Chapter 8 Programming by Tree Structure

Yonghui Wu

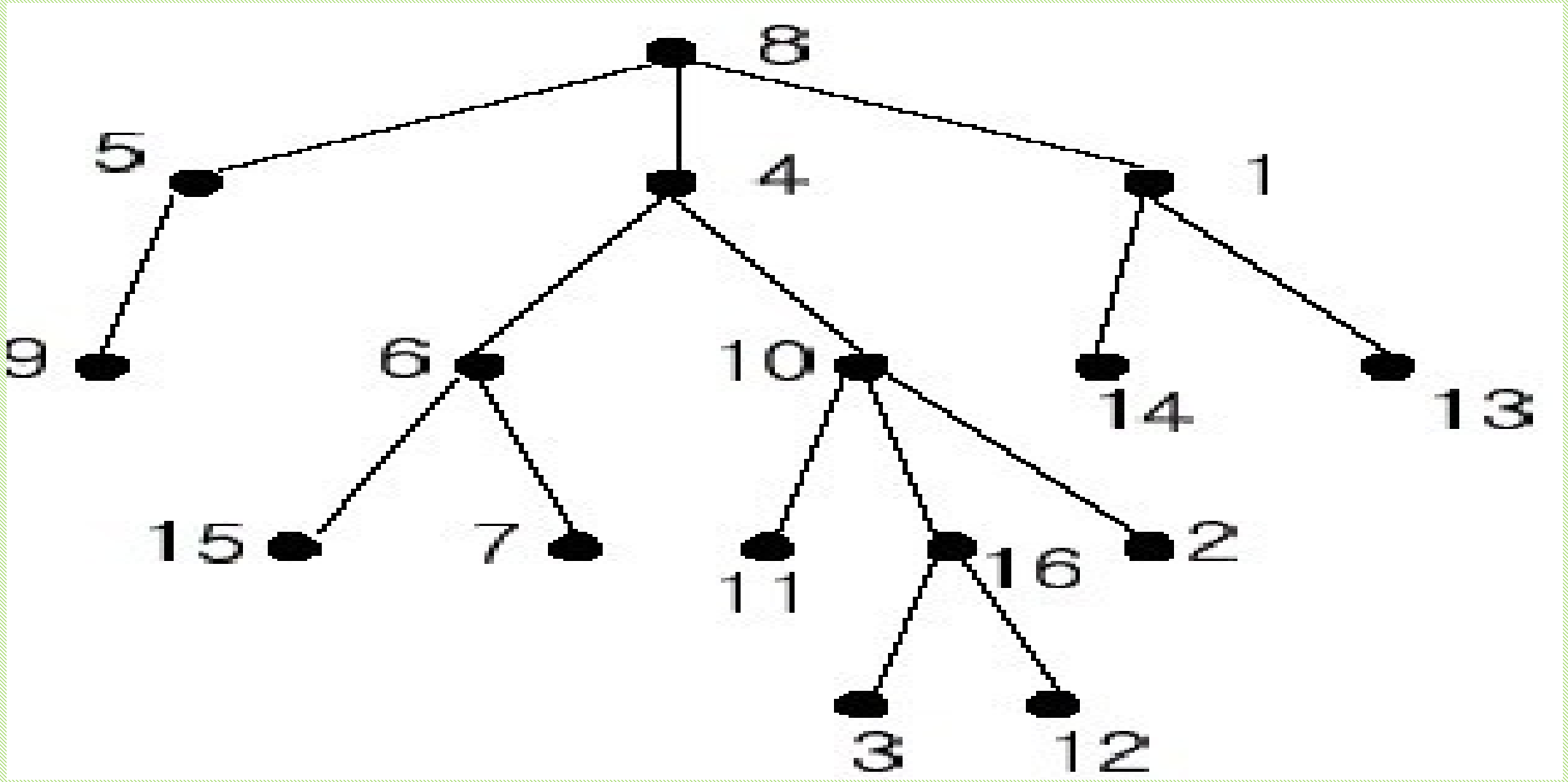
School of Computer Science, Fudan University

Email: yhwu@fudan.edu.cn

WeChat: 13817360465

- A tree can be defined recursively:
  - A tree is a collection of n vertices. The collection can be empty (n==0); otherwise a tree constitutes a distinguished vertex r, called the root; and zero or more nonempty subtrees that the root of each subtree is a child of r, and r is the parent of each subtree root.

An example for a rooted tree



- In the figure, each node is labeled with an integer from {1, 2,...,16}.
- Node 8 is the root of the tree.

- Node *x* is an ancestor of node y if node *x* is in the path between the root and node *y*.
  - Node 4 is an ancestor of node 16.
  - Node 10 is also an ancestor of node 16.
  - Nodes 8, 4, 10, and 16 are the ancestors of node 16.
- Remember that a node is an ancestor of itself.
  - Nodes 8, 4, 6, and 7 are the ancestors of node 7.

- A node x is called the nearest common ancestor of nodes y and z if x is a common ancestor of y and z and nearest to y and z among their common ancestors.
  - The nearest common ancestor of nodes 16 and 7 is node 4. Node 4 is nearer to nodes 16 and 7 than node 8 is.

8.1 Solving Hierarchical Problems by Tree Traversal

- A hierarchical structure can be modelled mathematically as a rooted tree:
  - The root of the tree forms the top level, and the children of the root are at the same level, under their common parent.
  - Vertices in a rooted tree constitute a partially ordered set. And the relations between vertices constitute relations of partial orders.

- Hierarchical problems can be represented as tree structures and can be solved by tree traversal.
- Tree traversal (also known as tree search) refers to the process of visiting (examining and/or updating) each vertex in a tree exactly once, in a systematic way.

Preorder traversal

- Preorder traversal:
  - Visit the root;
  - Preorder traversal for subtrees from left to right;
- void *preorder*(int *v*);
- { visit vertex *v*;
- for (*i* ∈ the set of adjacent vertices for *v*) // Pre-order traverse all adjacent unvisited vertices for *v*
- if (vertex *i* isn't visited)
- *preorder*(*i*);
- };

Postorder traversal

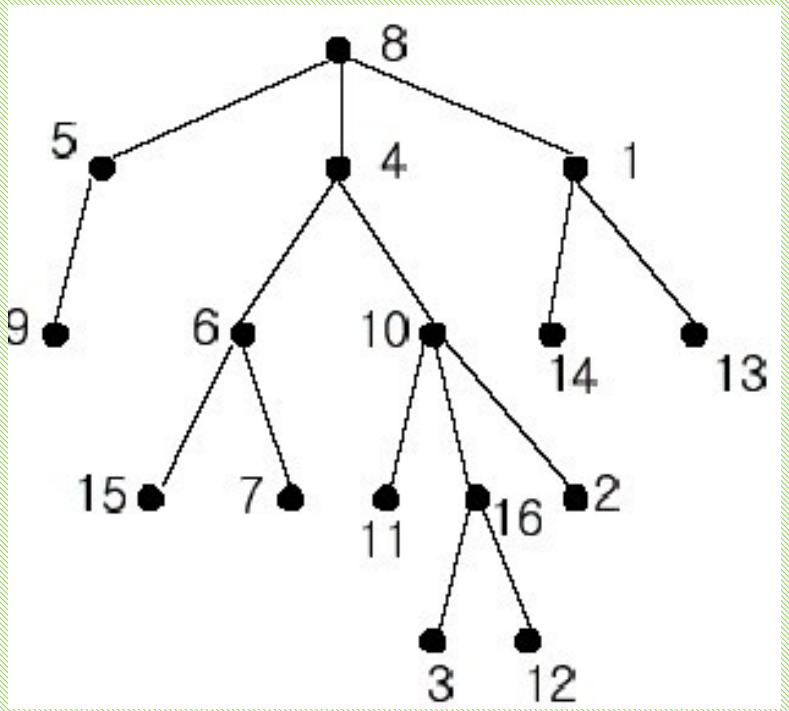
- Postorder traversal:
  - Postorder traversal for subtrees from left to right;
  - Visit the root.
- void *postorder*(int *v*);
- { for (*i* ∈ the set of adjacent vertices for *v*) // Post-order traverse all adjacent unvisited vertices for *v*
- if (vertex *i* isn't visited)
- *postorder* (*i*);
- visit vertex *v*;
- };

- An STL container can be used to define a multiple linked list for a tree. For example, a multiple linked list *adj*[*n*] can be defined as a vector, where *adj*[*x*].*push_back*(*y*) is to push *y* into the list for the children of *x*, and *y*=*adj*[*x*].*pop_back*( ) is to get a vertex from the list for the children of *x*.

8.1.1 Nearest Common Ancestor

- **Source: ACM Taejon 2002**
- **IDs for Online Judge: POJ 1330**

- A rooted tree is a well-known data structure in computer science and engineering. An example is shown below:



- In the figure, each node is labeled with an integer from {1, 2,...,16}. Node 8 is the root of the tree. Node *x* is an ancestor of node y if node *x* is in the path between the root and node *y*. For example, node 4 is an ancestor of node 16. Node 10 is also an ancestor of node 16. As a matter of fact, nodes 8, 4, 10, and 16 are the ancestors of node 16. Remember that a node is an ancestor of itself. Nodes 8, 4, 6, and 7 are the ancestors of node 7. A node x is called a common ancestor of two different nodes *y* and *z* if node *x* is an ancestor of node *y* and an ancestor of node *z*. Thus, nodes 8 and 4 are the common ancestors of nodes 16 and 7. A node *x* is called the nearest common ancestor of nodes *y* and *z* if *x* is a common ancestor of *y* and *z* and nearest to *y* and *z* among their common ancestors. Hence, the nearest common ancestor of nodes 16 and 7 is node 4. Node 4 is nearer to nodes 16 and 7 than node 8 is.

- For other examples, the nearest common ancestor of nodes 2 and 3 is node 10, the nearest common ancestor of nodes 6 and 13 is node 8, and the nearest common ancestor of nodes 4 and 12 is node 4. In the last example, if *y* is an ancestor of *z*, then the nearest common ancestor of *y* and *z* is *y*.
- Write a program that finds the nearest common ancestor of two distinct nodes in a tree.

- **Input**
- The input consists of *T* test cases. The number of test cases (*T*) is given in the first line of the input file. Each test case starts with a line containing an integer *N*, the number of nodes in a tree, $2 \leq N \leq 10{,}000$. The nodes are labeled with integers 1, 2,..., *N*. Each of the next *N* -1 lines contains a pair of integers that represent an edge --the first integer is the parent node of the second integer. Note that a tree with *N* nodes has exactly *N* - 1 edges. The last line of each test case contains two distinct integers whose nearest common ancestor is to be computed.
- **Output**
- Print exactly one line for each test case. The line should contain the integer that is the nearest common ancestor.

Analysis

- From each node in a tree, there is only one path to the root. Therefore any pair of nodes has common ancestors in a tree. A tree is represented with representation of parents and representation of multiple linked list. Each node's level number can be gotten by preorder traversal (the root's level number is 0, its children's level number is 1, and so on). The multiple linked list is represented by Class *vector*. And an integer array is used to represent parents and hierarchical data.

- The algorithm finding the nearest common ancestor for node *x* and node *y* is as follows.
- while (*x*≠*y*)
- { if (the level number of *x* is great than the level number of *y*)
- *x*=the parent of *x*;
- else
- *y*= the parent of *y*;
- }
- When the loop ends, *x* is the nearest common ancestor.

8.2 Union-Find Sets Supported by Tree Structure

- In some applications, *n* elements are divided into several groups. Each group is a set.
- Because such problems are mainly related to union and search for sets, they are called union-find sets.

Union-Find Sets

- Union-find sets are disjoint sets $S=\{S_1, S_2, \ldots, S_r\}$, where set S_i has an element $rep[S_i]$, called a representative.
- Any two different sets in S are disjoint.

Three operations for union-find sets

- 1. *Make_Set*(x):
- For union-find sets $S=\{S_1, S_2, \ldots, S_r\}$, a set containing only one element $\{x\}$ is added into union-find sets S, and $rep[\{x\}]=x$, where x is not in any S_i, $1\le i\le r$.
- Initially for each element x, *Make_Set*(x) is called.

Three operations for union-find sets

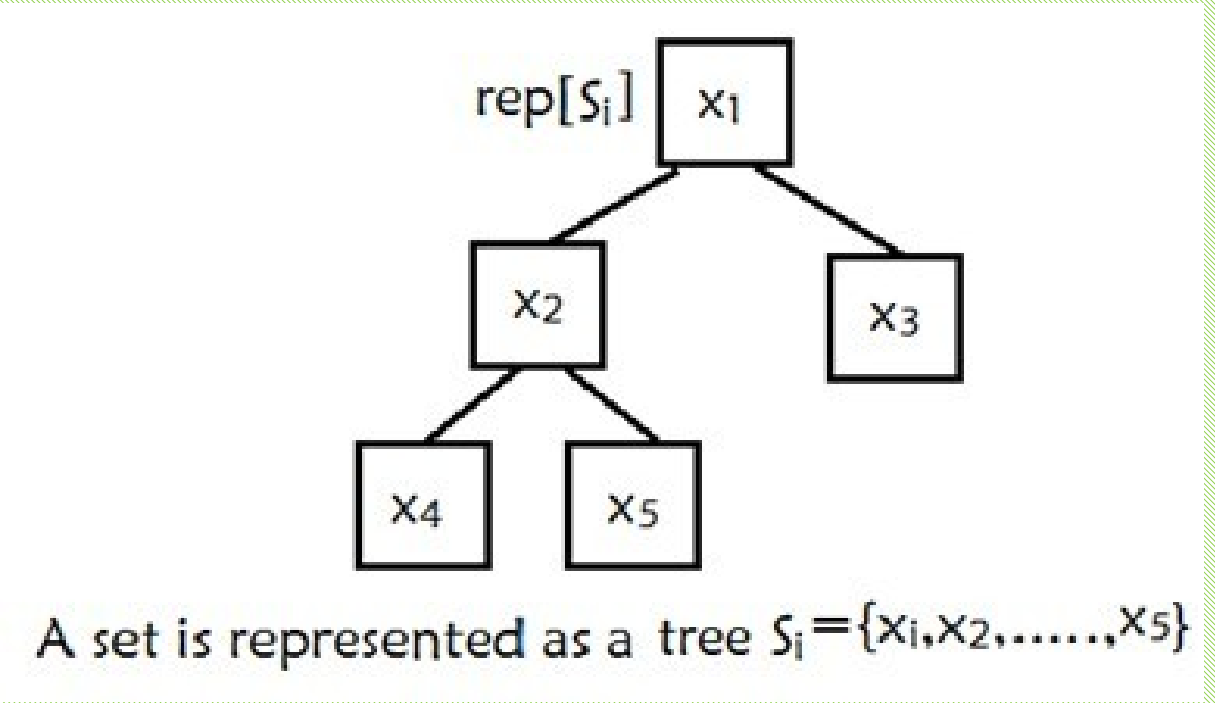
- 2. *join*(x, y):
- Merge two different sets containing x and y respectively. That is, S_x and S_y are deleted from S, and $S_x \cup S_y$ is added into S.

Three operations for union-find sets

- 3. *set_find*(*x*):
- Return representative *rep*[S_x] for Set S_x containing *x*.

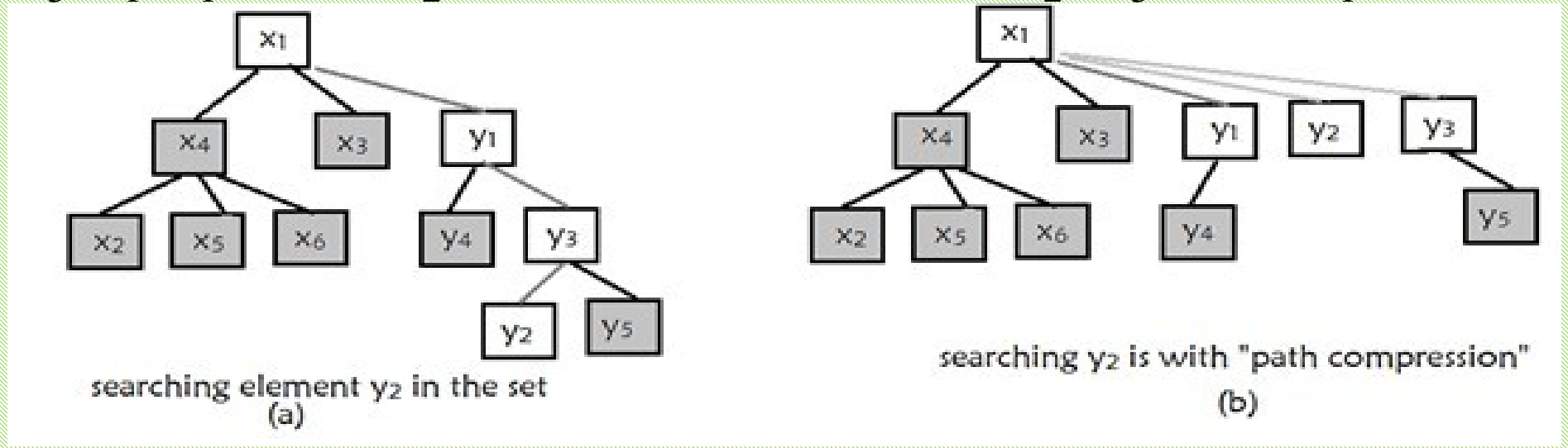
The storage structure for union-find sets

- **Tree Structure:**
- A set is represented as a tree, where the root is the representative for the set.



- Each node p has a pointer $set[p]$ pointing to its parent.
- If $set[p]<0$, p is the root node.
- Initially, a set is constructed for each element, that is, $set[x]=-1$ $(1 \le x \le n)$.

search operation: the search is with "path compression"

- the search with "path compression", to reduce the depth of the tree in the search process.
  - For example, to search element y_2 in the set. The path is y_2-y_3-y_1-x_1 from y_2. So set pointers for y_2, y_3, and y_1 point to x_1.
- 

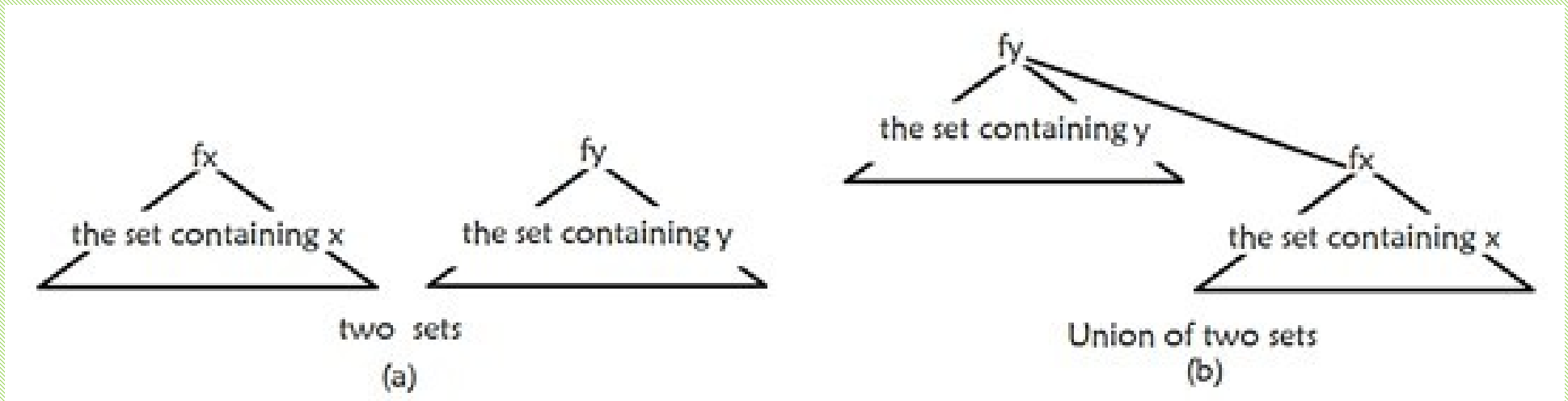
the search is with "path compression"

- from node *x*, through set pointers the root of the tree *f* (*set*[*f*]<0) is found;
- set pointers for all nodes on the path from *x* to *f* point to *f* to compress the path.

```
int set_find(int p)      // Search the representative of the set containing p, and compress the path
{
    if (set[p]<0)
        return p;
    return set[p]=set_find(set[p]);
}
```

Merging two sets

- Merging two sets is to connect roots for the two corresponding trees. That is, merging the set containing *x* (the tree root is *fx*) and the set containing *y* (the tree root is *fy*) is to let the set pointer for *fx* point to *fy*.



merging algorithm

- Calculate root *fx* in the tree for the union set containing *x*, and calculate root *fy* in the tree for the union-find set containing *y*.
- if *fx*==*fy*, then *x* and *y* are in the same union-find set;
- else the set containing *x* is merged into the set containing *y*, that is, the set pointer for *fx* points to *fy*.

- void *join*(int *p*, int *q*) // Merging the set containing *p* into the set containing *q*
- {
- *p*=*set_find*(*p*);
- *q*=*set_find*(*q*);
- if (*p*!=*q*)
- *set*[*p*]=*q*;
- }

- Search with "path compression" can reduce the length of a tree and can improve the time complexity.
- In algorithm complexity, an union-find set represented as a tree is better than as a linear list.

FRIENDS

- Source: Bulgarian National Olympiad in Informatics 2003
- IDs for Online Judge: UVA 10608

- There is a town with N citizens. It is known that some pairs of people are friends. According to the famous saying that “The friends of my friends are my friends, too” it follows that if A and B are friends and B and C are friends then A and C are friends, too.
- Your task is to count how many people there are in the largest group of friends.

- **Input**
- The first line of the input consists of N and M, where N is the number of town's citizens ($1 \leq N \leq 30000$) and M is the number of pairs of people ($0 \leq M \leq 500000$), which are known to be friends. Each of the following M lines consists of two integers A and B ($1 \leq A \leq N$, $1 \leq B \leq N$, $A \neq B$) which describe that A and B are friends. There could be repetitions among the given pairs.
- **Output**
- The output should contain one number denoting how many people there are in the largest group of friends.

Analysis

- A group of friends is a set. Calculating friends' relationships is the union operation of sets.
- Suppose
  - *set*[*k*] represents the representative in the group of friends containing *k*;
  - *s*[*k*] is the number of friends in the group of friends containing *k*;
  - *max* is the number of friends in the largest group of friends.

- Function *set_find*(*p*) is used to find the representative in the group of friends containing *p*. That is, *set*[*p*] is to be calculated.

- Function *join*(*p*, *q*) is used to merge two groups of friends containing *p* and *q* respectively.
  - Representatives in the group of friends containing *p* and *q* are found (*p*=*set_find*(*p*); *q*=*set_find*(*q*)).
  - If the two groups are different (*p*!=*q*), the group of friends containing *p* is merged into the group of friends containing *q* (*set*[*p*]=*q*); accumulate the number of friends in the group containing *q* (*s*[*q*]+=*s*[*p*]); and *max* is adjusted (*max*=*s*[*q*]>*max*? *s*[*q*]: *max*).

- First, each citizen is a group of friends; that is, *set*[*i*]=-1; *s*[*i*]=1; *max*=1.
- Then, while a pair of friends *A* and *B* is input, Function *join*(*A*, *B*) is called.
- Finally *max* is the solution to the problem.

8.2.1 Find them, Catch them

- **Source: POJ Monthly--2004.07.18**
- **IDs for Online Judge: POJ 1703**

- The police office in Tadu City decides to say ends to the chaos, as launch actions to root up the TWO gangs in the city, Gang Dragon and Gang Snake. However, the police first needs to identify which gang a criminal belongs to. The present question is, given two criminals; do they belong to a same clan? You must give your judgment based on incomplete information. (Since the gangsters are always acting secretly.)

- Assume N ($N \leq 10^5$) criminals are currently in Tadu City, numbered from 1 to N. And of course, at least one of them belongs to Gang Dragon, and the same for Gang Snake. You will be given M ($M \leq 10^5$) messages in sequence, which are in the following two kinds:
- 1. D [a] [b]
- where [a] and [b] are the numbers of two criminals, and they belong to different gangs.
- 2. A [a] [b]
- where [a] and [b] are the numbers of two criminals. This requires you to decide whether a and b belong to a same gang.

- **Input**
- The first line of the input contains a single integer T ($1 \le T \le 20$), the number of test cases. Then T cases follow. Each test case begins with a line with two integers N and M, followed by M lines each containing one message as described above.
- **Output**
- For each message "A [a] [b]" in each case, your program should give the judgment based on the information got before. The answers might be one of "In the same gang.", "In different gangs." and "Not sure yet."

Analysis

- Criminals in Gang Dragon and criminals in Gang Snake are two different sets respectively. Suppose *set*[*d*] is the representative of the set containing *d*, and *set*[*d+n*] is the representative of the “opposite” set for the set containing *d*, $1 \le d \le 2n$. Function *set_find*(*d*) is used to find the representative of the set containing *d*, and compress the path.
- Initially *set*[*d*]=-1. That is to say, each criminal constitutes a gang.

Then messages are processed as follow.

- Decide whether *a* and *b* belong to a same gang (*s*[0]=='A').
  - If *a* and *b* don't belong to a same gang (*set_find*(*a*)!=*set_find*(*b*)), and the gang that *a* belongs to and the "opposite" gang for *b* are different (*set_find*(*a*)!=*set_find*(*b+n*)), then we can't decide whether *a* and *b* belong to a same gang; else if the representative of the set containing *a* is the same as the representative of the set containing *b* (*set_find*(*a*)==*set_find*(*b*)), *a* and *b* belong to a same gang; else *a* and *b* belong to different gangs.
- Set up *a* and *b* belong to two different gangs (*s*[0]=='D').
  - If the gang that *a* belongs to isn't the "opposite" gang for *b* (*set_find*(*a*)! =*set_find*(*b+n*), then set up the gang that *a* belongs to is as the "opposite" gang for *b*, and the gang that *b* belongs to is also the "opposite" gang for *a* (*set*[*set_find*(*a*)] = *set_find*(*b+n*); *set*[*set_find*(*b*)] = *set_find*(*a+n*)).