数据结构与算法

1.链表

1).单链表的长度

```
    // 链表节点的创建
 public class Node<T> {
       public T data;// 数据域，保存数据元素
       public Node<T> next;// 地址域，引用后继节点
       //初始化单链表
       public Node(T data, Node<T> next) {
            this.data = data;
            this.next = next;
       }
       //空的单链表
       public Node() {
            this(null, null);
       }
 }
    //单链表的长度
public int length(Node head) {
    int i = 0;
    Node p = head;// p 从单聊表第一个节点开始
    while (p != null) {// 若单链表未结束
         i++;
         p = p.next;// p 到达其后继节点
    }
    return i;
}
```

2).单链表的增查操作，删除结点操作(O(n)和 O(1)的时间复杂度)

```
    //单链表的增加元素
// 将 x 对象插入在序号为 i 节点前
public void insert(int i, Node x) {
    if (x == null)
         return;// 不能插入空对象
    Node p = head;// p 指向头结点
    // 寻找插入位置
    for (int j = 0; p.next != null && j < i-1; j++)
         p = p.next;// 循环停止时,p 指向第 i-1 节点或最后一个节点
    // 插入 x 作为 p 节点
    p.next = x;
    x.next = p;
}
```

```
    //删除节点: O(1)
public static void deleteNode(ListNode head,ListNode node){
    //删除尾节点，采用顺序查找找到尾节点的前一节点
    if(node.next==null){
        while(head.next!=node){
            head=head.next;
        }
        head.next=null;
    }
    //要删除的节点是头结点
    else if(head==node){
        head=null;
    }
    //要删除的节点是中间普通节点: 把要删除的节点的后一个节点覆盖在要删除的节点上
    else{
        node.data=node.next.data;
        node.next=node.next.next;
    }
}
```

3).逆序创建单链表

```
    //逆序创建单链表
public static void createList(Node n, int[] a){
    n = null;
    for(int i = n -1; i>=0; --i){
        s = new Node();
        s.data = A[i];
        s.next = n;
        n = s;
    }
}
```

4).对链表排序

```
    //链表的排序
 public static void quickSort(Node begin, Node end){
    if(begin == null || begin == end)
        return;

    Node index = paration(begin, end);
    quickSort(begin, index);
    quickSort(index.next, end);
}
public static ListNode paration(Node begin, Node end){
    if(begin == null || begin == end)
```

```
            return begin;
        int val = begin.val;   //基准元素
        Node index = begin, cur = begin.next;
        while(cur != end){
            if(cur.val < val){   //交换
                index = index.next;
                int tmp = cur.val;
                cur.val = index.val;
                index.val = tmp;
            }
            cur = cur.next;
        }
        begin.val = index.val;
        index.val = val;

        return index;
}
```

5).删除链表中重复元素

```
    //删除链表中重复的元素
  //1.额外空间,用 HashMap 存储链表的节点以及个数
  public static void deleteDuplecate(Node head){
        HashMap<Integer, Integer> h = new HashMap();
        Node temp = head;
        Node pre = null;
        while(temp != null){
            if(h.containsKey(temp.data)){
                pre.next = temp.next
            }
        }
  }
  //2.不要用额外空间
   public ListNode deleteDuplication(Node pHead)
  {
if (pHead == null) {
    return null;
}
Node preNode = null;
Node node = pHead;
while (node != null) {
    if (node.next != null && node.val == node.next.val) {
        int value = node.val;
        while (node.next != null && node.next.val == value) {
            node = node.next;
```

```
            }
            if (preNode == null) {
                pHead = node.next;
            } else {
                preNode.next = node.next;
            }
        } else {
            preNode = node;
        }
                node = node.next;
}
return pHead;
    }
```

6).找出单链表中倒数第 k 个结点

```
    //单链表中倒数第 k 个节点:快慢指针,要求只能遍历一次单链表，可以设两个指针 p 和 q，
//最开始时它们都指向头结点，然后 p 向后移动 k 位，最后 p,q 同时向后移动直到 p 为最后一个结点，那么此时 q 即为所求。
public LNode reciprocalKNode(Node L, int k) {
    if (k < 0) {
        System.out.println("k 不可以为负数");
        return null;
    }
    if (L == null) {
        System.out.println("单链表为空");
        return null;
    }
    Node p = L;
    Node q = L;
    while (k > 0) {
        p = p.next;
        if (p == null) {
            System.out.println("单链表太短，不存在倒数第 k 个结点");
            return null;
        }
    }
    while (p.next != null) {
        p = p.next;
        q = q.next;
    }
    return p;
}
```

7).链表反转

```
    //链表反转:不用额外空间，记录下链表的断开位置
public static Node ReverseList(Node head) {

    if(head == null){
        return null;
    }
    Node rHead = null;//逆转后的头节点
    Node prior = null;//store prior,前一个节点
    Node q = head;//store current，记录当前节点

    while(q != null){
        Node next = q.next;//store the next

        if(next == null){
            rHead = q;
        }

        q.next = prior;

        prior = q;
        q = next;
    }

    return rHead;
}
```

8).从尾到头输出链表

```
    //从尾到头输出链表:递归(运用栈的特性)
public static void printList(Node head){
    if(head !=null){
        printList(head.next);
        System.out.println(head.data);
    }
}
```

9).寻找单链表的中间结点

```
    //寻找单边表的中间节点:快慢指针,慢指针走一步,快指针走两步
public static Node searchMid(Node head){
    Node p = head;
    Node q = head;
    while(p != null && p.next != null && p.next.next != null){
        p = p.next.next;
```

```
            q = q.next;
        }
        return q;
}
```

10).检测链表是否有环

```
    //检测链表是否有环:快慢指针,慢指针走一步,快指针走两步
public static boolean isLoop(Node head){
        Node fast = head;
        Node slow = head;
        if(fast==null){
            return false;
        }
        while(fast != null && fast.next != null){
            fast = fast.next.next;
            slow = slow.next;
            if(fast == slow){
                return true;
            }
        }
       return !(fast == null || fast.next == null);
}
```

11).在不知道头指针的情况下删除指定结点

```
    //在不知道头节点的情况下删除指定的节点,若 n 为尾节点则无法删除,因为不知道前驱
public static boolean deleteNode(Node n){
        if(n == null || n.next == null){
            return false;
        }
        //用后面一个节点覆盖当前节点
        int temp = n.data;
        n.data = n.next.data;
        n.next.data = temp;
        n.next = n.next.next;
        return true;
}
```

12).判断两个链表是否相交

```
    //判断两个链表是否相交:首先遍历两个链表得到他们的长度，就能知道哪个链表比较长，
//以及长的链表比短的链表多几个结点。在第二次遍历的时候，在较长的链表上先走若干步，
//接着再同时在两个链表上遍历，找到第一个相同的结点就是他们的第一个公共结点。
public Node findFirstCommonNode(Node root1,Node root2){
int length1 = getLength(root1);
```

```
int length2 = getLength(root2);
Node pointLongListNode = null;
Node pointShortListNode = null;
int dif = 0;
if(length1 >length2){
    pointLongListNode = root1;
    pointShortListNode = root2;
    dif = length1-length2;
}else{
    pointLongListNode = root2;
    pointShortListNode = root1;
    dif = length2 - length1;
}
for(int i = 0;i<dif;i++)
    pointLongListNode = pointLongListNode.next;
while(pointLongListNode != null && pointShortListNode != null && pointLongListNode !=
pointShortListNode){
    pointLongListNode = pointLongListNode.next;
    pointShortListNode = pointShortListNode.next;
}
return pointLongListNode; //如果返回为 null,则不相交
}
private int getLength(Node root){
    int result = 0;
    if(root == null)
        return result;
    Node point = root;
    while(point != null){
        point = point.next;
        result++;
    }
    return result;
}
```

13).链表的环的入口

```
    //链表环的入口: 判断是否有环,然后计算环中节点的个数 k,
//然后在用快慢指针,快指针先走 k 步,然后和慢指针同步走,最终相遇点就是环的入口
public Node entryNodeOfLoop(Node pHead){
    //判断是否有环, 并返回环中相遇节点
    Node meetingNode = findMeetingNode(pHead);
    if(meetingNode == null){
        return null;
    }
    //计算环的长度
```

```
        int len = 1;
        Node p1 = meetingNode;
        while(p1.next != meetingNode){
            p1.next = p1.next;
            len ++;
        }
        //快指针先移动 len
        Node fast = pHead;
        for(int i = 0; i < len; i ++){
            fast = fast.next;
        }
        //快慢指针同时移动
        Node slow = pHead;
        while(slow != fast){
            fast = fast.next;
            slow = slow.next;
        }
        return slow;
}
```

14).合并两个排序的链表

```
        //合并两个排序的链表:递归
public Node    Merge(Node pHead1,Node pHead2){
if(pHead1 == null)
        return pHead2;
else if(pHead2 == null)
        return pHead1;
Node pMergedHead = null;
if(pHead1.data <pHead2.data){
        pMergedHead = pHead1;
        pMergedHead.next = Merge(pHead1.next ,pHead2);
}else{
        pMergedHead = pHead2;
        pMergedHead.next = Merge(pHead1,pHead2.next);
}
return pMergedHead;
  }
```

15).复杂链表的复制

```
        //复制复杂链表
    public static Node copy(Node head){
        copyList(head);
        setSbiling(head);
        return disConnectList(head);
    }
```

```
    public static void copyList(Node head){
        Node node = head;
        while(node != null){
            Node copyNode = new Node();
            copyNode.value = node.value;
            copyNode.next = node.next;
            copyNode.sbiling = null;
            node.next = copyNode;
            node = copyNode.next;
        }
}
public static void setSbiling(Node head){
        Node node = head;
        while(node != null){
            Node copyNode = node.next;
            if(node.sbiling != null){
                copyNode.sbiling = node.sbiling.next;
            }
            node = copyNode.next;
        }
}
public static Node disConnectList(Node head){
        Node node = head;
        Node copyHead = null;
        Node copyNode = null;

        if(node != null){
            copyHead = node.next;
            copyNode = node.next;
            node.next = copyNode.next;
            node = node.next;
        }
        while(node != null){
            copyNode.next = node.next;
            copyNode = copyNode.next;
            node.next = copyNode.next;
            node = node.next;
        }
        return copyHead;
}
```

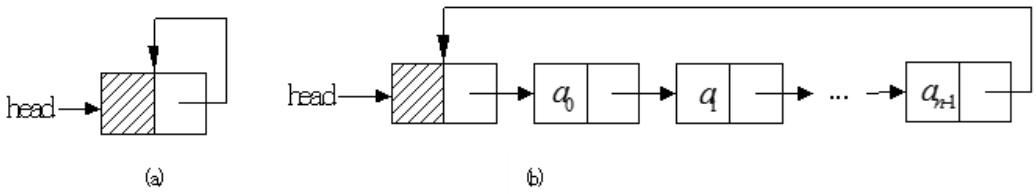
16).两个链表的第一个公共结点(与判断两个链表是否相交方法相同)

```
    //判断两个链表是否相交:首先遍历两个链表得到他们的长度，就能知道哪个链表比较长，
//以及长的链表比短的链表多几个结点。在第二次遍历的时候，在较长的链表上先走若干步，
```

```
//接着再同时在两个链表上遍历，找到第一个相同的结点就是他们的第一个公共结点。
public Node findFirstCommonNode(Node root1,Node root2){
int length1 = getLength(root1);
int length2 = getLength(root2);
Node pointLongListNode = null;
Node pointShortListNode = null;
int dif = 0;
if(length1 >length2){
    pointLongListNode = root1;
    pointShortListNode = root2;
    dif = length1-length2;
}else{
    pointLongListNode = root2;
    pointShortListNode = root1;
    dif = length2 - length1;
}
for(int i = 0;i<dif;i++)
    pointLongListNode = pointLongListNode.next;
while(pointLongListNode != null && pointShortListNode != null && pointLongListNode !=
pointShortListNode){
    pointLongListNode = pointLongListNode.next;
    pointShortListNode = pointShortListNode.next;
}
return pointLongListNode; //如果返回为 null,则不相交
}
private int getLength(Node root){
    int result = 0;
    if(root == null)
        return result;
    Node point = root;
    while(point != null){
        point = point.next;
        result++;
    }
    return result;
}
```

17).循环链表

表中的最后一个节点的指针域指向头结点，整个链表形成一个环. 和单链表相比，循环单链表的长处是从链尾到链头比较方便。当要处理的数据元素序列具有环型结构特点时，适合于采用循环单链表；和单链表相同，循环单链表也有**带头结点结构和不带头结点结构**两种，带头结点的循环单链表实现插入和删除操作时，算法实现较为方便.



（a）空链表；（b）非空链表

带头结点的循环单链表的操作实现方法和带头结点的单链表的操作实现方法类同，差别仅在于：

（i）在构造函数中，要加一条 head.next = head 语句，把初始时的带头结点的循环单链表设计成上图中（a）所示的状态。

（ii）在 index(i)成员函数中，把循环结束判断条件 current != null 改为 current != head。

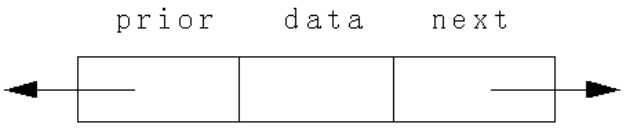
```
//单向循环链表类
public class CycleLinkList implements List {

    Node head; //头指针
    Node current;//当前结点对象
    int size;//结点个数

    //初始化一个空链表
    public CycleLinkList()
    {
        //初始化头结点，让头指针指向头结点。并且让当前结点对象等于头结点。
        this.head = current = new Node(null);
        this.size =0;//单向链表，初始长度为零。
        this.head.next = this.head;
    }

    //定位函数，实现当前操作对象的前一个结点，也就是让当前结点对象定位到要操作结点的前一个结点。
    //比如我们要在 a2 这个节点之前进行插入操作，那就先要把当前节点对象定位到 a1 这个节点，然后修改 a1 节点的指针域
    public void index(int index) throws Exception
    {
        if(index <-1 || index > size -1)
        {
           throw new Exception("参数错误！ ");
        }
        //说明在头结点之后操作。
        if(index==-1)      //因为第一个数据元素结点的下标是 0，那么头结点的下标自然就是-1 了。
```

```
            return;
        current = head.next;
        int j=0;//循环变量
        while(current != head&&j<index)
        {
            current = current.next;
            j++;
        }
    }
    @Override
    public void delete(int index) throws Exception {
        // TODO Auto-generated method stub
        //判断链表是否为空
        if(isEmpty())
        {
            throw new Exception("链表为空，无法删除！");
        }
        if(index <0 ||index >size)
        {
            throw new Exception("参数错误！");
        }
        index(index-1);//定位到要操作结点的前一个结点对象。
        current.setNext(current.next.next);
        size--;
    }
    @Override
    public Object get(int index) throws Exception {
        // TODO Auto-generated method stub
        if(index <-1 || index >size-1)
        {
            throw new Exception("参数非法！");
        }
        index(index);
        return current.getElement();
    }
    @Override
    public void insert(int index, Object obj) throws Exception {
        // TODO Auto-generated method stub
        if(index <0 ||index >size)
        {
            throw new Exception("参数错误！");
        }
        index(index-1);//定位到要操作结点的前一个结点对象。
        current.setNext(new Node(obj,current.next));
```

```
            size++;
        }
        @Override
        public boolean isEmpty() {
            // TODO Auto-generated method stub
            return size==0;
        }
        @Override
        public int size() {
            // TODO Auto-generated method stub
            return this.size;
        }
}
```

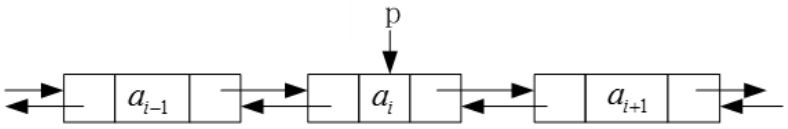
18).双链表

双向链表是每个结点除后继指针外还有一个前驱指针。和单链表类同，双向链表也有带头结点结构和不带头结点结构两种，带头结点的双向链表更为常用；另外，双向链表也可以有循环和非循环两种结构，循环结构的双向链表更为常用.



双向链表结点的图示结构

在双向链表中，有如下关系：设对象引用 p 表示双向链表中的第 i 个结点，则 p.next 表示第 i+1 个结点，p.next.prior 仍表示第 i 个结点，即 p.next.prior == p；同样地，p.prior 表示第 i-1 个结点，p.prior.next 仍表示第 i 个结点，即 p.prior.next == p。下图是双向链表上述关系的图示：



2.栈,堆和队列

1).O(1)时间求栈中最小元素

栈: 一种动态集合，它是一种 LIFO（last in first out 后进先出）结构

```
    //O(1)时间求栈中最小元素:加入一个辅助栈用来存储最小值集合
static Stack<Integer> ele = new Stack();
static Stack<Integer> min = new Stack();
public static void push(int data){
    ele.push(data);
    if (min.isEmpty()){
        min.push(data);
```

```
        }else {
            if (data < min.peek()){//peek 返回栈顶元素,不删除
                min.push(data);
            }
        }
    }
        public static int pop(){
        int topData = ele.peek();
        ele.pop();
        if (topData == min()){//如果在原栈中删除栈顶元素和最小栈的栈顶元素相等,为了同步,两个栈中都需要删除
            min.pop();
        }
        return topData;
    }
    public static int min(){
        if (min.isEmpty()){
            return Integer.MAX_VALUE;
        }else {
            return min.peek();
        }
    }
```

2).两个栈模拟队列

```
    //两个栈模拟队列:第一个栈临时保存插入的数据，当调用弹出函数的时候，
//如果 stack2 不为空则直接弹出；为空则把 stack1 中的数据全部弹出放到 stack2 中
private Stack<String> stack1 = new Stack<String>();
private Stack<String> stack2 = new Stack<String>();

public void appendTail(String s){
    stack1.push(s);
}
public String deletedHead() throws Exception{
    if(stack2.isEmpty()){
        while(!stack1.isEmpty()){
            stack2.push(stack1.pop());
        }
    }
    if(stack2.isEmpty()){
        throw new Exception("队列为空，不能删除");
    }
    return stack2.pop();
}
```

3).滑动窗口的最大值,给定一个滑动窗口的大小,找出所有滑动窗口的最大值,例如输入数组{2,3,4,2,6,2,5,1}及滑动窗口的大小,则一共存在 6 个滑动窗口,最大值分别为{4,4,6,6,6,5}

```
    //队列的最大值:同滑动窗口最大值(把窗口大小改为队列的大小即可)
public static ArrayList<Integer> maxInWindows(int [] num, int size){
        ArrayList<Integer> result = new ArrayList<Integer>();
        if (num == null || num.length < size || size == 0) {
            return result;
        }
        //双端队列
        Deque<Integer> deque = new ArrayDeque<Integer>();
        for (int i = 0; i < num.length; i++) {
            if (!deque.isEmpty() && (i - deque.peekFirst()) == size) {
                // 如果队列头部的元素已经超出滑动窗口的范围，就将头部元素退出
                deque.pollFirst();
            }
            while (!deque.isEmpty() && num[i] >= num[deque.peekLast()]) {
                // 如果新来的元素比队列最后一个元素大，则将最后一个元素退出
                deque.pollLast();
            }
            deque.offer(i);
            if (i >= (size - 1)) {
                // 如果遍历的元素已经达到一个滑动窗口的大小，就开始提取窗口的最大值了
                result.add(num[deque.peekFirst()]);
            }
        }
        return result;
    }
```

4).堆的性质

将数组的元素依次安排在二叉树中的根结点、根结点的左孩子、根结点的右孩子位置之上，再将剩余元素依次安排在根结点的左孩子的左孩子、根结点左孩子的右孩子、根结点右孩子的左孩子、根结点右孩子的右孩子位置之上.....按照如此顺序得到的二叉树，**若每个根结点元素都小（大）于其左右孩子，则称此二叉树为小（大）堆，称对应的数组具有堆属性（或此数组此时就是堆数据结构）**。

构造大顶堆: 通过数组构造,假设小标从 0 开始;

i)parent(t)=(t - 1) >> 1，即 t 的父节点的下标=(t-1)/2，注意此处是整除，例如：t = 6，parent(t) =2;

ii) left(t)=t << 1 + 1，即 t 的左孩子的节点下标=t * 2 + 1，例如：t = 2，left(t) =5;

iii) right(t)=t << 1 + 2，即 t 的右孩子的节点下标=t * 2 + 2，例如：t = 2, right(t) =6.

```
    //根据树的性质建堆，树节点前一半一定是分支节点，即有孩子的，所以我们从这里开始调整出初始堆
public static void adjust(List<Integer> heap){
```

```
        for (int i = heap.size() / 2; i > 0; i--)
            adjust(heap,i, heap.size()-1);//heap[heap.size()/2,...n]之后都为树的叶子节点，每一叶子节点本身就是包含一个元素的最大堆，故只需要构建 heap.size()/2 之前元素的最大堆

        System.out.println("================================================");
        System.out.println("调整后的初始堆：");
        print(heap);
    }
    public static void adjust(List<Integer> heap,int i, int n) {
        int child;
        for (; i <= n / 2; ) {
            child = i * 2;
            if(child+1 <= n && heap.get(child) < heap.get(child+1))
                child+=1;//使 child 指向值较大的孩子
            if(heap.get(i)< heap.get(child)){
                swap(heap,i, child);
                        //交换后，以 child 为根的子树不一定满足堆定义，所以从 child 处开始调整
                i = child;

            }   else break;
        }
    }
    public static void swap(List<Integer> heap, int a, int b) {
        //临时存储 child 位置的值
        int temp = (Integer) heap.get(a);
        //把 index 的值赋给 child 的位置
        heap.set(a, heap.get(b));
        //把原来的 child 位置的数值赋值给 index 位置
        heap.set(b, temp);
    }
```

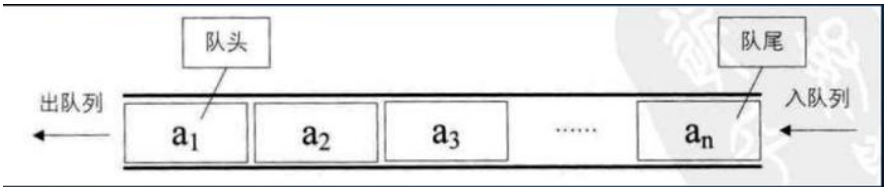
5).队列的最大值:定义一个队列得到队列里最大的值,时间复杂度为 O(n): 给定一个数组和滑动窗口的大小,找出所有滑动窗口里的最大值，例如：数组{2,3,4,2,6,2,5,1},窗口大小为 3，那么一共有 6 个滑动窗口，他们的最大值分别为{4,4,6,6,6,5}.

```
//队列的最大值:同滑动窗口最大值(把窗口大小改为队列的大小即可)
        public static ArrayList<Integer> maxInWindows(int [] num, int size){
                ArrayList<Integer> result = new ArrayList<Integer>();
                if (num == null || num.length < size || size == 0) {
                    return result;
                }
                //双端队列
                Deque<Integer> deque = new ArrayDeque<Integer>();
```

```
            for (int i = 0; i < num.length; i++) {
                if (!deque.isEmpty() && (i - deque.peekFirst()) == size) {
                    // 如果队列头部的元素已经超出滑动窗口的范围，就将头部元素退出
                    deque.pollFirst();
                }
                while (!deque.isEmpty() && num[i] >= num[deque.peekLast()]) {
                    // 如果新来的元素比队列最后一个元素大，则将最后一个元素退出
                    deque.pollLast();
                }
                deque.offer(i);
                if (i >= (size - 1)) {
                    // 如果遍历的元素已经达到一个滑动窗口的大小，就开始提取窗口的最大值了
                    result.add(num[deque.peekFirst()]);
                }
            }
            return result;
        }
```

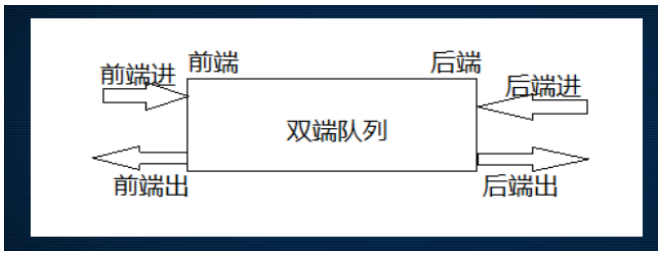
6).双端队列的结构以及性质

一般队列:



插入的一端称为队尾，删除的一端称为队头;

双端队列:



双端队列在经常使用到，如 LinkedList（链表存储结构）、ArrayDeque（顺序表存储结构）、LinkedBlockingDeque(阻塞式的双端队列)；其中 LinkedBlockingDeque 是线程安全的，因为它在每次插入和删除上都加了锁来控制。

7).栈的压入弹出序列: 输入两个整数序列，第一个序列表示栈的压入顺序，请判断第二个序列是否为该栈的弹出顺序。假设压入栈的所有数字均不相等。例如序列 1,2,3,4,5 是某栈的压入顺序，序列 4，5,3,2,1 是该压栈序列对应的一个弹出序列，但 4,3,5,1,2 就不可能是该压栈序列的弹出序列。（注意：这两个序列的长度是相等的）.

```
    //栈的压入弹出序列:用到辅助栈
public boolean IsPopOrder(int [] pushA,int [] popA) {
        if(pushA == null || popA == null){
            return false;
        }
        int m = pushA.length;
        int n = popA.length;

        if(m == 0 || n == 0){
            return false;
        }
        Stack<Integer> s = new Stack<Integer>();
        int start = 0;
        for(int i = 0; i < n; i++){
            while(s.empty() || s.peek() != popA[i]){

                if(start >= m){
                    return false;
                }
                s.push(pushA[start++]);
            }
            if(s.peek() != popA[i]){
                return false;
            }
             s.pop();
        }
        return true;
 }
```

3.排序算法

1).选择排序

```
    //选择排序
  public static void sort(int num[]){
        for(int i=0;i<num.length-1;i++){
            int min = i;
            for(int j=i+1;j<num.length;j++){
```

```
                if(num[min]>num[j]){
                    min = j;
                }
            }
            if(min!=i)
                Utils.exec(num,i,min);
        }
    }
```

2).插入排序

```
    //插入排序
public static void insertSort(int[] arr){
    int i, j;
    int n = arr.length;
    int target;

    //假定第一个元素被放到了正确的位置上
    //这样，仅需遍历 1 - n-1
    for (i = 1; i < n; i++)
    {
        j = i;
        target = arr[i];

        while (j > 0 && target < arr[j - 1])
        {
            arr[j] = arr[j - 1];
            j--;
        }

        arr[j] = target;
    }
}
```

3).冒泡排序

```
    //冒泡排序
public static void sort(int num[]){
    int Length= num.length;
    for(int i=0;i<Length-1;i++){
        for(int j=0;j<Length-i-1;j++){
            if(num[j]>num[j+1]){
                Utils.exec(num, j, j+1);
            }
        }
```

```
    }
}
```

4).希尔排序

```
    //希尔排序
public static void sort(int num[]){
    int N = num.length;
    int gap = 1;
    while(gap<N/2) gap = 2*gap+1;
    while(gap>=1){
        ////从第 gap 个元素，逐个对其所在组进行直接插入排序操作
        for(int i=gap;i<N;i++){
            ////插入排序采用交换法
            for(int j=i; j >= gap    && Utils.less(num[j], num[j-gap]);j-=gap){
                swap(num, j, j-gap);
            }
        }
        gap = gap/2;
    }
}
    public static void swap(int[] num, int left, int right) {
    int temp = num[left];
    num[left] = num[right];
    num[right] = temp;
}
```

5).快速排序

```
    //快速排序
public static void sort(int num[], int left, int right) {
    if (left < right) {
        int index = partition(num, left, right);//基准元素的下标，数组的划分调用
        sort(num, left, index-1);
        sort(num, index, right);
    }

}
    public static   int partition(int[] num, int left, int right) {
    if(num==null || num.length<=0 || left<0 || right>=num.length){
        return 0;
    }
    int prio = num[left + (right - left) / 2]; //基准元素，中间位置，可以是第一个或者最后一个
    while(left<=right){
        while (num[left] > prio)
            left++; //如果比基准元素小的位置不动
```

```
            while (num[right] < prio)
                right--; //如果比基准元素大的位置不动
            if (left<=right) { //如果有左边的元素大于右边的元素，则进行交换
                swap(num, left, right);
                left++;
                right--;
            }
        }
        return left;
}
```

6).堆排序

```
    //堆排序:手写
//把堆中的a,b 位置的值互换
public static void heapSort(){
    List<Integer> array = new ArrayList<Integer>(Arrays.asList(null,
        1, 2, 5, 10, 3, 7, 11, 15, 17, 20, 9, 15, 8, 16));
    adjust(array);//调整使 array 成为最大堆

    for (int i = heap.size()-1; i > 0; i--) {
    //把根节点跟最后一个元素交换位置，调整剩下的 n-1 个节点，即可排好序
        swap(heap,1, i);
            adjust(heap,1, i - 1);
        }
    }
 }
 //根据树的性质建堆，树节点前一半一定是分支节点，即有孩子的，所以我们从这里开始调整出初始堆
public static void adjust(List<Integer> heap){
    for (int i = heap.size() / 2; i > 0; i--)
        adjust(heap,i, heap.size()-1);//heap[heap.size()/2,...n]之后都为树的叶子节点，每一叶子节点本身就是包含一个元素的最大堆，故只需要构建 heap.size()/2 之前元素的最大堆

    System.out.println("================================================");
    System.out.println("调整后的初始堆：");
    print(heap);
}
public static void adjust(List<Integer> heap,int i, int n) {
    int child;
    for (; i <= n / 2; ) {
        child = i * 2;
        if(child+1 <= n && heap.get(child) < heap.get(child+1))
            child+=1;//使 child 指向值较大的孩子
        if(heap.get(i)< heap.get(child)){
            swap(heap,i, child);
```

```
                    //交换后，以 child 为根的子树不一定满足堆定义，所以从 child 处开始调整
                i = child;

            }   else break;
        }
    }
    public static void swap(List<Integer> heap, int a, int b) {
        //临时存储 child 位置的值
        int temp = (Integer) heap.get(a);

        //把 index 的值赋给 child 的位置
        heap.set(a, heap.get(b));

        //把原来的 child 位置的数值赋值给 index 位置
        heap.set(b, temp);
    }
```

7).归并排序

```
        //归并排序
    //初始 low=0; high=num.length-1
    public   static void sort(int num[],int low,int high){
        if(low<high){
            int mid = low+(high-low)/2;
            sort(num,low,mid);
            sort(num,mid+1,high);
            merge(num,low,mid,high);
        }

    }
    public static void merge(int[] num, int low, int mid, int high) {
        int copyNum[]=new int[num.length];
        for(int i=0;i<=high;i++){
            copyNum[i] = num[i];
        }
        int left = low;
        int right = mid+1;
        int current = low;
        while(left<=mid && right<=high){
            if(copyNum[left]<copyNum[right]){
                num[current++]=copyNum[left];
                left++;
            }else{
                num[current++]=copyNum[right];
                right++;
```

```
            }
        }
        int remaining = mid-left;
        for(int i=0;i<=remaining;i++){
            num[current+i]=copyNum[left+i];
        }
    }
```

8).计数排序

```
    //计数排序
public static int[] sort(int[] A) {
    int k = findMax(A);
    int[] C = new int[k+1]; //长度加1，同步伪代码
    int[] B = new int[A.length];

    //初始化
    for(int i=0;i<C.length;i++) {
        C[i] = 0;
    }
    //将带排序数组元素重复个数映射到辅助数组中
    for(int i=0;i < A.length;i++) {
        C[A[i]] = C[A[i]] + 1;
    }
    //处理辅助数组，使c[j]表示待排序数组中小于等于j 的元素个数
    for(int i=1;i<C.length;i++) {
        C[i] = C[i] + C[i-1];
    }
    //从后往前还原排序数组元素到一个新的数组
    int i = A.length-1;
    for(;i >= 0; i --) {
        B[C[A[i]]-1] = A[i];
        C[A[i]] = C[A[i]] - 1;
    }
    return B;
}
public static int findMax(int[] A) {
    int temp = 0;
    for(int i:A) {
        if(i > temp) {
            temp = i;
        }
    }
    return temp;
}
```

9).基数排序

```
    //基数排序
public static void sort(int []array){
    //1.找出数组 array 中最大值
    int max = findMax(array);
    int time = 0; //time 记录最大数的位数
    while(max>0){
        max/=10;
        time++;
    }
    //2.每位数都是从 0 到 9
    LinkedList<Integer> queue[]=new LinkedList[10];
    for(int i=0;i<10;i++){
        queue[i]=new LinkedList<>(); //为每位数申请一个 list 空间
    }
    //3.对每位数对应的数组排序
    for(int i=0;i<time;i++){ //0 代表个位,1 代表十位,...
        //3.1
        for(int j=0;j<array.length;j++){
            //queue[array[j]%(int)Math.pow(10, i+1)/(int)Math.pow(10, i)].add(array[j]);
            //(int)Math.pow(10, i): 10 的 i 次方除以 10 取余数
            //array[j]/(int)Math.pow(10, i)%10: array[j]除以 10 的 i 次方然后除以 10 取余数, 得到每个数
的个位, 十位,..对应的数字
            queue[array[j]/(int)Math.pow(10, i)%10].add(array[j]);
            //queue[0]存储个位为 0 的所有数, queue[1]存储个位为 1 的所有数
        }
        //3.2 按照位数重新组织数列(先按个位, 然后十位,...)
        int count = 0;
        for(int k=0;k<10;k++){
            while(!queue[k].isEmpty()){
                array[count++]=queue[k].remove(); //返回 queue[k]的第一个元素给 array[0]
            }
        }
    }


}
public static int findMax(int[] array) {
    int max = 0;
    for(int j=0;j<array.length;j++){
        if(array[j]>array[max]){
            max = j;
        }
    }
```

```
    return array[max];
}
```

10.桶排序

```
    //桶排序
    static void bucketSort(double arr[]){
        int n = arr.length;
        ArrayList arrList[] = new ArrayList [n];
        //把 arr 中的数均匀的的分布到[0,1)上，每个桶是一个 list，存放落在此桶上的元素
        for(int i =0;i<n;i++){
            int temp = (int) Math.floor(n*arr[i]);
            if(null==arrList[temp])
                arrList[temp] = new ArrayList();
            arrList[temp].add(arr[i]);
        }
        //对每个桶中的数进行插入排序
        for(int i = 0;i<n;i++){
            if(null!=arrList[i])
                insert(arrList[i]);
        }
        //把各个桶的排序结果合并
        int count = 0;
        for(int i = 0;i<n;i++){
            if(null!=arrList[i]){
                Iterator iter = arrList[i].iterator();
                while(iter.hasNext()){
                    Double d = (Double)iter.next();
                    arr[count] = d;
                    count++;
                }
            }
        }
    }

//用插入排序对每个桶进行排序
static void insert(ArrayList list){
    if(list.size()>1){
        for(int i =1;i<list.size();i++){
            if((Double)list.get(i)<(Double)list.get(i-1)){
                double temp = (Double) list.get(i);
                int j = i-1;
                for(;j>=0&&((Double)list.get(j)>(Double)list.get(j+1));j--)
                    list.set(j+1, list.get(j));
                list.set(j+1, temp);
```

```
            }
        }
    }
}
```

11).排序算法总结与对比

排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	排序方式	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	In-place	不稳定
插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
希尔排序	$O(n \log n)$	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(1)$	In-place	不稳定
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Out-place	稳定
快速排序	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	In-place	不稳定
堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	In-place	不稳定
计数排序	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$	Out-place	稳定
桶排序	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n + k)$	Out-place	稳定
基数排序	$O(n \times k)$	$O(n \times k)$	$O(n \times k)$	$O(n + k)$	Out-place	稳定

4.树

1).二叉树的基本性质

i). 包含 n(n>0)个元素的二叉树边数是 n-1

ii). 二叉树的高度或者深度是指该二叉树的层数; 若二叉树的高度为 h, h≥0,则该二叉树最少有 h 个元素，最多有 2^h−1 个元素

iii). 具有 n 个结点的完全二叉树的深度为（logn）+1

iv). 如果对一棵有 n 个结点的完全二叉树的结点按层序号遍历，对任意结点 i 有

a). 如果 i=1，则结点 i 是二叉树的根，无双亲；如果 i>1，则其双亲是结点 i/2

b). 如果 2i>n，则结点 i 无左孩子；否则其左孩子是结点 2i

c). 如果 2i+1>n，则结点 i 无右孩子；否则其右孩子是结点 2i+1

2).二叉排序(查找)树的性质

二叉排序树：或者是一棵空树，或者是具有下列性质的二叉树:

i). 若它的左子树不空，则左子树上所有结点的值均小于它的根结点的值

ii). 若它的右子树不空，则右子树上所有结点的值均大于它的根结点的值

iii). 它的左、右子树也分别为二叉排序树

3).二叉树的前序，中序，后序遍历，层次遍历

```
//二叉树节点类
public class BinaryNode<T> {
    public T data;// 数据域，存储数据元素
    public BinaryNode<T> left, right;// 链域，分别指向左右孩子结点
```

```
        // 构造结点，参数分别指向元素和左右孩子结点
        public BinaryNode(T data, BinaryNode<T> left, BinaryNode<T> right) {
            this.data = data;
            this.left = left;
            this.right = right;
        }
        // 构造指定值的叶子结点
        public BinaryNode(T data) {
            this(data, null, null);
        }
        public BinaryNode() {
            this(null, null, null);
        }
}
// 先序遍历以 p 结点为跟的子二叉树，递归方法
    public void preOrder(BinaryNode<T> p) {
        if (p != null) {// 若二叉树不为空
            System.out.print(p.data.toString() + " ");// 访问当前结点
            preOrder(p.left);// 按先跟次序遍历当前结点的左子树
            preOrder(p.right);// 按先跟次序遍历当前结点的右子树
        }
    }
// 中序遍历以 p 结点为跟的子二叉树，递归调用
    public void inOrder(BinaryNode<T> p) {
        if (p != null) {
            inOrder(p.left);// 中跟次序遍历左子树，递归调用
            System.out.print(p.data.toString() + " ");
            inOrder(p.right);// 中跟次序遍历右子树，递归调用
        }
    }
//  后跟次序遍历以 p 结点为跟的子二叉树，递归调用
        public void postOrder(BinaryNode<T> p) {
            if (p != null) {
                postOrder(p.left);
                postOrder(p.right);
                System.out.print(p.data.toString() + " ");
            }
        }
    // 按层次遍历二叉树
    public void levelOrder() {
        LinkedQueue<BinaryNode<T>> que = new LinkedQueue<BinaryNode<T>>();
        BinaryNode<T> p = this.root;
        System.out.println("层次遍历：");
        while (p != null) {
```

```
            System.out.print(p.data + "");
            if (p.left != null)
                que.enquenu(p.left); // p 的左孩子节点入队
            if (p.right != null)
                que.enquenu(p.right); // p 的右孩子节点入队
            p = que.dequeue(); // p 指向出对节点,若队列空返回 null
        }
        System.out.println();
    }
```

4).二叉排序树的插入

```
    /**在某个位置开始判断插入元素*/
public BinaryNode<T> insert(T t,BinaryNode<T> node){
        if(node==null)
        {
            //新构造一个二叉查找树
            return new BinaryNode<T>(t, null, null);
        }
        int result = t.compareTo(node.data);
        if(result<0)
            node.left= insert(t,node.left);
        else if(result>0)
            node.right= insert(t,node.right);
        else
            ;//doNothing
        return node;
}
```

5).已知先序遍历和中序遍历如何求后序遍历(重建二叉树): 输入某二叉树的前序遍历和中序遍历的结果，请重建出该二叉树。假设输入的前序遍历和中序遍历的结果中都不含重复的数字。例如输入前序遍历序列{1,2,4,7,3,5,6,8}和中序遍历序列{4,7,2,1,5,3,8,6}，则重建二叉树并返回.

方法:通过前序遍历知道根节点; 通过中序遍历以及根节点的值, 缺点左右子树; 然后中序遍历中根节点之前的均为左子树的结点, 右边均为右子树的结点; 最后递归重建

```
static class TreeNode{//二叉树结点类
        int val;
        TreeNode left;
        TreeNode right;
        TreeNode(int x) {
            val = x;
        }
    }
public static TreeNode reConstructBinaryTree(int [] pre,int [] in) {
```

```
        if(pre == null || in == null){
            return null;
        }
        //调用递归重建二叉树, pre 为先序遍历数组,in 为中序遍历数组.
        TreeNode mm = reConstructBinaryTreeCore(pre, in, 0, pre.length-1, 0, in.length-1);
        return mm;
    }
    //递归调用构建
    public static TreeNode reConstructBinaryTreeCore(int[] pre, int[] in, int preStart, int preEnd, int
inStart, int inEnd) {
        TreeNode tree = new TreeNode(pre[preStart]);
        tree.left = null;
        tree.right = null;
        if (preStart == preEnd && inStart == inEnd) {
            return tree;
        }
        int root = 0;
        for(root= inStart; root < inEnd; root++){
            if (pre[preStart] == in[root]) {//找到根节点
                break;
            }
        }
        int leifLength = root - inStart;//左子树的长度
        int rightLength = inEnd - root;//右子树的长度
        if (leifLength > 0) {
            //左子树
            tree.left = reConstructBinaryTreeCore(pre, in, preStart+1, preStart+leifLength, inStart, root
-1);
        }
        if (rightLength > 0) {
            //右子树
            tree.right = reConstructBinaryTreeCore(pre, in, preStart+1+leifLength, preEnd, root+1, inEn
d);
        }
        return tree;
    }
```

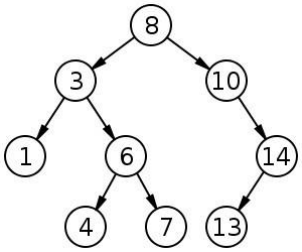
6).二叉树的下一个结点：给定一棵二叉树的和其中一个结点，如何找出中序遍历序列的下一个结点,树中的结点包括分别指向左右孩子的指针结点,指向父节点的指针.

分析: i).如果该结点有右子树,那么它的下一个结点就是它的右子树中的最左结点;

ii).如果没有右子树,并且该结点是其父节点的左子节点,那么下一个结点就是其父节点;

iii).如果没有右子树,并且是其父节点的右子节点，那么沿着该结点的父节点往上找,直

到找到一个是它父节点的左子节点的的结点，如果这个结点存在,那么找到的这个结点的父节点就是所求结点.如下图：要找结点 7 的下一个结点，其中序遍历的下一个结点是 8，沿着 7 的父节点往上找，发现 3 是其父节点的左子节点,所以 7 的下一个结点是 8




```
static class TreeLinkNode {
    int val;
    TreeLinkNode left = null;
    TreeLinkNode right = null;
    TreeLinkNode next = null;
    TreeLinkNode(int val) {
      this.val = val;
    }
  }
  public TreeLinkNode getNext(TreeLinkNode pNode) {
    if (pNode == null)
      return null;
    TreeLinkNode tmp = null;
    // 如果 pNode 的右子树为空
    if (pNode.right == null) {
      tmp = pNode;
      // 如果该节点是某个最右节点
      while (tmp.next != null && tmp == tmp.next.right)//如果父节点不为 null,且该结点为父节点的右子树,如果是其父节点的左子树,则 while 不执行,直接返回其父节点
        tmp = tmp.next;
      return tmp.next == null ? null : tmp.next; //返回其父节点
    }
    // 如果右子树不为空，找到右子树中的最左节点
    tmp = pNode.right;
    while (tmp.left != null) {
      tmp = tmp.left;
    }
    return tmp;
  }
```

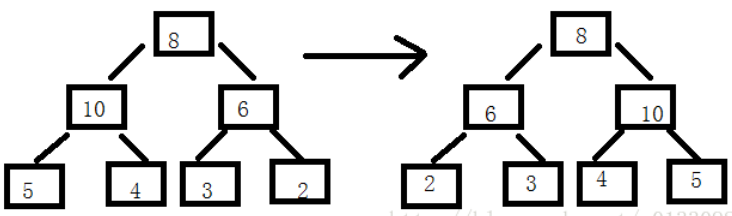
7).树的子结构:输入两棵二叉树 A,B，判断 B 是不是 A 的子结构.

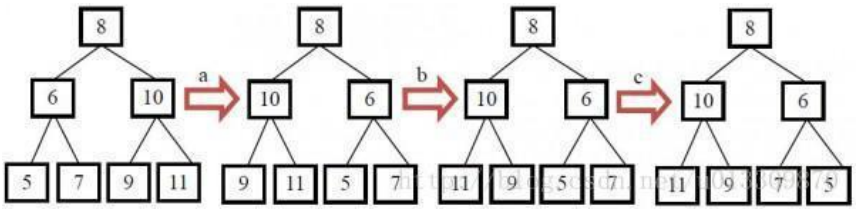
分析:i)在树 A 中找到和数 B 的根节点相同的节点,可以利用前序遍历 A 树中的节点，找到和 B 树的根节点相同的节点。

ii)在树 A 中找到和树 B 根节点相同的节点设为 R,再对比节点 R 在树 B 中的左孩子和右孩子是否和树 B 中的节点相同。
iii)如果 R 节点的值和树 B 中的节点的值不同，则不是子结构，如果相同，递归去判断他们的各自的左孩子和右孩子是否相同。递归的出口是到达树 A 或者树 B 的叶节点.

```
static class TreeNode {
    int val = 0;
    TreeNode left = null;
    TreeNode right = null;
    public TreeNode(int val) {
      this.val = val;
    }
  }
  public boolean hasSubtree(TreeNode root1,TreeNode root2){
    boolean result = false ;
    if(root1 != null && root2 != null)
    {
      if(root1.val == root2.val)
        result = isSubTree(root1,root2);
      if(result == false)
        result = hasSubtree(root1.left,root2);
      if(result == false)
        result = hasSubtree(root1.right,root2);
    }
    return result;
  }
  private boolean isSubTree(TreeNode root1,TreeNode root2){
    if(root2 == null)
      return true;
    if(root1 == null)
      return false;
    if(root1.val != root2.val)
      return false;
    return isSubTree(root1.left,root2.left)&&isSubTree(root1.right,root2.right);
  }
```

8).二叉树的镜像：输入一棵二叉树,输出它的镜像

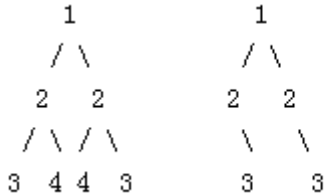




分析: 如果当前节点为空，返回，否则交换该节点的左右节点，递归的对其左右节点进行交换处理.

```
static class BinaryTreeNode {
      public int value;
      public BinaryTreeNode leftNode;
      public BinaryTreeNode rightNode;
      public BinaryTreeNode(){
      }
      public BinaryTreeNode(int value){
         this.value = value ;
         this.leftNode = null;
         this.rightNode = null;
      }
   }
   public void MirrorRecursively(BinaryTreeNode node){
      if(node == null)
         return;
      //叶子节点
      if(node.leftNode == null && node.rightNode == null)
         return;
      //交换左右节点
      BinaryTreeNode temp = node.leftNode;
      node.leftNode = node.rightNode;
      node.rightNode = temp;
      //递归左右子树
      if(node.leftNode != null)
         MirrorRecursively(node.leftNode);
      if(node.rightNode != null)
         MirrorRecursively(node.rightNode);
   }
```

9).对称二叉树:判断二叉树是否是对称的,如果它和它的镜像相同则为对称的



第一个是对称二叉树，第二个不是对称二叉树

定义二叉树的另一种遍历方式(前序对称遍历): 根节点->右子节点->左子节点;
方法: 如果该二叉树的前序遍历与前序对称遍历的结果相同, 那么该二叉树就是对称的

```
static class TreeNode {
    int val = 0;
    TreeNode left = null;
    TreeNode right = null;
    public TreeNode(int val) {
      this.val = val;
    }
  }
  public class Solution {
    boolean isSymmetrical(TreeNode r1,TreeNode r2) {
      if (r1 == null&&r2 == null)
        return true;
      if (r1 == null||r2 == null)
        return false;
      return r1.val==r2.val&&isSymmetrical(r1.left,r2.right)&&isSymmetrical(r1.right,r2.left);
    }
    boolean isSymmetrical(TreeNode pRoot)
    {
      return isSymmetrical(pRoot,pRoot);
    }
  }
```

10).判断数组是否为二叉搜索树的后序遍历结果

分析: 二叉搜索树中左<根<右, 所以数组中最后一个元素为二叉搜索树的根节点, 找到数组中所有比最后一个数字小的元素在左子树中, 比最后一个元素大的所有元素在右子树中; 递归实现.
例如: 数组{5,7,6,9,11,10,8}, 根节点为 8, {5,7,6}比 8 小, 所以在左子树中, {9,11,10}比 8 大, 所以在右子树中, 递归, 一次判断;
数组{7,4,6,5}, 5 是根节点,但是第一个元素 7 比 5 大,所以没有左子树, 但是 4 比 5 小, 但是右子树中所有节点比根节点大, 所以不满足二叉搜索树的性质.

```
public boolean verifySequence(int[] array,int start,int end) throws Exception{
    if(array == null || array.length < 2)
      return true;
    if(start < 0){
      throw  new Exception("first can't be less than 0");
    }
    if(end > array.length){
      throw new Exception("last can't be greater than the count of the element.");
    }
    int root = array[end];
    //在二叉搜索树中左子树的结点小于根节点
    int i = start;
```

```
    for(; i < end;i++){
        if(array[i]>root)
            break;
    }//找到分界点，比 arr[i]小的为左子树,比 arr[i]大的为右子树
    //在二叉搜索树中右子树的结点大于根节点
    int j = i;
    for(;j < end;j++){
        if(array[j] < root)
            return false;
    }
    //判断左子树是不是二叉搜索树
    boolean left = true;
    if(i >start)
        left =verifySequence(array ,start,i-1);
    //判断右子树是不是二叉搜索树
    boolean right = true;
    if(i < end)
        right = verifySequence(array,i,end-1);
    return (left && right);
  }
```

11).二叉树中和为某一值的路径:输入一棵二叉树和一个正数,打印出二叉树中结点值的和为输入正数的所有路径,从树的根节点开始往下一只到叶节点所经过的结点形成一条路径.

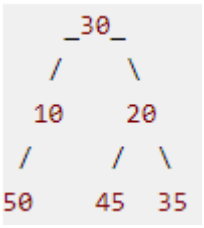
分析：从根节点开始，因此采用前序遍历，当访问到某一节点是,把该结点添加到路径上,并累加该结点的值,并且路径中结点值的和刚好等于输入的整数，则当前路径符合要求,打印出来，如果当前节点不是叶节点，子继续访问它的子节点，当前节点访问结束后自动回到它的父节点，因此,在函数退出之前要在路径上删除当前节点并减去当前节点的值.

```
static class TreeNode {
    int val = 0;
    TreeNode left = null;
    TreeNode right = null;
    public TreeNode(int val) {
        this.val = val;
    }
}
public void findPath(TreeNode root,int k){
    if(root == null)
        return;
    Stack<Integer> stack = new Stack<Integer>();
    //最终路径存储在 list 中返回
    ArrayList<ArrayList<Integer>> arrayList = new ArrayList<>();
    findPath(root,k,stack, arrayList);
```

```
    }
    public void findPath(TreeNode root,int k,Stack<Integer> path, ArrayList<ArrayList<Integer>> arr
ayList){
        if(root == null)
            return;
        if(root.left == null && root.right == null){
            if(root.val == k){
                System.out.println("路径开始");
                ArrayList<Integer> arrayList1 = new ArrayList<>();
                for(int i :path) {
                    System.out.print(i + ",");
                    arrayList1.add(i);
                }
                System.out.print(root.val);
                arrayList1.add(root.val);
                arrayList.add(arrayList1);
                System.out.println();
            }
        }
        else{
            path.push(root.val);
            findPath(root.left,k-root.val,path, arrayList);
            findPath(root.right,k-root.val,path, arrayList);
            path.pop();
        }
    }
```

12).二叉树序列化:序列化和反序列化二叉树

分析:利用二叉树的前序遍历顺序来序列化二叉树,当遇到 nullptr 时,用特殊字符”$”代替.



前序遍历为: 30,10,50,$,$,$,20,45,$,$,35,$,$

```
static class TreeNode {
    int val = 0;
    TreeNode left = null;
    TreeNode right = null;
    public TreeNode(int val) {
        this.val = val;
    }
```

```
String Serialize(TreeNode root) {
    StringBuilder s = new StringBuilder();
    if(root == null){
       s.append("$,");
       return s.toString();
    }
    s.append(root.val+",");
    s.append(Serialize(root.left));
    s.append(Serialize(root.right));
    return s.toString();
 }
 TreeNode Deserialize(String str) {
     index++;
    String[] DLRseq = str.split(",");
    TreeNode leave = null;
    if(!DLRseq[index].equals("$")){
       leave = new TreeNode(Integer.valueOf(DLRseq[index]));
       leave.left = Deserialize(str);
       leave.right = Deserialize(str);
    }
    return leave;
 }
```

13 二叉搜索树中第 k 大的结点

对二叉搜索树进行中序遍历，则遍历序列是递增排序的，因此中序遍历一颗二叉搜索树，可以很容易的得到它的第 k 大的节点。使用一个计数器变量，每遍历一个节点，计数器加 1，当计数器的值等于 k 时，root 节点即为所求节点.

```
static class TreeNode {
    int val = 0;
    TreeNode left = null;
    TreeNode right = null;
    public TreeNode(int val) {
        this.val = val;
}
int count = 0;   // 遍历计数
    TreeNode KthNode(TreeNode pRoot, int k) {
        if(pRoot == null || k <= 0)
            return null;
        TreeNode target = null;
        if(pRoot.left != null)          // 遍历左子树
            target = KthNode(pRoot.left, k);
        count++;       // 计数加 1
        if(target == null) {
            if(count == k) {       // 如果计数等于 k，则找到 pRoot
```

```
                target = pRoot;
                return target;
            }
        }
        if(target == null && pRoot.right != null)     // 遍历右子树
            target = KthNode(pRoot.right, k);
        return target;
    }
```

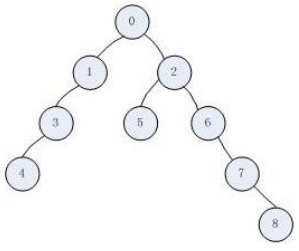
14).二叉树的深度:输入一个二叉树的根节点,求该树的深度,从根节点到叶节点依次经过的结点形成树的一条路径,最长路径的长度为树的深度.

分析:如果二叉树只有一个节点,那么深度为 1, 如果只有左子树,那么深度为左子树的深度加 1, 如果只有右子树,那么深度为右子树加 1, 如果既有左子树,又有右子树,那么深度取二者较大者.

```
static class TreeNode {
    int val = 0;
    TreeNode left = null;
    TreeNode right = null;
    public TreeNode(int val) {
        this.val = val;
}
public int TreeDepth(TreeNode root) {
    if(root == null){
      return 0;
    }
    int nLeft = TreeDepth(root.left);
    int nRight = TreeDepth(root.right);
    return (nLeft > nRight) ? (nLeft+1):(nRight+1);
  }
```

15).二叉树中结点的最大距离(两个结点之间边的个数最大)

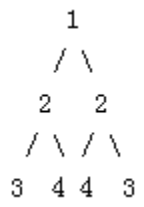
分析: 求出左子树距离根节点的最大距离记为 leftMaxDistance, 然后求出右子树距离根节点的最大距离记为 rightMaxDistance, 最后求出 maxDistance = leftMaxDistance + rightMaxDistance;




```
static class TreeNode{
    public int data;
```

```
    public TreeNode left;
    public TreeNode right;
    public int leftMax; //左子树距离根的最大距离
    public int rightMax;//右字数距离根的最大距离
    public TreeNode(int data){
      this.data = data;
      this.left = null;
      this.right = null;
    }
  }
  static int maxLen = 0;
  public void findMaxDistance(TreeNode root){
    if(root==null){
      return;
    }
    if(root.left==null){
      root.leftMax=0;
    }
    if(root.right==null){
      root.rightMax=0;
    }
    if(root.left!=null){
      findMaxDistance(root.left);
    }
    if(root.right!=null){
      findMaxDistance(root.right);
    }
    if(root.left!=null){
      int nTempMax = 0;
      nTempMax = root.left.leftMax > root.left.rightMax ? root.left.leftMax : root.left.rightMax;
      root.leftMax = nTempMax+1;
    }
    if(root.right!=null){
      int nTempMax=0;
      nTempMax=root.right.leftMax > root.right.rightMax ? root.right.leftMax : root.right.rightM
ax;
      root.rightMax=nTempMax+1;
    }
    if(root.leftMax + root.rightMax > maxLen){
      maxLen = root.leftMax + root.rightMax;
    }
  }
```

16).从上到下打印二叉树: 不分行从上到下打印二叉树,从上到下打印二叉树的每一个节点,同一层的结点按照从左到右的顺序打印.例如下图的打印结果为:1,2,2,3,4,4,3



```
static class TreeNode {
        int val = 0;
        TreeNode left = null;
        TreeNode right = null;
        public TreeNode(int val) {
              this.val = val;
}
public ArrayList<Integer> PrintFromTopToBottom(TreeNode root) {
       ArrayList list = new ArrayList<Integer>();
       if(root == null)
          return list;
       Queue<TreeNode> queue = new LinkedList<TreeNode>();
       queue.add(root);
       while(!queue.isEmpty()){
          TreeNode node = queue.poll();
          System.out.print(node.val+",");
          list.add(node.val);
          if(node.left != null)
             queue.add(node.left);
          if(node.right != null)
             queue.add(node.right);
       }
       return list;
   }
```

17).分行从上到下打印二叉树,同一层的结点按照从左到右的顺序打印,每一层打印到一行;例如上题中的打印结果为:

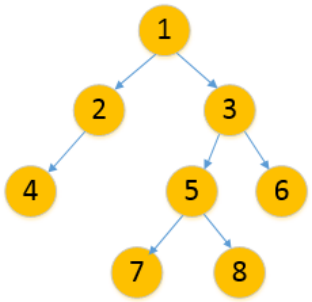
```
1
2 2
3 4 4 3
```

在上述问题的基础上增加一个变量用来记录没一层上节点的数量;

```
static class TreeNode<T> {
       public T val;
```

```
    public TreeNode<T> left;
    public TreeNode<T> right;
    public TreeNode(T val) {
        this.val = val;
        this.left = null;
        this.right = null;
    }
}
public static void printTreeInLine(TreeNode<Integer> root){
    if(root==null)
        return;
    Queue<TreeNode<Integer>> queue = new LinkedList<>();
    queue.offer(root);
    TreeNode<Integer> temp;
    while (!queue.isEmpty()){
        for(int size=queue.size();size>0;size--){
            temp = queue.poll();
            System.out.print(temp.val);
            System.out.print("\t");
            if(temp.left!=null)
                queue.offer(temp.left);
            if(temp.right!=null)
                queue.offer(temp.right);
        }
        System.out.println();
    }
}
```

18).之字形打印二叉树：请实现一个函数按照之字形打印二叉树，即第一行按照从左到右的顺序打印，第二层按照从右至左的顺序打印，第三行按照从左到右的顺序打印，其他行以此类推。



逐层打印的结果是：

1

3 2

4 5 6

8 7

分析:需要两个栈, 打印某一层节点时,把下一层的子节点保存到响应的栈里,如果当前打印的是奇数层(第一,第三层等),则先保存左子节点再保存右子节点到一个栈里; 如果当前打印的是偶数层,则先保存右子节点再保存左子节点到第二个栈里.

```
public static class TreeNode {
    public int val;
    public TreeNode left;
    public TreeNode right;
    public TreeNode(int val) {
      this.val = val;
    }
  }
  public static ArrayList<ArrayList<Integer>> printZigZag(TreeNode pRoot) {
    if(pRoot == null) {
      return null;
    }
    Stack<TreeNode> s1 = new Stack<>();
    Stack<TreeNode> s2 = new Stack<>();
    ArrayList<ArrayList<Integer>> res = new ArrayList<ArrayList<Integer>>();
    s1.push(pRoot);
    while(!s1.isEmpty() || !s2.isEmpty()) {
      if(!s1.isEmpty()) {
        ArrayList<Integer> arr1 = new ArrayList<>();
        while(!s1.isEmpty()) {
          TreeNode t1 = s1.pop();
          arr1.add(t1.val);
          //注意，一定要保证你插入到栈中的值不为空
          if(t1.left!=null) {
            s2.push(t1.left);
          }
          if(t1.right!=null) {
            s2.push(t1.right);
          }
        }
        res.add(arr1);
      } else if(!s2.isEmpty()){
        ArrayList<Integer> arr2 = new ArrayList<>();
        while(!s2.isEmpty()) {
          TreeNode t2 = s2.pop();
          arr2.add(t2.val);
          if(t2.right!=null) {
            s1.push(t2.right);
          }
```

```
            if(t2.left!=null) {
                s1.push(t2.left);
            }
        }
        res.add(arr2);
      }
    }
    return res;
  }
```

19).二叉搜索树与双向链表：输入一棵二叉搜索树，将该二叉搜索树转换成一个排序的双向链表。要求不能创建任何新的结点，只能调整树中结点指针的指向。

分析:由于二叉树的中序遍历具有可以实现递增的功能，同时二叉树的每一个节点都有两个指针，左指针和右指针，因此我们可以用这些特点将二叉树转换成一个双向链表，每次在递归遍历的时候设置一个 pre，记录中序遍历时当前节点的前节点，然后将当前节点的左指针指向 pre 节点，然后如果 pre 节点不为空则将 pre 的右节点指向当前节点，由此就形成了一个双向链表的前后指针。每次递归重复这两步，则可以形成一个完整的双向链表; 最后一步就是双向链表已经构建完成了，而题目要求返回双向链表，则从原二叉树的 root 节点往前遍历找到双向链表的头返回即可.

```
public TreeNode Convert(TreeNode pRootOfTree) {
    if (pRootOfTree == null) {
      return pRootOfTree;
    }
    TreeNode[] pre = new TreeNode[1];
    helper(pRootOfTree, pre);
    //已经完成双向链表的链接，找链表头节点返回
    TreeNode p = pRootOfTree;
    while (p.left != null) {
      p = p.left;
    }
    return p;
  }
  private void helper(TreeNode root, TreeNode[] pre) {
    if (root.left != null) {
      helper(root.left, pre);
    }
    root.left = pre[0];  //设置 left 指向前一个节点
    if (pre[0] != null) {
      pre[0].right = root;  //pre right 指向 root
    }
    //记录当前节点为尾节点
    pre[0] = root;
    if (root.right != null) {
      helper(root.right, pre);
```

```
    }
  }
```

20).二叉搜索树的第 k 小的结点:给定一棵二叉搜索树,找出其中第 k 小的结点.

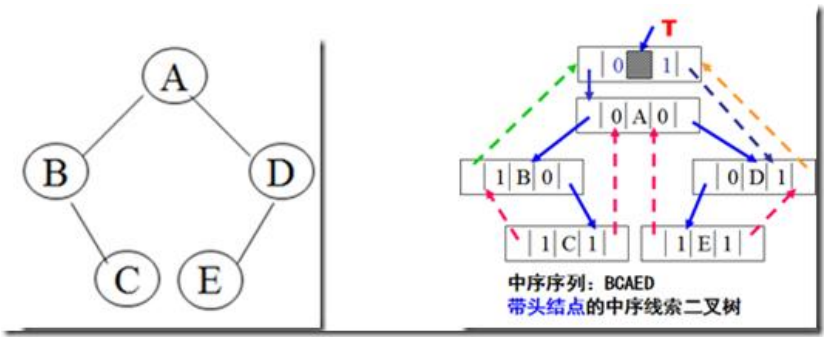
同问题 13

21).线索二叉树的的定义

二叉树是一种非线性结构，对二叉树进行遍历时，实际上那个是将二叉树这种非线性结构按某种需要转化成线性序列，但每次遍历二叉树时，都要用递归对其进行遍历，当二叉树的节点较多时，这样的效率是很低的. 在一棵只有 n 个结点的二叉树中，假设有 n 个结点，那么就有 2n 个指针域，因为二叉树只用到了其中的 n-1 个结点，所以只要利用剩下的 n+1 个节点，我们就能把中需遍历时所得到的中序二叉树保存下来，以便下次访问，中序二叉树的指针域有两种类型：一是用于链接二叉树本身；二是用于链接中序遍历序列。这类型的指针，左指指向中序遍历时节点顺序的前驱，右指针指向中序遍历时的后继。为了区别这两种指针域类型，我们在树的结点中要加上两个标志 lchild 和 rchild，分别标志左右指针域的类型。 其数据结构如下：

其中 LTag = 0 lchild 域指示节点的左孩子；LTag=1 lchild 域指示节点的前驱。

RTag = 0 rchild 域指示节点的左孩子；RTag=1 rchild 域指示节点的前驱



节点结构的定义：为了仿照线性表的存储结构，在二叉树的线索链表上添加一个头结点，令其 lchild 域指向二叉树的根节点，其 rchild 域的指针指向中序遍历访问的最后一个节点。反之，令二叉树中序序列中的第一个节点的 lchild 域指针和最后一个节点的 rchild 域的指针均指向头结点.

```
int data;
int LTag; // 0,1
int RTag; // 0,1
TBTreeNode lchild;
TBTreeNode rchild;
```

22).Huffman 树的定义以及使用

赫夫曼树（Huffman Tree），又称最优二叉树，是一类带权路径长度最短的树。假设有 n 个权值{w1,w2,...,wn}，如果构造一棵有 n 个叶子节点的二叉树，而这 n 个叶子节点的权值是{w1,w2,...,wn}，则所构造出的带权路径长度最小的二叉树就被称为赫夫曼树(树的带权路径长度指树中所有叶子节点到根节点的路径长度与该叶子节点权值的乘积之和，如果在一棵二叉树中共有 n 个叶子节点，用 Wi 表示第 i 个叶子节点的权值，Li 表示第 i 个也叶子节点到根节点的路径长度，则该二叉树的带权路径长度 WPL=W1*L1 + W2*L2 + ... Wn*Ln).

赫夫曼树的构建步骤如下：

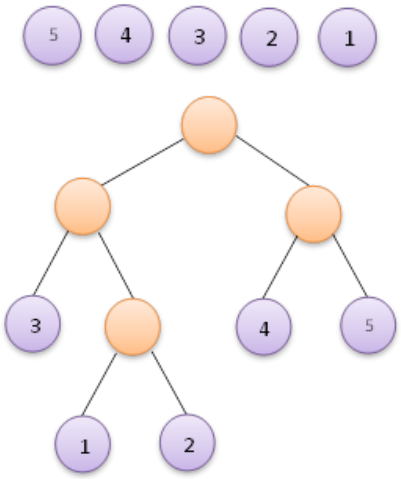
1)、将给定的 n 个权值看做 n 棵只有根节点（无左右孩子）的二叉树，组成一个集合 HT，每棵树的权值为该节点的权值。

2)、从集合 HT 中选出 2 棵权值最小的二叉树，组成一棵新的二叉树，其权值为这 2 棵二叉树的权值之和。

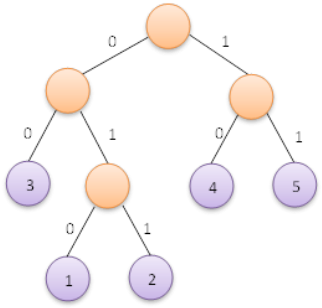
3)、将步骤 2 中选出的 2 棵二叉树从集合 HT 中删去，同时将步骤 2 中新得到的二叉树加入到集合 HT 中。

4)、重复步骤 2 和步骤 3，直到集合 HT 中只含一棵树，这棵树便是赫夫曼树。

假如给定如下 5 个权值：



Huffman 编码: 满足 Huffman 二叉树的结构的二进制编码(满足前缀编码:即任意一个字符的编码都不是另一个字符编码的前缀)



权值为 5 的也自己节点的赫夫曼编码为：11

权值为 4 的也自己节点的赫夫曼编码为：10

权值为 3 的也自己节点的赫夫曼编码为：00

权值为 2 的也自己节点的赫夫曼编码为：011

权值为 1 的也自己节点的赫夫曼编码为：010

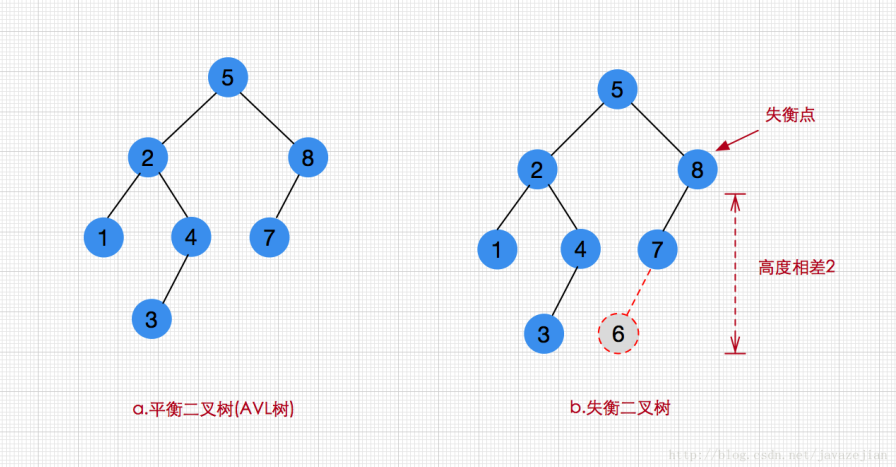
23).平衡二叉树的性质和优势以及操作

平衡二叉查找树，又称 AVL 树。 它除了具备二叉查找树的基本特征之外，还具有一个非常重要的特点：它 的左子树和右子树都是平衡二叉树，且左子树和右子树的深度之差的绝对值（**平衡因子**） 不超过 1。 也就是说 AVL 树每个节点的平衡因子只可能是-1、0 和（左子树高度减去右子树高度）.

如何保证平衡：当在二叉排序树中插入一个节点时，首先检查是否因插入而破坏了平衡，若破坏，则找出其中的最小不平衡二叉树，在保持二叉排序树特性的情况下，调整最小不平衡子树中节点之间的关系(旋转进行调整,左旋,右旋操作)，以达到新的平衡。所谓最小不平衡子树 指离插入节点最近且以平衡因子的绝对值大于 1 的节点作为根的子树.

平衡二叉树相比普通二叉查找树的优势：平衡二叉树的优势在于不会出现普通二叉查找树的最差情况。其查找的时间复杂度为 O(logN)， N 为 AVL 数的节点数.

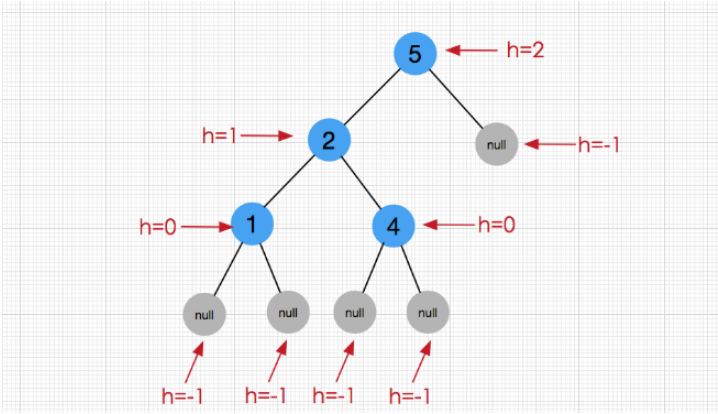
平衡二叉树的缺点：为了保证高度平衡，动态插入和删除的代价也随之增加；在大数据量查找环境下(比如说系统磁盘里的文件目录，数据库中的记录查询 等)，所有的二叉查找树结构(BST、AVL、RBT)都不合适。如此大规模的数据量（几 G 数据），全部组织成平衡二叉树放在内存中是不可能做到的。假如构造的平衡二叉树深度有 1W 层。那么从根节点出发到叶子节点很可能就需要 1W 次的硬盘 IO 读写。大家都知道，硬盘的机械部件读写数据的速度远远赶不上纯电子媒体的内存。 查找效率在 IO 读写过程中将会付出巨大的代价。在大规模数据查询这样一个实际应用背景下，平衡二叉树的效率就很成问题了.



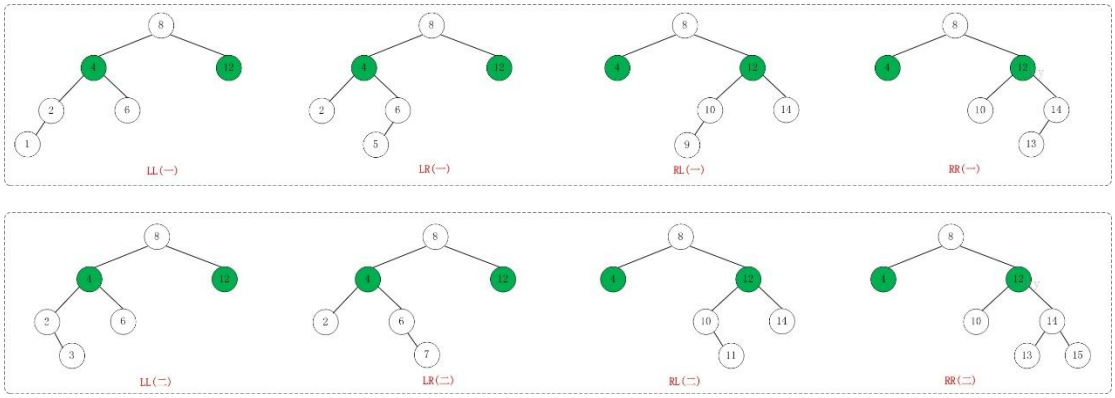

```
public class AVLNode<T extends Comparable> {
    public AVLNode<T> left;//左结点
    public AVLNode<T> right;//右结点
    public T data;
    public int height;//当前结点的高度
    public AVLNode(T data) {
        this(null,null,data);
    }
    public AVLNode(AVLNode<T> left, AVLNode<T> right, T data) {
        this(left,right,data,0);
    }
    public AVLNode(AVLNode<T> left, AVLNode<T> right, T data, int height) {
        this.left=left;
        this.right=right;
```

```
            this.data=data;
            this.height = height;
        }
}
```

高度和深度一组相反的概念，高度是指当前结点到叶子结点的最长路径，如所有叶子结点的高度都为 0，而深度则是指从根结点到当前结点的最大路径，如根结点的深度为 0。这里约定空结点（空子树）的高度为-1，叶子结点的高度为 0，非叶子节点的高度则根据其子树的高度而计算获取，如下图:



AVL 树的非平衡 8 中情况如下:



i). **LL**：LeftLeft，也称为"左左"。插入或删除一个节点后，根节点的左子树的左子树还有非空子节点，导致"根的左子树的高度"比"根的右子树的高度"大 2，导致 AVL 树失去了平衡。

例如，在上面 LL 情况中，由于"根节点(8)的左子树(4)的左子树(2)还有非空子节点"，而"根节点(8)的右子树(12)没有子节点"；导致"根节点(8)的左子树(4)高度"比"根节点(8)的右子树(12)"高 2。

ii). **LR**：LeftRight，也称为"左右"。插入或删除一个节点后，根节点的左子树的右子树还有非空子节点，导致"根的左子树的高度"比"根的右子树的高度"大 2，导致 AVL 树失去了平衡。

例如，在上面 LR 情况中，由于"根节点(8)的左子树(4)的左子树(6)还有非空子节点"，而"根节点(8)的右子树(12)没有子节点"；导致"根节点(8)的左子树(4)高度"比"根节点(8)的右子树(12)"高 2

AVL 树的旋转操作:

iii). **RL**：RightLeft，称为"右左"。插入或删除一个节点后，根节点的右子树的左子树还有非空子节点，导致"根的右子树的高度"比"根的左子树的高度"大 2，导致 AVL 树失去了平衡。

例如，在上面 RL 情况中，由于"根节点(8)的右子树(12)的左子树(10)还有非空子节

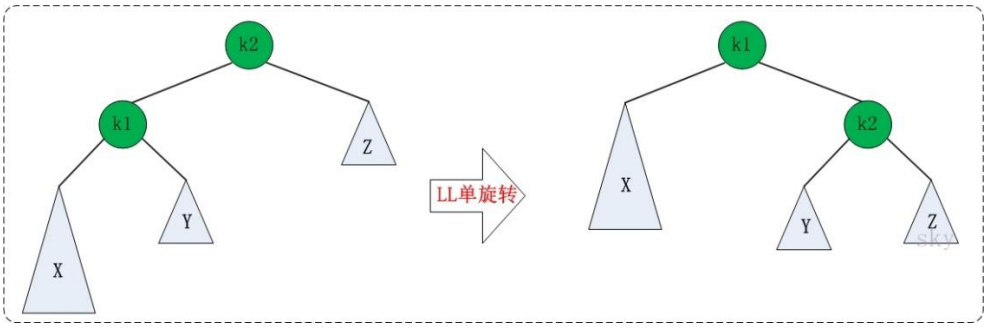
点"，而"根节点(8)的左子树(4)没有子节点"；导致"根节点(8)的右子树(12)高度"比"根节点(8)的左子树(4)"高 2.

iv). **RR**：RightRight，称为"右右"。插入或删除一个节点后，根节点的右子树的右子树还有非空子节点，导致"根的右子树的高度"比"根的左子树的高度"大 2，导致 AVL 树失去了平衡。

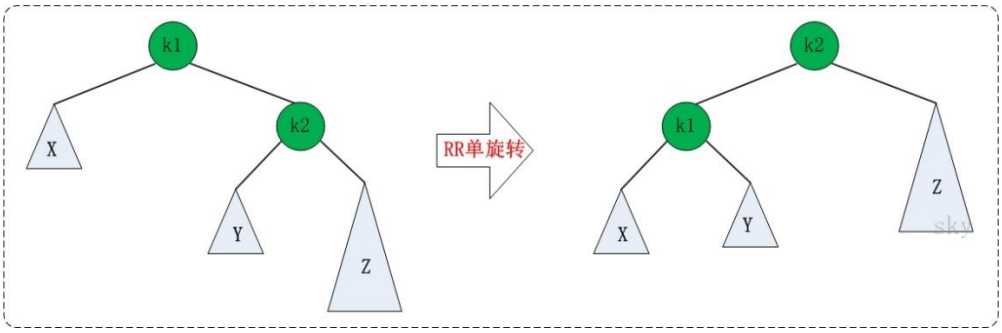
例如，在上面 RR 情况中，由于"根节点(8)的右子树(12)的右子树(14)还有非空子节点"，而"根节点(8)的左子树(4)没有子节点"；导致"根节点(8)的右子树(12)高度"比"根节点(8)的左子树(4)"高 2.

如果在 AVL 树中进行插入或删除节点后，可能导致 AVL 树失去平衡。AVL 失去平衡之后，可以通过旋转使其恢复平衡，下面分别介绍"LL(左左)，LR(左右)，RR(右右)和 RL(右左)"这 4 种情况对应的旋转方法：

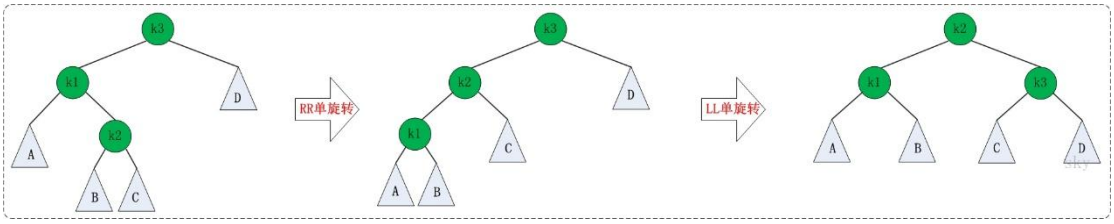
LL 的旋转：



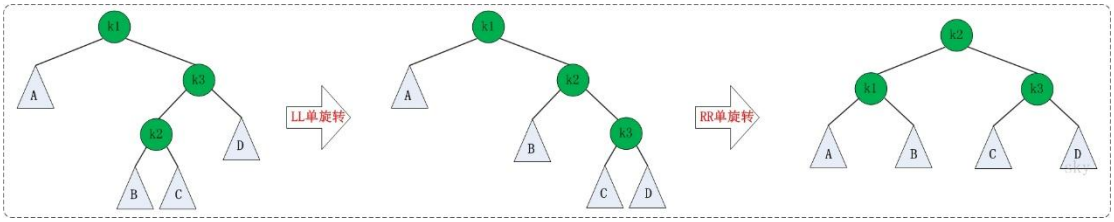
RR 的旋转：



LR 的旋转： LR 失去平衡的情况，需要经过两次旋转才能让 AVL 树恢复平衡

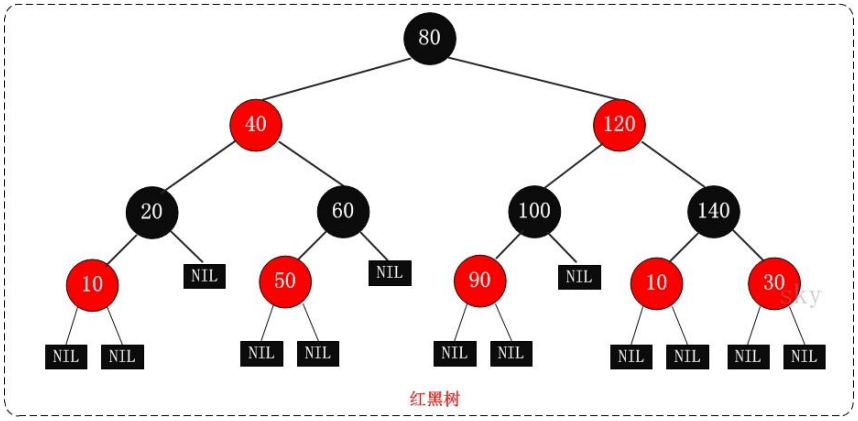


RL 的旋转：



24).平衡二叉树:输入一个二叉树的根节点,判断该树是不是平衡二叉树,如果某二叉树中任意节点的左右子树的深度相差不超过 1,那么它就是一棵平衡二叉树.

25).红黑树的性质优势以及操作



红黑树的特点:

i).每个节点要么是黑色,要么是红色;

ii).根节点是黑色的;

iii).每个叶节点(NIL)是黑色的(此处的叶节点是只为 NIL 或 NULL 的结点,而不是如上图中的红色节点 10);

iv).如果一个节点是红色的, 则它的子节点必须是黑色的;

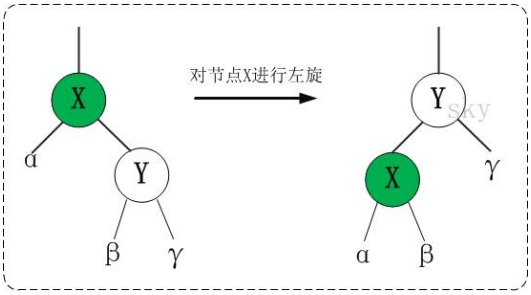
v).从一个节点到该节点的子孙节点的所有路径上包含相同数目的黑节点(所以红黑树是相对是接近平衡的二叉树).

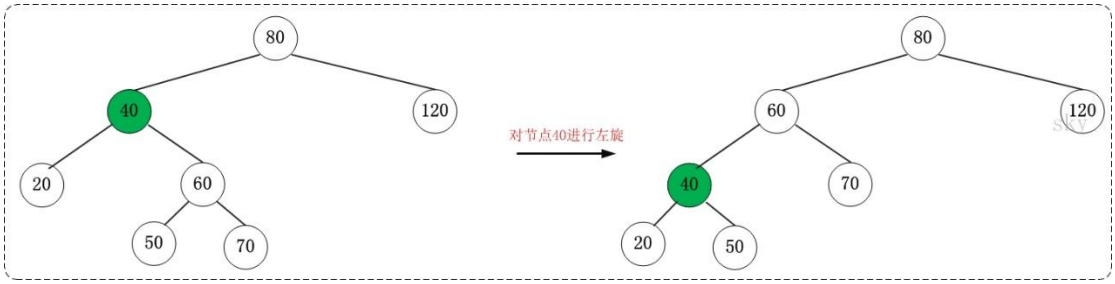
红黑树的应用:红黑树主要用来存储有序的数据,它的时间复杂度为 O(lgn), 效率非常高, Java 中的 TreeSet 和 TreeMap, C++ STL 中的 set, map 以及 Linux 虚拟内存的管理.

vi).**一棵含有 n 个节点的红黑树的高度至多为 2log(n+1)**

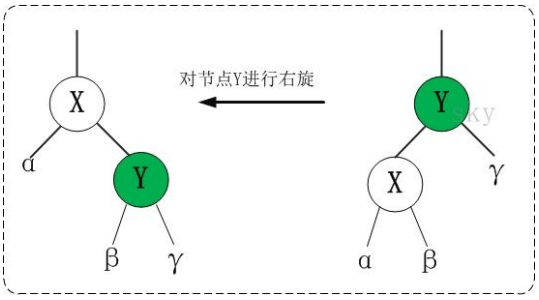
红黑树的操作: 红黑树的基本操作是**添加**、**删除**。在对红黑树进行添加或删除之后，都会用到旋转方法用来重新构造一个满足上述 5 条性质的红黑树.

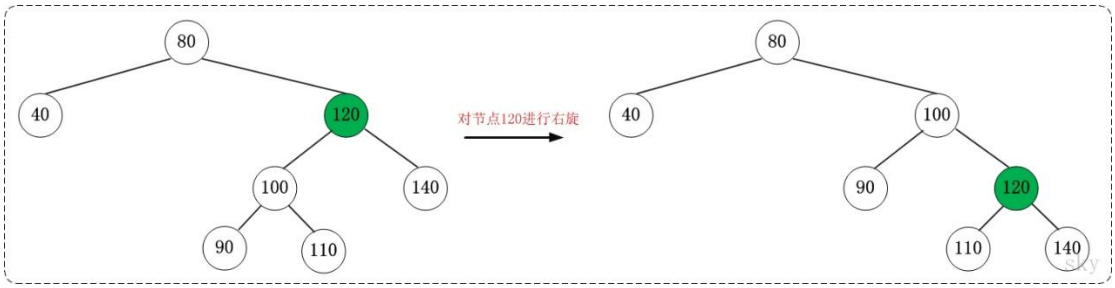
左旋操作:





右旋操作:





添加元素:

i).将红黑树当做一棵二叉树查找树,首先查找要插入的位置,时间复杂度为 O(N),将节点插入:红黑树本身就是一个二叉查找树,将节点插入后,仍然是一棵二叉查找树,即插入后树的键值仍然是有序的.

ii).将插入的结点着色为红色: 着色为红色不违背特性 5, 少违背一条特性对后续操作会方便很多, 插入节点着色为红色不会违背特性 1,2,3; 而会违背特性 4;

iii).对上述操作进行调整使其满足特性 4: 如果被插入的结点是根节点,则着色为黑色; 如果被插入的结点父节点是黑色的则直接插入; 如果被插入的结点的父节点是红色的,则不满足特性 5,则插入节点是一定存在非空祖父节点的.即重新调整使其满足红黑树的所有性质.

删除操作:

i).查找要删除的位置,时间复杂度为 O(N);

ii).用删除节点后继或者节点替换该结点(值进行数据替换即可,不必调整指针,后继节点是中序遍历紧挨着该节点的结点,即有孩子的最左孩子节点);

iii).如果删除节点的替换节点为黑色,则需要重新调整.

红黑树的 Java 数据结构:

```
public class RBTree<T extends Comparable<T>> {
    private RBTNode<T> mRoot;    // 根结点
    private static final boolean RED   = false;
    private static final boolean BLACK = true;
    public class RBTNode<T extends Comparable<T>> {
        boolean color;        // 颜色
        T key;                // 关键字(键值)
        RBTNode<T> left;    // 左孩子
        RBTNode<T> right;    // 右孩子
        RBTNode<T> parent;    // 父结点
        public RBTNode(T key, boolean color, RBTNode<T> parent, RBTNode<T> left, RBTNode<T>
right) {
            this.key = key;
            this.color = color;
```

```
            this.parent = parent;
            this.left = left;
            this.right = right;
        }
    }
}
```

左旋操作 Java:

```
/*
* 左旋示意图(对节点 x 进行左旋)：
 *      px                              px
 *     /                               /
 *    x                               y
 *   / \      --(左旋)-.           / \                #
 *  lx   y                          x  ry
 *     /   \                       /  \
 *    ly   ry                     lx  ly
*/
private void leftRotate(RBTNode<T> x) {
    // 设置 x 的右孩子为 y
    RBTNode<T> y = x.right;
    // 将 “y 的左孩子” 设为 “x 的右孩子”；
    // 如果 y 的左孩子非空，将 “x” 设为 “y 的左孩子的父亲”
    x.right = y.left;
    if (y.left != null)
        y.left.parent = x;
    // 将 “x 的父亲” 设为 “y 的父亲”
    y.parent = x.parent;
    if (x.parent == null) {
        this.mRoot = y;            // 如果 “x 的父亲” 是空节点，则将 y 设为根节点
    } else {
        if (x.parent.left == x)
            x.parent.left = y;    // 如果 x 是它父节点的左孩子，则将 y 设为“x 的父节点的左孩子”
        else
            x.parent.right = y;    // 如果 x 是它父节点的右孩子，则将 y 设为“x 的父节点的右孩子”
    }
    // 将 “x” 设为 “y 的左孩子”
    y.left = x;
    // 将 “x 的父节点” 设为 “y”
    x.parent = y;
}
```

右旋操作 Java:

```
/*
 * 对红黑树的节点(y)进行右旋转
 * 右旋示意图(对节点 y 进行左旋)：
```

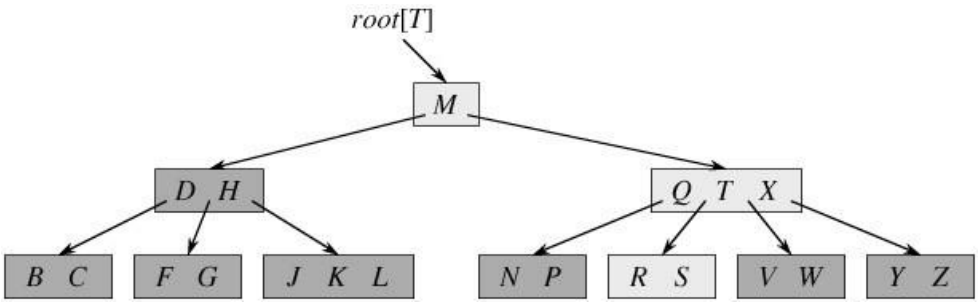
```
 *            py                              py
 *           /                               /
 *          y                               x
 *         / \      --(右旋)-.            / \                #
 *        x   ry                          lx   y
 *       / \                                 / \               #
 *      lx  rx                              rx  ry
 */
private void rightRotate(RBTNode<T> y) {
    // 设置 x 是当前节点的左孩子。
    RBTNode<T> x = y.left;
    // 将 “x 的右孩子” 设为 “y 的左孩子”；
    // 如果"x 的右孩子"不为空的话，将 “y” 设为 “x 的右孩子的父亲”
    y.left = x.right;
    if (x.right != null)
        x.right.parent = y;
    // 将 “y 的父亲” 设为 “x 的父亲”
    x.parent = y.parent;
    if (y.parent == null) {
        this.mRoot = x;            // 如果 “y 的父亲” 是空节点，则将 x 设为根节点
    } else {
        if (y == y.parent.right)
            y.parent.right = x;    // 如果 y 是它父节点的右孩子，则将 x 设为“y 的父节点的右孩子”
        else
            y.parent.left = x;    // (y 是它父节点的左孩子) 将 x 设为“x 的父节点的左孩子”
    }
    // 将 “y” 设为 “x 的右孩子”
    x.right = y;
    // 将 “y 的父节点” 设为 “x”
    y.parent = x;
}
```

26).B 树 B+树的优势和性质

大规模数据存储中，实现索引查询这样一个实际背景下，树节点存储的元素数量是有限的（如果元素数量非常多的话，查找就退化成节点内部的线性查找了），这样导致基于二叉查找树结构(二叉查找树,AVL 树,红黑树)由于**树的深度过大而造成磁盘 I/O 读写过于频繁，进而导致查询效率低下**，那么如何减少树的深度（当然是不能减少查询的数据量），一个基本的想法就是：采用**多叉树**结构（由于树节点元素数量是有限的，自然该节点的子树数量也就是有限的）.

B 树:B 树是为了磁盘或其它存储设备而设计的一种多叉（下面你会看到，相对于二叉，B 树每个内结点有多个分支，即多叉）平衡查找树。与本 blog 之前介绍的红黑树很相似，但在降低磁盘 I/O 操作方面要更好一些。许多数据库系统都一般使用 B 树或者 B 树的各种变形结构.B 树与红黑树最大的不同在于，B 树的结点可以有许多子女，从几个到几千个。那为什么又说 B 树与红黑树很相似呢?因为与红黑树一样，一棵含 n 个结点的 B 树的高度也为 O(lgn)，

但可能比一棵红黑树的高度小许多，应为它的分支因子比较大。所以，B 树可以在 O（logn）时间内，实现各种如插入（insert），删除（delete）等动态集合操作.



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.

用阶定义 B 树:

B 树又叫平衡多路查找树。**一棵** m 阶的 B 树:

i). 树中每个结点最多含有 m 个孩子（m>=2）

ii). 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）；

iii). 若根结点不是叶子结点，则至少有 2 个孩子（特殊情况：没有孩子的根结点，即根结点为叶子结点，整棵树只有一个根节点）；

iv). 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(可以看做是外部接点或查询失败的接点，实际上这些结点不存在，指向这些结点的指针都为 null);

v). 每个非终端结点中包含有 n 个关键字信息： (n，P0，K1，P1，K2，P2，……，Kn，Pn)。其中: Ki (i=1...n)为关键字，且关键字按顺序升序排序 K(i-1)< Ki; Pi 为指向子树根的接点，且指针 P(i-1)指向子树种所有结点的关键字均小于 Ki，但都大于 K(i-1); 关键字的个数 n 必须满足：[ceil(m / 2)-1]<= n <= m-1.

B 树的数据结构定义:

```
#define m 1024
struct BTNode;
typedef struct BTNode * PBTNode;
struct BTNode  {
    int  keyNum; /* 实际关键字个数，keyNum<m */
    PBTNode  parent;                /* 指向父结点*/
    PBTNode  *ptr;    /* 子树指针向量: ptr[0]…ptr[keyNum] */
    KeyType  *key;    /* 关键字向量: key [0]…key [keyNum－1] */
  }
typedef struct BTNode *BTree;
typedef BTree * PBTree;
```

B 树的高度: 若 n≥1，m≥3，则对任意一棵具有 n 个关键字的 m 阶 B-树，其树高 h 至多为 logt((n+1)/2)+1，这里 t 是每个(除根外)内部结点的最小度数，即 t=[m/2]. B-树的高度为 O(logtn)。于是在 B-树上查找、插入和删除的读写盘的次数为 O(logtn)，CPU 计算时间为 O(mlogtn).

B+树: B+树是应文件系统所需而产生的一种 B-tree 的变形树.

一棵 m 阶的 B+树和 m 阶的 B 树的异同点在于:

i). 有 n 棵子树的结点中含有 n-1 个关键字； B 树 n 棵子树的结点中含有 n 个关键字;

ii). 所有的叶子结点中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，且叶子结点本身依关键字的大小自小而大的顺序链接。(而 B 树的叶子节点并没有包括全部需要查找的信息);

iii). **所有的非终端结点可以看成是索引部分**，结点中仅含有其子树根结点中最大（或最小）关键字。 (而 B 树的非终节点也包含需要查找的有效信息).

为什么 B+树比 B 树更适合实际应用中操作系统的文件索引和数据库索引:

i).B+tree 的磁盘读写代价更低, B+树内部结点并没有指向关键字具体信息的指针。因此其内部结点相对 B 树更小。如果把所有同一内部结点的关键字存放在同一盘块中，那么盘块所能容纳的关键字数量也越多。一次性读入内存中的需要查找的关键字也就越多。相对来说 IO 读写次数也就降低了;

ii).查询效率稳定: 由于非终结点并不是最终指向文件内容的结点，而只是叶子结点中关键字的索引。所以任何关键字的查找必须走一条从根结点到叶子结点的路。所有关键字查询的路径长度相同，导致每一个数据的查询效率相当.

B 树的插入操作:

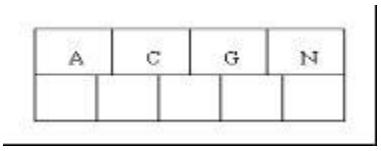
对高度为h的m阶B树，新结点一般是插在第h层。通过检索可以确定关键码应插入的结点位置。然后分两种情况讨论：

1，若该结点中关键码个数小于m-1，则直接插入即可。

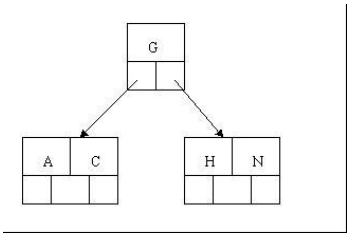
2，若该结点中关键码个数等于m-1，则将引起结点的分裂。以中间关键码为界将结点一分为二，产生一个新结点，并把中间关键码插入到父结点（h-1层）中；

重复上述工作，最坏情况一直分裂到根结点，建立一个新的根结点，整个B树增加一层。

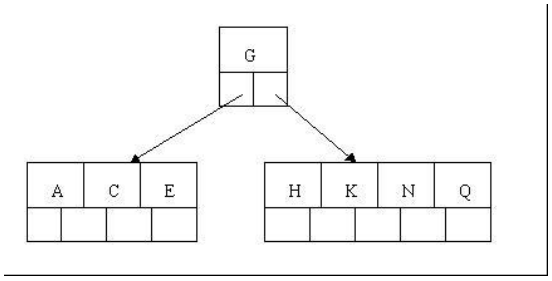
i).插入以下字符字母到一棵空的 B 树中（非根结点**关键字数**小了（小于 2 个）就合并，大了（超过 4 个）就分裂）：C N G A H E K Q M F W L T Z D P R X Y S，首先，结点空间足够，4 个字母插入相同的结点中，如下图：



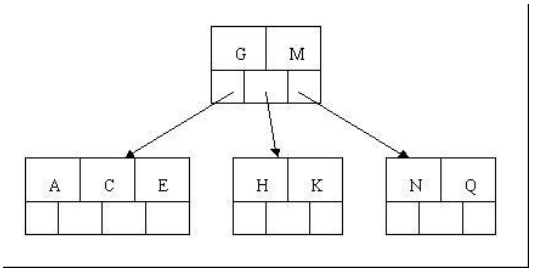
ii). 当试着插入 H 时，结点发现空间不够，以致将其分裂成 2 个结点，移动中间元素 G 上移到新的根结点中，在实现过程中，咱们把 A 和 C 留在当前结点中，而 H 和 N 放置新的其右邻居结点中。如下图：



iii). 当插入 E,K,Q 时，不需要任何分裂操作:



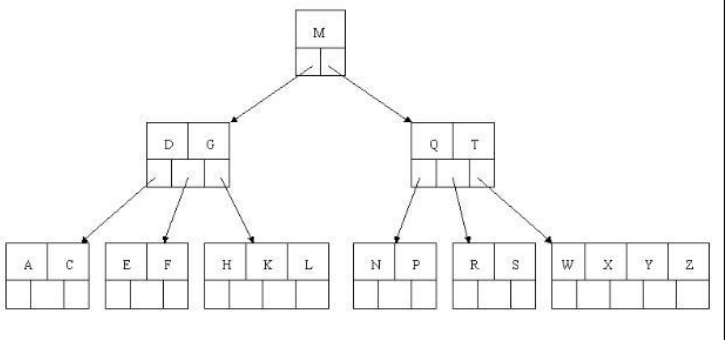
iv). 插入 M 需要一次分裂，注意 M 恰好是中间关键字元素，以致向上移到父节点中:



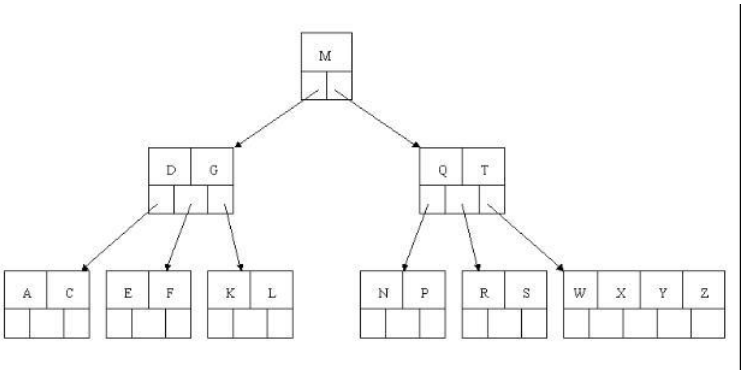
按照此规律一次插入.

B 树的删除操作: 删除元素，移动相应元素之后，如果某结点中元素数目（即关键字数）小于 ceil(m/2)-1，则需要看其某相邻兄弟结点是否丰满（结点中元素个数大于 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，即借了之后其结点数目小于 ceil(m/2)-1，则该结点与其相邻的某一兄弟结点进行“合并”成一个结点，以此来满足条件

一棵 5 阶 B 树（树中最多含有 m（m=5）个孩子，因此关键字数最小为 **ceil(m / 2)-1=2。**还是这句话，关键字数小了（小于 2 个）就合并，大了（超过 4 个）就分裂）：

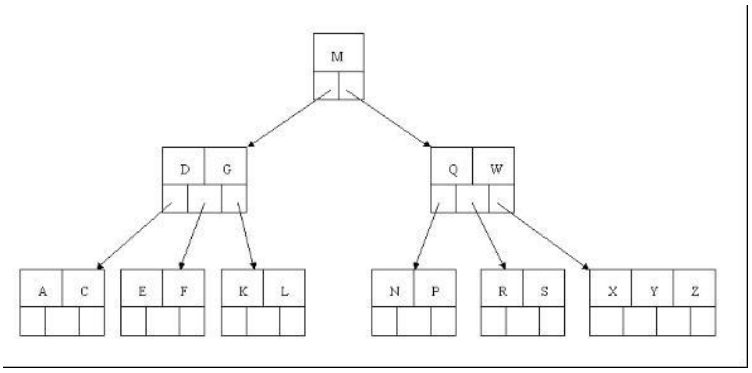


删除 H，首先查找 H，H 在一个叶子结点中，且该叶子结点元素数目 3 大于最小元素数目 ceil(m/2)-1=2，只需要移动 K 至原来 H 的位置，移动 L 至 K 的位置（也就是结点中删除元素后面的元素向前移动）：

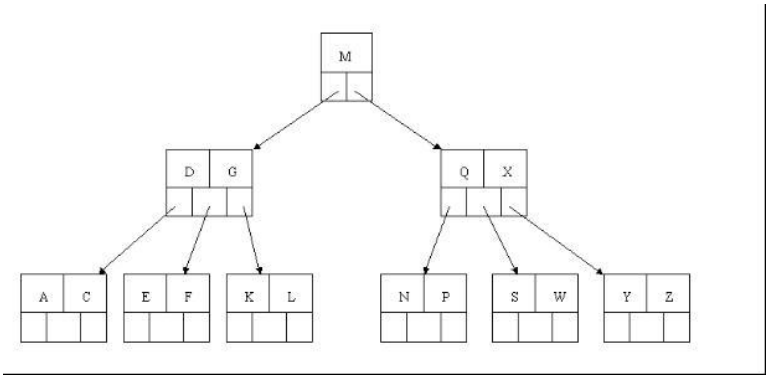


删除 T,因为 T 没有在叶子结点中，而是在中间结点中找到，咱们发现他的继承者 W(字母升序的

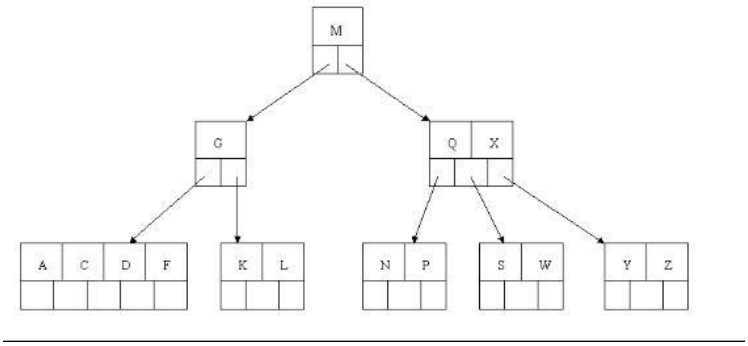
下个元素)，将 W 上移到 T 的位置，然后将原包含 W 的孩子结点中的 W 进行删除，这里恰好删除 W 后，该孩子结点中元素个数大于 2，无需进行合并操作:



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:

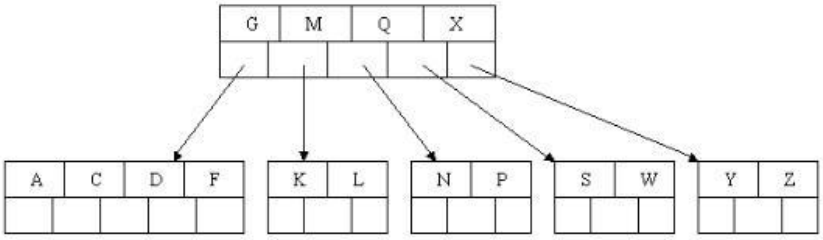


删除 E， 删除后会导致很多问题，因为 E 所在的结点数目刚好达标，刚好满足最小元素个数（ceil(5/2)-1=2）,而相邻的兄弟结点也是同样的情况，删除一个元素都不能满足条件，所以需要该节点与某相邻兄弟结点进行合并操作；首先移动父结点中的元素（该元素在两个需要合并的两个结点元素之间）下移到其子结点中，然后将这两个结点进行合并成一个结点。所以在该实例中，咱们首先将父节点中的元素 D 下移到已经删除 E 而只有 F 的结点中，然后将含有 D 和 F 的结点和含有 A,C 的相邻兄弟结点进行合并成一个结点:

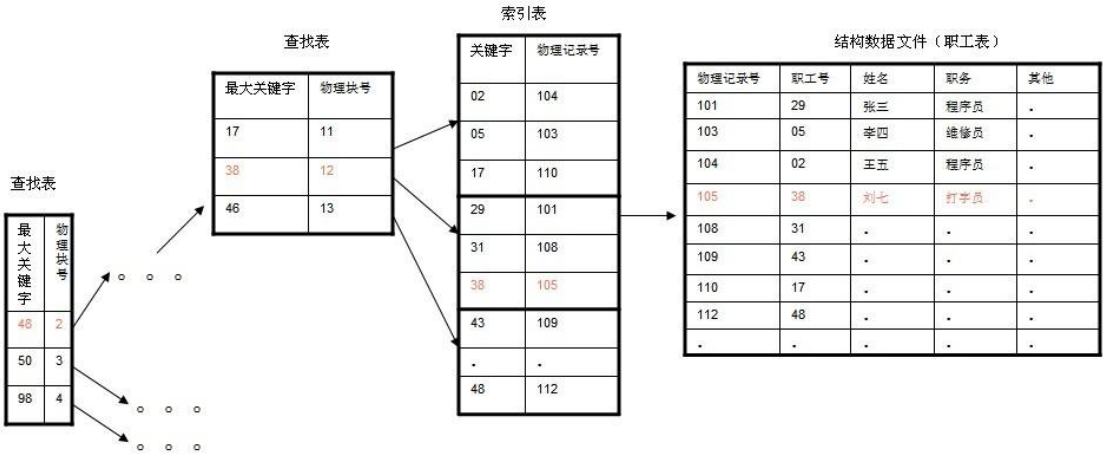


父节点只包含一个元素 G，没达标（因为非根节点包括叶子结点的关键字数 n 必须满足于 2=<n<=4，而此处的 n=1），这是不能够接受的。如果这个问题结点的相邻兄弟比较丰满，则可以向父结点

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:



为什么数据库索引采用 B+树而不是 B 树: B 树必须用中序遍历的方法按序扫库，而 B+树直接从叶子结点挨个扫一遍就完了，B+树支持 range-query 非常方便，而 B 树不支持。这是数据库选用 B+树的最主要原因.

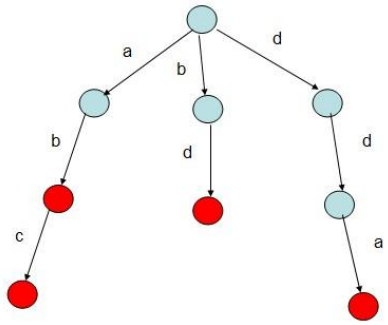


比如要查 5-10 之间的，B+树一把到 5 这个标记，再一把到 10，然后串起来就行了，B 树就非常麻烦。B 树的好处，就是成功查询特别有利，因为树的高度总体要比 B+树矮。不成功的情况下，B 树也比 B+树稍稍占一点点便宜; 有很多基于频率的搜索是选用 B 树，越频繁 query 的结点越往根上走，前提是需要对 query 做统计，而且要对 key 做一些变化; mysql 底层存储是用 B+树实现的，内存中 B+树是没有优势的，但是一到磁盘，B+树的威力就出来了.

27).Trie 树的性质以及作用

字典树（Trie）可以保存一些字符串->值的对应关系，它跟 Java 的 HashMap 功能相同，都是 key-value 映射，只不过 Trie 的 key 只能是字符串. Trie 的强大之处就在于它的时间复杂度。它的插入和查询时间复杂度都为 O(k) ，其中 k 为 key 的长度，与 Trie 中保存了多少个元素无关。Hash 表号称是 O(1) 的，但在计算 hash 的时候就肯定会是 O(k) ，而且还有碰撞之类的问题；Trie 的缺点是空间消耗很高.

如给出字符串"abc","ab","bd","dda"，根据该字符串序列构建一棵 Trie 树。则构建的树如下:




```
typedef struct Trie_node
{
    int count;                  // 统计单词前缀出现的次数
    struct Trie_node* next[26]; // 指向各个子树的指针
    bool exist;                 // 标记该结点处是否构成单词
}TrieNode , *Trie;
```

Trie 树的根结点不包含任何信息，第一个字符串为"abc"，第一个字母为'a'，因此根结点中数组 next 下标为'a'-97 的值不为 NULL，其他同理，构建的 Trie 树如图所示，红色结点表示在该处可以构成一个单词。很显然，如果要查找单词"abc"是否存在，查找长度则为 O(len)，len 为要查找的字符串的长度。而若采用一般的逐个匹配查找，则查找长度为 O(len*n)，n 为字符串的个数。显然基于 Trie 树的查找效率要高很多. 因为当查询如字符串 abc 是否为某个字符串的前缀时，显然以 b、c、d....等不是以 a 开头的字符串就不用查找了，这样迅速缩小查找的范围和提高查找的针对性。所以建立 Trie 的复杂度为 O(n*len).

Trie 树的应用:

i).统计 10 个频繁出现的词,在一个文本文件中有 1 万行,每行一个单词,要求统计出其中出现次数最频繁的 10 个单词:用 Trie 树统计每个词出现的次数,时间复杂度为 O(nl)(l 为单词的平均长度),最终找出出现最频繁的前 10 个词(可用堆实现,时间复杂度为 O(nlog)10).

ii).寻找热门查询:日志中每个查询穿的长度为 1~255 字节,假设有 1000 万条记录(查询的串的重复度比较高,去重后不超过 300 万),统计查询频率最高的 10 个串,内存不超过 1GB:可以用 Trie 树,观察关键字在该查询串出现的次数,没有出现则为 0,最后用 10 个元素的小顶堆来对出现的频率进行排序.

5.查找表

1).二分查找

```
public static int search(int[] arr, int key) {
        int start = 0;
        int end = arr.length - 1;
        while (start <= end) {
            int middle = (start + end) / 2;
            if (key < arr[middle]) {
                end = middle - 1;
            } else if (key > arr[middle]) {
                start = middle + 1;
```

```
            } else {
                return middle;
            }
        }
        return -1;
    }
```

2).二叉查找树

二叉排序树：或者是一棵空树，或者是具有下列性质的二叉树:

i). 若它的左子树不空，则左子树上所有结点的值均小于它的根结点的值

ii). 若它的右子树不空，则右子树上所有结点的值均大于它的根结点的值

iii). 它的左、右子树也分别为二叉排序树

3)hash 表的性质以及应用

散列表（Hash Table，也叫哈希表），是根据**关键码值**（key value）而进行直接访问的数据结构。它通过关键码值映射到表中的一个位置来访问记录，以加快访问的速度。这个**映射函数**叫做**散列函数**，存放的**数组**叫做**散列表**。

例如，关键字为 k，则把 k 放在 f(k)的存储位置上。由此，不需要比较便可直接获取所查的记录。f()就是这个散列函数，按这个思想建立的表叫做散列表。

对于不同的关键字，可能得到同一散列地址，即 k1 ≠ k2,而 f(k1) =f(k2),这种现象成为**碰撞(Collision)**.具有相同的函数值得关键字对该散列函数来说叫做同义词。 根据散列函数 f(k)和处理碰撞的方法将一组关键字映射到一段有限的连续的地址集上(区间)，并以**关键字在地址集中的"像"作为记录在表中的从存储位置**，这个表叫做散列表，这一映射过程叫做散列造表或者散列，所得的存储地址叫做散列地址。

对于任何一个关键字，经过散列函数映象到地址集合中任何一个地址上的概率是相等的，则称此类散列函数为均匀散列函数，这就是使关键字经过散列函数得到的一个"随机地址"，从而减少碰撞。

实际使用中根据不同的情况来选择相应的散列函数，通常考虑的因素有:

i)计算哈希函数所需时间;

ii)关键字的长度 ;

iii)哈希表的大小;

iv)关键字的分布情况 ;

v)记录的查找频率.

通常有一下几种**哈希函数:**

直接寻址法 ：取关键字或关键字的某个线性函数作为散列地址。即 H(key)=key 或者 H(key) = a*key + b,(a,b 为常数)（这种散列函数又叫做自身函数）。若其中 H(key)中已经有值了，就往下一个找，直到 H(key)中没有值了，就放进去;

数字分析法：分析一组数据，比如一组员工的出生年月日，这时我们发现出生年月日的前几位数字大体相同，这样的话，出现冲突的几率就会很大，但是我们发现年月日的后几位表示月份和具体日期的数字差别很大，如果用后面的数字来构成散列地址，则冲突的几率会明显降低。因此数字分析法就是找出数字的规律，尽可能利用这些数据来构造冲突几率较低的散列地址;

平方取中法 :当无法确定关键字中哪几位分布比较均匀时，可以先求出关键字的平方值，然后按需要取平方值的中间几位作为哈希地址。（原因：平方后中间几位和关键字中的每一位

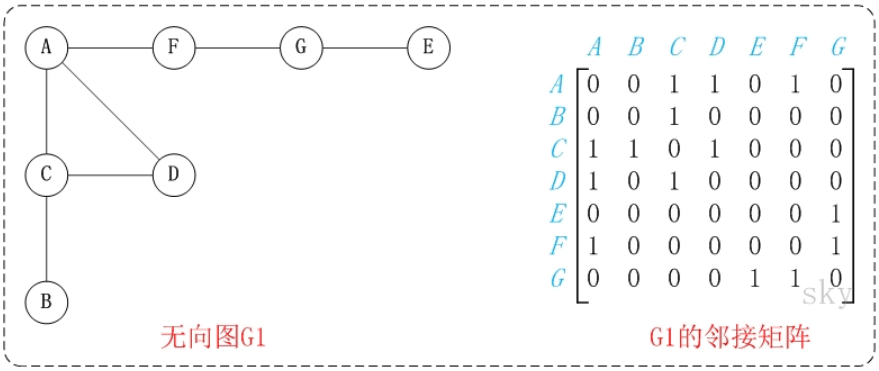
都有关，故不同关键字会以较高的概率产生不同的哈希地址）；
折叠法：将关键字分割成位数相同的几部分，最后一部分位数可以不同，然后取这几部分的叠加和（去除进位）作为散列地址。数位叠加可以有移位叠加和间界叠加两种方法。移位叠加是将分割后的每一部分的最低位对齐，然后相加；间界叠加是从一端向另一端沿分割界来回折叠，然后对齐相加;
除留余数法：取关键字被某个不大于散列表表长 m 的数 p 除后所得的余数为散列地址。即 H(key) = key MOD p,p<=m。不仅可以对关键字直接取模，也可在折叠、平方取中等运算之后取模。对 p 的选择很重要，一般取素数或 m，若 p 选的不好，容易产生同义词.
处理冲突的方法:
i)开放寻址法：Hi=(H(key) + di) MOD m,i=1,2，…，k(k<=m-1)，其中 H(key）为散列函数，m 为散列表长，di 为增量序列，可有下列三种取法：
a). di=1,2,3，…，m-1，称线性探测再散列；
b). di=1^2,-1^2,2^2,-2^2，⑶^2，…，±（k)^2,(k<=m/2）称二次探测再散列；
c). di=伪随机数序列，称伪随机探测再散列。
ii)再散列法：Hi=RHi(key),i=1,2，…，k RHi 均是不同的散列函数，即在同义词产生地址冲突时计算另一个散列函数地址，直到冲突不再发生，这种方法不易产生“聚集”，但增加了计算时间;
iii)链地址法（拉链法）

6.图

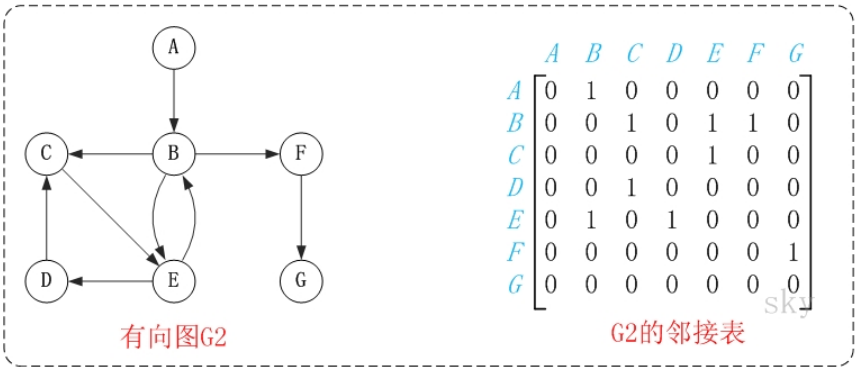
1) .图的存储结构

图(graph)是由一些点(vertex)和这些点之间的连线(edge)所组成的；其中，点通常被成为"顶点(vertex)"，而点与点之间的连线则被成为"边或弧"(edege)。通常记为，G=(V,E);
根据边是否有方向，将图可以划分为：**无向图**和**有向图.**
邻接点: 一条边上的两个顶点叫做邻接点; 在有向图中，除了邻接点之外；还有"入边"和"出边"的概念。
顶点的入边，是指以该顶点为终点的边。而顶点的出边，则是指以该顶点为起点的边
度: 在无向图中，某个顶点的度是邻接到该顶点的边(或弧)的数目); 在有向图中，度还有"入度"和"出度"之分, 某个顶点的入度，是指以该顶点为终点的边的数目。而顶点的出度，则是指以该顶点为起点的边的数目;顶点的度=入度+出度.
路径：如果顶点(Vm)到顶点(Vn)之间存在一个顶点序列。则表示 Vm 到 Vn 是一条路径。
路径长度：路径中"边的数量"。
简单路径：若一条路径上顶点不重复出现，则是简单路径。
回路：若路径的第一个顶点和最后一个顶点相同，则是回路。
简单回路：第一个顶点和最后一个顶点相同，其它各顶点都不重复的回路则是简单回路;
连通图：对无向图而言，任意两个顶点之间都存在一条无向路径，则称该无向图为连通图。对有向图而言，若图中任意两个顶点之间都存在一条有向路径，则称该有向图为强连通图;
连通分量：非连通图中的各个连通子图称为该图的连通分量;
邻接矩阵: 邻接矩阵是指用矩阵来表示图。它是采用矩阵来描述图中顶点之间的关系(及弧或边的权), 假设图中顶点数为 n，则邻接矩阵定义为:

$$A[i][j]=\begin{cases}1 & (\text{若Vi和Vj之间有弧或边存在})\\ 0 & (\text{Vi和Vj之间没有弧或边存在})\end{cases}$$



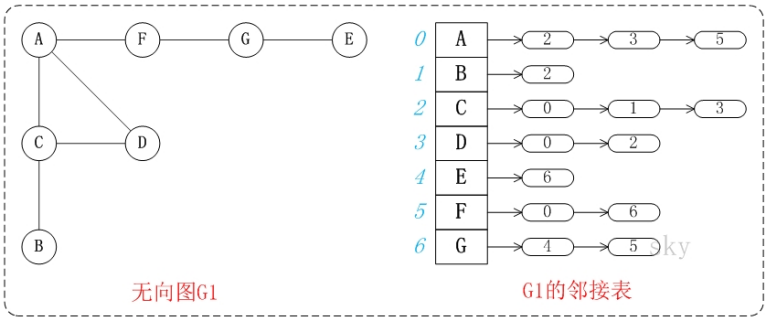
图中的G1是无向图和它对应的邻接矩阵。



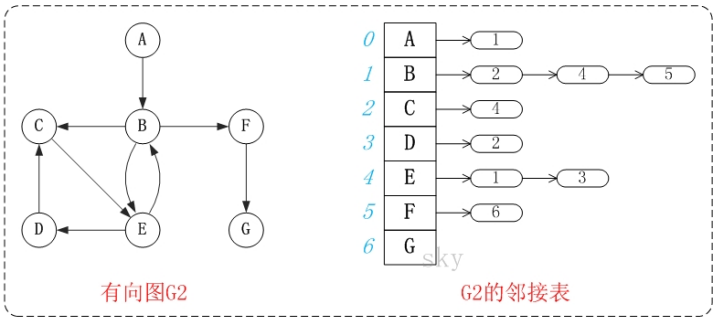
图中的G2是无向图和它对应的邻接矩阵。

通常采用两个数组来实现邻接矩阵：一个一维数组用来保存顶点信息，一个二维数组来用保存边的信息。 邻接矩阵的缺点就是比较耗费空间.

邻接表: 邻接表是图的一种链式存储表示方法。它是改进后的"邻接矩阵"，它的缺点是不方便判断两个顶点之间是否有边，但是相对邻接矩阵来说更省空间.



图中的G1是无向图和它对应的邻接矩阵。

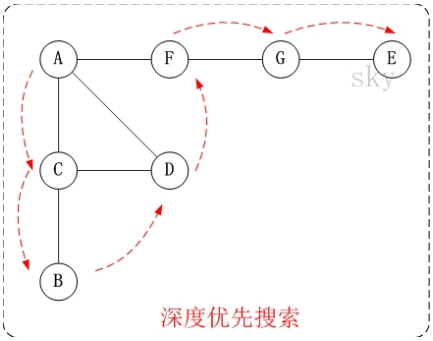


图中的G2是无向图和它对应的邻接矩阵。

2).深度优先遍历,广度优先遍历

深度优先遍历(DFS): 假设初始状态是图中所有顶点均未被访问，则从某个顶点 v 出发，首先访问该顶点，然后依次从它的各个未被访问的邻接点出发深度优先搜索遍历图，直至图中所有和 v 有路径相通的顶点都被访问到。 若此时尚有其他顶点未被访问到，则另选一个未被访问的顶点作起始点，重复上述过程，直至图中所有顶点都被访问到为止;

无向图的深度优先遍历:以上图中的五项图 G1 为例



第 1 步：访问 A。

第 2 步：访问(A 的邻接点)C。在第 1 步访问 A 之后，接下来应该访问的是 A 的邻接点，即"C,D,F"中的一个。但在本文的实现中，顶点 ABCDEFG 是按照顺序存储，C 在"D 和 F"的前面，因此，先访问 C。

第 3 步：访问(C 的邻接点)B。在第 2 步访问 C 之后，接下来应该访问 C 的邻接点，即"B 和 D"中一个(A 已经被访问过，就不算在内)。而由于 B 在 D 之前，先访问 B。

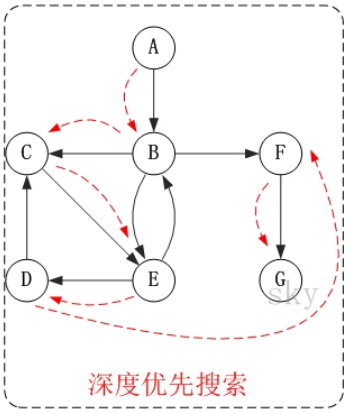
第 4 步：访问(C 的邻接点)D。在第 3 步访问了 C 的邻接点 B 之后，B 没有未被访问的邻接点；因此，返回到访问 C 的另一个邻接点 D。

第 5 步：访问(A 的邻接点)F。 前面已经访问了 A，并且访问完了"A 的邻接点 B 的所有邻接点(包括递归的邻接点在内)"；因此，此时返回到访问 A 的另一个邻接点 F。

第 6 步：访问(F 的邻接点)G。
第 7 步：访问(G 的邻接点)E。

因此访问顺序是：A -> C -> B -> D -> F -> G -> E

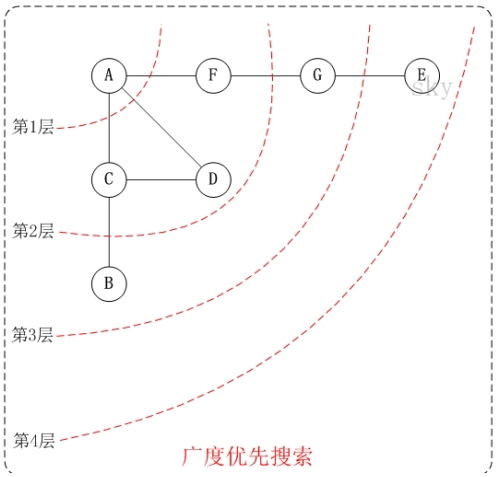
有向图的深度优先遍历:以上图中的有向图 G2 为例



第 1 步：访问 A。
第 2 步：访问 B。 在访问了 A 之后，接下来应该访问的是 A 的出边的另一个顶点，即顶点 B。
第 3 步：访问 C。 在访问了 B 之后，接下来应该访问的是 B 的出边的另一个顶点，即顶点 C,E,F。在本文实现的图中，顶点 ABCDEFG 按照顺序存储，因此先访问 C。
第 4 步：访问 E。 接下来访问 C 的出边的另一个顶点，即顶点 E。
第 5 步：访问 D。 接下来访问 E 的出边的另一个顶点，即顶点 B,D。顶点 B 已经被访问过，因此访问顶点 D。
第 6 步：访问 F。 接下应该回溯"访问 A 的出边的另一个顶点 F"。
第 7 步：访问 G。

因此访问顺序是：A -> B -> C -> E -> D -> F -> G

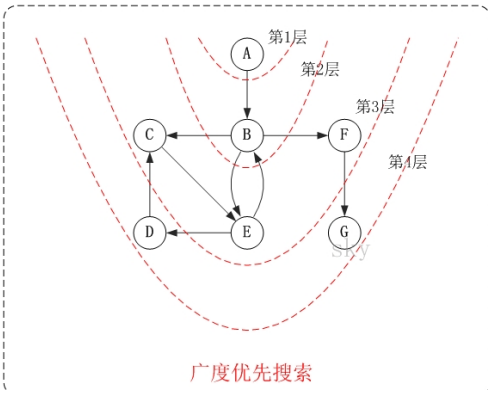
广度优先: 广度优先搜索算法(Breadth First Search)，又称为"宽度优先搜索"或"横向优先搜索"，简称 BFS；从图中某顶点 v 出发，在访问了 v 之后依次访问 v 的各个未曾访问过的邻接点，然后分别从这些邻接点出发依次访问它们的邻接点，并使得"先被访问的顶点的邻接点先于后被访问的顶点的邻接点被访问，直至图中所有已被访问的顶点的邻接点都被访问到。如果此时图中尚有顶点未被访问，则需要另选一个未曾被访问过的顶点作为新的起始点，重复上述过程，直至图中所有顶点都被访问到为止.
无向图的广度优先遍历:以上图 G1 为例



第 1 步：访问 A。
第 2 步：依次访问 C,D,F。 在访问了 A 之后，接下来访问 A 的邻接点。前面已经说过，在本文实现中，顶点 ABCDEFG 按照顺序存储的，C 在"D 和 F"的前面，因此，先访问 C。再访问完 C 之后，再依次访问 D,F。
第 3 步：依次访问 B,G。 在第 2 步访问完 C,D,F 之后，再依次访问它们的邻接点。首先访问 C 的邻接点 B，再访问 F 的邻接点 G。
第 4 步：访问 E。在第 3 步访问完 B,G 之后，再依次访问它们的邻接点。只有 G 有邻接点 E，因此访问 G 的邻接点 E。

因此访问顺序是：A -> C -> D -> F -> B -> G -> E

有向图的广度优先遍历:以上图中 G2 为例

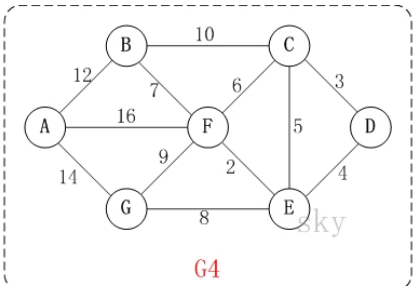


第 1 步：访问 A。
第 2 步：访问 B。
第 3 步：依次访问 C,E,F。 在访问了 B 之后，接下来访问 B 的出边的另一个顶点，即 C,E,F。前面已经说过，在本文实现中，顶点 ABCDEFG 按照顺序存储的，因此会先访问 C，再依次访问 E,F。
第 4 步：依次访问 D,G。在访问完 C,E,F 之后，再依次访问它们的出边的另一个顶点。还是按照 C,E,F 的顺序访问，C 的已经全部访问过了，那么就只剩下 E,F；先访问 E 的邻接点 D，再访问 F 的邻接点 G。

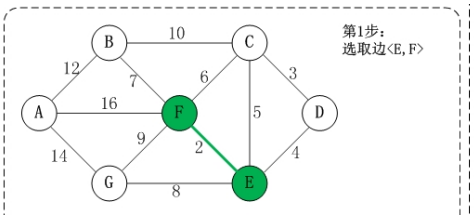
因此访问顺序是：A -> B -> C -> E -> F -> D -> G

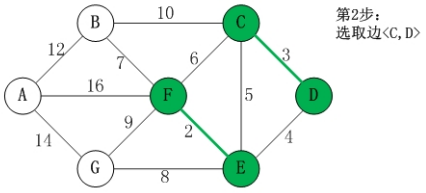
3).最小生成树,Kruskal 算法,Prim 算法

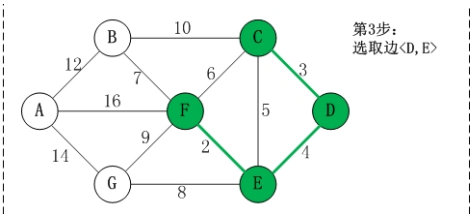
最小生成树：在含有 n 个顶点的连通图中选择 n-1 条边，构成一棵极小连通子图，并使该连通子图中 n-1 条边上权值之和达到最小，则称其为连通网的最小生成树.

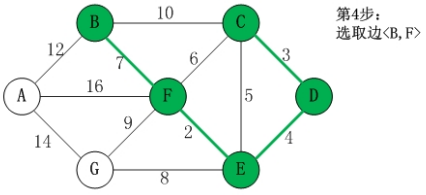


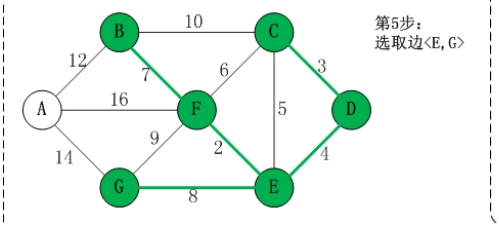
Kruskal 算法：按照权值从小到大的顺序选择 n-1 条边，并保证这 n-1 条边不构成回路；首先构造一个只含 n 个顶点的森林，然后依权值从小到大从连通网中选择边加入到森林中，并使森林中不产生回路，直至森林变成一棵树为止.

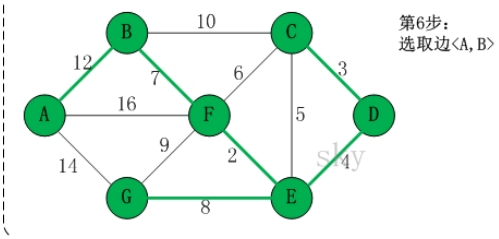












第 1 步：将边<E,F>加入 R 中。 边<E,F>的权值最小，因此将它加入到最小生成树结果 R 中。

第 2 步：将边<C,D>加入 R 中。 上一步操作之后，边<C,D>的权值最小，因此将它加入到最小生成树结果 R 中。

第 3 步：将边<D,E>加入 R 中。 上一步操作之后，边<D,E>的权值最小，因此将它加入到最小生成树结果 R 中。

第 4 步：将边<B,F>加入 R 中。 上一步操作之后，边<C,E>的权值最小，但<C,E>会和已有的边构成回路；因此，跳过边<C,E>。同理，跳过边<C,F>。将边<B,F>加入到最小生成树结果 R 中。

第 5 步：将边<E,G>加入 R 中。上一步操作之后，边<E,G>的权值最小，因此将它加入到最小生成树结果 R 中。

第 6 步：将边<A,B>加入 R 中。上一步操作之后，边<F,G>的权值最小，但<F,G>会和

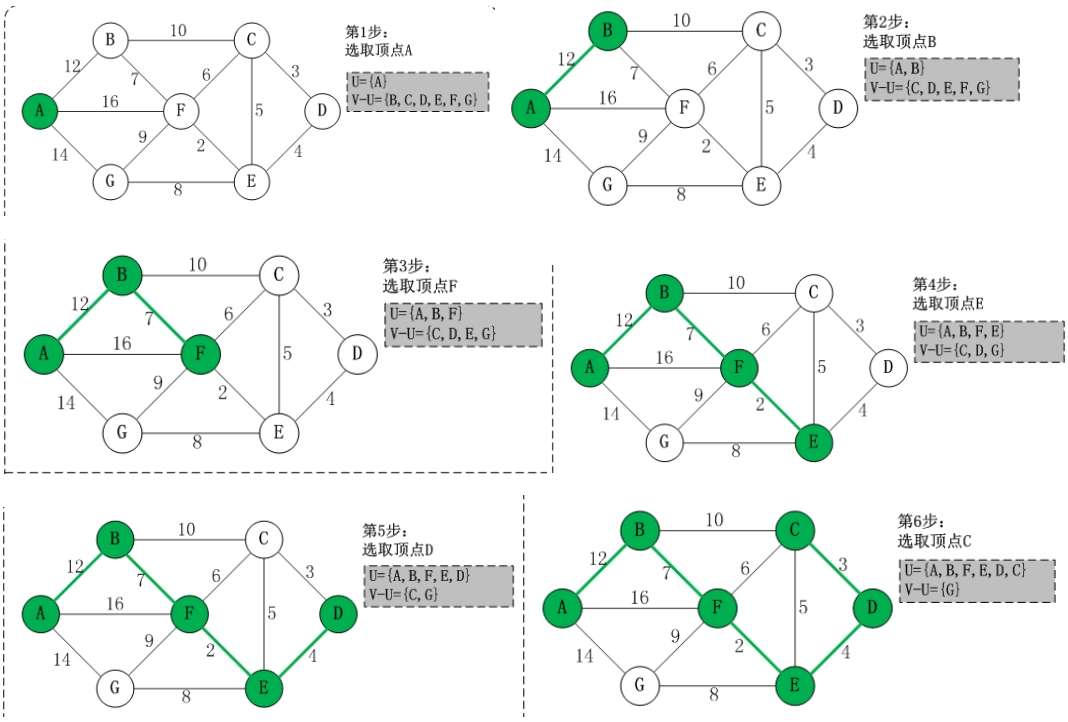
已有的边构成回路；因此，跳过边<F,G>。同理，跳过边<B,C>。将边<A,B>加入到最小生成树结果 R 中。

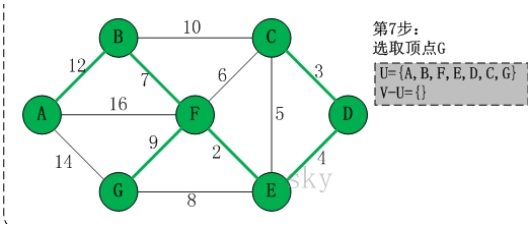
此时，最小生成树构造完成！它包括的边依次是：<E,F> <C,D> <D,E> <B,F> <E,G> <A,B>。

```
// 边的结构体
private static class EData {
    char start; // 边的起点
    char end;   // 边的终点
    int weight; // 边的权重
    public EData(char start, char end, int weight) {
        this.start = start;
        this.end = end;
        this.weight = weight;
    }
}
public class MatrixUDG {//邻接矩阵
    private int mEdgNum;        // 边的数量
    private char[] mVexs;       // 顶点集合
    private int[][] mMatrix;    // 邻接矩阵
    private static final int INF = Integer.MAX_VALUE;   // 最大值
}
/*
 * 克鲁斯卡尔 (Kruskal)最小生成树
 */
public void kruskal() {
    int index = 0;                      // rets 数组的索引
    int[] vends = new int[mEdgNum];     // 用于保存"已有最小生成树"中每个顶点在该最小树中的终点。
    EData[] rets = new EData[mEdgNum];  // 结果数组，保存 kruskal 最小生成树的边
    EData[] edges;                      // 图对应的所有边
    // 获取"图中所有的边"
    edges = getEdges();
    // 将边按照"权"的大小进行排序(从小到大)
    sortEdges(edges, mEdgNum);
    for (int i=0; i<mEdgNum; i++) {
        int p1 = getPosition(edges[i].start);      // 获取第 i 条边的"起点"的序号
        int p2 = getPosition(edges[i].end);        // 获取第 i 条边的"终点"的序号

        int m = getEnd(vends, p1);                 // 获取 p1 在"已有的最小生成树"中的终点
```

```
        int n = getEnd(vends, p2);                    // 获取 p2 在"已有的最小生成
树"中的终点
        // 如果 m!=n，意味着"边 i"与"已经添加到最小生成树中的顶点"没有形成环路
        if (m != n) {
            vends[m] = n;                             // 设置 m 在"已有的最小生成树"中
的终点为 n
            rets[index++] = edges[i];                 // 保存结果
        }
    }
    // 统计并打印"kruskal 最小生成树"的信息
    int length = 0;
    for (int i = 0; i < index; i++)
        length += rets[i].weight;
    System.out.printf("Kruskal=%d: ", length);
    for (int i = 0; i < index; i++)
        System.out.printf("(%c,%c) ", rets[i].start, rets[i].end);
    System.out.printf("\n");
}
```

Prim 算法: 对于图 G 而言，V 是所有顶点的集合；现在，设置两个新的集合 U 和 T，其中 U 用于存放 G 的最小生成树中的顶点，T 存放 G 的最小生成树中的边。 从所有 u∈U，v∈(V-U) (V-U 表示出去 U 的所有顶点)的边中选取权值最小的边(u, v)，将顶点 v 加入集合 U 中，将边(u, v)加入集合 T 中，如此不断重复，直到 U=V 为止，最小生成树构造完毕，这时集合 T 中包含了最小生成树中的所有边.



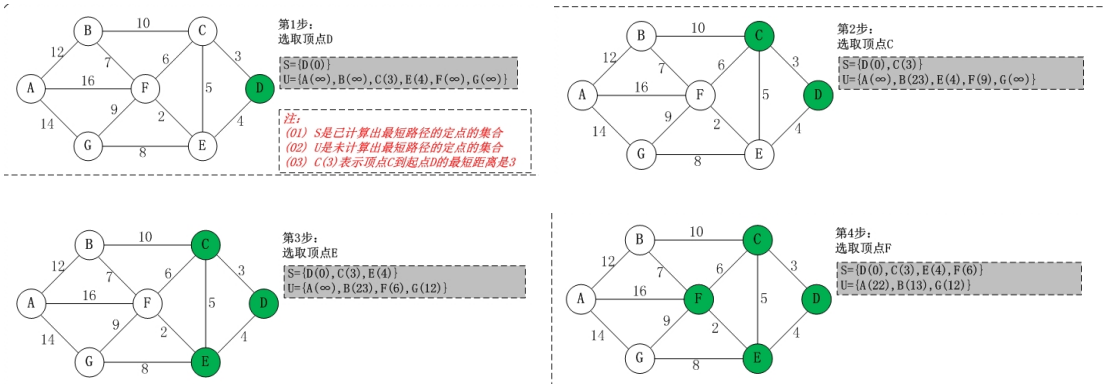


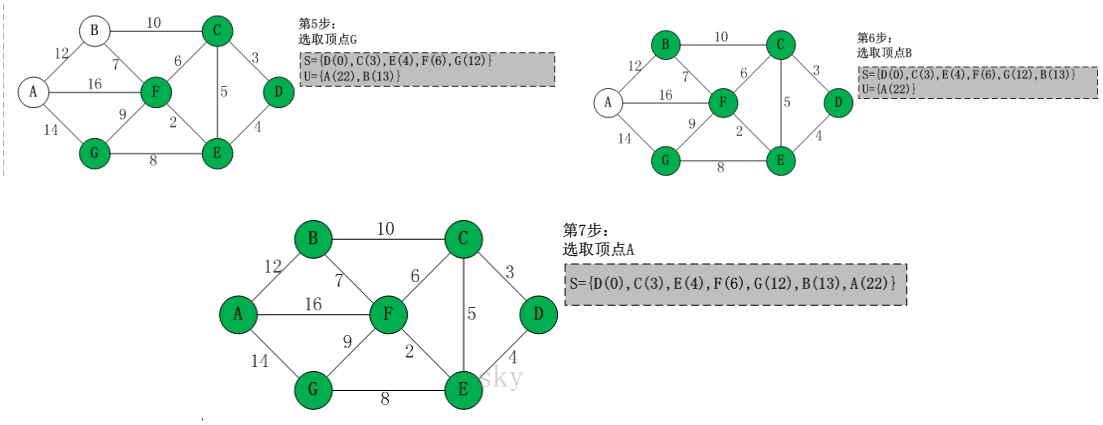
初始状态：V 是所有顶点的集合，即 V={A,B,C,D,E,F,G}；U 和 T 都是空！
第 1 步：将顶点 A 加入到 U 中。 此时，U={A}。
第 2 步：将顶点 B 加入到 U 中。 上一步操作之后，U={A}, V-U={B,C,D,E,F,G}；因此，边(A,B)的权值最小。将顶点 B 添加到 U 中；此时，U={A,B}。
第 3 步：将顶点 F 加入到 U 中。 上一步操作之后，U={A,B}, V-U={C,D,E,F,G}；因此，边(B,F)的权值最小。将顶点 F 添加到 U 中；此时，U={A,B,F}。
第 4 步：将顶点 E 加入到 U 中。 上一步操作之后，U={A,B,F}, V-U={C,D,E,G}；因此，边(F,E)的权值最小。将顶点 E 添加到 U 中；此时，U={A,B,F,E}。
第 5 步：将顶点 D 加入到 U 中。 上一步操作之后，U={A,B,F,E}, V-U={C,D,G}；因此，边(E,D)的权值最小。将顶点 D 添加到 U 中；此时，U={A,B,F,E,D}。
第 6 步：将顶点 C 加入到 U 中。 上一步操作之后，U={A,B,F,E,D}, V-U={C,G}；因此，边(D,C)的权值最小。将顶点 C 添加到 U 中；此时，U={A,B,F,E,D,C}。
第 7 步：将顶点 G 加入到 U 中。 上一步操作之后，U={A,B,F,E,D,C}, V-U={G}；因此，边(F,G)的权值最小。将顶点 G 添加到 U 中；此时，U=V。

此时，最小生成树构造完成！它包括的顶点依次是：A B F E D C G

4).单源最短路径

迪杰斯特拉(Dijkstra)算法：用于计算一个节点到其他节点的最短路径，主要特点是以起始点为中心向外层层扩展(广度优先搜索思想)，直到扩展到终点为止，初始时，S 中只有起点 s；U 中是除 s 之外的顶点，并且 U 中顶点的路径是"起点 s 到该顶点的路径"。然后，从 U 中找出路径最短的顶点，并将其加入到 S 中；接着，更新 U 中的顶点和顶点对应的路径。然后，再从 U 中找出路径最短的顶点，并将其加入到 S 中；接着，更新 U 中的顶点和顶点对应的路径。 ... 重复该操作，直到遍历完所有顶点.
下图例子以 D 为起点：





初始状态：S 是已计算出最短路径的顶点集合，U 是未计算除最短路径的顶点的集合！
第 1 步：将顶点 D 加入到 S 中。 此时，S={D(0)}, U={A(∞),B(∞),C(3),E(4),F(∞),G(∞)}。　　注:C(3)表示 C 到起点 D 的距离是 3。

第 2 步：将顶点 C 加入到 S 中。 上一步操作之后，U 中顶点 C 到起点 D 的距离最短；因此，将 C 加入到 S 中，同时更新 U 中顶点的距离。以顶点 F 为例，之前 F 到 D 的距离为∞；但是将 C 加入到 S 之后，F 到 D 的距离为 9=(F,C)+(C,D)。 此时，S={D(0),C(3)}, U={A(∞),B(23),E(4),F(9),G(∞)}。

第 3 步：将顶点 E 加入到 S 中。 上一步操作之后，U 中顶点 E 到起点 D 的距离最短；因此，将 E 加入到 S 中，同时更新 U 中顶点的距离。还是以顶点 F 为例，之前 F 到 D 的距离为 9；但是将 E 加入到 S 之后，F 到 D 的距离为 6=(F,E)+(E,D)。此时，S={D(0),C(3),E(4)}, U={A(∞),B(23),F(6),G(12)}。

第 4 步：将顶点 F 加入到 S 中。 此时，S={D(0),C(3),E(4),F(6)}, U={A(22),B(13),G(12)}。

第 5 步：将顶点 G 加入到 S 中。 此时，S={D(0),C(3),E(4),F(6),G(12)}, U={A(22),B(13)}。

第 6 步：将顶点 B 加入到 S 中。 此时，S={D(0),C(3),E(4),F(6),G(12),B(13)}, U={A(22)}。

第 7 步：将顶点 A 加入到 S 中。 此时，S={D(0),C(3),E(4),F(6),G(12),B(13),A(22)}。

此时，起点 D 到各个顶点的最短距离就计算出来了：A(22) B(13) C(3) D(0) E(4) F(6) G(12).

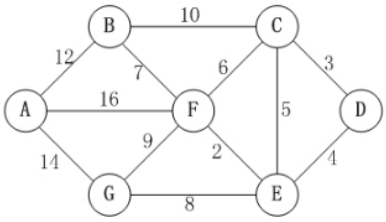
```
/*
 * Dijkstra 最短路径。
 * 即，统计图中"顶点 vs"到其它各个顶点的最短路径。
 *
 * 参数说明：
 *       vs -- 起始顶点(start vertex)。即计算"顶点 vs"到其它顶点的最短路径。
 *     prev -- 前驱顶点数组。即，prev[i]的值是"顶点 vs"到"顶点 i"的最短路径所经历的全部顶点中，位于"顶点 i"之前的那个顶点。
```

```
 *     dist -- 长度数组。即，dist[i]是"顶点 vs"到"顶点 i"的最短路径的长度。
 */
public void dijkstra(int vs, int[] prev, int[] dist) {
    // flag[i]=true 表示"顶点 vs"到"顶点 i"的最短路径已成功获取
    boolean[] flag = new boolean[mVexs.length];
    // 初始化
    for (int i = 0; i < mVexs.length; i++) {
        flag[i] = false;          // 顶点 i 的最短路径还没获取到。
        prev[i] = 0;              // 顶点 i 的前驱顶点为 0。
        dist[i] = mMatrix[vs][i];  // 顶点 i 的最短路径为"顶点 vs"到"顶点 i"的权。
    }
    // 对"顶点 vs"自身进行初始化
    flag[vs] = true;
    dist[vs] = 0;
    // 遍历 mVexs.length-1 次；每次找出一个顶点的最短路径。
    int k=0;
    for (int i = 1; i < mVexs.length; i++) {
        // 寻找当前最小的路径；
        // 即，在未获取最短路径的顶点中，找到离 vs 最近的顶点(k)。
        int min = INF;
        for (int j = 0; j < mVexs.length; j++) {
            if (flag[j]==false && dist[j]<min) {
                min = dist[j];
                k = j;
            }
        }
        // 标记"顶点 k"为已经获取到最短路径
        flag[k] = true;
        // 修正当前最短路径和前驱顶点
        // 即，当已经"顶点 k 的最短路径"之后，更新"未获取最短路径的顶点的最短路径和
前驱顶点"。
        for (int j = 0; j < mVexs.length; j++) {
            int tmp = (mMatrix[k][j]==INF ? INF : (min + mMatrix[k][j]));
            if (flag[j]==false && (tmp<dist[j]) ) {
                dist[j] = tmp;
                prev[j] = k;
            }
        }
    }
    // 打印 dijkstra 最短路径的结果
    System.out.printf("dijkstra(%c): \n", mVexs[vs]);
    for (int i=0; i < mVexs.length; i++)
        System.out.printf("  shortest(%c, %c)=%d\n", mVexs[vs], mVexs[i], dist[i]);
}
```

5).所有节点对的最短路径

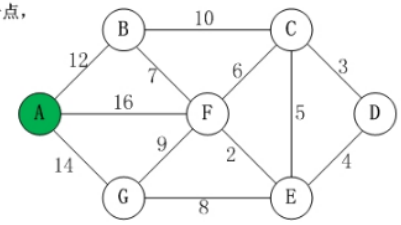
弗洛伊德(Floyd)算法：通过 Floyd 计算图 G=(V,E)中各个顶点的最短路径时，需要引入一个矩阵 S，矩阵 S 中的元素 a[i][j]表示顶点 i(第 i 个顶点)到顶点 j(第 j 个顶点)的距离；假设图 G 中顶点个数为 N，则需要对矩阵 S 进行 N 次更新。初始时，矩阵 S 中顶点 a[i][j]的距离为顶点 i 到顶点 j 的权值；如果 i 和 j 不相邻，则 a[i][j]=∞。 接下来开始，对矩阵 S 进行 N 次更新。第 1 次更新时，如果"a[i][j]的距离" > "a[i][0]+a[0][j]"(a[i][0]+a[0][j]表示"i 与 j 之间经过第 1 个顶点的距离")，则更新 a[i][j]为"a[i][0]+a[0][j]"。 同理，第 k 次更新时，如果"a[i][j]的距离" > "a[i][k]+a[k][j]"，则更新 a[i][j]为"a[i][k]+a[k][j]"。更新 N 次之后，操作完成.

第1步：
初始化矩阵S



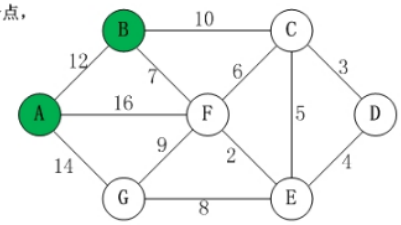
	A	B	C	D	E	F	G
A	0	12	INF	INF	INF	16	14
B	12	0	10	INF	INF	7	INF
C	INF	10	0	3	5	6	INF
D	INF	INF	3	0	4	INF	INF
E	INF	INF	5	4	0	2	8
F	16	7	6	INF	2	0	9
G	14	INF	INF	INF	8	9	0

第2步：
以顶点A为中介点，
更新矩阵S。



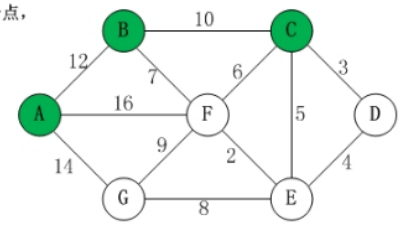
	A	B	C	D	E	F	G
A	0	12	INF	INF	INF	16	14
B	12	0	10	INF	INF	7	26
C	INF	10	0	3	5	6	INF
D	INF	INF	3	0	4	INF	INF
E	INF	INF	5	4	0	2	8
F	16	7	6	INF	2	0	9
G	14	26	INF	INF	8	9	0

第3步：
以顶点B为中介点，
更新矩阵S。



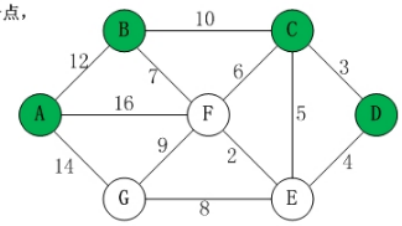
	A	B	C	D	E	F	G
A	0	12	22	INF	INF	16	14
B	12	0	10	INF	INF	7	26
C	22	10	0	3	5	6	36
D	INF	INF	3	0	4	INF	INF
E	INF	INF	5	4	0	2	8
F	16	7	6	INF	2	0	9
G	14	26	36	INF	8	9	0

第4步：
以顶点C为中介点，
更新矩阵S。



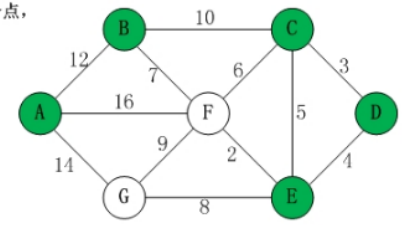
	A	B	C	D	E	F	G
A	0	12	22	25	27	16	14
B	12	0	10	13	15	7	26
C	22	10	0	3	5	6	36
D	25	13	3	0	4	9	39
E	27	15	5	4	0	2	8
F	16	7	6	9	2	0	9
G	14	26	36	39	8	9	0

第5步：
以顶点D为中介点，
更新矩阵S。



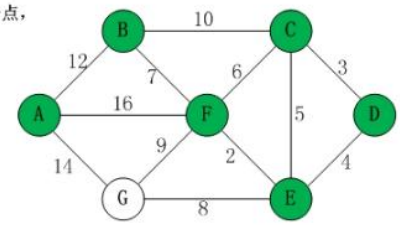
	A	B	C	D	E	F	G
A	0	12	22	25	27	16	14
B	12	0	10	13	15	7	26
C	22	10	0	3	5	6	36
D	25	13	3	0	4	9	39
E	27	15	5	4	0	2	8
F	16	7	6	9	2	0	9
G	14	26	36	39	8	9	0

第6步：
以顶点E为中介点，
更新矩阵S。



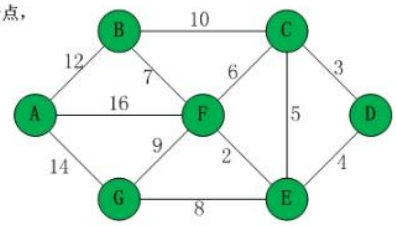
	A	B	C	D	E	F	G
A	0	12	22	25	27	16	14
B	12	0	10	13	15	7	23
C	22	10	0	3	5	6	13
D	25	13	3	0	4	6	12
E	27	15	5	4	0	2	8
F	16	7	6	6	2	0	9
G	14	23	13	12	8	9	0

第7步：
以顶点F为中介点，
更新矩阵S。



	A	B	C	D	E	F	G
A	0	12	22	22	18	16	14
B	12	0	10	13	9	7	16
C	22	10	0	3	5	6	13
D	22	13	3	0	4	6	12
E	18	9	5	4	0	2	8
F	16	7	6	6	2	0	9
G	14	16	13	12	8	9	0

第8步：
以顶点G为中介点，
更新矩阵S。



	A	B	C	D	E	F	G
A	0	12	22	22	18	16	14
B	12	0	10	13	9	7	16
C	22	10	0	3	5	6	13
D	22	13	3	0	4	6	12
E	18	9	5	4	0	2	8
F	16	7	6	6	2	0	9
G	14	16	13	12	8	9	0

初始状态：S 是记录各个顶点间最短路径的矩阵。
第 1 步：初始化 S。 矩阵 S 中顶点 a[i][j]的距离为顶点 i 到顶点 j 的权值；如果 i 和 j 不相邻，则 a[i][j]=∞。实际上，就是将图的原始矩阵复制到 S 中。 注:a[i][j]表示矩阵 S 中顶点 i(第 i 个顶点)到顶点 j(第 j 个顶点)的距离。

第 2 步：以顶点 A(第 1 个顶点)为中介点，若 a[i][j] > a[i][0]+a[0][j]，则设置 a[i][j]=a[i][0]+a[0][j]。 以顶点 a[1]6，上一步操作之后，a[1][6]=∞；而将 A 作为中介点时，(B,A)=12，(A,G)=14，因此 B 和 G 之间的距离可以更新为 26。

同理，依次将顶点 B,C,D,E,F,G 作为中介点，并更新 a[i][j]的大小。

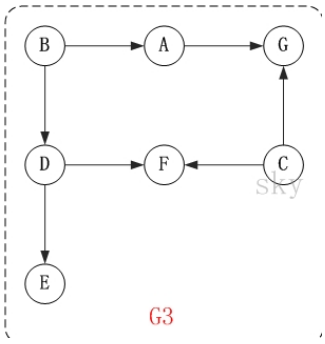
```
/*
 * floyd 最短路径。
 * 即，统计图中各个顶点间的最短路径。
 *
```

```
 * 参数说明:
 *     path -- 路径。path[i][j]=k 表示，"顶点 i"到"顶点 j"的最短路径会经过顶点 k。
 *     dist -- 长度数组。即，dist[i][j]=sum 表示，"顶点 i"到"顶点 j"的最短路径的长度是 sum。
 */
public void floyd(int[][] path, int[][] dist) {

    // 初始化
    for (int i = 0; i < mVexs.length; i++) {
        for (int j = 0; j < mVexs.length; j++) {
            dist[i][j] = mMatrix[i][j];    // "顶点 i"到"顶点 j"的路径长度为"i 到 j
的权值"。
            path[i][j] = j;                // "顶点 i"到"顶点 j"的最短路径是经过顶点
j。
        }
    }
    // 计算最短路径
    for (int k = 0; k < mVexs.length; k++) {
        for (int i = 0; i < mVexs.length; i++) {
            for (int j = 0; j < mVexs.length; j++) {
                // 如果经过下标为 k 顶点路径比原两点间路径更短，则更新 dist[i][j]和
path[i][j]
                int tmp = (dist[i][k]==INF || dist[k][j]==INF) ? INF : (dist[i][k]
+ dist[k][j]);
                if (dist[i][j] > tmp) {
                    // "i 到 j 最短路径"对应的值设，为更小的一个(即经过 k)
                    dist[i][j] = tmp;
                    // "i 到 j 最短路径"对应的路径，经过 k
                    path[i][j] = path[i][k];
                }
            }
        }
    }
    // 打印 floyd 最短路径的结果
    System.out.printf("floyd: \n");
    for (int i = 0; i < mVexs.length; i++) {
        for (int j = 0; j < mVexs.length; j++)
            System.out.printf("%2d  ", dist[i][j]);
        System.out.printf("\n");
    }
}
```

6).拓扑排序

拓扑排序(Topological Order)是指，将一个有向无环图(Directed Acyclic Graph 简称 DAG)进行排序进而得到一个有序的线性序列;

拓扑排序的应用：一个项目包括 A、B、C、D 四个子部分来完成，并且 A 依赖于 B 和 D，C 依赖于 D。现在要制定一个计划，写出 A、B、C、D 的执行顺序。这时，就可以利用到拓扑排序，它就是用来确定事物发生的顺序的;
对下图进行拓扑排序:



第 1 步：将 B 和 C 加入到排序结果中(B,C 没有依赖定点,即不存在以 B,C 为终点的边)。 顶点 B 和顶点 C 都是没有依赖顶点，因此将 B 和 C 加入到结果集 T 中。假设 ABCDEFG 按顺序存储，因此先访问 B，再访问 C。访问 B 之后，去掉边<B,A>和<B,D>，并将 A 和 D 加入到队列 Q 中。同样的，去掉边<C,F>和<C,G>，并将 F 和 G 加入到 Q 中。

(01) 将 B 加入到排序结果中，然后去掉边<B,A>和<B,D>；此时，由于 A 和 D 没有依赖顶点，因此并将 A 和 D 加入到队列 Q 中。

(02) 将 C 加入到排序结果中，然后去掉边<C,F>和<C,G>；此时，由于 F 有依赖顶点 D，G 有依赖顶点 A，因此不对 F 和 G 进行处理。

第 2 步：将 A,D 依次加入到排序结果中。第 1 步访问之后，A,D 都是没有依赖顶点的，根据存储顺序，先访问 A，然后访问 D。访问之后，删除顶点 A 和顶点 D 的出边。

第 3 步：将 E,F,G 依次加入到排序结果中。

因此访问顺序是：B -> C -> A -> D -> E -> F -> G

7.动态规划,贪心算法

1).剪绳子: 给定一根长度为 n 的绳子,请把绳子剪成 m 段(m,n 都是正整数, n>1, m>1),每段绳子的长度记为 k[0],k[1],...,k[m],求 k[0]xk[1]x...xk[m]的最大值

2).连续子数组的最大和: 向量中包含负数,数组中的一个或连续多个整数组成一个子数组,求所有子数组的和的最大值, 例如:{6,-3,-2,7,-15,1,2,2},连续子向量的最大和为 8(从第 0 个开始,到第 3 个为止)。你会不会被他忽悠住？(子向量的长度至少是 1).

3).礼物的最大价值: 在一个 m * n 的棋盘的每一个格都放有一个礼物，每个礼物都有一定价值（大于 0）。从左上角开始拿礼物，每次向右或向下移动一格，直到右下角结束。给定一个棋盘，求拿到礼物的最大价值

4). 最长不含重复字符的子字符串: 一个字符串(字符只包含'a'~'z')，求这个字符串中不包含重复字符的最长子串的长度，如 abba 返回 2，aaaaabc 返回 3，bbbbbbb 返回 1，等等上面是测试用例.

5).求数组中数对之差的最大值: 例如输入{1,4,17,3,2,9},则最大差值为 17-2=15.

6).长度为 A 的东西有 X 个,长度为 B 的东西有 Y 个,然后要组成长度为 K 的东西,不考虑顺序,求一共有多少种组合方式.

8.位运算

1).二进制数中 1 的个数: 输入一个整数，输出该数二进制表示中 1 的个数。其中负数用补码表示。

2).数组中数字出现一次的两个数字:一个整型数组中除了两个数字之外,其他数字都出现了两次,找出这两个只出现一次的数字,要求时间复杂度是 O(n),空间复杂度 O(1).

3). 数组中唯一出现一次的两个数字:一个整型数组中除了一个数字之外,其他数字都出现了三次,找出这个只出现一次的数字,要求时间复杂度是 O(n),空间复杂度 O(1).

9.字符串

1).替换空格: 请实现一个函数，将一个字符串中的空格替换成“%20”.例如，当字符串为 We Are Happy.则经过替换之后的字符串为 We%20Are%20Happy.

2).打印出从 1 到 n 的最大的 n 位数,例如输入 3,则打印出 1,2,3 一直到最大的 3 位数 999(大数问题)

3).正则表达式的匹配: 请实现一个函数用来匹配包括'.'和'*'的正则表达式。模式中的字符'.'表示任意一个字符，而'*'表示它前面的字符可以出现任意次（包含 0 次）。在本题中，匹配是指字符串的所有字符匹配整个模式。例如，字符串"aaa"与模式"a.a"和"ab*ac*a"匹配，但是与"aa.a"和"ab*a"均不匹配

4).表示数值的字符串: 请实现一个函数用来判断字符串是否表示数值（包括整数和小数）。例如，字符串"+100","5e2","-123","3.1416"和"-1E-16"都表示数值。 但是"12e","1a3.14","1.2.3","+-5"和"12e+4.3"都不是。

5).字符串的排列: 输入一个字符串,按字典序打印出该字符串中字符的所有排列。例如输入字符串 abc,则打印出由字符 a,b,c 所能排列出来的所有字符串 abc,acb,bac,bca,cab 和 cba。

6).把数字翻译成字符串: 给定一个数字，按照如下规则把它转化成字符串： 0 翻译成 a,1 翻译成 b,2 翻译成 c ，…… ，25 翻译成 z 。 一个数可能有多种翻译，比如数字 11 可以翻译成 bb 也可以翻译成 l ；例如数字 12258 有 5 中不同的翻译 ： bccfi 、bwfi 、bczi、mcfi 、mzi 这五种。我们现在输入 一个数字，计算这个数字一共有多少不同的翻译方法.

7).第一个只出现一次的字符: 在一个字符串(1<=字符串长度<=10000，全部由大写字母组成)中找到第一个只出现一次的字符,并返回它的位置。

8).翻转字符串:输入一个英文句子,翻转句子中的单词的顺序,但单词内的字符的顺序不变,例如:输入”I am a student.”, 则输出”student. a am I”.

9).左旋转字符串:输入”abcdefg”和数字 2, 则返回左旋转两位得到的结果为”cdefgab”.

10).字符串的包含:给定一个字符串 a 和一短字符串 b,如何判断出段字符串 b 中的所有字符是都在长字符串 a 中.

11).字符串转换为整数.

12).回文判断, 回文指正读和反读都是一样的.

13).最长回文子串，给定一个字符串,求最长的回文子串的长度

10.海量数据的处理

1).hash 法

2).Bit-map 法

3).Bloom Filter

4).数据库优化法

5).倒排索引

6).外排序

7).Trie 树

11.其他算法代码

1).数组中重复的数字: 在长度为 n 的数组中，所有的元素都是 0 到 n-1 的范围内。 数组中的某些数字是重复的，但不知道有几个重复的数字，也不知道重复了几次，请找出任意重复的数字。 例如，输入长度为 7 的数组{2,3,1,0,2,5,3}，那么对应的输出为 2 或 3.

2).二维数组的查找: 在一个二维数组中，每一行都按照从左到右递增的顺序排序，每一列都按照从上到下递增的顺序排序。请完成一个函数，输入这样的一个二维数组和一个整数，判断数组中是否含有该整数.

3).替换空格：请实现一个函数，将一个字符串中的空格替换成“%20”.例如，当字符串为 We Are Happy.则经过替换之后的字符串为 We%20Are%20Happy.

4).斐波拉契数列的递归实现以及优化

5).旋转数组的最小值

6).矩阵中的路径

7).机器人的运动范围

8).数值的整数次方

9).调整数组顺序使得奇数位于偶数前面,调整后前半部分是奇数,后半部分是偶数

10).顺时针打印矩阵:输入一个矩阵,按照从外到里以顺时针依次打印出每一个数字.

11).数组中次数超过一半的数: 数组中有一个数字出现的次数超过数组长度的一半，请找出这个数字。例如输入一个长度为 9 的数组{1,2,3,2,2,2,5,4,2}。由于数字 2 在数组中出现了 5 次，超过数组长度的一半，因此输出 2。如果不存在则输出 0。

12).最小的 K 个数: 输入 n 个整数，找出其中最小的 K 个数。例如输入 4,5,1,6,2,7,3,8 这 8 个数字，则最小的 4 个数字是 1,2,3,4,。

13).数据流中的中位数: 如何得到一个数据流中的中位数？如果从数据流中读出奇数个数值，那么中位数就是所有数值排序之后位于中间的数值。如果从数据流中读出偶数个数值，那么中位数就是所有数值排序之后中间两个数的平均值。

14).1~n 整数中 1 出现的次数: 求出 1~13 的整数中 1 出现的次数,并算出 100~1300 的整数中 1 出现的次数？为此他特别数了一下 1~13 中包含 1 的数字有 1、10、11、12、13 因此共出现 6 次,但是对于后面问题他就没辙了。ACMer 希望你们帮帮他,并把问题更加普遍化,可以很快的求出任意非负整数区间中 1 出现的次数。

15).数字序列中某一位的数字:数字以0123456789101112131415l6...的格式序列化到一个字符序列中,在这个序列中,第 5 位(从 0 开始)是 5,第 13 位是 1,第 19 位是 4,等等,求任意第 n 位对应的数字.

16).丑数: 把只包含因子 2、3 和 5 的数称作丑数（Ugly Number）。例如 6、8 都是丑数，但 14 不是，因为它包含因子 7。 习惯上我们把 1 当做是第一个丑数。求按从小到大的顺序的第 N 个丑数.

17).数组中的逆序对: 在数组中的两个数字，如果前面一个数字大于后面的数字，则这两个数字组成一个逆序对。输入一个数组,求出这个数组中的逆序对的总数 P。并将 P 对 1000000007 取模的结果输出。 即输出 P%1000000007.

18).在排序数组中查找数字: 统计一个数字在排序数组中出现的次数。

19).0~n-1 中缺失的数字:一个长度为 n-1 的递增排序数组中的所有数字都是唯一的,并且每一个数字都在范围 0~n-1 之内,在 0~n-1 内的 n 个数只有一个不在该数组中,找出这个数字.

20).数组中的数值和下标相等的元素:递增数组中每个整数(可正可负)都是唯一的,找出数组中任意一个数值等于下标的元素,例如在数组{-3,-1,1,3,5}中,数字 3 和它的下标相等.

21).和为 s 的两个数字:输入一个递增数组和一个数字 s,在数组中找到两个数,使得他们的和为 s,如果有多对则输出其中一对即可.

22).和为 s 的连续整数序列: 输入一个正数 s,输出所有和为 S 的连续正数序列。序列内按照从小至大的顺序，序列间按照开始数字从小到大的顺序，例如: 输入 15, 则 1+2+3+4+5=4+5+6=7+8=15, 所有打印出连续序列 1+2+3+4+5, 4+5+6,7+8.

23).扑克牌中的顺子:从扑克牌中随机抽 5 张,判断是不是一个顺子,即 5 张是不是连续的,2~10 位本身,A 为 1,J 为 11, Q 为 12,K 为 13,大小王可以看成任意数字.

24).圆圈中最后剩余的数字:0,1,2,…,n-1 这 n 个数字构成一个圆圈,充 0 开始,每次从这个圆圈里删除第 m 个数字,求出这个圆圈里剩下的最后一个数字.

方法 1: 循环链表实现:

```
//游戏类
public class Game {
    //单向循环链表
    CycleLinkList list = new CycleLinkList();
    //总人数
```

```
int num;
//数到几退出
int key;
//游戏初始化方法
public Game(int num,int key)
{
    this.num = num;
    this.key = key;
}
public void play() throws Exception
{
    for(int i=0;i<num;i++)
    {
        list.insert(i, i);
    }
    System.out.println("\n-------游戏开始之前---------\n");
    for(int i=0;i<list.size;i++)
    {
        System.out.print(list.get(i)+" ");
    }
    System.out.println("\n-------游戏开始---------\n");
    int iCount=num; //开始等于总人数 num
    int j=0; //累加器，计算是否能被 key 整除。
    Node node = list.head;
    while(iCount!=1)
    {
        if(node.getElement()!=null&& Integer.parseInt(node.getElement().toString())!=-1)
        {
            j++;
            if(j%key==0)
            {
                node.setElement(-1);
                iCount--;
                System.out.println();
                for(int i=0;i<list.size;i++)
                {
                    System.out.print(list.get(i)+" ");
                }
            }
        }
        node = node.next;
    }
    System.out.println("\n-------游戏结束---------\n");
    for(int i=0;i<list.size;i++)
```

```
            {
                System.out.print(list.get(i)+" ");
            }
        }
}
```

25).股票的最大利润：一个数组中，只能用后面的数字减去前面的数字，求出差值的最大值.

26).不用加减乘除做加法

27).构建乘积数组：给定一个数组 A[0,1,…,n-1],请构建一个数组 B[0,1,…,n-1],其中 B 中的元素 B[i]=A[0]A[1]…A[i-1]*A[i+1]…*A[n-1]。不能使用除法.

28).求数组中第二大的数,不能用排序算法,时间复杂度为 O(n)

29).找出数组中重复元素最多的数

30).数组循环右移 k 位,例如:12345678 循环右移两位后为 78123456

31).找出数组中第 k 个最小的数,不能用排序算法

32).递归求数组中最大的元素

33).求绝对值最小的数,输入{-10,-5,-2,15,50},则绝对值最小的为-2.

34).求数组中两个元素的最小距离，例如输入数组{4,5,6,4,7,4,6,4,7,8,5,6,4,3,10,8},输入 n1=4,n2=8,则输出为最小距离为 2.

35).求指定数字在数组中第一次出现的位置:给定数组 a={3,4,5,6,5,6,7,8,9,8},这个数组相邻元素之差都为 1,给定数字 9,它在数组中第一次出现的位置下标为 8.

36).对数组的两个子有序段进行合并,要求空间复杂度为 O(1), 数组 a[0,mid-1]和 amid,n-1[各自有序,对数组 a[0,n-1]合并

37).计算两个有序整数数组的交集

38).求最小三元组的距离: 有 3 个升序数组,在 3 个数组中个找一个元素使得三个元素之间的距离最小,假设三个元素为 a[i], b[j], c[k], 则 distance=max(|a[i]-b[j]|,|a[i]-c[k]|,|b[j]-c[k]|), 求所有 distance 的最小值.

39).按要求打印数组的排列情况:对 1,2,2,3,4,5 这 6 个数,打印出所有不同的排列,要求 4 不能再第三位,3 和 5 不能相连.