21级实验室暑假第三讲

目录

1. vector（变长数组）

- vector直接翻译为“向量”，一般说成“变长数组”，即“长度根据需要而自动改变的数组”；
- 在竞赛中，有些题目需要定义很大的数组，这样会出现“超出内存限制”的错误。比如，如果一个图的顶点太多，使用**邻接矩阵**就会**超出内存限制**，使用**指针实现邻接表**又很**容易出错**，而使用**vector**实现**简洁方便，还可以节省存储空间**；
- 它的内部实现基于**倍增**的思想，按照下面的思路大致可以实现vector；

设n,m为vector的实际长度和最大长度。向vector加入元素前，若n=m，则在内存中申请2m的连续空间，并把内容转移到新的地址上(同时释放旧的空间)，再执行插入。从vector中删除元素后，若 $n \le \frac{m}{4}$ 时，则释放一半的空间。

C++STL

1. vector（变长数组）

- 使用vector,需要添加vector的**头文件**

```
#include <vector>
using namespace std;
```

- vector的**定义**

```
vector <typename> name;
//定义一个一维数组name[size],size不确定,其长度可以根据需要而变化
//typename可以是任何基本类型,例如int ,double ,char ,结构体,也可以是vector

vector <int> a;//相当于一个长度动态变化的int数组
vector <double> b[233];//相当于第一维长233,第二维长度动态变化的double数组
struct rec{...};
vector <struct rec> c;//自定义的结构体类型保存在vector中
vector <vector<char> > d;//相当于一个一维和二维长度都是动态变化的char数组
```

1. vector（变长数组）

- vector的**访问** —— 一般通过**下标访问**或**迭代器访问**

✓ 第一种通过“下标”访问

对于容器 vector<int> v，可以使用 v[index]来访问它的第 index 个元素。其中，0≤index≤v.size()-1，v.size()表示 vector 中元素的个数

注: vector支持**随机访问**,即对于任意的下标0≤i<n,可以像数组一样用[i]取值。但它不是链表,不支持在任意位置O(1)插入。为了保证效率,元素的增删一般应该在末尾进行。

1. vector（变长数组）

- vector的**访问** —— 一般通过**下标访问**或**迭代器访问**

✓ 第二种通过“迭代器”访问

迭代器像STL容器中的“指针”，可以用星号‘*’操作符解除引用。

名字

一个保存为int的vector迭代器声明方式为: `vector<int> :: iterator it`

vector的迭代器是“随机访问迭代器",可以把vector的迭代器与一个整数相加减,其行为和指针的移动类似。可以把vector的两个迭代器相减,其结果和指针相减类似，得到两个迭代器对应下标之间的距离。

```
vector<int> v={1,2,3,4,5};
vector<int> :: iterator it1=v.begin();
vector<int> :: iterator it2=v.end();
for(int i=0;i<=4;i++)
    cout << *(it1+i) << ' ';
cout << endl << it2-it1 << endl;
```

输出结果:
1 2 3 4 5
5

C++STL

1. vector（变长数组）

- vector的**遍历方式**

```
vector<int> v={5,4,7,9,3,6};
//遍历方式1(auto 访问)
for(auto item:v)//有的C++较老版本不支持
    printf("%d ",item);
puts("");
//遍历方式2(下标访问)
for(int i=0;i<v.size();i++)
    printf("%d ",v[i]);
puts("");
//遍历方式3(迭代器访问)
for(vector <int> :: iterator i=v.begin();i!=v.end();i++)
    printf("%d ",*i);
```

1. vector（变长数组）

- vector的**常用函数**

➢ **size/empty**

size函数返回vector的实际长度(包含的元素个数)
empty函数返回一个bool类型，表明vector是否为空(为空，返回True；反之，返回False)
二者的时间复杂度均为O(1)

➢ **clear**

clear函数把vector清空，时间复杂度为O(N)

➢ **begin/end**

begin函数返回指向vector中第一个元素的迭代器，例如a是一个非空的vector，则*a.begin()和a[0]的作用相同
所有容器都可以视作一个**前闭后开**的结构，end函数返回vector的尾部，即第n个元素再完后的“边界”，*a.end()和a[n]都是越界访问，其中n=a.size()

➢ **front/back**

front函数返回vector的第一个元素，等价于*a.begin()和a[0]
back函数返回vector的最后一个元素，等价于*--a.end()和a[a.size()-1]

C++STL

1. vector（变长数组）

- vector的**常用函数**

➢ **push_back/pop_back**

a.push_back(x)把元素x插到vector a的尾部

a.pop_back()删除vector a的最后一个元素

➢ 用vector代替邻接表**保存有向图**

```
const int MAX_EDGES=100010;
vector <int> ver[MAX_EDGES],edge[MAX_EDGES];

void add(int x,int y,int z){//保存从x到y权值为z的有向边
    ver[x].push_back(y);
    edge[x].push_back(z);
}

for(int i=0;i<ver[x].size();i++){//遍历从x出发的所有边
    int y=ver[x][i];
    int z=edge[x][i];
       //有向边(x,y,z)
}
```

C++STL

➢ **保存有向图回顾:链式前向星**——以结构体保存或者不以结构体保存

链式前向星采用了一种静态链表存储方式，将**边集数组和邻接表**相结合，可以快速访问一个结点的所有邻接点，在算法竞赛中被广泛使用。

链式前向星的实现

边集数组：edge[ ]，edge[i] 表示第 i 条边；
头结点数组：head[ ]，head[i] 表示存储以 i 为起点的第 1 条边的下标（edge[ ] 中的下标）。

```
//非结构体方式
int ed[2*MAXM],val[2*MAXM],nex[2*MAXM];
int head[MAXN],idx=0;
```

```
//结构体方式
struct Edge{
    int ed;//该边的终点
    int val;//该边的权值
    int nex;//与上一个与起点相连的边
}edge[2*MAXM];//无向图记得开2倍

int head[MAXN],idx=0;
```

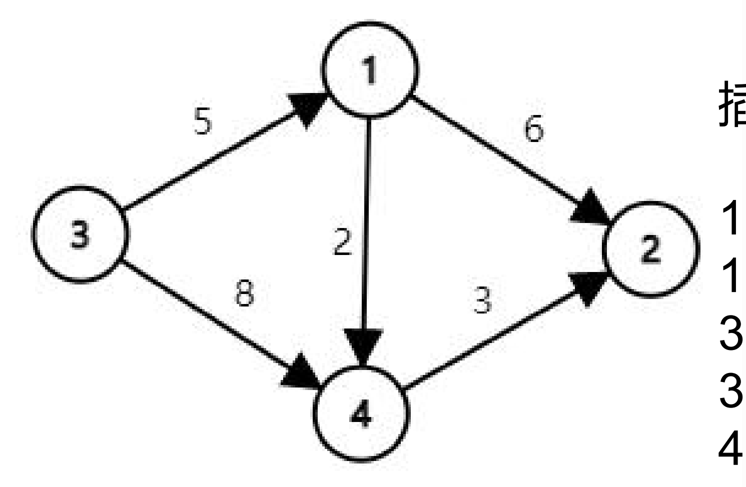
➢ **保存有向图回顾:链式前向星**——以结构体保存或者不以结构体保存

每条边的结构都如下图所示。

edge[i]	ed	val	nex

```
//结构体方式
struct Edge{
    int ed;//该边的终点
    int val;//该边的权值
    int nex;//与上一个与起点相连的边
}edge[2*MAXM];//无向图记得开2倍
int head[MAXN],idx=0;
```

以下图有向图为例，创建链式前向星如图。



插入顺序:

1 2 6
1 4 2
3 1 5
3 4 8
4 2 3

序号	head[]
1	1
2	-1
3	3
4	4

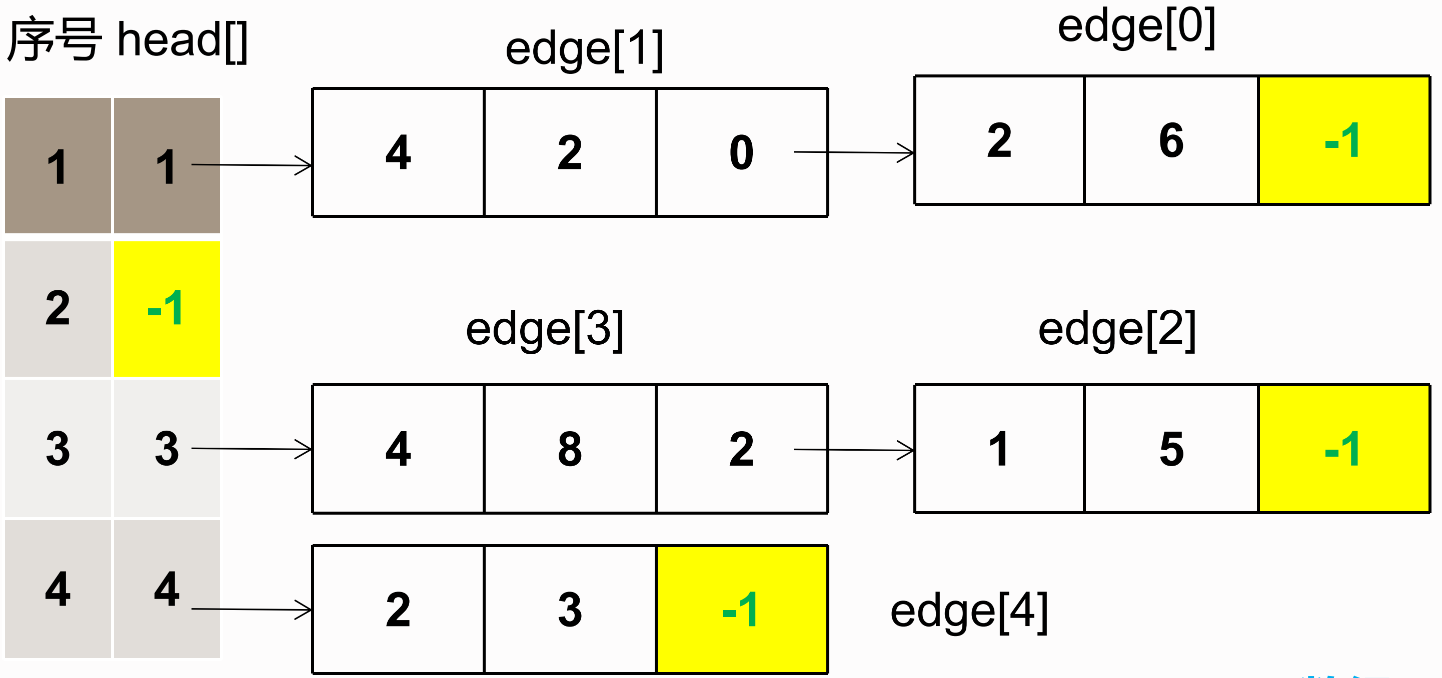
edge[1]: | 4 | 2 | 0 | → edge[0]: | 2 | 6 | -1 |

edge[3]: | 4 | 8 | 2 | → edge[2]: | 1 | 5 | -1 |

edge[4]: | 2 | 3 | -1 |

C++STL

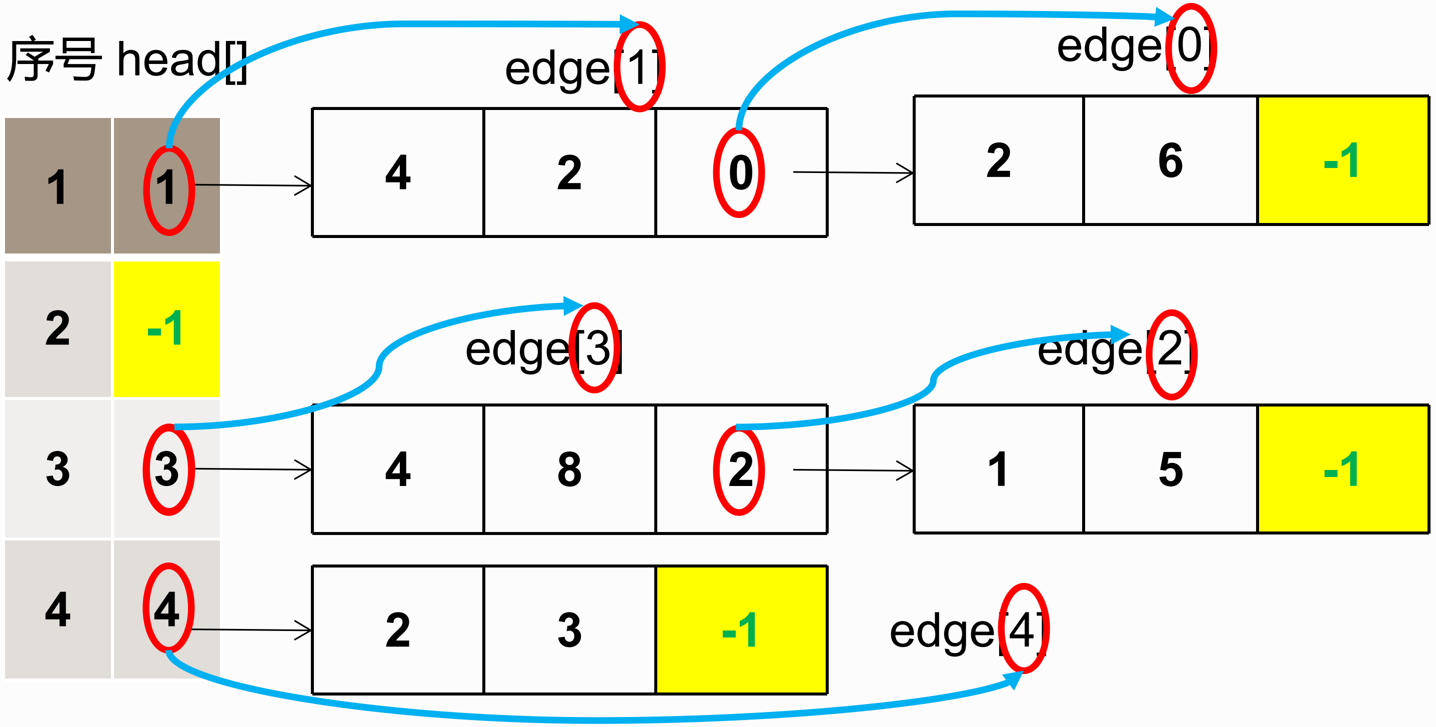
- **保存有向图回顾:链式前向星**——以结构体保存或者不以结构体保存



数组 head[] 和 egde[].nex 存的都是 edge[] 的索引

- 当 head[u] 或 edge[u].nex 为 -1 时，表示没有邻接点
- head[u] 的索引 u 表示第几个结点
- edge[ ] 存储边
- head[u] 存储 edge[ ] 中与该结点相关的边的索引（head[u] 存储 edge[ ] 的索引 i）
- edge[u].nex 存储与 u 相关的 edge[ ] 中其他边的索引

➢ **保存有向图回顾:链式前向星**——以结构体保存或者不以结构体保存



特性

- 和邻接表一样，因为链式前向星采用**头插法**进行链接，所以边的**输入顺序**不同，创建的链式前向星也不同
- 对于**无向图**，每输入一条边，都需要添加两条边，互为反向边
- 链式前向星具有边集数组和邻接表的功能，属于静态链表，**不需要频繁地创建结点，应用起来十分灵活**

C++STL

➢ **保存有向图回顾:链式前向星**——以结构体保存或者不以结构体保存

”头插法“完成添加操作

```
//结构体链式前向星
void add(int a,int b,int c){//idx 为 edge 索引
    edge[idx].ed=b;//保存终边
    edge[idx].val=c;//保存权值
    edge[idx].nex=head[a];//类似头插法，先取得头结点的信息，再更新头结点
    head[a]=idx++;//更新头结点
}
```

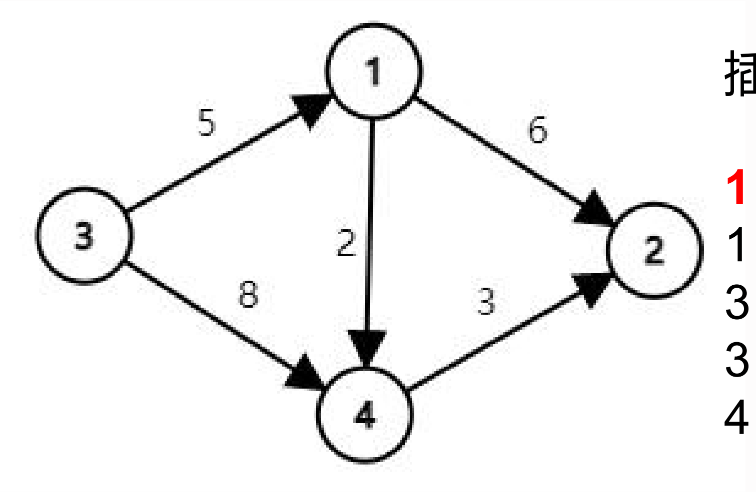
```
void add(int a,int b,int c){
    ed[idx]=b;
    val[idx]=c;
    nex[idx]=head[a];
    head[a]=idx++;
}
```

➢ **保存有向图回顾:链式前向星**——以结构体保存或者不以结构体保存

```
//结构体链式前向星
void add(int a,int b,int c){//idx 为 edge 索引
    edge[idx].ed=b;//保存终边
    edge[idx].val=c;//保存权值
    edge[idx].nex=head[a];//类似头插法，先取得头结点的信息，再更新头结点
    head[a]=idx++;//更新头结点
}
```

此部分及后面几页**有动画演示**可播放查看具体过程

还是以该图为例解释具体实现过程:



插入顺序:

1 2 6
1 4 2
3 1 5
3 4 8
4 2 3

序号	head[]
1	0
2	-1
3	-1
4	-1

	ed	val	nex
edge[0]	2	6	-1

当插入1 2 6 时，此时idx=0;
edge[0].ed=2;
edge[0].val=6;
edge[0].nex=head[1]=-1;//最开始点1无邻接点
head[1]=idx=0,之后idx+1=1;

C++STL

➢ **保存有向图回顾:链式前向星**——以结构体保存或者不以结构体保存

```
//结构体链式前向星
void add(int a,int b,int c){//idx 为 edge 索引
    edge[idx].ed=b;//保存终边
    edge[idx].val=c;//保存权值
    edge[idx].nex=head[a];//类似头插法，先取得头结点的信息，再更新头结点
    head[a]=idx++;//更新头结点
}
```

当插入1 4 2 时，此时idx=1;

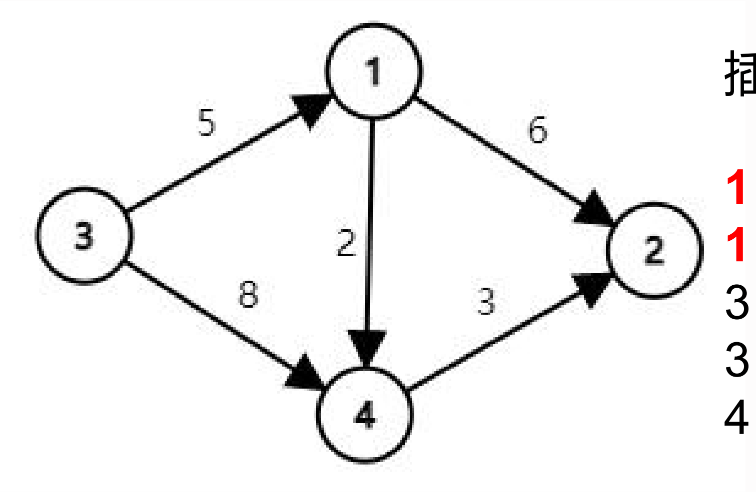
edge[1].ed=4;

edge[1].val=2;

edge[1].nex=head[1]=0;

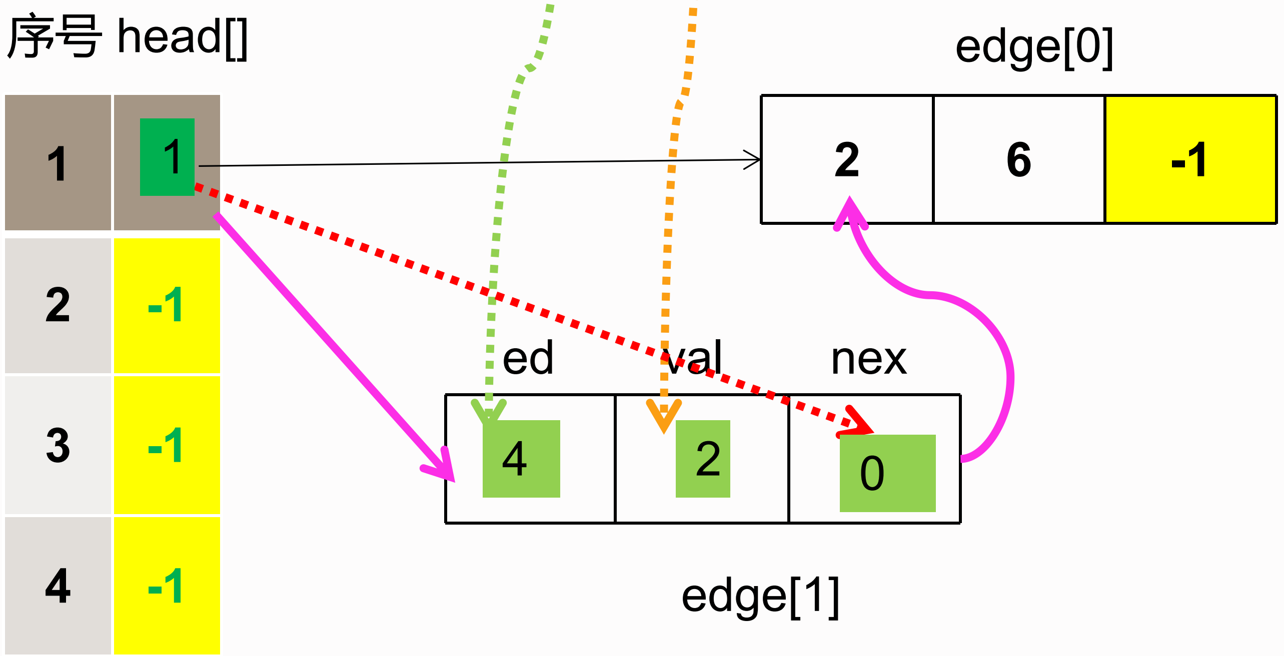
head[1]=idx=1,之后idx+1=2;

还是以该图为例解释具体实现过程:



插入顺序:

1 2 6
1 4 2
3 1 5
3 4 8
4 2 3



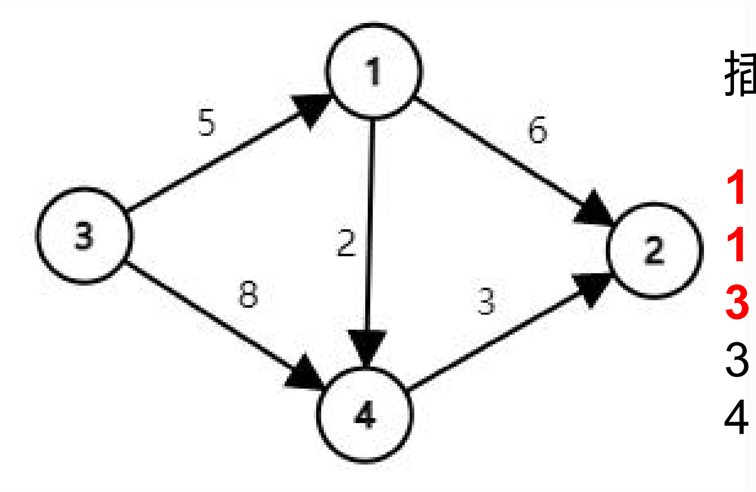
C++STL

➢ **保存有向图回顾:链式前向星**——以结构体保存或者不以结构体保存

```
//结构体链式前向星
void add(int a,int b,int c){//idx 为 edge 索引
    edge[idx].ed=b;//保存终边
    edge[idx].val=c;//保存权值
    edge[idx].nex=head[a];//类似头插法，先取得头结点的信息，再更新头结点
    head[a]=idx++;//更新头结点
}
```

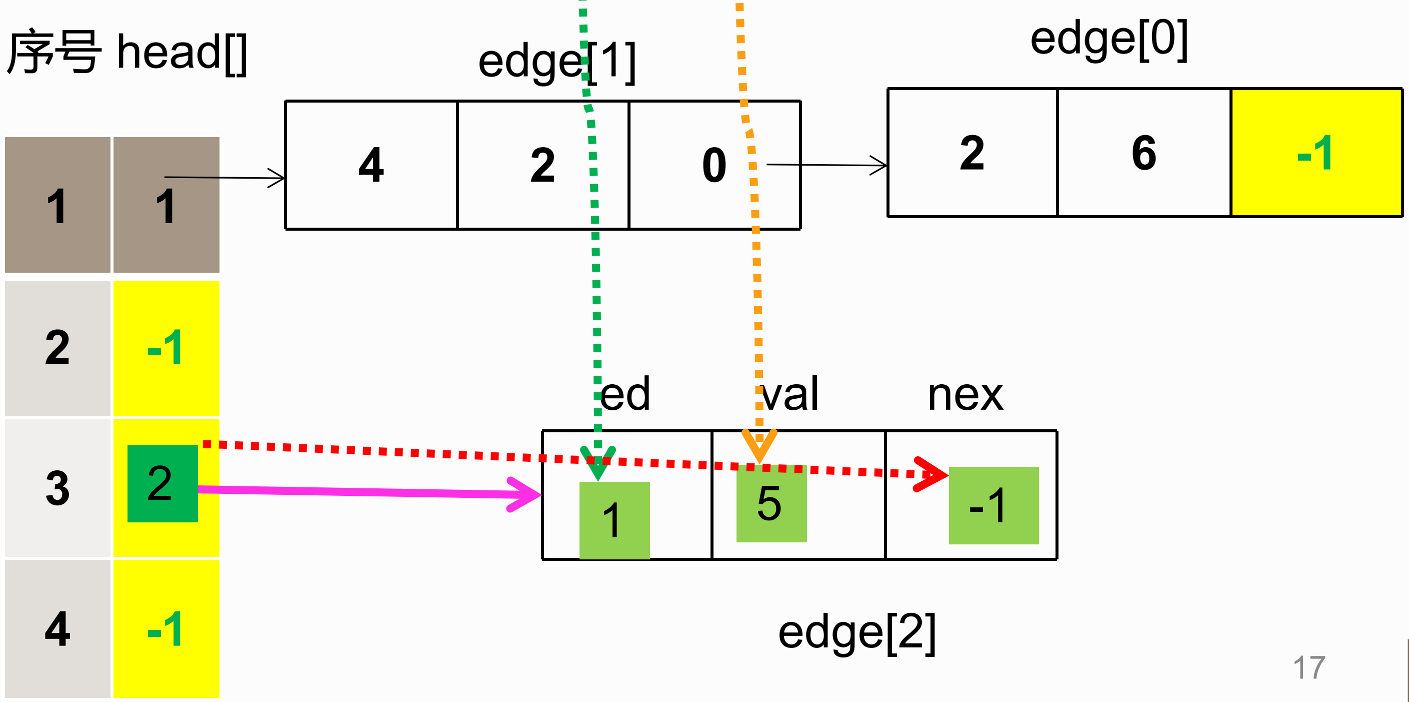
当插入3 1 5 时，此时idx=2;
edge[2].ed=1;
edge[2].val=5;
edge[2].nex=head[3]=-1;
head[3]=idx=2,之后idx+1=3;

还是以该图为例解释具体实现过程:



插入顺序:

1 2 6
1 4 2
3 1 5
3 4 8
4 2 3



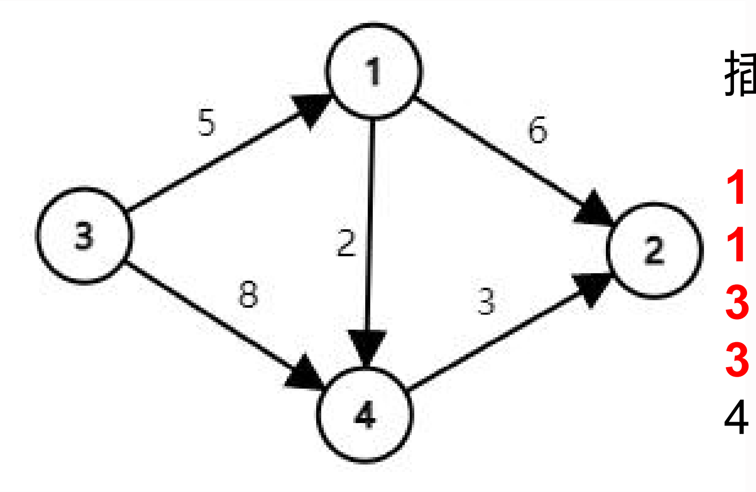
C++STL

➢ **保存有向图回顾:链式前向星**——以结构体保存或者不以结构体保存

```
//结构体链式前向星
void add(int a,int b,int c){//idx 为 edge 索引
    edge[idx].ed=b;//保存终边
    edge[idx].val=c;//保存权值
    edge[idx].nex=head[a];//类似头插法，先取得头结点的信息，再更新头结点
    head[a]=idx++;//更新头结点
}
```

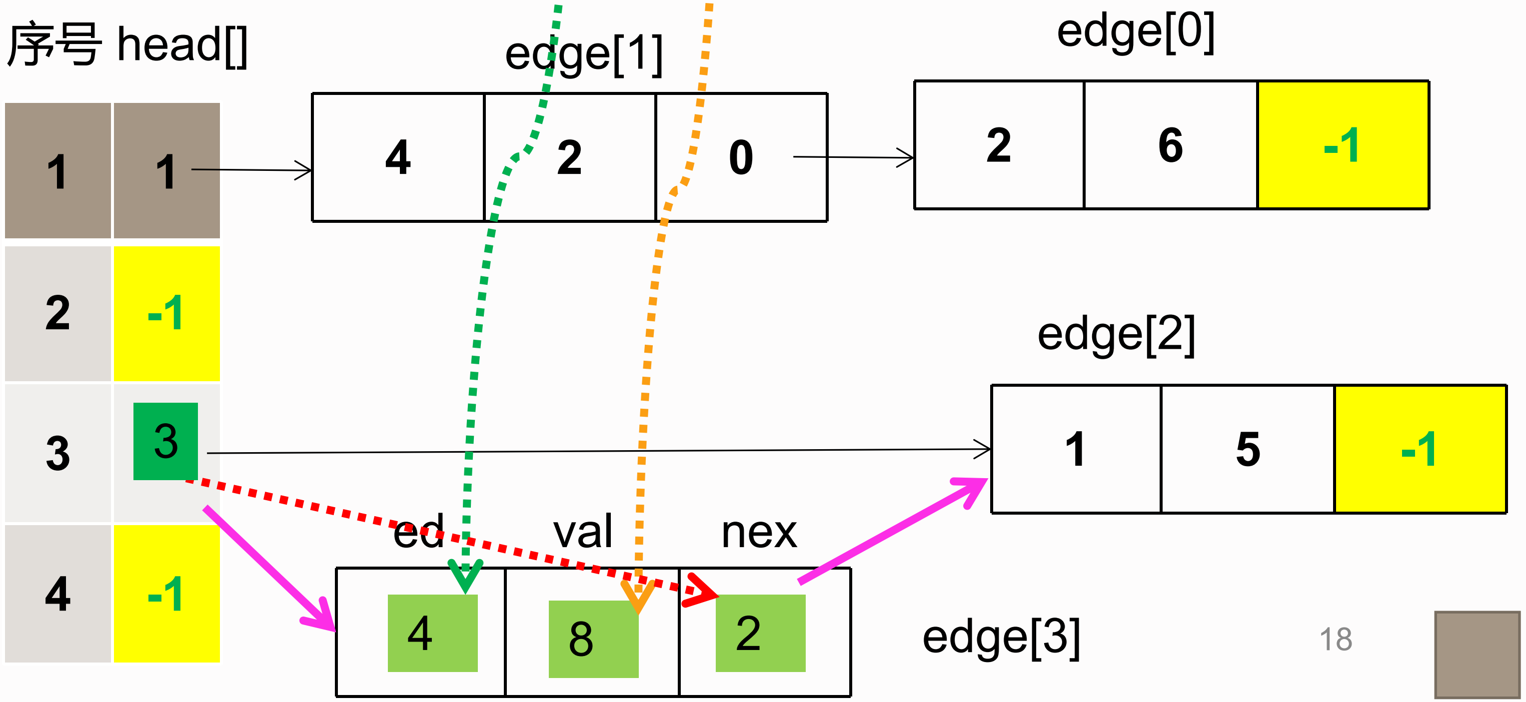
当插入3 4 8时，此时idx=3;
edge[3].ed=4;
edge[3].val=8;
edge[3].nex=head[3]=2;
head[3]=idx=3,之后idx+1=4;

还是以该图为例解释具体实现过程:



插入顺序:

1 2 6
1 4 2
3 1 5
3 4 8
4 2 3



C++STL

➢ **保存有向图回顾:链式前向星**——以结构体保存或者不以结构体保存

```
//结构体链式前向星
void add(int a,int b,int c){//idx 为 edge 索引
    edge[idx].ed=b;//保存终边
    edge[idx].val=c;//保存权值
    edge[idx].nex=head[a];//类似头插法，先取得头结点的信息，再更新头结点
    head[a]=idx++;//更新头结点
}
```

当插入4 2 3时，此时idx=4;

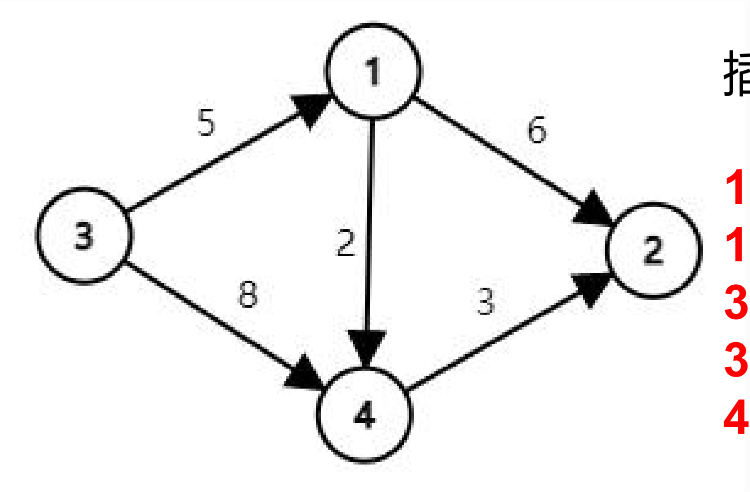
edge[4].ed=2;

edge[4].val=3;

edge[4].nex=head[4]=-1;

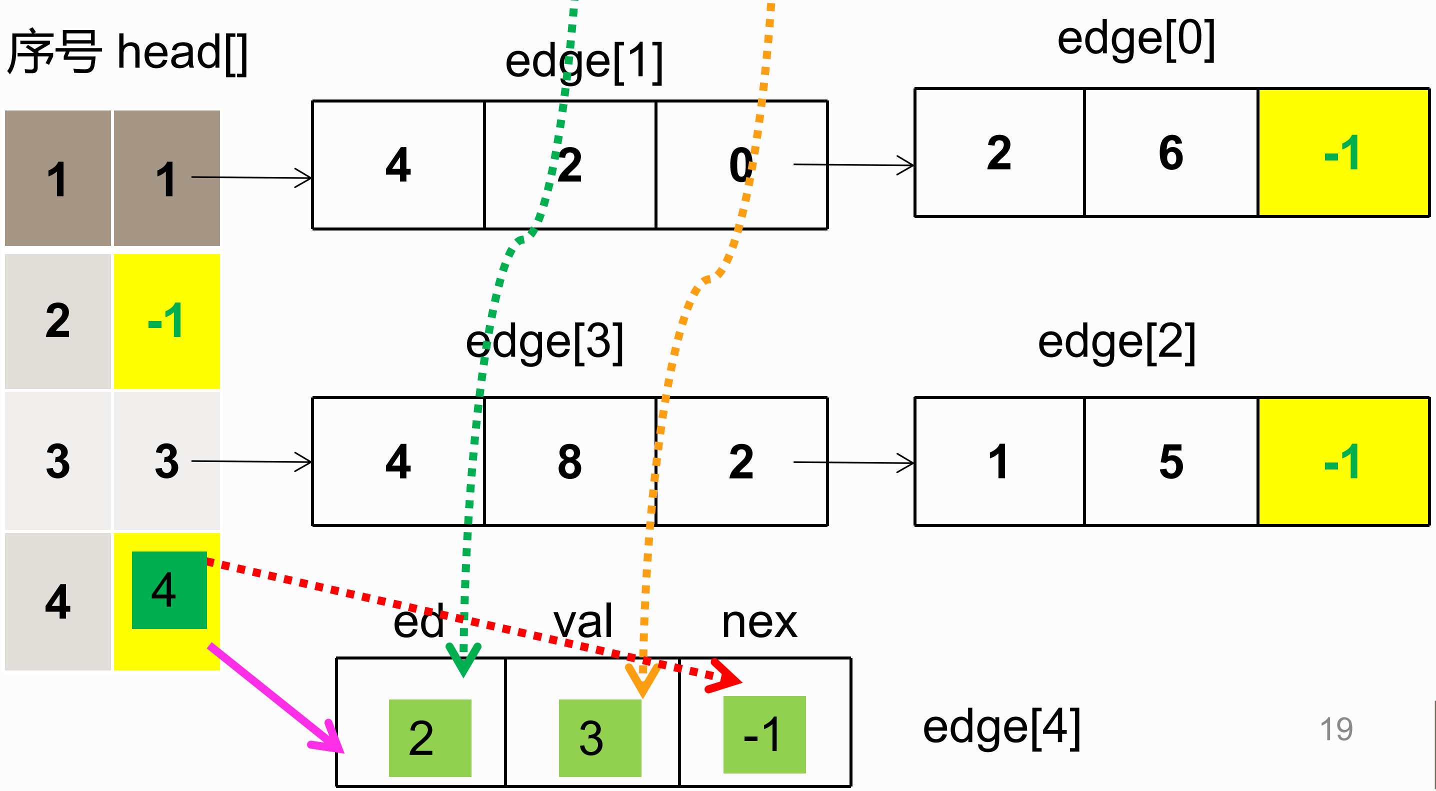
head[4]=idx=4,之后idx+1=5;

还是以该图为例解释具体实现过程:



插入顺序:

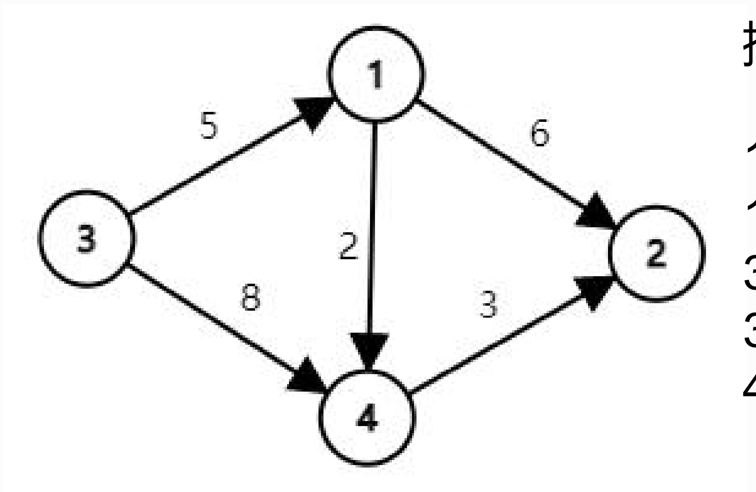
1 2 6
1 4 2
3 1 5
3 4 8
4 2 3



C++STL

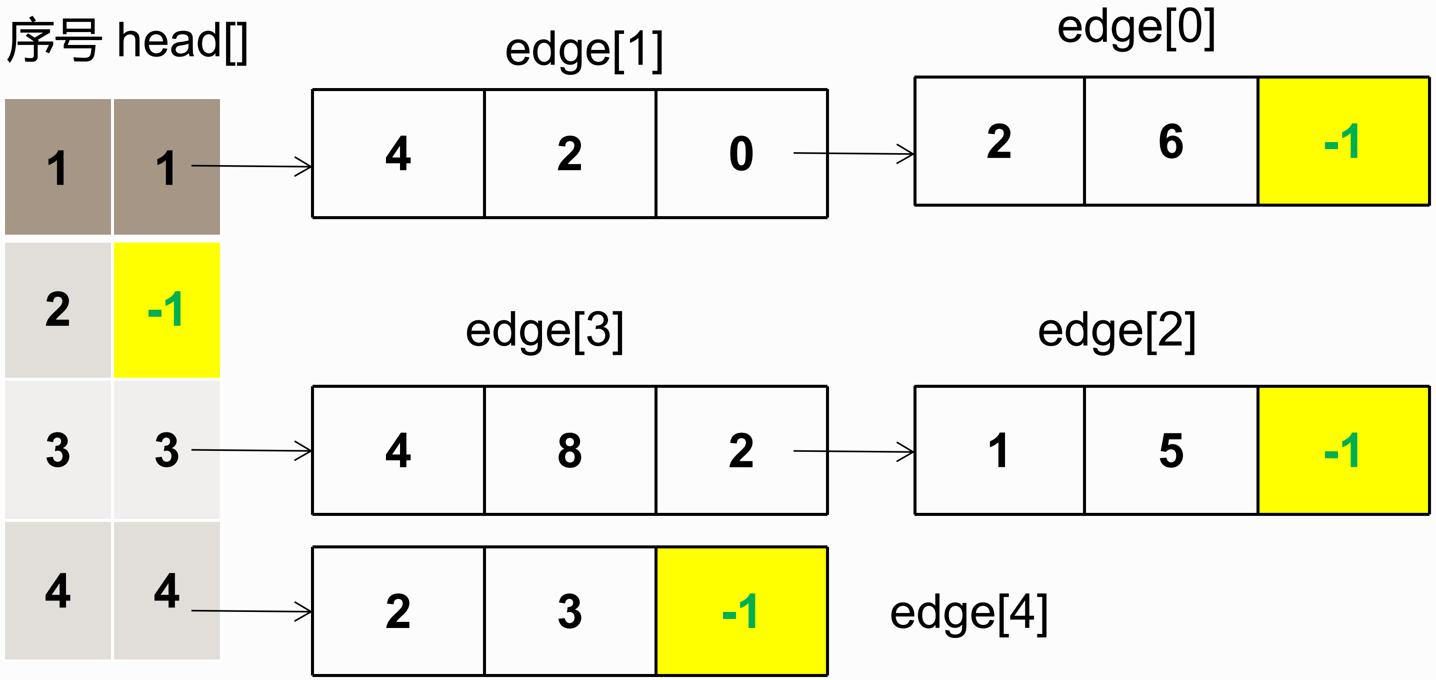
➢ **保存有向图回顾:链式前向星**——以结构体保存或者不以结构体保存

- 最后整理得到链式前向星的图



插入顺序:

1 2 6
1 4 2
3 1 5
3 4 8
4 2 3



C++STL

➢ 保存有向图回顾:链式前向星

题目描述

链式前向星模板题，读入n个点，m条边，以及flag，若flag==1则图有向，否则无向。对每个点输出它的每一条边。

输入格式

第一行三个数n,m,flag，题意如上所示 第2~1+m行，每行三个数，x,y,z，代表从x到y有一条长为z的边

输出格式

若flag=1则m行，flag=0则m*2行，每行三个数，即该点的编号、所指向点的编号，边的长度，先按第一个数升序排列，再以链式前向星中的顺序输出即可。（其实就是i从1到n，再按顺序查找边输出即可）特殊的，若该点无出边，单独一个空行

C++STL

- 保存有向图回顾:链式前向星

注:记得初始化head[]数组和idx归零

代码链接

输入 #1 复制

```
5 5 0
1 2 5
1 4 6
2 3 7
3 5 3
3 4 1
```

输出 #1 复制

```
1 4 6
1 2 5
2 3 7
2 1 5
3 4 1
3 5 3
3 2 7
4 3 1
4 1 6
5 3 3
```

输入 #2 复制

```
4 3 1
1 3 6
3 4 1
2 1 3
```

输出 #2 复制

```
1 3 6
2 1 3
3 4 1
```

2. set（集合）

set 翻译为集合，是一个**内部自动有序且不含重复元素**的容器,而multiset是一个**内部自动有序但含有重复元素**的容器。set 最主要的作用就是自动去重并按升序排序，因此遇到需要去重但是又不方便直接开数组的情况。set 中的元素是唯一的，set和multiset其内部采用“红黑树”（平衡树的一种）实现，它们支持的函数基本相同。

2. set（集合）

set的用途

- set最主要的作用是自动去重并按升序排序，因此碰到需要去重但是却不方便直接开数组的情况，可以用set解决；
- set中元素是唯一的，如果需要处理不唯一的情况，则需要使用multiset；
- unordered_set可以用来处理只去重但不排序的需求，速度比set要快得多。

set的头文件

```
#include <set>
using namespace std;
```

2. set（集合）

set的定义

```
//set <typename> name;
//multiset <typename> name;
//typename 可以是任何基本类型或者容器, name 是集合的名字
set <int> s;
struct rec{
    ...
};
set <struct rec> s;
multiset <double> muls;
```

2. set（集合）

set的常用函数

- **size/empty/clear**

与vector类似，分别表示元素个数、是否为空、清空。前两者的时间复杂度为O(1)

- **迭代器**

set和multiset的迭代器称为“双向访问迭代器”，不支持随机访问，支持'*'解除引用(不能使用*(it+i)和数组下标访问),仅支持‘++’和‘--’两个与算术相关的操作。

设it是一个迭代器，例如 `set<int> :: iterator it`

若把it++，则it将会指向“下一个”元素，这里的下一个是指在元素从小到大排序后的结果中，排在it下一位的元素。同理，若把it--，则it将会排在“上一个”的元素。执行“++”和“--”操作的时间复杂度为O(logn),执行操作前后，务必避免迭代器指向的位置超出首尾迭代器之间的范围。

```
set<int> s;
for(set<int> :: iterator it =s.begin();it!=s.end();it++)
    printf("%d ",*it);
for(auto it:s)
    printf("%d ",it);
```

2. set（集合）

set的常用函数

- **begin/end**

返回集合的首尾迭代器，时间复杂度为O(1)
s.begin()是指向集合中最小元素的迭代器
s.end()是指向集合中最大元素的下一个位置的迭代器。与vector类似，是一个“前闭后开”的形式，所以--s.end()是指向集合中最大元素的迭代器

- **insert**

s.insert(x)把一个元素x插入到集合s中，时间复杂度为O(logn)
在set中，若元素已存在，则不会重复插入该元素，对集合的状态无影响

- **find**

s.find(x)在集合s中查找等于x的元素，并返回指向该元素的迭代器。若不存在，则返回s.end()。时间复杂度为O(logn)

- **count（一般用在multiset中,set中使用的是去重后的个数，要么是0要么是1）**

s.count(x)返回集合中等于x的元素个数，时间复杂度为O(k+logn),其中k为元素x的个数

2. set（集合）

set的常用函数

- **lower_bound/upper_bound**

这两个函数的用法与find类似，但条件略有不同，时间复杂度为O(logn)
s.lower_bound(x),查找≥x的元素的最小的一个，并返回该元素的迭代器
s.upper_bound(x),查找>x的元素的最小的一个，并返回该元素的迭代器

- **erase**

设it是一个迭代器，s.erase(it)从s中删除迭代器it指向的元素，时间复杂度为O(logn)
设x是一个元素，s.erase(x)从s中删除所有等于x的元素，时间复杂度为O(k+logn），k为被删除元素的次数
如果想从multiset中删除**至多一个等于x**的元素，可执行以下代码:

```
if((it=s.find(x))!=s.end())//it是迭代器
     s.erase(it);
```

3. map（映射）

map翻译为映射，是STL中的常用容器。其实，数组就是一种映射，数组总是将int类型映射到其它基本类型（称为数组的基类型），这同时也带来了一个问题，有时候我们希望把string映射成一个int，数组就不方便了。这时就可以使用map，map可以将任何基本类型（包括STL容器）映射到任何基本类型（包括STL容器）。

map容器是一个键值对key-value的映射，其内部实现是一棵以key为关键码的红黑树。map的key和value可以是任意类型

3. map（映射）

map的用途

- 需要建立**字符（串）与整数**之间的映射，使用 map 可以减少代码量；
- 判断**大整数（比如几千位,或者负数）或者其他类型数据是否存在**，可以把map当布尔型数组使用（哈希表）；
- **字符串与字符串之间**的映射。

map的头文件

```
#include <map>
using namespace std;
```

3. map（映射）

map的**定义**

```
//map <key_type,val_type> name;
//key_type和val_type可以是任意类型

map < long long , bool > vis;
map < string , int > hash;
map < pair< int , int > , vector < int > > test;
```

- 在很多时候,map容器被当做hash表使用,建立复杂信息key(如字符串)到简单信息value(如一定范围内的整数)的映射;
- map是基于平衡树实现的，**它的大部分操作的时间复杂度都在O(logn)级别**，略慢于使用Hash函数实现的传统Hash表。从C++11开始，STL新增了unordered_map等基于Hash的容器，但部分算法竞赛并不支持C++11标准。
- unordered_map的头文件为#include <unordered_map>,由于函数与map相同实现上存在差异，此处还是以map具体分析

3. map（映射）

map的常见函数：

➢ size/empty/clear/begin/end
与vector类似，分别表示元素个数、是否为空、清空、首迭代器、尾迭代器

➢ 迭代器
map的迭代器与set一样，也是“双向访问迭代器”，对map点的迭代器解除引用后，将得到一个二元组
pair<key_type,value_type>

➢ insert/erase
与set类似，分别表示插入和删除。insert的参数是pair<key_type,value_type>,erase的参数可以是key值或者迭代器

```
map<int,int> mp;
mp.insert({-1,0});
mp.insert({5,7});
mp.insert(make_pair(3,8));
mp.insert(make_pair(-5,-5));
map <int,int>::iterator it=mp.begin();
```

```
mp.erase(it);//删除迭代器
it=mp.begin();
pair <int,int> p=*it;
printf("%d %d\n",p.first,p.second);
mp.erase(-1);//删除key及其映射
it=mp.begin();
p=*it;
printf("%d %d\n",p.first,p.second);
```

3. map（映射）

map的**常见函数**：

- **find**

mp.find(x)在变量名为mp的map中查找key值为x的二元组，并返回指向该二元组的迭代器，若不存在，返回mp.end()，时间复杂度为O(logn)

- **[ ]操作符**

mp[key]返回key映射到的value得引用，时间复杂度为O(logn)

[ ]操作是map最吸引人的地方，可以很方便通过mp[key]来得到key对应的value值，还可以对mp[key]进行赋值操作，改变key对应的value

需要注意的是，若查找的key不存在，则执行mp[key]后，mp会自动新建一个二元组(key,zero),并返回zero的引用，这里的zero表示一个广义的“零值”，如整数0、空字符串。如果查找之后不对mp[key]进行赋值，那么时间一长，mp会包含很多无用的“零值二元组”，白白占用内存，降低程序运行效率。建议**在用[ ]操作符查询之前，先用find方法检查key的存在性**。

3. map（映射）

map的常见函数举例:

给定n个字符串，m次询问，每次询问一个字符串出现的次数。
n≤20000,m≤20000,每个字符串的长度都不超过20。

```
map<string,int> mp;
for(int i=1;i<=n;i++){
      cin >> s;
      mp[s]++;
}
for(int i=1;i<=m;i++){
      cin >> s;
      if(mp.find(s)==mp.end())
            printf("0\n");
      else printf("%d\n",mp[s]);
}
```

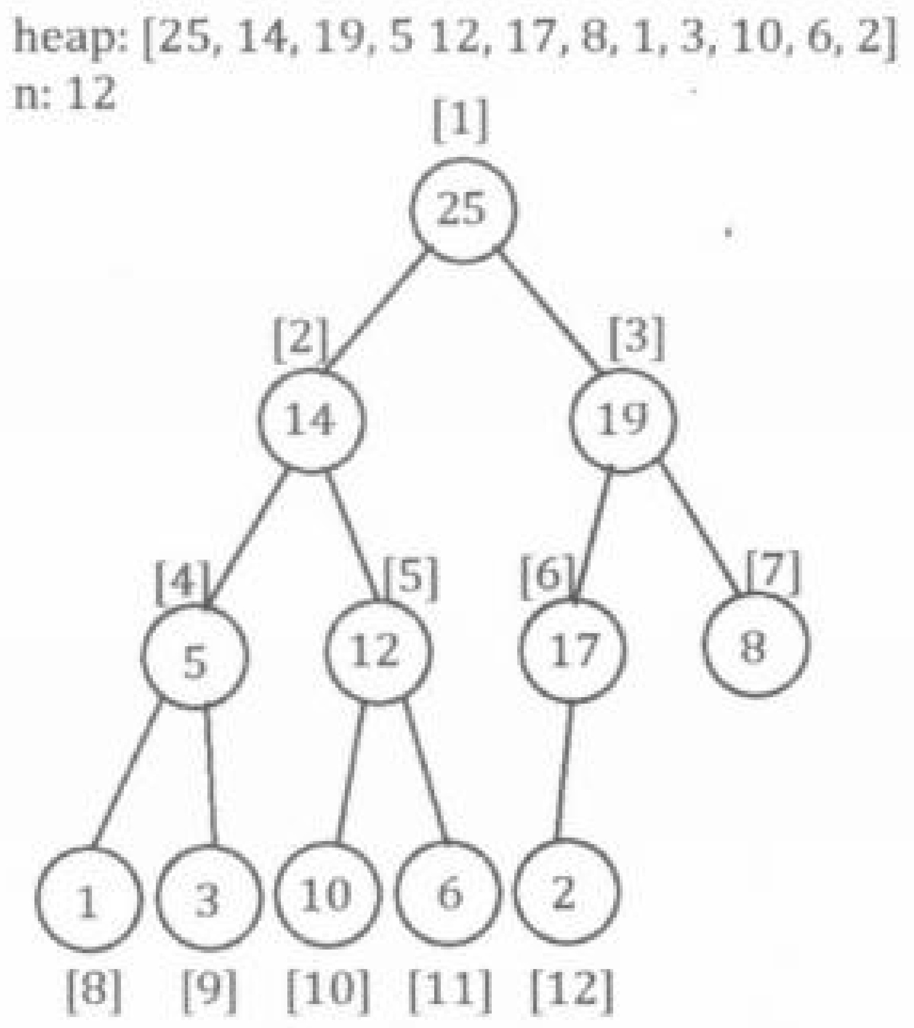
二叉堆

二叉堆是支持**插入、删除和查询最值**的数据结构。它是一棵满足“性质”的完全二叉树（叶子结点都在最后两层，且在最后一层集中于左侧的二叉树），树上的每个结点带有一个权值，若树中的任意一个结点的权值都小于等于其父结点的权值，则称该二叉树满足“**大根堆**性质”。若树中任意一个结点的权值都大于等于其父结点的权值，则称该二叉树满足“**小根堆**性质”。满足“大根堆性质”的完全二叉树就是“大根堆”，而满足“小根堆性质”的完全二叉树就是“小根堆”，二者都是二叉堆的形态之一。

哈夫曼树(优先队列)

二叉堆

根据完全二叉树的性质,我们可以采用层次序列存储方式，直接用一个数组来保存二叉堆。层次序列存储方式，就是逐层从左到右为树中的结点依次编号，把此结点作为结点在数组中存储的位置（下标）。在这种存储方式下，**父结点编号等于子结点编号除以2，左子结点编号等于父结点编号乘2，右子结点编号等于父结点编号乘2加1**



大根堆

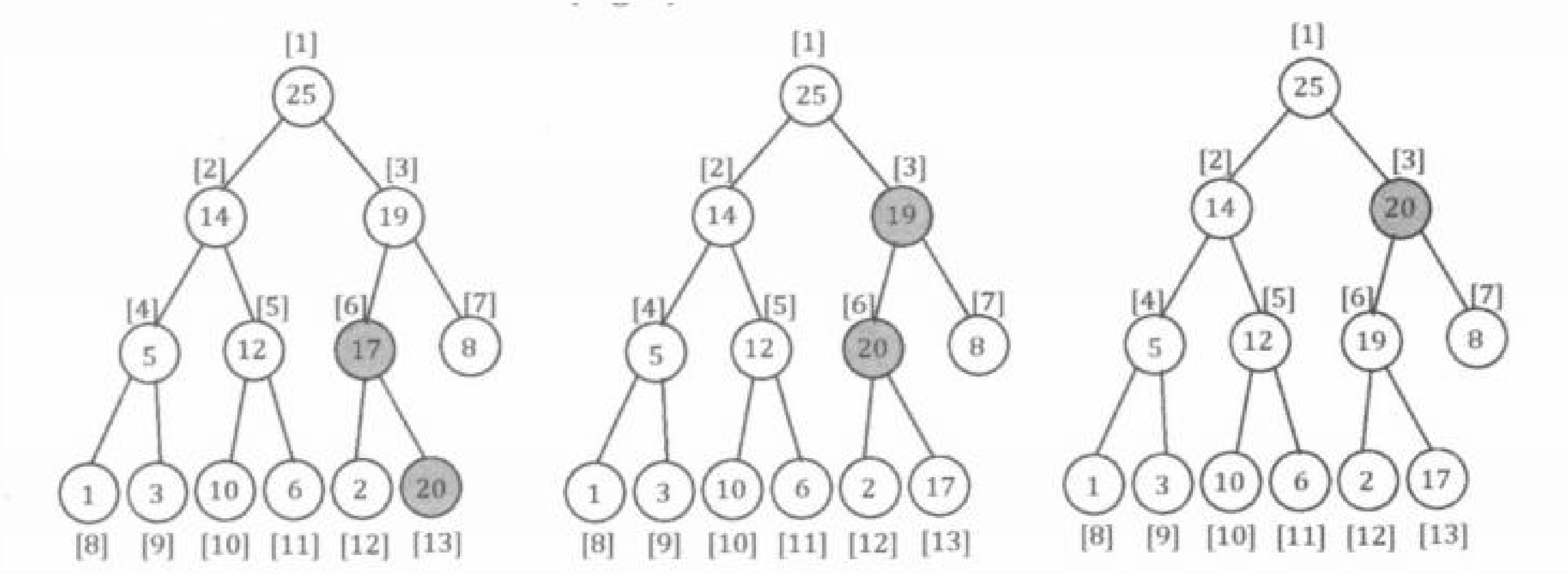
哈夫曼树(优先队列)

二叉堆

以大根堆为例解释操作

1. Insert插入

Insert(val)操作向二叉堆中插入一个带有权值val的新结点，把这个新结点直接放在存储二叉堆的数组末尾，然后通过交换的方式向上调整，直至满足堆性质。其时间复杂度为堆的深度，即O(logN)



哈夫曼树(优先队列)

二叉堆

以大根堆为例解释操作

1. Insert插入

Insert(val)操作向二叉堆中插入一个带有权值val的新结点，把这个新结点直接放在存储二叉堆的数组末尾，然后通过交换的方式向上调整，直至满足堆性质。其时间复杂度为堆的深度，即O(logN)

```
const int SIZE=10010;
int n,heap[SIZE];
void up(int p){//向上调整
    while(p>1){
        if(heap[p]>heap[p/2])//当前结点大于父结点,交换;反之,不交换
            swap(heap[p],heap[p/2]),p/=2;
        else break;
    }
}
void Insert(int val){//插入val
    heap[++n]=val;
    up(n);
}
```

哈夫曼树(优先队列)

二叉堆

以大根堆为例解释操作

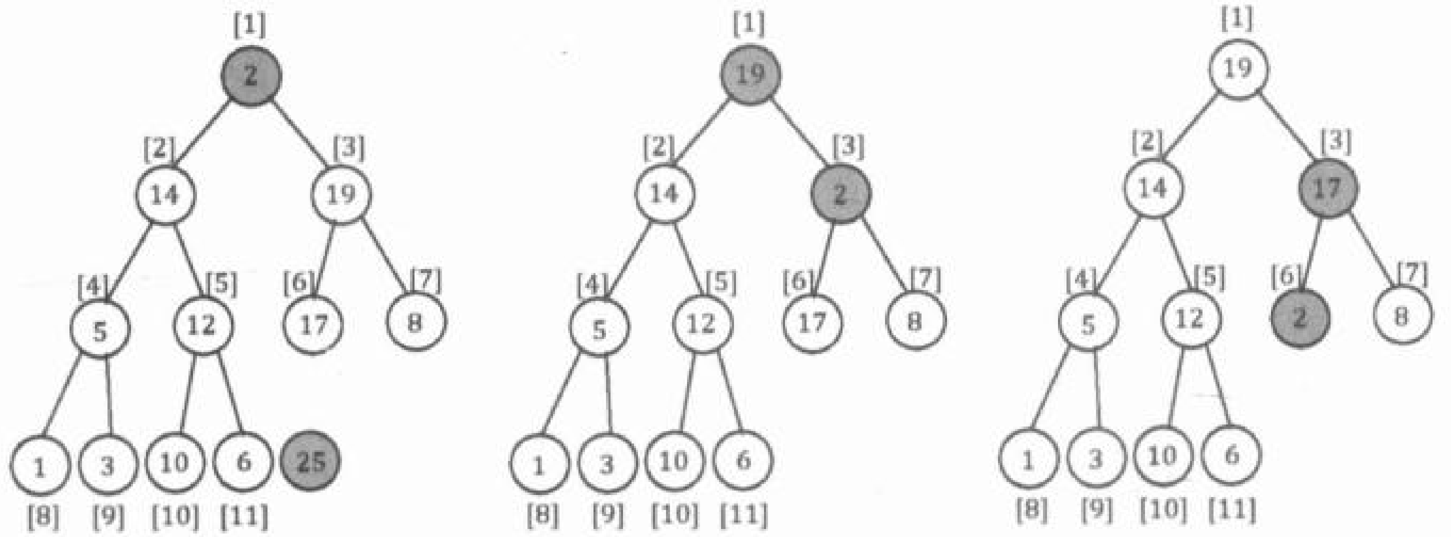
2. Gettop获得堆顶

GetTop操作返回二叉堆的堆顶权值，即最大值heap[1],复杂度为O(1)

```
int GetTop(){//查询最值
    return heap[1];
}
```

3. Extract删除堆顶

Extract操作把堆顶从二叉堆中移除。把堆顶heap[1]与存储在数组末尾的结点heap[n]交换,然后移除数组末尾结点(令n减小1),最后把堆顶通过交换的方式向下调整，直至满足堆性质。其时间复杂度为堆的深度,即O(logN)



哈夫曼树(优先队列)

二叉堆

以大根堆为例解释操作

3. Extract删除堆顶

Extract操作把堆顶从二叉堆中移除。把堆顶heap[1]与存储在数组末尾的结点heap[n]交换,然后移除数组末尾结点(令n减小1),最后把堆顶通过交换的方式向下调整，直至满足堆性质。其时间复杂度为堆的深度,即O(logN)

```
void down(int p)//向下调整{
    int s=p*2;//p的左结点
    while(s<=n){
        if(s<n&&heap[s]<heap[s+1])
      //左右子结点中选取较大值
            s++;
        if(heap[s]>heap[p]){
      //交换子结点和父结点
            swap(heap[s],heap[p]);
            p=s;
            s=p*2;
        }
        else break;
    }
}
void Extract(){//删除堆顶元素
    heap[1]=heap[n--];
    down(1);
}
```

哈夫曼树(优先队列)

二叉堆

完整二叉堆代码链接

以大根堆为例解释操作

4. Remove删除某个位置元素

Remove(p)操作把存储在数组下标p位置的结点从二叉堆中删除。与Extract相类似,把heap[p]与heap[n]交换,然后令n减小1。注意此时heap[p]既有可能需要向下调整，也有可能向上调整，需要分别进行检查和处理。时间复杂度为O(logN)

```
void Remove(int k){//删除二叉堆中第k个元素
    heap[k]=heap[n--];
    up(k);
    down(k);
}
```

C++的STL中的priority_queue(优先队列)默认实现了一个大根堆,支持push(Insert)、top(GetTop)和pop(Extract)操作，但不支持Remove操作。

哈夫曼树(优先队列)

优先队列priority_queue

优先队列(priority_queue)一般用于解决贪心问题,其底层是用**堆**来实现,在优先队列中，任何时刻，队首元素一定是当前队列中优先级最高（优先值最大）的那一个（大根堆），也可以是最小的那一个（小根堆）。可以不断往优先队列中添加某个优先级的元素，也可以不断弹出优先级最高的那个元素，每次操作其会自动调整结构，始终保证队首元素的优先级最高。

优先队列**头文件**

```
#include <queue>
using namespace std;
```

优先队列priority_queue

优先队列**定义**

```
priority_queue <typename> name;//默认是大根堆
//typename可以是任何基本类型或者容器，name为优先队列的名字

priority_queue <int> q1;//生成一个int类型的大根堆
//priority_queue < int , vector <int> ,less <int> > q1;//与上方优先队列等价

priority_queue < double, vector <double> ,greater <double> > q2;
//生成一个double类型的小根堆
```

哈夫曼树(优先队列)

优先队列priority_queue

关于结构体的优先队列

priority _ queue中存储的元素类型必须定义"小于号",较大的元素会被放在堆顶。内置的int, string等类型本身就可以比较大小。若使用自定义的结构体类型,则需要重载"<"运算符。

```
struct node{
    int x;
    int y;
};

int operator <(const struct node &a,const struct node &b){
    return a.x<b.x;//大根堆
    //return a.x>b.x;//小根堆
    //注意此处与结构体排序的cmp不同
}

priority_queue <struct node> q;
```

哈夫曼树(优先队列)

优先队列priority_queue

常见两种**小根堆**存储方式

```
priority_queue < double, vector <double> ,greater <double> > q1;
//方法一：利用greater进行转化
//方法二:以负数的形式存储，取得时候再取回其相反数，出来的就是最小的
```

哈夫曼树(优先队列)

优先队列priority_queue

优先队列**常见函数**

和queue不一样的是，priority_queue没有front()和back()，而只能通过top()或pop()访问队首元素（也称堆顶元素），也就是优先级最高的元素

- **push**

push(x)是将x加入优先队列，**时间复杂度为O(logN)**，n为当前优先队列中的元素个数。加入后会自动调整priority_queue的内部结构，以保证队首元素(堆顶元素)的优先级最高

- **top**

top()是获得队首元素(堆顶元素)，时间复杂度为O(1)

- **pop**

pop()是让队首元素(堆顶元素)出队，**时间复杂度为O(logN)**，n为当前优先队列中的元素个数。出队后会自动调整priority_queue的内部结构，以保证队首元素(堆顶元素)的优先级最高

优先队列priority_queue

优先队列(priority_queue)一般用于解决贪心问题,其底层是用堆来实现,在优先队列中，任何时刻，队首元素一定是当前队列中优先级最高（优先值最大）的那一个（大根堆），也可以是最小的那一个（小根堆）。可以不断往优先队列中添加某个优先级的元素，也可以不断弹出优先级最高的那个元素，每次操作其会自动调整结构，始终保证队首元素的优先级最高。

扩展:双端队列deque

双端队列deque是一个支持在两端高效插入或删除元素的连续线性存储空间。它就像**vector和queue的结合**。与vector相比,deque在头部增删元素仅需要O(1)的时间,与queue相比,deque像数组一样支持随机访问。但缺点就是占用内存较多。

双端队列**头文件**:

```
#include <queue>
#include <deque>//好像两者都可以

using namespace std;
```

哈夫曼树(优先队列)

扩展:双端队列deque

双端队列**常用函数**:

函数	描述	示例	时间复杂度
[ ]	随机访问（通过下标）	与 vector 类似	O(1)
begin/end	deque 的头/尾迭代器	与 vector 迭代器类似	O(1)
front/back	队头/队尾元素	与 queue 类似	O(1)
push_back	从队尾入队	q.push_back(x);	O(1)
push_front	从队头入队	q.push_front(y);	O(1)
pop_back	从队尾出队	q.pop_back(x);	O(1)
pop_front	从队头出队	q.pop_front(y);	O(1)
clear	清空队列	q.clear();	O(n)

它同样拥有size和empty的查询功能

扩展:双端队列deque

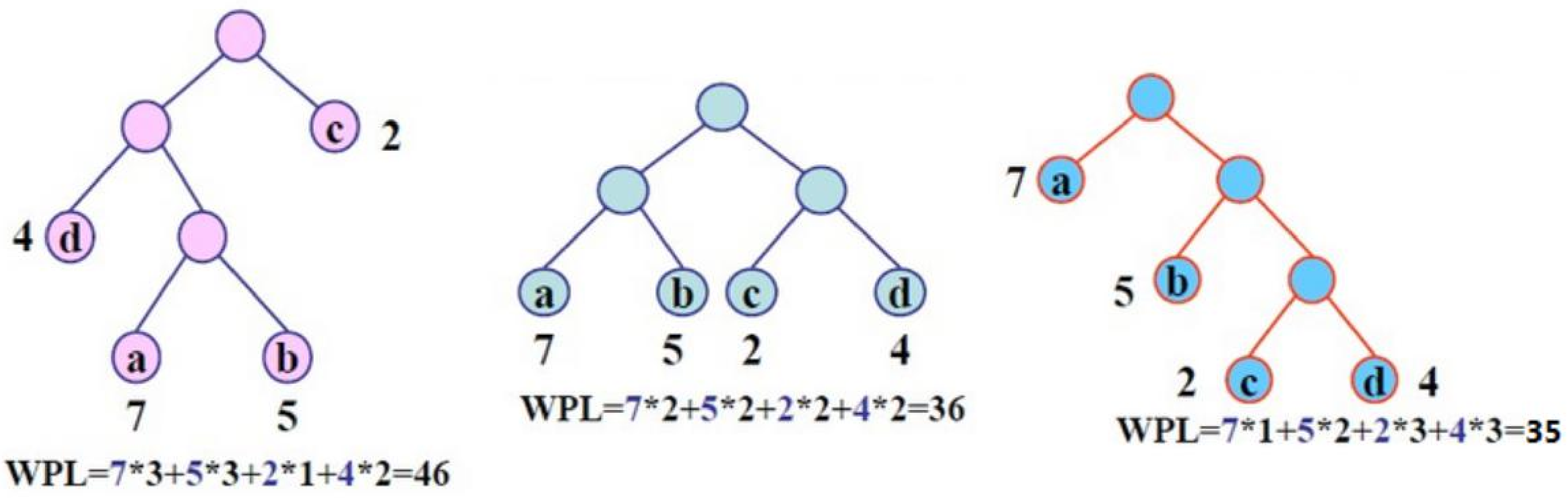
```
deque <int> q;
//q.push_back(x);//将x放入双端队列的队尾
//q.push_front(y);//将y放入双端队列的队头
q.push_back(52);
q.push_front(32);
q.push_back(54);
q.push_front(2);

//遍历方法与vector类似(其他两种就不写了)
for(deque <int> :: iterator i=q.begin()/*deque的头迭代器*/;i!=q.end()/*deque的尾迭代器*/;i++)
	printf("%d\n",*i);
int k1=q.front();//获得双端队列的队头元素
int k2=q.back();//获得双端队列的队尾元素

q.pop_back();//删除双端队列的队尾元素
q.pop_front();//删除双端队列的队头元素

q.clear();//清空所有deque中的元素
```

哈夫曼树(优先队列)

哈夫曼树

构造一个包含n个叶子结点的k叉树，其中第i个叶子结点带有权值w_i，要求最小化 $\sum w_i \times l_i$ 其中l_i表示第i个叶子结点到根结点的距离。该问题的解被称为k叉Huffman树(哈夫曼树)。

例 有4个结点，权值分别为7，5，2，4，构造有4个叶子结点的二叉树



【哈夫曼树】带权路径长度最小的二叉树就称为哈夫曼树或最优二叉树。上图第3种二叉树是哈夫曼树

哈夫曼树(优先队列)

哈夫曼树

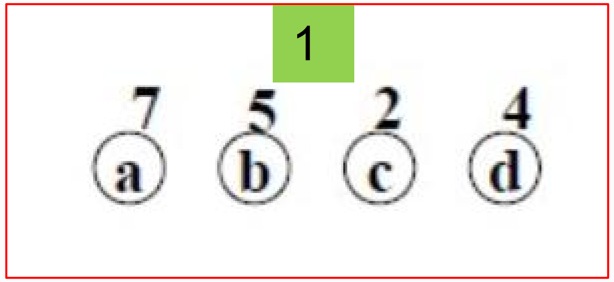
为了最小化 $\sum w_i \times l_i$,应让权值大的叶子结点的深度尽量小。当k=2时，容易想到用**贪心的思想**求出二叉哈夫曼树。

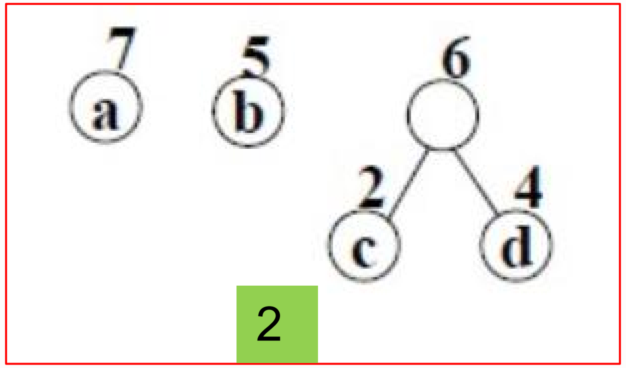
1. 建立二叉堆，插入这n个叶子结点的权值
2. 从堆中取出最小的两个权值w1和w2,令ans+=w1+w2
3. 建立一个权值为w1+w2的树结点p，令p成为权值为w1和w2的树结点的父亲
4. 在堆中插入权值w1+w2
5. 重复第2~4步，直到堆的大小为1

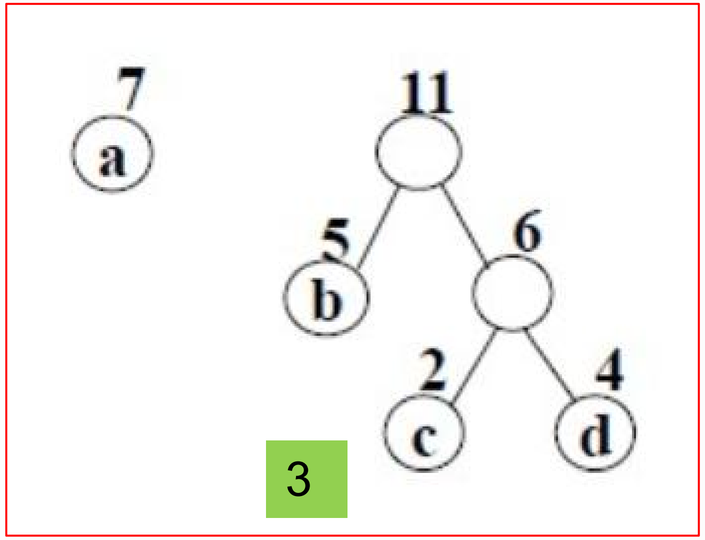
最后，由所有新建的p与原来的叶子结点构成的树就是Huffman树，变量ans就是 $\sum w_i \times l_i$ 的最小值。

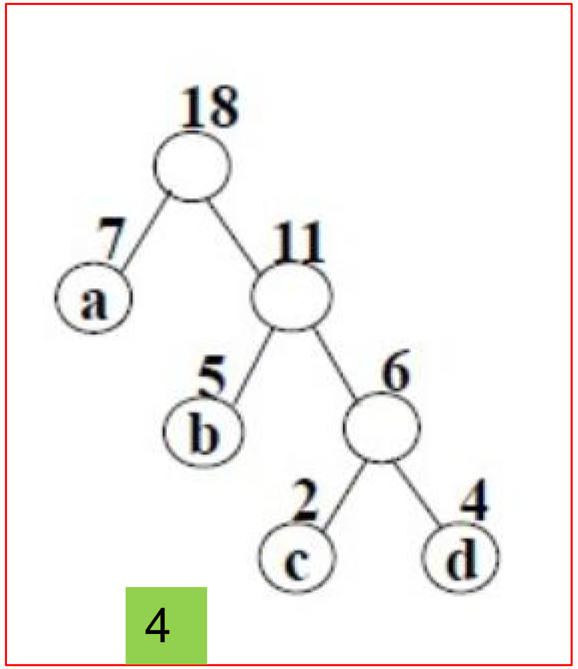
哈夫曼树(优先队列)

哈夫曼树









哈夫曼树(优先队列)

哈夫曼树

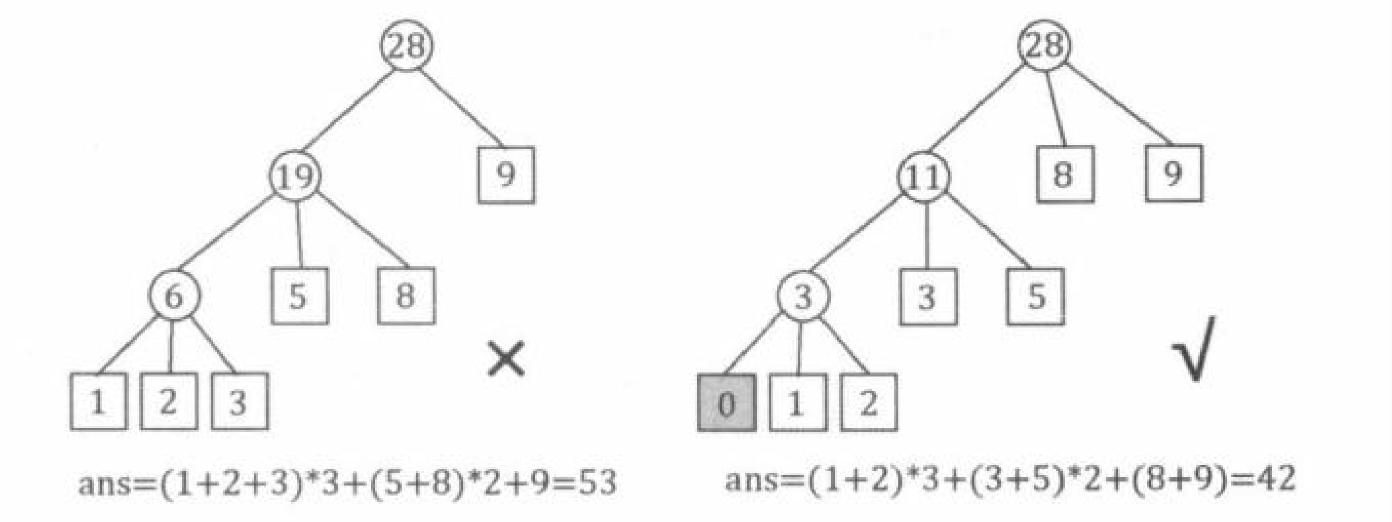
```
priority_queue<int,vector<int> ,greater<int> > q;
for(int i=1;i<=n;i++){
    int a;
    cin >> a;
    q.push(a);
}

while(q.size()>=2){
    int temp1=q.top();
    q.pop();
    int temp2=q.top();
    res+=temp1+temp2;
    q.pop();
    q.push(temp1+temp2);
}
cout << res << endl;
```

哈夫曼树(优先队列)

哈夫曼树拓展(了解)

对于k(k > 2)叉 Huffman树的求解，直观的想法是在上述贪心算法的基础上，改为每次从堆中取出最小的k 个权值。然而，仔细思考可以发现，如果在执行最后一轮循环时，**堆的大小在2~k -1之间(不足以取出k 个)**，那么整个Huffman树的根的子结点个数就小于k。这显然不是最优解——我们任意取Huffman树中一个深度最大的结点，把它改为树根的子结点，就会使$\sum w_i \times l_i$变小。

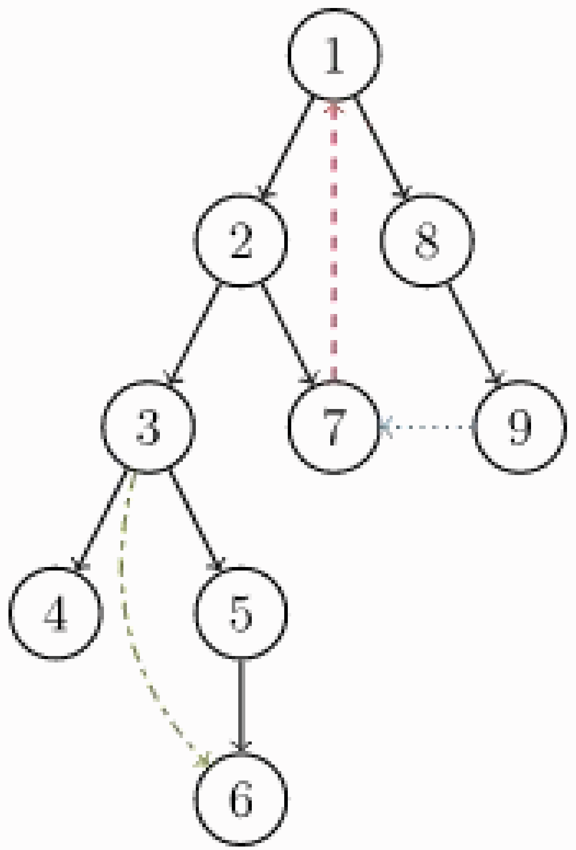
因此,我们应该在执行上述贪心算法之前，**补加一些额外的权值为0的叶子结点，使叶子结点的个数n 满足(n - 1) mod (k - 1)=0**。也就是说，我们让子结点不足k个的情况发生在最底层，而不是根结点处。在(n - 1) mod (k - 1)=0 时，执行“每次从堆中取出最小的k个权值”的贪心算法就是正确的。



强连通分量

概念：

- **流图：** 给定有向图 $G=(V,E)$ ，若存在点 $r \in V$，满足从点r出发能到达V中的所有的点，则称为G是一个流图(FLow Graph)，记为(G,r)，其中点r称为流图的源点。
- **流图(G,r)的搜索树：** 在一个流图(G,r)上从点r出发进行**深度优先遍历(dfs)**，每个点只访问一次。所有发生递归的边(x,y)（换言之，从x到y是对y的第一次访问）构成一棵以r为根的树，我们把它称为流图(G,r)的搜索树。
- 同时，在深度优先遍历的过程中，按照每个结点第一次被访问的时间顺序，依次给予流图中的N个结点1~N的整数标记，该标记被称为**时间戳**，记为**dfn[x]**



强连通分量

概念：

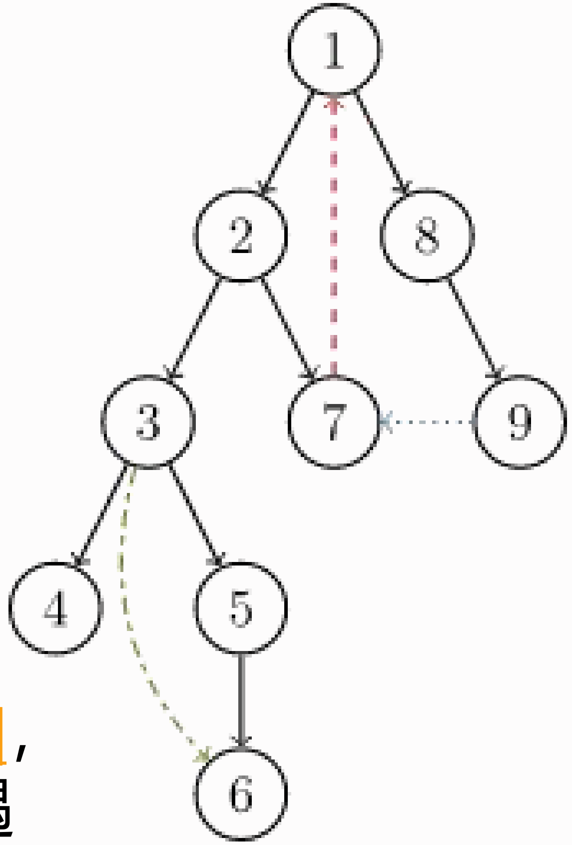
➢ 流图中每条有向边(x,y)必然是以下四种之一:

①**树枝边**，指搜索树的边，即x是y的父结点，如示意图以**黑色**边表示，每次搜索找到一个还没有访问过的结点的时候就形成了一条树边

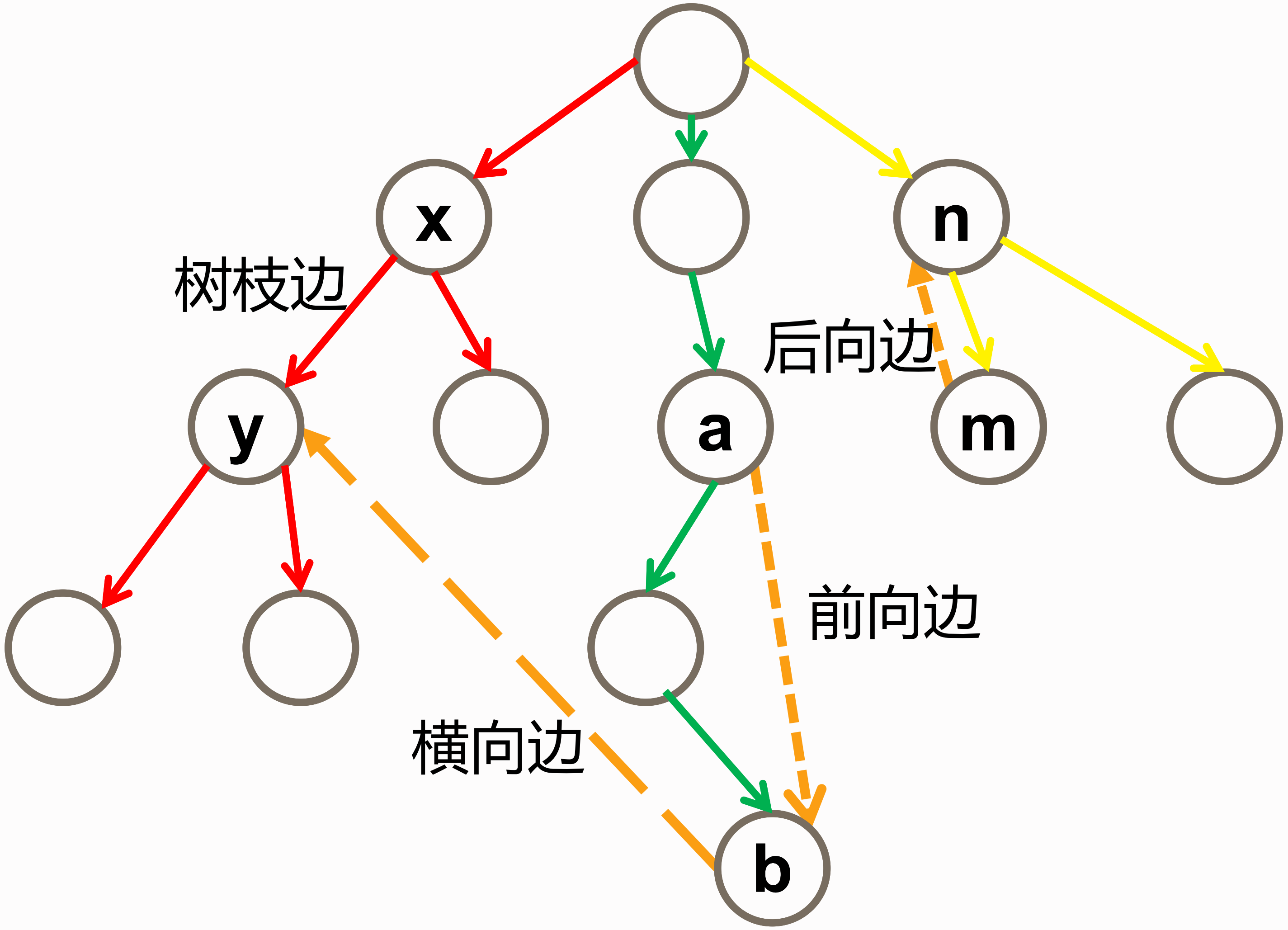
②**前向边**，指搜索树中x是y的祖先结点，如示意图中以**绿色**边表示（即 3->6 ）

③**后向边**，指搜索树中y是x的祖先结点，如示意图中以**红色**边表示（即7->1 ）

④**横叉边**，指除了以上三种情况之外的边，它一定满足dfn[y]<dfn[x]，如示意图中以**蓝色**边表示（即 9->7 ），它主要是在搜索的时候遇到了**一个已经访问过的结点**，但是这个结点 并不是 当前结点的祖先(注：7的横叉边一定没有8或9,因为一定是已经访问过的结点)



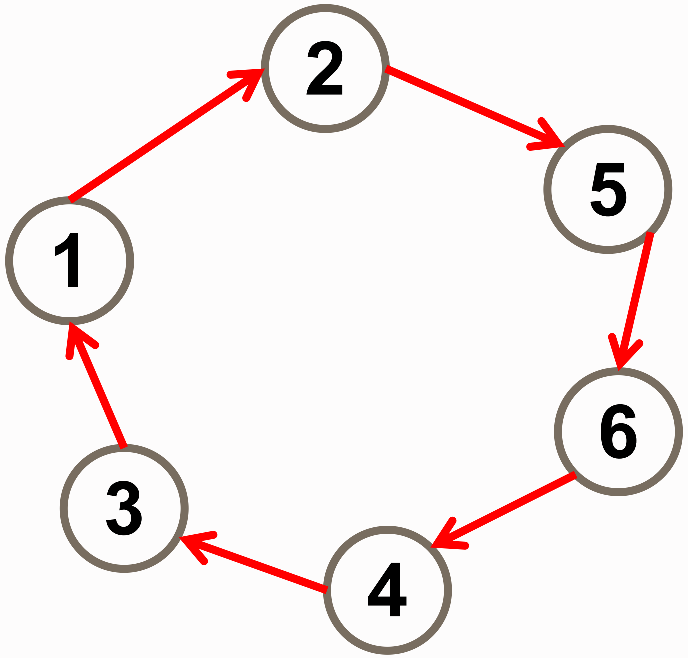
强连通分量

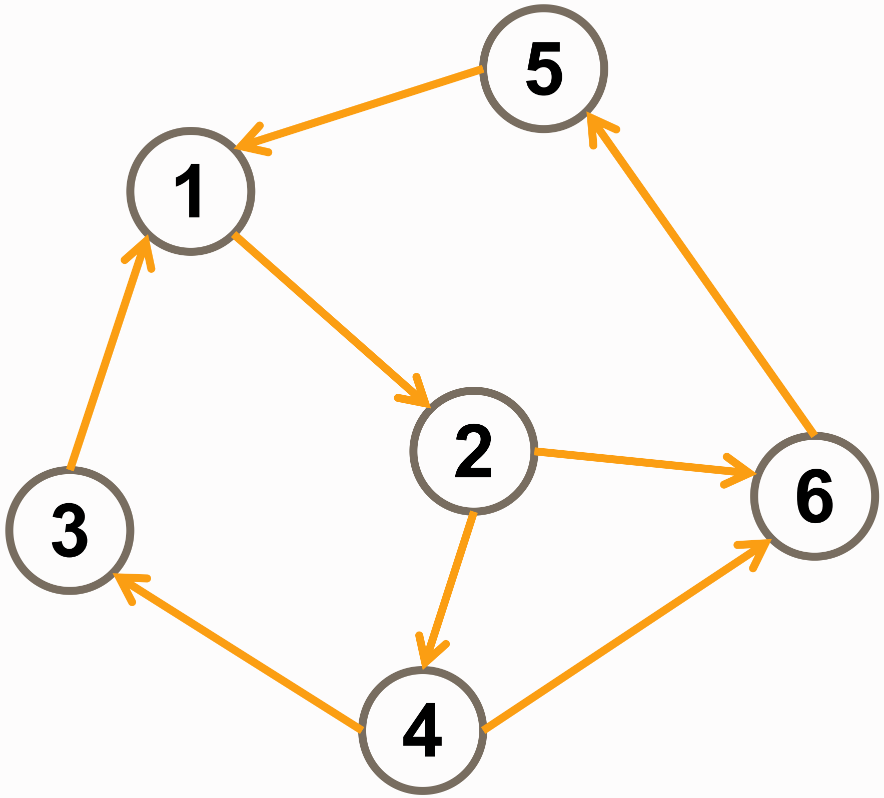


①**树枝边**(x,y)
②**前向边**(a,b)
③**后向边**(m,n),(b,a)
④**横叉边**(b,y),(b,x)
注: (b,n)和(b,m)不是横叉边

强连通分量

给定一张有向图，若对于图中任意两个结点x,y，**既存在从x到y的路径，也存在从y到x的路径**，则称该有向图为“强连通图”。

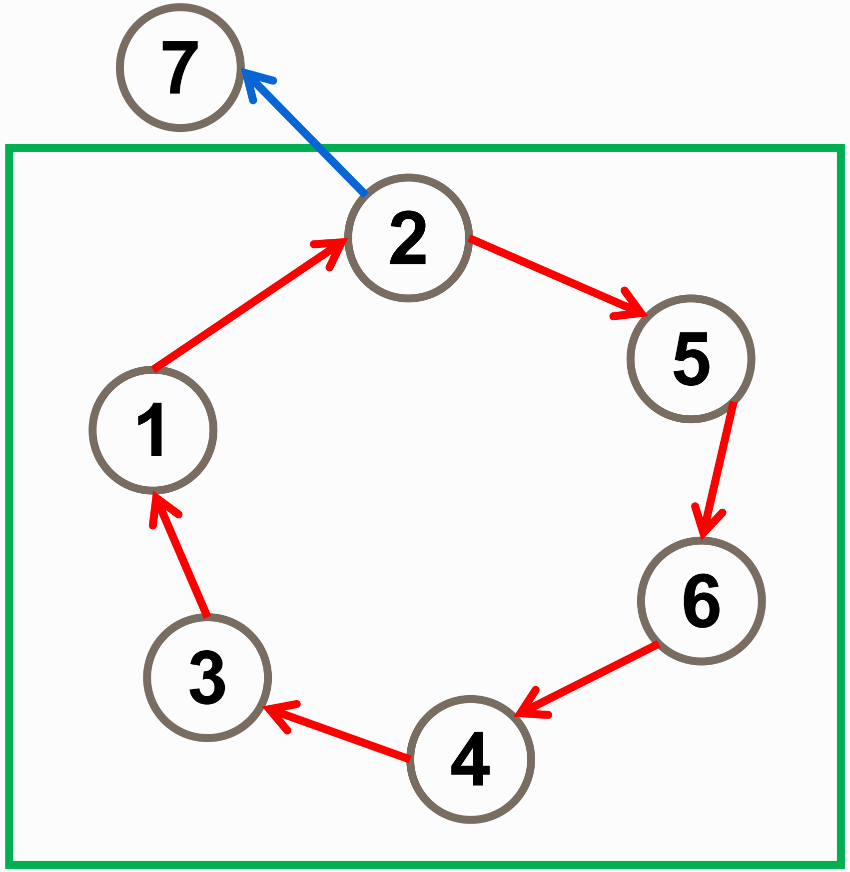


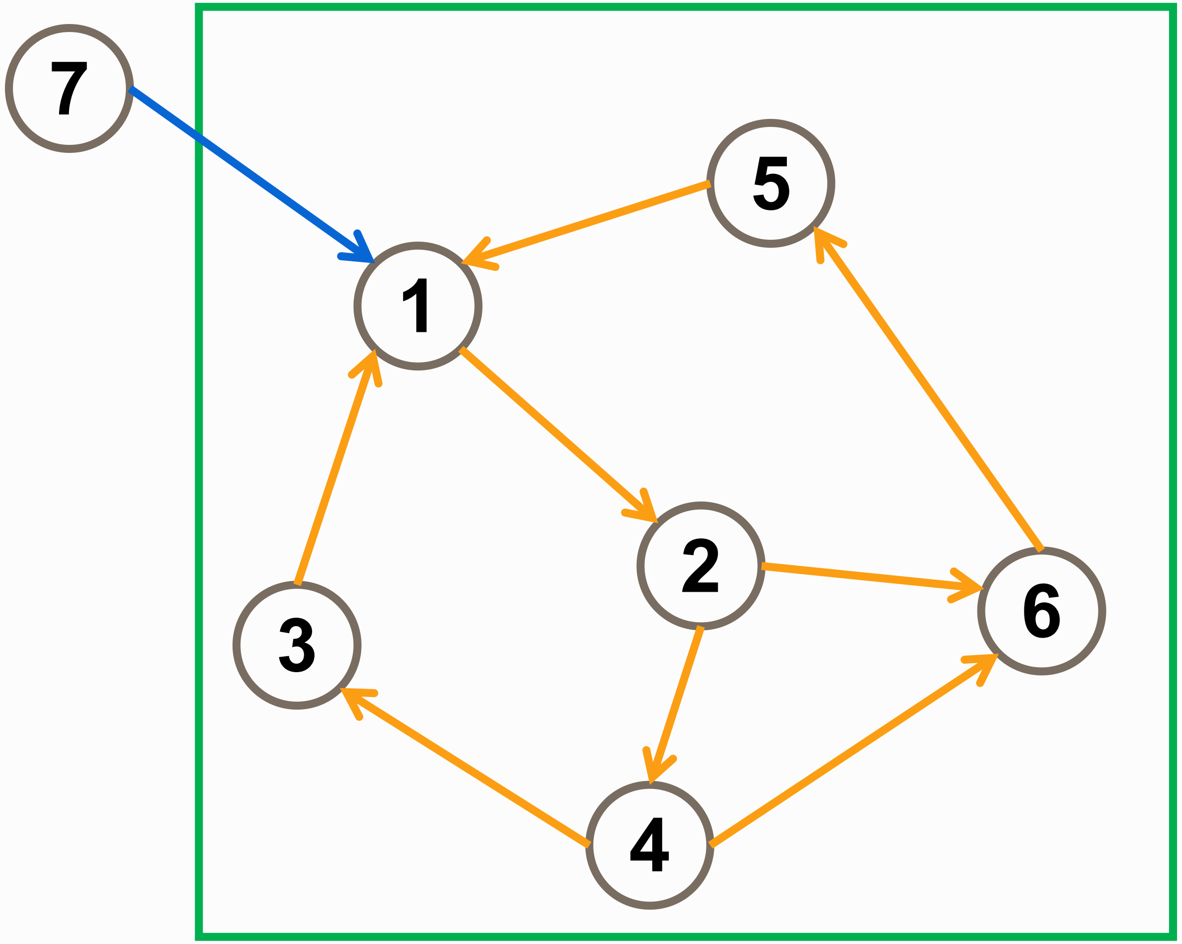


强连通分量

作用：将一个有向图缩点成有向无环图(DAG)

有向图的**极大强连通子图被称为“强连通分量”**，简记为**SCC**(Strongly Connected Component)，其中极大的强连通子图可理解为，在强连图图的基础上**加入一些点和路径，使得当前的图不再强连通**，称原来的强连通的部分为强连通分量



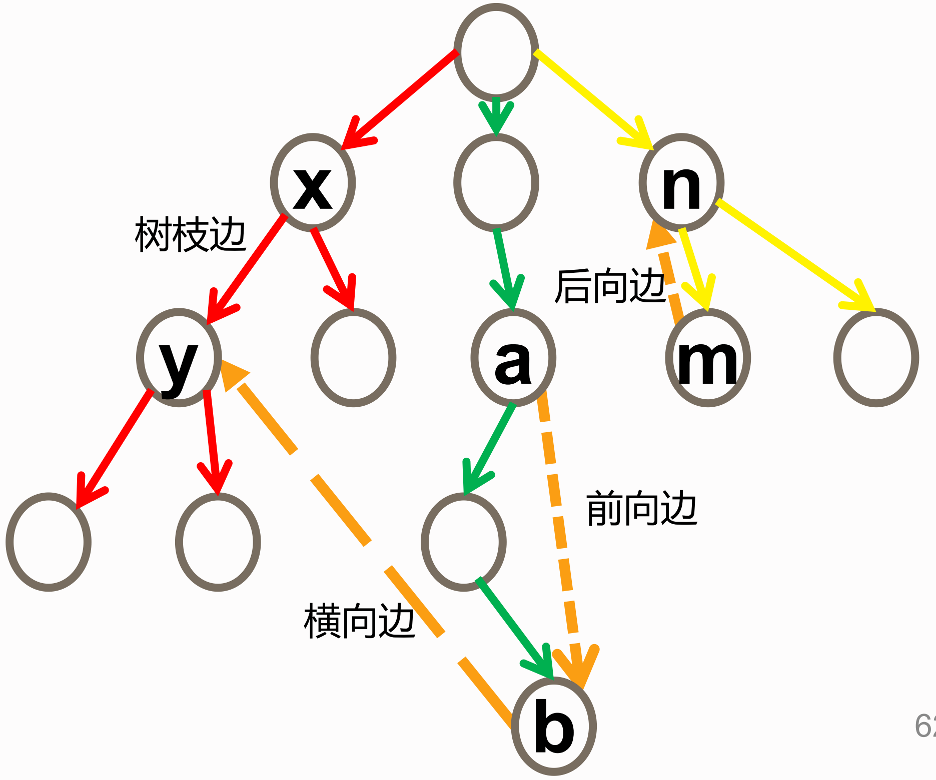


强连通分量

根据著名计算机科学家Robert Tarjan的名字命名的Tarjan算法能够在线性时间内求出**无向图的割点和桥，进而求出无向图的双连通分量**(不讲)。在有向图方面，Tarjan算法能够**求出有向图的强连通分量(讲)**、**必经点和必经边**(不讲)。Robert Tarjan在数据结构方面也做出了很多卓有成效的工作，包括证明并查集的时间复杂度，提出斐波那契堆、**Splay Tree(伸展树)** 和Lint-Cut Tree(动态树)等。

强连通分量

- ✓ Tarjan算法基于有向图的深度优先遍历，能够在线性时间内求出一张有向图的各个强连通分量。
- ✓ **一个“环”一定是强连通图**。如果既存在从x到y的路径，也存在从y到x的路径，那么x,y显然在一个环中。因此，Tarjan算法的基本思路就是对于每个点，尽量找到与它一起能构成环的所有结点。
- ✓ 容易发现，“前向边”(x,y)没有什么用处，因为搜索树上本来就存在x到y的路径。“后向边”(x,y)非常有用，因为它可以和搜索树上从y到x的路径一起构成环。而“横叉边”(x,y)视情况而定，如果y出发能找到一条路径回到x的祖先结点，那么“横叉边”(x,y)就是有用的。



强连通分量

为了找到通过“后向边”和“横叉边”构成的环，Tarjan算法在深度优先遍历的同时维护了一个栈。当访问到结点x时，栈中需要保存以下两类结点：

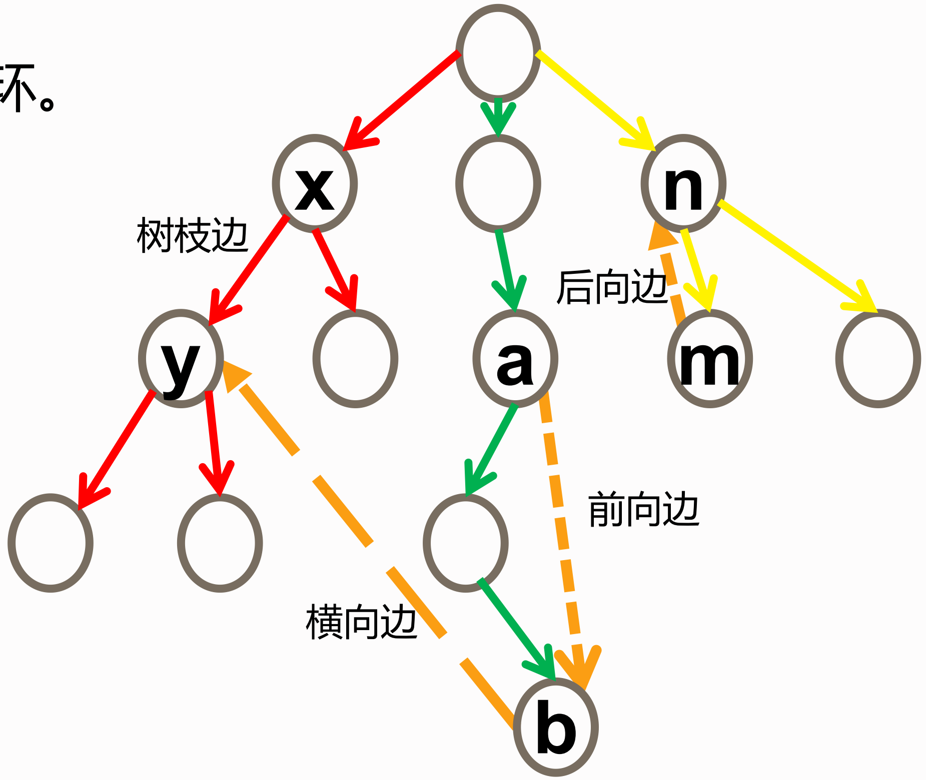
- 1. 搜索树上x的祖先结点，记为集合anc(x).

设y∈anc(x),若存在后向边(x,y)，则(x,y)与y到x的路径一起形成环。

- 2. 已经访问过,并且存在一条路径到达anc(x)的结点。

设z是这样一个点，从z出发存在一条路径到达y∈anc(x)。
若存在横叉边(x,z),则(x,z),z到y的路径、y到x的路径形成一个环。

综上所述，栈中的结点就是能与从x出发的”后向边”和“横叉边“形成环的结点。进而可以引入”追溯值“的概念。



强连通分量

追溯值

设subtree(x)表示流图的搜索树中以x为根的子树。**x的追溯值low[x]**定义为满足以下条件的结点的最小时间戳：

- 1. 该点在栈中；
- 2. 存在一条从subtree(x)出发的有向边，以该点为终点。

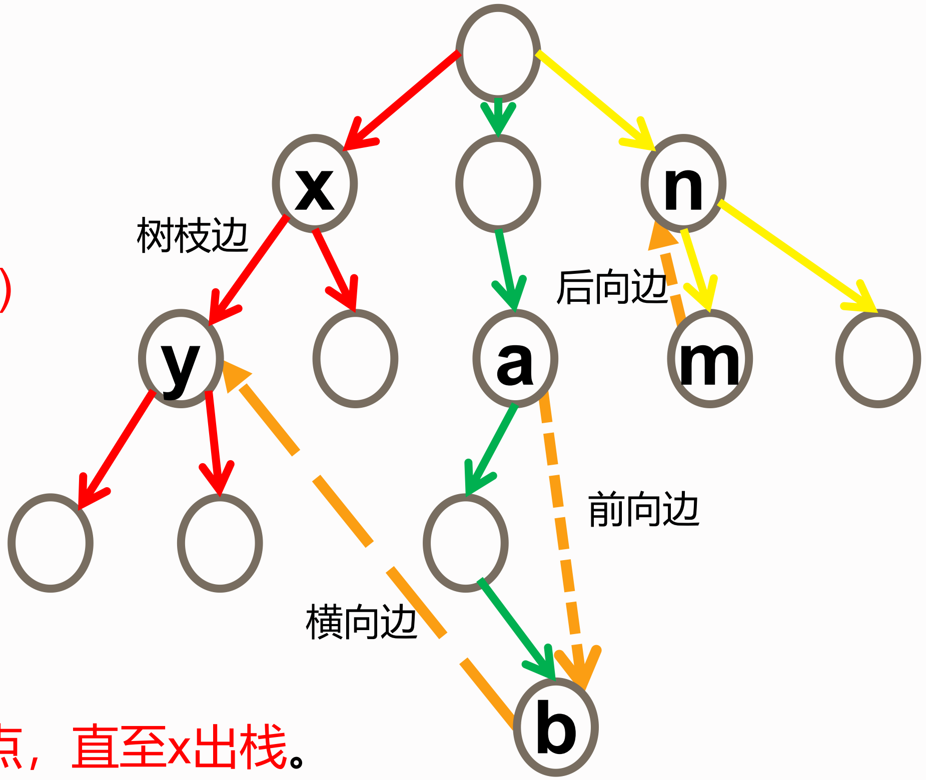
根据定义，Tarjan算法按照以下步骤计算“追溯值”：

1. 当结点x第一次被访问，把x入栈，初始化low[x]=dfn[x](时间戳timestamp)
2. 扫描从x出发的每条边(x,y)

(1) 若y没被访问，则说明(x,y)是树枝边，递归访问y，从y回溯之后，令low[x]=min(low[x],low[y]);

(2) 若y被访问过并且y在栈中，则令low[x]=min(low[x],dfn[y]);

3. 从x回溯之前，判断是否有low[x]=dfn[x]。若成立，则不断从栈中弹出结点，直至x出栈。



强连通分量

追溯值

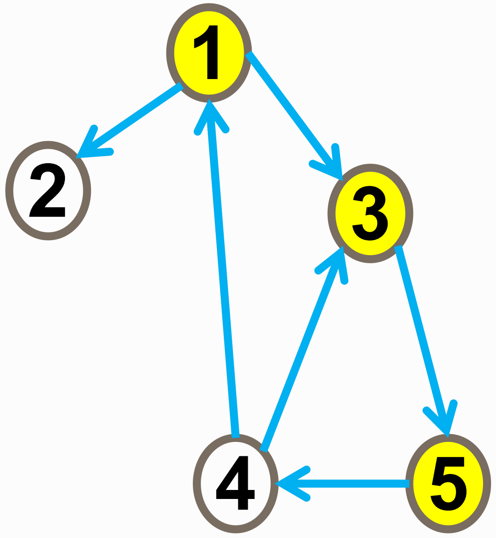
2. 扫描从x出发的每条边(x,y)

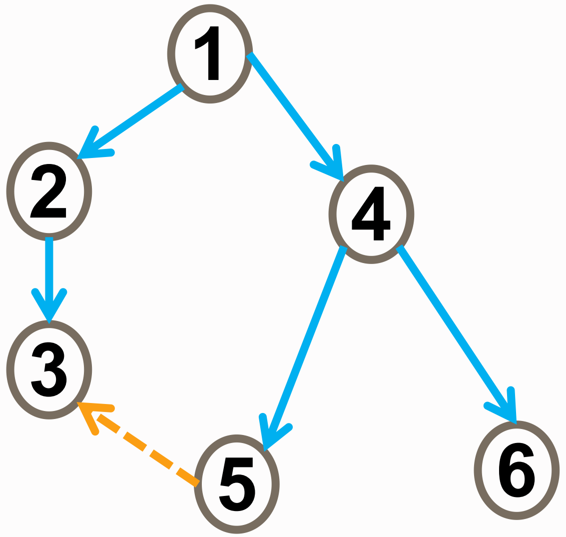
(1) 若y没被访问，则说明(x,y)是树枝边，递归访问y，从y回溯之后，令low[x]=min(low[x],low[y]);

y也许存在后向边到达比x还高的层,所以用y能到的最小dfn序(最高点)更新x能达到的(最小dfn序)最高点

(2) 若y被访问过并且y在栈中，则令low[x]=min(low[x],dfn[y]);

① y遍历回了遍历过的结点
遍历到结点4的时候，因为结点1和3一早就在栈里面了,所以low[4]可以被dfn[3]和dfn[4]更新，并且遍历回1，3结点的时候，都可以执行low[u] == dfn[u]，注意只会找到其中一个连通块，因为记录的时候,会出栈。
② y是x横叉边的点
遍历到结点5的时候，存在一条横叉边，使5遍历回了3，结点3早已经存在栈中，所以low[5]可以被dfn[3]更新了。





强连通分量

强连通分量判断法则

计算追溯值的第三步:

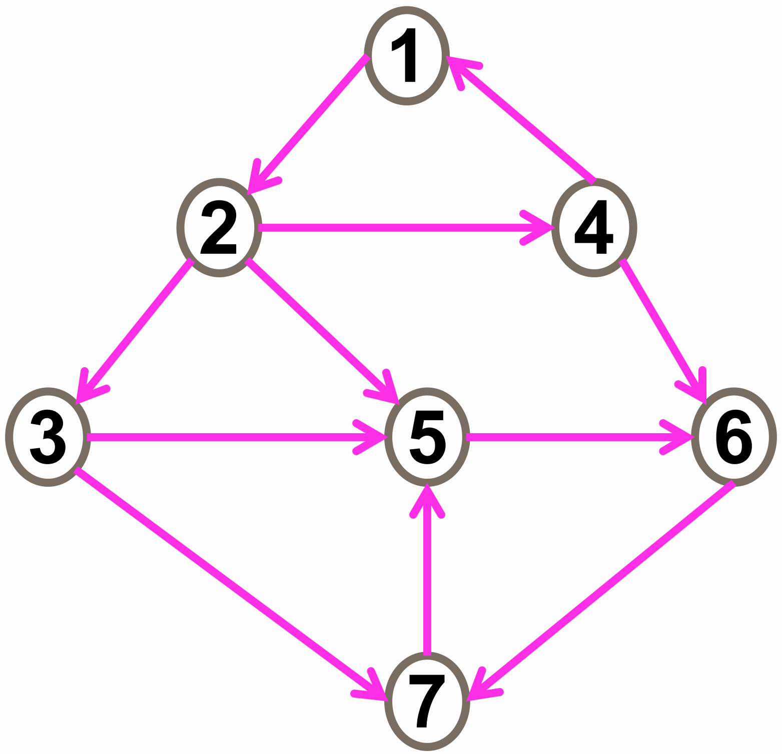
从x回溯之前，判断是否有low[x]=dfn[x]。若成立，则不断从栈中弹出结点，直至x出栈。

定理：在追溯值的计算过程中，若从x回溯前，有low[x] = dfn[x]成立，则栈中从x到栈顶的所有结点构成一个强连通分量。

在计算追溯值的第三步，如果low[x]=dfn[x],那么说明subtree(x)中的结点不能与栈中其他结点一起构成环。另外，因为横叉边的终点时间戳必定小于起点的时间戳，所以subtree(x)中的结点也不可能直接到达尚未访问的结点（时间戳更大）。综上，栈中从x到栈顶的所有结点不能与其他结点一起构成环。

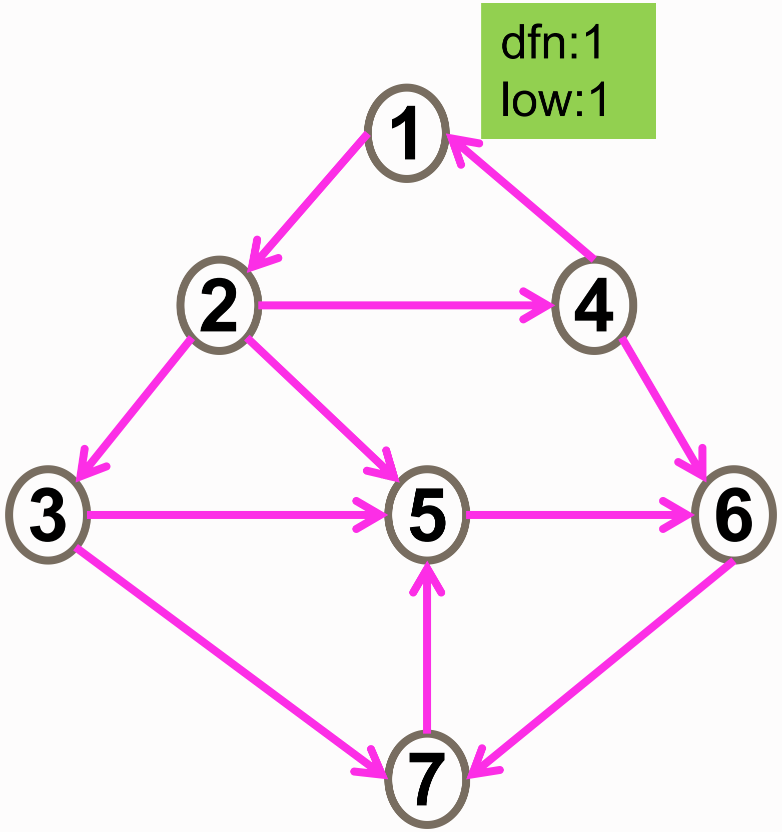
又因为及时进行判定和出栈操作，所以从x到栈顶的所有结点构成一个强连通分量。

Tarjan算法图示



强连通分量

Tarjan算法图示

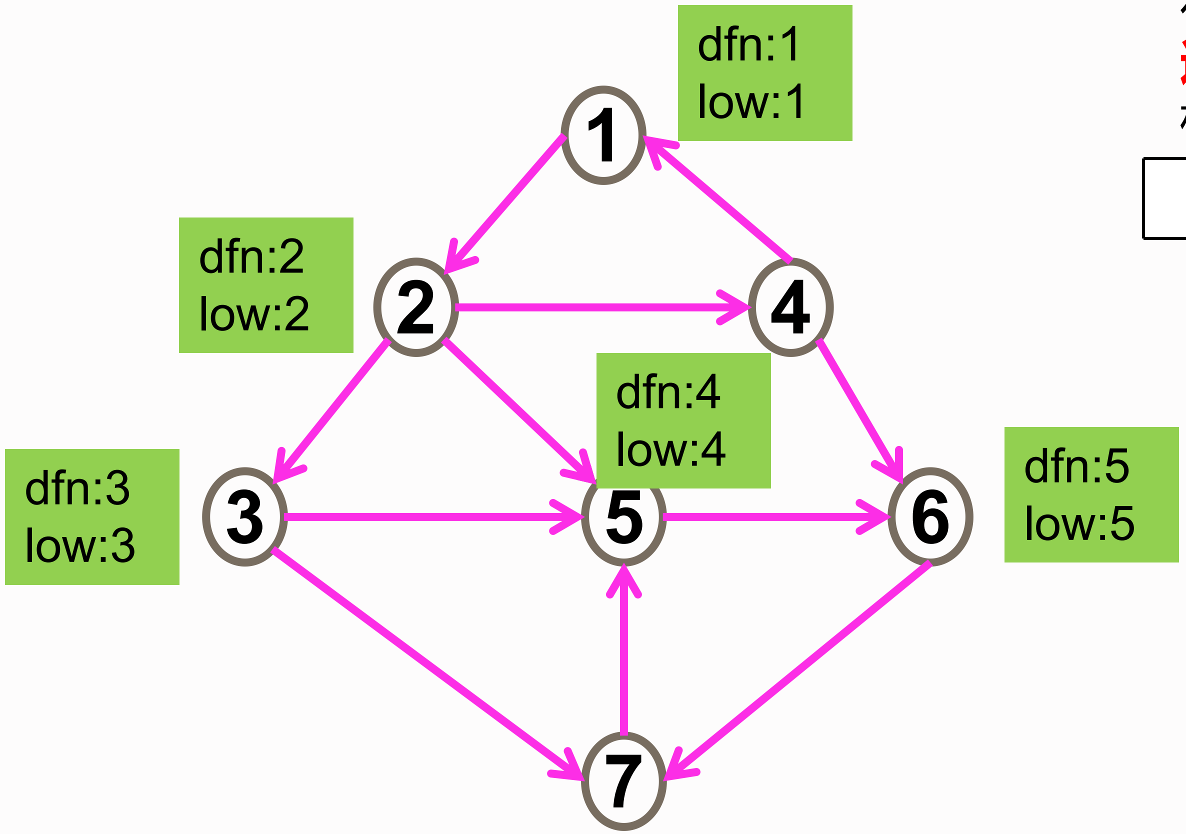


首先，从第一个点开始搜索，初始化它的dfn和low
栈内元素

1	

强连通分量

Tarjan算法图示



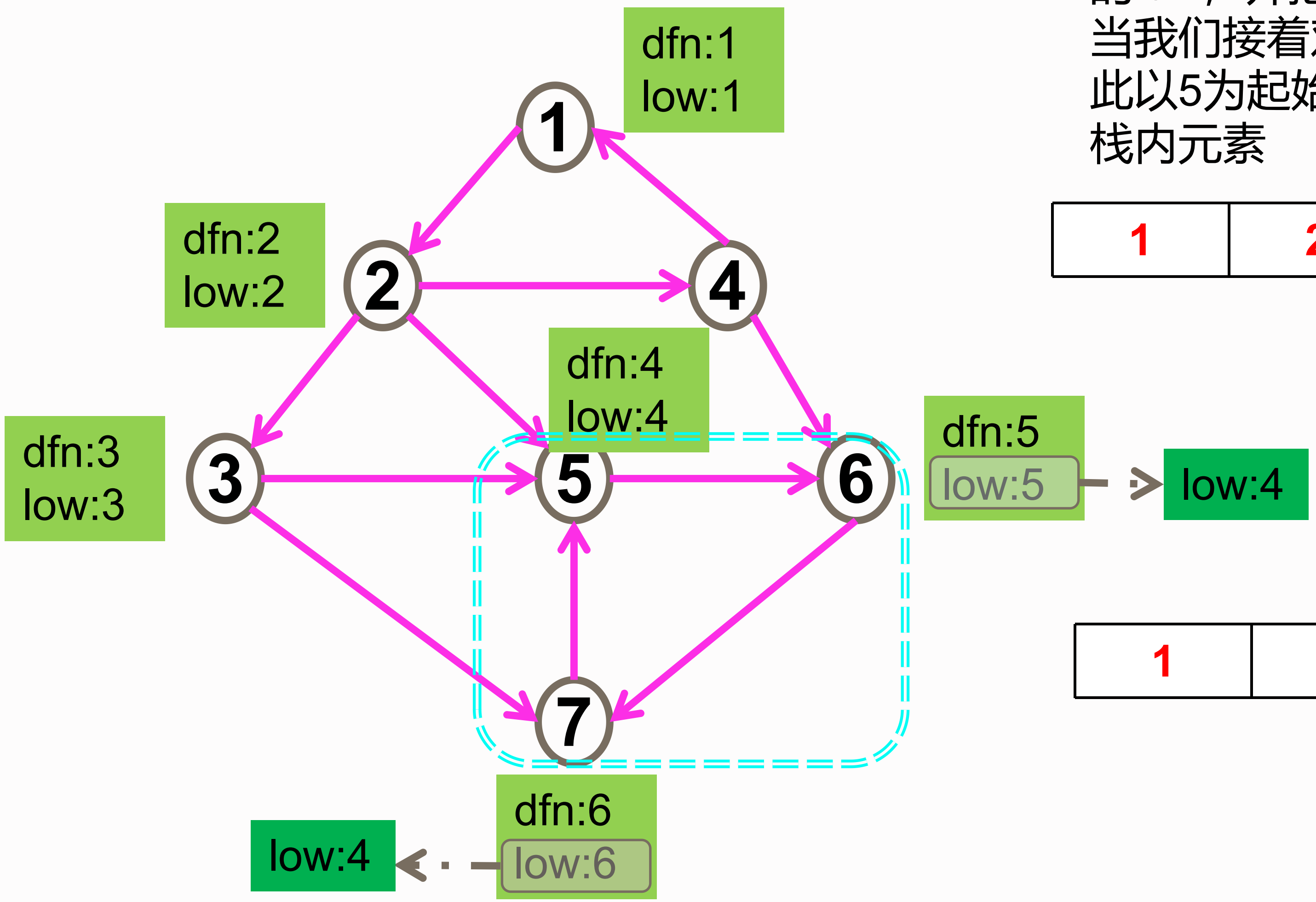
在没有遇到强连通分量前，按照**正常dfs进行搜点(此处跳过中间几步)**

栈内元素

1	2	3	5	6	

强连通分量

Tarjan算法图示



当搜索点7连接的点时，发现一个栈中的元素，接着更新它们的low，并把找到的第一个强连通分量除第一个元素依次弹出。当我们接着对点5搜索时，发现它已经没有其它指向的点，因此以5为起始点的强连通分量完全被找到了。把5也弹出。

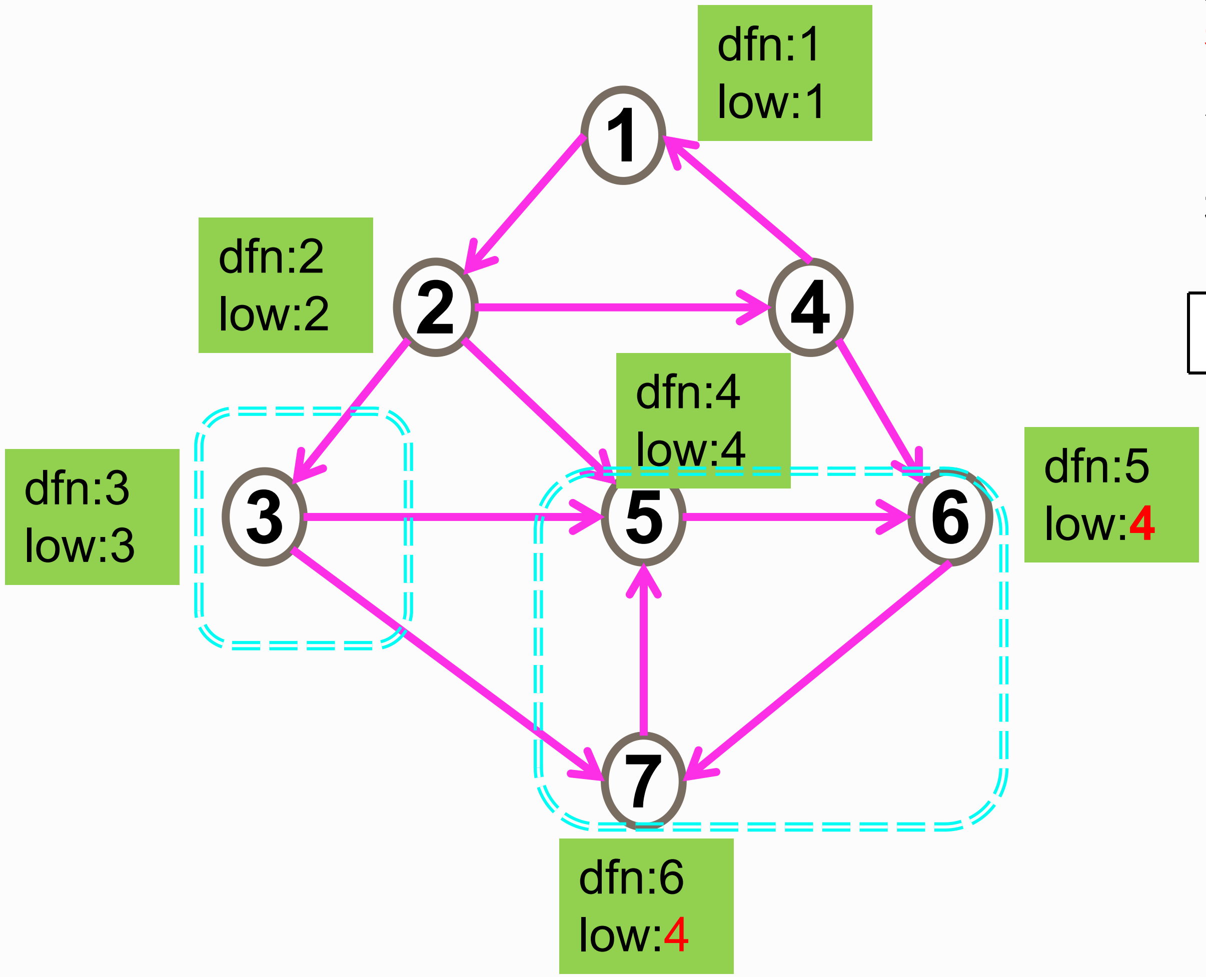
栈内元素

1	2	3	5	6	7	

1	2	3	

强连通分量

Tarjan算法图示

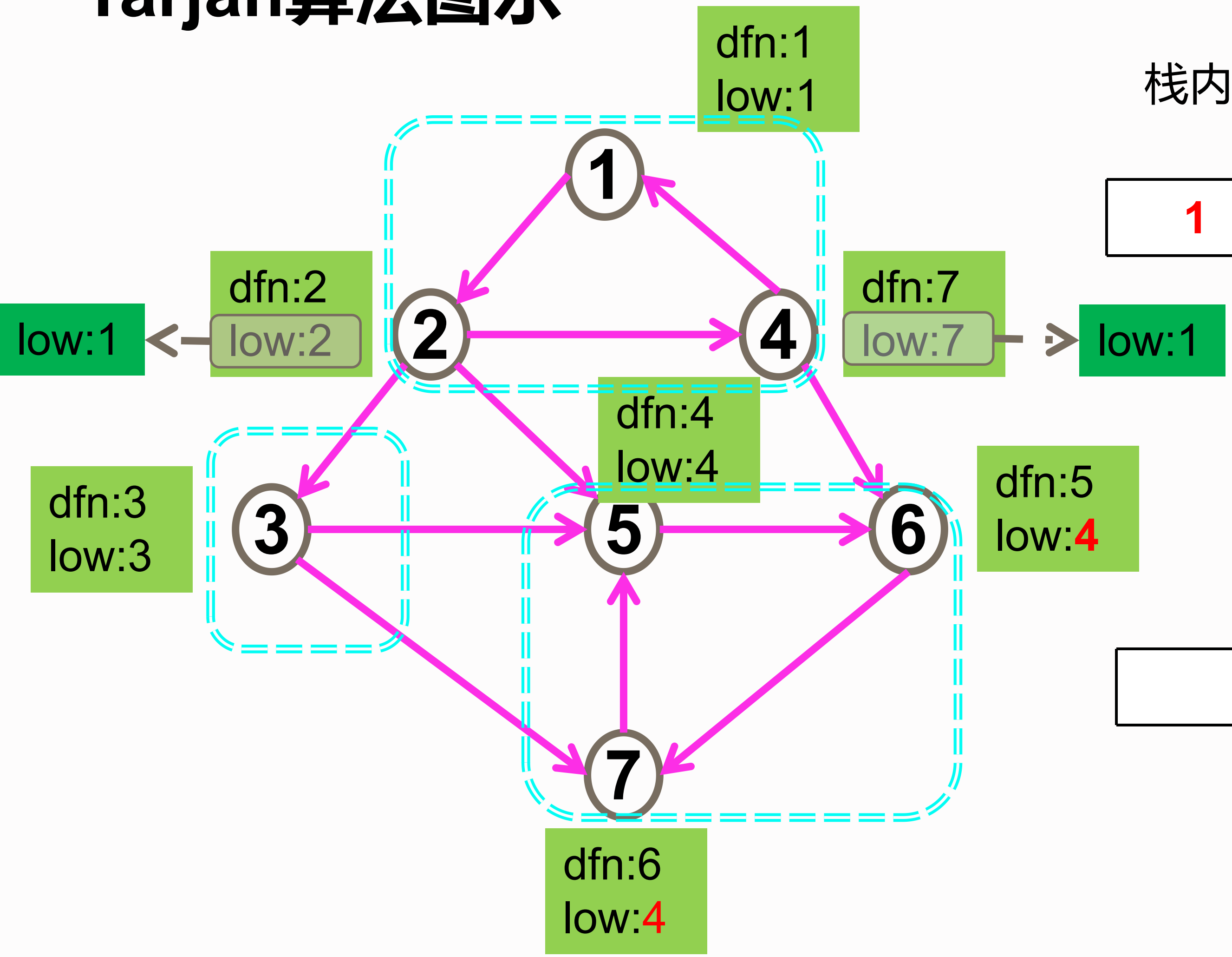


依据dfs的性质，回溯到点3继续进行搜索，搜索到点7，此时点7已经不再栈中，但是点7已经被搜索过了，我们只比较两者的low，看看是否能把点3加入到这个强连通分量中。显然点3比点7所在的强连通分量更早被发现，无法加入其中。此时点3没有剩余结点可以搜索，所以点3构成一个孤立的分量。栈内元素

1	2	

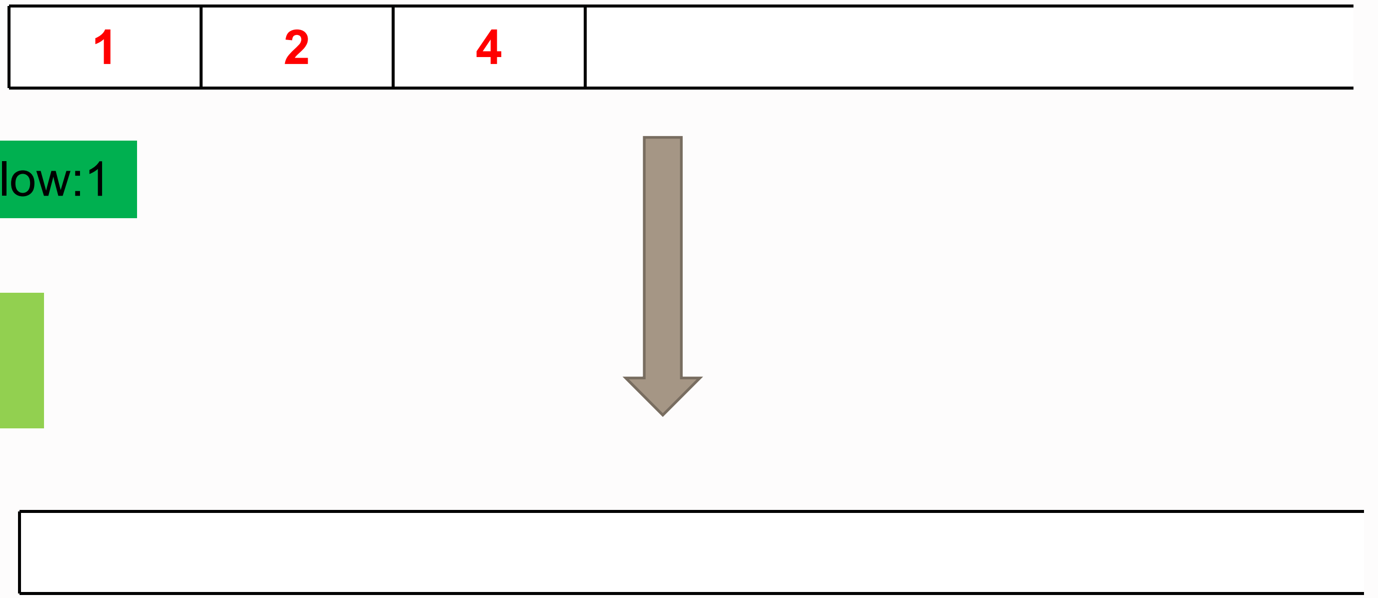
强连通分量

Tarjan算法图示



按照之前的原则推演，整个图被缩成三个分量

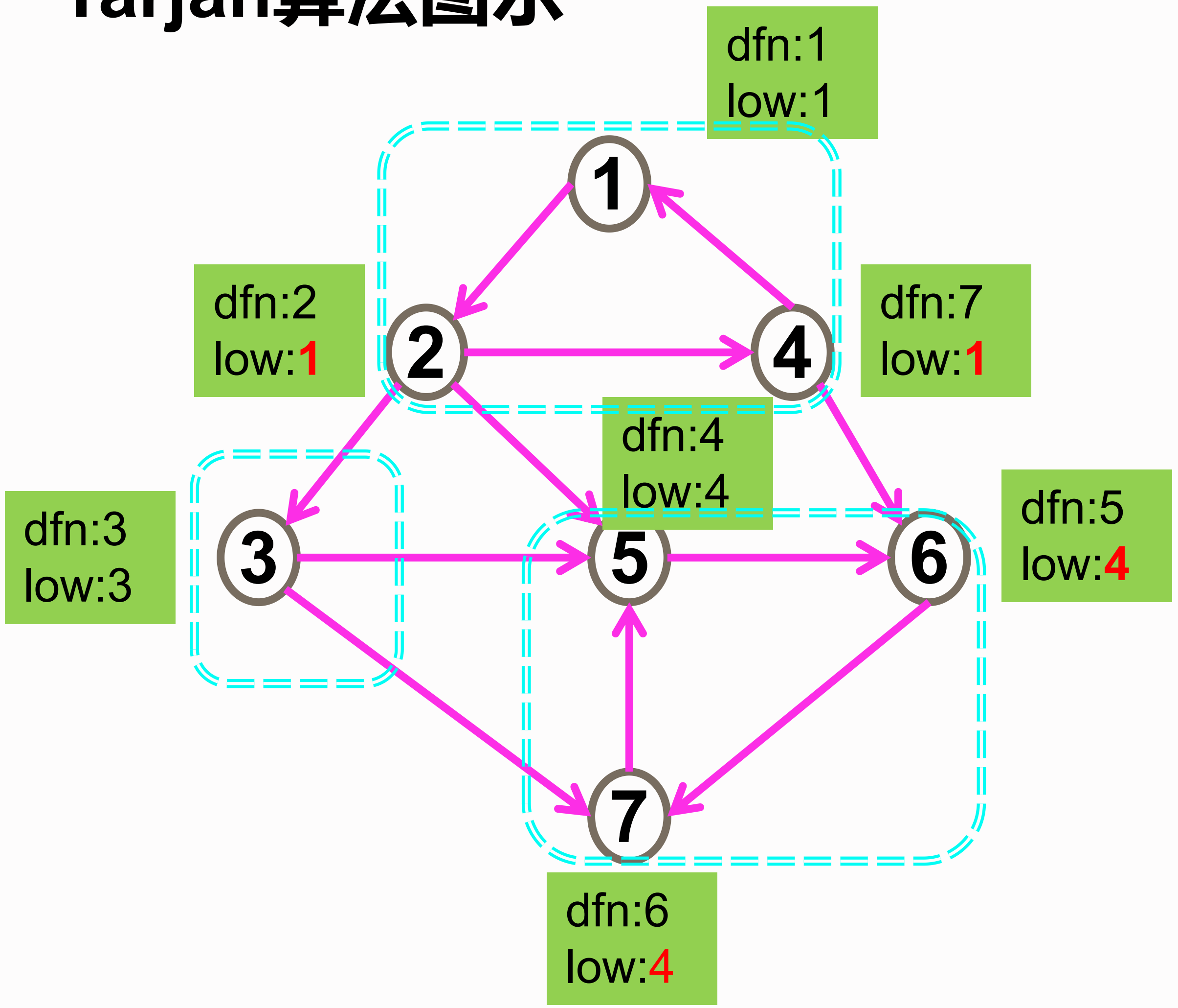
栈内元素

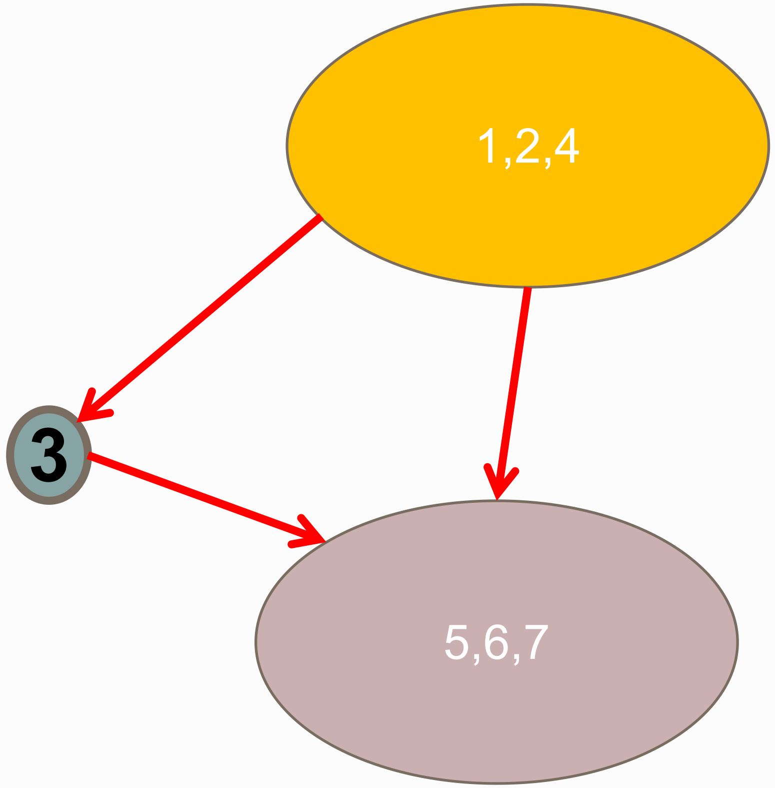


强连通分量

作用：将一个有向图缩点成有向无环图(DAG)

Tarjan算法图示

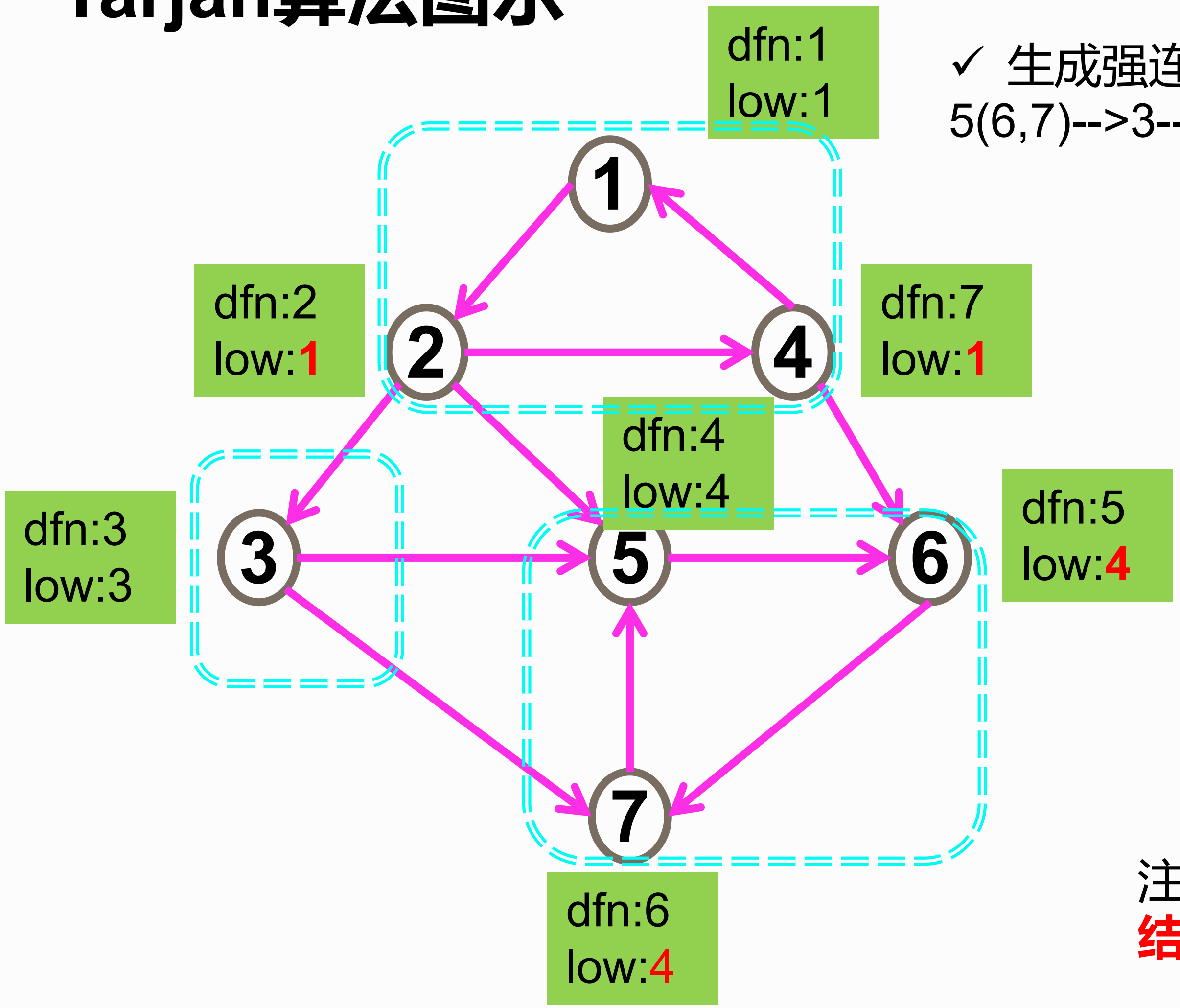




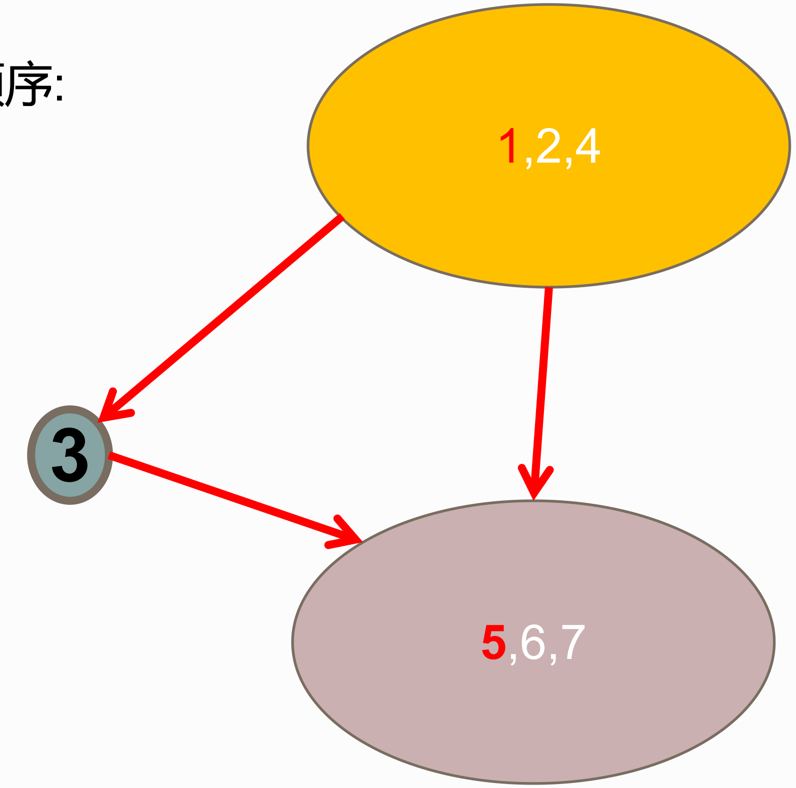
强连通分量

作用：将一个有向图缩点成有向无环图(DAG)

Tarjan算法图示



- ✓ 生成强连通分量的顺序:
5(6,7)-->3-->1(2,4)



- ✓ 该图的拓扑序是：1(2,4)-->3-->5(6,7)

注意的是做完tarjan算法之后,**拓扑序一定是按照强连通分量结点编号递减的顺序**，不需要再做一遍拓扑排序

强连通分量

缩点操作:我们也可以把每个SCC缩成一个点。对原图中的每条有向边 (x, y)，若id[x]≠ id[y]，则在编号为 id[x]与编号为 id[y]的SCC之间连边。最后，我们会得到一张有向无环图

下面的程序实现了Tarjan算法，求出数组id，其中id[x]表示x所在的强连通分量的编号。另外，它还求出了vector数组scc,scc[i]记录了编号为i的强连通分量中的所有结点。整张图有scc_cnt个强连通分量。

强连通分量

```
const int MAXN=100010,MAXM=1000010;

//链式前向星(new代表缩点后的图)
int head[MAXN],ed[MAXM],nex[MAXM],idx;
int head_new[MAXN],ed_new[MAXM],nex_new[MAXM],idx_new;

int dfn[MAXN],low[MAXN],timestamp;
//dfn[i]表示遍历到i的时间戳,timestamp时间戳计数
//low[i]表示从i开始走(遍历它的子树),所能遍历到的最小时间戳是什么(i结点当前所在强连通块的最高点)
int stk[MAXN],in_stk[MAXN],top;
//stk数组实现栈,top表示栈顶位置,in_stk[i]标记i是否在栈中
int id[MAXN],scc_cnt,scc_size[MAXN];//id[i]表示编号为i结点的强连通块编号
//scc_cnt表示强连通分量的个数,scc_size[i]表示编号为i的强连通分量中点的个数
vector<int> scc[MAXN];//scc[i]存储编号为i的强连通分量中的所有点
```

强连通分量

```
void init(){//初始化
    idx=idx_new=scc_cnt=top=0;
    memset(head,-1,sizeof(head));
    memset(head_new,-1,sizeof(head_new));
}
void add(int a,int b){//建立原图
    ed[idx]=b;
    nex[idx]=head[a];
    head[a]=idx++;
}
void add_new(int a,int b){//建立缩点图
    ed_new[idx_new]=b;
    nex_new[idx_new]=head_new[a];
    head_new[a]=idx_new++;
}
```

强连通分量

```
void tarjan(int u){//tarjan算法(采用数组栈实现)
    dfn[u]=low[u]=++timestamp;//初始化(第一次访问赋值新的时间戳)
    stk[++top]=u,in_stk[u]=1;//入栈操作,同时标记u在栈中
    for(int i=head[u];i!=-1;i=nex[i]){//遍历u的所有出边
        int j=ed[i];//u的一条出边的终点
        if(dfn[j]==0){//当前点没有遍历
            tarjan(j);
             //j也许存在反向边到达比u还高的层,
             //所以用j能到的最小dfn序(最高点)更新u能达到的(最小dfn序)最高点
            low[u]=min(low[u],low[j]);//更新low[u],注意此处是low[j]
        }
        else if(in_stk[j]==1)//j在栈中
            low[u]=min(low[u],dfn[j]);//更新low[u],注意此处是dfn[j]
    }
```

强连通分量

```
//回到了强连通块的最高点,那么就将这个强连通块的所有结点进行缩点
if(low[u]==dfn[u]){
    scc_cnt++;//强连通分量个数+1
    int y;
    do{
        y=stk[top--];//出栈
        in_stk[y]=0;//标记清0
        id[y]=scc_cnt;//赋值当前y的强连通分量的编号
        scc_size[scc_cnt]++;//size++
        scc[scc_cnt].push_back(y);//将点放入scc[scc_cnt]
    }while(u!=y);
}
}
```

强连通分量

```
int main()
{
    init();
    scanf("%d %d",&n,&m);
    for(int i=1;i<=m;i++)
    {
        int a,b;
        scanf("%d %d",&a,&b);
        add(a,b);
    }

    for(int i=1;i<=n;i++)
        if(dfn[i]==0)//当前未搜索过
            tarjan(i);
```

Tarjan算法代码链接

强连通分量

```
cout << scc_cnt << endl;//输出强连通分量的个数
//输出每个强连图分量的点
for(int i=1;i<=scc_cnt;i++){
    for(int j=scc[i].size()-1;j>=0;j--)
        cout << scc[i][j] << " ";
    cout << endl;
}
//缩点操作
for(int i=1;i<=n;i++)//枚举每一个点
    for(int j=head[i];j!=-1;j=nex[j]){//枚举i的每一条边
        int k=ed[j];
        if(id[i]!=id[k])//强连通分量的编号不同连接该边
            add_new(id[i],id[k]);
    }
return 0;
}
```