ФАКУЛТЕТ ЗА ИНФОРМАТИЧКИ НАУКИ И КОМПЈУТЕРСКО ИНЖЕНЕРСТВО

# Графови - вовед

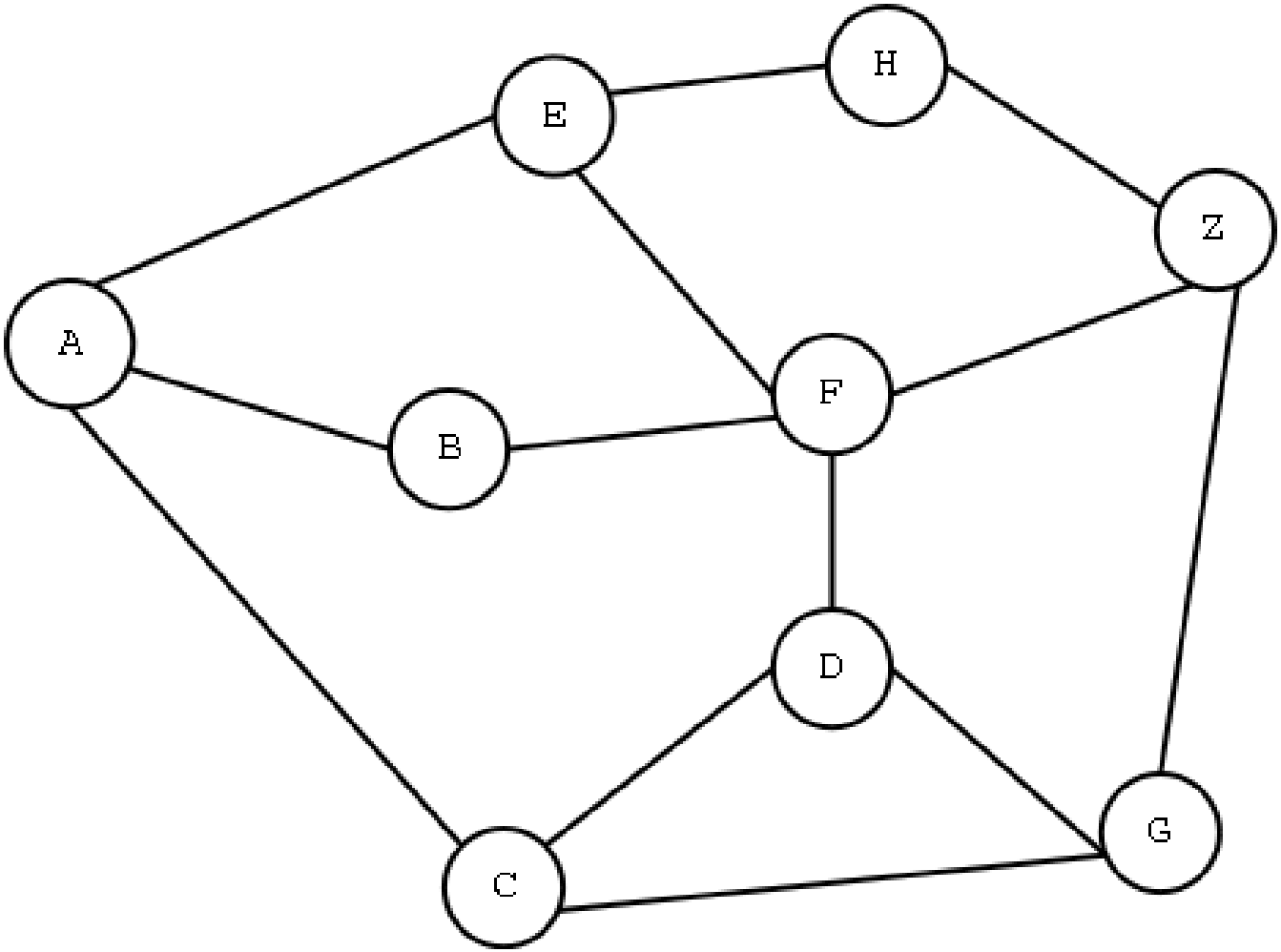
## Алгоритми и податочни структури

## Аудиториска вежба 10

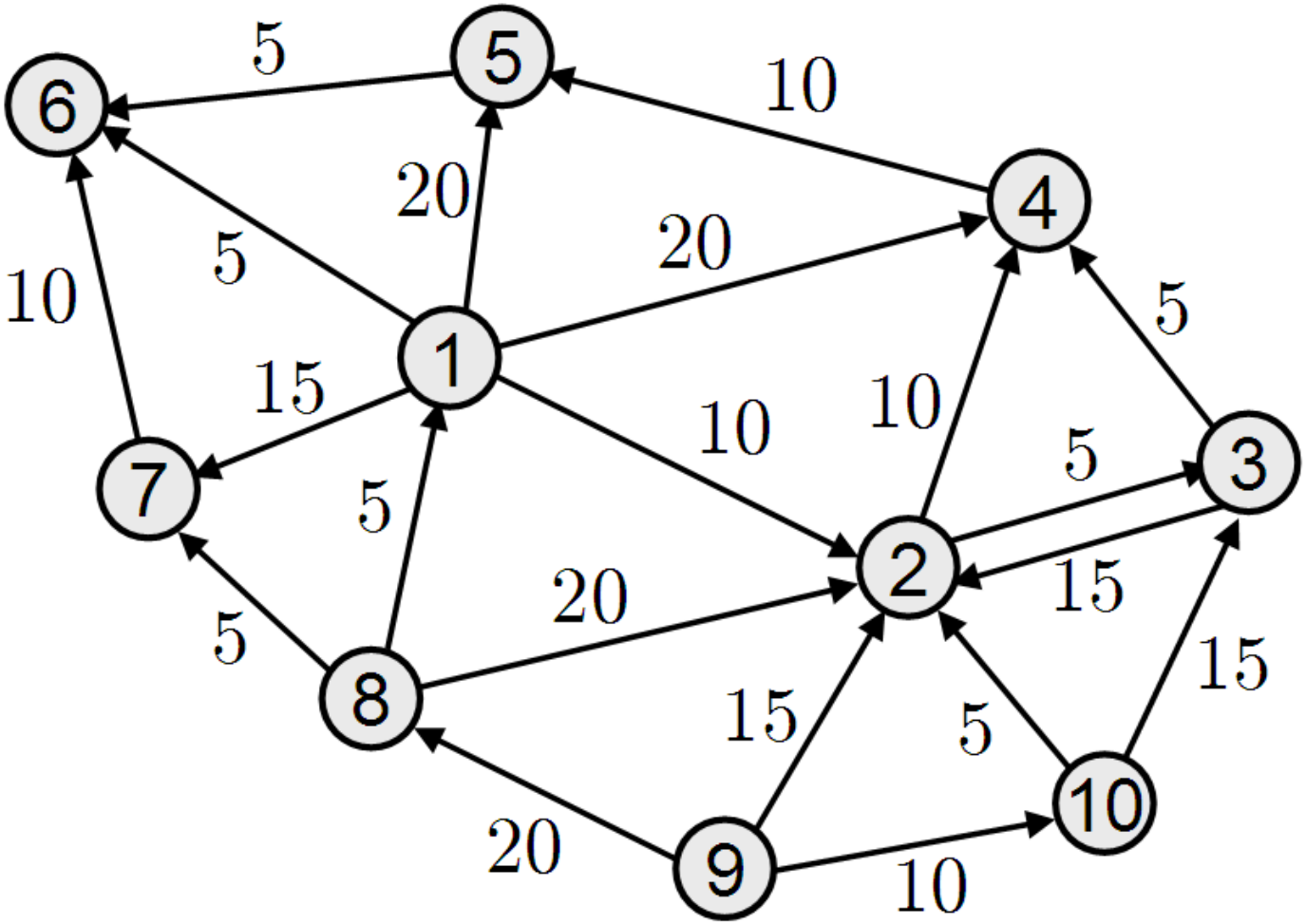
# Дефиниција

- Графовите претставуваат модел за апстрактна репрезентација на различни типови на мрежни системи како на пример транспортни системи, телекомуникациски мрежи или социјални мрежи.
- **Дефиниција**: Под граф $G = (V, E)$ се подразбираат двете множества $V$ кое е множество на јазли (темиња) и $E \subseteq (V \times V)$ кое е множество на ребра што поврзуваат две темиња во графот.

# Пример



# Пример



# Типови на графови

- *Ненасочен* (неориентиран) граф за кој важи (*u, v*) = (*v, u*) (ребрата не се насочени) и за сите *v*, (*v*, *v*) *E* т.е. не постојат јамки во графот
- *Насочен* (ориентиран) граф за кој важи дека (*u*, *v*) е ребро од *u* кон *v*, и се означува со *u* → *v* и притоа дозволени се јамки

- *Тежински* граф за кој важи дека на секое ребро е придружена тежина според некоја тежинска функција *w: E →R*

# Основни поими

- *Соседност на јазли*: за јазелот *v* се вели дека е соседен на јазелот *u* ако постои реброто (*v, u*)
- *Степен на јазел*: бројот на соседни јазли на даден јазел. Кај насоченият граф се разликуваат *влезен* и *излезен* степен на јазел, во зависност од насочеността на ребрата кон или од јазелот
- *Маршрута (route):* конечна низа од темиња така што две последователни темиња во маршрутата да се соседни
- *Верига*: маршрута во која ниту едно ребро не се повторува
- *Патека*: маршрута во која ниту едно теме не се повторува
- *Должина на патека*: бројот на ребра во патеката

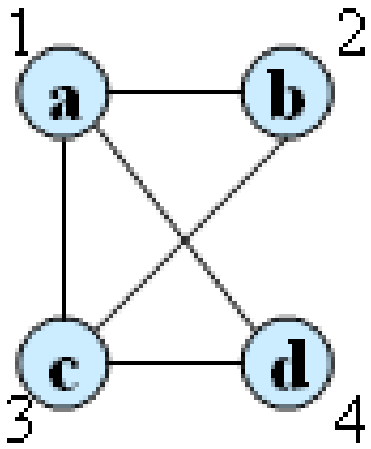
# Основни поими

- *Циклус (јамка)*: проста патека во која што првото и последното теме се исти
- *Поврзан граф*: граф кај кој постои постои патека помеѓу секој пар на јазли. Потребен (но не и доволен) услов за еден граф да е поврзан е $|E| \geq |V| - 1$.
- *Граф – дрво*: поврзан граф во кој постои единствена патека помеѓу секои два јазли. Притоа важи: $|E| = |V| - 1$.

# Репрезентација на графови

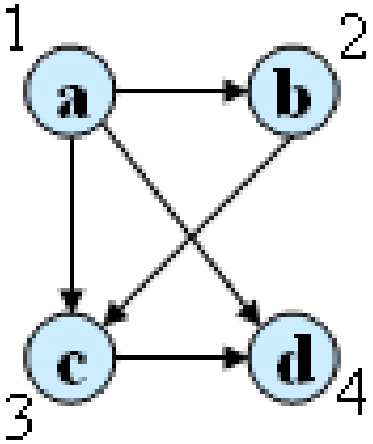
- Двете најчести репрезентации на графови се со помош на:
  - Матрица на соседство
  - Листа на соседство

# Матрица на соседство




ненасочен граф

| | a | b | c | d |
|---|---|---|---|---|
| a | 0 | 1 | 1 | 1 |
| b | 1 | 0 | 1 | 0 |
| c | 1 | 1 | 0 | 1 |
| d | 1 | 0 | 1 | 0 |




насочен граф

| | a | b | c | d |
|---|---|---|---|---|
| a | 0 | 1 | 1 | 1 |
| b | 0 | 0 | 1 | 0 |
| c | 0 | 0 | 0 | 1 |
| d | 0 | 0 | 0 | 0 |

# Матрица на соседство

- Ако графот има $|V|$ јазли, матрицата на соседство има димензија $|V|x|V|$. Колоните и редиците во оваа матрица ги претставуваат јазлите на графот.
- За ненасочен граф:

$$AdjM[i][j] = \begin{cases} 1, & akko postoi vrska(i,j) \\ 0, & inaku \end{cases}$$

- Матрицата на соседство за ненасочен граф е симетрична во однос на главната дијагонала.
- За насочен граф:

$$AdjM[i][j] = \begin{cases} 1, & akko postoi nasocena vrska(i,j) \\ 0, & inaku \end{cases}$$

- За тежински граф:

$$AdjM[i][j] = \begin{cases} c_{ij}, & akko postoi vrska(i,j) \\ \infty, & inaku \end{cases}$$

# Матрица на соседство - Java

```
import java.util.ArrayList;
import java.util.List;

public class AdjacencyMatrixGraph<T> {
    private int numVertices;
    private int[][] matrix;
    private T[] vertices;

    @SuppressWarnings("unchecked")
    public AdjacencyMatrixGraph(int numVertices) {
        this.numVertices = numVertices;
        matrix = new int[numVertices][numVertices];
        vertices = (T[]) new Object[numVertices];
    }

    public void addVertex(int index, T data) {
        vertices[index] = data;
    }

    public T getVertex(int index) {
        return vertices[index];
    }
```

# Матрица на соседство - Java

```
public void addEdge(int source, int destination) {
    matrix[source][destination] = 1;
    matrix[destination][source] = 1; // For undirected graph
}

public boolean isEdge(int source, int destination) {
    return matrix[source][destination] == 1;
}

public void removeEdge(int source, int destination) {
    matrix[source][destination] = 0;
    matrix[destination][source] = 0; // For undirected graph
}
```

# Матрица на соседство - Java

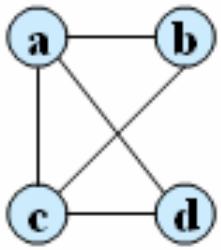
```
@SuppressWarnings("unchecked")
public void removeVertex(int vertexIndex) {
    if (vertexIndex < 0 || vertexIndex >= numVertices) {
    throw new IndexOutOfBoundsException("Vertex index out of bounds!");
    }
    int[][] newMatrix = new int[numVertices-1][numVertices-1];
    T[] newVertices = (T[]) new Object[numVertices-1];
    // Copy the vertices and matrix excluding the given vertex
    int ni = 0;
    for (int i = 0; i < numVertices; i++) {
        if (i == vertexIndex) continue;
        int nj = 0;
        for (int j = 0; j < numVertices; j++) {
            if (j == vertexIndex) continue;
            newMatrix[ni][nj] = matrix[i][j];
            nj++;}
        newVertices[ni] = vertices[i];
        ni++;}
    // Replace the old matrix and vertices with the new ones
    matrix = newMatrix;
    vertices = newVertices;
    numVertices--;}
```

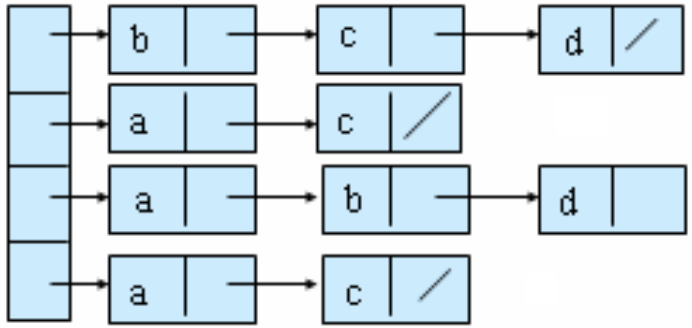
# Матрица на соседство - Java

```
    public List<T> getNeighbors(int vertexIndex) {
        List<T> neighbors = new ArrayList<>();
        for (int i = 0; i < matrix[vertexIndex].length; i++) {
            if (matrix[vertexIndex][i] == 1) {
                neighbors.add(vertices[i]);
            }
        }
    return neighbors;
    }
}
```

# Листа на соседство

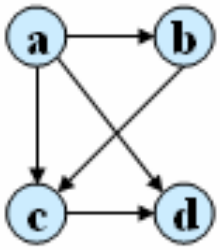
ненасочен граф

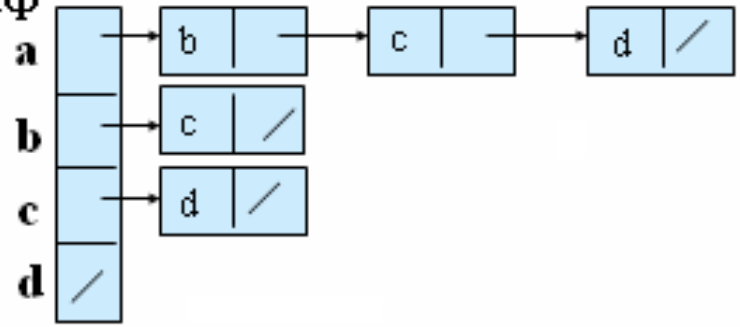



| Јазел | Листа на соседни јазли |
|---|---|
| a | b->c->d |
| b | a->c |
| c | a->b->d |
| d | a->c |

насочен граф




| Јазел | Листа на соседни јазли |
|---|---|
| a | b->c->d |
| b | c |
| c | d |
| d | |

# Листа на соседство - Java

```
import java.util.HashMap;
import java.util.HashSet;
import java.util.Set;
import java.util.Map;

public class AdjacencyListGraph<T> {
    private Map<T, Set<T>> adjacencyList;

    public AdjacencyListGraph() {
        this.adjacencyList = new HashMap<>();
    }

    // Add a vertex to the graph
    public void addVertex(T vertex) {
        if (!adjacencyList.containsKey(vertex)) {
            adjacencyList.put(vertex, new HashSet<>());
        }
    }

    // Remove a vertex from the graph
    public void removeVertex(T vertex) {
        // Remove the vertex from all adjacency lists
        for (Set<T> neighbors : adjacencyList.values()) {
            neighbors.remove(vertex);
        }
        // Remove the vertex's own entry in the adjacency list
        adjacencyList.remove(vertex);
    }
```

Set – колекција која ниту еден елемент не го содржи повеќе од еднаш (множество)
- операции за доадавање и бришење, но не и индексирање
- може да се изминат сите елементи, но не е гарантиран редоследот

# Листа на соседство - Java

```
    // Add an edge to the graph
    public void addEdge(T source, T destination) {
        addVertex(source);
        addVertex(destination);
        adjacencyList.get(source).add(destination);
        adjacencyList.get(destination).add(source); // for undirected graph
    }

    // Remove an edge from the graph
    public void removeEdge(T source, T destination) {
        if (adjacencyList.containsKey(source)) {
            adjacencyList.get(source).remove(destination);
        } if (adjacencyList.containsKey(destination)) {
            adjacencyList.get(destination).remove(source); // for undirected graph}
    }
    // Get all neighbors of a vertex
    public Set<T> getNeighbors(T vertex) {
        return adjacencyList.getOrDefault(vertex, new HashSet<>());
    }
}
```

# Задача 1

Да се напише функција којашто даден граф претставен со матрица на соседност ќе го даде истиот претставен со листа на соседи.

# Задача 1 - Решение

```
public AdjacencyListGraph<T> toAdjacencyList() {
    AdjacencyListGraph<T> result = new AdjacencyListGraph<>();

    for(int i=0;i<numVertices;i++) {
        result.addVertex(vertices[i]);
    }

    for(int i=0;i<numVertices;i++) {
        for(int j=0;j<numVertices;j++) {
            if(matrix[i][j] > 0) {
                result.addEdge(vertices[i], vertices[j]);
            }
        }
    }
    return result;
}
```

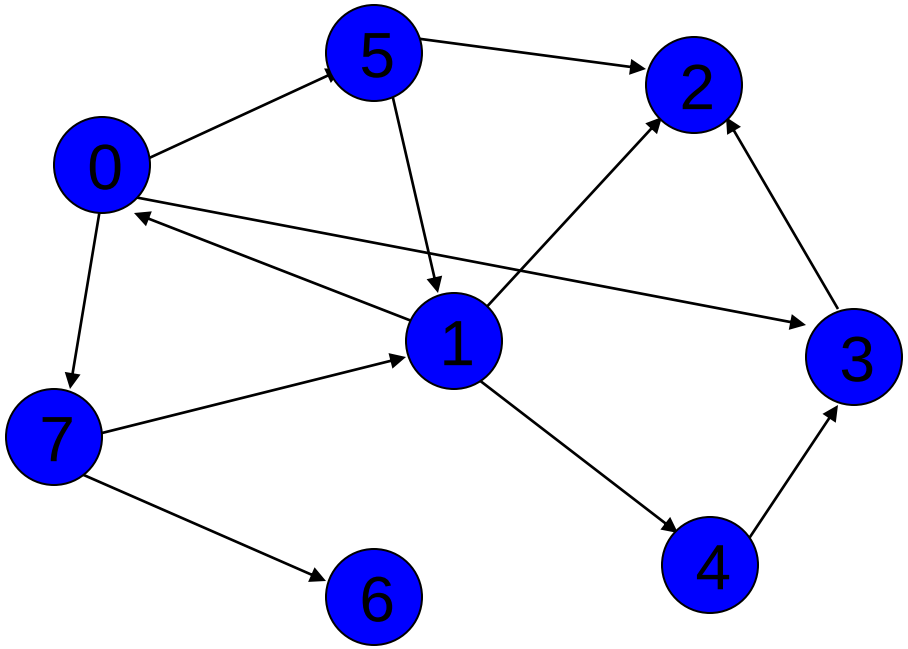
# Изминување на графови

- Изминување по длабочина (Depth – first search)
- Изминување по широчина (Breadth – first search)

# Изминување по длабочина

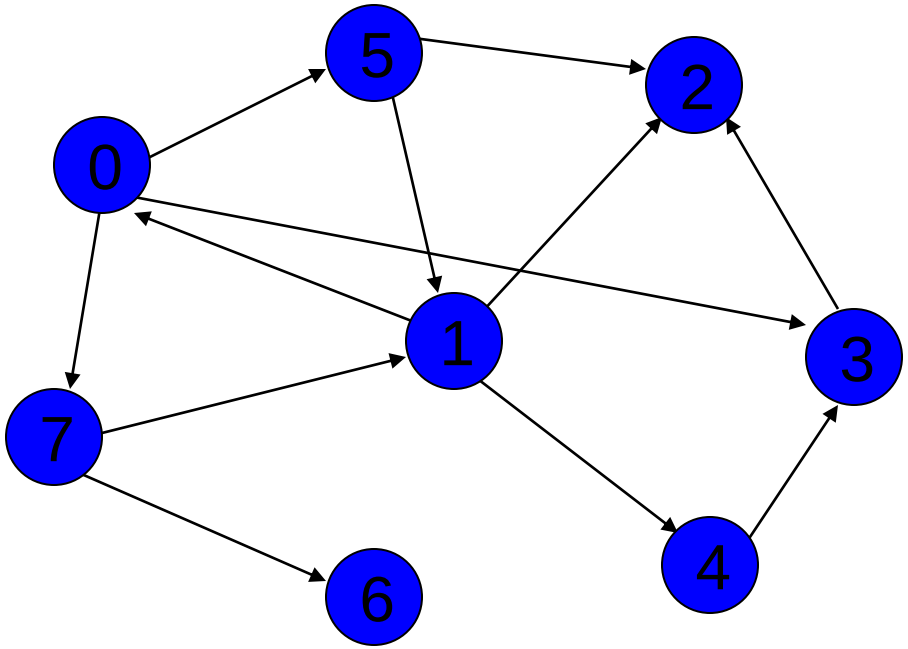
- Кај изминувањето по длабочина (depth-first search) во секој чекој се оддалечуваме што е можно повеќе и побрзо од почетниот јазел во графот.
- Нерекурзивна имплементација со стек

# Изминување по длабочина - илустрација



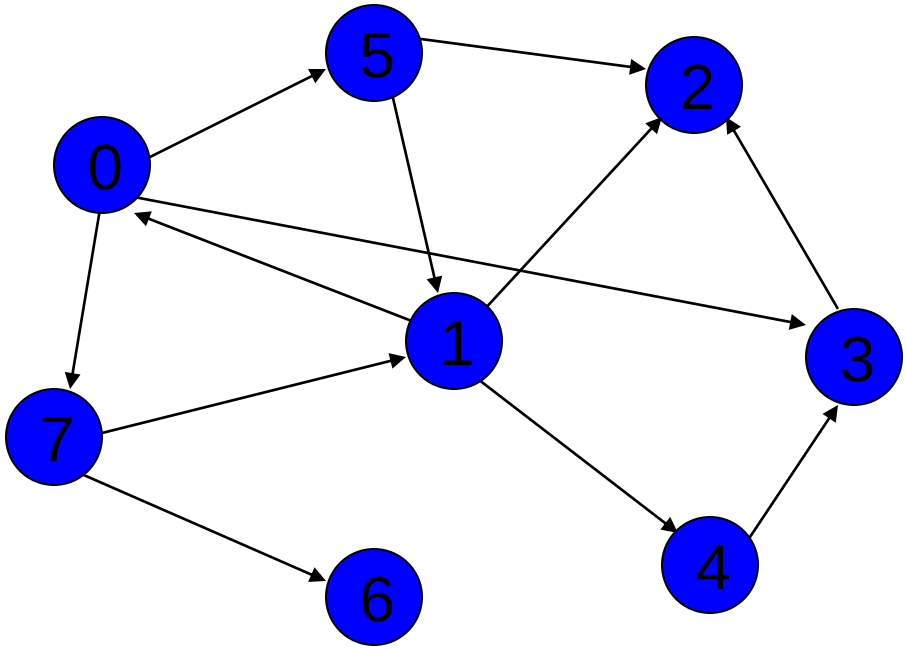
# Изминување по длабочина - илустрација

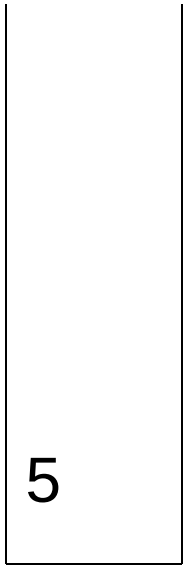
Започни со темето 5



# Изминување по длабочина - илустрација

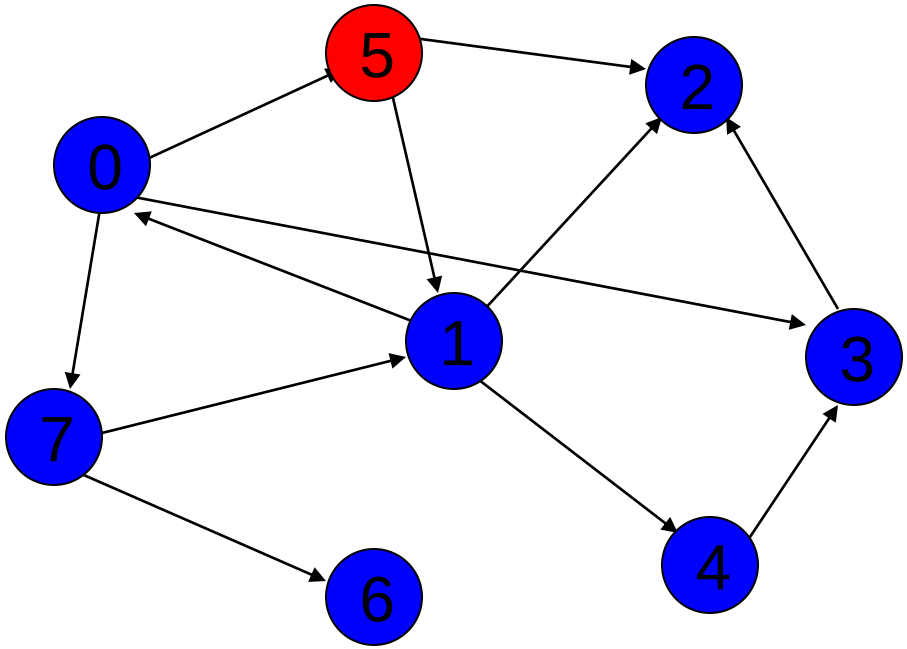
Започни со темето 5

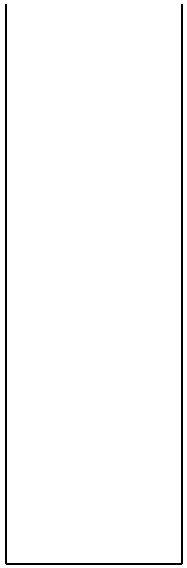



Push 5

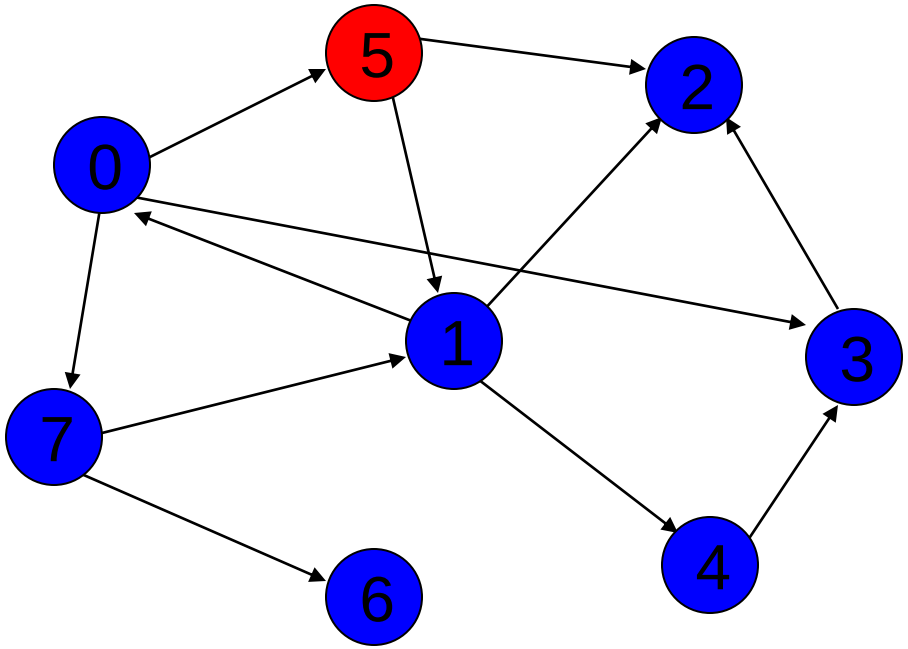
# Изминување по длабочина - илустрација

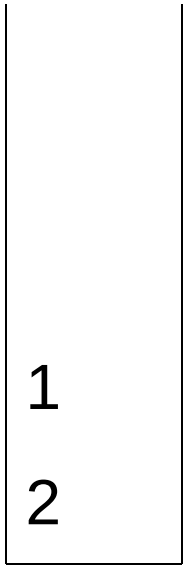


Pop/Visit/Mark 5

# Изминување по длабочина - илустрација

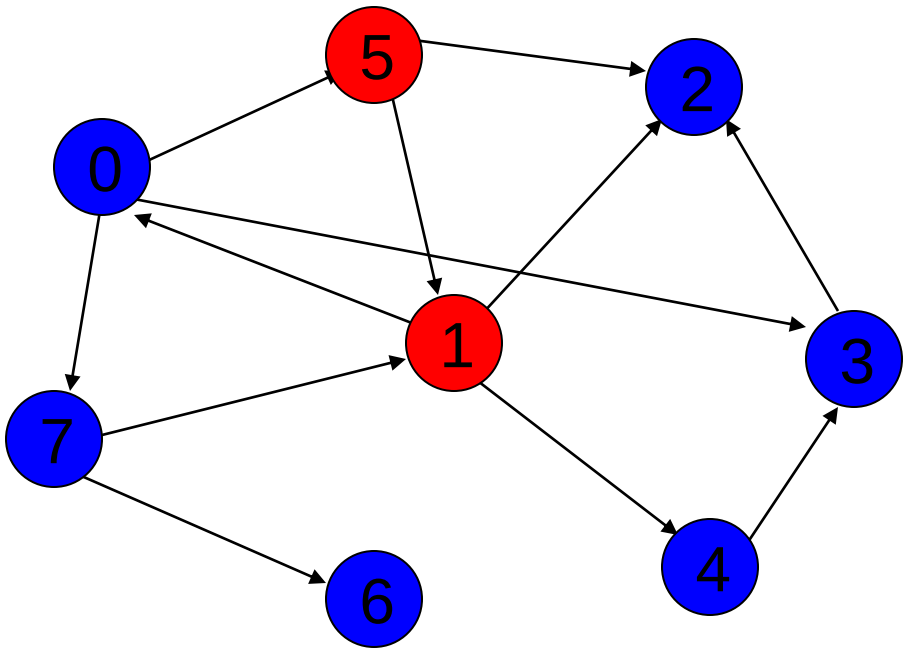


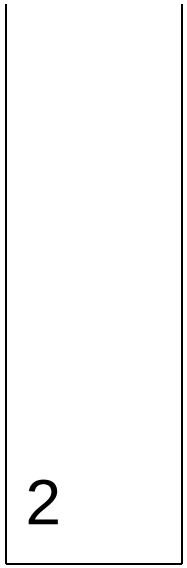




Push 2, Push 1

5

# Изминување по длабочина - илустрација

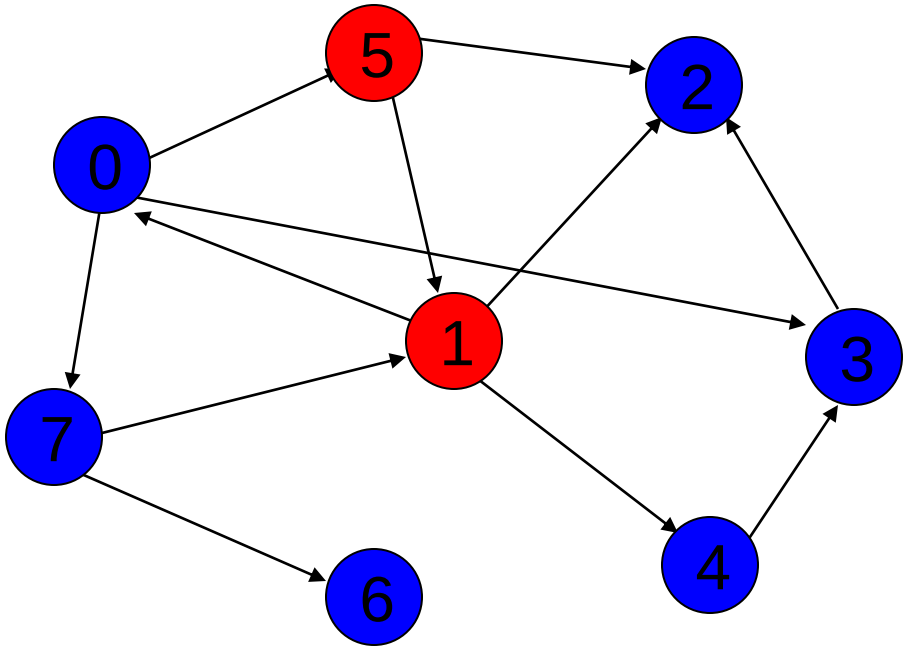



Pop/Visit/Mark 1

5 1

# Изминување по длабочина - илустрација

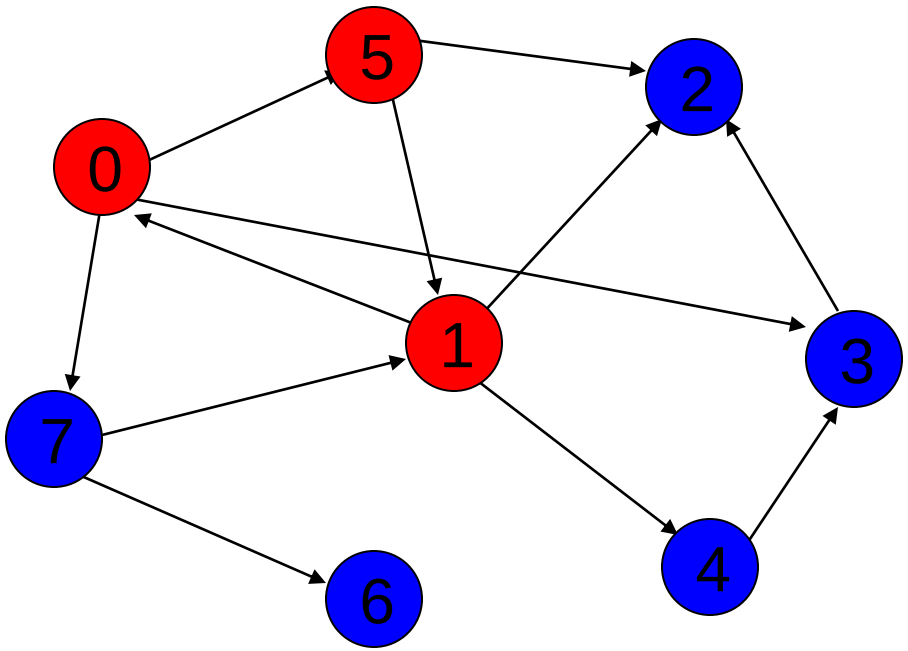



0
2
4
2

Push 4, Push 2,
Push 0

5 1

# Изминување по длабочина - илустрација

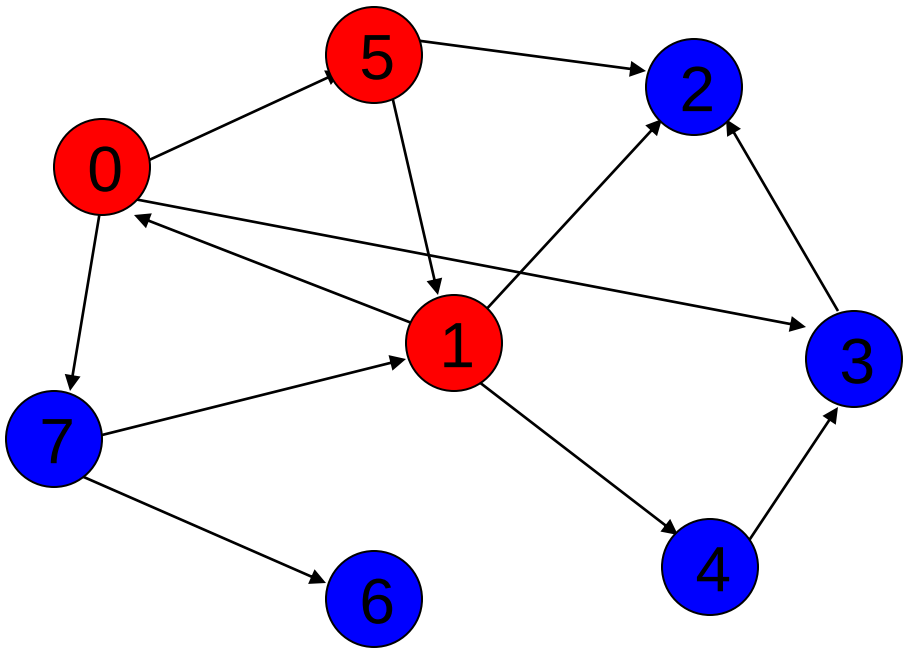



2
4
2

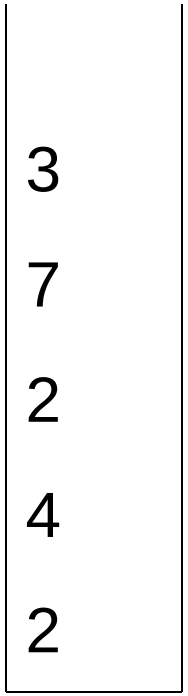
Pop/Visit/Mark 0

5 1 0

# Изминување по длабочина - илустрација

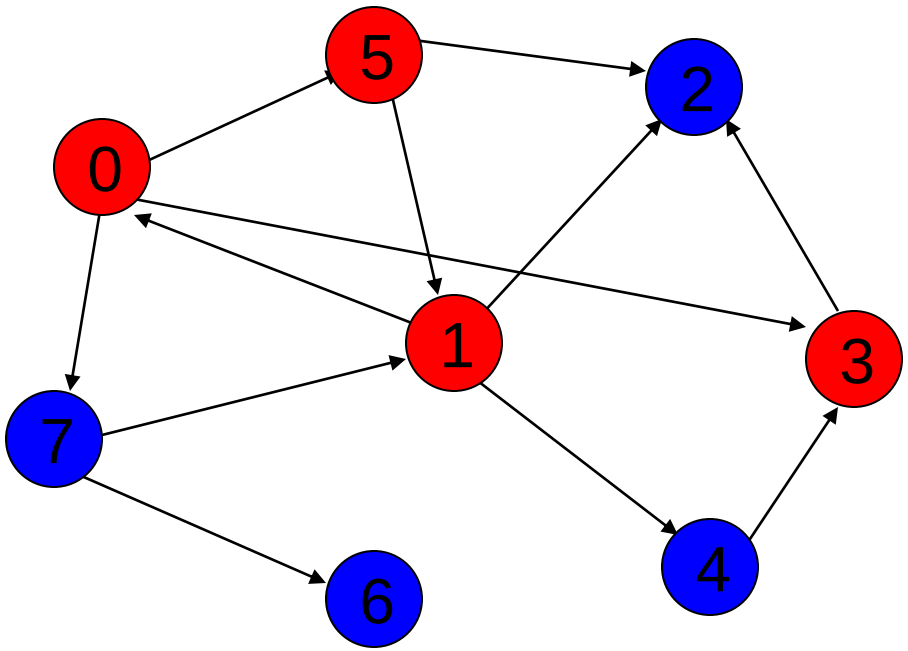


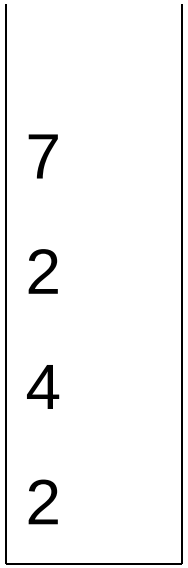

5 1 0




Push 7, Push5,
Push 3

# Изминување по длабочина - илустрација

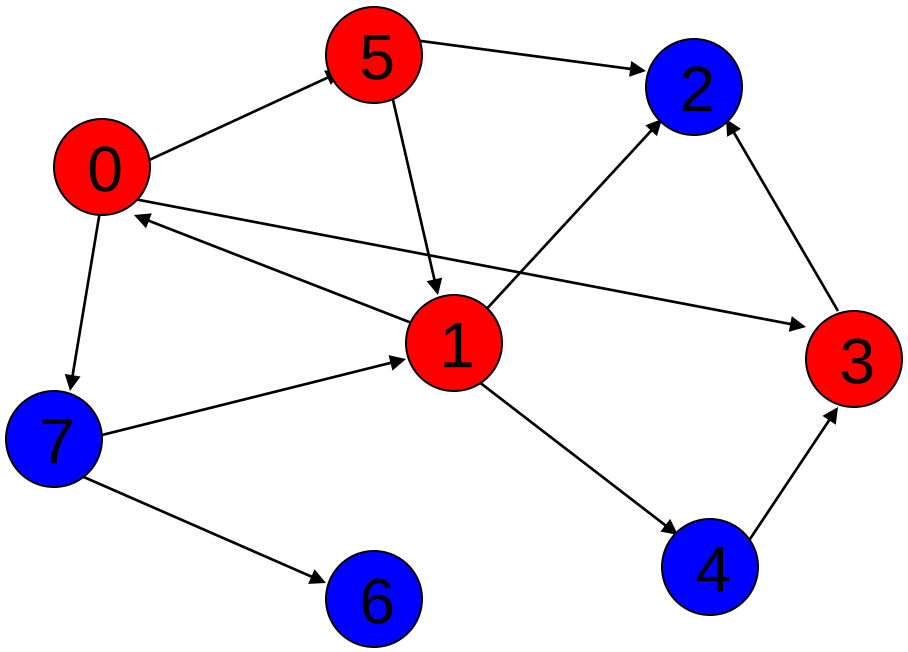


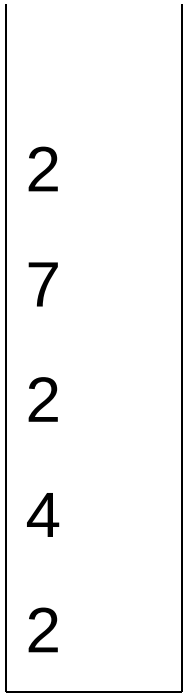




Pop/Visit/Mark 3

5 1 0 3

# Изминување по длабочина - илустрација

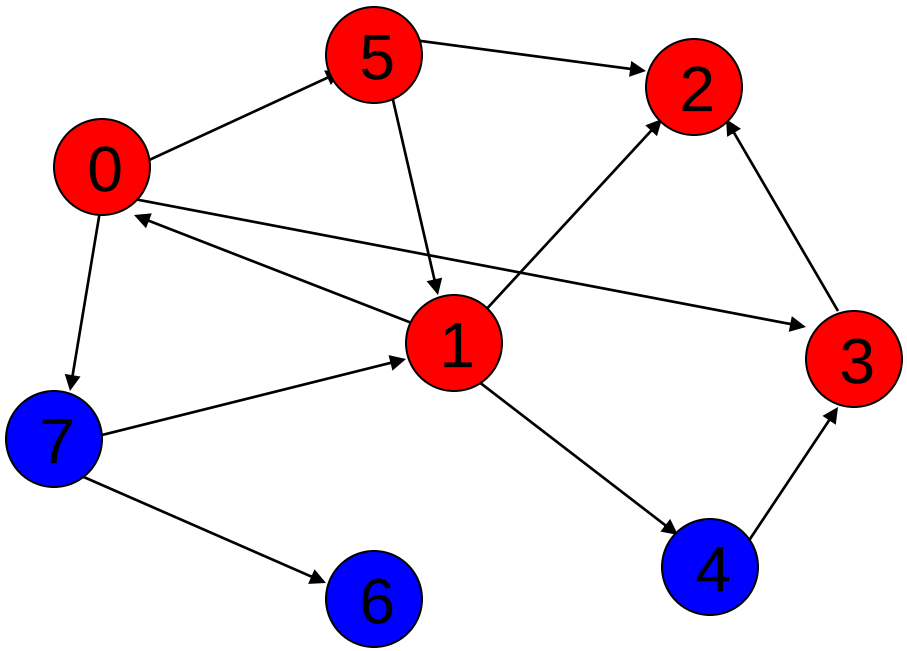


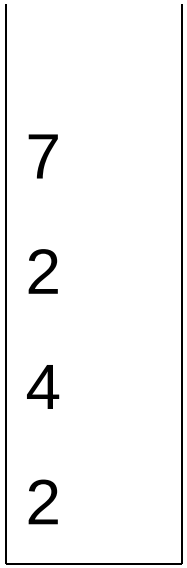




Push 2

5 1 0 3

# Изминување по длабочина - илустрација

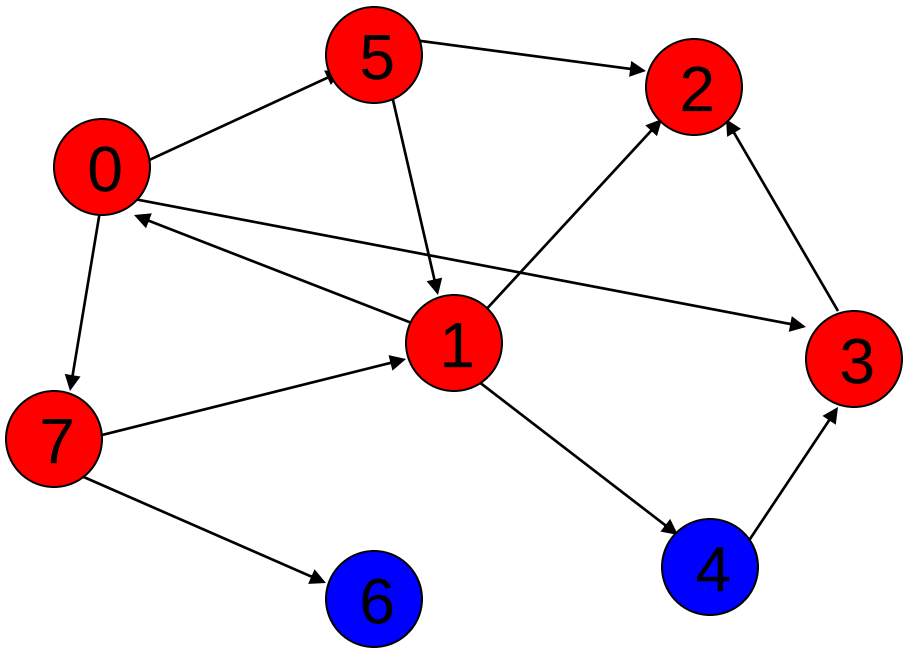


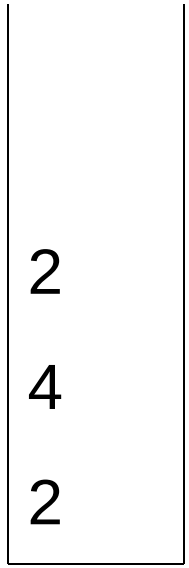




Pop/Mark/Visit 2

5 1 0 3 2

# Изминување по длабочина - илустрација

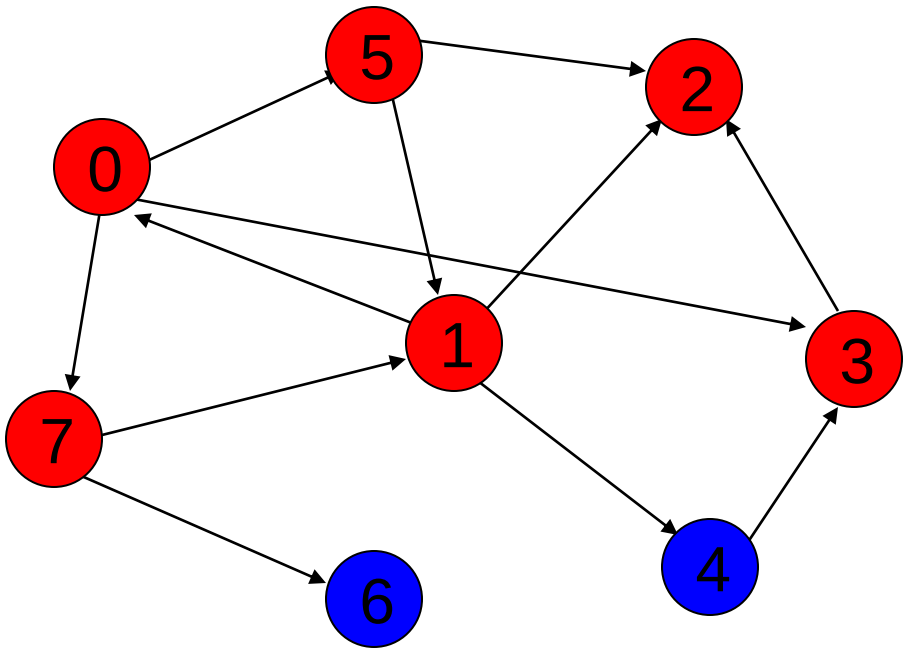


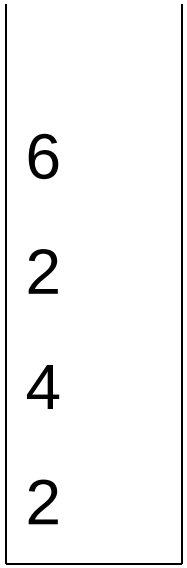




Pop/Mark/Visit 7

5 1 0 3 2 7

# Изминување по длабочина - илустрација

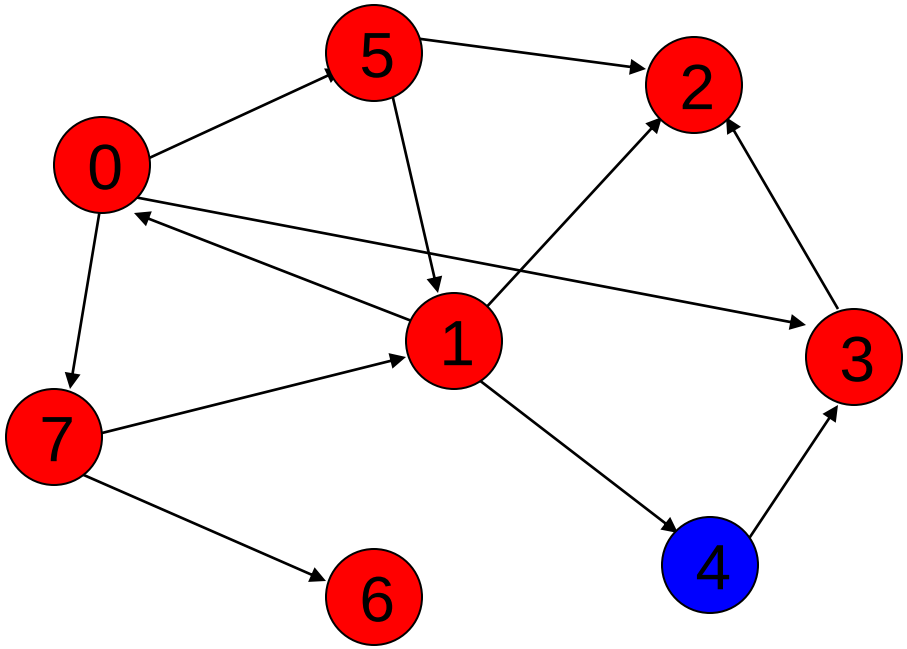


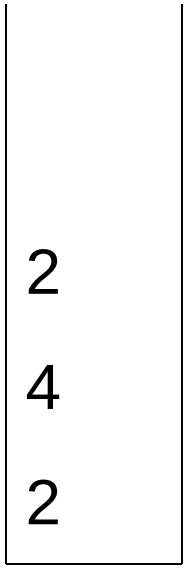




Push 6

5 1 0 3 2 7

# Изминување по длабочина - илустрација

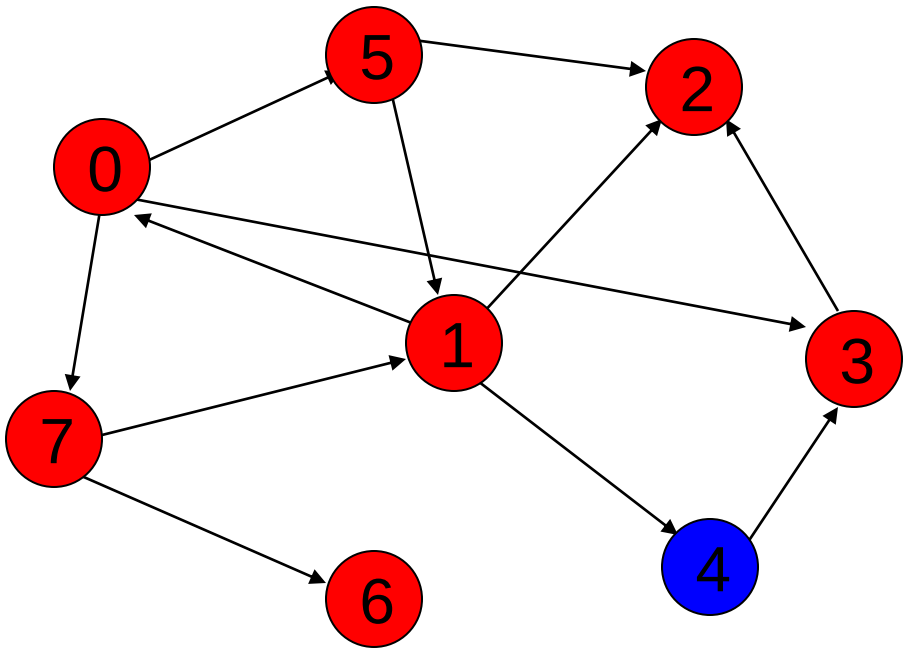


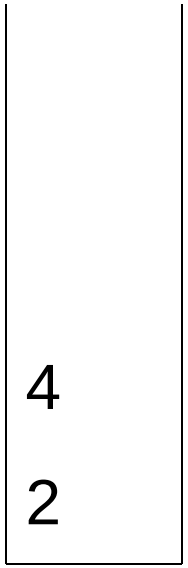




Pop/Mark/Visit 6

5 1 0 3 2 7 6

# Изминување по длабочина - илустрација

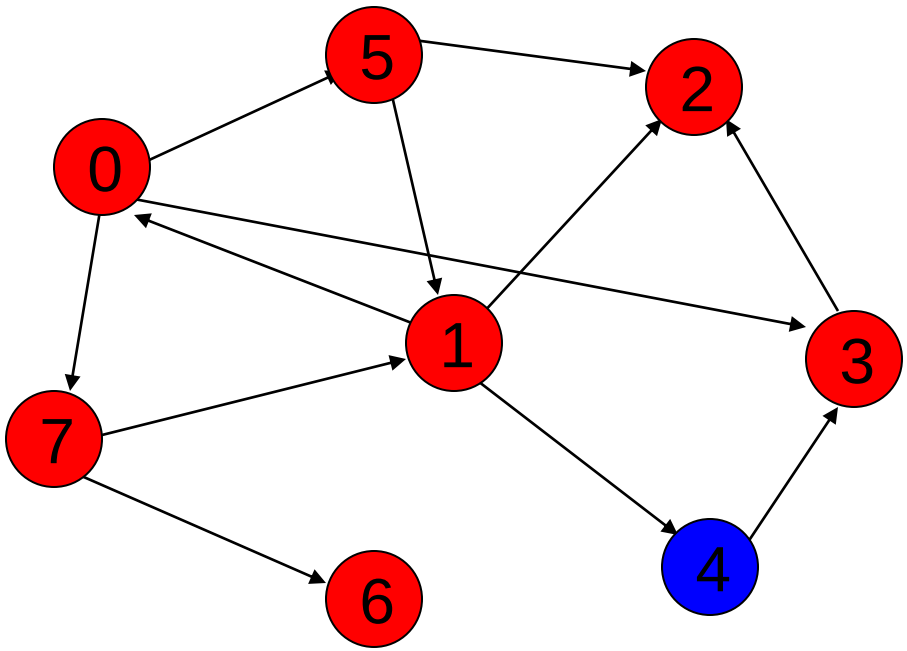


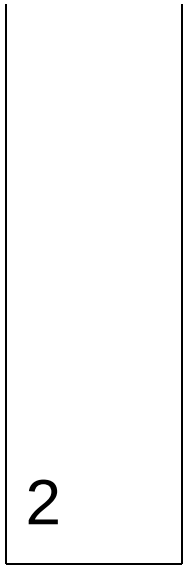




Pop (don’t visit) 2

5 1 0 3 2 7 6

# Изминување по длабочина - илустрација

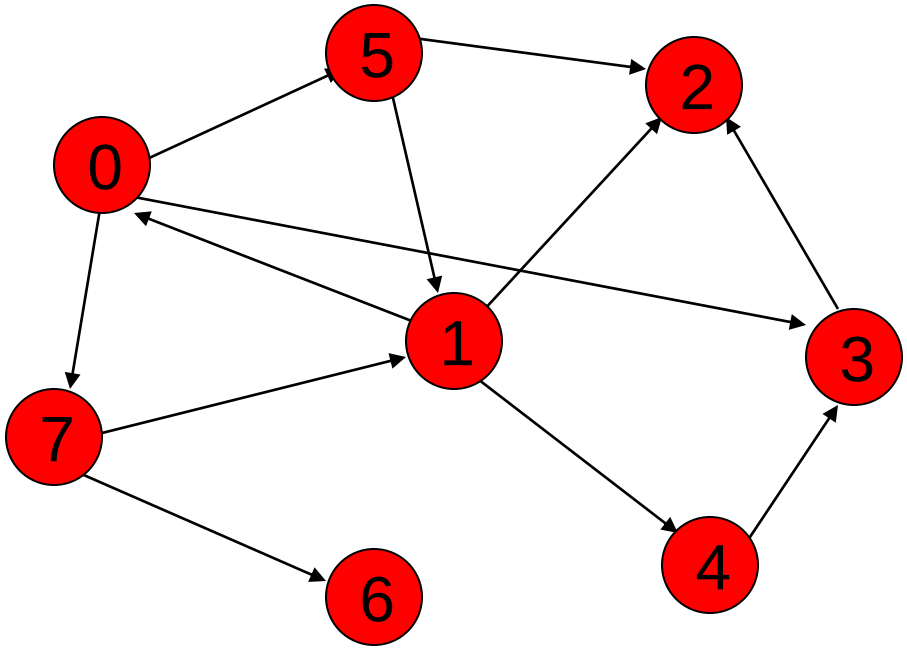


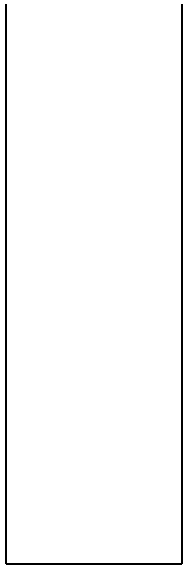




Pop/Mark/Visit 4

5 1 0 3 2 7 6 4

# Изминување по длабочина - илустрација

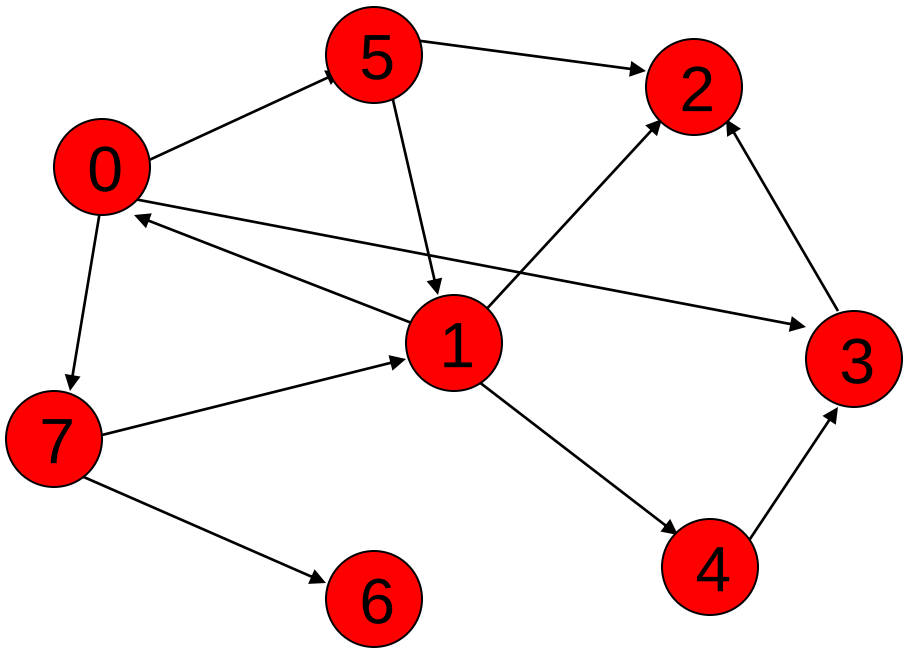


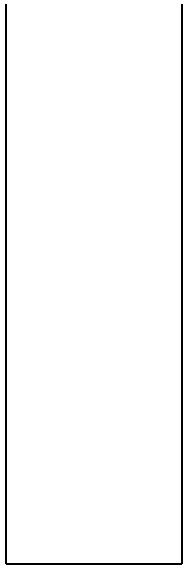



Pop (don’t visit) 2

5 1 0 3 2 7 6 4

# Изминување по длабочина - илустрација



Готово

5 1 0 3 2 7 6 4

# Изминување по длабочина - Java

- **Рекурзивно**

```
public void DFS(T startVertex) {
    Set<T> visited = new HashSet<>();
    DFSUtil(startVertex, visited);
    }

private void DFSUtil(T vertex, Set<T> visited) {
    // Mark the current node as visited and print it
    visited.add(vertex);
    System.out.print(vertex + " ");

    // Recur for all the vertices adjacent to this vertex
    for (T neighbor : getNeighbors(vertex)) {
        if (!visited.contains(neighbor)) {
            DFSUtil(neighbor, visited);
        }
    }
}
```

# Изминување по длабочина - Java

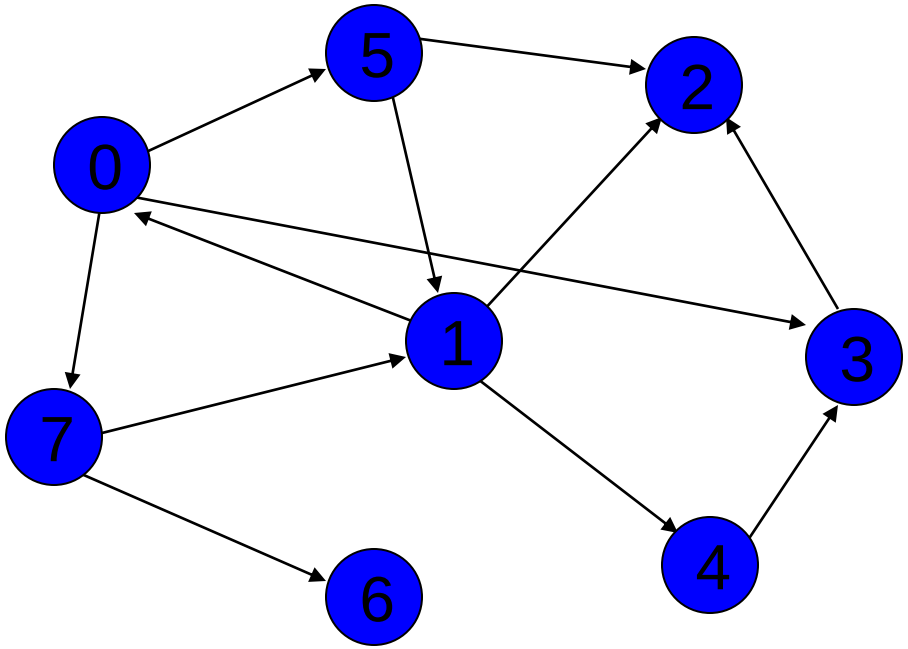
- **Нерекурзивно**

```
public void DFSNonRecursive(T startVertex) {
    Set<T> visited = new HashSet<>();
    Stack<T> stack = new Stack<>();

    stack.push(startVertex);
    while (!stack.isEmpty()) {
        T vertex = stack.pop();
        if (!visited.contains(vertex)) {
            visited.add(vertex);
            System.out.print(vertex + " ");
            for (T neighbor : getNeighbors(vertex)) {
                if (!visited.contains(neighbor)) {
                    stack.push(neighbor);
                }
            }
        }
    }
}
```

# Изминување по широчина

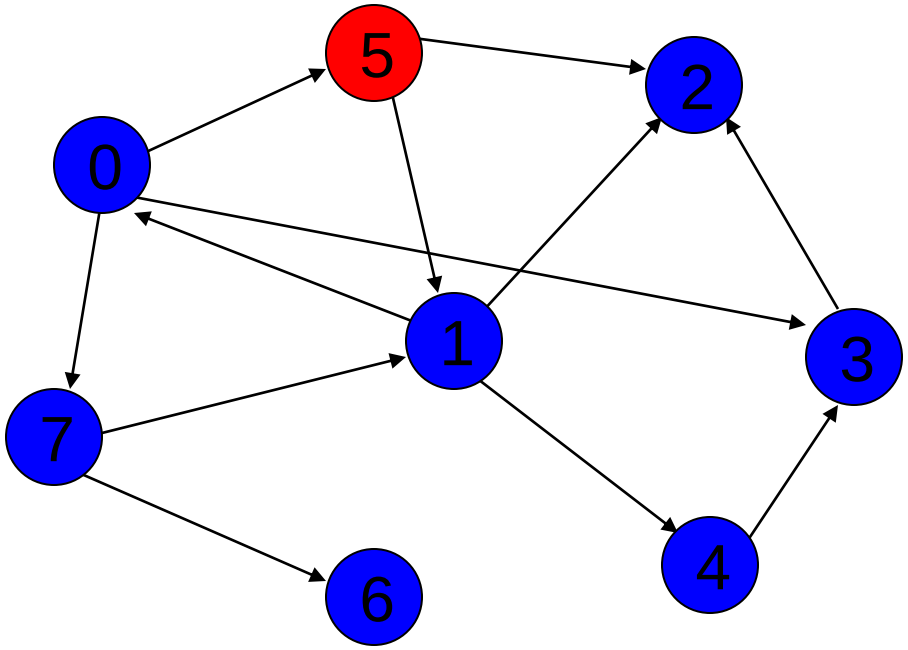
- Кај изминувањето по широчина (breath-first search) во секој чекој остануваме што е можно повеќе до почетниот јазел во графот.
- Нерекурзивна имплементација со редица

# Изминување по широчина - илустрација



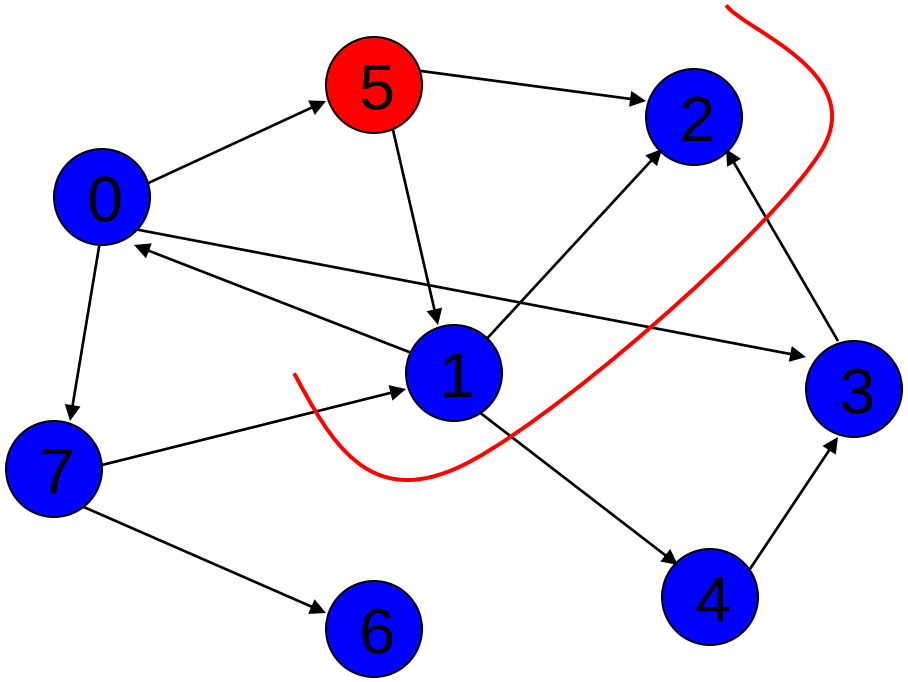
# Изминување по широчина - илустрација

Започни со темето 5



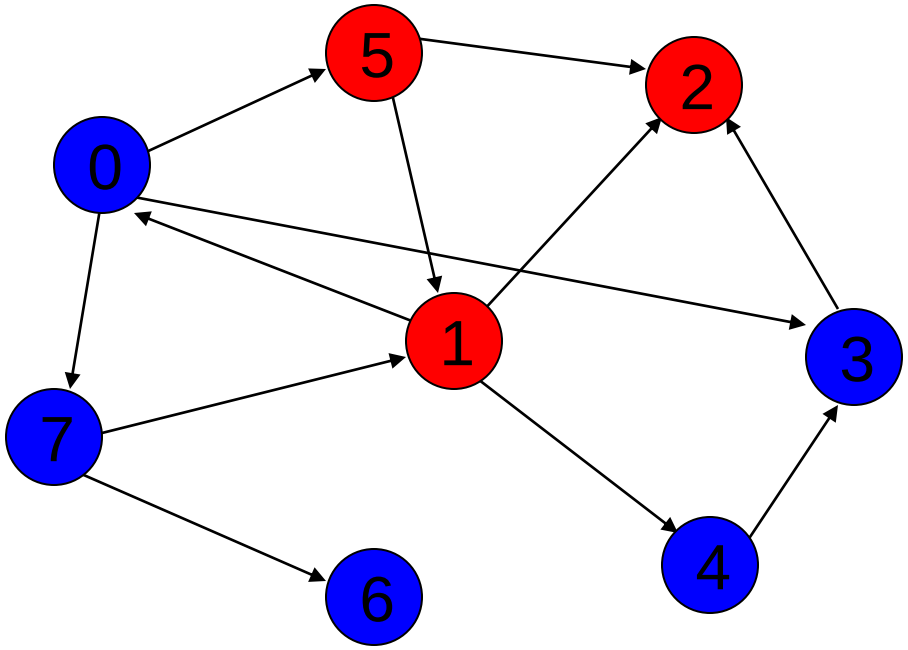
# Изминување по широчина - илустрација

Темиња на растојание 1



# Изминување по широчина - илустрација

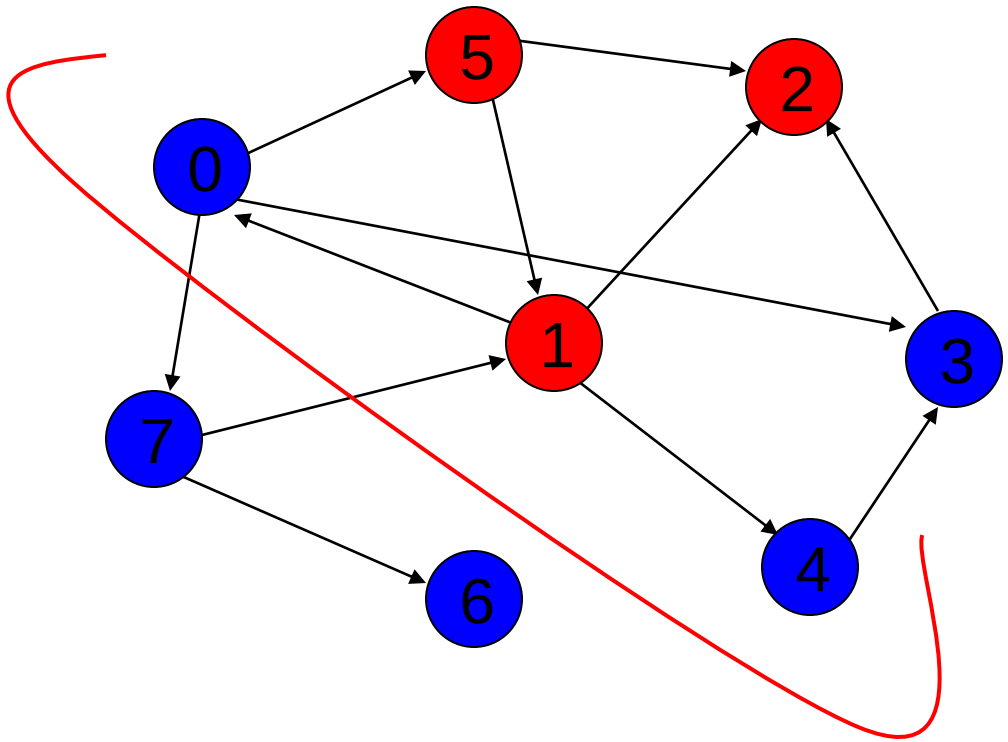
Посети ги темињата 1 и 2




5 1 2

# Изминување по широчина - илустрација

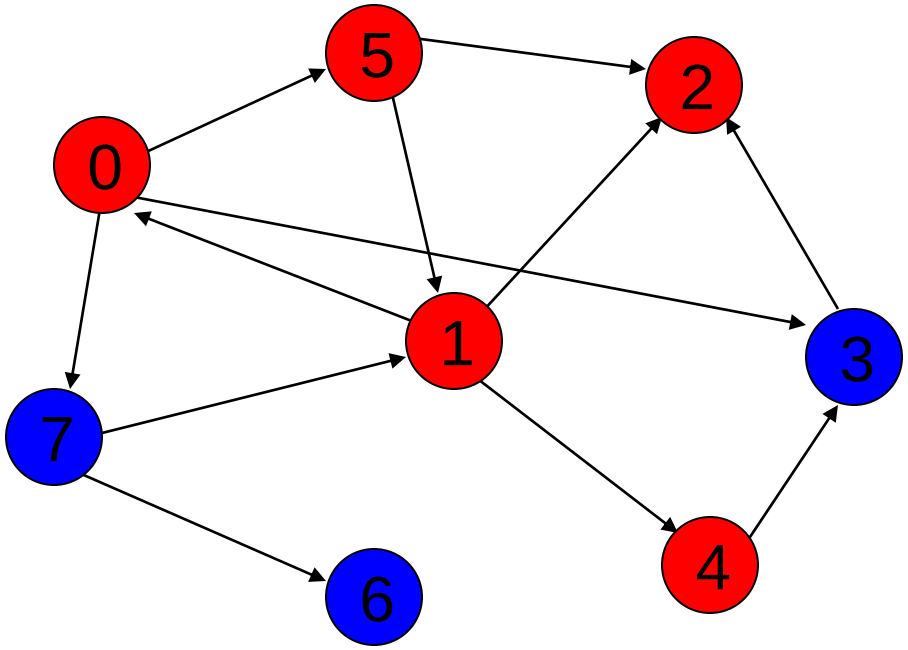
Темиња на растојание 2




5 1 2

# Изминување по широчина - илустрација

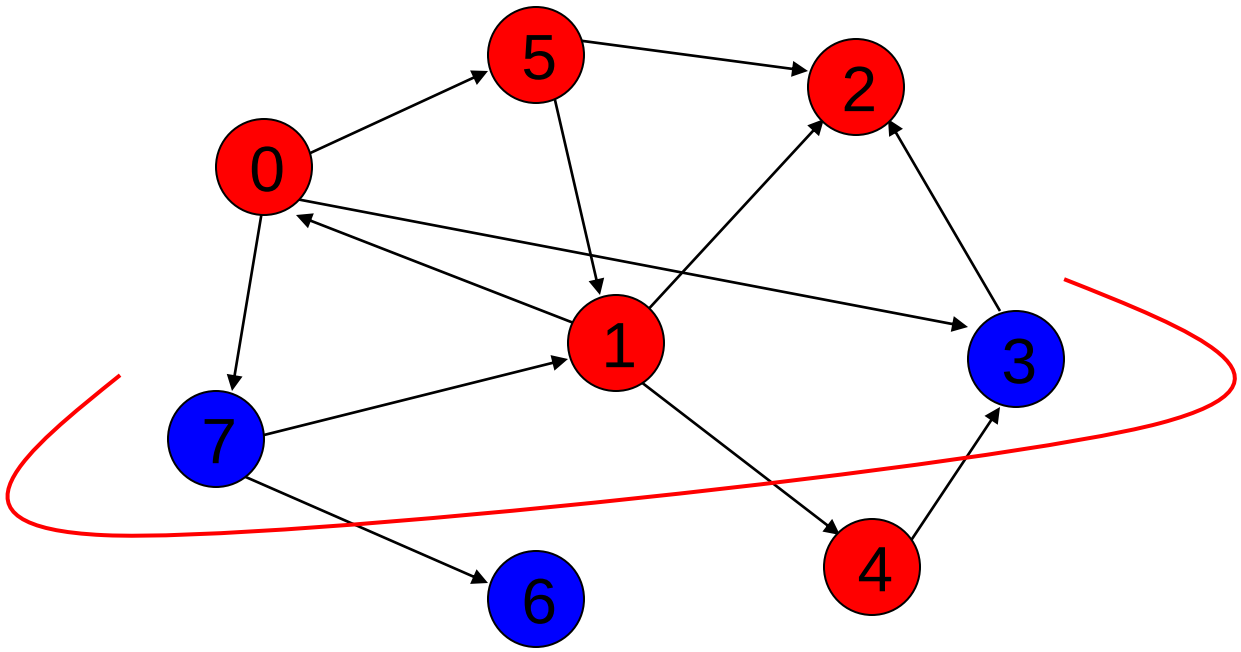
Посети ги темињата 0 и 4




5 1 2 0 4

# Изминување по широчина - илустрација

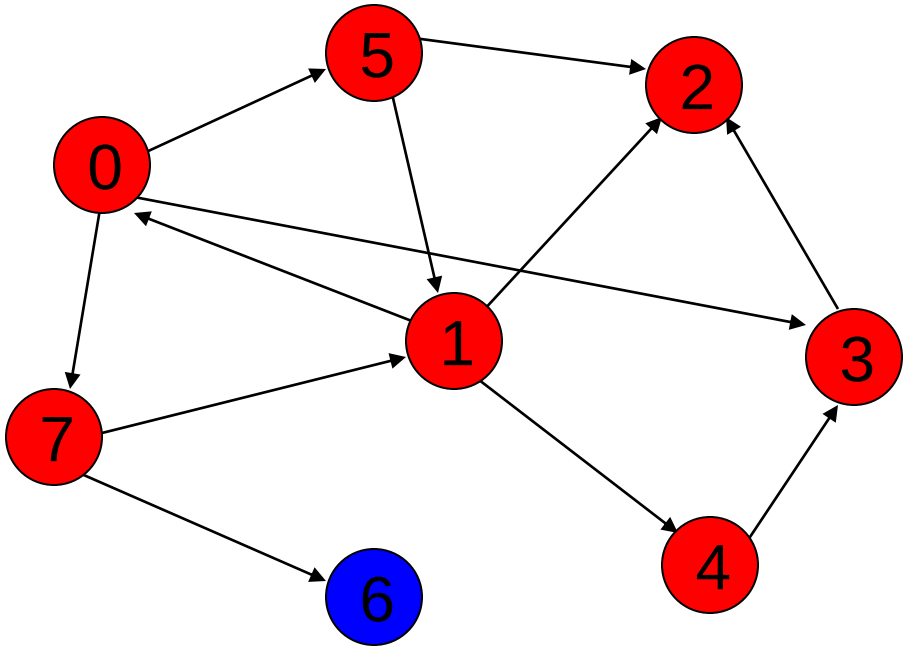
Темиња на растојание 3




5 1 2 0 4

# Изминување по широчина - илустрација

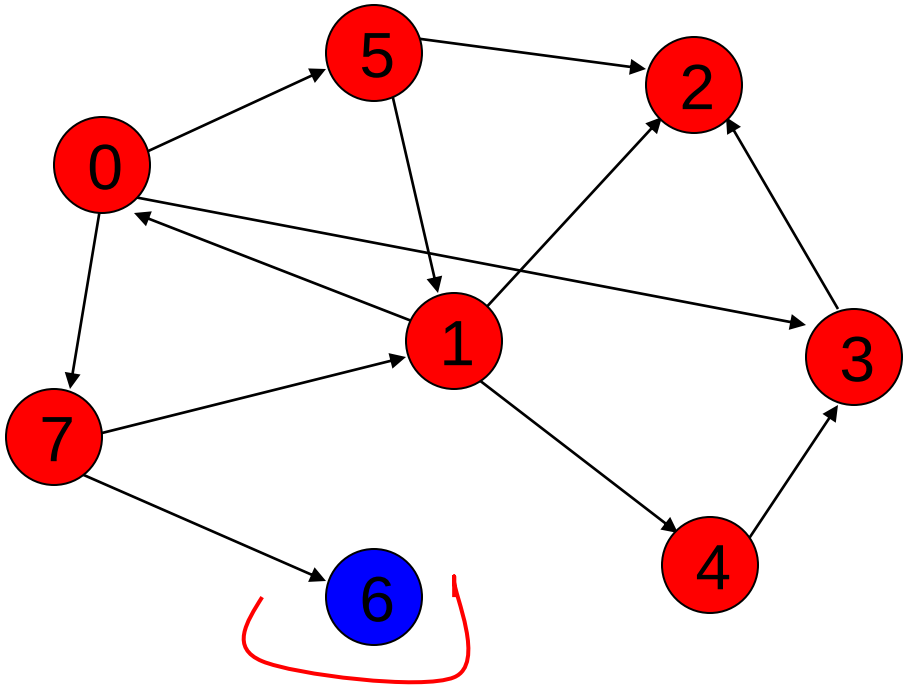
Посети ги темињата 3 и 7




5 1 2 0 4 3 7

# Изминување по широчина - илустрација

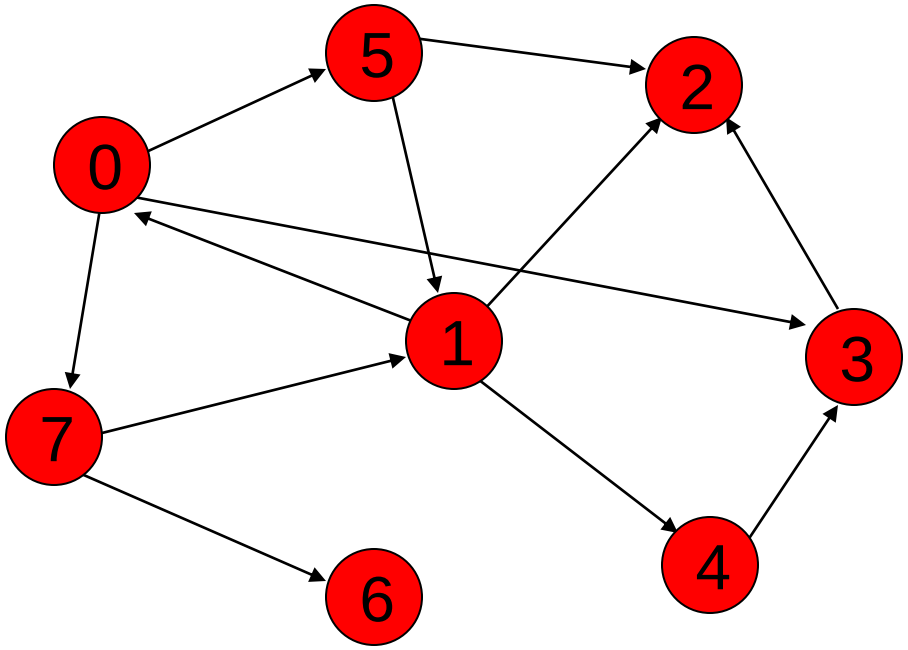
Темиња на растојание 4




5 1 2 0 4 3 7

# Изминување по широчина - илустрација

Посети го темето 6




5 1 2 0 4 3 7 6

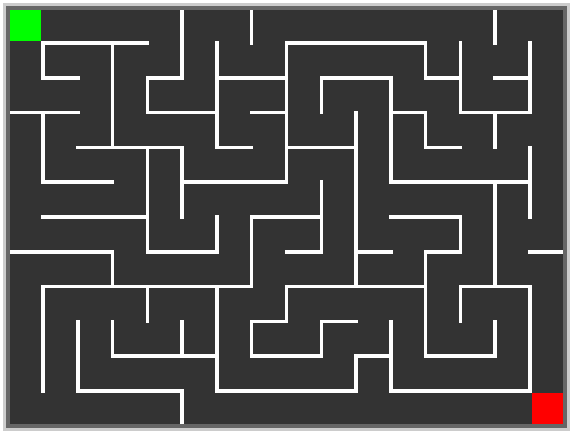
# Изминување по широчина - Java

- **Нерекурзивно**

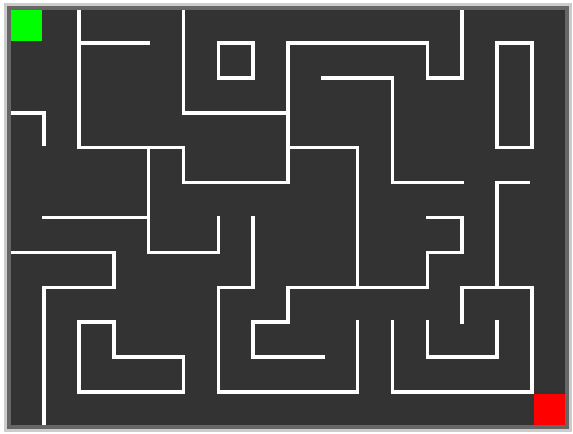
```
public void BFS(T startVertex) {
    Set<T> visited = new HashSet<>();
    Queue<T> queue = new LinkedList<>();

    visited.add(startVertex);
    queue.add(startVertex);

    while (!queue.isEmpty()) {
        T vertex = queue.poll();
        System.out.print(vertex + " ");

        for (T neighbor : getNeighbors(vertex)) {
            if (!visited.contains(neighbor)) {
                visited.add(neighbor);
                queue.add(neighbor);
            }
        }
    }
}
```

# Изминување на лавиринт

- Совршен лавиринт – лавиринт кој има само еден пат од една точка на лавиринтот до било која друга



**Совршен лавиринт**



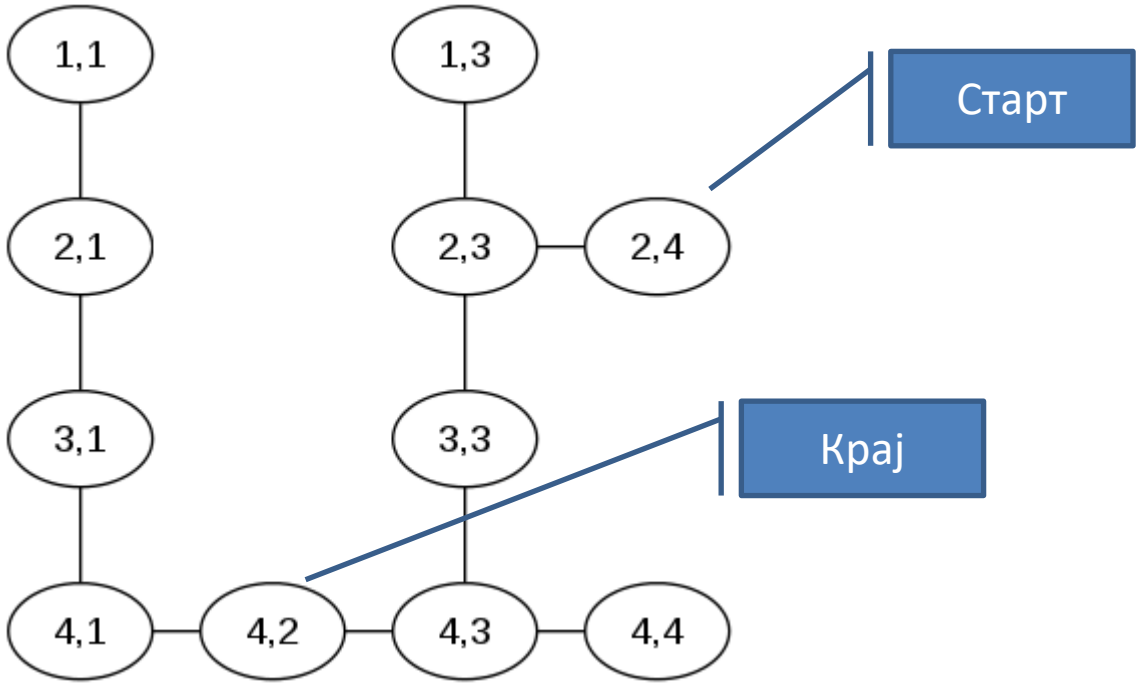
**Несовршен лавиринт**

# Изминување на лавиринт

- Нека лавиринтот е даден во следната форма (како влез од карактери)

```
6,6
######
#  #  ##
#  #  S#
#  #  ##
#  #  ##
#  E   #
######
```

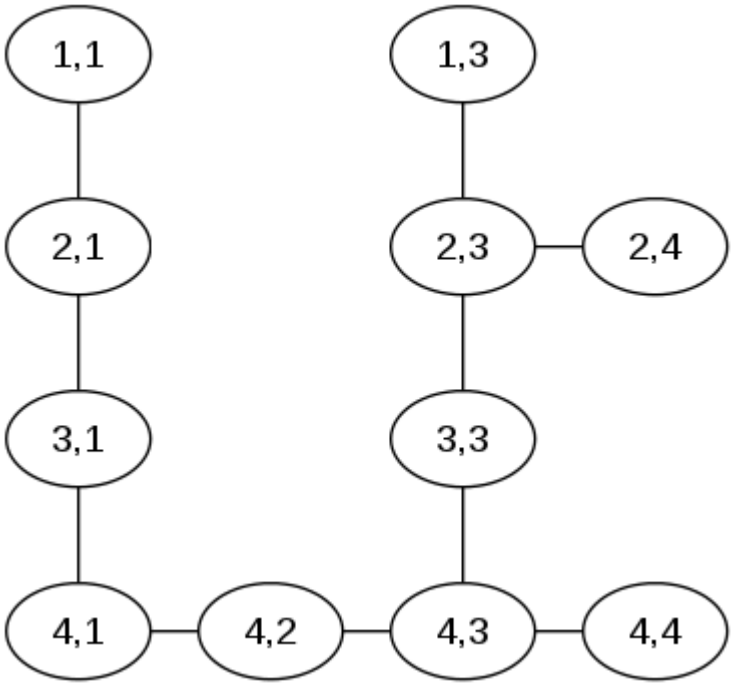
Влез (Лавиринт)



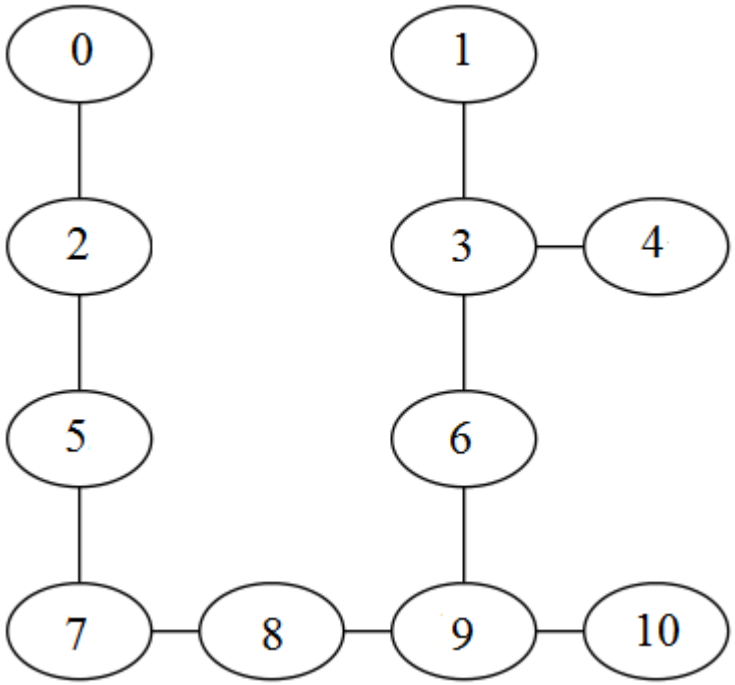

Претставување на лавиринтот со граф

# Изминување на лавиринт

- Нумерирање на јазлите (старт теме 4, крај теме 8)




Претставување на лавиринтот со граф




Нумерирање на јазлите

# Задача 2. Изминување на лавиринт

За дадениот влез (каде е дефиниран совршен лавиринт) да се генерира граф и со изминување по длабочина да се даде патеката која води од почетното до крајното теме.

# Задача 2. Изминување на лавиринт - Java

```
public void findPath(T startVertex, T endVertex) {
    Set<T> visited = new HashSet<>();
    Stack<T> invertedPath = new Stack<>();
    visited.add(startVertex);
    invertedPath.push(startVertex);

    while(!invertedPath.isEmpty() &&
                    !invertedPath.peek().equals(endVertex)) {
        T currentVertex = invertedPath.peek();
        T tmp = currentVertex;

        for(T vertex : getNeighbors(currentVertex)) {
            tmp = vertex;
            if(!visited.contains(vertex)) {
                break;
            }
        }
```

# Задача 2. Изминување на лавиринт - Java

```
        if(!visited.contains(tmp)) {
            visited.add(tmp);
            invertedPath.push(tmp);
        }
        else {
            invertedPath.pop();
        }
    }

    Stack<T> path = new Stack<>();
    while(!invertedPath.isEmpty()) {
        path.push(invertedPath.pop());
    }
    while(!path.isEmpty()) {
        System.out.println(path.pop());
    }
}
```

# Задача 2. Изминување на лавиринт - Java

```
public static void main(String args[]){
    Scanner sc = new Scanner(System.in);
    String tmp = sc.nextLine();
    String parts[] = tmp.split(",");

    int m = Integer.parseInt(parts[0]);
    int n = Integer.parseInt(parts[1]);

    String lines[] = new String[m];

    AdjacencyListGraph<String> mazeGraph = new    AdjacencyListGraph<>();

    String startVertex = "";
    String endVertex = "";
```

# Задача 2. Изминување на лавиринт - Java

```
for(int i=0;i<m;i++) {
    lines[i] = sc.nextLine();

    for(int j = 0; j < n; j++) {
        if(lines[i].charAt(j) != '#') {
            mazeGraph.addVertex(i + "," + j);

            if(lines[i].charAt(j) == 'S') {
                startVertex = i + "," + j;
            } else if(lines[i].charAt(j) == 'E') {
                endVertex = i + "," + j;
            }

            if(i>0 && lines[i-1].charAt(j) != '#') {
                mazeGraph.addEdge((i-1) + "," + j, i + "," + j);
            }

            if(j>0 && lines[i].charAt(j-1) != '#') {
                mazeGraph.addEdge(i + "," + (j-1), i + "," + j);
            }
        }
    }
}
```

# Задача 2. Изминување на лавиринт - Java

```
        sc.close();

        mazeGraph.findPath(startVertex, endVertex);
        }
}
```