



Wrocław  
University  
of Science  
and Technology

# Struktury danych

Wykład 2

## Struktury danych i złożoność obliczeniowa

dr inż. Jarosław Rudy





# Typ danych

## Typ danych

Określony zbiór wartości wraz z określonymi operacjami, które można wykonywać na tych wartościach.

Proste przykłady:

- ▶ Przedział liczb całkowitych wraz z operacjami dodawania, odejmowania, mnożenia i dzielenia (całkowitoliczbowego).
- ▶ Przedział liczb rzeczywistych wraz z operacjami dodawania, odejmowania, mnożenia i dzielenia. Problem zaokrąglania.
- ▶ Przedział liczb naturalnych z operacjami dodawania, odejmowania, mnożenia i dzielenia (całkowitoliczbowego) w systemie z resztą.



# Abstrakcyjny typ danych

## Abstrakcyjny typ danych (ADT)

Opis typu danych uwzględniający własności wartości i wykonywanych na nich operacji bez uwzględnienia (określenia) szczegółów reprezentacji danych i implementacji operacji.

- ▶ ADT opisuje wymagane własności operacji, ale nie sposób ich zapewniania (definiuje interfejs typu danych).
- ▶ ADT opisuje logiczny aspekt danych.
- ▶ ADT powinien dokładnie opisywać wszystkie istotne przypadki.
- ▶ Przykładem ADT jest lista.



# Struktura danych

## Struktura danych (SD)

Opis (format) danych, określający sposób ich reprezentacji, przechowywania i zarządzania nimi.

- ▶ SD stanowi konkretną realizację pewnego ADT (definiuje implementację typu danych).
- ▶ SD opisuje fizyczny aspekt danych, zależny od konkretnej platformy (oprogramowania i sprzętu).
- ▶ Przykładami SD implementującymi listę (ADT) są tablica dynamiczna i lista wiązana.
- ▶ SD zasadniczo dotyczy danych przechowywanych w pamięci operacyjnej (RAM).



# Struktura danych a ADT

- ▶ Czy struktura zbiorów rozłącznych jest strukturą danych czy ADT?
- ▶ Podział na struktury danych i ADT może być płynny.
- ▶ Lista lub słownik opisują tylko operacje i podstawowe własności, są więc ADT.
- ▶ Lista wiązana i tablica dynamiczna opisują konkretną implementację i strukturę w pamięci wykorzystując cechy języka programowania, są więc strukturami danych.
- ▶ Niektóre typy (np. kopiec, drzewo, graf, binarne drzewo poszukiwań) mogą być traktowane zarówno jako ADT jak i struktura danych, zależnie od kontekstu.
- ▶ Istotne jest co z przyjęcia danego ADT/SD wynika oraz które ADT/SD może być użyte do (wydajnej) implementacji których ADT/SD.



# Typowe operacje na ADT/SD

- ▶ Stwierdzenie czy kolekcja jest pusta.
- ▶ Zwrócenie liczby elementów w kolekcji.
- ▶ Wyszukanie elementu:
  - ▶ po pozycji,
  - ▶ po kluczu,
  - ▶ po wartości.
- ▶ Dodanie elementu (też wg pozycji).
- ▶ Usunięcie elementu (wg pozycji, klucza, wartości).
- ▶ Przejrzenie (przejście) przez kolekcję.



## Typy danych – przykłady (1)

Proste typy danych z języków programowania, niskopoziomowo zawsze są reprezentowane za pomocą bitów.

- ▶ Typy stałoprzecinkowe.
  - ▶ Typy liczbowe, z reguły całkowitoliczbowe.
  - ▶ Ze znakiem lub bez.
  - ▶ Reprezentacja w naturalnym kodzie binarnym lub kodzie uzupełnieniowym do 2.
  - ▶ Względnie wysoka precyzja i dokładność wyniku.
  - ▶ Względnie niski zakres wartości, problem nadmiaru.
  - ▶ Dzielenie „całkowitoliczbowe”.
  - ▶ Typy z C/C++: int, long int, unsigned int, short int, czasami też char i boolean.



## Typy danych – przykłady (2)

- ▶ Typy zmiennoprzecinkowe.
  - ▶ Typy liczbowe, odzwierciedlenie liczb rzeczywistych.
  - ▶ Ze znakiem.
  - ▶ Reprezentacja z użyciem 3 liczb binarnych: znaku, ułamka z zakresu [0, 1) i przesuniętego wykładnika.
  - ▶ Duży zakres wartości, dostateczna precyzja.
  - ▶ Dzielenie „rzeczywistoliczbowe” .
  - ▶ Ograniczona dokładność (błąd reprezentacji, zaokrąglenia), nadmiar, niedomiar, NaN, wyjątki, brak łączności.
  - ▶ Typy z C/C++: float, double.



## Typy danych – przykłady (3)

- ▶ Big numbers.
  - ▶ python domyślnie wspiera dowolnie duże liczby całkowite (np. zapis liczb w systemie o podstawie  $2^{30}$  przy użyciu tablicy).
  - ▶ Biblioteka BigNumbers pythona zapewnia dowolnie duże liczby dowolnej precyzji.
- ▶ Typ znakowy (char w C/C++).
  - ▶ Łańcuchy znaków (string).
    - ▶ Realizowane jako tablica.
    - ▶ Ograniczony rozmiar (przechowywany w zerowym elemencie tablicy), stosowane w Pascalu.
    - ▶ Null-terminated string (ASCIIZ, C string). Dowolna długość, łańcuch kończy pierwszy znak null. Powolne liczenie długości.



## Typy danych – przykłady (4)

- ▶ Typ tablicowy (tablice statyczne).
  - ▶ Typ złożony, przechowuje wiele elementów (zwykle identycznego typu) zajmujących ciągły obszar pamięci.
  - ▶ Dostęp przez indeks (arytmetyka wskaźników).
  - ▶ Ustalony rozmiar.
- ▶ Typ strukturalny.
  - ▶ Typ złożony, przechowuje wiele elementów (zajmując ciągły obszar pamięci), elementy mogą być różnego typu.
  - ▶ Dostęp poprzez nazwę elementu.
  - ▶ Typy zawarte w strukturze często są niezmienne (w przeciwieństwie do wartości).
  - ▶ Definiowany przez użytkownika.



## Typy danych – przykłady (5)

- ▶ Typ obiektowy.
  - ▶ Typ złożony rozszerzający typ strukturalny o definicje dozwolonych operacji.
  - ▶ Definiowany przez użytkownika.
  - ▶ Enkapsuluje własny kompletny typ danych (zbiór wartości powiązany z operacjami).
  - ▶ Jeden z paradymatów programowania obiektowego w wielu językach (C++, python, Java, Javascript, C# itd.).
  - ▶ Stanowi podstawę do tworzenia ADT i SD niewspieranych natywnie przez język.



## ADT – przykłady (1)

- ▶ Kolekcja (collection) i kontener (container) – ogólna grupa elementów ze sposobem przechowywania i dostępu.
- ▶ Lista (list) – zachowuje kolejność elementów (kolekcja liniowa). Elementy mogą się powtarzać. Dowolne wstawianie i usuwanie elementów.
- ▶ Zbiór (set) – brak kolejności, elementy nie mogą się powtarzać. Operacje z teorii zbiorów.
- ▶ Multizbiór (multiset) – odpowiednik zbioru, w którym elementy mogą się powtarzać.



## ADT – przykłady (2)

- ▶ Kolejka (queue) – zachowuje kolejność, operacje możliwe tylko na końcach kolejki.
  - ▶ Kolejka First In-First Out (FIFO) – dodawanie na jednym końcu, usuwanie z drugiego końca.
  - ▶ Kolejka Last In-First Out (LIFO), stos – dodawanie i usuwanie elementów na tym samym końcu.
  - ▶ Kolejka dwustronna (double-ended queue, deque) – dodawanie i usuwanie na obu końcach.
  - ▶ Kolejka priorytetowa – kolejność definiowana przez priorytet elementu (elementy o wyższym priorytecie są usuwane pierwsze).



## ADT – przykłady (3)

- ▶ Mapa (map), tablica asocjacyjna (associative table) lub słownik (dictionary)  
– przechowuje elementy w postaci pary klucz-wartość. Klucze nie mogą się powtarzać (klucz ma co najwyżej jedną wartość).
- ▶ Multimapa, multisłownik – odpowiednik mapy, w którym klucze mogą się powtarzać (klucz może mieć wiele wartości).
- ▶ Drzewo (tree) – każdy element ma jeden element rodzica (z wyjątkiem tzw. korzenia), każdy element może mieć potomków (dowolną liczbę).
- ▶ Graf (graph) – dowolne połączenia pomiędzy elementami, z lub bez określania kierunków.



# Struktury danych – przykłady

- ▶ Tablica dynamiczna – tablica o zmiennej liczbie elementów zajmujących ciągły obszar w pamięci. Często stosowana do implementacji wielu ADT (lista, stos, kolejka, drzewo binarne). Dostęp indeksowy (swobodny).
- ▶ Lista wiązana – elementy nie muszą zajmować ciągłego obszaru pamięci. Narzut pamięci. Dostęp sekwencyjny. Używana do implementacji ADT takich jak lista, stos czy kolejka.
- ▶ Tablica mieszająca (hash table) – korzysta z funkcji mieszających. Używana do implementacji słownika.
- ▶ Binarne drzewa poszukiwań, drzewa czerwono-czarne, drzewa AVL – używane do implementacji słowników.
- ▶ Macierz sąsiedztwa, lista sąsiedztwa, lista krawędzi – struktury danych używane do reprezentacji grafów.



# Złożoność obliczeniowa (1)

## Złożoność obliczeniowa

Ilość zasobów potrzebnych algorytmowi do poprawnej pracy.

- ▶ Złożoność obliczeniową określa się dla konkretnego problemu oraz konkretnego algorytmu, programu lub operacji.
- ▶ Najczęściej interesuje nas pojedyncza operacja ADT/struktury danych, ale możliwe jest rozważanie ciągu wielu operacji.
- ▶ Zwykle rozważa się albo konkretny rozmiar problemu (np. konkretny rozmiar listy, drzewa itd.), albo określa się złożoność jako funkcję, której argumentem jest rozmiar problemu.
  - ▶ Ile trwa wyszukanie elementu w liście  $N$  elementów?



## Złożoność obliczeniowa (2)

Zasoby rozpatrywane w złożoności obliczeniowej:

- ▶ Złożoność czasowa.
- ▶ Złożoność pamięciowa.
- ▶ Złożoność komunikacji.
- ▶ Złożoność „równoległa” (liczba procesorów).



# Złożoność czasowa

## Złożoność (obliczeniowa) czasowa

Czas potrzebny do zakończenia algorytmu.

- ▶ Często szacowana liczbą elementarnych operacji.
- ▶ W praktyce występują nie tylko operacje elementarne.
- ▶ Różne operacje zajmują różny czas na różnych maszynach.
- ▶ W praktyce ilość czas rzeczywistego (wall time) potrzebnego do zakończenia algorytmu.
  - ▶ Także CPU time, problem pamięci podręcznej, narzut systemu operacyjnego itd.
  - ▶ W niektórych sytuacjach – liczba operacji odczytu/zapisu pamięci operacyjnej.



# Złożoność pamięciowa (1)

Złożoność (obliczeniowa) pamięciowa

Ilość pamięci (liczba komórek, bajtów itp.) potrzebnych do zakończenia algorytmu.

- ▶ Pamięć wejścia – pamięć potrzebna do zapisania danych wejściowych.
- ▶ Pamięć pomocnicza (dodatkowa) – pamięć potrzebna do zakończenia algorytmu z pominięciem pamięci wejścia.
- ▶ W praktyce często podaje się jedynie pamięć pomocniczą.



## Złożoność pamięciowa (2)

### Algorytm działający w miejscu (in-place)

Algorytm który nie potrzebuje pamięci pomocniczej proporcjonalnej do wielkości danych wejściowych.

Pojęcie nieścisłe, różne definicje praktyczne:

- ▶ Algorytm którego pamięć pomocnicza ma rozmiar co najwyżej stały (tj.  $O(1)$ ).
- ▶ Algorytm którego pamięć pomocnicza ma rozmiar co najwyżej logarytmiczny (tj.  $O(\log N)$ ).
- ▶ Czasami dopuszcza się dodatkową pamięć  $O(N)$ , o ile nie służy do przetwarzania wejścia.



## Scenariusze złożoności (1)

Algorytmy mają różną złożoność dla różnych danych wejściowych. Dla uproszczenia analizy, często zakłada się konkretne scenariusze (przypadki).

- ▶ Przypadek pesymistyczny (worst-case) – dane dla których rozpatrywana złożoność jest największa.
- ▶ Bardzo często wykorzystywany.
- ▶ Gwarancja złożoności.
- ▶ Ograniczona użyteczność (ekstremalne przypadki są zwykle rzadkie).
- ▶ Jeśli nie można ustalić, to można zastąpić górnym ograniczeniem.



## Scenariusze złożoności (2)

- ▶ Przypadek średni/typowy (average-case) – przypadek który jest wartością oczekiwana dla całego rozkładu prawdopodobieństwa możliwych danych wejściowych.
  - ▶ Dość często wykorzystywany i użyteczny.
  - ▶ Często liczba zestawów danych wejściowych jest nieograniczona (nieskończona) – jak wyznaczyć?
- ▶ Przypadek optymistyczny (best-case) – dane dla których rozpatrywana złożoność jest najmniejsza.
  - ▶ Prawie nieużywny (niewielka użyteczność).
  - ▶ Jeśli nie można ustalić, to można zastąpić dolnym ograniczeniem.
- ▶ Inne – np. 95-ty percentyl.



## Asymptotyczne tempo wzrostu

- ▶ Konieczność porównywania złożoności algorytmów dla różnego rozmiaru danych wejściowych.
- ▶ Często trudno jest ustalić dokładną złożoność (np.  $15n^2 + 123n - 64$ ).
- ▶ Dla dużych  $n$  współczynniki (15, 123 i 64 powyżej) często nieistotne.
- ▶ Dla dużych  $n$  mniejsze miany ( $123n$  lub 64) często nieistotne przy większych ( $15n^2$ ).

Powyższe doprowadziło do pojęć asymptotycznego tempa wzrostu i rzędu funkcji złożoności obliczeniowej. Asymptotyczność oznacza, że rozpatruje się jak funkcja zachowuje się gdy  $n \rightarrow \infty$ .



# Notacja dużego O (1)

## Notacja dużego O

Dane są funkcje  $f(x)$  oraz  $g(x)$ . Mówimy, że  $f$  jest rzędu co najwyżej  $g$ , co zapisujemy  $f \in O(g)$ , wtedy i tylko wtedy gdy:

$$\exists_{c \in \mathbb{R} \setminus \{0\}} \quad \exists_{x_0 \in \mathbb{R}} \quad \forall_{x \geqslant x_0} : \quad f(x) \leqslant c \cdot g(x). \quad (1)$$

- ▶ Czyli możemy znaleźć taki (wspólny) ogon obu funkcji (od  $x_0$  „w prawo”) i taką skalę funkcji  $g$ , że w tym ogonie zawsze  $f(x)$  jest nie większa niż przeskalowane  $g(x)$ .
- ▶ Równoważnie można w definicji zapisać  $c \cdot f(x) \leqslant g(x)$ . Skala po prostu się odwraca (np. z  $\frac{7}{2}$  na  $\frac{2}{7}$ ).
- ▶ Często zamiast  $f \in O(g)$  zapisuje się  $f = O(g)$ , ale jest to mylące i nie ma nic wspólnego z matematyczną równością! Z faktu że  $f \in O(g)$  niekoniecznie wynika że  $g \in O(f)$ !



## Notacja dużego O (2)

- ▶  $100n^2 \in O(n^2)$  – współczynnik bez znaczenia.
- ▶  $n^5 + n^3 + n - 5 \in O(n^5)$  – mniejsze miany nie mają znaczenia.
- ▶  $n^2 \in O(n^3)$  – kwadrat jest co najwyżej sześcianem.
- ▶  $n^2 \in O(2^n)$  – pomimo, że na odcinku  $(2, 4)$  to  $n^2$  jest większe.
- ▶  $n^3 \notin O(n^2)$  – jakikolwiek ogon i skalę dobierzemy, sześcian w końcu przegoni kwadrat.



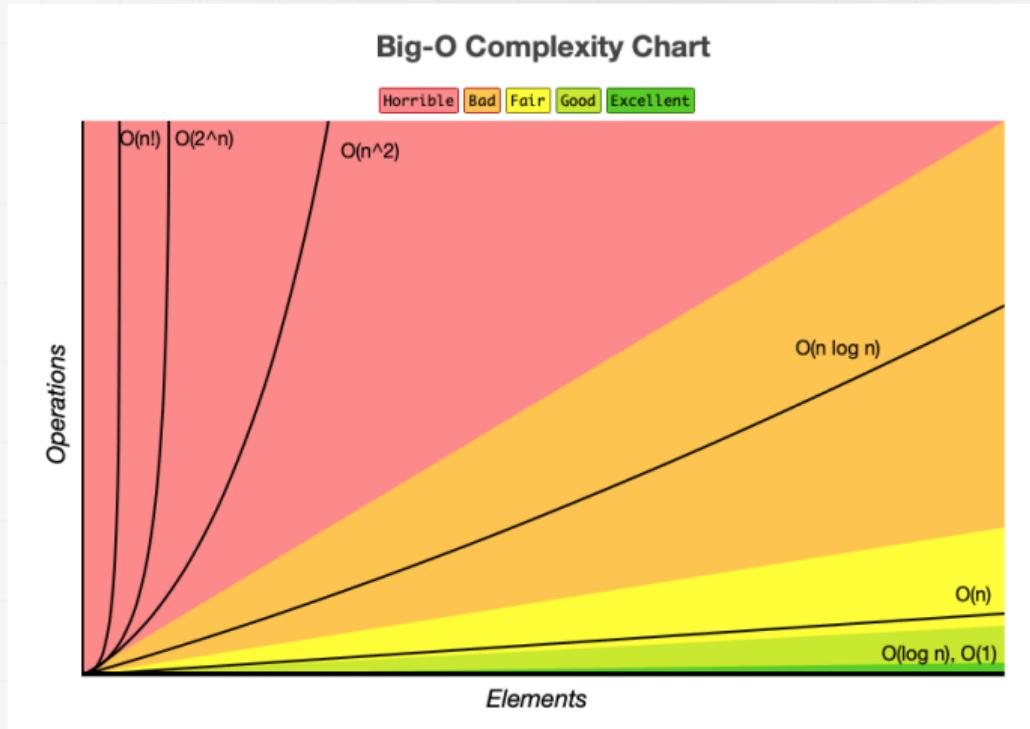
# Istotne rzędy złożoności (1)

Niektórzy rzędy złożoności obliczeniowej (w kolejności rosnącej):

- ▶  $O(1)$  – złożoność stała (tj. ograniczona z góry przez stałą).
- ▶  $O(\log n)$  – złożoność logarytmiczna, dla dużego  $n$  traktowana jak  $O(1)$ .
- ▶  $O(n)$  – liniowa.
- ▶  $O(n^2)$  – kwadratowa. Dla niektórych problemów uznawana za szybką, dla niektórych (np. sortowanie) za powolną.
- ▶  $O(n^3)$  – sześciian.
- ▶  $O(2^n)$  – wykładnicza, zwykle nieakceptowalnie powolna.
- ▶  $O(n!)$  – silnia.



# Istotne rzędy złożoności (2)



Source: [1]



## Duże O – niuanse i praktyka (1)

- ▶ Fakt, że duże O oznacza „co najwyżej” (ograniczenie z góry), nie znaczy, że służy tylko i wyłącznie do opisu najgorszego przypadku!
- ▶ Zadanie: w grupie  $n$  studentów znaleźć jednego nieobecnego.
- ▶ Optymistycznie potrzeba 1 sprawdzenia, średnio  $\frac{1+n}{2}$ , pesymistycznie  $n$ .
- ▶ Każdy z tych scenariuszy można ograniczyć od góry, odpowiednio  $O(1)$ ,  $O(n)$  oraz  $O(n)$ .
- ▶ Każdy można też ograniczyć poprzez np.  $O(n^2)$ , ale zwykle interesuje nas możliwe dokładne ograniczenie.



## Duże O – niuanse i praktyka (2)

- ▶ Czasami dla rzeczywistych rozmiarów danych algorytm o złożoności wyższego rzędu jest praktyczniejszy. Przykładowo:
  - ▶ Złożoność  $2^n$  kontra  $1\,000\,000n^2$ .
  - ▶ Złożoność  $2^n$  kontra  $n^{10}$ .
- ▶ Arytmetyka dużego O (przykłady):
  - ▶  $O(n) + O(n^2) = O(n + n^2) = O(n^2)$ .
  - ▶  $10n^2 - O(n) + 15 = O(n^2) - O(n) + O(1) = O(n^2 - n + 1) = O(n^2)$
  - ▶  $10n^2 - O(n) + 15 = O(10n^2 - n + 15) = O(n^2)$ .
  - ▶  $15n \cdot O(n^3) = O(15n^4) = O(n^4)$ .
  - ▶  $O(\log n) \cdot O(n^2) = O(n^2 \log n)$ .
  - ▶  $(3n + 1)^2 = 9n^2 + O(n)$ .



## Inne notacje

Wszystkie notacje zakładają, że rozważamy odpowiedni ogon funkcji i odpowiednie skale.

- ▶  $f \in O(g)$ , duże O –  $f$  jest co najwyżej rzędu  $g$ .
- ▶  $f \in \Omega(g)$ , duże omega –  $f$  jest co najmniej rzędu  $g$ .
- ▶  $f \in \Theta(g)$ , teta –  $f$  jest rzędu (dokładnie)  $g$ . Implikuje  $f \in O(g)$  oraz  $f \in \Omega(g)$ .
- ▶  $f \in o(g)$ , małe o –  $f$  jest rzędu niższego niż  $g$ . Implikuje  $f \in O(g)$ .
- ▶  $f \in \omega(g)$ , małe omega –  $f$  jest rzędu wyższego niż  $g$ . Implikuje  $f \in \Omega(g)$ .



## Koszt zamortyzowany

- ▶ Różne definicje. Najczęściej wychodzi się od pesymistycznego sumarycznego kosztu ciągu  $k$  operacji (tych samych lub różnych).
- ▶ Z otrzymanej sumy wyciągamy średnią.
- ▶ Ponieważ zakładamy pesymizm każdej operacji, to wynik nie jest tym samym co analiza średniego przypadku (nie opiera się na probabilistyczne).
- ▶ Koszt zamortyzowany pojedynczej operacji w ciągu może być różny (mniejszy) niż koszt tej samej operacji w tym samym ciągu obliczony z użyciem analizy najgorszego przypadku!



## Koszt zamortyzowany – przykład

- ▶ Mamy zakład z  $n$  maszynami. Uruchamiane są razem jednym przyciskiem (jedna czynność), ale raz na  $n$  uruchomień trzeba je najpierw pojedynczo przeglądać ( $n + 1$  czynności).
- ▶ Pesymistyczny czas pojedynczej operacji uruchomienia wynosi więc  $O(n)$ .
- ▶ Rozpatrzmy koszt pojedynczej operacji w ciągu  $n$  kolejnych uruchomień:
  - ▶ Klasyczna analiza najgorszego przypadku:

$$\frac{n \cdot O(n)}{n} = O\left(\frac{n^2}{n}\right) = O(n). \quad (2)$$

- ▶ Koszt zamortyzowany:

$$\overbrace{\frac{1+1+\dots+1+n+1}{n}}^{\text{n-1 razy}} = O\left(\frac{2n}{n}\right) = O(1). \quad (3)$$



# Bibliografia



<https://blog.teclado.com/time-complexity-big-o-notation-python/>



Wrocław  
University  
of Science  
and Technology

# Struktury danych

Wykład 3  
Listy

dr inż. Jarosław Rudy





# Lista (ADT)

- ▶ Lista jest kontenerem przechowującym elementy.
- ▶ Elementy mogą być identycznego lub różnego typu.
- ▶ Lista zachowuje (określa) kolejność elementów.
- ▶ Elementy mogą się powtarzać.
- ▶ Jeden element ma jedną wartość (ale tą wartością może być kolejna kolekcja).
- ▶ Lista ma zmienną długość i zawartość (elementy można dodawać, usuwać i modyfikować).



# Lista – operacje (1)

Na liście typowo rozważa się następujące operacje:

- ▶ Stworzenie pustej listy.
- ▶ Dostęp (zwrócenie) elementu na pozycji  $i$ .
- ▶ Dodanie elementu  $e$  na pozycji  $i$  (tj. przed elementem  $i$ -tym). Specjalne przypadki:
  - ▶ Dodanie elementu  $e$  na początek listy.
  - ▶ Dodanie elementu  $e$  na koniec listy.



## Lista – operacje (2)

- ▶ Usunięcie elementu  $e$  na pozycji  $i$ . Specjalne przypadki:
  - ▶ Dodanie elementu  $e$  na początku listy.
  - ▶ Dodanie elementu  $e$  na końcu listy.
- ▶ Zwrócenie rozmiaru (liczby elementów) listy.
- ▶ Sprawdzenie czy lista jest pusta.
- ▶ Wyszukanie elementu  $e$ .

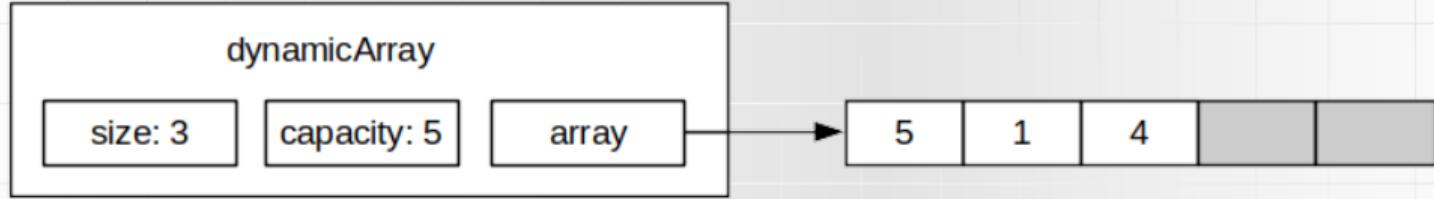


## Tablica dynamiczna (1)

- ▶ Struktura danych będąca implementacją listy (ADT).
- ▶ Najczęściej przechowuje:
  - ▶ array – wskaźnik na tablicę utworzoną dynamicznie (`new`, `malloc()` itp.).
  - ▶ capacity – rozmiar tablicy `array` (w liczbie elementów).
  - ▶ size – liczba przechowywanych elementów.
- ▶ Zajmowana pamięć:  $capacity + 3$ , czyli  $O(capacity)$ .
  - ▶ W praktyce  $capacity < 2n$ , więc zajmowany rozmiar to pomiędzy  $n + 3$  a  $2n + 3$ , czyli  $O(n)$ .



## Tablica dynamiczna (2)



„Pierwsza” implementacja operacji:

- ▶ Stworzenie pustej listy:

- ▶  $array \leftarrow null$
- ▶  $size \leftarrow 0$
- ▶  $capacity \leftarrow 0$
- ▶ czas:  $O(1)$ .



## Tablica dynamiczna (3)

- ▶ Zwrócenie rozmiaru listy:
  - ▶ Zwrócenie *size*.
  - ▶ czas:  $O(1)$ .
- ▶ Sprawdzenie czy lista jest pusta:
  - ▶ Zwrócenie wartości wyrażenia *size == 0*.
  - ▶ czas:  $O(1)$ .
- ▶ Zwrócenie elementu na pozycji  $i$  – w czasie  $O(1)$  dzięki arytmetyce wskaźników (tablica *array* ma dostęp swobodny przez indeks  $i$ ).
  - ▶ Błąd jeśli  $i < 0$  lub  $i \geq size$ .



## Tablica dynamiczna (4)

- ▶ Wyszukanie elementu  $e$ .
  - ▶ Sprawdzamy kolejne elementy, porównując je z  $e$  do czasu znalezienia pasującego elementu (zwrócenie  $e$ , prawdy itp.) lub do wyczerpania elementów (zwrócenie nulla, fałszu itp.).
  - ▶ Optymistycznie: sprawdzany jeden element – czas  $O(1)$ .
  - ▶ Średnio: sprawdzamy połowę elementów tj.  $\lceil \frac{n}{2} \rceil$  – czas  $O(n)$ .
  - ▶ Pesymistycznie: sprawdzamy wszystkie  $n$  elementów – czas  $O(n)$ .
- ▶ Przejrzenie (np. wypisanie wszystkich elementów) również zajmuje czas  $O(n)$ .



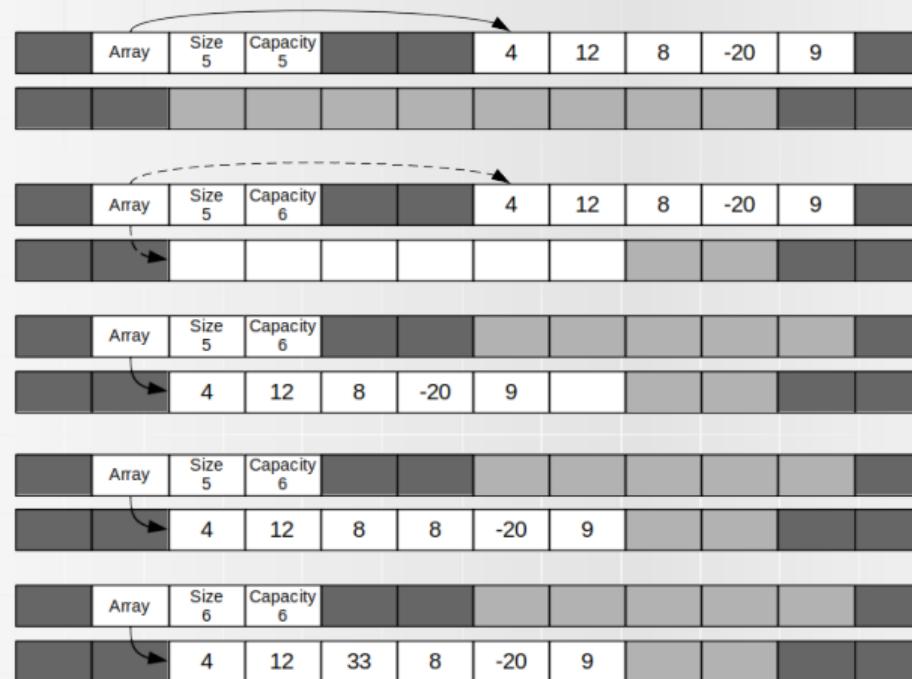
## Tablica dynamiczna (5)

W najprostszej implementacji dodanie elementu  $e$  na pozycji  $i$  składa się z kilku etapów:

- ▶ Zwiększenie rozmiaru tablicy  $array$  o 1.
  - ▶ Funkcja `realloc()` zmienia rozmiar tablicy i może wymagać przeniesienia tablicy do nowej lokalizacji (`memcpy()`).
  - ▶  $capacity \leftarrow capacity + 1$
  - ▶ Czas  $O(1)$  lub  $O(n)$ , zależnie czy kopiowano tablicę.
- ▶ Przeniesienie (np. w pętli) elementów od pozycji  $n - 1$  do  $i$  o jeden w prawo.
  - ▶ Czas wynosi  $O(n - i)$ , pesymistycznie  $O(n)$ .
- ▶ Wstawienie  $e$  na pozycję  $i$  w czasie  $O(1)$ :
  - ▶  $array[i] \leftarrow e$
  - ▶  $size \leftarrow size + 1$
- ▶ Czas całej operacji pesymistycznie i średnio  $O(n)$ , optymistycznie  $O(1)$ .

# Tablica dynamiczna (6)

Przykładowe wstawienie ( $e = 33, i = 2$ ).





## Tablica dynamiczna (7)

- ▶ Dodanie elementu  $e$  na początku ( $i = 0$ ) jest takie samo, ale czas jest zawsze  $O(n)$ , nawet jeśli nie trzeba przenosić tablicy podczas `realloc()`.
- ▶ Dodanie elementu  $e$  na końcu ( $i = n - 1$ ) jest takie samo, ale czas wynosi  $O(1)$  z wyjątkiem sytuacji, gdy `realloc()` przeniesie tablicę.
- ▶ Powyższa podstawowa implementacja operacji dodawania zakłada, że zaczynamy od pustej tablicy ( $capacity = 0$ ) i zwiększamy rozmiar zawsze o 1 (tzn. zawsze  $size = capacity$ ).



## Tablica dynamiczna (8)

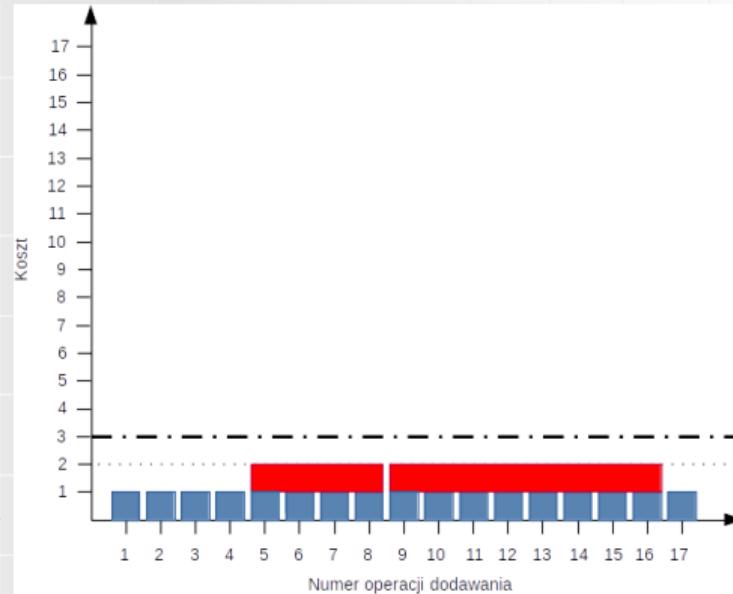
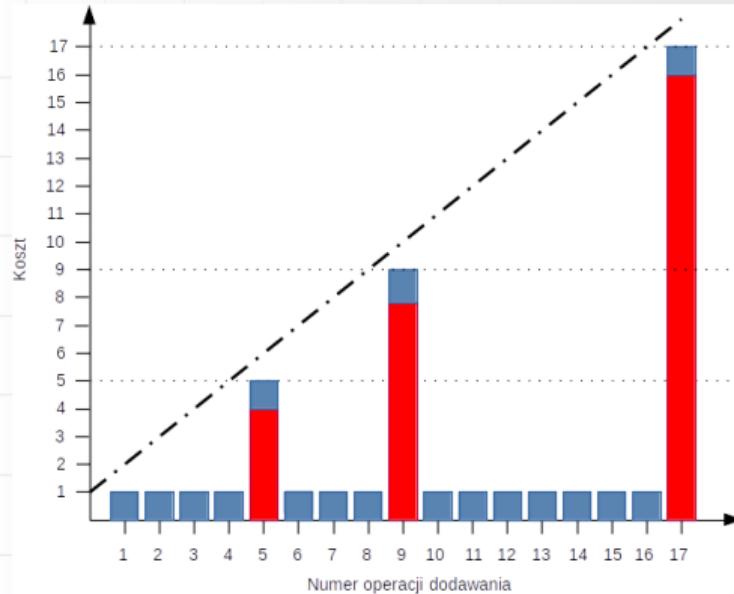
Implementacja ulepszona.

- ▶ Zaczynamy od zaalokowanej, ale pustej tablicy (np.  $size = 0$ ,  $capacity = 4$ ).
- ▶ Przez pierwsze 4 operacje dodawania, nie trzeba będzie kopiować pamięci!
- ▶ Za każdym razem kiedy brakuje miejsca, zwiększamy rozmiar tablicy dwukrotnie ( $capacity = 2size$ ).
- ▶ Podobnie jak poprzednio, przy każdym braku miejsca w tablicy może dojść do konieczności kopowania pamięci... ale brak miejsca zdarza się rzadziej.
- ▶ Ścisłej, jeśli odbywa się kopowanie zajmujące  $O(n)$ , to wiemy, że  $n$  poprzednich operacji dodawania nie wymagało kopowania!
  - ▶ Pesymistycznie jest dalej  $O(n)$ , ale w koszcie zamortyzowanym jest  $O(1)$ !



# Tablica dynamiczna (9)

Klasyczny analiza przypadku pesymistycznego vs koszt zamortyzowany





# Tablica dynamiczna (10)

Klasyczna analiza:

Operacja	Optymistycznie	Średnio	Pesymisycznie
Dodanie na dowolnej pozycji	$O(1)$	$O(n)$	$O(n)$
Dodanie na początku	$O(n)$	$O(n)$	$O(n)$
Dodanie na końcu	$O(1)$	$O(n)$	$O(n)$

Koszt zamortyzowany:

Operacja	Optymistycznie	Średnio	Pesymisycznie
Dodanie na dowolnej pozycji	$O(1)$	$O(n)$	$O(n)$
Dodanie na początku	$O(n)$	$O(n)$	$O(n)$
Dodanie na końcu	$O(1)$	$O(1)$	$O(1)$



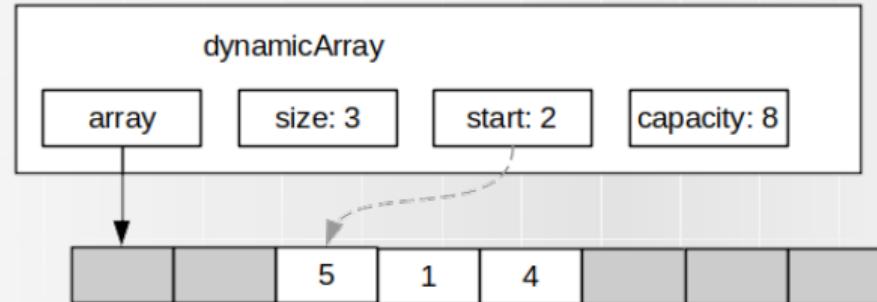
## Tablica dynamiczna (11)

Usuwanie elementu z pozycji  $i$  zasadniczo działa podobnie do dodawania, lecz kolejność jest inna:

- ▶ Skopiowanie elementów od  $i+1$  do  $n-1$  o jedną pozycję w lewo oraz zmniejszenie `size` o 1.
- ▶ Zmniejszenie rozmiaru tablicy (oraz *capacity*). Może następować o 1 lub co jakiś czas (analogicznie do dodawania).
- ▶ Jeśli usuwany element ma zostać zwrócony, należy go zapamiętać przed pierwszym krokiem!
- ▶ Zmniejszanie tablicy dalej może wymagać kopiowania pamięci (zależnie od implementacji `realloc()`).
- ▶ Można pominąć zmniejszanie rozmiaru tablicy, ale rozmiar struktury nie będzie wtedy  $O(n)$ .

## Tablica dynamiczna (12)

- ▶ Czy da się zredukować złożoność  $O(n)$  dla dodawania i usuwania na początku?
- ▶ Zaalokowanie więcej miejsca i przesunięcie indeksu fizycznego względem logicznego – zostaje z przodu miejsce na indeksy „ujemne”.
- ▶ Konieczne dodanie i aktualizacja pozycji początkowej.
- ▶ Alokacja i dealokacja raz na jakiś czas analogicznie jak poprzednio.





# Tablica dynamiczna (13)

Po dodaniu wspomnianego usprawienia:

Operacja	Optymistycznie	Średnio	Pesymisycznie
<code>addFront(e)</code>	$O(1)$	$O(1)$ (amort.)	$O(1)$ (amort.)
<code>removeFront()</code>	$O(1)$	$O(1)$ (amort.)	$O(1)$ (amort.)
<code>addBack(e)</code>	$O(1)$	$O(1)$ (amort.)	$O(1)$ (amort.)
<code>removeBack()</code>	$O(1)$	$O(1)$ (amort.)	$O(1)$ (amort.)
<code>add(e,i)</code>	$O(1)$	$O(n)$	$O(n)$
<code>remove(i)</code>	$O(1)$	$O(n)$	$O(n)$

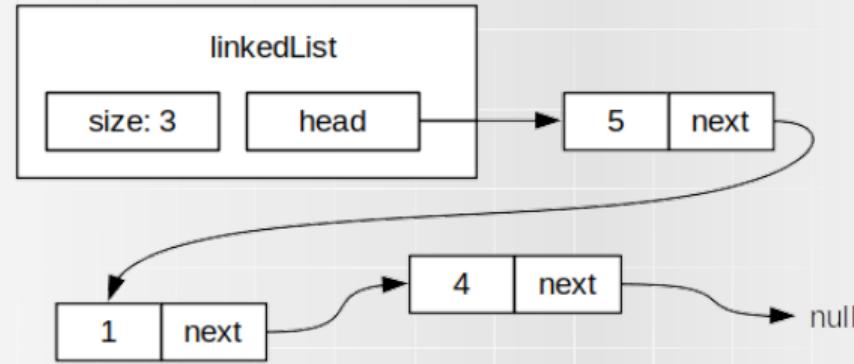


## Lista wiązana (jednokierunkowa)

- ▶ Struktura danych będąca implementacją listy (ADT).
- ▶ Każdy element (węzeł) zawiera:
  - ▶ Właściwą wartość (*value*).
  - ▶ Wskaźnik na kolejny element listy (*next*).
- ▶ Elementy mogą mieć różną lokalizację w pamięci!
- ▶ Lista kończy się, gdy następny wskaźnik ma wartość null.
- ▶ Struktura przechowuje wskaźnik *head* na pierwszy element. Oprócz tego przechowywany jest *size*.
- ▶ Zajmowana pamięć:  $2n + 2$ , czyli  $O(n)$ .



## Lista wiązana (2)



Tworzenie pustej listy (czas  $O(1)$ ):

- ▶  $size \leftarrow 0$
- ▶  $head \leftarrow null$



## Lista wiązana (3)

- ▶ Zwrócenie rozmiaru i sprawdzenie czy lista jest pusta działa identycznie jak dla tablicy dynamicznej.
- ▶ Wyszukanie elementu działa podobnie, lecz by przejść do następnego elementu, należy skorzystać ze wskaźnika *next* poprzedniego (i sprawdzić czy nie jest on null).
- ▶ Zwrócenie elementu na pozycji  $i$  wymaga przejścia przez elementy od 0 do  $i$ . Czas operacji jest więc  $O(i)$ :
  - ▶ Optymistycznie ( $i = 0$ ) mamy  $O(1)$ .
  - ▶ Średnio ( $i = \lceil \frac{n}{n} \rceil$ ) mamy  $O(n)$ .
  - ▶ Pesymistycznie ( $i = n - 1$ ) mamy  $O(n)$ .



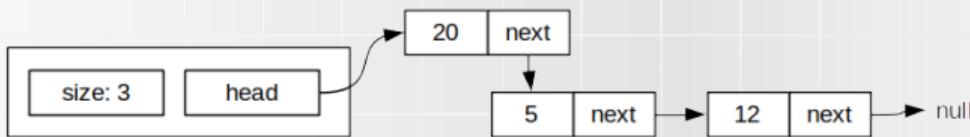
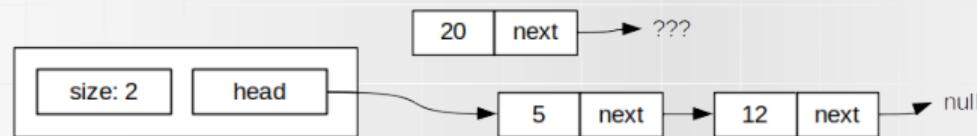
## Lista wiązana (4)

Dodanie elementu  $e$  na początek ( $O(1)$ )

- ▶ Tworzymy (dynamicznie, `new`, `malloc()` itp.) nowy węzeł i zapamiętujemy jego wskaźnik ( $node$ ).
- ▶  $node.value \leftarrow e$
- ▶  $node.next \leftarrow head$
- ▶  $head \leftarrow node$
- ▶  $size \leftarrow size + 1$



# Lista wiązana (5)





## Lista wiązana (6)

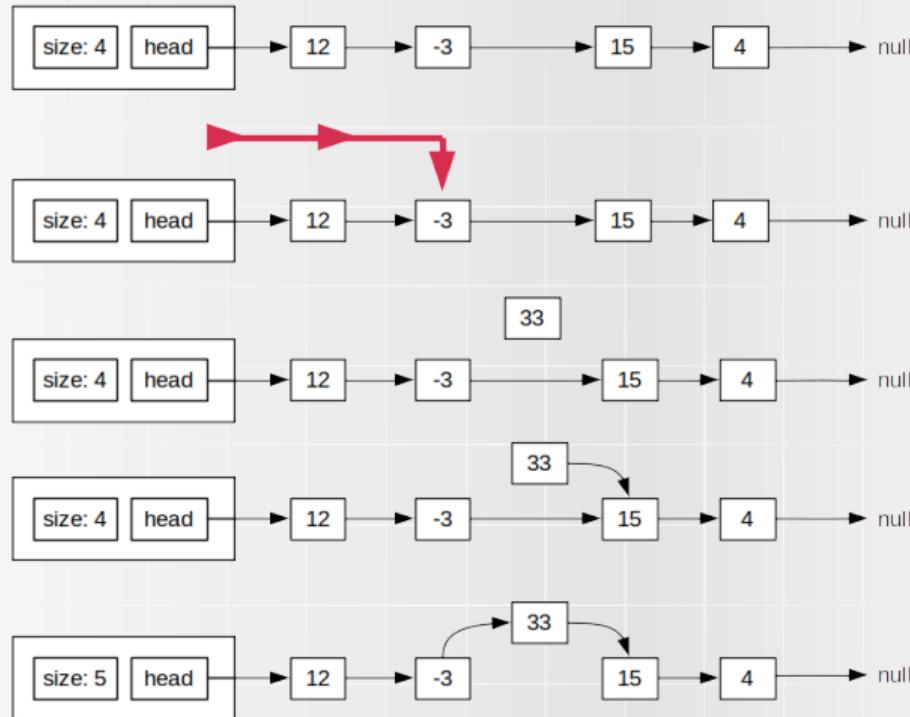
Dodanie elementu  $e$  na pozycji  $i$  (innej niż  $i = 0$  oraz  $i = n - 1$ ).

- ▶ Najpierw musimy dotrzeć do węzła  $i - 1$  (czas  $O(i)$ ), nazwijmy go  $old$ .
- ▶ Tworzymy nowy węzeł  $node$  i przypisujemy mu wartość  $e$ .
- ▶  $node.next \leftarrow old.next$
- ▶  $old.next \leftarrow node$
- ▶  $size \leftarrow size + 1$
- ▶ Optymistycznie  $O(1)$ , średnio i pesymistycznie  $O(n)$ .



# Lista wiązana (7)

Dodanie  $e = 33$  na pozycję 2:





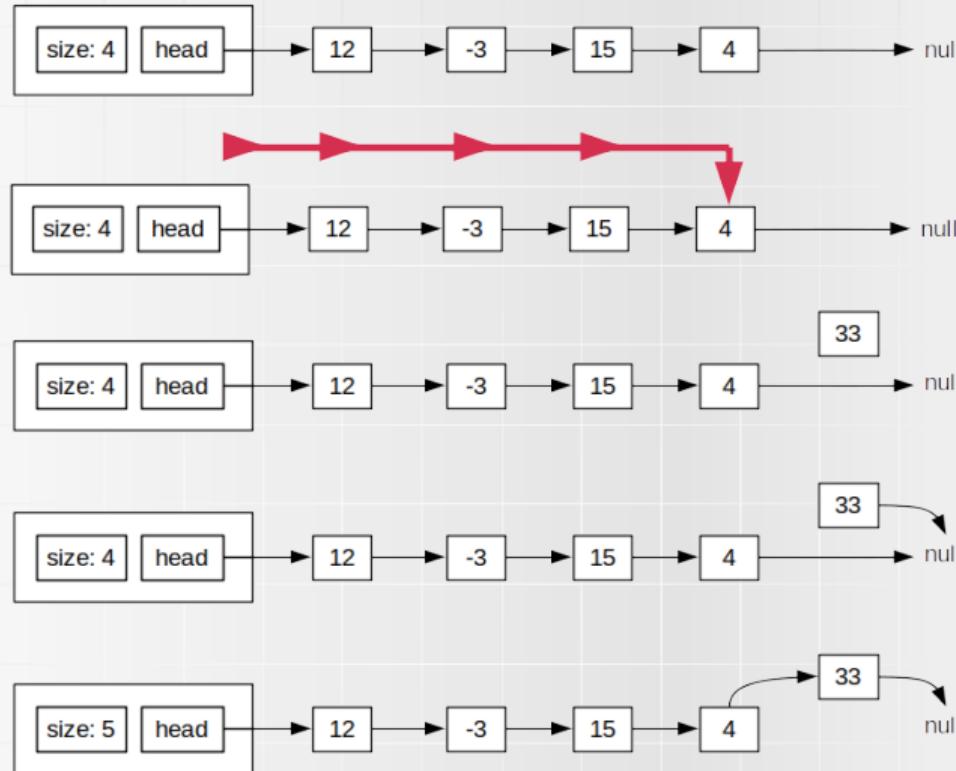
## Lista wiązana (6)

Dodanie elementu  $e$  na ostatniej pozycji ( $i = n - 1$ ).

- ▶ Identycznie jak dodawanie na dowolną pozycję, z tym że  $node.next$  należy ustawić na null.
- ▶ Trzeba dotrzeć do końca listy, więc czas wynosi  $O(n)$ .

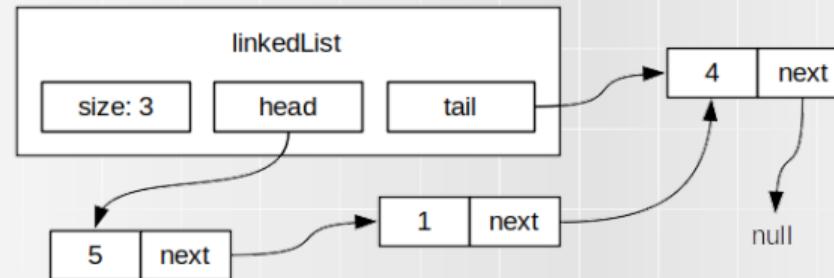


# Lista wiązana (7)



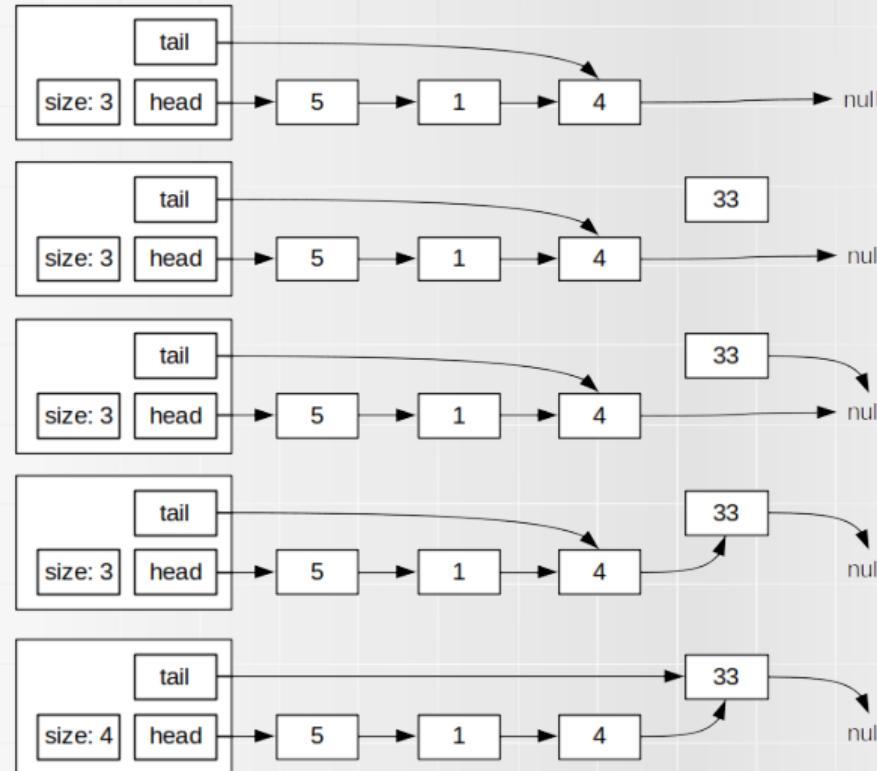
## Lista wiązana (8)

- ▶ Prostym ulepszeniem jest dodanie wskaźnika *tail* wskazującego na koniec listy (ostatni element lub null dla listy pustej).
- ▶ Należy go ustawić na null na początku i pamiętać o ustawieniu gdy zmienia się ostatni element.
- ▶ Umożliwia dodawanie na koniec listy w czasie  $O(1)$ .





# Lista wiązana (9)



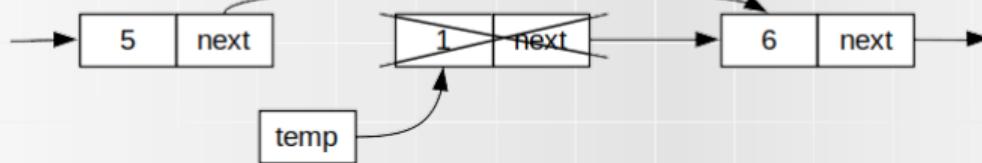
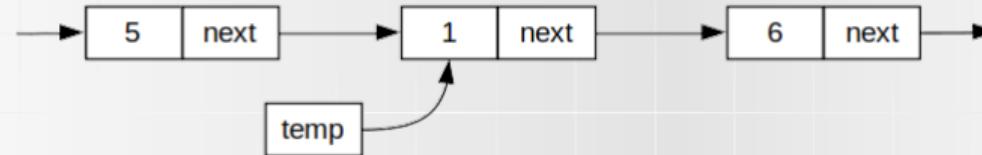
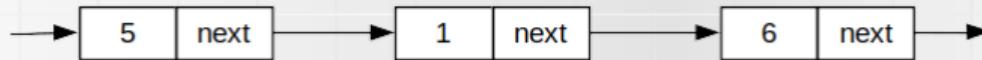


# Lista wiązana (10)

## Usuwanie elementu na pozycji $i$

- ▶ Dotarcie do węzła  $i - 1$  (nazwijmy go  $old$ ).
- ▶  $temp \leftarrow old.next$
- ▶  $old.next \leftarrow old.next.next$
- ▶ Usunięcie węzła  $temp$  (`delete, free()`).
- ▶ Specjalne przypadki:
  - ▶ Usunięcie pierwszego węzła (konieczność modyfikacji `head`).
  - ▶ Usunięcie ostatniego węzła (konieczność modyfikacji `tail`, jeśli jest).

# Lista wiązana (11)





# Lista wiązana (12)

Lista wiązana jednokierunkowa (tylko *head*)

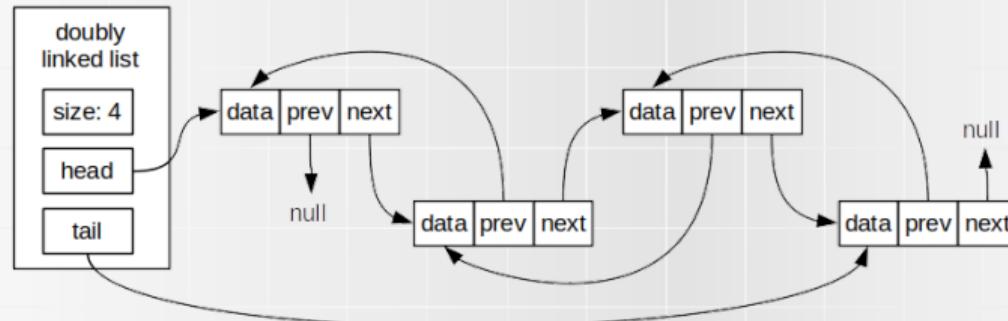
Operacja	Optymistycznie	Średnio	Pesymistycznie
Dodanie na dowolnej pozycji	$O(1)$	$O(n)$	$O(n)$
Dodanie na początku	$O(1)$	$O(1)$	$O(1)$
Dodanie na końcu	$O(n)$	$O(n)$	$O(n)$

Lista wiązana jednokierunkowa (*head* i *tail*)

Operacja	Optymistycznie	Średnio	Pesymistycznie
Dodanie na dowolnej pozycji	$O(1)$	$O(n)$	$O(n)$
Dodanie na początku	$O(1)$	$O(1)$	$O(1)$
Dodanie na końcu	$O(1)$	$O(1)$	$O(1)$

# Lista dwukierunkowa (1)

- ▶ Lista wiązana, gdzie każdy element, ma zarówno wskaźnik na następny element (*next*), jak i na poprzedni (*prev*).
- ▶ Dla ostatniego elementu *next* = *null*.
- ▶ Dla pierwszego elementu *prev* = *null*.
- ▶ Struktura przechowuje *head* i *tail*.



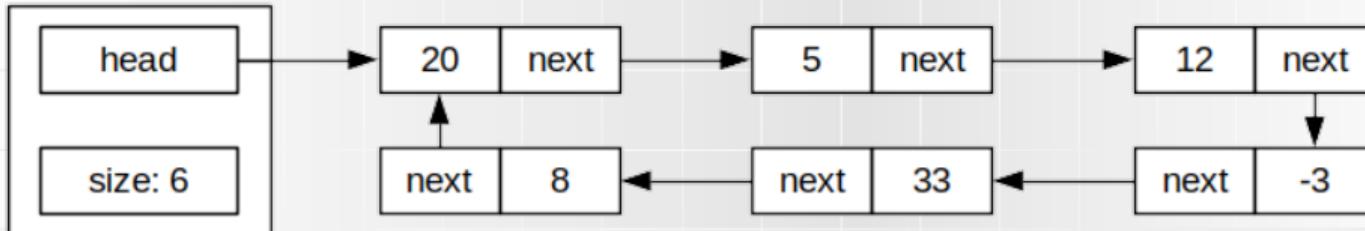


## Lista dwukierunkowa (2)

- ▶ Zajętość pamięci:  $3n + 3$  (wciąż  $O(n)$ , ale narzut jest znaczny).
- ▶ Łatwiejsze przemieszczanie się po liście.
- ▶ Czas dotarcia do węzła  $i$  dwukrotnie mniejszy (choć wciąż  $O(i)$ ).
  - ▶ Prawie dwukrotny mniejszy czas dodawania/usuwania elementów na dowolnej pozycji, szukania oraz przeglądania.
- ▶ Więcej wskaźników do ustawienia podczas operacji.
  - ▶ Nieco wolniejsze dodawanie/usuwanie na końcach.

# Lista cykliczna (1)

- ▶ Ostatni element wskazuje na pierwszy, zamiast na null.
- ▶ Dla listy dwukierunkowej, pierwszy element wskazuje też na ostatni.
- ▶ Wciąż istnieje *head* (inaczej nie można dostać się do listy), ale dowolny element jest początkiem/końcem listy.
- ▶ Koniec listy rozpoznajemy po dotarciu drugi raz do elementu początkowego





## Lista cykliczna (2)

- ▶ Poszukiwanie można zacząć od dowolnego elementu.
- ▶ Przydatna w implementacji niektórych kolejek, buforów cyklicznych czy kopca Fibonacciego
- ▶ Przydatna, gdy przechodzimy po liście wielokrotnie.
- ▶ Bardziej złożona, trudniejsza w kontroli (znalezienie końca, możliwość nieskończonych pętli itp.).



## Rozszerzenia listy wiązanej

Lista z wartownikiem (sentinel):

- ▶ Dodatkowy węzeł (wartownik) przed początkiem/po końcu.
- ▶ Ułatwia obsługę listy.
  - ▶ Każdy wskaźnik można wyłuskać (brak nulla).
  - ▶ Zawsze istnieje jakiś element, nawet jak lista jest pusta.

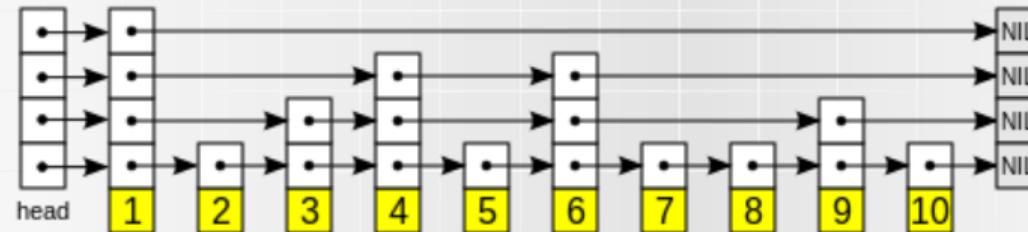
Lista wielokrotnie wiązana:

- ▶ Węzeł posiada więcej wskaźników.
- ▶ Możliwość posiadania różnych kolejności na tych samych danych.



# Lista z przeskokiem

- ▶ Probabilistyczna struktura danych.
- ▶ Lista wielokrotnie wiązana, wiązania mogą pomijać elementy (wg prawdopodobieństwa).
- ▶ Średni czas operacji dodawania/usuwania/wyszukiwania  $O(\log n)$ .
- ▶ Pesymistyczna zajętość pamięci  $O(n \log n)$ .



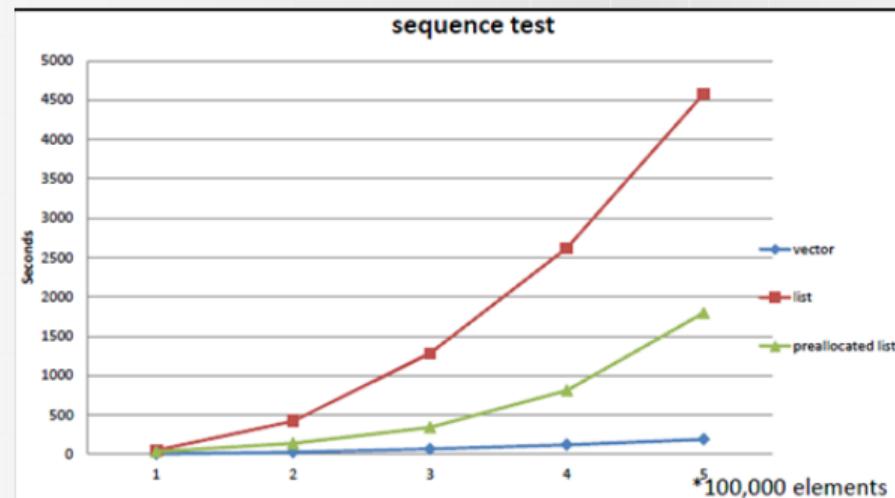


## Tablica dynamiczna vs lista wiązana (1)

- ▶ Tablica dynamiczna ma szybszy czas dostępu do węzła oraz dodatkową pamięć niż lista wiązana –  $O(1)$  vs  $O(n)$ .
- ▶ Lista wiązana ma szybsze dodawanie na początku ( $O(1)$  vs  $O(n)$ ) chyba, że zastosujemy usprawnienie z „ujemnymi” indeksami.
- ▶ Dodawanie na końcu jest w obu przypadkach  $O(n)$ , ale można je ulepszyć.
  - ▶ Tablica dynamiczna – koszt zamortyzowany plus odpowiednio rzadka alokacja pamięci.
  - ▶ Lista wiązana – dodanie wskaźnika na ogon.

## Tablica dynamiczna vs lista wiązana (2)

- ▶ Czy na pewno? Bjarne Stroustrup (twórca C++) przedstawił eksperyment, w którym lista okazała się wolniejsza.
- ▶ Winowiącą jest „niespójna” reprezentacja listy wiązanej w pamięci w połączeniu ze sposobem działania współczesnych pamięci podręcznych (cache).



[1]



## Samoorganizujące się listy (1)

- ▶ Średni czas wyszukiwania na liście (ADT) wynosi  $\lceil \frac{n}{2} \rceil = O(n)$ .
- ▶ Jest to znacznie gorzej niż czas wyszukiwania dla drzew poszukiwań binarnych czy posortowanych list/tablic.
- ▶ Możliwym rozwiązaniem są listy samoorganizujące, które zmieniają kolejność w wyniku kolejnych operacji dostępu/wyszukiwania.
- ▶ Najlepiej sprawdzają się dla sytuacji, gdzie żądania dostępu (lub ich rozkład) znane są z góry.
  - ▶ Zasada 80-20 (20% elementów jest celem 80% wyszukiwań).



## Samoorganizujące się listy (2)

Metoda move-to-front:

- ▶ Element, do którego był dostęp przesuwany jest na początek listy.
- ▶ Proste dla listy wiązanej (dodatkowy czas  $O(1)$ ).
- ▶ Trudniejsze dla tablicy dynamicznej (dodatkowy czas  $O(n)$ ).
- ▶ Względnie prosta implementacja.
- ▶ Podatny na przeszacowanie (przenoszenie na sam już po pierwszym dostępie).



## Samoorganizujące się listy (3)

Metoda transpose (swap):

- ▶ Element, do którego był dostęp przesuwany jest na pozycję o 1 wcześniej.
- ▶ Proste zarówno dla tablicy dynamicznej i listy wiązanej.
  - ▶ Jeśli lista nie jest dwukierunkowa, to należy pamiętać element poprzedni.
- ▶ Przenoszenie elementów do przodu jest stopniowe.
- ▶ Dobrze dostosowuje się do sytuacji, gdzie wzorzec (rozkład prawdopodobieństwa) żądanych elementów zmienia się w czasie.



## Samoorganizujące się listy (4)

Metoda count:

- ▶ Każdy węzeł ma licznik odwołań (ile razy był dostęp).
- ▶ Wymaga dodatkowo  $O(n)$  pamięci.
- ▶ Węzły układane są w kolejności malejącego licznika.
  - ▶ Po odwołaniu wykonuje się tyle swapów ile potrzeba, by uzyskać poprawne sortowanie.
  - ▶ Wyszukiwanie może wydłużyć się o dodatkowe  $O(n)$ , ale średnio będzie szybsze.
  - ▶ Podobnie zwykły dostęp dla listy wiązanej. Dla tablicy dynamicznej dostęp się pogorszy!



# Bibliografia

Wroclaw  
University  
of Science  
and Technology



<https://bulldozer00.blog/2012/02/09/vectors-and-lists/>



Wrocław  
University  
of Science  
and Technology

# Struktury danych

Wykład 4  
Kolejki

dr inż. Jarosław Rudy





## Kolejki (1)

- ▶ Kolejki (queue) to ADT, w którym dodawanie i usuwanie elementów jest ze sobą powiązane tj. usuwany element jest określony przez cechy elementów dodanych do tej pory (ich kolejność, priorytet itp.).
  - ▶ W praktyce kolejki utrzymują pewien (najczęściej liniowy) porządek elementów.
  - ▶ Wtedy dodawanie/usuwanie przekłada się na dodawane/usuwanie na kórymś lub obu końcach kolejki.
- ▶ Podobnie jak dla listy elementy mogą się powtarzać i być różnego typu.
- ▶ Kolejki generalnie mają zmienną długość, ale rozważa się też przypadki o stałym rozmiarze.



## Kolejki (2)

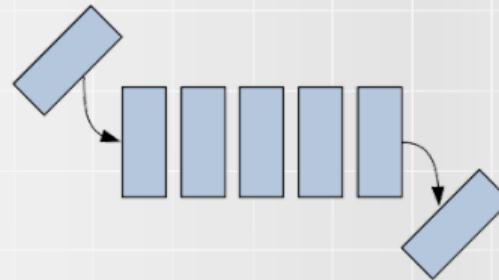
Typowe operacje na kolejkach:

- ▶ Dodanie elementu.
- ▶ Usunięcie elemenu.
- ▶ Sprawdzenie czy kolejka jest pusta (ewentualnie zwrócenie rozmiaru).
- ▶ Podgląd elementu do usunięcia (peek), ale bez usuwania.

Aby dostać się do elementu trzeciego, najpierw należy zdjąć elementy pierwszy i drugi! Kolejki zasadniczo nie służą do dostępu do dowolnego elementu, wyszukiwania czy przeglądania.

# Kolejka FIFO (1)

- ▶ Kolejka działająca wg zasady First-In First-Out (elementy dodany jako pierwszy jest usuwany jako pierwszy).
- ▶ Operacja dodawania nazywa się enqueue, elementy dodawane są do „końca” (back, tail).
- ▶ Operacja zdejmowania nazywa się dequeue, elementy zdejmowane są z „początku” (front, head).
- ▶ Bez dodatkowego kontekstu „kolejka” zwykle oznacza „kolejka FIFO”.





## Kolejka FIFO (2)

Implementacja z użyciem listy wiązanej jednokierunkowej (bez tail):

Operacja kolejki	Operacja listy	Czas pesymistyczny
enqueue( $e$ )	addBack( $e$ )	$O(n)$
dequeue()	removeFront()	$O(1)$
empty()	empty()	$O(1)$
peek()	get(0)	$O(1)$

Albo:

Operacja kolejki	Operacja listy	Czas pesymistyczny
enqueue( $e$ )	addFront( $e$ )	$O(1)$
dequeue()	removeBack()	$O(n)$
empty()	empty()	$O(1)$
peek()	get( $n - 1$ )	$O(n)$



## Kolejka FIFO (3)

Implementacja z użyciem listy wiązanej jednokierunkowej (z tail) lub dwukierunkowej:

Operacja kolejki	Operacja listy	Czas pesymistyczny
enqueue( $e$ )	addBack( $e$ )/addFront( $e$ )	$O(1)$
dequeue()	removeFront()/removeBack()	$O(1)$
empty()	empty()	$O(1)$
peek()	get(0)/get( $n - 1$ )	$O(1)$

Implementacja jest wydajna, ale współdzieli wady listy wiązanej (zajętość pamięci  $2n + O(1)$ , słabsza współpraca z pamięcią podręczną).



# Kolejka FIFO (4)

Implementacja z użyciem tablicy dynamicznej:

Operacja kolejki	Operacja tablicy	Czas pesymistyczny
enqueue(e)	addBack(e)	$O(1)$ (amortyzowany)
dequeue()	removeFront()	$O(1)$ (amortyzowany)
empty()	empty()	$O(1)$
peek()	get(0)	$O(1)$

Albo:

Operacja kolejki	Operacja tablicy	Czas pesymistyczny
enqueue(e)	addFront(e)	$O(1)$ (amortyzowany)
dequeue()	removeBack()	$O(1)$ (amortyzowany)
empty()	empty()	$O(1)$
peek()	get( $n - 1$ )	$O(1)$



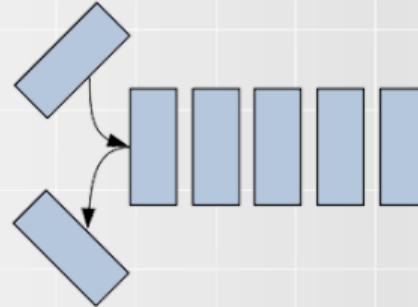
# Kolejka FIFO (5)

Niektóre zastosowania:

- ▶ Bufory.
- ▶ Przechowywanie żądań obsługiwanych w kolejności zgłoszeń.
- ▶ Modelowanie kolejek sklepowych.
- ▶ Przegląd wszerz drzewa.
- ▶ Potoki w unixie.
- ▶ Komunikacja strumieniowa (np. TCP).

# Stos (1)

- ▶ W stosie (ADT) element dodany najpóźniej jest zdejmowany jako pierwszy. Stos jest więc kolejką LIFO (Last-In First-Out).
- ▶ Elementy są więc dodawane i usuwane z tego samego końca (szczytu).
- ▶ Operacja dodawania nazywa się `push()`.
- ▶ Operacja zdejmowania nazywa się `pop()`.
- ▶ Operacja peek często nazywa się `top()`.





## Stos (2)

Implementacja z użyciem listy wiązanej jednokierunkowej (bez tail):

Operacja kolejki	Operacja listy	Czas pesymistyczny
enqueue( $e$ )	addBack( $e$ )	$O(n)$
dequeue()	removeBack()	$O(n)$
empty()	empty()	$O(1)$
top()	get( $n - 1$ )	$O(n)$

Albo:

Operacja kolejki	Operacja listy	Czas pesymistyczny
enqueue( $e$ )	addFront( $e$ )	$O(1)$
dequeue()	removeFront()	$O(1)$
empty()	empty()	$O(1)$
top()	get(0)	$O(1)$



# Stos (3)

Implementacja z użyciem listy wiązanej jednokierunkowej (z tail) lub dwukierunkowej:

Operacja kolejki	Operacja listy	Czas pesymistyczny
enqueue( $e$ )	addFront( $e$ )/addBack( $e$ )	$O(1)$
dequeue()	removeFront()/removeBack()	$O(1)$
empty()	empty()	$O(1)$
top()	get(0)/get( $n - 1$ )	$O(1)$



## Stos (4)

Implementacja z użyciem tablicy dynamicznej:

Operacja kolejki	Operacja tablicy	Czas pesymistyczny
enqueue( $e$ )	addBack( $e$ )	$O(1)$ (amortyzowany)
dequeue()	removeBack()	$O(1)$ (amortyzowany)
empty()	empty()	$O(1)$
top()	get( $n - 1$ )	$O(1)$

Albo:

Operacja kolejki	Operacja tablicy	Czas pesymistyczny
enqueue( $e$ )	addFront( $e$ )	$O(1)$ (amortyzowany)
dequeue()	removeFront()	$O(1)$ (amortyzowany)
empty()	empty()	$O(1)$
top()	get(0)	$O(1)$



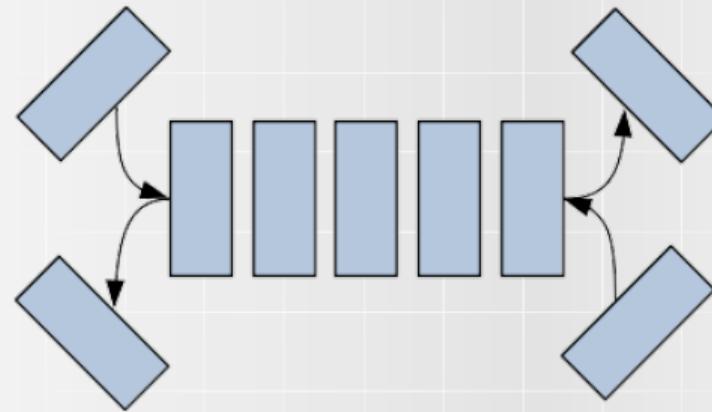
# Stos (5)

Niektóre zastosowania:

- ▶ Parsowanie wyrażeń (np. arytmetycznych) zapisanych w odwrotnej notacji polskiej.
- ▶ Przegląd w głąb drzewa.
- ▶ Algorytmy z nawrotami (backtracking).
- ▶ Stos programowy.

# Kolejka dwukierunkowa (1)

- ▶ Kolejka, w której można dodawać i usuwać elementy na obu końcach.
- ▶ Double-ended que = deque (czyt. „deku”).
- ▶ Nie mylić z operacją dequeue (czyt. „dikju”) kolejki FIFO!





## Kolejka dwukierunkowa (2)

- ▶ Stos i kolejka FIFO mogą być traktowane jako uszczegółowienie deque.
- ▶ Deque może być traktowana jako uszczegółowienie listy.

Operacja	List	Deque	FIFO	Stack
addFront( $e$ )	✓	✓	✗	✓
removeFront()	✓	✓	✓	✓
addBack( $e$ )	✓	✓	✓	✗
removeBack()	✓	✓	✗	✗
add( $e, i$ )	✓	✗	✗	✗
remove( $i$ )	✓	✗	✗	✗



## Kolejka dwukierunkowa (3)

Implementacja z użyciem listy wiązanej jednokierunkowej (bez tail):

Operacja kolejki	Operacja listy	Czas pesymistyczny
addFront( $e$ )	addFront( $e$ )	$O(1)$
addBack( $e$ )	addBack( $e$ )	$O(n)$
removeFront()	removeFront()	$O(1)$
removeBack()	removeBack()	$O(n)$
empty()	empty()	$O(1)$
front()	get(0)	$O(1)$
back()	get( $n - 1$ )	$O(n)$



## Kolejka dwukierunkowa (4)

Implementacja z użyciem listy wiązanej jednokierunkowej (z tail) lub dwukierunkowej:

Operacja kolejki	Operacja listy	Czas pesymistyczny
addFront( $e$ )	addFront( $e$ )	$O(1)$
addBack( $e$ )	addBack( $e$ )	$O(1)$
removeFront()	removeFront()	$O(1)$
removeBack()	removeBack()	$O(1)$
empty()	empty()	$O(1)$
front()	get(0)	$O(1)$
back()	get( $n - 1$ )	$O(1)$



# Kolejka dwukierunkowa (5)

Implementacja z użyciem tablicy dynamicznej:

Operacja kolejki	Operacja tablicy	Czas pesymistyczny
addFront(e)	addFront(e)	$O(1)$ (amortyzowany)
addBack(e)	addBack(e)	$O(1)$ (amortyzowany)
removeFront()	removeFront()	$O(1)$ (amortyzowany)
removeBack()	removeBack()	$O(1)$ (amortyzowany)
empty()	empty()	$O(1)$
front()	get(0)	$O(1)$
back()	get( $n - 1$ )	$O(1)$



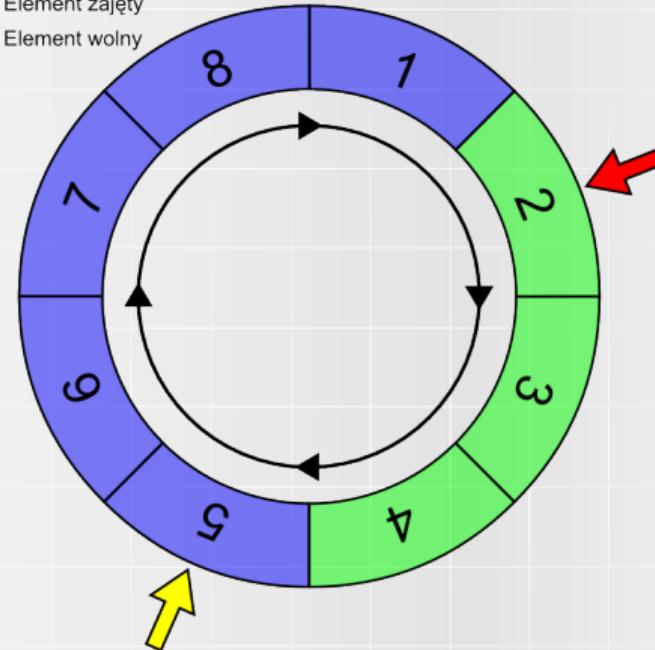
## Kolejka cykliczna (1)

- ▶ Kolejka (bufor) cykliczny jest strukturą danych, zwykle o stałym rozmiarze.
- ▶ Najczęściej działa w charakterze FIFO.
- ▶ Dwa wskaźniki na aktualny początek (odczyt) i koniec (zapis) kolejki.
- ▶ Pozycje wskaźników aktualizowane przy operacjach dodawania i zdejmowania.
- ▶ Stare dane są zapisywane przez nowe.
- ▶ Teoretycznie może służyć do przesyłu dowolnie dużych danych
  - ▶ Gdy jeden wskaźnik dogoni drugi, operacja możliwa dopiero po przesunięciu drugiego wskaźnika.



## Kolejka cykliczna (2)

- Wskaźnik odczytu
- Wskaźnik zapisu
- Element zajęty
- Element wolny





## Kolejka cykliczna (3)

- ▶ Prostota, oszczędność miejsca, szybki dostęp.
- ▶ Ograniczony rozmiar.
  - ▶ Dynamiczne bufory cykliczne.
- ▶ Stosowane np. do implementacji bufora klawiatury.
- ▶ Przy dobrej implementacji czas operacji wynosi  $O(1)$ .
  - ▶ Tablica dynamiczna wymaga „ręcznego” zawijania indeksów.
  - ▶ Lista wiązana działa dzięki pamiętaniu wskaźników.



## Kolejka priorytetowa (1)

- ▶ Kolejka w której każdy element ma przypisany priorytet liczbowy.
  - ▶ Ogólnie, elementy są parą klucz-wartość, gdzie na kluczach da się określić relację maksimum (minimum). Klucze muszą mieć więc częściowy porządek.
  - ▶ Priorytetem może być po prostu wartość elementu.
- ▶ O „kolejności” kolejki decyduje priorytet elementów:
  - ▶ Dodawanie elementu  $e$  o priorytecie  $p$  dodaje go (jakoś) do kolekcji.
  - ▶ Zdejmowanie zawsze zdejmuje element o największym (najmniejszym) priorytecie.
- ▶ Może istnieć wiele elementów o tym samym priorytecie. Różne strategie:
  - ▶ Stabilność – gwarancja zdejmowania takich elementów w kolejności ich dodawania (FIFO).
  - ▶ Niestabilne – brak takiej gwarancji.



## Kolejka priorytetowa (2)

Kolejka priorytetowa typu max:

- ▶  $\text{insert}(e,p)$  – dodanie elementu  $e$  o priorytecie  $p$ .
- ▶  $\text{extract-max}()$  – usunięcie i zwrócenie elementu o największym priorytecie.
- ▶  $\text{find-max}()$  – zwrócenie (podejrzenie) elementu o największym priorytecie.
- ▶  $\text{modify-key}(e,p)$  – zmiana priorytetu elementu  $e$  na  $p$ . Można podzielić na operacje  $\text{decrease-key}$  oraz  $\text{increase-key}$ .

Kolejka priorytetowa typu min jest analogiczna ( $\text{extract-min}$  zwraca element o najmniejszym priorytecie itp).



## Kolejka priorytetowa (3)

- ▶ Pierwszą (naiwną) implementacją kolejki priorytetowej jest wykorzystanie listy, czyli użycie wprost tablicy dynamicznej lub listy wiązanej.
- ▶ Zakładamy, że struktury posiadają odpowiednie usprawnienia, inaczej przedstawione dalej złożoności mogą nie być zawsze spełnione.
- ▶ Implementacja z użyciem listy wiązanej wykorzystuje więcej pamięci i może gorzej współpracować z pamięcią podręczną procesora.
- ▶ Przedstawiona implementacja dotyczy kolejki priorytetowej typu max. Implementacja kolejki typu min jest analogiczna.



## Kolejka priorytetowa (4)

- ▶ Dodajemy elementy na koniec jak w zwykłej kolejce FIFO. Przy zdejmowaniu należy znaleźć największy element.
- ▶ Operacje:
  - ▶  $\text{insert}(e,p)$  – za pomocą  $\text{addBack}(e)$ , czas  $O(1)$  (zwyczajny lub amortyzowany).
  - ▶  $\text{extract-max}()$  – za pomocą przeszukania listy, zwrócenia i usunięcia znalezionej elementu, czas  $O(n)$ .
  - ▶  $\text{peek}()$  – analogicznie do  $\text{extract-max}()$ , ale bez usuwania, czas  $O(n)$ .
  - ▶  $\text{modify-key}(e,p)$  – znalezienie elementu i modyfikacja jego priorytetu, czas  $O(n)$ .



## Kolejka priorytetowa (5)

- ▶ Możemy również odwrócić koncepcję – dodajemy elementy od razu w potrzebne miejsce (jak insert sort), wtedy lista jest posortowana i największy element będzie zawsze na początku.
- ▶ Operacje:
  - ▶  $\text{insert}(e, p)$  – za pomocą wyszukania odpowiedniego miejsca i wstawienia, czas  $O(n)$ .
  - ▶  $\text{extract-max}()$  – zwracamy i usuwamy pierwszy element, czas  $O(1)$  (amortyzowany lub nie).
  - ▶  $\text{peek}()$  – analogicznie do  $\text{extract-max}()$ , ale bez usuwania, czas  $O(1)$  (amortyzowany lub nie).
  - ▶  $\text{modify-key}(e, p)$  – znalezienie elementu, modyfikacja jego priorytetu i przeniesienie elementu w odpowiednie miejsce, czas  $O(n)$ .



Wrocław  
University  
of Science  
and Technology

# Struktury danych

Wykład 5  
Drzewa, kopce

dr inż. Jarosław Rudy





# Drzewa (1)

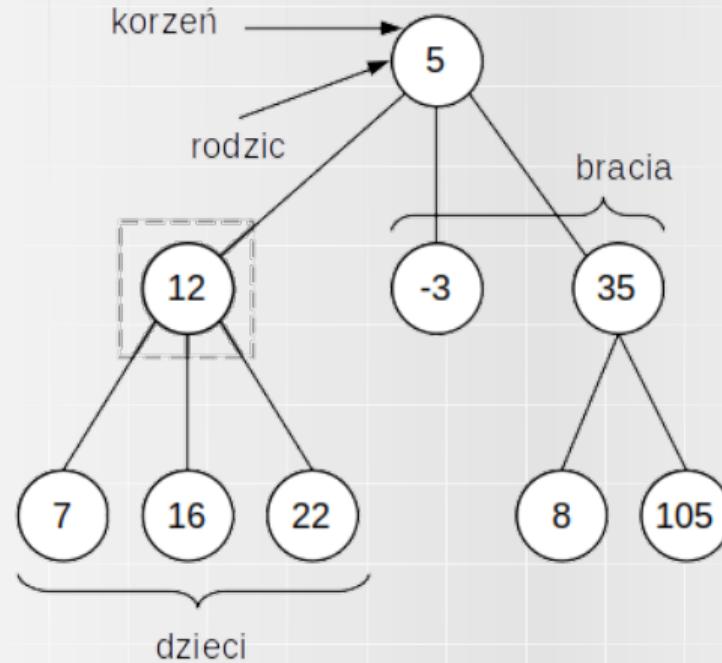
- ▶ Matematycznie drzewo jest zbiorem wierzchołków (węzłów) oraz łączących je krawędzi.
- ▶ Intuicyjnie, drzewo możemy podzielić na poziomy.
- ▶ Na zerowym poziomie jest tylko jeden wierzchołek (zwany korzeniem drzewa) i ma on połączenia jedynie z wierzchołkami na poziomie pierwszym.
- ▶ Wierzchołki na pozostałych poziomach:
  - ▶ Zawsze mają dokładnie jedno połączenie z jednym z wierzchołków na poziomie poprzednim. Wierzchołek ten nazywany jest rodzicem.
  - ▶ Mają dowolną (0 lub więcej) liczbę połączeń z wierzchołkami na poziomie kolejnym. Wierzchołki te nazywa się synami lub dziećmi (rzadziej potomkami).



## Drzewa (2)

- ▶ Analogicznie można określić dla danego węzła dziadka, pradziadka itd. (ogólnie poprzedników) oraz wnuków, prawnuków itd. (ogólniej następców lub potomków).
- ▶ Wierzchołki mającego tego samego rodzica nazywamy rodzeństwem lub braćmi itp.
- ▶ Wierzchołki bez dzieci nazywają się liśćmi.
- ▶ Dużo łatwiej i ściślej można zdefiniować drzewo używając pojęcia grafu – drzewo jest grafem, który jest jednocześnie nieskierowany, spójny i acykliczny.
- ▶ Tradycyjnie drzewa przedstawiane są z korzeniem na górze.
- ▶ Określenie korzenia jest kwestią wyboru – każdy wierzchołek w drzewie może być korzeniem, zaś jego wybór określa relacje pomiędzy wierzchołkami.

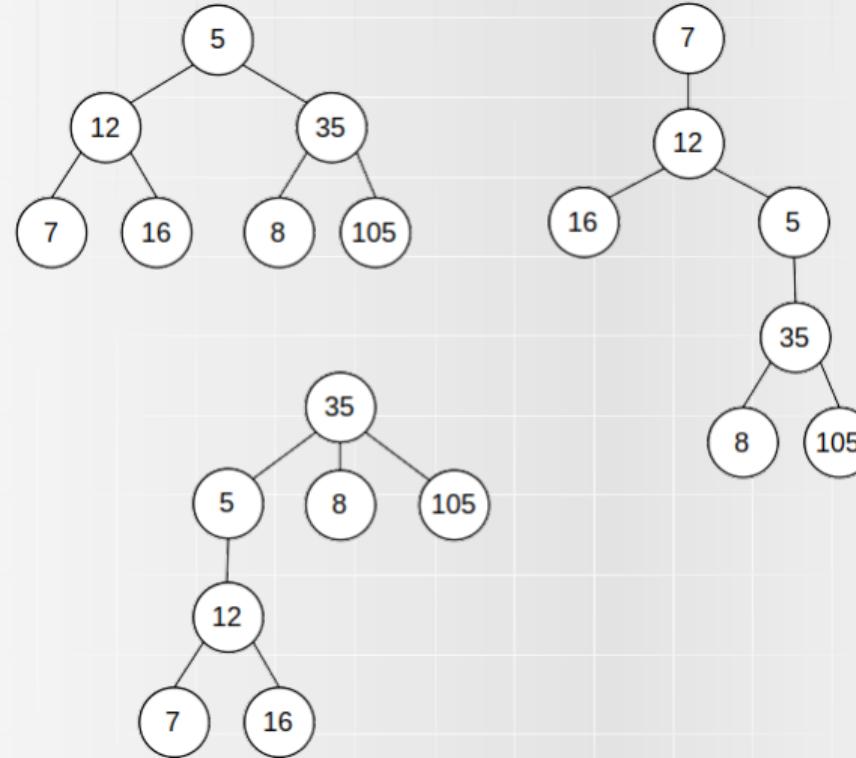
# Drzewa (3)



Wierzchołki 7, 16, 22, -3, 8 i 105 są liśćmi.



# Drzewa (4)





## Drzewa (5)

Drzewo jest ADT czy strukturą danych?

- ▶ Powszechnie drzewo uznawane jest za strukturę danych.
- ▶ Dzieje się tak pomimo faktu, że definicja drzewa nie określa sposobu rozmieszczenia danych w pamięci ani złożoności operacji.
- ▶ Z matematycznego i informatycznego punktu widzenia na drzewie można określić operacje typu dodanie/usunięcie węzła, wyszukanie węzła, rotacja węzła, przeszukanie drzewa (wszerz, wgłąb itp.).
- ▶ Takie ujęcie może wskazywać na charakter ADT.

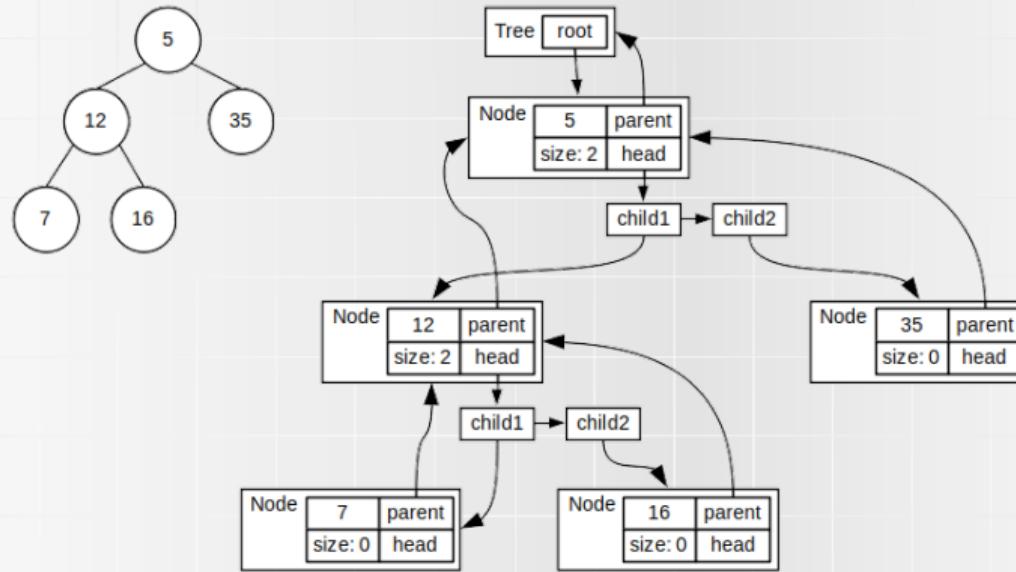


## Drzewa (6)

- ▶ W praktyce istotny jest kontekst.
  - ▶ Jeśli charakterystyka drzewa wynika wprost z zagadnienia (np. genealogia, ciąg decyzyjny), to może być bliżej ADT.
  - ▶ Jeśli charakterystyka drzewa wynika z wyboru (np. implementacja słownika), to może być bliżej strukturze danych.
- ▶ Drzewo często stoi pomiędzy pierwotnym ADT a końcową strukturą danych:
  - ▶ Dodaje dodatkowe założenia względem np. słownika.
  - ▶ Musi być zaimplementowane w konkretnym sposób.

# Drzewa (7)

Drzewo może oznaczać konkretną strukturę np. z użyciem list wiązanych:



Dla wielu drzew (statyczne, zupełne binarne itd.) istnieją dużo lepsze rozwiązania. W szczególności można użyć reprezentacji grafowej.



## Drzewa (8)

- ▶ Ścieżka (czasem też droga) – ciąg krawędzi łączący dwa wierzchołki (bez powtarzania się krawędzi).
- ▶ W drzewie zawsze istnieje jedna ścieżka od korzenia do danego wierzchołka.
- ▶ Liczba krawędzi na ścieżce nazywana jest jej długością.
- ▶ Dla danego wierzchołka  $A$  jego poziom to długość ścieżki od korzenia do  $A$ .
- ▶ Wysokość drzewa to największy z jego poziomów.



# Drzewa (10)

Typowe operacje wykonywane na drzewie:

- ▶ Dodanie elementu.
- ▶ Usunięcie elementu.
- ▶ Wyszukanie elementu.
- ▶ Usunięcie poddrzewa.
- ▶ Rotacja poddrzewa.
- ▶ Przeglądnięcie drzewa (odwiedzenie wszystkich wierzchołków w pewnej kolejności).



## Drzewa (11)

Wrocław  
University  
of Science  
and Technology

- ▶ Drzewo może być puste – brak wierzchołków, a tym samym brak korzenia.
- ▶ W drzewie można rozpatrywać poddrzewa – pewien wierzchołek  $A$  wraz ze wszystkimi lub częścią jego potomków. Wtedy  $A$  jest korzeniem poddrzewa.
- ▶ Zbiór (rozłącznych) drzew nazywany jest lasem.
- ▶ Istnieje wiele szczególnych przypadków drzew (drzewa binarne, BST, czerwono-czarne, ósemkowe itd).



# Drzewa (12)

Zastosowanie drzew:

- ▶ Modelowanie hierarchii (systemy plików, dokumenty HTML/XML, dziedziczenie klas).
- ▶ Procesy decyzyjne (giełda, problem plecakowy, komiwojażera itd.).
- ▶ Rozkład gramatyczny zdań (informatyka i lingwistyka).
- ▶ Implementacja słowników.
- ▶ Implementacja kolejek priorytetowych.



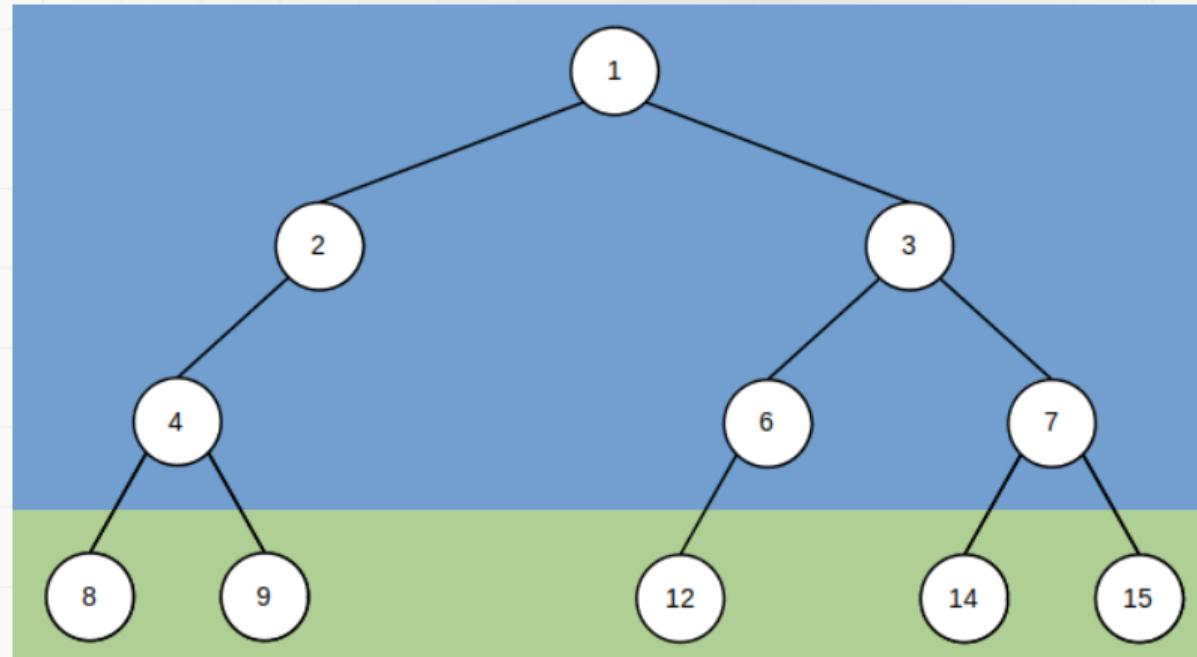
## Drzewa binarne (1)

- ▶ Drzewo binarne – drzewo, w którym każdy węzeł ma co najwyżej dwoje dzieci.
- ▶ Drzewo binarne pełne (full) – drzewo binarne, w którym 1) liście są tylko na ostatnim poziomie, 2) wszystkie nie-liście mają dokładnie 2 dzieci.
- ▶ Drzewo binarne zupełne (complete) – drzewo binarne, w którym 1) ostatni poziom wypełniany jest „od lewej”, 2) wszystkie pozostałe węzły mają dokładnie 2 dzieci (tzn. poziomy oprócz ostatniego są drzewem pełnym).



# Drzewa binarne (2)

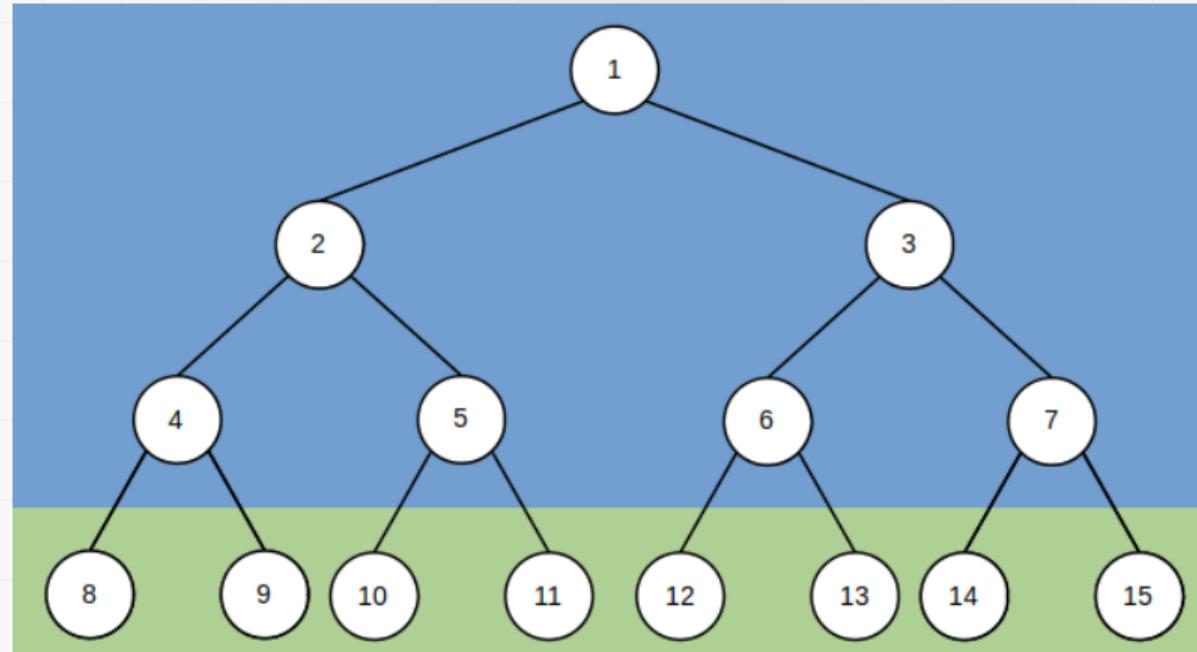
## Drzewo binarne





# Drzewa binarne (3)

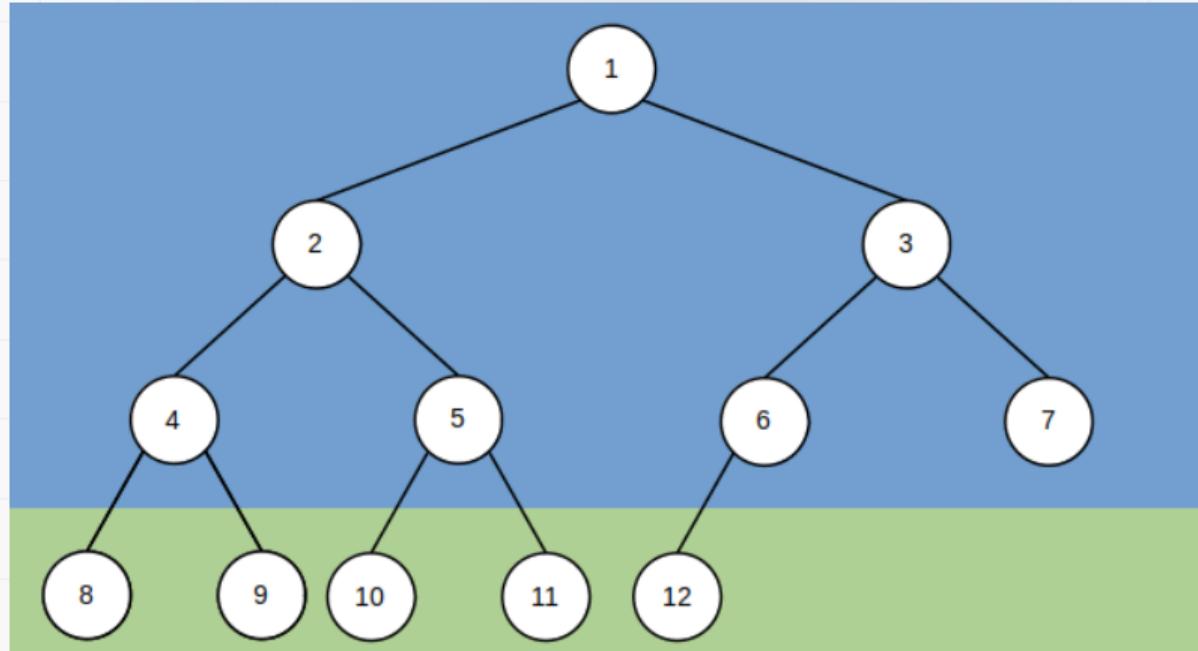
## Drzewo binarne pełne





# Drzewa binarne (4)

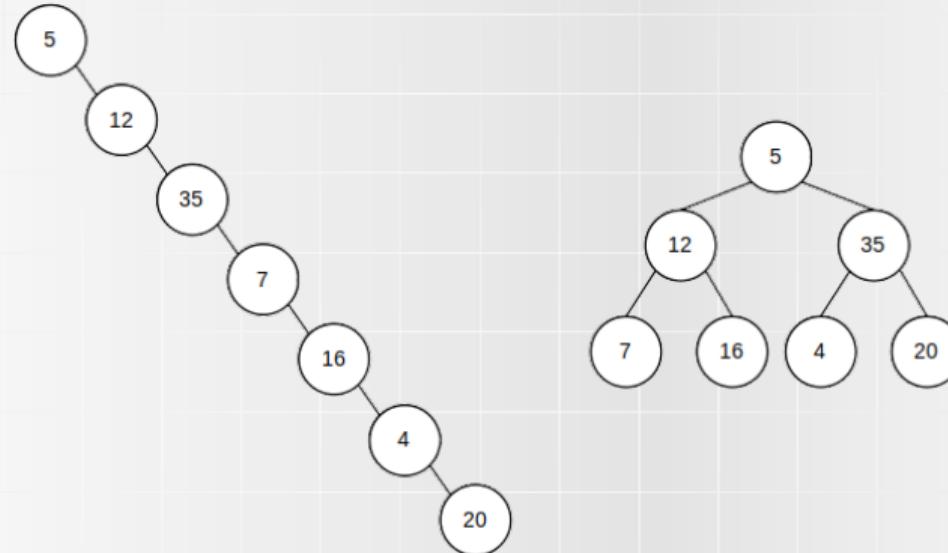
## Drzewo binarne zupełne





## Drzewa binarne (5)

Wysokość drzewa binarnego o  $n$  elementach wynosi  $O(n)$ , ale dla drzewa binarnego zupełnego już tylko  $O(\log n)$ !





## Przejście przez drzewo (1)

- ▶ Przejście przez drzewo oznacza odwiedzenie każdego węzła (zwykle w pewnej kolejności), często wykonując na każdym odwiedzanym węźle pewną czynność (np. wydrukowanie wartości).
- ▶ Istnieje kilka ważnych sposobów przejścia przez drzewo:
  - ▶ Breadth-first (wszerz) – rodzeństwo węzła jest odwiedzane przed jego dziećmi. Odpowiada użyciu kolejki FIFO.
  - ▶ Depth-first (wgłąb) – dzieci węzła są odwiedzane przed jego rodzeństwem. Odpowiada użyciu kolejki LIFO (stosu).
  - ▶ Best-first (najpierw najlepszy) – odwiedzamy węzły wg priorytetu. Odpowiada użyciu kolejki priorytetowej.



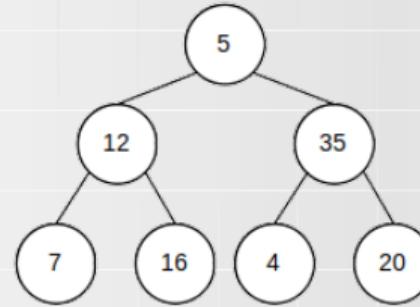
## Przejście przez drzewo (2)

Dla przejścia w głąb można rozważyć kilka wariantów:

- ▶ Pre-order – rodzic jest odwiedzany przed swoimi dziećmi.
- ▶ Post-order – rodzic jest odwiedzany po swoich dzieciach.
- ▶ In-order – najpierw odwiedzany jest lewy syn, potem rodzic, potem prawy syn (dotyczy drzew binarnych).



## Przejście przez drzewo (3)



- ▶ Breadth-first: 5, 12, 35, 7, 16, 4, 20.
- ▶ Depth-first:
  - ▶ Pre-order: 5, 12, 7, 16, 35, 4, 20.
  - ▶ Post-order: 7, 16, 12, 4, 20, 35, 5.
  - ▶ In-order: 7, 12, 16, 5, 4, 35, 20.
- ▶ Best-first: 5, 35, 20, 12, 16, 7, 4.



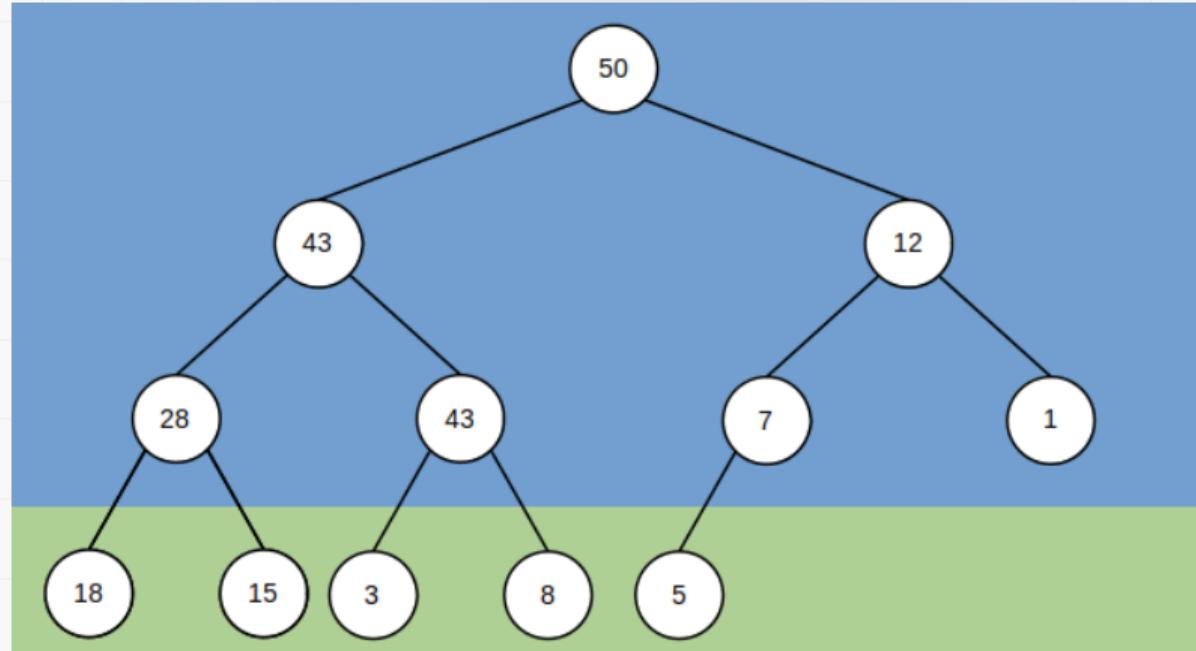
## Kopce (1)

- ▶ Kopiec (heap) – drzewo w którym zachowana jest zasada kopca tzn. klucz rodzica jest w stałej relacji z kluczami jego dzieci. Najczęściej rozróżniamy:
  - ▶ Max heap: rodzic jest nie mniejszy od swoich dzieci.
  - ▶ Min heap: rodzic jest nie większy od swoich dzieci.
- ▶ W efekcie każda ścieżka od korzenia do liścia jest posortowana (kopiec jest częściowo posortowany).
- ▶ Jeśli węzły nie mają klucza, to kluczem staje się sama wartość.
- ▶ Zbiór kluczy musi mieć określony częściowy porządek.
- ▶ Kopce również mogą być binarne, pełne i zupełne.



# Kopce (2)

Kopiec binarny zupełny typu max





## Kopce (3)

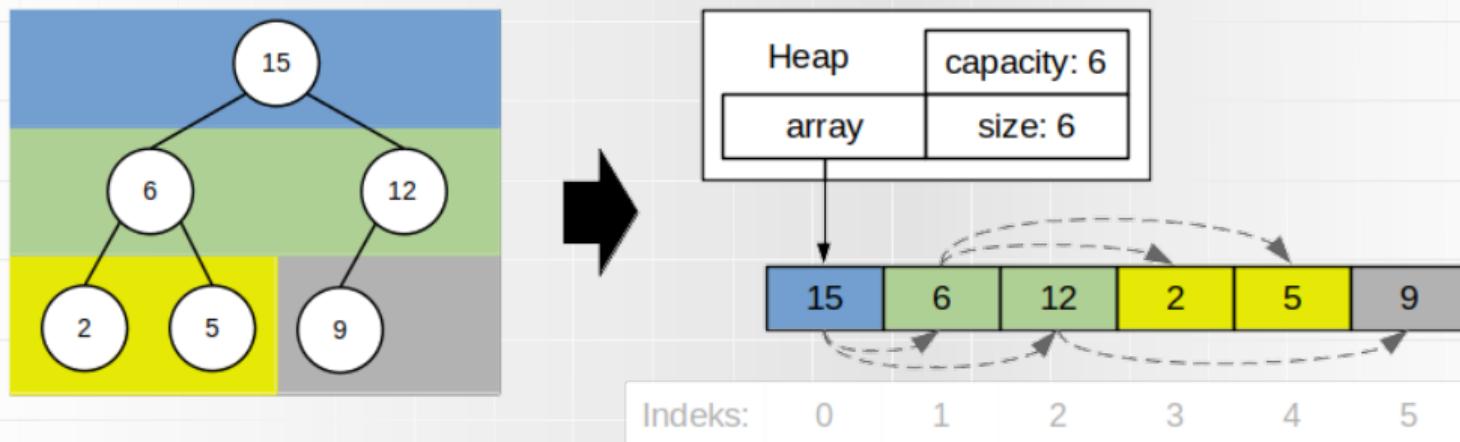
Kopiec binarny zupełny można wydajnie i prosto zaimplementować z użyciem tablicy dynamicznej:

- ▶ Przyjmując numerację od 0, jeśli rodzic ma indeks  $k$  w tablicy, to jego dzieci mają indeksy  $2k + 1$  oraz  $2k + 2$ .
- ▶ Jeśli węzeł ma indeks  $k$ , to jego rodzic ma indeks  $\lfloor \frac{k-1}{2} \rfloor$  (arytmetyka całkowitoliczbową).
- ▶ Ponieważ kopiec jest zupełny (ostatni poziom wypełniany od lewej do prawej), to tablica wykorzystywana jest w pełni bez dziur (nie licząc przestrzeni zaalokowanej na przyszłość).
- ▶ Znając indeks, dostęp do węzła jest w czasie  $O(1)$ .



# Kopce (4)

Przykład implementacji (binarnego zupełnego) kopca za pomocą tablicy dynamicznej:





## Kopce (5)

Operacje kopcowe (kopiec typu max):

- ▶ `insert()` – dodanie elementu.
- ▶ `extract-max()` – usunięcie i zwrócenie elementu o największym kluczu.
- ▶ `find-max()` – zwrócenie elementu o największym kluczu.
- ▶ `find()` – szukanie elementu o danej wartości/kluczu.
- ▶ `delete()` – usunięcie elementu o danej wartości.
- ▶ `decrease-key()/increase-key()` – zmiana wartości klucza elementu.
- ▶ `build()` – zbudowanie kopca z  $n$  elementów.



# Kopce (6)

Pomocnicze operacje do przywracania własności kopca.

► heapify-up

1. Zaczynamy od danego elementu  $e$ .
2. Jeśli  $e$  jest korzeniem lub jest w prawidłowej relacji ze swoim rodzicem – koniec algorytmu.
3. W przeciwnym razie (złamana własność kopca) zamień  $e$  miejscami ze swoim rodzicem i wróć do kroku 2.

► Pesymistyczny czas zależny od wysokości drzewa, czyli  $O(\log n)$ .



# Kopce (7)

- ▶ heapify-down
  1. Zaczynamy od danego elementu e.
  2. Jeśli e jest liściem lub jest w prawidłowej relacji ze swoimi dziećmi – koniec algorytmu.
  3. W przeciwnym razie (złamana własność kopca) zamień e miejscami ze swoim większym dzieckiem i wróć do kroku 2.
- ▶ Dla kopca typu min analogicznie (zamieniamy z mniejszym dzieckiem).
- ▶ Pesymistyczny czas  $O(\log n)$ .



# Kopce (8)

## ► insert()

- ▶ Wstawiamy element na pierwsze wolne miejsce (na ostatnim poziomie lub tworzymy nowy poziom od lewej).
- ▶ Wykonujemy heapify-up() na dodanym węźle.
- ▶ Czas  $O(\log n)$  (zamortyzowany dla tablicy dynamicznej).

## ► extract-max()

- ▶ Wstawiamy ostatni element w miejsce korzenia (zapamiętując stary korzeń).
- ▶ Wykonujemy heapify-down() na nowym korzeniu.
- ▶ Zwracamy zapamiętany stary korzeń.
- ▶ Czas  $O(\log n)$ .



## Kopce (9)

- ▶ `find-max()` – zwracamy korzeń, czas  $O(1)$ .
- ▶ `find()` – przeszukanie (przejście) drzewa, pesymistycznie  $O(n)$ .
- ▶ `delete()`
  - ▶ Znalezienie węzła  $e$  do usunięcia.
  - ▶ Zamienienie węzła miejscami z ostatnim.
  - ▶ Zmniejszenie rozmiaru tablicy.
  - ▶ Wykonanie `heapify-up()` lub `heapify-down()` do przywrócenia własności kopca (jeśli potrzebne).
  - ▶ Pesymistyczny czas  $O(n)$ .



# Kopce (10)

- ▶ increase-key()
  - ▶ Znajdź węzeł  $e$  do zmiany.
  - ▶ Zwiększ klucz.
  - ▶ Wykonaj heapify-up() na  $e$  (zakładając kopiec typu max).
- ▶ decrease-key()
  - ▶ Znajdź węzeł  $e$  do zmiany.
  - ▶ Zmniejsz klucz.
  - ▶ Wykonaj heapify-down() na  $e$  (zakładając kopiec typu max).



# Kopce (11)

Wrocław  
University  
of Science  
and Technology

- ▶ Jedna z operacji dla wielu struktur danych jest wypełnienie pustej struktury (dodanie po kolei wielu elementów).
- ▶ Dla  $n$  elementów zwykle polega to na  $n$ -krotnym wykonaniu operacji typu `insert()`.
- ▶ Dla kopca (binarnego zupełnego) nazywamy to budowaniem kopca.
- ▶ Naiwne podejście oznacza czas  $n \cdot O(\log n) \in O(n \log n)$ .
- ▶ Istnieje jednak szybszy sposób.



## Kopce (12)

- ▶ Najpierw dodajemy wszystkie elementy w dowolny sposób, nie dbając o właściwość kopca.
  - ▶ Najprościej jest zaalokować miejsce na  $n$  elementów i umieszczać element  $i$ -ty w indeksie  $i$ .
- ▶ Następnie w pętli dla indeksów od  $\lfloor \frac{n-1}{2} \rfloor$  do 0 wykonujemy heapify-down (zakładając kopiec typu max).
  - ▶ Procedura działa od dołu w górę (bottom-up).
  - ▶ Elementy na indeksach dalej niż  $\lfloor \frac{n-1}{2} \rfloor$  są liścimi – kopcowanie ich nie ma sensu.
- ▶ Z pomocą pewnego zbieżnego szeregu można wykazać, że taka operacja zajmuje pesymistycznie i średnio czas  $O(n)$ .



# Kopce (13)

Zastosowanie kopków:

- ▶ Sortowanie przez kopcowanie.
- ▶ Algorytmy selekcji.
- ▶ Kolejki priorytetowe.
- ▶ Algorytmy grafowe.
  - ▶ Algorytm Dijkstry.
  - ▶ Algorytm Prima.



# Kopce (14)

Kopce umożliwiają wydajną implementację kolejek priorytetowych:

- ▶ `insert()` –  $O(\log n)$ .
- ▶ `extract-max()` –  $O(\log n)$ .
- ▶ `find-max()` –  $O(1)$ .
- ▶ `modify-key()` – wciąż koszt  $O(n)$  (konieczność znalezienia węzła).
  - ▶ Możemy dodatkowo przechowywać słownik mapujący indeks na węzeł.
  - ▶ Koszt operacji spada do  $O(\log n)$ , ale stała wzrasta.
  - ▶ Słownik oznacza konieczność dodatkowej pamięci.



Wrocław  
University  
of Science  
and Technology

# Struktury danych

Wykład 6

Kopiec Fibonacciego, struktura zbiorów rozłącznych

dr inż. Jarosław Rudy





## Koszt amortyzowany – metoda księgowania (1)

- ▶ Jedną z alternatywnych metod analizy kosztu amortyzowanego jest metoda księgowania.
- ▶ Koszt zamortyzowany operacji oprócz kosztu rzeczywistego uwzględnia (nieujemny) „kredyt”.
- ▶ Jeśli operacja ma koszt zamortyzowany większy niż rzeczywisty, to zwiększa kredyt.
- ▶ Operacja może mieć koszt zamortyzowany mniejszy niż rzeczywisty, o ile w momencie jej wykonywania mamy wystarczający kredyt.
- ▶ Ponieważ kredyt jest nieujemny, obliczony w ten sposób koszt zamortyzowany ogranicza od góry koszt rzeczywisty.



## Koszt amortyzowany – metoda księgowania (2)

Przykład dodawania elementu na koniec tablicy dynamicznej:

- ▶ Każda operacja dodawania daje 1 kredyt.
- ▶ Rozważmy ciąg  $k + 1$  operacji:
  - ▶ Pierwsze  $k$  operacji nie wymaga rozszerzenia tablicy. Każda ma koszt rzeczywisty 1 i zamortyzowany 2, łącznie gromadząc kredyt wysokości  $k$ .
  - ▶ Ostatnia operacja musi rozszerzyć tablicę, więc jej rzeczywisty koszt to  $k + 1$  (czyli  $O(k)$ ).
    - ▶ Jak każde dodawanie, operacja przydziela 1 kredyt... ale możemy wykorzystać zgromadzone  $k$  kredytu.
    - ▶ Ostatecznie koszt zamortyzowany to  $1 + 1 + k - k = 2 \in O(1)$ .
  - ▶ Każda operacja w ciągu ma więc koszt zamortyzowany równy  $O(1)!$



## Koszt amortyzowany – metoda potencjału (1)

- ▶ Strukturze danych przypisujemy potencjał.
- ▶ Potencjał jest funkcją od stanu struktury.
- ▶ Potencjał mogą posiadać fragmenty struktury, potencjał całości jest wtedy sumą potencjałów części.
- ▶ Koszt zamortyzowany operacji  $i$  oblicza się jako koszt rzeczywisty plus różnica potencjałów (potencjał po operacji  $i$  i po operacji  $i_1$ ):

$$k_{\text{amortized}} = k_{\text{real}} + \Phi_i - \Phi_{i-1}. \quad (1)$$



## Koszt amortyzowany – metoda potencjału (2)

Przykład dodawania elementu na koniec tablicy dynamicznej:

- ▶ Zdefiniujmy potencjał tablicy jako  $\text{size} - \text{capacity}$ . W tym przypadku potencjał może być ujemny, ale niczemu to nie przeszkadza.
- ▶ Rozważmy ciąg  $k + 1$  operacji:
  - ▶ Każda z pierwszych  $k$  operacji ma koszt rzeczywisty równy 1, a potencjał wzrasta o 1.
    - ▶ Koszt zamortyzowany wynosi więc 2.
  - ▶ Ostatnia operacja ma koszt rzeczywisty  $k+1$ , ale potencjał spada o  $k-1$ .
    - ▶ Koszt zamortyzowany wynosi więc 2.
  - ▶ Każda operacja w ciągu ma więc koszt zamortyzowany równy  $O(1)$ !



## Koszt amortyzowany – metoda potencjału (3)

$i$	capacity	size	$\Phi_i$	$\Phi_i - \Phi_{i-1}$	$k_{\text{real}}$	$k_{\text{amortized}}$
init	4	0	-4	n/d	n/d	n/d
1	4	1	-3	1	1	2
2	4	2	-2	1	1	2
3	4	3	-1	1	1	2
4	4	4	0	1	1	2
5	8	5	-3	-3	$4 + 1 = 5$	2
6	8	6	-2	1	1	2
7	8	7	-1	1	1	2
8	8	8	0	1	1	2
9	16	9	-7	-7	$8 + 1 = 9$	2



# Kopiec Fibonacciego (1)

- ▶ Kopiec binarny pozwala na wydajną realizację kolejki priorytetowej.
- ▶ Najważniejsze operacje (`insert`, `find-min`, `extract-min`, `decrease-key`) mają złożoność  $O(\log n)$ .
- ▶ Jednak tylko operacja `find-min` ma czas  $O(1)$ .
- ▶ Lepszą złożoność teoretyczną oferuje kopiec Fibonacciego.
- ▶ Dalej opisujemy kopce typu min.



## Kopiec Fibonacciego (2)

- ▶ Kopiec Fibonacciego ma formę lasu złożonego z kopców.
- ▶ Kształt kopków jest mniej rygorystyczny.
- ▶ Większa elastyczność pozwala na zostawienie niektórych czynności na później (tzw. lazy approach).
- ▶ W pewnym momencie potrzeba jednak narzucić większy porządek (kosztem dodatkowego czasu operacji).



## Kopiec Fibonacciego (3)

- ▶ Stopień (liczba dzieci) każdego węzła wynosi co najwyżej  $\log n$ .
- ▶ Dla węzła o stopniu  $k$  rozmiar podrzewa zakończonego w nim wynosi co najmniej  $F_{k+2}$  ( $k + 2$ -ta liczba Fibonacciego).
- ▶ Dla nie-korzenia  $x$  możemy odciąć tylko jednego syna. Gdy odcinamy kolejnego,  $x$  samo jest odcinane i staje się osobnym drzewem.
- ▶ Zwiększa to liczbę drzew, ale drzewa można łączyć podczas innej operacji.



## Kopiec Fibonacciego (4)

- ▶ Do analizy złożoności kopca Fibonacciego stosowana jest metoda potencjału.
- ▶ Węzeł  $x$  jest znaczony (marked), jeśli co najmniej 1 jego syn został odcięty od czasu gdy  $x$  został czyimś synem.
  - ▶ Korzenie nigdy nie są znaczone.
- ▶ Potencjał kopca Fibonacciego to liczba kopców plus dwukrotność liczby znaczonych węzłów:

$$\Phi = t + 2m \quad (2)$$

- ▶ Czas zamortyzowany to czas rzeczywisty plus różnica potencjałów razy pewna dobrana wartość  $C$ .



# Kopiec Fibonacciego (5)

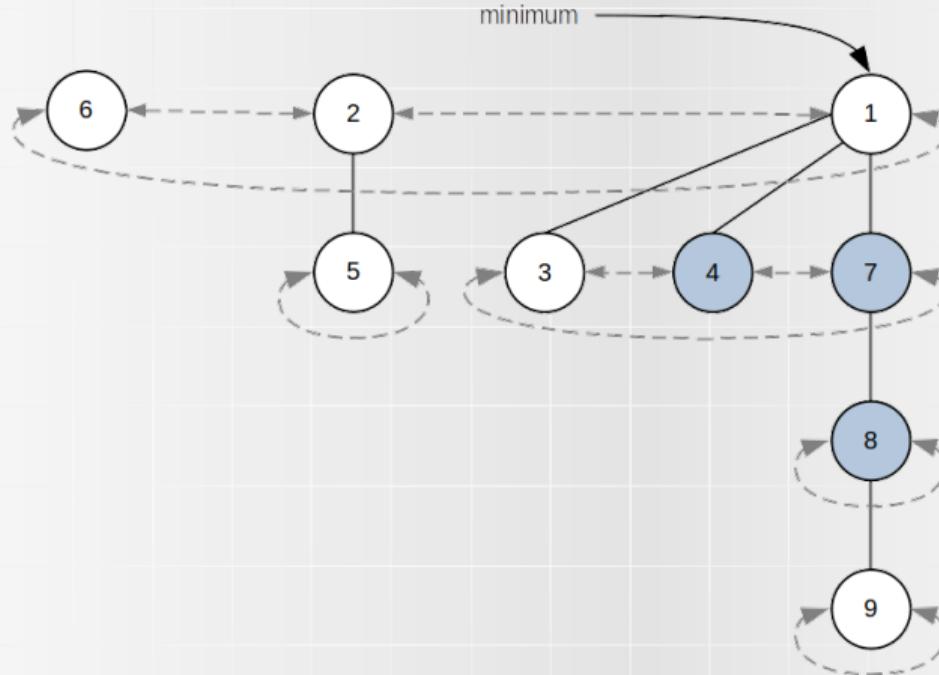
Dodatkowe założenia dotyczące realizacji struktury:

- ▶ Korzenie (las) połączone są cykliczną listą dwukierunkową.
- ▶ Rodzeństwo również połączone jest taką listą.
- ▶ Węzeł przechowuje stopień (liczbę dzieci) i to czy jest znaczony.
- ▶ Przechowujemy wskaźnik na najmniejszy korzeń.



# Kopiec Fibonacciego (6)

Przykład kopca Fibonacciego:





# Kopiec Fibonacciego (7)

Implementacja operacji:

- ▶ `find-min()` – zwracamy pamiętany wskaźnik na minimum.
  - ▶ Nie zmienia potencjału, czas zamortyzowany  $O(1)$ .
- ▶ `merge()` – operacja, którą można zdefiniować dla wielu ADT/struktur danych
  - ▶ Łączy dwie struktury (zwykle tego samego typu) w jedną.
  - ▶ Połączenie dwóch kopków Fibonacciego sprowadza się do połączenia dwóch list cyklicznych z korzeniami i wybrania nowego minimum z dwóch wartości.
  - ▶ Nie zmienia potencjału, czas zamortyzowany  $O(1)$ .



## Kopiec Fibonacciego (8)

- ▶ `insert()` – dodaje nowy 1-elementowy kopiec (sam korzeń), konieczność wyboru nowego elementu minimalnego z dwóch możliwych.
- ▶ Alternatywnie: tworzymy nowe drzewo i wykonujemy `merge()` z oryginalnym drzewem.
- ▶ Czas rzeczywisty  $O(1)$ .
- ▶ Potencjał zwiększa się o 1 (1 nowe drzewo bez węzłów znaczonych).
- ▶ Czas zamortyzowany  $O(1)$ .



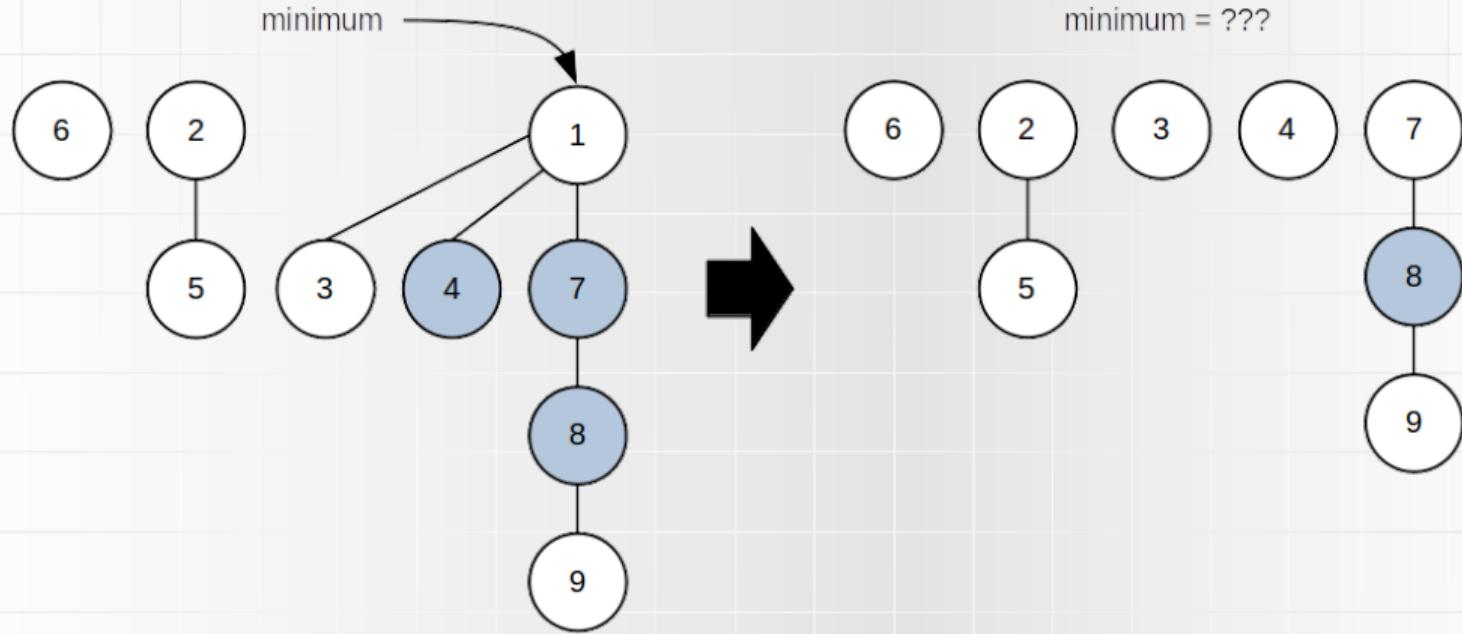
# Kopiec Fibonacciego (9)

extract-min() składa się z kilku faz:

- ▶ Faza pierwsza:
  - ▶ Usuwamy minimalny węzeł (do którego mamy wskaźnik).
  - ▶ Dzieci usuniętego węzła (który był korzeniem) stają się osobnymi drzewami.
  - ▶ Jeśli dzieci było  $k$ , to potencjał wzrasta o  $k - 1$  (1 korzeń znika,  $k$  się pojawia).
  - ▶ Czas to  $O(k) \in O(\log n)$ .



# Kopiec Fibonacciego (10)





# Kopiec Fibonacciego (11)

## ► Faza druga:

- Redukcja liczby korzeni – korzenie o tym samym stopniu są łączone (to nie jest operacja merge!).
- Jedno z łączonych drzew staje się synem drugiego (zgodnie z własnością kopca).
- Powtarzane dopóki pozostałe korzenie mają różne stopnie (będzie ich więc  $O(\log n)$ ).
- Przechowujemy tablicę rozmiaru  $O(\log n)$  przechowującą w indeksie  $i$  wskaźnik do korzenia o stopniu  $i$ , co pozwala na szybkie lokalizowanie i łączenie kopków.



## Kopiec Fibonacciego (12)

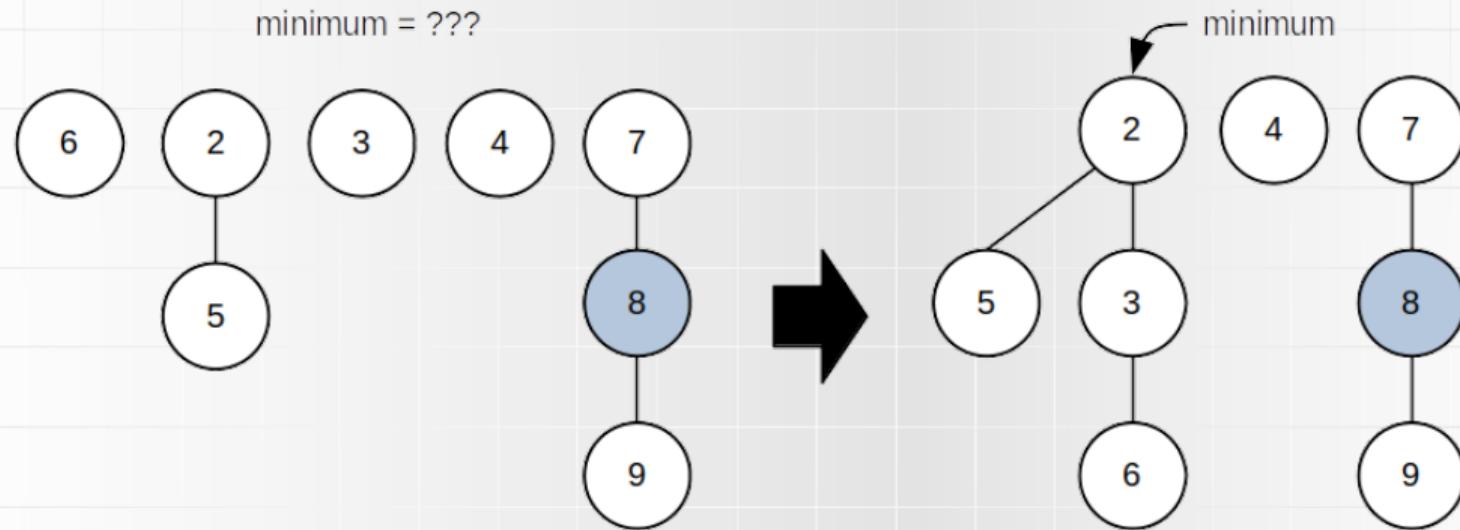
- ▶ Czas rzeczywisty:  $O(\log n + m)$ , gdzie  $m$  to liczba kopców (korzeni) na początku fazy.
- ▶ Zmiana potencjału:  $O(\log n) - m$ .
- ▶ Czas zamortyzowany:  $O(\log n + m) + C(O(\log n) - m)$ .
  - ▶ Przy odpowiednio dużym  $C$  wynik upraszcza się do  $O(\log n)$ .
- ▶ Faza trzecia – sprawdzamy wynikłe  $O(\log n)$  korzeni, by znaleźć nowe minimum.
  - ▶ Czas  $O(\log n)$ , brak zmian potencjału, więc koszt zamortyzowany również  $O(\log n)$ .

Cała operacja extract-min ma więc zamortyzowany koszt  $O(\log n)$ .



# Kopiec Fibonacciego (13)

Połączenie drzew (6) i (3), a następnie drzew (3, 6) i (2, 5).





# Kopiec Fibonacciego (13)

Operacja decrease-key():

- ▶ Znajdujemy węzeł (z odpowiednim słownikiem trwa to  $O(\log n)$ ).
- ▶ Zmniejszamy klucz węzła.
- ▶ Jeśli złamana jest zasada kopca, to odcinamy węzeł od rodzica i oznaczamy rodzica (chyba, że jest korzeniem).
- ▶ Jeśli rodzic był już oznaczony, to też go obcinamy i oznaczamy jego rodzica.
- ▶ Gdy proces się skończy (dotarliśmy do nieoznaczonego węzła), to określamy nowe minimum z dwóch wartości (zmniejszony klucz i nowe minimum).



## Kopiec Fibonacciego (13)

- ▶ Założmy, że proces utworzył  $k$  nowych drzew (korzeni).
- ▶ Wśród nich co najmniej  $k - 1$  było oznaczonych (odcięty jako pierwszy w procesie mógł nie być).
- ▶ Ponieważ stają się korzeniami, przestają być oznaczone.
- ▶ Jeden węzeł mógł zostać oznaczony.
- ▶ Dochodzi  $k$  drzew oraz  $-k + 2$  oznaczonych węzłów.
- ▶ Potencjał zmienia się więc o  $k + 2(-k + 2) = -k + 4$ .



## Kopiec Fibonacciego (14)

- ▶ Czas rzeczywisty:  $O(k)$ .
- ▶ Czas zamortyzowany:  $O(k) + C(-k + 4)$ .
  - ▶ Dla odpowiedniego  $C$  otrzymujemy  $O(1)$ .

Operacja `delete()`:

- ▶ Za pomocą `decrease-key()` ustawiamy wartość usuwanego węzła na  $-\infty$ .
- ▶ Za pomocą `extract-min()` usuwamy węzeł, który teraz stał się minimum.
- ▶ Czas  $O(1) + O(\log n) \in O(\log n)$ .



## Kopiec Fibonacciego (15)

- ▶ Kopiec Fibonacciego poprawia teoretyczną złożoność niektórych algorytmów grafowych np. algorytmu Dijkstry i algorytmu Prima.
- ▶ Dobra teoretyczna złożoność okupiona jest jednak skomplikowaną implementacją i mniejszą skutecznością praktyczną (zwłaszcza w niektórych sytuacjach).
  - ▶ Niektóre pojedyncze operacje na kopcu wykonują się długo, zaś wyższa stała zmniejsza korzyść względem  $O(\log n)$ .
- ▶ Wskazówki praktyczne:
  - ▶ Gdy nie potrzeba operacji `decrease-key()`: stosujemy kopce binarne.
  - ▶ Gdy potrzeba `decrease-key()`: stosujemy kopce parujące (pairing heaps).
    - ▶ Asymptotycznie gorsze od kopca Fibonacciego, ale bardzo dobre w praktyce i prostsze w implementacji.



# Kopiec Fibonacciego (16)

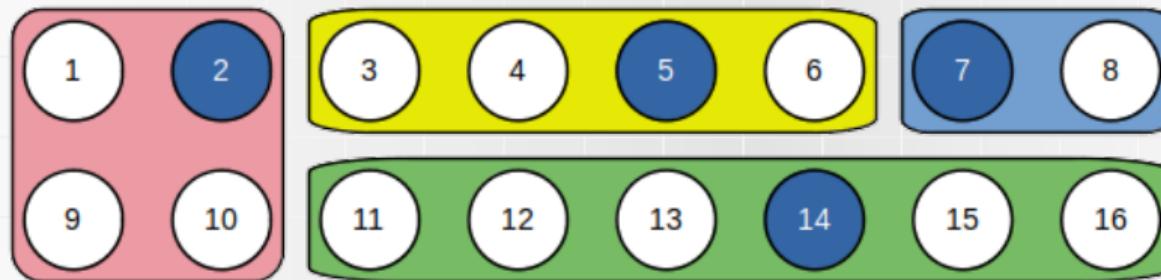
Podsumowanie złożoności niektórych odmian kopków:

Kopiec	insert()	extract-min	find-min()	decrease-key()	merge()
Binarny	$O(\log n)$	$\Theta(\log n)$	$\Theta(1)$	$O(\log n)$	$O(n)$
Dwumianowy	$\Theta(1)^a$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(\log n)$	$O(\log n)$
Parujący	$\Theta(1)$	$O(\log n)^a$	$\Theta(1)$	$o(\log n)^a$	$\Theta(1)$
Fibonacciego	$\Theta(1)$	$O(\log n)^a$	$\Theta(1)$	$\Theta(1)^a$	$\Theta(1)$

<sup>a</sup> – czas zamortyzowany

# Struktura zbiorów rozłącznych (1)

- ▶ Disjoint-set data structure – przechowuje elementy pogrupowane w zbiory tak, że każdy element należy do dokładnie jednego zbioru.
  - ▶ Z matematycznego punktu widzenia przechowuje pewne rozbicie zbioru.
- ▶ Każdy zbiór ma reprezentanta.
- ▶ Stosowana między innymi w algorytmie Kruskala.





## Struktura zbiorów rozłącznych (2)

- ▶ Operacje:
  - ▶ `insert()` – dodanie nowego elementu.
    - ▶ Powstaje nowy zbiór 1-elementowy.
  - ▶ `union()` – łączenie dwóch zbiorów w jeden.
  - ▶ `find()` – znalezienie reprezentanta zbioru.
    - ▶ Jeśli dwa elementy mają tego samego reprezentanta, to należą do tego samego zbioru.



# Struktura zbiorów rozłącznych (3)

Implementacja naiwna z użyciem samych list:

- ▶  $\text{insert}(x)$  – czas  $O(1)$ .
- ▶  $\text{union}(x,y)$  –  $O(\min\{|X|, |Y|\}) \in O(n)$ .
- ▶  $\text{find}(x)$  – czas  $O(|X|) \in O(n)$ .
- ▶  $X$  i  $Y$  to zbiory do których należą odpowiednio  $x$  i  $y$ .



## Struktura zbiorów rozłącznych (4)

W praktyce używamy implementacji lasu zbiorów rozłącznych (disjoint-set forest):

- ▶ Każdy zbiór reprezentowany jest przez osobne drzewo.
- ▶ Każdy węzeł pamięta rodzica (plus stopień węzła/rozmiar poddrzewa).
- ▶ Korzeń drzewa jest reprezentantem zbioru.
- ▶ Niektóre operacje mogą być długie, ale przekształcają strukturę na korzyść przyszłych operacji, dbając by drzewa miały małą wysokość.
- ▶ Czas zamortyzowany jest bardzo dobry.
- ▶ Implementacja jest zarówno szybka praktycznie jak i optymalna asymptotycznie (gorsza najwyżej o stałą od najlepszej możliwej).



# Struktura zbiorów rozłącznych (5)

Implementacja  $\text{insert}(x)$ :

- ▶ Stworzenie elementu  $x$ .
- ▶  $x.parent \leftarrow x$ .
- ▶  $x.rank \leftarrow 0$  (lub  $x.size \leftarrow 1$ ).
- ▶ Dodanie  $x$  do listy drzew (zbiorów).
- ▶ Czas  $O(1)$ .



# Struktura zbiorów rozłącznych (6)

Implementacja `find(x)`:

- ▶ W teorii wystarczy jedynie podążać (iteracyjnie lub rekurencyjnie) za rodzicem, do korzenia i go zwrócić.
- ▶ Drzewa mogą jednak zrobić się duże, więc `find()` wykorzystuje tą okazję by zmniejszyć ich wysokość.
- ▶ Efekt uzyskuje się przez zmianę wskaźników, by zmniejszyć czas dotarcia do korzenia w kolejnych wywołaniach `find()` dla tego samego zbioru.



## Struktura zbiorów rozłącznych (7)

Algorytm kompresji ścieżek (path compression):

- ▶ Dla każdego węzła na drodze od  $x$  do korzenia root wykonujemy:
  - ▶  $x.parent \leftarrow root$
- ▶ Przypisanie wymaga znajomości korzenia, więc trzeba wykonać 2 przejścia przez ścieżkę.
  - ▶ Podejście rekurencyjne – wymaga dodatkowej pamięci (zapis ścieżki na stosie).
  - ▶ Podejście iteracyjne – dwie pętle, pierwsza znajduje korzeń, druga przechodzi ponownie ścieżkę i zapisuje go. Wymaga stałej pamięci.



## Struktura zbiorów rozłącznych (8)

Algorytm dzielenia ścieżek (path splitting):

- ▶ Dla każdego węzła na drodze od  $x$  do korzenia wykonujemy:
  - ▶  $(x, x.parent) \leftarrow (x.parent, x.parent.parent)$
- ▶ Węzeł zamiast ojca wskazywać będzie od teraz na dziadka – za każdym wykonaniem  $\text{find}()$  będą wskazywać bliżej korzenia.
- ▶ Pesymistycznie tak samo jak dla kompresji ścieżek, ale lepszy w praktyce.

Algorytm połowicznej ścieżki (path halving):

- ▶ Identycznie, ale zmiana jest co drugi węzeł:
  - ▶  $x.parent \leftarrow x.parent.parent$
  - ▶  $x \leftarrow x.parent$



## Struktura zbiorów rozłącznych (9)

Implementacja  $\text{union}(x, y)$ :

- ▶ Używamy  $\text{find}(x)$  i  $\text{find}(y)$  by znaleźć reprezentantów (nazwijmy ich odpowiednio  $r_x$  i  $r_y$ ).
- ▶ Jeśli  $r_x = r_y$  to  $x$  i  $y$  są w tym samym zbiorze i algorytm się kończy.
- ▶ Jeśli  $r_x \neq r_y$ , to łączymy zbiory, czyniąc jeden korzeń synem drugiego.
- ▶ Wybór bardzo wpływa na wysokość przyszłych drzew! Zapobiegamy zbyt wysokim drzewom za pomocą dodatkowych czynności.
- ▶ Konieczność przechowywania w węzłach rozmiaru drzewa (dodatkowe  $O(\log n)$  bitów) lub rangi korzenia (dodatkowe  $O(\log \log n)$  bitów).



# Struktura zbiorów rozłącznych (10)

Łączenie przez rozmiar (union by size):

- ▶ Korzeniem złączonych drzew staje się korzeń drzewa o większym rozmiarze.
- ▶ Jeśli rozmiary są identyczne, wybór jest dowolny.
- ▶ Aktualizujemy rozmiar nowego korzenia.



# Struktura zbiorów rozłącznych (11)

Łączenie przez rangę (union by rank):

- ▶ Ranga jest górnym ograniczeniem na wysokość drzewa.
- ▶ Lepszy wskaźnik niż wysokość, bo nie zmienia się w czasie `find()`.
- ▶ Jeśli łączone drzewa mają różne rangi, to ten z większą staje się rodzicem i rangi się nie zmieniają.
- ▶ Jeśli rangi są takie same, to dowolne drzewo może stać się rodzicem, ale jego rangę trzeba zwiększyć o 1.



## Struktura zbiorów rozłącznych (12)

- ▶ `find()` i `union()` mają zamortyzowaną złożoność  $O(\alpha(n))$ , gdzie  $\alpha$  jest odwrotnością bardzo szybko rosnącej funkcji Ackermanna.
- ▶ Ponieważ  $O(1) \in O(\alpha(n))$ , to dowolny ciąg  $k$  operacji na lesie zbiorów rozłącznych działa w czasie  $O(k\alpha(n))$ .
- ▶ „Fizycznie” wydaje się niemożliwe uzyskanie  $\alpha(n) > 4$ , w praktyce można więc przyjąć, że  $O(\alpha(n)) \in O(1)$ .
- ▶ Wszystkie operacje działają więc zasadniczo w czasie  $O(1)$ , ale jest to czas zamortyzowany – niektóre operacje w ciągu mogą zająć długie czas.



Wrocław  
University  
of Science  
and Technology

# Struktury danych

Wykład 7

## Słowniki, binarne drzewa poszukiwań

dr inż. Jarosław Rudy





# Słowniki (1)

- ▶ Słownik (mapa, tablica asocjacyjna) jest ADT przechowującym elementy w postaci pary klucz-wartość.
- ▶ Klucze mogą być dowolnego typu (najczęściej są to liczby lub łańcuchy tekstowe).
- ▶ Słownik mapuje zbiór kluczy na zbiór wartości (jak funkcja matematyczna).
  - ▶ Część kluczy może nie mieć wartości (funkcja częściowa).
- ▶ Generalnie każdy klucz ma co najwyżej jedną wartość.
  - ▶ Multimapa – uogólnienie słownika, w którym z jednym kluczem może być związane kilka wartości (np. kolekcja).



## Słowniki (2)

Typowe operacje na słowniku:

- ▶  $\text{insert}(k, v)$  – dodanie do słownika elementu o kluczu  $k$  i wartości  $v$ .
  - ▶ Jeśli element dla klucza  $k$  już istnieje, to zostanie nadpisany przez  $v$ .
- ▶  $\text{find}(k)$ ,  $\text{lookup}(k)$  – zwraca  $S[k]$ , czyli element mapowany przez klucz  $k$  w słowniku  $S$ .
  - ▶ Jeśli taki element nie istnieje, to najczęściej zwracana jest specjalna wartość lub podnoszony jest wyjątek.
- ▶  $\text{delete}(k)$ ,  $\text{remove}(k)$  – usunięcie elementu o kluczu  $k$ , usuwa mapowanie klucza.



## Słowniki (3)

- ▶ `exists(k)` – zwraca czy klucz  $k$  ma przypisaną wartość w słowniku.
- ▶ `size()` – zwraca rozmiar słownika.
- ▶ `empty()` – zwraca czy słownik ma jakikolwiek przypisany klucz.
- ▶ `keys()` – zwraca (enumeruje) listę przypisanych kluczy słownika lub iterator na taką listę.
- ▶ `values()` – zwraca listę wartości słownika lub iterator na taką listę.



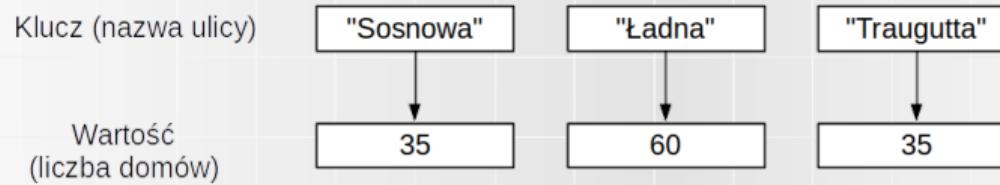
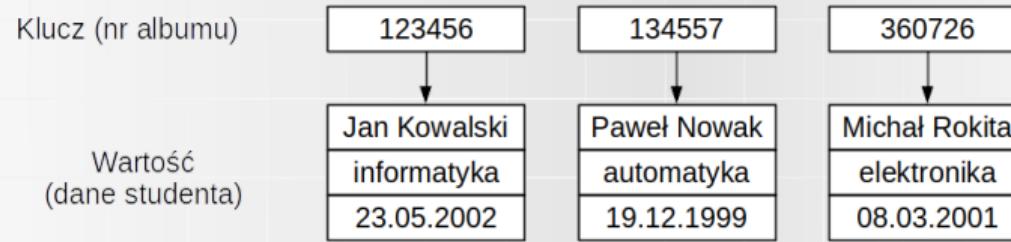
## Słowniki (4)

- ▶ Podstawowa definicja słownika nie wymaga, by na kluczach definiować porządek (relacje  $<$ ,  $\leqslant$  itd.), ani by przechowywać je w jakimś porządku.
- ▶ Niektóre z implementacji słowników będą jednak tego wymagały!
- ▶ Można jednak zdefiniować uporządkowany słownik, w którym elementy zachowują porządek przy listowaniu. Istnieją 2 powszechnie definicje:
  - ▶ Porządek określony przez sortowanie kluczy (niezależny od wstawiania).
  - ▶ Porządek określony przez kolejność wstawiania (niezależny od klucza).



# Słowniki (5)

Przykłady mapowania klucz-wartość w słownikach.





## Słowniki (6)

- ▶ Słownik zakłada szybkie wyszukiwanie elementu po zadanym kluczu.
- ▶ Dla implementacji z użyciem np. listy, średni i pesymistyczny czas wyszukiwania wyniesie  $O(n)$ .
- ▶ Listy z przeskokiem lub samoorganizujące się ulepszają przypadek średni do  $O(\log n)$ , ale pesymistycznie wciąż jest  $O(n)$ .
- ▶ Szybsze wyszukiwanie można uzyskać implementując słownik jako:
  - ▶ Binarne drzewo poszukiwań.
  - ▶ Tablicę mieszającą.



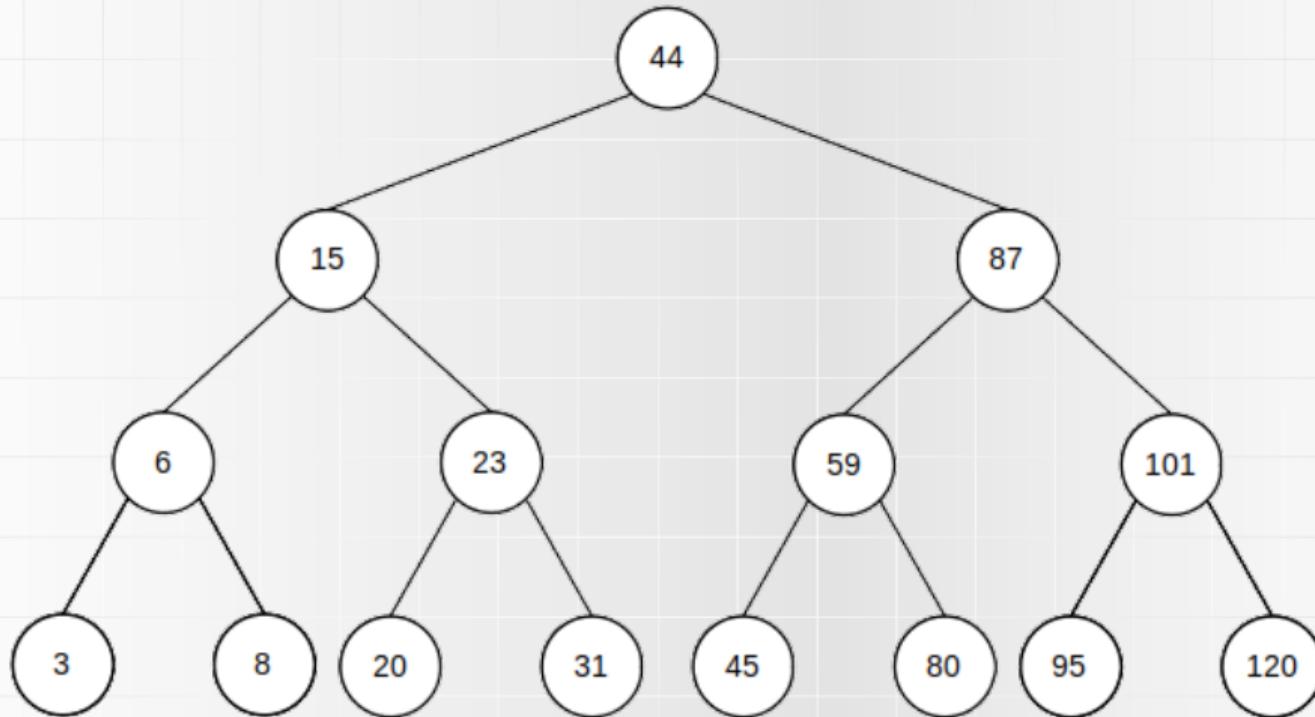
# Binarne drzewo poszukiwań (1)

Drzewo binarne (binary search tree, BST):

- ▶ Drzewo binarne w którym klucz węzła jest większy niż klucz lewego syna i mniejszy niż klucz prawego syna.
- ▶ Wymaga zdefiniowania porządku dla każdej pary kluczy.
- ▶ Wypisując BST metodą in-order otrzymamy elementy posortowane wg klucza (słownik uporządkowany).



## Binarne drzewo poszukiwań (2)





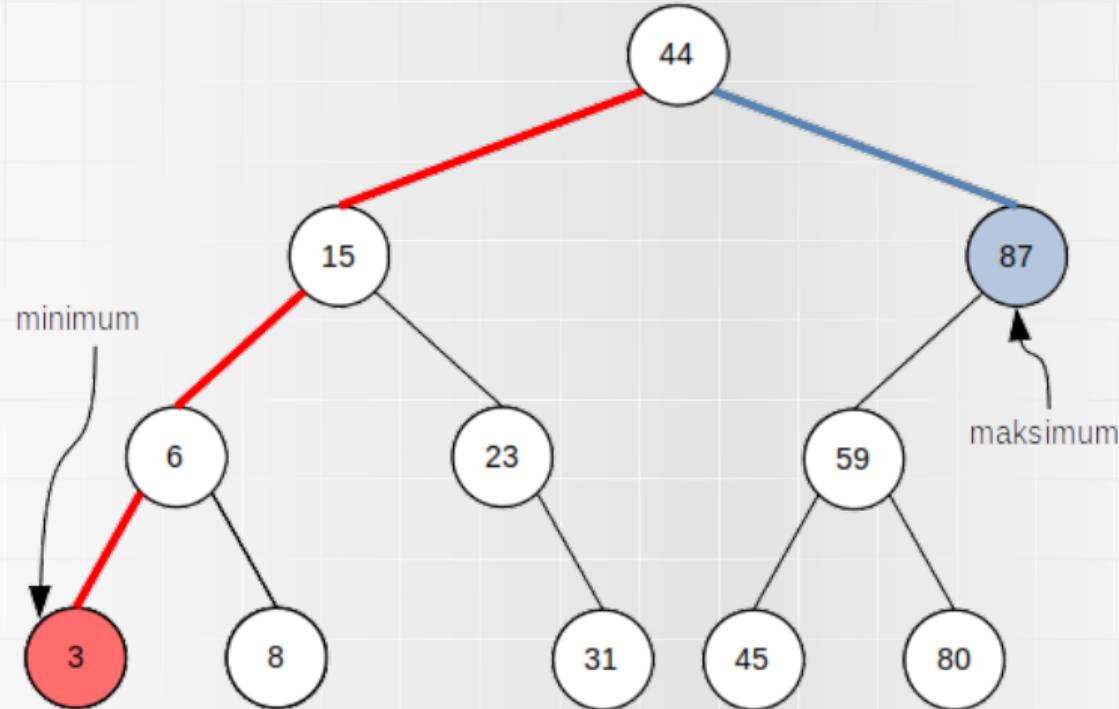
## Binarne drzewo poszukiwań (3)

Operacje pomocnicze:

- ▶ `minimum()` – znajdowanie elementu minimalnego w drzewie:
  - ▶ Podążamy lewym podrzewem dopóki istnieje.
  - ▶ Jeśli nie ma lewego syna, aktualny węzeł jest szukanym minimum.
- ▶ `maximum()` – analogicznie (podążamy prawym poddrzewem do końca).



## Binarne drzewo poszukiwań (4)



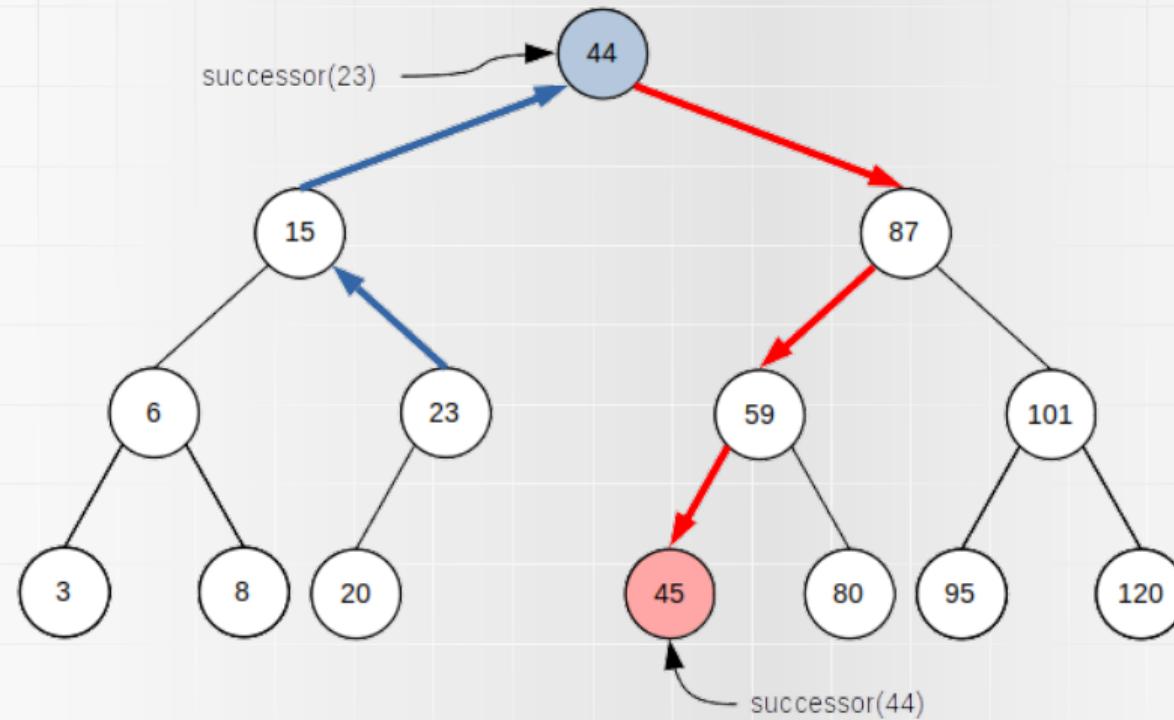


## Binarne drzewo poszukiwań (5)

- ▶  $\text{successor}(w)$  – znajdywanie następnika węzła  $w$  tj. węzła następnego w kolejności posortowanej (mającego najmniejszy klucz większy od  $w.\text{key}$ ):
  - ▶ Jeśli istnieje  $w.\text{right}$ , to  $\text{successor}(w) \leftarrow \text{minimum}(w.\text{right})$ .
  - ▶ Jeśli  $w.\text{right}$  nie istnieje, to następnikiem  $w$  jest pierwszy z przodków  $w$ , dla którego  $w$  leży w lewym poddrzewie.
    - ▶ Innymi słowy, zaczynając od  $v \leftarrow w$  przechodzimy cyklicznie do rodzica ( $v \leftarrow v.\text{parent}$ ), do czasu aż  $v.\text{parent.left} = v$ . Wtedy  $v.\text{parent}$  jest szukanym następnikiem.
- ▶  $\text{predecessor}(w)$  – znajdywanie poprzednika węzła  $w$  tj. węzła poprzedniego w kolejności posortowanej (mającego największy klucz mniejszy od  $w.\text{key}$ ):
  - ▶ Analogicznie do następnika.



# Binarne drzewo poszukiwań (6)





## Binarne drzewo poszukiwań (7)

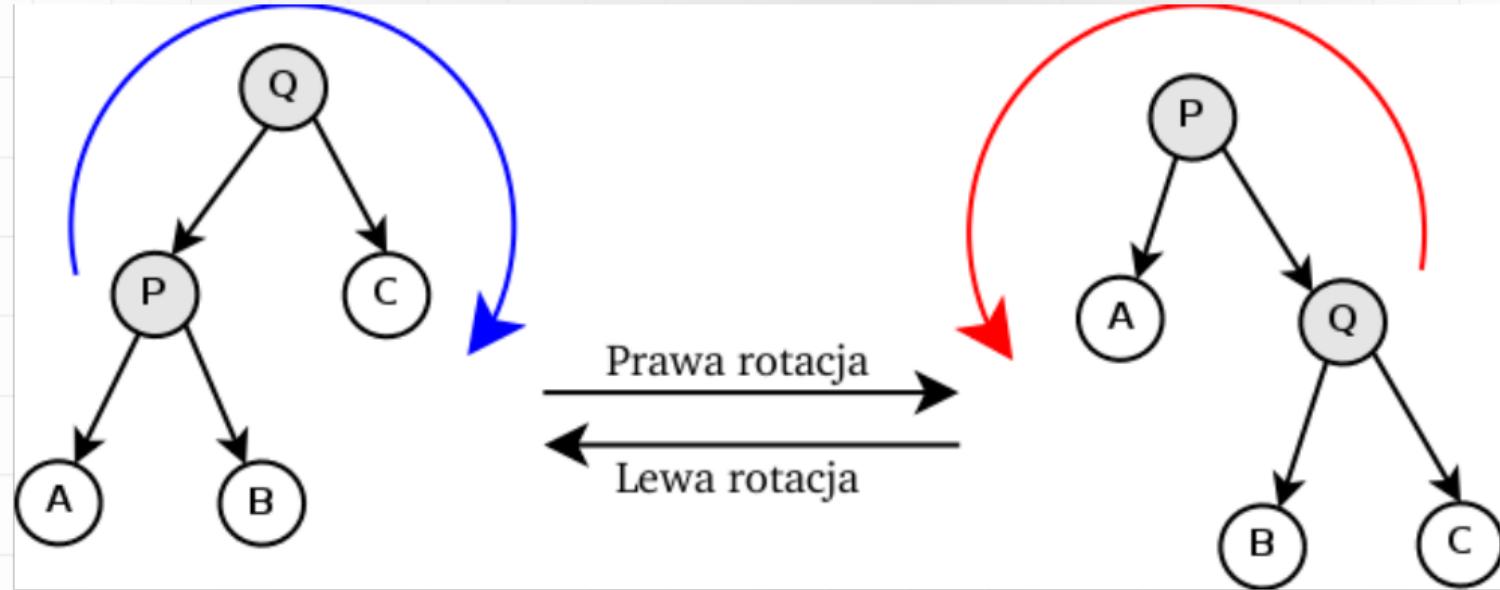
- ▶ Rotację definiujemy względem korzenia  $w$  pewnego poddrzewa (lub względem krawędzi łączącej ten korzeń go z jego synem).
- ▶ Rozróżniamy dwie podstawowe rotacje:
  - ▶  $\text{rotateLeft}(w)$ 
    - ▶  $\text{temp} \leftarrow w.\text{right}.\text{left}$
    - ▶  $w.\text{right}.\text{left} \leftarrow w$
    - ▶  $w.\text{right} \leftarrow \text{temp}$



## Binarne drzewo poszukiwań (8)

- ▶  $\text{rotateRight}(w)$ 
  - ▶  $\text{temp} \leftarrow w.\text{left}.\text{right}$
  - ▶  $w.\text{left}.\text{right} \leftarrow w$
  - ▶  $w.\text{left} \leftarrow \text{temp}$
- ▶ Rotacja zmniejsza głębokość jednego wierzchołka/poddrzewa kosztem zwiększenia głębokości innego wierzchołka/poddrzewa.
- ▶ Lewa rotacja jest operacją odwrotną do prawej rotacji i vice versa.
- ▶ Rotacje zachowują porządek kluczy (przegląd in-order).

# Binarne drzewo poszukiwań (9)



W obu przypadkach porządek kluczów to  $A < P < B < Q < C$ .



# Binarne drzewo poszukiwań (10)

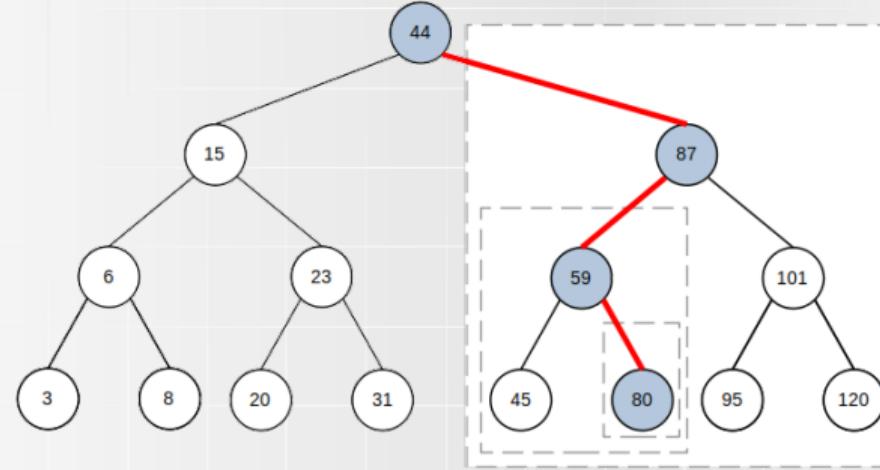
Operacja  $\text{find}(k)$ :

- ▶  $w \leftarrow \text{root}$
- ▶ Dopóki  $w$  istnieje (np. nie jest null):
  - ▶ Jeśli  $k = \text{key}(w)$ , zwracamy  $w$  (znaleziono klucz).
  - ▶ Jeśli  $k < \text{key}(w)$ , to  $w \leftarrow \text{left}(w)$  (przechodzimy do lewego poddrzewa).
  - ▶ Jeśli  $k > \text{key}(w)$ , to  $w \leftarrow \text{right}(w)$  (przechodzimy do prawego poddrzewa).
- ▶ Klucza nie ma w drzewie, zwróć odpowiednią wartość (null, wyjątek itp.).

Możliwa jest również wersja iteracyjna.

# Binarne drzewo poszukiwań (11)

Przykład szukania w BST wartości o kluczu równym 80.



Jeśli lewe i prawe podrzewa zawsze mają podobny rozmiar, to z każdą decyzją odrzucamy ok. połowę pozostałych węzłów do sprawdzenia (zasada algorytmu przeszukiwania binarnego).



# Binarne drzewo poszukiwań (12)

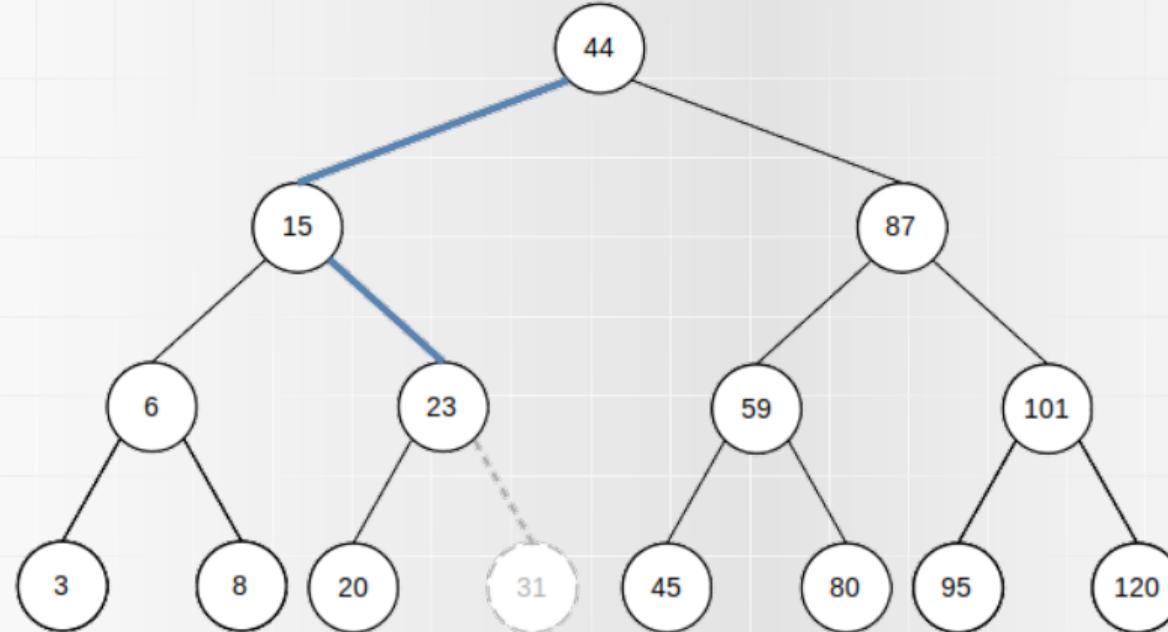
Operacja  $\text{insert}(k, v)$ :

- ▶ Tworzymy nowy element  $e = (k, v)$ .
- ▶ Jeśli drzewo jest puste, to dodajemy  $e$  jako korzeń.
- ▶ W przeciwnym razie, znajdujemy miejsce do wstawienia węzła.
  - ▶ Odbiera się to prawie identycznie jak w operacji  $\text{find}(k)$ .
  - ▶ Ponieważ  $k$  nie ma w drzewie, to natrafimy w końcu na null (brak poddrzewa w którym powinno być  $k$ ).
  - ▶ Wpisujemy  $e$  w miejsce nulla (tj. jako odpowiedni syn liścia).



# Binarne drzewo poszukiwań (13)

Przykład dodawania do drzewa węzła o kluczu 31.





## Binarne drzewo poszukiwań (14)

Operacja  $\text{delete}(k)$ :

- ▶ Znajdujemy węzeł  $w$  do usunięcia poprzez  $\text{find}(k)$ .
- ▶ Usuwamy znaleziony węzeł  $w$ .
- ▶ Konieczność reorganizacji drzewa, możliwe 3 przypadki:
  - ▶ Jeśli  $w$  nie miał synów (był liściem), to nie robimy nic.
  - ▶ Jeśli  $w$  miał jednego syna, to syn zastępuje  $w$ .
  - ▶ Jeśli  $w$  miał dwóch synów, to wstawiamy w miejsce  $w$  jego następnika.



## Binarne drzewo poszukiwań (15)

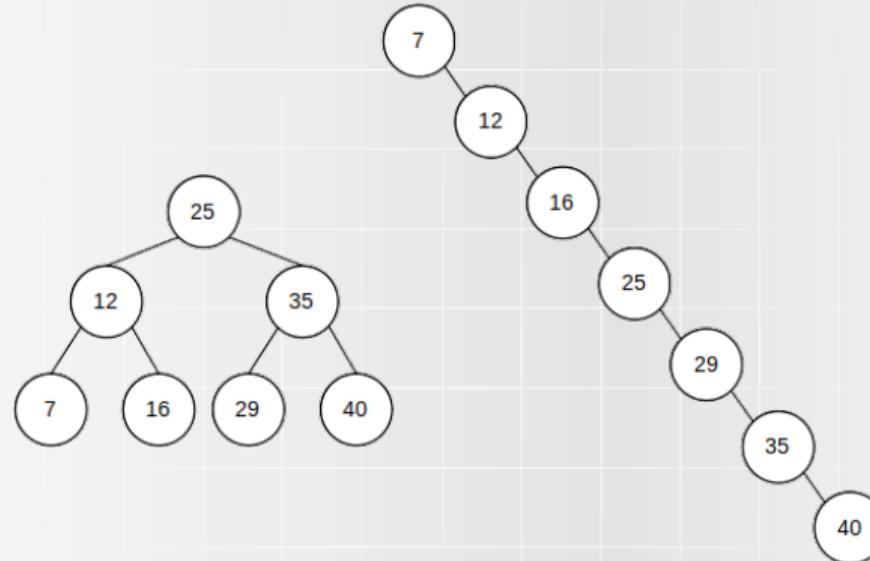
Złożoność operacji, zakładając drzewo o wysokości  $h$  z  $n$  elementami:

- ▶  $\text{insert}(k, v)$ ,  $\text{find}(k)$ ,  $\text{exists}(k)$  i  $\text{delete}(k)$ :
  - ▶ średnio  $O(\log n)$ ,
  - ▶ pesymistycznie  $O(h)$ .
- ▶  $\text{keys}()$  i  $\text{values}()$ :
  - ▶ średnio i pesymistycznie  $\Theta(n) \in O(n)$ .
- ▶  $\text{size}()$  i  $\text{empty}()$ :
  - ▶ średnio i pesymistycznie  $O(1)$ .



# Binarne drzewo poszukiwań (16)

Niekorzystne wstawianie zwiększa rozmiar drzewa. W najgorszym przypadku mamy  $h = n$  (drzewo binarne degraduje się do listy).





## Binarne drzewo poszukiwań (17)

- ▶ Możemy rozważać drzewo zrównoważone. 2 definicje:
  - ▶ Równoważenie wysokością: dla każdego węzła wysokość obu jego poddrzew różni się co najwyżej o stałą:
- ▶ Doskonale zrównoważone, jeśli wysokość różni się najwyżej o 1.
- ▶ Równoważenie wagą: dla każdego węzła jego waga różni się najwyżej o stały czynnik od wagi synów.
- ▶ Waga węzła to rozmiar jego poddrzewa plus 1.



# Binarne drzewo poszukiwań (18)

Niektóre sposoby równoważenia drzew:

- ▶ Algorytm równoważenia DSW.
- ▶ Drzewa samorównoważące się np.:
  - ▶ Drzewa AVL.
  - ▶ Drzew czerwono-czarne.
  - ▶ B-drzewa.



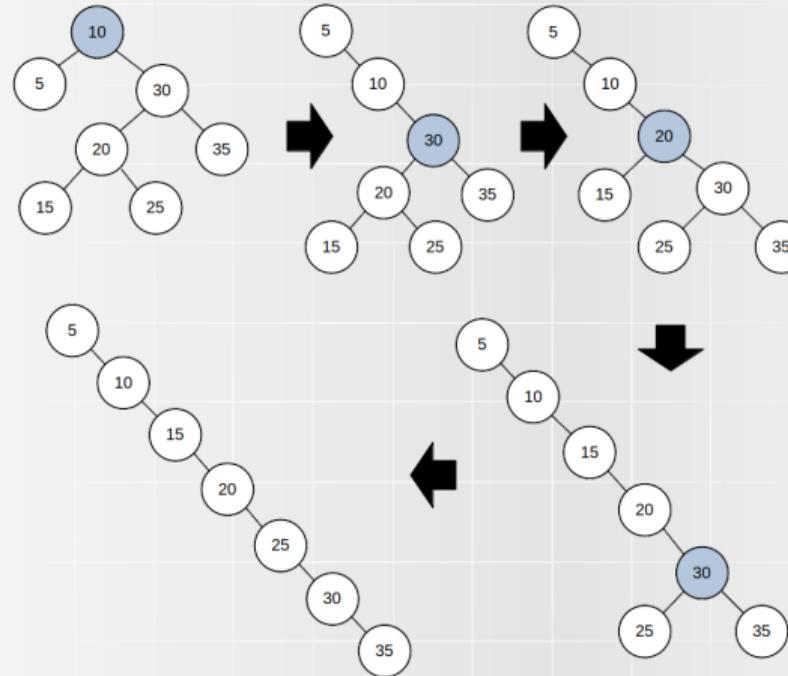
# Algorytm DSW (1)

- ▶ Algorytm równoważy zadane drzewo (gwarantując, że  $h \in O(\log n)$ ) w dwóch fazach z użyciem rotacji.
- ▶ W fazie pierwszej wykorzystywane są wielokrotne prawe rotacje, wykonywne od korzenia.
  - ▶ W wyniku drzewo zostaje zamienione w listę (tzw. kręgosłup, vine).
- ▶ W fazie drugiej iteracyjnie stosujemy lewe iteracje na co drugim węźle wzdłuż prawej gałęzi drzева.
- ▶ Zabieg ten stosowany jest wielokrotnie, za każdym razem zmniejszając wysokość drzева o połowę.

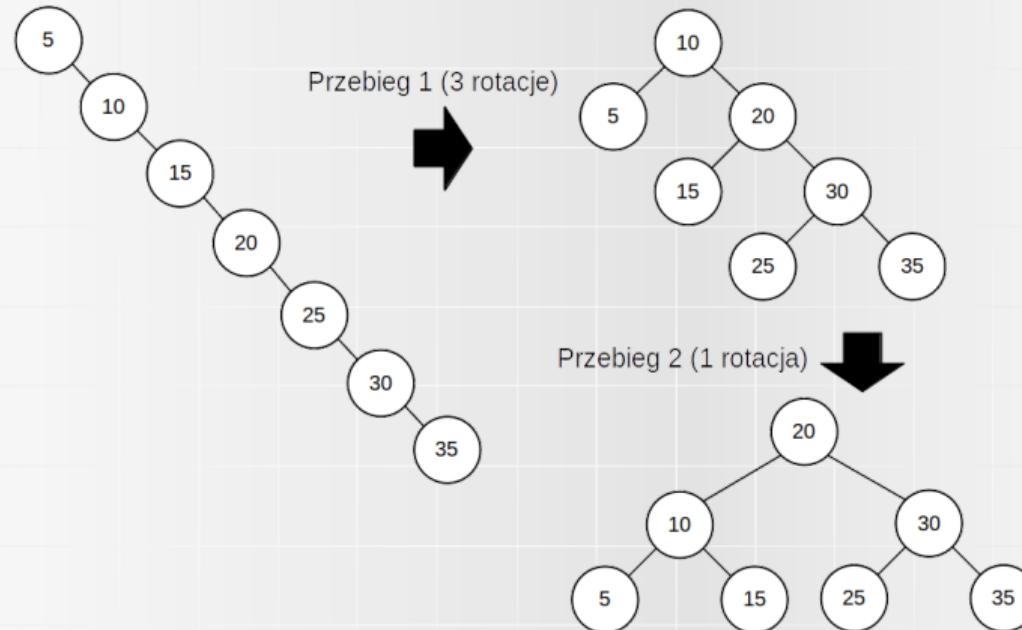


# DSW (2)

## Przykład fazy pierwszej



## Przykład fazy drugiej





# Algorytm DSW (4)

- ▶ Obie fazy działają w czasie  $O(n)$ .
- ▶ Niskie zapotrzebowanie na pamięć ( $O(1)$ ).
- ▶ Uzyskujemy drzewo doskonale zrównoważone.
- ▶ Brak potrzeby sortowania lub dalszej dekompozycji drzewa.
- ▶ Konieczność ręcznego stosowania.



# Drzewa AVL (1)

- ▶ Samorównoważące się BST.
- ▶ Każdy wierzchołek przechowuje współczynnik zrównoważenia (różnica wysokości jego lewego i prawego poddrzewa, 2 bity).
  - ▶ Wartości  $-1$ ,  $0$  oraz  $1$  są w porządku (ale zmiana może wymagać propagacji tej informacji).
  - ▶ Wartości  $-2$  oraz  $2$  wymagają naprawy poziomu wyważenia węzłów.
- ▶ Wyszukiwanie odbywa się identycznie jak dla normalnego BST, ale dzięki wyważeniu, gwarantowany jest czas  $O(\log n)$ .



## Drzewa AVL (2)

- ▶ Operacja wstawiania – początek działa jak dla zwykłego BST, po czym należy przeprowadzić proces aktualizacji wyważeń od węzła wzwyż (maksymalnie do korzenia), przy czym:
  - ▶ Wyważenie aktualizowane na 0 kończy proces bez konieczności dalszych zmian.
  - ▶ Wyważenie aktualizowane na  $-2$  lub  $2$  oznacza, że drzewo straciło właściwość AVL. Naprawa przebiega z użyciem 1–2 rotacji, po których algorytm się kończy.
  - ▶ Wyważenie aktualizowane na  $-1$  lub  $1$  wymaga aktualizacji zmiany w rodzicu (kontynuujemy algorytm).
  - ▶ Czas  $\Theta(\log n)$ .



## Drzewa AVL (3)

- ▶ Operacja usuwania – początek działa jak dla zwykłego BST, po czym należy przeprowadzić proces aktualizacji wyważeń od węzła wzwyż (maksymalnie do korzenia), przy czym:
  - ▶ Wyważenie aktualizowane na  $-1$  lub  $1$  kończy proces bez konieczności dalszych zmian.
  - ▶ Wyważenie aktualizowane na  $-2$  lub  $2$  oznacza, że drzewo straciło właściwość AVL. Naprawa przebiega z użyciem 1–2 rotacji, po czym proces należy kontynuować.
  - ▶ Wyważenie aktualizowane na  $0$  wymaga aktualizacji zmiany w rodzicu (kontynuujemy algorytm).
- ▶ Czas  $O(\log n)$ .



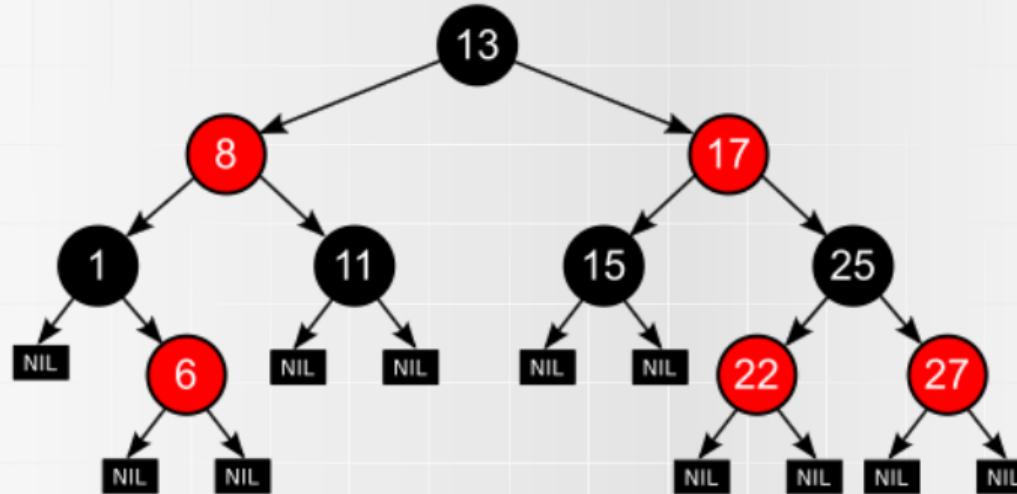
## Drzewa czerwono-czarne (1)

- ▶ Samorównoważące się BST.
- ▶ Każdy węzeł ma kolor (1 bit) a drzewo musi spełniać następujące własności:
  - ▶ Każdy węzeł jest czerwony lub czarny.
  - ▶ Korzeń jest czarny.
  - ▶ Każdy liść jest czarny.
  - ▶ Synowie czerwonego są czarni.
  - ▶ Dla węzła  $w$  ścieżki od  $w$  do jego potomków-liści mają tyle samo węzłów czarnych.
- ▶ Dla tego drzewa przez liść rozumiemy null (nil)!

Wyszukiwanie odbywa się identycznie jak dla normalnego BST, ale dzięki wyważeniu, gwarantowany jest czas  $O(\log n)$ .

# Drzewa czerwono-czarne (2)

Przykład drzewa czerwono-czarnego.



Dzięki własności, dla każdego węzła  $w$  jego najdłuższa ścieżka do liścia jest co najwyżej  $2x$  dłuższa niż najkrótsza.



## Drzewa czerwono-czarne (3)

- ▶ Operacja wstawiania:
  - ▶ Początek działa jak dla zwykłego BST.
  - ▶ Nowy węzeł kolorowany jest wstępnie na czerwono.
  - ▶ Mogły zostać złamane niektóre własności drzewa.
  - ▶ Przeprowadzamy korektę zależnie od występującego przypadku (kilka możliwych), w czasie  $O(1)$ , w tym co najwyżej jedna rotacja.
  - ▶ Czas  $O(\log n)$ .



## Drzewa czerwono-czarne (4)

- ▶ Operacja usuwania:
  - ▶ Początek działa jak dla zwykłego BST.
  - ▶ Dalsza część jest bardziej skomplikowana.
    - ▶ Możliwe jest wiele przypadków, zależnie od koloru usuwanego węzła i liczby jego dzieci.
    - ▶ Jednakże przywracanie własności drzewa czerwono-czarnego przy usuwaniu zawsze wymaga co najwyżej 2 rotacji.
  - ▶ Czas  $O(\log n)$ .



# AVL vs red-black tree

- ▶ Oba drzewa zapewniają `find()`, `insert()` i `delete()` w czasie  $O(\log n)$ .
- ▶ Drzewa AVL:
  - ▶ Są lepiej wyważone i w praktyce `find()` jest dla nich szybsze.
  - ▶ Większy koszt operacji `insert()` oraz `delete()` – możliwa konieczność przywracania własności wzduż całej wysokości drzewa.
- ▶ Drzewa czerwono-czarne:
  - ▶ Nie gwarantują doskonałego wyważenia.
  - ▶ Koszt naprawy własności drzewa dla `insert()` i `delete()` jest  $O(1)$ .
  - ▶ Wybór zależny od tego których operacji spodziewamy się więcej.



## Drzewa zrównoważone

- ▶ Zbalansowane BST zapewnia wiele operacji w czasie  $O(\log n)$ , w tym znalezienie minimum i maksimum.
- ▶ Takie BST może więc posłużyć do implementacji kolejki priorytetowej.
  - ▶ Zaletą jest możliwość implementacji jednocześnie operacji extract-min jak i extract-max (kolejka priorytetowa „dwustronna” ).
  - ▶ Wadą jest mniejsza szybkość:
    - ▶ Operacje BST mają zwykle większą stałą niż operacje kopcowe.
    - ▶ Reprezentacja kopca z użyciem tablicy dynamicznej lepiej współpracuje z pamięcią podręczną procesora niż BST.



Wrocław  
University  
of Science  
and Technology

# Struktury danych

Wykład 8

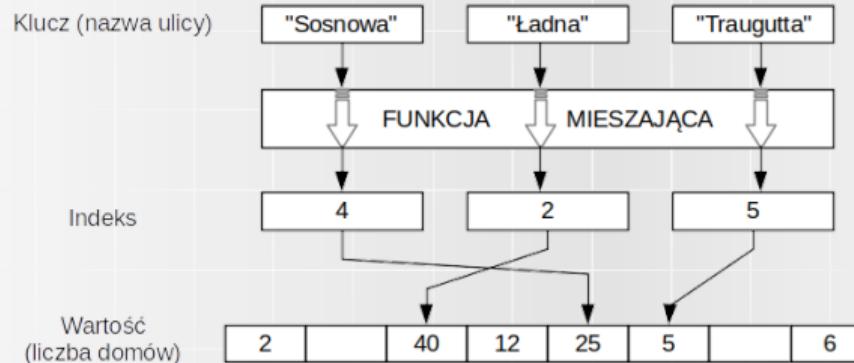
## Tablice mieszające

dr inż. Jarosław Rudy



# Tablica mieszająca (1)

- ▶ Tablica mieszająca (hash table) – struktura danych służąca do implementacji słownika.
- ▶ Za pomocą funkcji mieszającej klucz zamieniany jest na indeks.
- ▶ Indeks wykorzystywany jest do dostępu do tablicy „statycznej” .





## Funkcja mieszająca (1)

- ▶ Funkcja mieszająca  $h(x)$  przekształca klucz  $x$  na indeks elementu (kubełka) tablicy.
- ▶ Zbiór indeksów jest skończony i ma rozmiar  $m$  (nie mylić z  $n!$ ).
- ▶ Zbiór kluczy jest z reguły znacznie większy niż zbiór indeksów, może być też nieskończony.
- ▶ Z tego powodu funkcję mieszającą nazywa się też funkcją skrótu.
- ▶ Konieczność obliczenia funkcji mieszającej powoduje narzut czasowy.



## Funkcja mieszająca (2)

- ▶ Dla liczb całkowitych powszechną formą funkcji  $h(x)$  jest operacja modulo:

$$h(x) = x \bmod m \quad (1)$$

- ▶ Dzielenie jest stosunkowo powolne.
- ▶ Problem klasteryzacji.
- ▶ Wystarczająco dobre w praktyce w wielu przypadkach.
- ▶ Prostym wariantem dla łańcuchów tekstowych jest zsumowanie wszystkich znaków łańcucha.



## Funkcja mieszająca (3)

- ▶ Funkcja mieszająca z użyciem mnożenia:

$$h_\alpha(x) = \lfloor (\alpha x \bmod W)/(W/m) \rfloor \quad (2)$$

- ▶ Konieczność właściwego doboru  $\alpha$  i  $W$ .
- ▶ Haszowanie algebraiczne.
- ▶ Haszowanie Fibonacciego.
- ▶ Haszowanie z unikalną permutacją.
- ▶ Funkcja identyczności  $h(x) = x$ , sensowne jeśli zbiór kluczowy jest „mały”.



## Funkcja mieszająca (4)

- ▶ Ponieważ  $m$  jest mniejsze niż liczba kluczy, to będą istnieć takie klucze  $x_1 \neq x_2$ , że  $h(x_1) = h(x_2)$  (wynika to z zasady szufladkowej Dirichleta).
- ▶ Taką sytuację nazywamy kolizją.
- ▶ W przypadku kolizji dwa różne klucze mapowane są do tego samego miejsca (kubełka) w tablicy mieszającej.
  - ▶ Różne implementacje tablicy radzą sobie z tym w różny sposób.
  - ▶ Im mniej kolizji powoduje funkcja mieszająca tym lepiej.<sup>1</sup>

---

<sup>1</sup>W kryptografii dobiera się „większe” funkcje skrótu, przy których praktyczna szansa wystąpienia kolizji jest minimalna.



## Funkcja mieszająca (5)

### Paradoks dnia urodzin

Założymy, że w pokoju znajduje się  $k$  osób. Jakie jest prawdopodobieństwo, że co najmniej 2 z nich mają urodziny tego samego dnia?

- ▶ Dla  $k = 1$  szansa jest 0%.
- ▶ Dla  $k = 367$  szansa jest 100% (licząc rok przestępny).
- ▶ Dla jakiego  $k$  szansa jest  $\geq 50\%$ ? Naiwna interpolacja wskazała by wartość  $k \geq 183$ , ale w rzeczywistości jest to  $k \geq 23$ !
- ▶ Wniosek: pierwsza kolizja może wystąpić szybciej niż mogło by się wydawać!



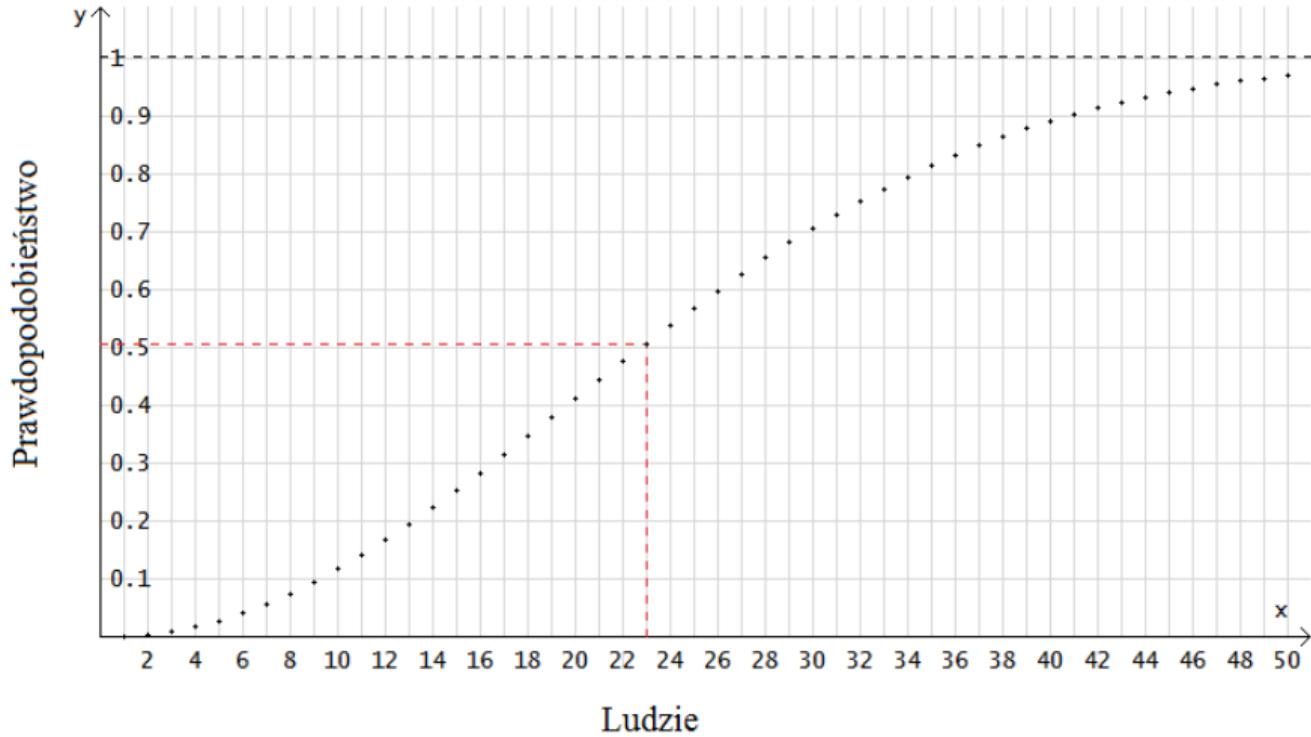
## Funkcja mieszająca (6)

Założymy tablicę mieszającą z  $m = 10^6$  i jednorodną funkcją mieszającą. Szansa na wystąpienie kolizji dla  $k$  operacji `insert()` wynosi:

- ▶ 10% przy  $k = 460$ ,
- ▶ 25% przy  $k = 759$ ,
- ▶ 50% przy  $k = 1178$ ,
- ▶ 90% przy  $k = 2146$ ,
- ▶ 99% przy  $k = 3034$  ( $0,3034\% m!$ ).



## Funkcja mieszająca (7)





## Funkcja mieszająca (8)

### Idealna funkcja mieszająca

- ▶ Przyporządkowuje każdemu kluczowi osobny indeks (kubek).
- ▶ Matematycznie jest funkcją różnowartościową.
- ▶ Brak kolizji.
- ▶ Możliwa do skonstruowania, gdy klucze znane są zawsze.
- ▶ Idealna funkcja mieszająca jest minimalna, gdy indeksy są kolejnymi liczbami całkowitymi.



## Funkcja mieszająca (9)

Cechy dobrej funkcji mieszającej:

- ▶ Deterministyczna.
- ▶ Jednorodna – każdy indeks powinien mieć zblizoną liczbę kluczy, które się na niego mapują.
- ▶ Małe zmiany klucza powinny powodować duże zmiany indeksu (redukuje problem klasteryzacji).
- ▶ Kompromis czasu obliczenia względem częstości kolizji.
- ▶ Pozwala na różny rozmiar klucza i różne wartości  $m$ .



# Rozwiązywanie kolizji (1)

Istnieją dwa podstawowe sposoby rozwiązywania kolizji:

- ▶ Metoda łańcuchowa (separate chaining).
  - ▶ Pojedynczy kubełek przechowuje wszystkie klucze, które się do niego mapują.
  - ▶ Konieczność organizacji wielu wartości w ramach kubełka.
- ▶ Adresowanie otwarte (open addressing).
  - ▶ Każdy kubełek przechowuje najwyżej 1 element.
  - ▶ Gdy drugi element zmapuje się do zajętego kubełka, to trzeba wyznaczyć mu inny kubełek.



# Współczynnik zajętości (1)

- ▶ Najważniejszym parametrem opisującym tablicę mieszającą jest współczynnik zajętości (load factor), definiowany jako:

$$\alpha = \frac{n}{m}, \quad (3)$$

gdzie  $n$  to liczba elementów przechowywanych w strukturze, a  $m$  to liczba kubełków.

- ▶ Wysokość load factor znacząco wpływa na wydajność operacji na tablicy mieszającej.



## Współczynnik zajętości (2)

- ▶ Dla open addressing dopuszczalne wartości  $\alpha$  są w przedziale  $[0, 1]$  tj.  $n \leq m$ .
  - ▶ Nie można przechować więcej niż  $m$  kluczy, bo każda wymaga osobnego kubełka.
- ▶ Gdy  $\alpha$  zbliża się do 1, coraz trudniej znaleźć wolny kubełek i wydajność drastycznie spada.
- ▶ Gdy  $\alpha$  przekracza wartość graniczną (w praktyce od 0.6 do 0.8), to należy zwiększyć rozmiar tablicy.
- ▶ Często tablicę zmniejsza się, gdy  $\alpha$  spadnie do  $\frac{1}{4}$  wartości granicznej.
  - ▶ Pozwala ograniczyć zużycie pamięci (typowo połowa lub więcej tablicy jest pusta!).

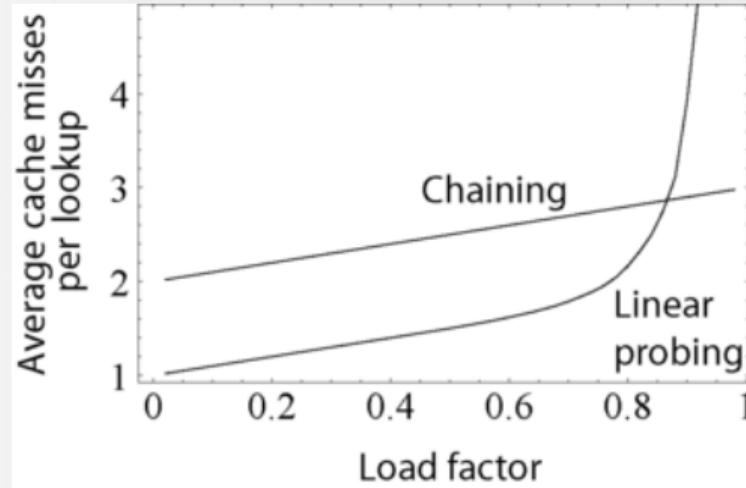


## Współczynnik zajętości (3)

- ▶ Dla separate chaining w teorii  $\alpha$  może być dowolnie duże dla stałego  $m$ .
  - ▶ Oznacza to, że średnio w jednym kubełku jest  $\alpha$  kluczy.
- ▶ Wydajność jednak wciąż spada wraz ze wzrostem  $\alpha$ , więc w praktyce stosuje się „miękką” wartość graniczną (zwykle pomiędzy 1 a 3), po przekroczeniu której zwiększa się rozmiar tablicy.
- ▶ Analogicznie tablicę można zmniejszać, gdy  $\alpha$  odpowiednio spadnie.

## Współczynnik zajętości (4)

Metody open addressing (np. linear probing) mogą średnio wymagać mniej czasu niż separate chaining, o ile  $\alpha$  nie jest bliskie 1.



W praktyce dobrze zarządzana tablica mieszająca typu open addressing rzadko potrzebuje przejrzeć więcej niż 3 kubelki.



## Zmiana rozmiaru (1)

- ▶ Zmiana rozmiaru tablicy oznacza zmianę liczby kubełków  $m$ , a to z kolei zmienia dopuszczalny zakres kluczy.
- ▶ Potrzebna jest więc nowa funkcja mieszająca, zaś operacja nie jest oczywista.
- ▶ Typowe metody zwiększenia rozmiaru:
  - ▶ Zmiana jednorazowa.
  - ▶ Zmiana stopniowa.
  - ▶ Linear hashing.



## Zmiana rozmiaru (2)

Zmiana jednorazowa:

- ▶ Dana jest stara tablica mieszajaca z funkcja mieszająca  $h_{\text{old}}(x)$ .
- ▶ Tworzymy nową (większą) tablicę mieszającą.
- ▶ Tworzymy nową funkcje mieszającą  $h_{\text{new}}(x)$ .
- ▶ Dla każdego zajętego kubełka w starej tablicy dodajemy jego zawartość do nowej tablicy (z użyciem  $h_{\text{new}}(x)$ ).
- ▶ Zajmuje dużo czasu.
  - ▶ Jeśli nowa tablica jest o czynnik (np. 2x) większa od starej, to zamortyzowany koszt zwiększenia jest stały (analogicznie jak dla tablicy dynamicznej).



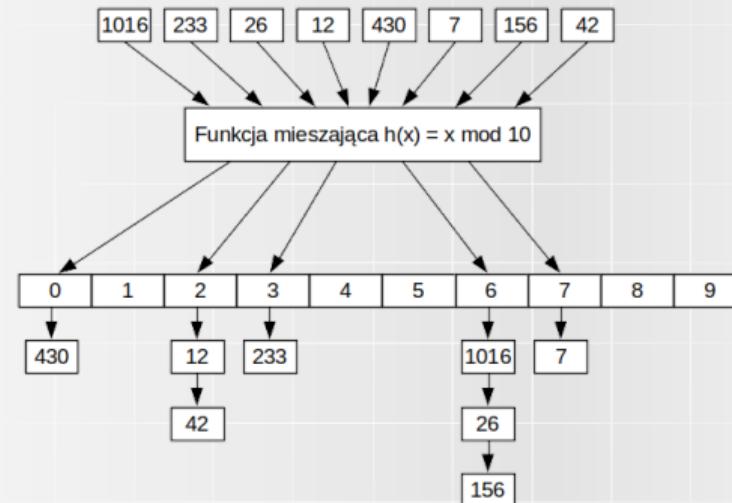
## Zmiana rozmiaru (3)

Zmiana stopniowa:

- ▶ Tworzymy nową tablicę i funkcję mieszającą  $h_{\text{new}}(x)$ .
- ▶ Na czas przenosin stosowane są obie tablice:
  - ▶ `find()` przeszukuje obie tablice (używając  $h_{\text{old}}(x)$  oraz  $h_{\text{new}}(x)$ ).
  - ▶ `insert()` dodaje do nowej tablicy (używając  $h_{\text{new}}(x)$ ).
  - ▶ Każdy `insert()` przenosi też  $k$  wpisów ze starej tablicy do nowej.
- ▶ Gdy przeniesione zostaną wszystkie elementy, starą tablicę można usunąć.

# Separate chaining (1)

- ▶ Domyślnie separate chaining w każdym kubełku przechowuje listę wiązaną elementów.
- ▶ Operacje `find()`/`insert()`/`delete()` muszą operować też na liście.





## Separate chaining (2)

- ▶ Lista wiązana ma pesymistyczny czas wyszukiwania  $O(n)$ , rzutujący na złożoność całej tablicy.
  - ▶ Możemy poprawić stosując zbalansowane drzewa BST.
- ▶ Lista wiązana słabiej współpracuje z pamięcią podręczną procesora.
  - ▶ Możemy poprawić stosując tablicę dynamiczną.
  - ▶ Problem pamięci podręcznej dotyczy też samej głównej tablicy – sięgamy do „losowych” indeksów, część indeksów jest pusta.



## Metoda 2-choice hashing:

- ▶ Modyfikacja separate chaining (choć może też działać dla innych metod rozwiązywania kolizji).
- ▶ Jedna tablica, dwie funkcje mieszające  $h_1(x)$  oraz  $h_2(x)$ .
- ▶ Podczas `insert()` stosowane są obie funkcje, zaś element trafia do kubełka mającego mniej elementów.
  - ▶ Jeśli kubełki mają równy rozmiar, to wykorzystujemy ten, który wskazała funkcja  $h_1(x)$ .
- ▶ Podobnie `find()` poszukuje elementu w obu możliwych kubełkach.
- ▶ Dzięki zasadzie power of 2 choices, znaczaco zmniejszamy możliwość wystąpienia dużych kubełków.



# Open addressing (1)

Metoda open addressing:

- ▶ Przy dodawaniu elementu najpierw obliczamy standardowo indeks funkcją mieszającą.
- ▶ Jeśli wyznaczony kubełek jest zajęty, wyznaczamy kolejny kubełek do sprawdzenia.
- ▶ Jeśli nowy kubełek jest zajęty, to sytuacja się powtarza aż do znalezienia pustego kubełka.
  - ▶ Powstaje ciąg poszukiwań (probing sequence), zaś jego długość określa wydajność tablicy.
- ▶ Analogiczny proces należy przeprowadzić dla find() i delete().



## Open addressing (2)

Istnieją różne sposoby tworzenia ciągu poszukiwań:

- ▶ Linear probing – kolejny sprawdzany kubełek jest w odstępie  $C$  kubełków (często  $C = 1$ ) od poprzedniego.
- ▶ Indeks  $k$ -tego kubełka do sprawdzenia dany jest więc wzorem:

$$h(x) + kC. \quad (4)$$

- ▶ Istotne jest by funkcja mieszająca unikała klasteryzacji/grupowania, gdzie indeksy kluczów gromadzą się blisko siebie.
  - ▶ Grupowanie, nawet jeśli nie powoduje kolizji, jest szkodliwe dla adresowania otwartego (ale nie dla metody łańcuchowej).
  - ▶ Lepiej współpracuje z pamięcią podręczną procesora (dla małych  $C$ ).



## Open addressing (3)

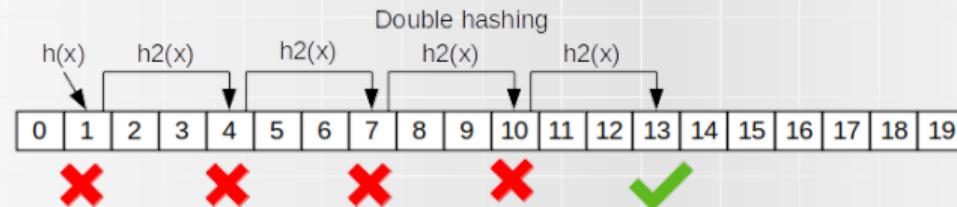
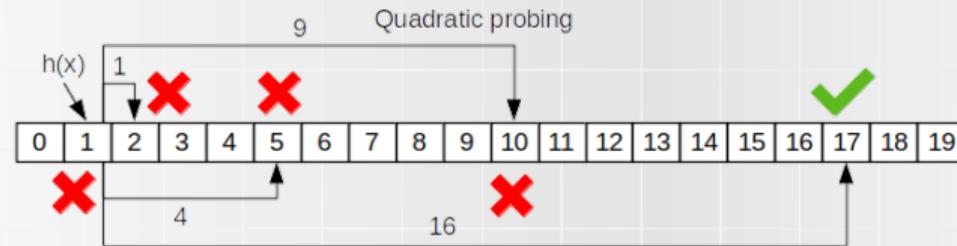
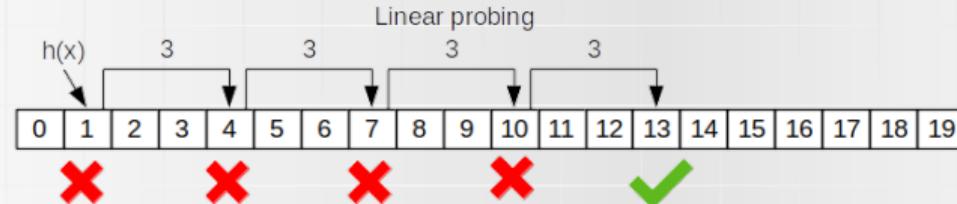
- ▶ Quadratic probing – kolejne odstępy od początkowego indeksu  $h(x)$  dane są kolejnymi wartościami pewnej funkcji kwadratowej.
- ▶ Bardziej odporne na grupowanie, podatne na grupowanie wtórne.
- ▶ Double hashing – odstęp jest liniowy, ale zależy od klucza  $x$  i dany drugą funkcją mieszającą  $h_2(x)$ . Indeks w  $k$ -tej próbie wynosi więc:

$$h(x) + kh_2(x). \quad (5)$$

- ▶ Odporne na problem grupowania.



# Open addressing (4)





# Cuckoo hashing (1)

- ▶ Haszowanie kukułcze – odmiana open addressing.
- ▶ Dwie tablice  $T_1$  oraz  $T_2$  o równym rozmiarze.
- ▶ Tablice mają osobne funkcje haszujące, odpowiednio  $h_1(x)$  oraz  $h_2(x)$ .
- ▶ Każdy klucz ma więc dwie możliwe lokalizacje: podstawową oraz alternatywną.
- ▶ Przy dodawaniu nowy element  $x_1$  wstawiany jest do  $T_1$ . Jeśli był tam już jakiś inny element  $x_2$ , to usuwamy  $x_2$  z  $T_1$  i wstawiamy go na jego alternatywne miejsce w  $T_2$ .
  - ▶ Procedura się powtarza dla  $x_2$ : jeśli miejsce  $x_2$  w  $T_2$  zajmowało  $x_3$ , to  $x_3$  jest usuwany i wstawiany w swoje alternatywne miejsce w  $T_1$ .



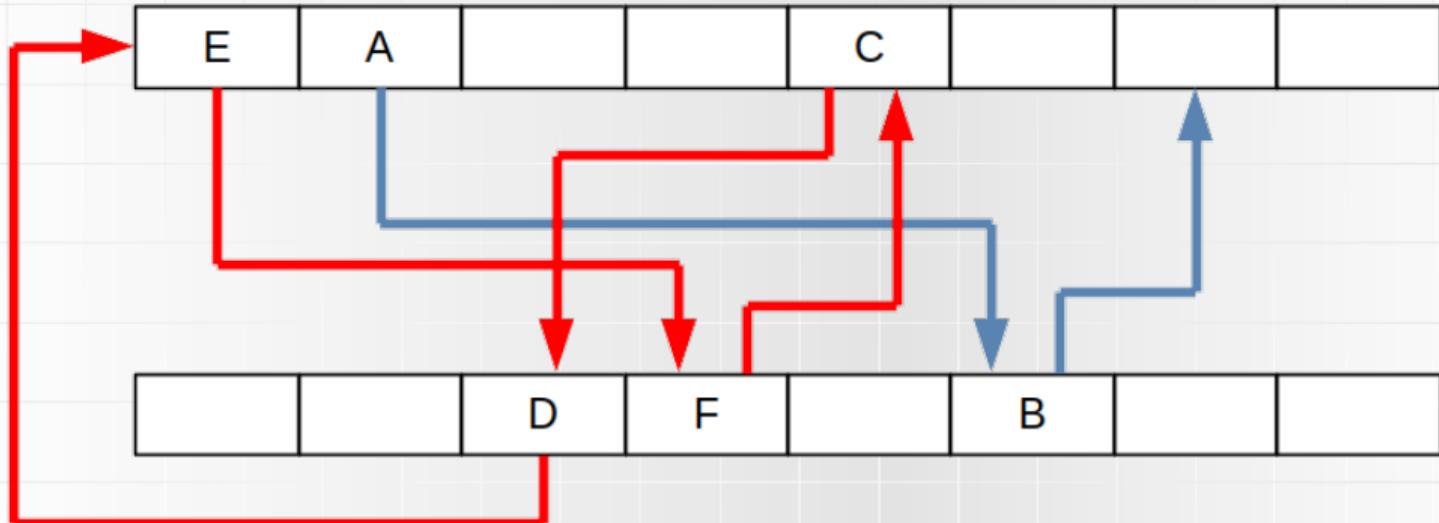
## Cuckoo hashing (2)

- ▶ Może wystąpić cykl.
  - ▶ Wykrywany przez przekroczenie licznika kroków wstawiania.
  - ▶ Naprawiany poprzez stworzenie nowych funkcji  $h_1(x)$  i  $h_2(x)$  oraz ponowne haszowanie zawartości tablic.
- ▶ Każdy klucz jest w  $h_1(x)$  w  $T_1$  lub w  $h_2(x)$  w  $T_2$ .
  - ▶ `find()` jest więc w pesymistycznym czasie  $O(1)$ , podobnie `remove()`.
  - ▶ `insert()` może zajść długi czas, ale koszt zamortyzowany jest  $O(1)$ , nawet uwzględniając konieczność przehaszowania tablic.



## Cuckoo hashing (3)

Wroclaw  
University  
of Science  
and Technology





## Cuckoo hashing (4)

- ▶ Haszowanie kukułcze ma więc bardzo dobrą złożoność teoretyczną.
- ▶ Złożoność zakłada jednak, że load factor jest poniżej 0.5.
  - ▶ Niefektywne użycie pamięci.
  - ▶ Można użyć 3 funkcji i trzech tablic, co średnio pozwala wykorzystać ponad 90% miejsca kosztem spadku wydajności (jednak wciąż  $O(1)$ ).
- ▶ W praktyce cuckoo hashing jest średnio wolniejsze od linear probing (więcej chybień podczas `find()`), ale może być przydatne w sytuacjach gdzie ważna jest redukcja przypadku pesymistycznego.



# Zastosowania

Zastosowania tablic mieszających:

- ▶ Słowniki.
- ▶ Zbiory (brak kolejności).
- ▶ Pamięci podręczne.
- ▶ Indeksy baz danych.
- ▶ Tablice transpozycji gier.



# Słowniki – podsumowanie (1)

Struktura	find()/remove()		insert()		Uporządk.
	Avg.	Worst	Avg.	Worst	
Hash table <sup>2</sup>	$O(1)$	$O(n)$	$O(1)$	$O(n)$	nie
Hash table <sup>3</sup>	$O(1)$	$O(\log n)$	$O(1)$	$O(\log n)$	nie
Hash table <sup>4</sup>	$O(1)$	$O(1)$	$O(1)$	$O(1)^5$	nie
AVL/black-red tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	tak
BST	$O(\log n)$	$O(n)$	$O(\log n)$	$O(n)$	tak
Lista <sup>6</sup>	$O(n)$	$O(n)$	$O(1)/O(n)$	$O(1)/O(n)$	„tak”

<sup>2</sup>Adresowanie otwarte lub kubełki z listą

<sup>3</sup>Adresowanie zamknięte plus kubełki ze zbalansowanym BST

<sup>4</sup>Cuckoo hashing

<sup>5</sup>Koszt zamortyzowany

<sup>6</sup>Dwa warianty: dodawanie na koniec lub dodawanie w pozycji posortowanej



Wrocław  
University  
of Science  
and Technology

# Struktury danych

Wykład 9  
Grafy

dr inż. Jarosław Rudy





## Graf (1)

- ▶ Graf jako ADT służy do reprezentacji grafów matematycznych.
- ▶ Matematycznie graf składa się z wierzchołków (węzłów, vertices) i łączących je krawędzi (edges).
  - ▶ Każda krawędź łączy dwa (domyślnie różne) wierzchołki grafu.
- ▶ Ścisłej graf  $G$  jest parą uporządkowaną (dwójką) zbioru wierzchołków  $V$  i zbioru krawędzi  $E$ :

$$G = (V, E). \quad (1)$$

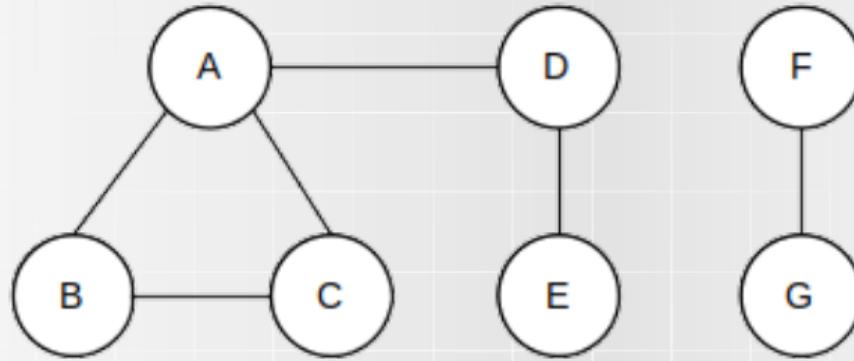
- ▶ Krawędzie zwykłe definiuje się jako zbiór  $E$  zbiorów dwuelementowych, tak że  $E$  jest podzbiorem zbioru „wszystkich” możliwych krawędzi:

$$E \subseteq \{\{u, v\} : u, v \neq u \in V\}. \quad (2)$$

krawędź  $\{u, v\}$  łączy wierzchołki  $u$  i  $v$ .

# Graf (2)

Przykład grafu



$$V = \{A, B, C, D, E, F, G\}$$

$$E = \{\{A, B\}, \{B, C\}, \{A, C\}, \{A, D\}, \{D, E\}, \{G, F\}\}$$



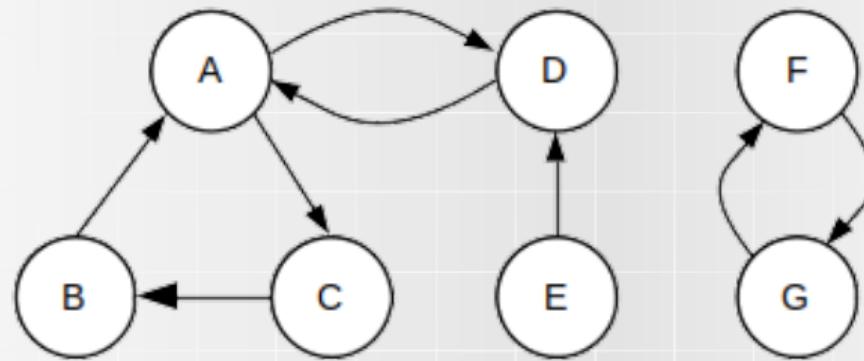
## Graf (3)

- ▶ Poprzednia definicja zakłada, że kierunek krawędzi nie ma znaczenia (graf jest nieskierowany).
- ▶ Można jednak zdefiniować zbiór krawędzi  $A$  tak by krawędzie miały kierunek tj. każda krawędź jest parą uporządkowaną

$$A \subseteq \{(u, v) : u, v \neq u \in V\}. \quad (3)$$

- ▶ Krawędź  $(u, v)$  biegnie od  $u$  do  $v$  i jest czymś innym niż krawędź  $(v, u)$ .
- ▶ Taki graf  $G = (V, A)$  nazywamy grafem skierowanym, zaś jego krawędzie nazywamy łukami (arcs).

## Przykład grafu



$$V = \{A, B, C, D, E, F, G\}$$

$$A = \{(A, C), (B, A), (C, B), (A, D), (D, A), (E, D), (F, G), (G, F)\}$$



## Graf (5)

- ▶ Często przyjmowana notacja:
  - ▶  $n = |V|$  – liczba wierzchołków grafu.
  - ▶  $k = |E|$  (lub  $k = |A|$ ) – liczba krawędzi/łuków grafu.
- ▶ Według standardowej definicji graf skierowany może mieć od 0 do  $n(n - 1)$  łuków.
- ▶ Analogicznie, graf nieskierowany może mieć od 0 do  $\frac{n(n-1)}{2}$  krawędzi.

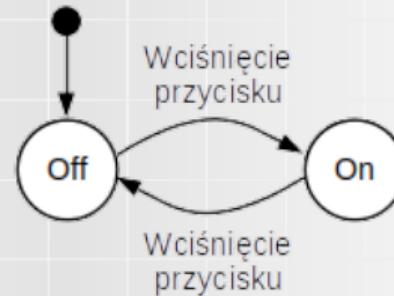
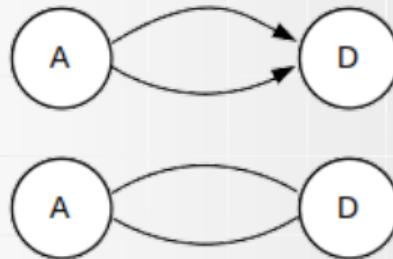
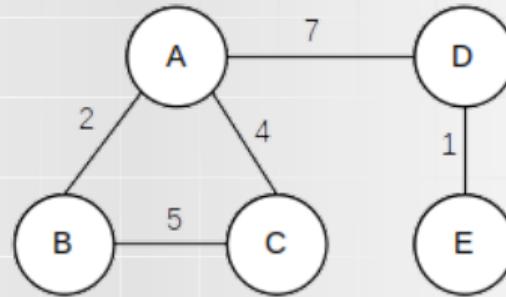
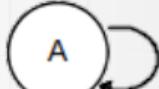


## Graf (6)

Można wprowadzać modyfikacje definicji grafu

- ▶ Dopuszczanie pętli tj. krawędzi/łuków zaczynających się i kończących się w tym samym wierzchołku.
- ▶ Multigraf – dopuszczanie wielokrotnych krawędzi pomiędzy tą samą parą wierzchołków.
- ▶ Graf z wagami – krawędziom i/lub wierzchołkom można przypisywać wartości liczbowe (wagi).
- ▶ Ogólniej można przepisywać dowolne wartości – etykiety.

## Graf (7)





# Typy grafów i pojęcia grafowe (1)

Wybrane typy grafów i pojęcia grafowe:

- ▶ Rząd grafu – liczba wierzchołków.
- ▶ Rozmiar grafu – liczba krawędzi.
- ▶ Droga – ciąg następujących po sobie krawędzi (trasa). Niekiedy w skład drogi wchodzą też znajdujące się na niej wierzchołki.
  - ▶ Droga prosta – droga na której nie powtarzają się krawędzie.
  - ▶ Cykl – droga kończąca się w tym samym wierzchołku, w którym się zaczęła.



## Typy grafów i pojęcia grafowe (2)

- ▶ Stopień wierzchołka (degree, deg):
  - ▶  $\deg(v)$  – w grafie nieskierowanym liczba krawędzi wchodzących do lub wychodzących z  $v$ .
  - ▶  $\text{indeg}(v)$  – w grafie skierowanym liczba krawędzi wchodzących do  $v$ .
  - ▶  $\text{outdeg}(v)$  – w grafie skierowanym liczba krawędzi wychodzących z  $v$ .
- ▶ Wierzchołek izolowany – wierzchołek o stopniu 0 (z którego nie wychodzą i do którego nie wchodzą krawędzie).



## Typy grafów i pojęcia grafowe (3)

- ▶ Graf pełny – liczba krawędzi jest maksymalna.
- ▶ Graf gęsty – duża liczba krawędzi ( $k$ ) w stosunku do liczby wierzchołków ( $v$ ). Różne praktyczne definicje np.:
  - ▶  $k \notin O(n)$ .
  - ▶  $k \in \Theta(n^2)$ .
  - ▶  $k \gg v$ .
- ▶ Graf rzadki – nieduża liczba krawędzi np.:
  - ▶  $k \in O(n)$ .
  - ▶  $k \leq n$ .
- ▶ Graf pusty – graf bez wierzchołków lub graf bez krawędzi.



## Typy grafów i pojęcia grafowe (4)

- ▶ Graf spójny (connected graph) – graf, w którym istnieje droga pomiędzy każdą parą wierzchołków  $u, v \neq u$ .
- ▶ Spójna składowa grafu (component) – maksymalny (tj. taki którego nie można już powiększyć) spójny podgraf grafu. Graf może mieć wiele spójnych składowych.
- ▶ Drzewo – graf nieskierowany, acykliczny (brak cykli) i spójny.
  - ▶ Dokładnie 1 droga pomiędzy każdą parą wierzchołków.



## Typy grafów i pojęcia grafowe (5)

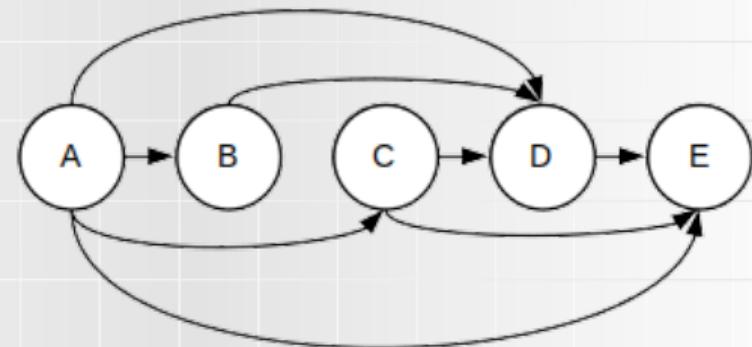
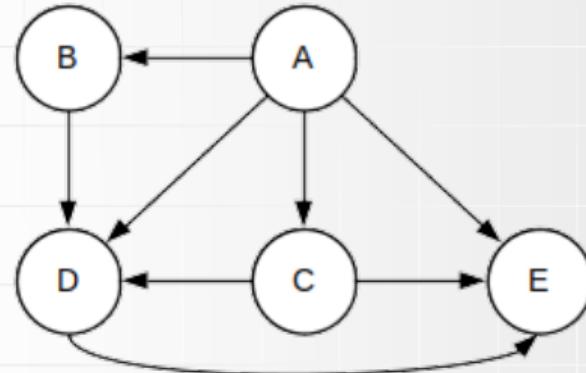
### Skierowany graf acykliczny (directed acyclic graph, DAG)

- ▶ Graf skierowany, w którym nie ma skierowanych cykli.
- ▶ W DAG podążanie krawędziami (zgodnie z ich kierunkiem) nigdy nie doprowadzi do cyklu.
  - ▶ Jeśli jest krawędź  $(u, v)$  to nie ma krawędzi  $(v, u)$ .
- ▶ Dla każdego DAG można wyznaczyć kolejność topologiczną, pewne wierzchołki są przed innymi („brak możliwości cofania się”).
- ▶ DAG są przydatne do modelowania pewnych procesów np. w produkcji.



## Typy grafów i pojęcia grafowe (6)

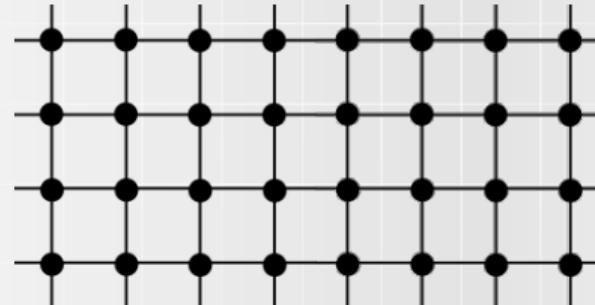
Przykładowy DAG i jego przykładowa kolejność topologiczna



## Typy grafów i pojęcia grafowe (8)

Graf typu krata/siatka (grid/lattice graph):

- ▶ Graf który po narysowaniu w Euklidesowej przestrzeni 2D ma strukturę regularnych kafelków.
- ▶ Kafelki mogą być kwadratowe, trójkątne, sześciennne itd.
- ▶ Przydatne do modelowania pewnych procesów np. w produkcji.





# Reprezentacje grafu (1)

Różne sposoby przechowywania (struktury) grafu w pamięci komputera:

- ▶ Macierz sąsiedztwa (adjacency matrix).
- ▶ Lista sąsiedztwa (adjacency list).
- ▶ Lista krawędzi (edge list).
- ▶ Macierz incydencji (incidence matrix).

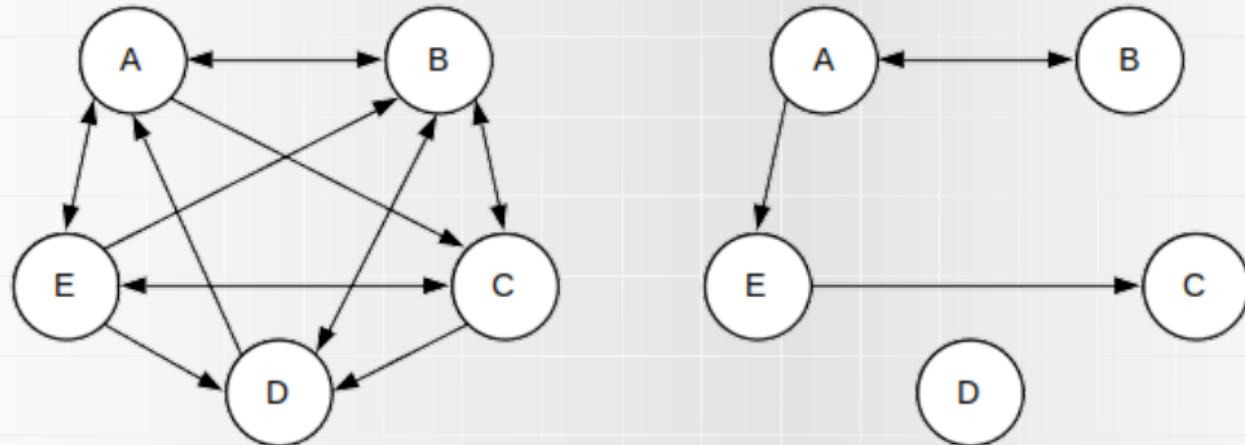
Istotne operacje:

- ▶ Dostęp do łuku  $(u, v)$  (lub krawędzi  $\{u, v\}$ ).
- ▶ Znalezienie (lub przegląd) sąsiadów wierzchołka  $v$ .

## Reprezentacje grafu (2)

Jak przykłady użyjemy dwóch grafów skierowanych:

- ▶ Gęstego  $v = 5, k = 15$  (75% krawędzi).
- ▶ Rzadkiego  $v = 5, k = 4$  (20% krawędzi).





## Macierz sąsiedztwa (1)

- ▶ Macierz dwuwymiarowa rozmiaru  $n \times n$ .
- ▶ W indeksie  $i, j$  macierzy przechowywana jest informacja o krawędzi  $(i, j)$  (lub  $\{i, j\}$ ).
  - ▶ Dla grafów bez wag/etykiet przechowujemy jedynie czy krawędź istnieje czy nie.
  - ▶ W innym przypadku przechowujemy wagę, etykietę lub wręcz wskaźnik/referencję na obiekt krawędzi (dbając o odpowiednie wartości jeśli krawędź nie istnieje).
- ▶ Dla grafów nieskierowanych  $\{u, v\} = \{v, u\}$ , więc przechowujemy tylko połowę macierzy (np. macierz górnopróbkątna).
- ▶ Realizowana tablicą dwuwymiarową, tablicami jednowymiarowymi, tablicami dynamicznymi itd.



## Macierz sąsiedztwa (2)

Dla przykładowych grafów:

	A	B	C	D	E
A	0	1	1	0	1
B	1	0	1	1	0
C	0	1	0	1	1
D	1	1	0	0	0
E	1	1	1	1	0

	A	B	C	D	E
A	0	1	0	0	1
B	1	0	0	0	0
C	0	0	0	0	0
D	0	0	0	0	0
E	0	0	1	0	0



## Macierz sąsiedztwa (3)

- ▶ Dostęp do krawędzi w czasie  $O(1)$ .
- ▶ Przegląd sąsiadów wierzchołka  $v$  w czasie  $n \in \Theta(n)$ .
  - ▶ Niezależne od tego ile wynosi  $N(v)$ , czyli liczba sąsiadów  $v$ !
  - ▶ Potrzebujemy co najmniej  $n^2$  (czyli  $O(n^2)$ ) pamięci.
    - ▶ Niezależne od  $k$ !
    - ▶ Wartości na przekątnej są bezużyteczne, chyba że graf dopuszcza pętle.
  - ▶ Dość wydajna (pamięciowo i czasowo) dla grafów gęstych.

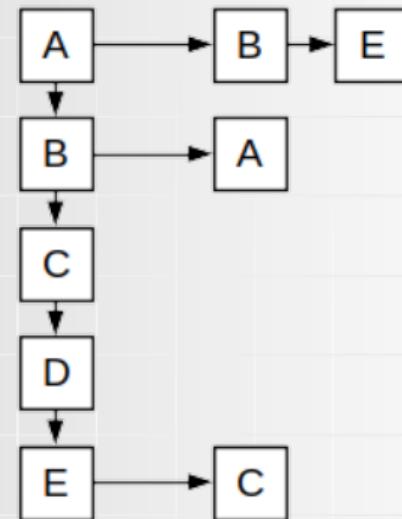
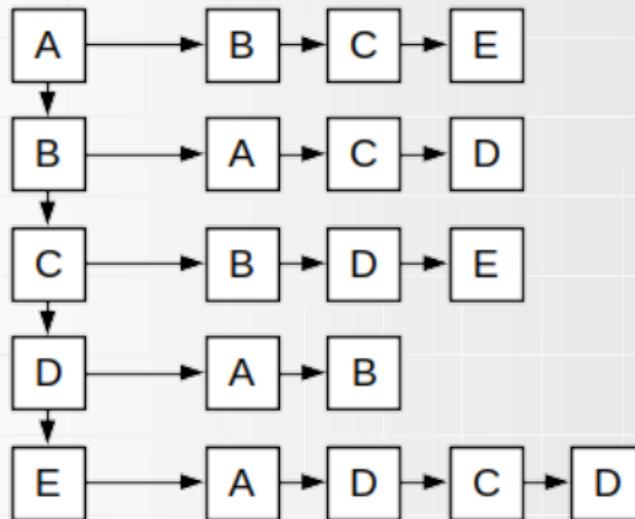


## Lista sąsiedztwa (1)

- ▶ Lista rozmiaru  $n$ .
- ▶ Elementami listy są kolejne listy.
- ▶ Lista  $i$ -ta przechowuje krawędzie zaczynające się w wierzchołku  $i$ -tym.
  - ▶ Najprościej przechować numer wierzchołka którym krawędź się kończy.
- ▶ Ponieważ przechowujemy tylko faktycznych sąsiadów, to każda podlista może mieć różną długość.

## Lista sąsiedztwa (2)

Dla przykładowych grafów:





## Lista sąsiedztwa (3)

- ▶ Lista wierzchołka  $v$  ma długość  $N(v)$ , zamiast  $n$ .
  - ▶ Przegląd sąsiadów  $v$  w czasie  $O(N(v))$ .
- ▶ Zwykle mniejsze zużycie pamięci  $O(n + k)$ .
  - ▶ Wydajna dla grafów rzadkich.
  - ▶ Czasami może wymagać więcej pamięci (np. graf pełny plus dopuszczanie pętli).
- ▶ Dostęp do krawędzi wierzchołka  $v$  w czasie  $O(N(v)) \in O(n)$ .



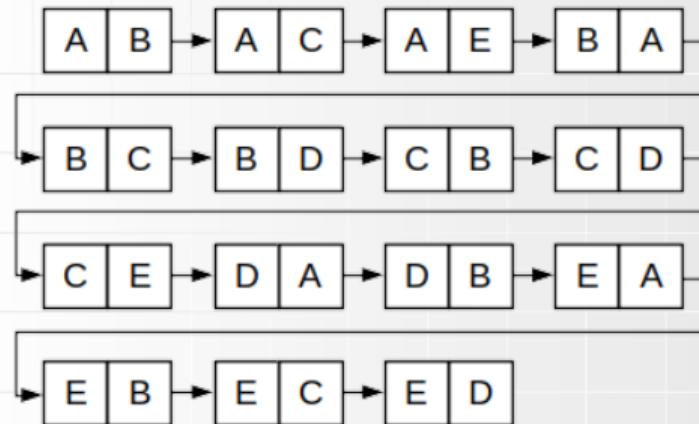
# Lista krawędzi (1)

- ▶ Lista rozmiaru  $k$ .
- ▶ Elementami listy są krawędzie.
  - ▶ Najprościej przechować krawędź jako parę: wierzchołek początkowy i końcowy.
- ▶ Zwykle krawędzie przechowywane są w losowej kolejności.
- ▶ Można przyjąć pewien porządek, ale nie ma to większego wpływu na efekt.



## Lista krawędzi (2)

Dla przykładowych grafów:





## Lista krawędzi (3)

- ▶ Lista krawędzi zajmuje  $O(k)$  pamięci.
  - ▶ Każdy element przechowuję parę, więc łączna pamięć jest zwykle większa niż dla listy sąsiedztwa.
- ▶ Dostępu do krawędzi wierzchołka  $v$  w czasie  $O(k)$ .
- ▶ Czas przeglądnięcia sąsiadów wierzchołka  $v$  w czasie  $O(k)$ .
- ▶ Zwykle skrajnie niewydajna dla grafów gęstych.
- ▶ Może być przydatna dla niektórych algorytmów (np. zmodyfikowana w kolejkę priorytetową).



# Macierz indydencji (1)

- ▶ Macierz rozmiaru  $nk$ .
- ▶ Indeks  $i, j$  przechowuje informacje czy  $j$ -ta krawędź jest incydentna (łączy się) z wierzchołkiem  $i$ . Typowe wartości:
  - ▶ 1 – krawędź zaczyna się w tym wierzchołku.
  - ▶  $-1$  – krawędź kończy się w tym wierzchołku.
  - ▶ 0 – krawędź nie jest incydentna z wierzchołkiem.
- ▶ Zwykle przechowuje tylko informacje o incydencji, ale można też dołączyć inne informacje (waga, etykieta).



## Macierz incydencji (2)

Dla przykładowych grafów:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	1	1	1	-1	0	0	0	0	0	-1	0	-1	0	0	0
B	-1	0	0	1	1	1	-1	0	0	0	-1	0	-1	0	0
C	0	-1	0	0	-1	0	1	1	1	0	0	0	0	-1	0
D	0	0	0	0	0	-1	0	-1	0	1	1	0	0	0	-1
E	0	0	-1	0	0	0	0	0	-1	0	0	1	1	1	1

	1	2	3	4
A	1	1	-1	0
B	-1	0	1	0
C	0	0	0	-1
D	0	0	0	0
E	0	-1	0	1

Zakładamy kolejność krawędzi taką jak pokazano w liście krawędzi.



## Macierz incydencji (3)

- ▶ Zajmuje  $O(nk)$  pamięci.
  - ▶ Czyli rzadko mniej niż  $O(n^2)$ , nawet dla grafów rzadkich.
  - ▶  $\frac{n-2}{n}$  tablicy zawiera zera.
- ▶ Dostęp do krawędzi  $(u, v)$  w czasie  $O(k)$ .
- ▶ Przeglądnięcie wszystkich sąsiadów w czasie  $O(k + nN(v))$ .
- ▶ Przydatna gdyby krawędź miała więcej niż 2 końce.



## Reprezentacje grafu (3)

Reprezentacja	Pamięć	Dostęp do krawędzi	Przegląd sąsiadów
Macierz sąsiedztwa	$O(n^2)$	$O(1)$	$O(n)$
Lista sąsiedztwa	$O(n + k)$	$O(N(v))$	$O(N(v))$
Lista krawędzi	$O(k)$	$O(k)$	$O(k)$
Macierz incydencji	$O(nk)$	$O(k)$	$O(k + nN(v))$



## Przeszukiwanie grafu (1)

- ▶ Analogicznie jak w przypadku drzew istnieją algorytmy do przeglądu grafu tj. do odwiedzenia wszystkich wierzchołków.
- ▶ Podstawowy algorytm:
  - ▶ Dodajemy pewien wierzchołek początkowy  $s$  do kolejki  $Q$ .
  - ▶ Dopóki  $Q$  nie jest puste:
    - ▶ Pobieramy (usuwamy) jeden wierzchołek  $v$  z  $Q$ .
    - ▶ Odwiedzamy  $v$ .
    - ▶ Dodajemy nieodwiedzonych jeszcze sąsiadów  $v$  do  $Q$ .
  - ▶ Węzły mogą być odkrywane wielokrotnie.
  - ▶ Problem z grafami niespójnymi.



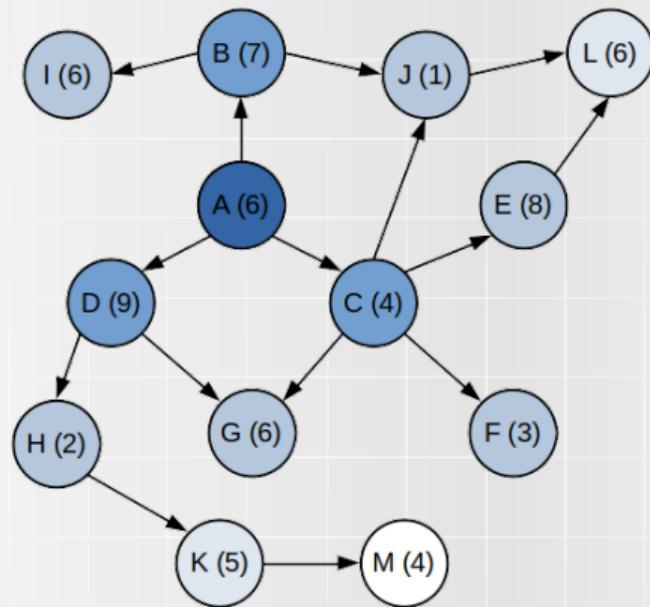
## Przeszukiwanie grafu (2)

Podstawowe typy przeglądu grafu:

- ▶ Przegląd wgłąb (depth-first) – ostatnio dodany wierzchołek odwiedzany jest najpierw ( $Q$  jest stosem).
- ▶ Przegląd wszerz (breadth-first) – ostatnio dodany wierzchołek odwiedzany jest na końcu ( $Q$  jest kolejką FIFO).
- ▶ Przegląd pierwszy najlepszy (best-first) – wierzchołki odwiedzamy wg priorytetu ( $Q$  jest kolejką priorytetową).
  - ▶ Dla grafów ważonych.
  - ▶ Priorytetem jest waga wierzchołka (na podobnej zasadzie działa algorytm Dijkstry) i/lub waga prowadzącej do niego krawędzi (np. algorytm najbliższego sąsiada dla problemu komiwojażera).

# Przeszukiwanie grafu (3)

Przykładowy skierowany graf ważony:





# Przeszukiwanie grafu (4)

Przykładowy przegląd dla przedstawionego grafu:

- ▶ Depth-first: A, D, H, K, M, G, C, J, L, F, E, B, I.
- ▶ Breadth-first: A, B, C, D, I, J, E, F, G, H, L, K, M
- ▶ Best-first: A, C, J, F, G, L, B, I, E, D, H, K, M.



# Zastosowania grafów (1)

- ▶ Transport, komunikacja (problem komiwojażera, sieci przepływowe).
- ▶ Analiza procesów (maszyny stanów, sieci Petriego).
- ▶ Modelowanie relacji/połączeń (social media, rozprzestrzenianie się chorób np. COVID-19).
- ▶ Poszukiwanie trasy.
- ▶ Optymalizacja produkcji (szeregowanie zadań).
- ▶ Inne.



Wrocław  
University  
of Science  
and Technology

# Struktury danych

Wykład 10

## Problem najkrótszej drogi

dr inż. Jarosław Rudy





# Problem najkrótszej drogi (1)

- ▶ W najprostszej wersji dany jest graf  $G = (V, E)$  oraz dwa wierzchołki: startowy  $v_s$  oraz końcowy  $v_e$ .
- ▶ Celem jest znalezienie (prostej) drogi od  $v_s$  do  $v_e$  o najmniejszej długości.
- ▶ Graf może być skierowany lub nie. Dla skierowanego istotne jest by poruszać się krawędziami w odpowiednim kierunku.
- ▶ Długość drogi liczona jest wagami krawędzi (ewentualnie krawędzi i wierzchołków).
  - ▶ Dla grafów bez wag długością drogi jest liczba jej krawędzi.
  - ▶ Droga od  $v_s$  do  $v_e$  może nie istnieć.



# Problem najkrótszej drogi (2)

Różne warianty problemu:

- ▶ Jedna para wierzchołków (single-pair shortest path problem) – znalezienie najkrótszej drogi od  $v_s$  do  $v_e$ .
  - ▶ Pesymistycznie i tak trzeba znaleźć drogę do innych wierzchołków.
- ▶ Tylko wierzchołek początkowy (single-source shortest path problem) – znalezienie najkrótszej drogi od  $v_s$  do każdego innego wierzchołka.
- ▶ Tylko wierzchołek końcowy (single-destination shortest path problem) – znalezienie najkrótszej drogi do  $v_e$  z każdego innego wierzchołka.
  - ▶ Może być sprowadzony do poprzedniego problemu (odwrócenie łuków).
- ▶ Wszystkie pary (all-pairs shortest path problem) – znalezienie najkrótszej drogi od każdego wierzchołka do każdego innego.



# Algorytm Dijkstry (1)

- ▶ Stworzony przez Edsgera Dijkstrę w 1956.
- ▶ Oryginalnie rozwiązywał single-pair shortest path problem.
- ▶ Obecnie jednak częsty wariant rozwiązuje single-source shortest path problem.
- ▶ Algorytm działa dla dowolnych skierowanych/nieskierowanych grafów ważowych bez ujemnych wag.
- ▶ Przy odpowiedniej implementacji i braku dodatkowych założeń (graf typu DAG, specyficzne wagi) jest to najlepszy znany algorytm dla tego problemu.



## Algorytm Dijkstry (2)

- ▶ Algorytm działa na zasadzie metody relaksacji – dla niektórych wierzchołków sprawdzamy czy nie da się zbudować do nich trasy krótszej niż najlepsza obecnie znana.
- ▶ Dla każdego wierzchołka  $v \in V$  przechowujemy (np. jako tablice) dwie informacje :
  - ▶  $dist[v]$  – długość najlepszej obecnie znanej drogi z  $v_s$  do  $v$  (długość tymczasowa).
  - ▶  $prev[v]$  – wierzchołek poprzedzający  $v$  na najlepszej obecnie znanej trasie od  $v_s$  do  $v$ .
- ▶ Potrzebujemy też przechowywać zbiór wierzchołków nieodwiedzonych *unvisited* oraz wierzchołek aktualny *current*.



## Algorytm Dijkstry (3)

Na początku ustawiamy:

- ▶  $unvisited \leftarrow V$ .
- ▶  $current \leftarrow v_s$ .
- ▶ Dla każdego  $v \in V$ :
  - ▶  $prev[v] \leftarrow undefined$  (np. -1)
  - ▶  $dist[v_s] \leftarrow 0$  (trasa od  $v_s$  do  $v_s$  ma znaną długość).
  - ▶ Dla każdego  $v \in V \setminus \{v_s\}$ :
    - ▶  $dist[v] \leftarrow \infty$  (np. -1)



# Algorytm Dijkstry (4)

Przebieg algorytmu:

1. Dla wierzchołka *current* sprawdzamy jego nieodwiedzonych sąsiadów i aktualizujemy ich tymczasową odległość, jeśli można do nich szybciej dotrzeć z *current*. Założymy sąsiada *neigh*:
  - ▶ Jeśli  $dist[current] + w(current, neigh) < dist[neigh]$  to:
    - ▶  $dist[neigh] \leftarrow dist[current] + w(current, neigh)$ .
    - ▶  $prev[neigh] \leftarrow current$ .
2. Usuwamy *current* z *unvisited*.
3. Jako *current* wybieramy wierzchołek *v* z *unvisited* o najmniejszym  $dist[v]$ .
4. Wracamy do punktu 1.



## Algorytm Dijkstry (5)

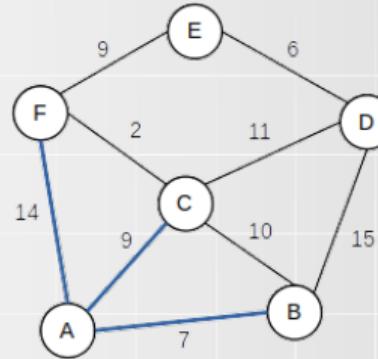
Warunek stopu algorytmu:

- ▶ Jeśli szukamy trasy od  $v_s$  do  $v_e$  algorytm kończy się, gdy:
  - ▶ W kroku 2 wierzchołek  $v_e$  został odwiedzony – możemy zbudować trasę na podstawie *prev*.
  - ▶ W kroku 3 za *current* ustawiono  $\infty$  – wierzchołek  $v_e$  jest nieosiągalny z wierzchołka  $v_s$ .
- ▶ Jeśli szukamy trasy od  $v_s$  do każdego innego wierzchołka to algorytm kończy się gdy w kroku 3 za *current* ustawiono  $\infty$ :
  - ▶ Oznacza to, że do wierzchołków obecnie w *unvisited* nie można dotrzeć z  $v_s$ . Trasy do pozostałych możemy zbudować w oparciu o *prev*.



## Algorytm Dijkstry (6)

Przykład szukania trasy od  $v_s = A$  do  $v_e = E$  (iteracja 1).



	A	B	C	D	E	F
dist	0	-1	-1	-1	-1	-1
prev	-1	-1	-1	-1	-1	-1

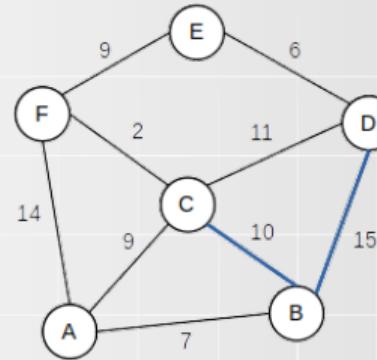
unvisited: { A , B , C , D , E , F }

current: A



## Algorytm Dijkstry (7)

Przykład szukania trasy od  $v_s = A$  do  $v_e = E$  (iteracja 2).



	A	B	C	D	E	F
dist	0	7	9	-1	-1	14
prev	-1	A	A	-1	-1	A

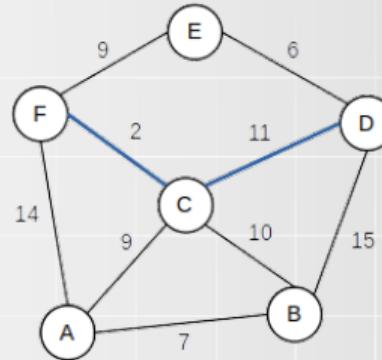
unvisited: { B , C , D , E , F }

current: B



## Algorytm Dijkstry (8)

Przykład szukania trasy od  $v_s = A$  do  $v_e = E$  (iteracja 3).



	A	B	C	D	E	F
dist	0	7	9	22	-1	14
prev	-1	A	A	B	-1	A

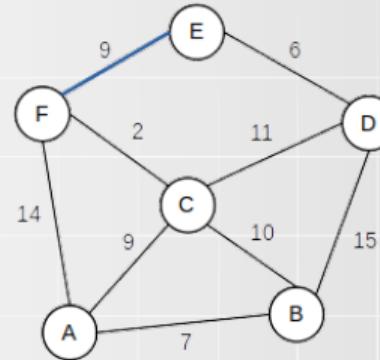
unvisited: { C, D , E , F }

current: C



# Algorytm Dijkstry (9)

Przykład szukania trasy od  $v_s = A$  do  $v_e = E$  (iteracja 4).



	A	B	C	D	E	F
dist	0	7	9	20	-1	11
prev	-1	A	A	C	-1	C

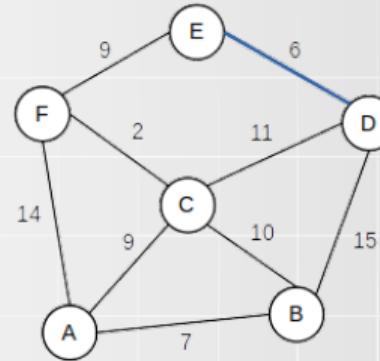
unvisited: { D , E , F }

current: F



# Algorytm Dijkstry (10)

Przykład szukania trasy od  $v_s = A$  do  $v_e = E$  (iteracja 5).



	A	B	C	D	E	F
dist	0	7	9	20	20	11
prev	-1	A	A	C	F	C

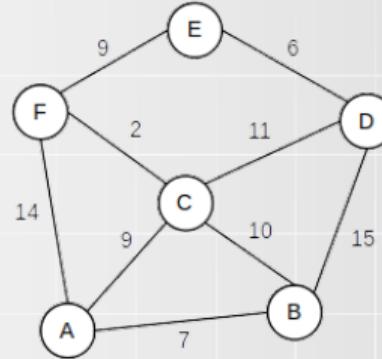
unvisited: { D , E }

current: E



# Algorytm Dijkstry (11)

Przykład szukania trasy od  $v_s = A$  do  $v_e = E$  (iteracja 6).



	A	B	C	D	E	F
dist	0	7	9	20	20	11
prev	-1	A	A	C	F	C

unvisited: { D }

koniec, trasa to (A, C, F, E)



## Algorytm Dijkstry (12)

- ▶ W kroku 3 musimy wybrać wierzchołek  $v$  o najmniejszym  $dist[v]$  spośród wszystkich wierzchołków w  $unvisited$ .
  - ▶ Potrzebujemy kolejki priorytetowej.
- ▶ Jeśli jako kolejkę użyjemy zwykłą listę i reprezentujemy graf macierzą sąsiedztwa, to pesymistyczny czas działania algorytmu wynosi  $O(n^2)$ .
- ▶ Jeśli jako kolejkę użyjemy kopca binarnego i reprezentujemy graf listą sąsiedztwa, to czas wynosi  $O((k + n) \log n)$ .
- ▶ Jeśli dodatkowo użyjemy kopca Fibonacciego to czas wyniesie  $O(k + n \log n)$ .



## Algorytm Dijkstry (13)

- ▶ Za każdym razem wybieramy lokalnie najlepszą decyzję – algorytm Dijkstry jest więc przykładem algorytmu zachłannego.
- ▶ „Rzadki” przykład algorytmu dla którego strategia zachłanna jest optymalna (otrzymana trasa jest zawsze najkrótsza).
- ▶ Algorytm Dijkstry może być też uznany za przykład programowania dynamicznego.
- ▶ „Rzadki” przykład programowania dynamicznego, które działa w czasie wielomianowym.



# Algorytm Bellmana-Forda (1)

- ▶ Jeśli graf ma ujemne wagi, to nie można użyć algorytmu Dijkstry.
- ▶ Można wtedy użyć wolniejszego, ale ogólniejszego algorytmu Bellmana-Forda.
  - ▶ Algorytm Bellmana-Forda dopuszcza obecność ujemnych wag, ale nie może być ujemnych cykli (tj. cykli o ujemnej długości) osiągalnych z  $v_s$ .
    - ▶ Po ujemnym cyklu można poruszać się bez końca uzyskując dowolnie krótką ścieżkę – najkrótsza ścieżka więc nie istnieje.
  - ▶ Algorytm rozwiązuje single-source shortest path problem.
  - ▶ Podobnie do algorytmu Dijkstry, algorytm Bellmana-Forda również polega na zasadzie relaksacji, ale nie na strategii zachłannej.



# Algorytm Bellmana-Forda (2)

Przebieg algorytmu:

1. Inicjalizacja tablicę  $dist$  oraz  $prev$  identyczna jak dla algorytmu Dijkstry.
2. Wykonujemy  $n - 1$  razy:
  - 2.1. Dla każdej krawędzi  $(u, v)$  próbujemy wykonać relaksację:
    - 2.1.1. Jeśli  $dist[u] + w(u, v) < dist[v]$  to
      - 2.1.1.1.  $dist[v] \leftarrow dist[u] + w(u, v)$ .
      - 2.1.1.2.  $prev[v] \leftarrow u$ .
  3. Wykonujemy jeszcze jedną iterację relaksacji.
    - ▶ Jeśli jakaś wartość w  $dist$  się zmniejszyła, to znaczy, że mamy ujemny cykl.



## Algorytm Bellmana-Forda (3)

- ▶ Czas działania to  $O(kn)$ .
  - ▶  $O(n^3)$  dla grafów gęstych.
  - ▶  $O(n^2)$  dla rzadkich.
- ▶ Proste usprawnienie – jeśli podczas iteracji nie dokonała się żadna relaksacja, to algorytm można zakończyć.
- ▶ Algorytm Bellmana-Forda można też wykorzystać w zadaniach, w których celem jest znalezienie ujemnego cyklu.



## Algorytm A\* (1)

- ▶ A\* rozwiązuje single-pair shortest path problem.
- ▶ Algorytm opiera się na obserwacji, że długość najkrótszej drogi od  $v_s$  do  $v_e$ , przez wierzchołek  $v$  jest równa:

$$sp(v_s, v) + sp(v, v_e), \quad (1)$$

gdzie  $sp$  jest najkrótszą drogą pomiędzy daną parą wierzchołków.

- ▶ Najlepszą obecnie znaną wartość  $sp(v_s, v)$  mamy dzięki  $dist(v)$ .
- ▶ Wartości  $sp(v, v_e)$  nie znamy... ale możemy ją oszacować za pomocą jakiejś heurystyki.



## Algorytm A\* (2)

- ▶ A\* korzysta więc z dodatkowej informacji o grafie i jako kolejny rozwija węzeł  $v$  o najmniejszej wartości funkcji  $f(v)$ :

$$f(v) = g(v) + h(v), \quad (2)$$

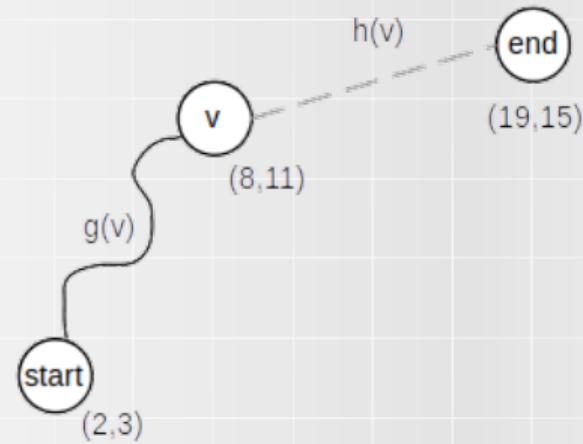
gdzie  $g(v)$  jest znaną długością drogi od  $v_s$  do  $v$ , zaś  $h(v)$  jest szacowaną długością drogi od  $v$  do  $v_e$ .

- ▶ Częstym przykładem są grafy w przestrzeni Euklidesowej, gdzie waga krawędzi  $(u, v)$  równa jest odległości Euklidesowej pomiędzy  $u$  i  $v$ .

# Algorytm A\* (3)

Przykład dla przestrzeni Euklidesowej:

$$h(v) = \sqrt{(19 - 8)^2 + (15 - 11)^2} = \sqrt{11^2 + 4^2} = \sqrt{137} \approx 11.7 \quad (3)$$





## Algorytm A\* (4)

- ▶ Jeśli heurystyka  $h(v)$  jest dopuszczalna (czyli nigdy nie przeszacowuje długości najlepszej drogi od  $v$  do  $v_e$ ), to A\* zawsze zwraca najkrótszą (optymalną) drogę od  $v_s$  do  $v_e$ , o ile taka istnieje.
- ▶ Dodatkowo jeśli  $h(v)$  jest spójna tj.:

$$h(x) \leq d(x, y) + h(y), \quad (4)$$

gdzie  $d(x, y)$  jest faktyczną odlegością między  $x$  oraz  $y$ , to nie istnieje algorytm, który dla heurystyki  $h()$  rozwija mniej węzłów niż A\* (stąd gwiazdka oznaczająca optymalność) i nigdy nie odwiedza danego węzła więcej niż raz.

- ▶ Im lepsza heurystyka (im bliżej wartości prawdziwej bez przeszacowywania), tym mniej węzłów rozwinie algorytm.



## Algorytm A\* (5)

- ▶ Sposób rozwiązywania remisów wpływa na skuteczność algorytmu (poleca się by remisy traktowane były w porządku stosowym).
- ▶ Specjalne przypadki:
  - ▶ Algorytm Dijkstry (gdy  $h(v) = 0$ ).
  - ▶ Przeszukiwanie wszerz.
  - ▶ Przeszukiwanie wgłąb.
- ▶ Jeśli  $h()$  nie jest dopuszczalna, to A\* może znaleźć drogę, która nie jest najkrótsza... ale może rozwinąć mniej węzłów.
  - ▶ Kontrolując przeszacowywanie heurystyki kontrolujemy dokładność i czas działania – przydatne w niektórych zastosowaniach (np. gry).

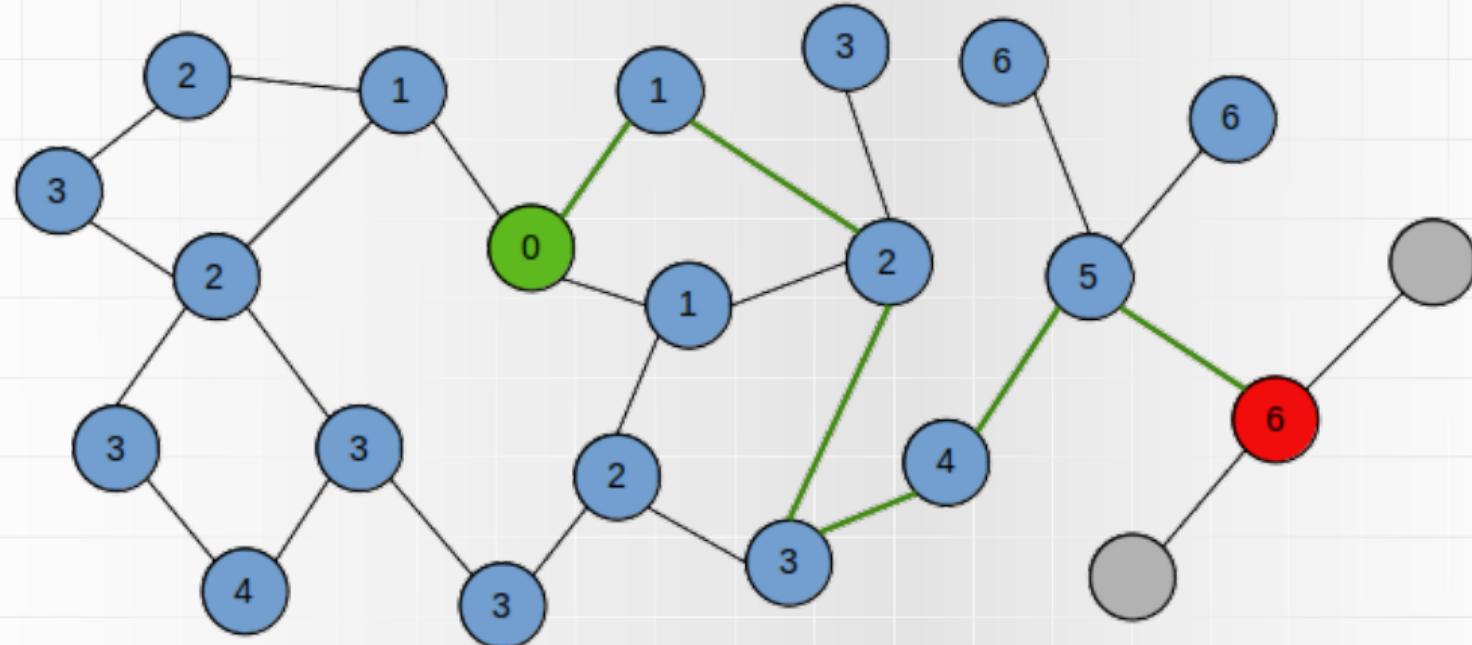


## Poszukiwanie wszerz (1)

- ▶ W przypadku grafu bez wag trasę od  $v_s$  do  $v_e$  można wydajnie znaleźć stosując zwykłe przeszukiwanie grafu wszerz.
- ▶ Zaczynamy od  $v_s$ .
- ▶ Sąsiedzi  $v_s$  mają odległość 1.
- ▶ (Nieodwiedzeni) sąsiedzi sąsiadów mają wartość 2 itd.
- ▶ Po dotarciu do  $v_e$  budujemy trasę cofając się do  $v_s$  poprzez zawsze wybieranie dowolnego wierzchołka o wartości o 1 mniejszej od naszej.
- ▶ Złożoność pesymistyczna  $O(n + k)$ .



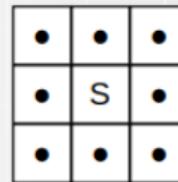
## Poszukiwanie wszerz (2)



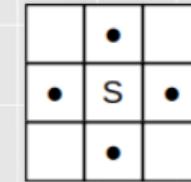
## Poszukiwanie wszerz (3)

- ▶ Szczególnie użyteczne dla grafów typu kratownica (np. pola w grze).
- ▶ Jeśli możemy iść w 8 kierunkach, to mamy do czynienia z sąsiedztwem Moore'a, zaś odległość od  $v_s$  do  $v_e$  odpowiada odległości Czebyszewa (szachowej).
- ▶ Jeśli możemy iść w 4 kardynalnych kierunkach to mamy do czynienia z sąsiedztwem von Neumanna, zaś odległość od  $v_s$  do  $v_e$  odpowiada odległości Manhattan (miejskiej).

Sąsiedztwo Moore'a



Sąsiedztwo von Neumanna





# Poszukiwanie wszerz (4)

Przykład dla sąsiedztwa von Neumanna:

2	1	2		13	14	15	16	17	18	19	20	21					
1	0	1		12	13	14	15	16	17	18	19	20		25			
2	1	2		11	12				18	19	20	21		25	24	25	
3	2	3		10	11				18	19	20			23	24	25	
4	3	4		9	10				16	17	18	19	20	21	22	23	24
5	4	5	6	7	8	9			15	16	17	18	19	20	21	22	23
6	5	6	7	8	9	10	11	12	13	14	15						
7	6	7		10	11	12	13	14	15	16		20	21	22			
8	7	8		11	12	13	14	15	16	17	18	19	20	21			
9	8	9		12	13	14	15	16	17	18							
10	9	10		13	14	15	16	17	18	19	20	21	22	23	24	25	



## Poszukiwanie wszerz (5)

Przykład dla zmodyfikowanego sąsiedztwa Moore'a (nie przechodzimy przez narożniki ścian):

1	1	1		12	12	12	13	14	15	16	17	18			
1	0	1		11	11	12	13	14	15	16	17	17		21	21
1	1	1		10	10				16	16	16	16		21	20
2	2	2		9	9				15	15	15			19	19
3	3	3		8	8				14	14	14	15	16	17	18
4	4	4	5	6	7	8			13	13	14	15	16	17	18
5	5	5	5	6	7	8	9	10	11	12	13				
6	6	6			8	8	9	10	11	12	13		16	16	17
7	7	7			9	9	9	10	11	12	13	14	15	16	17
8	8	8			10	10	10	10	11	12	13			20	21
9	9	9			11	11	11	11	11	12	13	14	15	16	17



# Algorytm Floyd-Warshall (1)

- ▶ Rozwiązuje all-pairs shortest path problem w grafie skierowanym.
  - ▶ Uzyskujemy macierz rozmiaru  $n \times n$ , której element  $(u, v)$  przechowuje długość najkrótszej drogi od  $u$  do  $v$ .
  - ▶ Zwraca tylko długości tras, a nie same trasy. Można jednak zmodyfikować go by znaleźć też trasy.
- ▶ Wagi grafu mogą być ujemne, ale nie może być ujemnego cyklu.
  - ▶ Podczas algorytmu należy sprawdzać okresowo przekątną macierzy, wartości ujemne są oznaką cykli ujemnych (gdyż najkrótsza droga od  $u$  do  $u$  powinna mieć zawsze długość 0).
- ▶ Algorytm działa w czasie  $O(n^3)$ .



# Algorytm Floyd-Warshall (2)

Przebieg algorytmu:

1. Tworzymy macierz  $dist$  rozmiaru  $n \times n$  z wszystkimi elementami ustawionymi na  $\infty$ .

2. Dla każdej krawędzi  $(u, v)$  wykonujemy:

$$dist[u][v] \leftarrow w(u, v).$$

3. Dla  $v$  od 0 do  $n - 1$  wykonujemy:

1. Dla  $i$  od 0 do  $n - 1$  wykonujemy:

1. Dla  $j$  od 0 do  $n - 1$  wykonujemy:

1. Jeżeli  $dist[i][j] > dist[i][v] + dist[v][j]$ :

1.  $dist[i][j] \leftarrow dist[i][v] + dist[v][j].$



## Algorytm Johnsona (1)

- ▶ Również rozwiązuje all-pairs shortest path problem w grafie skierowanym.
- ▶ Algorytm korzysta z algorytmu Bellmana-Forda i algorytmu Dijkstry.
- ▶ Złożoność to  $O(n^2 \log n + nk)$ .
  - ▶  $O(nk)$  z wywołania algorytmu Bellmana-Forda, zaś  $O(n^2 \log n)$  z  $n$ -krotnego wywołania algorytmu Dijkstry.
  - ▶ Dla grafów rzadkich sprowadza się to do  $O(n^2 \log n)$  – szybciej niż algorytm Floyda-Warshalla.



## Algorytm Johnsona (2)

1. Dodajemy dodatkowy węzeł  $q$  i łączymy z wszystkimi  $n$  wierzchołkami krawędziami o wagę 0.
2. Wywołujemy algorytm Bellmana-Forda na wierzchołku  $q$ .
  - ▶ Znajdujemy dla każdego wierzchołka  $v$  minimalną wagę  $h(v)$  na trasie z  $q$  do  $v$ .
  - ▶ Wykrycie ujemnego cyklu kończy cały algorytm.
3. Zmiana wag grafu:  $w(u, v) \leftarrow w(u, v) + h(u) - h(v)$ .
4. Usuwamy  $q$  i wykonujemy algorytm Dijkstry  $n$  razy (dla każdego wierzchołka początkowego osobno).
5. Odległość z  $u$  do  $v$  w oryginalnym grafie to odległość zwrócona przez algorytm Dijkstry plus  $h(v) - h(u)$ .



# Podsumowanie

Algorytm	Problem	Graf	Czas ogólny	Czas gęsty	Czas rzadki
Poszukiwanie wszerz	single-source	bez wag (identyczne wagi)	$O(n + k)$	$O(n^2)$	$O(n)$
Dijkstra	single-source	bez ujemnych wag	$O(k + n \log n)$	$O(n^2)$	$O(n \log n)$
A*	single-pair	bez ujemnych wag	zależny od heurystyki		
Bellman-Ford	single-source	bez ujemnych cykli	$O(kn)$	$O(n^3)$	$O(n^2)$
Floyd-Warshall	all-pairs	bez ujemnych cykli	$O(n^3)$	$O(n^3)$	$O(n^3)$
Johnson	all-pairs	bez ujemnych cykli	$O(n^2 \log n + nk)$	$O(n^3)$	$O(n^2 \log n)$



Wrocław  
University  
of Science  
and Technology

# Struktury danych

Wykład 10

## Problem najkrótszej drogi

dr inż. Jarosław Rudy





# Problem najkrótszej drogi (1)

- ▶ W najprostszej wersji dany jest graf  $G = (V, E)$  oraz dwa wierzchołki: startowy  $v_s$  oraz końcowy  $v_e$ .
- ▶ Celem jest znalezienie (prostej) drogi od  $v_s$  do  $v_e$  o najmniejszej długości.
- ▶ Graf może być skierowany lub nie. Dla skierowanego istotne jest by poruszać się krawędziami w odpowiednim kierunku.
- ▶ Długość drogi liczona jest wagami krawędzi (ewentualnie krawędzi i wierzchołków).
  - ▶ Dla grafów bez wag długością drogi jest liczba jej krawędzi.
  - ▶ Droga od  $v_s$  do  $v_e$  może nie istnieć.



# Problem najkrótszej drogi (2)

Różne warianty problemu:

- ▶ Jedna para wierzchołków (single-pair shortest path problem) – znalezienie najkrótszej drogi od  $v_s$  do  $v_e$ .
  - ▶ Pesymistycznie i tak trzeba znaleźć drogę do innych wierzchołków.
- ▶ Tylko wierzchołek początkowy (single-source shortest path problem) – znalezienie najkrótszej drogi od  $v_s$  do każdego innego wierzchołka.
- ▶ Tylko wierzchołek końcowy (single-destination shortest path problem) – znalezienie najkrótszej drogi do  $v_e$  z każdego innego wierzchołka.
  - ▶ Może być sprowadzony do poprzedniego problemu (odwrócenie łuków).
- ▶ Wszystkie pary (all-pairs shortest path problem) – znalezienie najkrótszej drogi od każdego wierzchołka do każdego innego.



# Algorytm Dijkstry (1)

- ▶ Stworzony przez Edsgera Dijkstrę w 1956.
- ▶ Oryginalnie rozwiązywał single-pair shortest path problem.
- ▶ Obecnie jednak częsty wariant rozwiązuje single-source shortest path problem.
- ▶ Algorytm działa dla dowolnych skierowanych/nieskierowanych grafów ważowych bez ujemnych wag.
- ▶ Przy odpowiedniej implementacji i braku dodatkowych założeń (graf typu DAG, specyficzne wagi) jest to najlepszy znany algorytm dla tego problemu.



## Algorytm Dijkstry (2)

- ▶ Algorytm działa na zasadzie metody relaksacji – dla niektórych wierzchołków sprawdzamy czy nie da się zbudować do nich trasy krótszej niż najlepsza obecnie znana.
- ▶ Dla każdego wierzchołka  $v \in V$  przechowujemy (np. jako tablice) dwie informacje :
  - ▶  $dist[v]$  – długość najlepszej obecnie znanej drogi z  $v_s$  do  $v$  (długość tymczasowa).
  - ▶  $prev[v]$  – wierzchołek poprzedzający  $v$  na najlepszej obecnie znanej trasie od  $v_s$  do  $v$ .
- ▶ Potrzebujemy też przechowywać zbiór wierzchołków nieodwiedzonych *unvisited* oraz wierzchołek aktualny *current*.



## Algorytm Dijkstry (3)

Na początku ustawiamy:

- ▶  $unvisited \leftarrow V$ .
- ▶  $current \leftarrow v_s$ .
- ▶ Dla każdego  $v \in V$ :
  - ▶  $prev[v] \leftarrow undefined$  (np. -1)
  - ▶  $dist[v_s] \leftarrow 0$  (trasa od  $v_s$  do  $v_s$  ma znaną długość).
  - ▶ Dla każdego  $v \in V \setminus \{v_s\}$ :
    - ▶  $dist[v] \leftarrow \infty$  (np. -1)



# Algorytm Dijkstry (4)

Przebieg algorytmu:

1. Dla wierzchołka *current* sprawdzamy jego nieodwiedzonych sąsiadów i aktualizujemy ich tymczasową odległość, jeśli można do nich szybciej dotrzeć z *current*. Założymy sąsiada *neigh*:
  - ▶ Jeśli  $dist[current] + w(current, neigh) < dist[neigh]$  to:
    - ▶  $dist[neigh] \leftarrow dist[current] + w(current, neigh)$ .
    - ▶  $prev[neigh] \leftarrow current$ .
2. Usuwamy *current* z *unvisited*.
3. Jako *current* wybieramy wierzchołek *v* z *unvisited* o najmniejszym  $dist[v]$ .
4. Wracamy do punktu 1.



## Algorytm Dijkstry (5)

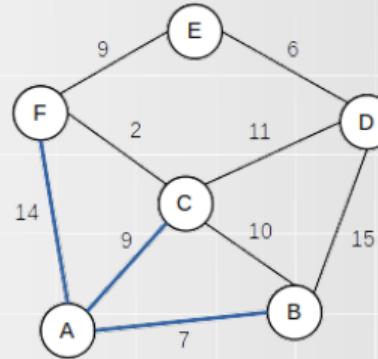
Warunek stopu algorytmu:

- ▶ Jeśli szukamy trasy od  $v_s$  do  $v_e$  algorytm kończy się, gdy:
  - ▶ W kroku 2 wierzchołek  $v_e$  został odwiedzony – możemy zbudować trasę na podstawie *prev*.
  - ▶ W kroku 3 za *current* ustawiono  $\infty$  – wierzchołek  $v_e$  jest nieosiągalny z wierzchołka  $v_s$ .
- ▶ Jeśli szukamy trasy od  $v_s$  do każdego innego wierzchołka to algorytm kończy się gdy w kroku 3 za *current* ustawiono  $\infty$ :
  - ▶ Oznacza to, że do wierzchołków obecnie w *unvisited* nie można dotrzeć z  $v_s$ . Trasy do pozostałych możemy zbudować w oparciu o *prev*.



## Algorytm Dijkstry (6)

Przykład szukania trasy od  $v_s = A$  do  $v_e = E$  (iteracja 1).



	A	B	C	D	E	F
dist	0	-1	-1	-1	-1	-1
prev	-1	-1	-1	-1	-1	-1

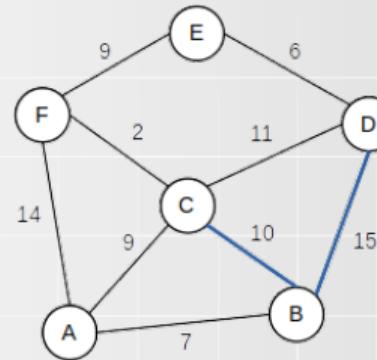
unvisited: { A , B , C , D , E , F }

current: A



## Algorytm Dijkstry (7)

Przykład szukania trasy od  $v_s = A$  do  $v_e = E$  (iteracja 2).



	A	B	C	D	E	F
dist	0	7	9	-1	-1	14
prev	-1	A	A	-1	-1	A

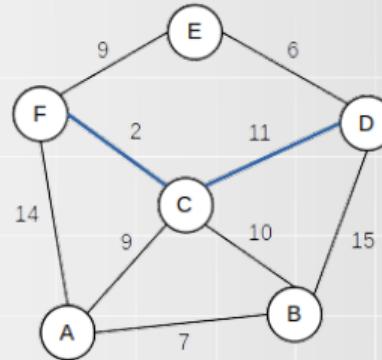
unvisited: { B , C , D , E , F }

current: B



## Algorytm Dijkstry (8)

Przykład szukania trasy od  $v_s = A$  do  $v_e = E$  (iteracja 3).



	A	B	C	D	E	F
dist	0	7	9	22	-1	14
prev	-1	A	A	B	-1	A

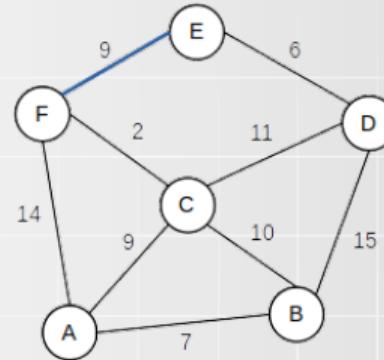
unvisited: { C, D , E , F }

current: C



# Algorytm Dijkstry (9)

Przykład szukania trasy od  $v_s = A$  do  $v_e = E$  (iteracja 4).



	A	B	C	D	E	F
dist	0	7	9	20	-1	11
prev	-1	A	A	C	-1	C

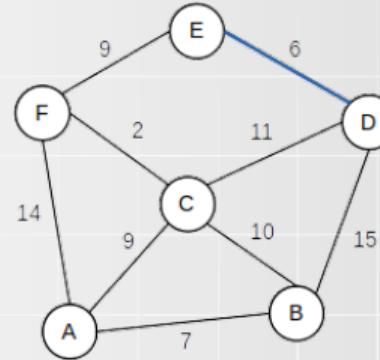
unvisited: { D , E , F }

current: F



# Algorytm Dijkstry (10)

Przykład szukania trasy od  $v_s = A$  do  $v_e = E$  (iteracja 5).



	A	B	C	D	E	F
dist	0	7	9	20	20	11
prev	-1	A	A	C	F	C

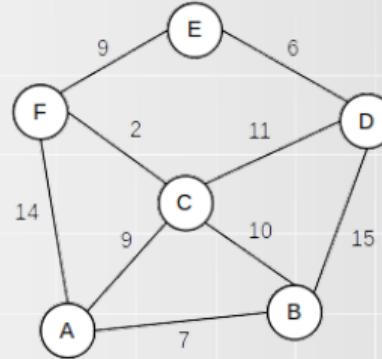
unvisited: { D , E }

current: E



# Algorytm Dijkstry (11)

Przykład szukania trasy od  $v_s = A$  do  $v_e = E$  (iteracja 6).



	A	B	C	D	E	F
dist	0	7	9	20	20	11
prev	-1	A	A	C	F	C

unvisited: { D }

koniec, trasa to (A, C, F, E)



## Algorytm Dijkstry (12)

- ▶ W kroku 3 musimy wybrać wierzchołek  $v$  o najmniejszym  $dist[v]$  spośród wszystkich wierzchołków w  $unvisited$ .
  - ▶ Potrzebujemy kolejki priorytetowej.
- ▶ Jeśli jako kolejkę użyjemy zwykłą listę i reprezentujemy graf macierzą sąsiedztwa, to pesymistyczny czas działania algorytmu wynosi  $O(n^2)$ .
- ▶ Jeśli jako kolejkę użyjemy kopca binarnego i reprezentujemy graf listą sąsiedztwa, to czas wynosi  $O((k + n) \log n)$ .
- ▶ Jeśli dodatkowo użyjemy kopca Fibonacciego to czas wyniesie  $O(k + n \log n)$ .



## Algorytm Dijkstry (13)

- ▶ Za każdym razem wybieramy lokalnie najlepszą decyzję – algorytm Dijkstry jest więc przykładem algorytmu zachłanego.
- ▶ „Rzadki” przykład algorytmu dla którego strategia zachłanna jest optymalna (otrzymana trasa jest zawsze najkrótsza).
- ▶ Algorytm Dijkstry może być też uznany za przykład programowania dynamicznego.
- ▶ „Rzadki” przykład programowania dynamicznego, które działa w czasie wielomianowym.



## Algorytm Bellmana-Forda (1)

- ▶ Jeśli graf ma ujemne wagi, to nie można użyć algorytmu Dijkstry.
- ▶ Można wtedy użyć wolniejszego, ale ogólniejszego algorytmu Bellmana-Forda.
  - ▶ Algorytm Bellmana-Forda dopuszcza obecność ujemnych wag, ale nie może być ujemnych cykli (tj. cykli o ujemnej długości) osiągalnych z  $v_s$ .
    - ▶ Po ujemnym cyklu można poruszać się bez końca uzyskując dowolnie krótką ścieżkę – najkrótsza ścieżka więc nie istnieje.
  - ▶ Algorytm rozwiązuje single-source shortest path problem.
  - ▶ Podobnie do algorytmu Dijkstry, algorytm Bellmana-Forda również polega na zasadzie relaksacji, ale nie na strategii zachłannej.



# Algorytm Bellmana-Forda (2)

Przebieg algorytmu:

1. Inicjalizacja tablicę  $dist$  oraz  $prev$  identyczna jak dla algorytmu Dijkstry.
2. Wykonujemy  $n - 1$  razy:
  - 2.1. Dla każdej krawędzi  $(u, v)$  próbujemy wykonać relaksację:
    - 2.1.1. Jeśli  $dist[u] + w(u, v) < dist[v]$  to
      - 2.1.1.1.  $dist[v] \leftarrow dist[u] + w(u, v)$ .
      - 2.1.1.2.  $prev[v] \leftarrow u$ .
  3. Wykonujemy jeszcze jedną iterację relaksacji.
    - ▶ Jeśli jakaś wartość w  $dist$  się zmniejszyła, to znaczy, że mamy ujemny cykl.



## Algorytm Bellmana-Forda (3)

- ▶ Czas działania to  $O(kn)$ .
  - ▶  $O(n^3)$  dla grafów gęstych.
  - ▶  $O(n^2)$  dla rzadkich.
- ▶ Proste usprawnienie – jeśli podczas iteracji nie dokonała się żadna relaksacja, to algorytm można zakończyć.
- ▶ Algorytm Bellmana-Forda można też wykorzystać w zadaniach, w których celem jest znalezienie ujemnego cyklu.



## Algorytm A\* (1)

- ▶ A\* rozwiązuje single-pair shortest path problem.
- ▶ Algorytm opiera się na obserwacji, że długość najkrótszej drogi od  $v_s$  do  $v_e$ , przez wierzchołek  $v$  jest równa:

$$sp(v_s, v) + sp(v, v_e), \quad (1)$$

gdzie  $sp$  jest najkrótszą drogą pomiędzy daną parą wierzchołków.

- ▶ Najlepszą obecnie znaną wartość  $sp(v_s, v)$  mamy dzięki  $dist(v)$ .
- ▶ Wartości  $sp(v, v_e)$  nie znamy... ale możemy ją oszacować za pomocą jakiejś heurystyki.



## Algorytm A\* (2)

- ▶ A\* korzysta więc z dodatkowej informacji o grafie i jako kolejny rozwija węzeł  $v$  o najmniejszej wartości funkcji  $f(v)$ :

$$f(v) = g(v) + h(v), \quad (2)$$

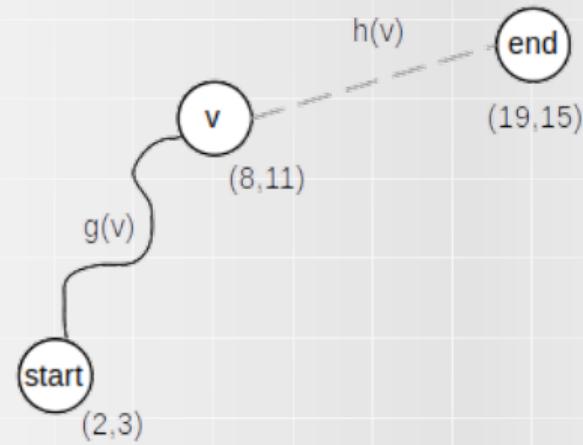
gdzie  $g(v)$  jest znaną długością drogi od  $v_s$  do  $v$ , zaś  $h(v)$  jest szacowaną długością drogi od  $v$  do  $v_e$ .

- ▶ Częstym przykładem są grafy w przestrzeni Euklidesowej, gdzie waga krawędzi  $(u, v)$  równa jest odległości Euklidesowej pomiędzy  $u$  i  $v$ .

# Algorytm A\* (3)

Przykład dla przestrzeni Euklidesowej:

$$h(v) = \sqrt{(19 - 8)^2 + (15 - 11)^2} = \sqrt{11^2 + 4^2} = \sqrt{137} \approx 11.7 \quad (3)$$





## Algorytm A\* (4)

- ▶ Jeśli heurystyka  $h(v)$  jest dopuszczalna (czyli nigdy nie przeszacowuje długości najlepszej drogi od  $v$  do  $v_e$ ), to A\* zawsze zwraca najkrótszą (optymalną) drogę od  $v_s$  do  $v_e$ , o ile taka istnieje.
- ▶ Dodatkowo jeśli  $h(v)$  jest spójna tj.:

$$h(x) \leq d(x, y) + h(y), \quad (4)$$

gdzie  $d(x, y)$  jest faktyczną odlegością między  $x$  oraz  $y$ , to nie istnieje algorytm, który dla heurystyki  $h()$  rozwija mniej węzłów niż A\* (stąd gwiazdka oznaczająca optymalność) i nigdy nie odwiedza danego węzła więcej niż raz.

- ▶ Im lepsza heurystyka (im bliżej wartości prawdziwej bez przeszacowywania), tym mniej węzłów rozwinie algorytm.



## Algorytm A\* (5)

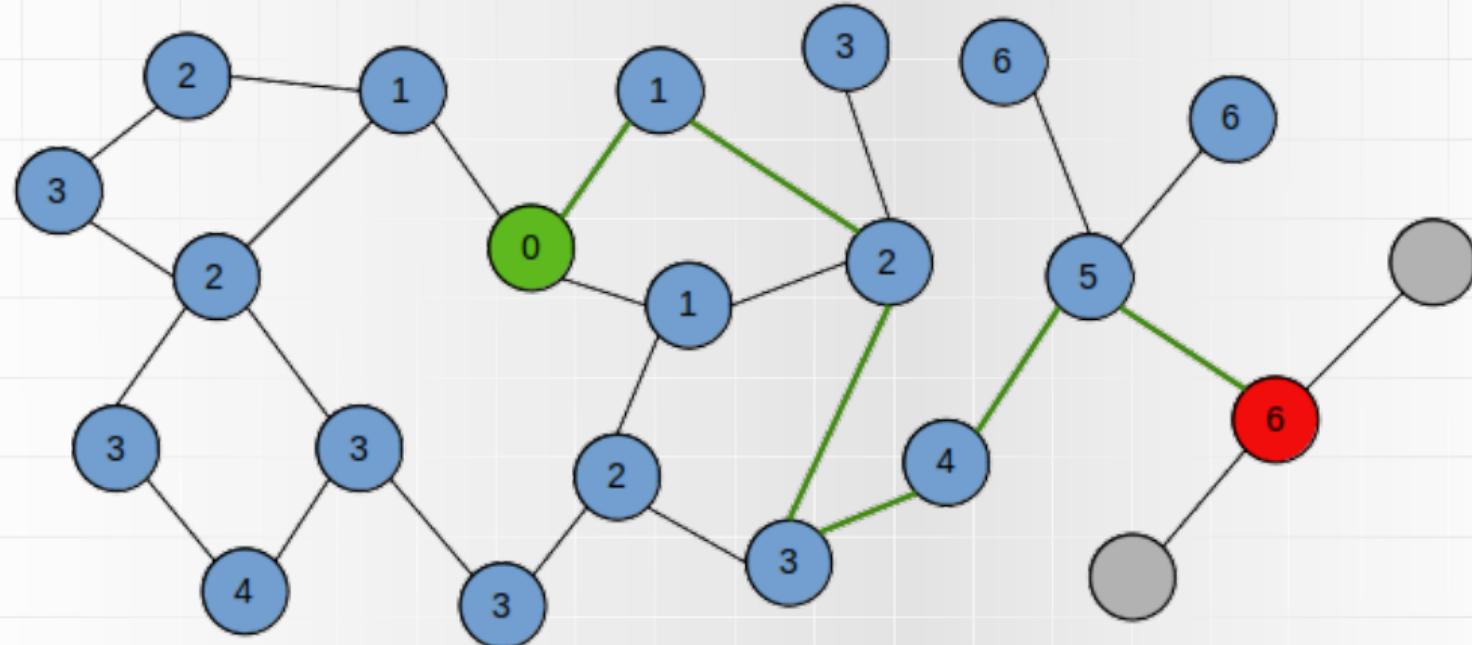
- ▶ Sposób rozwiązywania remisów wpływa na skuteczność algorytmu (poleca się by remisy traktowane były w porządku stosowym).
- ▶ Specjalne przypadki:
  - ▶ Algorytm Dijkstry (gdy  $h(v) = 0$ ).
  - ▶ Przeszukiwanie wszerz.
  - ▶ Przeszukiwanie wgłąb.
- ▶ Jeśli  $h()$  nie jest dopuszczalna, to A\* może znaleźć drogę, która nie jest najkrótsza... ale może rozwinąć mniej węzłów.
  - ▶ Kontrolując przeszacowywanie heurystyki kontrolujemy dokładność i czas działania – przydatne w niektórych zastosowaniach (np. gry).



## Poszukiwanie wszerz (1)

- ▶ W przypadku grafu bez wag trasę od  $v_s$  do  $v_e$  można wydajnie znaleźć stosując zwykłe przeszukiwanie grafu wszerz.
- ▶ Zaczynamy od  $v_s$ .
- ▶ Sąsiedzi  $v_s$  mają odległość 1.
- ▶ (Nieodwiedzeni) sąsiedzi sąsiadów mają wartość 2 itd.
- ▶ Po dotarciu do  $v_e$  budujemy trasę cofając się do  $v_s$  poprzez zawsze wybieranie dowolnego wierzchołka o wartości o 1 mniejszej od naszej.
- ▶ Złożoność pesymistyczna  $O(n + k)$ .

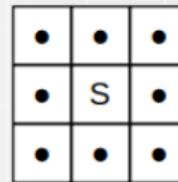
# Poszukiwanie wszerz (2)



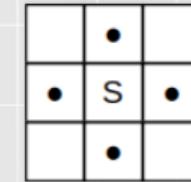
## Poszukiwanie wszerz (3)

- ▶ Szczególnie użyteczne dla grafów typu kratownica (np. pola w grze).
- ▶ Jeśli możemy iść w 8 kierunkach, to mamy do czynienia z sąsiedztwem Moore'a, zaś odległość od  $v_s$  do  $v_e$  odpowiada odległości Czebyszewa (szachowej).
- ▶ Jeśli możemy iść w 4 kardynalnych kierunkach to mamy do czynienia z sąsiedztwem von Neumanna, zaś odległość od  $v_s$  do  $v_e$  odpowiada odległości Manhattan (miejskiej).

Sąsiedztwo Moore'a



Sąsiedztwo von Neumanna





# Poszukiwanie wszerz (4)

Przykład dla sąsiedztwa von Neumanna:

2	1	2		13	14	15	16	17	18	19	20	21					
1	0	1		12	13	14	15	16	17	18	19	20		25			
2	1	2		11	12				18	19	20	21		25	24	25	
3	2	3		10	11				18	19	20			23	24	25	
4	3	4		9	10				16	17	18	19	20	21	22	23	24
5	4	5	6	7	8	9			15	16	17	18	19	20	21	22	23
6	5	6	7	8	9	10	11	12	13	14	15						
7	6	7		10	11	12	13	14	15	16		20	21	22			
8	7	8		11	12	13	14	15	16	17	18	19	20	21			
9	8	9		12	13	14	15	16	17	18							
10	9	10		13	14	15	16	17	18	19	20	21	22	23	24	25	



## Poszukiwanie wszerz (5)

Przykład dla zmodyfikowanego sąsiedztwa Moore'a (nie przechodzimy przez narożniki ścian):

1	1	1		12	12	12	13	14	15	16	17	18			
1	0	1		11	11	12	13	14	15	16	17	17		21	21
1	1	1		10	10				16	16	16	16		21	20
2	2	2		9	9				15	15	15			19	19
3	3	3		8	8				14	14	14	15	16	17	18
4	4	4	5	6	7	8			13	13	14	15	16	17	18
5	5	5	5	6	7	8	9	10	11	12	13				
6	6	6			8	8	9	10	11	12	13		16	16	17
7	7	7			9	9	9	10	11	12	13	14	15	16	17
8	8	8			10	10	10	10	11	12	13				20
9	9	9			11	11	11	11	11	12	13	14	15	16	17
															18
															19
															20



# Algorytm Floyd-Warshall (1)

- ▶ Rozwiązuje all-pairs shortest path problem w grafie skierowanym.
  - ▶ Uzyskujemy macierz rozmiaru  $n \times n$ , której element  $(u, v)$  przechowuje długość najkrótszej drogi od  $u$  do  $v$ .
  - ▶ Zwraca tylko długości tras, a nie same trasy. Można jednak zmodyfikować go by znaleźć też trasy.
- ▶ Wagi grafu mogą być ujemne, ale nie może być ujemnego cyklu.
  - ▶ Podczas algorytmu należy sprawdzać okresowo przekątną macierzy, wartości ujemne są oznaką cykli ujemnych (gdyż najkrótsza droga od  $u$  do  $u$  powinna mieć zawsze długość 0).
- ▶ Algorytm działa w czasie  $O(n^3)$ .



# Algorytm Floyd-Warshall (2)

Przebieg algorytmu:

1. Tworzymy macierz  $dist$  rozmiaru  $n \times n$  z wszystkimi elementami ustawionymi na  $\infty$ .

2. Dla każdej krawędzi  $(u, v)$  wykonujemy:

$$dist[u][v] \leftarrow w(u, v).$$

3. Dla  $v$  od 0 do  $n - 1$  wykonujemy:

1. Dla  $i$  od 0 do  $n - 1$  wykonujemy:

1. Dla  $j$  od 0 do  $n - 1$  wykonujemy:

1. Jeżeli  $dist[i][j] > dist[i][v] + dist[v][j]$ :

1.  $dist[i][j] \leftarrow dist[i][v] + dist[v][j].$



## Algorytm Johnsona (1)

- ▶ Również rozwiązuje all-pairs shortest path problem w grafie skierowanym.
- ▶ Algorytm korzysta z algorytmu Bellmana-Forda i algorytmu Dijkstry.
- ▶ Złożoność to  $O(n^2 \log n + nk)$ .
  - ▶  $O(nk)$  z wywołania algorytmu Bellmana-Forda, zaś  $O(n^2 \log n)$  z  $n$ -krotnego wywołania algorytmu Dijkstry.
  - ▶ Dla grafów rzadkich sprowadza się to do  $O(n^2 \log n)$  – szybciej niż algorytm Floyda-Warshalla.



## Algorytm Johnsona (2)

1. Dodajemy dodatkowy węzeł  $q$  i łączymy z wszystkimi  $n$  wierzchołkami krawędziami o wagę 0.
2. Wywołujemy algorytm Bellmana-Forda na wierzchołku  $q$ .
  - ▶ Znajdujemy dla każdego wierzchołka  $v$  minimalną wagę  $h(v)$  na trasie z  $q$  do  $v$ .
  - ▶ Wykrycie ujemnego cyklu kończy cały algorytm.
3. Zmiana wag grafu:  $w(u, v) \leftarrow w(u, v) + h(u) - h(v)$ .
4. Usuwamy  $q$  i wykonujemy algorytm Dijkstry  $n$  razy (dla każdego wierzchołka początkowego osobno).
5. Odległość z  $u$  do  $v$  w oryginalnym grafie to odległość zwrócona przez algorytm Dijkstry plus  $h(v) - h(u)$ .



# Podsumowanie

Algorytm	Problem	Graf	Czas ogólny	Czas gęsty	Czas rzadki
Poszukiwanie wszerz	single-source	bez wag (identyczne wagi)	$O(n + k)$	$O(n^2)$	$O(n)$
Dijkstra	single-source	bez ujemnych wag	$O(k + n \log n)$	$O(n^2)$	$O(n \log n)$
A*	single-pair	bez ujemnych wag	zależny od heurystyki		
Bellman-Ford	single-source	bez ujemnych cykli	$O(kn)$	$O(n^3)$	$O(n^2)$
Floyd-Warshall	all-pairs	bez ujemnych cykli	$O(n^3)$	$O(n^3)$	$O(n^3)$
Johnson	all-pairs	bez ujemnych cykli	$O(n^2 \log n + nk)$	$O(n^3)$	$O(n^2 \log n)$



Wrocław  
University  
of Science  
and Technology

# Struktury danych

Wykład 11  
Problem najdłuższej drogi

dr inż. Jarosław Rudy





# Problem najdłuższej drogi (1)

- ▶ Problem najdłuższej drogi może oznaczać kilka różnych rzeczy.
- ▶ Pierwszą definicją jest po prostu odwrócenie problemu najkrótszej drogi.
- ▶ Analogicznie otrzymujemy odwrócone warianty:
  - ▶ Single-pair longest path problem.
  - ▶ Single-source longest path problem.
  - ▶ Single-destination longest path problem.
  - ▶ All-pairs longest path problem.



## Problem najdłuższej drogi (2)

Tak rozumiany problem najdłuższej drogi w grafie  $G$  rozwiązuje się następująco:

- ▶ Niech —  $G$  będzie grafem  $G$  z zanegowanymi wagami krawędzi (ewentualnie też wierzchołków).
- ▶ Rozwiązuje się problem najkrótszej ścieżki w grafie —  $G$  za pomocą odpowiedniego algorytmu.
- ▶ Otrzymana ścieżka (o ile istnieje) jest żądaną najdłuższą ścieżką w  $G$ .
  - ▶ Jeśli interesuje nas tylko długość ścieżki, to należy ją zamienić na przeciwną.

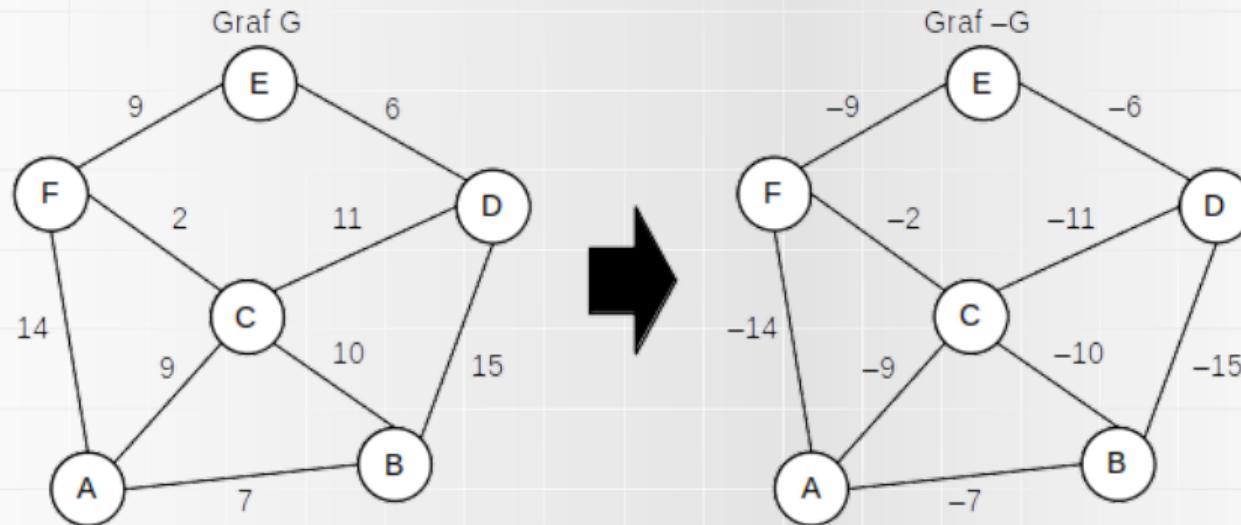


## Problem najdłuższej drogi (3)

- ▶ Ponieważ problemy są analogiczne, to występują te same trudności i ograniczenia jak dla problemu najkrótszej ścieżki:
  - ▶ Dla algorytmu Dijkstry nie może być krawędzi dodatnich w grafie  $G$  (ponieważ będą ujemne krawędzie w grafie  $-G$ ).
  - ▶ Dla algorytmu Bellmana-Forda nie może być cykli o dodatniej długości w  $G$  osiągalnych ze źródła (ponieważ będą cykle o ujemnej długości osiągalne ze źródła w  $-G$ ).
    - ▶ Jeśli są, to najdłuższa ścieżka (w tym sensie) nie istnieje w  $G$ .
- ▶ Podejście wymaga dodatkowego czasu  $O(n + k)$  (na konstrukcję grafu  $-G$ ), ale nie zmienia to złożoności algorytmów.

## Problem najdłuższej drogi (4)

Dla wielu grafów tak rozumiana najdłuższa droga nie istnieje, a pokazana transformacja jest bezużyteczna





## Problem najdłuższej drogi (5)

- ▶ Przy takiej definicji problemu i grafu („zwykły” graf o maksymalnie  $n(n - 1)$  krawędziach) problemy najkrótszej i najdłuższej ścieżki są wielomianowe (należą do  $\mathcal{P}$ ).
- ▶ Jest tak gdyż istnieje algorytm znajdujący ścieżkę (lub stwierdzający jej brak) w czasie wielomianowym.
  - ▶ Jest nim algorytm Bellmana-Forda działający w czasie  $O(nk)$ .
- ▶ O ile jednak tak zdefiniowany problem najkrótszej ścieżki ma sens (szukanie najtańszej trasy), to problem najdłuższej ścieżki w tym rozumieniu jest mniej praktyczny.
  - ▶ Po co szukać najdroższej trasy pomiędzy punktami?



## Problem najdłuższej drogi (6)

- ▶ W praktyce przez problem najdłuższej drogi (longest path problem, LPP) zwykle rozumie się inny, następujący problem:
  - ▶ Dany jest graf  $G$ . Celem jest znalezienie najdłuższej drogi prostej w całym grafie.
    - ▶ Druga musi być prosta tzn. nie mogą powtarzać się wierzchołki (a tym samym krawędzie).
    - ▶ Znaleziona droga będzie zaczynała się w pewnym wierzchołku  $v_s$  i kończyła w pewnym wierzchołku  $v_e$ , ale wierzchołki te nie są podawane na wejściu – algorytm sam musi je wskazać.



## Problem najdłuższej drogi (7)

- ▶ Wg takiej definicji praktycznie każdy graf ma najdłuższą drogę.
- ▶ Złożoność LPP jest jednak zupełnie inna niż w poprzedniej definicji – problem jest  $\mathcal{NP}$ -trudny.
- ▶ Dowód opiera się związku z problemem ścieżki Hamiltonowskiej.
- ▶ Nieznane są algorytmy rozwiązujące LPP w czasie wielomianowym dla dowolnego grafu (i nie będzie takich, chyba że  $\mathcal{P} = \mathcal{NP}$ ).
- ▶ Można jednak znaleźć wydajne algorytmy w niektórych szczególnych przypadkach.



## Problem najdłuższej drogi (8)

- ▶ Jeśli graf ma nie ma dodatkowych cykli, to można rozwiązać LPP w czasie  $O(n^3)$  w następujący sposób:
  - ▶ Standardowo konstruujemy graf  $-G$  (nie będzie w nim ujemnych cykli).
  - ▶ Wykonujemy algorytm Floyda-Warshalla na  $-G$ .
  - ▶ Wybieramy najkrótszą ze znalezionych tras.
- ▶ Jeśli potrzebne są same ścieżki, można zmodyfikować algorytm Floyda-Warshalla lub wykonać algorytm Bellmana-Forda na znalezionym wierzchołku początkowym.



# Problem najdłuższej drogi (9)



Floyd-Warshall

	A	B	C	D
A	0	9	15	19
B	$-\infty$	0	6	-10
C	$-\infty$	-8	0	4
D	$-\infty$	-12	-6	0

G

	A	B	C	D
A	0	-9	-15	-19
B	$\infty$	0	-6	10
C	$\infty$	8	0	-4
D	$\infty$	12	6	0



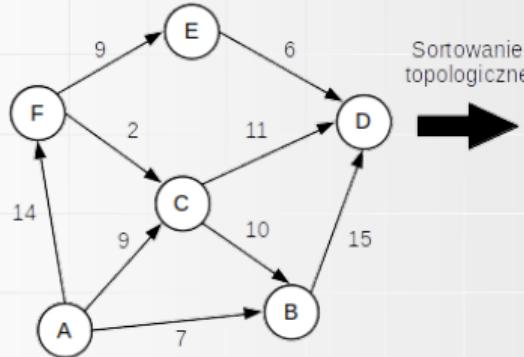
## Problem najdłuższej drogi (10)

Dla DAG można rozwiązać LPP w czasie  $O(n + k)$  tj. liniowym. Algorytm:

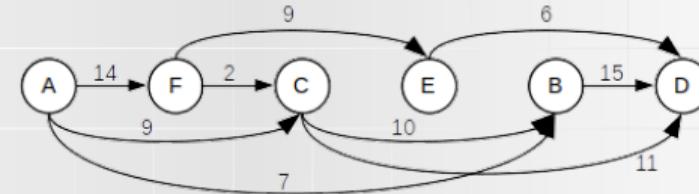
1. Znajdujemy porządek topologiczny (topological sorting).
2. Tworzymy listę  $\text{dist}$  i wpisujemy  $\text{dist}[v] \leftarrow -\infty$  dla każdego węzła  $v$ .
3. Dla kolejnych wierzchołków  $v$  w porządku topologicznym:
  - 3.1  $\text{dist}[v]$  wynosi największy  $\text{dist}$  poprzedników  $v$  plus krawędź łącząca poprzednika z  $v$ .
    - ▶ Jeśli  $v$  nie ma poprzedników to  $\text{dist}[v] \leftarrow 0$
4. Zaczynając od wierzchołka z największym  $\text{dist}[v]$  cofamy się (zawsze po poprzedniku z największym  $\text{dist}[v]$ ), aż do węzła bez poprzedników.
5. Wierzchołki odwiedzone podczas cofania (w odwrotnej kolejności) to szukana ścieżka.



# Problem najdłuższej drogi (11)



Sortowanie topologiczne



Najdłuższa ścieżka: A, F, C, B, D



A	B	C	D	E	F
0	-∞	-∞	-∞	-∞	-∞
0	-∞	-∞	-∞	-∞	14
0	-∞	16	-∞	-∞	14
0	-∞	16	-∞	23	14
0	26	16	-∞	23	14
0	26	16	41	23	14



## Problem najdłuższej drogi (12)

Algorytm Kahna wyznaczania kolejności topologicznej w  $O(n + k)$ :

1. Stwórz listę  $S$  węzłów bez poprzedników i pustą listę  $L$ .
2. Dopóki  $S$  nie jest puste:
  - 2.1 Usuń pewien węzeł  $u$  z  $S$  i dodaj go do  $L$ .
  - 2.2 Dla każdego węzła  $v$  będącego sąsiadem  $u$ :
    - 2.2.1 Usuń krawędź  $(u, v)$  z grafu.
    - 2.2.2 Jeśli  $v$  nie ma innych krawędzi wchodzących, to dodaj  $v$  do  $S$ .
3. Jeśli w grafie zostały krawędzie, to błąd (jest cykl, a graf nie jest DAG).
4. W przeciwnym razie  $L$  jest (któraś) kolejnością topologiczną.



# Problem najdłuższej drogi (13)

1. Problem LPP ma swój odpowiednik: problem znalezienia najkrótszej ścieżki prostej w całym grafie.
  - 1.1 Sytuacja w tym przypadku jest analogiczna, osobną kwestią jest sens praktyczny takiego problemu.
2. Jakie jest jednak praktyczne zastosowanie LPP?
  - 2.1 Zarządzanie projektami, szeregowanie zadań – wszędzie gdzie harmonogramujemy powiązane aktywności wymagające czasu.



# Szeregowanie zadań (1)

Rozpatrzmy projekt z 5 kamieniami milowymi (KM):

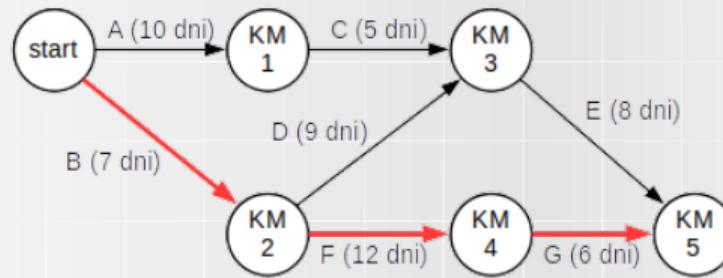
Kamień milowy	1	2	3	4	5
Wymagane zadania do ukończenia	A	B	C i D	F	E i G

Projekt składa się z 7 zadań:

Zadanie	A	B	C	D	E	F	G
Czas trwania (w dniach)	10	7	5	9	8	12	6
Wymagany kamień milowy	-	-	1	2	3	2	4

## Szeregowanie zadań (2)

Harmonogram projektu można reprezentować za pomocą DAG. Wierzchołki są kamieniami milowymi, krawędzie są zadaniami, zaś długość krawędzi jest czasem trwania zadania.

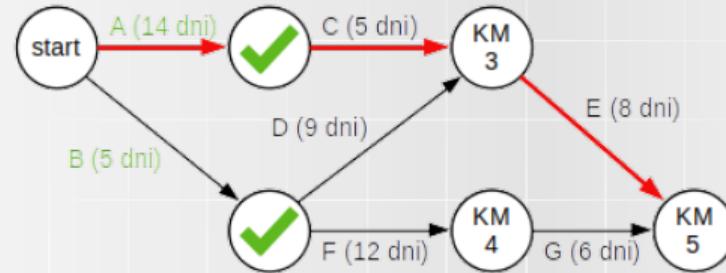


Najdłuższa ścieżka w grafie określa czas trwania projektu i zwana jest ścieżką krytyczną. Opóźnienia na ścieżce krytycznej wydłużają czas trwania projektu. Przyspieszenie zadań nie leżących na ścieżce krytycznej nie skróci czasu trwania projektu!



## Szeregowanie zadań (3)

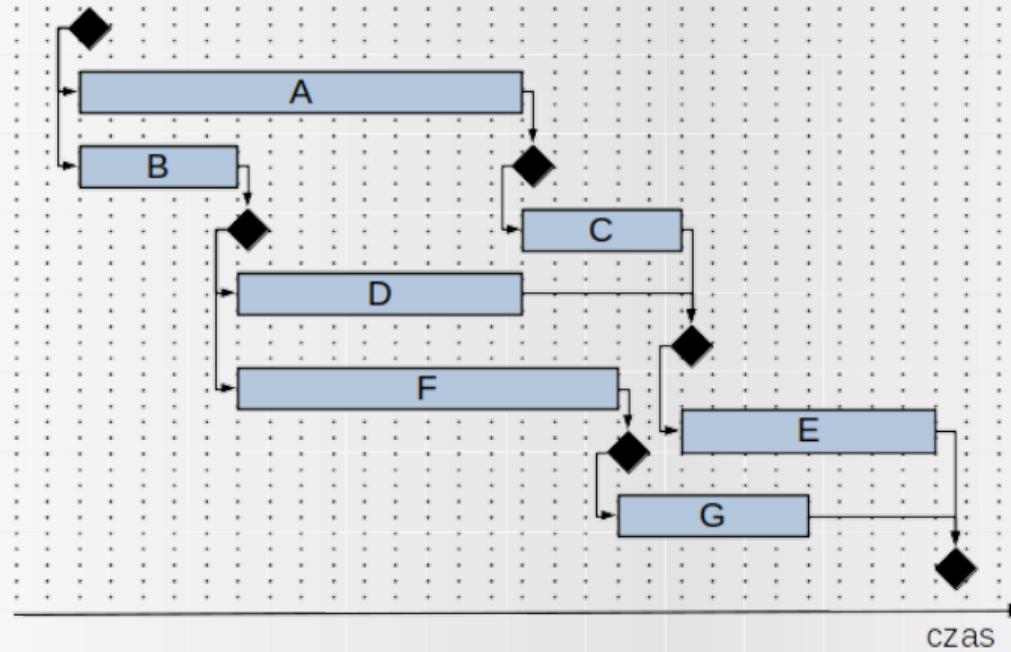
Przykład częściowo zakończonego projektu, w którym zadanie B zajęło 2 dni mniej niż zakładano, ale zadanie A opóźniło się 4 dni względem planu.



Ścieżka krytyczna zmieniła się (projekt trwa teraz 27 dni zamiast 25 pomimo skrócenia poprzedniej ścieżki krytycznej o 2 dni)!

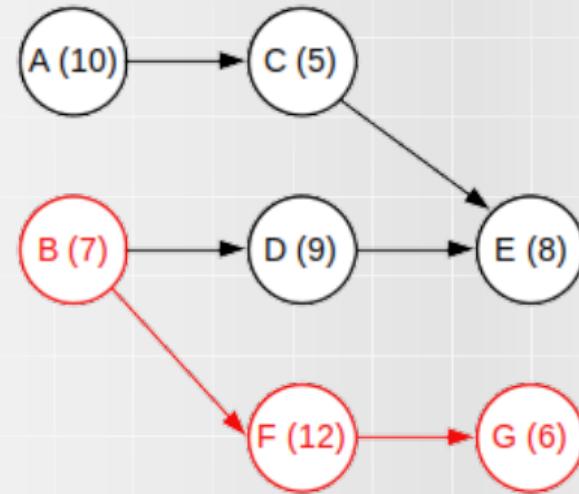
## Szeregowanie zadań (4)

Jednym ze sposobów wizualizowania harmonogramu projektu i analizowania ścieżek krytycznych może być diagram Gantta:



## Szeregowanie zadań (5)

Ten sam projekt można przedstawić w inny sposób: wierzchołki są zadaniami z przypisaną wagą (czasem zadania), zaś krawędzie (bez wag) określają wymaganą kolejność wykonywania zadań.





# Przepływowy problem szeregowania zadań (1)

Rozważmy następujący problem:

- ▶ Dany jest zakład produkcyjny (np. fabryka samochodów).
- ▶ Samochód tworzony jest w  $m = 4$  etapach: (1) zamontowanie podwozia, (2) zamontowanie silnika, (3) zamontowanie nadwozia, (4) lakierowanie.
  - ▶ Etapy muszą być wykonywane w tej kolejności (tzw. marszruta technologiczna).
  - ▶ Każdy etap ma inne stanowisko (tzw. maszynę).
- ▶ Mamy  $n = 6$  samochodów do wyprodukowania. Każdy taki samochód jest zadaniem.
  - ▶ Każde zadanie składa się z 4 operacji (bo mamy 4 etapy).
  - ▶ Każda operacja ma czas trwania.



## Przepływowy problem szeregowania zadań (2)

- ▶ Ponieważ maszyna może przetwarzać tylko jedno zadanie naraz, to musimy określić jaka będzie kolejność wykonywania zadań.
- ▶ Kolejność jest taka sama na każdej maszynie (jeśli A ma montowany silnik przed C, to A jest też lakierowany przed C).
- ▶ Celem jest określenie takiej kolejności (permutacji) wykonywania zadań, by zminimalizować czas całego procesu produkcji.
  - ▶ Szukamy więc takiej permutacji która minimalizuje czas najdłuższego (maksymalnego) zadania.
  - ▶ Taki problem nazywa się permutacyjnym przepływowym problemem szeregowania zadań.



# Przepływowowy problem szeregowania zadań (3)

Rozważmy konkretny przykład dla  $n = 6$  i  $m = 4$ :

Zadanie	A	B	C	D	E	F
Czas operacji 1 (podwozie)	4	3	7	2	4	8
Czas operacji 2 (silnik)	8	6	5	4	3	3
Czas operacji 3 (nadwozie)	5	9	3	7	1	6
Czas operacji 4 (lakier)	4	8	7	6	8	2

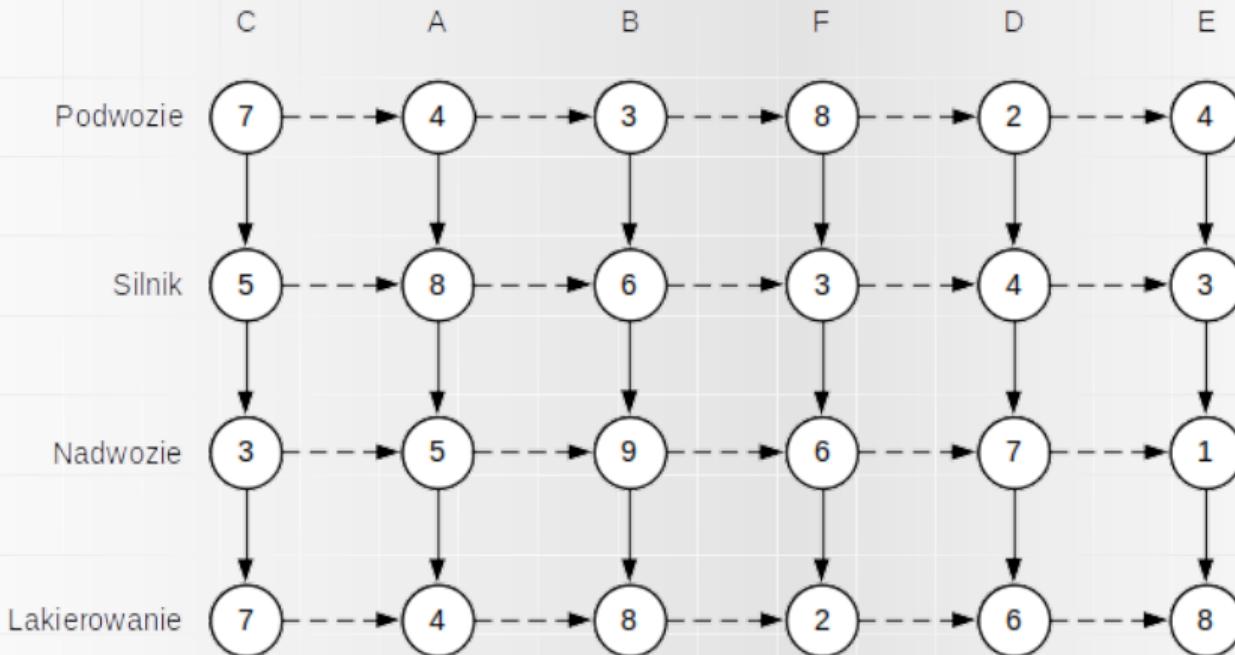


## Przepływowy problem szeregowania zadań (4)

- ▶ Rozważmy przykładowe rozwiązanie:  $(C, A, B, F, D, E)$ .
- ▶ Rozwiązanie to możemy przedstawić w postaci DAG, a konkretnie grafu typu kratownica.
- ▶ „Wiersze” reprezentują maszyny (ciąg operacji na jednej maszynie).
- ▶ „Kolumny” reprezentują zadania (ciąg operacji jednego zadania).
- ▶ Łuki określają które operacje wykonywane są po których.
  - ▶ Przez łuki ciągłe oznaczamy kolejność narzuconą przez problem (kolejność technologiczna).
  - ▶ Przez łuki kreskowane oznaczamy kolejność narzuconą przez rozwiązanie (kolejność zadaniowa).

# Przepływowowy problem szeregowania zadań (5)

Graf dla przytoczonego problemu i rozwiązania ( $C, A, B, F, D, E$ ).



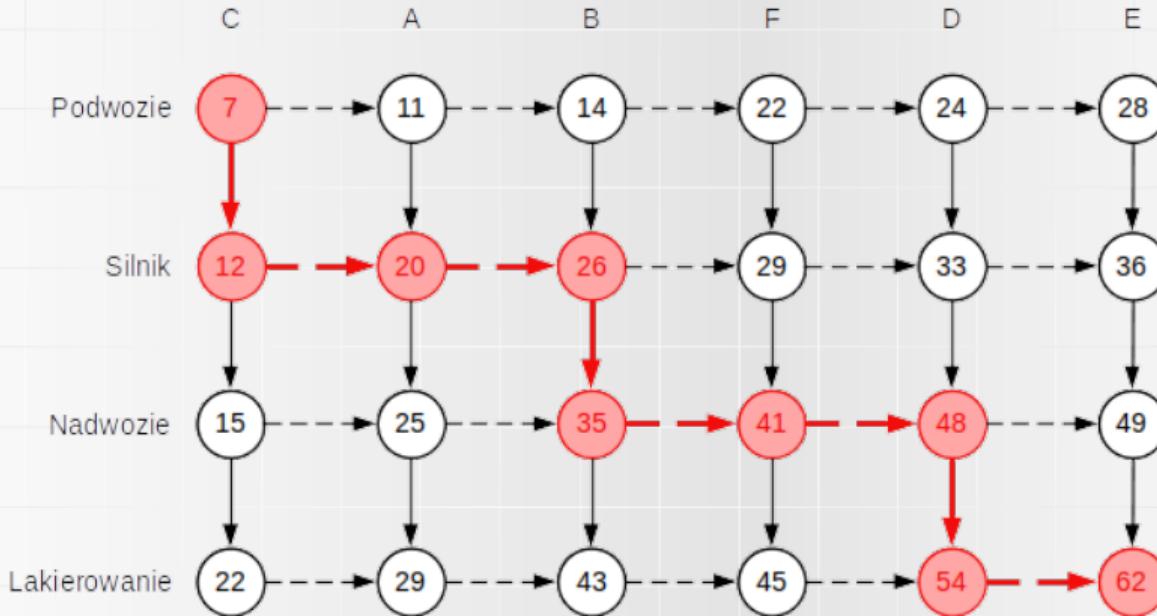


## Przepływowy problem szeregowania zadań (6)

- ▶ Dla operacji zadania  $i$  na maszynie  $j$  obliczymy jej czas zakończenia  $C_{i,j}$ .
- ▶ Operacja musi czekać na poprzednika technologicznego (na zakończenie operacji tego samego zadania na wcześniejszej maszynie).
- ▶ Operacja musi czekać na poprzednika zadaniowego (na zakończenie operacji wcześniejszego zadania na tej samej maszynie).
- ▶ Ostatecznie  $C_{i,j} = \max\{C_{i-1,j}, C_{i,j-1}\} + p_{i,j}$ .
  - ▶ Dla  $i = 1$  i/lub  $j = 1$  wzór się upraszcza (przy braku poprzednika przyjmujemy wartość 0).
  - ▶ Obliczamy macierz  $n \times m$  czasów zakończenia w odpowiedniej kolejności (musimy znać wartość z góry i z lewej).
    - ▶ Szukana wartość  $C_{\max} = C_{n,m}$ .

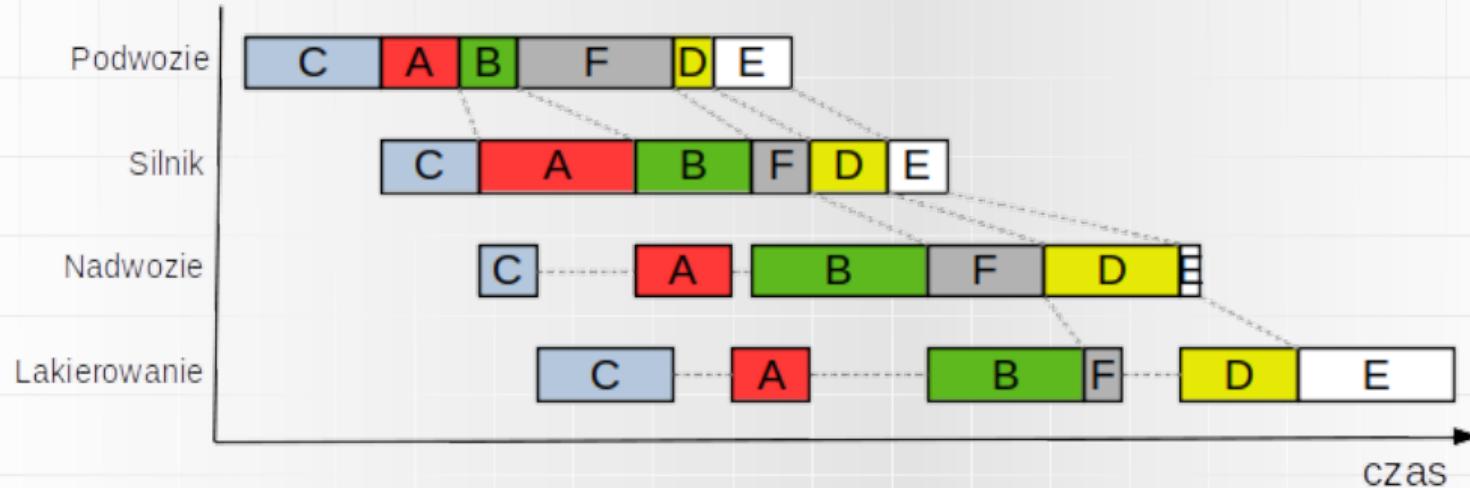
# Przepływowo problem szeregowania zadań (7)

Graf („macierz”) czasów zakończenia poszczególnych operacji z zaznaczoną ścieżką krytyczną:



# Przepływowy problem szeregowania zadań (8)

Diagram Gantta dla podanego przykładu:



Liniami poziomymi oznaczono, gdy operacja musiała czekać na poprzednika technologicznego, zaś ukośnymi gdy musiała czekać na poprzednika zadaniowego.

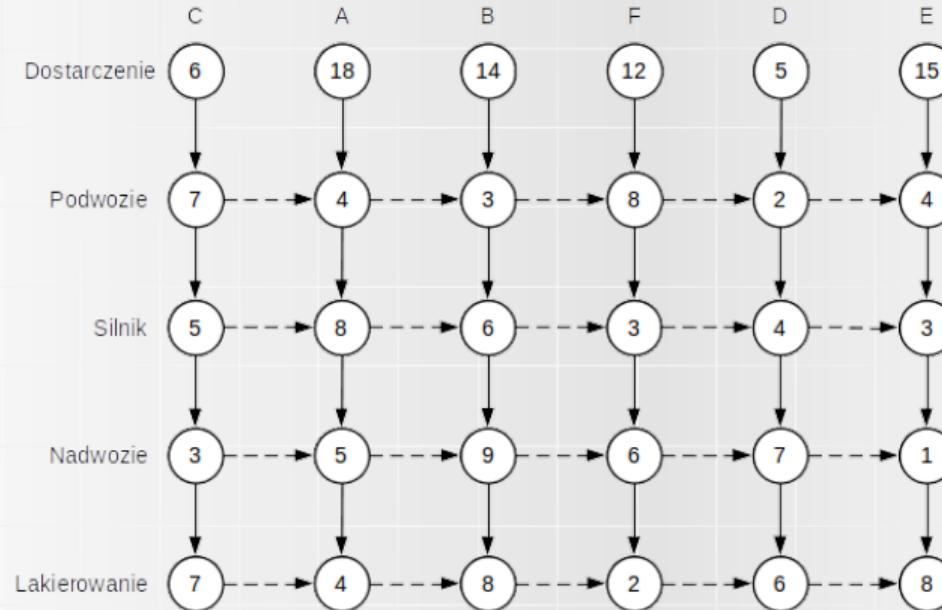


## Przepływowy problem szeregowania zadań (9)

- ▶ Rozpatrzmy modyfikację podstawowego problemu, które wpłyną na sposób konstrukcji grafu.
- ▶ Założymy, że przetwarzanie zadania może rozpocząć się dopiero po pewnym czasie (np. czekamy na dotarcie części).
  - ▶ Takie mechanizm najczęściej nazywa się czasem dostępności (ready time, release time, arrival time).
  - ▶ Możemy to zamodelować dodatkową operacją przed właściwym rozpoczęciem zadania.

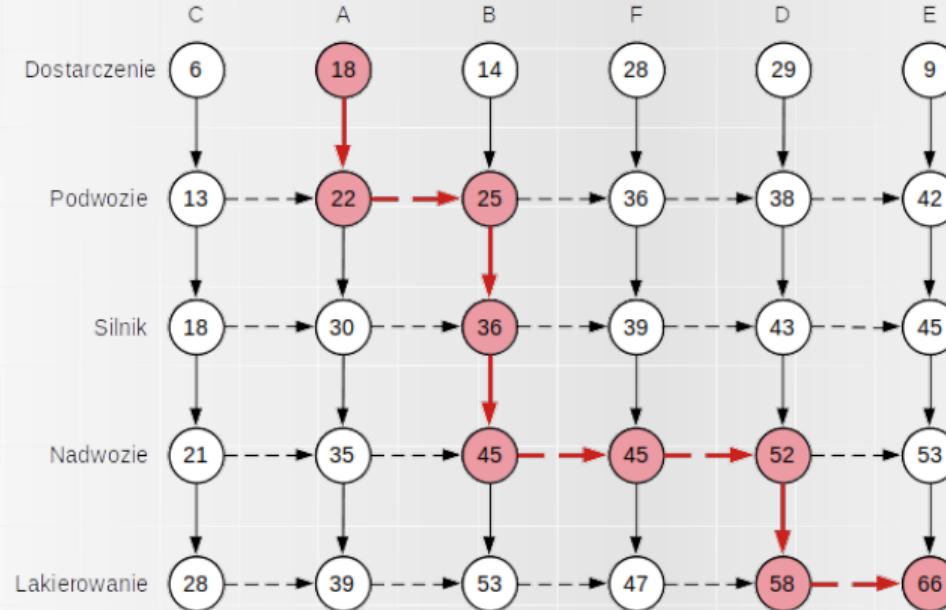
# Przepływowowy problem szeregowania zadań (10)

Graf rozwiązań (z przykładowymi czasami dostępności):



# Przepływowowy problem szeregowania zadań (11)

Graf czasów zakończenia z zaznaczoną ścieżką krytyczną:



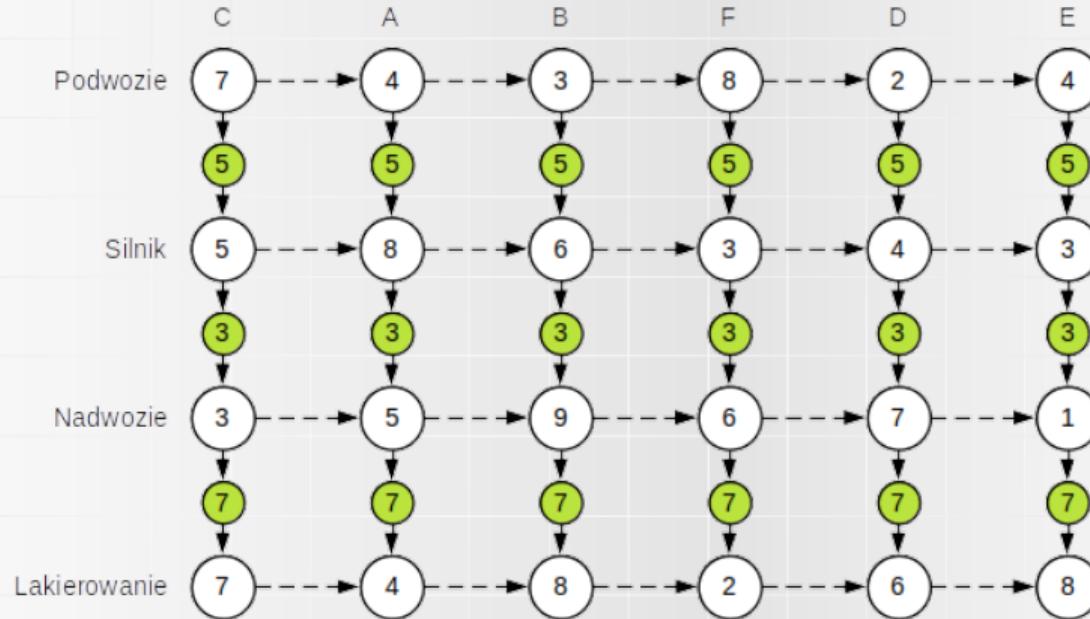


## Przepływowy problem szeregowania zadań (12)

- ▶ Rozpatrzmy inną modyfikację problemu. Założmy, że chcemy uwzględnić czas pomiędzy kolejnymi operacjami zadania (czas transportu).
  - ▶ Czas ten nie jest zaniedbywalny np. gdy kolejne etapy wykonywane są w różnych fabrykach.
- ▶ Czas transportu może być różny dla różnych zadań, ale założmy że jest stały (zależny tylko od tego z której na którą maszynę przenosimy).
- ▶ Założenie możemy zamodelować dodatkowym wierzchołkiem pomiędzy kolejnymi operacjami tego samego zadania tak że waga tego wierzchołka odda czas transportu.
  - ▶ Prościej jednak jest po prostu dodać wagę do istniejącej krawędzi pomiędzy operacjami zadania.

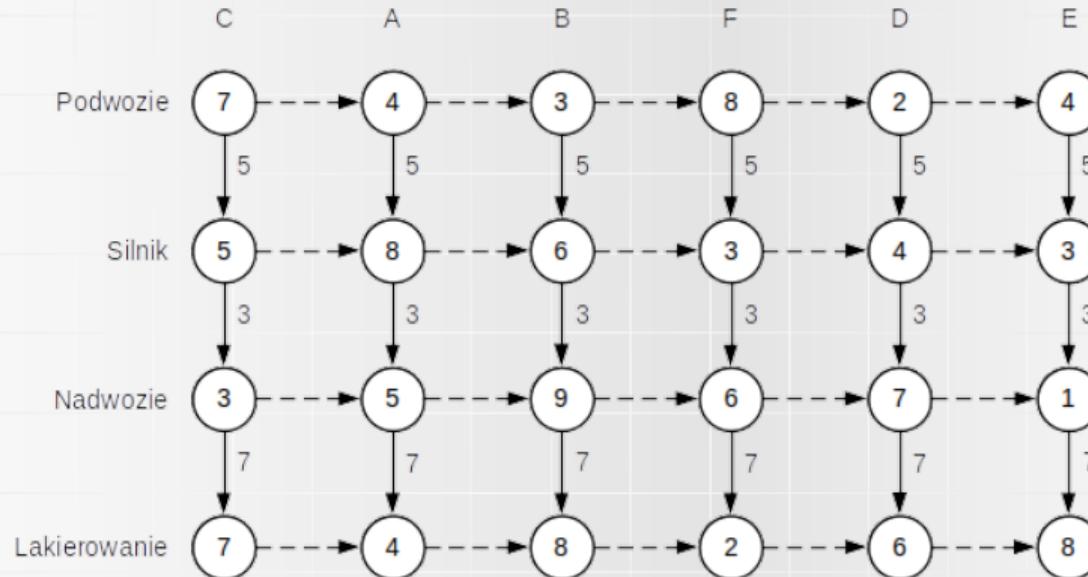
# Przepływowo problem szeregowania zadań (13)

Graf rozwiązań (z przykładowymi czasami transportu) w wersji z dodatkowymi wierzchołkami:



# Przepływowowy problem szeregowania zadań (14)

Graf rozwiązań (z przykładowymi czasami transportu) w wersji bez dodatkowych wierzchołków:



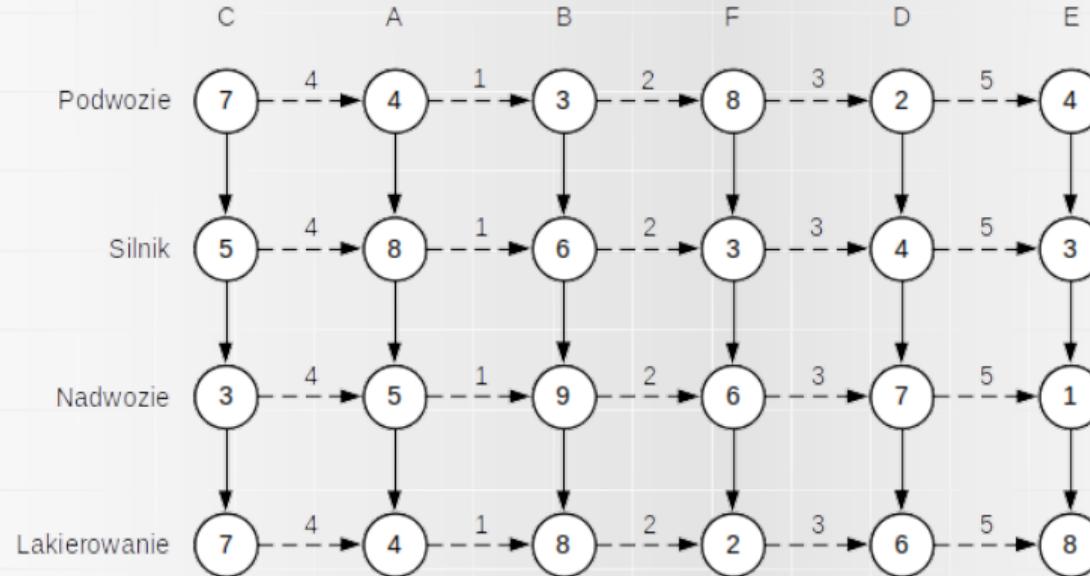


## Przepływowy problem szeregowania zadań (15)

- ▶ Kolejna modyfikacja: założmy, że pomiędzy operacjami maszynę trzeba przygotować przed kolejną operacją.
  - ▶ Może to obejmować np. czas potrzebny na wyładowanie poprzedniego detalu i załadowanie następnego, zmianę ustawień maszyny itp. W praktyce takie czynności nazywa się przezbrojeniem (setup).
- ▶ Znów założymy, że czas przezbrojenia jest stały dla maszyny (niezależny od tego z jakiego zadania na jakie przebrajamy).
- ▶ Czas przezbrojenia możemy zamodelować dodatkowymi wierzchołkami pomiędzy opercjami maszyny lub po prostu wagą krawędzi pomiędzy takimi operacjami.

# Przepływowo problem szeregowania zadań (16)

Graf rozwiązań (z przykładowymi czasami przezbrojenia) w wersji bez dodatkowych wierzchołków:





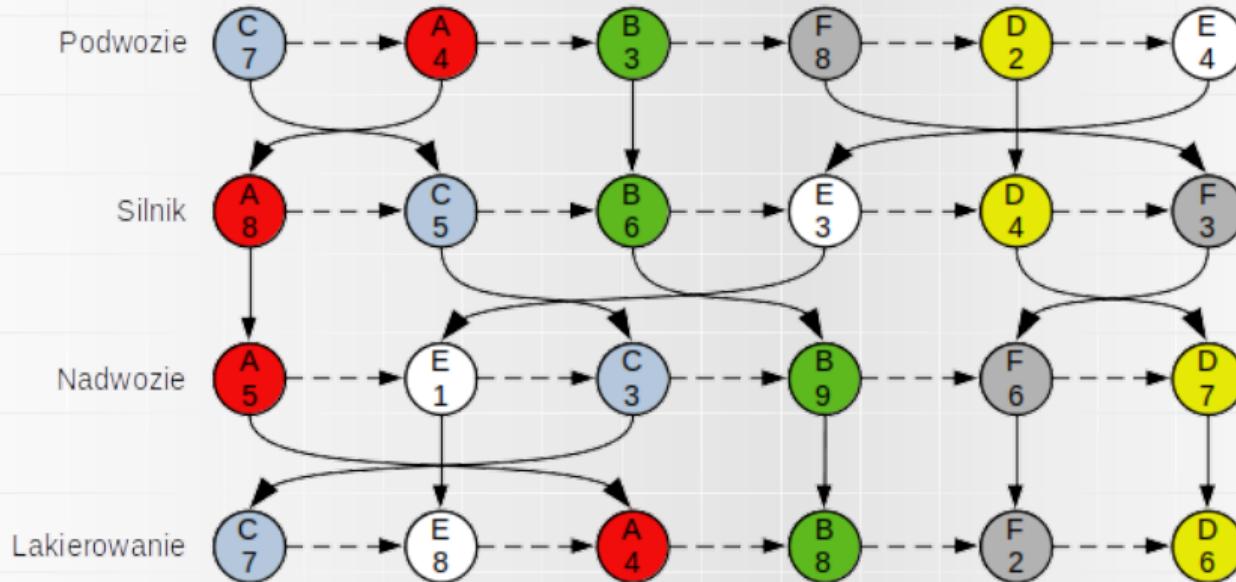
## Przepływowy problem szeregowania zadań (17)

- ▶ Permutacyjny problem przepływowego zakładał, że każda maszyna ma taką samą kolejność (permutację) wykonywania zadań.
- ▶ W ogólniejszym (niepermutacyjnym) problemie przepływowym każda maszyna może mieć inną kolejność.
  - ▶ Mamy więc tyle permutacji ile jest maszyn.
  - ▶ Taki problem wciąż możemy modelować grafem DAG, ale (w ogólności) nie jest on już kratownicą.

# Przepływowowy problem szeregowania zadań (18)

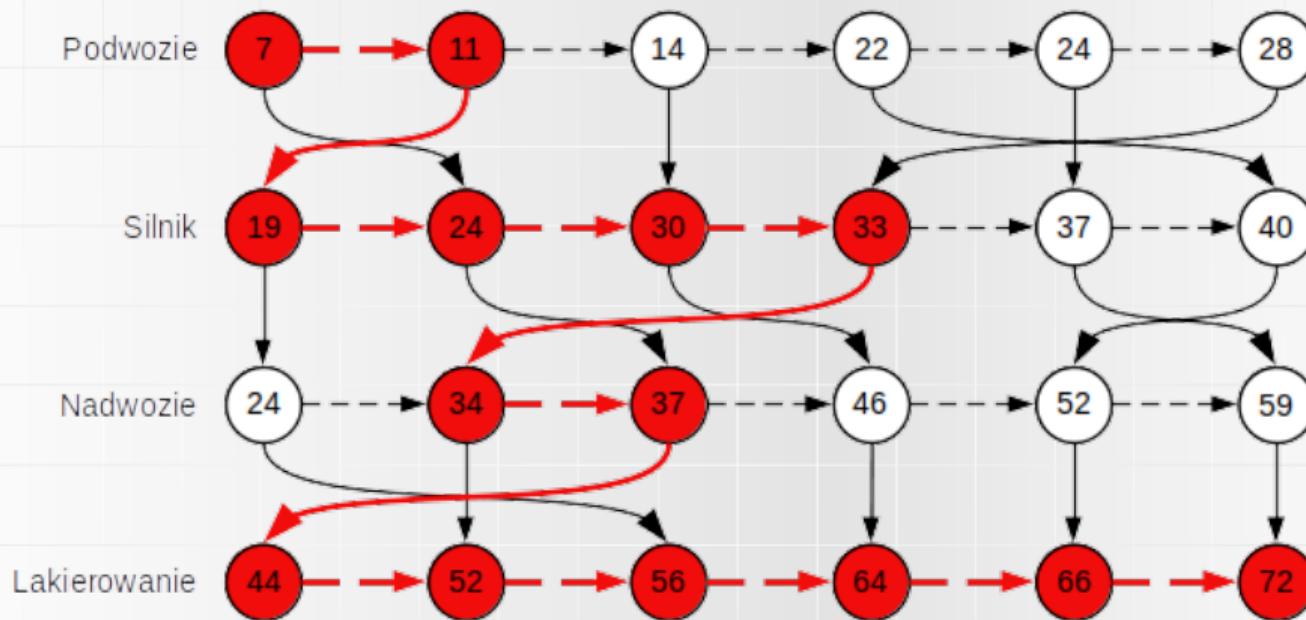
Graf dla przytoczonego problemu i rozwiązania

$((C, A, B, F, D, E), (A, C, B, E, D, F), (A, E, C, B, F, D), (C, E, A, B, F, D))$ .



# Przepływowo problem szeregowania zadań (19)

Graf czasów zakończenia poszczególnych operacji z zaznaczoną ścieżką krytyczną:





Wrocław  
University  
of Science  
and Technology

# Struktury danych

Wykład 12  
XML, DTD

dr inż. Jarosław Rudy





# XML (1)

- ▶ Dotychczas omawiane struktury zasadniczo dotyczyły danych przechowywanych w pamięci operacyjnej.
- ▶ Ponadto różne struktury miały różną reprezentację (format przechowywania danych).
- ▶ W praktyce istnieje potrzeba przechowywania danych poza aplikacją (pliki) oraz przekazywania danych pomiędzy aplikacjami (np. poprzez sieć).
- ▶ Potrzebny jest jeden ustandardzowany format mogący reprezentować praktycznie dowolne dane.
- ▶ Przykładem takiego formatu jest XML.



## XML (2)

- ▶ Extensible Markup Language (XML) – rozszerzalny język znaczników bazowany na języku SGML.
- ▶ Otwarty standard zaproponowany przez World Wide Web Consortium (W3C) opublikowany w 1998 roku.
- ▶ XML pozwala na serializację, przesłanie i odtworzenie danych pomiędzy dwoma systemami w ustandaryzowany sposób (interoperacyjność).
- ▶ Zaprojektowany z myślą o dokumentach, ale zasadniczo może służyć do reprezentacji dowolnych danych.



## Zastosowania XML-a:

- ▶ XHTML – przedstawienie HTML-a 4 w postaci XML-a.
- ▶ Office Open XML – format dla Microsoftowych dokumentów Word, Excel, PowerPoint.
- ▶ OpenDocument – analogiczny otwarty format dokumentów dla LibreOffice i OpenOffice.
- ▶ Pliki konfiguracyjne i dane wejściowe różnych aplikacji.
- ▶ Wiadomości RSS oraz Atom (kanały informacyjne, web feed).



- ▶ Protokoły komunikacyjne:
  - ▶ Simple Object Transfer Protocol (SOAP) – protokół komunikacyjny wykorzystywane w komunikacji usług sieciowych.
  - ▶ XMPP (dawniej Jabber) – protokół dla komunikatorów (GaduGadu, Tlen, Facebook, ICQ).
- ▶ Jeden z możliwych formatów dla Asynchronous JavaScript and XML (AJAX).
- ▶ Format wymiany informacji dla UML-a (poprzez XMI).
- ▶ Kompozycja graficzna (layout) dla aplikacji mobilnych Androida.
- ▶ MathML – matematyczny język znaczników.



Istotne pojęcia i składniki XML-a:

- ▶ Dokument XML składa się znaków. W wersji 1.1 dopuszczalne są wszystkie znaki Unicode (poza Nullem), choć niektóre są niezalecane.
- ▶ XML 1.0 posiada więcej znaków zakazanych.
- ▶ Przetwarzaniem dokumentów XML zajmuje się parser (procesor) XML. Specyfikacja określa sposób działania parsera.
- ▶ Dokument XML może zaczynać się opcjonalną deklaracją, która opisujące informacje o dokumencie np.:

```
<?xml version="1.0" encoding="UTF-8"?>
```



- ▶ Składnia XML-a wprowadza podział na znaczniki (markup) i zawartość (content). Generalnie są dwa rodzaje markupów:
  - ▶ Łańcuch znaków zaczynający się od znaku < i kończący się znakiem >.
  - ▶ Łańcuch znaków zaczynający się znakiem & i kończący się znakiem ; (tzw. encja, entity).
- ▶ Generalnie tekst nie będący markupiem jest zawartością.
- ▶ Wyjątkiem jest konstrukcja CDATA, która rozpoczyna się ciągiem <! [CDATA [ i kończy ciągiem ]]> – te dwa ciągi są traktowane jako markup, zaś tekst pomiędzy nimi jako zawartość.



```
<document>
  <book>
    <title>Pan Tadeusz</title>
    <author>Adam Mickiewicz</author>
    <year>1834</year>
  </book>
  <expression>123 &gt; 45</expression>
  <note>
    <![CDATA[<book> i &gt; to przykłady markupów ]]>
  </note>
</document>
```

Na rysunku kolorem wyróżniono markupy. Sekcja CDATA traktuje większość znaków dosłownie (nie jest to jednak to samo co komentarz!)



- ▶ Na bazie markupów wyróżnia się znaczniki (tag). Możliwe są 3 rodzaje tagów:
  - ▶ Znacznik otwierający np. <book>
  - ▶ Znacznik zamykający np. </book>
  - ▶ Znacznik elementu pustego np. <book/>
- ▶ Element – składnik dokumentu, w jednej z dwóch form:
  - ▶ Znacznik elementu pustego.
  - ▶ Ciąg od znacznika otwierającego do znacznika zamykającego.
    - ▶ Tekst pomiędzy znacznikami nazywany jest zawartością elementu. Zawartość ta może zawierać kolejne elementy (elementy potomne).
    - ▶ XML ma więc strukturę drzewiastą.

# XML

Przykład podziału dokumentu na tagi otwierające (zielone), zamkajające (czerwone) i elementu pustego (żółte).

```
<student>
    <name>Michał</name>
    <age>20</age>
    <grades>
        <algebra>5.0</algebra>
        <databases>4.0</databases>
        <algorithms>4.5</algorithms>
    </grades>
    <stipend/>
</student>
```



Ten sam dokument podzielony na elementy.

```
<student>
    <name>Michał</name>
    <age>20</age>
    <grades>
        <algebra>5.0</algebra>
        <databases>4.0</databases>
        <algorithms>4.5</algorithms>
    </grades>
    <stipend/>
</student>
```



## Atrybuty:

- ▶ „Argumenty” elementu występujące w znaczniku początkowym lub znaczniku elementu pustego.
- ▶ Atrybut jest parą nazwa-wartość zapisywana jako `nazwa="wartość"`.
- ▶ Atrybut o danej nazwie może w elemencie wystąpić tylko raz.
- ▶ Element może mieć dowolną liczbę atrybutów.



## Struktura uczelni z atrybutami elementów:

```
<uczelnia nazwa="Pwr" miasto="Wrocław">
    <wydział nazwa="W4" dyscyplina="ITT">
        <katedra nazwa="K28" kierownik="Wojciech Bożejko">
            <pracownik stanowisko="profesor">Wojciech Bożejko</pracownik>
            <pracownik stanowisko="adiunkt">Jarosław Rudy</pracownik>
            <pracownik stanowisko="asystent">Piotr Nowak</pracownik>
        </katedra>
        <katedra nazwa="K30" kierownik="Ewa Skubalska-Rafajłowicz">
            <pracownik stanowisko="profesor">Ewa Skubalska-Rafajłowicz</pracownik>
            <pracownik stanowisko="adiunkt">Tomasz Kubik</pracownik>
            <pracownik stanowisko="asystent">Łukasz Jeleń</pracownik>
        </katedra>
    </wydział>
    <wydział nazwa="W12" dyscyplina="AEE">
        <katedra nazwa="K29" kierownik="Ignacy Dulęba ">
            <pracownik stanowisko="profesor">Ignacy Dulęba</pracownik>
            <pracownik stanowisko="adiunkt">Krzysztof Arent</pracownik>
        </katedra>
    </wydział>
</uczelnia>
```



# XML

Decyzja czy niektóre informacje umieścić w atrybutach czy w elementach potomnych jest często kwestią preferencji lub wygody:

```
<uczelnia nazwa="Pwr" miasto="Wrocław">  
</uczelnia>  
  
<uczelnia>  
    <nazwa>PWr</nazwa>  
    <miasto>Wrocław</miasto>  
</uczelnia>
```



# XML

Atrybut może mieć tylko pojedynczą wartość. Gdy konieczne jest wiele wartości, należy je zakodować (np. jak lista oddzielona przecinkami, średnikami lub spacjami) i zdekodować osobno przy parsowaniu XML-a.

```
<projekt zespół="Marek,Kasia,Robert">  
  
<div style="font-size: 20px; padding: 2em; color: red">  
  
<div class="main-page fancy blue">
```



# XML

## Kodowanie znaków:

- ▶ Można wykorzystać kodowania zdefiniowane przez Unicode, w szczególności kodowania UTF-8 oraz UTF-16.

```
<?xml version="1.0" encoding="ASCII"?>  
<person>Alan Turing</person>
```

- ▶ Można wykorzystywać wiele innych kodowań wykorzystujących znaki Unicode, takich jak kodowanie ASCII, kodowanie ISO-8859-1 itd.

```
<?xml version="1.0" encoding="ISO-8859-1"?>  
<person>Erwin Schrödinger</person>
```

- ▶ XML może sam wykrywać kodowanie, ale standard gwarantuje to jedynie dla UTF-8 i (formalnie) UTF-16, reszta zależy od parsera.



Czasami bezpośrednie użycie niektórych znaków jest problematyczne:

- ▶ Znaki specjalne dla parsera XML jak < czy &.
- ▶ Znaki Unicode niedostępne w obecnym kodowaniu (np. znak 中 w kodowaniu ASCII).
- ▶ Znaki niemożliwe do wprowadzenia na komputerze użytkownika.
- ▶ Znaki o prawie identycznym wyglądzie (np. "A" w alfabetie łacińskim kontra "A" w cyrylicy).

Problemy te można rozwiązać stosując mechanizmy ucieczki (escaping).



# XML

XML definiuje 5 domyślnych encji (dużo mniej niż HTML!):

- ▶ Encja &lt; dla znaku <.
- ▶ Encja &gt; dla znaku >.
- ▶ Encja &amp; dla znaku &.
- ▶ Encja &apos; dla znaku '.
- ▶ Encja &quot; dla znaku ".



- ▶ Dodatkowe encje można używać po uprzednim ich zdeklarowaniu w specyfikacji Document Type Definition (o czym później). Wielkość znaków ma znaczenie.
- ▶ Znaki Unicode można wprowadzić również za pomocą ich numeru w Unicode:
  - ▶ Dziesiętnie: &#nnnn;
  - ▶ Szesnastkowo: &#xhhhh;
- ▶ Początkowe x musi być małą literą, pozostałe reguły są luźne.
- ▶ Przykładowo znak 中 może być wprowadzony jako &#20013; albo &#x4e2d;



- ▶ Sekcja CDATA (character data) służy do zapisu znaków, które nie mają być interpretowane przez parser („masowy” escaping).
  - ▶ Jest to szczególnie przydatne przy próbie zapisu kodu XML-a w XML-u.
  - ▶ Przykładowo dosłowny ciąg <brand>H&M</brand> można zapisać z użyciem encji lub CDATA:
    - ▶ &lt;brand&ampgtH&M&lt;/brand&ampgt.
    - ▶ <! [CDATA [<brand>H&M</brand>] ]>.
- ▶ Ciąg CDATA nie może się zagnieździć i nie może zawierać ciągu ]]>, ale drugi problem można obejść.



- ▶ Komentarze w XML-u zaczynają się ciągiem `<!--` i kończą ciągiem `-->`.
- ▶ Dla kompatybilności z SGML komentarze nie mogą zawierać ciągu `--` (więc nie mogą się zagnieździć).
- ▶ Wewnątrz komentarzy nie działają encje, więc można używać tylko znaków z aktualnego kodowania.

`<!-- kodowanie ciągu -->` w sekcjach CDATA `-->`  
`<![CDATA[ ]]]><![CDATA[>]]>`



## Przestrzenie nazw (namespaces)

- ▶ Pozwalają unikać niejednoznaczności identycznie nazwanych elementów.
- ▶ Deklaracja za pomocą atrybutu `xmlns:prefix="URI"`, gdzie URI to jakiś Uniform Resource Identifier (np. URL).
  - ▶ Elementy i atrybuty o nazwie zaczynające się od prefix: są w tej przestrzeni nazw, o ile mają powyższą deklarację.
  - ▶ Elementy potomne takiego elementu także są w tej przestrzeni nazw.
- ▶ Przestrzeń domyślną można deklarować z użyciem: `xmlns="URI"`



## Wersje XML-a:

- ▶ Wersja 1.0:
  - ▶ Obecnie piąta edycja.
  - ▶ Wciąż stosowana i rekomendowana.
  - ▶ Mniejszy zakres dopuszczalnych znaków.
  - ▶ Do edycji czwartej ograniczony był zbiór znaków mogących stanowić identyfikatory (nazwy elementów, atrybutów itp.).
  - ▶ W edycji piątej ograniczenia są mniejsze (jak w XML 1.1).



Przykład wykorzystania znaków spoza ASCII w nazwach elementów i atrybutów:

```
<?xml version="1.0" encoding="UTF-8"?>
<俄语_ЛбqнL="ռուսերեն">данные</俄语>
```

► Wersja 1.1:

- Obecnie druga edycja.
- Kilka spornych usprawnień.
- Implementacja nie jest powszechnie stosowana.
- Niezalecany, chyba że potrzebne są jego unikalne cechy.



► Wersja 2.0:

- Pojawiły się plany i dyskusje dotyczące m.in. dodania pewnych narzędzi (np. XML Base, XML Information Set) do standardu czy eliminacji DSD.
- Żadna organizacja nie prowadzi obecnie prac nad XML 2.0.

► MicroXML:

- Okrojony XML, może być przydatny w zastosowaniach, gdzie zwykły XML jest zbyt złożony.
- MicroXML ma uzupełniać XML, JSON i HTML, a nie je zastąpić.



# XML

Dokumenty XML muszą być poprawnie sformułowane (well-formed), czyli muszą spełniać wymogi składniowe XML-a. Inne dokumenty po prostu nie są XML-em. Niektóre reguły poprawnego formułowania:

- ▶ Brak znaków Unicode spoza użytego kodowania.
- ▶ Znaki specjalne (<, & itp.) stosowane tylko do określenia markupów.
- ▶ Tagi są poprawnie zagnieżdżone, sparowane (wielkość znaków ma znaczenie) itd.
- ▶ Tagi nie mogą zawierać niektórych znaków (w tym spacji) i nie mogą zaczynać się od cyfr, dywizu (-) czy kropki.
- ▶ Dokument ma pojedynczy element jako korzeń.



# XML

- ▶ Jeśli parser XML-a wykryje niezgodność, jest zobowiązany zwrócić błąd i zakończyć parsowanie.
- ▶ Jest to podejście ekstremalne i zupełnie inne niż HTML, dla którego przeglądarki starają się wygenerować rozsądную stronę nawet przy niezgodności pewnych elementów.
- ▶ HTML zaakceptuje <br> zamiast <br/>, a XHTML – nie.
- ▶ Podejście XML-a łamię więc tzw. zasadę Postela (robustness principle):

Bądź konserwatywny w tym, co wysyłasz, bądź liberalny w tym, co akceptujesz (nadawca powinien ściśle trzymać się specyfikacji, zaś odbiorca powinien akceptować odstępstwa, o ile przekaz pozostaje zrozumiały).



- ▶ Oprócz poprawnego sformułowania (well-formed), dokumenty XML mogą być zwalidowane (valid).
- ▶ W zwalidowanym XML-u elementy i atrybuty spełniają dodatkowe wymagania nałożone przez zdefiniowaną (w XML-u lub osobno) specyfikcję.
- ▶ Parsery różnią się tym czym przeprowadzają etap walidacji.
  - ▶ Parser musi zgłosić błąd walidacji, ale może kontynuować parsowanie.
  - ▶ Istnieje kilka sposobów określania schematów/gramatyk walidacyjnych dla XML-a m.in. DTD oraz XML Schema.



## Document Type Definition (DTD):

- ▶ Definiuje strukturę dokumentów SGML-a i jego pochodnych (XML-a, HTML-a i XHTML-a).
- ▶ Umożliwia m.in. określenie (z użyciem prostych regexów) jakie elementy i atrybuty ma lub może zawierać dokument XML.
- ▶ Można też definiować encje.
- ▶ Możliwy do zawarcia bezpośrednio w XML lub zewnętrznie w osobnym pliku.
- ▶ Druga opcja pozwala zastosować ten sam DTD dla różnych dokumentów XML.



## Zalety DTD (głównie względem XML Schema):

- ▶ Mocno rozpowszechniony ze względu na dołączenie do standardu XML 1.0
- ▶ Możliwość zawarcia DTD bezpośrednio w dokumencie XML.
- ▶ Większa zwięzłość.
- ▶ Możliwość deklarowania encji.
- ▶ Opis dokumentu w jednym miejscu.



## Wady DTD (głównie względem XML Schema):

- ▶ Brak wsparcia niektórych mechanizmów (przestrzenie nazw).
- ▶ Mniejsze możliwości (mniejsza ekspresywność, brak możliwości wyrażania konkretnej liczności, brak typów danych).
- ▶ Mniejsza czytelność, zwłaszcza przy zastosowaniu sparametryzowanych encji.
- ▶ DTD pisany jest w innym języku niż XML.



Deklaracja <ELEMENT! xyz abc> – opisuje regułę dla elementu xyz jako abc.  
Jako abc można użyć:

- ▶ EMPTY – element nie może mieć zawartości (ani tekst, ani elementy, z wyjątkiem spacji).
- ▶ ANY – dowolna zawartość.
- ▶ (#PCDATA) – zawartość tekstowa (parsed character data) tzn. element nie może mieć elementów potomnych.



- ▶ (#PCDATA | e11 | e12 | ...)\* – co najmniej dwoje dzieci, którymi może być tekst lub elementy podane na liście. W dowolnej kolejności i ilości.
- ▶ e1 – zawartością jest element o nazwie e1.
- ▶ (e11, e12, ...) – lista uporządkowana. Zawartością elementu muszą być elementy (nie tekst!) w podanej kolejności.
- ▶ e11 | e12 | ... – ciąg alternatyw. Zawartością elementu musi być dokładnie jeden z podanych elementów (nie tekst!).



Składnikom na listach można przypisać uproszczoną krotność za pomocą znaku bezpośrednio po składniku:

- ▶ + – składnik musi pojawić się 1 lub więcej razy (za każdym razem jego zawartość może być inna!).
- ▶ \* – składnik musi pojawić się 0 lub więcej razy (jest więc opcjonalny, za każdym razem zawartość może być inna).
- ▶ ? – składnik musi pojawić się 0 lub 1 raz (jest opcjonalny).
- ▶ Brak krotności – składnik musi pojawić się dokładnie raz (jest obowiązkowy). Jest to krotność domyślna.



Przykład DTD dla XML-a opisującego pracowników:

```
<!ELEMENT pracownicy (pracownik+)>
<!ELEMENT pracownik (daneOsobowe, dział?)>

<!ELEMENT daneOsobowe (imie, drugieimie?, nazwisko)>
<!ELEMENT imie (#PCDATA)>
<!ELEMENT drugieimie (#PCDATA)>
<!ELEMENT nazwisko (#PCDATA)>

<!ELEMENT dział (#PCDATA)>
```



Przykład XML-a dla wcześniejszego DTD, wraz z deklaracją zewnętrznego DTD:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE pracownicy SYSTEM "pracownicy.dtd">
<pracownicy>
    <pracownik>
        <daneOsobowe>
            <imie>Izabela</imie>
            <nazwisko>Nowakowska-Jasnorzębska</nazwisko>
        </daneOsobowe>
    </pracownik>
    <pracownik>
        <daneOsobowe>
            <imie>Zygflyd</imie>
            <drugieimie>Zenobiusz</drugieimie>
            <nazwisko>Wawrzyniak</nazwisko>
        </daneOsobowe>
        <dział>Public Relations</dział>
    </pracownik>
</pracownicy>
```



Przykład XML-a z wewnętrznym DTD:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE osoba [
    <!ELEMENT osoba (imie, drugieimie, nazwisko)>
    <!ELEMENT imie (#PCDATA)>
    <!ELEMENT drugieimie (#PCDATA)>
    <!ELEMENT nazwisko (#PCDATA)>
]>
<osoba>
    <imie>Amadeusz</imie>
    <drugieimie>Zenon</drugieimie>
    <nazwisko>Kowalski</nazwisko>
</osoba>
```



Deklaracja <!ATTLIST! pozwala określić dopuszczalne atrybuty danego elementu.

Dla każdego atrybutu określamy:

- ▶ nazwę atrybutu.
- ▶ typ wartości:
- ▶ obowiązkowość/wartość domyślną.

```
<!ATTLIST img
    src    CDATA          #REQUIRED
    id     ID             #IMPLIED
    sort   CDATA          #FIXED "true"
    print  (yes | no) "yes"
>
```



Niektóre sposoby definiowania wartości atrybutu:

- ▶ CDATA – wartość będąca „dowolnym” (patrz #FIXED) tekst.
- ▶ ID – tekst będący identyfikatorem. Zwalidowany dokument nie może mieć dwóch takich samych identyfikatorów.
- ▶ IDREF oraz IDREFS – identyfikator lub ich lista pasująca do jakiegoś id w dokumencie.
- ▶ ENTITY oraz ENTITIES – nazwa encji (lub ich lista) zadeklarowanej w DTD.
- ▶ (val1 | val2 | ...) – wylistowany zbiór dopuszczalnych wartości.



Obowiązkowość i wartości domyślne:

- ▶ #REQUIRED – atrybut musi wystąpić.
- ▶ #IMPLIED – atrybut nie musi wystąpić.
- ▶ #FIXED "val" – atrybut ma ustaloną wartość "val".
- ▶ "val" – w przypadku braku atrybutu domyślną wartością jest "val".



Obowiązkowość i wartości domyślne:

- ▶ #REQUIRED – atrybut musi wystąpić.
- ▶ #IMPLIED – atrybut nie musi wystąpić.
- ▶ #FIXED "val" – atrybut ma ustaloną wartość "val".
- ▶ "val" – w przypadku braku atrybutu domyślną wartością jest "val".



Przykład deklaracji i użycia encji:

```
<!DOCTYPE sgml [
    <!ELEMENT sgml ANY>
    <!ENTITY % std      "standard SGML">
    <!ENTITY % signature "&#x2014; &author;.">
    <!ENTITY % question  "Why couldn't I publish my books directly in %std;?">
    <!ENTITY % author    "William Shakespeare">
]>
```

```
<sgml>&question;&signature;</sgml>
```



Wrocław  
University  
of Science  
and Technology

# Struktury danych

Wykład 13  
XML (schema, parsowanie)

dr inż. Jarosław Rudy





# XML Schema

- ▶ Tak jak DTD jest to konkretny język schematu XML (ang. XML Schema) w przeciwieństwie do całej rodziny schematów (ang. XML schemas).
- ▶ Ze względu na powyższe nazywany też jest W3C XML Schema (WXS) lub XML Schema Definition (XSD).
- ▶ Standard XSD 1.0 rekomendowany przez W3C w 2001 roku.
  - ▶ Druga edycja opublikowana w 2004 roku.
- ▶ Standard XSD 1.1 rekomendowany przez W3C w 2012 roku.
- ▶ Pisany w XML-u (tj. dokumenty XSD przypominają dokumenty XML-a).



# XML Schema

- ▶ Dokumenty XSD definiują schemat tj. metadane zorganizowane za pomocą tzw. komponentów schematu.
- ▶ Dokumenty XSD i komponenty organizowane są za pomocą przestrzeni nazw.
  - ▶ Dokument XSD może dodawać (include) inne dokumenty XSD z tej samej przestrzeni nazw.
  - ▶ Dokument XSD może importować (import) inne dokumenty XSD z innych przestrzeni nazw.
- ▶ Podczas walidacji dokument XML kojarzony jest z pożdanym schematem za pomocą podania parametru dla parsera/walidatora lub specjalnych atrybutów (np. `xsi:schemaLocation`) wewnątrz XML-a.



# XML Schema

Komponenty schematu:

- ▶ Deklaracja elementu
  - ▶ Określenie nazwy i przestrzeni nazw.
  - ▶ Określenie typu elementu – narzucenie ograniczeń na możliwe atrybuty i zawartość elementu.
  - ▶ Możliwość uzależnienia typu od wartości atrybutów.
  - ▶ Możliwość występowania elementu w miejsce innego (substitution groups).
  - ▶ Możliwość wymagania unikalności elementu w danym poddrzewie.



# XML Schema

- ▶ Możliwość wymagania by wartość pasowała do istniejącego identyfikatora jakiegoś elementu.
- ▶ Deklaracje globalne lub lokalne (np. różne deklaracje dla elementów o taka samej nazwie).
- ▶ Deklaracja atrybutu
  - ▶ Określenie nazwy i przestrzeni nazw.
  - ▶ Określenie typu atrybutu – ograniczenie na dopuszczalne wartości.
  - ▶ Możliwość określenia wartości domyślnej.
  - ▶ Możliwość określenia stałej wartości.



# XML Schema

- ▶ Model (element) group.
- ▶ Attribute group.
- ▶ Attribute use – określenie czy atrybut dla danego typu złożonego (np. elementu) jest obowiązkowy.
- ▶ Element particle – określenie dla typu złożonego minimalnej i maksymalnej liczby wystąpień.
- ▶ Inne.



# XML Schema

## Proste typy danych

- ▶ Wykorzystywane do ograniczenia postaci wartości tekstowej elementu lub atrybutu (duża różnica względem DTD).
- ▶ 19 prymitywnych (podstawowych) typów danych m.in. decimal, double, duration, dateTime, string, boolean, anyURI, base64Binary.
- ▶ Pochodne typy danych tworzone poprzez:
  - ▶ Ograniczenie zakresu wartości typu prymitywnego.
  - ▶ Dopuszczenie listy wartości (typ sekwencyjny).
  - ▶ Unia (kilka dopuszczalnych typów wartości).



# XML Schema

- ▶ Specyfikacja XSD definiuje 25 typów pochodnych.
- ▶ Dalsze typy można definiować poprzez schematy.
- ▶ Przykładowe ograniczenia dla typu pochodnego:
  - ▶ Określenie wartości minimalnej i maksymalnej.
  - ▶ Wyrażenie regularne.
  - ▶ Długość łańcucha tekstowego.
  - ▶ Liczba cyfr.
  - ▶ (Dla XSD 1.1) asercje z użyciem wyrażeń XPath 2.0.



## Typy złożone

- ▶ Określają dopuszczalne atrybuty i zawartość elementu. Zawartością może być:
  - ▶ Tylko element (bez zawartości tekstowej).
  - ▶ Tylko tekst.
  - ▶ Brak zawartości.
  - ▶ Zawartość mieszana (element lub tekst).
- ▶ Podobnie jak wcześniej, typ złożony może być zmodyfikowany przez ograniczenie dopuszczalnych atrybutów/elementów lub asercje. Można też rozszerzyć dopuszczalny zakres atrybutów/elementów.



# XML Schema

## Przykład XML Schema (z W3Schools):

```
<?xml version="1.0" encoding="UTF-8" ?>
<xss:schema xmlns:xss="http://www.w3.org/2001/XMLSchema">
  <xss:element name="shiporder">
    <xss:complexType>
      <xss:sequence>
        <xss:element name="orderperson" type="xs:string"/>
        <xss:element name="shipto">
          <xss:complexType>
            <xss:sequence>
              <xss:element name="name" type="xs:string"/>
              <xss:element name="address" type="xs:string"/>
              <xss:element name="city" type="xs:string"/>
              <xss:element name="country" type="xs:string"/>
            </xss:sequence>
          </xss:complexType>
        </xss:element>
        <xss:element name="item" maxOccurs="unbounded">
          <xss:complexType>
            <xss:sequence>
              <xss:element name="title" type="xs:string"/>
              <xss:element name="note" type="xs:string" minOccurs="0"/>
              <xss:element name="quantity" type="xs:positiveInteger"/>
              <xss:element name="price" type="xs:decimal"/>
            </xss:sequence>
          </xss:complexType>
        </xss:element>
      </xss:sequence>
      <xss:attribute name="orderid" type="xs:string" use="required"/>
    </xss:complexType>
  </xss:element>
```



# XML Schema

Przykładowy XML dla powyższego XML Schema (z W3Schools):

```
<?xml version="1.0" encoding="UTF-8"?>
<shiporder orderid="889923"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="shiporder.xsd">
    <orderperson>John Smith</orderperson>
    <shipto>
        <name>Ola Nordmann</name>
        <address>Langgt 23</address>
        <city>4000 Stavanger</city>
        <country>Norway</country>
    </shipto>
    <item>
        <title>Empire Burlesque</title>
        <note>Special Edition</note>
        <quantity>1</quantity>
        <price>10.90</price>
    </item>
    <item>
        <title>Hide your heart</title>
        <quantity>1</quantity>
        <price>9.90</price>
    </item>
</shiporder>
```



# XML Schema

Wady XML Schema (m.in.):

- ▶ Złożoność specyfikacji.
- ▶ Niekonsekwencja (np. ograniczenie wartości dla atrybutów działa inaczej niż dla elementów).
- ▶ Słabe wsparcie dla list nieuporządkowanych.
- ▶ Nie można określić co jest korzeniem dokumentu.
- ▶ Nie można definiować zawartości w zależności od kontekstu.



# Parsowanie XML-a

- ▶ Cel projektowy XML-a zakładał łatwość jego przetwarzania.
- ▶ Pomimo tego sama specyfikacja zawiera niewiele informacji o sposobach przetwarzania lub API.
- ▶ W praktyce wykształciło się kilka metod:
  - ▶ SAX.
  - ▶ DOM.
  - ▶ StAX.
  - ▶ XSLT.
  - ▶ XML data binding.



## Simple API for XML (SAX):

- ▶ Technika oparta o zdarzenia (event-based). Zdarzenia (np. otwarcie znacznika) powodują wywołanie odpowiedniej funkcji zarejestrowana przez użytkownika (callback).
- ▶ Parser przechodzi przez dokument jednokrotnie.
- ▶ Niskie zużycie pamięci (proporcjonalne do wysokości drzewa).
- ▶ Zwykle szybsza niż podejście typu DOM.
- ▶ Trudny dostęp do dowolnego elementu, utrudniona walidacja (np. jedna część dokumentu odwołująca się do wcześniejszej).



# Parsowanie XML-a

Typowe zdarzenia SAX:

- ▶ Element start.
- ▶ Element end.
- ▶ Text node.
- ▶ Processing instruction.
- ▶ Comments.



## Document Object Model (DOM)

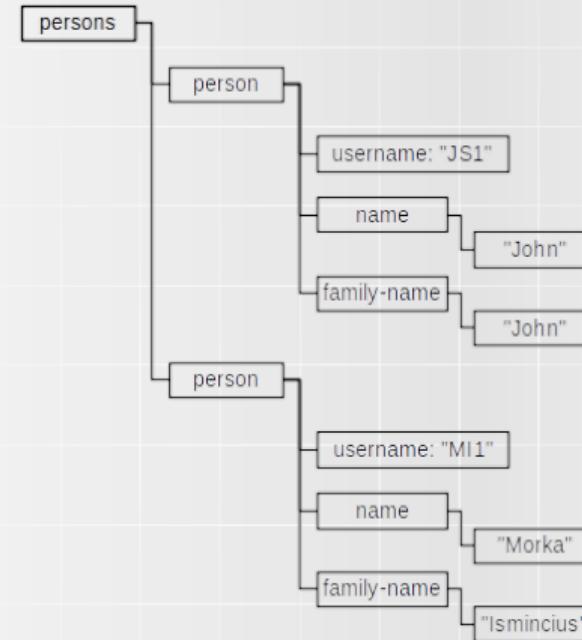
- ▶ Parser buduje drzewo na podstawie dokumentu.
- ▶ Elementy XML stają się węzłami DOM (tekst plus węzły potomne).
- ▶ Dzieci elementu XML stają się dziećmi węzła DOM.
- ▶ Zawartość tekstowa elementu XML staje się węzłem-liściem tekstowym, którego rodzicem jest węzeł DOM.
- ▶ Atrybuty elementu stają się węzłami-liściem potomnymi węzła DOM i są parą klucz-wartość.



Przykładowy XML:

```
<?xml version="1.0" ?>
<persons>
    <person username="JS1">
        <name>John</name>
        <family-name>Smith</family-name>
    </person>
    <person username="MI1">
        <name>Morka</name>
        <family-name>Ismincius</family-name>
    </person>
</persons>
```

DOM odpowiadający przykładowi:





- ▶ Możliwość łatwego znalezienia dowolnego elementu (przejrzenie drzewa lub przejście zadaną ścieżką).
- ▶ Łatwa modyfikacja dokumentu (dodawanie, usuwanie, podmiana elementów i/lub atrybutów).
- ▶ Większe zużycie pamięci (konieczność przechowywania całego drzewa).
- ▶ Wiele implementacji np. libxml2 w C (plus wrappery dla wielu języków), JAXP dla Javy.



- ▶ Streaming API for XML (StAX).
  - ▶ Parser przechowuje pozycję kurSORA, którą można przesuwać do przodu analizując kolejne fragmenty dokumentu (strumieniowo).
  - ▶ Podejście typu „pull” pomiędzy podejściem typu „push” (SAX) i podejściem drzewa (DOM).
- ▶ XML data binding.
  - ▶ Definiuje się mapowanie pomiędzy fragmentami dokumentu (XML) a obiektami i ich polami (programowanie obiektowe).
  - ▶ Dostęp do informacji z XML-a poprzez obiekty.



Extensible Stylesheet Language Transformations (XSLT):

- ▶ Język do transformacji dokumentów XML w inne dokumenty (w tym inne dokumenty XML).
  - ▶ Ogólnie wejściem może być dowolny dokument, z którego można zbudować model XPath (lub XQuery).
- ▶ Turing-zupełny język deklaratywny (w przeciwieństwie do języków imperatywnych typu C++ czy python) oparty o dopasowywanie wzorców.
- ▶ Parser na wejściu otrzymuje oryginalny dokument XML oraz plik XSLT. Na wyjściu otrzymujemy nowy dokument (oryginał nie ulega zmianie).
- ▶ Wykorzystuje XPath do wybierania i filtrowania elementów w drzewie XML.



## Przykładowy XLST:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
    <xsl:output method="xml" indent="yes"/>

    <xsl:template match="/persons">
        <root>
            <xsl:apply-templates select="person" />
        </root>
    </xsl:template>

    <xsl:template match="person">
        <name username="{@username}">
            <xsl:value-of select="name" />
        </name>
    </xsl:template>

</xsl:stylesheet>
```



Przykładowy wejściowy XML:

```
<?xml version="1.0" ?>
<persons>
    <person username="JS1">
        <name>John</name>
        <family-name>Smith</family-name>
    </person>
    <person username="MI1">
        <name>Morka</name>
        <family-name>Ismincius</family-name>
    </person>
</persons>
```



Wyjściowy dokument dla powyższego przykładu:

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
    <name username="JS1">John</name>
    <name username="MI1">Morka</name>
</root>
```



- ▶ Krytyka XML-a obejmuje jego złożoność, rozwlekłość i nadmiarowość.
- ▶ Mapowanie systemów takich jak języki programowania czy bazy danych na XML bywa trudne, ale nie do tego XML został stworzony.
- ▶ JSON opisywany jest jako bardziej zwięzła alternatywa dla XML-a.
- ▶ JSON ma jednak nieco inne zastosowanie (reprezentacja ustrukturyzowanych danych, a nie dokumentów) i ma znacznie mniej możliwości (mniej typów danych, brak komentarzy itd.).



Wrocław  
University  
of Science  
and Technology

# Struktury danych

Wykład 2

## Struktury danych i złożoność obliczeniowa

dr inż. Jarosław Rudy





# Typ danych

## Typ danych

Określony zbiór wartości wraz z określonymi operacjami, które można wykonywać na tych wartościach.

Proste przykłady:

- ▶ Przedział liczb całkowitych wraz z operacjami dodawania, odejmowania, mnożenia i dzielenia (całkowitoliczbowego).
- ▶ Przedział liczb rzeczywistych wraz z operacjami dodawania, odejmowania, mnożenia i dzielenia. Problem zaokrąglania.
- ▶ Przedział liczb naturalnych z operacjami dodawania, odejmowania, mnożenia i dzielenia (całkowitoliczbowego) w systemie z resztą.



# Abstrakcyjny typ danych

## Abstrakcyjny typ danych (ADT)

Opis typu danych uwzględniający własności wartości i wykonywanych na nich operacji bez uwzględnienia (określenia) szczegółów reprezentacji danych i implementacji operacji.

- ▶ ADT opisuje wymagane własności operacji, ale nie sposób ich zapewniania (definiuje interfejs typu danych).
- ▶ ADT opisuje logiczny aspekt danych.
- ▶ ADT powinien dokładnie opisywać wszystkie istotne przypadki.
- ▶ Przykładem ADT jest lista.



# Struktura danych

## Struktura danych (SD)

Opis (format) danych, określający sposób ich reprezentacji, przechowywania i zarządzania nimi.

- ▶ SD stanowi konkretną realizację pewnego ADT (definiuje implementację typu danych).
- ▶ SD opisuje fizyczny aspekt danych, zależny od konkretnej platformy (oprogramowania i sprzętu).
- ▶ Przykładami SD implementującymi listę (ADT) są tablica dynamiczna i lista wiązana.
- ▶ SD zasadniczo dotyczy danych przechowywanych w pamięci operacyjnej (RAM).



# Struktura danych a ADT

- ▶ Czy struktura zbiorów rozłącznych jest strukturą danych czy ADT?
- ▶ Podział na struktury danych i ADT może być płynny.
- ▶ Lista lub słownik opisują tylko operacje i podstawowe własności, są więc ADT.
- ▶ Lista wiązana i tablica dynamiczna opisują konkretną implementację i strukturę w pamięci wykorzystując cechy języka programowania, są więc strukturami danych.
- ▶ Niektóre typy (np. kopiec, drzewo, graf, binarne drzewo poszukiwań) mogą być traktowane zarówno jako ADT jak i struktura danych, zależnie od kontekstu.
- ▶ Istotne jest co z przyjęcia danego ADT/SD wynika oraz które ADT/SD może być użyte do (wydajnej) implementacji których ADT/SD.



# Typowe operacje na ADT/SD

- ▶ Stwierdzenie czy kolekcja jest pusta.
- ▶ Zwrócenie liczby elementów w kolekcji.
- ▶ Wyszukanie elementu:
  - ▶ po pozycji,
  - ▶ po kluczu,
  - ▶ po wartości.
- ▶ Dodanie elementu (też wg pozycji).
- ▶ Usunięcie elementu (wg pozycji, klucza, wartości).
- ▶ Przejrzenie (przejście) przez kolekcję.



## Typy danych – przykłady (1)

Proste typy danych z języków programowania, niskopoziomowo zawsze są reprezentowane za pomocą bitów.

- ▶ Typy stałoprzecinkowe.
  - ▶ Typy liczbowe, z reguły całkowitoliczbowe.
  - ▶ Ze znakiem lub bez.
  - ▶ Reprezentacja w naturalnym kodzie binarnym lub kodzie uzupełnieniowym do 2.
  - ▶ Względnie wysoka precyzja i dokładność wyniku.
  - ▶ Względnie niski zakres wartości, problem nadmiaru.
  - ▶ Dzielenie „całkowitoliczbowe”.
  - ▶ Typy z C/C++: int, long int, unsigned int, short int, czasami też char i boolean.



## Typy danych – przykłady (2)

- ▶ Typy zmiennoprzecinkowe.
  - ▶ Typy liczbowe, odzwierciedlenie liczb rzeczywistych.
  - ▶ Ze znakiem.
  - ▶ Reprezentacja z użyciem 3 liczb binarnych: znaku, ułamka z zakresu [0, 1) i przesuniętego wykładnika.
  - ▶ Duży zakres wartości, dostateczna precyzja.
  - ▶ Dzielenie „rzeczywistoliczbowe” .
  - ▶ Ograniczona dokładność (błąd reprezentacji, zaokrąglenia), nadmiar, niedomiar, NaN, wyjątki, brak łączności.
  - ▶ Typy z C/C++: float, double.



## Typy danych – przykłady (3)

- ▶ Big numbers.
  - ▶ python domyślnie wspiera dowolnie duże liczby całkowite (np. zapis liczb w systemie o podstawie  $2^{30}$  przy użyciu tablicy).
  - ▶ Biblioteka BigNumbers pythona zapewnia dowolnie duże liczby dowolnej precyzji.
- ▶ Typ znakowy (char w C/C++).
  - ▶ Łańcuchy znaków (string).
    - ▶ Realizowane jako tablica.
    - ▶ Ograniczony rozmiar (przechowywany w zerowym elemencie tablicy), stosowane w Pascalu.
    - ▶ Null-terminated string (ASCIIZ, C string). Dowolna długość, łańcuch kończy pierwszy znak null. Powolne liczenie długości.



## Typy danych – przykłady (4)

- ▶ Typ tablicowy (tablice statyczne).
  - ▶ Typ złożony, przechowuje wiele elementów (zwykle identycznego typu) zajmujących ciągły obszar pamięci.
  - ▶ Dostęp przez indeks (arytmetyka wskaźników).
  - ▶ Ustalony rozmiar.
- ▶ Typ strukturalny.
  - ▶ Typ złożony, przechowuje wiele elementów (zajmując ciągły obszar pamięci), elementy mogą być różnego typu.
  - ▶ Dostęp poprzez nazwę elementu.
  - ▶ Typy zawarte w strukturze często są niezmienne (w przeciwieństwie do wartości).
  - ▶ Definiowany przez użytkownika.



## Typy danych – przykłady (5)

- ▶ Typ obiektowy.
  - ▶ Typ złożony rozszerzający typ strukturalny o definicje dozwolonych operacji.
  - ▶ Definiowany przez użytkownika.
  - ▶ Enkapsuluje własny kompletny typ danych (zbiór wartości powiązany z operacjami).
  - ▶ Jeden z paradymatów programowania obiektowego w wielu językach (C++, python, Java, Javascript, C# itd.).
  - ▶ Stanowi podstawę do tworzenia ADT i SD niewspieranych natywnie przez język.



## ADT – przykłady (1)

- ▶ Kolekcja (collection) i kontener (container) – ogólna grupa elementów ze sposobem przechowywania i dostępu.
- ▶ Lista (list) – zachowuje kolejność elementów (kolekcja liniowa). Elementy mogą się powtarzać. Dowolne wstawianie i usuwanie elementów.
- ▶ Zbiór (set) – brak kolejności, elementy nie mogą się powtarzać. Operacje z teorii zbiorów.
- ▶ Multizbiór (multiset) – odpowiednik zbioru, w którym elementy mogą się powtarzać.



## ADT – przykłady (2)

- ▶ Kolejka (queue) – zachowuje kolejność, operacje możliwe tylko na końcach kolejki.
  - ▶ Kolejka First In-First Out (FIFO) – dodawanie na jednym końcu, usuwanie z drugiego końca.
  - ▶ Kolejka Last In-First Out (LIFO), stos – dodawanie i usuwanie elementów na tym samym końcu.
  - ▶ Kolejka dwustronna (double-ended queue, deque) – dodawanie i usuwanie na obu końcach.
  - ▶ Kolejka priorytetowa – kolejność definiowana przez priorytet elementu (elementy o wyższym priorytecie są usuwane pierwsze).



## ADT – przykłady (3)

- ▶ Mapa (map), tablica asocjacyjna (associative table) lub słownik (dictionary)  
– przechowuje elementy w postaci pary klucz-wartość. Klucze nie mogą się powtarzać (klucz ma co najwyżej jedną wartość).
- ▶ Multimapa, multisłownik – odpowiednik mapy, w którym klucze mogą się powtarzać (klucz może mieć wiele wartości).
- ▶ Drzewo (tree) – każdy element ma jeden element rodzica (z wyjątkiem tzw. korzenia), każdy element może mieć potomków (dowolną liczbę).
- ▶ Graf (graph) – dowolne połączenia pomiędzy elementami, z lub bez określania kierunków.



# Struktury danych – przykłady

- ▶ Tablica dynamiczna – tablica o zmiennej liczbie elementów zajmujących ciągły obszar w pamięci. Często stosowana do implementacji wielu ADT (lista, stos, kolejka, drzewo binarne). Dostęp indeksowy (swobodny).
- ▶ Lista wiązana – elementy nie muszą zajmować ciągłego obszaru pamięci. Narzut pamięci. Dostęp sekwencyjny. Używana do implementacji ADT takich jak lista, stos czy kolejka.
- ▶ Tablica mieszająca (hash table) – korzysta z funkcji mieszających. Używana do implementacji słownika.
- ▶ Binarne drzewa poszukiwań, drzewa czerwono-czarne, drzewa AVL – używane do implementacji słowników.
- ▶ Macierz sąsiedztwa, lista sąsiedztwa, lista krawędzi – struktury danych używane do reprezentacji grafów.



# Złożoność obliczeniowa (1)

## Złożoność obliczeniowa

Ilość zasobów potrzebnych algorytmowi do poprawnej pracy.

- ▶ Złożoność obliczeniową określa się dla konkretnego problemu oraz konkretnego algorytmu, programu lub operacji.
- ▶ Najczęściej interesuje nas pojedyncza operacja ADT/struktury danych, ale możliwe jest rozważanie ciągu wielu operacji.
- ▶ Zwykle rozważa się albo konkretny rozmiar problemu (np. konkretny rozmiar listy, drzewa itd.), albo określa się złożoność jako funkcję, której argumentem jest rozmiar problemu.
  - ▶ Ile trwa wyszukanie elementu w liście  $N$  elementów?



## Złożoność obliczeniowa (2)

Zasoby rozpatrywane w złożoności obliczeniowej:

- ▶ Złożoność czasowa.
- ▶ Złożoność pamięciowa.
- ▶ Złożoność komunikacji.
- ▶ Złożoność „równoległa” (liczba procesorów).



# Złożoność czasowa

## Złożoność (obliczeniowa) czasowa

Czas potrzebny do zakończenia algorytmu.

- ▶ Często szacowana liczbą elementarnych operacji.
- ▶ W praktyce występują nie tylko operacje elementarne.
- ▶ Różne operacje zajmują różny czas na różnych maszynach.
- ▶ W praktyce ilość czas rzeczywistego (wall time) potrzebnego do zakończenia algorytmu.
  - ▶ Także CPU time, problem pamięci podręcznej, narzut systemu operacyjnego itd.
  - ▶ W niektórych sytuacjach – liczba operacji odczytu/zapisu pamięci operacyjnej.



# Złożoność pamięciowa (1)

Złożoność (obliczeniowa) pamięciowa

Ilość pamięci (liczba komórek, bajtów itp.) potrzebnych do zakończenia algorytmu.

- ▶ Pamięć wejścia – pamięć potrzebna do zapisania danych wejściowych.
- ▶ Pamięć pomocnicza (dodatkowa) – pamięć potrzebna do zakończenia algorytmu z pominięciem pamięci wejścia.
- ▶ W praktyce często podaje się jedynie pamięć pomocniczą.



## Złożoność pamięciowa (2)

### Algorytm działający w miejscu (in-place)

Algorytm który nie potrzebuje pamięci pomocniczej proporcjonalnej do wielkości danych wejściowych.

Pojęcie nieścisłe, różne definicje praktyczne:

- ▶ Algorytm którego pamięć pomocnicza ma rozmiar co najwyżej stały (tj.  $O(1)$ ).
- ▶ Algorytm którego pamięć pomocnicza ma rozmiar co najwyżej logarytmiczny (tj.  $O(\log N)$ ).
- ▶ Czasami dopuszcza się dodatkową pamięć  $O(N)$ , o ile nie służy do przetwarzania wejścia.



## Scenariusze złożoności (1)

Algorytmy mają różną złożoność dla różnych danych wejściowych. Dla uproszczenia analizy, często zakłada się konkretne scenariusze (przypadki).

- ▶ Przypadek pesymistyczny (worst-case) – dane dla których rozpatrywana złożoność jest największa.
- ▶ Bardzo często wykorzystywany.
- ▶ Gwarancja złożoności.
- ▶ Ograniczona użyteczność (ekstremalne przypadki są zwykle rzadkie).
- ▶ Jeśli nie można ustalić, to można zastąpić górnym ograniczeniem.



## Scenariusze złożoności (2)

- ▶ Przypadek średni/typowy (average-case) – przypadek który jest wartością oczekiwana dla całego rozkładu prawdopodobieństwa możliwych danych wejściowych.
  - ▶ Dość często wykorzystywany i użyteczny.
  - ▶ Często liczba zestawów danych wejściowych jest nieograniczona (nieskończona) – jak wyznaczyć?
- ▶ Przypadek optymistyczny (best-case) – dane dla których rozpatrywana złożoność jest najmniejsza.
  - ▶ Prawie nieużywny (niewielka użyteczność).
  - ▶ Jeśli nie można ustalić, to można zastąpić dolnym ograniczeniem.
- ▶ Inne – np. 95-ty percentyl.



## Asymptotyczne tempo wzrostu

- ▶ Konieczność porównywania złożoności algorytmów dla różnego rozmiaru danych wejściowych.
- ▶ Często trudno jest ustalić dokładną złożoność (np.  $15n^2 + 123n - 64$ ).
- ▶ Dla dużych  $n$  współczynniki (15, 123 i 64 powyżej) często nieistotne.
- ▶ Dla dużych  $n$  mniejsze miany ( $123n$  lub 64) często nieistotne przy większych ( $15n^2$ ).

Powyższe doprowadziło do pojęć asymptotycznego tempa wzrostu i rzędu funkcji złożoności obliczeniowej. Asymptotyczność oznacza, że rozpatruje się jak funkcja zachowuje się gdy  $n \rightarrow \infty$ .



# Notacja dużego O (1)

## Notacja dużego O

Dane są funkcje  $f(x)$  oraz  $g(x)$ . Mówimy, że  $f$  jest rzędu co najwyżej  $g$ , co zapisujemy  $f \in O(g)$ , wtedy i tylko wtedy gdy:

$$\exists_{c \in \mathbb{R} \setminus \{0\}} \quad \exists_{x_0 \in \mathbb{R}} \quad \forall_{x \geqslant x_0} : \quad f(x) \leqslant c \cdot g(x). \quad (1)$$

- ▶ Czyli możemy znaleźć taki (wspólny) ogon obu funkcji (od  $x_0$  „w prawo”) i taką skalę funkcji  $g$ , że w tym ogonie zawsze  $f(x)$  jest nie większa niż przeskalowane  $g(x)$ .
- ▶ Równoważnie można w definicji zapisać  $c \cdot f(x) \leqslant g(x)$ . Skala po prostu się odwraca (np. z  $\frac{7}{2}$  na  $\frac{2}{7}$ ).
- ▶ Często zamiast  $f \in O(g)$  zapisuje się  $f = O(g)$ , ale jest to mylące i nie ma nic wspólnego z matematyczną równością! Z faktu że  $f \in O(g)$  niekoniecznie wynika że  $g \in O(f)$ !



## Notacja dużego O (2)

- ▶  $100n^2 \in O(n^2)$  – współczynnik bez znaczenia.
- ▶  $n^5 + n^3 + n - 5 \in O(n^5)$  – mniejsze miany nie mają znaczenia.
- ▶  $n^2 \in O(n^3)$  – kwadrat jest co najwyżej sześcianem.
- ▶  $n^2 \in O(2^n)$  – pomimo, że na odcinku  $(2, 4)$  to  $n^2$  jest większe.
- ▶  $n^3 \notin O(n^2)$  – jakikolwiek ogon i skalę dobierzemy, sześcian w końcu przegoni kwadrat.



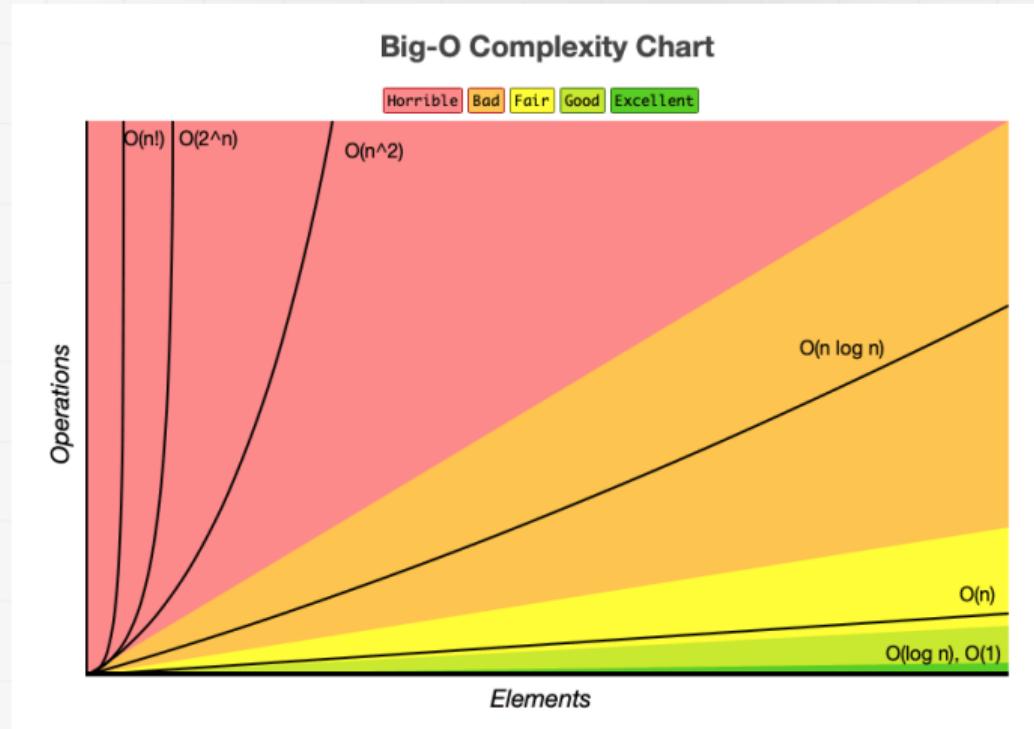
# Istotne rzędy złożoności (1)

Niektórzy rzędy złożoności obliczeniowej (w kolejności rosnącej):

- ▶  $O(1)$  – złożoność stała (tj. ograniczona z góry przez stałą).
- ▶  $O(\log n)$  – złożoność logarytmiczna, dla dużego  $n$  traktowana jak  $O(1)$ .
- ▶  $O(n)$  – liniowa.
- ▶  $O(n^2)$  – kwadratowa. Dla niektórych problemów uznawana za szybką, dla niektórych (np. sortowanie) za powolną.
- ▶  $O(n^3)$  – sześciian.
- ▶  $O(2^n)$  – wykładnicza, zwykle nieakceptowalnie powolna.
- ▶  $O(n!)$  – silnia.



# Istotne rzędy złożoności (2)



Source: [1]



## Duże O – niuanse i praktyka (1)

- ▶ Fakt, że duże O oznacza „co najwyżej” (ograniczenie z góry), nie znaczy, że służy tylko i wyłącznie do opisu najgorszego przypadku!
- ▶ Zadanie: w grupie  $n$  studentów znaleźć jednego nieobecnego.
- ▶ Optymistycznie potrzeba 1 sprawdzenia, średnio  $\frac{1+n}{2}$ , pesymistycznie  $n$ .
- ▶ Każdy z tych scenariuszy można ograniczyć od góry, odpowiednio  $O(1)$ ,  $O(n)$  oraz  $O(n)$ .
- ▶ Każdy można też ograniczyć poprzez np.  $O(n^2)$ , ale zwykle interesuje nas możliwie dokładne ograniczenie.



## Duże O – niuanse i praktyka (2)

- ▶ Czasami dla rzeczywistych rozmiarów danych algorytm o złożoności wyższego rzędu jest praktyczniejszy. Przykładowo:
  - ▶ Złożoność  $2^n$  kontra  $1\,000\,000n^2$ .
  - ▶ Złożoność  $2^n$  kontra  $n^{10}$ .
- ▶ Arytmetyka dużego O (przykłady):
  - ▶  $O(n) + O(n^2) = O(n + n^2) = O(n^2)$ .
  - ▶  $10n^2 - O(n) + 15 = O(n^2) - O(n) + O(1) = O(n^2 - n + 1) = O(n^2)$
  - ▶  $10n^2 - O(n) + 15 = O(10n^2 - n + 15) = O(n^2)$ .
  - ▶  $15n \cdot O(n^3) = O(15n^4) = O(n^4)$ .
  - ▶  $O(\log n) \cdot O(n^2) = O(n^2 \log n)$ .
  - ▶  $(3n + 1)^2 = 9n^2 + O(n)$ .



## Inne notacje

Wszystkie notacje zakładają, że rozważamy odpowiedni ogon funkcji i odpowiednie skale.

- ▶  $f \in O(g)$ , duże O –  $f$  jest co najwyżej rzędu  $g$ .
- ▶  $f \in \Omega(g)$ , duże omega –  $f$  jest co najmniej rzędu  $g$ .
- ▶  $f \in \Theta(g)$ , teta –  $f$  jest rzędu (dokładnie)  $g$ . Implikuje  $f \in O(g)$  oraz  $f \in \Omega(g)$ .
- ▶  $f \in o(g)$ , małe o –  $f$  jest rzędu niższego niż  $g$ . Implikuje  $f \in O(g)$ .
- ▶  $f \in \omega(g)$ , małe omega –  $f$  jest rzędu wyższego niż  $g$ . Implikuje  $f \in \Omega(g)$ .



## Koszt zamortyzowany

- ▶ Różne definicje. Najczęściej wychodzi się od pesymistycznego sumarycznego kosztu ciągu  $k$  operacji (tych samych lub różnych).
- ▶ Z otrzymanej sumy wyciągamy średnią.
- ▶ Ponieważ zakładamy pesymizm każdej operacji, to wynik nie jest tym samym co analiza średniego przypadku (nie opiera się na probabilistyczne).
- ▶ Koszt zamortyzowany pojedynczej operacji w ciągu może być różny (mniejszy) niż koszt tej samej operacji w tym samym ciągu obliczony z użyciem analizy najgorszego przypadku!



## Koszt zamortyzowany – przykład

- ▶ Mamy zakład z  $n$  maszynami. Uruchamiane są razem jednym przyciskiem (jedna czynność), ale raz na  $n$  uruchomień trzeba je najpierw pojedynczo przeglądać ( $n + 1$  czynności).
- ▶ Pesymistyczny czas pojedynczej operacji uruchomienia wynosi więc  $O(n)$ .
- ▶ Rozpatrzmy koszt pojedynczej operacji w ciągu  $n$  kolejnych uruchomień:
  - ▶ Klasyczna analiza najgorszego przypadku:

$$\frac{n \cdot O(n)}{n} = O\left(\frac{n^2}{n}\right) = O(n). \quad (2)$$

- ▶ Koszt zamortyzowany:

$$\overbrace{\frac{1+1+\dots+1+n+1}{n}}^{\text{n-1 razy}} = O\left(\frac{2n}{n}\right) = O(1). \quad (3)$$



# Bibliografia



<https://blog.teclado.com/time-complexity-big-o-notation-python/>



Wrocław  
University  
of Science  
and Technology

# Struktury danych

Wykład 3  
Listy

dr inż. Jarosław Rudy





# Lista (ADT)

- ▶ Lista jest kontenerem przechowującym elementy.
- ▶ Elementy mogą być identycznego lub różnego typu.
- ▶ Lista zachowuje (określa) kolejność elementów.
- ▶ Elementy mogą się powtarzać.
- ▶ Jeden element ma jedną wartość (ale tą wartością może być kolejna kolekcja).
- ▶ Lista ma zmienną długość i zawartość (elementy można dodawać, usuwać i modyfikować).



# Lista – operacje (1)

Na liście typowo rozważa się następujące operacje:

- ▶ Stworzenie pustej listy.
- ▶ Dostęp (zwrócenie) elementu na pozycji  $i$ .
- ▶ Dodanie elementu  $e$  na pozycji  $i$  (tj. przed elementem  $i$ -tym). Specjalne przypadki:
  - ▶ Dodanie elementu  $e$  na początek listy.
  - ▶ Dodanie elementu  $e$  na koniec listy.



## Lista – operacje (2)

- ▶ Usunięcie elementu  $e$  na pozycji  $i$ . Specjalne przypadki:
  - ▶ Dodanie elementu  $e$  na początku listy.
  - ▶ Dodanie elementu  $e$  na końcu listy.
- ▶ Zwrócenie rozmiaru (liczby elementów) listy.
- ▶ Sprawdzenie czy lista jest pusta.
- ▶ Wyszukanie elementu  $e$ .

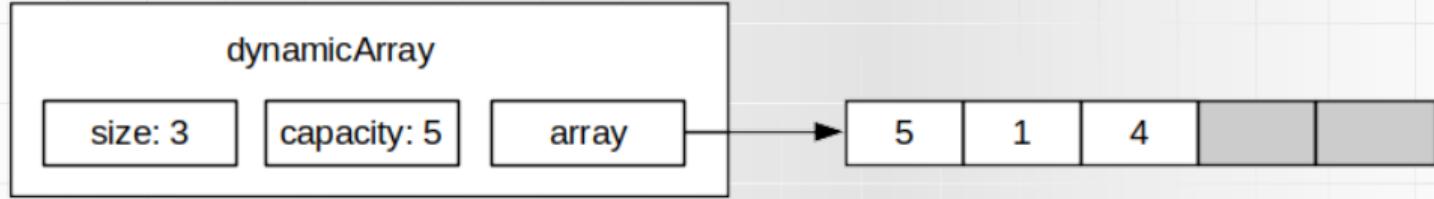


# Tablica dynamiczna (1)

- ▶ Struktura danych będąca implementacją listy (ADT).
- ▶ Najczęściej przechowuje:
  - ▶ array – wskaźnik na tablicę utworzoną dynamicznie (`new`, `malloc()` itp.).
  - ▶ capacity – rozmiar tablicy `array` (w liczbie elementów).
  - ▶ size – liczba przechowywanych elementów.
- ▶ Zajmowana pamięć:  $capacity + 3$ , czyli  $O(capacity)$ .
  - ▶ W praktyce  $capacity < 2n$ , więc zajmowany rozmiar to pomiędzy  $n + 3$  a  $2n + 3$ , czyli  $O(n)$ .



## Tablica dynamiczna (2)



„Pierwsza” implementacja operacji:

- ▶ Stworzenie pustej listy:

- ▶  $array \leftarrow null$
- ▶  $size \leftarrow 0$
- ▶  $capacity \leftarrow 0$
- ▶ czas:  $O(1)$ .



## Tablica dynamiczna (3)

- ▶ Zwrócenie rozmiaru listy:
  - ▶ Zwrócenie `size`.
  - ▶ czas:  $O(1)$ .
- ▶ Sprawdzenie czy lista jest pusta:
  - ▶ Zwrócenie wartości wyrażenia `size == 0`.
  - ▶ czas:  $O(1)$ .
- ▶ Zwrócenie elementu na pozycji  $i$  – w czasie  $O(1)$  dzięki arytmetyce wskaźników (tablica `array` ma dostęp swobodny przez indeks  $i$ ).
  - ▶ Błąd jeśli  $i < 0$  lub  $i \geq size$ .



## Tablica dynamiczna (4)

- ▶ Wyszukanie elementu  $e$ .
  - ▶ Sprawdzamy kolejne elementy, porównując je z  $e$  do czasu znalezienia pasującego elementu (zwrócenie  $e$ , prawdy itp.) lub do wyczerpania elementów (zwrócenie nulla, fałszu itp.).
  - ▶ Optymistycznie: sprawdzany jeden element – czas  $O(1)$ .
  - ▶ Średnio: sprawdzamy połowę elementów tj.  $\lceil \frac{n}{2} \rceil$  – czas  $O(n)$ .
  - ▶ Pesymistycznie: sprawdzamy wszystkie  $n$  elementów – czas  $O(n)$ .
- ▶ Przejrzenie (np. wypisanie wszystkich elementów) również zajmuje czas  $O(n)$ .



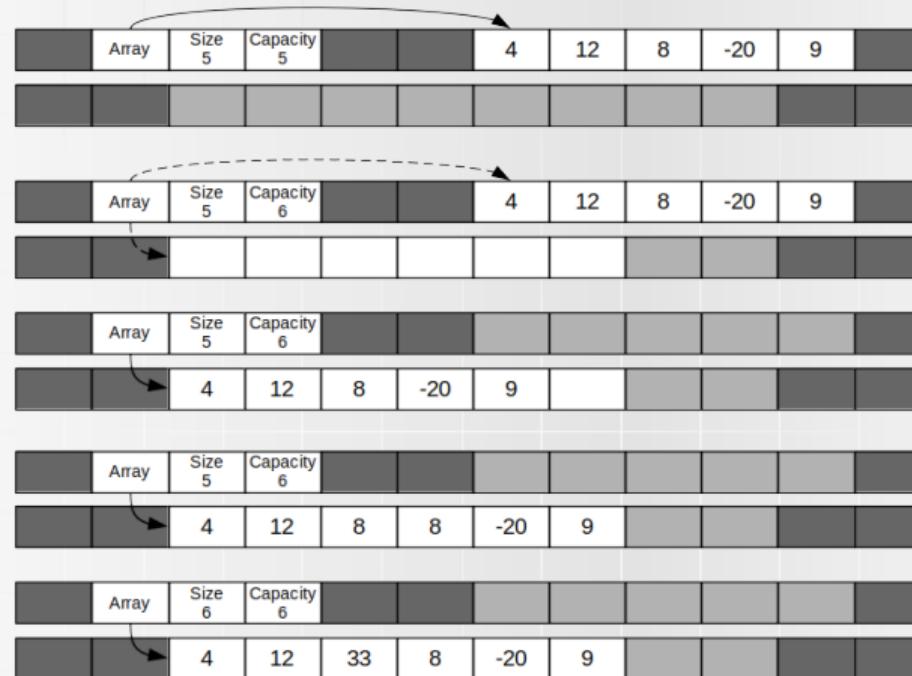
## Tablica dynamiczna (5)

W najprostszej implementacji dodanie elementu  $e$  na pozycji  $i$  składa się z kilku etapów:

- ▶ Zwiększenie rozmiaru tablicy  $array$  o 1.
  - ▶ Funkcja `realloc()` zmienia rozmiar tablicy i może wymagać przeniesienia tablicy do nowej lokalizacji (`memcpy()`).
  - ▶  $capacity \leftarrow capacity + 1$
  - ▶ Czas  $O(1)$  lub  $O(n)$ , zależnie czy kopiowano tablicę.
- ▶ Przeniesienie (np. w pętli) elementów od pozycji  $n - 1$  do  $i$  o jeden w prawo.
  - ▶ Czas wynosi  $O(n - i)$ , pesymistycznie  $O(n)$ .
- ▶ Wstawienie  $e$  na pozycję  $i$  w czasie  $O(1)$ :
  - ▶  $array[i] \leftarrow e$
  - ▶  $size \leftarrow size + 1$
- ▶ Czas całej operacji pesymistycznie i średnio  $O(n)$ , optymistycznie  $O(1)$ .

# Tablica dynamiczna (6)

Przykładowe wstawienie ( $e = 33, i = 2$ ).





## Tablica dynamiczna (7)

- ▶ Dodanie elementu  $e$  na początku ( $i = 0$ ) jest takie samo, ale czas jest zawsze  $O(n)$ , nawet jeśli nie trzeba przenosić tablicy podczas `realloc()`.
- ▶ Dodanie elementu  $e$  na końcu ( $i = n - 1$ ) jest takie samo, ale czas wynosi  $O(1)$  z wyjątkiem sytuacji, gdy `realloc()` przeniesie tablicę.
- ▶ Powyższa podstawowa implementacja operacji dodawania zakłada, że zaczynamy od pustej tablicy ( $capacity = 0$ ) i zwiększamy rozmiar zawsze o 1 (tzn. zawsze  $size = capacity$ ).



## Tablica dynamiczna (8)

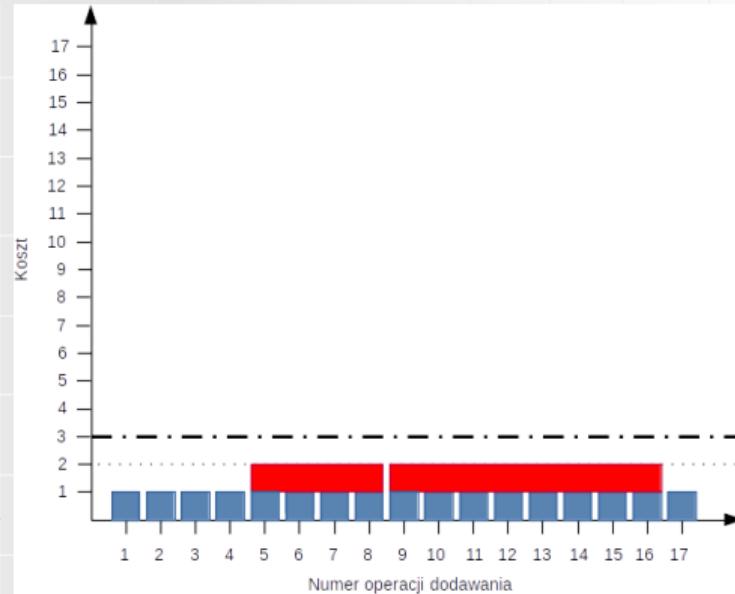
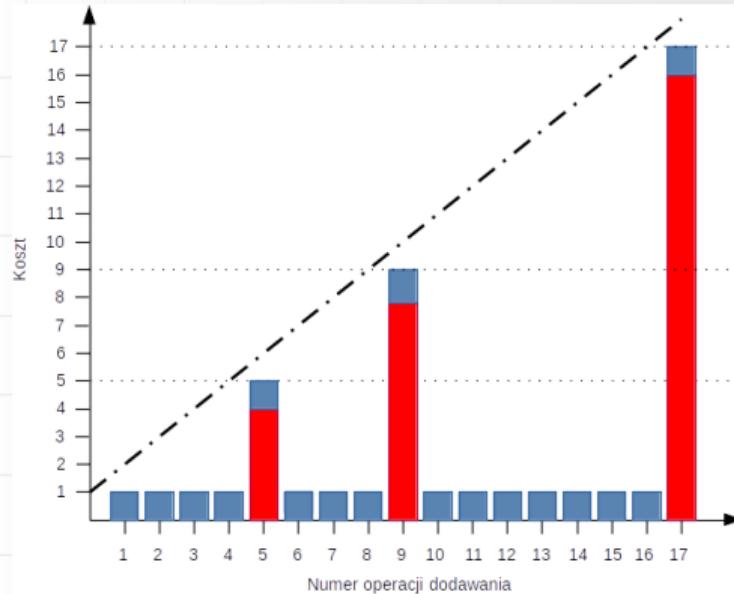
Implementacja ulepszona.

- ▶ Zaczynamy od zaalokowanej, ale pustej tablicy (np.  $size = 0$ ,  $capacity = 4$ ).
- ▶ Przez pierwsze 4 operacje dodawania, nie trzeba będzie kopiować pamięci!
- ▶ Za każdym razem kiedy brakuje miejsca, zwiększamy rozmiar tablicy dwukrotnie ( $capacity = 2size$ ).
- ▶ Podobnie jak poprzednio, przy każdym braku miejsca w tablicy może dojść do konieczności kopowania pamięci... ale brak miejsca zdarza się rzadziej.
- ▶ Ścisłej, jeśli odbywa się kopowanie zajmujące  $O(n)$ , to wiemy, że  $n$  poprzednich operacji dodawania nie wymagało kopowania!
  - ▶ Pesymistycznie jest dalej  $O(n)$ , ale w koszcie zamortyzowanym jest  $O(1)$ !



# Tablica dynamiczna (9)

Klasyczny analiza przypadku pesymistycznego vs koszt zamortyzowany





# Tablica dynamiczna (10)

Klasyczna analiza:

Operacja	Optymistycznie	Średnio	Pesymisycznie
Dodanie na dowolnej pozycji	$O(1)$	$O(n)$	$O(n)$
Dodanie na początku	$O(n)$	$O(n)$	$O(n)$
Dodanie na końcu	$O(1)$	$O(n)$	$O(n)$

Koszt zamortyzowany:

Operacja	Optymistycznie	Średnio	Pesymisycznie
Dodanie na dowolnej pozycji	$O(1)$	$O(n)$	$O(n)$
Dodanie na początku	$O(n)$	$O(n)$	$O(n)$
Dodanie na końcu	$O(1)$	$O(1)$	$O(1)$



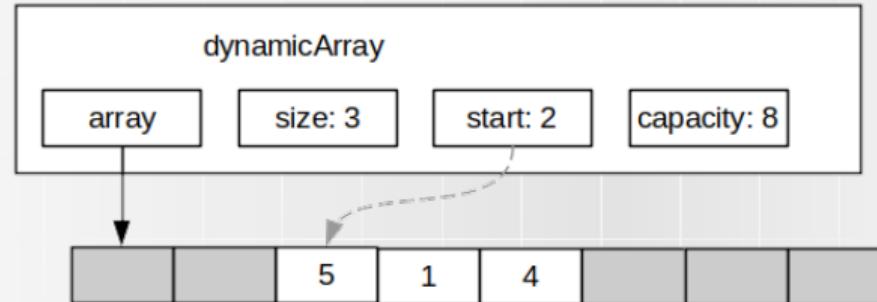
## Tablica dynamiczna (11)

Usuwanie elementu z pozycji  $i$  zasadniczo działa podobnie do dodawania, lecz kolejność jest inna:

- ▶ Skopiowanie elementów od  $i+1$  do  $n-1$  o jedną pozycję w lewo oraz zmniejszenie `size` o 1.
- ▶ Zmniejszenie rozmiaru tablicy (oraz *capacity*). Może następować o 1 lub co jakiś czas (analogicznie do dodawania).
- ▶ Jeśli usuwany element ma zostać zwrócony, należy go zapamiętać przed pierwszym krokiem!
- ▶ Zmniejszanie tablicy dalej może wymagać kopiowania pamięci (zależnie od implementacji `realloc()`).
- ▶ Można pominąć zmniejszanie rozmiaru tablicy, ale rozmiar struktury nie będzie wtedy  $O(n)$ .

## Tablica dynamiczna (12)

- ▶ Czy da się zredukować złożoność  $O(n)$  dla dodawania i usuwania na początku?
- ▶ Zaalokowanie więcej miejsca i przesunięcie indeksu fizycznego względem logicznego – zostaje z przodu miejsce na indeksy „ujemne”.
- ▶ Konieczne dodanie i aktualizacja pozycji początkowej.
- ▶ Alokacja i dealokacja raz na jakiś czas analogicznie jak poprzednio.





# Tablica dynamiczna (13)

Po dodaniu wspomnianego usprawienia:

Operacja	Optymistycznie	Średnio	Pesymisycznie
<code>addFront(e)</code>	$O(1)$	$O(1)$ (amort.)	$O(1)$ (amort.)
<code>removeFront()</code>	$O(1)$	$O(1)$ (amort.)	$O(1)$ (amort.)
<code>addBack(e)</code>	$O(1)$	$O(1)$ (amort.)	$O(1)$ (amort.)
<code>removeBack()</code>	$O(1)$	$O(1)$ (amort.)	$O(1)$ (amort.)
<code>add(e,i)</code>	$O(1)$	$O(n)$	$O(n)$
<code>remove(i)</code>	$O(1)$	$O(n)$	$O(n)$

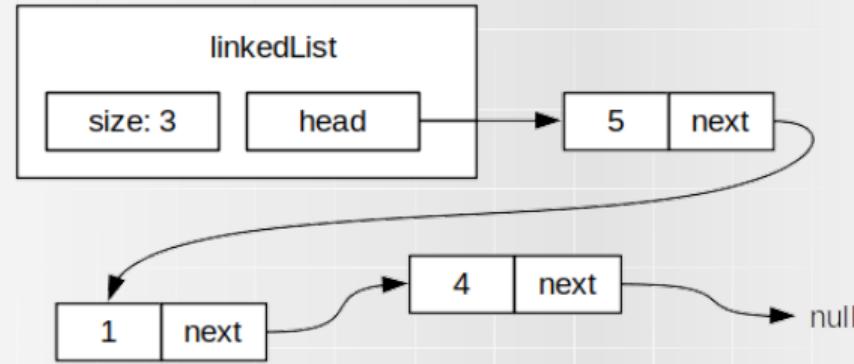


## Lista wiązana (jednokierunkowa)

- ▶ Struktura danych będąca implementacją listy (ADT).
- ▶ Każdy element (węzeł) zawiera:
  - ▶ Właściwą wartość (*value*).
  - ▶ Wskaźnik na kolejny element listy (*next*).
- ▶ Elementy mogą mieć różną lokalizację w pamięci!
- ▶ Lista kończy się, gdy następny wskaźnik ma wartość null.
- ▶ Struktura przechowuje wskaźnik *head* na pierwszy element. Oprócz tego przechowywany jest *size*.
- ▶ Zajmowana pamięć:  $2n + 2$ , czyli  $O(n)$ .



## Lista wiązana (2)



Tworzenie pustej listy (czas  $O(1)$ ):

- ▶  $size \leftarrow 0$
- ▶  $head \leftarrow null$



## Lista wiązana (3)

- ▶ Zwrócenie rozmiaru i sprawdzenie czy lista jest pusta działa identycznie jak dla tablicy dynamicznej.
- ▶ Wyszukanie elementu działa podobnie, lecz by przejść do następnego elementu, należy skorzystać ze wskaźnika *next* poprzedniego (i sprawdzić czy nie jest on null).
- ▶ Zwrócenie elementu na pozycji  $i$  wymaga przejścia przez elementy od 0 do  $i$ . Czas operacji jest więc  $O(i)$ :
  - ▶ Optymistycznie ( $i = 0$ ) mamy  $O(1)$ .
  - ▶ Średnio ( $i = \lceil \frac{n}{n} \rceil$ ) mamy  $O(n)$ .
  - ▶ Pesymistycznie ( $i = n - 1$ ) mamy  $O(n)$ .



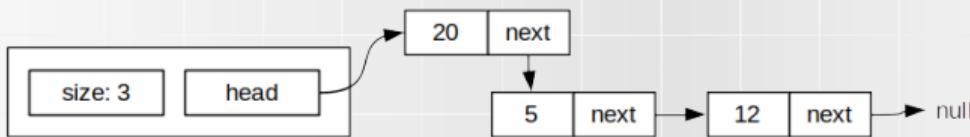
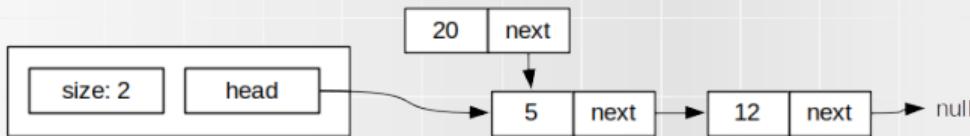
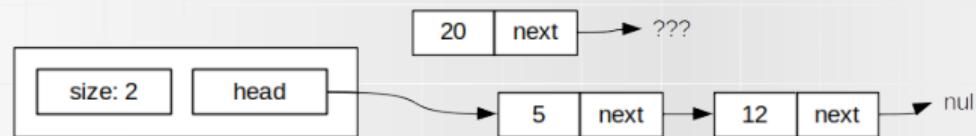
## Lista wiązana (4)

Dodanie elementu  $e$  na początek ( $O(1)$ )

- ▶ Tworzymy (dynamicznie, `new`, `malloc()` itp.) nowy węzeł i zapamiętujemy jego wskaźnik (*node*).
- ▶  $node.value \leftarrow e$
- ▶  $node.next \leftarrow head$
- ▶  $head \leftarrow node$
- ▶  $size \leftarrow size + 1$



# Lista wiązana (5)





## Lista wiązana (6)

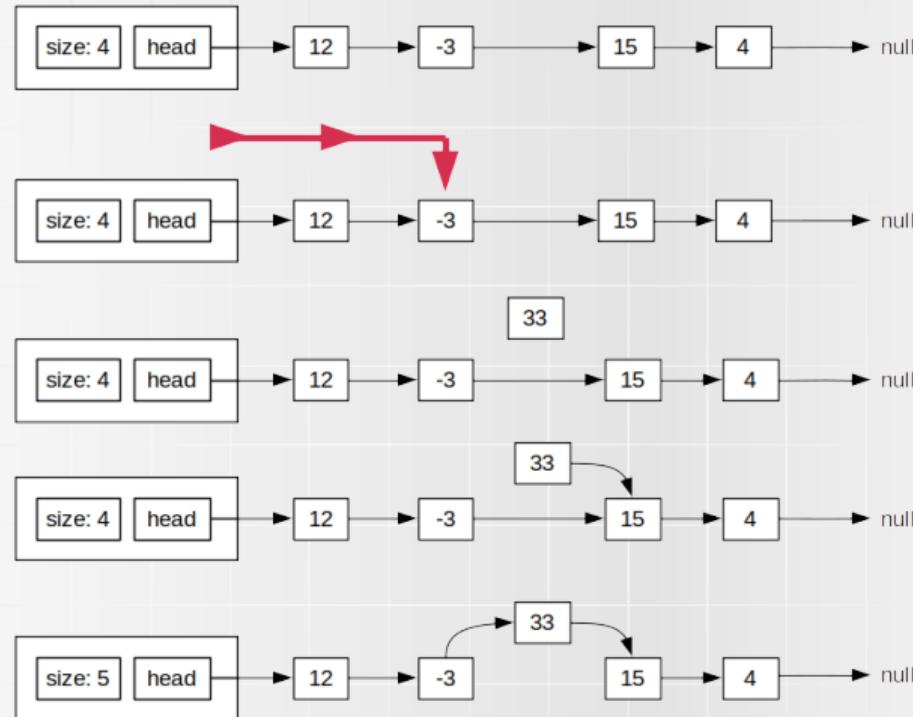
Dodanie elementu  $e$  na pozycji  $i$  (innej niż  $i = 0$  oraz  $i = n - 1$ ).

- ▶ Najpierw musimy dotrzeć do węzła  $i - 1$  (czas  $O(i)$ ), nazwijmy go  $old$ .
- ▶ Tworzymy nowy węzeł  $node$  i przypisujemy mu wartość  $e$ .
- ▶  $node.next \leftarrow old.next$
- ▶  $old.next \leftarrow node$
- ▶  $size \leftarrow size + 1$
- ▶ Optymistycznie  $O(1)$ , średnio i pesymistycznie  $O(n)$ .



# Lista wiązana (7)

Dodanie  $e = 33$  na pozycję 2:





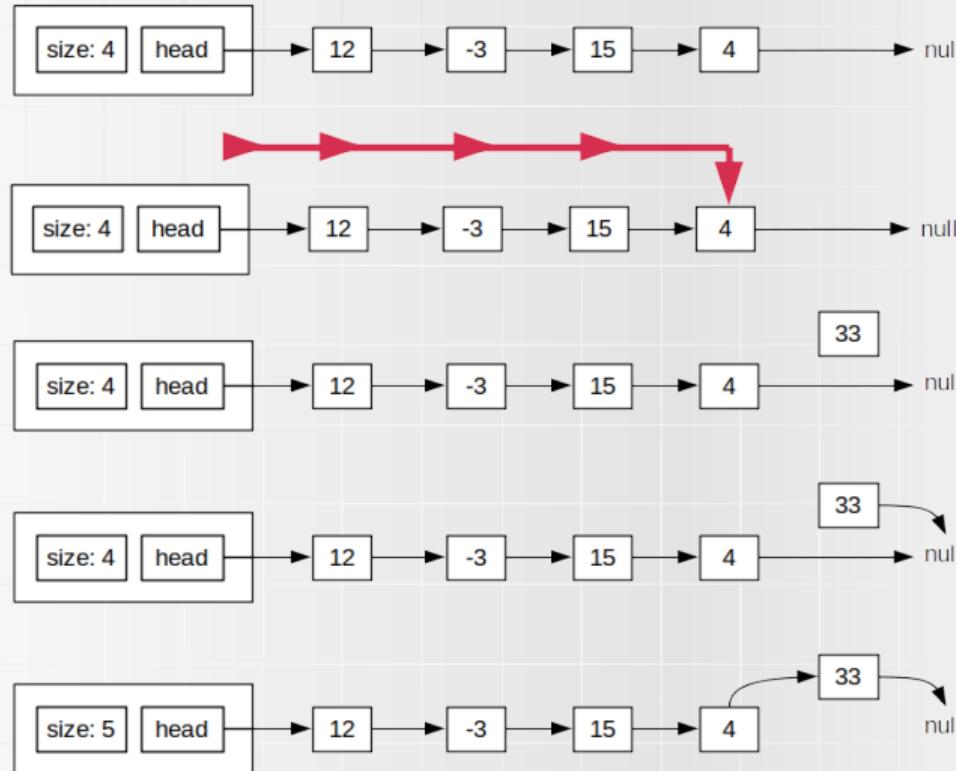
## Lista wiązana (6)

Dodanie elementu  $e$  na ostatniej pozycji ( $i = n - 1$ ).

- ▶ Identycznie jak dodawanie na dowolną pozycję, z tym że  $node.next$  należy ustawić na null.
- ▶ Trzeba dotrzeć do końca listy, więc czas wynosi  $O(n)$ .

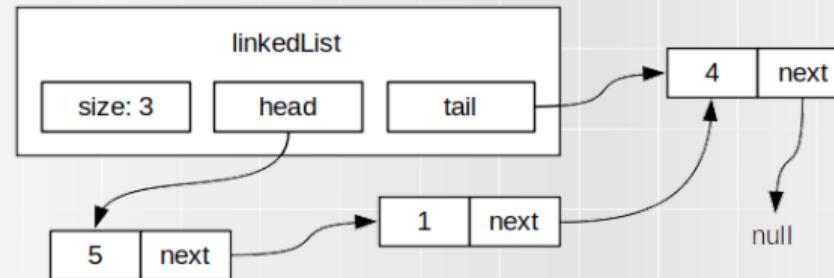


# Lista wiązana (7)



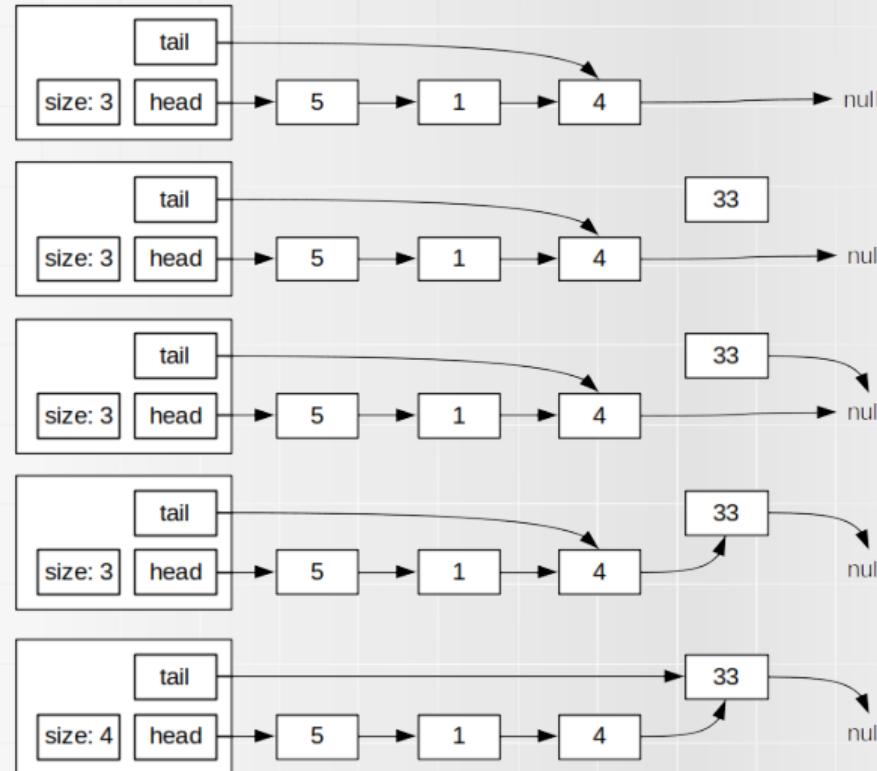
## Lista wiązana (8)

- ▶ Prostym ulepszeniem jest dodanie wskaźnika *tail* wskazującego na koniec listy (ostatni element lub null dla listy pustej).
- ▶ Należy go ustawić na null na początku i pamiętać o ustawieniu gdy zmienia się ostatni element.
- ▶ Umożliwia dodawanie na koniec listy w czasie  $O(1)$ .





# Lista wiązana (9)



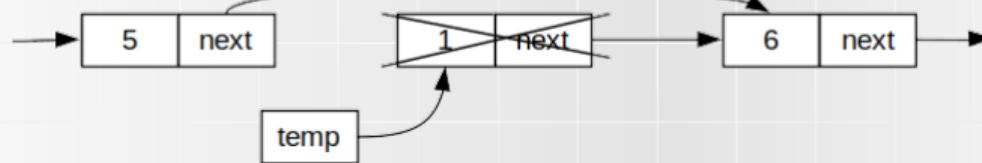
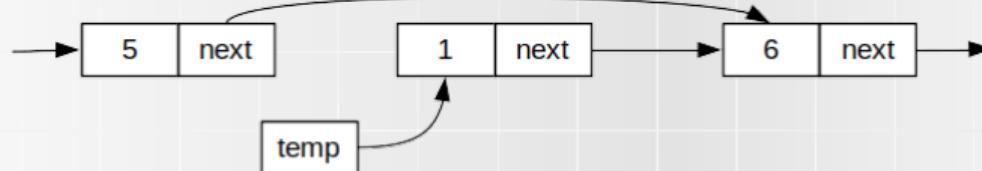
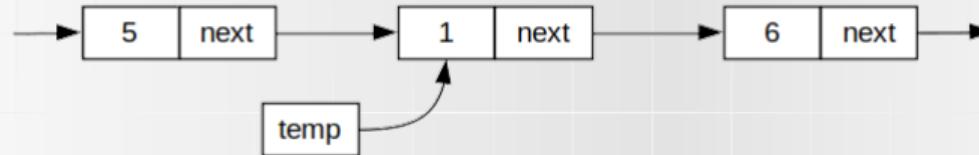
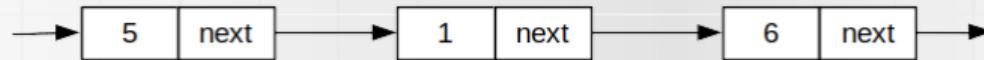


# Lista wiązana (10)

## Usuwanie elementu na pozycji $i$

- ▶ Dotarcie do węzła  $i - 1$  (nazwijmy go  $old$ ).
- ▶  $temp \leftarrow old.next$
- ▶  $old.next \leftarrow old.next.next$
- ▶ Usunięcie węzła  $temp$  (`delete, free()`).
- ▶ Specjalne przypadki:
  - ▶ Usunięcie pierwszego węzła (konieczność modyfikacji `head`).
  - ▶ Usunięcie ostatniego węzła (konieczność modyfikacji `tail`, jeśli jest).

# Lista wiązana (11)





# Lista wiązana (12)

Lista wiązana jednokierunkowa (tylko *head*)

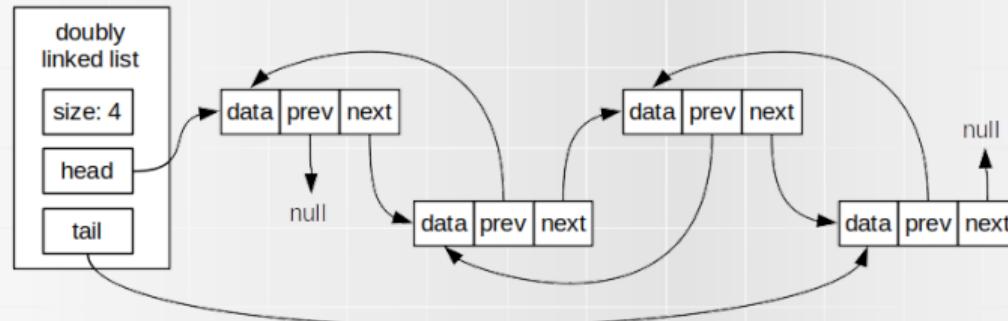
Operacja	Optymistycznie	Średnio	Pesymistycznie
Dodanie na dowolnej pozycji	$O(1)$	$O(n)$	$O(n)$
Dodanie na początku	$O(1)$	$O(1)$	$O(1)$
Dodanie na końcu	$O(n)$	$O(n)$	$O(n)$

Lista wiązana jednokierunkowa (*head* i *tail*)

Operacja	Optymistycznie	Średnio	Pesymistycznie
Dodanie na dowolnej pozycji	$O(1)$	$O(n)$	$O(n)$
Dodanie na początku	$O(1)$	$O(1)$	$O(1)$
Dodanie na końcu	$O(1)$	$O(1)$	$O(1)$

# Lista dwukierunkowa (1)

- ▶ Lista wiązana, gdzie każdy element, ma zarówno wskaźnik na następny element (*next*), jak i na poprzedni (*prev*).
- ▶ Dla ostatniego elementu *next* = *null*.
- ▶ Dla pierwszego elementu *prev* = *null*.
- ▶ Struktura przechowuje *head* i *tail*.



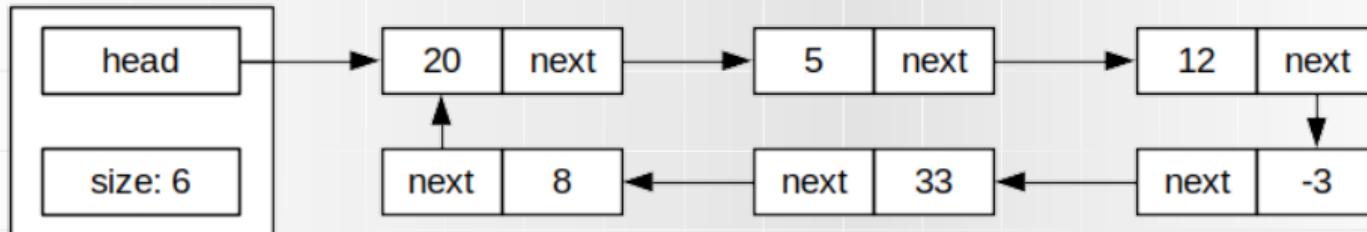


## Lista dwukierunkowa (2)

- ▶ Zajętość pamięci:  $3n + 3$  (wciąż  $O(n)$ , ale narzut jest znaczny).
- ▶ Łatwiejsze przemieszczanie się po liście.
- ▶ Czas dotarcia do węzła  $i$  dwukrotnie mniejszy (choć wciąż  $O(i)$ ).
  - ▶ Prawie dwukrotny mniejszy czas dodawania/usuwania elementów na dowolnej pozycji, szukania oraz przeglądania.
- ▶ Więcej wskaźników do ustawienia podczas operacji.
  - ▶ Nieco wolniejsze dodawanie/usuwanie na końcach.

# Lista cykliczna (1)

- ▶ Ostatni element wskazuje na pierwszy, zamiast na null.
- ▶ Dla listy dwukierunkowej, pierwszy element wskazuje też na ostatni.
- ▶ Wciąż istnieje *head* (inaczej nie można dostać się do listy), ale dowolny element jest początkiem/końcem listy.
- ▶ Koniec listy rozpoznajemy po dotarciu drugi raz do elementu początkowego





## Lista cykliczna (2)

- ▶ Poszukiwanie można zacząć od dowolnego elementu.
- ▶ Przydatna w implementacji niektórych kolejek, buforów cyklicznych czy kopca Fibonacciego
- ▶ Przydatna, gdy przechodzimy po liście wielokrotnie.
- ▶ Bardziej złożona, trudniejsza w kontroli (znalezienie końca, możliwość nieskończonych pętli itp.).



## Rozszerzenia listy wiązanej

Lista z wartownikiem (sentinel):

- ▶ Dodatkowy węzeł (wartownik) przed początkiem/po końcu.
- ▶ Ułatwia obsługę listy.
  - ▶ Każdy wskaźnik można wyłuskać (brak nulla).
  - ▶ Zawsze istnieje jakiś element, nawet jak lista jest pusta.

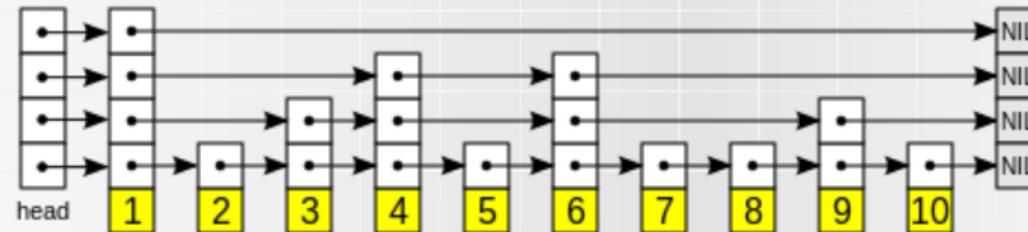
Lista wielokrotnie wiązana:

- ▶ Węzeł posiada więcej wskaźników.
- ▶ Możliwość posiadania różnych kolejności na tych samych danych.



# Lista z przeskokiem

- ▶ Probabilistyczna struktura danych.
- ▶ Lista wielokrotnie wiązana, wiązania mogą pomijać elementy (wg prawdopodobieństwa).
- ▶ Średni czas operacji dodawania/usuwania/wyszukiwania  $O(\log n)$ .
- ▶ Pesymistyczna zajętość pamięci  $O(n \log n)$ .



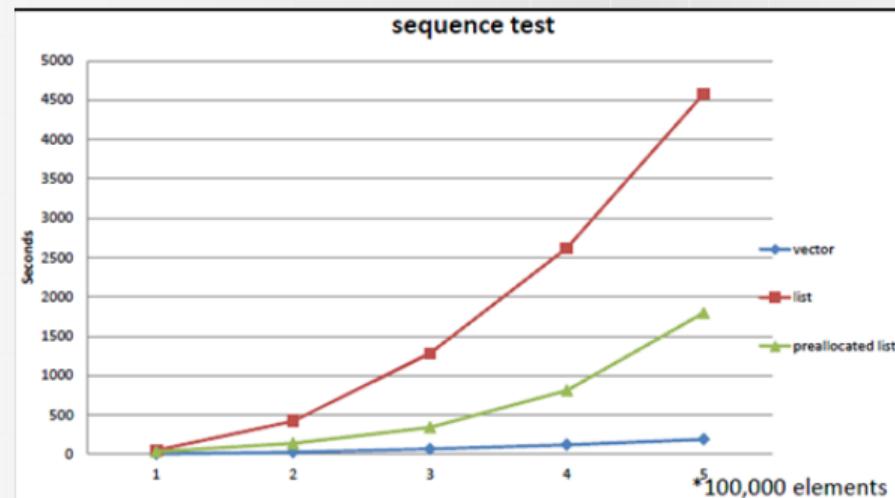


## Tablica dynamiczna vs lista wiązana (1)

- ▶ Tablica dynamiczna ma szybszy czas dostępu do węzła oraz dodatkową pamięć niż lista wiązana –  $O(1)$  vs  $O(n)$ .
- ▶ Lista wiązana ma szybsze dodawanie na początku ( $O(1)$  vs  $O(n)$ ) chyba, że zastosujemy usprawnienie z „ujemnymi” indeksami.
- ▶ Dodawanie na końcu jest w obu przypadkach  $O(n)$ , ale można je ulepszyć.
  - ▶ Tablica dynamiczna – koszt zamortyzowany plus odpowiednio rzadka alokacja pamięci.
  - ▶ Lista wiązana – dodanie wskaźnika na ogon.

## Tablica dynamiczna vs lista wiązana (2)

- ▶ Czy na pewno? Bjarne Stroustrup (twórca C++) przedstawił eksperyment, w którym lista okazała się wolniejsza.
- ▶ Winowiącą jest „niespójna” reprezentacja listy wiązanej w pamięci w połączeniu ze sposobem działania współczesnych pamięci podręcznych (cache).



[1]



## Samoorganizujące się listy (1)

- ▶ Średni czas wyszukiwania na liście (ADT) wynosi  $\lceil \frac{n}{2} \rceil = O(n)$ .
- ▶ Jest to znacznie gorzej niż czas wyszukiwania dla drzew poszukiwań binarnych czy posortowanych list/tablic.
- ▶ Możliwym rozwiązaniem są listy samoorganizujące, które zmieniają kolejność w wyniku kolejnych operacji dostępu/wyszukiwania.
- ▶ Najlepiej sprawdzają się dla sytuacji, gdzie żądania dostępu (lub ich rozkład) znane są z góry.
  - ▶ Zasada 80-20 (20% elementów jest celem 80% wyszukiwań).



## Samoorganizujące się listy (2)

Metoda move-to-front:

- ▶ Element, do którego był dostęp przesuwany jest na początek listy.
- ▶ Proste dla listy wiązanej (dodatkowy czas  $O(1)$ ).
- ▶ Trudniejsze dla tablicy dynamicznej (dodatkowy czas  $O(n)$ ).
- ▶ Względnie prosta implementacja.
- ▶ Podatny na przeszacowanie (przenoszenie na sam już po pierwszym dostępie).



## Samoorganizujące się listy (3)

Metoda transpose (swap):

- ▶ Element, do którego był dostęp przesuwany jest na pozycję o 1 wcześniej.
- ▶ Proste zarówno dla tablicy dynamicznej i listy wiązanej.
  - ▶ Jeśli lista nie jest dwukierunkowa, to należy pamiętać element poprzedni.
- ▶ Przenoszenie elementów do przodu jest stopniowe.
- ▶ Dobrze dostosowuje się do sytuacji, gdzie wzorzec (rozkład prawdopodobieństwa) żądanych elementów zmienia się w czasie.



## Samoorganizujące się listy (4)

Metoda count:

- ▶ Każdy węzeł ma licznik odwołań (ile razy był dostęp).
- ▶ Wymaga dodatkowo  $O(n)$  pamięci.
- ▶ Węzły układane są w kolejności malejącego licznika.
  - ▶ Po odwołaniu wykonuje się tyle swapów ile potrzeba, by uzyskać poprawne sortowanie.
  - ▶ Wyszukiwanie może wydłużyć się o dodatkowe  $O(n)$ , ale średnio będzie szybsze.
  - ▶ Podobnie zwykły dostęp dla listy wiązanej. Dla tablicy dynamicznej dostęp się pogorszy!



# Bibliografia

Wroclaw  
University  
of Science  
and Technology



<https://bulldozer00.blog/2012/02/09/vectors-and-lists/>



Wrocław  
University  
of Science  
and Technology

# Struktury danych

Wykład 4  
Kolejki

dr inż. Jarosław Rudy





## Kolejki (1)

- ▶ Kolejki (queue) to ADT, w którym dodawanie i usuwanie elementów jest ze sobą powiązane tj. usuwany element jest określony przez cechy elementów dodanych do tej pory (ich kolejność, priorytet itp.).
  - ▶ W praktyce kolejki utrzymują pewien (najczęściej liniowy) porządek elementów.
  - ▶ Wtedy dodawanie/usuwanie przekłada się na dodawane/usuwanie na kórymś lub obu końcach kolejki.
- ▶ Podobnie jak dla listy elementy mogą się powtarzać i być różnego typu.
- ▶ Kolejki generalnie mają zmienną długość, ale rozważa się też przypadki o stałym rozmiarze.



## Kolejki (2)

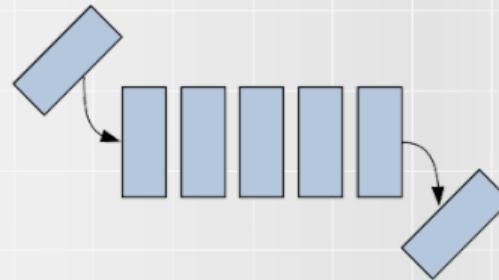
Typowe operacje na kolejkach:

- ▶ Dodanie elementu.
- ▶ Usunięcie elemenu.
- ▶ Sprawdzenie czy kolejka jest pusta (ewentualnie zwrócenie rozmiaru).
- ▶ Podgląd elementu do usunięcia (peek), ale bez usuwania.

Aby dostać się do elementu trzeciego, najpierw należy zdjąć elementy pierwszy i drugi! Kolejki zasadniczo nie służą do dostępu do dowolnego elementu, wyszukiwania czy przeglądania.

# Kolejka FIFO (1)

- ▶ Kolejka działająca wg zasady First-In First-Out (elementy dodany jako pierwszy jest usuwany jako pierwszy).
- ▶ Operacja dodawania nazywa się enqueue, elementy dodawane są do „końca” (back, tail).
- ▶ Operacja zdejmowania nazywa się dequeue, elementy zdejmowane są z „początku” (front, head).
- ▶ Bez dodatkowego kontekstu „kolejka” zwykle oznacza „kolejka FIFO”.





## Kolejka FIFO (2)

Implementacja z użyciem listy wiązanej jednokierunkowej (bez tail):

Operacja kolejki	Operacja listy	Czas pesymistyczny
enqueue( $e$ )	addBack( $e$ )	$O(n)$
dequeue()	removeFront()	$O(1)$
empty()	empty()	$O(1)$
peek()	get(0)	$O(1)$

Albo:

Operacja kolejki	Operacja listy	Czas pesymistyczny
enqueue( $e$ )	addFront( $e$ )	$O(1)$
dequeue()	removeBack()	$O(n)$
empty()	empty()	$O(1)$
peek()	get( $n - 1$ )	$O(n)$



## Kolejka FIFO (3)

Implementacja z użyciem listy wiązanej jednokierunkowej (z tail) lub dwukierunkowej:

Operacja kolejki	Operacja listy	Czas pesymistyczny
enqueue( $e$ )	addBack( $e$ )/addFront( $e$ )	$O(1)$
dequeue()	removeFront()/removeBack()	$O(1)$
empty()	empty()	$O(1)$
peek()	get(0)/get( $n - 1$ )	$O(1)$

Implementacja jest wydajna, ale współdzieli wady listy wiązanej (zajętość pamięci  $2n + O(1)$ , słabsza współpraca z pamięcią podręczną).



# Kolejka FIFO (4)

Implementacja z użyciem tablicy dynamicznej:

Operacja kolejki	Operacja tablicy	Czas pesymistyczny
enqueue(e)	addBack(e)	$O(1)$ (amortyzowany)
dequeue()	removeFront()	$O(1)$ (amortyzowany)
empty()	empty()	$O(1)$
peek()	get(0)	$O(1)$

Albo:

Operacja kolejki	Operacja tablicy	Czas pesymistyczny
enqueue(e)	addFront(e)	$O(1)$ (amortyzowany)
dequeue()	removeBack()	$O(1)$ (amortyzowany)
empty()	empty()	$O(1)$
peek()	get( $n - 1$ )	$O(1)$



# Kolejka FIFO (5)

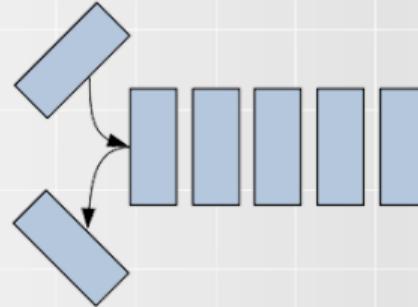
Niektóre zastosowania:

- ▶ Bufory.
- ▶ Przechowywanie żądań obsługiwanych w kolejności zgłoszeń.
- ▶ Modelowanie kolejek sklepowych.
- ▶ Przegląd wszerz drzewa.
- ▶ Potoki w unixie.
- ▶ Komunikacja strumieniowa (np. TCP).



# Stos (1)

- ▶ W stosie (ADT) element dodany najpóźniej jest zdejmowany jako pierwszy. Stos jest więc kolejką LIFO (Last-In Fist-Out).
- ▶ Elementy są więc dodawane i usuwane z tego samego końca (szczytu).
- ▶ Operacja dodawania nazywa się `push()`.
- ▶ Operacja zdejmowania nazywa się `pop()`.
- ▶ Operacja peek często nazywa się `top()`.





## Stos (2)

Implementacja z użyciem listy wiązanej jednokierunkowej (bez tail):

Operacja kolejki	Operacja listy	Czas pesymistyczny
enqueue( $e$ )	addBack( $e$ )	$O(n)$
dequeue()	removeBack()	$O(n)$
empty()	empty()	$O(1)$
top()	get( $n - 1$ )	$O(n)$

Albo:

Operacja kolejki	Operacja listy	Czas pesymistyczny
enqueue( $e$ )	addFront( $e$ )	$O(1)$
dequeue()	removeFront()	$O(1)$
empty()	empty()	$O(1)$
top()	get(0)	$O(1)$



## Stos (3)

Implementacja z użyciem listy wiązanej jednokierunkowej (z tail) lub dwukierunkowej:

Operacja kolejki	Operacja listy	Czas pesymistyczny
enqueue( $e$ )	addFront( $e$ )/addBack( $e$ )	$O(1)$
dequeue()	removeFront()/removeBack()	$O(1)$
empty()	empty()	$O(1)$
top()	get(0)/get( $n - 1$ )	$O(1)$



## Stos (4)

Implementacja z użyciem tablicy dynamicznej:

Operacja kolejki	Operacja tablicy	Czas pesymistyczny
enqueue( $e$ )	addBack( $e$ )	$O(1)$ (amortyzowany)
dequeue()	removeBack()	$O(1)$ (amortyzowany)
empty()	empty()	$O(1)$
top()	get( $n - 1$ )	$O(1)$

Albo:

Operacja kolejki	Operacja tablicy	Czas pesymistyczny
enqueue( $e$ )	addFront( $e$ )	$O(1)$ (amortyzowany)
dequeue()	removeFront()	$O(1)$ (amortyzowany)
empty()	empty()	$O(1)$
top()	get(0)	$O(1)$



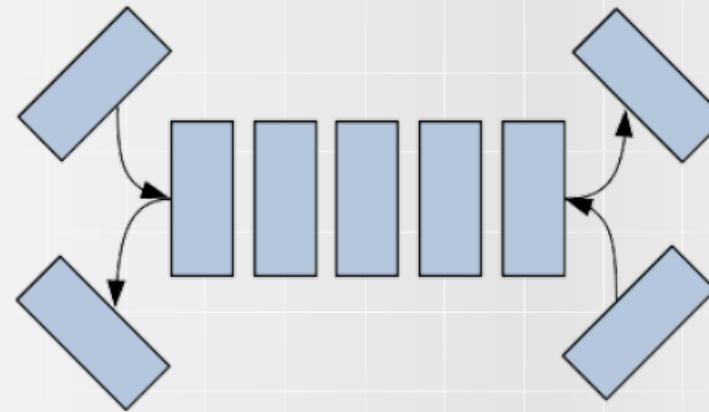
# Stos (5)

Niektóre zastosowania:

- ▶ Parsowanie wyrażeń (np. arytmetycznych) zapisanych w odwrotnej notacji polskiej.
- ▶ Przegląd w głąb drzewa.
- ▶ Algorytmy z nawrotami (backtracking).
- ▶ Stos programowy.

# Kolejka dwukierunkowa (1)

- ▶ Kolejka, w której można dodawać i usuwać elementy na obu końcach.
- ▶ Double-ended que = deque (czyt. „deku”).
- ▶ Nie mylić z operacją dequeue (czyt. „dikju”) kolejki FIFO!





## Kolejka dwukierunkowa (2)

- ▶ Stos i kolejka FIFO mogą być traktowane jako uszczegółowienie deque.
- ▶ Deque może być traktowana jako uszczegółowienie listy.

Operacja	List	Deque	FIFO	Stack
addFront( $e$ )	✓	✓	✗	✓
removeFront()	✓	✓	✓	✓
addBack( $e$ )	✓	✓	✓	✗
removeBack()	✓	✓	✗	✗
add( $e, i$ )	✓	✗	✗	✗
remove( $i$ )	✓	✗	✗	✗



## Kolejka dwukierunkowa (3)

Implementacja z użyciem listy wiązanej jednokierunkowej (bez tail):

Operacja kolejki	Operacja listy	Czas pesymistyczny
addFront( $e$ )	addFront( $e$ )	$O(1)$
addBack( $e$ )	addBack( $e$ )	$O(n)$
removeFront()	removeFront()	$O(1)$
removeBack()	removeBack()	$O(n)$
empty()	empty()	$O(1)$
front()	get(0)	$O(1)$
back()	get( $n - 1$ )	$O(n)$



## Kolejka dwukierunkowa (4)

Implementacja z użyciem listy wiązanej jednokierunkowej (z tail) lub dwukierunkowej:

Operacja kolejki	Operacja listy	Czas pesymistyczny
addFront( $e$ )	addFront( $e$ )	$O(1)$
addBack( $e$ )	addBack( $e$ )	$O(1)$
removeFront()	removeFront()	$O(1)$
removeBack()	removeBack()	$O(1)$
empty()	empty()	$O(1)$
front()	get(0)	$O(1)$
back()	get( $n - 1$ )	$O(1)$



# Kolejka dwukierunkowa (5)

Implementacja z użyciem tablicy dynamicznej:

Operacja kolejki	Operacja tablicy	Czas pesymistyczny
addFront(e)	addFront(e)	$O(1)$ (amortyzowany)
addBack(e)	addBack(e)	$O(1)$ (amortyzowany)
removeFront()	removeFront()	$O(1)$ (amortyzowany)
removeBack()	removeBack()	$O(1)$ (amortyzowany)
empty()	empty()	$O(1)$
front()	get(0)	$O(1)$
back()	get( $n - 1$ )	$O(1)$



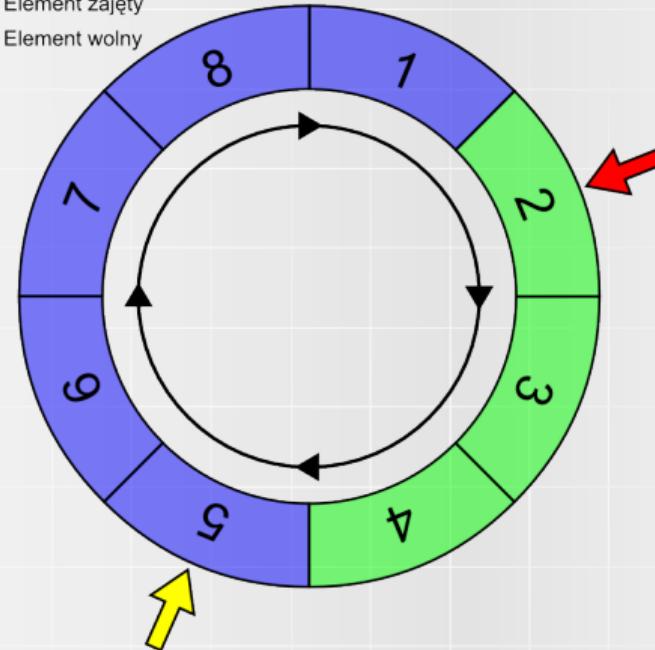
## Kolejka cykliczna (1)

- ▶ Kolejka (bufor) cykliczny jest strukturą danych, zwykle o stałym rozmiarze.
- ▶ Najczęściej działa w charakterze FIFO.
- ▶ Dwa wskaźniki na aktualny początek (odczyt) i koniec (zapis) kolejki.
- ▶ Pozycje wskaźników aktualizowane przy operacjach dodawania i zdejmowania.
- ▶ Stare dane są zapisywane przez nowe.
- ▶ Teoretycznie może służyć do przesyłu dowolnie dużych danych
  - ▶ Gdy jeden wskaźnik dogoni drugi, operacja możliwa dopiero po przesunięciu drugiego wskaźnika.



## Kolejka cykliczna (2)

- Wskaźnik odczytu
- Wskaźnik zapisu
- Element zajęty
- Element wolny





## Kolejka cykliczna (3)

- ▶ Prostota, oszczędność miejsca, szybki dostęp.
- ▶ Ograniczony rozmiar.
  - ▶ Dynamiczne bufory cykliczne.
- ▶ Stosowane np. do implementacji bufora klawiatury.
- ▶ Przy dobrej implementacji czas operacji wynosi  $O(1)$ .
  - ▶ Tablica dynamiczna wymaga „ręcznego” zawijania indeksów.
  - ▶ Lista wiązana działa dzięki pamiętaniu wskaźników.



## Kolejka priorytetowa (1)

- ▶ Kolejka w której każdy element ma przypisany priorytet liczbowy.
  - ▶ Ogólnie, elementy są parą klucz-wartość, gdzie na kluczach da się określić relację maksimum (minimum). Klucze muszą mieć więc częściowy porządek.
  - ▶ Priorytetem może być po prostu wartość elementu.
- ▶ O „kolejności” kolejki decyduje priorytet elementów:
  - ▶ Dodawanie elementu  $e$  o priorytecie  $p$  dodaje go (jakoś) do kolekcji.
  - ▶ Zdejmowanie zawsze zdejmuje element o największym (najmniejszym) priorytecie.
- ▶ Może istnieć wiele elementów o tym samym priorytecie. Różne strategie:
  - ▶ Stabilność – gwarancja zdejmowania takich elementów w kolejności ich dodawania (FIFO).
  - ▶ Niestabilne – brak takiej gwarancji.



## Kolejka priorytetowa (2)

Kolejka priorytetowa typu max:

- ▶  $\text{insert}(e,p)$  – dodanie elementu  $e$  o priorytecie  $p$ .
- ▶  $\text{extract-max}()$  – usunięcie i zwrócenie elementu o największym priorytecie.
- ▶  $\text{find-max}()$  – zwrócenie (podejrzenie) elementu o największym priorytecie.
- ▶  $\text{modify-key}(e,p)$  – zmiana priorytetu elementu  $e$  na  $p$ . Można podzielić na operacje  $\text{decrease-key}$  oraz  $\text{increase-key}$ .

Kolejka priorytetowa typu min jest analogiczna ( $\text{extract-min}$  zwraca element o najmniejszym priorytecie itp).



## Kolejka priorytetowa (3)

- ▶ Pierwszą (naiwną) implementacją kolejki priorytetowej jest wykorzystanie listy, czyli użycie wprost tablicy dynamicznej lub listy wiązanej.
- ▶ Zakładamy, że struktury posiadają odpowiednie usprawnienia, inaczej przedstawione dalej złożoności mogą nie być zawsze spełnione.
- ▶ Implementacja z użyciem listy wiązanej wykorzystuje więcej pamięci i może gorzej współpracować z pamięcią podręczną procesora.
- ▶ Przedstawiona implementacja dotyczy kolejki priorytetowej typu max. Implementacja kolejki typu min jest analogiczna.



## Kolejka priorytetowa (4)

- ▶ Dodajemy elementy na koniec jak w zwykłej kolejce FIFO. Przy zdejmowaniu należy znaleźć największy element.
- ▶ Operacje:
  - ▶  $\text{insert}(e,p)$  – za pomocą  $\text{addBack}(e)$ , czas  $O(1)$  (zwyczajny lub amortyzowany).
  - ▶  $\text{extract-max}()$  – za pomocą przeszukania listy, zwrócenia i usunięcia znalezionej elementu, czas  $O(n)$ .
  - ▶  $\text{peek}()$  – analogicznie do  $\text{extract-max}()$ , ale bez usuwania, czas  $O(n)$ .
  - ▶  $\text{modify-key}(e,p)$  – znalezienie elementu i modyfikacja jego priorytetu, czas  $O(n)$ .



## Kolejka priorytetowa (5)

- ▶ Możemy również odwrócić koncepcję – dodajemy elementy od razu w potrzebne miejsce (jak insert sort), wtedy lista jest posortowana i największy element będzie zawsze na początku.
- ▶ Operacje:
  - ▶  $\text{insert}(e, p)$  – za pomocą wyszukania odpowiedniego miejsca i wstawienia, czas  $O(n)$ .
  - ▶  $\text{extract-max}()$  – zwracamy i usuwamy pierwszy element, czas  $O(1)$  (amortyzowany lub nie).
  - ▶  $\text{peek}()$  – analogicznie do  $\text{extract-max}()$ , ale bez usuwania, czas  $O(1)$  (amortyzowany lub nie).
  - ▶  $\text{modify-key}(e, p)$  – znalezienie elementu, modyfikacja jego priorytetu i przeniesienie elementu w odpowiednie miejsce, czas  $O(n)$ .



Wrocław  
University  
of Science  
and Technology

# Struktury danych

Wykład 5  
Drzewa, kopce

dr inż. Jarosław Rudy





# Drzewa (1)

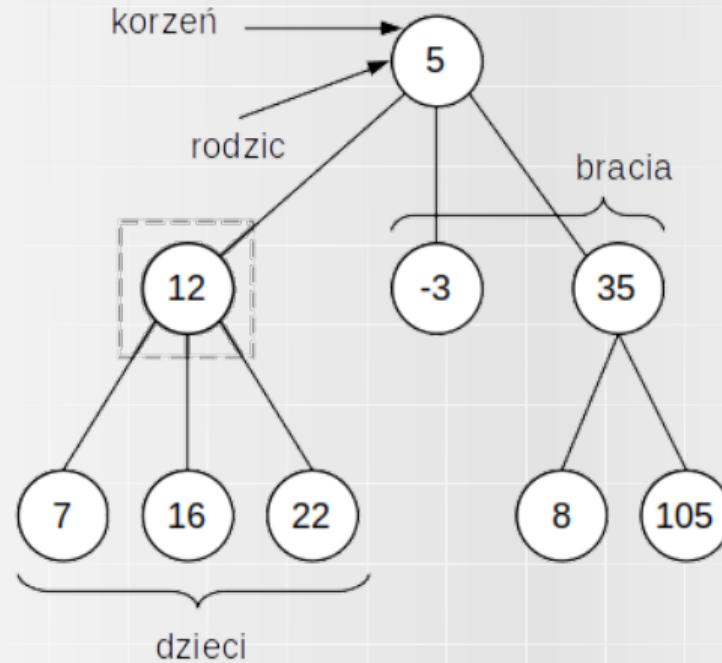
- ▶ Matematycznie drzewo jest zbiorem wierzchołków (węzłów) oraz łączących je krawędzi.
- ▶ Intuicyjnie, drzewo możemy podzielić na poziomy.
- ▶ Na zerowym poziomie jest tylko jeden wierzchołek (zwany korzeniem drzewa) i ma on połączenia jedynie z wierzchołkami na poziomie pierwszym.
- ▶ Wierzchołki na pozostałych poziomach:
  - ▶ Zawsze mają dokładnie jedno połączenie z jednym z wierzchołków na poziomie poprzednim. Wierzchołek ten nazywany jest rodzicem.
  - ▶ Mają dowolną (0 lub więcej) liczbę połączeń z wierzchołkami na poziomie kolejnym. Wierzchołki te nazywa się synami lub dziećmi (rzadziej potomkami).



## Drzewa (2)

- ▶ Analogicznie można określić dla danego węzła dziadka, pradziadka itd. (ogólnie poprzedników) oraz wnuków, prawnuków itd. (ogólniej następców lub potomków).
- ▶ Wierzchołki mającego tego samego rodzica nazywamy rodzeństwem lub braćmi itp.
- ▶ Wierzchołki bez dzieci nazywają się liśćmi.
- ▶ Dużo łatwiej i ściślej można zdefiniować drzewo używając pojęcia grafu – drzewo jest grafem, który jest jednocześnie nieskierowany, spójny i acykliczny.
- ▶ Tradycyjnie drzewa przedstawiane są z korzeniem na górze.
- ▶ Określenie korzenia jest kwestią wyboru – każdy wierzchołek w drzewie może być korzeniem, zaś jego wybór określa relacje pomiędzy wierzchołkami.

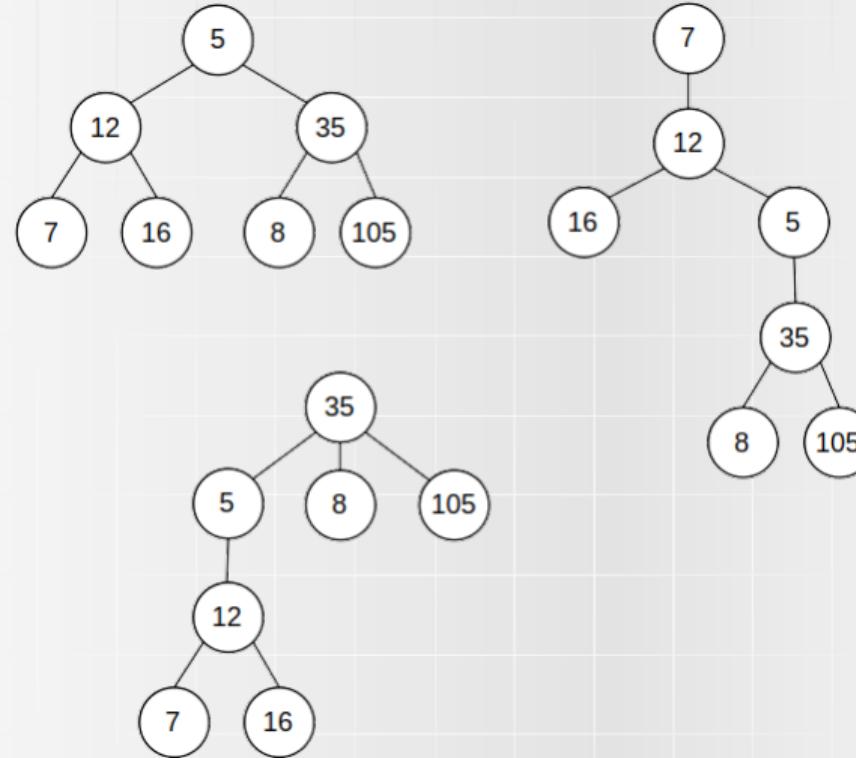
# Drzewa (3)



Wierzchołki 7, 16, 22, -3, 8 i 105 są liśćmi.



# Drzewa (4)





## Drzewa (5)

Drzewo jest ADT czy strukturą danych?

- ▶ Powszechnie drzewo uznawane jest za strukturę danych.
- ▶ Dzieje się tak pomimo faktu, że definicja drzewa nie określa sposobu rozmieszczenia danych w pamięci ani złożoności operacji.
- ▶ Z matematycznego i informatycznego punktu widzenia na drzewie można określić operacje typu dodanie/usunięcie węzła, wyszukanie węzła, rotacja węzła, przeszukanie drzewa (wszerz, wgłąb itp.).
- ▶ Takie ujęcie może wskazywać na charakter ADT.

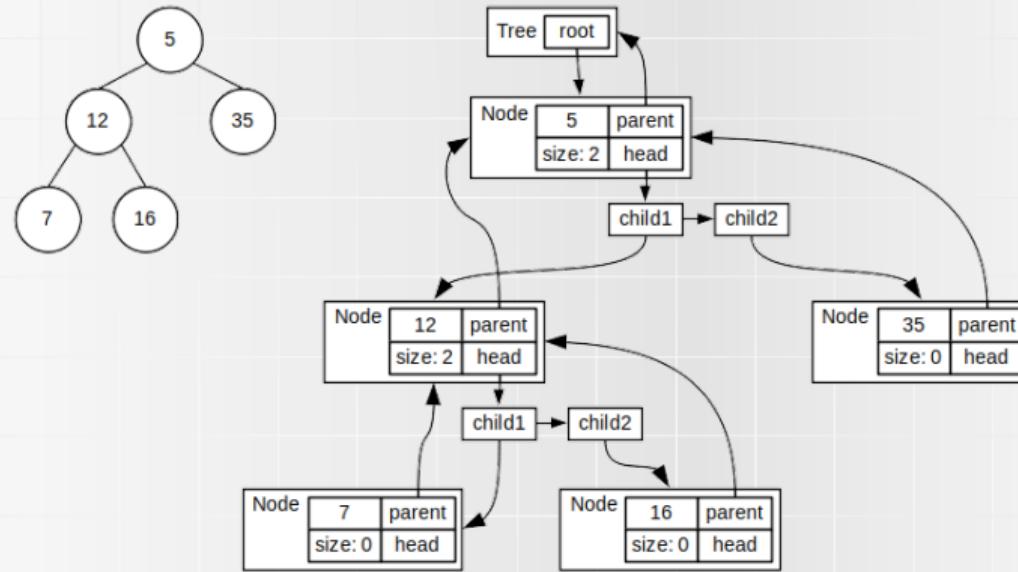


## Drzewa (6)

- ▶ W praktyce istotny jest kontekst.
  - ▶ Jeśli charakterystyka drzewa wynika wprost z zagadnienia (np. genealogia, ciąg decyzyjny), to może być bliżej ADT.
  - ▶ Jeśli charakterystyka drzewa wynika z wyboru (np. implementacja słownika), to może być bliżej strukturze danych.
- ▶ Drzewo często stoi pomiędzy pierwotnym ADT a końcową strukturą danych:
  - ▶ Dodaje dodatkowe założenia względem np. słownika.
  - ▶ Musi być zaimplementowane w konkretnym sposób.

# Drzewa (7)

Drzewo może oznaczać konkretną strukturę np. z użyciem list wiązanych:



Dla wielu drzew (statyczne, zupełne binarne itd.) istnieją dużo lepsze rozwiązania. W szczególności można użyć reprezentacji grafowej.



## Drzewa (8)

- ▶ Ścieżka (czasem też droga) – ciąg krawędzi łączący dwa wierzchołki (bez powtarzania się krawędzi).
- ▶ W drzewie zawsze istnieje jedna ścieżka od korzenia do danego wierzchołka.
- ▶ Liczba krawędzi na ścieżce nazywana jest jej długością.
- ▶ Dla danego wierzchołka  $A$  jego poziom to długość ścieżki od korzenia do  $A$ .
- ▶ Wysokość drzewa to największy z jego poziomów.



# Drzewa (10)

Typowe operacje wykonywane na drzewie:

- ▶ Dodanie elementu.
- ▶ Usunięcie elementu.
- ▶ Wyszukanie elementu.
- ▶ Usunięcie poddrzewa.
- ▶ Rotacja poddrzewa.
- ▶ Przeglądnięcie drzewa (odwiedzenie wszystkich wierzchołków w pewnej kolejności).



## Drzewa (11)

Wrocław  
University  
of Science  
and Technology

- ▶ Drzewo może być puste – brak wierzchołków, a tym samym brak korzenia.
- ▶ W drzewie można rozpatrywać poddrzewa – pewien wierzchołek  $A$  wraz ze wszystkimi lub częścią jego potomków. Wtedy  $A$  jest korzeniem poddrzewa.
- ▶ Zbiór (rozłącznych) drzew nazywany jest lasem.
- ▶ Istnieje wiele szczególnych przypadków drzew (drzewa binarne, BST, czerwono-czarne, ósemkowe itd).



# Drzewa (12)

Zastosowanie drzew:

- ▶ Modelowanie hierarchii (systemy plików, dokumenty HTML/XML, dziedziczenie klas).
- ▶ Procesy decyzyjne (giełda, problem plecakowy, komiwojażera itd.).
- ▶ Rozkład gramatyczny zdań (informatyka i lingwistyka).
- ▶ Implementacja słowników.
- ▶ Implementacja kolejek priorytetowych.



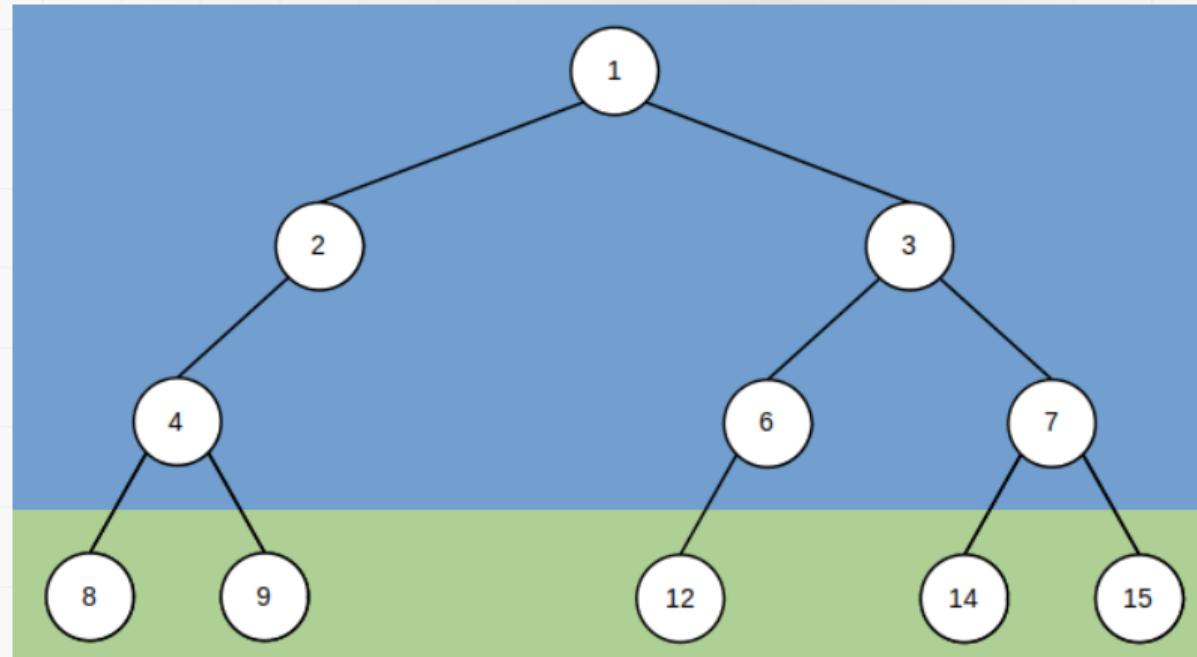
# Drzewa binarne (1)

- ▶ Drzewo binarne – drzewo, w którym każdy węzeł ma co najwyżej dwoje dzieci.
- ▶ Drzewo binarne pełne (full) – drzewo binarne, w którym 1) liście są tylko na ostatnim poziomie, 2) wszystkie nie-liście mają dokładnie 2 dzieci.
- ▶ Drzewo binarne zupełne (complete) – drzewo binarne, w którym 1) ostatni poziom wypełniany jest „od lewej”, 2) wszystkie pozostałe węzły mają dokładnie 2 dzieci (tzn. poziomy oprócz ostatniego są drzewem pełnym).



# Drzewa binarne (2)

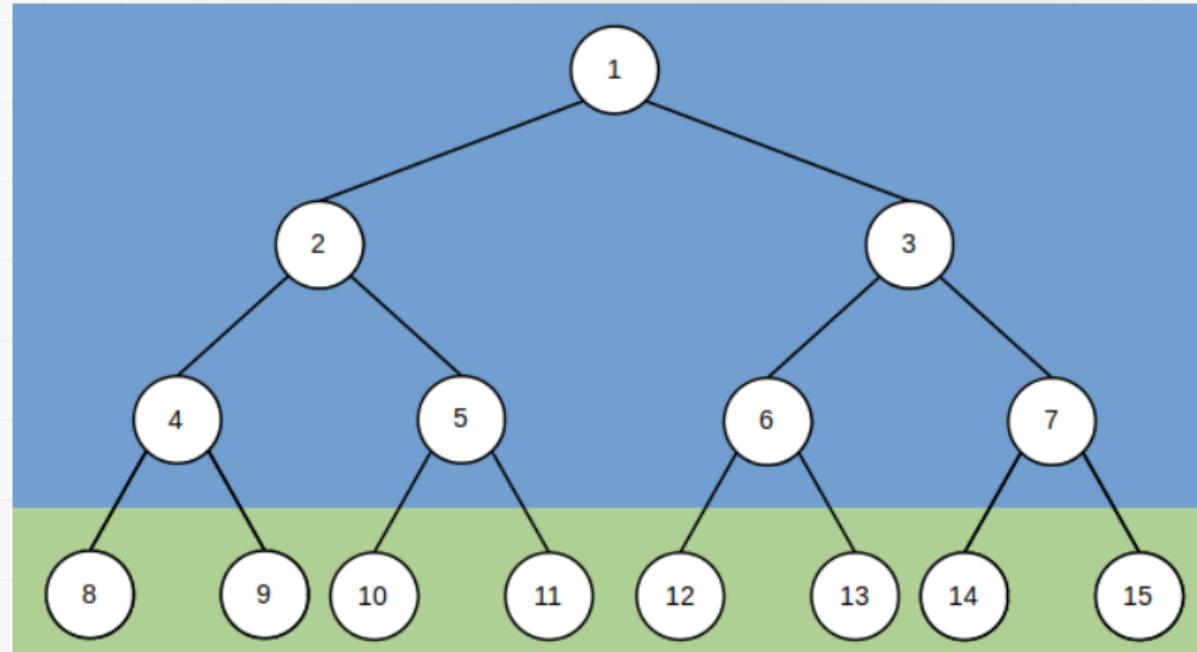
## Drzewo binarne





# Drzewa binarne (3)

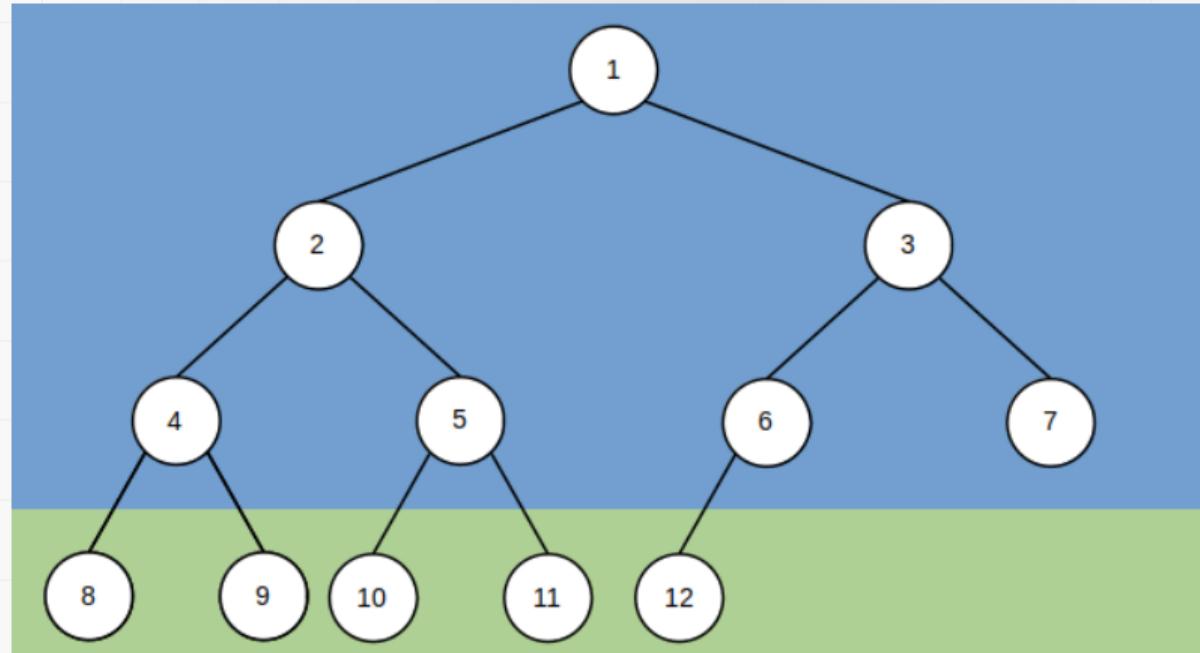
## Drzewo binarne pełne





## Drzewa binarne (4)

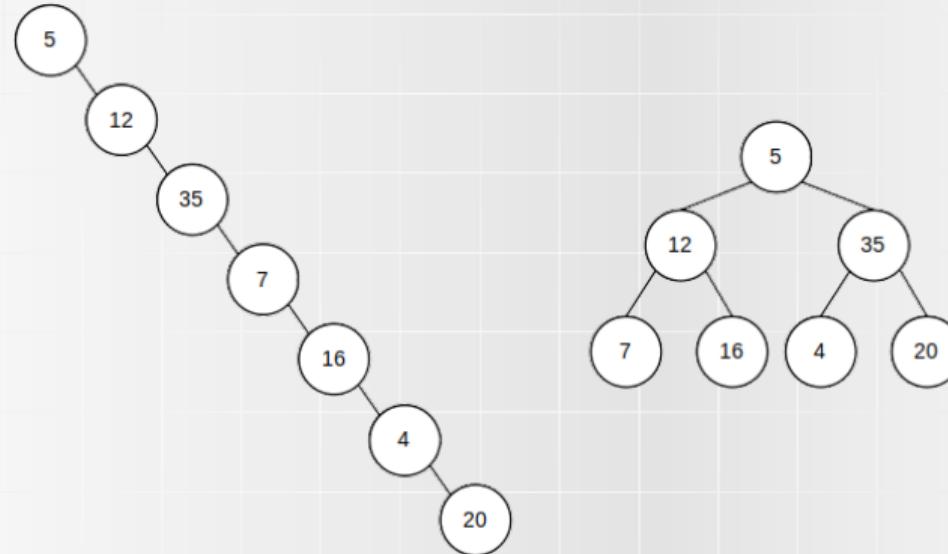
### Drzewo binarne zupełne





## Drzewa binarne (5)

Wysokość drzewa binarnego o  $n$  elementach wynosi  $O(n)$ , ale dla drzewa binarnego zupełnego już tylko  $O(\log n)$ !





## Przejście przez drzewo (1)

- ▶ Przejście przez drzewo oznacza odwiedzenie każdego węzła (zwykle w pewnej kolejności), często wykonując na każdym odwiedzanym węźle pewną czynność (np. wydrukowanie wartości).
- ▶ Istnieje kilka ważnych sposobów przejścia przez drzewo:
  - ▶ Breadth-first (wszerz) – rodzeństwo węzła jest odwiedzane przed jego dziećmi. Odpowiada użyciu kolejki FIFO.
  - ▶ Depth-first (wgłąb) – dzieci węzła są odwiedzane przed jego rodzeństwem. Odpowiada użyciu kolejki LIFO (stosu).
  - ▶ Best-first (najpierw najlepszy) – odwiedzamy węzły wg priorytetu. Odpowiada użyciu kolejki priorytetowej.



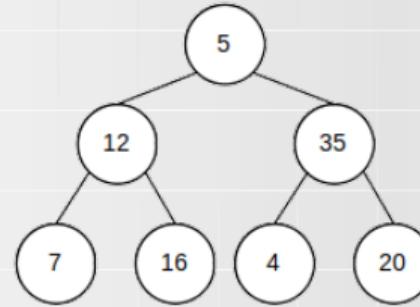
## Przejście przez drzewo (2)

Dla przejścia w głąb można rozważyć kilka wariantów:

- ▶ Pre-order – rodzic jest odwiedzany przed swoimi dziećmi.
- ▶ Post-order – rodzic jest odwiedzany po swoich dzieciach.
- ▶ In-order – najpierw odwiedzany jest lewy syn, potem rodzic, potem prawy syn ( dotyczy drzew binarnych).



## Przejście przez drzewo (3)



- ▶ Breadth-first: 5, 12, 35, 7, 16, 4, 20.
- ▶ Depth-first:
  - ▶ Pre-order: 5, 12, 7, 16, 35, 4, 20.
  - ▶ Post-order: 7, 16, 12, 4, 20, 35, 5.
  - ▶ In-order: 7, 12, 16, 5, 4, 35, 20.
- ▶ Best-first: 5, 35, 20, 12, 16, 7, 4.



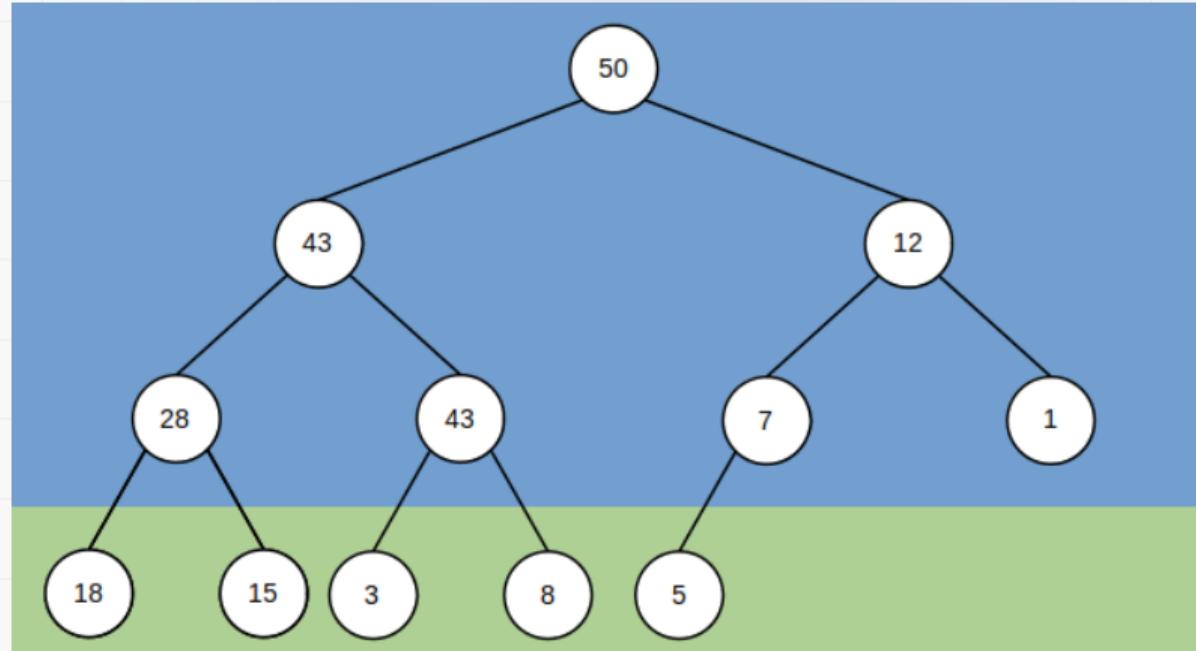
## Kopce (1)

- ▶ Kopiec (heap) – drzewo w którym zachowana jest zasada kopca tzn. klucz rodzica jest w stałej relacji z kluczami jego dzieci. Najczęściej rozróżniamy:
  - ▶ Max heap: rodzic jest nie mniejszy od swoich dzieci.
  - ▶ Min heap: rodzic jest nie większy od swoich dzieci.
- ▶ W efekcie każda ścieżka od korzenia do liścia jest posortowana (kopiec jest częściowo posortowany).
- ▶ Jeśli węzły nie mają klucza, to kluczem staje się sama wartość.
- ▶ Zbiór kluczy musi mieć określony częściowy porządek.
- ▶ Kopce również mogą być binarne, pełne i zupełne.



# Kopce (2)

Kopiec binarny zupełny typu max





## Kopce (3)

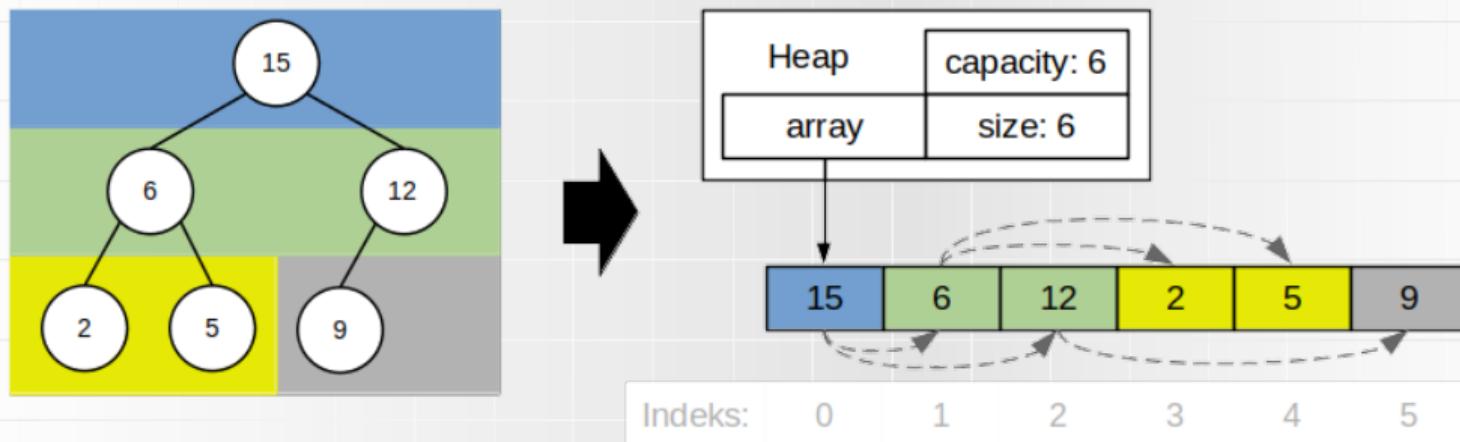
Kopiec binarny zupełny można wydajnie i prosto zaimplementować z użyciem tablicy dynamicznej:

- ▶ Przyjmując numerację od 0, jeśli rodzic ma indeks  $k$  w tablicy, to jego dzieci mają indeksy  $2k + 1$  oraz  $2k + 2$ .
- ▶ Jeśli węzeł ma indeks  $k$ , to jego rodzic ma indeks  $\lfloor \frac{k-1}{2} \rfloor$  (arytmetyka całkowitoliczbową).
- ▶ Ponieważ kopiec jest zupełny (ostatni poziom wypełniany od lewej do prawej), to tablica wykorzystywana jest w pełni bez dziur (nie licząc przestrzeni zaalokowanej na przyszłość).
- ▶ Znając indeks, dostęp do węzła jest w czasie  $O(1)$ .



# Kopce (4)

Przykład implementacji (binarnego zupełnego) kopca za pomocą tablicy dynamicznej:





## Kopce (5)

Operacje kopcowe (kopiec typu max):

- ▶ `insert()` – dodanie elementu.
- ▶ `extract-max()` – usunięcie i zwrócenie elementu o największym kluczu.
- ▶ `find-max()` – zwrócenie elementu o największym kluczu.
- ▶ `find()` – szukanie elementu o danej wartości/kluczu.
- ▶ `delete()` – usunięcie elementu o danej wartości.
- ▶ `decrease-key()/increase-key()` – zmiana wartości klucza elementu.
- ▶ `build()` – zbudowanie kopca z  $n$  elementów.



# Kopce (6)

Pomocnicze operacje do przywracania własności kopca.

► heapify-up

1. Zaczynamy od danego elementu  $e$ .
2. Jeśli  $e$  jest korzeniem lub jest w prawidłowej relacji ze swoim rodzicem – koniec algorytmu.
3. W przeciwnym razie (złamana własność kopca) zamień  $e$  miejscami ze swoim rodzicem i wróć do kroku 2.

► Pesymistyczny czas zależny od wysokości drzewa, czyli  $O(\log n)$ .



# Kopce (7)

- ▶ heapify-down
  1. Zaczynamy od danego elementu e.
  2. Jeśli e jest liściem lub jest w prawidłowej relacji ze swoimi dziećmi – koniec algorytmu.
  3. W przeciwnym razie (złamana własność kopca) zamień e miejscami ze swoim większym dzieckiem i wróć do kroku 2.
- ▶ Dla kopca typu min analogicznie (zamieniamy z mniejszym dzieckiem).
- ▶ Pesymistyczny czas  $O(\log n)$ .



# Kopce (8)

## ► insert()

- ▶ Wstawiamy element na pierwsze wolne miejsce (na ostatnim poziomie lub tworzymy nowy poziom od lewej).
- ▶ Wykonujemy heapify-up() na dodanym węźle.
- ▶ Czas  $O(\log n)$  (zamortyzowany dla tablicy dynamicznej).

## ► extract-max()

- ▶ Wstawiamy ostatni element w miejsce korzenia (zapamiętując stary korzeń).
- ▶ Wykonujemy heapify-down() na nowym korzeniu.
- ▶ Zwracamy zapamiętany stary korzeń.
- ▶ Czas  $O(\log n)$ .



## Kopce (9)

- ▶ `find-max()` – zwracamy korzeń, czas  $O(1)$ .
- ▶ `find()` – przeszukanie (przejście) drzewa, pesymistycznie  $O(n)$ .
- ▶ `delete()`
  - ▶ Znalezienie węzła  $e$  do usunięcia.
  - ▶ Zamienienie węzła miejscami z ostatnim.
  - ▶ Zmniejszenie rozmiaru tablicy.
  - ▶ Wykonanie `heapify-up()` lub `heapify-down()` do przywrócenia własności kopca (jeśli potrzebne).
  - ▶ Pesymistyczny czas  $O(n)$ .



# Kopce (10)

- ▶ increase-key()
  - ▶ Znajdź węzeł  $e$  do zmiany.
  - ▶ Zwiększ klucz.
  - ▶ Wykonaj heapify-up() na  $e$  (zakładając kopiec typu max).
- ▶ decrease-key()
  - ▶ Znajdź węzeł  $e$  do zmiany.
  - ▶ Zmniejsz klucz.
  - ▶ Wykonaj heapify-down() na  $e$  (zakładając kopiec typu max).



# Kopce (11)

Wrocław  
University  
of Science  
and Technology

- ▶ Jedna z operacji dla wielu struktur danych jest wypełnienie pustej struktury (dodanie po kolei wielu elementów).
- ▶ Dla  $n$  elementów zwykle polega to na  $n$ -krotnym wykonaniu operacji typu `insert()`.
- ▶ Dla kopca (binarnego zupełnego) nazywamy to budowaniem kopca.
- ▶ Naiwne podejście oznacza czas  $n \cdot O(\log n) \in O(n \log n)$ .
- ▶ Istnieje jednak szybszy sposób.



## Kopce (12)

- ▶ Najpierw dodajemy wszystkie elementy w dowolny sposób, nie dbając o właściwość kopca.
  - ▶ Najprościej jest zaalokować miejsce na  $n$  elementów i umieszczać element  $i$ -ty w indeksie  $i$ .
- ▶ Następnie w pętli dla indeksów od  $\lfloor \frac{n-1}{2} \rfloor$  do 0 wykonujemy heapify-down (zakładając kopiec typu max).
  - ▶ Procedura działa od dołu w górę (bottom-up).
  - ▶ Elementy na indeksach dalej niż  $\lfloor \frac{n-1}{2} \rfloor$  są liścimi – kopcowanie ich nie ma sensu.
- ▶ Z pomocą pewnego zbieżnego szeregu można wykazać, że taka operacja zajmuje pesymistycznie i średnio czas  $O(n)$ .



# Kopce (13)

Zastosowanie kopków:

- ▶ Sortowanie przez kopcowanie.
- ▶ Algorytmy selekcji.
- ▶ Kolejki priorytetowe.
- ▶ Algorytmy grafowe.
  - ▶ Algorytm Dijkstry.
  - ▶ Algorytm Prima.



# Kopce (14)

Kopce umożliwiają wydajną implementację kolejek priorytetowych:

- ▶ `insert()` –  $O(\log n)$ .
- ▶ `extract-max()` –  $O(\log n)$ .
- ▶ `find-max()` –  $O(1)$ .
- ▶ `modify-key()` – wciąż koszt  $O(n)$  (konieczność znalezienia węzła).
  - ▶ Możemy dodatkowo przechowywać słownik mapujący indeks na węzeł.
  - ▶ Koszt operacji spada do  $O(\log n)$ , ale stała wzrasta.
  - ▶ Słownik oznacza konieczność dodatkowej pamięci.



Wrocław  
University  
of Science  
and Technology

# Struktury danych

Wykład 6

Kopiec Fibonacciego, struktura zbiorów rozłącznych

dr inż. Jarosław Rudy





## Koszt amortyzowany – metoda księgowania (1)

- ▶ Jedną z alternatywnych metod analizy kosztu amortyzowanego jest metoda księgowania.
- ▶ Koszt zamortyzowany operacji oprócz kosztu rzeczywistego uwzględnia (nieujemny) „kredyt”.
- ▶ Jeśli operacja ma koszt zamortyzowany większy niż rzeczywisty, to zwiększa kredyt.
- ▶ Operacja może mieć koszt zamortyzowany mniejszy niż rzeczywisty, o ile w momencie jej wykonywania mamy wystarczający kredyt.
- ▶ Ponieważ kredyt jest nieujemny, obliczony w ten sposób koszt zamortyzowany ogranicza od góry koszt rzeczywisty.



## Koszt amortyzowany – metoda księgowania (2)

Przykład dodawania elementu na koniec tablicy dynamicznej:

- ▶ Każda operacja dodawania daje 1 kredyt.
- ▶ Rozważmy ciąg  $k + 1$  operacji:
  - ▶ Pierwsze  $k$  operacji nie wymaga rozszerzenia tablicy. Każda ma koszt rzeczywisty 1 i zamortyzowany 2, łącznie gromadząc kredyt wysokości  $k$ .
  - ▶ Ostatnia operacja musi rozszerzyć tablicę, więc jej rzeczywisty koszt to  $k + 1$  (czyli  $O(k)$ ).
    - ▶ Jak każde dodawanie, operacja przydziela 1 kredyt... ale możemy wykorzystać zgromadzone  $k$  kredytu.
    - ▶ Ostatecznie koszt zamortyzowany to  $1 + 1 + k - k = 2 \in O(1)$ .
  - ▶ Każda operacja w ciągu ma więc koszt zamortyzowany równy  $O(1)!$



## Koszt amortyzowany – metoda potencjału (1)

- ▶ Strukturze danych przypisujemy potencjał.
- ▶ Potencjał jest funkcją od stanu struktury.
- ▶ Potencjał mogą posiadać fragmenty struktury, potencjał całości jest wtedy sumą potencjałów części.
- ▶ Koszt zamortyzowany operacji  $i$  oblicza się jako koszt rzeczywisty plus różnica potencjałów (potencjał po operacji  $i$  i po operacji  $i_1$ ):

$$k_{\text{amortized}} = k_{\text{real}} + \Phi_i - \Phi_{i-1}. \quad (1)$$



## Koszt amortyzowany – metoda potencjału (2)

Przykład dodawania elementu na koniec tablicy dynamicznej:

- ▶ Zdefiniujmy potencjał tablicy jako  $\text{size} - \text{capacity}$ . W tym przypadku potencjał może być ujemny, ale niczemu to nie przeszkadza.
- ▶ Rozważmy ciąg  $k + 1$  operacji:
  - ▶ Każda z pierwszych  $k$  operacji ma koszt rzeczywisty równy 1, a potencjał wzrasta o 1.
    - ▶ Koszt zamortyzowany wynosi więc 2.
  - ▶ Ostatnia operacja ma koszt rzeczywisty  $k+1$ , ale potencjał spada o  $k-1$ .
    - ▶ Koszt zamortyzowany wynosi więc 2.
  - ▶ Każda operacja w ciągu ma więc koszt zamortyzowany równy  $O(1)$ !



## Koszt amortyzowany – metoda potencjału (3)

$i$	capacity	size	$\Phi_i$	$\Phi_i - \Phi_{i-1}$	$k_{\text{real}}$	$k_{\text{amortized}}$
init	4	0	-4	n/d	n/d	n/d
1	4	1	-3	1	1	2
2	4	2	-2	1	1	2
3	4	3	-1	1	1	2
4	4	4	0	1	1	2
5	8	5	-3	-3	$4 + 1 = 5$	2
6	8	6	-2	1	1	2
7	8	7	-1	1	1	2
8	8	8	0	1	1	2
9	16	9	-7	-7	$8 + 1 = 9$	2



# Kopiec Fibonacciego (1)

- ▶ Kopiec binarny pozwala na wydajną realizację kolejki priorytetowej.
- ▶ Najważniejsze operacje (insert, find-min, extract-min, decrease-key) mają złożoność  $O(\log n)$ .
- ▶ Jednak tylko operacja find-min ma czas  $O(1)$ .
- ▶ Lepszą złożoność teoretyczną oferuje kopiec Fibonacciego.
- ▶ Dalej opisujemy kopce typu min.



## Kopiec Fibonacciego (2)

- ▶ Kopiec Fibonacciego ma formę lasu złożonego z kopców.
- ▶ Kształt kopków jest mniej rygorystyczny.
- ▶ Większa elastyczność pozwala na zostawienie niektórych czynności na później (tzw. lazy approach).
- ▶ W pewnym momencie potrzeba jednak narzucić większy porządek (kosztem dodatkowego czasu operacji).



## Kopiec Fibonacciego (3)

- ▶ Stopień (liczba dzieci) każdego węzła wynosi co najwyżej  $\log n$ .
- ▶ Dla węzła o stopniu  $k$  rozmiar podrzewa zakończonego w nim wynosi co najmniej  $F_{k+2}$  ( $k + 2$ -ta liczba Fibonacciego).
- ▶ Dla nie-korzenia  $x$  możemy odciąć tylko jednego syna. Gdy odcinamy kolejnego,  $x$  samo jest odcinane i staje się osobnym drzewem.
- ▶ Zwiększa to liczbę drzew, ale drzewa można łączyć podczas innej operacji.



## Kopiec Fibonacciego (4)

- ▶ Do analizy złożoności kopca Fibonacciego stosowana jest metoda potencjału.
- ▶ Węzeł  $x$  jest znaczony (marked), jeśli co najmniej 1 jego syn został odcięty od czasu gdy  $x$  został czyimś synem.
  - ▶ Korzenie nigdy nie są znaczone.
- ▶ Potencjał kopca Fibonacciego to liczba kopców plus dwukrotność liczby znaczonych węzłów:

$$\Phi = t + 2m \quad (2)$$

- ▶ Czas zamortyzowany to czas rzeczywisty plus różnica potencjałów razy pewna dobrana wartość  $C$ .



# Kopiec Fibonacciego (5)

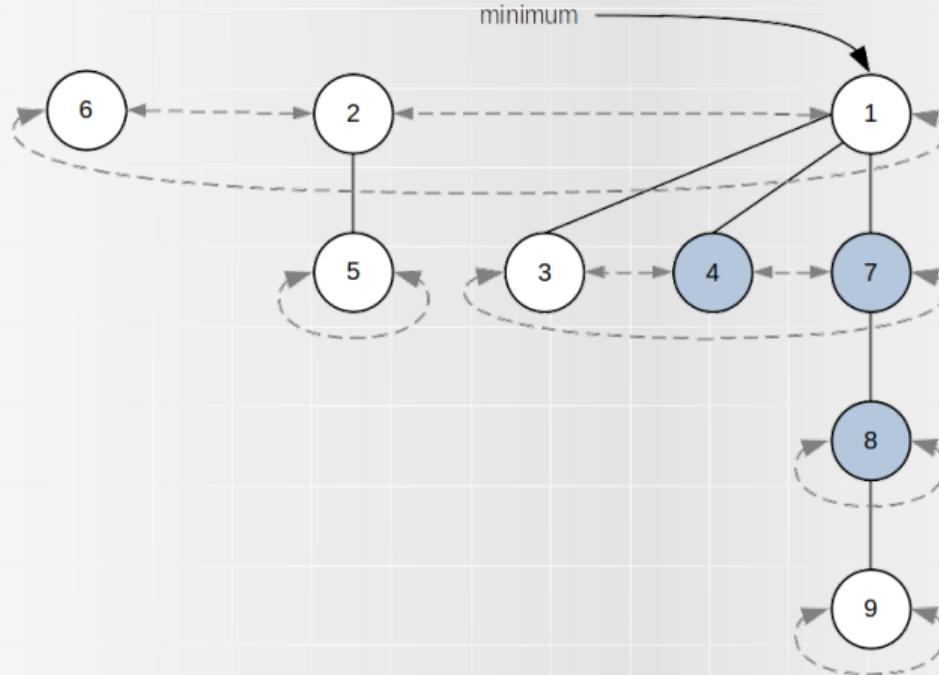
Dodatkowe założenia dotyczące realizacji struktury:

- ▶ Korzenie (las) połączone są cykliczną listą dwukierunkową.
- ▶ Rodzeństwo również połączone jest taką listą.
- ▶ Węzeł przechowuje stopień (liczbę dzieci) i to czy jest znaczony.
- ▶ Przechowujemy wskaźnik na najmniejszy korzeń.



# Kopiec Fibonacciego (6)

Przykład kopca Fibonacciego:





# Kopiec Fibonacciego (7)

Implementacja operacji:

- ▶ `find-min()` – zwracamy pamiętany wskaźnik na minimum.
  - ▶ Nie zmienia potencjału, czas zamortyzowany  $O(1)$ .
- ▶ `merge()` – operacja, którą można zdefiniować dla wielu ADT/struktur danych
  - ▶ Łączy dwie struktury (zwykle tego samego typu) w jedną.
  - ▶ Połączenie dwóch kopków Fibonacciego sprowadza się do połączenia dwóch list cyklicznych z korzeniami i wybrania nowego minimum z dwóch wartości.
  - ▶ Nie zmienia potencjału, czas zamortyzowany  $O(1)$ .



## Kopiec Fibonacciego (8)

- ▶ `insert()` – dodaje nowy 1-elementowy kopiec (sam korzeń), konieczność wyboru nowego elementu minimalnego z dwóch możliwych.
- ▶ Alternatywnie: tworzymy nowe drzewo i wykonujemy `merge()` z oryginalnym drzewem.
- ▶ Czas rzeczywisty  $O(1)$ .
- ▶ Potencjał zwiększa się o 1 (1 nowe drzewo bez węzłów znaczonych).
- ▶ Czas zamortyzowany  $O(1)$ .



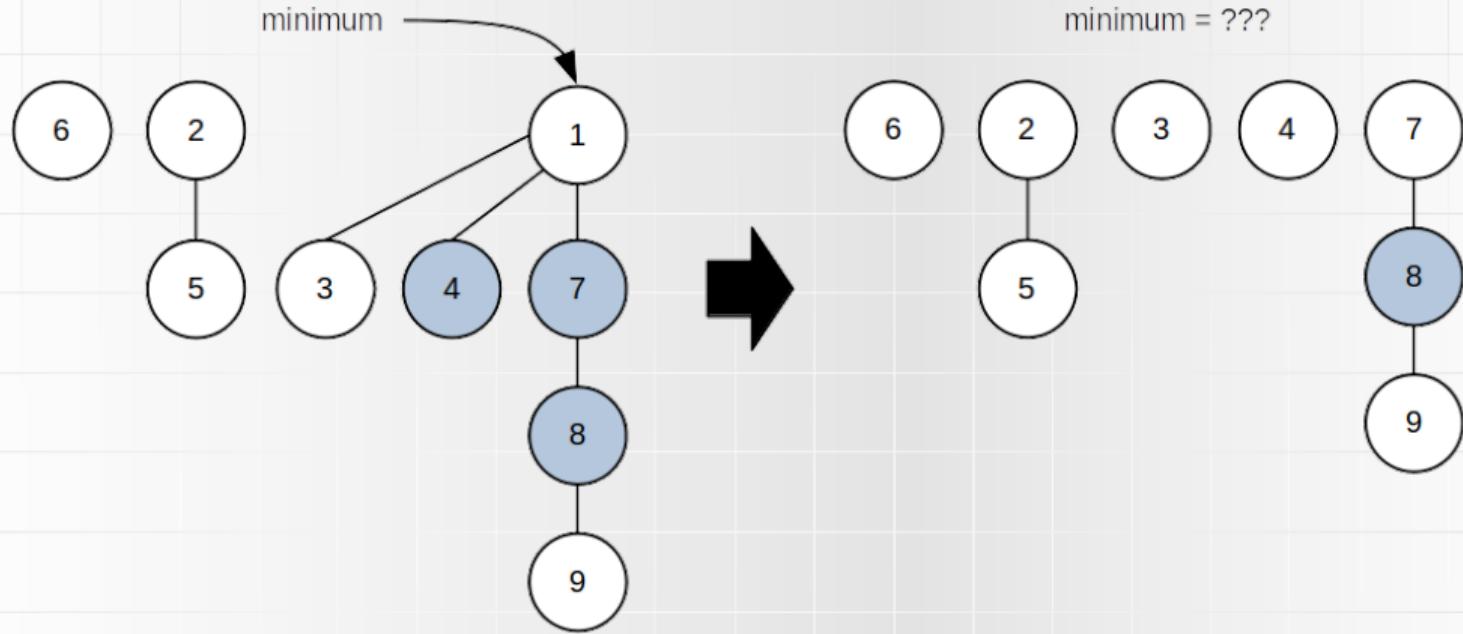
# Kopiec Fibonacciego (9)

extract-min() składa się z kilku faz:

- ▶ Faza pierwsza:
  - ▶ Usuwamy minimalny węzeł (do którego mamy wskaźnik).
  - ▶ Dzieci usuniętego węzła (który był korzeniem) stają się osobnymi drzewami.
  - ▶ Jeśli dzieci było  $k$ , to potencjał wzrasta o  $k - 1$  (1 korzeń znika,  $k$  się pojawia).
  - ▶ Czas to  $O(k) \in O(\log n)$ .



# Kopiec Fibonacciego (10)





# Kopiec Fibonacciego (11)

## ► Faza druga:

- ▶ Redukcja liczby korzeni – korzenie o tym samym stopniu są łączone (to nie jest operacja merge!).
- ▶ Jedno z łączonych drzew staje się synem drugiego (zgodnie z własnością kopca).
- ▶ Powtarzane dopóki pozostałe korzenie mają różne stopnie (będzie ich więc  $O(\log n)$ ).
- ▶ Przechowujemy tablicę rozmiaru  $O(\log n)$  przechowującą w indeksie  $i$  wskaźnik do korzenia o stopniu  $i$ , co pozwala na szybkie lokalizowanie i łączenie kopków.



## Kopiec Fibonacciego (12)

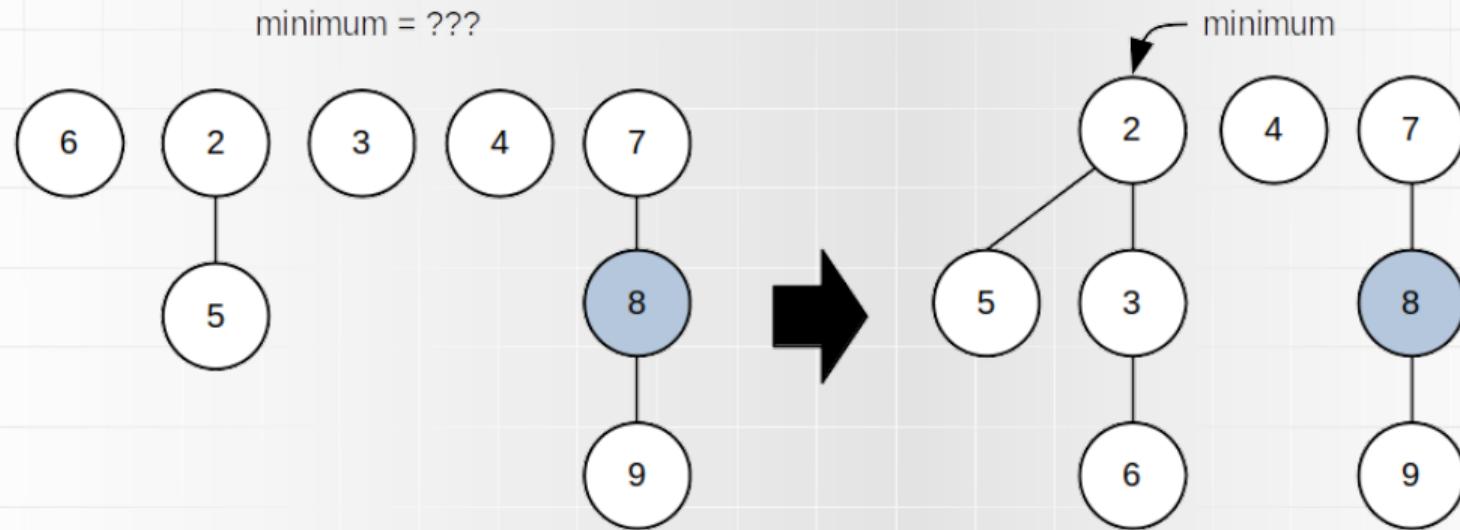
- ▶ Czas rzeczywisty:  $O(\log n + m)$ , gdzie  $m$  to liczba kopców (korzeni) na początku fazy.
- ▶ Zmiana potencjału:  $O(\log n) - m$ .
- ▶ Czas zamortyzowany:  $O(\log n + m) + C(O(\log n) - m)$ .
  - ▶ Przy odpowiednio dużym  $C$  wynik upraszcza się do  $O(\log n)$ .
- ▶ Faza trzecia – sprawdzamy wynikłe  $O(\log n)$  korzeni, by znaleźć nowe minimum.
  - ▶ Czas  $O(\log n)$ , brak zmian potencjału, więc koszt zamortyzowany również  $O(\log n)$ .

Cała operacja extract-min ma więc zamortyzowany koszt  $O(\log n)$ .



# Kopiec Fibonacciego (13)

Połączenie drzew (6) i (3), a następnie drzew (3, 6) i (2, 5).





# Kopiec Fibonacciego (13)

Operacja decrease-key():

- ▶ Znajdujemy węzeł (z odpowiednim słownikiem trwa to  $O(\log n)$ ).
- ▶ Zmniejszamy klucz węzła.
- ▶ Jeśli złamana jest zasada kopca, to odcinamy węzeł od rodzica i oznaczamy rodzica (chyba, że jest korzeniem).
- ▶ Jeśli rodzic był już oznaczony, to też go obcinamy i oznaczamy jego rodzica.
- ▶ Gdy proces się skończy (dotarliśmy do nieoznaczonego węzła), to określamy nowe minimum z dwóch wartości (zmniejszony klucz i nowe minimum).



## Kopiec Fibonacciego (13)

- ▶ Założmy, że proces utworzył  $k$  nowych drzew (korzeni).
- ▶ Wśród nich co najmniej  $k - 1$  było oznaczonych (odcięty jako pierwszy w procesie mógł nie być).
- ▶ Ponieważ stają się korzeniami, przestają być oznaczone.
- ▶ Jeden węzeł mógł zostać oznaczony.
- ▶ Dochodzi  $k$  drzew oraz  $-k + 2$  oznaczonych węzłów.
- ▶ Potencjał zmienia się więc o  $k + 2(-k + 2) = -k + 4$ .



# Kopiec Fibonacciego (14)

- ▶ Czas rzeczywisty:  $O(k)$ .
- ▶ Czas zamortyzowany:  $O(k) + C(-k + 4)$ .
  - ▶ Dla odpowiedniego  $C$  otrzymujemy  $O(1)$ .

Operacja `delete()`:

- ▶ Za pomocą `decrease-key()` ustawiamy wartość usuwanego węzła na  $-\infty$ .
- ▶ Za pomocą `extract-min()` usuwamy węzeł, który teraz stał się minimum.
- ▶ Czas  $O(1) + O(\log n) \in O(\log n)$ .



## Kopiec Fibonacciego (15)

- ▶ Kopiec Fibonacciego poprawia teoretyczną złożoność niektórych algorytmów grafowych np. algorytmu Dijkstry i algorytmu Prima.
- ▶ Dobra teoretyczna złożoność okupiona jest jednak skomplikowaną implementacją i mniejszą skutecznością praktyczną (zwłaszcza w niektórych sytuacjach).
  - ▶ Niektóre pojedyncze operacje na kopcu wykonują się długo, zaś wyższa stała zmniejsza korzyść względem  $O(\log n)$ .
- ▶ Wskazówki praktyczne:
  - ▶ Gdy nie potrzeba operacji `decrease-key()`: stosujemy kopce binarne.
  - ▶ Gdy potrzeba `decrease-key()`: stosujemy kopce parujące (pairing heaps).
    - ▶ Asymptotycznie gorsze od kopca Fibonacciego, ale bardzo dobre w praktyce i prostsze w implementacji.



# Kopiec Fibonacciego (16)

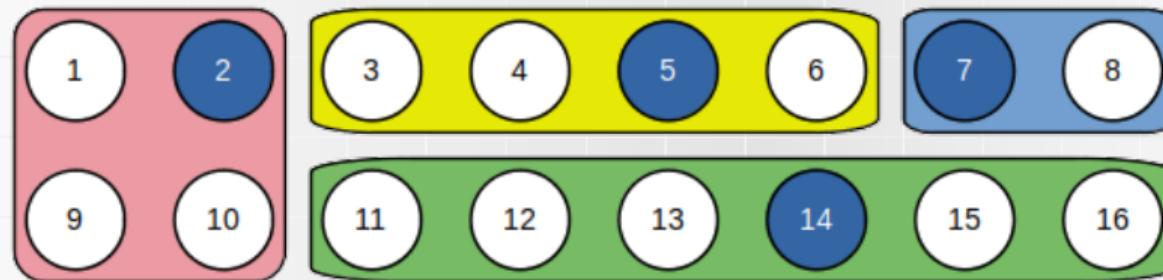
Podsumowanie złożoności niektórych odmian kopków:

Kopiec	insert()	extract-min	find-min()	decrease-key()	merge()
Binarny	$O(\log n)$	$\Theta(\log n)$	$\Theta(1)$	$O(\log n)$	$O(n)$
Dwumianowy	$\Theta(1)^a$	$\Theta(\log n)$	$\Theta(1)$	$\Theta(\log n)$	$O(\log n)$
Parujący	$\Theta(1)$	$O(\log n)^a$	$\Theta(1)$	$o(\log n)^a$	$\Theta(1)$
Fibonacciego	$\Theta(1)$	$O(\log n)^a$	$\Theta(1)$	$\Theta(1)^a$	$\Theta(1)$

<sup>a</sup> – czas zamortyzowany

# Struktura zbiorów rozłącznych (1)

- ▶ Disjoint-set data structure – przechowuje elementy pogrupowane w zbiory tak, że każdy element należy do dokładnie jednego zbioru.
  - ▶ Z matematycznego punktu widzenia przechowuje pewne rozbicie zbioru.
- ▶ Każdy zbiór ma reprezentanta.
- ▶ Stosowana między innymi w algorytmie Kruskala.





## Struktura zbiorów rozłącznych (2)

- ▶ Operacje:
  - ▶ `insert()` – dodanie nowego elementu.
    - ▶ Powstaje nowy zbiór 1-elementowy.
  - ▶ `union()` – łączenie dwóch zbiorów w jeden.
  - ▶ `find()` – znalezienie reprezentanta zbioru.
    - ▶ Jeśli dwa elementy mają tego samego reprezentanta, to należą do tego samego zbioru.



# Struktura zbiorów rozłącznych (3)

Implementacja naiwna z użyciem samych list:

- ▶  $\text{insert}(x)$  – czas  $O(1)$ .
- ▶  $\text{union}(x,y)$  –  $O(\min\{|X|, |Y|\}) \in O(n)$ .
- ▶  $\text{find}(x)$  – czas  $O(|X|) \in O(n)$ .
- ▶  $X$  i  $Y$  to zbiory do których należą odpowiednio  $x$  i  $y$ .



## Struktura zbiorów rozłącznych (4)

W praktyce używamy implementacji lasu zbiorów rozłącznych (disjoint-set forest):

- ▶ Każdy zbiór reprezentowany jest przez osobne drzewo.
- ▶ Każdy węzeł pamięta rodzica (plus stopień węzła/rozmiar poddrzewa).
- ▶ Korzeń drzewa jest reprezentantem zbioru.
- ▶ Niektóre operacje mogą być długie, ale przekształcają strukturę na korzyść przyszłych operacji, dbając by drzewa miały małą wysokość.
- ▶ Czas zamortyzowany jest bardzo dobry.
- ▶ Implementacja jest zarówno szybka praktycznie jak i optymalna asymptotycznie (gorsza najwyżej o stałą od najlepszej możliwej).



# Struktura zbiorów rozłącznych (5)

Implementacja  $\text{insert}(x)$ :

- ▶ Stworzenie elementu  $x$ .
- ▶  $x.parent \leftarrow x$ .
- ▶  $x.rank \leftarrow 0$  (lub  $x.size \leftarrow 1$ ).
- ▶ Dodanie  $x$  do listy drzew (zbiorów).
- ▶ Czas  $O(1)$ .



# Struktura zbiorów rozłącznych (6)

Implementacja `find(x)`:

- ▶ W teorii wystarczy jedynie podążać (iteracyjnie lub rekurencyjnie) za rodzicem, do korzenia i go zwrócić.
- ▶ Drzewa mogą jednak zrobić się duże, więc `find()` wykorzystuje tą okazję by zmniejszyć ich wysokość.
- ▶ Efekt uzyskuje się przez zmianę wskaźników, by zmniejszyć czas dotarcia do korzenia w kolejnych wywołaniach `find()` dla tego samego zbioru.



## Struktura zbiorów rozłącznych (7)

Algorytm kompresji ścieżek (path compression):

- ▶ Dla każdego węzła na drodze od  $x$  do korzenia root wykonujemy:
  - ▶  $x.parent \leftarrow root$
- ▶ Przypisanie wymaga znajomości korzenia, więc trzeba wykonać 2 przejścia przez ścieżkę.
  - ▶ Podejście rekurencyjne – wymaga dodatkowej pamięci (zapis ścieżki na stosie).
  - ▶ Podejście iteracyjne – dwie pętle, pierwsza znajduje korzeń, druga przechodzi ponownie ścieżkę i zapisuje go. Wymaga stałej pamięci.



## Struktura zbiorów rozłącznych (8)

Algorytm dzielenia ścieżek (path splitting):

- ▶ Dla każdego węzła na drodze od  $x$  do korzenia wykonujemy:
  - ▶  $(x, x.parent) \leftarrow (x.parent, x.parent.parent)$
- ▶ Węzeł zamiast ojca wskazywać będzie od teraz na dziadka – za każdym wykonaniem `find()` będą wskazywać bliżej korzenia.
- ▶ Pesymistycznie tak samo jak dla kompresji ścieżek, ale lepszy w praktyce.

Algorytm połowicznej ścieżki (path halving):

- ▶ Identycznie, ale zmiana jest co drugi węzeł:
  - ▶  $x.parent \leftarrow x.parent.parent$
  - ▶  $x \leftarrow x.parent$



## Struktura zbiorów rozłącznych (9)

Implementacja  $\text{union}(x, y)$ :

- ▶ Używamy  $\text{find}(x)$  i  $\text{find}(y)$  by znaleźć reprezentantów (nazwijmy ich odpowiednio  $r_x$  i  $r_y$ ).
- ▶ Jeśli  $r_x = r_y$  to  $x$  i  $y$  są w tym samym zbiorze i algorytm się kończy.
- ▶ Jeśli  $r_x \neq r_y$ , to łączymy zbiory, czyniąc jeden korzeń synem drugiego.
- ▶ Wybór bardzo wpływa na wysokość przyszłych drzew! Zapobiegamy zbyt wysokim drzewom za pomocą dodatkowych czynności.
- ▶ Konieczność przechowywania w węzłach rozmiaru drzewa (dodatkowe  $O(\log n)$  bitów) lub rangi korzenia (dodatkowe  $O(\log \log n)$  bitów).



# Struktura zbiorów rozłącznych (10)

Łączenie przez rozmiar (union by size):

- ▶ Korzeniem złączonych drzew staje się korzeń drzewa o większym rozmiarze.
- ▶ Jeśli rozmiary są identyczne, wybór jest dowolny.
- ▶ Aktualizujemy rozmiar nowego korzenia.



# Struktura zbiorów rozłącznych (11)

Łączenie przez rangę (union by rank):

- ▶ Ranga jest górnym ograniczeniem na wysokość drzewa.
- ▶ Lepszy wskaźnik niż wysokość, bo nie zmienia się w czasie `find()`.
- ▶ Jeśli łączone drzewa mają różne rangi, to ten z większą staje się rodzicem i rangi się nie zmieniają.
- ▶ Jeśli rangi są takie same, to dowolne drzewo może stać się rodzicem, ale jego rangę trzeba zwiększyć o 1.



## Struktura zbiorów rozłącznych (12)

- ▶ `find()` i `union()` mają zamortyzowaną złożoność  $O(\alpha(n))$ , gdzie  $\alpha$  jest odwrotnością bardzo szybko rosnącej funkcji Ackermanna.
- ▶ Ponieważ  $O(1) \in O(\alpha(n))$ , to dowolny ciąg  $k$  operacji na lesie zbiorów rozłącznych działa w czasie  $O(k\alpha(n))$ .
- ▶ „Fizycznie” wydaje się niemożliwe uzyskanie  $\alpha(n) > 4$ , w praktyce można więc przyjąć, że  $O(\alpha(n)) \in O(1)$ .
- ▶ Wszystkie operacje działają więc zasadniczo w czasie  $O(1)$ , ale jest to czas zamortyzowany – niektóre operacje w ciągu mogą zająć długie czas.



Wrocław  
University  
of Science  
and Technology

# Struktury danych

Wykład 7

## Słowniki, binarne drzewa poszukiwań

dr inż. Jarosław Rudy





# Słowniki (1)

- ▶ Słownik (mapa, tablica asocjacyjna) jest ADT przechowującym elementy w postaci pary klucz-wartość.
- ▶ Klucze mogą być dowolnego typu (najczęściej są to liczby lub łańcuchy tekstowe).
- ▶ Słownik mapuje zbiór kluczy na zbiór wartości (jak funkcja matematyczna).
  - ▶ Część kluczy może nie mieć wartości (funkcja częściowa).
- ▶ Generalnie każdy klucz ma co najwyżej jedną wartość.
  - ▶ Multimapa – uogólnienie słownika, w którym z jednym kluczem może być związane kilka wartości (np. kolekcja).



## Słowniki (2)

Typowe operacje na słowniku:

- ▶  $\text{insert}(k, v)$  – dodanie do słownika elementu o kluczu  $k$  i wartości  $v$ .
  - ▶ Jeśli element dla klucza  $k$  już istnieje, to zostanie nadpisany przez  $v$ .
- ▶  $\text{find}(k)$ ,  $\text{lookup}(k)$  – zwraca  $S[k]$ , czyli element mapowany przez klucz  $k$  w słowniku  $S$ .
  - ▶ Jeśli taki element nie istnieje, to najczęściej zwracana jest specjalna wartość lub podnoszony jest wyjątek.
- ▶  $\text{delete}(k)$ ,  $\text{remove}(k)$  – usunięcie elementu o kluczu  $k$ , usuwa mapowanie klucza.



## Słowniki (3)

- ▶ `exists(k)` – zwraca czy klucz  $k$  ma przypisaną wartość w słowniku.
- ▶ `size()` – zwraca rozmiar słownika.
- ▶ `empty()` – zwraca czy słownik ma jakikolwiek przypisany klucz.
- ▶ `keys()` – zwraca (enumeruje) listę przypisanych kluczy słownika lub iterator na taką listę.
- ▶ `values()` – zwraca listę wartości słownika lub iterator na taką listę.



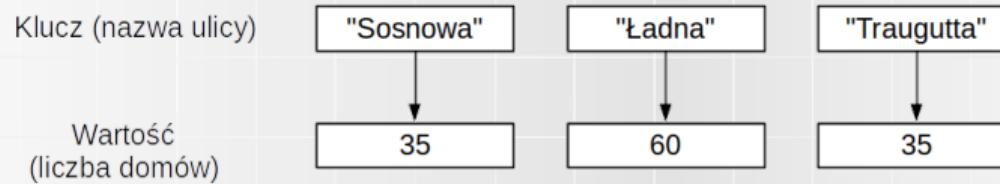
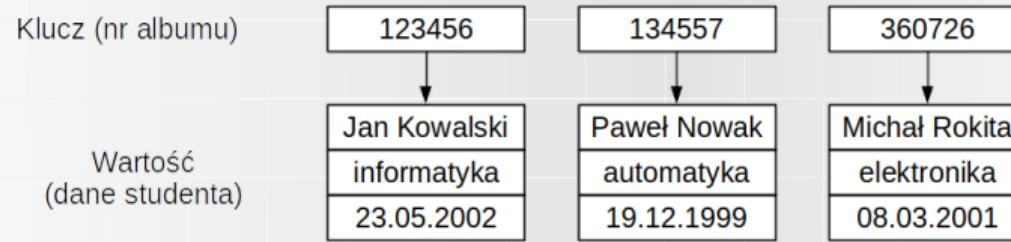
## Słowniki (4)

- ▶ Podstawowa definicja słownika nie wymaga, by na kluczach definiować porządek (relacje  $<$ ,  $\leqslant$  itd.), ani by przechowywać je w jakimś porządku.
- ▶ Niektóre z implementacji słowników będą jednak tego wymagały!
- ▶ Można jednak zdefiniować uporządkowany słownik, w którym elementy zachowują porządek przy listowaniu. Istnieją 2 powszechnie definicje:
  - ▶ Porządek określony przez sortowanie kluczy (niezależny od wstawiania).
  - ▶ Porządek określony przez kolejność wstawiania (niezależny od klucza).



# Słowniki (5)

Przykłady mapowania klucz-wartość w słownikach.





## Słowniki (6)

- ▶ Słownik zakłada szybkie wyszukiwanie elementu po zadanym kluczu.
- ▶ Dla implementacji z użyciem np. listy, średni i pesymistyczny czas wyszukiwania wyniesie  $O(n)$ .
- ▶ Listy z przeskokiem lub samoorganizujące się ulepszają przypadek średni do  $O(\log n)$ , ale pesymistycznie wciąż jest  $O(n)$ .
- ▶ Szybsze wyszukiwanie można uzyskać implementując słownik jako:
  - ▶ Binarne drzewo poszukiwań.
  - ▶ Tablicę mieszającą.



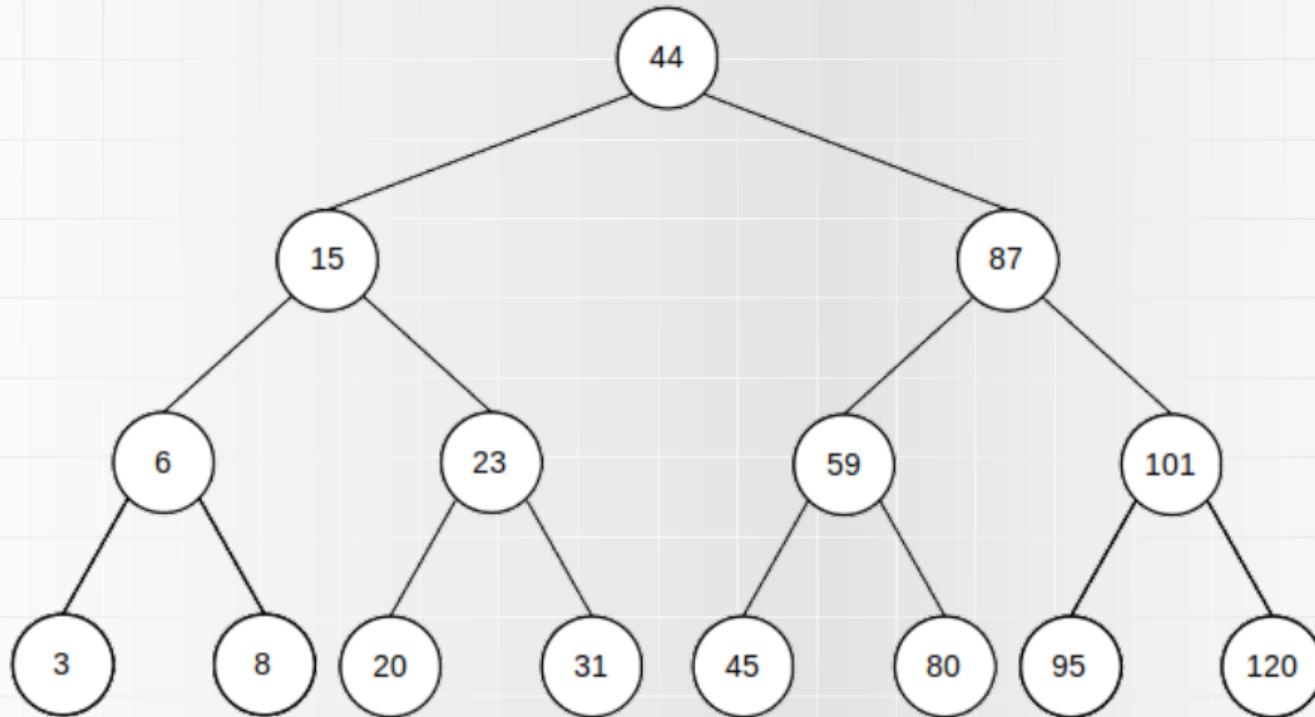
# Binarne drzewo poszukiwań (1)

Drzewo binarne (binary search tree, BST):

- ▶ Drzewo binarne w którym klucz węzła jest większy niż klucz lewego syna i mniejszy niż klucz prawego syna.
- ▶ Wymaga zdefiniowania porządku dla każdej pary kluczy.
- ▶ Wypisując BST metodą in-order otrzymamy elementy posortowane wg klucza (słownik uporządkowany).



## Binarne drzewo poszukiwań (2)





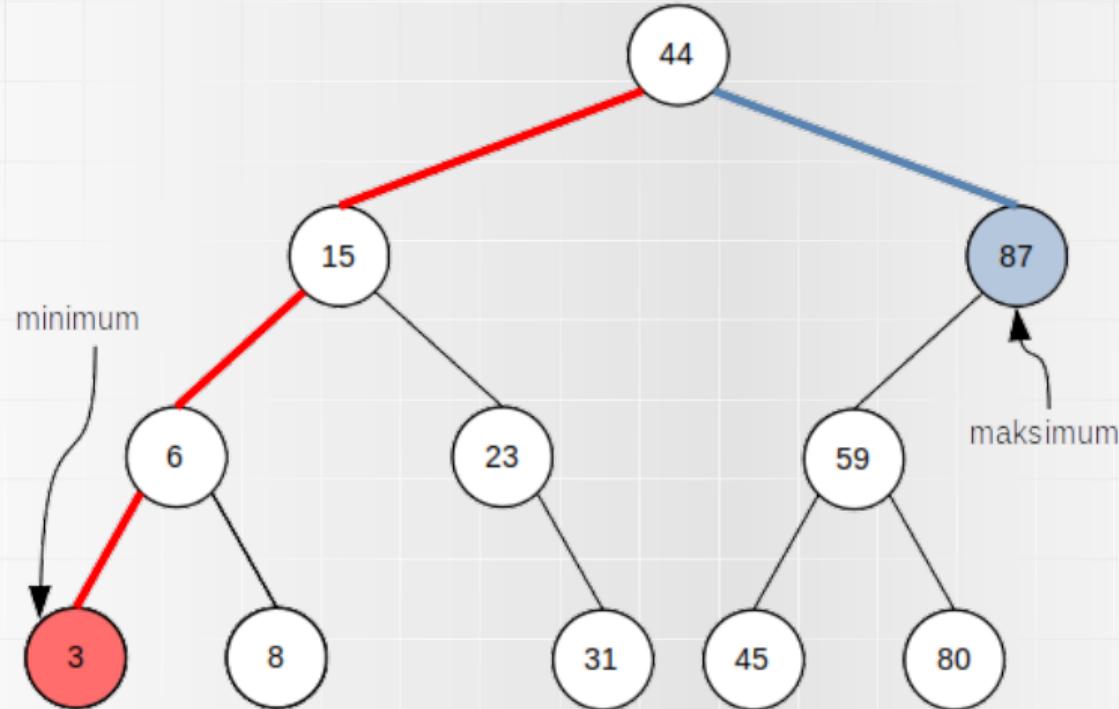
## Binarne drzewo poszukiwań (3)

Operacje pomocnicze:

- ▶ `minimum()` – znajdowanie elementu minimalnego w drzewie:
  - ▶ Podążamy lewym podrzewem dopóki istnieje.
  - ▶ Jeśli nie ma lewego syna, aktualny węzeł jest szukanym minimum.
- ▶ `maximum()` – analogicznie (podążamy prawym poddrzewem do końca).



## Binarne drzewo poszukiwań (4)



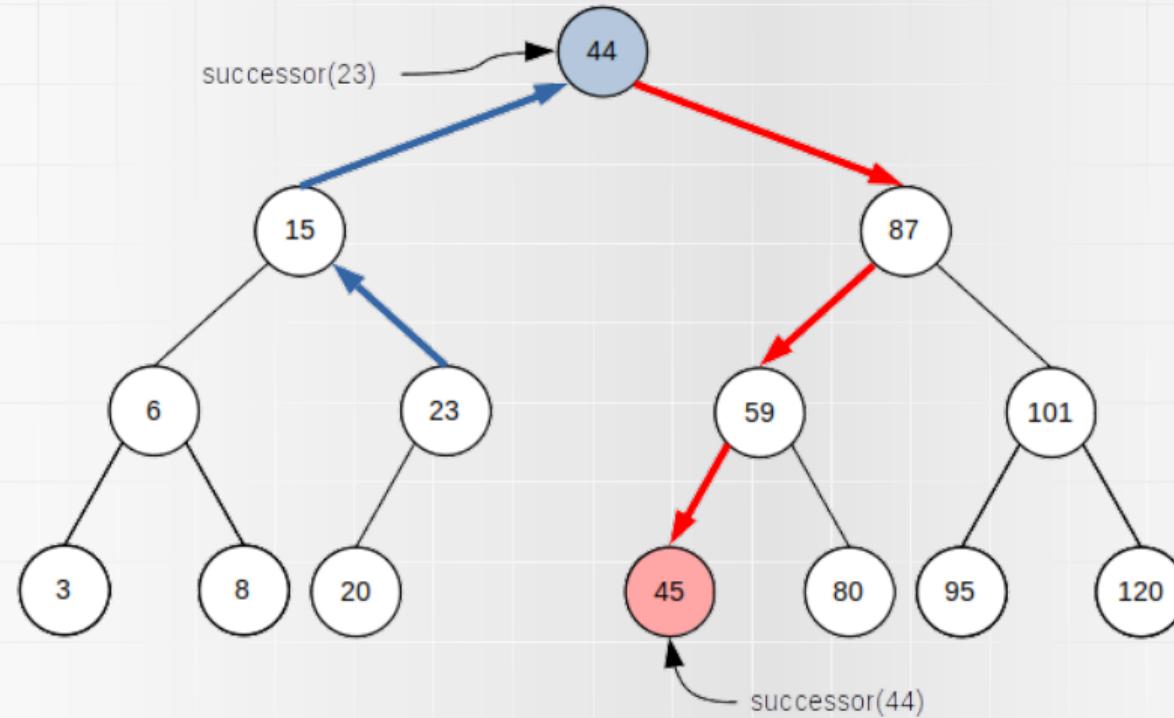


## Binarne drzewo poszukiwań (5)

- ▶  $\text{successor}(w)$  – znajdywanie następnika węzła  $w$  tj. węzła następnego w kolejności posortowanej (mającego najmniejszy klucz większy od  $w.\text{key}$ ):
  - ▶ Jeśli istnieje  $w.\text{right}$ , to  $\text{successor}(w) \leftarrow \text{minimum}(w.\text{right})$ .
  - ▶ Jeśli  $w.\text{right}$  nie istnieje, to następnikiem  $w$  jest pierwszy z przodków  $w$ , dla którego  $w$  leży w lewym poddrzewie.
    - ▶ Innymi słowy, zaczynając od  $v \leftarrow w$  przechodzimy cyklicznie do rodzica ( $v \leftarrow v.\text{parent}$ ), do czasu aż  $v.\text{parent.left} = v$ . Wtedy  $v.\text{parent}$  jest szukanym następnikiem.
- ▶  $\text{predecessor}(w)$  – znajdywanie poprzednika węzła  $w$  tj. węzła poprzedniego w kolejności posortowanej (mającego największy klucz mniejszy od  $w.\text{key}$ ):
  - ▶ Analogicznie do następnika.



# Binarne drzewo poszukiwań (6)





## Binarne drzewo poszukiwań (7)

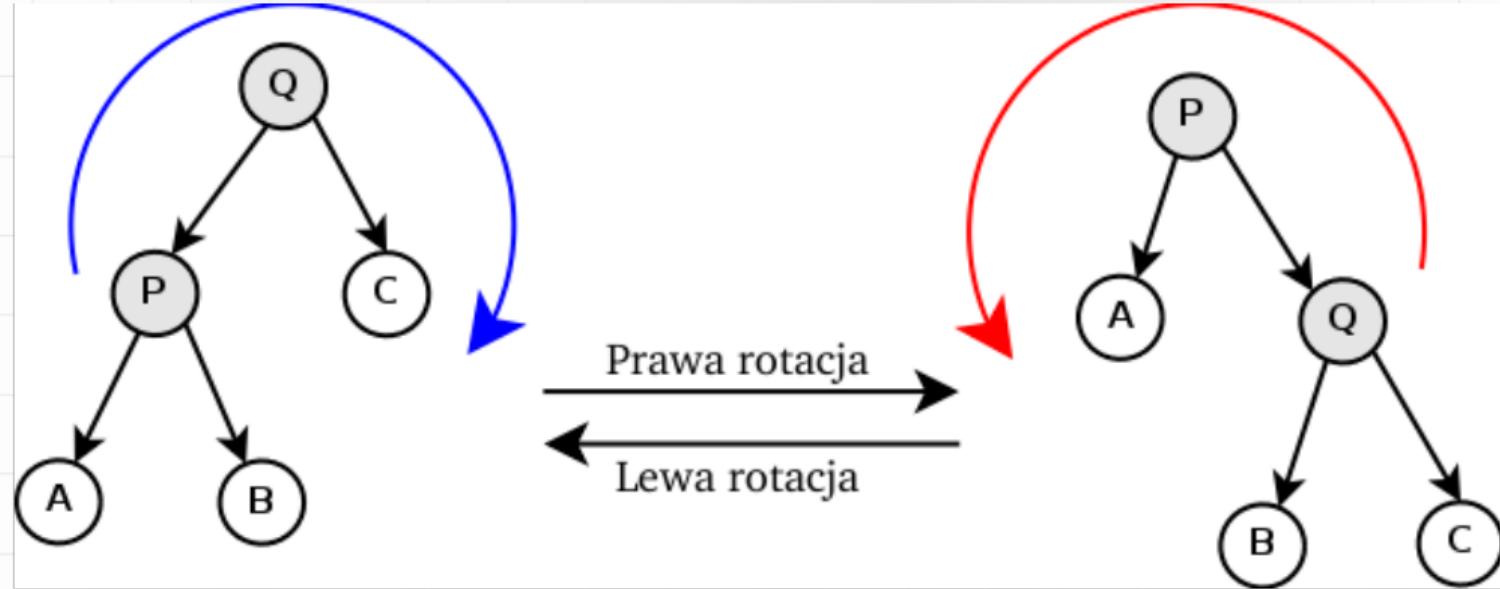
- ▶ Rotację definiujemy względem korzenia  $w$  pewnego poddrzewa (lub względem krawędzi łączącej ten korzeń go z jego synem).
- ▶ Rozróżniamy dwie podstawowe rotacje:
  - ▶  $\text{rotateLeft}(w)$ 
    - ▶  $\text{temp} \leftarrow w.\text{right}.\text{left}$
    - ▶  $w.\text{right}.\text{left} \leftarrow w$
    - ▶  $w.\text{right} \leftarrow \text{temp}$



## Binarne drzewo poszukiwań (8)

- ▶  $\text{rotateRight}(w)$ 
  - ▶  $\text{temp} \leftarrow w.\text{left}.\text{right}$
  - ▶  $w.\text{left}.\text{right} \leftarrow w$
  - ▶  $w.\text{left} \leftarrow \text{temp}$
- ▶ Rotacja zmniejsza głębokość jednego wierzchołka/poddrzewa kosztem zwiększenia głębokości innego wierzchołka/poddrzewa.
- ▶ Lewa rotacja jest operacją odwrotną do prawej rotacji i vice versa.
- ▶ Rotacje zachowują porządek kluczy (przegląd in-order).

# Binarne drzewo poszukiwań (9)



W obu przypadkach porządek kluczów to  $A < P < B < Q < C$ .



# Binarne drzewo poszukiwań (10)

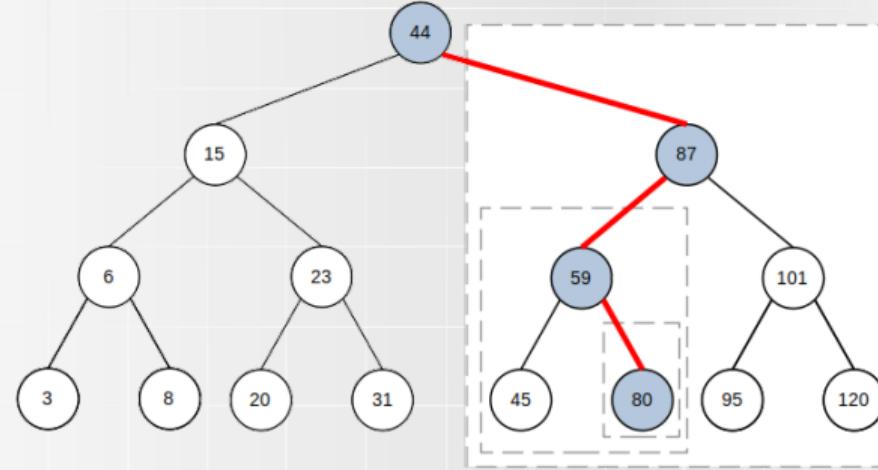
Operacja  $\text{find}(k)$ :

- ▶  $w \leftarrow \text{root}$
- ▶ Dopóki  $w$  istnieje (np. nie jest null):
  - ▶ Jeśli  $k = \text{key}(w)$ , zwracamy  $w$  (znaleziono klucz).
  - ▶ Jeśli  $k < \text{key}(w)$ , to  $w \leftarrow \text{left}(w)$  (przechodzimy do lewego poddrzewa).
  - ▶ Jeśli  $k > \text{key}(w)$ , to  $w \leftarrow \text{right}(w)$  (przechodzimy do prawego poddrzewa).
- ▶ Klucza nie ma w drzewie, zwróć odpowiednią wartość (null, wyjątek itp.).

Możliwa jest również wersja iteracyjna.

# Binarne drzewo poszukiwań (11)

Przykład szukania w BST wartości o kluczu równym 80.



Jeśli lewe i prawe podrzewa zawsze mają podobny rozmiar, to z każdą decyzją odrzucamy ok. połowę pozostałych węzłów do sprawdzenia (zasada algorytmu przeszukiwania binarnego).



# Binarne drzewo poszukiwań (12)

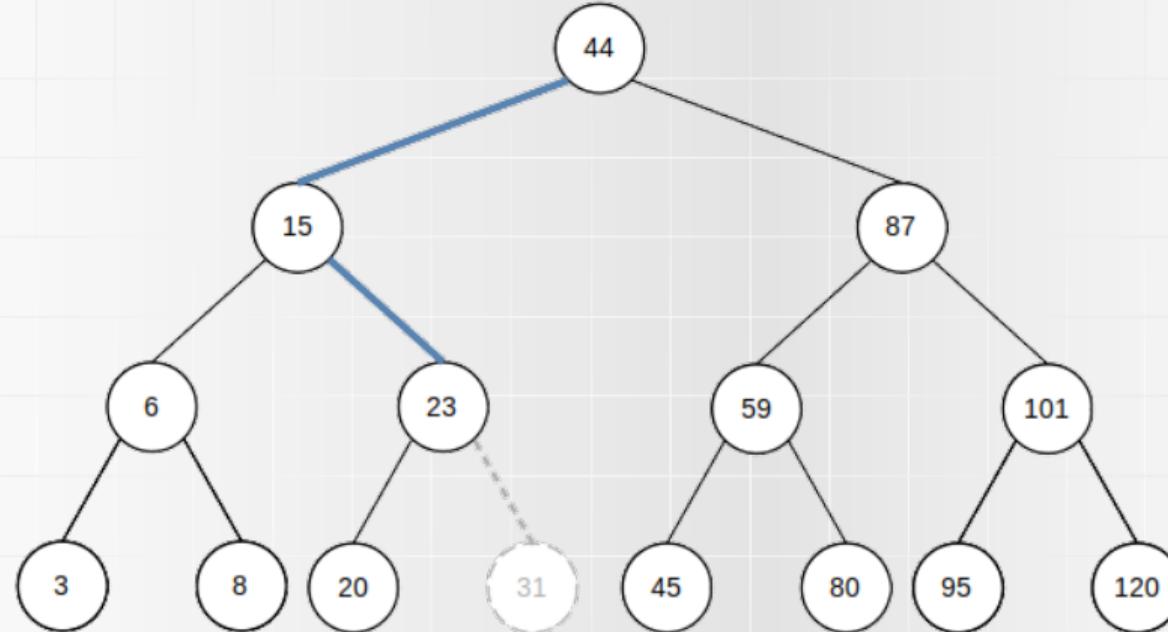
Operacja  $\text{insert}(k, v)$ :

- ▶ Tworzymy nowy element  $e = (k, v)$ .
- ▶ Jeśli drzewo jest puste, to dodajemy  $e$  jako korzeń.
- ▶ W przeciwnym razie, znajdujemy miejsce do wstawienia węzła.
  - ▶ Odbiera się to prawie identycznie jak w operacji  $\text{find}(k)$ .
  - ▶ Ponieważ  $k$  nie ma w drzewie, to natrafimy w końcu na null (brak poddrzewa w którym powinno być  $k$ ).
  - ▶ Wpisujemy  $e$  w miejsce nulla (tj. jako odpowiedni syn liścia).



# Binarne drzewo poszukiwań (13)

Przykład dodawania do drzewa węzła o kluczu 31.





## Binarne drzewo poszukiwań (14)

Operacja  $\text{delete}(k)$ :

- ▶ Znajdujemy węzeł  $w$  do usunięcia poprzez  $\text{find}(k)$ .
- ▶ Usuwamy znaleziony węzeł  $w$ .
- ▶ Konieczność reorganizacji drzewa, możliwe 3 przypadki:
  - ▶ Jeśli  $w$  nie miał synów (był liściem), to nie robimy nic.
  - ▶ Jeśli  $w$  miał jednego syna, to syn zastępuje  $w$ .
  - ▶ Jeśli  $w$  miał dwóch synów, to wstawiamy w miejsce  $w$  jego następnika.



## Binarne drzewo poszukiwań (15)

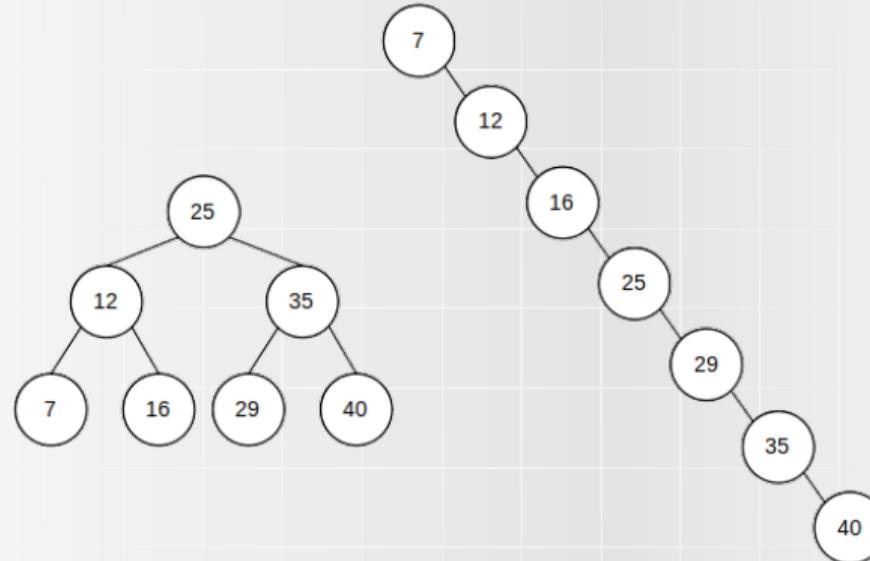
Złożoność operacji, zakładając drzewo o wysokości  $h$  z  $n$  elementami:

- ▶  $\text{insert}(k, v)$ ,  $\text{find}(k)$ ,  $\text{exists}(k)$  i  $\text{delete}(k)$ :
  - ▶ średnio  $O(\log n)$ ,
  - ▶ pesymistycznie  $O(h)$ .
- ▶  $\text{keys}()$  i  $\text{values}()$ :
  - ▶ średnio i pesymistycznie  $\Theta(n) \in O(n)$ .
- ▶  $\text{size}()$  i  $\text{empty}()$ :
  - ▶ średnio i pesymistycznie  $O(1)$ .



# Binarne drzewo poszukiwań (16)

Niekorzystne wstawianie zwiększa rozmiar drzewa. W najgorszym przypadku mamy  $h = n$  (drzewo binarne degraduje się do listy).





## Binarne drzewo poszukiwań (17)

- ▶ Możemy rozważać drzewo zrównoważone. 2 definicje:
  - ▶ Równoważenie wysokością: dla każdego węzła wysokość obu jego poddrzew różni się co najwyżej o stałą:
- ▶ Doskonale zrównoważone, jeśli wysokość różni się najwyżej o 1.
- ▶ Równoważenie wagą: dla każdego węzła jego waga różni się najwyżej o stały czynnik od wagi synów.
- ▶ Waga węzła to rozmiar jego poddrzewa plus 1.



# Binarne drzewo poszukiwań (18)

Niektóre sposoby równoważenia drzew:

- ▶ Algorytm równoważenia DSW.
- ▶ Drzewa samorównoważące się np.:
  - ▶ Drzewa AVL.
  - ▶ Drzew czerwono-czarne.
  - ▶ B-drzewa.



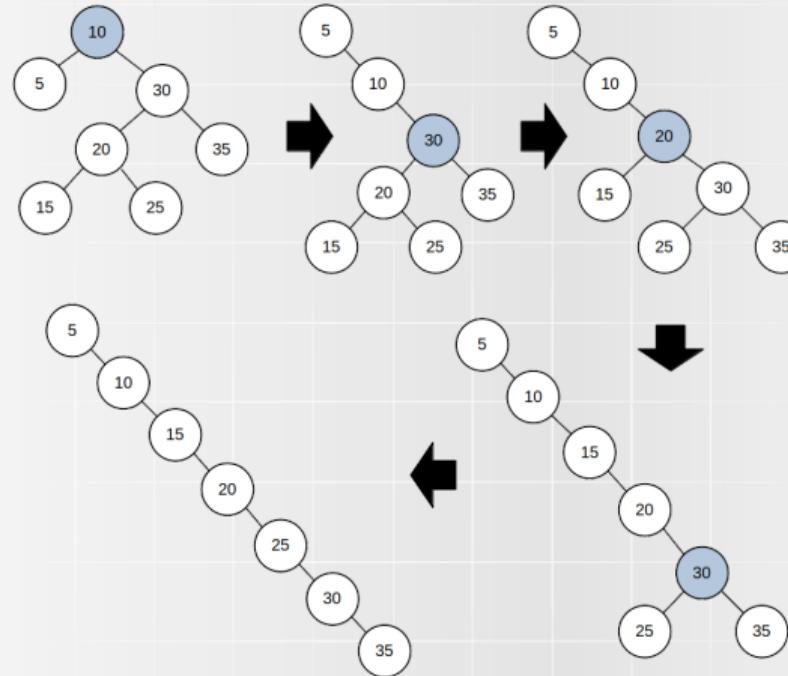
# Algorytm DSW (1)

- ▶ Algorytm równoważy zadane drzewo (gwarantując, że  $h \in O(\log n)$ ) w dwóch fazach z użyciem rotacji.
- ▶ W fazie pierwszej wykorzystywane są wielokrotne prawe rotacje, wykonywne od korzenia.
  - ▶ W wyniku drzewo zostaje zamienione w listę (tzw. kręgosłup, vine).
- ▶ W fazie drugiej iteracyjnie stosujemy lewe iteracje na co drugim węźle wzdłuż prawej gałęzi drzева.
- ▶ Zabieg ten stosowany jest wielokrotnie, za każdym razem zmniejszając wysokość drzева o połowę.

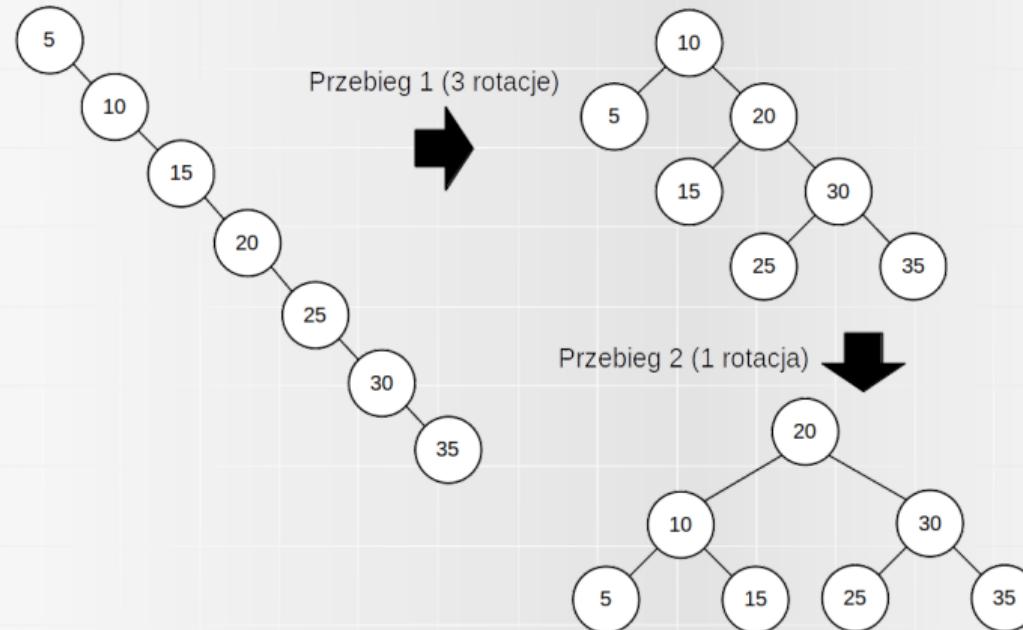


# DSW (2)

## Przykład fazy pierwszej



## Przykład fazy drugiej





## Algorytm DSW (4)

- ▶ Obie fazy działają w czasie  $O(n)$ .
- ▶ Niskie zapotrzebowanie na pamięć ( $O(1)$ ).
- ▶ Uzyskujemy drzewo doskonale zrównoważone.
- ▶ Brak potrzeby sortowania lub dalszej dekompozycji drzewa.
- ▶ Konieczność ręcznego stosowania.



## Drzewa AVL (1)

- ▶ Samorównoważące się BST.
- ▶ Każdy wierzchołek przechowuje współczynnik zrównoważenia (różnica wysokości jego lewego i prawego poddrzewa, 2 bity).
  - ▶ Wartości  $-1$ ,  $0$  oraz  $1$  są w porządku (ale zmiana może wymagać propagacji tej informacji).
  - ▶ Wartości  $-2$  oraz  $2$  wymagają naprawy poziomu wyważenia węzłów.
- ▶ Wyszukiwanie odbywa się identycznie jak dla normalnego BST, ale dzięki wyważeniu, gwarantowany jest czas  $O(\log n)$ .



## Drzewa AVL (2)

- ▶ Operacja wstawiania – początek działa jak dla zwykłego BST, po czym należy przeprowadzić proces aktualizacji wyważeń od węzła wzwyż (maksymalnie do korzenia), przy czym:
  - ▶ Wyważenie aktualizowane na 0 kończy proces bez konieczności dalszych zmian.
  - ▶ Wyważenie aktualizowane na  $-2$  lub  $2$  oznacza, że drzewo straciło właściwość AVL. Naprawa przebiega z użyciem 1–2 rotacji, po których algorytm się kończy.
  - ▶ Wyważenie aktualizowane na  $-1$  lub  $1$  wymaga aktualizacji zmiany w rodzicu (kontynuujemy algorytm).
  - ▶ Czas  $\Theta(\log n)$ .



## Drzewa AVL (3)

- ▶ Operacja usuwania – początek działa jak dla zwykłego BST, po czym należy przeprowadzić proces aktualizacji wyważeń od węzła wzwyż (maksymalnie do korzenia), przy czym:
  - ▶ Wyważenie aktualizowane na  $-1$  lub  $1$  kończy proces bez konieczności dalszych zmian.
  - ▶ Wyważenie aktualizowane na  $-2$  lub  $2$  oznacza, że drzewo straciło właściwość AVL. Naprawa przebiega z użyciem 1–2 rotacji, po czym proces należy kontynuować.
  - ▶ Wyważenie aktualizowane na  $0$  wymaga aktualizacji zmiany w rodzicu (kontynuujemy algorytm).
- ▶ Czas  $O(\log n)$ .



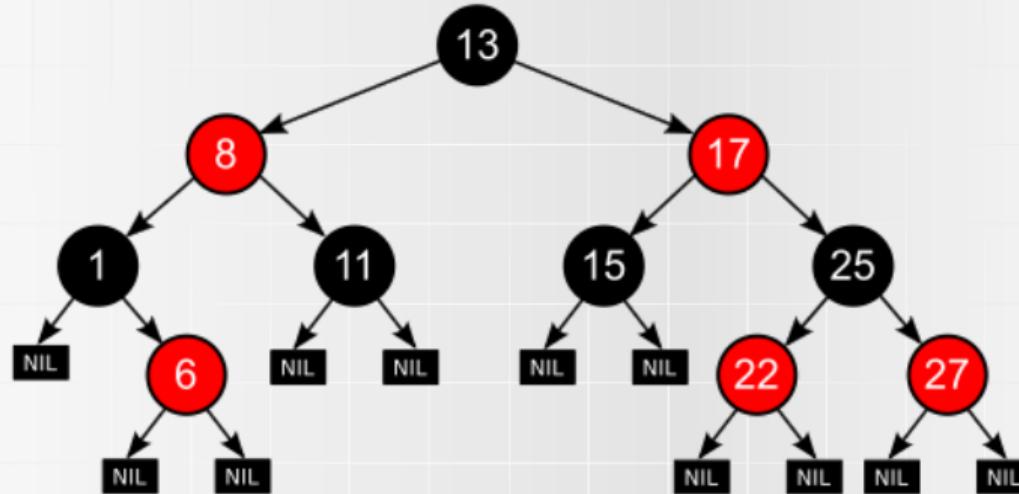
## Drzewa czerwono-czarne (1)

- ▶ Samorównoważące się BST.
- ▶ Każdy węzeł ma kolor (1 bit) a drzewo musi spełniać następujące własności:
  - ▶ Każdy węzeł jest czerwony lub czarny.
  - ▶ Korzeń jest czarny.
  - ▶ Każdy liść jest czarny.
  - ▶ Synowie czerwonego są czarni.
  - ▶ Dla węzła  $w$  ścieżki od  $w$  do jego potomków-liści mają tyle samo węzłów czarnych.
- ▶ Dla tego drzewa przez liść rozumiemy null (nil)!

Wyszukiwanie odbywa się identycznie jak dla normalnego BST, ale dzięki wyważeniu, gwarantowany jest czas  $O(\log n)$ .

# Drzewa czerwono-czarne (2)

Przykład drzewa czerwono-czarnego.



Dzięki własności, dla każdego węzła  $w$  jego najdłuższa ścieżka do liścia jest co najwyżej  $2x$  dłuższa niż najkrótsza.



## Drzewa czerwono-czarne (3)

- ▶ Operacja wstawiania:
  - ▶ Początek działa jak dla zwykłego BST.
  - ▶ Nowy węzeł kolorowany jest wstępnie na czerwono.
  - ▶ Mogły zostać złamane niektóre własności drzewa.
  - ▶ Przeprowadzamy korektę zależnie od występującego przypadku (kilka możliwych), w czasie  $O(1)$ , w tym co najwyżej jedna rotacja.
  - ▶ Czas  $O(\log n)$ .



## Drzewa czerwono-czarne (4)

- ▶ Operacja usuwania:
  - ▶ Początek działa jak dla zwykłego BST.
  - ▶ Dalsza część jest bardziej skomplikowana.
    - ▶ Możliwe jest wiele przypadków, zależnie od koloru usuwanego węzła i liczby jego dzieci.
    - ▶ Jednakże przywracanie własności drzewa czerwono-czarnego przy usuwaniu zawsze wymaga co najwyżej 2 rotacji.
  - ▶ Czas  $O(\log n)$ .



# AVL vs red-black tree

- ▶ Oba drzewa zapewniają `find()`, `insert()` i `delete()` w czasie  $O(\log n)$ .
- ▶ Drzewa AVL:
  - ▶ Są lepiej wyważone i w praktyce `find()` jest dla nich szybsze.
  - ▶ Większy koszt operacji `insert()` oraz `delete()` – możliwa konieczność przywracania własności wzduż całej wysokości drzewa.
- ▶ Drzewa czerwono-czarne:
  - ▶ Nie gwarantują doskonałego wyważenia.
  - ▶ Koszt naprawy własności drzewa dla `insert()` i `delete()` jest  $O(1)$ .
  - ▶ Wybór zależny od tego których operacji spodziewamy się więcej.



## Drzewa zrównoważone

- ▶ Zbalansowane BST zapewnia wiele operacji w czasie  $O(\log n)$ , w tym znalezienie minimum i maksimum.
- ▶ Takie BST może więc posłużyć do implementacji kolejki priorytetowej.
  - ▶ Zaletą jest możliwość implementacji jednocześnie operacji extract-min jak i extract-max (kolejka priorytetowa „dwustronna” ).
  - ▶ Wadą jest mniejsza szybkość:
    - ▶ Operacje BST mają zwykle większą stałą niż operacje kopcowe.
    - ▶ Reprezentacja kopca z użyciem tablicy dynamicznej lepiej współpracuje z pamięcią podręczną procesora niż BST.



Wrocław  
University  
of Science  
and Technology

# Struktury danych

Wykład 8

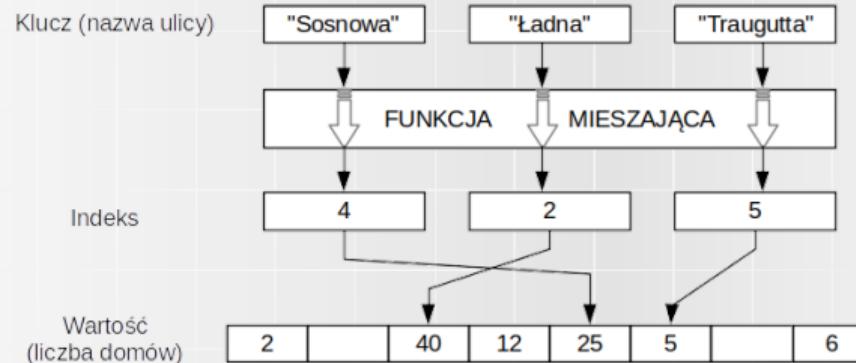
## Tablice mieszające

dr inż. Jarosław Rudy



# Tablica mieszająca (1)

- ▶ Tablica mieszająca (hash table) – struktura danych służąca do implementacji słownika.
- ▶ Za pomocą funkcji mieszającej klucz zamieniany jest na indeks.
- ▶ Indeks wykorzystywany jest do dostępu do tablicy „statycznej” .





## Funkcja mieszająca (1)

- ▶ Funkcja mieszająca  $h(x)$  przekształca klucz  $x$  na indeks elementu (kubełka) tablicy.
- ▶ Zbiór indeksów jest skończony i ma rozmiar  $m$  (nie mylić z  $n!$ ).
- ▶ Zbiór kluczy jest z reguły znacznie większy niż zbiór indeksów, może być też nieskończony.
- ▶ Z tego powodu funkcję mieszającą nazywa się też funkcją skrótu.
- ▶ Konieczność obliczenia funkcji mieszającej powoduje narzut czasowy.



## Funkcja mieszająca (2)

- ▶ Dla liczb całkowitych powszechną formą funkcji  $h(x)$  jest operacja modulo:

$$h(x) = x \bmod m \quad (1)$$

- ▶ Dzielenie jest stosunkowo powolne.
- ▶ Problem klasteryzacji.
- ▶ Wystarczająco dobre w praktyce w wielu przypadkach.
- ▶ Prostym wariantem dla łańcuchów tekstowych jest zsumowanie wszystkich znaków łańcucha.



## Funkcja mieszająca (3)

- ▶ Funkcja mieszająca z użyciem mnożenia:

$$h_\alpha(x) = \lfloor (\alpha x \bmod W)/(W/m) \rfloor \quad (2)$$

- ▶ Konieczność właściwego doboru  $\alpha$  i  $W$ .
- ▶ Haszowanie algebraiczne.
- ▶ Haszowanie Fibonacciego.
- ▶ Haszowanie z unikalną permutacją.
- ▶ Funkcja identyczności  $h(x) = x$ , sensowne jeśli zbiór kluczowy jest „mały”.



## Funkcja mieszająca (4)

- ▶ Ponieważ  $m$  jest mniejsze niż liczba kluczy, to będą istnieć takie klucze  $x_1 \neq x_2$ , że  $h(x_1) = h(x_2)$  (wynika to z zasady szufladkowej Dirichleta).
- ▶ Taką sytuację nazywamy kolizją.
- ▶ W przypadku kolizji dwa różne klucze mapowane są do tego samego miejsca (kubełka) w tablicy mieszającej.
  - ▶ Różne implementacje tablicy radzą sobie z tym w różny sposób.
  - ▶ Im mniej kolizji powoduje funkcja mieszająca tym lepiej.<sup>1</sup>

---

<sup>1</sup>W kryptografii dobiera się „większe” funkcje skrótu, przy których praktyczna szansa wystąpienia kolizji jest minimalna.



## Funkcja mieszająca (5)

### Paradoks dnia urodzin

Założymy, że w pokoju znajduje się  $k$  osób. Jakie jest prawdopodobieństwo, że co najmniej 2 z nich mają urodziny tego samego dnia?

- ▶ Dla  $k = 1$  szansa jest 0%.
- ▶ Dla  $k = 367$  szansa jest 100% (licząc rok przestępny).
- ▶ Dla jakiego  $k$  szansa jest  $\geq 50\%$ ? Naiwna interpolacja wskazała by wartość  $k \geq 183$ , ale w rzeczywistości jest to  $k \geq 23$ !
- ▶ Wniosek: pierwsza kolizja może wystąpić szybciej niż mogło by się wydawać!



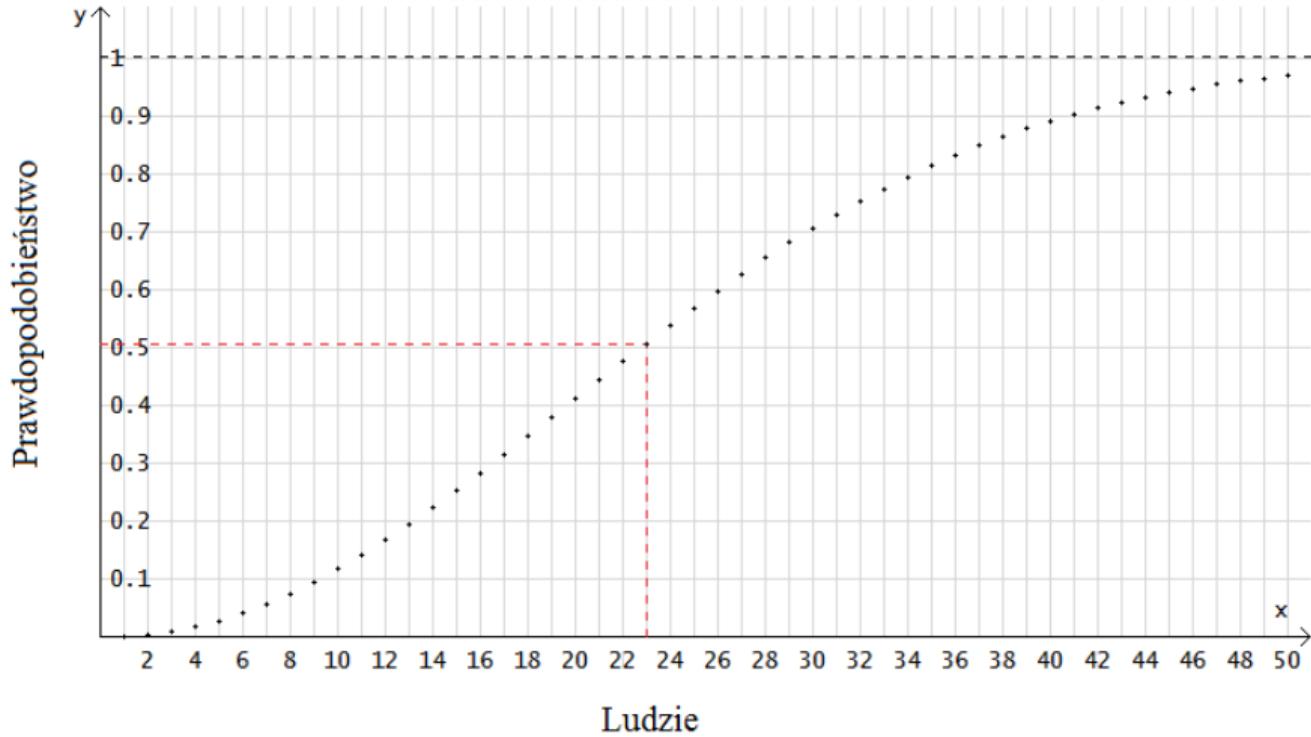
## Funkcja mieszająca (6)

Założymy tablicę mieszającą z  $m = 10^6$  i jednorodną funkcją mieszającą. Szansa na wystąpienie kolizji dla  $k$  operacji `insert()` wynosi:

- ▶ 10% przy  $k = 460$ ,
- ▶ 25% przy  $k = 759$ ,
- ▶ 50% przy  $k = 1178$ ,
- ▶ 90% przy  $k = 2146$ ,
- ▶ 99% przy  $k = 3034$  ( $0,3034\% m!$ ).



## Funkcja mieszająca (7)





## Funkcja mieszająca (8)

### Idealna funkcja mieszająca

- ▶ Przyporządkowuje każdemu kluczowi osobny indeks (kubek).
- ▶ Matematycznie jest funkcją różnowartościową.
- ▶ Brak kolizji.
- ▶ Możliwa do skonstruowania, gdy klucze znane są zawsze.
- ▶ Idealna funkcja mieszająca jest minimalna, gdy indeksy są kolejnymi liczbami całkowitymi.



## Funkcja mieszająca (9)

Cechy dobrej funkcji mieszającej:

- ▶ Deterministyczna.
- ▶ Jednorodna – każdy indeks powinien mieć zblizoną liczbę kluczy, które się na niego mapują.
- ▶ Małe zmiany klucza powinny powodować duże zmiany indeksu (redukuje problem klasteryzacji).
- ▶ Kompromis czasu obliczenia względem częstości kolizji.
- ▶ Pozwala na różny rozmiar klucza i różne wartości  $m$ .



# Rozwiązywanie kolizji (1)

Istnieją dwa podstawowe sposoby rozwiązywania kolizji:

- ▶ Metoda łańcuchowa (separate chaining).
  - ▶ Pojedynczy kubełek przechowuje wszystkie klucze, które się do niego mapują.
  - ▶ Konieczność organizacji wielu wartości w ramach kubełka.
- ▶ Adresowanie otwarte (open addressing).
  - ▶ Każdy kubełek przechowuje najwyżej 1 element.
  - ▶ Gdy drugi element zmapuje się do zajętego kubełka, to trzeba wyznaczyć mu inny kubełek.



# Współczynnik zajętości (1)

- ▶ Najważniejszym parametrem opisującym tablicę mieszającą jest współczynnik zajętości (load factor), definiowany jako:

$$\alpha = \frac{n}{m}, \quad (3)$$

gdzie  $n$  to liczba elementów przechowywanych w strukturze, a  $m$  to liczba kubełków.

- ▶ Wysokość load factor znacząco wpływa na wydajność operacji na tablicy mieszającej.



## Współczynnik zajętości (2)

- ▶ Dla open addressing dopuszczalne wartości  $\alpha$  są w przedziale  $[0, 1]$  tj.  $n \leq m$ .
  - ▶ Nie można przechować więcej niż  $m$  kluczy, bo każda wymaga osobnego kubełka.
- ▶ Gdy  $\alpha$  zbliża się do 1, coraz trudniej znaleźć wolny kubełek i wydajność drastycznie spada.
- ▶ Gdy  $\alpha$  przekracza wartość graniczną (w praktyce od 0.6 do 0.8), to należy zwiększyć rozmiar tablicy.
- ▶ Często tablicę zmniejsza się, gdy  $\alpha$  spadnie do  $\frac{1}{4}$  wartości granicznej.
  - ▶ Pozwala ograniczyć zużycie pamięci (typowo połowa lub więcej tablicy jest pusta!).

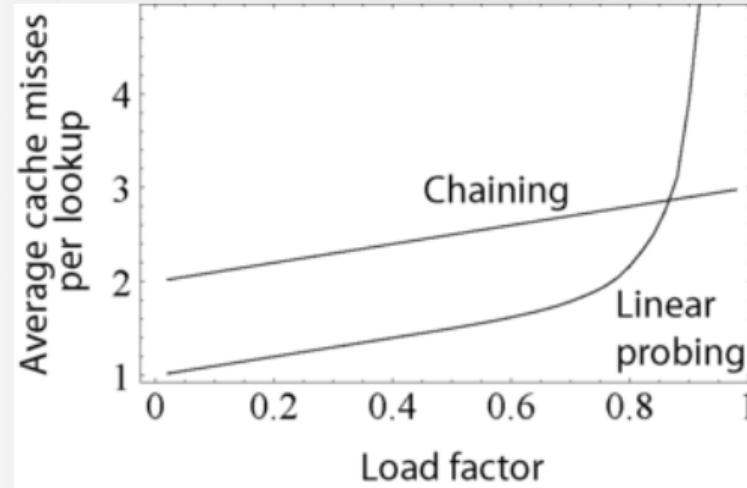


## Współczynnik zajętości (3)

- ▶ Dla separate chaining w teorii  $\alpha$  może być dowolnie duże dla stałego  $m$ .
  - ▶ Oznacza to, że średnio w jednym kubełku jest  $\alpha$  kluczy.
- ▶ Wydajność jednak wciąż spada wraz ze wzrostem  $\alpha$ , więc w praktyce stosuje się „miękką” wartość graniczną (zwykle pomiędzy 1 a 3), po przekroczeniu której zwiększa się rozmiar tablicy.
- ▶ Analogicznie tablicę można zmniejszać, gdy  $\alpha$  odpowiednio spadnie.

## Współczynnik zajętości (4)

Metody open addressing (np. linear probing) mogą średnio wymagać mniej czasu niż separate chaining, o ile  $\alpha$  nie jest bliskie 1.



W praktyce dobrze zarządzana tablica mieszająca typu open addressing rzadko potrzebuje przejrzeć więcej niż 3 kubelki.



## Zmiana rozmiaru (1)

- ▶ Zmiana rozmiaru tablicy oznacza zmianę liczby kubełków  $m$ , a to z kolei zmienia dopuszczalny zakres kluczy.
- ▶ Potrzebna jest więc nowa funkcja mieszająca, zaś operacja nie jest oczywista.
- ▶ Typowe metody zwiększenia rozmiaru:
  - ▶ Zmiana jednorazowa.
  - ▶ Zmiana stopniowa.
  - ▶ Linear hashing.



## Zmiana rozmiaru (2)

Zmiana jednorazowa:

- ▶ Dana jest stara tablica mieszajaca z funkcja mieszająca  $h_{\text{old}}(x)$ .
- ▶ Tworzymy nową (większą) tablicę mieszającą.
- ▶ Tworzymy nową funkcje mieszającą  $h_{\text{new}}(x)$ .
- ▶ Dla każdego zajętego kubełka w starej tablicy dodajemy jego zawartość do nowej tablicy (z użyciem  $h_{\text{new}}(x)$ ).
- ▶ Zajmuje dużo czasu.
  - ▶ Jeśli nowa tablica jest o czynnik (np. 2x) większa od starej, to zamortyzowany koszt zwiększenia jest stały (analogicznie jak dla tablicy dynamicznej).



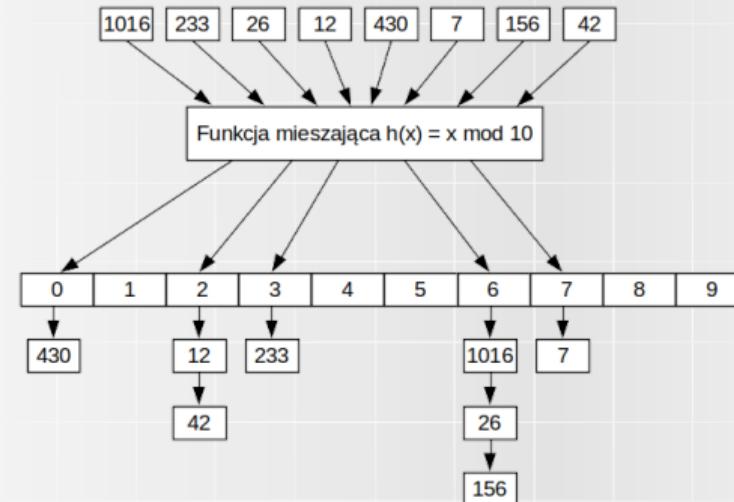
## Zmiana rozmiaru (3)

Zmiana stopniowa:

- ▶ Tworzymy nową tablicę i funkcję mieszającą  $h_{\text{new}}(x)$ .
- ▶ Na czas przenosin stosowane są obie tablice:
  - ▶ `find()` przeszukuje obie tablice (używając  $h_{\text{old}}(x)$  oraz  $h_{\text{new}}(x)$ ).
  - ▶ `insert()` dodaje do nowej tablicy (używając  $h_{\text{new}}(x)$ ).
    - ▶ Każdy `insert()` przenosi też  $k$  wpisów ze starej tablicy do nowej.
- ▶ Gdy przeniesione zostaną wszystkie elementy, starą tablicę można usunąć.

# Separate chaining (1)

- ▶ Domyślnie separate chaining w każdym kubełku przechowuje listę wiązaną elementów.
- ▶ Operacje `find()`/`insert()`/`delete()` muszą operować też na liście.





## Separate chaining (2)

- ▶ Lista wiązana ma pesymistyczny czas wyszukiwania  $O(n)$ , rzutujący na złożoność całej tablicy.
  - ▶ Możemy poprawić stosując zbalansowane drzewa BST.
- ▶ Lista wiązana słabiej współpracuje z pamięcią podręczną procesora.
  - ▶ Możemy poprawić stosując tablicę dynamiczną.
  - ▶ Problem pamięci podręcznej dotyczy też samej głównej tablicy – sięgamy do „losowych” indeksów, część indeksów jest pusta.



## Metoda 2-choice hashing:

- ▶ Modyfikacja separate chaining (choć może też działać dla innych metod rozwiązywania kolizji).
- ▶ Jedna tablica, dwie funkcje mieszające  $h_1(x)$  oraz  $h_2(x)$ .
- ▶ Podczas `insert()` stosowane są obie funkcje, zaś element trafia do kubełka mającego mniej elementów.
  - ▶ Jeśli kubełki mają równy rozmiar, to wykorzystujemy ten, który wskazała funkcja  $h_1(x)$ .
- ▶ Podobnie `find()` poszukuje elementu w obu możliwych kubełkach.
- ▶ Dzięki zasadzie power of 2 choices, znaczaco zmniejszamy możliwość wystąpienia dużych kubełków.



# Open addressing (1)

Metoda open addressing:

- ▶ Przy dodawaniu elementu najpierw obliczamy standardowo indeks funkcją mieszającą.
- ▶ Jeśli wyznaczony kubełek jest zajęty, wyznaczamy kolejny kubełek do sprawdzenia.
- ▶ Jeśli nowy kubełek jest zajęty, to sytuacja się powtarza aż do znalezienia pustego kubełka.
  - ▶ Powstaje ciąg poszukiwań (probing sequence), zaś jego długość określa wydajność tablicy.
- ▶ Analogiczny proces należy przeprowadzić dla find() i delete().



## Open addressing (2)

Istnieją różne sposoby tworzenia ciągu poszukiwań:

- ▶ Linear probing – kolejny sprawdzany kubełek jest w odstępie  $C$  kubełków (często  $C = 1$ ) od poprzedniego.
- ▶ Indeks  $k$ -tego kubełka do sprawdzenia dany jest więc wzorem:

$$h(x) + kC. \quad (4)$$

- ▶ Istotne jest by funkcja mieszająca unikała klasteryzacji/grupowania, gdzie indeksy kluczów gromadzą się blisko siebie.
  - ▶ Grupowanie, nawet jeśli nie powoduje kolizji, jest szkodliwe dla adresowania otwartego (ale nie dla metody łańcuchowej).
  - ▶ Lepiej współpracuje z pamięcią podręczną procesora (dla małych  $C$ ).



## Open addressing (3)

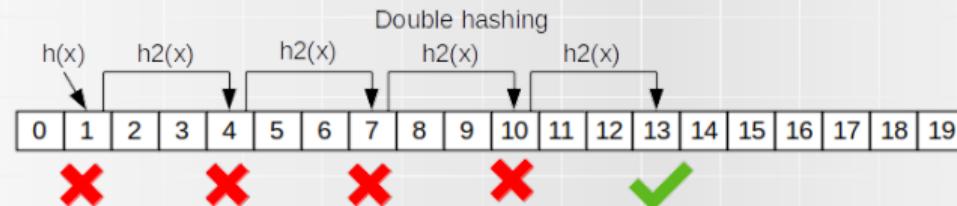
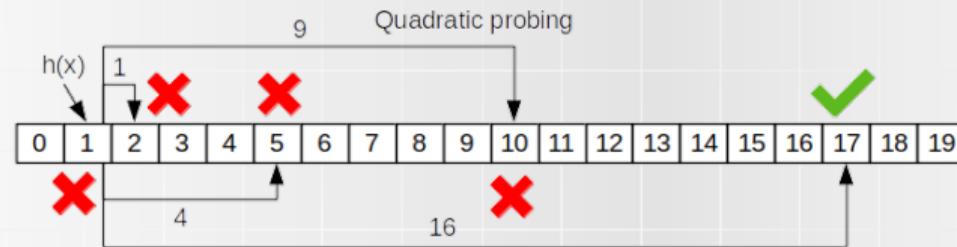
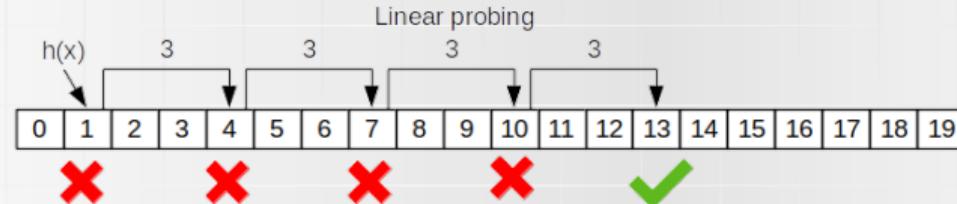
- ▶ Quadratic probing – kolejne odstępy od początkowego indeksu  $h(x)$  dane są kolejnymi wartościami pewnej funkcji kwadratowej.
- ▶ Bardziej odporne na grupowanie, podatne na grupowanie wtórne.
- ▶ Double hashing – odstęp jest liniowy, ale zależy od klucza  $x$  i dany drugą funkcją mieszającą  $h_2(x)$ . Indeks w  $k$ -tej próbie wynosi więc:

$$h(x) + kh_2(x). \quad (5)$$

- ▶ Odporne na problem grupowania.



# Open addressing (4)





# Cuckoo hashing (1)

- ▶ Haszowanie kukułcze – odmiana open addressing.
- ▶ Dwie tablice  $T_1$  oraz  $T_2$  o równym rozmiarze.
- ▶ Tablice mają osobne funkcje haszujące, odpowiednio  $h_1(x)$  oraz  $h_2(x)$ .
- ▶ Każdy klucz ma więc dwie możliwe lokalizacje: podstawową oraz alternatywną.
- ▶ Przy dodawaniu nowy element  $x_1$  wstawiany jest do  $T_1$ . Jeśli był tam już jakiś inny element  $x_2$ , to usuwamy  $x_2$  z  $T_1$  i wstawiamy go na jego alternatywne miejsce w  $T_2$ .
  - ▶ Procedura się powtarza dla  $x_2$ : jeśli miejsce  $x_2$  w  $T_2$  zajmowało  $x_3$ , to  $x_3$  jest usuwany i wstawiany w swoje alternatywne miejsce w  $T_1$ .



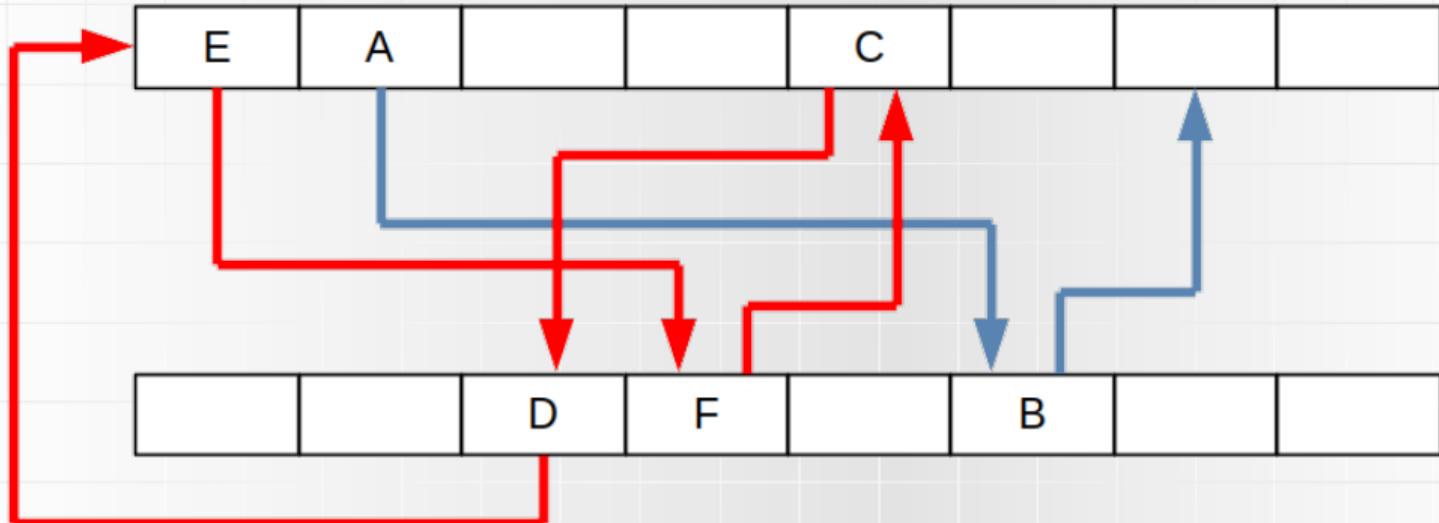
## Cuckoo hashing (2)

- ▶ Może wystąpić cykl.
  - ▶ Wykrywany przez przekroczenie licznika kroków wstawiania.
  - ▶ Naprawiany poprzez stworzenie nowych funkcji  $h_1(x)$  i  $h_2(x)$  oraz ponowne haszowanie zawartości tablic.
- ▶ Każdy klucz jest w  $h_1(x)$  w  $T_1$  lub w  $h_2(x)$  w  $T_2$ .
  - ▶ `find()` jest więc w pesymistycznym czasie  $O(1)$ , podobnie `remove()`.
  - ▶ `insert()` może zajść długi czas, ale koszt zamortyzowany jest  $O(1)$ , nawet uwzględniając konieczność przehaszowania tablic.



## Cuckoo hashing (3)

Wroclaw  
University  
of Science  
and Technology





## Cuckoo hashing (4)

- ▶ Haszowanie kukułcze ma więc bardzo dobrą złożoność teoretyczną.
- ▶ Złożoność zakłada jednak, że load factor jest poniżej 0.5.
  - ▶ Niefektywne użycie pamięci.
  - ▶ Można użyć 3 funkcji i trzech tablic, co średnio pozwala wykorzystać ponad 90% miejsca kosztem spadku wydajności (jednak wciąż  $O(1)$ ).
- ▶ W praktyce cuckoo hashing jest średnio wolniejsze od linear probing (więcej chybień podczas `find()`), ale może być przydatne w sytuacjach gdzie ważna jest redukcja przypadku pesymistycznego.



# Zastosowania

Zastosowania tablic mieszających:

- ▶ Słowniki.
- ▶ Zbiory (brak kolejności).
- ▶ Pamięci podręczne.
- ▶ Indeksy baz danych.
- ▶ Tablice transpozycji gier.



# Słowniki – podsumowanie (1)

Struktura	find()/remove()		insert()		Uporządk.
	Avg.	Worst	Avg.	Worst	
Hash table <sup>2</sup>	$O(1)$	$O(n)$	$O(1)$	$O(n)$	nie
Hash table <sup>3</sup>	$O(1)$	$O(\log n)$	$O(1)$	$O(\log n)$	nie
Hash table <sup>4</sup>	$O(1)$	$O(1)$	$O(1)$	$O(1)^5$	nie
AVL/black-red tree	$O(\log n)$	$O(\log n)$	$O(\log n)$	$O(\log n)$	tak
BST	$O(\log n)$	$O(n)$	$O(\log n)$	$O(n)$	tak
Lista <sup>6</sup>	$O(n)$	$O(n)$	$O(1)/O(n)$	$O(1)/O(n)$	„tak”

<sup>2</sup>Adresowanie otwarte lub kubełki z listą

<sup>3</sup>Adresowanie zamknięte plus kubełki ze zbalansowanym BST

<sup>4</sup>Cuckoo hashing

<sup>5</sup>Koszt zamortyzowany

<sup>6</sup>Dwa warianty: dodawanie na koniec lub dodawanie w pozycji posortowanej



Wrocław  
University  
of Science  
and Technology

# Struktury danych

Wykład 9  
Grafy

dr inż. Jarosław Rudy





## Graf (1)

- ▶ Graf jako ADT służy do reprezentacji grafów matematycznych.
- ▶ Matematycznie graf składa się z wierzchołków (węzłów, vertices) i łączących je krawędzi (edges).
  - ▶ Każda krawędź łączy dwa (domyślnie różne) wierzchołki grafu.
- ▶ Ścisłej graf  $G$  jest parą uporządkowaną (dwójką) zbioru wierzchołków  $V$  i zbioru krawędzi  $E$ :

$$G = (V, E). \quad (1)$$

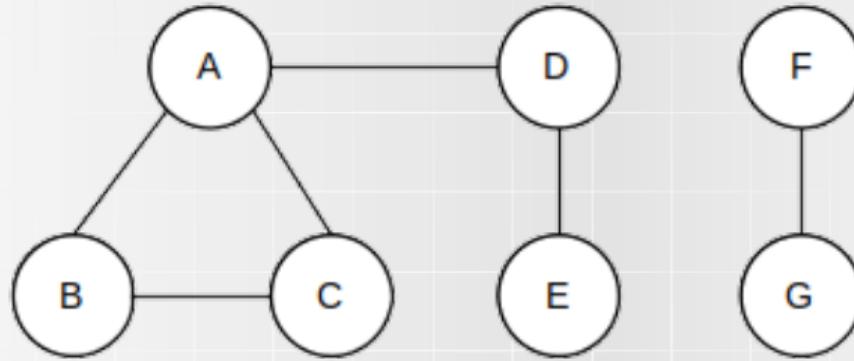
- ▶ Krawędzie zwykłe definiuje się jako zbiór  $E$  zbiorów dwuelementowych, tak że  $E$  jest podzbiorem zbioru „wszystkich” możliwych krawędzi:

$$E \subseteq \{\{u, v\} : u, v \neq u \in V\}. \quad (2)$$

krawędź  $\{u, v\}$  łączy wierzchołki  $u$  i  $v$ .

# Graf (2)

## Przykład grafu



$$V = \{A, B, C, D, E, F, G\}$$

$$E = \{\{A, B\}, \{B, C\}, \{A, C\}, \{A, D\}, \{D, E\}, \{G, F\}\}$$



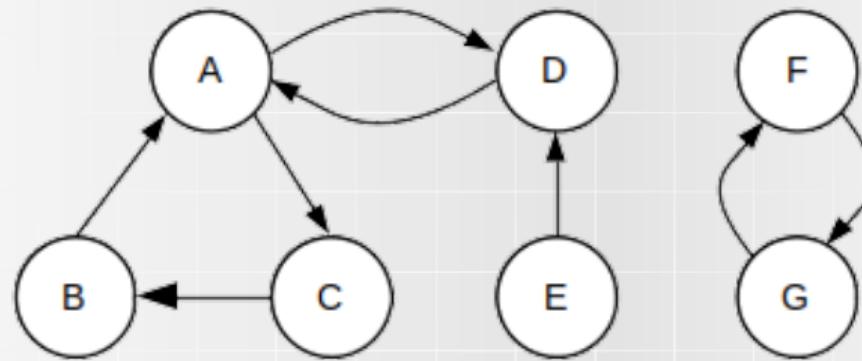
## Graf (3)

- ▶ Poprzednia definicja zakłada, że kierunek krawędzi nie ma znaczenia (graf jest nieskierowany).
- ▶ Można jednak zdefiniować zbiór krawędzi  $A$  tak by krawędzie miały kierunek tj. każda krawędź jest parą uporządkowaną

$$A \subseteq \{(u, v) : u, v \neq u \in V\}. \quad (3)$$

- ▶ Krawędź  $(u, v)$  biegnie od  $u$  do  $v$  i jest czymś innym niż krawędź  $(v, u)$ .
- ▶ Taki graf  $G = (V, A)$  nazywamy grafem skierowanym, zaś jego krawędzie nazywamy łukami (arcs).

## Przykład grafu



$$V = \{A, B, C, D, E, F, G\}$$

$$A = \{(A, C), (B, A), (C, B), (A, D), (D, A), (E, D), (F, G), (G, F)\}$$



## Graf (5)

- ▶ Często przyjmowana notacja:
  - ▶  $n = |V|$  – liczba wierzchołków grafu.
  - ▶  $k = |E|$  (lub  $k = |A|$ ) – liczba krawędzi/łuków grafu.
- ▶ Według standardowej definicji graf skierowany może mieć od 0 do  $n(n - 1)$  łuków.
- ▶ Analogicznie, graf nieskierowany może mieć od 0 do  $\frac{n(n-1)}{2}$  krawędzi.

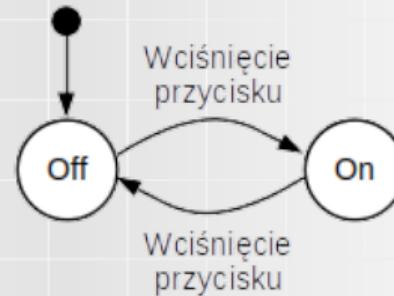
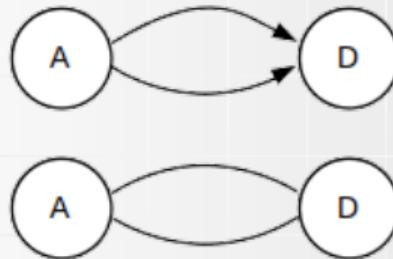
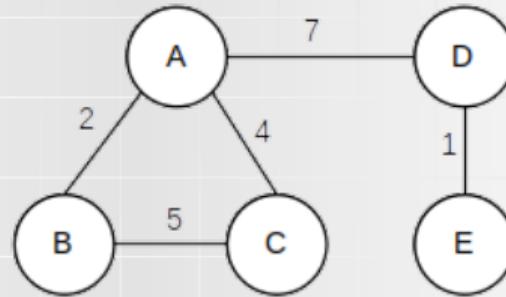
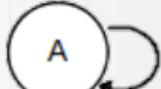


## Graf (6)

Można wprowadzać modyfikacje definicji grafu

- ▶ Dopuszczanie pętli tj. krawędzi/łuków zaczynających się i kończących się w tym samym wierzchołku.
- ▶ Multigraf – dopuszczanie wielokrotnych krawędzi pomiędzy tą samą parą wierzchołków.
- ▶ Graf z wagami – krawędziom i/lub wierzchołkom można przypisywać wartości liczbowe (wagi).
- ▶ Ogólniej można przepisywać dowolne wartości – etykiety.

## Graf (7)





# Typy grafów i pojęcia grafowe (1)

Wybrane typy grafów i pojęcia grafowe:

- ▶ Rząd grafu – liczba wierzchołków.
- ▶ Rozmiar grafu – liczba krawędzi.
- ▶ Droga – ciąg następujących po sobie krawędzi (trasa). Niekiedy w skład drogi wchodzą też znajdujące się na niej wierzchołki.
  - ▶ Droga prosta – droga na której nie powtarzają się krawędzie.
  - ▶ Cykl – droga kończąca się w tym samym wierzchołku, w którym się zaczęła.



## Typy grafów i pojęcia grafowe (2)

- ▶ Stopień wierzchołka (degree, deg):
  - ▶  $\deg(v)$  – w grafie nieskierowanym liczba krawędzi wchodzących do lub wychodzących z  $v$ .
  - ▶  $\text{indeg}(v)$  – w grafie skierowanym liczba krawędzi wchodzących do  $v$ .
  - ▶  $\text{outdeg}(v)$  – w grafie skierowanym liczba krawędzi wychodzących z  $v$ .
- ▶ Wierzchołek izolowany – wierzchołek o stopniu 0 (z którego nie wychodzą i do którego nie wchodzą krawędzie).



## Typy grafów i pojęcia grafowe (3)

- ▶ Graf pełny – liczba krawędzi jest maksymalna.
- ▶ Graf gęsty – duża liczba krawędzi ( $k$ ) w stosunku do liczby wierzchołków ( $v$ ). Różne praktyczne definicje np.:
  - ▶  $k \notin O(n)$ .
  - ▶  $k \in \Theta(n^2)$ .
  - ▶  $k \gg v$ .
- ▶ Graf rzadki – nieduża liczba krawędzi np.:
  - ▶  $k \in O(n)$ .
  - ▶  $k \leq n$ .
- ▶ Graf pusty – graf bez wierzchołków lub graf bez krawędzi.



## Typy grafów i pojęcia grafowe (4)

- ▶ Graf spójny (connected graph) – graf, w którym istnieje droga pomiędzy każdą parą wierzchołków  $u, v \neq u$ .
- ▶ Spójna składowa grafu (component) – maksymalny (tj. taki którego nie można już powiększyć) spójny podgraf grafu. Graf może mieć wiele spójnych składowych.
- ▶ Drzewo – graf nieskierowany, acykliczny (brak cykli) i spójny.
  - ▶ Dokładnie 1 droga pomiędzy każdą parą wierzchołków.



## Typy grafów i pojęcia grafowe (5)

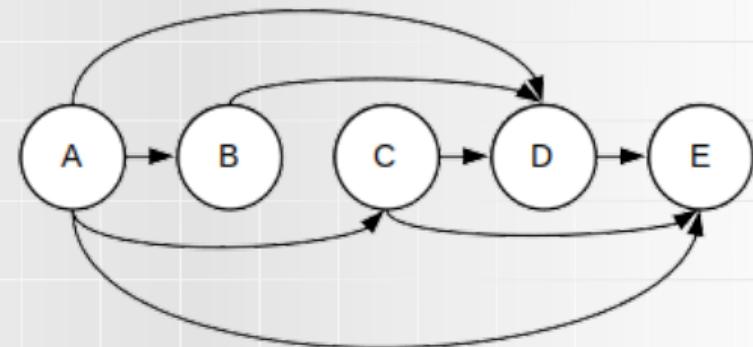
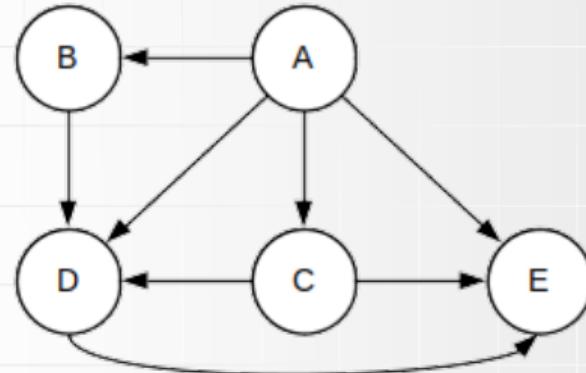
### Skierowany graf acykliczny (directed acyclic graph, DAG)

- ▶ Graf skierowany, w którym nie ma skierowanych cykli.
- ▶ W DAG podążanie krawędziami (zgodnie z ich kierunkiem) nigdy nie doprowadzi do cyklu.
  - ▶ Jeśli jest krawędź  $(u, v)$  to nie ma krawędzi  $(v, u)$ .
- ▶ Dla każdego DAG można wyznaczyć kolejność topologiczną, pewne wierzchołki są przed innymi („brak możliwości cofania się”).
- ▶ DAG są przydatne do modelowania pewnych procesów np. w produkcji.



## Typy grafów i pojęcia grafowe (6)

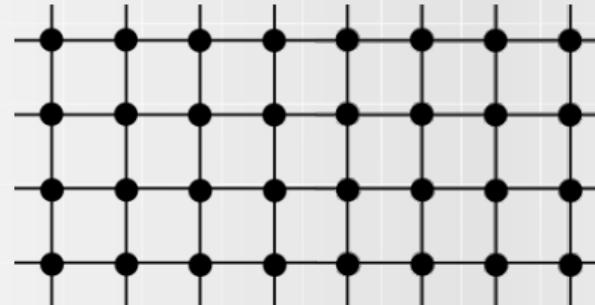
Przykładowy DAG i jego przykładowa kolejność topologiczna



## Typy grafów i pojęcia grafowe (8)

Graf typu krata/siatka (grid/lattice graph):

- ▶ Graf który po narysowaniu w Euklidesowej przestrzeni 2D ma strukturę regularnych kafelków.
- ▶ Kafelki mogą być kwadratowe, trójkątne, sześciennne itd.
- ▶ Przydatne do modelowania pewnych procesów np. w produkcji.





# Reprezentacje grafu (1)

Różne sposoby przechowywania (struktury) grafu w pamięci komputera:

- ▶ Macierz sąsiedztwa (adjacency matrix).
- ▶ Lista sąsiedztwa (adjacency list).
- ▶ Lista krawędzi (edge list).
- ▶ Macierz incydencji (incidence matrix).

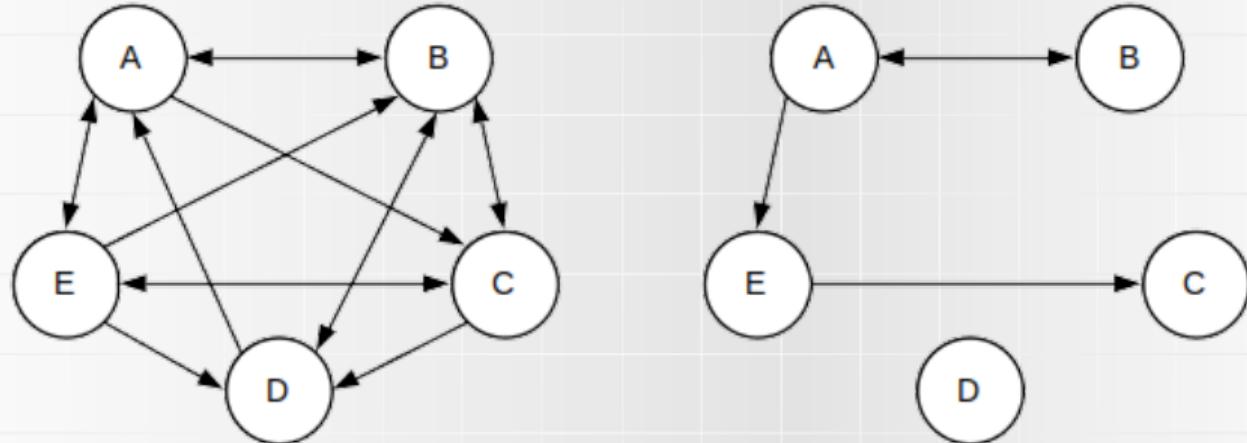
Istotne operacje:

- ▶ Dostęp do łuku  $(u, v)$  (lub krawędzi  $\{u, v\}$ ).
- ▶ Znalezienie (lub przegląd) sąsiadów wierzchołka  $v$ .

## Reprezentacje grafu (2)

Jak przykłady użyjemy dwóch grafów skierowanych:

- ▶ Gęstego  $v = 5, k = 15$  (75% krawędzi).
- ▶ Rzadkiego  $v = 5, k = 4$  (20% krawędzi).





## Macierz sąsiedztwa (1)

- ▶ Macierz dwuwymiarowa rozmiaru  $n \times n$ .
- ▶ W indeksie  $i, j$  macierzy przechowywana jest informacja o krawędzi  $(i, j)$  (lub  $\{i, j\}$ ).
  - ▶ Dla grafów bez wag/etykiet przechowujemy jedynie czy krawędź istnieje czy nie.
  - ▶ W innym przypadku przechowujemy wagę, etykietę lub wręcz wskaźnik/referencję na obiekt krawędzi (dbając o odpowiednie wartości jeśli krawędź nie istnieje).
- ▶ Dla grafów nieskierowanych  $\{u, v\} = \{v, u\}$ , więc przechowujemy tylko połowę macierzy (np. macierz górnopróbkątna).
- ▶ Realizowana tablicą dwuwymiarową, tablicami jednowymiarowymi, tablicami dynamicznymi itd.



## Macierz sąsiedztwa (2)

Dla przykładowych grafów:

	A	B	C	D	E
A	0	1	1	0	1
B	1	0	1	1	0
C	0	1	0	1	1
D	1	1	0	0	0
E	1	1	1	1	0

	A	B	C	D	E
A	0	1	0	0	1
B	1	0	0	0	0
C	0	0	0	0	0
D	0	0	0	0	0
E	0	0	1	0	0



## Macierz sąsiedztwa (3)

- ▶ Dostęp do krawędzi w czasie  $O(1)$ .
- ▶ Przegląd sąsiadów wierzchołka  $v$  w czasie  $n \in \Theta(n)$ .
  - ▶ Niezależne od tego ile wynosi  $N(v)$ , czyli liczba sąsiadów  $v$ !
  - ▶ Potrzebujemy co najmniej  $n^2$  (czyli  $O(n^2)$ ) pamięci.
    - ▶ Niezależne od  $k$ !
    - ▶ Wartości na przekątnej są bezużyteczne, chyba że graf dopuszcza pętle.
  - ▶ Dość wydajna (pamięciowo i czasowo) dla grafów gęstych.

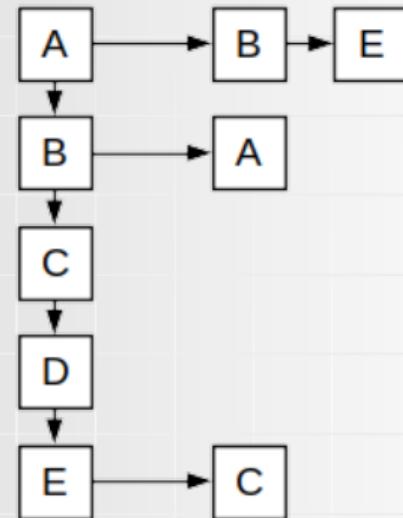
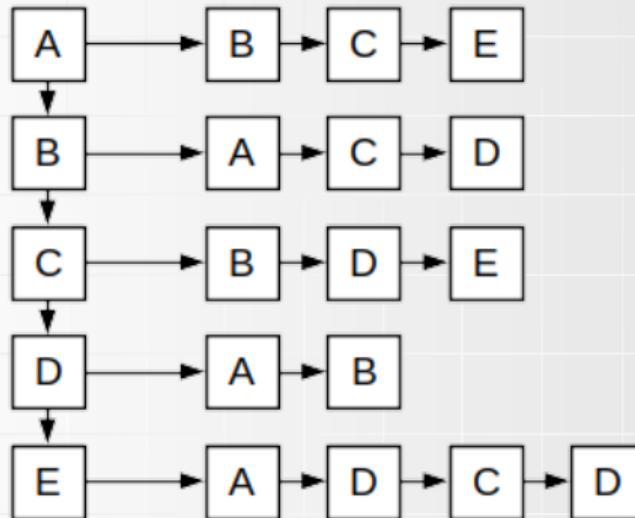


## Lista sąsiedztwa (1)

- ▶ Lista rozmiaru  $n$ .
- ▶ Elementami listy są kolejne listy.
- ▶ Lista  $i$ -ta przechowuje krawędzie zaczynające się w wierzchołku  $i$ -tym.
- ▶ Najprościej przechować numer wierzchołka którym krawędź się kończy.
- ▶ Ponieważ przechowujemy tylko faktycznych sąsiadów, to każda podlista może mieć różną długość.

# Lista sąsiedztwa (2)

Dla przykładowych grafów:





## Lista sąsiedztwa (3)

- ▶ Lista wierzchołka  $v$  ma długość  $N(v)$ , zamiast  $n$ .
  - ▶ Przegląd sąsiadów  $v$  w czasie  $O(N(v))$ .
- ▶ Zwykle mniejsze zużycie pamięci  $O(n + k)$ .
  - ▶ Wydajna dla grafów rzadkich.
  - ▶ Czasami może wymagać więcej pamięci (np. graf pełny plus dopuszczanie pętli).
- ▶ Dostęp do krawędzi wierzchołka  $v$  w czasie  $O(N(v)) \in O(n)$ .



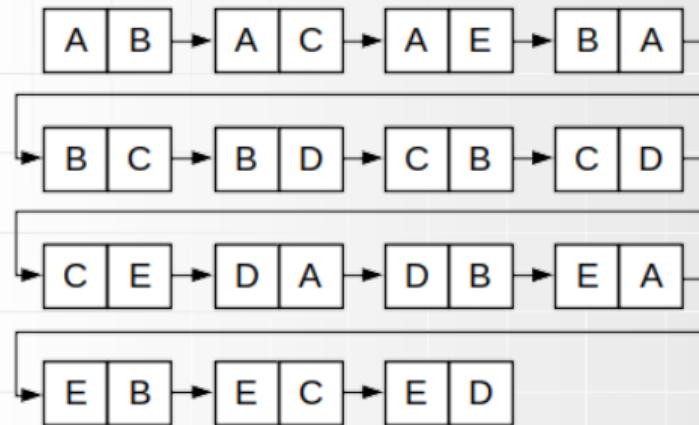
# Lista krawędzi (1)

- ▶ Lista rozmiaru  $k$ .
- ▶ Elementami listy są krawędzie.
  - ▶ Najprościej przechować krawędź jako parę: wierzchołek początkowy i końcowy.
- ▶ Zwykle krawędzie przechowywane są w losowej kolejności.
- ▶ Można przyjąć pewien porządek, ale nie ma to większego wpływu na efekt.



## Lista krawędzi (2)

Dla przykładowych grafów:





## Lista krawędzi (3)

- ▶ Lista krawędzi zajmuje  $O(k)$  pamięci.
  - ▶ Każdy element przechowuję parę, więc łączna pamięć jest zwykle większa niż dla listy sąsiedztwa.
- ▶ Dostępu do krawędzi wierzchołka  $v$  w czasie  $O(k)$ .
- ▶ Czas przeglądnięcia sąsiadów wierzchołka  $v$  w czasie  $O(k)$ .
- ▶ Zwykle skrajnie niewydajna dla grafów gęstych.
- ▶ Może być przydatna dla niektórych algorytmów (np. zmodyfikowana w kolejkę priorytetową).



# Macierz indydencji (1)

- ▶ Macierz rozmiaru  $nk$ .
- ▶ Indeks  $i, j$  przechowuje informacje czy  $j$ -ta krawędź jest incydentna (łączy się) z wierzchołkiem  $i$ . Typowe wartości:
  - ▶ 1 – krawędź zaczyna się w tym wierzchołku.
  - ▶  $-1$  – krawędź kończy się w tym wierzchołku.
  - ▶ 0 – krawędź nie jest incydentna z wierzchołkiem.
- ▶ Zwykle przechowuje tylko informacje o incydencji, ale można też dołączyć inne informacje (waga, etykieta).



## Macierz incydencji (2)

Dla przykładowych grafów:

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
A	1	1	1	-1	0	0	0	0	0	-1	0	-1	0	0	0
B	-1	0	0	1	1	1	-1	0	0	0	-1	0	-1	0	0
C	0	-1	0	0	-1	0	1	1	1	0	0	0	0	-1	0
D	0	0	0	0	0	-1	0	-1	0	1	1	0	0	0	-1
E	0	0	-1	0	0	0	0	0	-1	0	0	1	1	1	1

	1	2	3	4
A	1	1	-1	0
B	-1	0	1	0
C	0	0	0	-1
D	0	0	0	0
E	0	-1	0	1

Zakładamy kolejność krawędzi taką jak pokazano w liście krawędzi.



## Macierz incydencji (3)

- ▶ Zajmuje  $O(nk)$  pamięci.
  - ▶ Czyli rzadko mniej niż  $O(n^2)$ , nawet dla grafów rzadkich.
  - ▶  $\frac{n-2}{n}$  tablicy zawiera zera.
- ▶ Dostęp do krawędzi  $(u, v)$  w czasie  $O(k)$ .
- ▶ Przeglądnięcie wszystkich sąsiadów w czasie  $O(k + nN(v))$ .
- ▶ Przydatna gdyby krawędź miała więcej niż 2 końce.



## Reprezentacje grafu (3)

Reprezentacja	Pamięć	Dostęp do krawędzi	Przegląd sąsiadów
Macierz sąsiedztwa	$O(n^2)$	$O(1)$	$O(n)$
Lista sąsiedztwa	$O(n + k)$	$O(N(v))$	$O(N(v))$
Lista krawędzi	$O(k)$	$O(k)$	$O(k)$
Macierz incydencji	$O(nk)$	$O(k)$	$O(k + nN(v))$



## Przeszukiwanie grafu (1)

- ▶ Analogicznie jak w przypadku drzew istnieją algorytmy do przeglądu grafu tj. do odwiedzenia wszystkich wierzchołków.
- ▶ Podstawowy algorytm:
  - ▶ Dodajemy pewien wierzchołek początkowy  $s$  do kolejki  $Q$ .
  - ▶ Dopóki  $Q$  nie jest puste:
    - ▶ Pobieramy (usuwamy) jeden wierzchołek  $v$  z  $Q$ .
    - ▶ Odwiedzamy  $v$ .
    - ▶ Dodajemy nieodwiedzonych jeszcze sąsiadów  $v$  do  $Q$ .
  - ▶ Węzły mogą być odkrywane wielokrotnie.
  - ▶ Problem z grafami niespójnymi.



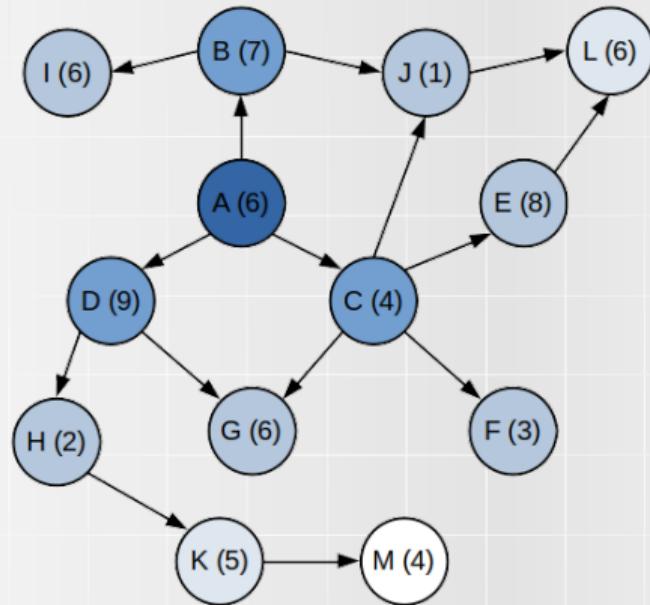
## Przeszukiwanie grafu (2)

Podstawowe typy przeglądu grafu:

- ▶ Przegląd wgłąb (depth-first) – ostatnio dodany wierzchołek odwiedzany jest najpierw ( $Q$  jest stosem).
- ▶ Przegląd wszerz (breadth-first) – ostatnio dodany wierzchołek odwiedzany jest na końcu ( $Q$  jest kolejką FIFO).
- ▶ Przegląd pierwszy najlepszy (best-first) – wierzchołki odwiedzamy wg priorytetu ( $Q$  jest kolejką priorytetową).
  - ▶ Dla grafów ważonych.
  - ▶ Priorytetem jest waga wierzchołka (na podobnej zasadzie działa algorytm Dijkstry) i/lub waga prowadzącej do niego krawędzi (np. algorytm najbliższego sąsiada dla problemu komiwojażera).

# Przeszukiwanie grafu (3)

Przykładowy skierowany graf ważony:





## Przeszukiwanie grafu (4)

Przykładowy przegląd dla przedstawionego grafu:

- ▶ Depth-first: A, D, H, K, M, G, C, J, L, F, E, B, I.
- ▶ Breadth-first: A, B, C, D, I, J, E, F, G, H, L, K, M
- ▶ Best-first: A, C, J, F, G, L, B, I, E, D, H, K, M.



# Zastosowania grafów (1)

- ▶ Transport, komunikacja (problem komiwojażera, sieci przepływowe).
- ▶ Analiza procesów (maszyny stanów, sieci Petriego).
- ▶ Modelowanie relacji/połączeń (social media, rozprzestrzenianie się chorób np. COVID-19).
- ▶ Poszukiwanie trasy.
- ▶ Optymalizacja produkcji (szeregowanie zadań).
- ▶ Inne.



Wrocław  
University  
of Science  
and Technology

# Struktury danych

Wykład 10

## Problem najkrótszej drogi

dr inż. Jarosław Rudy





# Problem najkrótszej drogi (1)

- ▶ W najprostszej wersji dany jest graf  $G = (V, E)$  oraz dwa wierzchołki: startowy  $v_s$  oraz końcowy  $v_e$ .
- ▶ Celem jest znalezienie (prostej) drogi od  $v_s$  do  $v_e$  o najmniejszej długości.
- ▶ Graf może być skierowany lub nie. Dla skierowanego istotne jest by poruszać się krawędziami w odpowiednim kierunku.
- ▶ Długość drogi liczona jest wagami krawędzi (ewentualnie krawędzi i wierzchołków).
  - ▶ Dla grafów bez wag długością drogi jest liczba jej krawędzi.
  - ▶ Droga od  $v_s$  do  $v_e$  może nie istnieć.



# Problem najkrótszej drogi (2)

Różne warianty problemu:

- ▶ Jedna para wierzchołków (single-pair shortest path problem) – znalezienie najkrótszej drogi od  $v_s$  do  $v_e$ .
  - ▶ Pesymistycznie i tak trzeba znaleźć drogę do innych wierzchołków.
- ▶ Tylko wierzchołek początkowy (single-source shortest path problem) – znalezienie najkrótszej drogi od  $v_s$  do każdego innego wierzchołka.
- ▶ Tylko wierzchołek końcowy (single-destination shortest path problem) – znalezienie najkrótszej drogi do  $v_e$  z każdego innego wierzchołka.
  - ▶ Może być sprowadzony do poprzedniego problemu (odwrócenie łuków).
- ▶ Wszystkie pary (all-pairs shortest path problem) – znalezienie najkrótszej drogi od każdego wierzchołka do każdego innego.



# Algorytm Dijkstry (1)

- ▶ Stworzony przez Edsgera Dijkstrę w 1956.
- ▶ Oryginalnie rozwiązywał single-pair shortest path problem.
- ▶ Obecnie jednak częsty wariant rozwiązuje single-source shortest path problem.
- ▶ Algorytm działa dla dowolnych skierowanych/nieskierowanych grafów ważowych bez ujemnych wag.
- ▶ Przy odpowiedniej implementacji i braku dodatkowych założeń (graf typu DAG, specyficzne wagi) jest to najlepszy znany algorytm dla tego problemu.



## Algorytm Dijkstry (2)

- ▶ Algorytm działa na zasadzie metody relaksacji – dla niektórych wierzchołków sprawdzamy czy nie da się zbudować do nich trasy krótszej niż najlepsza obecnie znana.
- ▶ Dla każdego wierzchołka  $v \in V$  przechowujemy (np. jako tablice) dwie informacje :
  - ▶  $dist[v]$  – długość najlepszej obecnie znanej drogi z  $v_s$  do  $v$  (długość tymczasowa).
  - ▶  $prev[v]$  – wierzchołek poprzedzający  $v$  na najlepszej obecnie znanej trasie od  $v_s$  do  $v$ .
- ▶ Potrzebujemy też przechowywać zbiór wierzchołków nieodwiedzonych *unvisited* oraz wierzchołek aktualny *current*.



## Algorytm Dijkstry (3)

Na początku ustawiamy:

- ▶  $unvisited \leftarrow V$ .
- ▶  $current \leftarrow v_s$ .
- ▶ Dla każdego  $v \in V$ :
  - ▶  $prev[v] \leftarrow undefined$  (np. -1)
  - ▶  $dist[v_s] \leftarrow 0$  (trasa od  $v_s$  do  $v_s$  ma znaną długość).
  - ▶ Dla każdego  $v \in V \setminus \{v_s\}$ :
    - ▶  $dist[v] \leftarrow \infty$  (np. -1)



# Algorytm Dijkstry (4)

Przebieg algorytmu:

1. Dla wierzchołka *current* sprawdzamy jego nieodwiedzonych sąsiadów i aktualizujemy ich tymczasową odległość, jeśli można do nich szybciej dotrzeć z *current*. Założymy sąsiada *neigh*:
  - ▶ Jeśli  $dist[current] + w(current, neigh) < dist[neigh]$  to:
    - ▶  $dist[neigh] \leftarrow dist[current] + w(current, neigh)$ .
    - ▶  $prev[neigh] \leftarrow current$ .
2. Usuwamy *current* z *unvisited*.
3. Jako *current* wybieramy wierzchołek *v* z *unvisited* o najmniejszym  $dist[v]$ .
4. Wracamy do punktu 1.



## Algorytm Dijkstry (5)

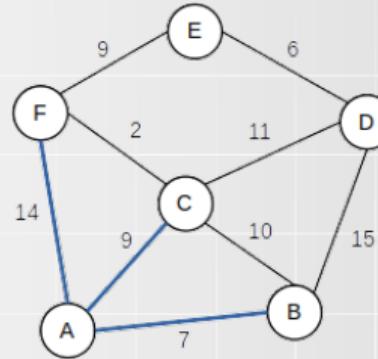
Warunek stopu algorytmu:

- ▶ Jeśli szukamy trasy od  $v_s$  do  $v_e$  algorytm kończy się, gdy:
  - ▶ W kroku 2 wierzchołek  $v_e$  został odwiedzony – możemy zbudować trasę na podstawie *prev*.
  - ▶ W kroku 3 za *current* ustawiono  $\infty$  – wierzchołek  $v_e$  jest nieosiągalny z wierzchołka  $v_s$ .
- ▶ Jeśli szukamy trasy od  $v_s$  do każdego innego wierzchołka to algorytm kończy się gdy w kroku 3 za *current* ustawiono  $\infty$ :
  - ▶ Oznacza to, że do wierzchołków obecnie w *unvisited* nie można dotrzeć z  $v_s$ . Trasy do pozostałych możemy zbudować w oparciu o *prev*.



## Algorytm Dijkstry (6)

Przykład szukania trasy od  $v_s = A$  do  $v_e = E$  (iteracja 1).



	A	B	C	D	E	F
dist	0	-1	-1	-1	-1	-1
prev	-1	-1	-1	-1	-1	-1

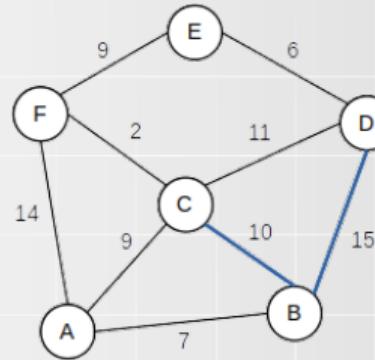
unvisited: { A , B , C , D , E , F }

current: A



## Algorytm Dijkstry (7)

Przykład szukania trasy od  $v_s = A$  do  $v_e = E$  (iteracja 2).



	A	B	C	D	E	F
dist	0	7	9	-1	-1	14
prev	-1	A	A	-1	-1	A

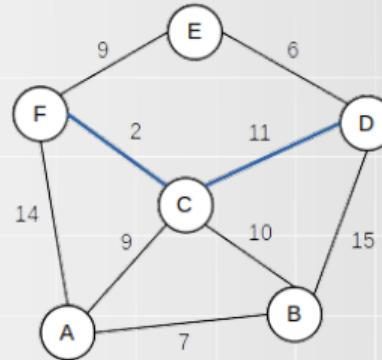
unvisited: { B , C , D , E , F }

current: B



## Algorytm Dijkstry (8)

Przykład szukania trasy od  $v_s = A$  do  $v_e = E$  (iteracja 3).



	A	B	C	D	E	F
dist	0	7	9	22	-1	14
prev	-1	A	A	B	-1	A

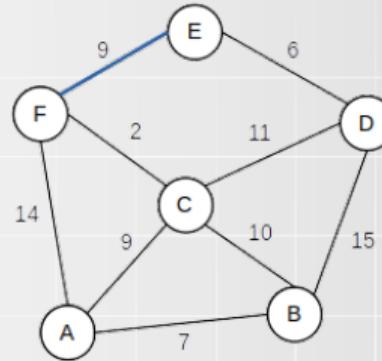
unvisited: { C, D , E , F }

current: C



# Algorytm Dijkstry (9)

Przykład szukania trasy od  $v_s = A$  do  $v_e = E$  (iteracja 4).



	A	B	C	D	E	F
dist	0	7	9	20	-1	11
prev	-1	A	A	C	-1	C

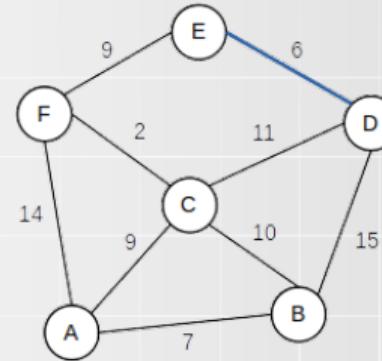
unvisited: { D , E , F }

current: F



# Algorytm Dijkstry (10)

Przykład szukania trasy od  $v_s = A$  do  $v_e = E$  (iteracja 5).



	A	B	C	D	E	F
dist	0	7	9	20	20	11
prev	-1	A	A	C	F	C

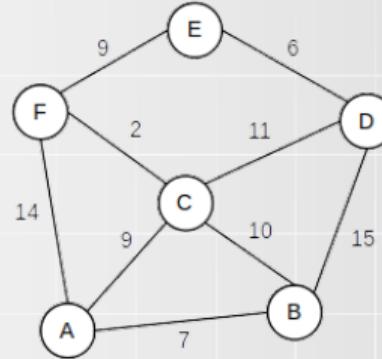
unvisited: { D , E }

current: E



# Algorytm Dijkstry (11)

Przykład szukania trasy od  $v_s = A$  do  $v_e = E$  (iteracja 6).



	A	B	C	D	E	F
dist	0	7	9	20	20	11
prev	-1	A	A	C	F	C

unvisited: { D }

koniec, trasa to (A, C, F, E)



## Algorytm Dijkstry (12)

- ▶ W kroku 3 musimy wybrać wierzchołek  $v$  o najmniejszym  $dist[v]$  spośród wszystkich wierzchołków w  $unvisited$ .
  - ▶ Potrzebujemy kolejki priorytetowej.
- ▶ Jeśli jako kolejkę użyjemy zwykłą listę i reprezentujemy graf macierzą sąsiedztwa, to pesymistyczny czas działania algorytmu wynosi  $O(n^2)$ .
- ▶ Jeśli jako kolejkę użyjemy kopca binarnego i reprezentujemy graf listą sąsiedztwa, to czas wynosi  $O((k + n) \log n)$ .
- ▶ Jeśli dodatkowo użyjemy kopca Fibonacciego to czas wyniesie  $O(k + n \log n)$ .



## Algorytm Dijkstry (13)

- ▶ Za każdym razem wybieramy lokalnie najlepszą decyzję – algorytm Dijkstry jest więc przykładem algorytmu zachłanego.
- ▶ „Rzadki” przykład algorytmu dla którego strategia zachłanna jest optymalna (otrzymana trasa jest zawsze najkrótsza).
- ▶ Algorytm Dijkstry może być też uznany za przykład programowania dynamicznego.
- ▶ „Rzadki” przykład programowania dynamicznego, które działa w czasie wielomianowym.



# Algorytm Bellmana-Forda (1)

- ▶ Jeśli graf ma ujemne wagi, to nie można użyć algorytmu Dijkstry.
- ▶ Można wtedy użyć wolniejszego, ale ogólniejszego algorytmu Bellmana-Forda.
  - ▶ Algorytm Bellmana-Forda dopuszcza obecność ujemnych wag, ale nie może być ujemnych cykli (tj. cykli o ujemnej długości) osiągalnych z  $v_s$ .
    - ▶ Po ujemnym cyklu można poruszać się bez końca uzyskując dowolnie krótką ścieżkę – najkrótsza ścieżka więc nie istnieje.
  - ▶ Algorytm rozwiązuje single-source shortest path problem.
  - ▶ Podobnie do algorytmu Dijkstry, algorytm Bellmana-Forda również polega na zasadzie relaksacji, ale nie na strategii zachłannej.



# Algorytm Bellmana-Forda (2)

Przebieg algorytmu:

1. Inicjalizacja tablicę  $dist$  oraz  $prev$  identyczna jak dla algorytmu Dijkstry.
2. Wykonujemy  $n - 1$  razy:
  - 2.1. Dla każdej krawędzi  $(u, v)$  próbujemy wykonać relaksację:
    - 2.1.1. Jeśli  $dist[u] + w(u, v) < dist[v]$  to
      - 2.1.1.1.  $dist[v] \leftarrow dist[u] + w(u, v)$ .
      - 2.1.1.2.  $prev[v] \leftarrow u$ .
  3. Wykonujemy jeszcze jedną iterację relaksacji.
    - ▶ Jeśli jakaś wartość w  $dist$  się zmniejszyła, to znaczy, że mamy ujemny cykl.



## Algorytm Bellmana-Forda (3)

- ▶ Czas działania to  $O(kn)$ .
  - ▶  $O(n^3)$  dla grafów gęstych.
  - ▶  $O(n^2)$  dla rzadkich.
- ▶ Proste usprawnienie – jeśli podczas iteracji nie dokonała się żadna relaksacja, to algorytm można zakończyć.
- ▶ Algorytm Bellmana-Forda można też wykorzystać w zadaniach, w których celem jest znalezienie ujemnego cyklu.



## Algorytm A\* (1)

- ▶ A\* rozwiązuje single-pair shortest path problem.
- ▶ Algorytm opiera się na obserwacji, że długość najkrótszej drogi od  $v_s$  do  $v_e$ , przez wierzchołek  $v$  jest równa:

$$sp(v_s, v) + sp(v, v_e), \quad (1)$$

gdzie  $sp$  jest najkrótszą drogą pomiędzy daną parą wierzchołków.

- ▶ Najlepszą obecnie znaną wartość  $sp(v_s, v)$  mamy dzięki  $dist(v)$ .
- ▶ Wartości  $sp(v, v_e)$  nie znamy... ale możemy ją oszacować za pomocą jakiejś heurystyki.



## Algorytm A\* (2)

- ▶ A\* korzysta więc z dodatkowej informacji o grafie i jako kolejny rozwija węzeł  $v$  o najmniejszej wartości funkcji  $f(v)$ :

$$f(v) = g(v) + h(v), \quad (2)$$

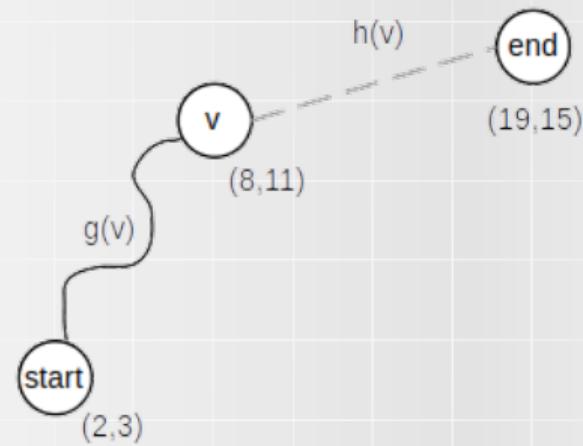
gdzie  $g(v)$  jest znaną długością drogi od  $v_s$  do  $v$ , zaś  $h(v)$  jest szacowaną długością drogi od  $v$  do  $v_e$ .

- ▶ Częstym przykładem są grafy w przestrzeni Euklidesowej, gdzie waga krawędzi  $(u, v)$  równa jest odległości Euklidesowej pomiędzy  $u$  i  $v$ .

# Algorytm A\* (3)

Przykład dla przestrzeni Euklidesowej:

$$h(v) = \sqrt{(19 - 8)^2 + (15 - 11)^2} = \sqrt{11^2 + 4^2} = \sqrt{137} \approx 11.7 \quad (3)$$





## Algorytm A\* (4)

- ▶ Jeśli heurystyka  $h(v)$  jest dopuszczalna (czyli nigdy nie przeszacowuje długości najlepszej drogi od  $v$  do  $v_e$ ), to A\* zawsze zwraca najkrótszą (optymalną) drogę od  $v_s$  do  $v_e$ , o ile taka istnieje.
- ▶ Dodatkowo jeśli  $h(v)$  jest spójna tj.:

$$h(x) \leq d(x, y) + h(y), \quad (4)$$

gdzie  $d(x, y)$  jest faktyczną odlegością między  $x$  oraz  $y$ , to nie istnieje algorytm, który dla heurystyki  $h()$  rozwija mniej węzłów niż A\* (stąd gwiazdka oznaczająca optymalność) i nigdy nie odwiedza danego węzła więcej niż raz.

- ▶ Im lepsza heurystyka (im bliżej wartości prawdziwej bez przeszacowywania), tym mniej węzłów rozwinie algorytm.



## Algorytm A\* (5)

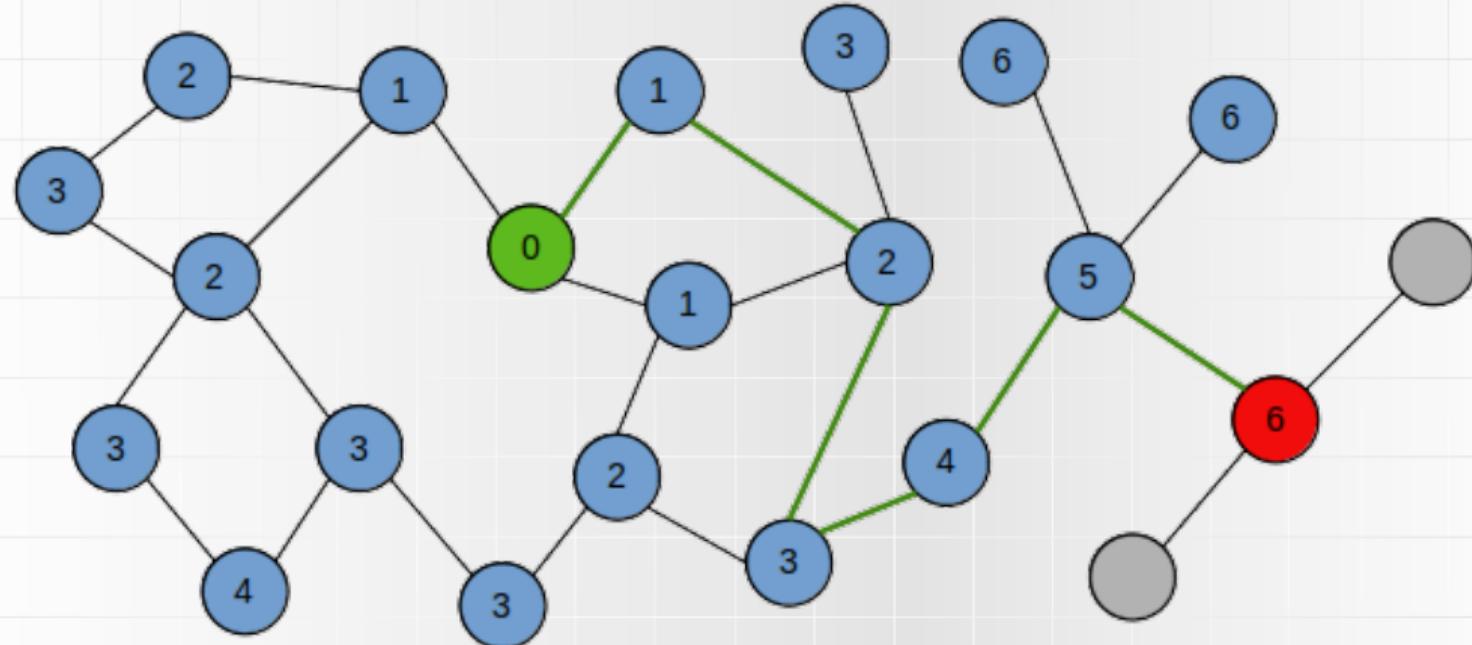
- ▶ Sposób rozwiązywania remisów wpływa na skuteczność algorytmu (poleca się by remisy traktowane były w porządku stosowym).
- ▶ Specjalne przypadki:
  - ▶ Algorytm Dijkstry (gdy  $h(v) = 0$ ).
  - ▶ Przeszukiwanie wszerz.
  - ▶ Przeszukiwanie wgłąb.
- ▶ Jeśli  $h()$  nie jest dopuszczalna, to A\* może znaleźć drogę, która nie jest najkrótsza... ale może rozwinąć mniej węzłów.
  - ▶ Kontrolując przeszacowywanie heurystyki kontrolujemy dokładność i czas działania – przydatne w niektórych zastosowaniach (np. gry).



## Poszukiwanie wszerz (1)

- ▶ W przypadku grafu bez wag trasę od  $v_s$  do  $v_e$  można wydajnie znaleźć stosując zwykłe przeszukiwanie grafu wszerz.
- ▶ Zaczynamy od  $v_s$ .
- ▶ Sąsiedzi  $v_s$  mają odległość 1.
- ▶ (Nieodwiedzeni) sąsiedzi sąsiadów mają wartość 2 itd.
- ▶ Po dotarciu do  $v_e$  budujemy trasę cofając się do  $v_s$  poprzez zawsze wybieranie dowolnego wierzchołka o wartości o 1 mniejszej od naszej.
- ▶ Złożoność pesymistyczna  $O(n + k)$ .

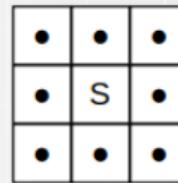
# Poszukiwanie wszerz (2)



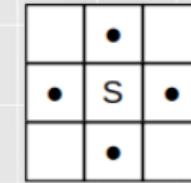
## Poszukiwanie wszerz (3)

- ▶ Szczególnie użyteczne dla grafów typu kratownica (np. pola w grze).
- ▶ Jeśli możemy iść w 8 kierunkach, to mamy do czynienia z sąsiedztwem Moore'a, zaś odległość od  $v_s$  do  $v_e$  odpowiada odległości Czebyszewa (szachowej).
- ▶ Jeśli możemy iść w 4 kardynalnych kierunkach to mamy do czynienia z sąsiedztwem von Neumanna, zaś odległość od  $v_s$  do  $v_e$  odpowiada odległości Manhattan (miejskiej).

Sąsiedztwo Moore'a



Sąsiedztwo von Neumanna





# Poszukiwanie wszerz (4)

Przykład dla sąsiedztwa von Neumanna:

2	1	2		13	14	15	16	17	18	19	20	21					
1	0	1		12	13	14	15	16	17	18	19	20		25			
2	1	2		11	12				18	19	20	21		25	24	25	
3	2	3		10	11				18	19	20			23	24	25	
4	3	4		9	10				16	17	18	19	20	21	22	23	24
5	4	5	6	7	8	9			15	16	17	18	19	20	21	22	23
6	5	6	7	8	9	10	11	12	13	14	15						
7	6	7		10	11	12	13	14	15	16		20	21	22			
8	7	8		11	12	13	14	15	16	17	18	19	20	21			
9	8	9		12	13	14	15	16	17	18							
10	9	10		13	14	15	16	17	18	19	20	21	22	23	24	25	



## Poszukiwanie wszerz (5)

Przykład dla zmodyfikowanego sąsiedztwa Moore'a (nie przechodzimy przez narożniki ścian):

1	1	1		12	12	12	13	14	15	16	17	18			
1	0	1		11	11	12	13	14	15	16	17	17		21	21
1	1	1		10	10				16	16	16	16		21	20
2	2	2		9	9				15	15	15			19	19
3	3	3		8	8				14	14	14	15	16	17	18
4	4	4	5	6	7	8			13	13	14	15	16	17	18
5	5	5	5	6	7	8	9	10	11	12	13				
6	6	6			8	8	9	10	11	12	13		16	16	17
7	7	7			9	9	9	10	11	12	13	14	15	16	17
8	8	8			10	10	10	10	11	12	13			20	21
9	9	9			11	11	11	11	11	12	13	14	15	16	17



# Algorytm Floyd-Warshall (1)

- ▶ Rozwiązuje all-pairs shortest path problem w grafie skierowanym.
  - ▶ Uzyskujemy macierz rozmiaru  $n \times n$ , której element  $(u, v)$  przechowuje długość najkrótszej drogi od  $u$  do  $v$ .
  - ▶ Zwraca tylko długości tras, a nie same trasy. Można jednak zmodyfikować go by znaleźć też trasy.
- ▶ Wagi grafu mogą być ujemne, ale nie może być ujemnego cyklu.
  - ▶ Podczas algorytmu należy sprawdzać okresowo przekątną macierzy, wartości ujemne są oznaką cykli ujemnych (gdyż najkrótsza droga od  $u$  do  $u$  powinna mieć zawsze długość 0).
- ▶ Algorytm działa w czasie  $O(n^3)$ .



# Algorytm Floyd-Warshall (2)

Przebieg algorytmu:

1. Tworzymy macierz  $dist$  rozmiaru  $n \times n$  z wszystkimi elementami ustawionymi na  $\infty$ .

2. Dla każdej krawędzi  $(u, v)$  wykonujemy:

$$dist[u][v] \leftarrow w(u, v).$$

3. Dla  $v$  od 0 do  $n - 1$  wykonujemy:

1. Dla  $i$  od 0 do  $n - 1$  wykonujemy:

1. Dla  $j$  od 0 do  $n - 1$  wykonujemy:

1. Jeżeli  $dist[i][j] > dist[i][v] + dist[v][j]$ :

1.  $dist[i][j] \leftarrow dist[i][v] + dist[v][j].$



## Algorytm Johnsona (1)

- ▶ Również rozwiązuje all-pairs shortest path problem w grafie skierowanym.
- ▶ Algorytm korzysta z algorytmu Bellmana-Forda i algorytmu Dijkstry.
- ▶ Złożoność to  $O(n^2 \log n + nk)$ .
  - ▶  $O(nk)$  z wywołania algorytmu Bellmana-Forda, zaś  $O(n^2 \log n)$  z  $n$ -krotnego wywołania algorytmu Dijkstry.
  - ▶ Dla grafów rzadkich sprowadza się to do  $O(n^2 \log n)$  – szybciej niż algorytm Floyda-Warshalla.



## Algorytm Johnsona (2)

1. Dodajemy dodatkowy węzeł  $q$  i łączymy z wszystkimi  $n$  wierzchołkami krawędziami o wagę 0.
2. Wywołujemy algorytm Bellmana-Forda na wierzchołku  $q$ .
  - ▶ Znajdujemy dla każdego wierzchołka  $v$  minimalną wagę  $h(v)$  na trasie z  $q$  do  $v$ .
  - ▶ Wykrycie ujemnego cyklu kończy cały algorytm.
3. Zmiana wag grafu:  $w(u, v) \leftarrow w(u, v) + h(u) - h(v)$ .
4. Usuwamy  $q$  i wykonujemy algorytm Dijkstry  $n$  razy (dla każdego wierzchołka początkowego osobno).
5. Odległość z  $u$  do  $v$  w oryginalnym grafie to odległość zwrócona przez algorytm Dijkstry plus  $h(v) - h(u)$ .



# Podsumowanie

Algorytm	Problem	Graf	Czas ogólny	Czas gęsty	Czas rzadki
Poszukiwanie wszerz	single-source	bez wag (identyczne wagi)	$O(n + k)$	$O(n^2)$	$O(n)$
Dijkstra	single-source	bez ujemnych wag	$O(k + n \log n)$	$O(n^2)$	$O(n \log n)$
A*	single-pair	bez ujemnych wag	zależny od heurystyki		
Bellman-Ford	single-source	bez ujemnych cykli	$O(kn)$	$O(n^3)$	$O(n^2)$
Floyd-Warshall	all-pairs	bez ujemnych cykli	$O(n^3)$	$O(n^3)$	$O(n^3)$
Johnson	all-pairs	bez ujemnych cykli	$O(n^2 \log n + nk)$	$O(n^3)$	$O(n^2 \log n)$



Wrocław  
University  
of Science  
and Technology

# Struktury danych

Wykład 11  
Problem najdłuższej drogi

dr inż. Jarosław Rudy





# Problem najdłuższej drogi (1)

- ▶ Problem najdłuższej drogi może oznaczać kilka różnych rzeczy.
- ▶ Pierwszą definicją jest po prostu odwrócenie problemu najkrótszej drogi.
- ▶ Analogicznie otrzymujemy odwrócone warianty:
  - ▶ Single-pair longest path problem.
  - ▶ Single-source longest path problem.
  - ▶ Single-destination longest path problem.
  - ▶ All-pairs longest path problem.



## Problem najdłuższej drogi (2)

Tak rozumiany problem najdłuższej drogi w grafie  $G$  rozwiązuje się następująco:

- ▶ Niech —  $G$  będzie grafem  $G$  z zanegowanymi wagami krawędzi (ewentualnie też wierzchołków).
- ▶ Rozwiązuje się problem najkrótszej ścieżki w grafie —  $G$  za pomocą odpowiedniego algorytmu.
- ▶ Otrzymana ścieżka (o ile istnieje) jest żądaną najdłuższą ścieżką w  $G$ .
  - ▶ Jeśli interesuje nas tylko długość ścieżki, to należy ją zamienić na przeciwną.

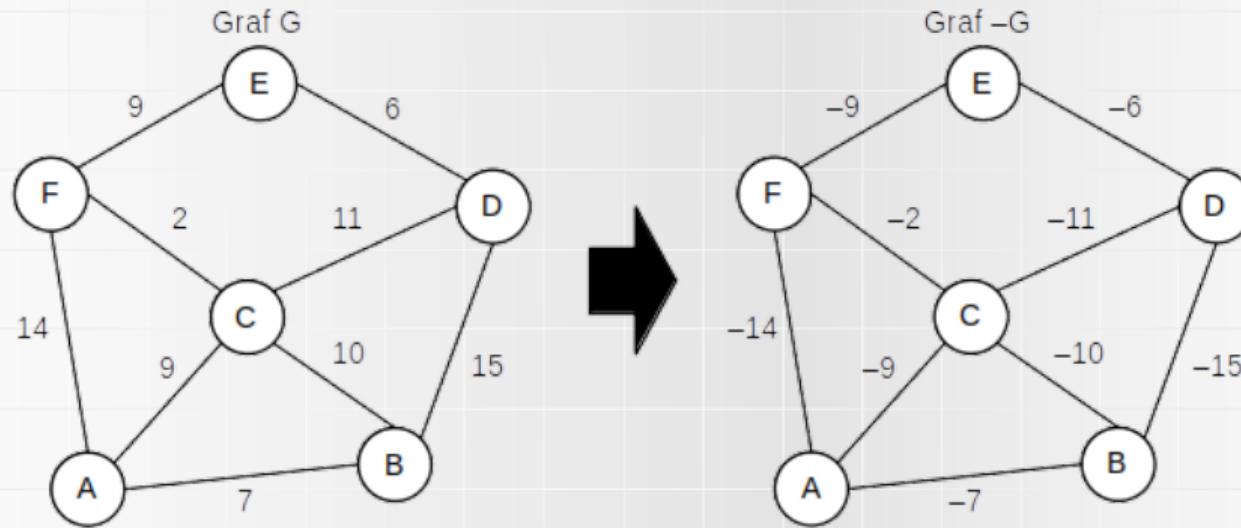


## Problem najdłuższej drogi (3)

- ▶ Ponieważ problemy są analogiczne, to występują te same trudności i ograniczenia jak dla problemu najkrótszej ścieżki:
  - ▶ Dla algorytmu Dijkstry nie może być krawędzi dodatnich w grafie  $G$  (ponieważ będą ujemne krawędzie w grafie  $-G$ ).
  - ▶ Dla algorytmu Bellmana-Forda nie może być cykli o dodatniej długości w  $G$  osiągalnych ze źródła (ponieważ będą cykle o ujemnej długości osiągalne ze źródła w  $-G$ ).
    - ▶ Jeśli są, to najdłuższa ścieżka (w tym sensie) nie istnieje w  $G$ .
- ▶ Podejście wymaga dodatkowego czasu  $O(n + k)$  (na konstrukcję grafu  $-G$ ), ale nie zmienia to złożoności algorytmów.

## Problem najdłuższej drogi (4)

Dla wielu grafów tak rozumiana najdłuższa droga nie istnieje, a pokazana transformacja jest bezużyteczna





## Problem najdłuższej drogi (5)

- ▶ Przy takiej definicji problemu i grafu („zwykły” graf o maksymalnie  $n(n - 1)$  krawędziach) problemy najkrótszej i najdłuższej ścieżki są wielomianowe (należą do  $\mathcal{P}$ ).
- ▶ Jest tak gdyż istnieje algorytm znajdujący ścieżkę (lub stwierdzający jej brak) w czasie wielomianowym.
  - ▶ Jest nim algorytm Bellmana-Forda działający w czasie  $O(nk)$ .
- ▶ O ile jednak tak zdefiniowany problem najkrótszej ścieżki ma sens (szukanie najtańszej trasy), to problem najdłuższej ścieżki w tym rozumieniu jest mniej praktyczny.
  - ▶ Po co szukać najdroższej trasy pomiędzy punktami?



## Problem najdłuższej drogi (6)

- ▶ W praktyce przez problem najdłuższej drogi (longest path problem, LPP) zwykle rozumie się inny, następujący problem:
  - ▶ Dany jest graf  $G$ . Celem jest znalezienie najdłuższej drogi prostej w całym grafie.
    - ▶ Druga musi być prosta tzn. nie mogą powtarzać się wierzchołki (a tym samym krawędzie).
    - ▶ Znaleziona droga będzie zaczynała się w pewnym wierzchołku  $v_s$  i kończyła w pewnym wierzchołku  $v_e$ , ale wierzchołki te nie są podawane na wejściu – algorytm sam musi je wskazać.



## Problem najdłuższej drogi (7)

- ▶ Wg takiej definicji praktycznie każdy graf ma najdłuższą drogę.
- ▶ Złożoność LPP jest jednak zupełnie inna niż w poprzedniej definicji – problem jest  $\mathcal{NP}$ -trudny.
- ▶ Dowód opiera się związku z problemem ścieżki Hamiltonowskiej.
- ▶ Nieznane są algorytmy rozwiązujące LPP w czasie wielomianowym dla dowolnego grafu (i nie będzie takich, chyba że  $\mathcal{P} = \mathcal{NP}$ ).
- ▶ Można jednak znaleźć wydajne algorytmy w niektórych szczególnych przypadkach.



## Problem najdłuższej drogi (8)

- ▶ Jeśli graf ma nie ma dodatkowych cykli, to można rozwiązać LPP w czasie  $O(n^3)$  w następujący sposób:
  - ▶ Standardowo konstruujemy graf  $-G$  (nie będzie w nim ujemnych cykli).
  - ▶ Wykonujemy algorytm Floyda-Warshalla na  $-G$ .
  - ▶ Wybieramy najkrótszą ze znalezionych tras.
- ▶ Jeśli potrzebne są same ścieżki, można zmodyfikować algorytm Floyda-Warshalla lub wykonać algorytm Bellmana-Forda na znalezionym wierzchołku początkowym.



# Problem najdłuższej drogi (9)



-G

Floyd-Warshall

	A	B	C	D
A	0	9	15	19
B	$-\infty$	0	6	-10
C	$-\infty$	-8	0	4
D	$-\infty$	-12	-6	0

G

	A	B	C	D
A	0	-9	-15	-19
B	$\infty$	0	-6	10
C	$\infty$	8	0	-4
D	$\infty$	12	6	0



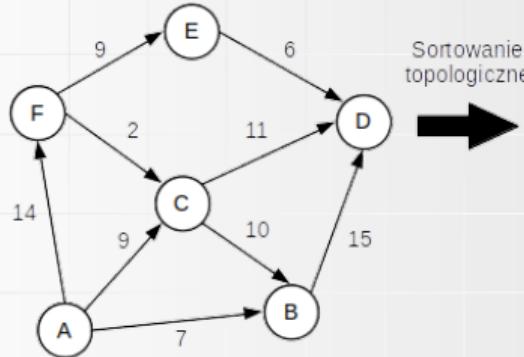
## Problem najdłuższej drogi (10)

Dla DAG można rozwiązać LPP w czasie  $O(n + k)$  tj. liniowym. Algorytm:

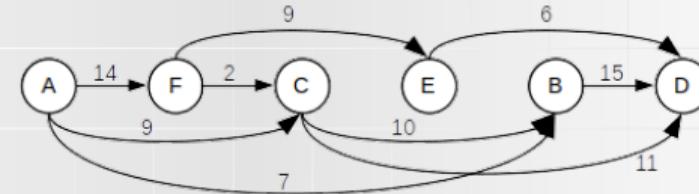
1. Znajdujemy porządek topologiczny (topological sorting).
2. Tworzymy listę  $\text{dist}$  i wpisujemy  $\text{dist}[v] \leftarrow -\infty$  dla każdego węzła  $v$ .
3. Dla kolejnych wierzchołków  $v$  w porządku topologicznym:
  - 3.1  $\text{dist}[v]$  wynosi największy  $\text{dist}$  poprzedników  $v$  plus krawędź łącząca poprzednika z  $v$ .
    - ▶ Jeśli  $v$  nie ma poprzedników to  $\text{dist}[v] \leftarrow 0$
4. Zaczynając od wierzchołka z największym  $\text{dist}[v]$  cofamy się (zawsze po poprzedniku z największym  $\text{dist}[v]$ ), aż do węzła bez poprzedników.
5. Wierzchołki odwiedzone podczas cofania (w odwrotnej kolejności) to szukana ścieżka.



# Problem najdłuższej drogi (11)



Sortowanie topologiczne



Najdłuższa ścieżka: A, F, C, B, D



A	B	C	D	E	F
0	-∞	-∞	-∞	-∞	-∞
0	-∞	-∞	-∞	-∞	14
0	-∞	16	-∞	-∞	14
0	-∞	16	-∞	23	14
0	26	16	-∞	23	14
0	26	16	41	23	14



# Problem najdłuższej drogi (12)

Algorytm Kahna wyznaczania kolejności topologicznej w  $O(n + k)$ :

1. Stwórz listę  $S$  węzłów bez poprzedników i pustą listę  $L$ .
2. Dopóki  $S$  nie jest puste:
  - 2.1 Usuń pewien węzeł  $u$  z  $S$  i dodaj go do  $L$ .
  - 2.2 Dla każdego węzła  $v$  będącego sąsiadem  $u$ :
    - 2.2.1 Usuń krawędź  $(u, v)$  z grafu.
    - 2.2.2 Jeśli  $v$  nie ma innych krawędzi wchodzących, to dodaj  $v$  do  $S$ .
3. Jeśli w grafie zostały krawędzie, to błąd (jest cykl, a graf nie jest DAG).
4. W przeciwnym razie  $L$  jest (któraś) kolejnością topologiczną.



# Problem najdłuższej drogi (13)

1. Problem LPP ma swój odpowiednik: problem znalezienia najkrótszej ścieżki prostej w całym grafie.
  - 1.1 Sytuacja w tym przypadku jest analogiczna, osobną kwestią jest sens praktyczny takiego problemu.
2. Jakie jest jednak praktyczne zastosowanie LPP?
  - 2.1 Zarządzanie projektami, szeregowanie zadań – wszędzie gdzie harmonogramujemy powiązane aktywności wymagające czasu.



# Szeregowanie zadań (1)

Rozpatrzmy projekt z 5 kamieniami milowymi (KM):

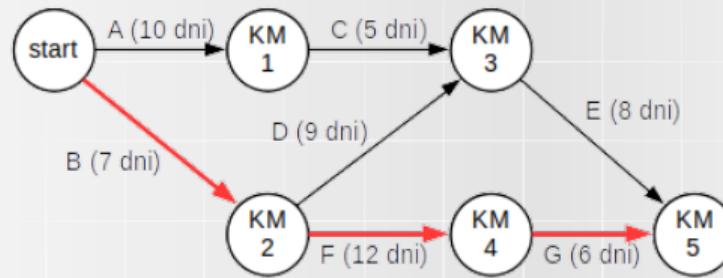
Kamień milowy	1	2	3	4	5
Wymagane zadania do ukończenia	A	B	C i D	F	E i G

Projekt składa się z 7 zadań:

Zadanie	A	B	C	D	E	F	G
Czas trwania (w dniach)	10	7	5	9	8	12	6
Wymagany kamień milowy	-	-	1	2	3	2	4

## Szeregowanie zadań (2)

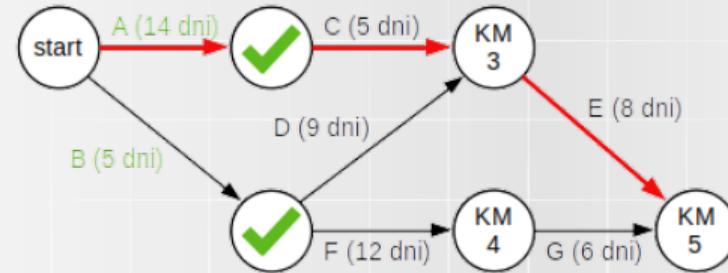
Harmonogram projektu można reprezentować za pomocą DAG. Wierzchołki są kamieniami milowymi, krawędzie są zadaniami, zaś długość krawędzi jest czasem trwania zadania.



Najdłuższa ścieżka w grafie określa czas trwania projektu i zwana jest ścieżką krytyczną. Opóźnienia na ścieżce krytycznej wydłużają czas trwania projektu. Przyspieszenie zadań nie leżących na ścieżce krytycznej nie skróci czasu trwania projektu!

## Szeregowanie zadań (3)

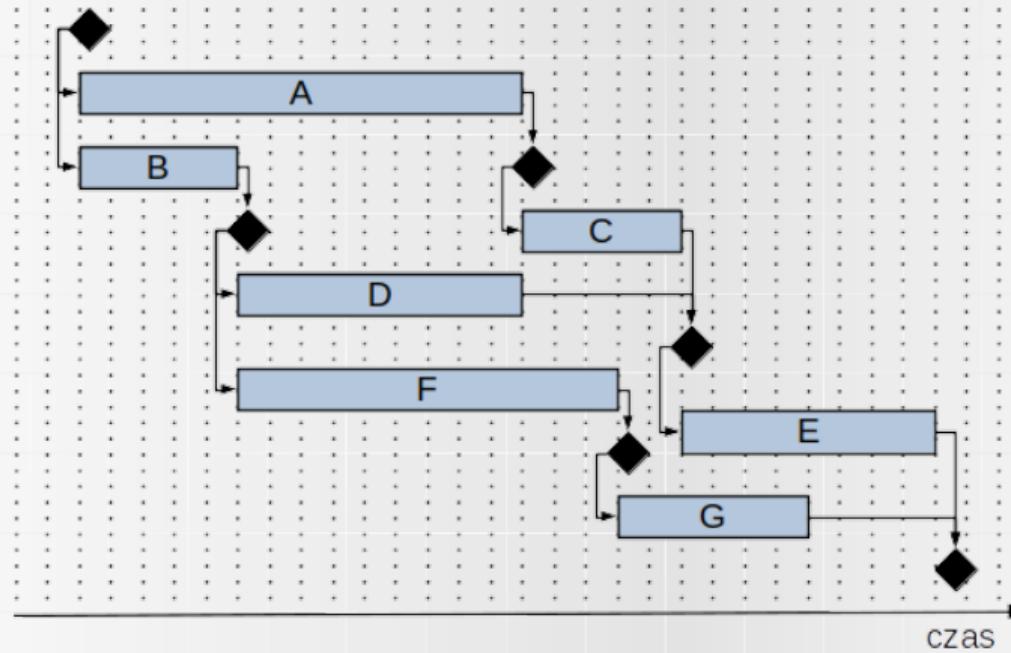
Przykład częściowo zakończonego projektu, w którym zadanie B zajęło 2 dni mniej niż zakładano, ale zadanie A opóźniło się 4 dni względem planu.



Ścieżka krytyczna zmieniła się (projekt trwa teraz 27 dni zamiast 25 pomimo skrócenia poprzedniej ścieżki krytycznej o 2 dni)!

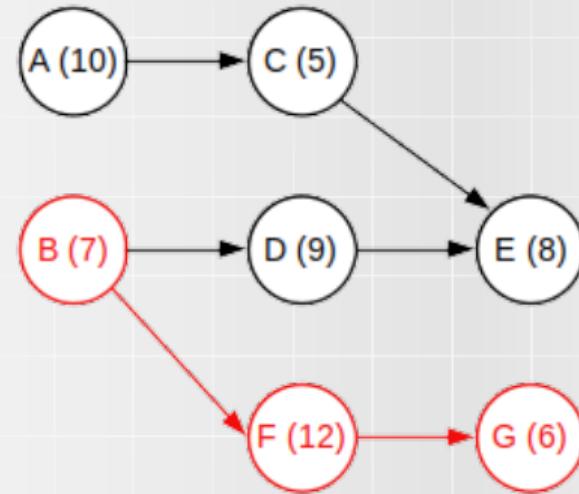
## Szeregowanie zadań (4)

Jednym ze sposobów wizualizowania harmonogramu projektu i analizowania ścieżek krytycznych może być diagram Gantta:



## Szeregowanie zadań (5)

Ten sam projekt można przedstawić w inny sposób: wierzchołki są zadaniami z przypisaną wagą (czasem zadania), zaś krawędzie (bez wag) określają wymaganą kolejność wykonywania zadań.





# Przepływowy problem szeregowania zadań (1)

Rozważmy następujący problem:

- ▶ Dany jest zakład produkcyjny (np. fabryka samochodów).
- ▶ Samochód tworzony jest w  $m = 4$  etapach: (1) zamontowanie podwozia, (2) zamontowanie silnika, (3) zamontowanie nadwozia, (4) lakierowanie.
  - ▶ Etapy muszą być wykonywane w tej kolejności (tzw. marszruta technologiczna).
  - ▶ Każdy etap ma inne stanowisko (tzw. maszynę).
- ▶ Mamy  $n = 6$  samochodów do wyprodukowania. Każdy taki samochód jest zadaniem.
  - ▶ Każde zadanie składa się z 4 operacji (bo mamy 4 etapy).
  - ▶ Każda operacja ma czas trwania.



## Przepływowy problem szeregowania zadań (2)

- ▶ Ponieważ maszyna może przetwarzać tylko jedno zadanie naraz, to musimy określić jaka będzie kolejność wykonywania zadań.
- ▶ Kolejność jest taka sama na każdej maszynie (jeśli A ma montowany silnik przed C, to A jest też lakierowany przed C).
- ▶ Celem jest określenie takiej kolejności (permutacji) wykonywania zadań, by zminimalizować czas całego procesu produkcji.
  - ▶ Szukamy więc takiej permutacji która minimalizuje czas najdłuższego (maksymalnego) zadania.
  - ▶ Taki problem nazywa się permutacyjnym przepływowym problemem szeregowania zadań.



# Przepływowowy problem szeregowania zadań (3)

Rozważmy konkretny przykład dla  $n = 6$  i  $m = 4$ :

Zadanie	A	B	C	D	E	F
Czas operacji 1 (podwozie)	4	3	7	2	4	8
Czas operacji 2 (silnik)	8	6	5	4	3	3
Czas operacji 3 (nadwozie)	5	9	3	7	1	6
Czas operacji 4 (lakier)	4	8	7	6	8	2

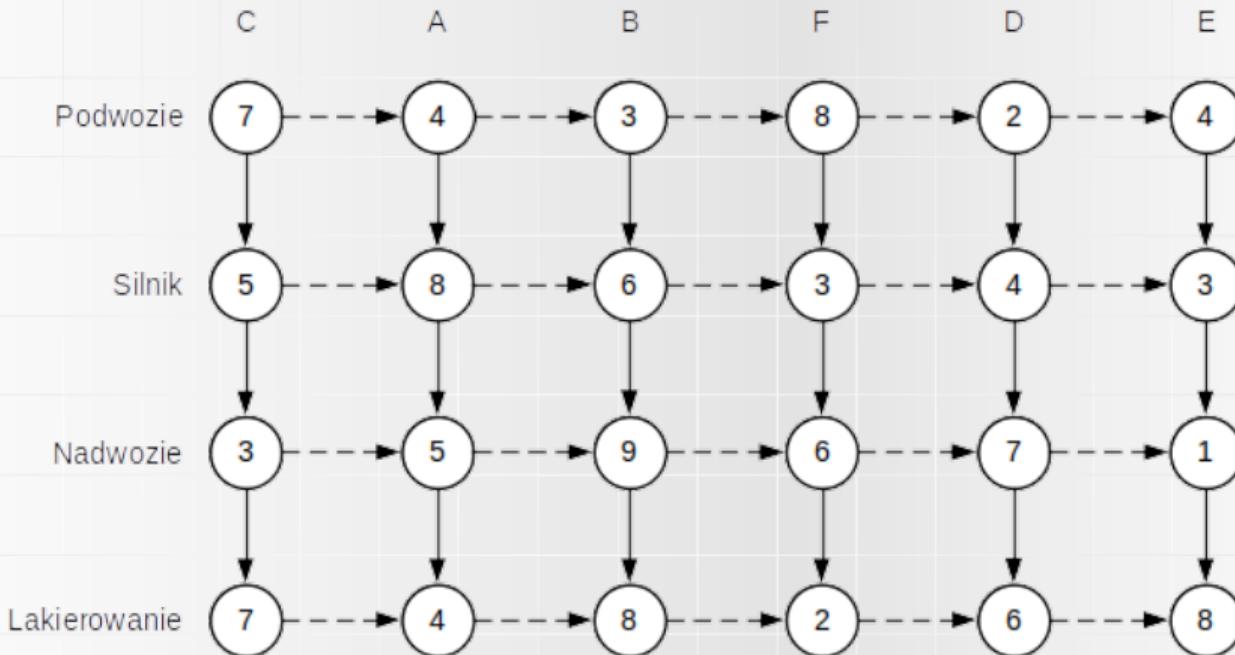


## Przepływowy problem szeregowania zadań (4)

- ▶ Rozważmy przykładowe rozwiązanie:  $(C, A, B, F, D, E)$ .
- ▶ Rozwiązanie to możemy przedstawić w postaci DAG, a konkretnie grafu typu kratownica.
- ▶ „Wiersze” reprezentują maszyny (ciąg operacji na jednej maszynie).
- ▶ „Kolumny” reprezentują zadania (ciąg operacji jednego zadania).
- ▶ Łuki określają które operacje wykonywane są po których.
  - ▶ Przez łuki ciągłe oznaczamy kolejność narzuconą przez problem (kolejność technologiczna).
  - ▶ Przez łuki kreskowane oznaczamy kolejność narzuconą przez rozwiązanie (kolejność zadaniowa).

# Przepływowowy problem szeregowania zadań (5)

Graf dla przytoczonego problemu i rozwiązania ( $C, A, B, F, D, E$ ).



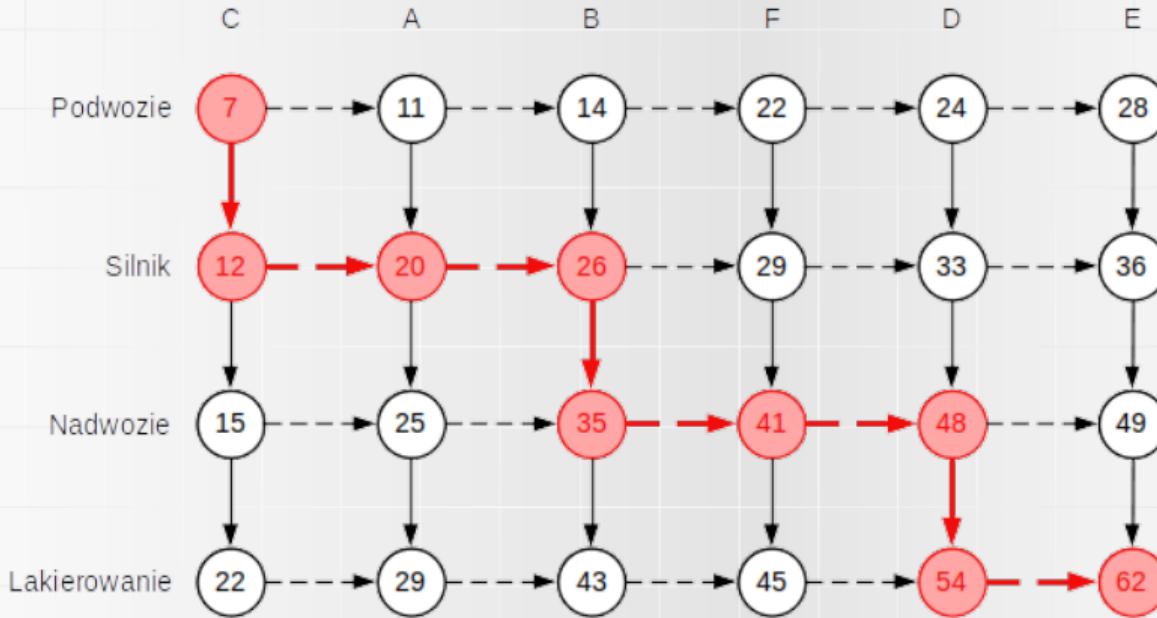


## Przepływowy problem szeregowania zadań (6)

- ▶ Dla operacji zadania  $i$  na maszynie  $j$  obliczymy jej czas zakończenia  $C_{i,j}$ .
- ▶ Operacja musi czekać na poprzednika technologicznego (na zakończenie operacji tego samego zadania na wcześniejszej maszynie).
- ▶ Operacja musi czekać na poprzednika zadaniowego (na zakończenie operacji wcześniejszego zadania na tej samej maszynie).
- ▶ Ostatecznie  $C_{i,j} = \max\{C_{i-1,j}, C_{i,j-1}\} + p_{i,j}$ .
  - ▶ Dla  $i = 1$  i/lub  $j = 1$  wzór się upraszcza (przy braku poprzednika przyjmujemy wartość 0).
  - ▶ Obliczamy macierz  $n \times m$  czasów zakończenia w odpowiedniej kolejności (musimy znać wartość z góry i z lewej).
    - ▶ Szukana wartość  $C_{\max} = C_{n,m}$ .

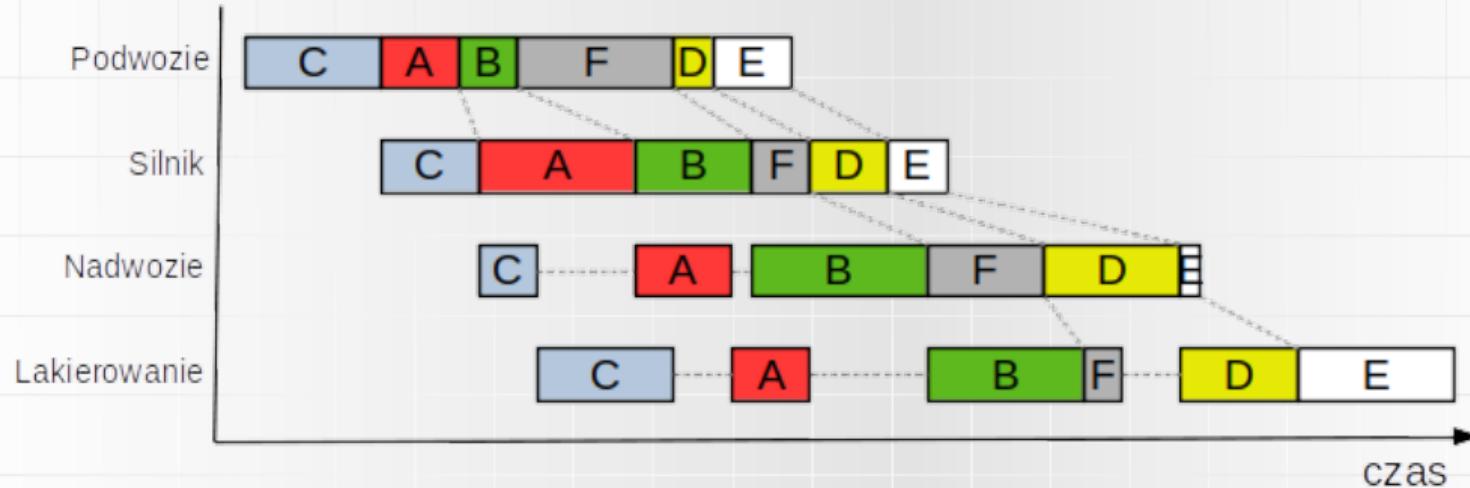
# Przepływowo problem szeregowania zadań (7)

Graf („macierz”) czasów zakończenia poszczególnych operacji z zaznaczoną ścieżką krytyczną:



# Przepływowy problem szeregowania zadań (8)

Diagram Gantta dla podanego przykładu:



Liniami poziomymi oznaczono, gdy operacja musiała czekać na poprzednika technologicznego, zaś ukośnymi gdy musiała czekać na poprzednika zadaniowego.

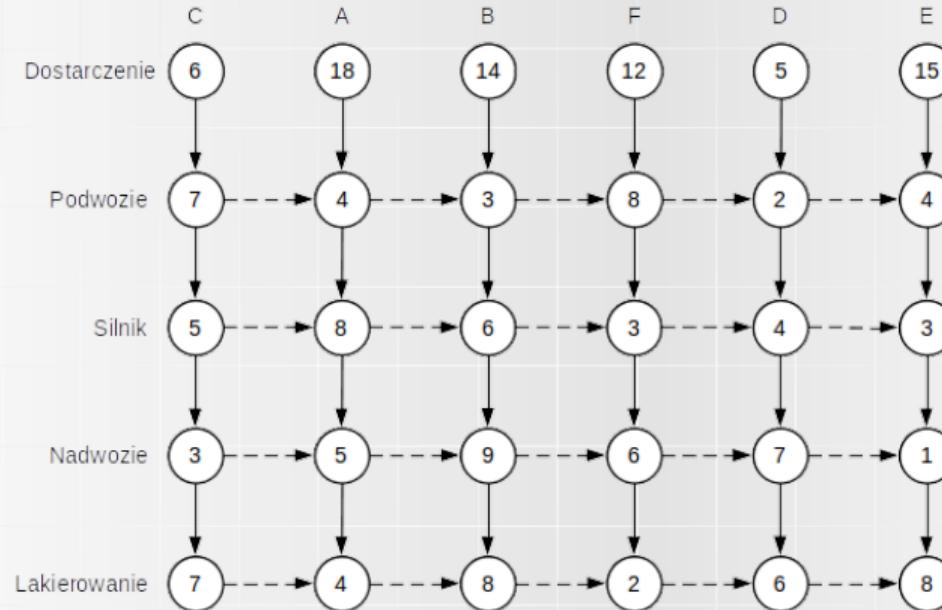


## Przepływowy problem szeregowania zadań (9)

- ▶ Rozpatrzmy modyfikację podstawowego problemu, które wpłyną na sposób konstrukcji grafu.
- ▶ Założymy, że przetwarzanie zadania może rozpocząć się dopiero po pewnym czasie (np. czekamy na dotarcie części).
  - ▶ Takie mechanizm najczęściej nazywa się czasem dostępności (ready time, release time, arrival time).
  - ▶ Możemy to zamodelować dodatkową operacją przed właściwym rozpoczęciem zadania.

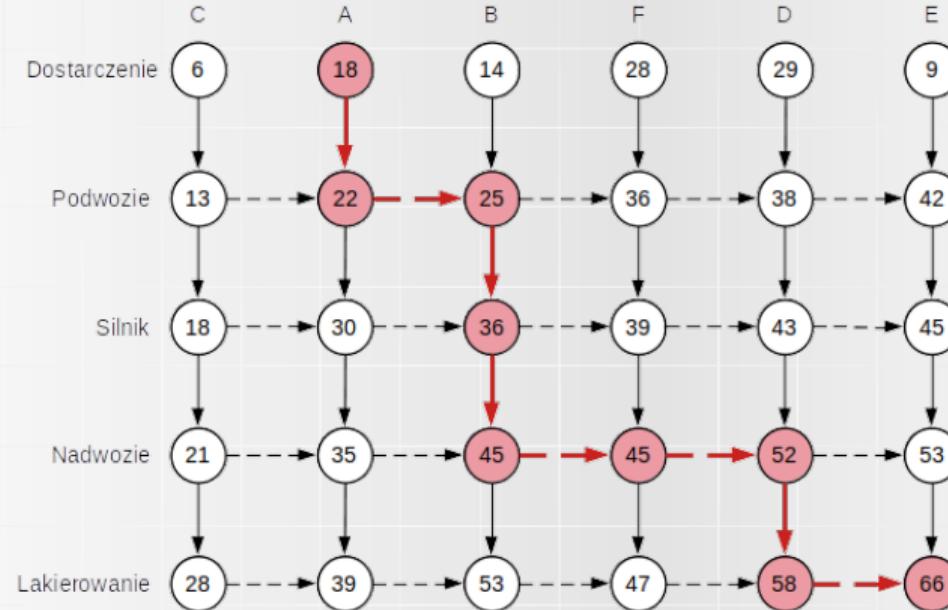
# Przepływowowy problem szeregowania zadań (10)

Graf rozwiązań (z przykładowymi czasami dostępności):



# Przepływowowy problem szeregowania zadań (11)

Graf czasów zakończenia z zaznaczoną ścieżką krytyczną:



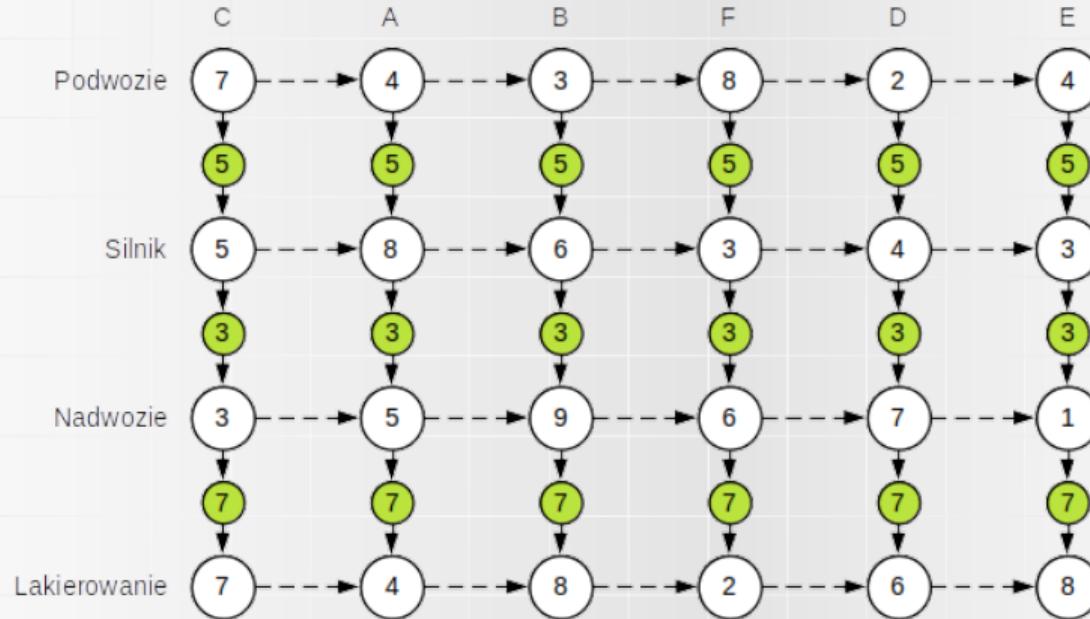


## Przepływowy problem szeregowania zadań (12)

- ▶ Rozpatrzmy inną modyfikację problemu. Założmy, że chcemy uwzględnić czas pomiędzy kolejnymi operacjami zadania (czas transportu).
  - ▶ Czas ten nie jest zaniedbywalny np. gdy kolejne etapy wykonywane są w różnych fabrykach.
- ▶ Czas transportu może być różny dla różnych zadań, ale założmy że jest stały (zależny tylko od tego z której na którą maszynę przenosimy).
- ▶ Założenie możemy zamodelować dodatkowym wierzchołkiem pomiędzy kolejnymi operacjami tego samego zadania tak że waga tego wierzchołka odda czas transportu.
  - ▶ Prościej jednak jest po prostu dodać wagę do istniejącej krawędzi pomiędzy operacjami zadania.

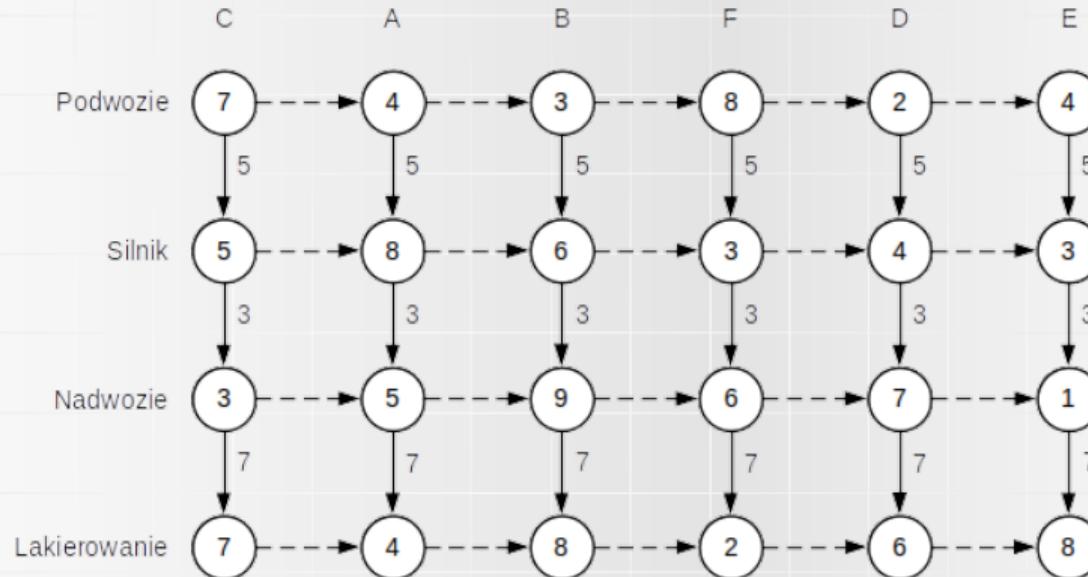
# Przepływowo problem szeregowania zadań (13)

Graf rozwiązań (z przykładowymi czasami transportu) w wersji z dodatkowymi wierzchołkami:



# Przepływowowy problem szeregowania zadań (14)

Graf rozwiązań (z przykładowymi czasami transportu) w wersji bez dodatkowych wierzchołków:



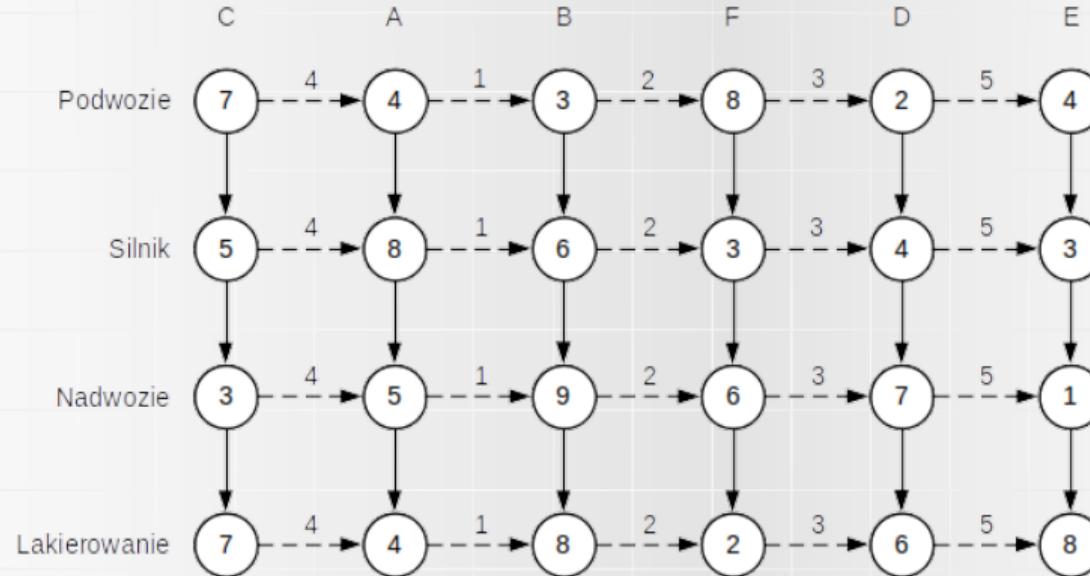


## Przepływowy problem szeregowania zadań (15)

- ▶ Kolejna modyfikacja: założmy, że pomiędzy operacjami maszynę trzeba przygotować przed kolejną operacją.
  - ▶ Może to obejmować np. czas potrzebny na wyładowanie poprzedniego detalu i załadowanie następnego, zmianę ustawień maszyny itp. W praktyce takie czynności nazywa się przezbrojeniem (setup).
- ▶ Znów założymy, że czas przezbrojenia jest stały dla maszyny (niezależny od tego z jakiego zadania na jakie przebrajamy).
- ▶ Czas przezbrojenia możemy zamodelować dodatkowymi wierzchołkami pomiędzy opercjami maszyny lub po prostu wagą krawędzi pomiędzy takimi operacjami.

# Przepływowo problem szeregowania zadań (16)

Graf rozwiązań (z przykładowymi czasami przezbrojenia) w wersji bez dodatkowych wierzchołków:





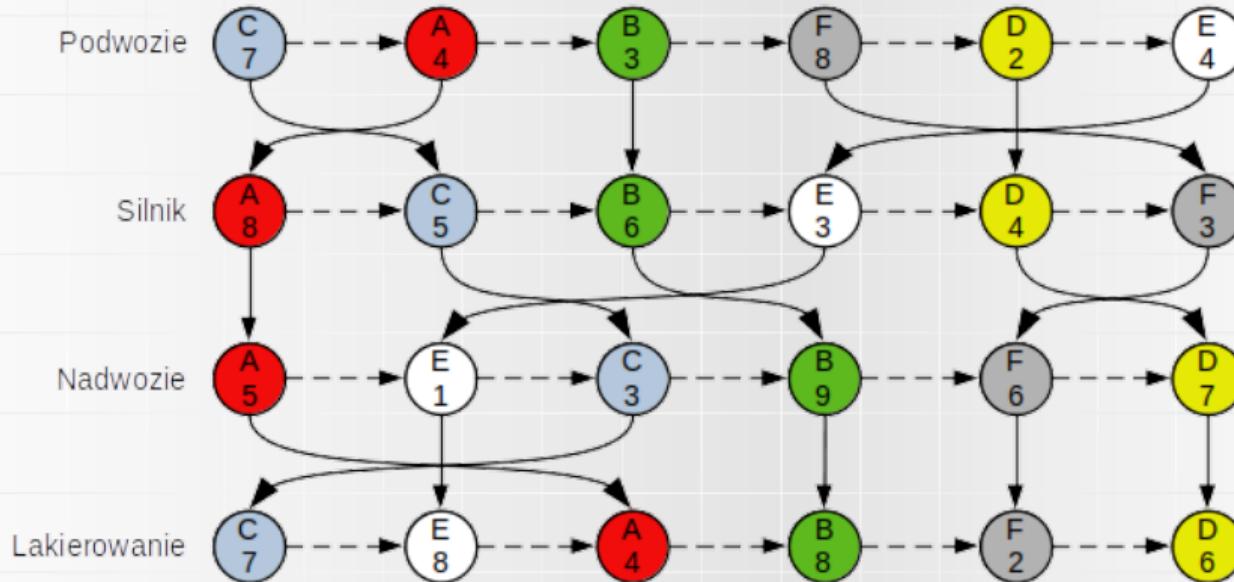
## Przepływowy problem szeregowania zadań (17)

- ▶ Permutacyjny problem przepływowego zakładał, że każda maszyna ma taką samą kolejność (permutację) wykonywania zadań.
- ▶ W ogólniejszym (niepermutacyjnym) problemie przepływowym każda maszyna może mieć inną kolejność.
  - ▶ Mamy więc tyle permutacji ile jest maszyn.
  - ▶ Taki problem wciąż możemy modelować grafem DAG, ale (w ogólności) nie jest on już kratownicą.

# Przepływowowy problem szeregowania zadań (18)

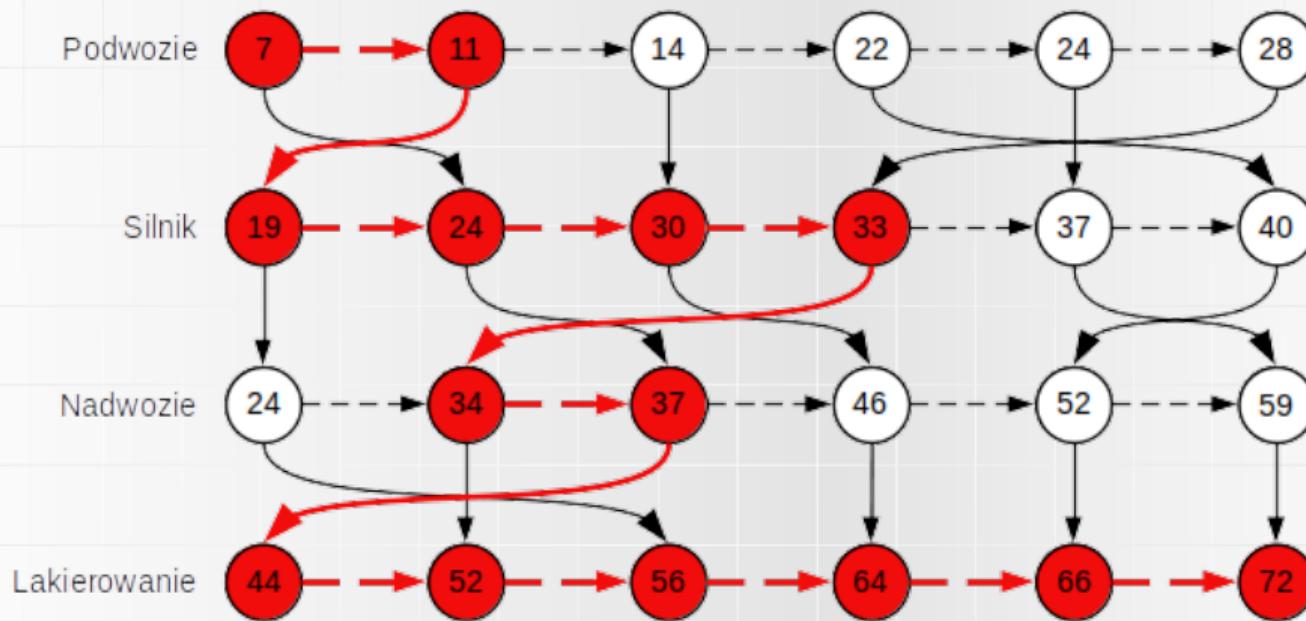
Graf dla przytoczonego problemu i rozwiązania

$((C, A, B, F, D, E), (A, C, B, E, D, F), (A, E, C, B, F, D), (C, E, A, B, F, D))$ .



# Przepływowo problem szeregowania zadań (19)

Graf czasów zakończenia poszczególnych operacji z zaznaczoną ścieżką krytyczną:





Wrocław  
University  
of Science  
and Technology

# Struktury danych

Wykład 12  
XML, DTD

dr inż. Jarosław Rudy





# XML (1)

- ▶ Dotychczas omawiane struktury zasadniczo dotyczyły danych przechowywanych w pamięci operacyjnej.
- ▶ Ponadto różne struktury miały różną reprezentację (format przechowywania danych).
- ▶ W praktyce istnieje potrzeba przechowywania danych poza aplikacją (pliki) oraz przekazywania danych pomiędzy aplikacjami (np. poprzez sieć).
- ▶ Potrzebny jest jeden ustandardzowany format mogący reprezentować praktycznie dowolne dane.
- ▶ Przykładem takiego formatu jest XML.



## XML (2)

- ▶ Extensible Markup Language (XML) – rozszerzalny język znaczników bazowany na języku SGML.
- ▶ Otwarty standard zaproponowany przez World Wide Web Consortium (W3C) opublikowany w 1998 roku.
- ▶ XML pozwala na serializację, przesłanie i odtworzenie danych pomiędzy dwoma systemami w ustandaryzowany sposób (interoperacyjność).
- ▶ Zaprojektowany z myślą o dokumentach, ale zasadniczo może służyć do reprezentacji dowolnych danych.



## Zastosowania XML-a:

- ▶ XHTML – przedstawienie HTML-a 4 w postaci XML-a.
- ▶ Office Open XML – format dla Microsoftowych dokumentów Word, Excel, PowerPoint.
- ▶ OpenDocument – analogiczny otwarty format dokumentów dla LibreOffice i OpenOffice.
- ▶ Pliki konfiguracyjne i dane wejściowe różnych aplikacji.
- ▶ Wiadomości RSS oraz Atom (kanały informacyjne, web feed).



- ▶ Protokoły komunikacyjne:
  - ▶ Simple Object Transfer Protocol (SOAP) – protokół komunikacyjny wykorzystywane w komunikacji usług sieciowych.
  - ▶ XMPP (dawniej Jabber) – protokół dla komunikatorów (GaduGadu, Tlen, Facebook, ICQ).
- ▶ Jeden z możliwych formatów dla Asynchronous JavaScript and XML (AJAX).
- ▶ Format wymiany informacji dla UML-a (poprzez XMI).
- ▶ Kompozycja graficzna (layout) dla aplikacji mobilnych Androida.
- ▶ MathML – matematyczny język znaczników.



Istotne pojęcia i składniki XML-a:

- ▶ Dokument XML składa się znaków. W wersji 1.1 dopuszczalne są wszystkie znaki Unicode (poza Nullem), choć niektóre są niezalecane.
- ▶ XML 1.0 posiada więcej znaków zakazanych.
- ▶ Przetwarzaniem dokumentów XML zajmuje się parser (procesor) XML. Specyfikacja określa sposób działania parsera.
- ▶ Dokument XML może zaczynać się opcjonalną deklaracją, która opisujące informacje o dokumencie np.:

```
<?xml version="1.0" encoding="UTF-8"?>
```



- ▶ Składnia XML-a wprowadza podział na znaczniki (markup) i zawartość (content). Generalnie są dwa rodzaje markupów:
  - ▶ Łańcuch znaków zaczynający się od znaku < i kończący się znakiem >.
  - ▶ Łańcuch znaków zaczynający się znakiem & i kończący się znakiem ; (tzw. encja, entity).
- ▶ Generalnie tekst nie będący markupiem jest zawartością.
- ▶ Wyjątkiem jest konstrukcja CDATA, która rozpoczyna się ciągiem <! [CDATA [ i kończy ciągiem ]]> – te dwa ciągi są traktowane jako markup, zaś tekst pomiędzy nimi jako zawartość.



```
<document>
  <book>
    <title>Pan Tadeusz</title>
    <author>Adam Mickiewicz</author>
    <year>1834</year>
  </book>
  <expression>123 &gt; 45</expression>
  <note>
    <![CDATA[<book> i &gt; to przykłady markupów ]]>
  </note>
</document>
```

Na rysunku kolorem wyróżniono markupy. Sekcja CDATA traktuje większość znaków dosłownie (nie jest to jednak to samo co komentarz!)



- ▶ Na bazie markupów wyróżnia się znaczniki (tag). Możliwe są 3 rodzaje tagów:
  - ▶ Znacznik otwierający np. <book>
  - ▶ Znacznik zamykający np. </book>
  - ▶ Znacznik elementu pustego np. <book/>
- ▶ Element – składnik dokumentu, w jednej z dwóch form:
  - ▶ Znacznik elementu pustego.
  - ▶ Ciąg od znacznika otwierającego do znacznika zamykającego.
    - ▶ Tekst pomiędzy znacznikami nazywany jest zawartością elementu. Zawartość ta może zawierać kolejne elementy (elementy potomne).
    - ▶ XML ma więc strukturę drzewiastą.

# XML

Przykład podziału dokumentu na tagi otwierające (zielone), zamkajające (czerwone) i elementu pustego (żółte).

```
<student>
    <name>Michał</name>
    <age>20</age>
    <grades>
        <algebra>5.0</algebra>
        <databases>4.0</databases>
        <algorithms>4.5</algorithms>
    </grades>
    <stipend/>
</student>
```



Ten sam dokument podzielony na elementy.

```
<student>
    <name>Michał</name>
    <age>20</age>
    <grades>
        <algebra>5.0</algebra>
        <databases>4.0</databases>
        <algorithms>4.5</algorithms>
    </grades>
    <stipend/>
</student>
```



## Atrybuty:

- ▶ „Argumenty” elementu występujące w znaczniku początkowym lub znaczniku elementu pustego.
- ▶ Atrybut jest parą nazwa-wartość zapisywana jako `nazwa="wartość"`.
- ▶ Atrybut o danej nazwie może w elemencie wystąpić tylko raz.
- ▶ Element może mieć dowolną liczbę atrybutów.



## Struktura uczelni z atrybutami elementów:

```
<uczelnia nazwa="Pwr" miasto="Wrocław">
    <wydział nazwa="W4" dyscyplina="ITT">
        <katedra nazwa="K28" kierownik="Wojciech Bożejko">
            <pracownik stanowisko="profesor">Wojciech Bożejko</pracownik>
            <pracownik stanowisko="adiunkt">Jarosław Rudy</pracownik>
            <pracownik stanowisko="asystent">Piotr Nowak</pracownik>
        </katedra>
        <katedra nazwa="K30" kierownik="Ewa Skubalska-Rafajłowicz">
            <pracownik stanowisko="profesor">Ewa Skubalska-Rafajłowicz</pracownik>
            <pracownik stanowisko="adiunkt">Tomasz Kubik</pracownik>
            <pracownik stanowisko="asystent">Łukasz Jeleń</pracownik>
        </katedra>
    </wydział>
    <wydział nazwa="W12" dyscyplina="AEE">
        <katedra nazwa="K29" kierownik="Ignacy Dulęba ">
            <pracownik stanowisko="profesor">Ignacy Dulęba</pracownik>
            <pracownik stanowisko="adiunkt">Krzysztof Arent</pracownik>
        </katedra>
    </wydział>
</uczelnia>
```



# XML

Decyzja czy niektóre informacje umieścić w atrybutach czy w elementach potomnych jest często kwestią preferencji lub wygody:

```
<uczelnia nazwa="Pwr" miasto="Wrocław">  
</uczelnia>  
  
<uczelnia>  
    <nazwa>PWr</nazwa>  
    <miasto>Wrocław</miasto>  
</uczelnia>
```



# XML

Atrybut może mieć tylko pojedynczą wartość. Gdy konieczne jest wiele wartości, należy je zakodować (np. jak lista oddzielona przecinkami, średnikami lub spacjami) i zdekodować osobno przy parsowaniu XML-a.

```
<projekt zespół="Marek,Kasia,Robert">  
  
<div style="font-size: 20px; padding: 2em; color: red">  
  
<div class="main-page fancy blue">
```



# XML

## Kodowanie znaków:

- ▶ Można wykorzystać kodowania zdefiniowane przez Unicode, w szczególności kodowania UTF-8 oraz UTF-16.

```
<?xml version="1.0" encoding="ASCII"?>  
<person>Alan Turing</person>
```

- ▶ Można wykorzystywać wiele innych kodowań wykorzystujących znaki Unicode, takich jak kodowanie ASCII, kodowanie ISO-8859-1 itd.

```
<?xml version="1.0" encoding="ISO-8859-1"?>  
<person>Erwin Schrödinger</person>
```

- ▶ XML może sam wykrywać kodowanie, ale standard gwarantuje to jedynie dla UTF-8 i (formalnie) UTF-16, reszta zależy od parsera.



Czasami bezpośrednie użycie niektórych znaków jest problematyczne:

- ▶ Znaki specjalne dla parsera XML jak < czy &.
- ▶ Znaki Unicode niedostępne w obecnym kodowaniu (np. znak 中 w kodowaniu ASCII).
- ▶ Znaki niemożliwe do wprowadzenia na komputerze użytkownika.
- ▶ Znaki o prawie identycznym wyglądzie (np. "A" w alfabetie łacińskim kontra "A" w cyrylicy).

Problemy te można rozwiązać stosując mechanizmy ucieczki (escaping).



# XML

XML definiuje 5 domyślnych encji (dużo mniej niż HTML!):

- ▶ Encja &lt; dla znaku <.
- ▶ Encja &gt; dla znaku >.
- ▶ Encja &amp; dla znaku &.
- ▶ Encja &apos; dla znaku '.
- ▶ Encja &quot; dla znaku ".



- ▶ Dodatkowe encje można używać po uprzednim ich zdeklarowaniu w specyfikacji Document Type Definition (o czym później). Wielkość znaków ma znaczenie.
- ▶ Znaki Unicode można wprowadzić również za pomocą ich numeru w Unicode:
  - ▶ Dziesiętnie: &#nnnn;
  - ▶ Szesnastkowo: &#xhhhh;
- ▶ Początkowe x musi być małą literą, pozostałe reguły są luźne.
- ▶ Przykładowo znak 中 może być wprowadzony jako &#20013; albo &#x4e2d;



- ▶ Sekcja CDATA (character data) służy do zapisu znaków, które nie mają być interpretowane przez parser („masowy” escaping).
  - ▶ Jest to szczególnie przydatne przy próbie zapisu kodu XML-a w XML-u.
  - ▶ Przykładowo dosłowny ciąg <brand>H&M</brand> można zapisać z użyciem encji lub CDATA:
    - ▶ &lt;brand&ampgtH&M&lt;/brand&ampgt.
    - ▶ <! [CDATA [<brand>H&M</brand>] ]>.
- ▶ Ciąg CDATA nie może się zagnieździć i nie może zawierać ciągu ]]>, ale drugi problem można obejść.



- ▶ Komentarze w XML-u zaczynają się ciągiem `<!--` i kończą ciągiem `-->`.
- ▶ Dla kompatybilności z SGML komentarze nie mogą zawierać ciągu `--` (więc nie mogą się zagnieździć).
- ▶ Wewnątrz komentarzy nie działają encje, więc można używać tylko znaków z aktualnego kodowania.

`<!-- kodowanie ciągu -->` w sekcjach CDATA `-->`  
`<![CDATA[ ]]]><![CDATA[>]]>`



## Przestrzenie nazw (namespaces)

- ▶ Pozwalają unikać niejednoznaczności identycznie nazwanych elementów.
- ▶ Deklaracja za pomocą atrybutu `xmlns:prefix="URI"`, gdzie URI to jakiś Uniform Resource Identifier (np. URL).
  - ▶ Elementy i atrybuty o nazwie zaczynające się od prefix: są w tej przestrzeni nazw, o ile mają powyższą deklarację.
  - ▶ Elementy potomne takiego elementu także są w tej przestrzeni nazw.
- ▶ Przestrzeń domyślną można deklarować z użyciem: `xmlns="URI"`



## Wersje XML-a:

- ▶ Wersja 1.0:
  - ▶ Obecnie piąta edycja.
  - ▶ Wciąż stosowana i rekomendowana.
  - ▶ Mniejszy zakres dopuszczalnych znaków.
  - ▶ Do edycji czwartej ograniczony był zbiór znaków mogących stanowić identyfikatory (nazwy elementów, atrybutów itp.).
  - ▶ W edycji piątej ograniczenia są mniejsze (jak w XML 1.1).



Przykład wykorzystania znaków spoza ASCII w nazwach elementów i atrybutów:

```
<?xml version="1.0" encoding="UTF-8"?>
<俄语_ЛбqнL="ռուսերեն">данные</俄语>
```

- ▶ Wersja 1.1:
  - ▶ Obecnie druga edycja.
  - ▶ Kilka spornych usprawnień.
  - ▶ Implementacja nie jest powszechnie stosowana.
  - ▶ Niezalecany, chyba że potrzebne są jego unikalne cechy.



► Wersja 2.0:

- Pojawiły się plany i dyskusje dotyczące m.in. dodania pewnych narzędzi (np. XML Base, XML Information Set) do standardu czy eliminacji DSD.
- Żadna organizacja nie prowadzi obecnie prac nad XML 2.0.

► MicroXML:

- Okrojony XML, może być przydatny w zastosowaniach, gdzie zwykły XML jest zbyt złożony.
- MicroXML ma uzupełniać XML, JSON i HTML, a nie je zastąpić.



# XML

Dokumenty XML muszą być poprawnie sformułowane (well-formed), czyli muszą spełniać wymogi składniowe XML-a. Inne dokumenty po prostu nie są XML-em. Niektóre reguły poprawnego formułowania:

- ▶ Brak znaków Unicode spoza użytego kodowania.
- ▶ Znaki specjalne (<, & itp.) stosowane tylko do określenia markupów.
- ▶ Tagi są poprawnie zagnieżdżone, sparowane (wielkość znaków ma znaczenie) itd.
- ▶ Tagi nie mogą zawierać niektórych znaków (w tym spacji) i nie mogą zaczynać się od cyfr, dywizu (-) czy kropki.
- ▶ Dokument ma pojedynczy element jako korzeń.



# XML

- ▶ Jeśli parser XML-a wykryje niezgodność, jest zobowiązany zwrócić błąd i zakończyć parsowanie.
- ▶ Jest to podejście ekstremalne i zupełnie inne niż HTML, dla którego przeglądarki starają się wygenerować rozsądную stronę nawet przy niezgodności pewnych elementów.
- ▶ HTML zaakceptuje <br> zamiast <br/>, a XHTML – nie.
- ▶ Podejście XML-a łamię więc tzw. zasadę Postela (robustness principle):

Bądź konserwatywny w tym, co wysyłasz, bądź liberalny w tym, co akceptujesz (nadawca powinien ściśle trzymać się specyfikacji, zaś odbiorca powinien akceptować odstępstwa, o ile przekaz pozostaje zrozumiały).



- ▶ Oprócz poprawnego sformułowania (well-formed), dokumenty XML mogą być zwalidowane (valid).
- ▶ W zwalidowanym XML-u elementy i atrybuty spełniają dodatkowe wymagania nałożone przez zdefiniowaną (w XML-u lub osobno) specyfikcję.
- ▶ Parsery różnią się tym czym przeprowadzają etap walidacji.
  - ▶ Parser musi zgłosić błąd walidacji, ale może kontynuować parsowanie.
  - ▶ Istnieje kilka sposobów określania schematów/gramatyk walidacyjnych dla XML-a m.in. DTD oraz XML Schema.



## Document Type Definition (DTD):

- ▶ Definiuje strukturę dokumentów SGML-a i jego pochodnych (XML-a, HTML-a i XHTML-a).
- ▶ Umożliwia m.in. określenie (z użyciem prostych regexów) jakie elementy i atrybuty ma lub może zawierać dokument XML.
- ▶ Można też definiować encje.
- ▶ Możliwy do zawarcia bezpośrednio w XML lub zewnętrznie w osobnym pliku.
- ▶ Druga opcja pozwala zastosować ten sam DTD dla różnych dokumentów XML.



## Zalety DTD (głównie względem XML Schema):

- ▶ Mocno rozpowszechniony ze względu na dołączenie do standardu XML 1.0
- ▶ Możliwość zawarcia DTD bezpośrednio w dokumencie XML.
- ▶ Większa zwięzłość.
- ▶ Możliwość deklarowania encji.
- ▶ Opis dokumentu w jednym miejscu.



## Wady DTD (głównie względem XML Schema):

- ▶ Brak wsparcia niektórych mechanizmów (przestrzenie nazw).
- ▶ Mniejsze możliwości (mniejsza ekspresywność, brak możliwości wyrażania konkretnej liczności, brak typów danych).
- ▶ Mniejsza czytelność, zwłaszcza przy zastosowaniu sparametryzowanych encji.
- ▶ DTD pisany jest w innym języku niż XML.



Deklaracja <ELEMENT! xyz abc> – opisuje regułę dla elementu xyz jako abc.  
Jako abc można użyć:

- ▶ EMPTY – element nie może mieć zawartości (ani tekst, ani elementy, z wyjątkiem spacji).
- ▶ ANY – dowolna zawartość.
- ▶ (#PCDATA) – zawartość tekstowa (parsed character data) tzn. element nie może mieć elementów potomnych.



- ▶ (#PCDATA | e11 | e12 | ...)\* – co najmniej dwoje dzieci, którymi może być tekst lub elementy podane na liście. W dowolnej kolejności i ilości.
- ▶ e1 – zawartością jest element o nazwie e1.
- ▶ (e11, e12, ...) – lista uporządkowana. Zawartością elementu muszą być elementy (nie tekst!) w podanej kolejności.
- ▶ e11 | e12 | ... – ciąg alternatyw. Zawartością elementu musi być dokładnie jeden z podanych elementów (nie tekst!).



Składnikom na listach można przypisać uproszczoną krotność za pomocą znaku bezpośrednio po składniku:

- ▶ + – składnik musi pojawić się 1 lub więcej razy (za każdym razem jego zawartość może być inna!).
- ▶ \* – składnik musi pojawić się 0 lub więcej razy (jest więc opcjonalny, za każdym razem zawartość może być inna).
- ▶ ? – składnik musi pojawić się 0 lub 1 raz (jest opcjonalny).
- ▶ Brak krotności – składnik musi pojawić się dokładnie raz (jest obowiązkowy).  
Jest to krotność domyślna.



Przykład DTD dla XML-a opisującego pracowników:

```
<!ELEMENT pracownicy (pracownik+)>
<!ELEMENT pracownik (daneOsobowe, dział?)>

<!ELEMENT daneOsobowe (imie, drugieimie?, nazwisko)>
<!ELEMENT imie (#PCDATA)>
<!ELEMENT drugieimie (#PCDATA)>
<!ELEMENT nazwisko (#PCDATA)>

<!ELEMENT dział (#PCDATA)>
```



Przykład XML-a dla wcześniejszego DTD, wraz z deklaracją zewnętrznego DTD:

```
<?xml version="1.0" encoding="UTF-8" standalone="no"?>
<!DOCTYPE pracownicy SYSTEM "pracownicy.dtd">
<pracownicy>
    <pracownik>
        <daneOsobowe>
            <imie>Izabela</imie>
            <nazwisko>Nowakowska-Jasnorzębska</nazwisko>
        </daneOsobowe>
    </pracownik>
    <pracownik>
        <daneOsobowe>
            <imie>Zygflyd</imie>
            <drugieimie>Zenobiusz</drugieimie>
            <nazwisko>Wawrzyniak</nazwisko>
        </daneOsobowe>
        <dział>Public Relations</dział>
    </pracownik>
</pracownicy>
```



Przykład XML-a z wewnętrznym DTD:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE osoba [
    <!ELEMENT osoba (imie, drugieimie, nazwisko)>
    <!ELEMENT imie (#PCDATA)>
    <!ELEMENT drugieimie (#PCDATA)>
    <!ELEMENT nazwisko (#PCDATA)>
]>
<osoba>
    <imie>Amadeusz</imie>
    <drugieimie>Zenon</drugieimie>
    <nazwisko>Kowalski</nazwisko>
</osoba>
```



Deklaracja <!ATTLIST! pozwala określić dopuszczalne atrybuty danego elementu.  
Dla każdego atrybutu określamy:

- ▶ nazwę atrybutu.
- ▶ typ wartości:
- ▶ obowiązkowość/wartość domyślną.

```
<!ATTLIST img
    src    CDATA          #REQUIRED
    id     ID             #IMPLIED
    sort   CDATA          #FIXED "true"
    print  (yes | no) "yes"
>
```



Niektóre sposoby definiowania wartości atrybutu:

- ▶ CDATA – wartość będąca „dowolnym” (patrz #FIXED) tekst.
- ▶ ID – tekst będący identyfikatorem. Zwalidowany dokument nie może mieć dwóch takich samych identyfikatorów.
- ▶ IDREF oraz IDREFS – identyfikator lub ich lista pasująca do jakiegoś id w dokumencie.
- ▶ ENTITY oraz ENTITIES – nazwa encji (lub ich lista) zadeklarowanej w DTD.
- ▶ (val1 | val2 | ...) – wylistowany zbiór dopuszczalnych wartości.



Obowiązkowość i wartości domyślne:

- ▶ #REQUIRED – atrybut musi wystąpić.
- ▶ #IMPLIED – atrybut nie musi wystąpić.
- ▶ #FIXED "val" – atrybut ma ustaloną wartość "val".
- ▶ "val" – w przypadku braku atrybutu domyślną wartością jest "val".



Obowiązkowość i wartości domyślne:

- ▶ #REQUIRED – atrybut musi wystąpić.
- ▶ #IMPLIED – atrybut nie musi wystąpić.
- ▶ #FIXED "val" – atrybut ma ustaloną wartość "val".
- ▶ "val" – w przypadku braku atrybutu domyślną wartością jest "val".



Przykład deklaracji i użycia encji:

```
<!DOCTYPE sgml [
    <!ELEMENT sgml ANY>
    <!ENTITY % std      "standard SGML">
    <!ENTITY % signature "&#x2014; &author;.">
    <!ENTITY % question "Why couldn't I publish my books directly in %std;?">
    <!ENTITY % author   "William Shakespeare">
]>
```

```
<sgml>&question;&signature;</sgml>
```



Wrocław  
University  
of Science  
and Technology

# Struktury danych

Wykład 13  
XML (schema, parsowanie)

dr inż. Jarosław Rudy





# XML Schema

- ▶ Tak jak DTD jest to konkretny język schematu XML (ang. XML Schema) w przeciwieństwie do całej rodziny schematów (ang. XML schemas).
- ▶ Ze względu na powyższe nazywany też jest W3C XML Schema (WXS) lub XML Schema Definition (XSD).
- ▶ Standard XSD 1.0 rekomendowany przez W3C w 2001 roku.
  - ▶ Druga edycja opublikowana w 2004 roku.
- ▶ Standard XSD 1.1 rekomendowany przez W3C w 2012 roku.
- ▶ Pisany w XML-u (tj. dokumenty XSD przypominają dokumenty XML-a).



# XML Schema

- ▶ Dokumenty XSD definiują schemat tj. metadane zorganizowane za pomocą tzw. komponentów schematu.
- ▶ Dokumenty XSD i komponenty organizowane są za pomocą przestrzeni nazw.
  - ▶ Dokument XSD może dodawać (include) inne dokumenty XSD z tej samej przestrzeni nazw.
  - ▶ Dokument XSD może importować (import) inne dokumenty XSD z innych przestrzeni nazw.
- ▶ Podczas walidacji dokument XML kojarzony jest z pożdanym schematem za pomocą podania parametru dla parsera/walidatora lub specjalnych atrybutów (np. `xsi:schemaLocation`) wewnątrz XML-a.



# XML Schema

Komponenty schematu:

- ▶ Deklaracja elementu
  - ▶ Określenie nazwy i przestrzeni nazw.
  - ▶ Określenie typu elementu – narzucenie ograniczeń na możliwe atrybuty i zawartość elementu.
  - ▶ Możliwość uzależnienia typu od wartości atrybutów.
  - ▶ Możliwość występowania elementu w miejsce innego (substitution groups).
  - ▶ Możliwość wymagania unikalności elementu w danym poddrzewie.



# XML Schema

- ▶ Możliwość wymagania by wartość pasowała do istniejącego identyfikatora jakiegoś elementu.
- ▶ Deklaracje globalne lub lokalne (np. różne deklaracje dla elementów o taka samej nazwie).
- ▶ Deklaracja atrybutu
  - ▶ Określenie nazwy i przestrzeni nazw.
  - ▶ Określenie typu atrybutu – ograniczenie na dopuszczalne wartości.
  - ▶ Możliwość określenia wartości domyślnej.
  - ▶ Możliwość określenia stałej wartości.



# XML Schema

- ▶ Model (element) group.
- ▶ Attribute group.
- ▶ Attribute use – określenie czy atrybut dla danego typu złożonego (np. elementu) jest obowiązkowy.
- ▶ Element particle – określenie dla typu złożonego minimalnej i maksymalnej liczby wystąpień.
- ▶ Inne.



# XML Schema

## Proste typy danych

- ▶ Wykorzystywane do ograniczenia postaci wartości tekstowej elementu lub atrybutu (duża różnica względem DTD).
- ▶ 19 prymitywnych (podstawowych) typów danych m.in. decimal, double, duration, dateTime, string, boolean, anyURI, base64Binary.
- ▶ Pochodne typy danych tworzone poprzez:
  - ▶ Ograniczenie zakresu wartości typu prymitywnego.
  - ▶ Dopuszczenie listy wartości (typ sekwencyjny).
  - ▶ Unia (kilka dopuszczalnych typów wartości).



# XML Schema

- ▶ Specyfikacja XSD definiuje 25 typów pochodnych.
- ▶ Dalsze typy można definiować poprzez schematy.
- ▶ Przykładowe ograniczenia dla typu pochodnego:
  - ▶ Określenie wartości minimalnej i maksymalnej.
  - ▶ Wyrażenie regularne.
  - ▶ Długość łańcucha tekstowego.
  - ▶ Liczba cyfr.
  - ▶ (Dla XSD 1.1) asercje z użyciem wyrażeń XPath 2.0.



## Typy złożone

- ▶ Określają dopuszczalne atrybuty i zawartość elementu. Zawartością może być:
  - ▶ Tylko element (bez zawartości tekstowej).
  - ▶ Tylko tekst.
  - ▶ Brak zawartości.
  - ▶ Zawartość mieszana (element lub tekst).
- ▶ Podobnie jak wcześniej, typ złożony może być zmodyfikowany przez ograniczenie dopuszczalnych atrybutów/elementów lub asercje. Można też rozszerzyć dopuszczalny zakres atrybutów/elementów.



# XML Schema

## Przykład XML Schema (z W3Schools):

```
<?xml version="1.0" encoding="UTF-8" ?>
<xss:schema xmlns:xss="http://www.w3.org/2001/XMLSchema">
  <xss:element name="shiporder">
    <xss:complexType>
      <xss:sequence>
        <xss:element name="orderperson" type="xs:string"/>
        <xss:element name="shipto">
          <xss:complexType>
            <xss:sequence>
              <xss:element name="name" type="xs:string"/>
              <xss:element name="address" type="xs:string"/>
              <xss:element name="city" type="xs:string"/>
              <xss:element name="country" type="xs:string"/>
            </xss:sequence>
          </xss:complexType>
        </xss:element>
        <xss:element name="item" maxOccurs="unbounded">
          <xss:complexType>
            <xss:sequence>
              <xss:element name="title" type="xs:string"/>
              <xss:element name="note" type="xs:string" minOccurs="0"/>
              <xss:element name="quantity" type="xs:positiveInteger"/>
              <xss:element name="price" type="xs:decimal"/>
            </xss:sequence>
          </xss:complexType>
        </xss:element>
      </xss:sequence>
      <xss:attribute name="orderid" type="xs:string" use="required"/>
    </xss:complexType>
  </xss:element>
```



# XML Schema

Przykładowy XML dla powyższego XML Schema (z W3Schools):

```
<?xml version="1.0" encoding="UTF-8"?>
<shiporder orderid="889923"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:noNamespaceSchemaLocation="shiporder.xsd">
    <orderperson>John Smith</orderperson>
    <shipto>
        <name>Ola Nordmann</name>
        <address>Langgt 23</address>
        <city>4000 Stavanger</city>
        <country>Norway</country>
    </shipto>
    <item>
        <title>Empire Burlesque</title>
        <note>Special Edition</note>
        <quantity>1</quantity>
        <price>10.90</price>
    </item>
    <item>
        <title>Hide your heart</title>
        <quantity>1</quantity>
        <price>9.90</price>
    </item>
</shiporder>
```



# XML Schema

Wady XML Schema (m.in.):

- ▶ Złożoność specyfikacji.
- ▶ Niekonsekwencja (np. ograniczenie wartości dla atrybutów działa inaczej niż dla elementów).
- ▶ Słabe wsparcie dla list nieuporządkowanych.
- ▶ Nie można określić co jest korzeniem dokumentu.
- ▶ Nie można definiować zawartości w zależności od kontekstu.



# Parsowanie XML-a

- ▶ Cel projektowy XML-a zakładał łatwość jego przetwarzania.
- ▶ Pomimo tego sama specyfikacja zawiera niewiele informacji o sposobach przetwarzania lub API.
- ▶ W praktyce wykształciło się kilka metod:
  - ▶ SAX.
  - ▶ DOM.
  - ▶ StAX.
  - ▶ XSLT.
  - ▶ XML data binding.



## Simple API for XML (SAX):

- ▶ Technika oparta o zdarzenia (event-based). Zdarzenia (np. otwarcie znacznika) powodują wywołanie odpowiedniej funkcji zarejestrowana przez użytkownika (callback).
- ▶ Parser przechodzi przez dokument jednokrotnie.
- ▶ Niskie zużycie pamięci (proporcjonalne do wysokości drzewa).
- ▶ Zwykle szybsza niż podejście typu DOM.
- ▶ Trudny dostęp do dowolnego elementu, utrudniona walidacja (np. jedna część dokumentu odwołująca się do wcześniejszej).



# Parsowanie XML-a

Typowe zdarzenia SAX:

- ▶ Element start.
- ▶ Element end.
- ▶ Text node.
- ▶ Processing instruction.
- ▶ Comments.



## Document Object Model (DOM)

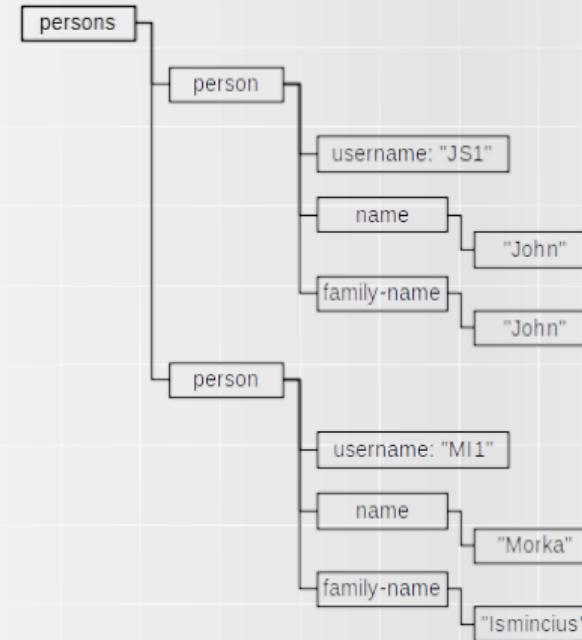
- ▶ Parser buduje drzewo na podstawie dokumentu.
- ▶ Elementy XML stają się węzłami DOM (tekst plus węzły potomne).
- ▶ Dzieci elementu XML stają się dziećmi węzła DOM.
- ▶ Zawartość tekstowa elementu XML staje się węzłem-liściem tekstowym, którego rodzicem jest węzeł DOM.
- ▶ Atrybuty elementu stają się węzłami-liściem potomnymi węzła DOM i są parą klucz-wartość.



Przykładowy XML:

```
<?xml version="1.0" ?>
<persons>
    <person username="JS1">
        <name>John</name>
        <family-name>Smith</family-name>
    </person>
    <person username="MI1">
        <name>Morka</name>
        <family-name>Ismincius</family-name>
    </person>
</persons>
```

DOM odpowiadający przykładowi:





- ▶ Możliwość łatwego znalezienia dowolnego elementu (przejrzenie drzewa lub przejście zadaną ścieżką).
- ▶ Łatwa modyfikacja dokumentu (dodawanie, usuwanie, podmiana elementów i/lub atrybutów).
- ▶ Większe zużycie pamięci (konieczność przechowywania całego drzewa).
- ▶ Wiele implementacji np. libxml2 w C (plus wrappery dla wielu języków), JAXP dla Javy.



- ▶ Streaming API for XML (StAX).
  - ▶ Parser przechowuje pozycję kurSORA, którą można przesuwać do przodu analizując kolejne fragmenty dokumentu (strumieniowo).
  - ▶ Podejście typu „pull” pomiędzy podejściem typu „push” (SAX) i podejściem drzewa (DOM).
- ▶ XML data binding.
  - ▶ Definiuje się mapowanie pomiędzy fragmentami dokumentu (XML) a obiektami i ich polami (programowanie obiektowe).
  - ▶ Dostęp do informacji z XML-a poprzez obiekty.



Extensible Stylesheet Language Transformations (XSLT):

- ▶ Język do transformacji dokumentów XML w inne dokumenty (w tym inne dokumenty XML).
  - ▶ Ogólnie wejściem może być dowolny dokument, z którego można zbudować model XPath (lub XQuery).
- ▶ Turing-zupełny język deklaratywny (w przeciwieństwie do języków imperatywnych typu C++ czy python) oparty o dopasowywanie wzorców.
- ▶ Parser na wejściu otrzymuje oryginalny dokument XML oraz plik XSLT. Na wyjściu otrzymujemy nowy dokument (oryginał nie ulega zmianie).
- ▶ Wykorzystuje XPath do wybierania i filtrowania elementów w drzewie XML.



## Przykładowy XLST:

```
<?xml version="1.0" encoding="UTF-8"?>
<xsl:stylesheet xmlns:xsl="http://www.w3.org/1999/XSL/Transform" version="1.0">
    <xsl:output method="xml" indent="yes"/>

    <xsl:template match="/persons">
        <root>
            <xsl:apply-templates select="person" />
        </root>
    </xsl:template>

    <xsl:template match="person">
        <name username="{@username}">
            <xsl:value-of select="name" />
        </name>
    </xsl:template>

</xsl:stylesheet>
```



Przykładowy wejściowy XML:

```
<?xml version="1.0" ?>
<persons>
    <person username="JS1">
        <name>John</name>
        <family-name>Smith</family-name>
    </person>
    <person username="MI1">
        <name>Morka</name>
        <family-name>Ismincius</family-name>
    </person>
</persons>
```



Wyjściowy dokument dla powyższego przykładu:

```
<?xml version="1.0" encoding="UTF-8"?>
<root>
    <name username="JS1">John</name>
    <name username="MI1">Morka</name>
</root>
```



- ▶ Krytyka XML-a obejmuje jego złożoność, rozwlekłość i nadmiarowość.
- ▶ Mapowanie systemów takich jak języki programowania czy bazy danych na XML bywa trudne, ale nie do tego XML został stworzony.
- ▶ JSON opisywany jest jako bardziej zwięzła alternatywa dla XML-a.
- ▶ JSON ma jednak nieco inne zastosowanie (reprezentacja ustrukturyzowanych danych, a nie dokumentów) i ma znacznie mniej możliwości (mniej typów danych, brak komentarzy itd.).