前言

- **时间仓促，整理的知识点，若有错，漏等，请见谅。手写部分，因时间不足，不再填写。**
- 修正版：前序中序唯一树确定；堆排序size-1调整；树部分应用补充；哈夫曼树的堆使用改进（原版自实现最小堆）；图论补充矩阵/表版，。
-

顺序表&&链表

- 顺序存储结构的插入，删除，查找算法

```
//顺序表存储结构
const int maxsize = 100;

struct Seqlist
{
    int data[maxsize]; //存储容量
    //int maxsize = 100;
    int last;//默认last指向空位置。，浴帘的为指向最后一个元素。
};
//类操作略，此后只写主要存储结构，不写类结构。类结构据题目需要自定义。

//按值查找(按序号查找稍微修改)
int search(Seqlist &arr, int n)
{
    int last = arr.last;
    for(int i = 0; i < last; i++)
    {
        if(data[i] == x) return i;
    }
    return x;
}

//插入数据。
bool Insert(Seqlist &arr, int index, int n)
{
    if(arr.last == maxsize-1) return false;
    if(index < 0 || index > maxsize-1) return false;
    for(int i = arr.last; i > index; i--)
    {
        arr.data[i] = data[i-1];
    }
    data[index] = x;
    arr.last++;
    return true;
}


//删除算法
bool remove(Seqlist &arr, int n, int &getNum)
{
    if(arr.last==0) return false;
    if(n < 1 || n > arr.last) return false;
    getNum = arr.data[n-1];//删除第n个,返回value
    for(int i = n; i < arr.last; i++)
        arr.data[i-1] = arr.data[i];
    arr.last--;
    return true;
}
```

```
//顺序表的合并操作等，略。太简单不考。考了就：
//并运算：从A中取元素，在B中检索Search，如果没有，插入B。
//交运算：从A中取元素，在B中检索，如果 不存在，删去remove。

/*
顺序表的优缺点：
可以取任意元素。
不好删除插入。
预先分配存储空间。
*/
```

- 链表的建立，插入，删除，查找算法

  代码如下：

```
//链表的存储结构
struct listNode
{
    int data; //存储数据
    listNode* next;
    listNode():next(nullptr){}
    listNode(int d):next(nullptr){data = d;}
};

//单链表类定义中，如果尾插入，可以设置一个表头指针，即虚拟指针first(dummy

//建立.尾插入版。此为虚拟头节点
void CreatList(listNode* dummy_root)
{
    int n;
    listNode* cur = dummy_root;
    while(cin>>n&&n!=-1)
    {
        listNode* tmp = new listNode(n);
        cur->next = tmp;
        cur = cur->next;  // cur=tmp;
    }
    //若有尾指针，可通过尾指针tail->next = tmp;
}

//建立.头插入版。
void CreatList(listNode*& root)
{
    int n;
    while(cin>>n&&n!=-1)
    {
        listNode* tmp = new listNode(n);
        tmp->next = root;
        root = tmp;
    }
}

//按值删除,尾插入版
void delNode(listNode* dummy_root, int val)
{
    listNode* cur = dummy_root->next;
    listNode* pre = dummy_root;
    while(cur!=nullptr)
    {
        if(cur->data==val)
        {
            pre->next = cur->next;
```

```
            delete cur;
        }
        else
        {
            pre = cur;
            cur = cur->next;
        }
    }
}

//按值删除，头插入版
void delNode(listNode*& root, int val)
{
    listNode* cur = root;
    listNode* pre = nullptr;
    while(cur!=nullptr)
    {
        if(cur->data==val)
        {
            if(cur==root)
                root = root->next;
            else
                pre->next = cur->next;
            delete cur;
            return;
        }
        else
        {
            pre = cur;
            cur = cur->next;
        }
    }
}

//查找，头尾类似
int search(listNode* dummy_root, int val)
{
    listNode* cur = dummy_root->next;
    while(cur!=nullptr)
    {
        if(cur->data == val) return 1;
        else cur = cur->next;
    }
    return -1;
}


//输出所有数据，表头结点版
void Print(listNode* first)
{
    listNode* cur = first->next;
    while(cur!=0)//cur!=nullptr
    {
        cout<<cur->data;
        cur = cur->next;
    }
}

//单个结点插入只需再创建节点的算法上更改while的内容。while(cur!=nullptr)
//尾插入时间效率为O(n)，头插入时间效率为O(1),若要提高尾插入效率，可设置一个tail指针，记录最后一个结点。
```

- 循环链表的建立，插入，删除

代码如下:

```
```

- 双端链表的建立，插入，删除

  代码如下:

```
```

栈与队列

- 栈的特点:
  - 后进先出(`LIFO`)，插入删除O(1)
- 栈的存储结构

```
//一切从简，只写int型
//顺序存储结构。即：连续型。
class Stack
{
private:
    int *elem; //栈数组。
    int top; //栈顶
    int maxsize; //最大值
    void overflow();//溢出处理。
public:
    //常见操作：
    Stack();//初始化。
    ~Stack();//析构
    void push();
    bool pop(int &x);
    bool top(int &x); //获取栈顶元素。
    bool isEmpty(){return top == -1; }
    bool isFull(){return top == maxsize-1;}
};

//链式栈
struct Node
{
public:
    int data;
    Node* next;
    Node();//构造函数。
};
class Stack
{
public:
    Node* top;//栈顶指针。用头插法。
    //...其余操作同。
};

//双端栈*,优化版，采用数组做指针。
class Stack
{
private:
    int *elem;
    int maxsize;
    int but[2]; //0,1号栈底
```

```
    int top[2]; //0，1号栈顶
    //主要操作：判空，判满
    Stack();//top[0]=but[0]=-1;top[1]=but[1]=maxsize;
    bool isFull(); //top[0]+1==top[1]
    bool isEmpty();  //but[0]==top[0] || but[1]==top[1];
};

//index为选择插入的栈，栈1还是栈2
bool Stack::push(int x, int index)
{
    if(top[0]+1==top[1]) return false;
    if(index==0) top[0]++;
    else top[1]++;
    //[++top]=x;
    elem[top[index]] = x;
    return 1;
}

bool Stack::pop(int &x, int index)
{
    if(top[index] == but[index]) return false; //栈空
    x = elem[top[index]];
    if(index == 0) top[index]--;else top[index]++;
    return true;
}

//其余操作略，看实验，无则自己手写一遍，不难。关键在两个极端点的判断。top==-1,top==maxsize-1;
```

- 进制转换（**先进栈的是低位。**），括号匹配（左括号进栈，右括号则，判断栈顶是否左括号，否则表示不匹配。），逆波兰表达式等用栈实现。

- 队列的特点：`FIFO(先进先出)`，插入删除O(1);
- 队列的存储结构

```
//顺序栈。用数组存储
//循环队列，解决假溢出，即rear=front=maxsize-1,但队列空
class Queue
{
private:
    int rear, front; //队尾，队头,队头指向永远为空。
    int *elem; //队数组
    int maxsize;
public:
    //常见操作
    Queue(){rear=front=0;}//不是-1，约定用于循环队列使用
    ~Queue();
    bool push(int x);
    bool pop(int &x);
    bool front(int &x);
    bool isEmpty(){}//改进版，解决假溢出
    bool isFull(){}
    int size(){return (rear-front+maxsize)%maxsize;/*+maxsize避免rear<front*/}
};

//循环队列解决：空跟满不分问题：即：当rear绕一圈到front的后边，和rear在front前边，的空和满无法区分。
//3种解法，1:增加计数器（即长度length）,2:设置布尔变量判空，判满;
/*最优解法：
少用一个数组格，当(rear+1)%maxsize==front;满
当 rear==front;空
初始化：rear = front=0
队头队尾进1：front = (front+1)%maxsize;//rear同理.*/

//入队
```

```
bool push(int x)
{
    if((rear+1)%maxsize==front) return false;//队满
    rear = (rear+1)%maxsize;
    elem[rear] = x;
    return true;
}

//出队
bool pop(int &x)
{
    if(rear==front) return false; //队空
    front = (front+1)%maxsize;
    x = elem[front];
    return true;
}


//链式存储结构
struct Node
{
private:
    int data;
    Node* next;
public:
    //其余操作一样，只是不需要顺序表的循环队列。
};
//若要写链式循环队列，则，只需要一个rear指针即可。
//rear指针村的永远是队尾元素，其下一个指向永远是队头元素。例子算法：

bool pop(int &x)
{
    if(rear==0) return false; //空
    Node* cur = rear->next;
    x = cur->data;
    rear->next = cur->next;//即使只有一个元素，自身成环也可运行。
    delete cur;
    return true;
}

bool push(int x)
{
    Node* cur = new Node(x);
    if(rear == nullptr)
    {
        rear = cur;
        cur->next = cur;
    }
    else
    {
        cur->next = rear->next;//新的队尾，链接队头
        rear->next = cur;
        rear = cur;
    }
    return true;
}
```

- 双端队列的使用

```

```

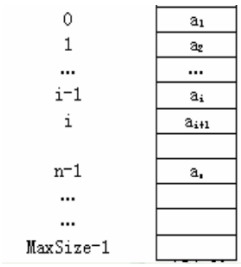
- 队列的应用：

- 递归算法的设计思想

```
//第一步，确定递归参数
//第二步，设置递归终止条件
//第三步，将递归拆分多个子递归问题。
```

---

s矩阵的存储与地址计算等（全部手工）

- 稀疏矩阵。时间复杂度，空间复杂度具体计算。
  - 

  - 如何求一维数组第i个元素起始地址？`Loc(ai)=?` $Loc(a_i) = Loc(a_1) + (i-1) *$ 每个元素所需空间（0号地址存a_1）
  - 二维数组（矩阵）`A[m][n]`：
    - 1）m个具n个元素的线性表组成（即一般看法，邻接表一样横着算的）
    - 2）n个具m个元素的线性表组成（即竖着看的）
    - 如何求二维数组`A[m][n]的第ij元素的起始地址:` $Loc(a_{ij})$

①按行主序存储：

$$Loc(a_{ij}) = Loc(a_{00}) + (i * n + j) *$$ 每个元素所需空间

特点：

元素在行之间是连续存储的。访问相邻元素的时间性能较好。

②按列主序存储：

$$Loc(a_{ij}) = Loc(a_{00}) + (j * n + i) *$$ 每个元素所需空间

特点：

元素在列之间是连续存储的。对于矩阵运算（如矩阵乘法）可能有性能优势。

  - n维矩阵：`A[m1][m2][m3]....[mn]`的第$i_1i_2i_3...i_n$元素起始地址$Loc(a_{i_1i_2i_3...i_n}) =?$
    - 按不同维度存储。略（太多了

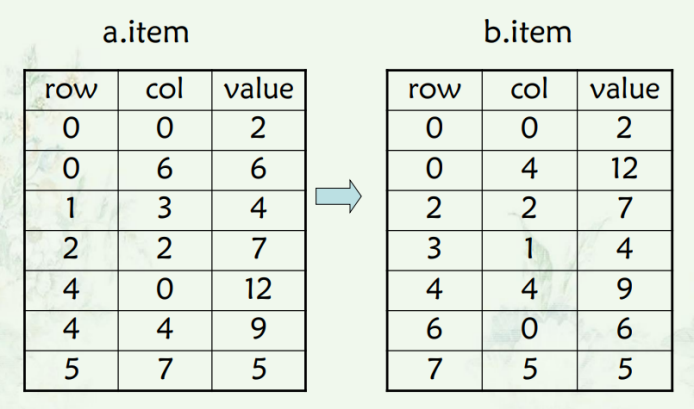
①按行：

$$Loc(a_{i_1i_2i_3...i_n}) = Loc(a_{00000...}) + (i_1 * m2 * m3... + i_2 * m3.... + i_n) *$$ 每个元素空间大小。

  - 特殊矩阵
    - 对称矩阵的存储方法&计算：一维存储，存储下三角，则$Loc(a_{ij}) = Loc(a_{00} + (i * (i+1)/2 + j) * d)$ 存储上三角，则：$Loc(a_{ij}) = Loc(a_{00} + (j * (j+1)/2 + i) * d)$
    - 三角矩阵：考试手工吧。尽力了。
  - **稀疏矩阵：很多元素为0，少部分不为0**
    - 存储结构：三元组表，行列值。

存储单元编号	行	列	值
0	4	5	4
1	1	2	2
2	3	2	1
3	3	4	3
4	4	5	-1
5			
.	.	..	.
.	..	.	.

- 稀疏矩阵的转置（即，因为按行主序，所以行保持从小到大有序）：



a.item

row	col	value
0	0	2
0	6	6
1	3	4
2	2	7
4	0	12
4	4	9
5	7	5

b.item

row	col	value
0	0	2
0	4	12
2	2	7
3	1	4
4	4	9
6	0	6
7	5	5

- 稀疏矩阵的快速转置：
  - 开一个列数长度的表，统计每一列的非零元素个数(**开桶计数**)，然后存入长度表`rnum`。
  - 开一个列数长度的表，统计每一列的非零元素的起始元素地址（**计算公式：每一列的首个非零元素起始地址=上一个非零元素个数+起始地址。）**，然后存入起始表`rstart`
  - 开始依次扫描当前`a.item`的三元组表，将扫描到的每一个列元素，存入`rstart`记录的位置，然后位置+1.
  -

k	0	1	2	3	4	5	6	7
rnum[k]	2	0	1	1	1	0	1	1
rstart[k]	0	2	2	3	4	5	5	6

- 广义表：$LS=(d_1,d_2\dots,d_n)$,其中d_i是不可再分的元素（原子）
  - 长度，表头，表尾，深度，空表计算：

$$eg1:A=(a,(),(b,(a,b)))$$

深度为3（即最大深度，先第一个a，然后第一个b，然后b内的下一个(a,b),即3层嵌套

长度为3，因为最外的广义表有3个元素：$a,(),(b,(a,b))$

表头是原子a,深度为0；

表尾是$((),(b,(a,b)))$,深度为3，长度为2

  - $$eg2:B=((a),b,c)$$

深度为$2:((a))$最深，长度为3（三个元素）

表头为(a)，深度为1，长度为1

表尾是(b,c)，深度为1，长度为2

空表的表头，表尾都是空表

取表头$HEAD(LS)$,取表尾$TAIL(LS)\dots$（求上述功能操作，递归）

树

- 树的基本概念（本次课程，约定树根为第一层）：
  - **树的深度：空树为0；根节点的深度为 1(第一层的节点深度为 1,即根)。**
  - 树的高度：数值上=深度。
  - 结点的度：多少个孩子。
  - 树的度：所有节点中，拥有最多孩子的结点的度，即$max(a_1,a_2,\dots)$
  - 祖先结点：直接前驱（即，路径上的都是）
  - 森林：n棵树的集合（n>=0,包含空森林）
- 树的表达方式：①树形②集合图③凹入表④广义表
- 树的基本术语以及计算方式：

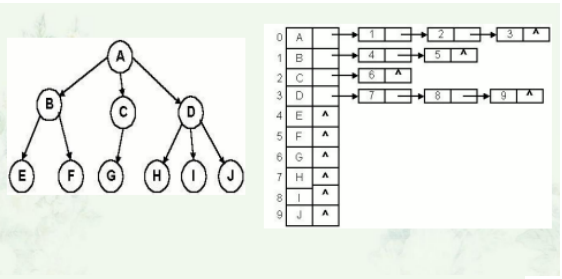
树的结点总数n与分枝数m与所有节点的度数之和$\sum deg(d_i)$关系：

$$m = n - 1(\text{欧拉公式}) \sum deg(d_i) = 2 * m = 2 * (n - 1)$$

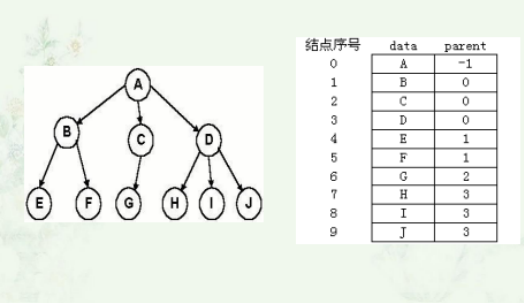
度为k的树中，第i层最多有?个计算结点：

$$(\text{即全部都为}k\text{结点，为最大})k^{i-1}\text{个}(i \geq 1, i == 1\text{根节点})$$

- 树的存储结构：1）普通链式结构。2）孩子链式结构。3）双亲表示结构
  - 孩子链式结构(可能考手工，见手工模拟部分)：



  - 双亲表示结构(可能考手工，见手工模拟部分)：



- 树，森林与二叉树互为转换（可能考手工，见手工模拟部分）
  - 前思路：①在兄弟之间加线连接。②对于每一个结点，只保留最左的孩子，然后将**原先**与孩子的连线抹除（注意：①的线不属于**原先**）③整理树型。**（注：一棵树，若像二叉树，仍需转换，不可直接说明为二叉树。）**
  - 后思路：①将每个节点的**左孩子**的**所有连续右孩子（兄弟）**重新与该节点相连。②将原本左孩子的连续右孩子之间的线抹除。③调整树形。（左孩子的右孩子为兄弟）
  - **森林的转换同理。**

- **二叉树的性质**：

$$(1)\text{二叉树第}i\text{层最多有?个结点。}2^{i-1}(i \geq 1)$$

$$(2)\text{深度为}h\text{的二叉树最多有?个结点。}2^h - 1(i \geq 1)$$

$$(3)\text{具有}n\text{个结点的完全二叉树的深度为：}log_2 n + 1\text{（假设为4个节点，计算。快速推导公式）}$$

$$(4)\text{对于具有}n\text{个结点的完全二叉树，对其结点进行编号，则对于编号为}i\text{的结点，有：}$$

$$\text{①结点从0开始编号：(双亲编号，左孩子，右孩子编号，存在条件)}$$

$$\text{②结点从1开始编号：(双亲编号，左孩子，右孩子编号，存在条件)} - > \text{具体见堆部分讲解}$$

$$(5)\text{对于任意一棵非空的二叉树，叶子总数}n_0\text{个，度为2的结点总数}n_2\text{个，度为1的结点总数为}n_{个}\text{，则}n_0, n_1, n_2\text{的关系：}$$

$$n - 1 = 2 * n_2 + 1 * n_3 + 0 * n_1(\sum \text{度} = \text{边} * 2)$$

$$\text{所以：}n2 = n - n1 - n0$$

$$\cdots\cdots$$

- 二叉树的存储结构：1）顺序存储（数组）。2）链式存储

```
//顺序存储
const int MAXSIZE = 1e5;
int BTree[MAXSIZE]; //根存0，则左右孩子公式：2i+1,2i+2; 根存1，则左右孩子公式:2i,2i+1

//链式存储
struct BTNode
{
    int data;
    BTNode *left, *right; //左右孩子
};
```

- 二叉树的遍历实现：`前序，中序，后序，层序`

```
//递归部分
//前序：中左右
void preOrder(BTNode* root)
{
    if(root==NULL) return;//终止条件
    cout<<root->data<<" ";//中
    preOrder(root->left);//左
    preOrder(root->right);//右
}
//中序：左中右
void midOrder(BTNode* root)
{
    if(root==NULL) return;
    midOrder(root->left);
    cout<<root->data<<" ";
    midOrder(root->right);
}
//后序：左右中
void postOrder(BTNode* root)
{
    if(root==NULL) return;
    midOrder(root->left);
    midOrder(root->right);
    cout<<root->data<<" ";
}

//非递归部分
void levelOrder(BTNode* root)
{
    if(root==NULL) return;
    queue<BTNode*> q;
    q.push(root);
    while(!q.empty())
    {
        BTNode* p = q.front();
        q.pop();
        cout<<p->data;
        if(p->left!=NULL) q.push(p->left);
        if(p->right!=NULL) q.push(p->right);
    }
}
//栈非递归模拟递归
void staPreOrder(BTNode* p)
{
    stack<BTNode*> s1;
    stack<int> s2
    BTNode* t = root;
    while(!s1.empty()||t!=0) //t!=0入while用
    {
        //将左孩子全部入。
        while(t!=0)
        {
            //if(t!=null)cout<<t->data<<" "//前序遍历
            s1.push(t);
            s2.push(0);//第一进栈标志，表示左子树.
            t = t->left; //左孩子
        }
        if(!s1.empty())
        {
            t = s1.top();//最底的左孩子
            int flag = s2.top(); //标志
            s1.pop();s2.pop();
```

```
            if(flag==1) //说明该节点是第二进栈结点，也就是右子树
            {
                //if(t!=null)cout<<t->data<<" ";//后序遍历处
                t = 0; //访问过，t置空，避免全部访问完，退不出大while
            }
            else
            {
                //if(t!=null)cout<<t->data<<" ";//中序遍历处
                s1.push(t);
                s2.push(1); //结点第二次进栈，即取右孩子
                t = t->right; //到右之后，又会跳到顶部循环，遍历右孩子的左孩子。
            }
        }
    }
}
```

- 二叉树遍历的应用：
  - 交换左右子树

```
void PreOrderExg(BTNode* root) //自顶向下交换，自底向上交换也可。后序遍历，交换一下顺序，把交换放最后
{
    if(root==nullptr) return;
    BTNode* tmp = root->left;
    root->left = root->right;
    root->right = tmp;
    PreOrderExg(root->left);
    PreOrderExg(root->right);
}
void PostOrderExg(BTNode* root)
{
    if(root==nullptr) return;
    PostOrderExg(root->left);
    PostOrderExg(root->right);

    BTNode* tmp = root->left;
    root->left = root->right;
    root->right = tmp;
}
```

  - 求叶子数量，求节点数量（略，这个只要判空。）

```
void Leaves(BTNode* root, int& counter) //其实基本每个遍历都行，只要判空即可。这里写PreOrder
{
    if(root==NULL) return;
    if(!root->left&&!root->right) counter++;
    Leaves(root->left, counter);
    Leaves(root->right, counter);
}//好几种写counter的方法。也可以递归return left+right;
```

  - 二叉树的建立算法（递归）

```
BTNode* Creat()//前序
{
    int num; //按需修改。此处默认-1空子树
    cin>>num;
    if(num == -1) return NULL;
    BTNode* cur = new BTNode(num);
    BTNode* cur->left = Creat(); //左子树
    BTNode* cur->right = Creat(); //右子树
    return cur;
```

```
}

BTNode* Creat()//中序
{
    int num; //按需修改。此处默认-1空子树
    cin>>num;
    if(num == -1) return NULL;
    BTNode* q = Creat(); //左子树
    BTNode* cur = new BTNode(num);
    cur->left = q; //连接,不可cur->left直接，因为父亲还是空
    cur->right = Creat(); //右子树
    return cur;
}

BTNode* Creat()//后序
{
    int num; //按需修改。此处默认-1空子树
    cin>>num;
    if(num == -1) return NULL;
    BTNode* q1 = Creat(); //左子树
    BTNode* q2 = Creat(); //右子树
    BTNode* cur = new BTNode(num);
    cur->left = q1;
    cur->right = q2;
    return cur;
}

//层次建立
BTNode* Creat()
{
    queue<BTNode* > q;
    int num;
    cin>>num;
    if(num==-1) return nullptr;//空树
    BTNode* root = new BTNode(num);
    q.push(root);
    while(!q.empty())
    {
        BTNode* now = q.front();
        q.pop();
        cin>>num;
        if(num == -1)
            now->left = NULL;
        else
        {
            BTNode* tmp = new BTNode(num);
            now->left = tmp;
            q.push(now->left);
        }
        cin>>num;
        if(num==-1)
            now->right= NULL;
        else
        {
            BTNode* tmp = new BTNode(num);
            now->right = tmp;
            q.push(now->right);
        }
    }
    return root;
}
```

- 二叉树-》广义表&&广义表-》二叉树：课本197~203

```
```

```
//二叉树到广义表，广义表到二叉树
//1
void BTtoTable(BTNode* root)
{
    if(root==nullptr) return;
    if(root)
    {
        cout<<root->data;
        if(root->l||root->r)
        {
            cout<<"(";
            BTtoTable(root->l);
            if(root->r)
            {
                cout<<",";
                BTtoTable(root->r);
            }
            cout<<")";
        }
    }
}
//2
BTNode* TabletoBT()
{
    stack<BTNode*> st;
    BTNode* root;
    BTNode* used;
    BTNode* cur;
    char ch;//字符
    int lor;//左右孩子判定符
    while(ch!='.') //自选终止字符
    {
        switch(ch)
        {
            case'(':
                st.push(used);lor=1;//左孩子下一个
                break;
            case',':
                lor=2;//右孩子下一个
                break;
            case')':
                st.pop();//处理完该点孩子
                break;
            default:
                used = new BTNode(ch);//生成value
                if(root==0) root=used;
                else//非头结点
                {
                    if(k==1)
                    {
                        cur = st.top();
                        cur->l = used;
                    }
                    else
                    {
                        cur = st.top();
                        cur->r = used;
                    }
                }
        }
    }
    return root;
}
```

- 求二叉树深度：层序遍历或后序遍历得到左右子树最高的一个+1.（空树为0）

```
int high(BTNode* root)
{
    if(root == 0) return 0;
    else
    {
        return max(high(root->l),high(root->r))+1;
    }
}
int high(BTNode* root)
{
    if(root==0) return 0;
    queue<pair<BTNode*,int> >q;
    q.push({root,1});
    int high = 1;
    while(!q.empty())
    {
        BTNode* cur = q.front().first;
        int h = q.front().second;
        q.pop();
        high = high > h ? high:h;
        if(cur->l) q.push({cur->l,h+1});
        if(cur->r) q.push({cur->r,h+1});
    }
    return high;
}
```

- 根据前序&中序建立唯一二叉树&&&根据中序&后序建立唯一二叉树。
  - 原理：中序找到根节点，根节点左边是左孩子**个数**，再根据左孩子个数，对前序/后序的根节点计算下一个根节点范围。注意细节，**前序和后序的索引范围**（因为都是`leftnum`，所以前序和后序略有不同）

```
//存储结构
class treeNode
{
public:
    int value;
    treeNode* left;
    treeNode* right;
    treeNode() :left(NULL), right(NULL) {}
};
int mid[1e6],lst[1e6];
//中序后序
int midFind(int root)
{
    for (int i = 0; i < mid.size(); i++)
        if (root == mid[i]) return i;
}
treeNode* builtTree(int l, int r, int ml, int mr)
{
    if (l > r) return NULL; // 寻找结束

    //后序找出根节点
    int root = lst[r];
    treeNode* cur = new treeNode();
    cur->value = root;

    //找到中序中，根节点的索引，划分左右子树
    int index = midFind(root);
    int numleft = index - ml;
```

```
    cur->left = builtTree(l,l+numleft-1, ml, index-1);
    cur->right = builtTree(l+numleft, r-1, index+1, mr);

    return cur;
}

//前序中序
int midFind(int root) {
    for (int i = 0; i < mid.size(); i++)
        if (root == mid[i]) return i;
}

// 修改参数，从中序后序改为前序中序
TreeNode* buildTree(int pl, int pr, int ml, int mr) {
    if (pl > pr) return NULL; // 寻找结束

    // 前序找出根节点
    int root = pre[pl];
    TreeNode* cur = new TreeNode();
    cur->value = root;

    // 找到中序中，根节点的索引，划分左右子树
    int index = midFind(root);
    int numLeft = index - ml;
    cur->left = buildTree(pl + 1, pl + numLeft-1, ml, index - 1);
    cur->right = buildTree(pl + numLeft + 1, pr, index + 1, mr);

    return cur;
}
```

- 堆部分：（此处为最小堆部分）
  - 每次插入堆尾后，向上调整
  - 每次删除堆头后，向下调整
  - 向上调整：循环由son或father决定都可。son决定，则终止条件为son>1(若使用0号单元则，son>0)或father>=1(若使用0号单元则，father>=0)。原理是：每次交换后，son和father会向上调整一层，调到顶层，会有son==1, 或者最后一次调整，会有father==1。

```
class MinHeap
{
private:
    int size;
    int last = 0;
    int *arr;
    .....//构造函数等略。
}
  //向下调整
void siftDown(int arr[], int index, int last)
{
    int father = index, son = father*2;
    int temp = arr[father];
    while(son <= last)//用son，用father会多进行一轮。用father<last-1不准确。
    {
      if(son < last && arr[son] > arr[son+1]) son++;
      if(temp < arr[son]) break;
      else
      {
        arr[father] = arr[son];
        father = son;
        son = father*2;
      }
    arr[father] = temp;
```

```
}
  //向上调整
void siftUp(int index)
{
    int son = index; father = index/2; //不需要-1.因从1号位开始
    int temp = arr[son];
    while(father >= 1)//若son则son>1.因为已经到底
    {
      if(temp<arr[father])
      {
        arr[son] = arr[father];
        son = father;
        father = son/2;
      }else break; //调整完毕
      arr[son] = temp; //调整完毕
    }
}
  //插入（建堆）
bool insert(int x)
{
    if(last==size) return false;//满
    arr[++last] = x;
    siftUp(last);
    return true;
}
  //删除堆头
bool del()
{
      if(last == 0) return false;
      arr[1] = arr[last];
      last--;
      siftDown(arr, 1, last);
}

void MinHeap(int hp[], int n) //初始化最小非空堆
{
    arr = new int[n+1];
    size = n+1;//多开一个位置n
    for(int i = 1; i <= n; i++)
        arr[i] = hp[i-1]; //传入的是从0号单元开始的。
    last = n;
    int cur = last/2;//找最后一个非叶子结点
    while(cur>=1)
    {
        siftDown(arr, cur, last);
        cur--;
    }
}
/*
----------------------------------------------------
----------------------------------------------------
----------------------------------------------------
*/
class MaxHeap
{
public:
    int *heap;//堆
    int last = 0;
    int maxsize = 1e6;
};

void MaxHeap::built(int w[], int n)
{
    if(n==0) return ;
    heap = new int[n+1]; //0号空间不用
```

```
    for(int i = 0;i < n; i++)
        heap[++last] = w[i];
    //自下向上调整堆
    int curIndex = n/2;
    while(curIndex>0) // 不能==0，0为空单元
    {
        siftDown(cur, last);
        cur--;
    }
}

bool MaxHeap::insertVal()
{
    if(last==maxsize) return false;
    int num;
    cin>>num;
    heap[++last] = num;
    siftUp(last);
    return true;
}
void MaxHeap::siftDown(int faIndex, int last)
{
    int father = faIndex, son = father*2;
    //heap[0] = heap[father]; //暂存单元。
    int temp = heap[father];
    while(son <= last) //用son！！！
    {
        if(son<last&&heap[son]<heap[son+1]) //大的上
            son++;
        if(temp > heap[son]) break;//注意顺序，先上面的，找到大儿子在判断
        heap[father] = heap[son];
        father = son;
        son = father*2;
    }
    heap[father] = temp;
}

void MaxHeap::siftUp(int index)
{
    int son = index, father = index/2;
    int temp = heap[son];
    while(father>=1) //son则>1,反正，也可以认为，谁的边界归谁管.
    {
        if(heap[father] > temp) break;
        heap[son] = heap[father];
        son = father;
        father = son/2;
    }
    heap[son] = temp;
}

bool remove()
{
    if(last==0) return false;
    heap[1] = heap[last];
    last--;
    siftDown(1);
    return true;
}
```

- 哈夫曼树（存储结构与浴帘算法略有不同。）
    - 建立，编码，解码，遍历算法

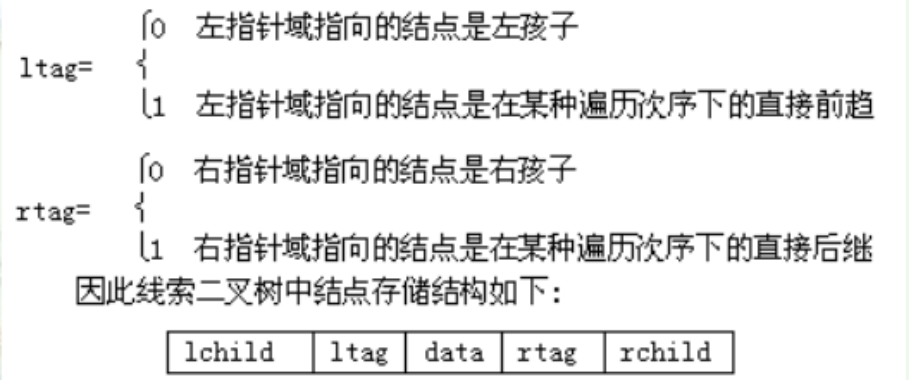
```
//哈夫曼存储结构
struct HuffmanNode
{
    int data;
    HuffmanNode* l, *r, *parent;
};
class HuffmanTree
{
private:
    HuffmanNode *root;
    int number;
public:
    .....
};
//建立
struct Cmp
{
    bool operator()(const HuffmanNode* p1, const HuffmanNode* p2)
        return p1->data > p2->data; //小顶堆
};

void HuffmanTree::HuffmanTree(int w[], int n)
{
    priority_queue<HuffmanNode*,vector<HuffmanNode* >, Cmp> hp;
    HuffmanNode *first, *second, *parent;
    for(int i = 0; i < n; i++)
    {
        HuffmanNode* p;
        p = new HuffmanNode;
        p->data = w[i];
        hp.push(p);//最小堆按data将指针搭建最小堆
    }
    //开始建树
    for(int i = 0; i < n-1; i++) //只需要进行n-1次，即每次取出两个，放回一个，相当于每次-1.直到只剩最后一个结点root
    {
        first = hp.top(); //第一小
        hp.pop();
        second = hp.top(); //第二小
        hp.pop();
        parent->l = first;parent->r = second;
        parent->data = first->data+second->data;
        hp.push(parent);
    }
    root = hp.top();//根。
    hp.pop();
}
//编码算法
void HuffmanTree::Encoding(HuffmanNode* node,string s, vector&<pair<int,string>>vec)
{
    if(node==NULL) return;
    if(node->l==NULL&&node->r==NULL) vec.push_back(pair{node->data, s});
    else
    {
        Encoding(node->l,s+'0',vec);
        Encoding(node->r,s+'1',vec);
    }
    return ;
}
//解码算法，计算值。若为字母，可修改为如下。按需修改即可
void HuffmanTree::Decoding(string BitCode,int & num)
{
    //string decodeText;
    HuffmanNode* cur = root; //从根开始
```

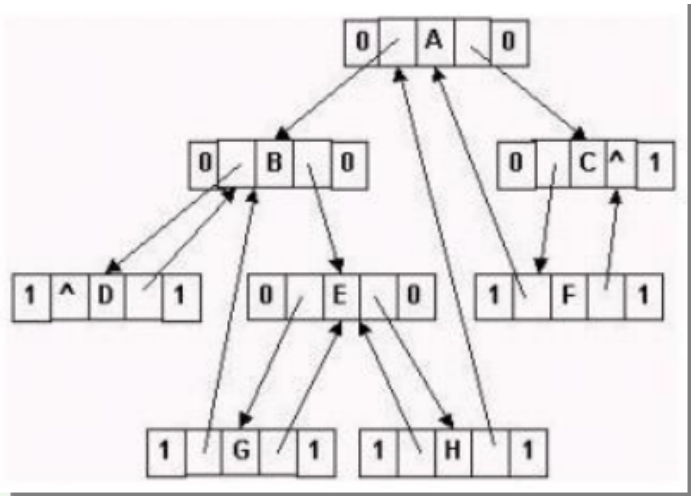
```
    for(char bit: BitCode) //(int i = 0; i < BitCode.size(); i++)
    {
        if(bit == '0' && cur != nullptr)
            cur = cur->l;
        else if(bit == '1' && cur != nullptr)
            cur = cur->r;

        if(cur != nullptr && cur->l == nullptr && cur->r == nullptr)
        {
            //decodeText += cur->data;//字母,翻译文本
            num+=cur->data;
            cur = root; //重新开始。
        }
    }
    //return decodeText;
}

//哈夫曼树的遍历得到所有编码。
void HuffmanTree::GetBitCode(HuffmanNode* root, string Bitcode, vector<string>& vec)
{
    if(root->l==nullptr&&root->r==nullptr) vec.push_back(Bitcode); //不必单项判空，由于哈夫曼要么叶子，要么二叉
    else
    {
        GetBitCode(root->l, Bitcode+'0', vec);
        GetBitCode(root->r, Bitcode+'1', vec);
    }
    return;
}
bool cmp(string a, string b)
{
    return a.length() > b.length();//从大到小
}
//打印获得的哈夫曼树编码，从长到短打印
void PrintBitCode(vector<string>& vec)
{
    //先sort一下
    sort(vec.begin(), vec.end()); //可自定义cmp, or:std::greater<string>();
    for(int i = 0; i < vec.size(); i++)
        cout<<vec[i]<<endl;
}
```

- *线索二叉树（只需手工）*
  - 在二叉树中，从任意节点出发只能找到该结点左右孩子，而在一般情况下而无法直接找到该节点在**某种遍历**的直接前驱和直接后继，而且，具有n格结点的二叉链表，必存在n+1个**空指针域**
  - 线索二叉树：在每个结点上增加两个线索标志域，一个是左线索（`ltag`），一个是右线索(`rtag`)
  - 

  - 例子：中序遍历：



- 更多具体可看我的 github：Vajackye_Codespace/数据结构课程/树.md at Vajackye · Vajackye/Vajackye_Codespace (github.com)

集合与搜索

- 存储结构：

```
//顺序存储
struct Rec
{
    int key;
    //..
} S[n+1];//元素存放在S[1]-S[n]

//链表存储
struct LinkNode
{
    int data;
    LinkNode* next;
};

//索引表（依稀记得浴帘说不考?）
//ASL=查找索引平均查找长度+查找顺序表平均查找长度。(即1/nΣ+1/kΣ)
```

- 二分查找算法的实现（递归，非递归）

```
//普通搜索算法（监视哨改进，非i<=n型）
int Search(Rec S[], int x, int n)
{
    //0号监视哨
    S[0].key = x;
    int i = n;//从尾到头扫描
    while(S[i].key!=x)i--;
    if(i==0) return 0;
    else return i;
}//缺陷是O(n)，当遇到有序序列时，一般有最坏情况。


//二分有序查找
//递归式
int binarySearch(Rec S[], int left, int right, int x)
{
    if(left > right) return 0;//false to check
    int mid = left + (right-left)/2;
    if(x == S[mid].key) return mid;
```

```
    else if(x > S[mid].key) return binarySearch(Rec S[], mid+1, right, x);
    else return binarySearch(Rec S[], left, mid-1, x);
}

//非递归式
int binarySearch(Rec S[], int right, int x)
{
    int l = 1, r = right;
    int mid;
    while(l<=r)
    {
        mid = l+(r-l)/2;
        if(x == S[mid].key)
            return mid;
        else if(x > S[mid].key)
            l = mid+1;
        else
            r = mid-1;
    }
    return -1;
}
```

- 平均查找长度（表长为n的二分查找判定树和含有n个结点的完全二叉树的深度相同）：

$$\text{假设} n=2^h-1 \text{并且查找概率相等（落在每个点的概率）：}$$

$$ASL_{bs}=1/n\sum_{i=1}^{n}C_i=1/n\sum_{j=1}^{h}j\times 2^{(j-1)}=\frac{n+1}{n}log_2(n+1)-1$$

$$n>50\text{时，近似看成最后一个结果}$$

$$\text{即：}ASL=\text{每个元素比较之和}/\text{总元素个数}$$

- 并查集(是一种树形结构。)

```
const int Maxsize = 100;
int parent[Maxsize]; //双亲数组
int n; //集合元素个数

//浴帘版：
//初始化操作
void init()
{
    for(int i = 0; i < n; i++)
        parent[i] = -1;
}
//归属集合查找算法
int find(int x)
{
    while(parent[x]>=0) x=parent[x];
    return x;//找到祖先
}
//合并成一棵树
int merge(int x, int y)
{
    parent[x]=y;//将x的祖先挂到y下面。效率很差。
}

//实用版
//初始化操作
void init()
{
    for(int i = 0; i < n; i++)
        parent[i]=i;
}
int find(int x)
```

```
{
    return parent[x]==x?x:find(parent[x]);
}
int merge(int x, int y)
{
    x = find(x); y = find(y);
    parent[x] = parent[y];//y成为x祖先。
}
```

- 二叉搜索（排序）树
  - 二叉排序树（即中序遍历结果为有序序列），即二叉查找树。二叉排序树不允许有相同元素（即二叉排序树本质是：**集合**）
  - 二叉排序树**默认**左小右大。即中序遍历结果为从小到大。（也可自定义为从大到小）
  - 二叉排序树有建树，插入，删除，查找。

```
//存储结构
struct BSTNode
{
    bool flag; //假删除。真删除不好写，要链接整棵树
    int data;
    BSTNode* left, *right;
    BST():left(NULL),right(NULL){flag = false;}
    BST(int x):left(NULL),right(NULL){data = x;flag = false;}
};

//类
class BST
{
private:
    BSTNode* root;
public:
    //...
    BSTNode* search();
    bool insert();
    bool Creat();
    bool remove();
}
```

  - 查找，插入，删除，建立算法

```
//查找，递归式,假设有类
BSTNode* BST::search(int x, BSTNode* node)
{
    if(node==0) return node;//nullptr
    if(node->data == x)return node;
    else if(node->data > x) return search(x, node->left);
    else return search(x, node->right);
}
//非递归式
BSTNode* BST::search(int x)
{
    BSTNode* cur = root;
    while(cur!=0)
    {
        if(cur->data==x) return cur;
        else if(cur->data > x) cur = cur->left;
        else cur = cur->right;
    }
}

//删除
```

```
//插入算法
//非递归
int BST::insert(int x)
{
    BSTNode* cur, *father, *tmp;
    cur = root;
    father = 0;
    while(cur!=0)
    {
        if(cur->data==x) return -1;//有元素
        else if(cur->data < x)
        {
            father = cur;
            cur = cur->right;
        }
        else
        {
            father = cur;
            cur = cur->left;
        }
    }
    tmp = new BSTNode(x);
    if(father==0) root = tmp;
    else if(x > father->data) father->right = tmp;
    else father->left = tmp;
    return 1;//插入成功。
}
//递归式
int BST::insert(int x, BSTNode *&root)
{
    if(root==0)
    {
        BSTNode* cur = new (x);//无p->l=p->r=null,构造函数已写。
        root = cur;
        return 1;
    }
    if(x<root->data) return insert(x, root->left);
    else return insert(x, root->right);
    return -1; //未找到。
}

//二叉树排序树的建立算法。
//非递归和递归式都由插入实现
void BST::creat()
{
    root = 0;//可在构造函数实现
    int x;
    while(cin>>x&&x!=-1)
    {
        insert(x);//非递归式
        //insert(x,root);//递归式
    }
    return;
}

//二叉排序树的删除算法
//非递归式
bool BST::remove(int x)
{
    BSTNode* cur = root;
    while(cur!=0)
    {
        if(cur->data == x)
        {
            cur->flag = true;
```

```
            return true;
        }//删除标志。添加这个，则上述的插入等算法也需要更改。视题目而定。考也只考一个部分。
        else if(cur->data > x) cur = cur->left;
        else cur = cur->right;
    }
    return false;
}
//递归式
bool BST::remove(int x, BST* root)
{
    if(root==nullptr) return false;
    if(root->data == x)
    {
        root->flag = true;
        return true;
    }
    else if(root->data > x)
    {
        return remove(x, root->left);
    }
    else
    {
        return remove(x, root->right);
    }
}
```

- 查找，插入，删除的性能分析：
  - 删除3种情况：1）叶子节点，2）只有左/右子树的结点，3）双树结点。
    - 1）略。
    - 2）被删除的结点只有左子树（右）：则将被删除结点的双亲指向其子树。
    - 3）被删除的是双树结点：将该二叉排序树以中序遍历的方式，得到有序数组，将待删除的结点的前驱结点（或后继节点）的值复制到待删除的结点处，再将前驱结点（或后继节点）删除。`[原理：前驱结点为最大的左子树值（后继节点为最小的右子树值，）它们都是最多只有一颗左（右）子树。删除后，按照2）的方式进行维护二叉排序树]`
- 平均查找长度(查找效率与甚么因素有关**？与二叉树的树形有关。**)：

$$（ASL = 每个元素比较次数之和/总元素个数）$$

$$ASL = \frac{(1+2+3+4+5)}{5} = 3$$

$$ASL = \frac{(1+2+3+2+3)}{5} = 2.2$$

理想情况下，当二叉排序树为高度最短的，即和完全二叉树的树高一样，则有：

$$h = log_2 n,$$

$$此时：ASL = log_2 n$$

$$所以，在平均情况下，二叉排序树的ASL = log_2 n$$

树高定义：从最远的叶子回到根，途径的树枝的条数，$eg:3$个节点，则有$1->2, 3->2$，都是一条树枝，所以高度为1.

- 平衡二叉树(`ALV树`)（**课堂讲的要求**先是二叉搜索树前提）的构建方法（手搓即可，不需算法实现|**hl-hr|<=1**。分为`RR,RL,LR,LL`型。见模拟部分。）
- *2-3树，B-树不要求。手工也麻烦（）*

- 散列查找函数的构建（即构建算法）
  - 直接定址法Hash(key) = a*key+b（线性法）
  - **求模（求余）法[除留余数法]（重点，可能要手工）：**
    - Hash(key) = key%p;（p一般取素数**（减少冲突）**，且素数小于等于表长，接近表长。）
  - 数字分析法：略。（上课掠过，且限制繁多:长度相同，分布均匀））附图：

- 

  - 折叠法：?看不懂讲稿。
  - 平方取中法：关键字^2->取中间几位。（适合关键字中每一位都有某些数字重复出现度高的）
  - 随机数法Hash(key) = rand(key)，适用于长度不等的关键字构造。

- 散列查找解决**冲突问题**的方法：
  - 开放地址法：为冲突地址H(key)求地址序列：$H0, H1...$

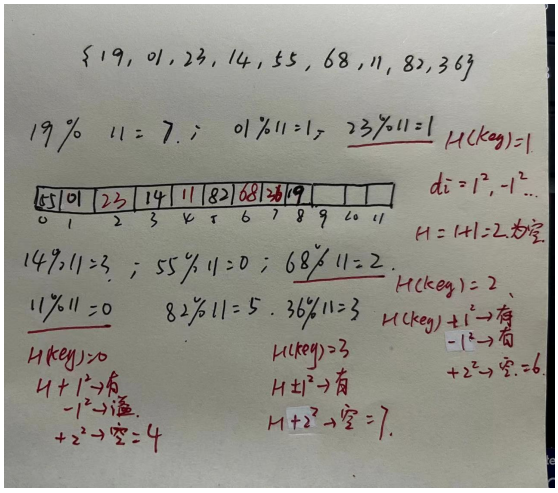
$$H_0 = H(key),\ H_1 = (H(key) + d_i) \qquad i = 1, 2, \dots s$$

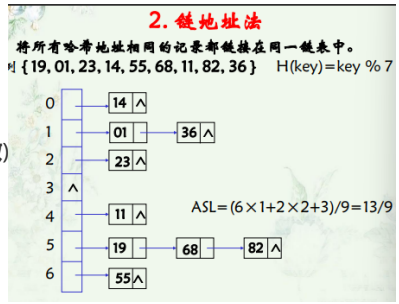
线性探查法求d_i：

1）线性探测再散列处理：$d_i = c * i$，即单重倍增。c为常数，最简单为$c = 1$.

2）平方探测再散列：$d_i = 1^2, -1^2, 2^2, -2^2 \dots$

3）随机探测再散列：d_i为伪随机数列，$or: d_i = i \times H_2(key)$

    - 举例子：`{19,01,23,14,55,68,11,82,36}`：**H(key)=key%11**（实际上有点问题，讲稿，此求余法，p应<=length）
      - 若线性探测再散列（即，线性哈希函数（都是求di，然后加上求余法的散列值，不断试错）探测后再放入）
      - 若二次探测再散列（即，平方哈希函数（都是求di，然后加上求余法的散列值，不断试错）探测后再放入）：
      

  - 拉链法：
    - 求余H(key)，然后冲突的，直接对索引表后链接（和邻接表一样）（**ASL计算即：比较次数(空，比较一次；有一个元素，下一个空，比较两次....)除以元素个数**）
    

    - 算法实现：

```
const int Maxsize = 100;
struct LinkNode
{
```

```
    int data;
    LinkNode* next;
    LinkNode();
    LinkNode():.....
};
struct node
{
    Linknode* head;
} H[Maxsize];//索引表

//插入
void insert()
{
    LinkNode* p;
    int key;
    cin>>key;
    int y = H(key);//散列函数。自定函数%p=k
    p = new LinkNode(key);
    p->next = H[y];//头插入
    H[y] = p;
}
//查找
bool find()
{
    LinkNode* p;
    int key;
    cin>>key;
    int y = H(key);
    p = H[y];
    while(p!=0)
    {
        if(p->data==key) return true;
        p=p->next;
    }
    return false;
}
```

- 平均查找长度的一般计算公式:

$$ASLsucc = (\text{每个元素比较次数之和})/\text{总的元素个数}$$

$$\text{顺序查找(顺序存储，链式存储)的查找效率：}$$
$$ASL = n*(n+1)/2(\text{成功找到}),\ = n(\text{失败})$$

图论

- 图的存储结构

```
//邻接矩阵（相邻矩阵）edge[n][n]，有边使两个顶点*直接*相连则1
class GraphMatrix
{
private:
    int ** edge; //邻接矩阵
    int *verti;  //顶点表
    int row, col;//行列
public:
    //相关操作。构造函数，获取邻接矩阵
    GraphMatrix(int r, int c):row(r),col(c){
```

```
        //顶点表看情况用。
        //一般邻接矩阵为正方矩。
        edge = new int*[r];
        for(int i = 0; i < r; i++)
            edge[i] = new int [c];
    }
    ~GraphMatrix()
    {
        for(int i = 0; i < r; i++)
            delete[]edge[i];
        delete []edge;
    }
    //其余操作看题目
    //矩阵初始化：主对角置0
};


//邻接表
//无向图的邻接表
struct Edge//边指针域
{
    int dest;//边的终点
    //int cost;//权值
    Edge* next;//下一个邻接顶点
};
struct Vertex//顶点表
{
    int data;//表内容，可改为char,string...
    Edge* adj;//邻接表的头指针
};
class GraphLink//也可以不用类的形式，考试时间不够就不写类了，以下类函数，稍微改一下函数参数个数即可直接使用
{
private:
    Vertex* NodeTable;//顶点表
    //获取每个顶点在图中位置，即遍历表，找到结点(data==your_ver)，返回位置，简单，不写。
public:
    //各种操作
    //初始化，建空表，创建顶点表，并初始化所有adj指针。
    //建立算法
    bool CreatNodeTable()
    {
        int n,m;
        Edge* p;
        cin>>n;//结点个数
        for(int i = 1; i <= n; i++)
        {
            NodeTable[i].adj=0;//初始化空链
            cin>>NodeTable[i].data;//输入结点值，简易算法则默认0~n，不需要此步
            cin>>m;//该结点的邻接节点数
            for(int j = 0; j < m; j++)
            {
                p = new Edge;cin>>p->dest;
                //头插法。这才是核心
                p->next = NodeTable[i].adj;
                NodeTable[i].adj=p;
            }
        }
    }
};
```

- 基于存储结构的图遍历算法：
  - 深度优先：

- 原理：一条路走到底，在返回走下一条路。直到所有路点都走过。

- 广度优先：
  - 原理：先将该结点的所有邻接值遍历，再将所有邻接值的邻接遍历..

- 邻接矩阵式实现算法：

```
//存储结构：邻接矩阵
class GraphMatrix
{
public://全部pub简单
    int** edge; //邻接矩阵
    //顶点表此处非必须
    int row,col;
    GraphMatrix():row(0),col(0){}
    GraphMatrix(int r, int c) :row(r), col(c)
    {
        Edge = new int* [row];
        for (int i = 0; i < r; i++)
            Edge[i] = new int[col];
        vertex = new int[row];
    }
    ~GraphMatrix()
    {
        delete[]vertex;
        for (int i = 0; i < row; i++)
            delete []Edge[i];
        delete[]Edge;
    }
    //输入所有节点关系，此处如上述从简，默认从0~n顺序的结点，若为英文字母等，修改即添加顶点表，辅助即可。
    void getVerAndEdge();
}

void GraphMatrix::getVerAndEdge()
{
    cout << "请输入每行数据" << endl;
    for (int i = 0; i < row; i++)
    {
        //cin>>vertex[i];
        for (int j = 0; j < col; j++)
            cin >> Edge[i][j];
    }
}
//深度优先遍历
//开始结点，邻接矩阵，元素已遍历个数。。
//默认为正方矩阵，非正方矩，迷宫问题，见下。
void dfs(int start, GraphMatrix& mat, int length)
{
    if(length == mat.col) return; //遍历完毕
    for(int i = 0; i < mat.row; i++)
    {
        if(mat.edge[start][i]==1&&!mat.edge[i][i])
        {
            cout << start<< " ";//可修改
            mat.edge[i][i] = 1;
            dfs(i, mat, length+1);
        }
    }
}

//广度优先遍历
void bfs(int start, GraphMatrix& mat)
{
    queue<int> q;//存邻接点的名称
    q.push(start);
    mat.edge[start][start] = 1;
```

```
    while(!q.empty())//遍历所有结点,可再定义一个length，判断是否有点无法到达
    {
        int sta = q.front();
        q.pop();
        for(int i = 0; i < mat.col; i++)
            if(mat.edge[sta][i]==1&&!mat.edge[i][i])
            {
                q.push(i);
                mat.edge[i][i]=1;//遍历过该点
                cout<<sta<<"->"<<i<<" ";//可修改
            }
    }
    //if(length!=mat.row) cout<<"there have Vertical can not arrive";
}
```

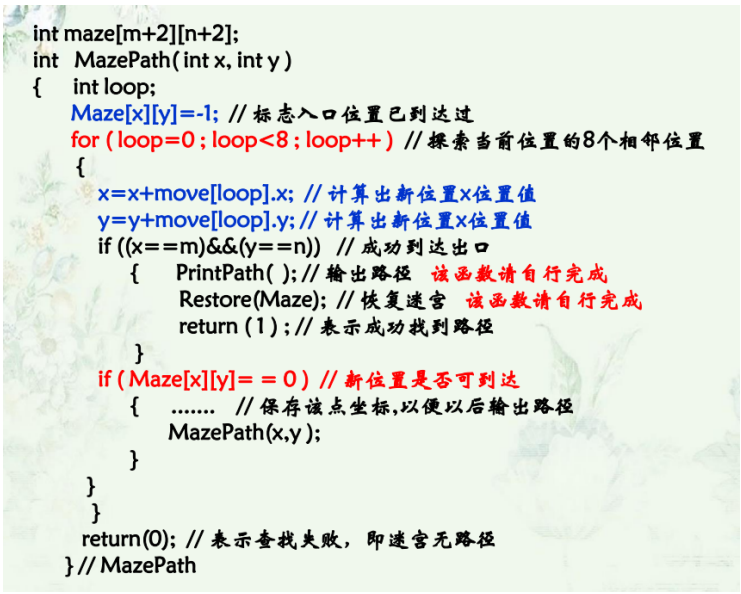
- 邻接表式实现算法

```
//存储结构：邻接表
struct edge
{
    int dest = 0;
    edge* next = NULL;
};
struct vertex
{
    bool visited = false;
    edge* adj = NULL;
    //析构函数不写了，非核心算法
};
//void addEdge(int a, int b, vertex table[])
//{
//  edge* p = new edge;
//    p->dest = b;
//    p->next = table[a].adj;
//    table[a].adj = p; //头插入
//}
//void Init(int v, vertex table[])
//{
//  int nod;
//  cout << "输入邻接表结点:" << endl;
//  while (cin >> nod && nod != -1)
//  {
//      add(v, nod, table);
//  }
//}

//深度优先实现算法：
void dfs(int v, vertex table[])
{
    table[v].visited = true; //遍历过
    edge* tmp = table[v].adj; //找所有邻接点
    cout<<v<<" ";//可修改，或table[v].data;//这个用于输出结点信息。
    while(tmp!=NULL)
    {
        if(!table[tmp->dest].visited)//该结点未被访问过
        {
            dfs(tmp->dest, table);
            tmp = tmp->next;
        }
        else tmp = tmp->next;
    }
    return;
}
```

```
void bfs(int v, vertex table[])
{
    queue<int>q;
    q.push(v);
    table[v].visited = true; //入列
    //for(int i = 0; i < n; i++)visited=false;//根据浴帘算法修改。修改存储结构
    while(!q.empty())
    {
        int tmp = q.front();
        cout<<tmp<<" ";//若为结点值,Nodetable[tmp].data;//需修改存储结构
        q.pop();
        edge* p = table[tmp].adj;//遍历
        while(p!=NULL)
        {
            if(!table[p->dest].visited)
            {
                q.push(p->dest);
                table[p->dest].visited=true;
                p = p->next;
            }
            else node = node->next;
        }
    }
}
```

- 走迷宫问题（8方向）
  - 见图。不写代码了，时间不足（深度：）



```
int maze[m+2][n+2];
int  MazePath( int x, int y )
{   int loop;
    Maze[x][y]=-1; // 标志入口位置已到达过
    for ( loop=0 ; loop<8 ; loop++ ) // 探索当前位置的8个相邻位置
    {
      x=x+move[loop].x; // 计算出新位置x位置值
      y=y+move[loop].y; // 计算出新位置x位置值
      if ((x==m)&&(y==n))  // 成功到达出口
         {    PrintPath( ); // 输出路径   该函数请自行完成
              Restore(Maze); // 恢复迷宫   该函数请自行完成
              return ( 1 ) ; // 表示成功找到路径
         }
      if ( Maze[x][y]= = 0 ) // 新位置是否可到达
         {    .....   // 保存该点坐标,以便以后输出路径
              MazePath(x,y );
         }
    }
    }
    return(0); // 表示查找失败，即迷宫无路径
  } // MazePath
```

  - 广度：

```
int maze[m+2][n+2];
int  MazePath( )
{  queue<DataType> q;  // 定义一个容量为m*n的队列    DataType { int x, int y , int pre; };
   DataType  Temp1,Temp2;       int x, y,loop;
    Temp1.x=1; Temp1.y=1; Temp1.pre=-1; Maze[1][1]=-1;  // 标志入口位置已到达过
    q.push( Temp1 );  // 将入口位置入列
    while ( !q.empty( ) )  // 队列非空，则反复探索
    {  Temp2=q.front( ); q.pop( );  // 队头元素出列
       for ( loop=0 ; loop<8 ; loop++ )  // 探索当前位置的8个相邻位置
       {  x=Temp2.x+move[loop].x;  // 计算出新位置x位置值
          y=Temp2.y+move[loop].y; // 计算出新位置x位置值
          if ((x==m)&&(y==n))   // 成功到达出口
          {     PrintPath( q ); // 输出路径   该函数请自行完成
                Restore(Maze);  // 恢复迷宫   该函数请自行完成
                return ( 1 ) ; // 表示成功找到路径
           }
          if ( Maze[x][y]= = 0 )  // 新位置是否可到达
          {    Temp1.x=x;  Temp1.y=y;
               //Temp1.pre=q.front;//设置到达新位置的前趋位置
               Maze[x][y]=-1;  //标志该位置已到达过
               q.push(Temp1);// 新位置入列
          }
       }
    }
    return(0);  // 表示查找失败，即迷宫无路径
} // MazePath
```

- 最小生成树的实现算法（`kruskal`和`prim`算法。都是基于贪心思想）：
  - `kruskal`步骤：
    - 将所有边放入最小堆
    - 依次取出最小边，放入并查集。若存在回路（即边的两个端点都在同一个集合中），则取下一条边。
    - 直到所有边被取出（即最小堆为空），或提前取出n-1条边（n个顶点，n-1条边），为终止条件。
  - `prim`步骤：
    - 从任意顶点开始，将顶点入最小堆。
    - 将出发点作为一组，其余点作为第二组，找出发点的所有边邻接的结点，检查是否遍历过，未遍历过则入最小堆
    - 从最小堆取出权值最小的边，将其两个顶点判断，是否都遍历过，是则再次取出，直到取出有 一边结点是未访问过的或堆为空才停止。
  - 邻接矩阵存储结构版代码：

```
const int N = 1e5;
int parent[N];
int **GraphMatrix;//邻接矩阵。简化版。
//邻接矩阵的输入，初始化等操作略。

//并查集
void init()
{
    for(int i = 0; i < N; i++)
        parent[i] = i;
}
int find(int x)
{
    return x == parent[x] ? x : find(parent[x]);
}
void merge(int x, int y)
{
    x = find(x); y = find(y);
    if (x != y)parent[x] = parent[y];
}

//Kruskal算法
struct Compare {
    bool operator()(const pair<int, pair<int, int>>& p1, const pair<int, pair<int, int>>&
p2) {
        // 自定义比较函数，按照升序排列
        return p1.first > p2.first;
    }
};
```

```
/*
priority_queue<pair<int, pair<int, int>>, vector<pair<int, pair<int, int>>>,
                    [](const pair<int, pair<int, int>>& p1, const pair<int, pair<int,
int>>& p2) {
                        return p1.first > p2.first;  // 根据你的需求定义比较逻辑
                    }> q;
*/
//卡鲁斯卡尔算法
int kruskal(int** GraphMatrix, int r)
{
    int count = 0; //够r-1，提前终止信号。
    init();
    priority_queue<pair<int, pair<int, int>>, vector<pair<int, pair<int, int>>>, Compare>
q;
    for (int i = 0; i < r; i++)
    {
        for (int j = i + 1; j < r; j++)
        {
            if (GraphMatrix[i][j])
                q.push({ GraphMatrix[i][j],{i,j } });
            else continue;
        }
    }
    int weight = 0;
    while (count < r - 1 && !q.empty())
    {
        auto elem = q.top();
        q.pop();
        if (find(elem.second.first) != find(elem.second.second))
        {
            count++;
            merge(elem.second.first, elem.second.second);
            cout << elem.second.first << "--" << elem.second.second << "  W:" <<
elem.first << endl;
            weight += elem.first;
        }
    }
    if (count != r - 1)return 0;//针对连通分量。
    return weight;
}
//Prim算法
//普利姆算法
int prim(int** GraphMa, int r)
{
    vector<bool> inMST(r, false); //判断该顶点是否在最小生成树中（第一组）
    vector<int> key(r, INT_MAX); //存储顶点的最小边权值
    priority_queue<pair<int, int>, vector<pair<int, int>>, greater<>> q;
    int count = 0; //记录顶点个数
    int weight = 0;

    //从第一个顶点开始
    q.push({ 0,0 });//key,index
    key[0] = 0;
    while (!q.empty() && count < r)
    {
        int u = q.top().second;
        int w = q.top().first;
        q.pop();

        if (inMST[u]) continue;//该点已存在最下生成树中
        inMST[u] = true;
        weight += w;
        count++;
        cout << "顶点" << u << "加入最小生成树 W:" << w << endl;
        for (int v = 0; v < r; v++)
```

```
        {
            if (GraphMa[u][v] && !inMST[v] && GraphMa[u][v] < key[v])//非环，v不在树中，边小于值
            {
                key[v] = GraphMa[u][v];
                q.push({ key[v],v });
            }
        }
    }
    if (count != r)return 0;
    return weight;
}
```

- 邻接表存储结构版代码（略，稍微修改一下邻接矩阵即可）

```
//Kruskal算法
开始，将table表的所有链遍历一遍（注意无向图，判重复，（提前声明也行？），将(w,(u,v))置入优先队列，每次取...
//Prim算法
 开vis数组，从某点出发，将其所有邻接点放入优先队列（注意无向图，判重复（（提前声明也行？）），取最小，判未走，加入边。
```

- 单源最短路径：
  - `dijkstra`算法（适用于正数边）
    - 步骤：
      - 初始化该节点对应的距离表`distance[]`（到每个点的距离），以及相应的访问标志表。标记出发点已访问。
      - 进行n-1轮（假设n个结点）循环，每次找到dis表内最短的距离的点，从该点遍历它的所有邻接点，更新距离表。完毕后将该点置为已访问。
      - 重复第二个操作。
    - 邻接矩阵版代码：

```
//存储结构
long long ** GraphMatrix; //邻接矩阵
long long * dis;//距离表

void dijkstra(int v0, int n, long long dis[], long long** GraphMatrix)
{
    for (int i = 0; i < n; i++)
        dis[i] = GraphMatrix[v0][i];
    GraphMatrix[v0][v0] = 1;  //用于表示访问过该点。
    for (int i = 0; i < n - 1; i++)  //完成n-1次,即n-1个其余点。
    {
        //找距离源点s最小距离点
        int min = INT_MAX;
        int pos = 0; //记录最小距离点的index
        for (int k = 0; k < n; k++)
        {
            if (GraphMatrix[k][k] == 0 && dis[k] < min)
            {
                pos = k;
                min = dis[k];//更新最短距离
            }
        }
        GraphMatrix[pos][pos] = 1;//获取最小距离，使其遍历过,接下来n-1轮不再遍历其
        for (int j = 0; j < n; j++)
        {
            if (GraphMatrix[j][j] == 0 && GraphMatrix[pos][j] + min < dis[j])//即该点到其他任意点距离短于源点到距离。更新源点到最短距离（路径）
                dis[j] = GraphMatrix[pos][j] + min;
```

```
            }
        }

    }
    void display(int v0, int n, long long dis[])
    {
        cout << "源点" << v0 << "到其他任意点的距离为：" << endl;
        for (int i = 0; i < n; i++)
        {
            if (i == v0)
                continue;
            if (dis[i] == INT_MAX)
                cout << "无法抵达" << i << "点" << "  ";
            else
                cout << dis[i] << " ";
        }
    }
```

- 邻接表版代码：

```
//略。稍微修改一下矩阵代码即可
//实现：
操作与邻接矩阵同，开一个dis，vis表，dis表存距离，开始初始化为INT32_MAX。，逐个遍历出发点的邻接点，修改dis，然后就可以和邻接矩阵一样了。每次挑最短的路径，记录索引（然后通过邻接表的table表头，查找到该结点，vis=1,从该结点出发。）
```

- 拓扑排序
  - 思路：
    - 在邻接表/矩阵中找到所有入度为0的点，入队
    - 依次迭代入度为0的点，将其所有邻接点入度-1，并检查是否为0，为0则入队
    - 反复第二操作，直到①全部顶点出现，②存在回路。停止。（回路即开计数器统计次数。）
  - 邻接矩阵版代码(略，稍微修改表即可)

```
/**
 * @param1: 邻接矩阵，adjMatrix[i][j] = 0 表示节点 i 和 j 之间没有边直接相连
 * @return: 拓扑序列
 * @description: 对用邻接表 adjMatrix 表示的图进行拓扑排序
 */
public static int[] tuopuSort(int[][] adjMatrix){
    //n表示图中的节点数
    int n=adjMatrix.length;
    //计算图中每个节点的入度,inDegree[i]=j 表示节点i 的入度为j
    int[] inDegree=new int[n];
    for(int j=0;j<n;j++){
        for(int i=0;i<n;i++){
            if(adjMatrix[i][j]!=0){
                inDegree[j]++;
            }
        }
    }
    //将入度为0的节点加入到队列中
    Queue<Integer> queue=new LinkedList<>();
    for(int i=0;i<n;i++){
        if(inDegree[i]==0){
            queue.offer(i);
        }
    }
    //记录拓扑序列
    int[] order=new int[n];
    //记录遍历节点的顺序
    int cnt=0;
```

```
        //通过BFS算法完成拓扑序列
        while(!queue.isEmpty()){
            //取出队首节点
            int cur=queue.poll();
            //取出节点的顺序即为拓扑排序的结果(这里就是开始记录,拓扑排序的齿轮转动了)
            order[cnt]=cur;
            cnt++;
            //遍历当前节点cur所指向的所有节点
            for(int next=0;next<n;next++){
                if(adjMatrix[cur][next]!=0){
                    //去掉cur指向next的边,故next的入度减1
                    inDegree[next]--;
                    //将入度为0的节点再次加入队列
                    if(inDegree[next]==0){
                        queue.offer(next);
                    }
                }
            }
        }
        if(cnt!=n){
            //图中存在环,拓扑排序不存在
            return new int[]{};
        }else{
            return order;
        }
    }
```

- 邻接表版代码

```
//存储结构
struct GraphNode
{
    int index; //存序号
    GraphNode* next = NULL;
};
struct GraphTable
{
    int w = 0;//权重
    bool vis = false;//入队置true
    GraphNode* adj = NULL;
};
//构造
void getNebNode(GraphTable table[], int n)
{
    int ind;
    for (int i = 0; i < n; i++)
        while (cin >> ind && ind != -1)
        {
            GraphNode* tmp = new GraphNode;
            tmp->index = ind;
            tmp->next = table[i].adj;
            table[i].adj = tmp;
        }
}
//计算每个结点入度
void countInd(GraphTable table[], int n)
{
    for (int i = 0; i < n; i++)
    {
        GraphNode* tmp = table[i].adj;
        while (tmp != NULL)
        {
            table[tmp->index].w++;
```

```
            tmp = tmp->next;
        }
    }
}
//topo排序
void topo(GraphTable table[], int n)
{
    countInd(table, n);
    queue<int> q; //索引表,入度为0则入度
    int count = 0;
    for (int i = 0; i < n; i++)
        if (table[i].w == 0)
        {
            q.push(i);//先将入度为0的入度
            count++;
        }
    //topo
    while (!q.empty())
    {
        //展示
        int index = q.front();
        q.pop();//取出第一个元素
        cout << index << " ";
        GraphNode* tmp = table[index].adj;
        table[index].vis = true;

        while (tmp != NULL)
        {
            table[tmp->index].w--;
            if (table[tmp->index].w == 0&& table[tmp->index].vis==false)
            {
                q.push(tmp->index);
                count++;
            }
            tmp = tmp->next;
        }
    }

    if (count != n)cout << "存在回路" << endl;
    else cout << endl;
}
```

排算法(手工部分统一放在手工模拟)

- 存储结构

```
//顺序表
const int Maxsize = 100;
class Seqlist
{
private:
    int *data;//排序数组
    int maxsize;
    int last;//数组最后一个元素位置
public:
    ....
};

//链式表
struct LinkNode
{
```

```
    int data;
    LinkNode* next;
};
....
```

- 直接插入排序

```
void InsertSort(int data[], int last)//此为类的。写为一般函数，稍微修改即可。
{
  int i, j;
  for(i = 2; i <= last; i++)  //依次将data[2],...data[last] 插入序列中
  {
    data[0] = data[i];
    j = i - 1;
    while( data[0] < data[j] ) //查找x（即原data[i]）的插入位置
      data[j+1] = data[j--]; //将序列中排序码大于x的记录往后移动。
    data[j+1] = data[0]; //填入data[i]
  }
}
//即从第二步开始，，逐一与前一个元素对比，大则前元素后放。注意，最终放入元素需j+1，确保是上一个已搬离元素的位置。
```

- 直接选择排序

```
void SeleteSort(int data[], int last)//此为类的。写为一般函数，稍微修改即可。
{
  int i, j, pos;
  for(i = 1; i < last; i++)  //共作n-1趟选择排序
  {
    pos = i; //用于记录当前排序码最小值的位置所在，即index
    for(j = i+1; j <= last; j++) //找最小
      if(data[j] < data[pos]) pos = j;
    if(pos!=i) //改进，免得本身最小，又交换，多走3步。
    {
      data[0] = data[i];  //data[0]为暂存单元
      data[i] = data[pos];
      data[pos] = data[0];
    }
  }
}
//即小交换到前，然后范围往后。
```

- 冒泡排序

```
//此为大数沉底。
void BubbleSort(int data[], int last) //此为类的，写为一般函数，稍微修改即可。
{
  int i, j;
  for(i = 1; i < last; i++) //共做last-1趟排序。
  {
    flag = 1; //设置交换标志，1表示无交换，0表示有交换。
    for(j = 1; j <= last - i; j++)
    {
      if(data[j]>data[j+1]) //交换
      {
        flag = 0; //存在交换
        data[0] = data[j]; //data[0]是用于记录交换的暂存单元。
        data[j] = data[j+1];
        data[j+1] = data[0];
```

```
      }
    }
    if(flag == 1) break; //全部有序。
  }
}
```

- 直接选择和冒泡的区别：冒泡是**邻居交换**，不断将大/小往后/前挪动。选择是在**一堆里面选最小**，逐个往前交换。
- 堆排序

```
//即向下调整。代码完全一样。
void adjust(int arr[], int start, int len)
{
  int father = last; int son = last*2+1;
  int temp = arr[father];
  while(son<=len)
  {
    if(son<len && arr[son] < arr[son+1])son++;
    if(arr[son] > temp)
    {
      arr[father] = arr[son];
      father = son;
      son = father*2;
    }
    else break;
  arr[father] = temp;
}

void HeapSort(int arr[], int size)
{
  for(int i = size/2-1; i >= 0; i--)//从最后一个非叶子结点向上调整。从下网上是因为：下调整好了，上往下调整后，依然保持有序。若上往下调整，调整一次后，仍上部可能有大小原因。
    adjust(arr, i, size-1);//从0开始，故-1
  //排序部分
  for(int i = size-1; i > 0; i--) //大于0，因为最后只有一个元素即，i=0时，就剩一个元素，不需要排序
  {
    int temp = arr[i];
    arr[i] = arr[0];
    arr[0] = arr[i];
    adjust(arr, 0, i-1);//头向下调整,注意：是i-1，不是size-1.
}

//输出结果。小排序，则大顶堆，大顶交换尾。
//for(int i = 0; i < n; i++) cout<<a[i]<<" ";
```

- 快速排序
  - 找基准值（一般为方便，选第一个元素为基准值。
  - 先从右往左找第一个小于基准值的value，置给基准值的位置。然后该位置空，从左往右找第一个大于基准值的值，将其置给前面这个数的位置。（即一直对调，直到左右指针相遇。）
  - 最后将基准值置入最后一个被置换走的元素位置处。然后将以基准值为分界线，其左部分与右部分依次递归上述操作。（当元素为1时，即开始双指针就相遇时为递归终止返回条件。）
  - 代码如下：

```
int QPass(int elem[], int low, int high)
{
    int l = low, r= high;
    int standard = elem[low];
    while(l<r)
    {
        while(l<r&&elem[r]>=standard) r--; //从右往左找第一个小于基准值standard的value
        if(l<r) elem[l++] = elem[r];
```

```
        while(l<r&&elem[l]<=standard) l++; //从左往右找第一个小于基准值standard的value
        if(l<r) elem[r--] = elem[l];
    }
    elem[l] = standard;
    return l;//mid
}

void Qsort(int elem[], int low, int high)
{
    //加一个终止条件，提提速。
    if(low>high) return;
    int mid;
    mid = QPass(elem, low, high);
    Qsort(elem, low, mid-1);
    Qsort(elem, mid+1, high);
}
```

- 手写部分：
  - 折半插入排序
  - 希尔排序
  - 归并排序
  - 桶排序
  - 基数排序
- 各算法的排序特征（所有排序具体计算，**将放于手工模拟部分**）：
- 稳定性定义：**相对次序**未发生改变。--稳定
排序方法	平均情况	最好情况	最坏情况	算法稳定性
直接插入	$O(n^2)$	$O(n)$	$O(n^2)$	稳定
折半插入	$O(n^2)$	$O(nlog_2n)$	$O(n^2)$	稳定
希尔排序	$O(n^{1.3})$	×	$O(n^{1.3})$	不稳定
直接选择	$O(n^2)$	$O(n^2)$	$O(n^2)$	不稳定
堆排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	不稳定
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	稳定
快速排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(n^2)$	不稳定
归并排序	$O(nlog_2n)$	$O(nlog_2n)$	$O(nlog_2n)$	稳定
桶排序	$O(n)$	$O(n)$	$O(n)$	稳定
基数排序	$O(n)$	$O(n)$	$O(m*(n+d))$	稳定

    - 对于直接插入和折半插入，当插入一个元素后，2个**相等的数**其在序列的**前后位置顺序**和排序后它们两个的前后位置顺序**相同。**举例：
      - 原始序列：$3_1, 1_2, 4_3, 2_4, 1_5$，下标表示元素在原始序列中的位置。
      - 第一轮排序：$1_2, 3_1, 4_3, 2_4, 1_5$，元素1_2插入到3_1之前，但元素1_5的相对位置没有改变。
      - 第二轮排序：$1_2, 2_4, 3_1, 4_3, 1_5$，元素2_4插入到3_1之前，但元素1_5的相对位置没有改变。
    - 对于希尔排序是不稳定的。因为增量划分，可能会导致相等元素划入不同组，进行排序后，按原增量放回数组可能导致相对次序发生变化。
    - 对于直接选择和堆排序，都是不稳定的，
      - 对于直接选择排序：eg：$40_1, 10_2, 40_3$->排序后:$10_2, 40_1, 40_3$，相对次序发生改变。
      - 对于堆排序：eg：
        略。可能相等元素被划入不同的子树后，在该子树，其中一个相同元素要向上调整，从而导致相对次序发生改变
    - 冒泡排序是稳定的，因为是邻居比较大小，交换位置下沉，例如：$40_1, 40_3$，10_2->$40_1, 10_2, 40_3$ -> $10_2, 40_1, 40_3$。未发生改变

- 快排是不稳定的，因为找到基准值后，将两边元素互换会导致，原本靠后的相同元素反而在前面，而原本靠前的元素反而在后面（因为双指针，一条往后走，一条往前走）。
- 归并排序是稳定的，因为按照折半划分区间，如果相同元素划入同一区间，则元素次序不发生改变。若划分如不同区间，排序后，相对次序仍不改变。
- 桶排序，基数排序是稳定的，桶排序是开桶链接**（尾插入）**，所以是稳定的。基数排序，是按照位数排序的，相同的，先出现，排前面...

手工模拟