Lecture 9: Kruskal's MST Algorithm : Disjoint Set Union-Find

A **disjoint set Union-Find** date structure supports three operation on x, and y:

1. **Create-Set(x)** Create a set containing a single item x.

2. **Find-Set(x)** Find the set that contains x

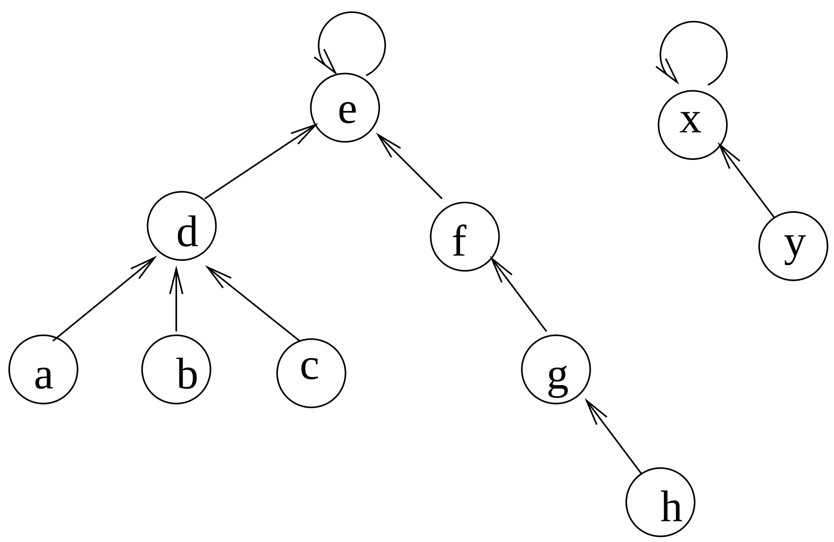
3. **Union(x, y)** Merge the set containing x, and another set containing y to a single set. After this operation, we have Find-Set(x)=Find-Set(y).

Up-tree implementation

Basic ideas:

- Every item is in a tree. The root of the tree is the represtative item of all items in that tree i.e., the root of the tree represents the whole items.

- In this up-tree implementation, every node (except the root) has a pointer pointing to its parent. The root element has a pointer pointing to itself.



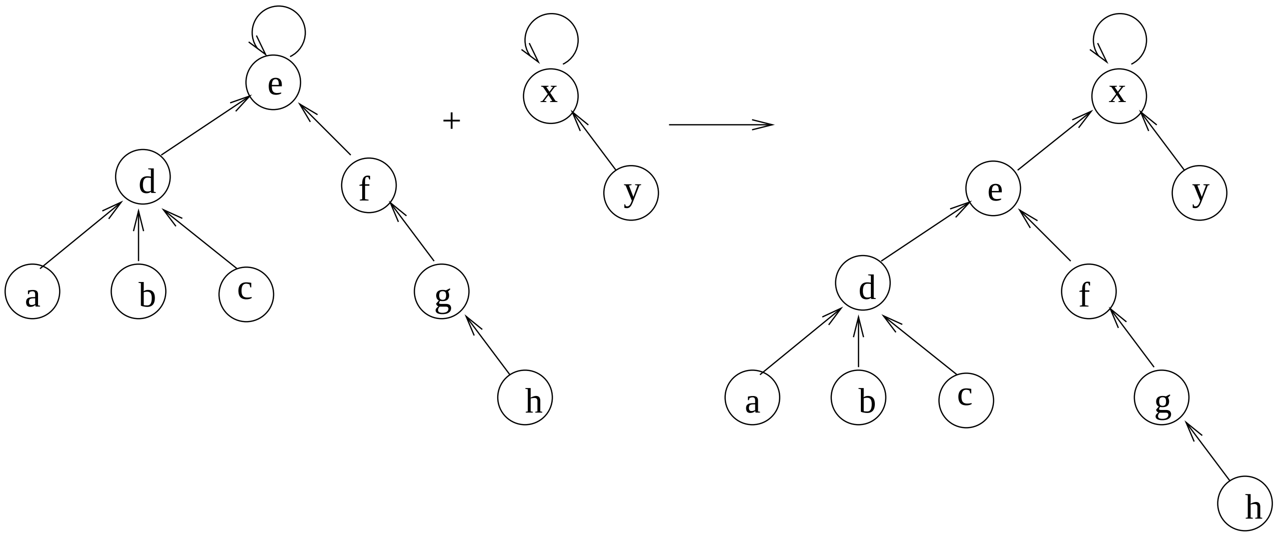
- The operation of Create-Set(x) is easy.

```
Create-Set(x)
 x->parent=x;
```

- The operation of Find-Set(x) is easy, we simple trace the parent point until we hit the root, then return the root element.

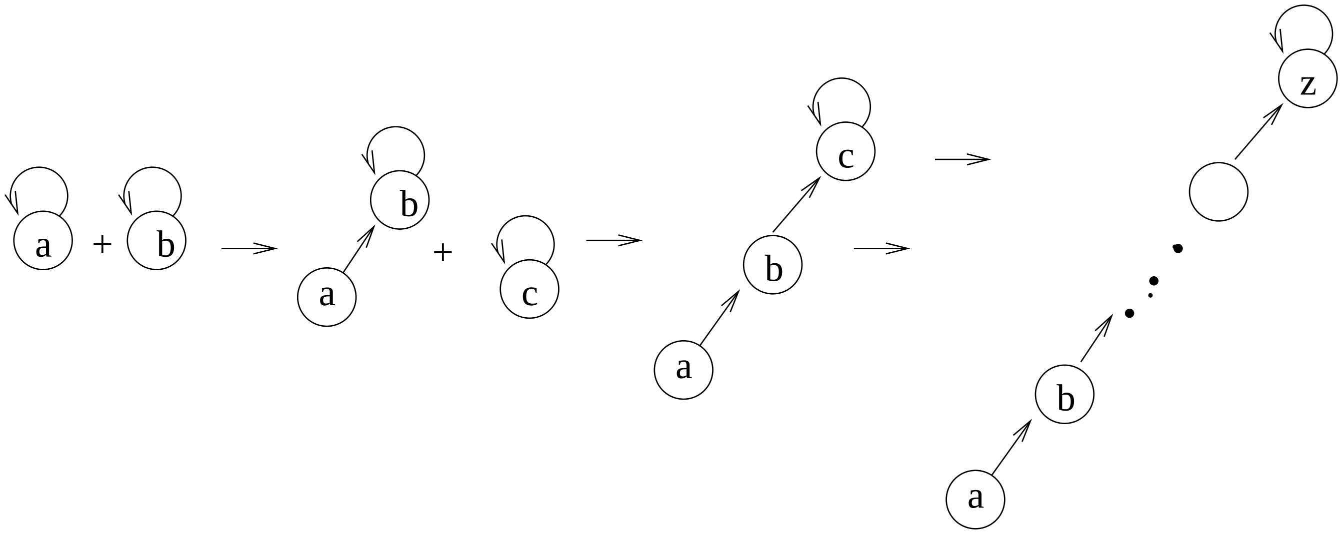
```
Find-Set(x)
 while (x!= x->parent)
  x = x->parent;
 return x;
```

- The operation of Union(x, y) is a little bit tricky. We can just simply putting the parent pointer of the representation of x pointing to the representation of y.



Is it a good idea?

The problem is that, it may become a linked-list at the end! Hence it is not efficient. Can we do better?



A simple trick: when we union two trees together, we always make the root of *taller* tree the parent of shorter tree. This trick is called **Union by height**.

Up-tree implementation : Union by height

The root of every tree also holds the height of the tree.

In case two trees has the same height, we choose the root of the first tree point to the root of second. And the tree height is increased by 1.

```
Union(x, y)
 a=Find-Set(x); b=Find-Set(y);
 if (a.height <= b.height)
  if (a.height == b.height) b.height++;
  a->parent=b;
 else b->parent=a;
```

Up-tree implementation : Union by height

Lemma For any root x of a tree, let $size(x)$ be the number of nodes, and $h(x)$ be the height of the tree. We have $size(x) \geq 2^{h(x)}$.

Proof (by induction)

1. At beginning, $h(x) = 0$, and $size(x) = 1$. We have $1 \geq 2^0 = 1$.

2. Suppose the assumption is tree for any x, and y before Union(x, y) operation. Let the size and height of the resulting tree be $size(x^{'})$, and $h(x^{'})$.

   - $h(x) < h(y)$, we have

$$
\begin{aligned}
size(x^{'}) &= size(x) + size(y) \\
&\geq 2^{h(x)} + 2^{h(y)} \\
&\geq 2^{h(y)} \\
&= 2^{h(x^{'})}.
\end{aligned}
$$

- $h(x) = h(y)$, we have

$$
\begin{aligned}
size(x') &= size(x) + size(y) \\
&\geq 2^{h(x)} + 2^{h(y)} \\
&\geq 2^{h(y)+1} \\
&= 2^{h(x')}.
\end{aligned}
$$

- $h(x) > h(y)$, it is similar to the first case

Lemma For n items, the running time of Create-Set is $O(1)$, Find-Set is $O(\log n)$, and Union is $O(\log n)$ respectively.

Proof Obviously, Create-Set(x) is $O(1)$, and the running time of Union(x, y) depends on Find-Set(x). Since the running time of Find-Set(x) depends on the height of the tree. From previous lemma, for any tree, we have

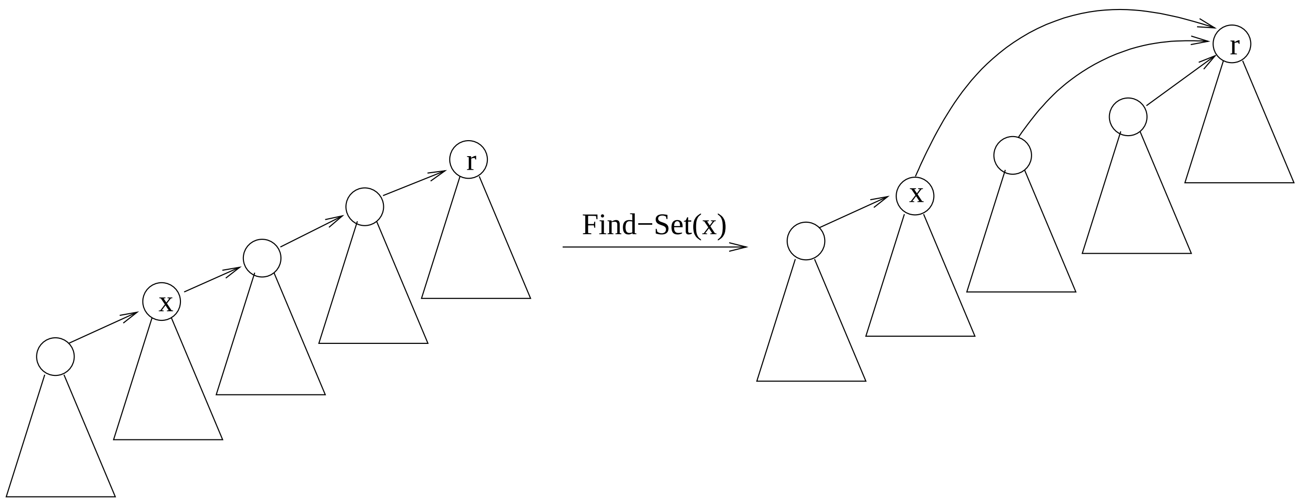
$$\begin{aligned} n &\geq 2^h \\ \Rightarrow h &\leq \log n \\ \Rightarrow h &= O(\log n) \end{aligned}$$

Hence we have Find-Set(x) = $O(\log n)$.

Up-tree implementation : path compression

We can make the running time even faster if we add another trick.

In the Find-Set(x), we trace the path from x to the root. Let r be the root of the tree, and the path from x to r is $xa_1a_2 \ldots a_kr$. We also make all the parent pointers of $x, a_1, a_2, \ldots a_k$ pointing to r directly. This idea is called **path compression**.



Up-tree implementation : path compression

It is expected that in some *sequence* Find-Set operation, the running time is expected to be faster (By how much?).

To *understand* the running time, we first have to define $\lg^{(i)} n$ and $\lg^* n$ (iterated logarithm).

Let the function $\lg^{(i)} n$ be defined recursively for non-negative integers i as

$$\lg^{(i)} n = \begin{cases} n & \text{if } i = 0 \\ \lg(\lg^{(i-1)} n) & \text{if } i > 0 \text{ and } \lg^{(i-1)} n > 0, \\ \text{undefined} & \text{if } i > 0 \text{ and } \lg^{(i-1)} n \leq 0, \\ & \text{or } \lg^{(i-1)} n \text{ is undefined.} \end{cases}$$

The iterated logarithm is defined as

$$\lg^* n = \min \{i \geq 0 : \lg^{(i)} n \leq 1\},$$

which is a *very slow growing function*. We have
$\lg^* 2 = 1, \lg^* 4 = 2, \lg^* 16 = 3,$
$\lg^* 65536 = 4, \lg^* 2^{65536} = 5.$

Up-tree implementation : path compression

The following theorem is stated without proof.

Theorem A sequence of m Create-Set, Find-Set and Union operations, n of which are Create-Set operations, can be performed on a disjointed-set forest with union by height and path compression in worst-case time $O(mlg^*n)$.

Question : What is the running time of Kruskal's algorithm if we employ this implementation of disjoint set Union-Find ?