数据结构-1

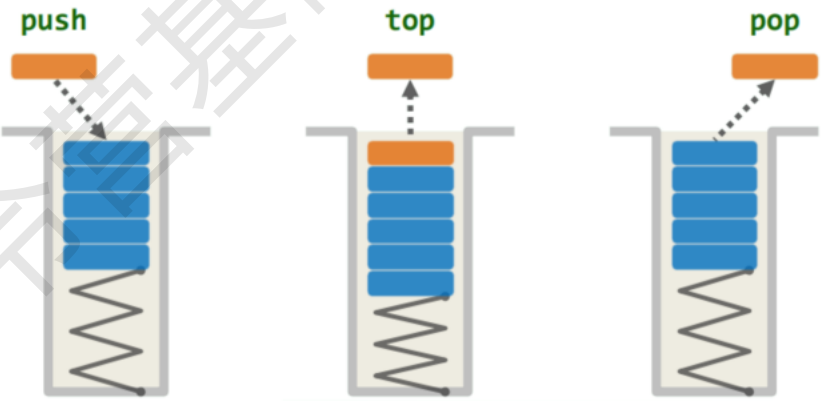
王文铎

栈

LIFO(Last in first out)表

只能在栈顶插入和删除

栈底为盲端



STL中的栈

```
#include <stack>        //头文件
stack <T> s;            //定义元素类型为T的栈
s.push(a);              //将a入栈
s.top();                //返回栈顶元素
s.pop();                //将栈顶出栈（需要保证栈非空）
s.empty();              //返回栈是否为空
s.size();               //返回栈内元素个数
```

实例

操作	输出	栈（左侧栈顶）
Stack()		
empty()	true	
push(5)		5
push(3)		3 5
pop()	3	5
push(7)		7 5
push(3)		3 7 5
top()	3	3 7 5
empty()	false	3 7 5

操作	输出	栈（左侧栈顶）
push(11)		11 3 7 5
size()	4	11 3 7 5
push(6)		6 11 3 7 5
empty()	false	6 11 3 7 5
push(7)		7 6 11 3 7 5
pop()	7	6 11 3 7 5
pop()	6	11 3 7 5
top()	11	11 3 7 5
size()	4	11 3 7 5

括号匹配

给出一个括号序列S，判断它是否是匹配的。

例子：

(((()())()))是匹配的

())(是不匹配的

括号匹配

一对相邻的括号它们之间一定是互相匹配的，那么原来是匹配的当且仅当把它们去掉之后也是匹配的。

如果找不到相邻的括号，要么已经是空串，要么是))(((形式。

如何每次找到相邻的括号?

顺序扫描表达式，用栈记录扫描过的部分，遇到(则进栈，遇到)则出栈，如果此时栈为空则为不匹配的。

扫描完如果栈非空则为不匹配的（有多余左括号）。

实现

```
bool paren( const char exp[], int lo, int hi ) { //exp[lo, hi)
   Stack<char> S; //使用栈记录已发现但尚未匹配的左括号
   for ( int i = lo; i < hi; i++ ) //逐一检查当前字符
      if ( '(' == exp[i] ) S.push( exp[i] ); //遇左括号：则进栈
      else if ( ! S.empty() ) S.pop(); //遇右括号：若栈非空，则弹出左括号
      else return false; //否则（遇右括号时栈已空），必不匹配
   return S.empty(); //最终栈空，当且仅当匹配
}
```

一些扩展

如果不止一种括号？例如：{[]({}[])}

遇到右括号时需要判断栈顶是否为对应的左括号，如果不是则为不匹配。

如果只有一种括号，实际上我们不需要知道栈中元素是什么。

只需要一个计数器，遇到左括号+1，遇到右括号-1。

后缀表达式

又称逆波兰表达式，不使用括号即可表示优先级，运算符放置于参与运算的数之后。

作为补偿，需要额外引入一个起分割作用的元字符（比如空格或者.）

例如：

0!+123+4*(5*6!+7!/8)/9

0 ! 123 + 4 5 6 ! * 7 ! 8 / + * 9 / +

后缀表达式

引入栈S。

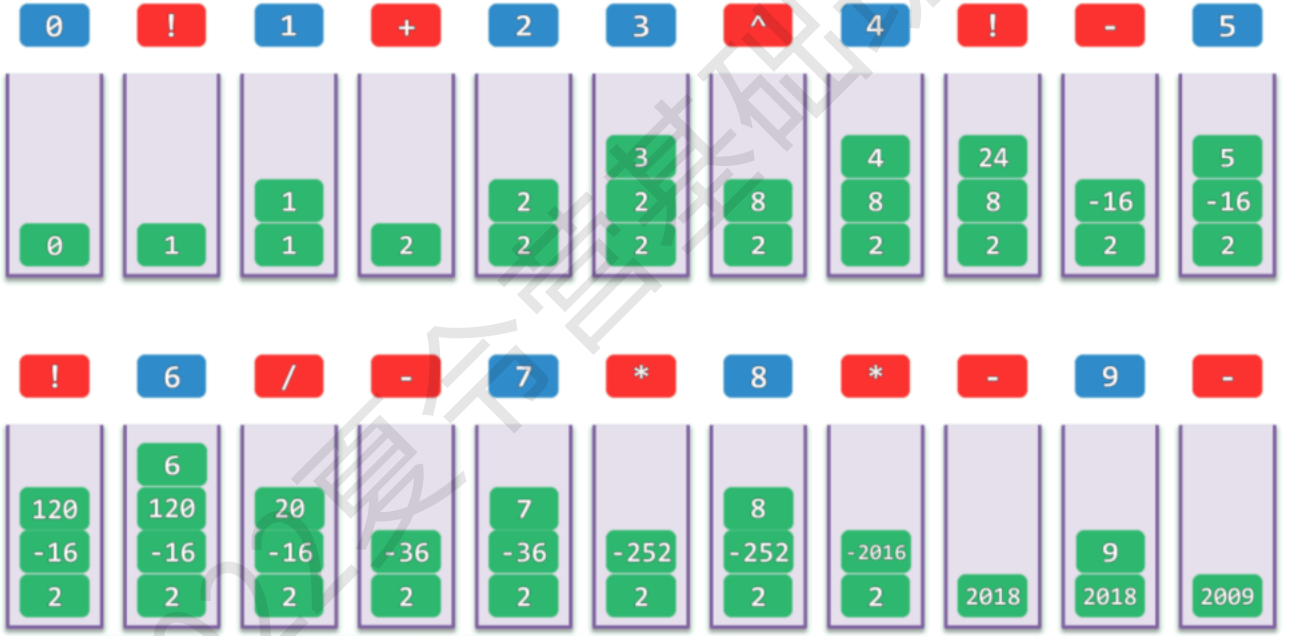
逐个处理每个元素。

如果是操作数，加入栈中。

如果是操作符，从栈顶取出对应个数的数，将运算后结果加入栈中。

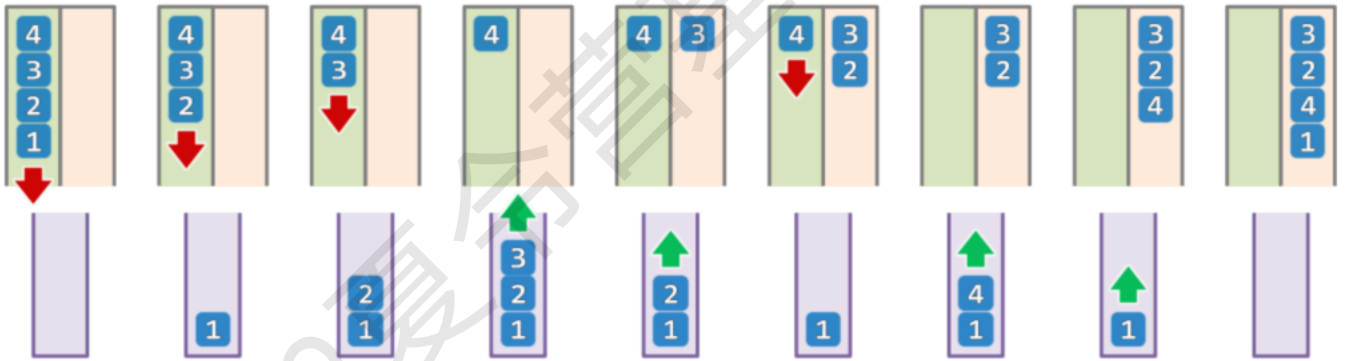
最后栈中只会剩下一个数就是答案。

0!1+23^4!-5!6/-7*8*-9-


0 ! 1 + 2 3 ^ 4 ! - 5
0 1 1 2 2 2 3 2 2 8 2 4 8 2 24 8 2 -16 2 5 -16 2
! 6 / - 7 * 8 * - 9 -
120 -16 2 6 120 -16 2 20 -16 2 -36 2 7 -36 2 -252 2 8 -252 2 -2016 2 2018 9 2018 2009

栈混洗

有一个1, ..., n的序列和一个空栈，每次你可以将序列最前端的数加入栈中或者将栈顶取出（栈非空）。求按取出顺序得到的新序列有多少种？



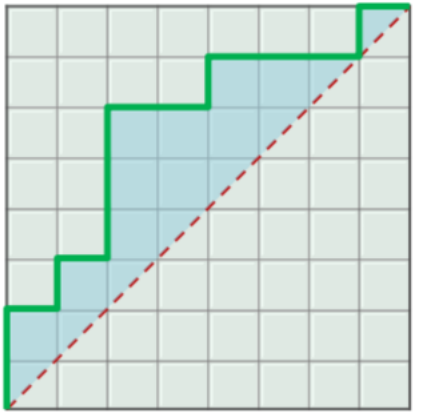
卡特兰数

将入栈看作向上走，出栈看作向右走。

问题即为求从(0, 0)到(n, n)仅向上走和向右走，且不经过右下部分的方案数。

答案为卡特兰数。$C(n) = \frac{(2n)!}{(n+1)!n!}$

题目也等价于长度为2n的括号序列的个数。



甄别栈混洗

输入序列1, ..., n的任一排列，判断是否为栈混洗。

非法情况

先看n=3的情况。

这时不是栈混洗的情况只有一种：312（为什么）。

可以发现，对于任何1<=i<j<k<=n，出现…, k, …, i, …, j, …必非栈混洗。

反过来，不存在312模式的序列，一定是栈混洗吗?

实现

这样我们得到了一个O(n^3)的算法。

更近一步，我们只需要判断i, j, j+1是否合法即可，这样就是O(n^2)的。

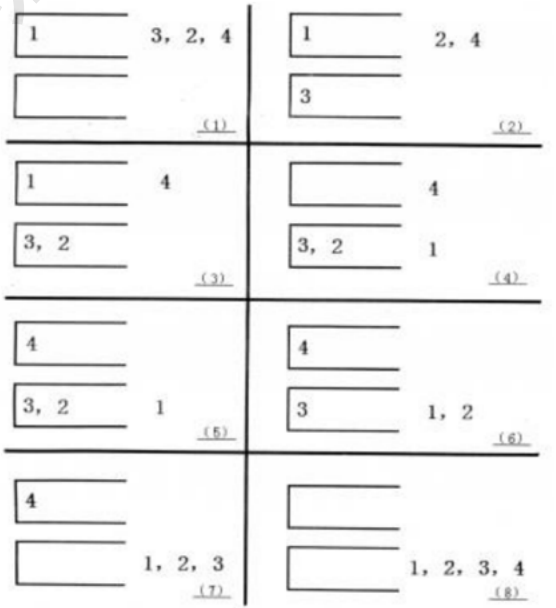
直接模拟！O(n)！

双栈排序

给出一个长度为n的排列，你有两个栈，每一次你可以想其中一个栈加入一个数或者弹出一个数。

给出一个操作方案，使得弹出顺序有序。

n<=1000



双栈排序

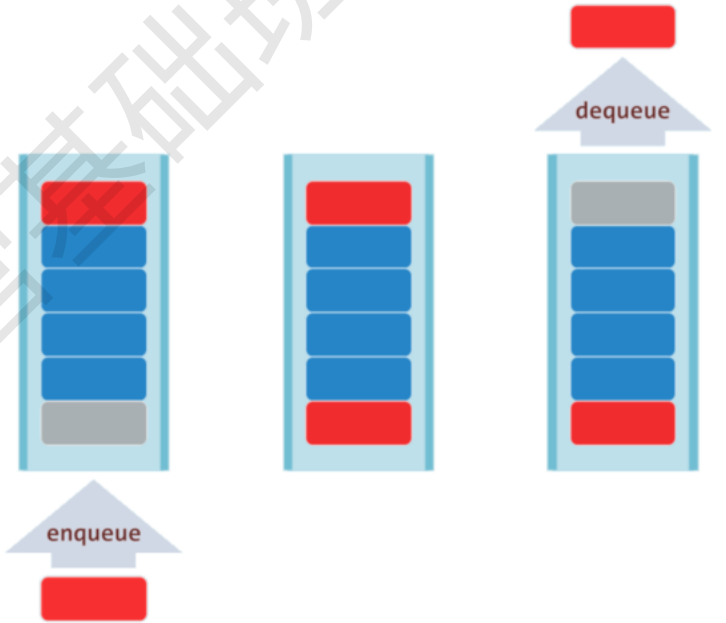
如果i和j不能进入同一个栈就连一条边。

二分图染色。

队列

FIFO(First in first out)表

只能在队尾插入，队头删除。



STL中的队列

```
#include <queue>        //头文件
queue <T> q;            //定义元素类型为T的队列
q.push(a);              //将a加入队列
q.front();              //返回队头元素
q.pop();                //将队头出队列（需要保证队列非空）
q.empty();              //返回队列是否为空
q.size();               //返回队列内元素个数
```

实例

操作	输出	队列（右侧为队头）			
Queue()					
empty()	true				
enqueue(5)		5			
enqueue(3)		3	5		
dequeue()	5	3			
enqueue(7)		7	3		
enqueue(3)		3	7	3	
front()	3	3	7	3	
empty()	false	3	7	3	

操作	输出	队列（右侧为队头）					
enqueue(11)		11	3	7	3		
size()	4	11	3	7	3		
enqueue(6)		6	11	3	7	3	
empty()	false	6	11	3	7	3	
enqueue(7)		7	6	11	3	7	3
dequeue()	3	7	6	11	3	7	
dequeue()	7	7	6	11	3		
front()	3	7	6	11	3		
size()	4	7	6	11	3		

LUCKY SEGMENTS

十进制下仅由数字4和数字7构成的数为幸运数

给定n个区间$[l_1, r_1]$, $[l_2, r_2]$, ... ,$[l_n, r_n]$

每次移动可以将一个区间向左/或向右平移一个单位，即[l,r]变为[l+1, r+1]或[l-1, r-1]

求在不超过k次移动内，n个区间的交集内最多有几个幸运数

$n<=10^5$, k, l_i, $r_i<=10^{18}$

CodeForces 121 D

LUCKY SEGMENTS

可能使用的幸运数约为$O(2^{19})$个

- $2^0+2^1+2^2+\ldots+2^{18}$

用一个队列维护幸运数集合

- 双指针（头、尾）
- 不断扩展头指针（入队），当不满足题设要求时，缩进尾指针（出队）

双栈当队

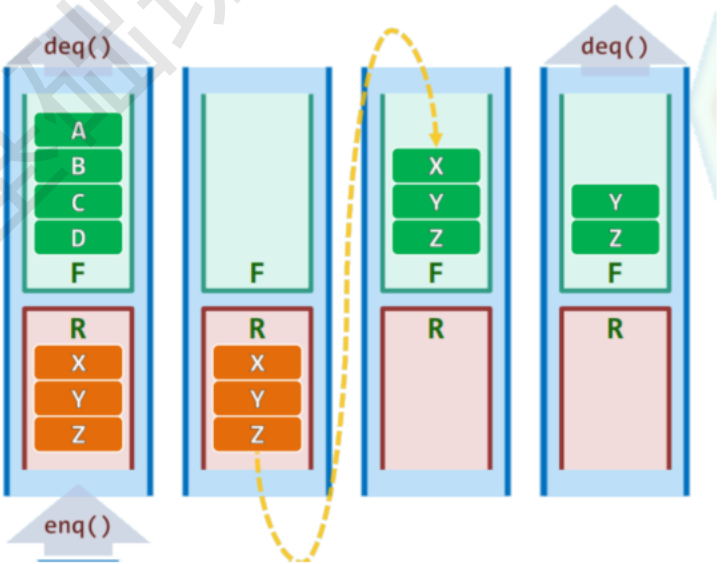
用两个栈实现一个队列。

双栈当队

入队时压入栈A中。

出队时从栈B取出。

栈B为空时将栈A元素依次取出压入栈B



约瑟夫问题

n个人围成一圈，从第一个开始报数，喊到m的人出局，之后从1重新开始喊，最后剩下一个，求出局的顺序。

例如n=6，m=5，出局的顺序是：5，4，6，2，3。

n，m <= 1000

约瑟夫问题

将n个人放入队列，每次将队头m-1人取出再次放入队列，再取出一人即为本轮出局的人。

重复n-1次即可。

复杂度O(nm)

循环队列

用数组实现队列时防止数组长度不够用。

设数组长度为M，则队尾为M-1时入队重置队尾为0.

使用模运算/与运算/判断语句实现

双端队列

队头队尾都可以加入删除的队列。

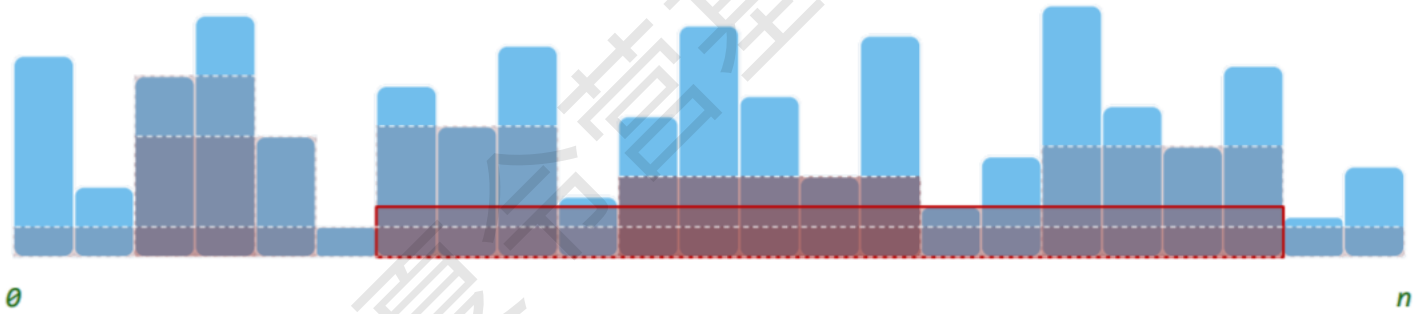
用数组实现时通常使用循环队列。

STL中的双端队列

```
#include <deque>          //头文件
deque <T> q;              //定义元素类型为T的双端队列
q.push_back(a);           //将a加入队尾
q.push_back(a);           //将a加入队头
q.back();                 //返回队尾元素
q.front();                //返回队头元素
q.pop_back();             //将队尾出队列（需要保证队列非空）
q.pop_front();            //将队头出队列（需要保证队列非空）
q.empty();                //返回队列是否为空
q.size();                 //返回队列内元素个数
```

最大矩形

给出一个直方图，求出最大矩形。

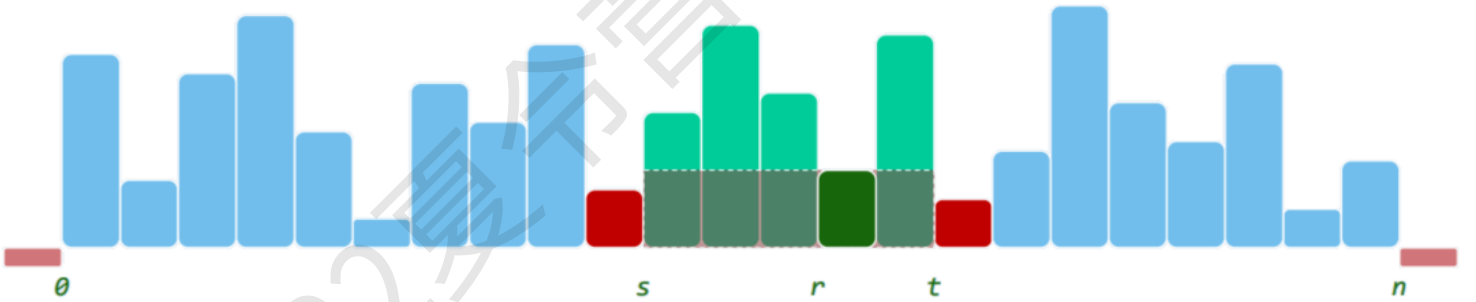


最大矩形

最大矩形一定卡在某一块的高度。

对于每一块，考虑矩形卡在这一块时最大的情况。

则只要知道左右第一个更矮的块。

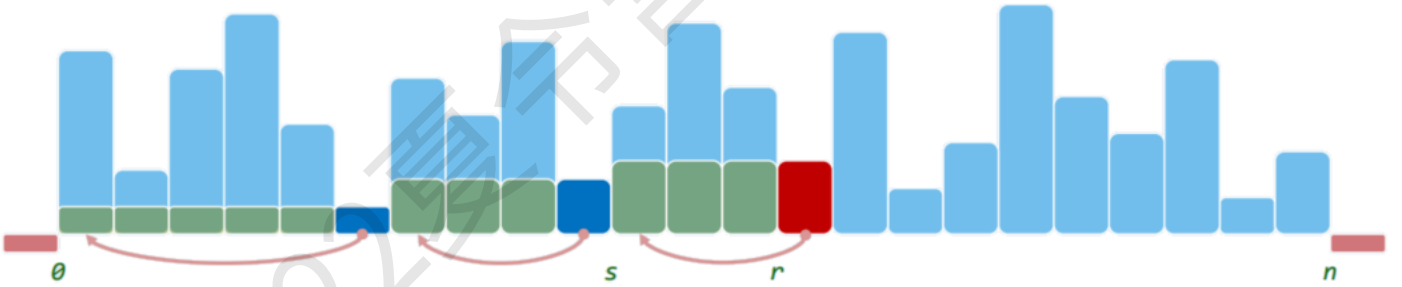


单调栈

考虑求出每一块左边第一个比它矮的块。

维护一个栈，从前往后扫描，不断出栈直到栈顶元素小于当前块，剩下的栈顶就是左边第一个比它矮的块，最后将当前块加入栈中。

保持栈中元素单调的栈就是单调栈。



滑动窗口

有一个长为n的序列A，对于每相邻k个数字求出它们中最小和最大值。

k <= n <= 10^6

Window position	Minimum value	Maximum value
[1 3 -1] -3 5 3 6 7	-1	3
1 [3 -1 -3] 5 3 6 7	-3	3
1 3 [-1 -3 5] 3 6 7	-3	5
1 3 -1 [-3 5 3] 6 7	-3	5
1 3 -1 -3 [5 3 6] 7	3	6
1 3 -1 -3 5 [3 6 7]	3	7

滑动窗口

考虑怎么求每一个最小值，最大值同理。

假设有i>j且ai<=aj，那么当扫描到i之后就不需要考虑j了，它一定不会成为答案。

同时扫描到i之后，不需要考虑i-k以前的位置了，它们也无法成为答案。

因此我们只需要维护如下序列：

i-k<p1<p2<...<pt=i满足$a_{p1}<a_{p2}<...<a_{pt}$

单调队列

用一个双端队列来维护这个序列。

扫描到i时，如果队尾值>=ai，那就从队尾出队，重复这个操作。

如果队头位置<=i-k，那就从队头出队，也重复这个操作。

将i从队尾插入队列，这时队头就是i-k+1到i中的最小值位置。

像这样保持队列中元素单调的队列就是单调队列。

常用于DP的斜率优化中。

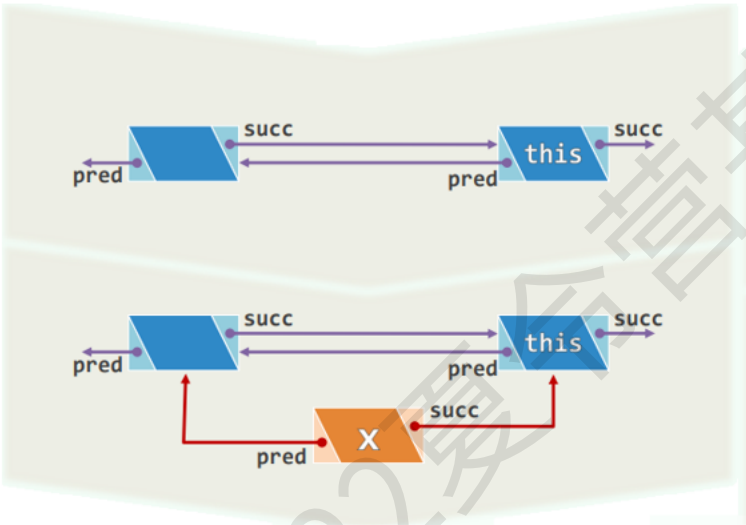
链表

相邻元素互相链接的列表。

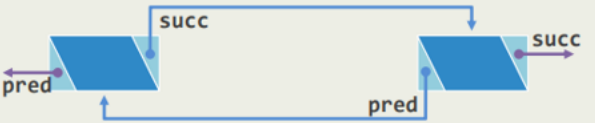
单向链表

双向链表

插入



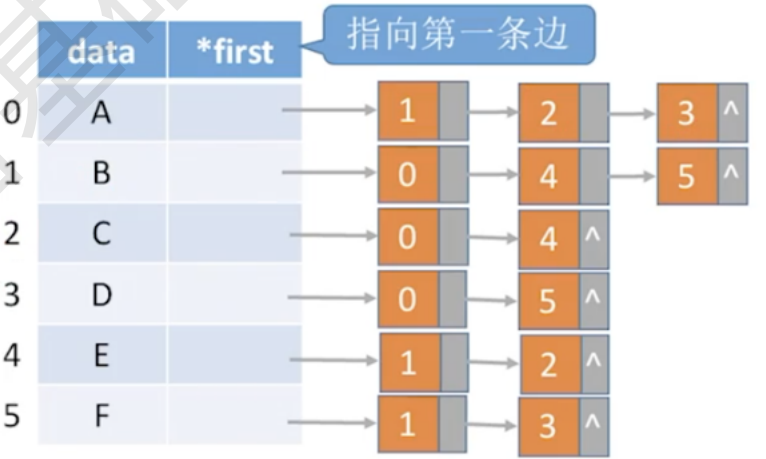
删除



应用：邻接表

最常用的储存图的方式。

用单项链表链接从一点出发的所有边。



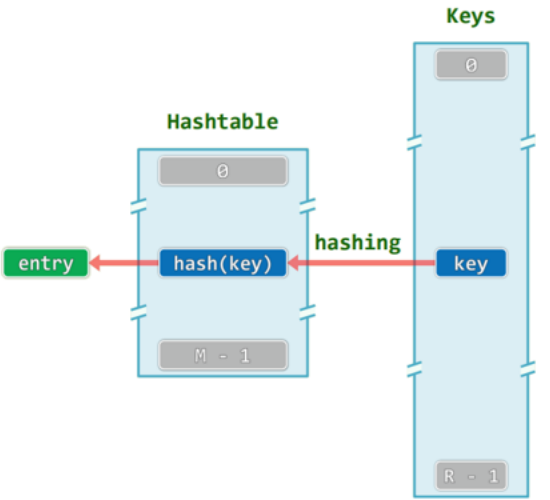
散列表HASH

散列表将关键码值映射到表中一个位置来访问记录，以加快访问速度。

映射函数叫做散列函数。

相同值映射后也相同。

不同值映射后大概率不同。



常用方法

除留余数法

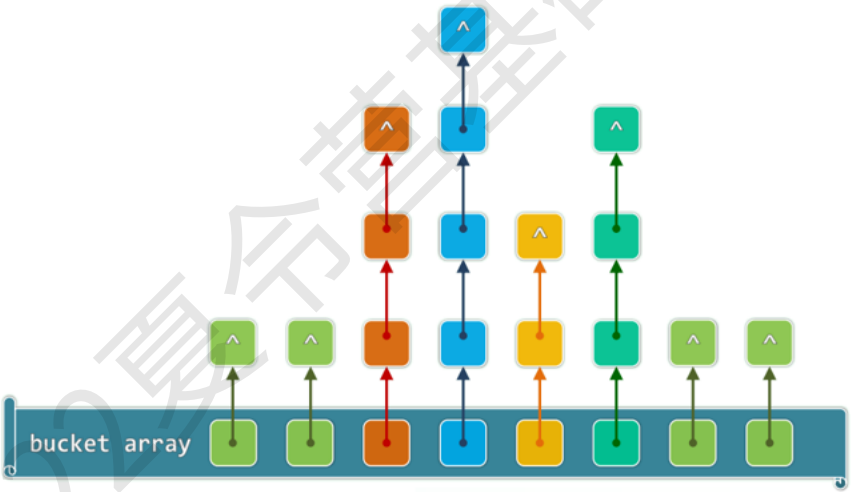
- 取关键字 mod p的值为散列地址。其中p不超过表长且多为质数

字符串hash

- (s[1]*k+s[2]*k^2+…+s[n]*k^n) mod p
- 快速得到一个子串的hash值。

冲突处理：拉链法

每个哈希值链接出一条链表记录所有值



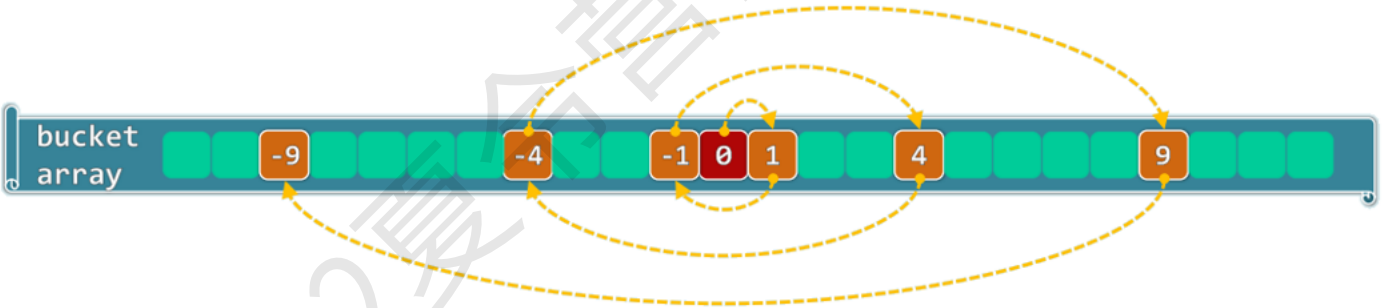
冲突处理：线性试探

遇到冲突跳到下一个位置继续判断是否冲突。

冲突处理：双向平方试探

遇到冲突向前后以平方数为距离跳跃试探。

需要表长为素数M=4k+3。



冲突处理：多模数

使用双模数或更多模数降低冲突概率。

WATCHMEN

给定二维平面的n个坐标，求满足曼哈顿距离等于欧几里得距离的点对数。

1<=n<=10^5, 1<=xi, yi<=10^9，可能有重点。

Codeforces650A

WATCHMEN

符合条件的再同一行/同一列。

用hash统计。

树的同构

给出两颗树，判断它们是否同构。

n<=100000

树的同构：有根树

自上而下hash。

为让子树顺序在同构的树种保持一致，可以将子树hash值排序，得到的顺序再以任何方式进行hash即可。

树的同构：无根树

将树的重心当作根。

PREFIX EQUALITY

给定两个整数序列A和B。Q次询问，每次问A的前xi个数字构成的集合和B的前yi个数字构成的集合是否相同。

1<=n, q<=2*10^5

1<=a[i], b[i]<=10^9

AtCoder Beginner Contest 250

PREFIX EQUALITY

首先重复数字没有用，可以将第二次出现和之后的都置为0。

考虑hash，对于每个前缀，我们需要一个和出现的顺序无关的hash方式。

可以求出前缀异或和即可。