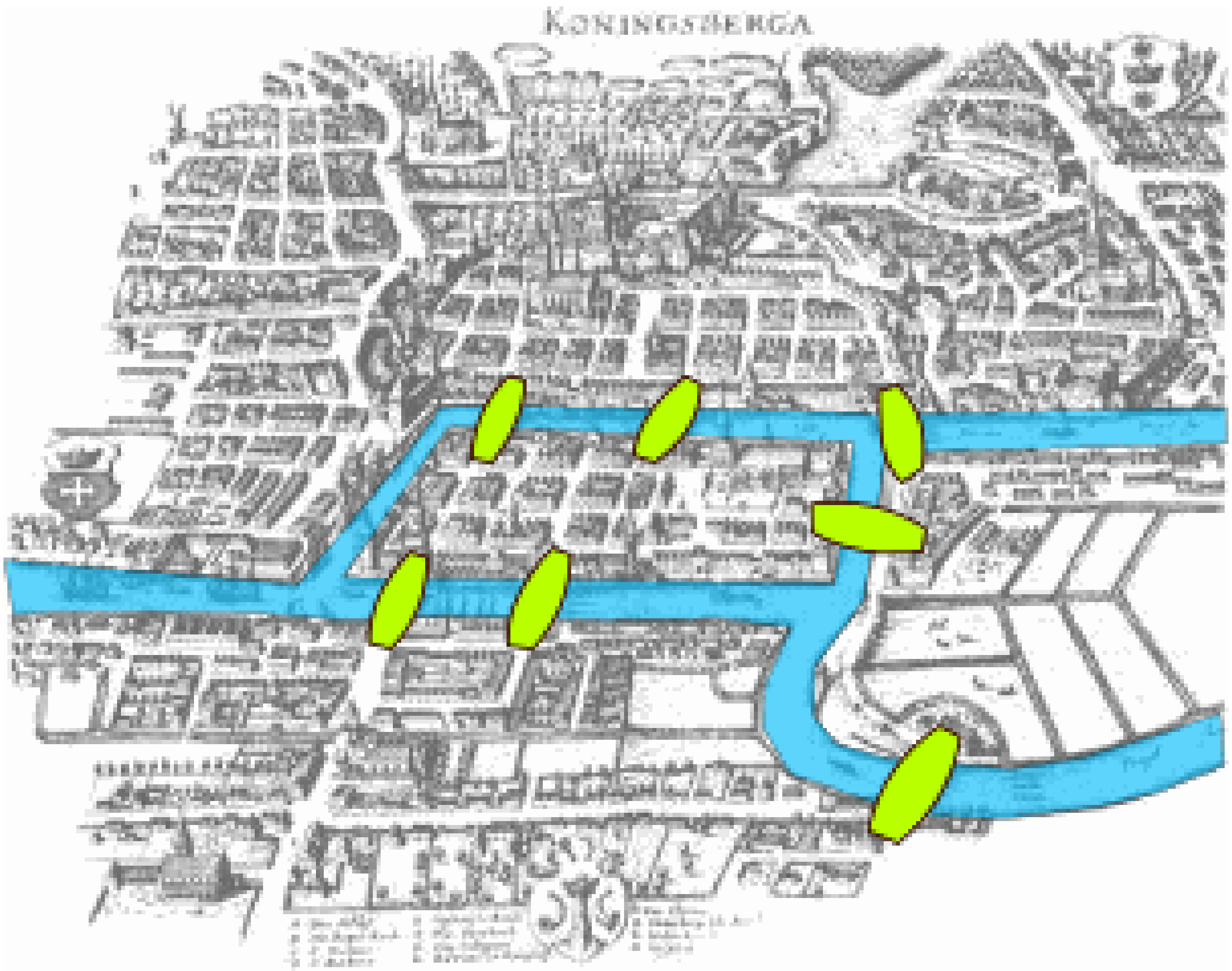
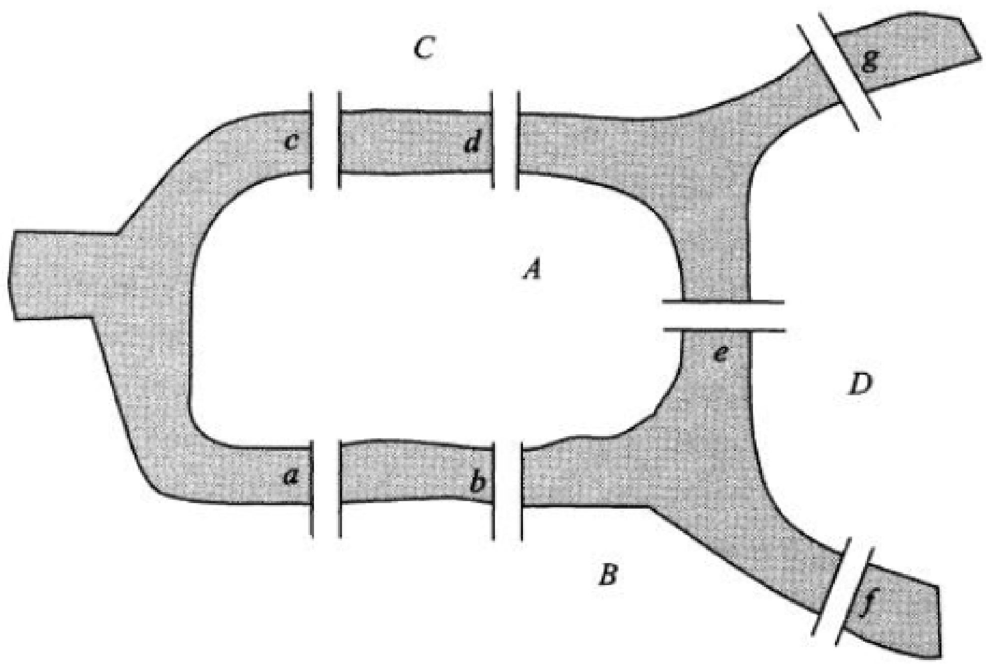
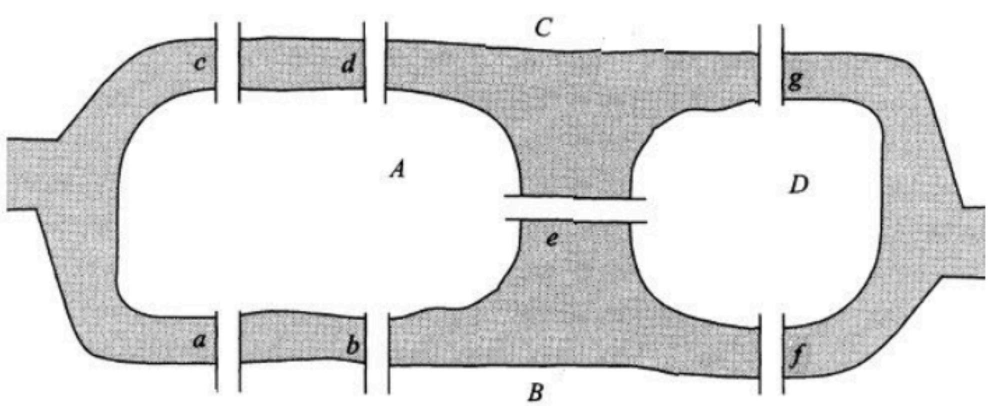
გრაფები

გრაფთა თეორიას საფუძველი ჩაეყარა მე-18 საუკუნეში, როდესაც ეილერმა (მათემატიკოსი) შეეცადა ამოეხსნა ამოცანა ქალაქ **კენიგსბერგის ხიდების** შესახებ. **კენიგსბერგი** მდებარეობას მდინარე **პრეგოლის** სანაპიროზე.





მეორე ვერსიის თანახმად

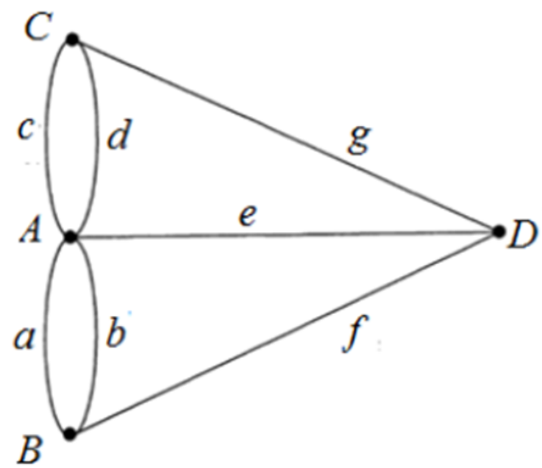


A, B, C, D - გამოსახავენ ქალაქის უბნებს, რომლებიც განლაგებულია მიდნარის ნაპირებზე და კუნძულებს, ხოლო a, b, c, d, e, f, g — ხიდებია ამ მდინარეზე.

ამოცანა უნდა დავიწყოთ ქალაქის დათვალიერება გარკვეული პუნქტიდან, გავიაროთ ყოველ ხიდზე მხოლოდ ერთხელ და დავბუნდეთ საწყის პუნქტში.

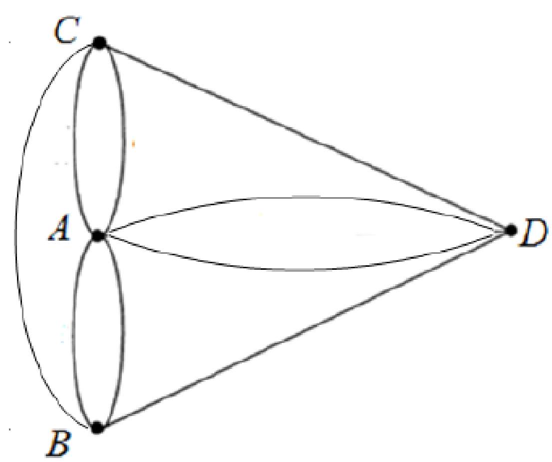
ეილერმა კენიგსბერგის წარმოადგინა გრაფის სახით, რომლის წვეროები ქალაქის უბნებს გამოსახავენ, ხოლო წიბოები ხიდებს, რომლებიც აკავშირებენ ქალაქის სხვადასხვა უბნებს ერთმანეთთან. ეილერმა დაამტკიცა რომ ზემოთ დასმულ ამოცანას ამონახსნი არ გააჩნია, ე.ი ასეთი მარშუტი არ არსებობს.

კენინგსბერგის ხიდების მათემატიკურ მოდელში გვაქვს:



$$\delta(A) = 5, \ \delta(B) = \delta(C) = \delta(D) = 3$$

თუ კენინგსბერგის ხიდებს დავუმატებთ კიდევ 2 ხიდს: შევაერთებთ A და D წვეროებს და B და C წვეროებს, მაშინ მივიღებთ ეილერის გრაფს.



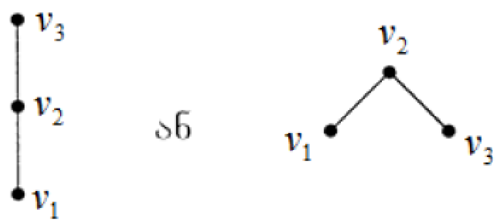
$$\delta(A) = 6, \ \delta(B) = \delta(C) = \delta(D) = 4$$

ძირითადი ცნებები და ტერმინოლოგია

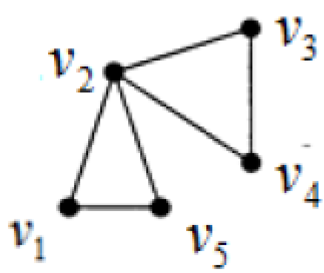
გრაფი წარმოადგენს სასრული V და E სიმარავლეების ერთობლიობას. V გრაფის წვეროების სიმრავლეა (vertex set), ხოლო E წიბოთა სიმარავლეს (edge set) - V სიმრავლის ორელემენტიანი ქვესიმრავლეების სიმრავლე. გრაფს აღნიშნავენ $G(V, E)$ სიმბოლოთი, v_1 და v_2 ელემენტებს უწოდებენ შეერთებულს $\{v_1, v_2\}$ წიბოთი, თუ $\{v_1, v_2\} \in E$.

ჩვეულებრი, გრაფს გამოსახავენ დიაგრამის საშუალებით, რომელშიც წვეროები აღნიშნულია წერტილებით, ხოლო ორი წერტილის შემაერთებელი წიბო - მონაკვეთით ან სხვა წირით.

მაგალითი 1: გრაფი, რომლის წვეროების სიმრაველეა $V\{v_1, v_2, v_3\}$, ხოლო წიბოების სიმრავლეა $E\{(v_1, v_2), (v_2, v_3)\}$ დიაგრამის საშუალებით გამოისახება:



მაგალითი 2: გრაფი, რომლის წვეროების სიმრავლეა $V\{v_1, v_2, v_3, v_4, v_5\}$, ხოლო წიბოების სიმრავლეა $E\{(v_1, v_2), (v_1, v_5), (v_2, v_5), (v_2, v_4)(v_2, v_3), (v_3, v_4)\}$ დიაგრამის საშუალებით გამოისახება:



- **მარყუჟი** - წიბო, რომლის ბოლოები ერთმანეთს ემთხვევა ე.ი {v,v} სახის წიბო.
- **მულტიგრაფი** - გრაფი, რომელშიც ერთი და იგივე წიბო რამდენჯერმე გვხვდება.
- **მარტივი გრაფი** - გრაფი, რომელიც არ არის მულტიგრაფი და არ შეიცავს მარყუჟს. ამიერიდან ვიგულისხმებთ, რომ გრაფი არის მარტივი, თუ სხვანაირად არ არის ნახსენები.
- **წვეროს ხარისხი** - რაიმე v წვეროს ხარისხი არის d თუ v წვეროდან გამოდის ზუსტად d ცალი წიბო.
- **მოძრაობა** - გრაფში მოძრაობა არის წვეროთა სასრული მიმდევრობა $(v_1, v_2, ..., v_n)$ რომლისთვისაც v_i წიბოთი უკავშირდება v_{i+1} მოძრაობის სიგრძე განსაზღვრულია მასში შემავალი წიბოების რაოდენობით (ანუ არის $n-1$ ტოლი).
- **გზა** - მოძრაობას, რომელშიც წვეროები არ მეორდება.
- **ციკლი** - ისეთი ჩაკეტილ მოძრაობას, რომლის მხოლოდ პირველი და ბოლო წვეროა იდენტური. მოძრაობა ჩაკეტილია, თუ მისი პირველი და ბოლო წვერო ერთმანეთს ემთხვევა.

განმარტება: თუ $\{u, v\}$ წარმოადგენს წიბოს, მაშინ u და v წვეროებს ეწოდებათ ამ წიბოების ბოლოები. $\{u, v\}$ წიბოს აგრეთვე უწოდებენ **ინციდენტურს** u და v წვეროებისადმი. ამბობენ პირიქითაც, რომ u და v წვეროები **ინციდენტურია** $\{u, v\}$ წიბოსი.

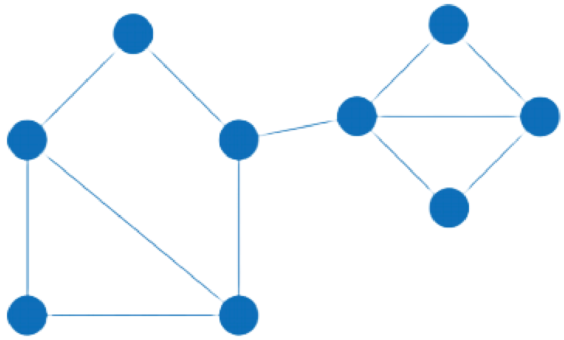
განმარტება: მანძილი ორ წვეროს შორის ეწოდება მინიმალური გზის სიგრძეს ამ წვეროებს შორის.

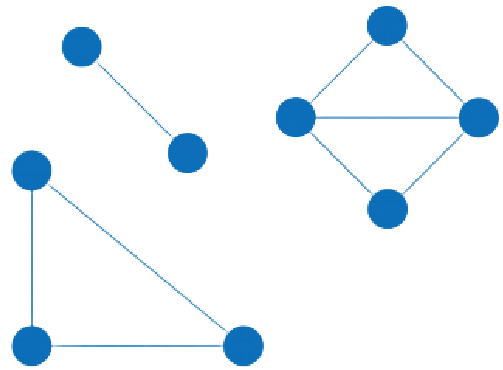
განმარტება: გრაფის ქვეგრაფი ეწოდება ისეთ გრაფს, რომელიც მიიღწევა საწყისი გრაფიდან რამდენიმე წვეროსა და წიბოს წაშლით.

განმარტება: $G(V, E)$ გრაფის **ქვეგრაფი** ეწოდება გრაფს $G'(V', E')$, თუ $V' \subseteq V$ და $E' \subseteq E$. ამრიგად G' გრაფის ყოველი წვერო წარმოადგენს G გრაფის წვეროსაც და G' გრაფის ყოველი წიბო წარმოადგენს G გრაფის წიბოსაც.

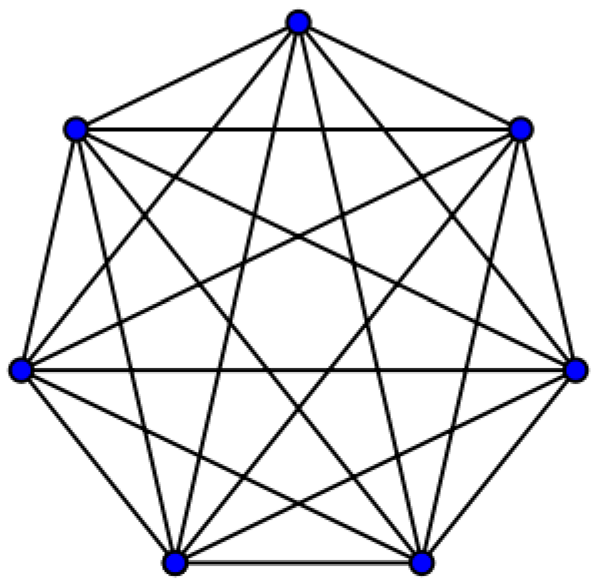
განმარტება: გრაფს ეწოდება ბმული თუ ნებისმიერ ორ წვეროს შორის არსებობს რაიმე გზა.

განმარტება: გრაფის კომპონენტი ეწოდება ბმულ ქვეგრაფს, რომელიც არ არის ნაწილი უფრო დიდი ბმული ქვეგრაფის.





განმარტება: n წვეროაინი სრული გრაფი (ხშირად აღინიშნება, როგორც Kn), ეწოდება ისეთ გრაფს, რომელშიც ნებისმიერი ორი წვერო დაკავშირებულია წიბოთი. ქვემოთ მოცემულია K7



გრაფთა წარმოდგენა

პროგრამირებაში არსებობს $G = (V, E)$ გრაფის წარმოდგენის ორი სტანდარტული მეთოდი:

ა) მოსაზღვრე წვეროების სიათა (adjancency-list representation) ჩამონათვალით;

```
#include <iostream>
#include <vector>

using namespace std;

void addEdge(vector<int> adj[], int u, int v) {
    adj[u].push_back(v);
    adj[v].push_back(u);
}

int main()
{
    int n, m, u, v;

    cin >> n >> m;

    vector<int> adj[n + 1];

    // მოსაზღვრე წვეროების შევსება
    for (int i = 1; i <= m; i++) {
        cin >> u >> v;
        addEdge(adj, u, v);
    }

    // მონაცემების გამოტანა
    for (int i = 1; i < n + 1; i++) {
        cout << i << " - ";
        for (int j = 0; j < adj[i].size(); j++) {
            cout << adj[i][j] << " ";
        }
        cout << "\n";
    }

    return 0;
}
```

ბ) მოსაზღვრეობის მატრიცით (adjancency matrix).

გრაფის წვეროების $V\{v_1, v_2, ..., v_n\}$ სიმრავლეზე ამოვწეროთ ე.წ. **მიმართების მატრიცა** M, რომელიც გვიჩვენებს, თუ გრაფის რომელი წვერო რომელთან არის დაკავშირებული. V სიმრავლეში ელემენტების რაოდენობას ავღნიშნავთ $|v|$ და განვიხილოთ კვადრატული მატრიცა ზომით $|v| \times |v|$, რომლის a_{ij} ელემენტები განისაზღვრებიან შემდეგნაირად

$$a_{ij} = \begin{cases} 1, \text{ თუ } v_i v_j \in E \\ 0, \text{ თუ } v_i v_j \notin E \end{cases}$$

მიმართების სიმეტრიულობა ნიშნავს, რომ მიღებული მატრიცა სიმეტრიულია მთავარი დიაგონალის მიმართ.

```cpp
#include <iostream>

using namespace std;

int main()
{
    int n, m, u, v;

    cin >> n >> m;

    int adj[n + 1][n + 1] = { 0 };

    // მატრიცის შევსება
    for (int i = 0; i < m; i++) {
        cin >> u >> v;
        u, v;
        adj[u][v] = 1;
        adj[v][u] = 1;
    }

    // მონცამების გამოტანა
    for (int i = 1; i < n + 1; i++) {
        for (int j = 1; j < n + 1; j++) {
            cout << adj[i][j] << " ";
        }
        cout << "\n";
    }

    return 0;
}
```

მაგალითი: მოცემულია გრაფი , რომლის წვეროების სიმრაველეა $V\{v_1, v_2, v_3, v_4, v_5\}$, ხოლო წიბოების სიმრავლეა $E\{(v_1, v_2), (v_1, v_5), (v_2, v_5), (v_2, v_4)(v_2, v_3), (v_3, v_4)\}$.

1. ამოვწეროთ წვეროების სია
2. ამოვწეროთ ამ მიმართების მატრიცა.

შესატანი მონაცემებიპირველი სტრიქონში პირველი ელემენტი არის წვეროების რაოდენობა, მეორე წევრი არის წიბოების რაოდენობა, დანარჩენი სტრიქონებში შემოდის მეზობელი წვეროები რომლებიც ერთამნეთან დაკავშირებული არიან რაიმე წიბოთი

5 6

1 2

1 5

2 5

2 4

2 3

3 4

გამოსატანი მონაცემი

მოსაზღვრე წვეროების სია

1 – 2 5

2 – 1 5 4 3

3 – 2 4

4 – 2 3

5 – 1

მიმართების მატრიცა

0 1 0 0 1

1 0 1 1 1

0 1 0 1 0

0 1 1 0 0

1 1 0 0 0

ხეები და მათი თვისებები

ერთ-ერთი ყველაზე მნიშვნელოვანი და გამოყენებადი გრაფის ფორმა არის ხე გრაფი.

ქვემოთ ჩამოთვლილი ნებისმიერი ორი ფაქტი ერთმანეთისაგან გამომდინარეობს, ამიტომაც ნებისმიერი მათგანი შეიძლება მიღებული იქნას, როგორც ხის განმარტება.

● გრაფი G ბმულია და აციკლური (ანუ ისეთი გრაფი, რომელსაც ციკლი არ აქვს).

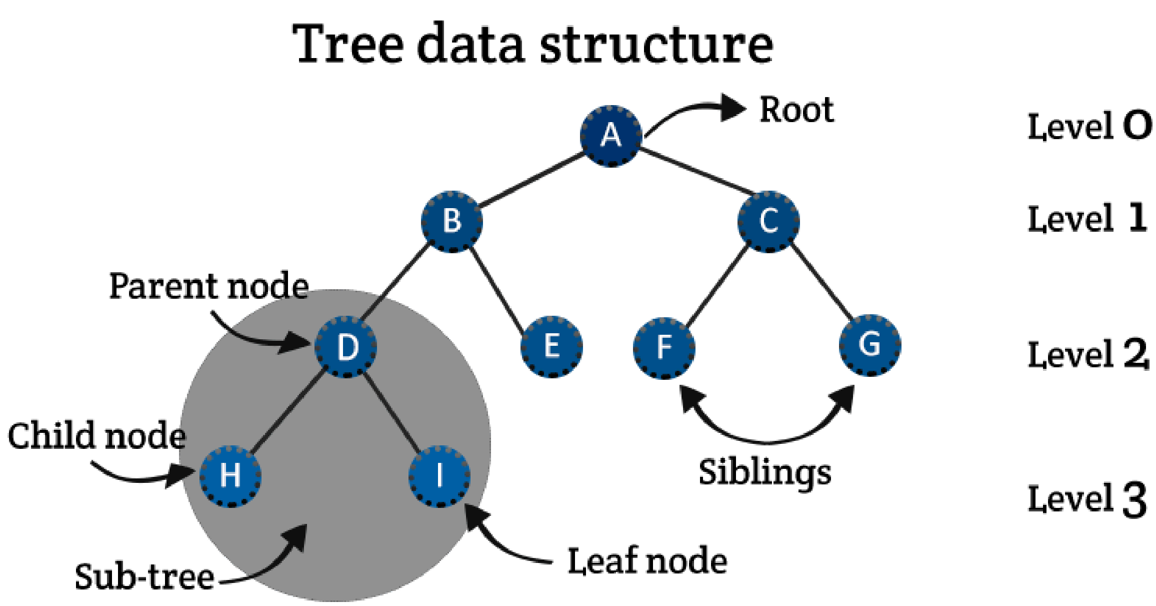
● გრაფი G აციკლურია და მარტივი ციკლი იქმნება ნემისმიერ ორ წვეროს შორის წიბოს დამატებით.

● გრაფი G ბმულია, თუმცა ნებისმიერი წიბოს წაშლა გრაფს ორ კომპონენტად დაყოფს.

● ნებისმიერ ორ წვეროს შორის არსებობს ერთადერთი გზა.

● გრაფი G ბმულია და აქვს n წვერო და n−1 წიბო.

● გრაფი G აციკლურია და აქვს n−1 წიბო.



● ხშირად მომგებიანია ხის, როგორც ფესვიან გრაფის განხილვა. ხის ფესვი შეიძლება იყოს ნებისმიერი წვერო და ის წარმოადგენს ყველაზე მაღალ წვეროს ხის გრაფის "იერარქიაში". ამ შემთხვევაში ფესვი იყოს A წვერო.

● v წვეროს სიღრმე ან დონე ეწოდება მანძილს ფესვიდან v წვერომდე.

● v წვეროს მშობელი ეწოდება ისეთ წვეროს, რომელიც არის v-ს მეზობელი და აქვს უფრო ნაკლები სიღრმე. იოლი მისახვედრია, რომ ყოველ წვეროს (გარდა ფესვისა) ჰყავს ზუსტად ერთი მშობელი.

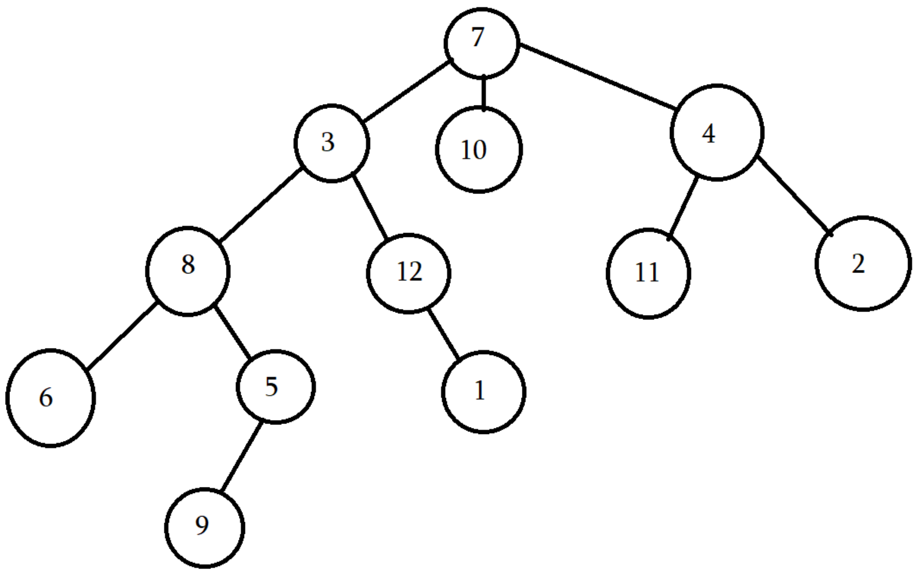
● v წვეროს შვილს ვუწოდებთ ნებისმიერ მეზობელ წვეროს გარდა მისი მშობლისა.

● ორი წვერო ერთმანეთის დედმამიშვილია თუ მათ ჰყავთ საერთო მშობელი.

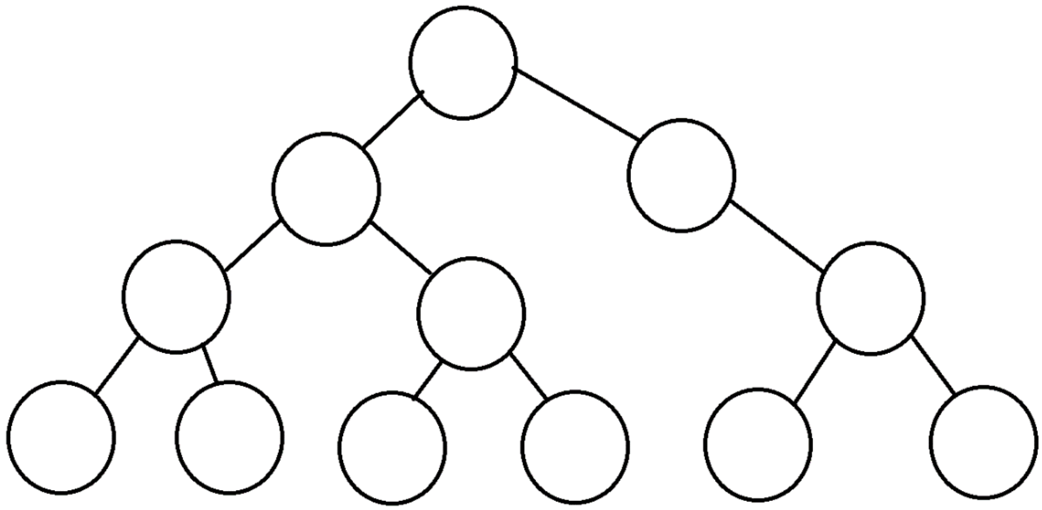
● ქვეხე ეწოდება ხის ისეთ ქვეგრაფს, რომელიც წარმოადგენს ხეს.

● ფოთოლი ქვია ისეთ წვეროს, რომლის ხარისხი ერთის ტოლია (წვეროებს რომლებიც მდებარეობს ხის ფუძეში (მათ არ ჰყავთ შვილები) უწოდებენ ფოთლებს.)

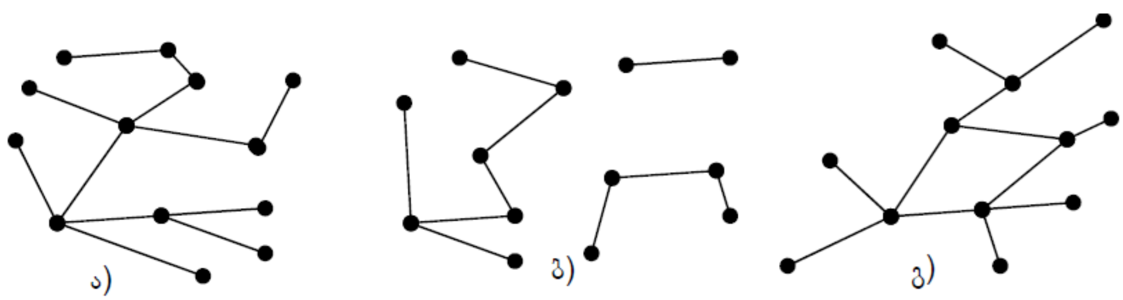
ხე ფესვით მიიღება თუ ხეში გამოვყოფთ რომელიმე წვეროს, რომელსაც ვუწოდებთ ფესვეს.



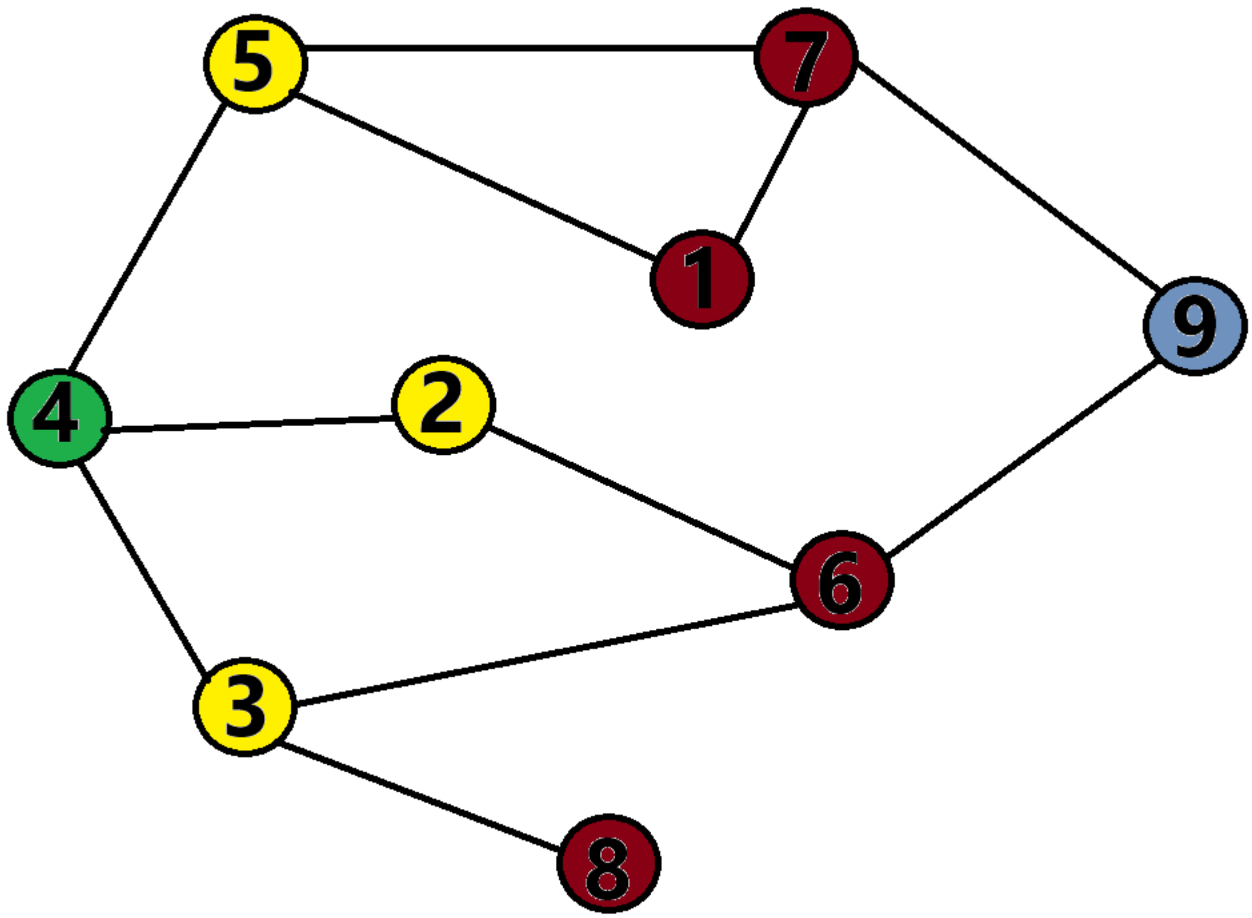
ბინარული ხეში ყოველ წვეროს ჰყავს მაქსიმუმ 2 შვილი. ბინარულ ხეში ყოველი წვეროდან ქვემოთ მიემართება არა უმეტეს ორი წიბოსი.



თუ ბინარული ხე არ შეიცავს წვეროებს, მაშინ მას ეწოდება **ცარიელი.** თუ მარცხენა ქვე-ხე ცარიელი არ არის, მაშინ მის ფესვს უწოდებენ მთლიანი ხის ფესვის მარცხენა შვილს (left child), ანალოგიურად მარჯვენა შვილი (right child).



BFS განივად ძებნის (breadth-first search) ალგორითმი



წვერო	1	2	3	4	5	6	7	8	9
მანძილი - d[n]	0	0	0	0	0	0	0	0	0
მშობელი - p[n]	-1	-1	-1	-1	-1	-1	-1	-1	
used[n]	0	0	0	1	0	0	0	0	0
რიგი FIFO	4								
წვერო	1	2	3	4	5	6	7	8	9
მანძილი - d[n]	0	1	1	0	1	0	0	0	0
მშობელი - p[n]	-1	4	4	-1	4	-1	-1	-1	-1
used[n]	0	1	1	1	1	0	0	0	0
რიგი FIFO	5	2	3						
წვერო	1	2	3	4	5	6	7	8	9
მანძილი - d[n]	2	1	1	0	1	0	2	0	0

მშობელი - p[n]	5	4	4	-1	4	-1	5	-1	-1
used[n]	1	1	1	1	1	0	1	0	0
რიგი FIFO	2	3	7	1					
წვერო	1	2	3	4	5	6	7	8	9
მანძილი - d[n]	2	1	1	0	1	2	2	0	0
მშობელი - p[n]	5	4	4	-1	4	2	5	-1	-1
used[n]	1	1	1	1	1	1	1	0	0
რიგი FIFO	3	7	1	6					
წვერო	1	2	3	4	5	6	7	8	9
მანძილი - d[n]	2	1	1	0	1	2	2	2	0
მშობელი - p[n]	5	4	4	-1	4	2	5	3	-1
used[n]	1	1	1	1	1	1	1	1	0
რიგი FIFO	7	1	6	8					

წვერო	1	2	3	4	5	6	7	8	9
მანძილი - d[n]	2	1	1	0	1	2	2	2	3
მშობელი - p[n]	5	4	4	-1	4	2	5	3	7
used[n]	1	1	1	1	1	1	1	1	1
რიგი FIFO	1	6	8	9					
წვერო	1	2	3	4	5	6	7	8	9
მანძილი - d[n]	2	1	1	0	1	2	2	2	3
მშობელი - p[n]	5	4	4	-1	4	2	5	3	7
used[n]	1	1	1	1	1	1	1	1	1
რიგი FIFO	6	8	9						
წვერო	1	2	3	4	5	6	7	8	9
მანძილი - d[n]	2	1	1	0	1	2	2	2	3

მშობელი - p[n]	5	4	4	-1	4	2	5	3	7
used[n]	1	1	1	1	1	1	1	1	1
რიგი FIFO	8	9							
წვერო	1	2	3	4	5	6	7	8	9
მანძილი - d[n]	2	1	1	0	1	2	2	2	3
მშობელი - p[n]	5	4	4	-1	4	2	5	3	7
used[n]	1	1	1	1	1	1	1	1	1
რიგი FIFO	9								
წვერო	1	2	3	4	5	6	7	8	9
მანძილი - d[n]	2	1	1	0	1	2	2	2	3
მშობელი - p[n]	5	4	4	-1	4	2	5	3	7
used[n]	1	1	1	1	1	1	1	1	1
რიგი FIFO									

```
#include <iostream>

#include <algorithm>

#include <vector>

#include <queue>


using namespace std;


void print(vector<int> adj[], int n) {

        for (int u = 1; u < n; u++) {

        cout << u << " - ";

        for (int v = 0; v < adj[u].size(); v++) {

        cout << adj[u][v] << " ";

        }

        cout << "\n";

        }

}


void bfs_path(int p[], int visited[], int to) {

        if (!visited[to]) cout << "No Path!";

        else {

        vector<int> path;

        for (int v = to; v != -1; v = p[v]) path.push_back(v);

        //for (int i = path.size()-1; i >= 0; i--) cout << path[i] << " ";

    reverse(path.begin(), path.end());

    for (int i = 0; i < path.size(); i++) cout << path[i] << " ";
```

```
        }
}

void bfs(vector<int> adj[], int n, int s) {

        // ნუმერაცია დაიწყება 1-დან, 0 ინდექს არ ვიყენებთ
        int v;
        int to;

        int d[n] = { 0 }; // მანძილი
        int p[n] = { -1 }; // მშობელი
        int visited[n] = { 0 }; // გამოყენებული წვეროები

        queue<int> q;
        q.push(s);

        visited[s] = 1;
        p[s] = -1;

        while (!q.empty()) {
        v = q.front();
        q.pop();
        for (int i = 0; i < adj[v].size(); i++) {
        to = adj[v][i];
        if (!visited[to]) {
```

```
        visited[to] = 1;
        q.push(to);
        d[to] = d[v] + 1;
        p[to] = v;
        }
        }
        }

        cout << "vertex" << "\t"; for (int i = 1; i < n; i++) cout << i << "\t"; cout << "\n";
        cout << "d" << "\t"; for (int i = 1; i < n; i++) cout << d[i] << "\t";  cout << "\n";
        cout << "p" << "\t"; for (int i = 1; i < n; i++) cout << p[i] << "\t";  cout << "\n";
        cout << "visited" << "\t"; for (int i = 1; i < n; i++) cout << visited[i] << "\t";  cout << "\n";

        to = 9;
        bfs_path(p, visited, to);
}

void addEdge(vector<int> adj[], int u, int v) {
        adj[u].push_back(v);
        adj[v].push_back(u);
}

int main() {

        int n; // წვეროების რაოდენობა
```

```
        int m; // წიბოების რაოდენობა
        int s; // საწყისი წვერო, წვეროების ნუმერაცია დაწყებულია 1-დან
        int u, v;

        cin >> n >> m >> s;

        vector<int> adj[n + 1]; // გრაფი

    for (int i = 0; i < m; i++) {
        cin >> u >> v;
        addEdge(adj, u, v);
        }

        print(adj, n + 1);
        cout << "\n";

        bfs(adj, n + 1, s);
}
```

```
print('BFS განივად ძებნის (breadth-first search) ალგორითმი')

 def _print(adj, n):
      for u in range(1, n):
      print(u, '-', end=' ')
      # print(adj[u])
      for v in range(0, len(adj[u])):
            print(adj[u][v], end=' ')
      print()

 def bsf_path(p, visited, to):
      if(visited[to]==0):
      print('No Path')
      path = []
      path.append(to)
      v = p[to]
      while v > -1:
      path.append(v)
      v = p[v]
      print(path[::-1])


 def bfs(adj, n, s):
      d = [0]*n
      p = [-1]*n
      visited = [0]*n

      #d[s] = 0
      #p[s] = -1
      visited[s] = 1

      q = []
      q.append(s)

      while len(q):
      print(q)
      v = q[0]
      q.pop(0)
      for i in range(0, len(adj[v])):
            to = adj[v][i]
            if(visited[to]==0):
            q.append(to)
            visited[to] = 1
                p[to] = v
            d[to] = d[v] + 1
      '''
      for i in range(1, n):
      print(i, end = '\t')
      print()
```

```
    for i in range(1, n):
    print(d[i], end = '\t')
    print()

    for i in range(1, n):
    print(p[i], end = '\t')
    print()

    for i in range(1, n):
    print(visited[i], end = '\t')
    print()
    '''
    to = 9
    bsf_path(p, visited, to)

def addEdge(adj, u, v):
    adj[u].append(v)
    adj[v].append(u)

txt = input().split()

n = int(txt[0])
m = int(txt[1])
s = int(txt[2])

adj = [0] * (n + 1)

for i in range(0, n + 1):
    adj[i] = list()

for i in range(0, m):
    txt = input().split()
    u = int(txt[0])
    v = int(txt[1])
    addEdge(adj, u, v)

# print(adj)
_print(adj, n+1)

print()

bfs(adj, n+1, s)
```

DFS სიღრმეში ძებნის (depth-first search) ალგორითმი

```cpp
// C++ program for DFS (Depth-First Search) of an undirected graph

#include <iostream>
#include <vector>

using namespace std;

void dfs(vector<int> adj[], int visited[], int s) {

	if (visited[s]) return;

	visited[s] = 1;

	for (int u = 0; u < adj[s].size(); u++) {
		dfs(adj, visited, adj[s][u]);
	}
}

void addEdge(vector<int> adj[], int u, int v) {
	adj[u].push_back(v);
	adj[v].push_back(u);
}
```

```
int main()

{

    int n, m, s;

    int u, v;


    cin >> n >> m >> s;


    vector<int> adj[n + 1];


    int visited[n + 1] = { 0 };


    for (int i = 1; i <= m; i++) {

    cin >> u >> v;

    addEdge(adj, u, v);

    }


    dfs(adj, visited, s);


    return 0;

}
```

```
/*
n m  s
9 11 4
4 5
4 2
4 3
5 7
5 1
2 6
3 6
3 8
1 7
7 9
6 9
*/
```

```
print('DFS სიღრმეში ძებნის (depth-first search) ალგორითმი')


def dfs(adj, visited, s):
  if visited[s]:
      return


  visited[s] = 1
```

```
  for i in range(0, len(adj[s])):
     dfs(adj, visited, adj[s][i])


def addEdge(adj, u, v):
   adj[u].append(v)
   adj[v].append(u)


txt = input().split()


n = int(txt[0])
m = int(txt[1])
s = int(txt[2])


adj = [0] * (n + 1)
visited = [0]*(n+1)


for i in range(0, n + 1):
   adj[i] = list()


for i in range(0, m):
   txt = input().split()
```

```
    u = int(txt[0])

    v = int(txt[1])

    addEdge(adj, u, v)



dfs(adj, visited, s)
```