Структури от данни и програмиране

Лекция 8

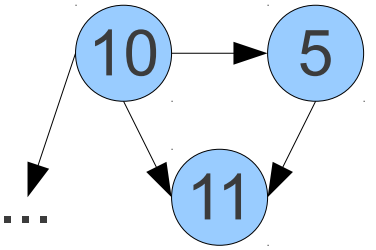
# Граф


# Проблем

- Как да стигнем до ФМИ?

# Друг проблем

- Отиди на гише 10
- Иди на гишето, към което са те насочили от 10, носейки документи, които дават от 11
- С бележката, която ще дадат от гише 10, иди на гише 11
- На гише 11 няма да те обслужат, ако нямаш печат от гише 5
- На гише 20 е необходимо да се представят уверения от 10 и 11
- С бележката от гише 10 отиди на гише 5

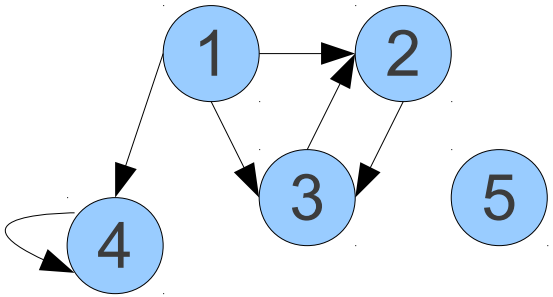


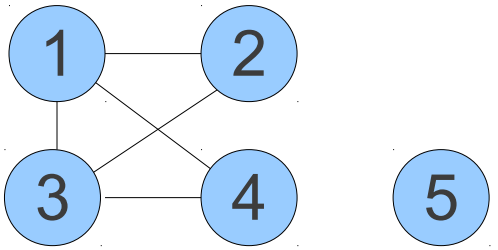

- В какъв ред можем да изпълним изброените задачи?

- За решението на тези и още много други проблеми може да използваме графи

# Основни дефиниции

- ***Краен ориентиран граф***:
  - Наредената двойка ($V$, $E$), където:
  - $V = \{v_1, v_2, \ldots, v_n\}$ – крайно множество от *върхове*
  - $E \subseteq V \times V$ – крайно множество от наредени двойки от вида $(v_i, v_j)$, $v_i, v_j \in V$, наречени *ребра* (*дъги*)
- ***Краен неориентиран граф***:
  - Като горното, но ребрата са ненаредени двойки



# Логическо описание

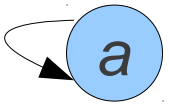
- Нелинейна структура, описваща обекти и връзките между тях
- Основни операции:
  - списък на върховете
  - списък на съседите на даден връх
  - проверка за съществуване на ребро
  - добавяне и премахване на връх
  - добавяне и премахване на ребро

# Някои от многото приложения на графите

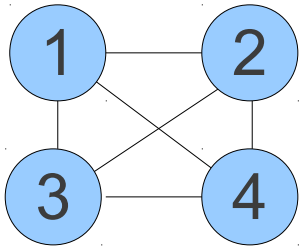
- Почти всяка съвкупност от обекти с дефинирани връзки може да бъде представена като граф:
  - Примери от транспорта – очевидни:
    - Градове и пътища между тях
    - Кръстовища и улици – двупосочни и еднопосочни
  - Социални мрежи – хора и връзки между тях
  - Шахмат, морски шах – всеки връх е конкретно състояние на дъската, а между два върха има дъга, ако от едното състояние може да се достигне до другото с точно един ход
  - Бинарни релации в математиката, напр. “<”
    - В задачата за планиране на дейности има релация между отделните дейности – “трябва да се изпълни преди”

# Още дефиниции

- *Примка* – ребро от вида (*a*,*a*)




- *Мултиграф* – граф, в който множеството от ребрата допуска повторения
- *Празен граф* – граф без ребра ($E = \emptyset$)
- *Пълен граф* – граф, в който между всеки два различни върха има ребро



- *Претеглен граф* – ($V$, $E$, $f$), където $f$:$E \rightarrow W$ е функция, която задава тегла на ребрата
  - Пример: върховете са градове, ребрата са пътища, а теглата са дължините на пътищата
  - Примерна задача: да се намери най-късият път между два града

# И още дефиниции

- Два върха *a* и *b* в ориентиран граф се наричат *съседни*, ако $(a, b) \in E \vee (b,a) \in E$, т.е. има поне една дъга между тях
- *a* и *b* са *краища* на дъгата (*a*, *b*)
- *a* е предшественик на *b*
- *b* е наследник на *a*

# Още и още дефиниции

- Нека *G* е ориентиран граф
- *Полустепен на изхода на връх a* – броят на дъгите (*a*, *b*), където *b* е връх на *G*
- *Полустепен на входа на връх a* – броят на дъгите (*b*, *a*), където *b* е връх на G
- *Степен на връх a* – сумата от полустепените на входа и изхода
  - При неориентиран граф – броят на ребрата (*a*,*b*)
- *Изолиран връх* – връх със степен 0

# Пътища в граф

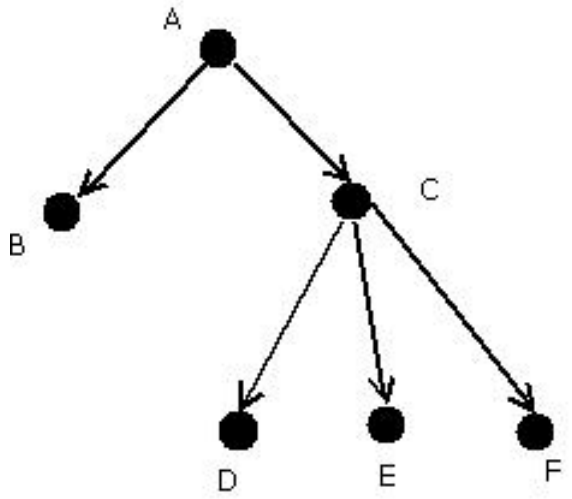
- *Път* в граф – редица $v_1$, $v_2$, ..., $v_n$, за която ($v_i$, $v_{i+1}$) $\in E$
- *Цикъл* – път, за който $v_1 = v_n$ и $n > 1$
- *Ацикличен (прост) път* – всички негови върхове са различни
- *Прост цикъл* – цикъл, в който всички върхове са различни, с изключение на $v_1$ и $v_n$

# Пътища в граф (2)

- *Слабо свързан* ориентиран граф – между всеки два негови върха *a* и *b* съществува път от *a* до *b* или от *b* до *a*
- *(Силно) свързан* ориентиран граф – между всеки два върха има пътища и в двете посоки
- Неориентиран граф е *свързан*, ако между всеки два върха има път

# Връзка с дървета

- Неориентиран свързан граф без цикли – дърво
- Ако изберем даден връх за корен – кореново дърво (тези, които бяха разгледани в една от предходните лекции)

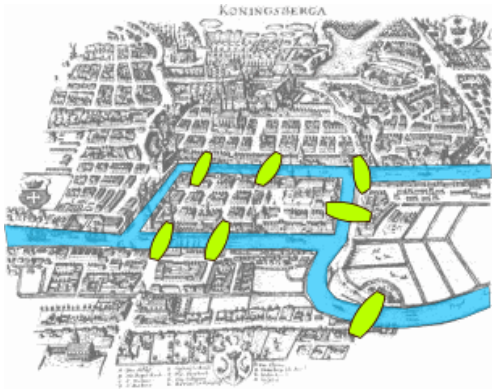


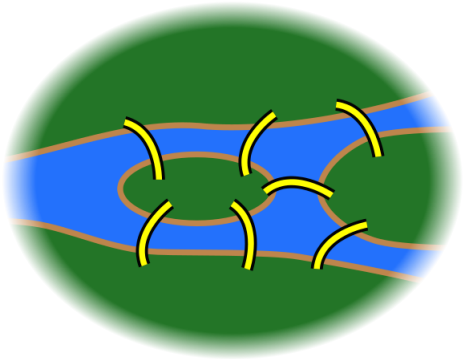
# Пътища в граф (3)

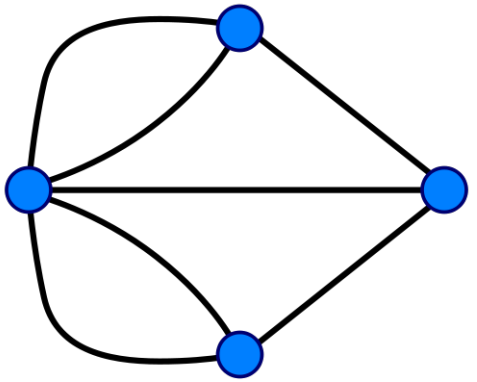
- *Хамилтонов път* – ацикличен път, който съдържа всички върхове на графа
- *Хамилтонов цикъл*
  - Класическо приложение: да се достави стока до всички градове, но да се измине най-кратко разстояние – *задача за търговския пътник*, използва се претеглен граф

# Пътища в граф (4)

- *Ойлеров път (цикъл)* – път (цикъл), който преминава през всички ребра на графа точно по веднъж
- Първоначален проблем: може ли да се направи разходка в гр. Кьонигсберг, при която да се премине по всеки от 7-те моста точно веднъж и обиколката да приключи в изходната позиция
- Леонард Ойлер показва, че не може
  - Преформулира проблема по абстрактен начин (с мултиграф)
- Начало на теорията на графите





# Моделиране с графи

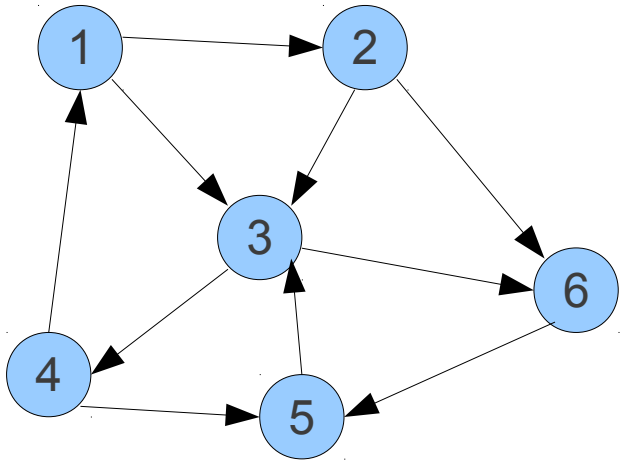
- Както вече видяхме в много примери, огромно разнообразие от реални ситуации може да се моделира с просто средство – граф
- Съществуват множество утвърдени алгоритми за графи

# Основни начини за представяне на граф

- Последователно
  - Матрица на съседство
  - Матрица на инцидентност
- Свързано
  - Списък на инцидентностите
  - Списък на ребрата
- Както винаги, всеки от начините има своите предимства и недостатъци

# Матрица на съседство, матрица на теглата

- Върховете на графа са номерирани от 1 до *n*
- Ако има дъга от връх *i* до връх *j*, то елементът на *i*-ти ред и *j*-ти стълб на матрицата е равен на 1 (или на теглото на дъгата), в противен случай – на 0




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

# Матрица на съседство – свойства

- Матрицата на съседство на неориентиран прост граф е симетрична
- Ако повдигнем матрицата на *k*-та степен, елементите ще показват дали съществува път между два върха с дължина *k*, например:
  - $A^0$ – единичната матрица, път с дължина 0 има само до същия връх
  - $A^1$ – оригиналната матрица, показва дали има път с дължина 1, т.е. ребро между два върха
  - $A^2$ – дали има път с дължина 2 между два върха

# Матрица на съседство – предимства и недостатъци

- Нека $|V| = n$, т.е. графът има $n$ върха

  \+ Проверка за съществуване на ребро между два върха – O(1)

  \- Намирането на всички наследници е с линейна сложност O($n$), дори и реално да са малко

  \- Използваната памет е O($n^2$), но често матриците са разредени (при малко ребра има много нулеви елементи)

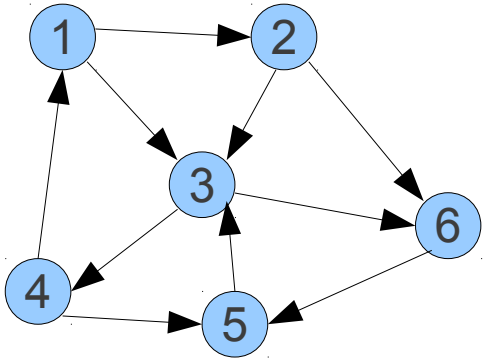
# Матрица на инцидентност между върхове и ребра

- Ако броят на върховете е $n$, а на ребрата – $m$, матрицата има размерност $n$x$m$
- Редовете представят върховете, а стълбовете – ребрата
- При ориентиран граф за всяко ребро $e_k=(i,j)$ $k$-ият стълб ще съдържа 1 в $i$-тата и -1 в $j$-тата си позиция
- При неориентиран – вместо -1 се слага 1, а при примка $(i,i)$ – 2 в $i$-тата позиция

# Матрица на инцидентност между върхове и ребра (2)

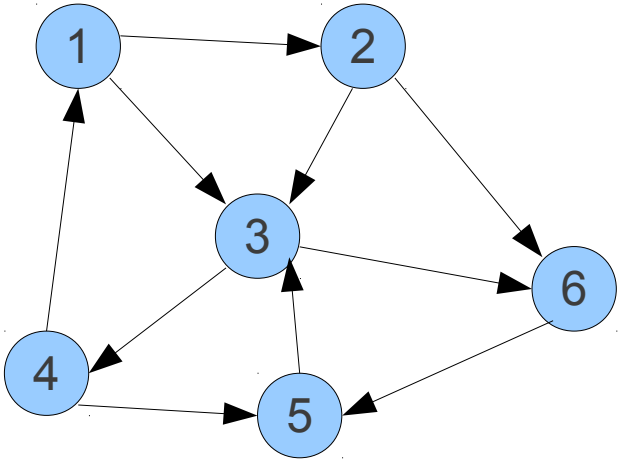
- Използва се рядко, за специфични задачи
- Недостатъци:
  - Използваната памет е O($n$*$m$) и матрицата е силно разредена
  - Проверката за съществуване на ребро има сложност O($m$) (в най-лошия случай O($n^2$))

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



# Списък на наследници

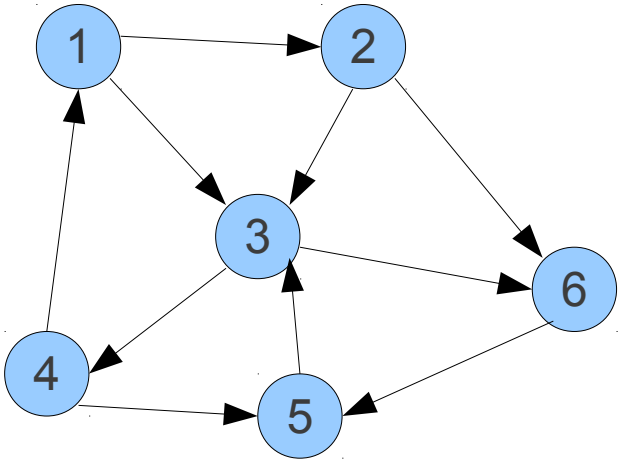
( (1, (2, 3) ),
  (2, (3, 6) ),
  (3, (4, 6) ),
  (4, (1, 5) ),
  (5, (3) ),
  (6, (5) ) )



# Списък на инцидентност

- Списък на дъгите

  ( (1, 2), (2, 3), (3, 6),
  (1, 3), (4, 1), (3, 4),
  (5, 3), (6, 5), (4, 5),
  (2, 6) )




- Представяне, най-близко до формалната дефиниция

# Сравнение на различните представяния

- В таблицата са посочени най-ниските възможни сложности, например при списък можем да изберем реализация, в която се извършва двоично търсене
- Възможно е даден граф да се представи едновременно по няколко начина, за да се възползваме от предимствата им

| | Матрица на съседство | Списък на наследници | Списък на ребрата | Матрица на инцидентност |
|---|---|---|---|---|
| Добавяне на ребро | $O(1)$ | $O(\log n)$ | $O(\log m)$ | $O(1)$ |
| Изтриване на ребро | $O(1)$ | $O(\log n)$ | $O(\log m)$ | $O(m)$ |
| Проверка за същ. ребро | $O(1)$ | $O(\log n)$ | $O(\log m)$ | $O(m)$ |
| Намиране на наследници | $O(n)$ | $O(d_i)$ - степен | $O(n+\log m)$ | $O(m)$ |
| Необходима памет | $O(n^2)$ | $O(m)$ | $O(m)$ | $O(n.m)$ |
| | | | | |

# Примерна реализация на граф

- (във външен файл)
- Кое от разгледаните представяния е използвано в предложената реализация?

# Обхождане на граф

- Последователно посещаване на всеки връх от графа точно по веднъж
- *Стратегията за обхождане* определя в какъв ред ще бъдат разгледани върховете

# Стратегии за обхождане на граф

- Фундаментални стратегии:
  - *Обхождане в дълбочина*
  - *Обхождане в ширина*
- Защо са важни тези две стратегии:
  - Намират приложение в много задачи
  - С тяхна помощ могат да се решат сложни алгоритмични проблеми
    - Евентуално с прилагане на леки модификации
  - Обикновено е добра алгоритмичната сложност

# Обхождане в дълбочина

- Depth-First Search (DFS)
- Псевдокод:

  DFS(*a*) {

    разглеждаме *a*;

    маркираме *a* като обходен;

    за всеки необходен наследник *b* на *a* изпълняваме рекурсивно DFS(*b*)

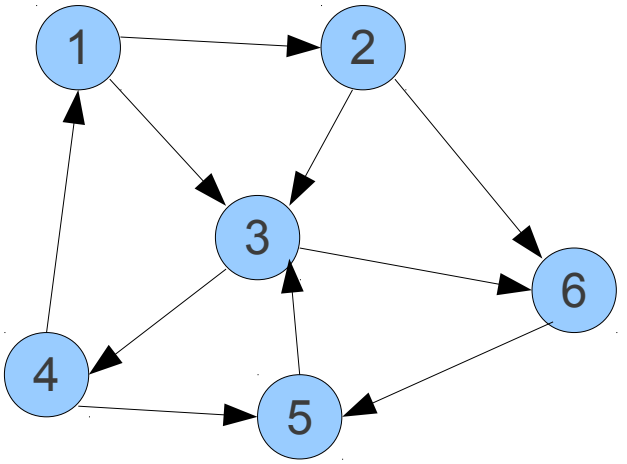
  }
- Какво може да се случи, ако не запазваме кои върхове са обходени?

# Обхождане в дълбочина (2)

- Пример:
- Начален връх: 1
- Върховете ще бъдат обходени в следния ред:

  1, 2, 3, 4, 5, 6

  (ако наследниците на всеки връх са сортирани във възходящ ред)




- Втори пример: при начален връх 2

  2, 3, 4, 1, 5, 6

- Примерна реализация – в `graphtest.cpp`

# Обхождане в ширина

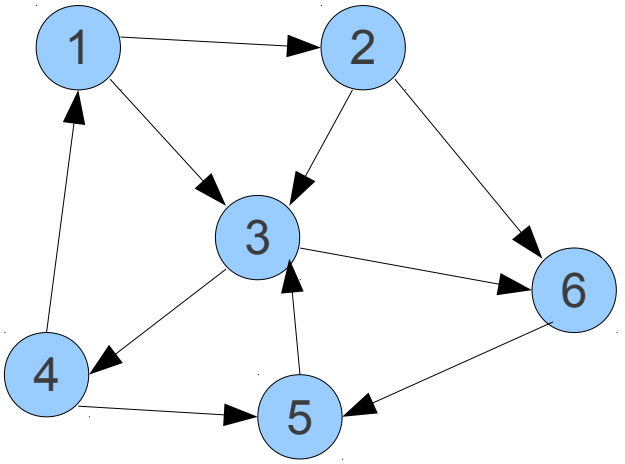
- Breadth-First Search (BFS)
- Обхождане “по нива”
- Псевдокод:

```
BFS(a) {
    разглеждаме a; маркираме a като обходен;
    конструираме редица с един елемент – a;
    докато има елементи в редицата:
        ако текущият връх b не е обходен:
            разглеждаме b; маркираме b като обходен;
            добавяме всички наследници на b в края на редицата;
}
```

# Обхождане в ширина (2)

- Пример:
- Начален връх: 1
- Върховете ще бъдат обходени в следния ред:

  1, 2, 3, 6, 4, 5



# Сравнение на двата алгоритъма

- В алгоритъма за BFS реално се използва опашка
- Ако сменим опашката със стек, се получава DFS
  - Нека си спомним за връзката между рекурсия и стек
- Т.е. двата алгоритъма могат да се реализират по един и същ шаблон, но с използването на различна структура за редицата от необходените възли

# Сравнение на двата алгоритъма (2)

- Еднаква сложност по време
  - (ако графът е представен с матрица на съседство, списък на наследниците или списък на ребрата)

# Търсене на път в граф

- Търси се ацикличен път от връх *a* до връх *b*
- Алгоритъм:

  ако *a* == *b* – намерен е път, край

  иначе:

    добави *a* в пътя

    за всеки наследник *c* на *a*, който не се съдържа в пътя:

      ако има път от *c* до *b* (рекурсивно извикване) – намерен е път, край

    ако не е намерен път от никое *c* до *b* – няма път, край.
- Примерна реализация – вж. функцията `findPath`

# Търсене на всички пътища в граф

- Търсим всички пътища от *a* до *b*
- Алгоритъм:

  ако *a* == *b* – намерен е поредният път

  иначе:

    добави *a* в пътя

    за всеки наследник *c* на *a*, който не се съдържа в пътя:

      търси всички пътища от *c* до *b* (рекурсивно извикване)

    премахни *a* от пътя
- Примерна реализация

# Проверка за цикли

- Алгоритъм:

  започваме от произволен връх *a*

  обхождаме (с DFS или BFS)

    ако достигнем обратно ребро: има цикъл

  ако обходим всички върхове: няма цикъл
- Примерна реализация

# Сравнение на DFS и BFS по отношение на търсенето на път

- BFS намира най-краткия път (по брой ребра)
  - За претеглен граф (напр. при търсене на път между градове) съществуват други алгоритми
    - Лесен, некрасив начин: търсим всички пътища и избираме този с най-ниска сума от теглата
- Теоретично, ако графът е безкраен:
  - С DFS може да се тръгне по безкраен път и никога да не се стигне до търсения връх
  - Ако има (краен) път от *a* до *b*, с BFS той гарантирано ще бъде намерен

# Топологично сортиране

- За ориентирани ациклични графи (DAG)
- Цел: да се намери линейна подредба на върховете на даден граф, така че за всяко ребро ($u$,$v$) $u$ е преди $v$
- Може да има няколко решения
- Алгоритъм:
  - Списък $L$ от върхове с $d^-(a)=0$ // няма влизащи дъги
  - За всеки връх $a$ от $L$
    - Обходи $a$
    - Премахни $a$ от графа
    - Добави всички наследници с $d^-(b)=0$ в $L$

# Топологично сортиране (2)

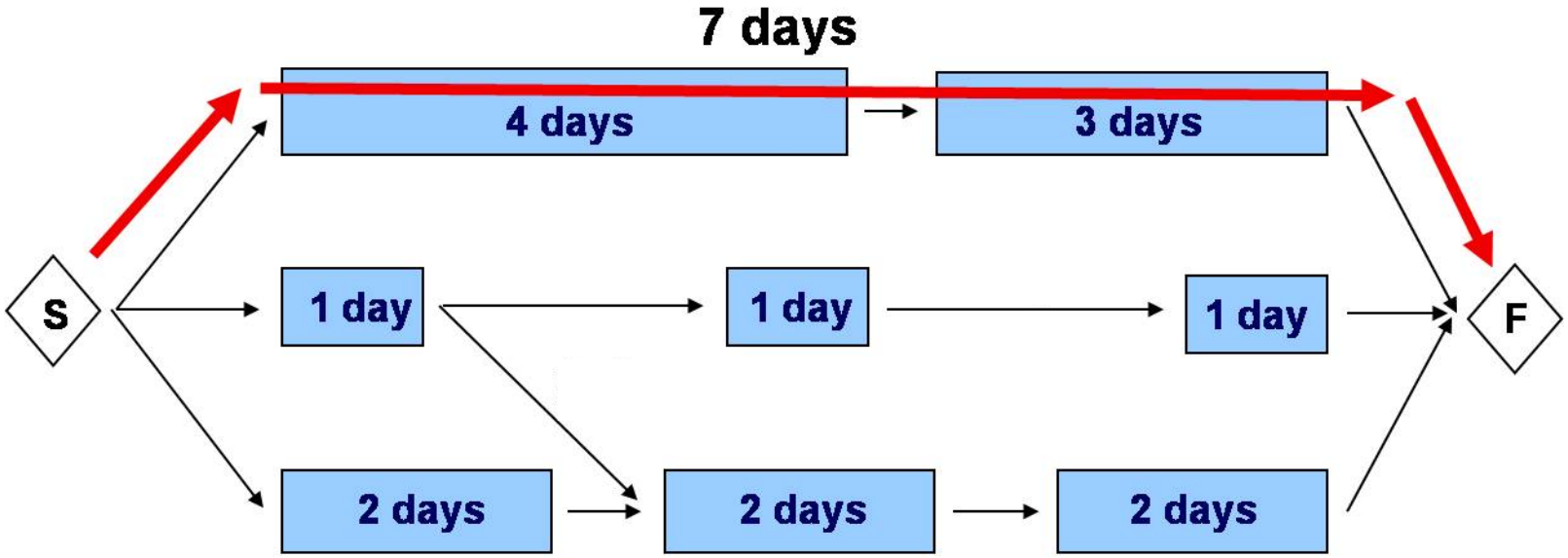
- Примерно приложение: за планиране на дейности, напр. задачата с гишетата в началото на презентацията
  - Дейностите се представят с върхове
  - Ако дейност *a* трябва да се извърши преди дейност *b*, се конструира ребро от *a* до *b*

# Метод на критичния път

- Отново се използва в управлението на проекти
- Дейностите се представят с претеглени ребра или възли (срещат се и двата подхода)
  - Теглото е очакваното време за изпълнение
- Най-дългият път (с макс. сума на теглата) е *критичен* – определя минималното време за изпълнение на проекта
  - Възможно е да има повече от един критичен път
  - Всяко забавяне на дейност от критичен път забавя приключването на целия проект
  - За дейностите, които не се намират на критичния път, е позволено забавяне в определени граници

# Метод на критичния път – пример

- Дейностите са представени с върхове




- Критичният път е с дължина 7 дена

# Обобщение

- Дефиниции
- Начини за представяне
- Примерна реализация
- Обхождане в дълбочина и в ширина
- Търсене на път
- Приложения на граф