并查集

1.1 并查集的背景

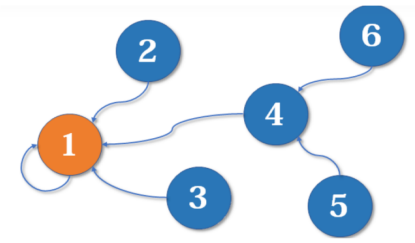
在历史上，分类的问题是一直伴随着人们的。在有了计算机后，我们该如何用计算机快速地区分元素的种类、判定一些元素是否属于同一个“等价类”呢？更具体一点，例如在图论中最小生成树的kruskal算法中，如何快速判断新加入的边和之前加入的边构成闭环了呢？处理这一类问题，便引入了并查集。

1.2 并查集的定义

并查集被认为是最简洁而优雅的数据结构之一，主要用于解决一些元素分组的问题。它管理一系列不相交的集合，并支持两种操作：

合并（Union）：把两个不相交的集合合并为一个集合。

查询（Find）：查询两个元素是否在同一个集合中。



并查集的概念图

并查集类概念代码实现

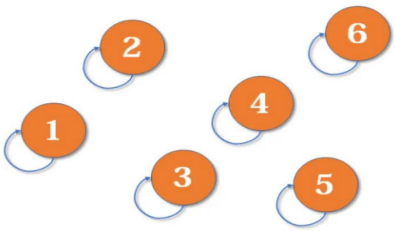
```
class Union_Find {
private:
    int father[Maxsize];// 每一个节点的父亲节点[并查集多用数组作为载体]
    //当然，我们也可以用结构体数组，只需保证每个结构体里面有一个int之类的元素存储根节点的编号就行。
    int element_NUM;//元素的个数
    int rank;//秩
    ///.........
public:
    Union_Find(int num) {
        element_NUM=num;
       }//简单的构造函数
    Union_Find(int* p1,int* p2,int num1,int num2)
    {
        for(int i=0;i<num1;i++)
        {
            father[i]=*(p+i);//把来自并查集1的元素直接赋值到现有数组；
        }
        for(int i=0;i<num2;i++)
        {

            father[i+num1]=*(p+num1+i)+num1;
        }
    }
    ~Union_Find() {}//析构函数
```

```
    int FIND(){}//查
    void merge(){}//并
    int GETRANK()
    {
        return rank;//得到并查集的秩;
    }
    int root()//获取并查集的根节点
    {
        return FIND(0);//对于任意一个节点，找到并查集的根节点后返回地址;
    }
    int GETNUM()//获取并查集的元素数目;
    {
        return element_NUM;
    }
    int* array_address()//获取并查集的根节点;
    {
        return &father[0];
    }
    //..........
    }
```

1.3 并查集_初始化INITIALIZE

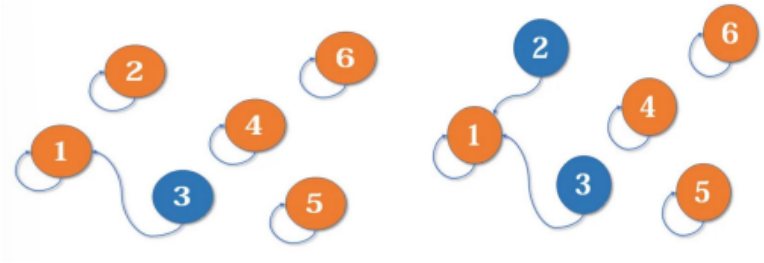
最开始,在没有构建并查集的时候，我们无法分别各个元素所属的"等价类"，此时不妨令他们自己是自己的根节点

```
class Union_Find {
private:
    //..........
public:
   // ..........
    void initialize()
    {
        for(int i=0;i<element_NUM;i++)
        father[i]=i;//每个节点令自己的节点号为自己根节点;
    }
    }
```

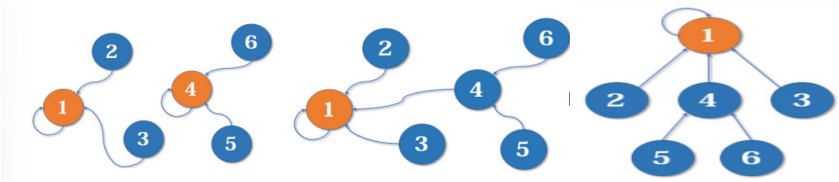


1.4 并查集_合并UNION

类似于生活中等价类的归并，不同的并查集是可以合成一个大的并查集的，如下图所示:



在上图的过程中，并查集{1}和并查集{3}先和成了一个新的并查集{1，3}其中元素1为根节点。之后并查集{1，3}和并查集{2}合成为了新的并查集{1，2，3}，其中1为根节点。

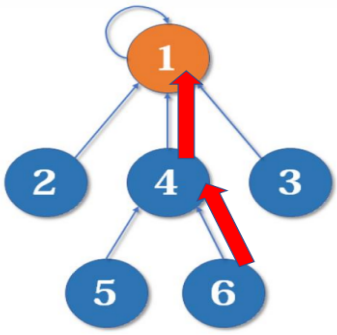


在这之后，{4}{5}{6}三个并查集合并成了一个并查集并且以4为根节点，随后{1，2，3}又和{4，5，6}合并成了一个较大的并查集，并且以1作为根节点。

注：由于还未引入并查集的Find操作，Union函数暂不给出。下面，我们介绍一下并查集的另外一个基础操作——Find。

1.5.1 并查集查询——Find

上文中提到的所谓的等价类的划分，都是凭借着一个元素的特有的性质或者“标识”进行的。并查集中，每一个元素均有着一个“根节点”用于指示其所存在的等价类，即若两个元素拥有相同的根节点，便可以确定它们是一个等价类。递归地，若两个元素根节点的根节点相同，甚至二者跨越了很多的“根节点”最终查询到了共同的祖先，我们也说它们同属于一个等价类。因此我们不难给出以下用来实现FInd操作的递归函数：




```
int FIND(int x)//查询下角标为x的元素的根节点是谁
{
    if(father[x]==x)
    return x;//一个元素最终根节点有个特点，即它一定是自己是自己的根节点；
    else
    return FIND(fater[x]);
}

```

1.5.2 merge函数——合并操作

有了FIND的操作后，我们可以很自然地写出合并操作——merge函数的代码了！

```
//本函数没有基于类实现，基于类的实现我们将要在后续的“按秩压缩”中运用。
//在“按秩压缩”一节中，类的思想会运用得更多一些；
void MERGE(int father[], int i,int j)//把i所在的并查集并到j所在的并查集上
{
    father[FIND(i)]=FIND(j);
//找到i的祖先所在的位置并且把其设成以j的祖先为自己的祖先；
}
```

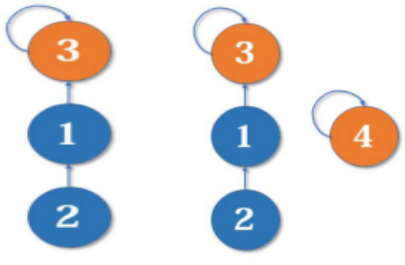
在引入两个并查集的基本操作函数后，我们简述一下并和查的时间复杂度：

在一个包含n个元素的并查集中，进行m次查找或者合并操作的的时间复杂度的最坏的情况为O(m×α(n))。

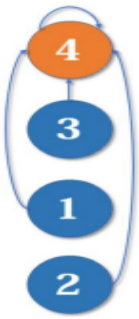
其中：
α(n)是一个随着n的增长非常缓慢的函数，对于n约为10的80次方数量级的时候，α(n)<=4。

1.6 并查集——路径压缩

更快的查询效率是我们使用并查集的初衷，在最好情况下，确立一个元素的根的时间复杂度为O(1)，但是并查集在使用的时候有时也会变得效率退化：



如图，对于并查集{1，2，3}，我们要是想寻找到{2}的祖先{3}的话，是需要经过{1}这个元素的过渡的。换句话说，如果我们此时再把{4}加到{2}的下面——即让{2}作为{4}的根节点的时候，再想要查询{4}的祖先的话，我们所需要经过的“查询路径”会变得更长。我们很容易意识到，如果对一个深度非常深的并查集来说，想要查询比较深处的节点的某个祖先时间复杂度是会比较高，为此，我们引入路径压缩:

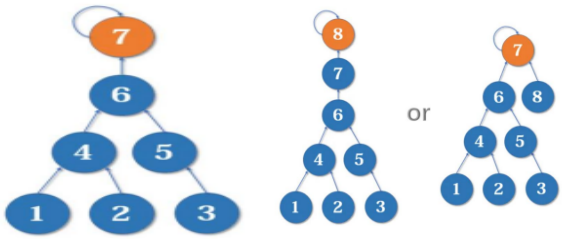


如图所示，当我们确定下来{2}的祖先是{4}的时候，我们不妨把2的根节点直接定义为{4}，此时我们可以递归地 把为了查询到{4}所经过的节点{1}{3}的根全部定义成为{4}。这样的话，我们通过把并查集的搜索路径压缩从而为以后的查询减少了负担——即我们不再通过二元关系的传递性去一点一点查找祖先，而是直接把祖先设立为根节点。我们给出路径压缩的示例代码如下：

```
int COMPRESSION(int x)//对father数组下标为x的元素进行压缩路径；
{
    if (x==father[x])//如果找到了祖先节点，就返回祖先节点的下标；
    return x;
else
{
    father[x]=COMPRESSIO(father[x]);//把遍历过的所有的节点的根节点设立成为搜索到的祖先节点；
}
}
//当然我们可以用问号表达式把上述的语句写成一行的形式，请大家自行给出；
//对于如何维护rank的问题，一般来说，当元素数目很多的时候，精准的rank的数值显得没那么重要。请读者到这里自行思考如何维护并查集的rank(在一定误差内便可)；
```

1.7 并查集——按秩压缩

秩的概念大家是很熟悉的，在线性代数中，秩指的是线性无关行的/列个数，表征矩阵的奇异性。在并查集中，我们一般可以定义一个并查集形成的树状结构的深度为它的秩。一般来说，并查集的秩越小，其查询的速度越快。对于已经存在的并查集来说，在新加入只有一个结点并查集的时候，我们可以使用路径压缩的方法来降低并查集的秩。但是，当我们想合并不同的并查集的时候呢?



举个例子，上图中，对于已经存在的并查集{1,2，3，4，5，6，7}我们想它和并查集{8}并入到一起，很明显，把{8}并到{1,2，3，4，5，6，7}上更加好，因为这样合并之后的新的并查集的秩更小，得到的新的并查集查询速度更快，并且路径压缩的时间复杂度更小，这说明，我们引入秩的概念是很有重要的。下面给出示例代码：

```
Union_Find MERGE_BY_RANK(Union_Find a,Union_Find b)//在按秩压缩前提下的merge函数。形参为两个哈希表的实体化的对象；
{
    Union_Find temp;//实体化对象temp,用于装载两个并查集合并之后的新并查集。
    //当然，我们也可以采用把一个并查集合并到另外一个上面的方法，两者之间有所不同；
    int  *p1,*p2,num1,num2;//用于访问数组,和统计分别多少元素；
    p1=a;p2=b;//
    int rank1=a.GETRANK();
    int rank2=b.GETRANK();
    num1=a.GETNUM();
    num2=b.GETNUM();
    element=num1+num2;//元素数目为二者相加；
    p1=a.array_address();
    p2=b.array_address();
        for(int i=0;i<num1;i++)
        {
            father[i]=*(p1+i);//第一步先把第一个并查集赋值给新的并查集；
        }
        for(int i=0;i<num2;i++)
```

```
        {
            father[num1+i]=*(p2+i)+num1;
        }
        if(rank1>rank2)
        {
            father[b.root()+num1]=father[a.root()]//再把并查集的新的根节点重新设定;
            rank=rank1;//从新设置并查集的秩;
        }
        else
        {
            father[a.root()]=father[b.root()+num1];
            rank=rank2;
        }
}

```

Hash-Table 哈希表

2.1 哈希表的引入和定义

我们学过很多的查询元素的方法，比如在一个有序数组中，我们可以通过折半查找的方法以O(logn)的复杂度查询到某个元素是否存在。可是，当元素规模变得庞大的时候、或者元素本身的结构变得复杂的时候(例如学生学号等)，我们再进行如“排序”“查找”的操作也至少需要O(nlogn)的时间复杂度。
因此，为了追求效率，我们可不可以寻找到一种结构，可以更快速的找到元素呢？如果存在的话，其时间复杂度是多少呢？(O(n)?O(logn)?.....)答案是：在哈希表的帮助下，我们的速度将达到O(1)!

下面给出哈希表的定义：哈希表——将一组关键字映射到一个有限的、地址连续的区间上，并以关键字在地址集中的“像”作为相应纪录在表中的存储的位置。即我们寻找一个函数(我们称之为哈希函数)，它把元素映射为计算机的地址，这样的话每次查找一个元素的时候，只需进行相应的函数值(哈希值)的计算，再在相应地址确认元素存不存在即可。下面给出哈希表的C++类的示例：

```
template <typename T>

class hash_table
{
private:
    T element[maxsize];//一般来说，hash表装载的元素是很长的字符串(多用string，例如学生学号)，或者数字等等;
    int element_num;
public:
hashTable(){element_num=0;}
void init()
{
    int num;
    T temp;
    cin>>num;//输入hash_table的数目;
    if(num>maxsize)
    {
        cout<<"error in init!"<<endl;
        return -1;
    }

    for(int i=0;i<num;i++)
```

```
    {
        cin>>temp;
        element[hash_func(temp)]=temp;
        //此处未考虑hash冲突，hash函数使用的细节，只是给出处理思想;
    }
    return 0;
}
int hash_func(){}//哈希函数;
void insert(){}//插入元素;
//......
};
```

2.2 哈希函数

哈希表的思想很好理解，但是哈希表的一个难点和最重要的一点就是构建哈希函数——即我们希望创造出来一个元素和地址之间的一一对应的关系，但是这在绝大多数情景下都是很困难的！当哈希表的规模远远超过元素的数目的时候，我们可以“不那么精心地”设计一个哈希函数去把元素映射到一个“大空间中”，但是这样的方法来说，空间复杂度是很高的！当我们试图压缩冗余的空间时，新的麻烦又出现了：不同的元素出现了被哈希函数作用之后映射到了同一个地址的情况，我们称之为“哈希冲突”。

首先，让我们花一部分精力到哈希函数上——即哈希函数应该又如下的特点：

1、简单，在较短时间之内计算出来结果；

2、定义域必须包含全部的关键字；

3、散列值应该在限定的地址范围只内；

4、理想的散列函数应该尽量随机；

5、对每一个输入，输出在值域上面应该等概率被取值到，利于减少冲突的发生；

下面我们给出几种有代表性的散列函数实现方式：

一、乘法散列：

对于key∈(0,1)，对其乘以一个常数M得到哈希值，即：

$$Hash(key)=M\times key;$$

对于更加一般的情况，key∈(s,t)，只需将其归一化：

$$Hash(key)=M\times(k-s)/(t-s);$$

乘法散列的特点：为一个线性散列，故我们对于key的取值有要求，要求其尽量分布均匀否则容易出现地址聚集的现象。下面给出示例函数：

```
//乘法散列函数;
int hash_Muti(T key,int M)//T为键值，可以为int等，M为乘法散列的系数，根据不同的输入可以有不同的取值;
{
    return M×key;
}
//或者归一化的乘法散列:
int hash_Muti(int s,int t,T key,int M)
{
    return M×(key-s)/(t-s);
}
```

二、模散列

一般来说，设定一个很大的质数M，对键值Key进行取模得到散列地址(M<表长)。

Hash(Key)=Key mod M;

理论上来讲，只要素数相比较Key的规模来说“充分大”，便可以很好的解决冲突的问题，但对于大质数的寻找是比较困难的事情，而且进行的计算操作代价会上升，所以我们一般都模散列和乘法散列结合,得到如下的散列形式:

Hash(Key)=(Key*a)mod M;

其中a我们常取黄金分割比0.618，原因是因为0.618有很好的的性质——它可以让元素更加均匀分布。

```
//模散列函数
int hash_Mod(int Key)
{
    return (int)(0.618*Key)%M;
}
```

三、数字分析法

对于n个d位的数，每一位可能有r种不同的符号，这r种符号出现的频率不尽相同。此时，我们根据可以根据hash表的规模的大小选取这些数字中“分布均匀”的某些位置作为进行散列的地址——例如，对于工人的编号，前几位几乎相同(公司信息、岗位分工等等)，最后几位不同，此时我们可以取最后几位作为存储的地址。
例如如下的程序:

```
int hash_numanalyse(int a[])//输入了工人的编号,取后四位作为哈希值;
{
    int address=0;
    for(int i=0;i<4;i++)
    {
        address+=a[maxsize-4+i]*pow(10,3-i);//得到相应的地址;
    }
    return address;
}

```

四、平方取中法

首先，我们先计算构成关键字标志符内码(如asicc码)(或者标志符本身就是int或者short类型)的平方，然后按照哈希表的大取求和得数的若干位作为散列地址。一般来说，此方法散列能力较好，因为平方的过程中，所有位的信息都被集中到了一起。下面给出示例：

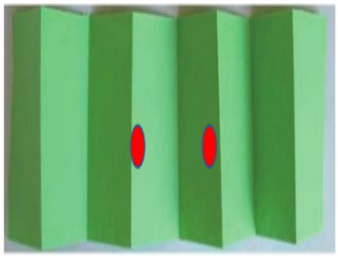
```
int hash_sqarefunc(int a[])//不妨还是假设输入的是工人的编号；
{
    int sum=0;
    for(int i=0;i<lenth;i++)
    {
        sum+=a[i]*pow(10,i);
    }
    sum=sum*sum;//平方；
    int address=0;
    for(int i=0;i<4;i++)
    {
        address+=(sum%(pow(10,i+1)))/(pow(10,i))*pow(10,i);//直接取后四位作为地址；
    }
    return address;
}

```

五、折叠法

所谓的折叠法就是把关键字从左到右分为和表长一样的很多部分(最后的部分可以短一点)，然后把这些数按某种规则加起来。这里给出两种常用的方法：

*1,移位法：把分割成的各个部分的最后一位对齐，相加后得到散列地址：Key=23938587841，则hash(key)=239+385+878+410=1912(这里简单地补上0，也可以根据需求用其它数补位甚至不补位)。

*2，分界法：把各个部分数据沿各个部分来回折叠，然后对齐相加；举个很简单的例子就是折纸的例子，你把一张纸来回对折后是"每一部分相对的点"重合到了一起：



上图所示，两个红点叠到了一起!
Example:Key=23938587841，则hash(key)=239+583+878+041=1741;

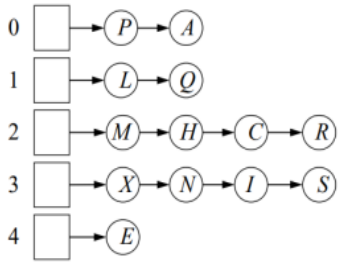
```
int hashf_yiwei(int a[])//移位法,按照三个一组分割;
{
    int len=strlen(a);//数组规模;
    int address=0;
    for(int i=0;i<strlen(a);;)
    {
        if(len>3)
        {
            address+=a[i]*100+a[i+1]*10+a[i+2];
        }
        else if(len==2)
```

```
        {
            address+=a[i]*10+a[i+1];//对齐
        }
        else if(len==1)
        {
            address+=a[i];
            //对齐;
        }
        len-=3;//
        i+=3;//
    }
    return address;
}



int hashf_fenjie(int a[])//分界法,按照三个一组分割;
{
    int len=strlen(a);//数组规模;
    int address=0;
    bool flag=1;//折叠时候该字符段正反方向的判断;
    for(int i=0;i<strlen(a);;)
    {
        if(len>3)
        {
            if(flag){
            address+=a[i]*100+a[i+1]*10+a[i+2];
            }
            else
            {
                address+=a[i+2]*100+a[i+1]*10+a[i];
            }
        }
        else if(len==2)
        {
            if(flag)
            address+=a[i]*100+a[i+1]*10;
            else
            address+=a[i+1]*10+a[i];
        }
        else if(len==1)
        {
            if(flag)
            address+=a[i]*100;
            else
            address+=a[i];
        }
        len-=3;
        i+=3;
        flag=!flag;
    }
    return address;
}

```

2.3 哈希冲突

所谓的哈希冲突(hash Collisions)定义如下：如果两个元素的关键字散列到了同一个地址，便称之发生了哈希冲突。无论怎么精心地去设计散列函数，都不可以完全避免冲突的发生。为此，我们一般采取如下的解决方法：开散列法、闭散列法。

所谓的开散列法是很容易理解的：对每个散列的地址建立一个链表，把散列到同一个地址的关键字存入链表中；



对于链地址法，我们可以定义装载因子为一个已经存入的元素个数N和表长M的比值N/M(链地址法支持N>M)。这时候，平均查找时间为O(N/M)。可见当N变得非常大的时候，哈希表的查找速度退化。

```
template <typename T>
struct hash_node
{
    T element;
    hash_node *next;
    bool load;//用于表示是否装载了元素；
    hash_node()
    {
        next=NULL;
        load=0;
    }
}
/*
注意，此时要对之前的hash表的类实现进行一点element类型修正；
class hash_table
{
private:
    hash_node element[maxsize];//一般来说，hash表装载的元素是很长的字符串(多用string，例如学生学号)，或者数字等等；
    int element_num;
    ......
}
*/
//我们基于T为int型变量给出一个实现的方式；
void list_hash(int a)
{
    int address=hash_func(a);
    if(element[address].load)
    {
        hash_node *p;
        p=element[address];
        while(p->load)
        p=p->next;
        p.element=a;
```

```
        p.load=1;
    }
    else
    {
        element[address].element=a;
        element[address].load=1;
    }
}
```

所谓的闭散列法(又称为开放定址法)：依靠存储空间余下部分解决冲突。元素不想开散列法一样"挂在"哈希表外，而是直接存在了哈希表中，故称之为闭散列法。

探测：检查给定的表位置上是否存在一个与带搜索关键词不同的元素称之为探测。最简单的探测是线性探测。

线性探测：当冲突发生时，即元素插入的时候已经被其它元素所占据的时候，我们顺序检查下一个位置；如果碰到空位置，便在空位上插入待插入的元素。若碰到了表尾，则回到表头继续搜索。

线性探测的缺点：随着哈希表的元素数目增加，会出现如元素聚集的现象，导致哈希表线性探测运行变得很慢；*为此我们可以用如下方法：

*双重散列：即不是直接检测下一个位置是否是空的，而是采用第二个散列函数得到一个固定的增量的序列。这样的话，探测效率会得到提升，并且避免了聚集。

对于第二个散列函数有如下的要求:

1、不能出现散列值为0的情况；

2、散列值需要和表长互质，否则某些探测序列会很短并且有可能陷入无限次地查找中。

下面给出线性探测的示例代码：

```
template <typename T>
struct hash_node
{
    T element;
    bool load;//用于表示是否装载了元素；
    hash_node()
    {
        load=0;
    }
}

/*
注意，此时要对之前的hash表的类实现进行一点element类型修正；
class hash_table
{
private:
    hash_node element[maxsize];//一般来说，hash表装载的元素是很长的字符串(多用string，例如学生学号)，或者数字等等；
    int element_num;
    ......
}
*/

//我们基于T为int型变量给出一个实现的方式；
void linersearch_hash(int a)
```

```
{
    int address=hash_func(a);
    if(element[address].load)
    {
        int search_address=address;
       while(element[search_address].load)
       {
        if(search_address>maxsize)
        search_address=0;//回到表头
        if(element[search_address].load==0)
        {
            element[search_adddress].element=a;
            element[search_address].load=1;
            break;
        }
        else
        {
            search_address+=1;
        }
       }
    }
    else
    {
        element[address].element=a;
        element[address].load=1;
    }
}

```

2.4 散列的删除

删除表中的某个散列对于哈希表来说是一个基本的操作。

对于链接地址法来说，删除的操作很简单，只需要简单地进行链表的删除就好。

对于开放定址的散列方法来说，一般来讲是不可以直接删除的。其原因在于开放定址散列中，元素除了携带着自身的信息之外还携带者探测的一些信息。比如一个元素a的哈希值为x，但是插入时候x位置有元素b了，因此a就放到了x+1位置上，当我们删除b的时候，想要搜索到a是不可能的，因为b信息丢失了！
为此我们有解决方案：1、元素加以标志后继续留在表中。2、对所有元素重新散列；

下面给出继续解决方案一的散列删除代码：

```
template <typename T>
struct hash_node
{
    T element;
    bool load;//用于表示是否装载了元素；
    bool delete;//表征元素是否被删除；
    hash_node()
    {
        load=0;
        delete=0;
    }
}
```

```
13
14  /*
15  注意，此时要对之前的hash表的类实现进行一点element类型修正；
16  class hash_table
17  {
18  private:
19      hash_node element[maxsize];//一般来说，hash表装载的元素是很长的字符串(多用
    string，例如学生学号)，或者数字等等；
20      int element_num;
21      ......
22  }
23  */
24
25  //我们基于T为int型变量给出一个实现的方式；
26  void delete_hash(int a)
27  {
28      int address=hash_func(a);
29      element[address].delete=1;//表征这个元素被“删除”了(或者称之为无视了更好)，但是其
    还在hash表中。这样之后要是有新的元素插入的时候，直接改变element[address].element的值并
    且置element[address].delete=0;
30  }
31
32
```

2.5 散列的应用之一——桶排序

依托哈希表的强大的查询功能，桶排序这种算法应运而生。(回忆高级排序算法需要时间复杂度O(nlogn)，能否更快？)

考虑利用哈希表和最简单的哈希函数hash(key)=key,我们可以实现一直快速的排序方法：

1、构造哈希表，初始化时间O(M);

2、把所有待排序的元素插入表中(采用开散列)，耗时O(N);

3、从头到尾遍历hash表，读出非空的元素，
耗时O(M);

整体耗时：O(M+N);

注意当M~N时候，此时时间复杂度为O(N)。并且桶排序也是一种稳定的排序方法!之所以桶排序是O(N)速度的排序是因为它不基于“比较”排序。

附录：参考文献以及图片来源

并查集部分的图片来源和资料参考：https://zhuanlan.zhihu.com/p/93647900;

2022年《数据与算法》课程戴凌龙老师PPT;

《数据结构》邓俊辉，清华大学计算机科学与技术系;