8.6. Implementation

Using dictionaries, it is easy to implement the adjacency list in Python. In our implementation of the Graph abstract data type we will create two classes (see Listing 1 and Listing 2), `Graph` , which holds the master list of vertices, and `Vertex` , which will represent each vertex in the graph.

Each `Vertex` uses a dictionary to keep track of the vertices to which it is connected, and the weight of each edge. This dictionary is called `connectedTo` . The listing below shows the code for the `Vertex` class. The constructor simply initializes the `id` , which will typically be a string, and the `connectedTo` dictionary. The `addNeighbor` method is used add a connection from this vertex to another. The `getConnections` method returns all of the vertices in the adjacency list, as represented by the `connectedTo` instance variable. The `getWeight` method returns the weight of the edge from this vertex to the vertex passed as a parameter.

Listing 1

```
class Vertex:
    def __init__(self,key):
        self.id = key
        self.connectedTo = {}

    def addNeighbor(self,nbr,weight=0):
        self.connectedTo[nbr] = weight

    def __str__(self):
        return str(self.id) + ' connectedTo: ' + str([x.id for x in self.connectedTo])

    def getConnections(self):
        return self.connectedTo.keys()

    def getId(self):
        return self.id

    def getWeight(self,nbr):
        return self.connectedTo[nbr]
```

The `Graph` class, shown in the next listing, contains a dictionary that maps vertex names to vertex objects. In Figure 4 (AnAdjacencyList.html#fig-adjlist) this dictionary object is represented by the shaded gray box. `Graph` also provides methods for adding vertices to a graph and connecting one vertex to another. The `getVertices` method returns the names of all of the vertices in the graph. In addition, we have implemented the `__iter__` method to make it easy to iterate over all the vertex objects in a particular graph. Together, the two methods allow you to iterate over the vertices in a graph by name, or by the objects themselves.

Listing 2

(AnAdjacencyList.html) (TheWordLadderP

```
class Graph:
    def __init__(self):
        self.vertList = {}
        self.numVertices = 0

    def addVertex(self,key):
        self.numVertices = self.numVertices + 1
        newVertex = Vertex(key)
        self.vertList[key] = newVertex
        return newVertex

    def getVertex(self,n):
        if n in self.vertList:
            return self.vertList[n]
        else:
            return None

    def __contains__(self,n):
        return n in self.vertList

    def addEdge(self,f,t,weight=0):
        if f not in self.vertList:
            nv = self.addVertex(f)
        if t not in self.vertList:
            nv = self.addVertex(t)
        self.vertList[f].addNeighbor(self.vertList[t], weight)

    def getVertices(self):
        return self.vertList.keys()

    def __iter__(self):
        return iter(self.vertList.values())
```

Using the `Graph` and `Vertex` classes just defined, the following Python session creates the graph in Figure 2 (VocabularyandDefinitions.html#fig-dgsimple). First we create six vertices numbered 0 through 5. Then we display the vertex dictionary. Notice that for each key 0 through 5 we have created an instance of a `Vertex` . Next, we add the edges that connect the vertices together. Finally, a nested loop verifies that each edge in the graph is properly stored. You should check the output of the edge list at the end of this session against Figure 2 (VocabularyandDefinitions.html#fig-dgsimple).

(AnAdjacencyList.html) (TheWordLadderP

```
>>> g = Graph()
>>> for i in range(6):
...     g.addVertex(i)
>>> g.vertList
{0: <adjGraph.Vertex instance at 0x41e18>,
 1: <adjGraph.Vertex instance at 0x7f2b0>,
 2: <adjGraph.Vertex instance at 0x7f288>,
 3: <adjGraph.Vertex instance at 0x7f350>,
 4: <adjGraph.Vertex instance at 0x7f328>,
 5: <adjGraph.Vertex instance at 0x7f300>}
>>> g.addEdge(0,1,5)
>>> g.addEdge(0,5,2)
>>> g.addEdge(1,2,4)
>>> g.addEdge(2,3,9)
>>> g.addEdge(3,4,7)
>>> g.addEdge(3,5,3)
>>> g.addEdge(4,0,1)
>>> g.addEdge(5,4,8)
>>> g.addEdge(5,2,1)
>>> for v in g:
...     for w in v.getConnections():
...         print("( %s , %s )" % (v.getId(), w.getId()))
...
( 0 , 5 )
( 0 , 1 )
( 1 , 2 )
( 2 , 3 )
( 3 , 4 )
( 3 , 5 )
( 4 , 0 )
( 5 , 4 )
( 5 , 2 )
```

You have attempted 1 of 1 activities on this page

user not logged in

(AnAdjacencyList.html) (TheWordLadderP