Graph Theory

UF Programming Team

Announcements

Career Development Workshop

This Monday (9/28) from 6-10pm at Ben Hill Griffin Stadium, UF is holding CDW, which allows students to talk to tech companies, drop off their resume, and set up interviews.

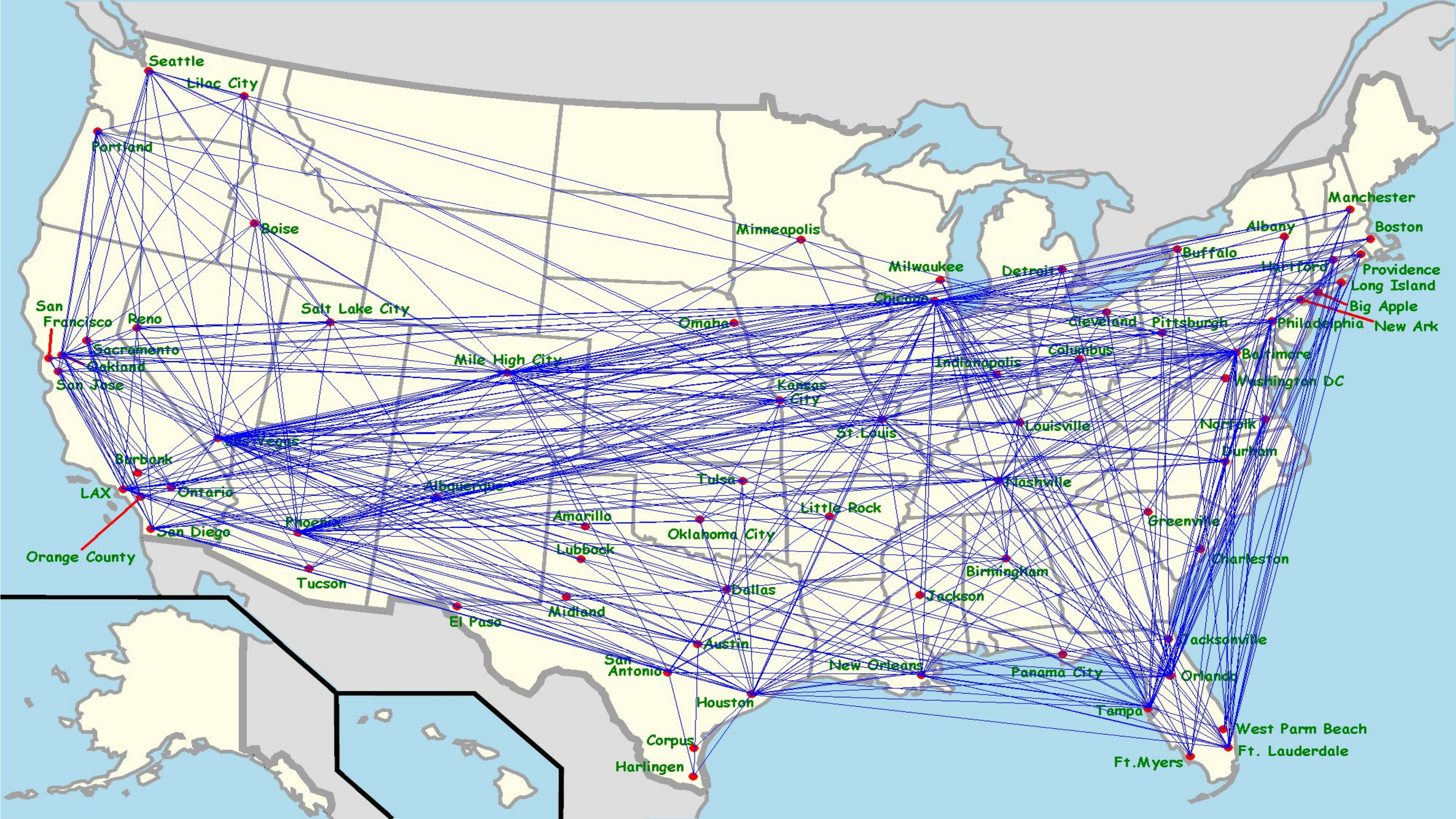
There will be no practice this Monday due to CDW, so get dressed up, swing by, and get yourself an internship or job!

*** We will be holding an **Interview Bootcamp** this Sunday from 2-6pm in E121 to help prepare everyone for CDW

What is a graph?

A **graph** consists of **vertices** (or nodes) and **edges** (or paths)

- Edges are connections between vertices
  - *e.g.*, roads between cities
- Edges can have one-way direction or can be bidirectional
- Edges can have weights
  - *e.g.,* time to travel between two cities


Seattle
Lilac City
Portland
Boise
Minneapolis
Manchester
Albany
Boston
Buffalo
Milwaukee
Detroit
Hartford
Providence
Long Island
Chicago
Big Apple
San Francisco
Reno
Salt Lake City
Omaha
Cleveland
Pittsburgh
Philadelphia
New Ark
Sacramento
Oakland
San Jose
Mile High City
Indianapolis
Columbus
Baltimore
Washington DC
Kansas City
St.Louis
Louisville
Norfolk
Las Vegas
Durham
Burbank
LAX
Ontario
Albuquerque
Tulsa
Nashville
Little Rock
Amarillo
Oklahoma City
Greenville
Phoenix
San Diego
Lubbock
Charleston
Orange County
Birmingham
Tucson
Dallas
Jackson
Midland
El Paso
Austin
Jacksonville
San Antonio
New Orleans
Panama City
Orlando
Houston
Tampa
Corpus
West Parm Beach
Ft. Lauderdale
Harlingen
Ft.Myers

Graph Terminology

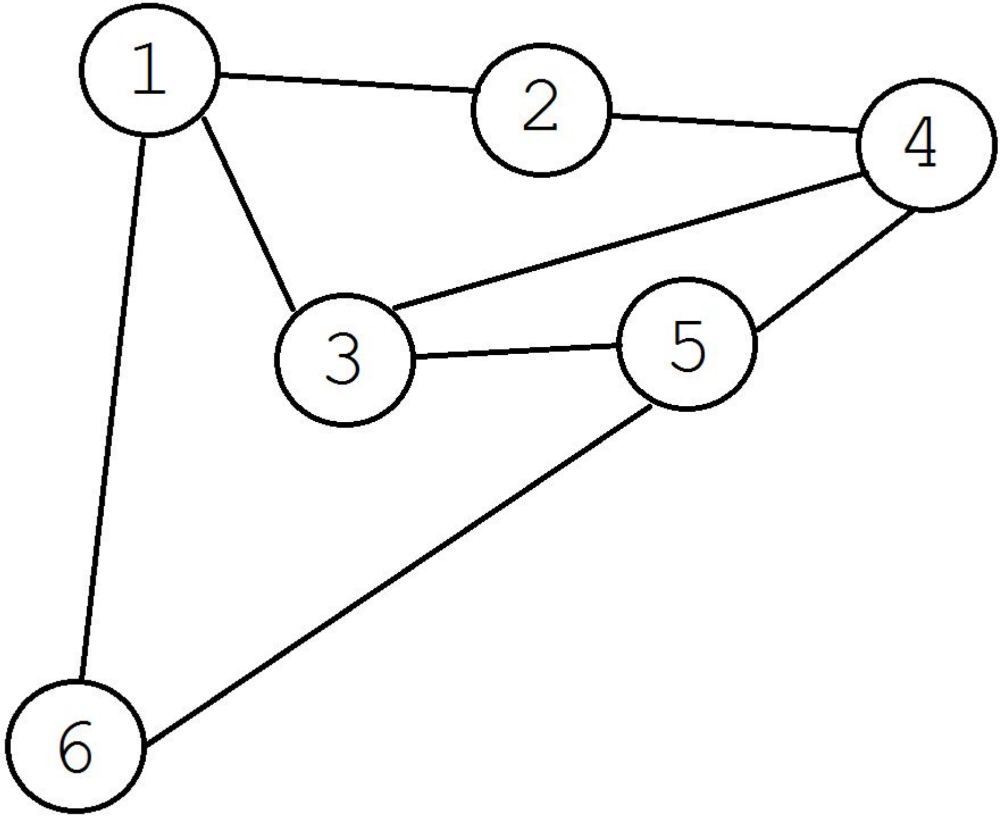
An **acyclic graph** contains no cycles (only *one* path from a vertex to any other vertex)

A **directional graph** has edges that are all directional (*i.e.*, if there's an edge from A to B, there might not be one from B to A)

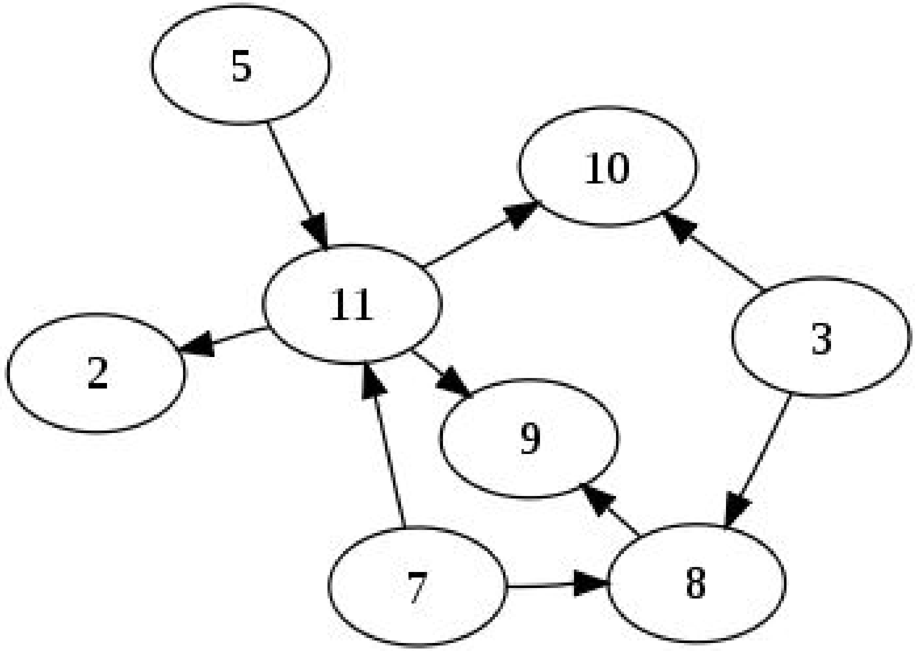
A **weighted graph** contains edges that hold a numerical value (*e.g.*, cost, time, etc.)

Examples of graphs

Unweighted, undirected cyclic graph



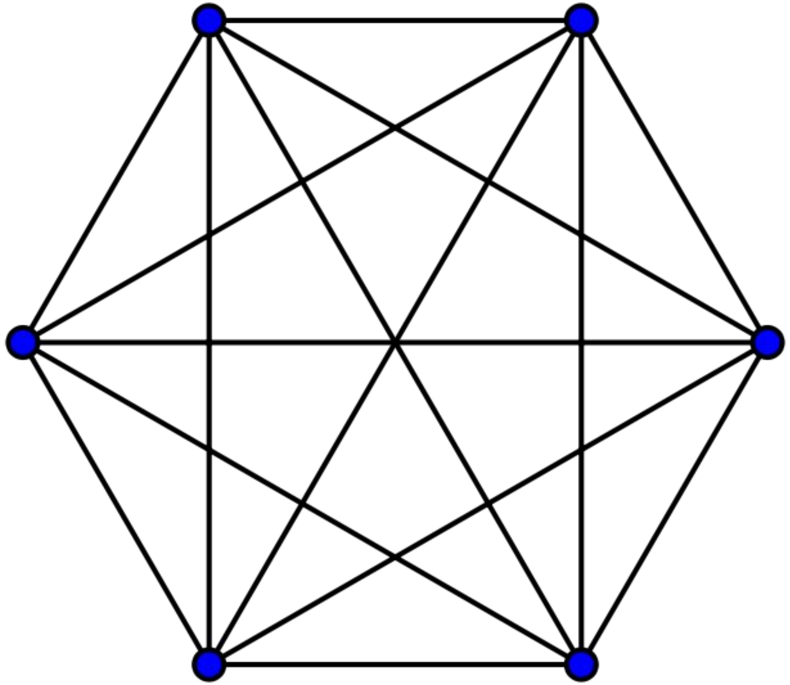
Examples of graphs



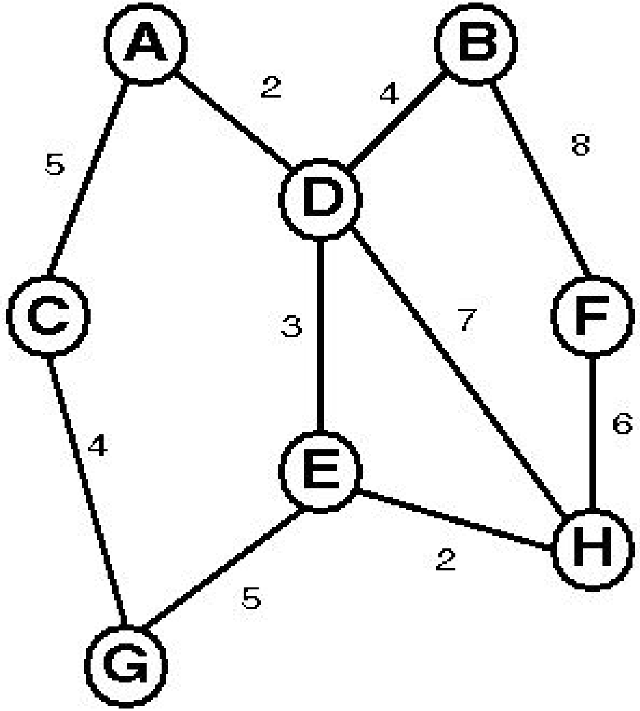
Directed acyclic graph (DAG)

Examples of graphs

Complete graph (all vertices have an edge to all other vertices in the graph)



Examples of graphs



Weighted cyclic graph

Representing a graph

How do we represent a graph?

Adjacency matrix

→ uses *2D arrays* to represent edges between two vertices

Adjacency list

→ uses a *list of lists* to represent edges between two vertices

Adjacency Matrix

There are many ways to represent a graph using an adjacency matrix, but we will look at two variations:

```
boolean[][] conn = new boolean[N][N];
```

→ Matrix `conn` tells us if two vertices are connected

```
int[][] cost = new int[N][N];
```

→ Matrix `cost` tells us the cost (edge weight) between two vertices

Matrix `conn`

```
boolean[][] conn = new boolean[N][N];
```

→ initialize the 2D array

```
conn[i][j] = true;
```

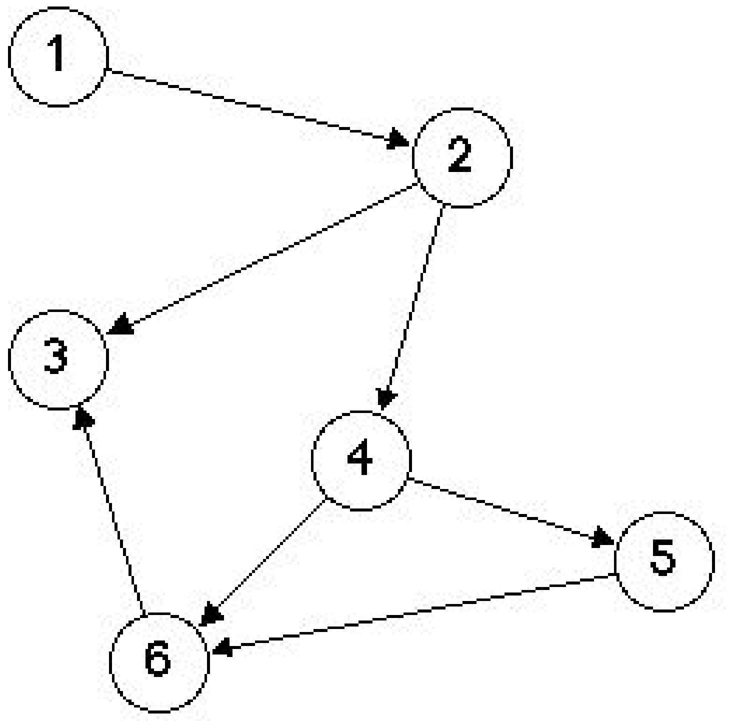
→ a value of true means there is an edge from *i* to *j*

```
conn[j][k] = false;
```

→ a value of false means there is **no** edge from *j* to *k*

Adjacency matrix example (using `conn`)

	1	**2**	**3**	**4**	**5**	**6**
1	0	1	0	0	0	0
2	0	0	1	1	0	0
3	0	0	0	0	0	0
4	0	0	0	0	1	1
5	0	0	0	0	0	1
6	0	0	1	0	0	0



Matrix `cost`

```
int[][] cost = new int[N][N];
```

→ initialize the 2D array

```
cost[i][j] = 7;
```

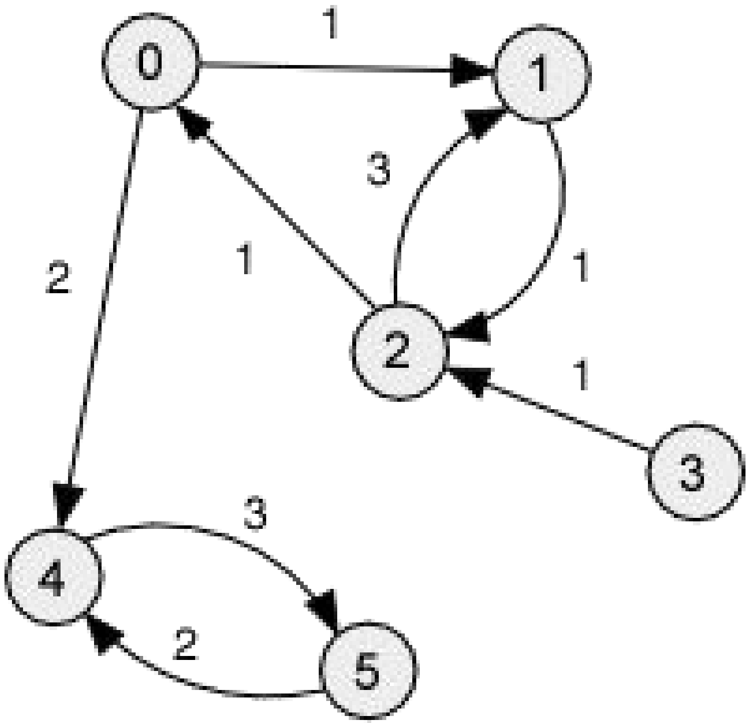
→ this means it costs **7 units** to travel from *i* to *j*

```
cost[j][k] = 0;
```

→ a cost of **0 units** *usually* means there is no edge from *j* to *k*

Adjacency matrix example (using `cost`)

	0	1	2	3	4	5
0	0	1	0	0	2	0
1	0	0	1	0	0	0
2	1	3	0	0	0	0
3	0	0	1	0	0	0
4	0	0	0	0	0	3
5	0	0	0	0	2	0



Adjacency Matrix: Pros and Cons

Pros

- Easy to check if there is an edge between *i* and *j* - just a single call to `matrix[i][j]`

Cons

- To find all neighbors of vertex *i*, you would need to check the value of `matrix[i][j]` for all *N* vertices, *j*
- Need to construct a 2D array of size *N*N*

Adjacency List

Rather than making space for all *N*N* possible edge connections, an *adjacency list* only keeps track of the vertices that a vertex, *i*, has an edge to.

We are able to do this by creating an *ArrayList* that contains *ArrayLists* holding the values of the vertices that a vertex is connected to.

Adjacency List

```
ArrayList<ArrayList<Integer>> graph = …
for (int i = 0; i <= N; i++) {
   graph.add( new ArrayList<Integer>() );
}
```

→ initialize the *ArrayList* of *ArrayLists*

→ the *ArrayList* returned at index *i* gives us all vertices that have an edge from *i*

Adjacency List

```
graph.get(i).add(j);
```

→ get the *ArrayList* for vertex *i* and add an edge to vertex *j*

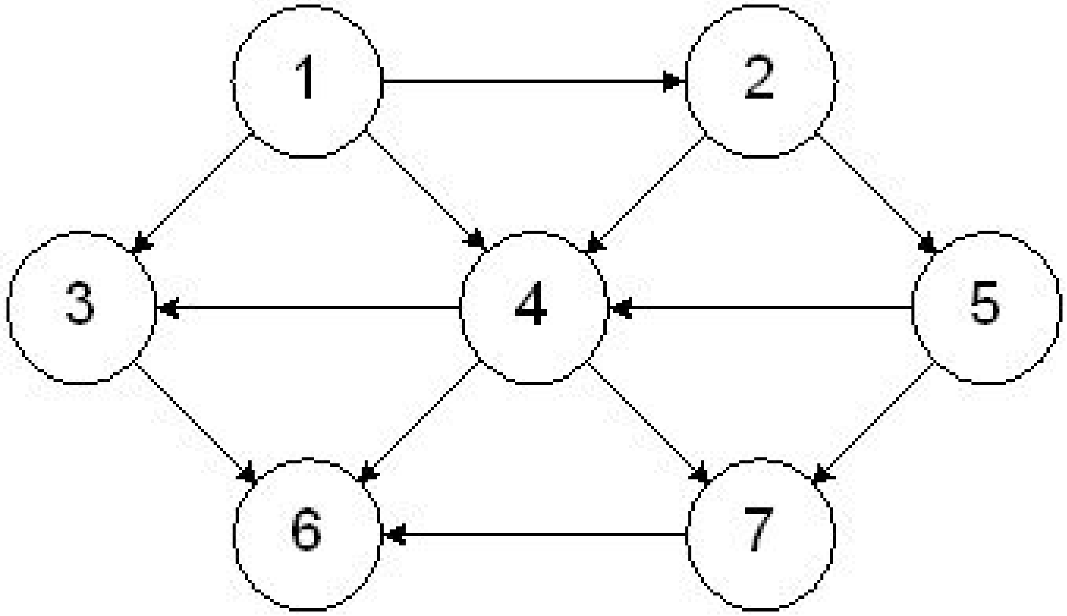
→ this means that there is an edge from *i* to *j*

```
ArrayList<Integer> neighbors = graph.get(k);
```

→ the *ArrayList* neighbors contains all vertices that have an edge coming from *k*

Adjacency list example

i	***graph.get(i)***
1	{2, 3, 4}
2	{4, 5}
3	{6}
4	{3, 6, 7}
5	{4, 7}
6	{}
7	{6}



Adjacency List: Pros and Cons

Pros

→ Saves a lot of memory by only keeping track of edges that a vertex has

→ Efficient to iterate over the edges of a vertex (doesn’t need to go through all *N* vertices and check if there is a connection)

Cons

→ Difficult to quickly determine if there is an edge from *i* to *j*

Traversing a Graph

Now that we have talked about how to construct a graph, we need a method from *traversing* a graph, or visiting the vertices of a graph.

There are two main ways to traverse a graph:

→ *Breadth-first search (BFS)*

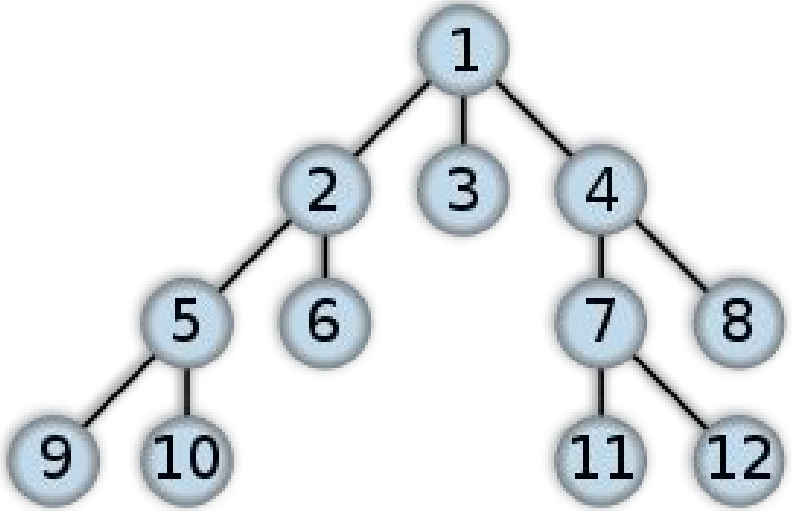
→ *Depth-first search (DFS)*

Breadth-first search

Form of graph traversal that starts at some vertex *i* and visits all of *i*'s neighbors. It then visits all of the neighbors of *i*'s neighbors. This process keeps going until there are no more vertices.

Imagine a "family tree"...

BFS will visit all vertices of the same level before moving on to the next level



Breadth-first search

We need to use some data structure that will allow us to visit the vertices “*level-by-level*”, that is, visit every vertex at level *i* before we visit any vertex at level *i+1*.

In order to do this, we will be using a *Queue* since it follows the “*first in, first out*” order. This means if we put all the vertices at level *i* into the queue before the vertices at level *i+1*, we are guaranteed to visit the lower level vertices first.

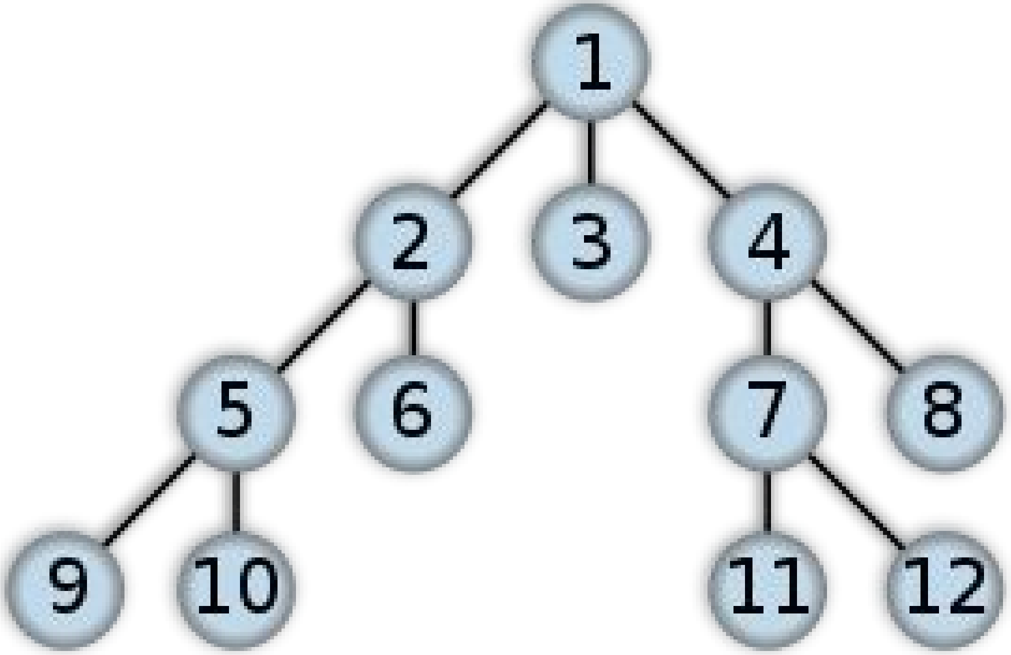
Breadth-first search

We will use the following graph to demonstrate BFS.

A red vertex means it has been visited.

A blue vertex means it is in the *Queue*.

A green vertex means it is the current vertex.

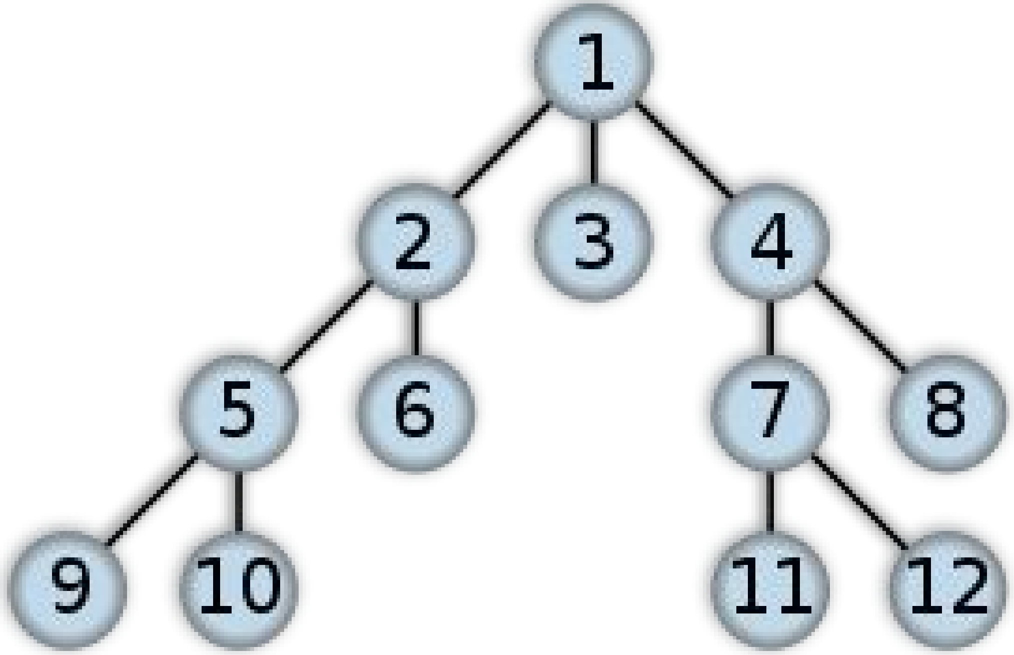


Breadth-first search

We will initially push *1* into the *Queue* to start the BFS.

Current: *null*

Queue: *{1}*

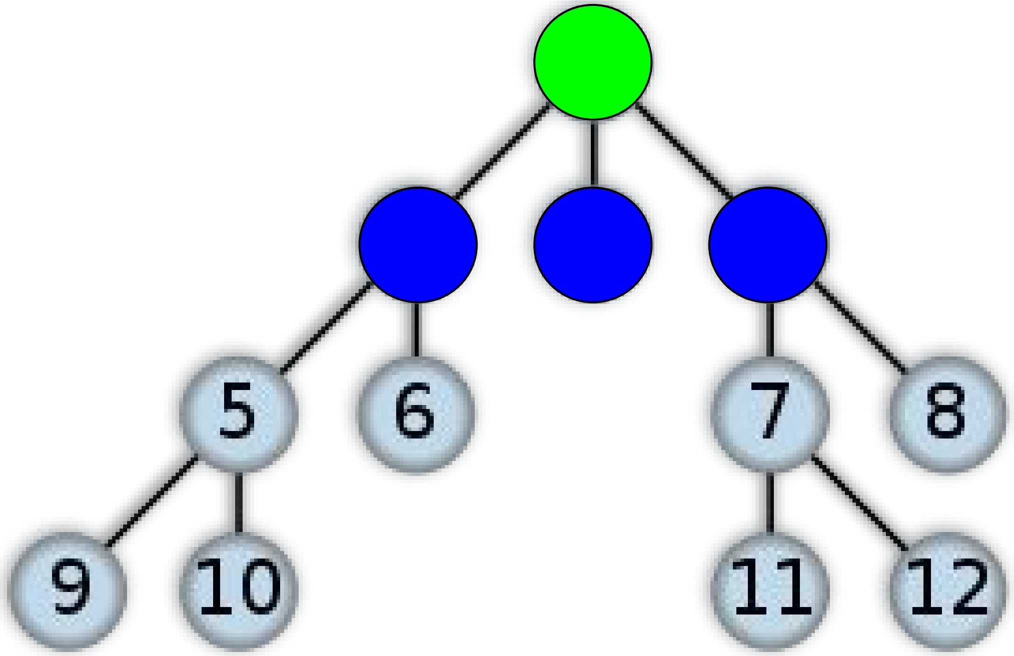


Breadth-first search

We are currently on vertex *1* and want to add all of its neighbors to the *Queue*

Current: *1*

Queue: *{2, 3, 4}*

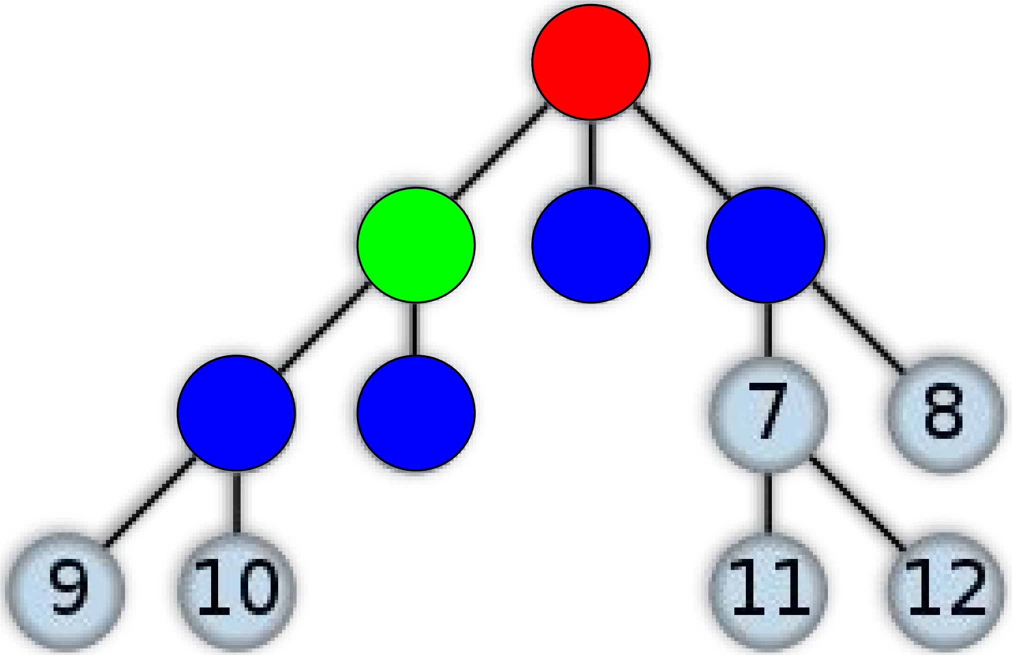


Breadth-first search

We are currently on vertex *2* and want to add all of its neighbors to the *Queue* (**note**: we do not add *1* to the *Queue* since we have already visited it)

Current: *2*

Queue: *{3, 4, 5, 6}*

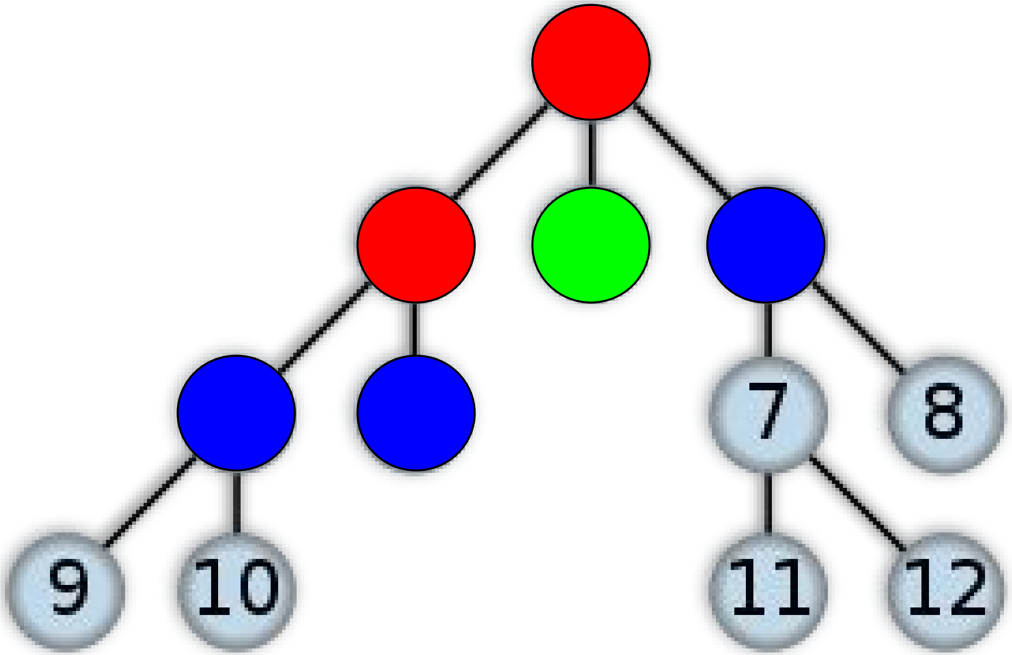


Breadth-first search

We are currently on vertex *3* and want to add all of its neighbors to the *Queue*, but it does not have any neighbors, so nothing is added to the *Queue*

Current: *3*

Queue: *{4, 5, 6}*

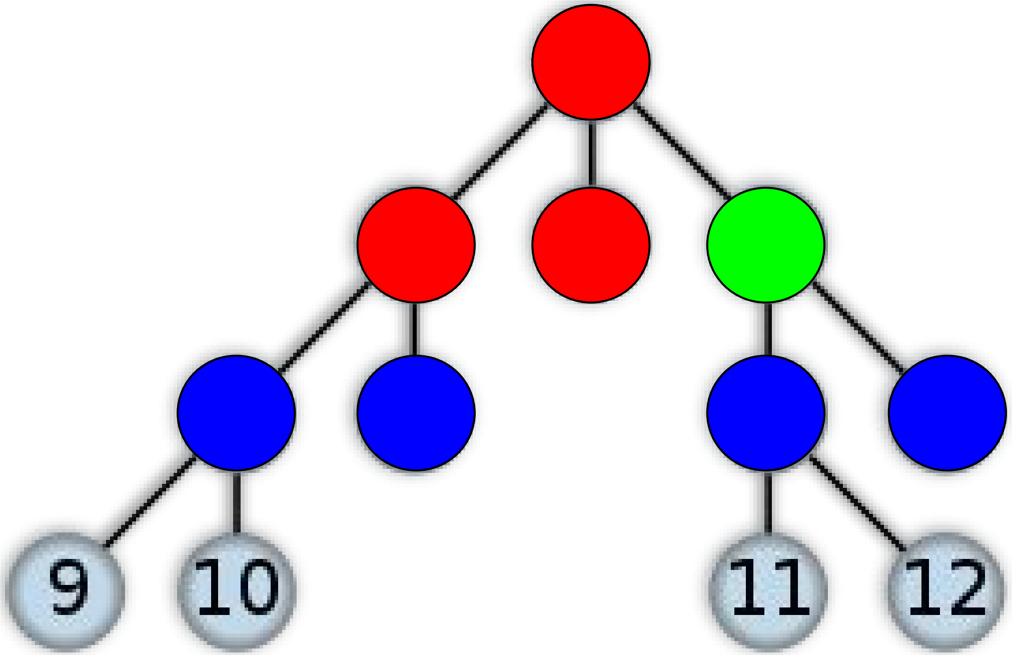


Breadth-first search

We are currently on vertex *4* and want to add all of its neighbors to the *Queue*

Current: *4*

Queue: *{5, 6, 7, 8}*

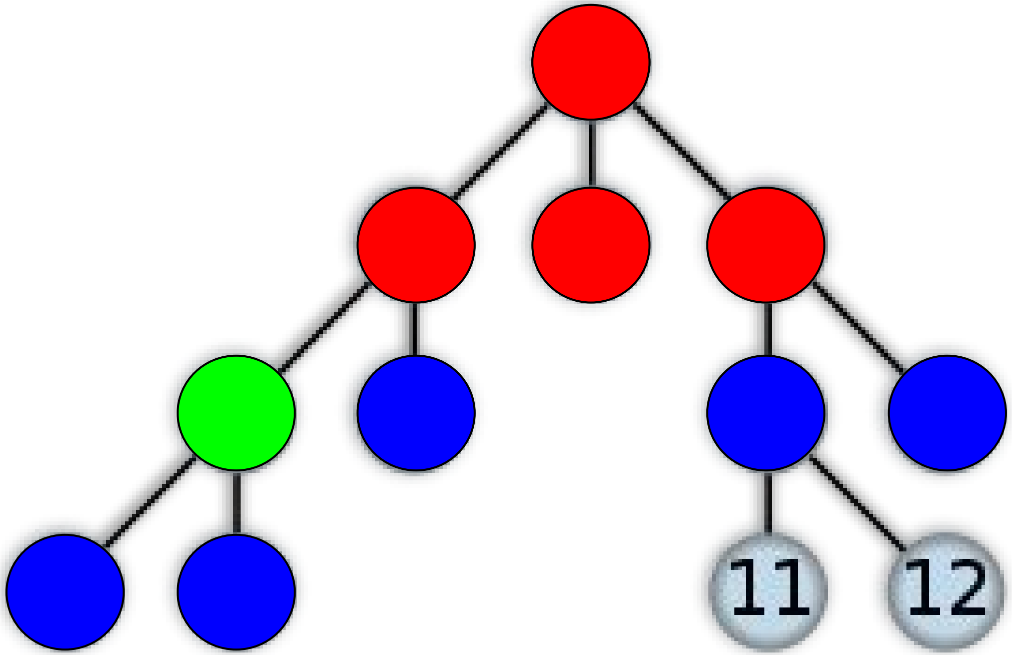


Breadth-first search

We are currently on vertex *5* and want to add all of its neighbors to the *Queue*

Current: *5*

Queue: *{6, 7, 8, 9, 10}*

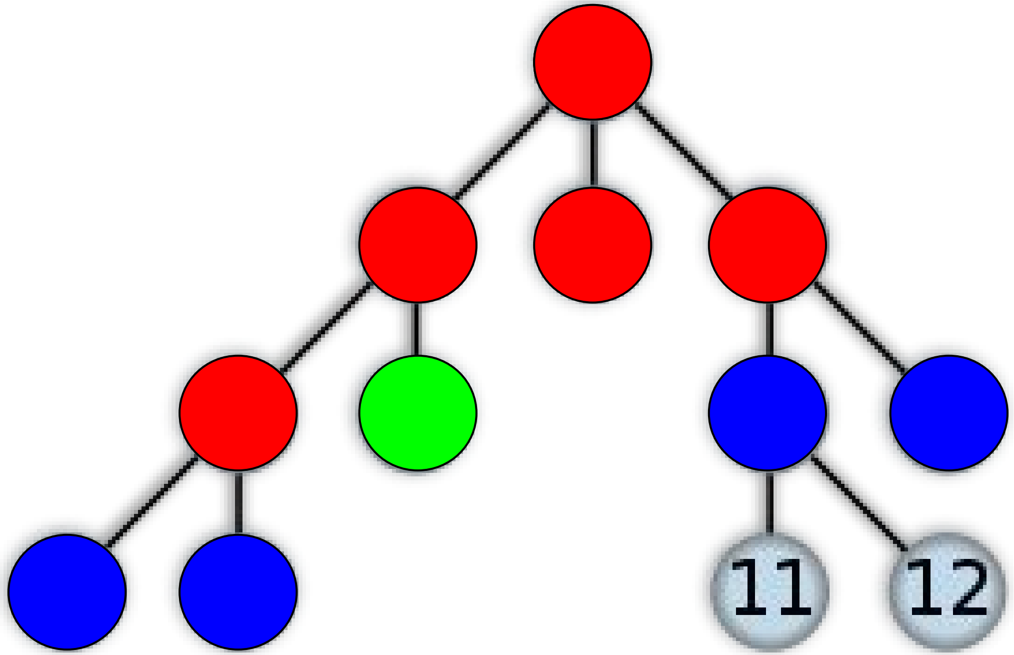


Breadth-first search

Vertex *6* has no neighbors that are unvisited, so we continue

Current: *6*

Queue: *{7, 8, 9, 10}*

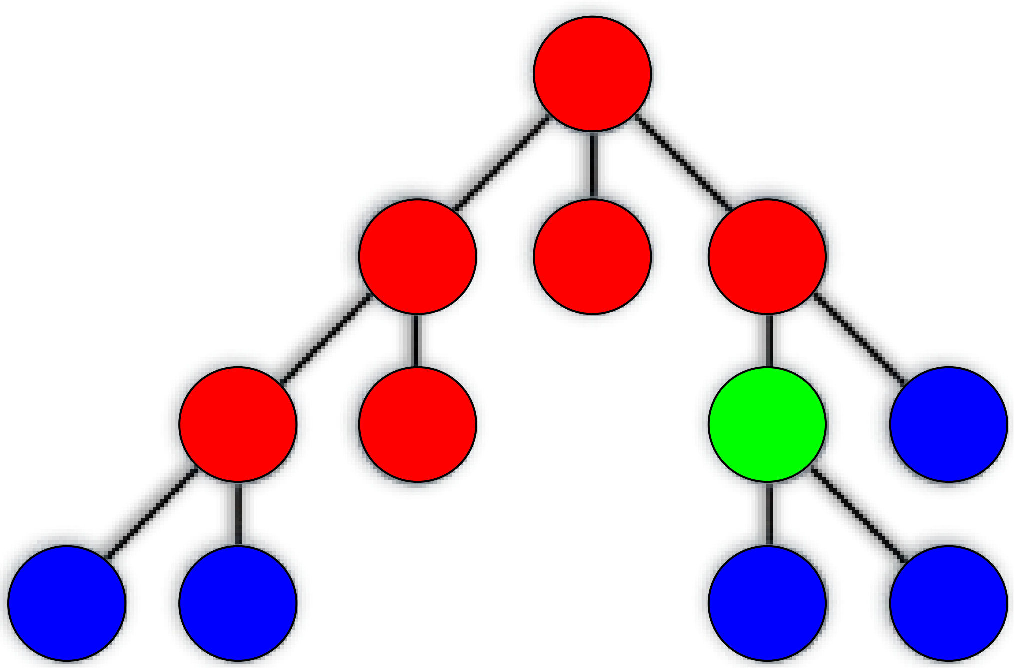


Breadth-first search

We are currently on vertex *7* and want to add all of its neighbors to the *Queue*

Current: *7*

Queue: *{8, 9, 10, 11, 12}*

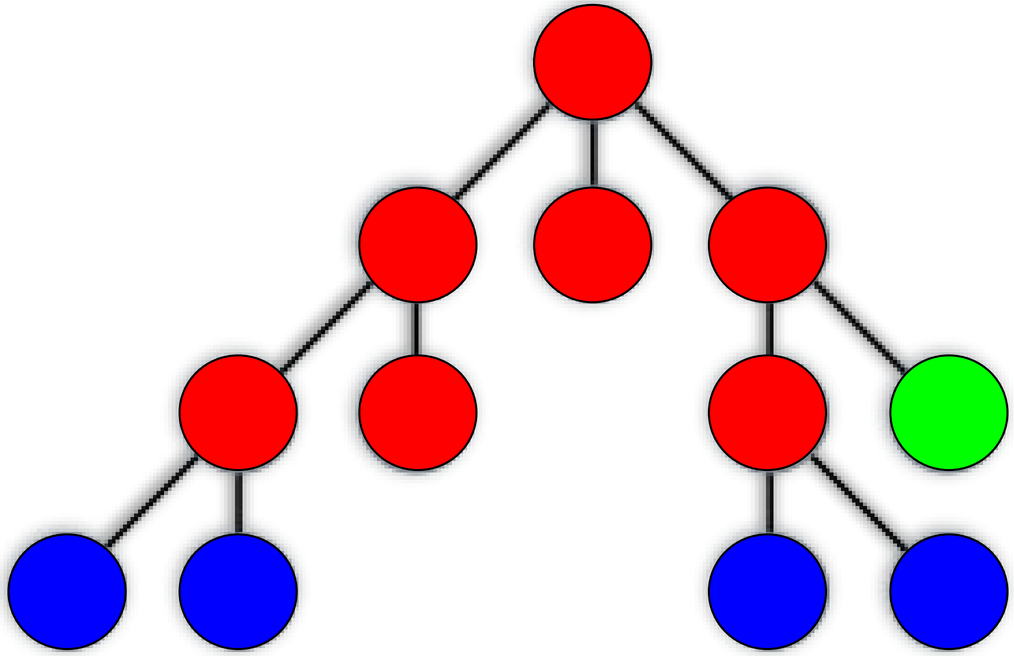


Breadth-first search

Vertex *8* has no neighbors that are unvisited, so we continue

Current: *8*

Queue: *{9, 10, 11, 12}*

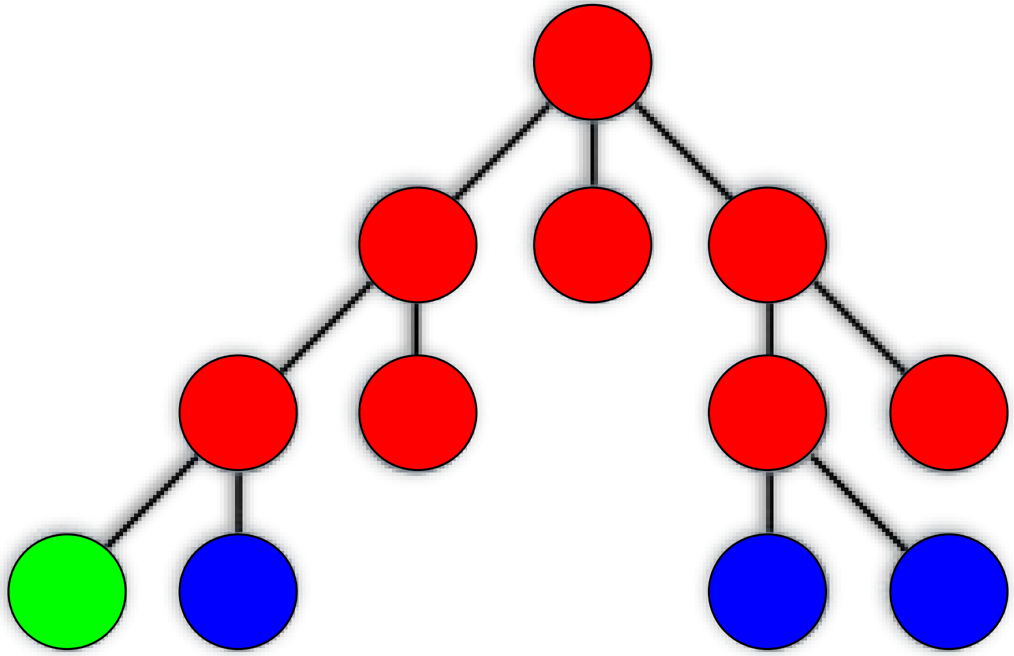


Breadth-first search

Vertex *9* has no neighbors that are unvisited, so we continue

Current: *9*

Queue: *{10, 11, 12}*

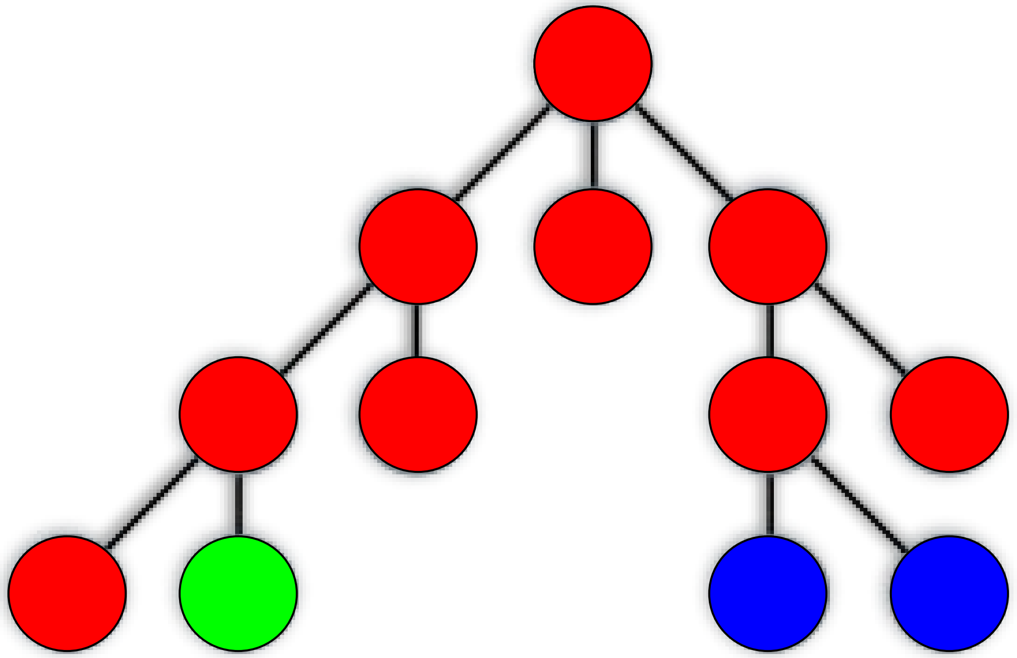


Breadth-first search

Vertex *10* has no neighbors that are unvisited, so we continue

Current: *10*

Queue: *{11, 12}*

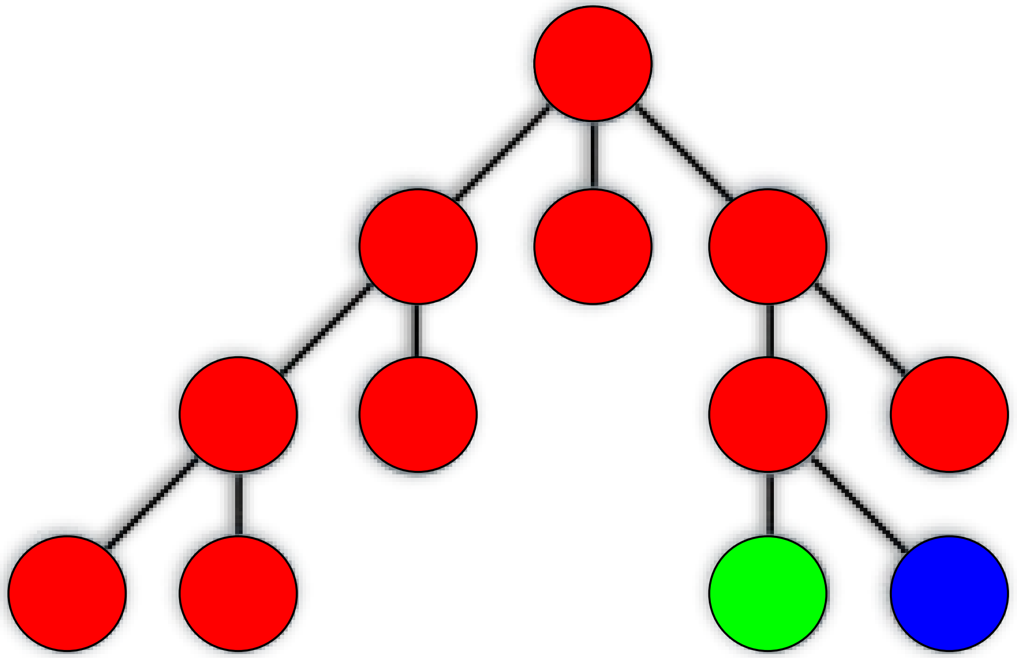


Breadth-first search

Vertex *11* has no neighbors that are unvisited, so we continue

Current: *11*

Queue: *{12}*

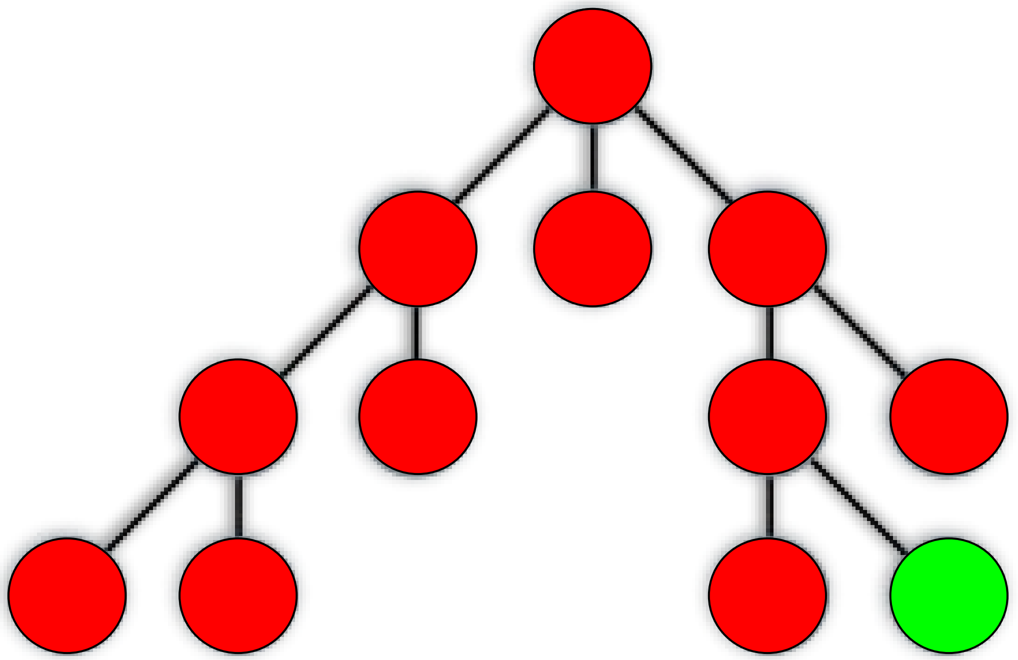


Breadth-first search

Vertex *12* has no neighbors that are unvisited, so we continue

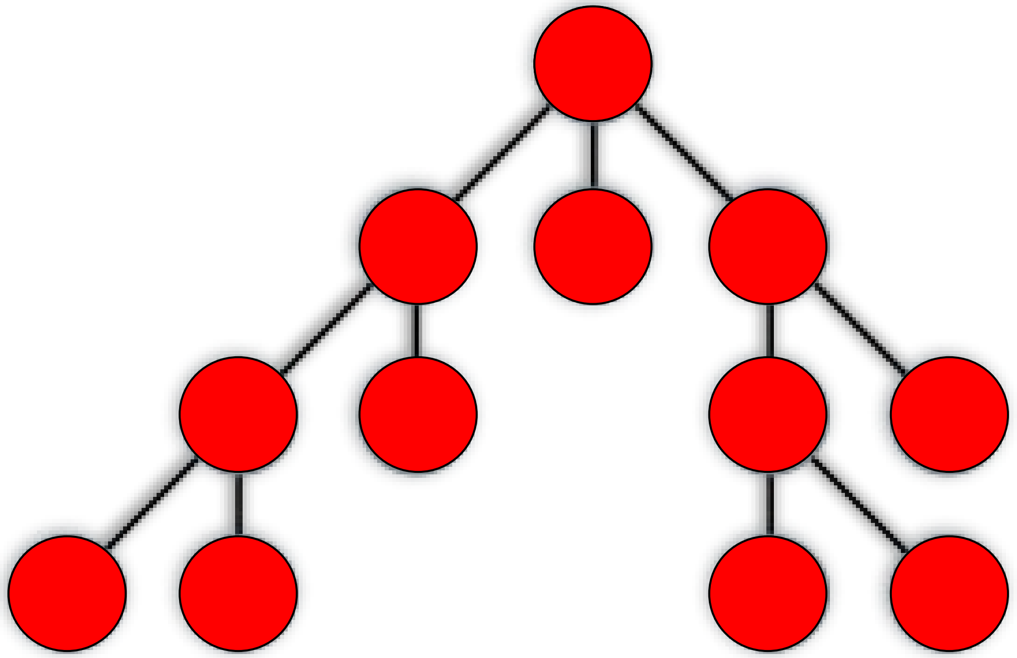
Current: *12*

Queue: *{}*

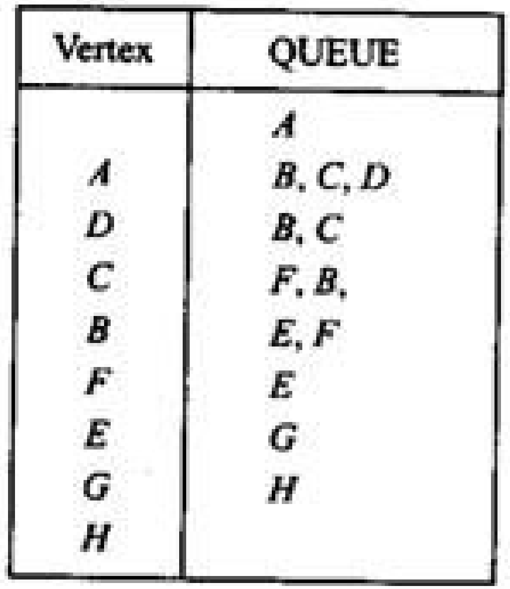


Breadth-first search

Once the *Queue* is empty, we are done with BFS, and have visited every vertex in the graph!

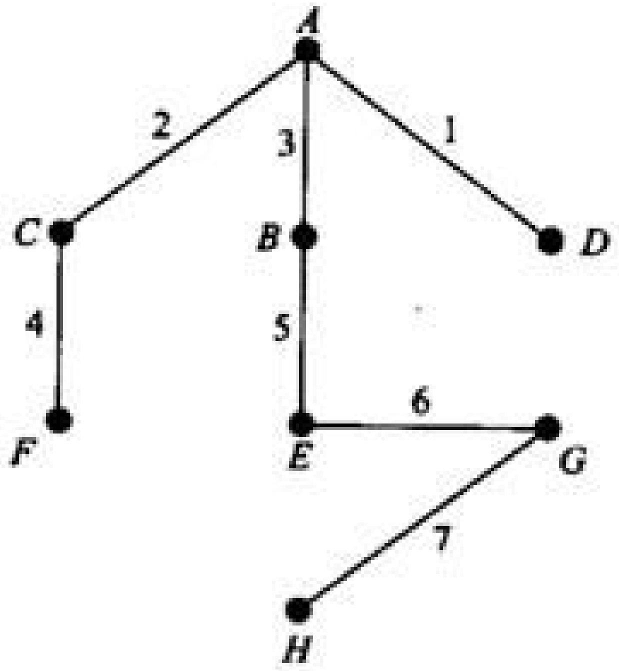


Breadth-first search



Vertex	QUEUE
	A
A	*B*, *C*, *D*
D	*B*, *C*
C	*F*, *B*,
B	*E*, *F*
F	*E*
E	*G*
G	*H*
H	

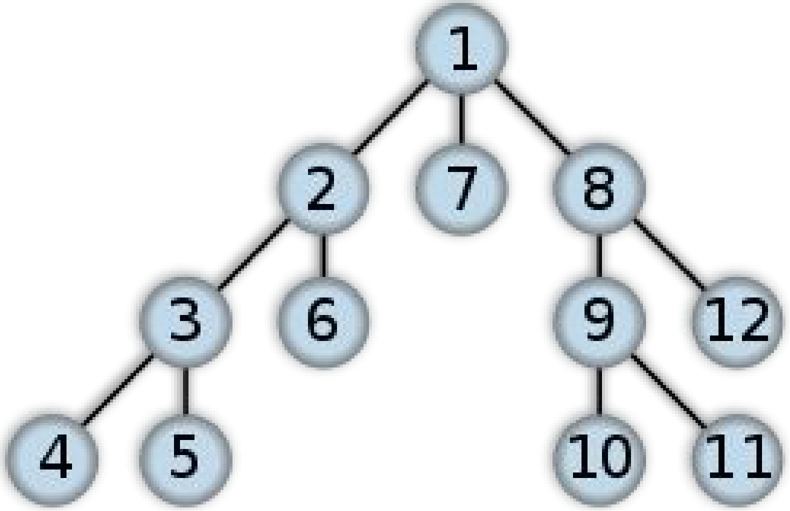
(*a*)



(*b*)

Depth-first search

Another form of graph traversal, but rather than visiting vertices "*level-by-level*", DFS aims to go as deep as possible in the graph before backtracking.



Depth-first search

The algorithm for DFS is very similar to that of BFS, except instead of using a *Queue*, we will be using a *Stack*

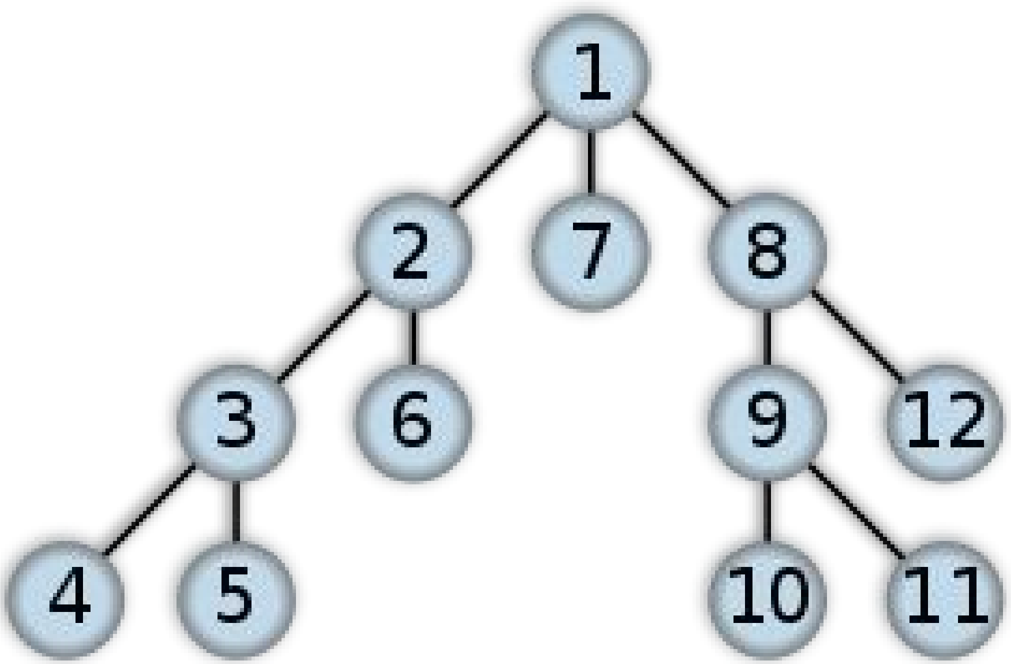
Since a *Stack* follows the "*last in, first out*" order, when we are adding neighbors of a vertex, the last one we push into the *Stack* will be the next one we visit, allowing us to constantly go deeper into the graph rather than traversing an entire level at a time.

Depth-first search

We will initially push *1* into the *Stack* to start the DFS.

Current: *null*

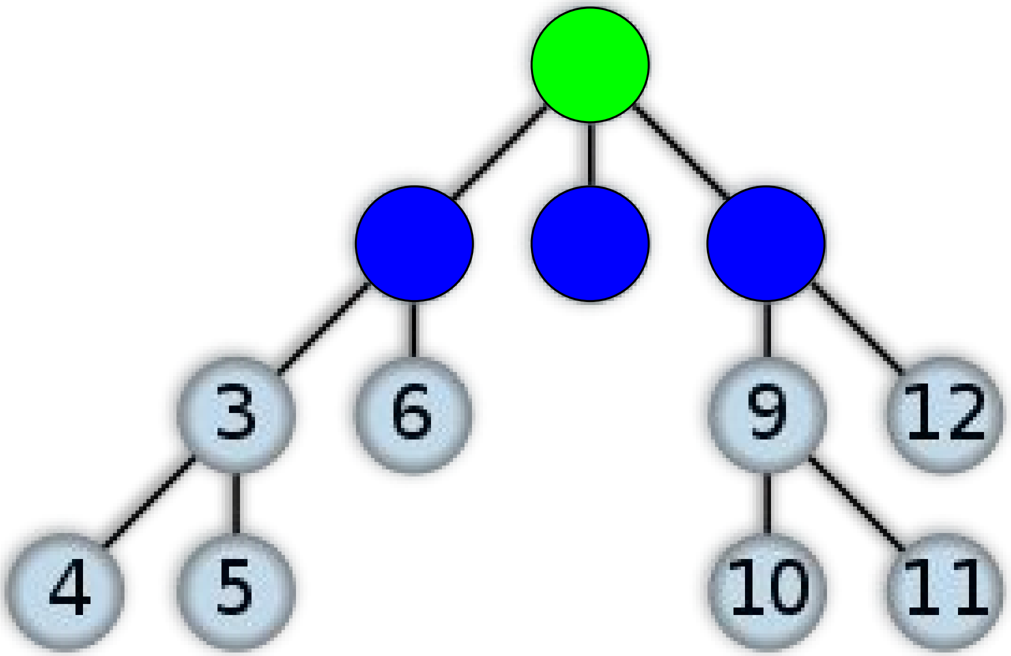
Stack: *{1}*



Depth-first search

Current: *1*

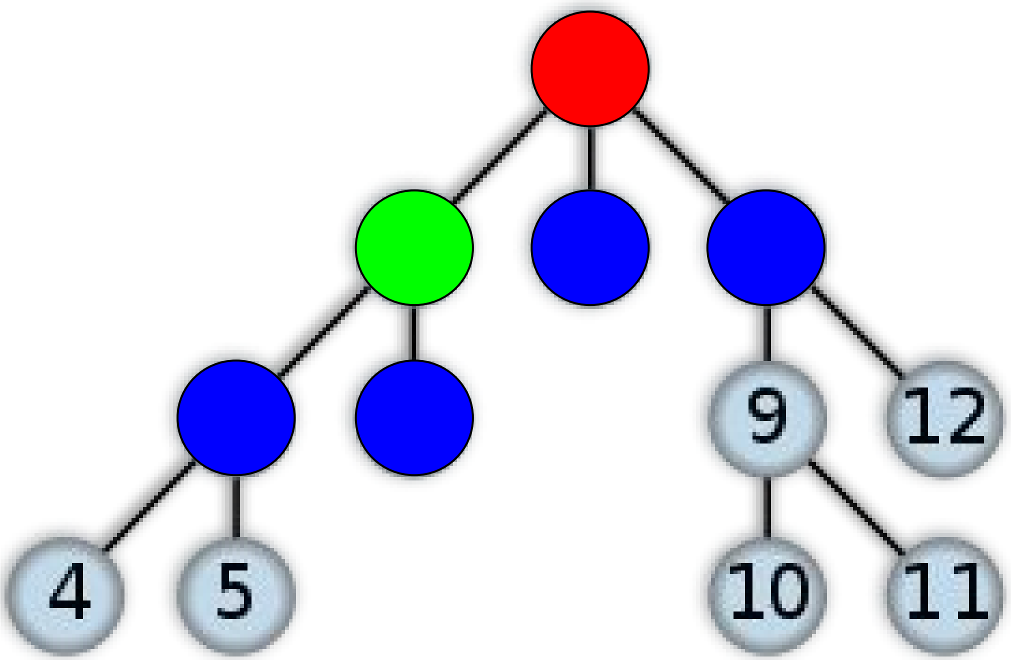
Stack: *{8, 7, 2}*



Depth-first search

Current: *2*

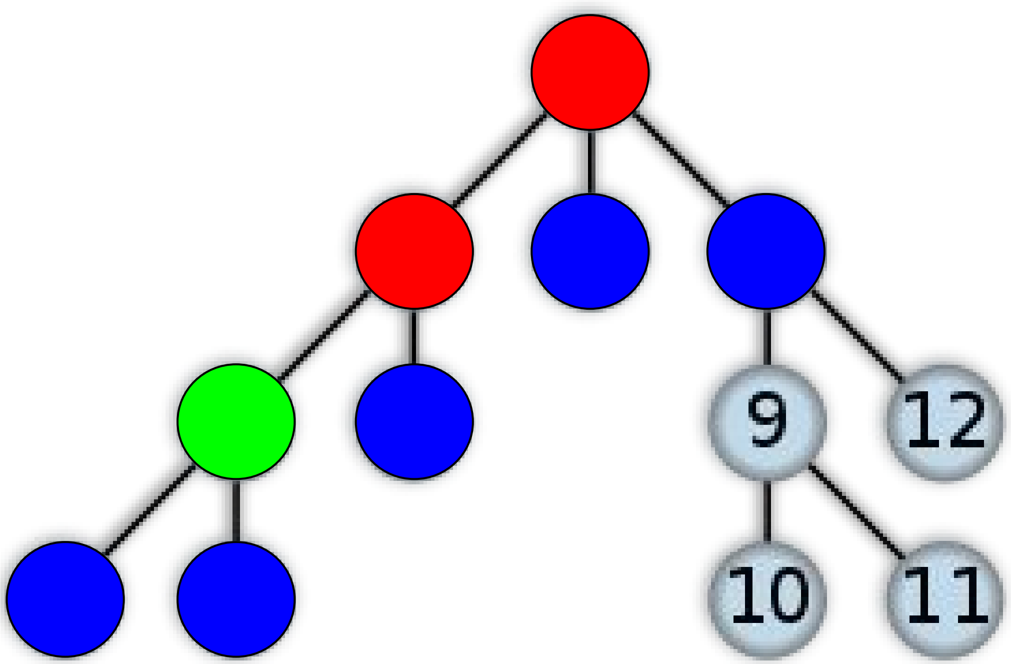
Stack: *{8, 7, 6, 3}*



Depth-first search

Current: *3*

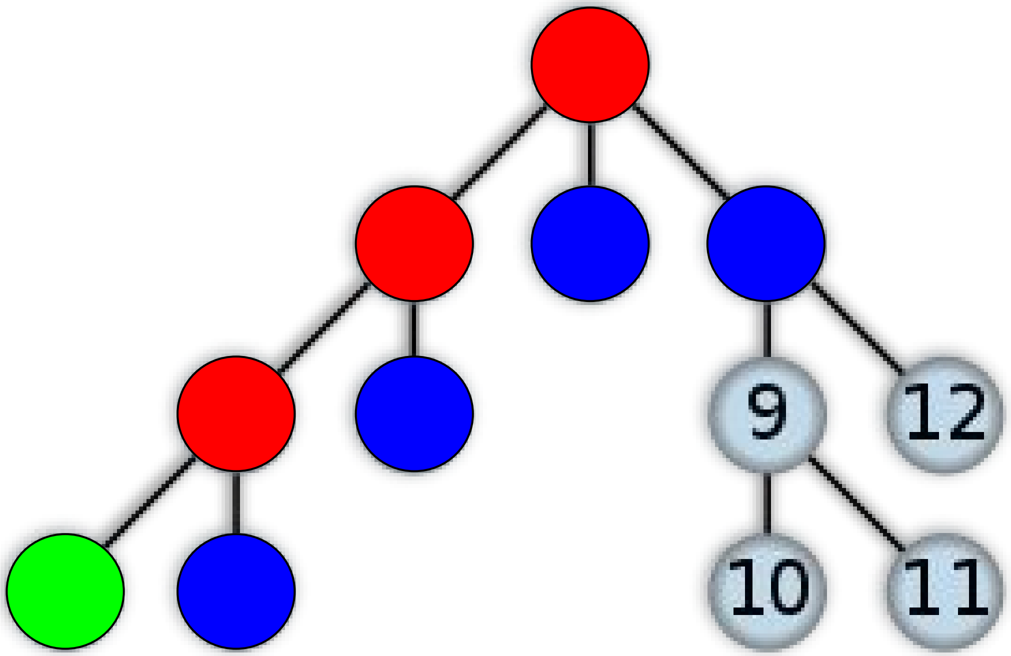
Stack: *{8, 7, 6, 5, 4}*



Depth-first search

Current: *4*

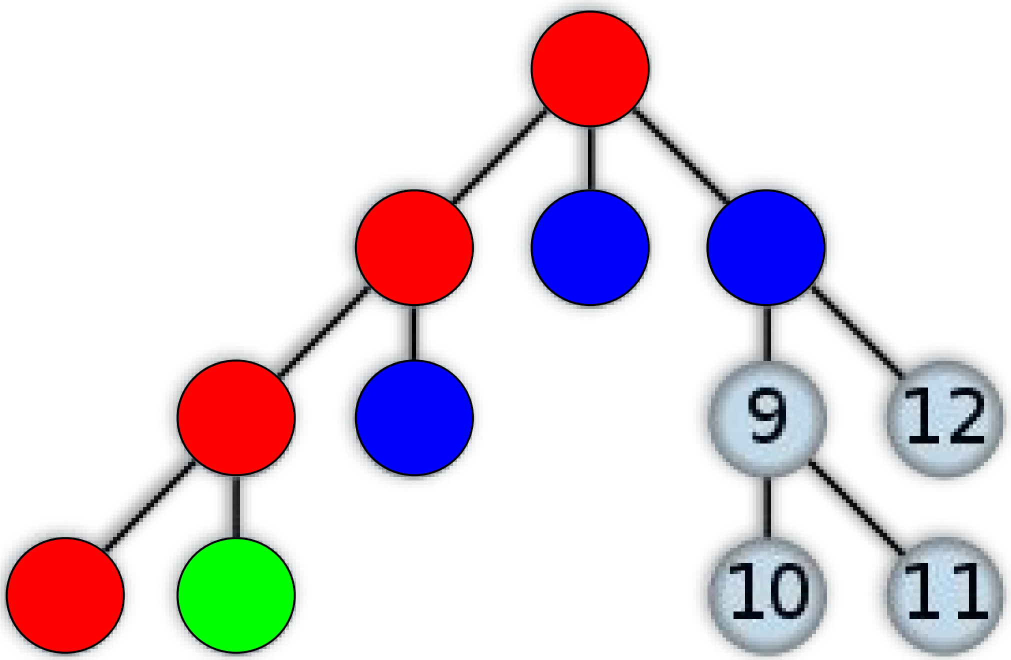
Stack: *{8, 7, 6, 5}*



Depth-first search

Current: *5*

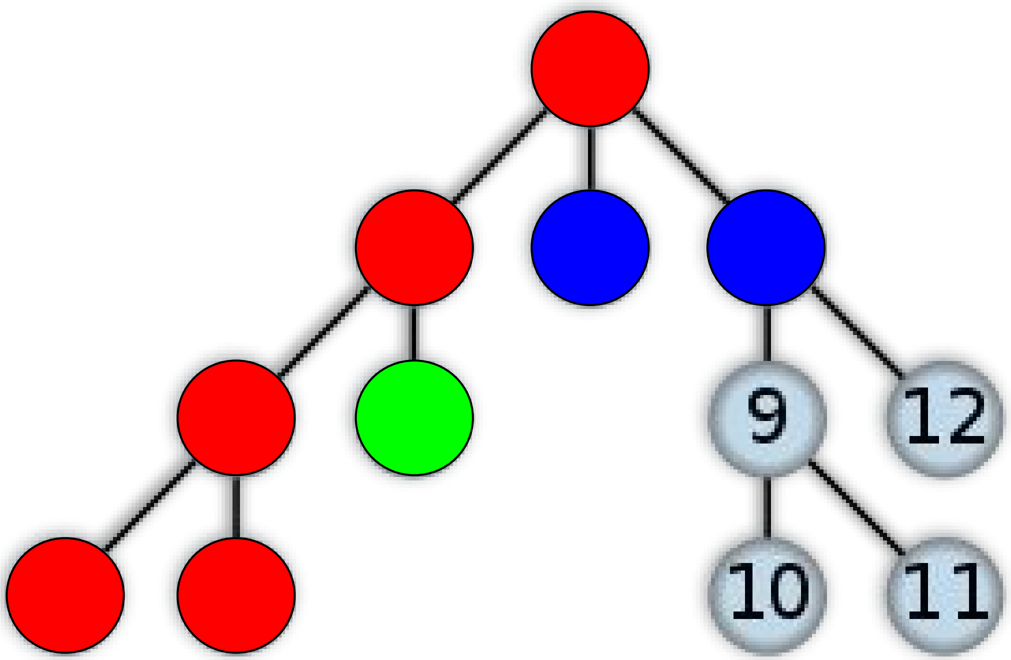
Stack: *{8, 7, 6}*



Depth-first search

Current: *6*

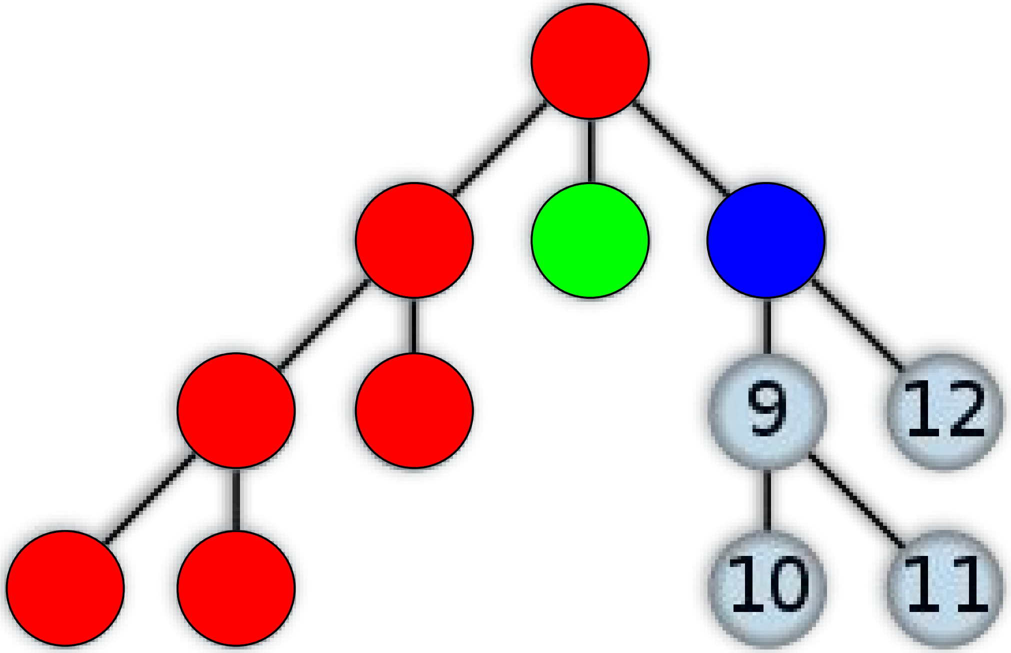
Stack: *{8, 7}*



Depth-first search

Current: 7

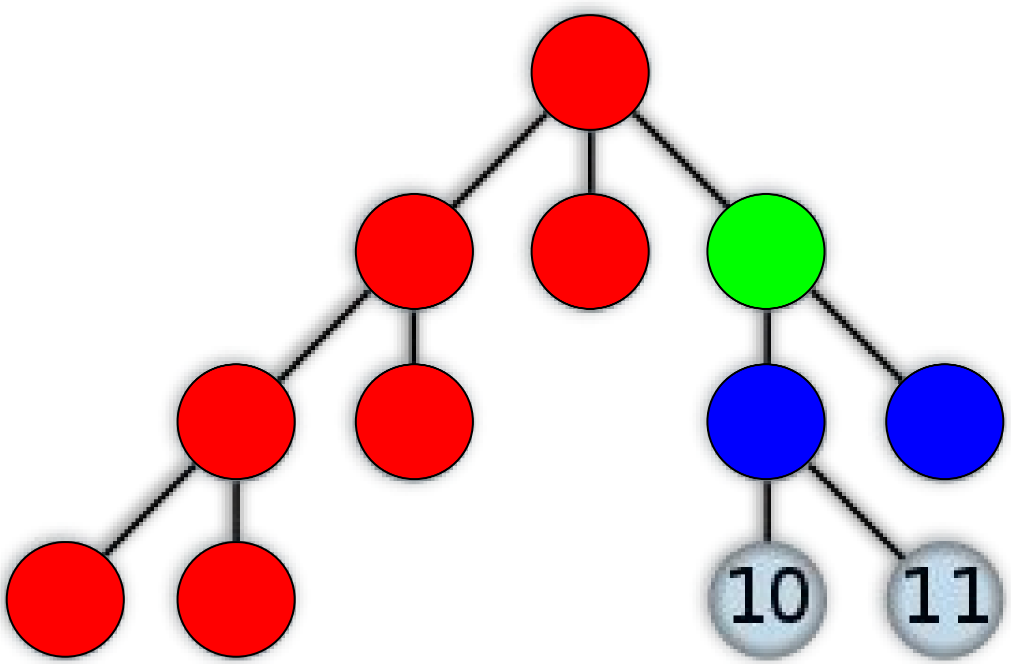
Stack: *{8}*



Depth-first search

Current: *8*

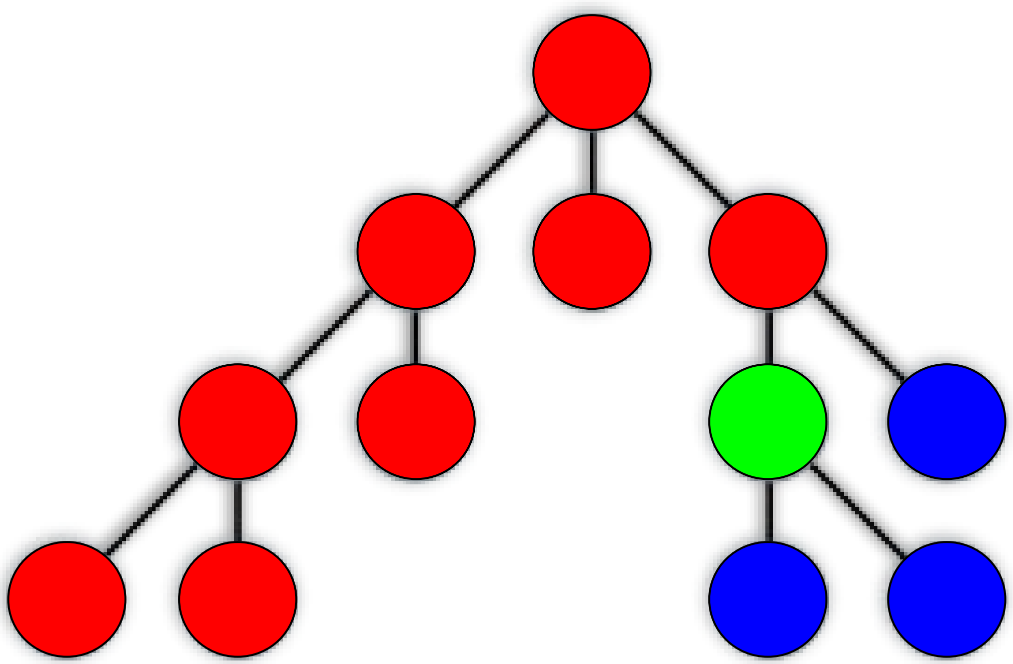
Stack: *{12, 9}*



Depth-first search

Current: *9*

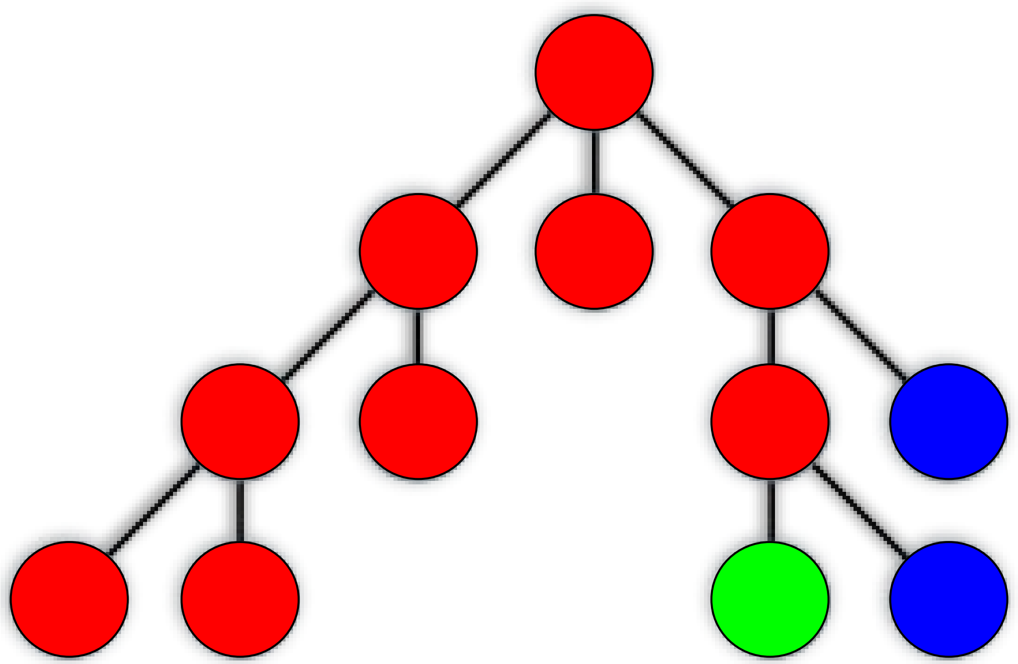
Stack: *{12, 11, 10}*



Depth-first search

Current: *10*

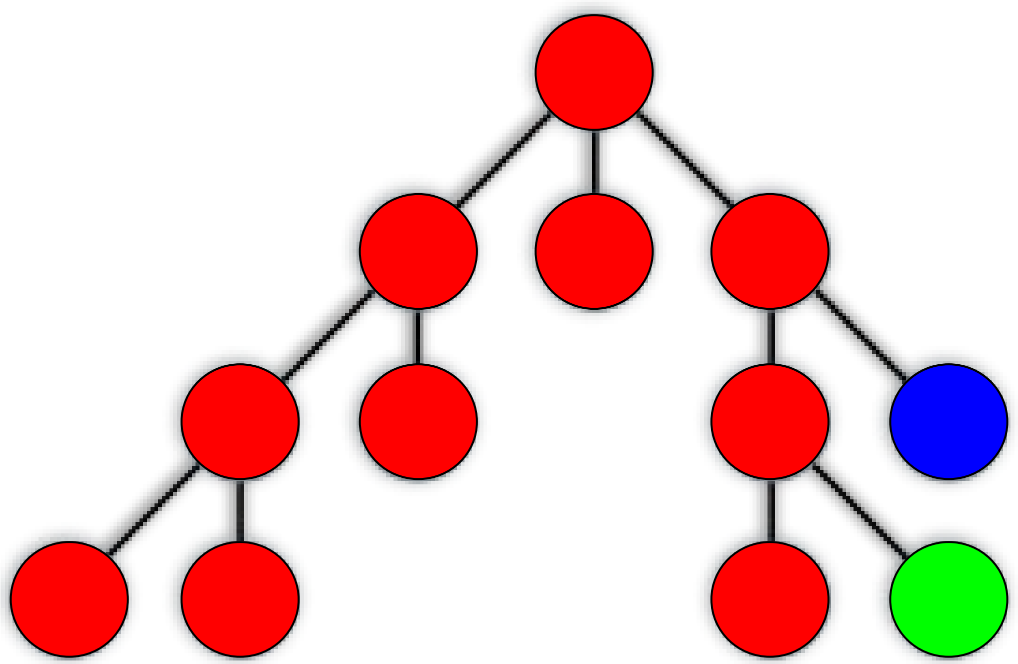
Stack: *{12, 11}*



Depth-first search

Current: *11*

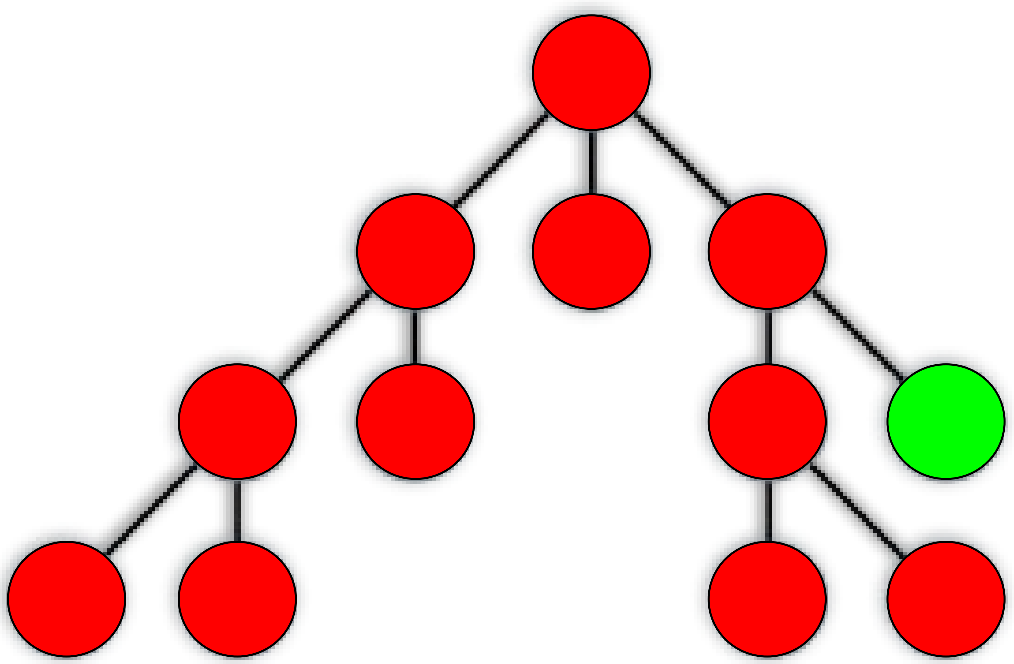
Stack: *{12}*



Depth-first search

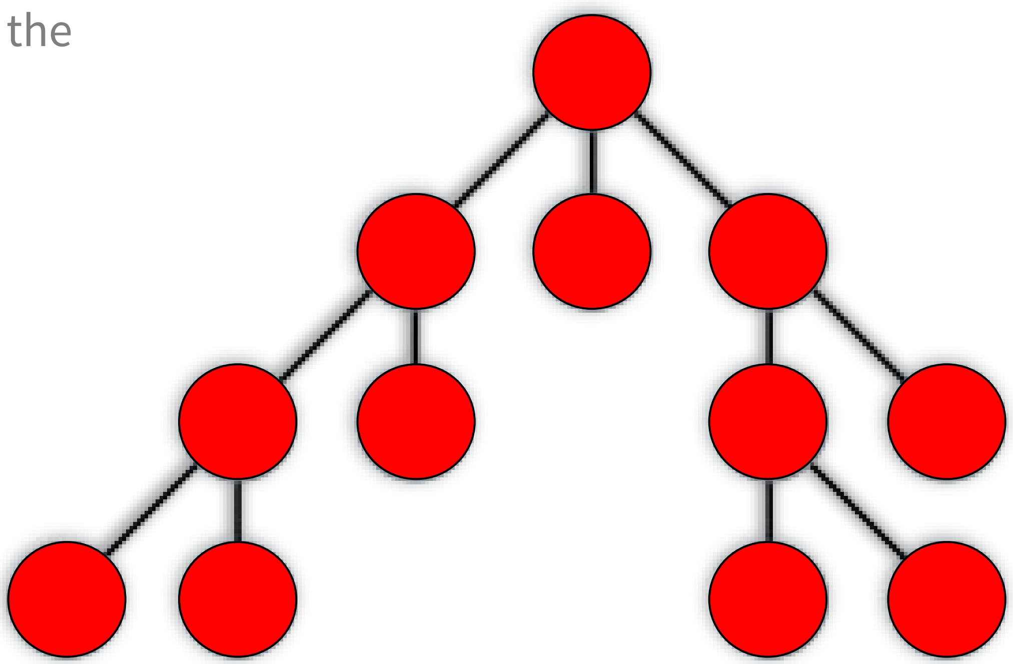
Current: *12*

Stack: *{}*



Depth-first search

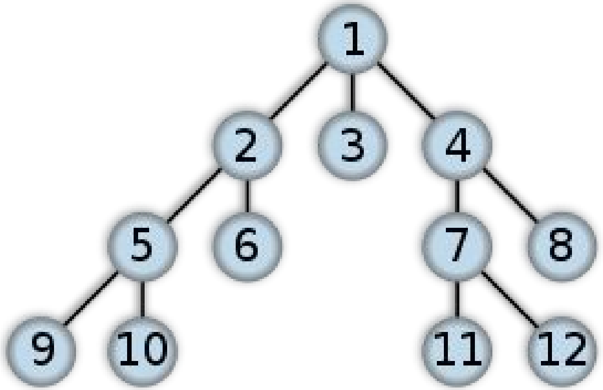
Once the *Stack* is empty, we are done with DFS, and have visited every vertex in the graph!

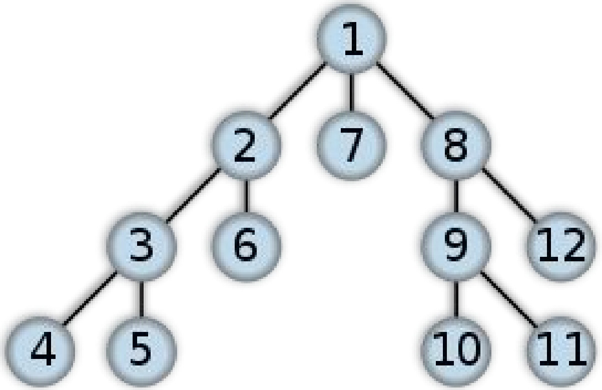


Trees

A *tree* is a special kind of graph; as mentioned before, it is an acyclic graph with *N* vertices and *N-1* edges.

The graphs that we used before for BFS and DFS were *trees*!





Tree

If we are given a graph, we can check to see if the graph is a tree by checking the following criteria:

1) If there are *N* vertices, are there *N-1* edges?
2) Is it acyclic?
    a) how can we determine if a graph is acyclic?

Problem!

Get into groups of 2-3 and solve the following problem!

spoj.com/problems/PT07Y

IMPORTANT: Reading input for this problem is weird. You will get time limit exceeded if you use *Scanner* in a normal way. Use the following method to read input:

```
public static int scan( Scanner in ) throws Exception {
    return in.nextInt();
}
```

In your main method, instantiate a *Scanner* object and pass it through as a parameter…

```
Scanner input = new Scanner(System.in);
int vertices = scan(input); // pass it through to the scan method
```

More Problems!

spoj.com/problems/BITMAP

spoj.com/problems/LABYR1

spoj.com/problems/PPATH

spoj.com/problems/MAKEMAZE

spoj.com/problems/ABCPATH

spoj.com/problems/PT07Z