```
#include <iostream>
#include <fstream>

#include <ctime>
#include <ratio>
#include <chrono>
#include <time.h>
#include <stdlib.h>

#include <string>
#include <cstring> // for string tokenizer and c-style string processing

using namespace std;

// implementing the dynamic List ADT using Linked List

class Node{

        private:
                int data;
                Node* nextNodePtr;

        public:
                Node(){}

                void setData(int d){
                        data = d;
                }

                int getData(){
                        return data;
                }

                void setNextNodePtr(Node* nodePtr){
                                nextNodePtr = nodePtr;
                }

                Node* getNextNodePtr(){
                        return nextNodePtr;
                }

};

class List{

        private:
                Node *headPtr;

        public:
```

```
List(){
        headPtr = new Node();
        headPtr->setNextNodePtr(0);
}

Node* getHeadPtr(){
        return headPtr;
}

bool isEmpty(){

        if (headPtr->getNextNodePtr() == 0)
                return true;

        return false;
}


void insert(int data){

        Node* currentNodePtr = headPtr->getNextNodePtr();
        Node* prevNodePtr = headPtr;

        while (currentNodePtr != 0){
                prevNodePtr = currentNodePtr;
                currentNodePtr = currentNodePtr->getNextNodePtr();
        }

        Node* newNodePtr = new Node();
        newNodePtr->setData(data);
        newNodePtr->setNextNodePtr(0);
        prevNodePtr->setNextNodePtr(newNodePtr);


}

void insertAtIndex(int insertIndex, int data){

        Node* currentNodePtr = headPtr->getNextNodePtr();
        Node* prevNodePtr = headPtr;

        int index = 0;

        while (currentNodePtr != 0){

                if (index == insertIndex)
                        break;

                prevNodePtr = currentNodePtr;
```

```
                currentNodePtr = currentNodePtr->getNextNodePtr();
                index++;
        }

        Node* newNodePtr = new Node();
        newNodePtr->setData(data);
        newNodePtr->setNextNodePtr(currentNodePtr);
        prevNodePtr->setNextNodePtr(newNodePtr);

}


int read(int readIndex){

        Node* currentNodePtr = headPtr->getNextNodePtr();
        Node* prevNodePtr = headPtr;
        int index = 0;

        while (currentNodePtr != 0){

                if (index == readIndex)
                        return currentNodePtr->getData();

                prevNodePtr = currentNodePtr;
                currentNodePtr = currentNodePtr->getNextNodePtr();

                index++;

        }

        return -1; // an invalid value indicating
                                // index is out of range

}

void modifyElement(int modifyIndex, int data){

        Node* currentNodePtr = headPtr->getNextNodePtr();
        Node* prevNodePtr = headPtr;
        int index = 0;

        while (currentNodePtr != 0){

                if (index == modifyIndex){
                        currentNodePtr->setData(data);
                        return;
                }

                prevNodePtr = currentNodePtr;
```

```
                currentNodePtr = currentNodePtr->getNextNodePtr();

                index++;
            }


        }


        void deleteElementAtIndex(int deleteIndex){

            Node* currentNodePtr = headPtr->getNextNodePtr();
            Node* prevNodePtr = headPtr;
            Node* nextNodePtr = headPtr;
            int index = 0;

            while (currentNodePtr != 0){

                if (index == deleteIndex){
                    nextNodePtr =
currentNodePtr->getNextNodePtr();
                    break;
                }

                prevNodePtr = currentNodePtr;
                currentNodePtr = currentNodePtr->getNextNodePtr();


                index++;
            }

            prevNodePtr->setNextNodePtr(nextNodePtr);

        }


        void deleteElement(int deleteData){

            Node* currentNodePtr = headPtr->getNextNodePtr();
            Node* prevNodePtr = headPtr;
            Node* nextNodePtr = headPtr;

            while (currentNodePtr != 0){

                if (currentNodePtr->getData() == deleteData){
                    nextNodePtr =
currentNodePtr->getNextNodePtr();
                    break;
                }
```

```
                        prevNodePtr = currentNodePtr;
                        currentNodePtr = currentNodePtr->getNextNodePtr();


                }

                prevNodePtr->setNextNodePtr(nextNodePtr);

        }


        void IterativePrint(){

                Node* currentNodePtr = headPtr->getNextNodePtr();

                while (currentNodePtr != 0){
                        cout << currentNodePtr->getData() << " ";
                        currentNodePtr = currentNodePtr->getNextNodePtr();
                }

                cout << endl;

        }



    // add any required member function here



};



class Graph{

    private:
        int numNodes;
        List* adjacencyList;

    public:

        Graph(int n){
                numNodes = n;
                adjacencyList = new List[numNodes];
        }
```

```
        void addEdge(int u, int v){

                adjacencyList[u].insert(v);
                adjacencyList[v].insert(u);

        }


        List getNeighborList(int id){
                return adjacencyList[id];
        }


        // add any required member function here


};


int main(){

        string graphEdgesFilename;
        cout << "Enter the file name for the edges of the graph: ";
        cin >> graphEdgesFilename;

        int numNodes;
        cout << "Enter number of nodes: ";
        cin >> numNodes;



        Graph graph(numNodes);

        ifstream graphEdgeFileReader(graphEdgesFilename);

        if (!graphEdgeFileReader){
                cout << "File cannot be opened!! ";
                return 0;
        }

        int numCharsPerLine = 25;

        char *line = new char[numCharsPerLine];
        // '25' is the maximum number of characters per line

        graphEdgeFileReader.getline(line, numCharsPerLine, '\n');
        // '\n' is the delimiting character to stop reading the line
```

```
        while (graphEdgeFileReader){

                char* cptr = strtok(line, " ");

                string uNodeToken(cptr);
                int uNodeID = stoi(uNodeToken);

                cptr = strtok(NULL, " ");

                string vNodeToken(cptr);
                int vNodeID = stoi(vNodeToken);

                graph.addEdge(uNodeID, vNodeID);


                graphEdgeFileReader.getline(line, numCharsPerLine, '\n');

        }


        cout << endl;


        // add code here to find and print the probabilities of finding vertices
with certain degree
        // and compute the average degree based on these probabilities


return 0;
}
```