第2章 数据结构基础

在本章中，我们将学习栈、队列、链表、ST 表、树和二叉堆等基础数据结构。此外，本章还介绍了算法竞赛中常用的 STL 函数和容器以及部分 pb_ds 库，帮助读者简洁、高效地编写程序。这些数据结构的设计思路来源于程序设计中的基本算法思想，为我们进一步研究和实现更加深入的算法提供了工具基础。

2.1 栈

栈是一种“先进后出”的线性数据结构。栈只有一端能够进出元素，我们称这一端为栈顶，另一端为栈底。栈最为重要的两个操作是进栈和出栈。添加元素时，我们只能将其插入到栈顶，称为进栈；删除时，我们只能将栈顶元素取出，称为出栈。

2.1.1 栈

例题　　【模板】栈（洛谷 B3614）

请你实现一个栈（stack），支持如下操作：

push(x)：向栈中加入一个数 x。

pop()：将栈顶弹出。如果此时栈为空则不进行弹出操作，输出 Empty。

query()：输出栈顶元素，如果此时栈为空则输出 Anguei!。

size()：输出此时栈内元素个数。

【输入格式】

本题单测试点内有多组数据。

输入第一行是一个整数 T，表示数据组数。对于每组数据，格式如下：

每组数据第一行是一个整数，表示操作的次数 n。

接下来 n 行，每行首先由一个字符串，为 push，pop，query 和 size 之一。若为 push，则其后有一个整数 x，表示要被加入的数，x 和字符串之间用空格隔开；若不是 push，则本行没有其它内容。

对于全部的测试点，保证 $1 \leq T, n \leq 10^6$，且单个测试点内的 n 之和不超过10^6，即 $\sum n \leq 10^6$。保证 $0 \leq x < 2^{64}$。

【输出格式】

对于每组数据，按照题目描述中的要求依次输出。每次输出占一行。

【分析】

我们可以用一个数组 st 和一个变量 tp（记录栈顶位置）实现栈，四种操作的时间复

杂度均为$O(1)$，总时间复杂度为 $O(\sum n)$。

```
#include <bits/stdc++.h>
using namespace std;

typedef unsigned long long ull;

const int M = (int)1e6;

ull st[M + 5], x;
int tp, n;

void work() {
    scanf("%d", &n); tp = 0;
    char op[10];
    for (int i = 1; i <= n; i++) {
        scanf("%s", op);
        if (op[0] == 'p' && op[1] == 'u') {  // push
            scanf("%llu", &x);
            st[++tp] = x;
        } else if (op[0] == 'p') {  // pop
            if (tp) tp--;
            else puts("Empty");
        } else if (op[0] == 'q') {  // query
            if (tp) printf("%llu\n", st[tp]);
            else puts("Anguei!");
        } else printf("%d\n", tp);  // size
    }
}

int main() {
    int T; scanf("%d", &T);
    for (int ca = 1; ca <= T; ca++) {
        work();
    }
    return 0;
}
```

例题　Editor（HDU-4699）

请你实现一个用于维护整数序列的编辑器，编辑器初始时序列为空，支持以下五种操作。

I x：在光标后插入 x。

D：删除光标后的元素。

L：光标向左移动一个位置。

R：光标向右移动一个位置。

Q k：设光标前的序列为 $\{a_1, a_2, \cdots, a_n\}$，$S_i = a_1 + a_2 + \cdots + a_i$。查询 $\max\limits_{1 \le i \le k} S_i$。

【输入格式】

输入包含多组测试数据。每组测试数据首先输入一个整数 Q 表示操作次数，接下来 Q 行，每行包含上述操作之一。

$1 \le Q \le 10^6, |x| \le 10^3, 1 \le k \le n$。

【输出格式】

对于每次查询操作，输出 $\max\limits_{1 \le i \le k} S_i$。

【分析】

维护两个栈 pre 和 suf，分别表示光标前的序列和光标后的序列。mx 数组维护光标前的序列的所有前缀的前缀和的最大值，即 $mx[k] = \max\limits_{1 \le i \le k} S_i$。

由于栈 pre 存储了从序列开头到光标前的元素，栈 suf 存储了光标后到序列结尾的元素，因此两个栈合起来就得到了整个序列，这样就很容易实现 IDLR 这四种操作。在光标从 $preTp - 1$ 移动到 $preTp$ 时，通过 $mx[preTp] = \max(mx[preTp - 1], S_{preTp})$ 维护 mx 数组，其中 S_{preTp} 可以通过增量 $O(1)$ 维护，查询时答案即为 $mx[k]$。

五种操作的时间复杂度均为 $O(1)$，总时间复杂度为 $O(\sum q)$。

```
const int M = (int)1e6;
const int inf = 0x3f3f3f3f;

int q, mx[M + 5];
int pre[M + 5], preTp;
int suf[M + 5], sufTp;

void work() {
    mx[0] = -inf;
    preTp = sufTp = 0;
    for (int i = 1, op, x, k, s = 0; i <= q; i++) {
        getchar(); op = getchar();
        if (op == 'I') {
            scanf("%d", &x);
            pre[++preTp] = x;
            s += x;
            mx[preTp] = max(mx[preTp - 1], s);
        } else if (op == 'D') {
            if (preTp) s -= pre[preTp--];
```

```
20.         } else if (op == 'L') {
21.             if (preTp) {
22.                 suf[++sufTp] = pre[preTp];
23.                 s -= pre[preTp--];
24.             }
25.         } else if (op == 'R') {
26.             if (sufTp) {
27.                 pre[++preTp] = suf[sufTp];
28.                 s += suf[sufTp--];
29.                 mx[preTp] = max(mx[preTp - 1], s);
30.             }
31.         } else {
32.             scanf("%d", &k);
33.             printf("%d\n", mx[k]);
34.         }
35.     }
36. }
37.
38. int main() {
39.     while (~scanf("%d", &q)) work();
40.     return 0;
41. }
```

2.1.2 单调栈

例题　【模板】单调栈（洛谷 P5788）

给出项数为 n 的整数数列 $a_{1\cdots n}$。

定义函数 $f(i)$ 代表数列中第 i 个元素之后第一个大于 a_i 的元素的下标，即 $f(i)=\min_{i<j\le n,a_j>a_i}\{j\}$。若不存在，则 $f(i)=0$。

试求出 $f(1\cdots n)$。

【输入格式】

第一行一个正整数 n。

第二行 n 个正整数 $a_{1\cdots n}$。

$1\le n\le 3\times 10^6, 1\le a_i\le 10^9$。

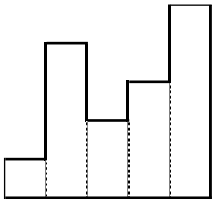
【输出格式】

一行 n 个整数表示 $f(1\cdots n)$ 的值。

【分析】

为了便于理解，我们赋予这道题目一个现实背景。有 n 个人排成一行，第 i 个人的身高是 a_i，$f(i)$ 表示第 i 个人向后平视看所能看到的最后一个人。例如 $a_{1\cdots n}=[1,4,2,3,5]$，

五个人的位置及身高情况可以抽象为下图。



可以发现第 i 个人会把编号大于 i 且其身高不大于 a_i 的人遮住。形式化来讲，对于 $k<i<j$，若 $a_i \geq a_j$ 那么第 k 个人无法看到第 j 个人。因此对于第 j 个人来说，如果存在身高满足 $a_i \geq a_j\ (i<j)$ 的人，那么 a_j 不会影响到 $f(1 \cdots i-1)$ 的取值，可以直接丢弃掉。这启发我们可以维护一个栈 st 存储人的编号，栈中元素满足 $a_{st[i]} < a_{st[j]}\ (i<j)$。枚举 i 从 n 到 1，比较 a_i 与编号为栈顶元素的人的身高，若 $a_i \geq a_{st[tp]}$ 则出栈，继续比较并出栈直到 $a_i < a_{st[tp]}$，于是 $f(i)=st[tp]$。之后再将 i 进栈，便可以使用同样的算法计算 $f(i-1)$。

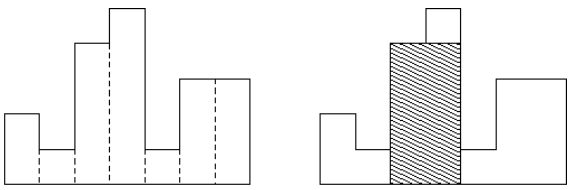
由于每个元素只进栈一次，只出栈一次，计算每个 f_i 的均摊时间复杂度为 $O(1)$，总时间复杂度为 $O(n)$。

```
1. for (int i = n; i >= 1; i--) {
2.     while (tp && a[i] >= a[st[tp]]) tp--;
3.     f[i] = st[tp];
4.     st[++tp] = i;
5. }
6. for (int i = 1; i <= n; i++) printf("%d%c", f[i], " \n"[i == n]);
```

这就是著名的单调栈算法，借助单调性处理问题的思想在于及时排除不可能的选项，保持策略集合的高度有效性和秩序性，从而为我们作出决策提供更多的条件和可能方法。

例题　Largest Rectangle in a Histogram（POJ2559）

如下图所示，在一条水平线上排列了若干个矩形，每个矩形的宽度都为 1，高度不尽相同，构成一张直方图。求直方图中的子矩形最大面积，例如在下图中答案为阴影部分面积。



【输入格式】

输入包含多组测试数据。每组测试数据首先输入一个整数 n 表示矩形个数，接下来输入 n 个整数 $h_1, h_2, \cdots, h_n$，其中 h_i 表示从左到右第 i 个矩形的高度。

$1 \leq n \leq 10^5, 0 \leq h_i \leq 10^9$。

【输出格式】

对于每组测试数据，输出一个整数表示直方图中的子矩形最大面积。

【分析】

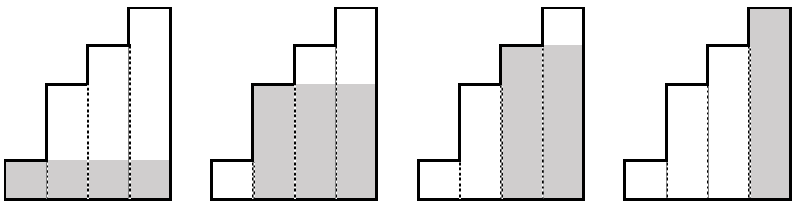
解法一

使用单调栈处理出两个数组 l_i 和 r_i，其中 l_i 表示第 i 个矩形左边最后一个高度不小于第 i 个矩形高度的编号，r_i 表示第 i 个矩形右边最后一个高度不小于第 i 个矩形高度的编号，即 $l_i = \min\limits_{1\le j\le i, h_i\le h_j} j$，$r_i = \max\limits_{i\le j\le n, h_i\le h_j} j$。答案即为 $\max\limits_{1\le i\le n}(r_i - l_i + 1)\times h_i$。时间复杂度为 $O(n)$。

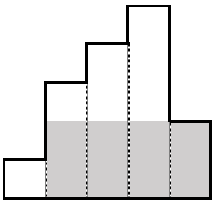
```
1. h[st[tp = 1] = 0] = -1;
2. for (int i = 1; i <= n; i++) {
3.     while (tp && h[st[tp]] >= h[i]) tp--;
4.     l[i] = st[tp] + 1;
5.     st[++tp] = i;
6. }
7. h[st[tp = 1] = n + 1] = -1;
8. for (int i = n; i >= 1; i--) {
9.     while (tp && h[st[tp]] >= h[i]) tp--;
10.    r[i] = st[tp] - 1;
11.    st[++tp] = i;
12. }
13. ll ans = 0;
14. for (int i = 1; i <= n; i++) ans = max(ans, (ll)(r[i] - l[i] + 1) * h[i]);
15. printf("%lld\n", ans);
```

解法二

我们首先来思考一个比较简单的情况，假设矩阵的高度从左到右递增。此时我们可以枚举每个矩形，将该矩形的高度作为答案矩形的高度，并把宽度延伸到右边界，如下图中灰色部分所示，所有枚举情况的面积最大值即为答案。



如果当前矩形的高度比上一个的小，那么该矩形想要与之前的矩形组合时，组合的子矩阵的高度一定不大于当前矩形的高度，也就是说下图中灰色上方的部分对之后的答案计算无影响。因此，我们可以把这些比该矩形高的矩形删掉，用一个宽度累加，高度为该矩形高度的新矩形代替，即下图中的灰色部分。



详细来说，我们可以维护一个单调栈 st 记录新矩形的高度，栈内元素从栈底到栈顶递增，同时维护一个宽度数组 w 表示新矩形的宽度。我们从左到右依次扫描每个矩形。如果当前矩形不比栈顶矩形高，则不断出栈直至当前矩形比栈顶矩形高。在出栈过程中，累计被弹出的矩形宽度之和，每弹出一个矩形就用它的高度乘以累积宽度更新答案。整个出栈过程结束后，我们把一个高度为当前矩形，宽度为累计宽度的新矩形入栈。为了简化代码，我们可以增加一个高度为 0 的矩形 h_{n+1}，以避免在扫描结束后栈中有剩余矩形。时间复杂度为 $O(n)$。

```
ll ans = 0;
h[n + 1] = tp = 0;
for (int i = 1; i <= n + 1; i++) {
    int width = 0;
    while (tp && st[tp] >= h[i]) {
        width += w[tp];
        ans = max(ans, (ll)width * st[tp]);
        tp--;
    }
    st[++tp] = h[i];
    w[tp] = width + 1;
}
printf("%lld\n", ans);
```

2.2 队列

队列是一种“先进先出”的线性数据结构。一般来讲，队列左端称为队头，右端称为队尾。队列最为重要的两个操作是入队和出队。添加元素时，我们只能将其插入到队尾，称为入队；删除时，我们只能将队头元素取出，称为出队。我们可以用一个数组 q 和两个变量 l,r（记录队头和队尾位置）来实现队列。此外，随着不断的入队和出队，数组 q 的开头部分空间将被浪费，我们可以将数组 q 看成首尾相连的环，队头和队尾在环上循环移动，即可解决空间浪费的问题，这称为循环队列。

2.2.1 队列

例题　Largest Rectangle in a Histogram（POJ2559）

有 t 个小组排队，每组有若干人，当一个人来到队伍时，若队伍中有同组成员，那么

他会直接排到同组成员的后面，否则排到队尾。给出入队和出队指令，输出出队的人员编号顺序。

【输入格式】

输入包含多组测试数据，每组测试数据先输入一个整数 t 表示小组个数，接下来 t 行，每行先输入一个整数 n_i 表示第 i 组的人数，然后输入 n_i 个整数 $id_{i,1},id_{i,2},\cdots,id_{i,n_i}$ 表示第 i 组的人员编号，接下来每行一个指令，指令种类如下。

ENQUEUE x：编号为 x 的人入队

DEQUEUE：出队

STOP：结束本次测试数据

$1 \le t,n_i \le 10^3, 0 \le id_{i,j} < 10^6$，最多有 2×10^5 条指令。

【输出格式】

对于每次出队指令，输出出队人员编号。

【分析】

维护一个小组编号队列 cat 和 t 个组内人员编号队列 $team_i$。

当 x 入队时，先获取 x 的小组编号 $tag[x]$，然后检查队列 $team[tag[x]]$ 是否为空，如果为空则将 $tag[x]$ 入队 cat，最后把 x 入队 $team[tag[x]]$。

当出队时，队列 $team[cat.front()]$出队，之后如果队列 $team[cat.front()]$ 为空则队列 cat 出队。

单个操作的时间复杂度为 $O(1)$。

```
const int M = (int)1e3;
const int N = (int)1e6;

struct queue {
    int q[M + 5], l, r;
    void push(int x) {q[++r] = x;}
    void pop() {l++;}
    int front() {return q[l];}
    void clear() {l = 1, r = 0;}
    bool empty() {return l == r + 1;}
} cat, team[M + 5];

int t, tag[N + 5];

void work() {
    cat.clear();
```

```
17.     for (int i = 1, n, x; i <= t; i++) {
18.         team[i].clear();
19.         scanf("%d", &n);
20.         for (int j = 1; j <= n; j++) {
21.             scanf("%d", &x);
22.             tag[x] = i;
23.         }
24.     }
25.     char op[10]; int x;
26.     while (scanf("%s", op) && op[0] != 'S') {
27.         if (op[0] == 'E') {
28.             scanf("%d", &x);
29.             if (team[tag[x]].empty()) cat.push(tag[x]);
30.             team[tag[x]].push(x);
31.         } else {
32.             printf("%d\n", team[cat.front()].front());
33.             team[cat.front()].pop();
34.             if (team[cat.front()].empty()) cat.pop();
35.         }
36.     }
37.     printf("\n");
38. }
39.
40. int main() {
41.     int ca = 0;
42.     while (scanf("%d", &t) && t) {
43.         printf("Scenario #%d\n", ++ca);
44.         work();
45.     }
46.     return 0;
47. }
```


2.2.2 双端队列

双端队列是一种特别的队列，队头和队尾都可以进行入队和出队两种操作。

例题　双端队列（黑暗爆炸 2457）

Sherry 现在碰到了一个棘手的问题，有 n 个整数需要排序。

Sherry 手头能用的工具就是若干个双端队列。

她需要依次处理这 n 个数，对于每个数，Sherry 能做以下两件事：

1. 新建一个双端队列，并将当前数作为这个队列中的唯一的数；

2. 将当前数放入已有的队列的头之前或者尾之后。

对所有的数处理完成之后，Sherry 要求将这些双端队列按照一定顺序连接起来可以得到

一个非降序序列，她想知道最少需要几个双端队列。

【输入格式】

第一行包含一个整数 n，表示整数的个数。接下来的 n 行每行包含一个整数 a_i，其中 a_i 表示所需处理的整数。

$1 \le n \le 2 \times 10^5$。

【输出格式】

输出一个整数表示 Sherry 最少需要的双端队列数。

【分析】

我们首先要确定最终的非降序序列，记 id_i 表示最终序列的第 i 个数在原序列中的下标，可以通过 STL 中的 sort 函数以时间复杂度 $O(nlogn)$ 求得。

不难想到，一个双端队列的元素一定是由连续的 a_{id_i} 构成的。此外，假设一个双端队列中的元素为 $[a_{id_l}, a_{id_{l+1}}, \cdots, a_{id_r}]$，那么一定存在 $mid\ (l \le mid \le r)$ 满足 $id_l > id_{l+1} > \cdots > id_{mid}$ 且 $id_{mid} < id_{mid+1} < \cdots < id_r$，即 $[id_l, id_{l+1}, \cdots, id_r]$ 具有单谷性质。因此，原问题等价于使用最少的单谷划分数组 id。考虑到数组 a 中可能存在相同的数，于是可以使用尺取以 $O(n)$ 的时间复杂度把数组 id 划分成若干段，其中第 i 段为 $[id_{sl_i}, id_{sl_i+1}, \cdots, id_{sr_i}]$，段内元素可以反转。例如当 $a = [3,6,0,9,6,3]$, $id = [3,1,6,2,5,4]$ 时，数组 id 可以划分成 $[[3],[1,6],[2,5],[4]]$ 四段，之后再贪心地对段进行单谷的划分得到 $[[3,1,6],[5,2,4]]$，所以答案为 2。时间复杂度瓶颈在于排序，时间复杂度为 $O(nlogn)$。

```
scanf("%d", &n);
for (int i = 1; i <= n; i++) scanf("%d", &a[i]), id[i] = i;
sort(id + 1, id + n + 1, [&](int x, int y) {
    if (a[x] != a[y]) return a[x] < a[y];
    return x < y;
});
int sc = 0;
for (int l = 1, r = 0; l <= n; l = r + 1) {
    while (r + 1 <= n && a[id[l]] == a[id[r + 1]]) ++r;
    ++sc;
    sl[sc] = l, sr[sc] = r;
}
int ans = 0;
for (int i = 1; i <= sc; ) {
    ++ans;
    while (i + 1 <= sc && id[sr[i + 1]] < id[sl[i]]) ++i;
    ++i;
    while (i + 1 <= sc && id[sl[i + 1]] > id[sr[i]]) ++i;
    ++i;
```

```
20. }
21. printf("%d\n", ans);
```

2.2.3 单调队列

例题　滑动窗口 /【模板】单调队列（洛谷 P1886）

有一个长为 n 的序列 a，以及一个大小为 k 的窗口。现在这个从左边开始向右滑动，每次滑动一个单位，求出每次滑动后窗口中的最大值和最小值。

例如 $a=[1,3,-1,-3,5,3,6,7]$, $k=3$ 则有下表。

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

【输入格式】

输入一共有两行，第一行有两个正整数 n,k。第二行 n 个整数，表示序列 a。

【输出格式】

输出共两行，第一行为每次窗口滑动的最小值，第二行为每次窗口滑动的最大值。

【分析】

此题为单调队列模板题，其思想与单调栈的思想有一定的相同之处，即及时排除不可能的选项。由于滑动窗口最小值与滑动窗口最大值的思想完全相同，因此下面以求解滑动窗口最小值为例。

记 f_i 表示子序列 $[a_{i-k},a_{i-k+1},\cdots,a_i]$ 的最小值。当 $a_{1\cdots n}=[1,3,-1,-3,5,3,6,7]$, $k=3$ 时，由于 a_3 比前面的元素小而且 a_3 的生命周期比前面元素的长，致使 a_1 和 a_2 对 $f_{3\cdots n}$ 的计算没有任何影响，因此可以直接排除掉 a_1 和 a_2。

我们可以维护一个队列 q，存储可能的最优决策下标满足 $q_l<q_{l+1}<\cdots<q_r$，使得 $a_{q_l}<a_{q_{l+1}}<\cdots<a_{q_r}$。当计算 f_i 时，首先弹出过时决策（$i-q_l\geq k$）以满足窗口大小为 k 的限制；然后更新决策，像单调栈那样不断弹出队尾元素 q_r 直到 $a_i\geq a_{q_r}$，再将 i 入队，那么 $f_i=a_{q_l}$。

由于每个元素只入队一次，出队一次，因此时间复杂度为 $O(n)$。

```
for (int i = 1, l = 1, r = 0; i <= n; i++) {
    while (l <= r && i - q[l] >= k) l++;
    while (l <= r && a[i] < a[q[r]]) r--;
    q[++r] = i;
    if (i >= k) printf("%d%c", a[q[l]], " \n"[i == n]);
}
for (int i = 1, l = 1, r = 0; i <= n; i++) {
    while (l <= r && i - q[l] >= k) l++;
    while (l <= r && a[i] > a[q[r]]) r--;
    q[++r] = i;
    if (i >= k) printf("%d%c", a[q[l]], " \n"[i == n]);
}
```

例题　最大子序和（AcWing135）

输入一个长度为 n 的整数序列 $a_{1\cdots n}$，从中找出一段长度不超过 m 的连续子序列，使得子序列中所有数的和最大。

注意： 子序列的长度至少是 1。

【输入格式】

第一行输入两个整数 n,m。

第二行输入 n 个数，代表长度为 n 的整数序列。

$1 \le n,m \le 3 \times 10^5$。

【输出格式】

输出一个整数，代表该序列的最大子序和。

【分析】

很容易联想到前缀和，记 $s_i = \sum_{1\le j\le i} a_j$，则区间 $[l,r]$ 的和可以表示为 $s_r - s_{l-1}$。我们可以枚举 r，使用单调队列维护满足 $r-(l-1) \le m$ 的最小的 s_{l-1}的下标，答案即为 $\sum_{1\le r\le n} s[r] - s[q[r]]$。时间复杂度为 $O(n)$。

```
scanf("%d %d", &n, &m);
for (int i = 1; i <= n; i++) scanf("%lld", &s[i]), s[i] += s[i - 1];
int l = 1, r = 0;
q[++r] = 0;
ll ans = -inf;
for (int i = 1; i <= n; i++) {
    while (l <= r && i - q[l] > m) l++;
    ans = max(ans, s[i] - s[q[l]]);
    while (l <= r && s[i] <= s[q[r]]) r--;
    q[++r] = i;
}
```

```
12. printf("%lld\n", ans);
```

2.3 链表

数组是一种支持随机访问，但不支持在任意位置插入或删除的数据结构。与之相对应，链表是一种支持在任意位置插入和删除的线性数据结构，由若干个链表节点组成。链表节点除了数据域以外，还额外使用指针记录链表节点之间的先后顺序，例如普通链表的链表节点使用指针 $next$ 记录后继链表节点，双向链表的链表节点使用指针 $prev$ 记录前驱链表节点，指针 $next$ 记录后继链表节点。为了避免在左右两端或者空链表中访问越界，我们通常建立额外的两个节点 $head$ 和 $tail$ 代表链表头尾，把实际的数据节点存储在 $head$ 与 $tail$ 之间，来减少链表边界处的判断。链表的正规形式一般通过动态分配内存、指针等实现，但在算法竞赛中，为了避免内存泄漏并且从编码复杂性上考虑，常常使用数组模拟链表，下标模拟指针。

2.3.1 链表

例题　　单链表（AcWing826）

实现一个单链表，链表初始为空，支持三种操作：

1. 向链表头插入一个数；
2. 删除第 k 个插入的数后面的数；
3. 在第 k 个插入的数后插入一个数。

现在要对该链表进行 M 次操作，进行完所有操作后，从头到尾输出整个链表。

注意:题目中第 k 个插入的数并不是指当前链表的第 k 个数。例如操作过程中一共插入了 n 个数，则按照插入的时间顺序，这 n 个数依次为：第 1 个插入的数，第 2 个插入的数，… 第 n 个插入的数。

【输入格式】

第一行包含整数 M，表示操作次数。

接下来 M 行，每行包含一个操作命令，操作命令可能为以下几种：

1. H x，表示向链表头插入一个数 x。
2. D k，表示删除第 k 个插入的数后面的数（当 k为 0 时，表示删除头结点）。
3. I k x，表示在第 k 个插入的数后面插入一个数 x（此操作中 k 均大于 0 ）。

$1 \leq M \leq 10^5$，保证所有操作合法。

【输出格式】

共一行，将整个链表从头到尾输出。

【分析】

链表节点使用变量 x 记录数据域，变量 nx 记录后继链表节点的下标，详情见代码。各个操作的时间复杂度为 $O(1)$，总时间复杂度为 $O(M)$。

```
const int M = (int)1e5;

struct node {
    int x, nx;
} node[M + 5];

void work() {
    int m; scanf("%d", &m);
    int head = 0, tail = M + 1;
    node[head].nx = tail;
    char op[10];
    for (int i = 1, cnt = 0, x, k; i <= m; i++) {
        scanf("%s", op);
        if (op[0] == 'H') {
            scanf("%d", &x);
            ++cnt;
            node[cnt].x = x;
            node[cnt].nx = node[0].nx;
            node[0].nx = cnt;
        } else if (op[0] == 'D') {
            scanf("%d", &k);
            node[k].nx = node[node[k].nx].nx;
        } else if (op[0] == 'I') {
            scanf("%d %d", &k, &x);
            ++cnt;
            node[cnt].x = x;
            node[cnt].nx = node[k].nx;
            node[k].nx = cnt;
        }
    }
    int cur = head;
    while (node[cur].nx != tail) printf("%d ", node[cur = node[cur].nx].x);
    printf("\n");
}

int main() {
    work();
    return 0;
}
```

2.3.2 双向链表

例题　邻值查找（AcWing136）

给定一个长度为 n 的序列 A，A 中的数各不相同。

对于 A 中的每一个数 A_i，求 $\min_{1\le j<i}|A_i-A_j|$，以及令上式取到最小值的 j（记为 P_i）。若最小值点不唯一，则选择使 A_j 较小的那个。

【输入格式】

第一行输入整数 n，代表序列长度。

第二行输入 n 个整数 $A_1,A_2,\cdots,A_n$ 代表序列的具体数值，数值之间用空格隔开。

$n\le 10^5,|A_i|\le 10^9$。

【输出格式】

输出共 $n-1$ 行，每行输出两个整数，数值之间用空格隔开。

分别表示当 i 取 $2\sim n$ 时，对应的 $\min_{1\le j<i}|A_i-A_j|$和 P_i 的值。

【分析】

把序列 A 排好序后按照从小到大连接成一张双向链表 $node$，同时计算出序列 id 和序列 $invid$，其中 id_i 表示链表的第 i 个节点在原序列 A 中的下标，$invid_i$ 是 id_i 的逆，即 A_i 在是链表中的第几个节点。

因为链表有序，所以我们只需要比较 $node_{invid_n}$ 的前驱和后继分别与 $node_{invid_n}$ 的差的绝对值，就能求出与 A_{invid_n} 最接近的值，同时可以借助数组 id 求出 P_n。接下来将链表节点 $node_{invid_n}$ 删除，求解 P_{n-1} 的问题便成了上述的一个子问题。时间复杂度为 $O(nlog)$，瓶颈在于排序。

```
typedef long long ll;

const int M = (int)1e5;
const ll linf = 0x3f3f3f3f3f3f3f3f;

struct node {
    ll x; int pre, nx;
} node[M + 5];

int a[M + 5];
int id[M + 5], invid[M + 5];
ll mi[M + 5]; int p[M + 5];

```

```
void work() {
    int n; scanf("%d", &n);
    int head = 0, tail = n + 1;
    node[head].nx = 1, node[head].x = linf;
    node[tail].pre = n, node[tail].x = linf;
    for (int i = head + 1; i < tail; i++) node[i].pre = i - 1, node[i].nx = i + 1;
    for (int i = 1; i <= n; i++) scanf("%d", &a[i]), id[i] = i;
    sort(id + 1, id + n + 1, [&](int x, int y) {return a[x] < a[y];});
    for (int i = 1; i <= n; i++) invid[id[i]] = i, node[i].x = a[id[i]];
    for (int i = n, u; i >= 2; i--) {
        u = invid[i];
        ll pre = node[node[u].pre].x, cur = node[u].x, nx = node[node[u].nx].x;
        mi[i] = min(abs(pre - cur), abs(nx - cur));
        if (abs(pre - cur) <= abs(nx - cur)) p[i] = id[node[u].pre];
        else p[i] = id[node[u].nx];

        node[node[u].pre].nx = node[u].nx;
        node[node[u].nx].pre = node[u].pre;
    }
    for (int i = 2; i <= n; i++) printf("%lld %d\n", mi[i], p[i]);
}
```

2.3.3 循环链表

循环链表是在原有链表的基础上，将链表头节点 $head$ 和链表尾节点 $tail$ 连接，构成一个环。按照链表指针的不同，循环链表可以分为单向循环链表和双向循环链表。

2.4 ST 表

给定一个长度为 n 的序列 a，ST 算法能在 $O(nlogn)$ 时间的预处理后，以 $O(1)$ 的时间复杂度在线解决静态 RMQ 问题，即无修改操作的区间最值查询问题。

设 $f[i,j]$ 表示子序列 $a[i],a[i+1],\cdots,a[i+2^j-1]$ 的最大值，也就是从 i 开始的 2^j 个数的最大值。递推边界是 $f[i,0]=a_i$，即序列 a 在子区间 $[i,i]$ 的最大值。

在递推时，我们把子区间的长度成倍增长，有公式 $f[i,j]=\max(f[i,j-1],f[i+2^{j-1},j-1])$，即长度为 2^j 的子区间的最大值是左右两半长度为 2^{j-1} 的子区间的最大值中较大的一个。

当询问任意区间 $[l,r]$ 的最值时，我们先计算出一个 k，满足 $2^k \le r-l+1 < 2^{k+1}$，那么“从 l 开始的 2^k 个数”和“以 r 结尾的 2^k 个数”这两段一定覆盖了整个区间 $[l,r]$，这两段的最大值分别是 $f[l,k]$ 和 $f[r-2^k+1,k]$，二者中较大的那个就是整个区间 $[l,r]$ 的最值。因为求的是最大值，所以这两段只要覆盖区间 $[l,r]$ 即可，即使有重叠也没关系。

例题 【模板】ST 表（洛谷 P3865）

给定一个长度为 n 的数列，和 m 次询问，求出每一次询问的区间内数字的最大值。

【输入格式】

第一行包含两个整数 n,m ，分别表示数列的长度和询问的个数。

第二行包含 n 个整数（记为 a_i），依次表示数列的第 i 项。

接下来 m 行，每行包含两个整数 l_i,r_i，表示查询的区间为 $[l_i,r_i]$。

【输出格式】

输出包含 m 行，每行一个整数，依次表示每一次询问的结果。

【分析】

按照上述原理实现的 ST 表即可，其中可以预处理函数 $\log_2()$ 以达到 $O(1)$ 查询区间最值。时间复杂度为 $O(nlogn)$。

```
1. const int M = (int)1e5;
2. const int N = (int)16;
3.
4. int lg[M + 5];
5. int f[M + 5][N + 1];
6.
7. void init() {
8.     lg[1] = 0;
9.     for (int i = 2; i <= M; i++) lg[i] = lg[i>>1] + 1;
10. }
11.
12. int query(int l, int r) {
13.     int k = lg[r - l + 1];
14.     return max(f[l][k], f[r - (1<<k) + 1][k]);
15. }
16.
17. void work() {
18.     int n, m; scanf("%d %d", &n, &m);
19.     for (int i = 1; i <= n; i++) scanf("%d", &f[i][0]);
20.     for (int j = 1; j <= lg[n]; j++)
21.         for (int i = 1; i + (1<<j) - 1 <= n; i++)
22.             f[i][j] = max(f[i][j - 1], f[i + (1<<(j-1))][j - 1]);
23.     for (int i = 1, l, r; i <= m; i++) {
24.         scanf("%d %d", &l, &r);
25.         printf("%d\n", query(l, r));
26.     }
27. }
28.
```

```
29. int main() {
30.     init();
31.     work();
32.     return 0;
33. }
```

2.5 树

数据结构中的树和现实生活中的树长得一样，只不过我们习惯于处理问题的时候把树根放到上方来考虑。这种数据结构看起来像是一个倒挂的树，因此得名。

2.5.1 树的基本概念

树的定义

一个没有固定根结点的树称为无根树。无根树有几种等价的形式化定义：

1. 有 n 个结点，$n-1$ 条边的连通无向图
2. 无向无环的连通图
3. 任意两个结点之间有且仅有一条简单路径的无向图
4. 任何边均为桥的连通图
5. 没有圈，且在任意不同两点间添加一条边之后所得图含唯一的一个圈的图

在无根树的基础上，指定一个结点称为根，则形成一棵有根树。

适用于无根树和有根树

森林：每个连通分量（连通块）都是树的图。按照定义，一棵树也是森林。

生成树：一个连通无向图的生成子图，同时要求是树。也即在图的边集中选择 $n-1$ 条，将所有顶点连通。

无根树的叶结点：度数不超过 1 的结点。

有根树的叶结点：没有子结点的结点。

只适用于有根树

父亲：对于除根以外的每个结点，定义为从该结点到根路径上的第二个结点。根结点没有父结点。

祖先：一个结点到根结点的路径上，除了它本身外的结点。根结点的祖先集合为空。

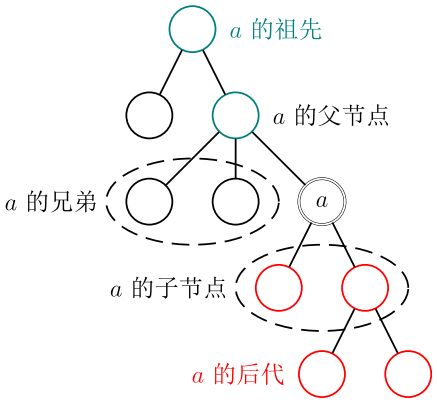
结点：如果 u 是 v 的父亲，那么 v 是 u 的子结点。子结点的顺序一般不加以区分，二叉树是一个例外。

结点的深度：到根结点的路径上的边数。

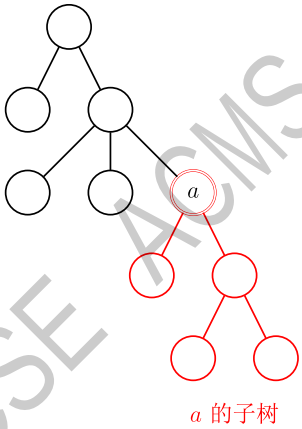
树的高度：所有结点的深度的最大值。

兄弟：同一个父亲的多个子结点互为兄弟。

后代：子结点和子结点的后代。或者理解成：如果 u 是 v 的祖先，那么 v 是 u 的后代。



子树：删掉与父亲相连的边后，该结点所在的子图。



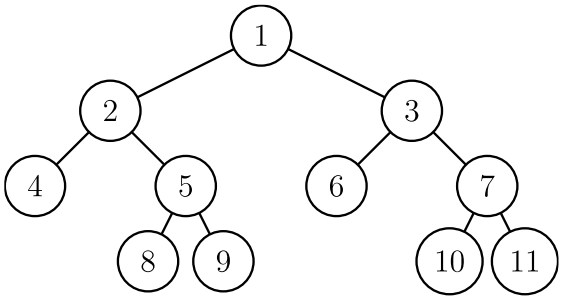
特殊的树

链：满足与任一结点相连的边不超过 2 条的树称为链。

菊花/星星：满足存在 u 使得所有除 u 以外结点均与 v 相连的树称为菊花。

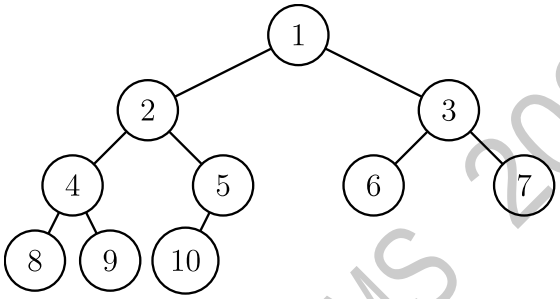
有根二叉树：每个结点最多只有两个儿子（子结点）的有根树称为二叉树。常常对两个子结点的顺序加以区分，分别称之为左子结点和右子结点。大多数情况下，二叉树一词均指有根二叉树。

完整二叉树：每个结点的子结点数量均为 0 或者 2 的二叉树。换言之，每个结点或者是树叶，或者左右子树均非空。



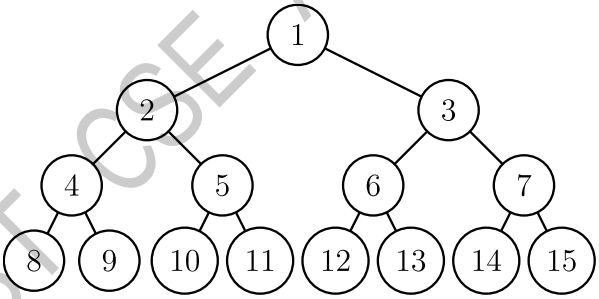
完整二叉树（proper binary tree）

完全二叉树：只有最下面两层结点的度数可以小于 2，且最下面一层的结点都集中在该层最左边的连续位置上。



完全二叉树（complete binary tree）

完美二叉树：所有叶结点的深度均相同的二叉树称为完美二叉树。



完美二叉树（perfect binary tree）

2.5.2 树的存储与遍历

无根树的存储与遍历

无根树一般用 n 个点和 $n-1$ 条形如 (u,v) 的无向边表示，我们可以使用 STL 中的 vector 存边。遍历时按照深度优先顺序遍历，，这个过程中最需要注意的是避免重复访问结点。我们可以记录当前节点是由哪个节点访问而来的（如 fa），此后进入除 fa 节点外的所有相邻节点，即可避免重复访问。

```
1. vector<int> g[M + 5];
2.
3. void dfs(int u, int fa) {
```

```
    for (const int &v: g[u]) if (v != fa) {
        dfs(v, u);
    }
}

for (int i = 2, u, v; i <= n; i++) {
    scanf("%d %d", &u, &v);
    g[u].push_back(v), g[v].push_back(u);
}
dfs(1, 0);
```

有根树的存储与遍历

有根树一般用 n 个点和 $n-1$ 条形如 $<fa,u>$ 的有向边表示，我们可以使用 STL 中的 vector 存边。遍历时按照深度优先顺序遍历，由于是有向边所以不存在重复访问的问题。

```
vector<int> g[M + 5];

void dfs(int u) {
    for (const int &v: g[u]) {
        dfs(v);
    }
}

for (int i = 2, fa; i <= n; i++) {
    scanf("%d %d", &fa, &i);
    g[fa].push_back(i);
}
dfs(1);
```

2.5.3 树的 dfs 序

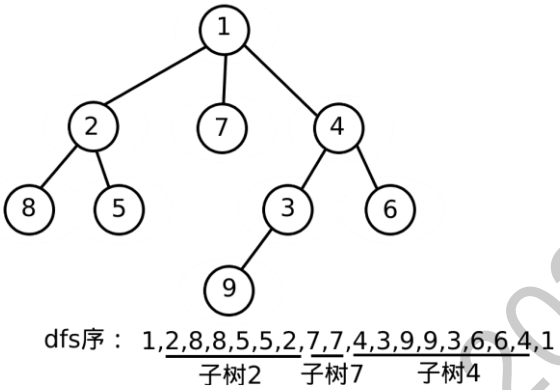
一般来讲，我们在对树进行深度优先遍历时，对于每个节点，在刚进入递归后以及即将回溯前个记录一次该点的编号，最后产生的长度为 $2n$ 的节点序列就称为树的 dfs 序。下面以有根树为例。

```
int dfn[M + 5], m; //  dfs 序及其长度

void dfs(int u) {
    dfn[++m] = u;
    for (const int &v: g[u]) {
        dfs(v);
    }
    dfn[++m] = u;
```

```
9. }
```

dfs 序的特点是：每个节点 x 的编号在序列中恰好出现两次。设这两次出现的位置为 $L[u]$ 与 $R[u]$，那么闭区间 $[L[u],R[u]]$ 就是以 x 为根的子树的 dfs 序。这使我们在很多与树相关的问题中，可以通过 dfs 序把子树统计转化为序列上的区间统计，如使用树状数组或线段树维护子树信息和树链剖分维护树链信息等问题，我们在后续的章节中会涉及到。

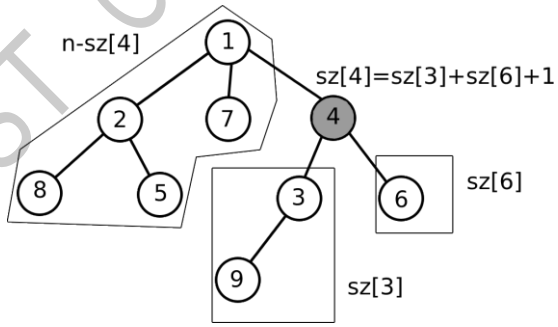


另外，二叉树的先序、中序与后序遍历序列，也是通过深度优先遍历产生的，大多数程序设计入门级的书籍上都有详细讲解，在此就不在赘述。

2.5.4 树的重心

树的重心定义

记 $sz[u]$ 表示子树 u 的大小，对于叶子节点，其大小为 1；若节点 u 有 k 个子节点 $v_1,v_2,\cdots,v_k$，则以 x 为根的子树的大小就是 $\sum_{1\le i\le k} sz[v_k]$。



对于树上的一个点 u，如果我们把它从树中删除，那么原来的一棵树可能会分成若干个不相连的部分，其中每一部分都是一颗子树。设 mx_u 表示在删除节点 u 后产生的子树中，最大的一颗的大小。使 mx_u 取到最小值的节点 p 就称为整棵树的重心。例如上图中的树的重心是节点 1。

树的重心性质

以树的重心为根时，所有子树的大小都不超过整棵树大小的一半。

树中所有点到某个点的距离和中，到重心的距离和是最小的；如果有两个重心，那么到

它们的距离和一样。

把两棵树通过一条边相连得到一棵新的树，那么新的树的重心在连接原来两棵树的重心的路径上。

在一棵树上添加或删除一个叶子，那么它的重心最多只移动一条边的距离。

例题 Balancing Act（POJ1655）

给定一颗 n 个点的无根树，求树的重心划分连通块的最大大小及其重心节点编号，若重心节点不止一个则输出编号最小的。

【输入格式】

第一行输入一个整数 T 表示测试数据组数，每组测试数据第一行输入一个整数 n 表示树的大小，接下来 $n-1$ 行，每行输入两个整数 u,v 表示一条边。

$1 \le T \le 20, 1 \le n \le 2 \times 10^4, 1 \le u,v \le n$，保证输入是一颗树。

【输出格式】

对于每组测试数据，输出两个整数表示重心

【分析】

假设节点 u 有 k 个儿子 $v_{1\cdots k}$，那么当节点 u 删去后，树被划分成了若干个连通块，分别是子树 v_1、子树 v_2、…、子树 v_k 和除子树 u 以外的部分。我们可以对每个节点计算出当其删除后，最大连通块的大小，然后取最小值便得到了答案。

```
const int M = (int)2e4;
const int inf = 0x3f3f3f3f;

int n;
int sz[M + 5];
vector<int> g[M + 5];
int mi, id;

void dfs(int u, int fa) {
    sz[u] = 1;
    int mx = 0;
    for (size_t i = 0; i < g[u].size(); i++) {
        int v = g[u][i];
        if (v == fa) continue;
        dfs(v, u);
        sz[u] += sz[v];
        mx = max(mx, sz[v]);
    }
    mx = max(mx, n - sz[u]);
```

```
20.     if (mx < mi || (mx == mi && u < id)) {
21.         mi = mx;
22.         id = u;
23.     }
24. }
25.
26. void work() {
27.     scanf("%d", &n);
28.     for (int i = 1; i <= n; i++) g[i].clear();
29.     for (int i = 2, u, v; i <= n; i++) {
30.         scanf("%d %d", &u, &v);
31.         g[u].push_back(v), g[v].push_back(u);
32.     }
33.     mi = inf;
34.     dfs(1, 0);
35.     printf("%d %d\n", id, mi);
36. }
```

2.5.5 树的直径

树上任意两节点之间最长的简单路径即为树的直径。

例题　Longest path in a tree（SPOJPT07Z）

给定一颗大小为 n 的无根树，求树的直径长度。

【输入格式】

第一行输入一个整数 n 表示树的大小。

接下来 $n-1$ 行，每行输入两个整数 (u,v) 表示一条边。

$1 \le n \le 10^4, 1 \le u,v \le n$。

【输出格式】

输出一个整数表示树的直径长度。

【分析】

首先从任意节点（例如 1）开始进行第一次 dfs，到达距离其最远的节点，记为 u，然后再从 u 开始做第二次 dfs，到达距离 u 最远的节点，记为 v，则 $dis(u,v)$ 即为树的直径。

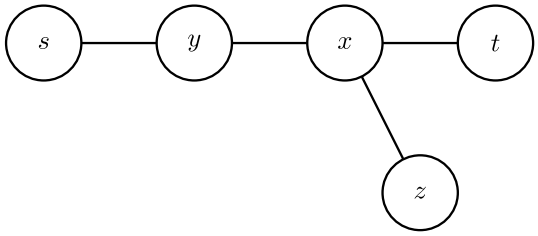
显然，如果第一次 dfs 到达的节点 u 是直径的一端，那么第二次 dfs 到达的节点 v 一定是直径的一端。我们只需证明在任意情况下，u 必为直径的一端。

定理：在一棵树上，从任意节点 y 开始进行一次 dfs，到达的距离其最远的节点 z 必为直径的一端。

证明：

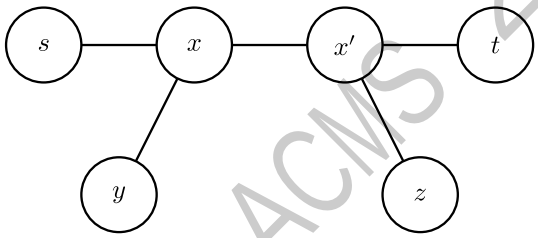
使用反证法。记出发节点为 y。设真实的直径是 $dis(s,t)$，而从 y 进行的第一次 DFS 到达的距离其最远的节点 z 不为 t 或 s。共分三种情况：

1. 若 y 在 $dis(s,t)$ 上：



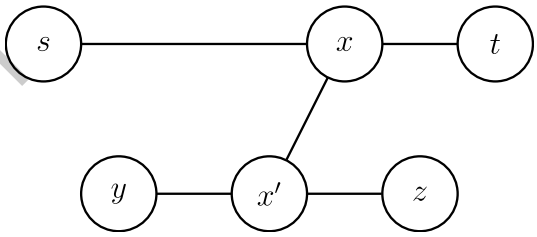
有 $dis(y,z) > dis(y,t) \Rightarrow dis(x,z) > dis(x,t) \Rightarrow dis(s,z) > dis(s,t)$ 与 $dis(s,t)$ 为树上任意两节点之间最长的简单路径矛盾。

2. 若 y 不在 $dis(s,t)$ 上，且 $dis(y,z)$ 与 $dis(s,t)$ 存在重合路径：



有 $dis(y,z) > dis(y,t) \Rightarrow dis(x,z) > dis(x,t) \Rightarrow dis(s,z) > dis(s,t)$，与 $dis(s,t)$ 为树上任意两节点之间最长的简单路径矛盾。

3. 若 y 不在 $dis(s,t)$ 上，且 $dis(y,z)$ 与 $dis(s,t)$ 不存在重合路径：



有 $dis(y,z) > dis(y,t) \Rightarrow dis(x',z) > dis(x',t) \Rightarrow dis(x,z) > dis(x,t)$

$\Rightarrow dis(s,z) > dis(s,t)$，与 $dis(s,t)$ 为树上任意两节点之间最长的简单路径矛盾。

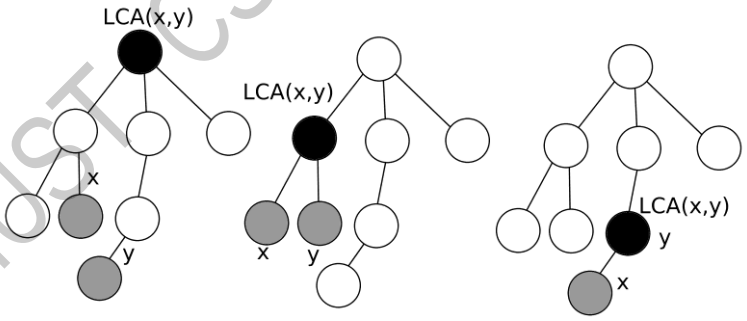
综上，三种情况下假设均会产生矛盾，故原定理得证。

```
1. const int M = (int)1e4;
2.
3. vector<int> g[M + 5];
4. int d[M + 5];
5.
6. void dfs(int u, int fa) {
7.     for (const int &v: g[u]) if (v != fa) {
```

```
 8.         d[v] = d[u] + 1;
 9.         dfs(v, u);
10.     }
11. }
12.
13. void work() {
14.     int n; scanf("%d", &n);
15.     for (int i = 2, u, v; i <= n; i++) {
16.         scanf("%d %d", &u, &v);
17.         g[u].push_back(v), g[v].push_back(u);
18.     }
19.     d[1] = 0; dfs(1, 0);
20.     int mx = -1, u, v;
21.     for (int i = 1; i <= n; i++) if (d[i] > mx) mx = d[i], u = i;
22.     d[u] = 0; dfs(u, 0);
23.     mx = -1;
24.     for (int i = 1; i <= n; i++) if (d[i] > mx) mx = d[i], v = i;
25.     printf("%d\n", mx);
26. }
```

2.5.6 最近公共祖先

给定一颗有根树，若节点 z 既是节点 x 的祖先，也节点 y 的祖先，则称 z 是 x,y 的公共祖先。在 x,y 的所有公共祖先中，深度最大的一个称为 x,y 的最近公共祖先，记为 LCA(x,y)。



LCA(x,y) 是 x 到根的路径与 y 到根的路径的交会点。它也是 x 与 y 之间的路径上深度最小的节点。求最近公共祖先的方法通常有三种：

向上标记法

从 x 向上走到根节点，并标记所有经过的节点。

从 y 向上走到根节点，当第一次遇到已标记过的节点时，就找到了 LCA(x,y)。

对于每个询问，向上标记法的时间复杂度最坏为 $O(n)$。

树上倍增法

设 $f[x,k]$ 表示 x 的 2^k 辈祖先，即从 x 向根节点走 2^k 步到达的节点。特别地，若该节点不存在，则令 $f[x,k]=0$。$f[x,0]$ 就是 x 的父节点。对于 $k\geq 1$，则有 $f[x,k]=f[f[x,k-1],k-1]$。

以上部分是预处理，时间复杂度为 $O(nlogn)$，之后可以多次对不同的 x,y 计算 LCA，每次询问的时间复杂度为 $O(logn)$。

基于 f 数组计算 LCA(x,y) 分为以下几步：

1. 设 $d[x]$ 表示 x 的深度。不妨设 $d[x]\geq d[y]$（否则可以交换 x,y）。
2. 用二进制拆分思想，把 x 向上调整到与 y 同一深度。
3. 若此时 $x=y$，说明已经找到了 LCA 且 LCA$(x,y)=y$。
4. 否则用二进制拆分思想，把 x,y 尽力同时向上调整，并保持深度一致且二者不相遇。
5. 此时 x,y 必定只差一步就相遇了，它们的父节点 $f[x,0]$ 就是 LCA。

LCA 的 Tarjan 算法

Tarjab 算法本质上是一个使用并查集对向上标记法的优化，此处暂不做讲解。

例题 【模板】最近公共祖先（LCA）（洛谷 3379）

给定一棵有根多叉树，请求出指定两个点直接最近的公共祖先。

【输入格式】

第一行包含三个正整数 n,m,s，分别表示树的结点个数、询问的个数和树根结点的序号。

接下来 $n-1$ 行每行包含两个正整数 x,y，表示 x 结点和 y 结点之间有一条直接连接的边（数据保证可以构成树）。

接下来 m 行每行包含两个正整数 a,b，表示询问 a 结点和 b 结点的最近公共祖先。

$1\leq n,m\leq 5\times 10^5, 1\leq x,y,a,b\leq n$。

【输出格式】

输出包含 m 行，每行包含一个正整数，依次为每一个询问的结果。

【分析】

实现前面所述的树上倍增法即可，时间复杂度为 $O((n+m)logn)$。

```
1. const int M = (int)5e5;
2. const int N = 18;
3.
4. int d[M + 5];
```

```
int f[M + 5][N + 1];
vector<int> g[M + 5];

void dfs(int u, int fa) {
    for (const int &v: g[u]) if (v != fa) {
        d[v] = d[u] + 1;
        f[v][0] = u;
        for (int i = 1; i <= N; i++) f[v][i] = f[f[v][i - 1]][i - 1];
        dfs(v, u);
    }
}

int lca(int x, int y) {
    if (d[x] < d[y]) swap(x, y);
    for (int i = N; i >= 0; i--) if (d[f[x][i]] >= d[y]) x = f[x][i];
    if (x == y) return x;
    for (int i = N; i >= 0; i--) if (f[x][i] != f[y][i]) x = f[x][i], y = f[y][i];
    return f[x][0];
}

void work() {
    int n, m, s; scanf("%d %d %d", &n, &m, &s);
    for (int i = 2, u, v; i <= n; i++) {
        scanf("%d %d", &u, &v);
        g[u].push_back(v), g[v].push_back(u);
    }
    d[s] = 1; dfs(s, 0);
    for (int i = 1, a, b; i <= m; i++) {
        scanf("%d %d", &a, &b);
        printf("%d\n", lca(a, b));
    }
}
```

例题　How far away ？（HDU2586）

给定一颗大小为 n 的无根树，m 次询问，每次询问节点 u 与节点 v 的距离。

【输入格式】

输入包含多组测试数据，每组测试数据第一行输入两个整数 n 和 m，接下来 $n-1$ 行，每行输入三个整数 u,v 和 w 表示无向边 (u,v) 的长度为 w，接下来 m 行，每行两个整数 u 和 v，表示询问节点 u 与节点 v 的距离。

$1 \le T \le 10, 2 \le n \le 4 \times 10^4, 1 \le m \le 200, 1 \le u,v \le n, 1 \le w \le 4 \times 10^4$。

【输出格式】

对于每次询问输出一个整数表示两点间距离。

【分析】

容易发现 $dis(u,v)=dis(u,root)+dis(v,root)-2dis(lca(u,v),root)$，因此我们只需要求出 LCA 并记录根到每个点的距离即可，时间复杂度为 $O((n+m)logn)$。

```
const int M = (int)4e4;
const int N = 15;

int d[M + 5], dis[M + 5];
int f[M + 5][N + 1];
vector<pair<int, int>> g[M + 5];

void dfs(int u, int fa) {
    for (const auto p: g[u]) {
        int v = p.first, w = p.second;
        if (v == fa) continue;
        d[v] = d[u] + 1;
        dis[v] = dis[u] + w;
        f[v][0] = u;
        for (int i = 1; i <= N; i++) f[v][i] = f[f[v][i - 1]][i - 1];
        dfs(v, u);
    }
}

int lca(int x, int y) {
    if (d[x] < d[y]) swap(x, y);
    for (int i = N; i >= 0; i--) if (d[f[x][i]] >= d[y]) x = f[x][i];
    if (x == y) return x;
    for (int i = N; i >= 0; i--) if (f[x][i] != f[y][i]) x = f[x][i], y = f[y][i];
    return f[x][0];
}

void work() {
    int n, m; scanf("%d %d", &n, &m);
    for (int i = 1; i <= n; i++) g[i].clear();
    for (int i = 2, u, v, w; i <= n; i++) {
        scanf("%d %d %d", &u, &v, &w);
        g[u].push_back({v, w}), g[v].push_back({u, w});
    }
    d[1] = 1; dfs(1, 0);
    for (int i = 1, a, b; i <= m; i++) {
        scanf("%d %d", &a, &b);
        int c = lca(a, b);
```

```
39.         printf("%d\n", dis[a] + dis[b] - 2 * dis[c]);
40.     }
41. }
```

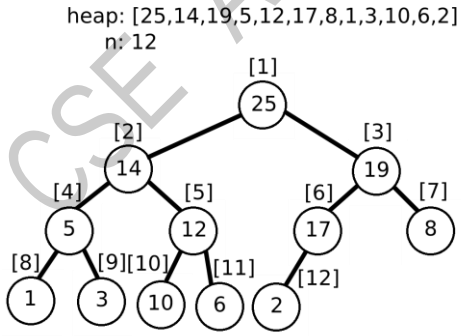
2.6 二叉堆

二叉堆是一种支持插入、删除、查询最值的数据结构。

2.6.1 二叉堆

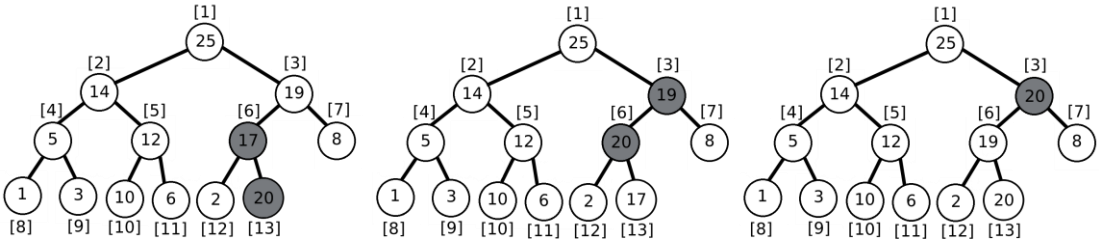
二叉堆是一种支持插入、删除、查询最值的数据结构。它其实是一颗满足“堆性质”的完全二叉树，树上的每个节点带有一个权值。若树中的任意一个父节点的权值都大于等于其子节点的权值，则称该二叉树满足“大根堆的性质”。若树中的任意一个父节点的权值都小于等于其子节点的权值，则称该二叉树满足“小根堆的性质”。

根据完全二叉树的性质，我们可以采用层次序列存储方式，直接用一个数组来保存二叉堆。层次序列存储方式，是逐层从左到右为树中的节点依次编号，把此编号作为节点在数组中的下标。在这种存储方式中，父节点编号等于子结点编号除以 2，左子节点编号等于父节点编号乘 2，右子节点编号等于父节点编号乘 2 加 1，如图是一个大根堆。接下来我们以大根堆为例介绍堆支持的几种常见操作的实现。



push

push(val) 操作向二叉堆中插入一个带有权值 val 的新节点。我们把这个新节点直接放在存储二叉堆的数组末尾，然后通过交换的方式向上调整，直至满足堆性质。其时间复杂度为堆的深度，即 $O(logn)$。

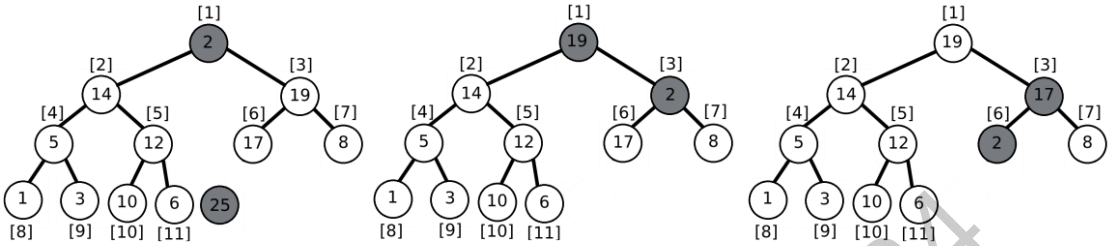


top

top 操作返回二叉堆的堆顶权值，即最大值 $heap[1]$，复杂度为 $O(1)$。

pop

pop 操作把堆顶从二叉堆中移除。我们把堆顶 $heap[1]$ 与存储在数组末尾的节点 $heap[n]$ 交换，然后溢出数组末尾节点（令 n 减小 1）最后把堆顶通过交换的方式向下调整，直至满足堆性质，其时间复杂度为堆的深度，即 $O(logn)$。



remove

remove(p) 操作把存储在数组下标 p 位置的节点从二叉堆中删除。与 Extract 相类似，我们先把 $heap[p]$ 与 $heap[n]$ 交换，然后令 n 减小 1.注意此时 $heap[p]$ 既有可能向下调整，也有可能向上调整，需要分别进行检查和处理。时间复杂度为 $O(logn)$。

例题　【模板】堆（洛谷 3378）

给定一个数列，初始为空，请支持下面三种操作：

1. 给定一个整数 x，请将 x 加入到数列中。
2. 输出数列中最小的数。
3. 删除数列中最小的数（如果有多个数最小，只删除 1 个）。

【输入格式】

第一行是一个整数，表示操作的次数 n。

接下来 n 行，每行表示一次操作。每行首先有一个整数 op 表示操作类型。

若 $op = 1$，则后面有一个整数 x，表示要将 x 加入数列。

若 $op = 2$，则表示要求输出数列中的最小数。

若 $op = 3$，则表示删除数列中的最小数。如果有多个数最小，只删除 1 个。

$1 \le n \le 10^6, 1 \le x < 2^{31}, op \in \{1,2,3\}$。

【输出格式】

对于每个操作 2，输出一行一个整数表示答案。

【分析】

我们可以实现一个小根堆来支持题目中的三种操作，或者按照前面所述的方法实现一个大根堆，堆内元素存储 $-x$，也可以达到相同的效果。时间复杂度为 $O(nlogn)$。

```
const int M = (int)1e6;

struct heap {
    int a[M + 5], n;
    void clear() {n = 0;}
    int size() {return n;}
    void up(int p) { //  向上调整
        while (p > 1) {
            if (a[p] > a[p>>1]) { //  子节点大于父节点，不满足大根堆性质
                swap(a[p], a[p>>1]);
                p >>= 1;
            } else break;
        }
    }
    void down(int p) { //  向下调整
        int s = p << 1;
        while (s <= n) {
            if (s < n && a[s] < a[s + 1]) s++; //  左右子节点中取较大者
            if (a[s] > a[p]) { //  子节点大于父节点，不满足大根堆性质
                swap(a[s], a[p]);
                p = s, s <<= 1;
            } else break;
        }
    }
    void push(int v) {
        a[++n] = v;
        up(n);
    }
    int top() {
        return a[1];
    }
    void pop() {
        a[1] = a[n--];
        down(1);
    }
    void remove(int k) {
        a[k] = a[n--];
        up(k), down(k);
    }
} h;

void work() {
    int n; scanf("%d", &n);
    for (int i = 1, op, x; i <= n; i++) {
```

```
        scanf("%d", &op);
        if (op == 1) {
            scanf("%d", &x);
            h.push(-x);
        } else if (op == 2) {
            printf("%d\n", -h.top());
        } else h.pop();
    }
}
```

例题 Supermarket（POJ1456）

给定 n 个商品，每个商品有利润 p_i 和过期时间 d_i，每天只能卖一个商品，过期的商品不能再卖，求如何安排每天卖的商品可以使收益最大。

【输入格式】

输入包含多组测试数据，每组测试数据先输入一个整数 n，接下来输入 n 对数表示 p_i 和 d_i。

$0 \leq n, p_i, d_i \leq 10^4$。

【输出格式】

对于每组测试数据，输出一个整数表示最大收益。

【分析】

容易想到一个贪心策略：在最优解中，对于每个时间（天数）t，应该在保证不卖出过期商品的前提下，尽量卖出利润前 t 大的商品。因此，我们可以依次考虑每个商品，动态维护一个满足上述性质的方案。

详细地说，我们把商品按照过期时间从小到大排序，建立一个初始为空的小根堆，堆节点权值为商品利润，然后依次扫描每个商品：

1. 若当前商品的过期时间 t 等于当前堆中的商品个数，则说明在目前方案下，前 t 天已经安排了 t 个商品卖出。此时，若当前商品的利润大于堆顶权值，则替换掉堆顶，即用当前商品替换掉原方案中利润最低的商品。
2. 若当前商品的过期时间 t 大于当前堆中的商品个数，直接把该商品插入堆。

最终，堆里的所有商品就是我们需要卖出的商品，它们的利润之和就是答案。时间复杂度为 $O(nlogn)$。

```
const int M = (int)1e4;

int n;
heap h;
```

```
int p[M + 5], d[M + 5], id[M + 5];

bool cmp(int a, int b) {
    return d[a] < d[b];
}

void work() {
    for (int i = 1; i <= n; i++) scanf("%d %d", &p[i], &d[i]), id[i] = i;
    sort(id + 1, id + n + 1, cmp);
    for (int i = 1, u; i <= n; i++) {
        u = id[i];
        if (h.size() < d[u]) h.push(-p[u]);
        else if (p[u] > -h.top()) {
            h.pop();
            h.push(-p[u]);
        }
    }
    int ans = 0;
    while (h.size()) {
        ans += -h.top();
        h.pop();
    }
    printf("%d\n", ans);
}

int main() {
    while (~scanf("%d", &n)) work();
    return 0;
}
```

2.6.2 对顶堆

例题　Running Median（POJ3784）

给定 n 个整数 $a_{1\cdots n}$，对于每个奇数 $i\,(1 \le i \le n)$，求出前缀 $a_{1\cdots i}$ 的中位数。

【输入格式】

第一行输入一个整数 T 表示测试数据组数。每组测试数据第一行输入测试数据组号和一个奇数 n 表示序列长度，第二行输入 n 个整数 $a_{1\cdots n}$。

$1 \le T \le 10^3, 1 \le n < 10^4$，$a_i$ 保证在 32 位有符号 int 类型的范围内。

【输出格式】

对于每组测试数据，第一行输出测试数据组好和中位数的个数，接下来输出中位数，每行最多 10 个。

【分析】

我们可以维护一个大根堆 $left$ 和一个小根堆 $right$，大根堆 $left$ 存储的是前缀 $a_{1\cdots i}$ 中前 $\lfloor i/2 \rfloor$ 小的数，小根堆 $right$ 存储前缀 $a_{1\cdots i}$ 剩余的部分，可以发现大根堆 $left$ 与小根堆 $right$ 共同存储了前缀 i。如此一来，中位数为 $right.top()$。时间复杂度为 $O(nlogn)$。

由于大根堆与小根堆上下对顶相连，形状类似沙漏，因此这种数据结构被称为对顶堆。

```
heap left, right;

void work() {
    left.clear(), right.clear();
    int ca, n; scanf("%d %d", &ca, &n);
    printf("%d %d\n", ca, (n + 1) / 2);
    for (int i = 1, x; i <= n; i++) {
        scanf("%d", &x);
        if (left.size() == 0 || x > left.top()) right.push(-x);
        else left.push(x);
        while (right.size() > left.size() + 1) {
            left.push(-right.top());
            right.pop();
        }
        while (right.size() < left.size()) {
            right.push(-left.top());
            left.pop();
        }
        if (i % 20 == 0) printf("\n");
        if (i & 1) printf("%d ", -right.top());
    }
    printf("\n");
}
```


2.7 STL

本节介绍 STL 中 vector，queue，priority_queue，deque，set，multiset， map 和 bitset 在算法竞赛中比较常用的容器。另外，我们也会介绍 algorithm 头文件中包含的部分函数。对于更详细的介绍，读者可参照 cplusplus.com。

2.7.1 #include <vector>

vector 可以理解为变长数组，它的内部实现基于倍增思想。按照下列思路可以大致实现一个 vector：设 n,m 为 vector 的实际长度和最大长度。向 vector 加入元素前，若 $n=m$，则在内存中申请 $2m$ 的连续空间，并把内容转移到新的地址上，同时释放旧的空间，再执

行插入。从 vector 中删除元素后，若 $n \leq m/4$，则释放一半空间。

vector 支持随机访问，即对于任意的下标 $0 \leq i < n$，可以像数组一样用 $[i]$ 取值。但它不是链表，不支持在任意位置 $O(1)$ 插入。为了保证效率，元素的删增一般应该在末尾进行。

声明

```
#include <vector>
vector<int> a; //  相当于一个长度动态变化的 int 数组
vector<int> b[233]; //  相当于第一维长 233，第二维长度动态变化的 int 数组
vector<int> c(233, 0); //  相当于一个长度动态变化的 int 数组，初始长度为 233，初始元素均为 0
struct rec{...};
vector<rec> c; //  自定义的结构体类型也可以保存在 vector 中
```

size/empty

size 函数返回 vector 的实际长度，empty 函数返回一个 bool 类型，表明 vector 是否为空。二者的时间复杂度都是 $O(1)$。

所有的 STL 容器都支持这两个方法，含义也都相同，之后我们就不再赘述。

clear

clear 函数把 vector 清空。

迭代器

迭代器就像 STL 容器的“指针”，可以用星号“*”操作符解除引用。

一个保存 int 类型的 vector 迭代器声明方法为：

```
vector<int>::iterator it;
```

vector 的迭代器是“随机访问迭代器”，可以把 vector 的迭代器与一个整数相加相减，其行为和指针的移动类似。可以把 vector 的两个迭代器相减，其结果也和指针相加减类似，得到两个迭代器相应下标之间的距离。

begin/end

begin 函数返回指向 vector 中第一个元素的迭代器，例如 a 是一个非空的 vector，则 a.begin() 与 $a[0]$ 的作用相同。

所有容器都可以视作一个“前闭后开”的结构，end 函数返回 vector 的尾部，即第 n 个元素再往后的“边界”。*a.end()与 $a[n]$ 都是越界访问，其中 $n = a$.size()。

下面两份代码都遍历了 vector<int> a，并输出它的所有元素。

```
for (size_t i = 0; i < a.size(); i++)
    cout << a[i] << endl;
```

```
3. for (vector<int>::iterator it = a.begin(), it != a.end(); it++)
4.     cout << *it << endl;
5. for (const auto& x: a)
6.     cout << x << endl;
```

front/back

front 函数返回 vector 的第一个元素，等价于 $*a.begin()$ 和 $a[0]$。

back 函数返回 vector 的最后一个元素，等价于 *--a.end() 和 $a[a.\text{size}()-1]$。

push_back/pop_back

a.push_back(x) 把元素 x 插入到 vector a 的尾部。

a.pop_back() 删除 vector a 的最后一个元素。

2.7.2 #include <queue>

头文件 queue 主要包括循环队列 queue 和优先队列 priority_queue 两个容器。

声明

```
1. queue<int> q;
2. struct rec{...}; queue<rec> q;
3. priority_queue<int> q;
4. /*
5. pair<>: C++内置的二元组，尖括号中分别指定二元组的第一元、第二元的类型。
6. 可以用 make_pair 函数创建二元组，用成员变量 first 访问第一元，second 访问第二元。
7. 在比较大小时，以第一元作为第一关键字，第二元作为第二关键字。
8. */
9. priority_queue<pair<int, int>> q;
```

循环队列 queue

方法	描述	示例	时间复杂度
push	入队（从队尾）	q.push(element);	$O(1)$
pop	出队（从队头）	q.pop();	$O(1)$
front	队头元素	int x = q.front();	$O(1)$
back	队尾元素	int y = q.back();	$O(1)$

优先队列 priority_queue

priority_queue 可理解为一个大根二叉堆。

方法	描述	示例	时间复杂度
push	把元素插入堆	q.push(element);	$O(logn)$
pop	删除堆顶元素	q.pop();	$O(logn)$

top	查询堆顶元素	int x = q.top();	$O(1)$

重载“<”运算符

priority_queue 实现小根二叉堆的方式一般有两种。

对于 int 等内置数值类型，可以把要插入元素的相反数放入堆中。等从堆中取出元素时，再把它取相反数变回原来的元素。这样就相当于把小的放在堆顶。

更为通用的解决方案是，建立自定义结构体类型，重载“小于号”，但是当作“大于号”来编写函数。

2.7.3 #include <deque>

双端队列 deque 是一个支持在两端高效插入或删除元素的连续线性存储空间。它就像是 vector 和 queue 的结合。与 vector 相比，deque 在头部增删元素仅需要 $O(1)$ 的时间；与 queue 相比，deque 像数组一样支持随机访问。

方法	描述	示例	时间复杂度
[]	随机访问	与 vector 类似	$O(1)$
begin/end	deque 的头/尾迭代器	与 vector 迭代器类似	$O(1)$
front/back	队头/队尾元素	与 queue 类似	$O(1)$
push_back	从队尾入队	q.push_back(x);	$O(1)$
push_front	从队头入队	q.push_front(y);	$O(1)$
pop_front	从队头出队	q.pop_front();	$O(1)$
pop_back	从队尾出队	q.pop_back();	$O(1)$
clear	清空队列	q.clear();	$O(n)$

2.7.4 #include <set>

头文件 set 主要包括 set 和 multiset 两个容器，分别是“有序集合”和“有序多重集”，即前者的元素不能重复，而后者可以包含若干个相等的元素。set 和 multiset 的内部实现时一颗红黑树，它们支持的函数基本相同。

声明

```
1. set<int> s;
2. struct rec{...}; set<rec> s;
3. multiset<double> s;
```

set 和 multiset 存储的元素必须定义“小于号”运算符。

size/empty/clear

与 vector 类似，分别为元素个数、是否为空、清空。前两者的时间复杂度为 $O(1)$。

迭代器

set 和 multiset 的迭代器称为“双向访问迭代器”，不支持随机访问，支持星号解除引用，仅支持“++”和“--”两个算数操作。

设 it 是一个迭代器，例如 set<int>::iterator it;

若把 it++，则 it 将会指向下一个元素，这里的下一个元素是指元素从小到大排序的结果中，排在 it 下一名的元素。同理，若把 it--，则 it 将会指向排在上一个的元素。二者时间复杂度都为 $O(logn)$。

begin/end

返回集合的首、尾迭代器，时间复杂度为 $O(1)$。

s.begin() 是指集合中最小元素的迭代器。

s.end() 是指向集合中最大元素的下一个位置的迭代器。换言之，就像 vector 一样，是一个前闭后开的形式。因此 --s.end() 是指向集合中最大元素的迭代器。

insert

s.insert(x) 把一个元素 x 插入到集合 s 中，时间复杂度为 $O(logn)$。

find

s.find(x) 在集合 s 中查找等于 x 的元素，并返回指向该元素的迭代器。若不存在则返回 s.end()。时间复杂度为 $O(logn)$。

lower_bound/upper_bound

s.lower_bound(x) 查找 $\geq x$ 的元素中最小的一个，并返回指向该元素的迭代器。

s.upper_bound(x) 查找 $> x$ 的元素中最小的一个，并返回指向该元素的迭代器。

二者时间复杂度都为 $O(logn)$。

erase

设 it 是一个迭代器，s.erase(it) 从 s 中删除迭代器 it 指向的元素，时间复杂度为 $O(logn)$。

设 x 是一个元素，s.erase(x) 从 s 中删除所有等于 x 的元素，时间复杂度为 $O(k+logn)$，其中 k 为被删除的元素个数。

如果想从 multiset 中删掉至多 1 个等于 x 的元素，可以执行：

```
1. if ((it = s.find(x)) != s.end()) s.erase(it);
```

count

s.count(x) 返回集合 s 中等于 x 的元素个数，时间复杂度为 $O(k+logn)$，其中 k 为元素 x 的个数。

2.7.5 #include <map>

map 容器是一个键值对 key-value 的映射。其内部实现是一颗以 key 为关键词的红黑树。map 的 key 和 value 可以是任意类型，其中 key 必须定义“小于号”运算符。

声明

map 的声明方法形式为 map<key_type, value_type>，例如：

```
1. map<long long, bool> vis;
2. map<string, int> hash;
3. map<pair<int, int>, vector<int>> test;
```

因为 map 基于平衡树实现，所以它的大部分操作的时间复杂度都在 $O(logn)$ 级别，略慢于使用 Hash 函数实现的传统 Hash 表。从 C++11 开始，STL 中新增了 unordered_set 和 unordered_map 等基于 Hash 的容器。

size/empty/clear/begin/end

与 set 类似，分别为元素个数、是否为空、清空、首迭代器、尾迭代器。

迭代器

map 的迭代器与 set 一样，也是双向访问迭代器。堆 map 的迭代器解除引用后，将得到一个二元组 pair<key_type, value_type>。

insert/erase

与 set 类似，分别为插入、删除。insert 的参数是 pair<key_type, value_type>, erase 的参数可以是 pair 或者迭代器。

find

h.find(x) 在变量名为 h 的 map 中查找 key 为 x 的二元组，并返回指向该二元组的迭代器。若不存在，返回 h.end()。时间复杂度为 $O(logn)$。

[]操作符

h[key]返回 key 映射到的 value 的引用，时间复杂度为 $O(logn)$。

需要注意的是，若查找的 key 不存在，则执行 h[key]后，h 会自动新建一个二元组，并返回零值，如整数 0，空字符串等。

2.7.6 #include <bitset>

bitset 可以看作一个多位二进制数，每 8 位占用一个字节，相当于采用了状态压缩的二

进制数组，并支持进本的位运算。在估算程序运行的时间时，我们一般以 32 位整数的运算次数作为基准，因此 n 位 bitset 执行依次位运算的复杂度可视为 $n/32$。

声明

```
1. bitset<10000> s; //  表示一个 10000 位二进制数
```

位运算操作符

~s：返回对 bitset s 按位取反后的结果。

&, |,^：返回对两个位数相同的 bitset 执行按位与、或、异或运算的结果。

>>,<<：返回把一个 bitset 右移、左移若干位的结果。

==,!=：比较两个 bitset 代表的二进制数是否相等。

[]操作符

s[k] 表示 s 的第 k 位，既可以取值，也可以赋值。

在 10000 位二进制数中，最低位为 s[0]，最高位为 s[9999]。

count

s.count() 返回有多少位为 1.

any/none

若 s 所有位都为 0，则 s.any() 返回 false，s.none() 返回 true。

若 s 至少一位为 1，则 s.any() 返回 true，s.none() 返回 false。

set/reset/flip

s.set() 把 s 所有位变为 1.

s.set(k,v) 把 s 的第 k 位改为 v，即 $s[k]=v$。

s.reset() 把 s 所有位变为 0。

s.reset(k) 把 s 的第 k 位改为 0，即 $s[k]=0$。

s.flip() 把 s 的所有位取反，即 $s=\sim s$。

s.flip(k) 把 s 的第 k 位取反，即 $s[k]\hat{}=1$。

2.7.7 #include <algorithm>

下面介绍的几个函数都作用在序列上，接收两个迭代器（或指针）l,r，对下标处于前闭后开区间 $[l,r)$ 的元素执行一系列操作。

reverse 翻转

```
1. reverse(a.begin(), a.end()); //  翻转 vector
2. reverse(a + 1, a + n + 1); //  翻转数组
```

unique 去重

返回去重之后的尾迭代器(或指针),这个尾迭代器是去重之后末尾元素的下一个位置。该函数常用于离散化，利用迭代器（或指针）的减法，可计算出去重后的元素个数 m。

```
1. a.erase(unique(a.begin(), a.end()), a.end()) //  vector 去重
2. int m = unique(a + 1, a + n + 1) - (a + 1); //  数组去重
```

random_shuffle 随机打乱

用法与 reverse 相同。

next_permutation 下一个排列

把两个迭代器（或指针）指定的部分看作一个排列，求出这些元素构成的全排列中，字典序排在下一个的排列，并直接在序列上更新。另外，若不存在排名更靠后的排列，则返回 false，否则返回 true。同理，也有 prev_permutation 函数。

```
1. for (int i = 1; i <= n; i++) a[i] = i;
2. do {
3.     for (int i = 1; i <= n; i++) cout << a[i] << " \n"[i == n];
4. } while (next_permutation(a + 1, a + n + 1));
```

sort 快速排序

对两个迭代器（或指针）指定的部分进行快速排序。可以在第三个参数传入定义大小比较的函数，或者重载“小于号”运算符。

```
 1. int a[M + 5];
 2. bool cmp(int a, int b) {return a > b;}
 3. sort(a + 1, a + n + 1, cmp);
 4.
 5. struct rec {int id, x, y;};
 6. bool operator<(const rec& a, const rec& b) {
 7.     return a.x < b.x || (a.x == b.x && a.y < b.y);
 8. }
 9. vector<rec> a;
10. sort(a.begin(), a.end());
```

lower_bound/upper_bound 二分

lower_bound 的第三个参数传入一个元素 x，在两个迭代器（或指针）指定的部分上执行二分查找，返回第一个大于等于 x 的元素的位置的迭代器（或指针）。

upper_bound 的用法和 lower_bound 大致相同，唯一区别是查找第一个大于 x 的元素。当然，两个迭代器（或指针）指定的部分应该是提前排好序的。

```
1. //  在有序数组中查找第一个大于等于 x 的元素的下标
2. int i = lower_bound(a + 1, a + n + 1, x) - a;
```

```
3. //  在有序 vector 中查找最后一个小于等于 x 的元素（假设存在）
4. int y = *--upper_bound(a.begin(), a.end(), x);
```

2.8 pb_ds

pb_ds 库全称 Policy-Based Data Structures。pb_ds 库封装了很多数据结构，比如哈希（Hash）表，平衡二叉树，字典树（Trie 树），堆（优先队列）等。就像 vector、set、map 一样，其组件均符合 STL 的相关接口规范。部分（如优先队列）包含 STL 内对应组件的所有功能，但比 STL 功能更多。下面以讲解 pb_ds 库中的堆和平衡树为例。

2.8.1 堆

模板形参

```
1. #include <ext/pb_ds/priority_queue.hpp>
2. using namespace __gnu_pbds;
3. __gnu_pbds ::priority_queue<T, Compare, Tag, Allocator>
```

T: 储存的元素类型

Compare: 提供严格的弱序比较类型

Tag: 是 __gnu_pbds 提供的不同的五种堆，Tag 参数默认是 pairing_heap_tag 五种分别是：

1. pairing_heap_tag：配对堆
2. binary_heap_tag：二叉堆
3. binomial_heap_tag：二项堆
4. rc_binomial_heap_tag：冗余计数二项堆
5. thin_heap_tag：除了合并的复杂度都和 Fibonacci 堆一样的一个 tag
6. Allocator：空间配置器，由于算法竞赛中很少出现，故这里不做讲解

算法竞赛中一般使用默认的 Tag 参数即可。

构造方式

```
1. __gnu_pbds ::priority_queue<int> __gnu_pbds::priority_queue<int, greater<int> >
2. __gnu_pbds ::priority_queue<int, greater<int>, pairing_heap_tag>
3. __gnu_pbds ::priority_queue<int>::point_iterator id; // 点类型迭代器
4. // 在 modify 和 push 的时候都会返回一个 point_iterator
5. id = q.push(1);
```

成员函数

push(): 向堆中压入一个元素，返回该元素位置的迭代器。时间复杂度 $O(1)$。

pop(): 将堆顶元素弹出。时间复杂度最坏 $O(n)$，均摊 $O(logn)$。

top(): 返回堆顶元素。

size() 返回元素个数。时间复杂度 $O(1)$。

empty() 返回是否非空。时间复杂度 $O(1)$。

modify(point_iterator, const key): 把迭代器位置的 key 修改为传入的 key，并对底层储存结构进行排序。时间复杂度最坏 $O(n)$，均摊 $O(logn)$。

erase(point_iterator): 把迭代器位置的键值从堆中擦除。时间复杂度最坏 $O(n)$，均摊 $O(logn)$。

join(__gnu_pbds::priority_queue &other): 把 other 合并到 *this 并把 other 清空。时间复杂度 $O(1)$。

示例

```cpp
#include <algorithm>
#include <cstdio>
#include <ext/pb_ds/priority_queue.hpp>
#include <iostream>
using namespace __gnu_pbds;
// 为了更好的阅读体验，定义宏如下 :
#define pair_heap __gnu_pbds ::priority_queue<int>
pair_heap q1;  // 大根堆，配对堆
pair_heap q2;
pair_heap ::point_iterator id;  // 一个迭代器

int main() {
    id = q1.push(1);
    // 堆中元素 : [1];
    for (int i = 2; i <= 5; i++) q1.push(i);
    // 堆中元素 : [1, 2, 3, 4, 5];
    std ::cout << q1.top() << std ::endl;
    // 输出结果 : 5;
    q1.pop();
    // 堆中元素 : [1, 2, 3, 4];
    id = q1.push(10);
    // 堆中元素 : [1, 2, 3, 4, 10];
    q1.modify(id, 1);
    // 堆中元素 : [1, 1, 2, 3, 4];
    std ::cout << q1.top() << std ::endl;
    // 输出结果 : 4;
    q1.pop();
    // 堆中元素 : [1, 1, 2, 3];
    id = q1.push(7);
```

```
    // 堆中元素 : [1, 1, 2, 3, 7];
    q1.erase(id);
    // 堆中元素 : [1, 1, 2, 3];
    q2.push(1), q2.push(3), q2.push(5);
    // q1 中元素 : [1, 1, 2, 3], q2 中元素 : [1, 3, 5];
    q2.join(q1);
    // q1 中无元素，q2 中元素 : [1, 1, 1, 2, 3, 3, 5];
}
```

关于 pb_ds 库中关于堆的详细信息，请见官方文档：https://gcc.gnu.org/onlinedocs/libstdc++/ext/pb_ds/pq_performance_tests.html。

2.8.2 平衡树

模板形参

```
#include <ext/pb_ds/assoc_container.hpp>  // 因为 tree 定义在这里 所以需要包含这个头文件
#include <ext/pb_ds/tree_policy.hpp>
using namespace __gnu_pbds;
__gnu_pbds ::tree<Key, Mapped, Cmp_Fn = std::less<Key>, Tag = rb_tree_tag,
                  Node_Update = null_tree_node_update,
                  Allocator = std::allocator<char> >
```

Key: 储存的元素类型，如果想要存储多个相同的 Key 元素，则需要使用类似于 std::pair 和 struct 的方法，并配合使用 lower_bound 和 upper_bound 成员函数进行查找

Mapped: 映射规则（Mapped-Policy）类型，如果要指示关联容器是集合，类似于存储元素在 std::set 中，此处填入 null_type，低版本 g++ 此处为 null_mapped_type；如果要指示关联容器是带值的集合，类似于存储元素在 std::map 中，此处填入类似于 std::map<Key, Value> 的 Value 类型

Cmp_Fn: 关键字比较函数，例如 std::less<Key>

Tag: 选择使用何种底层数据结构类型，默认是 rb_tree_tag。__gnu_pbds 提供不同的三种平衡树，分别是：

1. rb_tree_tag：红黑树，一般使用这个，后两者的性能一般不如红黑树
2. splay_tree_tag：splay 树
3. ov_tree_tag：有序向量树，只是一个由 vector 实现的有序结构，类似于排序的 vector 来实现平衡树。

Node_Update：用于更新节点的策略，默认使用 null_node_update，若要使用 order_of_key 和 find_by_order 方法，需要使用 tree_order_statistics_node_update

Allocator：空间分配器类型

构造方式

```
__gnu_pbds::tree<std::pair<int, int>, __gnu_pbds::null_type,
                 std::less<std::pair<int, int> >, __gnu_pbds::rb_tree_tag,
                 __gnu_pbds::tree_order_statistics_node_update>
    trr;
```

成员函数

insert(x)：向树中插入一个元素 x，返回 std::pair<point_iterator, bool>。

erase(x)：从树中删除一个元素/迭代器 x，返回一个 bool 表明是否删除成功。

order_of_key(x)：返回 x 以 Cmp_Fn 比较的排名。

find_by_order(x)：返回 Cmp_Fn 比较的排名所对应元素的迭代器。

lower_bound(x)：以 Cmp_Fn 比较做 lower_bound，返回迭代器。

upper_bound(x)：以 Cmp_Fn 比较做 upper_bound，返回迭代器。

join(x)：将 x 树并入当前树，前提是两棵树的类型一样，x 树被删除。

split(x,b)：以 Cmp_Fn 比较，小于等于 x 的属于当前树，其余的属于 b 树。

empty()：返回是否为空。

size()：返回大小。

示例

```
// Common Header Simple over C++11
#include <bits/stdc++.h>
using namespace std;
typedef long long ll;
typedef unsigned long long ull;
typedef long double ld;
typedef pair<int, int> pii;
#define pb push_back
#define mp make_pair
#include <ext/pb_ds/assoc_container.hpp>
#include <ext/pb_ds/tree_policy.hpp>
__gnu_pbds ::tree<pair<int, int>, __gnu_pbds::null_type, less<pair<int, int> >,
                  __gnu_pbds::rb_tree_tag,
                  __gnu_pbds::tree_order_statistics_node_update>
    trr;

int main() {
    int cnt = 0;
    trr.insert(mp(1, cnt++));
    trr.insert(mp(5, cnt++));
```

```
    trr.insert(mp(4, cnt++));
    trr.insert(mp(3, cnt++));
    trr.insert(mp(2, cnt++));
    // 树上元素 {{1,0},{2,4},{3,3},{4,2},{5,1}}
    auto it = trr.lower_bound(mp(2, 0));
    trr.erase(it);
    // 树上元素 {{1,0},{3,3},{4,2},{5,1}}
    auto it2 = trr.find_by_order(1);
    cout << (*it2).first << endl;
    // 输出排名 0 1 2 3 中的排名 1 的元素的 first:1
    int pos = trr.order_of_key(*it2);
    cout << pos << endl;
    // 输出排名
    decltype(trr) newtr;
    trr.split(*it2, newtr);
    for (auto i = newtr.begin(); i != newtr.end(); ++i) {
        cout << (*i).first << ' ';
    }
    cout << endl;
    // {4,2},{5,1} 被放入新树
    trr.join(newtr);
    for (auto i = trr.begin(); i != trr.end(); ++i) {
        cout << (*i).first << ' ';
    }
    cout << endl;
    cout << newtr.size() << endl;
    // 将 newtr 树并入 trr 树，newtr 树被删除。
    return 0;
}
```

关于 pb_ds 库中关于树的详细信息，请见官方文档：https://gcc.gnu.org/onlinedocs/libstdc++/ext/pb_ds/tree_based_containers.html。