Data Structures and Algorithms

(CS210A)

Semester I – **2014-15**

Lecture 32

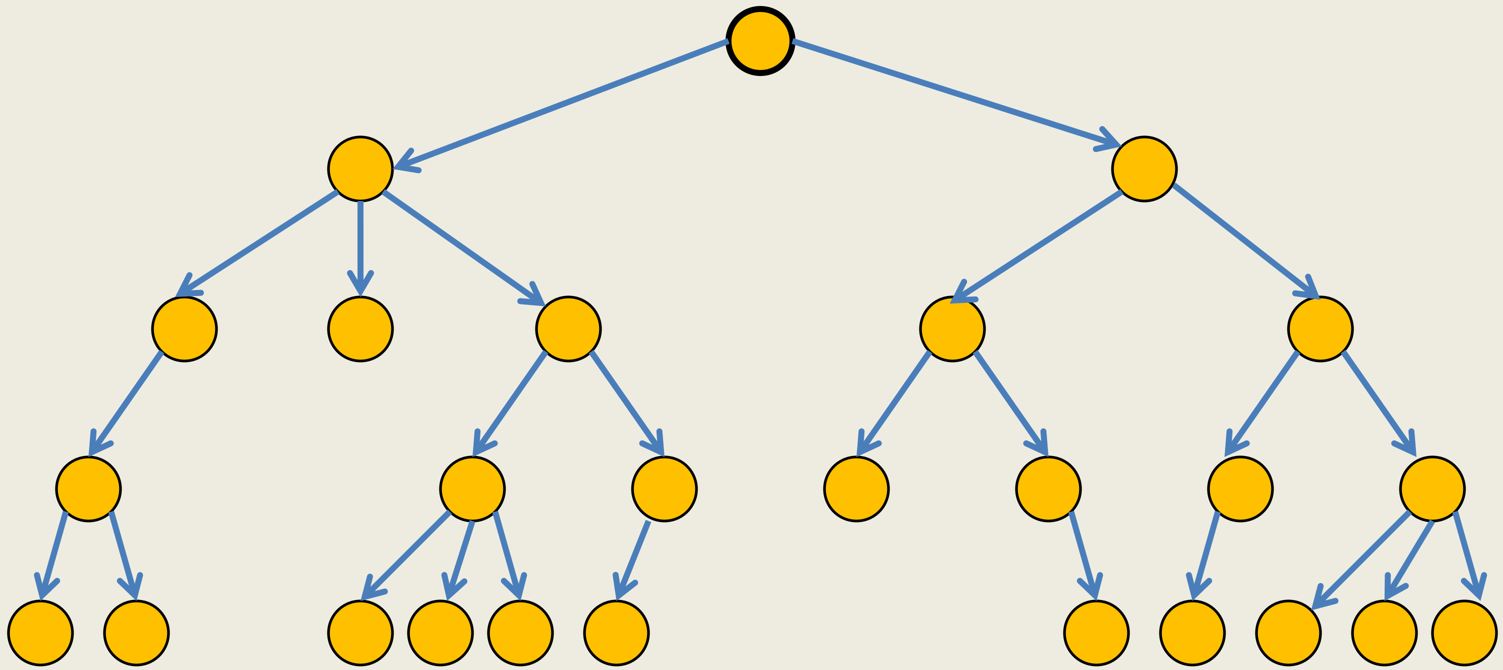
- **Magical application of binary trees – III**

Data structure for sets

Rooted tree

Revisiting and extending

A typical rooted tree we studied



Definition we gave:

Every vertex, except **root**, has exactly one **incoming** edge and has a path **from** the root.

Examples:

Binary search trees,

DFS tree,

BFS tree.

A typical rooted tree we studied

Question: what data structure can be used for representing a rooted tree ?

Answer:

Data structure 1:

- Each node stores a list of its children.
- To access the tree, we keep a pointer to the root node.

  (there is no way to access any node (other than root) directly in this data structure)

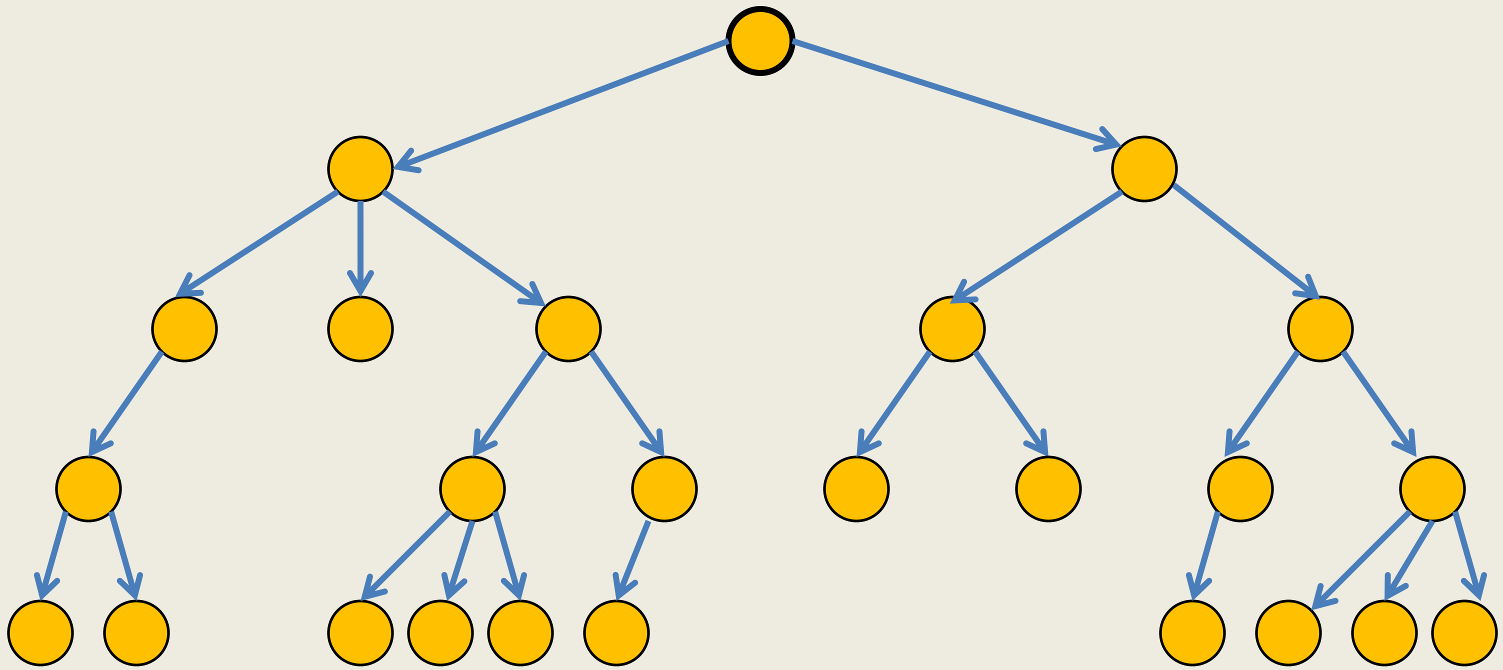
Data structure 2: **(**If nodes are labeled in a contiguous range [0..n-1])

rooted tree becomes an instance of a **directed graph**.

So we may use **adjacency list** representation.

Advantage: We can access each node directly.

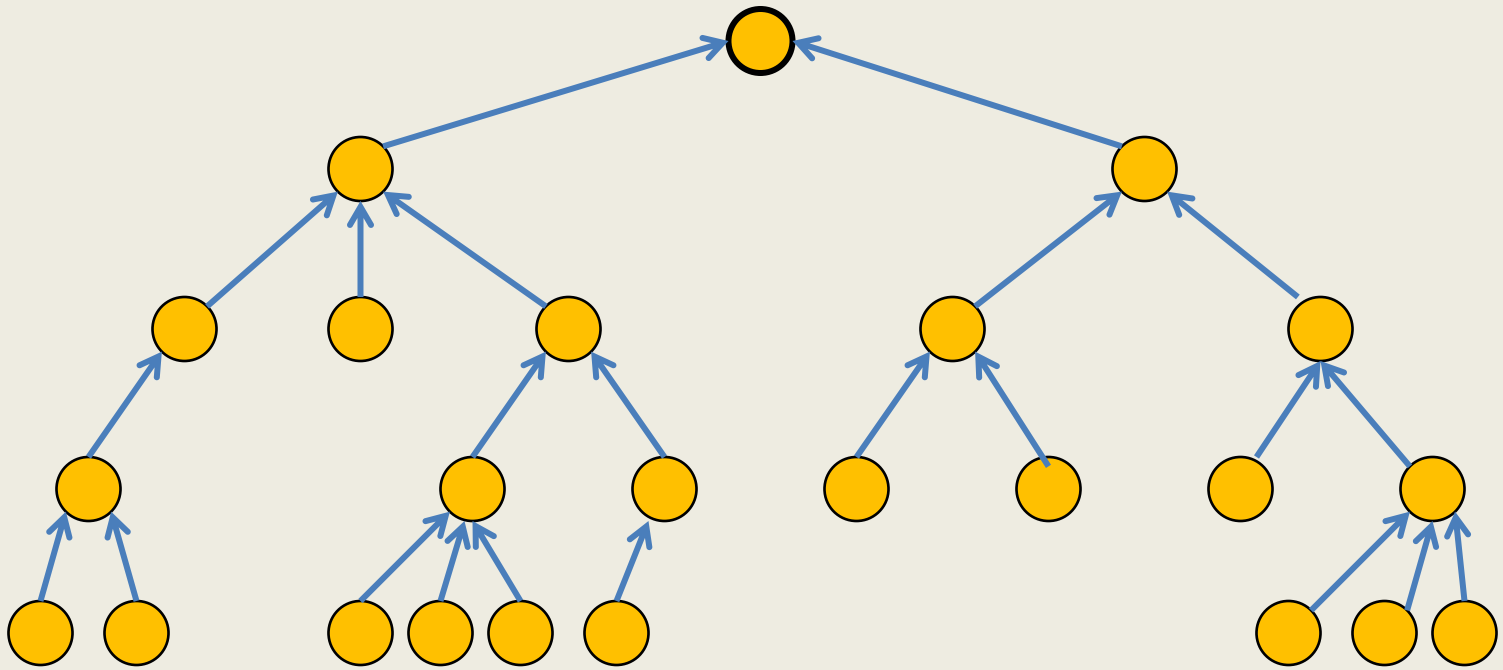
Extending the definition of rooted tree



Extended Definition:

Type 1: Every vertex, except **root**, has exactly one **incoming** edge and has a path **from** the root.

Extending the definition of rooted tree



Extended Definition:

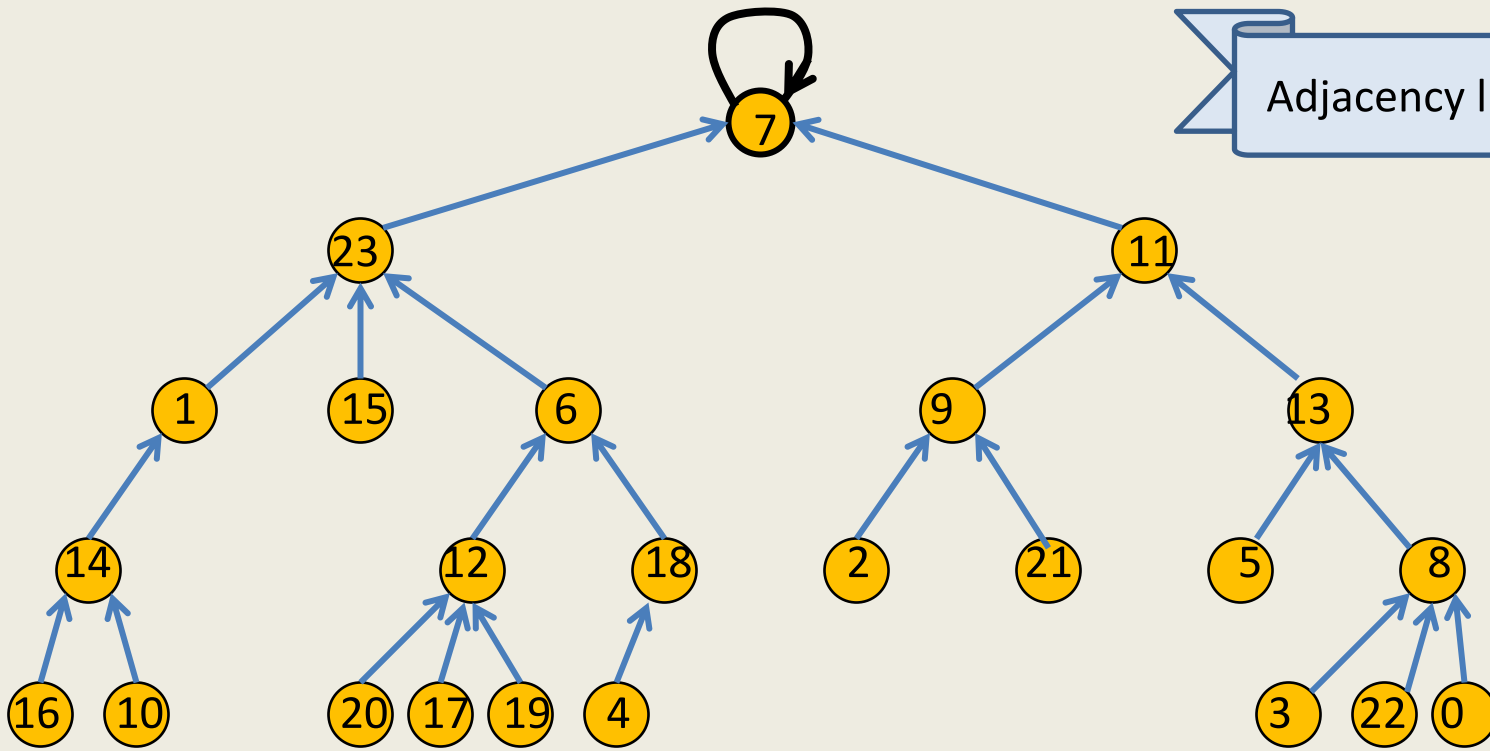
Type 1: Every vertex, except **root**, has exactly one **incoming** edge and has a path **from** the root.

OR

Type 2: Every vertex, except root, has exactly one **outgoing** edge and has a path **to** the root.

Data structure for rooted tree of type 2

Adjacency lists



If nodes are labeled in a **contiguous** range $[\mathbf{0}..\boldsymbol{n}-\mathbf{1}]$,

there is even simpler and more compact data structure

Guess ??

Parent	**8**	**23**	**9**	**8**	**18**	**13**	**23**	**7**	**13**	**11**	**14**	**7**	**6**	**11**	**1**	**23**	**14**	**12**	**6**	**12**	**12**	**9**	**8**	**7**
	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22	23

Application of rooted tree of type 2

Maintaining sets

Sets under two operations

Given: a collection of n singleton sets $\{0\}$, $\{1\}$, $\{2\}$, … $\{n-1\}$

Aim: a compact data structure to perform

- **Union**(i, j):

  Unite the two sets containing i and j.

- **Same_sets**(i, j):

  Determine if i and j belong to the same set.

Trivial Solution

Treat the problem as a graph problem: Connected component

- **V** = {0,…, $n-1$}, **E** = empty set initially.
- A set ⇔ a connected component.
- Keep array **Label[]** such that **Label**[i]=**Label**[j] iff i and j belong to the same component.

➔

Union(i, j) :

O(n) time

if (**Same_sets**(i, j) = **false**)

add an edge (i, j) and recompute connected components using **BFS/DFS**.

Sets under two operations

Given: a collection of n singleton sets $\{0\}$, $\{1\}$, $\{2\}$, … $\{n-1\}$

Aim: a compact data structure to perform

- **Union**(i, j):

  Unite the two sets containing i and j.

- **Same_sets**(i, j):

  Determine if i and j belong to the same set.

Efficient solution:

- A data structure which supports each operation in **O(log n)** time.

- **An additional heuristic**

  ➔ time complexity of an operation : practically **O**(1).

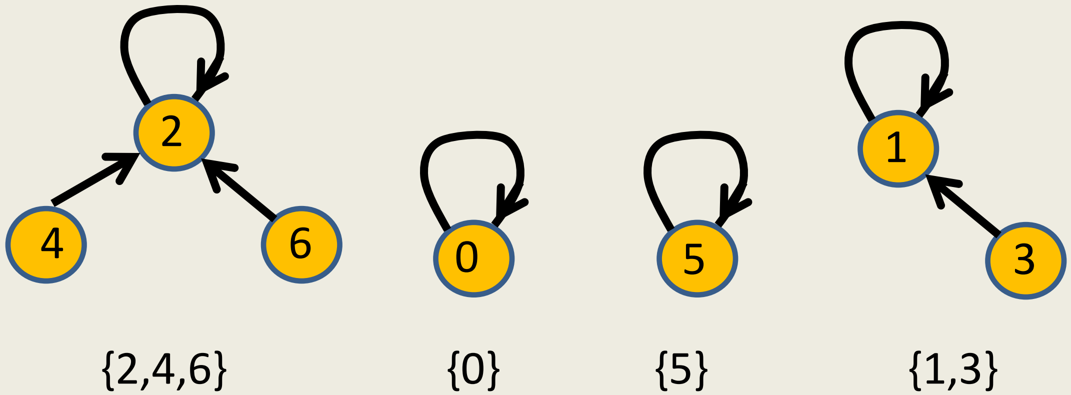
Data structure for sets

Maintain each set as a rooted tree.

Question: How to perform operation **Same_sets**(i, j) ?

Answer: Determine if i and j belong to the same tree.

(To do this, **find** root of i and root of j, and compare)



Data structure for sets

Maintain each set as a rooted tree.

Question: How to perform operation **Same_sets**(i, j) ?

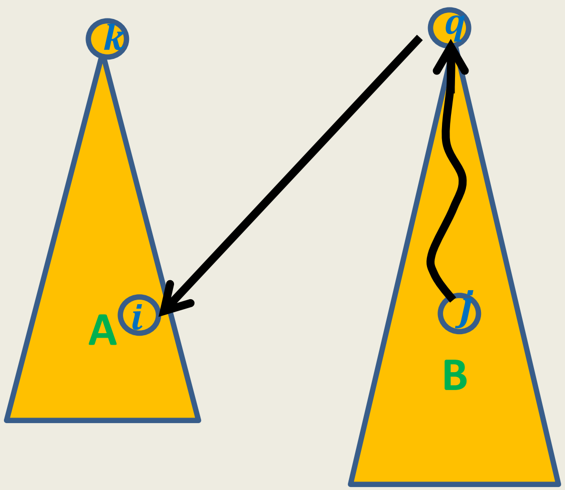
Answer: Determine if i and j belong to the same tree.

(To do this, **find** root of i and root of j, and compare)

Question: How to perform **Union**(i, j) ?

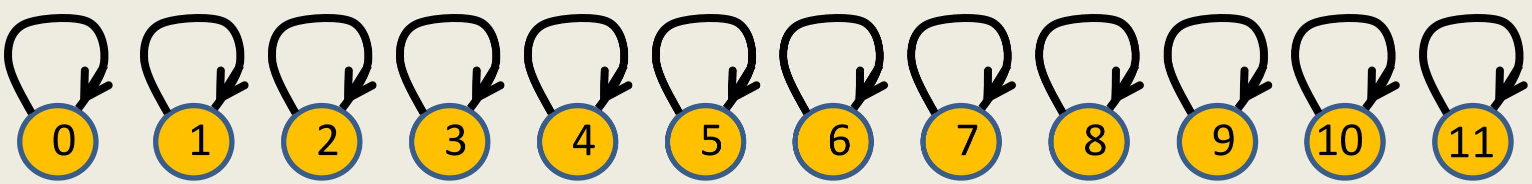
Answer:

- **find** root of j; let it be q.
- **Parent**(k) ← i.



A rooted tree as a data structure for sets

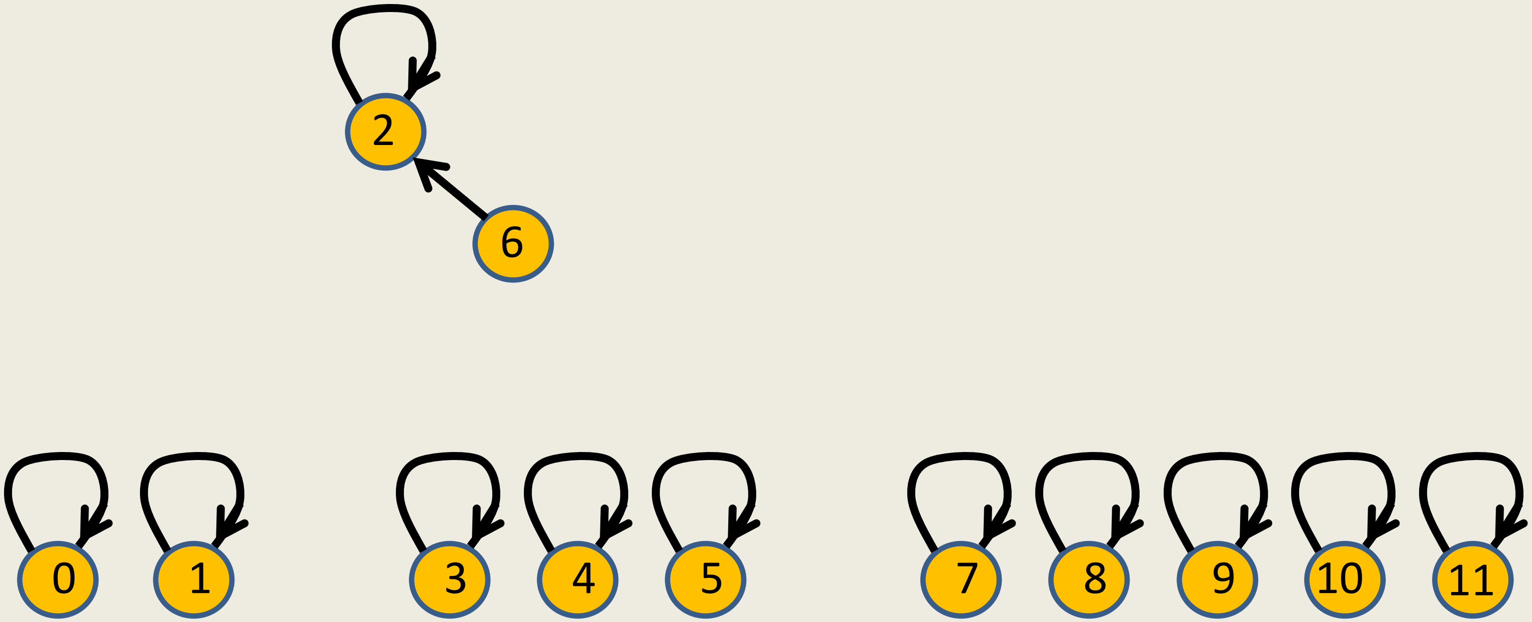
Union(2,6)



Parent	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
	0	1	2	3	4	5	6	7	8	9	10	11

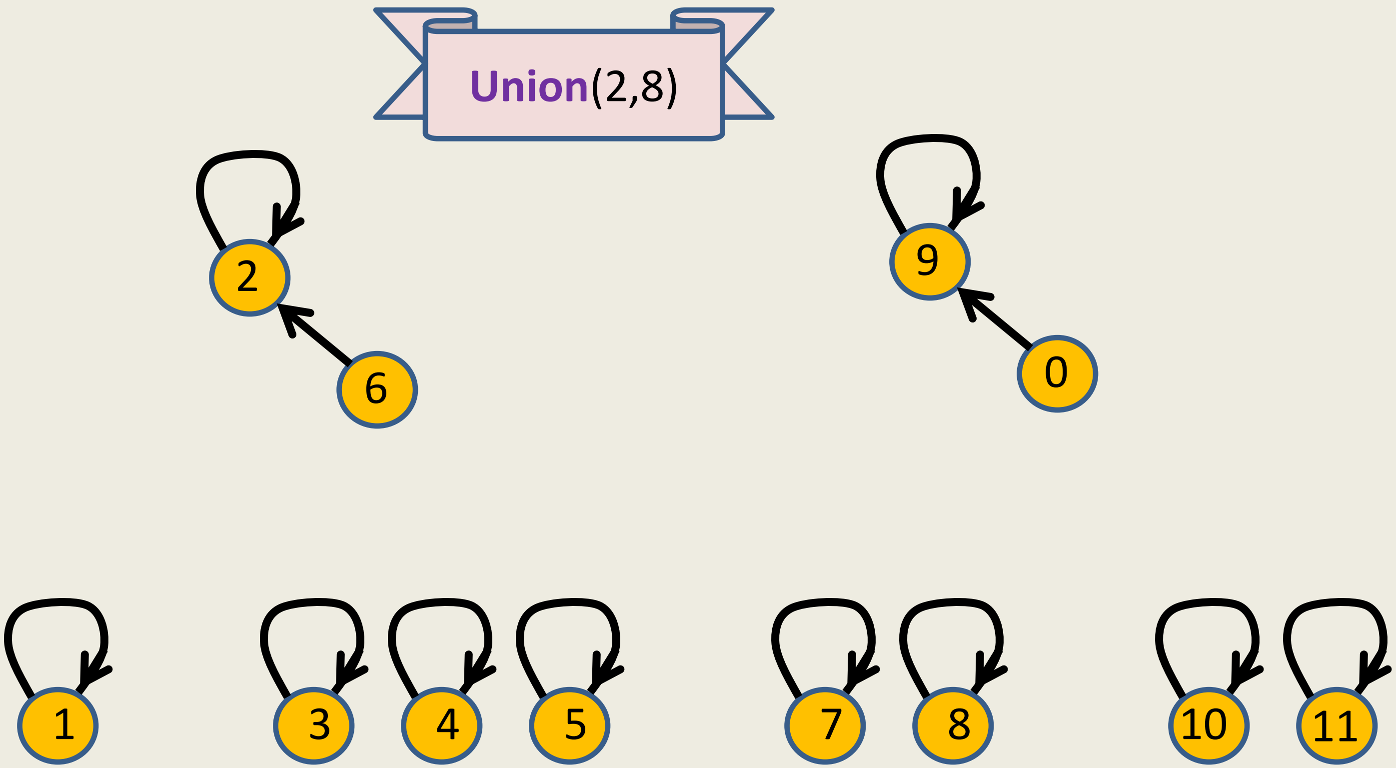
A rooted tree as a data structure for sets

Union(9,0)



Parent	**0**	**1**	**2**	**3**	**4**	**5**	**2**	**7**	**8**	**9**	**10**	**11**
	0	1	2	3	4	5	6	7	8	9	10	11

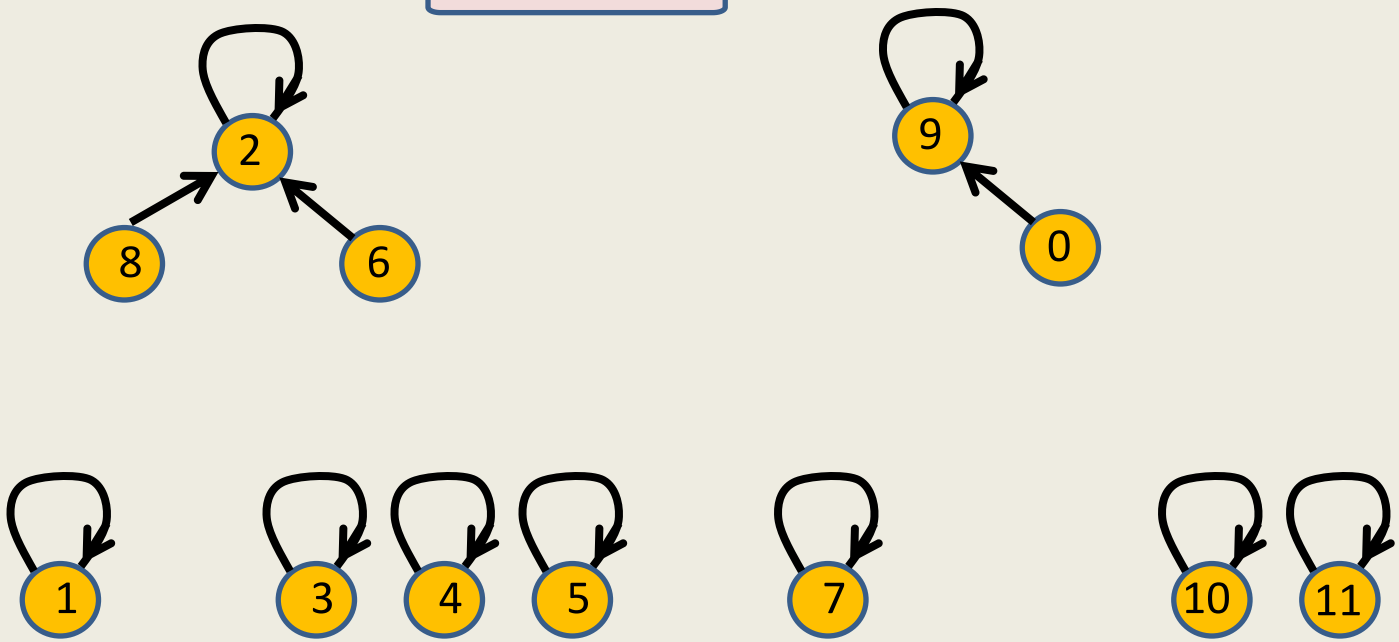
A rooted tree as a data structure for sets



Parent	**9**	**1**	**2**	**3**	**4**	**5**	**2**	**7**	**8**	**9**	**10**	**11**
	0	1	2	3	4	5	6	7	8	9	10	11

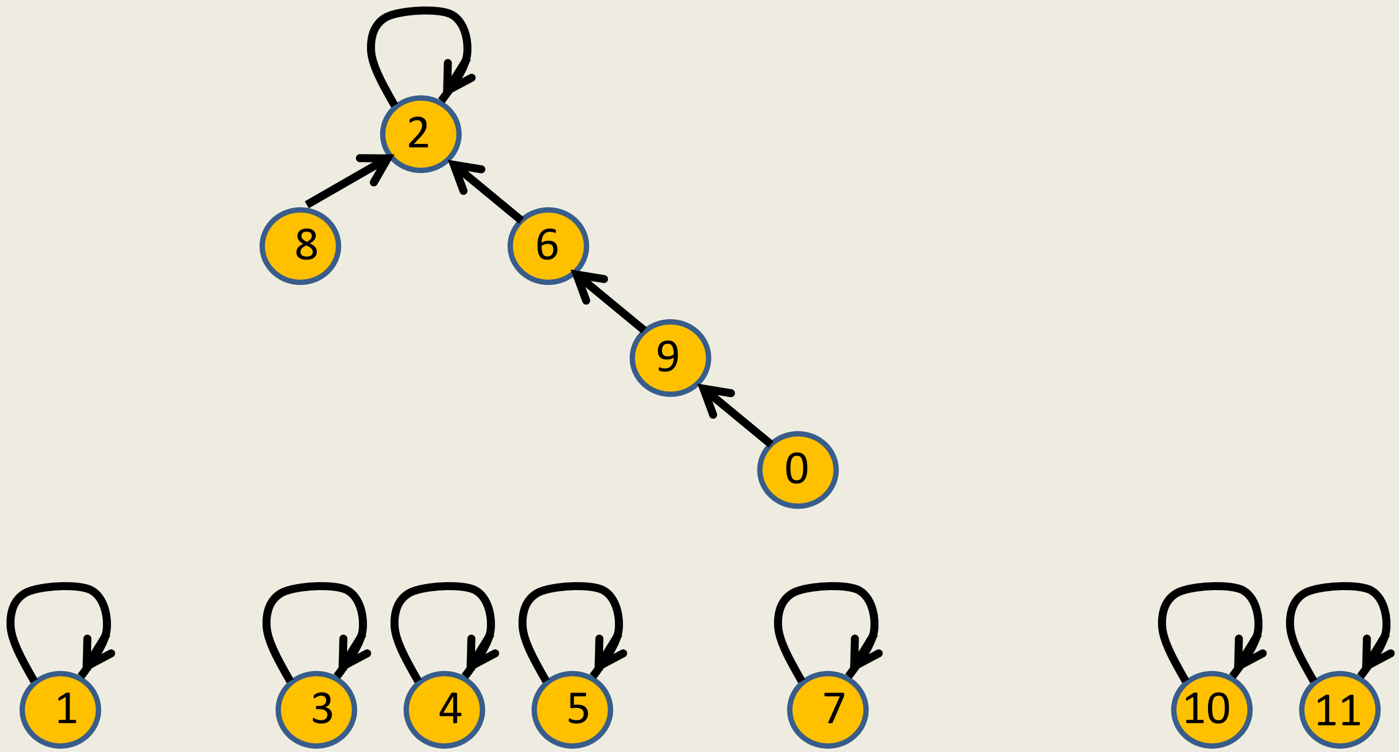
A rooted tree as a data structure for sets

Union(6,0)



Parent	**9**	**1**	**2**	**3**	**4**	**5**	**2**	**7**	**2**	**9**	**10**	**11**
	0	1	2	3	4	5	6	7	8	9	10	11

A rooted tree as a data structure for sets



Parent	**9**	**1**	**2**	**3**	**4**	**5**	**2**	**7**	**2**	**6**	**10**	**11**
	0	1	2	3	4	5	6	7	8	9	10	11

Pseudocode for Union and SameSet()

```
Find(i)    // subroutine for finding the root of the tree containing i
    If (Parent(i) = i)    return i ;
    else  return Find(Parent(i));
```

```
SameSet(i, j)
    k ← Find(i);
    l ← Find(j);
    If (k = l)    return true else return false


Union(i, j)
    k ← Find(j);
    Parent(k)← i;
```

Observation: Time complexity of **Union**(i, j) as well as **Same_sets**(i, j) is governed by the time complexity of **Find**(i) and **Find**(j).

Question: What is time complexity of **Find**(i) ?

Answer: depth of the node i in the tree containing i.

Time complexity of **Find**(i)

Union(0,1)

Union(1,2)

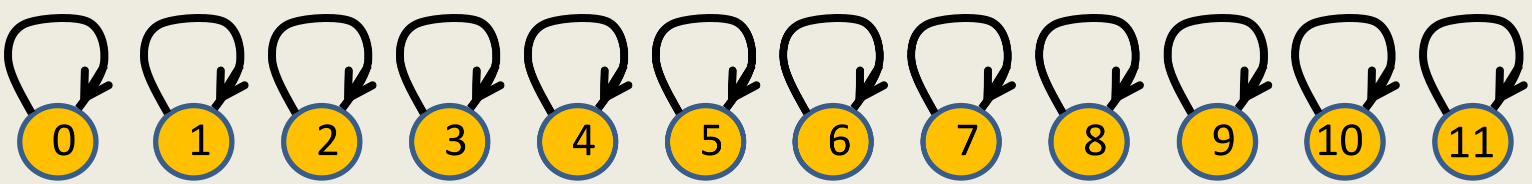
Union(2,3)

…

Union(9,10)

Union(10,11)

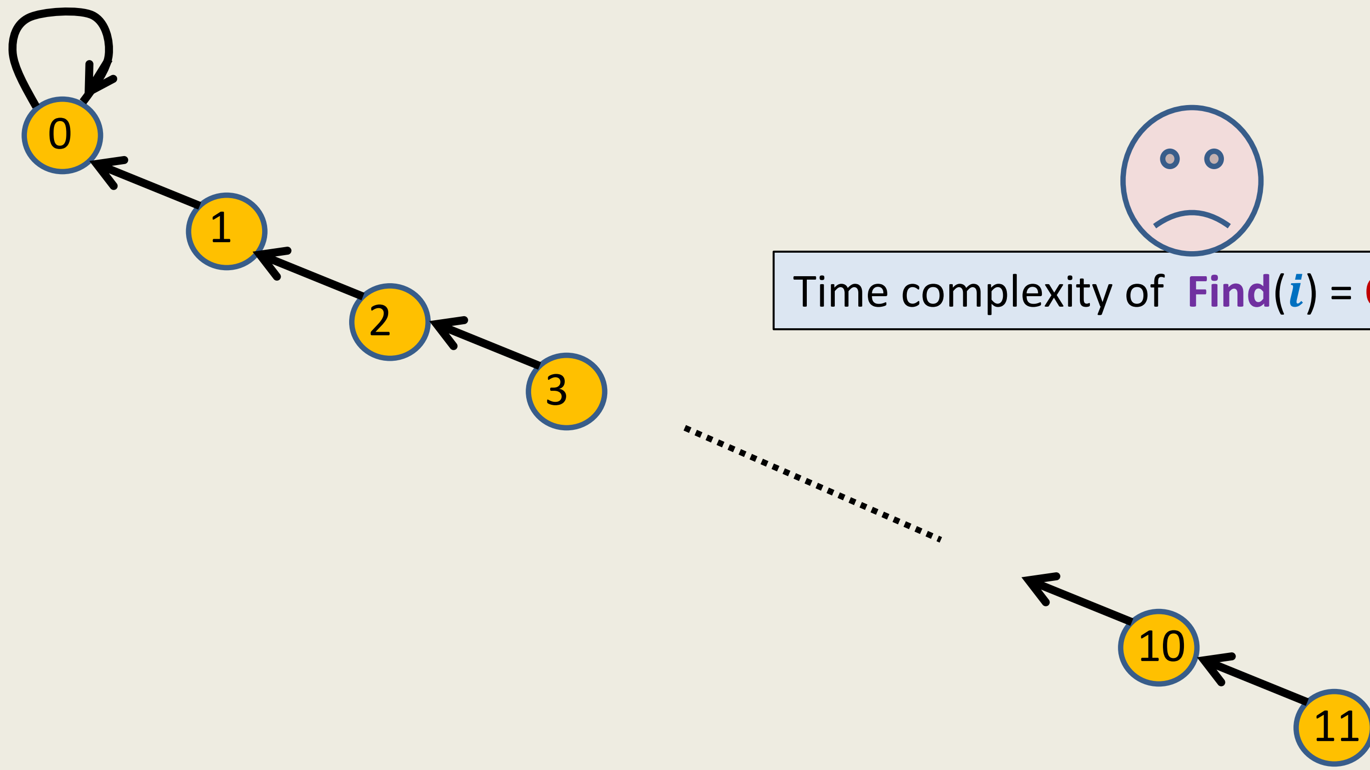
What will be the rooted tree structures after these union operations ?



Parent	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
	0	1	2	3	4	5	6	7	8	9	10	11

Time complexity of Find(i)

Union(0,1)
Union(1,2)
Union(2,3)
…
Union(9,10)
Union(10,11)



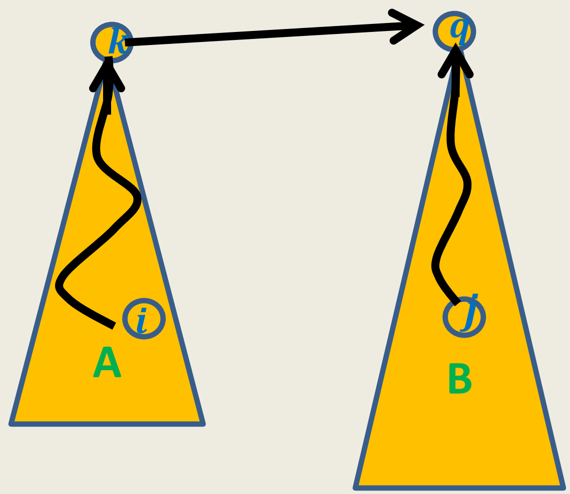
Time complexity of **Find**(i) = **O**(n)

Parent	**0**	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
	0	1	2	3	4	5	6	7	8	9	10	11

Improving the time complexity of Find(i)

Heuristic 1: Union by size

Improving the Time complexity



Key idea: Change the **union**(i,j) .

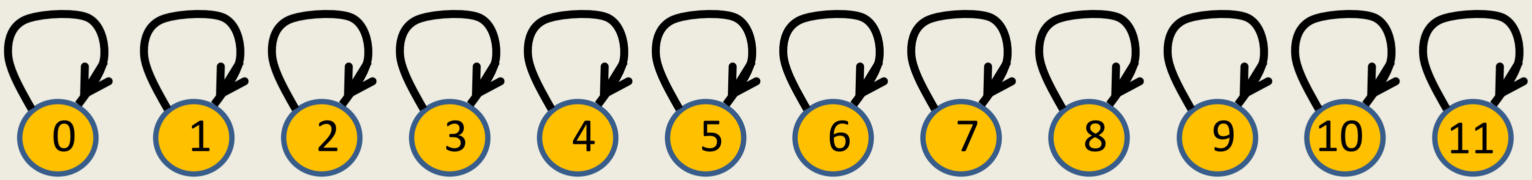
While doing **union**(i,j), hook the **smaller size** tree to the **root of the bigger size tree**.

For this purpose, keep an array **size[0,..,n-1]**

size[i] = number of nodes in the tree containing i

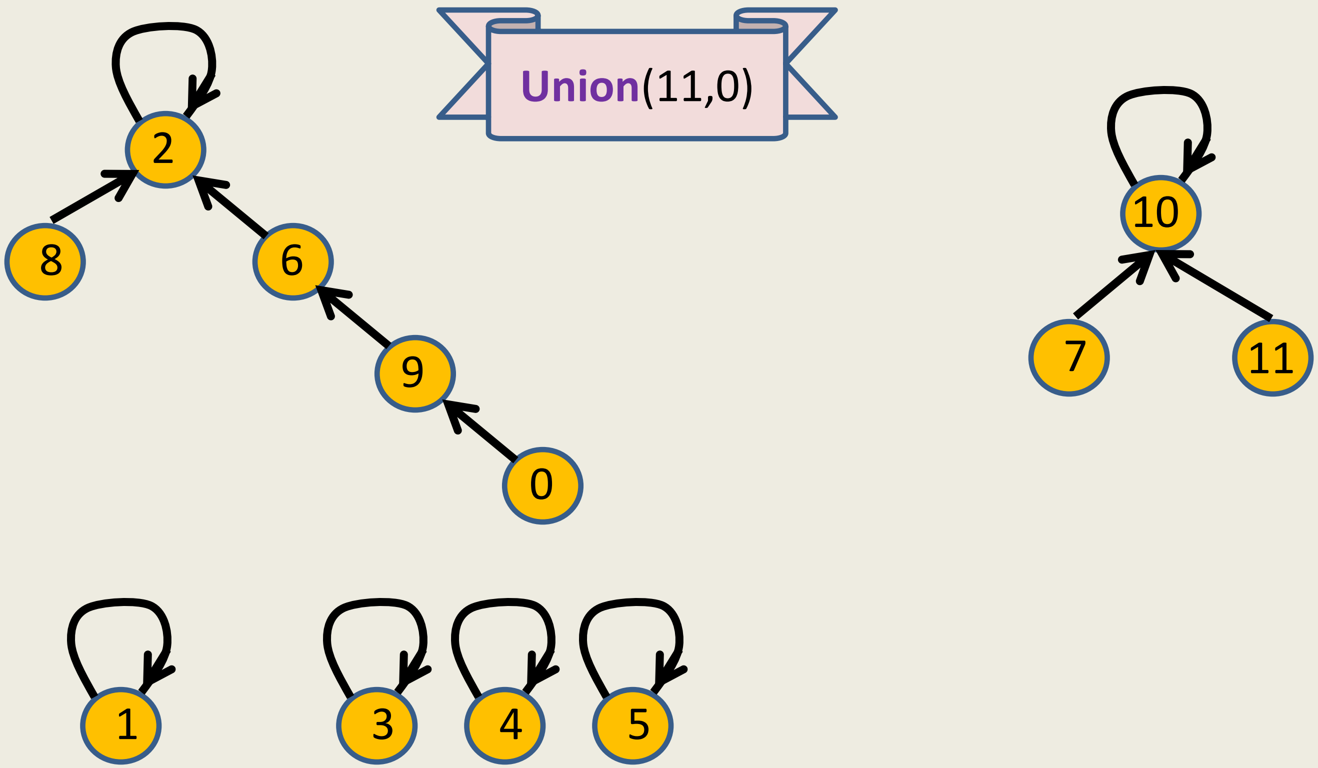
(**if** i is a **root** and zero otherwise)

Efficient data structure for sets



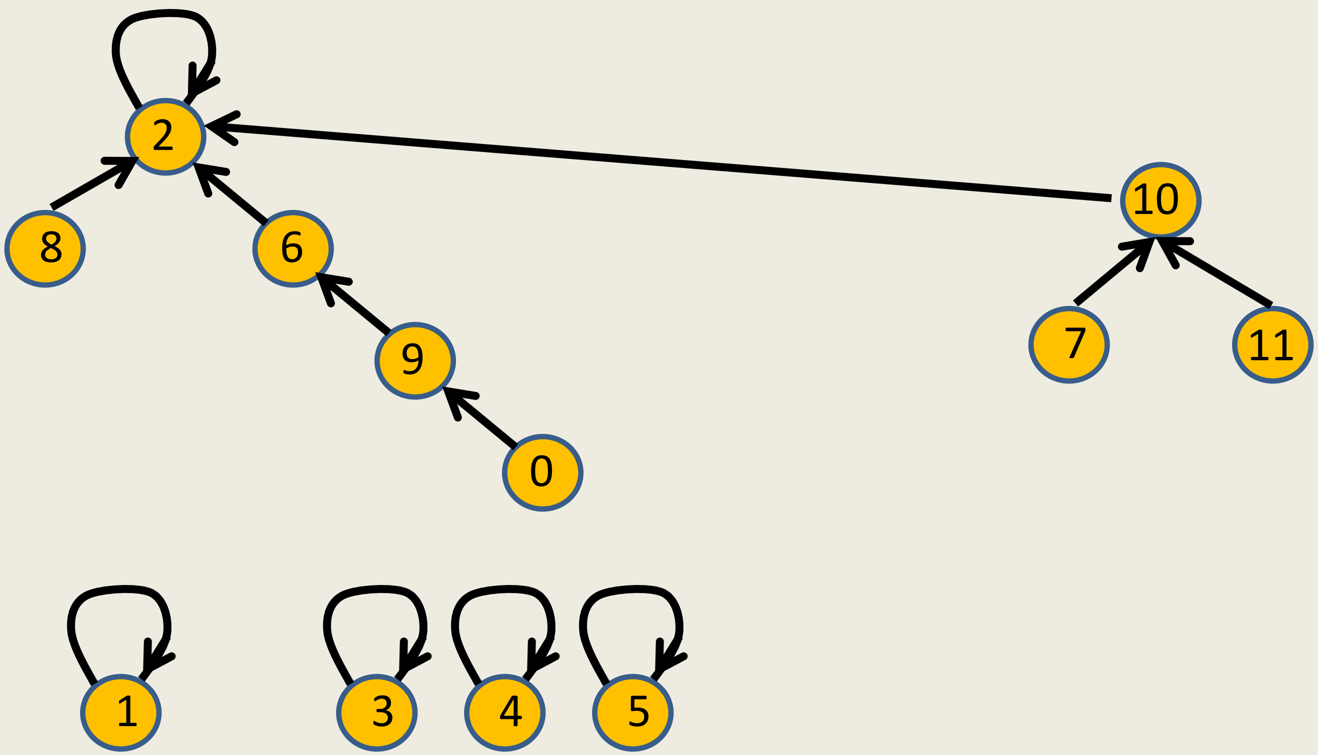
Parent	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**	**11**
	0	1	2	3	4	5	6	7	8	9	10	11
size	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**	**1**

Efficient data structure for sets



Parent	**9**	**1**	**2**	**3**	**4**	**5**	**2**	**10**	**2**	**6**	**10**	**10**
	0	1	2	3	4	5	6	7	8	9	10	11
size	**0**	**1**	**5**	**1**	**1**	**1**	**0**	**0**	**0**	**0**	**3**	**0**

Efficient data structure for sets



	0	1	2	3	4	5	6	7	8	9	10	11
Parent	**9**	**1**	**2**	**3**	**4**	**5**	**2**	**10**	**2**	**6**	**10**	**10**
size	**0**	**1**	**5**	**1**	**1**	**1**	**0**	**0**	**0**	**0**	**3**	**0**

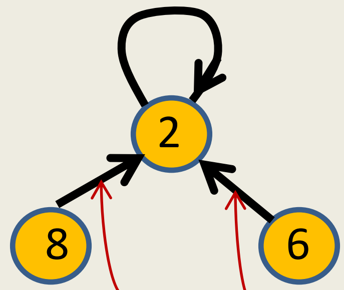
Pseudocode for modified Union

```
Union(i, j)
    k ← Find(i);
    l ← Find(j);
    If(size(k) < size(l) )
        l ← Parent(k);
        size(l) ← size(k) + size(l);
        size(k) ← 0;
    Else
        k ← Parent(l);
        size(k) ← size(k) + size(l);
        size(l) ← 0;
```

Question: How to show that **Find(i)** for any i will now take **O(log n)** time only ?

Answer: It suffices if we can show that **Depth(i)** is **O(log n).**

Can we infer history of a tree?

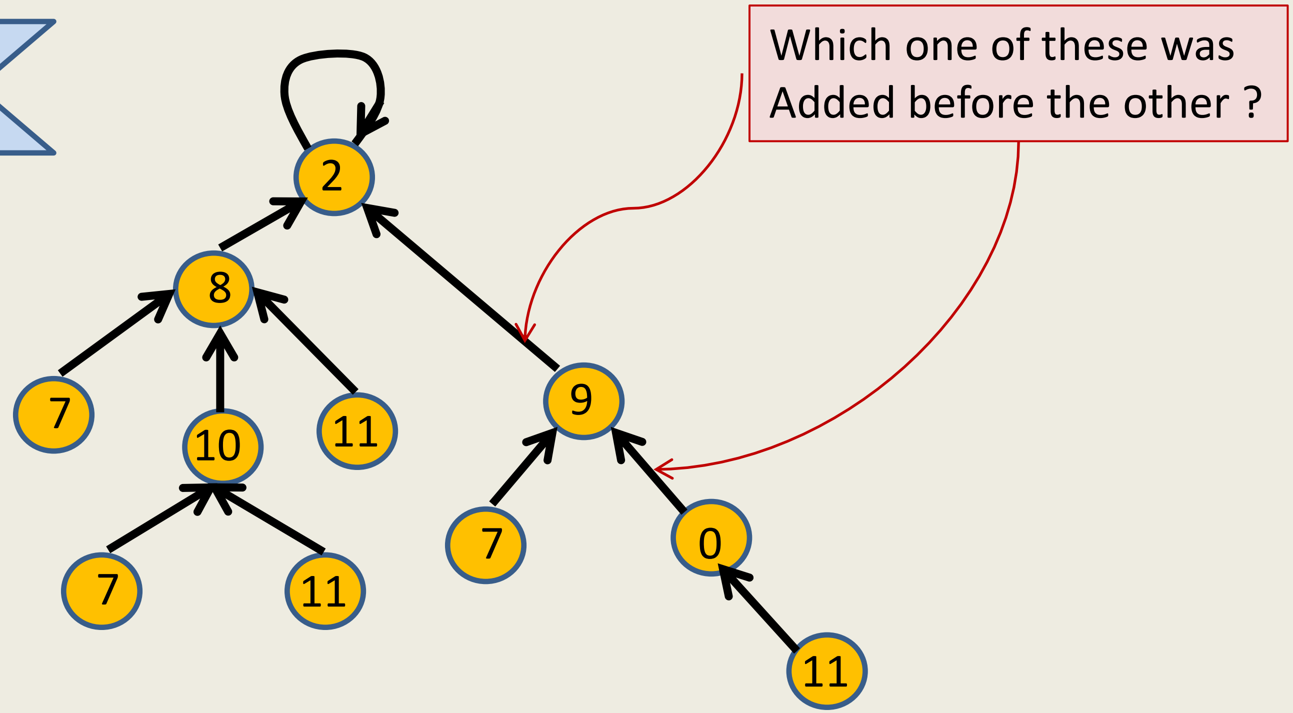


Which one of these was
Added before the other ?

Answer: Can not be inferred with any certainty ☹.

Can we **infer** history of a tree?

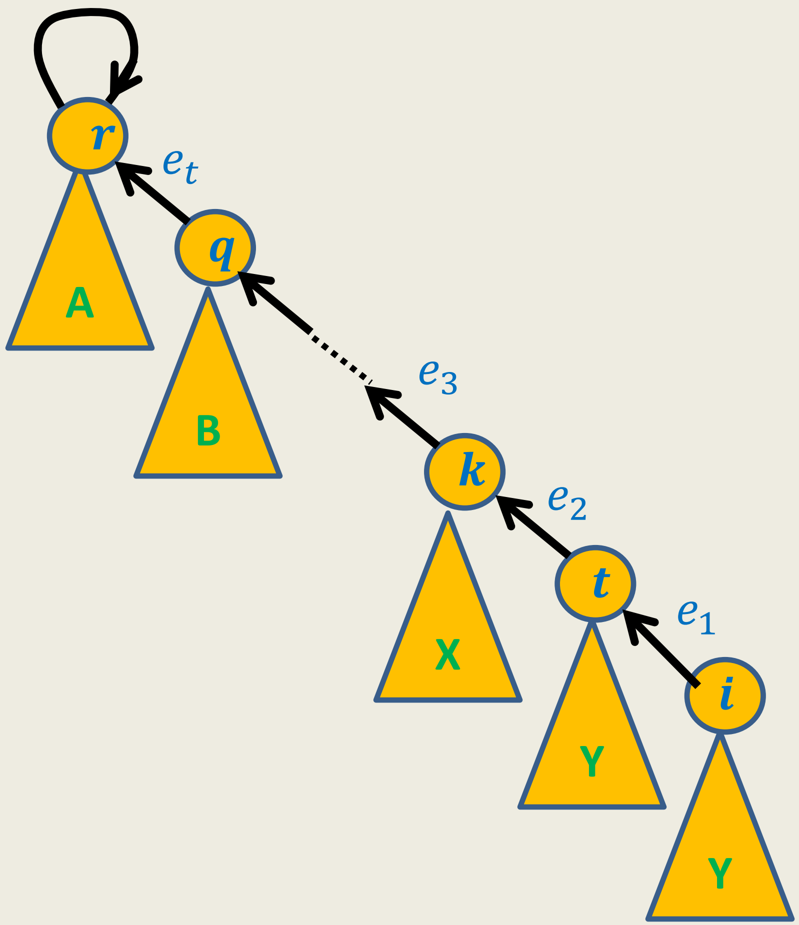
During union, we join **roots** of two trees.



→ (0→9) was added **before** (9→6).

Theorem: The edges on a **path** from node v to root were inserted in the order they appear on the **path**.

How to show that depth of any element = O(log n) ?



Let $e_1, e_2, \ldots, e_t$ be the edges on the path from i to the **root.**

Let us visit the history.
(how this tree came into being ?).

Edges $e_1, e_2, \ldots, e_t$ would have been added in the order:

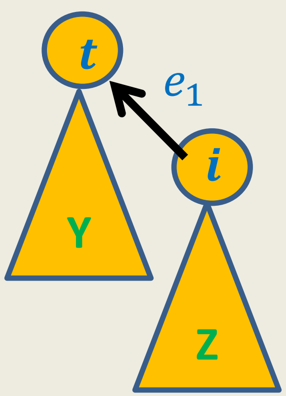
e_1
e_2
…
e_t

How to show that depth of any element = O(log n) ?

Let $e_1, e_2, \dots, e_t$ be the edges on the path from i to the **root.**

Consider the moment just before edge e_1 is inserted.

Let no. of elements in subtree $T(i)$ at that moment be n_i.



We added edge $i \rightarrow t$ (and **not** $t \rightarrow i$).

➔ no. of elements in $T(t) \geq n_i$.

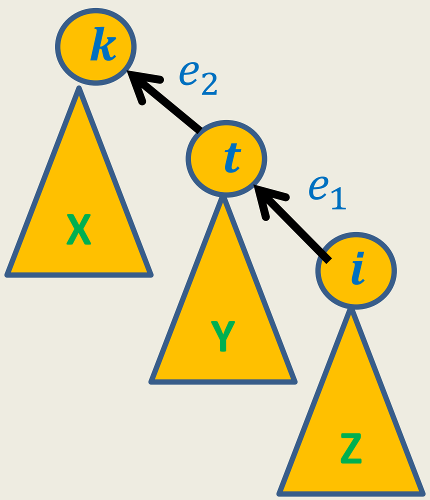
➔ After the edge $i \rightarrow t$ is inserted,

no. of element in $T(t) \geq 2n_i$

How to show that depth of any element = O(log n) ?

Let $e_1, e_2, \ldots, e_t$ be the edges on the path from i to the **root.**

Consider the moment just before edge e_2 is inserted.



no. of element in $T(t) \geq 2n_i$

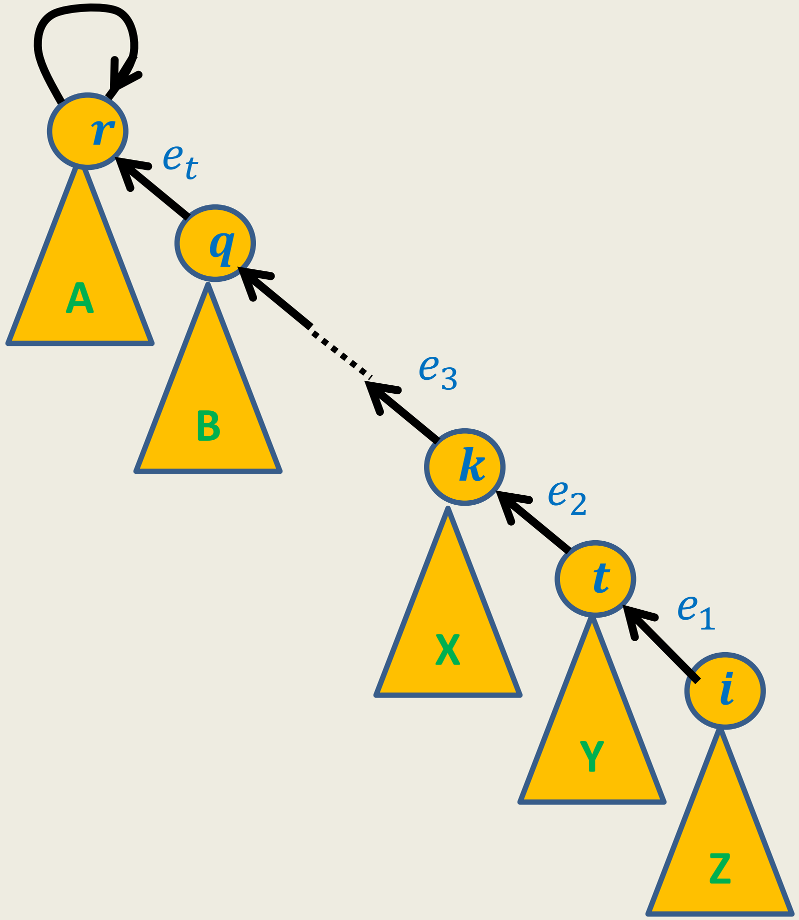
We added edge $t \rightarrow k$ (and **not** $k \rightarrow t$).

➔ # elements in $T(k) \geq 2n_i$.

➔ After the edge $t \rightarrow k$ is inserted,

no. of element in $T(k) \geq 4n_i$

How to show that depth of any element = O(log n) ?



Let $e_1, e_2, \ldots, e_t$ be the edges on the path from i to the **root.**

Arguing in a similar manner for edge $e_3, \ldots, e_t$ ➔

elements in $T(r)$ after insertion of $e_t \geq 2^t n_i$

Obviously $2^t n_i \leq n$

➔

Theorem: $t \leq \log_2 n$

Theorem: Given a collection of **n** singleton sets followed by a sequence of **union** and **find** operations, there is a data structure based on "union by size" heuristic that achieves **O**(log **n**) time per operation.

Question: Can we achieve even better bounds ?

Answer: Yes.

A new heuristic for better time complexity

Heuristic 2: Path compression

This is how this heuristic got invented

- The time complexity of a **Find(i)** operation is proportional to the depth of the node i in its rooted tree.
- If the elements are stored closer to the root, faster will the **Find()** be and hence faster will be the overall algorithm.

The algorithm for **Union** and **Find** was used in some application of **data-bases**.

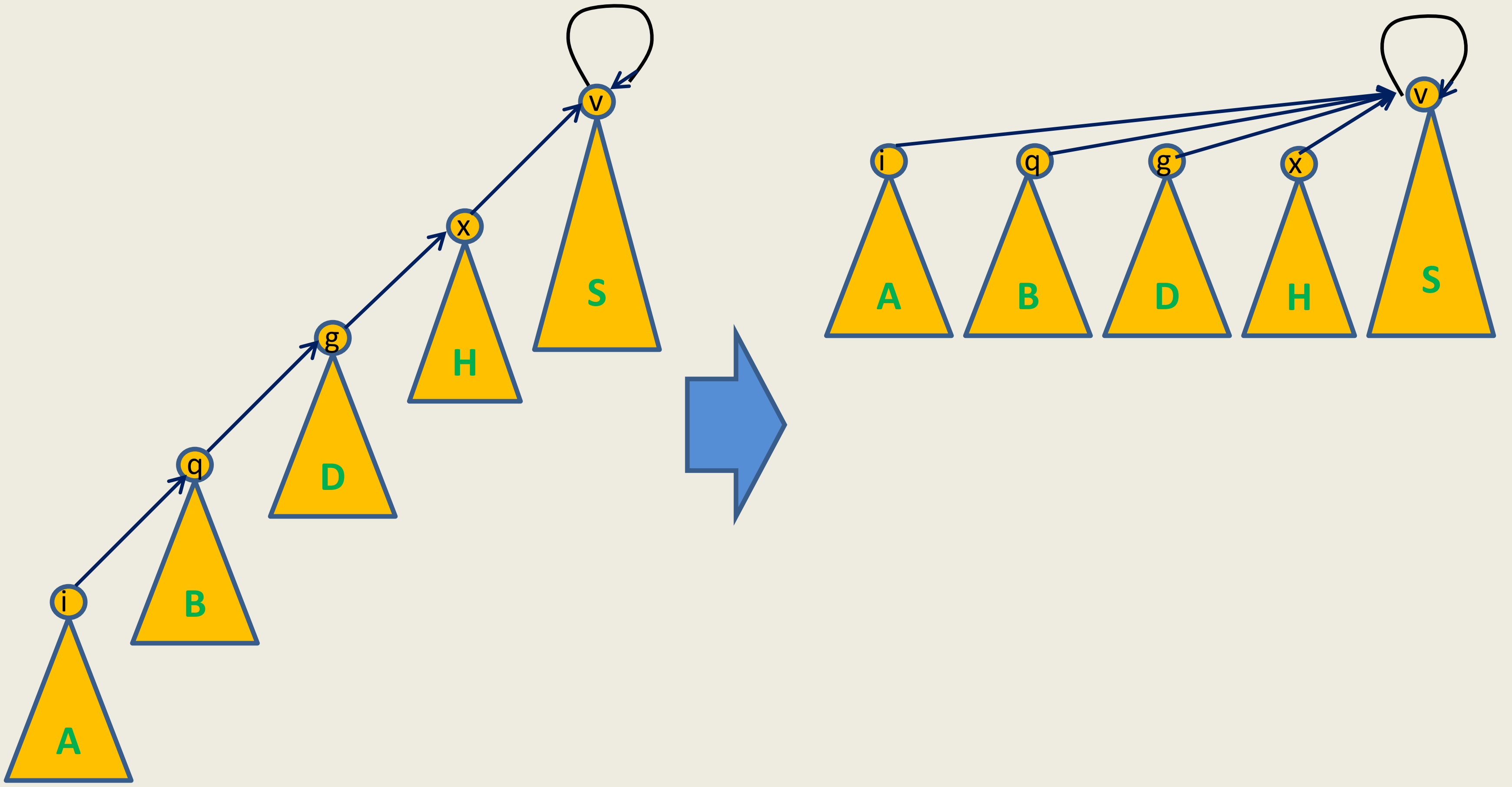
A clever programmer did the following modification to the code of **Find(i)**.

> While executing **Find(i)**, we traverse the path from node i to the root. Let v_1,v_2, …, v_t, be the nodes traversed with v_t being the root node. At the end of **Find(i)**, if we update parent of each v_k, $1 \le k < t$, to v_t , we achieve a reduction in depth of many nodes. This modification increases the time complexity of **Find(i)** by at most a constant factor. But this little modification increased the overall speed of the application very significantly.

The heuristic is called **path compression**. It is shown pictorially on the following slide.

It remained a <u>mystery for many years</u> to provide a theoretical explanation for its practical success.

Path compression during Find(i)



Pseudocode for the modified Find

```
Find(i)
    If (Parent(i) = i)    return i ;
    else
        j ← Find(Parent(i));
        Parent(i) ← j;
    return j
```

Concluding slide

Theorem: Given a collection of **n** singleton sets followed by a sequence of **m union** and **find** operations, there exists a data structure (using union by size heuristic and path compression heuristic) that achieves **O**(**m** + **n** **log*** **n**) time complexity.

Here **log*** **n** : the number of times we need to take **log** of a number till we get 1.

To see how "**extremely slow growing**" is the **log*** **n** function, see the following example.

If **n** = $2^{2^{2^{2^{2}}}}$ (> 2^{64000}),

Then **log*** **n** is just **5.**

Although log* **n** is effectively a small constant for every value of **n** in real life, the crazy theoreticians still **do not** consider it a constant☹ since it is an increasing function of **n**.

The proof will be discussed in one full lecture of **CS345.**

Keep pondering over it for next one year.☺

Lesson for all: There are simple algorithm which may have very difficult analysis.