数据项的速度均为 O(log(N))。
注意：这里的堆说的是一种数据结构，不要与 java 中的堆内存混淆。

堆的特点

1.它是完全二叉树，即除了树的最后一层不是满的，其他每一层从左到右都是满的。
2.它常常使用数组来实现
3.堆中每个节点都满足堆的条件，即每个节点的关键字都大于等于（小于等于）这个节点的子节点的关键字。(在下面的讲解中，假设我们需要的都是根节点最大的堆)

数组实现堆

在前面讲解二叉树的时候，我们实现二叉树使用的是链表，其实我们也是可以使用数组来实现的。怎么实现呢？

假设数组开始的索引是 0，那么对于任意一个节点(假设索引是 i)来说，它如果有父节点，那么父节点的索引就是(i-1)/2。它如果有子节点，子节点的索引就是 2*i+1,2*i+2。

弱序

堆和二叉搜索树相比是弱序的，二叉搜索树中可以按照一个简单的顺序遍历节点。堆中，按序遍历是困难的，因为各个节点之间没有什么明显的关系。

堆的移除操作

移除总是指移除关键字最大的节点，这个节点总是根节点(因为前面我们就说过，接下来讨论的都是最大堆)，就是索引为 0 的节点。但是一旦移除根节点，树就不是完全的了，数组中多了一个空白的单元，所以，删除的时候，我们需要保持一定的步骤，使得删除操作执行完成后二叉树的性质仍然能够得到满足。

三步走：

1. 移走根节点
2. 把最后一个节点移到根的位置
3. 一直向下筛选这个节点，直到它在一个大于它的节点之下，小于它的节点之上。

这个过程也被叫做向下筛选。

堆的插入操作

插入一个节点还是比较简单的。开始时把节点插入到数组中最后一个空着的单元中，而后向上筛选到合适位置，使得它在一个大于它的节点之下，在一个小于它的节点之上。

Java 实现堆

```
package com.algorithm.heap;

import java.util.Random;

/**
 * 堆的实现
 * @author yy
 *
 */
public class Heap {

    private Node[] heapArray; //存储堆节点的数组
    private int maxSize;  //堆节点的最大值
    private int currentSize;  //当前堆的大小

    public Heap(int mx) { //完成一些初始化操作
        maxSize = mx;
        currentSize = 0;
        heapArray = new Node[maxSize];
    }

    public boolean isEmpty() {    //判断堆是否为空
        return currentSize == 0;
    }

    //插入一个节点
    public boolean insert(int key) {
        if(currentSize == maxSize) return false;//已满 不能插入
        Node newNode = new Node(key);
        heapArray[currentSize] = newNode;    //先插入到最后
        trickleUp(currentSize++); //向上筛选
        return true;
    }

    //节点的替换    第j处的节点替换成数据项=value的新节点
    public void insertAt(int j, int value) {
        heapArray[j] = new Node(value);
        currentSize++;
    }

    //向上筛选
    private void trickleUp(int index) {
```

```
        int parent = (index-1)/2; //先找到父节点索引
        Node bottom = heapArray[index];
        while(index >0 && heapArray[parent].getKey() < bottom.getKey())
{
            //只要index>0或者父节点的数据项的值小于自己的值 就继续向上
            heapArray[index] = heapArray[parent];
            index = parent;
            parent = (index-1)/2;
        }
        heapArray[index] = bottom;
    }

    //删除一个最大节点(根)
    public Node remove() {
        Node root = heapArray[0];//移除根节点
        heapArray[0] = heapArray[--currentSize];//最后一个节点移动到根节点
的位置
        trickleDown(0);    //向下筛选
        return root;
    }

    //向下筛选
    private void trickleDown(int index) {
        int largeChild;
        Node top = heapArray[index];
        while(index < currentSize/2) {   //只要不是叶节点  就继续向下筛选
            int left = index*2+1;
            int right = index*2+2;
            if(right < currentSize && heapArray[left].getKey() <
heapArray[right].getKey()) {
                largeChild = right;
            }else {
                largeChild = left;
            }
            if(top.getKey() >= heapArray[largeChild].getKey()) break;
            else {
                heapArray[index] = heapArray[largeChild];
                index  = largeChild;
            }
        }
        heapArray[index] = top;
    }
```

```
    /**
     * 测试堆的使用是否正常
     * @param args
     */
    public static void main(String[] args) {
        Heap heap = new Heap(10);
        Random random = new Random();
        for (int i = 0; i < 10; i++) {
            heap.insertAt(i, random.nextInt(100));
        }
        for (int i = heap.currentSize/2-1; i >= 0; i--) {
            heap.trickleDown(i);
        }
        for (int i = 0; i < 10; i++) {
            System.out.println(heap.remove().getKey());
        }
    }

}
```

图

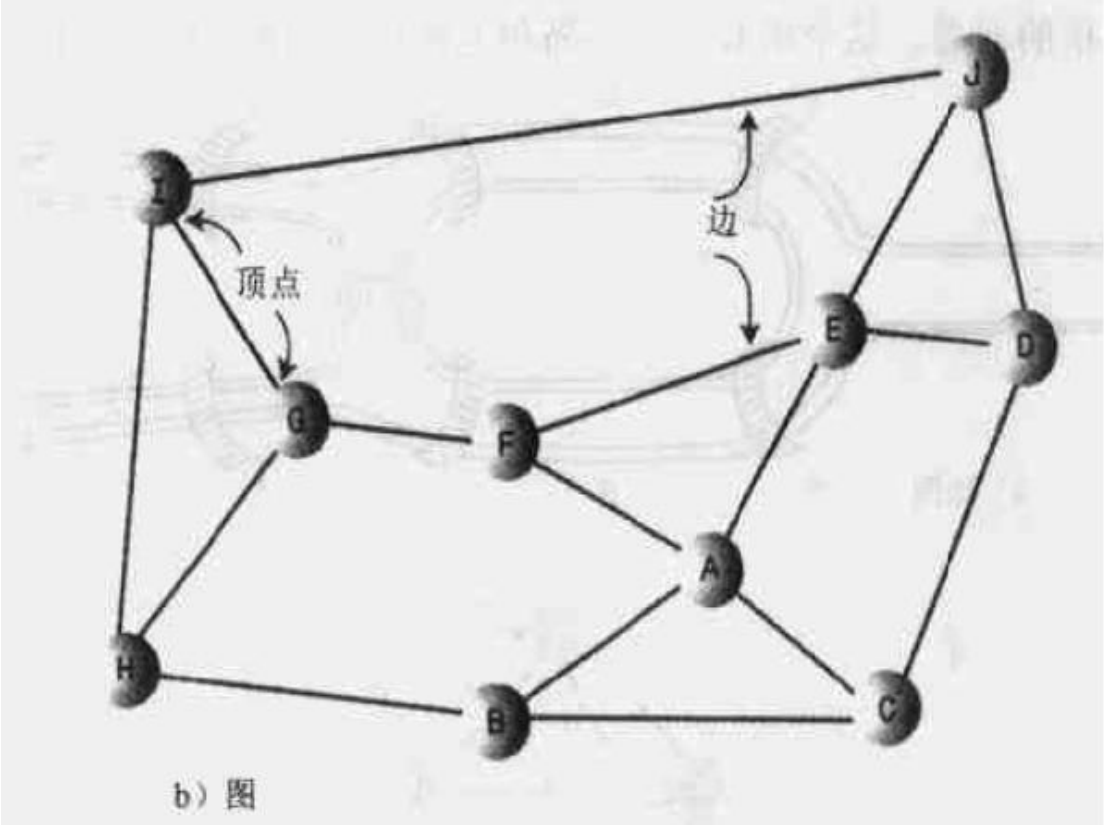
简介

图是一种和树比较像的数据结构，数学意义上来说，树是图的一种。

图通常有一种固定的形状，这是由物理或者抽象的问题所决定的，例如，图中节点可能表示城市，而边表示城市间的航班路线。另外一个更加抽象的例子是一个代表了几个单独任务的图，这些任务是完成一个项目所必须的。总之，图的形状取决于真实世界的具体情况。

定义

图中，基本构成元素有两个：顶点和连接两个顶点之间的边。



b）图

邻接

如果两个顶点被同一条边连接，就称这两个顶点是邻接的。如上图 I 和 G 是邻接的，I 和 F 就不是邻接的。

路径

路径是边的序列。如 B 到 J 可以通过 A 和 E 到达，那么 BAEJ 就是 B 到 J 的一条路径。

连通图

如果至少有一条边可以连接起所有的顶点，那么这个图被称为连通的。

有向图

无向图中 A 到 B 和 B 到 A 是等价的，而有向图中 A 到 B 和 B 到 A 是不等价的，因为它们方向不同。

带权图

有些图中，边被赋予一个权值，权值是一个数字，它能代表两个顶点之间的物理距离，或者一个顶点到另外一个顶点的时间，或者是两点间的距离。这样的图叫做带权图。

在程序中表示图

顶点

顶点大多数情况下表示某个真实世界的对象，这个对象必须使用数据项来描述，它可以代表城市、飞机……在这里我们简单的让它代表一个字符串。除了数据项，顶点类还需要一个布尔类型的标记，这个标记在后面的搜索算法中会介绍。

Vertex 类：

```
/**
 * 顶点类
 * @author yy
 *
 */
class Vertex{
    public String label;  //顶点存储的数据
    public boolean wasVisited;    //搜索使用  用来标记

    public Vertex(String ch) {
        label = ch;
        wasVisited = false;
    }
}
```

边

一般来说，表示图中的边我们有两种方法：邻接矩阵和邻接表。

邻接矩阵

邻接矩阵是一个二维数组，数据项表示两个顶点之间是否存在边，如果图中有 n 个顶

点，那么；邻接矩阵就是 n*n 的数组。

表 13.1　邻接矩阵

	A	B	C	D
A	0	1	1	1
B	1	0	0	1
C	1	0	0	0
D	1	1	0	0

上图就是一个邻接矩阵，上下三角是对称的，这就要求我们在修改的时候我们必须同时修改两处。

邻接表

邻接表是一个链表数组，每个单独的链表表示了有哪些顶点与当前节点邻接。如下图：

表 13.2　邻接表

顶点	包含邻接顶点的链表
A	B—>C—>D
B	A—>D
C	A
D	A—>B

插入一条边

假设要在 1 和 3 之间加一条边。

如果是在邻接矩阵中，只需要把邻接矩阵数组的[0][2]和[2][0]都设置为 1.

如果是在邻接表中，只需要在 1 的链表中加入 3,3 的链表中加入 1。

搜索

假设现在已经创建了一个图，现在需要一个算法来提供系统的方法，从某个特定顶点开始，沿右边移动到其他顶点，移动完毕后，要保证访问了和起始点相连的每一个顶点。

有两种常用的方式可以来搜索图，深度优先搜索和广度优先搜索。前者使用栈来完成，后者使用队列来完成。

深度优先

为了实现深度优先搜索，找一个起始点，需要做三件事：首先访问该顶点，然后把该顶点放入栈中，以便记住它，最后标记它，这样就不会再访问它。接下来需要遵循以下规则：

1.如果可能，访问一个和栈顶顶点相邻接的未访问节点，标记它，并把它放入栈中

2.当不能执行规则 1 时，如果栈不为空，就从栈中弹出一个顶点，继续 1 过程

3.如果不能执行 1 和 2，那么就完成了深度优先搜索的过程

广度优先

深度优先搜索在搜索的过程中看起来像是在尽可能的远离点初始点，而在广度优先搜索过程中，要尽可能的靠近初始点。

过程如下：

首先找一个起始点，并且访问它，标记为当前顶点，然后应用以下的规则：

1.访问下一个未访问的邻接顶点(如果存在)，这个顶点必须是当前顶点的邻接点，标记它，并把它插入到队列中。

2.如果因为没有未访问顶点而不能执行规则 1，那么从队列中取一个顶点(如果存在)，并使其成为当前顶点。

3.如果因为队列为空而不能执行规则 2，则搜索结束。

图的代码(含搜索操作):

```
package com.algorithm.graph;

import java.util.ArrayDeque;
import java.util.Deque;
import java.util.Queue;
import java.util.Stack;

/**
 * 无向图的实现 这里只用邻接矩阵的方式
 * @author yy
 *
 */
public class Graph {

    private final int MAX_VERTS = 20;    //最大顶点数
    private Vertex[] vertexList; //存放顶点的数组
    public int[][] adjMat;    //邻接矩阵
```

```
public int nVerts;//当前顶点的数量
private Stack theStack;   //栈  用作dfs
private Deque theQueue;   //队列 用作bfs

/**
 * 进行一些初始化操作
 */
public Graph() {
    vertexList = new Vertex[MAX_VERTS];
    adjMat = new int[MAX_VERTS][MAX_VERTS];
    nVerts = 0;
    //adjMax初始化
    for (int i = 0; i < MAX_VERTS; i++) {
        for (int j = 0; j < MAX_VERTS; j++) {
            adjMat[i][j] = Integer.MAX_VALUE;
        }
    }
    theStack = new Stack();
    theQueue = new ArrayDeque();
}


//添加一个顶点
public void addVertex(String ch) {
    vertexList[nVerts++] = new Vertex(ch);
}

//添加一条边
public void addEdge(int start, int end) {
    adjMat[start][end] = 1;
    adjMat[end][start] = 1;
}

//打印某个特定的节点
public void displayVertex(int v) {
    System.out.println(vertexList[v].label);
}

//深度优先搜索
public void dfs() {
    vertexList[0].wasVisited = true;
    displayVertex(0);
    theStack.push(0);
```

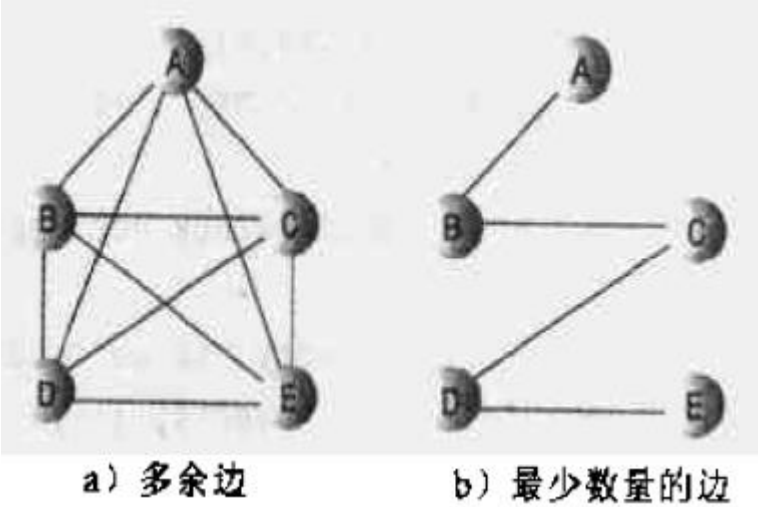
```
    while(!theStack.isEmpty()) {
        int v = getAdjUnvisitedVertex((int) theStack.peek());
        if(v == -1) {
            theStack.pop();
        }
        else {
            vertexList[v].wasVisited = true;
            displayVertex(v);
            theStack.push(v);
        }
    }

    //遍历结束
    for (int i = 0; i < nVerts; i++) {
        vertexList[i].wasVisited = false;
    }

}


//得到v所能到达的第一个未访问的节点
public int getAdjUnvisitedVertex(int v) {
    for (int i = 0; i < nVerts; i++) {
        if(adjMat[v][i] == 1 && vertexList[i].wasVisited == false) {
            return i;
        }
    }
    return -1;
}

//实现广度优先搜索
public void bfs() {
    vertexList[0].wasVisited = true;
    displayVertex(0);
    theQueue.addFirst(0);
    int v2;

    while(!theQueue.isEmpty()) {
        int v1 = (int) theQueue.removeFirst();
        while((v2 = getAdjUnvisitedVertex(v1)) != -1) {
            vertexList[v2].wasVisited = true;
            displayVertex(v2);
            theQueue.addFirst(v2);
        }
```

```
        }
        for (int i = 0; i < nVerts; i++) {
            vertexList[i].wasVisited = false;
        }
    }


}
/**
 * 顶点类
 * @author yy
 *
 */
class Vertex{
    public String label;  //顶点存储的数据
    public boolean wasVisited;    //搜索使用  用来标记

    public Vertex(String ch) {
        label = ch;
        wasVisited = false;
    }
}
```

最小生成树

定义

最小生成树就是一种算法，这种算法可以去掉图中多余的连线，用最少的边连接所有的顶点。



a）多余边　　b）最少数量的边

上图中，左面的图就有多余的边，右面的没有多余的边，所以我们可以认为在左图的基

础上运行最小生成树算法就可以得到右图。

最小生成树中边的数量总是比顶点的数量少一。在非带权图中，我们只需要考虑寻找最少的边，而不需要考虑边的长度。

求解

我们使用 DFS 算法就可以得到最小生成树，因为 DFS 访问所有的顶点，但是只访问一次，它决不会两次访问同一个顶点，当它看到某条边将到达一个已访问的顶点，它就不会走这条边，它从来不遍历那些不可能的边。

注意，我们运行 DFS 算法得到的最小生成树仅仅是多种允许结果中的一种。

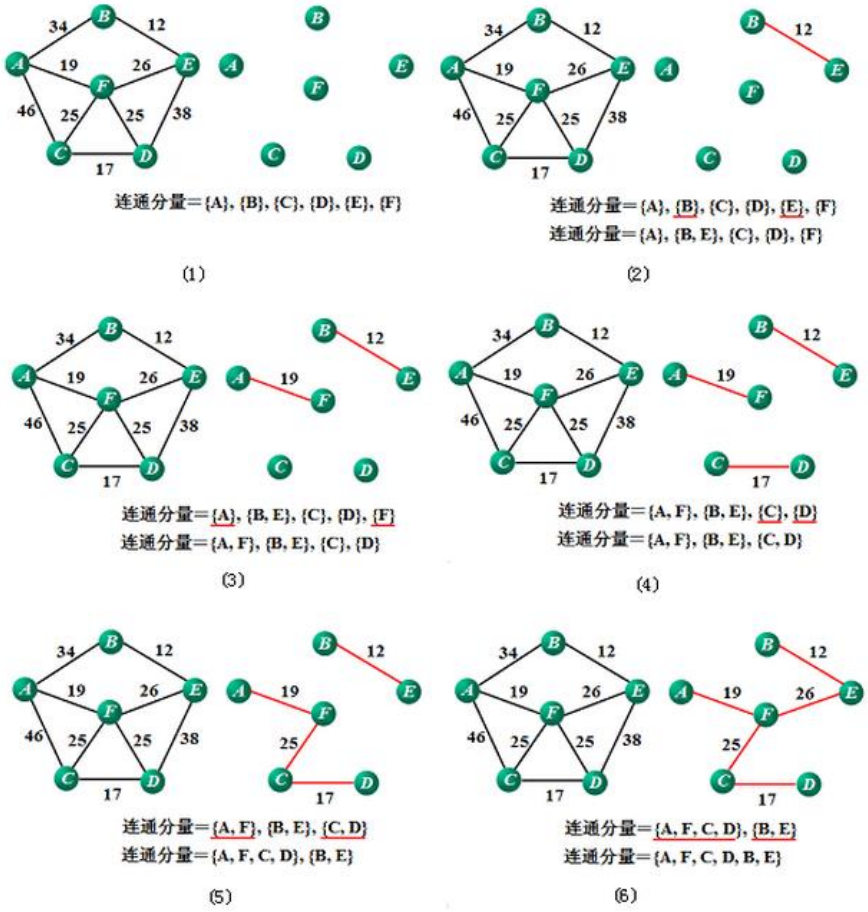
带权图的最小生成树

带权图的最小生成树算法和不带权图的相比要复杂一些，因为要考虑权重，我们不但要得到一个最小边组成的图，而且还有保证边上的权重之和是最小的。一般来说，带权图的最小生成树算法有两种：Kruskal 算法和 Prim 算法。

Kriskal 算法

设无向连通图为 G＝(V, E)，令 G 的最小生成树为 T＝(U, TE)，其初态为 U＝V，TE＝{ }，然后，按照边的权值由小到大的顺序，考察 G 的边集 E 中的各条边。若被考察的边的两个顶点属于 T 的两个不同的连通分量，则将此边作为最小生成树的边加入到 T 中，同时把两个连通分量连接为一个连通分量；若被考察边的两个顶点属于同一个连通分量，则舍去此边，以免造成回路，如此下去，当 T 中的连通分量个数为 1 时，此连通分量便为 G 的一棵最小生成树。

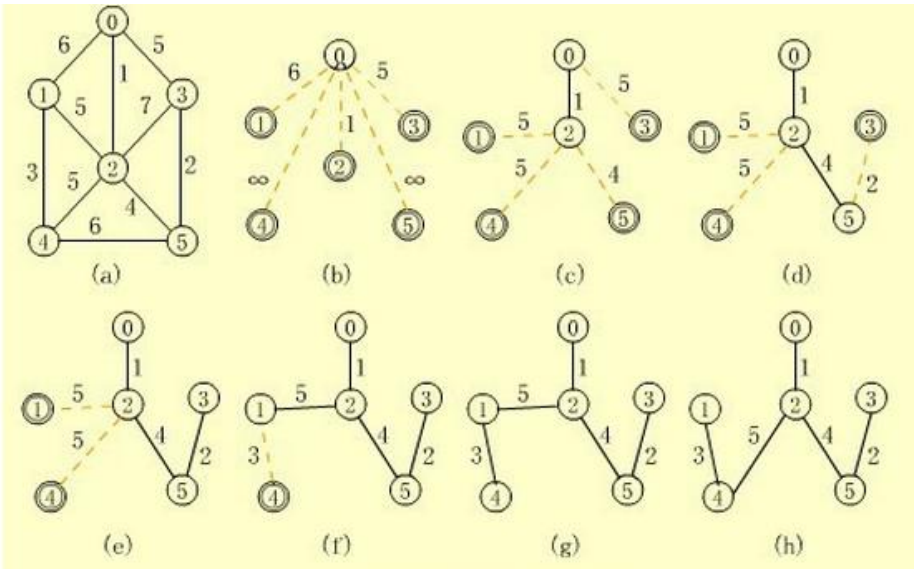
如下图，就是一个构建过程的展示：



Prim 算法

从指定顶点开始将它加入集合中,然后将集合内的顶点与集合外的顶点所构成的所有边中选取权值最小的一条边作为生成树的边,并将集合外的那个顶点加入到集合中,表示该顶点已连通.再用集合内的顶点与集合外的顶点构成的边中找最小的边,并将相应的顶点加入集合中,如此下去直到全部顶点都加入到集合中,即得最小生成树.

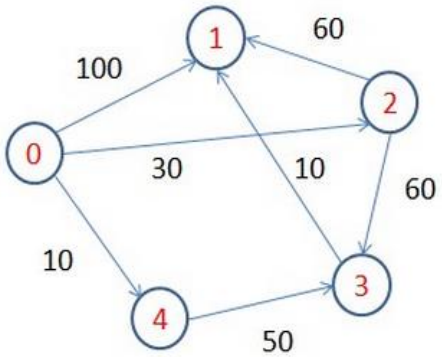
如下图:



最短路径

单源最短路径

问题引入



上图中，假设顶点代表的是各个公交车站。如果 2、1 顶点之间有边，那么就代表 2 乘坐公交车可以到达 1。边上的权值就代表公交车从 2 到达 1 需要的时间。现在要求从某个顶点出发到达其他各个顶点需要的最短时间(如果可达的话)，并且给出具体的路径。这就是单源最短路径问题。

最短路径的最优子结构性质

该性质描述为：如果 P(i, j)={Vi....Vk..Vs...Vj}是从顶点 i 到 j 的最短路径，k 和 s

是这条路径上的一个中间顶点，那么 P(k,s)必定是从 k 到 s 的最短路径。下面证明该性质的正确性。

假设 P(i,j)={Vi....Vk..Vs...Vj}是从顶点 i 到 j 的最短路径，则有 P(i,j)=P(i,k)+P(k,s)+P(s,j)。而 P(k,s)不是从 k 到 s 的最短距离，那么必定存在另一条从 k 到 s 的最短路径 P'(k,s)，那么 P'(i,j)=P(i,k)+P'(k,s)+P(s,j)<P(i,j)。则与 P(i,j)是从 i 到 j 的最短路径相矛盾。因此该性质得证。

Dijkstra 算法

由上述性质可知，如果存在一条从 i 到 j 的最短路径(Vi.....Vk,Vj)，Vk 是 Vj 前面的一顶点。那么(Vi...Vk)也必定是从 i 到 k 的最短路径。为了求出最短路径，Dijkstra 就提出了以最短路径长度递增，逐次生成最短路径的算法。譬如对于源顶点 V0，首先选择其直接相邻的顶点中长度最短的顶点 Vi，那么当前已知可得从 V0 到达 Vj 顶点的最短距离 D[j]=min{D[j],D[i]+matrix[i][j]}。根据这种思路，

假设存在 G=<V,E>，源顶点为 V0，U={V0},D[i]记录 V0 到 i 的最短距离，prve[i]记录从 V0 到 i 路径上的 i 前面的一个顶点。

1.从 V-U 中选择使 D[i]值最小的顶点 i，将 i 加入到 U 中；
2.更新与 i 直接相邻顶点的 D 值。(D[j]=min{D[j],D[i]+matrix[i][j]})
3.直到 U=V，停止。

此算法的时间复杂度是 O(N^2)。

Java 实现 Dijkstra

```
package com.algorithm.graph;

/**
 * 单源最短路径问题：迪杰斯特拉算法的实现 问题描述：求从v0出发到达其他各个顶点的最短路径
 *
 * @author yy
 *
 */
public class Dijkstra {

    /**
     * max=Integer.MAX_INT 表示两个顶点之间无法到达
     * D数组 来模拟向量D 存储最短路径长度的数组
     * prve[i]记录从V0到i路径上的i前面的一个顶点。
```

```
     * a 使用邻接矩阵的方式存储图信息
     */
    private static int max = Integer.MAX_VALUE;
    private static int D[] = new int[6];
    private static int prve[] = new int[6];
    private static int a[][] = { { 0, max, 10, max, 30, 100 },
            { max, 0, 5, max, max, max }, { max, max, 0, 50, max, max },
            { max, max, max, 0, max, 10 }, { max, max, max, 20, 0, 60 },
            { max, max, max, max, max, 0 } };

    /**
     * 迪杰斯特拉算法
     * @param v开始出发的节点
     * @param a邻接矩阵
     * @param D向量D
     * @param prve 保存各个顶点的前驱节点
     */
    public void dijkstra(int v, int[][] a, int D[], int prve[]) {

        int n = D.length-1;
        boolean[] s = new boolean[n+1];  //保存已经找到最短路径的节点 true
标示为已找到
        for (int i = 1; i <= n; i++) {
            D[i] = a[v][i];

            //prve[]数组存储源点到顶点vi之间的最短路径上的该节点的前驱结点
            if(D[i] < Integer.MAX_VALUE) {
                prve[i] = v;
            }else {
                prve[i] = -1;
            }
        }

        D[v] = 0;
        s[v] = true;   //初始化v节点已经找到最短路径
        for (int i = 1; i <= n; i++) {
            int u = v;
            int temp = Integer.MAX_VALUE;
            for (int j = 1; j <= n; j++) {
                if(!s[j] && (D[j]<temp)) {
                    u = j;
                    temp = D[j];
                }
            }
```

```
            //u顶点进s集合
            s[u] = true;
            //更新当前的最短路径以及长度
            for (int j = 0; j <= n; j++) {
                if(!s[j] && (a[u][j] < Integer.MAX_VALUE)) {    //当前顶点
不属于S集合 并且当前节点有前驱结点u
                    int newDist = D[u] + a[u][j];
                    if(newDist < D[j]) {
                        D[j] = newDist;    //更新后的最短路径
                        prve[j] = u;   //当前节点加入前驱节点集
                    }
                }
            }

        }

    }

    /**
     * 输出结果
     * @param m源点
     * @param p更新后的前驱节点集合
     * @param d向量D
     */
    public void outPath(int m, int[] p, int[] d) {

        for (int i = 0; i < d.length; i++) {
            if(d[i] < Integer.MAX_VALUE && (i != m)) {
                System.out.print("v" + i + "<--");
                int next = p[i]; // 设置当前顶点的前驱顶点
                while (next != m) { // 若前驱顶点不为一个，循环求得剩余前驱
顶点
                    System.out.print("v" + next + "<--");
                    next = p[next];
                }
                System.out.println("v" + m + ":" + d[i]);
            }else if(i!=m) {
                System.out.println("v" + i + "<--" + "v" + m + ":no
path");
            }
        }

    }
```

```
    public static void main(String[] args) {
        Dijkstra dijkstra = new Dijkstra();
        dijkstra.dijkstra(0, a, D, prve);
        dijkstra.outPath(0, prve, D);
    }

}
```

多源最短路径

问题描述

求两两顶点之间的最短路径问题。

解决思路

1. 对每个节点执行一次迪杰斯特拉算法 O(N^3)
2. 弗洛伊德算法(Floyd) 复杂度和上面相同 但形式上更简单、更快

Floyd 算法

若从 Vi 到 Vj 有弧，则从 Vi 到 Vj 存在一条长度为弧上权值（arcs[i][j] ）的路径，该路径不一定是最短路径，尚需进行 n 次试探。首先考虑从 Vi 到 Vj 经过中间顶点 V0 的路径（Vi，V0，Vj）是否存在，也就是判断弧（Vi，V0）和（V0，Vj）是否存在。若存在，则比较（Vi，Vj）和（Vi，V0，Vj）的路径长度取较短的为从 Vi 到 Vj 的中间顶点序号不大于 0 的最短路径。

在此路径上再增加一个顶点 V1，也就是说，如果（Vi，…V1）和（V1，…Vj）分别是当前找到的中间顶点序号不大于 0 的最短路径，那么，（Vi，…V1，…Vj）就有可能是从 Vi 到 Vj 的中间顶点序号不大于 1 的最短路径。将它和已经得到的从 Vi 到 Vj 中间顶点序号不大于 0 的最短路径相比较，从中选出最短的作为从 Vi 到 Vj 中间顶点序号不大于 1 的最短路径。然后，再增加一个顶点 V2 继续进行这个试探过程。

一般情况下，若（Vi，…Vk）和（Vk，…Vj）分别是从 Vi 到 Vk 和从 Vk 到 Vj 的中间顶点序号不大于 k-1 的最短路径，则将（Vi，…，Vk，…Vj）和已经得到的从 Vi 到 Vj 的中间顶点序号不大于 k-1 的最短路径相比较，其长度最短者即为从 Vi 到 Vj 的中间顶点序号不大于 k 的最短路径。

经过 n 次比较之后，最后求得的便是从 Vi 到 Vj 的最短路径。

Java 实现 Floyd

Floyd 是基于动态规划实现的。

```
package com.algorithm.graph;

/**
 * 弗洛伊德算法 解决全源最短路径问题
 *
 * @author yy
 *
 */
public class Floyd {

    /**
     * max代表无穷大得数 a代表图的邻接矩阵 D[i][j] 表示的是从vi节点到vj节点的最短路径
     */
    private static int max = Integer.MAX_VALUE;
    private static int a[][] = { { 0, max, 10, max, 30, 100 },
            { max, 0, 5, max, max, max }, { max, max, 0, 50, max, max },
            { max, max, max, 0, max, 10 }, { max, max, max, 20, 0, 60 },
            { max, max, max, max, max, 0 } };
    private static int[][] D = new int[6][6];

    /**
     * 弗洛伊德算法
     * @param a
     *            邻接矩阵
     * @param D
     *            二维向量
     * @param p
     *            用来表示路径
     */
    public void floyd(int[][] a1, int[][] D1) {

        for (int i = 0; i < D1.length; i++) {
            for (int j = 0; j < D1.length; j++) {
                D1[i][j] = a1[i][j];
            }
        }

        for (int i = 0; i < D1.length; i++) {
```

```
            for (int j = 0; j < D1.length; j++) {
                for (int j2 = 0; j2 < D1.length; j2++) {
                        if(D1[j][i]==Integer.MAX_VALUE || D1[i][j2] ==
Integer.MAX_VALUE) continue;
                        D1[j][j2] = Math.min(D1[j][j2], D1[j][i] +
D1[i][j2]);
                }
            }
        }
    }

    public void printgraph(int[][] G, int v) {
        for (int i = 0; i < v; i++) {
            for (int j = 0; j < v; j++) {
                if (G[i][j] == Integer.MAX_VALUE)
                    System.out.print("*   ");
                else
                    System.out.print(G[i][j] + "  ");
            }
            System.out.println();
        }
    }

    public static void main(String[] args) {
        Floyd floyd = new Floyd();
        floyd.floyd(a, D);
        floyd.printgraph(D, 6);
    }

}
```

动态规划

原理

定义

动态规划一般来求解最优化问题，其适用的条件是要求待求解的最优化问题具备两个因素：最优子结构和子问题重叠。通过求解一个个子问题，将解存入一张表中，当后续子问题

用到之前子问题的解时直接查表，每次查表的时间为常数时间。

示例

例如，现在要找一条从起点 A 到 B 的最短路径，那么在寻找 A 到 B 的最短路径时，可能会经过 x1、x2、x3 等几个点，然后从中选择出一条最短的路径即可。动态规划求解最短路径的一般思路是：先枚举从 A 到 B 可能要经过的所有路径，然后比较各种路径的长度，找出最短的路径。

求解过程

动态规划解决问题的关键步骤：

1.状态的定义

2.方案的枚举，没必要枚举全部，以前我们我们可能习惯全部枚举，但是当我们把具有某些性质的解作为一个集合一起考虑时，这个集合对应多个解，而在一个集合中只有最优解才有意义，因此要减少状态的个数。

3.最优子问题和无后效性保证。一个状态下的最优值取决于到达这个状态的全部子状态的最优解，且状态之间的最优解不互相影响。

4.子问题有重叠。

5.最为关键的是编写状态转移方程。说明后面状态和前面状态的关系。

下面会通过几个具体的问题来学习一下动态规划。

DP 实战

最大连续乘积子数组

问题描述

给定一个浮点数数组，任意取出数组中的若干个连续的数相乘，请找出其中乘积最大的子数组。例如，给定数组{-2.5,4,0,3,0.5,8,-1}，则取出的最大乘积子数组为{3,0.5,8}。也就是说，在上述数组中，3、0.5、8 这三个数是连续的，而且乘积是最大的。

分析与解法

蛮力轮询

最简单的思路就是采用两个 for 循环，但是时间复杂度是 O(N^2)，效率奇差。
代码如下：

```
/**
 * 蛮力轮询方式
 * @param a数组
 * @param length   数组长度
 * @return 返回最大值
 */
public static double MaxProductSubString(double[] a, int length) {

    double maxResult = a[0];
    for (int i = 0; i < length; i++) {
        double x = 1;
        for (int j = i; j < length; j++) {
            x *= a[j];
            if(x > maxResult) {
                maxResult = x;
            }
        }
    }

    return maxResult;

}
```

测试代码：

```
/**
 * 测试
 * @param args
 */
public static void main(String[] args) {
    double[] a = {-2.5,4,0,3,0.5,8,-1};
    System.out.println(MaxProductSubString(a, a.length));
}
```

动态规划

动态规划的思路中最关键的一步就是状态转移方程，它代表的是未来的状态和之前的子

状态之间的关系。

乘积子数组中可能有正数、负数、也可能有 0，因为有负数的存在，我们可以考虑同时找出最大乘积和最小乘积，于是问题简化成这样，在数组中找到这样一个子数组，使得它的乘积最大，同时再找到另一个子数组，使得它的乘积最小。也就是说，既要记录最大乘积，也要记录最小乘积。

假设数组是 a，直接利用动态规划来求解，考虑到负数的情况，用 maxend 表示以 a[i]结尾的最大的连续子数组的乘积值，用 minend 表示以 a[i]为结尾的最小的连续子数组的乘积值，那么状态转移方程为：

maxend = max(max(maxend*a[i],minend*a[i]),a[i]);

minend = min(min(maxend*a[i],minend*a[i]),a[i]);

初始状态就是 maxend=minend=a[0]。

下面谈一下个人对于这个方程的理解：这里以 maxend 为例，minend 和它类似，类比着理解即可。假设现在 maxend 表示以 a[i]结尾的最大的连续子数组的乘积值，那么下一步我们要求的就是以 a[i+1] 结尾的最大的连续子数组的乘积值，在求解的过程中，我们要基于 maxend 来求，它可能有如下情况：

1.如果 a[i+1]是负数，那么最大值就是 max(maxend，minend*a[i+1])；因为是负数的话就要乘以最小的乘积，然后和 maxend 比较，针对每一步，都有乘与不乘两种可能。

2.如果 a[i+1]是正数，那么最大值就是 max(maxend，maxend*a[i+1])；因为是正数的话就要乘以最大的乘积，然后和 maxend 比较，针对每一步，都有乘与不乘两种可能。

代码如下：

```
/**
 * 动态规划的方式
 * @param a数组
 * @param length   数组长度
 * @return 返回最大值
 */
public static double MaxProductSubString(double[] a, int length) {

    double maxEnd = a[0];
    double minEnd = a[0];
    double maxResult = a[0];
    for (int i = 1; i < length; i++) {
        double end1 = maxEnd*a[i], end2 = minEnd*a[i];
        maxEnd = Math.max(Math.max(end1, end2), a[i]);
        minEnd = Math.min(Math.min(end1, end2), a[i]);
        maxResult = Math.max(maxResult, maxEnd);
    }
    return maxResult;

}
```

背包问题

背包问题扩展起来的话有许多，这里只介绍 01 背包和完全背包，加深大家对于 DP 的理解和使用。

01 背包

问题引入

有 N 件物品和一个容量为 V 的背包。第 i 件物品的费用是 c[i]，价值是 w[i]。求解将哪些物品装入背包可使价值总和最大。

基本思路

这是最基础的背包问题，特点是：每种物品仅有一件，可以选择放或不放。

用子问题定义状态：即 f[i][v]表示前 i 件物品恰放入一个容量为 v 的背包可以获得的最大价值。则其状态转移方程便是：

f[i][v]=max{f[i-1][v],f[i-1][v-c[i]]+w[i]}

这个方程非常重要，基本上所有跟背包相关的问题的方程都是由它衍生出来的。所以有必要将它详细解释一下：“将前 i 件物品放入容量为 v 的背包中”这个子问题，若只考虑第 i 件物品的策略（放或不放），那么就可以转化为一个只牵扯前 i-1 件物品的问题。如果不放第 i 件物品，那么问题就转化为“前 i-1 件物品放入容量为 v 的背包中”，价值为 f[i-1][v]；如果放第 i 件物品，那么问题就转化为“前 i-1 件物品放入剩下的容量为 v-c[i]的背包中”，此时能获得的最大价值就是 f[i-1][v-c[i]]再加上通过放入第 i 件物品获得的价值 w[i]。

优化空间复杂度

以上方法的时间和空间复杂度均为 O(N*V)，其中时间复杂度基本已经不能再优化了，但空间复杂度却可以优化到 O(V)。

先考虑上面讲的基本思路如何实现，肯定是有一个主循环 i=1..N，每次算出来二维数组 f[i][0..V]的所有值。那么，如果只用一个数组 f[0..V]，能不能保证第 i 次循环结束后 f[v]中表示的就是我们定义的状态 f[i][v]呢？f[i][v]是由 f[i-1][v]和 f[i-1][v-c[i]]两个子问题递推而来，能否保证在推 f[i][v]时（也即在第 i 次主循环中推 f[v]时）能够得到 f[i-1][v]和 f[i-1][v-c[i]]的值呢？事实上，这要求在每次主循环中我们以

v=V..0 的顺序推 f[v]，这样才能保证推 f[v]时 f[v-c[i]]保存的是状态 f[i-1][v-c[i]]的值。伪代码如下：

```
for i=1..N
    for v=V..0
        f[v]=max{f[v],f[v-c[i]]+w[i]};
```

其中的 f[v]=max{f[v],f[v-c[i]]}一句恰就相当于我们的转移方程 f[i][v]=max{f[i-1][v],f[i-1][v-c[i]]}，因为现在的 f[v-c[i]]就相当于原来的 f[i-1][v-c[i]]。

事实上，使用一维数组解 01 背包的程序在后面会被多次用到，所以这里抽象出一个处理一件 01 背包中的物品过程，以后的代码中直接调用不加说明。

过程 ZeroOnePack，表示处理一件 01 背包中的物品，两个参数 cost、weight 分别表明这件物品的费用和价值。

```
procedure ZeroOnePack(cost,weight)
    for v=V..cost
        f[v]=max{f[v],f[v-cost]+weight}
```

注意这个过程里的处理与前面给出的伪代码有所不同。前面的示例程序写成 v=V..0 是为了在程序中体现每个状态都按照方程求解了，避免不必要的思维复杂度。而这里既然已经抽象成看作黑箱的过程了，就可以加入优化。费用为 cost 的物品不会影响状态 f[0..cost-1]，这是显然的。

有了这个过程以后，01 背包问题的伪代码就可以这样写：

```
for i=1..N
    ZeroOnePack(c[i],w[i]);
```

初始化的一些细节

我们看到的求最优解的背包问题题目中，事实上有两种不太相同的问法。有的题目要求“恰好装满背包”时的最优解，有的题目则并没有要求必须把背包装满。一种区别这两种问法的实现方法是在初始化的时候有所不同。

如果是第一种问法，要求恰好装满背包，那么在初始化时除了 f[0]为 0 其它 f[1..V]均设为-∞，这样就可以保证最终得到的 f[N]是一种恰好装满背包的最优解。

如果并没有要求必须把背包装满，而是只希望价格尽量大，初始化时应该将 f[0..V]全部设为 0。

为什么呢？可以这样理解：初始化的 f 数组事实上就是在没有任何物品可以放入背包时的合法状态。如果要求背包恰好装满，那么此时只有容量为 0 的背包可能被价值为 0 的 nothing“恰好装满”，其它容量的背包均没有合法的解，属于未定义的状态，它们的值就

都应该是-∞了。如果背包并非必须被装满，那么任何容量的背包都有一个合法解“什么都不装”，这个解的价值为 0，所以初始时状态的值也就全部为 0 了。

这个小技巧完全可以推广到其它类型的背包问题，后面也就不再对进行状态转移之前的初始化进行讲解。

完全背包

问题引入

有 N 种物品和一个容量为 V 的背包，每种物品都有无限件可用。第 i 种物品的费用是 c[i]，价值是 w[i]。求解将哪些物品装入背包可使这些物品的费用总和不超过背包容量，且价值总和最大

基本思路

这个问题非常类似于 01 背包问题，所不同的是每种物品有无限件。也就是从每种物品的角度考虑，与它相关的策略已并非取或不取两种，而是有取 0 件、取 1 件、取 2 件……等很多种。如果仍然按照解 01 背包时的思路，令 f[i][v]表示前 i 种物品恰放入一个容量为 v 的背包的最大权值。仍然可以按照每种物品不同的策略写出状态转移方程，像这样：

f[i][v]=max{f[i-1][v-k*c[i]]+k*w[i]|0<=k*c[i]<=v}

这跟 01 背包问题一样有 O(N*V)个状态需要求解，但求解每个状态的时间已经不是常数了，求解状态 f[i][v]的时间是 O(v/c[i])，总的复杂度是超过 O(VN)的。

将 01 背包问题的基本思路加以改进，得到了这样一个清晰的方法。这说明 01 背包问题的方程的确是很重要，可以推及其它类型的背包问题。但我们还是试图改进这个复杂度。

一个简单有效的优化

完全背包问题有一个很简单有效的优化，是这样的：若两件物品 i、j 满足 c[i]<=c[j]且 w[i]>=w[j]，则将物品 j 去掉，不用考虑。这个优化的正确性显然：任何情况下都可将价值小费用高得 j 换成物美价廉的 i，得到至少不会更差的方案。对于随机生成的数据，这个方法往往会大大减少物品的件数，从而加快速度。然而这个并不能改善最坏情况的复杂度，因为有可能特别设计的数据可以一件物品也去不掉。

这个优化可以简单的O(N^2)地实现，一般都可以承受。另外，针对背包问题而言，比较不错的一种方法是：首先将费用大于V的物品去掉，然后使用类似计数排序的做法，计算出费用相同的物品中价值最高的是哪个，可以O(V+N)地完成这个优化。

转化为01背包问题求解

既然01背包问题是最基本的背包问题，那么我们可以考虑把完全背包问题转化为01背包问题来解。最简单的想法是，考虑到第i种物品最多选V/c[i]件，于是可以把第i种物品转化为V/c[i]件费用及价值均不变的物品，然后求解这个01背包问题。这样完全没有改进基本思路的时间复杂度，但这毕竟给了我们将完全背包问题转化为01背包问题的思路：将一种物品拆成多件物品。

O(VN)的算法

这个算法使用一维数组，先看伪代码：

```
for i=1..N
    for v=0..V
        f[v]=max{f[v],f[v-cost]+weight}
```

你会发现，这个伪代码与01背包的伪代码只有v的循环次序不同而已。为什么这样一改就可行呢？首先想想为什么P01中要按照v=V..0的逆序来循环。这是因为要保证第i次循环中的状态f[i][v]是由状态f[i-1][v-c[i]]递推而来。换句话说，这正是为了保证每件物品只选一次，保证在考虑“选入第i件物品”这件策略时，依据的是一个绝无已经选入第i件物品的子结果f[i-1][v-c[i]]。而现在完全背包的特点恰是每种物品可选无限件，所以在考虑“加选一件第i种物品”这种策略时，却正需要一个可能已选入第i种物品的子结果f[i][v-c[i]]，所以就可以并且必须采用v=0..V的顺序循环。这就是这个简单的程序为何成立的道理。

这个算法也可以以另外的思路得出。例如，基本思路中的状态转移方程可以等价地变形成这种形式：

```
f[i][v]=max{f[i-1][v],f[i][v-c[i]]+w[i]}
```

将这个方程用一维数组实现，便得到了上面的伪代码。

最后抽象出处理一件完全背包类物品的过程伪代码，以后会用到：

```
procedure CompletePack(cost,weight)
    for v=cost..V
        f[v]=max{f[v],f[v-c[i]]+w[i]}
```

海量数据处理

散列分治

方法介绍

对于海量数据而言，由于无法将其一次性装进内存进行处理，不得不将其通过散列映射的方法分割成相应的小块数据，然后再针对各个小块数据通过 HashMap 进行统计或者其他操作。

那么什么是散列映射呢？

简单来说，为了方便计算机在有限的内存中处理大量数据，通过映射的方式让数据均匀地分布在对应的内存位置上，而这种映射的方式通常通过散列函数进行映射，好的散列函数能让数据均匀分布而减少冲突。

问题实例

寻找 Top IP

从海量日志数据中提取出某日访问百度次数最多的那个 IP。

解法，三个步骤：

1.散列映射/分而治之。先将该日志访问百度的所有 IP 从日志文件中提取出来，然后逐个写入一个大文件中，接着采取散列映射的方法(如 hash(IP)%1000)，把整个大文件的数据映射到 1000 个小文件中。

2.HashMap 进行统计，大文件转换成了小文件，便可以采用 HashMap(ip,value)分别对 1000 个小文件的 IP 进行频率统计，找出每个小文件中出现频率最高的 IP，总共 1000 个 IP。

3.堆/快速排序。统计出 1000 个频率最高的 IP 后，根据他们各自频率的大小进行排序，找出最终那个出现频率最高的 IP。

寻找热门 IP

搜索引擎会通过日志文件把用户每次检索所使用的所有查询字符串都记录下来，每次查询串的长度为 1-255 字节，假设目前有 1000 万条查询记录(因为查询串的重复度比较高，去掉重复后，不超过 300 万个)，要求使用的内存不超过 1GB。

解决思路：这类 topk 问题一般性的思路就是分而治之(如有必要)+HashMap+堆。

首先我们要看一下数据能否一下子放到内存中处理，因为去重复后最多 300w 个 query 串，每个最多 255 字节，那么最多占用内存 300w*255=0.765GB，所以可以将所有的字符串放到内存中处理。所以分而治之这一步我们可以省略。

解法，两步：

1.HashMap 对这批海量数据进行统计，key 为 query 串，value 是出现次数，我们可以在 O(n)时间内完成频率的统计。

2.借助堆这种数据结构找出 topk，时间复杂度是 O(n'logk)，也就是说，借助堆可以在对数级时间内查找或者调整移动。因此，维护一个 k 大小的最小堆，然后遍历 300w 个 query 串，分别和根元素进行比较，最终得到的时间复杂度是 O(n)+O(n'logk)。其中 n 是 1000w，n'是 300w。

寻找出现频率最高的 100 个词

有一个 1GB 大小的文件，里面的每一行是一个词，每个词的大小不超过 16 字节，内存大小限制是 1MB，请返回出现频率最高的 100 个词。

三步：

1.分而治之/散列映射，按照先后顺序读取文件，对于每个词 x，执行 hash(x)%5000，然后将该值存到 5000 个小文件中，此时，每个小文件的大小应该是 200KB，当然，如果有的小文件超过了 1MB，则可以按照类似的方法继续向下分，直到分解得到的每个小文件都不超过 1MB。

2.HashMap 统计，对每个小文件采用 HashMap/Trie 树等数据结构，统计每个小文件中出现的词及其相应的出现次数。

3.堆排序或者归并排序。取出出现次数最多的 100 个词后，再把 100 个词及出现的次数存入文件中，这样又得到了 5000 个文件，最后对这 5000 个文件进行归并。

寻找共同的 URL

给定 a 和 b 两个文件，各存放 50 亿个 URL，每个 URL 占 64 字节，内存限制是 4GB，请找出 a 和 b 文件中共同的 URL。

每个文件的大小大概是 50 亿*64=320GB，远远大于内存限制，所以考虑散列映射的方法。

1.分而治之/散列映射。遍历文件 a，对每个 URL 求取 hash(URL)%1000，然后根据取得的值把 URL 分别存储到 1000 个小文件中，这样每个小文件大约为 300MB，对 b 也这么处理，这样处理后，所有相同的 URL 都在对应的小文件中，不对应的小文件中不可能有相同的 URL。然后只要求出 1000 对小文件中相同的 URL 即可。

2.HashSet 统计，求每对小文件中相同的 URL 时，可以把其中一个小文件的 URL 存储到 HashSet 中，然后遍历另外一个小文件的每个 URL，看是否在刚才构建的 HashSet 中。如果在，就是共同的 URL。

位图

什么是位图？

所谓位图，就是用一个位(bit)来标记某个元素对应的值，而键就是该元素，由于采用了位为单位来存储数据，因此可以大大节省存储空间。

位图使用位数组来表示某些元素是否存在，可进行数据的快速查找、判重、删除。

来看一个具体的例子，假设我们要对 0-7 中的 5 个元素(4、7、2、5、3)进行排序，此时就可以使用位图的方式来达到目的。因为要表示 8 个数，所以只需要 8 位，由于 8 位等于 1 字节，所以开辟 1 字节的空间，并将这个空间的所有位都置为 0。此时 8 位分别是 00000000。

然后遍历五个元素，因为待排序数组的第一个元素是 4，所以把 4 对应的位重置为 1，因为从 0 开始计数，所以第 5 位被置为 1，此时 8 位上的数据分别是 00001000。

然后依次处理数组中接下来的数字。最后 8 位上的数据分别是 00111101。

现在遍历这个数组，把某位是 1 的位的编号(2、3、4、5、7)输出，这样就达到了目的。

下面是一个使用 BitMap 完成排序的例子。

代码如下:

```
package com.algorithm.bitmap;

import java.util.BitSet;

/**
 * 位图的使用
 * @author yy
 *
 */
public class BitMap {

    /**
     * 位图完成排序
     * @param a
     * @return
     */
    private static void sortByBitMap(int[] a) {
        BitSet bitSet = new BitSet(8);
        for (int i = 0; i < a.length; i++) {
            bitSet.set(a[i]);
        }
        for (int i = 0; i < bitSet.size(); i++) {
            if(bitSet.get(i)) {
                System.out.print(i+" ");
            }
        }
```

```
    }

    /**
     * 测试
     * @param args
     */
    public static void main(String[] args) {
        int[] a = {4,7,2,5,3};
        sortByBitMap(a);

    }

}
```

问题实例

电话号码的统计

已知某个文件内包含一些电话号码，每个号码为 8 位数字，统计不同号码的个数，8 位数字最多组成 99999999 个号码，大概需要 99 兆位，大概十几兆字节的内存即可。

2.5 亿个整数的去重

在 2.5 亿个整数中找出不重复的整数。注意，内存不足以容纳这 2.5 亿个整数。

采用 2 位图(每个数分配两位，00 表示不存在，01 表示出现一次，10 表示出现多次，11 表示无意义)，共需内存 2^32*2=1GB 的内存，可以接受。然后扫描这 2.5 亿个整数，查看位图中相应的位，如果是 00 就变为 01,01 就变为 10，如果是 10 就保持不变，扫描完之后，查看位图，把对应位是 01 的整数输出即可。

整数的快速查询

给定 40 亿个不重复的没排过序的 unsigned int 型整数，然后再给定一个数，如何快速判断这个数是否在这 40 亿个整数当中。

可以使用位图的方案，申请 512MB 的内存，一个位代表一个 unsigned int 类型的值，读入 40 亿个数，设置相应的位，读入要查询的数，查看相应的位是否是 1，如果是 1 表示存在，如果是 0 就表示不存在。

布隆过滤器

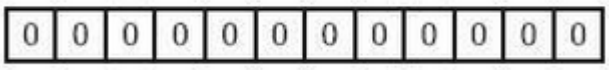
方法介绍

Bloom Filter 是一种空间效率很高的随机数据结构，它的原理是，当一个元素被加入集合时，通过 K 个 Hash 函数将这个元素映射成一个位阵列（Bit array）中的 K 个点，把它们置为 1。检索时，我们只要看看这些点是不是都是 1 就（大约）知道集合中有没有它了：如果这些点有任何一个 0，则被检索元素一定不在；如果都是 1，则被检索元素很可能在。这就是布隆过滤器的基本思想。

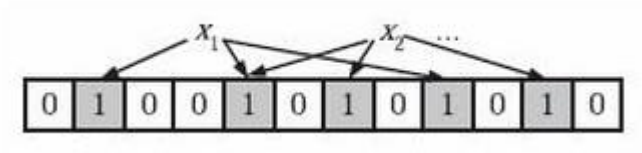
但 Bloom Filter 的这种高效是有一定代价的：在判断一个元素是否属于某个集合时，有可能会把不属于这个集合的元素误认为属于这个集合（false positive）。因此，Bloom Filter 不适合那些“零错误”的应用场合。而在能容忍低错误率的应用场合下，Bloom Filter 通过极少的错误换取了存储空间的极大节省。

集合表示和元素查询

下面我们具体来看 Bloom Filter 是如何用位数组表示集合的。初始状态时，Bloom Filter 是一个包含 m 位的位数组，每一位都置为 0。



为了表达 $S=\{x_1, x_2,\ldots,x_n\}$这样一个 n 个元素的集合，Bloom Filter 使用 k 个相互独立的哈希函数（Hash Function），它们分别将集合中的每个元素映射到$\{1,\ldots,m\}$的范围中。对任意一个元素 x，第 i 个哈希函数映射的位置 $h_i(x)$就会被置为 1（$1\le i\le k$）。注意，如果一个位置多次被置为 1，那么只有第一次会起作用，后面几次将没有任何效果。在下图中，k=3，且有两个哈希函数选中同一个位置（从左边数第五位，即第二个“1“处）。



在判断 y 是否属于这个集合时，我们对 y 应用 k 次哈希函数，如果所有 $h_i(y)$的位置都是 1（$1\le i\le k$），那么我们就认为 y 是集合中的元素，否则就认为 y 不是集合中的元素。下图中 y_1 就不是集合中的元素（因为 y1 有一处指向了“0”位）。y_2 或者属于这个集合，或者刚好是一个 false positive。

错误率估计

前面我们已经提到了，Bloom Filter 在判断一个元素是否属于它表示的集合时会有一定的错误率（false positive rate），下面我们就来估计错误率的大小。在估计之前为了简化模型，我们假设 kn<m 且各个哈希函数是完全随机的。当集合 $S=\{x_1, x_2,\ldots,x_n\}$的所有元素都被 k 个哈希函数映射到 m 位的位数组中时，这个位数组中某一位还是 0 的概率是：

$$p' = \left(1 - \frac{1}{m}\right)^{kn} \approx e^{-kn/m}.$$

其中 1/m 表示任意一个哈希函数选中这一位的概率（前提是哈希函数是完全随机的），(1-1/m)表示哈希一次没有选中这一位的概率。要把 S 完全映射到位数组中，需要做 kn 次哈希。某一位还是 0 意味着 kn 次哈希都没有选中它，因此这个概率就是（1-1/m）的 kn 次方。令 $p = e^{-kn/m}$ 是为了简化运算，这里用到了计算 e 时常用的近似：

$$\lim_{x\to\infty}\left(1 - \frac{1}{x}\right)^{-x} = e$$

令 ρ 为位数组中 0 的比例，则 ρ 的数学期望 E(ρ)= p’。在 ρ 已知的情况下，要求的错误率（false positive rate）为：

$$(1-\rho)^k \approx (1-p')^k \approx (1-p)^k.$$

(1-ρ)为位数组中 1 的比例，$(1-\rho)^k$ 就表示 k 次哈希都刚好选中 1 的区域，即 false positive rate。上式中第二步近似在前面已经提到了，现在来看第一步近似。p’只是 ρ 的数学期望，在实际中 ρ 的值有可能偏离它的数学期望值。M. Mitzenmacher 已经证明[2]，位数组中 0 的比例非常集中地分布在它的数学期望值的附近。因此，第一步的近似得以成立。分别将 p 和 p’代入上式中，得：

$$f' = \left(1 - \left(1 - \frac{1}{m}\right)^{kn}\right)^k = (1-p')^k$$

$$f = \left(1 - e^{-kn/m}\right)^k = (1-p)^k.$$

相比 p’和 f’，使用 p 和 f 通常在分析中更为方便。

最优的哈希函数个数

既然 Bloom Filter 要靠多个哈希函数将集合映射到位数组中，那么应该选择几个哈希函数才能使元素查询时的错误率降到最低呢？这里有两个互斥的理由：如果哈希函数的个数多，那么在对一个不属于集合的元素进行查询时得到 0 的概率就大；但另一方面，如果哈希函数的个数少，那么位数组中的 0 就多。为了得到最优的哈希函数个数，我们需要根据上一小节中的错误率公式进行计算。

先用 p 和 f 进行计算。注意到 $f = \exp(k \ln(1 - e^{-kn/m}))$，我们令 $g = k \ln(1 - e^{-kn/m})$，只要让 g 取到最小，f 自然也取到最小。由于 $p = e^{-kn/m}$，我们可以将 g 写成

$$g = -\frac{m}{n} \ln(p) \ln(1 - p),$$

根据对称性法则可以很容易看出当 p = 1/2，也就是 $k = \ln 2 \cdot (m/n)$时，g 取得最小值。在这种情况下，最小错误率 f 等于$(1/2)^k \approx (0.6185)^{m/n}$。另外，注意到 p 是位数组中某一位仍是 0 的概率，所以 p = 1/2 对应着位数组中 0 和 1 各一半。换句话说，要想保持错误率低，最好让位数组有一半还空着。

需要强调的一点是，p = 1/2 时错误率最小这个结果并不依赖于近似值 p 和 f。同样对于 $f' = \exp(k \ln(1 - (1 - 1/m)^{kn}))$，$g' = k \ln(1 - (1 - 1/m)^{kn})$，$p' = (1 - 1/m)^{kn}$，我们可以将 g’写成

$$g' = \frac{1}{n \ln(1 - 1/m)} \ln(p') \ln(1 - p'),$$

同样根据对称性法则可以得到当 p’ = 1/2 时，g’取得最小值。

位数组的大小

下面我们来看看，在不超过一定错误率的情况下，Bloom Filter 至少需要多少位才能表示全集中任意 n 个元素的集合。假设全集中共有 u 个元素，允许的最大错误率为 ϵ，下面我们来求位数组的位数 m。

假设 X 为全集中任取 n 个元素的集合，F(X)是表示 X 的位数组。那么对于集合 X 中任意一个元素 x，在 s = F(X)中查询 x 都能得到肯定的结果，即 s 能够接受 x。显然，由于

Bloom Filter 引入了错误，s 能够接受的不仅仅是 X 中的元素，它还能够 ϵ (u - n)个 false positive。因此，对于一个确定的位数组来说，它能够接受总共 n + ϵ (u - n)个元素。在 n + ϵ (u - n)个元素中，s 真正表示的只有其中 n 个，所以一个确定的位数组可以表示

$$\binom{n+\epsilon(u-n)}{n}$$

个集合。m 位的位数组共有 2^m 个不同的组合，进而可以推出，m 位的位数组可以表示

$$2^m\binom{n+\epsilon(u-n)}{n}$$

个集合。全集中 n 个元素的集合总共有

$$\binom{u}{n}$$

个，因此要让 m 位的位数组能够表示所有 n 个元素的集合，必须有

$$2^m\binom{n+\epsilon(u-n)}{n} \geq \binom{u}{n}$$

即：

$$m \geq \log_2 \frac{\binom{u}{n}}{\binom{n+\epsilon(u-n)}{n}} \approx \log_2 \frac{\binom{u}{n}}{\binom{\epsilon u}{n}} \geq \log_2 \epsilon^{-n} = n\log_2(1/\epsilon).$$

上式中的近似前提是 n 和 ϵu 相比很小，这也是实际情况中常常发生的。根据上式，我们得出结论：在错误率不大于 ϵ 的情况下，m 至少要等于 $n \log_2(1/\epsilon)$才能表示任意 n 个元素的集合。

上一小节中我们曾算出当 $k = \ln 2 \cdot (m/n)$时错误率 f 最小，这时 $f = (1/2)^k = (1/2)^{m \ln 2 / n}$。现在令 $f \leq \epsilon$，可以推出

$$m \geq n \frac{\log_2(1/\epsilon)}{\ln 2} = n \log_2 e \cdot \log_2(1/\epsilon).$$

这个结果比前面我们算得的下界 $n \log_2(1/\epsilon)$大了 $\log_2 e \approx 1.44$ 倍。这说明在哈希函数的个数取到最优时，要让错误率不超过 ϵ，m 至少需要取到最小值的 1.44 倍。

问题案例

给定 A 和 B 两个文件，各存放 50 亿条 URL，每条 URL 占用 64 字节，内存限制是 4GB，请找出 A 和 B 两个文件中共同的 URL。

分析：如果允许一定的误判率，可以使用布隆过滤器。4GB 内存大概可以表示 340 亿位，将其中一个文件的 URL 使用布隆过滤器映射到这 340 亿位，然后挨个读取另外一个文件中的 URL，检查这两个 URL 是否相等，如果是，那么该 URL 应该是共同的 URL。

多层划分

方法介绍

多层划分本质上还是遵循分而治之的思想，因为元素范围很大，不能利用直接寻址表，所以通过多次划分，逐步确定范围，然后在一个可以接受的范围内查找。

问题案例

寻找不重复的数

在 2.5 亿个整数中寻找不重复的整数的个数，注意，内存空间不足以容纳这 2.5 亿个整数。

分析：因为整数个数为 2^32，所以可以把这 2^32 个数分为 2^8 个区域，然后将数据划分到不同的区域，最后不同区域再利用位图进行统计就可以直接解决了。也就是说，只要有足够的内存空间，就可以很方便的解决。

寻找中位数

寻找 5 亿个 int 型数的中位数。

分析：首先将这 5 亿个数划分为 2^16 个区域，然后读取数据统计落到各个区域里的数的个数。根据统计结果就可以判断中位数落到哪个区域，并知道这个区域中的第几大数刚好是中位数。然后，第二次扫描只统计在这个区域中的那些数就可以了。

外排序

方法介绍

顾名思义，外部排序就是在内存外面的排序，当需要处理的数据量很大而不能一次性装入内存时，只能将数据放在读写较慢的外存储器(通常是硬盘)上。

外排序通常采用的是一种“排序-归并”的策略，在排序阶段，先读入能放在内存中的数据，将其排序后输出到一个临时文件，依次进行，将待排序数据组织为多个有序的临时文件，而后在归并阶段将这些临时文件组合成一个大的有序文件，即为排序结果。

问题实例

给 10^7 个数据的磁盘文件排序

给定一个文件，里面最多含有 n 个不重复的正整数，且每个数都小于等于 n(n=10^7)，请输出一个按照从小到大升序排列的包含所有输入整数的列表。假设最多有 1MB 的内存空间可用，但是磁盘空间足够，要求运行时间在 5min 之内，10s 为最佳结果。

位图方案

你可能会想到把磁盘文件进行归并排序，但题目要求你只有 1MB 的内存空间可用，所以，归并排序这个方法不行。

熟悉位图的朋友可能会想到用位图来表示这个文件集合。例如正如编程珠玑一书上所述，用一个 20 位长的字符串来表示一个所有元素都小于 20 的简单的非负整数集合，边框用如下字符串来表示集合{1,2,3,5,8,13}：

0 1 1 1 0 1 0 0 1 0 0 0 0 1 0 0 0 0 0 0

上述集合中各数对应的位置则置 1，没有对应的数的位置则置 0。

参考编程珠玑一书上的位图方案，针对我们的 10^7 个数据量的磁盘文件排序问题，我们可以这么考虑，由于每个 7 位十进制整数表示一个小于 1000 万的整数。我们可以使用一个具有 1000 万个位的字符串来表示这个文件，其中，当且仅当整数 i 在文件中存在时，第 i 位为 1。采取这个位图的方案是因为我们面对的这个问题的特殊性：1、输入数据限制在相对较小的范围内，2、数据没有重复，3、其中的每条记录都是单一的整数，没有任何其它与之关联的数据。

所以，此问题用位图的方案分为以下三步进行解决：

- 第一步，将所有的位都置为 0，从而将集合初始化为空。
- 第二步，通过读入文件中的每个整数来建立集合，将每个对应的位都置为 1。
- 第三步，检验每一位，如果该位为 1，就输出对应的整数。

经过以上三步后，产生有序的输出文件。令 n 为位图向量中的位数（本例中为 10000000），程序可以用伪代码表示如下：

```
//磁盘文件排序位图方案的伪代码

//第一步，将所有的位都初始化为 0

for i ={0,....n}

   bit[i]=0;

//第二步，通过读入文件中的每个整数来建立集合，将每个对应的位都置为 1。

for each i in the input file

   bit[i]=1;

//第三步，检验每一位，如果该位为 1，就输出对应的整数。
```

```
for i={0...n}
  if bit[i]==1
    write i on the output file
```

不过很快我们就意识到用此位图方法，严格来说还是不行，空间消耗1000000/(8*1024*1024)还是大于 1MB。

位图方案使用的时候我们一定要注意：适用于非重复数据处理。

多路归并

分而治之，大而化小，也就是把整个大文件分为若干大小的几块，然后分别对每一块进行排序，最后完成整个过程的排序。k 趟算法可以在 kn 的时间开销内和 n/k 的空间开销内完成对最多 n 个小于 n 的无重复正整数的排序。

比如可分为 2 块（k=2，1 趟反正占用的内存只有 1.25/2M），1~4999999，和 5000000~9999999。先遍历一趟，首先排序处理 1~4999999 之间的整数（用 5000000/8=625000 个字的存储空间来排序 0~4999999 之间的整数），然后再第二趟，对 5000001~1000000 之间的整数进行排序处理。

Trie 树

问题实例

10 个出现最频繁的词

在一个文本文件中大约有 1w 行，每行 1 个词，要求统计出其中出现次数最频繁的 10 个词。

分析：用 Trie 树统计每个词出现的次数，时间复杂度是 O(nl)，l 是单词的平均长度，最终找出出现最频繁的 10 个词。(可以使用堆来实现，时间复杂度是 O(nlog10))。

倒排索引

方法介绍

倒排索引是一种索引方法，用来存储在全文搜索下某个单词在一个文档或者一组文档中存储位置的映射，常用于搜索引擎和关键字查询等问题中。

以英文为例，下面是要被索引的文本：
T0 = “it is what it is”
T1 = “what is it”
T2 = “it is a banana”
我们就能得到如下的倒排索引：
“a” {2}
“banana” {2}
“is” {0,1,2}
“it” {0,1,2}
“what” {0,1}

问题实例

文档检索系统

请设计一个文档检索系统，用于查询哪些文件包含了某个单词，比如常见的学术论文的关键字搜索。
提示：建立倒排索引。

Java 实现一个简单的倒排索引

数据文件:news.txt

```
I am eriol
I live in Shanghai and I love Shanghai
I also love travelling
life in Shanghai
is beautiful
```

代码：

```
package com.algorithm.inverseindex;
import java.io.*;
import java.util.HashMap;
```

```java
import java.util.Map;

/**
 * 倒排索引的实现
 *
 */
public class InvertedIndex {

    private Map<String, Map<Integer, Integer>> index;  //倒排索引表
    private Map<Integer, Integer> subIndex;

    /**
     * 建立倒排索引
     * @param filePath
     */
    public void createIndex(String filePath) {
        index = new HashMap<String, Map<Integer, Integer>>();

        try {
            File file = new File(filePath);
            InputStream is = new FileInputStream(file);
            BufferedReader read = new BufferedReader(new
InputStreamReader(is));

            String temp = null;
            int line = 1;
            while ((temp = read.readLine()) != null) {
                String[] words = temp.split(" ");
                for (String word : words) {
                    if (!index.containsKey(word)) {
                        subIndex = new HashMap<Integer, Integer>();
                        subIndex.put(line, 1);
                        index.put(word, subIndex);
                    } else {
                        subIndex = index.get(word);
                        if (subIndex.containsKey(line)) {
                            int count = subIndex.get(line);
                            subIndex.put(line, count+1);
                        } else {
                            subIndex.put(line, 1);
                        }
                    }
                }
                line++;
```

```
            }
            read.close();
            is.close();
        } catch (IOException e) {
            System.out.println("error in read file");
        }
    }

    /**
     * 在倒排索引表中查询指定字符串
     * @param str  字符串之间以“ ”隔开
     */
    public void find(String str) {
        String[] words = str.split(" ");
        for (String word : words) {
            StringBuilder sb = new StringBuilder();
            if (index.containsKey(word)) {
                sb.append("word: " + word + " in ");
                Map<Integer, Integer> temp = index.get(word);
                for (Map.Entry<Integer, Integer> e : temp.entrySet()) {
                    sb.append("line " + e.getKey() + " [" + e.getValue()
+ "] , ");
                }
            } else {
                sb.append("word: " + word + " not found");
            }
            System.out.println(sb);
        }
    }

    public static void main(String[] args) {
        InvertedIndex index = new InvertedIndex();
        index.createIndex("news.txt");
        index.find("I love Shanghai today");
    }
}
```

输出：

```
word: I in line 1 [1] , line 2 [2] , line 3 [1] ,
word: love in line 2 [1] , line 3 [1] ,
word: Shanghai in line 2 [2] , line 4 [1] ,
word: today not found
```