

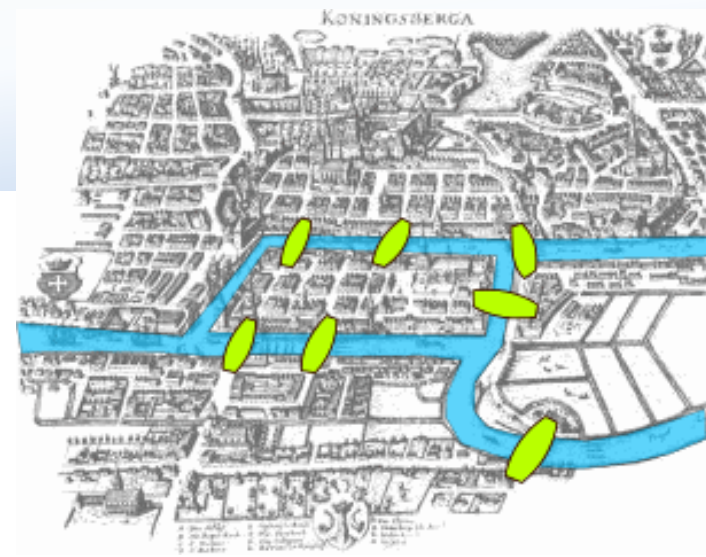
# アルゴリズムとデータ構造

## 第11回 グラフの探索 (1)

# 今日の内容

- グラフ
  - なんでグラフ？
  - グラフの数学的な定義
  - ネットワーク(重み付きグラフ)の定義
- グラフデータの取り扱い方
  - 隣接行列による表現
  - 隣接リストによる表現
- グラフの探索の仕方
  - 幅優先探索
  - 深さ優先探索

# ケーニヒスベルクの橋



- ケーニヒスベルクという街にはプレーゲル川が流れ、7つの橋が架かっていた
- Q: 7つすべての橋を、それぞれちょうど1回ずつ渡って歩くコースって、ある？

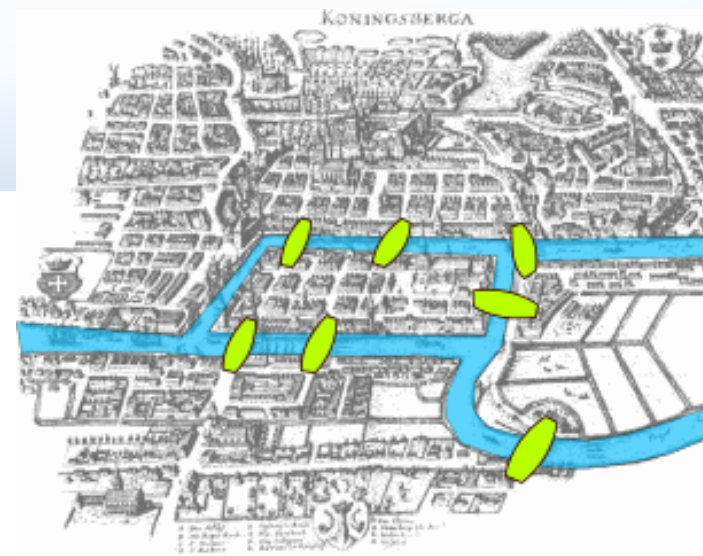
A: ない。  
その理由は ...



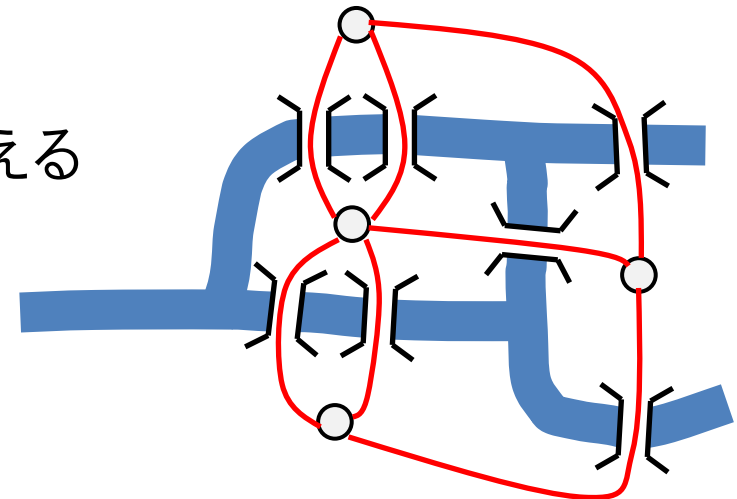
オイラー (Euler) 18世紀の数学者。天文学者や物理学者などでもある。解析学、整数論、トポロジー、グラフ理論などに大きな足跡を残した。

図は、<http://Wikipedia.org> より

# ケーニヒスベルクの橋

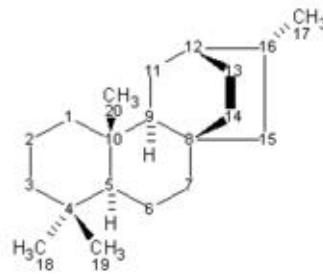
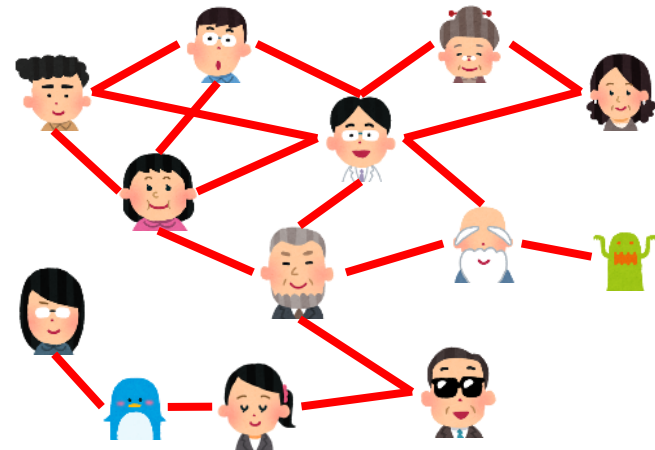
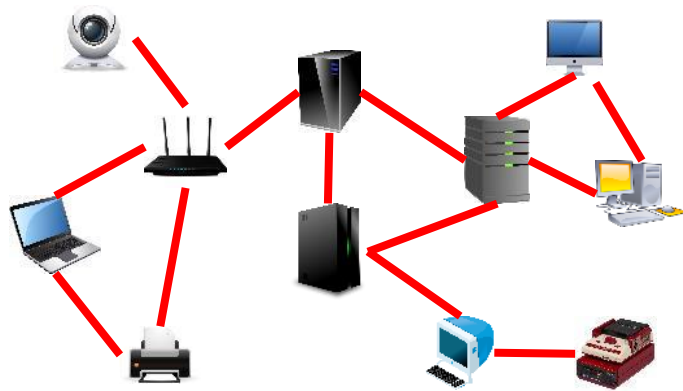


- ケーニヒスベルクという街にはプレーゲル川が流れ、7つの橋が架かっていた
- Q: 7つすべての橋を、それぞれちょうど1回ずつ渡って歩くコースって、ある？
- 川で別れる4つの土地を「頂点」、橋を「(頂点を結ぶ)辺」に置き換える
  - 街をグラフで表す
- グラフが一筆書きできる条件は ...
  - グラフの性質を調べる



# なんでグラフ？

モノとモノのつながりを簡潔に記述し，解析できる！



世の中のすべてはグラフ！

# グラフ (graph) $G = (V, E)$

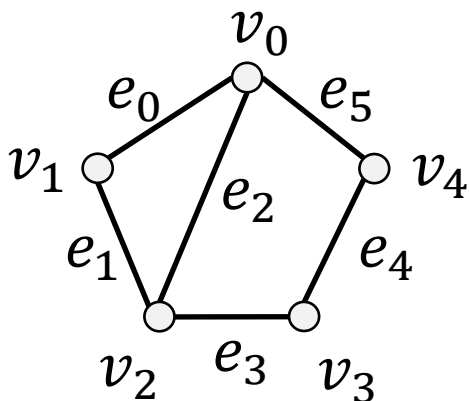
頂点 (vertex) の集合  $V$  と

頂点と頂点のペア

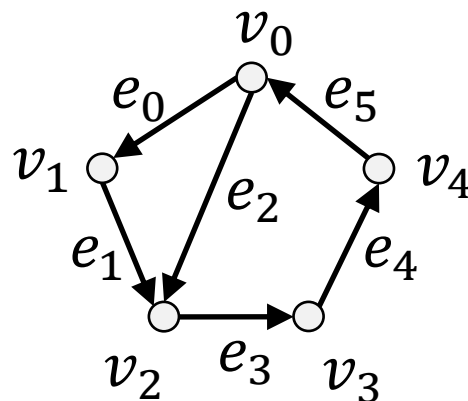
辺 (edge) の集合  $E \subseteq V \times V$  の組  $(V, E)$  のこと

例)  $V = \{v_0, v_1, v_2, v_3, v_4\}$ ,  $E = \{e_0, e_1, e_2, e_3, e_4, e_5\}$  のとき  
ただし,  $e_0 = (v_0, v_1)$ ,  $e_1 = (v_1, v_2)$ ,  $e_2 = (v_0, v_2)$ ,  
 $e_3 = (v_2, v_3)$ ,  $e_4 = (v_3, v_4)$ ,  $e_5 = (v_4, v_0)$

無向グラフ (undirected graph)



有向グラフ (directed graph)



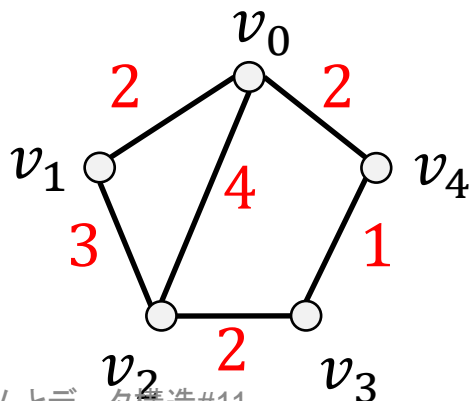
有向グラフの場合, 辺  $(u, v)$  は頂点  $u$  から頂点  $v$  への辺 (有向辺) を表す  
無向グラフでは辺  $(u, v)$  を  $\{u, v\}$  と書く場合もある

# ネットワーク (network)

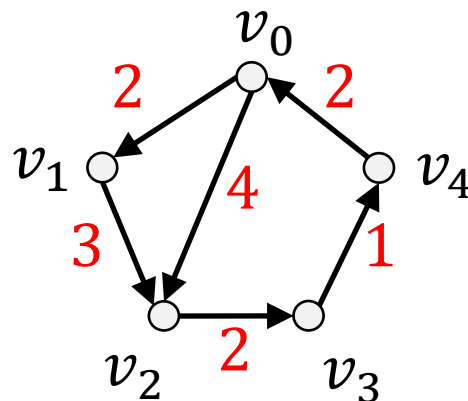
各辺 $e_i$ に**重み** (整数値または実数値)  $w_i$  が付いたグラフのこと. **重み付きグラフ** (weighted graph).

例)  $V = \{v_0, v_1, v_2, v_3, v_4\}$ ,  $E = \{e_0, e_1, e_2, e_3, e_4, e_5\}$  のとき  
ただし,  $e_0 = (v_0, v_1)$ ,  $e_1 = (v_1, v_2)$ ,  $e_2 = (v_0, v_2)$ ,  
 $e_3 = (v_2, v_3)$ ,  $e_4 = (v_3, v_4)$ ,  $e_5 = (v_4, v_0)$   
 $w_0 = 2, w_1 = 3, w_2 = 4, w_3 = 2, w_4 = 1, w_5 = 2$

## 無向ネットワーク

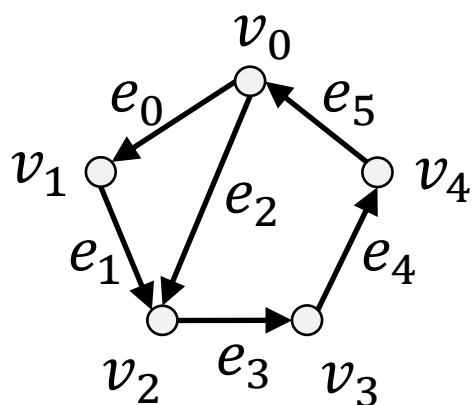


## 有向ネットワーク



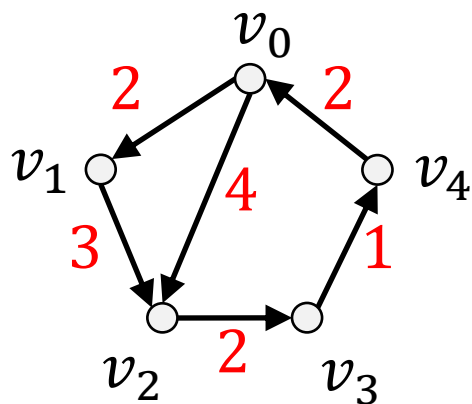
# 隣接行列 (adjacency matrix) による表現

有向グラフについて、各辺の有無を**行列**で表したもの  
(有向ネットワークの場合は重みを要素とした行列)



	$v_0$	$v_1$	$v_2$	$v_3$	$v_4$
$v_0$	0	1	1	0	0
$v_1$	0	0	1	0	0
$v_2$	0	0	0	1	0
$v_3$	0	0	0	0	1
$v_4$	1	0	0	0	0

$$A(i, j) = \begin{cases} 1 & ((v_i, v_j) \in E), \\ 0 & ((v_i, v_j) \notin E) \end{cases}$$



	$v_0$	$v_1$	$v_2$	$v_3$	$v_4$
$v_0$	0	2	4	0	0
$v_1$	0	0	3	0	0
$v_2$	0	0	0	2	0
$v_3$	0	0	0	0	1
$v_4$	2	0	0	0	0

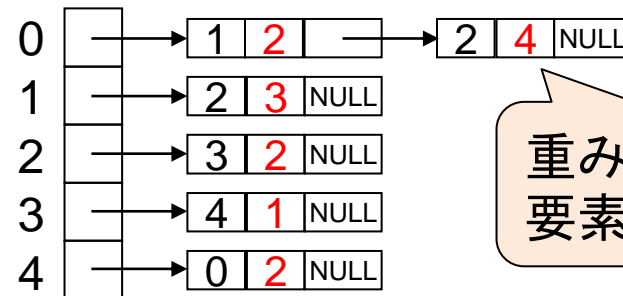
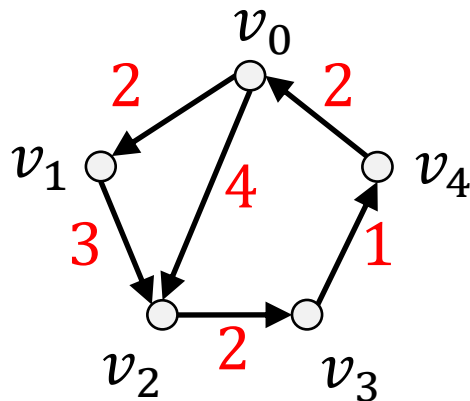
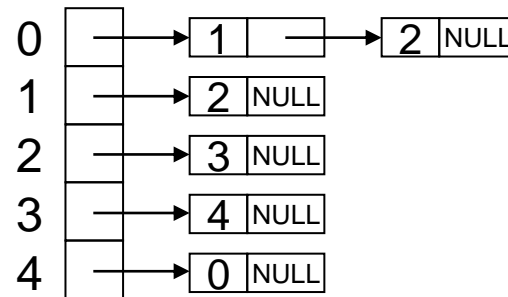
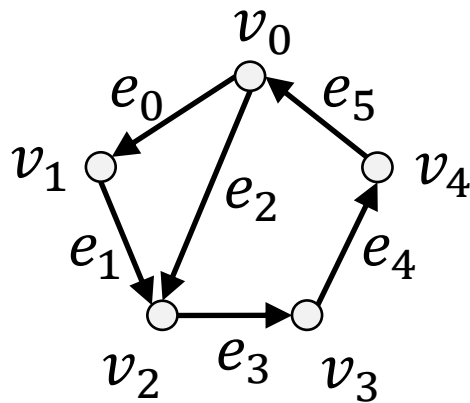
$$A(i, j) = \begin{cases} w_k & ((v_i, v_j) = e_k \in E), \\ 0 & ((v_i, v_j) \notin E) \end{cases}$$

※ 無向グラフの場合は、各無向辺  $(u, v)$  を2つの有向辺  $(u, v)$ ,  $(v, u)$  に置き換えた有向グラフとみなして表現する



# 隣接リスト (adjacency list) による表現

有向グラフについて、各辺の有無を**連結リストの配列**で表したもの(ネットワークの場合は重みを付加する)



重みは第2要素に格納

※ 無向グラフの場合は、各無向辺  $(u, v)$  を2つの有向辺  $(u, v)$ ,  $(v, u)$  に置き換えた有向グラフとみなして表現する

# 隣接行列と隣接リストの利点, 欠点

$$n = |V|, m = |E|$$

隣  
接  
行  
列

## 利点

2頂点間に辺があるか否かを $O(1)$ 時間でチェック可能

## 欠点

$O(n^2)$ の記憶領域が必要

1つの頂点の隣接頂点を求めるのに $O(n)$ 時間必要

隣  
接  
リ  
ス  
ト

## 利点

$O(m)$ の記憶領域で済む

1つの頂点の隣接頂点を求めるのは, その隣接頂点数に比例した時間だけで可能

## 欠点

2頂点間に辺があるか否かをチェックするのに, 隣接頂点数に比例した時間が必要

連結リストを線形探索するから

# 休憩

- ここで、少し休憩しましょう。
- 深呼吸したり、肩の力を抜いてから、次のビデオに進んでください。

# グラフの探索

頂点数 $n$ の入力グラフ  $G = (V, E)$  が与えられたとき、その**すべての頂点を訪問すること**。

- 入力グラフは、隣接リスト表現で与えられるとする
- 出力として、訪問した頂点の並びを出力する。  
ただし、同じ頂点は2回以上重複して出力しないものとする

## 応用

- ゲーム(迷路, 盤ゲーム), 画像処理, etc...

各種グラフ処理の  
基本アルゴリズム

探索法には主に次の2つがある。

1. **幅優先探索** (Breadth-First Search, **BFS**)
2. **深さ優先探索** (Depth-First Search, **DFS**)

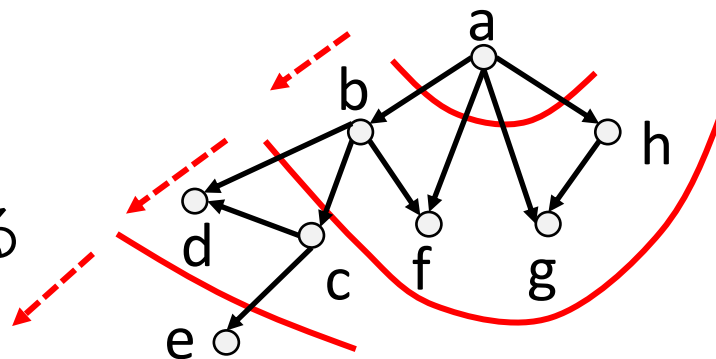
# 幅優先探索の考え方

## 基本的なアイデア

- ソース(スタート地点となる頂点) $s$ から「波」を伝播させて、波の「先端」(frontier)を横断的に訪問(展開)することで、すべての頂点を探索する

## 特長

- ソースから近い順番に訪問できる



## アルゴリズムの動き

- ソース  $s$  を一つ決め, キュー(FIFO)に入れる
- キューから一つ頂点  $v$  を取り出して  $v$  をたどる
- $v$  の隣接頂点のうち未訪問(かつ未発見)の頂点をすべてキューに入れる
- キューが空になるまで2と3を繰り返す

# 幅優先探索アルゴリズム BFS

**Procedure** BFS ( $G$ : グラフ)

```
1:  for each  $v \in V$  do 状態[ $v$ ] ← 白;
2:   $Q \leftarrow$  空のキュー;
3:  状態[ $s$ ] ← 赤; // ソース $s$ は適当な頂点
4:   $Q.Enqueue(s)$ ;
5:  while ( $Q$  が空でない) do begin
6:       $v \leftarrow Q.Dequeue()$ ;
7:       $v$  を出力する; // その頂点进行处理
8:      for each ( $v$  の隣接頂点  $u$ ) do
9:          if (状態[ $u$ ] = 白) then
10:             状態[ $u$ ] ← 赤;
11:              $Q.Enqueue(u)$ ;
12:          end if
13:      状態[ $v$ ] ← 黒;
14:  end while
```

頂点 $v$ の状態の意味

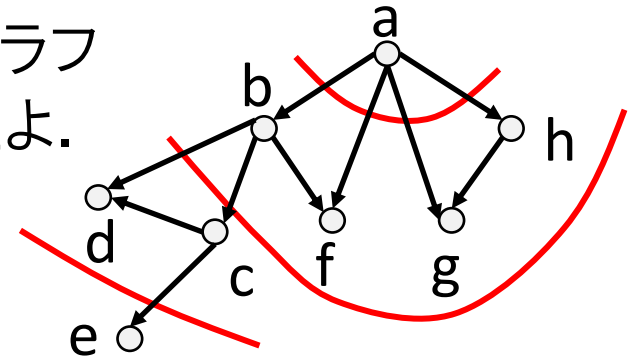
色	意味
白	未訪問
赤	発見済
黒	訪問済

最悪/平均の  
時間計算量は  $O(n + m)$   
( $n$ : 頂点数,  $m$ : 辺の本数)

# 幅優先探索の計算例

問い：右図の隣接リスト表現で与えられるグラフ  
 $G$ のBFSで出力される頂点リストを与えよ。

解答：a, b, f, g, h, d, c, e



ステップ	訪問 頂点 $v$	$v$ の隣接 頂点リスト	キューQの 内容
0	-		<u>a</u>
1	a	b, f, g, h	<u>b</u> , f, g, h
2	b	d, c, <del>x</del>	f, g, h, <u>d</u> , c
3	f	null	g, h, d, c
4	g	null	<u>h</u> , d, c
5	h	<del>x</del>	<u>d</u> , c
6	d	null	<u>c</u>
7	c	<del>x</del> , e	<u>e</u>
8	e	null	-

$G$ の隣接リスト表現

頂点	隣接リスト
a	b, f, g, h
b	d, c, f
c	d, e
d	null
e	null
f	null
g	null
h	g

※ 下線は次に取り出される頂点. 赤字は新たに追加された頂点.

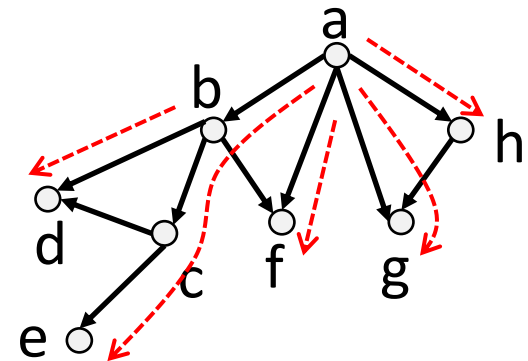
# 深さ優先探索の考え方

## 基本的なアイデア

- ソース(スタート地点となる頂点)  $s$  から, 未発見の頂点を優先的に可能な限り深い方へと探索する

## 特長

- メモリ使用量が少なくなることが多い
- 特に再帰版は実装が簡単



## アルゴリズムの動き

1. ソース  $s$  を一つ決め, **スタック**(LIFO)に入れる
2. スタックから一つ頂点  $v$  を取り出して  $v$  をたどる
3.  $v$  の隣接頂点のうち未訪問(かつ未発見)の頂点をすべてスタックに入れる
4. スタックが空になるまで2と3を繰り返す



# 深さ優先探索アルゴリズム DFS

幅優先探索 (BFS) と見比べて、変わった場所は？

**Procedure** DFS ( $G$ : グラフ)

```
1:  for each  $v \in V$  do 状態[ $v$ ] ← 白;
2:   $Q \leftarrow$  空のスタック;
3:  状態[ $s$ ] ← 赤; // ソース $s$ は適当な頂点
4:   $Q.\text{Push}(s)$ ;
5:  while ( $Q$  が空でない) do begin
6:       $v \leftarrow Q.\text{Pop}()$ ;
7:       $v$  を出力する; // その頂点进行处理
8:      for each ( $v$  の隣接頂点  $u$ ) do
9:          if (状態[ $u$ ] = 白) then
10:             状態[ $u$ ] ← 赤;
11:              $Q.\text{Push}(u)$ ;
12:          end if
13:      状態[ $v$ ] ← 黒;
14:  end while
```

頂点 $v$ の状態の意味

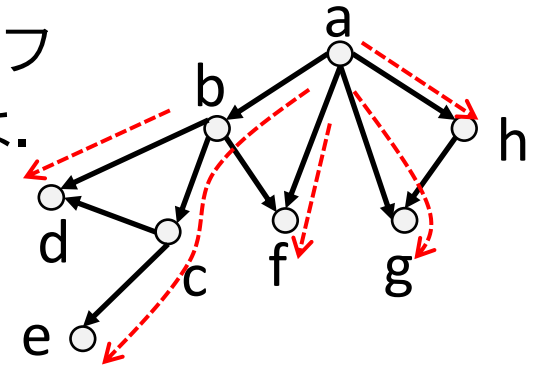
色	意味
白	未訪問
赤	発見済
黒	訪問済

最悪/平均の  
時間計算量は  $O(n + m)$   
( $n$ : 頂点数,  $m$ : 辺の本数)

# 深さ優先探索の計算例

問い：右図の隣接リスト表現で与えられるグラフ  
 $G$ のDFSで出力される頂点リストを与えよ。

解答：a, h, g, f, b, c, e, d



ステップ	訪問 頂点 $v$	$v$ の隣接 頂点リスト	スタックQの 内容
0	-		<u>a</u>
1	a	b, f, g, h	<b>b, f, g, <u>h</u></b>
2	h	<del>g</del>	b, f, <u>g</u>
3	g	null	b, <u>f</u>
4	f	null	<u>b</u>
5	b	d, c, <del>f</del>	<b>d, <u>c</u></b>
6	c	<del>d</del> , e	d, <u>e</u>
7	e	null	<u>d</u>
8	d	null	-

頂点	隣接リスト
a	b, f, g, h
b	d, c, f
c	d, e
d	null
e	null
f	null
g	null
h	g

※ 下線は次に取り出される頂点. 赤字は新たに追加された頂点.

# 深さ優先探索アルゴリズム DFS（再帰版）

**Procedure** DFS ( $G$ : グラフ)

```
1: for each ( $v \in V$ ) do 状態[ $v$ ] ← 白;  
2:  $s \leftarrow$  ソース頂点;    // ソース $s$ は適当な頂点  
3: Visit( $s$ );              // 再帰手続きの開始
```

**Procedure** Visit( $v$ : 頂点)

```
1:  $v$  を出力する;  
2: 状態[ $v$ ] ← 黒;  
3: for each ( $v$ の隣接頂点 $u$ ) do  
4:   if (状態[ $u$ ]=白) then  
5:     Visit( $u$ );    //再帰呼び出し  
6:   end if  
7: end for
```

頂点 $v$ の状態の意味

色	意味
白	未訪問
黒	訪問済

このアルゴリズム場合、隣接リストの順に未訪問の頂点を可能な限り深く探索するので、スタックを使うアルゴリズムとは出力順が異なる点に注意

【練習】先の例のグラフ $G$ に対してどのような順番で出力されるか？

# 今日のまとめ

- グラフ
  - なんでグラフ？
  - グラフの数学的な定義
  - ネットワーク(重み付きグラフ)の定義
- グラフデータの取り扱い方
  - 隣接行列による表現
  - 隣接リストによる表現
- グラフの探索の仕方
  - 幅優先探索
  - 深さ優先探索