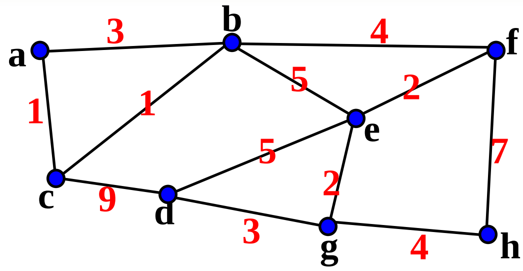
Unit 8
Minimum Spanning Trees

T.H. Cormen et al., "Introduction to Algorithms", 3rd ed., Chapter 23

R. Sedgewick, "Algorithms in C++", Chapter 30.

Weighted Graphs

***G*=(*V*,*E*)**



- Many graph problems are considered on weighted graphs.
- Here, we assume that all of the weights are on edges and are positive.

Minimum Spanning Trees (MST)

- Find the lowest-cost way to connect all of the points (the cost is the sum of weights of the selected edges).
- The solution must be a tree. (Why ?)
- A spanning tree : a subgraph that is a tree and connect all of the points.
- A graph may contains exponential number of spanning trees.

  (e.g. # spanning trees of a complete graph = n^{n-2}.)

A High-Level Greedy Algorithm for MST

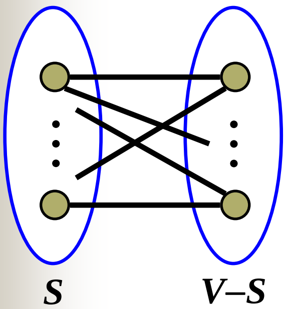
```
A = ∅;
while(T=(V,A) is not a spanning tree of G) {
  select a safe edge for A ;
}
```

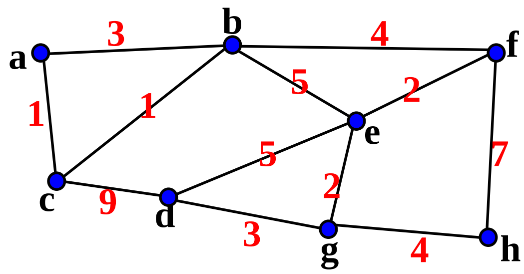
- The algorithm grows a MST one edge at a time and maintains that A is always a subset of some MST.
- An edge is ***safe*** if it can be safely added to without destroying the invariant.
- How to check that T is a spanning tree of G ?
- How to select a "safe edge" edge ?

MST 基本引理

Let Δ(*S*, *V*–*S*) = {*uv* | *u* ∈ *S* & *v* ∈ *V*–*S* }, *S* ≠ *V*, ∅.
If *xy* is an edge with minimum weight in Δ(*S*, *V*–*S*)
then *xy* is contained in some MST.

A *cut*



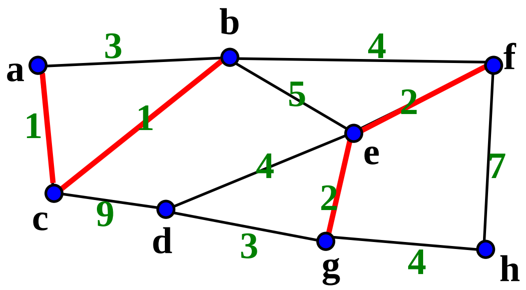


Kruskal's Algorithm (pseudo code 1)

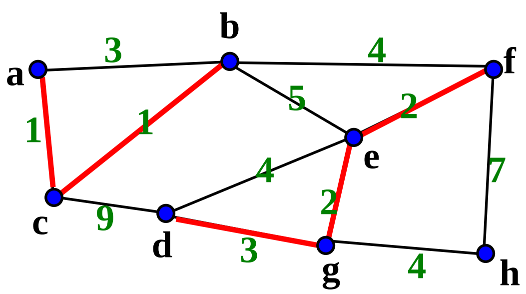
```
A= ∅;
for( each edge in order by nondecreasing weight )
  if( adding the edge to A doesn't create a cycle ){
       add it to A;
       if( |A| == n–1 ) break;
  }
```

☛ How to check that adding an edge does not create a cycle?

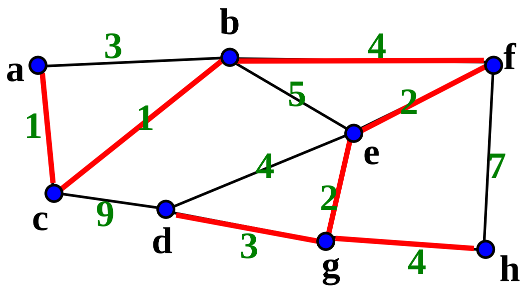
Kruskal's Algorithm (例 1/3)



Kruskal's Algorithm (例 2/3)



Kruskal's Algorithm (例 3/3)



MST cost = 17

Kruskal's Algorithm (pseudo code 2)

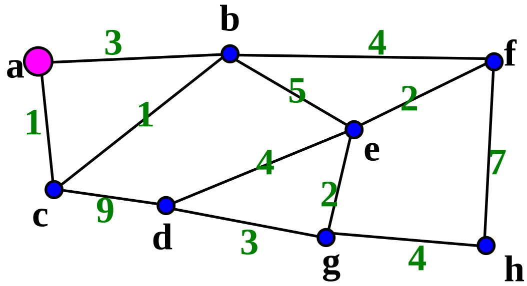
```
A = ∅;   initial(n); // for each node x construct a set {x}
for( each edge xy in order by nondecreasing weight)
  if ( ! find(x, y) ) {
     union(x, y);
     add xy to A;
     if( |A| == n–1 ) break;
  }
```

find(x, y) = true iff. x and y are in the same set
union(x, y): unite the two sets that contain x and y, respectively.

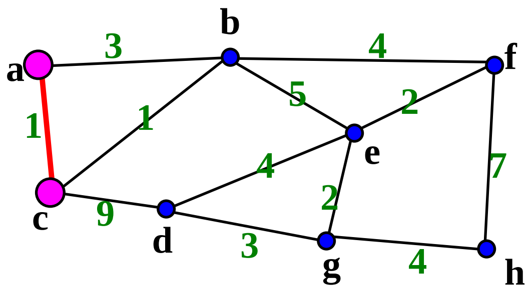
Prim's Algorithm (pseudo code 1)

ALGORITHM *Prim*(*G*)
// Input: A weighted connected graph $G=(V,E)$
// Output: A MST $T=(V, A)$
  $V_T \leftarrow \{ v_0 \}$ // Any vertex will do;
  $A \leftarrow \varnothing$;
  for $i \leftarrow 1$ to $|V|-1$ do
    find an edge $xy \in \Delta(V_T, V-V_T)$ **s.t. its weight is minimized among all edges in** $\Delta(V_T, V-V_T)$**;**
    $V_T \leftarrow V_T \cup \{ y \}$; $A \leftarrow A \cup \{ xy \}$;

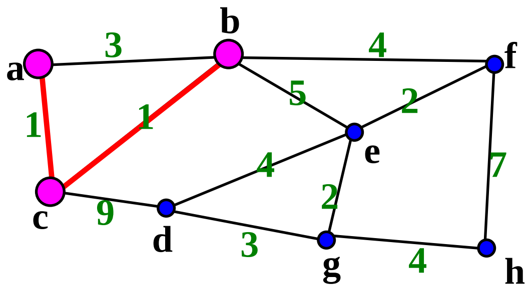
Prim's Algorithm (例 1/8)



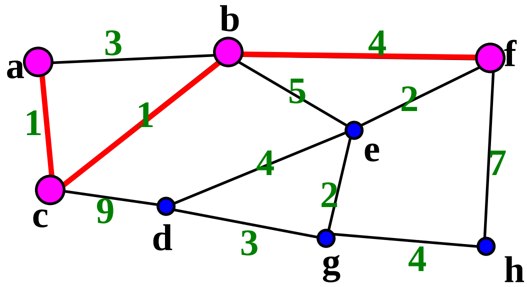
Prim’s Algorithm (例 2/8)



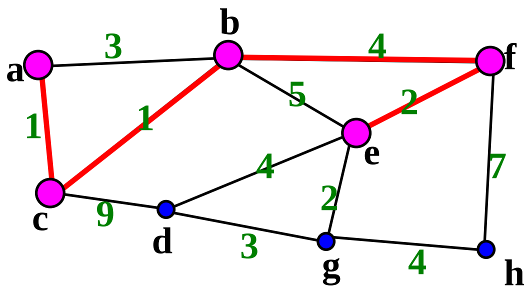
Prim’s Algorithm (例 3/8)



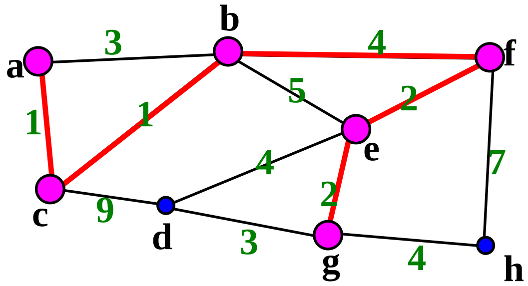
Prim’s Algorithm (例 4/8)



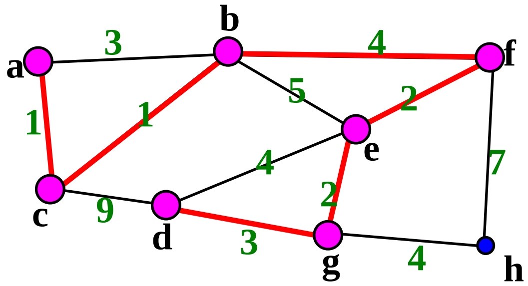
Prim’s Algorithm (例 5/8)



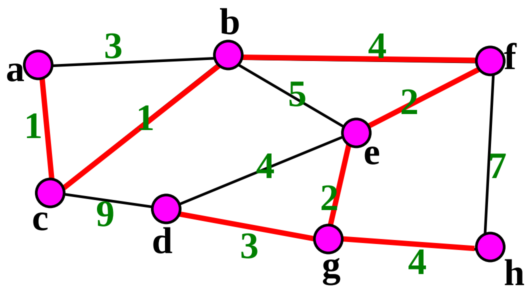
Prim's Algorithm (例 6/8)



Prim's Algorithm (例 7/8)



Prim’s Algorithm (例 8/8)



MST cost = 17

Prim’s Algorithm (pseudo code 2)

Built a priority queue Q for V with key[u] = ∞ $\forall$ $u \in V$;
key[v_0] = 0; π[v_0] = Nil; // Any vertex will do
While ($Q \neq \emptyset$) {
  u = Extract-Min(Q);
  for(each $v \in$ Adj(u))
    if ($v \in Q$ && $w(u, v) <$ key[v]) {
      π[v] = u;
      key[v] = $w(u, v)$;
      Change-Priority(Q, v, key[v]);
    }
}

Minimum Spanning Tree (分析)

- Let $n = |V(G)|$, $m = |E(G)|$.
- Execution time of Kruskal's algorithm: (use union-find operations, or disjoint-set unions)
  $O(m \log m) = O(m \log n)$
- Running time of Prim's algorithm:
  - adjacency lists + (binary or ordinary) heap:
    $O((m+n) \log n) = O(m \log n)$
  - adjacency matrix + unsorted list: $O(n^2)$
  - adjacency lists + Fibonacci heap:
    $O(n \log n + m)$

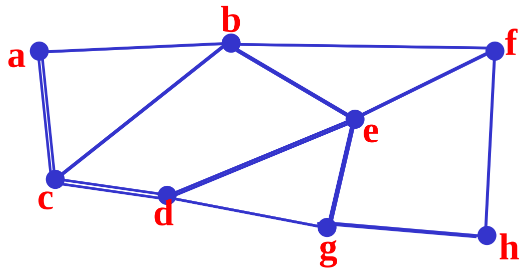
附錄

Union-Find Operations, or Disjoint-Set Unions

(Based on Chapter 30 of R. Sedgewick, "Algorithms in C++", 1992.)

Dynamic Connectivity Testing

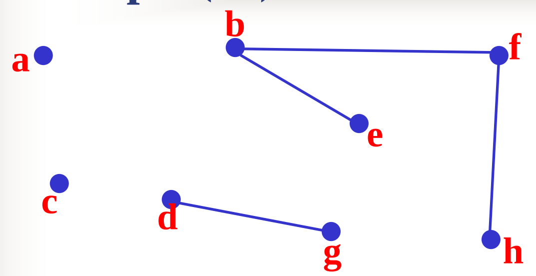
- Initially, a graph with empty edge set is given & edges are added one by one, intermixed with queries about whether two given vertices are in the same c.c. of the current graph.



Disjoint-Set Operations

- $S_1, S_2, \ldots, S_k$ are disjoint sets.
- Each set is identified by a ***representative***, which is some member of the set (sometimes, it doesn't matter which member is used).
- Possible operations:
  - **union(x , y)** : unite the two sets containing **x** and **y**, and "destroy" the two sets.
  - **find(x , y)** : are **x** and **y** in the same set?

An Example (1/2)



{a} , {b} , {c} , {d} , {e} , {f} , {g} , {h}

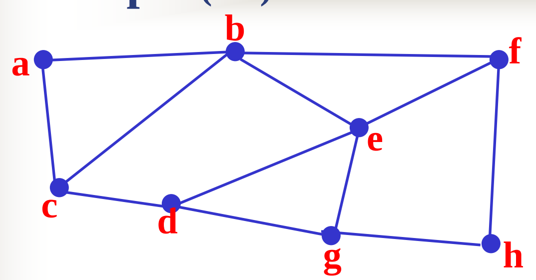
union(b,f): {a} , {b, f} , {c} , {d} , {e}, {g} , {h}

union(f,h): {a} , {b, f, h} , {c} , {d} , {e}, {g}

union(b,e): {a} , {b, e, f, h} , {c} , {d} , {g}

union(d,g): {a} , {b, e, f, h} , {c} , {d, g}

An Example (2/2)



union(d,g): {a} , {b, e, f, h} , {c} , {d, g}; find(f,d)=F

union(a,c): {a, c} , {b, e, f, h} , {d, g}; find(f,d)=F

union(g,e): {a, c} , {b, e, f, h, d, g}; find(f,d)=T

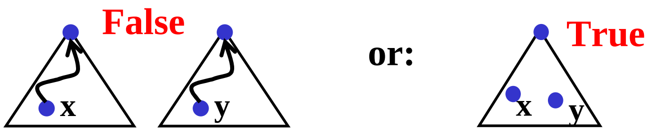
union(e,f): {a, c} , {b, e, f, h, d, g}

union(c,d): {a, c, b, e, f, h, d, g}

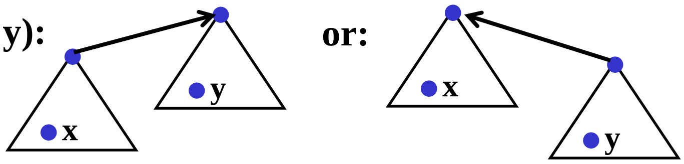
Implementations

- Each set is stored as a rooted tree.
- Each set is identified by a ***representative***, which is the element stored on the root.

find(x, y):

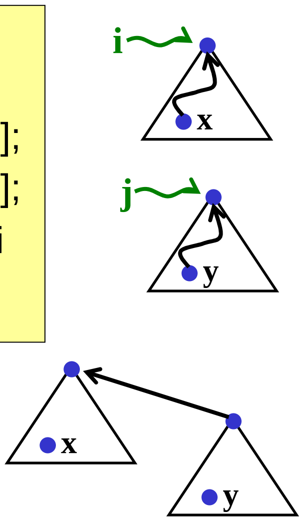


union(x, y):



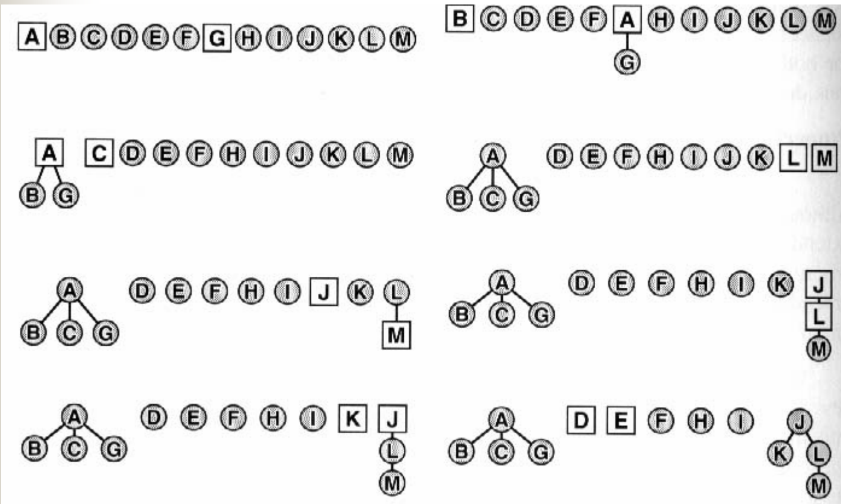
A Straightforward Procedure

```
find( x, y, doit)
{
  int i = x, j = y;
    while (dad[ i ] > 0) i = dad[ i ];
    while (dad[ j ] > 0) j = dad[ j ];
    if (doit && (i != j) ) dad[ j ] = i
    return ( i == j );
}
```

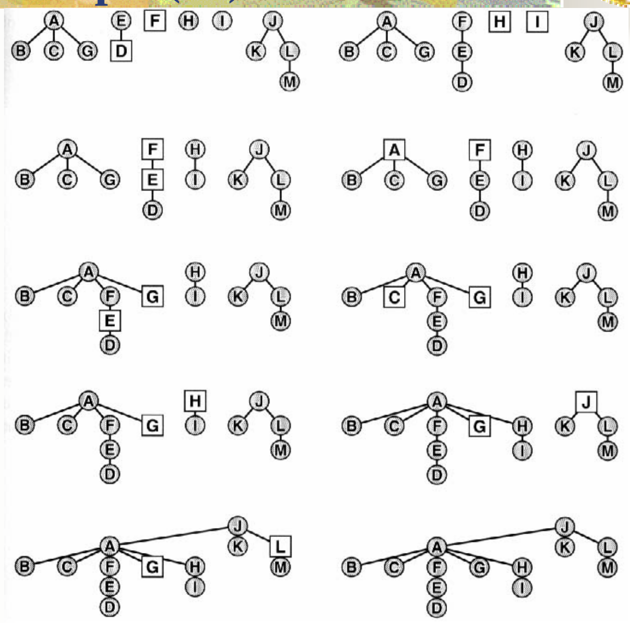


doit $\neq$ 0 $\Rightarrow$ execute find(x, y) + union(x, y)

An Example (1/2)



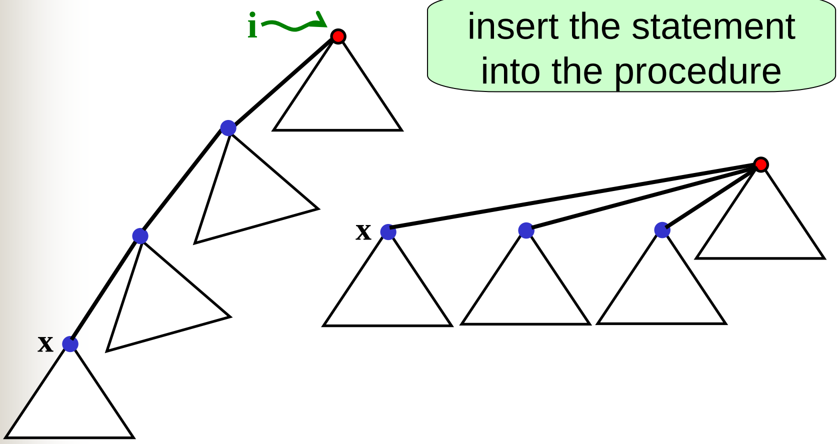
An Example (2/2)



改進 1: Path Compression:

```
while (dad[ x ] > 0)
    { t = x; x = dad[x]; dad[t] = i;  }
```

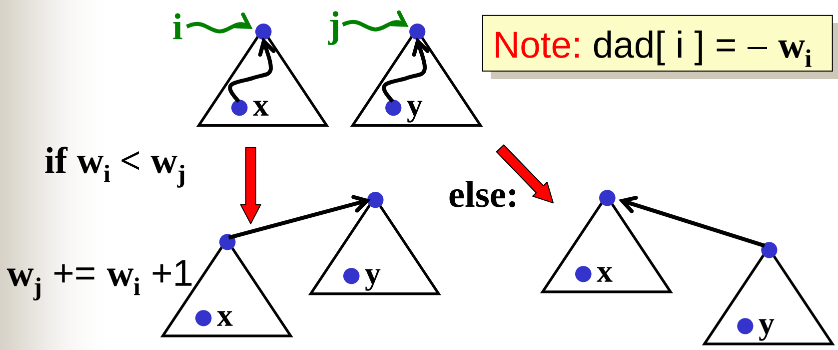
insert the statement into the procedure



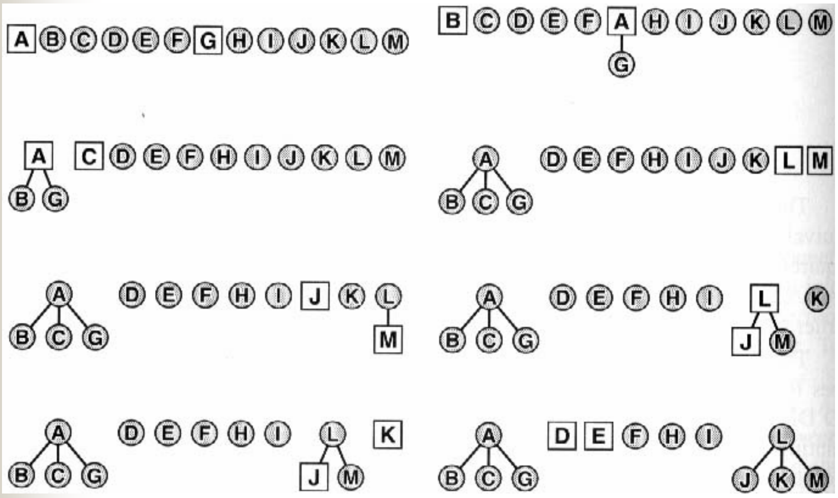
改進 2: Weight or Height Balancing

```
/* weight (#decedents) balancing   */
   if (dad[ j ] < dad[ i ])
      { dad[ j ] += dad[ i ] - 1; dad[ i ] = j;  }
   else
      { dad[ i ] += dad[ j ] - 1; dad[ j ] = i;  }
```

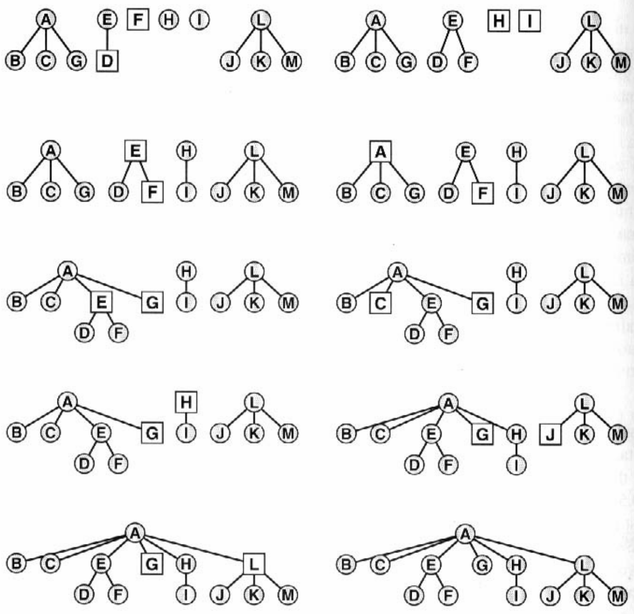
Note: dad[i] = – w_i



An Example (改進版)



An Example (改進版，續1)

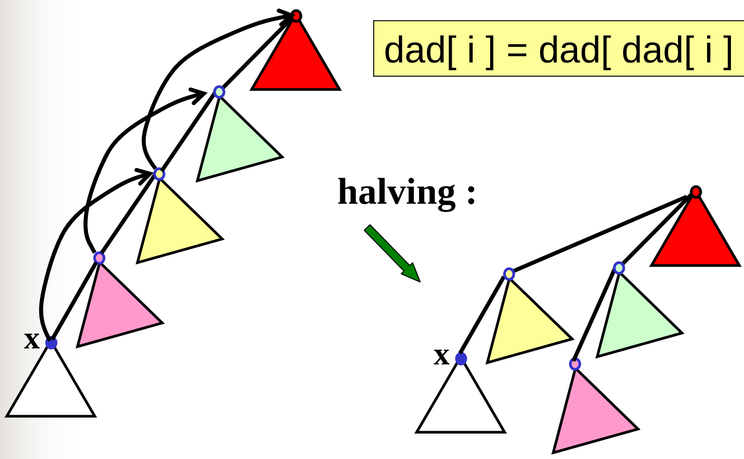


Complexity (Union-Find)

- If $O(m)$ operations are executed, then :
  - Execution time : $O(m\ \alpha(m))$ $(n = |V|, m = |E|)$
  - where $\alpha(m) < 4$ if $m < 2^{2^{16}} = 2^{65536}$
  - Space : $O(n)$
- In some situations, e.g. processing huge graphs, this is an **advantage** over DFS or BFS for finding connected components.

其他改進: Halving

dad[i] = dad[dad[i]]



A kind of ***one-pass*** method.

其他改進: Splitting (another one-pass method)