Graph Implementation

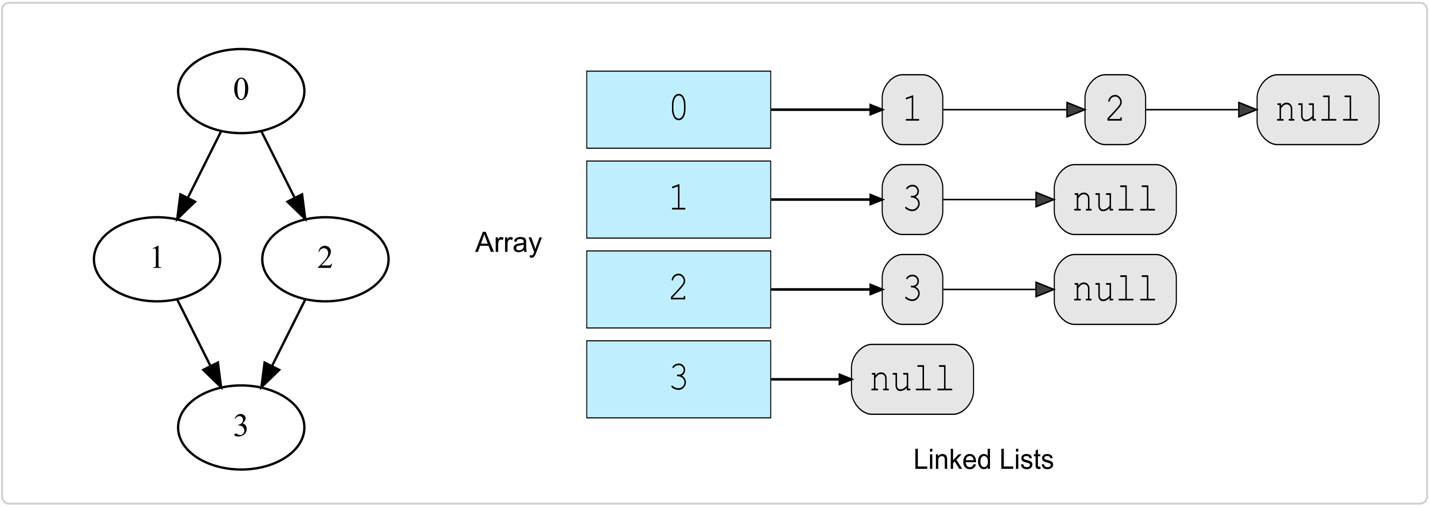
This lesson will cover the implementation of a directed Graph via the Adjacency List. We will also go through the time complexity of basic graph operations.

We'll cover the following

- Introduction
- Structure of Graph Class
  - Explanation (Describing Graph Structure)
    - Variables
    - Methods
- Code Snippet
  - Explanation (Method Calls)
  - 1. Graph g=new Graph (int);
  - 2. addEdge (int source, int destination);
  - 3. main();

Introduction

In this lesson, we will implement a **Directed Graph** in java using an *Adjacency List*.



Structure of Graph Class

Graph class consists of two data members: a total number of vertices in the graph, and an array of the linked list to store the adjacency list of vertices. Below is a code snippet of our Graph class:

```java
public class Graph{
    int vertices; //Total number of vertices in graph
    DoublyLinkedList<Integer> adjacencyList[]; //An array of linked lists to store adjacent

    void printGraph(); //Prints Graph (Adjaceny List)
    void addEdge(int source,int destination); //Adds an Edge from source vertex to destinat
}
```

Note: We are using a Doubly Linked list with Tail for the adjacency list implementation in the `Graph` class. The purpose of using a DLL with Tail is that the `insertAtEnd()` operation has $O(1)$ time complexity. We will be using this operation to add edges in the graph. It is not necessary to use DLL for Graphs. *Alternatively*, an SLL with a Tail can also be used here to achieve the same time complexity.

Explanation (Describing Graph Structure)

As you can see in the code snippet above, we have a basic structure of Graph class.

Variables

`vertices` to store the total number of vertices of the graph.

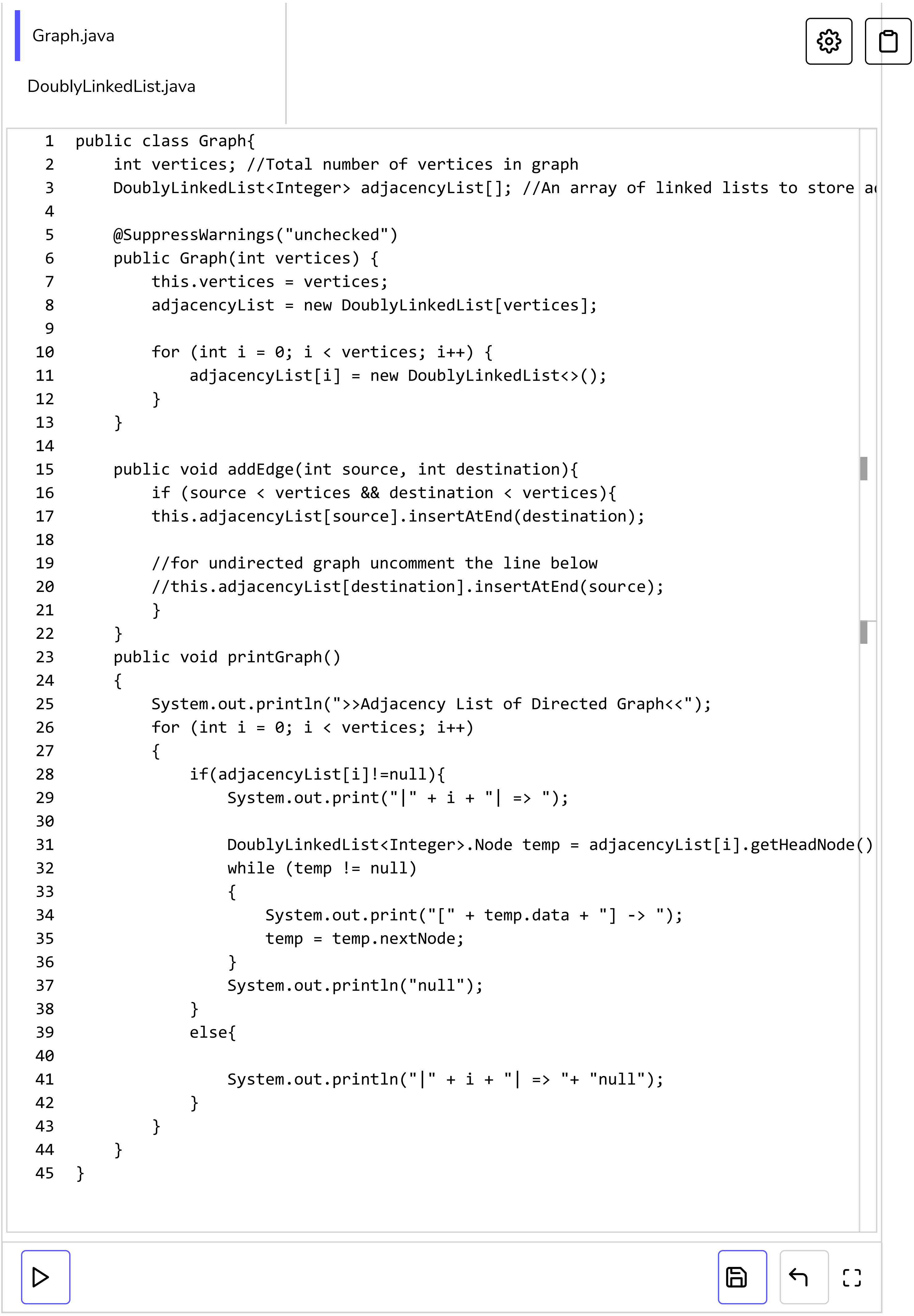
`adjacencyList` to store an array of linked lists. Each index of the array represents a vertex of the graph, and the `linked list` represents the adjacent vertices.

Methods

`printGraph()` method prints the graph (adjacency list). `addEdge()` creates a source and destination vertex and connects them with an edge.

Code Snippet

main.java



```java
public class Graph{
    int vertices; //Total number of vertices in graph
    DoublyLinkedList<Integer> adjacencyList[]; //An array of linked lists to store a

    @SuppressWarnings("unchecked")
    public Graph(int vertices) {
        this.vertices = vertices;
        adjacencyList = new DoublyLinkedList[vertices];

        for (int i = 0; i < vertices; i++) {
            adjacencyList[i] = new DoublyLinkedList<>();
        }
    }

    public void addEdge(int source, int destination){
        if (source < vertices && destination < vertices){
        this.adjacencyList[source].insertAtEnd(destination);

        //for undirected graph uncomment the line below
        //this.adjacencyList[destination].insertAtEnd(source);
        }
    }
    public void printGraph()
    {
        System.out.println(">>Adjacency List of Directed Graph<<");
        for (int i = 0; i < vertices; i++)
        {
            if(adjacencyList[i]!=null){
                System.out.print("|" + i + "| => ");

                DoublyLinkedList<Integer>.Node temp = adjacencyList[i].getHeadNode()
                while (temp != null)
                {
                    System.out.print("[" + temp.data + "] -> ");
                    temp = temp.nextNode;
                }
                System.out.println("null");
            }
            else{

                System.out.println("|" + i + "| => "+ "null");
            }
        }
    }
}
```

Explanation (Method Calls)

Let’s break the above code down into different sections:

1. Graph g=new Graph (int);

In the constructor, we create an array of size equal to the total number of vertices. In this case, our graph has **4** `vertices`. Hence constructor takes **4** as a parameter.

2. addEdge (int source, int destination);

Directed Graph:

The constructor of the `Graph` class creates an array of `linked lists`, where the size of the array is equal to the number of vertices in the graph. After creating the array, we need to add `nodes to linked lists`. Each node of the `linked list` will represent the adjacent vertex of that array index. The function `addEdge` takes two parameters, source vertex number, and destination vertex number. First, it checks whether the value of the source and destination vertex is valid or not. If the values are valid, it goes to index of the array equal to source number received in parameter and adds a node to the corresponding list. The added node contains the destination vertex number. The node is added to the list by the following line of code:

```
adjacencyList[source].insertAtEnd(destination);
```

3. main();

The main will create a graph like the figure given above.

It first calls the constructor of the graph class: `new Graph(x)` where **x** is the number of vertices in the graph.

- `g.addEdge(0,1)`; – Creates an edge from **0** to **1**
- `g.addEdge(0,2)`; – Creates an edge from **0** to **2**
- `g.addEdge(1,3)`; – Creates an edge from **1** to **3**
- `g.addEdge(2,3)`; – Creates an edge from **2** to **3**

After that, it simply prints the graph by calling `printGraph()` function.

In the next lesson, let's compare the time complexities of graph operations using the adjacency matrix and the adjacency list approach.

Back
Representation of Graphs

Next
Complexities of Graph Operations

✓ Mark as Completed

Report an Issue

Ask a Question
(https://discuss.educative.io/tag/graph-implementation__graphs__data-structures-for-coding-interviews-in-java)