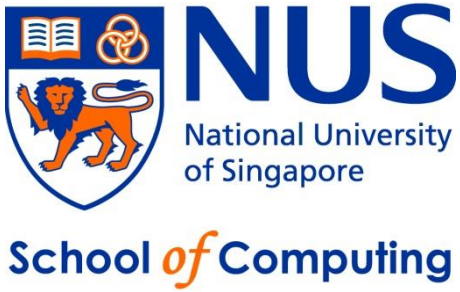
CS2010 – Data Structures and Algorithms II

Lecture 05 – The Foundations

chongket@comp.nus.edu.sg



Admins

- More on written quiz 1 (~80 to 90 mins)
  - 3 sections, 9 questions (some with multiple subtasks) in total
  - Section 1 – Analysis section,
  - Section 2 – More advanced understanding of DSes/algo taught
  - Section 3 – Application questions
  - Material is from lecture 1 to 5 and tutorials 1 to 4
  - Past year quizzes will be put up soon for you to try out

Outline of this Lecture

A. Union-Find Disjoint Sets (UFDS) Data Structure (CP3 Sec 2.4.2)
   - https://visualgo.net/**en/ufds**
B. Motivation on why you should learn graph
   - ***Quick review*** on graph terminologies (from CS1231)
C. Three Graph Data Structures (CP3 Section 2.4.1)
   - Adjacency Matrix
   - Adjacency List
   - Edge List
   - https://visualgo.net/**en/graphds**
D. This lecture is setup for the next half of the module on graph DSes and algorithms

A simple yet effective data structure to model disjoint sets…

This DS will be revisited in Week 07 during lecture on MST

https://visualgo.net**/en/ufds**

CP3, Section 2.4.2

UNION-FIND DISJOINT SETS DATA STRUCTURE

Union-Find Disjoint Sets (UFDS)

UFDS is a collection of disjoint sets

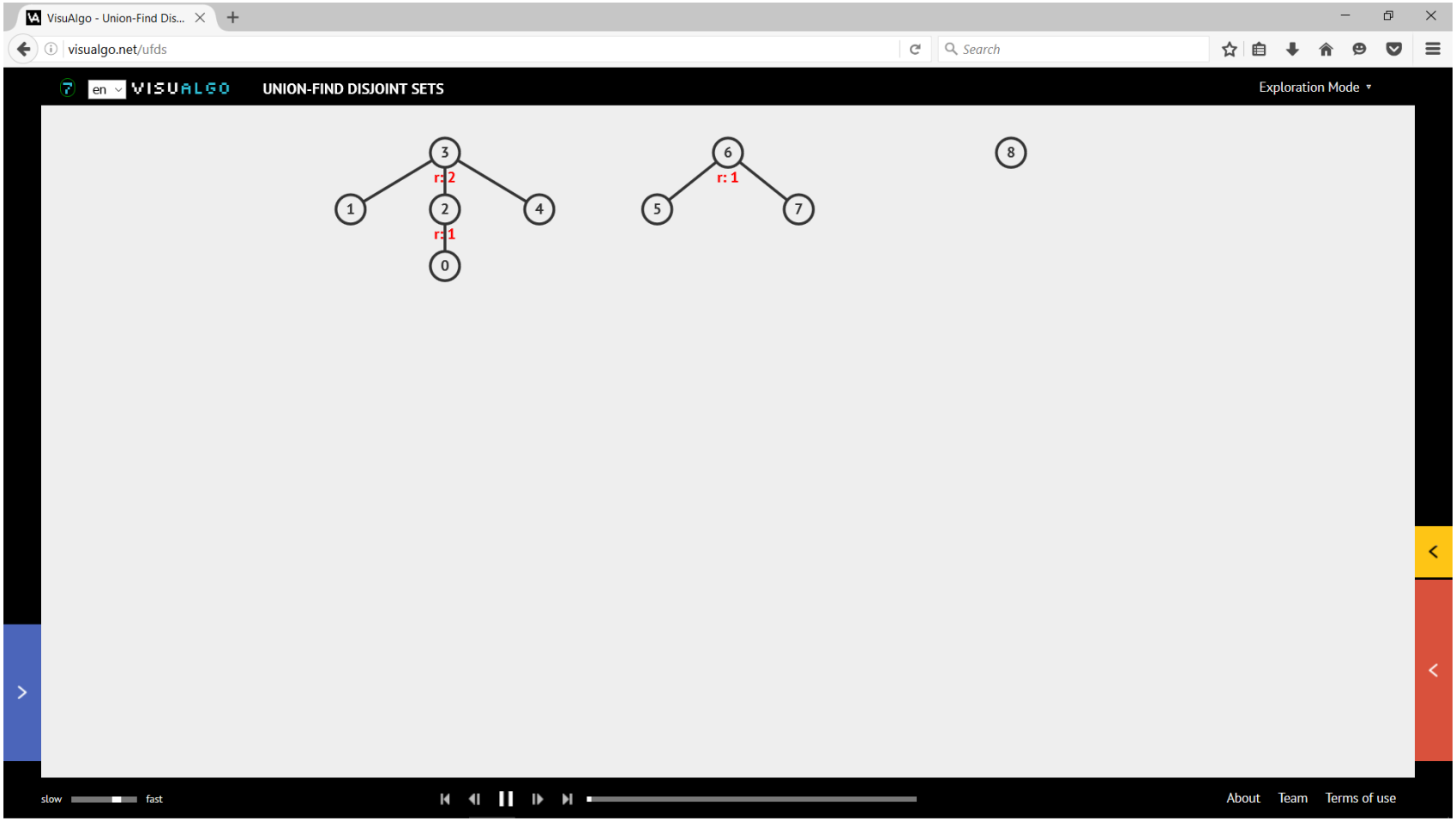
Given several disjoint sets in the UFDS…

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

Key ideas:

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

Example with 3 Disjoint Sets



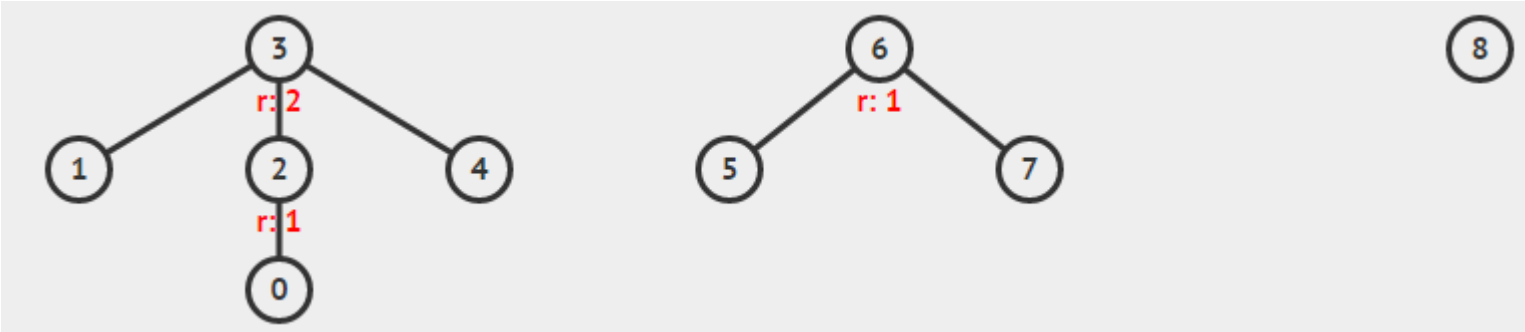
Data Structure to store UFDS

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}**

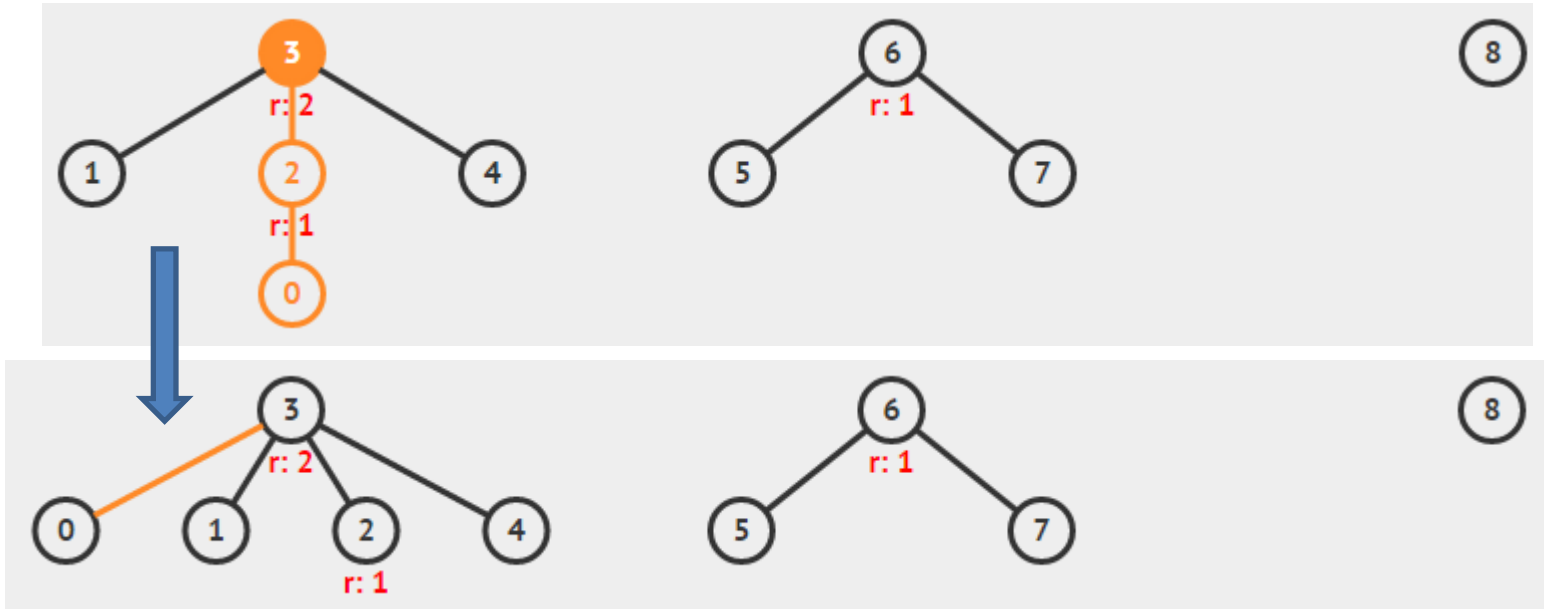
index: 0,1,2,3,4,5,6,7,8



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**; 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*



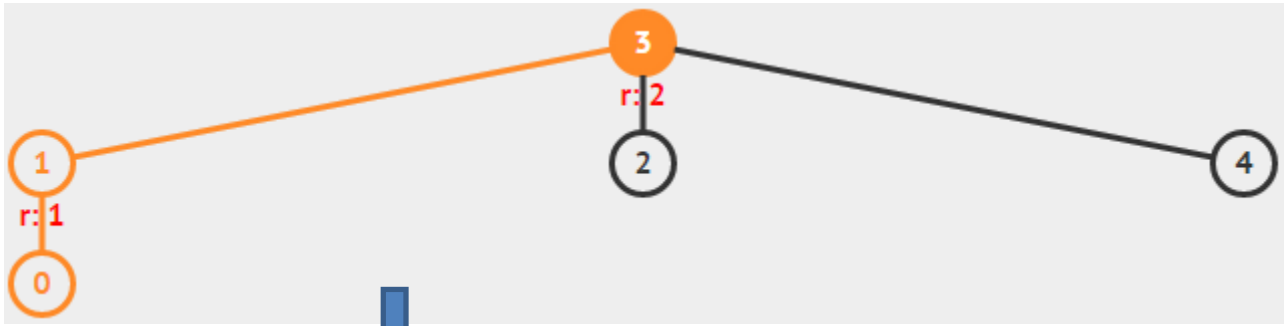
findSet code

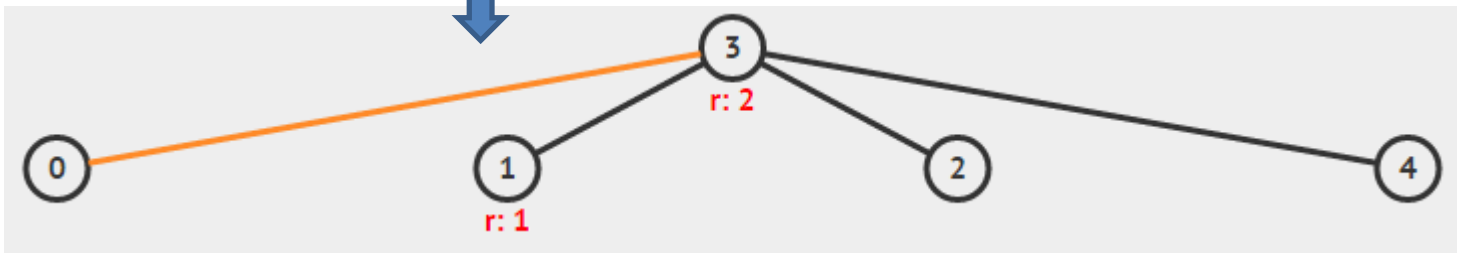
```
public int findSet(int i) {
  if (p.get(i) == i) return i;
  else {
    int ret = findSet(p.get(i));
    p.set(i, ret);
    return ret;
  }
}
```

```
return p.set(i,findSet(p.get(i));
```

Because method **set()** in **Java ArrayList** returns the element *previously at the* specified position

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);
}
```

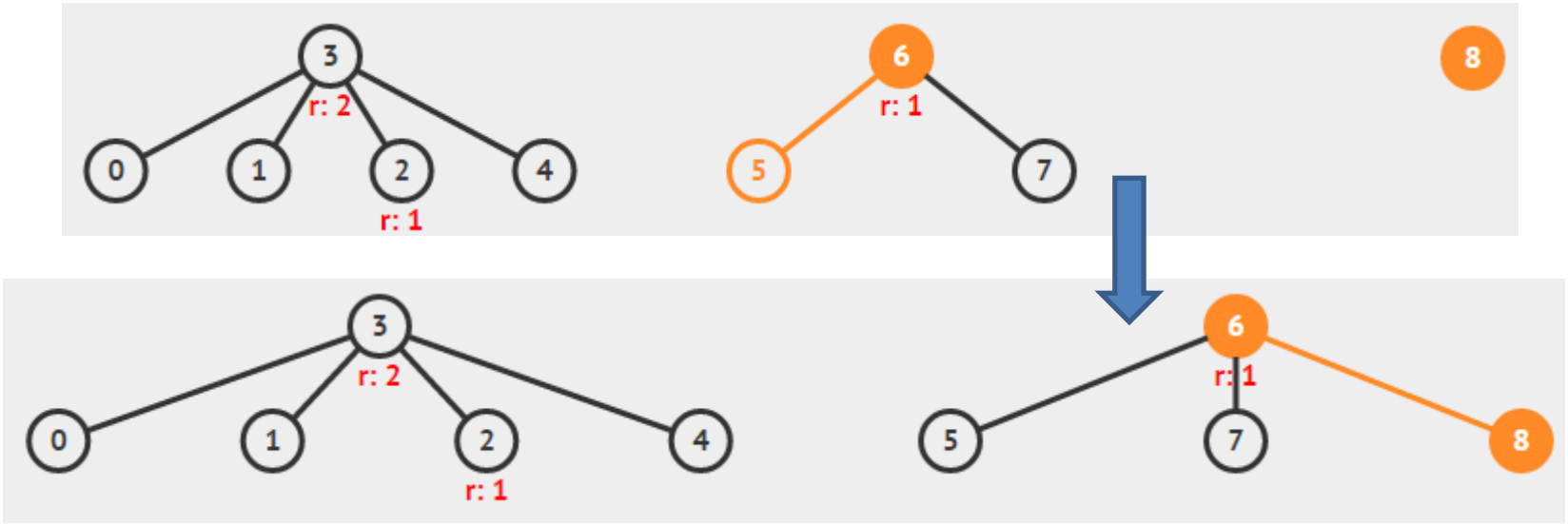
As the representative items of the sets that contains item 0 and 4 are different, we say that 0 and 4 are **not** in the same set!

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

- 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

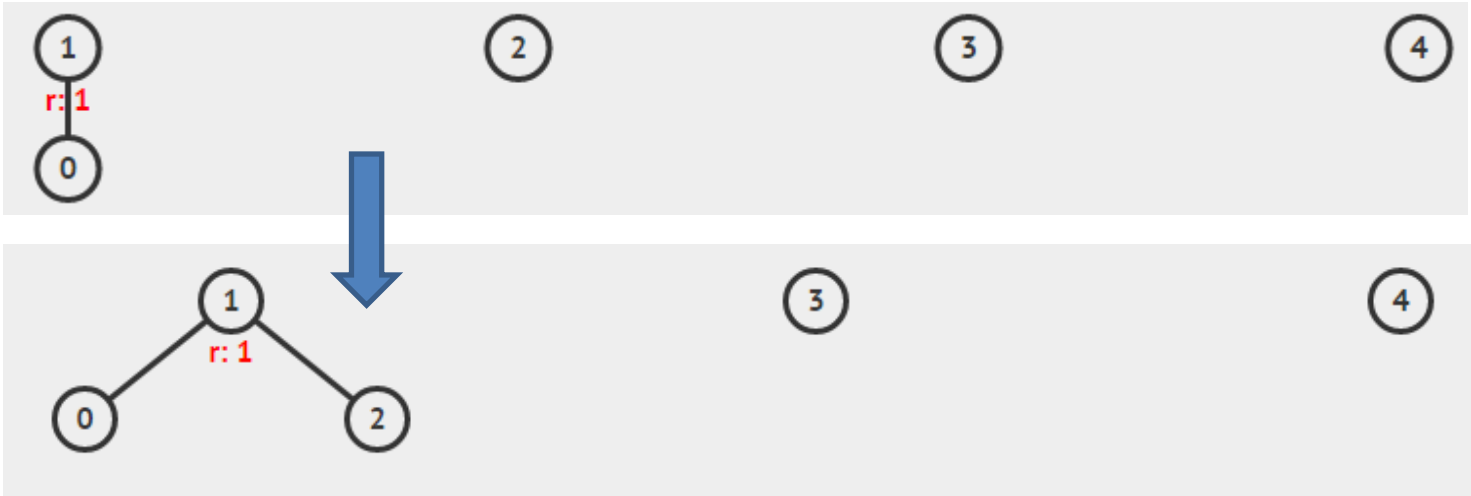
We use another integer array **rank,** where **rank[i]** stores the upper bound of the height of (sub)tree rooted at **i**

- 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]**

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.get(x) > rank.get(y))
      p.set(y, x);
    else {
      p.set(x, y);
      if (rank.get(x) == rank.get(y)) // rank increases
        rank.set(y, rank.get(y)+1); // only if both trees
    }                                // initially have the same rank
  }
}
```

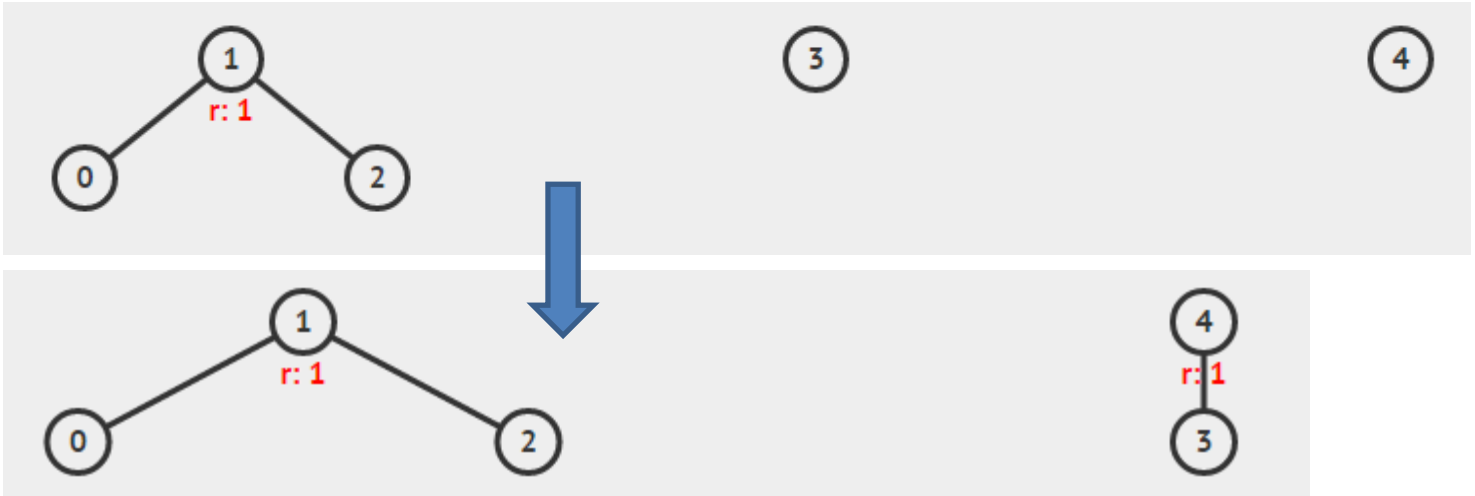
unionSet(0,2)



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.get(x) > rank.get(y))
      p.set(y, x);
    else {
      p.set(x, y);
      if (rank.get(x) == rank.get(y)) // rank increases
        rank.set(y, rank.get(y)+1); // only if both trees
    }                               // initially have the same rank
  }
}
```

unionSet(3,4)



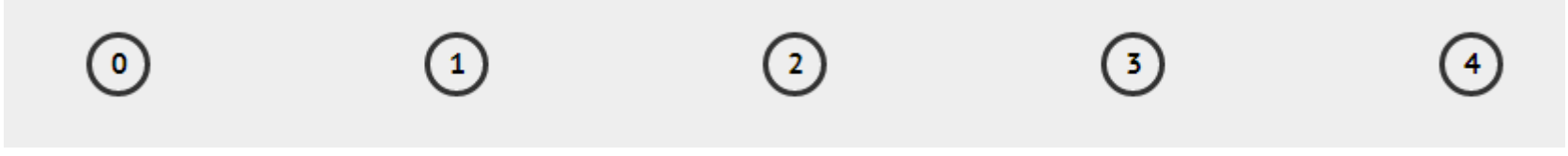
Constructor, UnionFind(N)

```
class UnionFind { // OOP style
  private ArrayList<Integer> p, rank;

  public UnionFind(int N) {
    p = new ArrayList<Integer>(N);
    rank = new ArrayList<Integer>(N);
    for (int i = 0; i < N; i++) {
      p.add(i);
      rank.add(0);
    }
  }

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

UnionFind(5)



UFDS – Summary

That's the basics… we will not go into further details

- UFDS operations runs in just $O(\alpha(N))$ if UFDS is implemented with both "union-by-rank" and "path-compression" heuristics
  - $\alpha(N)$ 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 (< 1M)
    - The analysis is quite hard and not for CS2010 level :O

Further References:

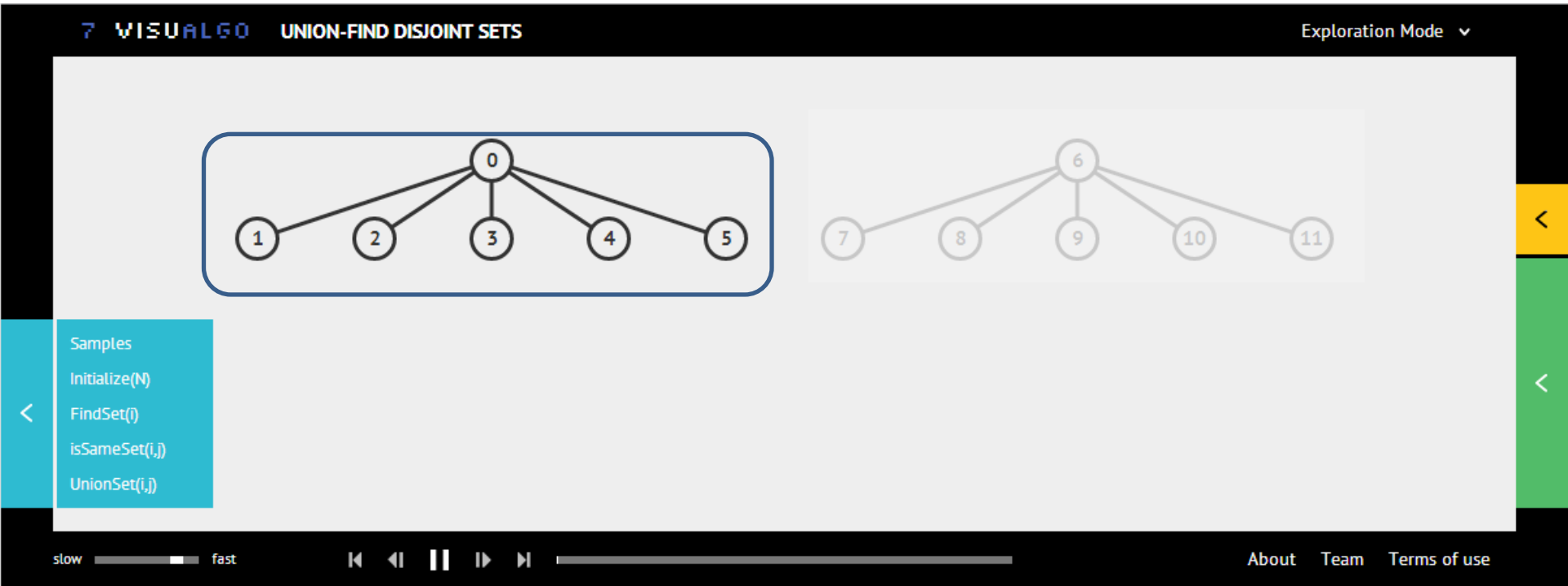
- **Introductions to Algorithms**, p505-509 in 2nd ed, ch 21.3
- **CP3**, Section 2.4.2 (UFDS) and 4.3.2 (MST, Kruskal's)
- **Algorithm Design**, p151-157, ch 4.6
- https://visualgo.net/en/ufds

N reduced to save time

VisuAlgo UFDS Exercise (1)

First, click “**Initialize(N)**”, enter **6**, then click “**Go**”

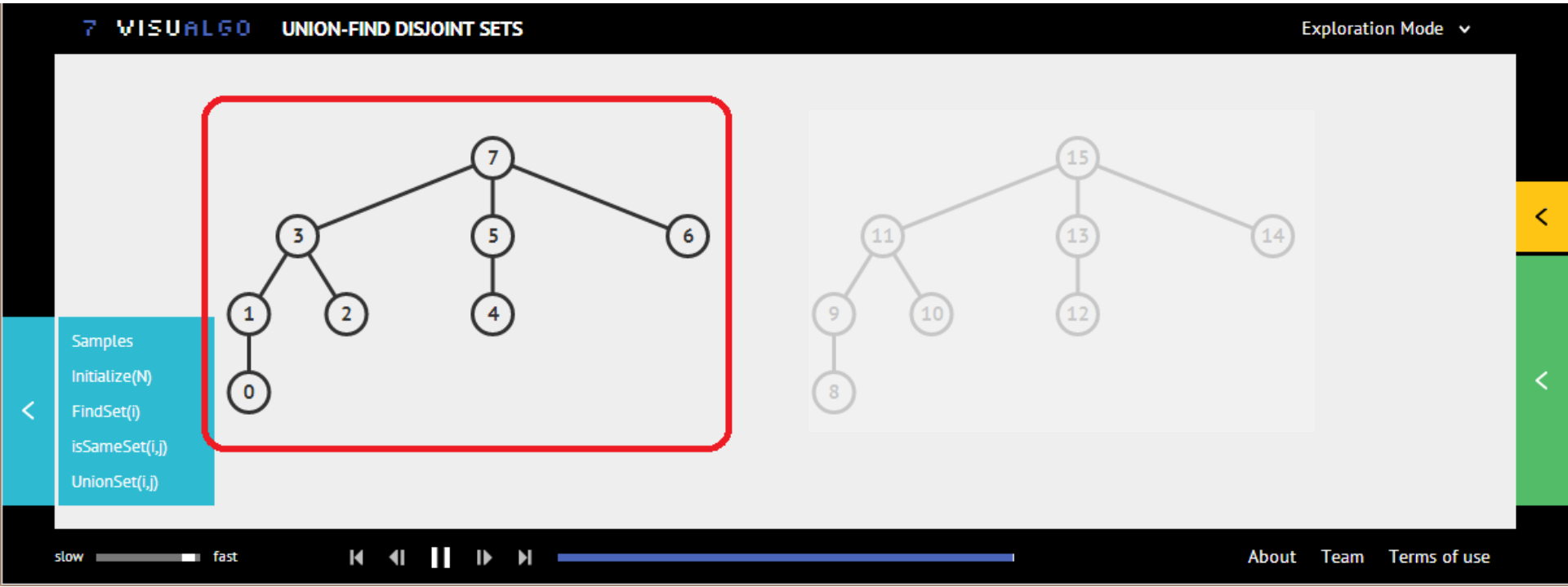
Do a sequence of union and/or find operations to get the left subtree of (**Samples: 2 Trees of Rank 1**)



VisuAlgo UFDS Exercise (2)

First, click “**Initialize(N)**”, enter **8**, then click “**Go**”

Do a sequence of union and/or find operations to get the left subtree of (**Samples: 2 Trees of Rank 3**)



Introductory material

Note that graph will appear from now onwards (Week06-13 :O)

GRAPH

Graph Terminologies (1)

Extension from what you already know: *(Binary) Tree*

- Vertex, Edge, Direction (of Edge), Weight (of Edge)

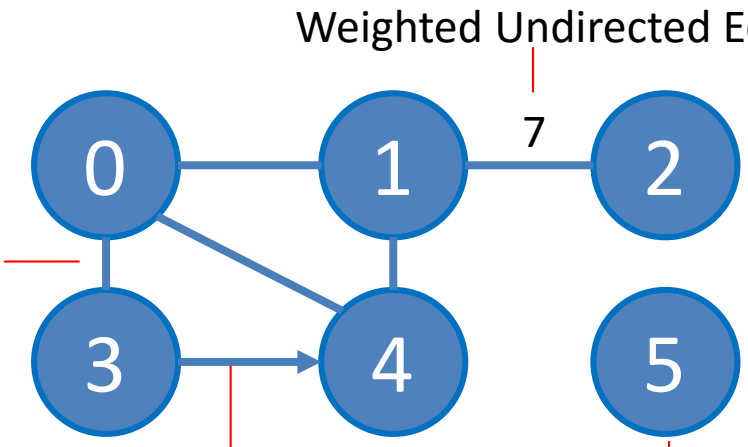
But in a general graph, there is no notion of:

- Root
- Parent/Child
- Ancestor/Descendant

Graph is…

(Simple) graph is a set of vertices where some $[0 .. {}_{N}C_{2}]$ pairs of the vertices are connected by edges

- We will ignore "multi graph" where there can be more than one edge between a pair of vertices



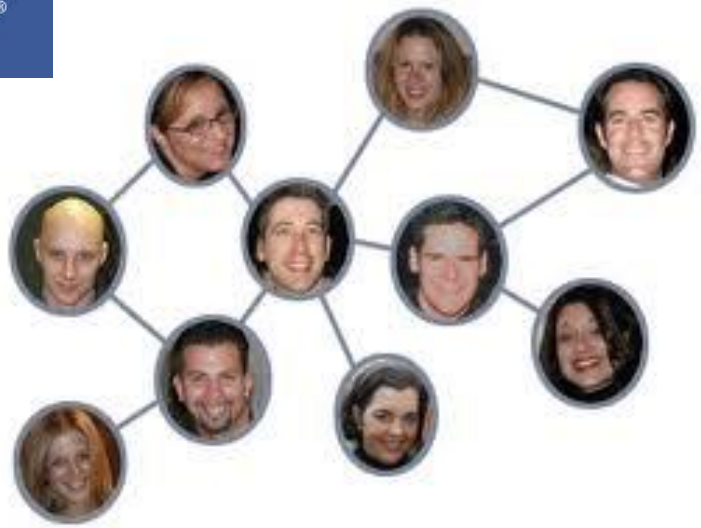
Undirected Edge
Thus vertex 0 and vertex 3 are called "adjacent"
or they are **neighbors** of each other

Directed Edge

A vertex, usually labeled from 0 to V-1

Social Network

facebook



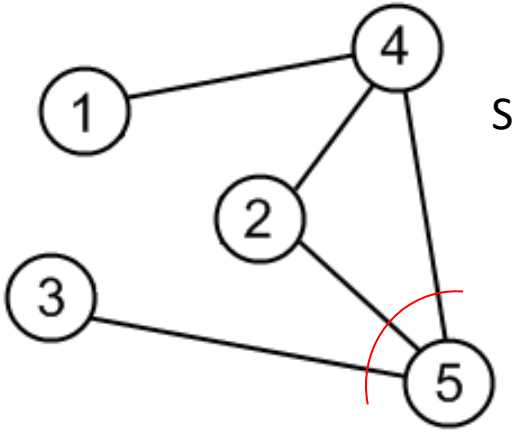
twitter

LinkedIn

Graph Terminologies (2)

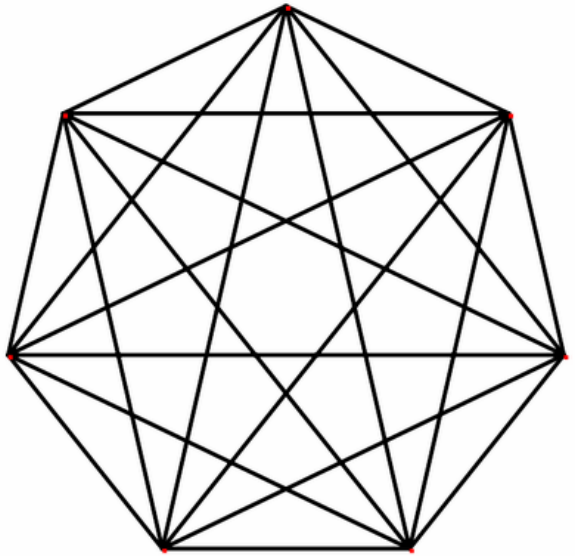
More terminologies (simple graph):

- Sparse/Dense
  - Sparse = not so many edges
  - Dense = many edges
  - No guideline for “how many”
- Complete Graph
  - Simple graph with N vertices and ${}_{N}C_{2}$ edges
- In/Out Degree of a vertex
  - Number of in/out edges from a vertex



Sparse Graph

In/out degree of vertex 5 = 3



Dense Graph

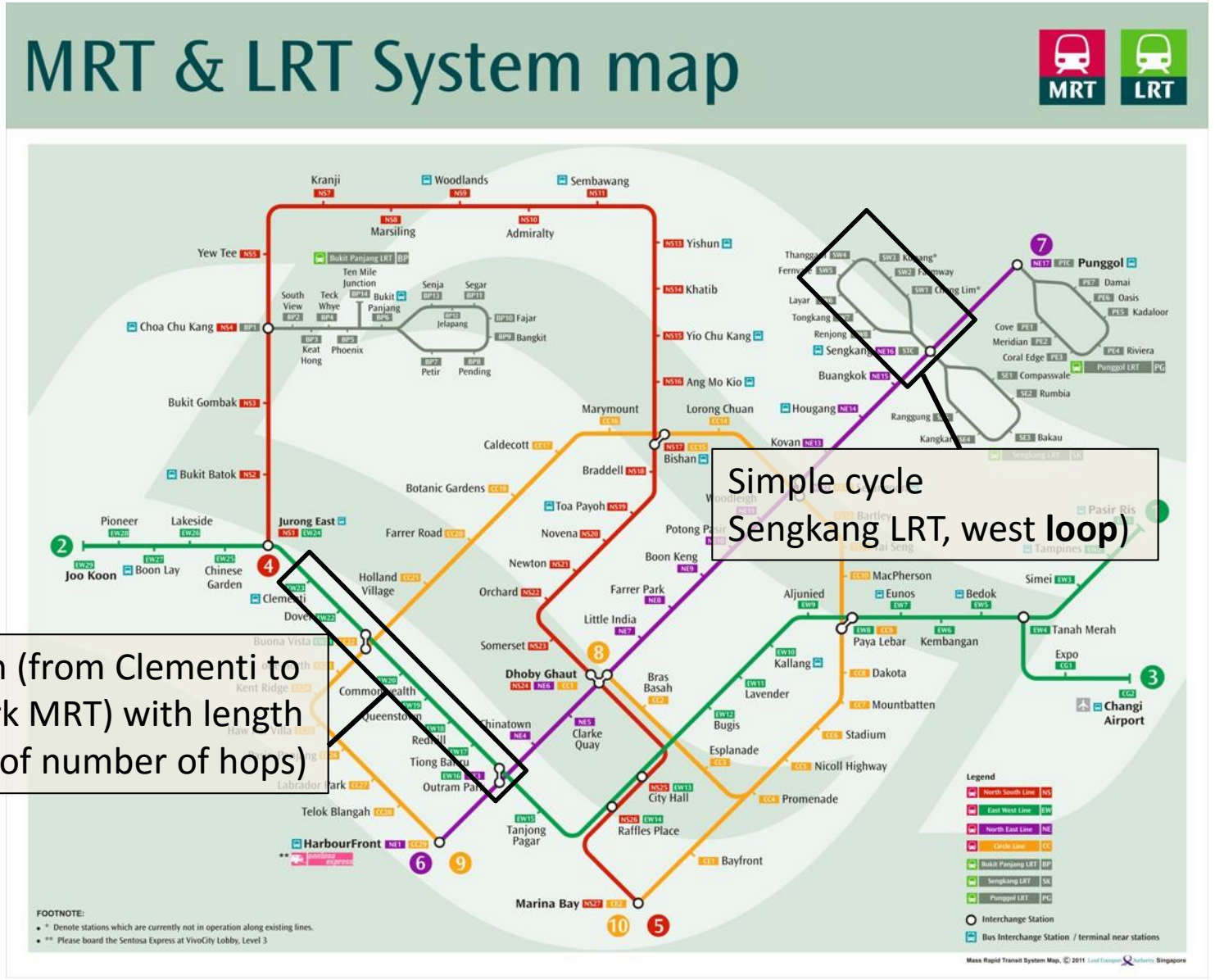
Complete Graph
7 vertices,
${}_{7}C_{2}$ = 21 edges

Graph Terminologies (3)

Yet more terminologies (example in the next slide):

- (Simple) Path
  - Sequence of vertices connected by a sequence of edges
  - Simple = no repeated vertex
- Path Length/Cost
  - In unweighted graph, usually number of edges in the path
  - In weighted graph, usually sum of edge weight in the path
- (Simple) Cycle
  - Path that starts and ends with the same vertex
  - With no repeated vertices except start/end vertex

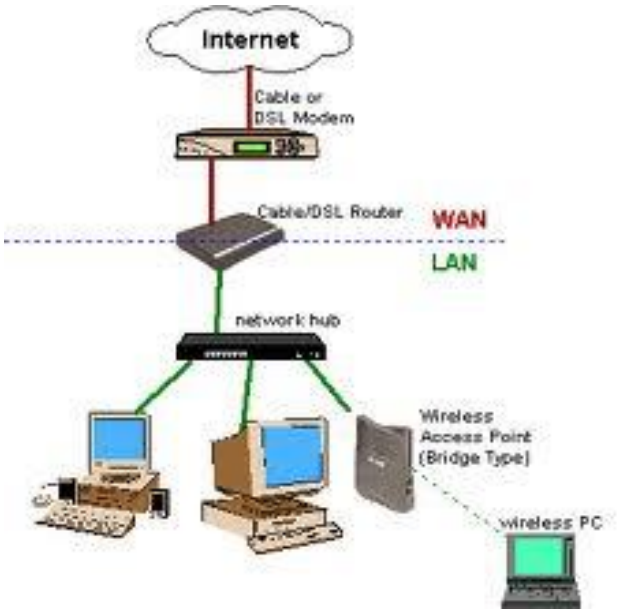
Transportation Network

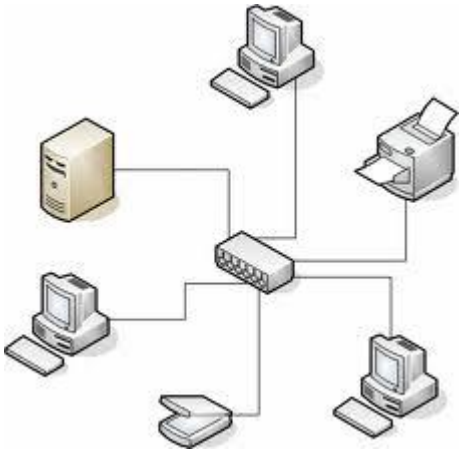


Simple cycle
Sengkang LRT, west **loop**)

Simple path (from Clementi to Outram Park MRT) with length 7 (in terms of number of hops)

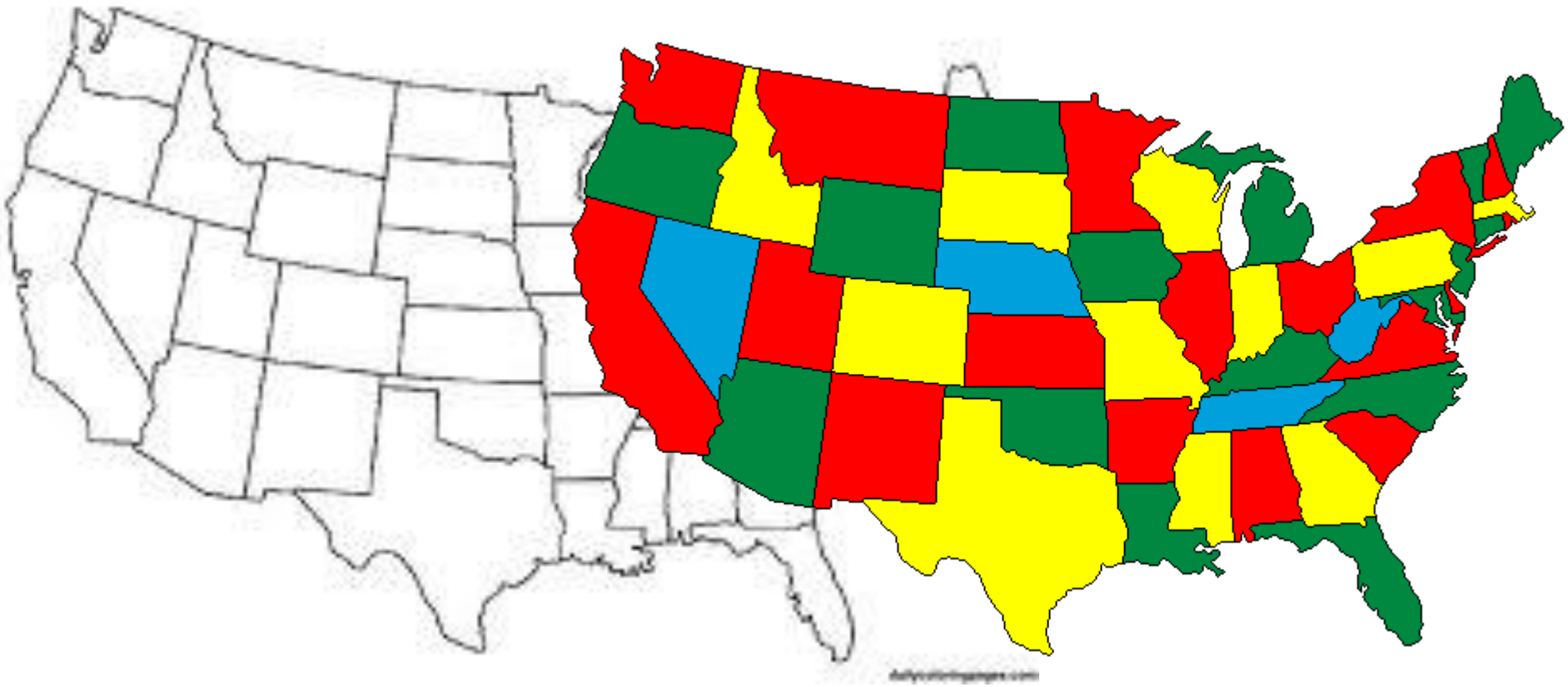
Internet / Computer Networks





Communication Network

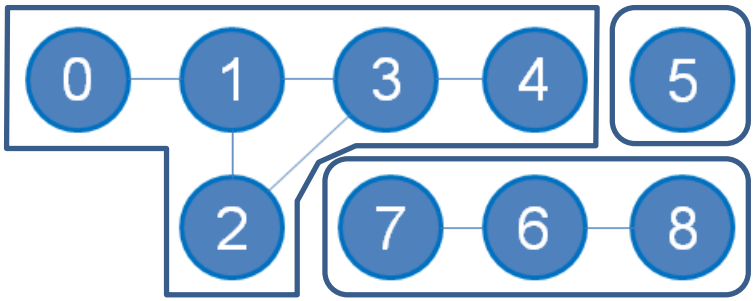
Optimization



Graph Terminologies (4)

Yet More Terminologies:

- Component
  - A group of vertices in an undirected graph that can visit each other via some path
- Connected graph
  - Undirected graph with 1 component
- Reachable/Unreachable Vertex
  - See example
- Sub Graph
  - Subset of vertices (and their connecting edges) of the original graph



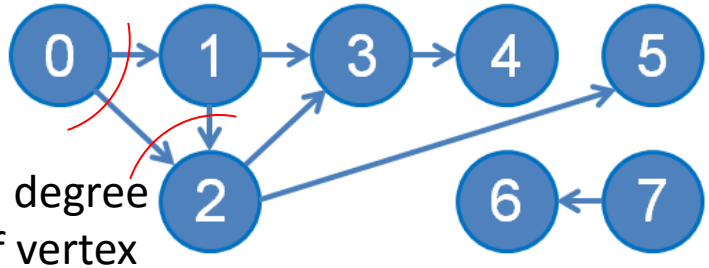
- There are 3 components in this graph
- Disconnected graph (since it has > 1 component)
- Vertices 1-2-3-4 are reachable from vertex 0
- Vertices 5, 6-7-8 are unreachable from vertex 0
- {7-6-8 5} is a sub graph of this graph

Graph Terminologies (5)

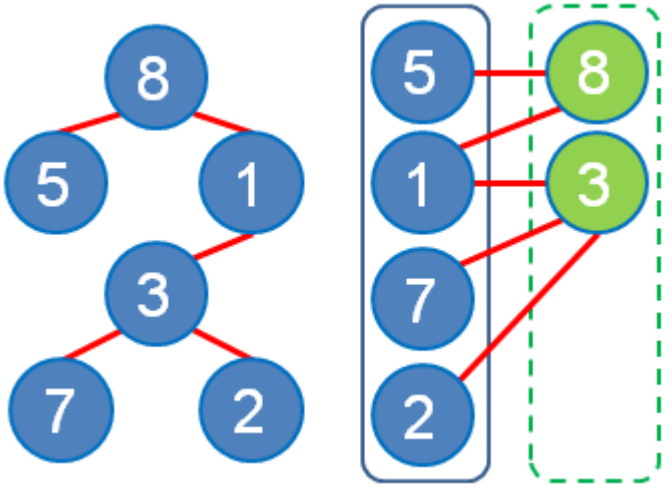
Yet More Terminologies:

- Directed Acyclic Graph (DAG)
  - Directed graph that has no cycle
- Tree (bottom left)
  - Connected graph, E = V - 1, one unique path between any pair of vertices
- Bipartite Graph (bottom right)
  - Undirected graph where we can partition the vertices into two sets so that there are no edges between members of the same set

Out degree of vertex 0 = 2



In degree of vertex 2 = 2



Next, we will discuss three Graph DS

https://visualgo.net/**en/graphds**

This DS will be revisited in Week 06-13 :O (i.e. very important)

Reference: CP3 Section 2.4.1

GRAPH DATA STRUCTURES

Adjacency Matrix

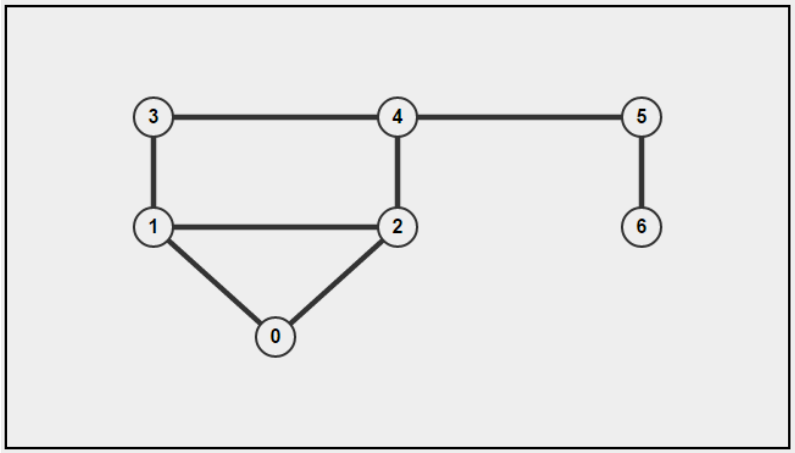
Format: a 2D array **AdjMatrix** (see an example below)

Cell **AdjMatrix[i][j]** contains value 1 if there exist an edge **i→j** in G, otherwise **AdjMatrix[i][j]** contains 0

- For weighted graph, **AdjMatrix[i][j]** contains the weight of edge **i→j**, not just binary values {1, 0}.

Space Complexity: **$O(V^2)$**

- **V** is |V| = number of vertices in G



Adjacency Matrix

	0	1	2	3	4	5	6
0	0	1	1	0	0	0	0
1	1	0	1	1	0	0	0
2	1	1	0	0	1	0	0
3	0	1	0	0	1	0	0
4	0	0	1	1	0	1	0
5	0	0	0	0	1	0	1
6	0	0	0	0	0	1	0

Adjacency List

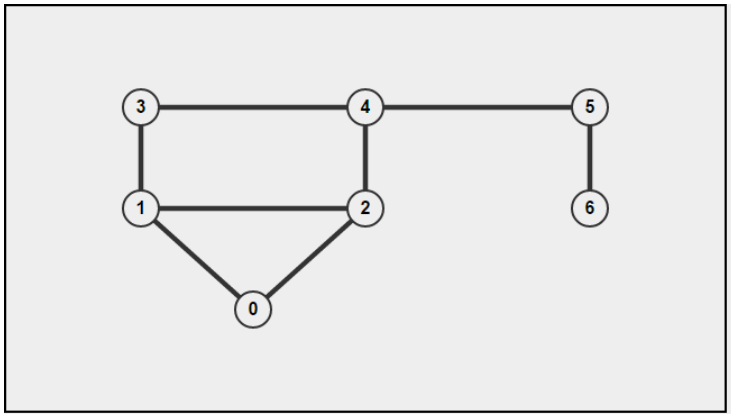
Format: **AdjList** is array of **V** lists, one for each vertex

For each vertex **i**, **AdjList[i]** stores list of **i**'s neighbors

- For weighted graph, stores **pair (neighbor, weight)**
  - Note that for unweighted graph, we can also use the same strategy as the weighted version using (neighbor, weight), but the weight is set to 0 (unused) or set to 1 (to say unit weight)

Space Complexity: **O(V+E)**

- **E** is |E| = number of edges in G, **$E = O(V^2)$**
- **V+E ~= max(V, E)**



Adjacency List			
0:	1	2	
1:	0	2	3
2:	0	1	4
3:	1	4	
4:	2	3	5
5:	4	6	
6:	5		

Edge List

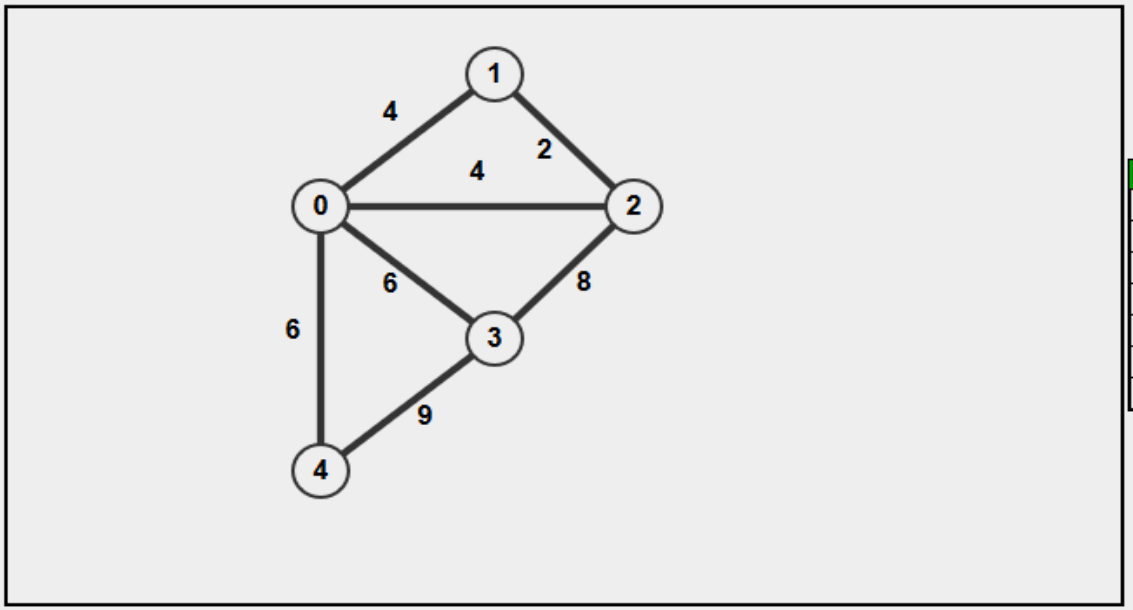
Format: array **EdgeList** of **E** edges

For each edge **i**, **EdgeList[i]** stores an (integer) triple {u, v, w(u, v)}

- For unweighted graph, the weight can be stored as 0 (or 1), or simply store an (integer) pair

Space Complexity: **O(E)**

- Remember, **$E = O(V^2)$**



Edge List			
0:	0	1	4
1:	0	2	4
2:	0	3	6
3:	0	4	6
4:	1	2	2
5:	2	3	8
6:	3	4	9

Java Implementation (1)

Adjacency Matrix: Simple built-in 2D array

```
int i, V = NUM_V; // NUM_V has been set before
int[][] AdjMatrix = new int[V][V];
```

Adjacency List: With Java Collections framework

```
ArrayList < ArrayList < IntegerPair > > AdjList =
  new ArrayList < ArrayList < IntegerPair > >();
// IntegerPair is a simple integer pair class
// to store pair info, see the next slide
```

Edge List: Also with Java Collections framework

```
ArrayList < IntegerTriple > EdgeList =
  new ArrayList < IntegerTriple >();
// IntegerTriple is similar to IntegerPair
```

PS: This is *one* implementation, there are other ways

Java Implementation (2)

```
class IntegerPair implements Comparable<IntegerPair> {
  Integer _first, _second;
  public IntegerPair(Integer f, Integer s) {
    _first = f;
    _second = s;
  }
  public int compareTo(IntegerPair o) {
    if (!this.first().equals(o.first())) // this.first() != o.first()
      return this.first() - o.first();   // is wrong!, we want to
    else                                 // compare their values,
      return this.second() - o.second(); // not their references
  }
  Integer first() { return _first; }
  Integer second() { return _second; }
}
// IntegerTriple is similar to IntegerPair, just that it has 3 fields
```

Summary

In this lecture, we looked at:

A. Union-Find Disjoint Sets (UFDS)
   - for Week 07 later

B. Graph terminologies + why we have to learn graph
   - for Week 06-13 later :O…

C. How to store graph info in computer memory

PS: Written Quiz 1 + Online Quiz 1 includes UFDS, and Graph DS