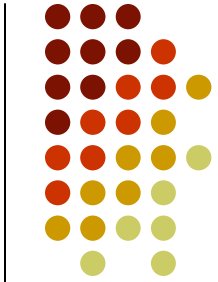
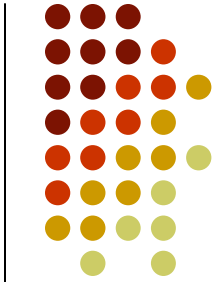
Using `nextAdjacent()`



- To list all vertices connected to a vertex u in graph G we can use the following loop:

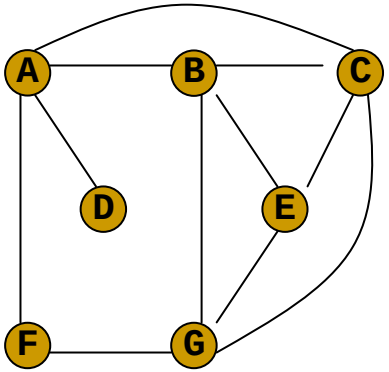
```
for (int v=G.nextAdjacent(u,-1); v>=0; v=G.nextAdjacent(u,v)) {
  System.out.print(v);
}
```

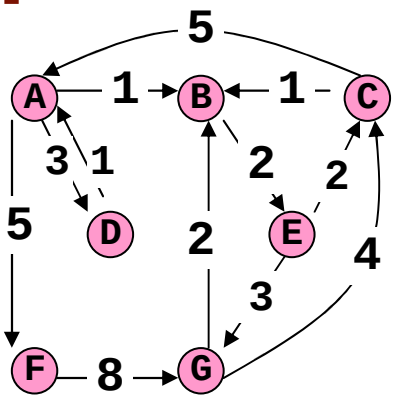
Adjacency Lists



- The edges are recorded in an array of size |V| of linked lists
- In an unweighted graph a list at index *i* records the keys of the vertices adjacent to vertex *i*
- In a weighted graph a list at index *i* contains pairs which record vertex keys (of vertices adjacent to *i*) and their associated edge weights
- Looking up an edge requires time proportional to the number of edges adjacent to a node (a degree of a vertex)
- Finding all vertices adjacent to a given vertex also takes time proportional to the degree of the vertex (which is minimal possible)
- The list requires O (|E|) space

Adjacency List Examples

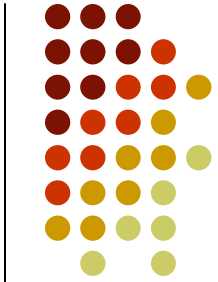




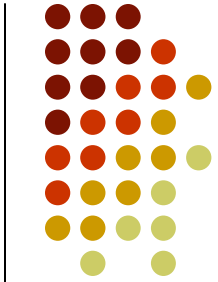
A	B	C	D	F
B	A	C	E	G
C	A	B	E	G
D	A			
E	B	C	G	
F	A	G		
G	B	C	E	F

A	B	1	D	3	F	5
B	E	2				
C	A	5				
D	A	1				
E	C	2	G	3		
F	G	8				
G	B	2	C	4		

Implementation with adjacency lists



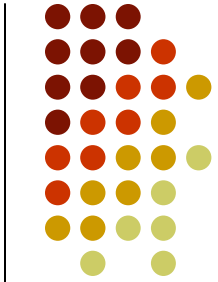
```
class Graph {
  // simple graph (no multiple edges); undirected; unweighted
  private int numVertices;
  private int numEdges;
  private List<Integer> adjList[];

  public Graph(int n) {
    numVertices = n;
    numEdges = 0;
    adjList = new List[n];
    for (int i=0; i<n; i++)
      adjList[i] = new List<Integer>();
  } // end constructor

  public int getNumVertices() { return numVertices; }

  public int getNumEdges() { return numEdges; }

  public boolean isEdge(int v, int w) {
    List<Integer> L = adjList[v];
    for (int i=1; i<=L.size(); i++)
      if (L.get(i).intValue() == w)
        return true;
    return false;
  } // end getWeight
```



```
public void addEdge(int v, int w) {
   if (!isEdge(v,w)) {
     adjList[v].add(adjList[v].size()+1,new Integer(w));
     adjList[w].add(adjList[w].size()+1,new Integer(v));
     numEdges++;
   }
 } // end addEdge

 public void removeEdge(int v, int w) {
   for (int i=1; i<=adjList[v].size(); i++)
     if (adjList[v].get(i).intValue() == w) {
       adjList[v].remove(i);
       break;
     }
   for (int i=1; i<=adjList[w].size(); i++)
     if (adjList[w].get(i).intValue() == v) {
       adjList[w].remove(i);
       numEdges--;
       break;
     }
 } // end remove
```

```
  public int kth_Adjacent(int v,int k)
  // return the k-th adjacent vertex to v
  // or -1 if degree of v is smaller than k
  {
    List<Integer> L = adjList[v];
    if (k>=1 && k<=L.size())
      return L.get(k).intValue();
    else
      return -1;
  }
} // end Graph
```

Using `kth_Adjacent()`



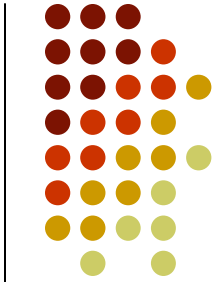
- To list all vertices connected to a vertex u in graph G we can use the following loop:

```
int v;
for (int k=1; (v=G.kth_Adjacent(u,k)) >=0; k++)
  System.out.print(v);
```

`nextAdjacent()` and `kth_Adjacent()`

- Why do we have different methods to access neighbors of a vertex in different implementations?
  - we could implement `nextAdjacent()` in the list implementation and `kth_Adjacent()` in the matrix implementation of graphs, but they would not be efficient
  - to have a common way how to list the neighbors in both implementations, we would have to use “iterators” (see section 9.5 in the textbook – not covered for this course)

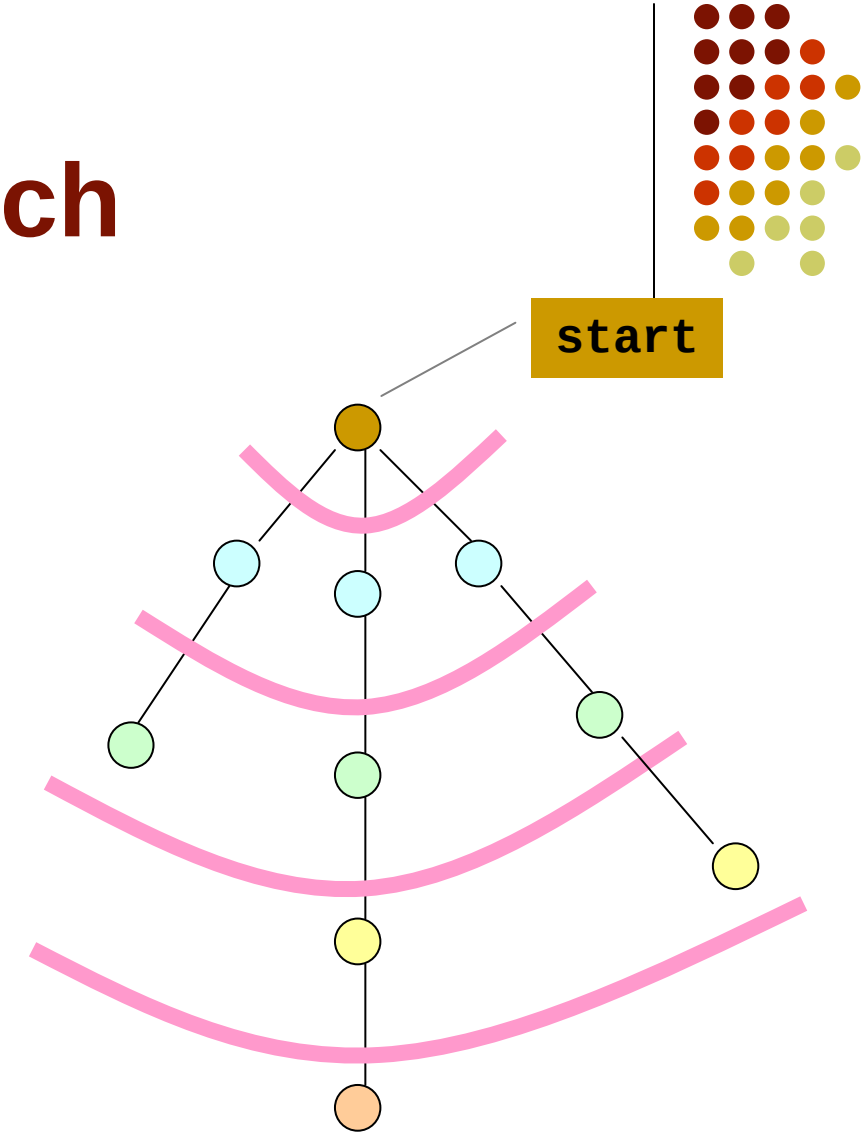
Graph Traversals



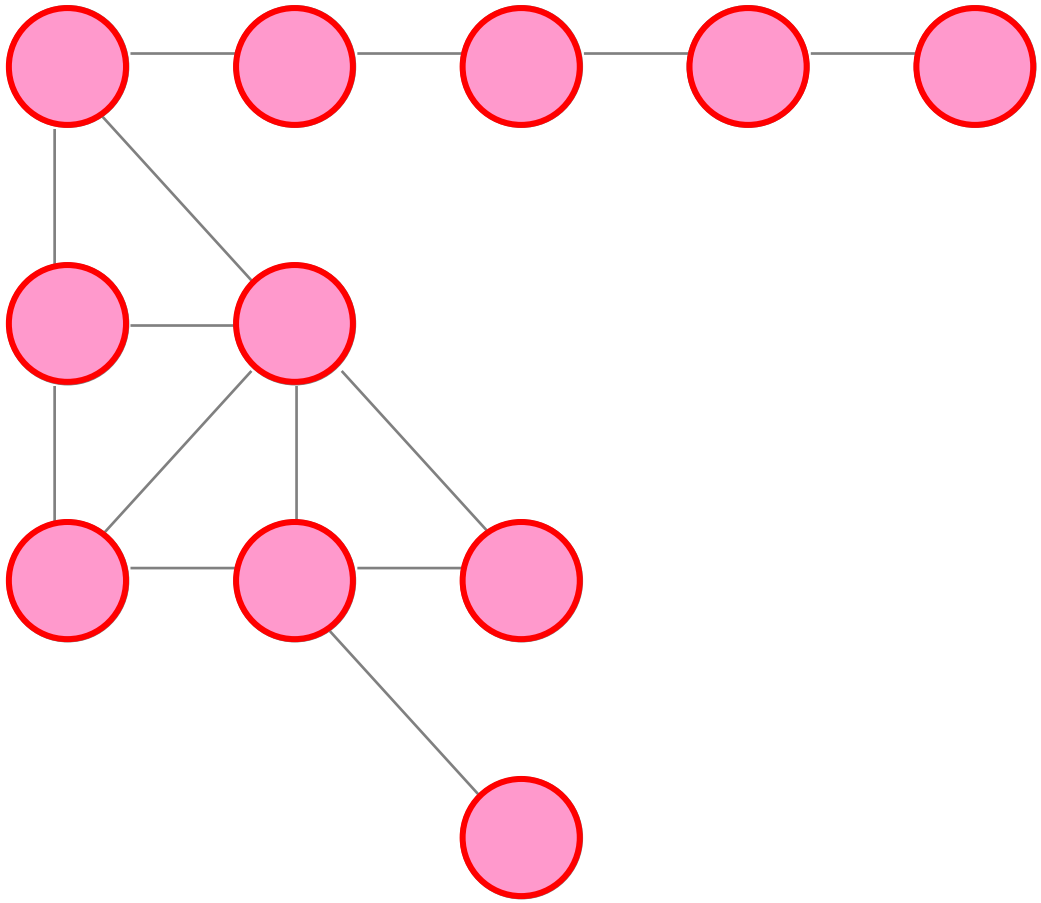
- A graph traversal algorithm visits all of the vertices that it can reach
  - If the graph is not connected all of the vertices will not be visited
  - Therefore a graph traversal algorithm can be used to see if a graph is connected (or count the number of connected parts (= components))
- Vertices should be marked as “visited”
  - Otherwise, a traversal will go into an infinite loop if the graph contains a cycle

Breadth First Search

- After visiting a vertex, *v*, visit every vertex adjacent to *v* before moving on
- Use a **queue** to store nodes whose neighbors we still need to visit
- BFS:
  - visit and insert start
  - while (q not empty)
    - remove node from q and make it *current*
    - visit and insert the unvisited nodes adjacent to *current*

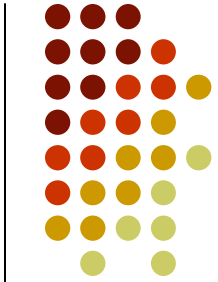


Breadth First Search Example



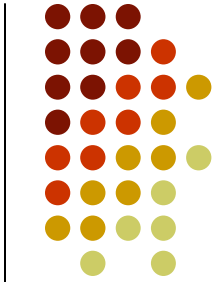
queue	visited
A	A
B	B
F	F
G	G
C	C
H	H
I	I
J	J
D	D
K	K
E	E

BFS - Remarks



- BFS is visiting nodes which are close to the start vertex first, then more distant nodes, etc.
- what distance is it?
  - a **graph distance** of two vertices u and v is the number of edges on (*length of*) the shortest path connecting u and v
  - in weighted graph, the sum of weights of edges of a path is called *weight* or *cost* of the path.

BFS Algorithm

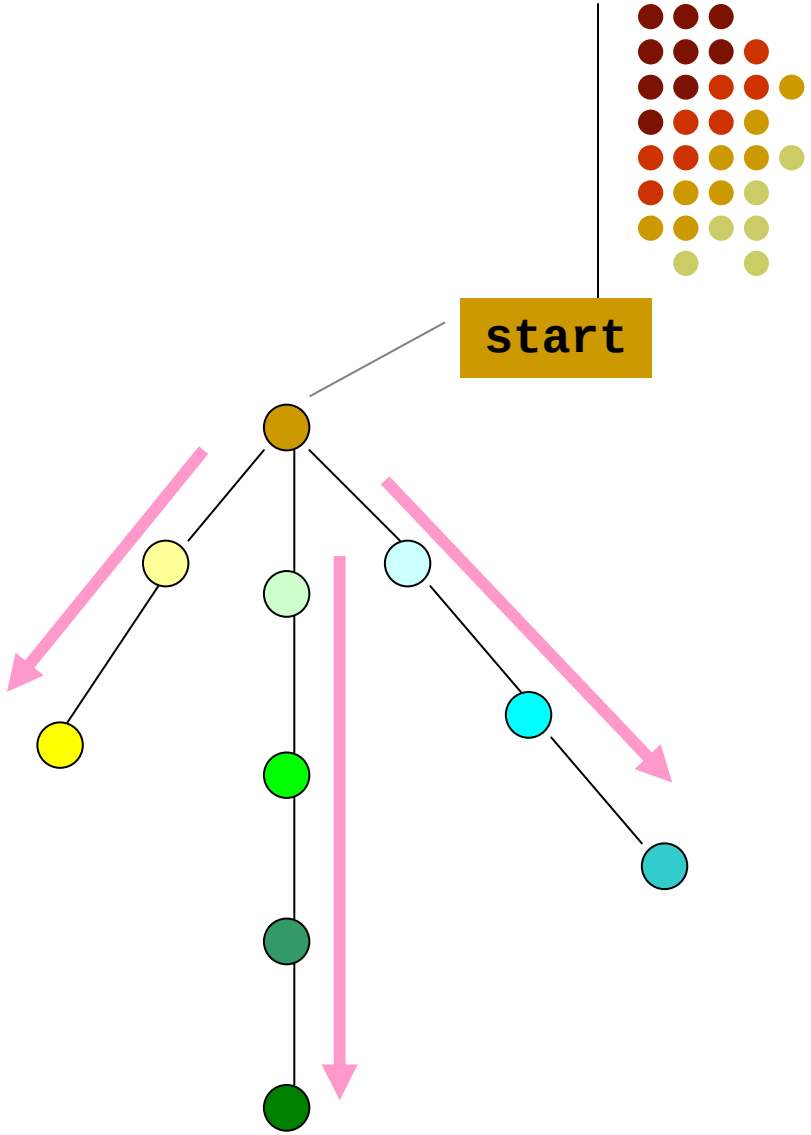


```
public static void BreadthFirstSearch(GraphList G, int start)
{
  int n = G.getNumVertices();
  // create an array where we mark visited vertices
  boolean[] visited = new boolean[n]; // initialized to false
  // create a queue
  Queue<Integer> q = new Queue<Integer>();

  q.enqueue(new Integer(start));
  visit(start);
  visited[start] = true;

  while (!q.isEmpty()) {
    Integer current = q.dequeue();
    int v;
    for (int k=1; (v=G.kth_Adjacent(current.intValue(),k)) >=0; k++)
      if (!visited[v]) {
        q.enqueue(new Integer(v));
        visit(v);
        visited[v] = true;
      }
  }
}
```

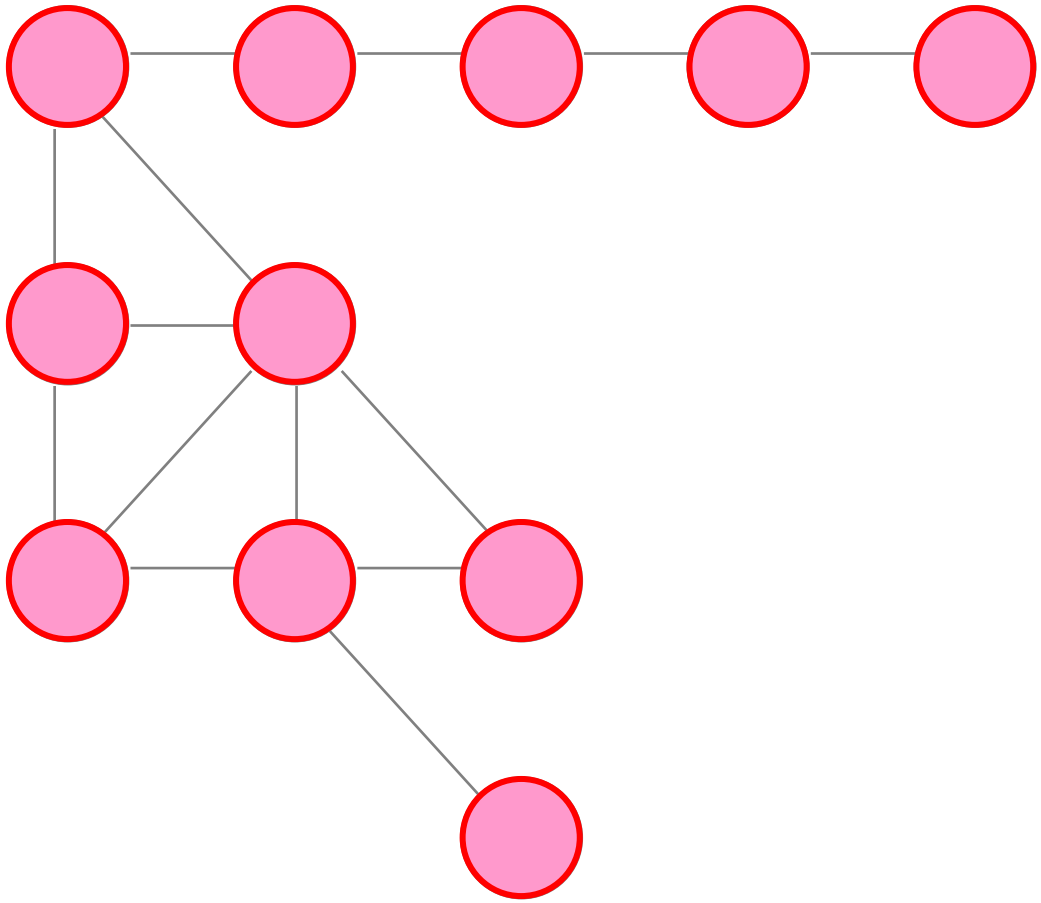
Depth First Search

- Visit a vertex, *v*, move from *v* as deeply as possible
- Use a stack s to store nodes

- DFS:
  - visit and push start
  - while (s not empty)
    - peek at node, *nd*, at top of s
    - if *nd* has an unvisited neighbour visit it and push it onto s
    - else pop *nd* from s



Depth First Search Example



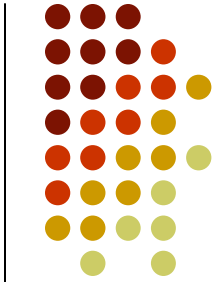
stack	visited
J K	A
E I	B
D H	C
C G	D
B F	E
A	F
	G
	H
	I
	J
	K

Invariant: the sequence of vertices on the stack forms a simple path in the graph starting in the vertex `start`.

DFS Algorithm

```
public static void DepthFirstSearch(GraphMatrix G, int start)
{
  int n = G.getNumVertices();
  // create an array where we mark visited vertices
  boolean[] visited = new boolean[n]; // initialized to false
  // create a queue
  Stack<Integer> s = new Stack<Integer>();

  s.push(new Integer(start));
  visit(start);
  visited[start] = true;

  while (!s.isEmpty()) {
    int u = s.peek().intValue(); // the vertex at the top of stack
    boolean hasUnvisitedNeighbor = false;
    for (int v=G.nextAdjacent(u,-1); v>=0; v=G.nextAdjacent(u,v))
      if (!visited[v]) {
        s.push(new Integer(v));
        visit(v);
        visited[v] = true;
        hasUnvisitedNeighbor = true;
        break;
      }
    if (!hasUnvisitedNeighbor)
      s.pop();
  }
}
```

DFS - Remarks



- what Depth First Search does is backtracking (which is also indicated since we are using stack)
- one could easily write a recursive version of DFS