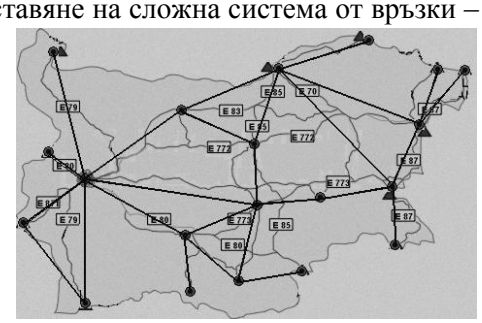
ВЪВЕДЕНИЕ В СТРУКТУРАТА ГРАФ

1. Определение за граф

Графът е нелинейна структура с голямо практическо приложение. В информатиката графите са едни от най-полезните абстрактни структури от данни. Много задачи, от различни области на науката и практиката, могат да бъдат моделирани с граф и решени, като се изпълни съответният алгоритъм върху него – намиране на пътища между две точки, за оцветяване на географска карта с минимален брой цветове, движение по шахматната дъска и др.

Графите са модели на реални обекти и служат за представяне на сложна система от връзки – авиолинии, транспортните и комуникационни мрежи, компютърни мрежи, web сайтове, схеми в електротехниката и други.



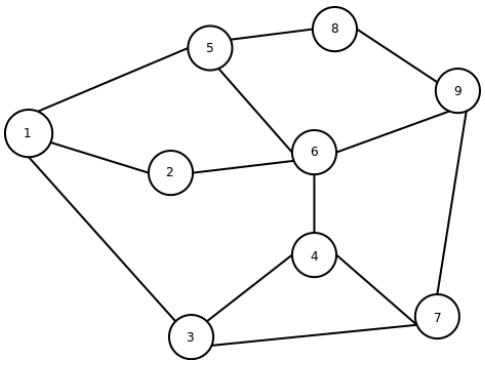
Пример: транспортната карта на Република България. На нея с точки са означени основните градове, а с линии - преките пътища между тях.

Определение: Краен ориентиран граф G е ще наричаме двойката (V, E), където:
$V = \{v_1, v_2, ... v_n\}$ е крайно множество от върхове, а $E = \{e_1, e_2, ... e_m\}$ е крайно множество от ориентирани ребра. Ще го означаваме G (V, E).

Всяко ребро представлява наредена двойка (v_i, v_j), където v_i и v_j са върхове и показва посоката на движение от връх v_i към v_j. Ако ребрата са неориентирани, т.е. двойките (v_i, v_j) са ненаредени, тогава графът се нарича *неориентиран*.

Неориентираният граф може да се разглежда като ориентиран, на който всички ребра са двупосочни. Затова в общия случай под граф се разбира ориентиран граф.

Най-често ориентираните графи се представят графично в равнината като множество от точки (кръгчета), означаващи върховете им, и свързващи ги стрелки - ребрата. Ако графът е неориентиран, тогава вместо стрелки ще чертаем линии. Ако например:



$V = \{1, 2, 3, 4, 5, 6, 7, 8, 9\}$

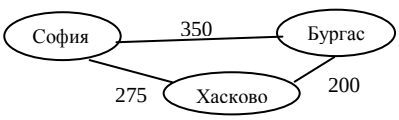
$E = \{(1, 2), (1, 3), (1, 5), (2, 6), (3, 4), (3, 7), (4, 6), (4, 7), (5, 6), (5, 8), (6, 9), (7, 9), (8, 9)\}$

е неориентиран граф, то бихме могли да го представим графично както е дадено на фигурата.

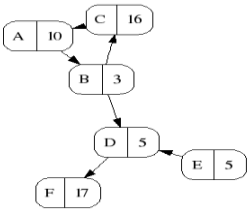
Стойностите на върховете са уникални и могат да бъдат произволни естествени числа или латински букви, но най-често се разглеждат първите n естествени числа. Затова вместо произволно множество V, ще приемем, че V е множеството от първите n естествени числа. Това води до редица удобства при реализацията – например, можем да използваме върховете като индекси на масив.

2. Понятия, свързани с графи

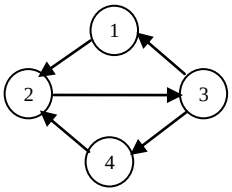
- **съседни върхове –** съществува ребро между тях
- **инцидентност** – на всяко ребро r съответства двойка върхове (v1, v2), казва се, че реброто r е инцидентно с v1 и v2.
- **степен на връх** – броят на ребрата, с които даден връх е свързан с другите върхове
- **изолиран връх** – връх от степен 0, /който не е инцидентен с ребро/
- **примка** – ребро, свързващо един и същ връх, т.е. началото и края му съвпадат
- **паралелни ребра** – две и повече ребра, които свързват два върха



- **тегло на ребро** – стойност, присвоена на ребро, с която най-често изразяваме разстоянието между върховете.
- **тегло на връх** – всеки връх има и стойност – например площ на населено място
- **път в граф** - път между два върха е последователност от съседни върхове или ребра, през които преминаваме, за да достигнем от единия до другия връх т.е. път между два върха съществува, ако е възможно да се премине по ребрата от първия до втория връх. Първият връх е начало, а вторият – край на пътя. Ако в пътя няма повтарящи се дъги, то пътят се нарича прост.



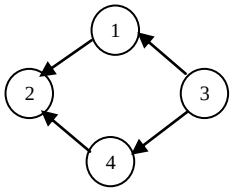
- **дължина на път** – броя на ребрата в пътя, или сумата от теглата им
- **разстояние между два върха** - дължината на най-краткия път между тях
- **цикъл** – път с начало и край един и същ връх.



Свързан цикличен граф

3. Видове графи

- **неориентиран**
- **ориентиран**
- **цикличен** – граф, който съдържа поне един цикъл в себе си
- **ацикличен** – граф, в който няма цикли
- **мултиграф** – граф, който съдържа паралелни ребра
- **свързан** – граф, в който между всеки два върха съществува път. Ако графът е ориентиран, трябва да има път и от първия към втория връх и обратно
- **несвързан** – между някой върхове няма път
- **претеглен** – всяко ребро или връх има тегло
- **пълен** – всеки два върха са свързани с ребро
- **празен** – граф без върхове

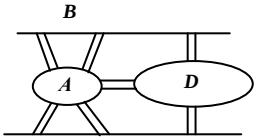


Несвързан ацикличен граф

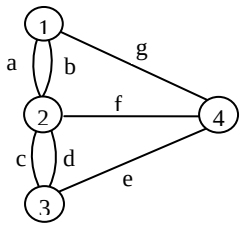
4. Видове цикли

а/ Ойлеров цикъл – Прост цикъл, съдържащ всички ребра, се нарича Ойлеров цикъл.

Град Кьонигсбергс (днешен Калининград) имал 7 моста, свързващи два острова и бреговете на две сливащи се реки. Възникнал въпросът дали е възможно тръгвайки от една точка на града, да се мине по седемте моста само по веднъж и отново да се стигне до началната точка.

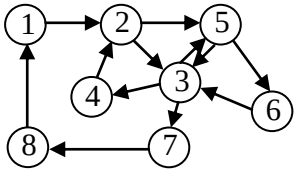


На следващата фигура е даден граф, представляващ модел на задачата за Кьонигсбергските мостове. Ребрата представляват мостовете, а върховете 2 и 4 – островите. Върховете 1 и 3 представят двата бряга на реката.

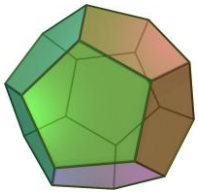


Чрез термините за графи задачата може да се формулира по следния начин: Даден е неориентиран граф G. Да се определи дали съществува път, който съдържа всички ребра на графа, но само по веднъж, т.е. дали съществува Ойлеров цикъл в графа. Ойлер дава отрицателен отговор.

Графът на следващата фигура съдържа Ойлеров цикъл:
1 2 3 4 2 5 3 5 6 3 7 8 1 или
(1,2),(2,3),(3,4),(4,2),(2,5),(5,3),(3,5),(5,6),(6,3),(3,7),(7,8),(8,1)



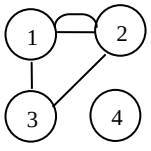
б/ Хамилтонов цилъл – Цикъл, съдържащ всички върхове на даден граф без повторение, се нарича Хамилтонов цикъл.

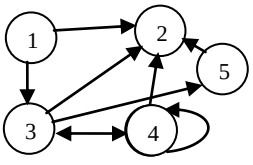


Додекаедърът е правилно геометрично тяло, състоящо се от 12 правилни петоъгълника. Хамилтън именува всеки един от 20$^{-те}$ върха на додекаедъра с име на град и формулира следната задача: Да се построи път, който тръгва от произволен град, минава през всички останали 19 града само по веднъж и се връща отново в началния град.

Додекаедърът може да се моделира с граф, съдържащ 20 върха и 25 ребра. В термините на граф задачата може да се формулира по следния начин: Даден е неориентиран граф. Да се определи дали съществува прост път, който съдържа всички върхове на графа само по веднъж.

Друг пример за Хамилтънов граф е задачата: “Да се обходят всички полета на шахматната дъска с шахматен кон, като на всяко поле се стъпи само веднъж?” Доказано е, че задачата има над 30 млн. решения.



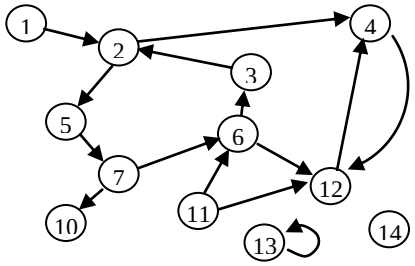


Зад. 1. Да се определи вида на графите в следващата фигура. Да се опишат множествата V и E, съдържащи върховете и ребрата.

Зад. 2. Да се начертаят всички възможни графи с три върха.

Зад. 3. С помощта на граф да се изобразят приятелските връзки между Иван, Петър, Вероника, Лилия и Ивайло, ако се знае, че Иван е приятел на Ивайло и Вероника, Петър е приятел на Лилия, Вероника и Ивайло, Лилия е приятел на Ивайло и Вероника.

Зад. 4. За дадения граф да се посочат:



а/ степените на върховете 1, 2, 4, 6, 12, 13;

б/ изолирани върхове, ако има такива;

в/ паралелни ребра, ако има такива;

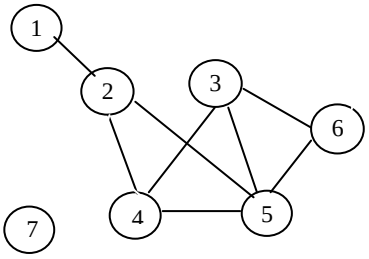
г/ върховете със степен 4

д/ върховете, които имат примки

е/ пътища между върховете 1 и 4, 2 и 6, 3 и 12, 4 и 5 и дължината на всеки от тях;

ж/ цикъл;

Зад. 5. За графа на следващата фигура да се посочат:



а/ множествата V и E

б/ изолирани върхове, ако има такива;

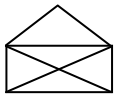
в/ паралелни ребра, ако има такива;

г/ степента на върховете му;

д/ пътища между върховете 2 и 4, 1 и 4, 1 и 6, 3 и 6, 1 и 7 и дължината на всеки от тях;

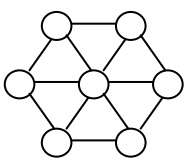
е/ цикъл;

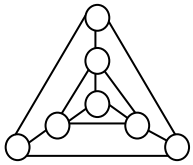
Зад. 6. Да се изчертае дадената фигура без вдигане на молива. Има ли Ойлеров цикъл? А Хамилтънов цикъл?



Зад. 7. Да се представят с графи телата: правилна триъгълна пирамида, правилна шестоъгълна пирамида, куб. Да се провери дали в тях има Ойлерови цикли.

Зад. 8. Да се определи дали в дадените графи има Хамилтонови цикли.





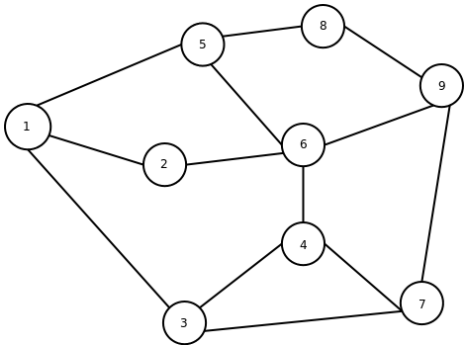
ФИЗИЧЕСКО ПРЕДСТАВЯНЕ НА ГРАФИТЕ

Има много начини за представяне на граф в компютъра, но най-често използваните са чрез матрица на съседство и списък на съседство. Употребата на единия или другия начин зависи от конкретното задание.

1. Представяне чрез матрица на съседство

Нека G е произволен граф с n върха.

Матрицата на съседство представлява квадратна таблица A[n][n], в която стойността на A[i][j] е равна на 1 ако съществува реброто (i, j) и 0 - в противен случай.



Ако графът е претеглен, тогава вместо единица записваме директно съответната тежест на реброто и говорим за *матрица на теглата*.

Ако графът е неориентиран, то тогава A[i][j] = A[j][i], т.е. матрицата е симетрична относно главния си диагонал.

Ето как би изглеждала матрицата на съседство за дадения неориентиран граф:

	1	2	3	4	5	6	7	8	9
1	0	1	1	0	1	0	0	0	0
2	1	0	0	0	0	1	0	0	0
3	1	0	0	1	0	0	1	0	0
4	0	0	1	0	0	1	1	0	0
5	1	0	0	0	0	1	0	1	0
6	0	1	0	1	1	0	0	0	1
7	0	0	1	1	0	0	0	0	1
8	0	0	0	0	1	0	0	0	1
9	0	0	0	0	0	1	1	1	0

За представянето на матрицата използваме двумерен масив с размерност n x n.

Представянето на граф чрез двумерен масив предоставя удобни възможности за извършване на някои операции с графи – като например проверката за наличие на ребро между два, добавяне или изтриване на ребро. Недостатък на това представяне е необходимостта от голямо количество памет за графи с голям брой върхове.

2. Представяне чрез списък на съседство /списък на инцидентност/

При това представяне за всеки връх от графа пазим списък на съседите му.

Ето как биха изглеждали списъците за горния граф:

1 -> 2, 3, 5
2 -> 6, 1
3 -> 1, 4, 7
4 -> 3, 6, 7
5 -> 1, 6, 8
6 -> 2, 4, 5, 9
7 -> 3, 4, 9
8 -> 5, 9
9 -> 6, 7, 8

Най-често за компютърна реализация се използва масив от свързани списъци или масив от вектори. Представянето на граф чрез списъци на съседство е удобно при графи с голям брой върхове, когато се налага в графа да се добавят или изключват върхове, за намиране на всички съседи на даден връх и др.

3. Създаване на граф чрез матрица на съседство

Първоначално се въвежда броя на върховете и на ребрата. След това на всеки ред се въвеждат двойка числа – стойностите на двата върха, свързани чрез съответното ребро. Ако графът е претеглен, за всяко ребро се въвеждат три числа, като третото е теглото на реброто.

Ще декларираме масив с размери 100x100, а ще използваме само част от него. Това ни позволява да работим с индекси от 1 до V. При глобално деклариране на масив, всички негови елементи получават автоматично стойност 0, затова ние ще дадем стойност 1 само на някои от тях. При въвеждане на два върха – x и y, определящи ребро, даваме стойност 1 на елементите G[x][y] и G[y][x], ако графът е неориентиран т.е. G[x][y]=G[y][x]=1. Ако графът е ориентиран, само G[x][y]=1.

Чрез функция Izhod ще изведем матрицата на съседство.

Следващата програма въвежда неориентиран граф и извежда матрицата му на съседство.

```
#include <iostream>
using namespace std;
int V, E, G[100][100];
void Vhod ()
{     int x, y;
      cout << "V = "; cin >> V;
      cout << "E = "; cin >> E;
      cout << "Enter first and second points:"<<endl;
      for(int i = 1; i <= E; i++)
      {      cin >> x >> y;
             G[x][y] = G[y][x] = 1;
      }
}
void Izhod ()
{     for (int i=1; i<=V; i++)
      {      for (int j=1; j<=V; j++)
                    cout<<G[i][j]<<" ";
             cout<<endl;
      }
}
int main()
{     Vhod();
      Izhod();
      return 0;
}
```

ТЕСТ: за графа от схемата въвеждаме:

```
V = 9
E = 13
 1 2
 1 3
 1 5
 2 6
 3 4
 3 7
 4 6
 4 7
 5 6
 5 8
 6 9
 7 9
 8 9
```

II начин: Обхождаме върховете и за всеки посочваме съседните върхове

```
void Vhod ()
{     int x;
      cout << "V = "; cin >> V;
      for (int i=1; i<=V; i++)
      {      cout<<"Sysedi na "<< i<<": ";
             while (cin>>x)
                    G[i][x]=G[x][i]=1;
             cin.clear();
      }
}
```

4*. Създаване на граф чрез списък на съседство – За всеки връх се пази списък на съседите

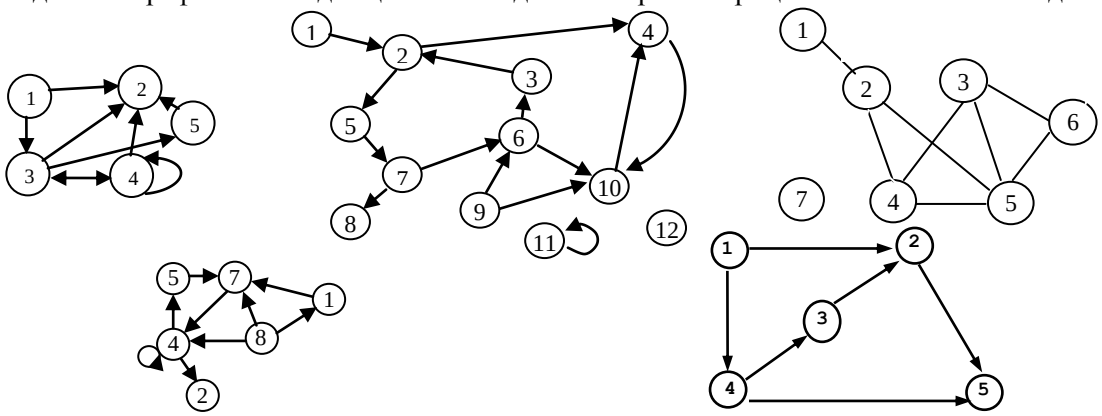
I начин: За реализацията използваме двумерен масив, като в елемента с индекс 0 на всеки ред ще пазим броя на съседите на този връх. При добавяне на нов съсед, ще увеличаваме стойността в тази клетка. При извеждането на съседите, от там ще вземем броя на съседите на всеки връх.

```
#include<iostream>
using namespace std;
int V, E, G[100][100];
void Vhod ()
{     int x, y;
      cout << "V = "; cin >> V;
      cout << "E = "; cin >> E;
      cout << "Enter first and second points:"<<endl;
      for(int i = 1; i <= E; i++)
      {  cin >> x >> y;
         G[x][0]++;
         G[y][0]++;
         G[x][G[x][0]] = y;
         G[y][G[y][0]] = x;
      }
}
void Izhod ()
{     for (int i=1; i<=V; i++)
      {   cout << i <<"->";
          for (int j=1; j<=G[i][0]; j++)
                    cout<<G[i][j]<<" ";
             cout<<endl;
      }
}
int main()
{   Vhod();    Izhod();     return 0;
}
```

II начин:. За реализацията ще използваме структура list или vector. И двете структури се използват по един и същ начин, но се предпочита vector, поради по-бързото обхождане на елементите. Тъй като за всеки връх трябва да запазим вектор от съседите му, се използва масив от вектори.

```
#include <iostream>
#include <vector>
using namespace std;
int V, E ;
vector <int> G[100];
void Vhod()
{     int x, y;
      cout << "V = "; cin >> V;
      cout << "E = "; cin >> E;
      cout << "Enter first and second points:"<<endl;
      for(int i = 1; i <= E; i++)
      {      cin >> x >> y;
             G[x].push_back(y);
             G[y].push_back(x);
      }
}
void Izhod ()
{     for (int i=1; i<=V; i++)
      {      cout<<i<<"->";
             vector <int>::iterator it;
             for (it=G[i].begin(); it !=G[i].end(); it++)
                    cout<<*it<<", ";
             cout<<endl;
      }
}
int main()
{     Vhod(); Izhod(); return 0;
}
```

Зад. 1. За графите от следващите схеми да се построи матрицата и списъка на съседство.



Зад. 2. Даден е граф с 6 върха, представен чрез следващата таблица :

	1	2	3	4	5	6
1	0	1	0	0	0	0
2	0	0	1	0	0	0
3	0	0	0	1	1	0
4	0	1	0	0	1	0
5	1	0	0	0	0	0
6	0	0	0	0	0	0

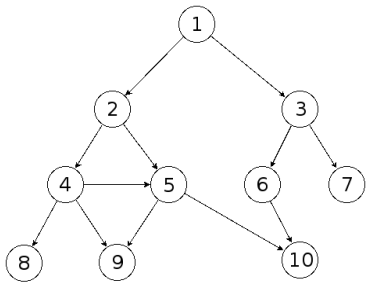
а/ Да се начертае графът.
б/ Да се определи дали графът е ориентиран или не.
в/ За всеки връх да се определи неговата степен.
г/ Има ли изолиран връх и ако да – посочете го.
д/ За всеки връх да се посочи броят на излизащите от него ребра.
е/ Да се определи има ли път между върховете 1 и 5. Ако да - един или няколко, посочете ги.
ж/ Да се провери дали има цикли в дадения граф.

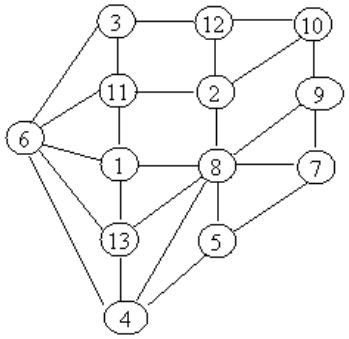
Зад. 3. Даден е граф с 6 върха, представен чрез списъците на съседство:

a -> b, d
b -> a, c, d
c -> a, d, e
d -> a, b, c
e -> a, b, d
f ->

а/ Да се определи дали графът е ориентиран или не.
б/ За всеки връх да се определи неговата степен.
в/ Има ли изолиран връх и ако да – посочете го.
г/ За всеки връх да се посочи броят на излизащите от него ребра.
д/ По дадени номера на два върха да се изведе има ли път между тях. Ако да – един или няколко, посочете ги.
е/ Да се провери дали има цикли в дадения граф.

Зад. 4. Дадения граф представете с матрица на съседство и списък на съседите



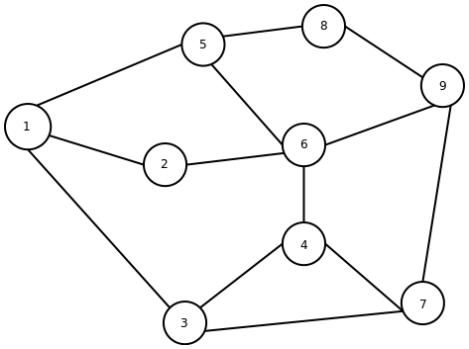


ОБХОЖДАНЕ НА ГРАФ В ДЪЛБОЧИНА
/Depth First Search/

Обхождането на даден граф е процедура за преминаване в определен ред през всички върхове на графа. Основни методи за обхождане на граф са: обхождане на граф в дълбочина и обхождане на граф в ширина.

1. Представяне на обхождането на граф в дълбочина

Обхождането на граф в дълбочина има изключително широко приложение при компютърната обработка на графи. Основава се на общата алгоритмична техника търсене с връщане назад. Приложим е както при ориентирани, така и при неориентирани графи. Този вид обхождане се стреми да се "спусне" колкото се може "по-надълбоко" в графа, т.е., тръгвайки от един връх по произволно негово ребро стигаме до следващия връх, от който продължаваме по същия начин като, се стремим да достигнем най-отдалечения връх по този път. Спираме, когато срещнем вече посетен връх или връх без наследник. След това се връщаме назад, за да преминем по другите ребра на всеки връх от пътя. Обхождането спира при връщане в началния връх.



За даден граф в общия случай има повече от едно възможно обхождане в дълбочина.

Пример: 1 2 6 4 7 9 8 5 3, 1 3 4 6 9 8 5 7 2, 1 5 6 2 9 8 7 3 4.

2. Описание на алгоритъма

Можем да реализираме алгоритъма рекурсивно или итеративно. Рекурсивният вариант е подходящ, защото алгоритъма се основава на техниката търсене с връщане назад, а рекурсията има 2 фази – при развиването слиза в дълбочина, а при свиването – се връща назад.

1. Започваме обхождането от някакъв начален връх t – извеждаме стойността му
2. Отбелязваме го като посетен
3. За всеки негов непосетен съсед изпълняваме същия алгоритъм /по произволно ребро достигаме до друг връх съсед, непосетен и извършваме същите действия рекурсивно./

3. Програмна реализация – функция dfs

1) Разглеждаме върха t, извършваме с него необходимите действия, напр. извеждаме t.

```
cout << t << " ";
```

2) Отбелязваме го като посетен.

За тази цел декларираме масив bool visited[100], в който да отбелязваме със стойност true, ако върхът е бил посетен. Елементите на масива автоматично получават стойност 0 /false/. Маркираме връх като посетен чрез **visited[t] = true.**

3) За всеки негов непосетен съсед i извикваме рекурсивно обхождането: dfs(i)

За да намерим съсед на връх t, обхождаме всички върхове и разглеждаме ред t от матрицата на съседство: ако G[t][i]==1, то връх i е съсед на t. Върхът i е непосетен, ако visited[i]==false. Затова условието връх i да е съсед на t и да не е посетен става: G[t][i]==1 && visited[i]==false, но може да се запише така: G[t][i] && !visited[i]. Следователно тази стъпка се реализира така:

```
for(int i = 1; i <= V; i++)
        if(G[t][i] && !visited[i])
            dfs(i);
```

Реализация чрез матрица на съседство

```
void dfs(int t)
{
   cout <<t<<" ";
   visited[t] = true;
   for(int i = 1; i <= V; i++)
         if(G[t][i] && !visited[i])
             dfs(i);
}
```

чрез списък на съседство

```
void dfs(int t)
{  cout<<t<<" ";
   visited[t]=true;
   int br=G[t].size();
   for(int i=0;i<=br-1;i++)
     if(!visited[G[t][i]])
         dfs(G[t][i]);
}
```

В main я извикваме като посочим начален връх – най-често е връх 1 – dfs(1).

Изход: 1 2 6 4 3 7 9 8 5

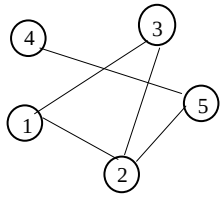
При тази реализация се обхождат всички върхове, които са свързани в една компонента. Ако има изолиран връх или други компоненти, не можем да достигнем до тях, тъй като се движим по ребрата. За да обходим всички върхове, в main добавяме цикъл, с който обхождаме всички върхове и за всеки непосетен връх пускаме dfs, т.е for (int i=1; i<=V; i++) if (!visited[i]) dfs(i);

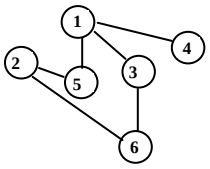
Ето и цялата програма.

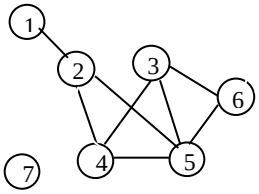
```
#include <iostream>
using namespace std;
int V, E, G[100][100];
bool visited[100];

void Vhod ()
{  int x, y;
   cout << "V = "; cin >> V;
   cout << "E = "; cin >> E;
   cout  <<  "Enter  first  and  second
points:"<<endl;
   for(int i = 1; i <= E; i++)
   {  cin >> x >> y;
      G[x][y] = G[y][x] = 1;
   }
}
```

```
void Izhod ()
{  for (int i=1; i<=V; i++)
   { for (int j=1; j<=V; j++)
        cout<<G[i][j]<<" ";
     cout<<endl;
   }
}
void dfs(int t)
{  cout <<t<<" ";
   visited[t] = true;
   for(int i = 1; i <= V; ++i)
      if(G[t][i] && !visited[i])
         dfs(i);
}
int main()
{  Vhod();
   Izhod();
   dfs(1);
   cout << endl;
   return 0;
}
```

Тест:







При компютърната реализация съседите на даден връх се обхождат в нарастващ ред, затова за горните примери ще изведе:

Пример 1: 1 2 3 5 4, Пример 2: 1 3 6 2 5 4 Пример 3: 1 2 4 3 5 6

4. Оценка на сложността

За всеки един връх dfs намира наследниците му и пуска обхождане от тях. Намираме наследниците със сложност $\Theta(V)$. И имайки предвид, че правим това точно по веднъж за всеки връх (тъй като пазим visited), то окончателно получаваме сложност $\Theta(V^2)$. Реализацията чрез списък на съседство има сложност $\Theta(V + E)$.

5*. Итеративно обхождане – използва стек, изисква включване на файл stack – #include <stack>

I начин – върхът се обявява за посетен при изваждане от стека и разглеждането му. Така в стека може да попадне един и същи връх, достигнат от два негови съседа. Това налага преди разглеждането му да проверим дали е все още е непосетен и тогава да го използваме и обявим за посетен.

```
void dfs(int v)
{    stack<int>s;
     s.push(v);
     while ( !s.empty())
     {     v=s.top();
           s.pop();
           if (!visited[v])
           {
           visited[v]=true;
           cout<<v<<" ";
           for (int i=1;i<=V;i++)
            if (G[v][i] && !visited[i])
                 s.push(i);
           }
     }
}
```

Редицата, която ще се изведе ще се различава от тази в предишната реализация, тъй като върховете се вадят от стека в обратен ред на постъпване. За да изведем същата редица, както при рекурсивната реализация /лексикографски подредена/, трябва в цикъла да зададем обратен ред на обхождане `for (i=V; i>=1; i--)`.

II начин – върхът се обявява за посетен още при поставянето му в стека, макар че извеждането му става по-късно когато го изведем от стека. Така този връх може да се постави в стека само веднъж. Той още не е изведен, но стои в стека и изчаква реда си.

Обявяваме началния връх за посетен
Поставяме началния връх в стека
Докато в стека има върхове:
    Изваждаме връх от стека
    Обхождаме го /извеждаме стойността му/
    За всички негови съседи, които не са били посетени:
        Обявяваме го за посетен

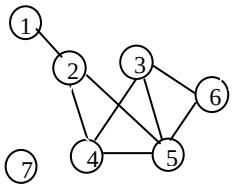
Поставяме го в стека

```
void dfs(int v)
{    visited[v]=true;
     stack<int>s;
     s.push(v);
     while ( !s.empty())
     {     v=s.top();
           s.pop();
           cout<<v<<" ";
           for (int i=1;i<=V;i++)
           if (G[v][i] && !visited[i])
           {     visited[i]=true;
                 s.push(i);
           }
     }
}
```

Зад. 1. Даден е неориентиран граф. Да се изведат компонентите на графа на отделни редове.

Решение: При обхода можем да достигнем само до върховете свързани в една компонента с началния връх. За да обходим и другите компоненти, трябва да пуснем обхождане в дълбочина за всеки непосетен връх на графа. В main() добавяме:

```
for (int i=1; i<=V; i++)
     if (!visited[i])
     {  dfs(i);
        cout<<endl;
     }
```



Тест: Вход: V= 7
E = 8
1 2 2 4 2 5 3 4 3 5 3 6 4 5 5 6
Изход: 1 2 4 3 5 6
7

Зад. 2. Даден е неориентиран граф. Да се намери броя на свързаните компоненти на графа.

Свързаната компонента се състои от няколко свързани върха. Несвързаният граф се състои от няколко компонента, а свързаният – само от 1.

Решение: В main () добавяме:

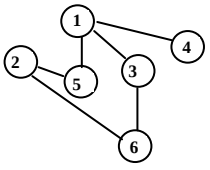
```
int br=0;
for (int i=1; i<=V; i++)
       if (!visited[i]){ dfs(i); br++;}
cout << br << endl;
```

В dfs премахваме реда, с който извеждаме стойността на текущия връх.

Зад. 3. Даден е неориентиран граф. Да се провери дали е свързан.

Решение: Проверяваме дали броя на свързаните компоненти е 1, тогава графът е свързан.

Зад. 4. Даденият граф илюстрира учител, учениците в група по СИП и телефонните контакти между тях. Налага се промяна в датата и часа на следващото занятие и учителят (1) трябва да съобщи на учениците промяната. Като знаете, че всеки ученик може да предаде съобщението на тези, с които е свързан, на колко най-малко ученика трябва да се обади учителят?



Решение:

I начин: В dfs преброяваме колко пъти сме пуснали dfs от началния връх 1 към негов наследник – т.е. ако t==1, увеличаваме брояча.

```
void dfs(int t)
{
   visited[t] = true;
   for(int i = 1; i <= V; ++i)
   if(G[t][i] && !visited[i])
   {
    if (t==1) br++;
    dfs(i);
   }
}
```

II начин: Премахваме ребрата между 1 и наследниците му и намираме броя на свързаните компоненти.

Задача C1. Company /ЕСЕНЕН ТУРНИР „Джон Атанасов“ 2011 г., автор: Александър Георгиев/

Ели започна работа в голяма софтуерна компания. Йерархията в компанията има дървовидна структура, като всеки човек (освен баш шефът) има точно по един пряк началник. Компанията е разделена на екипи, като един програмист с всичките си преки и непреки подчинени (ако има такива) се счита за един екип. Това означава, че един екип може да се състои от няколко други такива. За пример ще дадем компания като Майкрософт, където има екип, който се занимава с Office, който от своя страна се състои от екипи, които се занимават с Word, Excel, и т.н. В случая на Ели, например, Станчо е шеф на Пешо и Ели, Ели е шеф на Крис, а Пешо е шеф на Гошо и Тошо. Така Станчо, Пешо, Ели, Крис, Гошо и Тошо образуват един екип. Също така Ели и Крис са един екип, а Пешо, Тошо и Гошо са друг екип.

Фирмата има много дълъг, но за съжаление тесен офис, в който има място само за един ред от компютри. Шефът на фирмата е зает най-левия от тях и иска да разпредели програмистите по такъв начин, че:

a) Прекият началник на всеки програмист да се намира наляво от него.

b) Членове на всеки от екипите да заемат непрекъсната последователност от компютри (т.е. да са един до друг).

Ако вземем примера, който дадохме по-рано, едно възможно нареждане би било Станчо, Пешо, Гошо, Тошо, Ели, Крис. Помогнете на Ели да се подмаже на шефа, като напишете програма, която по дадена структура на фирмата, определя нареждането на програмистите.

Вход: На първия ред на стандартния вход ще бъде зададен броят програмисти във фирмата *N*. Ще представим програмистите с номера от 1 до N, включително, като 1 е шефът на фирмата (който няма пряк началник). На следващите N – 1 реда ще бъде зададена по една двойка числа *W1 W2* указващи, че *W1* е пряк началник на *W2*.

Изход: На стандартния изход изведете един ред, съдържащ *N* цели числа между 1 и N – подредбата на програмистите в изискания ред. Ако има повече от една възможна подредба, изведете лексикографски най-малката. Наредба A е лексикографски по-малка от наредба B, ако числото на първата позиция, в която се различават е по-малко в A от това в B. Например {1, 3, 4, 6, 7, 2, 5} е по-малко от {1, 3, 5, 2, 4, 6, 7}.

Ограничения: $1 \le N \le 200{,}000$, В 70% от тестовете *N* ще е по-малко или равно на 10000.

В 85% от тестовете не повече от 200 човека ще имат един и същ пряк началник.

Пример: **Вход:** **6** **Изход:** 1 2 5 4 3 6

1 2 2 5 4 3 1 4 4 6

Вход: **14** **Изход:** 1 2 4 5 6 3 7 8 9 10 12 13 11 14

9 11 1 9 10 12 1 3 3 8 2 4 2 5 10 13 1 2 3 7 9 10 2 6 11 14

Пояснение: В първи пример Станчо е с номер 1, Ели с 2, Крис с 5, Пешо е с 4, Гошо е с 3, а Тошо с 6.

Решение: Използваме неориентиран граф, за който изпълняваме dfs. При статичната реализация чрез масив, и с рекурсия и с итерация се хващат до 14 теста. При използване на вектор чрез рекурсия работи на 100%. В този случай трябва да сортираме наследниците на всеки връх.

```
#include <iostream>
using namespace std;
int n;
bool G[10001][10001]={0};
bool visited  [200001]={0};
void dfs(int t)
{   cout<<t;
    visited[t]=true;
    for(int i=1;i<=n;i++)
      if(G[t][i]==1 && !visited[i])
      {
          cout <<" ";
          dfs(i);
     }
}
int main()
{    int x, y, i;
    cin >> n;
    for(i = 1; i <= n-1; i++)
    {     cin >> x >> y;
          G[x][y]=1;
     }
   dfs(1);
   cout<<endl;
   return 0;
}
```

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
int n;
vector <int> G[200001];
bool visited  [200001]={0};
void dfs(int t)
{     cout<<t;
      visited[t]=true;
      int br=G[t].size();
      for(int i=0;i<=br-1;i++)
        if(!visited[G[t][i]])
        {  cout <<" ";
            dfs(G[t][i]);
    }
}
int main()
{   int x, y, i;
     cin >> n;
     for(i = 1; i <= n-1; i++)
     { cin >> x >> y;
       G[x].push_back(y);
     }
     for (i = 1; i<=n; i++)
       sort(G[i].begin(),G[i].end());
     dfs(1);
     cout<<endl;
     return 0;
}
```

НАМИРАНЕ НА ВСИЧКИ ПЪТИЩА МЕЖДУ ДВА ВЪРХА

Нека е даден ориентиран граф G и два негови върха t и u. Търсят се всички пътища в G, започващи от t и завършващи във u.

Всички пътища между два върха в граф се намират като тръгнем от първия връх и се спускаме надолу докато достигнем търсения връх /така намираме един път/, а след това трябва да се върнем една стъпка назад, за да потърсим друг възможен път от текущия връх нататък. Това ни подсказва, че алгоритъмът ще е подобен на алгоритъма за обхождане на граф в дълбочина.

Алгоритъм: Обхождайки графа в дълбочина, всички върхове, през които сме минали, се маркират като посетени и не се допуска повторното им обхождане. След разглеждане на някой връх и на всички негови наследници, при обратния ход на рекурсията този връх се обявява за непосетен. Това дава възможност за конструиране не само на един, а на всички възможни пътища от t до u. За съхраняване на текущия път ще използваме масив Path [100]. Той ще съдържа всички върхове, през които сме минали. Ако достигнем връх u, т.е намерили сме път от t до u, добавяме и върха u в масива, и извеждаме елементите му. Върхът u не се обявява за visited, за да може да се достигне и от другаде.

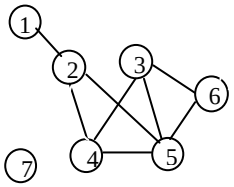
Този алгоритъм се свежда до пълно изчерпване на метода на търсенето с връщане назад.

Ще го реализираме чрез функция allPaths(int u, int v), където u и v са върховете в начало и края на пътя.

```
void allPaths(int t,int u)
{
  if (t!= u)
  {
   visited[t]=true;
   Path[vr++] = t;
   for (int i=1; i<=V; i++)
   if (G[t][i] && !visited[i]) allPaths(i, u);
   visited[t]=false;
   vr--;
  }
  else
  {
    Path[vr] = t;
    for (int i=1; i<=vr; i++) cout<<Path[i]<<" ";
    cout << endl;
  }
}
```

Забележка: Преди всички функции добавяме декларациите**:**int Path[100]; int vr=1;, а в main() извикваме функцията allPath, като посочим върховете, между които търсим път: allPath(1, 6);

Зад. 1. Даден е граф. Да се изведат всички пътища от връх 1 до връх 6.

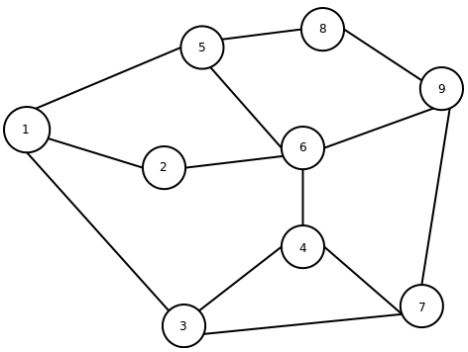


Тест: Вход: V= 7
E = 8
1 2 2 4 2 5 3 4 3 5 3 6 4 5 5 6

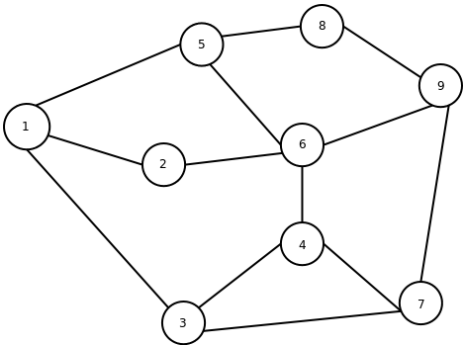
Изход:

```
1 2 4 3 5 6
1 2 4 3 6
1 2 4 5 3 6
1 2 4 5 6
1 2 5 3 6
1 2 5 4 3 6
1 2 5 6
```

Зад. 2. Изведете всички пътища от връх 1 до връх 6 в зададения граф. Намерете броя им.



ОБХОЖДАНЕ НА ГРАФ В ШИРИНА
/Breadth First Search/



1. Представяне на обхождането на граф в ширина

При обхождане в ширина разглеждането на върховете става по нива – първо се разглеждат всички наследници на текущия връх и едва след това техните наследници. Първият обходен връх се анализира пръв.

Започваме обхождането от произволен връх. След това обхождаме всички негови съседи. След тях обхождаме техните съседи и т.н., докато достигнем до вече посетен връх или връх без наследници.

2. Описание на алгоритъма:

Трябва да запазим върховете по реда на обхождане, за да разгледаме и наследниците им в този ред. За тази цел използваме структура опашка – всеки посетен връх се поставя в опашката и се извлича от нея, когато му дойде редът.

1. Разглежда се върхът t, извършваме с него необходимите действия, напр. извеждаме v
2. Обявяваме върхът t за посетен.
3. Добавяме го в опашката от върхове.
4. Докато опашката не е празна се повтаря следното:
   4. 1. Изважда се първият елемент от опашката.
   4. 2. Разглеждат се всички негови непосетени съседи и за всеки от тях
      4.2.1. извършва се някакво действие, напр. извеждаме стойността му
      4.2.2. обявява се за посетен
      4.2.3. добавя се в опашката.

3. Програмна реализация – функция bfs

```
//чрез матрица на съседство
void bfs(int t)
{ cout<<t <<" ";
   visited[t]=true;
   queue<int> q;
   q.push(t);
    while (!q.empty ())
    { t = q.front ();
      q.pop ();
      for (int i = 1; i <= V; i++)
      if(G[t][i] && !visited[i])
      {     cout<<i<<" ";
            visited[i]=true;
            q.push(i);
      }
    }
}
```

```
//чрез списък на съседство
void bfs(int t)
{  cout<<t<<" ";
   visited[t]=true;
   queue<int> q;
   q.push(t);
   while(!q.empty())
   {  t=q.front();
      q.pop();
      int br =G[t].size();
      for(int i=0;i<=br-1;i++)
         if(!visited[G[t][i]])
         {  cout<<G[t][i]<<" ";
            visited[G[t][i]] = true;
            q.push(G[t][i]);
         }
   }
}
```

Ето и цялата програма

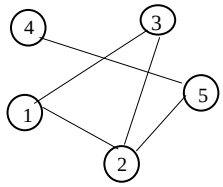
```
#include <iostream>
#include <queue>
using namespace std;
int V, E, G[100][100];
bool visited[100];
void Vhod ()
{   int x, y;
    cout << "V = "; cin >> V;
    cout << "E = "; cin >> E;
    cout<<"Enter first and second points:"<<endl;
    for(int i = 1; i <= E; i++)
    {      cin >> x >> y;
    G[x][y] = G[y][x] = 1;
   }
}

void Izhod ()
{  for (int i=1; i<=V; i++)
   {for (int j=1; j<=V; j++)
            cout<<G[i][j]<<" ";
     cout<<endl;
    }
  }
```

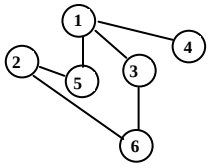
```
void bfs(int t)
{ cout<<t <<" ";
  visited[t]=true;
  queue<int> q;
  q.push(t);
  while (!q.empty ())
   {  t = q.front ();
      q.pop ();
      for (int i = 1; i <= V; i++)
         if(G[t][i]&& !visited[i])
          {       cout<<i<<" ";
                  visited[i]=true;
                  q.push(i);
           }
   }
```

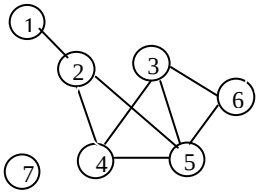
```
}

int main()
{
    Vhod();
    Izhod();
    bfs(1);
    cout << endl;
    return 0;
}
```

Като резултат от обхождането за дадения граф се извежда редицата: 1 2 3 5 6 4 7 8 9

Тест:







Резултат: 1 2 3 5 4 1 3 4 5 6 2 1 2 4 5 3 6 /без 7/

4. Оценка на сложността

BFS има същата сложност като DFS: $\Theta(V^2)$ с матрица на съседство и $\Theta(V + E)$ със списък на съседство.

II начин – при изваждането от опашката извеждаме стойността на върха и го обявяваме за посетен
Добавяме началния връх към опашката
Докато опашката не е празна
Вадим връх от нея
Ако върхът не е бил посетен
извеждаме стойността му
обявяваме го за посетен
всички негови непосетени наследници добавяме в опашката

```
void bfs(int t)
{
   queue<int> q;
   q.push(t);
    while (!q.empty ())
    {     t = q.front ();
        q.pop ();
        if (!visited[t]){
        cout<<t <<" ";
        visited[t]=true;
          for (int i = 1; i <= V; i++)
        if(G[t][i] && !visited[i])
          q.push(i);
    }
   }
}
```

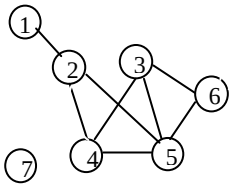
Като резултат от обхождането се извежда същата редица: 1 2 3 5 6 4 7 8 9

Зад. 1. Даден е граф и един негов връх. Да се намерят минималните пътища от дадения връх до всички останали.

Зад. 2. Даден е граф. Да се намери броя на свързаните компоненти и броя на върховете във всяка компонента.

НАМИРАНЕ НА НАЙ-КРАТЪК ПЪТ МЕЖДУ ДВА ВЪРХА /ПО БРОЙ НА РЕБРАТА/

Нека е даден граф G(V, E) и два негови върха t и u. Търсим път в G с начало върхът t и край върхът u, който има минимална дължина по брой върхове.



Тъй като търсим най-кратък път, ще разгледаме най-напред преките наследници на първия връх, след това техните наследници и т.н. ще обхождаме върховете по нива. Първият намерен път е най-късия.

Например, при търсене на път от 1 до 5, обхождането по нива е следното: I ниво – връх 1, II ниво – 2, III ниво – 4, 5, IV ниво – 3, 6. На ниво III е открит търсения връх 5, пътят е 1, 2, 5, с дължина 2 – броя на ребрата по пътя, или броя на нивата без 1.

Алгоритъм: Обхождаме графа в ширина като започваме от дадения връх t. При обхождането за всеки връх запомняме неговият родител, за да можем след това да възстановим пътя. За целта ще използваме масив Parent [100]. След като приключи обхождането, ще извикаме функция printWay, която да изведе пътя.

```
void minPath(int t,int u)
{ visited[t]=true;
  queue<int> q;
  q.push(t);
  while (!q.empty ())
   {   int s = q.front (); //Тъй като за извеждане на пътя ще трябва да запазим
                           //стойността на t, тук използваме променлива s.
       q.pop ();
       for (int i = 1; i <= V; i++)
       if(G[s][i] && !visited[i])
           { visited[i]=true;
             q.push(i);
             Parent[i]=s;
            }
   }
  cout<<"Min path from "<<t<<" to "<< u <<": ";
  printWay(t,u);
}
```

След приключване на обхождането, в масив Parent се съдържат предшествениците на всички върхове на графа, без началния, т.к. той няма такъв.

Функция printWay ще възстанови пътя по следния начин – тръгва от връх u, последен в пътя, преминава до неговия предшественик, след това до предшественика на предшественика и т.н. докато достигне началния връх. В сила е следната рекурсивна зависимост: пътя от t до u = пътя от t до Parent[u] и самия връх u. Затова и реализацията на функцията ще е рекурсивна. Базовият случай е когато двата върха t и u съвпаднат.

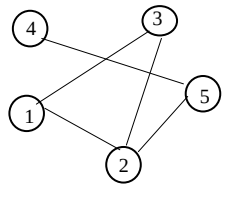
Например, за да възстанови път: 1, 2, 3, 4, най-напред се извиква printWay (1, 4), после за 1 и 3, после за 1 и 2, накрая за 1 и 1 /базов/. При рекурсивното извикване t=1 не се променя, а стойностите на u са: 4, 3, 2, 1 и представляват пътя, но в обратен ред, т.е. от u към t.

Пътя ще се изведе чрез извеждане на текущите стойности на u. Освен това, извеждането ще става при свиването на рекурсията, тъй като тогава стойностите на u ще се вземат от стека и пътя ще бъде описан от началния към крайния връх.

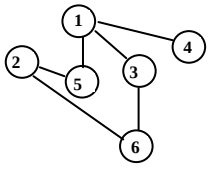
```
void printWay(int t, int u)
{     if(t != u)
            printWay(t, Parent[u]);
      cout << u << " ";
}
```

Тази функция се извиква от функция minPath, затова трябва да бъде разположена преди нея.

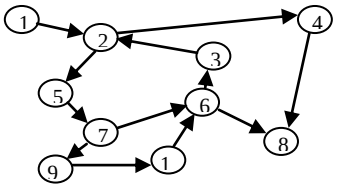
Тест:



Път от 1 до 5 е: 1, 2, 5



От 1 до 6: 1, 3, 6



От 1 до 6: 1, 2, 5, 7, 6

II начин – итеративно извеждане на пътя

```
void printWay(int t, int u)
{   stack <int> st;
    st.push(u);
    while (t!=Parent[u])
    {   u=Parent[u];
        st.push(u);
    }
    st.push(t);
    while (!st.empty())
    {   u=st.top();
        st.pop();
        cout << u << " ";
    }
    cout <<endl;
}
```