第5章 图

由于图刚开始接触时，会被一大堆概念搞晕，不清楚顺着一条什么思路去学这一节，
因此，本节去繁就简，按照如下思路快速掌握图的基本概念，把下面搞懂后，深入学习不再是难题。

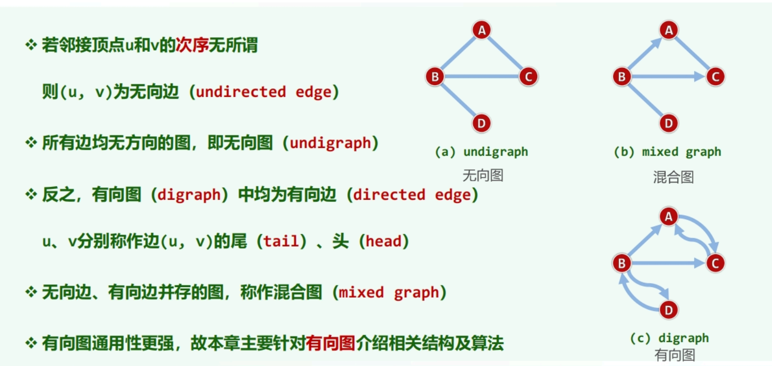
首先，讲解图的分类，到底有哪几种图，并且这里仅牵扯到一种分类方式，其他更多的概念目前都不需要，
不要看多了给自己搞混了，这里只要求看懂这是公猫还是母猫，就这么简单，更多的猫身上的细节，暂时不用管。

其次，上面介绍的几种图，如何进行表示，从具体的看得见的东西，怎么合理的表示出来，就跟前面学树的时候
一样，树是通过节点，以及节点之间的连接进行表示的，节点用结构体表示，连接关系用节点里面的指针表示，
这里的图，该如何表示呢？

经过上面两步，你懂了一个摆在面前可以直接画在纸上的具体的图，也懂了它的抽象表示方法，最后，介绍
在这种表示方法下，需要掌握的几个算法。这里仅仅只提最简单的几个算法，正如前面讲的，这里重在入门图的表示及算法！

一、图的分类

根据边有没有方向，分为有向图，无向图，和混合图，如下所示



二、图的表示

在树里面，两个关键要素就是节点信息+连接关系，但是到了图这里，由于实际问题需要，我们这里需要存储3个
关键要素：顶点信息+边信息+连接关系（简单起见，这里的信息就单指data，比如顶点信息就是人名，边信息就是借钱金额）
顶点信息和边的信息我们很容易表示，就是定义两个结构体，然后把信息录入到结构体里面不就完了，
但是连接关系如何表示呢，正如我在前面讲的，使用指针可以很好的表示连接关系，但是我们这里跳过这种方法，
我们这里介绍一种邻接矩阵的表示方法，先重点掌握其中一种表示方法入门再说。
等我们介绍完这种表示及算法后，然后再简单提一下，如果换一种方式表示，算法接口该怎么重构。

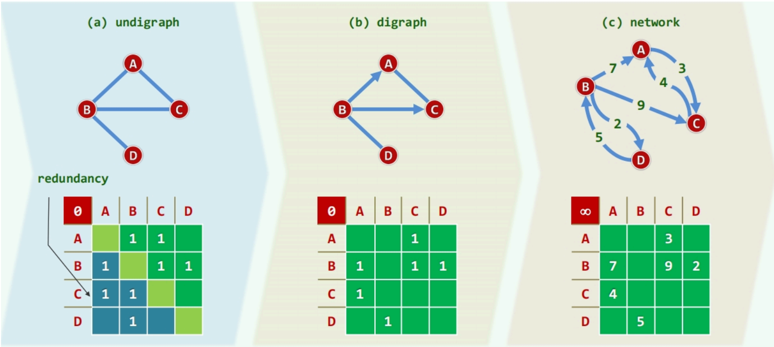
图的邻接矩阵表示(稠密图)

接着上面说的，每个顶点信息，都存储在结构体节点中，每条边也都在结构体节点中，此时，我们把所有顶点的结构体，
存到一个一维向量中，用这个向量管理访问所有顶点，这样每个顶点就对应一个下标，访问每个顶点只需要O(1)的复杂度，
比如顶点A放在向量的第0号位置，顶点B放在向量的1号位置，访问A和B分别用下标0和1访问即可；

然后对于边，我们把这些边存储在一个二维的向量中，具体怎么放呢，如下图，看第一张图，第一行表示从顶点A出发的所有边，
首先没有A-A这条边，于是第一行第一个元素为0或者NULL，然后有A-B和A-C这两条边，那么就把这两条边的结构体存储到第一行
的后面两个位置，然后一个位置表示的边不存在，同样用NULL表示；需要注意的是，下面的矩阵在有边的地方用1表示，和我们讲的
略有区别。

也就是说，如果把边的信息存储在一个二维向量中，那么所有的连接关系也自然清晰明了，比如C行，B列的位置为1，说明存在从顶点C
到顶点B的路径，至此，表示图的三个要素我们都搞定了，这里就叫图的邻接矩阵表示

为什么树不用这种表示方式？因为有n个点，就会产生nxn的矩阵，对于树，每个点的连接不超过3个，导致这个矩阵是超级稀疏矩阵，浪
费空间



三、图的定义及算法

In [1]:
```
#include <iostream>
#include <cstdio>
using namespace std;
#include "queue.h"
#include "stack.h"
```

3.1 顶点和边的定义

由于这里仅仅只介绍最简单的概念及算法，因此顶点和边就简单设置一个data，
不过对于顶点，遍历的时候涉及到状态的变化需要检测才能继续游走，所以给顶点一个status属性。

In [2]:
```
enum VStatus{Undiscovered,Discovered,Visited};
```

In [3]:
```
template <class Tv>
struct Vertex{
public:
    Tv data;
    VStatus status;
    Vertex(Tv v = 0):data(v), status(Undiscovered){}
};
```

In [4]:
```
template <class Te>
struct Edge{
public:
    Te data;
    Edge(Te e = 0):data(e){}
};
```

3.2 图的定义及基本属性

正如前面所说，这里的图本质就是两张表，一张表存顶点信息，一张表存边信息以及连接信息，连接信息是通过元素位置体现的；
因此这里定义的图，仅仅就是两张表，顺带加上这两张表的尺寸，用于描述这个图的顶点个数以及边的个数。

In [5]:
```
template <class Tv, class Te>
class GraphMatrix{
public:
    Vector< Vertex<Tv> > V;              // 第一张表V，存顶点
    Vector< Vector< Edge<Te>* > > E;  // 第二张表E，存边的指针
    int sizeV;                         // 顶点的个数
    int sizeE;                         // 边的条数

    // 构造函数
    GraphMatrix();

    // 基本属性
    VStatus& status(int i);
    bool exist(int i, int j);
    int firstNbr(int i);
    int nextNbr(int u, int v);

    // 边的插入删除
    void insert(Te e, int u, int v);
    Te remove(int u, int v);

    // 顶点的插入删除
    void insert(Tv v);

    // 打印
    void print();
};
```

构造函数
前面声明的两个向量V和E，每个占用16个字节(data指针8字节，size整数4字节，capacity整数4字节)，

然后sizeV和sizeE各4字节，一共16+16+4+4=40字节，并没有存储实际的东西，只是声明，初始化的东西也不是我们想要的，需要手动初始化。

In [6]:
```
template <class Tv, class Te>
GraphMatrix<Tv, Te>::GraphMatrix() {
    sizeV = 0;
    sizeE = 0;
    V = Vector<Vertex<Tv>>(0);
    E = Vector<Vector<Edge<Te>*>>(0);
}
```

基本属性
主要是一些状态判断，包括顶点是否被发现被访问，边是否存在，以及找出顶点的邻居，
这些基本性质，在后面其他算法中会经常用到，所以提前写好。

此外，由于顶点和边已经存入了向量当中，因此对顶点和边全是通过秩来访问的，下标秩就是顶点和边的唯一标识。

In [7]:
```
template <class Tv, class Te>
VStatus& GraphMatrix<Tv,Te>::status(int i){
    return V[i].status;
}
```

In [8]:
```
template <class Tv, class Te>
bool GraphMatrix<Tv,Te>::exist(int i, int j){
    if (E[i][j]) return true;
    return false;
}
```

In [9]:
```
template <class Tv, class Te>
int GraphMatrix<Tv,Te>::nextNbr(int u, int v){
    for (int i = v - 1; i >= 0; i--)
        if (exist(u,i)) return i;
    return -1;
}
```

In [10]:
```
template <class Tv, class Te>
int GraphMatrix<Tv,Te>::firstNbr(int i){
    return nextNbr(i,sizeV);
}
```

3.3 图的动态操作之插入删除

边的插入

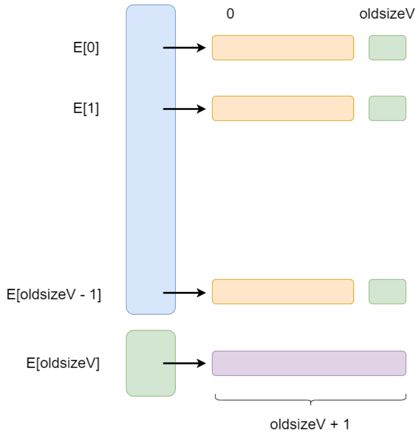
In [11]:
```
template <class Tv, class Te>
void GraphMatrix<Tv,Te>::insert(Te e, int u, int v){
    if (exist(u,v)) return;
    E[u][v] = new Edge<Te>(e);
    sizeE++;
}
```

边的删除

In [12]:
```
template <class Tv, class Te>
Te GraphMatrix<Tv,Te>::remove(int u, int v){
    if (!exist(u,v)) return;
```

```
    Te BAK = E[u][v] -> data;
    delete E[u][v];
    E[u][v] = NULL;
    sizeE--;
    return BAK;
}
```

顶点的插入
首先这个被插入的顶点作为最后一个编号，在顶点向量中，直接把顶点插入最后即可，
对于边，需要增加一行和一列，具体操作如下：
首先在最外层的向量中增加一个元素，这个元素就是内层的向量，然后把这个向量初始化为NULL，
然后依次把前面的每个向量都加长一个单位，每个加长的位置用NULL初始化
这样插入顶点后，这个顶点与其他顶点没有任何连接，邻接矩阵中这个顶点的行和列均为NULL；
值得注意的是，这个矩阵不是正方形的，两个维度的长度完全是分离的，比如外面的向量存储的元素
就是向量类型，但是这些向量长度可以不一样。操作的时候，一般先对外层操作，然后再操作每一个小向量



In [13]:
```
template <class Tv, class Te>
void GraphMatrix<Tv,Te>::insert(Tv v){
    int oldsizeV = sizeV;
    ++sizeV;                           // 顶点数+1

    for (int i = 0; i < oldsizeV; i++)
        E[i].insert(oldsizeV, NULL); // 给前面每个向量加一个元素，初始化为NULL

    E.insert(oldsizeV,Vector< Edge<Te>* >(oldsizeV+1)); // 矩阵添加一行到最后
    for (int i = 0; i < oldsizeV+1; i++) E[oldsizeV][i] = NULL;          // 令最后一行为NULL

    V.insert(oldsizeV,Vertex<Tv>(v));     // 将顶点插入V的最后一个位置

    return;
}
```

顶点的删除
由于构造一个图，我们只用的到顶点和边的插入操作，这里顶点删除算法就不写了，
析构函数就是删除所有的顶点和所有的边，这里也不写了，程序完全终结的时候内存自动释放

打印图
后面我们会构造一个图，然后想输出来看看构造的对不对，这里额外写一个打印函数，另外，xeus cling这里，
运行到 `E[oldsizeV][i] = NULL` 内核就会崩溃，即便把这句写到Vector初始化当中，依旧没用，
然后我尝试在main函数中运行代码，内核不会崩溃，但是没有任何输出显示，于是我把print函数改成输出到文件，
但是文件并没有被创建，总之尝试了各种方法，搜寻了很多，依旧无解。

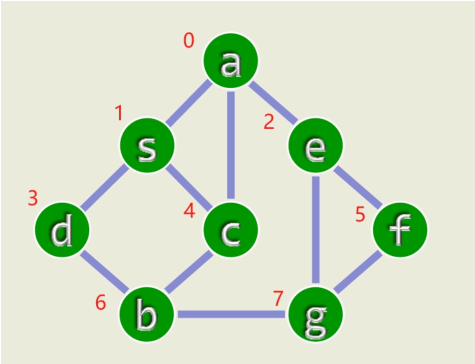
最后的办法，把这里的程序拷贝到DEV C++中运行，然后把运行结果截图到这里，后面都按照这种方式进行

In [14]:
```
template <class Tv, class Te>
void GraphMatrix<Tv,Te>::print(){
    cout << "Vertex:" << endl;
    for (int i = 0; i < sizeV; i++){
        cout << V[i].data << " ";
    }
    cout << endl;

    cout << "Edge:" << endl;
    for (int i = 0; i < sizeV; i++){
        for (int j = 0; j < sizeV; j++){
            if (exist(i,j)) cout << "1" << " ";
            else cout << "0" << " ";
        }
        cout << endl;
    }
}
```

3.4 构造一个图

按照如下顺序构造一个图



代码如下：

```
GraphMatrix<char,int> G1;
G1.insert('a');
G1.insert('s');
G1.insert('e');
G1.insert('d');
G1.insert('c');
G1.insert('f');
G1.insert('b');
G1.insert('g');
G1.print();
```

运行结果如下：

```
Vertex:
a s e d c f b g
Edge:
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
0 0 0 0 0 0 0 0
```

从运行结果可以看出来，我们加入了8个节点，并且邻接矩阵此时所有的边都是空的，
下面我们按照上面的图插入边，边的数值统一设置为0好了，代码如下：

```
G1.insert(0,0,1);
G1.insert(0,0,4);
G1.insert(0,0,2);

G1.insert(0,1,0);
G1.insert(0,1,3);
G1.insert(0,1,4);

G1.insert(0,2,0);
G1.insert(0,2,5);
G1.insert(0,2,7);

G1.insert(0,3,1);
G1.insert(0,3,6);

G1.insert(0,4,0);
G1.insert(0,4,1);
G1.insert(0,4,6);

G1.insert(0,5,2);
G1.insert(0,5,7);

G1.insert(0,6,3);
G1.insert(0,6,4);
G1.insert(0,6,7);

G1.insert(0,7,2);
```

```
    G1.insert(0,7,5);
    G1.insert(0,7,6);
    G1.print();
```

运行结果如下:

```
Vertex:
a s e d c f b g
Edge:
0 1 1 0 1 0 0 0
1 0 0 1 1 0 0 0
1 0 0 0 0 1 0 1
0 1 0 0 0 0 1 0
1 1 0 0 0 0 1 0
0 0 1 0 0 0 0 1
0 0 0 1 1 0 0 1
0 0 1 0 0 1 1 0
```

3.5 图的遍历

图的遍历主要有BFS,DFS和PFS，由于时间问题，这里只介绍BFS和DFS，
沿用之前二叉树的习惯，我们这里算法直接写在类的外面，不进行封装了

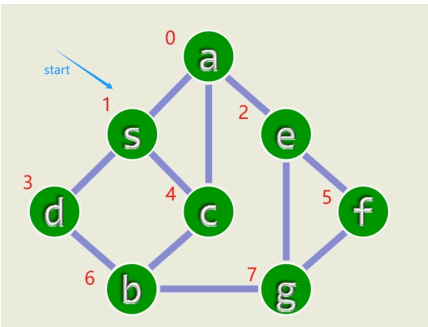
BFS(广度优先遍历，Breadth First Search)

算法思路:
首先所有顶点都是Undiscovered，未发现状态，然后我们从一个起始顶点出发，把顶点设为Visited状态，然后向四周环顾，
通过nextNbr发现所有的邻居，被发现的邻居首先状态要变成Discovered状态并加入一个队列，以下图为例，我们从s出发，
发现了邻居c,d,a，然后c,d,a依次入队并变为Discovered状态，ok，接下来就是把队列中的元素一个一个出队，出队的时候，
需要以当前出队的点为起始点，做上面从s出发做的同样的事情，就是环顾四周，看看有没有未发现状态的点，有就加入队列并设为
Discovered状态，比如c出队的时候，先设为Visited状态，然后依次找邻居b,s,a，但是只有b是Undiscovered，只有它才能入队，
处理完c的邻居后，队伍中紧接着就是d，然后d出队做同样的事，直到队列中一个不剩，所有的点就都被访问了一遍。
上述过程，所有的点从Undiscovered变成Discovered（入队），再变成Visited（出队），经历了三种状态。

In [15]:

```
template <class Tv, class Te>
void BFS(GraphMatrix<Tv,Te> &G1, int v){
    Queue<int> q;
    int u;
    int Nbr;
    q.enqueue(v);
    G1.V[v].status = Discovered;
    while(!q.empty()){
        u = q.dequeue();
        cout << G1.V[u].data << " ";  // 这里visit就是输出操作
        G1.V[u].status = Visited;
        for(Nbr = G1.firstNbr(u); Nbr != -1; Nbr = G1.nextNbr(u,Nbr)){
            if (G1.status(Nbr) == Undiscovered) {
                q.enqueue(Nbr);
                G1.V[Nbr].status = Discovered;
                        }
        }
    }
}
```

从顶点s出发，运行结果如下:




```
Vertex:
a s e d c f b g
Edge:
0 1 1 0 1 0 0 0
1 0 0 1 1 0 0 0
1 0 0 0 0 1 0 1
0 1 0 0 0 0 1 0
1 1 0 0 0 0 1 0
0 0 1 0 0 0 0 1
0 0 0 1 1 0 0 1
0 0 1 0 0 1 1 0

BFS:
s c d a b e g f
```

算法运行过程:

s 入队

s 出队(从大到小依次发现Undiscovered的邻居c，d，a)
c，d，a 入队，并标记为Discovered

c出队（其邻居仅b为Undiscovered）
b入队，并标记为Discovered

d出队（没有发现Undiscovered的邻居）

a出队（其邻居仅e为Undiscovered）
e入队，并标记为Discovered

b出队（其邻居仅g为Undiscovered）
g入队，并标记为Discovered

e出队（其邻居仅f为Undiscovered）
f入队，并标记为Discovered

g出队（没有发现Undiscovered的邻居）
f出队（没有发现Undiscovered的邻居）

出队序列依次是：
s, c, d, a, b, e, g, f

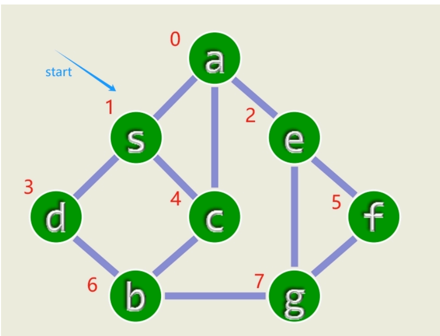
支撑树
实际上上述算法在遍历过程中，还对边的状态做了改变，最后筛选出保留的边，就会构成一棵树，这个就是支撑树，
但是前面讲了，我们这里仅入门，多余暂时用不上的东西一律忽略，这里全都没写

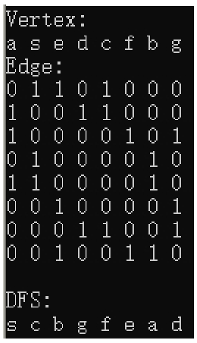
DFS(深度优先遍历，Depth First Search)

算法思路：
二叉树先序遍历完全一样，就是先访问当前节点，做点事情，然后挨个递归访问它的邻居(二叉树是它的左右孩子)
然后我看了书上是用递归实现的，也没写迭代形式，网上给的大多数也都是递归形式，所以就根据上面讲的几句话写出递归形式

In [16]:
```
template <class Tv, class Te>
void DFS(GraphMatrix<Tv,Te> &G1, int v){
    cout << G1.V[v].data << " ";  // 做点事情
    G1.V[v].status = Visited;
    for (int u = G1.firstNbr(v); u != -1; u = G1.nextNbr(v,u)) {
          if (G1.status(u) == Undiscovered) {
              G1.V[u].status = Discovered;      // 递归进来了，相当于入队或者入栈了，后面千万不能再次进来，否则死循环
              DFS(G1,u);
          }
    }
}
```

还是从顶点s出发，运行结果如下：





算法运行过程也十分简单，从s一路走到a，即
DFS(s) -> DFS(c) -> DFS(b) -> DFS(g) -> DFS(f) -> DFS(e) -> DFS(a)
然后回退，因为DFS(a)里面检查a所有邻居，找不到Undiscovered的了，函数执行结束，
回到上一层进来的地方，也就是DFS(e),a作为e的第二个邻居被递归DFS了(第一个已经Discovered了)
也就是在执行DFS(e)里面的第二个邻居也被执行结束了，这时候for循环找不到邻居了，于是DFS(e)
也执行完了，继续回溯到DFS(f),然后再回溯到DFS(g),再回到DFS(b)，回到DFS(b)的时候，for
已经执行完g这个邻居了，于是继续下一次for，找到了Undiscovered状态的d，于是：

执行了DFS(d),这时候继scbgfea之后，输出了d，因为一进去DFS，就执行输出。

后面不停回退，直到程序结束

四、换个表示方法，算法重构

这里简单提两句，除了上面讲的邻接矩阵表示法，图还有邻接表这种表示方法，并且更加节省空间，用了这种方法后，难道需要重构所有算法吗，

其实不是这样的，对于图的遍历算法，你会发现，都会固定调用那么几个图的基本操作，只需要把这几个图的基本操作接口重写一下，遍历算法
一行代码都不用改，可以直接使用。并且，调用库的时候，这几个基本图的操作，无论什么表示方法，这几个接口都是有的，所以这里写的
遍历算法几乎与图的表示无关，可以几乎不改直接用（访问方式需要改，邻接矩阵访问顶点和边都是循秩访问，给个数字即可，邻接表就不太一样了）。

In [ ]:

```

```

In [ ]:

```

```

In [ ]:

```

```

In [ ]:

```

```

In [ ]:

```

```