知识点汇总

# 基础算法

## 排序

冒泡排序

插入排序

选择排序

快速排序

计数排序

## 位运算

与、或、非、补码、左移、右移

## 搜索（参考BFS & DFS.pdf）

### BFS--宽度（广度）优先搜索

宽度优先搜索是一种“盲目”搜索，所有结点的拓展都遵循“先进先出”的原则，所以采用“队列”来存储这些状态。

算法框架：

**void bfs()**

**{**

**while (front <= rear) //队列为空**

**{**

**if (找到目标状态)**

**//做相应处理(如退出循环输出解、输出当前解、比较解的优劣);**

**else**

**{**

**if ( 拓展出的新结点没出现过 )**

**{**

**rear++; //将新结点插到队尾**

**}**

**}**

**front++; //取下一个结点**

**}**

**}**

### DFS--深度优先搜索

深度优先搜索是将当前状态按照一定的规则顺序，先拓展一步得到一个新状态，再对这个新状态递归拓展下去。如果无法拓展，则退回一步到上一个状态，再按照原先设定的规则顺序重新寻找一个状态拓展。如此搜索，直至找到目标状态，或者遍历完所有状态。所以，深度优先搜索也是一种“盲目”搜索。

深搜中“退回一步”的顺序符合“后进先出”的特点，所以采用“栈”存储状态。

算法框架：

**void dfs(int dep, 参数表)**

**{**

**if (当前是目标状态)**

**{**

**// 结果输出;**

**}**

**else**

**{**

**for (i = 1; i <= 状态的拓展可能数; i++)**

**{**

**if ( 第 i 种状态拓展可行)**

**{**

**dfs(dep+1, 参数表);**

**}**

**}**

**}**

**}**

## 回溯

**回溯法（back tracking）**也称“试探法”。它是从问题的某一状态出发，不断“试探”着往前走一步，当一条路走到“尽头”，不能再前进（拓展出新状态）的时候，再倒回一步或者若干步，从另一种可能的状态出发，继续搜索，直到所有的“路径（状态）”都一一试探过。这种不断前进、不断回溯，寻找解的方法，称为“回溯法”。

深度优先搜索求解的时候，当找到目标结点之后，还要回头寻找初始结点到目标结点的解路径。而回溯法则不同，找到目标结点之后，搜索路径就是一条从初始结点到目标结点的解路径。回溯法实际上是状态空间搜索中，深度优先搜索的一种改进，是更实用的一种搜索求解方法。

算法框架：

**void search(int dep)**

**{**

**自定义参数 ;**

**if( 当前是目标状态 )**

**{**

**输出解或者作计数和评价处理 ;**

**}**

**else**

**{**

**for(i = 1; i <= 状态的拓展可能数 ; i++)**

**{**

**if( 第 i 种状态拓展可行 )**

**{**

**保存现场 ( 断点 ), 维护自定义参数 ;**

**search(dep+1);**

**恢复现场 , 回溯到上一个断点继续执行 ;**

**}**

**}**

**}**

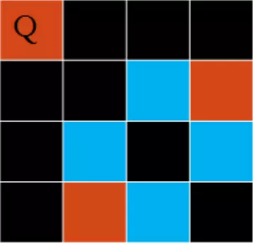
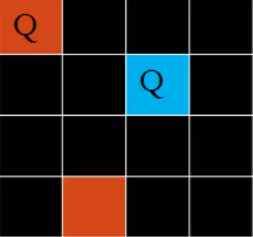
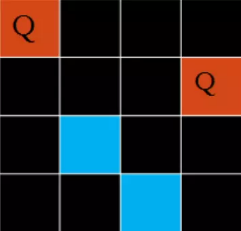
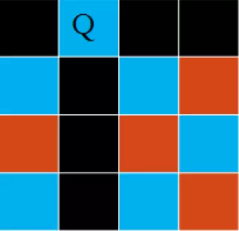
**}**

举例：八皇后问题

八皇后问题是一个古老而著名的问题，是回溯算法的典型例题。该问题是十九世纪著名的数学家高斯1850年提出：在8X8格的国际象棋上摆放八个皇后（棋子），使其不能互相攻击，即任意两个皇后都不能处于同一行、同一列或同一斜线上。

问题简化：下面我们将八皇后问题转化为四皇后问题，并用回溯法来找到它的解

**目的：**在4x4棋盘上，使得4个皇后不能在同行同列以及同斜线上。

**代码实现八皇后问题**

**我们将算法也设置成两步，**

* **第一步：我们要判断每次输入的皇后是否在同一行同一列，或者同一斜线上。**

**bool is\_ok(int row)**

**{**

**//判断设置的皇后是否在同一行，同一列，或者同一斜线上**

**for (int j = 0; j < row; j++)**

**{**

**if (queen[row] == queen[j] || row - queen[row] == j - queen[j] || row + queen[row] == j + queen[j])**

**return false;**

**}**

**return true;**

**}**

* **第二步：核心算法（回溯）**

**void back\_tracking(int row = 0) //算法函数，从第0行开始遍历**

**{**

**if (row == n)**

**t++; //判断若遍历完成，就进行计数**

**for (int col = 0; col < n; col++) //遍历棋盘每一列**

**{**

**queen[row] = col; //将皇后的位置记录在数组**

**if (is\_ok(row)) //判断皇后的位置是否有冲突**

**back\_tracking(row + 1); //递归，计算下一个皇后的位置**

**}**

**}**

## 分治

分治法将问题分(divide)成一些小的问题然后递归求解，而治(conquer)的阶段则将分的阶段得到的各答案“修补”在一起，即分而治之。

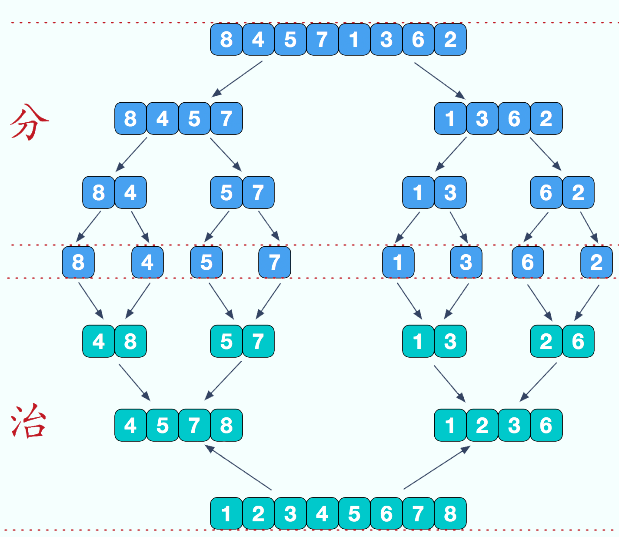
分而治之，把一个复杂的问题分解成很多规模较小的子问题，然后解决这些子问题，把解决的子问题合并起来，大问题就解决了。

### 二分查找

     二分查找的基本思想：设R[low，high]是当前的查找区间

1. 首先确定该区间的中点位置：
2. 然后将待查的K值与R[mid].key比较：若相等，则查找成功并返回此位置，否则须确定新的查找区间，继续二分查找。

### 归并排序



## 贪心

顾名思义，贪心算法总是作出在当前看来最好的选择。也就是说贪心算法并不从整体最优考虑，它所作出的选择只是在某种意义上的局部最优选择。

当然，希望贪心算法得到的最终结果也是整体最优的。虽然贪心算法不能对所有问题都得到整体最优解， 但对许多问题它能产生整体最优解。如单源最短路经问题，最小生成树问题等。在一些情况下，即使贪心算法不能得到整体最优解，其最终结果却是最优解的很好近似。

## 快速幂

快速幂的目的就是做到快速求幂，其原理是：

以求a的b次方为例。

1. 把b转换成[二进制数](https://baike.baidu.com/item/%E4%BA%8C%E8%BF%9B%E5%88%B6%E6%95%B0" \t "_blank)

该二进制数第i位的权为 https://gss1.bdstatic.com/9vo3dSag_xI4khGkpoWK1HF6hhy/baike/s%3D26/sign=6044c668f703738dda4a0b24b21ba471/faf2b2119313b07e13c5e9c00dd7912396dd8cfa.jpg 例如：https://gss3.bdstatic.com/7Po3dSag_xI4khGkpoWK1HF6hhy/baike/s%3D103/sign=885b42e39c16fdfadc6cc2ee878d8cea/d0c8a786c9177f3e48bd45c571cf3bc79e3d5668.jpg

11的二进制是1011

https://gss3.bdstatic.com/7Po3dSag_xI4khGkpoWK1HF6hhy/baike/s%3D247/sign=2a9c504edfc8a786ba2a4d0a5008c9c7/aa18972bd40735fae94ebe0d90510fb30e2408c3.jpg

1. 将计算a¹¹转化为计算 https://gss3.bdstatic.com/-Po3dSag_xI4khGkpoWK1HF6hhy/baike/s%3D95/sign=257afe75a8773912c0268964f919a226/e850352ac65c1038803a760ab3119313b17e89b6.jpg

代码：

**int Pow(int a, int b)**

**{**

**int ans = 1;**

**int base = a;**

**while (b)**

**{**

**if (b & 1)**

**{**

**ans \*= base;**

**}**

**base \*= base;**

**b >>= 1;**

**}**

**return ans;**

**}**

## 三分

类似于二分查找，三分搜索法也是比较常用的基于分治思想的高效查找方法。但是和二分不同，二分只适用于单调函数，三分用于单峰函数。

设当前求解区间为[l, r]，则m1 = l + (r – l)/3， m2 = r - (r – l)/3。

参考程序：

double l = 0, r = 1e9;

while (r - l >= 1e-3)

{

double m1 = l + (r - 1) / 3, m2 = r - (r - l) / 3;

if (f(m1) < f(m2))

l = m1;

else

r = m2;

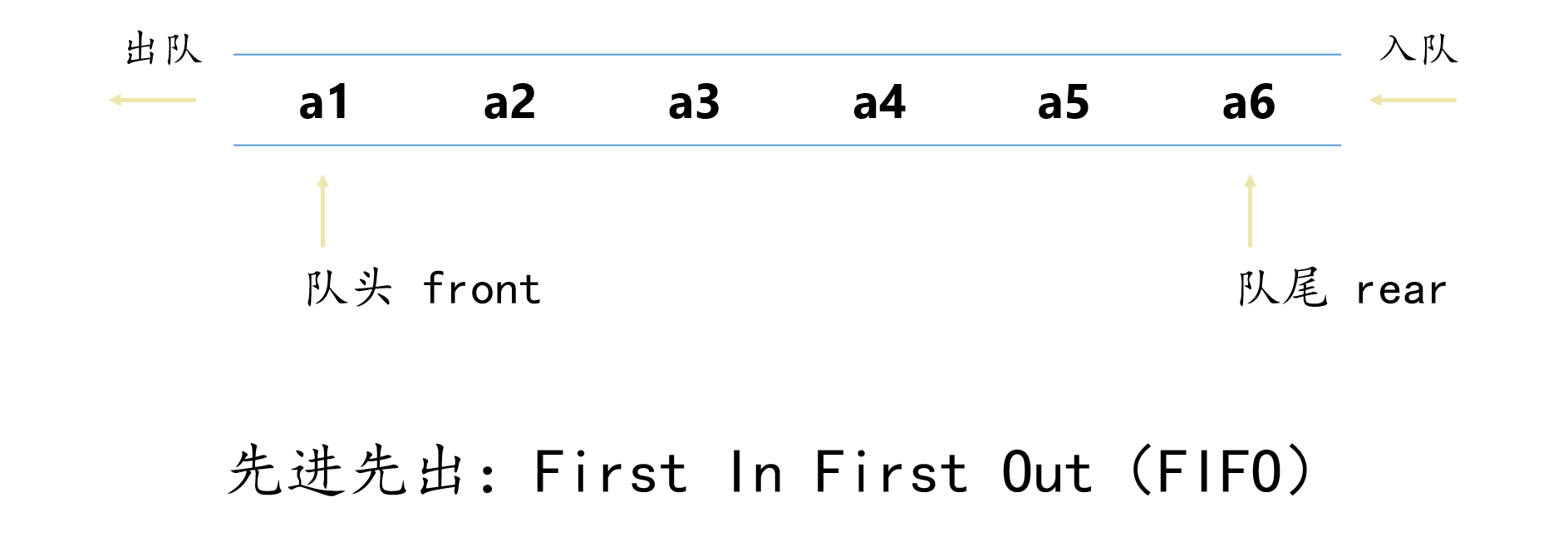
}

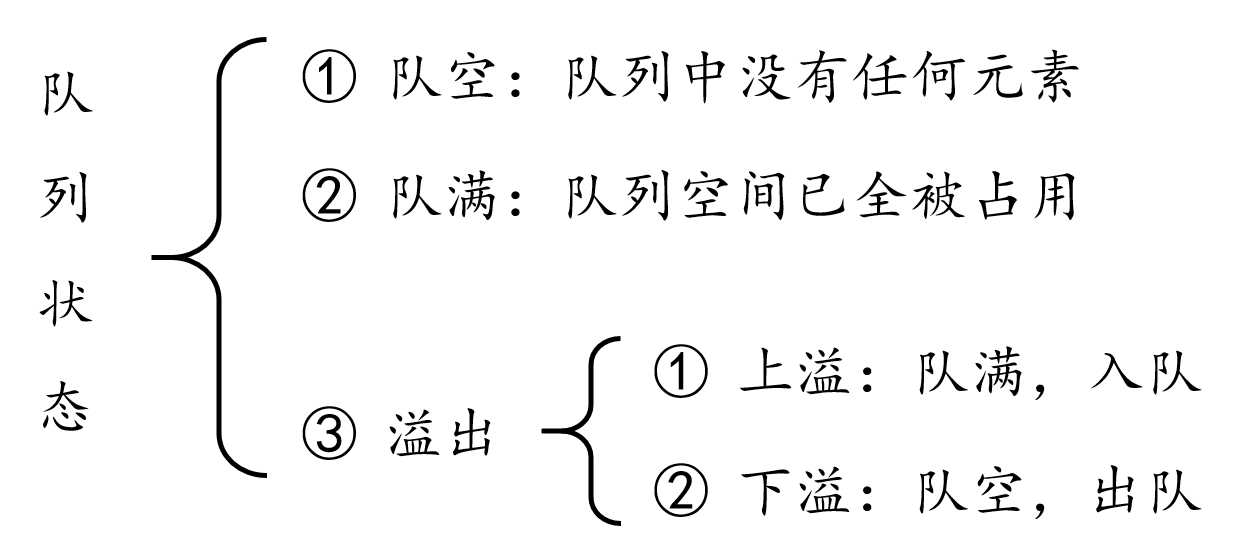
# 数据结构

## 链表

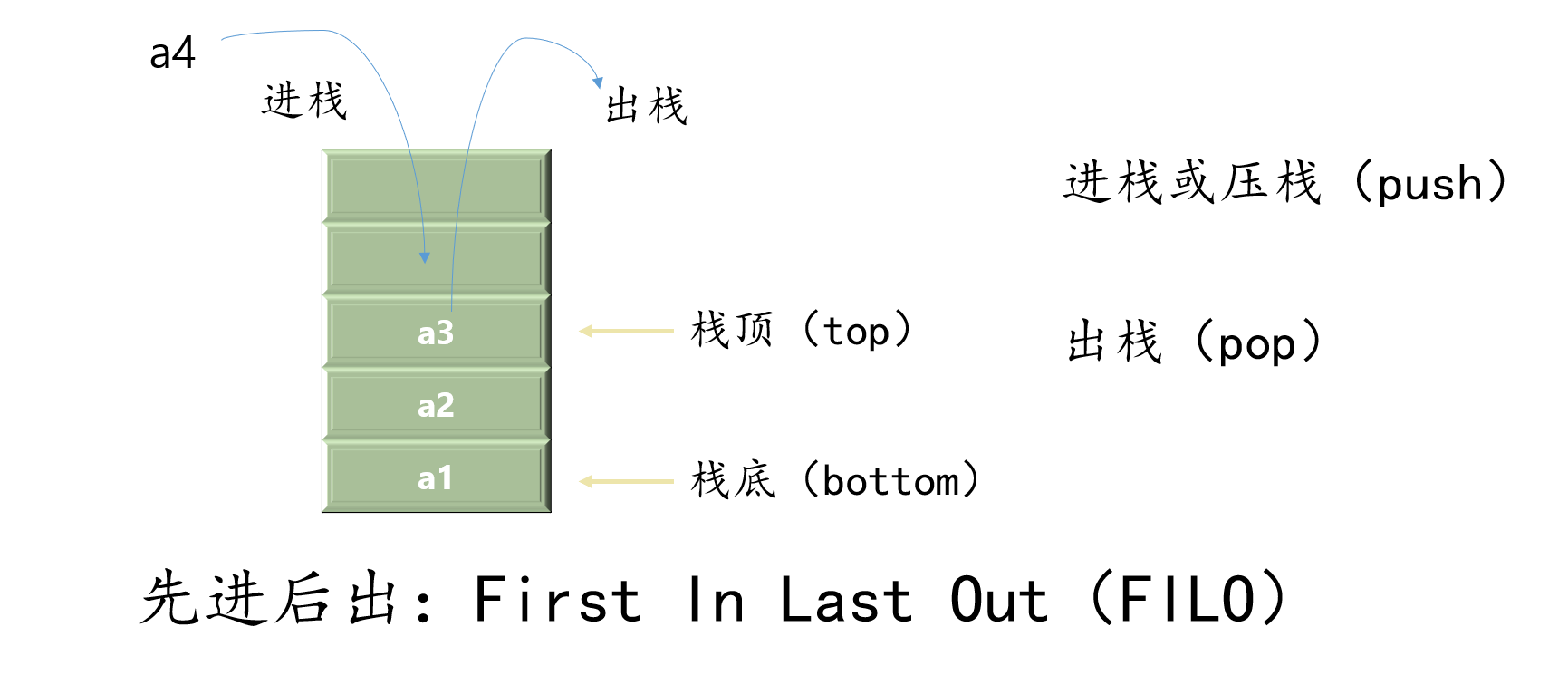
## 栈、队列

### 队列





### 栈



## 树、二叉树

### 树

树：由n(n>0)个元素组成的有限集合。

1. 每个元素称为结点（node）；
2. 有一个特殊结点，称为根结点或树根（root）；
3. 除根结点外，其余结点能分成m（m>=0）个互不相交的有限集合。

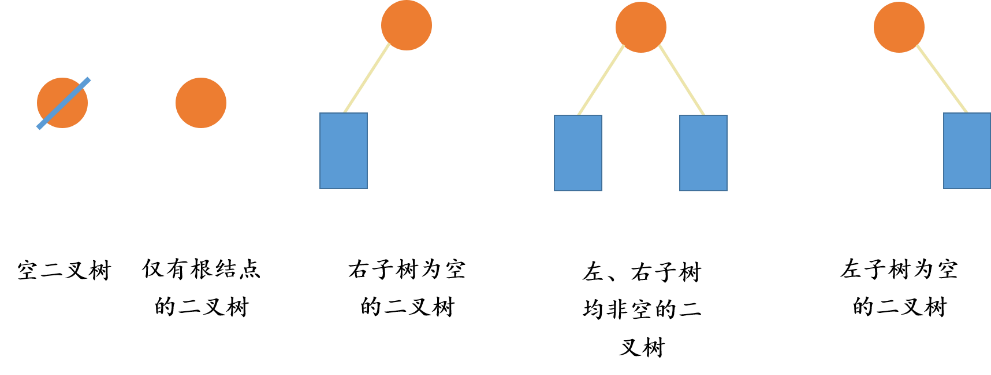
树的基本概念：

1. 一棵树至少有1个根结点（root）；
2. 前驱结点、后继结点；
3. 度（degree）：一个结点的子树个数；树中各结点的度的最大值称为树的度；
4. 叶结点（度为0）；
5. 父结点、子结点、兄弟结点；
6. 层次（level）：根结点的层次为1；
7. 深度（depth）：树中所有结点的层次的最大值。

树的遍历：先序、后序、层次、叶节点

### 二叉树

度数为2的树，二叉树的每个结点最多有两个子结点。

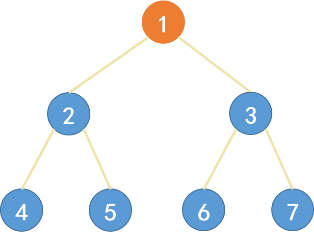


二叉树的性质：

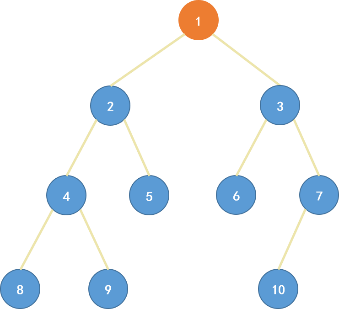
1. 性质1：二叉树第i层上的结点数目最多为 2i-1 (i≥1)；
2. 性质2：深度为k的二叉树至多有2k-1个结点(k≥1) ；
3. 性质3：包含n个结点的二叉树的深度至少为log2 (n+1);
4. 性质4：在任意一棵二叉树中，若叶结点的个数为n0，度为2的结点数为n2，则n0=n2+1。

二叉树的遍历：先序、中序、后序

满二叉树：除最后一层无任何子结点外，每一层上的所有结点都有两个子结点的二叉树。



完全二叉树：若设二叉树的深度为h，除第 h 层外，其它各层 (1～h-1) 的结点数都达到最大个数，第 h 层所有的结点都连续集中在最左边，这就是完全二叉树。



## 堆

1. 建立在完全二叉树的基础上；
2. 排序算法的一种，也是稳定效率最高的一种；
3. 实现STL中的优先队列(priority\_queue)。

堆分为最大堆和最小堆：

* 最大堆：父节点的值比子节点的值都要大；
* 最小堆：父节点的值比子节点的值都要小。

## 图

1. 图：二元组 G(V,E)称为图(graph)。 其中，V为结点(node)或顶点(vertex)集。 E为图中结点之间的边的集合；点：用数字0…n-1表示；
2. 边：点对(u,v)称为边(edge)或称弧(arc)；
3. 对于边(u,v)∈E，u和v邻接(adjacent)，e和u、v关联(incident)。
4. 子图(subgraph):边的子集和相关联的点集。

图的基本概念：

1. 有向图：边都是有向边，是单向的；
2. 无向图：边都是无向边，是双向的；
3. 带权图：图的边带一个权值，表示具体的含义，例如距离、费用、拥堵程度等等，权值可以是正值，也可以是0或负值；
4. 图的稠密性：n为顶点数，边数和n(n-1)/2相比非常少的图称为稀疏图，反之为稠密图；
5. 完全图：边数为n(n-1)/2的图称为完全图；
6. 点的度：图中与某一个点相连的边数称为该点的度；
7. 连通图（无向图 ）：如果图中任意两点都有路径相连, 则称图是连通图, 否则称图是非连通的；
8. 树：一种特殊的图，是n个点、n-1条边的连通图(无环连通图)；
9. DAG（有向无环图）：指任意一条边有方向，且不存在环路的图。

## 并查集

并查集（Union-Find Set）是一种用于分离集合操作的抽象数据类型，其处理的是集合（set）之间的关系，一般处理的是图的连通分量，当给出两个的元素的一个无序对 (a,b) 时，需要快速合并(union) a 和 b 所在的集合，这期间需要反复查找(find)某元素的集合。

### 并查集的操作：

1. Make(x)：建立一个新的集合，其仅有成员是 x(集合代表)，由于各集合是分离的，要求 x 没有在其他集合出现过；
2. Union(x,y)：将包含 x、y 的动态集合(Sx、Sy)合并为一个新的集合S。新集合 S 的集合代表是 Sx U Sy 的某个成员；
3. Find(x)：返回一个指向包含 x 的集合的代表。

### 具体操作

1. 初始化

for(int i =1; i <= n; i++)//每个元素属于单独的一个集合，所以每个元素以自己作为自己的根节点

father[i] = i;

1. 寻找根节点编号并压缩路径

int Find(int x)//非递归实现

{

while (father[x] != x)

x = father[x];

return x;

}

int Find(int x)//递归实现—未优化版本

{

if (father[x] != x)

return Find(father[x]);

else

return father[x];

}

int Find(int x)//递归实现—优化版本

{

if (father[x] != x)

father[x] = Find(father[x]);//路径压缩，优化核心

return father[x];

}

1. 合并两集合

void Union(int x,int y)

{

x=Find(x);

y=Find(y);

if(x!=y)

father[y]=x;

}

1. 判断元素是否属于同一集合

bool judge(int x, int y)

{

x = Find(x);

y = Find(y);

if (x == y)

return true;

else

return false;

}

1. 统计集合数目

int count\_sets(int n)

{

int cnt=0;

for(int i =1; i <= n; i++)

if (father[i] == i)

cnt++;

return cnt;

}

1. 统计每个集合中元素的个数

void countElements()

{

for (int i =1; i <= n; i++)

{

father[i] = Find(i);//寻找每个节点的父节点

num[father[i]]++;//统计父节点下的节点个数

}

for(int i =1; i <= n; i++)//统计父节点外的点的个数

num[i] = num[father[i]];

}

# 图论

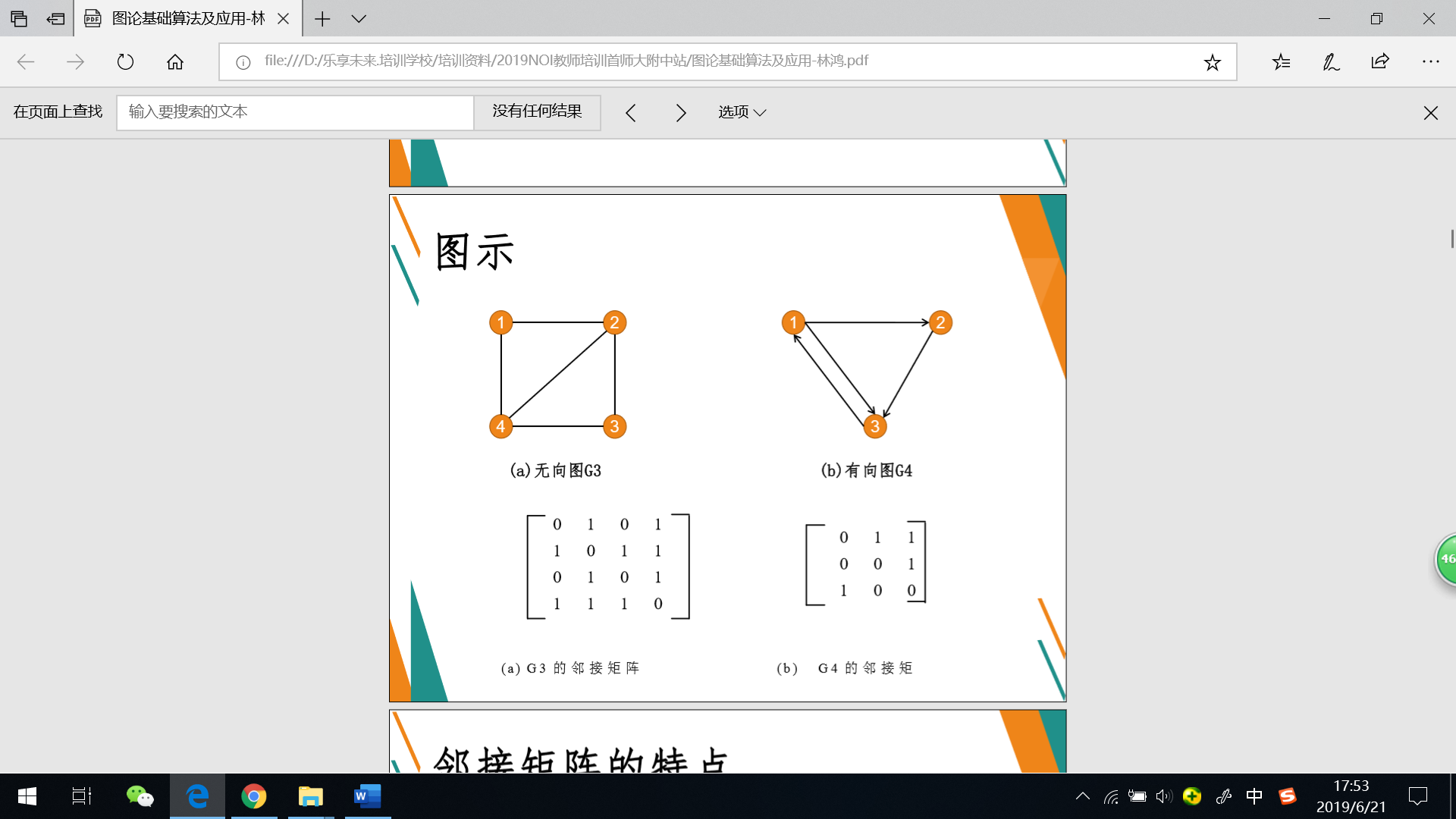
## 基本概念（参考2.4）

## 存储

### 邻接矩阵

二维数组g[i][j]：

* 设图中有n个点，用一个n\*n的矩阵(二维数组)g保存图的信息。
* g[i][j]=以i为起点，j为终点的边数(没有权值)或 g[i][j]=以i为起点，j为终点的权值；
* g[i][j] = 0表示不存在边，否则存在边。

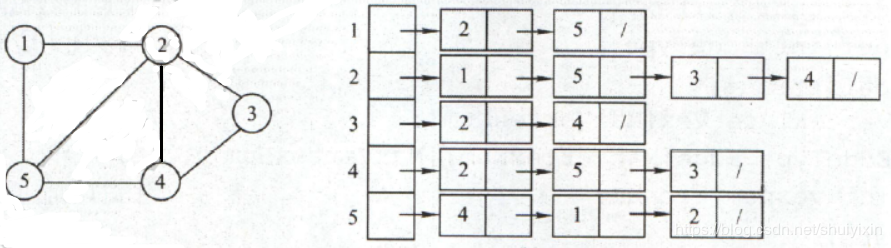
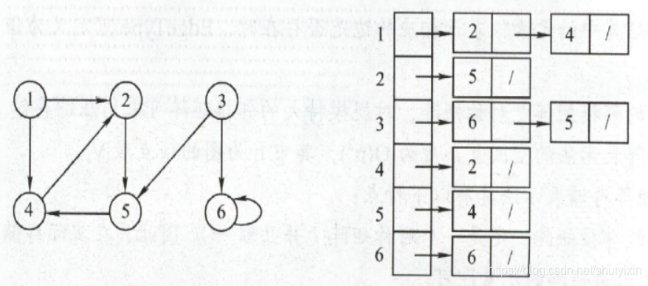


注意：

* 需要 O(n2) 的存储空间，对于 n = 100000 级别不适用；
* 用 0 还是用-1？g[i][j] 设成什么值？视具体情况决定；
* 可能有重边时，注意如何读入 g[i][j]+=1; or g[i][j]=1； 。

### 边表（邻接表）

边表又称为邻接表，每个结点的邻居形成一个链表。

无向图 有向图

举例：

4 5

1 4 9

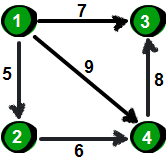
4 3 8

1 2 5

2 4 6

1 3 7

第一行两个整数n m。n表示顶点个数（顶点编号为1~n），m表示边的条数。接下来m行表示，每行有3个数x y z，表示顶点x到顶点y的边的权值为z。



## 遍历

遍历：以一定的顺序访问图的所有点和边，是图论算法的基础，对有向图和无向图都适用。两种图遍历算法: 宽度优先遍历 (BFS) 和深度优先遍历 (DFS)

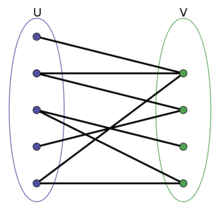
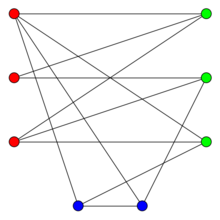
### BFS（宽度/广度优先遍历）

### DFS（深度优先遍历）

## 二分图

 二分图也称二部图，是图论里的一种特殊模型，也是一种特殊的网络流。其最大的特点在于，可以将图里的顶点分为两个集合，且集合内的点没有直接关联。

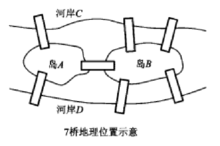
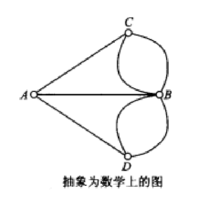
简而言之，就是顶点集V可分割为两个互不相交的子集，并且图中每条边依附的两个顶点都分属于这两个互不相交的子集，两个子集内的顶点不相邻。

二分图 非二分图

## 欧拉回路

欧拉回路是数学家欧拉在研究著名的德国哥尼斯堡(Koenigsberg)七桥问题时发现的。如下图所示，流经哥尼斯堡的普雷格尔河中有两个岛，两个岛与两岸共4处陆地通过7座桥彼此相联。7桥问题就是如何能从任一处陆地出发，经过且经过每个桥一次后回到原出发点。

* **相关概念**

欧拉路径（欧拉通路）：通过图中所有边的简单路。（换句话说，每条边都通过且仅通过一次）也叫”一笔画”问题。

欧拉回路：闭合的欧拉路径。（即一个环，保证每条边都通过且仅通过一次）。

欧拉图：包含欧拉回路的图。

* **欧拉回路判断条件**

无向图存在欧拉回路的充要条件

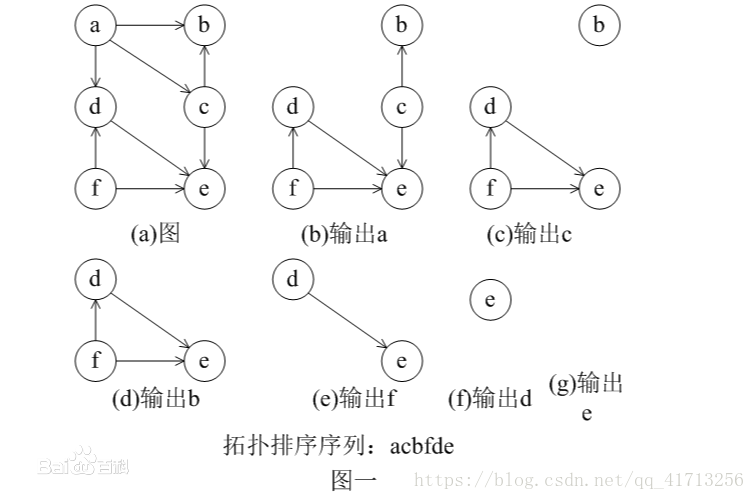
一个无向图存在欧拉回路，当且仅当该图所有顶点度数都为偶数,且该图是连通图。

有向图存在欧拉回路的充要条件

一个有向图存在欧拉回路，所有顶点的入度等于出度且该图是连通图。

## 拓扑排序

对一个[有向无环图](https://baike.baidu.com/item/%E6%9C%89%E5%90%91%E6%97%A0%E7%8E%AF%E5%9B%BE/10972513" \t "_blank)(Directed Acyclic Graph简称DAG)G进行拓扑排序，是将G中所有顶点排成一个线性序列。



示例代码：拓扑排序.cpp

## 树上倍增、LCA（最近公共祖先）

倍增的思想是二进制，思路是：开一个n×logn的数组，比如fa[n][logn],其中fa[i][j]表示**i节点的第2^j个父亲是谁**。

**fa[i][j] = fa[fa[i][j - 1]][j - 1]**

**i的第2^j个父亲 是i的第2^(j - 1)个父亲的第2^(j - 1)个父亲。**

**代码示例：LCA.cpp。**

## 最短路

最短路：找出一个点到另一个点的最短距离。

单源最短路径：单源就是从一个点到所有其他点的最短路径。

多源最短路径：图中所有点对（一对点）间最短路径。

### Dijkstra 算法（单源最短路径）

迪杰斯特拉(**Dijkstra**)算法是典型最短路径算法，用于计算一个节点到其他节点的最短路径。

Dijkstra算法是**贪心思想+BFS**实现的，首先把起点到所有点的距离存下来找个最短的，然后松弛一次再找出最短的，所谓的松弛操作就是，遍历一遍看通过刚刚找到的距离最短的点作为中转站会不会更近，如果更近了就更新距离，这样把所有的点找遍之后就存下了起点到其他所有点的最短距离。即：以起始点为中心向外层层扩展(BFS广度优先搜索思想)，直到扩展到终点为止。

松弛操作就是更新两点间的最短路径。

注：Dijkstra算法适用于**边权为正**的无向和有向图，**不适用**于有**负边权**的图！

示例代码：

int dijkstra(int n)

{

//初始化v[0]到v[i]的距离

for (int i=1; i <= n; i++)

dis[i] = w[0][i];

vis[0]=1;//标记v[0]点

for (int i = 1; i <= n; i++)

{

//查找最近点

int min = INF,k = 0;

for (int j = 0; j <= n; j++)

if (!vis[w] && dis[j] < min)

min = dis[w],k = j;

vis[k] = 1;//标记查找到的最近点

//判断是直接v[0]连接v[j]短，还是经过v[k]连接v[j]更短

for (int j = 1; j <= n; j++)

if (!vis[j] && min + w[k][j] < dis[j])

d[j] = min + w[k][j];

}

return dis[j];

}

### Floyd（多源最短路径）

Floyd算法又称为插点法，是一种利用动态规划的思想寻找给定的加权图中多源点之间最短路径的算法，与Dijkstra算法类似。该算法名称以创始人之一、1978年图灵奖获得者、斯坦福大学计算机科学系教授罗伯特·弗洛伊德命名。

Floyd算法适用于APSP(All Pairs Shortest Paths，多源最短路径)，是一种动态规划算法，稠密图效果最佳，边权可正可负。此算法简单有效，由于三重循环结构紧凑，对于稠密图，效率要高于执行|V|次Dijkstra算法，也要高于执行|V|次SPFA算法。

优点：容易理解，可以算出任意两个节点之间的最短距离，代码编写简单。

缺点：时间复杂度比较高，不适合计算大量数据。

* **算法实现步骤：**

1. 用邻接矩阵来建图，map[i][j]表示从i点到j点的距离，把自己到自己设为0，把自己到不了的边初始化为无穷大，代码为：

//初始化

for (int i =1; i<= n; i++)

for (int j = 1; j<= n; j++)

if (i == j)

map[i][j] = 0;

else

map[i][j] = inf;

//读入边

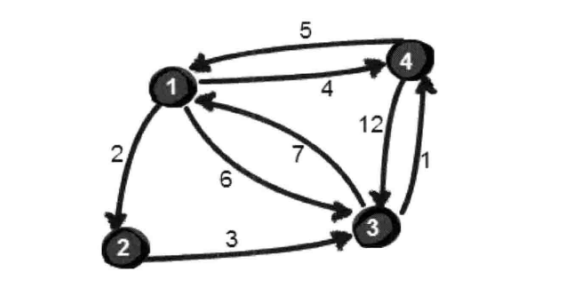
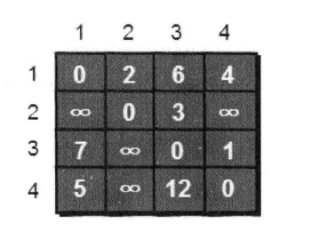
for(int i = 1; i<= m; i++)

{

cin >> t1 >> t2 >> t3;

map[t1][t2] = t3;

}

1. 遍历每一个点，并且以每一个点作为中转点看看它的值会不会改变，就可以得到从一个点到任意一个点的最短路径，也就是多源最短路。代码为：

for(int k = 1; k <= n; k++)

for(int i = 1; i <= n; i++)

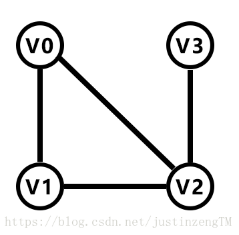
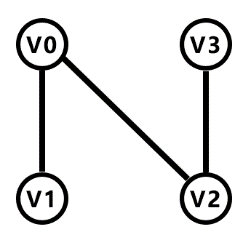
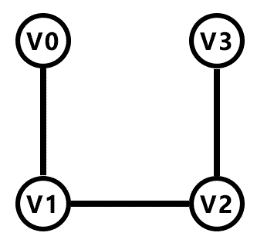
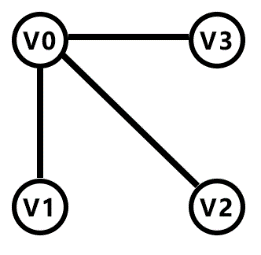
for(int j = 1; j<= n; j++)

if(map[i][k] + map[k][j ]< map[i][j])

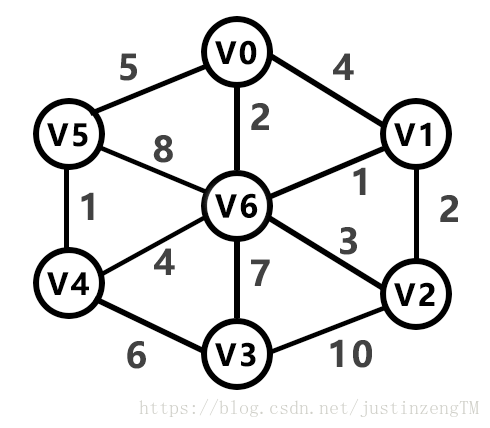
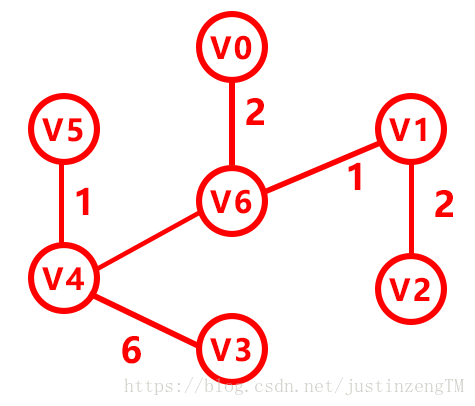
map[i][j] = map[i][k] + map[k][j];

## 最小生成树

将给出的所有点连接起来（即从一个点可到任意一个点），且连接路径之和最小的图叫最小生成树。

无向图 最小生成树

带权图 最小生成树

实现思路：将点分为在树中的点与不在树中的点，每次取出树中点的连接的最小路径，且该路径连接的点不在树中，然后将该路径连接的点加入树中，重复并进行路径更新，即松弛，当取出边达到n-1条时，树已建立。

最小生成树可以用kruskal（克鲁斯卡尔）算法或prim（普里姆）算法求出。

示例代码：kruskal.cpp、prim.cpp