## Today's topics

- Graphs
- BFS

# Assignment-3

- void post_a_msg(char msg[MAX_MSG_LEN], struct record *r)
  - Insert msg in the trie
  - Update history in the square node
  - push a reference to the square node in the stack of posts by the user(r)

- int delete_latest_post(struct record *r)
  - Pop the latest post from the stack of posts by the user(r)
  - Update history in the square node

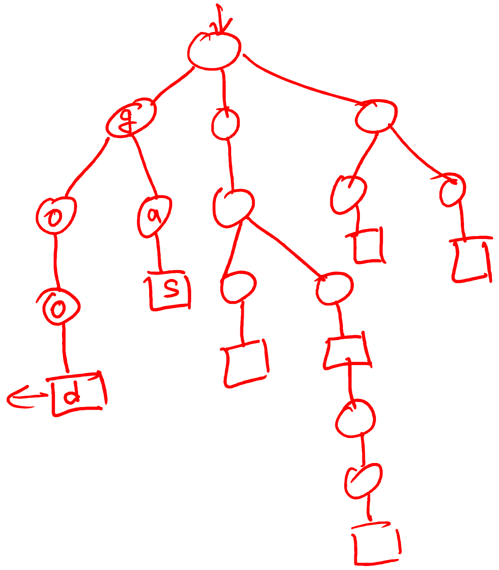
- int read_latest_post(struct record *r, char msg[MAX_MSG_LEN])
  - The top element in the stack of posts contains a reference to the latest post
  - Use the parent field in the trie node to read the message

struct record * , ~~char~~ * msg

## Post

u1

u2



## Delete

## Read

## Assignment-3

- void delete_all_posts(struct record *r)
- void cleanup_history(struct record *r)

- void destroy_trie()
- struct list_events* get_history(char msg[MAX_MSG_LEN])
- struct list_events* get_clean_history(char msg[MAX_MSG_LEN])

## Assignment-3

- Clean history

| |
|---|
| USER-3 deleted |
| USER-1 deleted |
| USER-4 posted |
| |
| USER-1 posted |
| USER-4 posted |
| USER-3 posted |
| USER-1 deleted |
| USER-3 posted |
| USER-2 posted |
| USER-1 posted |
| USER-1 posted |

# Assignment-3

- Clean history

| |
|---|
| ~~USER-3 deleted~~ |
| ~~USER-1 deleted~~ |
| USER-4 posted |
| ~~USER-3 posted~~ |
| ~~USER-1 posted~~ |
| USER-4 posted |
| USER-3 posted |
| ~~USER-1 deleted~~ |
| USER-3 posted |
| USER-2 posted |
| ~~USER-1 posted~~ |
| USER-1 posted |

## Assignment-3

- Cleanup history

| |
|---|
| USER-4 posted |
| USER-4 posted |
| USER-3 posted |
| USER-3 posted |
| USER-2 posted |
| USER-1 posted |

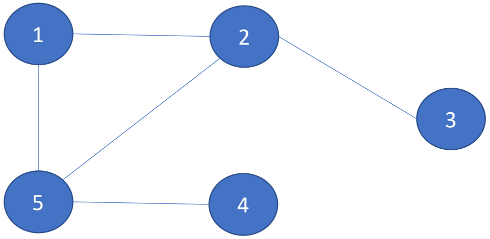
## Assignment-3

- void delete_all_posts(struct record *r)
- void destroy_trie()
- struct list_events* get_history(char msg[MAX_MSG_LEN])
- struct list_events* get_clean_history(char msg[MAX_MSG_LEN])

## References

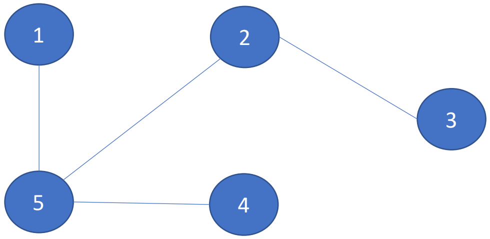
- Read chapter-20 of the CLRS book
- Read chapter-6 from Goodrich and Tamassia book

## Tree



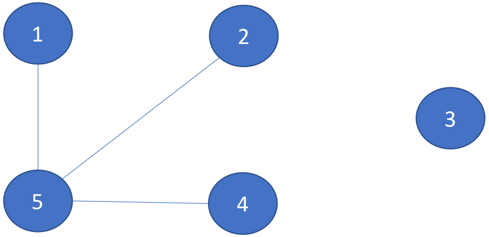
- A tree is connected graph without cycles
  - No notion of a root node, unlike the rooted tree that we discussed earlier
- Is this a tree? No

## Tree



- A tree is connected graph without cycles
  - No notion of a root node, unlike the rooted tree that we discussed earlier
- Is this a tree? Yes

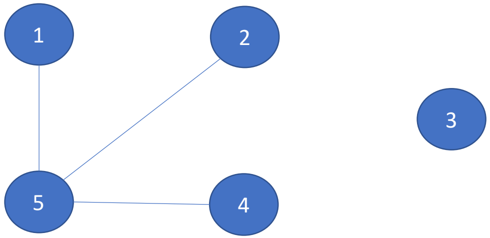
## Tree



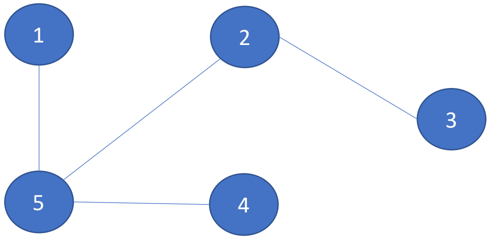
- A tree is connected graph without cycles
  - No notion of a root node, unlike the rooted tree that we discussed earlier
- Is this a tree? No

## Forest

- A forest is a collection of trees
- Is this a forest? Yes

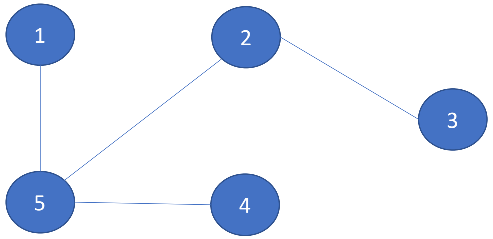


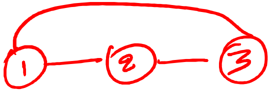
## Spanning tree



- Every connected graph has a spanning tree
- A spanning tree of a connected graph is a spanning subgraph that is also a tree

# Spanning tree





- Let G be an undirected graph with n vertices and m edges
- Prove that if G is a tree, then m = n-1

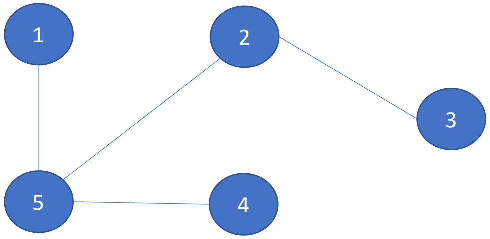
P(j):
Let's say m = n-1 is true for $2 \leq j \leq i$

Let's consider a tree with $i+1$ edges. $i \geq 2$

. Does this tree has a leaf

The proof is shown on the next slide.

# Spanning tree



- Let G be an undirected graph with n vertices and m edges
- Prove that if G is a tree, then the statement P(n): m = n-1 holds

Proof by induction:
Base case n = 2
Number of edges = 1 == n-1, thus P(2) holds

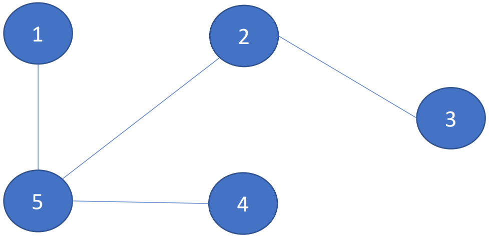
Show that if P(j) holds for every 2 <= j <= i, P(i+1) also holds

Let's look at a tree with i+1 vertices
Because a tree has no cycle, it must have a leaf node

After removing the leaf node, the rest of the tree with i vertices is still going to be all connected

The number of edges in the disconnected tree with i vertices is i-1 because P(i) holds
After adding the leaf node back, the number of edges in the tree with i+1 vertices is i
Thus P(i+1) also holds.
Hence proved by induction.

## Spanning tree



- Let G be an undirected graph with n vertices and m edges
- Prove that if G is connected, then m >= n-1

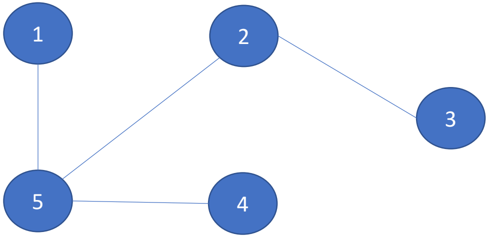
P : $\underline{m < n-1}$

G is not a tree because of the earlier proof

therefore, G must be a connected graph

The proof is shown on the next slide.

# Spanning tree



- Let G be an undirected graph with n vertices and m edges
- Prove that if G is connected, then m >= n-1

Proof by contradiction:
Let's assume that G is connected if m < n-1
G can't be a tree because the number of edges in a tree with n vertices is n-1, as proved earlier
It means that G has cycles
We can remove cycles from G while keeping it connected

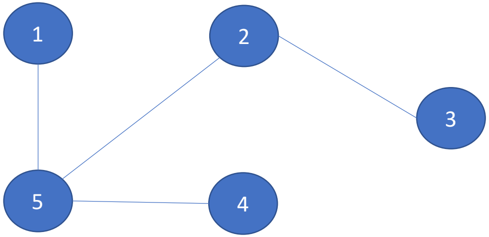
If we remove all cycles while keeping the graph connected, it becomes a tree

The number of edges after removing the cycles is less than m, and thus less than n-1

However, the number of nodes in a tree must be n-1

Hence, proved by contradiction.

## Spanning tree



- Let G be an undirected graph with n vertices and m edges
- Prove that if G is forest, then m <= n-1

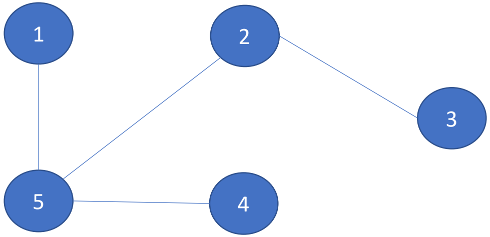
P: $m > n-1$

for one tree it doesn't hold

for two trees:

The proof is shown on the next slide.

## Spanning tree



- Let G be an undirected graph with n vertices and m edges
- Prove that if G is forest, then m <= n-1

Proof by contradiction:
Let's assume that G is a forest if m > n-1
Let's say the forest contains only one tree
The assumption doesn't hold because a tree has n-1 nodes

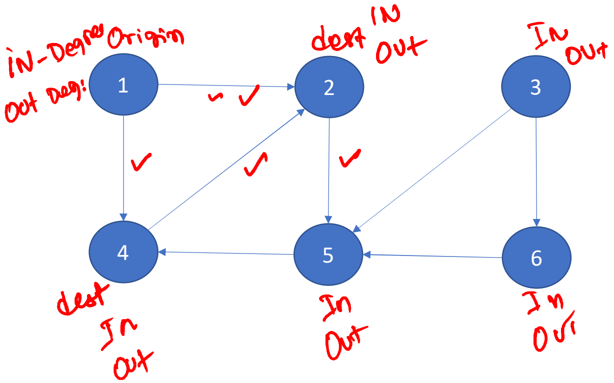
If the forest contains two trees
We can make the forest a tree by adding an edge between both trees
Therefore, the total number of edges in the final tree is m+1, i.e., greater than n-1
But the number of nodes in a tree is n-1. Therefore the assumption doesn't hold for a forest with two trees.

Let's say there are k trees in the forest, where k >= 2. We can combine all trees to create one tree by adding k-1 additional edges.
Therefore, the number of edges in the final tree is m+k-1, which is greater than n-1.
However, the number of nodes in a tree is n-1
Hence proved by contradiction.

## Terminology

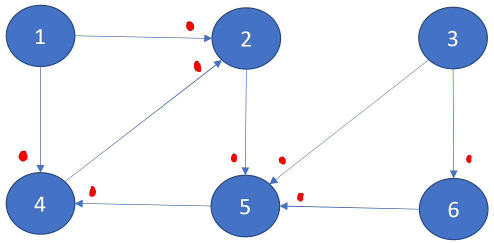
- For a graph G = (V, E), we will use |V| for number of vertices and |E| for number of edges

# Terminology (directed graph)



- origin and destination of an edge
- Incoming edges of a vertex v
    - all edges whose destination is v
- Outgoing edges of a vertex v
    - all edges whose origin is v
- in-degree
    - number of incoming edges
- out-degree
    - number of outgoing edges

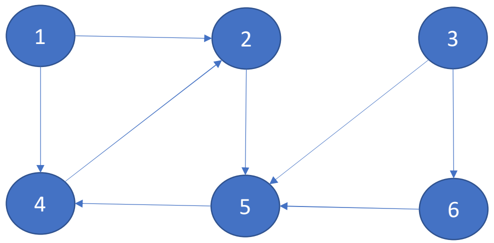
## Terminology (directed graph)



$$\sum_{v \in V} in\_degree(v) = |E|$$

While computing the sum of all in degrees, we are counting every edge only once. Therefore, the sum of the in degrees of all nodes is |E|.

## Terminology (directed graph)



$$\sum_{v \in V} out\_degree(v) = |E|$$

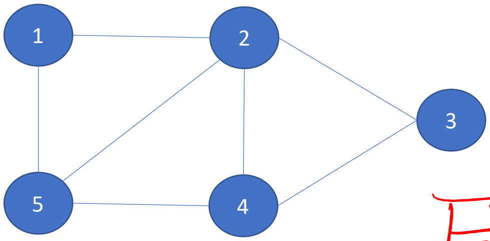
While computing the sum of all out degrees, we are counting every edge only once. Therefore, the sum of the in degrees of all nodes is |E|.

## Applications of graphs

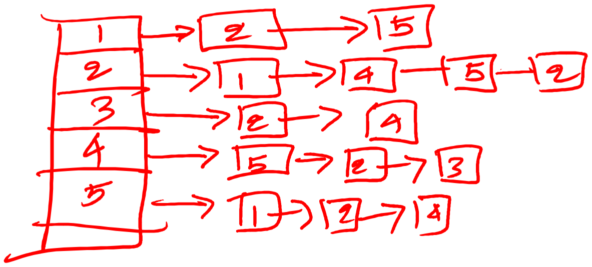
- Graphs are used in many real-world applications
  - Google Maps use graphs
    - Vertices are intersections of roads
    - An edge is a road connecting two intersections
  - Airline routes can be represented using graphs
    - Each airport is a vertex
    - Two airports are connected using an edge if there is a nonstop flight between them
    - We would like the graph to be connected such that it is possible to reach from any airport to any other airport
  - Web pages can be stored as vertices in a graph
    - Vertices A and B are connected using a directed edge if A contains a reference to B

## Representations of graphs

## Representation of graphs



- How can we store a graph?
  - walk all vertices
  - walk all edges
  - determine whether an edge exists between two given vertices
  - traverse all outgoing edges from a vertex

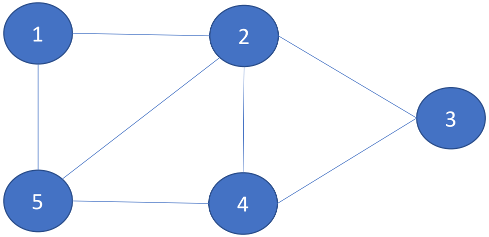


We can store a graph like a tree, in which every node is a vertex, and the children of a node are all the outgoing edges. The children can be stored in a linked list. A node in the graph contains a reference to the first children. The problem with this approach is that in order to walk all vertices, we may need to walk all edges, which could be very expensive. Another approach is to store all vertices in an array, where vertices are numbered from 1 to |V| or 0 to |V|-1. An entry corresponding to a vertex in the array also contains a reference to the list of vertices connected using outgoing edges. This representation is also called the adjacency list, as we discuss next.

## Adjacency list

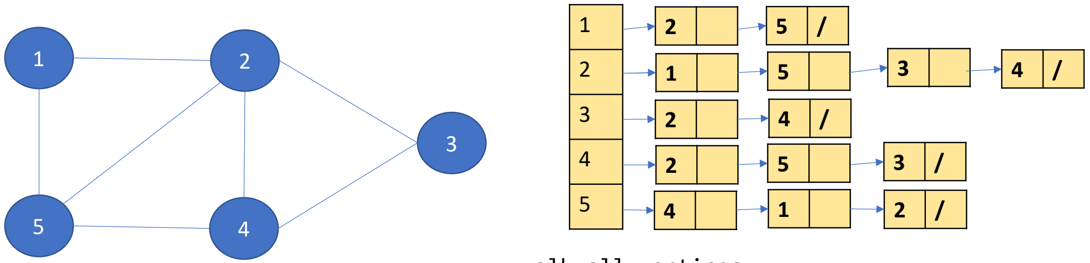
- In the adjacency list representation, for every vertex, we keep a list of all the adjacent vertices
- The size of a list is the degree of the corresponding vertex
- The total space requirement for graph G = (V, E) is $O(|E|+|V|)$
- For undirected graphs, each edge (u,v) appears in two lists corresponding to the vertices u and v

## Adjacency list (undirected)



- `walk all vertices`
- `walk all edges`
- `determine whether an edge exists between two given vertices`
- `traverse all outgoing edges from a vertex`
- `Space requirement`

# Adjacency list (undirected)



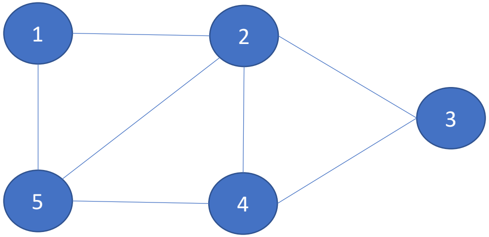
- `walk all vertices`
- `walk all edges`
- `determine whether an edge exists between two given vertices`
- `traverse all outgoing edges from a vertex`
- `Space requirement`

In an adjacency list representation, the vertices are stored in an array. Each vertex points to a linked list that contains all the adjacent vertices. The space requirement for the linked list is the sum of degrees of all vertices, i.e., 2*|E|. The space requirement for the array of vertices is |V|. Therefore, the total space requirement is O(|V| + |E|). Walk all vertices requires O(|V|) operations. Walk all edges requires O(|E|) operations. Determining whether an edge exists between vertex u and v requires minimum(degree(u), degree(v)) operations. Traverse all outgoing edges from a vertex v requires degree(v) operations. This representation is better for sparse graphs, where |E| is much less than $|V|^2$.

## Adjacency matrix

- In the adjacency matrix representation, we store edges in a two-dimensional array of size n x n, where n is the number of vertices
- If A is an adjacency matrix, A[u][v] is set to one if there is an edge between u and v; otherwise, A[u][v] is set to zero
- The space requirement for graph G= (V, E) is $O(|V|^2)$

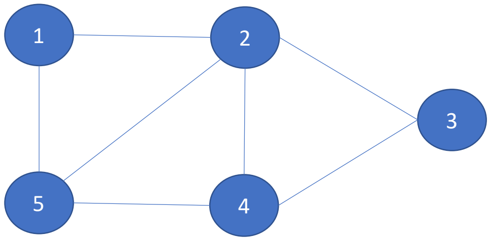
# Adjacency matrix (undirected)



- walk all vertices
- walk all edges
- determine whether an edge exists between two given vertices
- traverse all outgoing edges from a vertex
- Space requirement

$$\begin{array}{c|c|c|c|c|c|}
 & 1 & 2 & 3 & 4 & 5 \\ \hline
1 & & 1 & & & 1 \\ \hline
2 & 1 & & 1 & 1 & 1 \\ \hline
3 & & 1 & & 1 & \\ \hline
4 & & 1 & 1 & & 1 \\ \hline
5 & 1 & 1 & & 1 & \\ \hline
\end{array}$$

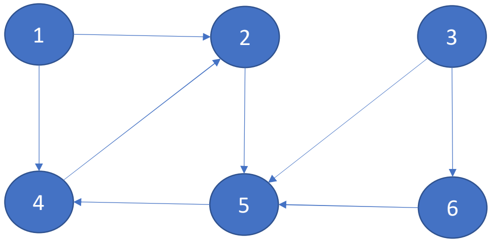
## Adjacency matrix (undirected)



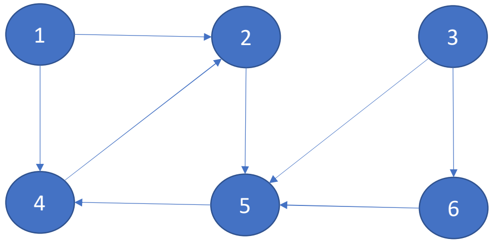
| | 1 | 2 | 3 | 4 | 5 |
|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 0 | 1 |
| 2 | 1 | 0 | 1 | 1 | 1 |
| 3 | 0 | 1 | 0 | 1 | 0 |
| 4 | 0 | 1 | 1 | 0 | 1 |
| 5 | 1 | 1 | 0 | 1 | 0 |

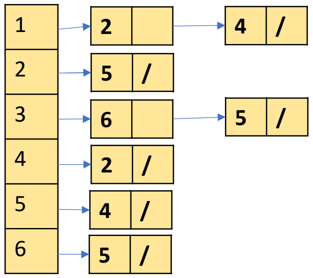
In an adjacency matrix representation, the edges are stored in 2 dimensional |V| x |V| array. If A is an adjacency matrix, A[u][v] is set to one if there is an edge between u and v; otherwise, A[u][v] is set to zero. The space requirement for the two-dimensional array is $O(|V|^2)$. Walking all vertices requires O(|V|) operations. Walking all edges requires $O(|V|^2)$ operations. Determining whether an edge exists between vertex u and v requires O(1) operations. Traverse all outgoing edges from a vertex v requires O(|V|) operations. This representation is better for dense graphs, where |E| is close to $|V|^2$.

## Adjacency list (directed)



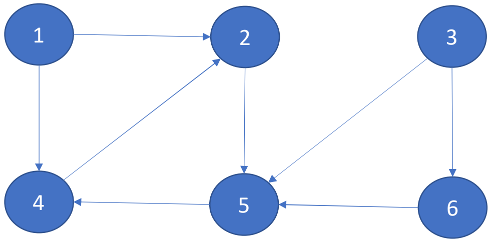
## Adjacency list (directed)



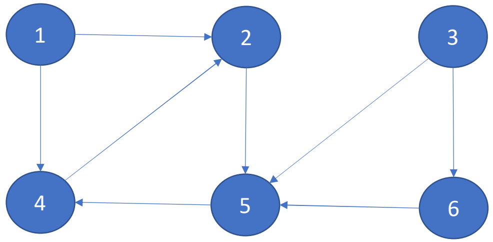


In a directed adjacency list, only outgoing edges are stored in the linked list. The number of linked list nodes in an adjacency list is |E|. Checking if there is an edge between two vertices u, v can be done in O(out_degree(u)+out_degree(v)).

# Adjacency matrix (directed)



# Adjacency matrix (directed)



|   | 1 | 2 | 3 | 4 | 5 | 6 |
|---|---|---|---|---|---|---|
| 1 | 0 | 1 | 0 | 1 | 0 | 0 |
| 2 | 0 | 0 | 0 | 0 | 1 | 0 |
| 3 | 0 | 0 | 0 | 0 | 1 | 1 |
| 4 | 0 | 1 | 0 | 0 | 0 | 0 |
| 5 | 0 | 0 | 0 | 1 | 0 | 0 |
| 6 | 0 | 0 | 0 | 0 | 1 | 0 |

Only outgoing edges are stored in the linked list in a directed adjacency matrix.

## Adjacency matrix vs. adjacency list

- Checking if there is an edge between two vertices can be done in O(1)
- Adjacency-matrix is beneficial for dense graphs in which |E| is close to $|V|^2$
- Space requirement is $O(|V|^2)$
- Iterating all outgoing edges from a vertex v requires O(|V|) operations

- Checking if there is an edge between two vertices u, v can be done in O(out_degree(u)+out_degree(v)), in a directed graph and min(degree(u), degree(v)) in an undirected graph
- Adjacency-list is beneficial for sparse graphs in which the |E| is much less than $|V|^2$
- Space requirement is O(|V|+|E|)
- Iterating all outgoing edges from a vertex v requires O(degree(v)) operations