SOURCE CODE

```
/*
William Kelley
OOP - Assignment 2
Graphs
Source for Help: A lot of stackoverflow, nothing in particular was used but
definitely pulled a lot of references and similarities from.
https://www.geeksforgeeks.org/breadth-first-search-or-bfs-for-a-graph/
https://www.geeksforgeeks.org/depth-first-search-or-dfs-for-a-graph/
*/

#include <iostream>
#include <cstdlib>
#include <string>
#include <cstring>
#include <iomanip>
#include <vector>
#include <list>
#include <algorithm>
#include <fstream>
#include <sstream>

std::ifstream infile;
std::ofstream outfile;

using namespace std;

template <class T>
class Graph {
public:
	Graph() {};
	~Graph<T>() {};
	virtual bool adjacent(T x, T y) { return false; };
	virtual vector<T> neighbors(T x) {
		vector<T> graph;
		graph.push_back(1);
		return graph;
	};
	virtual void addNode() { cout << "Parent class 'addNode()'" << endl; };
	virtual void deleteNode() { cout << "Parent class 'deleteNode()'" <<
endl; };
	virtual void addEdge() { cout << "Parent class 'addEdge()'" << endl; };
	virtual void deleteEdge() { cout << "Parent class 'deleteEdge()'" <<
endl; };
	virtual void BFS() { cout << "Parent class 'BFS()'" << endl; };
	virtual void DFSUtil() { cout << "Parent class 'DFSUtil()'" << endl; };
	virtual void DFS() { cout << "Parent class 'DFS()'" << endl; }
};

template <class T>
class AdjacencyMatrix : public Graph<T> {
private:
	std::vector<T> adjmatrix;
```

```
public:
	AdjacencyMatrix() {};
	~AdjacencyMatrix() {};
	std::vector<T> getGraph() const;
	void inputGraph(string s);
	void printGraph() const;
	void printGraph(vector<T>) const;
	bool adjacent(T x, T y);
	std::vector<T> neighbors(T x);
	void addNode(T x);
	void deleteNode(T x);
	void addEdge(T x, T y);
	void deleteEdge(T x, T y);
	void BFS(T v);
	void DFSUtil(T v, bool visited[]);
	void DFS(T v);
};

template <class T>
class AdjacencyList : public Graph<T> {
private:
	std::list<T> adjlist;
public:
	AdjacencyList() {};
	~AdjacencyList() {};
	std::list<T> getGraph() const;
	void inputGraph(string s);
	void printGraph() const;
	void printGraph(vector<T>) const;
	bool adjacent(T x, T y);
	std::vector<T> neighbors(T x);
	void addNode(T x);
	void deleteNode(T x);
	void addEdge(T x, T y);
	void deleteEdge(T x, T y);
	void BFS(T v);
	void DFSUtil(T v, bool visited[]);
	void DFS(T v);
};

template<class T>
std::vector<T> AdjacencyMatrix<T>::getGraph() const
{
	return adjmatrix;
}

template<class T>
void AdjacencyMatrix<T>::inputGraph(string s)
{
	string input;
	T array[20];

	infile.open(s);
	while (std::getline(infile, input))
```

```
	{
		T x;
		replace(input.begin(), input.end(), ',', ' ');
		replace(input.begin(), input.end(), ':', ' ');
		input.erase(std::remove(input.begin(), input.end(), ' '),
input.end());
		for (auto i = 0; i < input.length(); ++i)
		{
			array[i] = input[i] - '0';
		}
		x = array[0];
		for (auto j = 0; j < input.length(); ++j)
		{
			addEdge(x, array[j]);
		}
	}
	infile.close();
}

template<class T>
void AdjacencyMatrix<T>::printGraph() const
{
	cout << endl;
	for (auto i = adjmatrix.begin(); i != adjmatrix.end(); ++i)
	{
		std::cout << (*i / 10) << "->" << (*i % 10) << '\n';
	}
	cout << endl;
}

template<class T>
void AdjacencyMatrix<T>::printGraph(vector<T> x) const
{
	for (auto i = x.begin(); i != x.end(); ++i)
	{
		std::cout << (*i / 10) << "->" << (*i % 10) << '\n';
	}
	cout << endl;
}

template<class T>
bool AdjacencyMatrix<T>::adjacent(T x, T y)
{
	T point = (x * 10) + y;
	std::vector<int>::iterator it;
	it = find(adjmatrix.begin(), adjmatrix.end(), point);
	if (it != adjmatrix.end())
	{
		return true;
	}
	else
	{
		return false;
	}
```

```
}

template<class T>
std::vector<T> AdjacencyMatrix<T>::neighbors(T x)
{
	std::vector<T> returnVector;
	std::vector<T> currentVector = getGraph();
	T singleDigit = NULL;
	T nodeId = (x * 10);
	T max = nodeId + 9;
	for (nodeId; nodeId <= max; ++nodeId)
	{
		for (auto i = currentVector.begin(); i != currentVector.end(); +
+i)
		{
			if (*i == nodeId)
			{
				returnVector.push_back(*i);
			}
			else
			{
				void;
			}
		}
	}
	cout << endl;

	return returnVector;
}

template<class T>
void AdjacencyMatrix<T>::addNode(T x)
{
	T point = x + (x * 10);		// this allows for the user to combine
the points into a single value to be stored
	adjmatrix.erase(std::remove(adjmatrix.begin(), adjmatrix.end(), point),
adjmatrix.end());
	adjmatrix.push_back(point);
	sort(adjmatrix.begin(), adjmatrix.end());
}

template<class T>
void AdjacencyMatrix<T>::deleteNode(T x)
{
	T point = (x * 10);
	for (T i = 0; i <= 9; ++i)
	{
		adjmatrix.erase(std::remove(adjmatrix.begin(), adjmatrix.end(),
point+i), adjmatrix.end());
		for (T i = 0; i <= 9; ++i) {
			adjmatrix.erase(std::remove(adjmatrix.begin(),
adjmatrix.end(), ((i * 10) + (point/10))), adjmatrix.end());
		}
	}
```

```
	sort(adjmatrix.begin(), adjmatrix.end());
}

template<class T>
void AdjacencyMatrix<T>::addEdge(T x, T y)
{
	T point = (x * 10) + y;
	adjmatrix.erase(std::remove(adjmatrix.begin(), adjmatrix.end(), point),
adjmatrix.end()); //deleting edge if it exists and re-adding just for ease
	adjmatrix.push_back(point);
	sort(adjmatrix.begin(), adjmatrix.end());
}

template<class T>
void AdjacencyMatrix<T>::deleteEdge(T x, T y)
{
	T point = (x * 10) + y;
	std::vector<int>::iterator it;
	it = find(adjmatrix.begin(), adjmatrix.end(), point);
	if (it != adjmatrix.end())
	{
		cout << "Edge found, deleteing edge." << endl;
		adjmatrix.erase(std::remove(adjmatrix.begin(), adjmatrix.end(),
point), adjmatrix.end());
	}
	else
	{
		cout << "Unable to locate edge." << endl;
	}
	sort(adjmatrix.begin(), adjmatrix.end());
}

template<class T>
void AdjacencyMatrix<T>::BFS(T v)
{
	const int V = 10;
	T first;
	T second;
	vector<int> adj[10];
	for (auto i = adjmatrix.begin(); i != adjmatrix.end(); ++i)
	{
		first = *i / 10;
		second = *i % 10;
		adj[first].push_back(second);
	}

	// Mark all the vertices as not visited
	bool *visited = new bool[V];
	for (int i = 0; i < V; i++)
		visited[i] = false;

	// Create a queue for BFS
	list<int> queue;
```

```
	// Mark the current node as visited and enqueue it
	visited[v] = true;
	queue.push_back(v);

	// 'i' will be used to get all adjacent
	// vertices of a vertex
	list<int>::iterator i;

	while (!queue.empty())
	{
		// Dequeue a vertex from queue and print it
		v = queue.front();
		cout << v << " ";
		queue.pop_front();

		// Get all adjacent vertices of the dequeued
		// vertex s. If a adjacent has not been visited,
		// then mark it visited and enqueue it
		for (auto i = adj[v].begin(); i != adj[v].end(); ++i)
		{
			if (!visited[*i])
			{
				visited[*i] = true;
				queue.push_back(*i);
			}
		}
	}
}

template<class T>
void AdjacencyMatrix<T>::DFSUtil(T v, bool visited[])
{
	const int V = 10;
	T first;
	T second;
	vector<int> adj[10];
	for (auto i = adjmatrix.begin(); i != adjmatrix.end(); ++i)
	{
		first = *i / 10;
		second = *i % 10;
		adj[first].push_back(second);
	}

	// Mark the current node as visited and
	// print it
	visited[v] = true;
	cout << v << " ";

	// Recur for all the vertices adjacent
	// to this vertex
	list<int>::iterator i;
	for (auto i = adj[v].begin(); i != adj[v].end(); ++i)
		if (!visited[*i])
			DFSUtil(*i, visited);
```

```
}

template<class T>
void AdjacencyMatrix<T>::DFS(T v)
{
	// Mark all the vertices as not visited
	bool *visited = new bool[10];
	for (auto i = 0; i < 10; i++)
		visited[i] = false;

	// Call the recursive helper function
	// to print DFS traversal
	DFSUtil(v, visited);
}

template<class T>
std::list<T> AdjacencyList<T>::getGraph() const
{
	return adjlist;
}

template<class T>
void AdjacencyList<T>::inputGraph(string s)
{
	string input;
	T array[20];

	infile.open(s);
	while (std::getline(infile, input))
	{
		T x;
		replace(input.begin(), input.end(), ',', ' ');
		replace(input.begin(), input.end(), ':', ' ');
		input.erase(std::remove(input.begin(), input.end(), ' '),
input.end());
		for (auto i = 0; i < input.length(); ++i)
		{
			array[i] = input[i] - '0';
		}
		x = array[0];
		for (auto j = 0; j < input.length(); ++j)
		{
			addEdge(x, array[j]);
		}
	}
	infile.close();
}

template<class T>
void AdjacencyList<T>::printGraph() const
{
	cout << endl;

	for (auto i = adjlist.begin(); i != adjlist.end(); ++i)
```

```
	{
		std::cout << (*i / 10) << "->" << (*i % 10) << '\n';
	}
	cout << endl;
}

template<class T>
void AdjacencyList<T>::printGraph(vector<T> x) const
{
	for (auto i = x.begin(); i != x.end(); ++i)
	{
		std::cout << (*i/10) << "->" << (*i%10) << '\n';
	}
	cout << endl;
}

template<class T>
bool AdjacencyList<T>::adjacent(T x, T y)
{
	T point = (x * 10) + y;
	std::list<int>::iterator it;
	it = find(adjlist.begin(), adjlist.end(), point);
	if (it != adjlist.end())
	{
		return true;
	}
	else
	{
		return false;
	}
}

template<class T>
std::vector<T> AdjacencyList<T>::neighbors(T x)
{
	std::vector<T> returnVector;
	std::list<T> currentList = getGraph();
	T singleDigit = NULL;
	T nodeId = (x * 10);
	T max = nodeId + 9;
	for (nodeId; nodeId <= max; ++nodeId)
	{
		for (auto i = currentList.begin(); i != currentList.end(); ++i)
		{
			if (*i == nodeId)
			{
				returnVector.push_back(*i);
			}
			else
			{
				void;
			}
		}
	}
```

```
	cout << endl;

	return returnVector;
}

template<class T>
void AdjacencyList<T>::addNode(T x)
{
	T point = x + (x * 10);		// this allows for the user to combine
the points into a single value to be stored
	adjlist.erase(std::remove(adjlist.begin(), adjlist.end(), point),
adjlist.end());
	adjlist.push_back(point);
	adjlist.sort();
}

template<class T>
void AdjacencyList<T>::deleteNode(T x)
{
	T point = (x * 10);
	for (T i = 0; i <= 9; ++i)
	{
		adjlist.erase(std::remove(adjlist.begin(), adjlist.end(), point +
i), adjlist.end());
		for (T i = 0; i <= 9; ++i)
		{
			adjlist.erase(std::remove(adjlist.begin(), adjlist.end(),
((i*10)+(point/10))), adjlist.end());
		}
	}
	adjlist.sort();
}

template<class T>
void AdjacencyList<T>::addEdge(T x, T y)
{
	T point = (x * 10) + y;
	adjlist.erase(std::remove(adjlist.begin(), adjlist.end(), point),
adjlist.end());   //deleting edge if it exists and re-adding just for ease
	adjlist.push_back(point);
	adjlist.sort();
}

template<class T>
void AdjacencyList<T>::deleteEdge(T x, T y)
{
	T point = (x * 10) + y;
	std::list<int>::iterator it;
	it = find(adjlist.begin(), adjlist.end(), point);
	if (it != adjlist.end())
	{
		cout << "Edge found, deleteing edge." << endl;
		adjlist.erase(std::remove(adjlist.begin(), adjlist.end(), point),
adjlist.end());
```

```
	}
	else
	{
		cout << "Unable to locate edge." << endl;
	}
	adjlist.sort();
}

template<class T>
void AdjacencyList<T>::BFS(T v)
{
	const int V = 10;
	T first;
	T second;
	vector<int> adj[10];
	for (auto i = adjlist.begin(); i != adjlist.end(); ++i)
	{
		first = *i / 10;
		second = *i % 10;
		adj[first].push_back(second);
	}

	// Mark all the vertices as not visited
	bool *visited = new bool[V];
	for (int i = 0; i < V; i++)
		visited[i] = false;

	// Create a queue for BFS
	list<int> queue;

	// Mark the current node as visited and enqueue it
	visited[v] = true;
	queue.push_back(v);

	// 'i' will be used to get all adjacent
	// vertices of a vertex
	list<int>::iterator i;

	while (!queue.empty())
	{
		// Dequeue a vertex from queue and print it
		v = queue.front();
		cout << v << " ";
		queue.pop_front();

		// Get all adjacent vertices of the dequeued
		// vertex s. If a adjacent has not been visited,
		// then mark it visited and enqueue it
		for (auto i = adj[v].begin(); i != adj[v].end(); ++i)
		{
			if (!visited[*i])
			{
				visited[*i] = true;
				queue.push_back(*i);
```

```
                }
            }
      }
}

template<class T>
void AdjacencyList<T>::DFSUtil(T v, bool visited[])
{
      const int V = 10;
      T first;
      T second;
      vector<int> adj[10];
      for (auto i = adjlist.begin(); i != adjlist.end(); ++i)
      {
            first = *i / 10;
            second = *i % 10;
            adj[first].push_back(second);
      }

      // Mark the current node as visited and
      // print it
      visited[v] = true;
      cout << v << " ";

      // Recur for all the vertices adjacent
      // to this vertex
      list<int>::iterator i;
      for (auto i = adj[v].begin(); i != adj[v].end(); ++i)
            if (!visited[*i])
                  DFSUtil(*i, visited);
}

template<class T>
void AdjacencyList<T>::DFS(T v)
{
      // Mark all the vertices as not visited
      bool *visited = new bool[10];
      for (auto i = 0; i < 10; i++)
            visited[i] = false;

      // Call the recursive helper function
      // to print DFS traversal
      DFSUtil(v, visited);
}


int main()
{
      AdjacencyMatrix<int> matrix;
      AdjacencyList<int> list;

      int inputX;
      int inputY;
```

```
	string fileName = "text.txt";

	cout << "Note to testers: only accepts single digit values.\n\n";

	cout << "Input File and Print for Matrix\n";

	matrix.inputGraph(fileName);
	matrix.printGraph();

	cout << "Input File and Print for List\n";

	list.inputGraph(fileName);
	list.printGraph();

	cout << "Add Node to Matrix(enter 0 to exit): \n";
	cin >> inputX;
	while (inputX != 0) {
		matrix.addNode(inputX);
		cout << "\nEnter another node to enter or enter 0 to exit\n";
		cin >> inputX;
	}

	matrix.printGraph();

	cout << "Delete Node from Matrix(enter 0 to exit): \n";
	cin >> inputX;
	while (inputX != 0) {
		matrix.deleteNode(inputX);
		cout << "\nEnter another node to delete or enter 0 to exit\n";
		cin >> inputX;
	}

	matrix.printGraph();

	cout << "Add Edge to Matrix(enter 0 to exit): x,y with values separated
by space\n";
	cin >> inputX >> inputY;
	while (inputX != 0) {
		matrix.addNode(inputX);	// by my theory if there's a node,
then it relates to itself as well
		matrix.addEdge(inputX, inputY);
		cout << "\nEnter another edge to enter or enter 0 0 to exit\n";
		cin >> inputX >> inputY;
	}

	matrix.printGraph();

	cout << "Delete Edge from Matrix(enter 0 to exit): x,y with values
separated by space\n";
	cin >> inputX >> inputY;
	while (inputX != 0) {
		matrix.deleteEdge(inputX, inputY);
		cout << "\nEnter another edge to delete or enter 0 0 to exit\n";
		cin >> inputX >> inputY;
```

```
	}

	matrix.printGraph();

	cout << "What Node would you like to know the neighbors of? (enter node
or 0 to exit)\n";
	cin >> inputX;
	while (inputX != 0) {
		matrix.printGraph(matrix.neighbors(inputX));
		cout << "\nEnter another node to find out neighbors or enter 0 to
exit\n";
		cin >> inputX;
	}

	cout << "Enter nodes to determine whether or not they are adjacent
(enter x,y values seperated by a space or 0 0 to exit)\n";
	cin >> inputX >> inputY;
	while (inputX != 0) {
		matrix.adjacent(inputX, inputY);
		cout << "\nEnter another edge to determine whether it has
adjacency (enter x,y values separated by a space or 0 0 to exit)\n";
		cin >> inputX >> inputY;
	}

	cout << "What BFS vertex would you like to know the nodes of? (enter
node or 0 to exit)\n";
	cin >> inputX;
	while (inputX != 0) {
		matrix.BFS(inputX);
		cout << "\nEnter another node to find BFS of or enter 0 to
exit\n";
		cin >> inputX;
	}

	cout << "What DFS vertex would you like to know the nodes of? (enter
node or 0 to exit)\n";
	cin >> inputX;
	while (inputX != 0) {
		matrix.DFS(inputX);
		cout << "\nEnter another node to find BFS of or enter 0 to
exit\n";
		cin >> inputX;
	}

	cout << "\n\nSince my functions are virtually the same for either, I've
only displayed the Matrix\n";

	system("PAUSE");
}
```

CODE RUNNING

```
Note to testers: only accepts single digit values.

Input File and Print for Matrix

1->1
1->2
1->3
1->4
2->2
2->4
2->5
3->1
3->3
3->5
4->1
4->4

Input File and Print for List

1->1
1->2
1->3
1->4
2->2
2->4
2->5
3->1
3->3
3->5
4->1
4->4

Add Node to Matrix(enter 0 to exit):
5

Enter another node to enter or enter 0 to exit
6

Enter another node to enter or enter 0 to exit
0

1->1
1->2
1->3
1->4
2->2
2->4
2->5
3->1
3->3
3->5
4->1
```

```
4->4
5->5
6->6

Delete Node from Matrix(enter 0 to exit):
6

Enter another node to delete or enter 0 to exit
5

Enter another node to delete or enter 0 to exit
0

1->1
1->2
1->3
1->4
2->2
2->4
3->1
3->3
4->1
4->4

Add Edge to Matrix(enter 0 to exit): x,y with values separated by space
5 3

Enter another edge to enter or enter 0 0 to exit
0 0

1->1
1->2
1->3
1->4
2->2
2->4
3->1
3->3
4->1
4->4
5->3
5->5

Delete Edge from Matrix(enter 0 to exit): x,y with values separated by space
5 5
Edge found, deleteing edge.

Enter another edge to delete or enter 0 0 to exit
0 0

1->1
1->2
1->3
1->4
```

```
2->2
2->4
3->1
3->3
4->1
4->4
5->3

What Node would you like to know the neighbors of? (enter node or 0 to exit)
1

1->1
1->2
1->3
1->4


Enter another node to find out neighbors or enter 0 to exit
0
Enter nodes to determine whether or not they are adjacent (enter x,y values
seperated by a space or 0 0 to exit)
1 2

Enter another edge to determine whether it has adjacency (enter x,y values
separated by a space or 0 0 to exit)
0 0
What BFS vertex would you like to know the nodes of? (enter node or 0 to
exit)
1
1 2 3 4
Enter another node to find BFS of or enter 0 to exit
0
What DFS vertex would you like to know the nodes of? (enter node or 0 to
exit)
2
2 4 1 3
Enter another node to find BFS of or enter 0 to exit
0


Since my functions are virtually the same for either, I've only displayed the
Matrix
Press any key to continue . . .
```