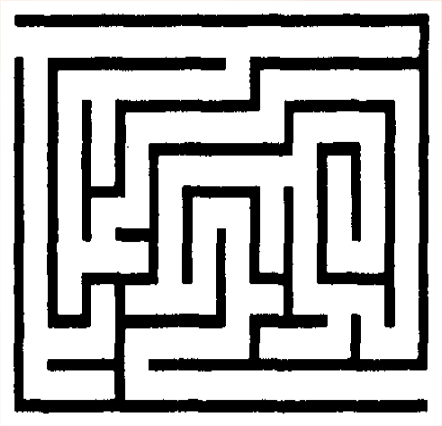
计算机问题求解－论题2-15
-用于动态等价关系的数据结构

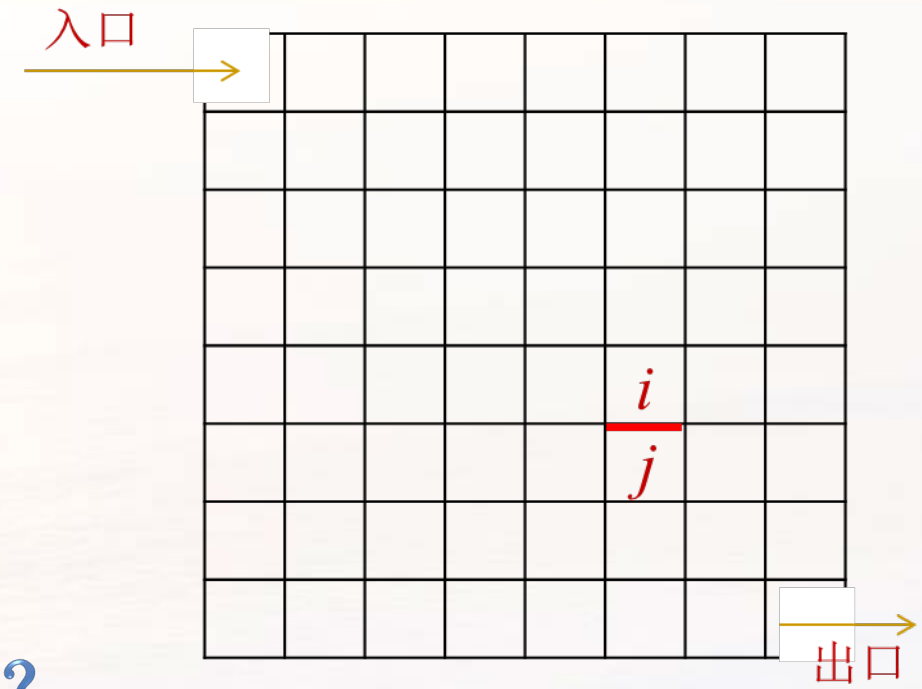
TC-17、21

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？



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

集合$S=\{S_1,S_2,\ldots,S_k \mid S_i\cap S_j=\phi,\ i,j\in\{1,2,\ldots,k\},\ 且 i\neq j\}$

定义操作集如下：

(创建) Make-Set(x): x为不属于其它任意已创建集合的对象，操作结果是$\{x\}$

(结构) Union(x,y): x,y是任意两个对象，假设他们分别属于集合S_x, S_y，操作结果：用集合$S_x\cup S_y$替换S 中原来的S_x和S_y。

(查询) Find-Set(x): x是任意对象，操作结果是指向x所属集合的指针。

问题2：

如果要实现这一结构，还需要考虑什么？

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?

表示方式称为“well-defined”的要求是什么?

你还记得以前我们讨论等价类“乘法”时碰到过的问题吗?

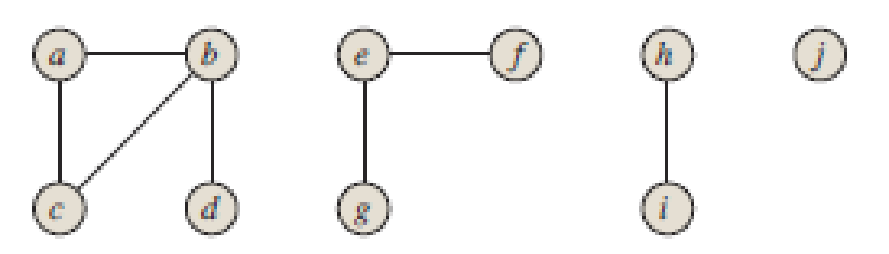
We care only that if we ask for the representative of a dynamic set twice without modifying the set between the requests, we get the same answer both times.

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？

考虑n次MakeSet, m次各种操作（三种）的序列的代价。

将无向图分解为连通分支的集合



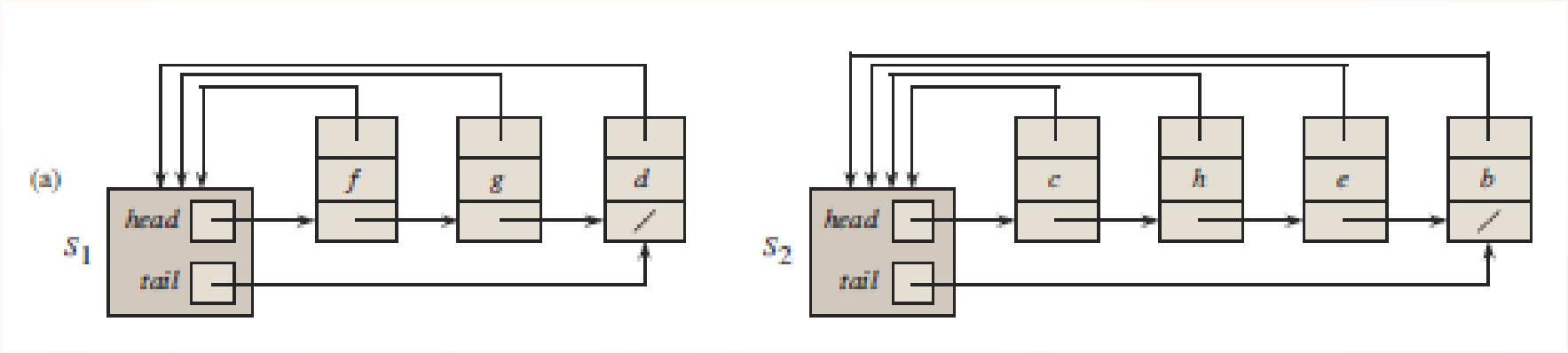

```
CONNECTED-COMPONENTS(G)
1  for each vertex v ∈ G.V
2      MAKE-SET(v)
3  for each edge (u, v) ∈ G.E
4      if FIND-SET(u) ≠ FIND-SET(v)
5          UNION(u, v)

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

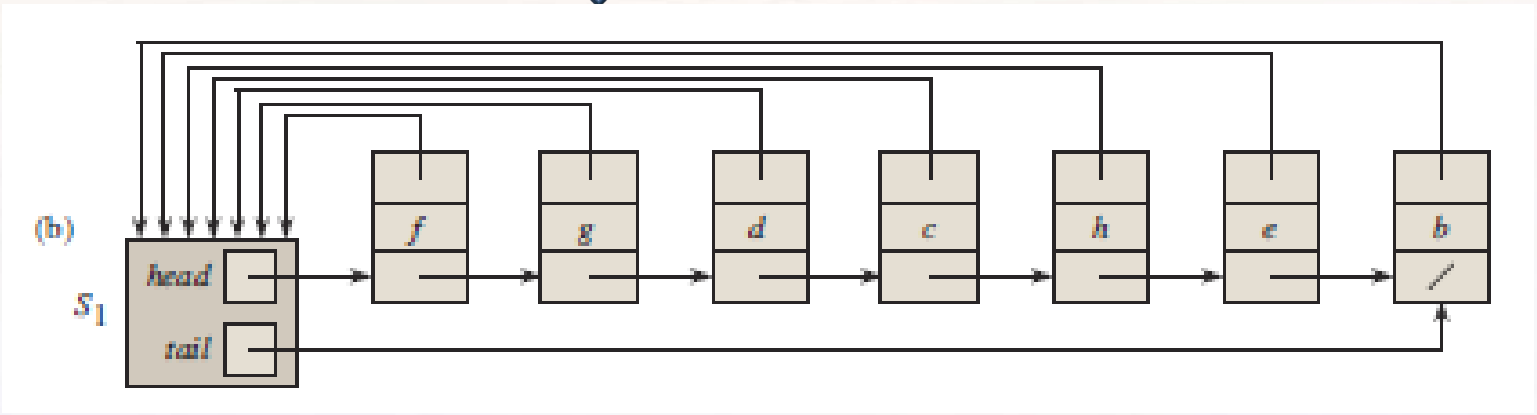
Edge processed	Collection of disjoint sets									
initial sets	{a}	{b}	{c}	{d}	{e}	{f}	{g}	{h}	{i}	{j}
(b,d)	{a}	{b,d}	{c}		{e}	{f}	{g}	{h}	{i}	{j}
(e,g)	{a}	{b,d}	{c}		{e,g}	{f}		{h}	{i}	{j}
(a,c)	{a,c}	{b,d}			{e,g}	{f}		{h}	{i}	{j}
(h,i)	{a,c}	{b,d}			{e,g}	{f}		{h,i}		{j}
(a,b)	{a,b,c,d}				{e,g}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

注意：
上面的代码段并不输出左表最下面一行，只是提供查询“服务”。

Implementing by Linked-List



操作union(*g*,*e*)执行后



问题5：

为什么用链表实现，每个操作的平均代价可能会是线性的？

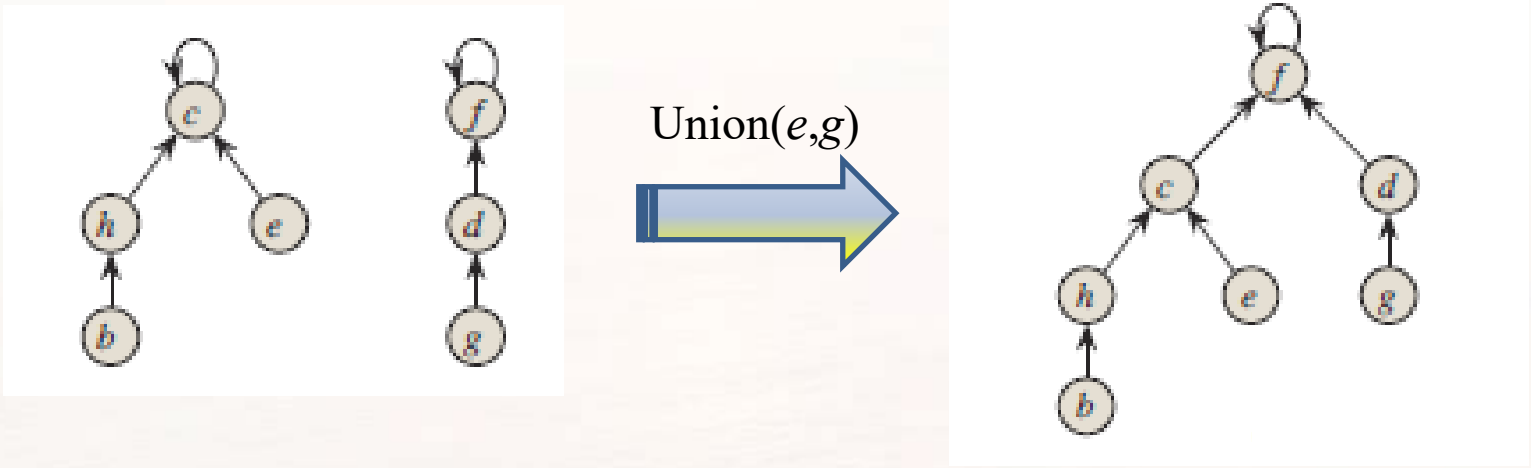
Union操作对象的次序不影响结果，却影响效率，为什么？这对你有什么启发？

当我们打算合并两个链表时，应该总是选择小的并并入大的，而不是相反或者“随意”！

这就是“weighted-union”，虽然单看一次操作，代价仍然可能是线性的，但涉及n个初始对象长度为m的（含3种操作）序列的代价为：

$$O(m+n\lg n)$$

更适合的实现结构：
inTree 与 disjoint-set Forest



问题6:

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7：

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？这对改进算法有什么启示？

什么地方需要改进？

Find(x)的代价与x的深度有关。

控制树高度： Union by Rank

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

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
```

问题8：

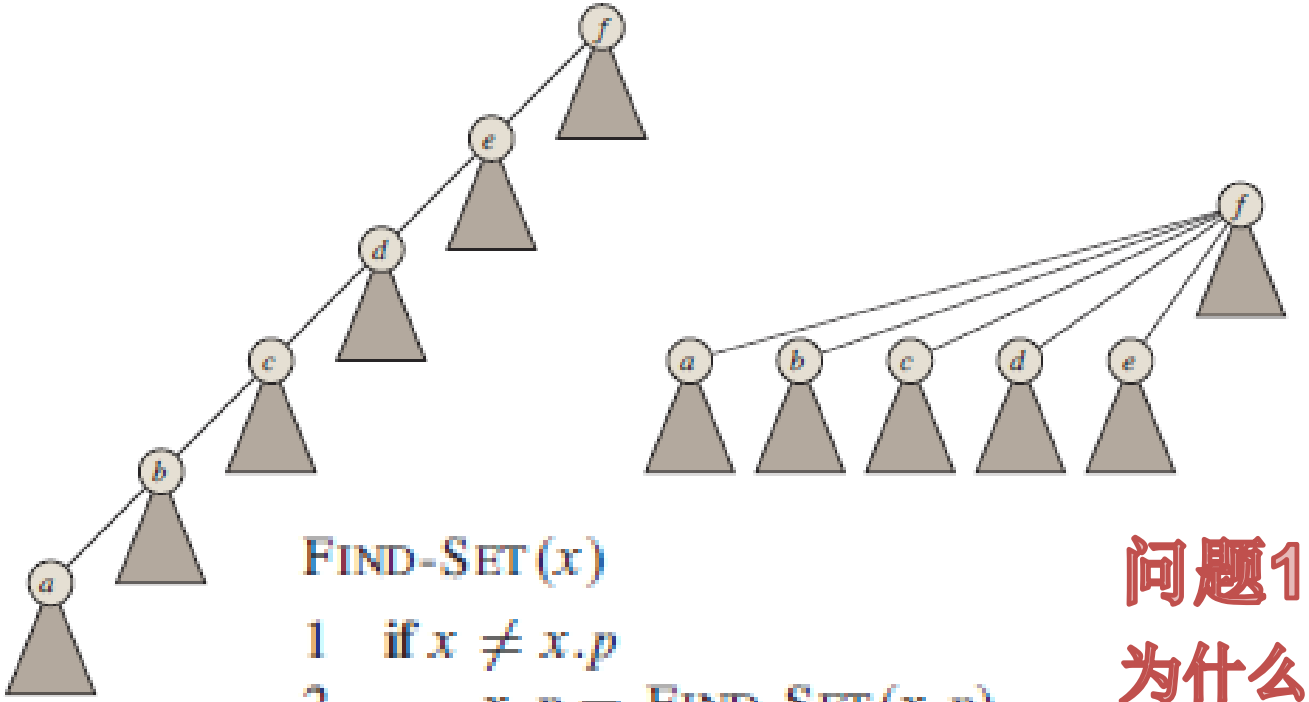
你能解释rank的值及其更新的意义吗？

问题9：

哪里体现树高度控制，与weighted-union有什么异同？

为什么前面不需要修改rank?

降低结点深度： Path Compression




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

书上说“a *two-pass method*”，什么意思?

问题10：

为什么一个递归语句就能使左图变成右边的样子？

Part II
Amortized Analysis

k 位二进计数器

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
```

当前$A[i]$=0，否则“溢出”

问题11：

按照flip次数计，加1最多要做k次操作，那么从0加到n，似乎worst-case代价为O(kn)，这合理吗？

大代价操作执行次数的上限

考虑计数器的值从0上升到n，即执行n次increment操作。

问题12：

上面的例子中，计数器增值到32，increment操作最多做5次flip，这样的操作仅有1次，为什么？需要3次flip的increment操作需要几次？

代价大的情况发生频度是受限制的，而且这个限制与较小代价操作的数量有关。

Amortized Analysis: Aggregate 方法

计算连续n次increment操作worst-case的总次数。

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

所以，总操作次数为：

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

换句话说：最坏情况下，操作平均代价为$O(1)$
而执行n次increment操作的代价是$O(n)$

问题17：

这为什么不是“average case”?

Amortized Analysis: Accounting 方法

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时刻，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**(0变为1)操作，而while循环内的flip(line 3)为**reset**(1变为0)操作。

则accounting cost指定如下：**set**：2（“用1存1”）**reset**：0（“积分支付”）

任何时刻，计数器中“1”的位数即当前“积分”数，因此不会为负（不会“透支”）；总代价也不会大于当前计数器的值的2倍。

Amortized Analysis: Potential 方法

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：

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

实际代价与“势能”变化量之和

在计数器问题中，定义$\Phi(D_i)$=第i次操作后计数器中1的个数b_i。$\Phi(D_0)$=0。

注意：如果将第i次操作中有t_i位被reset（置0），则 $\boldsymbol{b_i \le b_{i-1} - t_i + 1}$

注意：

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

所以，只要 $\Phi(D_n) \ge \Phi(D_0)$

Amortized cost就可以作为实际代价的上限。

由上式可知：

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

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

“双重改进”的Union-Find 的效率

问题13：
为什么对于Union-Find操作序列的代价分析应采用amortized方法？

考虑到每个操作的实际代价，从“蓄能”与“耗能”的角度考虑，每个操作具有什么特性？

为了便于理清楚，书上是将union分拆为find和link分别考虑的。

Analysis: the Basic Idea

- *c*Find may be an expensive operation, in the case that find(*i*) is executed and the node *i* has great depth.
- However, such *c*Find can be executed only for limited times, relative to other operations of lower cost.
- So, amortized analysis applys.

Amortized Time Analysis

- Amortized equation:

  amortized cost = actual cost + accounting cost

- Design goals for accounting cost
  - In any legal sequence of operations, the sum of the accounting costs is nonnegative.
  - The amortized cost of each operation is fairly regular, in spite of the wide fluctuate possible for the actual cost of individual operations.

Co-Strength of *wUnion* and *cFind*

- The number of link operations done by a *Union-Find* program implemented with *wUnion* and *cFind*, of length m on a set of n elements is in $O((n+m)\lg^*(n))$ in the worst case.

- What's $\lg^*(n)$?
  - Define the function H as following:

$$\begin{cases} H(0)=1 \\ H(i)=2^{H(i-1)} \ \ for\ \ i>0 \end{cases}$$

  - Then, $\lg^*(j)$ for $j \geq 1$ is defined as:

$$\lg^*(j)=\min\{\, k \mid H(k) \geq j \,\}$$

Definitions with a *Union-Find* Program *P*

- Forest *F*: the forest constructed by the sequence of *union* instructions in *P*, assuming:
  - *wUnion* is used;
  - the *find*s in the *P* are ignored
- Height of a node *v* in any tree: the height of the subtree rooted at *v*
- Rank of *v*: the height of *v* ***in F***

Note: *cFind* changes the height of a node, but the rank for any node is invariable.

Constraints on Ranks in *F*

- The upper bound of the number of nodes with rank r ($r \geq 0$) is $\frac{n}{2^r}$
  - Remember that the height of the tree built by *wUnion* is at most $\lfloor \lg n \rfloor$, which means the subtree of height r has at least 2^r nodes.
  - The subtrees with root at rank r are disjoint.
- There are at most $\lfloor \lg n \rfloor$ different ranks.
  - There are altogether n elements in S, that is, n nodes in F.

Increasing Sequence of Ranks

- The ranks of the nodes on a path from a leaf to a root of a tree in F form a strictly increasing sequence.
- When a *cFind* operation changes the parent of a node, the new parent has higher rank than the old parent of that node.
  - Note: the new parent was an ancestor of the previous parent.

A Function Growing Extremely Slowly

- Function H:

$$\begin{cases} H(0)=1 \\ H(i+1)=2^{H(i)} \end{cases}$$

that is: $H(k)=2^{2^{\cdot^{\cdot^{\cdot^{2}}}}}$ (k 2's)

Note:

H grows extremely fast:

$H(4)=2^{16}=65536$

$H(5)=2^{65536}$

- Function Log-star

$\lg^*(j)$ is defined as the least i such that:

$H(i)\geq j$ for $j>0$

- Log-star grows extremely slowly

$$\lim_{n\to\infty}\frac{\lg^*(n)}{\log^{(p)} n}=0$$

p is any fixed nonnegative constant

For any x: $2^{16}+1\leq x\leq 2^{65536}$, $\lg^*(x)=5$!

Grouping Nodes by Ranks

- Node $v \in s_i$ ($i \geq 0$) iff. lg*(1+rank of v)=i
  - which means that: if node v is in group i, then

    $r_v \leq H(i)-1$, but not in group with smaller labels
- So,
  - Group 0: all nodes with rank 0
  - Group 1: all nodes with rank 1
  - Group 2: all nodes with rank 2 or 3
  - Group 3: all nodes with its rank in [4,15]
  - Group 4: all nodes with its rank in [16, 65535]
  - Group 5: all nodes with its rank in [65536, ???]

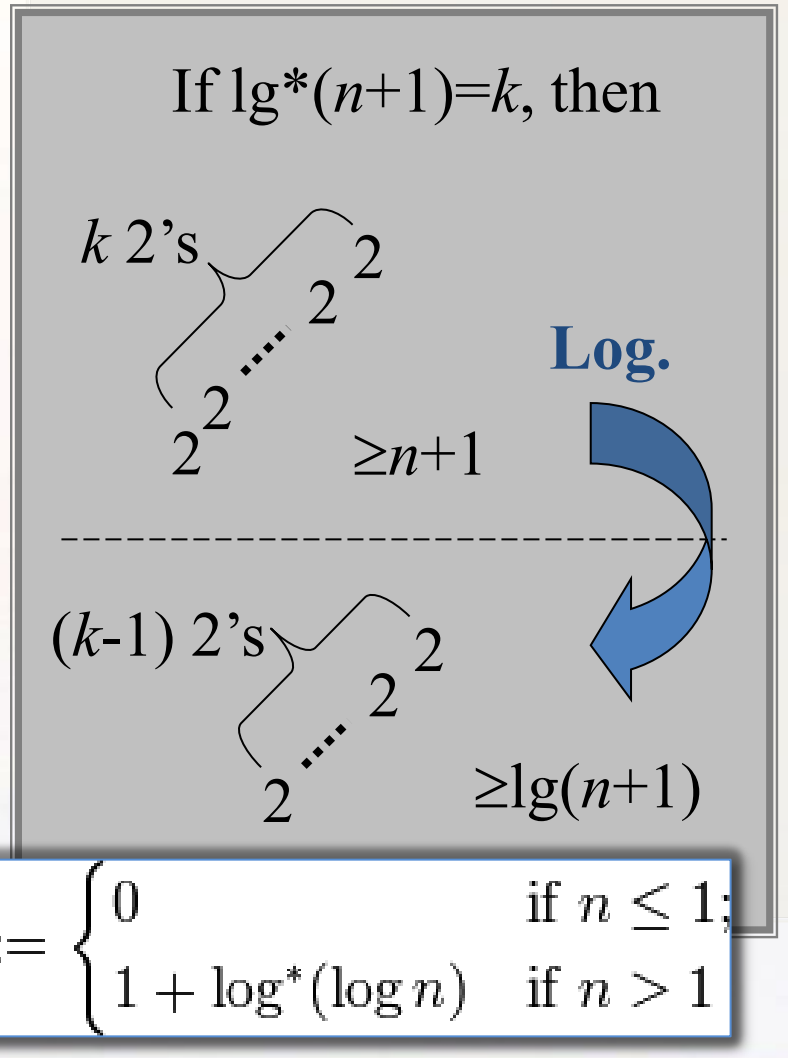
Group 5 exists only when n is at least 2^{65536}. What is that?

Very Few Groups

- **Node $v \in S_i$ ($i \geq 0$) iff.**

  $\lg^*(1+\text{rank of } v)=i$

- Upper bound of the number of distinct node groups is $\lg^*(n+1)$
  - The rank of any node in F is at most $\lfloor \lg n \rfloor$, so the largest group index is $\lg^*(1+\lfloor \lg n \rfloor)=\lg^*(\lceil \lg n+1 \rceil) = \lg^*(n+1)-1$



$$\log^* n := \begin{cases} 0 & \text{if } n \leq 1; \\ 1 + \log^*(\log n) & \text{if } n > 1 \end{cases}$$

Amortized Cost of *Union-Find*

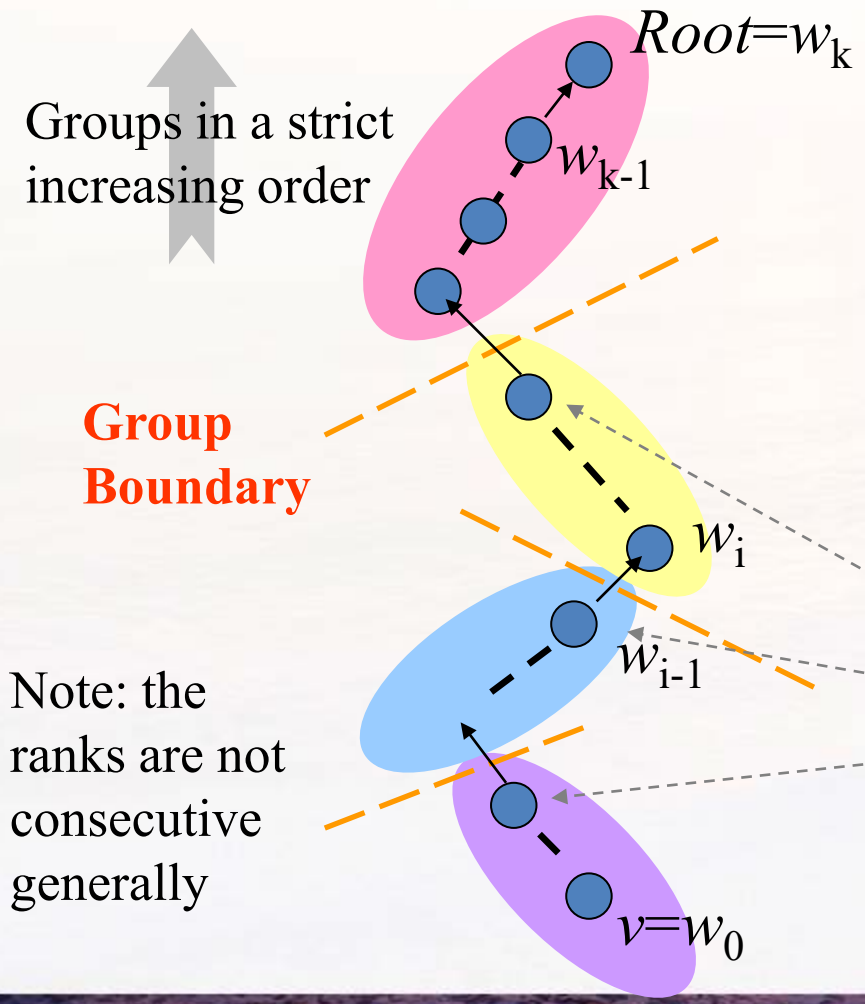
- Amortized Equation Recalled

  amortized cost

  = actual cost + accounting cost

- The operations to be considered:
  - *n* makeSets
  - *m* union & find (with at most *n*-1 unions)

One Execution of *cFind*(w_0)



Note: the ranks are not consecutive generally

Only when k=0,1, there is no parent change

For one *cFind* operation, the actual cost is $2k$ Not $2(k+1)$

Accounting cost is -2 for each pair of (w_{i-1}, w_i) for the 2 nodes in ths **same group only**, which we call a **withdrawal**.

Amortizing Scheme for *wUnion-cFind*

- makeSet
  - Accounting cost is $4\lg^*(n+1)$
  - So, the amortized cost is $1+4\lg^*(n+1)$
- *wUnion*
  - Accounting cost is 0
  - So the amortized cost is 1
- *cFind*
  - Accounting cost is describes as in the previous page.
  - Amortized cost $\leq 2k-2((k-1)-(\lg^*(n+1)-1))=2\lg^*(n+1)$
    (Compare with the worst case cost of *cFind,* $2\lg n$)

Number of withdrawal

Validation of the Amortizing Scheme

- We must be assure that **the sum of the accounting costs is never negative**.
- The sum of the negative charges, incurred by *cFind*, does not exceed $4n\lg^*(n+1)$
  - We prove this by showing that at most $2n\lg^*(n+1)$ withdrawals on nodes occur during all the executions of *c*Find.

Key Idea in the Derivation

- For any node, the number of withdrawal will be less than the number of different ranks in the group it belong to
  - When a *cFind* changes the parent of a node, the new parent is always has higher rank than the old parent.
  - Once a node is assigned a new parent in a higher group, no more negative amortized cost will incurred for it again.
- The number of different ranks is limited within a group.

Derivation

a loose upper bound of ranks in a group

- The number of withdrawals for all $w \in S$ is:

$$\sum_{i=0}^{\lg^*(n+1)-1} H(i)\,(\text{number of nodes in group i})$$

Note: number of nodes in group i is at most:

$$\sum_{r=H(i-1)}^{H(i)-1} \frac{n}{2^r} \le \frac{n}{2^{H(i-1)}} \sum_{j=0}^{\infty} \frac{1}{2^j} = \frac{2n}{2^{H(I-1)}} = \frac{2n}{H(i)}$$

So,

$$\sum_{i=0}^{\lg^*(n+1)-1} H(i) \frac{2n}{H(i)} = 2n \lg^*(n+1)$$

The Conclusion

- The number of link operations done by a *Union-Find* program implemented with *wUnion* and *cFind*, of length m on a set of n elements is in $O((n+m)lg^*(n))$ in the worst case.
  - Note: since the sum of accounting cost is never negative, the actual cost is always not less than amortized cost. And, the upper bound of amortized cost is: $(n+m)(1+4\lg^*(n+1))$

结论

When we use both union by rank and path compression, the worst-case running time is $O(m \cdot \alpha(n))$, where $\alpha(n)$ is a *very* slowly growing function. 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.