CSP-J数据结构

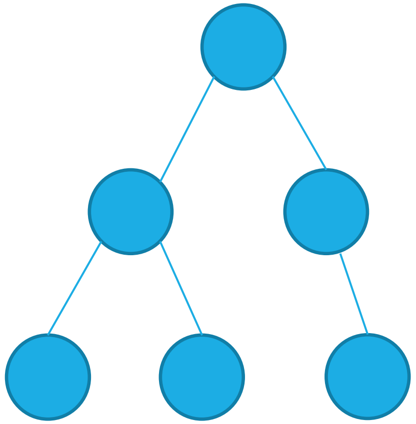
蔡子逸

什么是数据结构

数据结构其实大家并不陌生，最简单的，一个数组、一个字符串、一棵树也算是数据结构。

数据结构储存了一定类型的数据，比如字符、数字。

数据结构具有一定的组织形式，比如一个字符串，每个字符和前后字符连在一起；一棵树，每个节点和父亲、儿子节点连在一起。



CSP-J 中的数据结构

年份	题目	考查内容
NOIP2017	跳房子	单调队列优化dp
NOIP2018		
CSP-J2019		
CSP-J2019(JX)		
CSP-J2020	直播获奖	桶
CSP-J2021	小熊的果篮	链表

数组-前缀和

数组是最简单也是最基础的数据结构。

前缀和是最简单也是最常用的一种维护区间信息的手段。

可以用前缀和维护的信息都是可合并的。

一般情况下，如果维护的信息是可逆的，我们可以通过前缀和差分获取任意区间的信息。

如果是不可逆的信息，我们就要利用可合并性，将区间拆分成处理过的部分。

FOOL

有一个长度为 n 的自然数数组 a_n，一个正整数 L，一个正整数 M。

有 q 次询问，每次询问给定一个长度一定为 L 的区间 $[l_i, r_i]$。

对于每个询问，你需要输出下标 i 在这段区间的 a_i 的乘积，结果对 M取模。

$1 \le n, q \le 10^7$

FOOL

模运算下的乘法不可逆，没法前缀和差分。

注意到询问的长度一定是 L。

我们每 L 个数分一段，维护每一段内前后缀的乘积。

询问就可以拆成两段直接乘起来了。

时间复杂度 $O(n+q)$。

数组-桶

当值域不过大时，桶是一种维护集合内元素出现次数的有效手段。

常常和双指针配合使用。

CSP-J2020 直播获奖

给定 w。

输入 n 个整数，对于前 $i(=1,2,\ldots,n)$ 个整数组成的集合，输出从大到小第 $\max(1,\lfloor p\times w\%\rfloor)$ 个数。

$1\leq n\leq 10^5$

给定的整数非负且大小不超过 600。

CSP-J2020 直播获奖

这题考场上有许多千奇百怪的做法，诸如对顶堆等等。

但是注意到整数数值范围很小，其实我们可以开一个桶记录每个整数出现次数。

每次计算答案的时候从大到小暴力枚举，当数量达到要求的时候停止即可。

洛谷1638 逛画展

一个长度为 n 的数组，每个元素都是 $[1,m]$ 的整数。

请计算一个最短的区间，使得区间内 $[1,m]$ 每个数都出现了至少一次。

$1 \le n \le 10^6, 1 \le m \le 2 \times 10^3$

逛画展

假设我们固定了区间的左端点，寻找区间右端点。

我们肯定是找到第一个满足所有数都出现的位置。

使用桶维护每个数的出现次数。右端点不断向右移动并更新桶。当一个数出现次数从 0 变为非 0 的时候，出现的数的种类 $+1$。当种类数目为 m 的时候，我们可以停下了。

逛画展

目前的复杂度是 $O(n^2)$。注意到左端点移动的时候，我们右端点需要重头开始，非常浪费。

既然桶可以处理增加一个数，为什么不能处理减少一个数呢？当出现次数重新变为 0 的时候，出现种类数 -1。

左端点向右移动一格，右端点的合法位置不可能倒退，所以不必重头来过。

时间复杂度 $O(n)$，因为两个端点都是单向移动。

```
for (int l = 1, r = 1, cnt = 0;l <= n;++ l) {
	for (;r <= n && cnt < m;++ r)
		cnt += !bin[a[r]], ++ bin[a[r]];
	if (cnt == m) {
		// [l, r - 1] -> ans
	}
	-- bin[a[l]], cnt -= !bin[a[l]];
}
```

链表

与数组不同，链表常被用于处理离散的数据。

链表删除和插入一个元素很方便，但是定位一个元素位置需要线性的时间。

如果我们有办法快速定位一个元素，或者这个定位的过程刚好和其他操作重合，那链表就是最有力的工具。

CSP-J2021 小熊的果篮

n 个水果排成一排，每个水果要么是苹果要么是桔子。

连续排在一起的同一种水果叫做一“块”。

把这一排水果挑到若干个果篮里，具体方法是：每次都把每一个“块”中最左边的水果同时挑出，组成一个果篮。重复这一操作，直至水果用完。注意，每次挑完一个果篮后，“块”可能会发生变化。比如两个苹果“块”之间的唯一桔子被挑走后，两个苹果“块”就变成了一个“块”。

请计算每个果篮里包含的水果。

$1 \le n \le 2 \times 10^5$

CSP-J2021 小熊的果篮

首先对输入序列建双向链表，维护每一个“假块”头建双向链表，共维护两个链表。

这里的“假块”指每个“假块”中水果种类必定相同，但相邻“假块”中水果种类可能相同。

我们可以使用双向链表的删除元素来模拟吃一个水果。

不断循环遍历“假块”头链表，遍历过程中记录上一个被吃水果种类。遍历到某个块头时，若其指向的水果与上一个被吃水果种类相同，直接将这个块头删除，相当于合并块；若不同，吃掉这个水果，更新上一个被吃水果种类，将这个块头指向的水果变成被吃水果的下一个水果。

CSP-J2021 小熊的果篮

关于一个假块被吃完的处理方法，此时这个假块的块头一定会指向下一个块头。若这个块头的种类与被吃水果的种类不同，删掉这个块，因为遍历下一个块时将会吃掉这个水果；若相同，不动，因为接下来的过程将会把下一个块的块头删除。这样保证遍历时不会出现长度小于一的假块。

若假块没有被吃完，其指向的下一个水果一定与吃掉的种类相同，同样不做任何处理。

时间复杂度 $O(n)$。

栈与队列

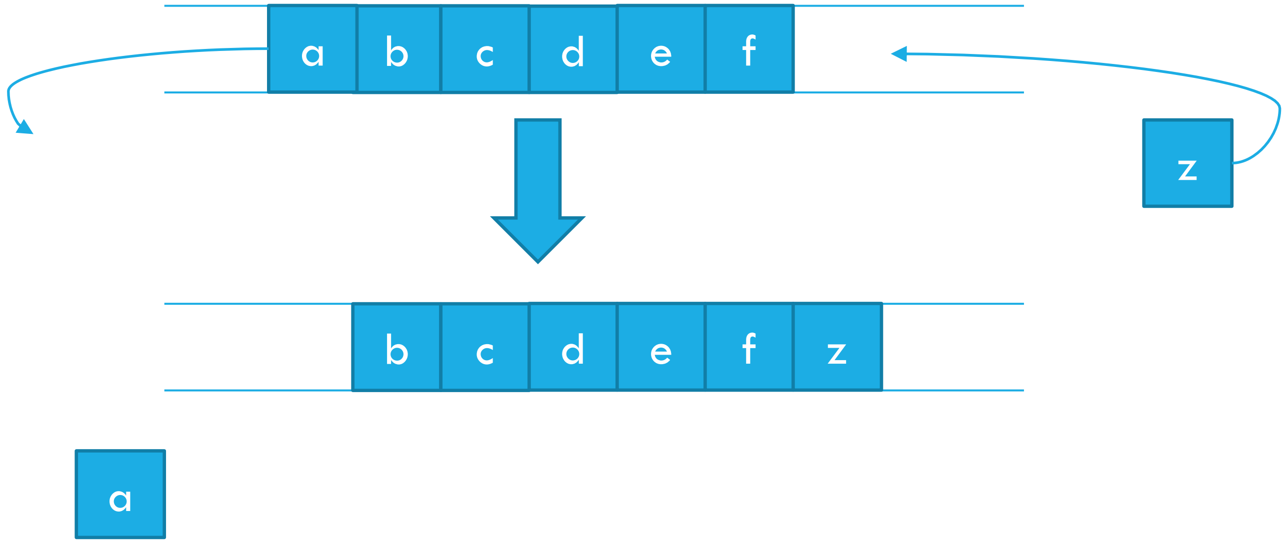
作为最基本的数据结构，栈与队列被广泛应用，我们先给出定义，然后通过例题来熟悉他们。

队列

定义：

队列是一种特殊的线性表，特殊之处在于它只允许在表的前端（$front$）进行删除操作，而在表的后端（$rear$）进行插入操作。

一般的，我们用数组来实现：维护两个变量l,r，表示队头为$a[l]$，队尾为$a[r]$，删除队头时，$a[l++] = 0$，加入队尾时$a[++r] = z$;



NOI.AC 2830 INTRODUCE A LITTLE ANARCHY

有 m 个班总共 n 个同学排队打饭。

第 i 个同学班级是 a_i。

维护一个“队列”，要求支持

- push x：同学 x 入队，如果前面有同班同学，则排在最后一个同班同学的后面。否则排在队尾。
- pop：队伍最前面的同学出队。

每次出队操作输出对应同学的编号。

$1 \le n \le 10^5, 1 \le m \le 300$

操作次数不超过 10^5

NOI.AC 2830 INTRODUCE A LITTLE ANARCHY

可以发现同班的同学一定是队伍中连续的一段。

主队列维护队伍中从前到后的班级。

m 个副队列维护每个班的排队信息。

时间复杂度 $O(n)$。

实验舱 #2584 STACK

维护一个 0/1 栈，要求支持以下操作。

普通的压栈和弹栈，同时还有栈翻转，以及询问从栈顶到栈底依次执行与非运算的值。

$x \text{ nand } y = \text{not}(x \text{ and } y)$

$1 \leq n \leq 10^5$

实验舱 #2584 STACK

观察与非的运算表，只要运算数有 0 就返回 1，否则是 0。

所以本质上我们只关注最后一个 0 的位置，从这个位置开始，运算结果就变成 0/1 不断重复。

使用双端队列维护栈中所有 0 的位置，以及它们之间的间隔，栈翻转只需要记录队列方向实现。

求结果的话，找到最后一个 0，然后用它离栈底的位置的奇偶性得出结果。

时间复杂度线性。

COCI2007-2008#5 AVOGADRO

一张 3 行 n 列的表格，然后将整数 1 到 n 写进表格。

对于表格第一行，每个整数只出现一次。对于其余两行，每个数字可以出现任意次或者不出现。

现在可以删去任意一些列。完成后，对这个表的每一行进行升序排序。

目标是使得表中的三行在升序排序后完全相同。请求出他至少需要删去多少列。

$1 \le n \le 10^5$

COCI2007-2008#5 AVOGADRO

重要条件：第一行每个数出现且仅出现一次。

若两个集合大小相同，且其中一个集合出现过的数在另一个集合中均出现，等价于两个集合一定相同。

所以，在第二第三行中没有出现过的，一定要在第一行中删掉。

在第一行删掉会导致第二第三行对应的数被删，如果导致了新的数字不出现，又要在第一行删掉。

这是一个类似于bfs扩展的过程，使用队列来维护即可。

时间复杂度 $O(n)$。

洛谷1886 滑动窗口

给出一个长度为 n 的数组，给定 k，求出每 k 个连续的数中的最大值和最小值。

$1 \leq n \leq 10^7$

滑动窗口

直接暴力的时间复杂度是 $O(nk)$。

很显然，这其中进行了很多重复工作。

滑动窗口

我们考虑从左往右计算每连续 k 个数的极值（下面以最大值为例）。

明显，当一个数进入所要“寻找”最大值的范围中时，若这个数比其之前的数要大，显然，这些之前的数会比这个数先离开范围且不再可能是最大值。

也就是说，一旦满足这些条件，这些数我们永远不会再次考虑。

单调队列

这启发我们使用一个队列来维护。

队列从头到尾单调递减，储存的是今后可能会考虑的数。

每次向右移动时，我们不断将新进入范围的数和队尾比较，如果能替换队尾就让队尾出队，直到队尾大于新的数。

取区间最大值，只需要不断让队头出队直到队头在区间内，取队头即可。

每个数入队一次，出队一次，时间复杂度是线性的。

滑动窗口

```
//维护最小值，序列要单调递增
int back = 1, front = 0;//队列头尾（back是头，打的时候）
for(int i = 1; i <= n; ++i){
    //如果如果尾部的在滑块之外，则删掉他们
    while(back <= front && q[back]+k <= i) ++back;
    //如果头部的比当前的大，那么a[i]从头进去，队列就不递增了，所以删掉比a[i]大的头部
    while(back <= front && a[i] < a[q[front]]) --front;
    q[++front] = i;//头部插入i
    if(i >= k) printf("%d ", a[q[back]]);//因为单调递增，所以答案在a的下标就是q[back]
}
```

栈

定义：

栈是仅允许在表的一端进行插入和删除运算。这一端被称为栈顶，相对地，把另一端称为栈底。

一般的，我们用数组来实现，把数组的第一位作为栈底，维护一个变量r同时表示栈里元素个数与栈顶元素的位置。

若把$a[1]$作为栈底，删除栈顶时，$a[r--]=0$；加入新元素x时$a[++r]=x$。

HDU4699 EDITOR

使用文档编辑器维护一个初始为空的数列，要求支持五种操作：

I x: 在光标后面插入 x 且光标后移一位

D: 删除光标前的一个数

L: 将光标左移一位

R: 将光标右移一位

Q x: 查询前 x 个数组成的序列的最大前缀和

$q \leq 10^6$

HDU4699 EDITOR

两个栈分别维护光标前后的数

至于最大前缀和，可以对光标前的栈实时维护。

时间复杂度 $O(n)$。

洛谷2866 BAD HAIR DAY

有 n 头牛从左到右排成一排，每头牛有一个高度 h_i，设左数第 i 头牛与它右边第一头高度大于等于 h_i 的牛之间有 c_i 头牛，试求 $\sum_{i=1}^{n} c_i$。

$1 \leq n \leq 10^6$

单调栈

从右往左计算，考虑动态维护这个第一头比当前高的牛。

对于当前位置右边，从左往右高度依次为 h_1 和 h_2 的两头牛。

如果 $h_1 \geq h_2$ 则 h_2 完全没有考虑的价值，因为左侧的牛向右找更高的牛，只会找到 h_1 而不可能是 h_2。

使用一个栈维护所有还有考虑价值的牛，具体来说就是一个从栈顶到栈底单调递增的栈。

考虑到 i 时，用 h_i 弹出所有不符合要求的栈顶，那么最后的栈顶就是要找的牛，接着把 h_i 入栈。

时间复杂度 $O(n)$。

堆

堆是一种用于动态维护集合极值的数据结构，广泛用于各类题目中。

最基础的堆是二叉堆，插入和弹出堆顶都是 $O(\log n)$ 的，

更高级的堆可以高效支持插入操作和合并操作，但是用的不多。

在实际比赛中，一般都会使用STL封装好的priority_queue或者set代替。

对顶堆

对顶堆是用于解决动态维护集合第 k 大元素的一种技巧。顾名思义，对顶堆就是两个堆，一个是小根，另一个是大根。

小根堆维护大值即前 k 大的值（包含第 k 个），大根堆维护小值即比第 k 大数小的其他数。

对顶堆

维护：当小根堆的大小小于 k 时，不断将大根堆堆顶元素取出并插入小根堆，直到小根堆的大小等于 k；当小根堆的大小大于 k 时，不断将小根堆堆顶元素取出并插入大根堆，直到小根堆的大小等于 k。

插入：若插入的元素大于等于小根堆堆顶元素，则将其插入小根堆，否则将其插入大根堆，然后维护对顶堆。

查询第 k 大元素：小根堆堆顶元素即为所求；删除第 k 大元素：删除小根堆堆顶元素，然后维护对顶堆。

显然，查询第 k 大元素的时间复杂度是 $O(1)$ 的。由于插入删除后，小根堆大小和期望的 k 最多相差 1。因此每次维护只需要对两个堆各进行一次调整，所以这些操作时间复杂度都是 $O(\log n)$ 的。

对顶堆模板

[Running Median - POJ 3784 - Virtual Judge (vjudge.net)]

并查集

我们常常遇到这样一种问题：维护若干个集合，集合里面有若干的元素。

也就是说，我们需要实现这样的操作：

1，合并两个不相交的集合

2，询问一个元素在哪个集合，进而回答两个元素是否在同一个集合。

我们使用并查集，能够快速解决这个问题。

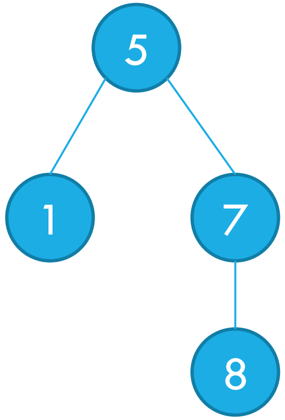
并查集

我们在计算机中常使用有根树来表示一个集合，例如一个集合$S = \{1,5,7,8\}$，它可能会被表示为如右图的树：

即我们先用点记录数值，然后把他们连成一颗树。

树根是随意指定的。当我们询问两个点是否在同一个集合里的时候，我们仅需要查看两个点所在的树是否有同一个根。

当合并两个集合的时候，我们只需要把一个集合的树根连接至另一颗树的根即可。



让我们看一道例题熟悉一下。

例题：亲戚

若某个家族人员过于庞大，要判断两个是否是亲戚，确实还很不容易，现在给出某个亲戚关系图，求任意给出的两个人是否具有亲戚关系。

规定：x和y是亲戚，y和z是亲戚，那么x和z也是亲戚。如果x,y是亲戚，那么x的亲戚都是y的亲戚，y的亲戚也都是x的亲戚。

具体地，给出n个人，m个亲戚关系，以及p个询问，每对关系与询问均包含两个数x,y，表示x和y是亲戚或询问他们是否是亲戚。

50%的数据：$n,m,p \leq 5000$

100%的数据：$n,m,p \leq 10^5$

例题：亲戚

对于50%的数据：

在表示集合的有根树里面，我们设$fa[i]$表示第i个人的父亲节点。特别的，如果他是根，我们令$fa[i]=i$。

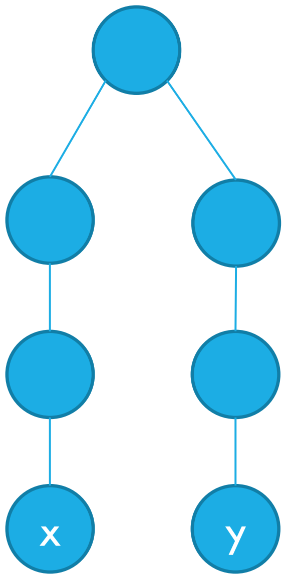
做法非常简单，初始时，每个点都是单独的一棵树，读入m组关系的同时进行集合的合并，即寻找到x,y的根，并把根连起来。之后对p组询问(x,y)，只需要让x和y不断利用fa数组往上跳，然后比较x,y最终跳到的节点是否是同一个即可。

并查集：路径压缩优化

注意到上面的方法并不能通过100%的数据，因为在最坏情况下，查询的时间复杂度会很大。

比如：所有人都在同一颗树里，但是树仅由两条链构成，每次查询的点位于链的末端。这个时候单次查询复杂度$O(n)$，那么总复杂度$O(np)$，不能承受。

我们要想办法优化。



并查集：路径压缩优化

观察到最坏情况中，询问时走的路径重复了很多，那么我们是否可以走了一次之后就不再重复走?

观察到，我们只是想知道树的根是谁，并不需要知道走了一条怎样的路。

那么得出优化：每次从x_0向上跳的时候，假设最终跳到了根$root$，路径上经过了$\{x_1, x_2, x_3 \dots x_n\}$，我们令$fa[x_i] = root, i = 0 \dots n$。

下次再询问$x_{0..n}$时，我们只需要跳一次即可，节省了大量时间。

对于添加此优化后的时间复杂度无法严格分析，但实践上一般情况下其略小于$O(nlogn)$。

为了达到 $O(n\alpha(n))$ 的复杂度，需要按秩合并优化，此处略去。

```
int fa[100001];
int get_root(int i)
{
    return fa[i]==i? i:fa[i]=get_root(fa[i]);
}
int main()
{
    int n,m;
    n=read();m=read();
    int a,b;
    for(int i=1;i<=n;i++) fa[i]=i;
    for(int i=1;i<=m;i++)
    {
        a=read();b=read();
        fa[get_root(a)]=get_root(b);
    }
    int fa,fb;
    int p=read();
    for(int i=1;i<=p;i++)
    {
        a=read();b=read();
        fa=get_root(a);fb=get_root(b);
        if(fa==fb) printf("Yes\n");
        else printf("No\n");
    }
    return 0;
}
```

```
int merge(int x, int y) {
    int fx = getfather(x), fy = getfather(y);
    if (rk[fx] > rk[fy]) fa[fy] = fx, rk[fx] += rk[fx] == rk[fy];
    else fa[fx] = fy, rk[fy] += rk[fx] == rk[fy];
}
```

DESTROYING ARRAY

给定由 n 个非负整数组成的数列 $\{a_i\}$，每次可以删除一个数，求每次删除操作后的形成的数列集合中，和最大的是多少。

$1 \le n \le 10^5$

DESTROYING ARRAY

倒序执行，将删除数变为添加数，实时维护最大数列和。

每次添加一个数，就看看它前后是否以及被添加了，如果已经被添加了就使用并查集并起来。并查集顶部维护集合的和。每次合并完后，用新的和去更新当前最优解。

时间复杂度 $O(n\alpha(n))$。