# CS2351
# Data Structures

## Lecture 9:
## Basic Data Structures II

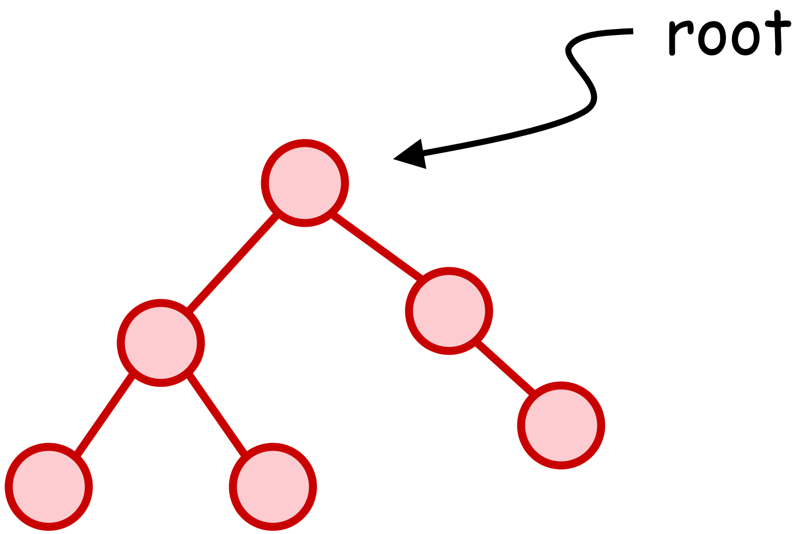
# About this lecture

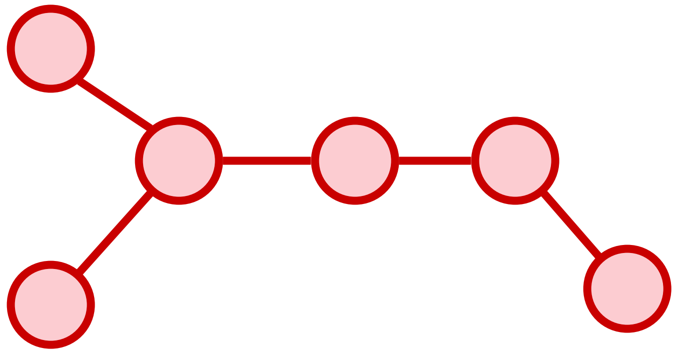
- A graph consists of a set of nodes and a set of edges joining the nodes
  - A tree is a special kind of graph, where there is one connected component, and that it contains no cycles

- In this lecture, we introduce how to store a tree, and how to store a graph

# Tree

# Classification of Trees



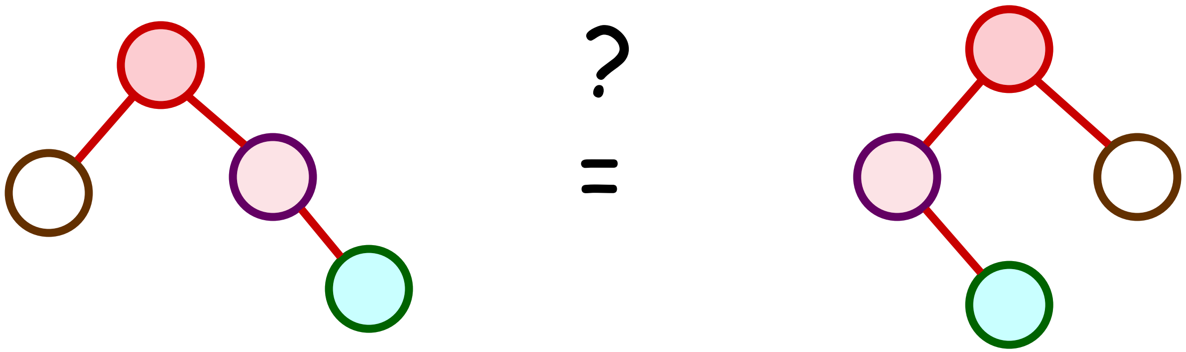
rooted

Each edge connects
a parent to a child

unrooted

No parent-child
relationship in an edge

# Classification of Rooted Trees



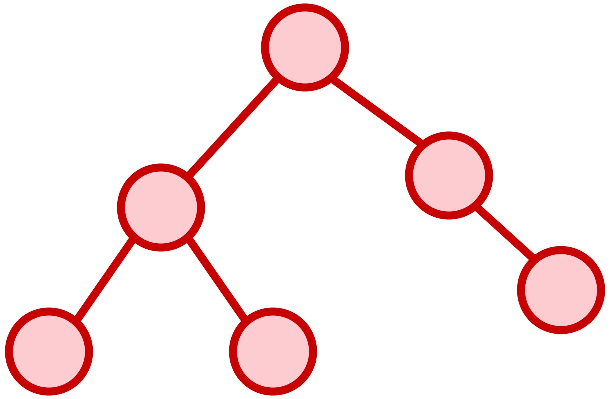

ordered

Has ordering
among children

unordered

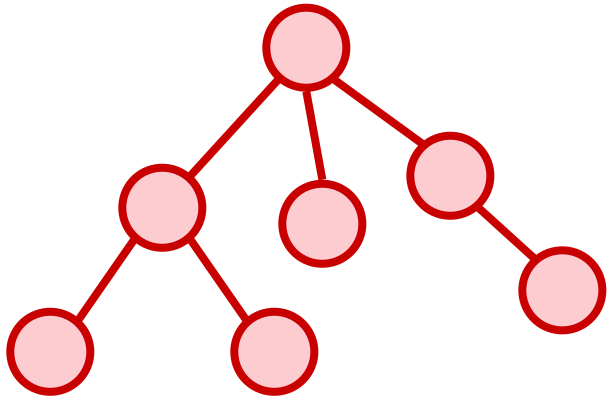
No ordering
among children

# Classification of Rooted Trees



binary

Each node has at most 2 children



non-binary

No restrictions

# Implementing an Ordered Rooted Binary Tree

- Each node contains pointers that point to the left child and the right child :

```
struct node {
   ...
   struct node *left, *right ;
} ;
```

# Implementing an Ordered Rooted Binary Tree

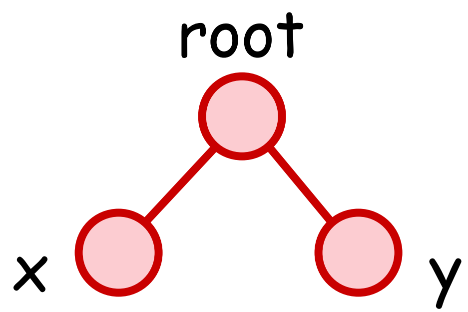
- Also, each node may contain some info
- Ex: In a search tree for a set of integers, each node contains an integer key

```
struct node {
  int key ;
  struct node *left, *right ;
} ;
```

# Implementing an Ordered Rooted Binary Tree

- Once the definition of a node is done, we can create a tree

```
struct node root, x, y ;
root.left = &x ;
root.right = &y ;
x.left = x.right = y.left = y.right = NULL;
```



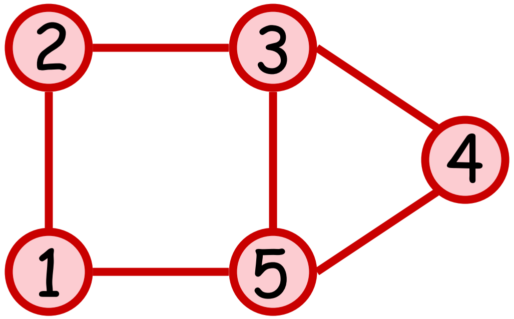
# Remarks

- It is easy to modify the definition of a node to implement a rooted non-binary tree (how?)
- Sometimes, we may also want to store a pointer from a node to its parent, so as to speed up movement in a tree

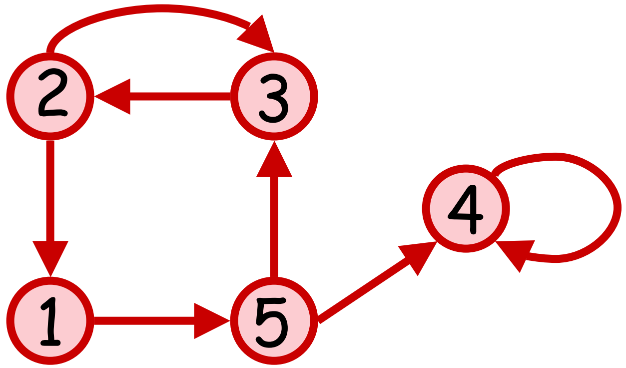
```
struct node {
  int key ;
  struct node *left, *right, *parent;
} ;
```

# Graph

# Graph



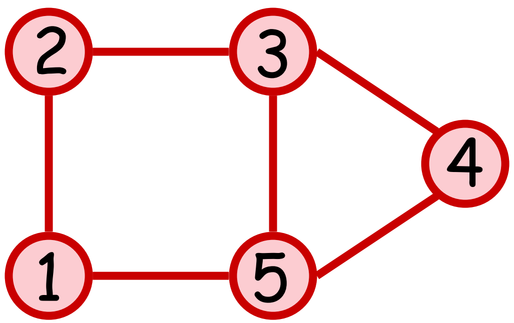

undirected

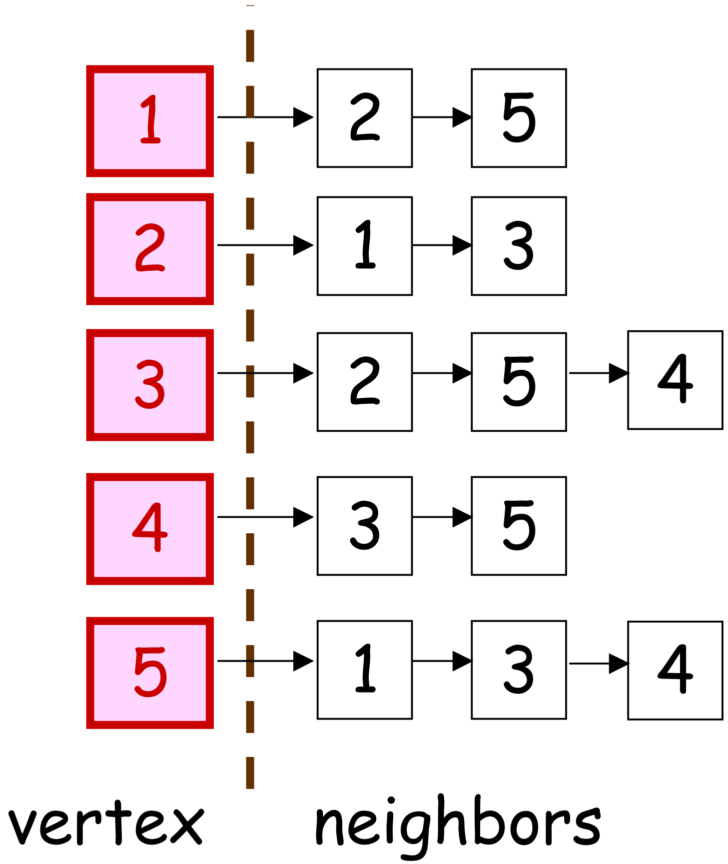



directed

# Adjacency List (1)

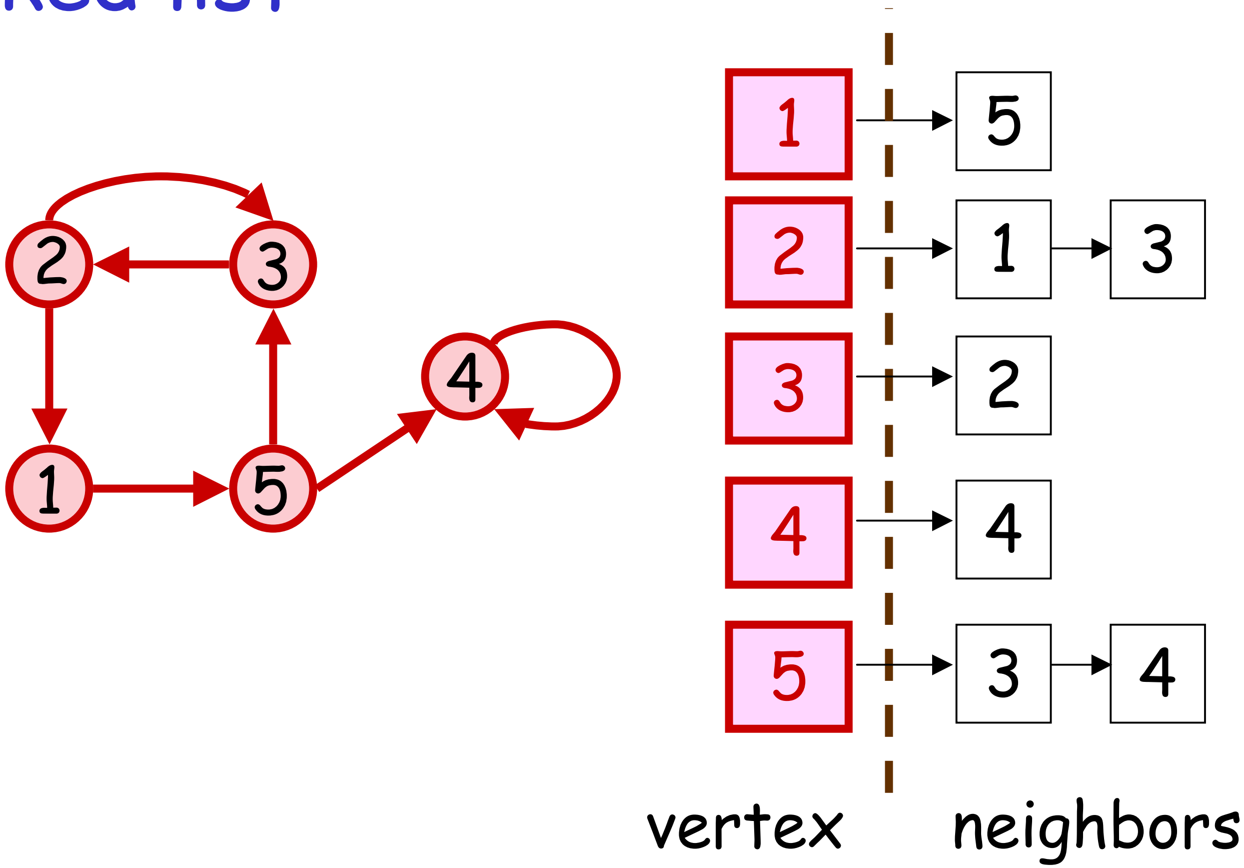
- For each vertex u, store its neighbors in a linked list



# Adjacency List (2)

- For each vertex u, store its neighbors in a linked list



# Adjacency List (3)

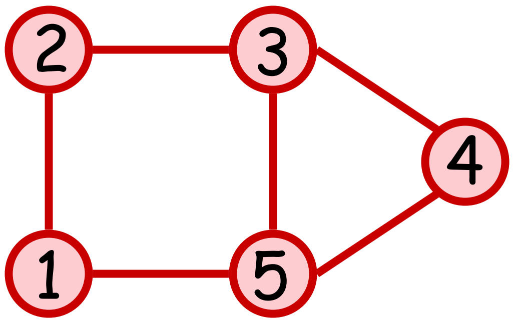
- Let $G = (V, E)$ be an input graph
- Using Adjacency List representation :
  - Space : $O(|V| + |E|)$

    ➔ Excellent when $|E|$ is small
  - Easy to list all neighbors of a vertex
  - Takes $O(|V|)$ time to check if a vertex $u$ is a neighbor of a vertex $v$
- can also represent weighted graph

# Adjacency Matrix (1)

- Use a $|V| \times |V|$ matrix $A$ such that

$$A(u,v) = 1 \quad \text{if} \quad (u,v) \text{ is an edge}$$
$$A(u,v) = 0 \quad \text{otherwise}$$

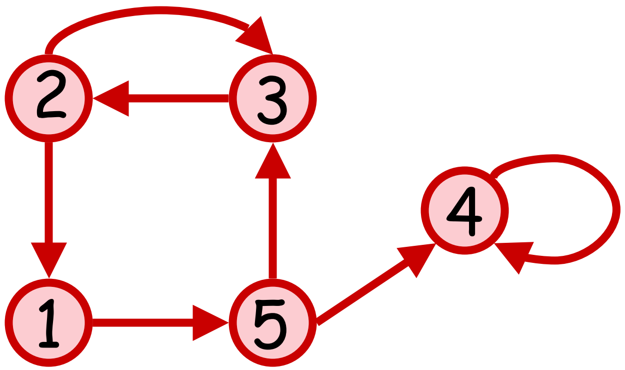



| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 | 1 |
| 2 | 1 | 0 | 1 | 0 | 0 |
| 3 | 0 | 1 | 0 | 1 | 1 |
| 4 | 0 | 0 | 1 | 0 | 1 |
| 5 | 1 | 0 | 1 | 1 | 0 |

# Adjacency Matrix (2)

- Use a $|V| \times |V|$ matrix $A$ such that

$$A(u,v) = 1 \quad \text{if } (u,v) \text{ is an edge}$$
$$A(u,v) = 0 \quad \text{otherwise}$$




| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 0 | 0 | 0 | 1 |
| 2 | 1 | 0 | 1 | 0 | 0 |
| 3 | 0 | 1 | 0 | 0 | 0 |
| 4 | 0 | 0 | 0 | 1 | 0 |
| 5 | 0 | 0 | 1 | 1 | 0 |

# Adjacency Matrix (3)

- Let $G = (V, E)$ be an input graph
- Using Adjacency Matrix representation :
  - Space : $O(|V|^2)$

    ➔ Bad when $|E|$ is small
  - $O(1)$ time to check if a vertex $u$ is a neighbor of a vertex $v$
  - $\Theta(|V|)$ time to list all neighbors
- can also represent weighted graph