// A C++ program to check if a given directed graph is Eulerian or not

#include<bits/stdc++.h>

using namespace std;

typedef pair<int,int> pii ;

multimap<pii,string> mp ;

vector<pii> path ;

int sflag , source;

// A class that represents an undirected graph

class Graph

{

int V; // No. of vertices

list<int> \*adj; // A dynamic array of adjacency lists

int \*in;

public:

// Constructor and destructor

Graph(int V);

~Graph() { delete [] adj; delete [] in; }

// function to add an edge to graph

void addEdge(int v, int w) { adj[v].push\_back(w); (in[w])++; }

// Method to check if this graph is Eulerian or not

bool isEulerianCycle();

// Method to check if all non-zero degree vertices are connected

bool isSC();

// Function to do DFS starting from v. Used in isConnected();

void DFSUtil(int v, bool visited[]);

Graph getTranspose();

// extra checking this is euler or not

bool checkFun() ;

};

Graph::Graph(int V)

{

this->V = V;

adj = new list<int>[V];

in = new int[V];

for (int i = 0; i < V; i++)

in[i] = 0;

}

bool Graph::checkFun(){

int cnt0=0 , cnt1= 0 , cntmins1 =0 ;

int fNode , lNode ;

sflag= 0;

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

int outD = adj[i].size() ;

int inD = in[i] ;

int diff = outD - inD ;

if(diff ==0){

cnt0++ ;

}

else if( diff == 1){

cnt1++ ;

fNode = i ;

}

else if( diff == -1){

cntmins1 ++ ;

lNode = i ;

}

else{

return false ;

}

}

if(cnt1==0 && cntmins1 == 0 ){

return true ;

}

else if( cnt1 == 1 && cntmins1== 1){

addEdge(lNode,fNode );

sflag = 1 ;

source = fNode ;

return true ;

}

else{

return false ;

}

}

/\* This function returns true if the directed graph has an eulerian

cycle, otherwise returns false \*/

bool Graph::isEulerianCycle()

{

// check is graph is euler or not

if( checkFun() == false){

return false ;

}

// Check if all non-zero degree vertices are connected

if (isSC() == false)

return false;

// Check if in degree and out degree of every vertex is same

for (int i = 0; i < V; i++)

if (adj[i].size() != in[i])

return false;

return true;

}

// A recursive function to do DFS starting from v

void Graph::DFSUtil(int v, bool visited[])

{

// Mark the current node as visited and print it

visited[v] = true;

// Recur for all the vertices adjacent to this vertex

list<int>::iterator i;

for (i = adj[v].begin(); i != adj[v].end(); ++i)

if (!visited[\*i])

DFSUtil(\*i, visited);

}

// Function that returns reverse (or transpose) of this graph

// This function is needed in isSC()

Graph Graph::getTranspose()

{

Graph g(V);

for (int v = 0; v < V; v++)

{

// Recur for all the vertices adjacent to this vertex

list<int>::iterator i;

for(i = adj[v].begin(); i != adj[v].end(); ++i)

{

g.adj[\*i].push\_back(v);

(g.in[v])++;

}

}

return g;

}

bool Graph::isSC()

{

// Mark all the vertices as not visited (For first DFS)

bool visited[V];

for (int i = 0; i < V; i++)

visited[i] = false;

// Find the first vertex with non-zero degree

int n;

for (n = 0; n < V; n++)

if (adj[n].size() > 0)

break;

// Do DFS traversal starting from first non zero degree vertex.

DFSUtil(n, visited);

// If DFS traversal doesn't visit all vertices, then return false.

for (int i = 0; i < V; i++)

if (adj[i].size() > 0 && visited[i] == false)

return false;

// Create a reversed graph

Graph gr = getTranspose();

// Mark all the vertices as not visited (For second DFS)

for (int i = 0; i < V; i++)

visited[i] = false;

// Do DFS for reversed graph starting from first vertex.

// Staring Vertex must be same starting point of first DFS

gr.DFSUtil(n, visited);

// If all vertices are not visited in second DFS, then

// return false

for (int i = 0; i < V; i++)

if (adj[i].size() > 0 && visited[i] == false)

return false;

return true;

}

void printCircuit( vector < vector<int> > adj ){

map<int,int> edge\_count ;

for(int i=0 ; i<adj.size() ; i++){

edge\_count[i] = adj[i].size() ;

}

if(!adj.size() ){

return ; // empty graph

}

stack<int> curr\_path ;

vector<int> circuit ;

int curr\_v ;

if(sflag){

curr\_path.push(source) ;

curr\_v = source ;

}

else{

int x ;

for(x=0; x<adj.size() ;x++){

if(edge\_count[x]>0) break ;

}

curr\_path.push(x) ;

curr\_v = x ;

}

while( !curr\_path.empty()){

if(edge\_count[curr\_v]){

curr\_path.push(curr\_v) ;

int next\_v = adj[curr\_v].back() ;

edge\_count[curr\_v]-- ;

adj[curr\_v].pop\_back() ;

curr\_v = next\_v ;

}

else{

circuit.push\_back(curr\_v) ;

curr\_v = curr\_path.top() ;

curr\_path.pop() ;

}

}

// printing the path

vector<int> seq ;

for(int i=circuit.size()-1 ; i>=0 ; i--){

seq.push\_back(circuit[i]) ;

}

// clearing pair of path

path.clear() ;

for(int i=0 ; i<seq.size()-1 ; i++){

path.push\_back(pii( seq[i],seq[i+1] ) ) ;

}

for(int i=0 ; i<path.size() ; i++){

multimap <pii,string> :: iterator it ;

it = mp.find(path[i]);

if(i== 0 ){

cout<<it->second ;

}

else{

cout<< " " <<it->second ;

}

mp.erase(it) ;

}

cout<<endl ;

}

int main(){

//freopen("input.txt","r",stdin);

// freopen("output.txt","w",stdout) ;

int t,tcase , n;

string str ;

scanf("%d",&t) ;

tcase = 1;

while(t--){

Graph g(100) ;

vector < vector<int> > adj ;

adj.resize(100) ;

mp.clear() ;

scanf("%d",&n) ;

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

cin>>str ;

int u = (str[0]-'a') ;

int v = (str[str.size()-1] - 'a') ;

g.addEdge(u,v) ;

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

mp.insert(pair<pii,string>(pii(u,v),str)) ;

}

if(g.isEulerianCycle() == true ){

printf("Case %d: Yes\n",tcase++) ;

printCircuit(adj) ;

}

else{

printf("Case %d: No\n",tcase++) ;

}

}

return 0 ;

}