第八次讨论课

课程知识回顾与整理

经典算法设计技术研讨

（一）数据结构

1.线性结构；

2.树结构；

3.图结构；

（二）算法

4.查找；

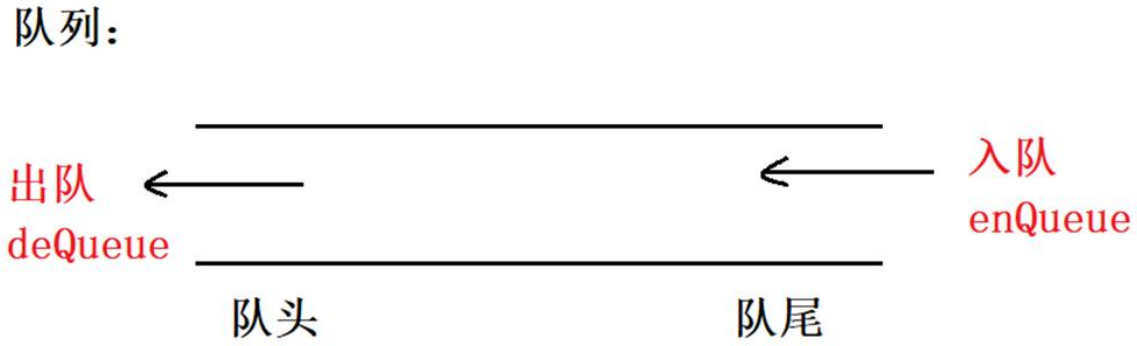
5.排序；

6.经典算法设计技术

线性结构

算法思想

线性结构 队列




```
queue<int >q;   //创建队列
        q.front();   //队列第一个元素
        q.pop();     //出队
        q.push();    //入队
        q.size();    //队列大小
        q.back();    //返回一个指向队列的最
后一个元素 的 引用
        q.empty();    //判空
```

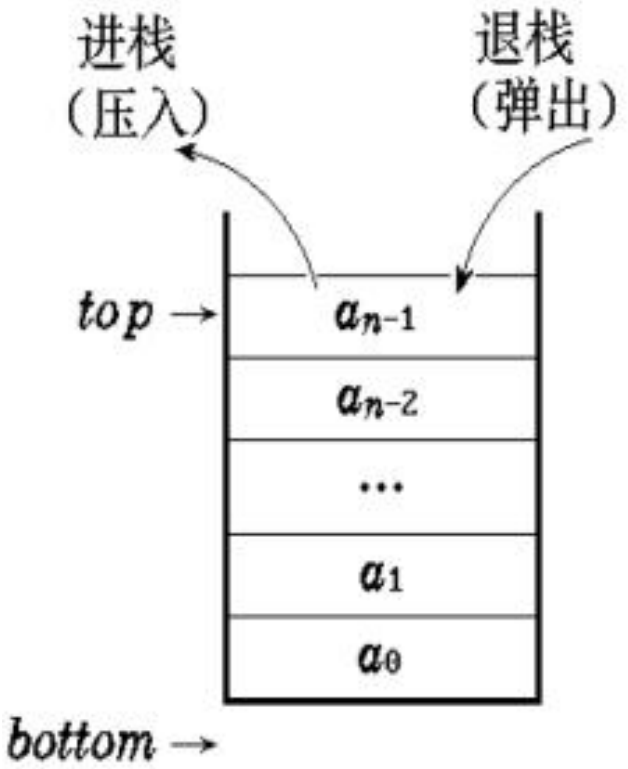
算法思想

线性结构 栈

后进先出

只允许一端进行插入和删除，此端叫做栈顶。
另一端称为栈底。

```
stack<int > s;
s.push(); //压入元素
s.pop();  //弹出元素
s.empty();              // 判断元素是否为空
```



算法思想

线性结构 向量 Vector

vector < int > v；定义一个vector容器。
v1 = v2 //把v1的元素替换为 v2 中元素的副本。
vector<int>v(10);
v.push_back(-1) //将-1推入向量末尾
v.pop_back() //删除向量末尾元素

v.size() //返回向量中元素个数

v.clear() //清除向量中所有元素，向量长度归0

v.insert(it,x) //在向量a的迭代器it所指位置插入元素x

v.erase(it) //删除迭代器it指代的元素

v.erase(it1,it2) //删除区间[it1,it2)内的所有元素

算法思想

线性结构 线性表

```
template <typename E> class List { // List ADT
private:
    void operator =(const List&) {}      // Protect assignment
    List(const List&) {}                 // Protect copy constructor
public:
    List() {}                 // Default constructor
    virtual ~List() {} // Base destructor
    virtual void clear() = 0;
    virtual void insert(const E& item) = 0;
    virtual void append(const E& item) = 0;
    virtual E remove() = 0;
    virtual void moveToStart() = 0;
    virtual void moveToEnd() = 0;
    virtual void prev() = 0;
    virtual void next() = 0;
    virtual int length() const = 0;
    virtual int currPos() const = 0;
    virtual void moveToPos(int pos) = 0;
    virtual const E& getValue() const = 0;
};
```

算法思想

```
template <typename  E> class LList:public  List <E> {
private:
    Link<E>*  head;              //  Pointer  to  list  header
    Link<E>*  tail;              //  Pointer  to  last  element
    Link<E>*  curr;              //  Access  to  current  element
    int  cnt;                                //  Size  of  list

    void  init()  {                  //  Intialization  helper  method
    head = tail = curr = new Link<E>;
    cnt=0;
    }

    void  removeall()  {      //  Return  link  nodes  to  free  store
    while(head !=NULL)
    head = head->next;
    }

public:
    LList(int  size=500)  {
        init();          //  Constructor
    }
    ~LList()  {
        removeall();          //  Destructor
    }
    //  使用空格分隔输出线性表中的所有数据，并最终换行
    //无数据时直接输出空行
    void  print(){
    int num = cnt;
    curr = head->next;
    while(num!=0)
        {
    cout<<curr->element<<" ";
    curr=curr->next;
    num--;
        }
    cout<<endl;
    }
```

```
void  clear()
{
    removeall();          //  Clear  list
    init();
}

void  insert(const  E&  it)
{
    curr->next =new Link<E>(it ,curr->next);
    if(tail == curr)  tail = curr->next;
    cnt++;
}

void  append(const  E&  it)  {  //  Append  "it"  to  lis
    tail = tail->next = new Link<E>(it,NULL);
    cnt++;
}

E  remove()  {
Assert(curr->next  !=  NULL,  "No  element");  //如当前元
E it = curr->next->element;
Link<E>* Itemp = curr->next;
if(tail ==  curr->next) tail = curr;
curr->next = curr->next->next;
delete Itemp;
cnt--;
return it;
}

void  moveToStart()  {
curr = head;
}

void  moveToEnd()  {
curr = tail;
}

void  prev()  {
if(curr==head) return;
Link<E>* temp = head;
while(temp->next!=curr)
temp = temp->next;
curr = temp;
}
```

算法思想

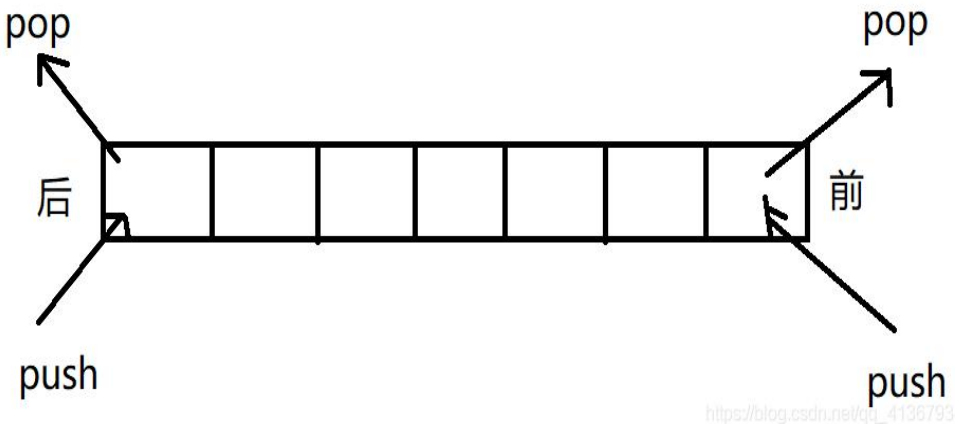
线性结构

```
void  next()  {
    if(curr!=tail)
    curr = curr->next;
    }

int  length() const {
    int num=0;
    Link<E>* temp = head->next;
    while(temp!=NULL)
    {
    temp = temp->next;
    num++;
    }
    return num;
}

int  currPos()  const  {
Link<E>* temp = head;
    int i;
       for(i=0;curr!=temp;i++)
       temp =temp->next;
return i;
}

void  moveToPos(int  pos)  {
Assert  ((pos>=0)&&(pos<=cnt),  "Position  out  of  range");
int it;
curr = head;
for(it=0;it<pos;it++){
curr=curr->next;
}
}

const  E&  getValue()  const  {
Assert(curr->next  !=  NULL,  "No  value");
return curr->next->element;
}
```

算法思想

线性结构 双端队列

push_back：在后面插入元素
push_front：在前面插入元素
pop_back：删除最后一个元素
pop_front：删除第一个元素
get_back：返回最后一个元素
get_front：返回第一个元素
empty :如果队列为空，则返回true
full :如果队列已满，则返回true



算法

1、排序

排序算法时间复杂度

排序算法	平均时间复杂度	最好情况	最坏情况	空间复杂度	排序方式	稳定性
冒泡排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	In-place	不稳定
插入排序	$O(n^2)$	$O(n)$	$O(n^2)$	$O(1)$	In-place	稳定
希尔排序	$O(n \log n)$	$O(n \log^2 n)$	$O(n \log^2 n)$	$O(1)$	In-place	不稳定
归并排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$	Out-place	稳定
快速排序	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$	In-place	不稳定
堆排序	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(1)$	In-place	不稳定
计数排序	$O(n + k)$	$O(n + k)$	$O(n + k)$	$O(k)$	Out-place	稳定
桶排序	$O(n + k)$	$O(n + k)$	$O(n^2)$	$O(n + k)$	Out-place	稳定
基数排序	$O(n \times k)$	$O(n \times k)$	$O(n \times k)$	$O(n + k)$	Out-place	稳定

算法思想

选择排序

简单选择排序：最简单的选择方法是顺序扫描序列中的元素，记住最小的元素（一次扫描完毕就找到了一个最小的元素。反复扫描就能完成排序工作）。
每次选择出序列最小的元素依次进行排序。

10. (单选题,5.0分) 有一组关键字序列（50,40,95,20,15,70,60,45,80），采用简单选择排序方法进行递增排序，假定每一轮比较均是选择最小值，则第4交换和选择后未排序记录序列（即无序表）为（　　）。

A. 50,95,70,60,80

B. 50,70,60,95,80

C. 50,60,70,80,95

D. 50,70,95,60,80

50	40	95	20	15	70	60	45	80	
50	40	95	20	15	70	60	45	80	第一选择
15	40	95	20	50	70	60	45	80	第一交换
15	40	95	20	50	70	60	45	80	第二选择
15	20	95	40	50	70	60	45	80	第二交换
15	20	95	40	50	70	60	45	80	第三选择
15	20	40	95	50	70	60	45	80	第三交换
15	20	40	95	50	70	60	45	80	第四选择
15	20	40	45	50	70	60	95	80	第四交换，后半部分是无序表

算法思想

选择排序

```
void selsort(int  *A, int n) {
        for (int i=0; i<n-1;i++) {
         int lowindex=i;

        for (int j=n-1; j>i;j--)    // Find least

        if (A[j]<A[lowindex]))
         lowindex=j;

         swap(A, i, lowindex);

}
```

算法思想

快速排序

1. (分录题,20.0分)

以关键字(15,18,29,12,35,32,27,23)为例，请依次写出执行shaffer教材上快速排序算法的前两趟的排序结果。（枢轴选取第一个关键字，选取后与最后一个交换位置）

注：

（1）答案形如：15 18 29 12 35 32 27 23

（2）填写答案时，数字之间只能有一个英文空格，不能有其他符号，如逗号，否则系统会判为0分。

正确答案：

(1) 12 15 29 23 35 32 27 18

(2) 12 15 18 23 27 29 35 32

算法思想

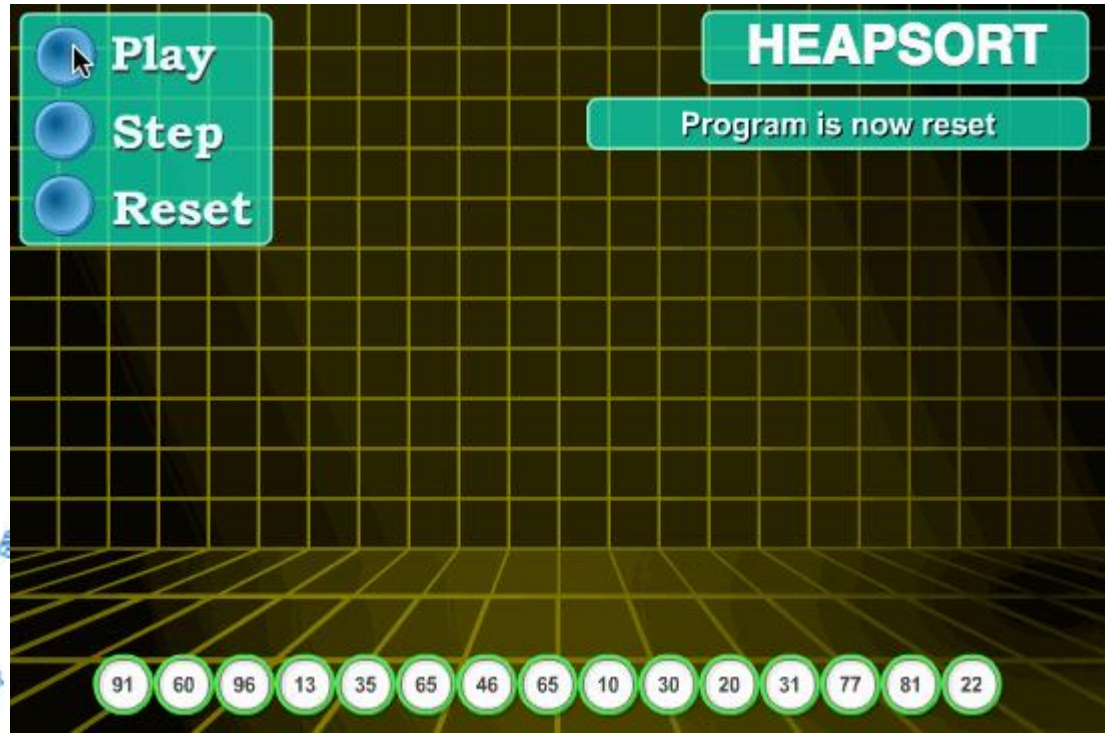
快速排序

```
inline int findpivot(int *A, int i, int j)
{
    return (i + j) / 2;
}
inline int Partition(int *A, int left, int right, int& pivot)
{
    do
    {
    while(A[++left] < pivot);
    while((left < right) && (pivot < A[--right]));
        swap(A[left], A[right]);
    }
    while(left < right);
    return left;
}
```

```
void qsort(int *A, int left, int right)
{

if(left >= right) return;

int pivotindex = findpivot(A, left , right);

swap( A[pivotindex] , A[right]);

int k = Partition(A, left-1 , right , A[right]);

swap( A[k], A[right]);

qsort(A, left , k-1);
qsort(A ,k+1 , right);

}
```

算法思想

堆排序




```
void adjust(int arr[], int len, int index)
{
    int left = 2*index + 1;
    int right = 2*index + 2;
    int maxIdx = index;
    if(left<len && arr[left] > arr[maxIdx]) maxIdx = left;
    if(right<len && arr[right] > arr[maxIdx]) maxIdx = right;  // maxIdx是
    if(maxIdx != index)                    // 如果maxIdx的值有更新
    {
        swap(arr[maxIdx], arr[index]);
        adjust(arr, len, maxIdx);          // 递归调整其他不满足堆性质的部分
    }

}
void heapSort(int arr[], int size)
{
    for(int i=size/2 - 1; i >= 0; i--)  // 对每一个非叶结点进行堆调整(从最后一个非叶结点开始)
    {
        adjust(arr, size, i);
    }
    for(int i = size - 1; i >= 1; i--)
    {
        swap(arr[0], arr[i]);              // 将当前最大的放置到数组末尾
        adjust(arr, i, 0);                 // 将未完成排序的部分继续进行堆排序
    }
}
```

算法思想

基数排序

1. (填空题,10.0分)
对数据序列（265,271,751,139,317,863,742,594,056,238），采用最低位优先的基数排序进行升序排序，第1趟后的排序结果为（ ）（注：序列中数字以英文的逗号分隔,直接写数字序列）。

1	251	751
2	742	
3	863	
4	594	
5	265	
6	056	
7	317	
8	238	
9	139	

正确答案:

(1) 271,751,742,863,594,265,056,317,238,139;271，751，742，863，594，265，056，317，238，139

算法思想

直接插入排序

1. (填空题,5.0分) 对一组数据(4,48,96,23,12,60,45,73)采用直接插人排序算法进行递增排序，当把60插人到有序表中时，为寻找插入位置需比较（　）次。

4	48	96	23	12	60	45	73	
4	48	96	23	12	60	45	73	
4	48	96	23	12	60	45	73	
4	48	96	23	12	60	45	73	
4	23	48	96	12	60	45	73	
4	12	23	48	96	60	45	73	
4	12	23	48	60	96	45	73	60比较2次
4	12	23	45	48	60	96	73	
4	12	23	45	48	60	73	96	

算法思想

冒泡排序

9. (单选题,5.0分) 采用冒泡排序法（按从前往后冒泡）对序列{20,78,13,7,34,41,56,92,62}进行升序排列，如果从前往后比较，第一趟冒泡的结果为_____。

A. 7,20,78,13,34,41,56,62,92

B. 13,20,7,34,41,56,78,62,92

C. 20,13,7,34,41,56,78,62,92

D. 7,13,20,34,41,56,62,78,92

20	78	13	7	34	41	56	92	62
20	78	13	7	34	41	56	92	62
20	13	7	34	41	56	78	92	62
20	13	7	34	41	56	78	62	92

78比中间的数都大，需要冒泡到56的位置

算法思想

归并排序

```
void mergesort(int *A,int *temp , int left , int right){      //数组，临时数组，左下标，右下标
        if (left == right) return;
        int mid = (right + left)/2;
mergesort(A, temp , left, mid);
mergesort(A, temp , mid+1, right);

        for (int i = left ;  i <=right ; i++)
        {
                temp[i]= A[i];
        }
int i1 = left; int i2= mid +1;
        for( int curr= left ; curr <= right ; curr++ ){
                if(i1 == mid+1)
                A[curr] = temp[i2 ++];

                else if(i2 > right)
                A[curr] = temp [i1++];

                else if(temp[i1] < temp[i2])
                A[curr] = temp[i1++];

                else A [curr] = temp [i2++];
```

算法思想

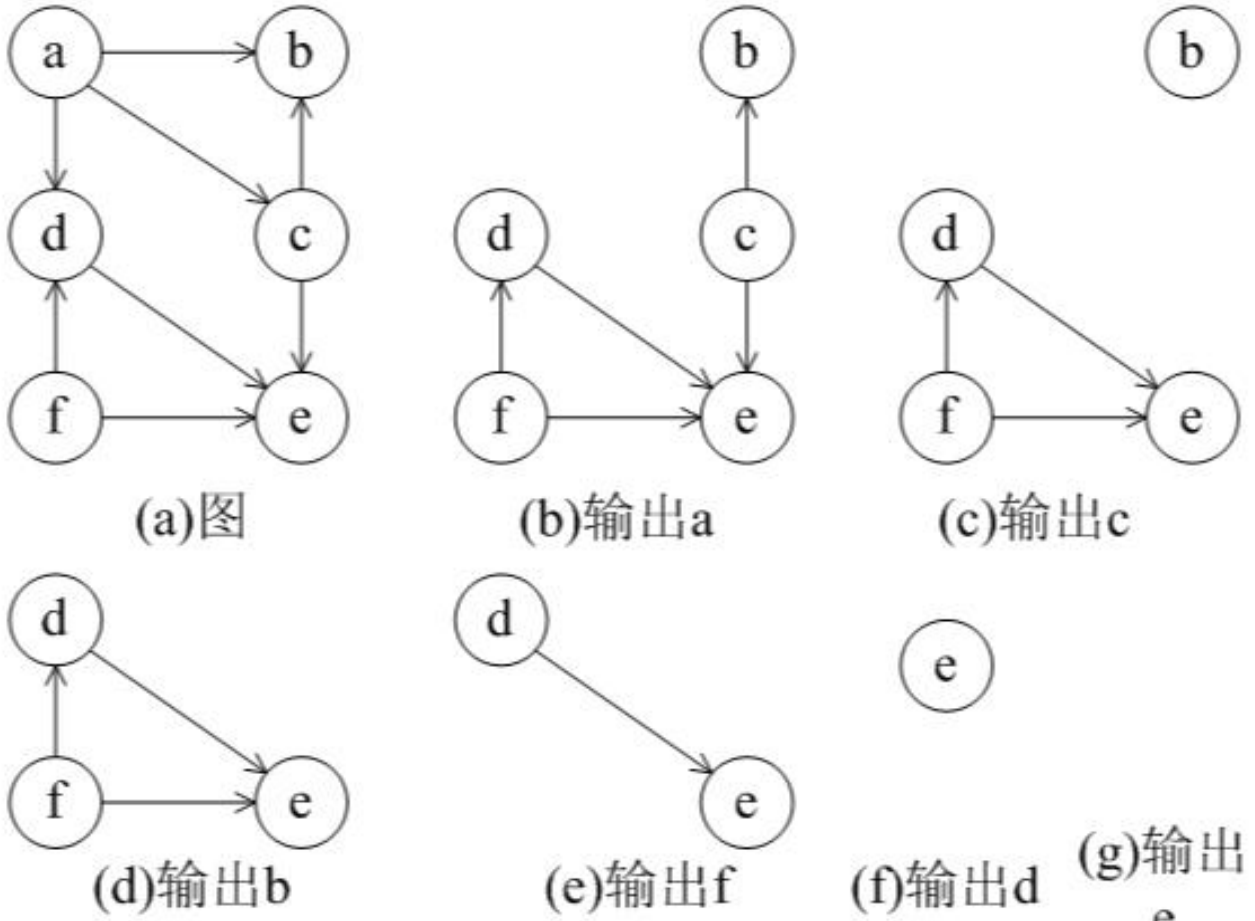
希尔排序

```
void insort(int *A, int n ; int incr){
        for( int i = incr; i < n ; i += incr){
        for ( int j = i ; ( j>=incr )&&(A[j]<A[j-incr]); j -
=incr)
                        swap(A[j],A[j-incr]);
        }
}
void shellsort(int *A, int n){
        for(int i=n/2; i>2 ; i/=2){
                for(int j = 0 ; j < i; j++ ){
                        insort(&A[j], n - j, i );
                }
        }
        insort(A, n, 1);
}
```

算法思想

拓扑排序

先统计所有节点的入度，对于入度为0的节点就可以分离出来，然后把这个节点指向的节点的入度减一。
一直做改操作，直到所有的节点都被分离出来。



拓扑排序序列：acbfde

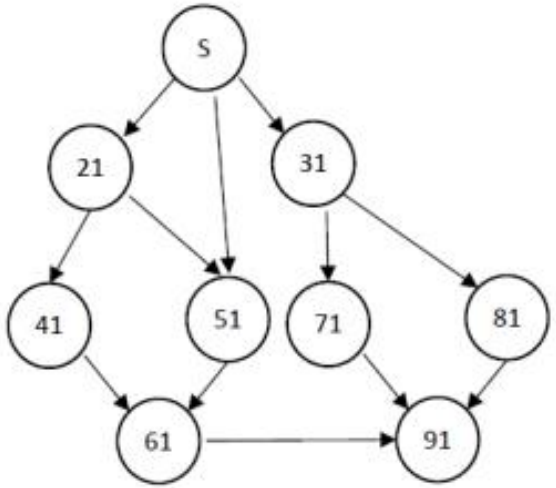
图一

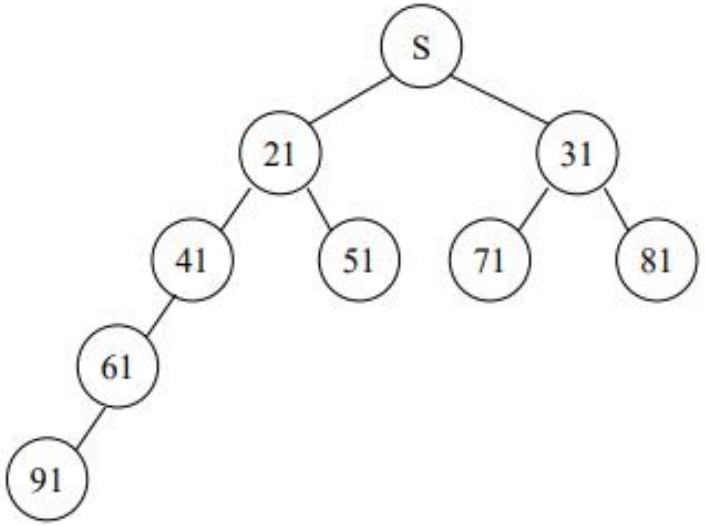
算法思想

- 对于下图

（1）画出起始点为S 的DFS 搜索树；

（2）并简要描述拓扑排序的算法思想，给出一个拓扑排序序列。





先统计所有节点的入度，对于入度为0的节点就可以分离出来，然后把这个节点指向的节点的入度减一。

一直做改操作，直到所有的节点都被分离出来。

故答案为：S 21 41 61 91 51 31 71 81

按照字典序输出：S 21 31 41 51 61 71 81 91

排序例题

1，概念理解

1，稳定性：
稳定性和不稳定性的定义
通俗地讲就是能保证排序前两个相等的数其在序列的前后位置顺序和排序后它们两个的前后位置顺序相同。

举个简单的应用场景
比如一次考试成绩排序，学校规定本次考试分数前三名同学可以分别获得一等奖，二等奖，三等奖。A同学本次考试获得了第一名无可争议，他获得了一等奖。但是由于本次考试有两个人分数相同，我们设这两个人是B同学和C同学，他们两个并列第二名，但是二等奖只有一个，那该这个时候就应该用稳定排序来处理。即如果B同学上一次考试的成绩比C同学的分数高，那么即便这次考试B和C成绩相同，B同学也理应排在第二名（获二等奖），C同学应该排在第三名（获三等奖）。但是如果用了不稳定排序的话，可能会造成B同学最终第三名，拿到了三等奖，这显然对他不公平。**(但是不要沉迷于过去的辉煌)**

2,影响排序效率的因素

排序过程中，耗时主要来源于两个方面：比较次数，交换次数。但是也有其他与底层有关次重要的因素，接下来我们通过一个例子来了解。

堆排序和快速排序都是nlogn的时间复杂度，为什么实际中堆排序的效率比快速排序慢呢？这个例子可以通过前面两个基本要素进行分析。

1，对于同样的数据，在排序过程中，堆排序算法的数据交换次数要多于快速排序。

对于基于比较的排序算法来说，整个排序过程是由两个基本操作组成的，比较和交换。快速排序交换的次数不会比逆序度多。

但是堆排序的第一步是建堆，建堆的过程会打乱数据原有的相对选择顺序，导致数据有序度降低。比如对于一组已经有序的数据来说，经过建堆之后，数据反而变得更无序了。

2，其他原因：对于快速排序来说，数据是顺序访问的。而对于堆排序来说，数据是跳着访问的。比如，堆排序中，最重要的一个操作就是数据的堆化。比如下面这个例子，对堆顶进行堆化，会依次访问数组下标是1，2，4，8的元素，而不像快速排序那样，局部顺序访问，所以，这样对CPU缓存是不友好的。

逆序度：在一个排列中，如果一对数的前后位置与大小顺序相反（即前面的数大于后面的数），那么这一对数就被称为一个逆序。
2.一个排列中逆序的总数就称为这个排列的逆序度。

经典排序算法的优化

1，插入排序的优化：折半插入排序是对直接插入排序的一种改良方式，在直接插入排序中，每次向已排序序列中插入元素时，都要去寻找插入元素的合适位置，但是这个过程是从已排序序列的最后开始逐一去比较大小的，这其实很是浪费，因为每比较一次紧接着就是元素的移动。折半排序就是通过折半的方式去找到合适的位置，然后一次性进行移动，为插入的元素腾出位置

```
void BinInsertSort(){
    //插入排序：
    //默认第一个数有序，将后面的每一个数依次插入进前面的有序数列中
    for(int i=1;i<len;i++){//默认第一个数有序
        int left=0;
        int right=i-1;
        while(left<=right){
            int mid=(left+right)/2;
            if(arr[mid]<=arr[i]){//某一个要取等,避免找不到元素而死循环
                //i位置的元素应该插入到mid的右边
                left=mid+1;
            }
            if(arr[mid]>arr[i]){
                right=mid-1;
            }
        } //最终mid的位置就是应该插入的位置
        int temp=arr[i];
        for(int j=i-1;j>=left;j--){
            arr[j+1]=arr[j];
        }
        //如果插入位置不是本身所在的位置，那么才进行插入
        if(left!=i){
            arr[left]=temp;
        }
    }
}
```

快速排序的优化1--序列长度达到一定大小时，使用插入排序

当快排达到一定深度后，划分的区间很小时，再使用快排的效率不高。当待排序列的长度达到一定数值后，可以使用插入排序

当待排序列长度为5~20之间，此时使用插入排序能避免一些有害的退化情形

```
template <class T>
void QSort(T arr[],int low,int high)
{
    int pivotPos;
    if (high - low + 1 < 10)
    {
        InsertSort(arr,low,high);
        return;
    }
    if(low < high)
    {
        pivotPos = Partition(arr,low,high);
        QSort(arr,low,pivotPos-1);

QSort(arr,pivotPos+1,high);
    }
}
```

快速排序的优化2--尾递归优化

1，快排算法和大多数分治排序算法一样，都有两次递归调用。但是快排与归并排序不同，归并的递归则在函数一开始， 快排的递归在函数尾部，这就使得快排代码可以实施尾递归优化。使用尾递归优化后，可以缩减堆栈的深度，由原来的O(n)缩减为O(logn)。

2，尾递归概念：
如果一个函数中所有递归形式的调用都出现在函数的末尾，当递归调用是整个函数体中最后执行的语句且它的返回值不属于表达式的一部分时，这个递归调用就是尾递归。尾递归函数的特点是在回归过程中不用做任何操作，这个特性很重要，因为大多数现代的编译器会利用这种特点自动生成优化的代码。

3，尾递归原理：
当编译器检测到一个函数调用是尾递归的时候，它就覆盖当前的活动记录而不是在栈中去创建一个新的。编译器可以做到这点，因为递归调用是当前活跃期内最后一条待执行的语句，于是当这个调用返回时栈帧中并没有其他事情可做，因此也就没有保存栈帧的必要了。通过覆盖当前的栈帧而不是在其之上重新添加一个，这样所使用的栈空间就大大缩减了，这使得实际的运行效率会变得更高。

样例

```
 int fact(int n)              //线性递归{
    if (n < 0)
        return 0;
    else if(n == 0 || n == 1)
        return 1;
    else
        return n * fact(n - 1);
}
 int facttail(int n, int a)   //尾递归
{
    if (n < 0)
        return 0;
    else if (n == 0)
        return 1;
    else if (n == 1)
        return a;
    else
        return facttail(n - 1, n * a);
}
```

线性递归
fact(5)
{5*fact(4)}
{5*{4*fact(3)}}
{5*{4*{3*fact(2)}}}
{5*{4*{3*{2*fact(1)}}}}
{5*{4*{3*{2*1}}}}
{5*{4*{3*2}}}
{5*{4*6}}
{5*24}
120

尾递归
facttail(5,1)
facttail(4,5)
facttail(3,20)
facttail(2,60)
facttail(1,120)
120

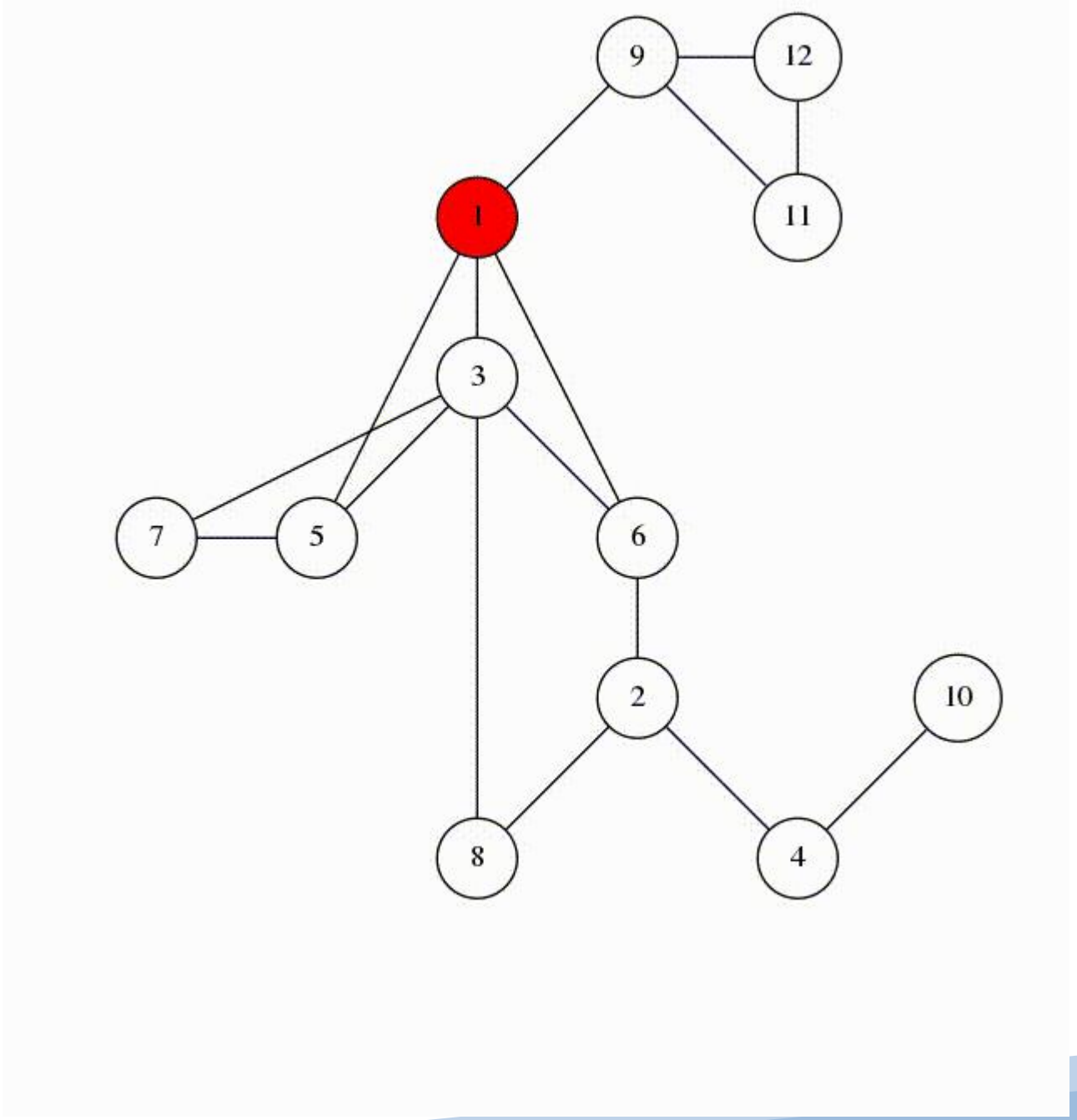
快速排序尾递归代码

```
template <class T>
void QSort(T arr[],int low,int high)
{
    int pivotPos;
    if (high - low + 1 < 10)
    {
        InsertSort(arr,low,high);
        return;
    }
    while(low < high)
    {
        pivotPos = Partition(arr,low,high);
        QSort(arr,low,pivotPos-1);
        low = pivotPos + 1;
    }
}
```

图

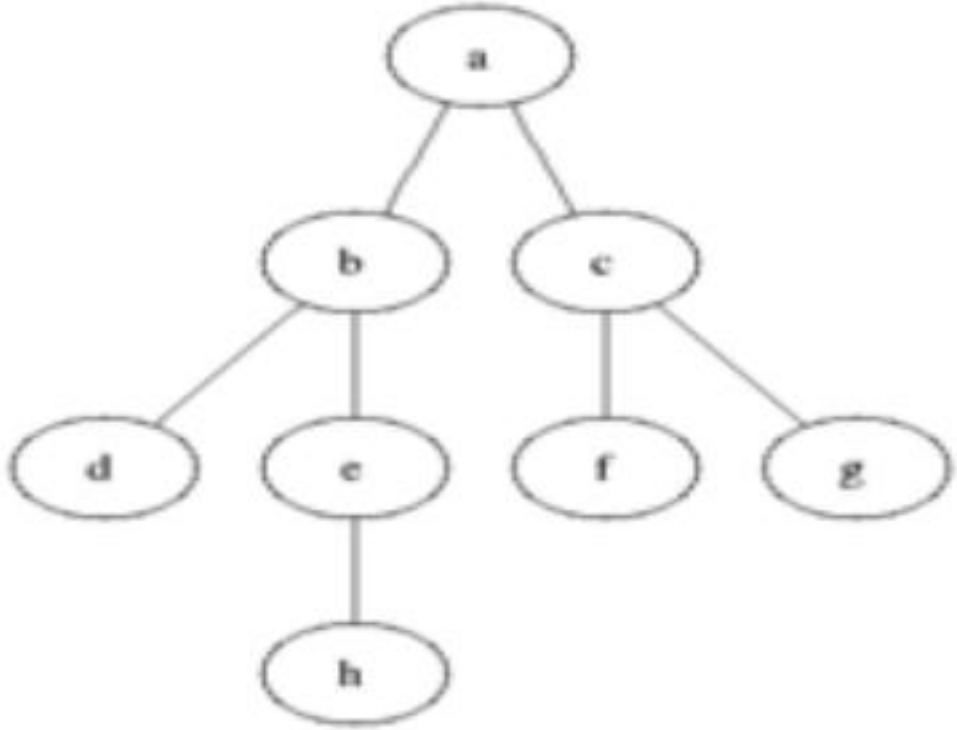
算法思想

DFS（深度优先算法）



算法思想

BFS（广度优先算法）



算法思想

图的DFS搜索

```
void DFS(int v, void (*PreVisit)(int v), void
(*PostVisit)(int v), void (*Visiting)(int v))   // Depth first
search
    {
PreVisit(v);
Visiting(v);
G->setMark(v, VISITED);
for (int w = G->first(v); w < G->n(); w = G->next(v, w))
    if (G->getMark(w) == UNVISITED)
        DFS(w, *PreVisit, *PostVisit, *Visiting);

PostVisit(v);
    }
```

算法思想

图的BFS搜索

```
void BFS(int start, void (*PreVisit)(int v), void (*PostVisit)(int v), void
(*Visiting)(int v))
    {
int v,w;
queue<int> q;
q.push(start);
PreVisit(start);
G->setMark(start, VISITED);
while (q.size()!=0)
{
    v = q.front();
    q.pop();
    Visiting(v);
    for (w = G->first(v); w < G->n(); w = G->next(v, w))
    {
        if (G->getMark(w) == UNVISITED)
        {
            G->setMark(w,VISITED);
            PreVisit(w);
            q.push(w);
        }
    }
    PostVisit(v);
}
```

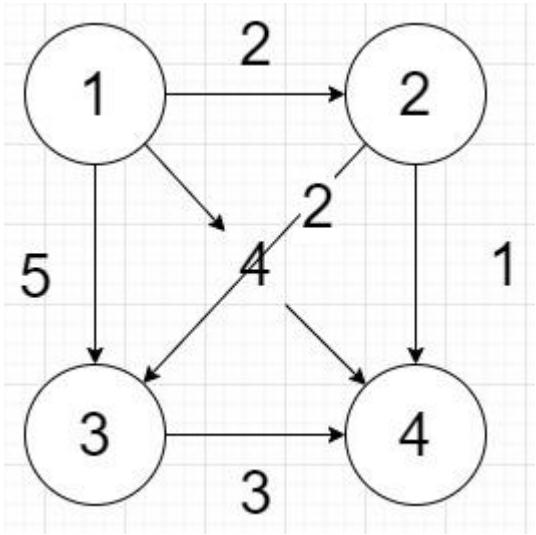
单源最短路径问题

单源最短路径

给定一个带权有向图G=（V,E），其中每条边的权是一个实数。另外，还给定V中的一个顶点，称为源。要计算从源到其他所有各顶点的最短路径长度。这里的长度就是指路上各边权之和。这个问题通常称为单源最短路径问题。

迪杰斯特拉算法思想

假设有这么一个图现在我们要求1到各个顶点的最短路径1->1间的最短路径不用想了，肯定是0
1->2之间的最短路径很明显只有一条1->2，距离为2
1->3之间，有两条路径，一条是1->3，一条是1->2->3，因为1->3之间的距离为5，而1-　　>2->3的距离为4，所以1->3的最短路径为4
1->4之间有三条路径，分别是1->2->4, 1->4, 1->3->4，而三条路线中，最短的那条线为1->2->4，距离为3
至此所有的最短路径就都求出来了



算法思想

Dijkstra算法

```
void Dijkstra1(int* D, int s)
    {
int i, v, w;
for (i = 0; i < G->n(); i++)
{
    v = minVertex(D);
    if (D[v] == INFINITY) return;
    G->setMark(v, VISITED);
    for (w = G->first(v); w < G->n(); w = G-
>next(v, w))
        if (D[w] > (D[v] + G->weight(v, w)))
            D[w] = D[v] + G->weight(v, w);
}
    }
```

```
int minVertex(int* D)   // Find min cost vertex
  {
      int i, v = -1;
      // Initialize v to some unvisited vertex
      for (i = 0; i < G->n(); i++)
          if (G->getMark(i) == UNVISITED)
          {
              v = i;
              break;
          }
      for (i++; i < G->n(); i++) // Now find smallest
D value
          if ((G->getMark(i) == UNVISITED) && (D[i]
< D[v]))
              v = i;
      return v;
  }
```

弗洛伊德算法思想

Floyd算法是另一种经典的最短路径算法，不同的是，dijkstra算法仅计算了一个起点出发的最短路径，而floyd算法可以计算全部节点到其他节点的最短路径。相比之下，Floyd算法复杂度为n3，而dijkstra算法为n2。Floyd算法的基本思想也是松弛。这是一个动态规划的经典例子，在求解各个点到其他点的最短路径的过程中往往会有很多的重叠问题，通过一个表D[][]将这些问题保存下来，节省了重复的计算。

这个算法看起来非常简洁。Floyd算法第k次迭代后得到的中间结果的实际含义是：在允许路径经过从第0个到第k个点的条件下，任意两点间的最短路径长度。也就是说当k=0时，d的值和邻接表恰好是一样的，因此初始化时将邻接表复制给d即可。floyd算法需要一个p矩阵保存路径信息，p[i][j]=x的意思是，从i到j的最短路径，先走(i->x)，再走(x->j)，至于这两段路分别怎么走，那就再查p[i][x]和p[x][j]。
Floyd算法的基本步骤是：

1. 初始化。将邻接表复制给d，同时给p赋初值p

2. 计算允许经过前k个节点的条件下任意两点的最短路径：依次考虑第i个出发的点到第j个点的当前路径，和经过了k这个点的路径长度进行比较，取一个小的，并记录路径数据。

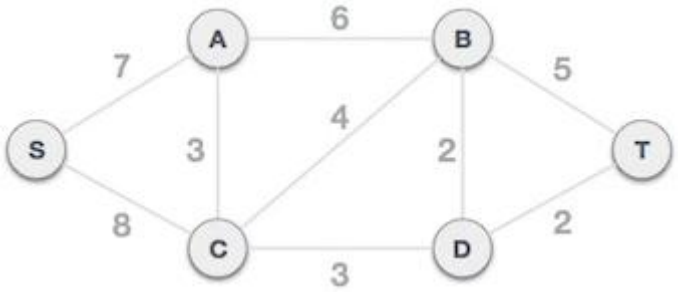
Floyd算法代码

```
void floyd(int g[][], int d[][], int p[][], int n)
{
        int i, j, k;
        // 初始化
        for (i = 0; i < n; ++i)
        {
                for (j = 0; j < n; ++j)
                {
                        d[i][j] = g[i][j];
                        p[i][j] = j;
                }
        }
        for (k = 0; k < n; ++k)
        {
                for (i = 0; i < n; ++i)
                {
                        for (j = 0; j < n; ++j)
                        {
                                if (d[i][j] > d[i][j] + d[k][j])
                                {
                                        d[i][j] = d[i][k] + d[k][j];
                                        p[i][j] = k;
                                }
                        }
                }
        }
}
```

最小生成树问题

一个有 n 个结点的连通图的生成树是原图的极小连通子图，且包含原图中的所有 n 个结点，并且有保持图连通的最少的边。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。

克鲁斯卡尔算法

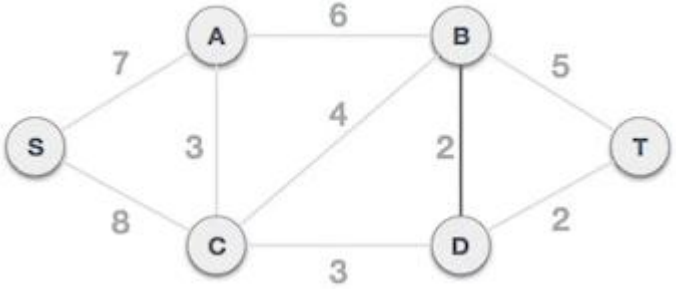


举个例子，图 1 是一个连通网，克鲁斯卡尔算法查找图 1 对应的最小生成树，需要经历以下几个步骤：

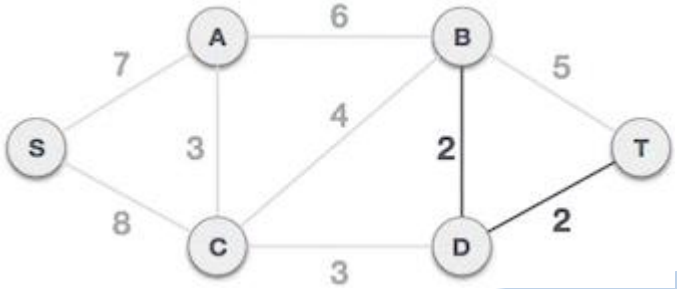
1) 将连通网中的所有边按照权值大小做升序排序：

B, D	D, T	A, C	C, D	C, B	B, T	A, B	S, A	S, C
2	2	3	3	4	5	6	7	8

2) 从 B-D 边开始挑选，由于尚未选择任何边组成最小生成树，且 B-D 自身不会构成环路，所以 B-D 边可以组成最小生成树。

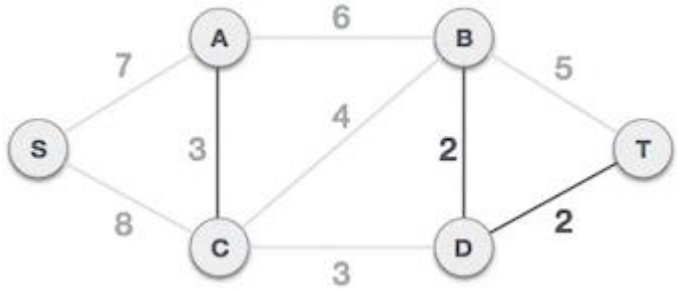


3) D-T 边不会和已选 B-D 边构成环路，可以组成最小生成树：

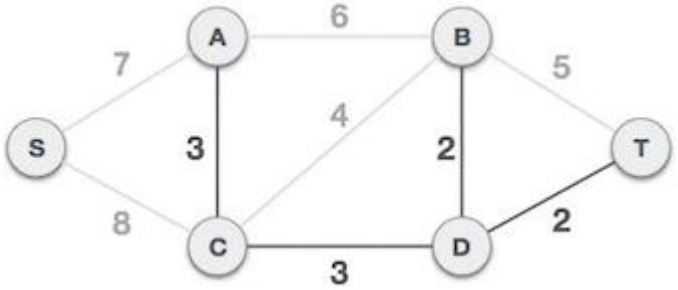


克鲁斯卡尔算法

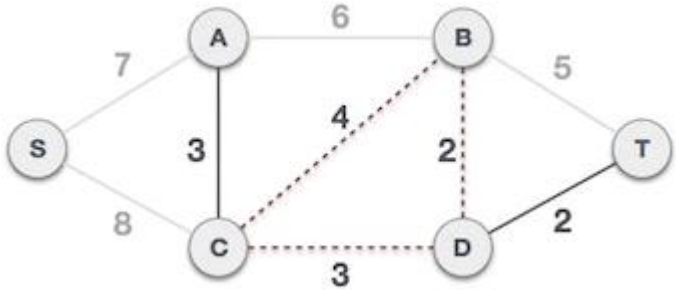
4) A-C 边不会和已选 B-D、D-T 边构成环路，可以组成最小生成树



5) C-D 边不会和已选 A-C、B-D、D-T 边构成环路，可以组成最小生成树：

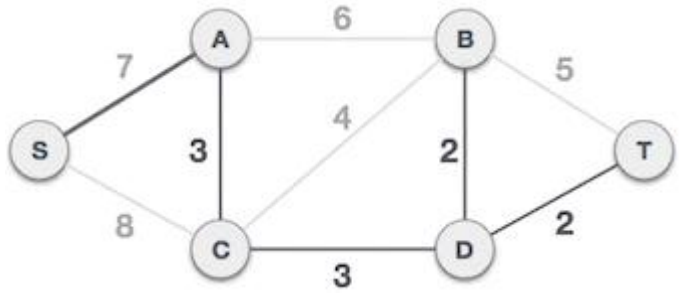


6) C-B 边会和已选 C-D、B-D 边构成环路，因此不能组成最小生成树：

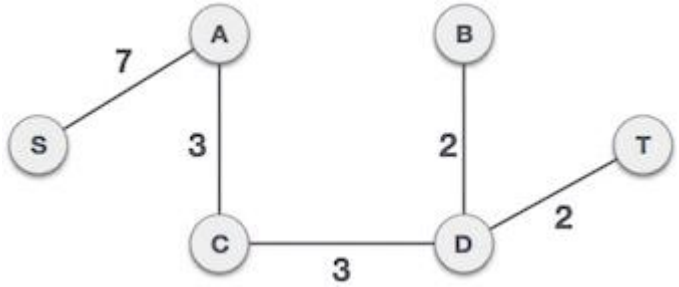


克鲁斯卡尔算法

7) B-T 、A-B、S-A 三条边都会和已选 A-C、C-D、B-D、D-T 构成环路，都不能组成最小生成树。而 S-A 不会和已选边构成环路，可以组成最小生成树。



8) 如图 7 所示，对于一个包含 6 个顶点的连通网，我们已经选择了 5 条边，这些边组成的生成树就是最小生成树。



克鲁斯卡尔算法

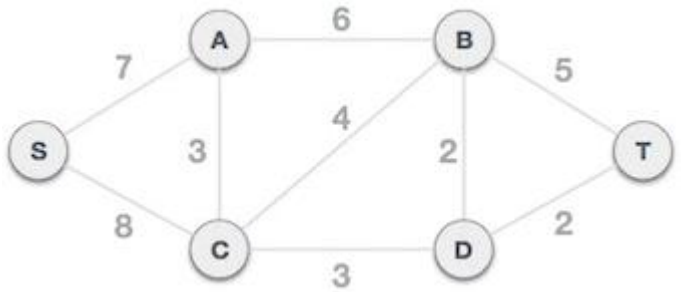
基本思想：按照权值从小到大的顺序选择 n-1 条边，并保证这 n-1 条边不构成回路
具体做法：首先构造一个只含 n 个顶点的森林，然后依权值从小到大从连通网中选择边加入到森林中，并使森林中不产生回路，直至森林变成一棵树为止

```
int Find(int *parent, int f) {
    while (parent[f]>0) {
        f = parent[f];
    }
    return f;
}

//最小生成树，克鲁斯卡尔算法
void Kruskal(MGraph *G) {

    int parent[MAXVERTEX];//存放最小生成树的顶点
    for (int i = 0; i < G->numvertex; i++) {
        parent[i] = 0;
    }
    int m, n;
    for (int i = 0; i < G->numedges; i++) {
        n = Find(parent, G->edges[i].begin);
        m = Find(parent, G->edges[i].end);
        if (n != m) { //m=n说明有环
            parent[n] = m;
            printf("(%d,%d) %d\t", G->edges[i].begin, G->edges[i].end, G->edges[i].wight);//打印边和权值
        }
    }
}
```

Prim算法

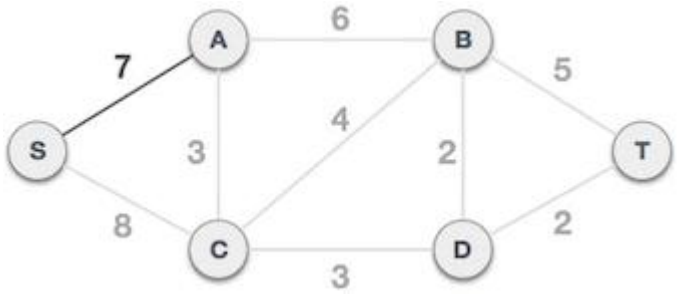
1.普里姆算法的实现思路是：将连通网中的所有顶点分为两类（假设为 A 类和 B 类）。初始状态下，所有顶点位于 B 类；
2.选择任意一个顶点，将其从 B 类移动到 A 类；
3.从 B 类的所有顶点出发，找出一条连接着 A 类中的某个顶点且权值最小的边，将此边连接着的 A 类中的顶点移动到 B 类；
4.重复执行第 3 步，直至 B 类中的所有顶点全部移动到 A 类，恰好可以找到 N-1 条边。

下图是一个连通网，使用普里姆算法查找最小生成树，需经历以下几个过程：



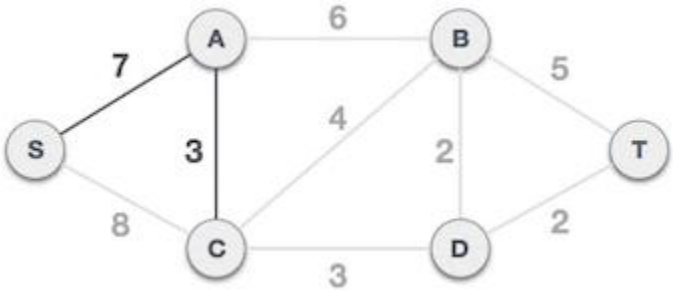
Prim算法

1) 将图中的所有顶点分为 A 类和 B 类，初始状态下，A = {}，B = {A, B, C, D, S, T}。



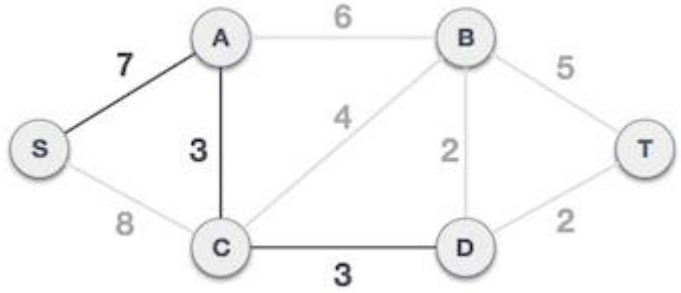
2) 从 B 类中任选一个顶点，假设选择 S 顶点，将其从 B 类移到 A 类，A = {S}，B = {A, B, C, D, T}。从 A 类的 S 顶点出发，到达 B 类中顶点的边有 2 个，分别是 S-A 和 S-C，其中 S-A 边的权值最小，所以选择 S-A 边组成最小生成树，将 A 顶点从 B 类移到 A 类，A = {S, A}，B = {B, C, D, T}。

3) 从 A 类中的 S、A 顶点出发，到达 B 类中顶点的边有 3 个，分别是 S-C、A-C、A-B，其中 A-C 的权值最小，所以选择 A-C 组成最小生成树，将顶点 C 从 B 类移到 A 类，A = {S, A, C}，B = {B, D, T}。

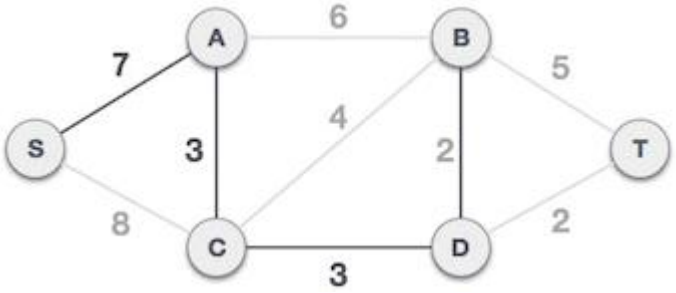


Prim算法

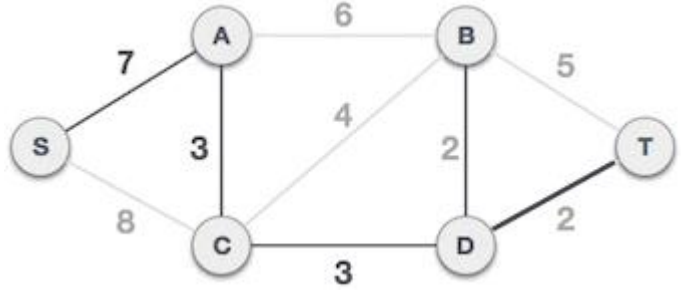
4) 从 A 类中的 S、A、C 顶点出发，到达 B 类顶点的边有 S-C、A-B、C-B、C-D，其中 C-D 边的权值最小，所以选择 C-D 组成最小生成树，将顶点 D 从 B 类移到 A 类，A = {S, A, C, D}，B = {B, T}。



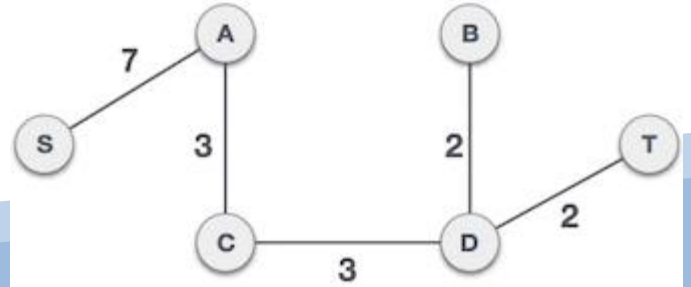
5) 从 A 类中的 S、A、C、D 顶点出发，到达 B 类顶点的边有 A-B、C-B、D-B、D-T，其中 D-B 和 D-T 的权值最小，任选其中的一个，例如选择 D-B 组成最小生成树，将顶点 B 从 B 类移到 A 类，A = {S, A, C, D, B}，B = {T}。



6) 从 A 类中的 S、A、C、D、B 顶点出发，到达 B 类顶点的边有 B-T、D-T，其中 D-T 的权值最小，选择 D-T 组成最小生成树，将顶点 T 从 B 类移到 A 类，A = {S, A, C, D, B, T}，B = {}。



7) 由于 B 类中的顶点全部移到了 A 类，因此 S-A、A-C、C-D、D-B、D-T 组成的是一个生成树，而且是一个最小生成树，它的总权值为 17。



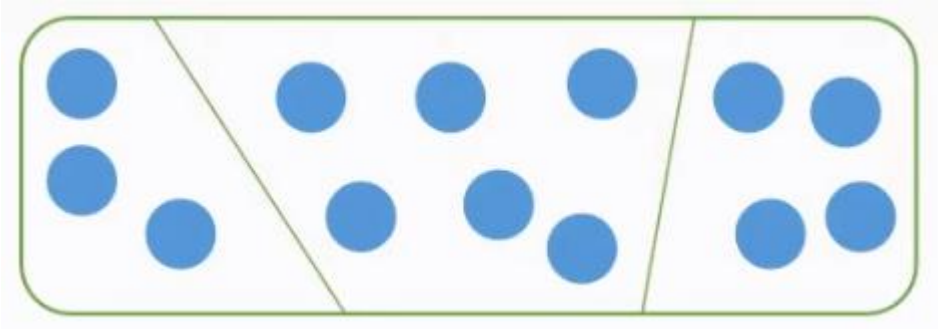
Prim算法

```
void Prim(int* D, int s)   // Prim's MST algorithm
  {
     int v[G->n()];
  for(int i=0;i<G->n();i++)
     {
        G->setMark(i,UNVISITED);
        D[i]=G->weight(s,i)?G->weight(s,i):INFINITY;
        v[i]=G->weight(s,i)?s:-1;
     }
     D[s]=0;G->setMark(s,VISITED);
     for(int i=0;i<G->n();i++)
     {
        int p=-1;int min=INFINITY;
        for(int j=0;j<G->n();j++)
        {
           if(G->getMark(j)==UNVISITED&&min>D[j])
           {
              p=j;
              min=D[j];
           }
        }
if(p==-1) break;
           AddEdgetoMST(v[p],p);
           G->setMark(p,VISITED);
           for(int j=0;j<G->n();j++)
           {
              if(G->weight(p,j)&&G->weight(p,j)<D[j])
              {
                 v[j]=p;
                D[j]= G->weight(p,j);
              }
           }
     }
```

树

并查集树

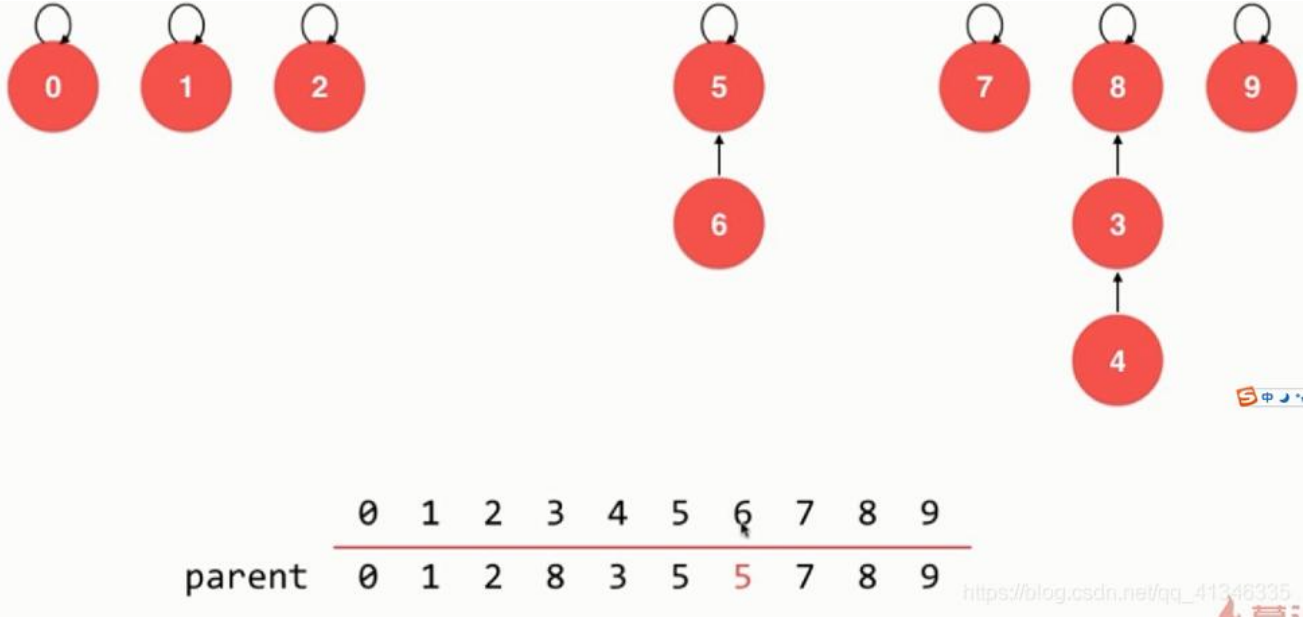
并查集：顾名思义，集是集合的意思，多个相同属性的元素组合在一起叫做集合，一般在最初情况下，各个元素是独自存在的，唯一的联系就是他们相同的属性，并，就是把相同属性的元素合并起来，查就是查询某两个元素是否是属于同一集合



左图为3个集合，每个集合里面的元素都具有一定的共同属性

并查集树的物理结构

这种元素之间的关系可以用一种树来描述：并查集树
那么怎样构建能表示这种关系的树呢。
就像社会帮派国家一样，每个集体都有自己的领导人，所以从某个角度来看，区分两个人是否属于一个集体只要判断他们的领导人是否相同即可，所以这个并查集树可以用数组描述。每个位置存储的值为他的最高领导人。
就像这样



为什么8的位置是8呢，因为8是最高领导人，我们规定自己是自己的最高领导人。

并查集树的代码实现

通过递归实现查找某元素的最高领导人 /查找祖宗

```
int getGrand(int a[],int x)
{
        if(a[x]==x)
                return x;
        else
                a[x]=getGrand(a,a[x]);//父节点设为根节点
                return a[x];
} //这里实现了路径压缩，某个人的直属领导人就是最高领导人，增加了查找效率

//合并
void merge(int a[],int t1,int t2)
{
        int x1=getGrand(a,t1);
        int x2=getGrand(a,t2);
        //判断是否是同一个祖先
        if(x1!=x2)
                //靠左原则(非常重要)左边的人是领导人
        a[x2]=x1; //领导人确定领导关系，所以用x1,x2

}
```

并查集树的优化

上述merge中，使用了靠左原则进行合并，但是如果帮派人数少的那一边的领导人成为了两个帮派的最高领导人，一是会引起不服，而是会增加查找领导人的时间。
优化解决：让帮派人数多的一方的领导人成为两个帮派的最高领导人

```
int numofX(int x){
        int res = 0;
        int leader = find(x);
        for(int i = 0;i < numOfAll; i++){if(a[i] == leader){res++;}
        return res;
}
void merge(int x,int y){
        //首先判断规模
        int nx = numofX(x);
        int ny = numofX(y);
        if(nx > ny){a[find(y)] = find(x);}
        else if(ny > nx){a[find(x)] = find(y);}
```

按秩合并

并查集树的应用
--Kruskal最小生成树算法

Kruskal算法先按照边权从小到达排序，由最小边权值出发，如果在图中加上这两条边不形成环，则这两条边在最小生成树中存在，如果形成环则不添加。

重点在成环问题上，难道用dfs搜索吗，dfs虽然很强，但是这里并不适用，成环不就是两个点属于一个集合吗，此时使用并查集树即可解决

知道思路之后就用伪代码描述这个思路吧。这个程序不能运行呢。

```
int MinimumSpanningTree(Verge verge[],int numofverge){
    /*
    Verge结构体：
    Elemtype in,out;//端点
    int weight;//权值
    */
    Sort(verge,verge+numofverge,cmp);//按照weight从小到大排序
    /*
    并查集树:int a[];
    int getGrand(int a[],int x){
        if(a[x]==x)
            return x;
        else
            a[x]=getGrand(a,a[x]);
        return a[x];
    }
    void merge(int t1,int t2){
        int x1=getGrand(a,t1);
        int x2=getGrand(a,t2);
        if(x1!=x2)
        a[x2]=x1;
    }
    */
    int weight=0;
    for(int v=0;v<numofverge;v++){
        if(getGrand(verge[v].in)!=getGrand(verge[v].out)){
            merge(verge[v].in,verge[v].out);
            weight+=verge[v].weight;
        }
    }
    return weight;
}
```

返回最小生成树的权值

THANKS!