Graph Implementation

This lesson will cover the implementation of a directed graph via adjacency list in Python.

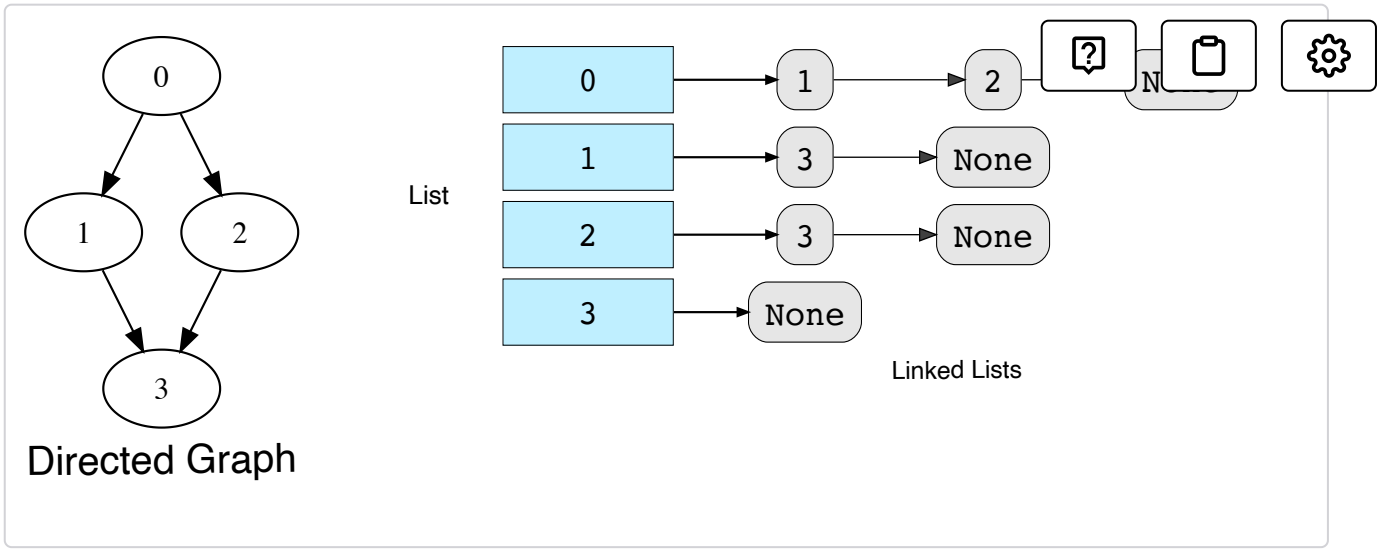
We'll cover the following

Introduction

At this point, we've understood the theoretical concepts of graphs. In this lesson, we will use this knowledge to implement the graph data structure in Python. Our graph will be directed and have no bidirectional edges.

The implementation will be based on the **adjacency list** model. The linked list class we created earlier will be used to represent adjacent vertices.

As a refresher, here is the illustration of the graph we'll be producing using an adjacency list:



The Graph Class

Graph class consists of two data members:

- The total number of vertices in the graph
- A list of linked lists to store adjacent vertices

So let's get down to the implementation!

Graph.py

```
class Graph:
    def __init__(self, vertices):
        # Total number of vertices
        self.vertices = vertices
        # Defining a list which can hold multiple LinkedLists
        # equal to the number of vertices in the graph
        self.array = []
        # Creating a new LinkedList for each vertex/index of the list
        for i in range(vertices):
            # Appending a new LinkedList on each array index
            self.array.append(LinkedList())
```

We've laid down the foundation of our `Graph` class. The variable `vertices` contains an integer specifying the total number of vertices.

The second component is `array`, which will act as our adjac[illegible]is simply have to run a loop and create a linked list for each vertex.

Additional Functionality

Now, we'll add two methods to make this class functional:

1. `print_graph()` - Prints the content of the graph
2. `add_edge()` - Connects a source with a destination

main.py

Graph.py

LinkedList.py

Node.py

```
from Graph import Graph

g = Graph(4)
g.add_edge(0, 2)
g.add_edge(0, 1)
g.add_edge(1, 3)
g.add_edge(2, 3)

g.print_graph()

print(g.array[1].get_head().data)
```

Let's break down the two new functions that we've implemented.

add_edge (self, source, destination)

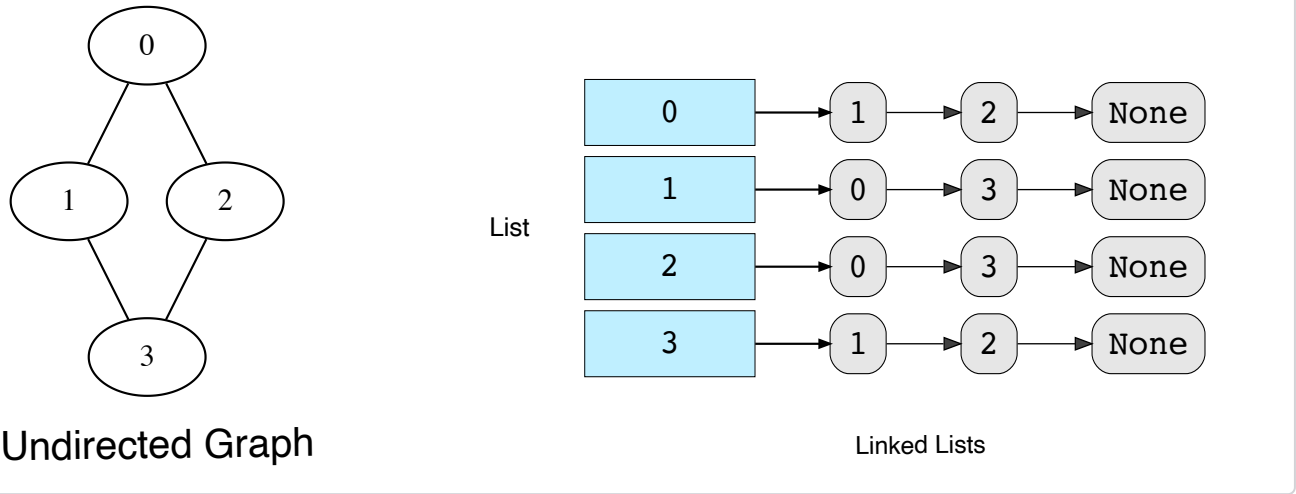
Thanks to the graph constructor, `source` and `destination` a[illegible] stored as indices of our list. This function simply inserts a destination vertex into the adjacency linked list of the source vertex by running the following line of code:

```
array[source].insert_at_head(destination)
```

One important thing to note is that we are implementing a directed graph, so `add_edge(0, 1)` is not equal to `add_edge(1, 0)`. In the case of an undirected graph, we will have to create an edge from the source to the destination and from the destination to the source, making it a bidirectional edge:

```
array[source].insert_at_head(destination)
array[destination].insert_at_head(source)
```

The figure below illustrates the corresponding undirected graph with bidirectional edges.



`addEdge()` will not work if `source` is less than zero and greater than or equal to the number of vertices. Likewise, `destination` also has to be

greater than or equal to 0 and less than the number of vertic[illegible] th[illegible]u[illegible] production code, you need to cover the error handling of these edge cases.

print_graph(self)

This function uses a simple nested loop to iterate through the adjacency list. Each linked list is being traversed here.

We've seen the `add_edge` and `print_graph` methods. What do you think is the time complexity of these functions? The next lesson will answer this question.

Interviewing soon? We've partnered with Hired so that companies apply to you instead of you applying to them. See how

Back

Representation of Graphs

Next

Complexities of Graph Operations

Completed

Report an Issue