Graphs

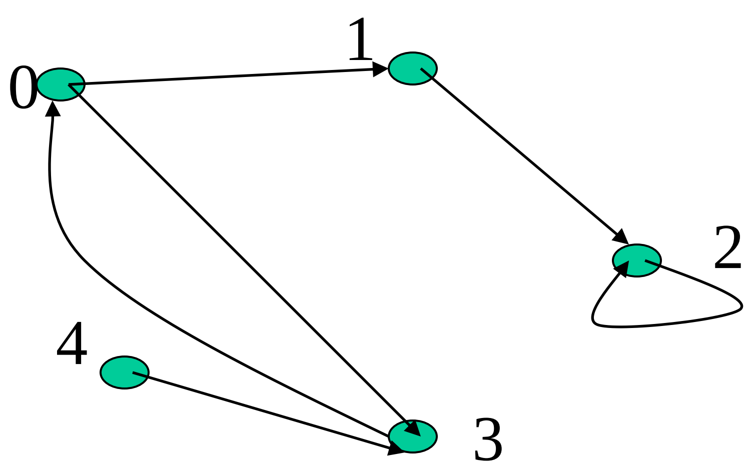
- Definition of Graphs and Related Concepts
- Representation of Graphs
- The Graph Class
- Graph Traversal
- Graph Applications

Definition of Graphs

- A graph is a finite set of nodes with edges between nodes
- Formally, a graph G is a structure (V,E) consisting of
  – a finite set V called the set of nodes, and
  – a set E that is a subset of VxV. That is, E is a set of pairs of the form (x,y) where x and y are nodes in V

Examples of Graphs

- V={0,1,2,3,4}
- E={(0,1), (1,2), (0,3), (3,0), (2,2), (4,3)}

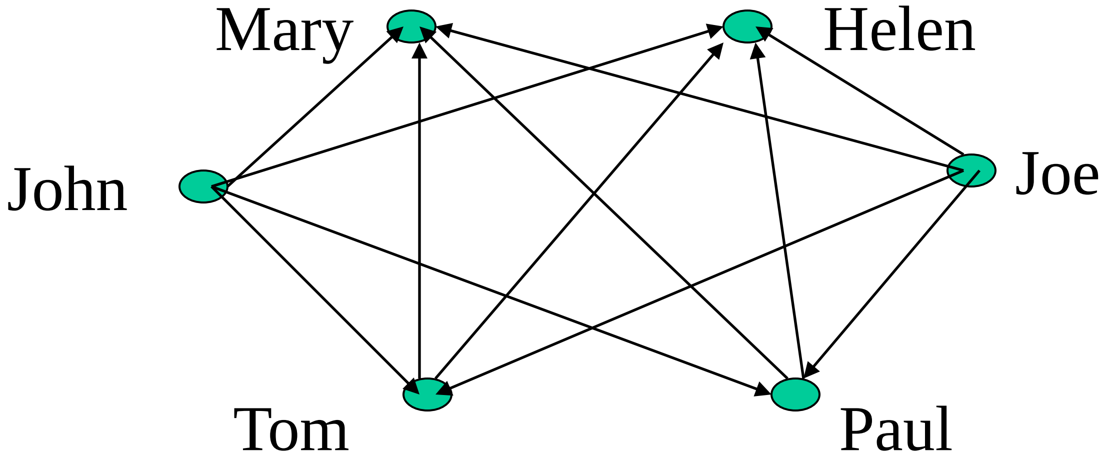


When (x,y) is an edge, we say that x is *adjacent to* y, and y is *adjacent from* x.

0 is adjacent to 1.
1 is not adjacent to 0.
2 is adjacent from 1.

A "Real-life" Example of a Graph

- V=set of 6 people: John, Mary, Joe, Helen, Tom, and Paul, of ages 12, 15, 12, 15, 13, and 13, respectively.
- E ={(x,y) | if x is younger than y}



Intuition Behind Graphs

- The nodes represent entities (such as people, cities, computers, words, etc.)
- Edges (x,y) represent relationships between entities x and y, such as:
  - “x loves y”
  - “x hates y”
  - “x is a friend of y” (note that this not necessarily reciprocal)
  - “x considers y a friend”
  - “x is a child of y”
  - “x is a half-sibling of y”
  - “x is a full-sibling of y”
- In those examples, each relationship is a different graph

Graph Representation

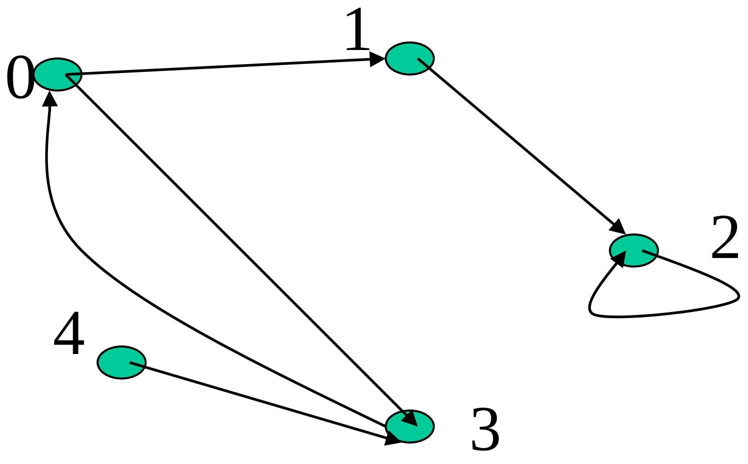
- For graphs to be computationally useful, they have to be conveniently represented in programs
- There are two computer representations of graphs:
  - Adjacency matrix representation
  - Adjacency lists representation

Adjacency Matrix Representation

- In this representation, each graph of n nodes is represented by an n x n matrix A, that is, a two-dimensional array A
- The nodes are (re)-labeled 1,2,…,n
- A[i][j] = 1 if (i,j) is an edge
- A[i][j] = 0 if (i,j) is not an edge

Example of Adjacency Matrix

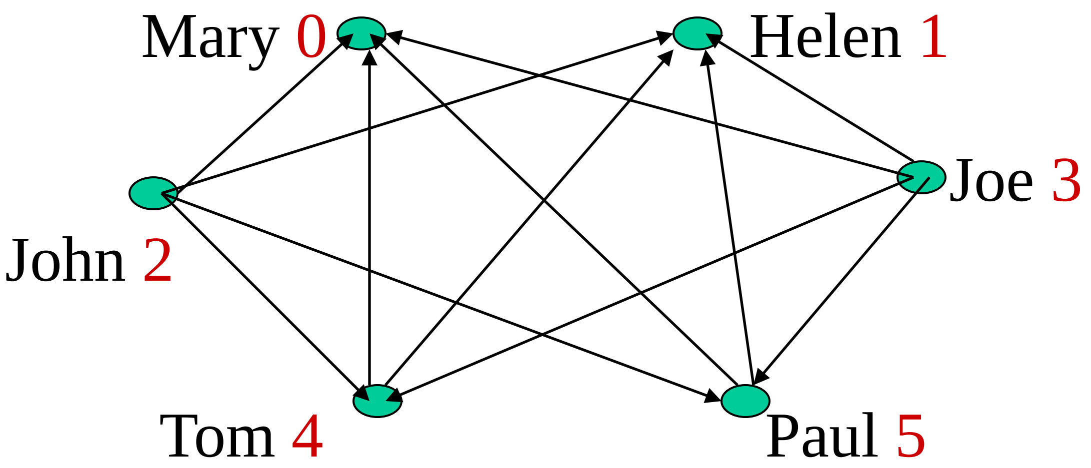
$$A = \begin{bmatrix} 0 & 1 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 4 & 0 \end{bmatrix}$$



Another Example of Adj. Matrix

- Re-label the nodes with numerical labels

$$A = \begin{bmatrix} 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 1 & 1 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \end{bmatrix}$$



Pros and Cons of Adjacency Matrices

- Pros:
  - Simple to implement
  - Easy and fast to tell if a pair (i,j) is an edge: simply check if A[i][j] is 1 or 0
- Cons:
  - No matter how few edges the graph has, the matrix takes $O(n^2)$ in memory

Adjacency Lists Representation

- A graph of n nodes is represented by a one-dimensional array L of linked lists, where
  - L[i] is the linked list containing all the nodes adjacent from node i.
  - The nodes in the list L[i] are in no particular order

Example of Linked Representation

L[0]: empty

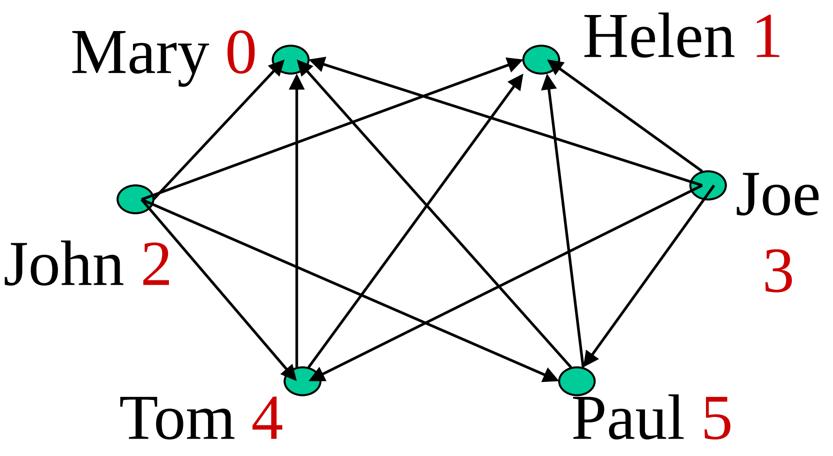
L[1]: empty

L[2]: 0, 1, 4, 5

L[3]: 0, 1, 4, 5

L[4]: 0, 1

L[5]: 0, 1



Pros and Cons of Adjacency Lists

- Pros:
  - Saves on space (memory): the representation takes as many memory words as there are nodes and edge.
- Cons:
  - It can take up to O(n) time to determine if a pair of nodes (i,j) is an edge: one would have to search the linked list L[i], which takes time proportional to the length of L[i].

The Graph Class

```
class Graph {
  public:
     typedef int  datatype;
     typedef datatype * datatypeptr;
     Graph( int n=0); // creates a graph of n nodes and no edges
     bool isEdge( int i, int j);
     void setEdge( int i, int j, datatype x);
     int getNumberOfNodes(){return numberOfNodes;};
private:
     datatypeptr *p;      //a 2-D array, i.e., an adjacency matrix
     int numberOfNodes;
};
```

Graph Class Implementation

```
Graph::Graph( int n){
    assert(n>=0);
    numberOfNodes=n;
    if (n==0)   p=NULL;
    else{
       p = new datatypeptr[n];
       for (int i=0;i<n;i++){
           p[i] = new datatype[n];
           for (int j=0;j<n;j++)
               p[i][j]=0;
       }
    }
};
```

```
bool Graph::isEdge(int i, int j){
        assert(i>=0 && j>=0);
        return p[i][j] != 0;
};
```

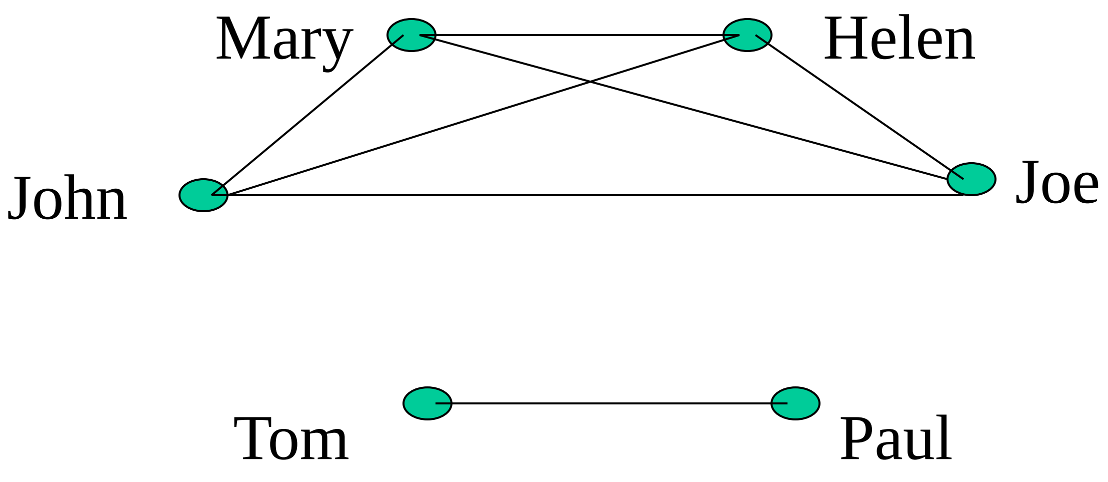
```
void Graph::;setEdge(int i,
                int j, datatype x){
        assert(i>=0 && j>=0);
        p[i][j]=x;
};
```

Directed vs. Undirected Graphs

- If the directions of the edges matter, then we show the edge directions, and the graph is called a *directed graph* (or a *digraph*)
- The previous two examples are digraphs
- If the relationships represented by the edges are symmetric (such as (x,y) is edge if and only if x is a sibling of y), then we don’t show the directions of the edges, and the graph is called an *undirected graph*.

Examples of Undirected Graphs

- V=set of 6 people: John, Mary, Joe, Helen, Tom, and Paul, where the first 4 are siblings, and the last two are siblings
- E ={(x,y) | x and y are siblings}



if (x,y) is an edge: we say that x is *adjacent to* y, & y adjacent to x. We also say that x and y are *neighbors*

Representations of Undirected Graphs

- The same two representations for directed graphs can be used for undirected graphs
- Adjacency matrix A:
  - A[i][j]=1 if (i,j) is an edge; 0 otherwise
- Adjacency Lists:
  - L[i] is the linked list containing all the neighbors of i

Example of Representations

Linked Lists:

L[0]: 1, 2, 3

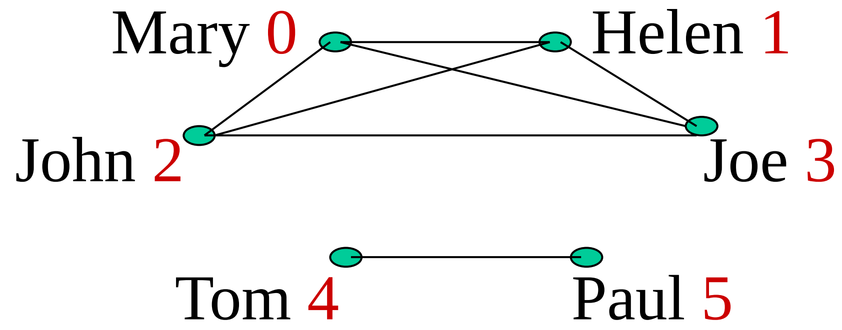
L[1]: 0, 2, 3

L[2]: 0, 1, 3

L[3]: 0, 1, 2

L[4]: 5

L[5]: 4



Adjacency Matrix:

$$A = \begin{bmatrix} 0 & 1 & 1 & 1 & 0 & 0 \\ 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 0 \\ 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{bmatrix}$$

Definition of Some Graph Related Concepts

- Let G be a directed graph
  - The *indegree* of a node x in G is the number of edges coming to x
  - The *outdegree* of x is the number of edges leaving x.
- Let G be an undirected graph
  - The *degree* of a node x is the number of edges that have x as one of their end nodes
  - The *neighbors* of x are the nodes adjacent to x

Things for You To Do

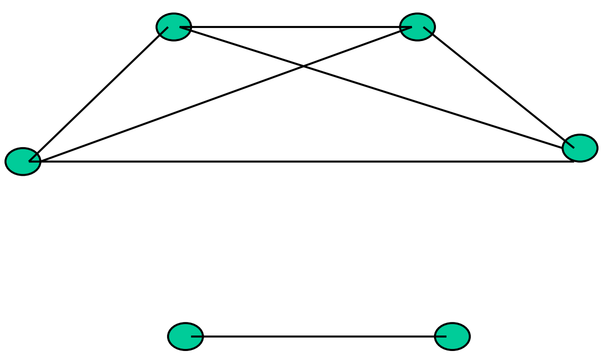
- Add a member function to the class graph, called getIndegree(int x), which returns the indegree of node x
- Add a member function to the class graph, called getOutdegree(int x), which returns the outdegree of node x

Paths

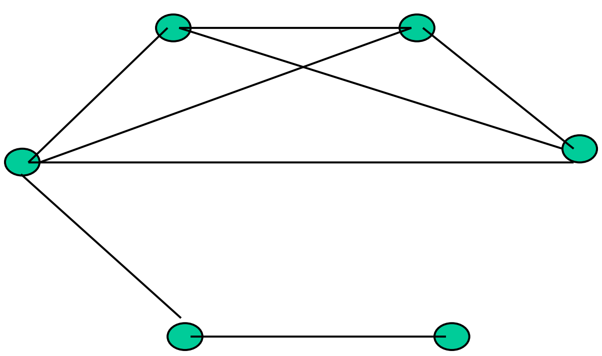
- A path in a graph G is a sequence of nodes x_1, x_2, …,x_k, such that there is an edge from each node the next one in the sequence
- For example, in the first example graph, the sequence 3, 0, 1, 2 is a path, but the sequence 0, 3, 4 is not a path because (0,3) is not an edge
- In the “sibling-of” graph, the sequence **John, Mary, Joe, Helen** is a path, but the sequence **Helen, Tom, Paul** is not a path

Graph Connectivity

- An undirected graph is said to be *connected* if there is a path between every pair of nodes. Otherwise, the graph is *disconnected*
- Informally, an undirected graph is connected if it hangs in one piece



Disconnected



Connected

Connected Components

- If an undirected graph is not connected, then each "piece" is called a connected component.
  - A piece in itself is connected, but if you bring any other node to it from the graph, it is no longer connected
- If the graph is connected, then the whole graph is one single connected component
- Of Interest: Given any undirected graph G,
  - Is G connected?
  - If not, find its connected components.

Graph Traversal Techniques

- The previous connectivity problem, as well as many other graph problems, can be solved using graph traversal techniques
- There are two standard graph traversal techniques:
  - *Depth-First Search* (DFS)
  - *Breadth-First Search* (BFS)

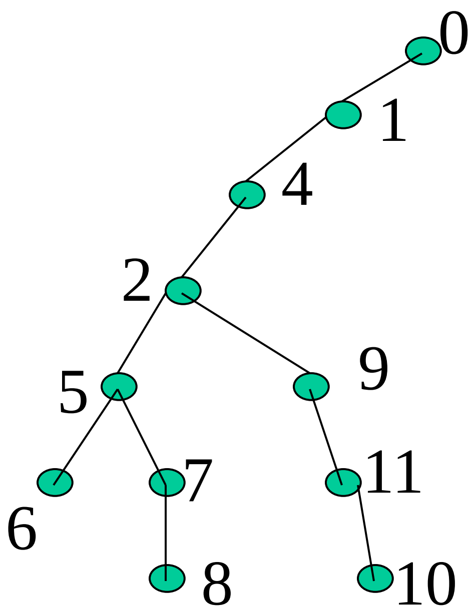
Graph Traversal (Contd.)

- In both DFS and BFS, the nodes of the undirected graph are visited in a systematic manner so that every node is visited exactly one.
- Both BFS and DFS give rise to a tree:
  - When a node x is visited, it is labeled as visited, and it is added to the tree
  - If the traversal got to node x from node y, y is viewed as the parent of x, and x a child of y

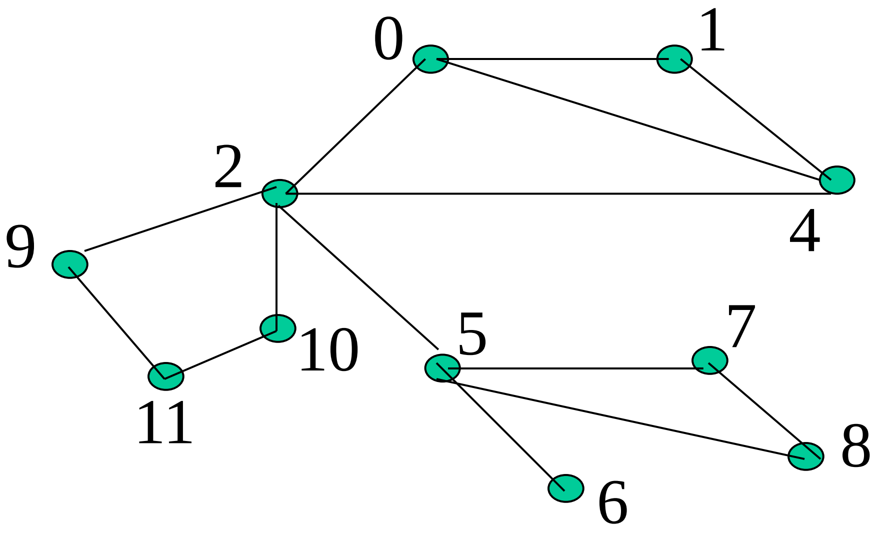
Depth-First Search

- DFS follows the following rules:
  1. Select an unvisited node x, visit it, and treat as the current node
  2. Find an unvisited neighbor of the current node, visit it, and make it the new current node;
  3. If the current node has no unvisited neighbors, backtrack to the its parent, and make that parent the new current node;
  4. Repeat steps 3 and 4 until no more nodes can be visited.
  5. If there are still unvisited nodes, repeat from step 1.

Illustration of DFS



DFS Tree



Graph G

Implementation of DFS

- Observations:
  - the last node visited is the first node from which to proceed.
  - Also, the backtracking proceeds on the basis of "last visited, first to backtrack too".
  - This suggests that a stack is the proper data structure to remember the current node and how to backtrack.

Illustrate DFS with a Stack

- We will redo the DFS on the previous graph, but this time with stacks
- In Class

DFS (Pseudo Code)

```
DFS(input: Graph G) {
    Stack S; Integer x, t;
    while (G has an unvisited node x){
        visit(x); push(x,S);
        while (S is not empty){
            t := peek(S);
            if (t has an unvisited neighbor y){
                visit(y); push(y,S); }
            else
                pop(S);
        }
    }
}
```

C++ Code for DFS

```
int * dfs(Graph G){ // returns a parent array representing the DFS tree
        int n=G.getNumberOfNodes();
        int * parent = new int[n];
        Stack S(n);  bool visited[n];
        for ( int i=0; i<n; i++)      visited[i]=false;
        int x=0;// begin DFS from node 0
        int numOfConnectedComponents=0;
        while (x<n){       // begin a new DFS from x
          numOfConnectedComponents++;
          visited[x]=true;  S.push(x); parent[x] = -1; // x is root
          while(!S.isEmpty())       // traverse the current piece
                // insert here the yellow box from the next slide
           x= getNextUnvisited(visited,n,x);
        }
        cout<<"Graph has "<< numOfConnectedComponents<<
                              " connected components\n";
        return p;
}
```

```
{
    int t=S.peek( );
    int y=getNextUnvisitedNeighbor(
                    t,G,visited,n);
    if (y<n){
        visited[y]=true;
        S.push(y);
        parent[y]=t;
    }
    else    S.pop( );
}
```

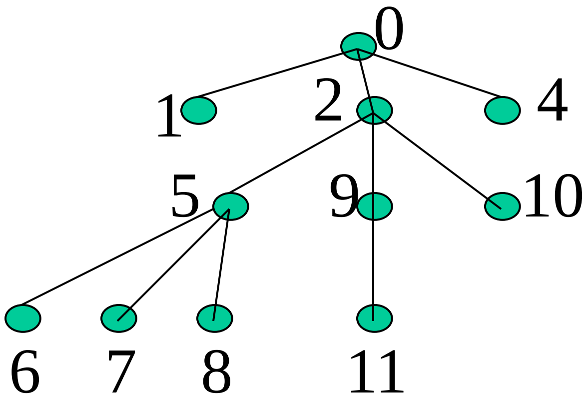
```
// Put this before dfs(…)
// returns the leftmost unvisited
// neighbor of node t. If none
// remains, returns n.
int getNextUnvisitedNeighbor(int t,
      graph G,  bool visited[],int n){
   for (int j=0;j<n;j++)
     if (G.isEdge(t,j) && !visited[j])
           return j;
   // if no unvisited neighbors left:
   return n;
}
```

```
//Put this before dfs(…). This returns the next unvisited node, or n otherwise
int getNextUnvisited(bool visited[],int n, int lastVisited){
    int j=lastVisited+1;
    while (visited[j] && j<n)    j++;
    return j;
}
```

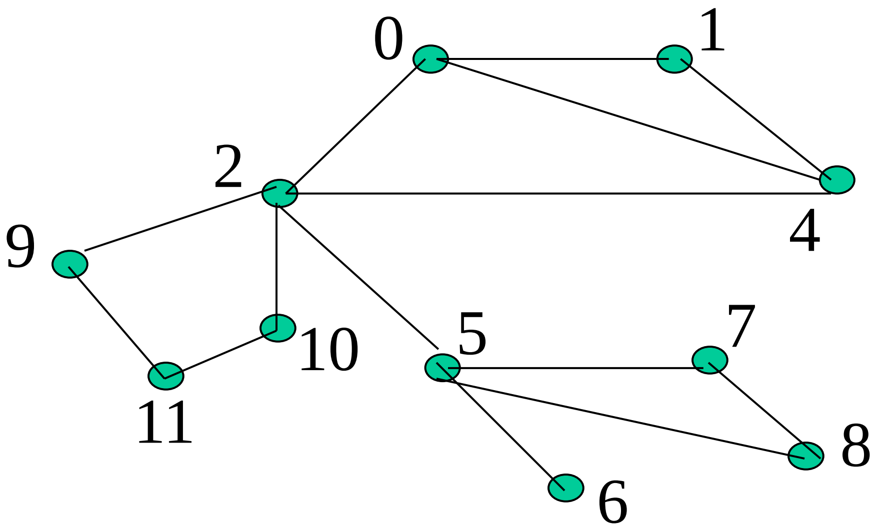
Breadth-First Search

- BFS follows the following rules:
  1. Select an unvisited node x, visit it, have it be the root in a BFS tree being formed. Its level is called the current level.
  2. From each node z in the current level, in the order in which the level nodes were visited, visit all the unvisited neighbors of z. The newly visited nodes from this level form a new level that becomes the next current level.
  3. Repeat step 2 until no more nodes can be visited.
  4. If there are still unvisited nodes, repeat from Step 1.

Illustration of BFS



BFS Tree



Graph G

Implementation of DFS

- Observations:
  - the first node visited in each level is the first node from which to proceed to visit new nodes.

- This suggests that a queue is the proper data structure to remember the order of the steps.

Illustrate BFS with a Queue

- We will redo the BFS on the previous graph, but this time with queues
- In Class

BFS (Pseudo Code)

```
BFS(input: graph G) {
    Queue Q;    Integer x, z, y;
    while (G has an unvisited node x) {
            visit(x); Enqueue(x,Q);
            while (Q is not empty){
                        z := Dequeue(Q);
                        for all (unvisited neighbor y of z){
                                    visit(y); Enqueue(y,Q);
                        }
            }
    }
}
```

Things for you to Do

- Give a C++ implementation of BFS, using the pseudo code as your guide
- Use BFS as a way to determine of the input graph G is connected, and if not, to output the number of connected components
- Modify BFS so that the level of each node in the BFS tree is computed.
- Give another class for graph, this time using a linked lists representation.