堆
Heap

河南省实验中学信息技术组

2026 年 02 月 23 日

知识回顾

- 二叉树的存储
- 满二叉树和完全二叉树

满二叉树和完全二叉树

- 满二叉树：在一棵二叉树中，如果所有内部结点都有左右子树，并且所有的叶子都在同一层上，这样的二叉树称为满二叉树。
- 完全二叉树：对一棵具有 n 个结点的的二叉树按照层序编号，如果编号为 $i(1 \leq i \leq n)$ 的结点与同样深度的满二叉树中编号为 i 的结点在二叉树中的位置完全相同，则这棵二叉树称为完全二叉树。

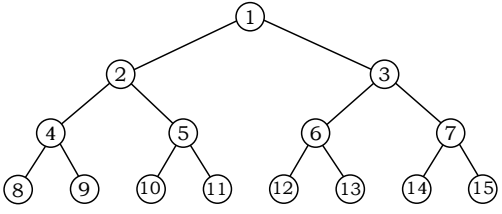


图: 满二叉树

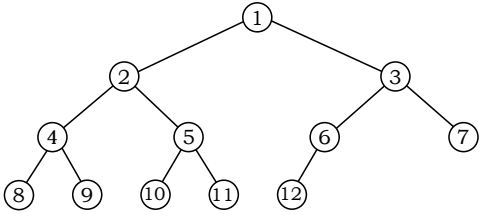


图: 完全二叉树

完全二叉树性质

- **性质：具有 n 个结点的完全二叉树的深度为 $\lceil \log_2(n+1) \rceil$。**
  证明：深度为 k 的二叉树最多结点个数 $n \le 2^k - 1$，最少结点个数 $n > 2^{k-1} - 1$，因此

$$2^{k-1} - 1 < n \le 2^k - 1$$
$$2^{k-1} < n + 1 \le 2^k$$
$$k - 1 < \log_2(n+1) \le k$$

  因深度只能是整数，因此 $k = \lceil \log_2(n+1) \rceil$。

完全二叉树性质

- 如果将一个有 n 个结点的完全二叉树的结点按照层次从上向下，每层从左到右编号，则对于任意一个结点 $i(1 \le i \le n)$，有:
  1. 如果 $i = 1$，则结点 i 是完全二叉树的根，无双亲；如果 $i > 1$，则其双亲结点编号为 $\lfloor \frac{i}{2} \rfloor$。
  2. 如果 $2i > n$，则结点 i 无左孩子；否则，其左孩子结点编号为 $2i$。
  3. 如果 $2i + 1 > n$, 则结点 i 无右孩子；否则，其右孩子结点编号为 $2i + 1$。
  4. 若结点编号 i 为偶数，则它是左孩子，它的右兄弟结点为 $i + 1$(如果存在)。
  5. 若结点编号 i 为奇数数且 $i \neq 1$，则它是右孩子，它的左兄弟结点为 $i - 1$。
  6. 结点 i 所在的层次为 $\lceil \log_2(i + 1) \rceil$。

堆的定义

- 堆 (Heap) 是一种数据结构，有很多类，例如二叉堆、斐波那契堆、二项式堆等，一般说的堆都指二叉堆。
- 二叉堆是一棵完全二叉树，其中每个结点的值都大于等于 (小于等于) 其孩子结点 (如果有孩子) 的值。
- 如果每个结点的值都大于等于其孩子结点 (如果有孩子) 的值，那么完全二叉树的根结点的值最大，我们称这种堆为最大堆(大根堆)。
- 如果每个结点的值都小于等于其孩子结点 (如果有孩子) 的值，那么完全二叉树的根结点的值最小，我们称这种堆为最小堆(小根堆)。
- 完全二叉树的根结点被称为堆顶。
- 从根结点到叶子结点的值是单调递增 (递减) 的。
- 当完全二叉树的结点满足堆的特点时被称为堆有序。

堆的定义

- 左图中二叉树每个结点的值都比它的左右孩子大，而根为最大值，即堆顶为 90；右图正好相反，每个结点的值都比它的左右孩子小，而根为最小值，即堆顶为 10。

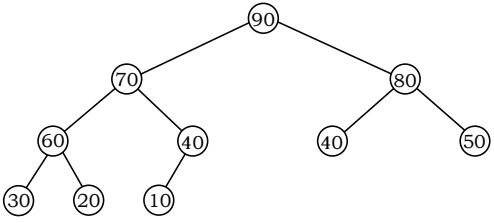


图: 最大堆

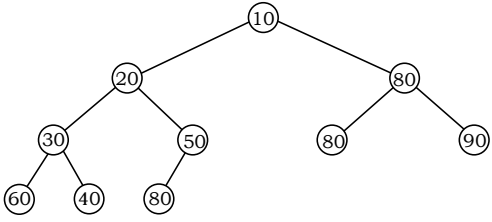


图: 最小堆

存储方式

- 堆的存储时按照完全二叉树的编号方式，将对应结点数据存储在数组对应下标的位置上。
- 对于任意一个结点 i，它的左孩子为 $2*i$(如果有)，右孩子为 $2*i+1$(如果有)，双亲为 $\lfloor i/2 \rfloor$(如果有)。
- 利用数组的下标可以实现树中上下层次移动：上移一层，$i=i/2$；下移一层，$i=2*i$ 或者 $i=2*i+1$。
- 下面我们以最小堆为例来介绍存储方式，最大堆与之相似。

下标	0	1	2	3	4	5	6	7	8	9	10
元素		10	20	80	30	50	80	90	60	40	80

表: 最小堆存储结构

堆操作

- 堆的存储

```
const int N = 100;
int a[N + 10];    // 存储堆的数组
int n = 0;        // 堆大小
```

- 判断堆是否为空

```
bool empty()
{
    return n == 0;
}
```

- 获取堆顶

```
// 使用前一定要判断堆是否为空
int top()
{
    return a[1];
}
```

插入元素

- 将新结点插入到数组的尾部，成为完全二叉树的最后一个结点，堆大小加 1。
- 此时，堆的结构可能被破坏，即新插入结点比其双亲结点的值小。
- 为了修复堆，沿着新结点到根结点的路径上，将新结点与其双亲结点比较交换，直到新结点成为根结点或双亲结点的值小于等于新结点的值，这种操作被称为上浮操作。

插入元素

- 例如，对于前面的最小堆，插入新结点 10，操作过程如下：

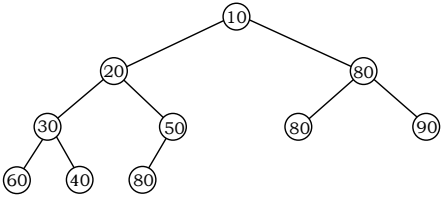


图: 最小堆

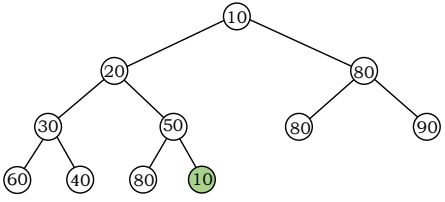


图: 插入新结点 10

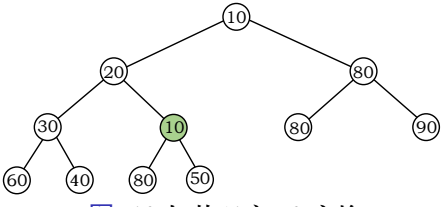


图: 10 与其双亲 50 交换

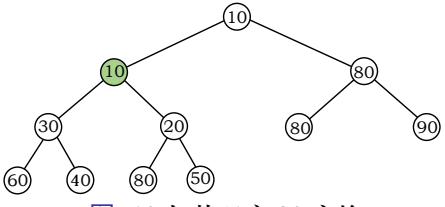


图: 10 与其双亲 20 交换

插入元素

```
1  // 上浮操作
2  void swim(int k)
3  {
4      while(k > 1 && a[k] < a[k / 2]) // 不是根结点并且值比其双亲结点小
5      {
6          swap(a[k], a[k / 2]);   // 和双亲数据交换
7          k = k / 2;              // 向上移动
8      }
9  }
10
11 void push(int x)
12 {
13     a[++n] = x; // 分配空间 并将堆大小加 1
14     swim(n);    // 上浮
15 }
```

- 时间复杂度：时间主要消耗在上浮操作中，每执行一次上浮操作时间复杂度为 O(1)，最多执行 $\log N$ 次，时间复杂度为 O($\log N$)。

删除堆顶

- 首先，用堆的最后一个结点的值覆盖堆顶值，同时将堆大小减 1。
- 此时，堆的结构可能被破坏，即堆顶可能不再小于等于其孩子的值。
- 为了修复堆，沿着根结点到叶子结点的路径上，将结点与其孩子结点比较交换，直到结点成为叶子结点或结点的值小于等于孩子结点的值，这种操作被称为下沉操作。
- 例如，对于前面的最小堆，删除堆顶 10，操作过程如下：

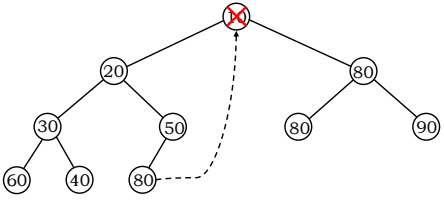


图: 用结点 80 覆盖堆顶 (删除堆顶)

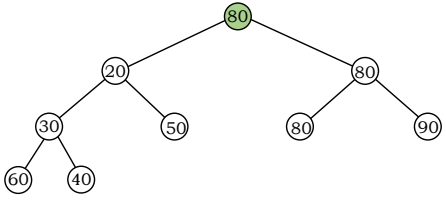


图: 删除堆顶后的堆

删除堆顶

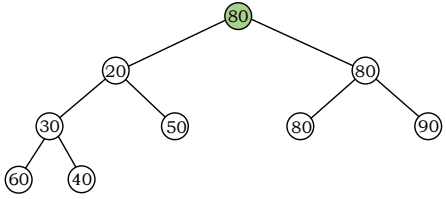


图: 删除堆顶后的堆

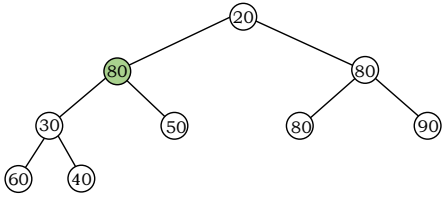


图: 80 与左孩子 20(较小) 交换

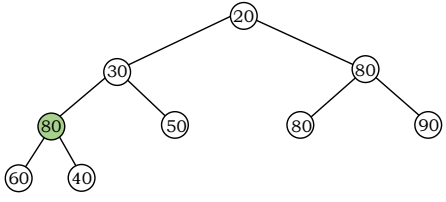


图: 80 与左孩子 30(较小) 交换

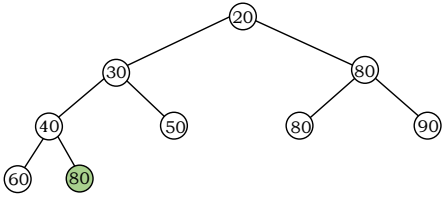


图: 80 与右孩子 40(较小) 交换

删除堆顶

```
1  // 下沉操作
2  void sink(int k)
3  {
4      while(2 * k <= n)  // 非叶结点
5      {
6          int j = 2 * k;
7          if(j + 1 <= n && a[j] > a[j + 1])  // 如果存在有孩子且右孩子较小
8              j = j + 1;
9          if(a[k] <= a[j]) break; // 如果结点值小于等于其孩子结点就结束下沉
10         swap(a[k], a[j]);
11         k = j;
12     }
13 }
14
15 // 使用前一定要判断堆是否为空
16 void pop()
17 {
18     a[1] = a[n--];  // 最后一个结点移到根
19     sink(1);        // 从根结点开始下沉
20 }
```

- **时间复杂度：时间主要消耗在下沉操作中，时间复杂度为** $\mathrm{O}(\log N)$。

建堆

- 建堆就是将一个不满足堆性质的序列建成一个堆。
- 建立堆方法一般有两种：
  - 上浮建堆
  - 下沉建堆

上浮建堆

- 算法 1：首先建立一个空堆，将元素逐一插入堆中。

```
1 void makeheap()
2 {
3     // m 为元素个数 序列存储在数组 a[] 中 插入完成后堆的大小 n=m
4     for(int i = 1; i <= m; ++i) push(a[i]); // 入堆
5 }
```

- 算法 2：通过观察可以发现，序列 a 的每一个元素的位置恰好就是入堆时插入的的位置，因此对序列 a 的的每个元素执行上浮操作即可。

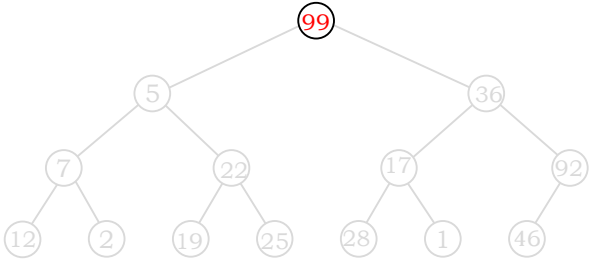
```
1 void makeheap()
2 {
3     n = m; // 堆的大小初始化为元素个数
4     // 对于第 i 个元素，它的上浮操作不会影响后面的元素，我们可以看作是插入第 i 个元素
5     for(int i = 1; i <= n; ++i) swim(i);
6 }
```

- 因在每一次插入元素时会执行一次上浮操作，故称之为上浮建堆。
- 最坏的情况下，每一个元素都要上浮到根结点，时间复杂度为 $O(N \log N)$。

上浮建堆

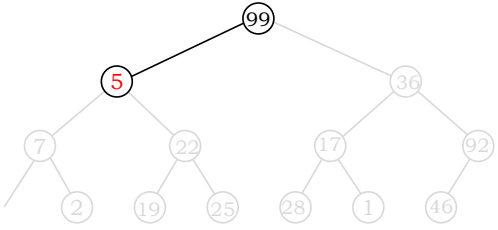
下标	1	2	3	4	5	6	7	8	9	10	11	12	13	14
数据	99	5	36	7	22	17	92	12	2	19	25	28	1	46

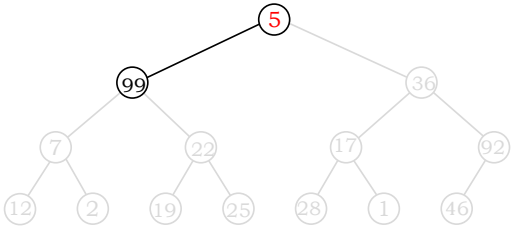
- 首先，对 1 号进行上浮调整，它自己构成的树显然满足堆的性质。



上浮建堆

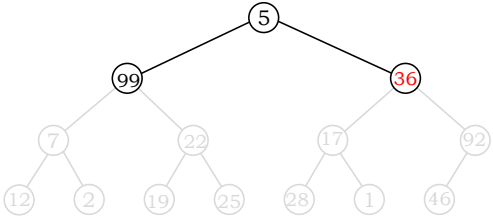
- 对 2 号进行上浮调整。





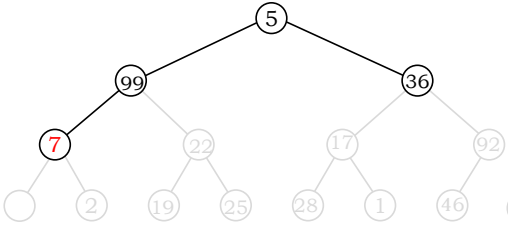
上浮建堆

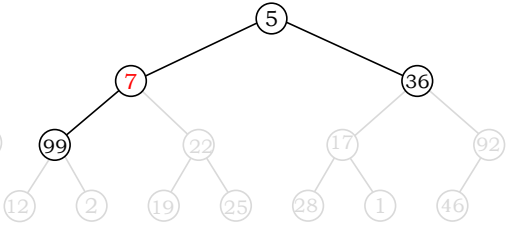
- 对 3 号进行上浮调整，此时不需要上浮。



上浮建堆

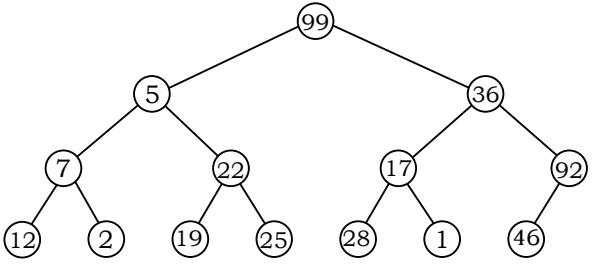
- 对 4 号进行上浮调整。





- 其他顶点的上浮操作依此类推。

下沉建堆

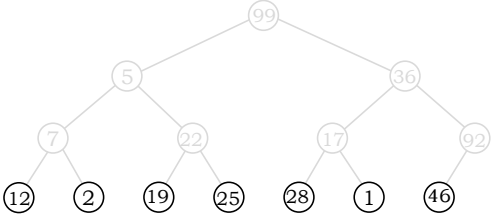


下标	1	2	3	4	5	6	7	8	9	10	11	12	13	14
数据	99	5	36	7	22	17	92	12	2	19	25	28	1	46

- 从最后一个结点开始，依次判断以这结点为根的子树是否符合最小堆的特性，如果不符合，就下沉调整使之符合。
- 调整到所有的子树都符合最小堆的特性，这个过程就是下沉建堆。

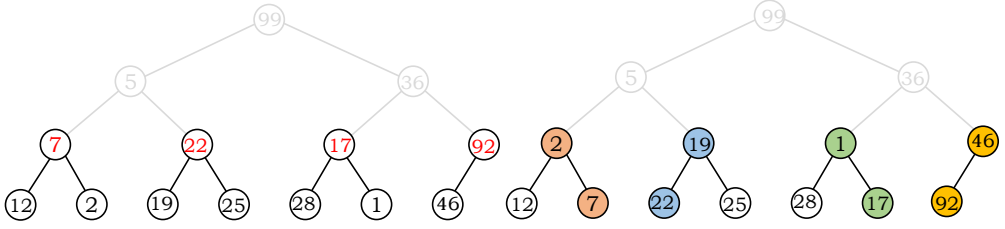
下沉建堆

- 首先，从叶子结点开始。因为叶子结点没有孩子，所以叶子结点为根结点的树一定符合最小堆的特性。



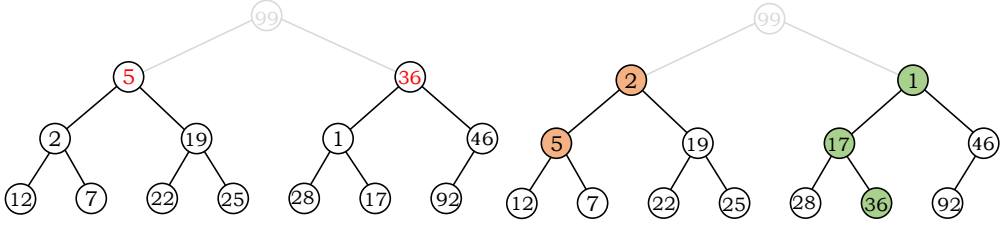
下沉建堆

- 因此，第一个可能不符合最小堆特性的结点就是最后一个非叶子结点，它的编号是 $\lfloor n/2 \rfloor$。
- 以 7 号、6 号、5 号、4 号结点为根的子树不符合最小堆的特性，需要下沉调整。



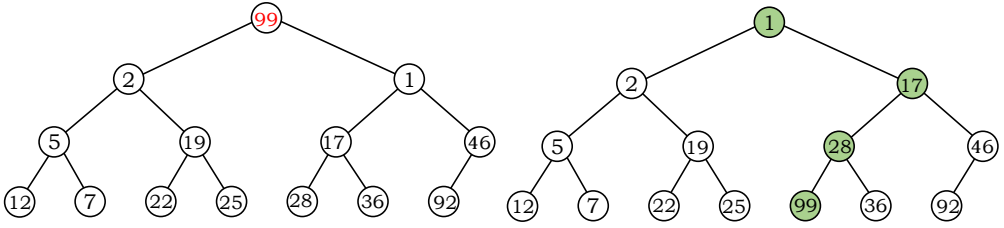
下沉建堆

- 以 3 号、2 号结点为根的树不符合最小堆的特性，需要下沉调整。



下沉建堆

- 最后，以 1 号结点为根的树不符合最小堆的特性，需要下沉调整。



- 调整完成后，整棵树符合最小堆的特性。

下沉建堆

- 下沉建堆算法：从最后一个非叶子结点到根结点，逐个扫描所有的结点，并根据需要进行下沉操作。
- 时间复杂度：$\Theta(N)$，感兴趣自行证明。
- 此建堆方法是在数组 $a[]$ 中直接建堆，所以堆大小要初始化为序列的长度。如果是默认建堆，堆大小应为 0。
- 推荐使用下沉建堆法。

```
1 // 下沉建堆
2 void makeheap()
3 {
4     n = m; // 堆的大小初始化为元素个数
5     // 从最后一个非叶子结点 n/2 开始到根结点执行下沉操作
6     for(int i = n / 2; i >= 1; --i) sink(i);
7 }
```

优先队列

- 相比于普通队列先进先出 (FIFO)，优先队列 (Priority queue) 每个元素都有一个优先级，按照优先级高低确定出队顺序的，优先级高的先出队。
- 优先队列可以用堆来实现，堆顶为优先级最大的元素。
- STL 中提供了一个优先队列的实现priority_queue，使用时需要引用头文件#include<queue>。

操作	功能
`priority_queue<int> pq;`	建立一个存放int类型数据的优先队列
`pq.empty()`	判断优先队列是否为空
`pq.size()`	返回优先队列大小
`pq.top()`	返回队头元素
`pq.push(int x)`	插入元素
`pq.pop()`	删除队头元素

表: 优先队列操作说明

优先队列


```cpp
int a[9] = { 0, 1, 2, 3, 4, 8, 9, 3, 5 };// 定义数组
// 定义优先队列，并将数组 a[1:8] 的值复制进去 默认建立最大堆
priority_queue<int> pq(a + 1, a + 9);
cout << pq.size();          // 输出 8，优先队列中有 8 个元素
cout << pq.top();           // 输出 9，队列中优先级最高的为 9
//输出队列的前 6 个元素，输出 9 8 5 4 3 3，从大到小
for (int i = 1; i <= 6; i++)
{
    cout << pq.top() << " ";
    pq.pop();
}
//此时队列中只剩下 2 1
pq.push(7);                 // 新元素 7 入队
cout << pq.size();  // 输出 3，优先队列中有 3 个元素
cout << pq.top();   // 输出队列头元素为 7，因为队列中 7 最大

//清空队列，并依次输出队中元素，从大到小输出 7 2 1
while (!pq.empty())
{
    cout << pq.top() << " ";
    pq.pop();
}
```

优先队列

- 如果想让队列按照从小到大的顺序排队呢?
- 方法 1：把数据的值全部变成相反数，然后放在队列中，那么就可以从小到大排序，但是要注意出队入队都要变成相反数。
- 方法 2：定义从小到大排序的队列。例如建立一个全为整数最小堆，写为`priority_queue<int, vector<int>, greater<int> > pq;`[1]。

[1] `greater<int>`和最后一个`>`之间要有空格，是为了与输出符号`>>`区分，在 C++11 及其以后的标准中可以省去空格。

优先队列

- 如果元素值有多个关键字怎么办?
- 如果只有两个关键字，那么可以考虑使用 pair，那么将按照 pair 的 first 元素从大到小排队，如果 first 相同将按照 second 元素从大到小排队。
- 当然如果你想从小到大排序，可以将值改为相反数，也可以定义从小到大排队的队列。

```
pair<int, int> a = {3, 5}; // a.first=3 是第一个元素 a.second=5 是第二个元素
priority_queue<pair<int,int> > pq; // 按照 first 和 second 从大到小排序
pq.push({1, 2});
pq.push(a);
pair<int, int> b = pq.top();

// 按照 first 和 second 从小到大排序
priority_queue<pair<int,int>, vector<pair<int,int>>, greater<pair<int,int>>> pq;
```

优先队列

- 如果有三个关键字以上，怎么办？[2]
- 自己实现堆。
- 使用多个 pair 嵌套。

```
priority_queue<pair<pair<int,int>, int>> pq; // 3 个关键字
priority_queue<pair<pair<int,int>, pair<int,int>>> pq; // 4 个关键字
```

- 定义结构体并重载小于比较方式。

```
struct Node
{
    int x, y, z;
    bool operator<(const Node &b) const
    {
        retun a.x > b.x || a.x == b.x && a.y > b.y;
    }
};

priority_queue<Node> pq;  // 按照 x, y 从大到小排队
```

[2] 自行学习

【例】中位数

【题目描述】
给定一个长度为 n 的整数数列 a，求数列 a 的中位数。
中位数：将长度为 n 的序列从小到大排序，如果 n 为奇数，则中位数为正中间的数，如果 n 为偶数，则中位数为正中间两个数的平均数。
要求：程序中在任意时刻不能存储的数列 a 中的数据不能超过 $\frac{3}{4}$。
【输入格式】
第一行一个正整数 $n(n \leq 5 \times 10^5)$，代表数列 a 的长度。
第二行包含 n 个用空格隔开的整数，代表数列 a 的每个元素。
【输出格式】
一行一个整数，表示数列 a 的中位数，保留一位小数。

【样例 1 输入】

```
3
2 3 1
```

【样例 1 输出】

```
2.0
```

【样例 2 输入】

```
4
3 4 1 2
```

【样例 2 输出】

```
2.5
```

【例】中位数

- 初始化一个最小堆 a，设置其大小最大为 $\lfloor n/2 \rfloor + 1$。
- 输入一个数 x，如果此时堆未满，那么就让该数 x 入堆，此时堆顶就是此前输入序列的最小值。
- 如果此时堆已满，那么比较该数 t 和堆顶 $a[1]$:
  - 如果 $x \leq a[1]$，说明在序列中 x 至少小于等于 $\lfloor n/2 \rfloor + 1$ 个数，那么 x 一定不可能是中位数，直接舍弃。
  - 如果 $x > a[1]$，说明堆顶一定不是中位数，而 x 可能是中位数的候选，让堆顶出堆，让 x 入堆。
- 当所有数据都输入完成后，此时堆内存储的是序列 A 中前 $\lfloor n/2 \rfloor + 1$ 大的数。
  - 如果 n 为奇数，此时堆顶即为答案。
  - 如果 n 为偶数，进行两次出堆的数的平均值即为答案。

【例】中位数

```
int m; cin >> m;
for(int i = 1; i <= m / 2 + 1; ++i)
{
    int x; cin >> x;
    insert(x);
}
for (int i = m / 2 + 2; i <= m; ++i)
{
    int x; cin >> x;
    insert(x);
    pop();
}
double ans = 0;
if (m % 2) ans = top(); // 奇数，中位数就是堆顶
else
{
    ans += top(); pop();
    ans += top(); pop();
    ans /= 2;
}
```

【例】中位数

```
1  priority_queue<int, vector<int>, greater<int> > pq; // 最小堆
2  int n; cin >> n;
3  for(int i = 1; i <= n; ++i)
4  {
5      int x; cin >> x;
6      if(pq.size() < n / 2 + 1) pq.push(x); // 堆未满
7      else if(x > pq.top()) pq.push(x), pq.pop(); // 堆满 x 可能是中位数的候选
8  }
9  double ans = 0;
10 if(n % 2) ans = pq.top();  // n 为奇数，中位数就是堆顶
11 else
12 {
13     ans += pq.top(); pq.pop();
14     ans += pq.top(); pq.pop();
15     ans /= 2;
16 }
```

堆的性质

- 性质 1：堆能在动态增加或删除最值时元素维护序列最大值或最小值。
- 性质 2：如果限定堆的大小为 k，那么可以在增加元素时实时维护前 k 大 (小) 值。
  - 如果要维护前 k 大的值，需要建立最小堆，如果元素比堆顶大，则需要将堆顶剔除，将元素入堆；
  - 如果要维护前 k 小的值，需要建立最大堆，如果元素比堆顶小，则需要将堆顶剔除，将元素入堆。
- 例如，设最大 (小) 堆大小最大为 $k = 4$ 时，依次增加的元素为 $5, 2, 1, 6, 9, 10, 3, 8, 4, 7$，观察堆内元素变化。

【例】畜栏预定

【题目描述】
有 N 头牛在畜栏中吃草。每个畜栏在同一时间段只能提供给一头牛吃草，所以可能会需要多个畜栏。给定 N 头牛和每头牛开始吃草的时间 A 以及结束吃草的时间 B，每头牛在 $[A, B]$ 这一时间段内都会一直吃草。当两头牛的吃草区间存在交集时（包括端点），这两头牛不能被安排在同一个畜栏吃草。求需要的最小畜栏数目和每头牛对应的畜栏方案。
【输入格式】
第 1 行一个整数 $N(N \leq 5 \times 10^4)$。
接下来 N 行，每行两个整数 $A, B(1 \leq A \leq B \leq 10^6)$，表示每头牛吃草的开始和结束时间。
【输出格式】
第一行一个整数，表示需要的最少畜栏数。
接下来 N 行，第 i 行一个整数表示第 i 头牛被安排到的畜栏编号，编号是从 1 开始的连续整数。
方案可能有多解，输出其中一种即可。

【例】畜栏预定

【样例输入】

```
5
1 10
2 4
3 6
5 8
4 7
```

【样例输出】

```
4
1
2
3
2
4
```

【样例解释】

畜栏编号	奶牛顺序
1	1
2	2,4
3	3
4	5

【例】畜栏预定

- 首先，按照牛的吃草开始时间从小到大排序。
- 排序后的第 1 头牛肯定要安排一个畜栏，但是对于后续的第 i 头牛，如果前面有畜栏的奶牛吃草结束时间早于这头牛，那么这头奶牛就可以被安排到对应畜栏，否则就需要新开畜栏。
- 所以需要一个数组来维护畜栏安排进去的最后一头牛和牛吃草的结束时间，最后答案就是数组使用的大小。
- 注意要求输出的畜栏编号要按照排序前奶牛的顺序。

```
1  struct Node
2  {
3      int id, x, y;
4  } a[N];  // 牛的吃草时间和编号
5  int b[N], c[N];    // 畜栏的结束时间和每头牛安排的畜栏编号
6
7  bool cmp(const Node &a, const Node &b)
8  {
9      return a.x < b.x;
10 }
```

【例】畜栏预定

```
for(int i = 1; i <= n; ++i) cin >> a[i].x >> a[i].y, a[i].id = i;
sort(a + 1, a + n + 1, cmp); // 按照开始时间从小到大排序
int cnt = 0;    // 安排的畜栏总数
for(int i = 1; i <= n; ++i)
{
    int x = a[i].x, y = a[i].y, id = a[i].id; // 牛的开始和结束时间 牛的编号
    int k = 0;      // 可以使用的畜栏
    for(int j = 1; j <= cnt; ++j)
        if(b[j] < x) { k = j; break; } // 找到可以使用的畜栏
    if(k) c[id] = k, b[k] = y;      // 有可以使用的畜栏
    else c[id] = ++cnt, b[cnt] = y;// 需要新建畜栏
}
```

- 时间复杂度：$O(N^2)$。

【例】畜栏预定

- 每次可以选择结束时间最早的畜栏，使用堆维护已经开设的畜栏。

```
// 最小堆 存放畜栏中最后一头牛结束吃草时间和畜栏编号
priority_queue<pair<int, int>, vector<pair<int, int> >,
                                greater<pair<int, int> > > pq;
int cnt = 0;
for(int i = 1; i <= n; ++i)
{
    if(pq.empty() || a[i].x <= pq.top().first) // 当前没有畜栏或者结束时间都晚于当前牛
    {
        c[a[i].id] = ++cnt;
        pq.push({a[i].y, cnt});
    }
    else
    {
        pair<int, int> t = pq.top(); pq.pop();
        c[a[i].id] = t.second;
        pq.push({a[i].y, t.second});
    }
}
```

- 时间复杂度：$O(N \log N)$。

【例】最小函数值

【题目描述】
有 n 个函数，分别为 $f_1, f_2, ..., f_n$，定义 $f_i(x) = a_i x^2 + b_i x + c_i (x \geq 1)$。给定这些 a_i b_i c_i，请求出所有函数的所有函数值中最小的 m 个 (如有重复的要输出多个)，其中 $n, m \leq 5 \times 10^5, a_i, b_i, c_i \geq 0$。
【输入格式】
第一行输入两个正整数 n, m。
以下 n 行每行三个正整数，其中第 i 行的三个数分别为 a_i, b_i, c_i。
【输出格式】
输出将这 n 个函数所有可以生成的函数值排序后的前 m 个元素。
这 m 个数应该输出到一行，用空格隔开，并且最后一个数右侧也有一个空格。

【例】最小函数值

【样例输入】

```
3 10
4 5 3
3 4 5
1 7 1
```

【样例输出】

```
9 12 12 19 25 29 31 44 45 54
```

【样例解释】

对于函数 $f_1(x) = 4x^2 + 5x + 3$，它的最小 10 个函数值为
$12, 29, 54, 87, 128, 177, 234, 299, 372, 453$；
对于函数 $f_2(x) = 3x^2 + 4x + 5$，它的最小 10 个函数值为
$12, 25, 44, 69, 100, 137, 180, 229, 284, 345$；
对于函数 $f_3(x) = x^2 + 7x + 1$，它的最小 10 个函数值为
$9, 19, 31, 45, 61, 79, 99, 121, 145, 171$；
那么最小的前 10 个函数值是 $9, 12, 12, 19, 25, 29, 31, 44, 45, 54$。

【例】最小函数值

- 每个函数的函数值是递增的，但是它们彼此之间的函数值大小则无确定关系。
- 而只求最小的 m 个函数值，那么每个函数最多只需要求前 m 个函数值。
- 将求出的 $n \times m$ 个函数值从小到大排序，取前 m 个即可。

```
long long d[N * M];   // 所有函数值
int a[N], b[N], c[N];
long long f(int a, int b, int c, int x)
{
    return (long long)a * x * x + b * x + c;
}

int tot = 0;       // 当前计算的函数值总数
for(int i = 1; i <= n; ++i)
  for(int x = 1; x <= m; ++x)
    d[++tot] = f(a[i], b[i], c[i], x);
sort(d + 1, d + tot + 1);
for(int i = 1; i <= m; ++i) cout << d[i] << " ";
```

- 时间复杂度：$O(MN \log(MN))$，空间复杂度：$O(MN)$，超时并且内存超限。

【例】最小函数值

- 首先，考虑最小的函数值，必定是所有函数 $x = 1$ 时的函数值的最小值，设这个函数为 f_k。
- 其次，第二小的函数值必定除了 f_k 之外的所有函数 $x = 1$ 时的函数值和函数 f_k 当 $x = 2$ 时的函数值的最小值，其他函数值依次类推。
- 那么每次只需要保留每个函数的一个函数值 (最小的) 即可，每次在这些函数值中找最小的就是答案。
- 当某个函数的最小值被输出后，就继续计算该函数的下一个函数值。

【例】最小函数值

```
1  long long d[N]; // 当前每个函数的函数值
2  int x[N];       // 每个函数当前的自变量值
3  long long f(int a, int b, int c, int x)
4  {
5      return (long long)a * x * x + b * x + c;
6  }
7
8  // 主函数
9  for(int i = 1; i <= n; ++i) // 先求每个函数 x=1 时的值
10 {
11      d[i] = f(a[i], b[i], c[i], 1);
12      x[i] = 1;
13 }
14 for(int i = 1; i <= m; ++i)
15 {
16     int k = 1;    // 最小的函数
17     for(int i = 2; i <= n; ++i) if(d[k] > d[i]) k = i;
18     cout << d[k] << " ";
19     d[k] = f(a[k], b[k], c[k], x[k] + 1);   // 计算下一个函数值
20     x[k] = x[k] + 1;
21 }
```

- 时间复杂度：O(MN)，超时。

【例】最小函数值

- 每次取最小函数值时都是通过枚举法，时间复杂度较高，可以用堆优化。
- 可以将函数值放在最小堆中，每次取堆顶，每次计算的新的函数值放入堆中即可。
- 但是只放函数值是不够的，因为每一次还要知道最小的函数值是哪个函数，所以堆内需要同时存储函数值和函数编号。

【例】最小函数值

```
1  int x[N];       // 每个函数当前的自变量值
2  long long f(int a, int b, int c, int x)
3  {
4      return (long long)a * x * x + b * x + c;
5  }
6  // 堆内需要存储函数值和函数编号
7  priority_queue<pair<long long, int>, vector<pair<long long, int> >,
8                                       greater<pair<long long, int> > > pq;
9  for(int i = 1; i <= n; ++i) // 先求每个函数 x=1 时的值
10 {
11      pq.push({f(a[i], b[i], c[i], 1), i});
12      x[i] = 1;
13 }
14 for(int i = 1; i <= m; ++i)
15 {
16     pair<long long, int> t = pq.top(); pq.pop();  // 取堆顶
17     cout << t.first << " ";
18     int k = t.second;   // 当前函数编号
19     pq.push({f(a[k], b[k], c[k], x[k] + 1), k}); // 计算下一个函数值
20     x[k] = x[k] + 1;
21 }
```

- 时间复杂度：$O(M \log N)$。

【例】超市

【题目描述】
超市里有 n 件商品，每个商品都有利润 p_i 和过期时间 d_i，每天只能卖一件商品，过期商品 (在第 k 天，$k > d_i$) 不能再卖。
求合理安排每天卖的商品的情况下，可以得到的最大收益是多少。
【输入格式】
第 1 行一个整数 $n(n \leq 10000)$。
接下来 n 行，每行两个整数表示商品的利润和过期时间 $(1 \leq p_i, d_i \leq 10000)$。
【输出格式】
一行一个整数表示最大收益值。

【样例输入】

```
7
20 1
2 1
10 3
100 2
8 2
5 20
50 10
```

【样例输出】

```
185
```

【例】超市

- 商品过期前能卖尽量卖，但是由于每天只能卖一件，而商品又特别多，所以第 t 天时，可能会有很多商品过期不能卖掉。所以，第 t 天时，在保证不卖出过期商品的前提下，尽量卖出利润前 t 大的的商品。
- 把商品按照过期时间从小到大排序，然后依次处理每个商品。
- 为了始终维护利润前若干大的商品，建立一个最小堆来存储售卖的商品 (结点为利润)。
- 当处理到第 i 件商品时:
  - 如果 d_i 大于当前堆大小，那么直接将该商品的利润入堆 (过期前肯定有时间可以卖)。
  - 如果 d_i 等于当前堆大小，说明在目前方案中，前 d_i 天已经安排了 d_i 个商品卖出。此时，如果当前商品利润大于堆顶值 (已经安排的 d_i 个商品中的最低利润)，那么用当前商品的利润替换堆顶 (用当前商品替换掉原方案中利润最低的商品)。
- 堆内存储的即为要卖出商品的利润，堆中所有元素的和就是答案。

【例】超市

```
sort(g + 1, g + n + 1, cmp);  // 按照过期时间从小到大排序
priority_queue<int, vector<int>, greater<int> > pq;
for(int i = 1; i <= n; ++i)
{
    // 前 d 天中所有合法商品的前 d 大未找完
    if(pq.size() < g[i].d) pq.push(g[i].p);
    // 前 d 大已经确定 但是第 i 件物品更优
    else if(pq.size() == g[i].d && pq.top() < g[i].p)
    {
        pq.pop();
        pq.push(g[i].p);
    }
}
int ans = 0;
while(!pq.empty()) ans += pq.top(), pq.pop();
```

【例】鱼塘钓鱼

有 n 个鱼塘排成一排，每个鱼塘里都有一定数量的鱼。当你在某一个鱼塘钓鱼时，每小时能钓的鱼数量会逐渐减少，例如开始时一小时能钓 10 条鱼，每过一个小时，钓鱼的数量都会减少 2 条，那么当第 6 个小时你就钓不到鱼了。

那么我们可以选择到后续其他鱼塘钓鱼，但是从每个鱼塘到下一个相邻鱼塘都要花费一定时间，现在给你一个截止时间 $t(t \leq 1000)$，请你设计一个方案，使得你从第 1 个鱼塘出发，能钓到最多的鱼 (所有的时间都为整数)。

例如，有 5 个鱼塘，它们的首小时钓鱼量、每小时钓鱼减少量、到下一个相邻鱼塘所花费的时间如下表:

鱼塘编号	1	2	3	4	5
第 1 小时能钓到的鱼	10	14	20	16	9
每小时钓鱼减少量	2	4	6	5	3
到相邻下一个鱼塘所花费的时间	3	5	4	4	

那么，如果给定的时间 $t = 14$，那么最多可以钓 76 条鱼。

【例】鱼塘钓鱼

【输入格式】
第 1 行一个整数 n，表示鱼塘的数目。
第 2 行为第 1 小时各个鱼塘能钓到的鱼的数量，每个数据之间用一空格隔开。
第 3 行为每过 1 小时各个鱼塘钓鱼数的减少量，每个数据之间用一空格隔开。
第 4 行为当前鱼塘到下一个相邻鱼塘需要的时间。
第 5 行为截止时间 t。
【输出格式】
输出仅一个整数，表示你的方案能钓到的最多的鱼，输入保证答案在 int 范围内。

【样例输入】

```
5
10 14 20 16 9
2 4 6 5 3
3 5 4 4
14
```

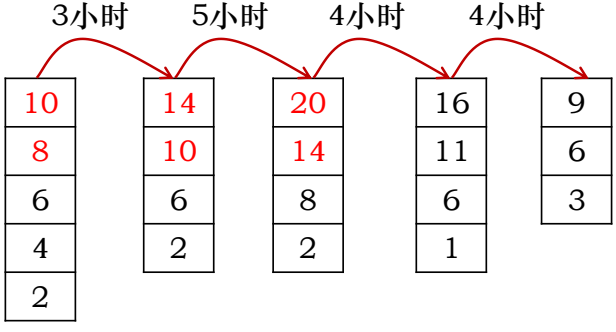
【样例输出】

```
76
```

【例】鱼塘钓鱼

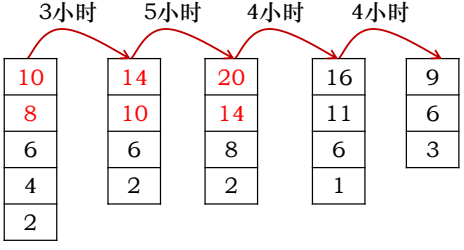
【样例解释】

- 前 2 小时在 1 号池塘钓鱼，钓到 10+8=18 条；
- 接下来用 3 小时前往 2 号池塘，在 2 号池塘钓 2 小时，钓到 14+10=24 条；
- 然后用 5 小时前往 3 号池塘，在 3 号池塘钓 2 小时，钓到 20+14=34 条。



【例】鱼塘钓鱼

- 因为要从第 1 个鱼塘向后走，那么如果选择了在前几个鱼塘钓鱼，那么只需要按照先后顺序走即可，而花费在路上的时间只跟最后一个到达的鱼塘位置有关。
- 因此枚举方案中最后到达的鱼塘编号 k，那么前 k 个鱼塘可以钓鱼的总时间就是 $t = T-$ 路上消耗的时间。
- 那么在这 t 小时内，每次都可以在前 k 个鱼塘中贪心的选择能钓鱼最多的鱼塘，在该鱼塘掉完一次后，更新该鱼塘的钓鱼数量。
- 可以用最大堆来维护鱼塘当前可以钓到的鱼的数量，每次取堆顶，然后更新钓鱼数量后放入堆中即可。



【例】鱼塘钓鱼

```cpp
int ans = 0;
for(int k = 1; k <= n; ++k)  // 枚举在前 k 个鱼塘钓鱼
{
    int t = T;
    for(int i = 1; i < k; ++i) t -= c[i];
    if(t <= 0) break;   // 如果没有剩余时间 就没必要进行了
    priority_queue<pair<int, int>> pq;
    for(int i = 1; i <= k; ++i) pq.push({a[i], i}); // 将每个鱼塘的钓鱼量入堆
    int cnt = 0;        // 钓鱼数
    while(t--)
    {
        if(pq.empty()) break;   // 所有鱼塘都钓不到鱼
        int y = pq.top().first;
        int id = pq.top().second;
        pq.pop();
        cnt += y;
        if(y - b[id] > 0) pq.push({y - b[id], id});   // 该鱼塘下一个小时还能钓鱼
    }
    ans = max(ans, cnt);
}
```

练习

- 堆的基本操作 (A0230)
- 排序测试 (C0637)
- 第 k 大的数 (COGS 3906)
- 最小函数值 (COGS 2334)
- Stall Reservation(COGS 3448)
- Supermarket(COGS 3097)
- 鱼塘钓鱼 (COGS 1748)
- Runing Median(COGS 3415)
- 黑盒子 (COGS 601)
- 数字对数 (COGS 3759)
- 建筑抢修 [JSOI 2007](COGS 929)

堆排序

- 选择排序：每次都从剩余待排序的序列中找出最小 (大) 元素，并将每次找出的最小 (大) 元素放到排好序的序列后。
- 选择排序时间复杂度：查找最小 (大) 元素的复杂度为 $O(N)$，共需要寻找 $O(N)$ 次，故整体的时间复杂度为 $O(N^2)$。
- 查找最小 (大) 元素使用遍历序列的方式效率较低，可以使用堆来动态维护序列的最小 (大) 元素，每次取出堆顶即为最小 (大) 元素，这就是堆排序。
- 堆排序时间复杂度：查找最小 (大) 元素的复杂度为 $O(\log N)$，故整体的时间复杂度为 $O(N \log N)$。

堆排序算法 1

- 将待排序序列利用下沉建堆的方式建立最小堆；
- 将堆顶元素取出，存储到结果有序序列中，剩余的元素调整为最小堆；
- 重复上述步骤，直到最小堆为空。

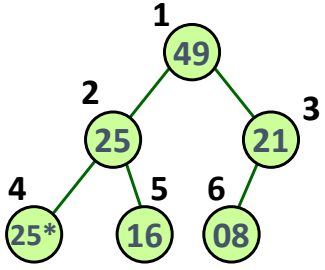
```
1  const int N = 100;
2  int a[N + 10];  // 存储堆
3  int n;          // 堆的大小
4  int b[N + 10];  // 存储排好序的序列
5
6  for(int i = n / 2; i >= 1; --i) sink(i);  // 下沉建堆
7  int m = n;    // 删除堆顶会导致 n 变小
8  for(int i = 1; i <= m; ++i)
9  {
10     b[i] = a[1];   // 获取堆顶最小值
11     pop();
12 }
```

- 时间复杂度：$O(N \log N)$。
- 空间复杂度：$O(N)$。

堆排序算法 2

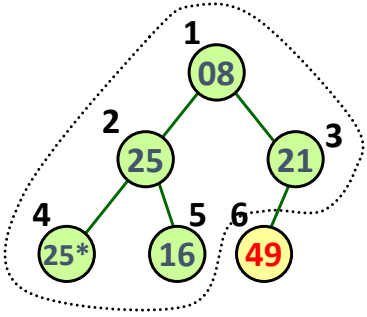
- 将待排序序列利用下沉建堆的方式建立最大堆 (为什么是最大堆而不是最小堆？)；
- 将堆顶元素和堆对应的数组中最后一个元素交换，将堆大小减 1，此时堆顶元素不满足堆的性质，于是利用下沉法对堆进行调整；
- 重复上述步骤，直到全部元素已经有序。

堆排序过程



49	25	21	25*	16	08

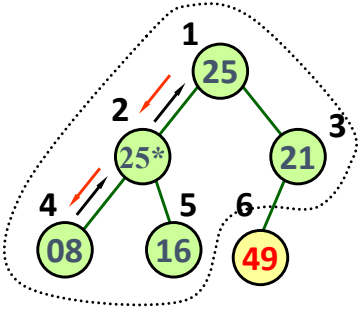
初始最大堆



08	25	21	25*	16	49

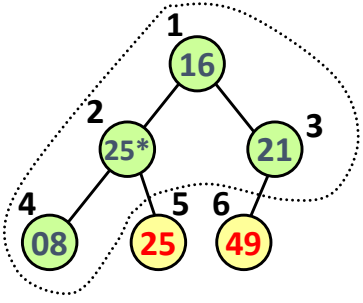
交换1号与6号元素，
6号元素就位

堆排序过程



25	25*	21	08	16	49

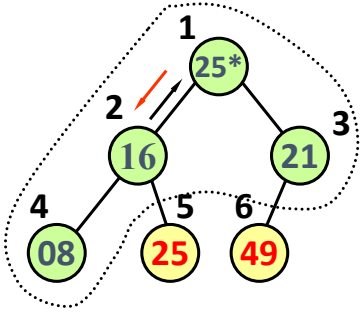
从1号到5号，重新调整为最大堆



16	25*	21	08	25	49

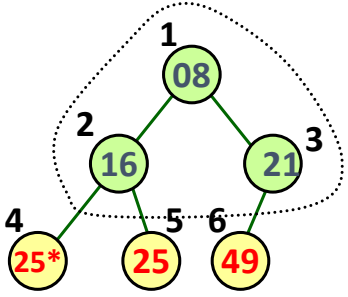
交换1号与5号元素，5号元素就位

堆排序过程



25*	16	21	08	25	49

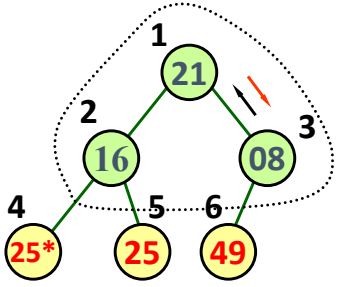
从1号到4号，重新调整为最大堆



08	16	21	25*	25	49

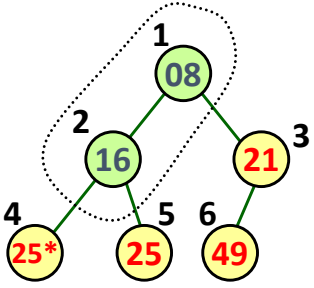
交换1号与4号元素，4号元素就位

堆排序过程



21	16	08	25*	25	49

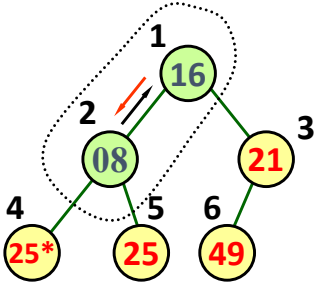
从1号到3号，重新调整为最大堆



08	16	21	25*	25	49

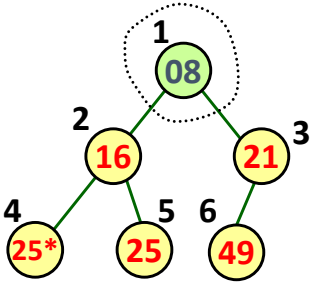
交换1号与3号元素，3号元素就位

堆排序过程



16	08	21	25*	25	49

从1号到2号，重新调整为最大堆



08	16	21	25*	25	49

交换1号与2号元素，2号元素就位

堆排序算法 2 代码

```
1  // 下沉操作
2  void sink(int k)
3  {
4      while(2 * k <= n)  // 非叶结点
5      {
6          int j = 2 * k;
7          if(j + 1 <= n && a[j] < a[j + 1])  // 如果存在有孩子且右孩子较小
8              j = j + 1;
9          if(a[k] >= a[j]) break; // 如果结点值小于等于其孩子结点就结束下沉
10         swap(a[k], a[j]);
11         k = j;
12     }
13 }
14
15 n = m;
16 for(int i = n / 2; i >= 1; --i) sink(i); // 下沉建堆
17 for(int i = 1; i <= m; ++i)
18 {
19     swap(a[1], a[n]); // 交换堆顶与最后一个元素
20     --n;
21     sink(1);          // 下沉调整 此时堆的大小为 i-1
22 }
```

堆排序算法 2 分析

- 时间复杂度：$O(N \log N)$
  - 下沉建堆：$O(N)$
  - 交换元素和堆调整：$O(N \log N)$
- 空间复杂度：$O(1)$，指的是排序过程中没有使用过多额外空间[3]。
- 一般说的堆排序算法指的是这种算法。

[3] 这种排序算法一般称之为原地排序算法。

STL 堆[4]

- STL 中没有提供heap容器，而是提供了堆的操作函数，诸如make_heap()、push_heap()、pop_heap()、sort_heap()等。
- 使用时，需用数组或容器 (建议用vector) 加上堆操作函数来实现堆。
- 使用堆操作函数必须要引入头文件`#include<algorithm>`。

操作	功能
make_heap()	建堆
push_heap()	插入元素
pop_heap()	删除堆顶元素
sort_heap()	堆排序

表: 堆操作函数

[4]感兴趣的读者请自行了解。

STL 堆


```
int a[9] = { 0, 1, 2, 3, 4, 8, 9, 3, 5 };
int n = 8;  // 数组有效长度
void print()
{
    for (int i = 1; i <= n; ++i) cout << a[i] << " ";
    cout << endl;
}

make_heap(a + 1, a + n + 1);                    // 最大堆
print();                                        // 输出: 9 8 3 5 2 3 1 4
make_heap(a + 1, a + n + 1, greater<int>());    // 最小堆
print();                                        // 输出: 1 2 3 4 8 3 9 5
a[++n] = 1;                                     // 先在数组中增加元素 然后执行入堆操作
push_heap(a + 1, a + n + 1, greater<int>());
print();                                        // 输出: 1 1 3 2 8 3 9 5 4
pop_heap(a + 1, a + n + 1, greater<int>());     // 并不是真的弹出元素而是放在了数组尾部
n--;                                            // 数组长度减少，等价于删除尾部元素
print();                                        // 输出: 1 2 3 4 8 3 9 5
// 因为建立了最小堆 所以只能从大到小排 如果需要从小到大排 需要建立最大堆
sort_heap(a + 1, a + n + 1, greater<int>());    // 堆排序，使用过后，堆序性将受到破坏
print();                                        // 输出: 9 8 5 4 3 3 2 1
```

STL 堆


```
int a[9] = {0, 1, 2, 3, 4, 8, 9, 3, 5};
vector<int> v(a + 1, a + 9); //定义向量，并复制数组进入
void print()            //打印向量中的元素
{
    for (int i = 0; i < v.size(); ++i) cout << v[i] << ' ';
    cout << endl;
}

make_heap(v.begin(), v.end(), greater<int>());  //最小堆
print();                                        // 输出: 1 2 3 4 8 9 3 5
make_heap(v.begin(), v.end());                  // 最大堆，参数为向量的开始和结束，默认
print();                                        // 输出: 9 8 3 5 2 3 1 4
v.push_back(7);                                 // 先在容器中添加新元素，再执行入堆操作
push_heap(v.begin(), v.end());
print();                                        // 输出: 9 8 3 7 2 3 1 4 5
pop_heap(v.begin(), v.end());                   // 并不是真的弹出元素而是放在了向量尾部
v.pop_back();                                   // 弹出向量的尾部元素，此时为最值
print();                                        // 输出: 8 7 3 5 2 3 1 4
// 因为建立了最大堆 所以只能从小到大排 如果需要从大到小排 需要建立最小堆
sort_heap(v.begin(), v.end());                  // 堆排序，使用过后，堆序性将受到破坏
print();                                        // 输出: 1 2 3 3 4 5 7 8
```