上海师范大学 Shanghai Normal University

《算法设计与分析》

5-堆和不相交数据结构 (Heap and Disjoint Set Data Structures)

杨启哲

上海师范大学信机学院计算机系

2024 年 10 月 17 日

主要内容

- 堆
- 不相交集数据结构

堆

数组中的操作

上海师范大学 Shanghai Normal University

给定两个数组 $A[1,\ldots,n]$ 和 $B[1,\ldots,n]$，其中 B 是一个从小到大排列的有序的数组。

- 在两个数组中分别寻找最大元需要多少时间?
- 向两个数组中分别插入一个元素需要多少时间?
- 在两个数组中分别删除最大元素需要多少时间?

优先队列 (Priority Queue)

支持插入和寻找最大值元素的数据结构称为优先队列。

Data Structure matters!

我们将介绍一种数据结构-堆，使得上述操作的时间复杂度都是 $O(\log n)$。

二叉树

二叉树

定义 1 **[二叉树].**

二叉树是顶点的一个有限集合，该集合或者为空，或者由一个根节点和两棵不相交的分别称为根节点的左子树和右子树的二叉树组成。

二叉树的种类

- 满二叉树：除了叶节点外，每个节点都有两个子节点的二叉树称为满二叉树。
- 完全二叉树：所有叶子在同一层的满二叉树称为完全二叉树。
- 几乎完全的二叉树：除了最后一层外，每一层都是满的，且最后一层上的叶子都尽可能地靠左的二叉树称为几乎完全的二叉树。

关于树的命名 (I)

上述的命名可能和一般的数据结构会有所区别，我们这里作进一步的解释。

我们从英文的命名来说明，考虑一颗 $\mathfrak{m}$-ary 树，其指的是“each node has no more than $\mathfrak{m}$ children”，即没有一个节点有超过 $\mathfrak{m}$ 个子节点。

而事实上，我们一般关注的有如下几种特殊的树：

- Full $\mathfrak{m}$-ary tree.
- Complete $\mathfrak{m}$-ary tree.
- Perfect $\mathfrak{m}$-ary tree.

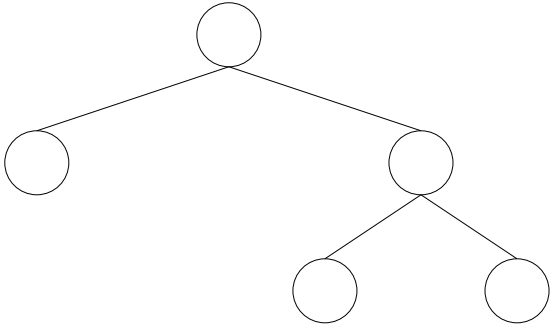
关于树的命名 (II)

上海师范大学 Shanghai Normal University

定义 2 **[Full m-ary tree].**

A full $\mathbf{m}$-ary tree is an m-ary tree where within each level every node has 0 or $\mathbf{m}$ children.

也就是说，每个节点要么是叶子节点，要么有 $\mathbf{m}$ 个子节点。

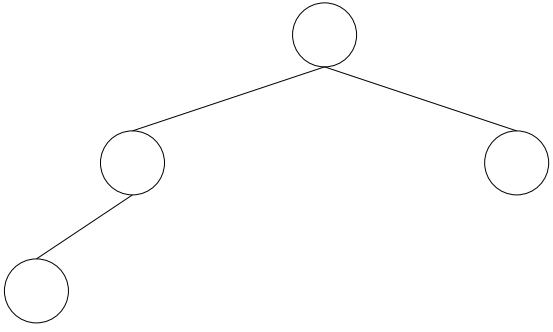


关于树的命名 (III)

定义 3 **[Complete m-ary tree].**

A complete m-nary tree is a m-nary tree in which every level, except possibly the last, is completely filled, and all nodes in the last level are as far left as possible.

也就是说，除了最后一层外，每一层都是满的，且最后一层上的叶子都尽可能地靠左的



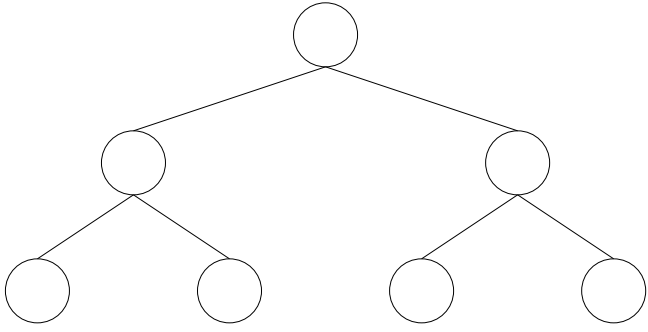
关于树的命名 (IV)

上海师范大学 Shanghai Normal University

定义 4 **[Perfect m-ary tree].**

A perfect $\mathbf{m}$-nary tree is a $\mathbf{m}$-nary tree in which all interior nodes have two children and all leaves have the same depth or same level.

也就是说，树所有的叶子节点都在同样的深度上，并且所有的非叶子节点都有 m 个子节点。



关于树的命名 (V)

可以看到：

1. 一颗 perfect $\mathfrak{m}$-nary tree 一定是 full $\mathfrak{m}$-nary tree，也是 complete $\mathfrak{m}$-nary tree。
2. full $\mathfrak{m}$-nary tree 和 complete $\mathfrak{m}$-nary tree 之间没有必然的联系。

英文的定义也有歧义！

事实上，关于 complete $\mathfrak{m}$-nary tree 和 perfect $\mathfrak{m}$-nary tree，英文的定义也有区别，有的定义中：

1. 将complete $\mathfrak{m}$-nary tree定义为perfect $\mathfrak{m}$-nary tree。
2. 上述定义中的complete $\mathfrak{m}$-nary tree称为nearly complete $\mathfrak{m}$-nary tree或者almost complete $\mathfrak{m}$-nary tree。

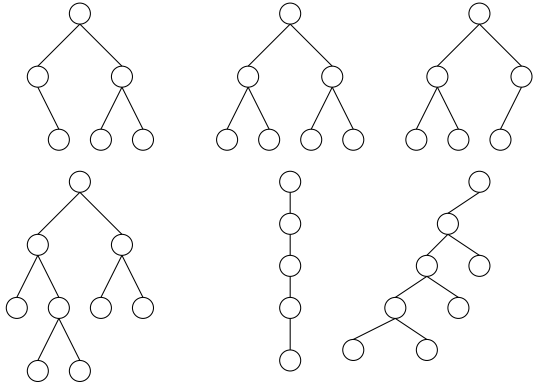
关于树的命名 (VI)

到这可以看到，关于中文，我们实际上是翻译的区别：

英文	本课教材 & 课件	数据结构课程 & 其他书籍
Full m-ary tree	满 m 叉树	正则 m 叉树
Complete m-ary tree	几乎完全的 m 叉树	完全 m 叉树
Perfect m-ary tree	完全 m 叉树	满 m 叉树

- 我个人还是更倾向于使用本课教材的命名方式。
- 大家也要注意，命名其实不关键，注意使用的 scope 即可。在相应的领域，重名其实是很常见的，还是要看定义的描述。

二叉树



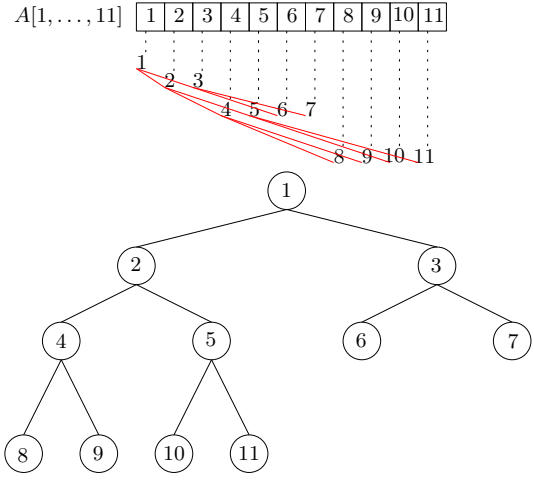
二叉树的性质

1. 在二叉树中，第 j 层的顶点数最多是 2^j。
2. 令二叉树 T 的顶点数是 n，高度是 h，则有：$n \leqslant \sum_{j=0}^{h} 2^j = 2^{h+1} - 1$。
3. 任何 n 个顶点的二叉树的高度至少是 $\lfloor \log n \rfloor$，最多是 $n - 1$。
4. 有 n 个顶点的几乎完全的或完全二叉树的高度是 $\lfloor \log n \rfloor$。

二叉树的数组表示

我们可以用数组 $A[1,\ldots,n]$ 来表示一颗 n 个顶点的几乎完全的或者完全的二叉树:

- 数组 $A[j]$ 中对应二叉树中的点的左子顶点和右子顶点分别是 $A[2j]$ 和 $A[2j+1]$。
- 数组 $A[j]$ 中对应二叉树中的父节点存储在 $A[\lfloor \frac{j}{2} \rfloor]$ 中。



定义 5 **[最大堆].**

一个最大堆是一颗几乎完全的二叉树，其每个节点满足如下性质：如果 v 和 $p(v)$ 是某个节点和其父节点存储的键值，则 $p(v) \geqslant v$。一个最大堆支持如下四种操作：

- $\mathrm{Insert}(H, x)$：将元素 x 插入到堆 H 中。
- $\mathrm{DeleteMax}(H)$: 删除并返回堆 H 中的最大元素。
- $\mathrm{Delete}(H, x)$: 删除堆 H 中的元素 x。
- $\mathrm{MakeHeap}(A)$: 将数组 A 转换为一个最大堆。

最小堆

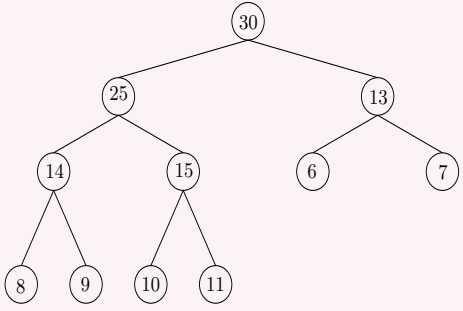
与之对应的还有最小堆的概念，即把定义中 $p(v) \geqslant v$ 的要求改为 $p(v) \leqslant v$。

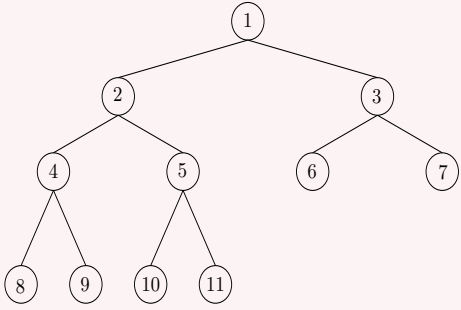
最大堆和最小堆的例子

上海师范大学 Shanghai Normal University

堆的例子

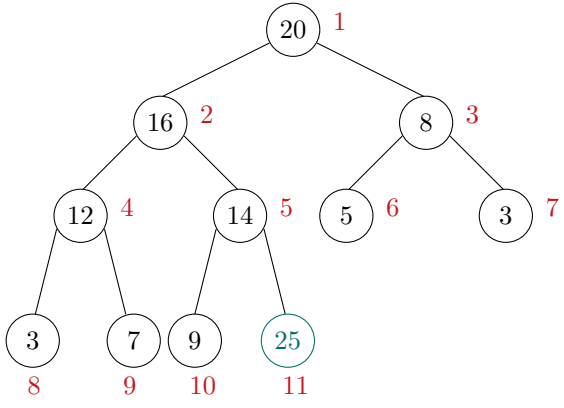
下图给出了一个最大堆和一个最小堆的例子:





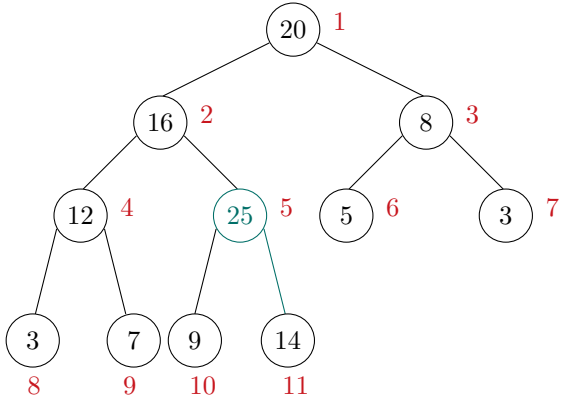
修复堆操作 SiftUp

如果一个堆中某个节点的值比父节点大了该如何处理?



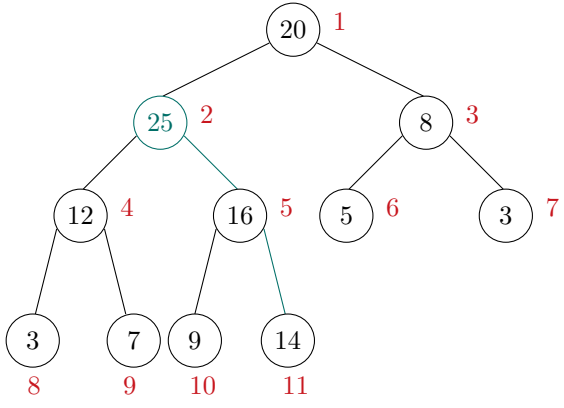
修复堆操作 SiftUp

如果一个堆中某个节点的值比父节点大了该如何处理?



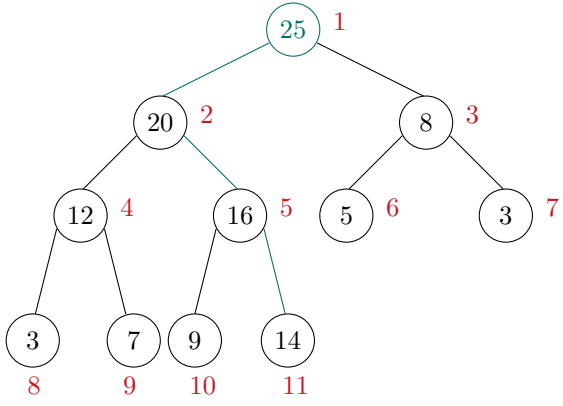
修复堆操作 SiftUp

如果一个堆中某个节点的值比父节点大了该如何处理?



修复堆操作 SiftUp

如果一个堆中某个节点的值比父节点大了该如何处理?



算法 SiftUp

上海师范大学 Shanghai Normal University

算法：SiftUp(H, i)

输入： 数组 H[1, . . . , n] 和 1 ~ n 之间的索引 i

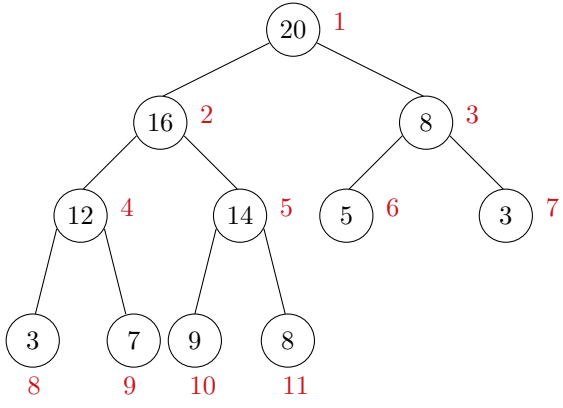
输出： 上移 H[i] 直到满足最大堆的性质

```
1:  done ← false
2:  if i = 1 then exit                          ▷ H[i] 是根节点
3:  repeat
4:      p ← ⌊i/2⌋                               ▷ p 是 H[i] 的父节点
5:      if H[i] ⩽ H[p] then
6:          done ← true
7:      else
8:          交换 H[i] 和 H[p]
9:          i ← p
10: until done or i = 1
```

$O(\log n)!$

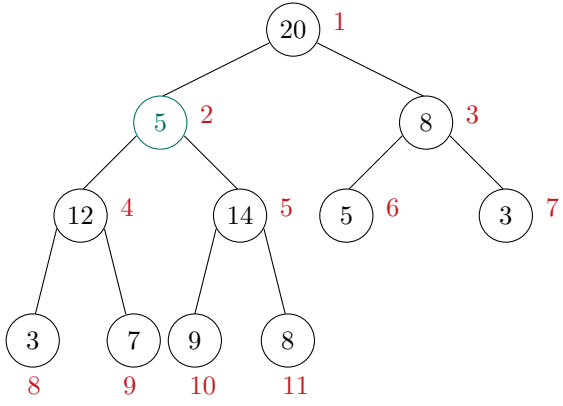
修复堆操作 SiftDown

如果一个堆中某个节点的值比其子节点了该如何处理?



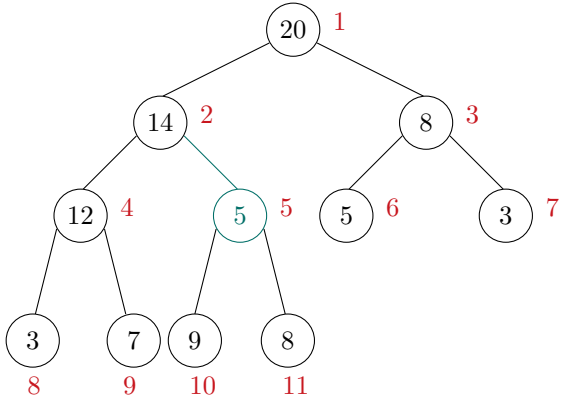
修复堆操作 SiftDown

如果一个堆中某个节点的值比其子节点了该如何处理?



修复堆操作 SiftDown

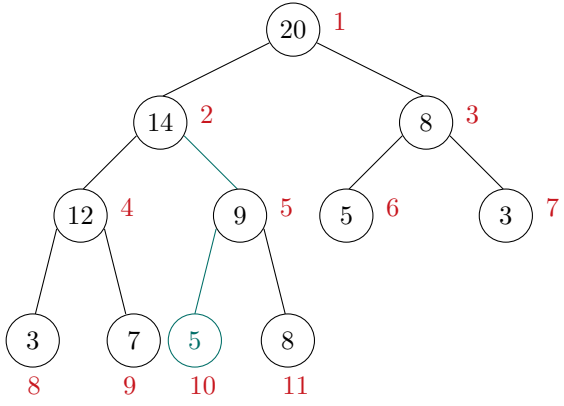
如果一个堆中某个节点的值比其子节点了该如何处理?



每次跟较大的子节点比较替换，直到渗透到合适的位置。

修复堆操作 SiftDown

如果一个堆中某个节点的值比其子节点了该如何处理?



每次跟较大的子节点比较替换，直到渗透到合适的位置。

算法 SiftDown

上海师范大学 Shanghai Normal University

算法：SiftDown(H, i)

输入： 数组 $H[1,\ldots,n]$ 和 $1 \sim n$ 之间的索引 i

输出： 下移 $H[i]$ 直到满足最大堆的性质

```
1:  done ← false
2:  if 2i > n then exit                          ▷ H[i] 是叶子节点
3:  repeat
4:      c ← 2i                                   ▷ c 是 H[i] 的左子节点
5:      if c < n and H[c + 1] > H[c] then
6:          c ← c + 1                            ▷ c 是 H[i] 的右子节点
7:      if H[i] ⩾ H[c] then
8:          done ← true
9:      else
10:         交换 H[i] 和 H[c]
11:         i ← c
12: until done or 2i > n
```

$O(\log n)!$

堆操作-插入 Insert

有了 SiftUp 之后堆的插入操作变得很容易。

- 将堆的大小加一。
- 将新元素 x 放置堆的末尾。
- 利用 SiftUp 调整 x 的位置。

算法： $\mathrm{Insert}(H, x)$

输入: 数组 $H[1, \ldots, n]$ 和元素 x

输出: 新的堆 $H[1, \ldots, n+1]$，包含 x

1: $n \leftarrow n + 1$
2: $H[n] \leftarrow x$
3: $\mathrm{SiftUp}(H, n)$

$O(\log n)!$

堆操作-删除 Delete

删除堆中的一个元素 $H[i]$ 也非常简单，可以用 $H[n]$ 替代后再调用 SiftDown 或者 SiftUp。

算法： $\mathrm{Delete}(H, x)$

输入: 数组 $H[1, \ldots, n]$ 和 $1 \sim n$ 之间的索引 i

输出: 新的堆 $H[1, \ldots, n-1]$，不包含 $H[i]$

1: $x \leftarrow H[i],\ H[i] \leftarrow H[n]$
2: $n \leftarrow n - 1$
3: **if** $i = n + 1$ **then exit**
4: **if** $x > H[i]$ **then**
5: $\quad \mathrm{SiftUp}(H, i)$
6: **else**
7: $\quad \mathrm{SiftDown}(H, i)$

$O(\log n)!$

堆操作-删除最大值 DeleteMax

注意到一个最大堆中，最大值一定是根节点 $H[1]$，因此删除最大值可以简化为删除根节点。

算法： $\mathrm{Delete}(H)$

输入: 数组 $H[1,\ldots,n]$

输出: 新的堆 $H[1,\ldots,n-1]$，不包含 $H[1]$，同时返回 $H[1]$

1: $x \leftarrow H[1]$
2: $\mathrm{Delete}(H,1)$
3: **return** x

$O(\log n)!$

创建堆 MakeHeap

通过转化为二叉树的形式，我们可以在 $O(\log n)$ 的时间内完成插入删除和寻找最大值的操作。

如何构造一个堆？

朴素的方法

对于一个含有 n 个元素的数组，我们可以想象开始是一个空堆，然后依次插入元素，直至所有元素都插入堆中。

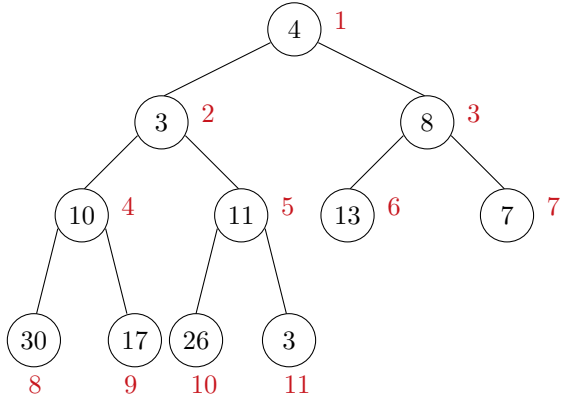
- 插入第 j 个元素的时间复杂度为 $O(\log j)$。
- 总共插入 n 个元素，因此总的时间复杂度为 $O(n \log n)$。

$O(n \log n)!$

O(n) 时间创建堆 MakeHeap

对于一个数组 $A[1,\ldots,n]$，其已经可以看成是一个几乎完全的二叉树。

因此，我们可以尝试直接在上面进行调整使其成为最大堆。



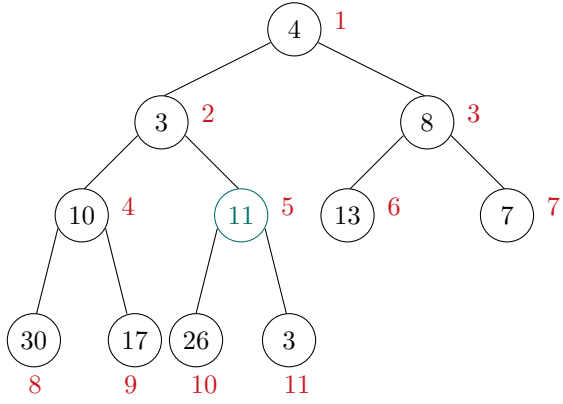
$A[1,\ldots,11]:$

4	3	8	10	11	13	7	30	17	26	3

O(n) 时间创建堆 MakeHeap

对于一个数组 $A[1,\ldots,n]$，其已经可以看成是一个几乎完全的二叉树。

因此，我们可以尝试直接在上面进行调整使其成为最大堆。



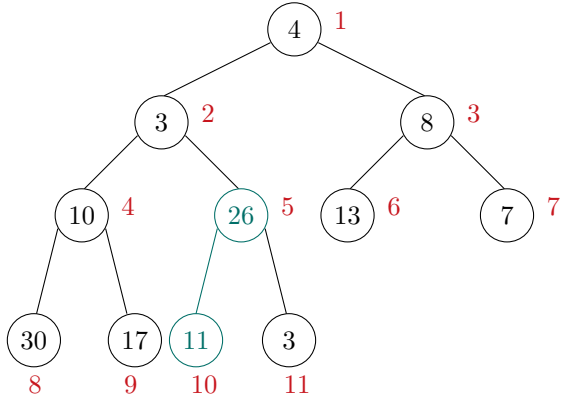
$A[1,\ldots,11]:$

4	3	8	10	11	13	7	30	17	26	3

O(n) 时间创建堆 MakeHeap

对于一个数组 $A[1,\ldots,n]$，其已经可以看成是一个几乎完全的二叉树。

因此，我们可以尝试直接在上面进行调整使其成为最大堆。



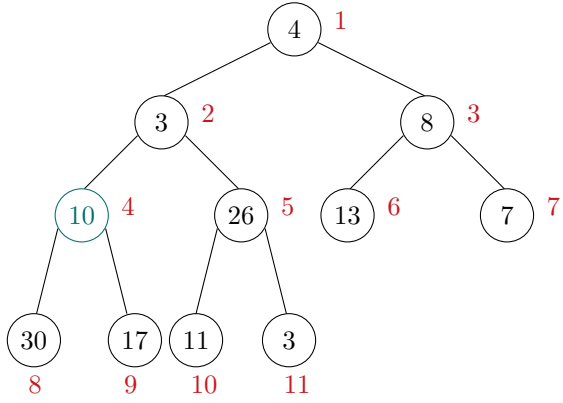
$A[1,\ldots,11]$:

4	3	8	10	26	13	7	30	17	11	3

O(n) 时间创建堆 MakeHeap

对于一个数组 $A[1,\ldots,n]$，其已经可以看成是一个几乎完全的二叉树。

因此，我们可以尝试直接在上面进行调整使其成为最大堆。



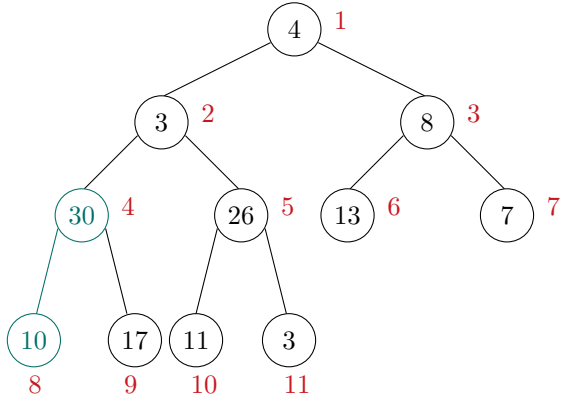
$A[1,\ldots,11]$:

4	3	8	10	26	13	7	30	17	11	3

O(n) 时间创建堆 MakeHeap

对于一个数组 $A[1,\ldots,n]$，其已经可以看成是一个几乎完全的二叉树。

因此，我们可以尝试直接在上面进行调整使其成为最大堆。



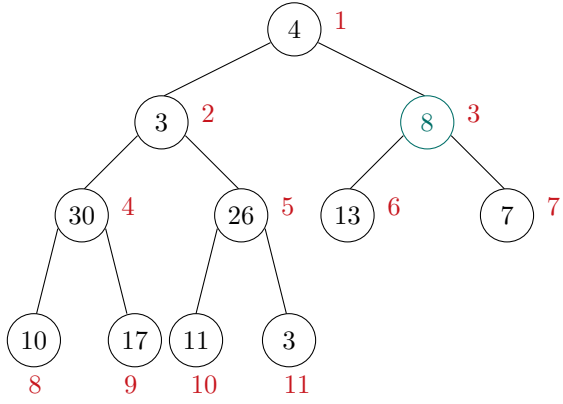
$A[1,\ldots,11]:$

4	3	8	30	26	13	7	10	17	11	3

O(n) 时间创建堆 MakeHeap

对于一个数组 $A[1,\ldots,n]$，其已经可以看成是一个几乎完全的二叉树。

因此，我们可以尝试直接在上面进行调整使其成为最大堆。



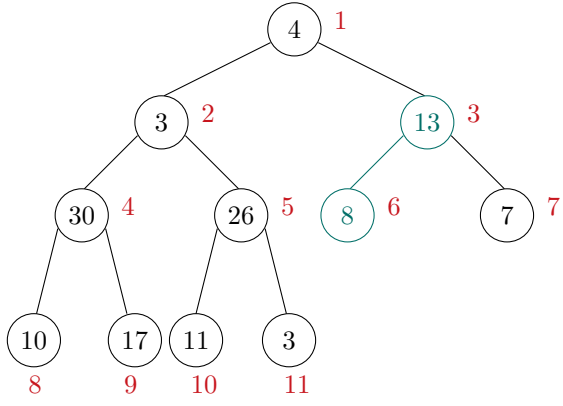
$A[1,\ldots,11]$:

4	3	8	30	26	13	7	10	17	11	3

O(n) 时间创建堆 MakeHeap

对于一个数组 $A[1,\ldots,n]$，其已经可以看成是一个几乎完全的二叉树。

因此，我们可以尝试直接在上面进行调整使其成为最大堆。



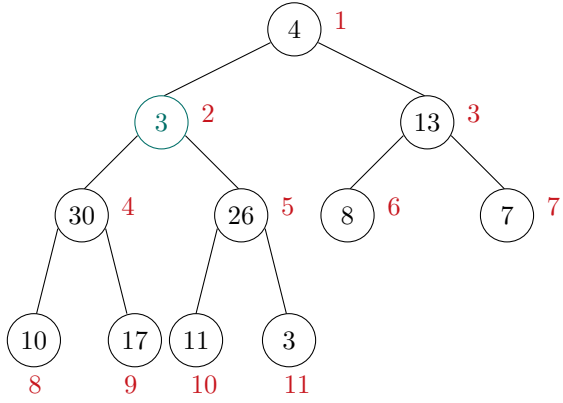
$A[1,\ldots,11]:$

4	3	13	30	26	8	7	10	17	11	3

O(n) 时间创建堆 MakeHeap

对于一个数组 $A[1,\ldots,n]$，其已经可以看成是一个几乎完全的二叉树。

因此，我们可以尝试直接在上面进行调整使其成为最大堆。



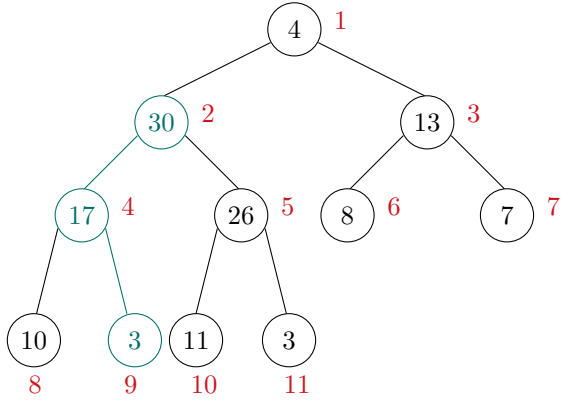
$A[1,\ldots,11]$:

4	3	13	30	26	8	7	10	17	11	3

O(n) 时间创建堆 MakeHeap

对于一个数组 $A[1,\ldots,n]$，其已经可以看成是一个几乎完全的二叉树。

因此，我们可以尝试直接在上面进行调整使其成为最大堆。



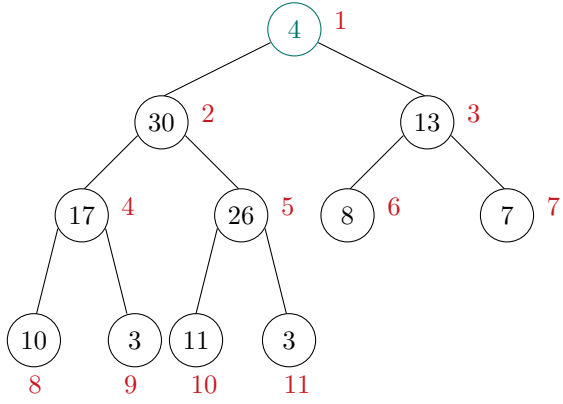
$A[1,\ldots,11]:$

4	30	13	17	26	8	7	10	3	11	3

O(n) 时间创建堆 MakeHeap

对于一个数组 $A[1,\ldots,n]$，其已经可以看成是一个几乎完全的二叉树。

因此，我们可以尝试直接在上面进行调整使其成为最大堆。



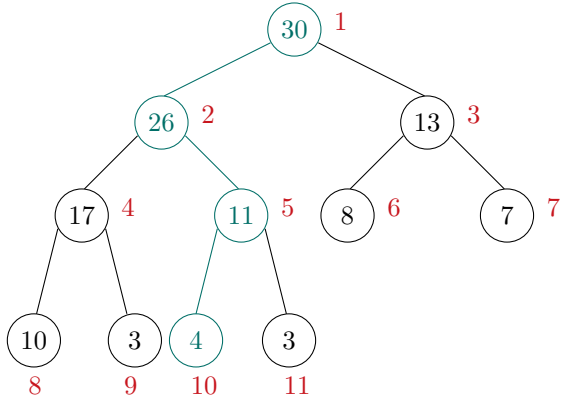
$A[1,\ldots,11]:$

4	30	13	17	26	8	7	10	3	11	3

O(n) 时间创建堆 MakeHeap

对于一个数组 $A[1,\ldots,n]$，其已经可以看成是一个几乎完全的二叉树。

因此，我们可以尝试直接在上面进行调整使其成为最大堆。



$A[1,\ldots,11]:$

30	26	13	17	11	8	7	10	3	4	3

算法 MakeHeap

上海师范大学 Shanghai Normal University

算法 MakeHeap(A)

输入: 数组 $A[1, \ldots, n]$

输出: 数组 $A[1, \ldots, n]$ 成为最大堆

```
1: i ← ⌊n/2⌋
2: while i ⩾ 1 do
3:     SiftDown(A, i)
4:     i ← i − 1
```

算法 MakeHeap 分析

- 令 T 为数组 $A[1,\ldots,n]$ 对应的几乎完全的二叉树, 高度为 h。对于第 i 层的点，其调用 SiftDown 时产生的循环次数为 $h-i$。
- 注意到第 i 层的点在数组中处在 $A[2^i,\ldots,2^{i+1}-1]$ 中，循环执行的总次数上界为:

$$\sum_{i=0}^{h-1}(h-i)\cdot 2^i=\sum_{i=1}^{h}i2^{h-i}=2^h\sum_{i=1}^{h}\frac{i}{2^i}<2n$$

- SiftDown 中的每次循环至多产生两次元素的比较，梅调用一次 SiftDown 至少执行一次循环。

定理 6.

算法 MakeHeap 的时间复杂度为 $O(n)$。令 $C(n)$ 表示 SiftDown 的比较次数, 则 $n-1\leqslant C(n)<4n$。

堆和数组的比较

现在我们来比较一下堆和数组操作上的区别：

	最大值	插入	删除最大值
数组	$O(n)$	$O(1)$	$O(n)$
有序数组	$O(1)$	$O(n)$	$O(1)$
堆	$O(1)$	$O(\log n)$	$O(\log n)$

堆应用-堆排序

最大堆的一个简单运用便是排序。在创建好堆后，不停的删除最大值，即可得到一个有序的序列。

堆排序 $\mathrm{HeapSort}(A)$

输入: 数组 $A[1,\ldots,n]$

输出: 数组 $A[1,\ldots,n]$ 升序排列

1: $\mathrm{MakeHeap}(A)$
2: **for** $i \leftarrow n$ downto 2 **do**
3: 交换 $A[1]$ 与 $A[i]$
4: $\mathrm{SiftDown}(A[1,\ldots,j-1],1)$

$O(n \log n)!$

堆的其他应用举例

- 优先队列的实现方式。
- 数据压缩，如 Huffman 编码。
- 图上的搜索算法，如 Dijkstra 算法，Prim 算法。
- 统计方面，如保持最大的 k 个数。
- 系统方面，如负载均衡等。
- …

不相交集数据结构

不相交数据结构

现在我们来考察这样一个结构。假设有一个 n 个不同元素的集合 S，最初令每个元素自成一个集合。每个集合选取其中一个元素作为其独特的标识，并且令这种标识是唯一的。我们定义如下两个操作：

- $\mathrm{Find}(x)$：返回 x 所在的集合表示。
- $\mathrm{Union}(x, y)$：将包含 x 和 y 的两个集合合并成一个集合。新的集合标识为原来两个集合的标识之一。

通过一个 m 次 Find 和 Union 的操作序列 σ 后，S 会被分解成若干个不相交的集合。一个自然的问题是执行一个 m 次 Find 和 Union 的操作序列 σ 需要多久？

例 7.

考察集合 $S = \{1, 2, 3, 4, 5\}$，在经历了如下操作序列 σ:

$$\mathrm{Union}(1, 2); \mathrm{Union}(3, 4); \mathrm{Find}(2); \mathrm{Find}(4); \mathrm{Union}(2, 5)$$

集合 S 被成了两个不相交集 $\{1, 2, 5\}$ 和 $\{3, 4\}$，我们可以用 $1, 4$ 来分别标识这两个集合。

不相交数据结构应用-连通性问题

不相交数据结构的一个重要应用是解决**连通性问题**。这相当于问在一个操作序列 σ 后，两个元素 p, q 是否在同一个集合中，记该操作为 $\mathrm{Connected}(p, q)$。

例 8.

考察集合 $S = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$，在经历了如下操作序列 σ:

$$\mathrm{Union}(4, 3); \mathrm{Union}(3, 8); \mathrm{Union}(6, 5); \mathrm{Union}(9, 4); \mathrm{Union}(2, 1)$$

我们可以询问 $\mathrm{Connected}(0, 7)$，$\mathrm{Connected}(8.9)$，即其是否在一个集合里。

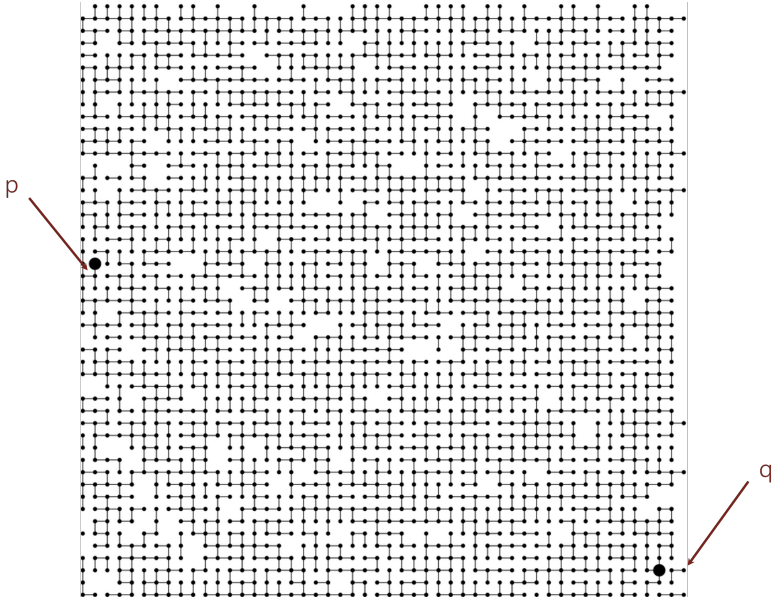
这个过程可以是连续的，我们也可以接着执行:

$$\mathrm{Union}(5, 0); \mathrm{Union}(7, 2); \mathrm{Union}(6, 1)$$

再询问 $\mathrm{Connected}(0, 7)$。

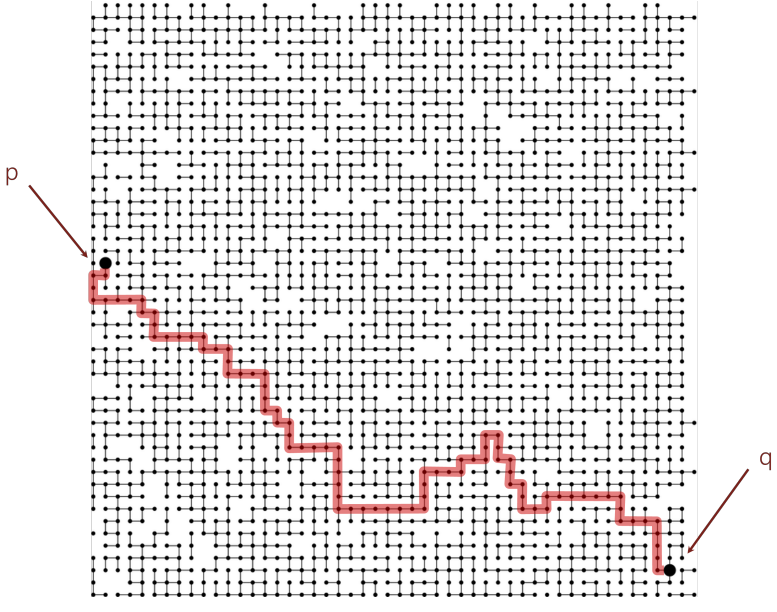
不相交数据结构应用-更复杂的连通性问题

连通性问题的一个应用即迷宫找路。



不相交数据结构应用-更复杂的连通性问题

连通性问题的一个应用即迷宫找路。



不相交数据结构的表示方法-数组表示法

上海师范大学 Shanghai Normal University

数组表示法

数组表示显得非常直观简单。我们可以用一个数组 $A[1,\ldots,n]$ 来表示 n 个元素，其中 $A[i]$ 表示元素 i 所在的集合标识。

例 9.

对于一个不相交的集合 $1:\{1,7,10,11\},\ 3:\{2,3,5,6\},\ 8:\{4,8,9\}$, 我们可以表示为：

	1	2	3	4	5	6	7	8	9	10	11
$A[]:$	1	3	3	8	3	3	1	8	8	1	1

数组表示的实现

Union 与 Find

- Find(x): 返回 A[x]。
- Union(x, y): 将所有值为 A[x] 的值 A[i] 赋为 A[y]。

经过 n 次操作后，

- Find(x) 所需的时间: O(1).
- Union(x, y) 所需的时间: O(n).

因此执行一个m 次 Find 和 Union 混合的操作序列 σ 需要O(mn)的时间。

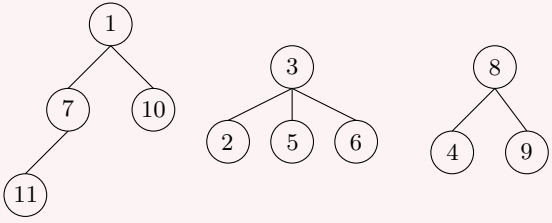
不相交数据结构的表示方法-树表示法

树表示法

树表示法，即每个集合用**一颗树**来表示，其根即为集合的标识。每个元素 x 拥有一个指针指向其父节点 $p(x)$，而根节点则是空指针。对于一个任意一个元素 x，我们使用 $root(x)$ 表示其根。注意树也可以用数组来表示，即每个元素 x 的父节点为 $A[x]$，而根节点的父节点为 0。

例 10.

对于一个不相交的集合 $1:\{1,7,10,11\}$, $3:\{2,3,5,6\}$, $8:\{4,8,9\}$, 我们可以表示为：



	1	2	3	4	5	6	7	8	9	10	11
$A[]:$	0	3	0	8	3	3	1	0	8	1	7

Union 与 Find

- Find(x): 跟随指针直到根节点 root(x)。
- Union(x, y): 将 root(x) 的指针指向 root(y)。

经过 n 次操作后，

- Find(x) 所需的时间: O(n).
- Union(x, y) 所需的时间: O(n).

因此执行一个m 次 Find 和 Union 混合的操作序列 σ 需要O(mn)的时间。

Union 慢的原因

对于 n 个元素的初始情况，考虑下面一个操作序列：

$$\texttt{Union}(1,2);\texttt{Union}(2,3);\ldots;\texttt{Union}(n-1,n)$$

此时树退化成了**链表**，从而之后的每次 Find 最坏都需要 $O(n)$ 的时间。

改进方法-按秩合并

从上述例子可以看到，要解决这个问题，我们需要控制树的**高度**，不让其过于倾斜。

因此我们在节点 x 处引入$\mathrm{rank}(x)$的概念:

- 初始时每个节点 x 满足 $\mathrm{rank}(x) = 0$。
- 当执行 $\mathrm{Union}(x, y)$，时比较 x 与 y 的 rank:
  - $\mathrm{rank}(x) < \mathrm{rank}(y)$, 将 x 的父节点指向 y。
  - $\mathrm{rank}(x) > \mathrm{rank}(y)$, 将 y 的父节点指向 x。
  - $\mathrm{rank}(x) = \mathrm{rank}(y)$, 将 x 的父节点指向 y，并且 $\mathrm{rank}(y) \leftarrow \mathrm{rank}(y) + 1$。

需要观察到的是根节点 x 的秩其实是树的高度。

引理 11.

包括根节点在内的树中的节点的个数**至少**是 $2^{\mathrm{rank}(x)}$。

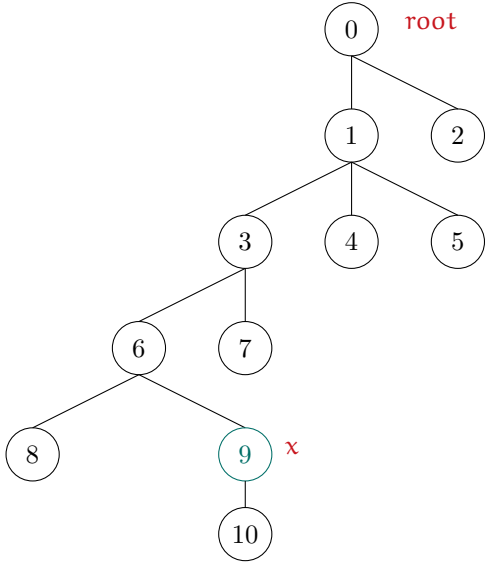
从而经过 n 次操作后，

- $\mathrm{Find}(x)$ 所需的时间: $O(\log n)$.
- $\mathrm{Union}(x, y)$ 所需的时间: $O(\log n)$.

因此执行一个m 次 Find 和 Union 混合的操作序列 σ 需要$O(m \log n)$的时间。

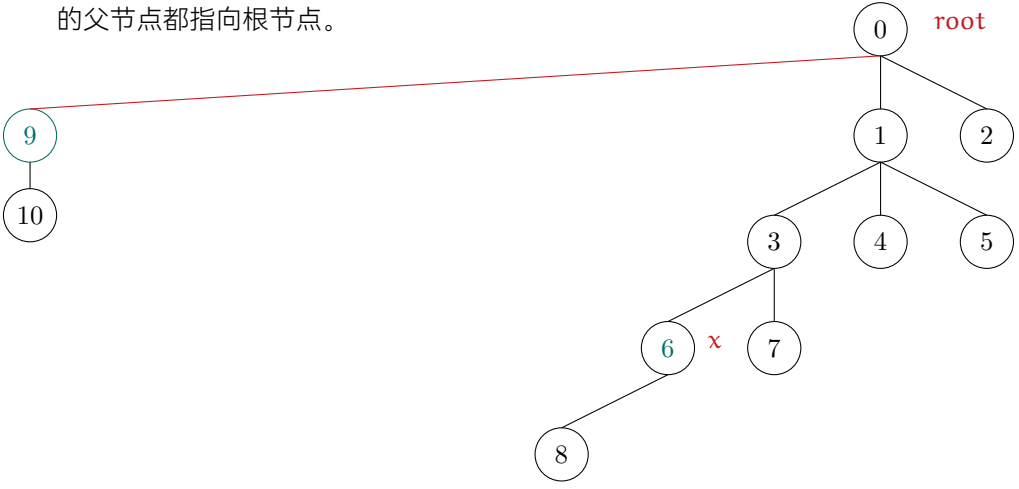
进一步改进-路径压缩

尽管我们已经通过秩的思路改进了树的高度，但事实上在 Union 与 Find 操作里我们并只关心根节点的位置。因此我们可以进一步压缩路径，即将 x 到根节点的路径上的所有节点的父节点都指向根节点。



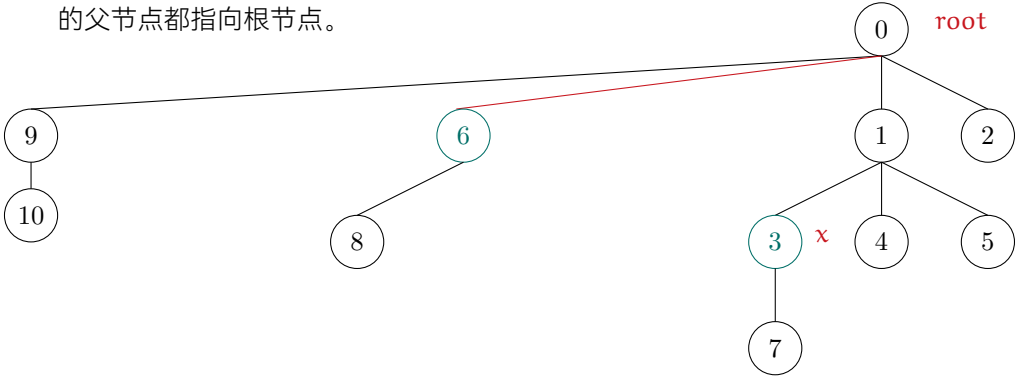
进一步改进-路径压缩

尽管我们已经通过秩的思路改进了树的高度，但事实上在 Union 与 Find 操作里我们并只关心根节点的位置。因此我们可以进一步压缩路径，即将 x 到根节点的路径上的所有节点的父节点都指向根节点。



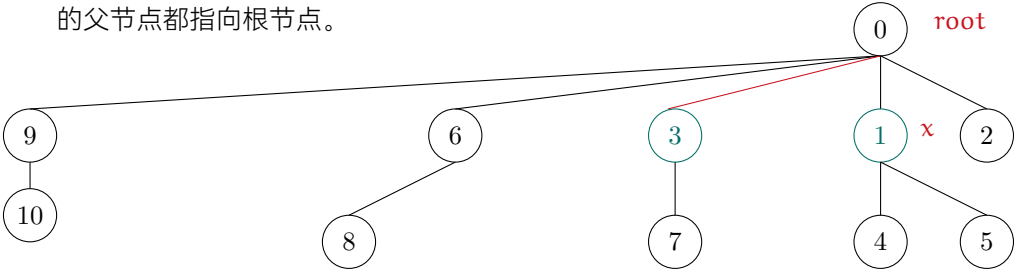
进一步改进-路径压缩

尽管我们已经通过秩的思路改进了树的高度，但事实上在 Union 与 Find 操作里我们并只关心根节点的位置。因此我们可以进一步压缩路径，即将 x 到根节点的路径上的所有节点的父节点都指向根节点。



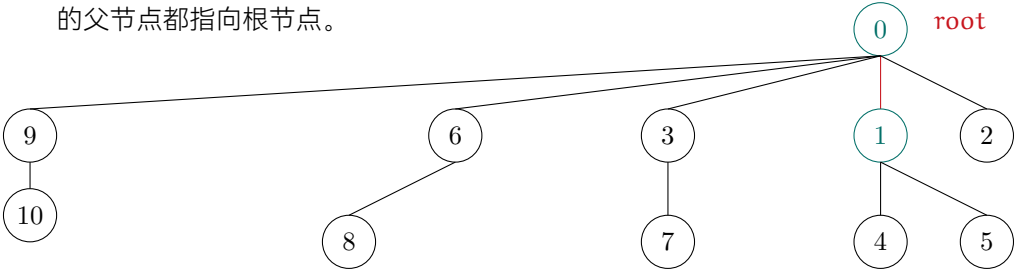
进一步改进-路径压缩

尽管我们已经通过秩的思路改进了树的高度，但事实上在 Union 与 Find 操作里我们并只关心根节点的位置。因此我们可以进一步压缩路径，即将 x 到根节点的路径上的所有节点的父节点都指向根节点。



进一步改进-路径压缩

尽管我们已经通过秩的思路改进了树的高度，但事实上在 Union 与 Find 操作里我们并只关心根节点的位置。因此我们可以进一步压缩路径，即将 x 到根节点的路径上的所有节点的父节点都指向根节点。



路径压缩可以在 Find 中完成。

路径压缩下的算法 Find

上海师范大学 Shanghai Normal University

算法 Find(x)

输入: 节点 x

输出: $root(x)$

```
1: y ← x
2: while p(y) ≠ null do
3:     y ← p(y)
4: root ← y, y ← x
5: while p(y) ≠ null do
6:     w ← p(y)
7:     p(y) ← root
8:     y ← w
9: return root
```

路径压缩下的算法 Union

算法 $\mathrm{Union}(x, y)$

输入: 节点 x, y

输出: 包含 x, y 的两棵树的合并

1: $u \leftarrow \mathrm{Find}(x)$, $v \leftarrow \mathrm{Find}(y)$
2: **if** $\mathrm{rank}(u) \leqslant \mathrm{rank}(v)$ **then**
3: $\quad p(u) \leftarrow v$
4: $\quad$ **if** $\mathrm{rank}(u) = \mathrm{rank}(v)$ **then**
5: $\quad\quad \mathrm{rank}(v) \leftarrow \mathrm{rank}(v) + 1$
6: **else**
7: $\quad p(v) \leftarrow u$

路径压缩下的算法执行时间

我们不给出详细的证明，有兴趣的同学可以阅读教材 3.3.4。

定理 12.

在路径压缩下的实现中，m 个 Find 与 Union 运算的序列所需的运行时间为 $O(n + m\log^* n)$，这里 $\log^*$ 的定义为：

$$\log^* n = \begin{cases} 0 & n \leqslant 1 \\ 1 + \log^*(\log n) & n > 1 \end{cases}$$

$\log^*$ 的大小

事实上 $\log^*$ 非常接近线性：

n	$\log^* n$
16	3
65536	4
2^{65536}	5

惊奇的是，可以证明不存在**线性时间**的算法！

总结

本节内容

- 堆
  - 堆的概念。
  - 堆操作的实现，创建堆。
  - 堆的应用，堆排序。
- 不相交数据结构
  - 不相交数据结构的概念。
  - 不相交数据结构的实现, 按秩合并，路径压缩。