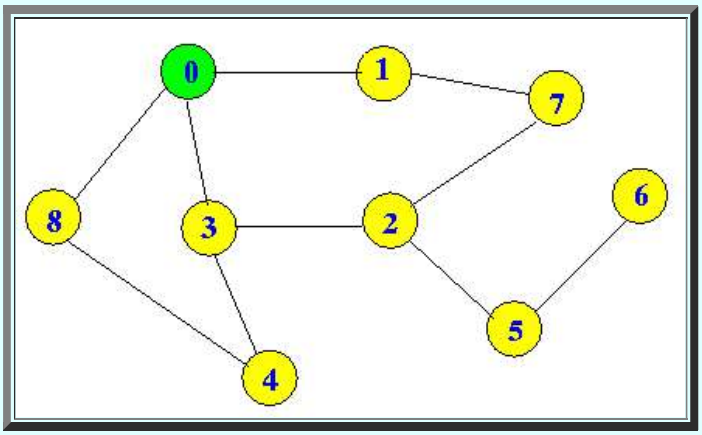
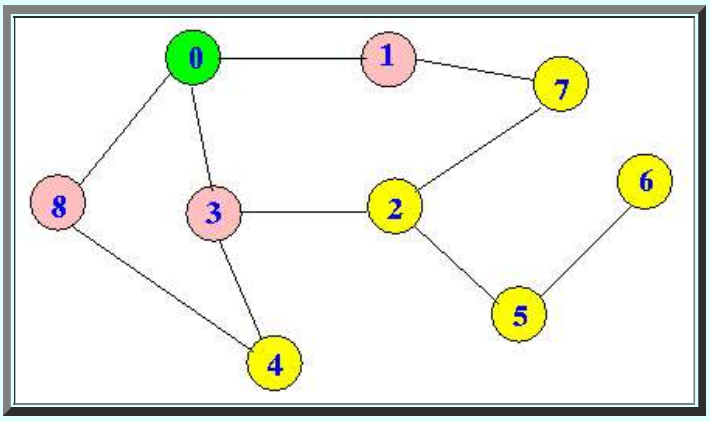
The *Breadth* First Search Graph traversal algorithm

- ***Breadth* First Search: visit the *closest* nodes *first***
  - **Description** of the **Breadth First Search** algorithm:

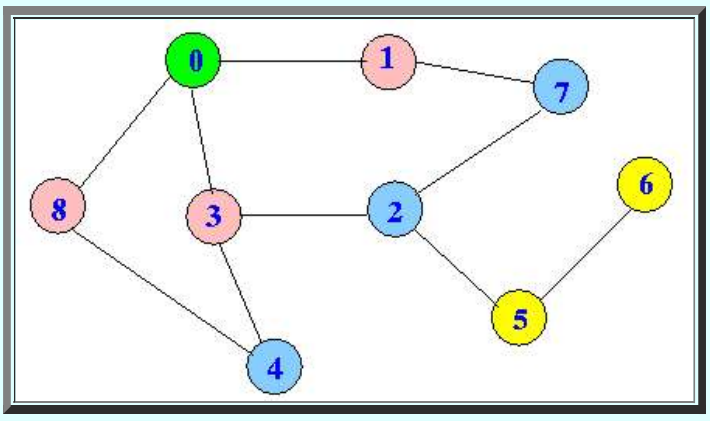
    - Start at some node (e.g., **node 0**):



    - Visit ***all* the neighbors** of **node 0** first:



    - **Then** visit the ***neighbors'* neighbors**:



    - **And so on**

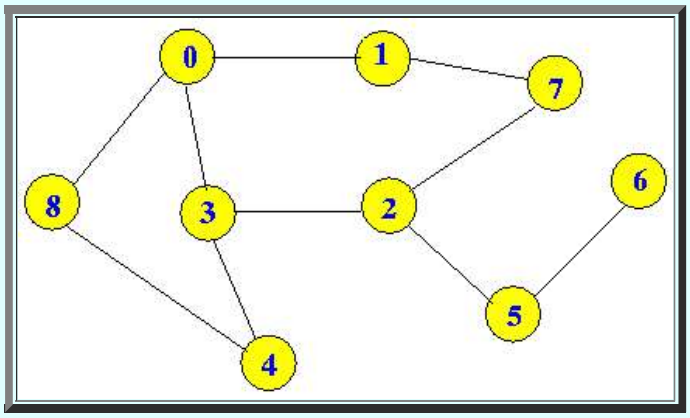
- **Implementing the BFS algorithm**
  - The **BFS** algorithm is **implmented** by:

    - Using a ***queue*** to **store** the **nodes** in the `toVisitNodes` data structure.

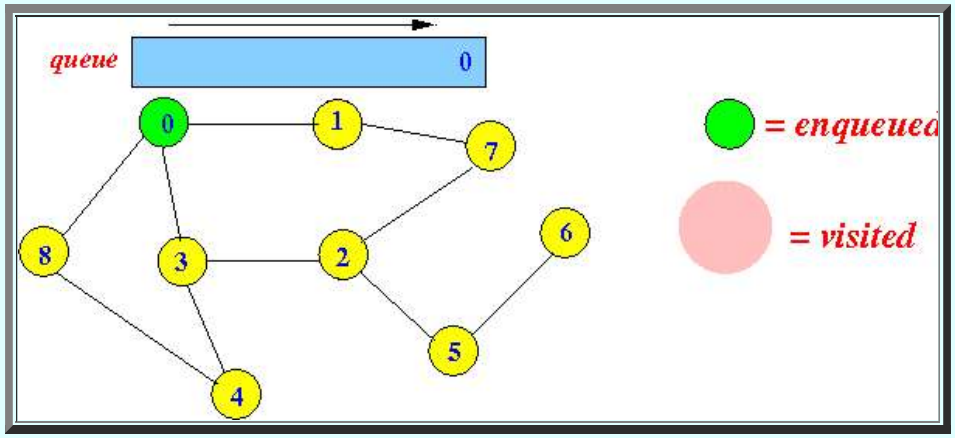
- **Pseudo code:**

```
Set all nodes to "not visited";

q = new Queue();

q.enqueue(initial node);

while ( q ≠ empty ) do
{
   x = q.dequeue();

   if ( x has not been visited )
   {
      visited[x] = true;          // Visit node x !

      for ( every edge (x, y)  /* we are using all edges ! */ )
         if ( y has not been visited )
            q.enqueue(y);        // Use the edge (x,y) !!!
   }
}
```

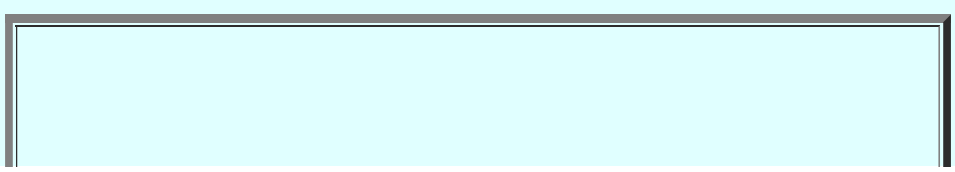
- **Example of the BFS algorithm**
  - **Example:**
    - **Graph:**

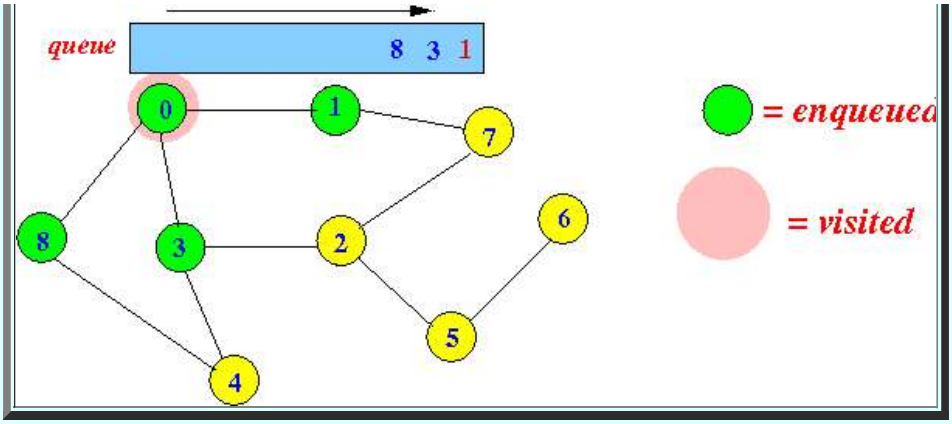


    - **Initial state:** node 0 is *enqueued*



    - **State *after* visiting 0**

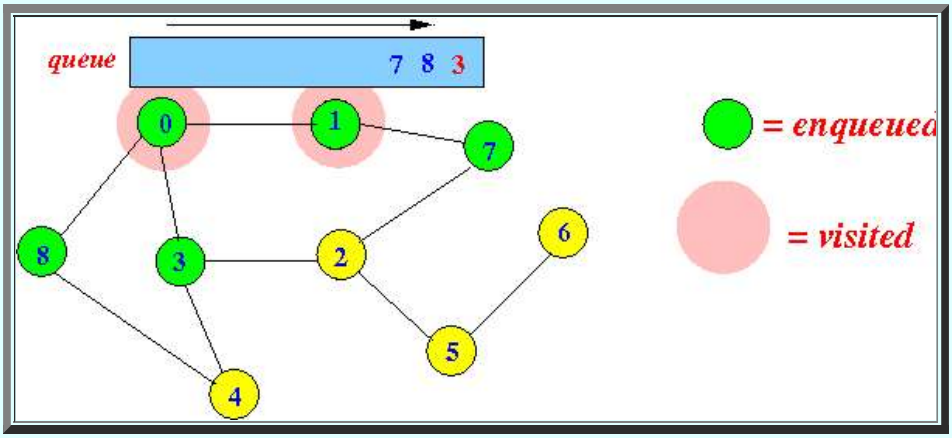




Enqueue the ***unvisited*** **neighbor nodes**: 1, 3, 8

Next, **visit** the ***first*** **node** in the **queue**: 1

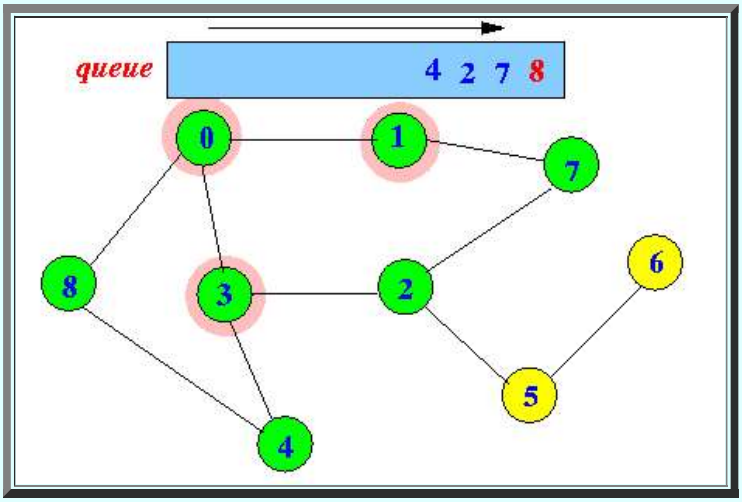
- **State *after* visiting 1**



Enqueue the ***unvisited*** **neighbor nodes**: 7

Next, **visit** the ***first*** **node** in the **queue**: 3

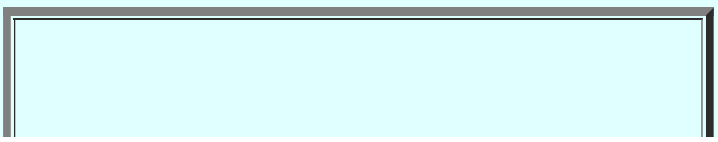
- **State *after* visiting 3**

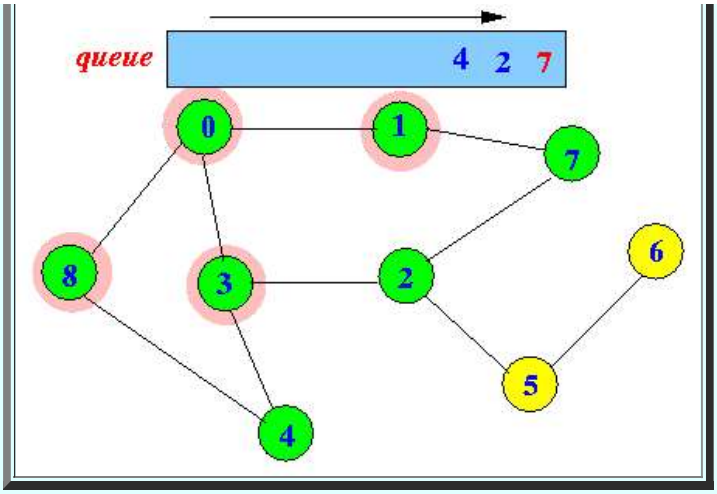


Enqueue the ***unvisited*** **neighbor nodes**: 2, 4

Next, **visit** the ***first*** **node** in the **queue**: 8

- **State *after* visiting 8**

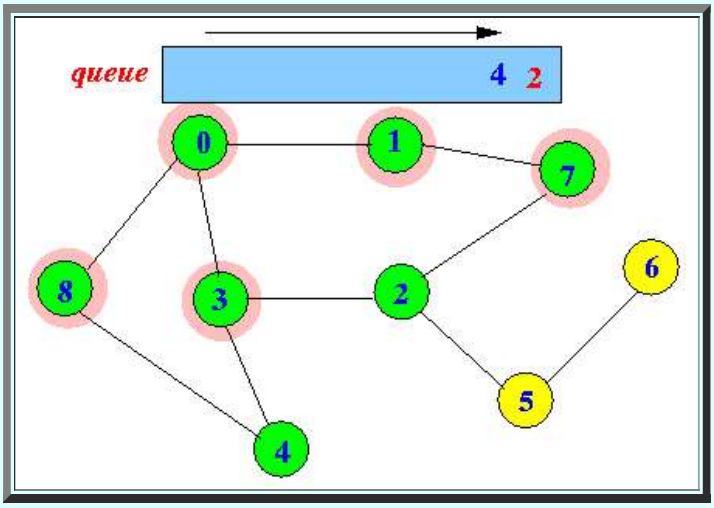




Enqueue the ***unvisited*** **neighbor nodes**: none (Note: 4 is enqueued again, but won't be visited twice, so I leave it out)

Next, **visit** the ***first*** **node** in the **queue**: **7**

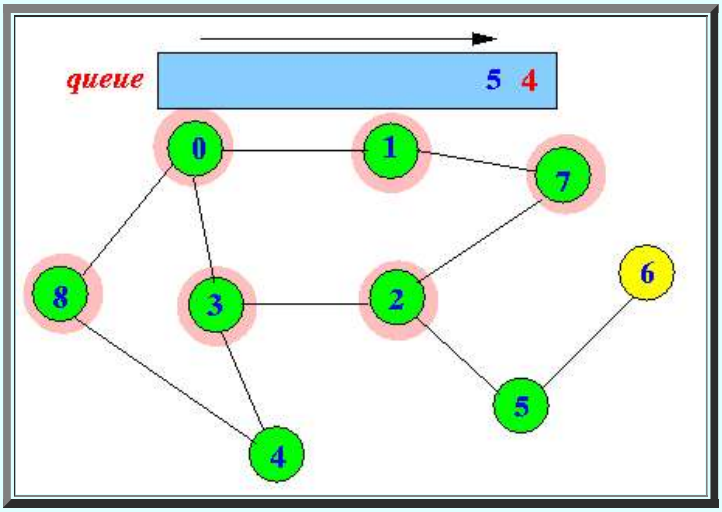
- **State *after* visiting 7**



Enqueue the ***unvisited*** **neighbor nodes**: none (Note: 2 is enqueued again, but won't be visited twice, so I leave it out)

Next, **visit** the ***first*** **node** in the **queue**: **2**

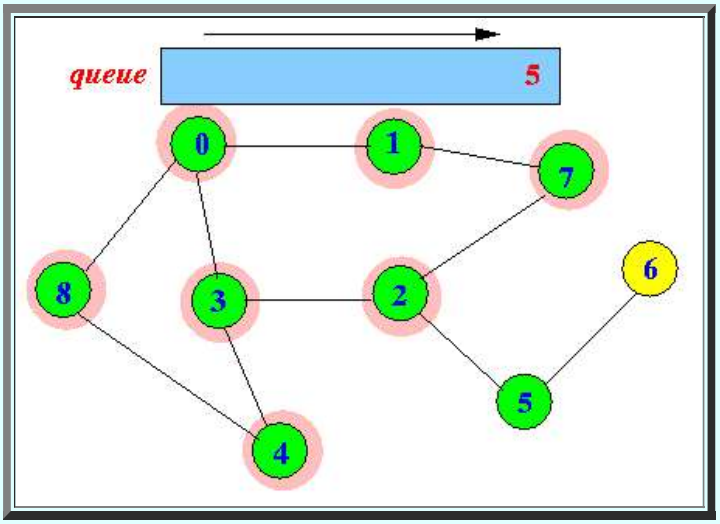
- **State *after* visiting 2**



Enqueue the ***unvisited*** **neighbor nodes**: **5**

Next, **visit** the ***first*** **node** in the **queue**: **4**

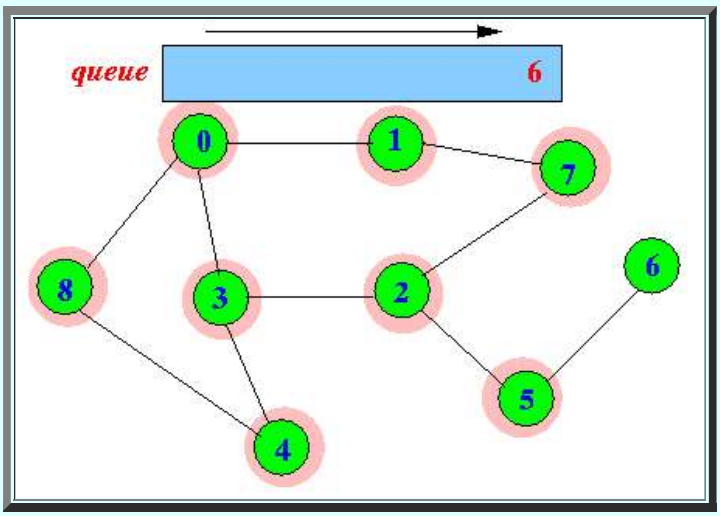
- **State *after* visiting 4**



Enqueue the ***unvisited*** **neighbor nodes**: **none**

Next, **visit** the ***first*** **node** in the **queue**: **5**

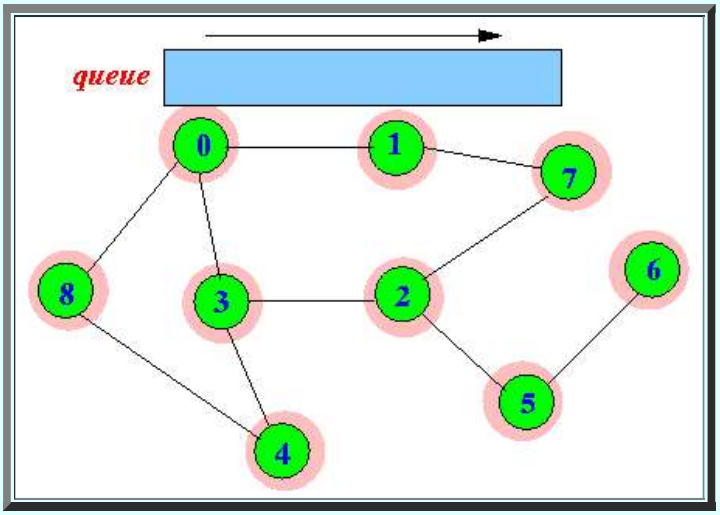
- **State *after* visiting 5**



Enqueue the ***unvisited*** **neighbor nodes**: **6**

Next, **visit** the ***first*** **node** in the **queue**: **6**

- **State *after* visiting 6**



- **DONE**

(The **queue** has become **empty**)

- **Implementation in Java**
  - **Java code:**

```
   public void BFS()
   {
      // BFS uses Queue data structure

      Queue q = new LinkedList();   // I use Queue class in Java's library

      for (i = 0; i < visited.length; i++)
         visited[i] = false;        // Clear visited[]

      q.add(0);                     // Start the "to visit" at node 0

      /* ============================================
         Loop as long as there are "active" node
         ============================================ */
      while( ! q.isEmpty() )
      {
         int nextNode;               // Next node to visit
         int i;

         nextNode = q.remove();

         if ( ! visited[nextNode] )
         {
            visited[nextNode] = true;    // Mark node as visited
            System.out.println("nextNode = " + nextNode );

            for ( i = 0; i < NNodes; i++ )
               if ( adjMatrix[nextNode][i] > 0 && ! visited[i] )
                  q.add(i);
         }
      }
   }
```

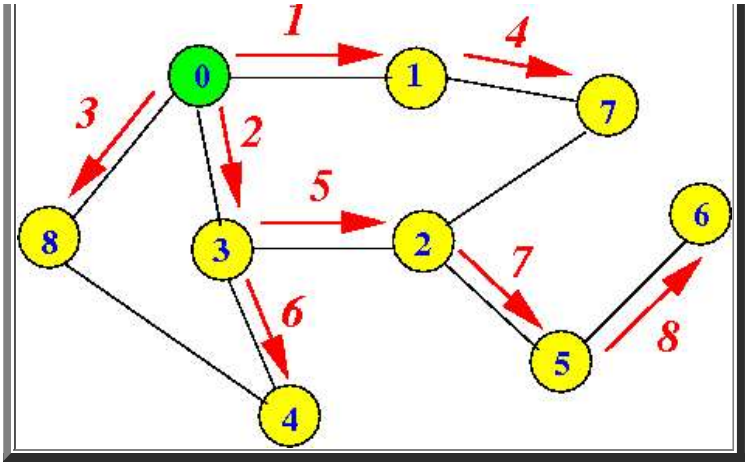
  - **Test program** using the **graph** given in this **webpage**:

```
   public static void main(String[] args)
   {
//                       0  1  2  3  4  5  6  7  8
// ===================================================
      int[][] conn = {  { 0, 1, 0, 1, 0, 0, 0, 0, 1 },  // 0
                        { 1, 0, 0, 0, 0, 0, 0, 1, 0 },  // 1
                        { 0, 0, 0, 1, 0, 1, 0, 1, 0 },  // 2
                        { 1, 0, 1, 0, 1, 0, 0, 0, 0 },  // 3
                        { 0, 0, 0, 1, 0, 0, 0, 0, 1 },  // 4
                        { 0, 0, 1, 0, 0, 0, 1, 0, 0 },  // 5
                        { 0, 0, 0, 0, 0, 1, 0, 0, 0 },  // 6
                        { 0, 1, 1, 0, 0, 0, 0, 0, 0 },  // 7
                        { 1, 0, 0, 0, 1, 0, 0, 0, 0 } };// 8


      Graph G = new Graph(conn);

      G.BFS();

   }
```

Output:

```
   nextNode = 0
   nextNode = 1
   nextNode = 3
   nextNode = 8
   nextNode = 7
   nextNode = 2
   nextNode = 4
   nextNode = 5
   nextNode = 6
```

Traversal order:



- **Example Program:** (Demo above code) *Example*

  - The **BFS** Prog file: click here
  - A **Test program**: click here

  How to run the program:

  - **Right click** on link(s) and **save** in a scratch directory
  - To compile: `javac TestProg.java`
  - To run: `java TestProg`
