JAVA24 Gdańsk

Programowanie I

Spotkanie #1

Programowanie I
poziom podstawowy

Rozkład jazdy

- wyrażenia regularne
- pseudokod
- algorytmika
- złożoność obliczeniowa

Wyrażenia regularne

Wyrażenia regularne

Wyrażenia regularne
= Regular Expressions
= Regex

Wyrażenia regularne pozwalają opisywać wzorce tekstu (określać wzór pasujących łańcuchów).
Stosowane są głównie przy badaniu oraz modyfikowaniu tekstu.

Wyrażenie regularne opisujące adres IP (v4):

```
/^(?:(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)\.){3}(?:25[0-5]|2[0-4][0-9]|[01]?[0-9][0-9]?)$/
```

Wyrażenia regularne

składnia - znaki

. – dowolny znak
\d – cyfra [0-9]
\D – inny znak niż cyfry [^0-9]
\s – znak biały [\t\n..]
\S – inny znak niż znak biały [^\s]
\w – znak [a-zA-Z_0-9]
\W – inny znak niż \w [^\w]

Wyrażenia regularne

składnia - kwantyfikatory

? – 0 lub 1 wystąpienie

* – 0 lub więcej wystąpień

+ – 1 lub więcej wystąpień

{n} – dokładnie n razy

{n,} – przynajmniej n razy

{n,m} – przynajmniej n lecz nie więcej niż m razy

Wyrażenia regularne

składnia - meta-znaki

\ – wskazanie, że meta-znak ma się stać zwykłym znakiem (\., *, \/, \?, \:, \., \^, \+, \\, \=, \|)

^ – oznaczenie początku wzorca

$ – oznaczenie końca wzorca

| – alternatywa

(...) – grupowanie znaków

[...] – dowolny ze zbioru znaków

[^...] - dowolny z niewymienionych znaków

Wyrażenia regularne

java.util.regex.*

java.util.regex.Pattern - klasa odpowiedzialna za definiowanie wyrażeń

java.util.regex.Matcher - klasa odpowiedzialna za dopasowanie wzorca wyrażeń do tekstu

String::matches(String regex)
String::split(String regex)
String::replaceAll(String regex, String replacement)
String::replaceFirst(String regex, String replacement)

Wyrażenia regularne

narzędzia online

Testowanie online:
https://regexr.com
https://regex101.com

Tutorial:
https://regexone.com

Przykłady kodu / zadania

https://github.com/softwaredevelepmentacademy/java24gda_pro1

Zadania

#regex

Zadania

#regex

1. Napisz metody, które sprawdzą, czy podany ciąg znaków jest:
   a. poprawnym kodem pocztowym
   b. poprawnym adresem strony internetowej (przyjmijmy format: http(s)://example.com)
   c. poprawną datą dla formatu dd.mm.yyyy

2. Napisz metodę, która cenzurować będzie wybrane słowa w dowolnym tekście.

3. Napisz metodę sumującą cyfry w tekście podanym przez użytkownika.

Pamiętaj o wykorzystaniu repozytorium kodu Git!

Pseudokod

Pseudokod

```
uruchom_rzutnik()
    jezeli (RZUTNIK_PODLACZONY_DO_PRADU)
        jezeli (RZUTNIK_PODLACZONY_DO_KOMPUTERA)
            WLACZ_RZUTNIK
            KONIEC
        inaczej
            PODLACZ_DO_KOMPUTERA
            WROC_DO_POPRZEDNIEGO_ETAPU
    inaczej
        PODLACZ_DO_PRADU
        WROC_DO_POPRZEDNIEGO_ETAPU
```

Pseudokod

1. Każda instrukcja zaczyna się od nowej linii
2. Przepływ z góry na dół
3. Należy przestrzegać wcięć i czytelnego rozłożenia tekstu
4. Nie ma jednego ustalonego zapisu, jednak należy trzymać się przyjętej konwencji w całym zapisie algorytmu
5. Musi być czytelny i zrozumiały dla osób nie znających konkretnego języka programowania
6. Rozpoczynamy nazwą algorytmu, w nawiasach możemy podać argumenty

Pseudokod

Przypisanie
```
a ← b + c
```

Porównanie
```
a = b, a > b, a < b, a >= b, a <= b, a != b
```

Wypisanie / pobieranie
```
wypisz a, pobierz a
```

Instrukcja warunkowa
```
if a > 0
    then wypisz instrukcje na tak
    else wypisz instrukcje na nie
```

Pętla for
```
for i ← 1 to 10 do
    instrukcje
```

Pętla while
```
while i < 10 do
    instrukcje
    i ← i + krok
```

Operatory specjalne

a mod b	reszta z dzielenia
a > 0 and b = 0	logiczne „i” (koniunkcja)
a > 0 or b = 0	logiczne „lub” (alternatywa)

Inne

TRUE / FALSE	logiczna wartość – prawda / fałsz
return	zwrócenie wartości
break	przerwanie wykonywania pętli
continue	przejście do kolejnego obrotu pętli

Zadania

Napisz pseudokod:

1. wczytujący dwie liczby i wypisujący ich sumę
2. liczący silnię podanej liczby
3. sprawdzający, czy podana liczba jest pierwsza
4. opisujący procedurę zakupu w sklepie Internetowym

Algorytm

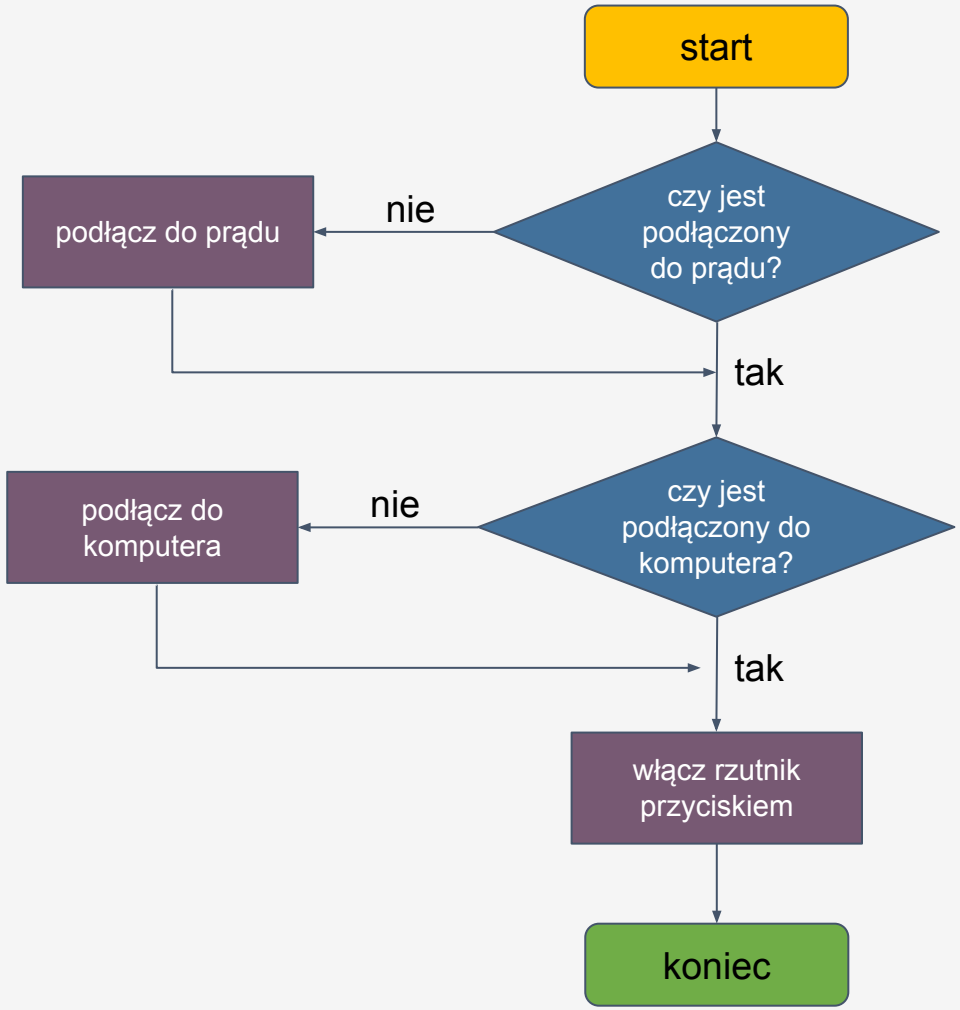
to co to właściwie jest?

Algorytm

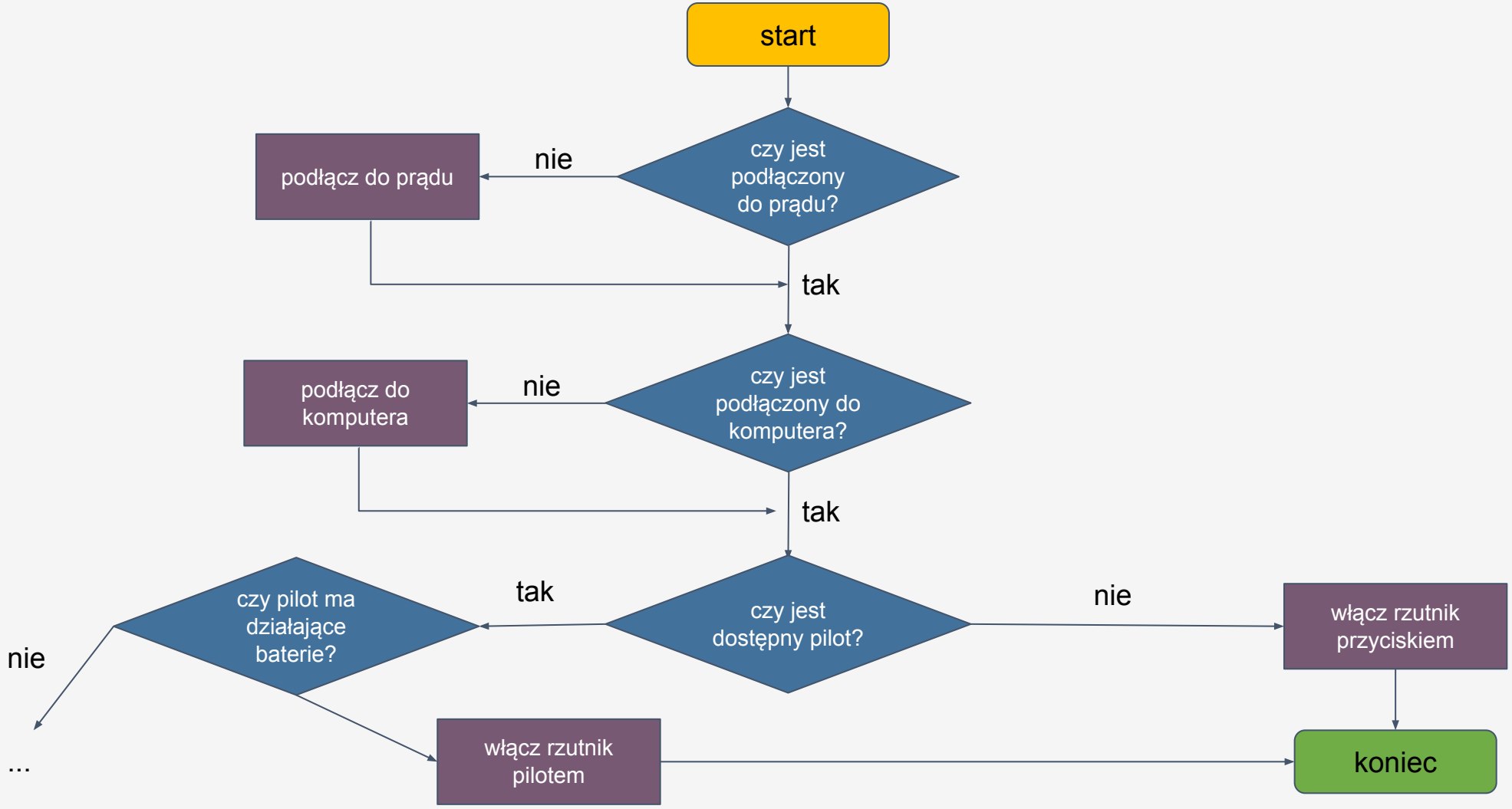
Skończony ciąg jasno zdefiniowanych czynności, koniecznych do wykonania pewnego rodzaju zadań. Sposób postępowania prowadzący do rozwiązania problemu - @wiki

Zadaniem algorytmu jest przeprowadzenie systemu z pewnego stanu początkowego do pożądanego stanu końcowego.

Algorytm - uruchamianie i podłączanie rzutnika



Algorytm - uruchamianie i podłączanie rzutnika



Schemat blokowy

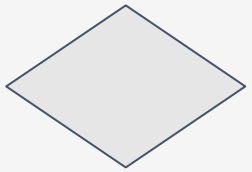
Schemat blokowy

1. Sformalizowany zapis
2. Częściami składowymi schematów blokowych są proste figury geometryczne, np. prostokąt, romb, koło, równoległobok
3. Przepływ z góry na dół
4. Wystarczy kartka i coś do pisania
5. Może występować jako dokumentacja techniczna
6. Nie wymaga znajomości konkretnego języka programowania
7. Należy unikać rysowania przecinających się ścieżek sterowania

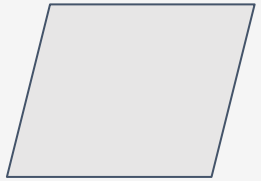
Schemat blokowy

Oznacza początek lub koniec algorytmu. W każdym algorytmie musi się znaleźć dokładnie jedna taka figura z napisem "Start" oraz "Koniec"

Oznacza proces. Umieszczamy tu np. obliczenia, podstawienia.



Oznacza blok decyzyjny. Umieszcza się w nim warunek. Wychodzą dwie strzałki. Jedna z etykietą „Tak”, druga „Nie”.



Używany do instrukcji wejścia / wyjścia danych, np. czytaj x, drukuj x na ekranie.

Oznacza proces który jest już zdefiniowany wcześniej i nie podlega opisowi w tym algorytmie.

Oznacza łącznik stronicowy.

Zadania - lucidchart.com

Zbuduj schemat blokowy dla algorytmu:

1. który pozwoli przygotować Ci Twoją ulubioną potrawę
2. liczącego silnię podanej liczby
3. wczytującego liczby z wejścia aż do chwili gdy podana liczba jest równa zero, a następnie wyświetlającego sumę i średnią arytmetyczną tych liczb

Algorytmika

- nauka o algorytmach
- występuje w informatyce, cybernetyce, matematyce, naukach przyrodniczych i wielu innych

Przykładowe typy algorytmów:

- algorytmy genetyczne - są w stanie same się rozwijać i uczyć na podstawie własnych błędów
- algorytmy sztucznej inteligencji - uczą się jak działać lepiej, poprawiają siebie same
- algorytmy wykorzystujące uczenie maszynowe - potrafią wyciągać wnioski na podstawie dostarczonych danych

Algorytmika - po co?

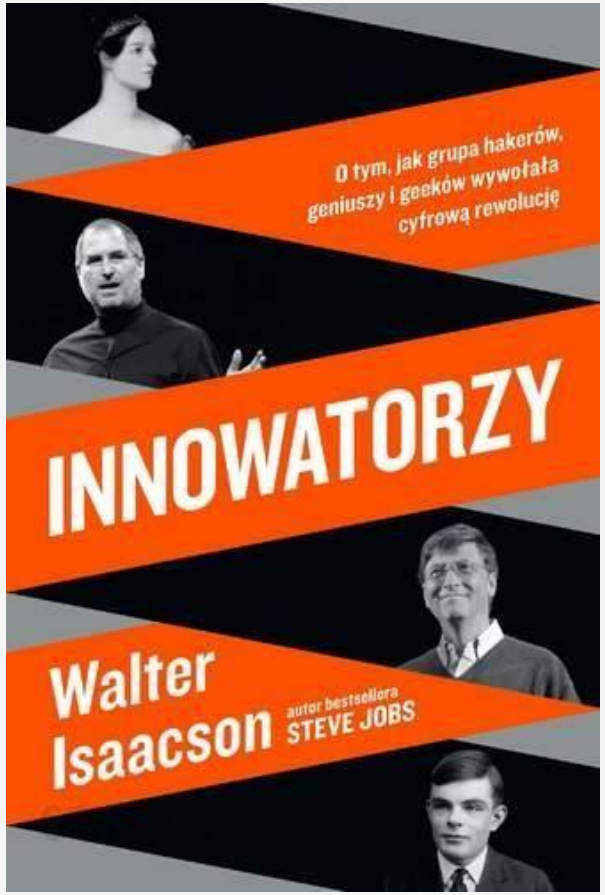
- ułatwia pracę - dzieli ją na etapy
- pomaga zrozumieć sposób "myślenia" komputera
- uczy tworzyć perfekcyjny kod
- pozwala wyznaczać cele
- pozwala opracować skrypt

Turing

- angielski matematyk i kryptolog
- twórca "bomby Turinga" (Enigma)
- uważany za ojca informatyki i sztucznej inteligencji
- twórca Maszyny Turinga (abstrakcyjny model komputera służący do rozwiązywania algorytmów) - większość dzisiejszych komputerów wykorzystuje ten model
- "Test Turinga"

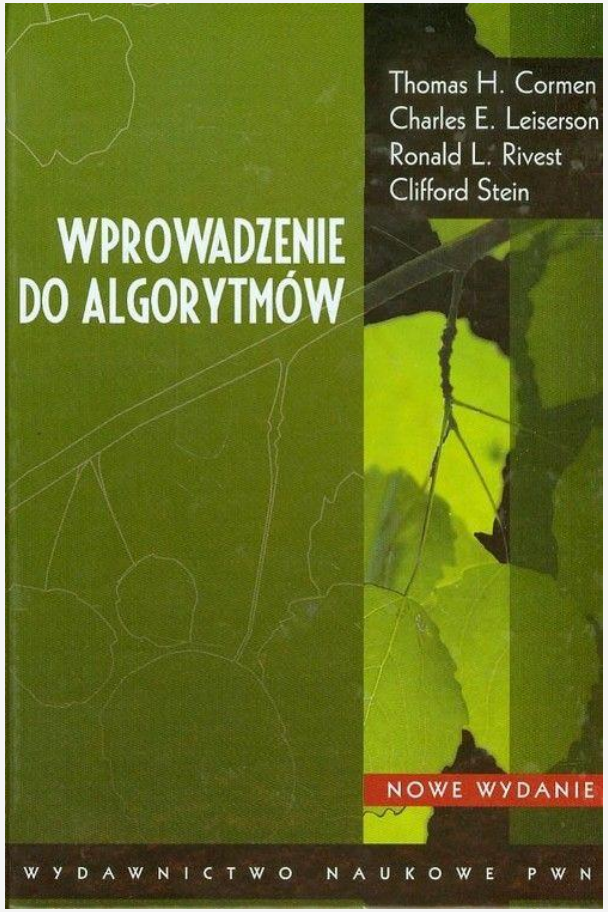
Przy zwłokach Turinga znaleziono nadgryzione jabłko i słój z cyjankiem potasu!
"Gra Tajemnic"

"Innowatorzy" - Walter Isaacson



https://audioteka.com/pl/audiobook/innowatorzy

"Wprowadzenie do algorytmów"



https://www.ceneo.pl/16684097

Klasyfikacje algorytmów

Klasyfikacje algorytmów

- **dziel i zwyciężaj** - dzielimy problem na kilka mniejszych, a te znowu dzielimy, aż ich rozwiązania staną się oczywiste

- **programowanie dynamiczne** - problem dzielony jest na kilka, ważność każdego z nich jest oceniana i po pewnym wnioskowaniu wyniki analizy niektórych prostszych zagadnień wykorzystuje się do rozwiązania głównego problemu

- **metody zachłanne** - nie analizujemy podproblemów dokładnie, tylko wybieramy najbardziej obiecującą w danym momencie drogę rozwiązania

Klasyfikacje algorytmów

- **programowanie liniowe** - oceniamy rozwiązanie problemu przez pewną funkcję jakości i szukamy jej minimum

- **wyszukiwanie wyczerpujące** - przeszukujemy zbiór danych, aż do momentu znalezienia rozwiązania

- **heurystyka** - człowiek na podstawie swojego doświadczenia tworzy algorytm, który działa w najbardziej prawdopodobnych warunkach, rozwiązanie zawsze jest przybliżone

Techniki implementacji

Techniki implementacji

- **proceduralność** – algorytm dzielimy na szereg podstawowych procedur, wiele algorytmów współdzieli wspólne biblioteki standardowych procedur, z których są one wywoływane w razie potrzeby

- **praca sekwencyjna** – wykonywanie poszczególnych procedur algorytmu, według kolejności ich wywołań, naraz pracuje tylko jedna procedura

- **praca wielowątkowa** – procedury wykonywane są sekwencyjnie, lecz kolejność ich wykonania jest trudna do przewidzenia dla programisty

Techniki implementacji

- **praca równoległa** – wiele procedur wykonywanych jest w tym samym czasie, wymieniają się one danymi
- **rekurencja** – procedura lub funkcja wywołuje sama siebie, aż do uzyskania wyniku lub błędu
- **obiektowość** – procedury i dane łączymy w pewne klasy reprezentujące najważniejsze elementy algorytmu oraz stan wewnętrzny wykonującego je systemu
- **algorytm probabilistyczny** – działa poprawnie z bardzo wysokim prawdopodobieństwem, ale wynik nie jest pewny

Spotkanie #2

Programowanie I
poziom podstawowy

Szybka powtórka

- wyrażenia regularne
- pseudokod
- algorytmika

Rozkład jazdy

09:00 - **Maven** + **Guava**
10:30 - przerwa krótka
10:40 - tablice (podstawy + klasa **Arrays** + varargs)
11:30 - tablice (sortowanie)
12:40 - przerwa długa
13:00 - listy (podstawy + klasa **Collections** + **LinkedList**)
14:30 - przerwa krótka
14:40 - mini - projekt
16:00 - koniec zajęć :)

Maven

XML - format danych

Extensible Markup Language

XML

dokument tekstowy zapisany w specjalnym formacie opracowanym przez W3C (ang. World Wide Web Consortium) w 1998 r. Format ten jest czytelny zarówno dla maszyn jak i ludzi. Ułatwia przechowywanie i przesyłanie informacji pomiędzy systemami informatycznymi.

- **XML** to zestaw elementów tworzących strukturę dokumentu i przechowujących dane w strukturze drzewiastej
- na szczycie jest jeden element główny, tzw. korzeń (ang. root)
- pod nim może znajdować się dowolnie wiele elementów-potomków
- każdy element składa się ze znacznika otwierającego i zamykającego oraz zawartości - która może być tekstem albo zestawem elementów-potomków
- element XML może posiadać atrybuty
- narzędzia/specyfikacje związane z XML:
  - **DOM**, **SAX**, **StAX**, **JAXB** - narzędzie do parsowania i tworzenie dokumentów XML
  - **DTD**, **XSD** - walidacja poprawności dokumentów XML
  - **XPath** - język pozwalający na wskazywanie elementów i atrybutów w dokumencie XML
  - **XSLT** - język do transformowania dokumentów XML do innych formatów np. HTML, PDF itp

XML - przykład

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

<course>
    <name>JavaGda24</name>
    <blocks>
        <block hours=”64”>Wprowadzenie</block>
        <block hours=”8”>Git</block>
        <block hours=”32”>Programowanie 1</block>
    </blocks>
    <!-- komentarze wewnątrz XMLa -->
    <period start=”2018-10-01” end=”2019-06-01” />
</course>
```

prolog zawierający wersję XMLa i rodzaj kodowania

białe znaki pomiędzy elementami są ignorowane

element główny (root) - może być tylko jeden

element z zawartością tekstową

znacznik otwierający

znacznik zamykający

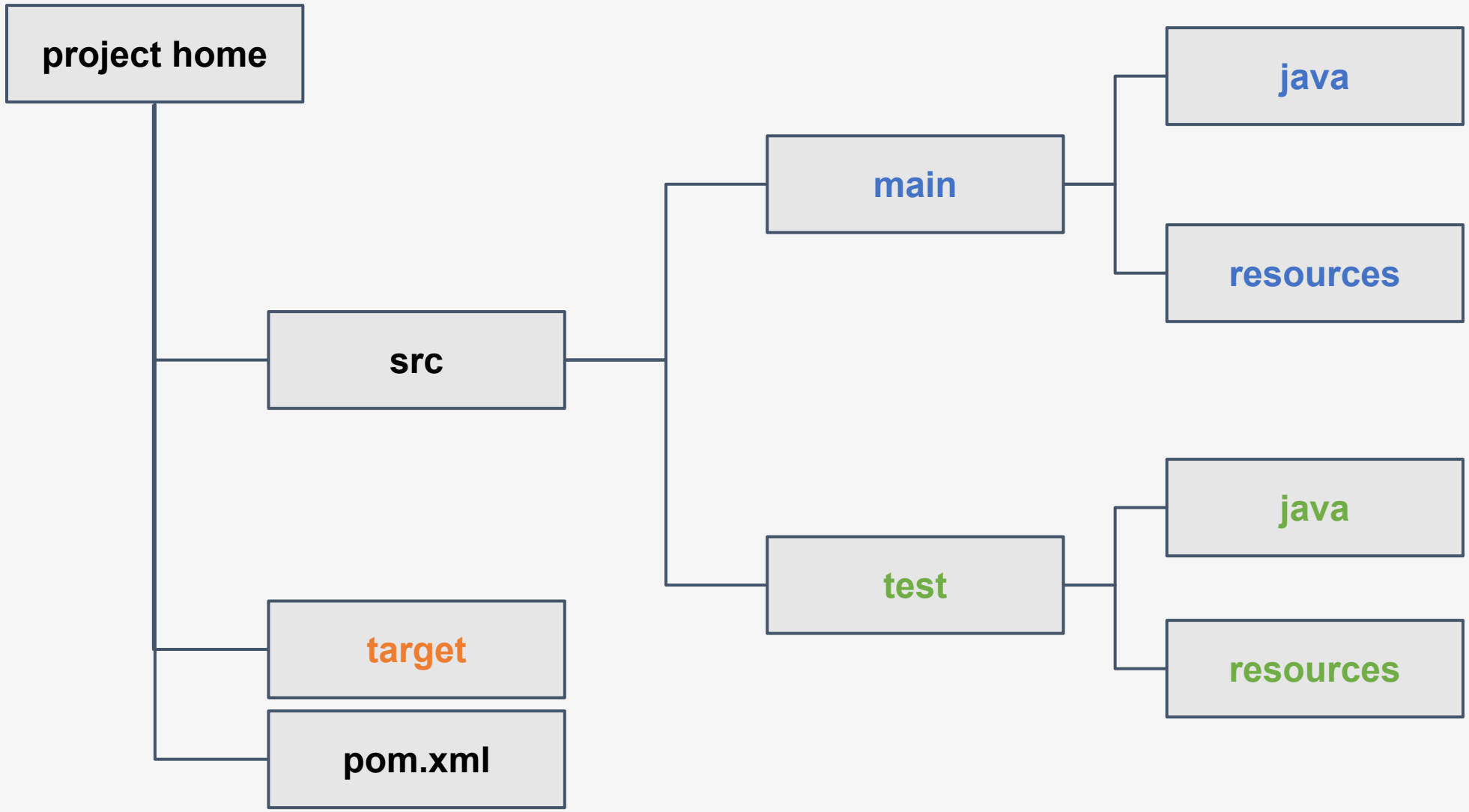
element bez zawartości

atrybut elementu

Maven - narzędzie do budowania projektu Java

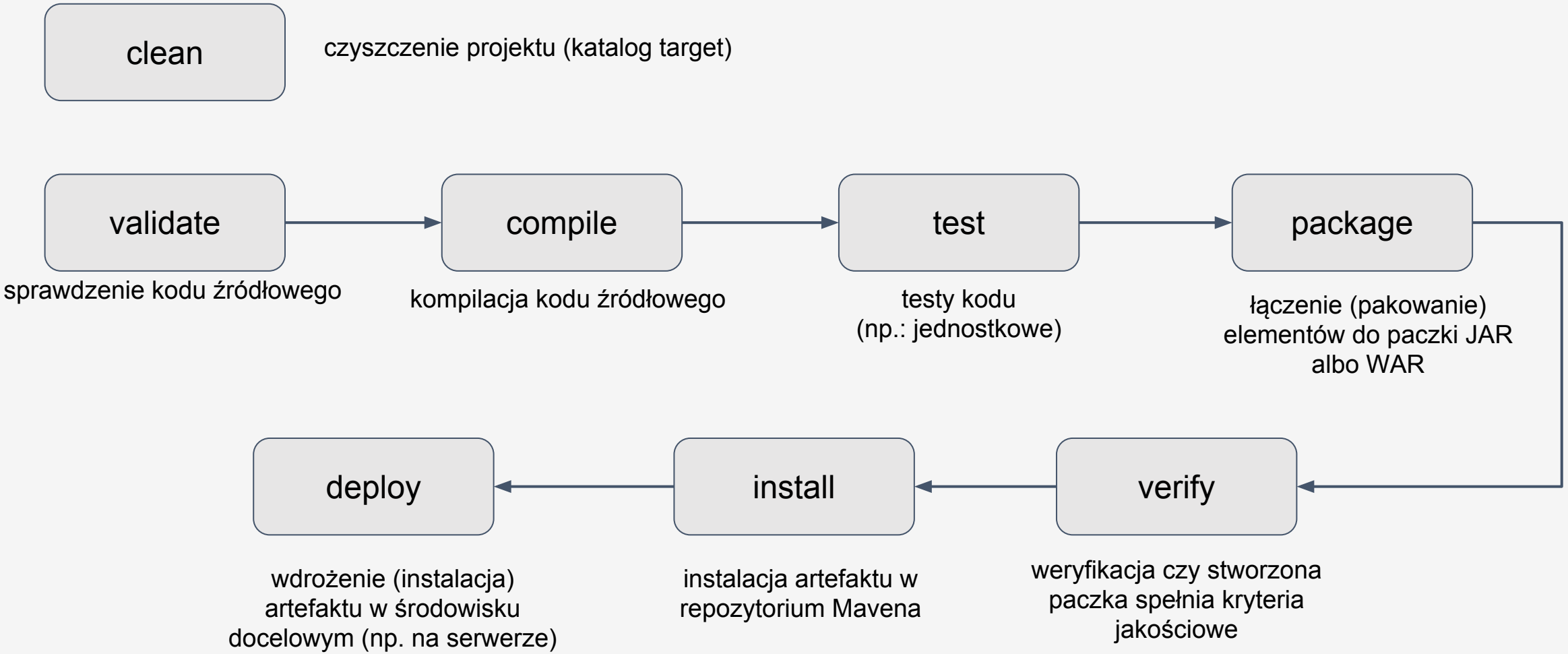
- **automatyzacja** - łatwe budowanie projektów Java
- **standaryzacja** - tego jak projekt ma wyglądać (struktury katalogów)
- **konwencja nad konfiguracją** - szybkie uruchamianie
- **rozszerzalność** - przez system pluginów
- **zarządzanie zależnościami projektu:**
  - **artefakt** - moduł/zależność w projekcie - identyfikacja 3-składnikowa:
    - **groupId**, **artifactId**, **version**
  - **repozytorium -** miejsce gdzie trzymane są zbiory artefaktów może być:
    - lokalny - np.: C:\Users\{user_name}**.m2**\repository
    - zdalne - np.: https://repo.maven.apache.org/maven2/

Maven - struktura katalogów



https://maven.apache.org/guides/introduction/introduction-to-the-pom.html

Maven - cykl życia



Guava - biblioteka od Google

```
<dependency>
    <groupId>com.google.guava</groupId>
    <artifactId>guava</artifactId>
    <version>27.0.1-jre</version>
</dependency>
```

- popularna biblioteka od Google
- zawiera własne implementacje kolekcji, m.in: Multiset, Multimap, Table
- klasy do obsługi Stringów, IO, cache
- i wiele innych ...

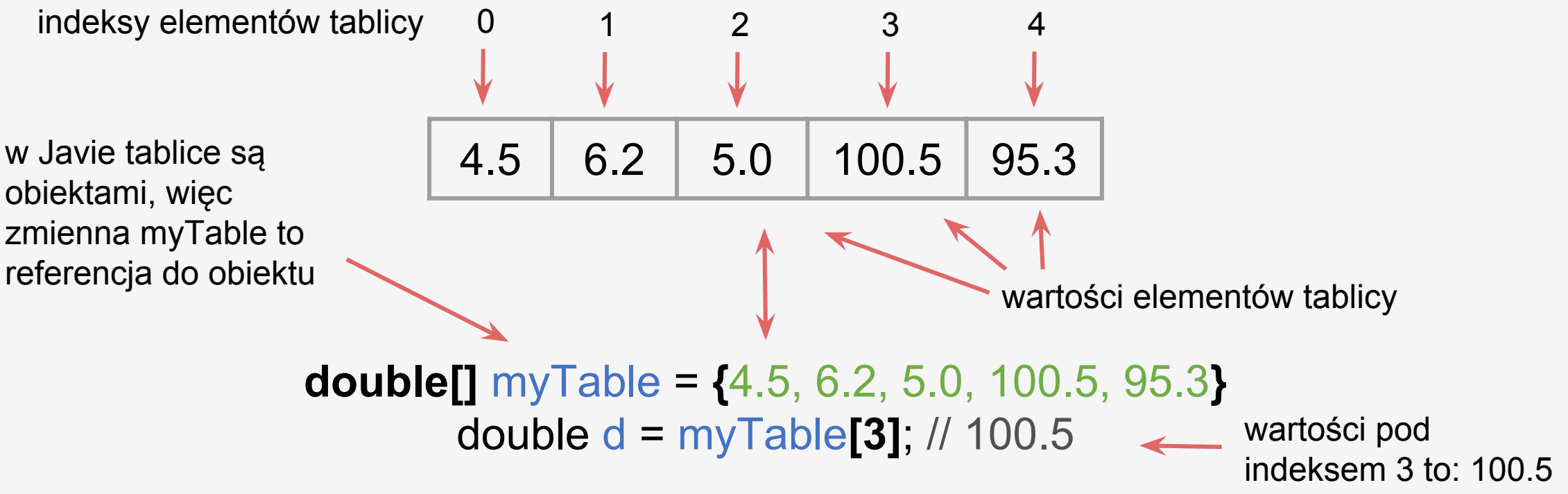
Przykłady w kodzie: pl.sda.dependencies.GuavaSamples

Tablice

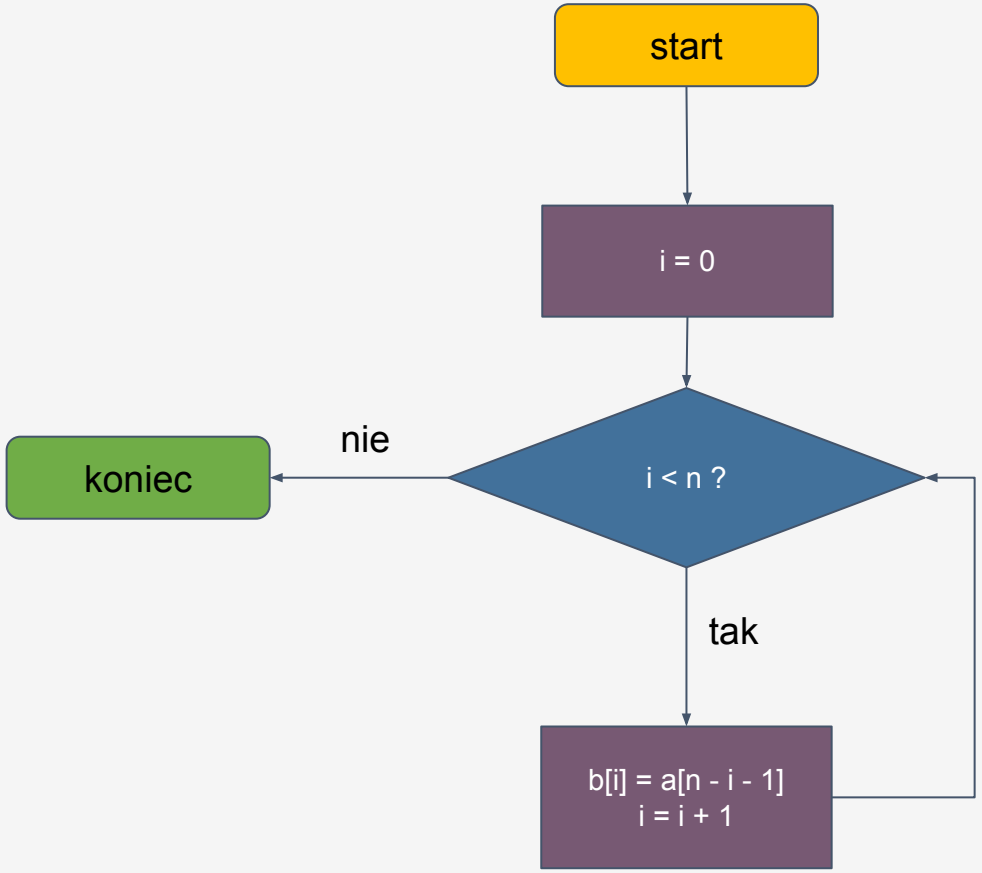
Tablica - definicja

Tablica

zestaw elementów (wartości) tego samego typu, ułożonych na określonych pozycjach. Do każdego z tych elementów mamy bezpośredni dostęp poprzez nazwę tablicy i pozycję elementu w zestawie (określaną jako **indeks** tablicy)

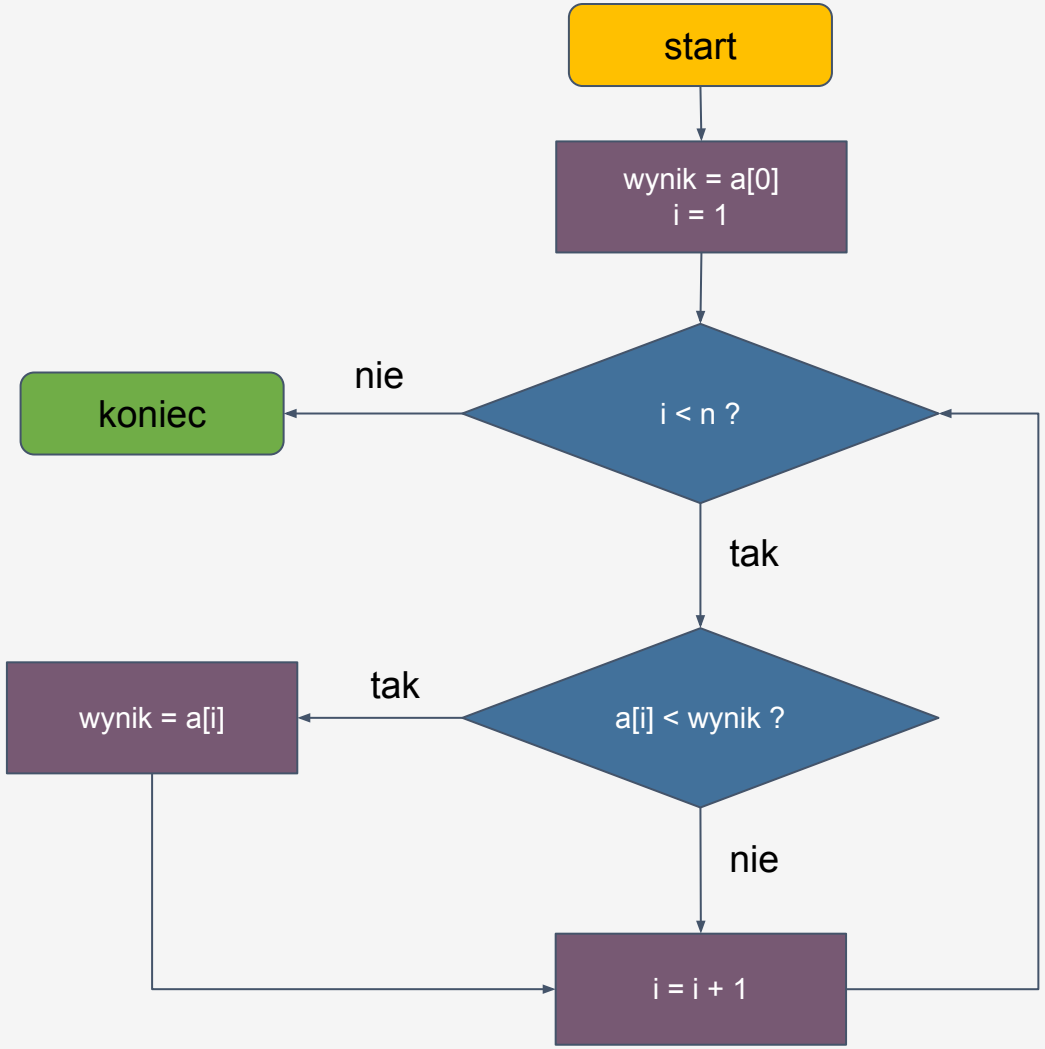


Tablica - odwrócenie kolejności



a[] - tablica elementów wejściowych
b[] - tablica elementów z odwróconą kolejnością
i - index aktualnie kopiowanego elementu
n - wielkość tablicy

Tablica - szukanie elementu minimalnego



a[] - tablica elementów do szukania
wynik - minimalny element
i - index aktualnie sprawdzanego elementu
n - wielkość tablicy

Metody o zmiennej liczbie argumentów (varargs)

Varargs
mechanizm pozwalający na tworzenie metod o zmiennej ilości argumentów, bez konieczności tworzenia tablic przechowujących te argumenty.

[typ] nazwa_metody([lista_parametrów], [typ]... nazwa_parametru)

Deklaracja:
void getNumbers(int… numbers)

zmienna typu tablicowego

float compute(float first, float… theRest)

Wywołanie:
getNumbers()
getNumbers(1)
getNumbers(1, 2, 3, 4, 5,)

compute(1.0F)
compute(5.5F, 6.1F, 6.6F)

Przykłady w kodzie: pl.sda.arrays.Samples - metoda varargsSamples()

Interfejs Comparable

Interfejs **java.lang.Comparable**:

- służy do określania relacji większy-mniejszy pomiędzy obiektami klasy, które implementują ten interfejs
- dzięki niemu możliwe jest stwierdzenie czy obiekt X jest większy, mniejszy czy też równy obiektowi Y
- wyznacza tzw. “naturalny” porządek obiektów klas
- używamy go najczęściej do sortowania kolekcji (tablic, list) - przez metody: Collections.sort() i Arrays.sort()
- zawiera jedną metodę compareTo() - która służy do porównywania dwóch obiektów
- zaleca się spójność między metodami compareTo() i equals()

Interfejs Comparable - przykład użycia

```
public class Person implements Comparable<Person> {
    private String name;
    private String surname;
    …
    public int compareTo(Person o) {
        if(surname.compareTo(o.surname) != 0) {
            return surname.compareTo(o.surname);
        }

        return name.compareTo(o.name);
    }
    ….
```

Metoda compareTo() powinna zwrócić wartość:

- mniejszą od zera jeżeli obiekt **this** jest mniejszy
- równą zeru jeżeli obiekt **this** jest równy
- większą od zera jeżeli obiekt **this** jest większy

Przykłady w kodzie: pl.sda.arrays.Samples - metoda arraysSorting()

Interfejs Comparator

Interfejs **java.util.Comparator**:

- służy do określania relacji większy-mniejszy pomiędzy różnymi obiektami tej samej klasy
- nie wymusza implementowania żadnych interfejsów przez klasy
- dzięki niemu można stworzyć “dodatkowe” i dowolne sposoby sortowania naszych klas
- używamy go najczęściej do sortowania kolekcji (tablic, list) - przez metody: Collections.sort() i Arrays.sort()
- zawiera jedną metodę compare() - która służy do porównywania dwóch obiektów
- zaleca się spójność między metodami compare() i equals()

Interfejs Comparator - przykład użycia

```
public class PersonByAgeComparator implements Comparator<Person> {
    public int compare(Person o1, Person o2) {
        ...
        return o1.getAge() - o2.getAge();
    }
}
```

```
Comparator<Person> comparator = Comparator.comparing(Person::getName)
                                          .thenComparing(Person::getAge);
```

Przykłady w kodzie: pl.sda.arrays.Samples - metoda arraysSorting()

Zadania

#arrays

Zadania

#arrays

1. Stwórz klasę, która będzie przechowywać tablicę Stringów. W konstruktorze należy podać rozmiar tablicy.
   Dodaj metody:
   a. do dodawania elementów - w przypadku gdy tablica nie pomieści więcej elementów metoda powinna wyrzucić wyjątek.
   b. pobierania elementów tablicy pojedynczych po indexie
   c. pobierania wszystkich elementów.

2. Dodaj do klasy z pkt 1 możliwość powiększania tablicy w przypadku gdy nowe elementy nie mieszczą się w niej.
   Podpowiedź: wykorzystaj metodę **Tasks.addElementToArray()** z projektu **java_pro1**

3. Dodaj do klasy z pkt 1 metody, które zwrócą:
   a. kopię tablicy, ale z odwróconą kolejnością elementów
   b. kopię tablicy z posortowanymi elementami
   c. najmniejszy element tablicy
   d. największy element tablicy

4. * Zmień klasę z pkt 1 tak żeby mogła przechowywać dowolne elementy - jedyny warunek jest taki że muszą implementować interfejs **Comparable**

5. * https://pl.spoj.com/problems/TRN/

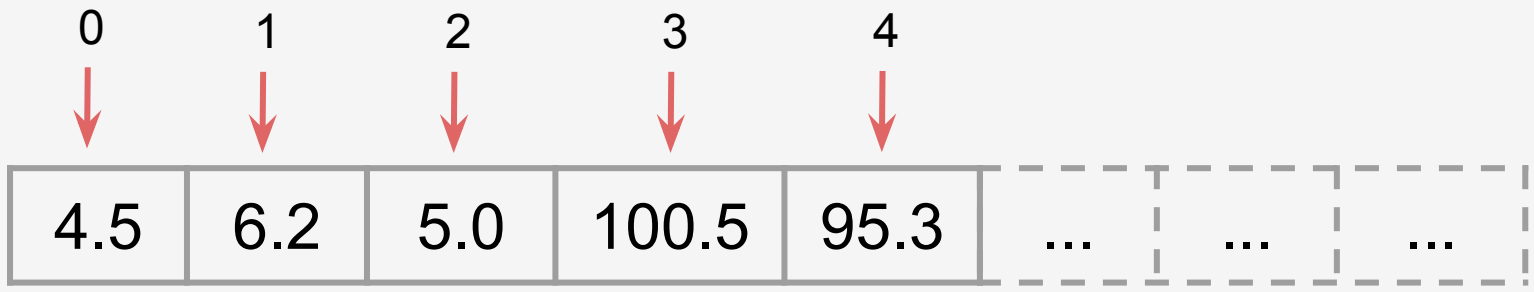
Pamiętaj o wykorzystaniu repozytorium kodu Git! * Dla chętnych.

Listy

ArrayList vs LinkedList

ArrayList:

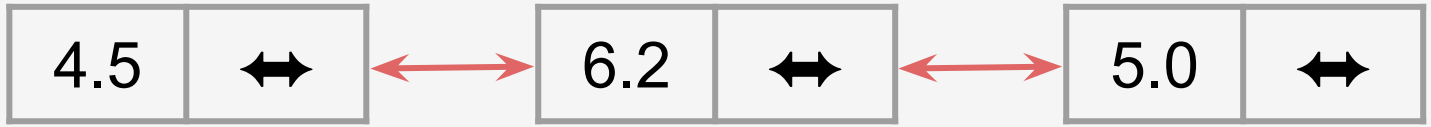
- lista oparta na tablicy
- dynamiczna (zmienia rozmiar w trakcie)
- udostępnia szereg dodatkowych metod do pracy z elementami




```
List<String> colors = new ArrayList<>(10);
```

LinkedList:

- lista łączona dwustronnie
- każdy element ma referencje do poprzednika i następnika
- udostępnia szereg dodatkowych metod do pracy z elementami

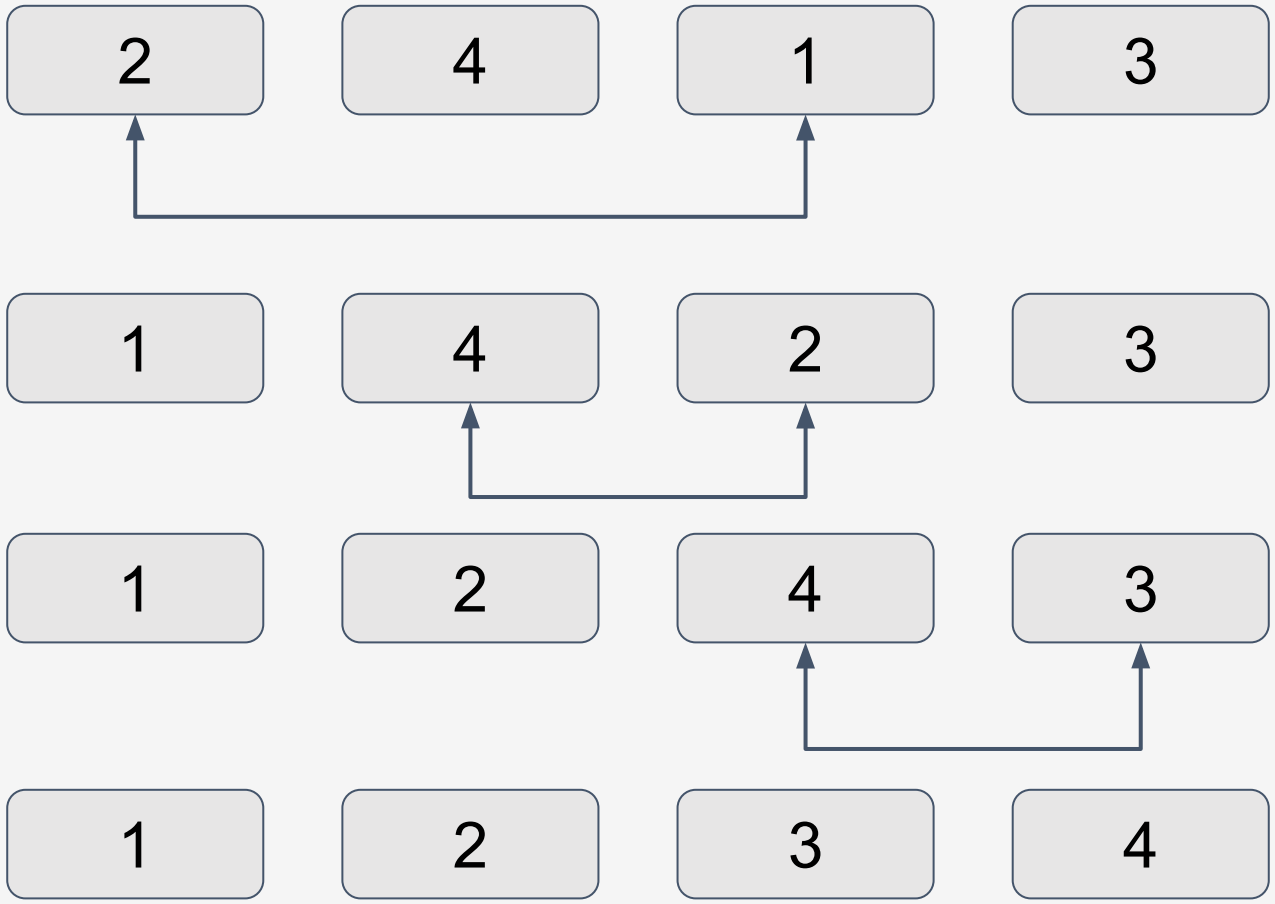



```
List<String> colors = new LinkedList<>();
```

ArrayList vs LinkedList

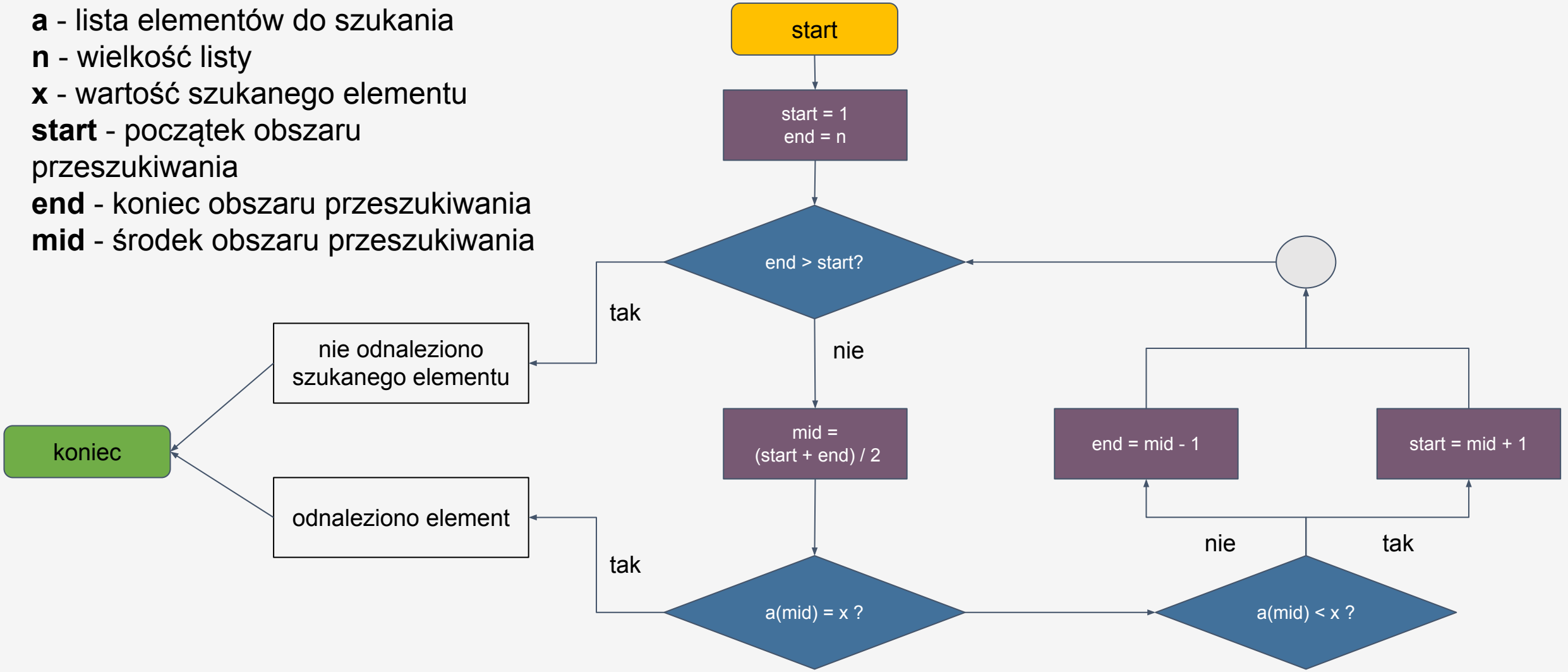
	ArrayList	**LinkedList**
	dynamiczna (zmieniająca rozmiar) tablica	podwójnie linkowana lista
dostęp swobodny - metoda get()	O(1)	O(n)
dodaj(add()) / usuń z końca listy	amortyzowane O(1)	O(1)
dodaj / usuń z początku listy	O(n)	O(1)
jedna iteracja po liście	O(1)	O(1)
dodaj / usuń ze środka listy przez iterator	O(n)	O(1)
dodaj / usuń ze środka listy przez index	O(n)	O(n)

Lista - sortowanie przez wstawianie



Lista - szukanie binarne

a - lista elementów do szukania
n - wielkość listy
x - wartość szukanego elementu
start - początek obszaru przeszukiwania
end - koniec obszaru przeszukiwania
mid - środek obszaru przeszukiwania



Zadania

#lists

Zadania

#lists

1. Uzupełnij klasę **pl.sda.list.mini_project.BooksManager** tak żeby metoda *findBooks()* zwracałą listę książek zapisanych w tej klasie.

2. Dodaj kod do metody klasy z pkt 1, który umożliwi dodawanie książek

3. Dodaj kod do metody klasy z pkt 1, który umożliwi usuwanie książek

4. Dodaj kod do metody klasy z pkt 1, który umożliwi sortowanie książek

5. * Zmień metodę do sortowania przez wstawianie tak żeby przyjmowała listę dowolnych obiektów (wystarczy że będą implementować klasę Comparable)

6. * Zmień metodę do szukania binarnego tak żeby przyjmowała listę dowolnych obiektów (wystarczy że będą implementować klasę Comparable)

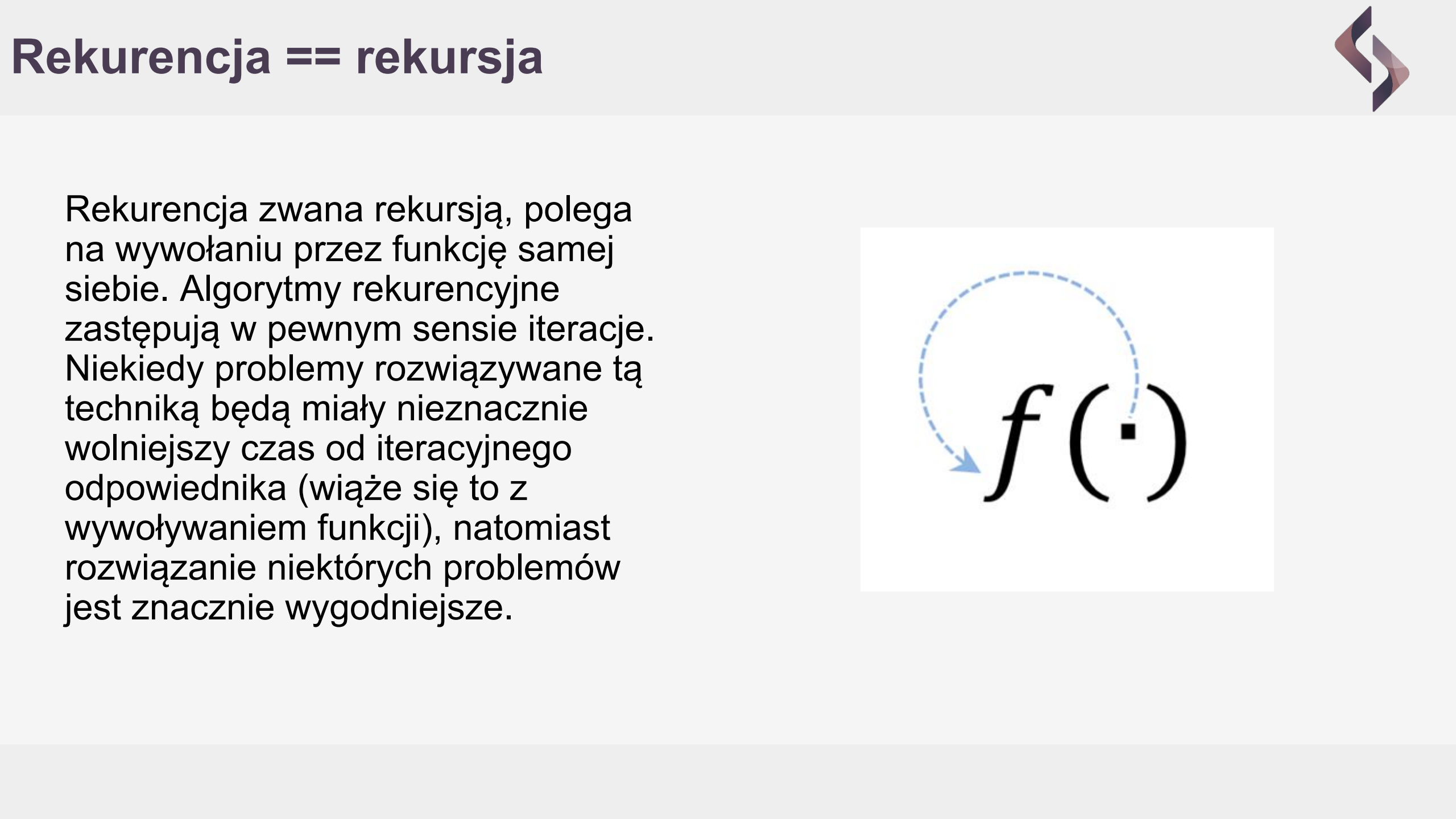
Pamiętaj o wykorzystaniu repozytorium kodu Git! * Dla chętnych.

Pytania?

Spotkanie #3

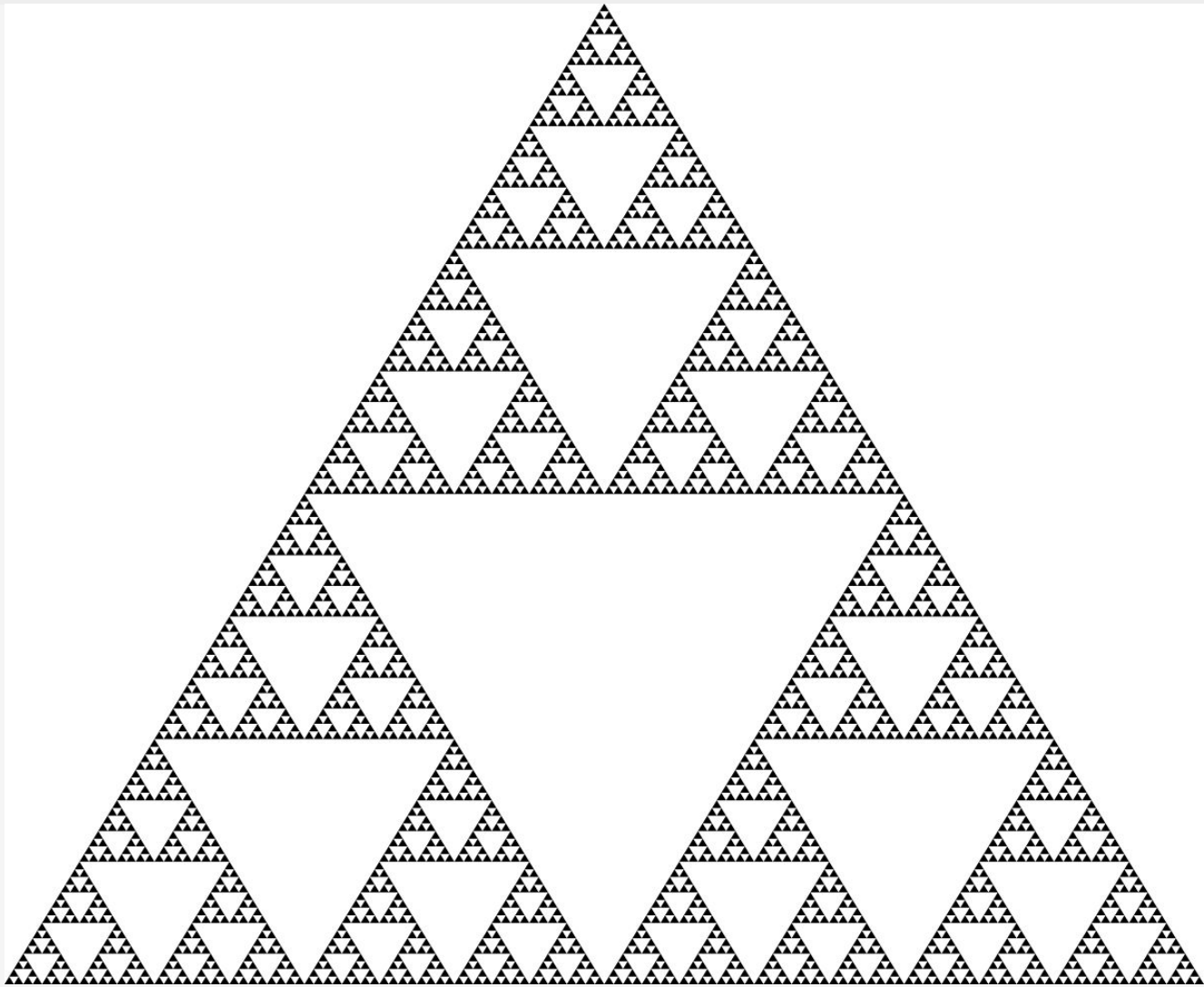
Programowanie I
poziom podstawowy

Rekurencja == rekursja



Rekurencja zwana rekursją, polega na wywołaniu przez funkcję samej siebie. Algorytmy rekurencyjne zastępują w pewnym sensie iteracje. Niekiedy problemy rozwiązywane tą techniką będą miały nieznacznie wolniejszy czas od iteracyjnego odpowiednika (wiąże się to z wywoływaniem funkcji), natomiast rozwiązanie niektórych problemów jest znacznie wygodniejsze.

Rekurencja - przykład graficzny



Rekurencja - zadania

1. Napisz metodę, która pozwoli obliczyć zadany element ciągu Fibonnaciego z wykorzystaniem rekurencji -> https://pl.wikipedia.org/wiki/Ci%C4%85g_Fibonacciego

   (zacznij od napisania kodu, który będzie realizował wyliczanie ciągu bez użycia rekurencji)

2. Napisz metodę, która pozwoli obliczyć silnię z danej liczby wykorzystując przy tym rekurencję.

Złożoność obliczeniowa

Złożoność obliczeniowa

Program komputerowy rozwiązujący określony problem posiada do swej dyspozycji dwa podstawowe zasoby:

- czas
- pamięć

Przez czasową złożoność obliczeniową rozumiemy ilość czasu niezbędnego do rozwiązania problemu w zależności od liczby danych wejściowych.

Złożoność pamięciowa określa z kolei liczbę komórek pamięci, która będzie zajęta przez dane i wyniki pośrednie tworzone w trakcie pracy algorytmu.

Odmiany złożoności

Ponieważ często zużycie zasobów w algorytmie uzależnione jest od postaci przetwarzanych danych, zarówno złożoność czasowa jak i pamięciowa może występować w trzech odmianach:

- $T_O(n)$ - optymistycznej
- $T_A(n)$ - średniej
- $T_W(n)$ - pesymistycznej

Przykład liczenia złożoności

Zadanie: Zastanówmy się ile wyniesie złożoność obliczeniowa optymistyczna, średnia i pesymistyczna dla poniższego problemu:

Dopóki w koszu są jabłka, wyjmij jedno jabłko z kosza, obejrzyj je, jeśli jest robaczywe, to zakończ. Inaczej odłóż je na bok i wróć do początku.

Przykład liczenia złożoności - rozwiązanie

Rozważmy, ile operacji dominujących (ocena jabłka) wykona ten algorytm dla n jabłek:

- zakładamy **przypadek optymistyczny** - robaczywe jabłko napotkamy za pierwszym razem:
  $T_o(n) = 1$ - złożoność optymistyczna
- zakładamy **przypadek najgorszy** - w koszu brak jabłek robaczywych lub robaczywe jabłko będzie wyjęte z kosza jako ostatnie:
  $T_w(n) = n$ - złożoność pesymistyczna
- w **przypadku typowym** robaczywe jabłko znajdziemy po przejrzeniu połowy jabłek w koszu - tzn. raz będzie ono wyciągnięte wcześniej, raz później, a średnio w $^n/_2$ teście:
  $T_A(n) = {}^n/_2$ - złożoność średnia, oczekiwana

Liczenie złożoności - podsumowanie

Złożoność optymistyczna określa zużycie zasobów dla najkorzystniejszego zestawu danych.

Złożoność średnia określa zużycie zasobów dla typowych (tzw. losowych) danych.

Złożoność pesymistyczna określa zużycie zasobów dla najbardziej niekorzystnego zestawu danych

Liczenie czasowej złożoności obliczeniowej - przykład

Zadanie: Zastanówmy się ile wyniesie czasowa złożoność obliczeniowa dla poniższego problemu:

Dane jest n liczb naturalnych, zsumuj je.

Wejście:

n - określa ile kolejnych liczb naturalnych ma być sumowane

Wyjście:

suma *n* kolejnych liczb naturalnych

Liczenie czasowej złożoności obliczeniowej - rozwiązanie

Krok	Operacja	Czas wykonania
Krok 1:	**Czytaj** n	$1 \times t_1$
Krok 2:	$suma \leftarrow 0$	$1 \times t_2$
Krok 3:	$i \leftarrow 1$	$1 \times t_2$
Krok 4:	**Jeśli** $i > n$, **to idź do** K08	$(n + 1) \times t_{4a}$ dla **jeśli** ..., $1 \times t_{4b}$ dla **idź do** ...
Krok 5:	$suma \leftarrow suma + i$	$n \times t_5$
Krok 6:	$i \leftarrow i + 1$	$n \times t_6$
Krok 7:	**Idź do** K04	$n \times t_{4b}$
Krok 8:	**Pisz** $suma$	$1 \times t_8$
Krok 9:	**Zakończ**	$1 \times t_9$

Liczenie czasowej złożoności obliczeniowej - podsumowanie

Dobrym sposobem określenia złożoności czasowej jest wyznaczenie w algorytmie operacji dominującej i zliczenie liczby jej wykonań. Pozostałe operacje traktujemy jako nieistotne - tzn. ich czas wykonania jest pomijalnie mały w porównaniu z czasem wykonania wszystkich operacji dominujących. W naszym algorytmie taką operacją dominującą może na przykład jeden obieg pętli sumującej liczby naturalne.

Wniosek: złożoność algorytmu jest liniowa

Klasy złożoności

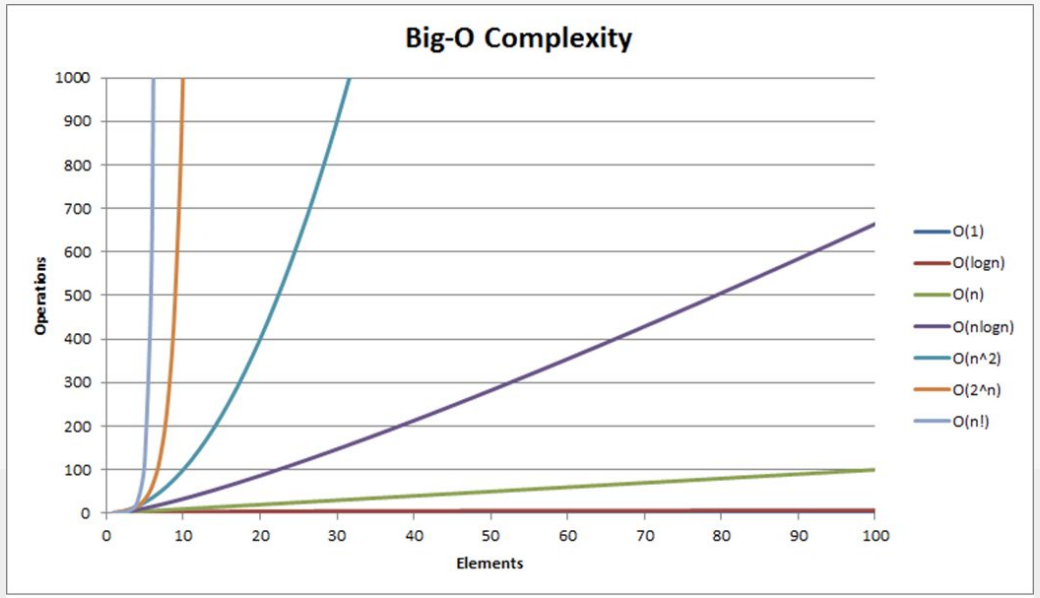
Przy analizie algorytmów korzysta się z tzw. **klas złożoności obliczeniowej**, które określają rząd funkcji T(*n*). Jednym ze sposobów określania rzędu tej funkcji jest popularna notacja omikron (zwana także notacją dużego O)

- O(1) - **stała** klasa czasowej złożoności obliczeniowej - Algorytm wykonuje stałą liczbę operacji bez względu na rozmiar danych n.
- O(n) - **liniowa** klasa czasowej złożoności obliczeniowej - Algorytm wykonuje stałą liczbę operacji dla każdej danej n.
- $O(n^2)$ - **kwadratowa** klasa czasowej złożoności obliczeniowej - Dla każdej danej n algorytm wykonuje proporcjonalną do n liczbę operacji.
- Oprócz powyższych istnieją również inne charakterystyczne klasy złożoności obliczeniowej: ($O(\log n)$ - **logarytmiczna**, $O(n \log n)$ - **liniowo logarytmiczna**, $O(2^n)$, $O(n!)$ - **wykładnicza**).

Analogicznie wyglądają klasy złożoności pamięciowe.

Klasy złożoności

n	1	2	3	4	5	6	7	8	9	10	11	12	13	14
logarytmiczna (log n)	0	0	0	1	1	1	1	1	1	1	1	1	1	1
liniowa (n)	1	2	3	4	5	6	7	8	9	10	11	12	13	14
liniowo logarytmiczna (nlog n)	0	1	1	2	3	5	6	7	9	10	11	13	14	16
kwadratowa (n^2)	1	4	9	16	25	36	49	64	81	100	121	144	169	196
sześcienna (n^3)	1	8	27	64	125	216	343	512	729	1000	1331	1728	2197	2744
wykładnicza (2^n)	2	4	8	16	32	64	128	256	512	1024	2048	4096	8192	16384
silnia (n!)	1	2	6	24	120	720	5040	40320	362880	3628800	39916800	4,79E+08	6,23E+09	8,72E+10

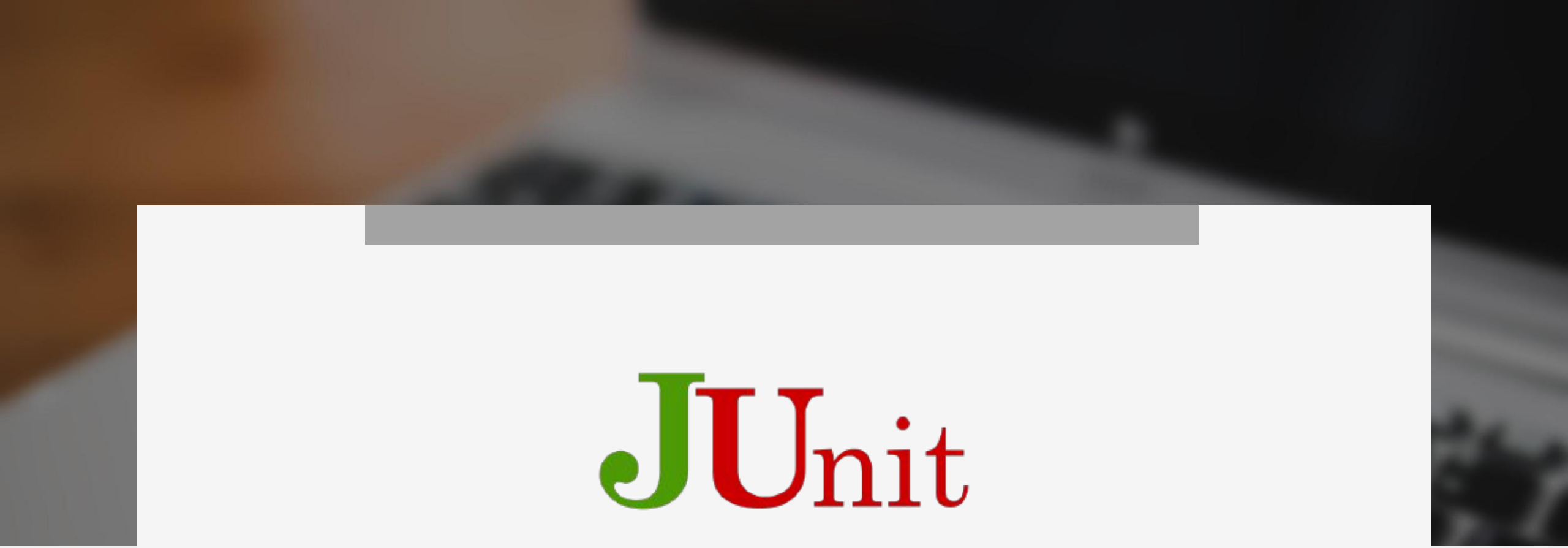


Zadania - pl.spoj.com

- https://pl.spoj.com/problems/TEST/

- https://pl.spoj.com/problems/PRIME_T/
- https://pl.spoj.com/problems/PA05_POT/
- https://pl.spoj.com/problems/PP0501A/
- https://pl.spoj.com/problems/FCTRL3/
- https://pl.spoj.com/problems/CALC/

Dla każdego z powyższych postaraj się ocenić klasę złożoności Twojego algorytmu!



JUnit

Stosy i kolejki

Operacje na stosie i kolejce

stos (stack)

- odłożenie elementu (push)
- zdjęcie elementu (pop)
- podejrzenie wierzchniego elementu (peek)
- sprawdzenie czy stos jest pusty

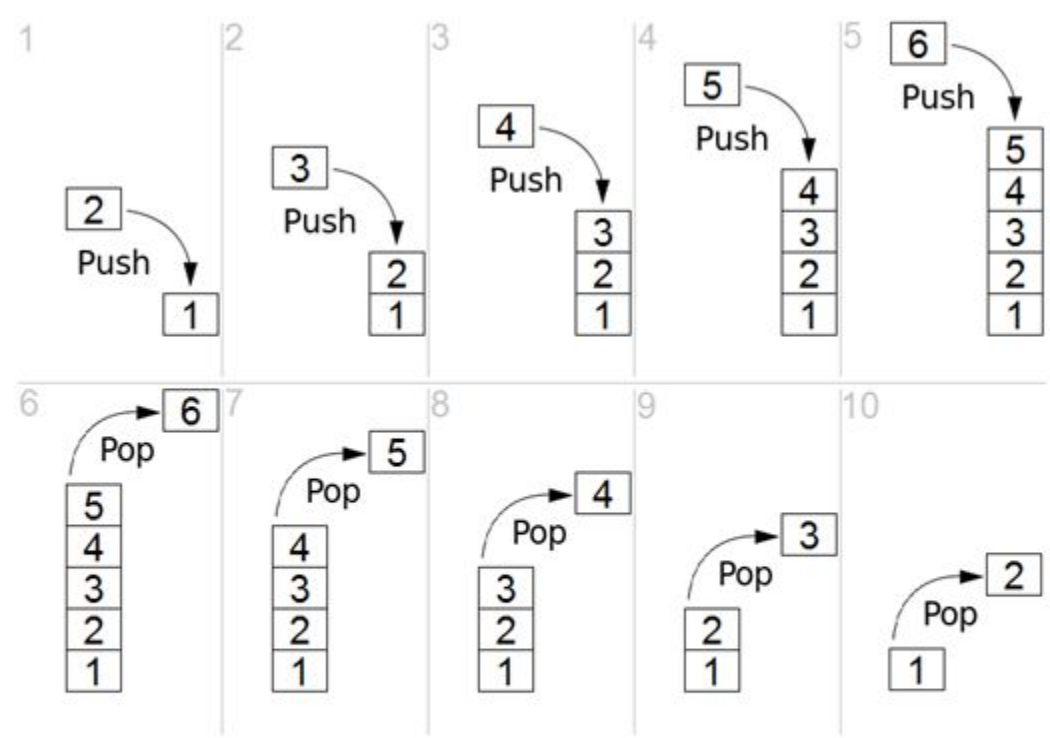
kolejka jednokierunkowa (queue)

- zakolejkowanie elementu (enqueue / offer)
- usunięcie elementu z kolejki (dequeue / poll)
- podejrzenie elementu na początku kolejki (peek)
- sprawdzenie czy kolejka jest pusta

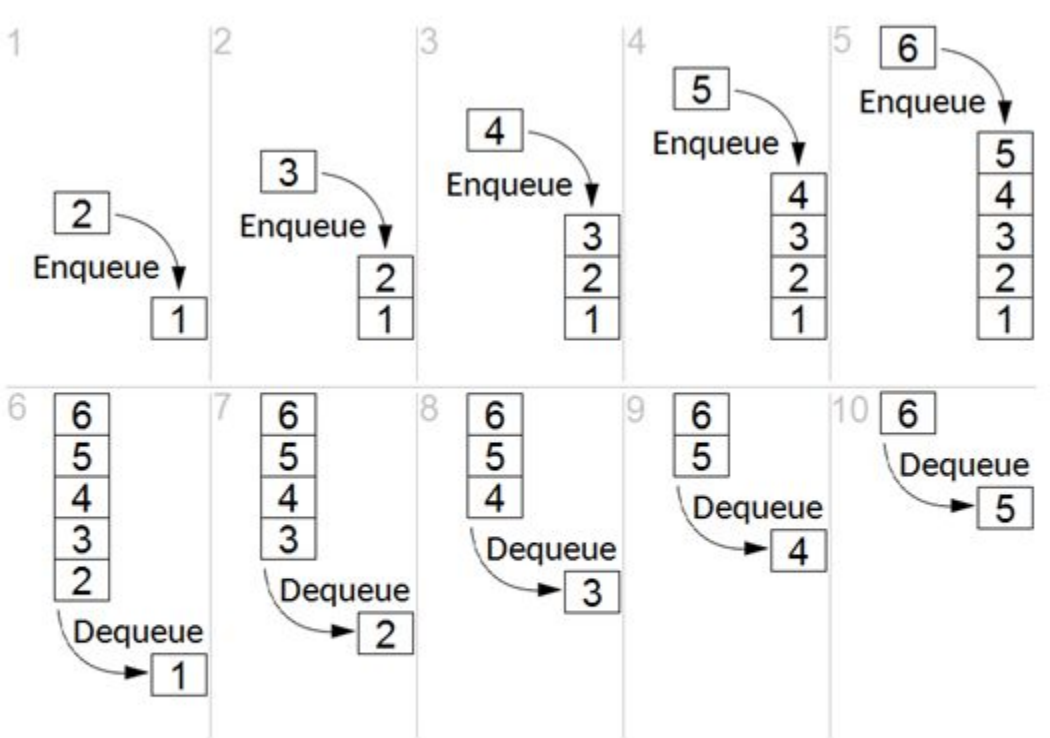
kolejka dwukierunkowa (dequeue)

- pozwala dodawać i usuwać elementy z obu stron
- uogólnienie stosów i kolejek

LIFO vs FIFO - https://visualgo.net/en/list



LIFO – *Last In, First Out*



FIFO – *First In, First Out*

Stos

Stos jest strukturą liniowo uporządkowanych danych, gdzie:

- jedynie ostatni element, zwany wierzchołkiem, jest w danym momencie dostępny
- nowy element można dodać jedynie na wierzchołku stosu
- usunąć element można jedynie z wierzchołka stosu

java.util.Stack

Podział na grupy

Dzielimy się na 5 grup
1 - Alpha
2 - Beta
3 - Gamma
4 - Delta
5 - Epsilon

Stos - zadanie - http://bit.ly/2E6E5DJ

A) Zaimplementuj stos jako klasę, która może przechowywać elementy zadeklarowanego typu. Dodaj i zaimplementuj metody push(), pop(), peek(), isEmpty(). Napisz testy (2++).

Alpha, Gamma, Epsilon -> wykorzystując tablicę
Beta, Delta -> wersja obiektowa (bez tablic, kolekcji)

B) Napisz metodę, która jako parametry przyjmuje liczbę dziesiętną oraz podstawę systemu pozycyjnego, a następnie zwraca w formie tekstowej zapis liczby w wybranym systemie (pseudokod dostępny jest na kolejnym slajdzie)

Wejście:
L – wartość liczby; **p** – podstawa systemu docelowego
Wyjście:
Liczba L w systemie p

https://pl.spoj.com/problems/STOS/

Stos - zadanie - pseudokod

```
K01: Utwórz pusty stos S
K02: S.push(L mod p)
K03: L = L div p
K04: Jeśli L > 0 idź do K02
K05: Dopóki !S.empty() wykonuj K06 i K07
K06:     S.peek() // wypisz cyfrę ze stosu
K07:     S.pop()
K08: Zakończ
```

Stos - zadanie - testy

- should push element to empty stack
- should push element to filled stack
- should delete element after popping
- should receive element when peeking but not delete it
- should return zero as size of empty stack
- should return correct size for non empty stack
- ...

Kolejka

java.util.**Queue** - kolejki dzielą się na dwa typy – **LIFO** (last-in, first-out) oraz **FIFO** (first-in, first-out). Ideą kolejki jest przechowywanie obiektów do przetworzenia w określonej kolejności. Wyróżniamy głowę oraz ogon kolejki.

- java.util.**LinkedList**
- java.util.**ArrayDeque**

Przechowuje elementy w tablicy. Zapewnia dostęp FIFO oraz LIFO (poprzez ogon).

- java.util.**PriorityQueue**

Nie pozwala na wartości **null**. Każdy element ma swój priorytet wykonania.

Złożoność czasowa operacji na ArrayDeque

operacja				złożoność
stos	**kolejka**	**kolejka dwukierunkowa**	**lista**	
push	offer	offerFirst, offerLast	add, addFirst, addLast	*O*(1)*
pop	poll	pollFirst, pollLast	remove, removeFirst, removeLast	*O*(1)
peek	peek	peekFirst, peekLast	getFirst, getLast	*O*(1)
size				*O*(1)
contains				*O*(*n*)
clear				*O*(*n*)

Zadania

1. Zaimplementuj klasę kolejki na podstawie klasy stos. Zmień implementację metod push(), pop(), peek(), isEmpty(). Utwórz interfejsy będące API każdej z klas. Napisz testy jednostkowe.
2. Napisz metodę pozwalającą sprawdzić podany ciąg znaków zawierający nawiasy okrągłe: "(", ")". Metoda powinna zwrócić true jeśli nawiasy są prawidłowo zagnieżdżone. Np. dla "(())" zwraca true, ale dla "())(" zwraca false. Inne znaki są ignorowane. Wykorzystaj testy do sprawdzenia swojego kodu. Uwaga: stos, czy kolejka? **Napisz pseudokod w 1 kolejności.**
3. * Rozszerz implementację kolejki i spraw by ta była dwukierunkowa. Dodaj do swojej implementacji odpowiednie metody.
4. * Zaimplementuj na swoim stosie kalkulator zachowujący kolejność działań dla: +, -, *, / (StringTokenizer, postfix notation) - https://pl.wikipedia.org/wiki/Odwrotna_notacja_polska

Pamiętaj o wykorzystaniu repozytorium kodu Git! * Dla chętnych.

Spotkanie #4

Programowanie I
poziom podstawowy

Rozkład jazdy

09:00 - powtórka
09:30 - kolekcja **Set**
10:30 - przerwa krótka
10:35 - klasy wewnętrzne i iteratory
11:30 - kolekcja **Map**
13:00 - przerwa długa
13:30 - logi aplikacji (**slf4j** + **logback**)
14:30 - przerwa krótka
14:35 - logi aplikacji (**slf4j** + **logback**)

Set & Map

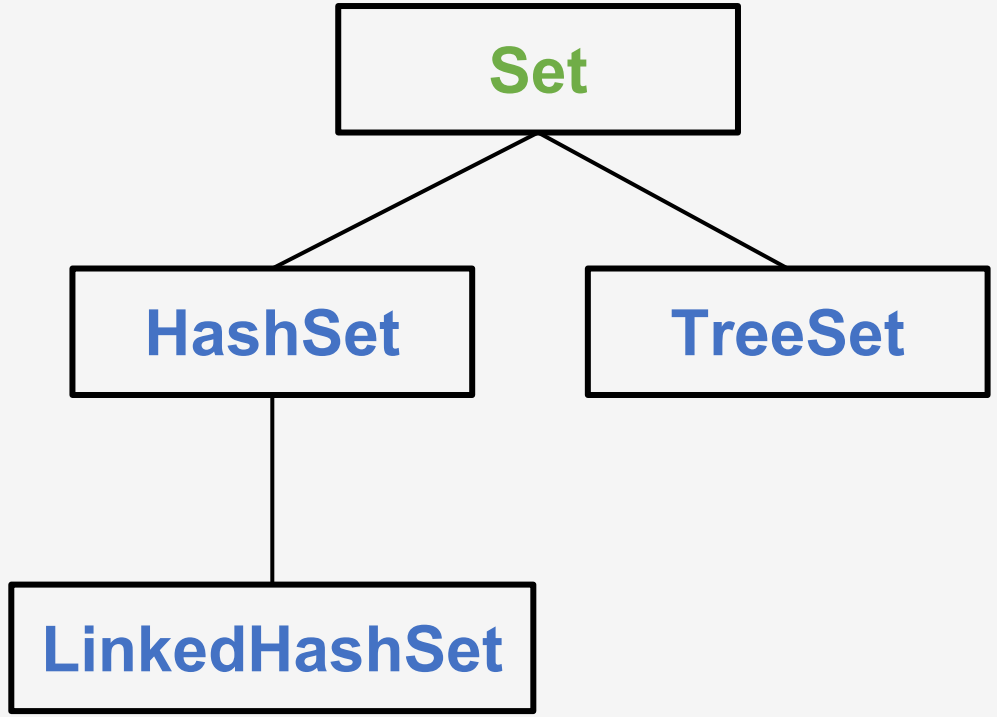
java.util.Set

Set

kolekcja danych, w której elementy nie mogą się powtarzać. Może zawierać co najwyżej jeden element o wartości null. Elementy w kolekcji są nieuporządkowane (jest jeden wyjątek - **TreeSet**!), nie można ich pobrać za pomocą indeksu.

O tym czy dwa obiekty są takie same decyduje:

- w przypadku **HashSet** i **LinkedHashSet** metoda ***equals()***
- w przypadku **TreeSet** metoda ***compareTo()***/***compare()***



equals() vs ==

Operator == porównuje adresy obiektów w pamięci aplikacji

Klasa **PersonOne** nie nadpisuje metody ***equals()***

```
PersonOne adam1 = new PersonOne("Adam", "Nowak");
PersonOne adam2 = new PersonOne("Adam", "Nowak");

System.out.println(adam1 == adam2); // zwraca false
System.out.println(adam1.equals(adam2)); // zwraca false
```

equals() vs ==

Metoda ***equals()*** domyślnie również porównuje adresy.
Nadpisana powinna porównywać obiekty według logiki biznesowej danej aplikacji.

Klasa **PersonTwo** nadpisuje metodę ***equals()*** - sprawdza czy podane obiekty mają te same imię i nazwisko

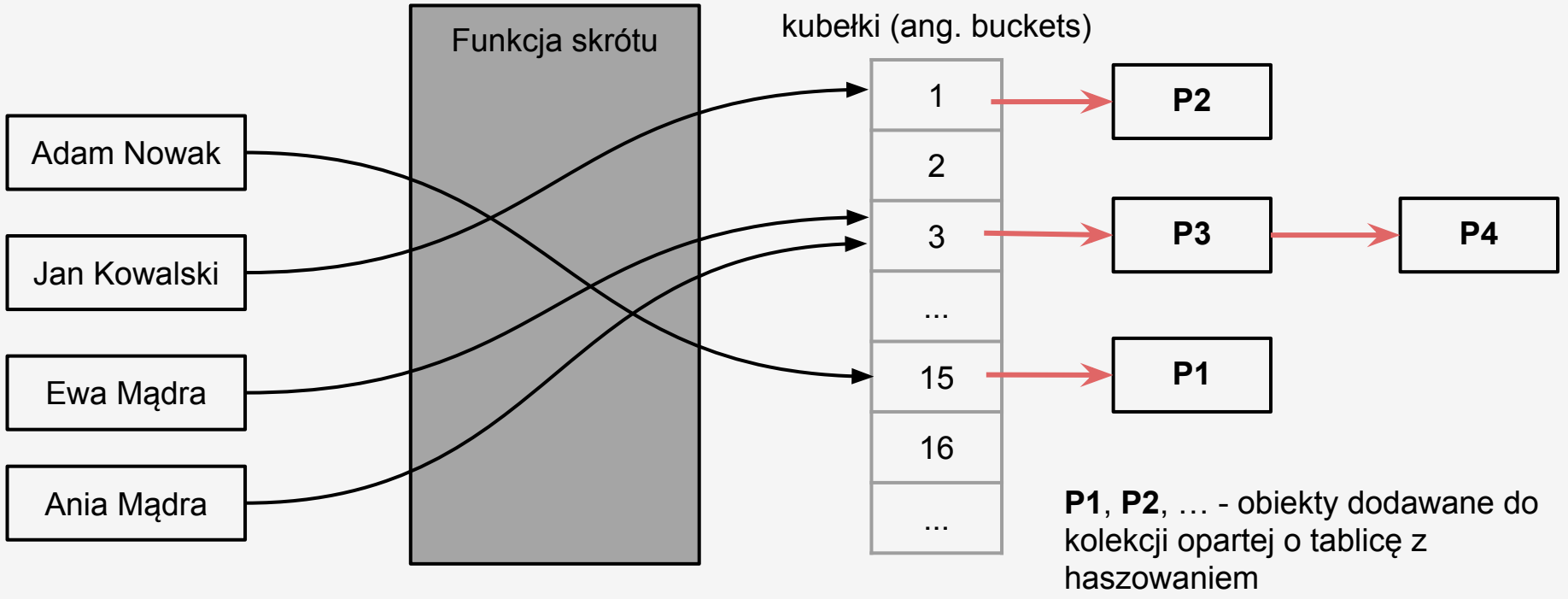
```
PersonTwo marek1 = new PersonTwo("Marek", "Kropka");
PersonTwo marek2 = new PersonTwo("Marek", "Kropka");

System.out.println(marek1  == marek2); // zwraca false
System.out.println(marek1.equals(marek2)); // zwraca true
```

Pełny kod: pl.sda.set.Samples metoda *hashCodeAndEquals()*

java.util.HashSet i java.util.LinkedHashSet

Tablica z haszowaniem (ang. hash table)

struktura danych umożliwiająca szybki dostęp do danych, które przechowuje. Złożone są ze zwykłych tablic indeksowanych liczbami. Do obliczania indeksu dla podanego klucza służy funkcja haszująca (skrótu), która przekształca klucz do liczby w odpowiednim zakresie.



P1, **P2**, … - obiekty dodawane do kolekcji opartej o tablicę z haszowaniem

W Javie do wyliczania indeksu tablicy z haszowaniem (jako składowa funkcji skrótu) używana jest wartość zwracana przez metodę ***hashCode()***.

Z tego mechanizmu korzystają:

- **HashSet**
- **LinkedHashSet**
- **HashMap**
- **LinkedHashMap**

hashCode() & equals()

- **equals** - służy do porównywania obiektów oraz powinna być:
  - zwrotna - object.equals(object) == true
  - symetryczna - a.equals(b) == b.equals(a)
  - przechodnia - a.equals(b), b.equals(c), a.equals(c)
  - spójna - zawsze zwraca ten sam wynik dla tego samego porównania

Kontrakt między metodami *equals()* i *hashCode()*

1. **a**.equals(**b**) == true wówczas wymagane jest aby **a**.hashCode() == **b**.hashCode(),
2. Jeśli **a**.hashCode() == **b**.hashCode() to nie jest wymagane aby **a**.equals(**b**) == true.

- **hashCode** - metoda zwracająca "skrót" danego obiektu (hash) w postaci danej typu **int**
  - najlepiej rozkład jednostajny
  - przyporządkowanie do grupy (tzw. kubełków lub wiaderek)
  - więcej niż jeden obiekt może mieć ten sam hash
  - zawsze zwraca tą samą wartość dla tego samego obiektu

HashSet vs LinkedHashSet vs TreeSet

	HashSet	**LinkedHashSet**	**TreeSet**
Opis	tablica z haszowaniem	tablica z haszowaniem + dwustronnie połączona lista	drzewo czerwono-czarne
Wymagania	obiekty muszą mieć zaimplementowane (i zgodne) metody *hashCode()* i *equals()*	obiekty muszą mieć zaimplementowane (i zgodne) metody *hashCode()* i *equals()*	obiekt musi implementować Comparable albo odpowiedni Comparator musi zostać dostarczony
dodaj, usuń - metody add(), remove()	O(1) - średni czas O(log n) - najgorszy (gdy jest jeden kubełek)	O(1) - średni czas O(log n) - najgorszy	O(log n)
metoda contains()	O(1) - średni czas O(log n) - najgorszy	O(1) - średni czas O(log n) - najgorszy	O(log n)
iteracja po kolekcji	O(1)	O(1)	O(1)

Klasy wewnętrzne

```
public class Product {
   protected static class ByNameReversed {
      public int compare(Product o1, Product o2) {
          ....
      }
   }

   protected class ByNameAndId {
      public int compare(Product o1, Product o2) {
          ....
      }
   }
   ...
}
```

- klasa wewnętrzna to klasa zdefiniowana wewnątrz innej klasy
- klasy wewnętrzne mogą być ukryte przed innymi klasami pakietu (mogą być private!)
- dzięki nim można uniknąć kolizji nazw
- porządkują kod
- mogą być niestatyczne - wtedy mają dostęp do wszystkich składowych klasy otaczającej
- mogą być statyczne - zadeklarowane ze specyfikatorem static - wtedy mają dostęp tylko do statycznych składowych klasy otaczającej

Przykłady w kodzie: pl.sda.inner.Samples

Iteratory

```
public class Iterable<E> {
        Iterator<E> iterator();
        ...
}
public class Iterator<E> {
        boolean hasNext();
        E next();
        default void remove() {...}
        ...

}
List<String> names = …;
Iterator<String> iterator = names.iterator();
while (iterator.hasNext()) {
    String name = iterator.next();
    ...
}
```

- interfejs Iterable służy do implementowania klas-kolekcji, których można użyć w pętli foreach
- iteratory to obiekty umożliwiające iterowanie (czyli pobieranie) poszczególnych elementów ze zbioru danych np. z listy, mapy czy seta
- żeby stworzyć własny iterator trzeba zaimplementować interfejs Iterator
- metoda **remove()** jest opcjonalna - nie trzeba jej implementować. Jeżeli tego nie zrobimy a metoda zostanie wywołana wyrzucony zostanie wyjątek **UnsupportedOperationException**
- listy mają bardziej rozbudowane iteratory: ListIterator

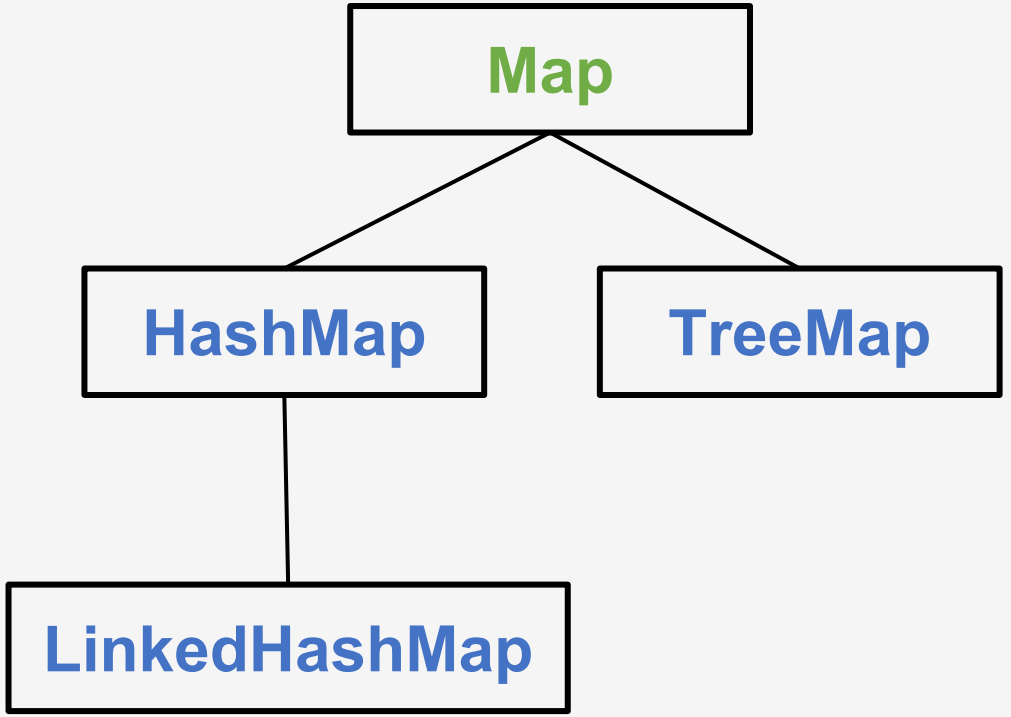
Przykłady w kodzie: pl.sda.iterator.Sample

java.util.Map

Map (słownik, tablica asocjacyjna)
kolekcja danych, w której przechowywane są pary danych: klucz i wartość. Klucze nie mogą się powtarzać. Umożliwia przypisanie do klucza konkretną wartość - jak w słowniku gdzie np. słowo po polsku to klucza a po angielsku to wartość.

O tym czy dwa klucze są takie same decyduje:

- w przypadku **HashMap** i **LinkedHashMap** metoda ***equals()***
- w przypadku **TreeMap** metoda ***compareTo()*/*compare()***



HashMap vs LinkedHashMap vs TreeMap

	HashMap	**LinkedHashMap**	**TreeMap**
Opis	tablica z haszowaniem	tablica z haszowaniem + dwustronnie połączona lista	drzewo czerwono-czarne
Wymagania	obiekty muszą mieć zaimplementowane (i zgodne) metody *hashCode()* i *equals()*	obiekty muszą mieć zaimplementowane (i zgodne) metody *hashCode()* i *equals()*	obiekt musi implementować Comparable albo odpowiedni Comparator musi zostać dostarczony
dodaj, usuń - metody add(), remove()	O(1) - średni czas O(log n) - najgorszy (gdy jest jeden kubełek)	O(1) - średni czas O(log n) - najgorszy	O(log n)
metoda contains()	O(1) - średni czas O(log n) - najgorszy	O(1) - średni czas O(log n) - najgorszy	O(log n)
iteracja po kolekcji	O(1)	O(1)	O(1)

Zadania

#set&map

Zadania

#set&map

1. Stwórz klasę, która będzie przechowywać listę obiektów klasy **Citizen**. Dodaj metody, które dodadzą i usuną obywatela z listy. Zaimplementuj w nowej klasie interfejs **Iterable** jako zwykłą iterację po liście.
2. Dodaj do klasy z pkt 1 metodę, która zwróci obiekt **Iterable** to przeszukiwania listy w odwrotnej kolejności.
3. * Zmodyfikuj klasę z pkt 1 tak żeby przyjmowała dowolne obiekty.
4. * Dodaj do klasy z pkt 3 metodę która przyjmie obiekt klasy **Predicate**, a zwróci obiekt **Iterable,** który zwróci tylko te obiekty które spełniają warunek zapisane w klasie **Predicate**.
5. Dodaj kod do metody **pl.sda.mini_project.BooksManager**.*getAuthors()* tak żeby zwracała zbiór autorów (imię i nazwisko jako String) posortowanych alfabetycznie.
6. Dodaj kod do metody **BooksManager**.*getAuthorsBooks()* tak żeby zwracała mapę gdzie kluczem jest autor (imię i nazwisko jako **String)**, a wartością lista książek (obiekty klasy **Book**) tego autora znajdujących się w bibliotece.

Pamiętaj o wykorzystaniu repozytorium kodu Git! * Dla chętnych.

Logi aplikacji

Definicje

Logi aplikacji

komunikat wygenerowany w kodzie (np. informacja o błędzie, o zakończeniu przetwarzania jakiś danych itp.) i przekazywany do odpowiednich miejsc, np. zapisany w pliku, wypisany na konsoli, wysłany do bazy danych itp.

Log

pojedynczy komunikat wygenerowany w kodzie i zapisany w odpowiednim miejscu, zawiera on różne informacje o zdarzeniu: czas, miejsce w kodzie (nazwa klasy, numer linii kodu), poziomie itp. Przykład:
2018-10-12 18:47:34.553 INFO [main] Samples[10] - Start of application.

Logger

obiekt klasy Logger używany do generowanie logów aplikacji w kodzie. Podstawowa klasa do logowania zdarzeń z kodu. Każdy logger ma swoją nazwę nadawana w momencie tworzenia loggera - dobrą praktyką jest używanie nazw klas jako nazw loggerów, wtedy loggery mogą tworzyć hierarchię zgodną z hierarchią klas - to ułatwia konfigurację logowania dla różnych części aplikacji.

Logback i SLF4J

```
<dependency>
    <groupId>ch.qos.logback</groupId>
    <artifactId>logback-classic</artifactId>
    <version>1.2.3</version>
</dependency>
```

- **Simple Logging Facade for Java (SLF4J)** - API służące do logowania zdarzeń w programach Java. To tylko interfejs (API), żeby móc logować zdarzenia trzeba dodać bibliotekę która to API obsługuje
- **Logback** - jedna z najpopularniejszych bibliotek do logowania zdarzeń w aplikacjach Java. Implementuje natywnie API **SLF4J.**
- Istnieje wiele innych bibliotek do logowania: Java Logging API (**JUL**), **Log4J** (jego następcą jest właśnie **Logback**), tinylog, **Jakarta Commons Logging** / **Apache Commons Logging**

Przykład

Kod, który generuje pojedynczy log

```
public class EngineX {
    Logger logger = LoggerFactory.getLogger(getClass());

    public void startEngine(int level) {
        logger.info("Engine started with level: {}", level);
    }
    ....
```

nazwa loggera

poziom

wiadomość z parametrem

Przykład pojedynczego loga

data i godzina zdarzenia

nazwa loggera

nr linii kodu

2018-14-12 19:34:52 INFO [main] pl.sda.systemX.EngineX[13] - Engine started with level: 10

poziom

nazwa wątka

wiadomość

Konfiguracja logback - plik logback.xml

appender - miejsce gdzie trafiają logi

wzorzec loga

```
<configuration>
    <appender name="STDOUT" class="ch.qos.logback.core.ConsoleAppender">
        <encoder>
            <pattern>%d{YYYY-dd-MM HH:mm:ss.SSS} %-5level [%thread] %logger[%L] - %msg%n</pattern>
        </encoder>
    </appender>

    <root level="INFO">
        <appender-ref ref="STDOUT"/>
    </root>

    <logger name="pl.sda.logging.systemX" level="DEBUG" />
        <appender-ref ref="FILE_X"/>
    </logger>
</configuration>
```

odwołanie do appendera zdefiniowanego wyżej

konfiguracja root'a - bazowego loggera - po nim dziedziczą wszystkie inne

konfiguracja konkretnego loggera

Więcej na stronie: https://logback.qos.ch/manual/configuration.html

Poziomy logowanie i symbole we wzorcu

Poziomy logów i co powinniśmy w nich logować:

- **ERROR** - krytyczne błędy, które uniemożliwiają bądź mocno utrudniają pracę aplikacji
- **WARN** - ostrzeżenia o błędach które wystąpiły, ale zostały obsłużone i nie wpływają na pracę aplikacji
- **INFO** - informacje do audytu, zakończenie realizacji ważnego procesu
- **DEBUG** - szczegółowe informacje developerskie
- **TRACE** - bardziej szczegółowe informacje

Symbole w wzorcu:

- **%msg** - komunikat wprowadzony w kodzie aplikacji
- **%logger** - nazwa loggera który stworzył komunikat
- **%L** - nr linii w której komunikat został stworzony
- **%date** - data zdarzenia
- **%level** - poziom loga
- **%thread** - nazwa wątku z którego wyszedł komunikat
- **%n** - symbol nowego wiersza

Więcej na stronie: https://logback.qos.ch/manual/layouts.html

Zadania

#logging

Zadania

#logging

1. Dodaj do wszystkich metod klasy pl.sda.mini_project.**BooksManager** szczegółową informację (poziom DEBUG) o wykonaniu
2. Dodaj do klasy z pkt 1 logowanie błędu gdy user próbuje usunąć książkę której id nie ma na liście.
3. Dodaj do klasy z pkt 1 logowanie informacji zaraz po tym jak klasa zostanie zainicjalizowana w konstruktorze.
4. Skonfiguruj logbacka tak żeby wypisywał na konsolę wszystkie logi
5. Dodaj w konfiguracji zapis do pliku - tylko logów z poziomem WARN i wyżej

Pamiętaj o wykorzystaniu repozytorium kodu Git!

Pytania?

Spotkanie #5

Programowanie I
poziom podstawowy

Rozkład jazdy

Sortowanie

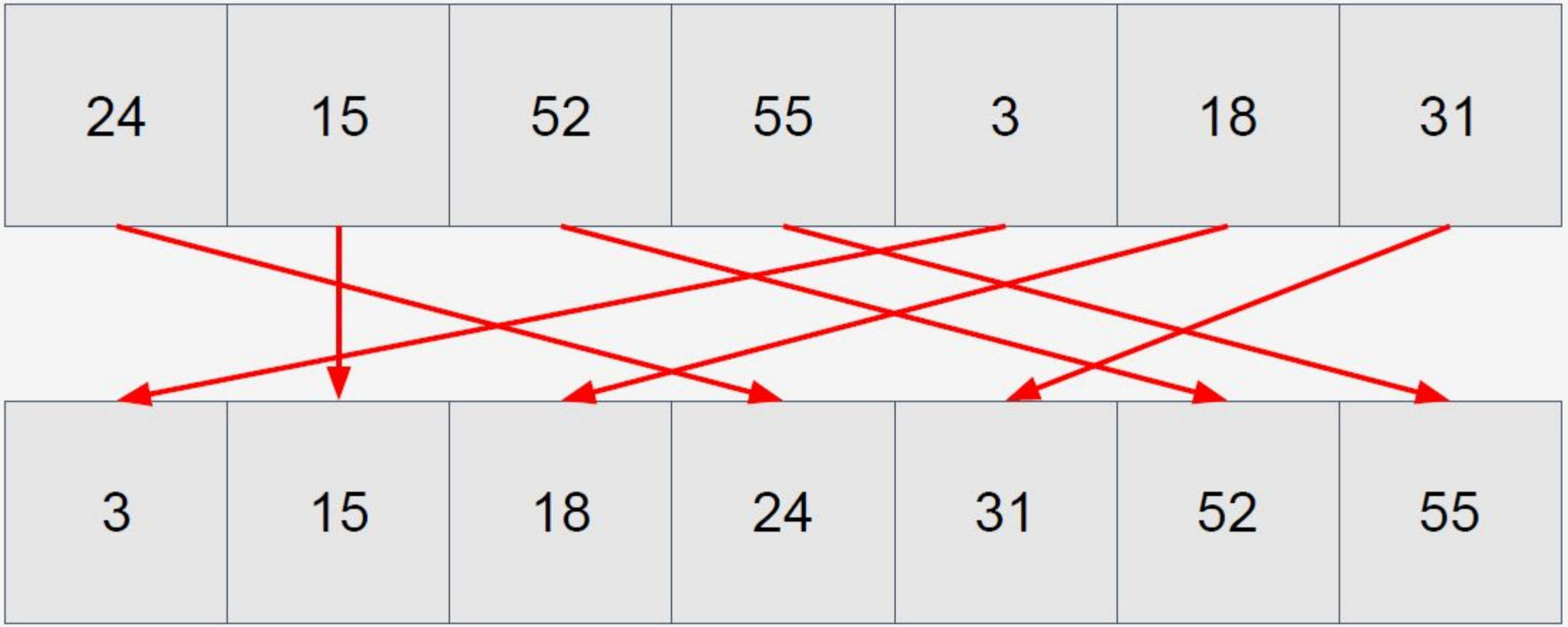
Sortowanie danych jest jednym z podstawowych problemów programowania komputerów, z którym prędzej czy później spotka się każdy programista. Proces ten polega na uporządkowaniu zbioru danych względem pewnych cech charakterystycznych dla każdego elementu tego zbioru.

sport - wyniki uzyskane przez poszczególnych zawodników należy ułożyć w określonej kolejności, aby wyłonić zwycięzcę oraz podać lokatę każdego zawodnika

grafika - wiele algorytmów graficznych wymaga porządkowania elementów, np. ścian obiektów ze względu na odległość od obserwatora, uporządkowanie takie pozwala później określić, które ze ścian są zakrywane przez inne ściany dając w efekcie obraz trójwymiarowy

bazy danych - informacja przechowywana w bazie danych może wymagać różnego rodzaju uporządkowania, np. lista książek może być alfabetycznie porządkowana wg autorów lub tytułów, co znacznie ułatwia znalezienie określonej pozycji

W jaki sposób sortuje człowiek, a w jaki sposób komputer?



https://bost.ocks.org/mike/algorithms/

Rodzaje algorytmów

https://www.cs.usfca.edu/~galles/visualization/ComparisonSort.html
https://visualgo.net/en/sorting

stabilne - gdy po posortowaniu elementy o równej wartości będą występowały w takiej samej kolejności, jaką miały w zbiorze nieposortowanym

- sortowanie bąbelkowe (ang. bubble sort) – **$O(n^2)$**
- sortowanie przez wstawianie (ang. insertion sort) – **$O(n^2)$**
- sortowanie przez scalanie (ang. merge sort) – **O(nlog n)** wymaga **O(n)** dodatkowej pamięci
- sortowanie przez zliczanie (ang. counting sort) – **O(n+k)** wymaga **O(n+k)** dodatkowej pamięci

niestabilne - gdy po posortowaniu elementy o równej wartości nie będą występowały w takiej samej kolejności, jaką miały w zbiorze nieposortowanym

- sortowanie przez wybieranie (ang. selection sort) **$O(n^2)$**
- sortowanie szybkie – (ang. quicksort) **O(nlog n)** pesymistyczny **$O(n^2)$**
- sortowanie przez kopcowanie – (ang. heapsort) **O(nlog n)**

Kolekcje w Java domyślnie wykorzystują merge sort

Sortowanie bąbelkowe (ang. bubble sort)

Prosta metoda sortowania o złożoności czasowej $O(n^2)$ i pamięciowej $O(1)$. Oznacza to, że sortowanie bąbelkowe nie poradzi sobie z porządkowaniem większych zbiorów.

Polega na porównywaniu dwóch kolejnych elementów i zamianie ich kolejności, jeżeli zaburza ona porządek, w jakim się sortuje tablicę. Sortowanie kończy się, gdy podczas kolejnego przejścia nie dokonano żadnej zmiany.

```
procedure bubbleSort( A : lista elementów do posortowania )
  n = liczba_elementów(A)
   do
    for (i = 0; i < n-1; i++) do:
      if A[i] > A[i+1] then
        swap(A[i], A[i+1])
      end if
    end for
    n = n-1
  while n > 1
end procedure
```

$$[\underbrace{4,2}_{4>2},5,1,7] \rightarrow [2,\underbrace{4,5}_{4<5},1,7] \rightarrow [2,4,\underbrace{5,1}_{5>1},7] \rightarrow [2,4,1,\underbrace{5,7}_{5<7}]$$

$$[\underbrace{2,4}_{2<4},1,5,7] \rightarrow [2,\underbrace{4,1}_{4>1},5,7] \rightarrow [2,1,\underbrace{4,5}_{4<5},7]$$

$$[\underbrace{2,1}_{2>1},4,5,7] \rightarrow [1,\underbrace{2,4}_{2<4},5,7]$$

$$[\underbrace{1,2}_{1<2},4,5,7]$$

https://youtu.be/lyZQPjUT5B4

Podział na grupy

Alpha, Beta, Gamma, Delta, Epsilon

Sortowanie bąbelkowe - zadania

1. Napisz metodę, która generować będzie tablicę N liczb całkowitych z zakresu A .. B.
2. Napisz metodę wykorzystującą algorytm sortowania bąbelkowego w oparciu o tablice.
3. Wygeneruj różnej wielkości tablice losowych liczb [10, 100, 1000, 1000000] i wykorzystaj metodę do posortowania wartości. Sprawdź czas działania dla poszczególnych sortowań.
4. Napisz testy sprawdzające poprawność działania algorytmu.

Testy:

- should sort array ascending by default
- should sort array depending on the order
- should leave sorted array unchanged
- should throw exception if array is null
- should not throw exception if array is zero-sized
- should sort if elements are not unique

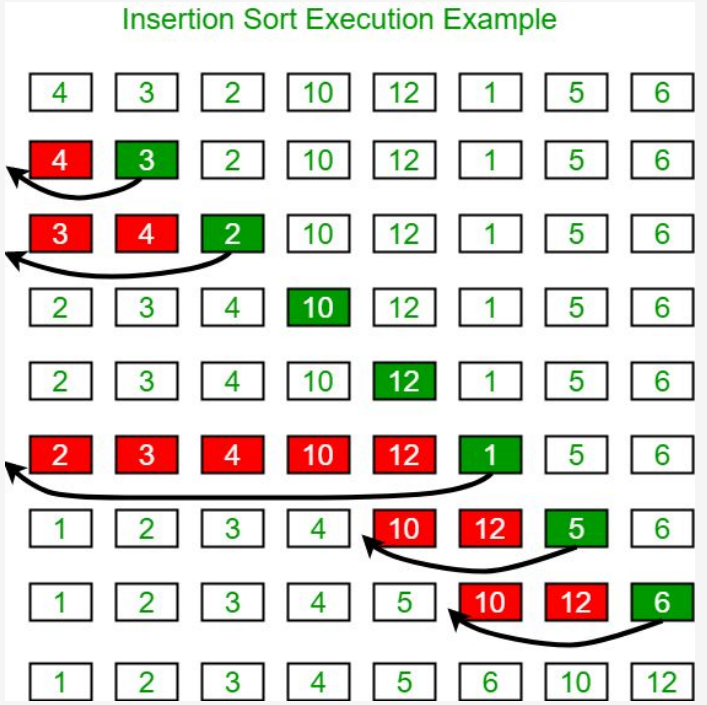
Dodatkowo: https://pl.spoj.com/problems/FR_01_09/

Sortowanie przez wstawianie (ang. insertion sort)

Jest efektywny dla niewielkiej liczby elementów, a jego złożoność wynosi $O(n^2)$. Intuicyjnie można porównać do układania kolejnych kart w talii. Najpierw bierzemy pierwszą kartę. Następnie pobieramy kolejne, aż do wyczerpania talii. Każdą pobraną kartę porównujemy z kartami, które już trzymamy w ręce i szukamy dla niej miejsca. Gdy znajdziemy takie miejsce, rozsuwamy karty i nową wstawiamy na przygotowane w ten sposób miejsce. Operacja jest powtarzana, aż cały zbiór zostanie posortowany.

A – tablica danych przeznaczonych do posortowania (indeksowana od 1 do n)
n – liczba elementów w tablicy A

```
0. Insert_sort(A, n)

1.  for i=2 to n :
2.      klucz = A[i]
3       # Wstaw A[i] w posortowany ciąg A[1 ... i-1]
4.      j = i - 1

5.      while j>0 and A[j]>klucz:
6.          A[j + 1] = A[j]
7.          j = j - 1
8.      A[j + 1] = klucz
```



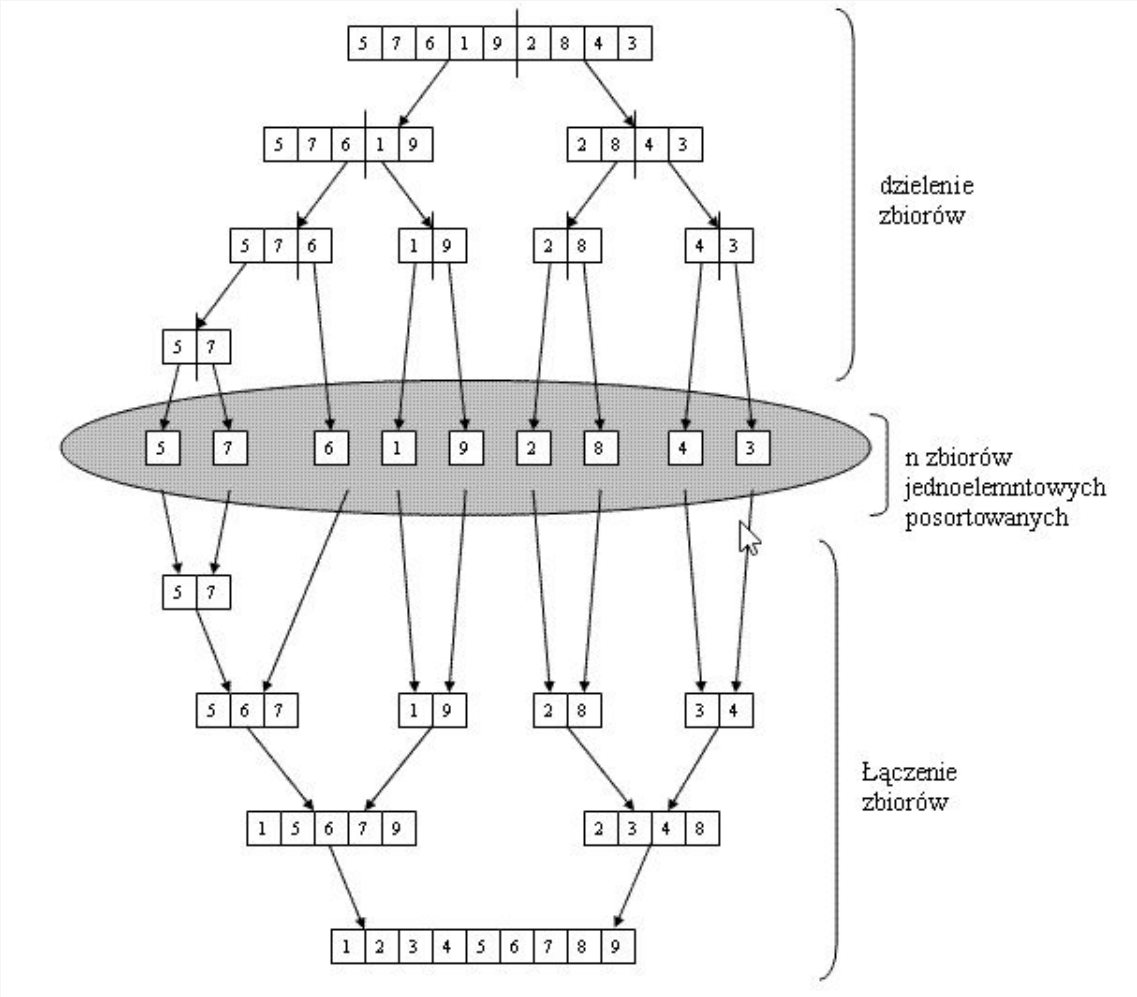
https://youtu.be/ROaIU379l3U

Sortowanie przez scalanie (ang. merge sort)

Rekurencyjny algorytm sortowania danych, stosujący metodę dziel i zwyciężaj. Złożoność obliczeniowa wynosi **O(nlog n)**, ale wymaga przy tym **O(n)** dodatkowej pamięci.

Wyróżnić można trzy podstawowe kroki:

1) podziel zestaw danych na dwie części
2) zastosuj sortowanie przez scalanie dla każdej z nich oddzielnie, chyba że pozostał już tylko jeden element
3) połącz posortowane podciągi w jeden ciąg posortowany



https://youtu.be/XaqR3G_NVoo

Sortowanie przez zliczanie (ang. counting sort)

Metoda sortowania danych, która polega na sprawdzeniu ile wystąpień kluczy mniejszych od danego występuje w sortowanej tablicy. Jej złożoność obliczeniowa wynosi **O(n+k)** - gdzie n jest liczbą elementów zbioru do posortowania, a k jest ilością znaków zastosowanego przedziału, ale wymaga **O(n+k)** dodatkowej pamięci. Algorytm zakłada, że klucze elementów należą do skończonego zbioru (np. są to znaki z przedziału A - Z), co ogranicza możliwości jego zastosowania.

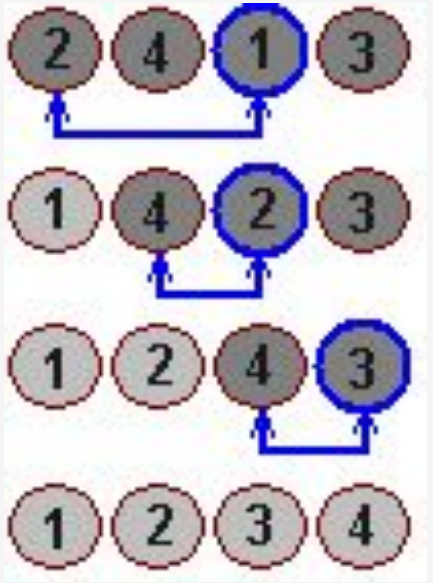
Zobrazowanie algorytmu działania:
https://www.youtube.com/watch?v=7zuGmKfUt7s

Sortowanie przez wybieranie (ang. selection sort)

Jedna z prostszych metod sortowania o złożoności **O(n^2)**. Polega na wyszukaniu elementu mającego się znaleźć na żądanej pozycji i zamianie miejscami z tym, który jest tam obecnie. Operacja jest wykonywana dla wszystkich indeksów sortowanej tablicy.

Algorytm przedstawia się następująco:
1) wyszukaj minimalną wartość z tablicy spośród elementów od i do końca tablicy
2) zamień wartość minimalną, z elementem na pozycji i

Gdy zamiast wartości minimalnej wybierana będzie maksymalna, wówczas tablica będzie posortowana od największego do najmniejszego elementu.



https://youtu.be/Ns4TPTC8whw

Sortowanie szybkie (ang. quicksort)

Algorytm sortowania szybkiego jest uważany za najszybszy algorytm dla danych losowych.
Zasada jego działania opiera się o metodę dziel i zwyciężaj. Zbiór danych zostaje podzielony na dwa podzbiory i każdy z nich jest sortowany niezależnie od drugiego. Złożoność obliczeniowa wynosi **O(nlog n)** (pesymistyczna **O(n^2)**).

```
Quicksort(A as array, low as int, high as int) {
    if (low < high) {
        pivot_location = Partition(A, low, high)
        Quicksort(A, low, pivot_location - 1)
        Quicksort(A, pivot_location + 1, high)
    }
}
```

```
Partition(A as array, low as int, high as int) {
    pivot = A[high]
    i = low - 1

    for j = low to high {
        if (A[j] <= pivot) then {
            i = i + 1
            swap(A[j], A[i])
        }
        j = j + 1
    }

    i = i + 1
    swap(A[high], A[i])
    return i
}
```

https://youtu.be/ywWBy6J5gz8

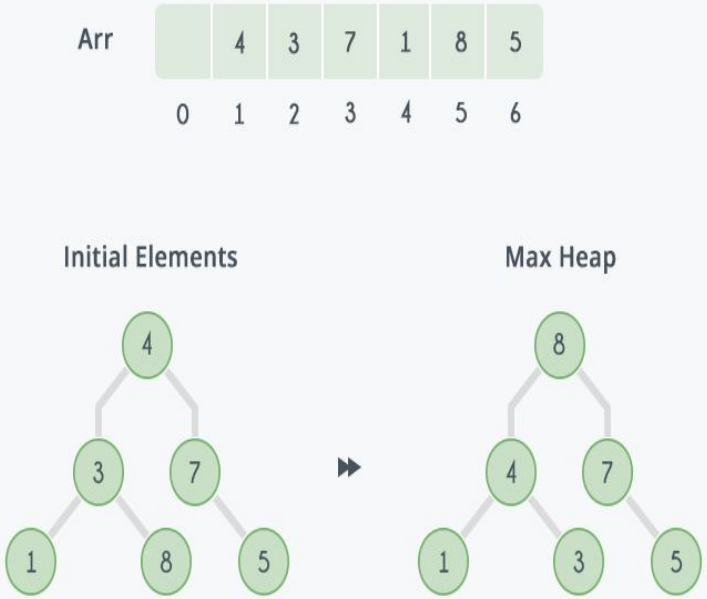
Sortowanie przez kopcowanie (ang. heapsort)

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

Zwane również sortowaniem stogowym – choć niestabilny, to jednak szybki i niepochłaniający wiele pamięci (złożoność czasowa wynosi **O(nlog n)**, a pamięciowa – **O(n)**

```
Heapsort(A) {
   BuildHeap(A)
   for i <- length(A) downto 2 {
      exchange A[1] <-> A[i]
      heapsize <- heapsize -1
      Heapify(A, 1)
}

BuildHeap(A) {
   heapsize <- length(A)
   for i <- floor( length/2 ) downto 1
      Heapify(A, i)
}
```

```
Heapify(A, i) {
   le <- left(i)
   ri <- right(i)
   if (le<=heapsize) and (A[le]>A[i])
      largest <- le
   else
      largest <- i
   if (ri<=heapsize) and (A[ri]>A[largest])
      largest <- ri
   if (largest != i) {
      exchange A[i] <-> A[largest]
      Heapify(A, largest)
   }
}
```



https://youtu.be/Xw2D9aJRBY4

Podział na grupy

Grupa Alpha -> sortowanie przez wstawianie
Grupa Beta -> sortowanie przez scalanie
Grupa Gamma -> sortowanie przez zliczanie
Grupa Delta -> sortowanie szybkie
Grupa Epsilon -> sortowanie przez wybieranie

Sortowania - zadania

1. Napisz metodę wykorzystującą algorytm sortowania [???].
2. Wygeneruj różnej wielkości tablice losowych liczb [10, 100, 1000, 1000000] i wykorzystaj metodę do posortowania wartości. Sprawdź czas działania dla poszczególnych sortowań.
3. Napisz testy sprawdzające poprawność działania algorytmu.
4. Porównaj z innymi grupami czasy działania Waszych algorytmów.

Testy:

- should sort array ascending by default
- should sort array depending on the order
- should leave sorted array unchanged
- should throw exception if array is null
- should not throw exception if array is zero-sized
- should sort if elements are not unique

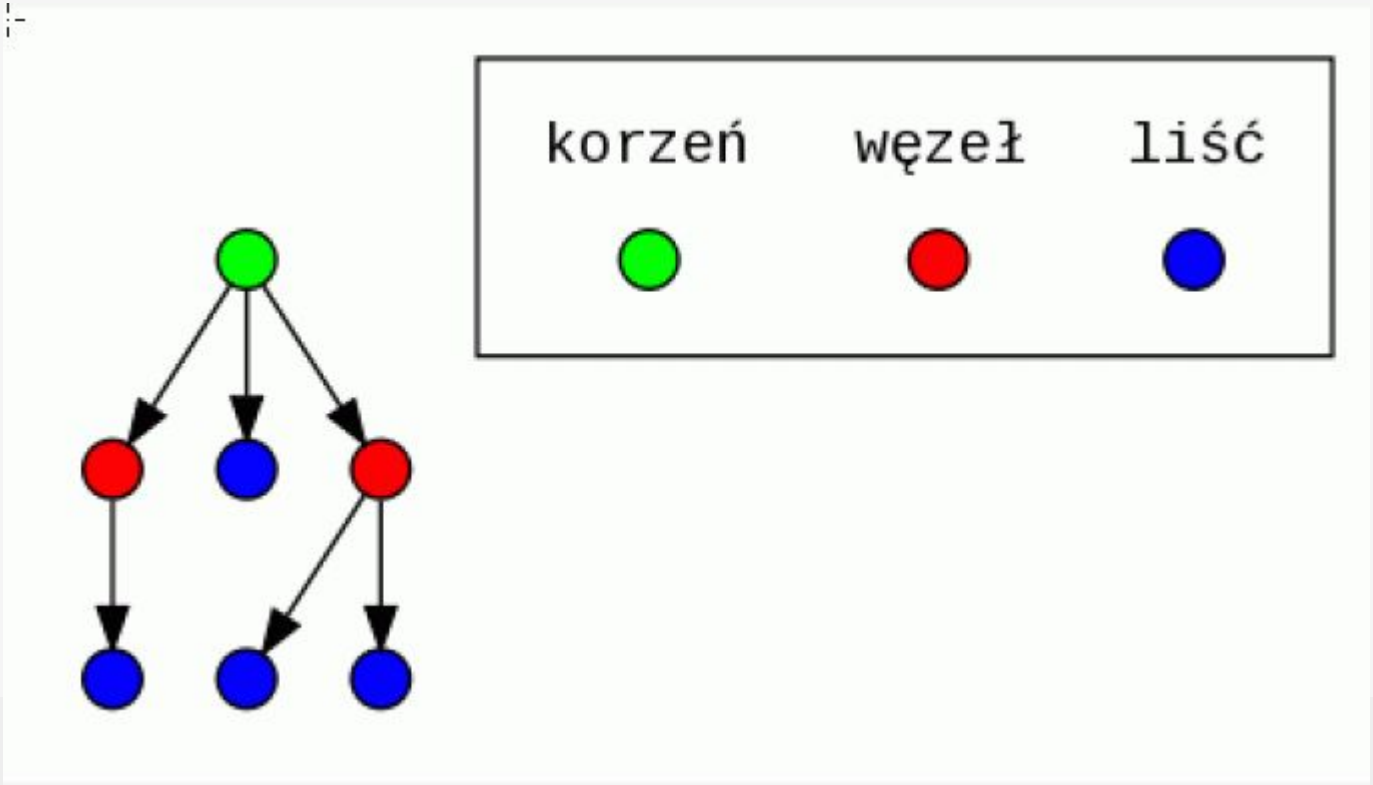
Zadania dodatkowe

1. https://www.spoj.com/problems/TSORT/
2. https://pl.spoj.com/problems/JSORTBIZ/
3. https://pl.spoj.com/problems/XYZSORT/
4. https://pl.spoj.com/problems/SBANK/
5. https://www.spoj.com/problems/CODESPTB/
6. * https://www.spoj.com/problems/SSORT/ -> http://isolvedaproblem.blogspot.com/2012/02/silly-sort.html

Drzewa

Drzewa

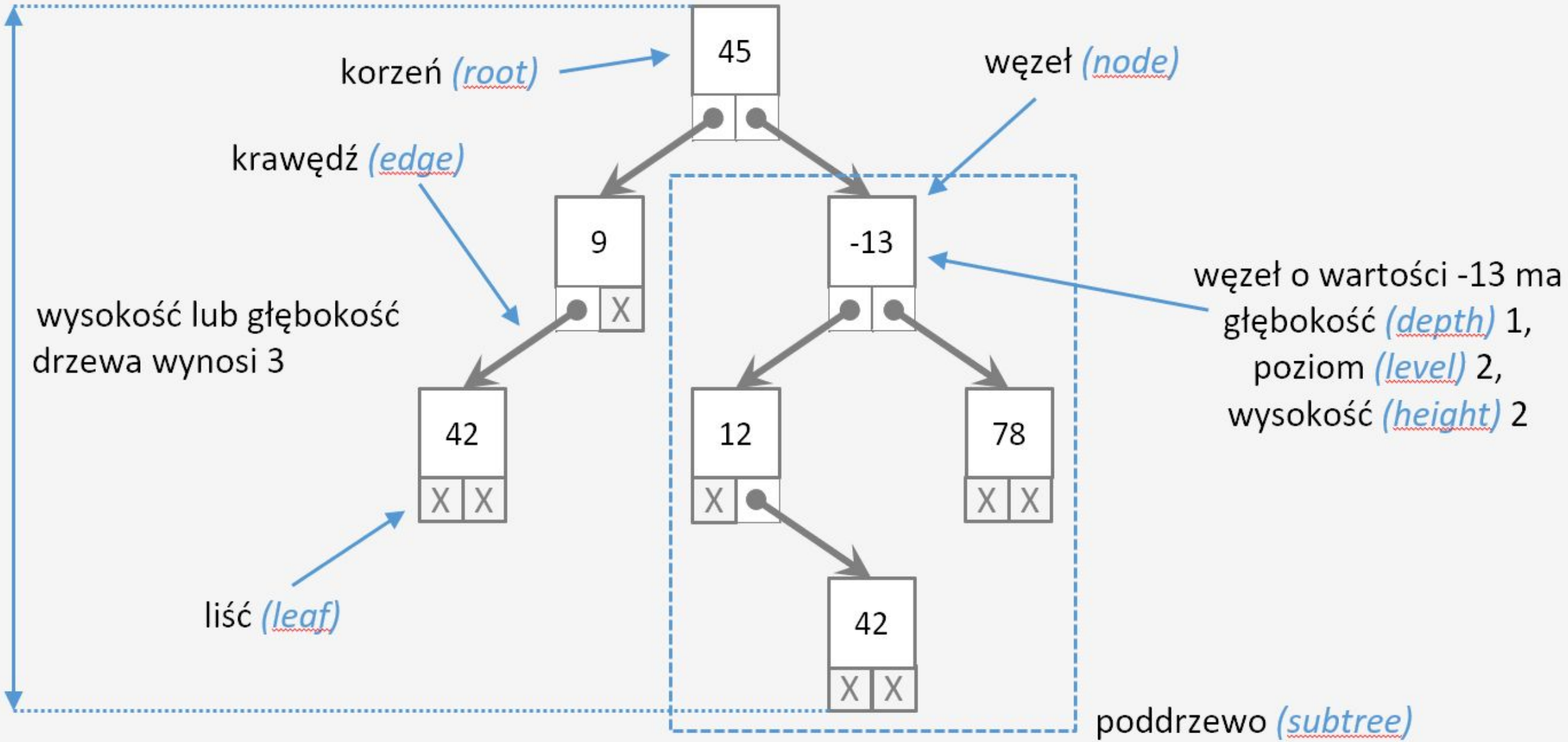
Drzewo jest bardziej skomplikowaną strukturą niż poprzednie. Dla każdego drzewa wyróżniony jest jeden, charakterystyczny element - korzeń. Korzeń jest jedynym elementem drzewa, który nie posiada elementów poprzednich. Dla każdego innego elementu określony jest dokładnie jeden element poprzedni. Dla każdego elementu, oprócz ostatnich, tzw. liści, istnieje co najmniej 1 element następny. Jeżeli liczba następnych elementów wynosi nie więcej niż 2 to drzewo nazywamy binarnym, jeżeli natomiast liczba elementów wynosi dokładnie 2 to drzewo nazywamy pełnym drzewem binarnym.



Drzewa - pojęcia

- zbudowane z węzłów i to w nich trzymane są dane
- węzły są ze sobą powiązane w sposób hierarchiczny za pomocą krawędzi
- pierwszy węzeł drzewa nazywa się korzeniem
- od niego "wyrastają" pozostałe węzły, które będziemy nazywać synami
- synowie są węzłami podrzędnymi w strukturze hierarchicznej
- synowie tego samego ojca są nazywani braćmi
- węzeł nadrzędny w stosunku do syna nazwiemy ojcem
- ojcowie są węzłami nadrzędnymi w strukturze hierarchicznej
- jeśli węzeł nie posiada synów, to nazywa się liściem
- jeśli węzeł posiada syna nazywa się węzłem wewnętrznym

Drzewa


korzeń (root)
węzeł (node)
krawędź (edge)
wysokość lub głębokość drzewa wynosi 3
węzeł o wartości -13 ma głębokość (depth) 1, poziom (level) 2, wysokość (height) 2
liść (leaf)
poddrzewo (subtree)
45
9
-13
42
12
78
42
X

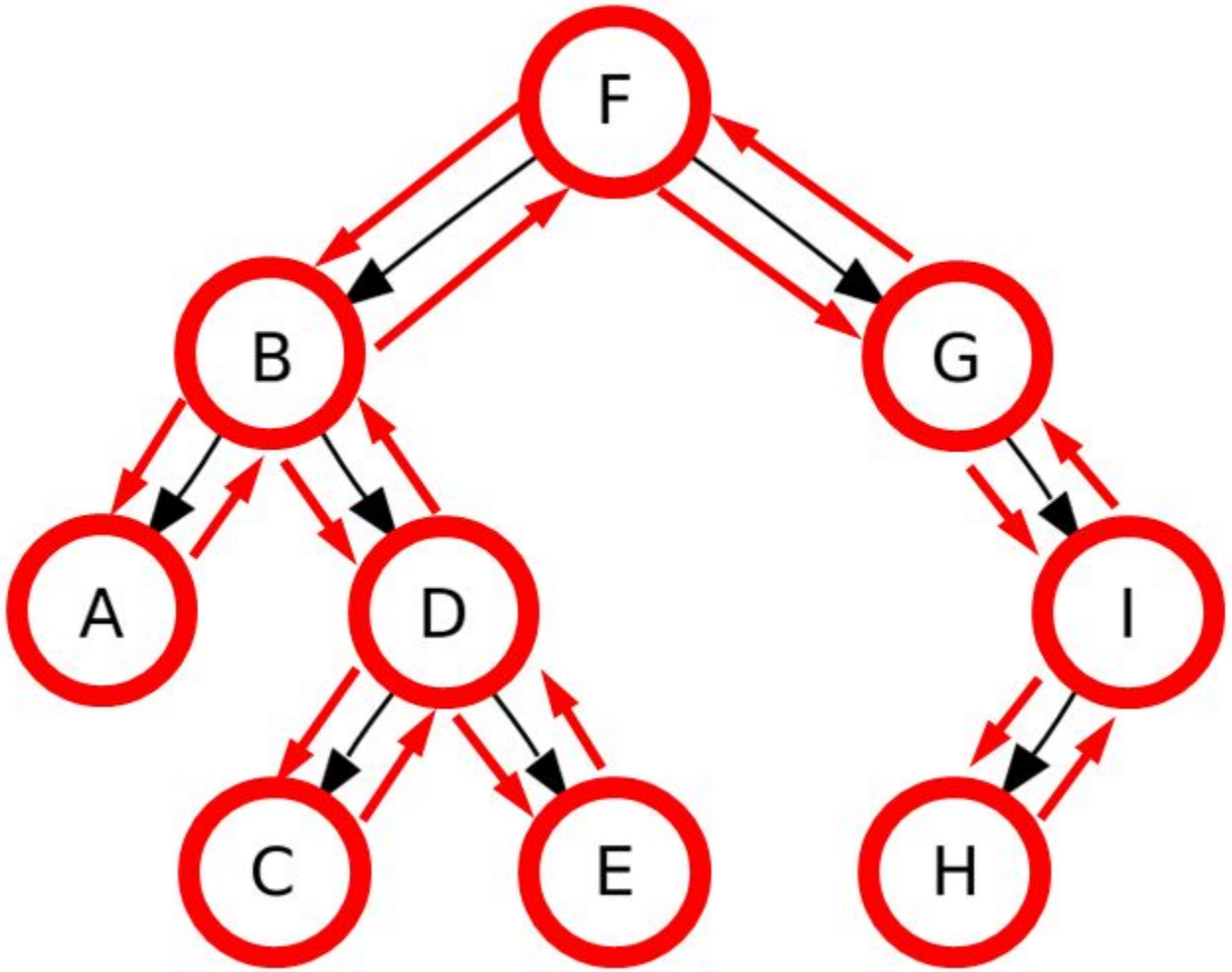
Drzewa - przykładowe operacje

- wyszukiwanie elementu
- dodanie elementu
- usunięcie elementu
- znajdywanie najniższego wspólnego przodka dwóch węzłów
- przechodzenie drzewa (tree traversal)
  - wzdłużne (pre-order)
  - poprzeczne (in-order)
  - wsteczne (post-order)
  - poziomami (level-order)

Drzewa - przejście wzdłużne (pre-order)

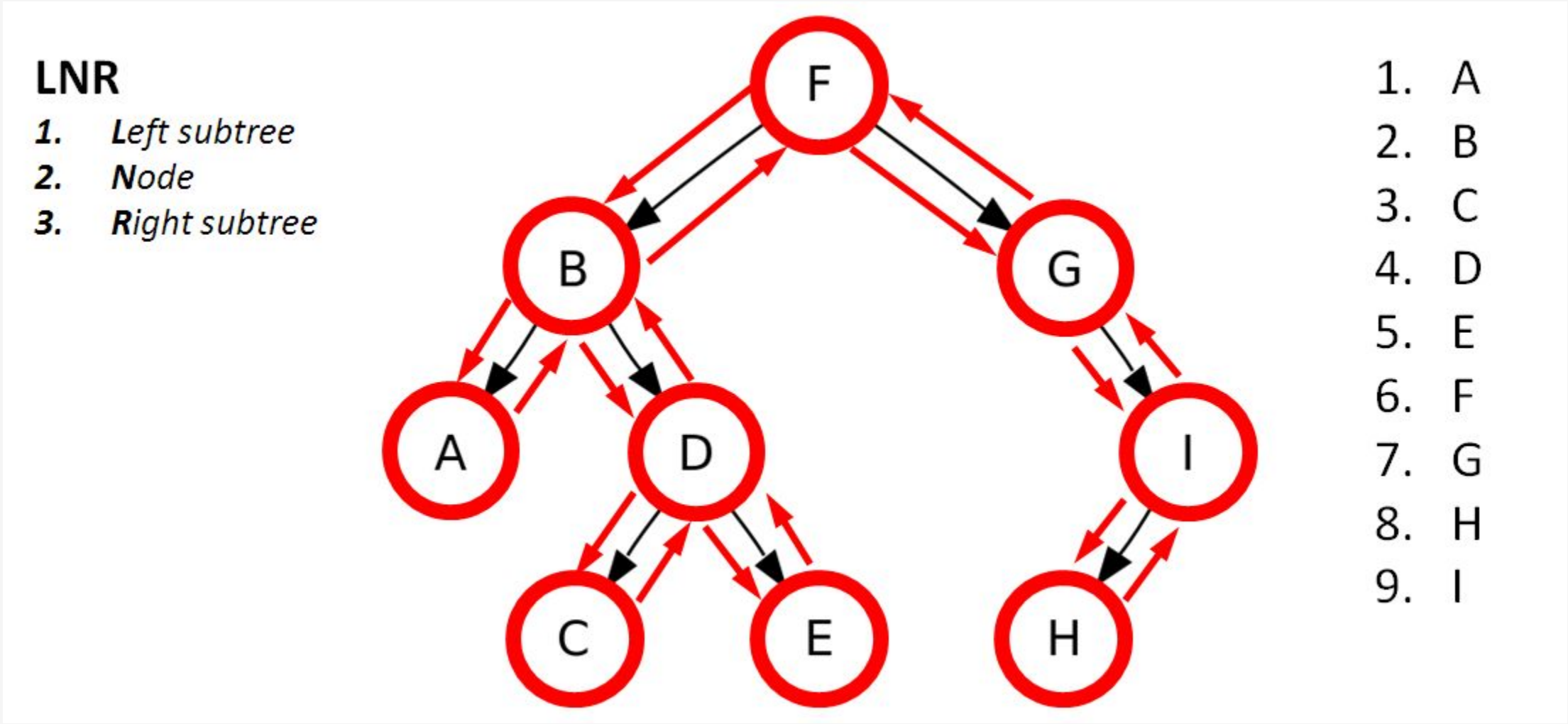
NLR

1. ***N**ode*
2. ***L**eft subtree*
3. ***R**ight subtree*

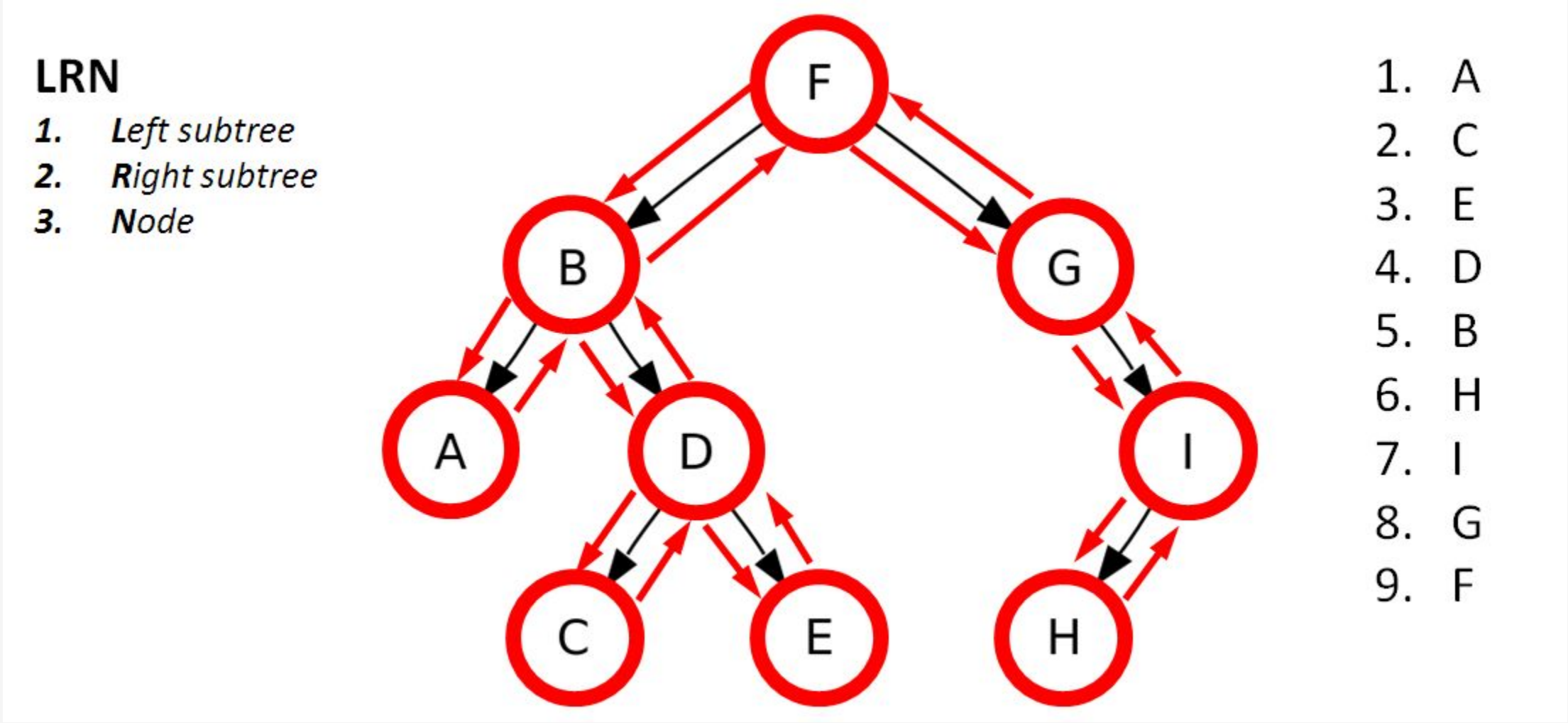


1. F
2. B
3. A
4. D
5. C
6. E
7. G
8. I
9. H

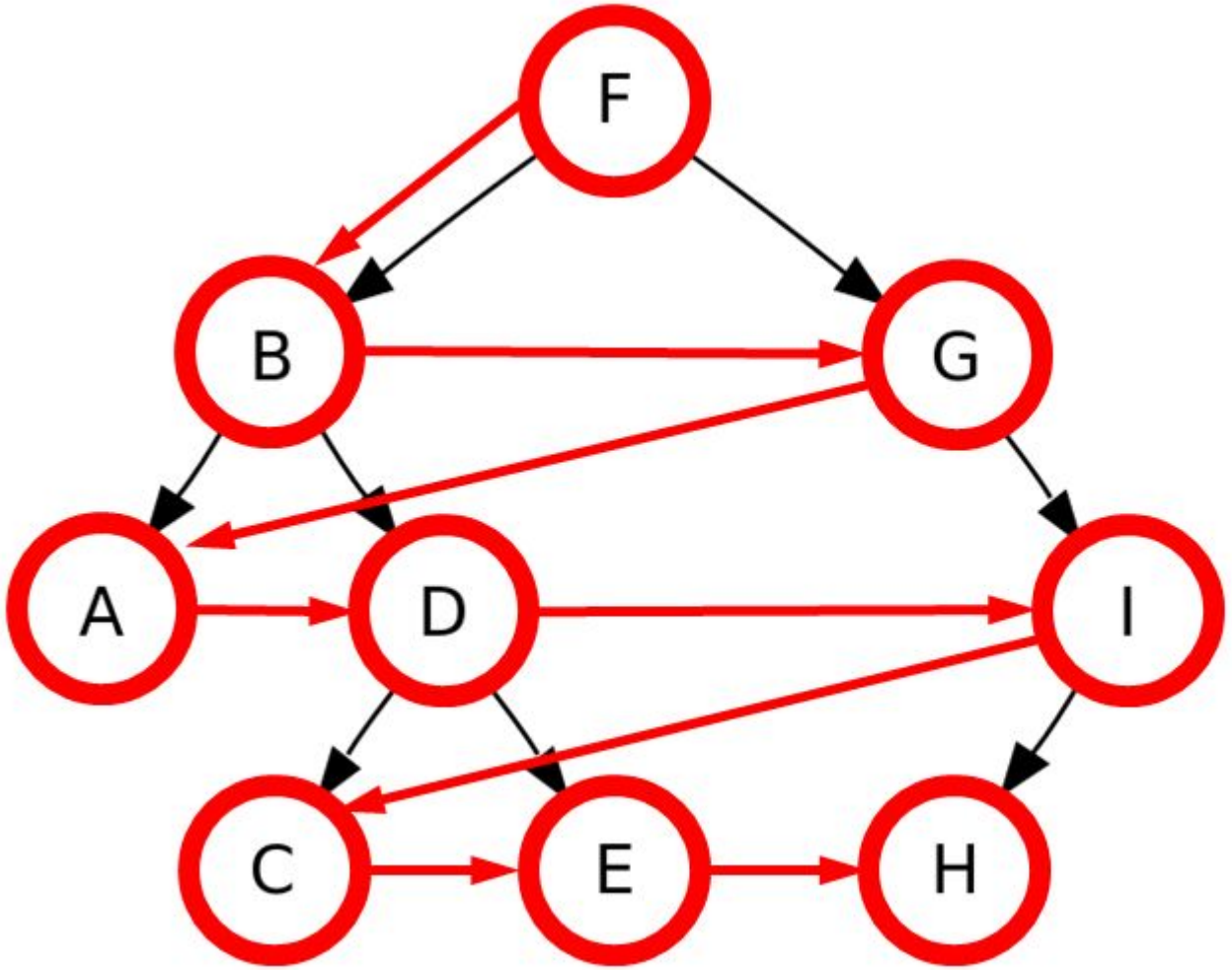
Drzewa - przejście poprzeczne (in-order)



Drzewa - przejście wsteczne (post-order)

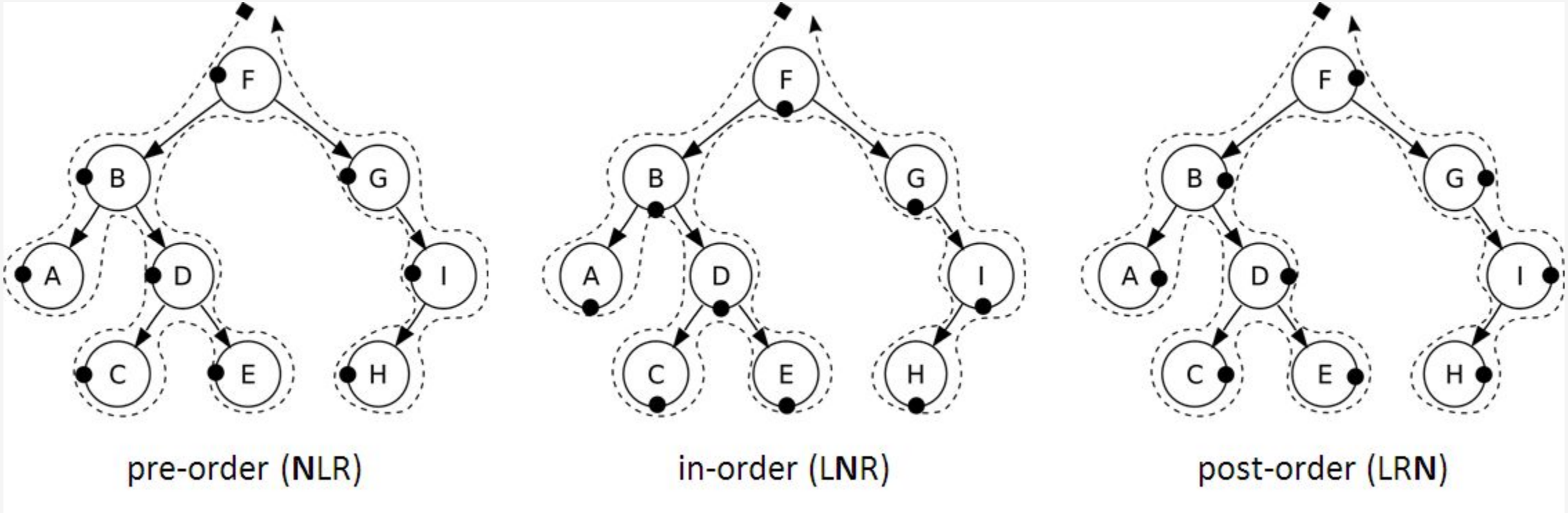


Drzewa - przejście poziomami (level-order)



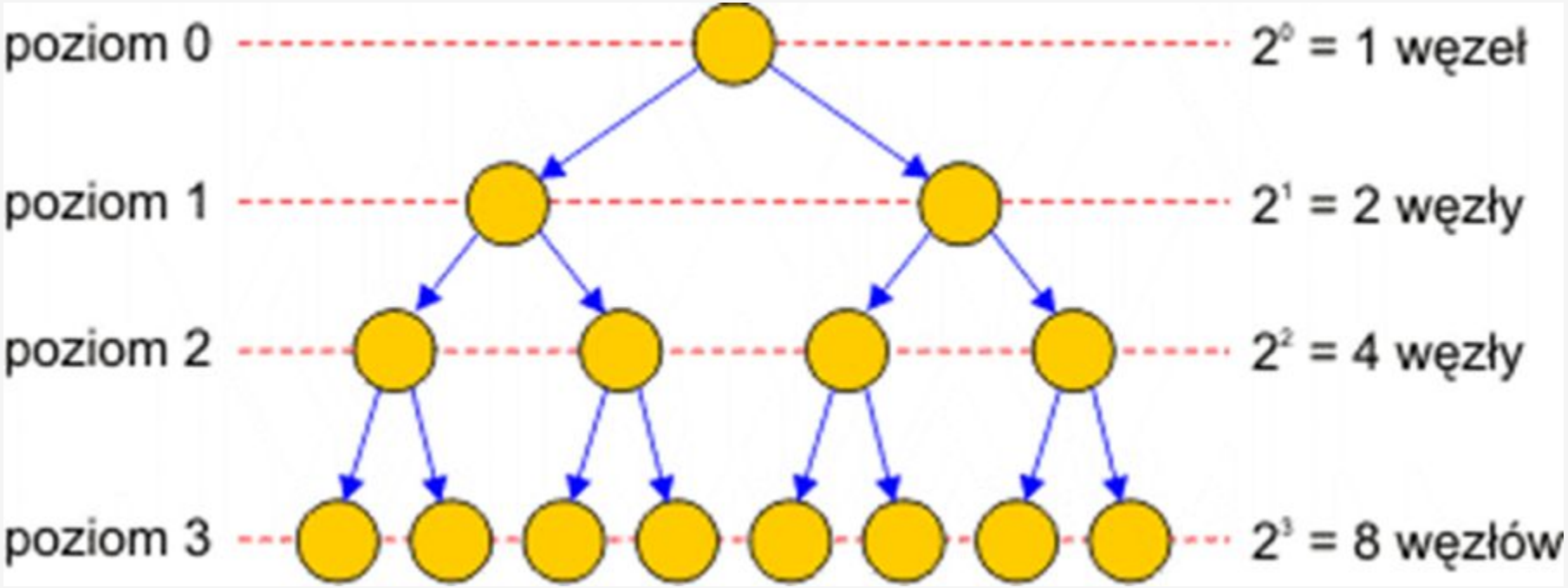
1. F
2. B
3. G
4. A
5. D
6. I
7. C
8. E
9. H

Drzewa - porównanie przejść



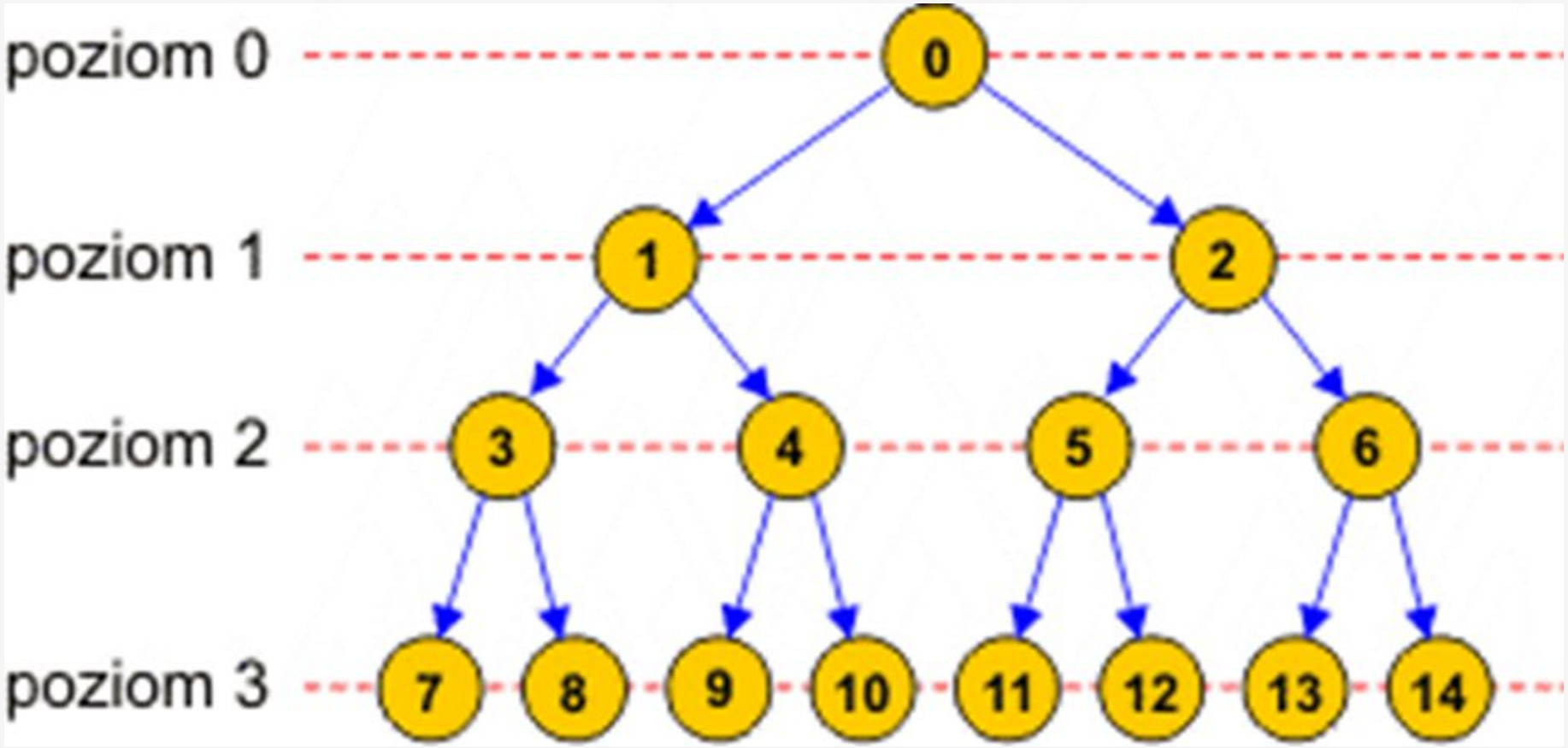
Drzewa binarne

Dla regularnego drzewa binarnego liczba węzłów na poziomie k-tym jest zawsze równa 2^k
Liczba wszystkich węzłów, czyli rozmiar drzewa jest równa $2^p - 1$, gdzie p oznacza liczbę poziomów



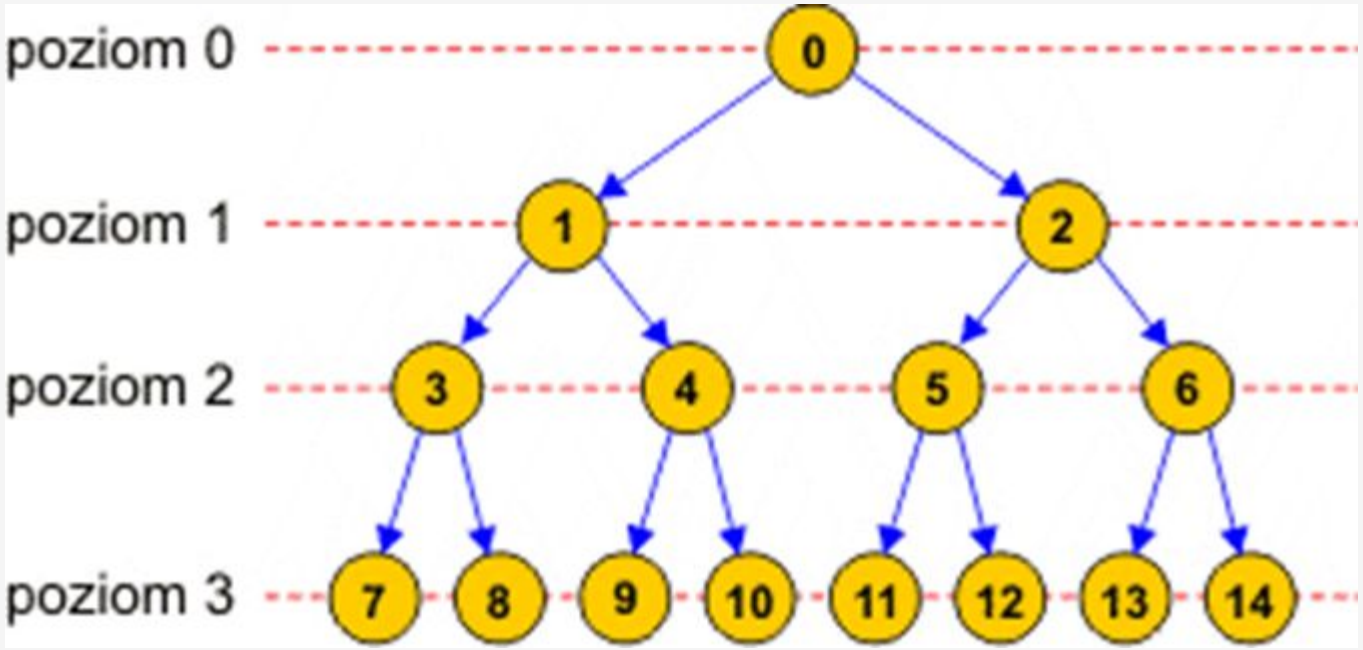
Drzewa binarne

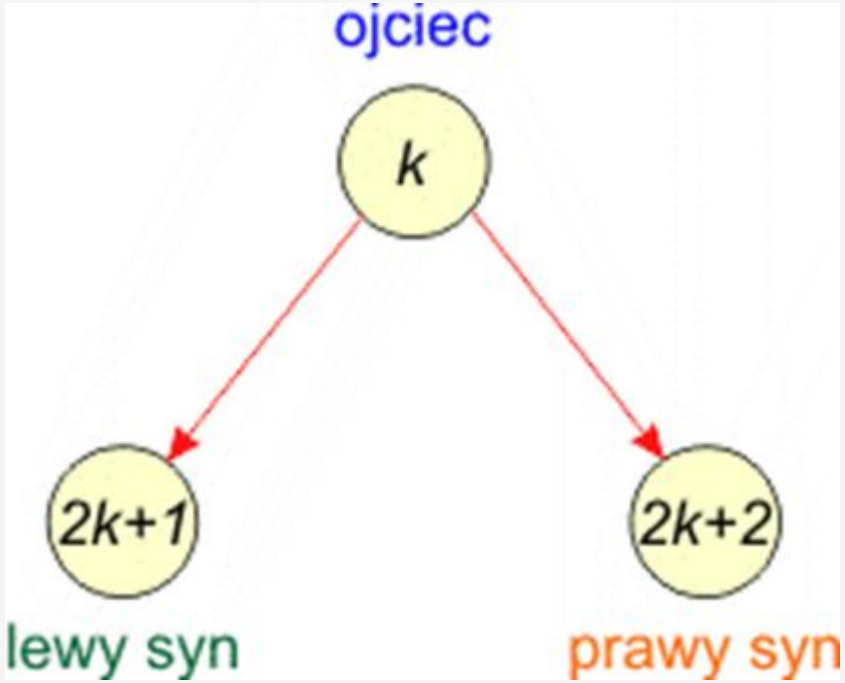
Każdy z węzłów można ponumerować idąc od lewej do prawej



Drzewa binarne

Numery węzłów są powiązane ze strukturą hierarchii drzewa prostymi zależnościami





Zadania - https://visualgo.net/en/bst

1. Napisz klasy Tree i Node, które pozwolą zbudować binarne drzewo poszukiwań (ang. Binary Search Tree (**BST**)) przechowujące wartości typu int.
2. Napisz metodę przechodzącą drzewo systemem (zacznij od pseudokodu):
   a. in-order
   b. * pre-order
   c. * post-order
3. * Napisz metodę zwracającą liczbę liści w drzewie (pseudokod na następnym slajdzie)
4. * Napisz metodę, która obliczy wysokość drzewa (pseudokod na następnym slajdzie)
5. * Napisz testy sprawdzające Twój kod.

Pamiętaj o wykorzystaniu repozytorium kodu Git! * Dla chętnych.

Pseudokod

getLeafCount(node)

1) If node is NULL then return 0.
2) Else If left and right child nodes are NULL return 1.
3) Else recursively calculate leaf count of the tree using below formula.
   Leaf count of a tree = Leaf count of left subtree + Leaf count of right subtree

maxDepth(root)

1. If tree is empty then return 0
2. Else
   (a) Get the max depth of left subtree recursively i.e., call maxDepth(tree -> left-subtree)
   (a) Get the max depth of right subtree recursively i.e., call maxDepth(tree -> right-subtree)
   (c) Get the max of max depths of left and right subtrees and add 1 to it for the current node.
      max_depth = max(max dept of left subtree, max depth of right subtree) + 1
   (d) Return max_depth

Spotkanie #6

Programowanie I
poziom podstawowy

Rozkład jazdy

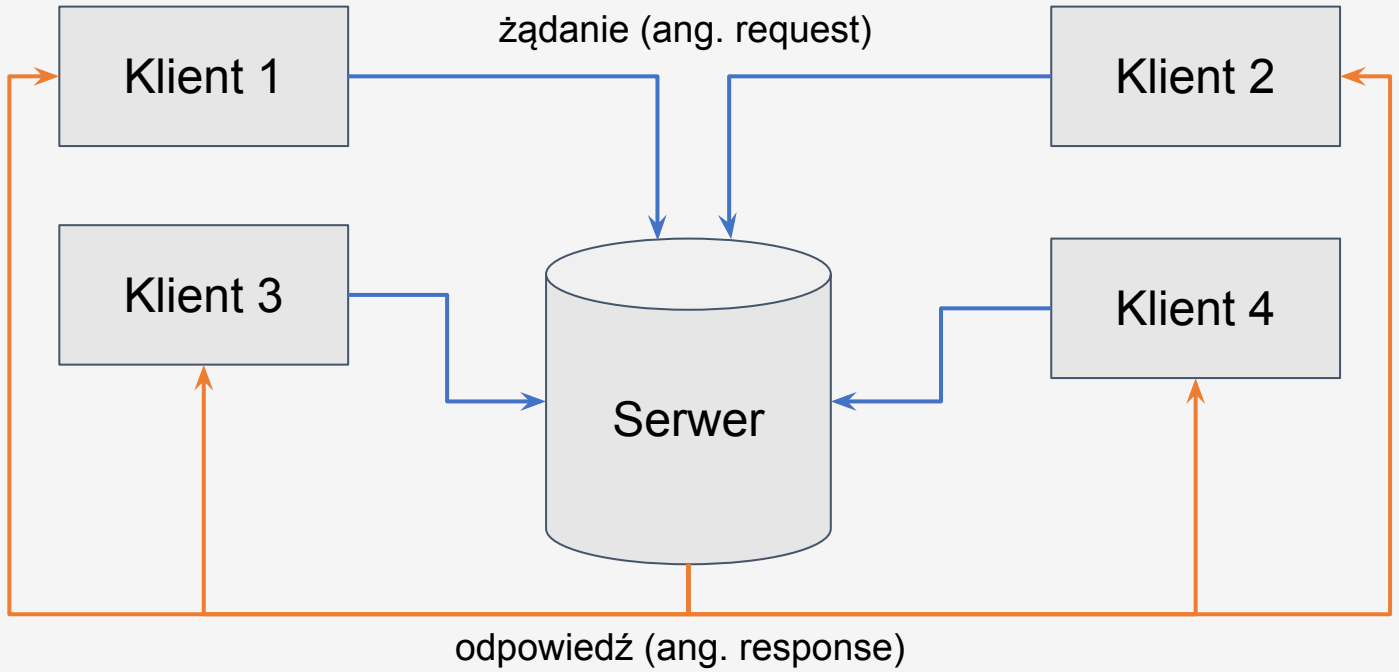
09:00 - powtórka -TEST
09:30 - projekt **R-A-V**
10:30 - przerwa krótka
10:35 - projekt **R-A-V**
12:00 - przerwa długa
12:30 - podsumowanie testu
13:00 - projekt **R-A-V**
14:30 - przerwa krótka
14:35 - projekt **R-A-V**

Test

Podsumowanie w praktyce

Architektura wielowarstwowa

Architektura wielowarstwowa (ang. multi-tier architecture/n-tier architecture)
architektura komputerowa (typu klient-serwer), która definiuje kilka niezależnych warstw: interfejsu użytkownika, przetwarzania danych i składowania danych.
Każda z warstw może być oddzielnie rozwijana i aktualizowana, co ułatwia ich utrzymanie i nie wpływa negatywnie na funkcjonowanie pozostałych warstw.



- **architektura dwuwarstwowa** – przetwarzanie i składowanie danych odbywa się w jednym module
- **architektura trójwarstwowa** – przetwarzanie i składowanie danych następuje w dwóch osobnych modułach
- **architektura wielowarstwowa** – przetwarzanie, składowanie i inne operacje na danych odbywają się w wielu osobnych modułach.

Architektura trójwarstwowa

- interfejs użytkownika - interakcja z użytkownikiem: wyświetlanie danych i przyjmowanie żądań
- front-end
- HTML5, JavaScript, CSS

- przetwarzanie danych, wdrażanie reguł biznesowych, przekazywanie danych między warstwami
- back-end
- Java, .NET, C#, Python, C++

- baza danych, przechowywanie, pobieranie i zapisywanie danych
- back-end
- MySQL, Oracle, PostgreSQL, Microsoft SQL Server, MongoDB

Projektowanie aplikacji - podejście bottom-up

- Zaczynamy od zdefiniowania podstawowych danych/encji które powinny wejść w skład aplikacji
- Tworzymy z nich **model domeny/dziedziny** (ang. domain model)
- Do każdego z elementów modelu dziedziny tworzymy klasy: nadajemy im nazwy i tworzymy pola z danymi
- Dodajemy funkcjonalności od prostych po bardziej skomplikowane
- Wprowadzamy relacje między obiektami, niejako przenosimy się jeden stopień wyżej i na podstawie tego co zrobiliśmy w poprzednich krokach budujemy dalsze funkcjonalności
- Tworzymy bazę danych na podstawie modelu dziedziny. Mapujemy klasy/obiekty na tabele w bazie danych. Dodajemy kod do obsługi połączenia z bazą.
- Tworzymy interfejs użytkownika i zapełniamy go danymi, które stworzyliśmy wcześniej

Projekt R-A-V - Rent A Vehicle

Wymagania klienta:

1. Potrzebuję program, który będzie umożliwiał wynajem różnego rodzaju pojazdów przez moich klientów.
2. Przykładowe rodzaje pojazdów: samochód, motorówka, amfibia. W przyszłości mogą dojść inne.
3. Pojazdy będą podzielone na trzy główne kategorie: pojazdy jeżdżące, latające i pływające. Może się zdarzyć że jeden pojazd będzie zawierał się w więcej niż jednej kategorii (np. amfibia)
4. Każda z kategorii ma swoja specyficzne właściwości (np. pojazdy pływające - wyporność)
5. Chciałbym mieć możliwość tworzenia i aktualizacji bazy pojazdów. Operacje na tej bazie powinny być zastrzeżone tylko dla administratorów platformy.
6. Moi klienci powinni mieć możliwość przeglądania pojazdów do wypożyczenia, filtrowania po: dostępności, kategorii i właściwościach specyficznych (np. rodzaj silnika, maksymalny dystans, maksymalna wysokość, ilość osób itp).
7. Klienci powinni też mieć możliwość zamówienia pojazdu na dany okres czasu, a także zwrócenia go gdy przestanie im być potrzebny.

Projekt R-A-V - Zadanie nr 1

1. Kilka pytań na początek - zastanówmy się:
   a. jakie elementy powinny wejść w skład projektu?
   b. co jest jego podstawowym zadaniem?
   c. na jakich danych powinien działać?
   d. kto ma korzystać z aplikacji?

2. Na podstawie powyższych przemyśleń spróbuj stworzyć model dziedziny, czyli stwórz klasy odpowiadające poszczególnym elementom projektu.

Projekt R-A-V - model domeny

Pojazdy do wypożyczenia:

1. samochód (ang. **Car**) - właściwości:
   a. identyfikator (ang. **vin**)
   b. status (ang. **status**)
   c. nazwa (ang. **name**)
   d. data produkcji (ang. **productionDate**)
   e. maksymalny dystans w kilometrach (ang. **maxDistance**)
   f. rodzaj nadwozia (ang. **bodyType**)

2. motorówka (ang. **Motorboat**) - właściwości:
   a. identyfikator (ang. **vin**)
   b. status (ang. **status**)
   c. nazwa (ang. **name**)
   d. data produkcji (ang. **productionDate**)
   e. maksymalny zasięg w milach morskich (ang. **maxDistance**)
   f. wyporność (ang. **displacement**)

3. amfibia (ang. **Amphibian**) - właściwości:
   a. identyfikator (ang. **vin**)
   b. status (ang. **status**)
   c. nazwa (ang. **name**)
   d. data produkcji (ang. **productionDate**)
   e. maksymalny dystans w kilometrach (ang. **maxDistanceKm**)
   f. maksymalny dystans w milach morskich (ang. **maxDistanceMiles**)
   g. wyporność (ang. **displacement**)

Projekt R-A-V - model domeny

Użytkownicy i zamówienia:

1. użytkownik (ang. **User**) - właściwości:
   a. login (ang. **login**)
   b. hasło (ang. **password**)
   c. typ: klient/administrator (ang. **userType**)

2. zamówienie (ang. **Order**) - właściwości:
   a. identyfikator (ang. **id**)
   b. klient (ang. **customer**)
   c. pojazd (ang. **vehicle**)
   d. data wypożyczenia (ang. **startDate**)
   e. data zwrotu (ang. **endDate**)

Zasady/reguły wytwarzania programowania

DRY (ang. *Don't Repeat Yourself*)
Nie powtarzaj się!

- reguła zalecająca unikanie różnego rodzaju powtórzeń wykonywanych przez programistów
- w tym powtórzeń w kodzie źródłowym, zamiast ‘kopiuj-wklej’ stosujmy:
  - dobrze zdefiniowane metody, klasy
  - dziedziczenie
  - kompozycje
- automatyzuj co się da:
  - proces budowania aplikacji: kompilowanie, testowanie, wdrażanie - Maven!
  - zamiast ręcznie wykonywanych czynności - pisz skrypty

Projekt R-A-V - Zadanie nr 2

1. Stwórz klasę typu **DAO** (**ang. Data Access Object**) to przechowywania i aktualizacji pojazdów oraz użytkowników (**VehiclesDao**, **UsersDao**).
2. Dodaj metody do pobierania wszystkich pojazdów i użytkowników
3. Dodaj metodę do dodawania pojazdów i użytkownika.
4. Dodaj metodę do usuwania pojazdów i użytkownika po id. Jeżeli pojazd/użytkownik o podanym id nie istnieje metoda powinna zwrócić false, w przeciwnym wypadku true.
5. Dodaj logowanie zdarzeń w klasie - wystarczy zalogować zdarzenia wyjątkowe (np.: brak danych)
6. Dodaj testy do metod klasy

Zasady/reguły wytwarzania programowania

KISS (ang. *Keep It Simple, Stupid*)
Nie komplikuj, głupcze!

- reguła zalecająca pisanie kodu tak prostego jak to możliwe
- nie piszmy kodu “na zapas” - który może kiedyś się przyda
- twój kod podczas analizy nie powinien zmuszać do zbytniego myślenia.
- gdy po jakimś czasie wracasz do swojego kodu i nie wiesz co tam się dzieje, to znak, że musisz nad tym popracować ;)
- zasada powiązana z **TDD** (*Test Driven Development*) - piszemy tyle kodu ile potrzebne żeby nasz test przeszedł
- polski odpowiednik: **BUZI** - Bez Udziwnień Zapisu, Idioto :)

Projekt R-A-V - Zadanie nr 3

1. Dodaj do klasy **VehiclesDao** metodę do wyszukiwania pojazdów po parametrach:
   a. kategoria: pojazd jeżdżący, latający lub pływający
   b. status: dostępny/niedostępny
   c. wiek pojazdu: 0-2 lat, 2-5 lat, starsze
   d. dla pojazdów jeżdżących wyszukiwanie po typie nadwozia
   e. dla pojazdów pływających wyszukiwanie po wyporności

2. Dodaj logowanie zdarzeń w klasie - wystarczy zalogować zdarzenia wyjątkowe (np.: brak danych)

3. Dodaj testy do nowej metody

Zasady/reguły wytwarzania programowania

SOLID

S - **Single responsibility** (pojedyncza odpowiedzialność)
O - **Open - closed** (otwarte – zamknięte)
L - **Liskov substitution** (podstawianie Liskov)
I - **Interface segregation** (segregacja interfejsów)
D - **Dependency inversion** (odwracanie zależności)

Projekt R-A-V - Zadanie nr 4

1. Stwórz klasę do zapisywania i zarządzania zamówieniami: **OrdersDao**.
2. Dodaj metodę do zamówienia pojazdu.
3. Dodaj metodę do zwrócenia pojazdu.
4. Dodaj metodę do sprawdzenia czy dany pojazd jest dostępny w danym przedziale czasu.
5. Dodaj metodę zwracającą historię zamówień podanego klienta.
6. Dodaj logowanie zdarzeń w klasie
7. Dodaj testy do metod klasy

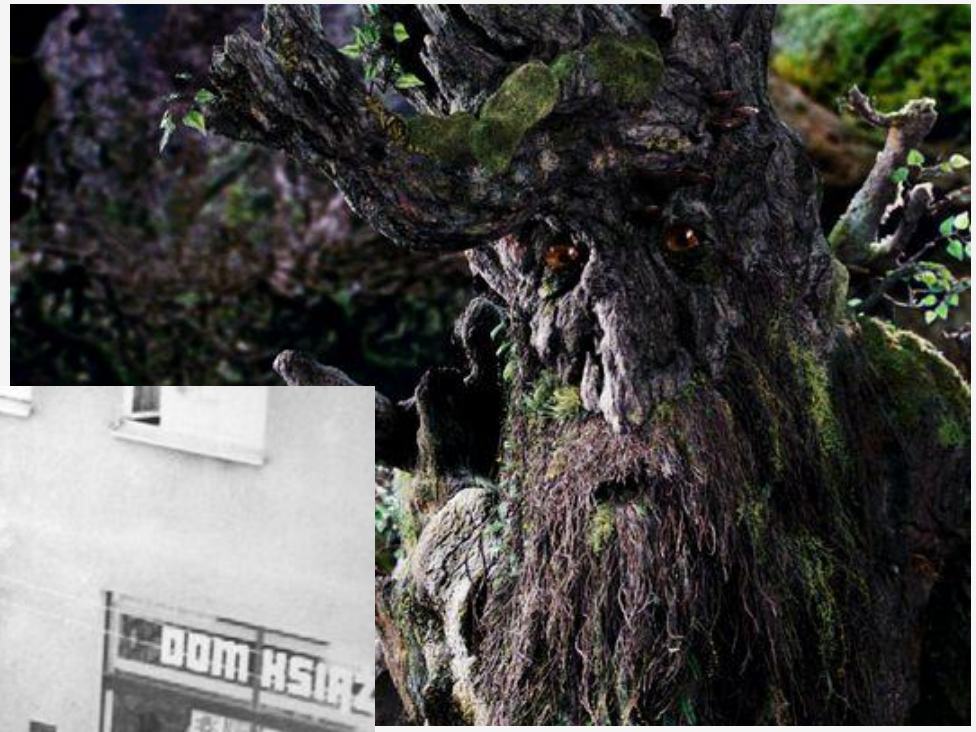
Projekt R-A-V - Zadanie nr 5

1. Stwórz główną klasę-serwis do zarządzania aplikacją R-A-V: **RavService**
2. Dodaj metodę do logowania użytkownika.
3. Dodaj metodę do wyszukiwania pojazdów.
4. Dodaj metody do wypożyczenia i zwrotu pojazdu.
5. Dodaj metodę zwracającą historię zamówień zalogowanego klienta.
6. Dodaj logowanie zdarzeń w klasie
7. Dodaj testy do metod klasy

KONIEC

Dziękuję ;)

Rozkład jazdy



JUnit