CS2020 – Data Structures and Algorithms Accelerated

Lecture 15 – Minimum Spanning Tree

stevenhalim@gmail.com

NUS National University of Singapore

School of Computing

Outline

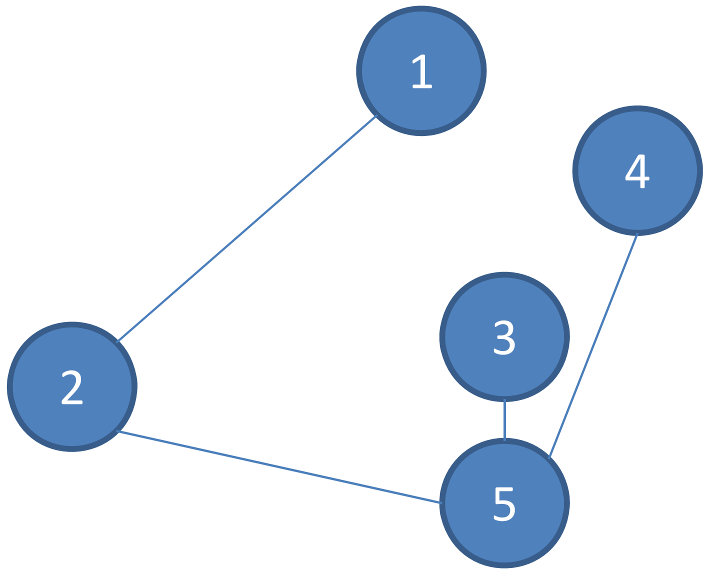
- What are we going to learn in this lecture?
  - Review + PS6 Reminder/PS7 Preview
  - Minimum Spanning Tree (MST)
    - Motivating Example & Some Definitions
  - Algorithms to solve MST
    - Kruskal’s (more detailed)
      - Sorting graph edges based on weight
      - Introduction to greedy algorithm
      - Data structure to prevent cycling
        - Union-Find Disjoint Sets
    - Prim’s (just an overview)
      - Priority Queue (heap!!)

Lecture Review

- Definitions that we have learned before
  - **Tree T**
    - **T** is a **connected graph** that has **V** vertices and **V-1** edges
    - One unique path between any two pair of vertices in **T**
  - **Spanning** Tree **ST** of connected graph **G**
    - **ST** is a tree that spans (covers) every vertices in **G**
    - Recall the **BFS and DFS Spanning Tree**
- Sorting problem & several sorting algorithms
  - Rearrange set of objects so that every pair of objects **(a, b; a < b)** in the final arrangement satisfies that **a** is before **b**

Is This A Tree?

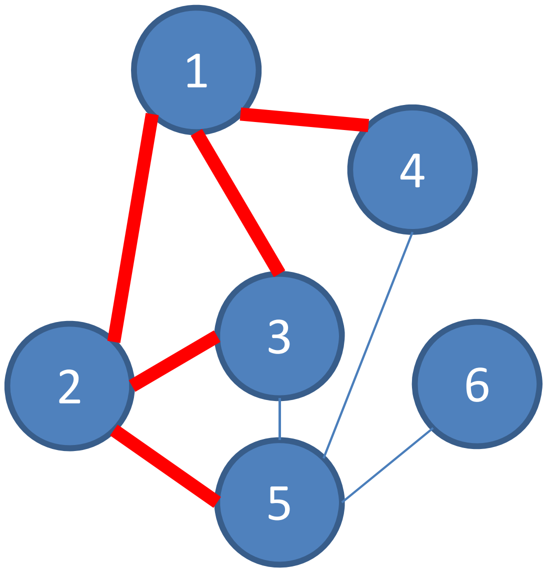
1. Yes, why __________
2. No, why __________



0 of 54

Are the edges highlighted in red part of a spanning tree of the original graph?

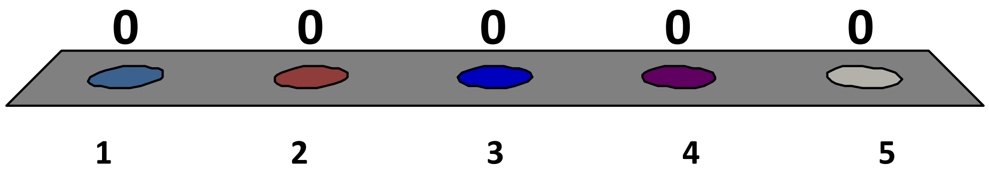
1. Yes, why ___________
2. No, why ___________



0 of 54

Which sorting algorithm is the best?

1. Bubble Sort
2. Insertion Sort
3. Merge Sort
4. Quick Sort
5. Heap Sort, you teach us this one, this must be the best ☺



0 of 54

PS6 Reminder + PS7 Preview

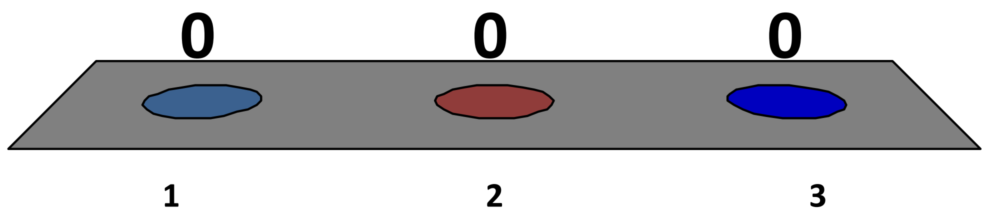
- PS6
  - Deadline is tomorrow, Wed 16 March 2011, 2pm
- PS7
  - There are two (simple?) graph-related programming tasks
    - One is simply a variant of MST problem discussed today
    - One more is a simplest? form of SSSP problem discussed this coming Friday and next Tuesday
  - Deadline for PS7 is Wed 23 March 2011, 2pm

Motivating Example

- Government Project
  - Want to link rural villages with roads
  - The cost to build a road depends on the terrain, etc
  - You only have limited budget
  - How are you going to build the roads?

Last week, many/~20 of you said that you know Prim's/Kruskal's, is it because:

1. You have seen how they work in CS1231
2. You have known these two algorithms (not the implementation) even before entering NUS SoC
3. You have solved/implement several MST problems in some online judges before…



0 of 54

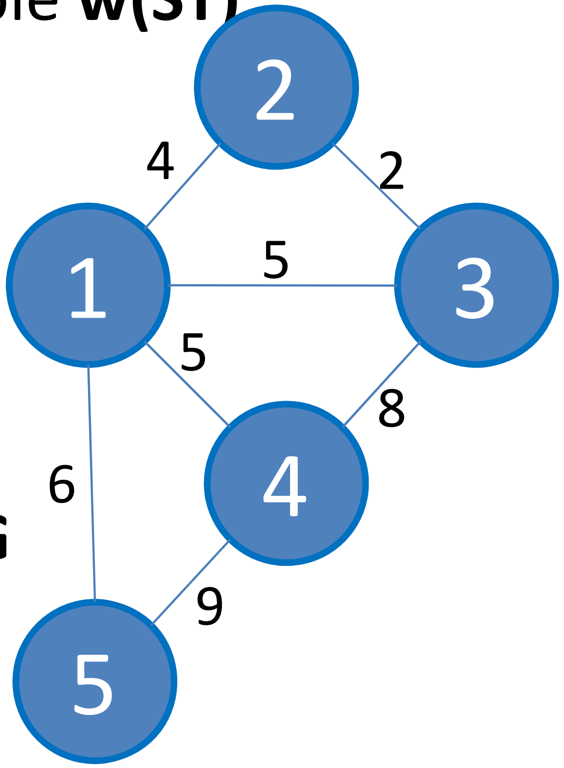
More Definitions (1)

- **Weighted** Graph: **G(V, E), w(u, v): E→R**
  - See below for **w(u, v)**
- Vertex **V** (e.g. street intersections, houses, etc)
- Edge **E** (e.g. streets, roads, avenues, etc)
  - Generally undirected (e.g. bidirectional road, etc)
  - Weighted (e.g. distance, time, toll, etc)
- Weight function **w(u, v): E→R**
  - Sets the weight of edge from **u** to **v**
- **Connected** undirected graph **G**
  - There is a path from any vertex **u** to any other vertex **v** in **G**

More Definitions (2)

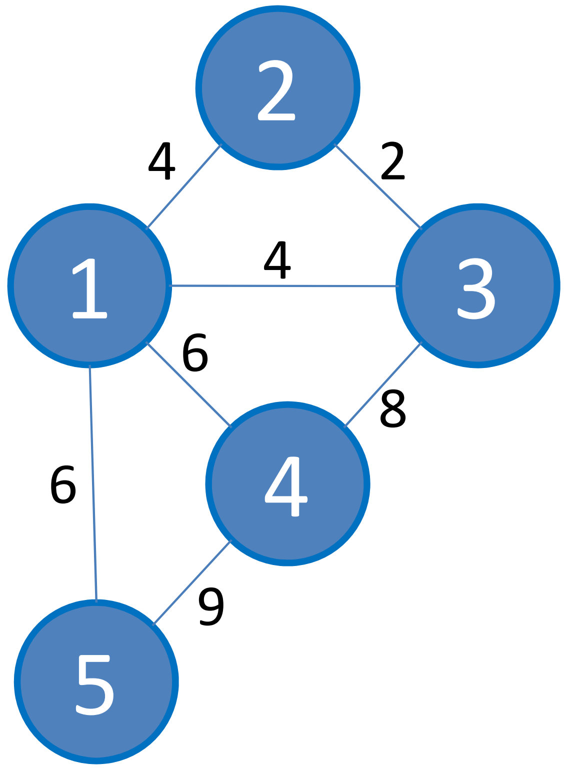
$$w(ST) = \sum_{(u,v)\in ST} w(u,v)$$

- Tree **T** & Spanning Tree **ST** of **G**
  - Let **w(ST)** denotes the total weight of edges in **ST**
- **Minimum Spanning Tree (MST)** of connected weighted graph **G**
  - **MST** of **G** is an **ST** of **G** with minimum possible **w(ST)**
- **The (standard) MST Problem**
  - Input: A connected weighted graph **G(V, E)**
  - Select some edges of **G** such that the graph is still connected, but with min possible total weight
  - Output: Minimum Spanning Tree **(MST)** of **G**

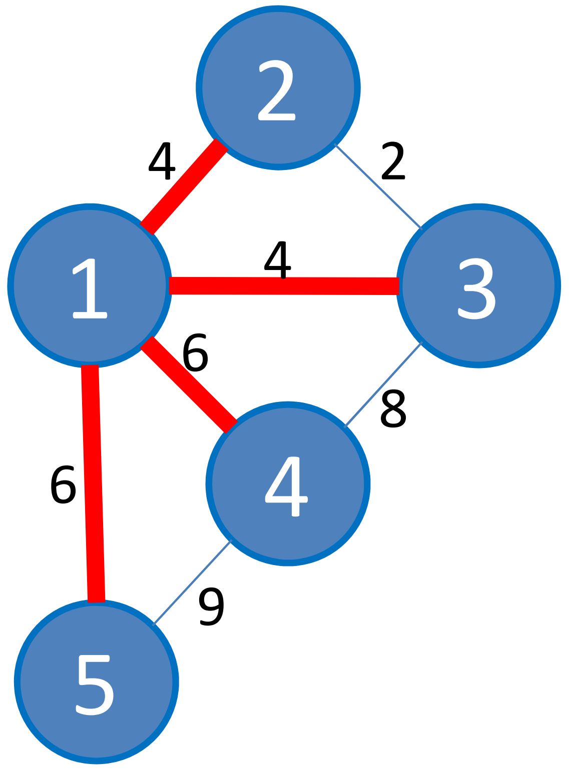


Example

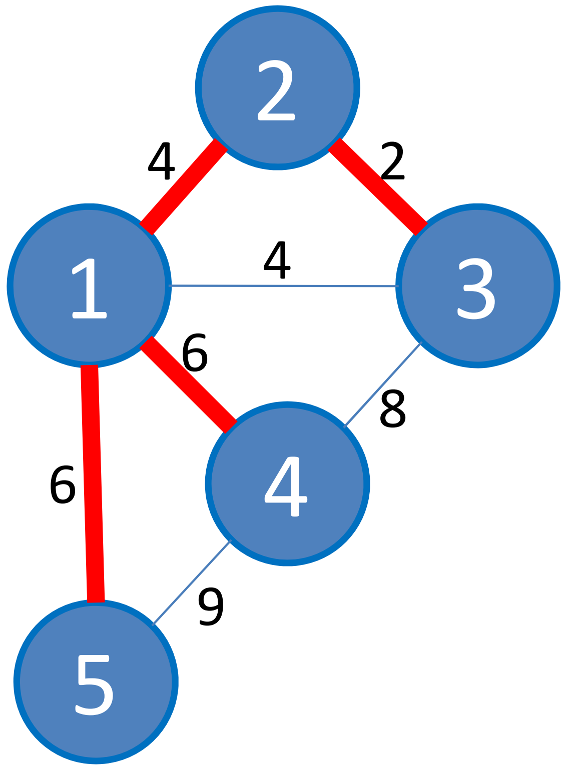
The Original Graph



A Spanning Tree
Cost: 4+4+6+6 = 20

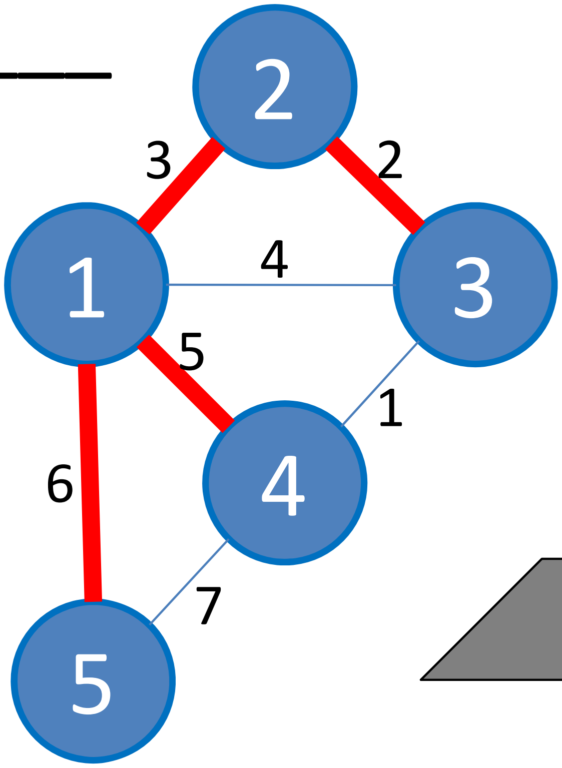


An MST
Cost: 4+6+6+2 = 18



Are the edges highlighted in red part of an MST of the original graph?

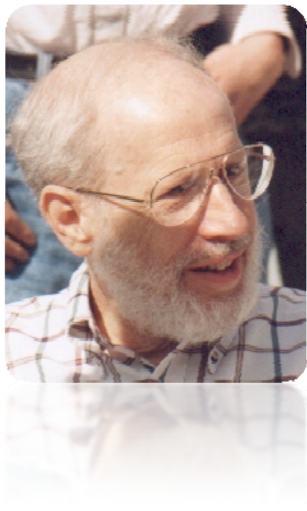
1. Yes
2. No, the correct answer must be ______________



MST Algorithms

- MST is a well-known Computer Science problem
- Several efficient (polynomial) algorithms:
  - Kruskal’s greedy algorithm
  - Jarnik’s/Prim’s greedy algorithm
    (discussed briefly at the end of this lecture)
  - Boruvka’s greedy algorithm (not discussed here)
  - And few other more advanced variants/special cases…

Kruskal’s Algorithm (1)



- Pseudo code (very simple)

```
sort E edges by increasing weight
T ← {}
while there are unprocessed edges left
  pick an unprocessed edge e with min cost
  if adding e to T does not form a cycle
    add e to T
T is an MST
```

- Let’s see how it works first…

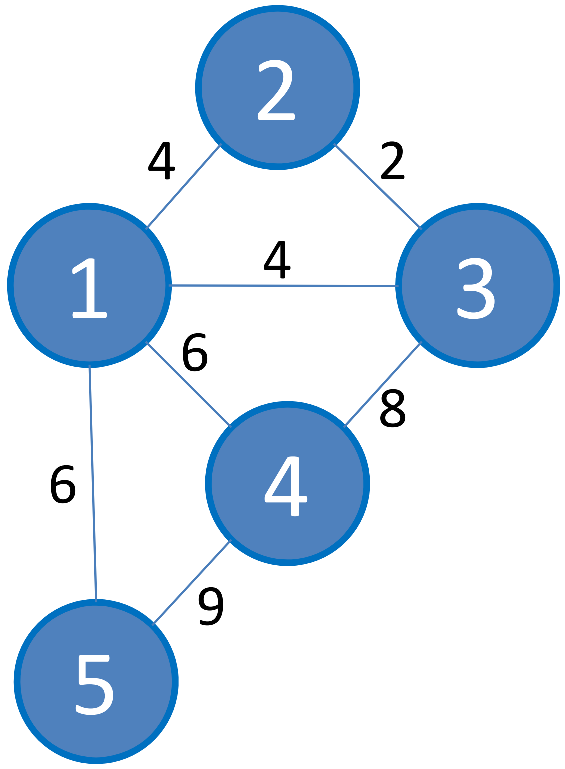
Survey: Do you agree that Kruskal's algorithm is very simple?

1. Yes, the pseudo code looks short
2. No, it cannot be this simple, you must have hidden something from us...

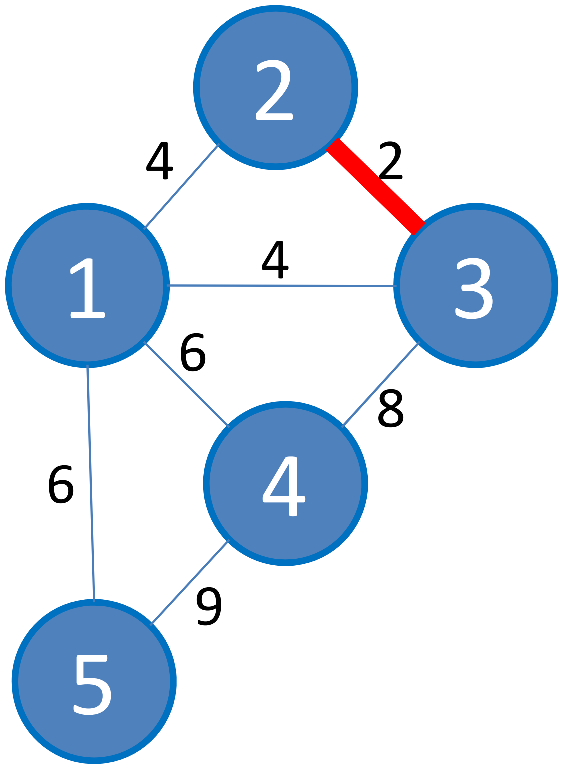
0 of 54

Kruskal's Animation (1)

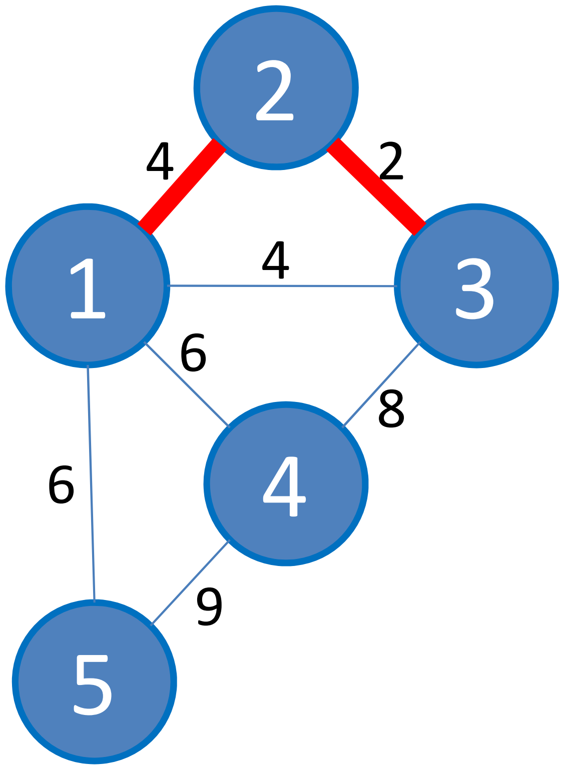
The original graph,
no edge is selected



Connect 2 and 3
As this edge is smallest



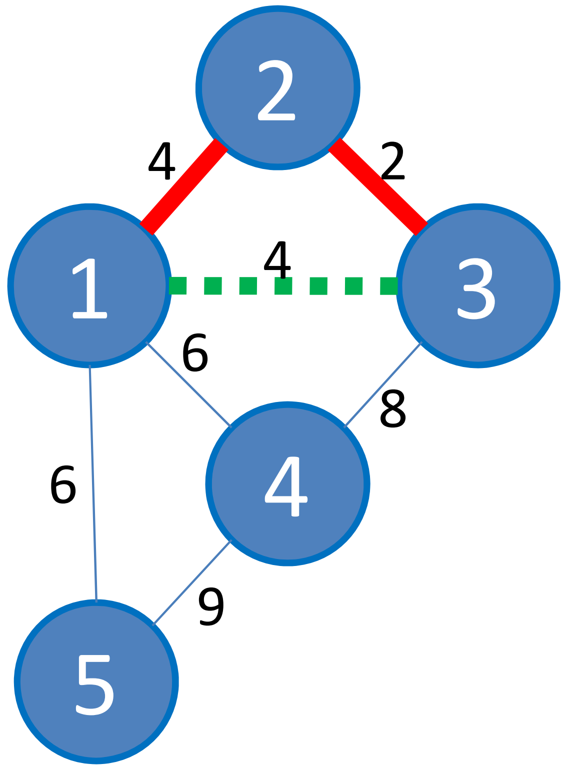
Connect 2 and 1
No cycle is formed



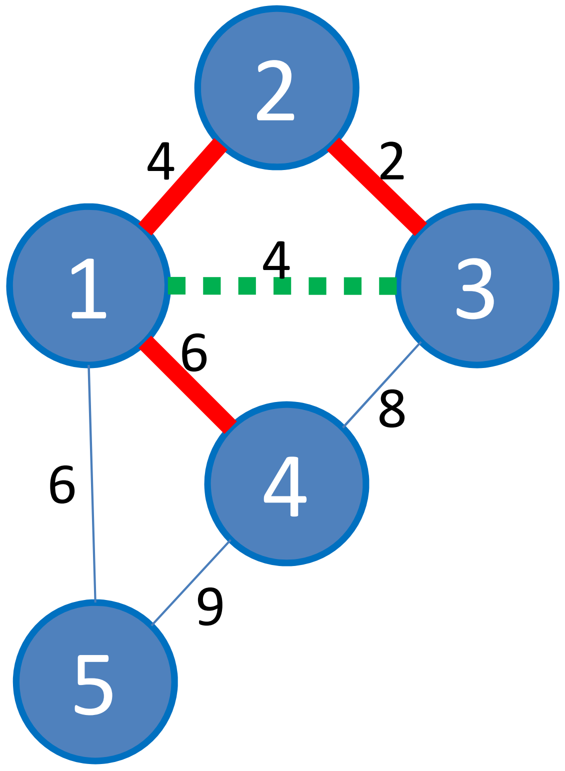
Note: The sorted order of the edges determines how the MST formed. Observe that we can also choose to connect vertex 3 and 1 also with weight 4!

Kruskal’s Animation (2)

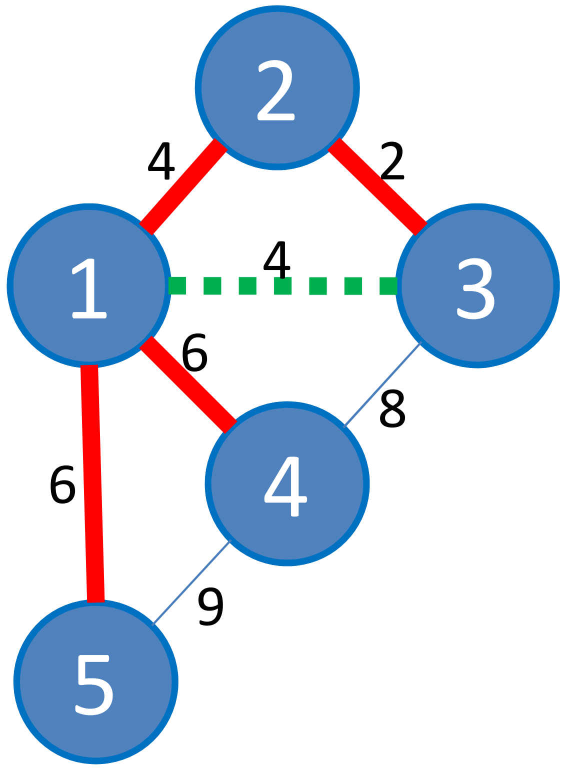
Cannot connect 1 and 3
As it will form a cycle



Connect 1 and 4
The next smallest edge



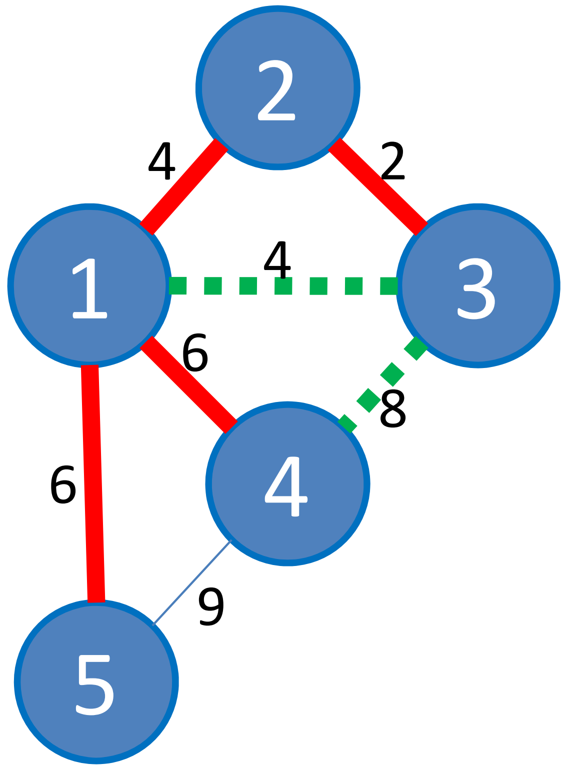
Connect 1 and 5
MST is formed…



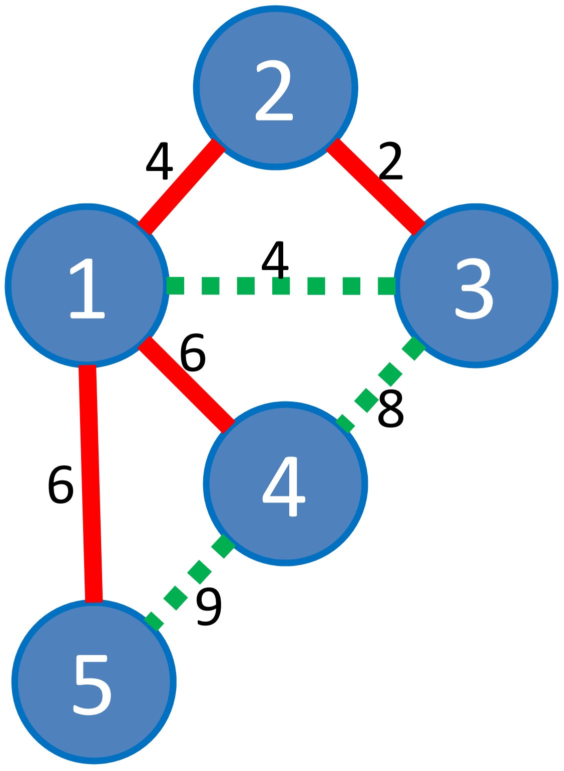
Note: Again, the sorted order of the edges determines how the MST formed;
Connecting 1 and 5 is also a valid next move

Kruskal's Animation (3)

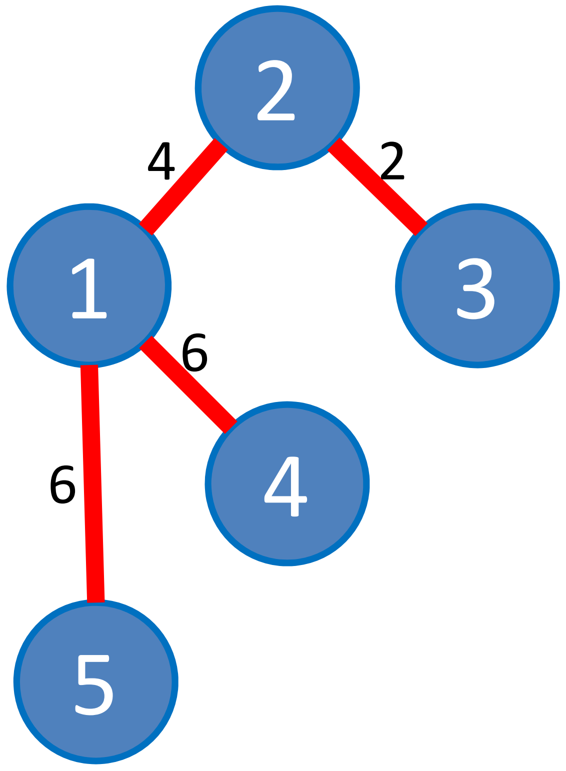
But (standard) Kruskal's algorithm will still continue



However, it will not modify anything else



This is the final MST with cost 18

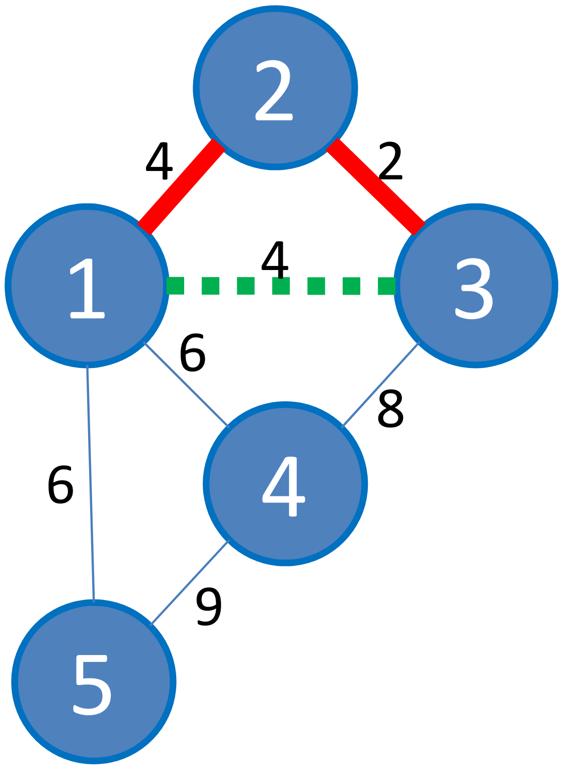


Why It Works? (1)

- First, we have to realize that **Kruskal's algorithm** is a **greedy algorithm**
  - This is because **at each step**, it always try to select the next unprocessed edge e with **minimal weight** (greedy!)
  - Greedy algorithm is usually simple to implement
  - However, it usually requires "proof of correctness"
- Simple proof on how this greedy strategy works
  - Loop invariant: every edge **e** that is added into **T** by Kruskal's algorithm is part of the MST

Why It Works? (2)

Cannot connect 1 and 3
As it will form a cycle



Loop invariant: every edge **e** that is added into **T** by Kruskal's algorithm is part of the MST.

```
sort E edges by increasing weight
T ← {}
while there are unprocessed edges left
  pick an unprocessed edge e with min cost
  if adding e to T does not form a cycle
    add e to T
T is an MST
```

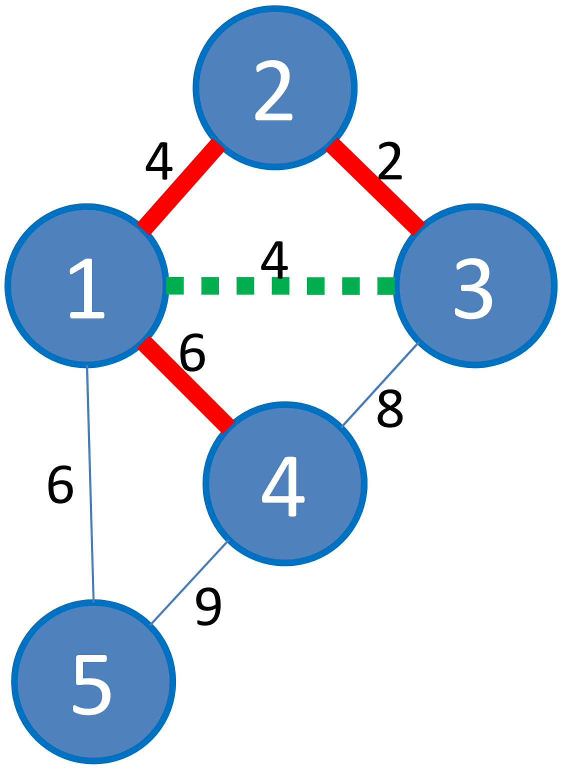
Kruskal's algorithm has a special **cycle check** before adding an edge **e** into **T**. Edge **e** will never form a cycle.

At the start of every loop, **T** is always part of **MST**.

At the end of the loop, we have selected **V-1** edges from a connected weighted graph **G** without having any cycle. This implies that we have a **Spanning Tree**.

Why It Works? (3)

Connect 1 and 4
The next smallest edge



Loop invariant: every edge **e** that is added into **T** by Kruskal's algorithm is part of the MST.

```
sort E edges by increasing weight
T ← {}
while there are unprocessed edges left
  pick an unprocessed edge e with min cost
  if adding e to T does not form a cycle
    add e to T
T is an MST
```

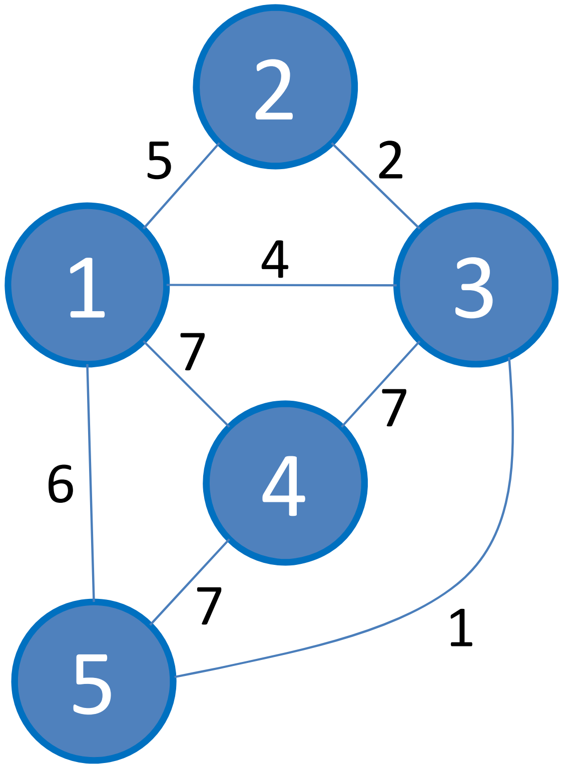
By keep adding the next unprocessed edge **e** with min cost, **w(T U e) ≤ w(T U any other unprocessed edge that does not form cycle)**.

At the start of every loop, **T** is always part of **MST**.

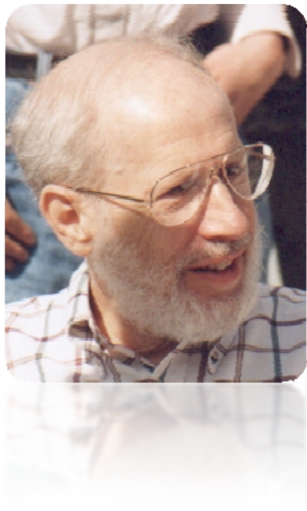
At the end of the loop, the Spanning Tree **T** must have minimal weight **w(T)**, so **T** is the final **MST**.

Quick Challenge

- Find MST of this connected weighted graph
  - Sort the edges and do the greedy strategy as shown earlier



Kruskal’s Algorithm (2)

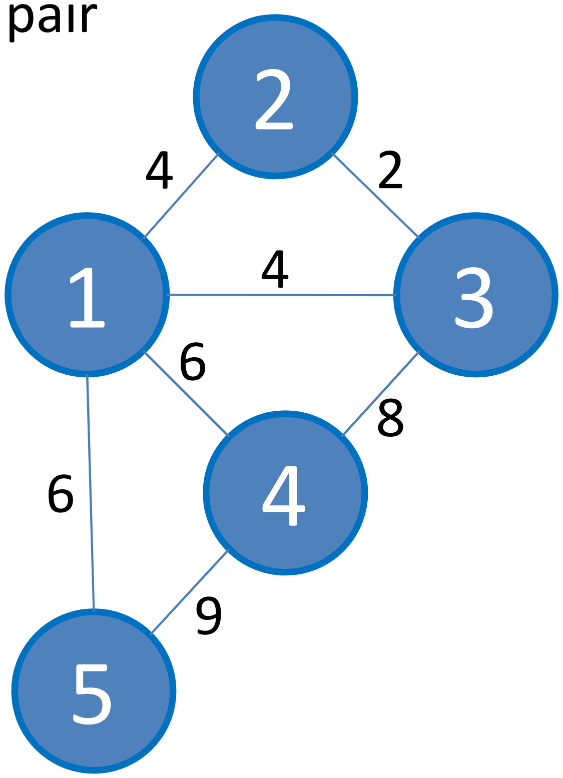


```
sort E edges by increasing weight // O(E log E)
T ← {}
while there are unprocessed edges left // O(E)
  pick an unprocessed edge e with min cost // O(1)
  if adding e to T does not form a cycle // O(X)
    add e to the T // O(1)
T is an MST
```

- To sort the edges:
  - We use a **new** way to store graph information: **EdgeList**
  - Then use “any” sorting algorithm that we have seen before
- To test for cycles:
  - We will use a **new** data structure: **Union Find Disjoint Sets**

Edge List

- Format: array **EdgeList** of E edges
- For each edge i, **EdgeList[i]** stores an (integer) triple {u, v, weight (u, v)}
  - For unweighted graph, weight can be stored as 0 (or 1), or simply store an (integer) pair
- Space Complexity: O(E)
  - Remember, E = O(V^2)
- Adjacency Matrix/List that we have learned earlier are *not suitable* for edge-sorting task!



i	u	v	w
0	2	3	2
1	1	2	4
2	1	3	4
3	1	4	6
4	1	5	6
5	3	4	8
6	4	5	9

Java Implementation (1)

- Introducing class **iii** (**Integer triple**)
  - Used to store 3 attributes: u, v, weight(u, v)
  - Class **iii** implements **Comparable**
    - This allows a collection of this class to be *sorted*
  - Class **iii** has **toString()** method to show its content
- To implement **EdgeList**, we can just use **Vector < iii >**
- We can sort **EdgeList** by using ***one liner***
  Java Collections.sort :O
  - After all efforts for teaching you merge/quick sort… :O
- See the first part of KruskalDemo.java

Survey… then 10 mins break ☺

1. Before this lecture, I don't know that we can sort edges of a graph (by weight) very easily like that… :O
2. I already know that trick ☺

0 of 54

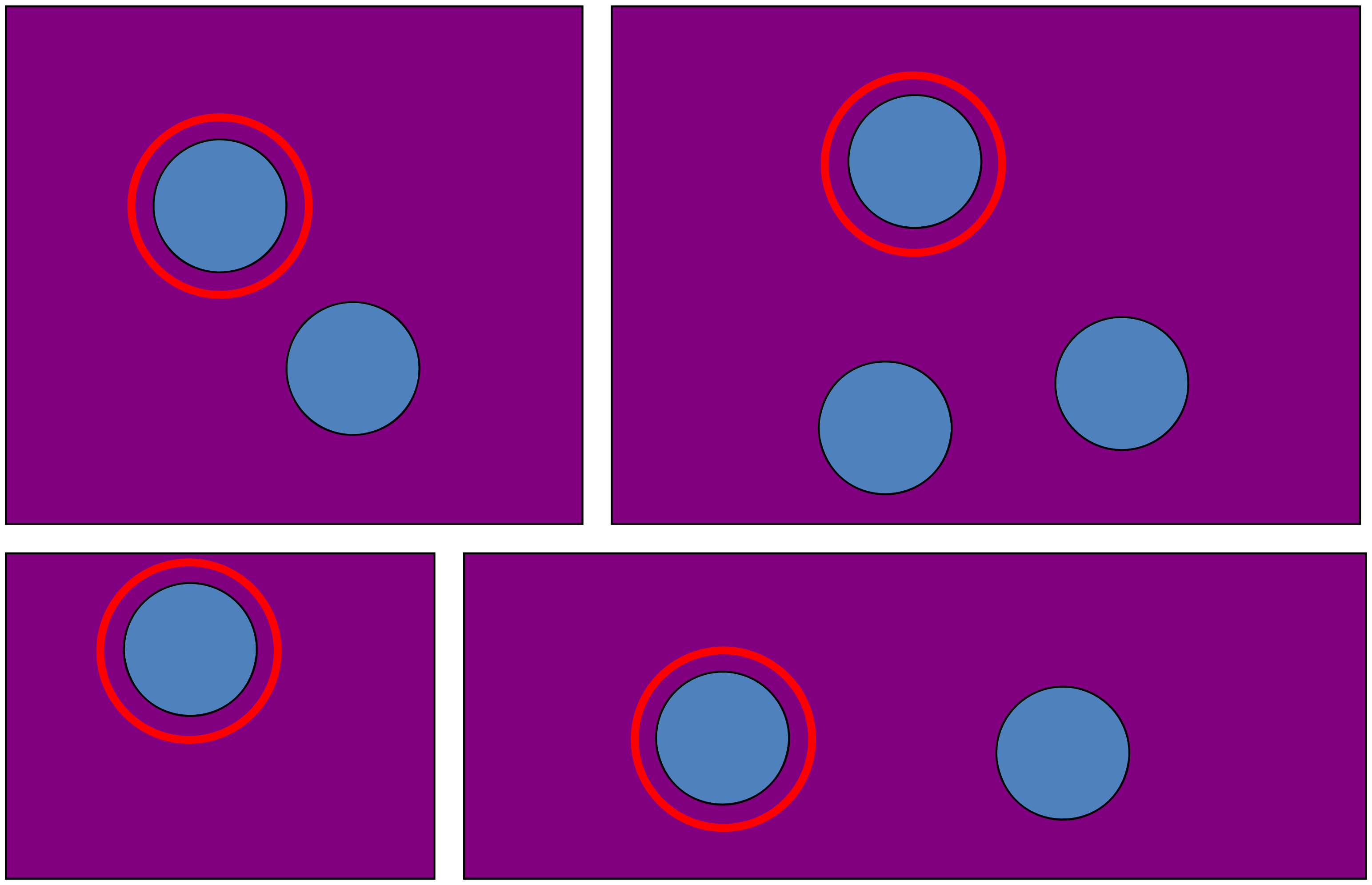
A simple yet effective data structure to model disjoint sets…

UNION-FIND DISJOINT SETS DATA STRUCTURE

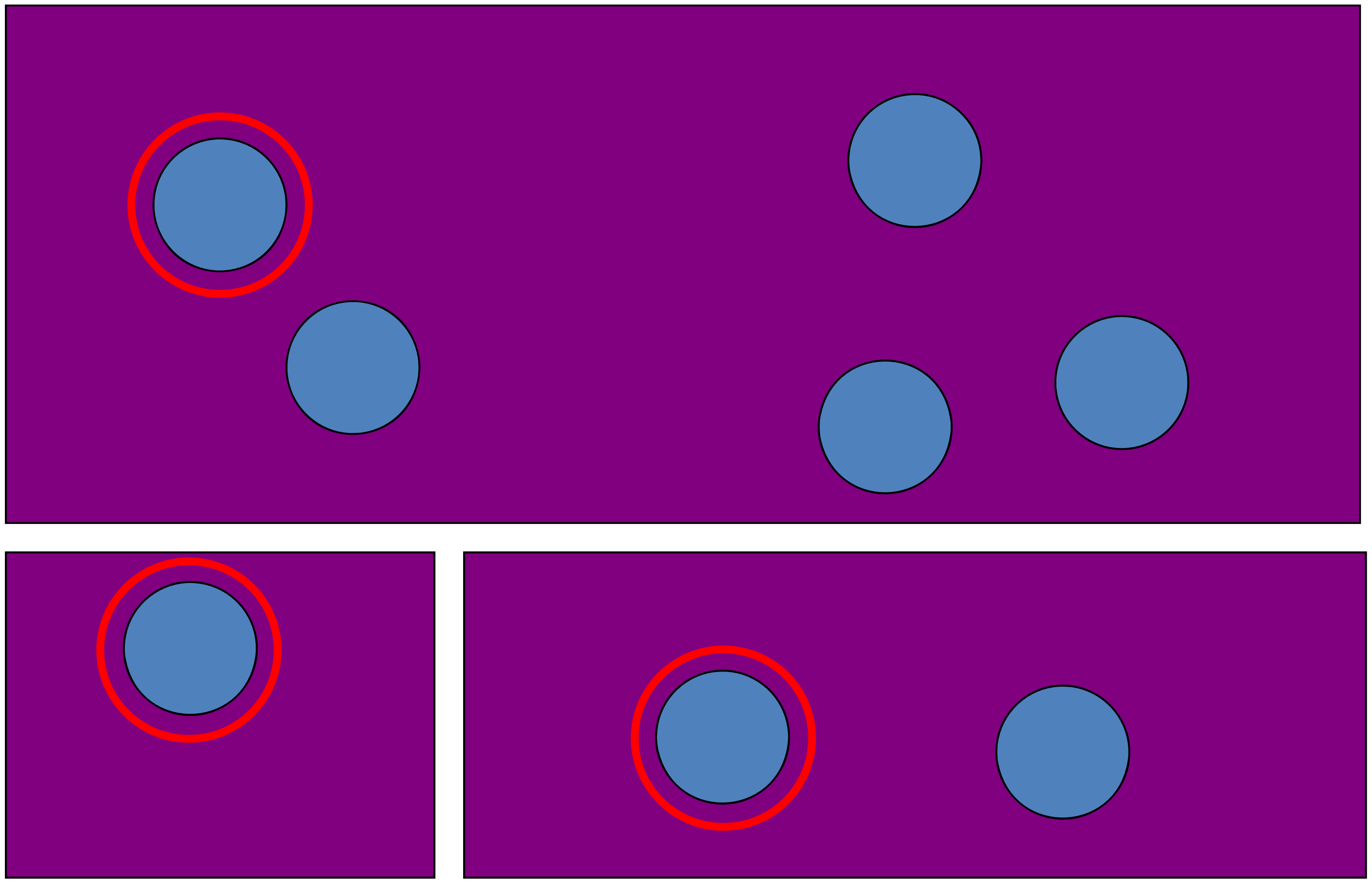
Union-Find Disjoint Sets (1)

- Union-Find Disjoint Sets DS
  - Given several disjoint sets initially…
  - **Combine** them when needed!
  - Key ideas (see the figures over the next few slides):
    - Each set is represented by a “parent” member
    - Initially, each disjoint set has itself as parent
    - When performing union of two members A and B (i.e. combining set containing A and another set containing B), we set one parent (either A’s or B’s parent) to be the parent of the combined set
    - When testing if member x and y belong to the same set, check if their “parent” refer to the same parent

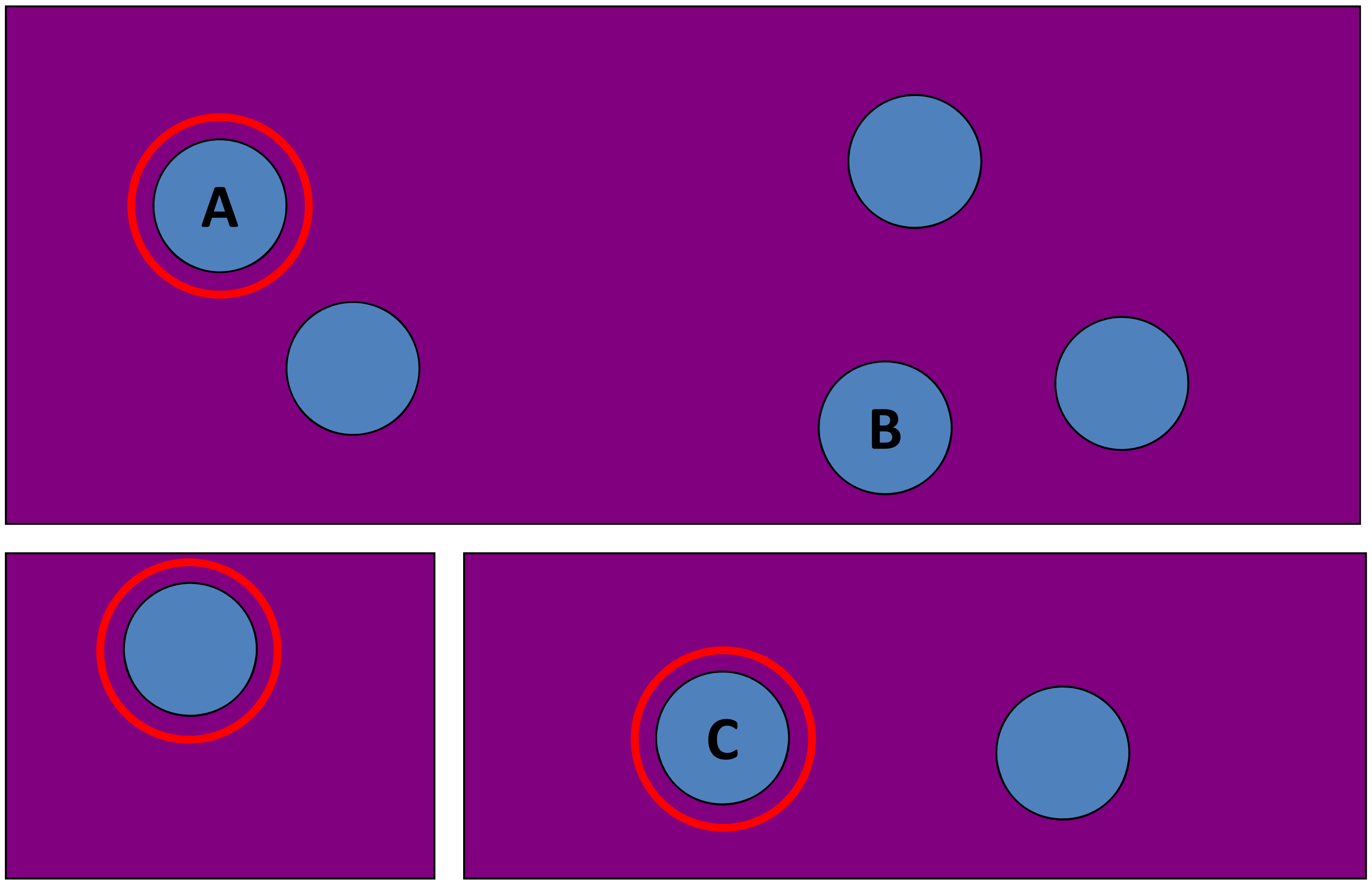
Overview



Operation Union



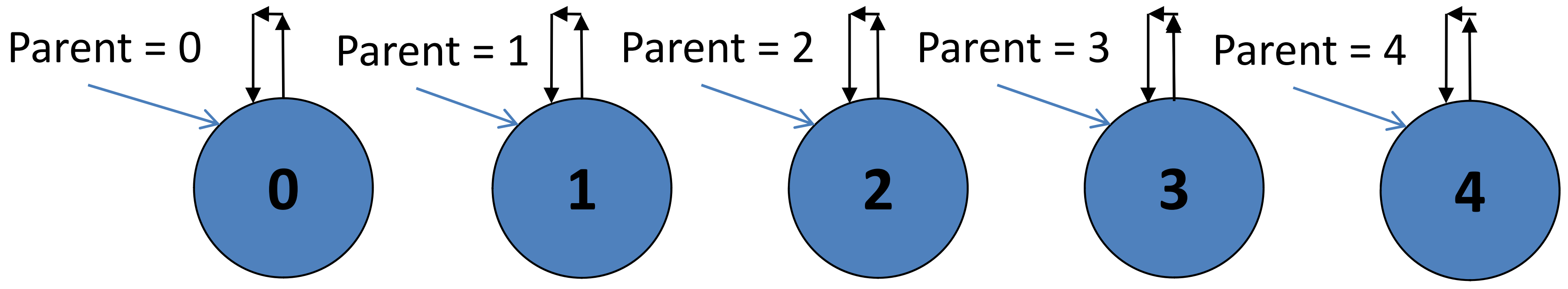
Operation Find



initSet(5)

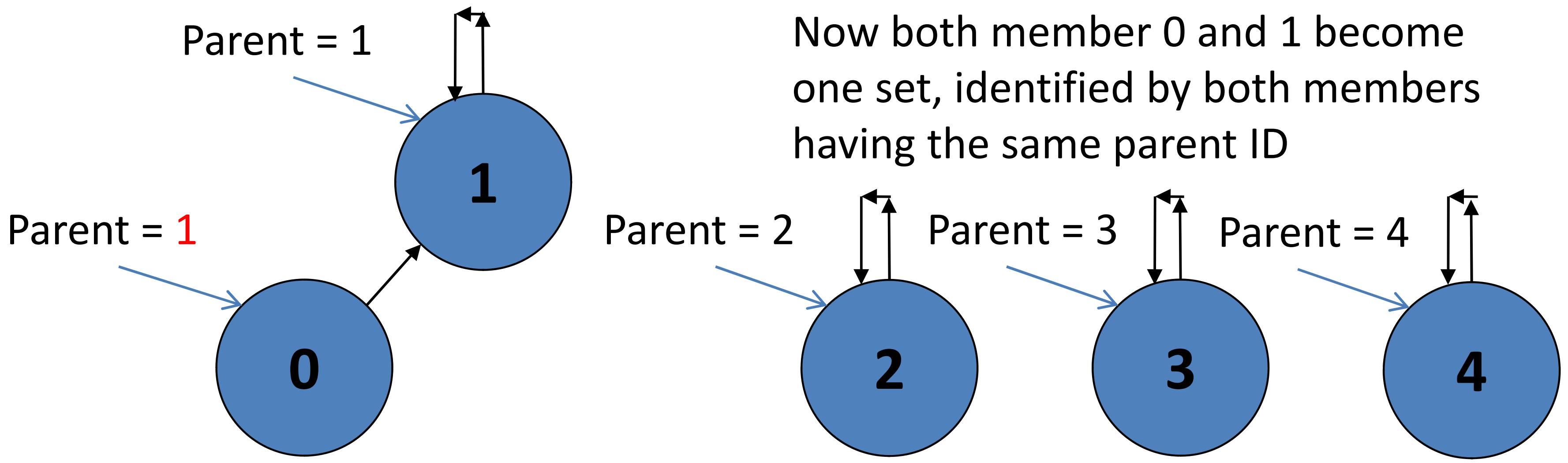
```
private static Vector<Integer> pset;

private static void initSet(int _size) {
  pset = new Vector<Integer>(_size);
  for (int i = 0; i < _size; i++)
    pset.add(i);
}
```



unionSet(0, 1)

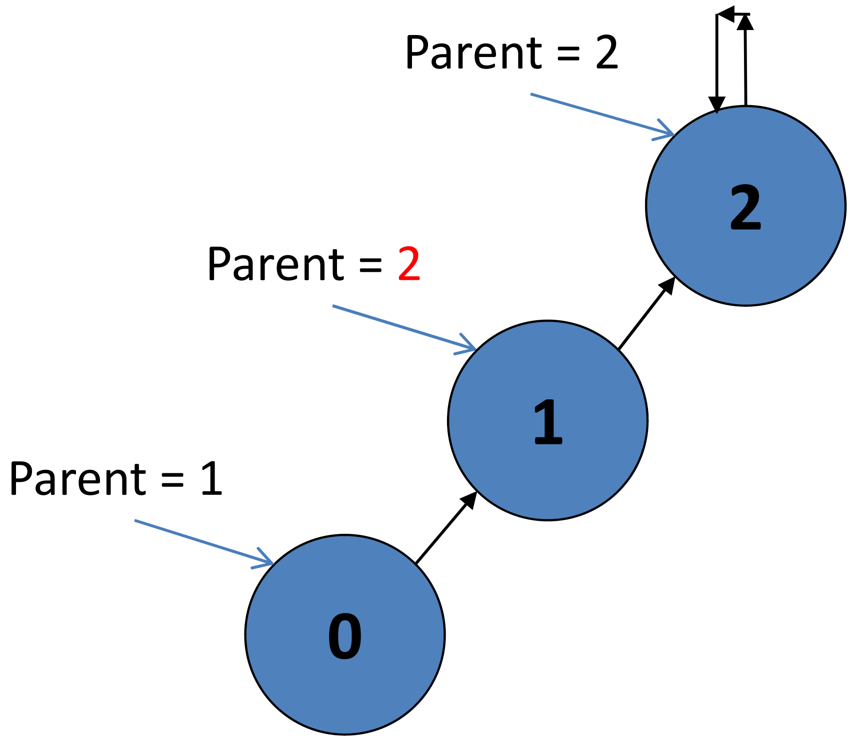
```
private static void unionSet(int i, int j) {
  pset.set(findSet(i), findSet(j));
}
```



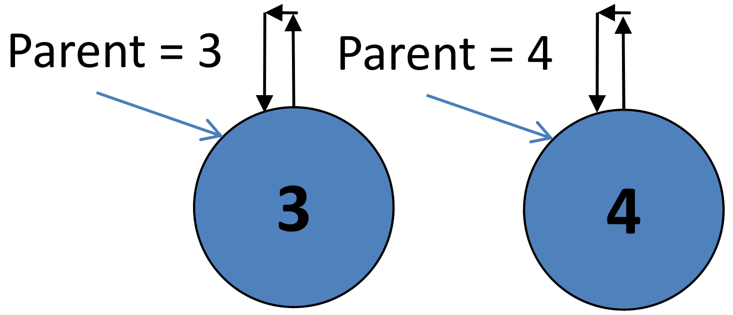
Now both member 0 and 1 become one set, identified by both members having the same parent ID

unionSet(1, 2)

```
private static void unionSet(int i, int j) {
  pset.set(findSet(i), findSet(j));
}
```

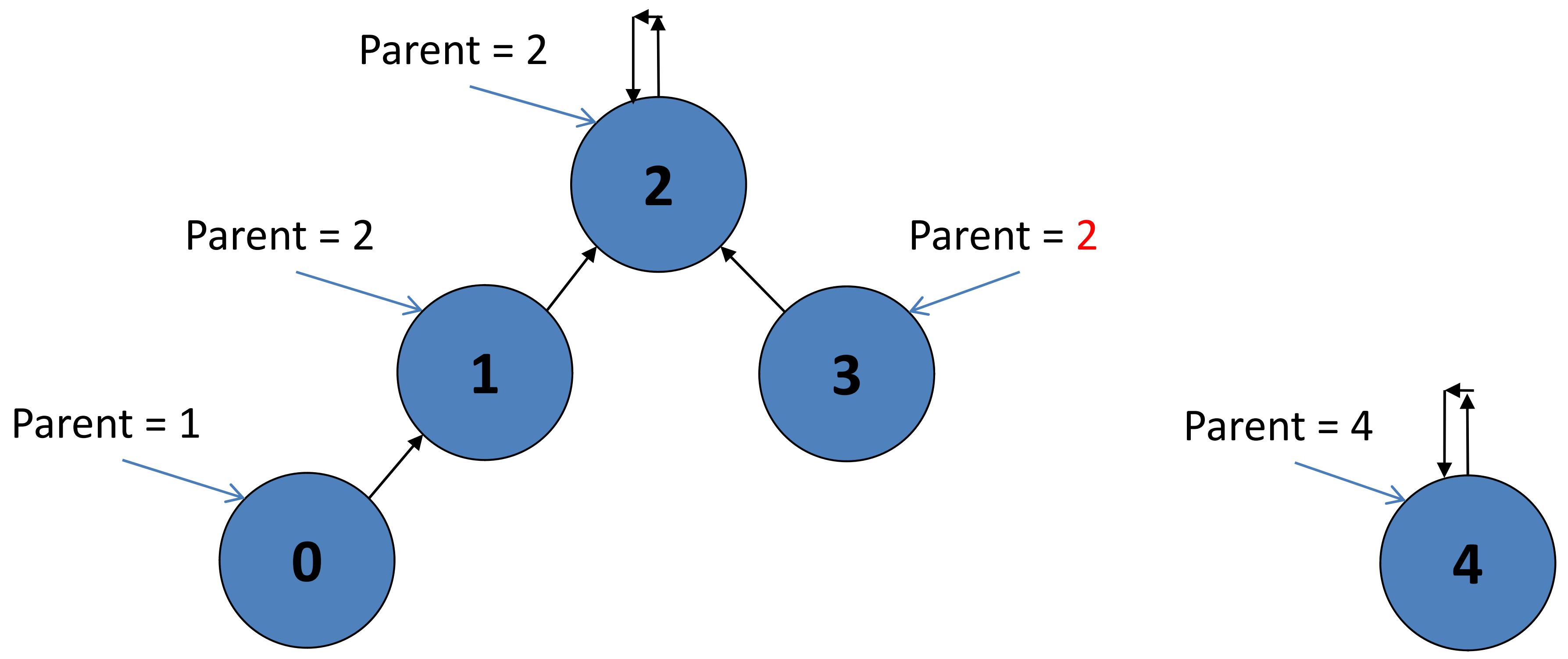


Now members 0, 1, 2 become one set, identified by all three members having the same parent ID, directly **or indirectly**!



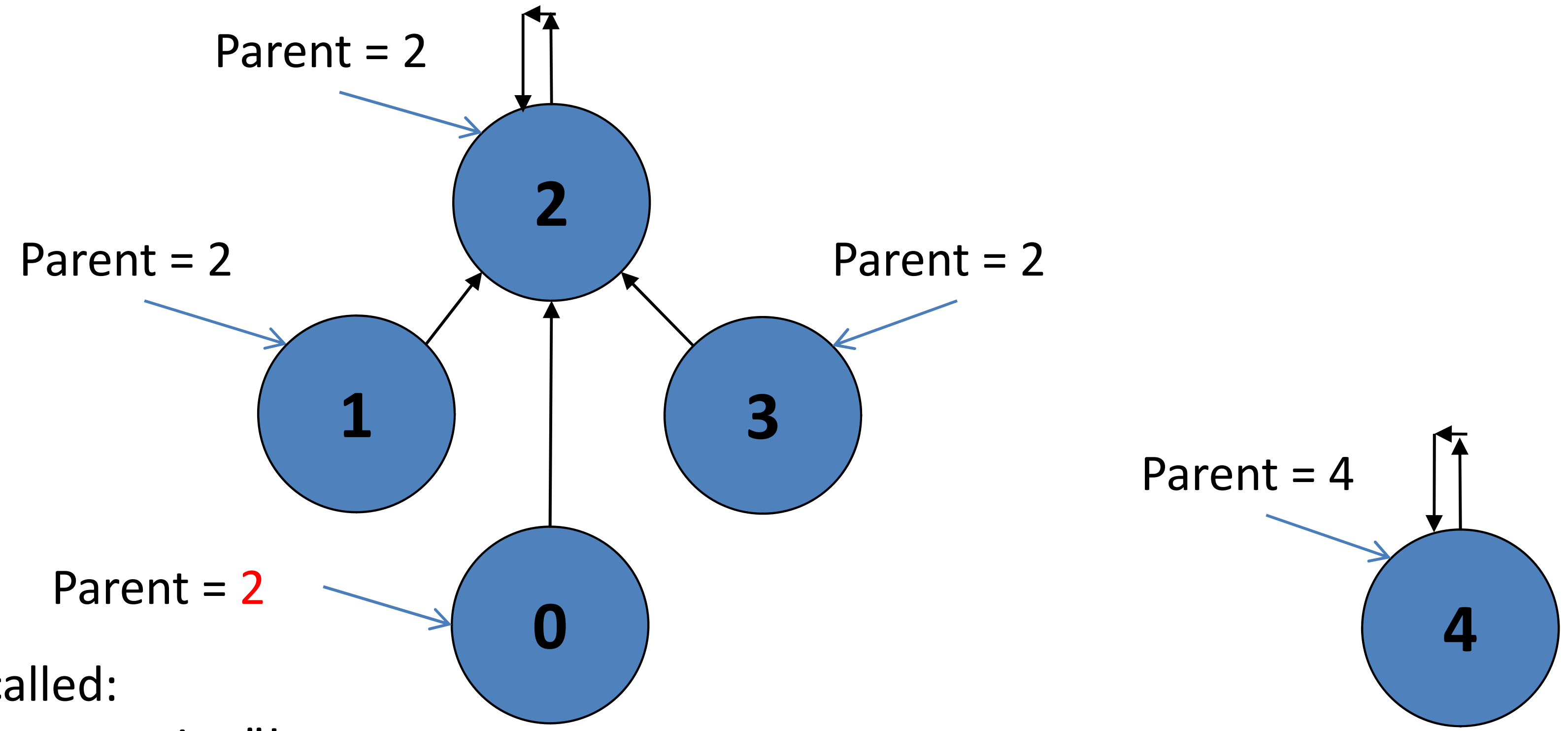
unionSet(3, 1)

```
private static void unionSet(int i, int j) {
  pset.set(findSet(i), findSet(j));
}
```



findSet(0)

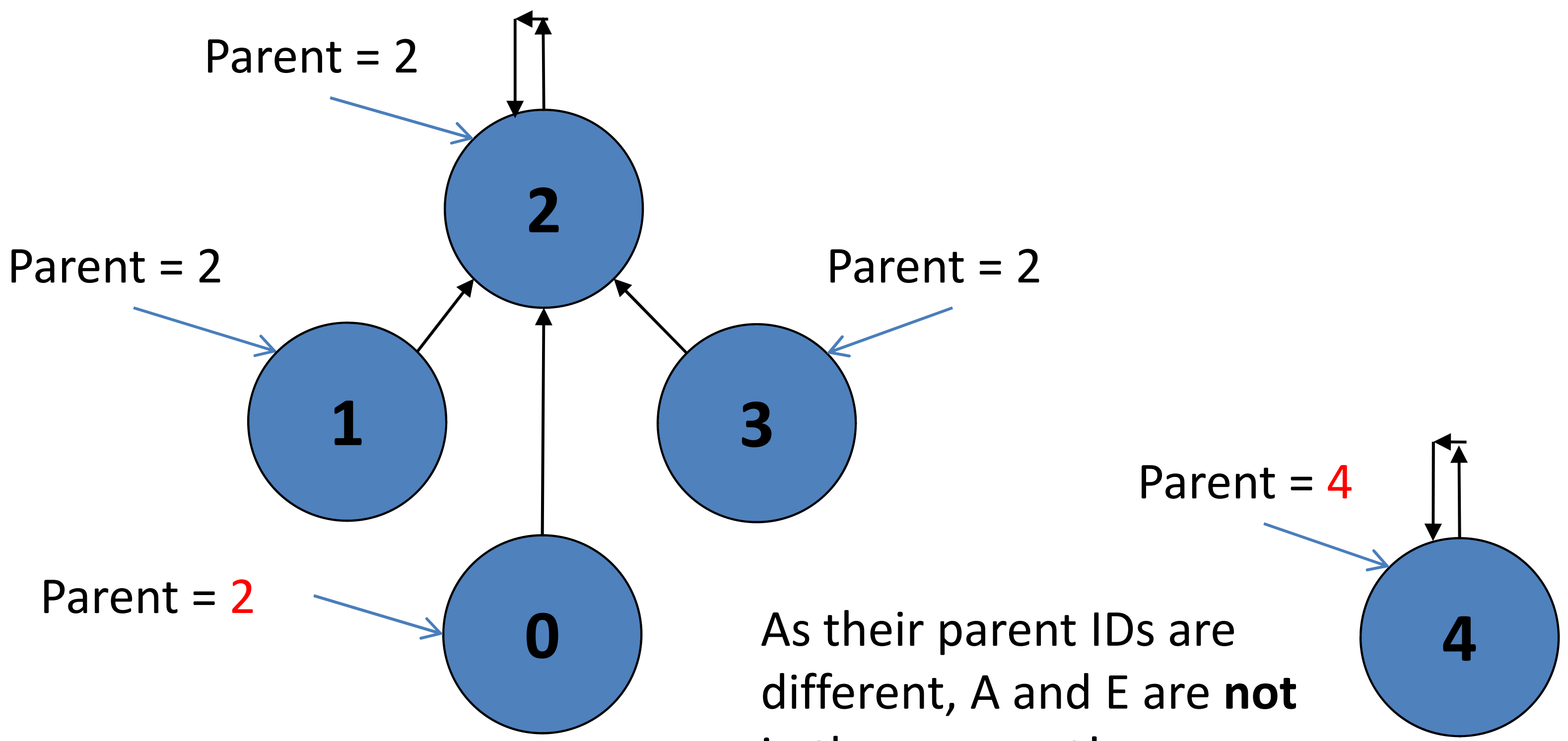
```
public int findSet(int i) {
  if (pset.get(i) == i) return i;
  else {
    pset.set(i, findSet(pset.get(i)));
    return pset.get(i);
  }
}
```



This is called:
"Path Compression"!

isSameSet(0, 4)

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



As their parent IDs are different, A and E are **not** in the same set!

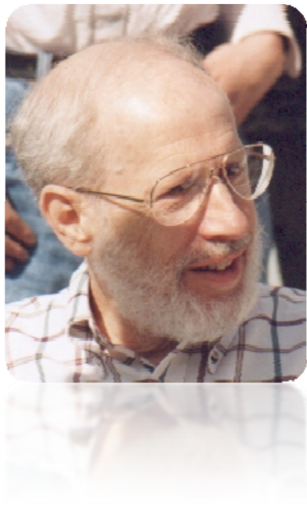
Union-Find Disjoint Sets (2)

- That’s the basics…
  - We will not go into further details
  - Each of these Disjoint Sets operations is just $O(\alpha(V))$
    - $\alpha(V)$ is called the **inverse Ackermann** function
    - This function grows very slowly
    - You can assume it is “constant” for practical values of V (< 1M)
- Further References:
  - **Introductions to Algorithms**, p505-509, ch21.3
  - **Algorithm Design**, p151-157, ch4.6

Java Implementation (2)

- Full implementation of Union Find Disjoint Sets DS
  - See UnionFind.java

Kruskal’s Algorithm (3)



```
sort E edges by increasing weight // O(E log E)
T ← {}
while there are unprocessed edges left // O(E)
  pick an unprocessed edge e with min cost // O(1)
  if adding e to T does not form a cycle // O(α(V))
    add e to the T // O(1)
T is an MST
```

- To sort the edges, we need O(E log E)
- To test for cycles, we need O(α(V)) – small, assume constant
- In overall
  - Kruskal’s runs in O(E log E + ~~E α(V)~~) // E log E dominates!
  - As E = O(V^2), thus Kruskal’s runs in O(E log V^2) = **O(E log V)**

Java Implementation (3)

- Full implementation of Kruskal’s algorithm
- Example of running Kruskal’s algorithm
- See the second part of KruskalDemo.java

Prim’s Algorithm – FYI

- People do like choices
  - So instead of limiting you to Kruskal’s algorithm to solve MST problems, I will *briefly* discuss Prim’s algorithm
- Note that almost? all MST problems are solvable by either one of these two algorithms
- See PrimDemo.java

If given an MST problem, I will…

1. Use/code Kruskal’s algorithm as it is discussed in details in today’s lecture
2. Use/code Prim’s algorithm -- I still think it is simpler

0 of 54

Summary

- Introducing the MST problem
- Introducing the Kruskal's algorithm for finding MST
  - You *may* learn MST/Kruskal's again in CS3230…
- Introducing the EdgeList and technique to sort edges
- Introducing the Union-Find Disjoint Sets DS
- Showing Prim's algorithm for finding MST