```
public int binarySearch(int[] nums, int key) {
    int l = 0, h = nums.length - 1;
    while (l <= h) {
        int m = l + (h - l) / 2;
        if (nums[m] == key) {
            return m;
        } else if (nums[m] > key) {
            h = m - 1;
        } else {
            l = m + 1;
        }
    }
    return -1;
}
```

二分查找可以有很多变种，变种实现要注意边界值的判断。例如在一个有重复元素的数组中查找 key 的最左位置的实现如下:

```
public int binarySearch(int[] nums, int key) {
    int l = 0, h = nums.length - 1;
    while (l < h) {
        int m = l + (h - l) / 2;
        if (nums[m] >= key) {
            h = m;
        } else {
            l = m + 1;
        }
    }
    return l;
}
```

归并

```
/**
 * 归并排序: Java
 *
 * @author skywang
 * @date 2014/03/12
 */

public class MergeSort {

    /*
     * 将一个数组中的两个相邻有序区间合并成一个
     *
```

```
     * 参数说明:
     *     a -- 包含两个有序区间的数组
     *     start -- 第1个有序区间的起始地址。
     *     mid   -- 第1个有序区间的结束地址。也是第2个有序区间的起始地址。
     *     end   -- 第2个有序区间的结束地址。
     */
    public static void merge(int[] a, int start, int mid, int end) {
        int[] tmp = new int[end-start+1];    // tmp是汇总2个有序区的临时区域
        int i = start;            // 第1个有序区的索引
        int j = mid + 1;          // 第2个有序区的索引
        int k = 0;                // 临时区域的索引

        while(i <= mid && j <= end) {
            if (a[i] <= a[j])
                tmp[k++] = a[i++];
            else
                tmp[k++] = a[j++];
        }

        while(i <= mid)
            tmp[k++] = a[i++];

        while(j <= end)
            tmp[k++] = a[j++];

        // 将排序后的元素，全部都整合到数组a中。
        for (i = 0; i < k; i++)
            a[start + i] = tmp[i];

        tmp=null;
    }

    /*
     * 归并排序(从上往下)
     *
     * 参数说明:
     *     a -- 待排序的数组
     *     start -- 数组的起始地址
     *     endi -- 数组的结束地址
     */
    public static void mergeSortUp2Down(int[] a, int start, int end) {
        if(a==null || start >= end)
            return ;

        int mid = (end + start)/2;
        mergeSortUp2Down(a, start, mid); // 递归排序a[start...mid]
        mergeSortUp2Down(a, mid+1, end); // 递归排序a[mid+1...end]

        // a[start...mid] 和 a[mid...end]是两个有序空间，
        // 将它们排序成一个有序空间a[start...end]
        merge(a, start, mid, end);
    }


    /*
     * 对数组a做若干次合并: 数组a的总长度为len，将它分为若干个长度为gap的子数组；
     *              将"每2个相邻的子数组" 进行合并排序。
     *
```

```
     * 参数说明:
     *     a -- 待排序的数组
     *     len -- 数组的长度
     *     gap -- 子数组的长度
     */
    public static void mergeGroups(int[] a, int len, int gap) {
        int i;
        int twolen = 2 * gap;     // 两个相邻的子数组的长度

        // 将"每2个相邻的子数组" 进行合并排序。
        for(i = 0; i+2*gap-1 < len; i+=(2*gap))
            merge(a, i, i+gap-1, i+2*gap-1);

        // 若 i+gap-1 < len-1，则剩余一个子数组没有配对。
        // 将该子数组合并到已排序的数组中。
        if ( i+gap-1 < len-1)
            merge(a, i, i + gap - 1, len - 1);
    }

    /*
     * 归并排序(从下往上)
     *
     * 参数说明:
     *     a -- 待排序的数组
     */
    public static void mergeSortDown2Up(int[] a) {
        if (a==null)
            return ;

        for(int n = 1; n < a.length; n*=2)
            mergeGroups(a, a.length, n);
    }

    public static void main(String[] args) {
        int i;
        int a[] = {80,30,60,40,20,10,50,70};

        System.out.printf("before sort:");
        for (i=0; i<a.length; i++)
            System.out.printf("%d ", a[i]);
        System.out.printf("\n");

        mergeSortUp2Down(a, 0, a.length-1);         // 归并排序(从上往下)
        //mergeSortDown2Up(a);                      // 归并排序(从下往上)

        System.out.printf("after  sort:");
        for (i=0; i<a.length; i++)
            System.out.printf("%d ", a[i]);
        System.out.printf("\n");
    }
}
```

java自带排序

```
```

```
 Collections.sort(intList,new Comparator<Integer>() {

            @Override
            public int compare(Integer o1, Integer o2) {
                // 返回值为int类型，大于0表示正序，小于0表示逆序,从小到大；从大到小。
                return o2-o1;
            }
        });
```

还有一个arrays.sort()使用的快速排序，不稳定。

```
import java.util.*;
public class paixu3 {
    public static void main(String args[]){
        Scanner in=new Scanner(System.in);
        int num=in.nextInt();
        TT arr[]=new TT[num];
        for(int i=0;i<num;i++){
            int num1=in.nextInt();
            int num2=in.nextInt();
            arr[i]=new TT(num1,num2);
        }
        Arrays.sort(arr, new Comparator<TT>() {
            @Override
            public int compare(TT o1, TT o2) {
                if(o1.num1-o2.num1>0){
                    return 1;
                }
                else if(o1.num1-o2.num1==0){
                    if(o1.num2-o2.num2>=0){
                        return 1;
                    }else{
                        return -1;
                    }
                }
                return -1;
            }
        });
        for(int i=0;i<num;i++){
            System.out.println(arr[i].num1+" "+arr[i].num2);
        }
    }
}
class TTT {
    int num1;
    int num2;
    TTT(int n1,int n2){
        num1=n1;
        num2=n2;
    }
}
```

java内置优先队列　优先队列PriorityQueue是Queue接口的实现，可以对其中元素进行排序

```
peek()//返回队首元素
poll()//返回队首元素，队首元素出队列
add()//添加元素
size()//返回队列元素个数
isEmpty()//判断队列是否为空，为空返回true,不空返回false
    PriorityQueue<Integer> minHeap = new PriorityQueue<Integer>(); //小顶堆，默认容量为11
PriorityQueue<Integer> maxHeap = new PriorityQueue<Integer>(11,new Comparator<Integer>(){ //大顶堆，容量11
    @Override
    public int compare(Integer i1,Integer i2){
        return i2-i1;
    }
});
```

boolean	add(E e) 将指定的元素插入此优先级队列。
void	clear() 从此优先级队列中移除所有元素。
Comparator<? super E>	comparator() 返回用来对此队列中的元素进行排序的比较器；如果此队列根据其元素的自然顺序进行排序，则返回 null。
boolean	contains(Object o) 如果此队列包含指定的元素，则返回 true。
Iterator<E>	iterator() 返回在此队列中的元素上进行迭代的迭代器。
boolean	offer(E e) 将指定的元素插入此优先级队列。
E	peek() 获取但不移除此队列的头；如果此队列为空，则返回 null。
E	poll() 获取并移除此队列的头，如果此队列为空，则返回 null。
boolean	remove(Object o) 从此队列中移除指定元素的单个实例（如果存在）。
int	size() 返回此 collection 中的元素数。
Object[]	toArray() 返回一个包含此队列所有元素的数组。
<T> T[]	toArray(T[] a) 返回一个包含此队列所有元素的数组；返回数组的运行时类型是指定数组的类型。

```
Queue<node1> dui=new PriorityQueue<>();
```

```
class node1 implements Comparable<node1>{
    int id;
    long dis=-1;
    boolean goast=false;
    ArrayList<road>roads=new ArrayList<>();
    public node1(int id) { this.id=id; }

    @Override
    public int compareTo(node1 o) { return Long.compare(this.dis,o.dis); }
}
```

二叉树+前中后序遍历+查找key节点+找最小最大节点+找前驱后继节点+插入删除节点+打印二叉树

```
/**
 * Java 语言: 二叉查找树
 *
 * @author skywang
 * @date 2013/11/07
 */

public class BSTree<T extends Comparable<T>> {

    private BSTNode<T> mRoot;    // 根结点

    public class BSTNode<T extends Comparable<T>> {
        T key;                // 关键字(键值)
        BSTNode<T> left;    // 左孩子
        BSTNode<T> right;    // 右孩子
        BSTNode<T> parent;    // 父结点

        public BSTNode(T key, BSTNode<T> parent, BSTNode<T> left, BSTNode<T>
right) {
            this.key = key;
            this.parent = parent;
            this.left = left;
            this.right = right;
        }

        public T getKey() {
            return key;
        }

        public String toString() {
            return "key:"+key;
        }
    }

    public BSTree() {
        mRoot=null;
    }

    /*
     * 前序遍历"二叉树"
```

```
     */
    private void preOrder(BSTNode<T> tree) {
        if(tree != null) {
            System.out.print(tree.key+" ");
            preOrder(tree.left);
            preOrder(tree.right);
        }
    }

    public void preOrder() {
        preOrder(mRoot);
    }

    /*
     * 中序遍历"二叉树"
     */
    private void inOrder(BSTNode<T> tree) {
        if(tree != null) {
            inOrder(tree.left);
            System.out.print(tree.key+" ");
            inOrder(tree.right);
        }
    }

    public void inOrder() {
        inOrder(mRoot);
    }


    /*
     * 后序遍历"二叉树"
     */
    private void postOrder(BSTNode<T> tree) {
        if(tree != null)
        {
            postOrder(tree.left);
            postOrder(tree.right);
            System.out.print(tree.key+" ");
        }
    }

    public void postOrder() {
        postOrder(mRoot);
    }


    /*
     * (递归实现)查找"二叉树x"中键值为key的节点
     */
    private BSTNode<T> search(BSTNode<T> x, T key) {
        if (x==null)
            return x;

        int cmp = key.compareTo(x.key);
        if (cmp < 0)
            return search(x.left, key);
        else if (cmp > 0)
            return search(x.right, key);
```

```
        else
            return x;
    }

    public BSTNode<T> search(T key) {
        return search(mRoot, key);
    }

    /*
     * (非递归实现)查找"二叉树x"中键值为key的节点
     */
    private BSTNode<T> iterativeSearch(BSTNode<T> x, T key) {
        while (x!=null) {
            int cmp = key.compareTo(x.key);

            if (cmp < 0)
                x = x.left;
            else if (cmp > 0)
                x = x.right;
            else
                return x;
        }

        return x;
    }

    public BSTNode<T> iterativeSearch(T key) {
        return iterativeSearch(mRoot, key);
    }

    /*
     * 查找最小结点: 返回tree为根结点的二叉树的最小结点。
     */
    private BSTNode<T> minimum(BSTNode<T> tree) {
        if (tree == null)
            return null;

        while(tree.left != null)
            tree = tree.left;
        return tree;
    }

    public T minimum() {
        BSTNode<T> p = minimum(mRoot);
        if (p != null)
            return p.key;

        return null;
    }

    /*
     * 查找最大结点: 返回tree为根结点的二叉树的最大结点。
     */
    private BSTNode<T> maximum(BSTNode<T> tree) {
        if (tree == null)
            return null;

        while(tree.right != null)
```

```
            tree = tree.right;
        return tree;
    }

    public T maximum() {
        BSTNode<T> p = maximum(mRoot);
        if (p != null)
            return p.key;

        return null;
    }

    /*
     * 找结点(x)的后继结点。即，查找"二叉树中数据值大于该结点"的"最小结点"。
     */
    public BSTNode<T> successor(BSTNode<T> x) {
        // 如果x存在右孩子，则"x的后继结点"为 "以其右孩子为根的子树的最小结点"。
        if (x.right != null)
            return minimum(x.right);

        // 如果x没有右孩子。则x有以下两种可能:
        // (01) x是"一个左孩子"，则"x的后继结点"为 "它的父结点"。
        // (02) x是"一个右孩子"，则查找"x的最低的父结点，并且该父结点要具有左孩子"，找到的这个"最低的父结点"就是"x的后继结点"。
        BSTNode<T> y = x.parent;
        while ((y!=null) && (x==y.right)) {
            x = y;
            y = y.parent;
        }

        return y;
    }

    /*
     * 找结点(x)的前驱结点。即，查找"二叉树中数据值小于该结点"的"最大结点"。
     */
    public BSTNode<T> predecessor(BSTNode<T> x) {
        // 如果x存在左孩子，则"x的前驱结点"为 "以其左孩子为根的子树的最大结点"。
        if (x.left != null)
            return maximum(x.left);

        // 如果x没有左孩子。则x有以下两种可能:
        // (01) x是"一个右孩子"，则"x的前驱结点"为 "它的父结点"。
        // (01) x是"一个左孩子"，则查找"x的最低的父结点，并且该父结点要具有右孩子"，找到的这个"最低的父结点"就是"x的前驱结点"。
        BSTNode<T> y = x.parent;
        while ((y!=null) && (x==y.left)) {
            x = y;
            y = y.parent;
        }

        return y;
    }

    /*
     * 将结点插入到二叉树中
     *
     * 参数说明:
```

```
     *     tree 二叉树的
     *     z 插入的结点
     */
    private void insert(BSTree<T> bst, BSTNode<T> z) {
        int cmp;
        BSTNode<T> y = null;
        BSTNode<T> x = bst.mRoot;

        // 查找z的插入位置
        while (x != null) {
            y = x;
            cmp = z.key.compareTo(x.key);
            if (cmp < 0)
                x = x.left;
            else
                x = x.right;
        }

        z.parent = y;
        if (y==null)
            bst.mRoot = z;
        else {
            cmp = z.key.compareTo(y.key);
            if (cmp < 0)
                y.left = z;
            else
                y.right = z;
        }
    }

    /*
     * 新建结点(key)，并将其插入到二叉树中
     *
     * 参数说明:
     *     tree 二叉树的根结点
     *     key 插入结点的键值
     */
    public void insert(T key) {
        BSTNode<T> z=new BSTNode<T>(key,null,null,null);

        // 如果新建结点失败，则返回。
        if (z != null)
            insert(this, z);
    }

    /*
     * 删除结点(z)，并返回被删除的结点
     *
     * 参数说明:
     *     bst 二叉树
     *     z 删除的结点
     */
    private BSTNode<T> remove(BSTree<T> bst, BSTNode<T> z) {
        BSTNode<T> x=null;
        BSTNode<T> y=null;

        if ((z.left == null) || (z.right == null) )
            y = z;
```

```
        else
            y = successor(z);

        if (y.left != null)
            x = y.left;
        else
            x = y.right;

        if (x != null)
            x.parent = y.parent;

        if (y.parent == null)
            bst.mRoot = x;
        else if (y == y.parent.left)
            y.parent.left = x;
        else
            y.parent.right = x;

        if (y != z)
            z.key = y.key;

        return y;
    }

    /*
     * 删除结点(z)，并返回被删除的结点
     *
     * 参数说明:
     *     tree 二叉树的根结点
     *     z 删除的结点
     */
    public void remove(T key) {
        BSTNode<T> z, node;

        if ((z = search(mRoot, key)) != null)
            if ( (node = remove(this, z)) != null)
                node = null;
    }

    /*
     * 销毁二叉树
     */
    private void destroy(BSTNode<T> tree) {
        if (tree==null)
            return ;

        if (tree.left != null)
            destroy(tree.left);
        if (tree.right != null)
            destroy(tree.right);

        tree=null;
    }

    public void clear() {
        destroy(mRoot);
        mRoot = null;
    }
```

```
    /*
     * 打印"二叉查找树"
     *
     * key        -- 节点的键值
     * direction  --  0，表示该节点是根节点;
     *               -1，表示该节点是它的父结点的左孩子;
     *                1，表示该节点是它的父结点的右孩子。
     */
    private void print(BSTNode<T> tree, T key, int direction) {

        if(tree != null) {

            if(direction==0)    // tree是根节点
                System.out.printf("%2d is root\n", tree.key);
            else                // tree是分支节点
                System.out.printf("%2d is %2d's %6s child\n", tree.key, key,
direction==1?"right" : "left");

            print(tree.left, tree.key, -1);
            print(tree.right,tree.key,  1);
        }
    }

    public void print() {
        if (mRoot != null)
            print(mRoot, mRoot.key, 0);
    }
}
```

平衡二叉树

```
public class AVLTree<T extends Comparable<T>> {
    private AVLTreeNode<T> mRoot;    // 根结点

    // AVL树的节点(内部类)
    class AVLTreeNode<T extends Comparable<T>> {
        T key;                // 关键字(键值)
        int height;           // 高度
        AVLTreeNode<T> left;    // 左孩子
        AVLTreeNode<T> right;    // 右孩子

        public AVLTreeNode(T key, AVLTreeNode<T> left, AVLTreeNode<T> right) {
            this.key = key;
            this.left = left;
            this.right = right;
            this.height = 0;
        }
    }

    ......
}
/*
```

```
 * 获取树的高度
 */
private int height(AVLTreeNode<T> tree) {
    if (tree != null)
        return tree.height;

    return 0;
}

public int height() {
    return height(mRoot);
}

/*
 * 比较两个值的大小
 */
private int max(int a, int b) {
    return a>b ? a : b;
}

/*
 * LL: 左左对应的情况(左单旋转)。
 *
 * 返回值: 旋转后的根节点
 */
private AVLTreeNode<T> leftLeftRotation(AVLTreeNode<T> k2) {
    AVLTreeNode<T> k1;

    k1 = k2.left;
    k2.left = k1.right;
    k1.right = k2;

    k2.height = max( height(k2.left), height(k2.right)) + 1;
    k1.height = max( height(k1.left), k2.height) + 1;

    return k1;
}

/*
 * RR: 右右对应的情况(右单旋转)。
 *
 * 返回值: 旋转后的根节点
 */
private AVLTreeNode<T> rightRightRotation(AVLTreeNode<T> k1) {
    AVLTreeNode<T> k2;

    k2 = k1.right;
    k1.right = k2.left;
    k2.left = k1;

    k1.height = max( height(k1.left), height(k1.right)) + 1;
    k2.height = max( height(k2.right), k1.height) + 1;

    return k2;
}

/*
 * LR: 左右对应的情况(左双旋转)。
```

```
     *
     * 返回值: 旋转后的根节点
     */
    private AVLTreeNode<T> leftRightRotation(AVLTreeNode<T> k3) {
        k3.left = rightRightRotation(k3.left);

        return leftLeftRotation(k3);
    }

    /*
     * RL: 右左对应的情况(右双旋转)。
     *
     * 返回值: 旋转后的根节点
     */
    private AVLTreeNode<T> rightLeftRotation(AVLTreeNode<T> k1) {
        k1.right = leftLeftRotation(k1.right);

        return rightRightRotation(k1);
    }



    /*
     * 将结点插入到AVL树中，并返回根节点
     *
     * 参数说明:
     *     tree AVL树的根结点
     *     key 插入的结点的键值
     * 返回值:
     *     根节点
     */
    private AVLTreeNode<T> insert(AVLTreeNode<T> tree, T key) {
        if (tree == null) {
            // 新建节点
            tree = new AVLTreeNode<T>(key, null, null);
            if (tree==null) {
                System.out.println("ERROR: create avltree node failed!");
                return null;
            }
        } else {
            int cmp = key.compareTo(tree.key);

               if (cmp < 0) {    // 应该将key插入到"tree的左子树"的情况
                tree.left = insert(tree.left, key);
                // 插入节点后，若AVL树失去平衡，则进行相应的调节。
                if (height(tree.left) - height(tree.right) == 2) {
                    if (key.compareTo(tree.left.key) < 0)
                        tree = leftLeftRotation(tree);
                    else
                        tree = leftRightRotation(tree);
                }
            } else if (cmp > 0) {    // 应该将key插入到"tree的右子树"的情况
                tree.right = insert(tree.right, key);
                // 插入节点后，若AVL树失去平衡，则进行相应的调节。
                if (height(tree.right) - height(tree.left) == 2) {
                    if (key.compareTo(tree.right.key) > 0)
                        tree = rightRightRotation(tree);
                    else
                        tree = rightLeftRotation(tree);
```

```
            }
        } else {    // cmp==0
            System.out.println("添加失败: 不允许添加相同的节点! ");
        }
    }

    tree.height = max( height(tree.left), height(tree.right)) + 1;

    return tree;
}

public void insert(T key) {
    mRoot = insert(mRoot, key);
}

/*
 * 删除结点(z)，返回根节点
 *
 * 参数说明:
 *     tree AVL树的根结点
 *     z 待删除的结点
 * 返回值:
 *     根节点
 */
private AVLTreeNode<T> remove(AVLTreeNode<T> tree, AVLTreeNode<T> z) {
    // 根为空 或者 没有要删除的节点，直接返回null。
    if (tree==null || z==null)
        return null;

    int cmp = z.key.compareTo(tree.key);
    if (cmp < 0) {        // 待删除的节点在"tree的左子树"中
        tree.left = remove(tree.left, z);
        // 删除节点后，若AVL树失去平衡，则进行相应的调节。
        if (height(tree.right) - height(tree.left) == 2) {
            AVLTreeNode<T> r =  tree.right;
            if (height(r.left) > height(r.right))
                tree = rightLeftRotation(tree);
            else
                tree = rightRightRotation(tree);
        }
    } else if (cmp > 0) {    // 待删除的节点在"tree的右子树"中
        tree.right = remove(tree.right, z);
        // 删除节点后，若AVL树失去平衡，则进行相应的调节。
        if (height(tree.left) - height(tree.right) == 2) {
            AVLTreeNode<T> l =  tree.left;
            if (height(l.right) > height(l.left))
                tree = leftRightRotation(tree);
            else
                tree = leftLeftRotation(tree);
        }
    } else {    // tree是对应要删除的节点。
        // tree的左右孩子都非空
        if ((tree.left!=null) && (tree.right!=null)) {
            if (height(tree.left) > height(tree.right)) {
                // 如果tree的左子树比右子树高；
                // 则(01)找出tree的左子树中的最大节点
                //   (02)将该最大节点的值赋值给tree。
                //   (03)删除该最大节点。
```

```
                // 这类似于用"tree的左子树中最大节点"做"tree"的替身；
                // 采用这种方式的好处是：删除"tree的左子树中最大节点"之后，AVL树仍然是平衡
的。
                AVLTreeNode<T> max = maximum(tree.left);
                tree.key = max.key;
                tree.left = remove(tree.left, max);
            } else {
                // 如果tree的左子树不比右子树高(即它们相等，或右子树比左子树高1)
                // 则(01)找出tree的右子树中的最小节点
                //   (02)将该最小节点的值赋值给tree。
                //   (03)删除该最小节点。
                // 这类似于用"tree的右子树中最小节点"做"tree"的替身；
                // 采用这种方式的好处是：删除"tree的右子树中最小节点"之后，AVL树仍然是平衡
的。
                AVLTreeNode<T> min = maximum(tree.right);
                tree.key = min.key;
                tree.right = remove(tree.right, min);
            }
        } else {
            AVLTreeNode<T> tmp = tree;
            tree = (tree.left!=null) ? tree.left : tree.right;
            tmp = null;
        }
    }

    return tree;
}

public void remove(T key) {
    AVLTreeNode<T> z;

    if ((z = search(mRoot, key)) != null)
        mRoot = remove(mRoot, z);
}
```

哈夫曼树

```
public class HuffmanNode implements Comparable, Cloneable {
    protected int key;              // 权值
    protected HuffmanNode left;     // 左孩子
    protected HuffmanNode right;    // 右孩子
    protected HuffmanNode parent;   // 父结点

    protected HuffmanNode(int key, HuffmanNode left, HuffmanNode right,
HuffmanNode parent) {
        this.key = key;
        this.left = left;
        this.right = right;
        this.parent = parent;
    }

    @Override
    public Object clone() {
        Object obj=null;
```

```
        try {
            obj = (HuffmanNode)super.clone();//Object 中的clone()识别出你要复制的是哪
一个对象。
        } catch(CloneNotSupportedException e) {
            System.out.println(e.toString());
        }

        return obj;
    }

    @Override
    public int compareTo(Object obj) {
        return this.key - ((HuffmanNode)obj).key;
    }
}

import java.util.List;
import java.util.ArrayList;
import java.util.Collections;

public class Huffman {

    private HuffmanNode mRoot;  // 根结点

    /*
     * 创建Huffman树
     *
     * @param 权值数组
     */
    public Huffman(int a[]) {
        HuffmanNode parent = null;
        MinHeap heap;

        // 建立数组a对应的最小堆
        heap = new MinHeap(a);

        for(int i=0; i<a.length-1; i++) {
            HuffmanNode left = heap.dumpFromMinimum();  // 最小节点是左孩子
            HuffmanNode right = heap.dumpFromMinimum(); // 其次才是右孩子

            // 新建parent节点，左右孩子分别是left/right；
            // parent的大小是左右孩子之和
            parent = new HuffmanNode(left.key+right.key, left, right, null);
            left.parent = parent;
            right.parent = parent;

            // 将parent节点数据拷贝到"最小堆"中
            heap.insert(parent);
        }

        mRoot = parent;

        // 销毁最小堆
        heap.destroy();
    }

    /*
```

```
     * 前序遍历"Huffman树"
     */
    private void preOrder(HuffmanNode tree) {
        if(tree != null) {
            System.out.print(tree.key+" ");
            preOrder(tree.left);
            preOrder(tree.right);
        }
    }

    public void preOrder() {
        preOrder(mRoot);
    }

    /*
     * 中序遍历"Huffman树"
     */
    private void inOrder(HuffmanNode tree) {
        if(tree != null) {
            inOrder(tree.left);
            System.out.print(tree.key+" ");
            inOrder(tree.right);
        }
    }

    public void inOrder() {
        inOrder(mRoot);
    }


    /*
     * 后序遍历"Huffman树"
     */
    private void postOrder(HuffmanNode tree) {
        if(tree != null)
        {
            postOrder(tree.left);
            postOrder(tree.right);
            System.out.print(tree.key+" ");
        }
    }

    public void postOrder() {
        postOrder(mRoot);
    }

    /*
     * 销毁Huffman树
     */
    private void destroy(HuffmanNode tree) {
        if (tree==null)
            return ;

        if (tree.left != null)
            destroy(tree.left);
        if (tree.right != null)
            destroy(tree.right);
```

```
        tree=null;
    }

    public void destroy() {
        destroy(mRoot);
        mRoot = null;
    }

    /*
     * 打印"Huffman树"
     *
     * key         -- 节点的键值
     * direction  --  0，表示该节点是根节点;
     *               -1，表示该节点是它的父结点的左孩子;
     *                1，表示该节点是它的父结点的右孩子。
     */
    private void print(HuffmanNode tree, int key, int direction) {

        if(tree != null) {

            if(direction==0)    // tree是根节点
                System.out.printf("%2d is root\n", tree.key);
            else                // tree是分支节点
                System.out.printf("%2d is %2d's %6s child\n", tree.key, key,
direction==1?"right" : "left");

            print(tree.left, tree.key, -1);
            print(tree.right,tree.key,  1);
        }
    }

    public void print() {
        if (mRoot != null)
            print(mRoot, mRoot.key, 0);
    }
}
```

邻接矩阵实现的无向图的bfs和dfs

```
import java.io.IOException;
import java.util.Scanner;

public class MatrixUDG {

    private char[] mVexs;       // 顶点集合
    private int[][] mMatrix;    // 邻接矩阵

    /*
     * 创建图(自己输入数据)
     */
    public MatrixUDG() {
```

```
        // 输入"顶点数"和"边数"
        System.out.printf("input vertex number: ");
        int vlen = readInt();
        System.out.printf("input edge number: ");
        int elen = readInt();
        if ( vlen < 1 || elen < 1 || (elen > (vlen*(vlen - 1)))) {
            System.out.printf("input error: invalid parameters!\n");
            return ;
        }

        // 初始化"顶点"
        mVexs = new char[vlen];
        for (int i = 0; i < mVexs.length; i++) {
            System.out.printf("vertex(%d): ", i);
            mVexs[i] = readChar();
        }

        // 初始化"边"
        mMatrix = new int[vlen][vlen];
        for (int i = 0; i < elen; i++) {
            // 读取边的起始顶点和结束顶点
            System.out.printf("edge(%d):", i);
            char c1 = readChar();
            char c2 = readChar();
            int p1 = getPosition(c1);
            int p2 = getPosition(c2);

            if (p1==-1 || p2==-1) {
                System.out.printf("input error: invalid edge!\n");
                return ;
            }

            mMatrix[p1][p2] = 1;
            mMatrix[p2][p1] = 1;
        }
    }

    /*
     * 创建图(用已提供的矩阵)
     *
     * 参数说明:
     *     vexs  -- 顶点数组
     *     edges -- 边数组
     */
    public MatrixUDG(char[] vexs, char[][] edges) {

        // 初始化"顶点数"和"边数"
        int vlen = vexs.length;
        int elen = edges.length;

        // 初始化"顶点"
        mVexs = new char[vlen];
        for (int i = 0; i < mVexs.length; i++)
            mVexs[i] = vexs[i];

        // 初始化"边"
        mMatrix = new int[vlen][vlen];
        for (int i = 0; i < elen; i++) {
```

```
            // 读取边的起始顶点和结束顶点
            int p1 = getPosition(edges[i][0]);
            int p2 = getPosition(edges[i][1]);

            mMatrix[p1][p2] = 1;
            mMatrix[p2][p1] = 1;
        }
    }

    /*
     * 返回ch位置
     */
    private int getPosition(char ch) {
        for(int i=0; i<mVexs.length; i++)
            if(mVexs[i]==ch)
                return i;
        return -1;
    }

    /*
     * 读取一个输入字符
     */
    private char readChar() {
        char ch='0';

        do {
            try {
                ch = (char)System.in.read();
            } catch (IOException e) {
                e.printStackTrace();
            }
        } while(!((ch>='a'&&ch<='z') || (ch>='A'&&ch<='Z')));

        return ch;
    }

    /*
     * 读取一个输入字符
     */
    private int readInt() {
        Scanner scanner = new Scanner(System.in);
        return scanner.nextInt();
    }

    /*
     * 返回顶点v的第一个邻接顶点的索引，失败则返回-1
     */
    private int firstVertex(int v) {

        if (v<0 || v>(mVexs.length-1))
            return -1;

        for (int i = 0; i < mVexs.length; i++)
            if (mMatrix[v][i] == 1)
                return i;

        return -1;
    }
```

```
    /*
     * 返回顶点v相对于w的下一个邻接顶点的索引，失败则返回-1
     */
    private int nextVertex(int v, int w) {

        if (v<0 || v>(mVexs.length-1) || w<0 || w>(mVexs.length-1))
            return -1;

        for (int i = w + 1; i < mVexs.length; i++)
            if (mMatrix[v][i] == 1)
                return i;

        return -1;
    }

    /*
     * 深度优先搜索遍历图的递归实现
     */
    private void DFS(int i, boolean[] visited) {

        visited[i] = true;
        System.out.printf("%c ", mVexs[i]);
        // 遍历该顶点的所有邻接顶点。若是没有访问过，那么继续往下走
        for (int w = firstVertex(i); w >= 0; w = nextVertex(i, w)) {
            if (!visited[w])
                DFS(w, visited);
        }
    }

    /*
     * 深度优先搜索遍历图
     */
    public void DFS() {
        boolean[] visited = new boolean[mVexs.length];       // 顶点访问标记

        // 初始化所有顶点都没有被访问
        for (int i = 0; i < mVexs.length; i++)
            visited[i] = false;

        System.out.printf("DFS: ");
        for (int i = 0; i < mVexs.length; i++) {
            if (!visited[i])
                DFS(i, visited);
        }
        System.out.printf("\n");
    }

    /*
     * 广度优先搜索（类似于树的层次遍历）
     */
    public void BFS() {
        int head = 0;
        int rear = 0;
        int[] queue = new int[mVexs.length];            // 辅组队列
        boolean[] visited = new boolean[mVexs.length];  // 顶点访问标记

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

```
            visited[i] = false;

        System.out.printf("BFS: ");
        for (int i = 0; i < mVexs.length; i++) {
            if (!visited[i]) {
                visited[i] = true;
                System.out.printf("%c ", mVexs[i]);
                queue[rear++] = i;  // 入队列
            }

            while (head != rear) {
                int j = queue[head++];  // 出队列
                for (int k = firstVertex(j); k >= 0; k = nextVertex(j, k)) { //k
是为访问的邻接顶点
                    if (!visited[k]) {
                        visited[k] = true;
                        System.out.printf("%c ", mVexs[k]);
                        queue[rear++] = k;
                    }
                }
            }
        }
        System.out.printf("\n");
    }

    /*
     * 打印矩阵队列图
     */
    public void print() {
        System.out.printf("Martix Graph:\n");
        for (int i = 0; i < mVexs.length; i++) {
            for (int j = 0; j < mVexs.length; j++)
                System.out.printf("%d ", mMatrix[i][j]);
            System.out.printf("\n");
        }
    }

    public static void main(String[] args) {
        char[] vexs = {'A', 'B', 'C', 'D', 'E', 'F', 'G'};
        char[][] edges = new char[][]{
            {'A', 'C'},
            {'A', 'D'},
            {'A', 'F'},
            {'B', 'C'},
            {'C', 'D'},
            {'E', 'G'},
            {'F', 'G'}};
        MatrixUDG pG;

        // 自定义"图"(输入矩阵队列)
        //pG = new MatrixUDG();
        // 采用已有的"图"
        pG = new MatrixUDG(vexs, edges);

        pG.print();   // 打印图
        pG.DFS();     // 深度优先遍历
        pG.BFS();     // 广度优先遍历
    }
```

```
}
```

邻接矩阵实现的有向图的bfs和dfs

```
import java.io.IOException;
import java.util.Scanner;

public class MatrixDG {

    private char[] mVexs;       // 顶点集合
    private int[][] mMatrix;    // 邻接矩阵

    /*
     * 创建图(自己输入数据)
     */
    public MatrixDG() {

        // 输入"顶点数"和"边数"
        System.out.printf("input vertex number: ");
        int vlen = readInt();
        System.out.printf("input edge number: ");
        int elen = readInt();
        if ( vlen < 1 || elen < 1 || (elen > (vlen*(vlen - 1)))) {
            System.out.printf("input error: invalid parameters!\n");
            return ;
        }

        // 初始化"顶点"
        mVexs = new char[vlen];
        for (int i = 0; i < mVexs.length; i++) {
            System.out.printf("vertex(%d): ", i);
            mVexs[i] = readChar();
        }

        // 初始化"边"
        mMatrix = new int[vlen][vlen];
        for (int i = 0; i < elen; i++) {
            // 读取边的起始顶点和结束顶点
            System.out.printf("edge(%d):", i);
            char c1 = readChar();
            char c2 = readChar();
            int p1 = getPosition(c1);
            int p2 = getPosition(c2);

            if (p1==-1 || p2==-1) {
                System.out.printf("input error: invalid edge!\n");
                return ;
            }

            mMatrix[p1][p2] = 1;
        }
    }

    /*
```

```
     * 创建图(用已提供的矩阵)
     *
     * 参数说明:
     *     vexs  -- 顶点数组
     *     edges -- 边数组
     */
    public MatrixDG(char[] vexs, char[][] edges) {

        // 初始化"顶点数"和"边数"
        int vlen = vexs.length;
        int elen = edges.length;

        // 初始化"顶点"
        mVexs = new char[vlen];
        for (int i = 0; i < mVexs.length; i++)
            mVexs[i] = vexs[i];

        // 初始化"边"
        mMatrix = new int[vlen][vlen];
        for (int i = 0; i < elen; i++) {
            // 读取边的起始顶点和结束顶点
            int p1 = getPosition(edges[i][0]);
            int p2 = getPosition(edges[i][1]);

            mMatrix[p1][p2] = 1;
        }
    }

    /*
     * 返回ch位置
     */
    private int getPosition(char ch) {
        for(int i=0; i<mVexs.length; i++)
            if(mVexs[i]==ch)
                return i;
        return -1;
    }

    /*
     * 读取一个输入字符
     */
    private char readChar() {
        char ch='0';

        do {
            try {
                ch = (char)System.in.read();
            } catch (IOException e) {
                e.printStackTrace();
            }
        } while(!((ch>='a'&&ch<='z') || (ch>='A'&&ch<='Z')));

        return ch;
    }

    /*
     * 读取一个输入字符
     */
```

```
    private int readInt() {
        Scanner scanner = new Scanner(System.in);
        return scanner.nextInt();
    }

    /*
     * 返回顶点v的第一个邻接顶点的索引，失败则返回-1
     */
    private int firstVertex(int v) {

        if (v<0 || v>(mVexs.length-1))
            return -1;

        for (int i = 0; i < mVexs.length; i++)
            if (mMatrix[v][i] == 1)
                return i;

        return -1;
    }

    /*
     * 返回顶点v相对于w的下一个邻接顶点的索引，失败则返回-1
     */
    private int nextVertex(int v, int w) {

        if (v<0 || v>(mVexs.length-1) || w<0 || w>(mVexs.length-1))
            return -1;

        for (int i = w + 1; i < mVexs.length; i++)
            if (mMatrix[v][i] == 1)
                return i;

        return -1;
    }

    /*
     * 深度优先搜索遍历图的递归实现
     */
    private void DFS(int i, boolean[] visited) {

        visited[i] = true;
        System.out.printf("%c ", mVexs[i]);
        // 遍历该顶点的所有邻接顶点。若是没有访问过，那么继续往下走
        for (int w = firstVertex(i); w >= 0; w = nextVertex(i, w)) {
            if (!visited[w])
                DFS(w, visited);
        }
    }

    /*
     * 深度优先搜索遍历图
     */
    public void DFS() {
        boolean[] visited = new boolean[mVexs.length];       // 顶点访问标记

        // 初始化所有顶点都没有被访问
        for (int i = 0; i < mVexs.length; i++)
            visited[i] = false;
```

```
        System.out.printf("DFS: ");
        for (int i = 0; i < mVexs.length; i++) {
            if (!visited[i])
                DFS(i, visited);
        }
        System.out.printf("\n");
    }

    /*
     * 广度优先搜索（类似于树的层次遍历）
     */
    public void BFS() {
        int head = 0;
        int rear = 0;
        int[] queue = new int[mVexs.length];            // 辅组队列
        boolean[] visited = new boolean[mVexs.length];  // 顶点访问标记

        for (int i = 0; i < mVexs.length; i++)
            visited[i] = false;

        System.out.printf("BFS: ");
        for (int i = 0; i < mVexs.length; i++) {
            if (!visited[i]) {
                visited[i] = true;
                System.out.printf("%c ", mVexs[i]);
                queue[rear++] = i;  // 入队列
            }

            while (head != rear) {
                int j = queue[head++];  // 出队列
                for (int k = firstVertex(j); k >= 0; k = nextVertex(j, k)) { //k
是为访问的邻接顶点
                    if (!visited[k]) {
                        visited[k] = true;
                        System.out.printf("%c ", mVexs[k]);
                        queue[rear++] = k;
                    }
                }
            }
        }
        System.out.printf("\n");
    }

    /*
     * 打印矩阵队列图
     */
    public void print() {
        System.out.printf("Martix Graph:\n");
        for (int i = 0; i < mVexs.length; i++) {
            for (int j = 0; j < mVexs.length; j++)
                System.out.printf("%d ", mMatrix[i][j]);
            System.out.printf("\n");
        }
    }

    public static void main(String[] args) {
        char[] vexs = {'A', 'B', 'C', 'D', 'E', 'F', 'G'};
```

```
        char[][] edges = new char[][]{
            {'A', 'B'},
            {'B', 'C'},
            {'B', 'E'},
            {'B', 'F'},
            {'C', 'E'},
            {'D', 'C'},
            {'E', 'B'},
            {'E', 'D'},
            {'F', 'G'}};
        MatrixDG pG;

        // 自定义"图"(输入矩阵队列)
        //pG = new MatrixDG();
        // 采用已有的"图"
        pG = new MatrixDG(vexs, edges);

        pG.print();   // 打印图
        pG.DFS();     // 深度优先遍历
        pG.BFS();     // 广度优先遍历
    }
}
```

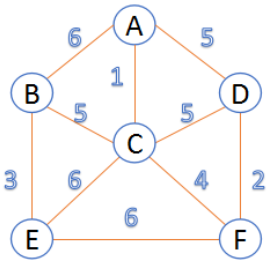
最小生成树的两种算法（只有方法没代码）

最小生成树算法

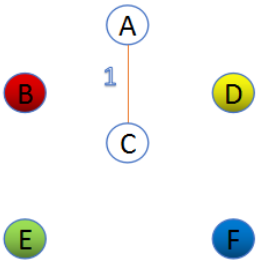
¶ Kruskal算法

此算法可以称为“加边法”，初始最小生成树边数为0，每迭代一次就选择一条满足条件的最小代价边，加入到最小生成树的边集合里。

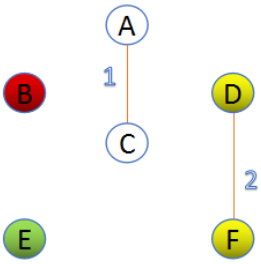
1. 把图中的所有边按代价从小到大排序；
2. 把图中的n个顶点看成独立的n棵树组成的森林；
3. 按权值从小到大选择边，所选的边连接的两个顶点ui,viui,vi,应属于两颗不同的树，则成为最小生成树的一条边，并将这两颗树合并作为一颗树。
4. 重复(3),直到所有顶点都在一颗树内或者有n-1条边为止。



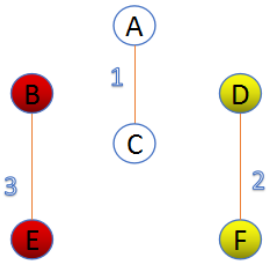
连通网G



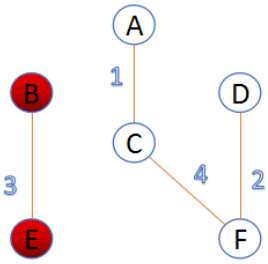
1.选择代价最小的边(A,C);并保证A,C不在同一颗树上,然后合并A,C



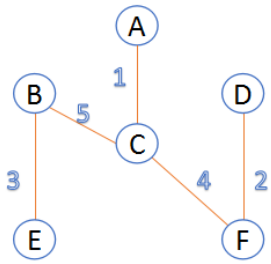
2.选择代价最小的边(D,F);并保证D,F不在同一颗树上,然后合并D,F



3.选择代价最小的边(B,E);并保证B,E不在同一颗树上,然后合并B,E



4.选择代价最小的边(C,F);并保证C,F不在同一颗树上,然后合并C,F所在的树



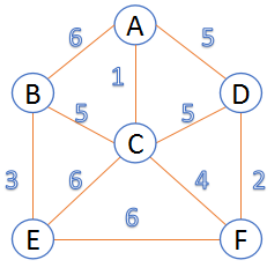
5.选择代价最小的边(A,D),顶点A,D在同一颗树上,丢弃;
选择最小的边(C,D),顶点C,D在同一颗树上,丢弃;
选择最小的边(B,C),顶点B,C不在同一颗树上,加入此边,然后合并B,C所在的树,此时所有顶点在同一颗树上，返回;

¶ Prim算法

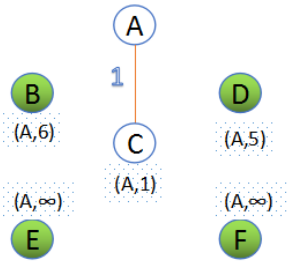
此算法可以称为“加点法”，每次迭代选择代价最小的边对应的点，加入到最小生成树中。算法从某一个顶点s开始，逐渐长大覆盖整个连通网的所有顶点。

- 图的所有顶点集合为VV；初始令集合u={s},v=V−uu={s},v=V−u;
- 在两个集合u,vu,v能够组成的边中，选择一条代价最小的边(u0,v0)(u0,v0)，加入到最小生成树中，并把v0v0并入到集合u中。
- 重复上述步骤，直到最小生成树有n-1条边或者n个顶点为止。

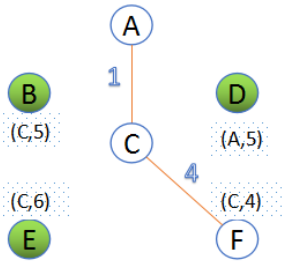
由于不断向集合u中加点，所以最小代价边必须同步更新；需要建立一个辅助数组closedge,用来维护集合v中每个顶点与集合u中最小代价边信息，：



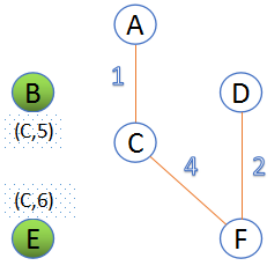
连通网G



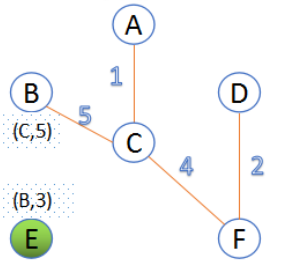
1.初始u={A},v={B,C,D,E,F};顶点B下方(A,6),表示与集合u中A的代价为6作为最小代价边。选择最小的代价边(A,C),把C并入到集合u中。



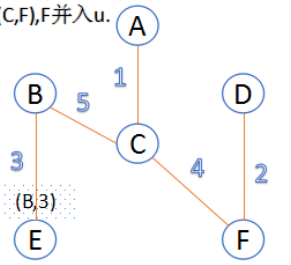
2.u={A,C},v={B,D,E,F};更新v中顶点与集合u的最小的代价边;例如:顶点E之前为(A,∞),更新为(C,6)；选择最小代价边(C,F),F并入u.



3.u={A,C,F},v={B,D,E};更新v中顶点与集合u的最小的代价边;选择最小代价边(F,D),D并入u.



4.u={A,C,F,D},v={B,E};更新v中顶点与集合u的最小的代价边;选择最小代价边(C,B),B并入u.



5.u={A,C,F,D,B},v={E};更新v中顶点与集合u的最小的代价边;选择最小代价边(B,E),E并入u.

¶ 总结

因为Kruskal涉及大量对边的操作，所以它适用于稀疏图；普通的prim算法适用于稠密图，但堆优化的prim算法更适用于稀疏图，因为其时间复杂度是由边的数量决定的。

##

最短路径（只有方法没有代码）

路径长度最短的最短路径的特点:

- 在这条路径上，必定只含一条弧，并且这条弧的权值最小。
- 下一条路径长度次短的最短路径的特点:
- 它只可能有两种情况: 或者是直接从源点到该点(只含一条弧)； 或者是从源点经过顶点v1，再到达该顶点(由两条弧组成)。

问题解法:

- 求从某个源点到其余各点的最短路径 — Dijkstra算法
- 每一对顶点之间的最短路径 — Floyd算法

¶ 最短路径算法

¶ Dijkstra算法

1.定义概览

Dijkstra(迪杰斯特拉)算法是典型的单源最短路径算法，用于计算一个节点到其他所有节点的最短路径。主要特点是以起始点为中心向外层层扩展，直到扩展到终点为止。Dijkstra算法是很有代表性的最短路径算法，在很多专业课程中都作为基本内容有详细的介绍，如数据结构，图论，运筹学等等。注意该算法要求图中不存在负权边。

问题描述: 在无向图 G=(V,E) 中，假设每条边 E[i] 的长度为 w[i]，找到由顶点 V0 到其余各点的最短路径。(单源最短路径)

2.算法描述

1)算法思想: 设G=(V,E)是一个带权有向图，把图中顶点集合V分成两组，第一组为已求出最短路径的顶点集合(用S表示，初始时S中只有一个源点，以后每求得一条最短路径 , 就将加入到集合S中，直到全部顶点都加入到S中，算法就结束了)，第二组为其余未确定最短路径的顶点集合(用U表示)，按最短路径长度的递增次序依次把第二组的顶点加入S中。在加入的过程中，总保持从源点v到S中各顶点的最短路径长度不大于从源点v到U中任何顶点的最短路径长度。此外，每个顶点对应一个距离，S中的顶点的距离就是从v到此顶点的最短路径长度，U中的顶点的距离，是从v到此顶点只包括S中的顶点为中间顶点的当前最短路径长度。

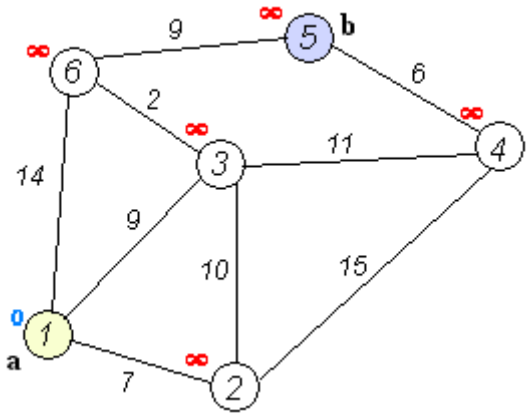
2)算法步骤:

a.初始时，S只包含源点，即S = {v}，v的距离为0。U包含除v外的其他顶点，即:U={其余顶点}，若v与U中顶点u有边，则<u,v>正常有权值，若u不是v的出边邻接点，则<u,v>权值为∞。

b.从U中选取一个距离v最小的顶点k，把k，加入S中(该选定的距离就是v到k的最短路径长度)。

c.以k为新考虑的中间点，修改U中各顶点的距离；若从源点v到顶点u的距离(经过顶点k)比原来距离(不经过顶点k)短，则修改顶点u的距离值，修改后的距离值的顶点k的距离加上边上的权。

d.重复步骤b和c直到所有顶点都包含在S中。



###

拓扑排序（无向有环图才能用）

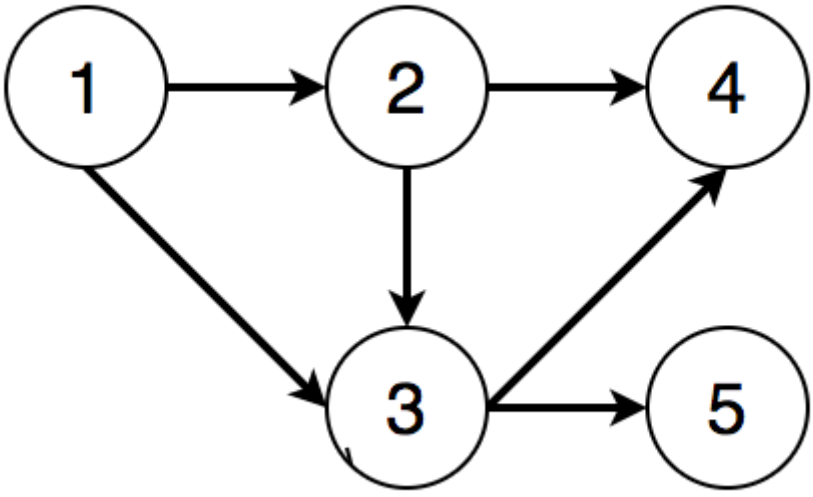
拓扑排序前提

当且仅当一个有向图为有向无环图(directed acyclic graph，或称DAG)时，才能得到对应于该图的拓扑排序。这里有两点要注意:

- 对于有环图，必然会造成循环依赖(circular dependency)，不符合拓扑排序定义；
- 对于每一个有向无环图都至少存在一种拓扑排序；

`不唯一的情况`:

上图中若我们删 4、5结点之前的有向边，上图变为如下所示:



则我们可得到两个不同的拓扑排序结果: [1, 2, 3, 4, 5]和[1, 2, 3, 5, 4]。

¶ 拓扑排序算法

为了说明如何得到一个有向无环图的拓扑排序，我们首先需要了解有向图结点的入度(indegree)和出度(outdegree)的概念。

假设有向图中不存在起点和终点为同一结点的有向边。

入度: 设有向图中有一结点v，其入度即为当前所有从其他结点出发，终点为v的的边的数目。也就是所有指向v的有向边的数目。

出度: 设有向图中有一结点v，其出度即为当前所有起点为v，指向其他结点的边的数目。也就是所有由v发出的边的数目。

在了解了入度和出度的概念之后，再根据拓扑排序的定义，我们自然就能够得出结论: 要想完成拓扑排序，我们每次都应当从入度为0的结点开始遍历。因为只有入度为0的结点才能够成为拓扑排序的起点。否则根据拓扑排序的定义，只要一个结点v的入度不为0，则至少有一条边起始于其他结点而指向v，那么这条边的起点在拓扑排序的顺序中应当位于v之前，则v不能成为当前遍历的起点。

由此我们可以进一步得出一个改进的深度优先遍历或广度优先遍历算法来完成拓扑排序。以广度优先遍历为例，这一改进后的算法与普通的广度优先遍历唯一的区别在于我们应当保存每一个结点对应的入度，并在遍历的每一层选取入度为0的结点开始遍历(而普通的广度优先遍历则无此限制，可以从该吃呢个任意一个结点开始遍历)。这个算法描述如下:

- 初始化一个int[] inDegree保存每一个结点的入度。
- 对于图中的每一个结点的子结点，将其子结点的入度加1。
- 选取入度为0的结点开始遍历，并将该节点加入输出。
- 对于遍历过的每个结点，更新其子结点的入度: 将子结点的入度减1。
- 重复步骤3，直到遍历完所有的结点。
- 如果无法遍历完所有的结点，则意味着当前的图不是有向无环图。不存在拓扑排序。

¶ 拓扑排序代码实现

广度优先遍历拓扑排序的Java代码如下:

```
public class TopologicalSort {
    /**
     * Get topological ordering of the input directed graph
     * @param n number of nodes in the graph
     * @param adjacencyList adjacency list representation of the input directed
graph
     * @return topological ordering of the graph stored in an List<Integer>.
     */
    public List<Integer> topologicalSort(int n, int[][] adjacencyList) {
        List<Integer> topoRes = new ArrayList<>();
        int[] inDegree = new int[n];
        for (int[] parent : adjacencyList) {
            for (int child : parent) {
                inDegree[child]++;
            }
        }

        Deque<Integer> deque = new ArrayDeque<>();

        // start from nodes whose indegree are 0
        for (int i = 0; i < n; i++) {
            if (inDegree[i] == 0) deque.offer(i);
        }

        while (!deque.isEmpty()) {
            int curr = deque.poll();
            topoRes.add(curr);
            for (int child : adjacencyList[curr]) {
                inDegree[child]--;
                if (inDegree[child] == 0) {
```

```
                    deque.offer(child);
                }
            }
        }

        return topoRes.size() == n ? topoRes : new ArrayList<>();
    }
}
```

1二分查找

1归并

3使用java自带排序

4 java内置的优先队列

6二叉树+前中后序遍历+查找key节点+找最小最大节点+找前驱后继节点+插入删除节点+打印二叉树

12平衡二叉树

16哈夫曼树

19邻接矩阵实现的无向图的bfs和dfs

24邻接矩阵实现的有向图的bfs和dfs

28最小生成树的两种算法（只有方法没代码）

30最短路径（只有方法没有代码）

31拓扑排序（无向有环图才能用）