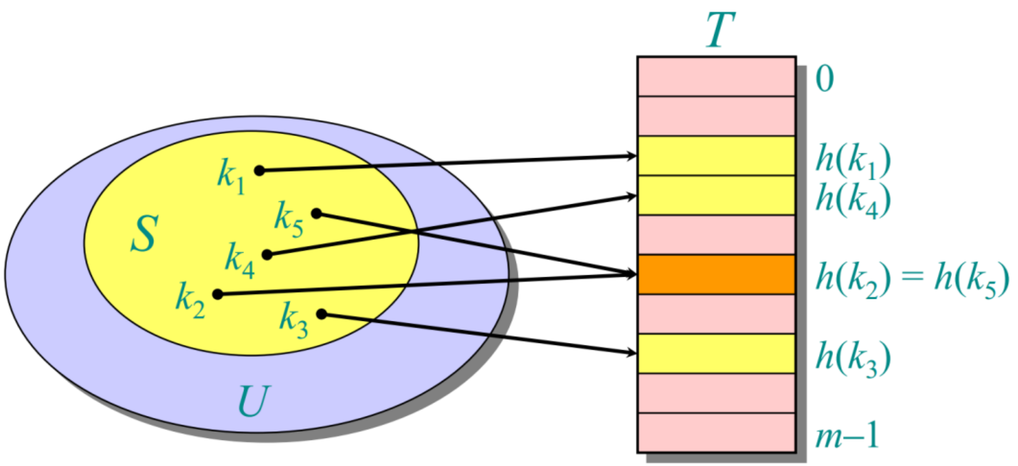
Disjoint-Set Data Structure Union-Find(Amortized Analysis)

Instructor: Shizhe Zhou

Course Code:00125401

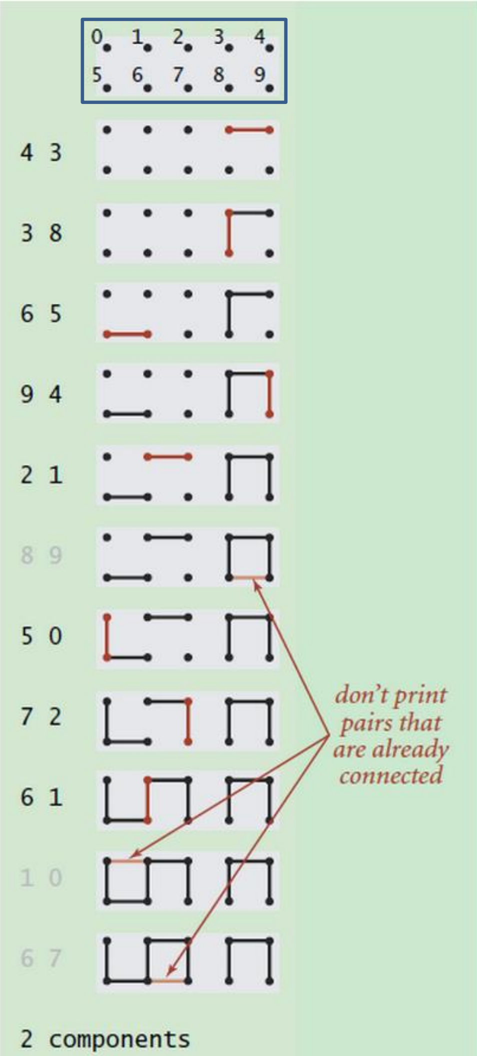
Dynamic Set

- We knew Hash table already



- Disjoint Set
  - Another type of Dynamic Set
  - Pairwise connectivity

Union Find(并查集)



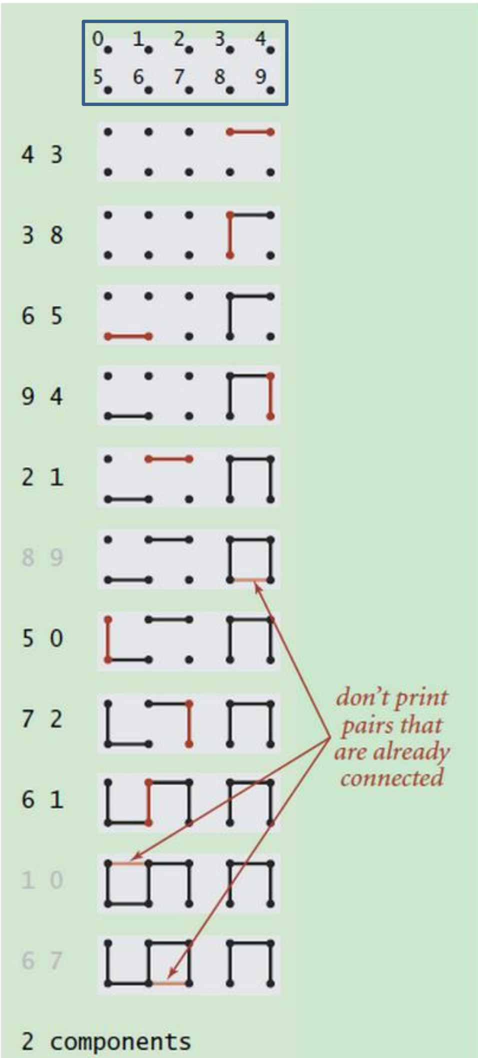
- **动态连通性问题:**

假设我们输入了一组整数对，即上图中的(4, 3), (3, 8)等等，每对整数代表这两个points/sites是连通的。那么随着数据的不断输入，整个图的连通性也会发生变化.

--对每个联通组,要能快速知道一个独一无二的代表性元素.

--对每个元素，要能快速查询所在连通分支组号.

Union Find(并查集)



- **动态连通性问题:**

对动态连通性这个场景而言，我们需要解决的问题可能是:

- 给出两个节点，判断它们是否连通，如果连通，不需要给出具体的路径 **(union find problem)**
- 给出两个节点，判断它们是否连通，如果连通，需要给出具体的路径 (graph traversal problem)

Disjoint-Set Data Struture(Union-Find)

Problem: Maintain a dynamic collection of *pairwise-disjoint* sets $\mathsf{S} = \{S_1, S_2, \ldots, S_r\}$.
Each set S_i has one element distinguished as the representative element, $rep[S_i]$.

Must support 3 operations:

仅对还未为编组的元素编组！

- MAKE-SET(x): adds new set $\{x\}$ to S with $rep[\{x\}] = x$ (for any $x \notin S_i$ for all i).
- UNION(x, y): replaces sets S_x, S_y with $S_x \cup S_y$ in S for any x, y in distinct sets S_x, S_y .
- FIND-SET(x): returns representative $rep[S_x]$ of set S_x containing element x.

Naïve solution 1 (数组解法)

- 为每个元素初始化一个组号：

```java
int id[count]; // access to component id (site indexed)
int count; // number of components
public UF(int N)
{ // Initialize component id array.
  for(int i = 0; i < size; i++)
      id[i] = i;
}
```

- 在union(p,q)时，首先判断p和q的组号是否相同。如果相同，不需做任何操作。否则，执行组合并.其实质是将p和q所在组的所有成员组号修改为同一个组号。

```java
public void union(int p, int q)
{
    // 获得p和q的组号
    int pID = find(p);
    int qID = find(q);
    // 如果两个组号相等，直接返回
    if (pID == qID) return;
    // 遍历一次，改变组号使他们属于一个组
    for (int i = 0; i < id.length; i++)
        if (id[i] == pID) id[i] = qID;
    count--;
}
```

- 这样的find可以总是维持$\Theta(1)$.

```java
public int find(int p)
{ return id[p]; }
```

Naïve solution 1 (数组解法)

- 例如, 输入的Pair是(5, 9)，那么首先通过find方法发现它们的组号并不相同，然后在union的时候通过一次遍历，将组号1都改成8。(or, 由8改成1.)

find examines id[5] *and* id[9]

p	q	0	1	2	3	4	5	6	7	8	9
5	9	1	1	1	8	8	1	1	1	8	8

union has to change all 1s to 8s

p	q	0	1	2	3	4	5	6	7	8	9
5	9	1	1	1	8	8	1	1	1	8	8
		8	8	8	8	8	8	8	8	8	8

Quick-find overview

- 随着输入规模的扩大，由于每次合并集合需要修改组号，因此就必须对整个数组进行遍历，找到需要修改的节点，逐一修改，这一下每次Union的复杂度就是线性的。在我们的动态连通性问题中:如果要添加的新路径的数量是M，节点数量是N，那么最后的时间复杂度就是MN，是一个平方复杂度.

```java
public void union(int p, int q)
{
    // 获得p和q的组号
    int pID = find(p);
    int qID = find(q);
    // 如果两个组号相等，直接返回
    if (pID == qID) return;
    // 遍历一次，改变组号使他们属于一个组
    for (int i = 0; i < id.length; i++)
        if (id[i] == pID) id[i] = qID;
    count--;
}
```

Naïve solution 1 (数组解法)

- 数组法效率低下：牵一发而动全身！

find examines id[5] *and* id[9]

p	q	0	1	2	3	4	5	6	7	8	9
5	9	1	1	1	8	8	1	1	1	8	8

union has to change all 1s to 8s

p	q	0	1	2	3	4	5	6	7	8	9
5	9	1	1	1	8	8	1	1	1	8	8
		8	8	8	8	8	8	8	8	8	8

Quick-find overview

因此要提高union方法的效率，让它不再需要遍历整个数组

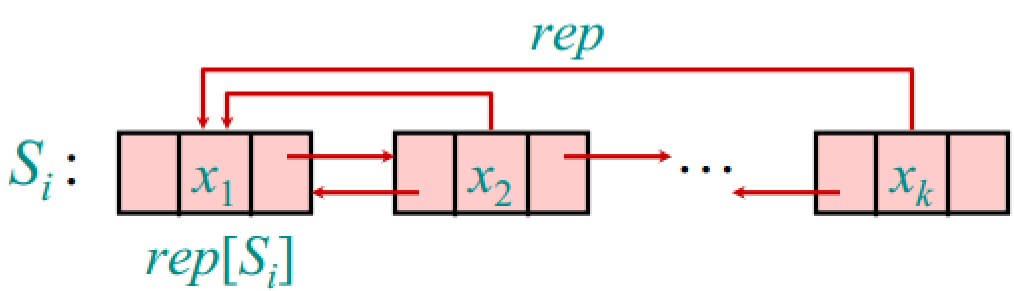
```
public void union(int p, int q)
{
    // 获得p和q的组号
    int pID = find(p);
    int qID = find(q);
    // 如果两个组号相等，直接返回
    if (pID == qID) return;
    // 遍历一次，改变组号使他们属于一个组
    for (int i = 0; i < id.length; i++)
        if (id[i] == pID) id[i] = qID;
    count--;
}
```

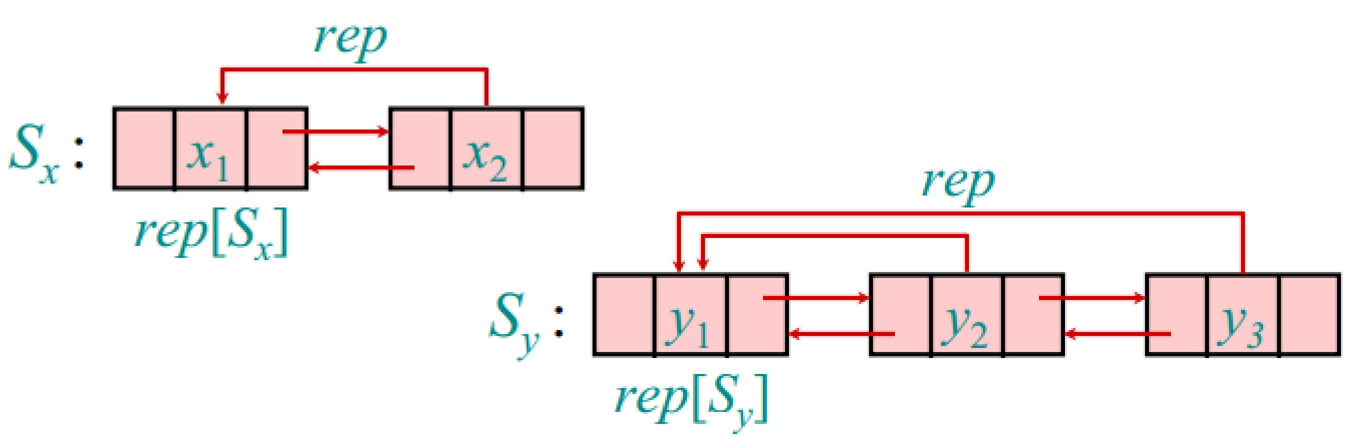
Naïve solution 2 (double-linked list)

- Find(x) == return rep[x]

  – $\Theta(1)$

- Union(x, y) : 链接两个list(包含x和y的),然后更新y的list中所有元素的rep指针. – $\Theta(n)$



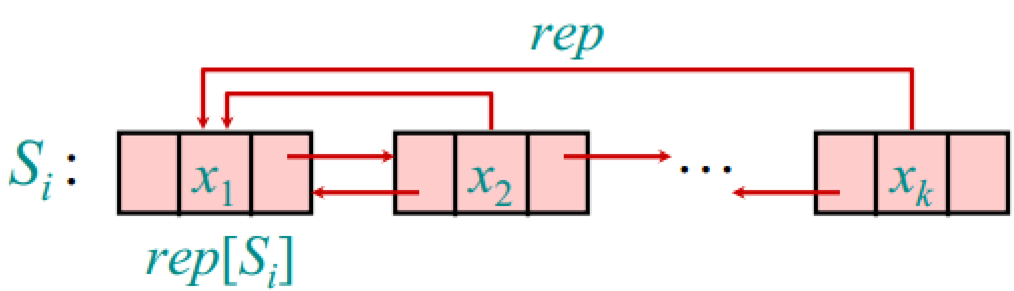


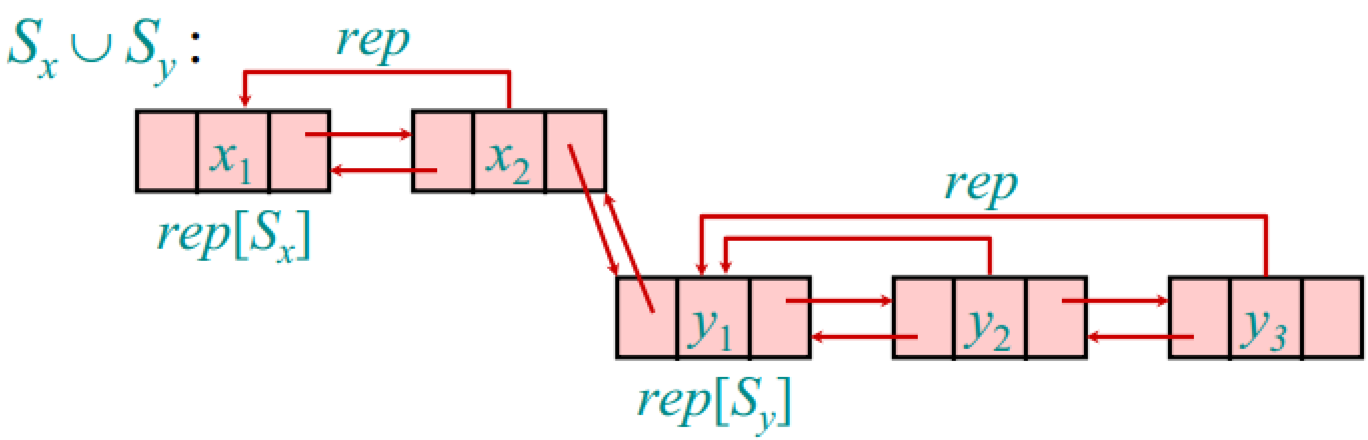
Naïve solution 2 (double-linked list)

- Find(x) == return rep[x]

  – $\Theta(1)$

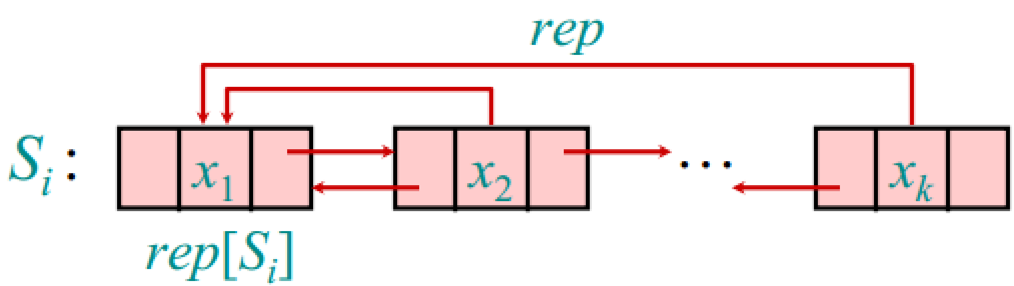
- Union(x, y) : 链接两个list(包含x和y的),然后更新y的list中所有元素的rep指针. – $\Theta(n)$



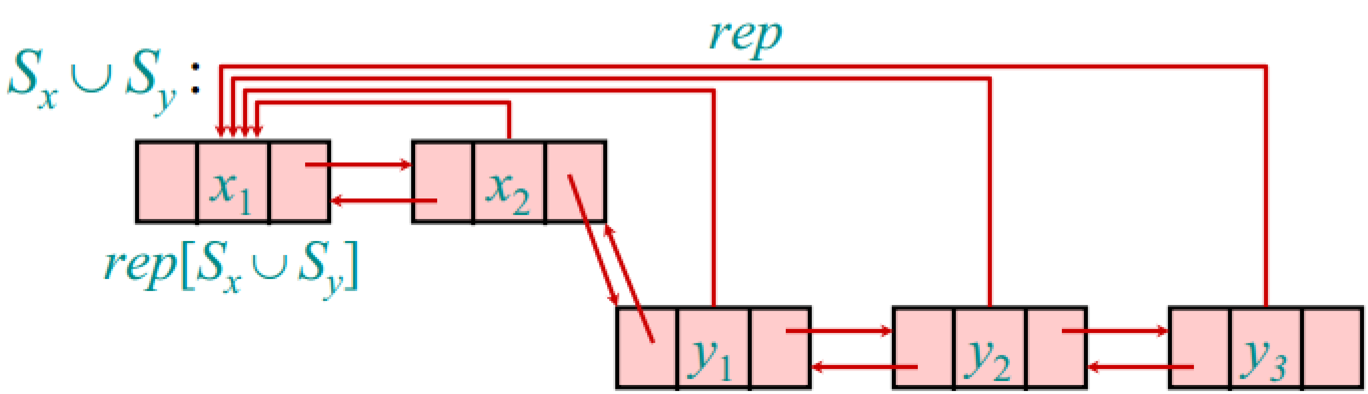


Naïve solution 2 (double-linked list)

- Find(x) == return rep[x]
  – $\Theta(1)$



- Union(x, y) : 链接两个list(包含x和y的),然后更新y的list中所有元素的rep指针.   – $\Theta(n)$



Union-Find算法1-QuickUnion

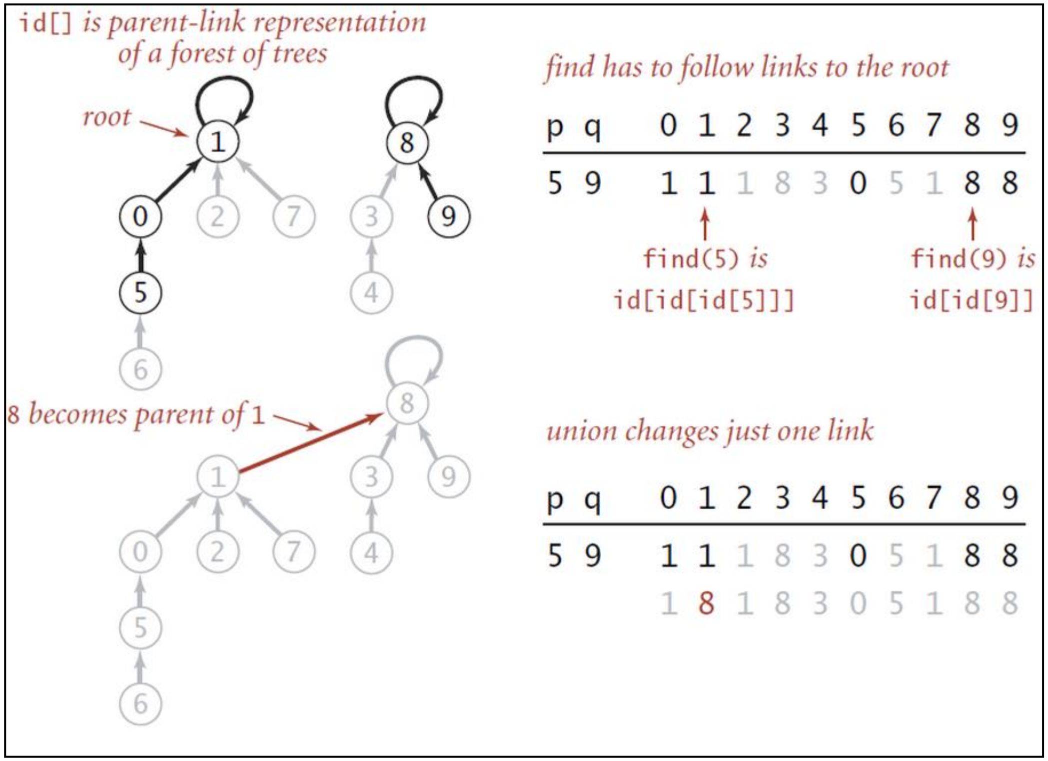
- 采用parent-link的方式将节点组织起来.
- id[p]的值就是p节点的父节点的序号
- 如果p是树根的话，id[p]的值就是p
- 经过若干次查找，一个节点总能找到它的根节点，即满足id[root] = root的节点,i.e.组的根节点

```
private int find(int p)
{
    // 寻找p节点所在组的根节点，根节点具有性质id[root] = root
    while (p != id[p]) p = id[p];
    return p;
}
```

```
public void union(int p, int q)
{
    // Give p and q the same root.
    int pRoot = find(p);
    int qRoot = find(q);
    if (pRoot == qRoot)
        return;
    id[pRoot] = qRoot;    // 将一颗树(即一个组)变成另外一课树(即一个组)的子树
    count--;
}
```

Union-Find算法1-QuickUnion

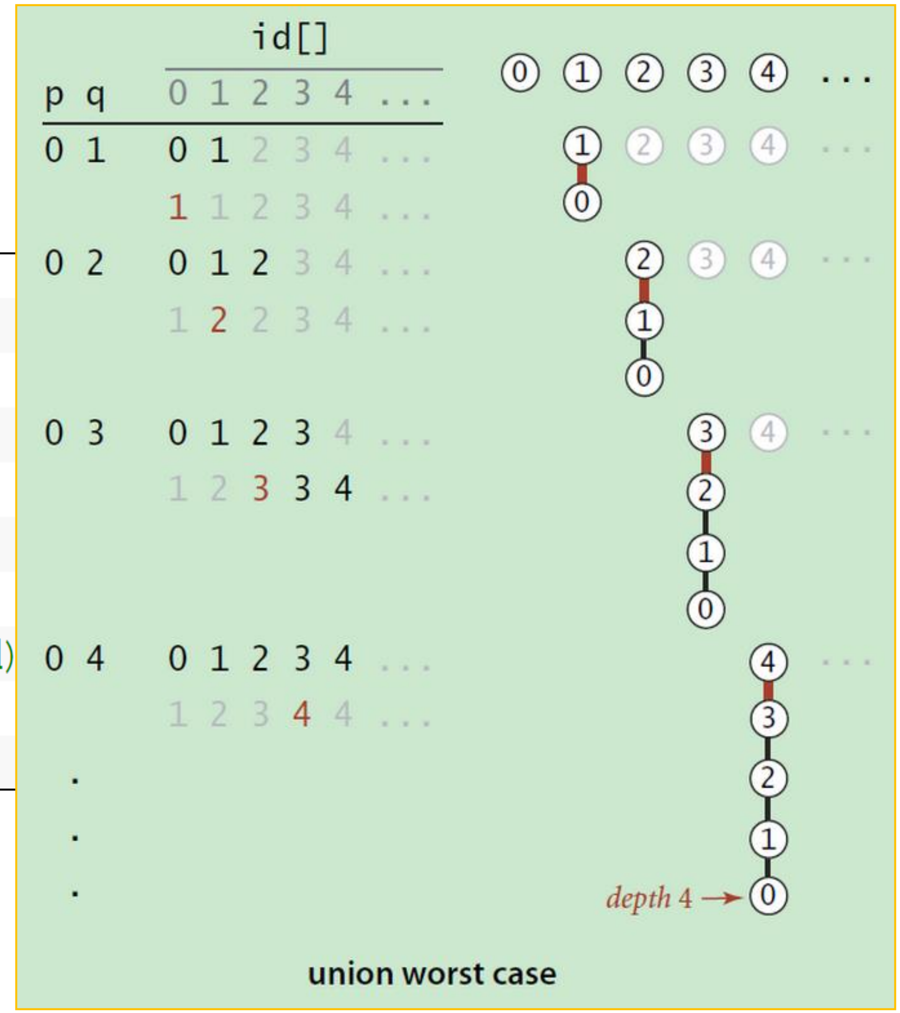
- 采用parent-link的方式将节点组织起来.
- id[p]的值就是p节点的父节点的序号
- 如果p是树根的话，id[p]的值就是p
- 经过若干次查找，一个节点总能找到它的根节点，即满足id[root] = root的节点,i.e.组的根节点



Union-Find算法1-QuickUnion

- 缺陷：易出现极端case

```
public void union(int p, int q)
{
    // Give p and q the same root.
    int pRoot = find(p);
    int qRoot = find(q);
    if (pRoot == qRoot)
        return;
    id[pRoot] = qRoot;    // 将一颗树(即一个组)变成另外一课树(即一个组)
    count--;
}
```

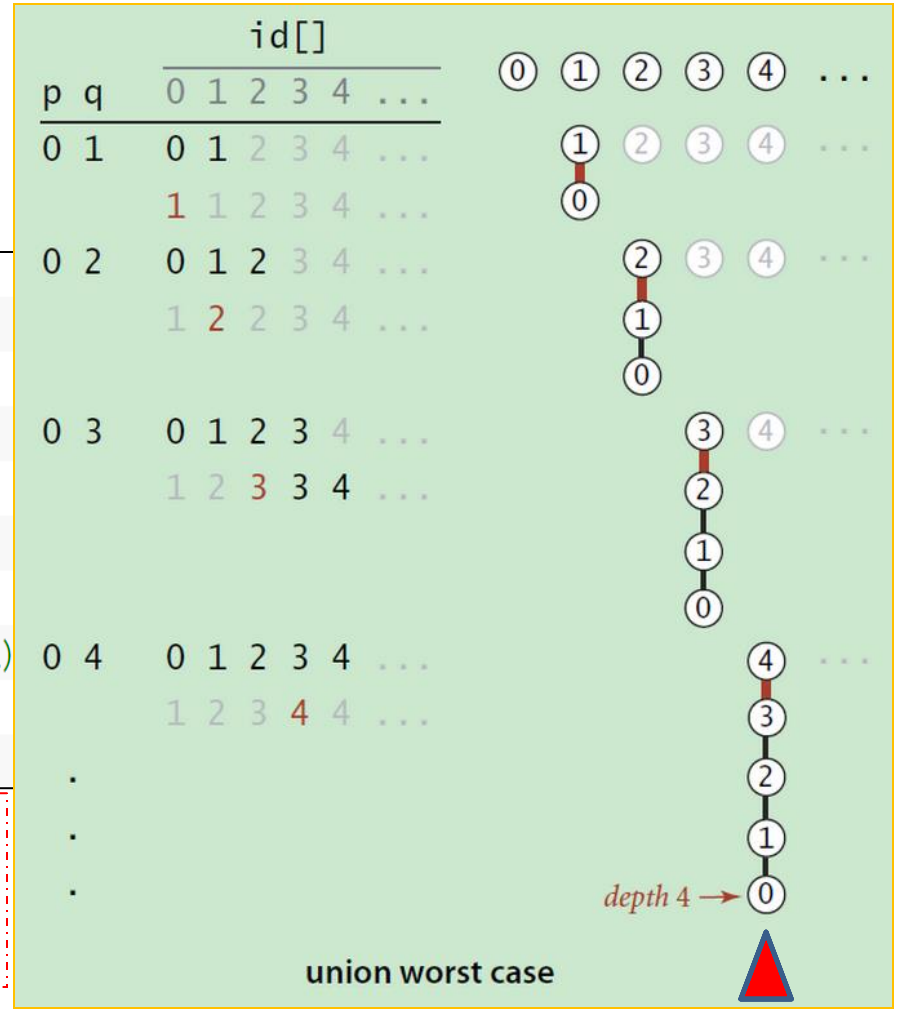


Union-Find算法1-QuickUnion

- 解决方法：使用AVL或者红黑树？Ok,but sometimes the nodes are not comparable……

```
public void union(int p, int q)
{
    // Give p and q the same root.
    int pRoot = find(p);
    int qRoot = find(q);
    if (pRoot == qRoot)
        return;
    id[pRoot] = qRoot;    // 将一颗树(即一个组)变成另外一课树(即一个组)
    count--;
}
```

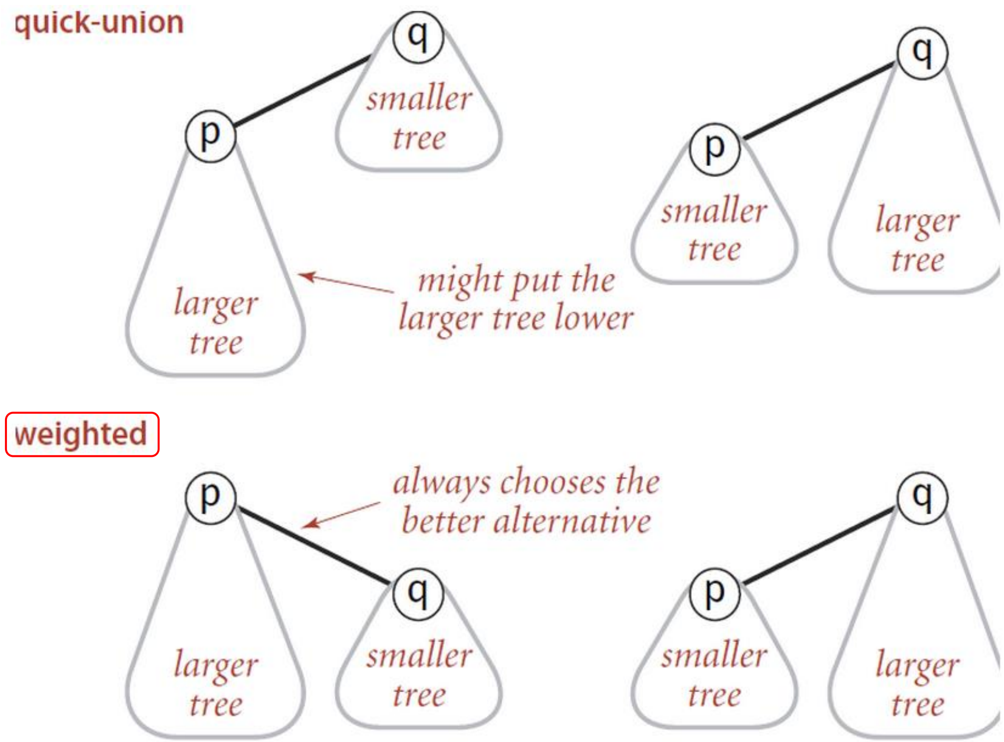
id[pRoot] = qRoot 这行代码不合理(典型的“硬编码”)，导致p所在的树总是会被作为q所在树的子树。I.e p所在set总在不停增加!



畸形树

Union-Find算法2-Weighted Union

- 解法:考虑树的大小，小树并入大树



总是size小的树作为子树和size大的树进行合并。
Intuitively,可以尽量的保持整棵树的平衡.

Union-Find算法2-Weighted Union

- 需要额外标记每个Set的元素个数:

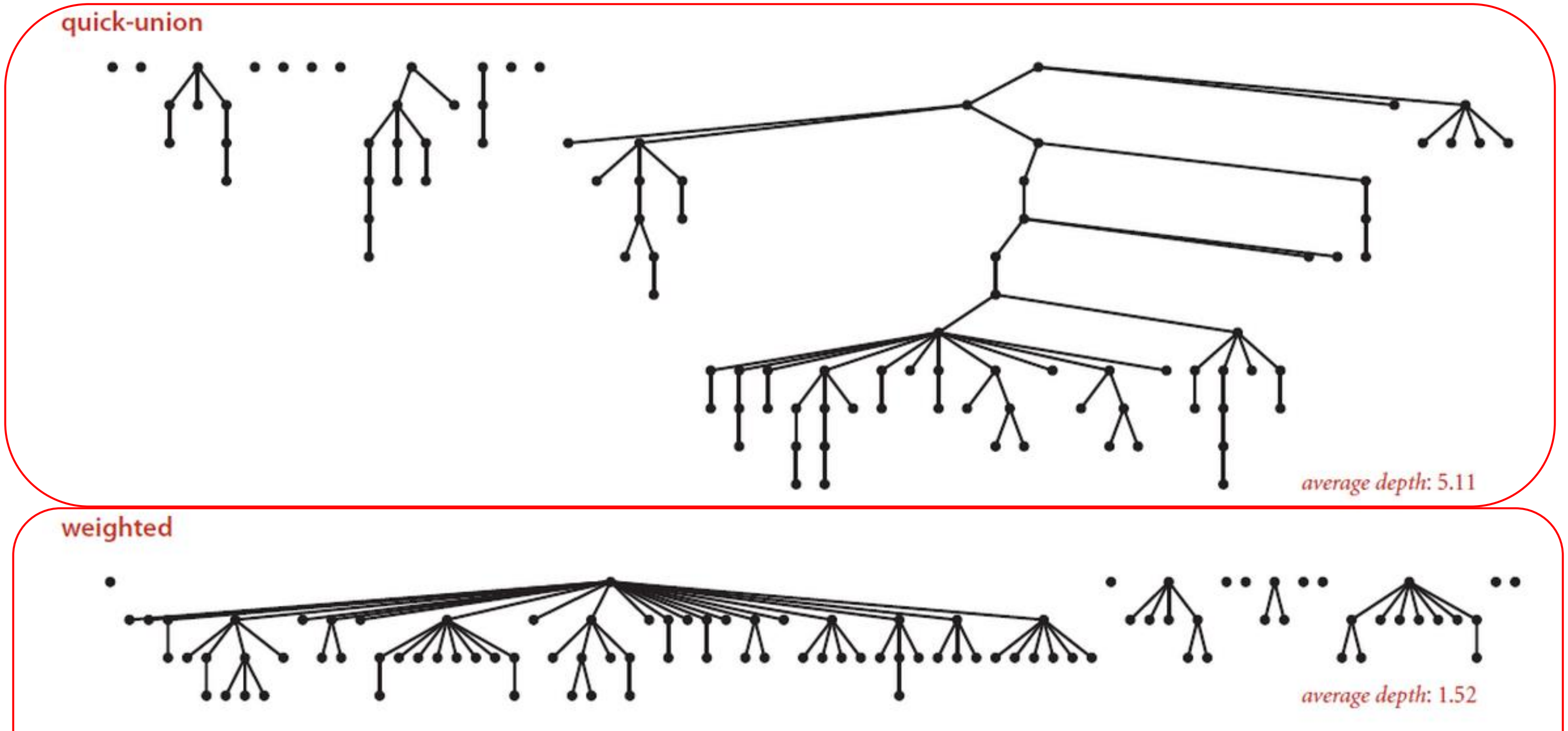
```
for (int i = 0; i < N; i++)
    sz[i] = 1;     // 初始情况下，每个组的大小都是1
```

```
public void union(int p, int q)
{
    int i = find(p);
    int j = find(q);
    if (i == j) return;
    // 将小树作为大树的子树
    if (sz[i] < sz[j]) { id[i] = j; sz[j] += sz[i]; }
    else { id[j] = i; sz[i] += sz[j]; }
    count--;
}
```

总是size小的树作为子树和size大的树进行合并。
Intuitively,可以尽量的保持整棵树的平衡.

Union-Find算法2-Weighted Union

- Union-Find算法1 和 Weighted Quick-Union 的比较



Quick-union and weighted quick-union (100 sites, 88 union() operations)

最后得到的树的高度
大幅度减小了

find方法的效率增加!

Union-Find算法3-Weighted Union with Path Compression!

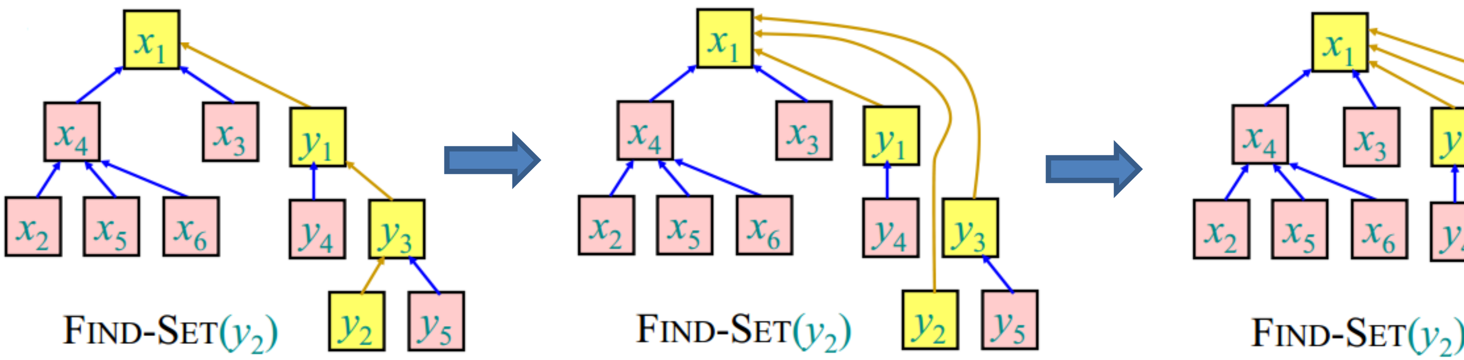
- What can be further improved? **All tree are with height == 1!**

目标：约束仅仅生成十分扁平的树

  - 所有的孩子节点应该都在height为1的地方==
  - 所有的孩子都直接连接到根节点==
  - 保证find操作的最高效率的组织结构

Union-Find算法3-Weighted Union with Path Compression!

- **Path Compression:**




```
private int find(int p)
{
    // 寻找p节点所在组的根节点，根节点具有性质id[root] = root
    while (p != id[p]) p = id[p];
    return p;
}
```

Add only 1 line of code!
Cost of find is still $\boxed{\Theta(depth[x])}$

```
private int find(int p)
{
    while (p != id[p])
    {
        // 将p节点的父节点设置为它的爷爷节点
        id[p] = id[id[p]];
        p = id[p];
    }
    return p;
}
```

Union-Find算法3-Weighted Union with Path Compression!

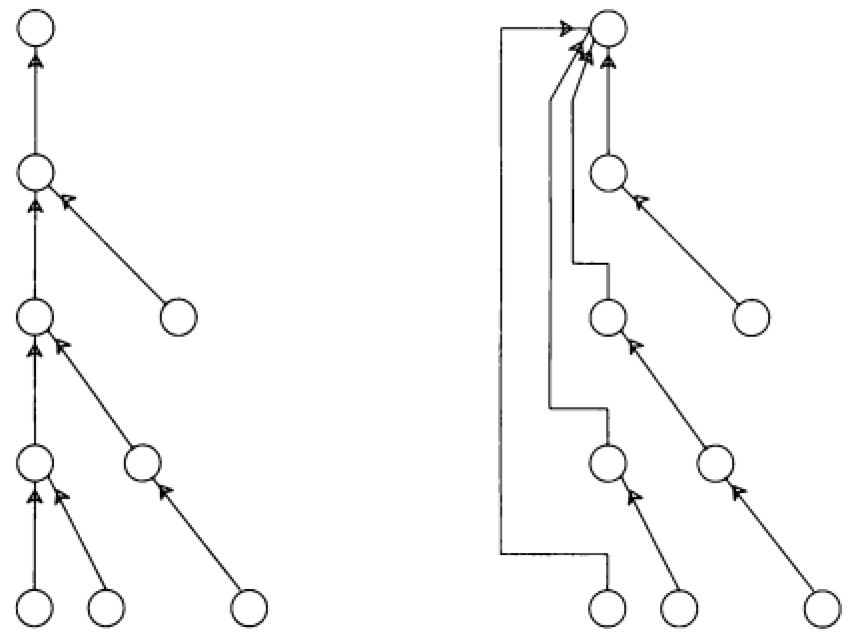
- **Path Compression_2(Textbook Page59):**

```
int find(int x){ //recording the walked path
    array path;

    while(id[x]!=x){
        p = id[p];
        path.add(p);
    }

    ForIndex(i, path.size()){
        id[path[i]] = p;
    }
    return p;
}
```

Cost of find is still $\boxed{\Theta(depth[x])}$



(a) (b)

Figure 4.17 Path compression: (a) Before. (b) After.

Complexity

Algorithm	Constructor	Union	Find
Quick-Find	N	N	1
Quick-Union	N	Tree height	Tree height
Weighted Quick-Union	N	lgN	lgN
Weighted Quick-Union With Path Compression	N	Very near to 1 (amortized)	Very near to 1 (amortized)

相比课本P59最后一段的path compression， 节约了保存中间路径的开销。均摊效率几乎相等， no visible difference.

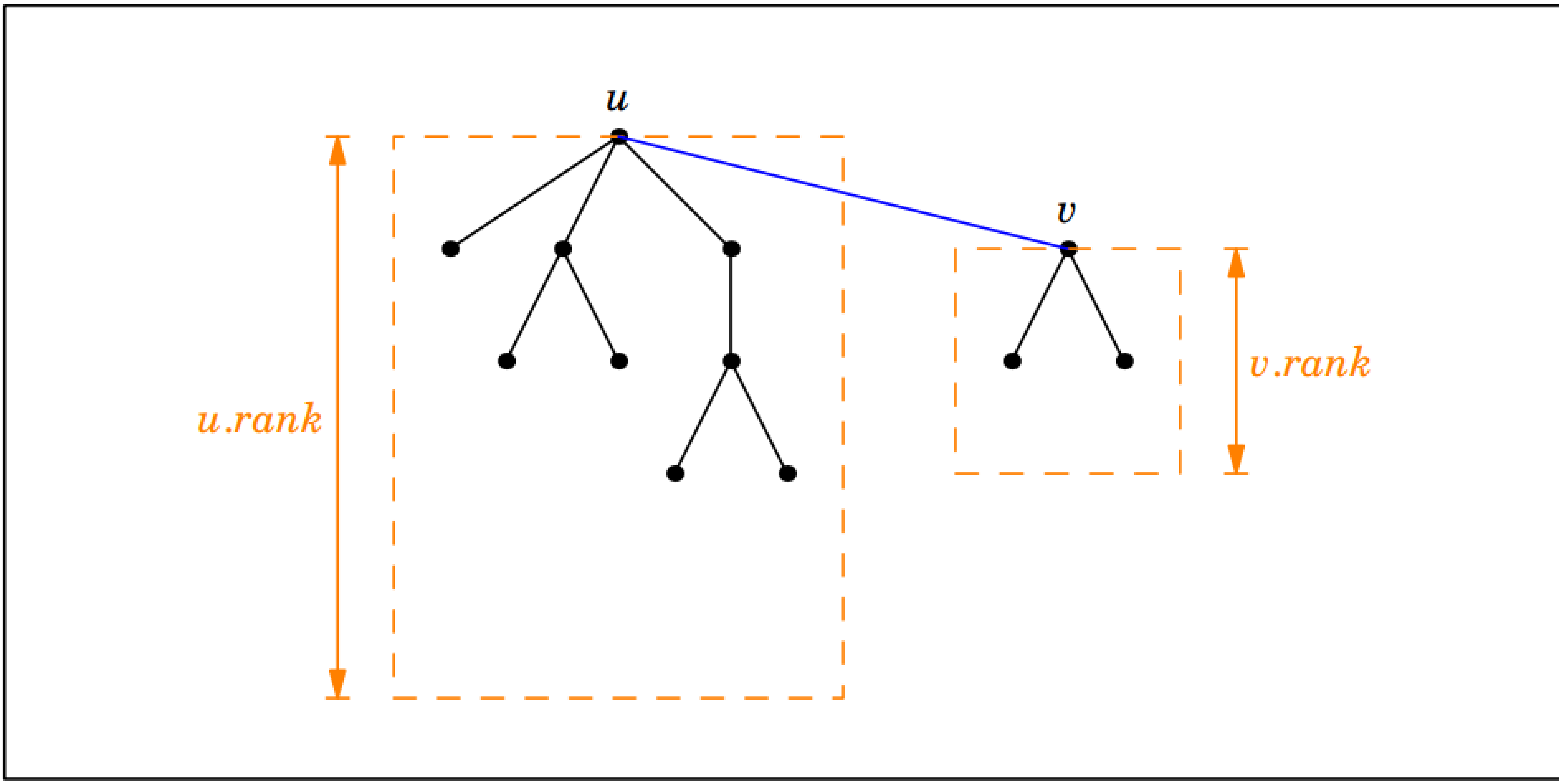
Conclusion

- For Disjoint-set data structure, the best solution is Union-Find with path compression

- 2 Tricks improve O(n) → O(lgn)→ O(lglglg..lgn)
  – Smaller tree merged into larger tree
  – Path compression

Limitation

- Very promising performance(Amortized, though.) – $\Theta(\log^*(n))$, not rigorous linear..
  - Operator log*(k) <=5 for k<=2^{65536} .
- Do NOT support any tasks related to path-searching. Unlike with DFS BFS scanning…



u
v
u.rank
v.rank