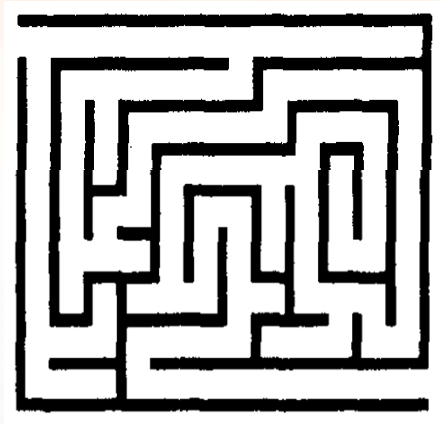
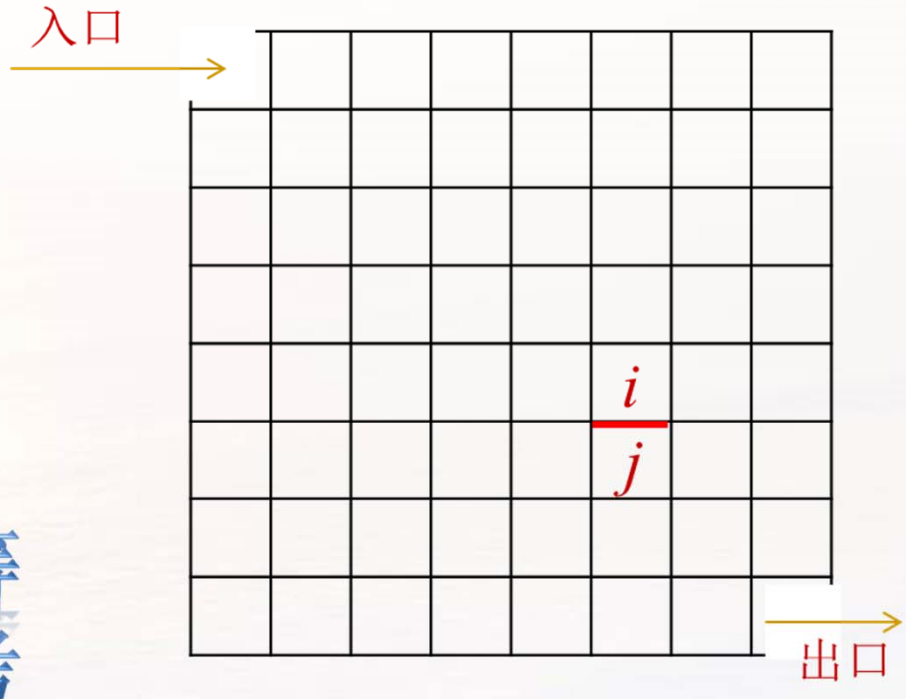
计算机问题求解－论题2-14
-用于动态等价关系的数据结构

2019年05月27日

Part I
Union-Find





问题1：

你能否基于动态等价关系的概念来考虑如何“建”迷宫？

问题2:

你认为建立迷宫需要什么样的“基本”操作？

如何判定一个对象属于哪个等价类?
如何将两个等价类合并为一个?

UnionFind – 一种抽象数据类型

MAKE-SET(x) creates a new set whose only member (and thus representative) is x. Since the sets are disjoint, we require that x not already be in some other set.

UNION(x, y) unites the dynamic sets that contain x and y, say S_x and S_y, into a new set that is the union of these two sets. We assume that the two sets are disjoint prior to the operation. The representative of the resulting set is any member of $S_x \cup S_y$, although many implementations of UNION specifically choose the representative of either S_x or S_y as the new representative. Since we require the sets in the collection to be disjoint, conceptually we destroy sets S_x and S_y, removing them from the collection $\mathcal{S}$. In practice, we often absorb the elements of one of the sets into the other set.

FIND-SET(x) returns a pointer to the representative of the (unique) set containing x.

问题3:

什么是动态集合的representative?

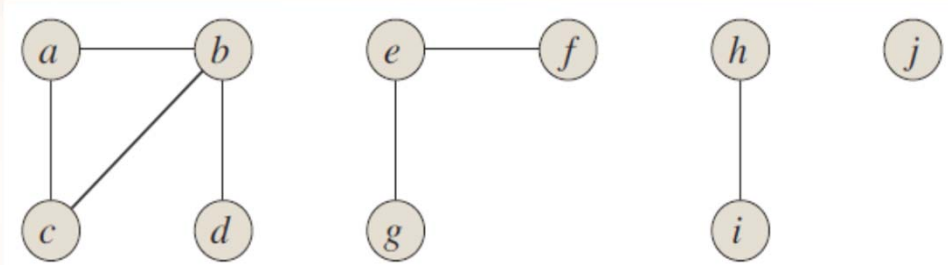
讨论数学与讨论数据结构时它有什么差别?

we care only that if we ask for the representative of a dynamic set twice without modifying the set between the requests, we get the same answer both times.

问题4:

我们讨论的不是一个算法，而是一个数据结构，那所谓“时间复杂性分析”究竟是什么意思呢？

n次MakeSet, m次各种操作（三种）序列的代价。



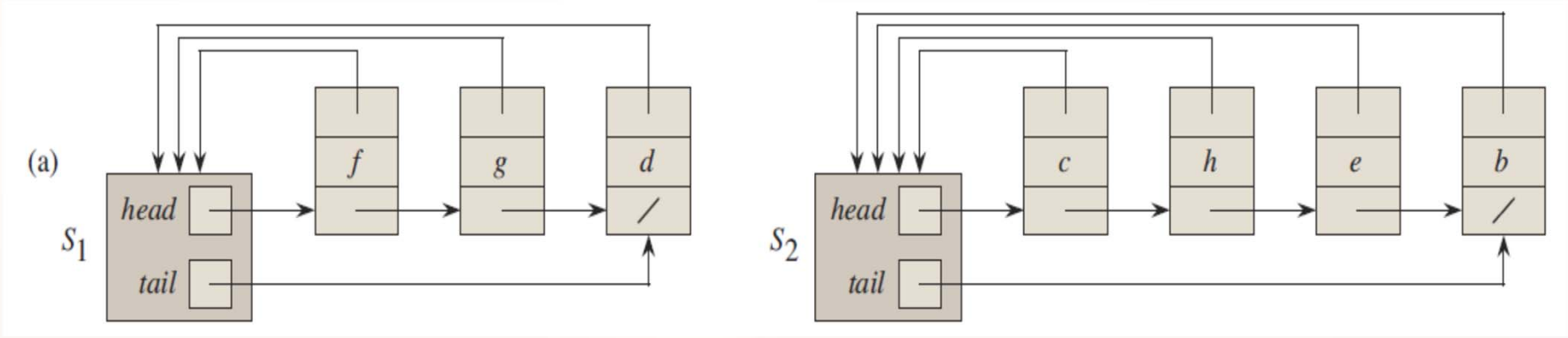

```
CONNECTED-COMPONENTS(G)
1  for each vertex v ∈ G.V
2      MAKE-SET(v)
3  for each edge (u, v) ∈ G.E
4      if FIND-SET(u) ≠ FIND-SET(v)
5          UNION(u, v)

SAME-COMPONENT(u, v)
1  if FIND-SET(u) == FIND-SET(v)
2      return TRUE
3  else return FALSE
```

Edge processed	Collection of disjoint sets									
initial sets	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}	{j}
(b,d)	{a}	{b,d}	{c}		{e}	{f}	{g}	{h}	{i}	{j}
(e,g)	{a}	{b,d}	{c}		{e,g}	{f}		{h}	{i}	{j}
(a,c)	{a,c}	{b,d}			{e,g}	{f}		{h}	{i}	{j}
(h,i)	{a,c}	{b,d}			{e,g}	{f}		{h,i}		{j}
(a,b)	{a,b,c,d}				{e,g}	{f}		{h,i}		{j}
(e,f)	{a,b,c,d}				{e,f,g}			{h,i}		{j}
(b,c)	{a,b,c,d}				{e,f,g}			{h,i}		{j}

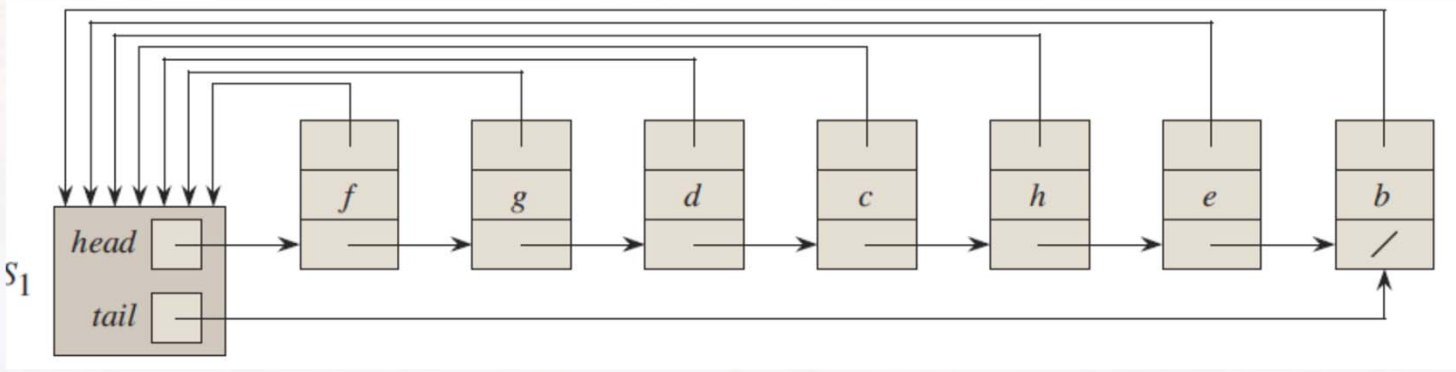
问题5：

假如我们想知道原来未直接相连的两个顶点连起来是否会形成回路，应该如何解决？

Implementing by Linked-List



操作union(g,e)执行后



问题6：

为什么用链表实现，每个操作的平均代价可能会是线性的？

Union操作对象的次序不影响结果，
却影响效率，这对你有什么启发?

问题7：

为什么weighted-union能降低操作平均代价？

A single UNION operation can still take $\Omega(n)$ time if both sets have $\Omega(n)$ members.

顺便问一句：weight指什么？

Theorem 21.1

Using the linked-list representation of disjoint sets and the weighted-union heuristic, a sequence of m MAKE-SET, UNION, and FIND-SET operations, n of which are MAKE-SET operations, takes $O(m + n \lg n)$ time.

问题8：

你能否说出此定理证明最核心的思想？

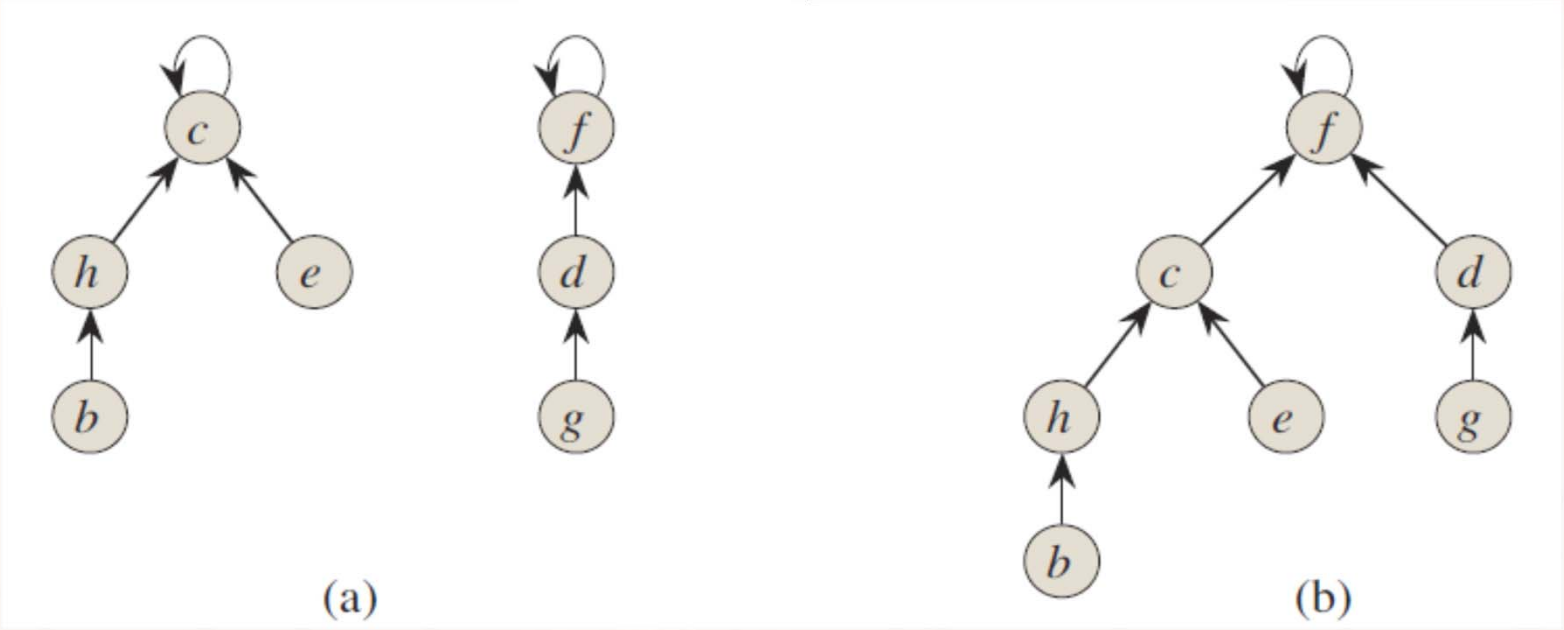
最关键的代价是union操作时元素指向head的指针更新的次数，但每个元素可能更新的总次数是有上限的。

因为每执行一次union, x所属集合的大小至少增大一倍。
集合最大含n个object, 因此x经历union的次数最多$\lceil \lg n \rceil$。

问题9：

书上是如何引入更好的数据结构来进一步改进算法的？

ADT inTree 和 Disjoint-set Forest



问题10:

这些树和前面介绍的搜索树有什么不同？你认为不同的算法意义在哪里？

问题11：

你觉得disjoint-set forest中的树结构性质中哪些只与操作代价有关，却与操作结果无关？这对你有什么启示？

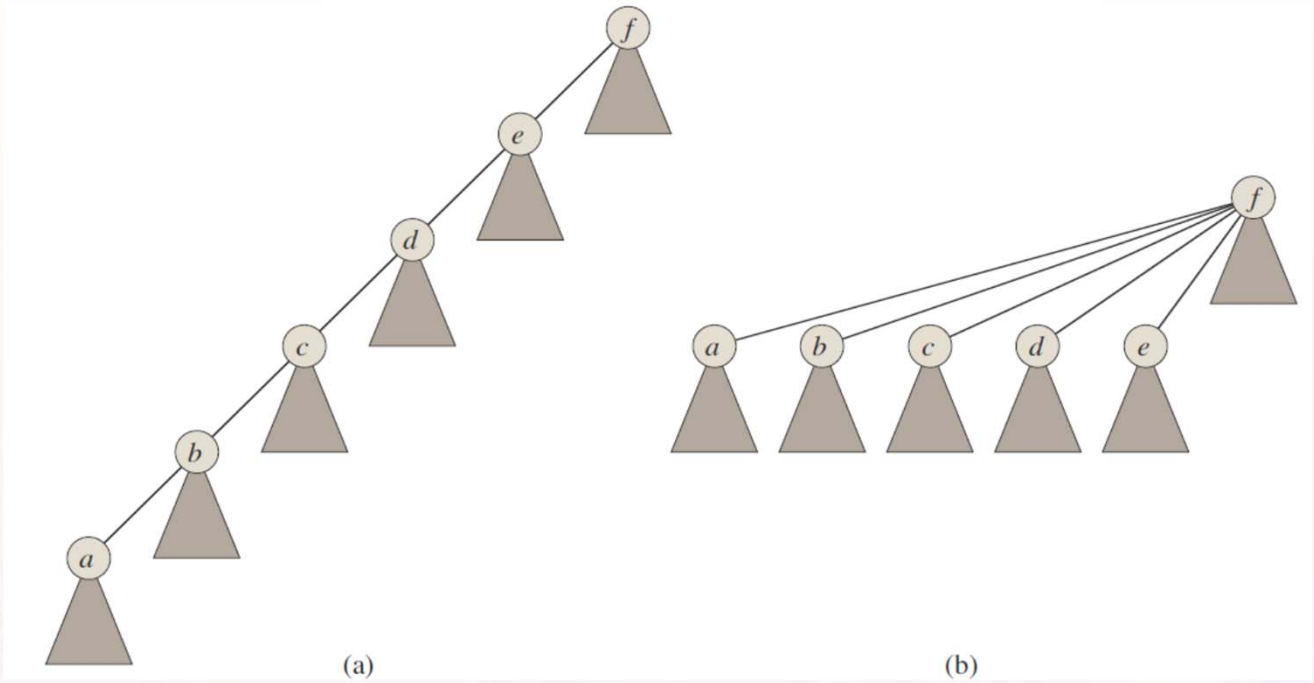
Union by Rank

问题12：

你能比较一下链表实现中的weighted Union和这里的Union by Rank吗？

*rank*究竟是什么？

Path Compression



问题13：
好处在
哪里？

Heuristic:只是“想当然”

注意：
rank 如何定义与修改。

```
MAKE-SET(x)
1  x.p = x
2  x.rank = 0
```

问题14：

路径“压缩”在何处实现？

```
UNION(x, y)
1  LINK(FIND-SET(x), FIND-SET(y))
```

```
LINK(x, y)
1  if x.rank > y.rank
2        y.p = x
3  else x.p = y
4        if x.rank == y.rank
5              y.rank = y.rank + 1
```

```
FIND-SET(x)
1  if x ≠ x.p
2        x.p = FIND-SET(x.p)
3  return x.p
```

rank, which is an upper bound on the height of the node

Part II
Amortized Analysis

Counter value	A[7]	A[6]	A[5]	A[4]	A[3]	A[2]	A[1]	A[0]	Total cost
0	0	0	0	0	0	0	0	0	0
1	0	0	0	0	0	0	0	1	1
2	0	0	0	0	0	0	1	0	3
3	0	0	0	0	0	0	1	1	4
4	0	0	0	0	0	1	0	0	7
5	0	0	0	0	0	1	0	1	8
6	0	0	0	0	0	1	1	0	10
7	0	0	0	0	0	1	1	1	11
8	0	0	0	0	1	0	0	0	15
9	0	0	0	0	1	0	0	1	16
10	0	0	0	0	1	0	1	0	18
11	0	0	0	0	1	0	1	1	19
12	0	0	0	0	1	1	0	0	22
13	0	0	0	0	1	1	0	1	23
14	0	0	0	0	1	1	1	0	25
15	0	0	0	0	1	1	1	1	26
16	0	0	0	1	0	0	0	0	31

```
INCREMENT(A)
1  i = 0
2  while i < A.length and A[i] == 1
3      A[i] = 0
4      i = i + 1
5  if i < A.length
6      A[i] = 1
```

问题15：
你能否根据这个例子解释一下：逐个操作考虑的worst-case分析结果可能很“粗糙”？

问题16：

同样的操作，有时代价很小，有时又很大，为什么？背后有什么更深层的原因使得分析方法有可能更合理些？

代价大的情况发生频度是受限制的，而且这个限制于小代价操作的数量有关。

Amortized Analysis: Aggregate方法

```
INCREMENT(A)
1  i = 0
2  while i < A.length and A[i] == 1
3      A[i] = 0
4      i = i + 1
5  if i < A.length
6      A[i] = 1
```

我们考虑一个k位的二进计数器，连续执行n次increment操作的总代价 (初始值为0)

显然：$A[i]$ (i=0,1,2,…k-1) 在“每k次”操作中只被flip一次。所以，总操作次数为：

换句话说：最坏情况下，操作平均代价为常量

$$\sum_{i=0}^{k-1} \left\lfloor \frac{n}{2^i} \right\rfloor < n \sum_{i=0}^{\infty} \frac{1}{2^i} = 2n ,$$

问题17：

这为什么不是“average case”?

Amortized Analysis: Accounting方法

```
INCREMENT(A)
1  i = 0
2  while i < A.length and A[i] == 1
3      A[i] = 0
4      i = i + 1
5  if i < A.length
6      A[i] = 1
```

同样考虑*n*个increment操作的序列。
不简单计算“总价”，而是针对不同的操作（或者同一操作的不同情况）采用不同的“记账”方式。

按照新的“记账”方式计算的代价称为“accounting cost”，
如果希望accounting cost能作为实际代价的上限，则必须保证操作序列过程中的任何适合，实际代价不大于accounting cost。
(这相当于为了未来开支预先存些钱)

While循环外的flip(line 6)为**set**操作，而while循环内的flip(line 3)为**reset**操作。
则accounting cost指定如下：**set**: 2 **reset**: 0

Amortized Analysis: Potential方法

```
INCREMENT(A)
1  i = 0
2  while i < A.length and A[i] == 1
3      A[i] = 0
4      i = i + 1
5  if i < A.length
6      A[i] = 1
```

不是为每个操作预先设置“备用金”，而是在操作序列中积累（或释放）“势能”。将执行完第i次操作后整个结构的“势能”定义为$\Phi(D_i)$, 每一步操作(不论是什么操作)的amortized cost为：

$$\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1}) .$$

这里将执行第i次操作后计数器中1的个数b_i定义为“势能”值。起始势能为0。

注意：

$$\begin{aligned}\sum_{i=1}^{n} \hat{c}_i &= \sum_{i=1}^{n} (c_i + \Phi(D_i) - \Phi(D_{i-1})) \\ &= \sum_{i=1}^{n} c_i + \Phi(D_n) - \Phi(D_0) .\end{aligned}$$

只要：$\Phi(D_n) \geq \Phi(D_0)$

Amortized cost就可以作为实际代价的上限。

注意：如果将第i次操作中置0(reset)的位数记为t_i，则$b_i \leq b_{i-1} - t_i + 1$

$$\begin{aligned}\Phi(D_i) - \Phi(D_{i-1}) &\leq (b_{i-1} - t_i + 1) - b_{i-1} \\ &= 1 - t_i .\end{aligned}$$

$$\begin{aligned}\hat{c}_i &= c_i + \Phi(D_i) - \Phi(D_{i-1}) \\ &\leq (t_i + 1) + (1 - t_i) \\ &= 2 .\end{aligned}$$

“双重改进”效果显著
- in Amortized Sense

When we use both union by rank and path compression, the worst-case running time is $O(m\,\alpha(n))$, where $\alpha(n)$ is a *very* slowly growing function, which we define in Section 21.4. In any conceivable application of a disjoint-set data structure, $\alpha(n) \leq 4$; thus, we can view the running time as linear in m in all practical situations. Strictly speaking, however, it is superlinear.

问题18:

什么意思?

一个增长“极慢”的函数

For integers $k \geq 0$ and $j \geq 1$, we define the function $A_k(j)$ as

$$A_k(j) = \begin{cases} j+1 & \text{if } k = 0\,, \\ A_{k-1}^{(j+1)}(j) & \text{if } k \geq 1\,, \end{cases}$$

“逆”

$$\begin{aligned} A_4(1) &= A_3^{(2)}(1) \\ &= A_3(A_3(1)) \\ &= A_3(2047) \\ &= A_2^{(2048)}(2047) \\ &\gg A_2(2047) \\ &= 2^{2048} \cdot 2048 - 1 \\ &> 2^{2048} \\ &= (2^4)^{512} \\ &= 16^{512} \\ &\gg 10^{80}\,, \end{aligned}$$

$$\alpha(n) = \min\{k : A_k(1) \geq n\}$$

$$\alpha(n) = \begin{cases} 0 & \text{for } 0 \leq n \leq 2\,, \\ 1 & \text{for } n = 3\,, \\ 2 & \text{for } 4 \leq n \leq 7\,, \\ 3 & \text{for } 8 \leq n \leq 2047\,, \\ 4 & \text{for } 2048 \leq n \leq A_4(1)\,. \end{cases}$$

在“任何实际意义”上不大于4

结点的*rank*有什么性质?

Lemma 21.4

For all nodes x, we have $x.rank \leq x.p.rank$, with strict inequality if $x \neq x.p$. The value of $x.rank$ is initially 0 and increases through time until $x \neq x.p$; from then on, $x.rank$ does not change. The value of $x.p.rank$ monotonically increases over time.

理解rank的几个问题:

- 任一结点与其父结点rank的值是什么关系?
- rank的值什么情况下会改变? 如何改变? 什么情况下就不再会改变了?
- 从任一结点到其所在的树的根经过的点的rank值分布有什么规律吗?
- rank值有上限吗?

把*rank*和函数A, α联系起来

$$\text{level}(x) = \max\{k : x.p.rank \geq A_k(x.rank)\} .$$

That is, level(x) is the greatest level k for which A_k, applied to x's rank, is no greater than x's parent's rank.

$$0 \leq \text{level}(x) < \alpha(n)$$

$$\text{iter}(x) = \max\{i : x.p.rank \geq A^{(i)}_{\text{level}(x)}(x.rank)\} .$$

That is, iter(x) is the largest number of times we can iteratively apply $A_{\text{level}(x)}$, applied initially to x's rank, before we get a value greater than x's parent's rank.

$$1 \leq \text{iter}(x) \leq x.rank \qquad (x.rank \geq 1)$$

Amortized Analysis

$$\phi_q(x) = \begin{cases} \alpha(n) \cdot x.rank & \text{if } x \text{ is a root or } x.rank = 0\,, \\ (\alpha(n) - \text{level}(x)) \cdot x.rank - \text{iter}(x) & \text{if } x \text{ is not a root and } x.rank \geq 1\,. \end{cases}$$

Corollary 21.9

If node x is not a root and $x.rank > 0$, then $\phi_q(x) < \alpha(n) \cdot x.rank$.

Lemma 21.10

Let x be a node that is not a root, and suppose that the qth operation is either a LINK or FIND-SET. Then after the qth operation, $\phi_q(x) \leq \phi_{q-1}(x)$. Moreover, if $x.rank \geq 1$ and either level(x) or iter(x) changes due to the qth operation, then $\phi_q(x) \leq \phi_{q-1}(x) - 1$. That is, x's potential cannot increase, and if it has positive rank and either level(x) or iter(x) changes, then x's potential drops by at least 1.

如果第q个操作是link或find，“势能”不会增加，甚至可能下降至少1

Amortized Cost of Link – as an example

假设第q个操作是Link(x,y), 并且新的root是y:

注意：link操作只可能使x,y, 或link前y的后裔结点的势能发生变化。可以证明其中只有y的势能可能增加，且最多增加α(n)

- By Lemma 21.10, any node that is y's child just before the LINK cannot have its potential increase due to the LINK.
- From the definition of $\phi_q(x)$, we see that, since x was a root just before the qth operation, $\phi_{q-1}(x) = \alpha(n) \cdot x.rank$. If $x.rank = 0$, then $\phi_q(x) = \phi_{q-1}(x) = 0$. Otherwise,

$$\begin{aligned} \phi_q(x) &< \alpha(n) \cdot x.rank \quad \text{(by Corollary 21.9)} \\ &= \phi_{q-1}(x) , \end{aligned}$$

  and so x's potential decreases.
- Because y is a root prior to the LINK, $\phi_{q-1}(y) = \alpha(n) \cdot y.rank$. The LINK operation leaves y as a root, and it either leaves y's rank alone or it increases y's rank by 1. Therefore, either $\phi_q(y) = \phi_{q-1}(y)$ or $\phi_q(y) = \phi_{q-1}(y) + \alpha(n)$.

The increase in potential due to the LINK operation, therefore, is at most $\alpha(n)$. The amortized cost of the LINK operation is $O(1) + \alpha(n) = O(\alpha(n))$. ■

课外作业

- TC pp.564-: ex.21.1-2, 21.1-3
- TC pp.567-: ex.21.2-1, 21.2-3, 21.2-6
- TC pp.572-: ex.21.3-1, 21.3-2, 21.3-3
- TC pp.582-: prob.21-1