线性数据结构

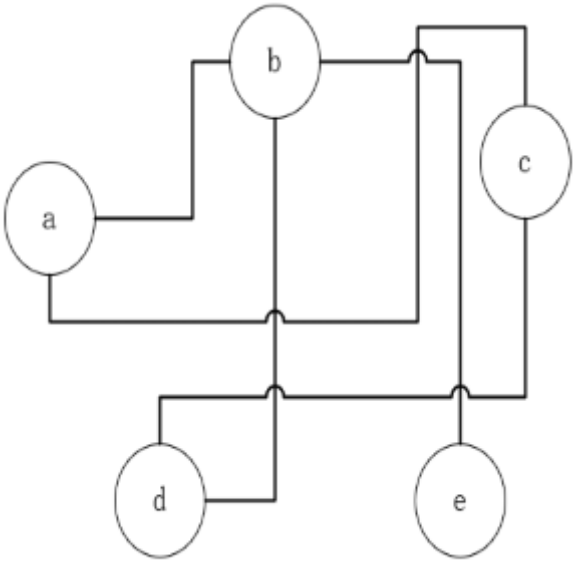
理论：

这一部分暂时**不需要思考怎么编程实现**，只需要理解某个东西是什么，是个什么原理，有个概念就ok了。

数据结构：

存储两种数据信息（元素+元素关系）。当你面对题目所给的很多个数据元素时，你要怎么存储它们的值和它们之间的关系。

比如，有n个村落，编号为a,b,c,d,e，各个村落间存在公路相连。



常见的数据结构有线性表，栈，队列，树，图等。

数据结构是一种思想类东西（类似一种思路），在编程时可以用不同存储结构实现。

比如：只考虑前后关系的元素，可以用静态数组，也可以用链表。

算法题=数据结构+算法，

线性数据结构：

具有单一前驱和单一后继的数据结构。常见的有静态数组，链表，栈，队列等。

线性表：

包括顺序表和链表

顺序表：

数据元素存储空间（元素）连续，下标大小代表数据元素的前后关系（元素关系）。

5	7	6	1	2
0	1	2	3	4

一般用静态数组实现

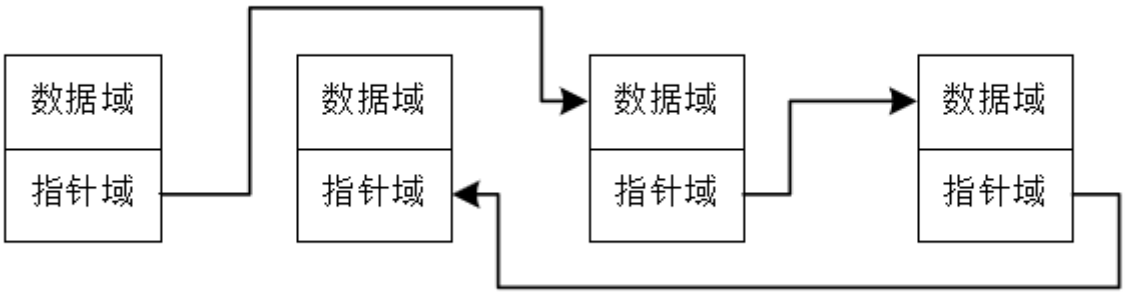
静态数组和动态数组：

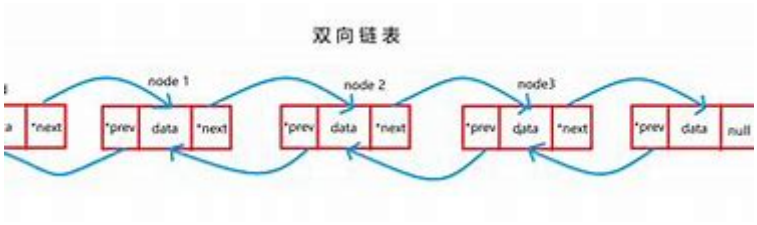
简单区分，没有涉及指针便是静态数组，涉及指针便是动态数组

链表：

数据元素的存储空间不一定连续，数据元素之间的关系由指针域确定。数据域存储元素，指针域存储元素关系。

常见的还有单链表，双链表等。

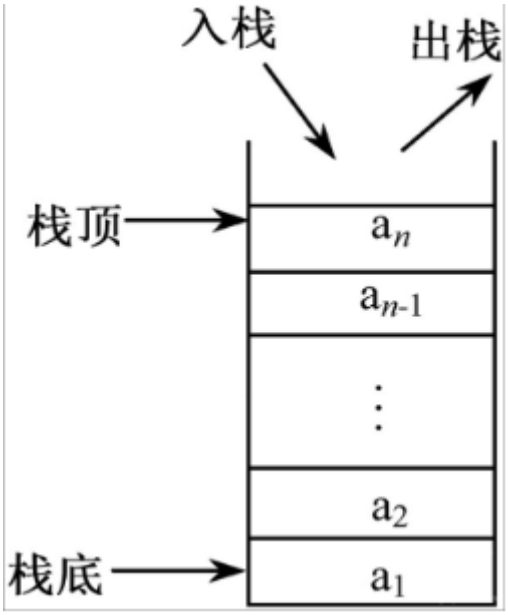




一般用动态数组实现（即使用指针）。

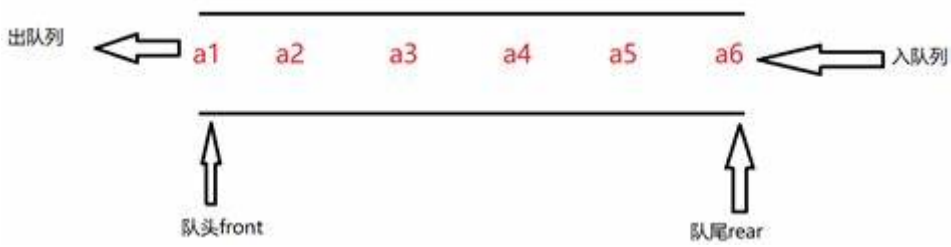
栈：

特殊之处在于入栈（存储元素）和出栈（访问元素）均必须在栈顶执行。



队列：

特殊之处在于入队（存储元素）必须在队尾执行，出队（访问元素）必须在队头执行。



实操：

这一部分只讲解算法竞赛中实用的。

上述很多数据结构在实际竞赛中我们一般使用STL中现成的容器。

线性表：

//线性表是一种思想类的东西（思路），下面讲线性表常用的存储结构

静态数组：

```
int arr[];
vecter<int> vr;
```

链表：

```
list<int> lt;
deque<int> de;//更常用
```

不用学内部怎么实现，只需要会用STL容器和对应的方法即可。

静态双链表：

比list好用的原因：在特定的题目环境下，可以以O(1)的复杂度实现查找，插入，删除。

双链表的静态数组实现，结构体+数组。

```
//声明部分
const int N=100000;
struct node//定义双链表的结点结构体
{
    int data;//数据域，存放元素值
    int be;//该结点前驱结点，此处为前驱结点在数组中的下标
    int ar;//该结点后继结点，此处为后继结点在数组中的下标
};
int cur;//记录目前数组中第一个空位置
node delist[N];//双链表的静态数组

//查找部分,查找数组中下标为k的元素
delist[k];

//初始化函数，设置空双链表，O(1)
void init()
{
    delist[0].ar=1;//delist[0]--头结点，不存储元素，仅作为双链表链头标志
    delist[1].be=0;//delist[1]--尾结点，不存储元素，仅作为双链表链尾标志
    cur=2;//后续插入元素是直接插入在最后面，cur为第一个空的位置
}

//插入函数，将元素x插入在数组下标为index后面，O(1)
void insert(int index,int x)
{
    //为新插入结点赋值
    delist[cur].data=x;//值域赋值
    delist[cur].be=index;//前驱赋值
    delist[cur].ar=delist[index].ar;//后继赋值
    //处理插入后元素x的前驱和后继
    int nar=delist[index].ar;//记录此时元素x的后继
    delist[nar].be=cur;//更新此时元素x的后继节点的前驱关系
    delist[index].ar=cur;//更新此时元素x的前继节点的后继关系
    cur++;//更新目前第一个空的位置。
}

//删除函数，删除数组下标为index的元素，O(1)
void remove(int index)
{
    int nbe=delist[index].be;//记录被删除节点的前驱节点下标
    int nar=delist[index].ar;//记录被删除节点的后驱节点下标
    delist[nbe].ar=nar;//更新被删除结点的前驱节点的后继关系
    delist[nar].be=nbe;//更新被删除结点的后驱节点的前驱关系
}
```

栈：

常规：

```
//一般情况下使用STL里的stack
stack<int> s;
```

单调栈：

作用：

找出从左/右遍历第一个比它小/大的元素位置。

例题：

给定一个整型数组，求出数组中每个元素的左边第一个比它小的数组元素下标，若不存在，则置为-1。

```
//声明部分
#include<bits/stdc++.h>
using namespace std;
int n;//整型数组元素个数
int arr[1000];//存储输入整型数据元素
int ans[1000];//存储最终答案
void getans();//答案将通过这个函数求得
int main()
{
    cin>>n;
    for(int i=1;i<=n;i++)
        cin>>arr[i];
    getans();
    for(int i=1;i<=n;i++)
        cout<<ans[i]<<' ';
    return 0;
}
```

1.暴力法 -- O(n^2) --大概率TLE（超时）

```
void getans()
{
    for(int i=1;i<=n;i++)
    {
        int j;
        for(j=i-1;j>=1;j--)
        {
            if(arr[j]<arr[i])
            {
                ans[i]=j;
                break;
            }
        }
        if(j==0) ans[i]=-1;
    }
}
```

2.单调栈 -- O(n)

```
void getans()
{
    stack<int> s;//声明栈
    for(int i=1;i<=n;i++)//遍历整型数组
```

```
    {
        //思考两种情况：
        //1.如果上一个元素（栈中）比目前元素小，上一个元素就是答案
        //2.如果上一个元素（栈中）比目前元素大，那么只能往前找，并且边往前边弹出，因为弹出的
        //  那些已经不可能作为答案了（为什么这样后面解释），直到找到左边第一个比目前元素小的，
        //  将它作为答案。
        while(!s.empty()&&arr[i]<=arr[s.top()])//第一种情况，往前找，找的过程注意栈空的情况
            s.pop();
        if(s.empty()) ans[i]=-1;//如果栈空，说明没找到比目前元素小的，赋-1
        else ans[i]=s.top();//找到了，那么将其作为答案
        s.push(i);//同时将目前元素下标入栈，因为目前元素下标有可能作为后续的答案
    }
}
//解释为什么可以边往前边弹出
//思考两种情况：
//1.如果下一个元素（相对于目前元素）比目前元素大，那么目前元素便是下一个元素的答案，不会是弹出的元素
//2.如果下一个元素（相对于目前元素）比目前元素小，那么之前弹出的那些因为比目前元素大，那么肯定也比下一个元素大，
//  所以更不可能作为下一个元素的答案了。
```

队列：

常规：

```
queue<int> q;
priority_queue<int> q;
```

单调队列：

作用：

一般用来在一个动态小区间中寻找极值。

特点：

1. 队列中元素间具有单调性
2. 队首和队尾都可以进行出队操作，只有队尾可以进行入队操作，本质是双向队列deque实现。
3. 最后单调队列队头不是最大元素，就是最小元素。

例题：

给定一个整数数组 nums，有一个大小为k的滑动窗口从数组的最左侧移动到数组最右侧，滑动窗口每次只向右移动一位，返回每次滑动窗口的最大值。

1. 暴力法--O（nk）,可能TLE
2. 单调队列--O（n），单调递减队列

```
#include<bits/stdc++.h>
using namespace std;
const int N=100;
int nums[N],que[N];
//nums用于存储给定的序列值
//que用于模拟单调队列，存储的是下标，非序列值
```

```
int main()
{
    ios::sync_with_stdio(0);
    cin.tie(0); cout.tie(0);
    int n,k; cin>>n>>k;
    //n--序列数，k--滑动窗口长度
    for(int i=1;i<=n;i++)
        cin>>nums[i];
    //输入序列数
    int front,rear;
    //front为que队列队头元素下标
    //rear为que队列队尾元素下标
    front=rear=1;
    //初始时都置为1
    //当队列非空时，front<=rear
    //当队列为空时，front>rear
    que[front]=1;
    //首先将序列的第一个元素下标入队
    if(k==1) cout<<nums[que[front]]<<'\n';
    //防止当k=1时，第一次窗口没有输出
    for(int i=2;i<=n;i++)
    {
        //入队时思考两种情况：
        //1. 当滑动时，有些元素已经不在窗口内了
        //2. 当滑动时，边滑动边去掉无效元素，后面解释为什么可行

        //第一种情况的处理，若出了窗口范围的元素，出队
        //经过此操作，保证队列中的元素均是窗口中的元素
        if(i-que[front]==k)
            front++;
        //第二种情况的处理，若队列中出现比目前要插入的元素小的元素，弹出
        //思考两种情况：
        //1. 如果目前插入的元素与队列中的元素仍能够共存在同一个窗口,那么队列中比插入元素小的元素一定不可能是该次窗口的最大值
        //2. 那有没有可能是下一次窗口的最大值呢，也不可能，原因是在之后的操作中如果队列中比插入元素小的元素(在这次操作中)
        //    存在在窗口中，那么插入的这个元素也必存在在此操作中，那么最大元素也不可能是这些元素。
        //因此，在每次操作中，队列中比插入元素小的元素都不可能是最大值了，所以可以弹出。
        while(front<=rear&&nums[i]>nums[que[rear]])
            rear--;
        que[++rear]=i;//将目前元素插入
        if(i>=k) cout<<nums[que[front]]<<'\n';
        //如果到达滑动窗口最大长度，输出答案
    }
    return 0;
}
```

散列（哈希）表：

背景：

在大多数据结构中，查找某个值的位置，均得通过一个一个比较查找值与元素，直到相等，才算找到查找值的元素，难以用O(1)复杂度实现查找（二分也得log(n)）,哈希表实现以O(1)复杂度查找

思路：

在存储一组值时，将每个值通过一个函数计算出一个地址，并将对应值存储到对应地址上。这样，在给定一个待查找值时，便可以通过这个函数直接计算出对应地址。

eg. 给定一个整型序列，对于数组中每个整数a_i，通过一个函数计算出一个整数 j，将a_i存储在数组中下标为 j 的位置中。

其中，这个函数叫散列函数，这个存储空间叫散列表，存储位置叫散列地址，存储的值叫关键码。

问题：

1. 如何确定散列函数
2. 如果不同值通过散列函数得到相同散列地址，怎么处理（这种情况即冲突）

散列函数：

要求：

1. 简单，复杂度不能太高
2. 使关键码的存储位置均匀

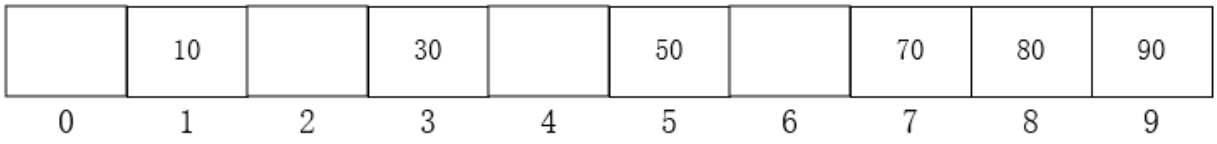
常见的有三种方法：

1. 直接定址法
2. 除留余数法
3. 平方取中法

直接定址法：

散列函数为线性函数：H(key)=a · key+b (a,b为常数)

eg. 关键码集合{10,30,50,70,80,90}，散列函数为H(key)=key/10，存储结构如下：



适用于事先知道关键码的分布，且关键码集合不是很大，连续性较好的情况

除留余数法：

散列函数为取余：H(key)=key mod p

一般情况下，若散列表表长为m，通常p为小于或等于表长（接近m）的最大素数或不包含小于20质因子的合数

eg. 关键码集合{10,30,50,70,82,90}，散列函数为H(key)=key mod 7，存储结构如下：

70	50	30	10		82	90	
0	1	2	3	4	5	6	7

适用于事先不知道关键码分布

平方取中法：

散列函数为将关键码平方取中间几位

eg. $(1234)^2$=1522756，可取22或27

适用于事先不知道关键码的分布且关键码的位数不是很大的情况

处理冲突：

在大多数情况下，寻找一个完全没有冲突的散列函数很难，因此，存在冲突是常态，那么，如何解决？

常用的方法有：

1. 开放地址法
2. 拉链法

开放地址法：

当遇到冲突时，寻找下一个空的散列地址：$(H(key)+d_i)\%m$

其中，H(key)为关键码的散列地址（已有元素），m为散列表长度。

根据d_i不同可以分为线性探测法和二次探测法

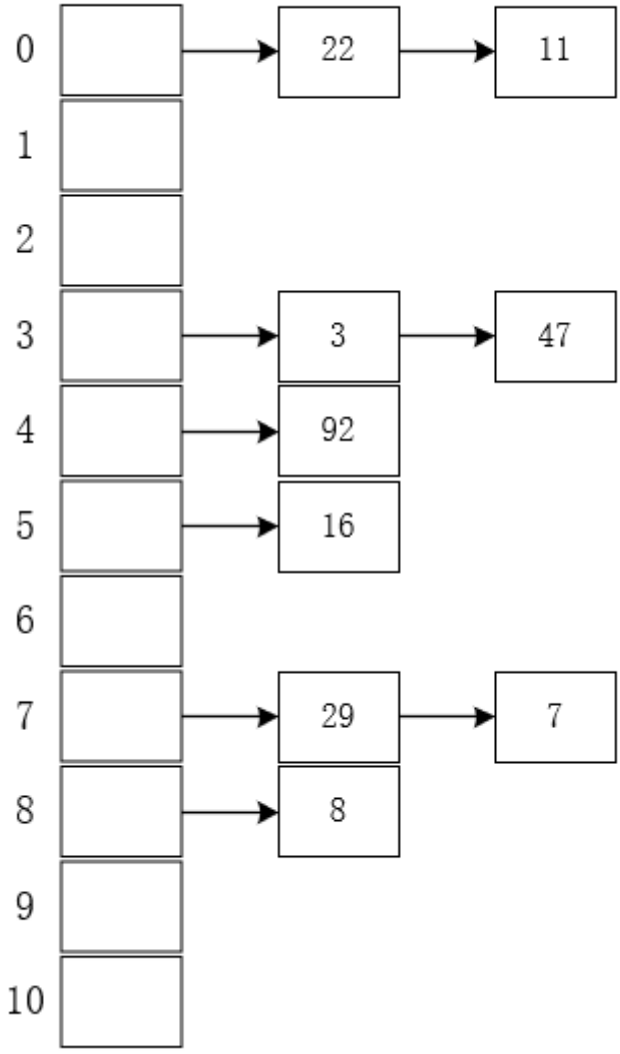
线性探测法：$d_i=1,2,3...m-1$

二次探测法：$d_i=1^2, -1^2, 2^2, -2^2 ... q^2, -q^2, q \leq \sqrt{m}$

拉链法：

当遇到冲突时，用链表存储散列后相同的关键码

eg. 关键码集合{47,7,29,11,16,92,22,8,3}，散列函数为H(key)=key mod 11，存储结构如下图：



手写哈希表：

手写STL中的map，代码如下：

```
const int sz=100;//元素最大个数
struct hash_map//手写
{
    struct data
    {
        long long u;//key
        int v;//value
        int nex;//冲突时指向下一个
    };
    data ve[sz];
    int nn;//目前元素个数
    int h[sz];
    int hash(long long x)
    {
        return x%(sz-3);//哈希函数
    }
    int& operator [] (long long x)//查找操作
    {
        int index=hash(x);
        //查找成功
        for(int i=h[index];i;i=ve[i].nex)
```

```
            if(ve[i].u==x) return ve[i].v;
        //查找失败，添加
        ve[++nn]={x,-1,h[index]};
        h[index]=nn;
        return ve[nn].v;
    }
    hash_map()//初始化
    {
        nn=0;
        memset(h,0,sizeof(h));
    }
};
```