**// Linked List**

class LinkedList {

**// First index is 1, not 0**

**// Get size**

int size() {

Node cur = head;

int k = 0;

while (cur != null) {

k++;

cur = cur.next;

}

return k;

}

**// Add last node**

void addLast(int value) {

Node node = new Node(value);

if (isEmpty()) {

head = tail = node;

} else {

tail.next = node;

tail = node;

}

}

**// Add first node**

void addFirst(int value) {

Node node = new Node(value);

if (isEmpty()) {

head = tail = node;

} else {

node.next = head;

head = node;

}

}

**// Add node after a specific node**

void addAfterIndex(int value, int index) {

if (index < 0) {

addFirst(value);

return;

}

if (index >= size()) {

addLast(value);

return;

}

int count = 1;

Node cur = head;

while (cur != null && count != index) {

count++;

cur = cur.next;

}

if (cur != null) {

Node node = new Node(value);

node.next = cur.next;

cur.next = node;

}

}

**// Delete last node**

void deleteLast() {

if (isEmpty()) {

System.out.println("Empty list!");

return;

}

if (head.next == null) {

head = null;

}

Node cur = head;

while (cur.next.next != null) {

cur = cur.next;

}

cur.next = null;

tail = cur;

}

**// Delete first node**

void deleteFirst() {

if (isEmpty()) {

System.out.println("Empty list!");

return;

}

if (head.next == null) {

head = null;

}

Node cur = head;

head = cur.next;

}

**// Delete specific node**

void deleteIndex(int index) {

if (index <= 1) {

deleteFirst();

return;

}

if (index >= size()) {

deleteLast();

return;

}

int pos = 1;

Node cur = head;

while (pos < index - 1) {

cur = cur.next;

++pos;

}

cur.next = cur.next.next;

}

**// Get node followed by index (start at 0)**

Node pos(int k) {

int i = 0;

Node p = head;

while (p != null) {

if (i == k) return (p);

i++;

p = p.next;

}

return (null);

}

**void delete(Node q)** {

if (isEmpty() || q == null) return;

if (q == head) {

removeFirst();

return;

}

Node f = head;

while (f != null && f.next != q) f = f.next;

if (f == null) return;

Node q1 = q.next;

f.next = q1;

if (f.next == null) tail = f;

}

**void sort(int k, int h)** {

if (k > h) return;

if (k < 0) k = 0;

int n = size();

if (h > n - 1) h = n - 1;

Node u = pos(k);

Node v = pos(h + 1);

Node pi, pj;

Bat x;

for (pi = u; pi != v; pi = pi.next) {

for (pj = pi.next; pj != v; pj = pj.next) {

if (pj.info.price < pi.info.price) {

x = pi.info;

pi.info = pj.info;

pj.info = x;

}

}

}

}

**void sort()** {

Node pi, pj;

Object x;

for (pi = head; pi != null; pi = pi.next) {

for (pj = pi.next;pj != null;pj = pj.next) {

if (pj.info.value < pi.info.value) {

x = pi.info;

pi.info = pj.info;

pj.info = x;

}

}

}

}

**void reverse()** {

MyList t = new MyList();

Node p = head;

while (p != null) {

t.addFirst(p.info.value);

p = p.next;

}

head = t.head;

tail = t.tail;

}

}

**// Tree**

class BSTree {

**void insert(int x)** {

Node node = new Node(x);

if (isEmpty()) {

root = node;

return;

}

Node cu = root;

Node father = null;

while (cu != null) {

if (cu.value == x) return;

father = cu;

if (cu.value < x) cu = cu.right;

else if (cu.value > x) cu = cu.left;

}

assert(father != null);

if (father.value > x) father.left = node;

else father.right = node;

}

**void preOrder(Node p)** {

if (p == null) return;

visit(p);

preOrder(p.left);

preOrder(p.right);

}

**void postOrder(Node p)** {

if (p == null) return;

postOrder(p.left);

postOrder(p.right);

visit(p);

}

**void inOrder(Node p)** {

if (p == null) return;

inOrder(p.left);

visit(p);

inOrder(p.right);

}

**void BreathFirstOrder(Node p)** throws Exception {

Queue queue = new Queue();

if (isEmpty()) return;

queue.enqueue(root);

while (!queue.isEmpty()) {

Node node = queue.dedueue();

if (node.left != null) queue.enqueue((node.left));

if (node.right != null) queue.enqueue((node.right));

visit(node);

}

}

**Node getParent(Node p)** {

if (p == root) return null;

Node father = null, cu = root;

while (cu != null && cu.info.value != p.info.value) {

father = cu;

if (cu.info.value < p.info.value) cu = cu.right; else cu = cu.left;

}

if (cu == null) return null;

return father;

}

**Node findNode(int key)** {

Node cu = root;

while (cu != null) {

if (cu.info.value == key) return cu;

cu = cu.info.value < key ? cu.right : cu.left;

}

return null;

}

**void deleteByCopyL(Node p)** {

if (p == null) return;

if (p.left == null) return;

if (p.left.right == null) {

p.value = p.left.value;

p.left = p.left.left;

} else {

Node father = p.left;

while (father.right.right != null) father = father.right;

p.value = father.right.value;

if (father.right.left == null) {

father.right = null;

} else father.right = father.right.left;

}

}

**void deleteByCopy(Node x)** {

Node f, p;

f = null;

p = root;

while (p != null) {

if (p.value == x.value) break;

f = p;

if (x.info.value < p.info.value) p = p.left; else p = p.right;

}

if (p == null) return; // not found

if (p.left == null && p.right == null) { // p is a leaf node

if (f == null) { // p is root

root = null;

return;

}

if (p == f.left) f.left = null; else f.right = null;

}

if (p.left != null && p.right == null) { // p has left son only

if (f == null) { // p is root

root = p.left;

return;

}

if (p == f.left) f.left = p.left; else f.right = p.left;

}

if (p.left == null && p.right != null) { // p has right son only

if (f == null) { // p is root

root = p.right;

return;

}

if (p == f.left) f.left = p.right; else f.right = p.right;

}

if (p.left != null && p.right != null) { // p has both 2 sons

Node q = p.left;

// find the right-most node in the left sub-tree

Node frp, rp;

frp = null;

rp = q;

while (rp.right != null) {

frp = rp;

rp = rp.right;

}

p.info = rp.info;

if (frp == null) p.left = q.left; else frp.right = rp.left;

}

}

**void deleteByMerging(Node p)** {

Node father = getParent(p);

if (father == null) {

if (p.value != root.value) {

return;

}

if (root.left == null) {

root = root.right;

return;

}

if (root.left.right == null) {

root.left.right = root.right;

root = root.left;

return;

}

Node q = root.left;

while (q.right != null) {

q = q.right;

}

q.right = p.right;

root = p.left;

return;

}

if (p.left == null) {

if (p.value < father.value) father.left = p.right;

else father.right = p.right;

return;

}

Node q = p.left;

while (q.right != null) q = q.right;

q.right = p.right;

if (p.value < father.value) father.left = p.left;

else father.right = p.left;

}

**void rotateRight(Node p)** {

if (p == null || p.left == null) return;

Node c = p.left;

p.left = c.right;

c.right = p;

Node father = getParent(p);

if (father == null) root = c;

else {

if (father.info.value > p.info.value) father.left = c;

else father.right = c;

}

}

**void rotateLeft(Node p)** {

if (p == null || p.right == null) return;

Node c = p.right;

p.right = c.left;

c.left = p;

Node father = getParent(p);

if (father == null) root = c;

else {

if (father.info.value > p.info.value) father.left = c;

else father.right = c;

}

}

**int height(Node p)** {

if (p == null) {

return 0;

}

int leftHeight, rightHeight, height;

leftHeight = height(p.left);

rightHeight = height(p.right);

height = leftHeight > rightHeight ? leftHeight : rightHeight;

return height + 1;

}

**void getHeap(int[] a)** {

int k, tmp;

for (int i = 0; i < a.length; i++) {

if (i != 0) {

k = i;

while ((k - 1) / 2 >= 0) {

if (a[k] >= a[(k - 1) / 2]) {

tmp = a[k];

a[k] = a[(k - 1) / 2];

a[(k - 1) / 2] = tmp;

k = (k - 1) / 2;

if (k == 0) break;

}

else break;

}

}

}

**// Print heap**

for (int i = 0; i < a.length; i++) {

System.out.print(a[i] + " ");

}

}

/\* **Returns true if a binary tree is a binary search tree** \*/

int isBST(Node node)

{

if (node == null)

return 1;

/\* false if the max of the left is > than us \*/

if (node.left != null && maxValue(node.left) >= node.data)

return 0;

/\* false if the min of the right is <= than us \*/

if (node.right != null && minValue(node.right) <= node.data)

return 0;

/\* false if, recursively, the left or right is not a BST \*/

if (!isBST(node.left) || !isBST(node.right))

return 0;

/\* passing all that, it's a BST \*/

return 1;

}

}

**class Graph {**

// All search function

void breadthFirst(char x) {

int k = (int) (x - 65);

Queue my = new Queue();

boolean[] b = new boolean[n];

Arrays.fill(b, true);

my.enqueue(k);

b[k] = false;

while (!my.isEmpty()) {

int p = my.dequeue();

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

if (b[i] && a[i][p] != 0) {

b[i] = false;

my.enqueue(i);

}

}

visit(p);

}

}

void breadthFirst(int k) {

breadthFirst((char) (k + 65));

}

void depthFirst(boolean[] b, char x) {

int k = (int) (x - 65);

visit(k);

b[k] = false;

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

if (b[i] && a[i][k] != 0) {

depthFirst(b, (char) (i + 65));

}

}

}

void depthFirst(char x) {

int k = (int) (x - 65);

boolean[] b = new boolean[n];

Arrays.fill(b, true);

b[k] = false;

depthFirst(b, x);

}

void depthFirst(int x) {

depthFirst((char) (x + 65));

}

// Dijkstra algorithm

int degree(int i) { // Bac cua 1 dinh nao do

int s, j;

s = 0;

for (j = 0; j < n; j++) s += a[i][j];

s += a[i][i];

return s;

}

void dijkstra(int fro, int to, RandomAccessFile f) throws Exception {

int INF = 999; // infinity value

boolean[] S = new boolean[n]; // kiem tra xem da duyet dinh chua

int[] d = new int[n]; // luu gia tri duong di ngan nhat tai dinh do

int[] p = new int[n]; // luu gia tri dinh gan no nhat

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

S[i] = false;

d[i] = a[fro][i];

p[i] = fro;

}

ArrayList<Integer> ss = new ArrayList<>(); // cac dinh duoc lay

S[fro] = true; // da duyet dinh fro

ss.add(fro); // them fro vao ss

// Duyet cac dinh, tim min quang duong, them vao tap ss

int k, t;

while (true) {

k = -1;

t = INF;

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

if (S[i] == true) continue;

if (d[i] < t) {

k = i;

t = d[i];

}

}

if (k == -1) return; // no solution

S[k] = true; // Da duyet dinh k

ss.add(k); // Nhet k vao tap ss

if (k == to) break;

// Recalculate d[i]

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

if (S[i] == true) continue;

if (d[i] > d[k] + a[k][i]) {

d[i] = d[k] + a[k][i];

p[i] = k;

}

}

}

// truy nguoc lai cac dinh tu to ve fro

Stack s = new Stack();

int x = to;

while (true) {

s.push(x);

if (x == fro) break;

x = p[x]; // truy nguoc ve dinh lien truoc no

}

ArrayList<Integer> pp = new ArrayList<>(); // Luu tru cac dinh tu fro den to

while (!s.isEmpty()) { // loi tu stack s sang pp

x = s.pop();

pp.add(x);

}

// In ra cac dinh duoc chon lan luot theo thuat toan

f.writeBytes("" + v[ss.get(0)]);

for (int i = 1; i < ss.size(); i++) f.writeBytes(" " + v[ss.get(i)]);

f.writeBytes("\r\n");

// In ra cac dinh tu fro den to

f.writeBytes("" + v[pp.get(0)]);

for (int i = 1; i < pp.size(); i++) f.writeBytes(" " + v[pp.get(i)]);

f.writeBytes("\r\n");

// In ra min quang duong tai cac dinh tu fro den to

f.writeBytes("" + d[pp.get(0)]);

for (int i = 1; i < pp.size(); i++) f.writeBytes(" " + d[pp.get(i)]);

f.writeBytes("\r\n");

}

// All Euler function

boolean isUndirected() {

for (int i = 0; i < n; i++) for (int j = 0; j < n; j++) if (

a[i][j] != a[j][i]

) return false;

return true;

}

boolean isEvenDegree() {

int bac;

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

bac = 0;

for (int j = 0; j < n; j++) bac += a[i][j];

if (bac % 2 == 1) return false;

}

return true;

}

boolean isConnected() {

boolean[] pushed = new boolean[20];

for (int i = 0; i < n; i++) pushed[i] = false;

Stack s = new Stack();

s.push(0);

pushed[0] = true;

int r;

while (!s.isEmpty()) {

r = s.pop();

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

if (i == r) continue;

if (!pushed[i] && a[r][i] > 0) {

s.push(i);

pushed[i] = true;

}

}

}

for (int i = 0; i < n; i++) if (!pushed[i]) return false;

return true;

}

void checkEulerCycle(RandomAccessFile f) throws Exception {

// Check directed

if (isUndirected()) f.writeBytes(

"The graph is undirected.\r\n"

);

else f.writeBytes("The graph is directed.\r\n");

// Check connected

if (isConnected()) f.writeBytes(

"The graph is connected.\r\n"

);

else f.writeBytes("The graph is not connected.\r\n");

// Check even degree

if (isEvenDegree()) f.writeBytes(

"All vertices have even degree.\r\n"

);

else f.writeBytes("The graph has a vertex with odd degree\r\n");

// Check Euler's cycle

if (isUndirected() && isConnected() && isEvenDegree()) f.writeBytes(

"Conditions for Euler's cycle are satisfied.\r\n"

);

else f.writeBytes("Conditions for Euler's cycle are not satisfied.\r\n");

}

boolean hasEulerCycle() {

boolean ok = true;

if (!isUndirected()) {

System.out.println("The graph is directed.\r\n");

ok = false;

}

if (!isConnected()) {

System.out.println("The graph is not connected.\r\n");

ok = false;

}

if (!isEvenDegree()) {

System.out.println("The graph has a vertex with odd degree\r\n");

ok = false;

}

if (!ok) {

System.out.println("Conditions for Euler's cycle are not satisfied.\r\n");

return false;

}

return true;

}

// Chua phan tich ti nao

void EulerCycle(int k, RandomAccessFile f) throws Exception {

if (k >= n) return;

if (!hasEulerCycle()) return;

Stack s = new Stack();

int[][] b = new int[20][20];

int[] eu = new int[20];

int m;

int i, j, r, t;

int[] x = new int[50];

for (i = 0; i < n; i++) for (j = 0; j < n; j++) b[i][j] = a[i][j];

s.push(k); // Dua dinh k vao Stack

m = 0; // Ban dau chu trinh chua co phan tu nao

t = 0;

x[0] = k;

while (!s.isEmpty()) {

r = s.top();

i = 0;

while (i < n && b[r][i] == 0) i++; // Tim i dau tien de b[r][i] != 0

if (i == n) { // r da la dinh co lap, dua r vao chu trinh Euler

eu[m++] = r;

s.pop(); // Lay dinh co lap ra khoi Stack

}

else {

x[++t] = i;

s.push(i);

b[r][i]--; // Loai canh (i,r) khoi do thi

b[i][r]--; // Loai canh (i,r) khoi do thi

}

}

// In chu trinh Euler

for (i = 0; i < m; i++) f.writeBytes(v[eu[i]] + " ");

f.writeBytes("\r\n");

}

}