Disjoint Sets
& Minimum Spanning Trees

Adapted from Kenneth Lee's slides (February 2009)

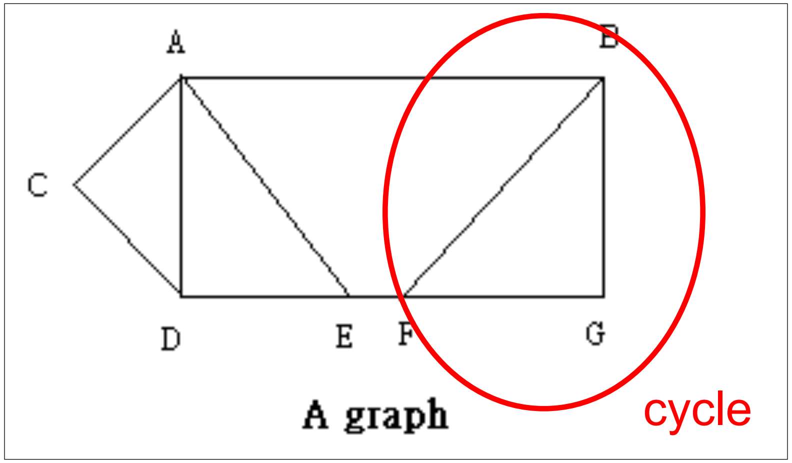
Outline

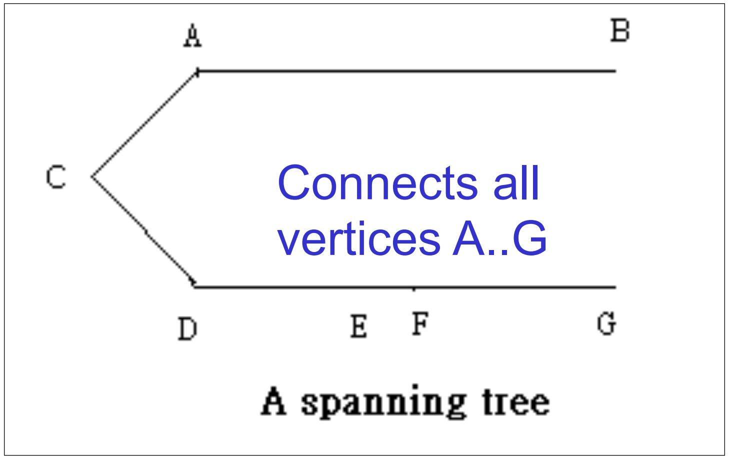
- Definition
- Minimum Spanning Tree algorithms
  - Prim's Algorithm
  - Kruscal's Algorithm
- Implementation of Kruscal's Algorithm
- Disjoint Set
- Exercises

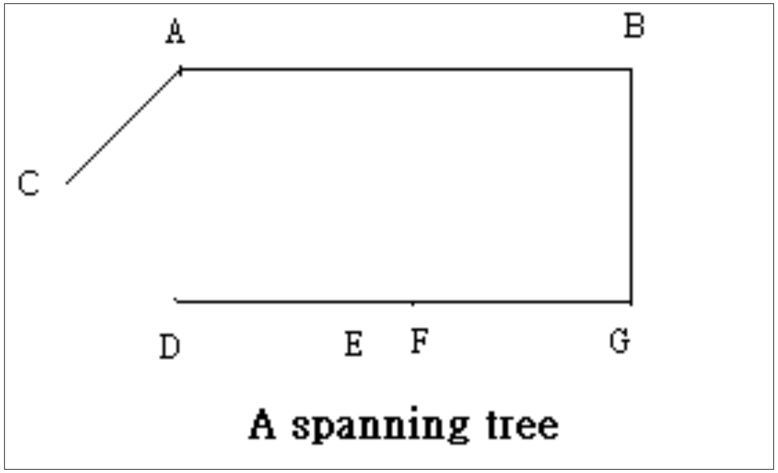
Definition of spanning tree

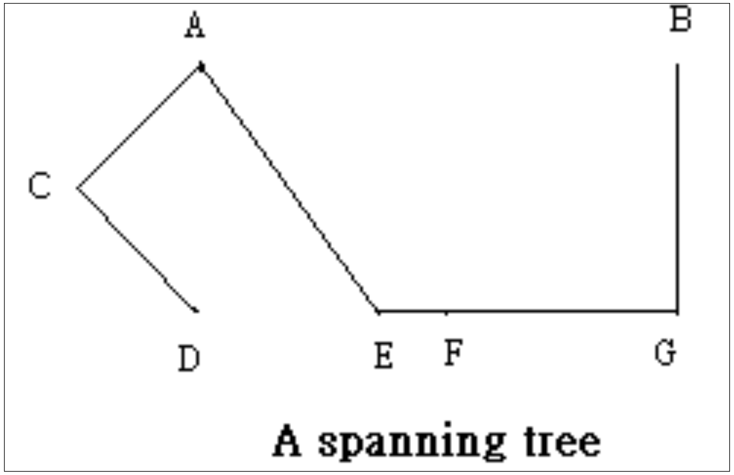
- It is a tree and hence acyclic *(i.e., no cycle)*
- It must connect all the vertices
- It is generated from the given graph
  i.e., spanning tree is a subset of the original (probably cyclic) graph

Spanning Trees









Minimum Spanning Tree

- It is the spanning tree formed from a weighted undirected graph in which the total sum of values on edge has the minimum total weight

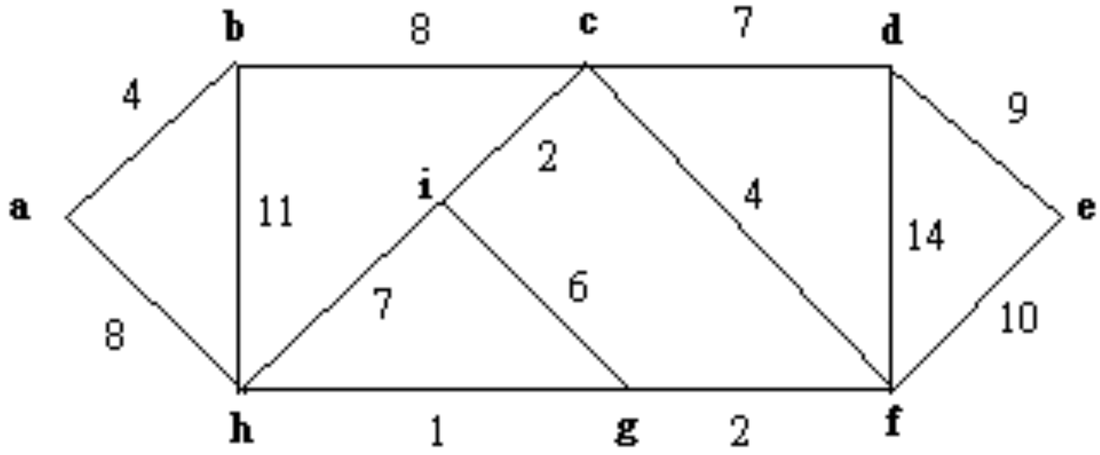
- It can be found by 2 algorithms:
  → Prim’s algorithm
  → Kruskal’s algorithm

Prim’s algorithm

- In each stage, *the set of known vertices is grown by one vertex:*

- *Find an unknown vertex which is nearest to the set of known vertices*
  - *What is the distance from a vertex to a set?*

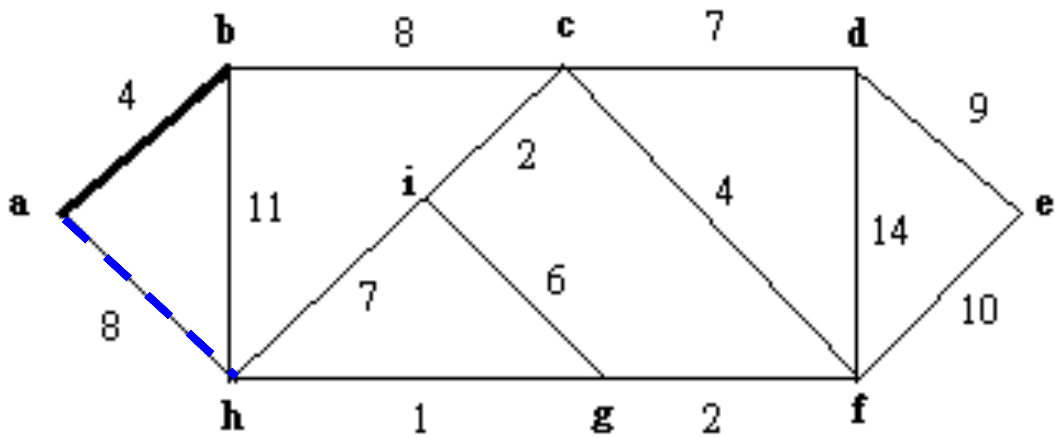
Prim's algorithm



Start Prim's algorithm with any vertex in the graph

For example, we start with {a} here…

Prim’s algorithm



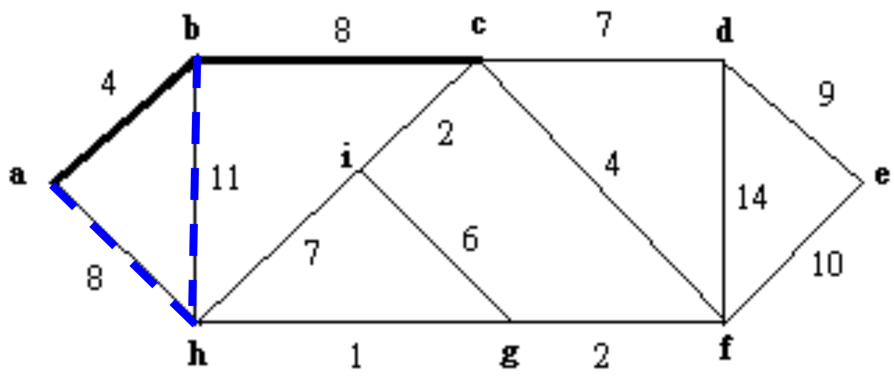
{a} connects to neighbors via two edges:

a-b (cost of 4) and

a-h (cost of 8)

Since edge a-b is having the smallest weight,
Edge a-b is selected and nodes selected in MST = {a,b}

Prim's algorithm



{a,b} connects to neighbors *(nodes outside {a,b})* via three edges:

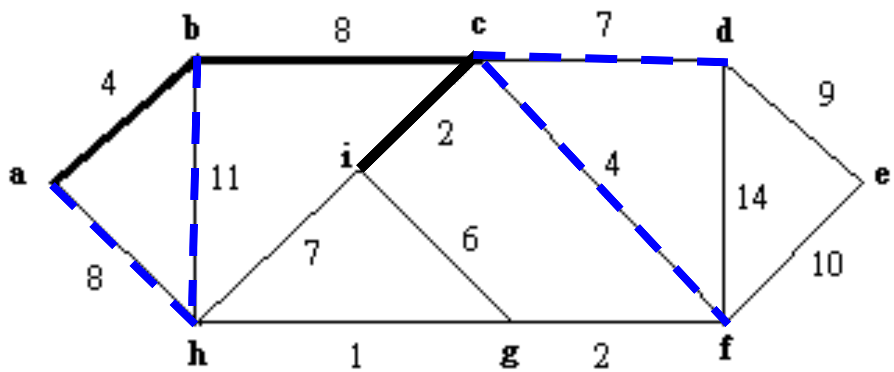
a-h (cost of 8), //old one…

b-h (cost of 11), and

b-c (cost of 8)

Since edge b-c is having the smallest weight *(and c precedes h…)*,
Edge b-c is selected and nodes selected in MST = {a,b,c}

Prim's Algorithm



{a,b,c} connects to neighbors via five edges:

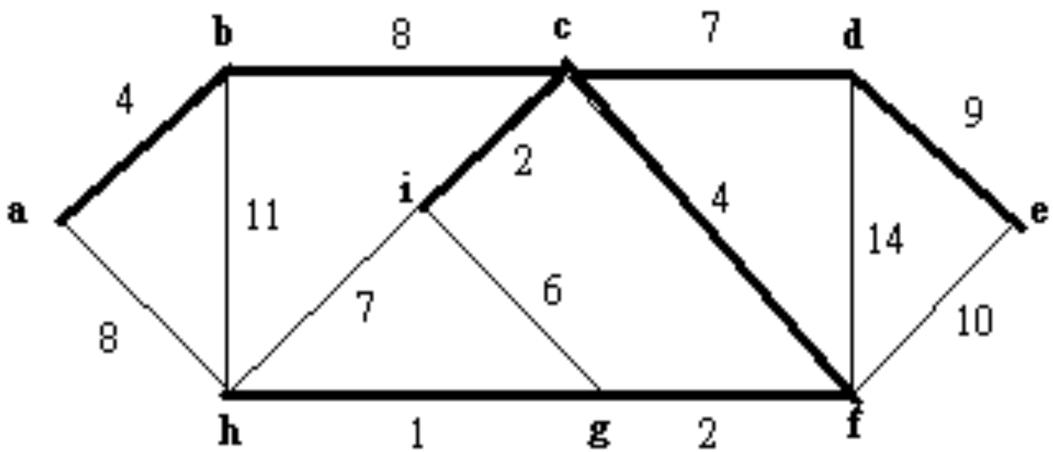
a-h (cost of 4), b-h (cost of 8), //old ones…

c-d (cost of 7), c-f (cost of 4), c-i (cost of 2)

Since edge c-i is having the smallest weight,
Edge c-i is selected and nodes selected in MST = {a,b,c,i}

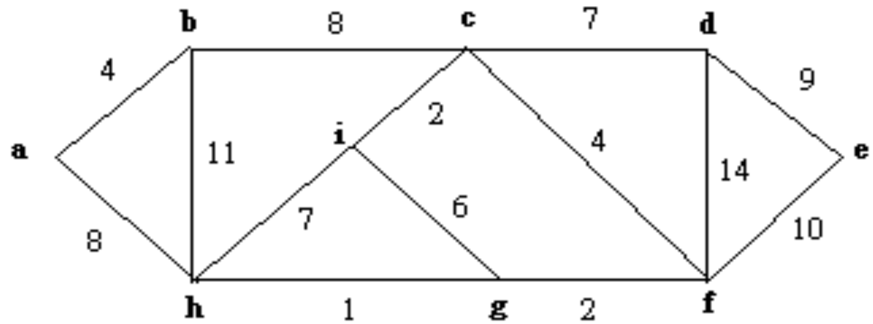
Prim's Algorithm

- Final result:



Observation: each time, a new node is added and CONNECTED

Kruskal's algorithm



From the weighted graph,

1. SORT the edges in non descending order by weights

So we get : hg, ic, gf, ab, cf, ig, cd, hi, ah, bc, de, fe, bh, df

Kruskal’s algorithm

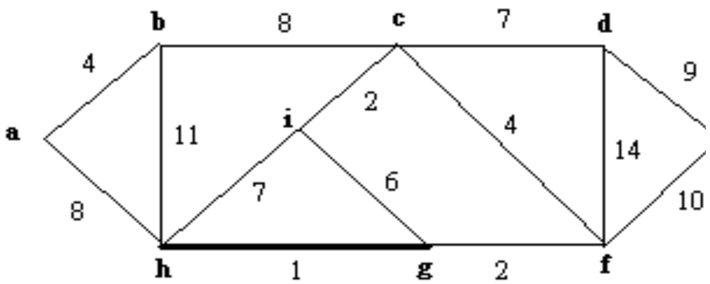
2. Start using the edge with least weight,
   - if the edge does not form a cycle, this edge is accepted;
   - if the edge forms a cycle, this edge is rejected.

     (implementation: how to check for cycle?)

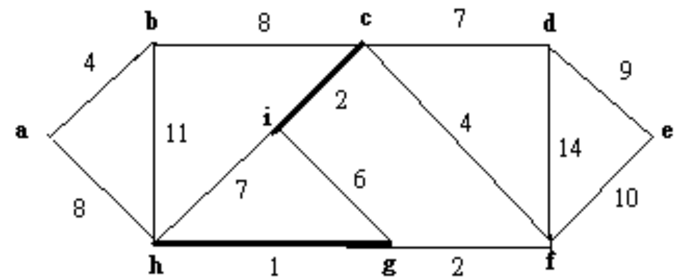
3. Ignore all *examined* edges, repeat step 2 until all the vertices are connected

Kruskal's algorithm

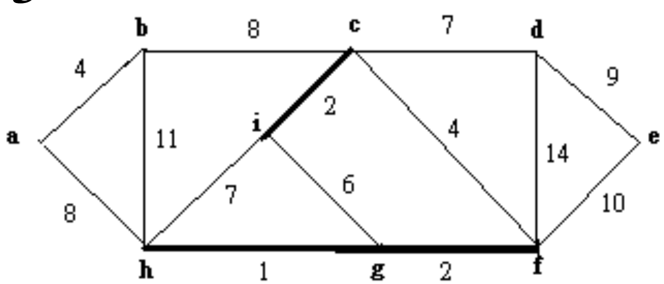
1



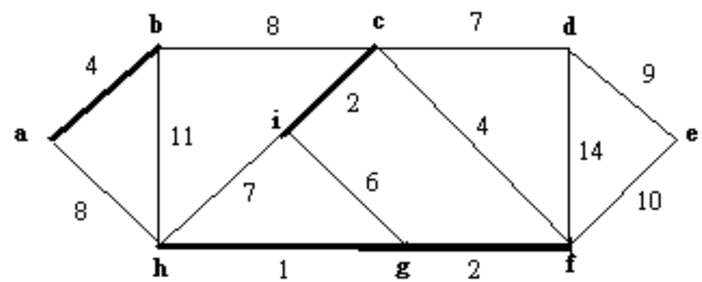
2



3

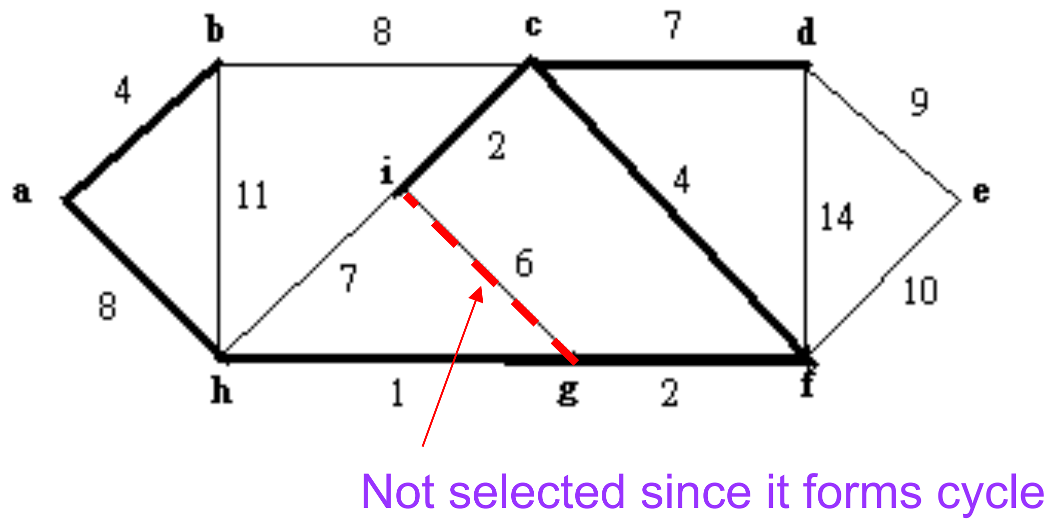


4

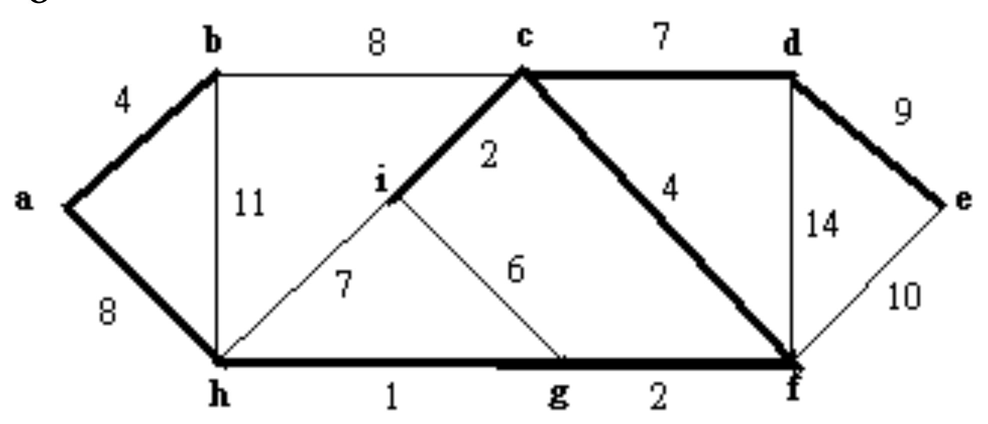


Kruskal’s algorithm

5



6



Minimum Spanning Tree

- As we can see, 2 different algorithms can generate 2 different minimum spanning trees, but with the same total weight

- This implies that a graph can have more than 1 minimum spanning tree (not unique)

Sorting in C…

- qsort() function in <stdlib.h> can be used to sort integers, chars and structures
- Example:

```
int  A[5]={1,3,7,5,2};
qsort(A,5,sizeof(int),CMP);
```

CMP is a function written by programmer used to specify sorting order and comparison criteria

Size of element, in here, sizeof(int) = 4 bytes

Number of elements

Array Name, i.e. pointer/address of first element

Sorting in C…

- Prototype of CMP() function…

int CMP(const void *a, const void *b)

Address/pointer of any one element put Into qsort()

Address/pointer of another element put Into qsort()

CMP() should compare the two elements and see whether a or b should place first

- a should be put before b: return –ve integer
- a should be put after b: return +ve integer
- a and b are identical: return 0

Sorting in C…

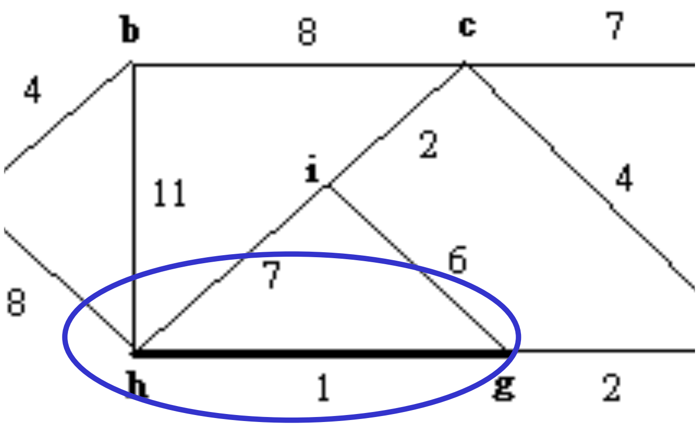
- To sort array A[] in ascending order:

```
int CMP(const void *a, const void *b) {
    int *A = (int*)a;        //cast address to int pointer
    int *B = (int*)b;
    return (*A) – (*B);   //how about descending order ?
}
```

How about sorting edge…

```
struct Edge{
    int From, To, Length;
} All_Edges[1000];
                    :
const int CMP(const void *a, const void *b) {
    Edge *A = (Edge*)a;
    Edge *B = (Edge*)b;
    return A->Length – B->Length;
}
                    :
qsort(All_Edges,nEdges,sizeof(Edge),CMP);
```

Coding Kruskal’s algorithm

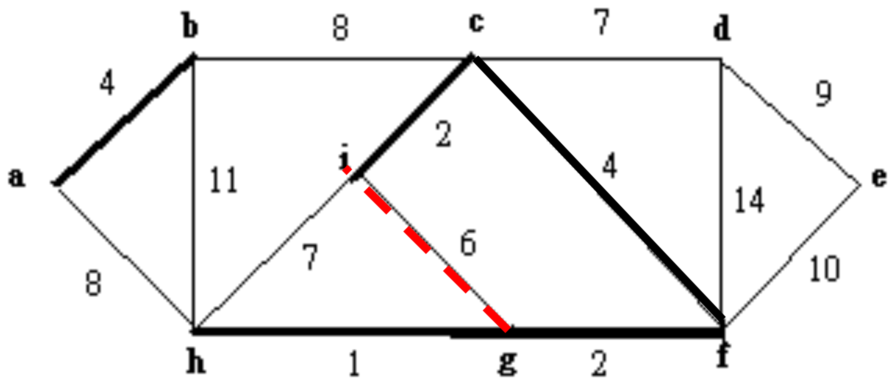


- Edge **h-g** is having min weight
- Since h and g are not already (directly/indirectly) connected, so we accept this edge
- Make h and g belong to the same set….. *How?*

a	b	c	d	e	f	g	h	i
a	b	c	d	e	f	g	h	i

a	b	c	d	e	f	g	h	i
a	b	c	d	e	f	h	h	i

Kruskal's algorithm



- Assume we have added edges hg, ic, gf, ab, cf

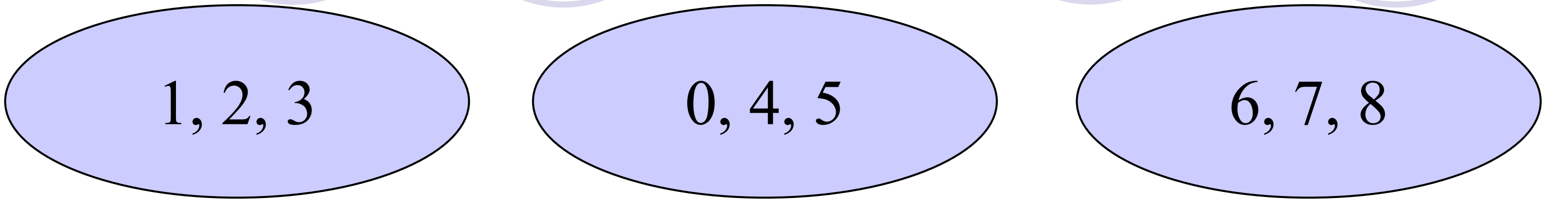
a	b	c	d	e	f	g	h	i
a	b	h	d	e	h	h	h	h

- Now, we check edge ig
- Vertex i and vertex g are already in the same Set h, that means it is connected already, hence we can reject this edge

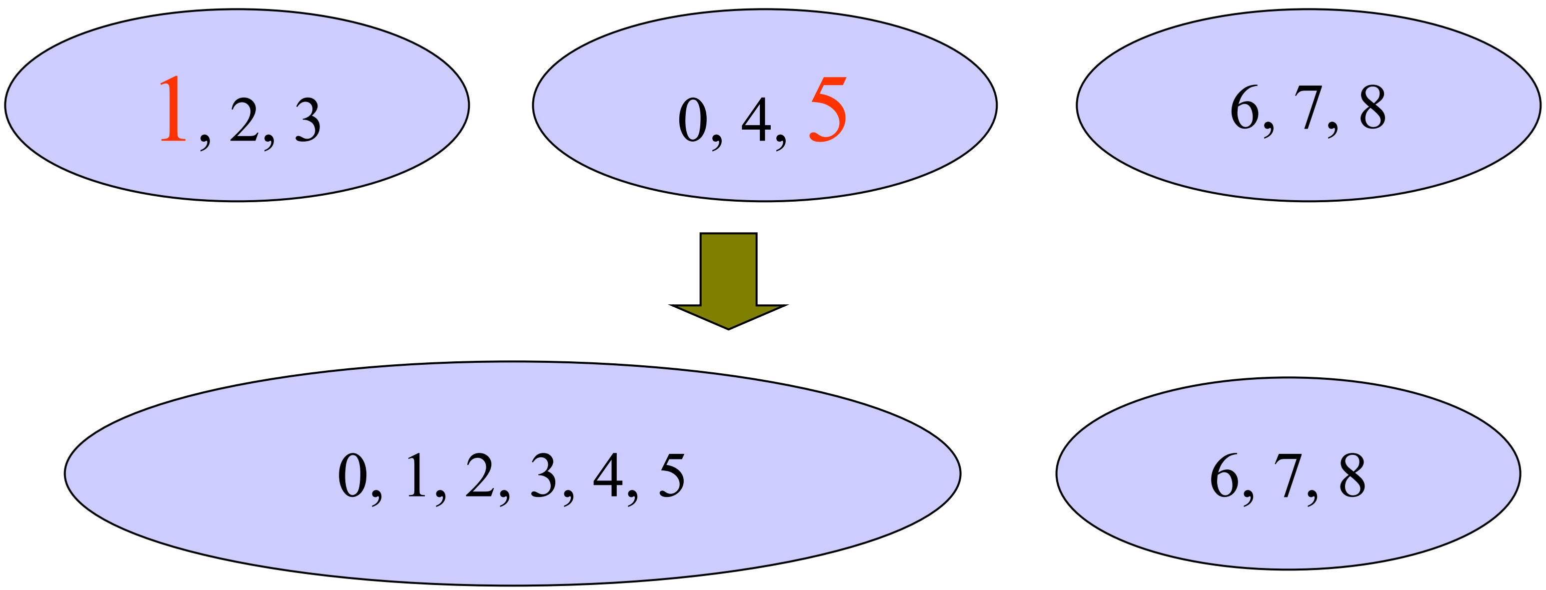
Further More on Set

- The classical operation of disjoint set are as follow:
- "Are the items '1' and '2' in the same set?" *(query)*
- "Union the set containing '1' with the set containing '2'" *(merge)*
- Note: Each item must belong to exactly one set.

Disjoint Set Operation

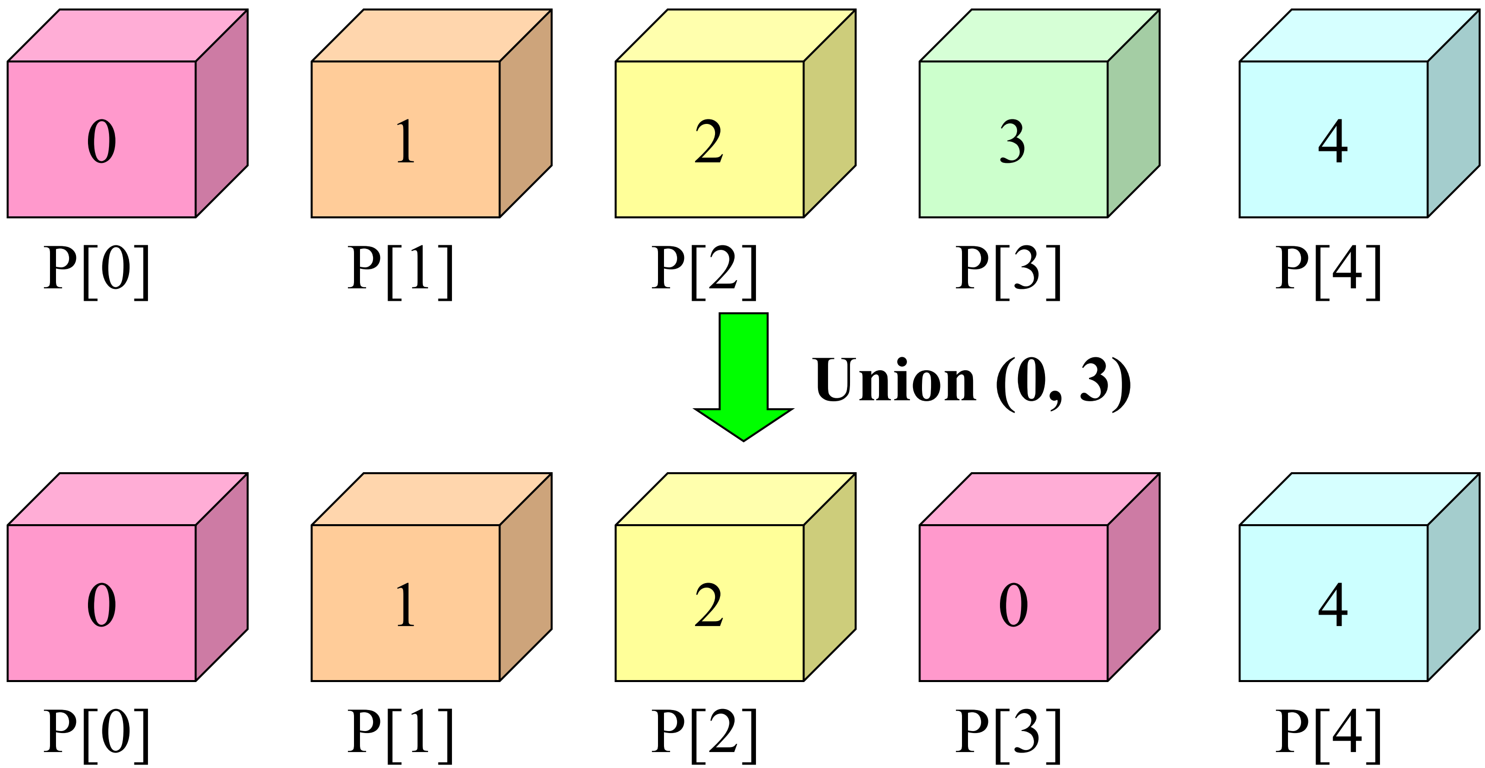


Union the set containing '1' with the set containing '5'

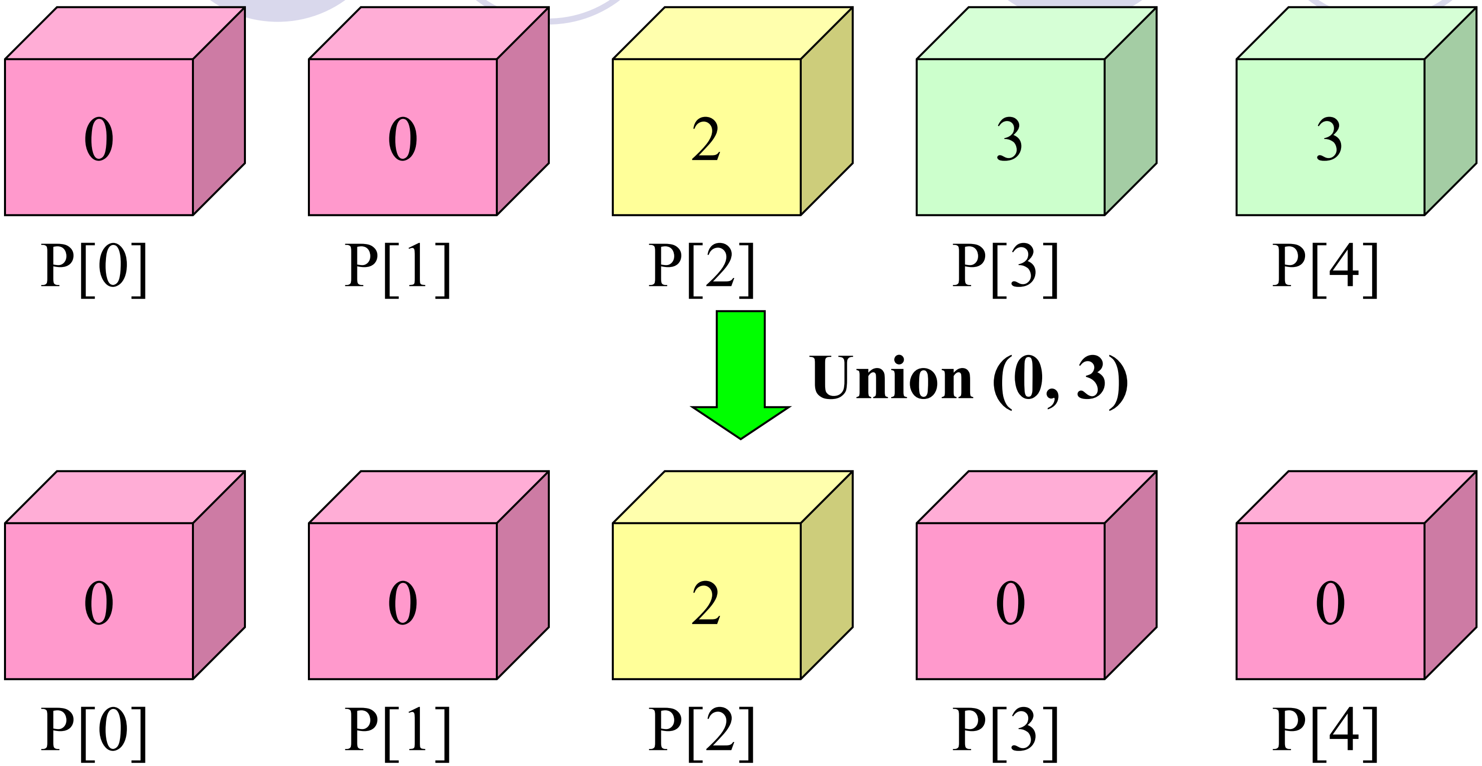


How to implement?

- Simple Structure – First, assign each element to a set containing its own. *(Implement with an array)*



How to implement?



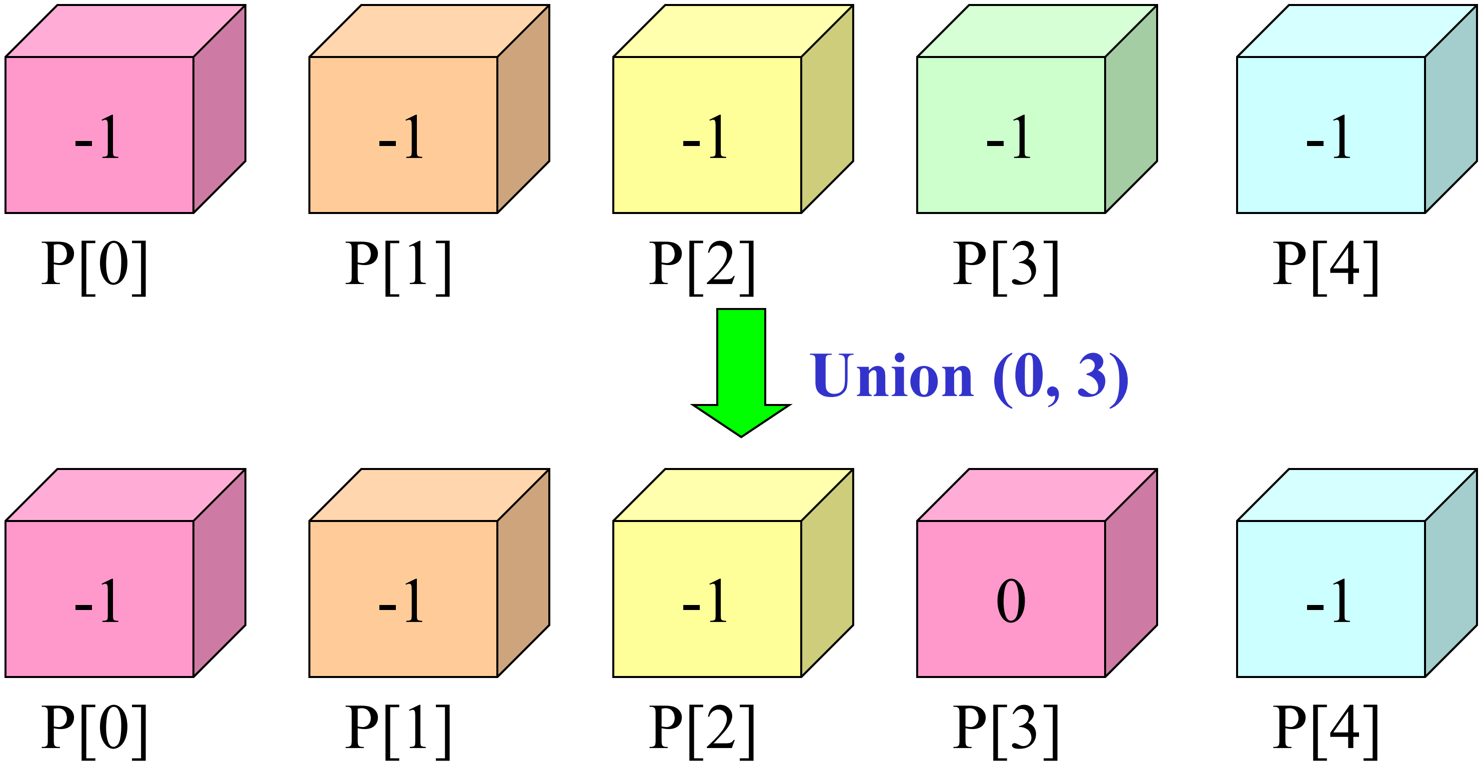
Fast *Query*….but how about *Merge* operation?

How to implement?

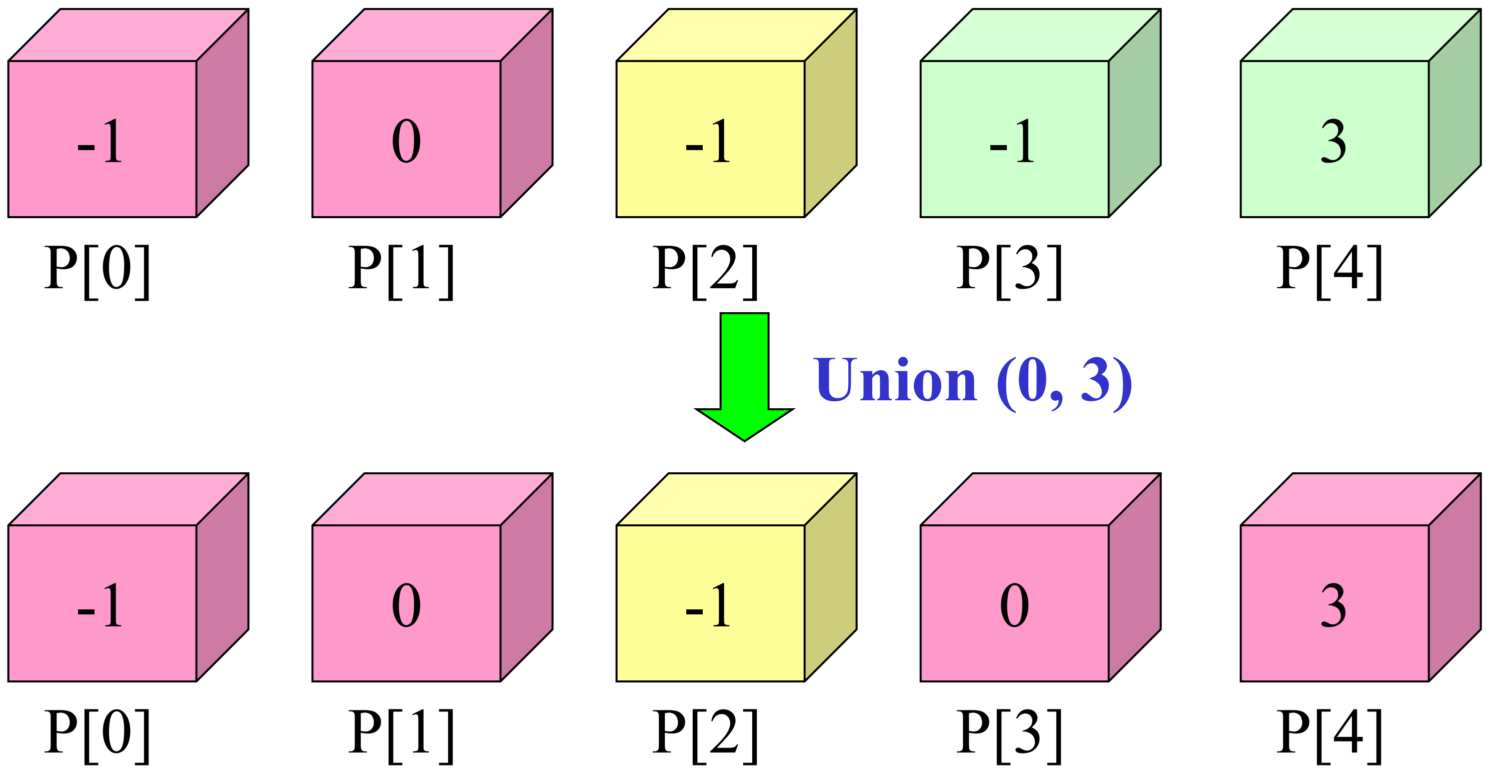
```
// Union the sets of two elements
void Union (int element1, int element2) {
      UnionSet(Find(element1), Find(element2));
}

// Union 2 sets... What if we merge 1000 items?
void UnionSet (int set1, int set2) {
      for (int j=0; j<SIZE; j++)
            if (A[j] == set2)
                  A[j] = set1;
}

// return the Set ID
int Find (int element)  {
      return A[element];
}
```

How to implement

- **Forest Method**
- Initialize : Assign Each element the value of –1, representing the index is the set ID



How to implement?



How to implement?

```
void Union (int element1, int element2) {
      UnionSet(Find(element1), Find(element2));
}

void UnionSet (int set1, int set2) {
      A[set2] = set1;
}


int Find (int element)  {
      if (A[element] == -1)
            return element;
      else
            return Find(A[element]);
}
```

Find by Recursion

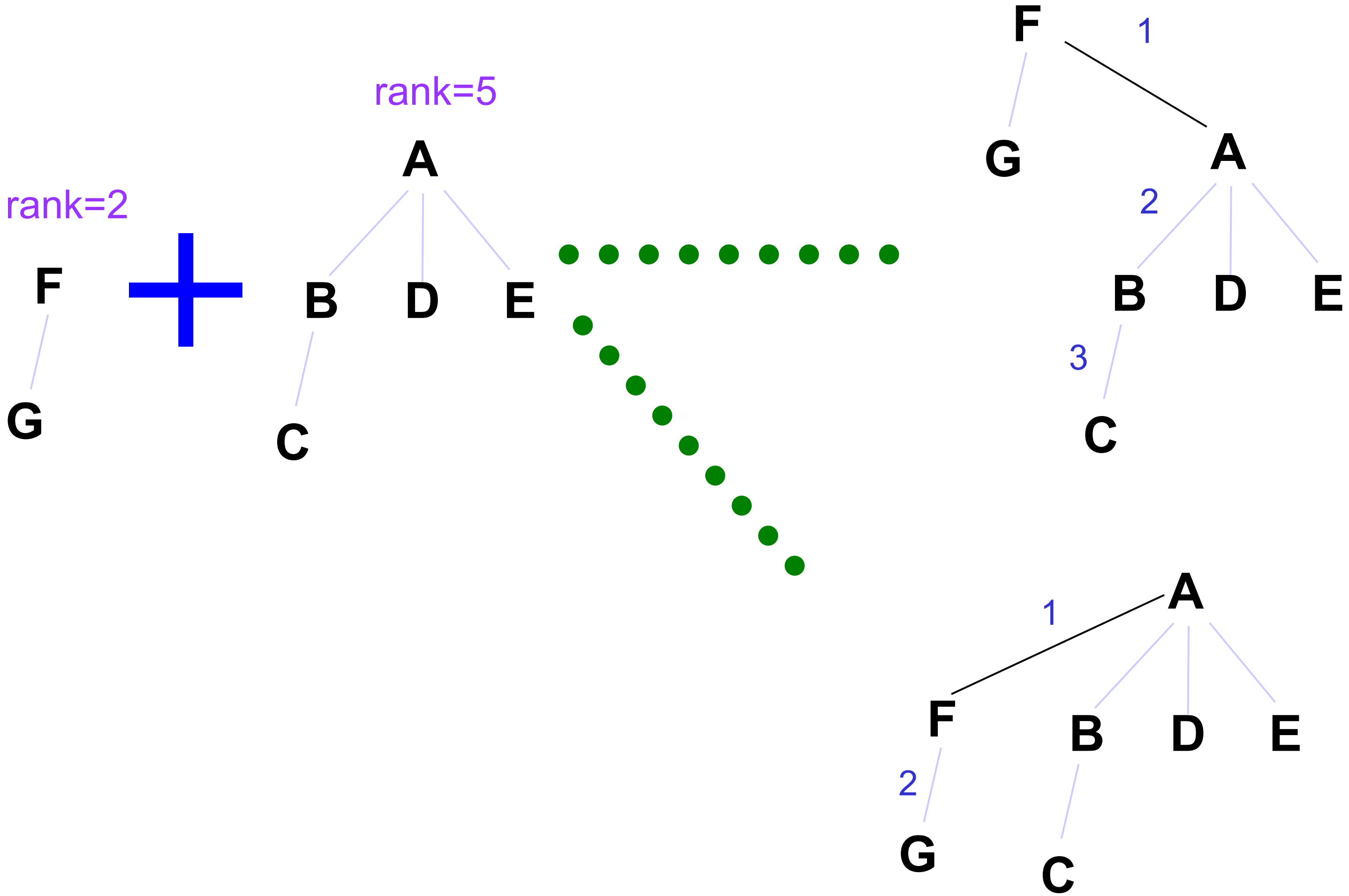
Fast *Merge* operation ….but …

How to implement : Enhancement

- Worse case:
  Disjoint Set Forest -> Linked-list representation
  - Time waste in Path-Finding!
- Two heuristics to improve it
  - Union by size
  - Path Compression
- Union by Size:
  - Idea: let root with fewer nodes point to the root with more nodes
  - Implement: rank = upper bound on the height of tree *(i.e. worse case: skewed tree)*

How to implement : Enhancement

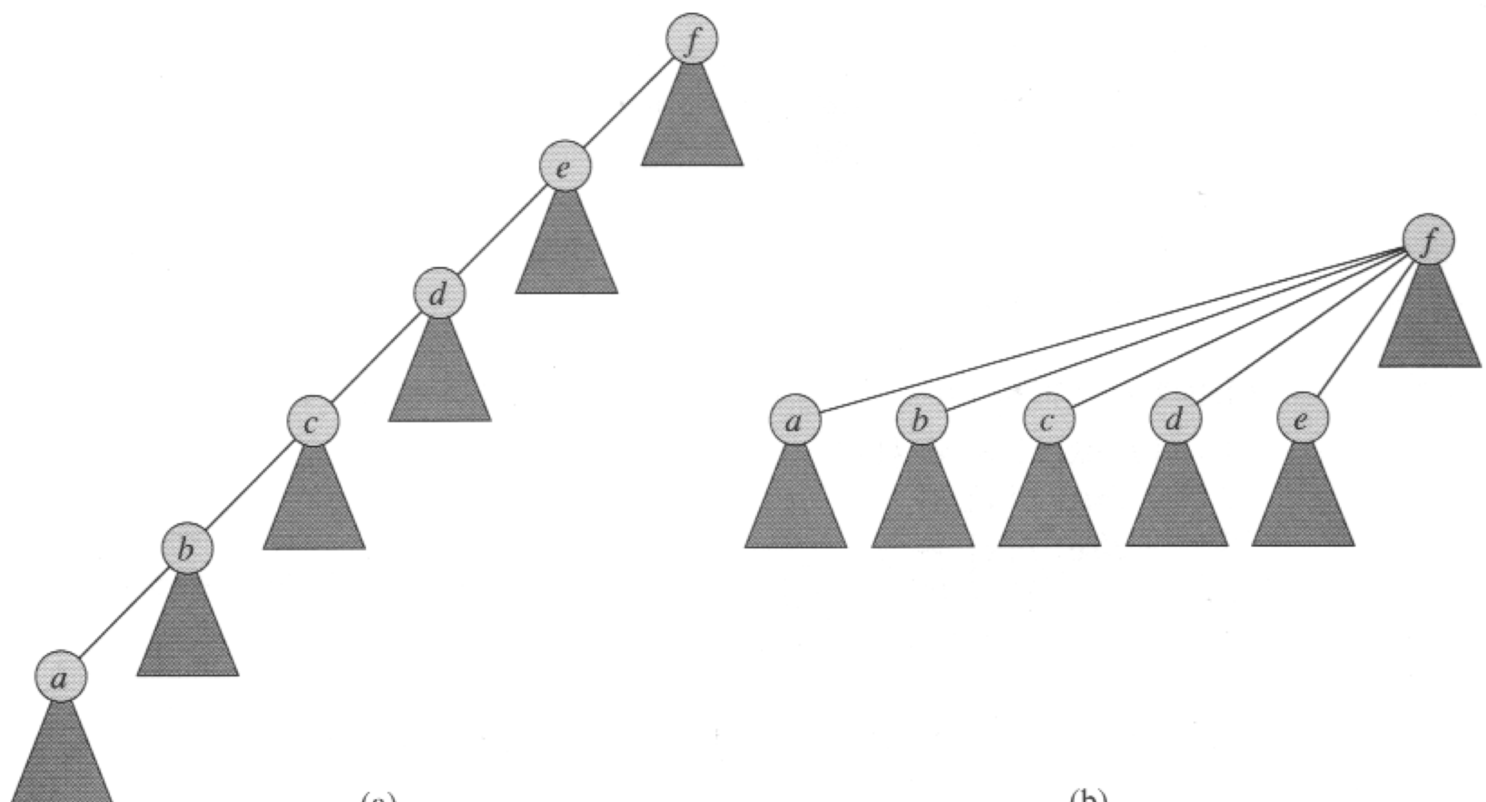
Merge **F**’s set with **A**’s set:



How to implement?

```
void Union (int element1, int element2) {
        int root1=Find(element1);
        int root2=Find(element2);
        if  (A[root1] < A[root2]) //root1 has more member
                UnionSet(root1, root2);
        else
                UnionSet(root2, root1);
}
void UnionSet (int set1, int set2) {
        A[set1] += A[set2]; // A[ root of set ] is negative, and its
        A[set2] = set1;     // magnitude is the num of members
}
int Find (int element)  {
        if (A[element] < 0)
                return element;
        else
                return Find(A[element]);
}
```

How to implement : Enhancement



(a) (b)

Path Compression

```
int Find (int element)  {
        if (A[element] < 0)
                return element;
        else
                return A[element] = Find(A[element]);

}
```

Path Compression!

Whenever Find() is performed, all items along the path update its parent to the topmost root

Next query (Find()) will be faster…

Exercises

- UVA Online judge:
  - 10147 Highways
  http://acm.uva.es/p/v101/10147.html
  - 534 Frogger
  http://acm.uva.es/p/v5/534.html
  - 10369 Arctic Network
  http://acm.uva.es/p/v103/10369.html