#pragma once

#include "lib.h"

///////////////////////////////////// Trees and Graphs ///////////////////////////////////////

class Node {

public:

int id;

vector<Node\*> child;

Node(int name) {

id = name;

}

};

class BNode {

public:

int id;

BNode \*left;

BNode \*right;

BNode(int name) {

id = name;

left = NULL;

right = NULL;

}

};

class Tree {

public:

Node root;

};

void build\_tree(Node \* root) {

(root->child).push\_back(new Node(2));

(root->child).push\_back(new Node(34));

(root->child).push\_back(new Node(50));

(root->child).push\_back(new Node(60));

(root->child).push\_back(new Node(70));

(root->child[0]->child).push\_back(new Node(15));

(root->child[0]->child).push\_back(new Node(20));

(root->child[1]->child).push\_back(new Node(30));

(root->child[2]->child).push\_back(new Node(40));

(root->child[2]->child).push\_back(new Node(100));

(root->child[2]->child).push\_back(new Node(20));

(root->child[0]->child[1]->child).push\_back(new Node(25));

(root->child[0]->child[1]->child).push\_back(new Node(50));

}

void build\_binary\_tree(BNode \* root) {

(root->left) = (new BNode(2));

(root->right) = (new BNode(34));

(root->left->left) = (new BNode(15));

(root->left->right) = (new BNode(20));

(root->right->left) = (new BNode(30));

(root->right->right) = (new BNode(40));

(root->left->left->left) = (new BNode(25));

(root->left->left->right) = (new BNode(50));

}

class Graph {

public:

Graph(int n) : adj\_list\_vector(n) //without knowing the graph connectivity or its size

{

//adj\_list\_vectorr = new vector<Node>[n];

}

~Graph() {

delete[] adj\_list\_vectorr;

//delete[] adj\_list\_mapp;

}

void addEdge(int v, int u);

void addEdge(Node v, Node u);

//virtual void addEdge() = 0;

//virtual ~Graph() {}

//Graph and its representations

//(1) Adjacency Matrix

//Pros: Insert, Remove, Check\_if\_edge\_exist: O(1), is the most optimized for edge search

//Cons: Large Space O(v^2), Adding a vertex: O(V), BFS/DFS become costly as we can’t quickly get all adjacent of a node.

//vector<Node, vector<Node>> adj\_matrix\_vector; //better to do vector<Node, Node>\* adj\_matrix\_vectorr; and new/delete

//unordered\_map<Node, vector<Node>> adj\_matrix\_map;// better to do unordered\_map<Node, Node>\* adj\_matrix\_mapp; and new/delete

//(2) Adjacency List

//Pros: Saves space O(|V|+|E|) . In the worst case, there can be C(V, 2) number of edges in a graph thus consuming O(V^2) space. Adding a vertex is easier.

//Cons: Queries like edge search can be done O(V).

vector< vector<int>> adj\_list\_vector; // Is this same as this?

vector< int>\* adj\_list\_vectorr;

//IDK HOW TO handle this:

map<Node, Node>\* adj\_list\_mapp; //map<Node, vector<Node>> adj\_list\_map;

// summary (both directed or undirected graphs)

// always use adj\_list e.g. traversing, dfs, bfs,... (use vector if node are tagged by numbers, use map (ordered) if labled and not numbered)

// unless ONLY u need to check quickly connectivities between x and y, use adj\_matrix (use unordered\_map)

};

void Graph::addEdge(int v, int u)

{

adj\_list\_vector[v].push\_back(u);

if (u != v)

adj\_list\_vector[u].push\_back(v);

}

void Graph::addEdge(Node v, Node u)

{

//adj\_list\_mapp[v].push\_back(u);

//adj\_list\_mapp[u].push\_back(v);

}

/\*class Int\_Graph : public Graph { //node with integer label

public:

int X;

void addEdge();

};

void Int\_Graph::addEdge(int v, int u, double w)

{

}

void addEdge(Graph\* t)

{

return t->addEdge();

}

\*/

void build\_graph() {

Graph \*G = new Graph(5);

G->addEdge(0, 1);

G->addEdge(1, 2);

G->addEdge(2, 3);

G->addEdge(2, 4);

G->addEdge(4, 4);

//G->my\_function<int>(1,2,3);

/\*

(root->child).push\_back(new Node(2));

(root->child).push\_back(new Node(34));

(root->child).push\_back(new Node(50));

(root->child).push\_back(new Node(60));

(root->child).push\_back(new Node(70));

(root->child[0]->child).push\_back(new Node(15));

(root->child[0]->child).push\_back(new Node(20));

(root->child[1]->child).push\_back(new Node(30));

(root->child[2]->child).push\_back(new Node(40));

(root->child[2]->child).push\_back(new Node(100));

(root->child[2]->child).push\_back(new Node(20));

(root->child[0]->child[1]->child).push\_back(new Node(25));

(root->child[0]->child[1]->child).push\_back(new Node(50));

\*/

}

void print\_graph(Graph const & G, int n) {

for (int i = 0; i < n; ++i) {

cout << i << ": ";

for (int neighbour : G.adj\_list\_vector[i]) {

cout << neighbour << " ";

}

cout << endl;

}

}

void inOrderTraverseB(BNode\* node) {

if (node != NULL) {

inOrderTraverseB(node->left);

cout << node->id << endl;

inOrderTraverseB(node->right);

}

}

//BNode \* root = new BNode(20);

//build\_binary\_tree(root);

//inOrderTraverseB(root);

//build\_graph();

/\*Graph \*G = new Graph(5);

G->addEdge(0, 1);

G->addEdge(1, 2);

G->addEdge(2, 3);

G->addEdge(2, 4);

G->addEdge(4, 4);

print\_graph(\*G,5);

\*/