# Граф

доц. д-р. Нора Ангелова

# Граф

**Логическо представяне**
Графът се определя като наредена двойка $(V, E)$,
където $V = \{v_1, v_2, \ldots, v_n\}$ е *крайно* множество от върхове,
а $E = \{r_1, r_2, \ldots, r_m\}$ е *крайно* множество от ребра.

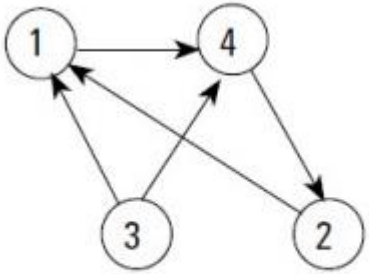
Всяко ребро се задава чрез двойка върхове.
Двойките могат да бъдат наредени ($vi, vj \in V$ и $1 \leq i, j \leq n$), които се наричат **ориентирани ребра** или **дъги.**
Ако ребрата на граф са ориентирани, графът се нарича **ориентиран**.

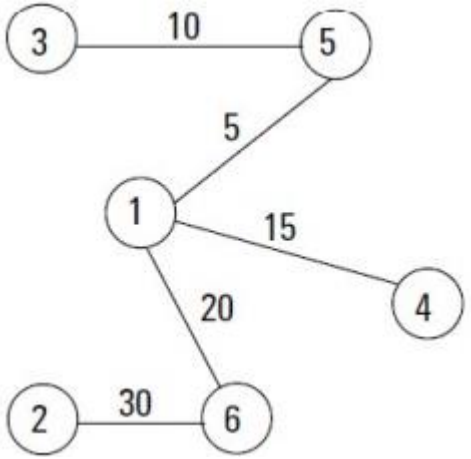
Ако ребрата на графа са зададени чрез ненаредени двойки $(vi, vj)$
от върхове ($vi, vj \in V$ и $1 \leq i, j \leq n$), графът се нарича **неориентиран**.

Ребрата могат да се свързват с етиките – тегла.

# Граф

Примери:



# Граф

**Операции**

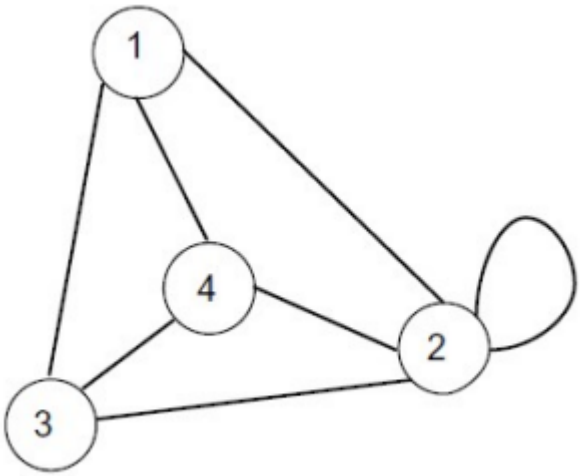
- Създаване на празен граф;
- Достъп до връх;
- Включване/изключване на връх;
- Включване/изключване на ребро;
- Проверка дали граф е празен;
- Проверка дали елемент е връх на граф;
- Проверка дали съществува ребро от един връх до друг връх на граф;
- Намиране на наследниците на даден връх;

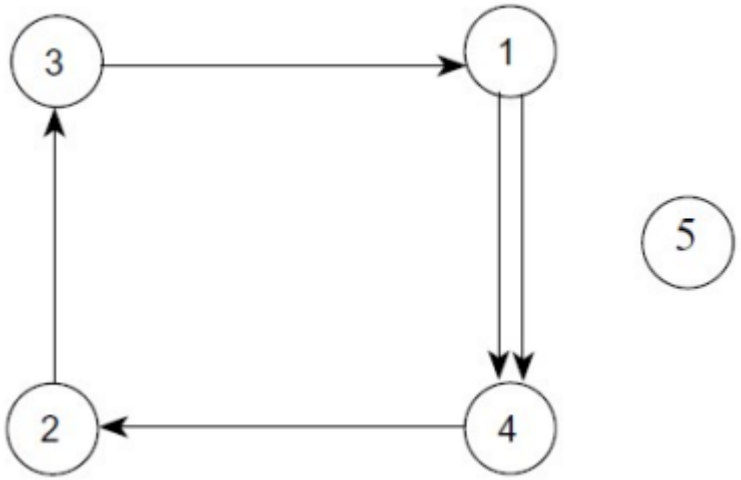
- **Обхождане;**
- **Търсене на (оптимални) пътища и цикли;**

# Граф

**Дефиниции**

- **примка** – ребро, което излиза и влиза в един и същ връх.
- **мултиграф** – граф, при който се допуска се повторение в множеството на ребрата



# Граф

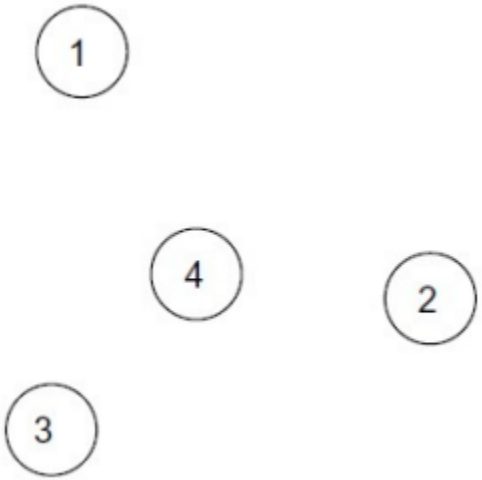
**Дефиниции**

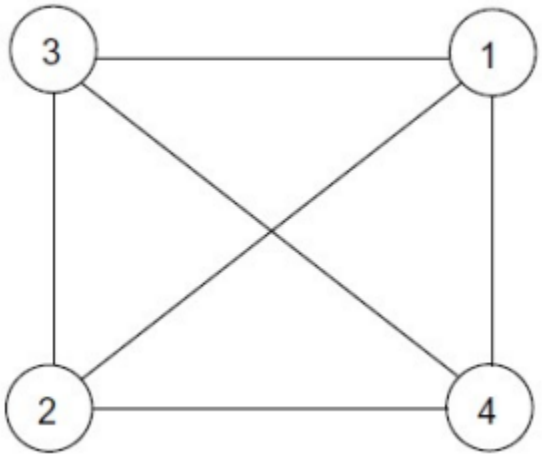
- **полустепен на изхода** на връх $a$ – броят на дъгите $\langle a, b \rangle$, където $b$ е връх на графа
- **полустепен на входа** на връх $a$ – броят на дъгите $\langle b, a \rangle$, където $b$ е връх на графа
- **степен на връх** – сумата от полустепените на входа и изхода на връх
- **изолиран връх** – връх със степен 0

# Граф

**Дефиниции**

- **празен граф –** граф, който не съдържа ребра
- **пълен граф –** граф, за който .всеки два различни върха на граф са свързани с **дъга**

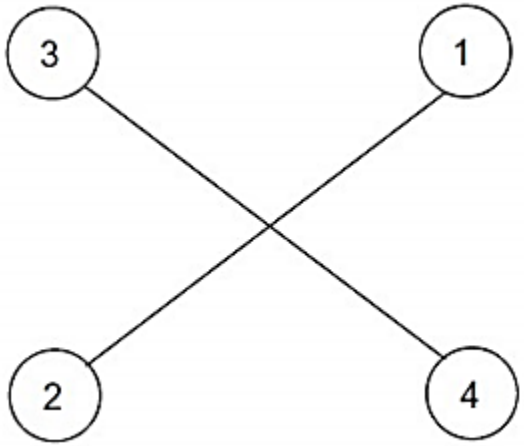


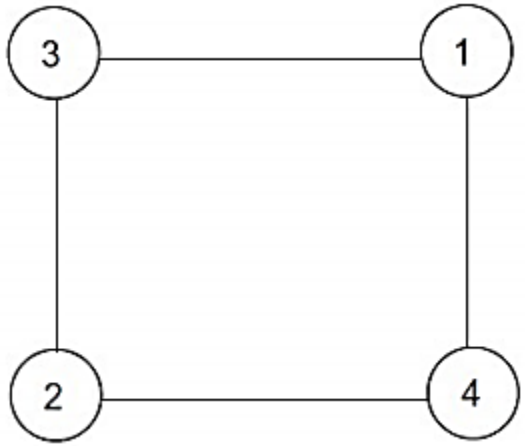



# Граф

**Дефиниции**

Дадени са графите $G_1 = (V, E_1)$ и $G_2 = (V, E_2)$.
Ако за всеки два различни върха *a* и *b* (a, b є V) дъгата принадлежи или на $E_1$, или на $E_2$, то графът $G_2$ се нарича **допълнение на графа** $G_1$,
Графът $G_1$ също се нарича **допълнение на графа** $G_2$.



# Граф

Физическо представяне:

- Последователно
- Свързано

# Граф

Последователно представяне:

- Матрица на съседство
- Матрица на инцидентност

# Граф

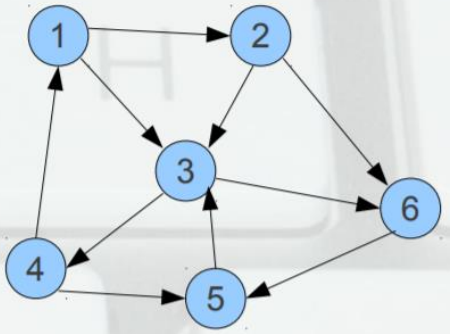
Последователно представяне:

- Матрица на съседство

n – брой върхове в графа.
Матрицата е с размерност n x n.

Нека (i,j) е дъга - елементът в i-ти ред и j-ти стълб на матрицата е 1.
Всички останали елементи на матрицата са 0.

$$\begin{pmatrix} 0 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 1 & 0 & 1 \\ 1 & 0 & 0 & 0 & 1 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 \end{pmatrix}$$



# Граф

Недостатъци на представянето:

- Матриците на съседство заемат прекалено много памет и при много алгоритми водят до квадратично време за изпълнение.
- **Разредени матрици** на съседство – матрици с голям брой нулеви елементи.

# Граф

Предимство на представянето:

- Проверката дали съществува ребро между два върха е **от порядъка на константа**.
- Намирането на наследниците на даден връх е с **линейна сложност**.

# Граф

Последователно представяне:

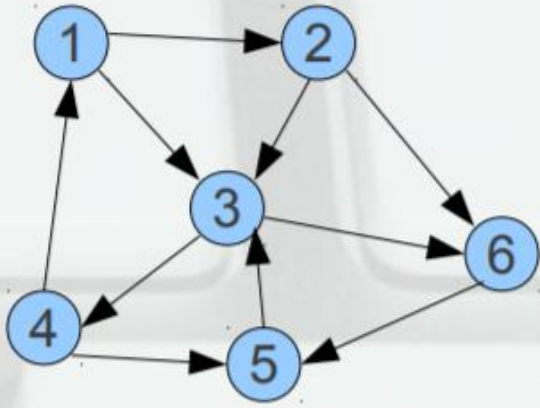
- Матрица на инцидентност

n – брой върхове в графа.
m – брой дъги в графа.
Матрицата е с размерност n x m.

Стойности: -1 / 0 / 1

край x начало на дъга

$$\begin{pmatrix} -1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & -1 & 0 & 0 & 0 & 1 & 0 & 1 & 0 & 0 \\ 0 & 0 & -1 & 1 & 0 & -1 & -1 & 0 & 1 & 0 \\ 1 & 0 & 0 & -1 & 1 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & -1 & 0 & 1 & 0 & 0 & -1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 0 & -1 & -1 & 1 \end{pmatrix}$$



# Граф

Свързано представяне:

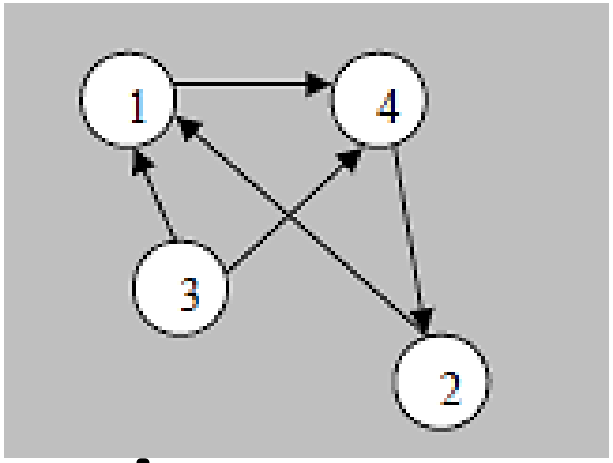
- Свързани списъци (списък на наследниците).
- Свързани списъци (списък на ребрата).

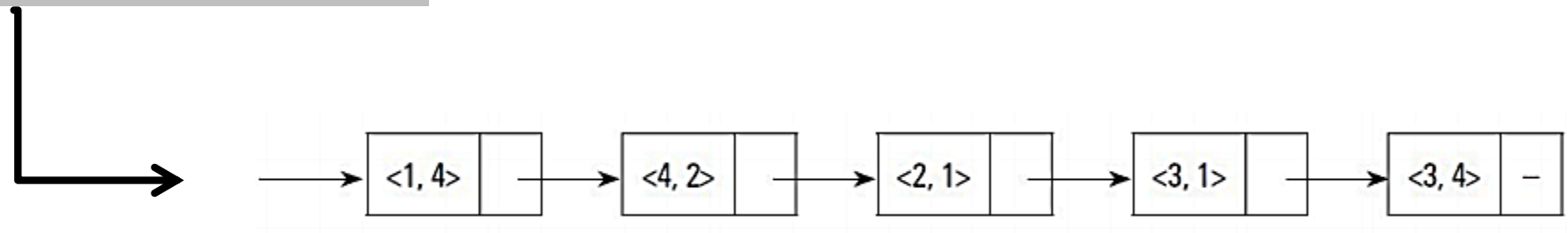
# Граф

Свързано представяне:

- Свързани списъци (списък на ребрата/инцидентност).

Представя се като списък от наредени двойки.



# Граф

Свързано представяне:

- Свързани списъци (списък на наследниците).

n – брой върхове в графа.
Свързан списък с n елемента.

Елементите на списъка са списъци.
Всеки списък започва с връх `i` на графа и съдържа всички върхове `j`, за които съществува дъга от `i` до `j`.

** Използват се само ориентирани графи.*
** Неориентираните графи се представят като ориентирани.*

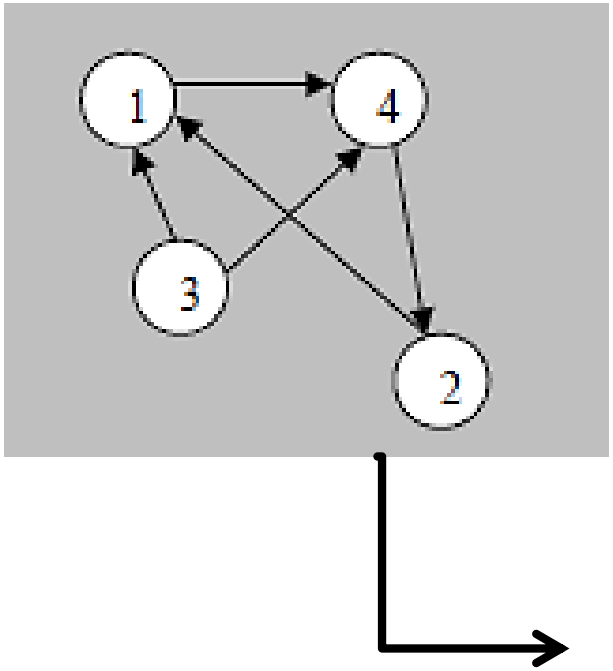
# Граф

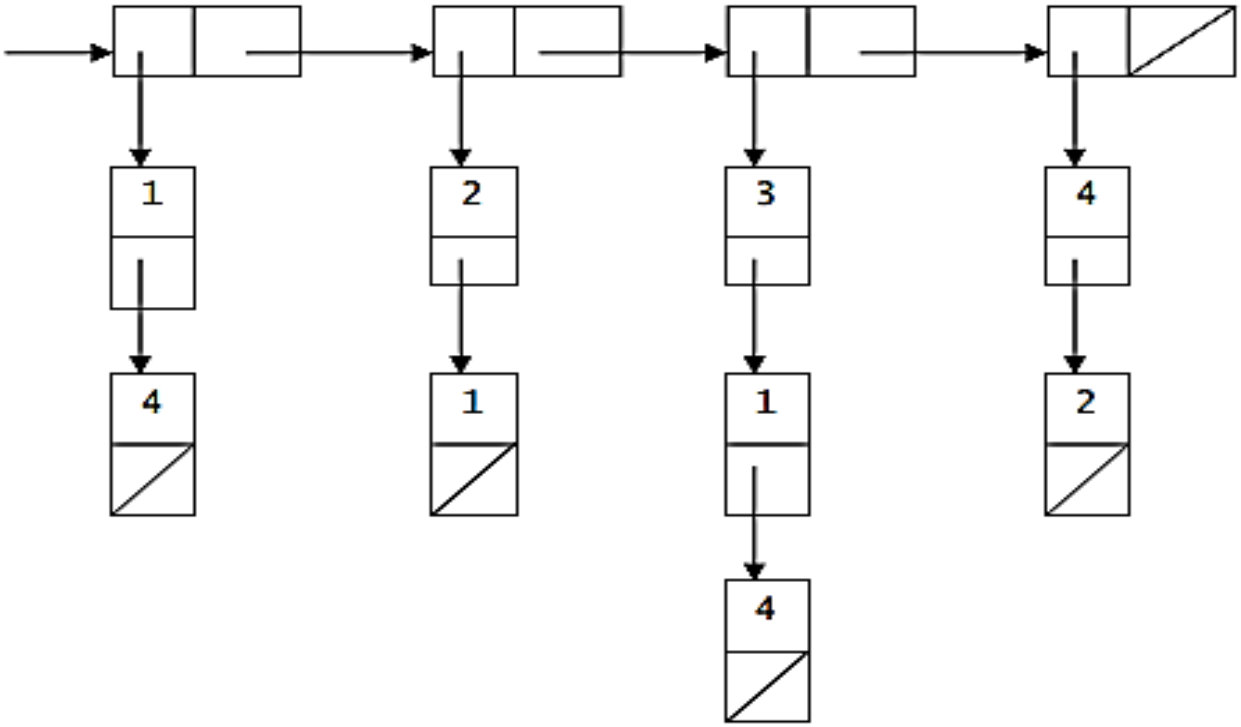
Свързано представяне:

- Свързани списъци (списък на наследниците).

Сложност за намиране на даден връх - от порядъка на броя на върховете.

# Граф



# Граф

Свързано представяне:

- Това представяне ще реализираме чрез шаблона на класа `LinkedList`, дефиниращ свързан списък с една връзка.

** Може да се използва std::vector и друга подходяща структура от данни*

Специализациите:

```
typedef LinkedList<int> IntList;
typedef LinkedList<IntList> IntGraph;
```

определят класа `IntGraph`, задаващ граф с цели числа.

# Граф

- Създаване на граф

Алгоритъм:

1. Въвежда се връх на графа.
   Стъпката може да се изпълнява многократно.

2. Въвеждат се наредени двойки от върхове, означаващи дъги в графа, които се включват в графа.
   Стъпката може да се изпълнява многократно.

# Граф

Обхождане на граф

- Обхождане в ширина, съкратено BFS (от Breadth-FirstSearch).

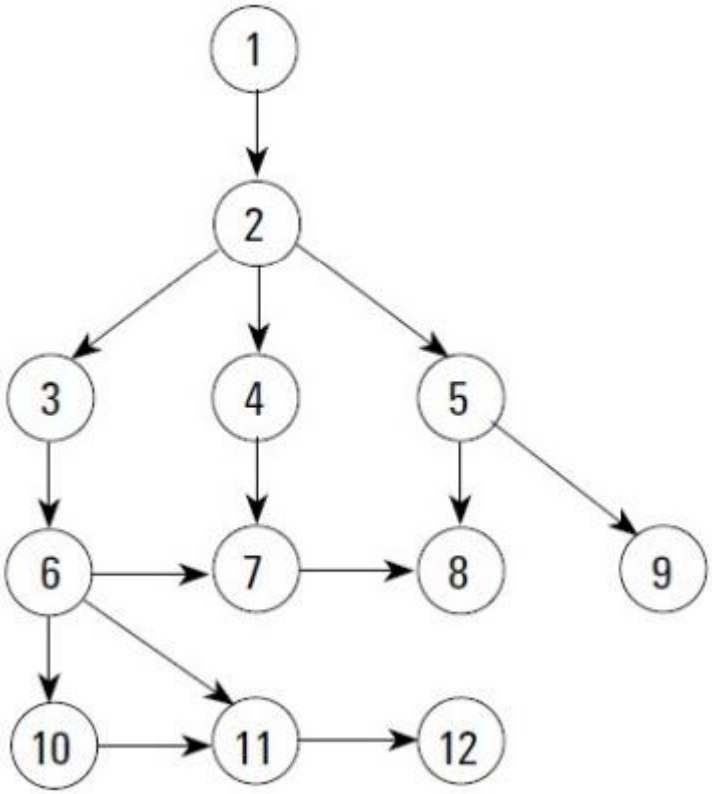
Обхождането в ширина, започващо от даден връх на графа се осъществява по следния начин:
* Започва се от посоченият връх, например *v1 (обхождане и маркиране)*;
* Разглеждат се **<u>всички</u> негови преки наследници**, които още не са обходени.
* Извършва се обхождане в ширина, започващо от всеки от преките наследници на *v1*.

https://www.cs.usfca.edu/~galles/visualization/BFS.html

# Граф

Обхождане на граф

- Обхождане в ширина, съкратено BFS (от Breadth-FirstSearch).




**От връх 3:**

стъпка 1: 3
стъпка 2: 6
стъпка 3: 11, 10, 7
стъпка 4: 12, 8.

# Граф

Обхождане на граф

- Обхождане в дълбочина, съкратено DFS (от Depth-FirstSearch).

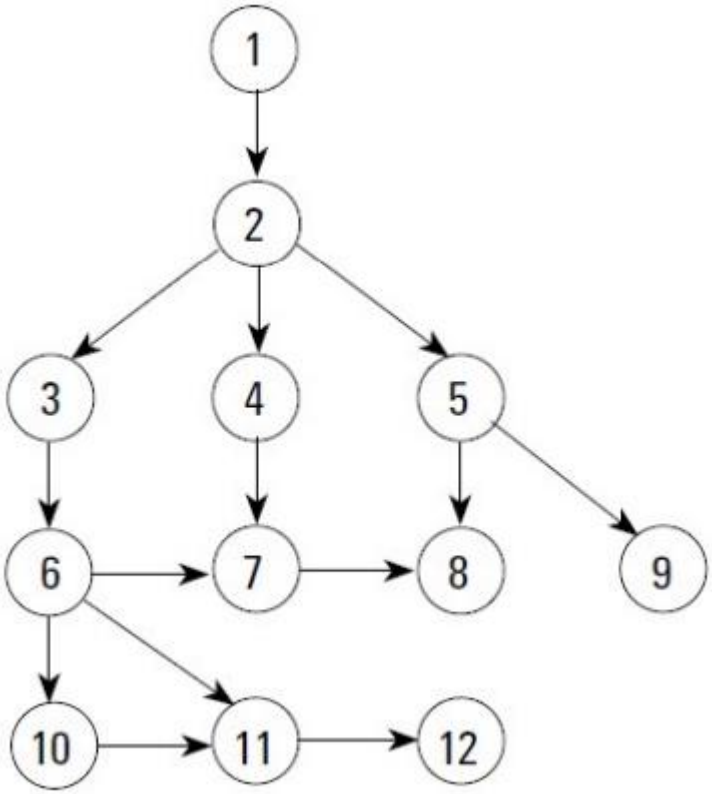
Обхождането в дълбочина започващо от даден връх на граф (DFS(a)), се реализира по следния начин:
* Обхожда се върхът, например *v1*.
* Върхът се маркира, че е обработен/посетен.
* За всяка дъга $\langle a, b \rangle$ на графа, такава че върхът $b$ е необходен се изпълнява **обхождане** в дълбочина DFS(b)

https://www.cs.usfca.edu/~galles/visualization/DFS.html

# Граф

Обхождане на граф

- Обхождане в дълбочина, съкратено DFS (от Depth-FirstSearch).




От връх 1:

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

От връх 3:

3, 6, 11, 12, 10, 7, 8.

# Граф

Път в ориентиран граф G = (V, E) се нарича редица от върхове
$v_1, v_2, \ldots, v_n$, където i = 1, 2, …, n-1 и е в сила, че $<v_i,v_j> \in E$.

Върховете $v_1$ и $v_n$ се наричат **краища на пътя**.
Ако $v_1$ съвпада с $v_n$, пътят се нарича **цикъл**.
Ако всички върхове на пътя са различни, пътят се нарича **прост** (или **ацикличен**).

Ако върховете $v_1$ и $v_n$ съвпадат, а останалите върхове на пътя $v_1, v_2, \ldots, v_n$ са различни, цикълът се нарича прост.
Ако граф съдържа поне един цикъл, той се нарича **цикличен**.
В противен случай графът се нарича **ацикличен**.

# Граф

- Ориентиран граф е **слабо свързан**, ако всеки два негови върха $a$ и $b$ са краища на поне един път, т.е. съществува поне един път от $a$ до $b$ или от $b$ до $a$.
- Ориентиран граф е **силно свързан** (или само **свързан**), ако всеки два негови върха $a$ и $b$ са краища както на път от $a$ до $b$, така и на път от $b$ до $a$.
- Неориентиран граф е **свързан**, ако съществува път между всяка двойка негови върхове.
- Свързан граф без цикли се нарича **дърво**.
- Ацикличен път в граф, който съдържа всички върхове на графа, се нарича **Хамилтонов път**.

# Граф

Намиране на пътища в граф:

- Ацикличен път между два върха.

Алгоритъм:

Съществува път от връх *a* до връх *b*, ако:

- *a* == *b* или
- съществува връх *c*, така че от *a* до *c* има ребро и има път от връх *c* до връх *b*.

** Ако a не съвпада с b и е намерен връх c, така че от a до c има ребро, възможно е задачата за търсене дали има път от c до b да се сведе до търсене на път от a до b и да се стигне до зацикляне.*

** За да се избегне зациклянето може да се използва помощен списък, в който да се записват върховете, които са преминати в процеса на търсене на път.*

# Граф

```
// Общ алгоритъм
bool way(int a, int b, IntGraph &graph, IntList &list) {
    list.insertLast(a);
    if (a == b) {
        return true;
    }

    LinkedListIterator<int> successorsIt = graph.successors(a);
    while (successorsIt) {
        if (!member(*successorsIt, list) && way(*successorsIt, b, graph, list)) {
            return true;
        }
        successorsIt++;
    }

    // Връщане назад
    list.deleteLast();
    return false;
}
```

**O(n!*p)**

# Граф

Сложност:

- n е брой на върховете на графа.
- Намира всички ациклични пътища => сложността може да достигне до n!.
- member – линейна сложност.

- **O(n!*p)**

# Граф

```
// Общ алгоритъм
bool findPathDFSrec(int a, int b, IntGraph &graph, IntList &visited, IntList &path) {
    // Обхождаме a и го добавяме в пътя
    path.insertLast(a);
    visited.insertLast(a);

    if (a == b) {
        return true;
    }

    LinkedListIterator<int> successorsIt = graph.successors(u);
    while (successorsIt) {
        if (!member(*successorsIt, visited) &&
                    findPathDFSrec(*successorsIt, b, graph, visited, path)) {
            return true;
        }
        successorsIt++;
    }

    // Не е намерен път от връх a, той се прехва от пътя
    path.deleteLast();

    // Върхът е бил посетен и път не е намерен, не се премахва от списъка
    // visited.remove(a);
    return false;
}
```

**O(n*p)**

# Граф

Сложност:

- n е брой на върховете на графа.
- Търсим дали съществува път.
  Не повтаряме върхове => сложността може да достигне до n.
- member – линейна сложност.

- **O(n*p)**

# Граф

**Намиране на пътища в граф**

- Всички ациклични пътища между два върха.

Алгоритъм:
Намира всички пътища от връх a до връх b на графа graph.

Всеки път се конструира в списъка list, след което се извежда.

Глобалната булева променлива flag получава стойност true ако съществува път от a до b в graph.

# Граф

```
// Общ алгоритъм
void allWays(int a, int b, IntGraph& graph, IntList& list) {
    list.insertLast(a);
    if (a == b) {
        printIntList(list);
        std::cout << std::endl;
    }
    else {
        LinkedListIterator<int> successorsIt = graph.successors(a);
        while (successorsIt) {
            if (!member(*successorsIt, list)) {
                allWays(*successorsIt, b, graph, list);
            }
            successorsIt++;
        }
    }

    // Връща се назад за търсене на други пътища
    list.deleteLast();
}
```

# Граф

**Топологично сортиране**

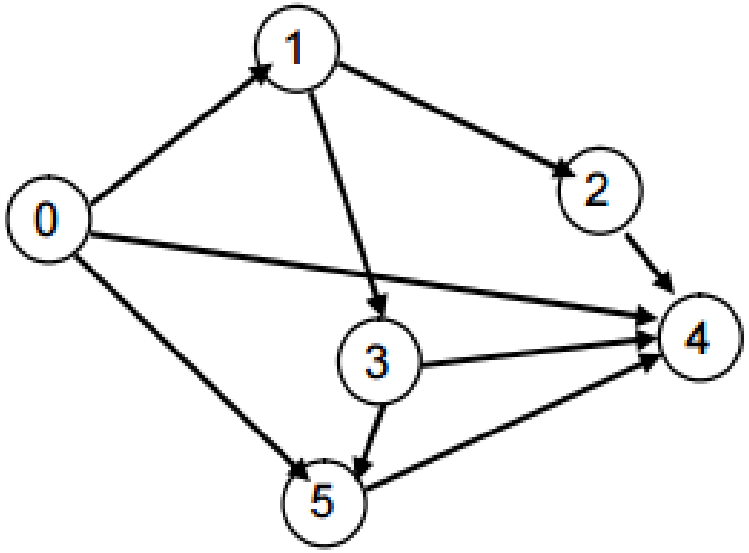
- Изпълнява се само за ориентирани ациклични графи.
- Резултат – редица от номера на върхове, наречена топологичен ред или топологична подредба.
- На един граф могат да съответстват няколко топологични реда, но обикновено е достатъчно да се намери един от тях.

# Граф

Топологичен ред се получава по следния алгоритъм:

- Ако графът няма върхове без предшественици, той е цикличен и не подлежи на топологично сортиране.
- Един по един, в произволен ред записваме в топологичния ред върховете без предшественици и ги изтриваме.
- Ако още има неизтрити върхове, преминаваме към операция 1, ако са изтрити всички, топологичното сортиране е завършило.

# Граф




0, 1, 2, 3, 5, 4.

# Граф

Недостатъци:

- Програмната реализация трябва да използва вътрешното представяне на графа.

Нека сме избрали матрица на съседство.
Ако искаме да следваме описания алгоритъм точно, ще трябва след всяко изтриване на връх да променяме матрицата на съседство.
По този начин създаването на топологичен ред е съпроводено от унищожаване на матрицата на съседство, което често не е желателно.

# Граф

Програмен алгоритъм:

- Регистрират се всички върхове като непосетени.
- Намира се входната степен (преброяват се предшествениците) на всеки връх.
- Всички върхове без предшественици се изпращат в опашката на чакащите върхове.
- Един по един се вземат чакащите върхове от опашката, извеждат се в топологичния ред, регистрират се като посетени, преброяват се като посетени, намалява се входната степен на всеки техен наследник.
  Всички наследници  с входна степен 0 се изпращат в опашката.
- Алгоритъмът продължава докато в опашката има чакащи върхове. Когато опашката остане празна има две възможности:
  - Посетени са всички върхове и топологичният ред е създаден.
  - Посетени са само част от върховете - графът е цикличен и не се поддава на топологично сортиране.

# Граф

- Намиране на най-късите пътища от връх
  - Алгоритъм на Дийкстра
  - Алгоритъм на Флойд
- Минимално покриващо дърво
  - Алгоритъм на Прим
  - Алгоритъм на Крускал
- Максимален поток в граф
  - Алгоритъм на Форд-Фулкерсон

Следва продължение…