CS2040 Data Structures and Algorithms
Lecture Note #13

Graphs

Part 4: Minimum Spanning Tree (MST), Modified Dijkstra's

Dijkstra's algorithm

A quick review:

- At initialization
  - set the distances of all nodes to infinity (except s)
  - enter all nodes into a priority queue
- Dequeue nodes one by one
  - For all neighbors of a node, update distance with pq.decreaseKey(w, d) if d is better

- $\rightarrow$ Need priority queue with decreaseKey(w, d) function
- $\rightarrow$ Need auxiliary mapping data structure to get $O(\log V)$ performance

Dijkstra's algorithm with Lazy PQ

A new idea:

- "Lazy" use of the priority queue
  - Note: in CS "lazy" generally just means that some operations or book-keeping is deferred until later
- Enter nodes when they are visited
- Allow enqueuing nodes more than once!
- But, ignore nodes that have already been processed
- If a node is in the queue more than once, how to keep track of whether it was processed?
  - The lowest-distance copy of a node will always be processed first! (Thus, later copies can be ignored.)

Modified Implementation of Dijkstra's Algorithm

Key ideas:

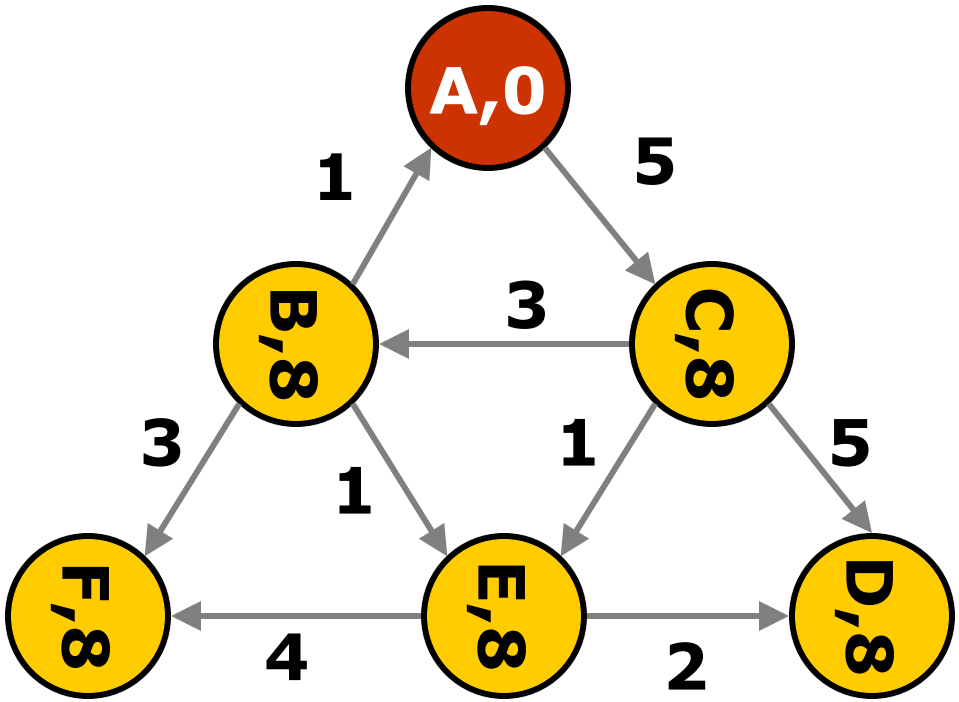
- Allow a vertex to be possibly processed multiple times as detailed below and in the next slide
- Use a **built-in** priority queue in **Java Collections** to order the next vertex **u** to be processed based on its **D[u]**
  - This vertex information is stored as IntegerPair **(d, u)** where **d** = **D[u]** (the current shortest path estimate)
- But with modification: We use **"Lazy Data Structure"** strategy
  - **Main idea:** No need to maintain just one IntegerPair (shortest path estimate) for each vertex **v** in the PQ
  - Can have multiple shortest path estimates to exist in the PQ for a vertex **v**

Modified Implementation of Dijkstra's Algorithm

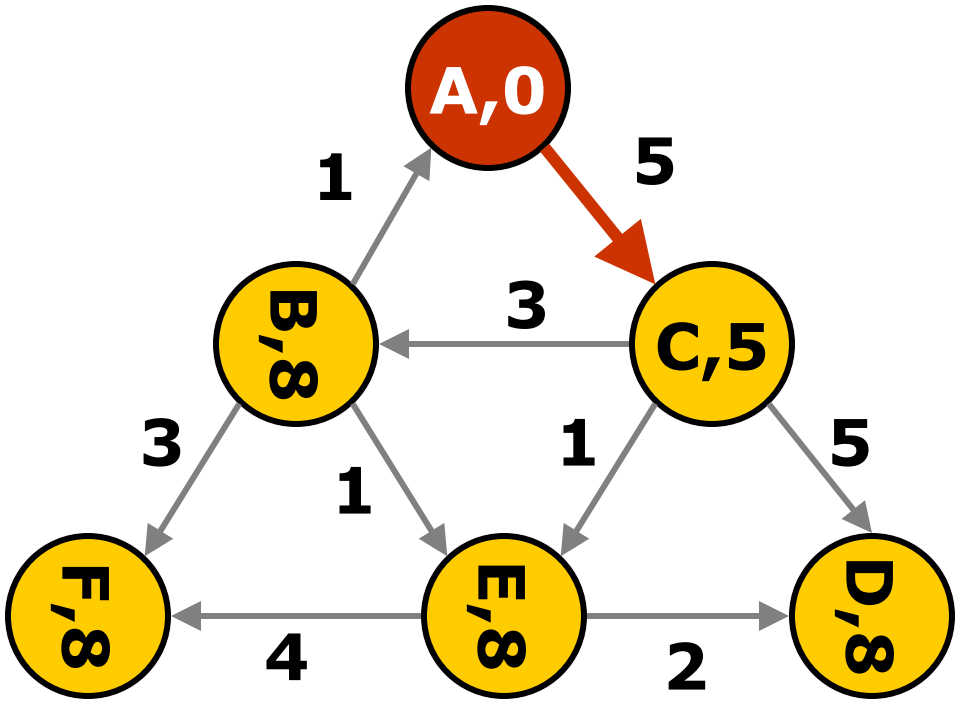
Lazy DS: Extract pair **(d, u)** in **front of the priority queue PQ** with the minimum shortest path estimate ***so far***

- If **d == D[u]**, we relax all edges out of **u**,
  else if **d > D[u]**, we discard this inferior **(d, u)** pair
  - Since there can be multiple copies of **(d, u)** pair we only want the most up to date copy
  - See below to understand how we get multiple copies!
- If during edge relaxation, **D[w]** of a neighbor **w** of **u** *decreases*, enqueue a new **(D[w], w)** pair for *future propagation* of shortest path estimate
  - No need to find the **w** in the **PQ** and update it!
  - Thus, no need to implement **DecreaseKey** (which you don't have in Java PriorityQueue class) or need bBST implementation of PQ!

Dijkstra's algorithm with Lazy PQ

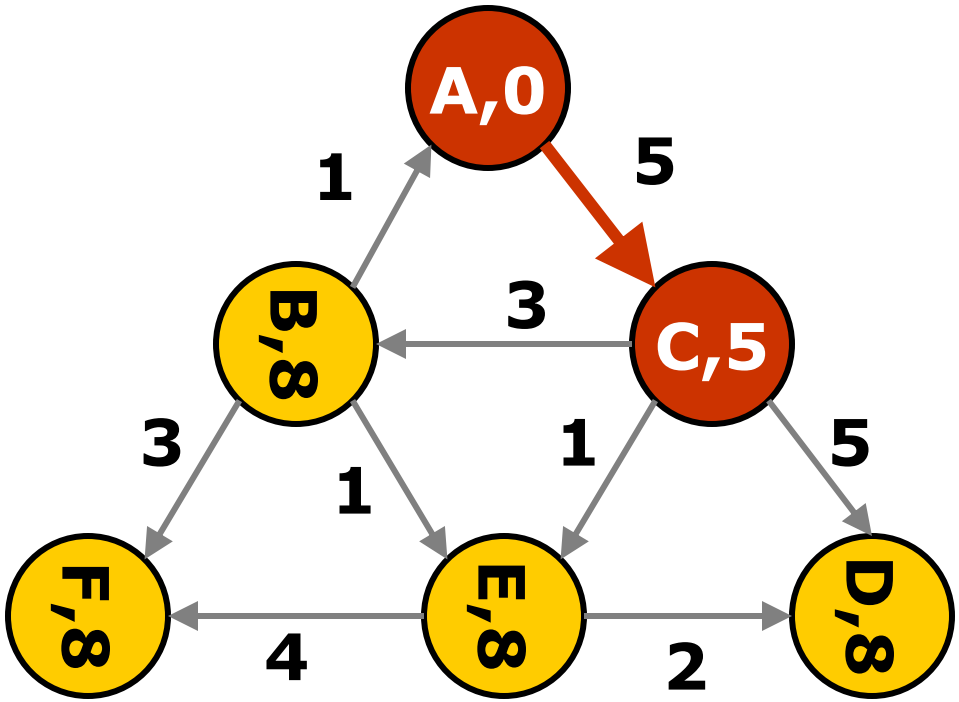


Dijkstra's algorithm with Lazy PQ

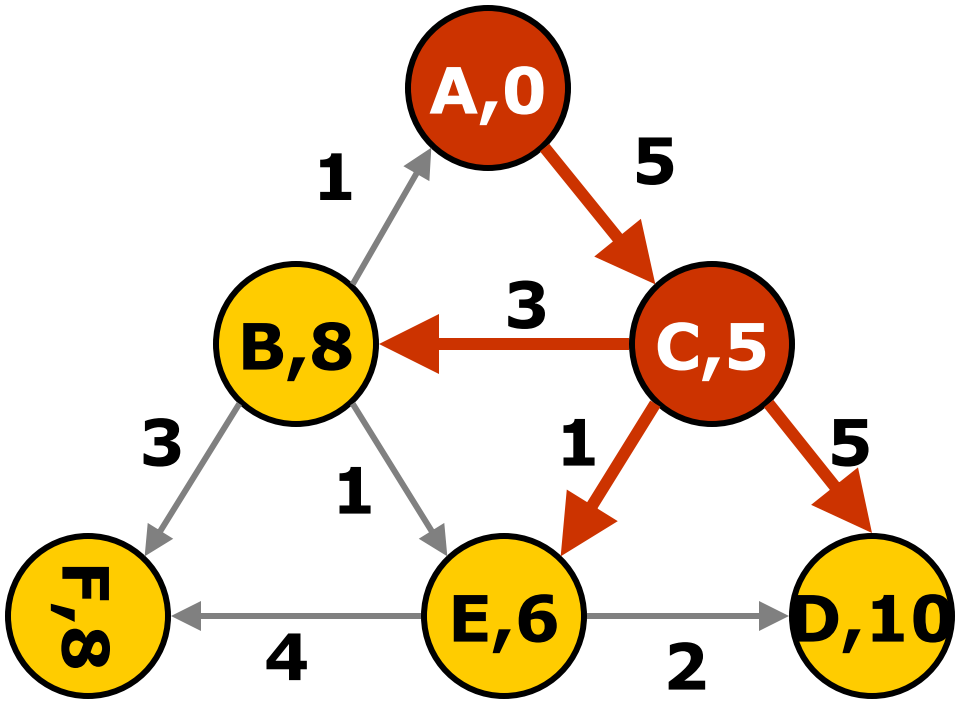


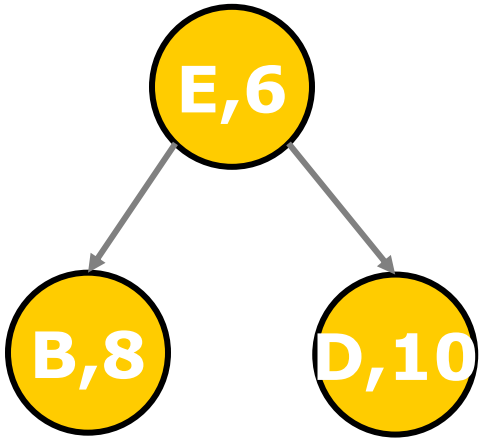
C,5

Dijkstra's algorithm with Lazy PQ

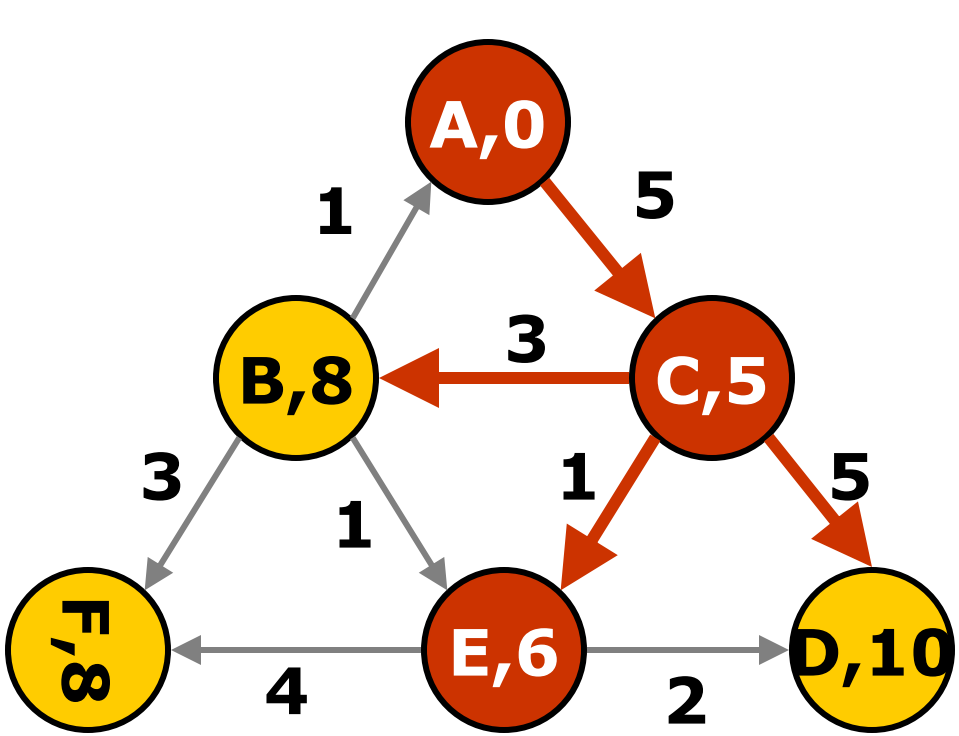


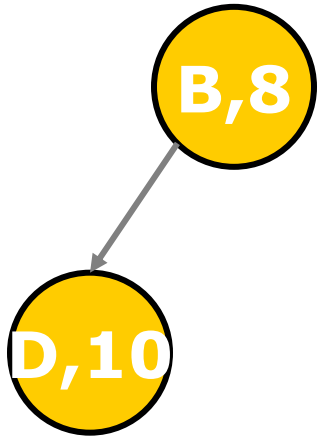
Dijkstra's algorithm with Lazy PQ



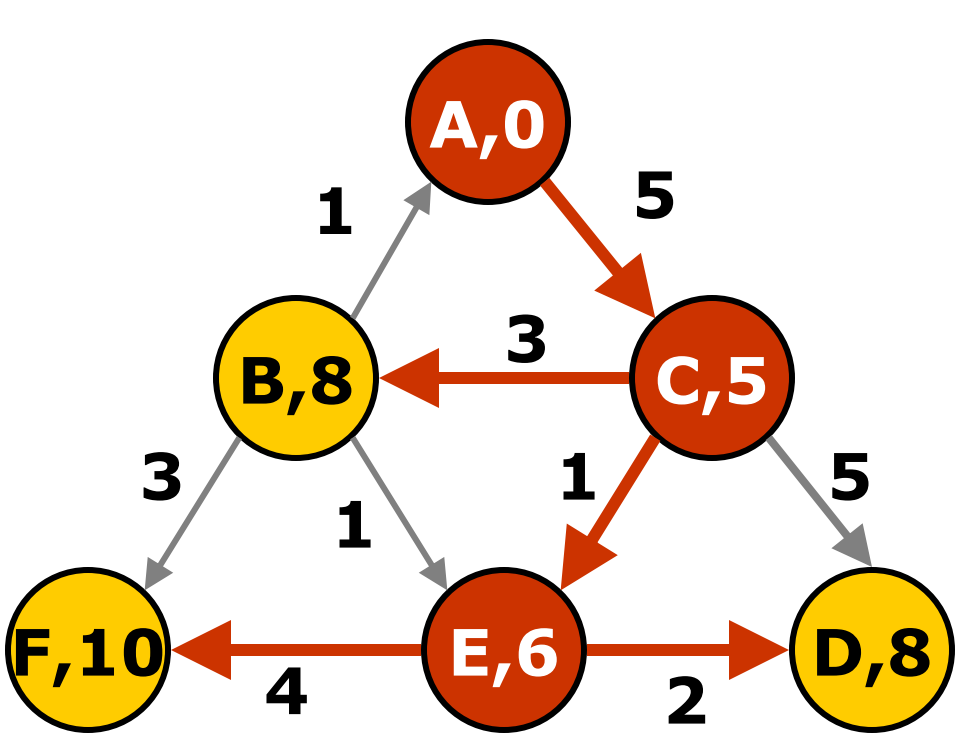


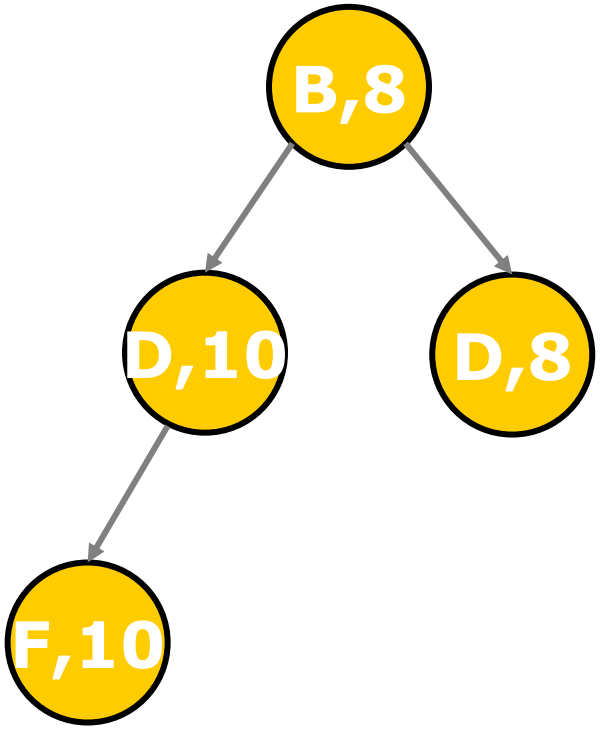
Dijkstra's algorithm with Lazy PQ



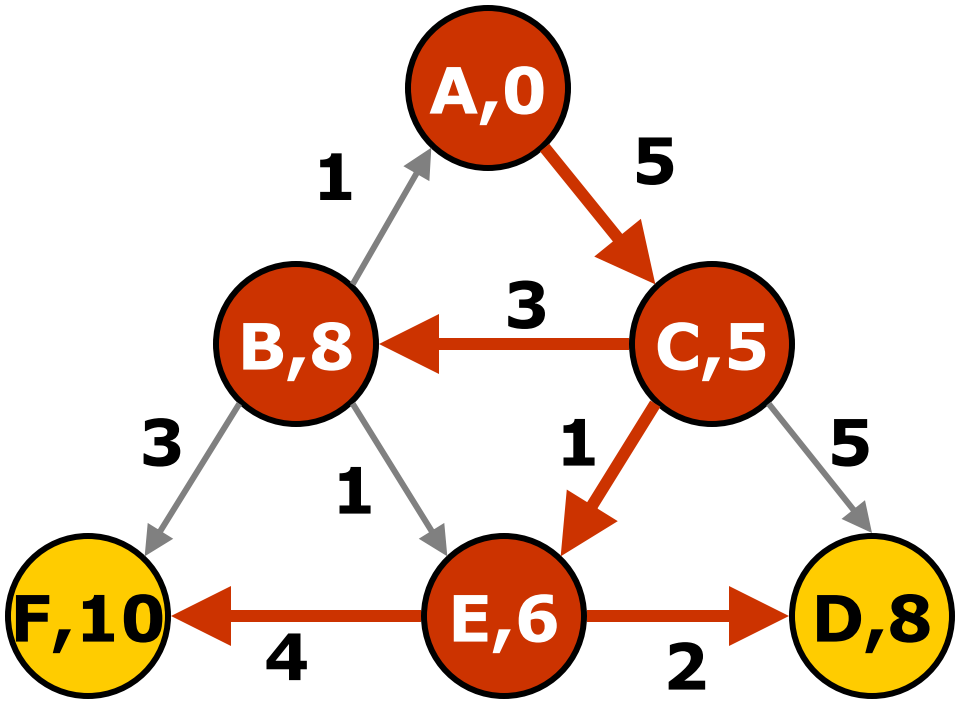


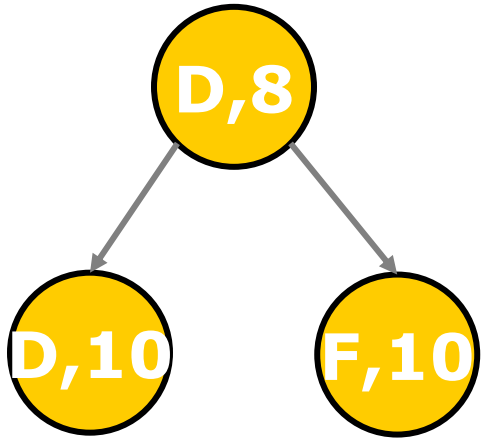
Dijkstra's algorithm with Lazy PQ



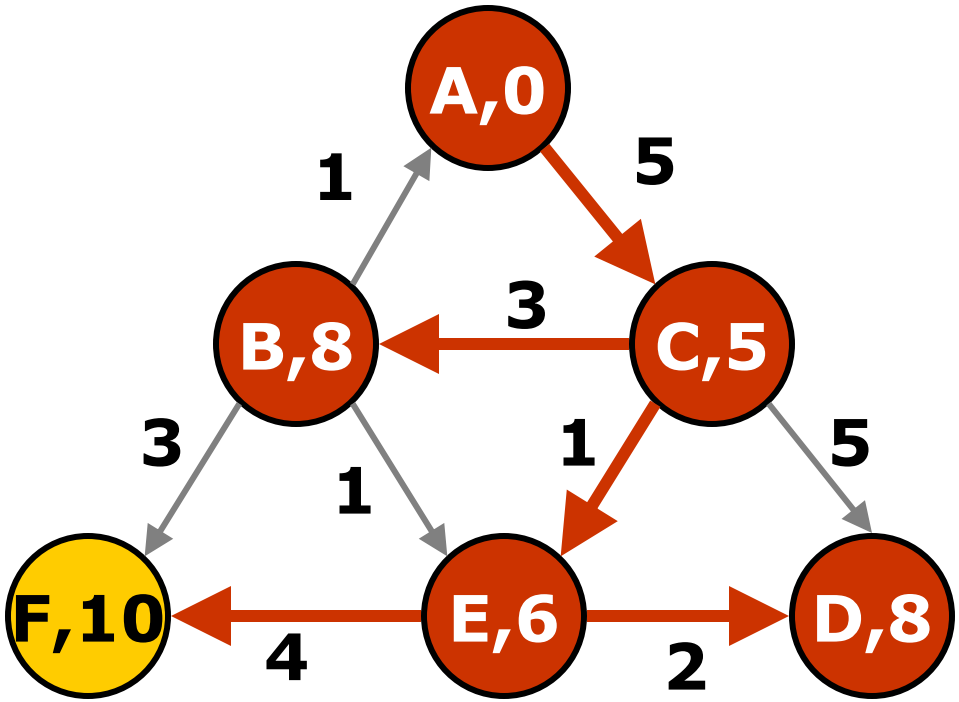


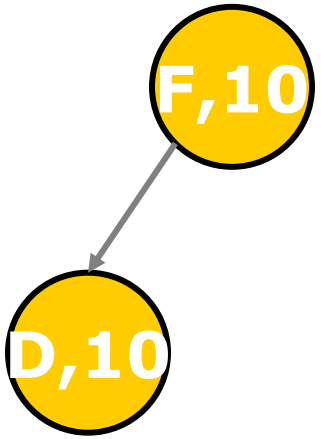
Dijkstra's algorithm with Lazy PQ



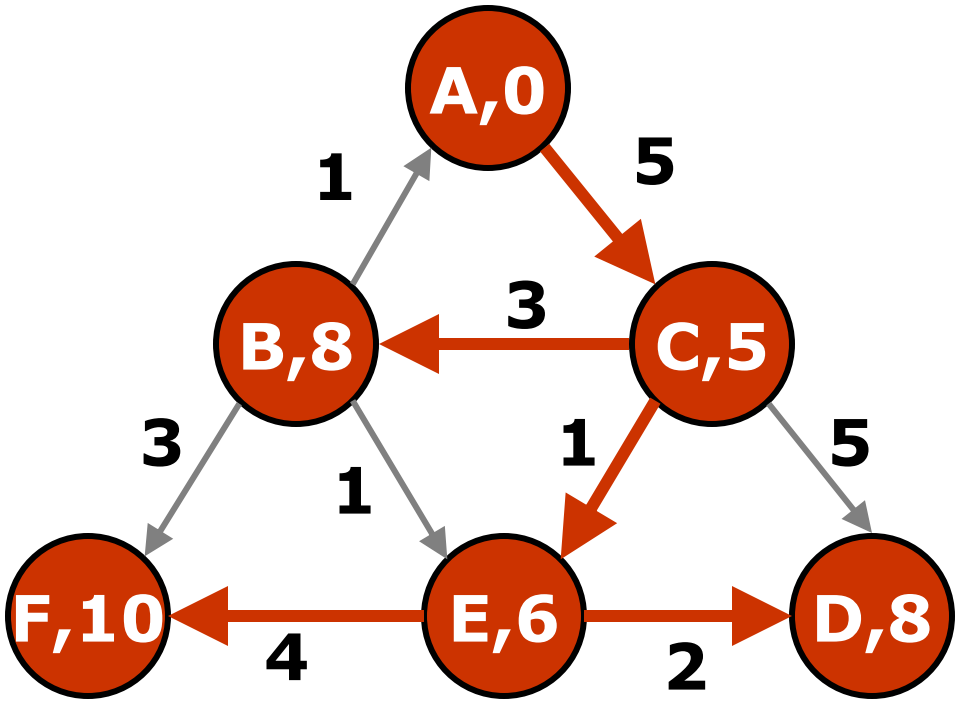


Dijkstra's algorithm with Lazy PQ





Dijkstra's algorithm with Lazy PQ



D,10

Initialization O(V)

```
foreach vertex w
   D[w] = INFINITY          // distance array
D[s] = 0
pq = new PriorityQueue()
pq.push(s, 0)   // enqueue source, d = 0
count = |V|     // optimization to terminate
                // early
```

Main loop O((V+E) log V)

```
while (pq is not empty && count > 0)
   (d, u) = pq.deleteMin()
   if d == D[u] then
      count--
      foreach neighbour w of u
          if D[w] > D[u] + cost(u,w) then
              D[w] = D[u] + cost(u,w)  // relax
              pq.push((D(w), w))  // (re-)encode
              parent(w) = u
```

Modified Implementation of Dijkstra's Algorithm

Key ideas:

- Allow a vertex to be possibly processed multiple times as detailed below and in the next slide
- Use a **built-in** priority queue in **Java Collections** to order the next vertex **u** to be processed based on its **D[u]**
  - This vertex information is stored as IntegerPair **(u, d)** where **d** = **D[u]** (the current shortest path estimate)
- But with modification: We use **"Lazy Data Structure"** strategy
  - **Main idea:** No need to maintain just one IntegerPair (shortest path estimate) for each vertex **v** in the PQ
  - Can have multiple shortest path estimates to exist in the PQ for a vertex **v**

Modified Implementation of Dijkstra's Algorithm

Lazy DS: Extract pair **(d, u)** in **front of the priority queue PQ** with the minimum shortest path estimate ***so far***

- If **d == D[u]**, we relax all edges out of **u**,
  else if **d > D[u]**, we discard this inferior **(d, u)** pair
  - Since there can be multiple copies of **(d, u)** pair we only want the most up to date copy
  - See below to understand how we get multiple copies!
- If during edge relaxation, **D[w]** of a neighbor **w** of **u** *decreases*, enqueue a new **(D[w], w)** pair for *future propagation* of shortest path estimate
  - No need to find the **w** in the **PQ** and update it!
  - Thus, no need to implement **DecreaseKey** (which you don't have in Java PriorityQueue class) or need bBST implementation of PQ!

Complexity

- Let's have a look at the complexity and whether it differs from regular Dijkstra's
- How long can the queue get?
  - The number of edges is an upper bound on the queue size (let's call it $|E|$)
  - The upper bound for the number of edges is $|E| \leq |V^2|$
  - Each enqueue/dequeue operation is $O(\log E)$ on the queue size $|E|$, $|E| \leq |V^2|$, thus $O(\log V^2) = O(2 \times \log V) = O(\log V)$
- Same big-*O* complexity, but possibly a constant factor difference

End of Part 1

Minimum Spanning Tree

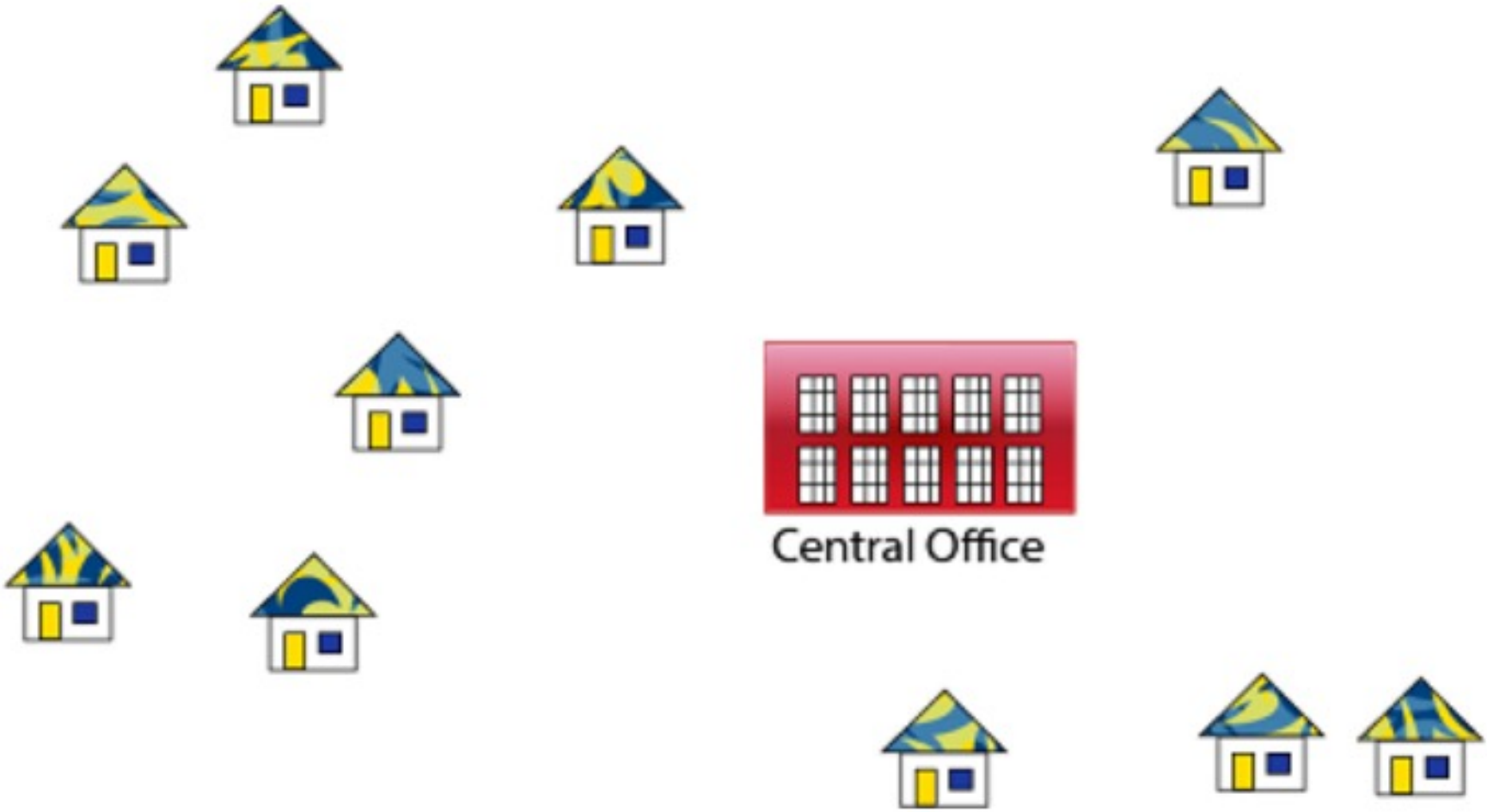
- Minimum Spanning Tree (MST) problem: Given connected graph G with positive edge weights, find a min weight set of edges that connects all of the vertices.

- *How many edges does a minimum spanning tree have?*
  A minimum spanning tree has (V – 1) edges where V is the number of vertices in the given graph.

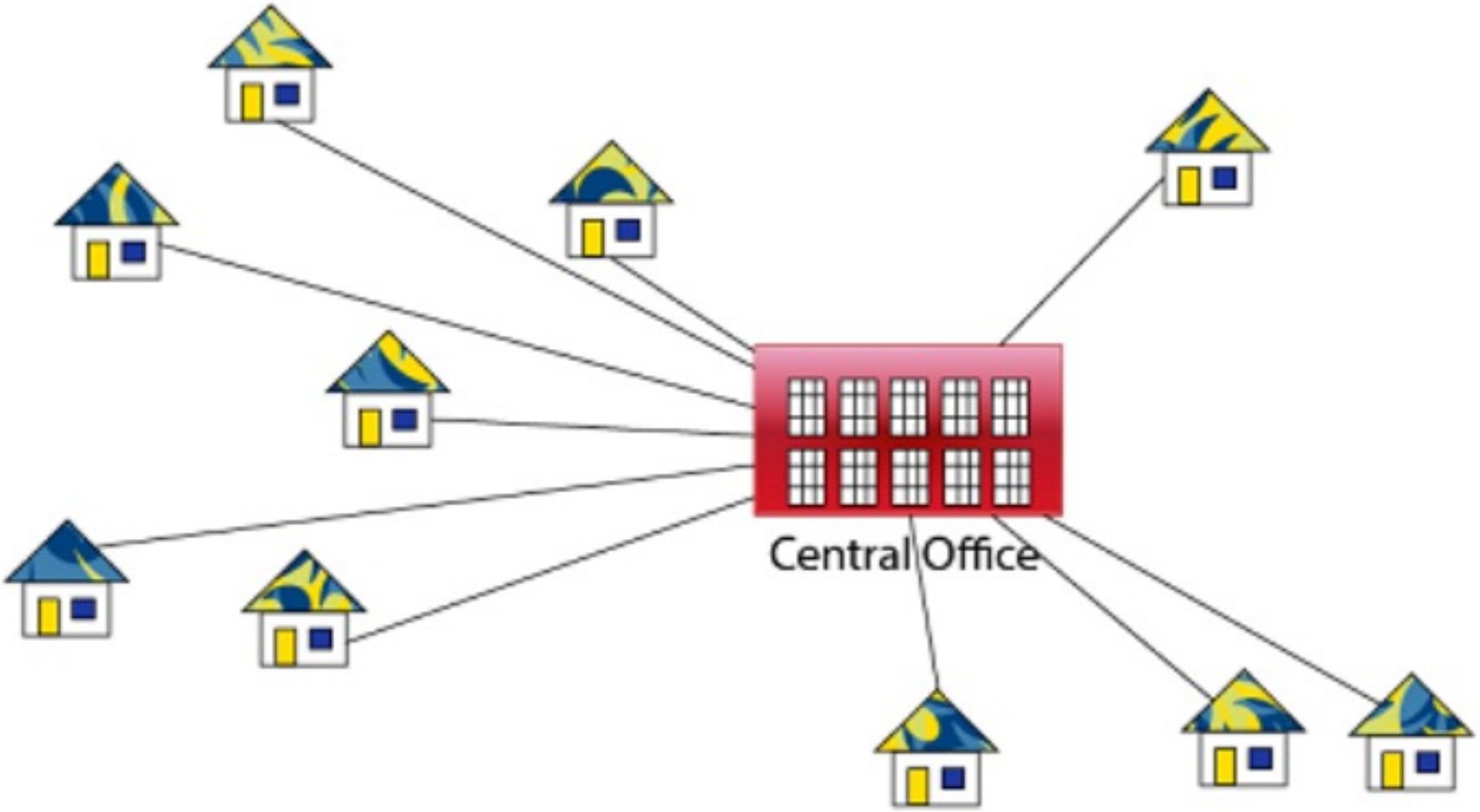
Applications

- **Network design**
  - *telephone*
  - *electrical*
  - *hydraulic*
  - *TV cable*
  - *computer*
  - *road*

For Example, Problem laying Telephone Wire.

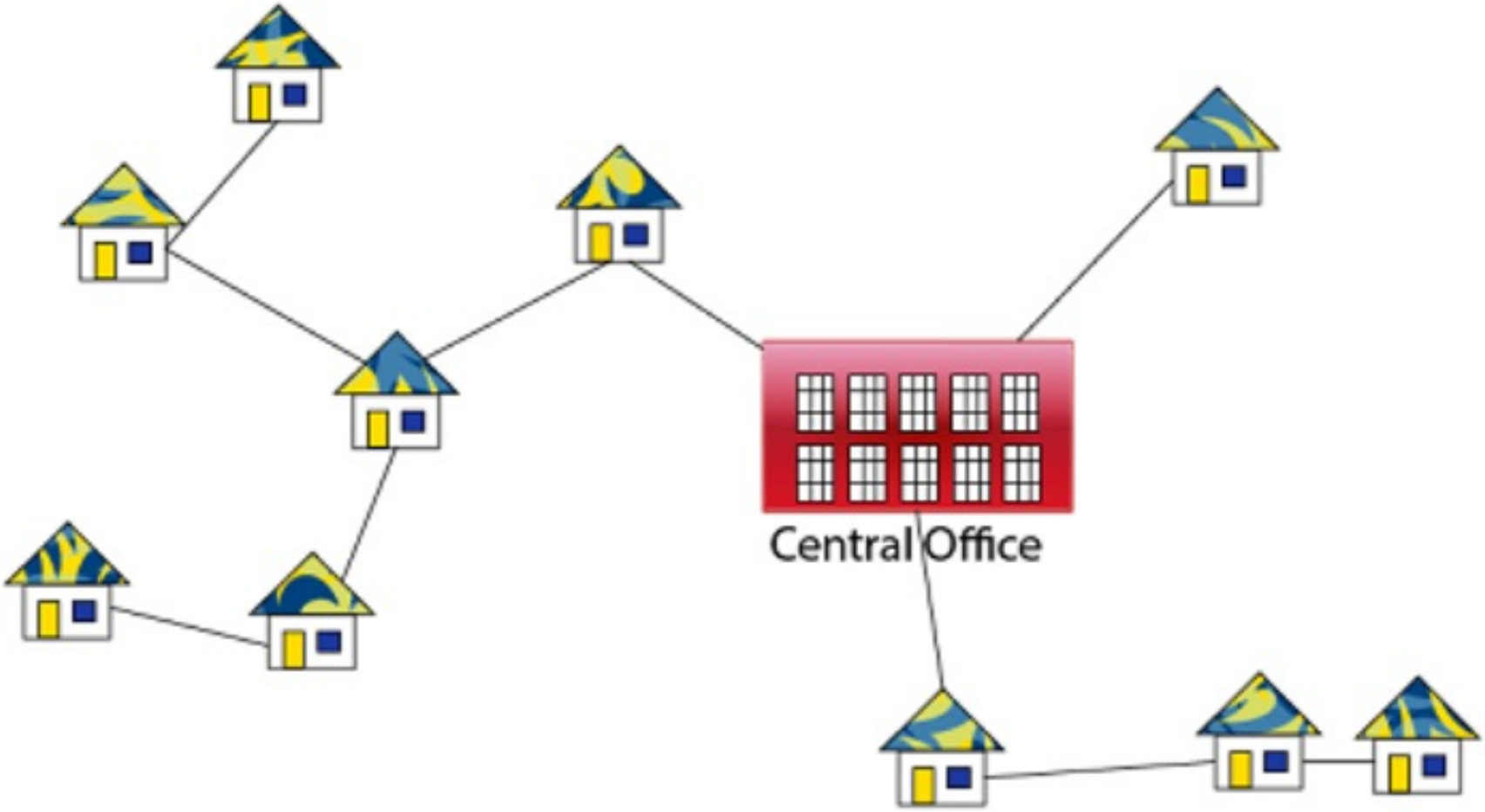


Wiring : Naive Approach



Expensive!

Wiring : Better Approach



Minimize the total length of wire connecting the customers

Two MST algorithms

- **Kruskal's Algorithm**
- Prim's Algorithm
- Both are Greedy Algorithm.
- The Greedy Choice is to put the smallest weight edge that does not become a cycle in the MST constructed so far.

Gordon Gekko in 1987 film *Wall Street*:
"Greed, for lack of a better word, is good."

UNION-FIND DISJOINT SETS DATA STRUCTURE

Union-Find Disjoint Sets (UFDS)

UFDS is a collection of disjoint sets

Given several disjoint sets in the UFDS the operations we have are

- Union two disjoint sets when needed
- Find which set an item belongs to
- Check if two items belong to the same set

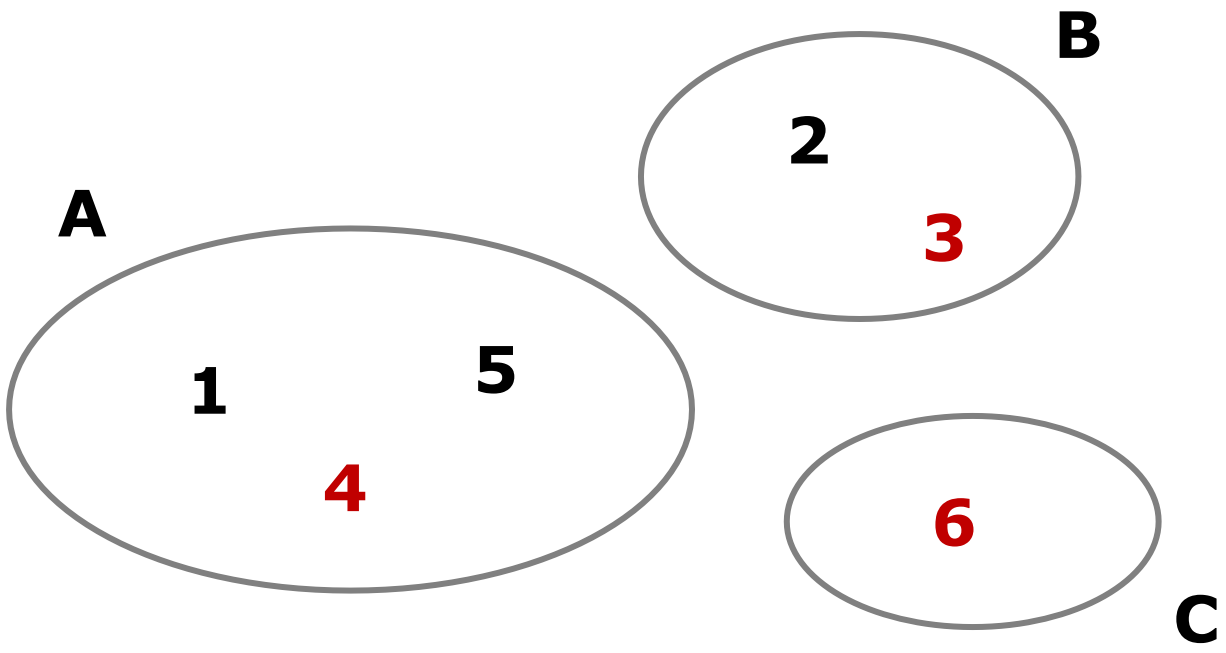
- Ordering of items in the disjoint sets are not important
- For our discussion, we will only deal with integer values in the disjoint sets

Union-Find Disjoint Sets (UFDS)

Key ideas:

- Each set is modeled **as a tree**
  - Thus, a collection of disjoint sets forms **a forest of trees**
- Each set is represented by a representative item
  - which is the root of the corresponding tree of that set

Ex. with 3 Disjoint Sets



- We represent sets with a principal or representative element

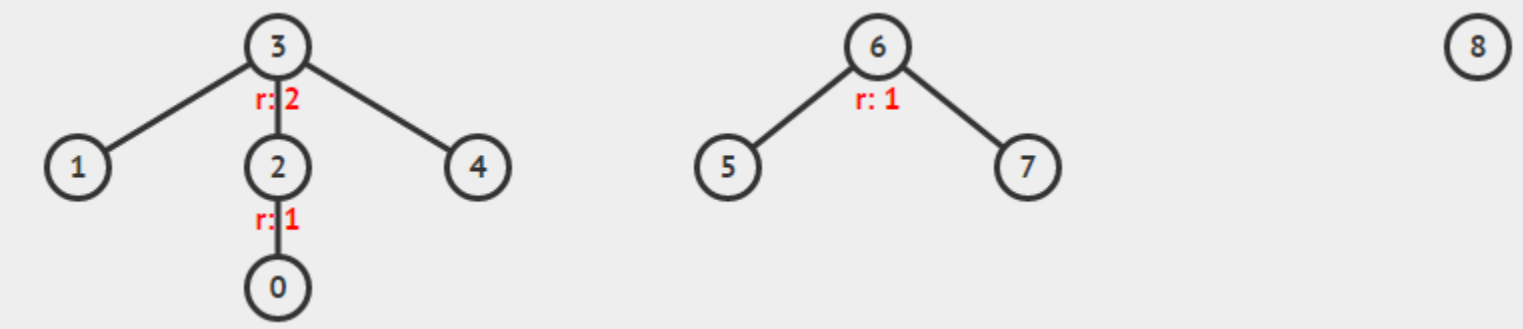
Data Structure to store UFDS

We can record this forest of trees with an array **p**

- **p[i]** records the parent of item **i**
- if **p[i] = i**, then **i** is a root
  - and also the representative item of the set that contains **i**

For the example below,
we have **p = {2,3,3,3,3,6,6,6,8}**

index: 0,1,2,3,4,5,6,7,8

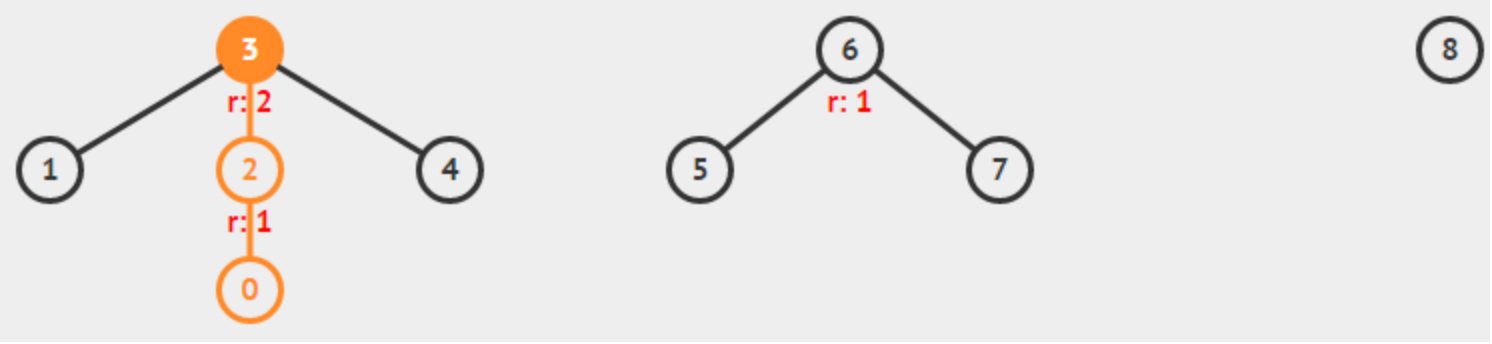


UFDS– findSet(i) Operation

For each item **i**, we can **find** the representative item of the set

that contains item **i**

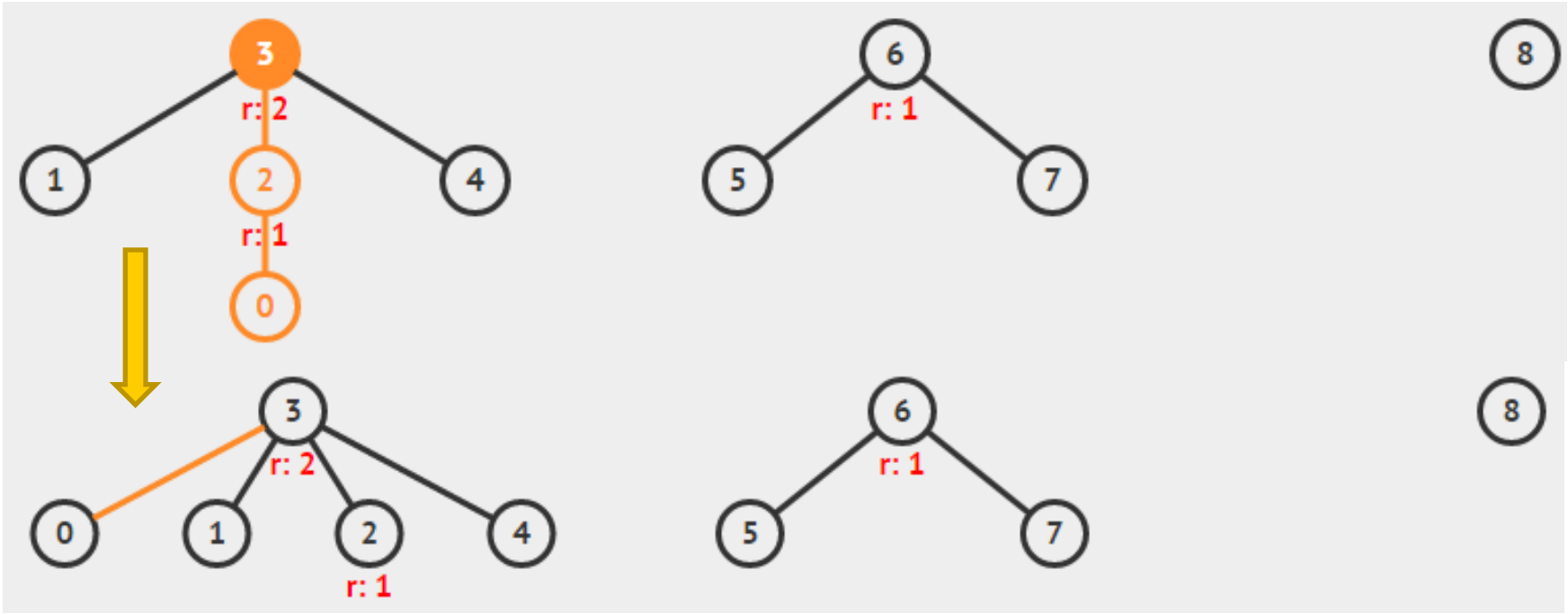
by recursively visiting **p[i]** until **p[i] = i**;



UFDS– findSet(i) Operation

Then, we *compress the path* to make future find operations (very) fast, i.e., O(1)

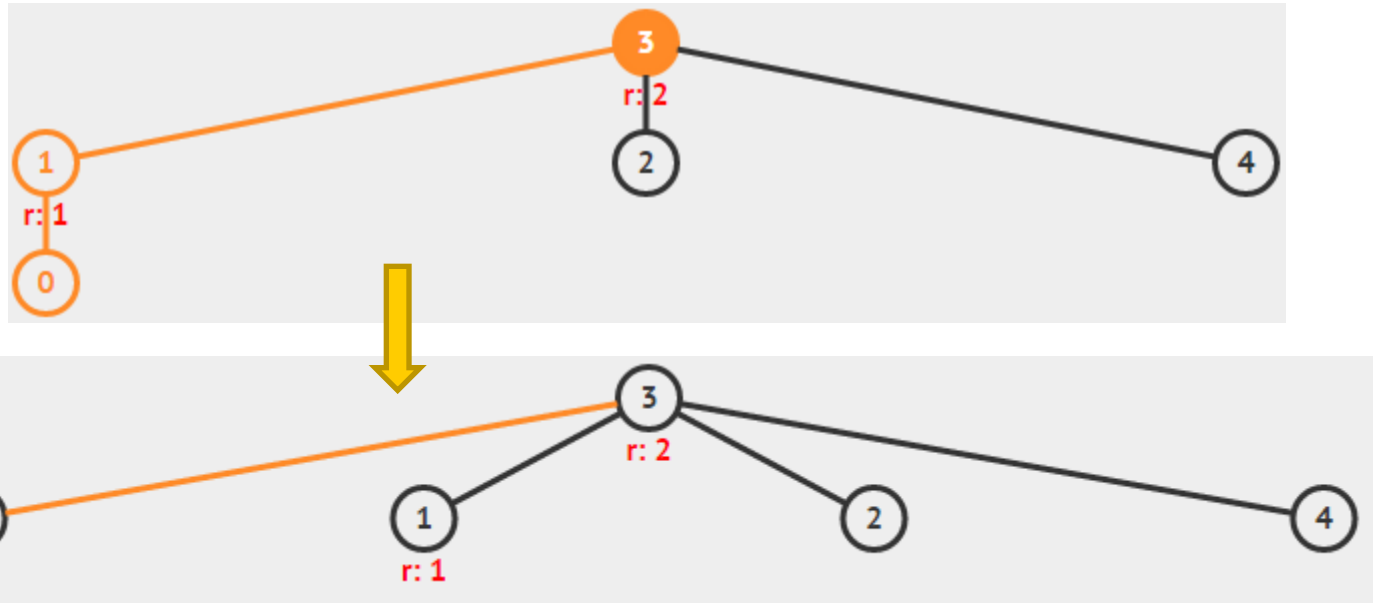
- Example of findSet(0), *ignore attribute 'r' for now*



findSet code

```
public int findSet(int i) {
   if (p[i] == i)
      return i;
   else {
      p[i] = findSet(p[i]);
      return p[i];
   }
}
```

findSet(0)



UFDS – isSameSet(i,j) Operation

For item **i** and **j** we can check whether they are in the same set in O(1) by finding the representative item for i and j and checking if they are the same or not

- Example: isSameSet(0,4) will return false

isSameSet code

```
public Boolean isSameSet(int i, int j) {
   return findSet(i) == findSet(j);
}
```

As the representative items of the sets that contains item 0 and 4 are different, we say that 0 and 4 are not in the same set!

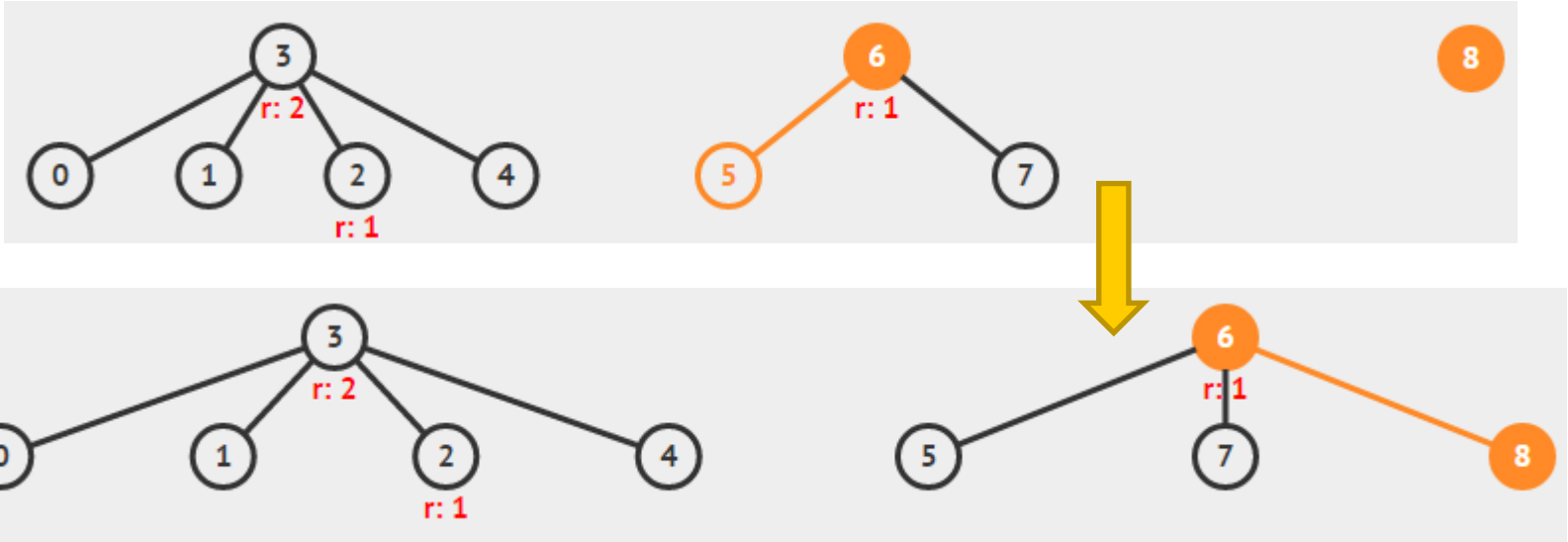
isSameSet (0,4)

UFDS – unionSet(i,j) Operation (1)

If two items **i** and **j** currently belong to different disjoint sets, we can **union** them by setting the representative item of *the one with taller* tree* to be the new representative item of the combined set

UFDS – unionSet(i,j) Operation (1)

- Example of unionSet(5, 8), *see attribute 'r' (elaborated soon)*



UFDS – unionSet(i,j) Operation (2)

This is called the *"Union-by-Rank"* ***heuristic***

- This helps to make the resulting combined tree shorter
  - Convince yourself that doing the opposite action will make the resulting tree taller (we do not want this)

If both trees are equally tall, this heuristic is not used

We use another integer array **rank,** where **rank[i]** stores the upper bound of the height of (sub)tree rooted at **i**

- This is just an upper bound as path compressions can make (sub)trees shorter than its upper bound and we do not want to waste effort maintaining the correctness of **rank[i]**

Complexity

- findSet() complexity "sketch"
  - Why is union-by-rank helpful?
  - To increase the rank, the fastest way to do that is to combine two sets of similar size, i.e., same rank
  - → Need to double the number of nodes to increase the rank by 1
  - → The height of the sets is logarithmic in the size of the sets, so findSet is O(log N)
- With **path compression**, complexity can be close to constant, i.e., O(1)

unionSet code

```
public void unionSet(int i, int j) {
  if (!isSameSet(i, j)) {
    int x = findSet(i), y = findSet(j);
    // rank is used to keep the tree short
    if (rank[x] > rank[y])
      p[y] = x;
    else {
      p[x] = y;
      if (rank[x] == rank[y]   // rank increases
        rank[y] = rank[y]+1;   // only if both trees
    }                          // initially have the
                               // same rank
  }
}
```

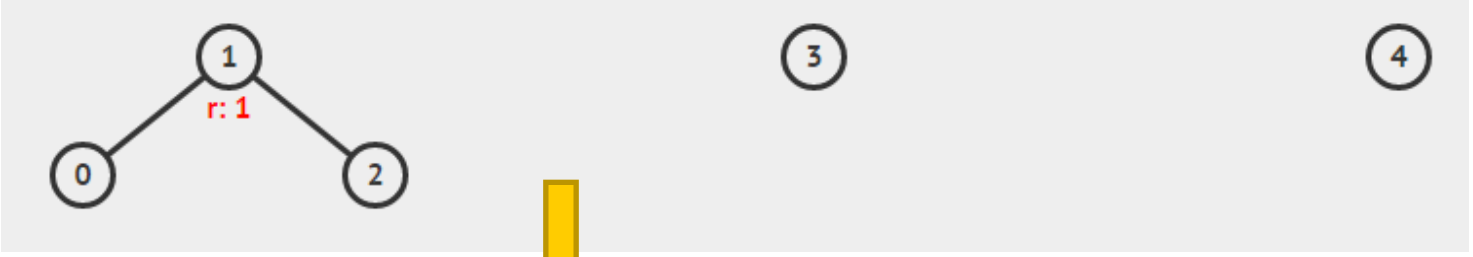
unionSet code

```
public void unionSet(int i, int j) {
  if (!isSameSet(i, j)) {
    int x = findSet(i), y = findSet(j);
    // rank is used to keep the tree short
    if (rank[x] > rank[y])
      p[y] = x;
    else {
      p[x] = y;
      if (rank[x] == rank[y])   // rank increases
        ra
    }

  }
}
```

unionSet(0,2)

unionSet code

```
public void unionSet(int i, int j) {
  if (!isSameSet(i, j)) {
    int x = findSet(i), y = findSet(j);
    // rank is used to keep the tree short
    if (rank[x] > rank[y])
      p[y] = x;
    else {
      p[x] = y;
      if (rank[x] == rank[y]) // rank increases
        rank[y] = rank[y]+1;  // only if both trees
    }

  }
}
```



unionSet(3,4)

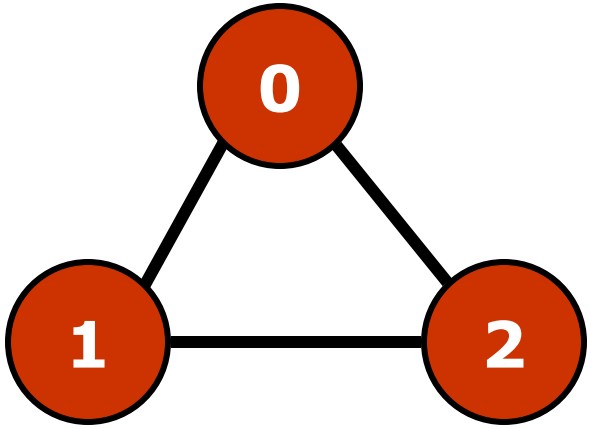
Constructor, UnionFind(N)

```
class UnionFind {
  public int[] p;
  public int[] rank;

  public UnionFind(int N) {
    p = new int[N];
    rank = new int[N];
    for (int i = 0; i < N; i++) {
      p[i] = i;
      rank[i] = 0;
    }
  }

  // ... other methods in the previous slides
}
```

UnionFind(5)

How to detect a cycle in graph?

- Consider the following graph which has no self loop. That is, a vertex will not point to itself.
  - Note: Cycle can be detected with DFS, but that would be slow.

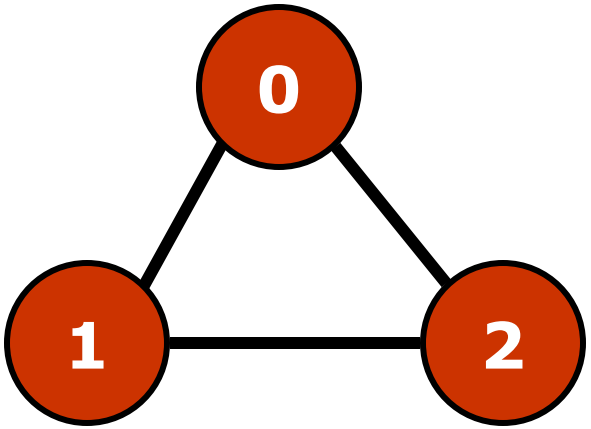


How to detect a cycle in graph?

- As in UFDS, we use an array to keep track of the subsets.

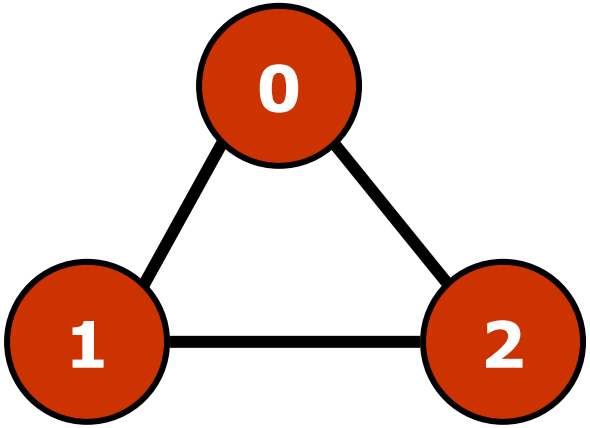
Vertex	**0**	**1**	**2**
Parent	-1	-1	-1

*: -1 means self-loop



How to detect a cycle in graph?

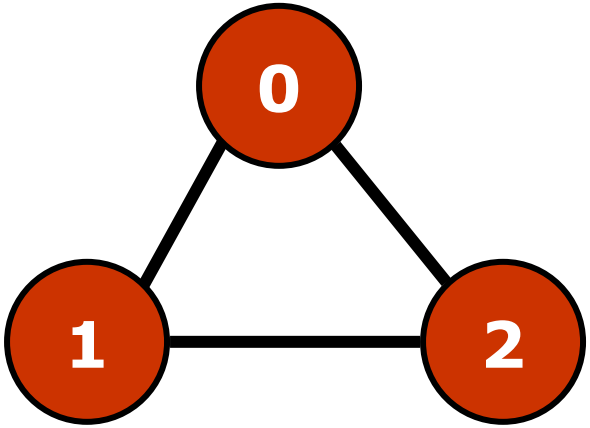
- Process all edges one by one
- Edge 0-1, the two vertices are in different subsets, we union them. Make 1 the parent of 0
- 1 is now the representative of subset {0,1}



Vertex	**0**	**1**	**2**
Parent	1	-1	-1

How to detect a cycle in graph?

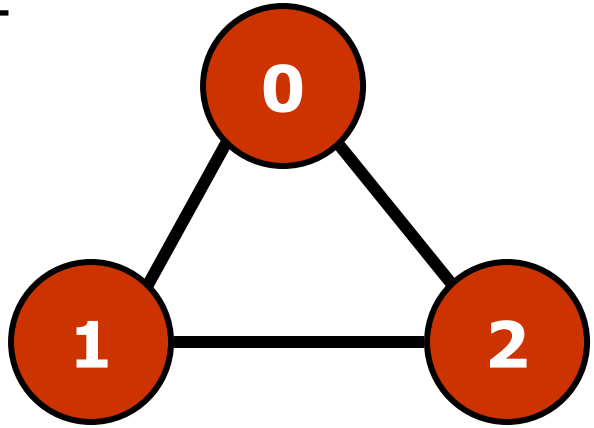
- Edge 1-2, the two vertices are in different subsets, we union them. Make 2 the parent of 1.
- 2 is not the representative of subset {0,1,2}



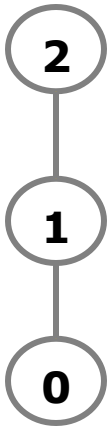
Vertex	**0**	**1**	**2**
Parent	1	2	-1

How to detect a cycle in graph?

- Edge 0-2, 0 is in subset 2 and 2 is also in subset 2.
- Hence, including this edge forms a **cycle**.



Vertex	**0**	**1**	**2**
Parent	1	2	-1



UFDS – Summary

These are the basics … we will not go into further details

- UFDS operations runs in just O(α(N)) if UFDS is implemented with both "union-by-rank" and "path-compression" heuristics
  - α(N) is called the **inverse Ackermann** function
    - This function grows very slowly
    - You can assume it is "constant", i.e., O(1) for practical values of N ($\leq$ 1M)
    - The analysis is quite hard and not for CS2040 level
- Note that UFDS is a static DS since we cannot add new items to the sets in the UFDS after it is created

Further References:

- **Introductions to Algorithms**, p505-509 in 2nd ed, ch 21.3
- **CP3**, Section 2.4.2 (UFDS) and 4.3.2 (MST, Kruskal's)
- **Algorithm Design**, p151-157, ch 4.6
- https://visualgo.net/en/ufds

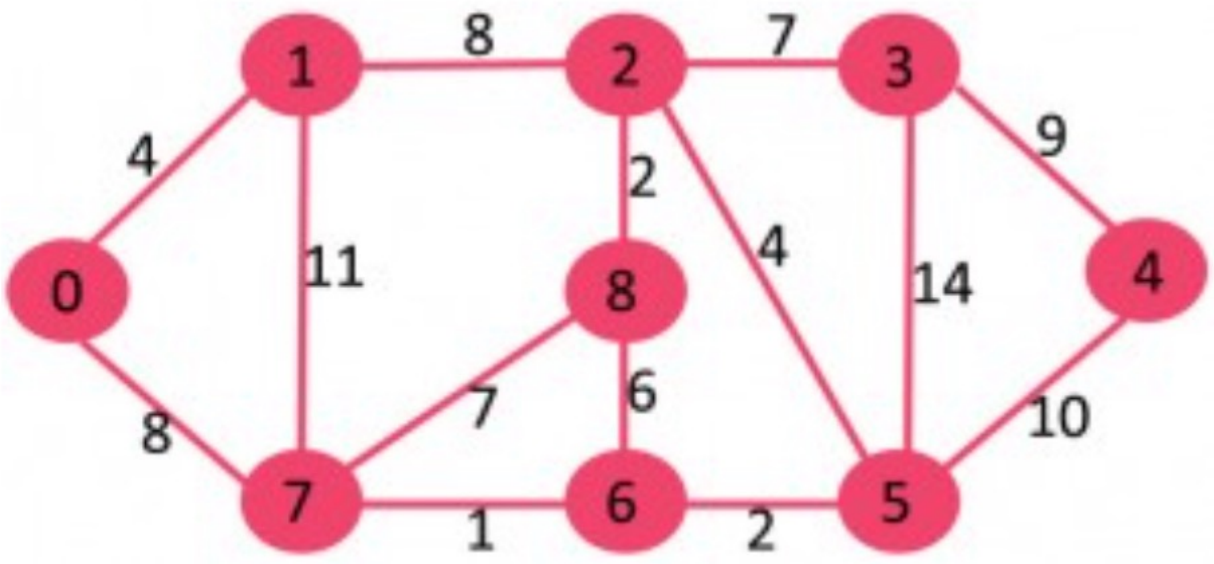
Kruskal's algorithm

1. Sort all the edges in non-decreasing order of their weight.
2. Pick the smallest edge. Check if it forms a cycle with the spanning tree formed so far. If cycle is not formed, include this edge. Else, discard it.
3. Repeat step#2 until there are (V-1) edges in the spanning tree.

- The step #2 uses Union-Find algorithm to detect cycles.

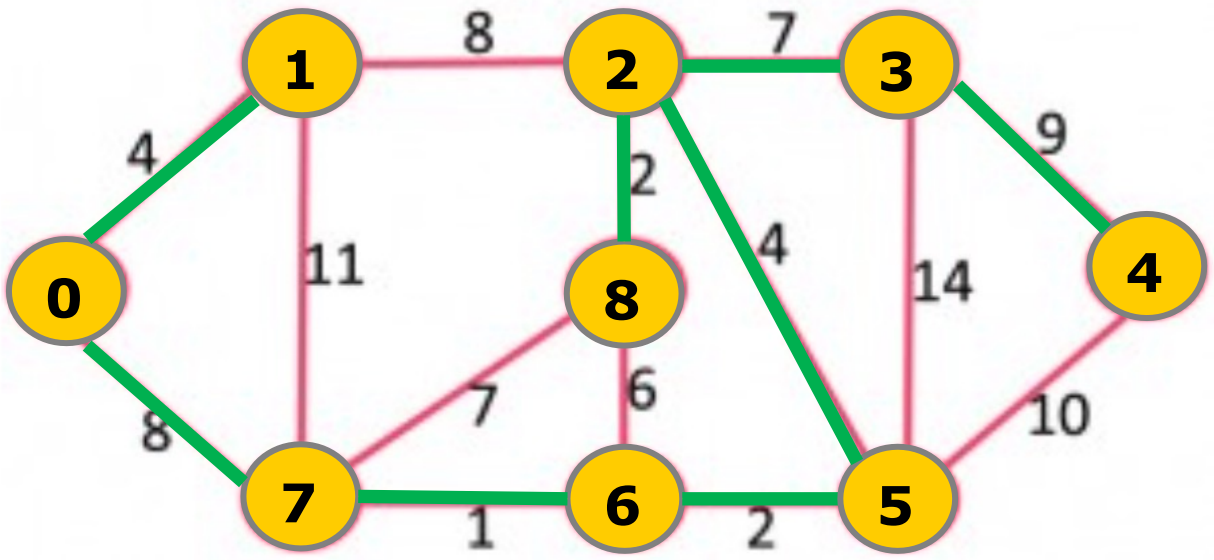
Example

Wt	1	2	2	4	4	6	7	7	8	8	9	10	11	14
src	7	8	6	0	2	8	2	7	0	1	3	5	1	3
dest	6	2	5	1	5	6	3	8	7	2	4	4	7	5



Example

Wt	1	2	2	4	4	6	7	7	8	8	9	10	11	14
src	7	8	6	0	2	8	2	7	0	1	3	5	1	3
dest	6	2	5	1	5	6	3	8	7	2	4	4	7	5



- The graph contains 9 vertices and 14 edges. So, the minimum spanning tree formed will be having (9 – 1) = 8 edges.

Complexity

- Sorting of edges takes O(E log E) time
- Iterating through all edges and apply union-find algorithm takes at most O(log V) time
- Overall complexity is O(E log E + E log V)
- As E can be at most $|V^2|$,
  O(log E) = O(log V)
- therefore, the overall complexity is
  O(E log E) or O(E log V)

End of Part 2

Minimum Spanning Tree

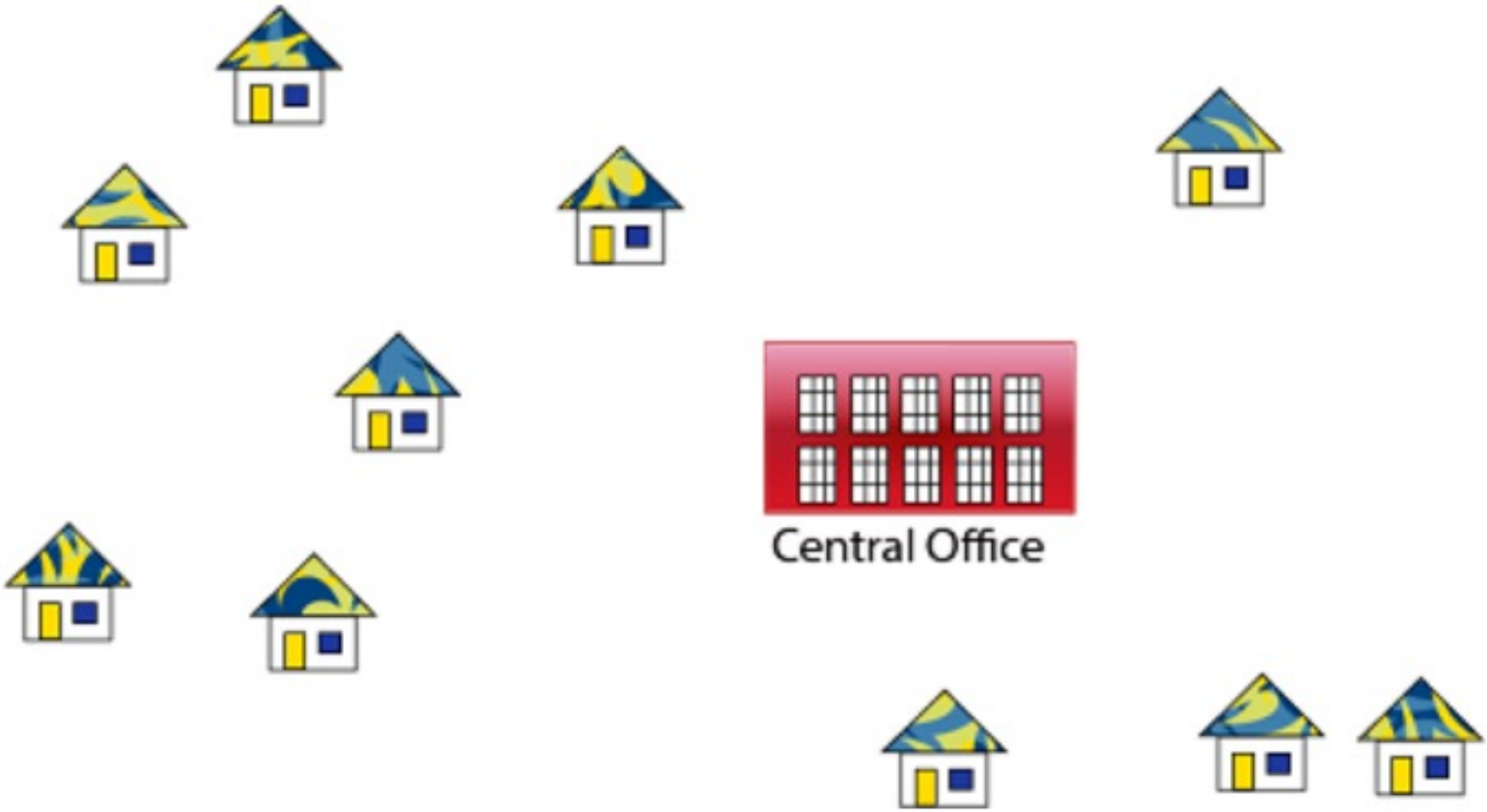
- Minimum Spanning Tree (MST) problem: Given connected graph G with positive edge weights, find a **min weight set** of edges that connects **all of the vertices**.

- *How many edges does a minimum spanning tree have?*
  A minimum spanning tree has (V – 1) edges where V is the number of vertices in the given graph.

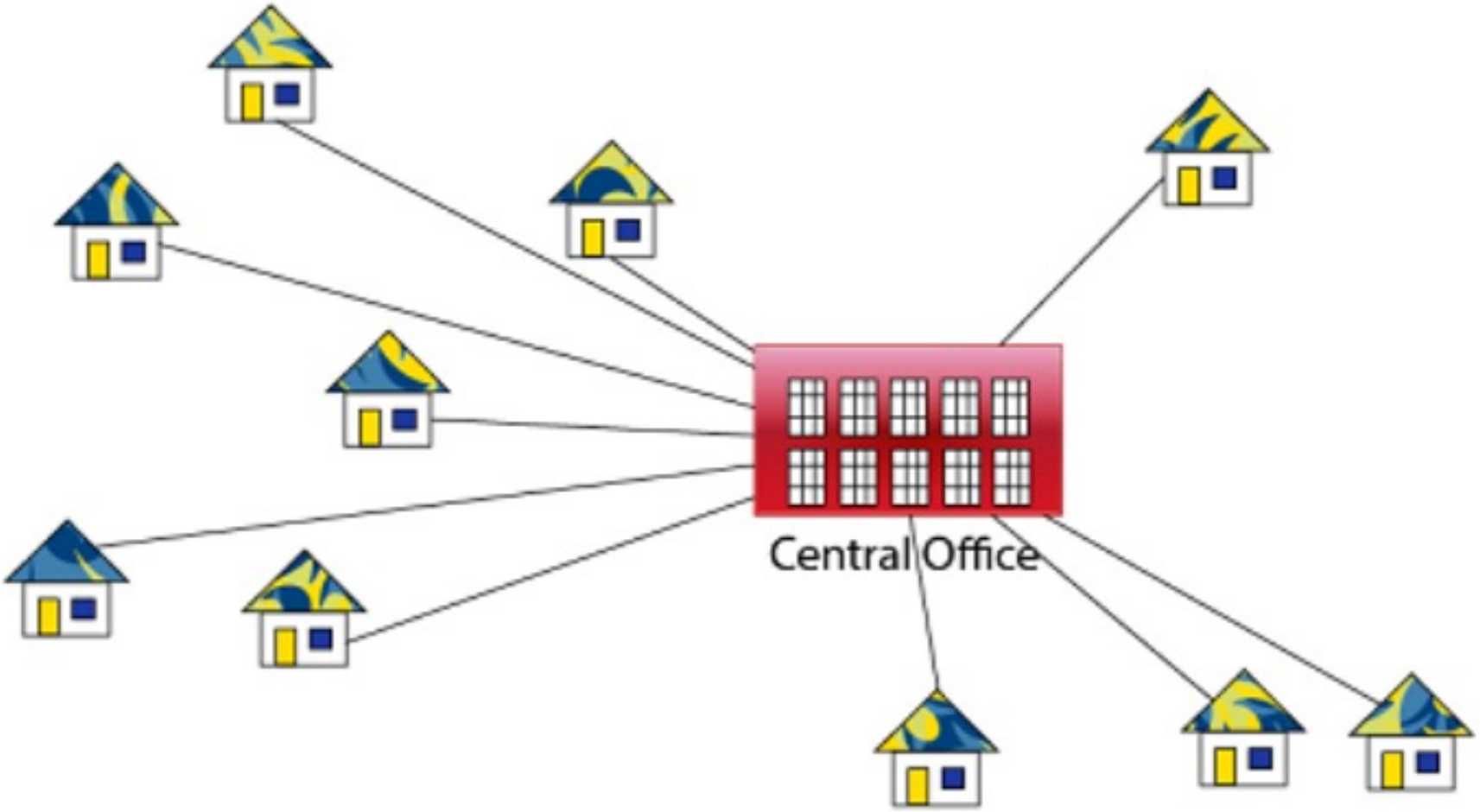
Applications

- **Network design**
  - *telephone*
  - *electrical*
  - *hydraulic*
  - *TV cable*
  - *computer*
  - *road*

For Example, Problem laying Telephone Wire.

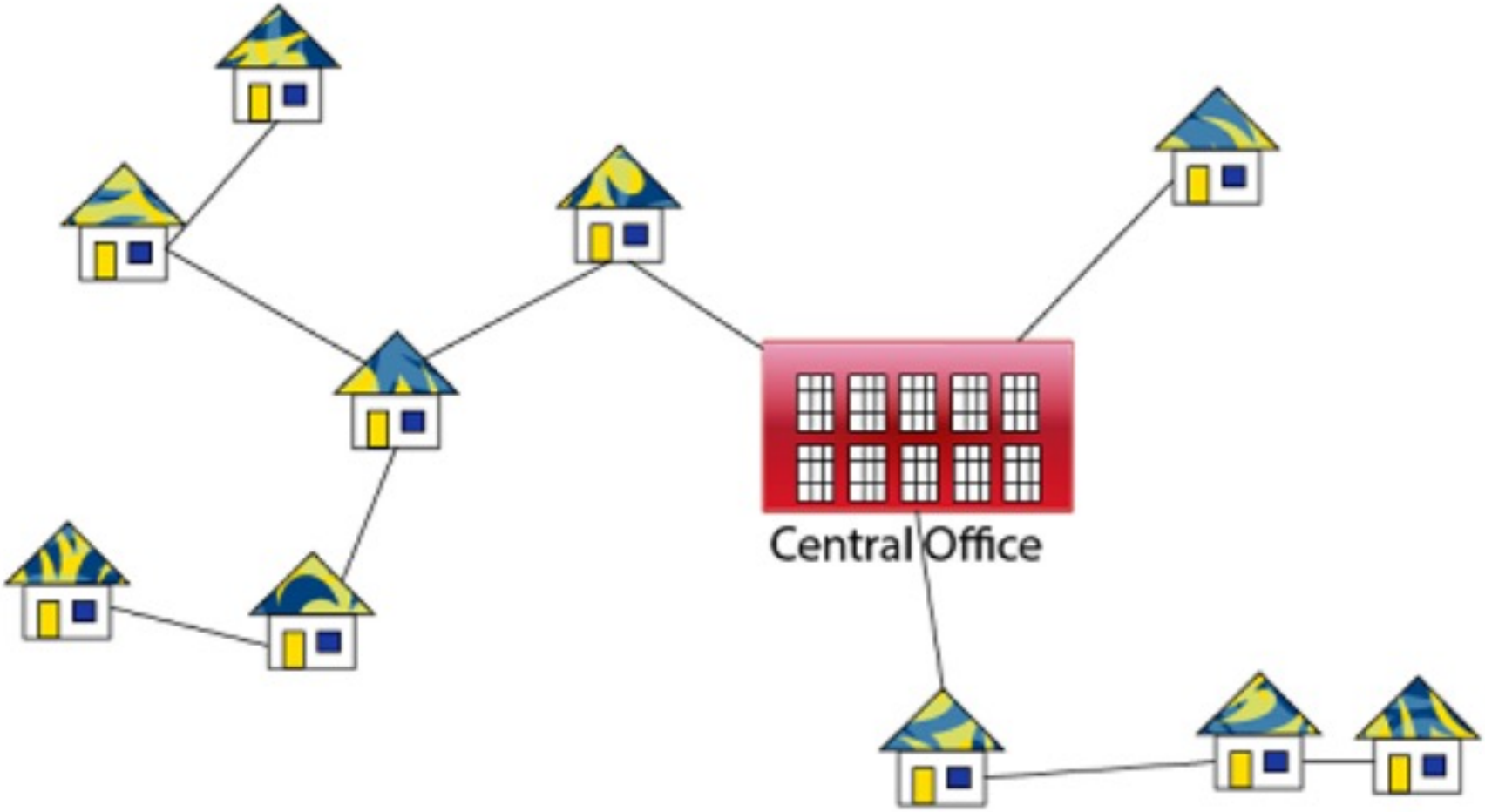


Wiring : Naive Approach



Expensive!

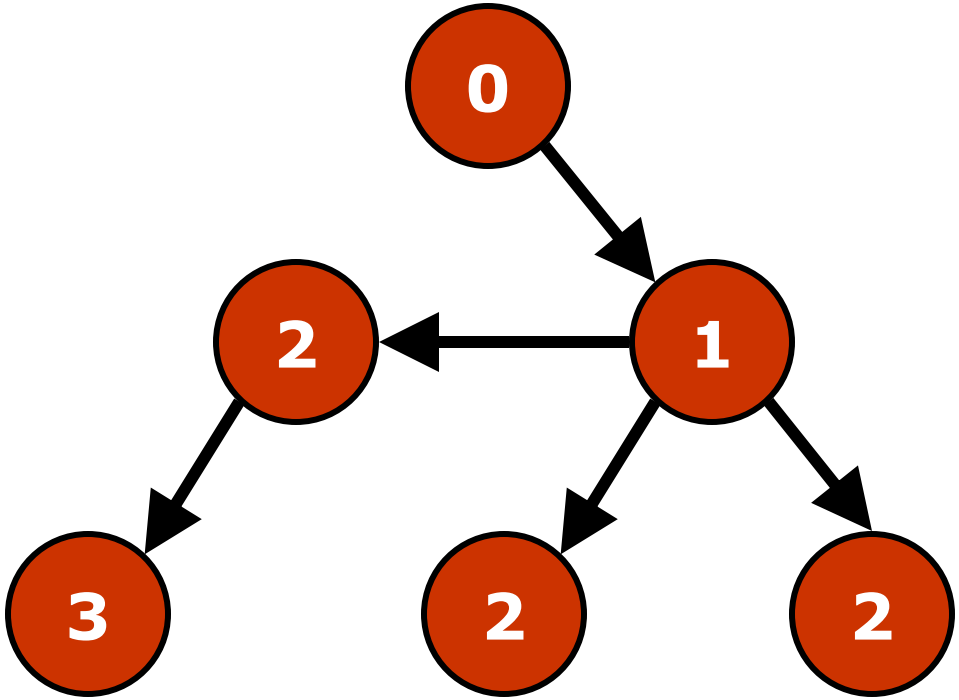
Wiring : Better Approach



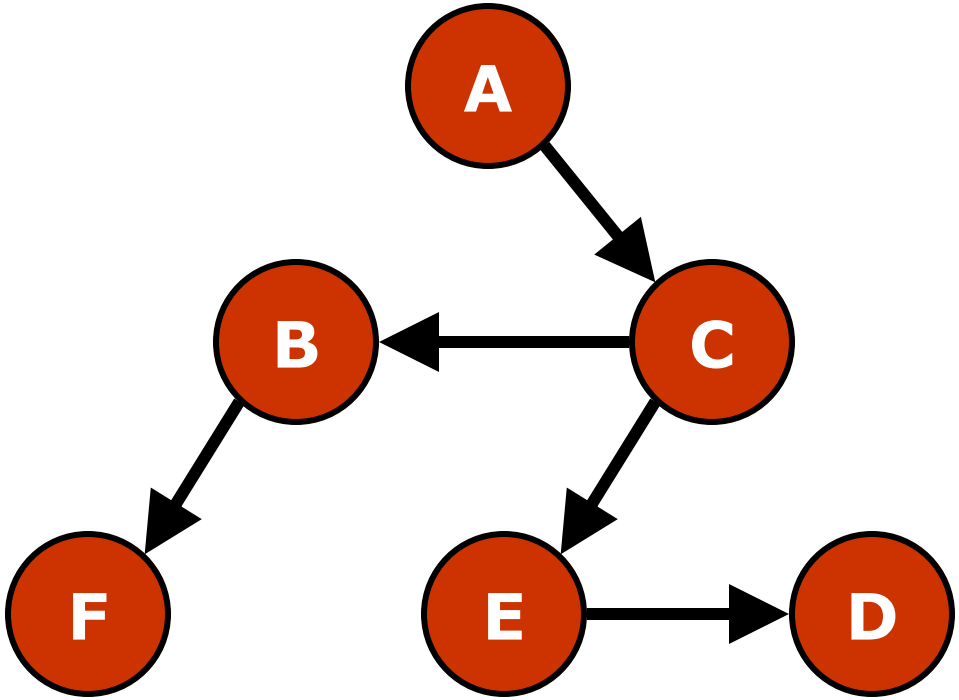
Minimize the total length of wire connecting the customers

Breadth-first search

BFS Tree



Depth-first search



Two MST algorithms

- Kruskal's Algorithm
- **Prim's Algorithm**
- Both are Greedy Algorithm.
- The Greedy Choice is to put the smallest weight edge that does not become a cycle in the MST constructed so far.

Gordon Gekko in 1987 film *Wall Street*:
"Greed, for lack of a better word, is good."

Prim's Algorithm

- Start with an empty spanning tree.
- Maintain two sets of vertices.
  - 1st set contains the vertices in the MST
  - 2nd set contains vertices not yet included
- At every step, consider all the edges that connect the two sets, and pick the minimum weight edge from these edges.
- Move the other endpoint of the edge to the set containing the MST.

Prim's Algorithm

- Definition:
  A group of edges that connects two sets of vertices in a graph is called a graph cut.

- So at every step of Prim's algorithm, find a cut of the two sets which is the minimum weight edge from the cut and include this vertex into the MST.

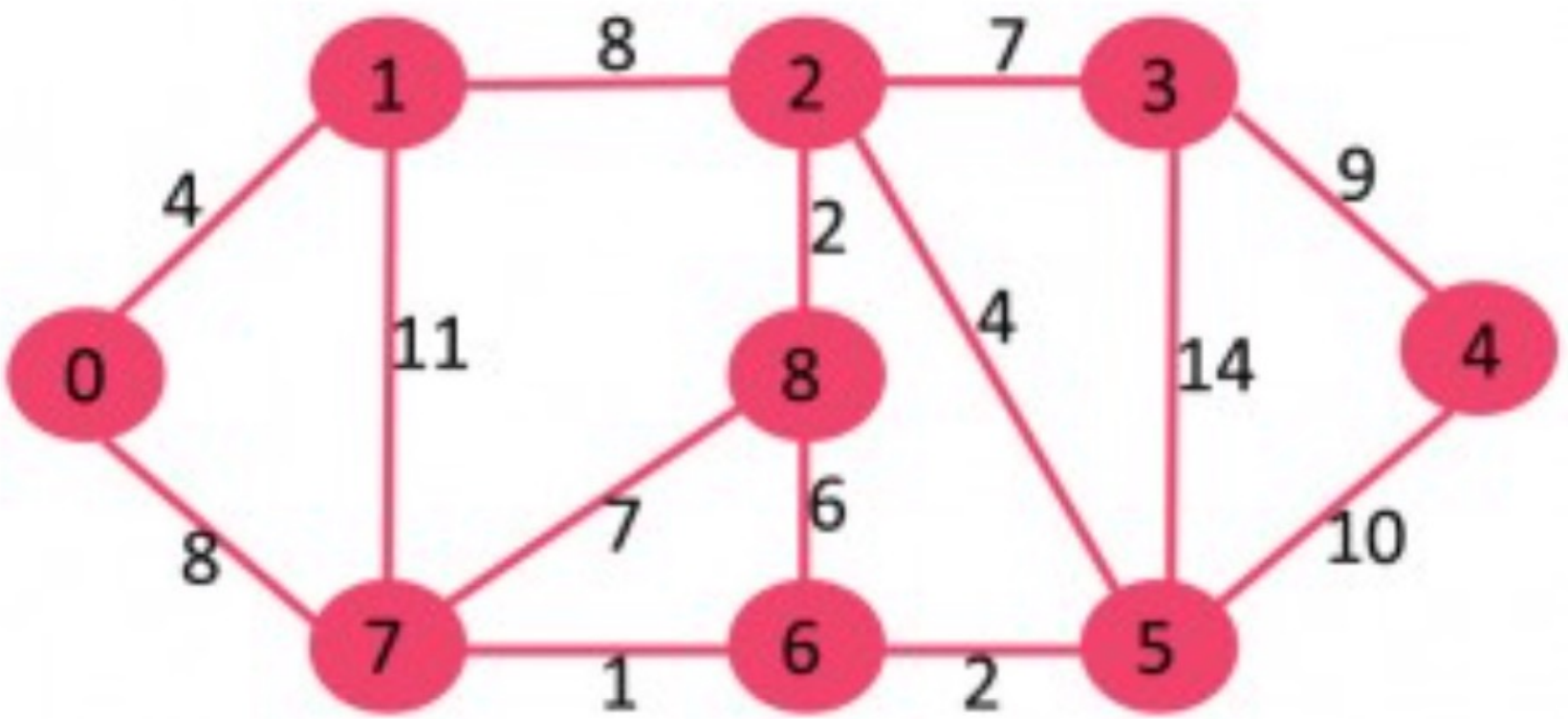
Prim's Algorithm

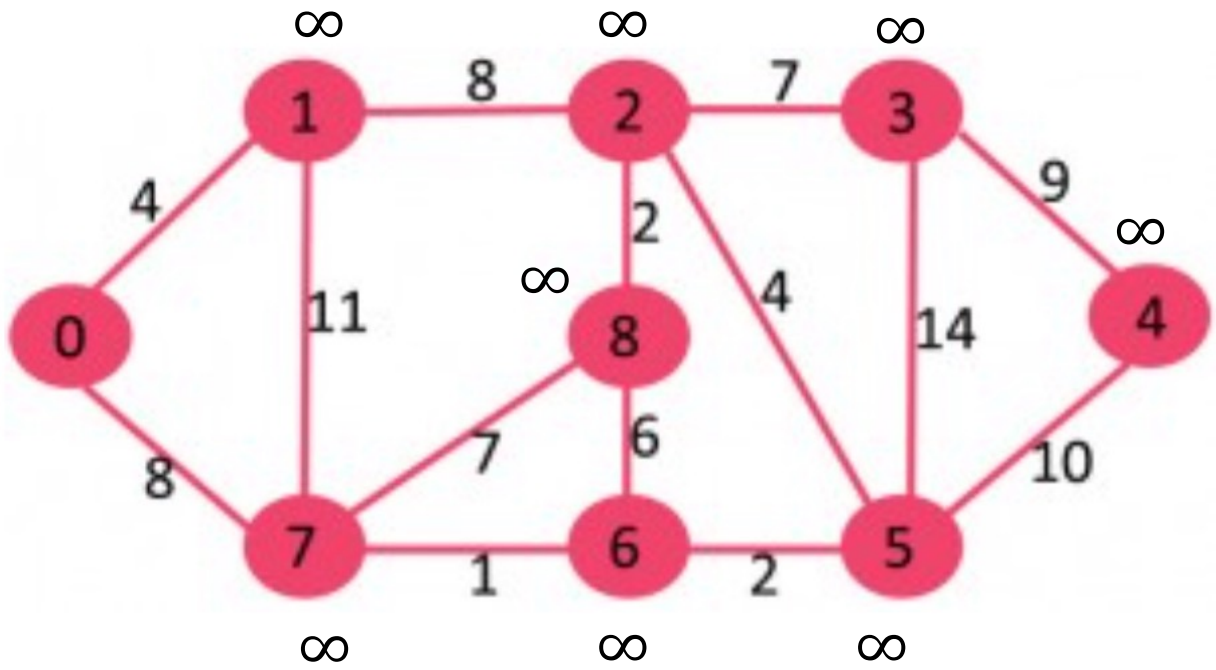
1. Create a set *mstSet* that keeps track of vertices already included in the MST.
2. Assign a key value to all vertices in the graph.
   - the first vertex has key value 0
   - the rest have key value ∞
3. While *mstSet* does not include all vertices, do the following:

Prim's Algorithm

a) Pick a vertex *u* which is not in *mstSet* and has minimum key value.
b) Include *u* into *mstSet*.
c) Update key values of all adjacent vertices of *u*.
   - For every adjacent vertex *v*, if weight of edge *u*–*v* is less than the previous key value of *v*, update the key value as weight *u*–*v*.

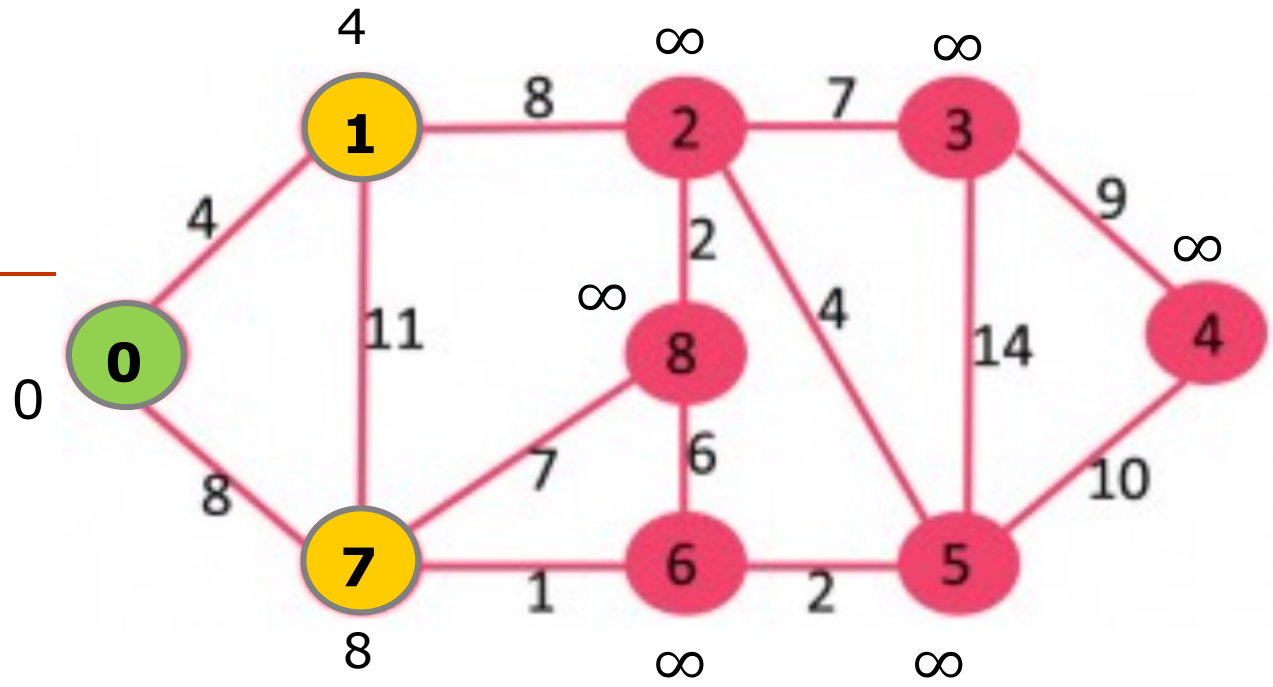
Example





mstSet = {}

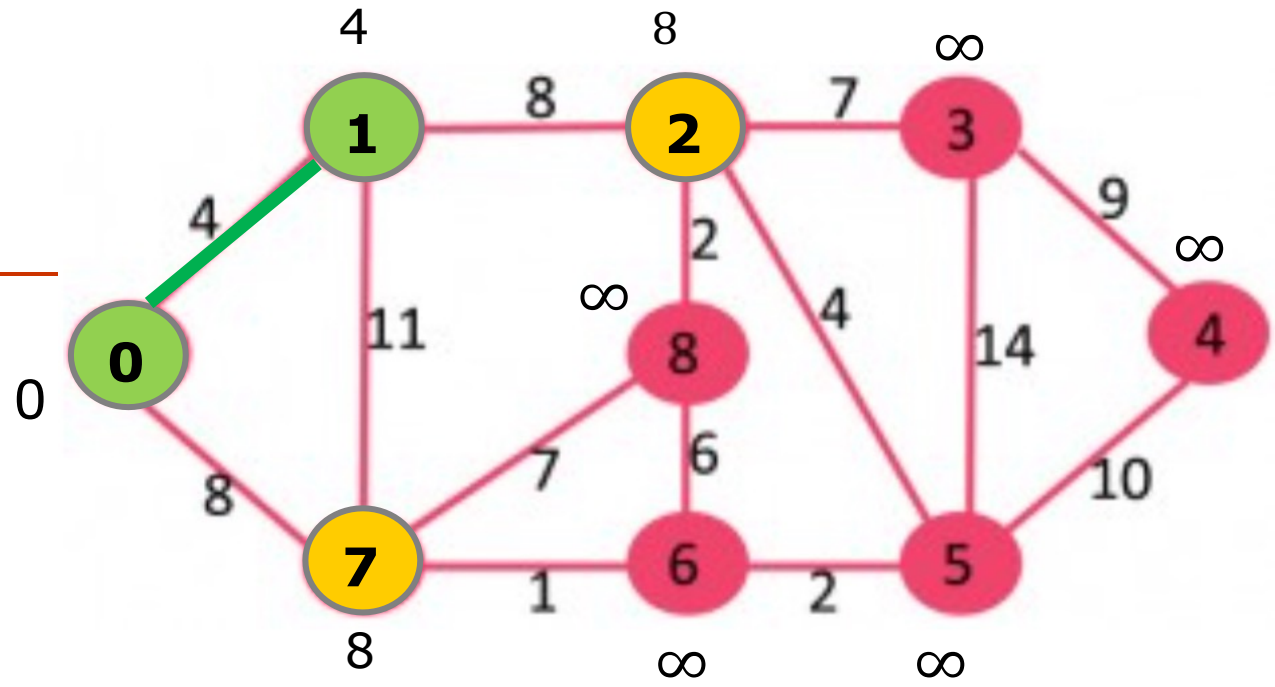
Key	**0**	∞	∞	∞	∞	∞	∞	∞	∞
Vertex	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Parent	**-1**								



Pick vertex 0, as it has the minimum key value; Update vertex 1 and 7

mstSet = {0}

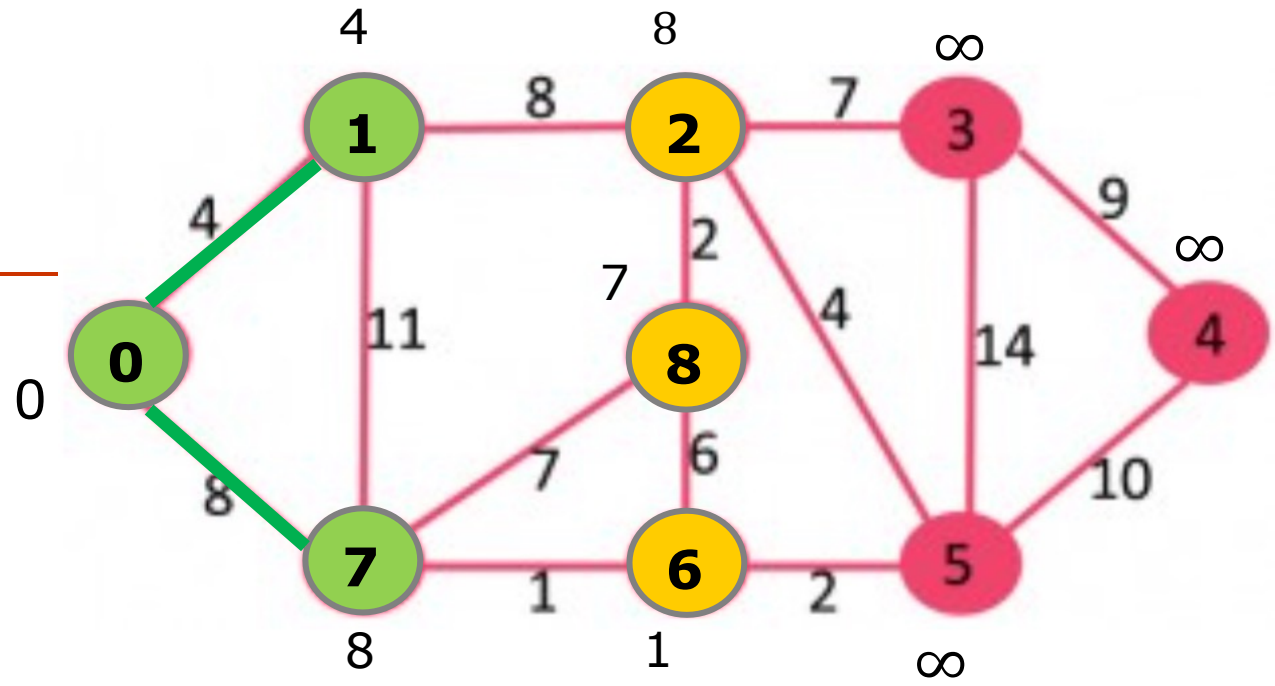
Key	**0**	**4**	∞	∞	∞	∞	∞	**8**	∞
Vertex	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Parent	**-1**	**0**						**0**	



Pick vertex 1, as it has the minimum key value; Update vertex 2

mstSet = {0,1}

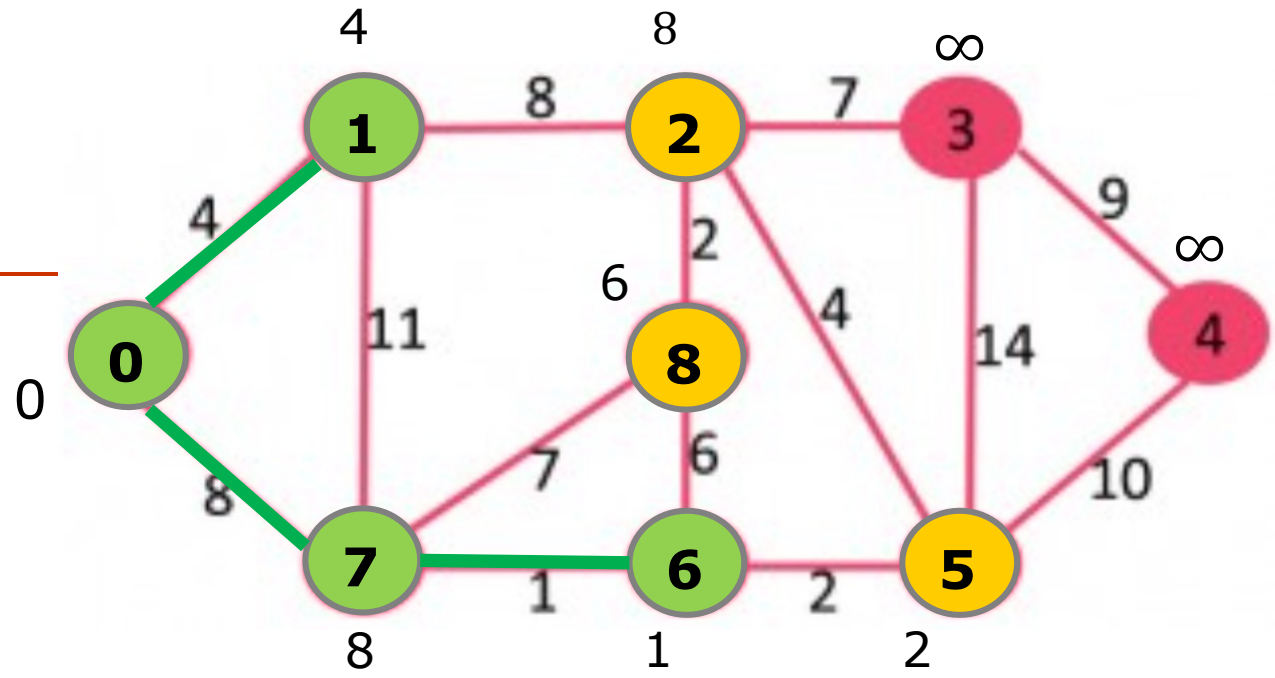
Key	**0**	**4**	**8**	∞	∞	∞	∞	**8**	∞
Vertex	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Parent	**-1**	**0**	**1**					**0**	



Pick vertex 7, as it has the minimum key value;
Update vertex 6 and 8

mstSet = {0,1,7}

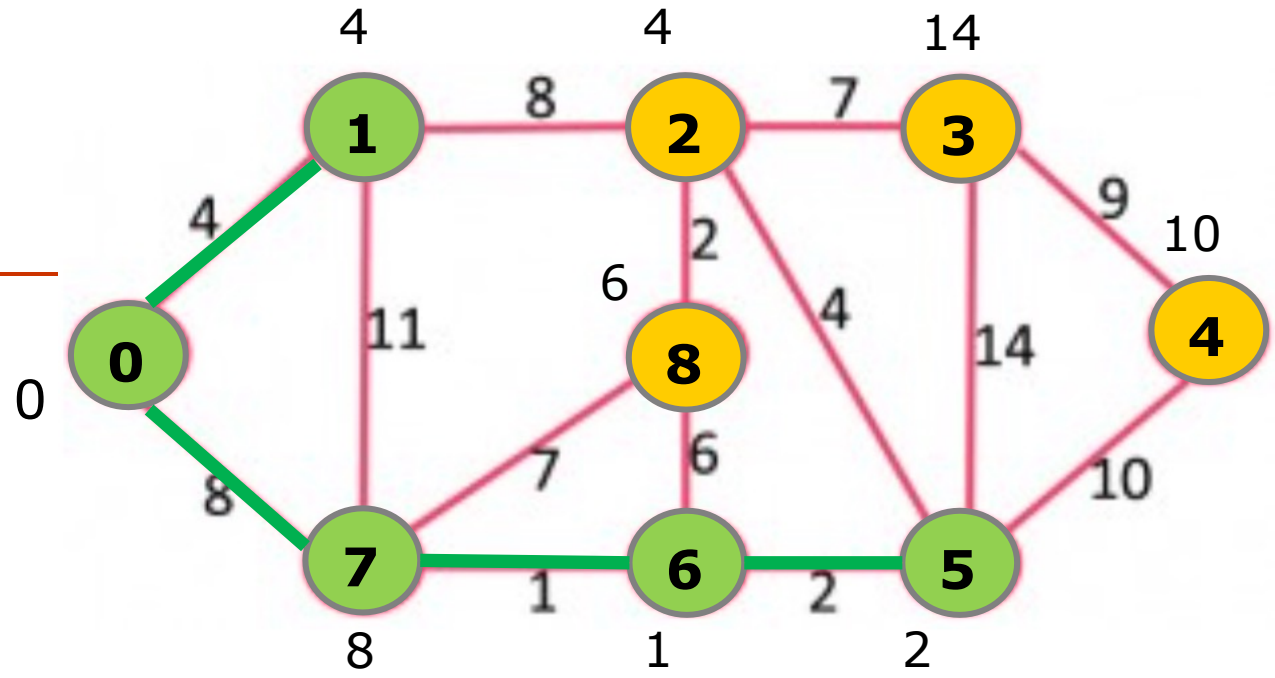
Key	**0**	**4**	**8**	∞	∞	∞	**1**	**8**	**7**
Vertex	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Parent	**-1**	**0**	**1**				**7**	**0**	**7**



Pick vertex 6, as it has the minimum key value;
Update vertex 5 and 8

mstSet = {0,1,7,6}

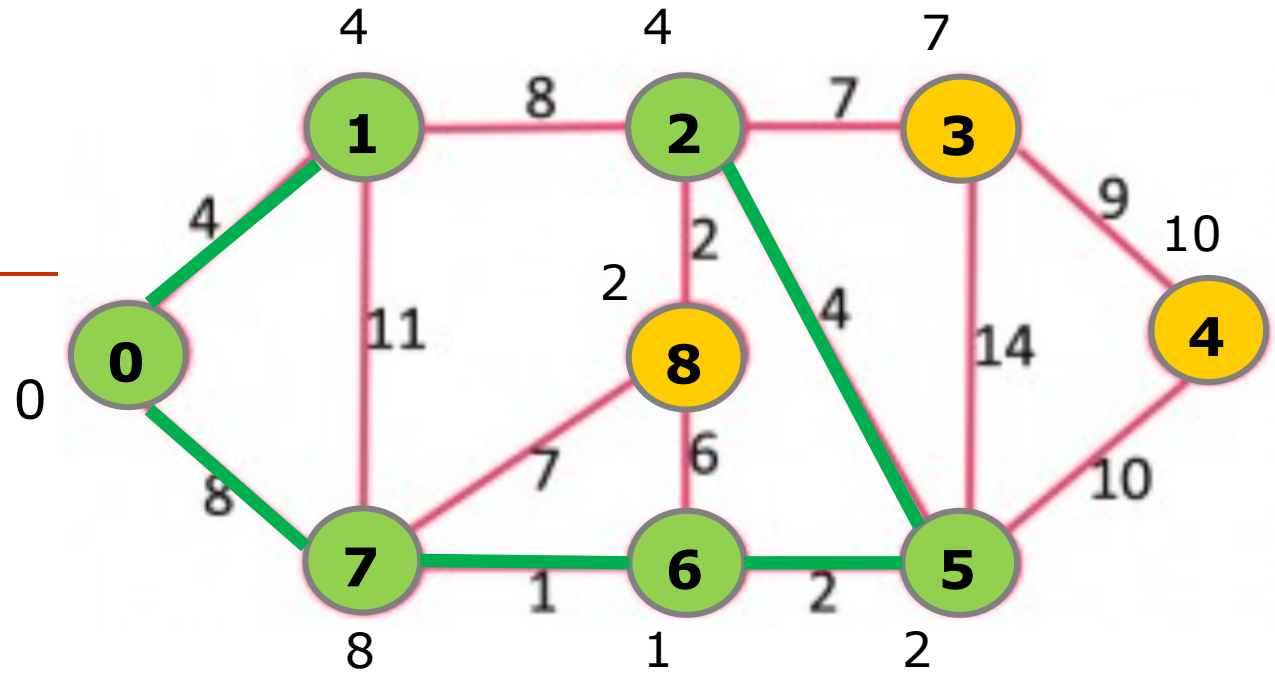
Key	**0**	**4**	**8**	∞	∞	**2**	**1**	**8**	**6**
Vertex	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Parent	**-1**	**0**	**1**			**6**	**7**	**0**	**6**



Pick vertex 5, as it has the minimum key value;
Update vertex 2, 3 and 4

mstSet = {0,1,7,6,5}

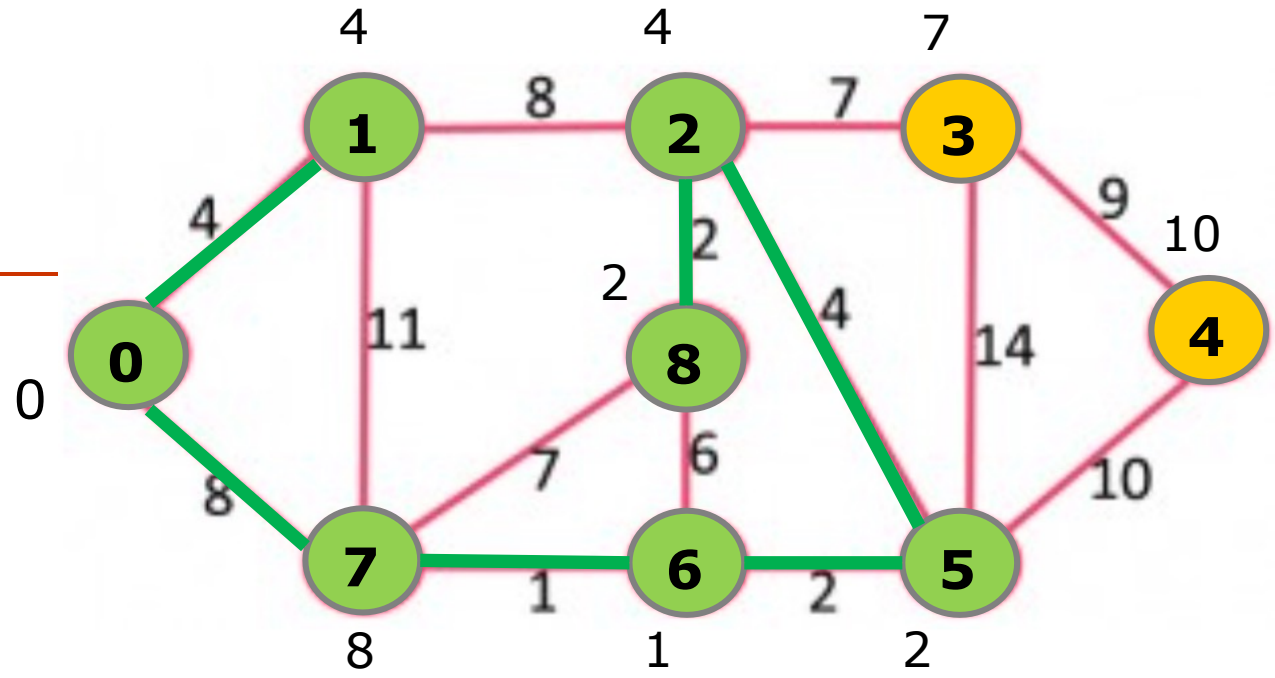
Key	**0**	**4**	**4**	**14**	**10**	**2**	**1**	**8**	**6**
Vertex	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Parent	**-1**	**0**	**5**	**5**	**5**	**6**	**7**	**0**	**6**



Pick vertex 2, as it has the minimum key value;
Update vertex 3 and 8

mstSet = {0,1,7,6,5,2}

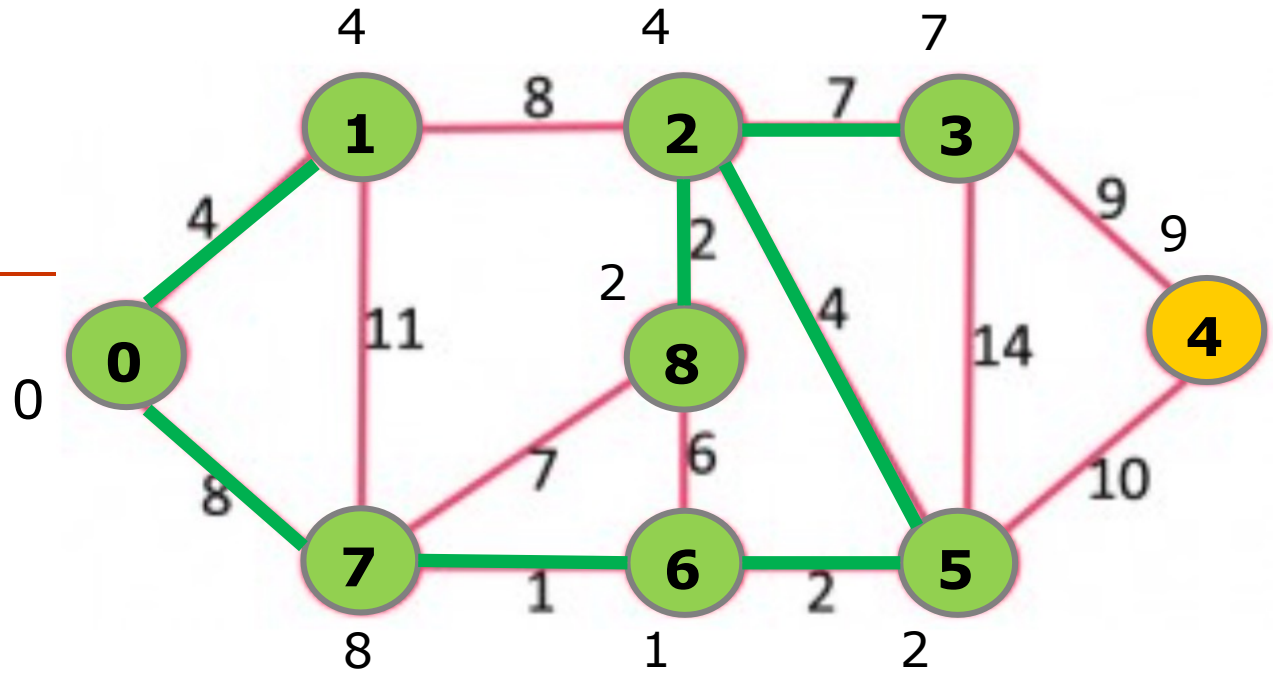
Key	**0**	**4**	**4**	**7**	**10**	**2**	**1**	**8**	**2**
Vertex	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Parent	**-1**	**0**	**5**	**2**	**5**	**6**	**7**	**0**	**2**



Pick vertex 8, as it has the minimum key value;
Update no vertex

mstSet = {0,1,7,6,5,2,8}

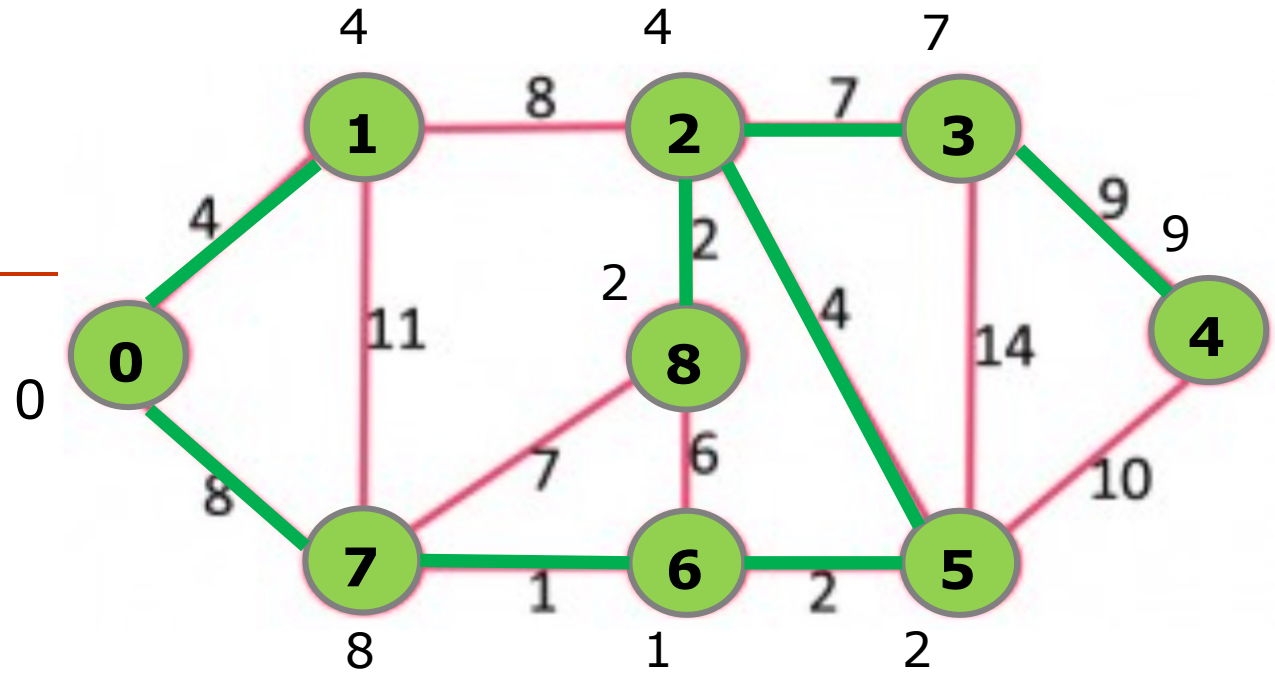
Key	**0**	**4**	**4**	**7**	**10**	**2**	**1**	**8**	**2**
Vertex	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Parent	**-1**	**0**	**5**	**2**	**5**	**6**	**7**	**0**	**2**



Pick vertex 3, as it has the minimum key value;
Update no vertex

mstSet = {0,1,7,6,5,2,8,3}

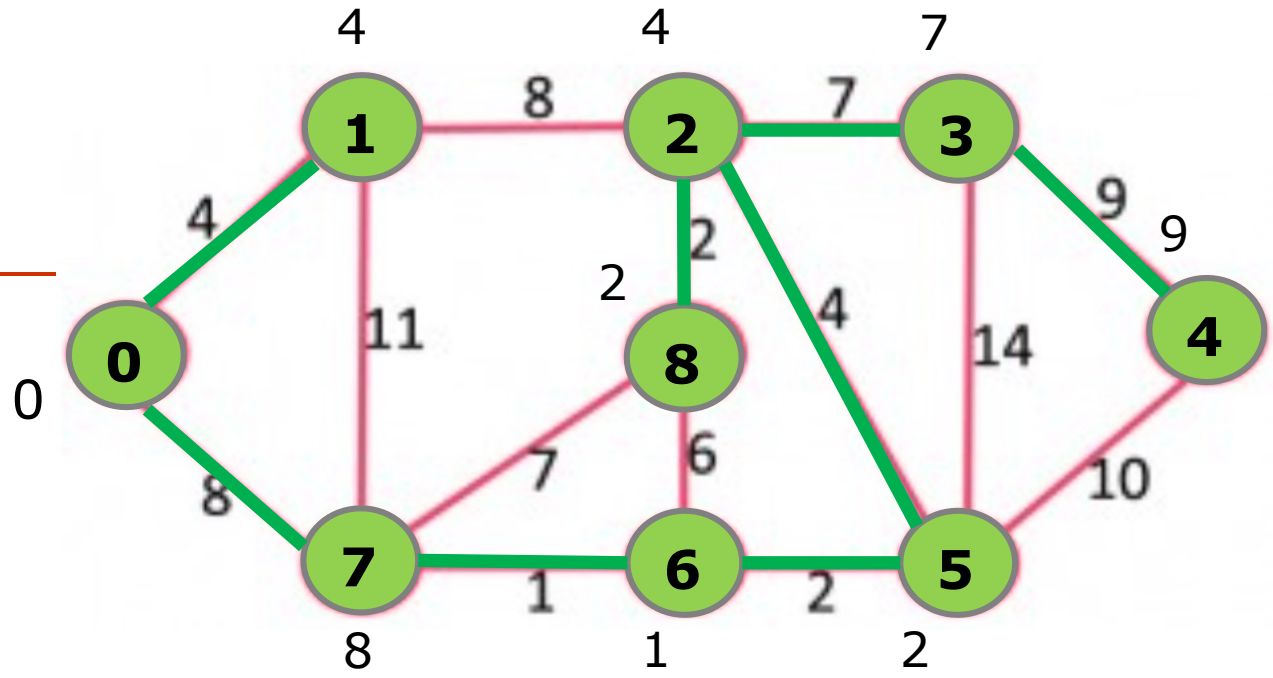
Key	**0**	**4**	**4**	**7**	**9**	**2**	**1**	**8**	**2**
Vertex	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Parent	**-1**	**0**	**5**	**2**	**3**	**6**	**7**	**0**	**2**



Pick the last vertex 4

mstSet = {0,1,7,6,5,2,8,3,4}

Key	**0**	**4**	**4**	**7**	**9**	**2**	**1**	**8**	**2**
Vertex	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Parent	**-1**	**0**	**5**	**2**	**3**	**6**	**7**	**0**	**2**



The parent array is the output which is used to show the constructed MST

mstSet = {0,1,7,6,5,2,8,3,4}; the MST may not be unique

Key	**0**	**4**	**4**	**7**	**9**	**2**	**1**	**8**	**2**
Vertex	**0**	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**
Parent	**-1**	**0**	**5**	**2**	**3**	**6**	**7**	**0**	**2**

Time Complexity

- Complexity of Prim's algorithm is $O(V^2)$ if the input graph is represented using an adjacency list
- With the help of a binary heap, the time complexity can be reduced to O(E log V) (same as Kruskal's)

Comparisons

- SSSP: Dijkstra
  - Shortest path from one source to all other nodes
  - Optimization value (distance) is measured from **source** to a node

- MST: Prim and Kruskal
  - MST: minimize sum of tree edge weights
  - Optimization value is **single edge** weight
    - Prim: Start with small tree (single node) and grow it
    - Relaxation
  - Kruskal
    - Create a forest of trees, grow and connect them

The End

(End of Part 3)