Politechnika Warszawska

Algorytmy i struktury danych

Odc. 1. Wprowadzenie

Dr hab. inż. **Andrzej Zalewski**

Wydział Elektroniki i Technik Informacyjnych

POLITECHNIKA WARSZAWSKA

O prowadzącym

Afiliacja: na WEiTI od 1991 r., Instytut Automatyki i Informatyki Stosowanej od 1996 [od mgr inż. (1996) do dr hab. (2015)]

Na czym się znam:

- **Inżyniera oprogramowania:**
  - **metody(ki) wytwarzania systemów informatycznych**
  - **różne składowe procesu wytwarzania oprogramowania - wymagania, architektura (konstrukcja), zaprogramowanie (implementacja) testowanie, wdrażanie**
- **zarządzanie projektami**
- **zarządzanie informatyką**

O prowadzącym c.d.

***Funkcje* w PW**:

- **lider zespołu inżynierii oprogramowania w IAiIS PW,**
- **kierownik studiów podyplomowych Zarządzanie projektami i Zarządzanie zasobami IT**

***Funkcje poza PW*: bieglý sądowy i ekspert z branży informatyki (andrzejzalewski.pl)**

***Kontakt*: mail: andrzej.zalewski@elka.pw.edu.pl**

***Gdzie mnie znaleźć*: pok. 562 (V piętro, korytarz C)**

***Konsultacje*: czwartek 12:15 - 13:00, e-mail, MS Teams**

Organizacja

Terminy:

- *wykład: poniedziałki 16-18, czwartki 10-12*
- *laboratorium: wg podziału na grupy*
- *ćwiczenia: brak ☺*

Zapisani:

> ca. 120 osób (stan na 27.02.2022, g. 9:00)

Proporcje obciążenia: laboratorium : wykład – 30 : 60

Zasady zaliczeń

Punkty:

- *laboratorium: 30 pkt (szczegóły ustalą prowadzący lab.)*
- *wykład: 65 pkt – 20 pkt kolokwium, 50 pkt egz. końcowy*
- *Przepisywanie ocen:*
- *tylko z porównywalnych uczelni*

Zwolnienia z egzaminu:

- *brak*

Kolokwium i egzamin

1. Suma punktów: 30 – laboratorium, 20 – kolokwium, 50 – egzamin
2. Kolokwium: 20 pkt, układ zadań do ustalenia

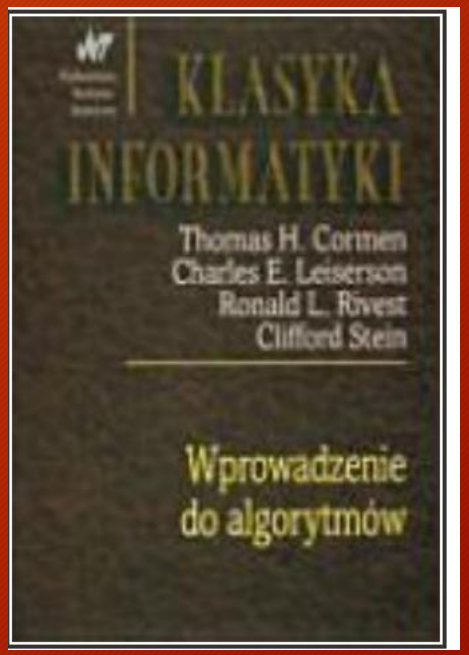
Warunki zaliczenia

Koniunkcja warunków:

- Zaliczenie laboratorium: ponad 10 pkt
- Suma punktów z laboratorium, kolokwium i egzaminu większa niż 50 pkt
- Wynik z egzaminu powyżej 20 pkt
- „Powyżej” oznacza, że każda liczba punktów większa niż progowa zalicza więc także liczba progowa z ułamkiem – np. 10,25 pkt – zalicza laboratorium, a 9,75 nie

Literatura

- *Cormen i in. Wprowadzenie do algorytmów. WNT lub PWN (najnowsze wydanie).*
- *Sensowne materiały z sieci*
- *Inna wskazana podczas zajęć*



„Chwyt” organizacyjny

Algorytmy

Podstawy informatyki formalnej i teoretycznej

ZAGADNIENIA WPROWADZAJĄCE

Motto: *„W informatyce wszystko „kręci” się wokół danych”*

Lista zagadnień

Przechowywanie danych

- Pamięć komputera i język maszynowy
- Klasyfikacja struktur danych
- Mechanizmy zarządzania pamięcią
- Persystencja (trwałe przechowywanie danych)
- Równoległy dostęp do danych

Przetwarzanie danych

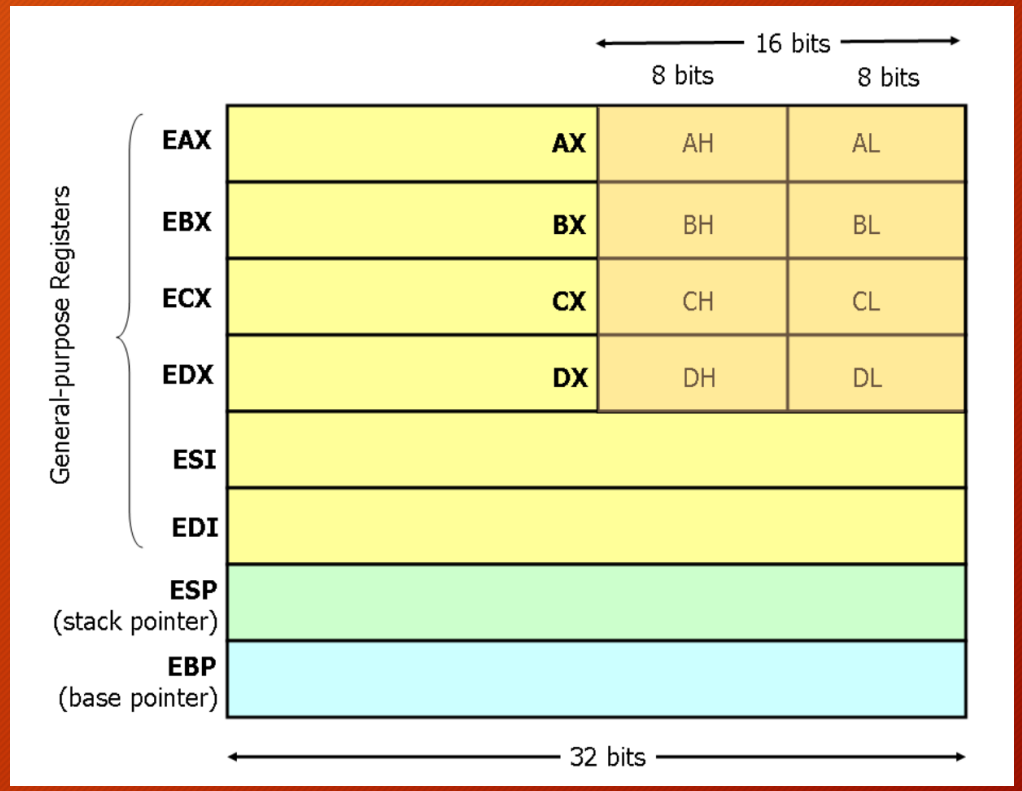
- Imperatywne języki programowania
- Problemy obliczeniowe i algorytmy

Pamięć komputera

Pamięć operacyjna

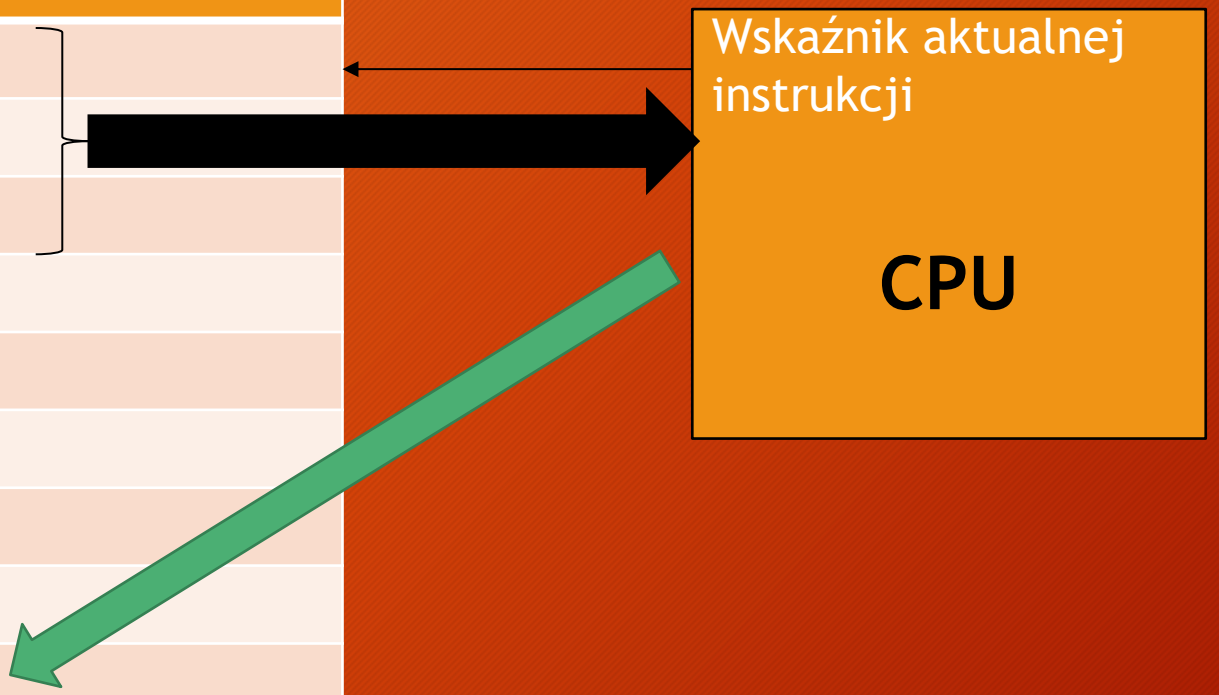
ADRES	Pamiętana wartość
0000	FF
0001	AA
0002	AB
…	…
EF02	01
EF03	12
EF04	1C
EF05	20
EF06	FE

Rejestry procesora (np. x86)



Program i dane w pamięci

ADRES	Pamiętana wartość
0000	FF
0001	AA
0002	AB
…	…
EF02	01
EF03	12
EF04	1C
EF05	20
EF06	FE



Program (kod) maszynowy

- Standardowe struktury danych (dostępne dla użytkownika):
  - Liczby całkowite (zawsze) – rejestry CPU, pamięć operacyjna
  - Liczby zmiennoprzecinkowe (rzeczywiste) – rejestry CPU, pamięć operacyjna
  - Flagi binarne – w komórkach pamięci CPU
- Instrukcje procesora (np. dodawanie, odejmowanie, porównanie, przesunięcie) są zapisywane jako określone liczby (opcodes – operation codes)

Program (kod) maszynowy (cd.)

- Wykonywanie operacji wymaga załadowania odpowiednich danych do rejestrów, a po operacji zapisania wyniku w pamięci – np.

  mov *addr1*, EAX

  mov *addr2*, EBX

  sub EAX, EBX

  mov EAX, *addr3*

- Instrukcje sterujące:
  - Skok bezwzględny pod adres – np. jmp addr
  - Wywołanie i powrót z procedury (!!!)
  - Instrukcje skoku warunkowego

Program (kod) maszynowy (cd.)

- Skoki warunkowe – schemat:
  - Operacja ---> ustawia odpowiednią flagę
  - Sprawdzenie flagi i przejście do odpowiedniego adresu jeśli jest ustawiona lub jeśli nie jest ustawiona
  - Przykład:
  - cmp eax, ebx
  - jle *addr* [jump less equal]
- Dla zainteresowanych przegląd języka maszynowego procesorów rodziny x86: https://www.cs.virginia.edu/~evans/cs216/guides/x86.html

Refleksja nr 1

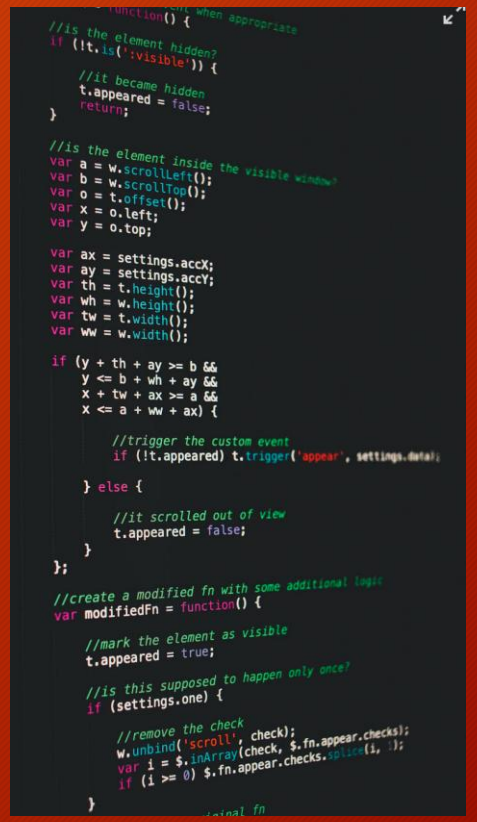
- Jak się spojrzy na finalny budulec to ma się wrażenie, że oprogramowanie które budujemy jest czymś niezwykłym!

Refleksja nr 2

Aby nauczyć się dowolnego imperatywnego języka programowania należy poznać:

- Instrukcje (operacje)
- Typy i struktury danych
- Mechanizmy zarządzania i gospodarowania pamięcią

tego języka programowania.



Klasyfikacja struktur danych

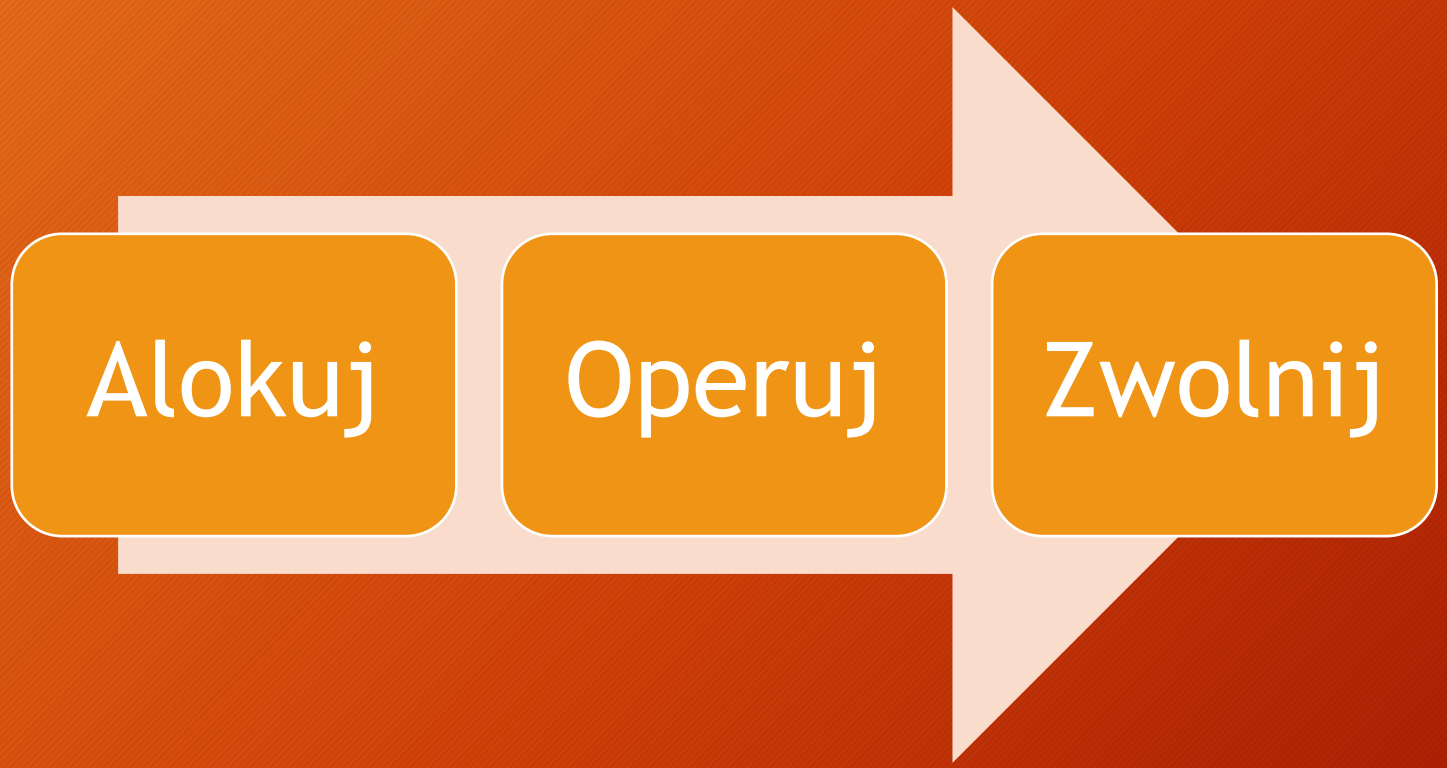
Klasyfikacja nr 1. Wg zawartości

- Proste – zmienne typów prostych (np. całkowite, rzeczywiste, znaki)
- Złożone – struktura składa się z wielu zmiennych prostych lub złożonych (tablica, klasa, struktura, rekord, lista, kolekcja)

Klasyfikacja nr 2. Wg zajmowanego miejsca w pamięci

- Statyczne – zajmują stałą wielkość pamięci w czasie całego działania programu
- Dynamiczne – zajmują zmienną ilość miejsca w pamięci (mogą się rozrastać i kurczyć)

Zarządzania pamięcią – *jawne* (1)

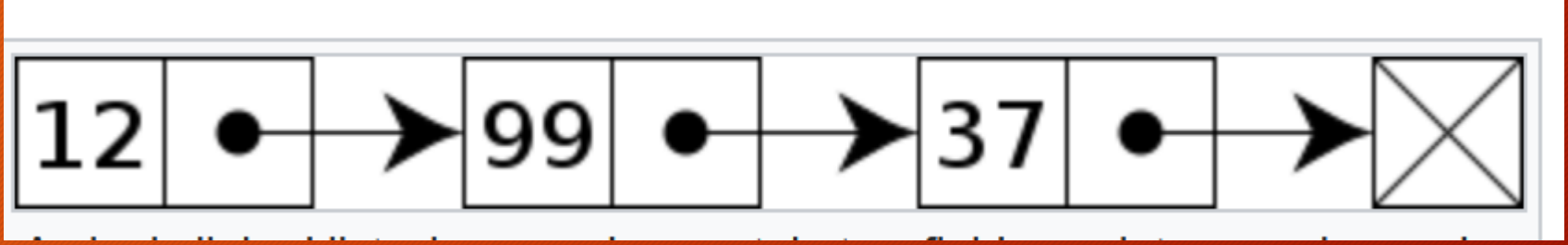


- Pascal
- C
- C++
- Java (bez zwolnij – częściowo jawne)

Zarządzania pamięcią – *jawne* (2)

Programista musi dbać o:

1) Nie przekraczanie granic przydzielonej pamięci (np. w tablicy)
2) Rozszerzanie przestrzeni zajmowanej przez strukturę (ważne w przypadku tablic)
3) Niepogubienie kawałków alokowanej pamięci (brak garbage collectora!)
4) Usuwanie niepotrzebnych kawałków zaalokowanej pamięci
5) Ciągłość powiązań



Zarządzania pamięcią – *niejawne*

1. Środowisko wykonawcze wykrywa przekroczenie rozmiaru struktury
2. O powiększanie obszarów zajmowanych przez tablice dba środowisko wykonawcze
3. Zgubione fragmenty pamięci wyszukuje i zwalnia garbage collector
4. Brak operacji zwalniania pamięci – wystarczy zgubić link do danego elementu struktury i trafia on do śmietnika (garbage)
5. Problem ciągłości powiązań uchylony w Pythonie przez gotowe struktury typu kolekcje, listy i iteratory (niemniej można tworzyć struktury bazujące na powiązanych referencjami węzłach)

Warianty zarządzania pamięcią: zyski i straty

- ????

Persystencja

Jak zapamiętać dane użytkowane przez oprogramowanie na czas, kiedy ono nie działa?

- Trwały nośnik: taśma (archaiczne, za wyjątkiem kopii zapasowych), dysk magnetyczny (teraz HDD), dysk elektroniczny (SSD, pendrive)
- Klasycznie i do tej pory: PLIK, plik – *clue*: format
- Bazy danych (struktury powiązanych tabel /relacji/, zbiory dokumentów)

Persystencja (cd.)

Dla pobudzenia wyobraźni:

- Jak zorganizować przechowywanie zawartości przesyłanej pocztą elektroniczną np. w pliku danych
- Jak przechowywać w jednym pliku różne dane np. listy i terminy kalendarza

Równoległy dostęp do danych

- Problemem występujący w wielu sytuacjach praktycznych
  - Bazy danych: równolegle jest wykonywanych wiele wyszukiwań, ale i wiele zapisów
    - Jak rozwiązać problem równoległego zapisu do tych samych danych?
    - Jak rozwiązać problem częściowych wyników obliczeń w wynikach wyszukiwania
  - W „normalnych” programach – próby jednoczesnego zmieniania tych samych danych przez kilka wątków / procesów

Zagadnienia wprowadzające: *Przetwarzanie danych*

Problem obliczeniowy – algorytm – program

Schemat ogólny:

Przykład:

Języki programowania

- Budulec/środek wyrazu: języki programowania
- Języki imperatywne (większość stosowanych w praktyce)
- Inne:
  - Języki funkcyjne (Haskell, F#, Lisp)
  - Języki programowania w logice (PROLOG)

Składowe imperatywnych języków programowania

- Składowe języków imperatywnych
  - Składnia
  - System typów danych
  - Instrukcje:
    - pętle
    - wyrażenia
    - rozejścia warunkowe (if, case),
    - skoki (przestrzałe, właściwie wyeliminowane)
    - wyjątki (obecne)
    - organizujące program (funkcje)

Język programowania – dwa elementy

Składnia → Semantyka

https://docs.python.org/3.8/reference/grammar.html

Algorytmy – uwagi wstępne

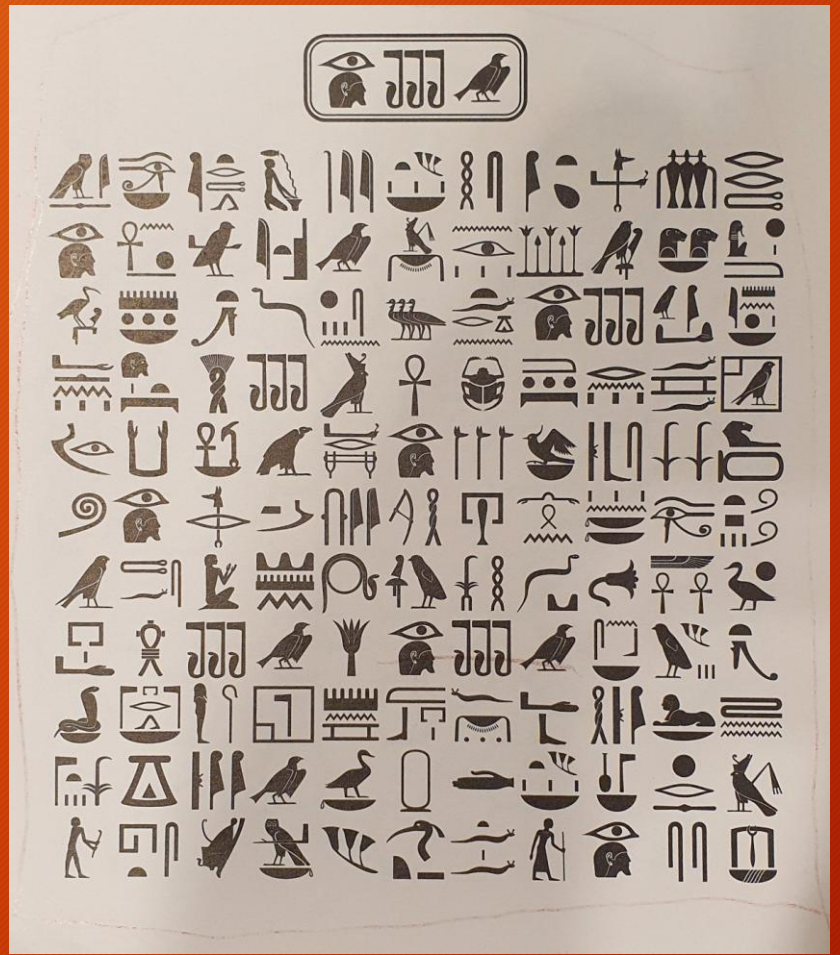
Algorytm – źródłosłów

W średniowieczu: abacist / algorist

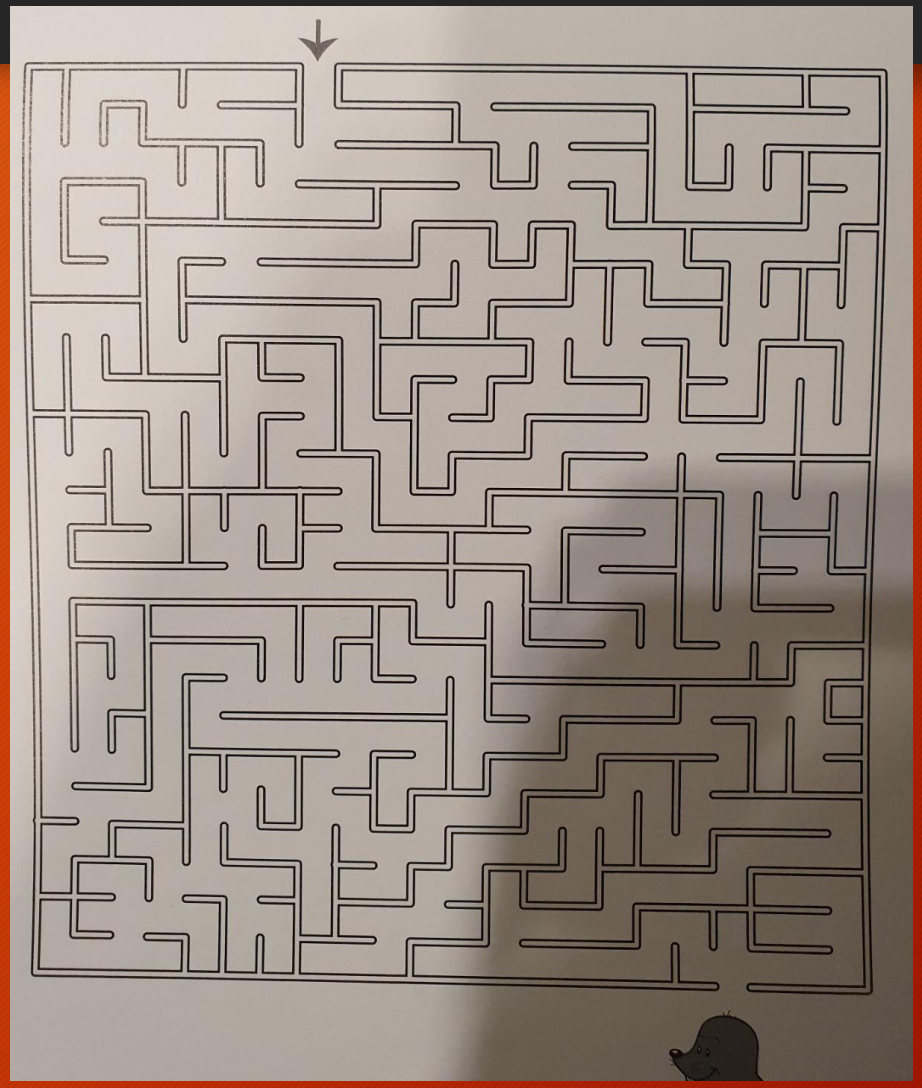
Od nazwiska arabskiego autora traktatu o algebrze (arytmetyce) i nazwy geograficznej rejonu Jeziora Aralskiego

https://library.ethz.ch/en/locations-and-media/platforms/virtual-exhibitions/fibonacci-un-ponte-sul-mediterraneo/fibonaccis-significance-for-the-present-day/dispute-between-abacists-and-algorists.html

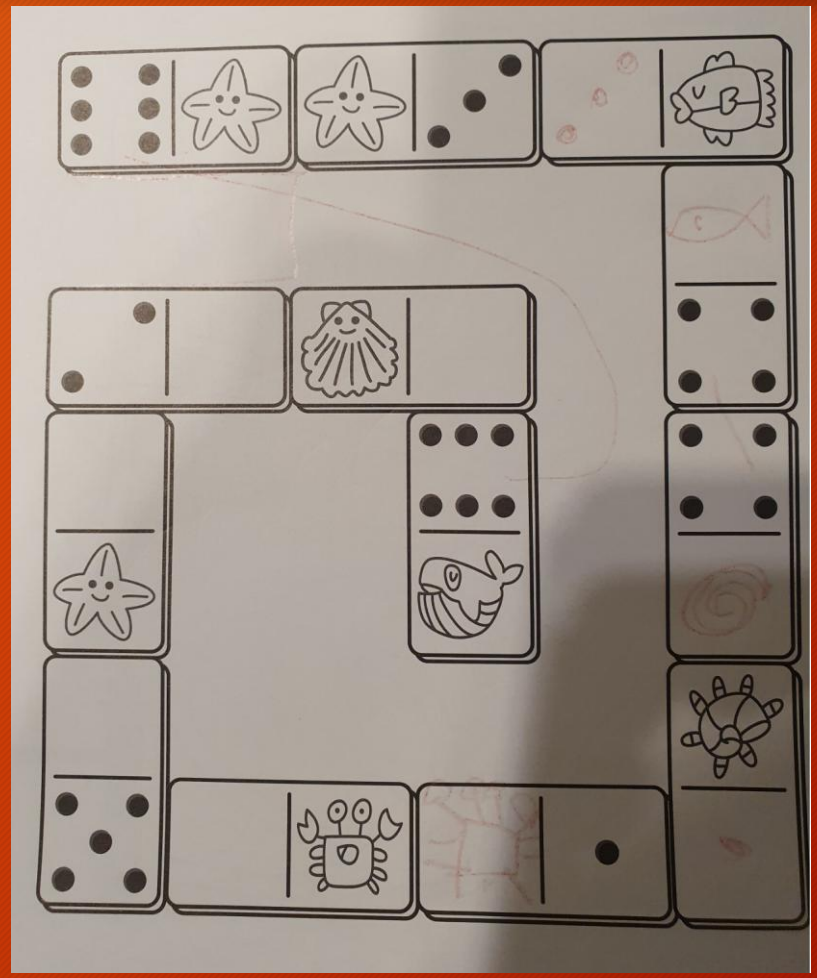
Problem 1



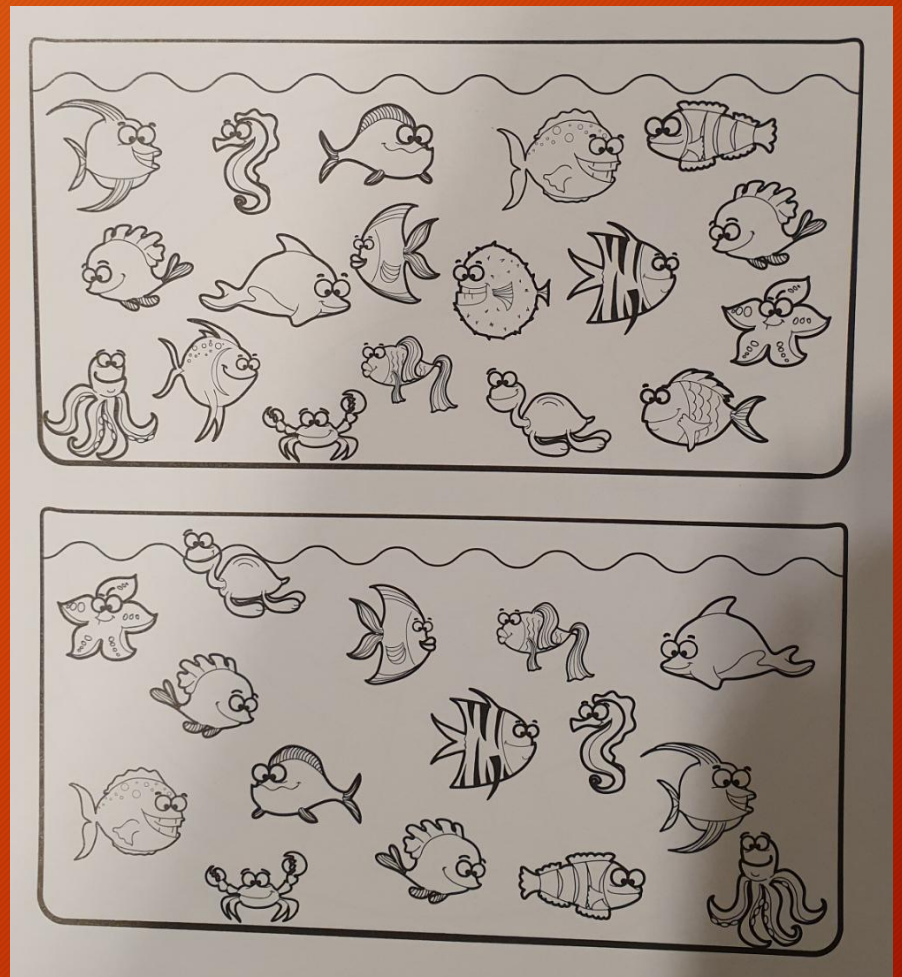
Problem 2



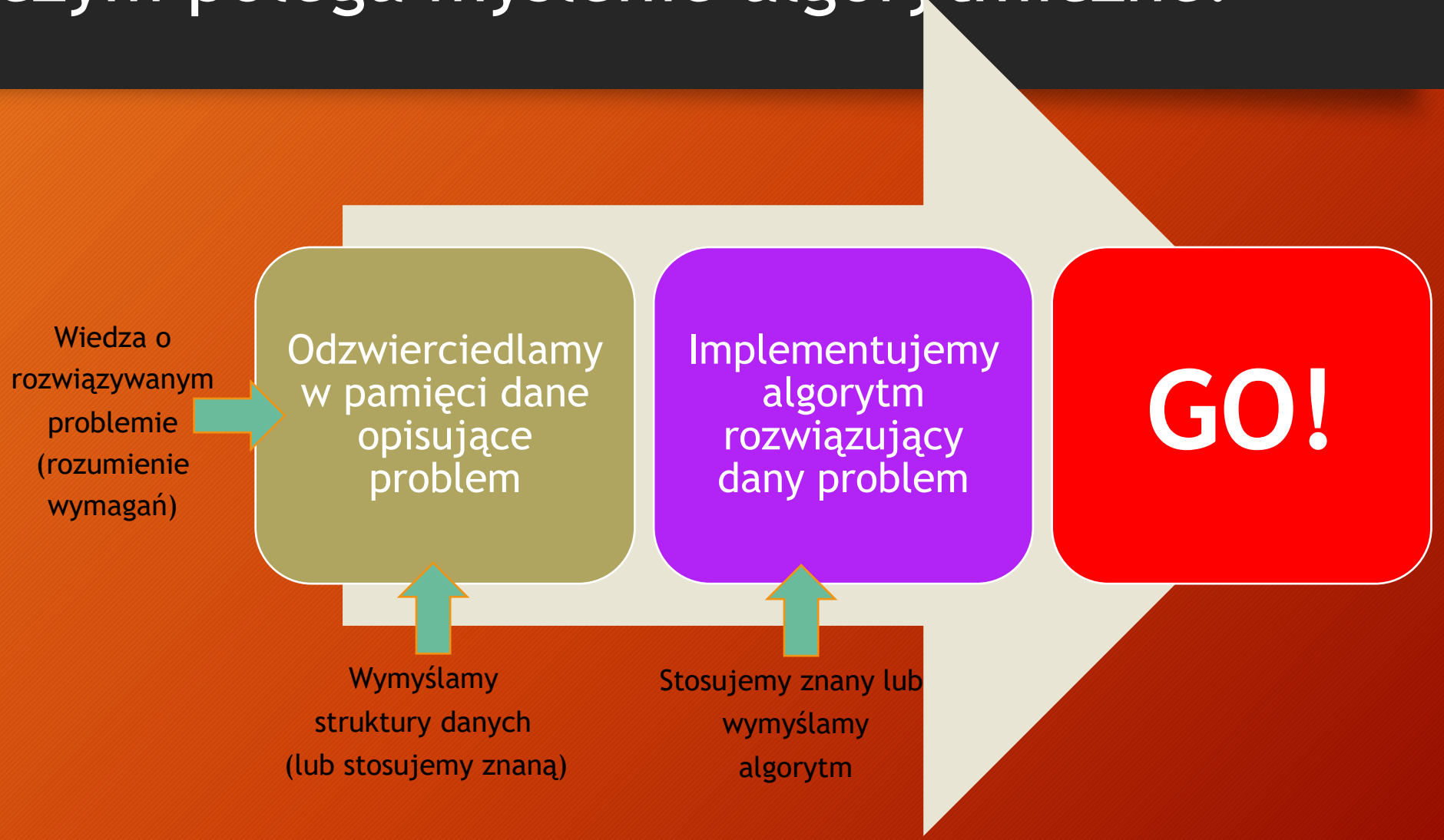
Problem 3



Problem 4



Na czym polega myślenie algorytmiczne?



Algorytm

- Algorytm – każda mechaniczna procedura obliczeniowa
- Mechaniczny – czyli jaki?

Algorytm – czy łatwo wynaleźć?

- ***TAK*** – programować może każdy, nie święci garnki lepią. Programują nie tylko geniusze matematyczni

- ***NIE*** – *proszę wymyśleć algorytm znajdowania największego wspólnego dzielnika (alg. Euklidesa) lub algorytm znajdowania najdłuższego wspólnego podciągu*

Algorytm – w czym tkwi trudność wynalezienia?

- ***PROBLEM, JEGO ROZUMIENIE I ODPOWIEDNIE PRZEDSTAWIENIE!!!***
- Zrozumienie problemu największego wspólnego dzielnika
  - gcd(a, b) = gcd (b, a mod b)

Właściwości algorytmów: jak porównywać algorytmy?

Ch-ka wykonania programu

- T – czas działania programu dla danego zadania (np. danych wejściowych)
- V – wielkość pamięci potrzebnej do wykonania programu dla danego zadania (np. danych wejściowych)

Przykład analizy wydajności

```
Suma = 0
For i = 1 to N
    For k = zaokr.w.dół(i/2) to N
        Suma = Suma + i*k
```

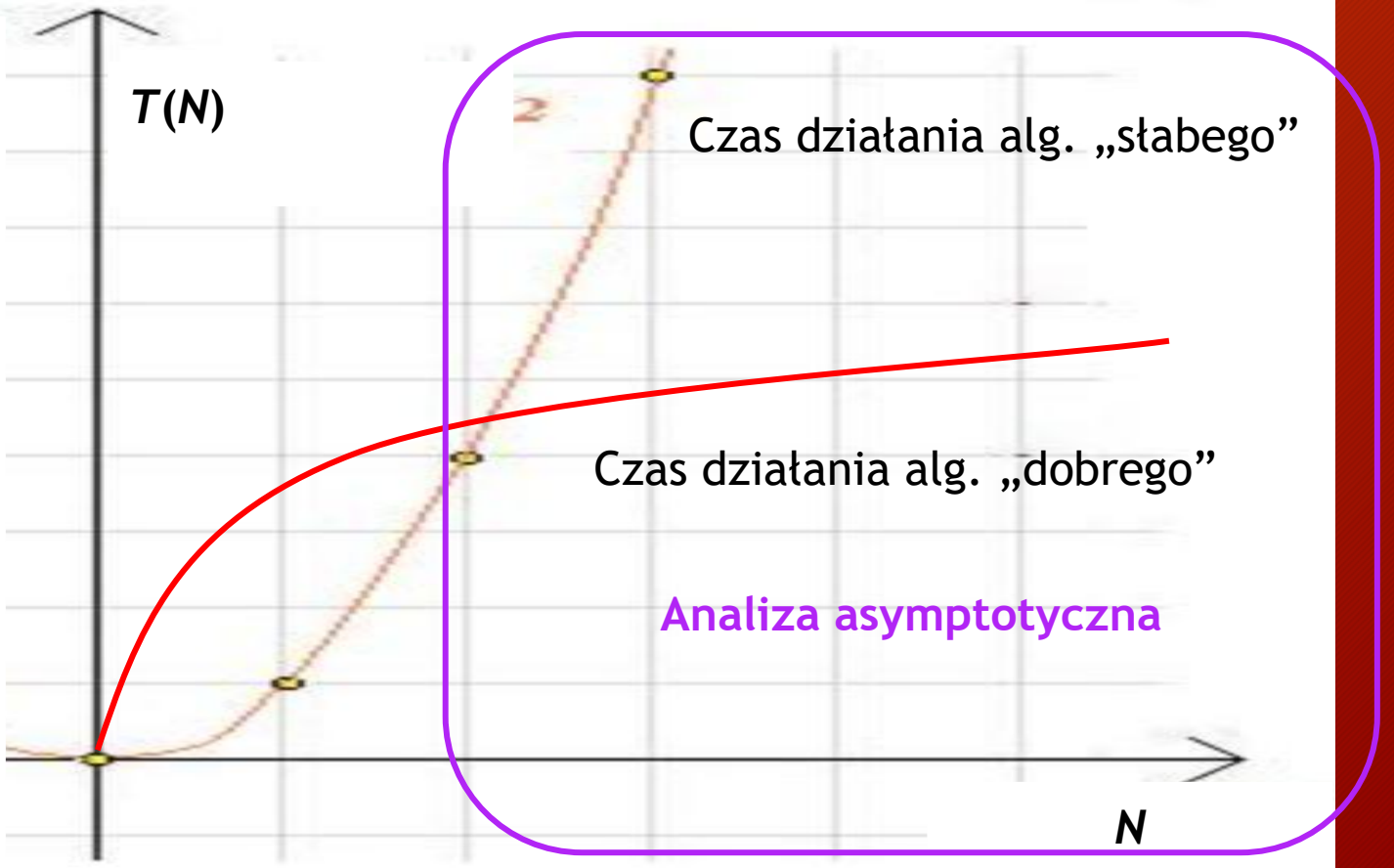
V(N) = kilka bajtów

T(N) = suma N wyrazów ciągu arytm. (N+1), (N), … (N/2)

Czyli $(N+1 + N/2) / 2 * N = a N^2 + bn + c$

Wydajność asymptotyczna vs.dla małych zadań

Wydajność średnia algorytmów sortowania przez wstawianie i quick sort dla małych i dużych zbiorów

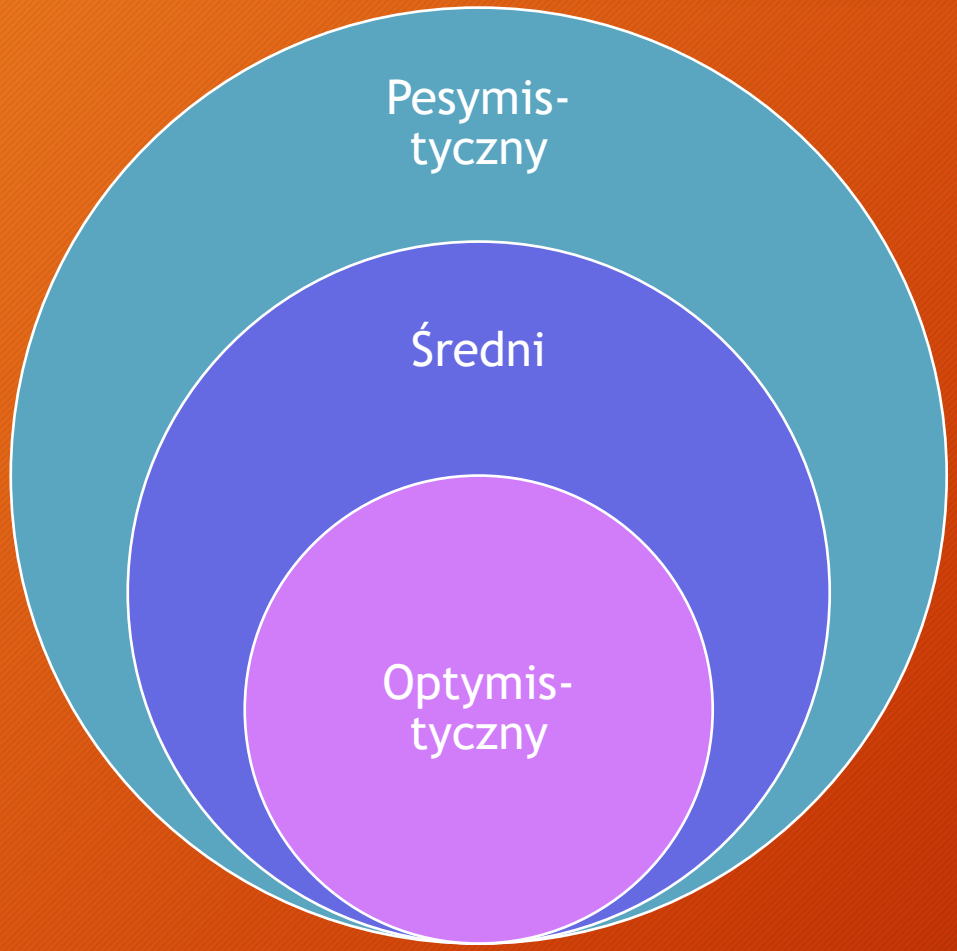


Ch-ka wydajnościowa algorytmu

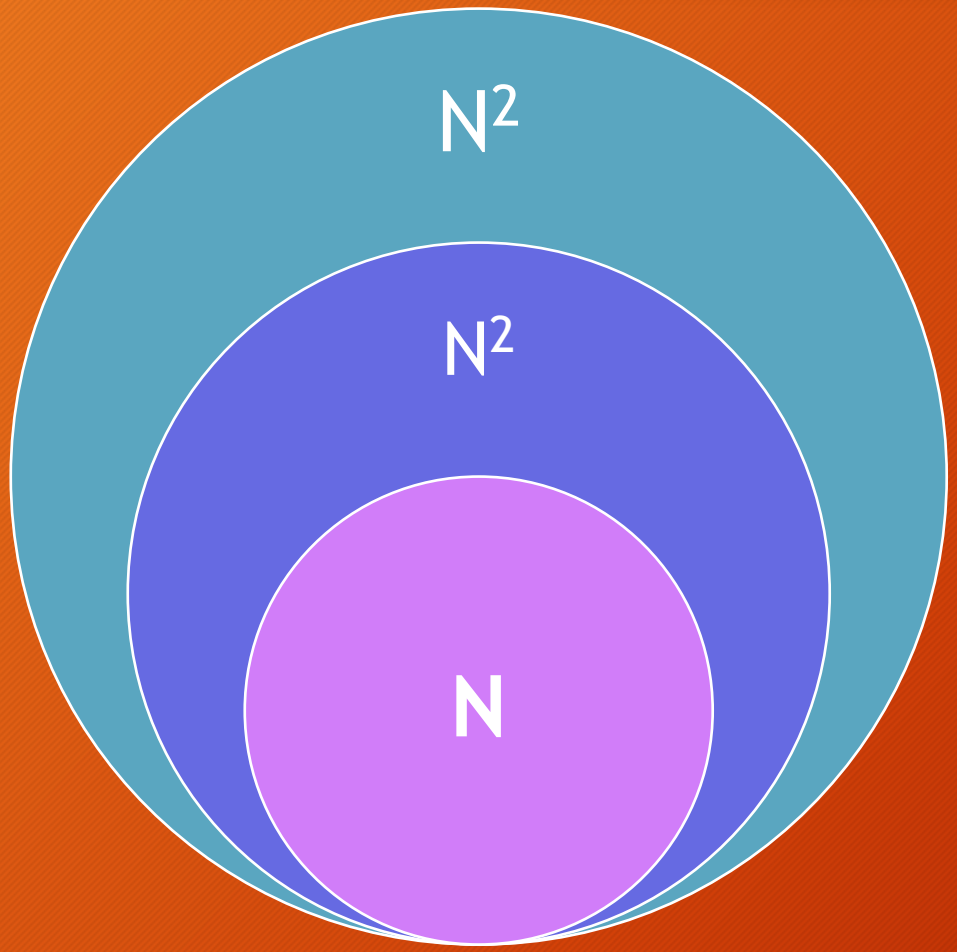
$T(N), V(N) \overset{N \to \infty}{\dashrightarrow} f(N)$,

czyli jak szybko zwiększa się czas i zajętość pamięci w zależności od wymiaru zadania

Warianty analizy wydajności



Wydajności – sortowania przez wstawianie



Wymiar zadania

- Czasem nieoczywisty – jaki jest wymiar zadania w problemie największego wspólnego dzielnika?
- Typowo: liczba danych wejściowych

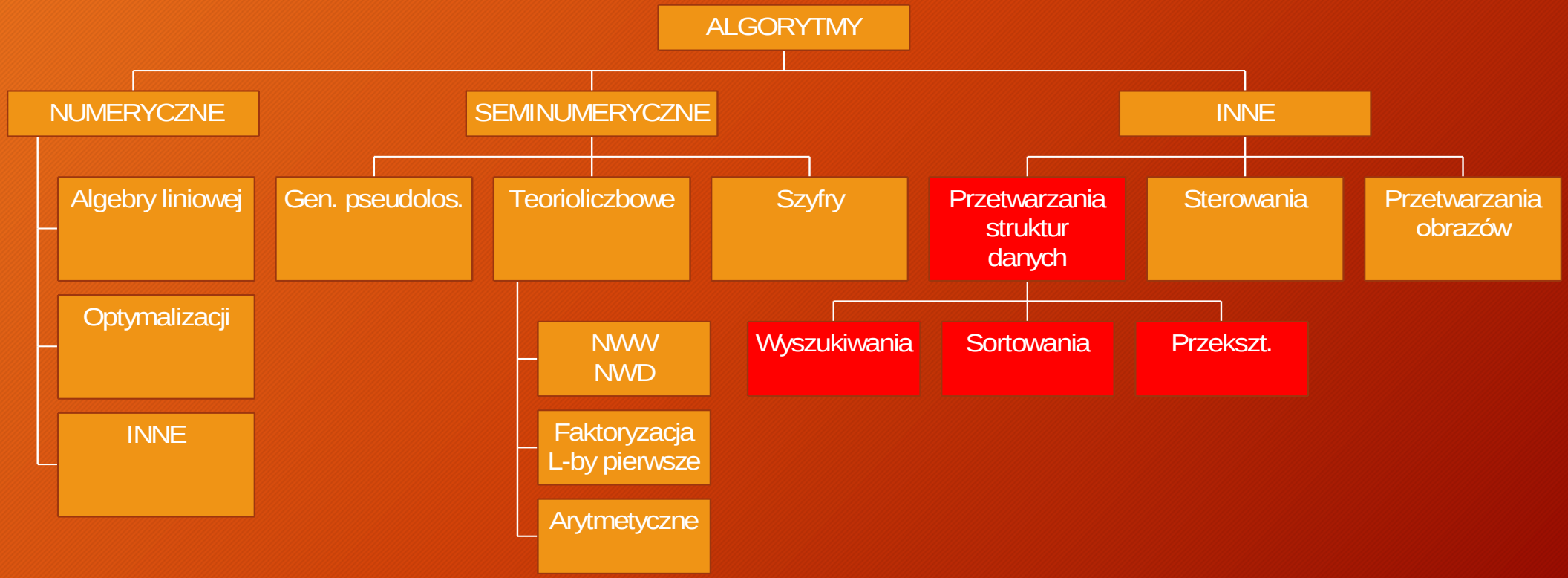
Notacje asymptotyczne

- Notacja: (Θ, O, Ω),
  - $\Theta(g(N))$ – rodzina funkcji $f(N)$, których wzrost jest „nie szybszy i nie wolniejszy niż szybkość wzrostu funkcji $g(N)$”
  - $O(g(N))$ – rodzina funkcji rosnących nie szybciej niż $g(N)$
  - $\Omega(g(N))$ – rodzina funkcji rosnących nie wolniej niż $g(N)$
- Notacja (o, ω)
  - notacja $o(g(N))$ – zbiór funkcji pomijalnych przy $g(N)$ dla dużych N (w granicy)
  - notacja $\omega(g(N))$ - zbiór funkcji $f(N)$, przy których $g(N)$ jest pomijalna [stop]

Złożoności obliczeniowe

Złożoność	Trudność problemu	Klasa algorytmu
Stały	Bardzo, bardzo proste.	Złoty strzał
Logarytmiczny	Bardzo proste.	Divide et impera
Liniowy	Proste.	Przeglądanie struktur linearnych
Liniowo-logarytmiczne	Proste.	Divide et impera + przeglądanie linearne
Wielomianowy (st. wiel. > 1)	Łatwe, trudne i bardzo trudne.	Ogromna większość prakt.
Wykładniczy	Bardzo, bardzo trudne.	Przeglądanie struktur wykładn.

Klasyfikacje algorytmów – wg dziedziny problemu



Klasyfikacja alg. numerycznych

Wg własności znalezionego rozwiązania

- Znajdujące rozwiązanie dokładne
  - założenie: jesteśmy w stanie zdefiniować je bezpośrednio lub jego własności (częściej)
- Znajdujące rozwiązanie przybliżone
  - Zbieżne do wyniku dokładnego
    - założenie jak wyżej, zbieżność trzeba wykazać
  - Heurystyczne (znajdują rozwiązanie, nie koniecznie spełniające wszystkie warunki)
    - stosowane w „*trudnych*” zadaniach (np. w problemach kombinatorycznych – układanie planu zajęć, problem komiwojażera)

Wg strategii konstrukcji

- Dziel i rządź
- Zachłanne
- Niezachłanne (rozsądne)
- Losowe

ZAPRASZAM!

Politechnika Warszawska

Algorytmy i struktury danych

Odc. 2. Listy i wyszukiwanie

Dr hab. inż. **Andrzej Zalewski**

Wydział Elektroniki i Technik Informacyjnych
POLITECHNIKA WARSZAWSKA

Lista

Lista złożona z element $\in$ *Elementy to:*

- *lista-pusta*
- *lista-niepusta = element, lista*
- w konsekwencji: lista = lista , lista

Inna interpretacja:

lista = {(pozycja-na-liście, element): pozycja-na-liście $\in N$, *element* $\in$ *Elementy }*

- Przykłady:
  - Lista = 1, 5, 10, 2, 9, 8, 7, 2, 1

Lista operacje

- *Znajdź*(*Lista*, *element*)
- *Znajdź_k-ty_element*(*Lista*, *k*)
- *Następny*(*Lista*, *element*)
- *Poprzedni*(*Lista*, *element*)
- *Wstaw*(*Lista*, *element*, *pozycja*)
- *Usuń*(*Lista*, *element*)
- *Początek*(*Lista*)
- *Koniec*(*Lista*)

Element – albo wskaźnik (referencja) elementu, albo indeks element

W Pythonie…

- Wszystkie operacje w standardzie – niczego nie trzeba implementować
- Wyszukanie:
  - *Element* **in** *lista*
  - *Lista.index(element)*
  - *Lista.count(element)*
- Usuwanie: lista.remove(element), lista.pop(index)
- *Listy w liście etc.*

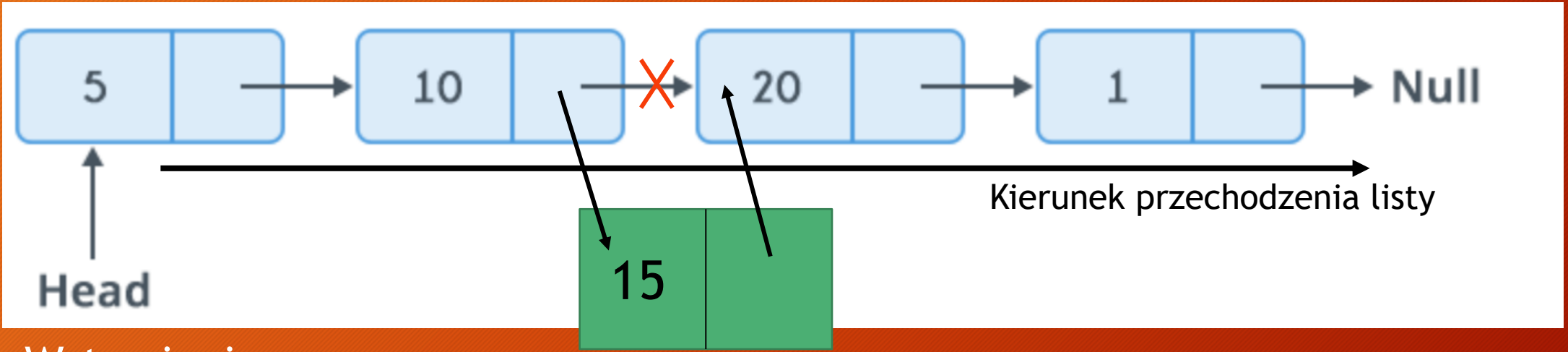
Implementacje listy – tablica

Indeks w tablicy	Wartość
0	ALA
1	MA
2	KOTA
3	A
4 [N-1]	<end>
…	
M-2	
M-1	

Właściwości:

- Tam, gdzie znamy indeks – operacje wyszukiwania w czasie stałym
- Wyszukanie konkretnej wartości średnio $\Theta(N)$;
- Usunięcie elementu: albo średnio $\Theta(N)$ (usuwamy i przesuwamy elementy), albo $\Theta(1)$ (usuwamy i oznaczamy miejsce jako puste)
- Dodanie elementu: czas stały (do wypełnienia tablicy), przepisanie tablicy, pominięcie pustych (zawsze: $\Theta(N)$).

Implementacja łańcuchowa, jednokierunkowa



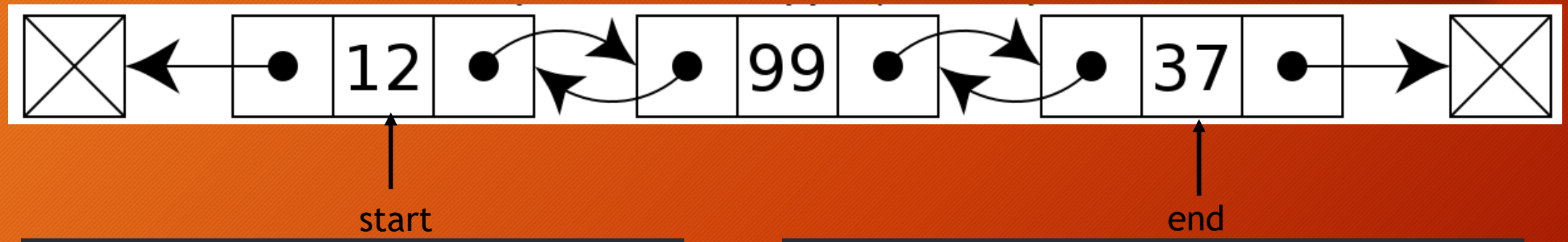
Wstawianie:

1) utwórz w pamięci wstawiany węzeł
2) zlokalizuj miejsce wstawiania
3) przepnij wskaźniki (referencje)

Usuwanie:

1) Znajdź usuwany węzeł
2) przepnij wskaźniki (referencje)
3) Zwolnij pamięć (C/C++/Pascal)

Implementacja łańcuchowa dwukierunkowa



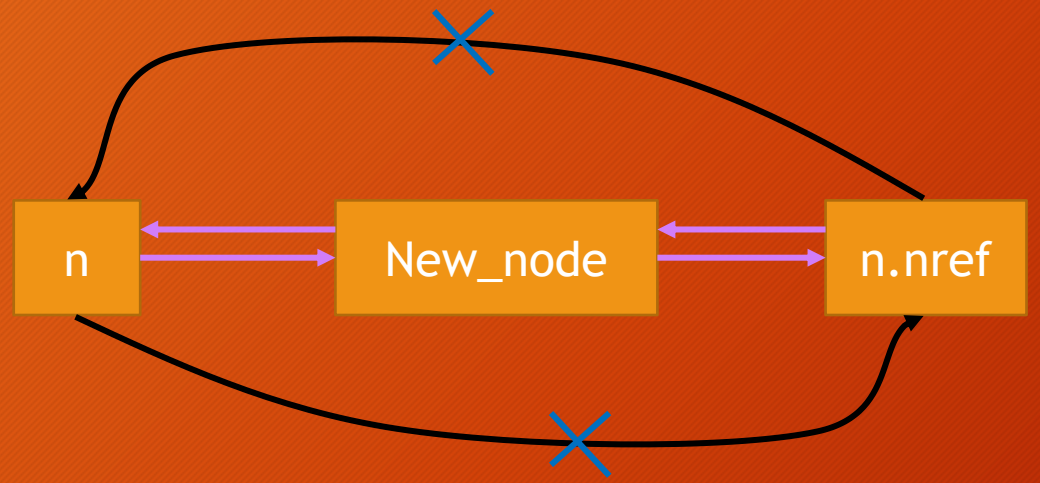

```
class Node:
    def __init__(self, data):
        self.item = data
        self.nref = None
        self.pref = None
```

```
class DoublyLinkedList:
    def __init__(self):
        self.start_node = None
```

Przykładowy kod:
https://stackabuse.com/doubly-linked-list-with-python-examples/

Implementacja w Pythonie (dla hobbystów)

```
class Node:
    def __init__(self, data):
        self.item = data
        self.nref = None
        self.pref = None
```




```
def insert_after_item(self, x, data):
    if self.start_node is None:
        print("List is empty")
        return
    else:
        n = self.start_node
        while n is not None:
            if n.item == x:
                break
            n = n.nref
        if n is None:
            print("item not in the list")
        else:
            new_node = Node(data)
            new_node.pref = n
            new_node.nref = n.nref
            if n.nref is not None:
                n.nref.prev = new_node
            n.nref = new_node
```

Wyszukiwanie w listach

- Lista nieuporządkowana: przeglądanie kolejnych elementów
  - Czas optymistyczny?
  - Czas średni?
  - Czas pesymistyczny?
- Lista uporządkowania:
  - Wyszukiwanie-binarne(*L*):

    1) Jeśli lista ma 1 element i jest on równy elementowi szukanemu zwróć ten element

    2) Podziel listę *L* na pół (*L1*, *L2*)

    2) Sprawdź po wartości, w której połowie *Ln* listy jest szukany element (*L1* czy *L2*)

    3) „Odpal” wyszukiwanie-binarne(*Ln*)

Czas działania…

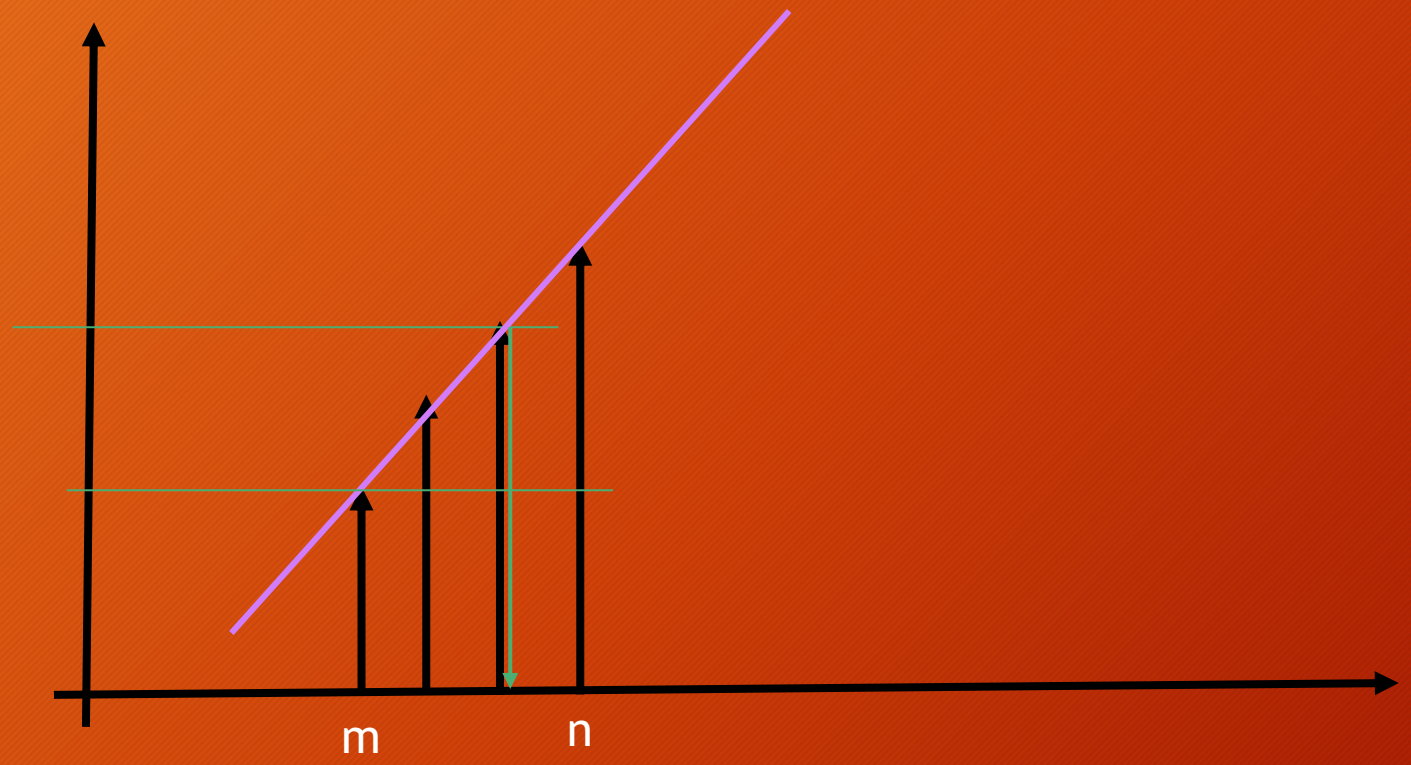
- Dla tablicy czas logarytmiczny (!!!) /pesymistyczny, średni optymistyczny/
- Dla listy łańcuchowej???
  - Jaki jest czas znajdowania połowy pozostałej do zbadania połowy listy?
  - Ile tych połówek jest do zbadania?
  - N/2, N/4, N/8, N/16…. ---> nieskończony ciąg geometryczny (oszac. górne): S = N * ½ / (1 – ½) = N…
  - Czyli czas jest liniowy przypadku… /p, ś, o/

Wyszukianie binarne w tablicach

- Dana jest tablica uporządkowana T[1..N] i szukany klucz k_s, *niech* $m=1$, $n=N$
- r=⌊(m+n)/2⌋ - pkt. podziału przedziału (na pół)
- Sprawdzamy T[r] < k_s
  - jeśli spełnione – wykonujemy sprawdzenie w podtablicy T[r .. n] (tzn. m:= ⌊(m+n)/2 ⌋),
  - w przeciwnym razie w podtablicy T[m .. r +1] (tzn. n= ⌊ (m+n)/2 +1⌋)
  - trzeba jeszcze dopisać warunek stopu…
- Działa w czasie logarytmicznym /średnio i pesymistycznie/ ($\sim\log_2 N$)

Wyszukiwanie binarne, interpolacyjne

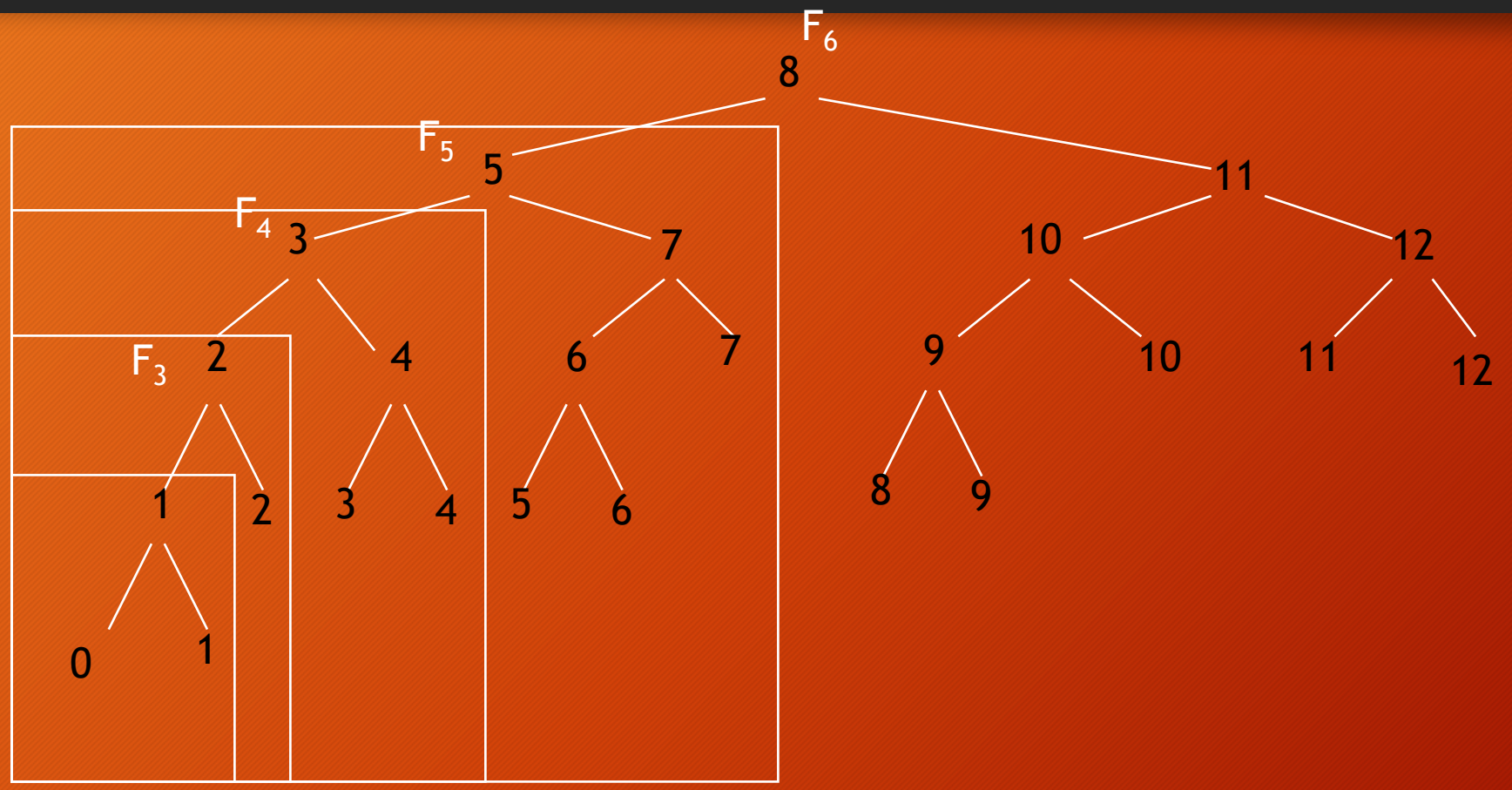
- Zmieniamy sposób podziału przedziału indeksów <m, n> – próbkujemy *T*[*r*], gdzie indeks *r* wyznaczony proporcjonalnie do odległości k_s od *T[m], w przedziale* k_m, k_n
  - $r = m + [(k_s - T[m]) / (T[m] - T[n]) * (n - m)]$
- *Średnio: log log n*
- *Pesymist: ~ n*
- *Kiedy?*
  - *Dane rozłożone równomiernie*


m
n

Wyszukiwanie metodą drzewa Fibbonacciego

- **Wyszukiwanie metodą drzewa Fibonacciego**
- Liczby Fibonacciego $F_0=0$, $F_1=1$, $F_2=1$, $F_{k+1} = F_k + F_{k-1}$, czyli $F_3=2$, $F_4=3$, $F_5=5$, $F_6=8$
- Drzewo Fibonacciego w tablicy:
  - rzędu k=0, 1 po prostu 0
  - rzędu k>=2 korzeniem jest element F_k, lewym poddrzewem drzewo rzędu F_{k-1}, prawym poddrzewem drzewo rzędu F_{k-2} z elementami o indeksach powiększonych o F_K

Przykład drzewa Fibonacciego [stop]



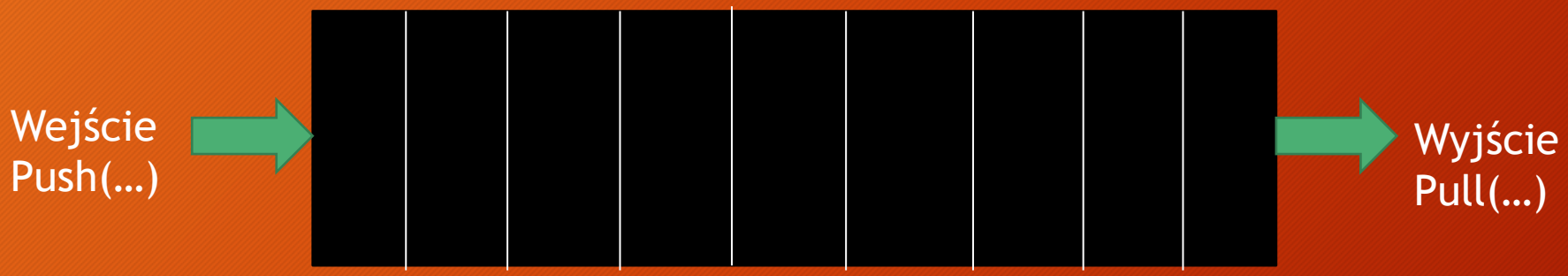
$k=6, F_6=8$

Poruszanie się po drzewie Fibonacciego

- Niech $i=F_k$, $p=F_{k-1}$, $q=F_{k-2}$ (*i – na początku wskazuje korzeń*)
- Przejście do lewego poddrzewa
  - $i := i - q$ ($F_k=F_{k-1}+F_{k-2} => F_{k-1}=F_k - F_{k-2}$)
  - $(p,q) = (q, p - q)$ (F_{k-2}, F_{k-3})
- Przejście do prawego poddrzewa
  - $i := i + q$ (*zgodnie z definicją drzewa*)
  - $p := p - q$ /F_{k-3}/ , $q = 2q - p$ [$q - (p-q)$] /$F_{k-4}= F_{k-2} - F_{k-3}$/

Stos i kolejka

- FIFO - kolejka



Stos

Wierzch Push(…) / Pull

- (3)(1) E()
- (2)(1) D()
- (1) B()

Spód

1) A(….)
    1) B(…)
    2) C(…)

2) B(…)
    1) D(…)
    2) …
3) D(…)
    1) E(…)

Wyszukiwanie w tablicach (listach) z haszowaniem

- Zamiast porównywać wyszukiwany klucz, z kluczami w tablicy /drzewie (itp.)/ znajdujemy pozycję w tablicy na podstawie samej wartości klucza, tzn.
  - dana jest funkcja $H(k)$: K -> I, gdzie:
    - K – zbiór kluczy,
    - I – zbiór indeksów
  - zauważmy, że zwykle: $|K| >> |I|$
- szansa: wyszukiwanie/wstawianie/usuwanie w czasie stałym (!)
  - jeśli wyznaczenie $H(k)$ nie jest „czasochłonne” obliczeniowo
- K – zwykle zbiór liczb naturalnych

Funkcje haszujące – wymagane właściwości

- Równomierne rozrzucanie:
  - dla losowo wybranego klucza każda pozycja w indeksie jest jednakowo prawdopodobna niezależnie od odwzorowania innych kluczy
- „Całkowite wypełnianie” zbioru I
- W ogólności dobór funkcji haszującej zależy od właściwości zbioru kluczy
  - np. dla $k \in <0,1)$ [klucze – liczby rzeczywiste], $H(k)=\lfloor k\ m \rfloor$, gdzie, |I| = m, haszuje równomiernie po tablicy *m*-elementowej
- Typowo: I = N

Haszowanie

- Dla kluczy nie całkowitoliczbowych – przekształcamy klucz w liczbę naturalną
  - dla ciągów znaków: $H(k) = (h_1(c_1) + h_2(c_2) + ...) \bmod m$, h_1, h_2 ... – pewna funkcja mieszająca
  - dla ciągów składających się z kilku liczb sklejamy poszczególne fragmenty klucza korzystając z operacji *mod w* lub *xor*
  - traktujemy tekst lub jego fragment jako liczbę w określonym systemie pozycyjnym – np. ab – w systemie 24-kowym...
- Dalej na wykładzie rozważamy już tylko *klucze* będące liczbami naturalnymi

Modularna funkcja haszująca

- $H(k) = k \bmod m$ /*k* – klucz całkowitoliczbowy/
  - problem dobór *m*:
    - dobre *m* – liczba pierwsza,
    - *m* parzyste - zły wybór – miesza po połowie przestrzeni 0...m (*k* – parzyste => *H*(*k*) – parzyste, *k* – nie parzyste => *H*(*k*) nieparzyste)
    - *m* – 2^k – obcięcie klucza do *k* najmniej znaczących bitów klucza
  - *Ogólnej reguły niema!*

Haszowanie przez mnożenie

- $H(k) = \lfloor m\ (k\ A \bmod 1) \rfloor$
  - $x \bmod 1$ – ułamkowa część x
  - $A \in (0,1)$,
  - m - *wielkość tablicy (chętnie 2^p)*
- m – mało istotne – działa dla dowolnego m

Haszowanie - kolizja

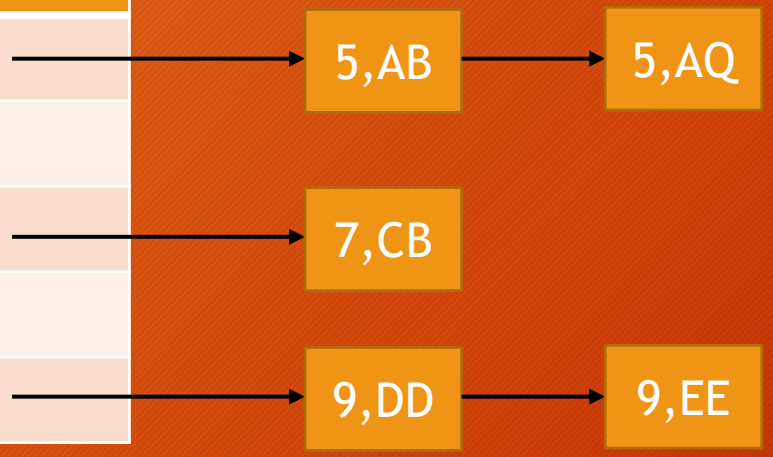
- Kolizja: $H(k_1) = H(k_2)$, przy $k_1 <> k_2$
- Jak bardzo jest prawdpodobne?
- Paradoks dnia urodzin

Rozwiązywanie kolizji

- metoda łańcuchowa
  - tablica jest *de facto* tablicą wskaźników – normalnie wskazuje zero lub jeden element, w przypadku kolizji dodajemy następne elementy do tak zaczętych list
  - oczekiwana złożoność obliczeniowa przy $n<m$: O(1)!
  - ile pesymistyczna?

Ilustracja – metoda łańcuchowa

Indeks	Wsk. / ref.
5	
6	
7	
8	
9	



Rozwiązywanie kolizji – adresowanie otwarte

- adresowanie otwarte
  - istota pomysłu: w przypadku kolizji sprawdzamy inne miejsce w tabeli, itd. aż do znalezienia wolnego miejsca
  - problem w usuwaniu….
    - trzeba oznaczać miejsca wolne, ale kiedyś zajęte
- – stosujemy funkcję $H(k, i)$, gdzie i – numer próby wyszukania/wstawienia
  - liniowe: próbkujemy kolejno: $H(k)$, $H(k) -1$, $H(k) - 2$ itd. jeśli dotrzemy do pustego miejsca nie znalazłszy wcześniej klucza – klucza nie ma w tablicy – możemy wstawić nowy klucz lub stwierdzić brak klucza szukanego – $H(k, i) = (H'(k) + i) \bmod m$
  - haszowanie kwadratowe $H(k, i)=(H'(k)+c_1 i+c_2 i^2) \bmod m$, i – numer próby, c_1, c_2 – stałe całkowite;

Koniec!

Politechnika Warszawska

Algorytmy i struktury danych

Odc. 3. Sortowanie

Dr hab. inż. **Andrzej Zalewski**

Wydział Elektroniki i Technik Informacyjnych

POLITECHNIKA WARSZAWSKA

Algorytmy sortowania przegląd tematyki

- Tablice / listy
- Ograniczenie efektywności sortowania z porównywaniem elementów
- Algorytmy proste
  - przez prosty wybór
  - przez wstawiania
  - przez prostą zamianę (bąbelkowe)
- Algorytm Shella
- Algorytmy asymptotycznie efektywne
  - przez podział i scalanie (merge-sort)
  - przez przesiewanie przez kopiec (heap-sort)
  - Hoare'a – tzw. sortowanie szybkie (quick-sort)
- Sortowanie w czasie liniowym
  - przez zliczanie (count-sort)
  - sortowanie pozycyjne (radix-sort)
  - sortowanie kubełkowe (bucket-sort)

Tablice jeszcze kilka uwag

- T: Id -> V
  - **Id** - Indeksy - typowo: Id $\subset$ N (liczby naturalne), Id $\subset N^k$ (dla tablicy *k*-wymiarowej)
  - **V** - Wartości - liczby rzeczywiste, całkowite, znaki, łańcuchy znaków, typy złożone (rzadziej), wskaźniki typów złożonych
- Przypadki szczególne
  - WEKTOR: Id = {1, ..., m}, V = R
  - MACIERZ: Id = {1, ..., m}$\times$ {1, ..., n}, V = R

Tablice reprezentacja

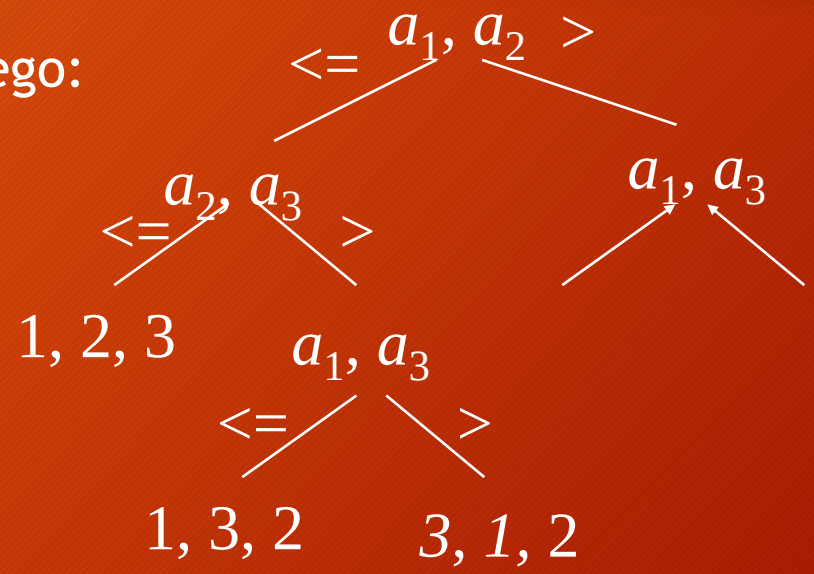
- Typowo ciągły obszar pamięci potrzebny do przechowania poszczeg. wartości
  - Tablice gęste
    - Macierze i wektory gęste
- Tablice rzadkie
  - macierze i wektory rzadkie
    - różne rozwiązania
  - np. listy par: {(Indeks, Wartość), ...}

Dolne ograniczenie czasu sortowania

z porównywaniem elementów

Dolne ograniczenie na czas sortowania z porównywaniem elementów

- Rozpatrzymy sortowanie ciągu 3 elementowego:
  - $\{a_1, a_2, a_3\}$



- Tw. **Powyższe drzewo decyzyjne ma wysokość nie mniejszą niż N $\log_2$ N**

To znaczy…

- Przy sekwencyjnym porównywaniu elementów **nie da się skonstruować** algorytmu szybszego niż N log N

Dowód tw. o wysokości drzewa decyzyjnego

- Liczba liści w drzewie = liczbie permutacji elementów sortowanej tablicy i wynosi $N!$
- Drzewo o wysokości h ma co najwyżej 2^h
- Zachodzi więc nierówność: $N! \leq 2^h$, *czyli*:
  - $h \geq \log_2 N!$,
  - *wzór Stirlinga* $N! > (N / e)^N$, $e = 2{,}718$
  - $h \geq \log_2 (N / e)^N = N \log_2 N - N \log_2 e$
  - $h = O(N \log_2 N)$

Sortowanie proste i „mniej proste”

- Algorytmy sortowania prostego
  - przez wstawianie
  - przez prostą zamianę (bąbelkowe)
  - przez prosty wybór
- Sortowanie Shella (tzw. metodą malejących przyrostów)

Ciąg arytmetyczny

- $S_n=a_1+a_2+\ldots a_n = n*(a_1+a_n)/2$

Sortowanie przez prosty wybór

- Przez prosty wybór
  - {3, 4, 2, 8, 1} => {1, 4, 2, 8, 3}
  - {1, 4, 2, 8, 3} => {1, 2, 4, 8, 3}
  - {1, 2, 4, 8, 3}
  - $a_1=1$
  - $a_n = n - 1$
  - $S_N=n^2/2 \in \Theta(n^2)$

Proste algorytmy sortowania

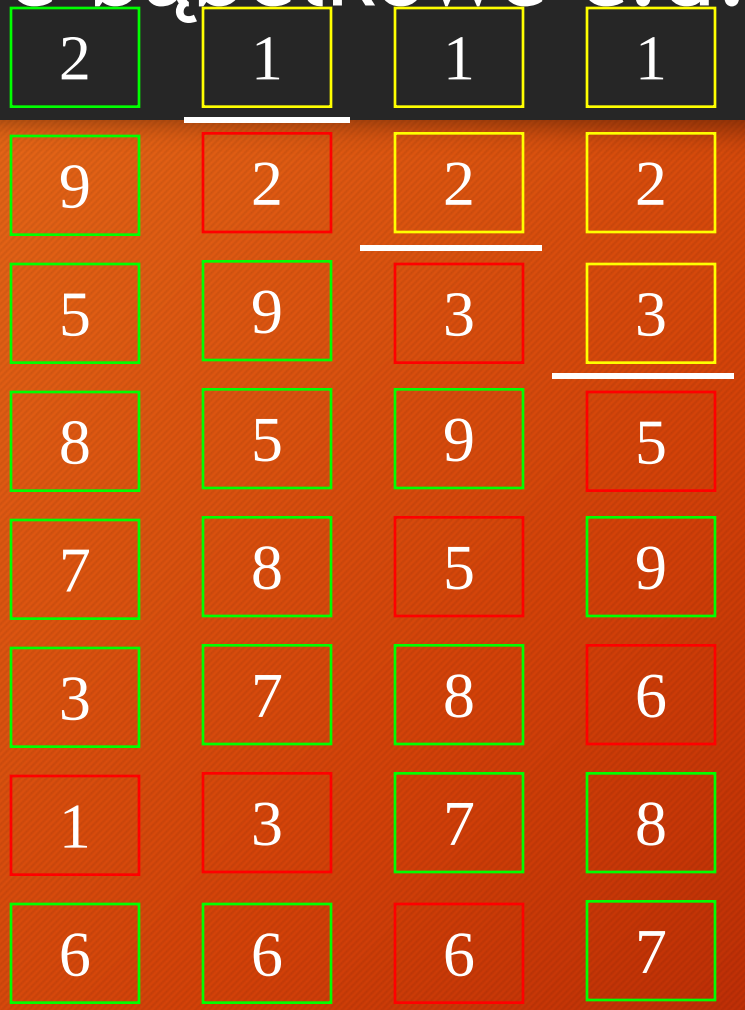
- Przez wstawianie
  - K1: {5, 3, 4, 2, 7} => {3, 5, 4, 2, 7}
  - K2: {3, 5, 4, 2, 7} => {3, ..., 5, 2, 7} => {3, 4, 5, 2, 7}
  - K3: {3, 4, 5, 2, 7} => {2, 3, 4, 5, 7}

W przypadku listy dowiązaniowej….

Sortowanie bąbelkowe

- Przez prostą zamianę (bąbelkowe) jeden przejazd przez tablicę (*fragment*)

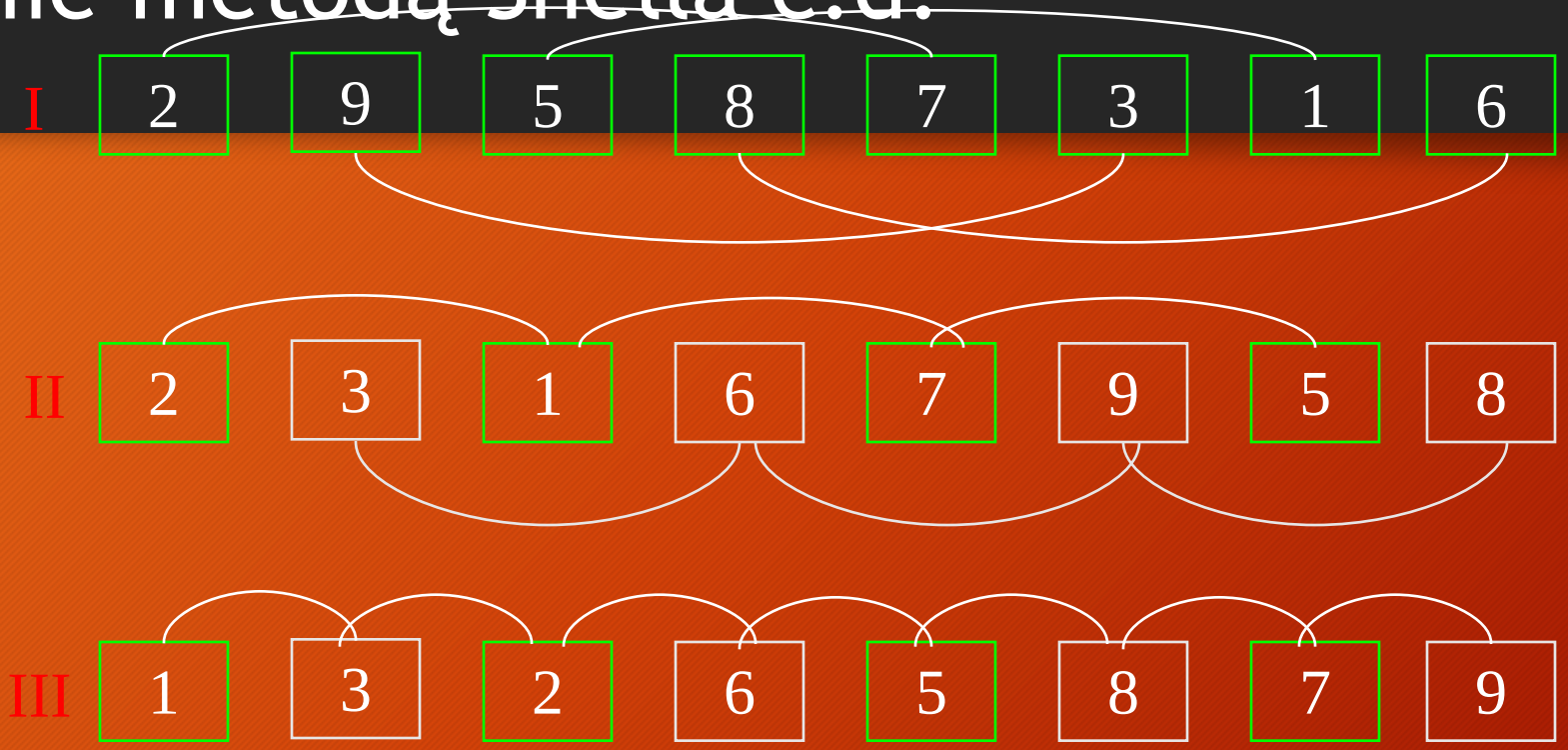
Sortowanie bąbelkowe c.d.



Sortowanie metodą Shella [stop]

- Sortowanie metodą malejących przyrostów
- Sortujemy kolejno grupy elementów oddalonych o $\{h_n, h_{n-1}, \ldots, h_0\}$, gdzie $h_0 = 1$ (koniecznie!)
- $\{h_k\}$ – malejący ciąg przyrostów
- Dobór ciągu przyrostów dowolny zakończony jedynką, ale są podobno lepsze i gorsze
- Np.
  - $\{2^{k-1}\}_{k=n,\ldots 1}$
  - Ciąg Sedgewicka
  - $$h_s = \begin{cases} 9 * 2^s - 9 * 2^{s/2} + 1, \text{ dla s parzystych} \\ 8 * 2^s - 6 * 2^{(s+1)/2} + 1, \text{ dla s nieparzystych} \end{cases}$$
- W przykładzie przyjęto ciąg przyrostów {4, 2, 1},

Sortowanie metodą Shella c.d.

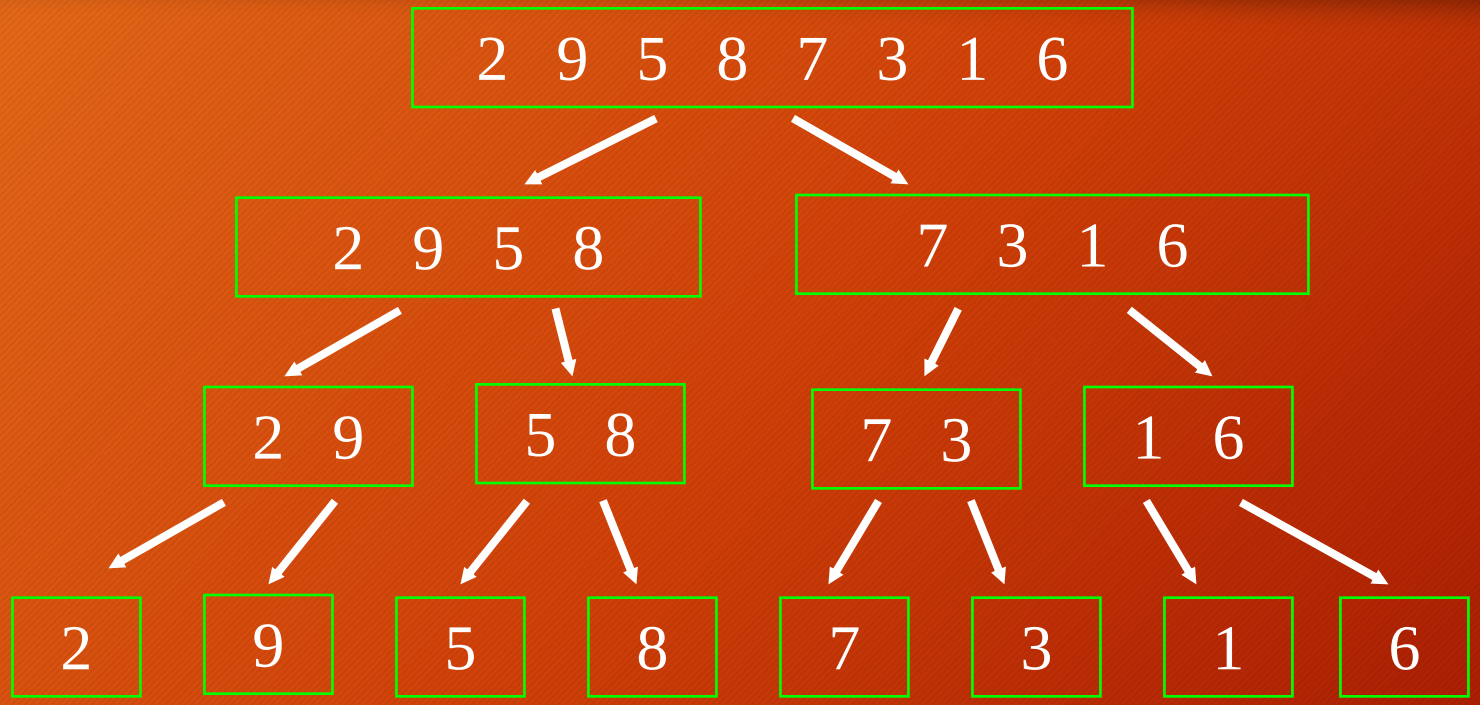


- Czas działania trudny do oszacowania
  - proporcjonalny do N $(\log_2(N))^2$ dla pewnych $\{h_k\}$
  - proporcjonalny do $N^{4/3}$ dla ciągu Sedgewicka

Sortowanie przez scalanie

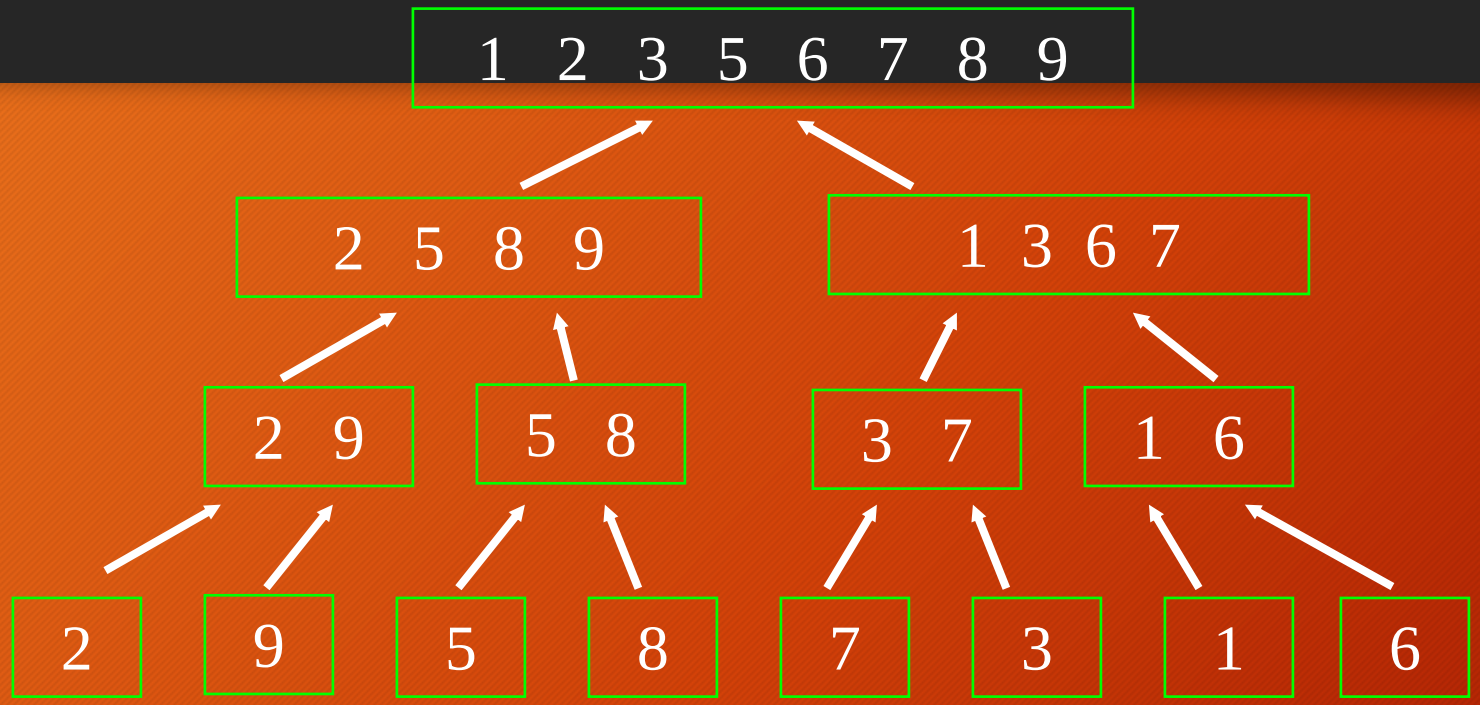
(ang. merge-sort)

Sortowanie przez podział i scalanie (1)



- Rekurencyjny podział

Sortowanie przez podział i scalanie (2)

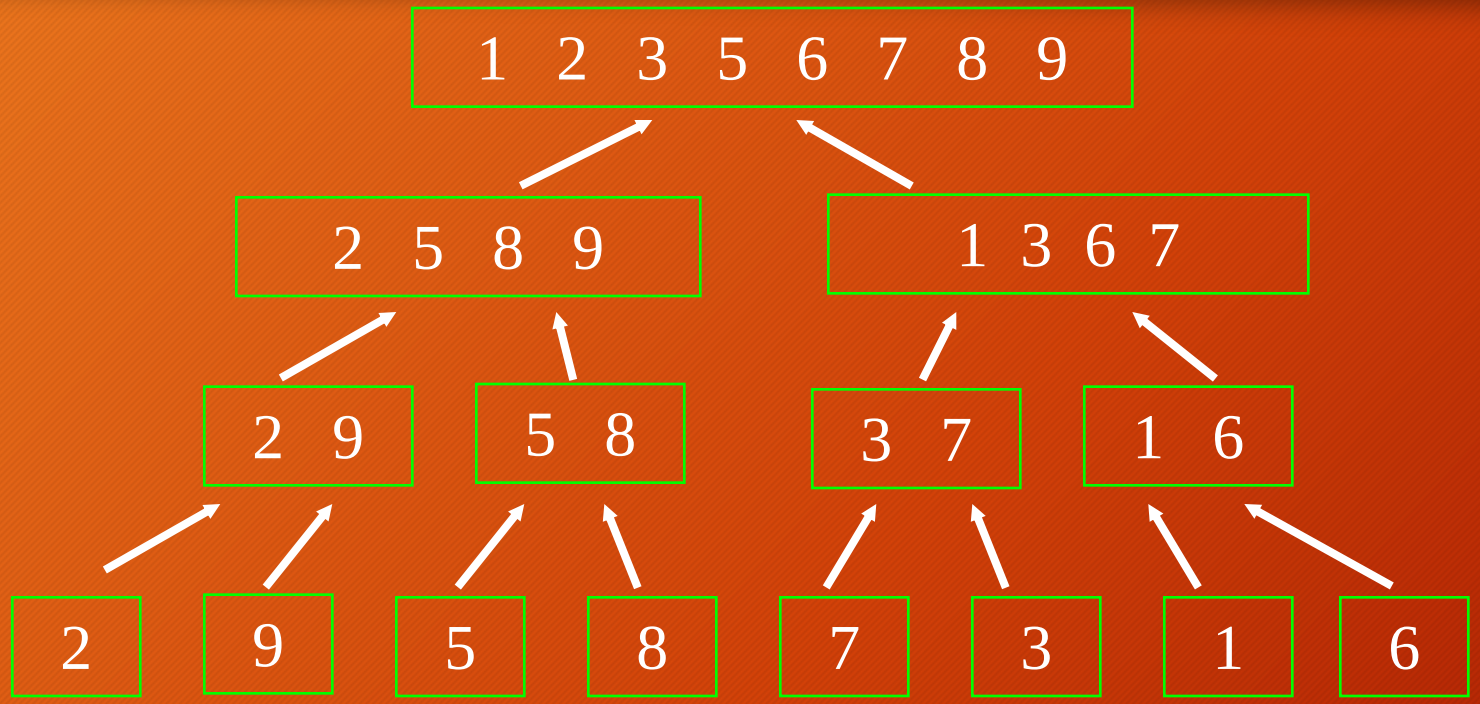


- Rozwiązanie rekurencji – scalanie
- Jaka jest wada tej procedury – gdzie źródło poprawy efektywności?

Zapis algorytmu

1. *m-sort*(*i*, *j*)
    1. q=⌊(i+j)/2⌋
    2. *m-sort*(i, q)
    3. *m-sort*(q+1, j)
    4. *merge*(*i*, *j*, *q*)

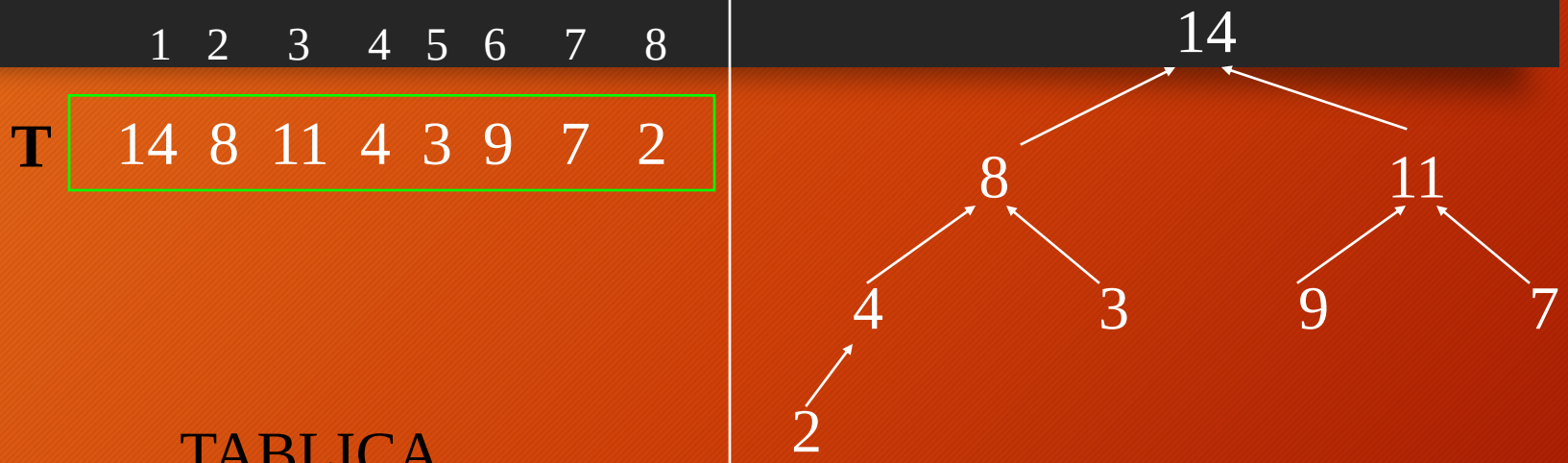
Złożoność obl. sortowania przez scalanie



- wysokość drzewa: $\lfloor \log_2(N) \rfloor$, gdzie N – liczba sortowanych elementów
- liczba porównań przy scalaniu na każdym poziomie: $\sim N$
- razem czas sortowania $\sim N \log_2(N)$

Sortowania przez przesiewanie przez kopiec

Tablica jako kopiec

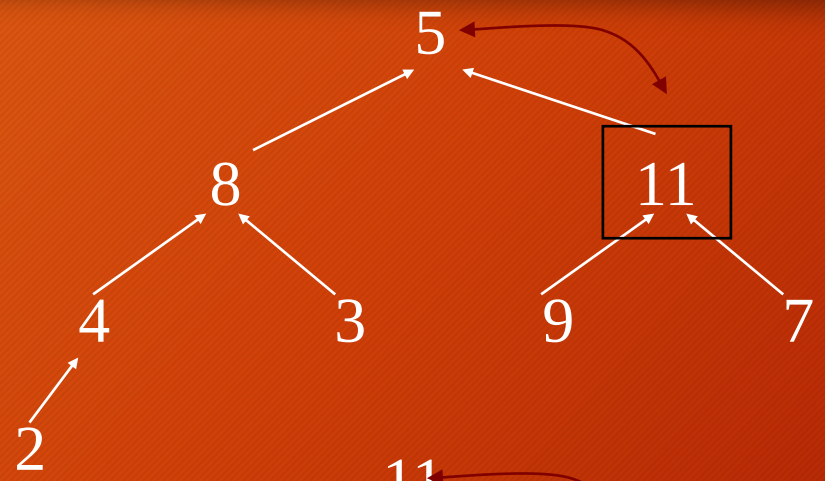


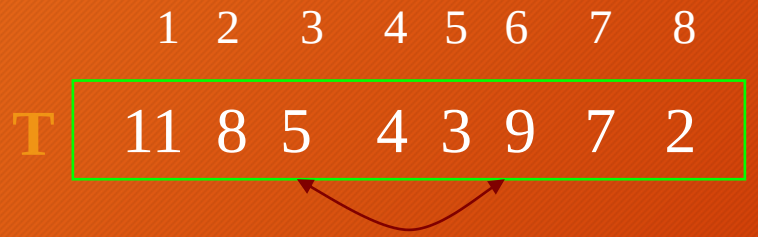
- Kopiec: tablica o indeksach ze zbioru $I=\{1, 2, \ldots, N\}$
- dla danego $i \in I$: mamy
  - $left(i) = T[2 * i]$
  - $right(i) = T[2 * i + 1]$
  - $parent(i) = \lfloor i/2 \rfloor$ dla $i > 1$
- własność kopca: dla każdego $i \in I$, $i>1$ $T[parent(i)] \geq T[i]$
- wniosek: największy element jest w korzeniu!

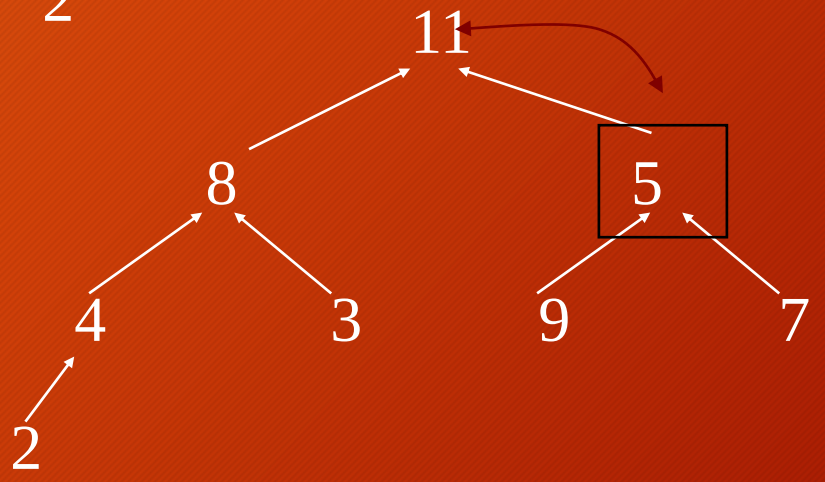
Przywracanie własności kopca

- Warunki wejściowe, dana *N*-elementowa tablica *T*:
  - *left*(*i*) i *right*(*i*) są wierzchołkami kopców,
  - $T[i] \leq T[left(i)]$ lub $T[i] \leq T[right(i)]$ (naruszenie własności kopca)
- Warunek wyjściowy
  - *i* jest wierzchołkiem kopca zawartego w tablicy *T*

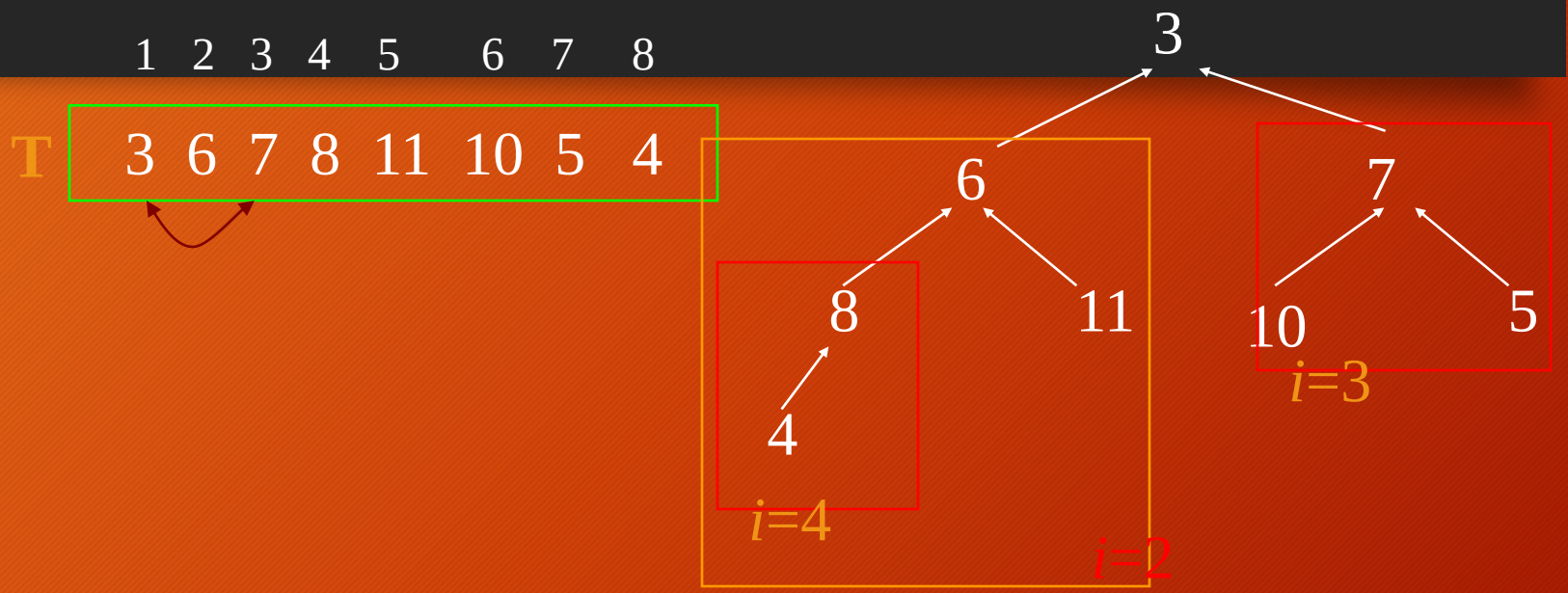
Przywracanie własności kopca przykład







Budowanie kopca w tablicy



- MAKEHEAP(*N*)
  - FOR *i* = ⌊*N* / 2⌋ downto 1 *HEAPIFY*[*i*]

Heapsort (sort. przez kopcowanie)

1. Budujemy kopiec w tablicy *T*
2. Pierwszy (największy element kopca) zamieniamy z ostatnim.
3. Skracamy kopiec o 1
4. Przywracamy własność kopca począwszy od wierzchołka (1)

- FORMALNIE:
- HEAPSORT(N)
  - MAKEHEAP(N)
  - FOR *i*=*N* downto 2
    - T[i] $\leftrightarrow$ T[1]
    - HEAPIFY(1, *i* - 1)

Czas działania proc. heapsort.

- nie gorzej niż:
  - MAKEHEAP – nie wolniej niż ~N
  - można pokazać, że każde z *N* – 1 wywołań zajmuje ~$\log_2(N)$,
  - RAZEM: $N \log_2(N)$

Sortowanie szybkie Hoare’a = sortowanie przez zamianę

Zasada konstrukcyjna

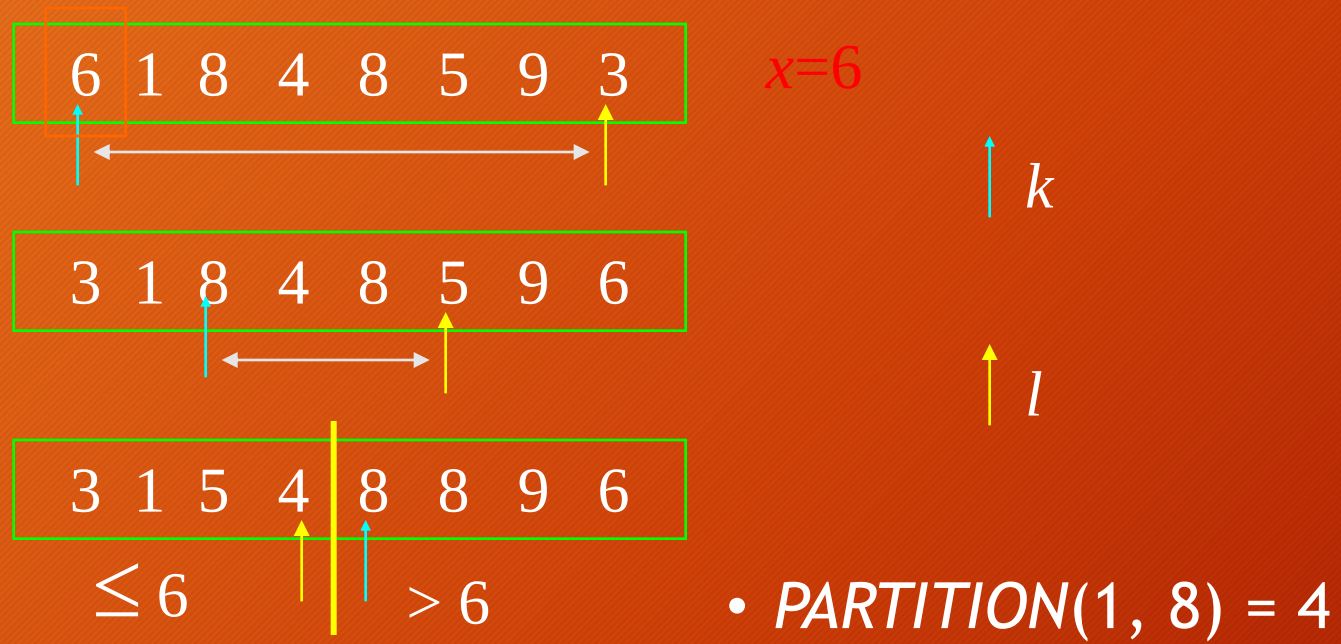
- Dana jest N-elementowa tablica T o indeksach $i \in I = \{i, i+1, \ldots, j-1, j\}$
- Podziel tablicę na dwie części, tzn. wykonaj takie zamiany elementów w tablicy i znajdź taki element q, by uzyskać tablicę o następujących własnościach:
  - dla każdego ($k = i,\ldots q$, $l = q + 1, \ldots, j$) $T[k] < T[l]$
- posortuj podtablice $T[i, q]$ oraz $T[q+1, j]$
  - przez „identyczną” zamianę i podział podtablicy

- 2 1 5 4 9 6 8 7
- 1 2 4 5 6 7 8 9

Procedura podziału tablicy

1. Wybierz element rozdzielający x (np. 1-szy, czyli $T[i]$)
2. Znajdź element $T[l] \leq x$ szukając od prawej do lewej
3. znajdź element $T[k] > x$ szukając od lewej do prawej
4. $T[l] \leftrightarrow T[k]$ – zamień elementy, jeśli $l < k$
5. *kontynuuj wg p. 3 posuwając się w lewo* (l) *i w prawo* (k)

Działanie funkcji podziału



- *PARTITION*(1, 8) = 4

- 8 3 1 4 7 5 9 6
- >=6
- __ __ __ __ do zbadania el. Rozdz.
  - <6 >=6

- 3|8 14796
- 31|8 47596
- 314|87 596
- 3145|67 98

Procedura QUICKSORT

DANA $T[i, i+1, \ldots, j-1, j]$

1. QUICKSORT(i, j)
2. IF $i < j$ THEN
    1. q = *PARTITION*(i, j)
    2. QUICKSORT(i, q)
    3. QUICKSORT($q + 1$, j)

Wersja randomizowana

- w procedurze podziału losujemy element rozdzialający spośród elementów z podtablicy $T[p, q]$
- pozwala „pokonać” posortowane fragmenty tablic

Algorytmy sortowania w czasie liniowym

Sortowanie przez zliczanie

- Założenie: tablica $T[1...N]$ zawiera liczby naturalne z przedziału $[1,...,M]$
  - relacja miedzy M i N dowolna ($M<N$, $M>N$)
- Algorytm:
  - Budujemy tablicę pomocniczą: *Count*[*M*] i inicjujemy zerami
  - **for** *i*=1 to *N* do *Count*[*T*[*i*]] = *Count*[*T*[*i*]]+1 (zliczamy)
  - *i*=1;
  - **for** *j*=1 to *M* do
    - **while** (*Count*[*j*] > 0) do
      - *Count*[*j*] = *Count*[*j*] – 1; *T*[*i*] = j; *i* = *i* +1

Sortowanie przez zliczanie c.d.

- sortuje tablicę *T* w czasie $\Theta(N + M)$ *(liczba elementów w tablicy + liczba możliwych wartości)*

Sortowanie pozycyjne (ang. radix-sort)

- Dane wejściowe
  - dana jest tablica $T[1...N][1...d]$ zawierająca cyfry, które w danym wierszu interpretujemy jako liczbę d-cyfrową
  - najmniej znaczącą cyfrą w i-tym wierszu jest $T[i][d]$, najbardziej – $T[i][1]$
- *Procedura sortowania pozycyjnego*
  - **for** $i = d$ to 1
    - *posortuj stabilnie wiersze tablicy T względem i-tej kolumny*
- Sortowanie jest stabilne, jeśli liczby o tych samych wartościach w tablicy wynikowej znajdą się w tej samej kolejności, co przed sortowaniem

- 1 2 3 5
- 2 3 2 5
- 2 2 3 4

(4) (3) (2) (1)

- 1 2 3 5
- 2 2 3 4
- 2 3 2 5

Sortowanie pozycyjne c.d.

- Czas działania zależy od zastosowanego algorytmu sortowania stabilnego
  - zysk daje zastosowanie sortowania przez zliczanie: jeśli liczby należą do przedziału od [1...M], to czas jest ~ d*(N+M)

Sortowanie kubełkowe

- Dane wejściowe:
  - dana tablica $T[1...N]$, $T[i] \in <0, 1)$
- Algorytm
  - B[1...N] – tablica N list
  - zdefiniowano funkcję *insert*($B[i]$, v), v - wartość
  - Algorytm
    - **for** i=1 to N *Insert*($\lfloor N * T[i] \rfloor$, $T[i]$)
    - **for** i=1 to N *posortuj* $B[i]$
    - wstaw elementy z list $B[1...M]$ do tablicy wynikowej
- Oczekiwana wartość długości list wynosi 1 (dla jednostajnego rozkładu danych), złożoność obliczeniowa sortowania $\sim N$

Statystyki pozycyjne

- Sortowanie
- Bez sortowania

Koniec!

Cz. 3

Politechnika Warszawska

Algorytmy i struktury danych

Odc. 4. Drzewa

Dr hab. inż. **Andrzej Zalewski**

Wydział Elektroniki i Technik Informacyjnych

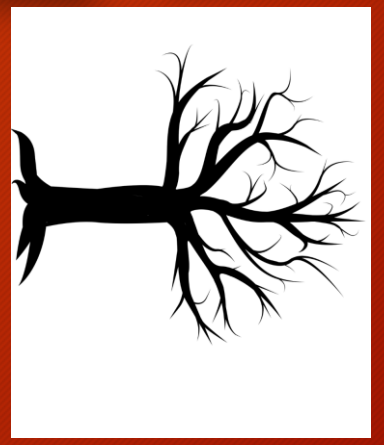
POLITECHNIKA WARSZAWSKA

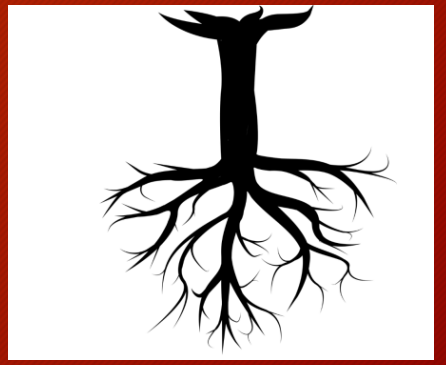
Plan

- Drzewa binarne
  - ogólnie
  - BST
  - AVL
  - SPLAY
- B-drzewa

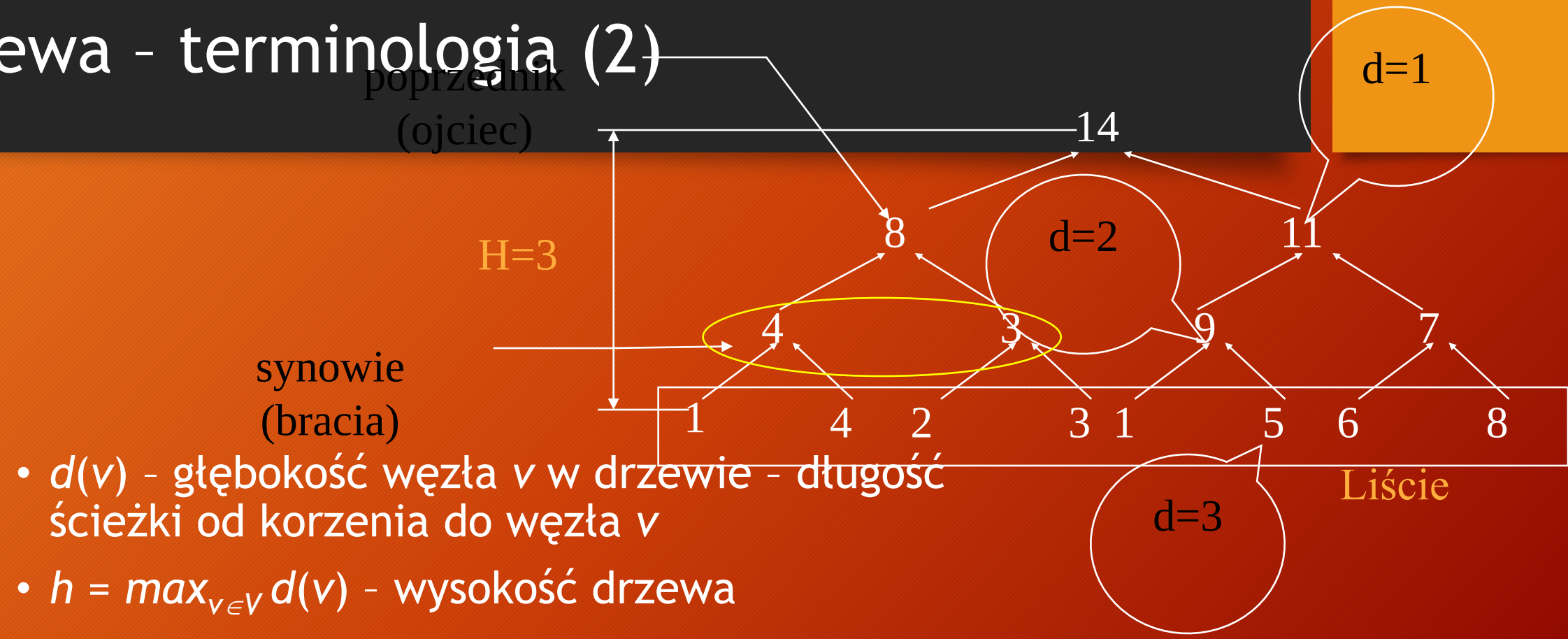
Drzewa – terminologia (1)

- Drzewo wolne:
  - spójny (z dowolnego węzła można przejść po krawędziach do dowolnego innego), acykliczny (jasne) graf nieskierowany
- Las:
  - graf acykliczny
- Drzewo ukorzenione
  - drzewo wolne, z wyróżnionym jednym wierzchołkiem zwanym korzeniem
- Drzewo uporządkowane – gdy następniki poszczególnych węzłów są uporządkowane (wyróżniony jest 1-szy, 2-gi, 3-ci itd. następnik)





Drzewa – terminologia (2)



- $d(v)$ – głębokość węzła v w drzewie – długość ścieżki od korzenia do węzła v
- $h = max_{v \in V} d(v)$ – wysokość drzewa

Drzewa – terminologia (3)

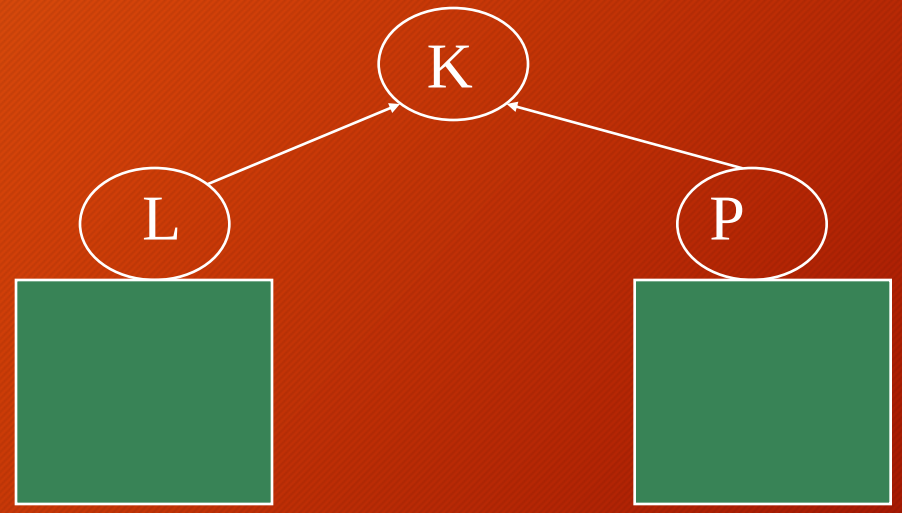
- Drzewo rzędu *k* – drzewo ukorzenione, w którym węzeł ma co najwyżej *k* następników.
  - pytanie: *przykład sytuacji praktycznej, kiedy rząd drzewa nie może być a priori ograniczony*?
- Drzewo pełne – drzewo ukorzenione rzędu *k*, w którym wszystkie liście mają tę samą głębokość.
- Drzewo binarne – drzewo rzędu 2.

Właściwości drzew

- Drzewo rzędu *k* o wysokości *h*
  - *maksymalna liczba liści*: k^h
  - *maksymalna liczba węzłów drzewa*:
    - $k^0 + k^1 + \ldots + k^h = (k^{h+1} - 1) \ / \ (k - 1)$
    - dla drzewa binarnego: $(2^{h+1} - 1)$
- Drzewo pełne
  - drzewo pełne rzędu *k* o $n = k^h$ liściach
    - wysokość: $h = \log_k n$ *(bo* $n = k^h$*)*

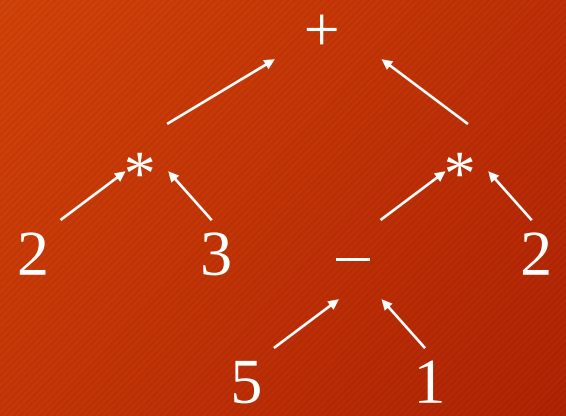
Przechodzenie drzew binarnych

- *preorder*:
  - Korzeń
  - Lewe
  - Prawe
- *inorder*:
  - Lewe
  - Korzeń
  - Prawe
- *postorder*:
  - Lewe
  - Prawe
  - Korzeń



Porządki przechodzenia a notacje wyrażeń

- y = 2 * 3 + (5 – 1) * 2
- *preorder*:
  + * 2 3 * – 5 1 2
- *inorder*:
  2 * 3 + (5 – 1) * 2
- *postorder*:
  2 3 * 5 1 – 2 * +



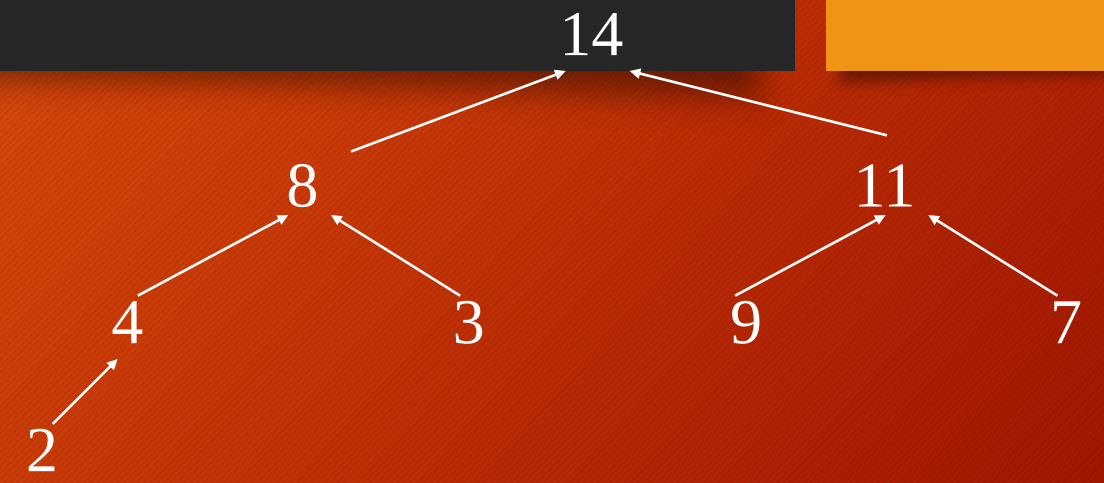
preorder i *postorder* – tzw. notacja polska przed- i przyrostkowa

Reprezentacje drzew binarnych

- Jako struktura z dowiązaniami
- Reprezentacja tablicowa (kopcowa)

Tablica jako drzewo binarne (tzw. kopiec)

	1	2	3	4	5	6	7	8
T	14	8	11	4	3	9	7	2



TABLICA

DRZEWO

- Kopiec: tablica o indeksach ze zbioru $I=\{1, 2, \ldots, N\}$
- dla węzła o indeksie $i \in I$: mamy
  - $left(i) = T[2 * i]$
  - $right(i) = T[2 * i + 1]$
  - $parent(i) = \lfloor i/2 \rfloor$ dla $i > 1$
- własność kopca: dla każdego $i \in I$, $i>1$ $T[parent(i)] \geq T[i]$

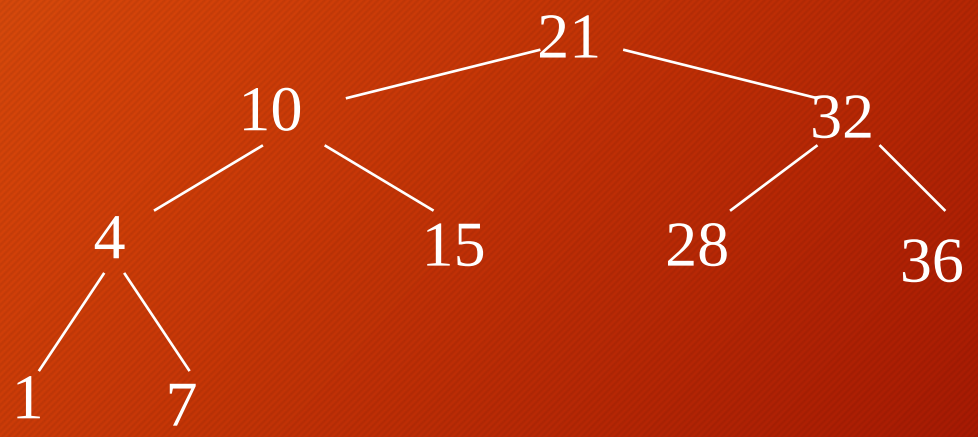
Drzewa wyszukiwania binarnego (BST)

BST – ang. Binary Search Tree

Drzewo poszukiwań binarnych [stop]

- Drzewo binarne, węzeł standardowy (klucz, dane dodatkowe) + wskaźnik lewego, prawego
- $key(left(x)) \leq key(right(x))$ /*x* pewien węzeł/

- przykład:



Przejście

- Przejście w porządku *inorder* (zgodny z porządkiem kluczy):

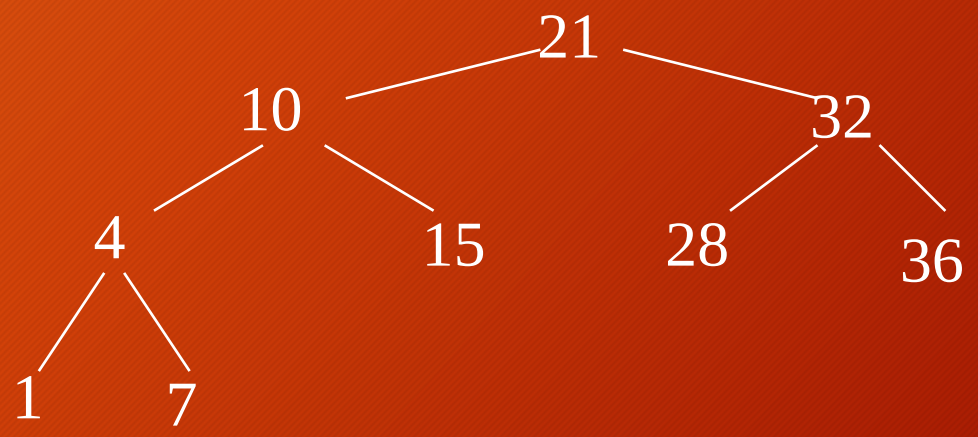
```
print(N)
    if N<>NIL
        print(left(N))
        print-key(N)
        print(right(N))
```

Wyszukiwanie

```
Search(N, k)
    if N=NIL or key(N)=k return N
    if k < key(N)
            search(left(N), k)
    else
            search(right(N), k)
```

Min, max

- znalezienie odpowiednio najbardziej „lewego” i najbardziej „prawego” węzła drzewa

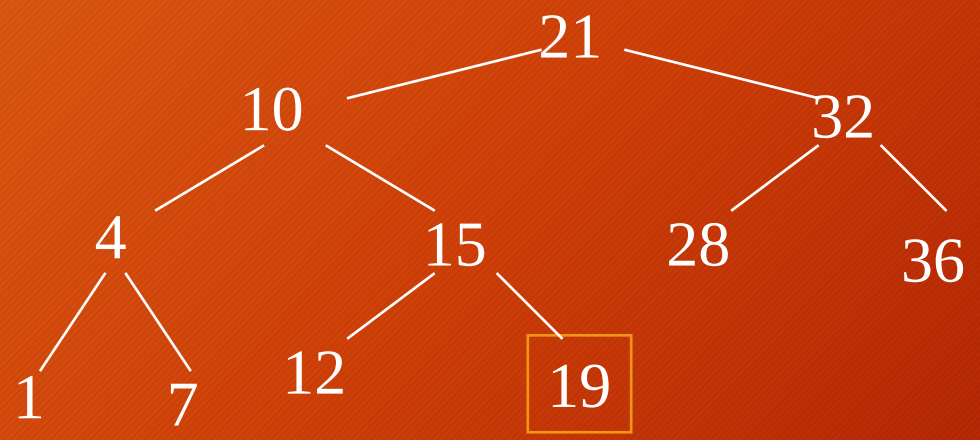


Następnik (poprzednik)

- Jeśli right(N) <> NIL, to
  - następnik jest równy min(right(N))
- w przeciwnym razie
  - następnik *N* jest najniższym przodkiem *N*, którego lewy następnik jest także przodkiem *N [np. następnik(19)]*

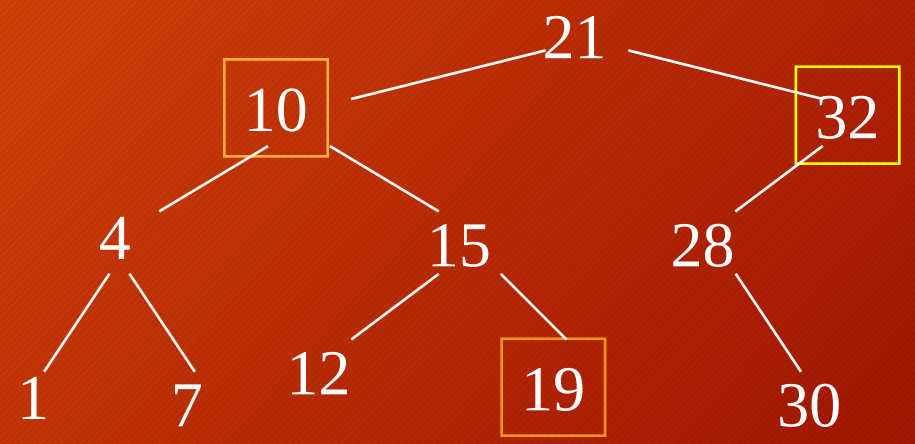
Wstawianie w BST

- Znajdujemy miejsce na nowy liść zgodnie z porządkiem wyznaczonym przez relację ojca i dzieci



Usuwanie węzła o zadanym kluczu

- Trzy przypadki
  - Liść – po prostu usuwamy (1, 7, …)
  - Węzeł z jednym synem – łączymy syna z ojcem (32)
  - Węzeł wewnętrzny (10)
    - znajdujemy następnik
    - usuwamy następnik zapamiętując jego wartość w węźle z usuwanym kluczem



Zdefiniowanie problemu równoważenia

- Operacje na drzewach BST są realizowane w czasie nie gorszym niż ~ wysokości drzewa
- Drzewa BST mogą rosnąć nierównomiernie lub „wyrodnieć w wyniku wstawiania/usuwania elementów”
- Wysokość „zwyrodniałego” drzewa będzie większa aniżeli to konieczne z punktu widzenia liczby jego węzłów
- Wprowadzić mechanizmy „kontroli” geometrii drzewa => drzewa zrównoważone (AVL, drzewa czerwono-czarne i inne)

Ogólna ch-ka drzew (z)równoważonych

- Ograniczają różnice wysokości poddrzew
  - Stąd: ograniczają czas wyszukiwania w drzewie
    - a więc także znajdowania *min*, *max*, *succ*, *pred*;
- Koszt: większa złożoność operacji wstawiania i usuwania oraz związany z nim nakład czasowy

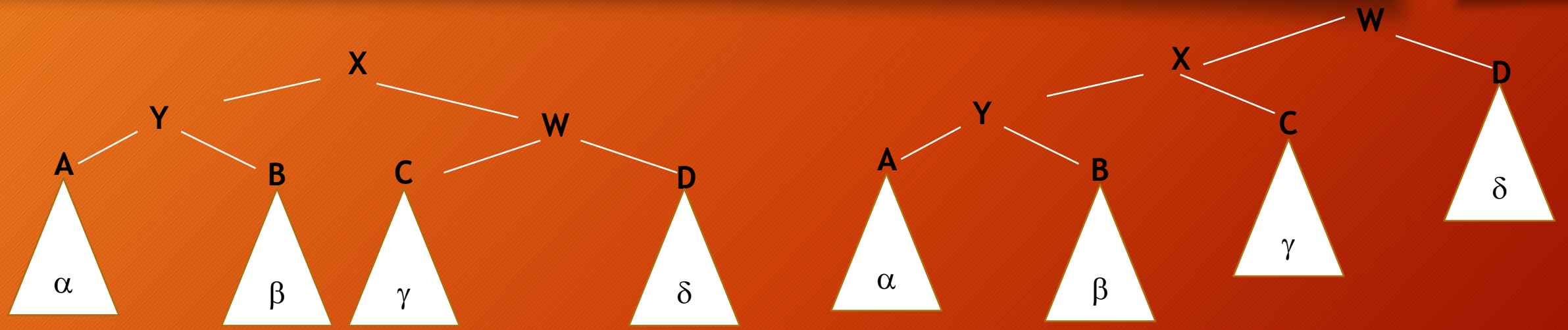
Drzewa AVL

AVL = Adelson-Wielskij – Łandis

Definicja

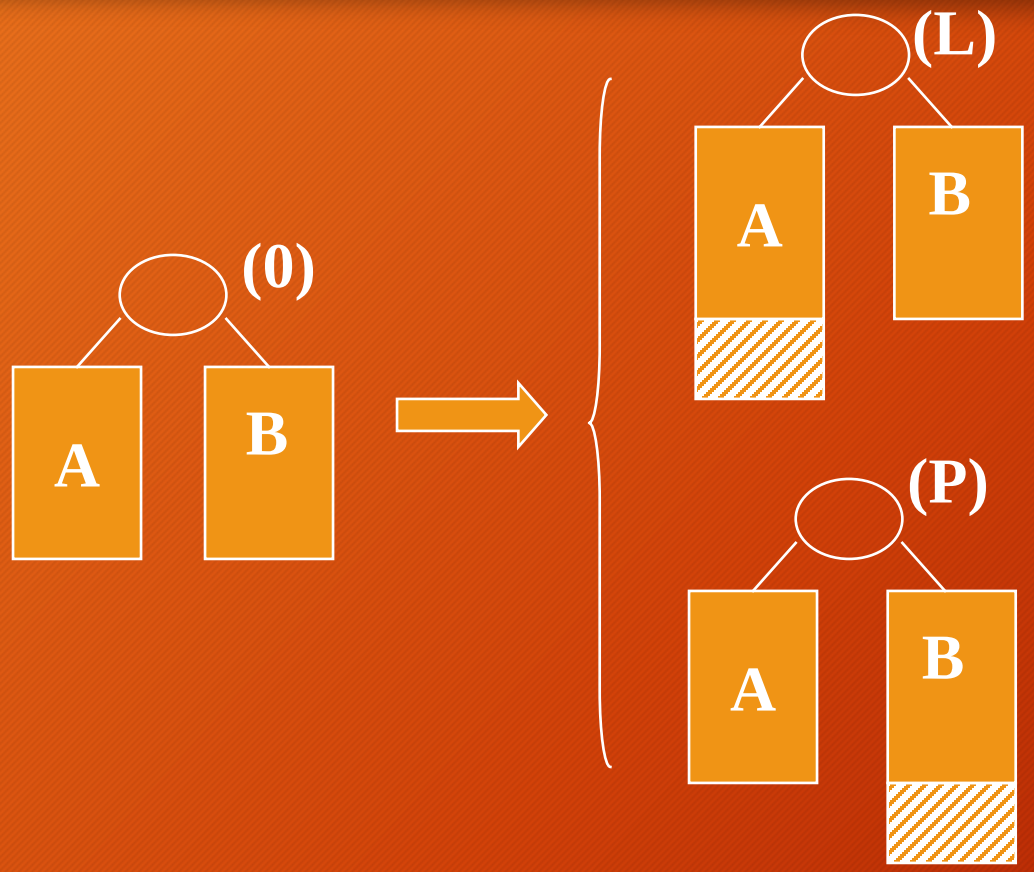
- Drzewo BST nazywamy AVL-zrównoważonym, jeśli dla dowolnego węzła *x* różnica wysokości poddrzew zaczepionych w tym węźle nie jest większa niż 1
- (stop)

Obserwcja „centralna” dla drzew bin., zrównoważonych



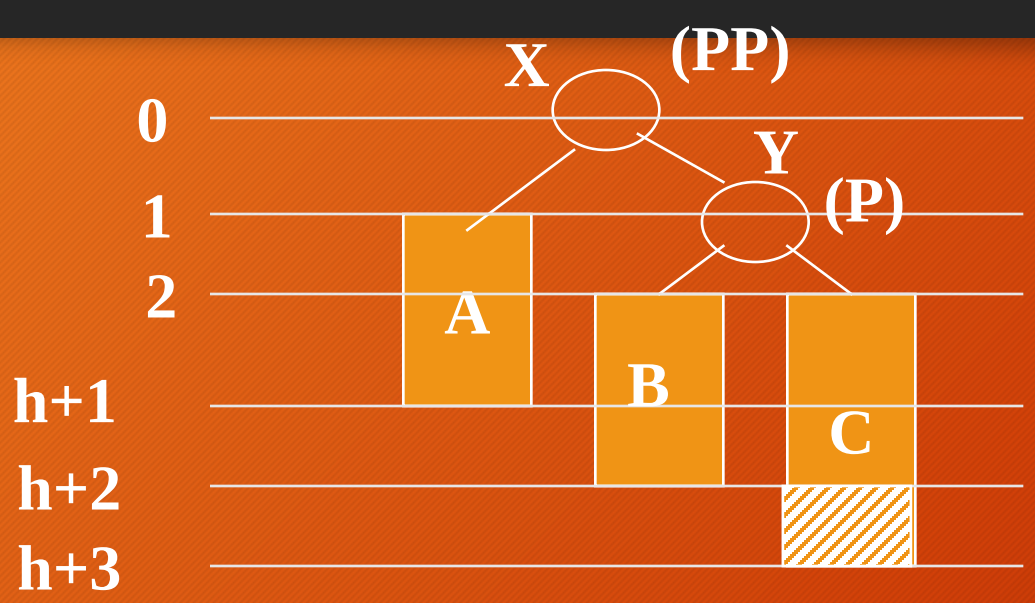
- $\alpha <_T \beta <_T \gamma <_T \delta$
- $T_1 <_T T_2$ – każdy klucz *k* w T_1 mniejszy od dowolnego klucza z T_2

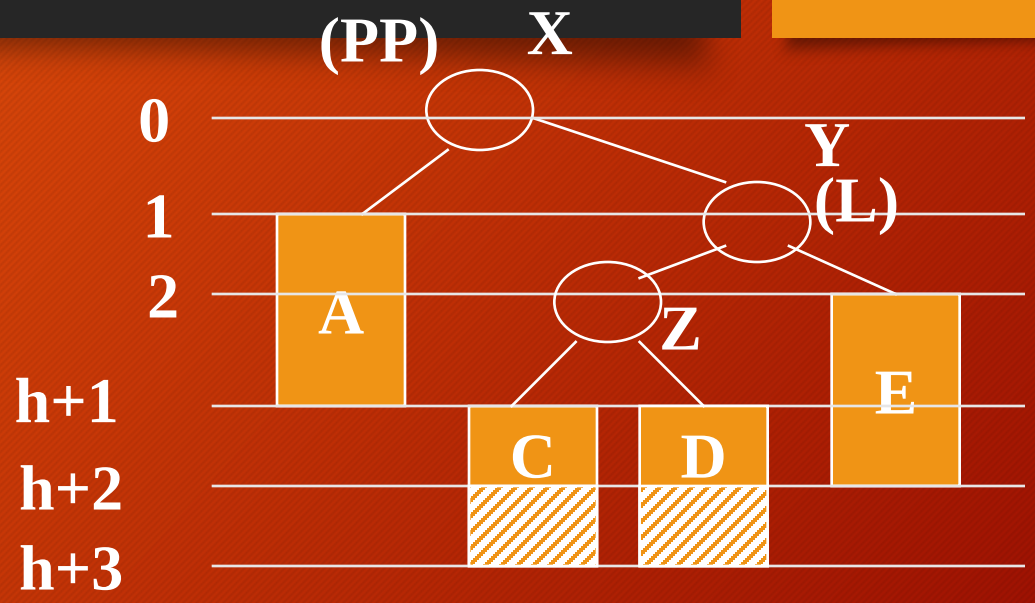
Oznaczenie stanu zrównoważenia



- Wprowadzamy wskaźnik zrównoważenia drzewa zaczynającego się węźle *N* $B(N) \in \{(0), (L), (P), (PP), (LL)\}$
- W implementacji praktycznej wygodniej jest przyjąć 0 zrównoważone,
  1 i -1 odpowiednio dla „przechyłu” w lewo lub prawo

Równoważenie (1)



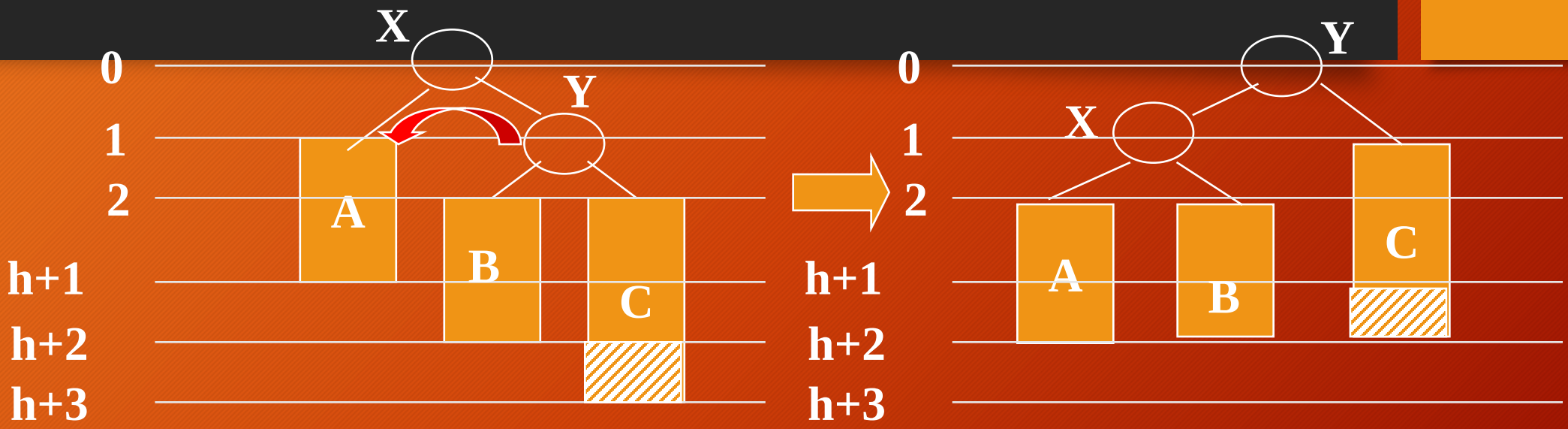


Przypadek 1

Przypadek 2 (podwójny)

- Przypadek 1’ i 2’ – symetryczne – do przećwiczenia w domu

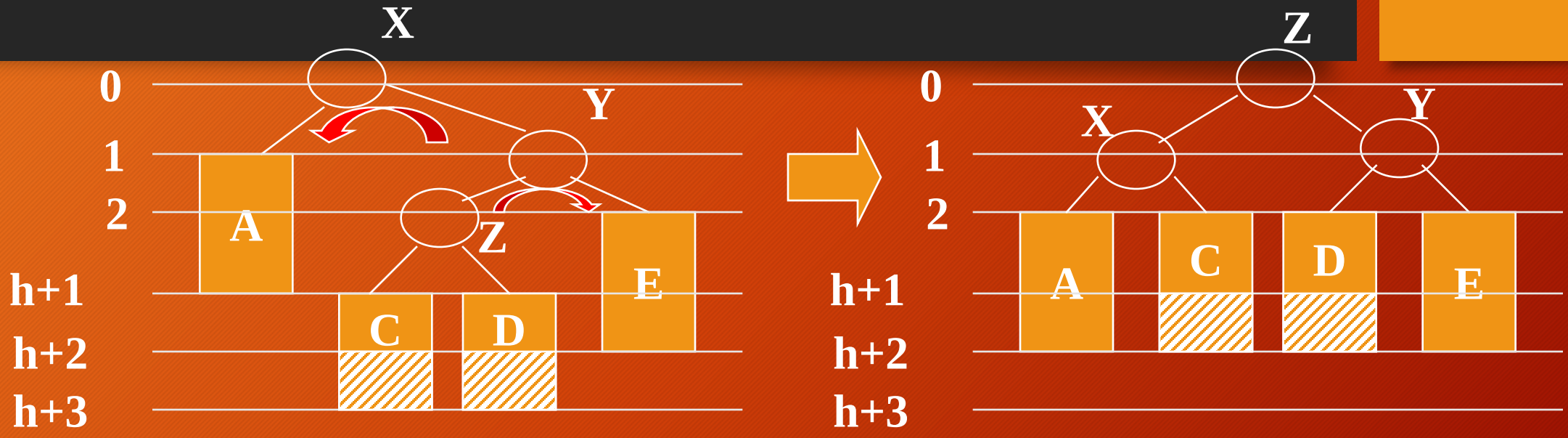
Równoważenie (2)



Przypadek 1

- Operacja tzw. lewej rotacji

Równoważenie (3)

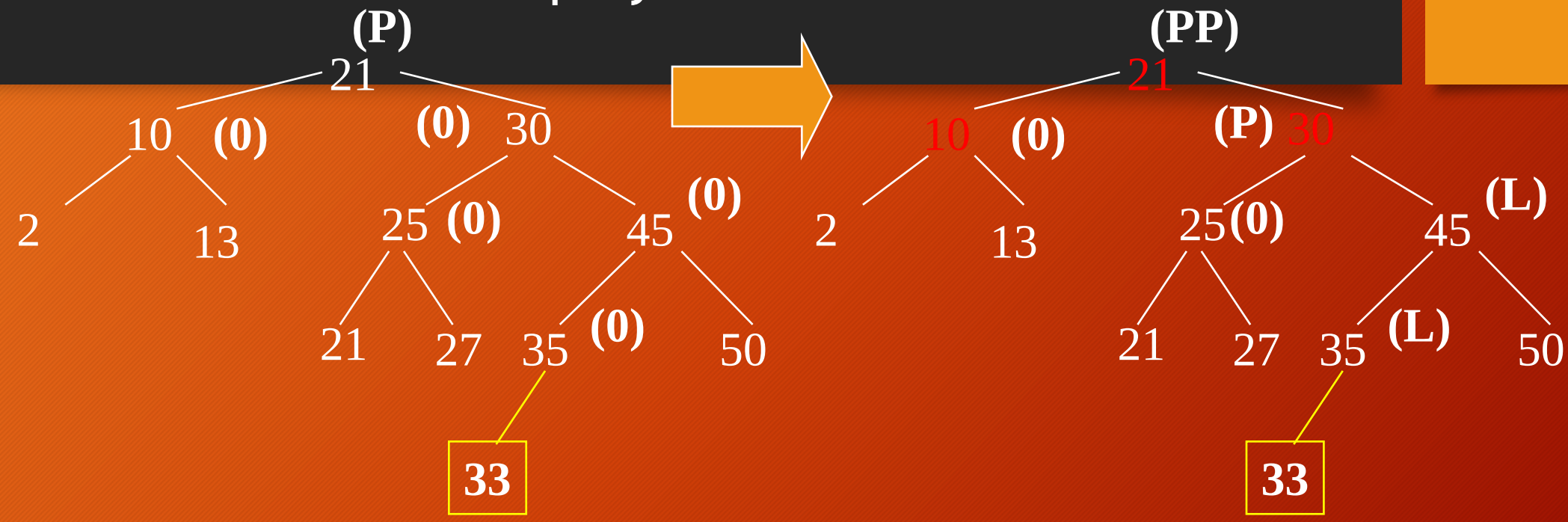


Przypadek 2

- Prawa rotacja Z-Y
- Lewa rotacja Z-X

Przypadek 2

Kontrola zrównoważenia – przykład



- Po dodaniu 33 mamy przypadek 1

Kontrola zrównoważenia

- Propagujemy w górę drzewa informację o urośnięciu poddrzewa
  - B(N)=(L) + urosło L => B(N)=(LL) (!!! zrównoważ)
  - B(N)=(P) + urosło P => B(N)=(LL) PP (!!! zrównoważ)
  - B(N)=(P) + urosło L => B(N)=(0)
  - B(N)=(L) + urosło P => B(N)= (0)
  - B(N)=(0) + urosło L => B(N)= (L)
  - B(N)=(0) + urosło (P) => B(N)= (P)
- Rozróżnienie przypadków 1 i 2:
  - na podstawie wartości wskaźnika zrównoważenia dla lewego lub prawego poddrzewa
  - różne podejścia – np. na podstawie informacji o „urośnięciu” oraz wartości wstawionego klucza

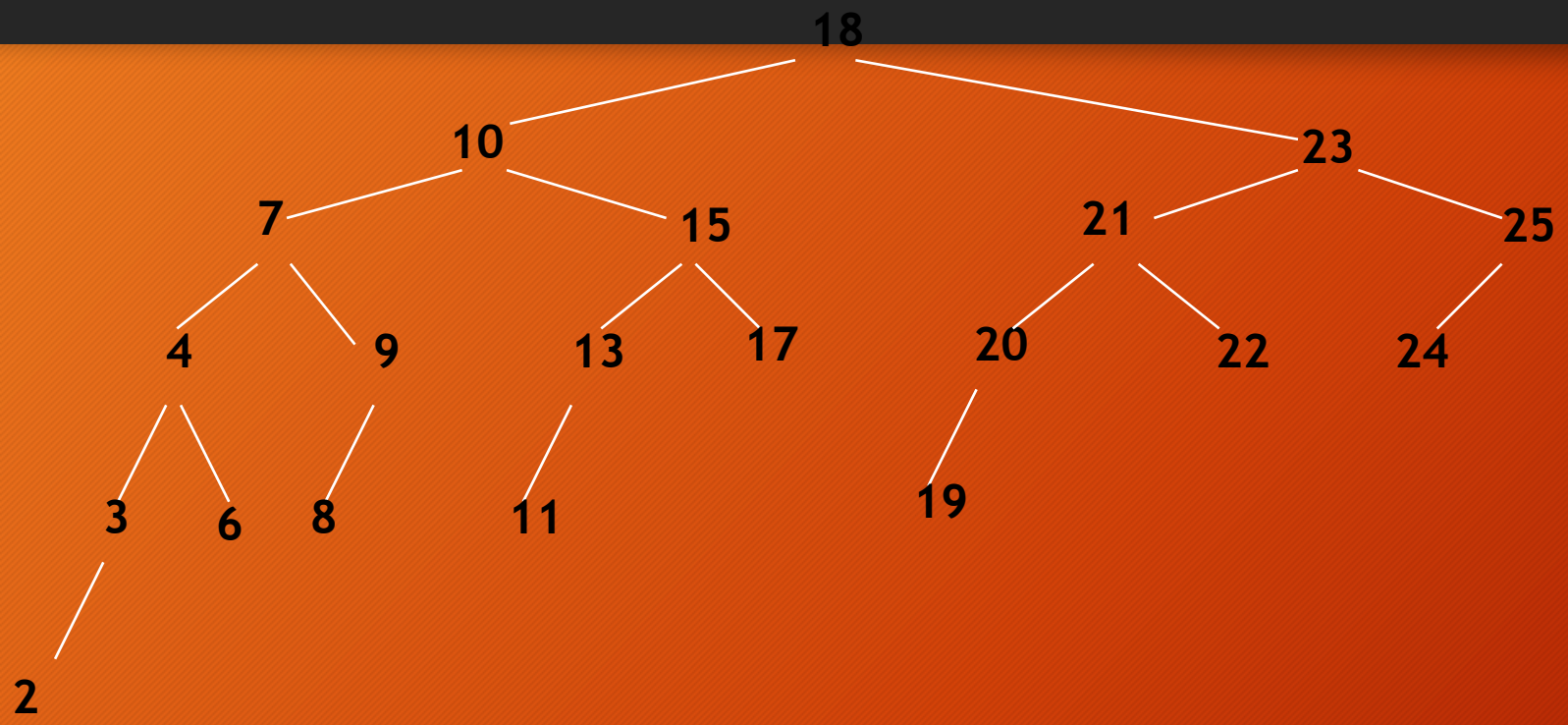
Usuwanie

- Przypadek 1: skróciło się lewe poddrzewo
  - A – prawe poddrzewo jest zrównoważone lub przechylone w prawo (działa rotacja w lewo)
  - B – prawe drzewo jest przechylone w lewo (nie działa rotacja w lewo – trzeba wyk. rotację w prawo a potem w lewo)
- Przypadek 2: skróciło się prawe poddrzewo
  - A – lewe poddrzewo jest zrównoważone lub przechylone w lewo (działa prawa rotacja)
  - B – lewe poddrzewo jest przechylone w prawo (nie działa prawa rotacja, trzeba zrobić rot. lewo-prawo)

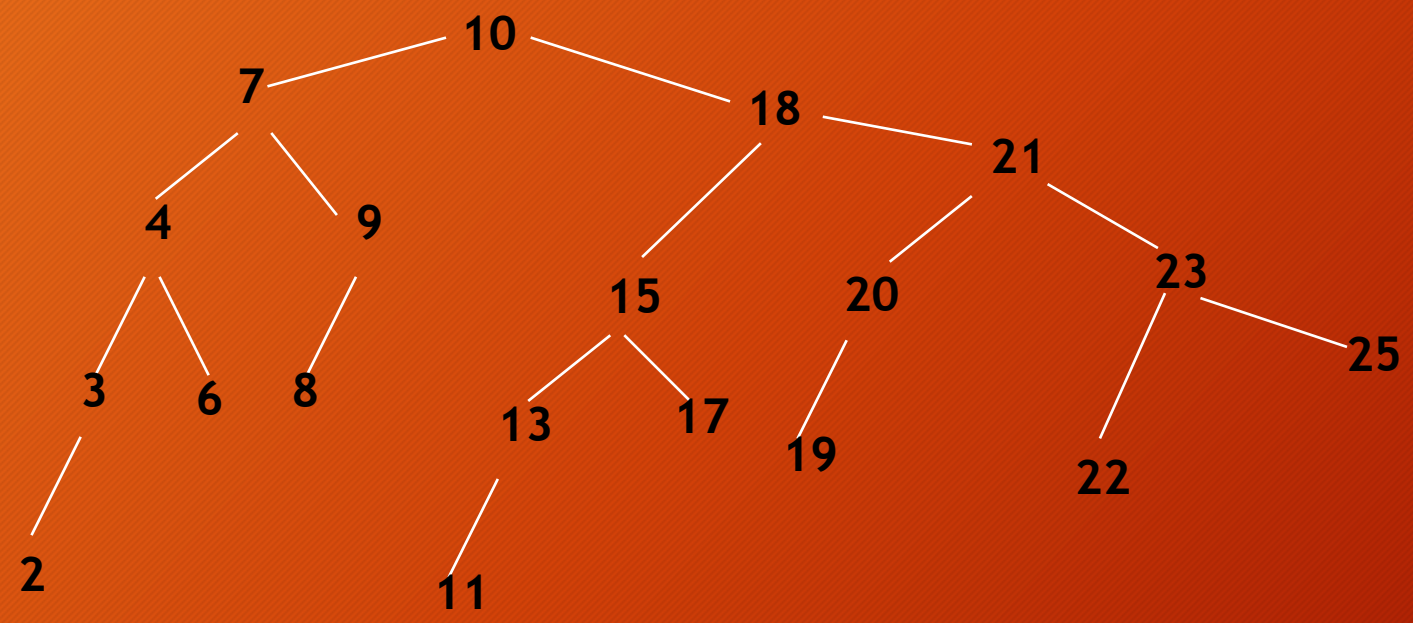
Usuwanie

- I niestety, jeśli trzeba aż do wierzchołka
- Trudny przypadek – drzewo Fibbonacciego

Drzewo Fibonacciego a drzewo AVL



Drzewo Fibonacciego a drzewo AVL



Drzewa SPLAY

Tzw. drzewa Sleatora i Tarjana

Drzewa binarne typu „SPLAY”

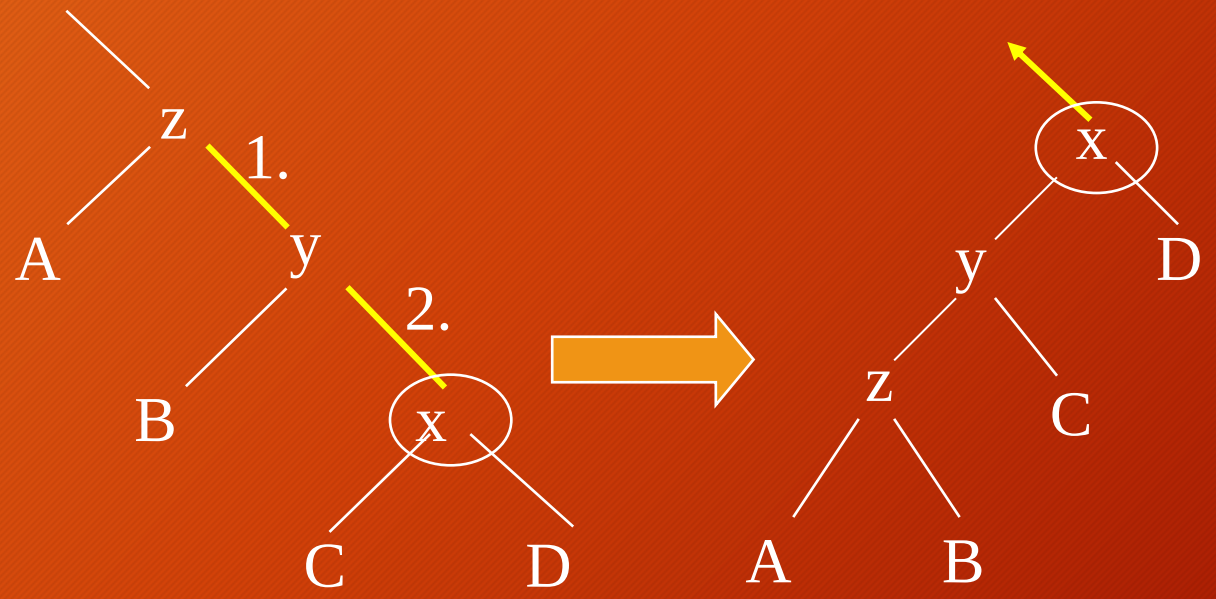
- Drzewa samoadaptujące się
  - przez „przesuwanie ku wierzchołkowi” najczęściej/ostatnio wyszukiwanych elementów
- Heurystyki
  - stopniowego przesuwania elementu wyszukanego *x* ku korzeniowi przez rotację krawędzi łączącej *x* z *parent*(*x*)
  - przesuwania elementu wyszukanego do wierzchołka drzewa przez rotacje krawędzi x z *parent*(*x*) aż do samego wierzchołka drzewa

Drzewa binarne typu „SPLAY” c.d.

- Rozwiązanie właściwe (Sleator & Tarjan, 1985)
  - udoskonalona II-ga z heurystyk
  - rozpatruje się trzy przypadki „splayingu”
- Niech x oznacza węzeł, który został wyszukany
- Przypadek 1. Parent(x) jest korzeniem drzewa.
  - wykonujemy rotację krawędzi <Parent(x), x> (lewą lub prawą w zależności od geometrii)

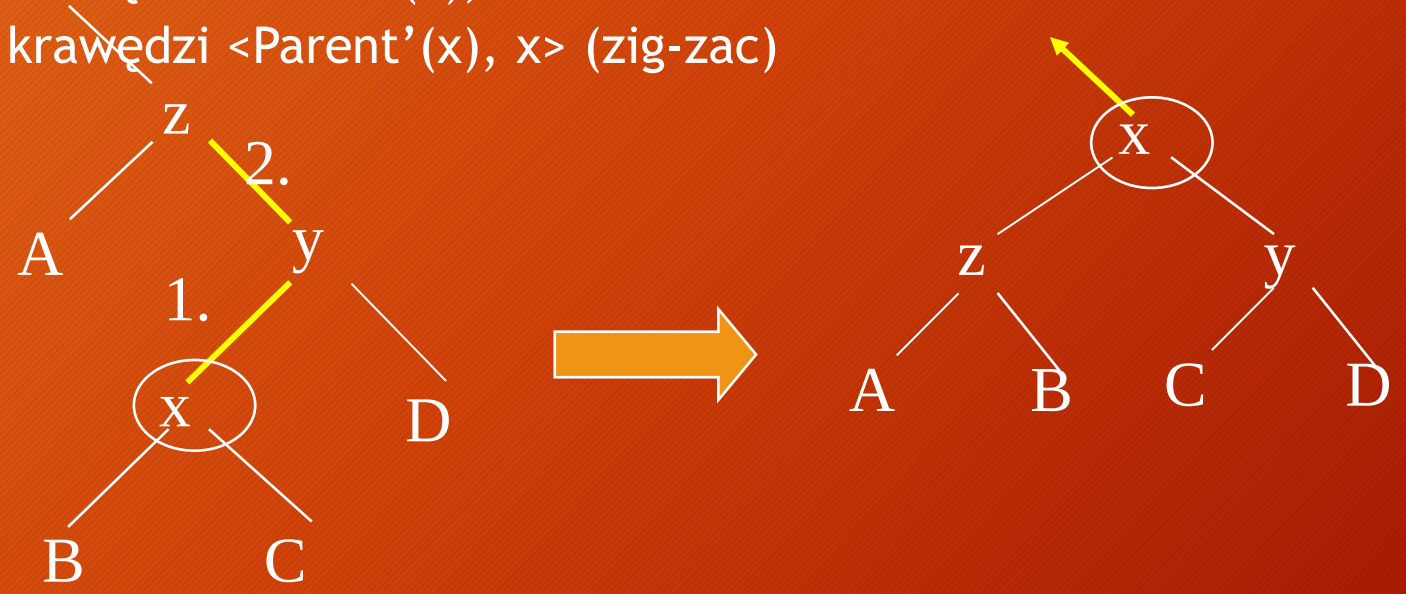
Drzewa binarne typu „SPLAY” c.d.

- Przypadek 2. x i Parent(x) są oba lewymi albo oba prawymi dziećmi i Parent(x) nie jest korzeniem (stosujemy pojed. rotacje)
  1. rotacja krawędzi <G-Parent(x), Parent(x)>
  2. rotacja krawędzi <Parent’(x), x> (zig-zig)



Drzewa binarne typu „SPLAY” c.d.

- Przypadek 3. x i Parent(x) są odpowiednio dzieckiem lewym i prawym lub vice versa
  1. rotacja krawędzi <Parent(x), x>
  2. rotacja krawędzi <Parent’(x), x> (zig-zac)

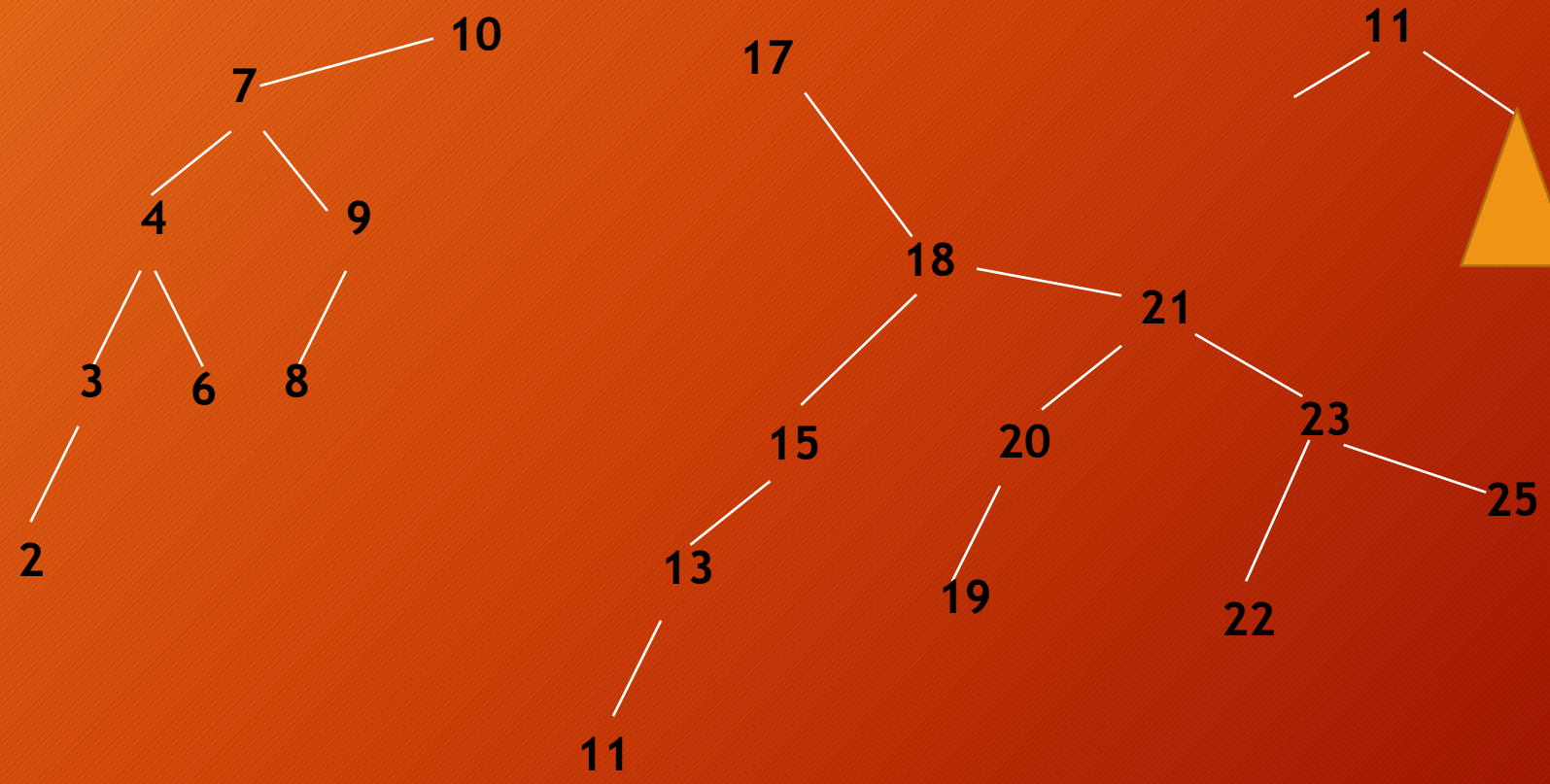


Drzewa binarne typu „SPLAY” c.d.

- „Splaying” powoduje, że znaleziony węzeł znajdzie się w wierzchołku drzewa, w lewym poddrzewie klucze mniejsze, w prawym większe (jak w drzewie BST)
- Operacje na drzewach binarnych typu „SPLAY”
  - wyszukiwanie
    - jak w drzewie BST
    - jeśli znajdziemy element – wykonujemy „splaying” począwszy od odnalezionego węzła, jeśli wyszukiwanie się nie powiodło – od ostatniego niepustego węzła
  - wstawianie klucza nie występującego w drzewie
  - usuwanie klucza występującego w drzewie

Operacje pomocnicze

- łączenie drzew t_1 i t_2 (*join*) – założenie: klucze z t_1 mniejsze od kluczy z t_2
  - wyszukujemy największy element w t_1
  - towarzyszy temu „splaying” tak, jak przy dowolnym wyszukiwaniu
  - t_1 ma puste prawe poddrzewo – można podłączyć zatem poddrzewo t_2


10
7
4
9
3
6
8
2
17
18
21
15
20
23
25
13
19
22
11
11

Operacje pomocnicze

- podział drzewa *t* na dwa drzewa t_1 i t_2 (*split*), w których elementy drzewa t_1 są mniejsze od pewnej *v*, zaś elementów drzewa t_2 tej wartości większe
  - wyszukujemy *v* w drzewie *t*
  - wykonujemy „splaying”
  - wierzchołek z lewym poddrzewem to t_1, t_2 – prawe poddrzewo

Wstawianie

- Wstawianie klucza *v*:
  - dzielimy drzewo względem klucza *v* (operacja *split*)
  - powstałe drzewo t_1 staje się lewym poddrzewem drzewa zaczepionego w korzeniu zawierającym *v*, t_2 zaś prawym poddrzewem
- Usuwanie klucza *v*:
  - wyszukujemy klucz *v* ze splayingiem – tj. powodujemy przemieszczenie go do korzenia
  - usuwamy korzeń, łączymy dwa poddrzewa

Właściwości drzew typu „splay”

- „Splaying” zmniejsza o połowę głębokość węzłów na ścieżce do korzenia
- Drzewa „splay” są:
  - dla dużej liczby wyszukiwań – jest ono równie jak w dowolnym drzewie zrównoważonym
  - tak efektywne jak optymalne drzewa wyszukiwania;
  - czas dostępu do *v* można oszacować jako log (1 + N), gdzie *N* – liczba różnych kluczy wyszukiwana od czasu ostatniego dostępu do *v (zbiór N+1 najczęściej wyszukiwanych elementów jest wyszukiwanych w czasie logarytmicznym), N<< liczby elementów w drzewie*

B-drzewa

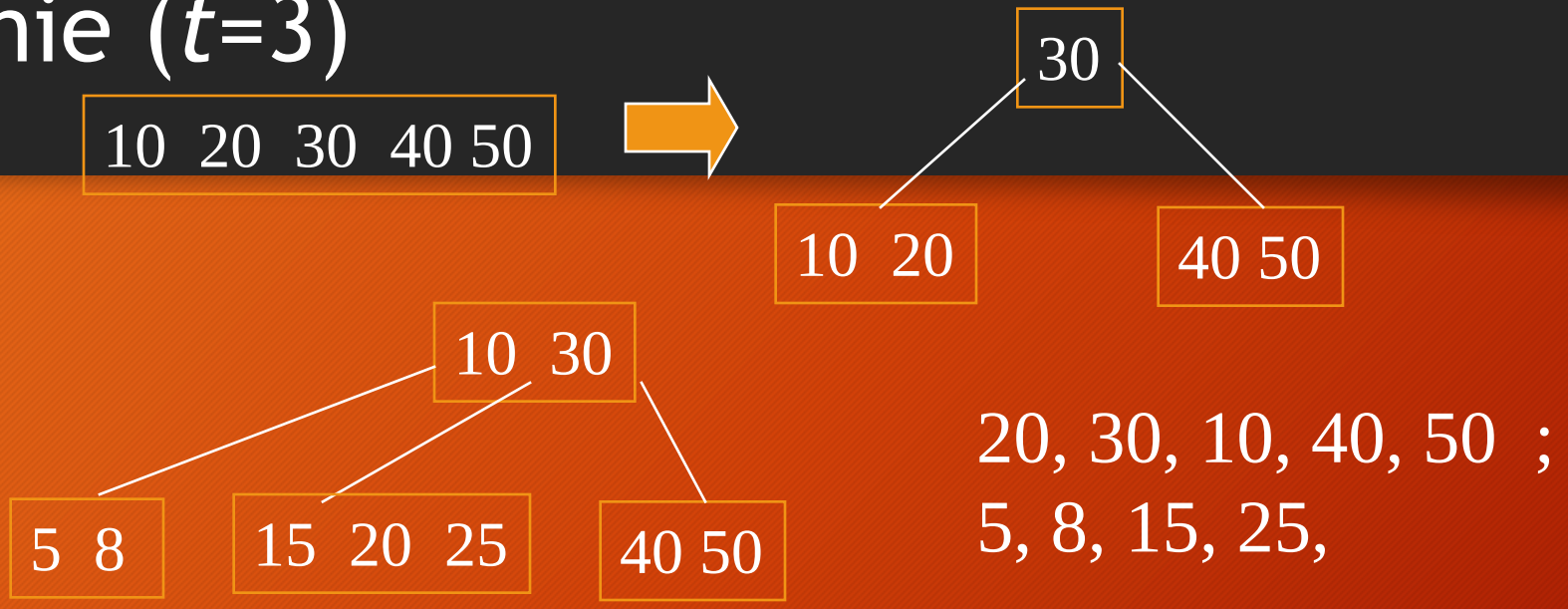
B-drzewa - definicja

- Węzeł x:
  - N[x] – liczba kluczy pamiętanych w x
  - $key_1[x],...,key_{n[x]}[x]$ – klucze w porządku niemalejącym
  - leaf[x] – pole wskazujące, czy dany węzeł jest liściem, czy węzłem wewnętrznym
  - węzły wewnętrzne zawierają n[x] + 1 wskaźników do synów (każdy węzeł zawiera n[x] + 1 następników)
  - klucze pamiętane w poddrzewach leżą w przedziałach wyznaczonych kluczami węzła
  - minimalny stopień drzewa *t* ogranicza maksymalną liczbę kluczy w danym węźle
    - węzeł różny od korzenia musi mieć co najmniej t – 1 kluczy
    - co najwyżej 2t – 1 kluczy

Przykład B-drzewa

- Wstawianie / usuwanie:
  - strategia prewencyjna: uniemożliwiamy zaburzenie struktury przy wstawianiu / usuwaniu
  - strategia reaktywna: gdy wystąpi zaburzenie, to korygujemy strukturę drzewa

Wstawianie (*t*=3)



20, 30, 10, 40, 50 ;
5, 8, 15, 25,

- Wstawiamy tylko do liścia (!)
- Przed przejściem od jednego węzł. wewn. do nast. sprawdzamy, czy węzeł nie jest pełny, jeśli jest pełny => rozbijamy – strat. prewencyjna!!!
- Działa też strategia reaktywna!

B-drzewa usuwanie (prewencyjne)

- Niech *x* oznacza węzeł, z którego usuwamy klucz *k*
- Procedura rekurencyjna (uwaga: przed wywołaniem gwarantujemy, że liść (z wyjątkiem korzenia) ma co najmniej *t* kluczy) [strategia prewencyjna!]
  1. *x* – jest liściem i zawiera *k*
  2. *x* – jest węzłem wewnętrznym i zawiera *k*
  3. *x* – jest węzłem wewnętrznym i nie zawiera *k*

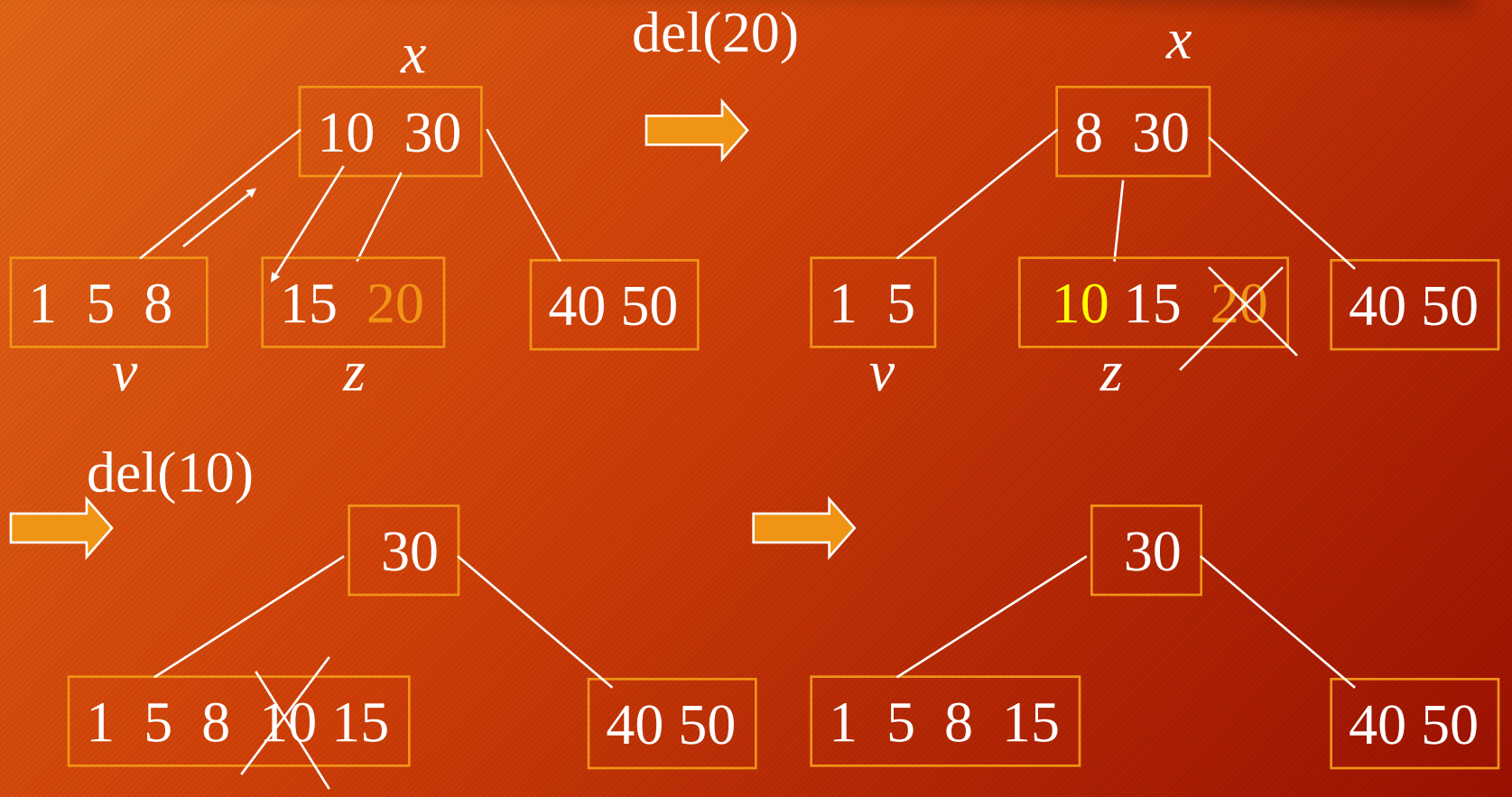
B-drzewa usuwanie c.d.

1. [klucz jest w liściu] Usuń *k* z liścia *x*
2. [klucz jest w węźle wewn. x]
   a. jeśli *y* jest synem poprzedzającym *x* (co do wartości klucza) i ma co najmniej *t* kluczy, to wyznacz następnik *succ*(*k*), nadpisz nim *k* oraz rekurencyjnie usuń succ(*k*) z *y*;
   b. jw. tylko *y* jest synem następującym po *x* (względem klucza *k*)
   c. następnik i poprzednik mają po t – 1 kluczy – scal, tj. przenieś klucze z poprzednika do następnika i usuń *k* z *x*;
3. [klucz nie w węźle wewn. x] Wyznacz węzeł *z*, rozpocz. poddrzewo, które może zawierać *k* - usuń rekurencyjnie *k* z tego poddrzewa, jeśli *z* zawiera *t* – 1 elementów – popraw drzewo [c.d.n]

Usuwanie c.d.

- 3a) jeden z braci *z* (ozn. *v*) ma *t* kluczy (lub więcej) – przenieś odp. klucz z *x* do *z* oraz z *v* do *x*
- 3b) obaj sąsiedni bracia *z* mają *t* – 1 kluczy – połącz i przesuń w środek nowego węzła klucz rozdzielający z *x*

Usuwanie – przypadki 3a i 3b.



Koniec!

Politechnika Warszawska

Algorytmy i struktury danych

Odc. 5. Kopce i uzupełnienia

Dr hab. inż. **Andrzej Zalewski**

Wydział Elektroniki i Technik Informacyjnych

POLITECHNIKA WARSZAWSKA

Kopce = kolejki priorytetowe

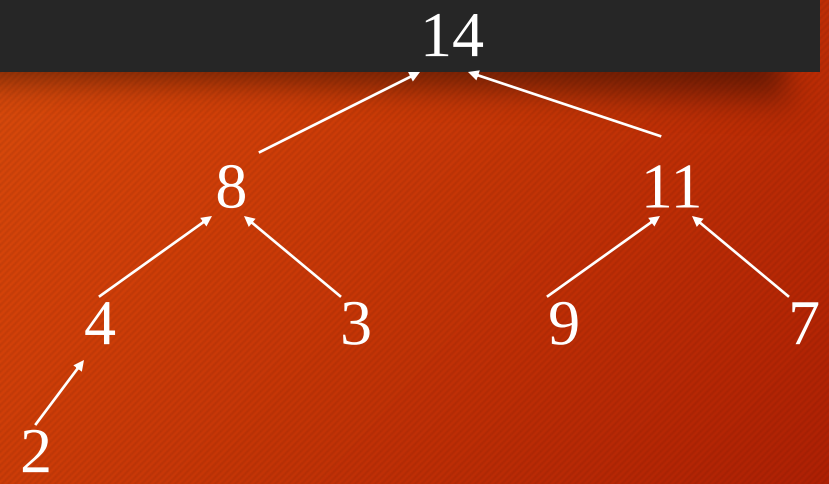
Kopce złączalne – definicja

- Operacje na kopcach złączalnych (ang. Heap)
  - *Insert*(*H*, *x*) – wstawienie elementu *x* o kluczu *key*[*x*]
  - *Minimum*(*H*) – zwraca wskaźnik najmniejszego elementu w kopcu *H*
  - *Extract-Min*(*H*) – usuwa z kopca H węzeł o minimalnym kluczu
  - *Union*(H_1, H_2) – łączy kopce H1 i H2 w jeden kopiec
  - *Decrease-Key*(*H*, *x*, *k*) – zmienia wartość klucza *key*[*x*] na *k*<= *key*[*x*]
  - *Delete*(*H*, *x*) – usuwa węzeł *x* z kopca *H*

Kopiec binarny przypomnienie

	1	2	3	4	5	6	7	8
T	14	8	11	4	3	9	7	2

TABLICA



DRZEWO

- Kopiec: tablica o indeksach ze zbioru $I=\{1, 2, \ldots, N\}$
- dla danego $i \in I$: mamy
  - $left(i) = T[2 * i]$
  - $right(i) = T[2 * i + 1]$
  - $parent(i) = \lfloor i/2 \rfloor$ dla $i > 1$
- własność kopca: dla każdego $i \in I$, $i>1$ $T[parent(i)] \geq T[i]$
- wniosek: największy element jest w korzeniu!

Kopiec binarny

- *Insert*(*H*, *x*) – wstawiamy „na koniec” i przesuwamy w góre (czas log)
- *Minimum*(*H*) – jasne (wierzchołek) (czas stały)
- *Extract-Min*(*H*) – zamieniamy wierzchołek z ostatnim węzłem (czas log)
- *Union*(H_1, H_2) – budujemy nową tablicę i robimy z niej kopiec (czas liniowy)
- *Decrease-Key*(*H*, *x*, *k*) – przesuwamy po kopcu w górę (czas log)
- *Delete*(*H*, *x*) – (czas log)

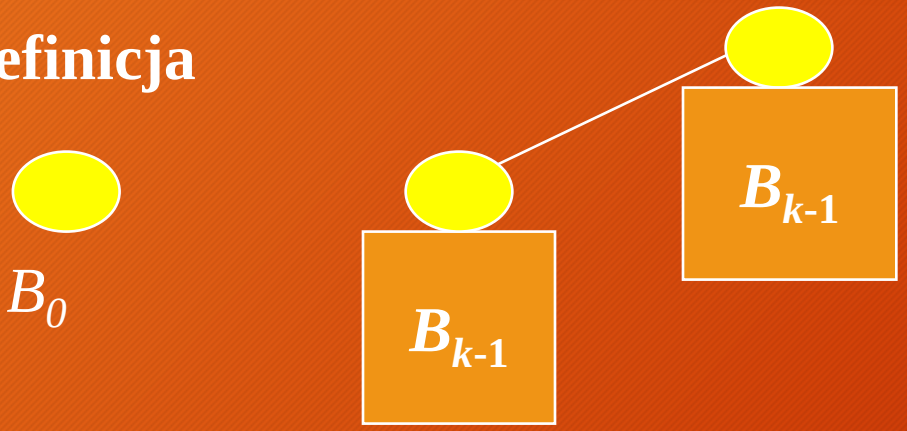
Kopce dwumianowe

Kopce dwumianowe – definicja

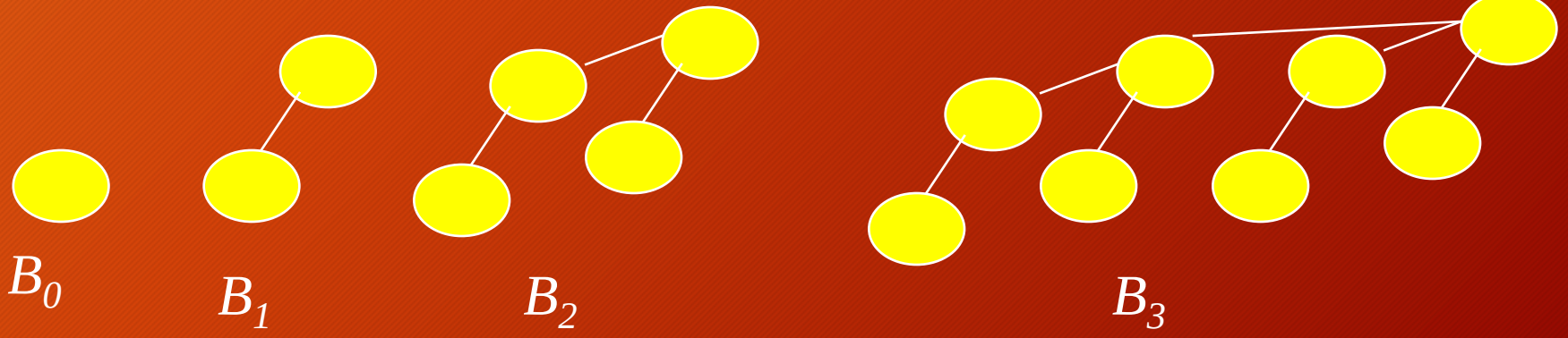
- Kopiec dwumianowy *H* jest zbiorem drzew dwumianownych o nast. właściwościach:
  - $key[x] \geq key[parent[x]]$ /klucz w węźle jest nie mniejszy niż klucz ojca – własność kopca/
  - dla każdego stopnia $d \geq 0$ istnieje w kopcu co najwyżej jedno drzewo dwumianowe

Drzewa dwumianowe – def. i przykład

Definicja



Przykład



Drzewa dwumianowe – właściwości

- Jeśli B_k jest drzewem dwumianowym, to:
  - zawiera ono 2^k węzłów;
  - ma wysokość k;
  - dokładnie $symb\_newt(k, i)$ (ka nad i) węzłów znajduje się na głębokości i
  - jeśli synów korzenia ponumerujemy od k – 1 do 0, to węzły te są korzeniami drzew B_{k-1}, B_{k-2}, ..., B_0 /z definicji/

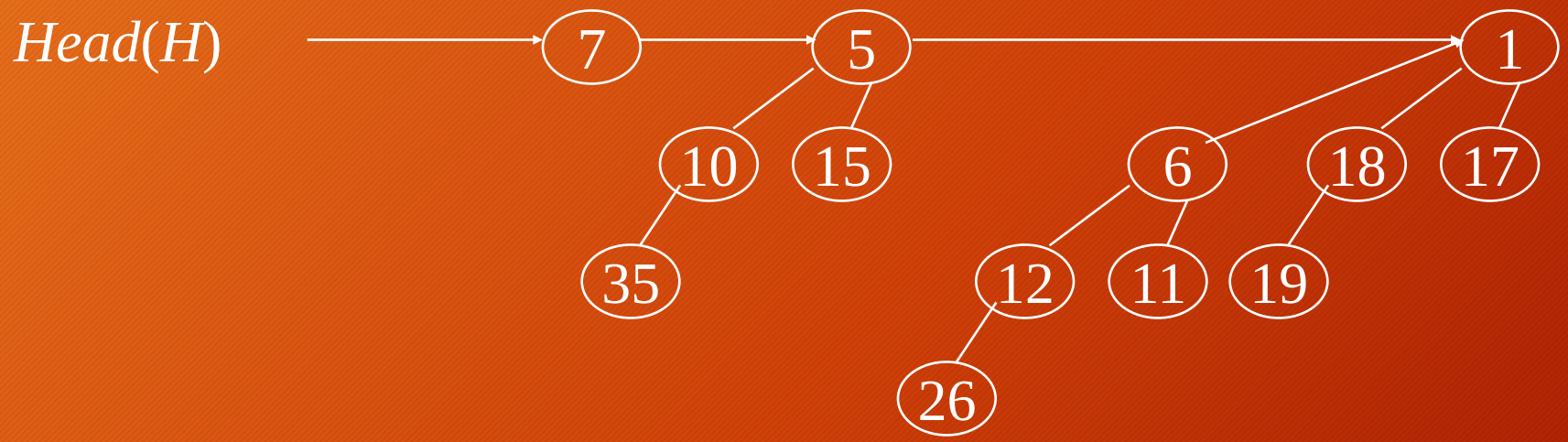
Kopiec dwumianowy właściwości

- Kopiec dwumianowy, w którym najwyższy stopień drzewa wynosi d zawiera co najwyżej $(2^{d+1} - 1)$ elementów
- Odwrotnie: liczba drzew w kopcu dwumianowym o n elementach jest nie większa niż $\lfloor \log_2 n \rfloor + 1$ (ile bitów potrzeba by zapisać n)

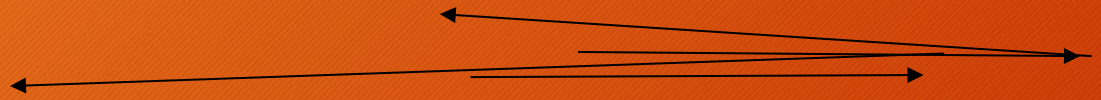
Kopiec dwumianowy jako struktura danych

- Drzewa dwumianowe zapamiętujemy zgodnie z regułą: na lewo syn, na prawo bracia (węzły na tym samym poziomie tworzą listę)
- *x*=(*key*, *parent*, *child*, *sibling*, *degree*)
  - *child* – wskazuje skrajnie lewego syna węzła *x*
  - *sibling* – wskazuje brata z prawej strony węzła *x*
  - *degree* – liczba synów węzła *x*
- Lista korzeni drzew jest uporządkowana wg stopnia (rosnąco)

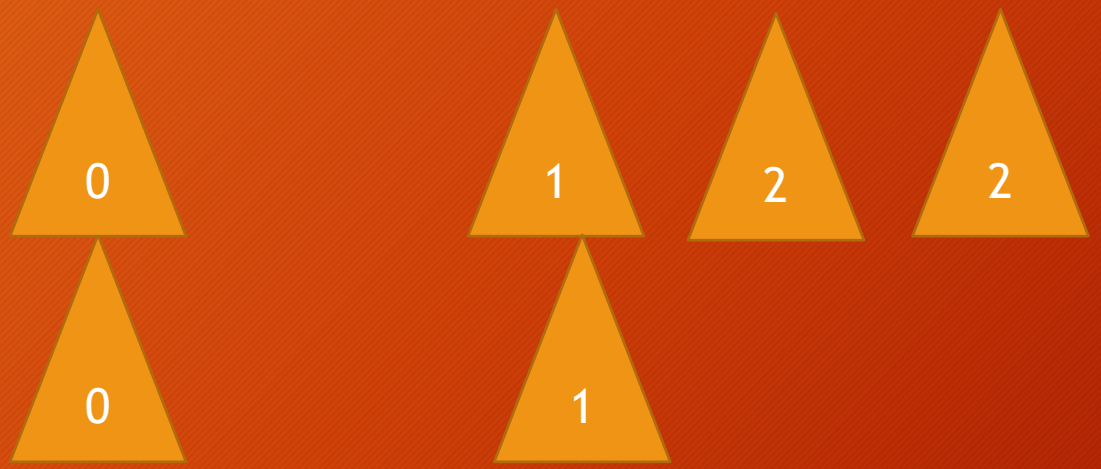
Kopiec dwumianowy - przykład



H1



H2



Kopiec dwumianowy – operacje

- *Minimum*(*H*)
  - przeglądamy listę korzeni drzew od *x*=*head*(*H*), kolejno przez *x*=*sibling*(*x*), do końca listy korzeni – złożoność proporcjon. do liczby drzew w kopcu
- *Union*(*H*1, *H*2) /łączenie dwóch kopców/
  - pomysł (pierwsze przybliżenie):
    - drzewa o tych samych stopniach w obu kopcach łączymy w nowe drzewo dwumianowe
      - w wyniku połączenia drzew B_k powstaje drzewo B_{k+1}, które może powstać z drzewa B_k, które trzeba będzie połączyć z innym drzewem B_k
    - drzewa o różnych stopniach dodajemy do listy drzew (w odpowiednim miejscu)

Kopiec dwumianowy – łączenie

- Łączenie – szczegóły:
  - łączymy listy korzeni obu kopców *H1* i *H2*, tak by na nowo powstałej liście korzeni stopnie drzew zaczynających się w nich były ułożone niemalejąco
  - w wyniku utworzenia listy i wcześniejszego łączenia drzew mogą wystąpić następuj. sytuacje (*x* – oznacza badany korzeń):
    1. *x* i *sibling*[*x*] mają różne stopnie => zostawiamy je na liście, przesuwamy x na *sibling*[*x*]
    2. *x* jest pierwszym z trzech korzeni o tym samym stopniu (możliwe tylko w wyniku wcześniejszego połączenia drzew) => przesuwamy *x* jak w przypadku 1 */**skąd ten przypadek**/*
    3. *x* jest pierwszym z dwóch korzeni o równych stopniach łączymy drzewa zaczynające się w *x* i w *sibling*[*x*], przy czym korzeniem nowego drzewa zostaje *x* jeśli
       key[*x*] < *key*[*sibling*[*x*]] lub *sibling*[*x*] w innym razie

Kopiec dwumianowy – łączenie

- Uwaga: łączenie kopców działa w czasie proporcjonalnym do liczby drzew w obu kopcach

Kopiec dwumianowy – wstawianie, usuwanie najmn.

- Wstawianie /*Insert*(*H*, *x*)/
  - tworzymy kopiec H_1 – jednoelementowy zawierający *x*
  - scalamy z kopcem *H*: *Union*(H_1, *H*)
- *Extract-Min*(*H*) /usuwanie najmn. elem./
  - znajdujemy korzeń *x* o najmniejszym *key*[*x*]
  - usuwamy korzeń z listy korzeni
  - odwracamy kolejność dzieci węzła *x* i tworzymy z niej kopiec H_1
  - *Union*(*H*, H_1) /uwaga: *H* ma usunięte drzewo, które zawierało korzeń/

Kopiec dwumianowy – zmniejszanie wart. klucza

- *Decrease-key*(*H*, *x*, *k*) /uwaga *key*[*x*]>*k*/
  - jeśli zmniejszony klucz *k* jest mniejszy od *key*[*parent*[*x*]], to zamieniamy dane *x* i *parent*[*x*]
  - przesuwamy *x*:=*parent*[*x*] itd.
- Procedura analogiczna do przesiewania w kopcu binarnym
- *Delete*(*H*, *x*) /usuwanie/
  - *Decrease-key*(*H*, *x*, -∞)
  - *Extract_min*(*H*)

Kopiec dwumianowy – złożoność / czasy operacji

- *Insert*(H, x) – ~log_2(n)
- *Minimum*(H) – max. $\lfloor log_2\ n \rfloor + 1$ /l-ba sprawdzeń
- *Extract-Min*(H) – ~$\log_2$(n) (znalezienie min, odwrócenie listy korzeni i złączenie)
- *Union*(H_1, H_2) – ~($log_2(n_1) + log_2(n_2)$)
- *Decrease-Key*(H, x, k) – co najwyżej $\lfloor \log_2\ n \rfloor$ /głębokość drzewa/
- *Delete*(H, x) – ~$log_2(n)$

Kopce Fibbonacciego

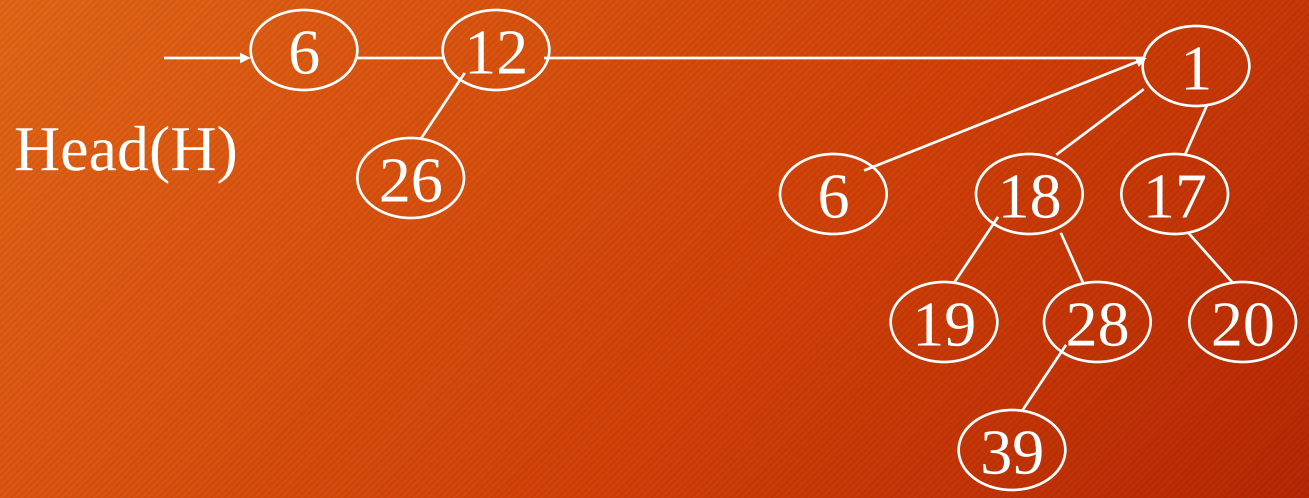
Kopce Fibbonacciego

- Rozluźniamy warunki dla kopca dwumianowego:
  - lista korzeni nieuporządkowana wzgl. stopnia
  - kopiec składa się w przybliżeniu (z wyjątkiem niektórych przypadków) z nieuporządkowanych drzew dwumianowych – jeśli stopień korzenia wynosi k>0, to drzewo U_k składa się z dwóch nieuporządkowanych drzew dwumianowych U_{k-1} w których jedno jest dowolnym synem drugiego
  - synowie korzenia drzewa U_k tworzą cykliczną nieuporządkowaną listę korzeni drzew U_{k-1}, ..., U_0
  - dzieci danego węzła tworzą listę cykliczną,
- Uwaga: brak uporządkowania i cykliczność pozwalają wykonać operacje łączenia list i usunięcia elementu z listy w czasie stałym!

Kopce Fibbonacciego

- Składowe węzła:
  - x=(key, left, right, parent, degree, mark)
    - *left* i *right* – służą do budowy listy cyklicznej
    - *degree* – liczba dzieci węzła
    - *mark* – czy węzeł stracił syna od ostatniej chwili, kiedy sam został synem innego węzła /tzw. *węzeł zaznaczony*/
- Składowe opisujące kopiec:
  - *H*=(*min*, *n*)
    - *min* – wskazuje najmniejszy element w kopcu (zapewnia dostęp do listy korzeni)
    - *n* – liczba węzłów w kopcu *H*

Kopiec Fibbonacciego – przykład



Kopiec F. - operacje

- Porządkowanie struktury odkładamy na później (tu: do operacji: *Extract-min*), większość operacji pogłębia chaos
- Wstawianie */Insert*(*H*, *x*)*/*:
  - tworzymy jednoelementową listę cykliczną zawierającą *x*;
  - dodajemy ją do cyklicznej listy korzeni
  - aktualizujemy *min*[*H*] oraz *n*[*H*]

Kopiec F. – operacje c.d.

- *Min*(*H*) – trywialne
- *Union*(*H1*, *H2*)
  - łączymy listy korzeni kopców *H1* i *H2*;
  - ustal nowe *min*[*H*] oraz *n*[*H*]
- Wszystkie powyższe operacje działają w czasie stałym (same przypisania, żadnych iteracji)!!!!
- Opeacje: Extract-min, Decrease-Key oraz Delete-Key – skomplikowane

Kopiec F. – usuwanie najmn.

- *Extract-Min*(*H*) – ogólnie:
  - znajdujemy węzeł minimalny
  - usuwamy węzeł minimalny z listy korzeni, listę korzeni poddrzew drzewa minimalnego wstawiamy do listy korzeni
  - w liście korzeni łączymy korzenie tego samego stopnia czyniąc węzeł *x* synem węzła *y*, gdy *key*[*x*] > *key*[*y*], operację tę kontynuujemy dopóty, dopóki na liście korzeni są tylko korzenie różnych stopni (pole *degree*)

Kopiec F. – scalanie drzew

- Scalanie wykonujemy po usunięciu elementu minimalnego i wstawieniu odp. poddrzew do listy korzeni kopca
- Na czym polega trudność:
  - Lista cykliczna korzeni nie jest uporządkowana względem stopnia
  - Lista cykliczna korzeni może zawierać wiele drzew tego samego stopnia (np. w wyniku scalenia lub wstawiania – stopień 1)
- Efektywna realizacja operacji scalania …

Kopiec F. – scalanie szczegóły

- Rozwiązanie:
  - budujemy tablicę $A[0...max\_deg(n[H]))$ wskaźników korzeni
    - $max\_deg(n)=\lfloor \lg_\phi n \rfloor$, $\phi=(1 + \mathrm{sqrt}(5))/2$ (wsp. złotego podziału)
    - inicjujemy wskaźnikami pustymi
  - przeszukujemy listę korzeni drzew kopca (x – kolejny korzeń, $d[x]$ jego stopień)
    - $A[d]$ jest puste, to przypisujemy mu wskazanie na x
    - w przeciwnym razie łączymy $A[d]$ z x w zależności od wartości $key[x]$ i $key[A[d]]$, „zerujemy” przypisanie $A[d]$, adres korzenia powstałego drzewa zapamiętujemy w A[d+1]
  - z wpisów w tablicy A tworzymy listę korzeni i ustalamy wskaźnik $min[H]$ itp.

Kopiec F. – zmniejszanie wartości klucza

- *decrease-key*(*H*, *x*, *k*): /*k*≤*key*[*x*]/ (!)
- Dwa przypadki:
  - nowa wartość klucza nie narusza porządku w kopcu => struktura bez zmian
  - jeśli narusza: „wprowadzamy kontrolowany bałagan”
    - zasadniczo: odcinamy *x* od jego ojca (wraz z całym poddrzewem) i dodajemy do listy korzeni
    - (*) odcinając węzeł sprawdzamy, czy ojciec był zaznaczony, jeśli nie był – zaznaczamy i koniec, jeśli był – odcinamy i dodajemy do listy korzeni i idziemy dalej w górę drzewa
    - danemu węzłowi można „bezkarnie” odciąć tylko jedno poddrzewo
    - jeśli x stało się w pewnym momencie korzeniem, to jest „odznaczane”
  - przypadki brzegowe:
    - *x* jest korzeniem – nie trzeba nic zmieniać (poza *max*[*H*])

Kopiec F. – usuwanie węzła

- *Delete*(*H*, *x*)
  - *Decrease-Key*(*H*, *x*, -∞)
  - *Extract-Min*(*H*)
- Tak jak w kopcach dwumianowych (!)

Kopiec Fibb. – czasy operacji

- stały: *min*, *union*
- $O(D(n))$ – *extract min*
- *decrease-key* – czas stały
- delete – O(D(n))
- $D(n) \leq \lfloor lg_{\phi} n \rfloor$

Porównanie kopców

Operacja	*K. binarny cz. pesymist.*	*K. dwumian. cz. pesym.*	*K. Fibb cz. zamort.*
Insert	$\Theta(\lg n)$	$O(\lg n)$	$\Theta(1)$
Min	$\Theta(1)$	$O(\lg n)$	$\Theta(1)$
Extract-min	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$
Union	$\Theta(n)$	$O(\lg n)$	$\Theta(1)$
Decrease-key	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(1)$
Delete	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$

Wyszukiwanie wzorca w tekście

Odc. 7.

Plan wykładu

- Wyszukiwanie wzorca w tekście
  - algorytm naiwny
  - algorytm Rabina-Karpa
  - algorytm z wykorzystaniem automatu
  - alg. Knutha-Morrisa-Pratta (KMP)
  - alg. Boyer’a-Moore’a (B&M)

Sformułowanie problemu

- Dany jest:
  - n, m – naturalne, $n >> m$
  - $T[1…n]$ – łańcuch znaków
  - P[1…m] – wzorzec szukany w T
- Zadanie, znaleźć takie przesunięcie $s \in \{0, 1, ..., n - m + 1\}$ wzorca, że:
  - $P[1...m] = T[1+s ... s + m]$
    - *„przesunięty o s wzorzec pasuje do tekstu”*

Algorytm naiwny

- Sprawdzamy warunek P[1...m]=T[s+1...s+m], dla *s*=0...*n* – *m* + 1
- W pesymistycznym przypadku czas jest proporcjonalny do (n – m+1)*m
  - T=aaaaaaaaaaaaaaaaaaaaaa, W=aaaab
- Wada:
  - nie korzystamy z informacji o tekście uzyskanej przy sprawdzaniu dla poprzedniego *s*.

Algorytm Rabina-Karpa

- Idea:
  - ograniczamy liczbę porównań całego wzorca do przypadków „bardziej” prawdopodobnych
- Realizacja:
  - traktujemy *W* jako liczbę *w* (dla uproszczenia zakładamy tu, że alfabet *A*={0, 1, 2, ..., 9})
  - wszystkie badane podsłowa zapisujemy jako liczbę $t_s=10^m * T[s+1]+10^{m-1} * T[s+2] + ... + 10 * T[s+m-1] + T[s+m]$

_, _, _ s+1, s+2, ..., s+m- s+m, s+m+1, _

Algorytm Rabina-Karpa

_, _, _ s+1, s+2, ..., s+m- s+m, s+m+1, _

- t_s=10^{m-1} * T[s+1]+10^{m-2} * T[s+2] + …. + 10 * T[s+m-1] + T[s+m]

_, _, _ s+1, s+2, ..., s+m- s+m, s+m+1, _

  - t_{s+1}=10^{m-1} * T[s+2]+10^{m-2} * T[s+3] + …. + 10 * T[s+m] + T[s+m+1] = t_s * 10 – 10^m * T[s+1] + T[s+m+1]
  - możemy liczyć t_{s+1} na podstawie t_s przy małym nakładzie obliczeń.

Algorytm Rabina-Karpa

- *s* jest prawidłowym przesunięciem, jeśli $t_s = w$
- Jeśli
  - *w* możemy obliczyć w czasie prop. do *m* (liczymy raz, najefektywniej schematem Hornera)
  - wszystkie wartości t_s w łącznym czasie proporcjonalnym do *n*, to cała złożoność zamknie się czasem proporcjonalnym do $n + m$

Algorytm Rabina-Karpa

- Problem: w i t_s mogą być duże!
- Rozwiązanie: operacje wykonujemy mod q, gdzie q dobiera się tak, że $10q$ mieści się w słowie komputera, ogólnie dla d-elementowego alfabetu $d*q$ winno mieścić się w słowie maszyny
- Problem: $t_s =_{\text{mod } q} w$ nie implikuje $T[s+1...s+m] = W[1...m]$ – liczby mogą być kongruentne (?) (przystają, tzn. mają tą samą resztę z dzielenia)
- Rozwiązanie: dla $t_s=w$ – sprawdzamy, sprawdzeń nie będzie „dużo”, jeśli q jest odpowiednio duże!!!
- Czas działania: proporcj. do m + n + czas na weryfikację błędnych strzałów

Algorytm Rabina-Karpa

- Można przyjąć, że liczba błędnych strzałów nie przekracza n/q (1/q – prawdopodobieństwo, że t_s przystaje do w mod q)
- Czas oczekiwany jest proporcjonalny do n + m

Algorytmy wysz. wzorca + automaty skończone

- Pomysł:
  - budujemy automat rozpoznający wzorzec
  - zbudowany automat „karmimy” tekstem – każdy znak jest wtedy badany jeden raz => czas proporcjonalny do N
  - problem:
    - czas zbudowania automatu

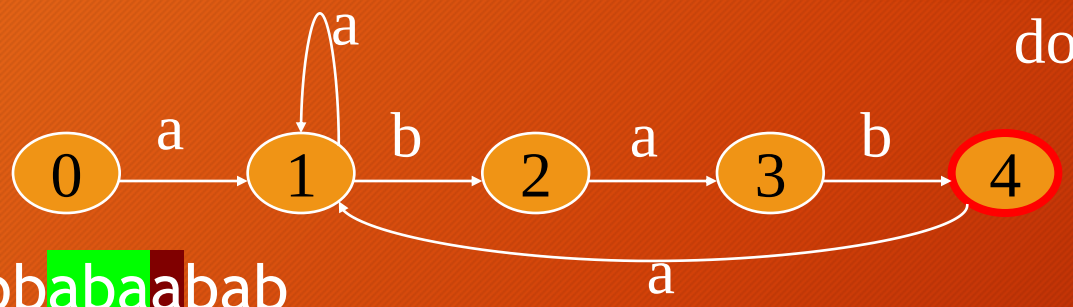
Automaty skończone

- $M = (S, s_0, S_A, A, \delta)$:
  - S – zbiór stanów
  - s_0 – stan początkowy
  - S_A – zbiór stanów akceptujących
  - A – alfabet wejściowy
  - $\delta: A \times S \rightarrow S$ – funkcja przejść

Automat rozp. wzorzec przykład

Pominięto: linie prowadzące z powrotem do stanu 0

- W = abab



- W = bbbabaabab

  - automat nie zawsze jest łatwo skonstruować (powroty w różne miejsca):
    - bacabacaba
  - konstrukcja winna polegać na określeniu sposobu powrotu

Algorytm wykorzystujący automat

- S = 0; /stan początkowy/
- dla i=1...n
  - S= δ(T[i], S)
  - jeżeli *S=m* /stan końcowy/ => znaleziono wzorzec – wypisz przesunięcie s = i – m

Budowa automatu skończonego rozpozn. wzorzec

- W naszym przypadków składowe automatu definiujemy następująco:
  - $S=\{0, \ldots, m\}$ /liczba zaakceptowanych znaków/;
  - $\delta(a, s) = \sigma(W_s a)$, $a \in A$, W_s – s znakowy przedrostek (prefiks) wzorca
  - $\sigma(x) = \max\{k: \text{suffix}(W_k, x)\}$, ozn. to, że wartością funkcji jest długość najdł. prefiksu (przedrostka) wzorca, który jest także sufiksem x
  - np. W=aca, $\sigma(acabaca)=3$, $\sigma(acbac)=2$,

Automaty skończ. rozp. wzorzec

- *W*=bacabacaca
- $\delta(b, 4)$ = 5 długość najdłuższego prefiksu *W* będącego jednocześnie przyrostkiem {b b} lub {ba b} {bac b} {baca b} {bacabb}... /W_4=baca/ {baca b}
- δ(b, *8*) = 5, długość najdłuższego prefiksu *W*, będącego przyrostkiem bacabacab /bacab/

Szkic algorytmu /z definicji automatu/

- przeglądamy wszystkie stany s={0...m},
  - w stanach *s* dla wszystkich znaków alfabetu {A, ..., Z} wejściowego, badamy
    - długość najdłuższych przedrostków wzorca będących przyrostkami $\{W_s a\}$
- dla małych wzorców czas budowy automatu nie jest wielkim problemem (czas proporcjonalny do $m^{3*}|A|$)
- jak rozwiązać problem budowy automatu dla dłuższych wzorców – algorytm Knutha-Morrisa-Pratta.

Algorytm KMP i B&M

- Idea:
  1. Wróćmy do algorytmu naiwnego
  2. KMP: w każdym przesunięciu wykonujmy dokładnie tyle porównań tekstu ze wzorcem ile naprawdę potrzeba (wyeliminujmy porównania „beznadziejne”)
  3. B&M: badajmy tylko takie przesunięcia, które naprawdę coś rokują, skaczmy po tekście pomijając miejsca, do których na pewno wzorzec nie pasuje

- ALGORYTM
- KNUTHA-MORRISA-PRATTA

Algorytm KMP – szkic (wreszcie)

- Dana jest funkcja $\pi(j)$, j=1...m wskazująca, jakie przesunięcia wzorca warto sprawdzić, gdy $W[j+1]$ różne od $T[i]$ (kolejny znak wzorca nie zgadza się z kolejnym znakiem tekstu);
- Algorytm KMP
  - niech i i j zmienne takie, że: ostatnie j znaków wzorca zgadza się z tekstem aż do *i-tego jego* znaku tekstu T;
  - dla każdego i
    - jeśli następny znak wzorca $W[j+1]$ nie pasuje do $T[i]$, to badamy, czy pasuje do wzorca przesuniętego o $j:=\pi(j)$, jeśli nie – powtarzamy $j:=\pi(j)$, aż $j = 0$ (nie ma czego szukać, trzeba badać następny znak tekstu);
    - jeśli pasuje – zwiększamy j i testujemy następny znak.

Rysunek!

abaababa

Co to jest funkcja $\pi(\ )$?

- $\pi(q)$, określającą maksymalną długość prefiksu W_q, która jest jednocześnie jego sufiksem:
  - $\pi(q)=\{\max k: k<q$ i P_k sufiksem $P_q\}$,
  - $\pi(1)=0$
- przewaga funkcji prefiksowej nad funkcją przejść automatu: łatwo i szybko się ją wyznacza...

Algorytm KMP

- Dla wzorca W=„baababab”, mamy:
  - $\pi(8)=0$ (W_8=„baababab”, $W_{k,max}$= „”)
  - $\pi(7)=2$ (W_7=„baababa”, $W_{k,max}$= „ba”)
  - $\pi(6)=0$ (W_6=„baabab”, $W_{k,max}$= „”)
  - $\pi(5)=2$ (W_5=„baaba”, $W_{k,max}$= „ba”)
  - ….
- Dla wzorca W = „bababababa”, dla wzorca W=„aaaaaaaaaaaaaaaa”

Algorytm KMP – budowa funkcji prefiksowej

- Znajdowanie $\pi(q)$
  - $\pi(1)=0$, k=0
  - dla $q=2,\ldots,m$
    - dopóki $k>0$ i $W[k+1] \neq W[q]$ (1)
      - k:= $\pi(k)$
    - jeżeli $W[k+1] = W[q]$ wtedy $k=k+1$ (2)
    - $\pi(q)=k$ (3)

Algorytm KMP - efektywność

- Czas działania nie gorszy niż proporcjonalny do n + m (!!!)
- W dalszym ciągu wiele testów wykonujemy bez potrzeby – np.
  - W=„abababab”, gdy porównując tekst ze wzorcem w tekście spotkamy literę „c” (można by przesunąć się w tekście o tyle liter ile do tego momentu rozpoznaliśmy znaków wzorca)
  - =>algorytm B&M

- ALGORYTM
- BOYER’A-MOORE’A

Algorytm B&M

- Badamy przesunięcia s od 0 do $n - m$
  - porównujemy tekst ze wzorcem (od końca wzorca do jego początku!), aż okaże się, że są identyczne (koniec), lub spotkamy różnicę, a wtedy
    - przeskakujemy do przesunięcia
      - $s = s + \max (\gamma[j], j - \lambda[T[s+j]])$, gdzie j pozycja we wzorcu pierwszego nie pasującego znaku
  - λ – skok wyznaczony heurystyką niezgodności
  - γ – skok wyznaczany heurystyką „dobrego sufiksu”

Alg. B&M – heurystyka niezgodności

- Przypadek, w którym spotykamy znak nie należący do wzorca
- Ogólnie: niech $0 \leq k \leq m$ będzie największym indeksem takim, że $T[s + j] = W[k]$, jeśli takie k istnieje, w przeciwnym razie przyjmujemy $k=0$ => wtedy możemy przesunąć wzorzec o $j - k$

Alg. B&M – heurystyka niezgodności

- $\lambda: A \rightarrow \{1...m\}$ – określa ostatnie wystąpienie (pozycję ost. wyst.) poszczególnych znaków alfabetu we wzorcu, jeśli znak *c* nie występuje $\lambda(c)=0$
- powoduje takie przesunięcie wzorca, że niepasujący znak w tekście jest zestawiany ze znakiem wzorca lub wzorzec jest przesuwany poza ostatnio sprawdzane okno

Algorytm B&M – heurystyka dobregu sufiksu

- $\gamma : \{1...m\} \rightarrow \{1...m - 1\}$
- *A)*
- T =a l e b a l a m a l a p a l a
- W=a l e m a l a b a l a
- a l e m a l a b a l a
- B)
- T = a l a b a l a b a l a
- W= b a l a a l a b a l a

Koniec

Politechnika Warszawska

Algorytmy i struktury danych

Odc. 8. Grafy (cz. I)

Dr hab. inż. **Andrzej Zalewski**

Wydział Elektroniki i Technik Informacyjnych

POLITECHNIKA WARSZAWSKA

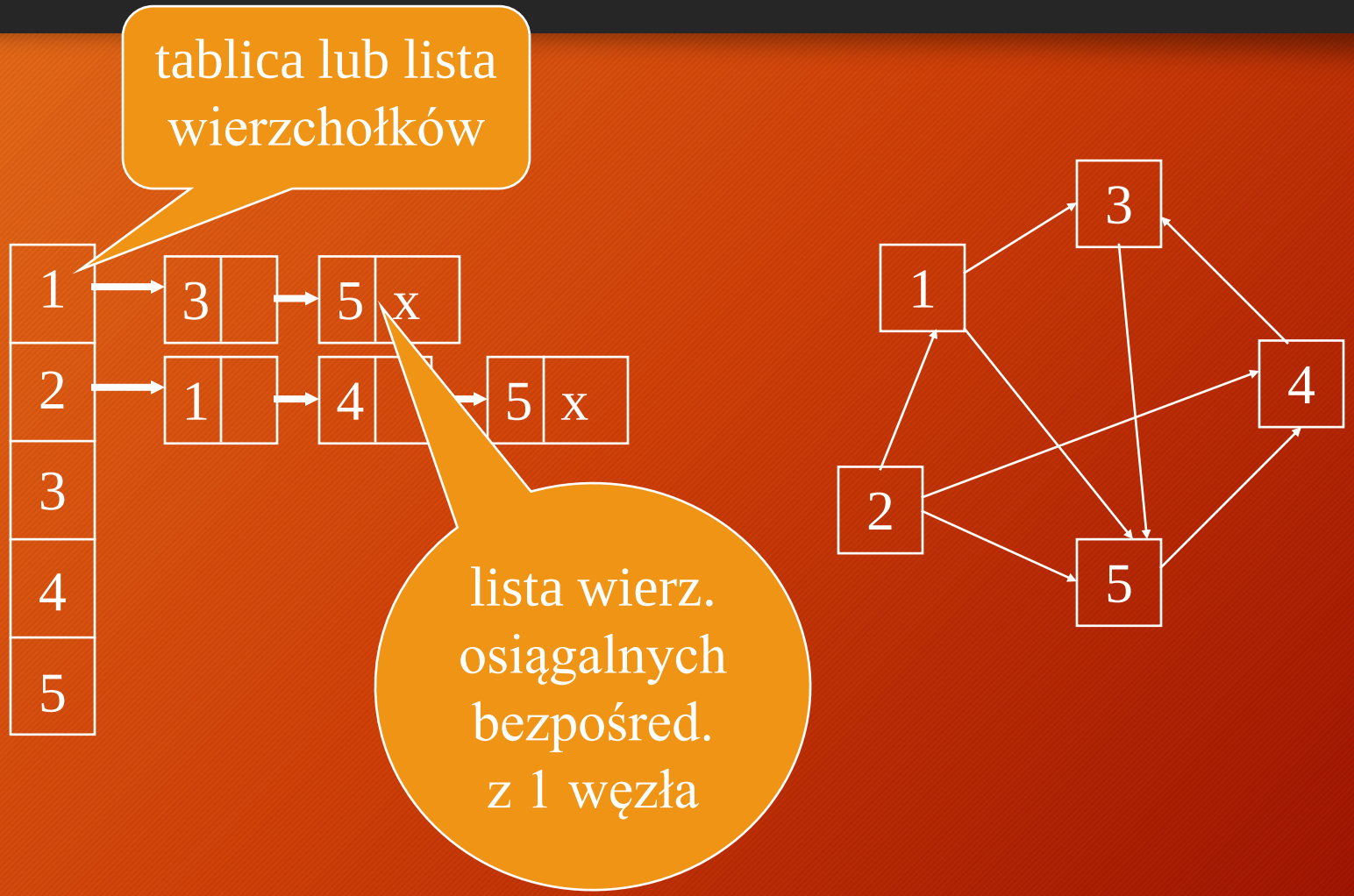
Plan wykładu

- Uwagi o reprezentacji grafów
- Podstawowe algorytmy grafowe
- Problem minimalnego drzewa rozpinającego graf:
  - algorytmy Prima i Kruskala
- Problem najkrótszej ścieżki
  - algorytmy Dijkstry i Bellmana-Forda

Reprezentacja grafów

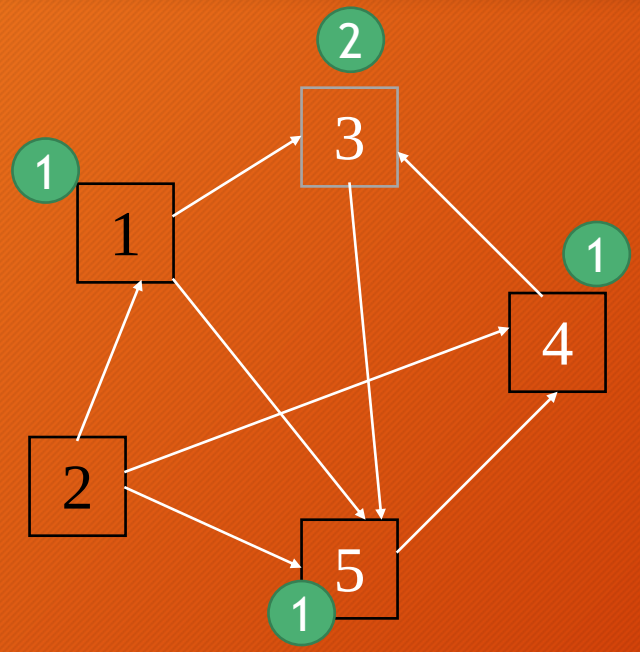
- Macierz połączeń *n* na *n*, gdzie *n* – liczba wierzchołków
- Listy sąsiedztwa – dla każdego wierzchołka *v* lista wierzchołków, do których można przejść z *v*

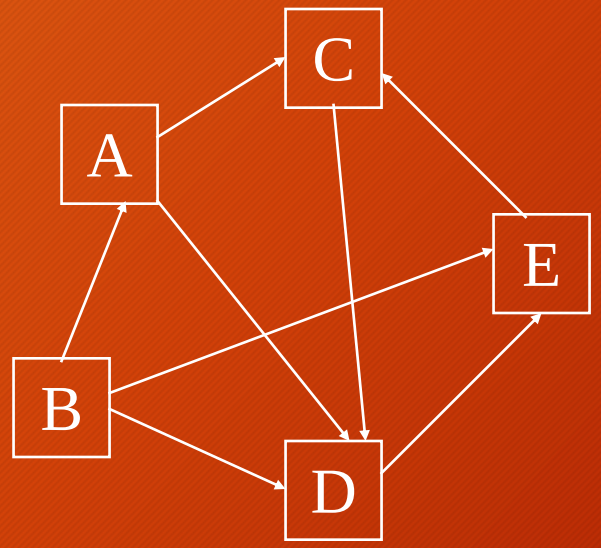
Grafy – implem. listowa



Przeszukiwanie wszerz

- Idea:
  - zaczynamy od wybranego wierzchołka początkowego
  - z każdego wierzchołka odwiedzamy kolejno wszystkie dostępne (idziemy wszerz!), a nie odwiedzone do tej pory wierzchołki; robimy tak dopóki jest jeszcze jakiś nie odwiedzony wierzchołek
- Problem: jak zapobiec powtórnemu odwiedzaniu wierzchołków?





1 5 4 3 B

Przeszukiwanie wszerz - szczegóły

- Rozwiązanie:
  - początkowo wierzchołki kolorujemy na biało;
  - wierzchołki dopiero co odwiedzone kolorujemy na szaro,
  - wierzchołki, z których odwiedzono już wszystkich sąsiadów kolorujemy na czarno (ta zmiana jest właściwie bez znaczenia)

Przeszukiwanie wszerz – algor.

- Wierzchołki grafu są na początku koloru białego, wierzchołek początkowy *s* koloru szarego
- Wstawiamy *s* do kolejki
- Pętla trwająca dopóki kolejka wierzchołków jest nie pusta:
  - wyjmujemy wierzchołek *u* z kolejki,
  - wszystkie białe wierzchołki *v* osiągalne z *u* zaznaczamy na szaro, *parent*(*v*)=*u*, wstawiamy *v* do kolejki
  - Usuwamy *u* z kolejki i kolorujemy na czarno

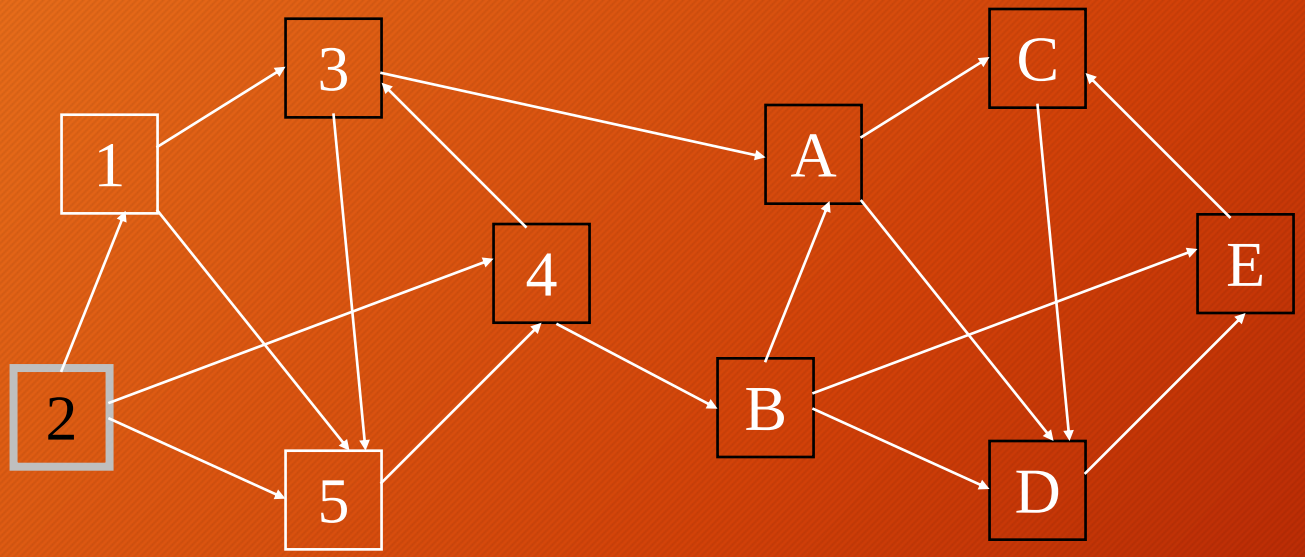
Właściwości algorytmu i rozwiązania

- Algorytm znajduje najkrótsze ścieżki, w sensie liczby pokonywanych krawędzi, między *s* a dowolnym węzłem
  - obserwacja ilustrująca powyższą własność: węzły położone bliżej od *s* znajdują się (są umieszczane) w kolejce przed, tymi które są położone dalej
  - użycie kolejki FIFO jest obligatoryjne!
- Czas działania: inicjacja (V), przeszukiwanie *E* – czas proporcj. do |V| + |E|

Przeszukiwanie w głąb

- Idea:
  - z każdego wierzchołka idziemy do następnego itd. najdalej, jak się da, dbając by nie odwiedzać odwiedzonych wcześniej wierzchołków
  - gdy nie mamy dokąd pójść (wszyscy sąsiedzi zostali odwiedzeni) wracamy do wierzchołka, z którego przeszliśmy do danego i poszukujemy „otwartych” dróg

2 4



Przeszukiwanie w głąb – szczegóły

- Wierzchołki grafu kolorujemy na biało, zerujemy wskaźniki poprzedników w ścieżce przeszukiwania (*parent*) i wskaźniki czasu *d i f*
- *time:=0 // zmienna globalna*
- Dla każdego białego wierzchołka *v* aplikujemy procedurę visit(*u*)
  - zmień kolor *u* na szary, *time*:=*time*+1; *d*(*u*)=*time*
  - dla każdego białego wierzchołka *v* połączonego z wierz. *u*,
    - *parent*(*v*)=*u*; wywołaj rekurenc. *visit*(*v*)
  - pokoloruj *u* na czarno, *time*:=*time*+1
  - *f*(*u*)=*time*

Przeszukiwanie w głąb - właściwości

- Procedurę można zrealizować z wykorzystaniem stosu (nie jest to zupełnie oczywiste)
  - W efekcie, traktując początek odwiedzin jako otwarcie nawiasów, a koniec jako zakończenie otrzymujemy poprawne wyrażenie nawiasowe
  - Można pokazać, że przedziały $<d(u), f(u)>$ oraz $<d(v), f(v)>$ są albo całkowicie rozłączne, albo całkowicie zawierają się w sobie dla dowolnej pary wierzchołków
- Czas: prop. do |V| + |E|

Sortowanie topologiczne

- Zadanie: dany jest acykliczny graf skierowany, wierzchołki grafu należy uporządkować (tzn. określić wartości całkowite $f(v)$ dla każdego wierzchołka v), tak że jeśli krawędź (u, v) należy do grafu, to wierzchołek u występuje przed v (*tzn.* $f(u) < f(v)$)

Sortowanie topologiczne – algorytm

- *//modyfikacja przeglądania w głąb*
- przeglądaj w głąb wyznaczając $f(v)$, jak tylko v zostanie całkowicie przetworzony wstaw go na początek listy
- wynik – lista wierzchołków

Sortowanie topologiczne - zastosowania

- Typowe:
  - grafy reprezentujące kolejne etapy przedsięwzięcia są acykliczne jeśli przedsięwzięcie ma początek i koniec
  - można w ten sposób wyznaczyć kolejność czynności

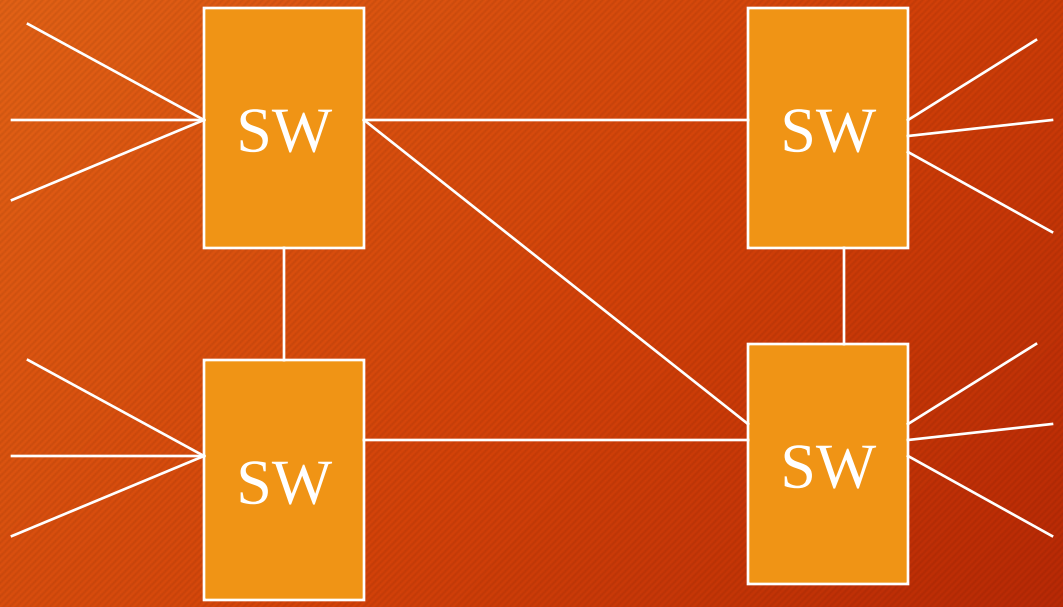
Minimalne drzewo rozpinające

Problem minimalnego drzewa rozpinającego

- Minimum Spanning Tree
- Szukamy takiego, spójnego, acyklicznego podzbioru krawędzi, że:
  - suma wag tych krawędzi jest najmniejsza z możliwych

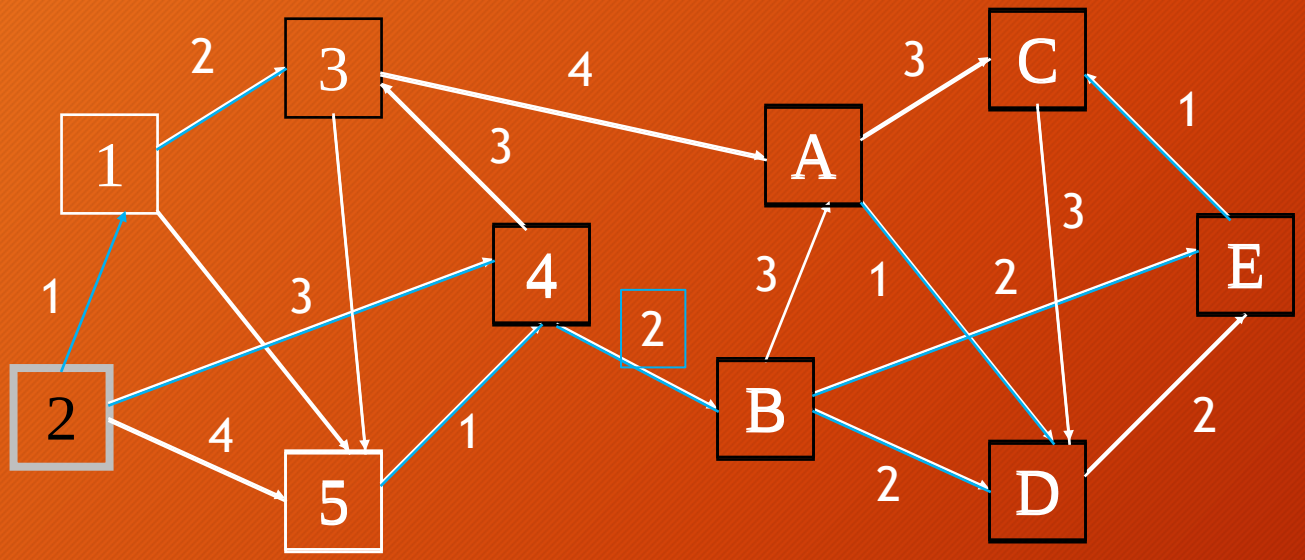
Zastosowania ST

- rozwiązywanie pętli w warstwy łącza danych sieci komputerowych



Algorytm Kruskala

- „Hodowanie lasu”
  - A = zbiór pusty
  - dla każdego wierzchołka *v* grafu G
    Make-Set(*v*)
  - dla kolejnych krawędzi (u, v) w kolejności niemalejących wag (kolejka priorytetowa(?))
    - jeżeli *Find-Set*(*u*) <> *Find-Set*(*v*) // czy krawędź należy do tego samego drzewa
      - $A = A \cup \{(u, v)\}$
      - *Union*(*u*, *v*)
  - wynik zawarty w zbiorze A


1
2
3
4
5
A
B
C
D
E
2
3
4
3
3
1
3
1
2
4
1
3
1
2
2
2

Struktury danych dla zbiorów rozłącznych

Struktury danych dla zbiorów rozłącznych

- Dana jest rodzina rozłącznych zbiorów $\{S_1, S_2, \ldots, S_k\}$
- Reprezentant – element zbioru S_i reprezentujący ten zbiór, pytając dwukrotnie powinniśmy otrzymywać tą samą odpowiedź
- Operacje:
  - *Make-Set*(x) tworzy zbiór o jedynym elem. *x*
  - *Union*(x, y) – suma zbiorów zawierających *x* i *y* (usuwa oba z rodziny, wstawia ich sumę)
  - *Find-Set*(*x*) – zwraca wskaźnik reprezentanta zbioru zawierającego *x*

- {1} {2} {3} {4} {5} {6} {7}

- {1 2} {3} {4 5} {6 7}
- {1 2 4 5} { 3} {6 7}
- {1 2 4 5 6 7} {3}

Reprezentacje: listowa

- Listowa:
  - pierwszy element listy – reprezentant
  - każdy element ma wskaźnik następnika oraz reprezentanta
- Operacja Find-Set, Make-Set w czasie stałym
- Operacje scalania: łączymy listy, uaktualniamy wskaźniki (na nowego reprezentanta!) – czas liniowy

Reprezentacje: las drzew rozłącznych

- Każde drzewo reprezentuje jeden zbiór
- Każdy element zawiera jedynie wskaźnik do swego ojca
- Korzeń zawiera reprezentanta, wskaźnik ojca wskazuje na niego samego
- Operacje:
  - *Make-set* – czas stały,
  - *Find-set* – spacer aż do korzenia, w niekorzystnym przypadku czas liniowy
  - Union – operacja scalania zamienia wskaźnik ojca w jednym z drzew na korzeń drugiego drzewa

Las drzew rozłącznych c.d.

- Wersja podstawowa daje pesymistyczny czas liniowy
- Łączenie wg rangi lub liczby elementów (Diks)– przyłączamy mniejsze do większego lub niższe do wyższego
- *find-set*(*x*) z kompresją ścieżki: wszystkie wierzchołki na drodze od *x* do reprezentanta zbioru przepinamy do tego reprezentanta (deklarujemy go jako ojca każdego z tych wierz.)
- czas wykonania ciągu *m* operacji na *n* elem. rodzinie – ca. liniowy względem *m* (stały wzgl. *n* – b. wolno rosnący wzgl. *n*)

Przykład zastosowania

- Podział grafu nieskierowanego na spójne składowe:
  - utwórz rodzinę jednoelementowych zbiorów zawierających poszczeg. wierzchołki grafu
  - dla każdej krawędzi (u, v)
    - jeśli Find-Set(u) <> Find-Set(v) //jest połączenie
      - Union(u, v)

Algorytm Kruskala

- „Hodowanie lasu”
  - A = zbiór pusty
  - dla każdego wierzchołka *v* grafu G
    Make-Set(*v*)
  - dla kolejnych krawędzi (u, v) w kolejności niemalejących wag (kolejka priorytetowa(?))
    - jeżeli *Find-Set*(*u*) <> *Find-Set*(*v*) // czy krawędź należy do tego samego drzewa
      - $A = A \cup \{(u, v)\}$
      - *Union*(*u*, *v*)
  - wynik zawarty w zbiorze A

Algorytm Prima

- Badanie kosztu połączenia przekroju grafu (węzły grafu dzielimy na podzbiory Q oraz V – Q), do V – Q stopniowo dodajemy kolejne węzły, przez które przechodzi drzewo rozpinające
  - wybór węzła – najniższy koszt połączenia między węzłami wchodzącymi w skład drzewa (V – Q) a pozostałą częścią przekroju: Q.

Algorytm Prima – szczegóły

- Prima(G, r) // *r* – węzeł początkowy, przyjm. arbitralnie
  - ładujemy wszystkie węzły do kolejki *Q*, *key*(*Q*)=∞,
  - dla *r* przypisujemy *key*[*r*]=0 (żeby określić co wyjmujemy)
  - dopóki *Q* niepusta => *u*=*Extract-Min*(*Q*)
    - dla każdego *v* takiego, że istnieje krawędź (*v*, *u*)
      - jeśli *v* jest w *Q* (tzn. nie ma jej jeszcze w drzewie) i waga krawędzi (*v*, *u*) < *key*(*v*)
        - zaktualizuj wartość klucza w kolejce
        - wskaż *u* jako potencjalnego ojca *v* w drzewie
  - W kolejce na początku stoją wierzchołki, które mają połączenie z już odkrytym drzewem

Algorytm Prima a kopce (!)

- Do realizacji możemy użyć kopca binarnego
  - utworzenie kolejki: Build-heap – czas proporcjonalny do $|V|$
  - pętla główna $|V|$ razy
  - każde Extract-Min – proporcj. do $\log_2 |V|$, łącznie $|V| \log_2 |V|$
  - wyszukiwanie sąsiednich krawędzi: łącznie $2|V|$
  - badanie przynależności do Q –w czasie stałym (znacznik)
  - zmiana wartości klucz – Decrease-Key – $\log_2 |V|$
  - razem: proporcj. do $(|V|\log_2|V| + E \log_2|V|)$

Algorytm Prima a kopca c.d.

- Dla kopca Fibbonacciego
  - zysk na Decrease-Key – czas stały
  - daje to czas proporcj. do $E + |V| \log_2 |V|$

Problem najkrótszej ścieżki

(z jednego źródła)

Problem najkrótszej ścieżki

- Ścieżką w grafie nazywamy ciąg węzłów tego grafu $p=(v_1, v_2, \ldots, v_k)$ //warunek połączenia
- Wagą ścieżki p jest suma wag krawędzi tworzących tę ścieżkę

Algorytm Dijkstry

- Prawie identycznie jak algorytm Prima tylko dla grafu skierowanego o wagach nieujemnych ... no i z tzw. relaksacją
  - czynności wstępne: dla wszystkich wierzchołków $d[v] = \infty$ (estymata odległości od wybranego wierzchołka s), $p[v]$=nil (wskaźnik poprzednika na ścieżce), $d[s]$=0 (źródło)
  - relaksacja krawędzi (u, v):
    - stwierdza, czy przechodząc przez wierzchołek u można znaleźć krótszą ścieżkę do v, a jeśli tak, to aktualizowane jest $d[v]$ i $p(v)$

Algorytm Dijkstry – relaksacja

- Relax(u, v)
  - jeżeli $d[v] > d[u] + w(u, v)$ wtedy
    - $d[v] = d[u] + w(u, v)$
- Dijkstra(s)
  - czynności wstępne, S – zbiór pusty, do kolejki priorytetowej Q (uwaga! – klucz – $d(v)$) wstawiamy wszystkie węzły grafu
  - dopóki kolejka priorytetowa Q nie pusta $u=Extract\text{-}Min(Q)$
    - $S = S \cup \{u\}$
    - dla każdego wierzchołka v sąsiadującego z u
      - Relax(u, v) // wyznacz nowe estymaty odległości od s

Algorytm Dijkstry – implem.

- Kolejka *Q* jako
  - lista liniowa – daje czas działania algorytmu rzędu $|V|^2$
  - dla grafów rzadkich – kopiec binarny, czas działania $(|V|+|E|) \log |V|$ (relaksacja jako Decrease-Key)
  - kopiec Fibbonacciego – $|E|\log|V| + |E|$

Algorytm Belmana-Forda

- Procedura
  - Wykonaj czynności wstępne (jak w alg. Dijkstry)
  - wykonaj |V| – 1 razy
    - relaksację wszystkich krawędzi grafu
  - sprawdź czy nie ma cyklu o ujemnej wadze:
    - jeśli $d(v) > d(u) + w(u, v)$, to jest cykl o ujemnej wadze [sprzeczny wynik obliczeń – krócej jest do v przez u, niż to co oszacowano!]

Algorytm Belmana-Forda

- Działa w czasie prop. do ($V * E$)
- Wyjaśnienie konstrukcji
  - skąd taka liczba iteracji?:
    - rozważmy graf składający się z N węzłów, każdy jest następnikiem poprzedniego węzła (szereg) i rozważmy działanie algorytmu
      - zapewnienie propagacji informacji od źródła
- Algorytm ma wersję równoległą
  - stanowi podstawę protokołu routingu RIP podstawowego w sieciach TCP/IP

Najkrótsze ścieżki między wszystkimi parami węzłów

- Rozwiązanie najprostsze – rozwinięcie alg. B-F
- Dane: $W=\{w_{ij}\}$ macierz incydencji z wagami, N – liczba wierzchołków
- $l^{(m)}_{ij}$ najmniejsza waga (koszt) ścieżki $i->j$ zawierającej co najwyżej m krawędzi (macierz najkrótszych odległości)
- Jak wyznaczyć $l^{(m)}$ na podst. $l^{(m-1)}$?

Najkrótsze ścieżki *N* x *N*...

- *(A)*
  - $l^{(m)}_{ij} = l^{(m-1)}_{ij}$, jeśli ścieżki nie można poprawić przeglądając wszystkie poprzedniki wierzchołka *j* (zakładając ścieżkę *i* -> *k* -> *j*, gdzie *u* – poprzednik wierzchołka *j* zgodnie z macierzą incydencji *W*)
- *lub (B)*
  - $l^{(m)}_{ij} = \min_{k=1\ldots N} (l^{(m-1)}_{ik} + w_{kj})$
- Zapis łączny: $l^{(m)}_{ij} = \min(\ (A),\ (B)\ )$
- *Bardzo podobne do Alg. B-F.*

Najkrótsze ścieżki *N* x *N*…

- Algorytm:
  - $l^{(1)}_{ij}=w_{ij}$
  - Wyznacz macierze $l^{(m)}$, dla m=1…N – *1 // N*
  - Czyniąc to w nast. sposób:
  - dla każdego (i, j) // *N^2
    - $l^{(m+1)}_{ij}$ = nieskończ.
    - dla k=1…N // *N
      - sprawdź kolejne możliwe przejścia przez sąsiadów j
      - $l^{(m+1)}$ = min ($l^{(m)}_{ij}$, $l^{(m)}_{ik} + w_{kj}$)
  - Złożoność N^4

Najkrótsze ścieżki *N* x *N*

- Jak przyspieszyć algorytm?
- Wyznaczanie $[l^{(m)}]$ na podstawie $[l^{(m-1)}]$ ma te same własności co mnożenie macierzy (min=+, + = *)
- Mamy więc *de facto*
- $L^{(N-1)} = W^{(N-1)}$, gdzie $L^{(K-1)} = L^{(K)}$, dla $K>=N$
- Możemy więc wyznaczyć L^2 -> L^4 -> $L^{/N/}$
- Mnożeń jest wtedy *de facto* $\lceil \log_2(N-1) \rceil - 1$

Alg. Floyda-Warshalla

- Początkowa macierz przybliżeń minimalnych odl. między węzłami
  - $l^{(0)}_{ij} = w_{ij}$
- Wybieramy kolejno węzły k od 1 do N
  - sprawdzamy, czy droga z i do j nie jest krótsza przez ten właśnie k
  - $l^{(k)}_{ij} = \min(l^{(k)}_{ij}, l_{ik}^{(k-1)} + l^{(k-1)}_{kj})$
- Złożoność N^3

Koniec

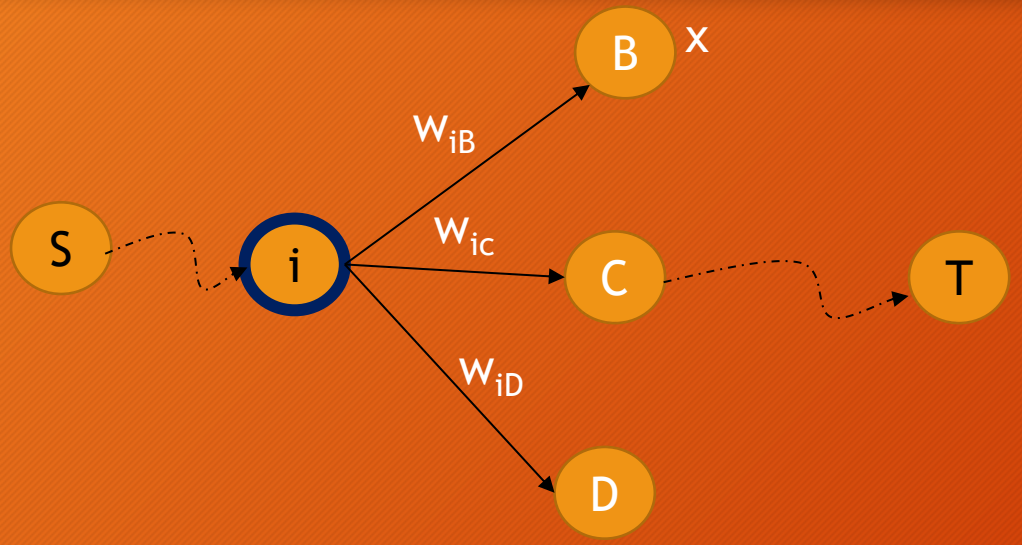
Algorytmy i struktury danych

Odc. 10. Algorytm A* (a star). Maksymalny przepływ w grafie. Metoda Fulkersona-Forda i jej odmiany.

Algorytm A*

- Zadanie: najkrótsza ścieżka z węzła początkowego (S) do końcowego (T)
- Zmieniamy algorytm Dijkstry jako wagę węzłów w grafie przyjmujemy:
  - d(i)=dist-from-start(i) + dist-to-dest(i), gdzie
  - dist-from-start(*i*) – odległość pokonana od S do i
  - dist-to-dest(j) – szacowana heurystyką odległość od j do T (końca)

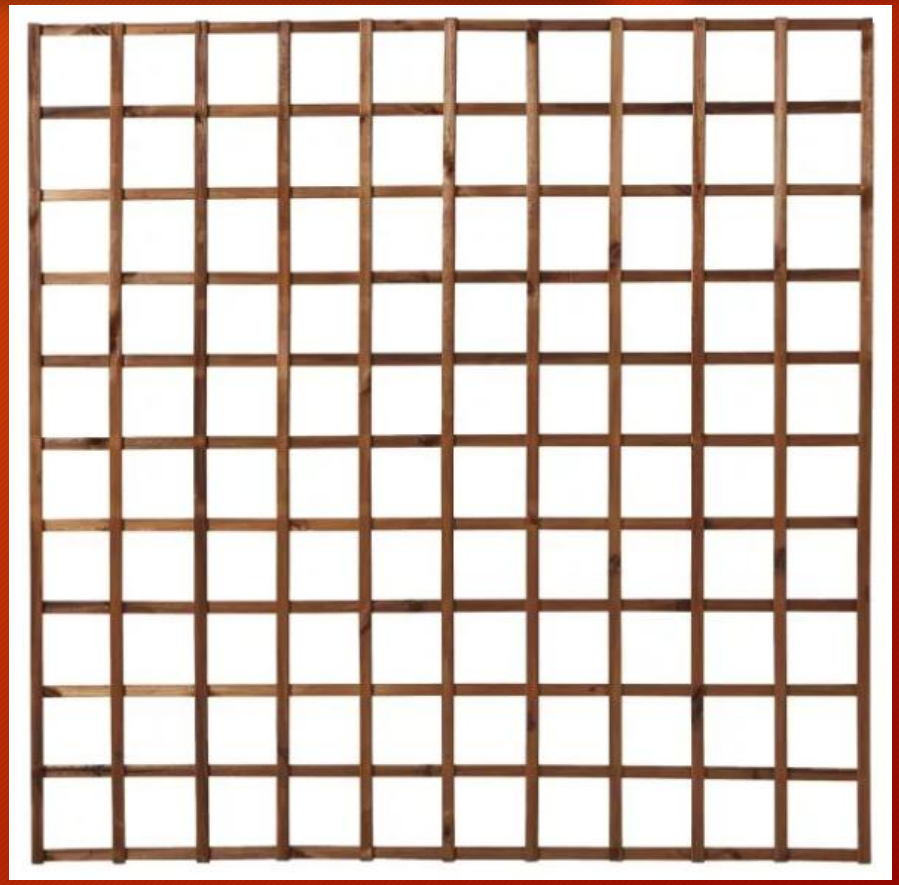
Szkic algorytmu

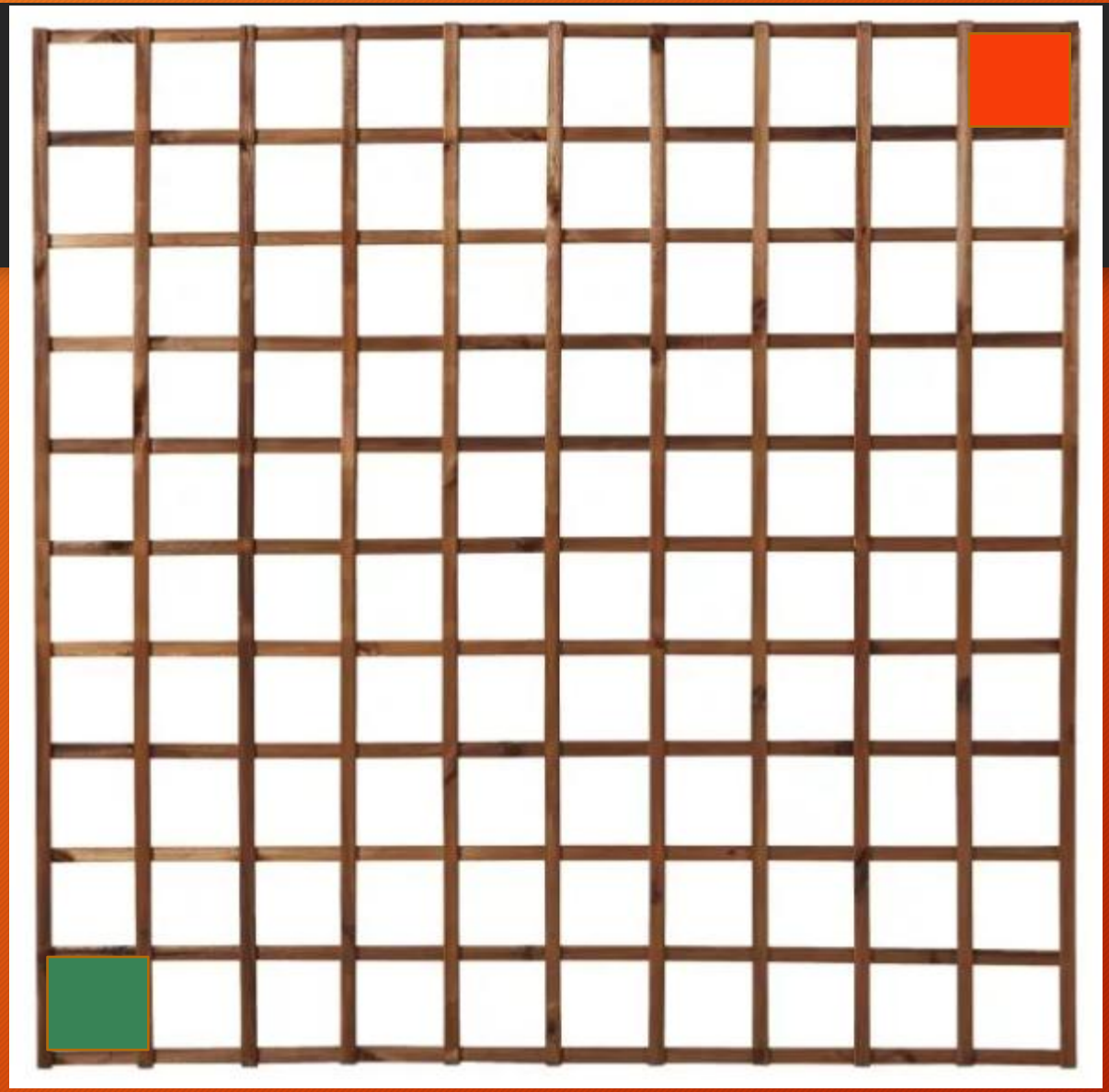


- Przygotuj struktury (jak u Dijkstry), d(*S*)=0
- Wyjmij ***i*** z kolejki **zgodnie z priorytetem**
- Dla każdego ***x*** *będącego sąsiadem* ***i***
  - jeżeli (*x* nie było jeszcze odwiedzane) lub (*dist-from-start*(x) > *dist-from-start*(*i*) + w_{ix}
    - Zaktualizuj dist-from-start(x)
    - Zaktualizuj priorytet x, d(x)= dist-from-start(x) + dist-to-dest(x)

A* - Heurystyki

- Jeśli grafem jest siatka rastrowa N x N
  - Metryka miejska
  - Odległość euklidesowa
- Można przeliczyć wszystkie odległości przed odpaleniem A*





A* - przypadki brzegowe

- Jeśli heurystyka nie przeszacowuje kosztu dotarcia do T to algorytm zawsze znajduje najkrótszą ścieżkę
- *dist-to-dest*(i) = 0, dla każdego i (bez heurystyki) – priorytetem jest odległość od początku → alg. Dijkstry
- *dist-to-dest*(i) – zawsze dokładnie odpowiada odległości z i do T – algorytm podąża najkrótszą ścieżką z *S* do *T*
- *dist-to-dest*(i) „czasem” większe od odległości z i do T – algorytm może zabłądzić (brak zbieżności).
- *dist-to-dest*(i) >> dest-from-start(i) kierujemy się odległością do celu, czyli mamy przeszukiwanie w głąb

Problem maksymalnego przepływu

Metoda Fulkersona-Forda i jej odmiany

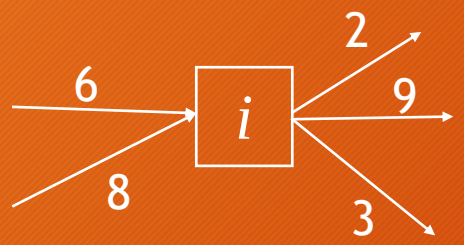
Założenia – sieci przepływowej

Dany jest graf skierowany, ważony G = (V, E), gdzie:

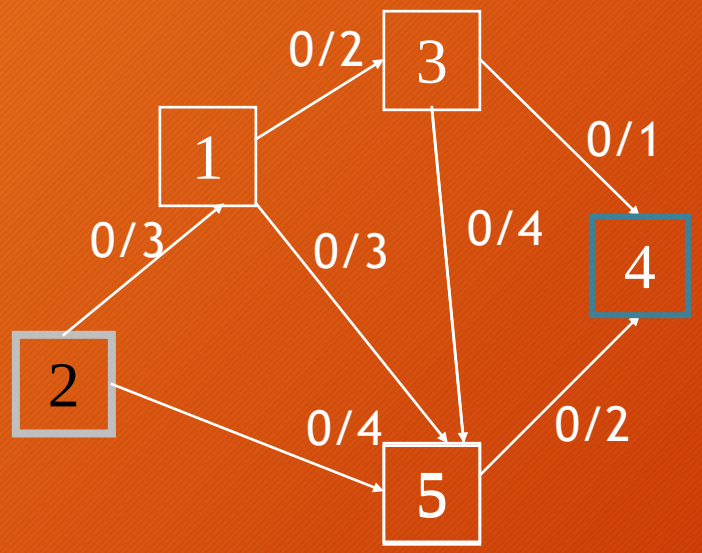
1) każdej krawędzi (i, j) przypisano: *c(i,j)* – przepustowość krawędzi, *f*(*i*,*j*) - przepływ
2) f(i, j) <= c(i, j), dla każdej pary węzłów (i, j)$\in$V$\times$V
3) wyróżniono węzeł źródło ***s*** i węzeł docelowy ***t***
4) *f(i, j) = -f(j, i) (ważne dla rozumienia dalej !!!)*
5) *Prawo Kirchhoffa – dla wszystkich* i *węzłów poza s i t suma przepływów wpływających i wypływających równa się zero*

Ilustracja prawa zachowania przepływu (Kirchhoffa)

przepływy

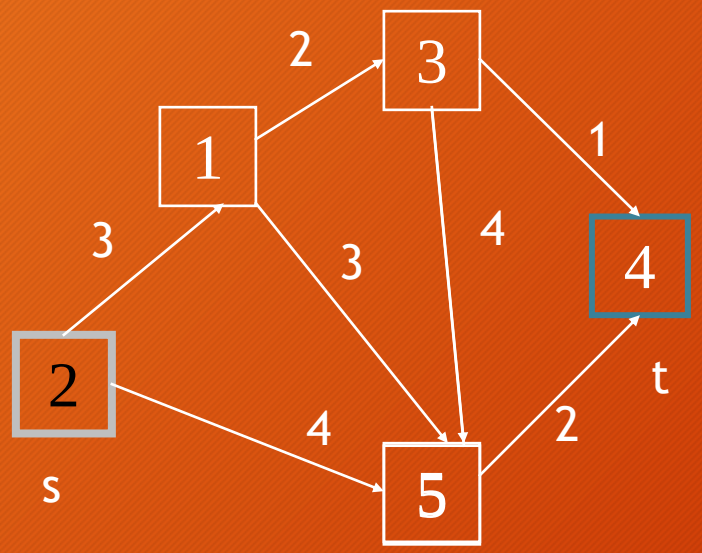


Punkt startowy



- Graf, z określonymi przepustowościami, początkowe przepływy
- (0 / 3) – oznacza, ze krawędzią o przepustowości 3 przepływ wynosi zero

Sieć rezydualna



- Graf, w którym wagami jest niewykorzystana przepustowość grafu ($c_f(i,j)=c(i, j) - f(i, j)$), tam gdzie = $c(i,j) = f(i,j)$ /nasycenie krawędzi/ można przyjąć, że krawędzi (i, j) nie ma.
- Ścieżka powiększająca – każda ścieżka z *s* do *t* w grafie rezydualnym $G_f=(V, C_f)$

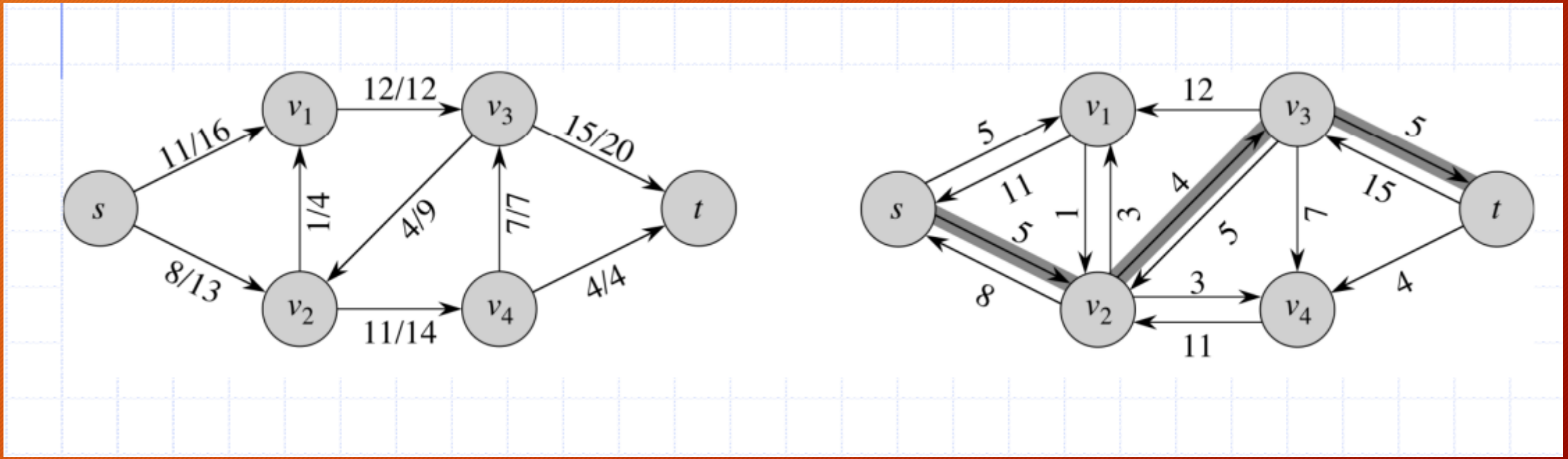
Metoda Forda-Fulkersona

1) Przyjmij f(i, j)=0 dla wszystkich (i, j) [dla wszystkich krawędzi]
2) Dopóki istnieje ścieżka *p* powiększająca przepływ z s do *t (czyli dopóki istnieje ścieżka w sieci rezydualnej)*

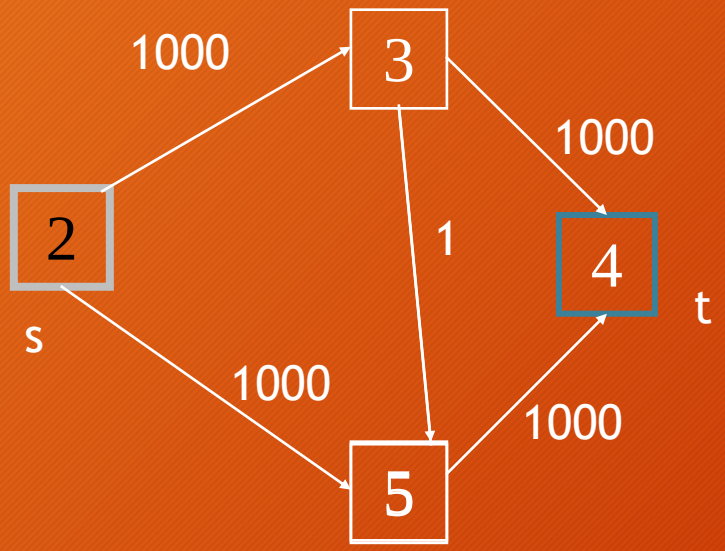
   Powiększ przepływy na ścieżce p o najmniejszy przepływ rezydualny na tej scieżce

- *Ustalone w ten sposób przepływy są maksymalne*

Ścieżka powiększająca przepływ



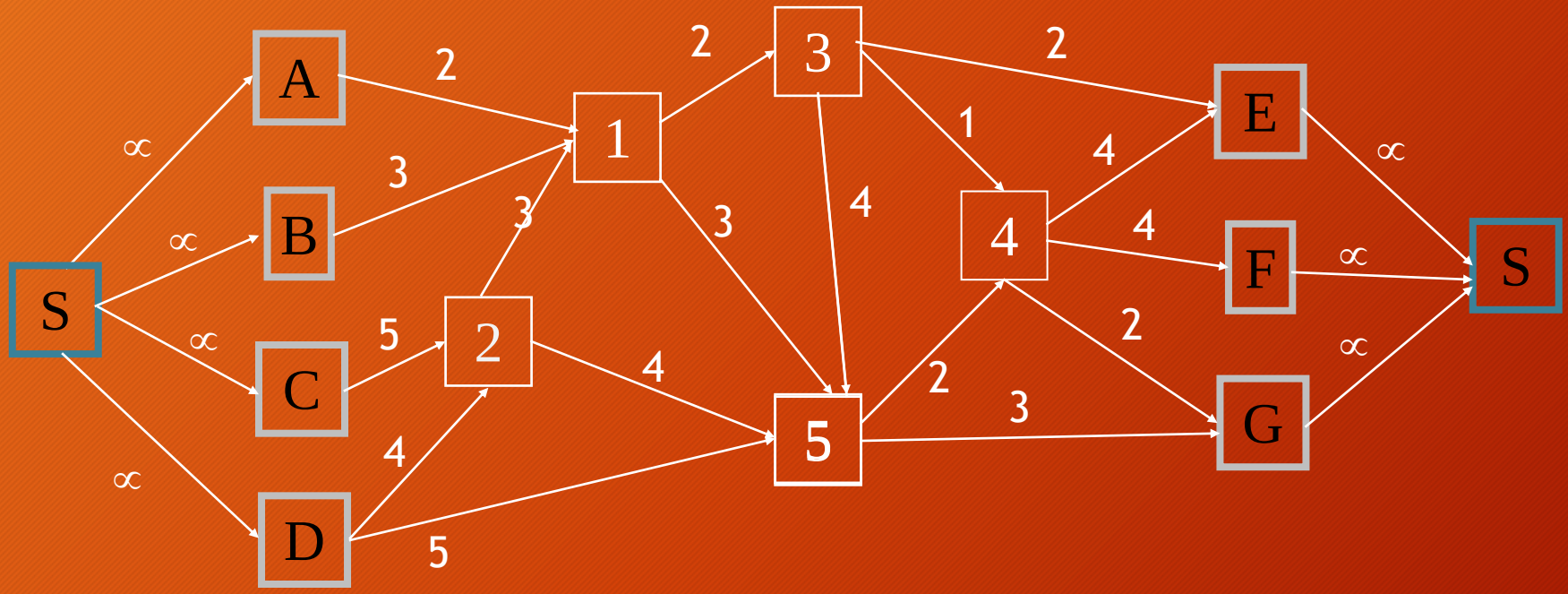
Problem z metodą Forda-Fulkersona



Algorytm Edmonsona-Karpa

- Metoda Forda-Fulkersona + wyszukiwanie ścieżek przeszukiwaniem wszerz

Problem maksymalnego przepływu z wieloma źródłami



Algorytmy i struktury danych

Odc. 11. Ścieżka i cykl Eulera. Problemy „trudne”. Cykl Hamiltona w grafie.

Cykl Eulera

Problem / definicje

Ścieżka Eulera

- *Ścieżka przebiegająca przez wszystkie krawędzie grafu, ale przez każdą z nich tylko raz*
- *Oznacza to, ze przez ten sam wierzchołek można przechodzić wiele razy*

Cykl Eulera

- *Cykliczna ścieżka Eulera*

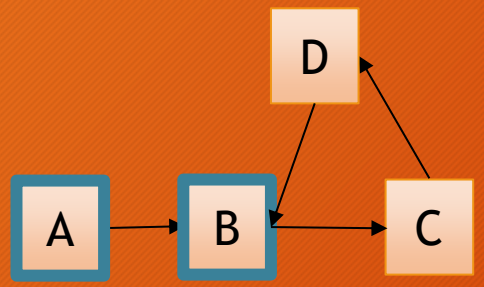
Kiedy w grafie jest ścieżka/cykl Eulera?

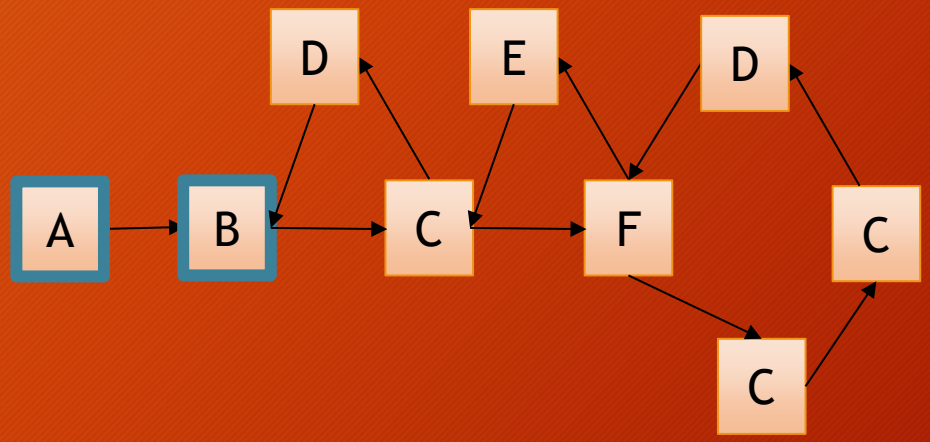
- Intuicja: *aby istniał cykl Eulera w żadnym węźle nie możemy „utknąć” w sytuacji, w której wszystkie krawędzie wychodzące zostały już odwiedzone, chyba że właśnie przeszliśmy ostatnią nie odwiedzoną krawędź*
- *To oznacza, że:* w grafie nieskierowanym: kiedy liczba węzłów o nieparzystym stopniu (liczbie następników) wynosi zero (istnieje cykl Eulera) lub 2 (istnieje ścieżka Eulera) – węzły o nieparzystym stopniu są odpowiednio początkiem i końcem ścieżki
- W grafie skierowanym: w każdym węźle jest tyle samo krawędzi wchodzących, co wychodzących, a liczba tych, w których różnica ta wynosi 1 jest równa dwa

Kiedy w grafie jest ścieżka/cykl Eulera? (cd.)

- *W* grafie nieskierowanym: kiedy liczba węzłów o nieparzystym stopniu (liczbie następników) wynosi zero (istnieje cykl Eulera) lub dwa (istnieje ścieżka Eulera) – węzły o nieparzystym stopniu są odpowiednio początkiem i końcem ścieżki
- W grafie skierowanym: w każdym węźle jest tyle samo krawędzi wchodzących, co wychodzących (cykl Eulera) lub
  - i jeden mający o jedną krawędź wychodzącą więcej niż wchodzących (początek ścieżki Eulera)
  - istnieje jeden węzeł mający jedną krawędź wchodzącą więcej niż wychodzącą (koniec ścieżki E.)

Ilustracja





Początek/koniec

Algorytmy znajdowania ścieżek/cyklu Eulera

- Algorytm Flerry’ego
- Algorytm Hierholzera

Przypomnienie algorytmów pomocniczych

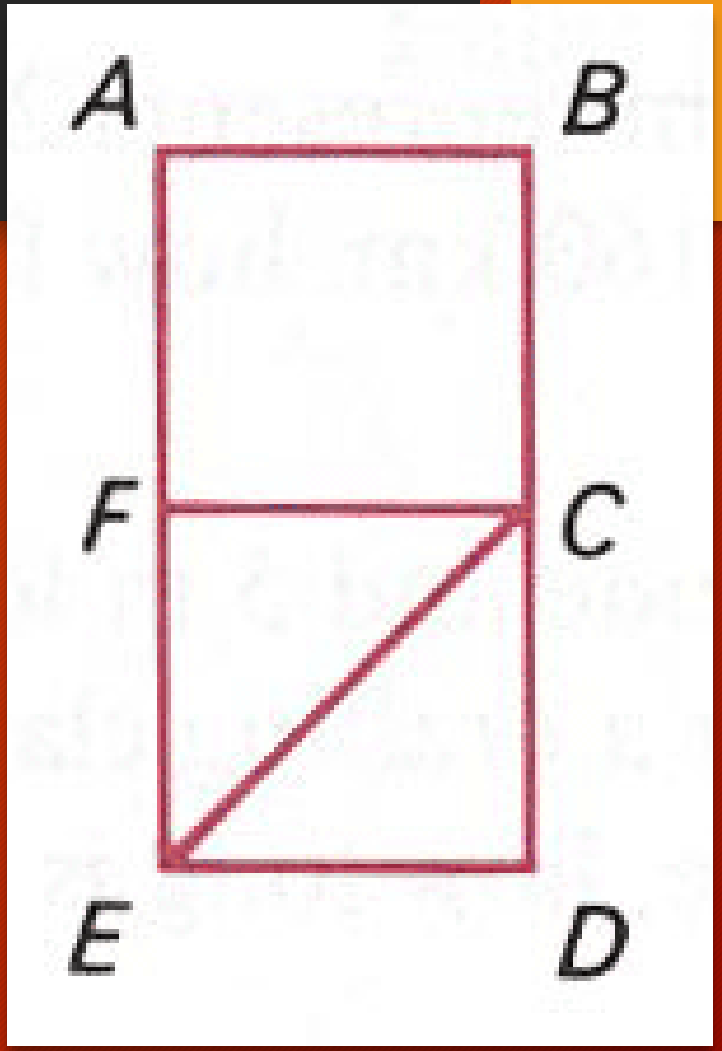
- Przeglądanie grafu w głąb

Algorytm Hierholzer’a O(|E|)

- Przygotowanie – sprawdzamy liczby krawędzi wchodzących i wychodzących z każdego węzła (najwygodniej reprezentować graf listą incydencji), ustalamy węzeł początkowy i końcowy (w przypadku ścieżki)

1. Zacznij od wybranego wierzchołka *v* i dodawaj do ścieżki krawędzie tak długo, aż trafisz z powrotem do *v* (masz już pewien cykl) [obserwacja: takie działanie musi do tego doprowadzić, z powodu spełnienia warunków istnienia cyklu Eulera]
2. Z każdego wierzchołka *u* należącego do ścieżki, z którego jest jakaś krawędź nie włączona jeszcze do ścieżki Eulera wystartuj analogiczny spacer jak w 1 aż wrócisz do *u*.

Przykład Alg. Hierholzera



Algorytm Fleury'ego (dla cyklu)

1. Zacznij od dowolnego wezła *v* (cykl E.)
2. Wybierz dowolną krawędź *v* wyłączając mosty (krawędź, której usunięcie spowoduje niespójność grafu) i przejdź do następnego węzła *v'* dostępnego z *v*
3. Usuń (zatruj) z grafu krawędź, którą właśnie przeszedłeś
4. Dodaj usuniętą krawędź do ścieżki/cyklu E
5. Kontynuuj 2 od węzła, do którego przeszedłeś

Złożoność! $O(E^2)$ Dlaczego? Każdą krawędź dodawaną do ścieżki E. sprawdzamy czy nie jest mostem…

?

Dlaczego ścieżkę Eulera łatwo znaleźć?

Ścieżka Hamiltona

- Cykliczna ścieżka przez wszystkie wierzchołki, każdy wierzchołek występuje tylko raz

?

Dlaczego problem znalezienia cyklu Hamiltora jest b. trudny?

Wykład

Odc. 7

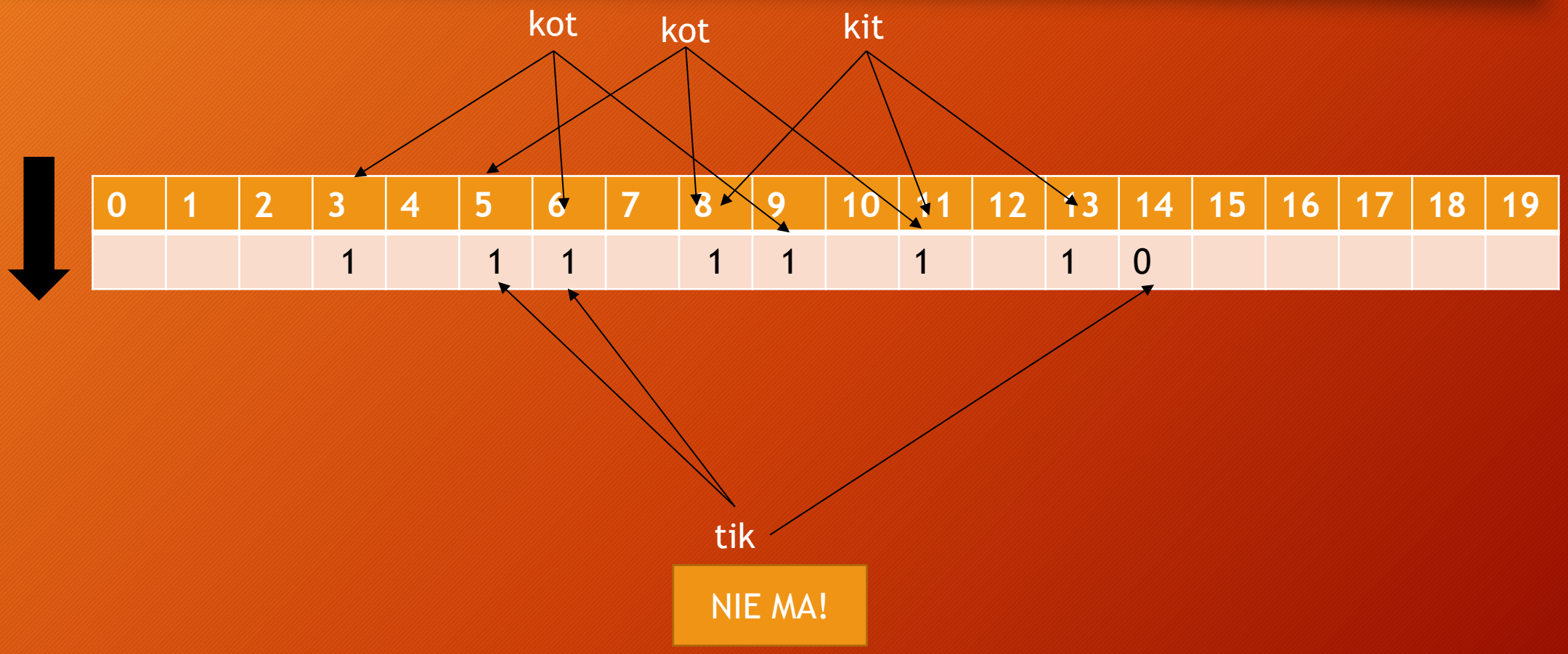
Intermezzo (przy kolokwium). Filtr Bloma, podstawy obliczeń równoległych.

Filtr Blooma

Jak efektywnie wykluczyć, że w naszym zbiorze / strukturze danych jest dana o określonej wartości (k) → filtr Blooma

- Dana jest tablica bitów o długości N i indeksach ze zbioru 0… *N – 1*
- Dane jak *k* funkcji haszujących *mod N*
- *Każdy klucz hashujemy k krotnie -> otrzymujemy nr bitów, które zapalamy*

Ilustracja

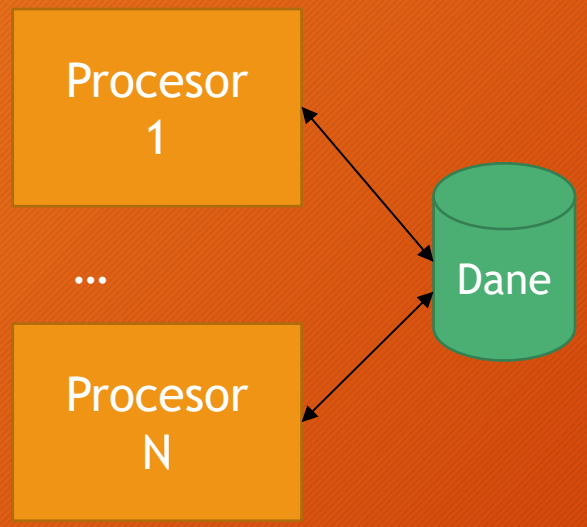

kot
kot
kit
0 1 2 3 4 5 6 7 8 9 10 11 12 13 14 15 16 17 18 19
1 1 1 1 1 1 0
tik
NIE MA!

Obliczenia i algorytmy równoległe

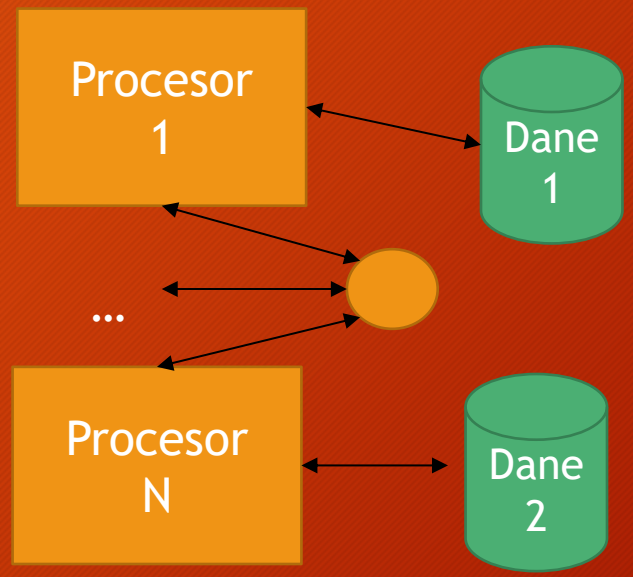
Uwaga wstępna i zasadnicza

1) Wszystkie współczesne systemy są równoległe
2) I to na bardzo wielu poziomach
3) Wiele: rdzenie, procesorów, procesów (programów), systemów

Modele obliczeń równoległych



MISD = Multiple Instruction Single Data

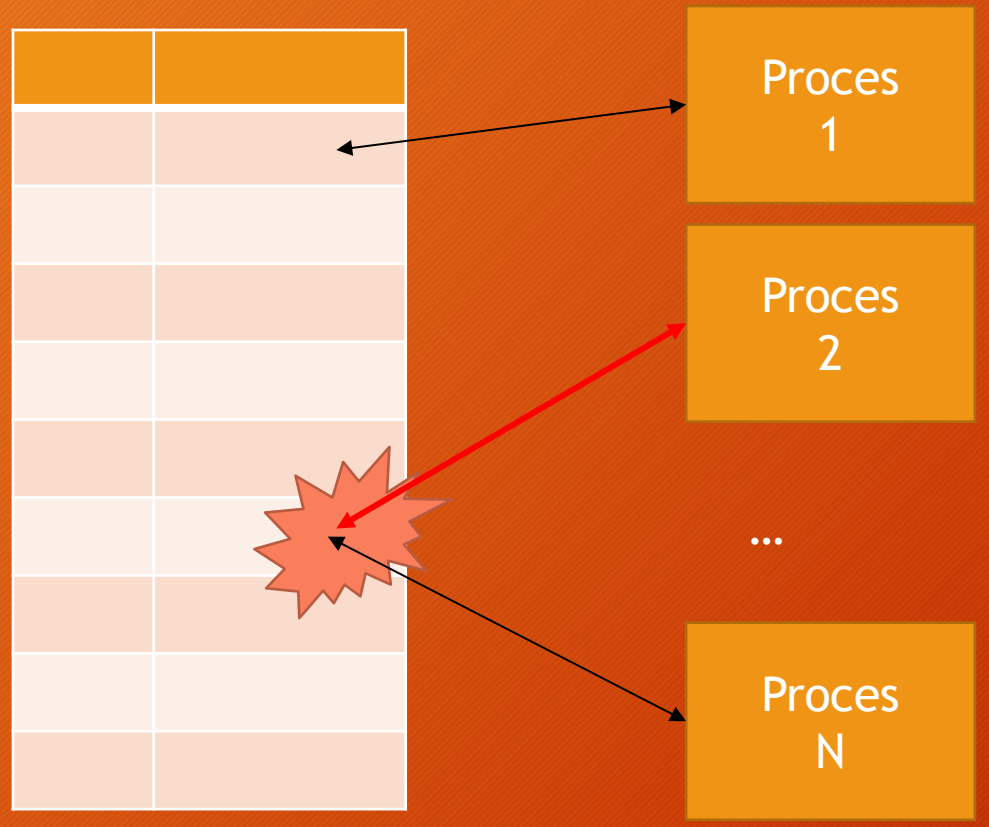


MIMD = Multiple Instruction Multiple Data

Problemy obliczeń równoległych

- Współdzielenie danych
- Komunikacja
- Koordynacja i synchronizacja obliczeń

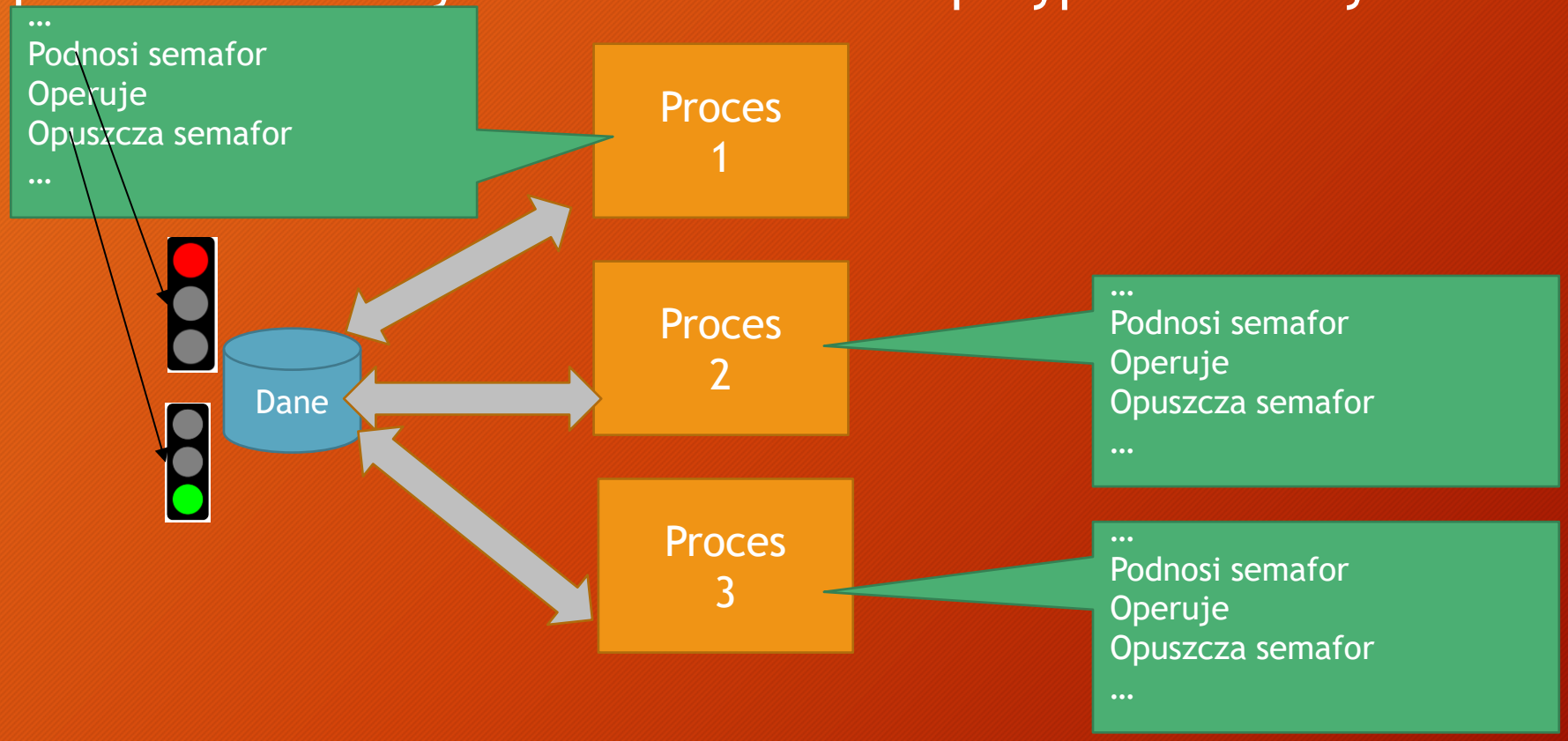
Współdzielenie (praca na wspólnych) danych



- Co tu się może stać?
  - Niechciane nadpisanie, nadpisanie przez kilka procesów → zapamiętanie przypadkowych danych (z ostatniego piszącego)
  - Zmiana części danych przez inny proces w trakcie przetwarzania (przypadkowy/zniekształcony wynik w trakcie przetwarzania, obliczenia na niespójnych danych)

Wykluczanie jednoczesnego dostępu

- Współdzielenie danych dobrze działa w przypadku odczytu…



Zakleszczenie

Proces 1

Zablokuj zasób 1

Operuj

Zablokuj zasób 2

Operuj

Odblokuj zasób 2

Operuj

Odbloku zasób 1

Proces 2

Zablokuj zasób 2

Operuj

Zablokuj zasób 1

Operuj

Odblokuj zasób 2

Operuj

Odbloku zasób 1

Tu albo proces 1 będzie czekał na zwolnienie zasobu 2 zablokowawzy zasób 1 a proces 2 będzie czekał na zwolnienie zasobu 1 co będzie niemożliwe, bo zasób 1 jest zajęty przez proces pierwszy

Lub

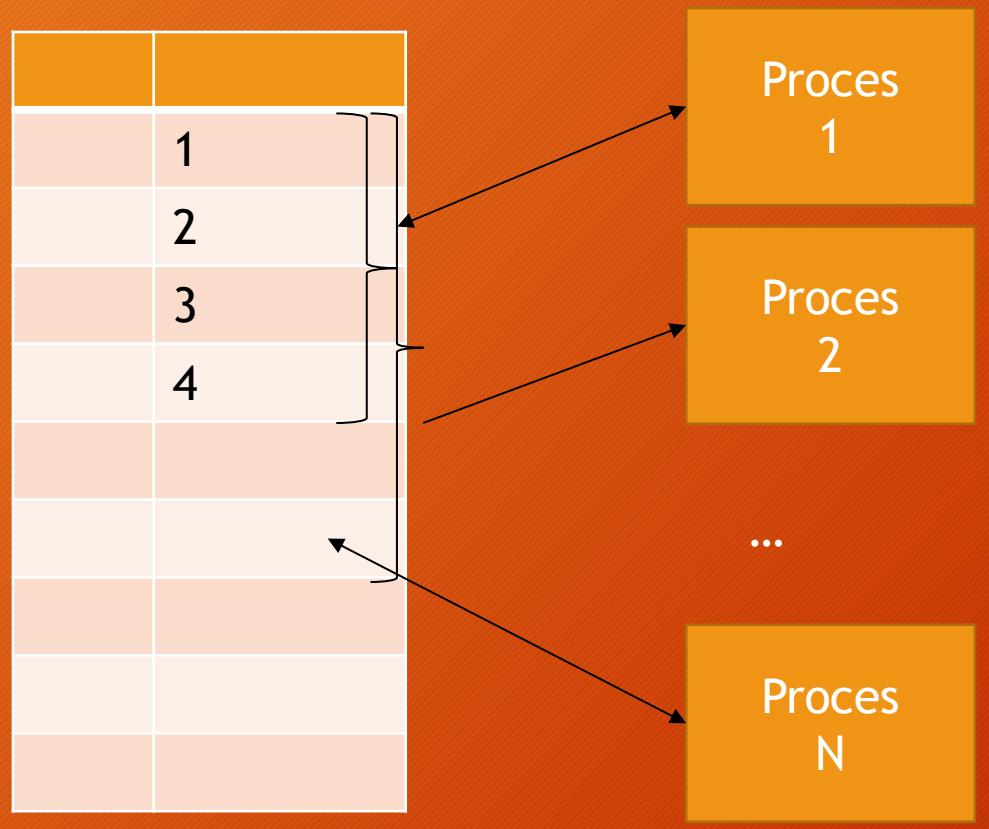
Symetrycznie dla drugiego procesu

Dining philosophers (ucztujący filozofowie)

- Czy też problem jedzących Azjatów…
- Do jedzenia potrzeba dwóch sztućców…
- Jeśli każdy weźmie widelec z lewej to finalnie dojdzie do zakleszczenia

Ilustracja problemu

Jakie dane widzą inne procesy?



Komunikacja synchroniczna

Proces 1

- …
- Wyślij-dane(D, Proces 2)
- Zaczekaj na odbiór
- [dalsze obliczenia]

Proces 2

- …
- Zaczekaj-na-dane(Proces1)
- Odbierz-dane
- [dalsze obliczenia]

Procesy 1 i 2 czekają na siebie, ich dalsze obliczenia są wstrzymywane

Komunikacja asynchroniczna

Proces 1

- …
- Wstaw-dane-do-kolejki-i-kontynuuj
- [dalsze obliczenia]

Proces 2

- … [tu najczęściej jest początek procesu]
- Odczytaj daną z kolejki
- [dalsze obliczenia]

Procesy 1 i 2 nie czekają na siebie.

Synchronizacja obliczeń

- Analogicznie, jak przy komunikacji synchronicznej (tzw. mechanizm spotkań) – proces zatrzymuje się w oczekiwaniu na inny proces

Zrównoleglenie algorytmu

- Np. suma macierzy.
- Załóżmy, że mamy funkcję suma-wiersza(i), która sumuje elementy i-tego wiersz macierzy A i B ze sobą (suma macierzy)
- Równoległa wersja algorytmu
- *suma-macierzy*(A, B) [A, B, w pamięci wspólnej, A + B = C]
  - For i=1 to N
    - suma-wiersza(i) [każde wywołanie powołuje oddzielny proces obliczeniowy]
  - Poczekaj, aż skończą się obliczenia w oddzielnych procesach

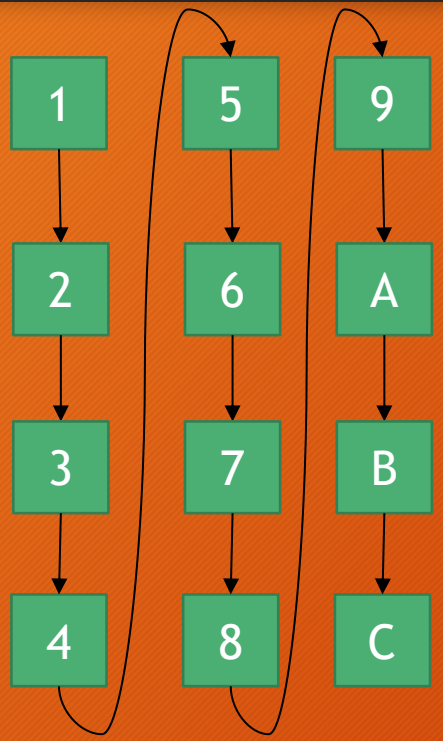
Algorytm dobrze zrównoleglający się

- Algorytm Belmana-Forda (!)
- Zbiór wyników – tablica $d(i)$, i=1…N (węzły), wejście w_{ij} – odległości
- Niech dana jest funkcja relax(), która relaksuje wszystkie krawędzie grafu, aktualizując $d(i)$
- Algorytm BF
  - Odpal N-1 relax() w oddzielnych procesach (ze wspólnymi danymi d i w)
  - Zaczekaj na koniec wszystkich procesów

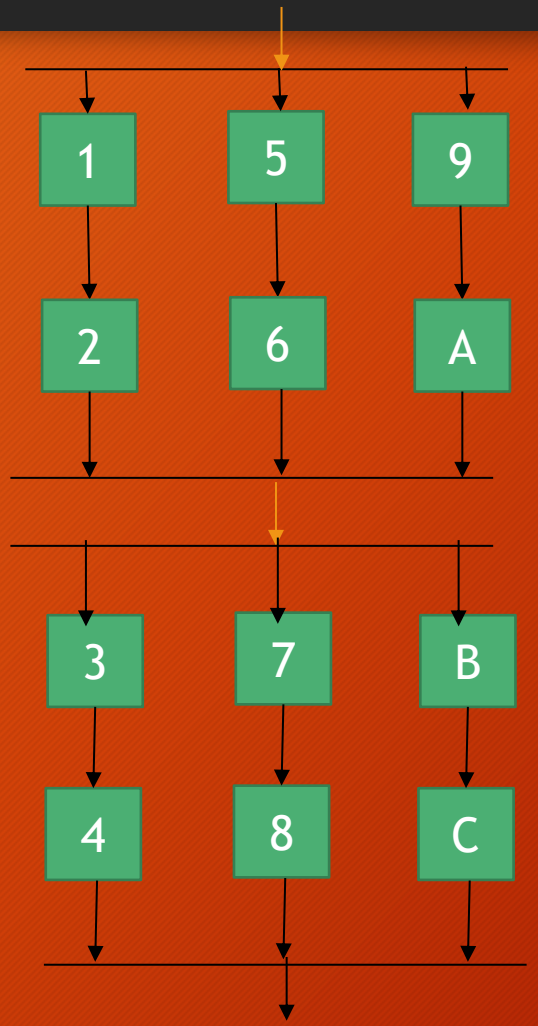
Problemy trudne do zrównoleglenia

- Generalnie: te, które wymagają sekwencyjnego działania, w których dalsze przetwarzanie zależy od wyników uzyskanych wcześniej
- Przykład: generowanie kolejnych wyrazów ciągów zdanych rekurencyjnie (równaniem różnicowym)
  - X[0]…X[N – 1] dane wyjściowe
  - X[k] = f(x[k–1], x[k-2], …, x[k-N])

Koordynacja obliczeń



Proces sekwencyjny



Proces zrównoleglony
(możliwy wariant)

„*Do wyniesienia*” z wykładu nt. równoległości

Aneks do wykładu o haszowaniu

Haszowanie uniwersalne i doskonałe

Haszowanie uniwersalne(***)

- Sytuacja: mamy haszowanie z łańcuchowym rozwiązywaniem kolizji
- Problem – złośliwy przeciwnik może nauczyć się funkcji haszującej i złośliwe podawać wartości
- Inaczej – mamy zbiór wartości, które chcemy wprowadzić do tablicy haszującej

Haszowanie uniwersalne(***)

- Rozwiązanie: losujemy funkcję haszującą ze zdefiniowanej rodziny funkcji
  - **Zapewniając, ze funkcja haszująca jest losowana z pewnego „dobrego” zbioru funkcji gwarantujemy, że średni czas ciągu *n* operacji jest liniowy względem *n*, tzn. czas pojedynczej operacji jest stały**
- Haszowanie uniwersalne
- „Dobry” zbiór funkcji – rodzina uniwersalna funkcji haszujących

Haszowanie uniwersalne

- Rodzina uniwersalna funkcji haszujących
  - rodzina $H^* = \{ h: K \rightarrow I \}$, że
    - liczba funkcji h w H^* odwzorowujących dowolną parę różnych kluczy p i q w ten sam indeks tj. takich, że $H(p)=H(q)$ jest nie większa niż $|H^*|/m$, $m=|I|$ (!)
      - uwaga H^* liczniejsze od m (!)
    - mamy zatem do wyboru: $|H^*|$ funkcji, z czego $|H^*|/m$ dają kolizję, zatem, jeśli losowo wybieramy jedną H, to:
    - prawd. kolizji między k a l wynosi $1/m$
  - w konsekwencji: czas wykonania serii wstawień i usunięć jest liniowy

Rodzina uniwersalna funkcji haszujących

- Niech p – liczba pierwsza większa od największego „haszowanego” klucza
- niech $a \in \{0, 1, \ldots, p - 1\}$, $b \in \{1, 2, \ldots, p - 1\}$
- $h_{a,b}(k) = ((a*k+b) \bmod p) \bmod m$, $p>m$
- ciekawe: m dowolna liczba, nie koniecznie pierwsza (m – liczba indeksów w tablicy)
- $H_{a,b,} = \{h_{a,\,b}\}$ – jest rodziną uniwersalną (Cormen, str. 234, 235), $|\{h_{a,\,b}\}| = p*(p - 1)$

Haszowanie doskonałe

- Cel:
  - dany jest statyczny zbiór danych. Należy zdefiniować sposób wyszukiwania kluczy w czasie stałym.
- Def. Haszowanie jest doskonałe, jeśli w pesymistycznym przypadku wymaga stałej liczby odwołań do tablicy
- Rozwiązanie:
  - analog do rozwiązywania kolizji łańcuchowo:
    - zamiast tworzyć listę elementów odwzorowywanych na dany indeks *i* tworzymy dodatkowe tablice z haszowaniem, starannie dobierając funkcje H_j

Haszowanie doskonałe – konstrukcja rozwiązania

- I poziom – rozrzucamy m kluczy w n miejsc za pomocą funkcji haszującej *h* „starannie” wybranej z uniwersalnej rodziny funkcji haszujących
- II poziom – n_j – liczba kluczy *k*, dla których $H(k)=j$;
  - budujemy dodatkową tablicę haszującą dla poszcz. *j*, gdzie $m_j = n_j^2$
  - stosujemy starannie dobraną funkcje haszującą h_j

Haszowanie doskonałe

- Lemat: jeśli n kluczy zapamiętujemy w tablicy o n^2 elementów za pomocą funkcji haszującej losowo wybranej z rodziny uniwersalnej funkcji haszujących to prawdopodobieństwo jakiejkolwiek kolizji jest mniejsze niż ½.
- Dowód: wartość oczekiwana liczby kolizji: $newton(n, 2) \ / \ n^2 < ½$
  - W haszowaniu uniwersalnym w tablicy o n^2 elementach prawdopodobieństwo kolizji na parze różnych kluczy p i q wynosi $1/n^2$
  - Pr(X>t) <= E[X]/t /wykorzystano nierówność Markowa, dla t=1/

Haszowanie doskonałe

- Własność ta pozwala dla małych zbiorów pozwala znaleźć funkcję haszującą na tablicy o wymiarze n^2, dla większych wymaga haszowania 2 poziomowego
- Znalezienie funkcji haszującej bez kolizji wymaga „kilku prób” – funkcję losujemy! /tak jak przy rzucie monetą wyrzucenie orła/

Haszowanie doskonałe – dobór funkcji

- m = n – liczba kluczy
- h_1 – wybieramy z rodziny: $H_{p,\ m}$, gdzie *p* liczba pierwsza większa od dowolnej wartości klucza
- h_j – wybieramy z rodziny $H_{p,\ m_j}$, gdzie m_j – kwadrat liczby kluczy kolidujących na indeksie *j*
  - *Dlaczego? Por. poprzednie slajdy*

Haszowanie doskonałe

- Co z pamięcią?
  - okazuje się, że jeśli m=n, to wart. oczekiwana sumy długości tablic drugiego poziomu nie jest większa niż 2n
  - prawdopodobieństwo, że tablice 2 poziomu zajmą więcej niż $4n<1/2$

KONIEC ☺

Algorytmy i struktury danych

Odc. 9. Programowanie dynamiczne. Najkrótsze ścieżki między wszystkimi węzłami w grafie. Najdłuższy wspólny podciąg

Problem najdłuższego wspólnego podciągu

Najdłuższy wspólny podciąg

- Niech $X=\{x_1, \ldots, x_M\}$ – dany ciąg
- Def. Ciąg $Z=\{z_1, \ldots, z_K\}$ jest podciągiem X, jeśli istnieje taki ściśle rosnący ciąg indeksów, $\{i_1, \ldots, i_K\}$, że $z_{ij}=x_j$.
- Np. X = ALAMAKOTA, Z = LMKT /podciąg,ciąg indeksów {2, 4, 6, 8}/

Sformułowanie zadania

- Zadanie:
- Dane są dwa ciągi:
  - $X=\{x_1, \ldots, x_M\}$, $Y=\{y_1, \ldots, y_N\}$
- Należy znaleźć taki ciąg $Z=\{z_1, \ldots, z_K\}$, że jest on najdłuższym podciągiem ciągu X i Y, tj. K – największe z możliwych /NWP/
- Wskaźnik oceny rozwiązania (*F*) – długość ciągu (liczba elementów)

Zależności „belmanowskie”

- Niech:
  - X_i – ciąg składający się z pierwszych *i* wyrazów ciągu *X*; analogicznie Y_i;
  - $Z = \{z_1, \ldots, z_K\}$ – najdłuższy wspólny podciąg *X* i *Y*; analogicznie Z_i
- Zachodzą następujące właściwości:
  - $X_M = \{x_1, \ldots, x_M\}$,
  - $Y_N = \{y_1, \ldots, y_N\}$
  - $Z = \{z_1, \ldots, z_K\}$

1) jeśli $x_M = y_N$, to Z_{K-1} jest *NWP* X_{M-1} i Y_{N-1}
2) jeśli $x_M <> y_N$ *i* $x_M <> z_K$, to *Z* jest *NWP* (X_{M-1} , *Y*)
3) *jeśli* $x_M <> y_N$ *i* $y_N <> z_K$, to Z jest *NWP* (*X*, Y_{N-1})

Ilustracja

- Przypadek I
  - X={ALABAM}, Y={BALALAM}, Z={ALAAM}
  - X’={ALABA}, Y’={BALALA}, Z’={ALAA}
- Przypadek 2.
  - X={ALABAM}, Y={BALALA}, Z={ALAA}
  - X’={ALABA}, Y={BALALA}, Z={ALAA}
- Przypadek 3 /symetrycznie/

Rekurencyjna zależność na długość NWP

- $c[i, j]$ długość NWP ciągów X_i, Y_j
- Zależność:
  - c[i,j]=0 dla i=0 lub j=0
  - c[i,j]=c[i-1, j-1]+1 jeśli i,j>0, $x_i=y_j$ (*)
  - c[i,j]=max ($c[i, j-1]$ /a/, $c[i-1, j]$ /b/), jeśli $i,j>0$, $x_i<>y_j$ (**)

Algorytm - szkic

- $c[i, 0]=0$, $c[0,j]=0$, $i=1\ldots M$, $j=1\ldots N$
- Wyznaczamy długość najdł. wsp. podciągów ciągów
  - X_1 i $Y_1, \ldots, Y_N$
  - X_2 i $Y_1, \ldots, Y_N$
  - …
  - X_M i $Y_1, \ldots, Y_N$
  - korzystając z zależności rekurencyjnej