数据结构期末

序章

- 理解性掌握算法复杂度
- 理解性掌握数据结构中的基本概念和术语
  - 数据结构
  - 数据逻辑结构
  - 数据物理结构
- 了解数据结构的研究范畴
- 理解下掌握算法的度量
  - 时间复杂度
    - 求时间复杂度，左边是“变的”（i++类型直接写i，连乘写成次方），右边是“不变的”（即条件，如果是数则为O(1)，n为n）
  - 空间复杂度

线性表

- 理解性掌握线性表的特点以及线性表的存储，会灵活运用
- **重点掌握**单链表的插入、删除、求长度、遍历的算法及其思想，了解时间复杂度
  - 遍历

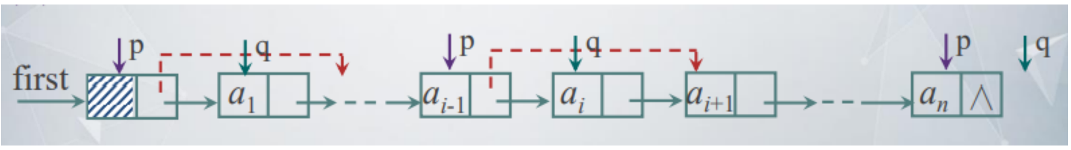
```
void LinkList<DataType> :: PrintList( ) {
    Node<DataType> *p = first->next; //工作指针p初始化
    while (p != nullptr) {
        cout << p->data << "\t";
        p = p->next; //工作指针p后移，注意不能写作p++
    }
    cout << endl;
}
```

  - 插入

```
void LinkList<DataType> :: Insert(int i, DataType x) {
    Node<DataType> *p = first, *s = nullptr; //工作指针p初始化
    int count = 0;
    while (p != nullptr && count < i - 1) { //查找第i - 1个结点
        p = p->next; //工作指针p后移
        count++;
    }
    if (p == nullptr) throw "插入位置错误"; //没有找到第i-1个结点
    else {
        s = new Node<DataType>;
        s->data = x; //申请结点s，数据域为x
        s->next = p->next;
        p->next = s; //将结点s插入到结点p之后
    }
```

```
}
```

- 删除



表尾的特殊情况： 虽然被删结点不存在， 但其前驱结点却存在!

```
DataType LinkList<DataType> :: Delete(int i) {
    DataType x;
    int count = 0;
    Node<DataType> *p = first, *q = nullptr; //工作指针p指向头结点
    while (p != nullptr && count < i - 1) { //查找第i-1个结点
        p = p->next;
        count++;
    }
    if (p == nullptr || p->next == nullptr) throw "删除位置错误"; // 特殊情况  p->next == nullptr
    else {
        q = p->next;
        x = q->data; //暂存被删结点
        p->next = q->next; //摘链
        delete q;
        return x;
    }
}
```

- 掌握单链表的构造算法（头插法、尾插法）及其思想
  - 头插法

```
LinkList<DataType> :: LinkList(DataType a[ ], int n) {
    first = new Node<DataType>;  // 头结点初始化
    first->next = nullptr; //初始化一个空链表
    for (int i = 0; i < n; i++) {
        Node<DataType> *s = nullptr;
        s = new Node<DataType>;
        s->data = a[i];
        s->next = first->next;  // 重要的两步
        first->next = s; //将结点s插入到头结点之后
    }
}
```

  - 尾插法

```
LinkList<DataType> :: LinkList(DataType a[ ], int n) {
    first = new Node<DataType>; //生成头结点
    Node<DataType> *r = first, *s = nullptr; //尾指针初始化
    for (int i = 0; i < n; i++) {
        s = new Node<DataType>;
        s->data = a[i];
        r->next = s;
        r = s; //将结点s插入到终端结点之后
    }
    r->next = nullptr; //单链表建立完毕，将终端结点的指针域置空
}
```

- 掌握顺序表的插入、删除、查找算法，了解其时间复杂度
- 理解顺序表和链表的比较，时间性能和空间性能（什么时候使用顺序表、链表比较好）

栈与队列

- 深刻领会栈的定义、特点。理解栈底、栈顶的含义。掌握运用栈的特点。会写进栈、出栈序列。
    - 后缀表达式：**将操作数依次写下来，再将算数符插在它的两个操作数后面**
      例如a*（b+c）-d的后缀表达式为**abc+*d-**
    - 不可能输出序列的题目技巧：在输出序列中任意元素后面不能出现**比该元素小并且是升序（元素的序号）的两个元素**
    - 左右括号匹配为何使用栈：每个右括号与它前面的最后一个没有匹配的左括号配对
    - 顺序栈入栈出栈

```
void SeqStack<DataType> :: Push(DataType x) {
    if (top == StackSize - 1) throw "上溢";
    data[++top] = x;  // 重点 出栈为top--
}
```

- 深刻领会队列的定义、特点。会运用队列写进队、出队序列。了解队列的典型应用。了解队列有哪些基本操作。
    - 计算队列元素个数公式： **(rear-front+n) %n**
    - 使用一个队尾指针rear指向队尾。入队时只需在队尾进行追加元素即可，时间性能为O(1)。但是出队的操作需要移动n-1个元素，即所有元素向前移动，时间性能为O(n)。、
    - 也可使用front指针指向队首元素，入队时rear+1，出队时front+1。
      且约定：front指向队首元素的前一个元素，rear指向队尾元素的位置。（rear-front=队长）
    - 问题：产生假溢出
- 什么是循环队列。理解性掌握判断队列满和空的条件，会灵活应用。
    - 循环队列将队列看成为首尾相接的循环结构，操作语句为**rear = (rear + 1) % QueueSize**
    - 为了区分判定队满和队空的条件(front==rear)，一般浪费一个数组元素单元，让队头队尾位置相差1，即队满的条件为 (rear + 1) % QueueSize == front
- 掌握在链队列中，入队和出队的算法。
    - 主要使用循环链表 尾指针的方式进行
    - 入队 循环列表 尾指针

```
void Enqueue(Node* rear, int x){
    Node* s = new Node;
    s->data = x;
    if(rear == nullptr){
        rear = s;
        rear->next = s;
    } else {
        s->next = rear->next;
        rear->next = s;
        rear = s;
    }
}
```

- 出队 注意只有一个结点的情况

```
void Dequeuq(Node* rear){
    if(rear==nullptr)
        return ;
    else {
        Node* s = rear->next;
        if(s==rear)
            rear = nullptr; // 链表只有一个结点
        else
            rear->next = s->next;
        delete s;
    }
}
```

字符串

- 深刻领会什么是串、主串、子串
  - 串（字符串）：是n个字符组成的有限序列
  - 串长：串中所包含的字符个数
  - 空串：长度为 0 的串
  - 子串：串中任意个连续的字符组成的子序列
    - 在**主串中寻找子串**的过程叫做**模式匹配**
  - 主串：包含子串的串
  - 子串的位置：子串的第一个字符在主串中的序号
  - KMP中的next数组求法也需要注意一下
  - 存储地址类题目：由列下标得出每一行有多少个元素，算出要求的位置前面总共有多少元素（行数差*每行元素个数+列数差），最后乘上每个元素的大小。
- 了解特殊矩阵有哪些（对称矩阵、三角矩阵...）
  - 对称矩阵、三角矩阵、对角矩阵
  - 稀疏矩阵不是特殊矩阵
    - 因为稀疏矩阵中0没有规律，才需要使用三元组表方法存储

树

- 树的定义，树的基本概念（结点、结点的度、叶子结点、有序树、无序树）
  - 结点的度：结点所拥有的**子树的个数**
  - 树的度：树中各结点度的**最大值**
  - 叶子结点：度为 0 的结点，也称为终端结点
  - 有序树：若将树中每个结点的各子树看成是从左到右有次序的(即不能互换)，则称该树为有序树(Ordered Tree)
  - 无序树：若将树中每个结点的各子树从左到右是没有次序的(即可以互换)，则称该树为无序树
  - 求度为1的结点数、叶子结点数等
  - 完全二叉树：完全二叉树是一种特殊的二叉树，它的每一层都被完全填满，除了最后一层可能不满。这种树的特点是叶子结点只能出现在最下面两层，并且最下面一层的叶子结点都靠左排列。
    - 深度为k的完全二叉树**至少有2^(k-1)个结点**，**至多有2^k-1个结点**
  - 二叉树：每个结点最多有两个孩子的树（不是度为2的树，左右斜树度为1也是二叉树）
- 熟练掌握二叉树的性质，会灵活运用
  - 对于一棵具有n个结点的树，其所有结点的度之和为**n-1** (根节点没有线进入)
  - n = n0 + n1 + n2 + n3 + ...   n = B + 1 (B = n1 + 2n2 + 3n3 + ...) n为结点数
  - 在一棵**二叉树**中，如果叶子结点数为 n0，度为 2 的结点数为 n2，则有:
  - 二叉树的**第 i 层**上**最多有**

    个结点（i≥1）（可从结构上找到规律）
  - 一棵**深度为 k** 的二叉树中，最多有

    个结点（第i层上结点的最大个数之和）
    - 例题：设高度为h的二叉树上只有度为0和度为2的结点，该二叉树的结点数可能到的最大值为（2^h-1），最小值为（2h-1）（第一层为1，其他层只有2个结点）。
  - 具有 n 个结点的完全二叉树的深度为
  - 对一棵具有 n 个结点的**完全二叉树**中从 1 开始按层序编号，对于任意的序号为 i（1≤i≤n）的结点（简称结点 i），有:
    - 如果 i > 1，则结点 i 的双亲结点的序号为 i/2，否则结点 i 无双亲结点
    - 如果 2i≤n，则结点 i 的左孩子的序号为 2i，否则结点 i 无左孩子
    - 如果 2i+1≤n，则结点 i 的右孩子的序号为2i+1，否则结点 i 无右孩子
    - 例：具有100个结点的完全二叉树的叶子节点数为（50）
- **深刻理解 重点掌握** 二叉树的先序、中序、后序遍历算法，会灵活运用
  - 求叶子结点**高度**、左右子树交换、求双亲结点的个数...
    - 求树的结点数

```
int count = 0;
void Count(BiNode* root){
    if(root!=nullptr){
        Count(root->lchild);
        count++;
        Count(root->rchild);
    }
}
```

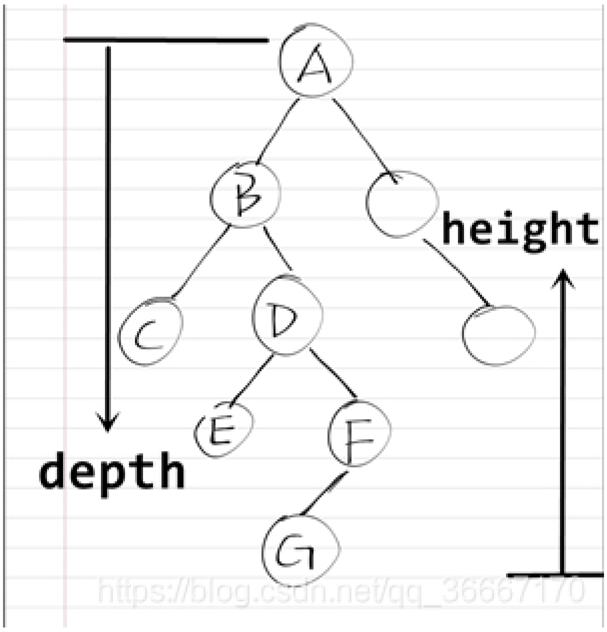
- 求树的高度

```
int Depth(BiNode* root){
    if(root==nullptr)
        return 0;
    else {
        int hl = Depth(root->lchild);
        int hr = Depth(root->rchild);
        return max(hl,hr)+1;
    }
}
```

- 交换左右子树

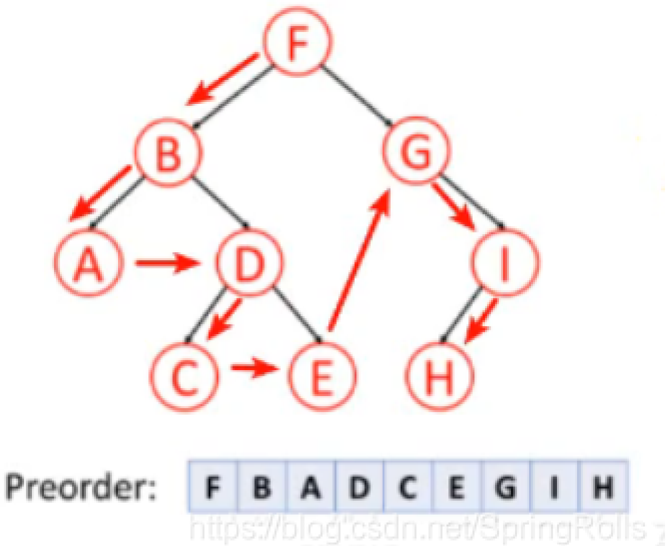
```
void ExchangeTree(BiTree &T) {
    BiTree temp;
if (T != NULL) { // 判断T是否为空，非空进行转换，否则不转换
        temp = T->lchild;
T->lchild = T->rchild; // 直接交换结点地址
        T->rchild = temp;
        ExchangeTree(T->lchild);
        ExchangeTree(T->rchild);
    }
}
```

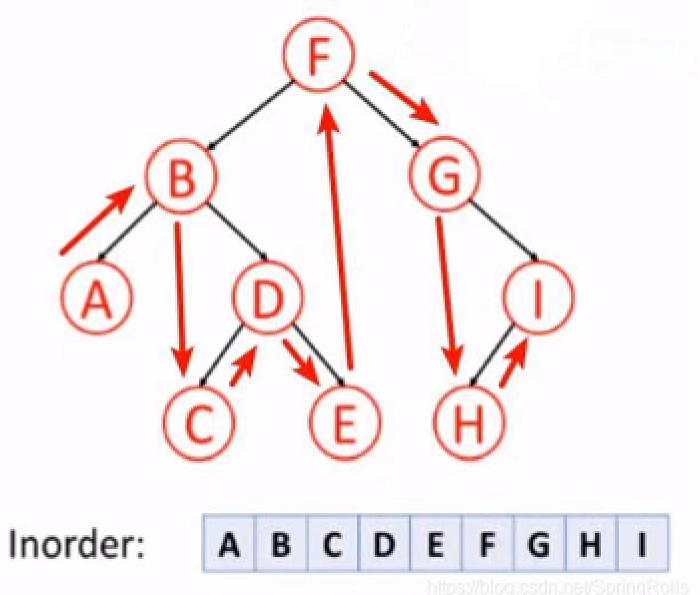
- 求双亲结点个数：使用树的双亲表示法（数组）
- 求叶子结点高度

- 

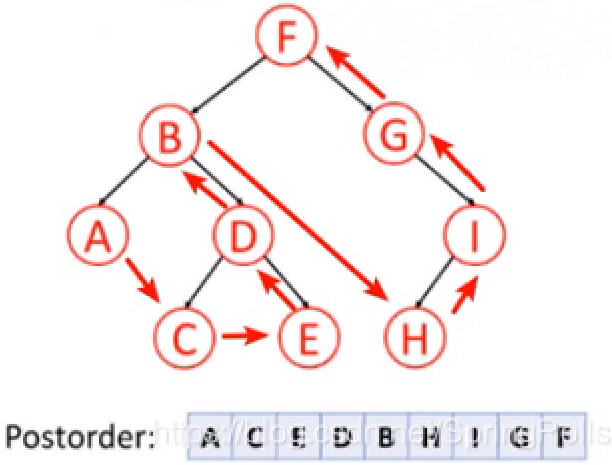
深度定义是从上往下的，高度定义是从下往上的。

```
int leafHeight(TreeNode* root) {
  if (root == NULL) return 0;

    // 如果当前结点是叶子结点，返回高度为1
  if (root->left == NULL && root->right == NULL) return 1;

    // 否则递归求左右子树的叶子结点高度，取最大值
    return max(leafHeight(root->left), leafHeight(root->right));
}
```

- 掌握二叉树的遍历方法（先序、中序、后序），会写遍历序列
  - 前序遍历
    - 

    - 访问根结点
    - 前序遍历根结点的左子树
    - 前序遍历根结点的右子树
  - 中序遍历
    - 

    - 中序遍历根结点的左子树
    - 访问根结点
    - 中序遍历根结点的右子树

- 后序遍历
    - 

    - 后序遍历根结点的左子树
    - 后续遍历根结点的右子树
    - 访问根结点
- **重点掌握**已知二叉树的先序（后序）和中序序列，如何确定一棵二叉树
  - 如果你知道二叉树的先序序列和中序序列，那么你可以使用这两个序列来构建二叉树。首先，在先序序列中，第一个元素一定是根节点的值。接下来，你可以在中序序列中找到根节点的位置，然后根据中序序列中根节点左右两侧的元素将先序序列分成两部分，分别对应根节点的左子树和右子树。接下来，对左子树和右子树分别递归地使用同样的方法进行构建。
- **重点掌握**树如何转换为二叉树，二叉树如何还原成树
  - 树转换成二叉树
    - 加线——树中所有相邻兄弟结点之间加一条连线
    - 去线——对数中的每个结点，只保留它与第一个孩子之间的连线，删去它与其他孩子之间的连线
    - 层次调整——按照二叉树结点之间的关系进行层次调整
  - 二叉树转化为树
    - 加线——若某结点x是双亲结点y的左孩子，则把结点x的右孩子、右孩子的右孩子...都与结点y连起来
    - 去线——删去原二叉树中所有的双亲结点与右孩子结点的连线
    - 层次调整——调整由前两步所得到的树或森林
- **重点掌握**森林如何转换为二叉树，二叉树如何还原为森林
  - 森林转化为二叉树
    - 将森林每一棵树转化为二叉树（树转二叉树）
    - 将每棵树的根结点视为兄弟，在所有根结点之间加上连线
    - 按照二叉树结点之间的关系进行层次调整
- 理解性掌握哈夫曼树的构造过程，会灵活运用\
  - 具有n个叶子结点的哈夫曼树中有n-1个分支结点
  - 哈夫曼树总共有2n-1个结点
  - 哈夫曼树的构造过程：每次选取最小和次小的两个结点合并组成一个新的结点，如此往复得到一棵哈夫曼树。
- 课后习题代码
  - 按前序次序打印二叉树的叶子结点

```
void PreOrderPrint(BiNode* root) {
    if (root != nullptr) {
        if (!root->lchild && !root->rchild)
            cout << root->data;
        PreOrderPrint(root->lchild);
        PreOrderPrint(root->rchild);
    }
}
```

- 求二叉树深度

```
int Depth(BiNode* root) {
    if (root == nullptr)
        return 0;
    else {
        int hl = Depth(root->lchild);
        int hr = Depth(root->rchild);
        return max(hl, hr) + 1;
    }
}
```

- 以二叉链表为存储结构，编写算法求二叉树结点x的双亲

```
BiNode* p = nullptr; // p为全局变量
BiNode* Parent(BiNode* root, int x) {
    if (root != nullptr) {
        if (root->data == x)
            return p;
        else {
            p = root;
            Parent(root->lchild, x);
            Parent(root->rchild, x);
        }
    }
}
```

- 以二叉链表为存储结构，在二叉树中删除以值为x为根结点的子树

```
BiNode* p = nullptr;
void Delete(BiNode* root, int x) {
    if (root != nullptr) {
        if (root->data == x) {
            if (p == nullptr)
                root = nullptr;
            else if (p->lchild == root)
                p->lchild = nullptr;
            else
                p->rchild = nullptr;
        } else {
            p = root;
            Delete(root->lchild, x);
            Delete(root->rchild, x);
        }
    }
}
```

图

- **深刻领会**图的基本概念（有向图、无向图、有向完全图、无向完全图、连通图、连通分量、强连通图）
    - 若图的任意两个边之间的边都是无向边，则称该图为无向图，否则称有向图
    - 无向完全图：无向图中，任意两个顶点之间都存在边
        - 度之和为边数的两倍 2e
        - 有n×(n-1)/2条边
    - 有向完全图：有向图中，任意两个顶点之间都存在方向相反的两条弧
        - 度之和为边数 e
        - 有n×(n-1)条边
    - 若**任意顶点之间均有路径**，则称该图为**连通图**
    - 非连通图的最大连通子图称为连通分量
    - 在**有向图**中，对任意顶点vi和vj，**若从顶点vi到vj均有路径**，则称该有向图是**强连通图**
    - 求顶点数、边数
- 掌握度、入度、出度的计算方法，会根据存储结构求有向图的入度、出度。
    - 顶点的度：在**无向图**中，顶点 v 的度是指依附于该顶点的边数，通常记为TD (v)
    - 顶点的入度：在**有向图**中，顶点 v 的入度是指以该顶点为弧头（指向）的弧的数目，记为ID (v)
    - 顶点的出度：在**有向图**中，顶点 v 的出度是指以该顶点为弧尾（指出）的弧的数目，记为OD (v)
    - 在具有 n 个顶点、e 条边的**无向图**中，各顶点度之和等于边数的两倍（2e）
    - 在具有 n 个顶点、e 条边的**有向图**中，各顶点入读等于出度等于边数（e）
- 掌握图的广度优先搜索遍历算法
    - 头指针负责出队，尾指针负责入队
    - 邻接矩阵

```
template <class T>
void MGraph<T>::BFSTraverse(int v) {
    int Q[MaxSize]; // 定义队列
    int front = -1, rear = -1; // 头指针 尾指针
    cout << vertex[v]; // 打印顶点数据
    visited[v] = 1; // 第一个顶点设置为访问
    Q[++rear] = v; // 第一个顶点进入队列
    while (front != rear) { // 当队列不为空时
        int w = Q[++front]; // 头指针++并取头指针指向的顶点
        for (int j = 0; j < vertexNum; j++) { // 搜索取出来的顶点的周围顶点
            if (arc[w][j] == 1 && visited[j] == 0) { // 如果有边且未访问
                cout << vertex[j]; // 打印邻接点数据
                visited[j] = 1; // 邻接点设置为已访问
                Q[++rear] = j; // 邻接点入队
            }
        }
    }
}
```

    - 邻接表

```
template <typename DataType>
void ALGraph<DataType> :: BFTraverse(int v) {
    int w, j, Q[MaxSize]; // 定义队列
    int front = -1, rear = -1;
    EdgeNode *p = nullptr;
    cout << adjlist[v].vertex; // adjlist是顶点表
    visited[v] = 1;
    Q[++rear] = v;
    while (front != rear) {
        w = Q[++front];
        p = adjlist[w].firstEdge;
        while (p != nullptr) {
            j = p->adjvex; // 取邻接点的编号
            if (visited[j] == 0) {
                cout << adjlist[j].vertex;
                visited[j] = 1;
                Q[++rear] = j;
            }
            p = p->next;
        }
    }
}
```

- 掌握图的两种存储结构（邻接矩阵、邻接表），**重点掌握**图的邻接矩阵、邻接表的表示方法，会灵活运用
  -
- **重点掌握**Prim和Kruskal算法如何求最小生成树，会求最小生成树的权值
  - 最小生成树：连通图的生成树是包含图中**全部顶点**的一个极小连通子图，在图的所有生成树中，代价最小的生成树称为**最小生成树。**
  - ### Prim算法

    加点法

    算法：Prim
    输入：无向连通网G=(V，E)
    输出：最小生成树T=(U，TE)
    1. 初始化：$U = \{v\}$；TE={ }；
    2. 重复下述操作直到$U = V$：
       2.1 在E中寻找最短边(i，j)，且满足$i \in U$，$j \in V$-U;
       2.2 $U = U + \{j\}$；
       2.3 TE = TE + {(i，j)}；

    Prim算法的关键是如何**找到链接U和V-U的最短边来扩充生成树T**

    总共需要迭代n-1次

存储结构

- 图的存储结构：需要不断读取任意两个顶点之间边的权值 --> 邻接矩阵
- 候选最短边集：
    - adjvex[n] --> 候选最短边的邻接点
    - lowcost[n] --> 候选最短边的权值
    - adjvex[i] = j; lowcast[i] = w; 候选最短边(i, j) **(j,i)** 的权值为w

顶点的**更新过程**下：

- lowcast[i] = min{lowcast[i], 边(i, j)的权值} **非联通状态赋值为无限**
- adjvex[i] = j (如果边(i,j)的权值 < lowcast[i])

```
int MinEdge(int lowcast[], int vertexNum) {
    int minest = 0x7fffffff;
    int index = -1;
    for (int i = 0; i < vertexNum; i++) {
        if (lowcast[i] != 0 && lowcast[i] < minest) {
            minest = lowcast[i];
            index = i;
        }
    }
    return index;
}

template <class DataType>
void MGraph<DataType>::Prim(int v) {
    int adjvex[MaxSize], lowcast[MaxSize];
    for (int i = 0; i < vertexNum; i++) {
        lowcast[i] = arc[i][v];
        adjvex[i] = v;
    }
    lowcast[v] = 0; // 将顶点v加入集合U
    for (int k = 1; k < vertexNum; k++) { // 总共迭代n-1次
        int j = MinEdge(lowcast, vertexNum);
        cout << "<" << j << "," << adjvex[j] << ">" << lowcast[j] <<
endl;
        lowcast[j] = 0; // 顶点j加入集合U
        for (int i = 0; i < vertexNum; i++) { // 调整辅助数组
            if (arc[i][j] < lowcast[i] && arc[i][j] != 0) { // 用无限会更
好 如果从新的顶点出发的点边的权值小于旧的
                lowcast[i] = arc[i][j];
                adjvex[i] = j;
            }
        }
    }
}
```

- ### Kruskal算法

加边法

算法：Kruskal算法
输入：无向连通网G=(V，E)
输出：最小生成树T=(U，TE)

1. 初始化：U=V；TE={ }；
2. 重复下述操作直到所有顶点位于一个连通分量：
   2.1 在E中选取最短边(u，v);
   2.2 如果顶点 u、v 位于两个连通分量，则
      2.2.1 将边 (u，v) 并入TE；
      2.2.2 将这两个连通分量合成一个连通分量；
   2.3 在 E 中标记边 (u，v)，使得 (u，v) 不参加后续最短边的选取；

Kruskal算法中心思想是：将T中的顶点各自构成一个联通分量，然后按照**边的权值由小到大的顺序，依次考察边集E中的各条边**。若被考察的**两个顶点属于两个不同的连通分量**，则将此边**加入到TE**中，同时把**两个连通分量链接为一个连通分量。**若被考察边的两个顶点**属于同一个连通分量**，则**舍去此边**，以免造成回路。

存储结构

- 图的存储结构：边集数组表示法

下标：	0	1	2	3	4	5
vertex[6]=	v_0	v_1	v_2	v_3	v_4	v_5

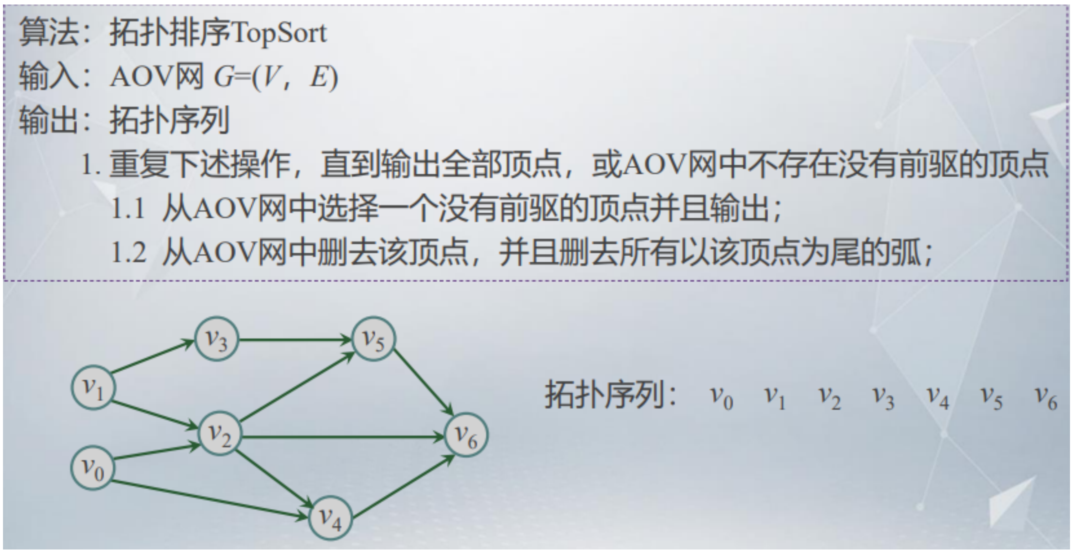
from	1	2	0	2	3	4	0	3	0
to	4	3	5	5	5	5	1	4	2
weight	12	17	19	25	25	26	34	38	46

- 连通分量的顶点所在的集合：由于涉及到集合的查找和合并等操作，考虑采用查并集来实现。查并集是将集合中的元素组织成树的形式。**当合并两个集合时，只需要将一个集合的根结点作为另一个集合根结点的孩子。**
- 查并集的存储结构：双亲表示法 parent[n]
  - 设元素parent[i]表示顶点i的父节点的下标
  - 初始时令parent[i] = -1，表示顶点没有父节点
  - 对于边(u，v)，设vex1和vex2分别表示两个顶点所在集合的根的下标，如果vex1≠vex2，则顶点u和顶点v一定位于两个集合
  - 令parent[vex2] = vex1即可合并两个集合

```
#include<iostream>
using namespace std;
const int MaxVertex = 10;
const int MaxEdge = 100;

struct Edge{
```

```
    int from;
    int to;
    int weight;
};

template <typename DataType>
class EdgeGraph {
    public:
        EdgeGraph(DataType a[ ], int n, int e);
        ~EdgeGraph( );
        void Kruskal( );
    private:
        int FindRoot(int parent[ ], int v);
        DataType vertex[MaxVertex];
        struct Edge edge[MaxEdge];
        int vertexNum, edgeNum;
};

template <typename DataType>
void EdgeGraph<DataType>::Kruskal() {
    int vex1, vex2;
    int parent[vertexNum]; // 使用父节点表示法存储查并集
    for (int i = 0; i < vertexNum; i++) {
        parent[i] = -1;
    }
    for (int i = 0, num = 0; num < vertexNum - 1; i++) {
        // 会有很多边 而且有一部分边是不满足条件的
        vex1 = FindRoot(parent, edge[i].from);
        vex2 = FindRoot(parent,edge[i].to);
        if(vex1!=vex2){
            cout<<"("<<edge[i].from<<","<<edge[i].to<<")"
<<edge[i].weight;
            parent[vex2] = vex1; // 合并边集
            num++; // 加入图中的边的个数 最后加到n-1
        }
    }
}

template <typename DataType>
int EdgeGraph<DataType>::FindRoot(int parent[], int v){
    int t = v;
    while(parent[t]>-1) // 不断寻找父节点
        t = parent[t];
    return t; // 即使没有找到父节点也不会返回-1 而是返回自身的下标
}
```

- 掌握拓扑排序的思想和用途，**重点掌握**给定一个图如何进行拓扑排序，会写出拓扑排序的序列
    - 拓扑排序：设有向图G=(V，E)具有 n 个顶点，则顶点序列 v0 , v1 , ..., vn-1 称为一个拓扑序列，当且仅当满足下列条件：若从顶点 vi 到 vj有一条路径，则在顶点序列中顶点 vi 必在顶点 vj之前
    -




```
void TopSort(){
    int count = 0;
    int S[MaxSize], top = -1;
    EdgeNode *p = nullptr;
    for(int i=0;i<vertexNum;i++) // 扫描顶点表
        if(adjlist[i].in==0) // 入读为0压入栈
            S[++top] = i;
    while(top != -1){
        int j = S[top--]; // 取出顶点
        cout<<adjlist[i].vertex;
        count++;
        p = adjlist[j].firstEdge; // 工作指针p初始化
        while(p != nullptr){
            int k = p->adjvex;
            adjlist[k].in--;
            if(adjlist[k].in==0)
                S[++top] = k;
            p = p->next;
        }
    }
    if(count<vertexNum)
        cout<<"有回路";
}
```

- 会计算关键路径以关键路径长度
  - ### 关键路径

    在表示工程的带权有向图中，用**顶点表示事件**，用**有向边表示活动**，边上的权值表示活动的持续时间。P204

    关键路径：AOE网中从源点到终点的最长路径

    AOE网具有以下两个性质：

    - 只有在进入某顶点的各活动都已经结束，该顶点所代表的事件才能发生
    - 只有在某顶点所代表的事件发生后，从该顶点出发的各活动才能开始

    设带权有向图 G=(V，E)含有 n 个顶点 e 条边，设置 4 个一维数组：

    - **事件**的**最早发生**时间 ve[n] （从开始的顶点，选取路径上的最长时间；扩展时，首先找入度小的）

        - ve[0] = 0
        - ve[k] = max{ve[j] + len} (∈p[k])
    - **事件的最迟发生**时间 vl[n] （从结束的顶点，选取路径上的最短时间；扩展时，首先找出度小的）
        - vl[n-1] = ve[n-1]
        - vl[k] = min{vl[j]-len} （∈s[k]）s[k]： 所有从 vk发出的有向边
    - 活动的最早开始时间 ae[e]
        - ae[i] = ve[k] （只有前一个事件发生了，活动才能开始）
    - 活动的最晚开始时间 al[e]
        - al[i] = vl[j] - len<vk,vj> （后一个事件最迟vl减去边值）
    - **ee[i]和el[i]相等的活动是关键活动，其的边组成关键路径**

- 采用邻接矩阵表示法构造一个无向图的算法
    - 邻接矩阵
      分为**顶点数组vertex[]**和**边集数组arcNum[][]**

```
template <class T>
MGraph<T>::MGraph(T a[], int n, int e) {
    vertexNum = n;
    arcNum = e;
    for (int i = 0; i < vertexNum; i++) // 存储顶点
        vertex[i] = a[i];
    for (int i = 0; i < vertexNum; i++)
        for (int j = 0; j < vertexNum; j++)
            arc[i][j] = 0;
    for (int k = 0; k < arcNum; k++) {
        int i, j;
        cout << "请输入边的两个顶点的序号：";
        cin >> i >> j;
        arc[i][j] = 1; // 边集为1
        arc[j][i] = 1;
    }
}
```

- 采用邻接表表示法构造一个无向图的算法
    - 邻接表
      分为**顶点数组adjlist**，**顶点结点vertexNode**，**边集结点EdgeNode**。

```
struct EdgeNode { // 边集结点
    int adjvex; // 邻接点顶点的编号
    EdgeNode *next;
} ;

template <typename DataType>
struct VertexNode { // 顶点结点
    DataType vertex;
    EdgeNode *firstEdge;
};

// 图的建立
template <typename DataType>
ALGraph<DataType> :: ALGraph(DataType a[ ], int n, int e) {
    int i, j, k;
    EdgeNode *s = nullptr;
```

```
    vertexNum = n;
    edgeNum = e;
    for (i = 0; i < vertexNum; i++) { //输入顶点信息，初始化顶点表
        adjlist[i].vertex = a[i];
        adjlist[i].firstEdge = nullptr;
    }
    for (k = 0; k < edgeNum; k++) { //依次输入每一条边
        cin >> i >> j; //输入边所依附的两个顶点的编号
        s = new EdgeNode;
        s->adjvex = j; //生成一个边表结点s
        // 头插法
        s->next = adjlist[i].firstEdge; //将结点s插入到第i个边表的表头
        adjlist[i].firstEdge = s;
    }
}
```

- 课后习题代码
  - 邻接矩阵转邻接表

```
// 邻接矩阵转换为邻接表
void MatoList(AdjMatrix &A, AdjList &B) {
    for (int i = 0; i < A.vertexNum; i++)
        B.adjlist[i].firstedge = nullptr;
    for (int i = 0; i < A.vertexNUm; i++) {
        for (int j = 0; j < i; j++) {
            if (A.arc[i][j] != 0) {
                ArcNode *p = new ArcNode;
                p->adjvex = j;
                p->next = B.adjlist[i].firstedge;
                B.adjlist[i].firstedge = p;
            }
        }
    }
}
```

  - 统计图中出度为0的顶点个数

```
// 统计邻接矩阵中出度为0的顶点个数
int SumZero(AdjMatrix A) {
    int count = 0;
    for (int i = 0; i < A.vertexNum; i++) {
        int tag = 0;
        for (int j = 0; j < A.vertexNum; j++) {
            if (arcs[i][j] != 0) {
                tag = 1;
                break;
            }
        }
        if (tag == 0)
            count++;
    }
    return count;
}
```

查找

- 查找的基本概念
  - 标识一个记录的某个数据项称为关键码（key），关键码的值称为键值（keyword）
  - 不涉及插入和删除的查找称为静态查找，反之称为动态查找
  - 静态查找不超过时，返回一个不成功的标志；动态查找不成功时，需要将被查找的记录插入到集合中
- **重点掌握**顺序查找、折半查找的递归与非递归算法及其思想，了解其的存储结构，会灵活运用
  - ### 顺序查找

    改进：通过将第一个元素设置为哨兵进而改良查找速度

    ```
    int LineSearch :: SeqSearch2(int k) {
        int i = n;
        data[0] = k;
        while (data[i] != k)
            i--;
        return i;
    }
    ```

    - 查找的时间复杂度为**O(n)**
    - 不考虑元素的有序性，插入删除的时间复杂度为**O(1)**
  - ### 折半查找

    折半查找（对半查找、二分查找）的基本思想：在有序表（假设为递增）中， 取中间记录作为比较对象，**若给定值与中间记录相等，则查找成功**；若给定值小于中间记录，则在有序表的左半区继续查找；若给定值大于中间记录，则在有序表的右半区继续查找。不断重复上述过程，直到查找成功，或查找区域无记录，查找失败

    非递归算法

    ```
    int LineSearch::BinSearch1(int k) {
        int mid, low = 1, high = n; // 假设区间为[1,n]
        while (low <= high) {
            mid = (low + high) / 2;
            if (data[mid] == mid)
                return mid;
            else if (k < data[mid])
                high = mid - 1;
            else
                low = mid + 1;
        }
    }
    ```

    递归算法

    ```
    int LineSearch::BinSearch2(int low, int high, int k) {
        if (data[mid] == k)
            return mid;
        else if (k < data[mid])
            BinSearch2(low, mid - 1, k);
        else
            BinSearch2(mid + 1, high, k);
    }
    ```

- 查找时间复杂度为**O(logn)**
- 插入删除的时间复杂度为**O(n)**

- 什么是二叉排序树，**重点掌握**如何在二叉排序树中插入或删除一个元素，如何构造一棵二叉排序树
  - ### 二叉排序树

    定义：或者是一棵空的二叉树，或者是具有下列性质的二叉树。**与二分查找类似。**

    - 若它的左子树不空，则**左子树上所有结点的值均小于根结点的值**
    - 若它的右子树不空，则**右子树上所有结点的值均大于根结点的值**
    - 它的左右子树也都是二叉排序树

    插入

```
BiNode<int>* BiSortTree::InsertBST(BiNode<int>*bt, int x) {
    if (bt == nullptr) {
        BiNode<int>*s = new BiNode<int>;
        s->data = x;
        bt = s;
        return bt; // 返回叶子结点的地址 构造时使用
    } else if (x < bt->data) // 如果不是空结点
        bt->lchild = InsertBST(bt->lchild, x); // 链接
    else
        bt->rchild = InsertBST(bt->rchild, x);
}
```

    构造

```
BiSortTree::BiSortTree(int a[ ], int n) {
    root = nullptr;
    for (int i = 0; i < n; i++)
        root = InsertBST(root, a[i]);
}
```

    删除

    二叉排序树的删除比较麻烦。删除后的二叉树仍要保持二叉排序树的特性。不失一般性，设待**删除结点为p，其双亲结点为f**，且 **p 是 f 的左孩子**

```
void BiSortTree::DeleteBST(BiNode<int> *p, BiNode<int> *f ) {
    if ((p->lchild == nullptr) && (p->rchild == nullptr)) {
        //p为叶子 直接将父节点对应位置置空
        f->lchild = NULL;
        delete p;
        return;
    }
    if (p->rchild == nullptr) {
        //p只有左子树 将父节点左子树链接p的左子树
        f->lchild = p->lchild;
        delete p;
        return;
    }
    if (p->lchild == nullptr) {
        //p只有右子树 将父节点左子树链接p右子树
        f->lchild = p->rchild;
        delete p;
        return;
```

```
    }

    // p的左右子树均不空
    // 查找p左子树的最右下结点的值替换p的值
    BiNode<int> *par = p, *s = p->lchild;
    while (s->rchild != nullptr) {
        par = s;
        s = s->rchild;
    }
    p->data = s->data;
    // 特殊情况：左子树中的最大值结点是被删结点的孩子
    // 替换的都是par
    if (par == p)
        par->lchild = s->lchild;
    else
        par->rchild = s->lchild
    // 删除替换的结点
    delete s;
}
```

- 理解性掌握平衡二叉树的构造方法
  - ### 平衡二叉树

    平衡因子：该结点的**左子树的深度减去右子树的深度**

    平衡二叉树：或者是一棵空的二叉排序树，或者是具有下列性质 的二叉排序树：

    - 根结点的左子树和右子树的**深度最多相差 1**
    - 根结点的左子树和右子树也都是平衡二叉树

    最小不平衡子树：以距离插入结点最近的、且**平衡因子的绝对值大于 1** 的结点为根的子树

    一旦产生最小不平衡子树，就进行调整 => 不影响其他结点

    对于新结点插入的位置分为LL、RR、LR、RL四种类型（只走两步）。其中前两种调整一次，后两种调整两次。根据扁担原理和旋转优先进行调整。

    - **扁担原理：**将根结点看成是扁担中肩膀的位置
    - **旋转优先：**旋转下来的**结点作为新根结点的孩子**（撞到另外一棵树的结点转化到另一颗树上）
- 什么是散列表、散列函数、散列函数的设计方法，**重点掌握**散列表处理冲突的方法，会灵活运用
- ## 散列函数

  #### 直接定址法

  散列函数是关键码的线性函数，即：

  #### 平方取中法

  对关键码平方后，按散列表大小，取中间的若干位作为散列地址。

  #### 除留余数法

  适用于：最简单、最常用，不要求事先知道关键码的分布

  - 开散列表：使用**拉链法**处理冲突构造的散列表叫开散列表
  - 闭散列表：使用**开放寻址法**处理冲突构造的散列表叫闭散列表

- ## 处理冲突

  开放定址法

  用开放定址法处理的冲突得到的散列表叫做**闭散列表**

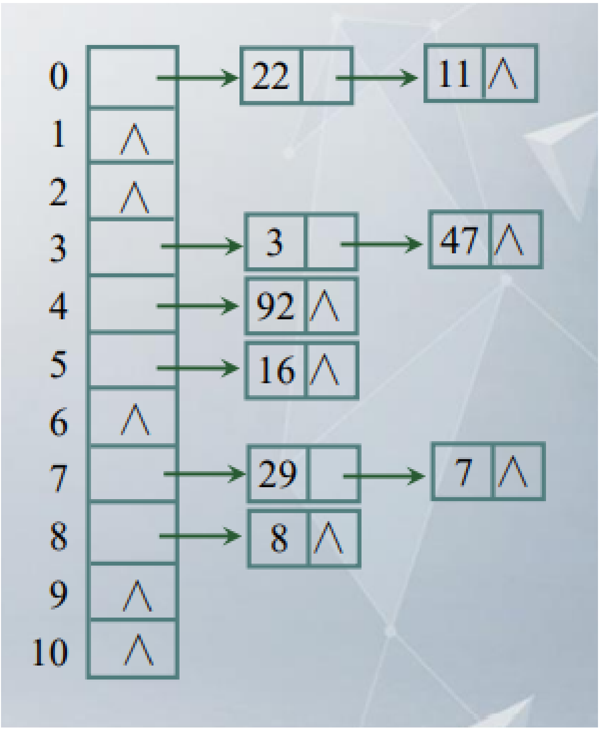
```
// 查找算法
int HashTable1 :: Search(int k) {
    int i, j = H(k); //计算散列地址
    i = j; //设置比较的起始位置
    while (ht[i] != 0) {
        if (ht[i] == k) return i; //查找成功
        else i = (i + 1) % m; //向后探测一个位置
    }
    return -1; //查找失败
}
```

  拉链法

  同义词子表：所有散列地址相同的记录构成的单链表。

  开散列表：用拉链法处理冲突得到的散列表。

  开散列表中存储同义词子表的**头指针**，开散列表不会出现堆积现象




```
Node<int>* ht[MaxSize]; // 开散列表
// 构造算法
j = H(k); // 哈希函数得出来的key
Node<int> *p = ht[j];
while (p != nullptr) {
    if (p->data == k) break;
    else p = p->next;
}
if (p == null) {
    q = new Node<int>;
    q->data = k;
```

```
    q->next = ht[j];
    ht[j] = q;
}
```

```
// 查找算法
Node<int> * HashTable2 :: Search(int k) {
    int j = H(k);
    Node<int> *p = ht[j];
    while (p != nullptr) {
        if (p->data == k) return p;
        else p = p->next;
    }
    return nullptr;
}
```

- **重点掌握**如何在二叉排序树中查找一个元素的算法
  - 查找的过程：
    - 若 bt 是空树，则查找失败
    - 若k = bt->data，则查找成功
    - 若k < bt->data，则在 bt 的左子树上查找
    - 若k > bt->data，在 bt 的右子树上查找

```
BiNode<int>* BiSortTree::SearchBST(BiNode<int>*bt, int k){
    if(bt== nullptr) // 没有查询到
        return nullptr;
    if(bt->data == k)
        return bt;
    else  if(k<bt->data)
        return SearchBST(bt->lchild, k);
    else
        return SearchBST(bt->rchild, k);
}
```

排序

- 了解排序的基本概念，了解一趟的概念
  - 一趟：**将待排序的记录序列扫描一遍称为一趟**
- 理解性掌握所有排序的排序过程
  - ## 插入排序

    ### 直接插入排序

    基本思想：依次将**待排序序列中的每一个记录**插入到**已排好序的序列**中，直到全部记录都排好序。

```
void InsertSort(int r[ ], int n) {
    int i,j;
    for (i = 1; i < n; i++) { // 对于长度为n的数组 排序执行n-1次
        int temp = r[i];
        // 与待查元素之前的元素进行比较 j--的存在可以使j处在空白位后一位
        for (j = i - 1; j >= 0 && temp < r[j]; j--) {
            r[j + 1] = r[j];
        }
        r[j + 1] = temp;
    }
}
```

时间复杂度为O(n^2)

- **重点掌握**冒泡排序、快速排序、希尔排序的排序方法及其过程
- 什么是堆，如何构建一个堆，**重点掌握**堆排序的调整过程及其排序过程
    - ## 堆排序

    堆的定义：堆是具有以下性质的完全二叉树

        - 每个结点的值都小于或等于其左右孩子结点的值（小根堆）
        - 每个结点的值都大于或等于其左右孩子结点的值（大根堆）

    堆排序的基本思想：首先将待排序序列构造成一个大根堆，即**选出了堆中所有记录的最大者**，将它从堆中移走，并将剩余的记录再调整成堆，这样又找出了次小的记录，以此类推，直到堆中只有一个记录。

    建堆

```
void Sift(int data[], int k, int last) {
    int i, j, temp;
    i = k;
    j = 2 * i + 1; // 子节点位置
    while (j <= last) {
        if (j < last && data[j] < data[j + 1])
            j++;
        if (data[i] > data[j]) // 已经是堆
            break;
        else {
            temp = data[i];
            data[i] = data[j];
            data[j] = temp;
            i = j;
            j = 2 * i + 1; // 被调整结点位于结点j的位置
        }
    }
}
```

    建完初始堆后，将堆顶元素输出放入有序区，之后在重建堆，直至全部有序。

```
void HeapSort(int data[], int length) {
    int i, temp;
    for (i = ceil(length / 2) - 1; i >= 0; i--)
        // 从最后一个分支结点到根结点 初始化堆
        Sift(data, i, length - 1);
    for (i = 1; i < length; i++) {
        temp = data[0]; // 获取堆顶
        data[0] = data[length - 1];
        data[length - 1] = temp;
        Sift(data, 0, length - i - 1); // 重建堆
    }
}
```

时间复杂度为**O(log2n)**

- 掌握希尔排序的算法
  - ### 希尔排序

希尔排序是对直接插入排序的一种改进。

改进点：

- 若待排序记录基本有序，直接插入排序的效率很高
- 待排序记录个数较少时效率也很高

基本思想：将整个待排序序列分割成若干个子序列，在子序列内部分别进行插入排序，待到序列整体有序的时候再对全体进行直接插入排序。

相距d非常重要。对相距d的元素进行排序。

```
void ShellSort(int r[ ], int n) {
    int i, j, d;
    for (d = n / 2; d >= 1; d = d / 2) { // 增量为d进行直接插入排序
        for (i = d; i < n; i++) { // 原来为1的地方改成d
            int temp = r[i];
            for (j = i - d; j >= 0 && temp < r[j]; j = j - d) {
                r[j + d] = r[j];
            }
            r[j + d] = temp;
        }
    }
}
```

- 掌握冒泡排序的算法
  - ### 冒泡排序

基本思想：两两比较相邻记录，如果反序则交换，知道没有反序记录为止。

```
void bubble_sort(int array[], int n) {
    int j, exchange, bound, temp;
    exchange = n - 1; // 第一趟冒泡排序区间是[0~n-1]
    while (exchange != 0) {
        bound = exchange; // 确定边界
        exchange = 0;
        for (j = 0; j < bound; j++) {
            if (array[j > array[j + 1]]) {
                temp = array[j];
```

```
                    array[j] = array[j + 1];
                    array[j + 1] = temp;
                    exchange = j;
                }
            }
        }
    }
```

- 掌握快速排序的算法
    - ### 快速排序

      改进点：记录的比较和移动从两端向中间进行，值较大的记录一次就能从前面移动到后面，值较小类似。记录移动的距离较远，从而减少比较和移动次数。

      基本思想：首先确定一个**轴值**，将待排序记录划分成两部分，左侧记录均小于或等于轴值，右侧记录均大于或等于轴值（最终情况）。然后分别对这两部分重复上述过程，直到整个序列有序。

      一般而言，轴值选取第一个记录。

```
// 划分算法
int Partition(int arr[], int first, int last) {
    int i = first, j = last, temp;
    while (i < j) {
        while (i < j && arr[i] <= arr[j])
            j--; // 右侧扫描
        // 跳出循环说明产生异常情况 => 交换元素
        if (i < j) {
            Swap(arr[i], arr[j]);
            i++;
        }

        while (i < j && arr[i] <= arr[j])
            i++; // 左侧扫描
        // 跳出循环说明产生异常情况 => 交换元素
        if (i < j) {
            Swap(arr[i], arr[j]);
            j--;
        }
    }
    return i; // i为轴值记录的最终位置
}

void QuickSort(int arr[], int first, int last) {
    if (first >= last)
        return ; // 区间长度为，递归结束
    else {
        int pivot = Partition(arr, first, last);
        QuickSort(arr, first, pivot - 1); // 对左侧子序列进行快速排序
        QuickSort(arr, pivot + 1, last); // 对右侧子序列进行快速排序
    }
}
```

      时间复杂度为**O(nlogn)**

- 了解各种算法的综合比较

排序方法	平均情况	最好情况	最坏情况
直接插入排序	$O(n^2)$	$O(n)$	$O(n^2)$
希尔排序	$O(n\log_2 n)$~$O(n^2)$	$O(n^{1.3})$	$O(n^2)$
起泡排序	$O(n^2)$	$O(n)$	$O(n^2)$
快速排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n^2)$
简单选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$
堆排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$
归并排序	$O(n\log_2 n)$	$O(n\log_2 n)$	$O(n\log_2 n)$

考试

- 闭卷考试
- 选择题 20题 40分
- 画图题 4题 24题
- 综合应用 2题 16分
- 程序设计 3题 20分

第五章和第六章为全书重点（二十分+）

第二章、第七章、第八章（十几分）

第一章、第三章、第四章（几分）

学者网试题