CHAPTER 6
GRAPH

6.1 Learning Objectives

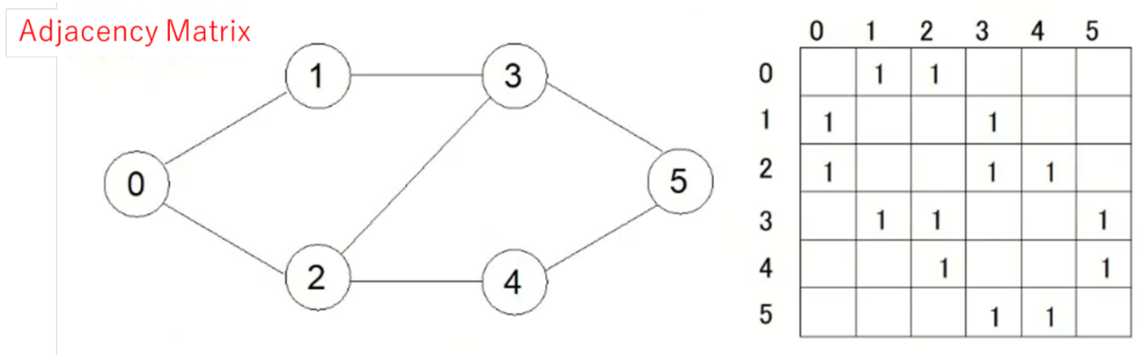
1. Students are able to understand the graph data structure.
2. Students are able to represent a graph using adjacency matrix and adjacency list.
3. Students are able to understand the concept of BFS dan DFS in a graph.

6.2 Material

6.2.1 Graph

Graph is one of the practical and often-used data structures. A graph consists of nodes and branches. A branch shows the connection/relationship between nodes. In this chapter, we are going to implement graph in java.

A graph is represented by G = {V, E}. Where V is the set of vertices/nodes and E is the set of edges. Graphs are also divided into directed and undirected graphs. A directed graph is a graph whose edges have direction information added. There are two common ways to represent a graph. One is using the adjacency matrix and the other is with the adjacency list. Figure 1 shows an example of an adjacency matrix of a graph with 6 nodes and 7 edges.



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

Figure 1. Graph representation using adjacency matrix

In this case, the adjacency matrix is a 6x6 square matrix. Node-0 is connected to node-1 and node-2, thus “1” exists in column-1 and column-2 of row-0 of the matrix. Similarly, Node-1 is connected with Node-0 and Node-3, thus “1” exists in column-0 and column-3 of row-1 and so on. The number “1” exists if there is an edge connecting two nodes.

If the same graph is represented by linked lists, it will look like Figure 2. Node-0 is connected node-1 and node-2, so there will be a link list with the following connection 0→1→2. Since there is nothing else after 2, the part that becomes the pointer next to it contains a null. Same thing applies to node-1 that connects with node-0 and node-3 (1→0→3).

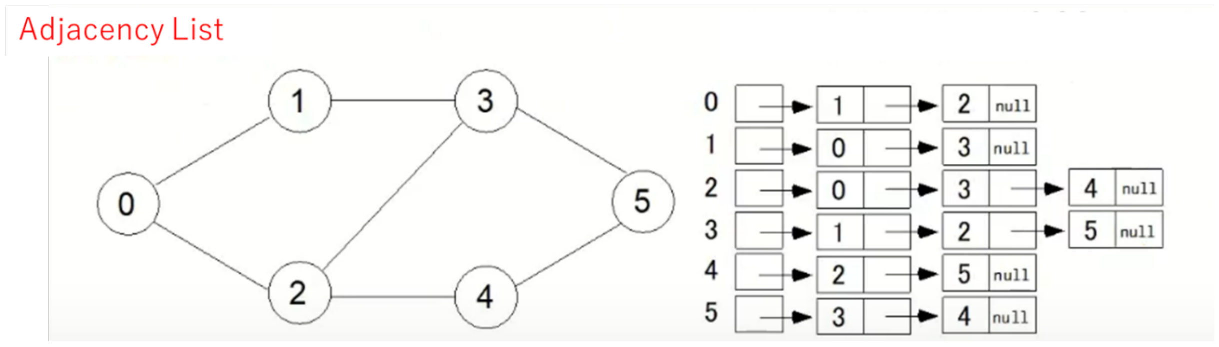


Figure 2. Graph representation using adjacency list

In real applications, often we need to traverse/search a graph to find some important information (such as shortest route in google map etc). Traversal (search) in a graph is almost similar to that in a tree, the different is, unlike trees, graphs may contain cycles, so we may come to the same node again. To avoid processing a node more than once, we can use a boolean visited array. There are two types of graph traversal: BFS (Breadth First Search/Traversal) and DFS (Depth First Search/Traversal).

BFS is particularly useful for finding the shortest path on unweighted graphs. It starts at some arbitrary node of a graph and explores the neighbour nodes first, before moving to the next level neighbours. BFS uses a queue data structure to track which node to visit next. Upon reaching a new node, BFS adds it to the queue to visit it later.

Meanwhile, DFS itself is not too useful, but when augmented to perform other tasks such as count connected components, determine connectivity, or find bridge/articulation points then DFS starts to shine. As its name implies, DFS plunges depth first into a graph without regard for which edge it takes next until it cannot go any further at which point it backtracks and continues.

6.2.2. Graphical Illustration of BFS Algorithm

Here, the state of each node will be represented by three different colors: white, orange, and red. White nodes are nodes that have not been discovered, orange nodes are ones that have been discovered but not yet visited, and red nodes are ones that have been visited. A queue will be used to store the orange nodes. Suppose we want to start traversing the graph shown in Figure 3 from node-0. Since we start from node-0, node-0 is considered to be discovered, so we add node-0 (or simply 0) to the queue.

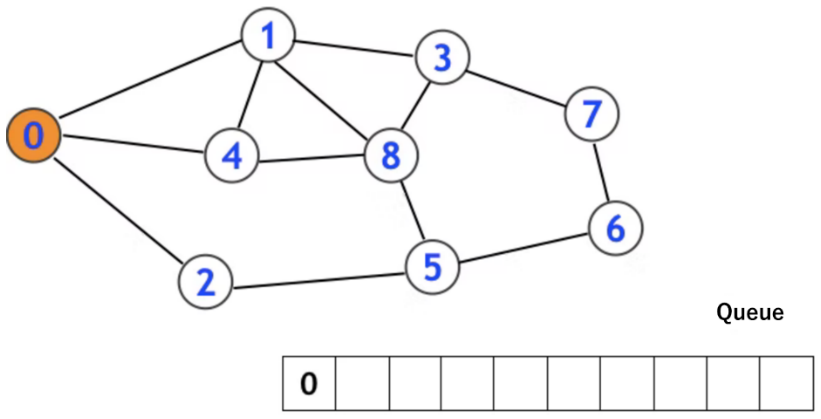


Figure 3. Discovered node-0

Next, we retrieve and remove the front end of the queue (i.e., 0) and visit node-0, so its color turns red as shown in Figure 4. From node-0, we discover node-(1, 4, 2), hence we add 1, 4, and 2 to the queue. These nodes are then shown in orange. Also, Figure 4, the numerical value "1" is written at nodes-(1, 4, 2), which means that these nodes can be reached from node-0 in 1 step. Similarly, node-0 can be reached from node-0 (itself) in 0 steps.

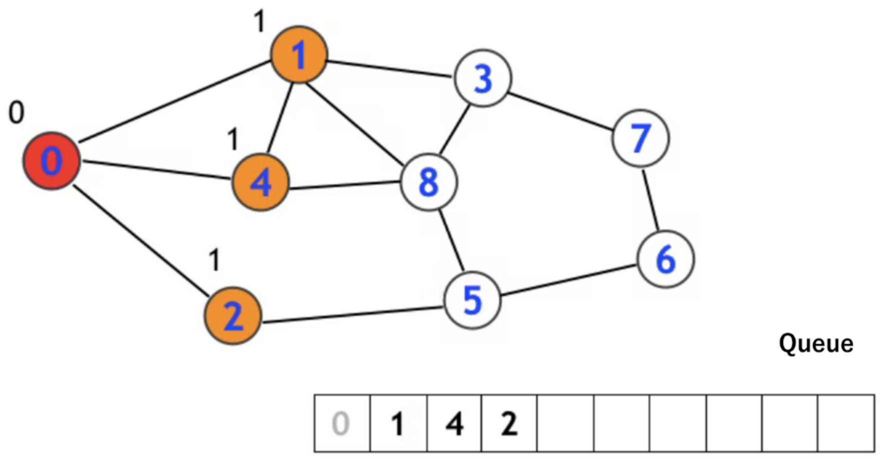


Figure 4. Visit node-0, newly discovered nodes-(1, 4, 2)

Next, we retrieve and remove the front end of the queue (i.e., 1) and visit node-1 and we newly discover nodes-(3, 8), hence we add 3 and 8 to the queue. At this time, "2" is entered at nodes-(3, 8) as in Figure 5. This means that nodes-(3, 8) can be reached in 2 steps starting from node-0.

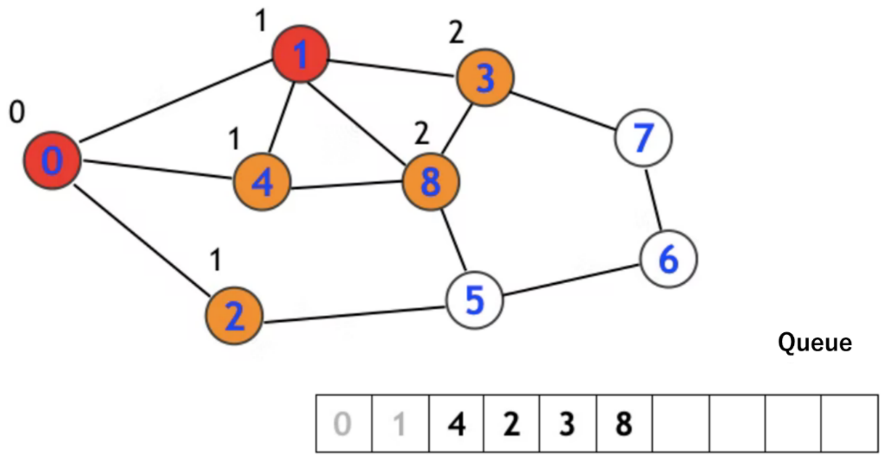


Figure 5. Visit node-1, newly discovered nodes-(3, 8)

Next, we retrieve and remove the front end of the queue (i.e., 4) and visit node-4. Since there is no white node directly connected to node-4, we finish the step without adding any new node to the queue.

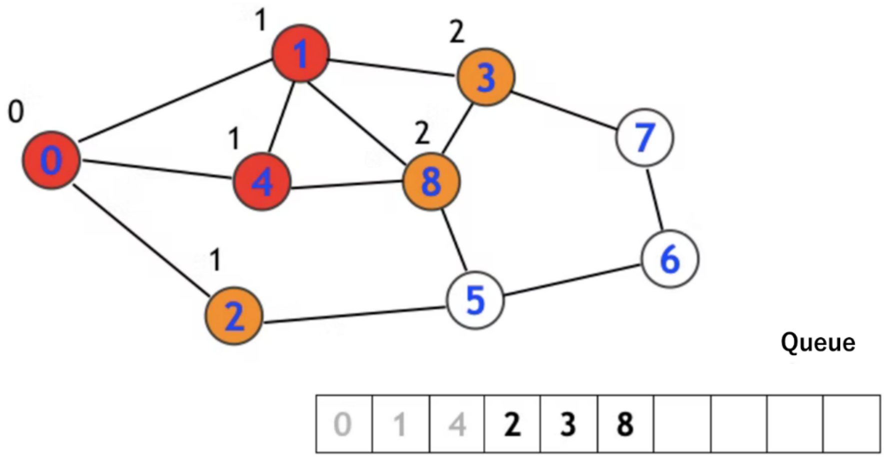


Figure 6. Visit node-4, no newly discovered node

Next, we retrieve and remove the front end of the queue (i.e., 2) and visit node-2. Then, we newly discover node-5 and hence 5 is newly added to the queue.

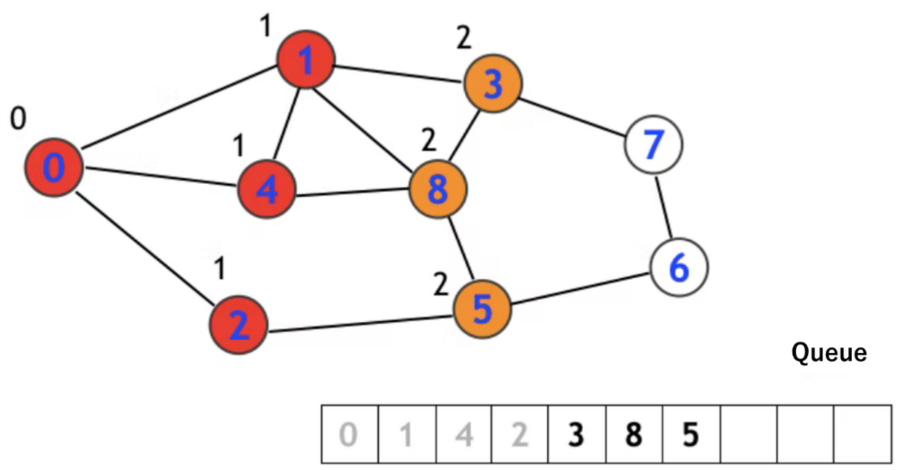


Figure 7. Visit node-2, newly discovered node-5

After that, by following similar procedures as mentioned above, the color of the graph and the queue will change as shown in the following figures. After visiting node-3 and 8, respectively:

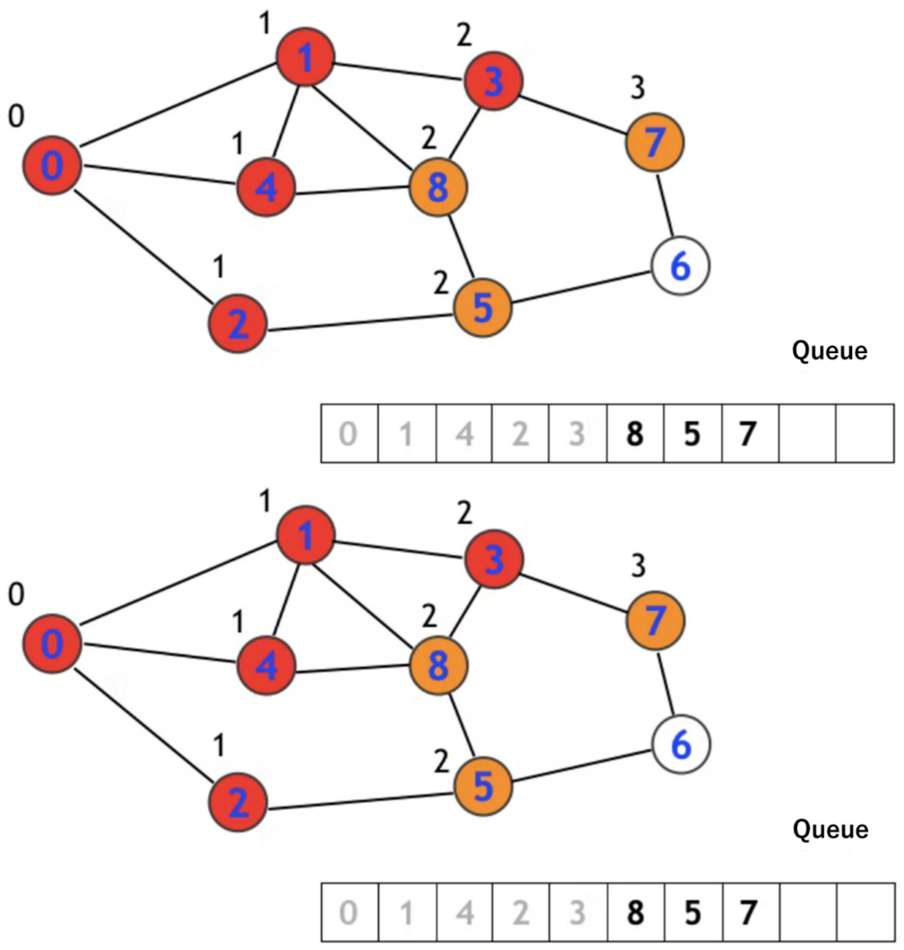


Figure 8. Visit node-3→newly discovered node-7, visit node-8→no newly discovered node

After visiting node-5, node-7, and node-6, respectively:

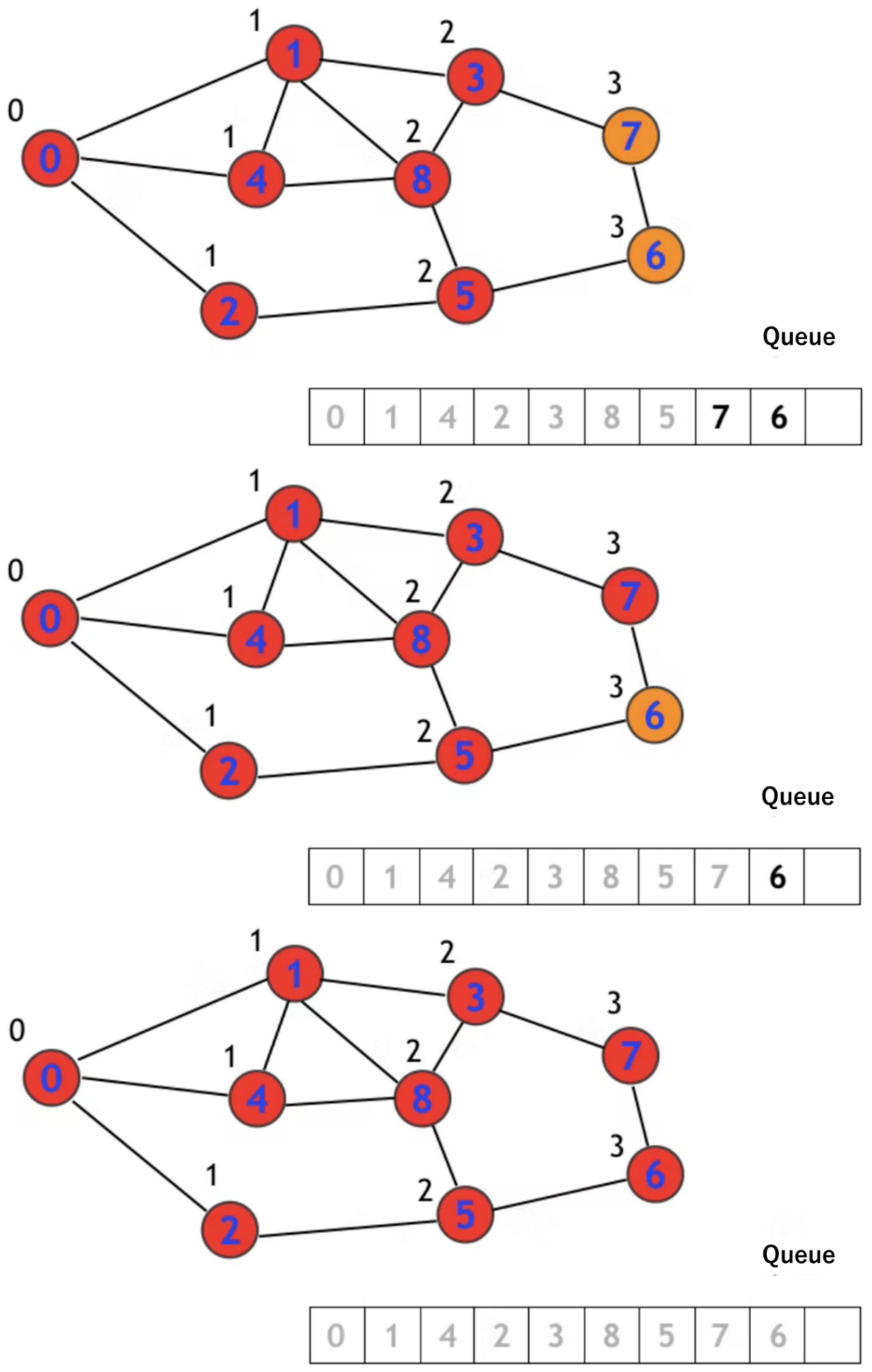
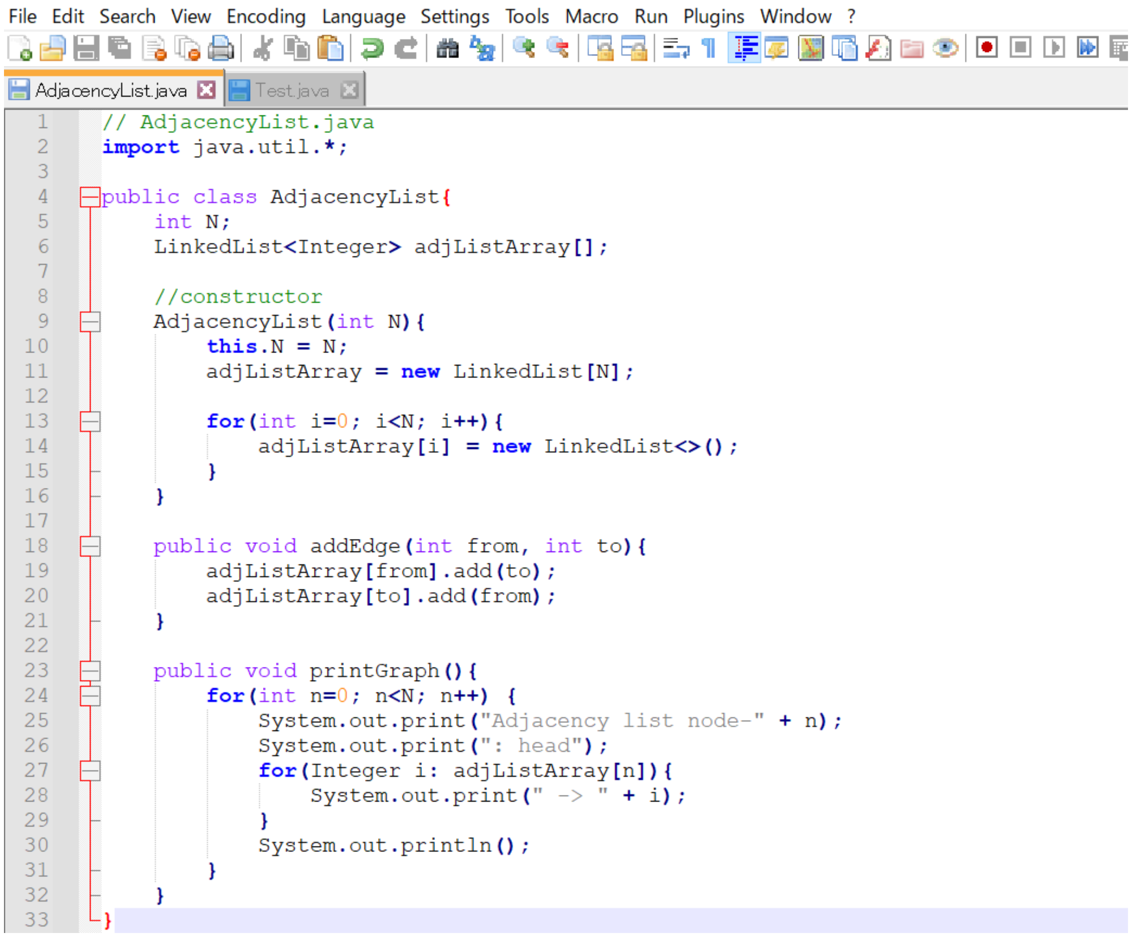


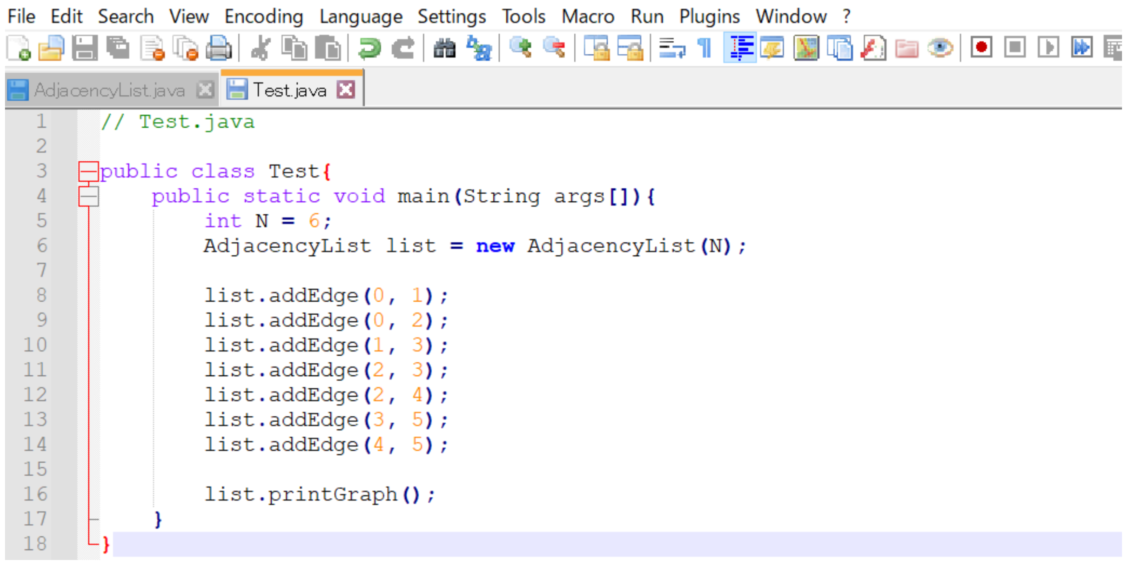
Figure 9. Visit node-5→newly discovered node-6 (all nodes are discovered), visit node-7, and finally visit node-6→graph traversal is completed

6.2.3 Java Implementation

The java implementation for representing the graph shown in Figure 2 is as shown below. Here, there are two classes: AdjacencyList and Test. As their names imply, one is use for the linked list representation while the other one is for testing.

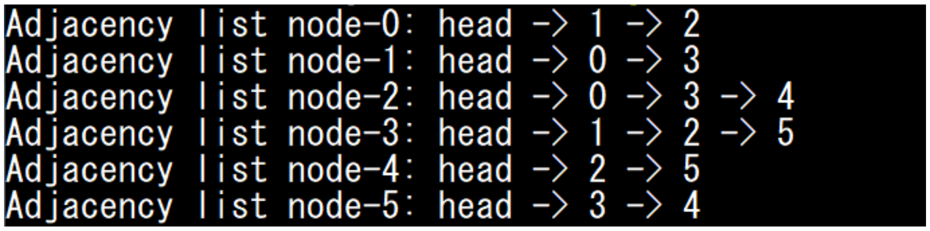

File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
AdjacencyList.java
Test.java

```java
// AdjacencyList.java
import java.util.*;

public class AdjacencyList{
    int N;
    LinkedList<Integer> adjListArray[];

    //constructor
    AdjacencyList(int N){
        this.N = N;
        adjListArray = new LinkedList[N];

        for(int i=0; i<N; i++){
            adjListArray[i] = new LinkedList<>();
        }
    }

    public void addEdge(int from, int to){
        adjListArray[from].add(to);
        adjListArray[to].add(from);
    }

    public void printGraph(){
        for(int n=0; n<N; n++) {
            System.out.print("Adjacency list node-" + n);
            System.out.print(": head");
            for(Integer i: adjListArray[n]){
                System.out.print(" -> " + i);
            }
            System.out.println();
        }
    }
}
```


File Edit Search View Encoding Language Settings Tools Macro Run Plugins Window ?
AdjacencyList.java
Test.java

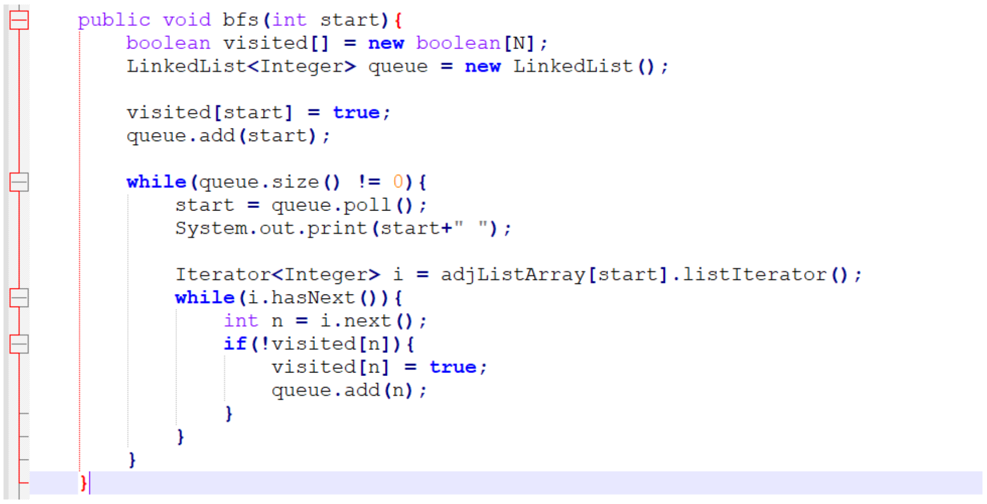
```java
// Test.java

public class Test{
    public static void main(String args[]){
        int N = 6;
        AdjacencyList list = new AdjacencyList(N);

        list.addEdge(0, 1);
        list.addEdge(0, 2);
        list.addEdge(1, 3);
        list.addEdge(2, 3);
        list.addEdge(2, 4);
        list.addEdge(3, 5);
        list.addEdge(4, 5);

        list.printGraph();
    }
}
```

The result after running Test is as follows.



```
Adjacency list node-0: head -> 1 -> 2
Adjacency list node-1: head -> 0 -> 3
Adjacency list node-2: head -> 0 -> 3 -> 4
Adjacency list node-3: head -> 1 -> 2 -> 5
Adjacency list node-4: head -> 2 -> 5
Adjacency list node-5: head -> 3 -> 4
```

Add the following method to AdjacencyList.java for the BFS implementation.



```
public void bfs(int start){
    boolean visited[] = new boolean[N];
    LinkedList<Integer> queue = new LinkedList();

    visited[start] = true;
    queue.add(start);

    while(queue.size() != 0){
        start = queue.poll();
        System.out.print(start+" ");

        Iterator<Integer> i = adjListArray[start].listIterator();
        while(i.hasNext()){
            int n = i.next();
            if(!visited[n]){
                visited[n] = true;
                queue.add(n);
            }
        }
    }
}
```

In this implementation, the poll() and the add() methods of Queue Interface are used. The poll() method is used to retrieve and remove the front end of the queue, and the add() method is used to add newly discovered nodes to the queue. Try calling the bfs() method from Test.java using list.bfs(start); (substitute start with the node number from which you want to start the search/travers). Check whether the traversing order matches the theoretical result.

6.3 Assignments

1. Try implementing the adjacency matrix-version for the graph representation.
2. Modify the method bfs() so that it works for the adjacency matrix-version.
3. Read more about DFS and add a method named dfs() that performs Depth First Search to the graph.