```
#include <iostream>
#include <fstream>
#include <iomanip>
#include <cstdlib>
#include <vector>
#include <string>
#include <algorithm>
#include <utility>
using namespace std;



class Node{
public:
	Node(int n,int lvl): level(lvl){
		number = n;
		totalNodesCreated++;
	}
	//Copy constructor
	Node(Node& copyConstruction){
		this->index = copyConstruction.index;
		this->level = copyConstruction.level;
		this->number = copyConstruction.number;
		this->pAdjacent = copyConstruction.pAdjacent;
		this->totalNodesCreated = Node::totalNodesCreated;
	}
	int number; //Node number
	int level;
	int index;  //Identifies different instances of the same node
	static int totalNodesCreated;
	vector<class Node *> pAdjacent; //Pointers to adjacent nodes

	//We must define exactly what we mean when we say one node comes before
the other.
	//This is essential because we want to sort the nodes.
	friend bool operator>(Node& nodePtr0, Node& nodePtr){
		if(nodePtr0.level > nodePtr.level)
			return true;
		else
			return false;
	}

	~Node(){
		totalNodesCreated--;
	}
};//Node;

class Tree{
public:
	//Our network representation
	vector<vector<int>> network;

	Node* root;

	ofstream output;  //Pointer to the output file
	string outputFileName;
```

```
	Node* constructTree(vector<int> /*visited*/, int /*curNode*/);
	Tree(vector<vector<int>> net, int n, string fileName);

	void listPaths(int destNode);
	void listPathUtility(Node*, vector<pair<int, int>>, int destNode);



	//-------------------------------------------------------------------------
--------------
	//-------------------------------------------------------------------------
--------------
	//We want to index all the node instances. For this purpose we use the
garbage container.
	//This container was created specifically to simplify the process of
freeing memory.
	void attachIndex();
	void myBubbleSort();

	//-------------------------------------------------------------------------
--------------
	//-------------------------------------------------------------------------
--------------

	void printPath(vector<pair<int, int>> path);

	friend ofstream& operator<<(ofstream& output, vector<pair<int, int>>
path){
		for(vector<pair<int, int>>::iterator it = path.begin(); it !=
path.end(); it++)
			output << "(" << setw(5) << it->second << ")" << setw(5) <<
++(it->first) << setw(5) << " ";
		output << endl;
		return output;
	}

	~Tree();

	//Keeps track of the allocated memory. Makes the release of memory
painless.
	//This way, we ensure that there are no memory leaks.
	//Alternatively, We could have provided the Node class with its own
destructor.
	//This, however, would have required the use of recursion to release
the memory.
	//The method we use here is more preferable as it makes it clear that
the Tree instance owns all its Node instances.
	vector<Node*> garbage;


	//Try to optimize the indexing process by introducing a vector of
vectors
	vector<vector<Node*>> optimum;
	//void createOptimizationVector(Node* node);
	void attachIndex(bool optimized);  //To differetiate it from the
standard one difined before
```

```
};


//Initialization of the static variable
int Node::totalNodesCreated = 0;


Tree::Tree(vector<vector<int>> net, int n, string
fileName):output(fileName.c_str()){
	network = net;
	outputFileName = fileName;

	if(output == 0){
		cout << "Could not open output file: " << fileName.c_str();
		exit(0);
	}

	cout << "Constructing the tree." << endl;
	root = constructTree(vector<int>(), n);
	cout << "Done with construction of the tree." << "\n"
		<< "Total number of nodes created: " << garbage.size() << "\n"
		<< "Attaching index values." << endl;
	attachIndex(true);
}



//curNode >= 0
Node* Tree::constructTree(vector<int> visited, int curNode){
	static int tmpLevel = 0;
	//If we find the current node in the list of nodes we have visited we
must return.
	if(find(visited.begin(), visited.end(), curNode) != visited.end())
		return 0;

	//Insert the current node into the list of nodes we have visited.
	Node* tmpNode = new Node(curNode, tmpLevel);
	visited.insert(visited.end(), curNode);

	if( tmpLevel >= optimum.size() ){
		vector<Node*> tmpVec;
		tmpVec.insert(tmpVec.end(), tmpNode);
		optimum.insert(optimum.end(), tmpVec);
	}else{
		(optimum.begin() + tmpLevel)->insert( (optimum.begin() +
tmpLevel)->end(), tmpNode);
	}

	//Control the depth of the tree
	if(tmpLevel == 11){
		//cout << "The end of the path!" << endl;
		return tmpNode;
	}

	//Going down one level
	tmpLevel++;
```

```
	//Records all the memmory that has been allocated
	garbage.insert(garbage.end(), tmpNode);

	//Optimize indexing by introducing a vector
//	this->createOptimizationVector(tmpNode);


	//Move to the correct row
	vector<int> tmpRow( (network.begin() + curNode)->begin(),
(network.begin() + curNode)->end());

	for(int nodeCounter = 0; nodeCounter < network.begin()->size();
nodeCounter++){
		if(nodeCounter == curNode || tmpRow.at(nodeCounter) == 0)
			continue;
		Node* tmpNode2 = constructTree(visited, nodeCounter);

		if(tmpNode2 != 0) //Add the Node pointer only if it is not NULL
			tmpNode->pAdjacent.insert(tmpNode->pAdjacent.end(),
tmpNode2);
	}

	//Going back up one level
	tmpLevel--;

	return tmpNode;
}

Tree::~Tree(){

	for(int nodeCounter = 0; nodeCounter < garbage.size(); nodeCounter++){
		delete garbage.at(nodeCounter);
		garbage.at(nodeCounter) = 0;
		Node::totalNodesCreated--;
	}
}

void Tree::listPaths(int destNode){

	listPathUtility(root, vector<pair<int, int>>(), destNode);
	output.close();

	cout << "The output file should appear in a shortwhile..." << endl;

	system(outputFileName.c_str());	//Opens the file containing the
output once done.
}

void Tree::listPathUtility(Node* curNode, vector<pair<int, int>> path, int
destNode){
	path.insert(path.end(), make_pair(curNode->number, curNode->index));
	if(destNode >= 0 && destNode == curNode->number){
		output << path;
		//printPath(path);
		return;
	}else if(curNode->pAdjacent.size() == 0 && destNode < 0){
```

```
                  output << path;
                  //printPath(path);
                  return;
            }

      for(int nodeCounter = 0; nodeCounter < curNode->pAdjacent.size();
nodeCounter++){
            listPathUtility(curNode->pAdjacent[nodeCounter], path, destNode);
      }

}


void Tree::attachIndex(){
      //We sort the garbage container by the level data member.
      myBubbleSort();

      //Attach the index
      for(int index = 0; index < garbage.size(); index++)
            garbage[index]->index = index;

      return;
}


void Tree::myBubbleSort(){

      for(int count = 0; count < garbage.size(); count++){
            for(int index = (garbage.size() - 1); index > count; index--){
                  Node tmpNode0 = *garbage[index];
                  Node tmpNode = *garbage[index-1];
                  //This uses the operator> defined in(for) the Node data
structure.
                  if( tmpNode > tmpNode0 ){
                        Node* tmpNode = garbage[index];
                        garbage[index] = garbage[index-1];
                        garbage[index-1] = tmpNode;
                  }
            }
      }
      return;
}


void Tree::attachIndex( bool ){
      vector<vector<Node*>>::iterator it = optimum.begin();
      int index = 0;
      for( ; it != optimum.end(); it++){
            for(vector<Node*>::iterator it0 = it->begin(); it0 != it->end();
it0++){
                  (*it0)->index = index;
                  index++;
            }
      }
      return;
}
```

```
int main(){

	const int LENGTH = 33;
	int tmpMat[LENGTH][LENGTH];
	{
	for(int countR = 0; countR < LENGTH; countR++)
		for(int countC = 0; (countC) < LENGTH; countC++)
			tmpMat[countR][countC] = 0;
	}


	//Raw input in the form of a node followed by its neighbours
	string fileName;
	cout << "Enter the input file: ";
	cin >> fileName;
	ifstream input0(fileName.c_str());
	if(input0 == 0){
		cout << "Could not open the requisite file...." << fileName <<
endl;
		exit(0);
	}

	int row = -1;
	int col = -1;

	for(int index = LENGTH; index > 0; index--){
		input0 >> row;
		row-=1;

		input0 >> col;
		col-=1;
		for( ;col >= 0; ){
			tmpMat[row][col] = 1;
			input0 >> col;
			col-=1;
		}
		input0.get();	//Throw away the newline character terminating
the input line
	}
	input0.close();


	for(int countR = 0; countR < LENGTH; countR++){
		for(int countC = 0; (countC) < LENGTH; countC++)
			cout << tmpMat[countR][countC];
		cout << endl;
	}


	//Convert the array representation of the network as vector.
	vector<vector<int>> net;
	vector<int> tmpVec;


	for(int rows = 0; rows < LENGTH; rows++){
```

```
            for(int cols = 0; cols < LENGTH; cols++)
                  tmpVec.insert(tmpVec.end(), tmpMat[rows][cols]);
            net.insert(net.end(), tmpVec);
            tmpVec.erase(tmpVec.begin(), tmpVec.end());
      }

      //string fileName;
      cout << "Enter the name of the output file: ";
      cin >> fileName;  //Reusing the variable

      int sourceNode;
      cout << "Enter the source node: ";
      cin >> sourceNode;

      Tree T(net, --sourceNode, fileName);

      cout << "Enter the destination node: ";
      cin >> sourceNode;      //Reusing the variable
      T.listPaths(--sourceNode);

      return 0;
}
```