数据结构笔记

队列

Josephus（约瑟夫）问题

```
void Josephus(List head,int n,int m){
  // n个人，数m
  Node p,q;
  p = head;
  int c;//计数器
  int k;//剩余人数
  while(k>1){
        if(c==m-1){//这一圈数到第m个人了
            //删去p的下一节点q
            q = p.link;
            p.link = q.link;
            free(q);
            c = 0;k--;
      }else{//数第1--m个人，不做特殊操作
            c++;
            p = p.link;//p每次往后移动一位
      }
  }
  return p;
}
void Josephus2(List head, int n, int m ){
    Node p = head;
    Node pre = NULL;
    int i, j;
    for(i=0;i<n-1;i++){
        for(j = 0;j<m;j++){//共数m个
```

```
            pre = p;
            p = p.link;
        }
        //删掉数m的节点
        pre.link = p.link;
        free(p);
        p = p.link;
        }
    return p;
}
```

堆

实现

```
package com.company;

import java.util.Arrays;

public class Heap {
    // 最大堆的实现
    public static void SiftDown(int []arr,int i,int n){//n表示最后一个元素的索引
        int num = arr[i];
        int child = i*2+1;
        while(child <= n){
            if(child + 1 <= n && arr[child+1] > arr[child]){//右孩子存在且右孩子大于左孩
子
                child++;//和有孩子比较
            }
            if(arr[child] < num){//当前节点大于 两个孩子中的最大值，无需继续调整
                break;
            }
            //否则
            arr[i] = arr[child];
            i = child;
            child = 2*i+1;
        }
        arr[i] = num;
    }
    //从堆的最后一个节点[n-1]向上依次判断
    public static void SiftUp(int []arr,int n){//n表示最后一个元素的索引
        int num = arr[n];
        int father = (n-1)/2;
        while (arr[father] < num && n!=0){//n表示当前元素的索引
            arr[n] = arr[father];
```

```
            n = father;
            father = (n-1)/2;
        }
        arr[n] = num;
    }
    public static void Delete(int []arr,int n){//n表示最后一个元素的索引
        arr[0] = arr[n];
        arr[n] = -1;
        SiftDown(arr,0,n-1);
    }
    public static void Insert(int []arr,int num,int n){//n表示添加后，最后一个元素的索引
        arr[n] = num;
        SiftUp(arr,n);
    }
    public static void Create(int []arr, int n){
        for(int i = n/2 - 1;i>=0;i--){
            SiftDown(arr,i,n);
        }
    }

    public static void main(String[] args) {
        int[] data = { 15, 13, 1, 5, 20, 12, 8, 9, 11 };
        System.out.println(Arrays.toString(data));
        // 测试建堆
        Create(data, data.length - 1);
        System.out.println(Arrays.toString(data));
        // 测试删除
        Delete(data, data.length - 1);
        Delete(data, data.length - 2);
        System.out.println(Arrays.toString(data));
        // 测试插入
        Insert(data, 3, data.length - 2);
        System.out.println(Arrays.toString(data));

    }


    // 注意：  双亲是 (i-1)/2,最大非叶结点是 : n/2 - 1
```

树

设F是一个森林,B是由F变换得到的二叉树。若F中有n个非终端结点,则B中右指针域为空的结点有()个

- n-1
- n
- n+1
- n+2

解：

1. 极端例子： a b 这样一个森林， n = 1，变成二叉树后， 还是这个样子， 右指针为空的节点为两个。 所以是n+1 当然， 森林由多棵树组成都可以， 最终结果都是n+1

链接： https://www.nowcoder.com/questionTerminal/b4e60d4df3dc43ac904b44114885430d 来源：牛客网

2. 设二叉树结点数为X，则有：右空链域+右非空链域=X（因为每个节点有一个 右链域 ） 同时由二叉树性质：X= 左非 空链域+ 右非 空链域+1 （ *根节点加左右孩子数=总节点数* ） 然后如果二叉树中某节点有左孩子，则在原来的树中该节点一定为非叶节点， 因此： 左非 空链域=n => X=n + 右非 空链域+1，代入 X= 右空链域+右非 空链域，得n+1= 右空链域，原题得解。

遍历方式	**先序**	**中序**	**后序**
查找第一个	最上面的根节点	最**左**下的节点	最**左**下的节点
查找最后一个	最右**下**的叶子节点	最**右**下的节点	最上面的根节点

问题

1. 含有n个节点的平衡二叉树的最大深度：

若平衡二叉树的高度为6，且所有非叶子节点的平衡因子均为1，则该平衡二叉树的节点总数为()

构造高度为h的平衡二叉树所需的最少节点数：

解：

平衡二叉树节点数的递推公式：N0 = 0,N1 = 1,N2 = 2, $N_h = N_{h-1} + N_{h-2} + 1$ (h为平衡二叉树高度，Nh为构造此高度的平衡二叉树所需的最少节点数，= 左子树 + 右子树 + 根节点)。

2. 一颗高度为h的满m叉数有如下性质：根节点所在的层次为第一层，第h层上的节点都是叶节点，其余各层上的每个节点都有m棵非空子树，如果按层次自顶向下，同一层自左向右，顺序从1开始对全部节点进行编号，试问：编号为i的节点的第一个字节点的编是多少；编号为i的节点的双亲节点的编号是多少；

   满k叉树编号为i的节点第一个孩子的编号 j 满足 $j = (i-1) * k + 2$; 推导如下：

   设: 节点 i 处在该 m 叉树的第 h层, (h = 1, 2, 3...)

   前$h-1$ 层共有节点 $N_1 = \frac{m^{h-1}-1}{m-1}$个

   前h 层共有节点 $N_2 = \frac{m^{h}-1}{m-1}$ 个

   第 i 个节点是第 h 层第 $i - N_1$ 个节点，其有左兄弟 $i - N_1 - 1$ 个

   故节点i 的第一个孩子共有 $(i - N_1 - 1) * k$个左堂兄弟

   则节点 i的第一个孩子的编号为 $N2 + (i - N_1 - 1) * k + 1$

整理 即为 $(i-1)*m+2$

第i 个节点的双亲节点的编号为 $\lfloor \frac{(i-2)}{m} \rfloor + 1$

遍历

```
# include<stdio.h>
typedef struct BiNode{
    int data;
    struct BiNode *lchild, *rchild;
}BiNode,*BiTree;


void PreOrderTraverse(BiNode *root){
    InitStack(S);
    BiNode *p = root;
    while(p!=NULL || !S.empty()){
        if(p != NULL){
            visit(p);
            S.Push(p);
            p = p->lchild;
        }else{
            BiNode node = S.pop();
            p = node->rchild;
        }
    }
}
void InOrderTraverse(BiNode *root){
    InitStack(S);
    BiNode p = root;
    while(p!=NUll || !S.empty()){
        if(p!= NUll){
            S.push(p);
            p = p->lchild;
        }else{
            BiNode node = S.pop()
            visit(node);
            p = node.rchild;
        }
    }
}

void PostOrderTraverse(BiNode * root){
    InitStack(S);
    BiNode curr = root;
    BiNode pre = root;
    while(curr != NULL || !S.empty()){
```

```
        if(curr != NULL){
            S.push(curr);
            curr = curr.lchild;
        }else{
            //查看当前栈顶元素
            curr = S.top()
            //如果其右子树也为空，或者右子树已经访问
            //则可以直接输出当前节点的值
            if(curr.right == NULL|| curr.rchild == pre){
                visit(top);
                S.pop();
                pre = curr;  // visit 后 将此节点设为pre , pre只标注上一个被visit的节点，只有这一段代码进行了visit
                curr = NULL; // 这个节点的子树访问结束
            }else{
                curr = curr.rchild;
            }
        }
    }
}
// 从上到下，从左到右
void LevelTraverse(BiNode *root){
    if (root == NULL ){
        return;
    }
    InitQueue(Q);
    Q.push(root);
    while(!Q.empty()){
        BiNode *node = Q.pop()
        visit(node);
        if(node.lchild)Q.push(node.lchild);
        if(node.rchild)Q.push(node.rchild);
    }
}
// 从上到下，从右到左
void LevelTraverse(BiNode * root){
    if(root ==NULL)return ;
    Init(Queue);
    Queue.push(root);
    while(!Queue.empty()){
        BiNode *node = Queue.pop();
        visit(node);
        if(node.rchild)Queue.push(node.rchild);
        if(node.lchild)Queue.push(node.lchild);
    }
}
```

最大高度

```
//非递归 求数的高度-- 计数器法
int getHight(Binode *root){
    if(root ==NULL)return 0;
    Init(Queue);
    Queue.push(root);
    int count = 1;//储存当前层的长度
    while(!Queue.empty()){
        BiNode * node = Queue.pop();
        count-- ;// 栈中出去一个元素，count--;
        if(node.lchild)Queue.push(node.lchild);
        if(node.rchild)Queue.push(node.rchild);
        if(count == 0 ){
            count == Queue.length();// count = 0 时，上一层的元素遍历结束，此时队列的长度
就是下一层的长度
        }
    }
}
//非递归 求数的高度，标志位法
int getHight(BiNode *root){
    if(root == NULL)return 0;
    InitQueue(Queue);
    Queue.push(root);
    Queue.push(NULL);// 在每一层结束的地方放置一个空指针
    while(!Queue.empty()){
        BiNode * node = Queue.pop();
        if(node == NULL){// 如果访问到NULL，则在队尾放置一个空指针
            Queue.push(NULL);
            continue;
        }
        if(node.lchild)Queue.push(node.lchild);
        if(node.rchild)Queue.push(node.rchild);
    }
}
```

最大深度

leetcode559. N叉树的最大深度

方法一 递归算法

```
int maxDepth(Node* root) {
    if (!root) return 0;
    int m = 0;
    for (Node* it : root->children)
        m = max(m, maxDepth(it));
    return ++m;
}
```

方法二 DFS迭代

```
int maxDepth(Node* root) {
    if (!root) return 0;
    stack<pair<Node*,int>>stack;
    stack.push(pair<Node*, int>(root,1));
    int max_depth = 0;
    while (!stack.empty()) {
        Node* node = stack.top().first;
        int depth = stack.top().second;
        stack.pop();
        for (Node* it : node->children)
            stack.push(pair<Node*, int>(it, depth + 1));
        max_depth = max(max_depth, depth);
    }
    return max_depth;
}
```

方法三 BFS迭代

```
int maxDepth(Node* root) {
    if (!root) return 0;
    queue<Node*>queue;
    queue.push(root);
    int max_depth = 0;
    while (!queue.empty()) {
        max_depth++;
        for (int size = queue.size(); size; size--) {
            Node* curr = queue.front(); queue.pop();
            for (Node* it : curr->children)
                queue.push(it);
        }
    }
    return max_depth;
}
```

判断BST

https://blog.csdn.net/fly_yr/article/details/52172839

1. 错误解法：对每一个节点，检测其值是否大于左子树节点，是否小于右子树节点。

```
bool isBST(TreeNode* root)
{
    if (root == NULL)
        return true;
    if (root->left != NULL && root->left->data> root->data)
        return false;
    if (root->right != NULL && root->right->data < root->data)
        return false;
    if (!isBST(root->left) || !isBST(root->right))
        return false;
    return true;
}
反例
                    3
              /      \
           2          5
        /     \
    1          4
```

2. 把中序遍历的结果存起来，然后判断是否递增
3. 但是上述方法需要额外线程空间保存遍历结果，在此可以省去该空间开销，只需一个变量保存访问当前节点时上一节点的值即可
4. 另一种方法，对于每一个子树：left<=current<right；

```
/*方法一，将中序遍历结果保存到数组 T(n)=O(n) S(n)=O(n)*/
    void inOrder(TreeNode *root,vector<int> &v)
    {
        if (root == NULL)
            return;
        inOrder(root->left, v);
        v.push_back(root->val);
        inOrder(root->right, v);
    }

    bool checkBST1(TreeNode* root)
    {
        vector<int> ret;
        inOrder(root, ret);
        for (auto i = ret.begin()+1; i != ret.end(); ++i)
        {
            if (*i < *(i - 1))
                return false;
        }
        return true;
    }
/*方法二、省掉线性空间，保存遍历的最后一个节点*/
```

```
    int lastVal = INT_MIN;// 如果要考虑到元素的值会取INT的最大最小值 double lastVal
= - Double.MAX_VALUE;
    bool checkBST2(TreeNode* root) {
        if (!root)
            return true;

        /*递归检查左子树*/
        if (!checkBST2(root->left))
            return false;

        /*比较当前节点，并更新已遍历节点最后的值*/
        if (root->val <= lastVal)
            return false;
        lastVal = root->val;

        /*递归检查右子树*/
        if (!checkBST2(root->right))
            return false;
        return true;
    }
/*方法三，最大最小值法*/
    bool checkBST3(TreeNode* root) {
        // write code here
        if (!root)
            return true;
        return checkBST3(root, INT_MAX, INT_MIN);
    }
    bool checkBST3(TreeNode *root, int maxVal, int minVal)
    {
        if (!root)
            return true;
        if (root->val < minVal || root->val >= maxVal)
            return false;
        if (!checkBST3(root->left, root->val, minVal) || !checkBST3(root->right,
maxVal, root->val))
            return false;
        return true;
```

————————————————


```
//


BinaryNode min(BinaryNode n){
    if(n == null)return null;
```

```
    while(n.left != null)n = n.left;
    return n;
}
BinaryNode max(BinaryNode n){
    if(n==null)return null;
    while(n.right != null)n=n.right;
    return n;
}
int isBST(BinaryNode t){
    if(t == null)return 1;
    else if(t.left != null && max(t.left).data > t.data)return 0;
    else if(t.right != null && min(t.right).data < t.data)return 0;
    else return isBST(t.left)&&isBST(t.right);
}
```

个人觉得第二种最好理解。

BST的插入

```
void insert(BinaryTreeNode<T>* root ;int x){
    if(root==NULL){
        root = new BinaryTreeNode(x);
    }
    if(root->data == x){
        return;
    }else if(root->data < x){
        insert(root->rightChild,x);
    }else{
        insert(root->leftChild;x);
    }
}
```

如果二叉查找树的每个结点都存储了关键码data和含有相同关键码data的结点的个数count

```
void insert(BinaryTreeNode<T>* root ;int x){
    if(root==NULL){
        root = new BinaryTreeNode(x);
        root->cout = 1;
    }
    if(root->data == x){
        count++;
    }else if(root->data < x){
        insert(root->rightChild,x);
    }else{
        insert(root->leftChild;x);
    }
}
```

孩子兄弟表示法--求各节点的度

```
class BinaryNode{
        int data,degree;
    BinaryNode firstChild;
    BinaryNode nextSibling;
}
public void degree(BinaryNode t){
    if(t==null)return;
    t.degree = 0;
    if(t.firstChild!=null){
        t.degree++;
        BinaryNode p = t.nextSibling;
        while(p.nextSibling!=null){
            t.degree++;
            p = p.nextSibling;
        }
    }
    degree(t.firstChild);
    degree(t.nextSibling);
}
```

已知先序中序建立二叉树

```
/*
 * 一颗二叉树各个节点互不相同，其先序遍历和中序遍历分别存于两个二维数组 A[1...n],B[1...n]
中，编写算法建立该二叉树的二叉链表
 * para@ A 前序遍历的数组
 * para@ B 后序遍历的数组
 * para@ Aleft 子树的前序在A中对应的开始索引
 * para@ Aleft 子树的前序在A中对应的结束索引
 * para@ Bleft 子树的后序在B中对应的开始索引
 * para@ Bleft 子树的后序在B中对应的结束索引
*/
BiNode *CreateBiTreeFromPreAndInOrder(int *A,int *B,int Aleft,int Aright,int Bleft,int
Bright){
    int i;
    int llen;//左孩子的长度
    int rlen;//右孩子的长度
    BiNode *T =(BiNode *)malloc(sizeof(BiNode));
    T.data = A[Aleft];
    for(i = Bleft;B[i]!=T.data;i++);//将i移动到根元素在B中的位置
    llen = i-Bleft; //B中i左边的为 左子树
    rlen = Bright - i; // B中i右边的为 右子树
    if(llen!=0){
```

```
        T.lchild =
CreateBiTreeFromPreAndInOrder(A,B,Aleft+1,Aleft+llen,Bleft,Bleft+llen-1);
    }else{
        T.lchild = NULL;
    }
    if(rlen!=0){
        T.rchild = CreateBiTreeFromPreAndInOrder(A,B,Aright-rlen+1,Aright,Bright-
rlen+1,Bright);
    }else{
        T.rchild = NULL;
    }
    return T;
}
```

B树和B+树

并查集

相关博客

AVL树

1.最大高度

给定节点数，求最大高度。

eg. 11个节点的最大高度

```
f(0) = 0;
f(1) = 1;
f(2) = 2;
f(3) = 4;
f(4) = 7;
f(5) = 12;
```

高度为5至少需要12个节点，故最大高度为4,

2.最小节点数

给定高度，求最少节点数。

```
f(0) = 0;
f(1) = 1;
f(2) = 2;
```

```
递推公式：
f(h) = f(h-1) + f(h-2) + 1;
```

费波那契数树是具有相同高度的所有平衡二叉树中结点 个数最少的

相同结点数目，高度最大

图

v —> w

弧头: w

弧尾：v

从v到w

v邻接到w

w邻接于(自)v

从图的邻接表表示转换成邻接矩阵表示

```
void Convert(ALGraph &G, int arcs[M][N]){
  for(int i =0 ;i < n;i++){//顶点i
    p = (G->v[i]).firstarc;//去除定点i的第一条出边
    while(p!=NULL){
      arcs[i][p->data] = 1;
      p = p->nextarc; //顶点i的下一条出边
    }
  }
}
```

邻接表转逆邻接表

```
//图定义
struct Edge{
        int dest;//
        int weight;
        Edge * link;
}
struct Vertex{
        int data;
```

```
        Edge *adj;//邻接表的头指针;
}
class Graph{
        private:
                Vertex * NodeTable;//顶点表
                int cnt;
}
for(int i = 0;i< numVertexs;i++){
        Edge* p = NodeTable[i].adj;
        while(p!=Null){
                Edge* q = new Edge(i,p->weight);
                q->link = NodeTable[p->dest].adj.link;//q 指向p->的第一个节点
                NodeTable[p->dest].reberseAdj = q;//逆邻接表头指向q

                p = p->link;
        }
}
```

广度优先搜索

```
bool visited[MAX_VERTEX_NUM];//访问标记数组
void BFSTraverse(Graph G){
  for (int i = 0;i<G.vexnum;i++){
    visited[i] = FALSE;
  }
  InitQueue(Q);
  for (int i = 0;i<G.vetnum;i++){
    if(!visited[i]){ //对每个连通分量调用一次BFS
      BFS(G,i);
    }
  }
}
//从顶点v出发，广度优先遍历G
void BFS(Grapy G,int v){
  visit(v);
  visited[v] = TRUE;
  Enqueue(v);
  while(!isEmpty(Q)){
    Dequeue(Q,v);//弹出队首元素，存储为v
    for (w = FirstNeighbor(G,v);w >= 0;w = NextNeighbor(G,v,w)){
      if(!visited[w]){
        visit(w);
        visited[w] = TRUE;
        Enqueue(w);
      }
    }
  }
```

```
}
```

复杂度分析：

空间复杂度，最坏情况下，每个节点需要入队一次 O(|V|)

时间复杂度：

- 采用邻接表：
  - 每个顶点均需要搜索一次 O(|V|)，在搜索节点的邻接点时，每条边至少要访问一次，O(|E|),时间复杂度为O(|E|+|V|)
- 采用邻接矩阵
  - 每个顶点需要访问一次，查找每个顶点的所有邻接点需要O(|V|),故时间复杂度为O(|V|^2)

用BFS算法求解单源最短路径问题

```
void BFS_MIN_Distance(Graph G, int u){
  for (int i = 0;i< G.m=vexnum;i++){
    d[i] = InFinity;
  }
  visited[u] = TRUE;
  d[u] = 0;
  EnQueue(Q,u);
  while(!isEmpty(Q)){
    Dequeue(Q,u);
    for (w = FirstNeighbor(G,u);w>=0;w = NextNeighbor(G,u,w)){
      if(!visited[w]){
        visited[w] = TRUE;
        d[w] = d[u]+1;
        EnQueue(Q,w);
      }
    }
  }
}
```

深度优先搜索

```
bool visited[MAX_VERTEX_NUM];
void DFSTraverse(Graph G,int v){
  for (int i = 0;i<G.vexnum;i++){
    visited[i] = FALSE;
  }
  for(int i = 0;i<G>vexnum;i++){
    if(!visited[i]){
      DFS(G,i);
    }
  }
```

```
}
void DFS(Graph G,int v){
  visit(i);
  visited[i] = TRUE;
  for (w = FirstNeighbor(G,v)w>=0;w = NextNeighbor(G,v,w)){
    if(!visited[w]){
      DFS(G,w)
    }
  }
}
```

非递归实现

```
// 类似于树的先序遍历
bool visited[MAX_VEX_NUM];
void DFSTraverse(Graph & G, int v){
  for(int i=0;i<G.vexnum;i++){
    visited[i] = FALSE;
  }
  InitStack(S);
  for(int i=0;i<G.vexnum;i++){ //每个连通分量
        if(!visited[i]){
      DFS(G,v);
    }
  }
}
void DFS(Graph &G,int v){
  visited(v)
  visited[v] = TRUE;
  p = v
  while(p >=0 &&!isEmpty(S)){
        if(p>=0){
      visit(p);
      visited[p] = TRUE;
      Push(S,p);
      for(w = First(G,p),w>=0;w=NextNeighbor(G,t,p)){
        if(!visited[w]){
          p = w;
          break;
        }
        p = -1;//如果后面没有节点，则设为-1
      }
    }else{
      Pop(S,t);
      for(w = First(G,p),w>=0;w=NextNeighbor(G,t,p)){//寻找下一未访问过的节点
        if(!visited[w]){
          p = w;
          break;
        }
```

```
        p = -1;//
      }
    }
  }
}
```

另一种实现

使用栈S记录下一步可能访问的节点，同时visited数组标记节点是否在栈内或曾经在栈内，若是，则以后它不能进栈。

由于使用了栈，使得遍历的方式是从右端到左端的顺序。

```
void DFS_Non_RC(Graph & G, int v){
  int w;
  InitStack(S);
  for(int i=0;i<G.vexnum;i++){
    visited[i]=FALSE;
  }
  Push(S,v);
  visited[v] = TRUE;
  while(!isEmpty(S)){
    k = Pop(S);
    visit(k);//出栈时访问
    for (w = FirstNeighbor(G,k);w>=0;w = NextNeighbor(G,k,w)){
      if(!visited[w]){//此邻接点没有入过栈
        Push(S,w);
        visited[w]=TRUE;
      }
    }
  }
}
```

复杂度分析

- 空间
  - 需要一个递归工作栈，空间复杂度为O(|V|)
- 时间
  - 邻接表 O(|V|+|E|)
  - 邻接矩阵 O(|V|^2)

算法题：

1. 判断一个无向图是否为一棵树

分析：数 等价于 G是一个无环路连通图

或者 G是一个有n-1条边的连通图

判断连通：是否可以通过一次DFS遍历所有顶点

判断 n-1 条边： 遍历的时候计算边的个数

```
bool visited[MAX_VERTEX_NUM];
bool isTree(Graph &G){
  int Vnum = 0;
  int Enum = 0;
  for (int i=0;i<G.vexnum;i++){
    visited[i] = FALSE;
  }
  DFS(G,1,Vnum,Enum,visited);
  if(Vnum == G.Vexnum && Enum = 2*(G.Vexnum - 1)){
    return TRUE;
  }else{
    return FALSE;
  }

}
void DFS(Graph & G,int v,int & Vnum,int & Enum,int visited[]){
  visited[v] = TRUE;
  Vnum++;
  for (w = FirstNeighbor(G,v);w>=0;w=NextNeighbor(G,v,w)){
    Enum++; //不管邻接点有没有被访问过，边都是存在的
    if(!visited[w]){
      visited[w] = TRUE;
      DFS(G,w,Vnum,Enum,visited[]);
    }
  }
}
```

2. 判断i到j有没有路径
    1. 用DFS

```
bool visited[MAXSIZE];
int Exist_Path_DFS(Graph & G,int i ,int j){
  for (int i=0;i<G.vexnum;i++){
    visited[i]=FALSE;
  }
  if(i==j){//递归出口
    return 1
  }
  int p;
  for(p = FirstNeighbor(G,i);p>=0;p=NextNeighbor(G,i,p)){
    if(!visited[p] && Exist_Path_DFS(G,p,j)){//递归调用
      return 1;
    }
  }
```

```
  return 0
}
```

2. BFS

```
int visited[MAX_NUM]={0}
int Exist_Path_BFS(Graph & G,int i,int j){
  InitQueue(Q);EnQueue(i);
  while(!isEmpty(Q)){
    DeQueue(Q,i);
    visited[i]=1;
    for (p = FirstNeighbor(G,i);p>=0;p=NextNeighbor(G,i,p)){
      k = p.adjvex;//该弧指向的节点
      if(k==j)return 1;
      if(!visited[k]){
        EnQueue(Q,k)
      }
    }
  }
}
```

3. 利用以i为参数的 DFS(G,i) 或 BFS(G,i),执行结束后判断 visited[j] 是否为 TURE，但这样每次都耗费最坏时间复杂度，需要遍历Vi 连通的所有顶点。

3. 输出从 Vi 到 Vj 的所有 简单路径

   基于递归的深度优先算

```
void FindPath(AGraph &G, int u,int v,int path[],int d){//d : 路径长度,初始为-1
  //声明要用到的变量
  int wl;
    ArcNode *p;
  d++;
  path[d] = u;
  visited[u]=1;
  if(u == v){
    print(path);
  }
  p = G->adjust[u].firstarc;//p指向v的第一个相邻边
  while(p!=NULL){
    w = p->adjvex;//p指向的顶点
    if(!visited[w]){
      FindPath(G,w,v,path,d);
    }
    p = p->nextarc;//p指向下一个相邻边节点
  }
    visited[u]=0;//回溯
}
```

最小生成树

Prim	Kruskal
|V|^2（邻接矩阵）O(E + V log(V)) (邻接表)	|E|log(|E|)
适合稠密图	适合稀疏图
	采用堆存储边的集合
当带权连通图的任意一个环中所包含的边权值均不相同时，其MST唯一(充分条件)	

最短路径

算法	Dijkatra	Floyd
时间复杂度	|V|^2	|V|^3
	不适用于边上带有负权值	允许带有负权值的边，但不允许包含负权值的边组成的回路

拓扑排序

有向无环图（Directed Acycline Graph, DAG）是一类特殊的有向图。DAG有着广泛应用，AOE网和AOV网都是DAG的典型应用。

AOV网（Activity On Vertex NetWork）用顶点表示活动，边表示活动（顶点）发生的先后关系。 若网中所有活动均可以排出先后顺序（任两个活动之间均确定先后顺序），则称网是拓扑有序的，这个顺序称为网上一个全序。(详情参见离散数学/图论相关内容)。

在AOE网上建立全序的过程称为拓扑排序的过程，这个算法并不复杂：

1. 在网中选择一个入度为0的顶点输出
2. 在图中删除该顶点及所有以该顶点为尾的边
3. 重复上述过程，直至所有边均被输出。

AOE网(Activity On Edge Network)是边表示活动的网，AOE网是带权有向无环图。边代表活动，顶点代表 所有指向它的边所代表的活动 均已完成 这一事件。由于整个工程只有一个起点和一个终点，网中只有一个入度为0的点（源点）和一个出度为0的点（汇点）

拓扑排序算法实现

```
bool TopologicalSort(Graph G){
  //如果G存在拓扑序列，返回true
  InitStack(S);//栈用来存储入度为0的点
  for (int i=0;i<G.vexnum;i++){
    if(indegree[i]==0){
```

```
      Push(S,i);//将所有入度为0的带你进栈
    }
  }
  int count=0;// 计数，记录当前已经输出的顶点数
  while(!IsEmpty(S)){
    Pop(S,i);
    print(i);
    count++;
    for(p=G.vertices[i].firstarc;p;p=p->nextarc){
      v=p->adjvex;
      if(!--indegree[v]){// 先让入度减一 ，入度为零，入栈
        Push(v)
      }
    }
  }
  if(count<G.vexnum){
    return false;
  }else{
    return true;
  }
}
```

由于输出每个顶点的同时还要删除以它为起点的边(遍历边，使边节点对应的顶点的入度减一)，故

如果使用 邻接表存储时间复杂度为O(|V|+|E|)

如果使用 邻接矩阵存储 时间复杂度为 O(|V|^2) 因为每次遍历边节点的时候都需要遍历邻接矩阵中的那一行

错题

1. 判断有向图是否有环：

- 深度优先遍历，如果从有向图上某个顶点u出发，在DFS(u)结束之前出现一条从v到u的边，则必定有环
- 拓扑排序，当某个顶点不为任何边的头是，才能加入序列，存在环路是环路中的顶点一直是某条边的头，不能加入拓扑序列。
- 关键路径

2. 最短路径一定是简单路径
3. 求出强连通分量数目

当某个顶点只有出弧没有入弧时，其他顶点无法到达这个顶点，不可能与其他顶点和边构成强连通分量(这个顶点单独构成一个强连通分量)

```
    * 顶点1无入弧，构成一个强连通分量，删除该顶点和以它为弧尾的弧
    * 顶点2 。。。
    * 。。
    * 以此类推
```

算法

利用DFS实现DAG的拓扑排序

思路：在DFS递归调用中，祖先会调用子孙节点的DFS，所以如果 u—>v,则v的DFS要比u的DFS先结束。

```
bool visited[MAX_VERTEX_NUM]
int finishTime[MAX_VERTEX_NUM]={0}
void DFSTraverse(Graph G){
    for(int i=0;i<G.vexnum;i++){
        visited[i]=false;
    }
    time=0;//设定一个初始时间戳
    for(int i=0;i<G.vexnum;i++){
        if(!visited[i])
            DFS(G,i);
    }
}
void DFS(Graph G,int v){
    visited[v]=true;
    visit(v);
    for(w=FirstNeighbor(G,v),w>=0;w=NextNeighbor(G,v,w)){
        if(!visited[w]){
            DFS(G,w);
        }
        time=time+1;//每访问一层，时间戳++
        finishTime[w]=time;
    }
}
```

查找

静态查找	动态查找
顺序查找、折半查找、散列查找	二叉排序树（二叉平衡树、B树）、散列查找

1.顺序查找

	一般顺序表的顺序查找	有序表的顺序查找	折半查找	分块查找
成功是平均查找长度	(n+1)/2	(n+1)/2	$log_2 n$	
不成功	n+1	$\frac{n}{2}+\frac{n}{n+1}$	$log_2 n$	
优点	对数据元素的存储没有要求，顺序或链式，对表中记录的有序性也没有要求		要求存储必须是迅疾存取，适合线性表的顺序存储，	吸取了顺序查找和折半查找各自的优点，既有动态结构，又适于快速查找。
缺点	当n较大时，平均查找长度较大，效率低		不适合链式存储，且要求关键字有序.	
备注	(对线性链表只能用顺序查找)	(对线性链表只能用顺序查找)	判定"树高 $\lceil log(n+1) \rceil$ 或 $\lfloor log_k n \rfloor + 1$	将长度为n的查找表均匀分成b块，每块有s个记录，在等概率下，平均查找长度为 $ASL = L_I + L_S = \frac{b+1}{2} + \frac{n+1}{2} = \frac{s^2+2s+n}{2s}$此时若$s = \sqrt{n}$，则平均查找长度取最小值：$\sqrt{n}+1$ 若对索引表采用折半查找，则：$ASL = L_I + L_S = log_2(b+1) + \frac{n+1}{2}$

2. 折半查找(二分查找)

适用于有序的顺序表

```
int Binary_Search(SeqList L,ElemType key){
  int left=0;
  int right = L.TableLen-1;
  int mid;
  while(left <= right){
    mid = (left+right)/2;
    if(L.elem[mid] == key){
      return mid;
    }else if(L.elem[mid] > key){
      right = mid -1;
    }else{
      left = mid +1;
    }
  }
  return -1;
}
```

折半查找的递归解法

```
typedef struct{
  ElemType *elem;
```

```
  int length;
}SSTable;
int BinSearchRec(SSTable ST,ElemType key, int left, int right){
  if(left > right){
    return 0;
  }
  mid = (left+right)/2;
  if(key>ST.elem[mid]){
    BinSearchRec(ST,key,mid+1,right);
  }
  else if(key < ST.elem[mid]){
    BinSearchRec(ST,key,left,mid-1);
  }else{
    return mid;
  }
}
```

递归：时间复杂度：$log_2 n$ 空间复杂度$log2n$

3. 分块查找(索引顺序查找)

见上表

4.B 树和 B+树

错题

1. 具有n个关键字的m阶B树，应有($n+1$) 个叶节点

   B树叶节点对应查找失败的情况，对n个关键字查找，失败的可能性有N+1种
2. 高度为5的3阶B树，至少有 ($2^5-1=31$) 个节点，至多有 ($(3^5-1)/2=121$) 个节点
3. 高度为2 的5 阶B树中，所含关键字最少为 (5)

   m阶B树，只要保证：除根节点外的所有非叶节点的关键字 n 满足 ($\lceil m/2 \rceil -1 \le n \le m-1$)

   并不需要有某个节点有m个子节点
4. 一颗具有15个关键字的4阶B树中，含有关键字的节点个数最多是()
5. 含有n个非叶几点的m阶B树中至少包含 ($(n-1)(\lceil m/2 \rceil)+1$) 个关键字
6. 已知一颗5阶B树种共有53个关键字，则树的最大高度是(4)，最小高度是(3)

   $log_m(n+1) \le h \le log_{\lceil m/2 \rceil}[(n+1)/2]+1$

   1. 因为B树中每个节点最多有m个子树，m-1个关键字，所以

      $n \le (m-1)(1+m+m^2+\ldots+m^{h-1})=m^h-1$

      故 $h \ge log_m(n+1)$

2. 若让每个节点中的关键字个数达到最少，则B树的高度最大:第1层最少 1 个节点，第二层最少2个节点，第三层最少 $2\lceil m/2\rceil$个节点 。。。第 h+1 层最少$2\lceil m/2\rceil^{h-1}$个节点，(第h+1层是不包含信息的叶节点)，对于关键子数为n的B数，查找不成功的节点为 n+1 ， 由此 $n+1 \geq 2\lceil m/2\rceil^{h-1}$ 即$h \leq log_{\lceil m/2\rceil}[(n+1)/2]+1$

7. B树只支持多路查找，不支持顺序查找，而B+树可以顺序查找

B树 & B+树 的插入删除操作 blog

散列(Hash)表

冲突： 散列函数可能会把两个或两个以上的不同关键字映射到同一地址，称这种情况为"冲突"，发生碰撞的不同关键字称为同义词

散列表：根据关键字而直接进行访问的数据结构

理想情况下，对散列表进行查找的时间复杂度为O(1)

散列函数的构造方法

注意

- 散列函数的定义域必须包含全部需要存储的关键字
- 值域范围依赖于散列表的大小或地址范围
- 散列函数计算出来的地址应该能等概率、均匀地分布在整个地址空间，从而减少冲突的发生
- 散列函数应该尽量简单，能够在较短时间内就计算出任一关键字对应的散列地址

常用的散列函数

名称	直接地址法	除数留余法	数字分析法	平方取中法	折叠法
函数	$H(key)=a\times key+b$	$H(key)=key$	设关键字是r进制数，而r个数码在各位上出现的频率不一定相同，可能在某些位上分布的均匀写，在某些位上分布的不军均匀，则应选择分布较均匀的若干位作为散列地址。	取关键字的平方值的中间几位作为散列地址。	将关键字分割成位数相同的几部分(最后一部分的位数可以短一些)，然后取这几部分的叠加和作为散列地址。
优点	计算最简单，并且不会冲突	最简单、最常用的方法		得到的散列地址与关键字的每一位都有关系，使得散列地址分布地比较均匀	
缺点	若关键字不连续，空位较多，造成存储空间浪费		如果换了关键字，就需要重新构建新的散列函数		
适用情况	适合关键字分布基本连续的情况	假定散列表表长m。取一个不大于m但最接近或等于m的质数p	适合已知的关键字集合	适用于关键字的每一位取值都不够均匀或均小于散列地址所需的位数	关键字位数很多，关键字的每一位上数字分布大致均匀

处理冲突的方法

1. 开放地址法

   $H_i=(H(key)+d_i)\%m\ i=0,1,2,\dots,k(k\leq m-1)$ m表示散列表长，d_i 为增量序列

	线性探测	平方探测(二次探测法)	再散列法(双散列法)	伪随机序列法
探测方法	$d_i=0,1,2,\dots,m-1$	$d_i=0^2,1^2,2^2,\dots,k^2,-k^2$ 其中$k\leq m/2$，散列表长度m必须是一个可以表示成 4k+3 的素数	$d_i=Hash_2(Key)$ $H_i=(H(key)+i*Hash_2(Key))\%m$	$d_i=$伪随机序列
优点		避免出现"堆积"问题		
缺点	造成大量元素在相邻的散列地址堆积起来，大大降低查找效率	不能探测到散列表上的所有单元，但至少能探测到一半单元		
			初始探测位置 $H_0=H(Key)\%m$，i是冲突次数，初始为0，最多经过m-1次探测会遍历表中所有的位置，回到H_0位置	

2. 拉链法(链接法)

把所有同义词存储在一个线性链表中，这个线性链表由其散列地址唯一标识

散列查找和性能分析

查找效率取决于： 散列函数、处理冲突的方法、装填因子

装填因子：$\alpha = \frac{\text{表中记录数}n}{\text{散列表长度}m}$

散列表的平均查找长度依赖于散列表的装填因子α，而不直接依赖于m或n，α越大，表示装填的记录越"满"，发生冲突的可能性越大，范围发生冲突的可能性越小。

要求：可以在求出散列表的基础上计算查找成功时的平均查找长度和查找不成功的平均查找长度。

错题

1. 若 $\alpha < 1$,则冲突可以避免 (X)。冲突是不可避免的，与装填因子无关。
2. 如何在散列表中删除一个记录

   在拉链法情况下可以物理地删除，在开放地址法情况下，不能物理地删除，只能做删除标记。该地址可能是该记录的同义词查找路径上的地址，物理地删除会中断查找路径。因为查找时碰到空地址就认为是查找失败。
3. 查找长度 = 比较次数 = 探测次数 + 1 = 冲突次数 +1
4. 求查找失败的平均查找长度有两种观点：其一，任务比较到空节点才算失败，所以比较次数等于冲突次数+1，其二，任务只有与关键字的比较才算比较次数。

KMP算法

blog

那就是说需要搜索的词如果内部完全没有重复，那这个算法就退化成了遍历?

模式串下标从0开始

据《最大长度表》，失配时，模式串向右移动的位数 = 已经匹配的字符数 - 失配字符的上一位字符的最大长度值 而根据《next 数组》，失配时，模式串向右移动的位数 = 失配字符的位置 - 失配字符对应的 next 值

其中，从0开始计数时，失配字符的位置 = 已经匹配的字符数（失配字符不计数），

而失配字符对应的next 值 = 失配字符的上一位字符的最大长度值，两相比较，结果必然完全一致。

GetNext()

```
void GetNext(char *p,int next[]){
    int pLen = strlen(p);
    next[0] = -1;
    int k=-1;
    int j= 0;
    while(j < pLen-1){
```

```
        if(k==-1 || p[j] == p[k]){//递归地找到了，或找失败
                        ++k;
            ++j;
            next[j] = k;
        }else{
            k = next[k];//递归地查找
        }
    }
}
```

改进的GetNext()

```
void GetNext(char *p,int next[]){
    int pLen = strlen(p);
    next[0] = -1;
    int k=-1; //k保存的是上一个j对应的next值
    int j= 0;
    while(j < pLen-1){
        if(k==-1 || p[j] == p[k]){//递归地找到了，或找失败
                        ++k;
            ++j;
            if(p[j]!=p[k]){
                next[j] = k;// 下一个j的next = 上一个j对应的next(也就是k) + 1
            }else{
                next[j] = next[k];
            }
        }else{
            k = next[k];//递归地查找
        }
    }
}
```

KMP

```
int KMPSearch(char *s ,char *p){
    int i=0;
    int j=0;
    int sLen = strlen(s);
    int pLen = strlen(p);
    while(i < slen && j < pLen){
        //①如果j = -1，或者当前字符匹配成功（即S[i] == P[j]），都令i++，j++
        if (j == -1 || s[i] == p[j]){
            i++;
            j++;
        }else{
            j = next[j];//失配时，移动j到对应next处
        }
    }
```

```
    if(j==pLen){
        return 1;
    }else{
        return 0;
    }
}
```

排序

排序	最好时间	最坏时间	平均时间	平均空间	最坏空间	稳定性	适用
直接插入排序	n	n^2	n^2	1	1	√	顺序&链式(从前往后查找指定元素的位置)
折半插入			n^2	1	1	√	顺序
希尔排序		n^2	$n^{1.3}$	1	1	X	
冒泡排序	n	n^2	n^2	1	1	√	
快速排序	$nlog_2n$	n^2	$nlog_2n$	log_2n	n	X	第i趟排序，一定会有超过i个元素排到最终位置
简单选择排序	n^2	n^2	n^2	1	1	X	与元素的初始状态无关
堆排序	$nlog_2n$	$nlog_2n$	$nlog_2n$	1	1	X	
归并排序	$nlog_2n$	$nlog_2n$	$nlog_2n$	n	n	√	2-路归并排序,比较次数与初始状态无关
基数排序	$d(n+r)$	$d(n+r)$	$d(n+r)$	r	r	√	先比较个数数
外部排序—多路归并排序							

对于任意序列进行基于比较的排序，最少的比较次数，应该考虑在最坏的情况下。比较次数至少为$\lceil log_2n! \rceil$

比较次数&交换次数

	稳定性	比较次数	交换次数	空间复杂定	使用场景
冒泡排序	稳定	n(n-1)/2	逆序数	O(1)	
选择排序	不稳定	n(n-1)/2	0~(n-1)	O(1)	
插入排序	稳定	最好为(n-1)，最差为n(n-1)/2	最好为0，最差为n(n-1)/2	O(1)	初始序列大量有序
归并排序	稳定	nlog2(n)/2~nlog2(n)-n+1		O(n)	大量数据排序，外排序
快速排序	不稳定	nlog(n)，最差为n(n-1)/2	无法分析	log(n)	
堆排序	不稳定	nlog(n)，和初始顺序关系不大	无法分析	O(1)	

1. 冒泡：对于n个元素，相邻元素均要比较，共有(n-1)次。经过一回合冒泡过程后，最大元素沉淀到最右位置。第二回合， 只剩下(n-1)个元素，只需要比较(n-2)次。依次类推，其他比较次数为(n-3),.....,2,1. 所以总共比较次数为n(n-1)/2，而且是固定为这个数目
2. 选择：下面是比较次数。对于n个元素的序列，找出最小元素需要比较(n-1)次。第一回合后，序列只剩下(n-1)个元素，下一次找最小元素还需要(n-2)次比较。最后直到2个元素需要比较1次。所以最后比较次数总共为(n-1)+(n-2)+...+1=n(n-1)/2，且固定不变。
3. 插入：最好情况下（有序），需要比较(n-1)次，移动0次；最差情况下，需要比较1+2+...+(n-1)=n(n-1)/2次，移动n(n-1)/2次

插入排序

直接插入排序

```
public static void InsertSort(int [] a){
  int len = a.length;
  int i;
  int j;
  int temp;
  for (i = 1;i<len;i++){
    j = i-1;
    temp = a[i];
    while(j>=0 && a[j] > temp){
      a[j+1] = a[j];
      j--;
```

```
    }
    a[j+1] = temp;
  }
}
```

折半插入

```
public static void BinaryInsertSort(int []a){
  int i,j,left,right,mid;
  int temp;
  int len = a.length;
  for (i=1;i<len;i++){
    temp = a[i];
    left = 0;
    right = i-1;
    while(left<=right){
      mid = (left+right)/2;
      if(a[mid] > temp){
        right = mid-1;
      }else{
        left = mid+1;
      }
    }
    for(j = i-1;j>right;j--){
      a[j+1] = a[j];
    }
    a[right+1] = temp;
  }
}
```

希尔排序

```
public static void ShellSort(int []a){
  int d = a.length/2;
  int temp;
  int i,j,k;
  while (d>=1){
    for(k = 0;k<d;k++){//每组的第一个元素为a[k]
      for(i = k+d;i<a.length;i = i+d){
        temp = a[i];
        j = i-d;
        while (j>=k && a[j] > temp){
          a[j+d] = a[j];
          j -= d;
        }
        a[j+d] = temp;
      }
    }
```

```
    d =d/2;
  }
}
```

交换排序

冒泡排序

```
public static void BubbuleSort(int []a){
  int len = a.length;
  int i,j,temp;
  for(i = 0;i<len-1;i++){
    for(j = len-1;j>0;j--){
      if(a[j] < a[j-1]){
        temp = a[j];
        a[j] = a[j-1];
        a[j-1] = temp;
      }
    }
  }
}
//如果某一趟没有发生交换，则说明已经排好序，直接跳出即可
public static void BubbuleSort2(int []a){
  int len = a.length;
  int i,j,temp;
  boolean flag;
  for(i = 0;i<len-1;i++){
    flag = true;
    for(j = len-1;j>0;j--){
      if(a[j] < a[j-1]){
        temp = a[j];
        a[j] = a[j-1];
        a[j-1] = temp;
        flag  = false;
      }
    }
    if(flag)break;
  }
}
//双向排序
public static void DoubleBubbleSort(int[] a) {
  int start = 0;
  int end = a.length - 1;
  boolean flag = true;
  int i, temp;
  while (start < end && flag) {
    flag = false;
    for (i = start; i < end; i++) {
      if (a[i] > a[i + 1]) {
```

```
        temp = a[i];
        a[i] = a[i + 1];
        a[i + 1] = temp;
        flag = true;
      }
    }
    end--;
    for (i = end; i > start; i--) {
      if (a[i] < a[i - 1]) {
        temp = a[i];
        a[i] = a[i - 1];
        a[i - 1] = temp;
        flag = true;
      }
    }
    start++;
  }
}
```

快排

```
public static void QuickSort(int[] arr, int begin, int end) {
  if (end <= begin) return;
  int middle = partition2(arr, begin, end);
  QuickSort(arr, begin, middle - 1);
  QuickSort(arr, middle + 1, end);
}

public static int partition(int[] arr, int begin, int end) {
  int key = arr[begin];
  while (begin < end) {
    while (begin < end && key <= arr[end]) end--;
    arr[begin] = arr[end];
    while ((begin < end && key >= arr[begin])) begin++;
    arr[end] = arr[begin];
  }
  arr[begin] = key;
  return begin;
}
```

提高算法效率：

1. 递归过程中划分得到的子序列的规模较小时，不要再继续递归调用快速排序，可以直接采用直接插入排序算法进行后续工作
2. 尽量选取一个可以将数据中分的数据元素
    1. 从序列头尾和中间选取三个元素，去三个元素的中间值
    2. 随机选取轴元素

选择排序

简单选择排序

```
public static void SelectSort(int[] arr) {
  int i, j, min, temp;
  for (i = 0; i < arr.length; i++) {
    min = i;
    for (j = i + 1; j < arr.length; j++) {
      if (arr[j] < arr[min]) {
        min = j;
      }
    }
    if (min != i) {
      temp = arr[min];
      arr[min] = arr[i];
      arr[i] = temp;
    }
  }
}
```

堆排序

1. 向具有n个节点的堆中插入一个元素的时间复杂度为 $O(log_2 n)$，删除一个元素的时间复杂度为 $log_2 n$
2. 构建一个n个记录的初始堆，时间复杂度为 n ，堆排序的时间复杂度为 $log_2 n$

```
    public static void HeapSort(int []arr){
        for(int i=arr.length/2-1;i>=0;i--){
            HeapAdjust(arr,i,arr.length-1);
        }
        for(int i = arr.length-1;i>=0;i--){
            swap(arr,0,i);
            HeapAdjust(arr,0,i-1);
        }
    }
    public static void HeapAdjust(int []arr, int i,int n){
        int num = arr[i];
        int child = i*2+1;
        while (child <= n){
            if(child+1 <=n && arr[child+1] > arr[child]){
                child++;
            }
            if(arr[child] < num){
                break;
            }
            arr[i] = arr[child];
            i = child;
```

```
            child = 2*i+1;
        }
        arr[i] = num;
    }
    public static void swap(int []arr,int i,int j){
        int temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
    }
```

归并排序和基数排序

归并排序

```
public static void Merge(int arr[],int start ,int mid, int end){
  int[] temp =new int[end-start+1];
  int i = start;
  int j = mid+1;
  int k = 0;
  while (i<= mid && j<=end){
    if( arr[i] < arr[j]){
      temp[k++] = arr[i++];
    }else{
      temp[k++] = arr[j++];
    }
  }
  while (i<= mid){
    temp[k++] = arr[i++];
  }
  while(j <= end){
    temp[k++] = arr[j++];
  }
  for(int t = 0;t<temp.length;t++){
    arr[t+start] = temp[t];
  }
}
public static void MergeSort(int arr[],int start,int end){
  int mid = (start+end)/2;
  if(start < end){
    MergeSort(arr,start,mid);
    MergeSort(arr,mid+1,end);
    Merge(arr,start,mid,end);
  }
}
```

算法

1.顺序存储的线性表，每个元素都是不相同的整数元素，设计算法把奇数移动到偶数前面。

- 方法一，改一改快排

```
public static void Move(int []arr,int begin,int end){
  if(end <= begin)return;
  int middle = partition(arr,begin,end);
  Move(arr,begin,middle-1);
  Move(arr,middle+1 ,end);
}
public static int partition(int []arr,int begin ,int end){
  int key = arr[begin];
  while (begin<end){
    while (begin<end && arr[end]%2 == 0)end--;
    arr[begin] = arr[end];
    while (begin<end && arr[begin]%2 != 0)begin++;
    arr[end] = arr[begin];
  }
  arr[begin] = key;
  return begin;
}
```

- 方法二，进行一趟快速排序，时间复杂度 n 空间复杂度 1

```
public static void Move2(int []arr){
  int i = 0;
  int j = arr.length-1;
  int temp;
  while (i<j){
    while (i<j && arr[i]%2 !=0 )i++;
    while (i<j && arr[j]%2 ==0)j--;
    if(i<j){
      temp = arr[i];
      arr[i] = arr[j];
      arr[j] = temp;
    }
  }
}
```

2. 找出数组中第k小的元素

基于快排的划分操作（找出排好序的数组中第k个下标）

- 进行一趟快排
- 如果轴元素的下标=k，则找到了
- 如果轴元素下标 > k ,则说明在 轴元素 的半边继续找
- 如果轴元素下标 < k，则说明要在轴元素右半边继续找。

```
//找到排好序后下标为k的元素
```

```
public static int KthElem(int[] arr, int begin, int end, int k) {
  int key = arr[begin];
  int begin_temp = begin;
  int end_temp = end;
  while (begin < end) {
    while (begin < end && arr[end] > key) end--;
    arr[begin] = arr[end];
    while (begin < end && arr[begin] < key) begin++;
    arr[end] = arr[begin];
  }
  arr[begin] = key;

  if (begin == k) {
    return arr[begin];//如果当前中间节点下标为k，则返回
  } else if (begin > k - 1) {
    return KthElem(arr, begin_temp, begin - 1, k);
  } else {
    return KthElem(arr, begin + 1, end_temp, k);
  }
}
public static void main(String[] args) {
  int a[] = {0, 8, 7, 6, 5, 4, 3, 2, 1};
  int begin = 0;
  int end = a.length - 1;

  System.out.println(KthElem(a, begin, end, k - 1));//第k个元素的最后的下标为k-1

}
```

3. 荷兰国旗问题：设有一个仅由红、白、蓝三种颜色的条块组成的条块序列，编写时间复杂度为$O(n)$算法，使得这些条块按红、白、蓝的顺序排好，排成荷兰国旗

```
public static void FlagOfHolland(int []arr){
  int i = 0;
  int j = arr.length-1;
  int temp;
  //先把1全挪到最前面
  while (i<j){
    while (i<j && arr[j]!=1)j--;//找到第一个等于1的
    while (i<j && arr[i]==1)i++;//找到第一个不等于1的
    if(i<j){
      temp=arr[i];
      arr[i] = arr[j];
      arr[j] = temp;
    }
  }
  //再把3全挪到最后面
  j = arr.length-1;
  while (i<j){
```

```
    while (i<j && arr[j]==3)j--;//找到第一个不等于3的
    while (i<j && arr[i]!=3)i++;//找到第一个等于3的
    if(i<j){
      temp=arr[i];
      arr[i] = arr[j];
      arr[j] = temp;
    }
  }
}
```

- 另法，设立三个指针，j为工作指针表示当前扫描的元素，i以前的元素全为红色，k以后的元素全为蓝色。根据j所指元素的颜色，决定将其交换到序列的前部或尾部，初始时 i = 0;k = n-1;

```
public static void FlagOfHolland2(int []arr){
  int i = 0;
  int j = 0;
  int k = arr.length-1;
  int temp;
  while (j<=k){
    switch (arr[j]){
      case 1:{
        temp = arr[i];
        arr[i] = arr[j];
        arr[j] = temp;
        i++;j++;
        break;}
      case 2:j++;break;
      case 3:{
        temp = arr[j];
        arr[j] = arr[k];
        arr[k] = temp;
        k--;}
        //这里没有 j++ ： 如果交换后 arr[j]仍为3，则需要arr[j]再次交换
        //上面case1 不用考虑这个问题是因为，i,j是从同一方向出发的，j 扫描到1的时候，i,肯定不
是1
    }
  }
}
```