Data Structures and Algorithms

The Foundations

Acknowledgement

- The contents of these slides have origin from School of Computing, National University of Singapore.
- We greatly appreciate support from Dr. Steven Halim for kindly sharing these materials.

Policies for students

- These contents are only used for students PERSONALLY.
- Students are NOT allowed to modify or deliver these contents to anywhere or anyone for any purpose.

Recording of modifications

- Currently, there are no modification on these contents.

Outline of this Lecture

A. Union-Find Disjoint Sets (UFDS) Data Structure (CP3 Sec 2.4.2)
   - http://visualgo.net/**ufds.html**
B. Bitmask Data Structure (CP3 Section 2.2)
   - http://visualgo.net/**bitmask.html**
C. Motivation on why you should learn graph
   - ***Very quick review*** on graph terminologies (from CS1231)
D. Three Graph Data Structures (CP3 Section 2.4.1)
   - Adjacency Matrix
   - Adjacency List
   - Edge List
   - http://visualgo.net/**graphds.html**

A simple yet effective data structure to model disjoint sets…

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

http://visualgo.net**/ufds.html**

CP3, Section 2.4.2

UNION-FIND DISJOINT SETS DATA STRUCTURE

Union-Find Disjoint Sets (UFDS)

Given several disjoint sets initially…

- Union two disjoint sets when needed
- Find which set an item belongs to
  - More typical usage: To check if two items belong to the same set

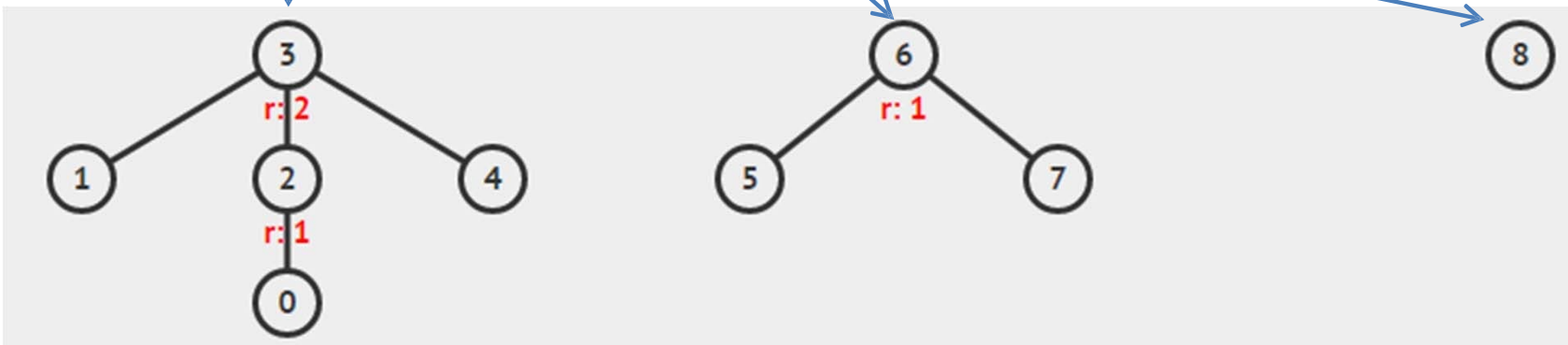
Key ideas (see the figures in the next few slides):

- 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

There are three disjoint sets (trees) in the figure below:

1. Set that contains {0, 1, 2, **3**, 4}
   - Representative item = **vertex 3**, highlighted with **bold + underline,** the root of that subtree
2. Set that contains {5, **6**, 7}
3. Set that contains {**8**}



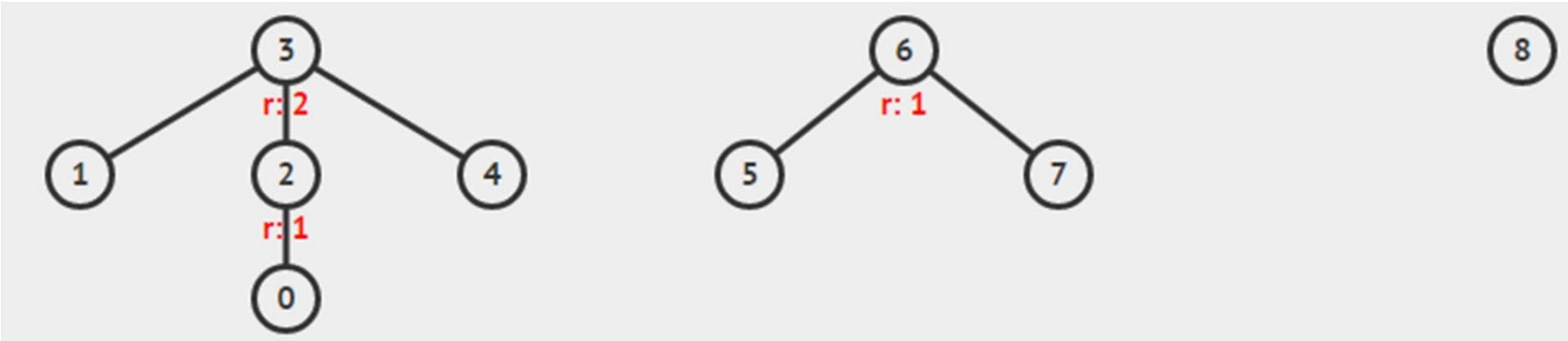
Representation of UFDS

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 this set

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

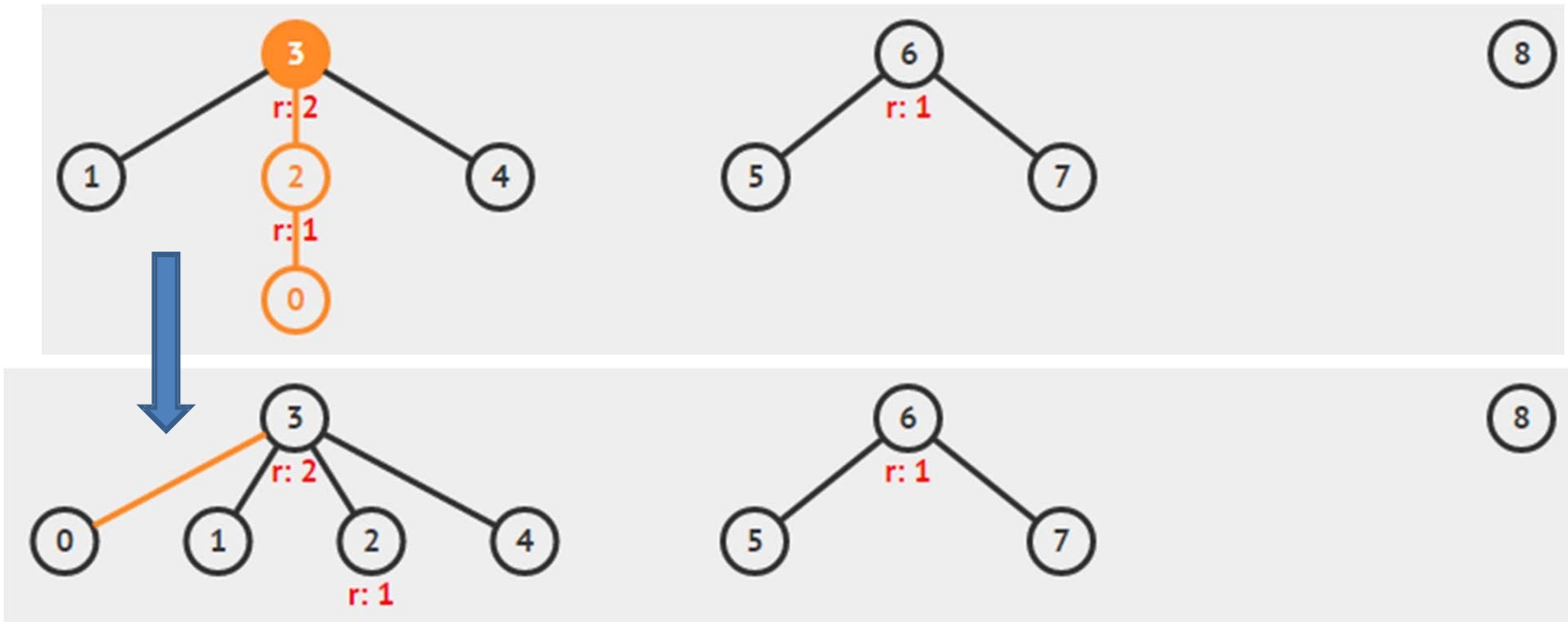
0,1,2,3,4,5,6,7,8



UFDS – FindSet 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)

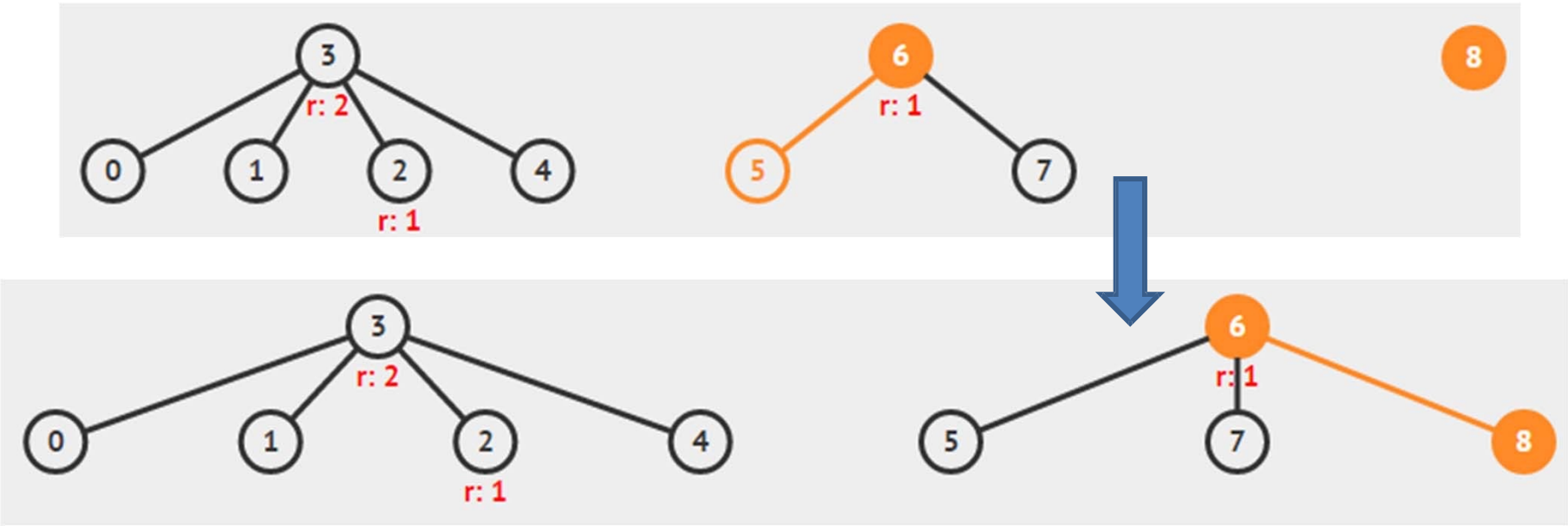
- Example of findSet(0)



UFDS – UnionSet Operation (1)

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

- Example of unionSet(5, 8)



UFDS – UnionSet 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 array **rank[i]** to store 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]**

Constructor, e.g. UnionFind(5)

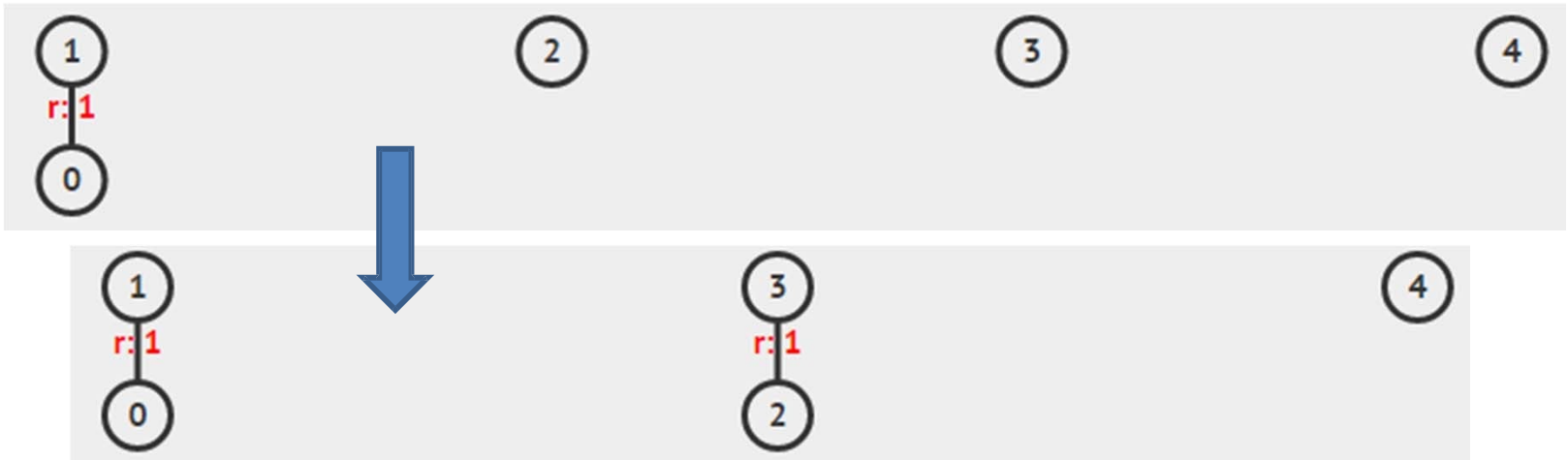
```
class UnionFind { // OOP style
  private Vector<Integer> p, rank;

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

  // ... other methods in the next few slides
}
```

unionSet(0, 1) then unionSet(2, 3)

```
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(4, 3)

```
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
  }
}
```

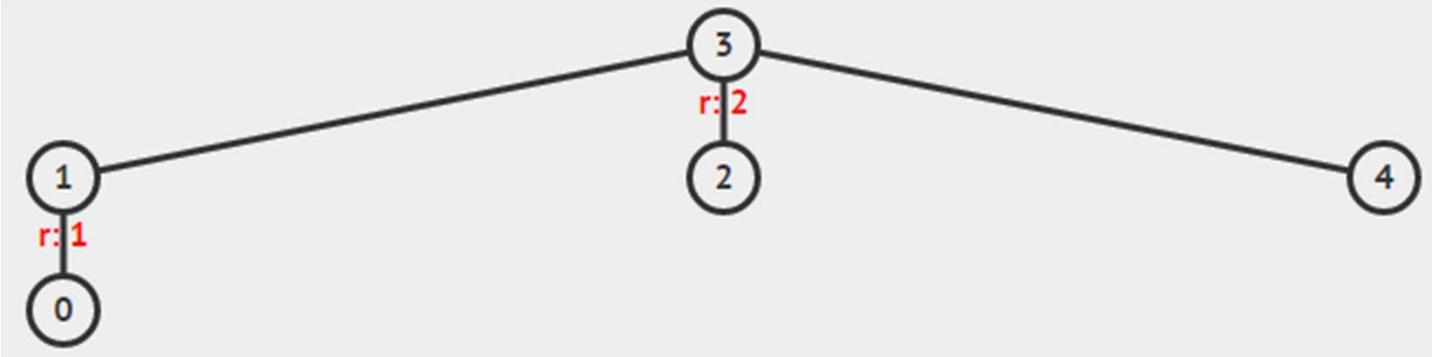
isSameSet(0, 4)

```
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!

unionSet(0, 3)

```
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
  }
}
```

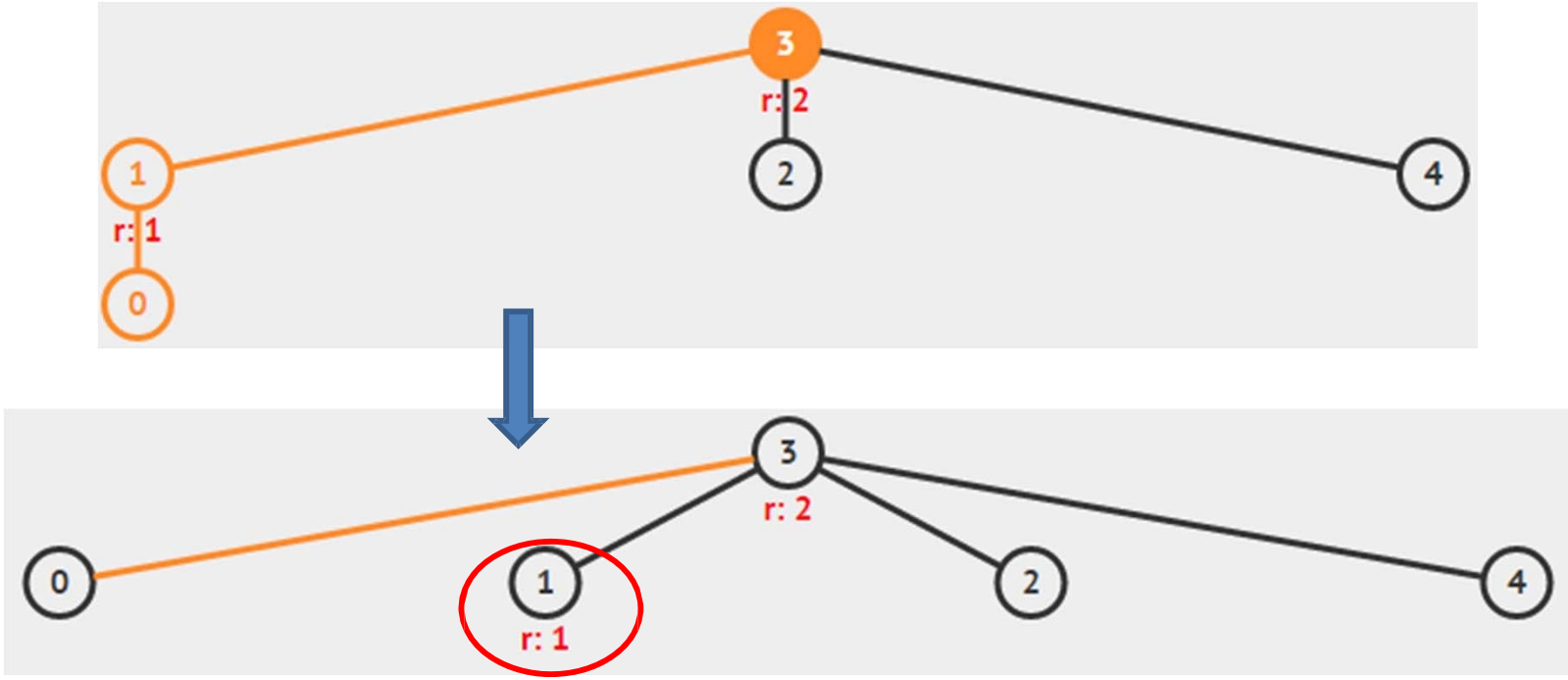


findSet(0)

```
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;
   }
}
```

Note, I have to first **set** the new value of **p[i]** before returning the new value, because method **set()** in **Java Vector** returns the element *previously at the* specified position, i.e. I cannot use:

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



UFDS – Summary

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

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

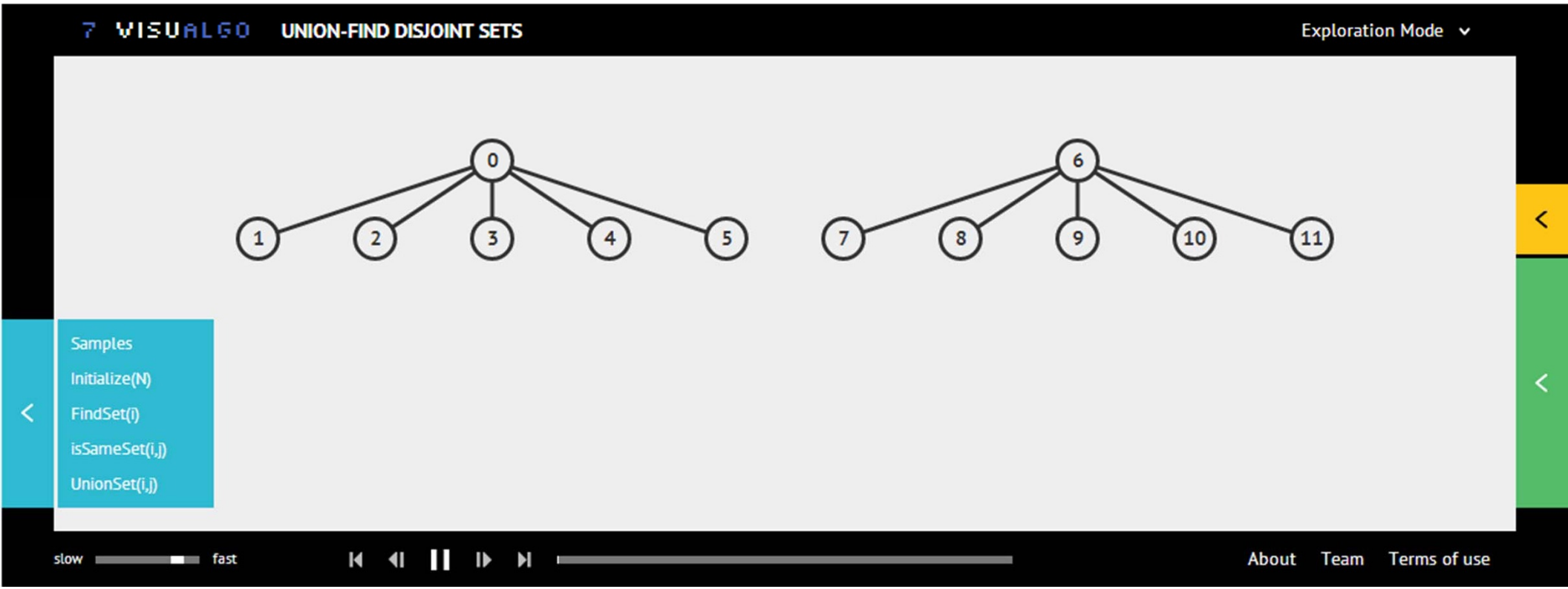
Further References:

- **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
- http://visualgo.net/ufds.html

VisuAlgo UFDS Exercise (1)

First, click "**Initialize(N)**", enter **12**, then click "**Go**"

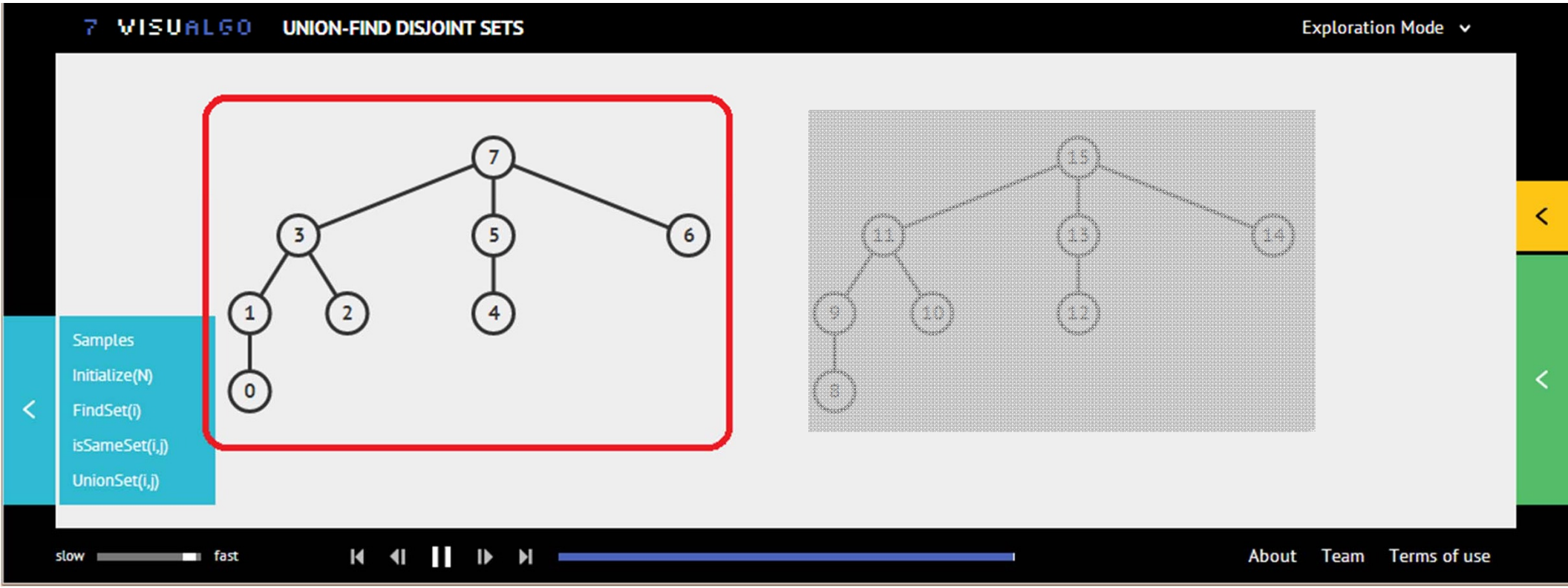
Do a sequence of union and/or find operations to get the screen shot below (**Samples: 2 Trees of Rank 1**)



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**)



Interesting way to model a lightweight (small) set of Booleans…

This DS will be revisited in Week 11 during lecture on TSP

http://visualgo.net**/bitmask.html**

CP3, Section 2.2

BITMASK DATA STRUCTURE

Lightweight (Small) Set of Booleans

An integer is stored in binary in computer memory

- int x = 7_{10} (base 10/decimal) is actually 111_2 (base 2/binary)
- int y = 12_{10} is 1100_2
- int z = 83_{10} is 1010011_2

We can use this sequence of 0s and 1s to represent a small set of Boolean very efficiently

- **N**-bits integer can represent **N** Boolean objects
  - Theoretically up to 32 Boolean objects for a 32-bit integer
  - If **i**-th bit is 1, we say object **i** is in the set/active/used
    Otherwise, object **i** is not in the set/not active/not used

Bit Operations (1)

To check whether bit **i** is on or off in **x**, we use **x & (1<<i)**

- Example:
  - x = 25_{10} (11001_2), check if bit i = **2** (from right, 0-based indexing) is on
  - x & (1<<**2**) = 25_{10} & $\mathbf{4}_{10}$

    11001_2

    $00\mathbf{1}00_2$

    -------- & (bitwise AND operation)

    $00\mathbf{0}00_2$
  - The result is $\mathbf{0}_{10}$ ($00\mathbf{0}00_2$), that means bit i = **2** (from right) is **off**

Bit Operations (2)

To check whether bit **i** is on or off in **x**, we use **x & (1<<i)**

- Example:
  - x = 25_{10} (11001_2), check if bit i = **3** (from right, 0-based indexing) is on
  - x & (1<<**3**) = 25_{10} & $\mathbf{8}_{10}$

    11001_2

    $0\mathbf{1}000_2$

    -------- & (bitwise AND operation)

    $0\mathbf{1}000_2$
  - The result is $\mathbf{8}_{10}$ ($0\mathbf{1}000_2$), that means bit i = **3** (from right) is **on**

Bit Operations (3)

To turn on bit **i** of an integer **x**, we use **x = x | (1<<i)**

- Example:
  - x = 25_{10} (11001_2), turn on bit i = **2** (from right, 0-based indexing)
  - x = x | (1<<**2**) = 25_{10} | $\mathbf{4}_{10}$ =

    11001_2

    $00\mathbf{1}00_2$

    -------- | (bitwise OR operation)

    $11\mathbf{1}01_2$
  - x = 29_{10} (11101_2) now, bit **2** (from right) is now turned on

Bit Operations (4)

To turn on bit **i** of an integer **x**, we use **x = x | (1<<i)**

- Example:
  - x = 25_{10} (11001_2), turn on bit i = **3** (from right, 0-based indexing)
  - x = x | (1<<**3**) = 25_{10} | $\mathbf{8}_{10}$ =

    11001_2

    01000_2

    -------- | (bitwise OR operation)

    11001_2
  - x = 25_{10} (11001_2), no change as bit **3** (from right) is already on

Bit Operations (5)

To turn on all bits of a set of Boolean with size **n**, we use **x = (1<<n) - 1**

- Example:
  - n = 4
  - 1<<4 = 16_{10} (10000_2)
  - x = (1<<4) - 1 = 15_{10} (1111_2)
    - Notice that all n = 4 bits are turned on

VisuAlgo Bitmask Exercise

First, click Click "**Set S**", enter **31064**, and click "**Go**"

Explore "**Set/Check bit**", enter **7**, and click "**Go**"

Then *self-explore*: Try other integer **S** and other integer **j**

Also, *self-explore* "**Toggle/Clear/Least Significant bit**"

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)

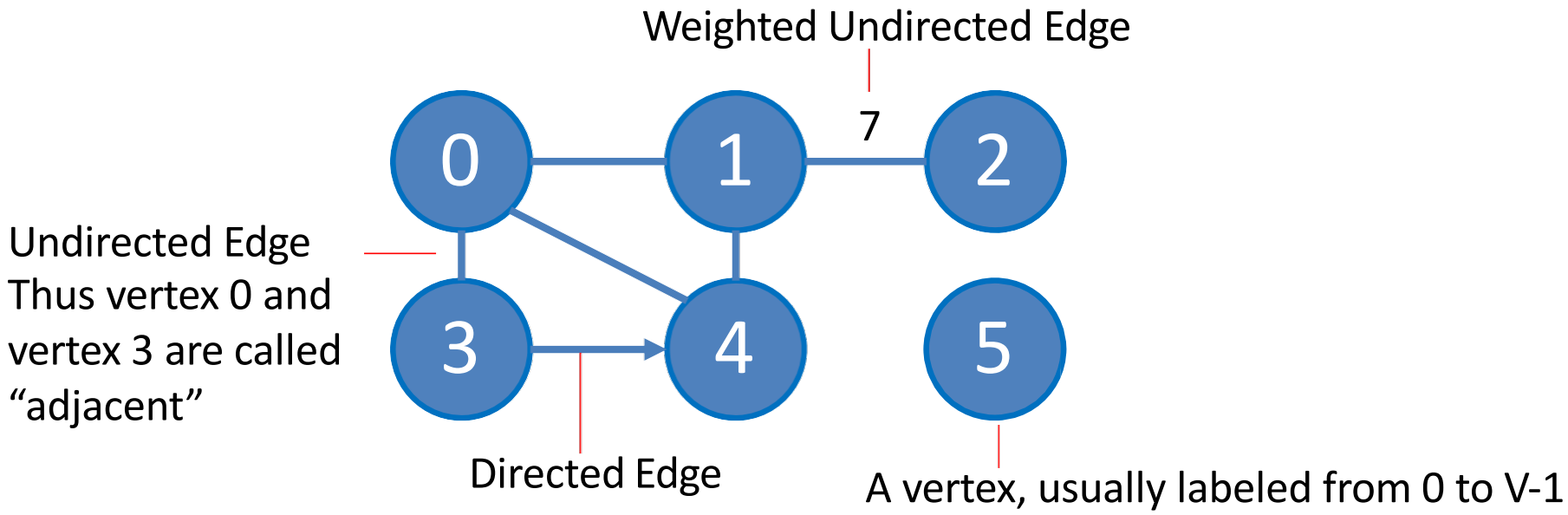
But in general graph, there is no notion of:

- 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

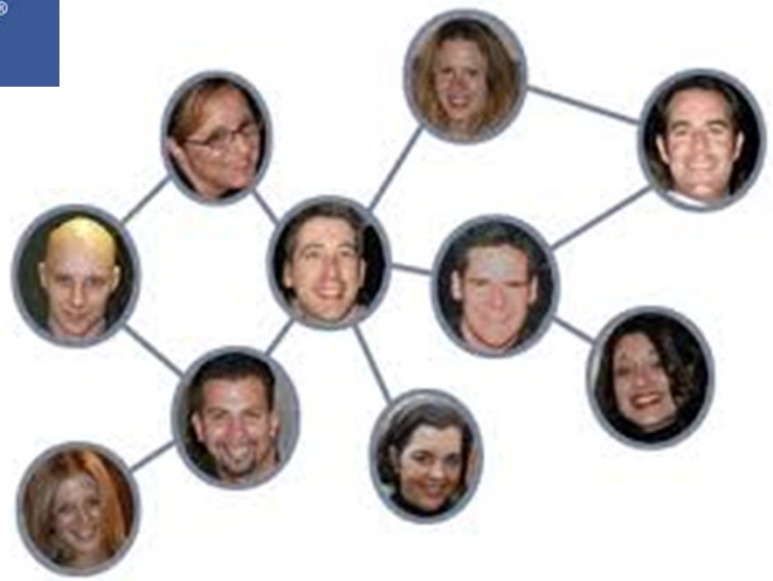
- We will ignore "multi graph" where there can be more than one edge between a pair of vertices



Social Network

facebook

twitter

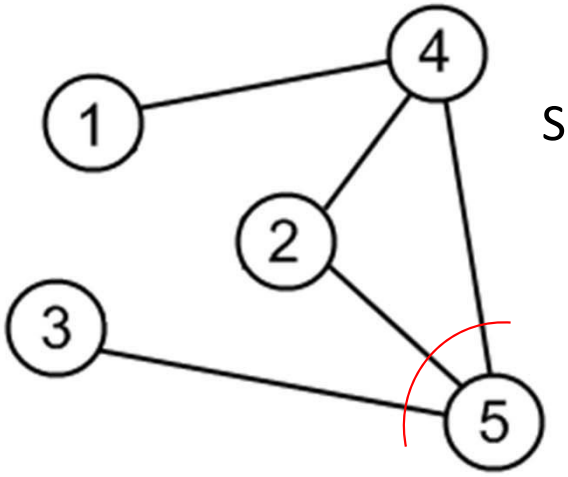


LinkedIn

Graph Terminologies (2)

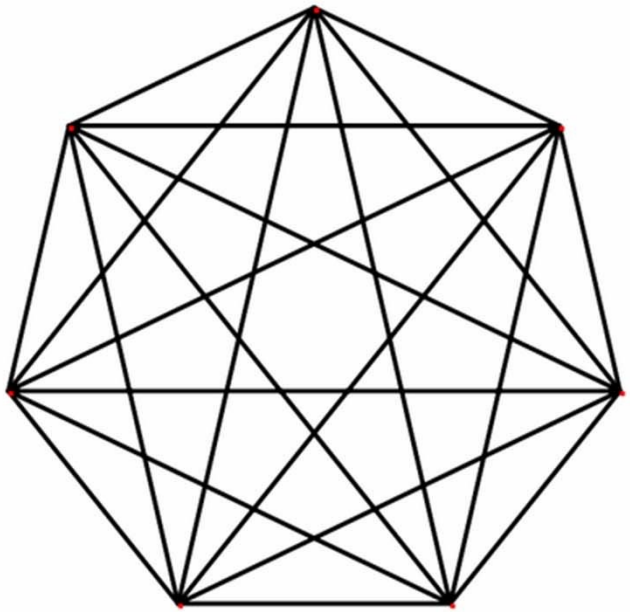
More terminologies (simple graph):

- 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

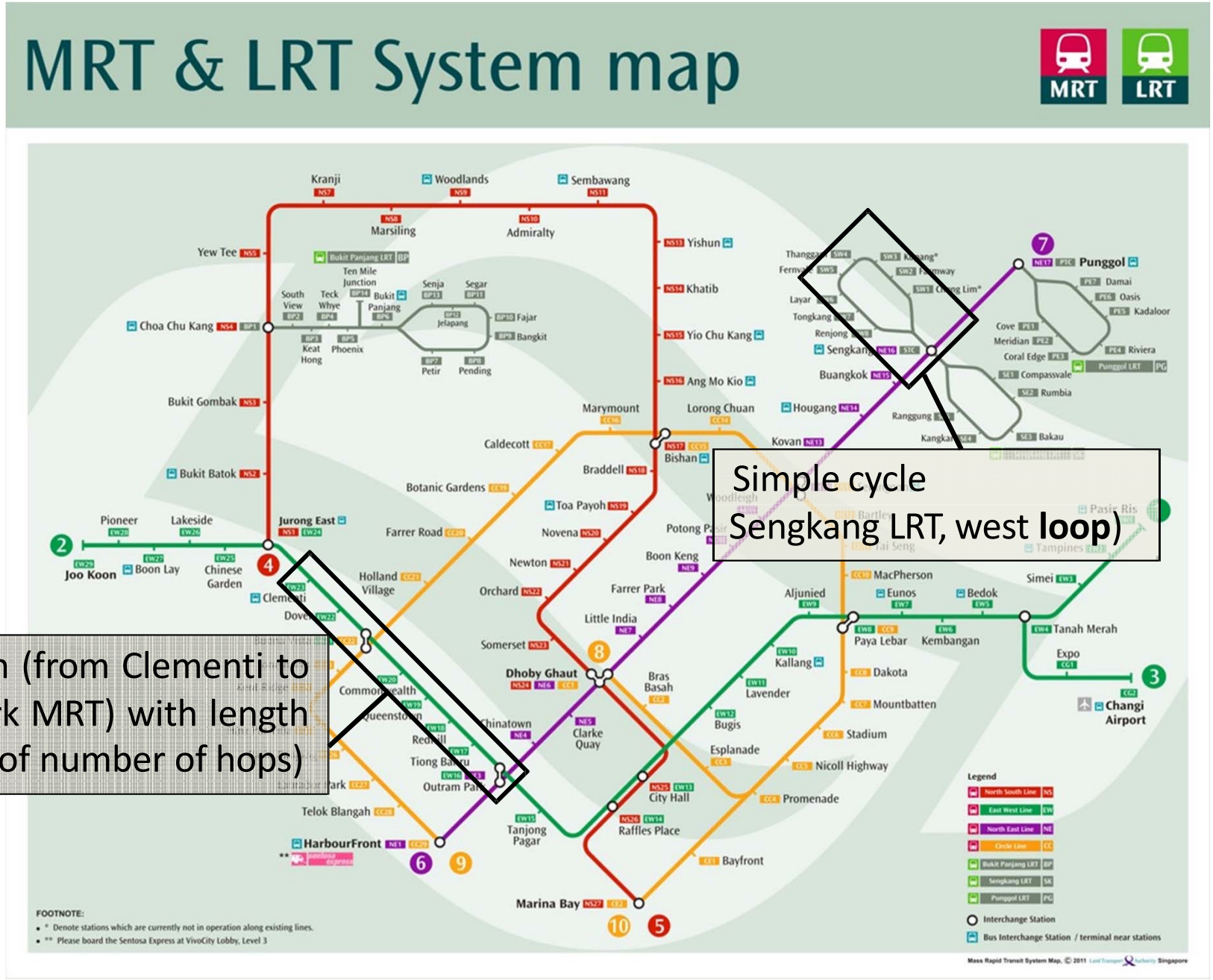
Complete Graph
7 vertices,
${}_{7}C_{2}$ = 21 edges

Graph Terminologies (3)

Yet more terminologies (example in the next slide):

- (Simple) Path
  - Sequence of vertices adjacent to each other
  - 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 vertex except start/end

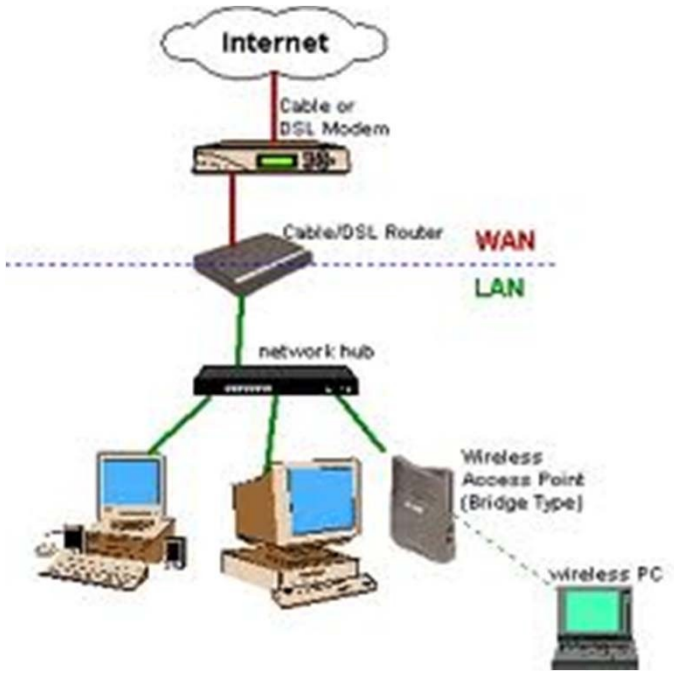
Transportation Network

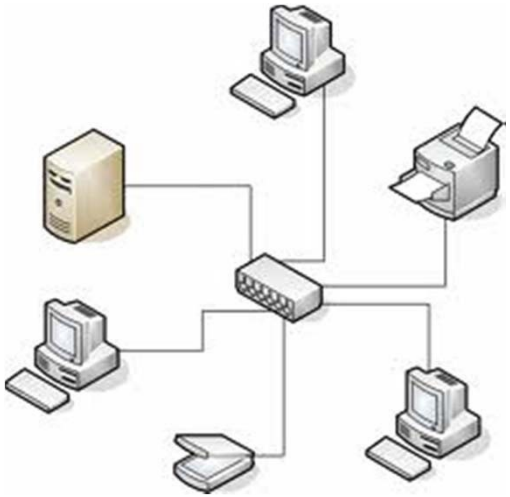


Simple cycle
Sengkang LRT, west **loop**)

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

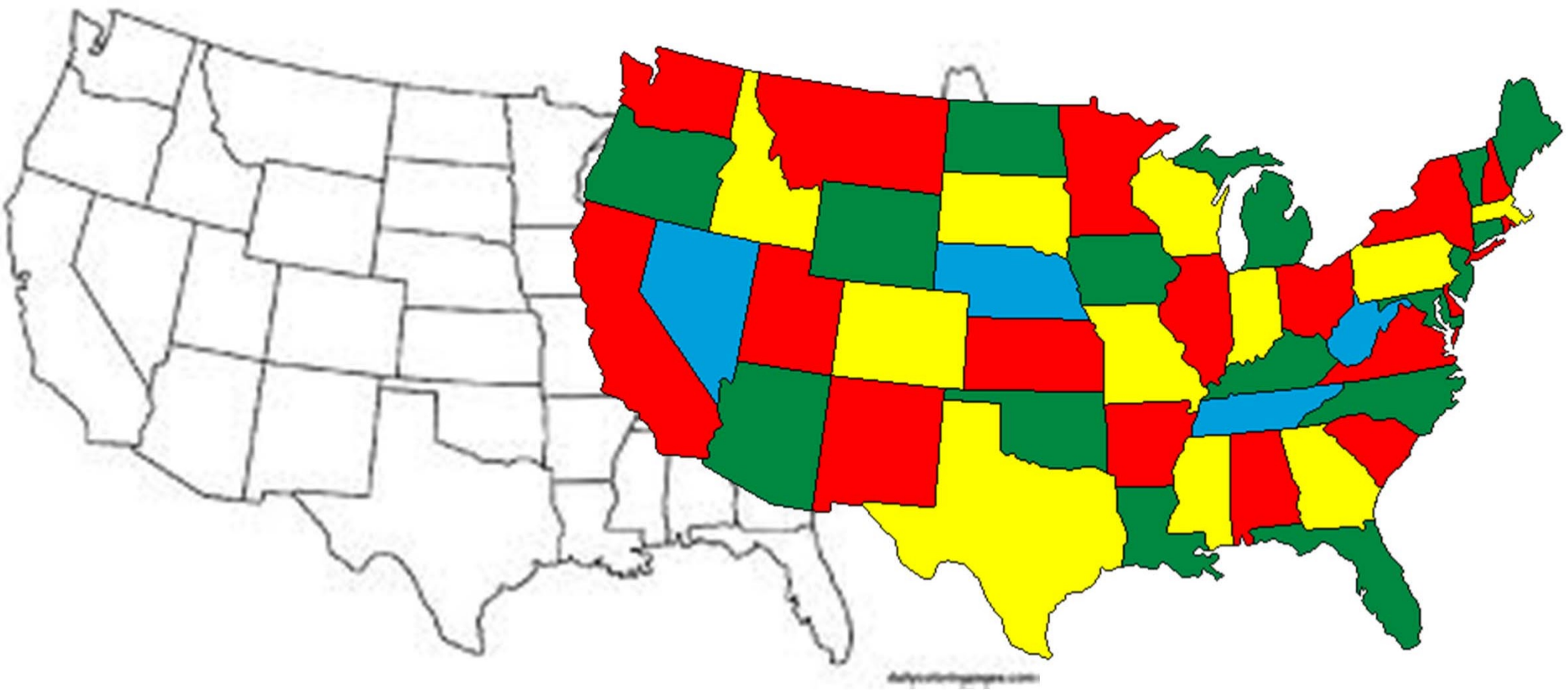
Internet / Computer Networks





Communication Network

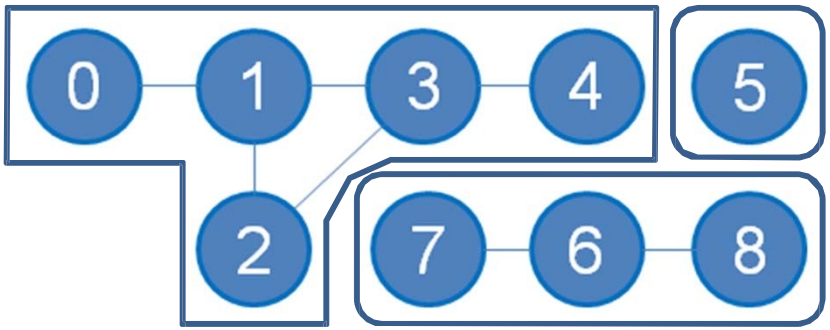
Optimization



Graph Terminologies (4)

Yet More Terminologies:

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



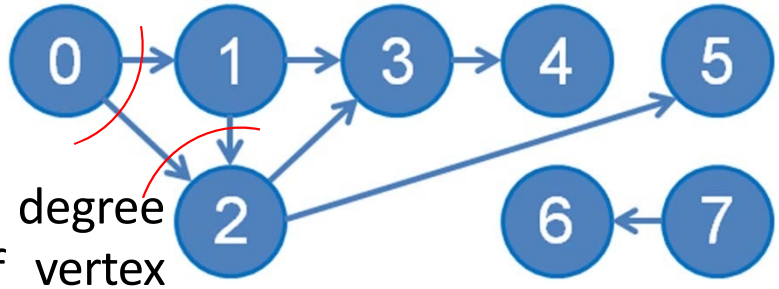
-There are 3 components in this graph
- Disconnected graph
(since it has > 1 component)
-Vertices 1-2-3-4 are reachable from vertex 0
-Vertices 5, 6-7-8 are unreachable from vertex 0
- {7-6-8} is a sub graph of this graph

Graph Terminologies (5)

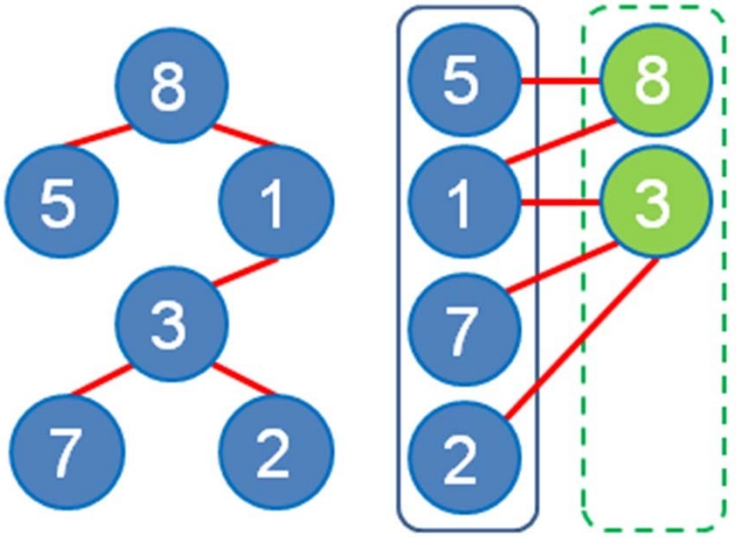
Yet More Terminologies:

- 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)
  - If we can partition the vertices into two sets so that there is no edge 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

http://visualgo.net**/graphds.html**

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

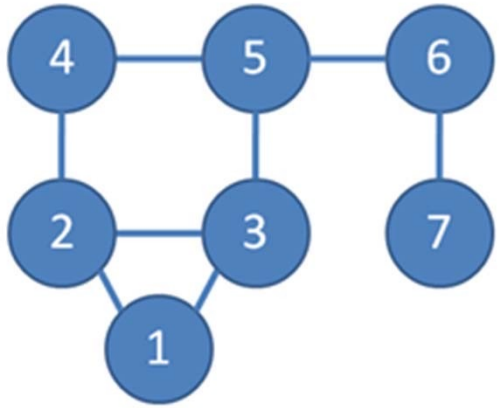
Format: a 2D array **AdjMatrix** (see an example below)

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



	1	2	3	4	5	6	7
1		1	1				
2	1		1	1			
3	1	1			1		
4		1			1		
5			1	1		1	
6					1		1
7						1	

Adjacency List

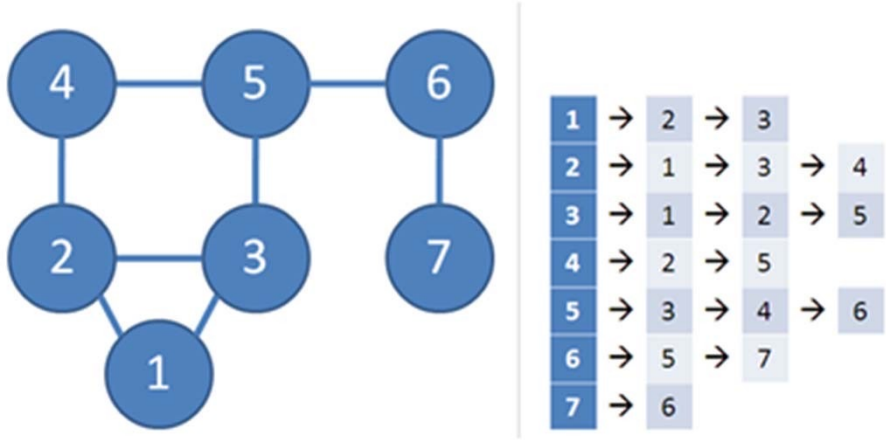
Format: array **AdjList** of **V** lists, one for each vertex

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

- For weighted graph, stores **pairs (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)**



Edge List

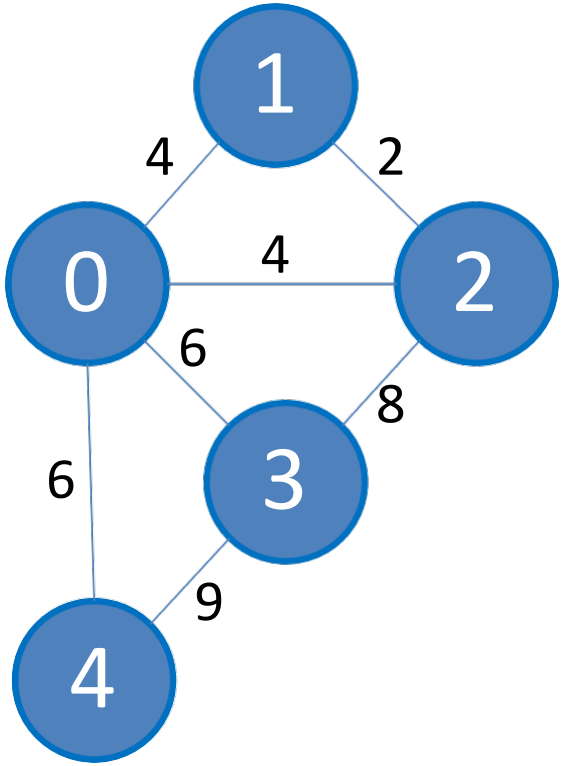
Format: array **EdgeList** of **E** edges

For each edge **i**, **EdgeList[i]** stores an (integer) triple {w(u, v), 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(V²)**



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

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

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

Edge List: Also with Java Collections framework

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

PS: This is *my* 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
```

Java Implementation (3)

We will use AdjList for most graph problems in CS2010

- Vector < Vector < IntegerPair > > AdjList;
  - Why do we use IntegerPair?
    - We need to store pair of information for each edge: (neighbor number, weight), weight = 0/unused for unweighted graph
  - Why do we use Vector of IntegerPairs?
    - For Vector's **auto-resize feature** ☺: If you have **k** neighbors of a vertex, just add **k** times to an initially empty Vector of IntegerPair of this vertex
    - You can replace this with Java List or ArrayList if you want to…
  - Why do we use Vector of Vectors of IntegerPairs?
    - For Vector's **indexing feature** ☺: if we want to enumerate neighbors of vertex u, use **AdjList.get(u)** to access the correct Vector of IntegerPairs

Summary

In this lecture, we looked at:

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

B. Bitmask Data Structure
   - for Week 11 later

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

D. How to store graph info in computer memory

PS: Online Quiz 1 next week includes UFDS, Bitmask, and Graph DS