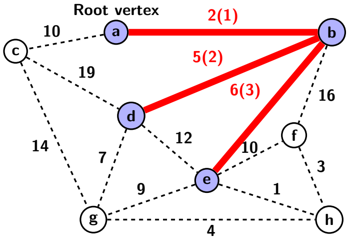
# Implementation Aspect

- Use a priority queue to store tuple ($v$, d[$v$])
- d[$v$] is the weight of the lightest edge connecting $v$ to some vertex in $S$.
- pred[$v$] = $u$, where $u \in S$ is the end point of the lightest edge connecting $v$.
- So, the collection of pred[.] yields the MST.
- d[$u$] should be used as a key to extract the triple from the priority queue.

# Implementation Aspect




min $\longrightarrow$

| $v$ | d[$v$] | pred[$v$] |
|---|---|---|
| $c$ | 10 | $a$ |
| $f$ | 10 | $e$ |
| $g$ | 7 | $d$ |
| $h$ | 1 | $e$ |

- The lightest edge connecting a $v \in S$ to a $w \in V - S$: $(e, h)$.
- After deleting this triple, update d[$v$] and pred[$v$] in relaxation step.

# Pseudo Code

```
foreach v ∈ V {
    d[v] = ∞;
    color[v] = "W"; // No path at all
    pred[v] = "U"; // undefined
}
d[s] = 0;        // source vertex s
pred[s] = nil; // so, s has no pred
Q = newPriorityQ((v,d[v]) for all v ∈ V);
while (!isEmpty(Q)) {
     u = Q.deleteMIN(); // minimum d[v]
     foreach v ∈ ADJ[u]
          RelaxEdge(G, (u,v));
     color[u] = "B" // included in S
}
```

# Code for Relaxation

```
RelaxEdge(G, (u,v)) {
    if (color[v] = "W" && (w(u,v) < d[v])){
        d[v] = w(u,v); // new lightest edge
        Q.decreaseCost(v, d[v]); // decrease d[v]
        pred[v] = u; // Current predecessor
    }
}
```

# Running Time

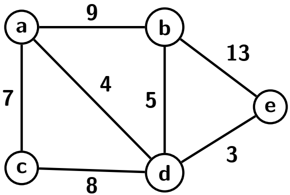
- O($\log |V|$) to extract vertex out of queue.
- Done once for each vertex, total time O($|V| \log |V|$)
- O($\log n$) to decrease d[$v$] of neighboring vertex.
- Done at most once for each edge: O($|E| \log |V|$)
- Total cost O($(|V| + |E|) \log |V|$).

# Kruskal's Algorithm

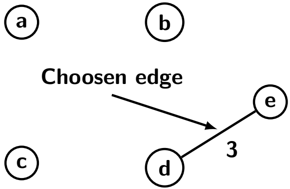
1. Sort all the edges in non-decreasing order of edge weights.
2. Pick the lightest edge and add to an initially empty tree, insert the edge if no cycle is formed.
3. Otherwise discard the edges and return back to Step 2 until $|V| - 1$ edges are included in the tree.

```
Q = newPriorityQ(allTriplets);   // (u,v,w_uv)
T = Φ; // No edge initially
foreach (v ∈ V)
     S_v = {v}; // Each v is a set
while (|T| < |V| − 1) {
     (u,v,w_uv) = Q.deleteMIN(); // minimum w_uv
     S_v = FIND(v);
     S_u = FIND(u);
     if (S_v ≠ S_u) {
         UNION(S_u, S_v);
         T = T ∪ {(u,v)};
     }
}
```

# Kruskal Example




Original graph




Initial forest

min $\longrightarrow$

| Edge | Weight |
| --- | --- |
| (d, e) | 3 |
| (a, d) | 4 |
| (b, d) | 5 |
| (a, c) | 7 |
| (a, b) | 9 |
| (b, d) | 13 |

$T = \{(d,e)\}$

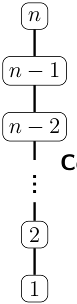
# UNION and FIND

- Operation on disjoint set is a motivation for Kruskal's algorithm.
- It is one of the simplest algorithms but have an involved proof for running time.
- The simplest representation is a vector $R$ where $R[i]$, for $1 \leq i \leq n$ contains the name of the set to which $i$ belongs.
- So, initially $R[i] = i$ because each set $\{i\}$ is a set by itself.

# UNION and FIND

- For UNION($A, B, C$), scan $R$ change entries having names $A$ and $B$ to $C$.
- The cost for this is O($n$).
- A better way is to use trees, where each set as a rooted tree.
- The root of the tree contains the name of the set.
- Every vertex including the root assume to represent an element.
- We also maintain a count of the number of nodes in each tree.

# A Pathological Case



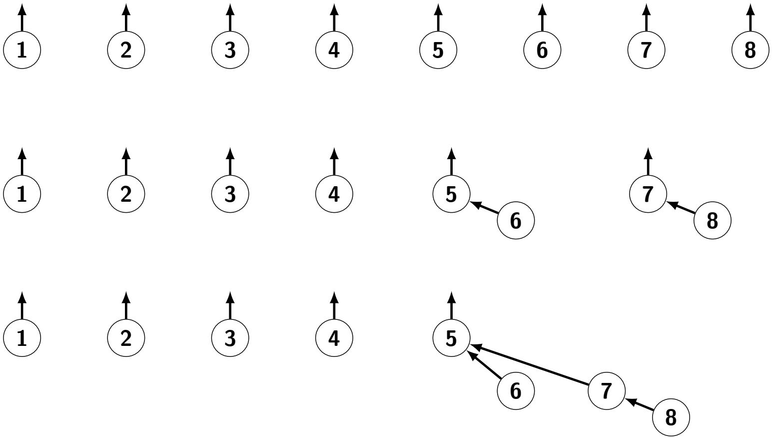

**Cost of FINDs** $= \sum_{0}^{n-1} i$

UNION(1, 2, 2)
UNION(2, 3, 3)
⋮
UNION($n-1$, $n$, $n$)
FIND(1)
FIND(2)
⋮
FIND($n$)

# UNION and FIND

- Cost of merging tree is O(1) as it just requires the root of one tree become a child of the root of other tree.
- So UNION can be done in O(1) time.
- FIND can take at most time of O($n$) time, as we need to climb up the tree from the vertex (a element) to the root of the tree (name of the set) where it belongs.
- Merging is performed by keeping track of size of each tree.
- Then making the root of the smaller tree as child of the root of the larger tree.

# UNION and FIND



# UNION and FIND

**Lemma**

*Executing UNION makes the root of smaller tree a child of root of larger tree then no tree in forest has a height $\geq h$ unless it has $2^h$ vertices.*

**Proof.**

- Proof is by induction on height of the tree.
- For $h = 0$, every tree has $2^0 = 1$ vertex .
- Assume it to be true for all values $\leq h - 1$.

$\square$

**Proof.**

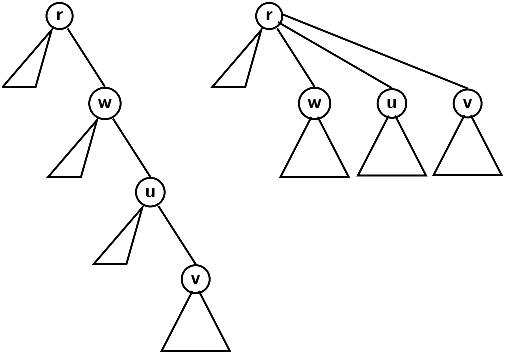
- Now consider tree of $T$ of height $h$ with fewest number of nodes.
- $T$ must have been obtained by merging of two trees $T_1$ (larger) and $T_2$ (smaller) where,
  - $T_1$ has a height $h-1$ and $T_2$ has no more than number of nodes in $T_1$.
  - $T_1$ by induction has $2^{h-1}$ nodes.
  - $T_2$, therefore has $2^{h-1}$
- Hence, $T$ has at least $2^h$ nodes.

$\square$

# UNION and FIND

- Since the root of the smaller tree becomes the child of the root of larger tree, no tree can have a height larger than $\log n$.
- So, O($n$) UNION and FIND can cost at most O($n \log n$).
- The running time can be improved to O($G(n)n$) using path compression, where
  - $G(n) \leq 5$ for $n \leq 2^{65536}$.
- Analysis is involved and is not covered.

# Path Compression




- Effect of path compression during FIND($v$).
- The cost is amortised across all future FIND operations.

# Running Time

- Creating disjoint sets: O($|V|$).
- Building priority queue: O($|E|$)
- For queue manipulation: O($|E| \log |V|$).
- Total time O($|E| \log |E|$).

# Biconnectivity

**Definition (Articulation Point)**

A vertex $a$ is called articulation point of $G$ if there exists a pair of vertices $v$ and $w$ such that $a$, $v$ and $w$ are distinct and every path between $v$ to $w$ passes through $a$. A graph is said to be biconnected if it does not have any articulation point.

- Removal of $a$ will split $G$ into two or more parts.
- In other words, $G$ is biconnected if $G$ contains no articulation point.