```
#include <iostream>
#include <vector>
#include <string>
#include <utility>
#include <limits>

using namespace std;

const int MAXSIZE = 20;

template<class type>
class uGraph
{
private:
#define noEdge 0
        int adjacency[MAXSIZE][MAXSIZE];
        vector<type> nodes;
        enum state = { unknown, visted, complete };

        void recur_dfs(int u, state states[], std::function<void(const type &)> process)
        {
                states[u] = visted;
                process(nodes[u]);
                for (int v = 0; v < nodes.size(); v++)
                {
                        if (adjacent(nodes[u], nodes[v]) && states[v] == unknown)
                        {
                                recur_dfs(v, states);
                        }
                }
                states[u] = complete;
        }

public:
        uGraph()
        {
                type t;
                for (int i = 0; i < MAXSIZE; i++)
                {
                        for (int j = 0; j < MAXSIZE; j++)
                        {
                                adjacency[i][j] = noEdge;
                        }
                        nodes.push_back(t);
                }
        }
        bool adjacent(type x, type y)
        {
                int m = -1;
                int n = -1;
                for (int i = 0; i < nodes.size(); i++)
                {
                        if (nodes[i] == x)
                        {
                                m = i;
                        }
                        else if (nodes[i] == y)
                        {
                                n = i;
                        }
```

```
		}
		if (m >= 0 && n >= 0)
		{
			if (adjacency[m][n] != noEdge && m != n)
			{
				return true;
			}
		}
		return false;
	}
	vector<type> neighbors(type x)
	{
		vector<type> set;
		int m = -1;
		for (int i = 0; i < nodes.size(); i++)
		{
			if (nodes[i] == x)
			{
				m = i;
			}
		}
		if (m >= 0)
		{
			for (int j = 0; j < nodes.size(); j++)
			{
				if (adjacency[j][m] != noEdge && j != m)
				{
					set.push_back(nodes[j]);
				}
			}
		}
		return set;
	}
	void addNode(type x)
	{
		for (int i = 0; i < MAXSIZE; i++)
		{
			if (adjacency[i][i] == noEdge)
			{
				nodes[i] = x;
				adjacency[i][i] = INT_MAX;
				return;
			}
		}
		return;
	}
	void deleteNode(type x)
	{
		for (int i = 0; i < nodes.size(); i++)
		{
			if (nodes[i] == x)
			{
				for (int j = 0; j < MAXSIZE; j++)
				{
					adjacency[i][j] = noEdge;
					adjacency[j][i] = noEdge;
				}
				return;
			}
		}
	}
```

```
	void addEdge(type x, type y)
	{
		int m = -1;
		int n = -1;
		for (int i = 0; i < nodes.size(); i++)
		{
			if (nodes[i] == x)
			{
				m = i;
			}
			else if (nodes[i] == y)
			{
				n = i;
			}
		}
		if (m >= 0 && n >= 0)
		{
			adjacency[m][n] = 1;
			adjacency[n][m] = 1;
		}
		return;
	}

	void addEdge(type x, type y, int weight)
	{
		int m = -1;
		int n = -1;
		for (int i = 0; i < nodes.size(); i++)
		{
			if (nodes[i] == x)
			{
				m = i;
			}
			else if (nodes[i] == y)
			{
				n = i;
			}
		}
		if (m >= 0 && n >= 0)
		{
			adjacency[m][n] = weight;
			adjacency[n][m] = weight;
		}
		return;
	}

	void deleteEdge(type x, type y)
	{
		int m = -1;
		int n = -1;
		for (int i = 0; i < nodes.size(); i++)
		{
			if (nodes[i] == x)
			{
				m = i;
			}
			else if (nodes[i] == y)
			{
				n = i;
			}
		}
```

```
        if (m >= 0 && n >= 0)
        {
                adjacency[m][n] = noEdge;
                adjacency[n][m] = noEdge;
        }
        return;
}

void dfs(std::function<void(const type &)> process)
{
        state states[nodes.size()];
        for (int i = 0; i < nodes.size(); i++)
        {
                states[i] = unknown;
        }
        int u = 0;
        while (nodes[u] != INT_MAX)
        {
                u++;
        }
        recur_dfs(u, state, process);
        return;
}

void bfs(std::function<void(const type &)> process)
{
type u = nodes[0];// starting vertex
vector<type> visted;
vector<type> complete;
vector<pair<type, type>> edges;
visted.push_back(u);


while (visted.size() != 0)
{
        vector<type> set = neighbors(u);
        for (int i = 0; i < set.size(); i++)
        {
                bool notFound = true;
                for (int j = 0; j < visted.size(); j++)
                {
                        if (visted[j] == set[i])
                        {
                                notFound = false;
                        }
                }
                for (int j = 0; j < complete.size(); j++)
                {
                        if (j < complete.size() && complete[j] == set[i])
                        {
                                notFound = false;
                        }
                }

                if (notFound == true)
                {
                        visted.push_back(set[i]);
                }
        }
        for (int i = 0; i < visted.size(); i++)
        {
```

```
                        if (adjacent(u, visted[i]))
                        {
                                edges.push_back(make_pair(u, visted[i]));
                                edges.push_back(make_pair(visted[i], u));//handles undirected
graph
                        }
                }

                for (int i = 0; i < visted.size() - 1; i++)
                {
                        visted[i] = visted[i + 1];
                }
                visted.pop_back();
                complete.push_back(u);
                if (visted.size() > 0)
                {
                        u = visted[0];
                }
                if (true);
        }
        return;
        }
};

template<class type>
class dGraph
{
private:
#define noEdge 0
        int adjacency[MAXSIZE][MAXSIZE];
        vector<type> nodes;
        enum state = { unknown, visted, complete };

        void recur_dfs(int u, state states[], std::function<void(const type &)> process)
        {
                states[u] = visted;
                process(nodes[u]);
                for (int v = 0; v < nodes.size(); v++)
                {
                        if (adjacent(nodes[u], nodes[v]) && states[v] == unknown)
                        {
                                recur_dfs(v, states);
                        }
                }
                states[u] = complete;
        }

public:
        dGraph()
        {
                type t;
                for (int i = 0; i < MAXSIZE; i++)
                {
                        for (int j = 0; j < MAXSIZE; j++)
                        {
                                adjacency[i][j] = noEdge;
                        }
                        nodes.push_back(t);
                }
        }
        bool adjacent(type x, type y)
```

```
	{
		int m = -1;
		int n = -1;
		for (int i = 0; i < nodes.size(); i++)
		{
			if (nodes[i] == x)
			{
				m = i;
			}
			else if (nodes[i] == y)
			{
				n = i;
			}
		}
		if (m >= 0 && n >= 0)
		{
			if (adjacency[m][n] != noEdge)
			{
				return true;
			}
			else if (adjacency[n][m] != noEdge)
			{
				return true;
			}
		}
		return false;
	}
	vector<type> neighbors(type x)
	{
		vector<type> set;
		int m = -1;
		for (int i = 0; i < nodes.size(); i++)
		{
			if (nodes[i] == x)
			{
				m = i;
			}
		}
		if (m >= 0)
		{
			for (int j = 0; j < nodes.size(); j++)
			{
				if (adjacency[j][m] != noEdge && j != m)
				{
					set.push_back(nodes[j]);
				}
				else if (adjacency[m][j] != noEdge && j!= m)
				{
					set.push_back(nodes[j]);
				}
			}
		}
		return set;
	}
	void addNode(type x)
	{
		for (int i = 0; i < MAXSIZE; i++)
		{
			if (adjacency[i][i] == noEdge)
			{
				nodes[i] = x;
```

```
                        adjacency[i][i] = INT_MAX;
                        return;
                }
        }
        return;
}
void deleteNode(type x)
{
        for (int i = 0; i < nodes.size(); i++)
        {
                if (nodes[i] == x)
                {
                        for (int j = 0; j < MAXSIZE; j++)
                        {
                                adjacency[i][j] = noEdge;
                                adjacency[j][i] = noEdge;
                        }
                        return;
                }
        }
}
void addEdge(type x, type y)
{
        int m = -1;
        int n = -1;
        for (int i = 0; i < nodes.size(); i++)
        {
                if (nodes[i] == x)
                {
                        m = i;
                }
                else if (nodes[i] == y)
                {
                        n = i;
                }
        }
        if (m >= 0 && n >= 0)
        {
                adjacency[m][n] = 1;
        }
        return;
}

void addEdge(type x, type y, int weight)
{
        int m = -1;
        int n = -1;
        for (int i = 0; i < nodes.size(); i++)
        {
                if (nodes[i] == x)
                {
                        m = i;
                }
                else if (nodes[i] == y)
                {
                        n = i;
                }
        }
        if (m >= 0 && n >= 0)
        {
                adjacency[m][n] = weight;
```

```
        }
        return;
}

void deleteEdge(type x, type y)
{
        int m = -1;
        int n = -1;
        for (int i = 0; i < nodes.size(); i++)
        {
                if (nodes[i] == x)
                {
                        m = i;
                }
                else if (nodes[i] == y)
                {
                        n = i;
                }
        }
        if (m >= 0 && n >= 0)
        {
                adjacency[m][n] = noEdge;
        }
        return;
}

void dfs(std::function<void(const type &)> process)
{
        state states[nodes.size()];
        for (int i = 0; i < nodes.size(); i++)
        {
                states[i] = unknown;
        }
        int u = 0;
        while (nodes[u] != INT_MAX)
        {
                u++;
        }
        recur_dfs(u, state, process);
        return;
}

void bfs(std::function<void(const type &)> process)
{
        type u = nodes[0];// starting vertex
        vector<type> visted;
        vector<type> complete;
        vector<pair<type, type>> edges;
        visted.push_back(u);


        while (visted.size() != 0)
        {
                vector<type> set = neighbors(u);
                for (int i = 0; i < set.size(); i++)
                {
                        bool notFound = true;
                        for (int j = 0; j < visted.size(); j++)
                        {
                                if (visted[j] == set[i])
                                {
```

```
                                        notFound = false;
                                }
                        }
                        for (int j = 0; j < complete.size(); j++)
                        {
                                if (j < complete.size() && complete[j] == set[i])
                                {
                                        notFound = false;
                                }
                        }

                        if (notFound == true)
                        {
                                visted.push_back(set[i]);
                        }
                }
                for (int i = 0; i < visted.size(); i++)
                {
                        if (adjacent(u, visted[i]))
                        {
                                edges.push_back(make_pair(u, visted[i]));
                        }
                }

                for (int i = 0; i < visted.size() - 1; i++)
                {
                        visted[i] = visted[i + 1];
                }
                visted.pop_back();
                complete.push_back(u);
                if (visted.size() > 0)
                {
                        u = visted[0];
                }
                if (true);
        }
        return;
    }
};


int main()
{


    system("pause");
    return 0;
}
```