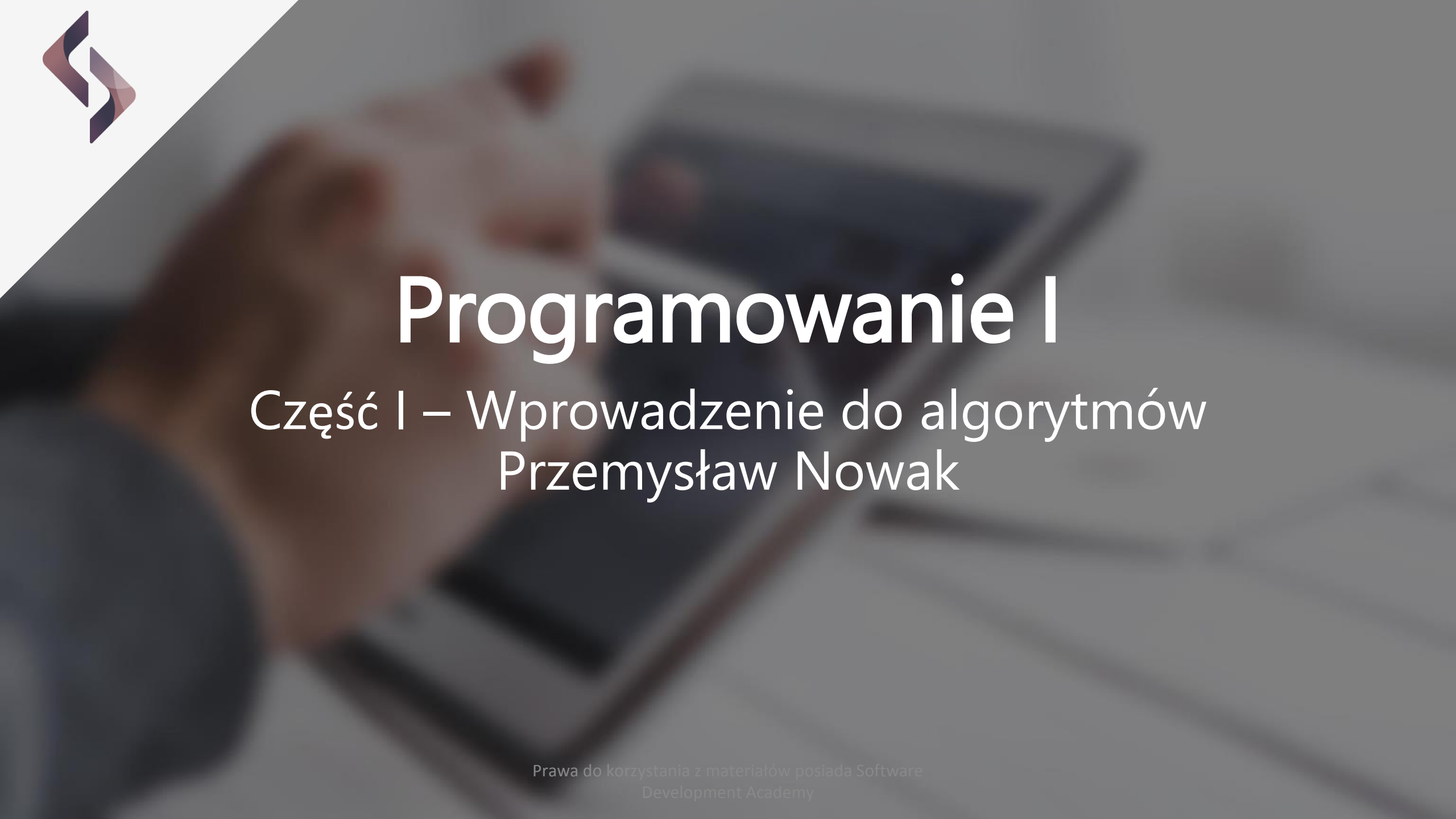


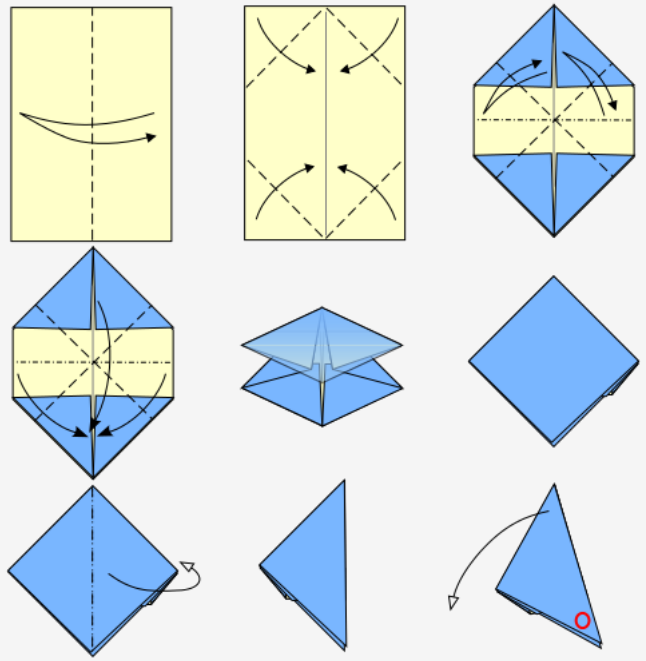
Programowanie I

Część I – Wprowadzenie do algorytmów

Przemysław Nowak

Czym jest algorytm? (algorithm)

Algorytm – skończony ciąg jasno zdefiniowanych czynności, koniecznych do wykonania pewnego rodzaju zadań. Słowo „algorytm” pochodzi od staroangielskiego słowa algorism, oznaczającego wykonywanie działań przy pomocy liczb arabskich. Zadaniem algorytmu jest przeprowadzenie systemu z pewnego stanu początkowego do pożądanego stanu końcowego. Badaniem algorytmów zajmuje się algorytmika. Algorytm może zostać zaimplementowany w postaci programu komputerowego.



```
  34061
1260257 : 37
- 111
  150
- 148
    225
  - 222
      37
    - 37
      0
```

Opisywanie algorytmów

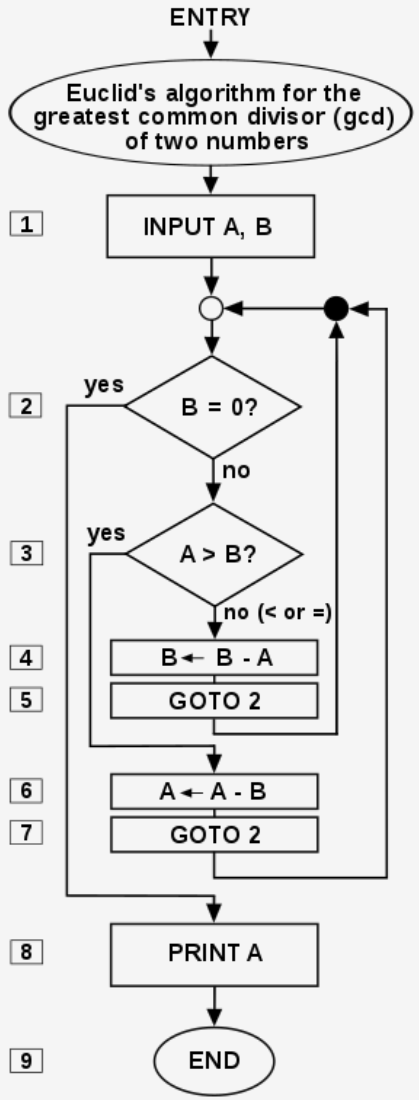
Aby policzyć NWD *(GCD)*:

1. przyjmij zmienne wejściowe *a* i *b*
2. jeśli *a* jest równe *b*, zwróć *a* jako wynik
3. jeśli *a* jest większe od *b* przypisz do *a* różnicę *a* i *b*
4. w przeciwnym wypadku przypisz do *b* różnicę *b* i *a*
5. wróć do punktu 2.

```
function gcd(a, b)
    while a ≠ b
        if a > b
            a ← a − b
        else
            b ← b − a
    return a
```

pseudokod
(pseudocode)

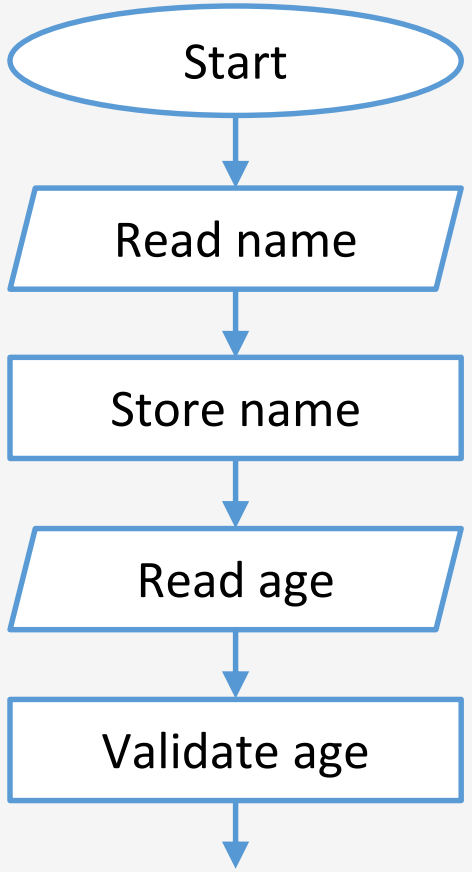
schemat blokowy
(flowchart)



Podstawowe elementy algorytmów

sekwencja instrukcji

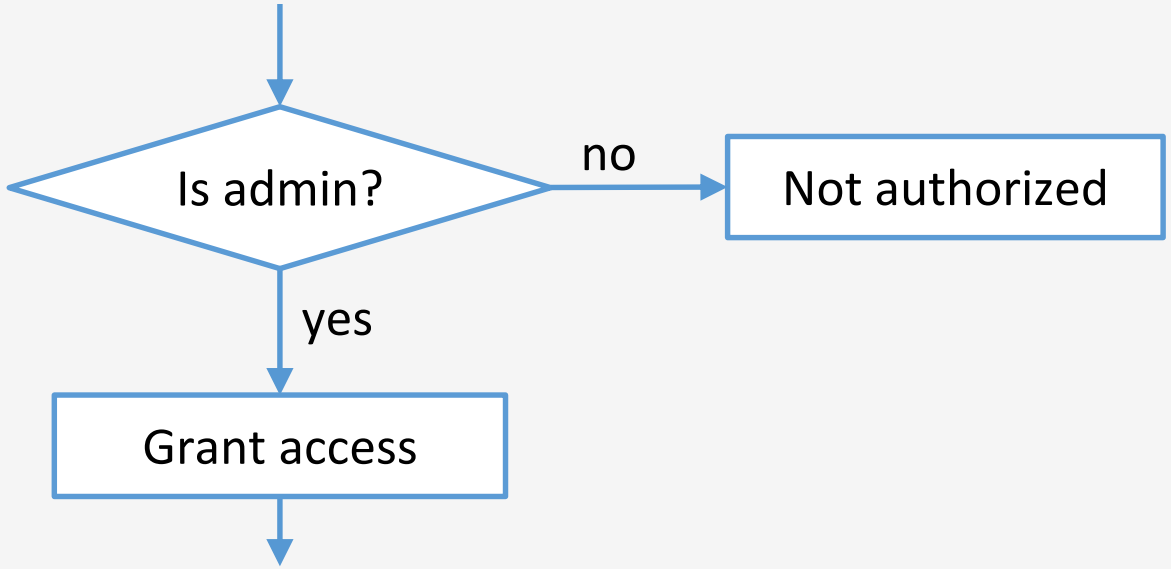
```
String name = nextLine();
storeName(name);
int age = Integer.parseInt(nextLine());
validateAge(age);
```



Podstawowe elementy algorytmów

instrukcje warunkowe

```
switch (color) {
    case RED:   return 0xFF0000;
    case GREEN: return 0x00FF00;
    case BLUE:  return 0x0000FF;
}
```




```
if (user.isAdmin()) {
    grantAccess();
} else {
    throw new NotAuthorizedException();
}
```

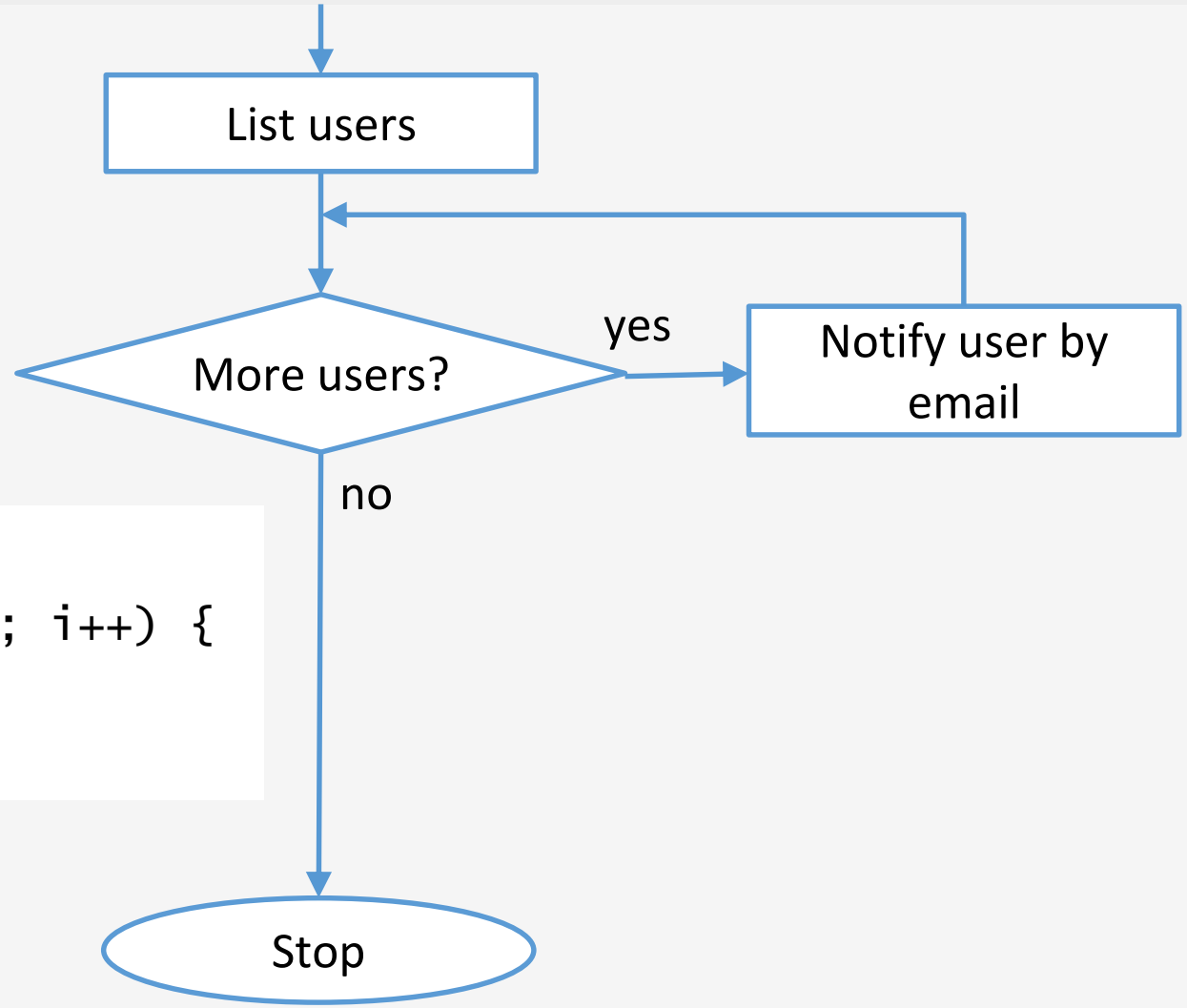
Podstawowe elementy algorytmów

powtarzanie instrukcji

```
for (user : listUsers()) {
    notifyByEmail(user);
}
```

```
List<User> users = listUsers();
for (int i = 0; i < users.size(); i++) {
    notifyByEmail(users.get(i));
}
```

```
while (reader.hasMore()) {
    execute(reader.next());
}
```



Iteracja (iteration) a rekurencja (recursion)

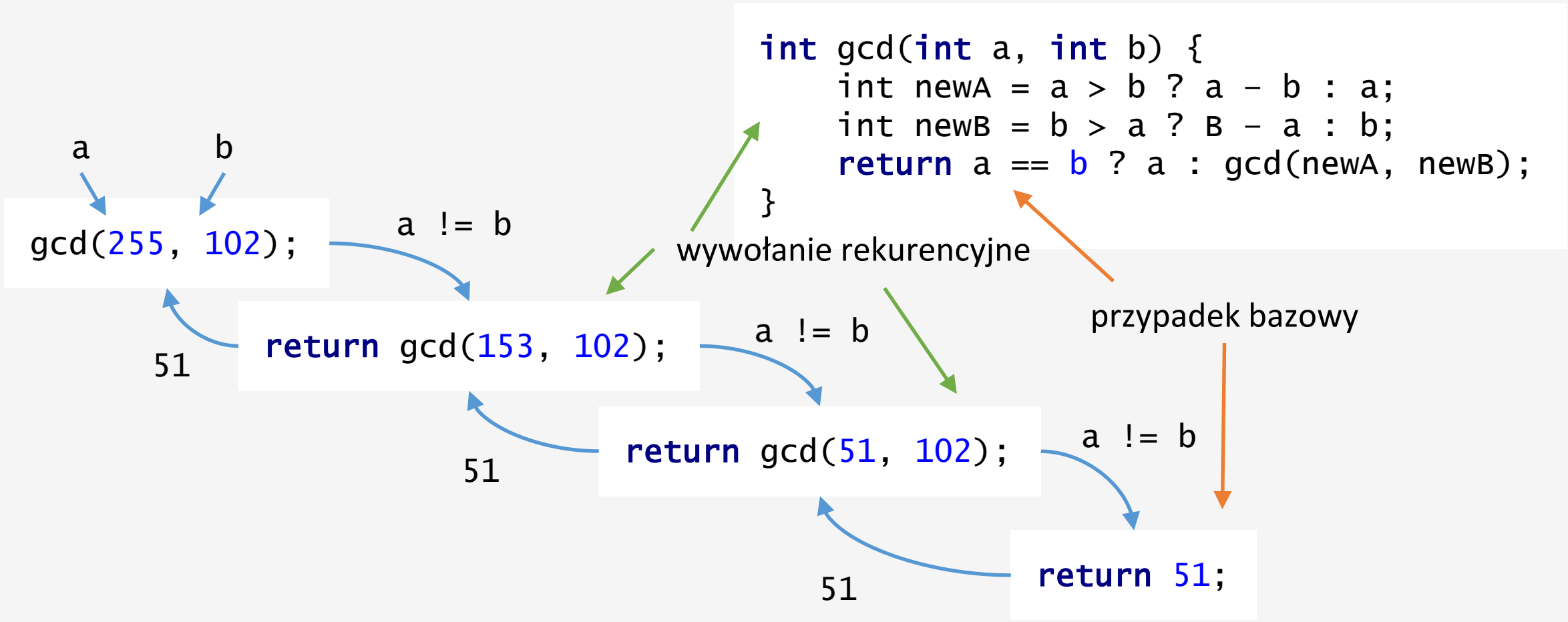
```
int gcd(int a, int b) {
    while (a != b) {
        if (a > b)
            a = a - b;
        else
            b = b - a;
    }
    return a;
}
```

iteracja z
wykorzystaniem pętli

```
int gcd(int a, int b) {
    int newA = a > b ? a - b : a;
    int newB = b > a ? B - a : b;
    return b == 0 ? a : gcd(newA, newB);
}
```

funkcja rekurencyjna
z wykorzystaniem rekurencji
ogonowej *(tail recursion)*

Jak działa rekurencja

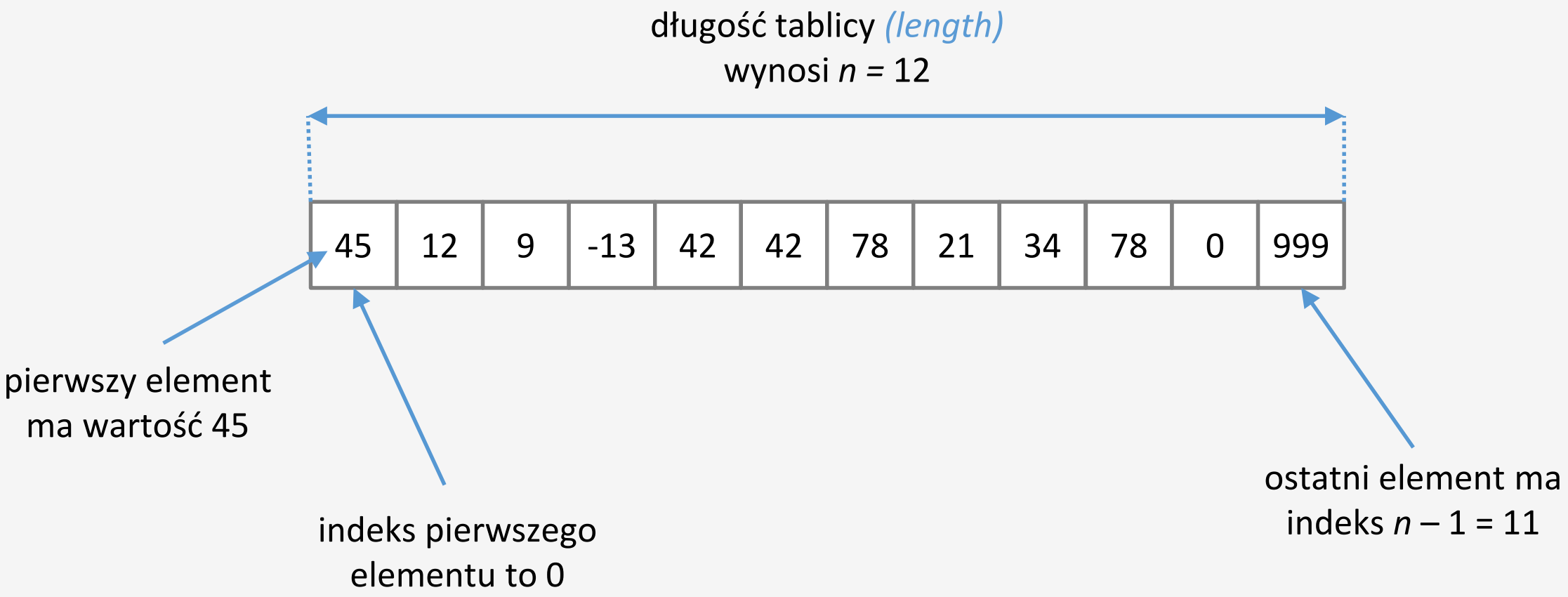


Tablica (array)

Najprostsza struktura danych

- bardzo krótki czas dostępu
- wydajna zamiana elementów
- prostota implementacji
- “płaska” struktura
- z góry określony rozmiar
- brak możliwości dziedziczenia
- tablica nie implementuje interfejsu Collection

Tablica (array)



Zadania!

Tasks_Arrays

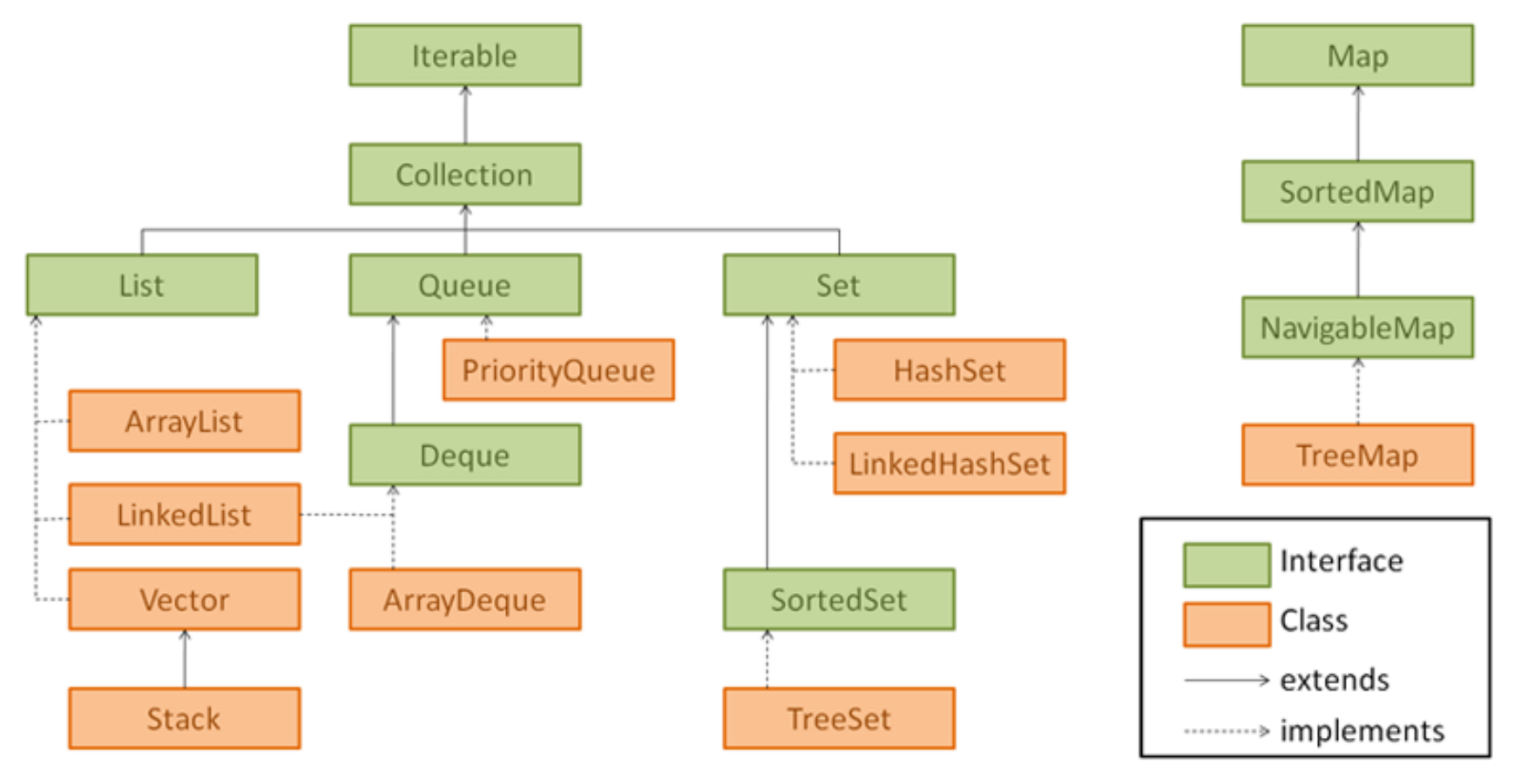
Czym są struktury danych? (data structures)

- sposób uporządkowania informacji
- umożliwiają implementację efektywnych algorytmów
- struktury danych a abstrakcyjne typy danych *(ADT – Abstract Data Type)*
  - *Struktury danych – konkretne implementacje których możemy użyć przy implementacji*
  - *ADT – abstrakcja, zdefiniowane zachowania ale nie implementacje*

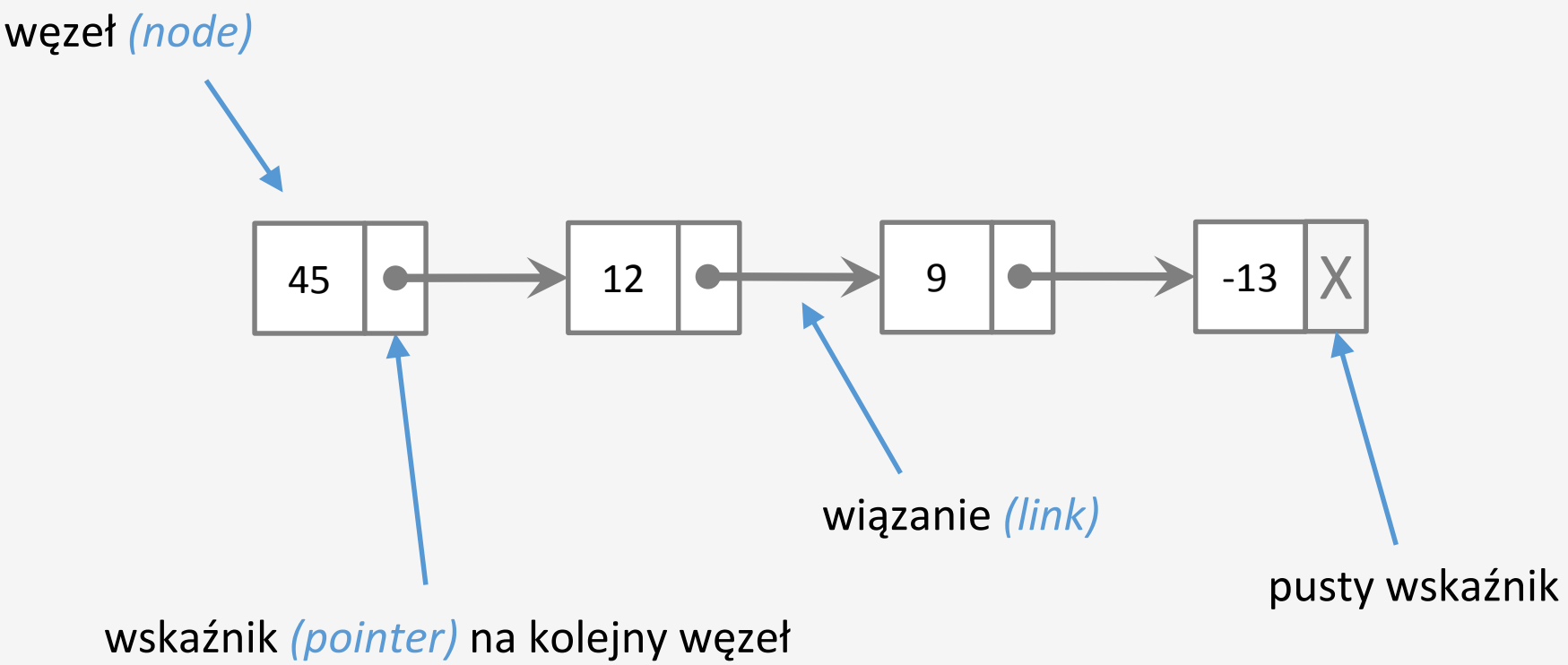
Podstawowe ADT

- lista *(list)*
- stos *(stack)*
- kolejka *(queue)*
- drzewo *(tree)*
- graf *(graph)*
- zbiór *(set)*
- multizbiór/wielozbiór *(multiset)*
- mapa *(map)*
- multimapa *(multimap)*
- mapa dwukierunkowa *(bidirectional map)*

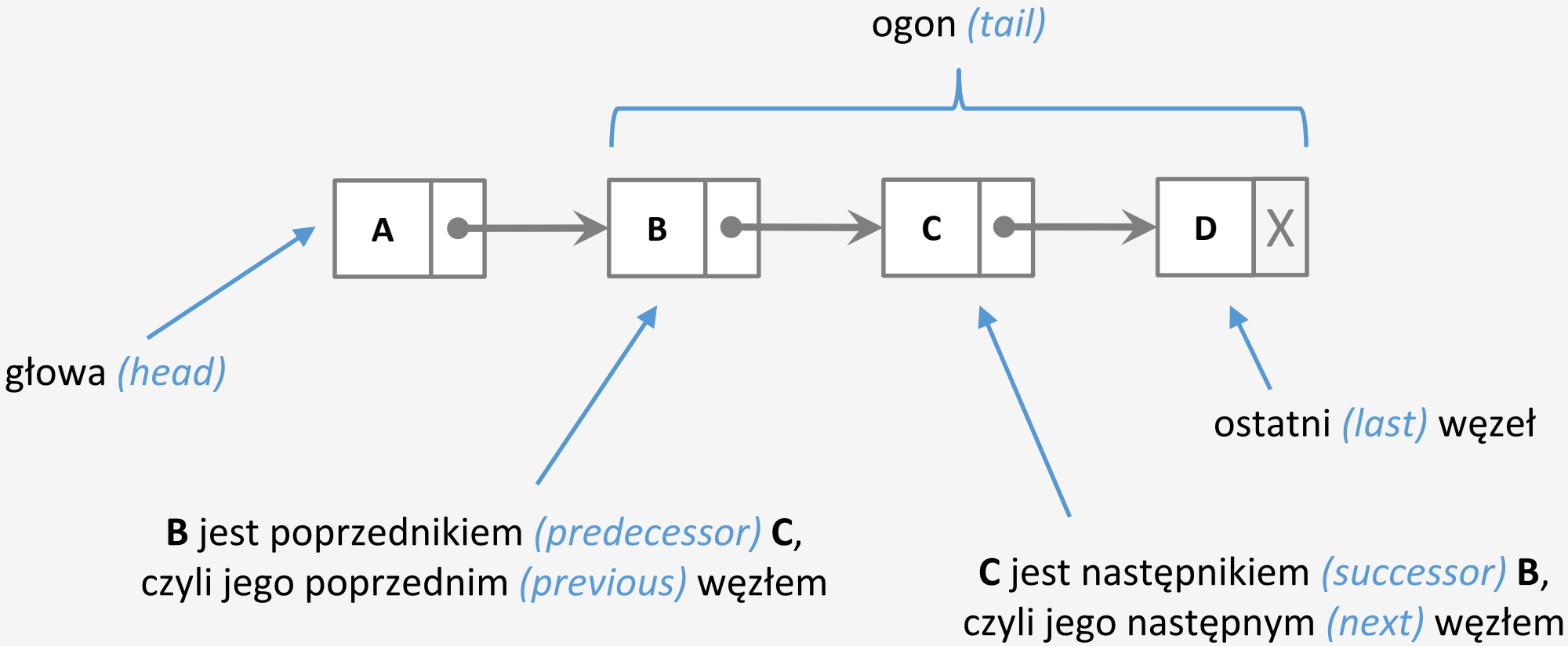
Java Data Structures



Lista wiązana jednokierunkowa (singly linked list)



Lista wiązana jednokierunkowa



Lista wiązana dwukierunkowa (doubly linked list)

Operacje na liście

- stworzenie pustej listy
- sprawdzenie czy lista jest pusta
- sprawdzenie rozmiaru listy
- dodanie elementu do listy
  - na początku *(prepend)*
  - na końcu *(append)*
  - w środku *(insert)*
- pobranie elementu z listy
- usunięcie elementu z listy

Listy w JDK

Zadania!

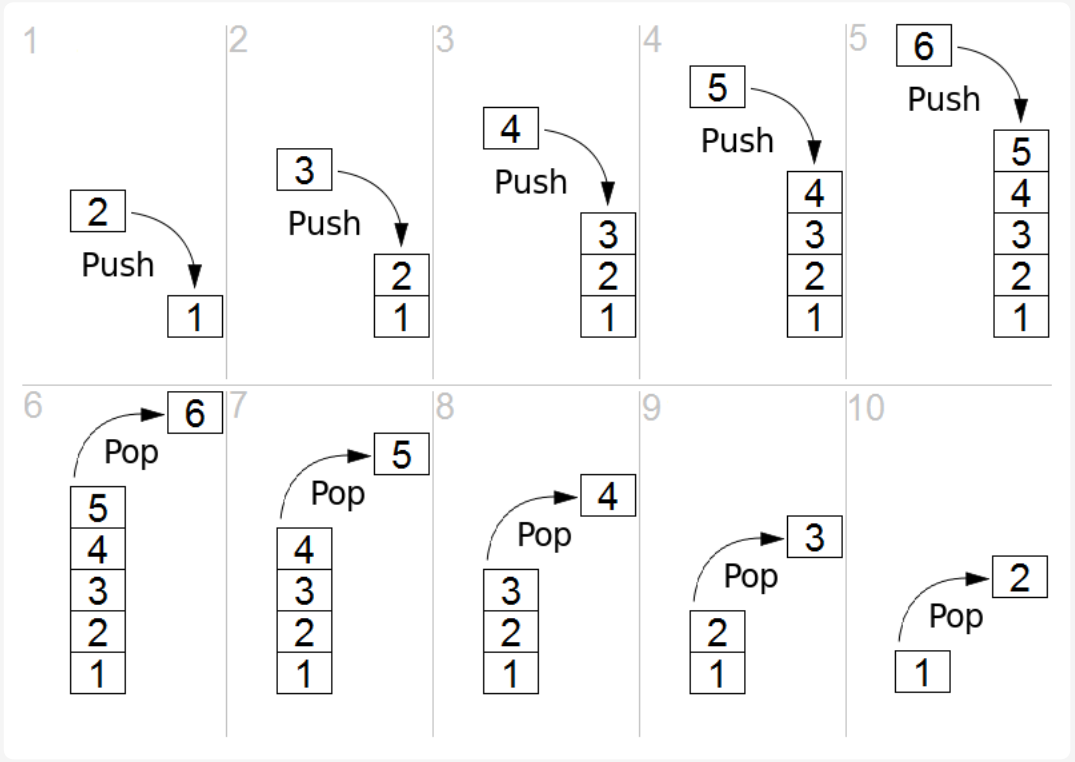
Tasks_LinkedList

Stos i kolejka

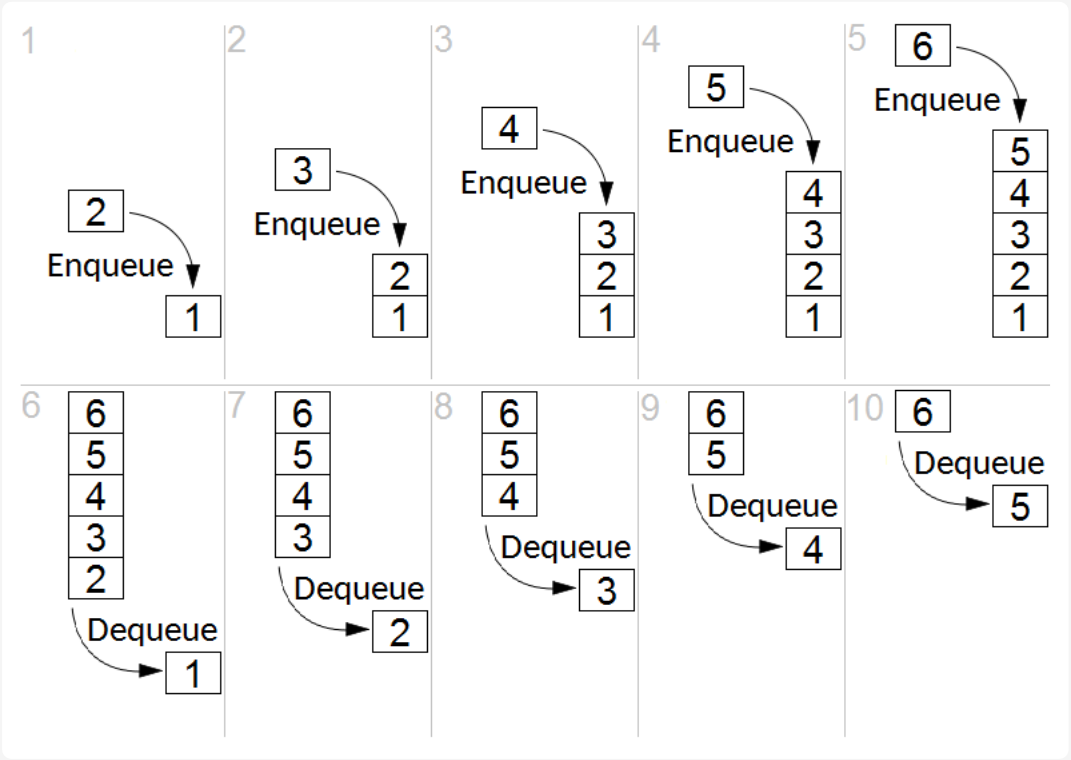
stos *(stack)*

kolejka *(queue)*

Stos i kolejka



LIFO – *Last In, First Out*



FIFO – *First In, First Out*

Operacje na stosie i kolejce

stos:

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

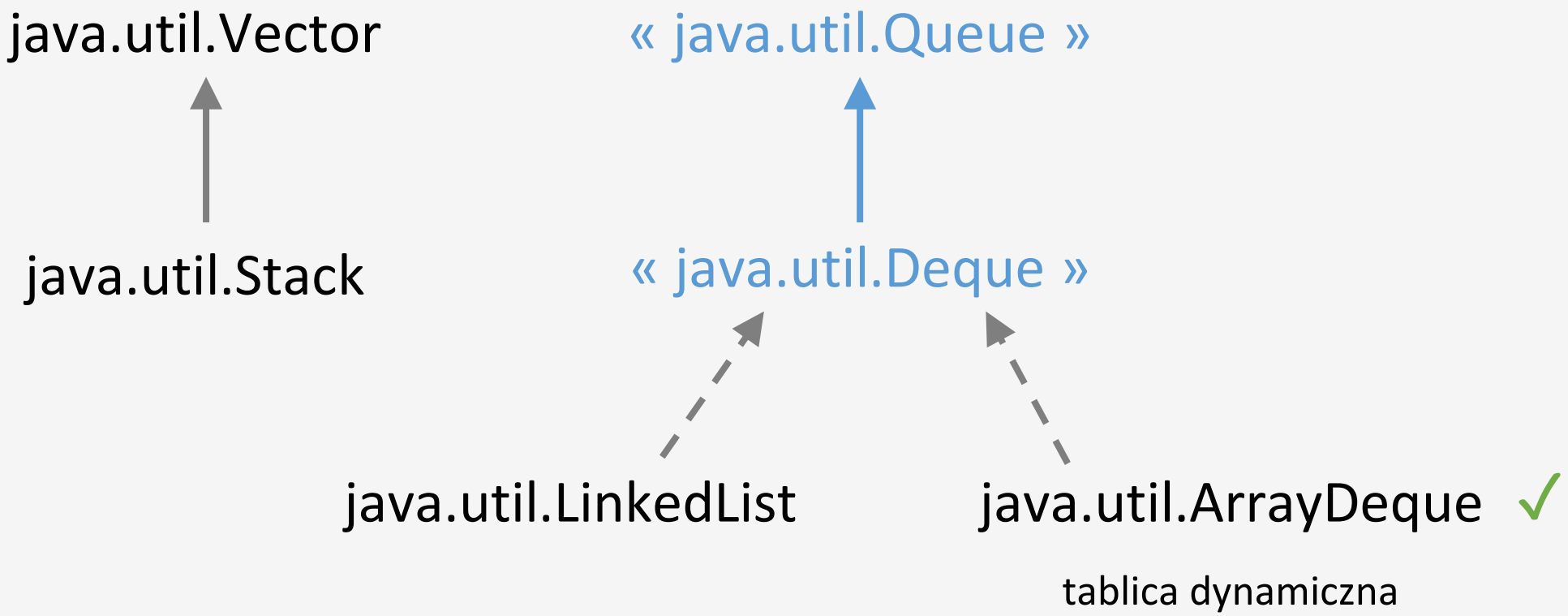
kolejka:

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

Kolejka dwustronna (deque)

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

Stosy i kolejki w JDK



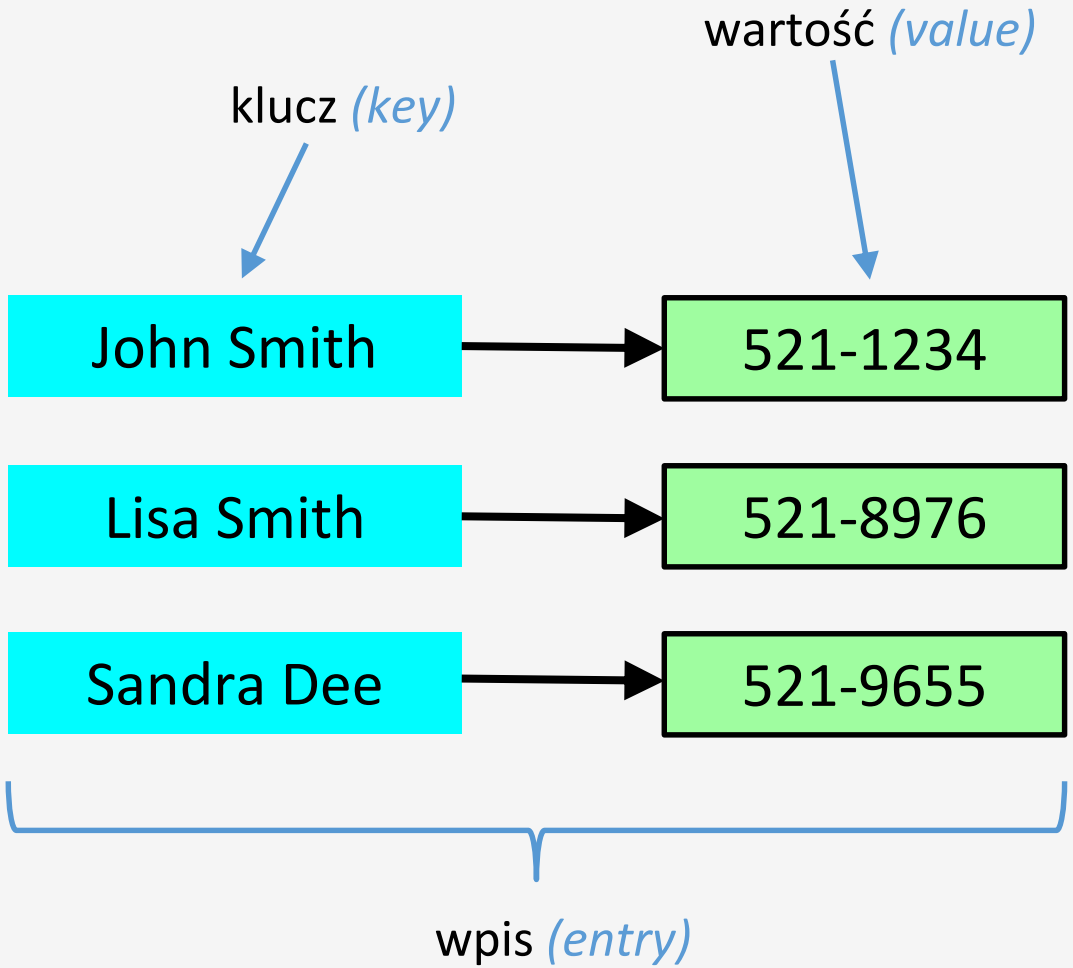
Zadania!

Tasks_Stack && Tasks_Queue

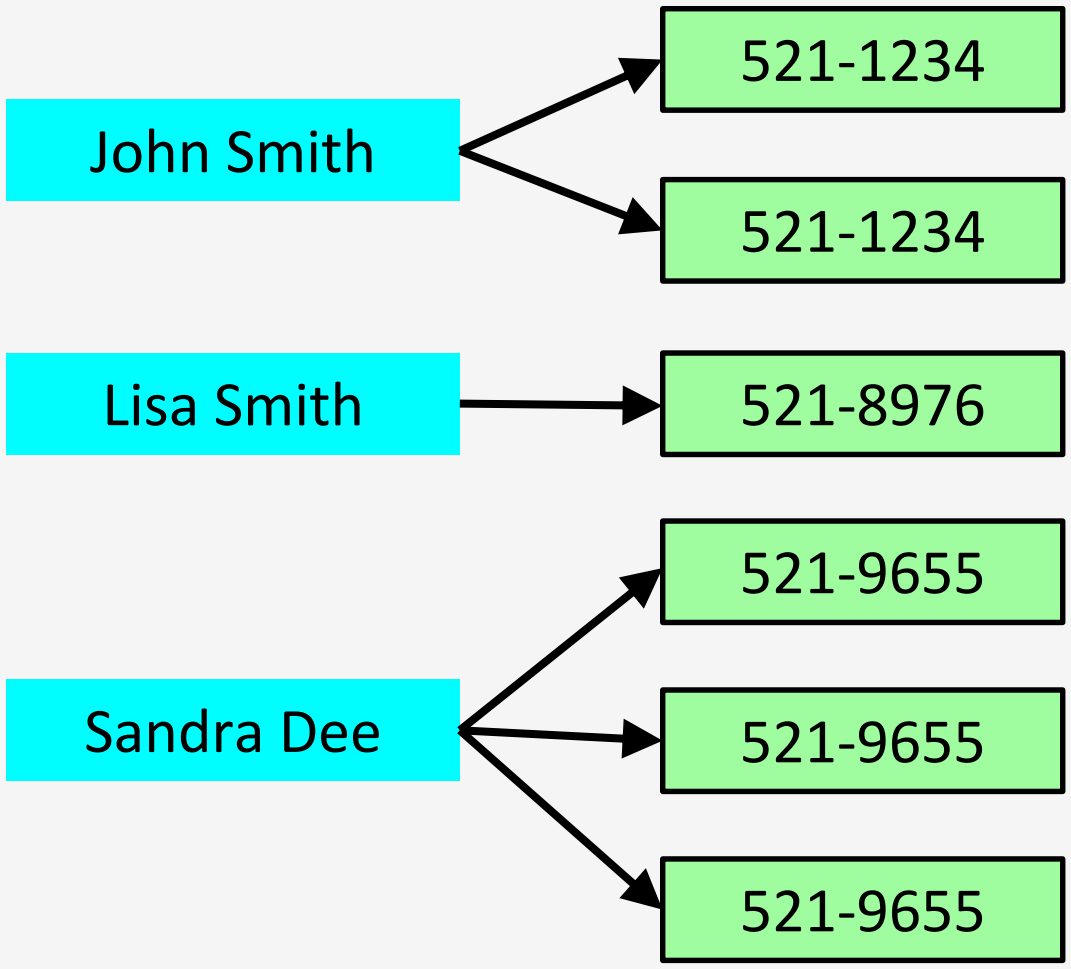
Mapa (map)

inaczej:
- tablica asocjacyjna *(associative array)*
- słownik *(dictionary)*

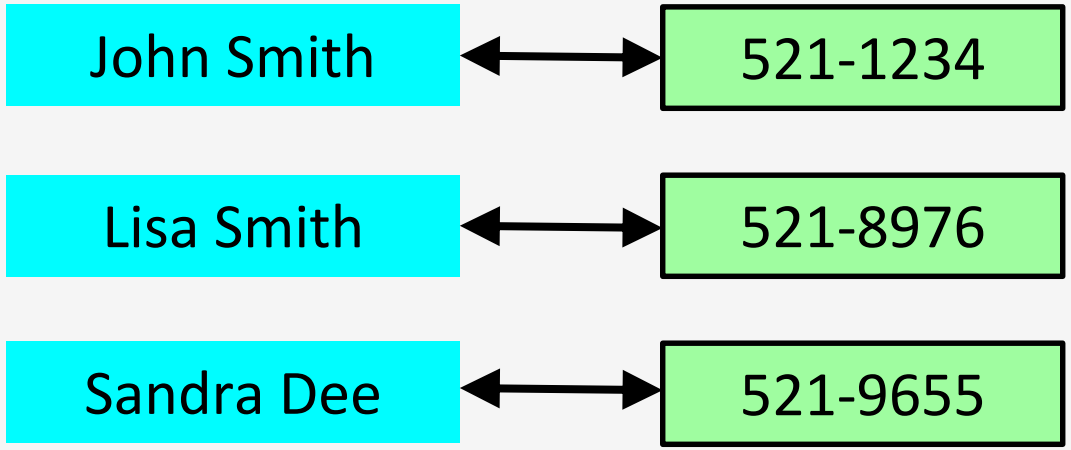
operacje:
- dodanie wpisu
- nadpisanie wpisu
- usunięcie wpisu
- pobranie wartości



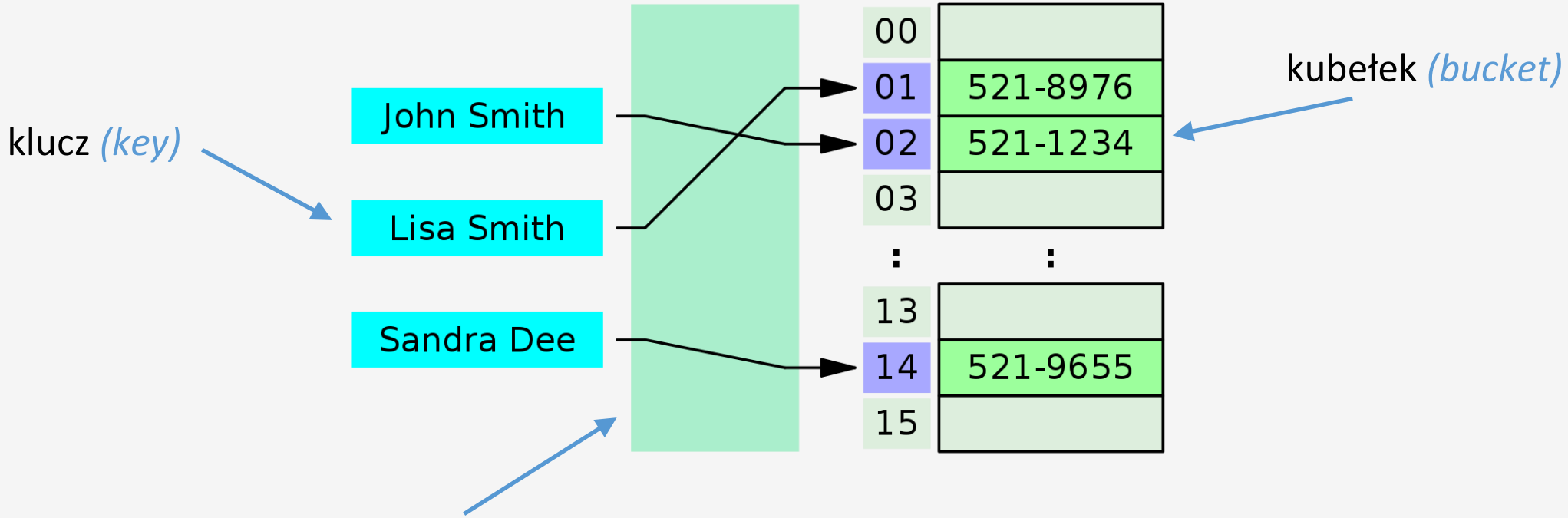
Multimapa (multimap)



Mapa dwukierunkowa (bidirectional map)

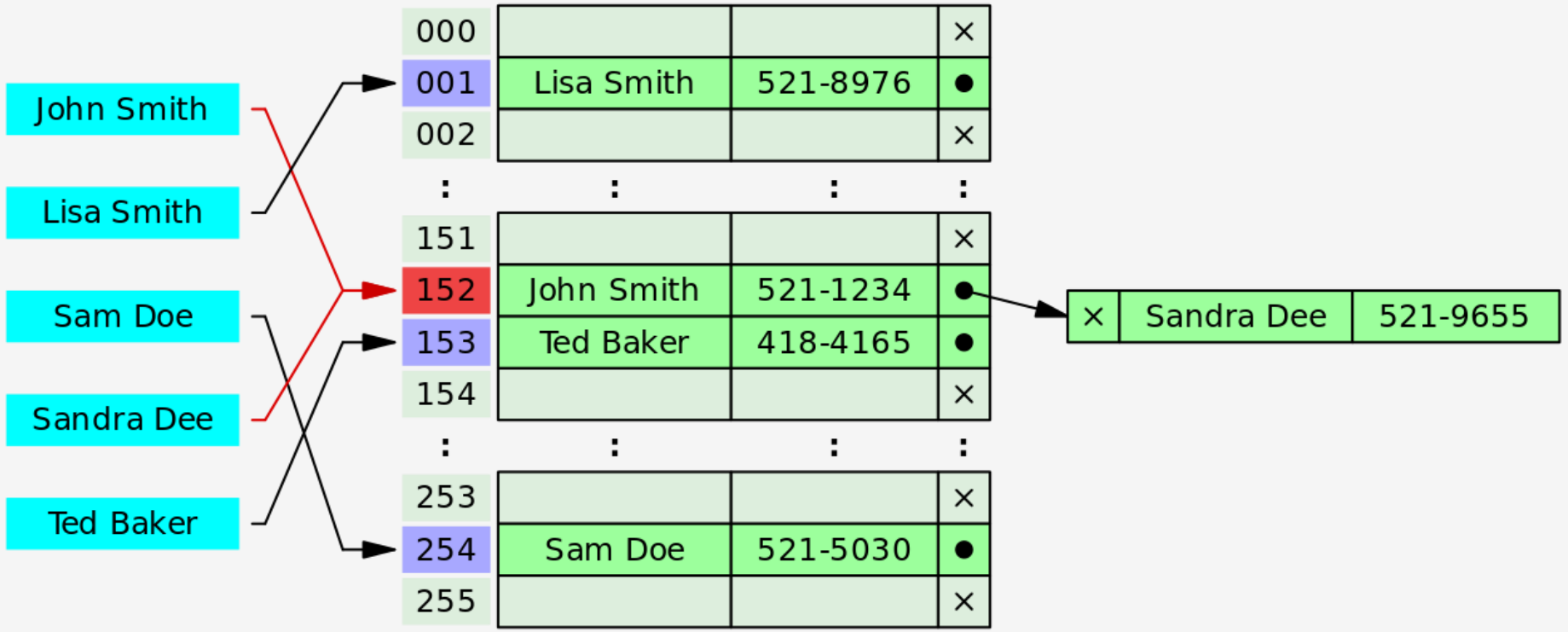


Tablica mieszająca (hash table)



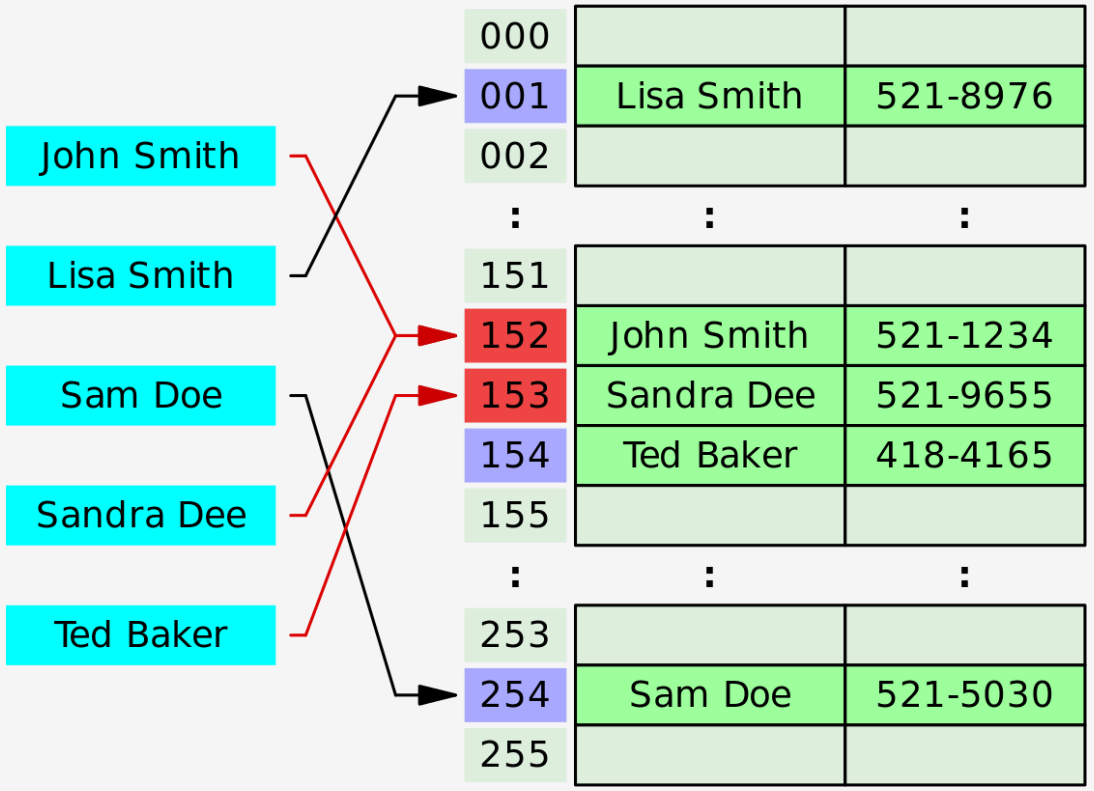
- Oblicza na podstawie klucza stałą liczbę o określonym zakresie.
- Powinna być względnie szybka.
- Powinna równomiernie rozmieszczać elementy w kubełkach.

Kolizje (hash collisions)



metoda łańcuchowa jako rozwiązanie na kolizje *(separate chaining)*

Kolizje



adresowanie otwarte jako rozwiązanie na kolizje *(open addressing)*

Zadania!

Tasks_Map

Zbiór (set)

Jeśli przyjmiemy, że wartością jest sam klucz, otrzymamy zbiór.

operacje:

- dodanie elementu
- usunięcie elementu
- sprawdzenie czy element należy do zbioru

Analogicznie, z multimapy możemy otrzymać multizbiór/wielozbiór.

Zadania!

Tasks_Set

Mapy i zbiory w JDK i Guava

Mapy:

- ~~Hashtable~~
- HashMap
- LinkedHashMap
- ConcurrentHashMap
- TreeMap
- Java 9: Map.of(...)

Zbiory:

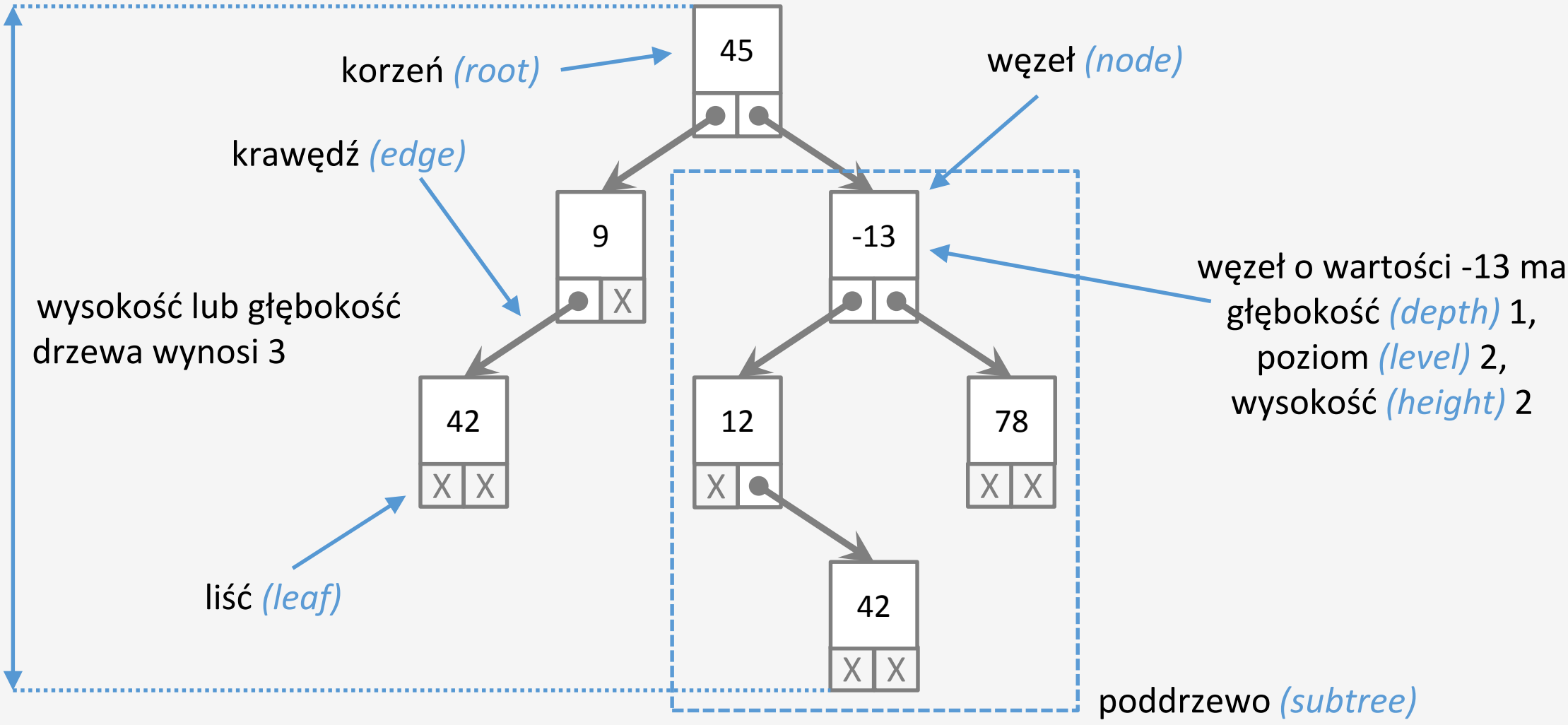
- HashSet
- LinkedHashSet
- ConcurrentHashMap.newKeySet()
- TreeSet
- Java 9: Set.of(...)

Guava:

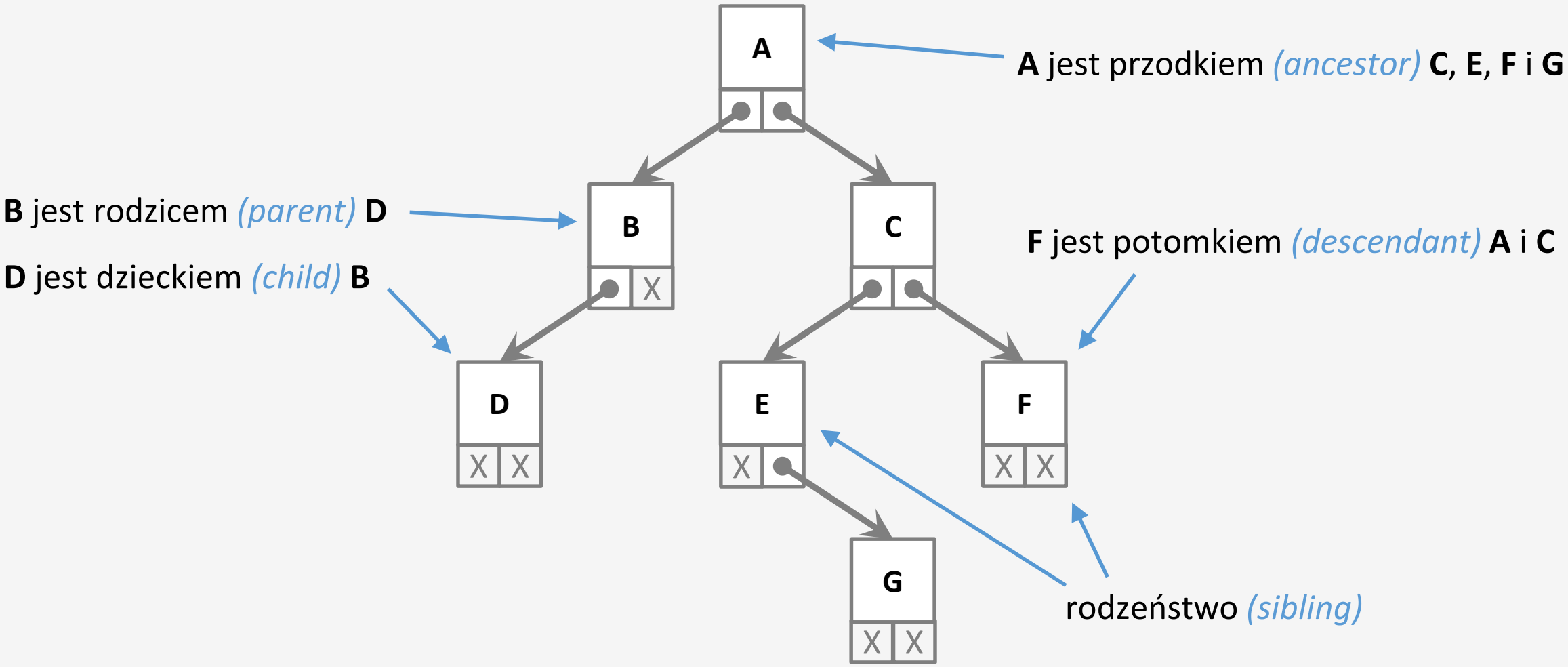
- ImmutableMap, ImmutableSet
- Multimap, BiMap, Multiset wraz z różnymi implementacjami

https://github.com/google/guava/wiki/NewCollectionTypesExplained

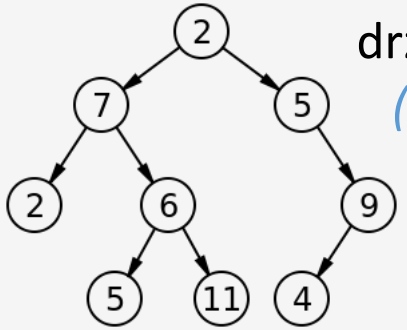
Drzewo (tree)



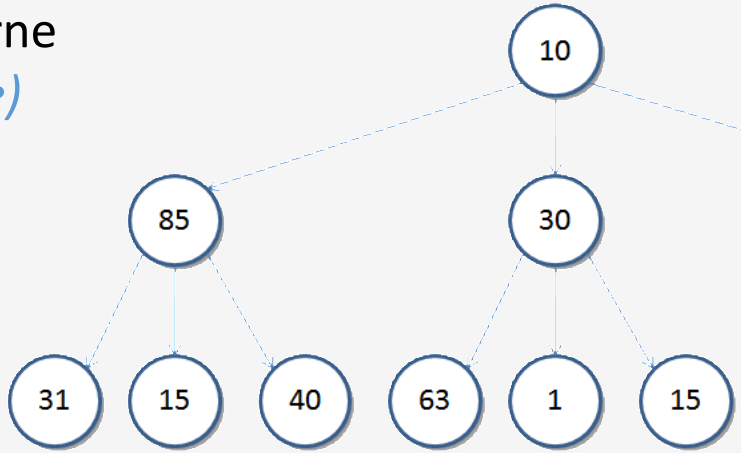
Drzewo



Różne współczynniki rozgałęzienia (branching factor)

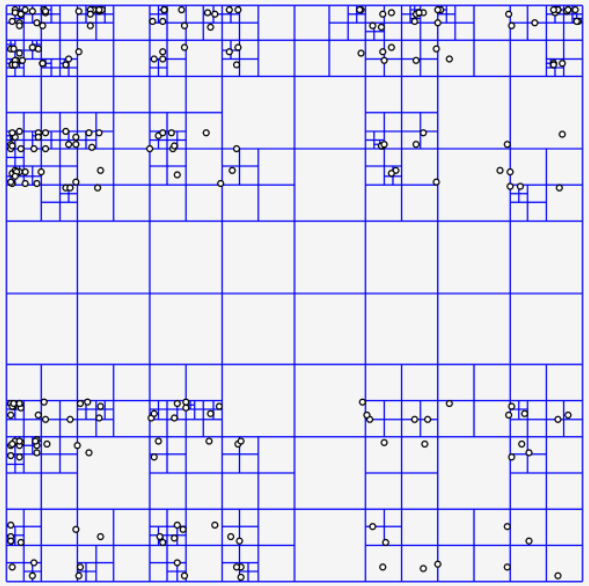


drzewo binarne
(binary tree)



drzewo trójkowe
(ternary tree)

drzewo ósemkowe
(octree)



drzewo czwórkowe
(quadtree)

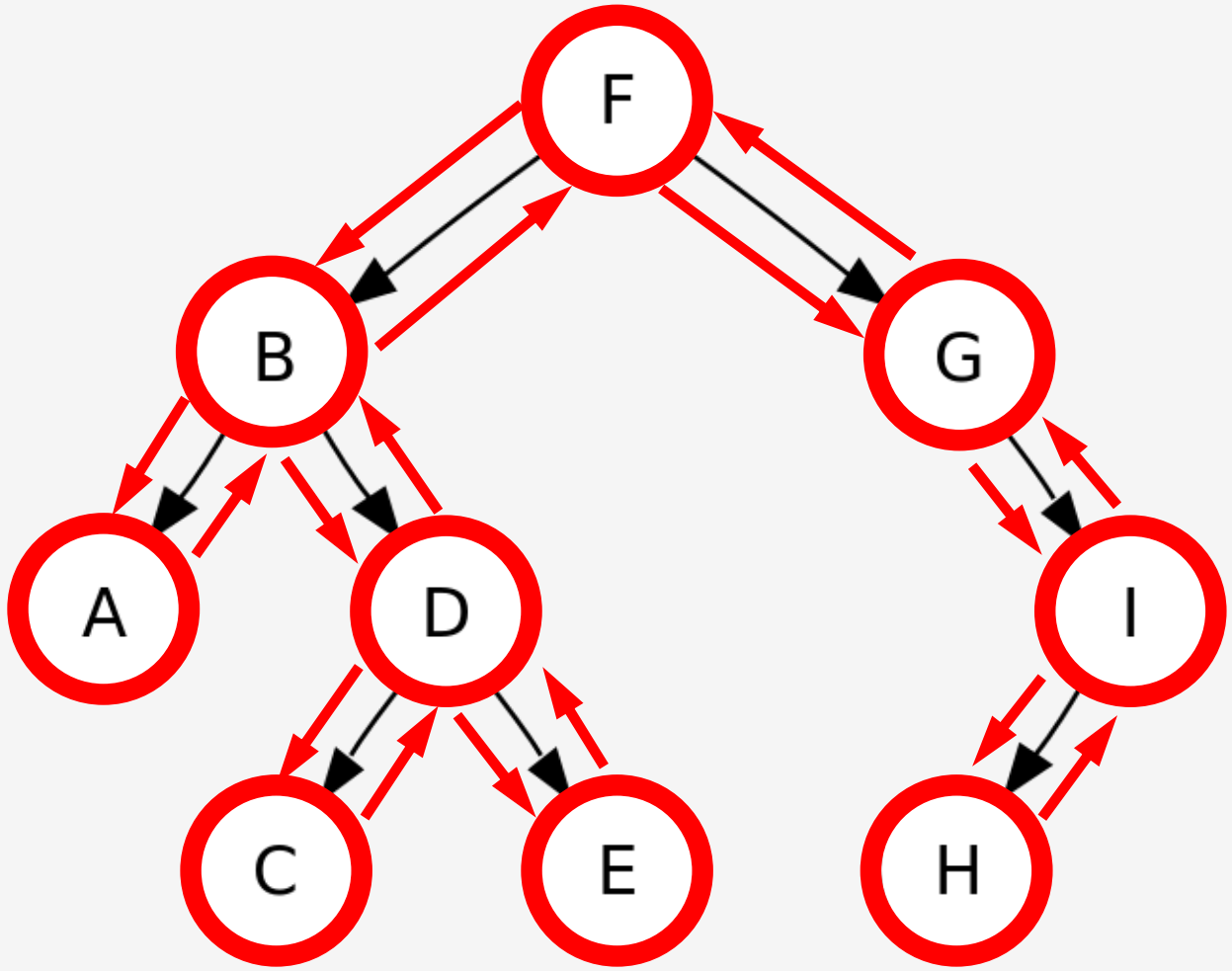
Przykładowe operacje na drzewie

- 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)*

Pre-order

NLR

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

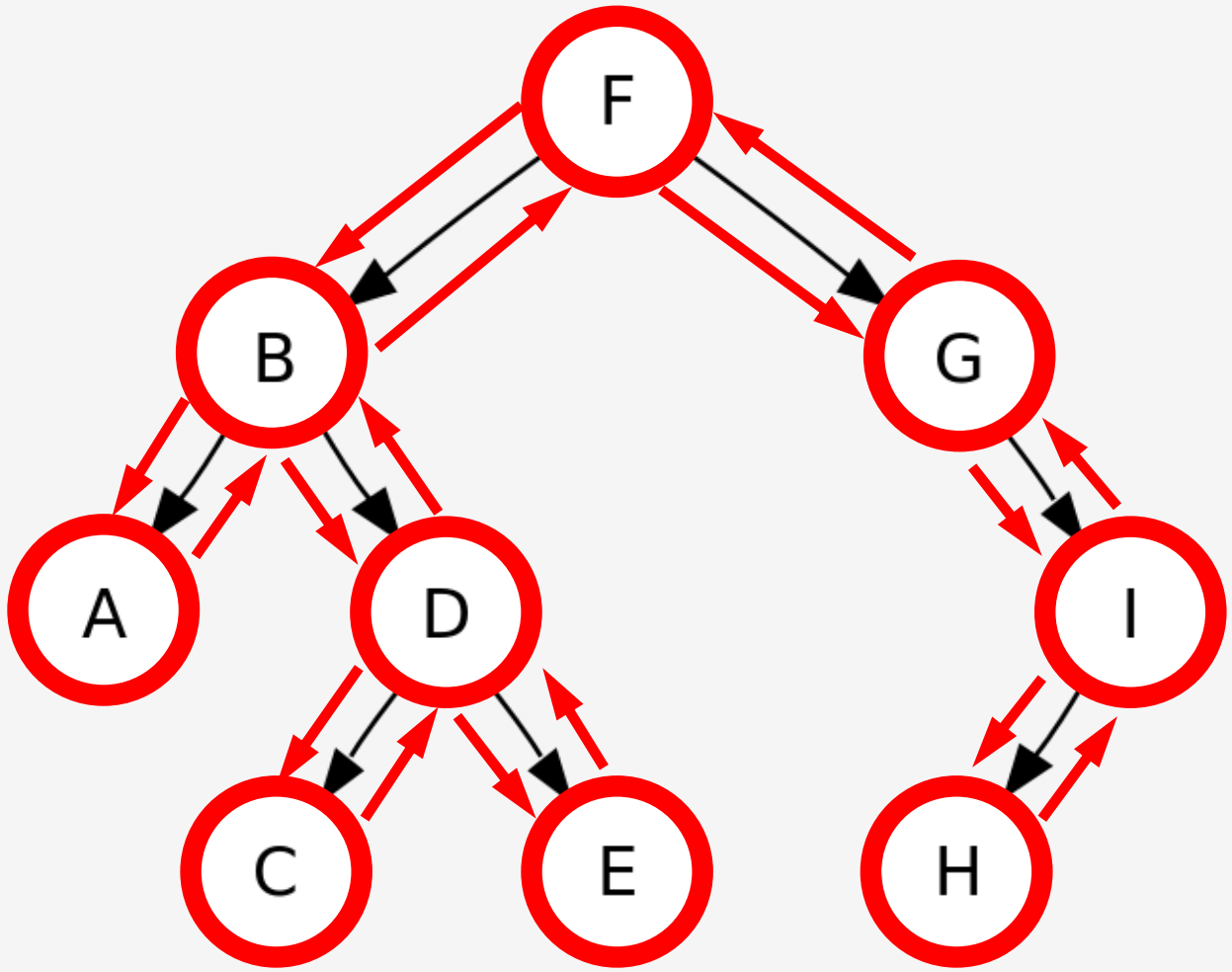


1. F
2. B
3. A
4. D
5. C
6. E
7. G
8. I
9. H

In-order

LNR

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

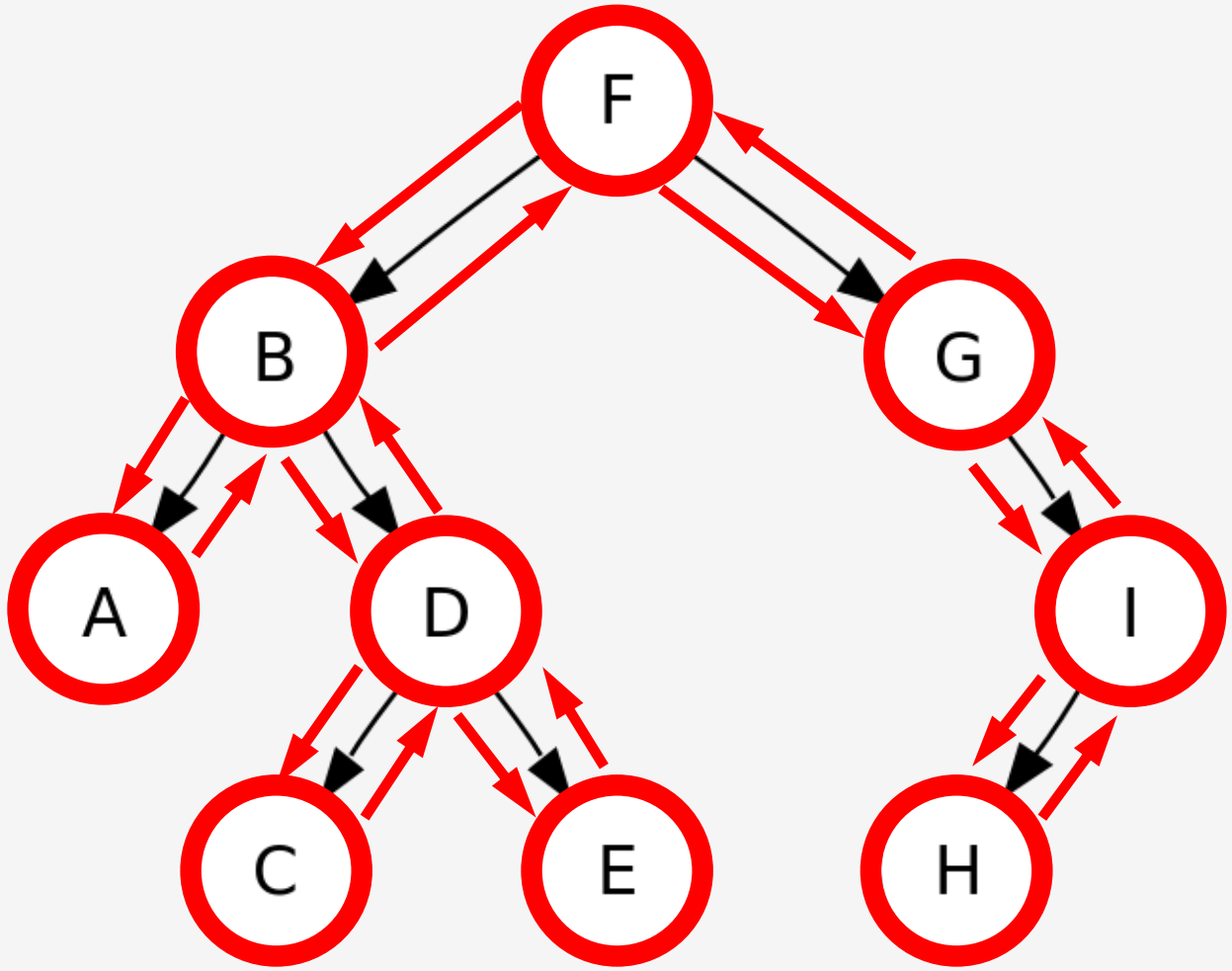


1. A
2. B
3. C
4. D
5. E
6. F
7. G
8. H
9. I

Post-order

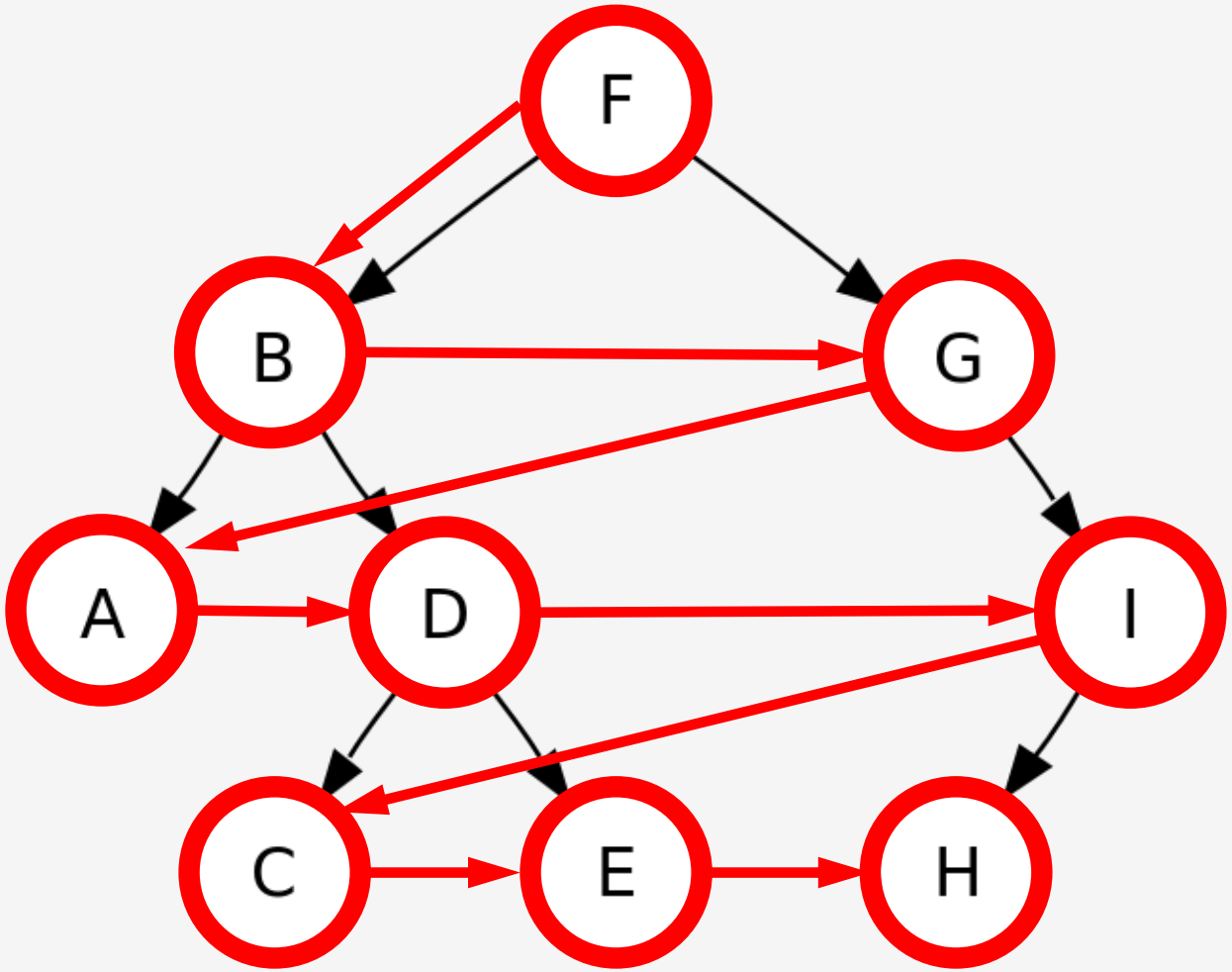
LRN

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



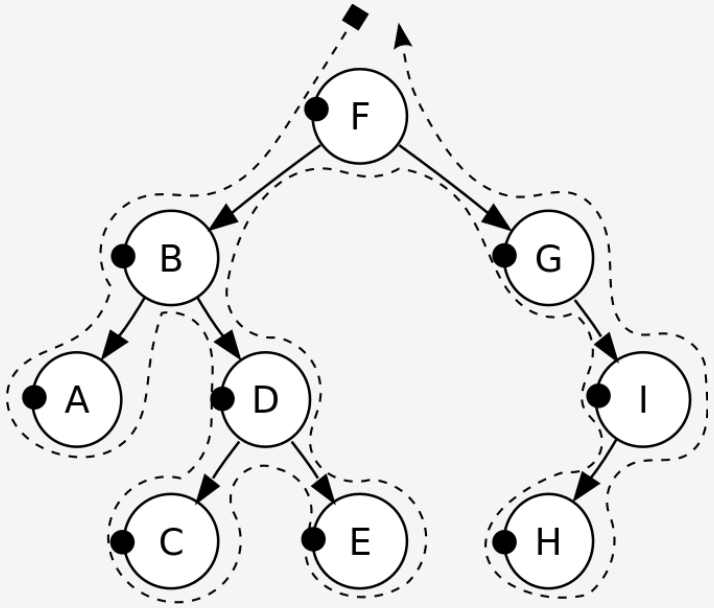
1. A
2. C
3. E
4. D
5. B
6. H
7. I
8. G
9. F

Level-order

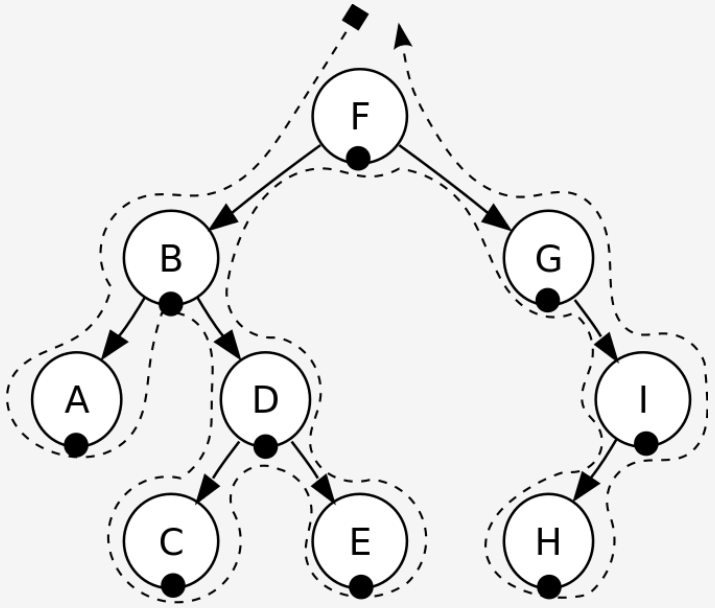


1. F
2. B
3. G
4. A
5. D
6. I
7. C
8. E
9. H

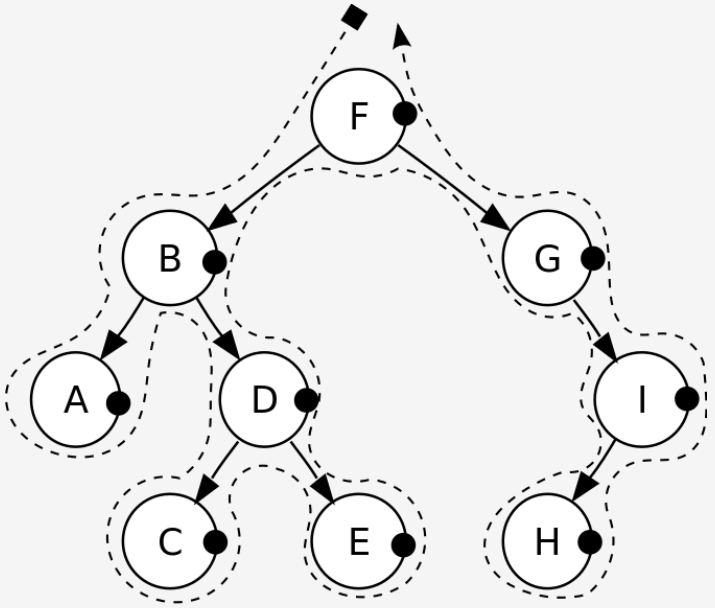
Porównanie



pre-order (**N**LR)



in-order (L**N**R)

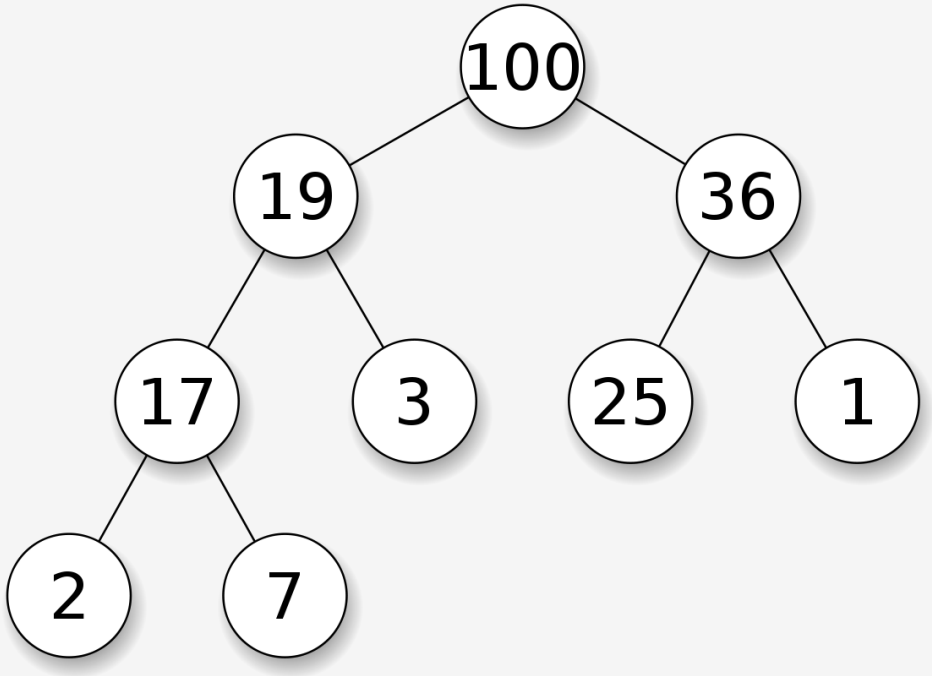


post-order (LR**N**)

Zadania!

Tasks_Tree

Kopiec (heap)



Specjalne drzewo, w którym:

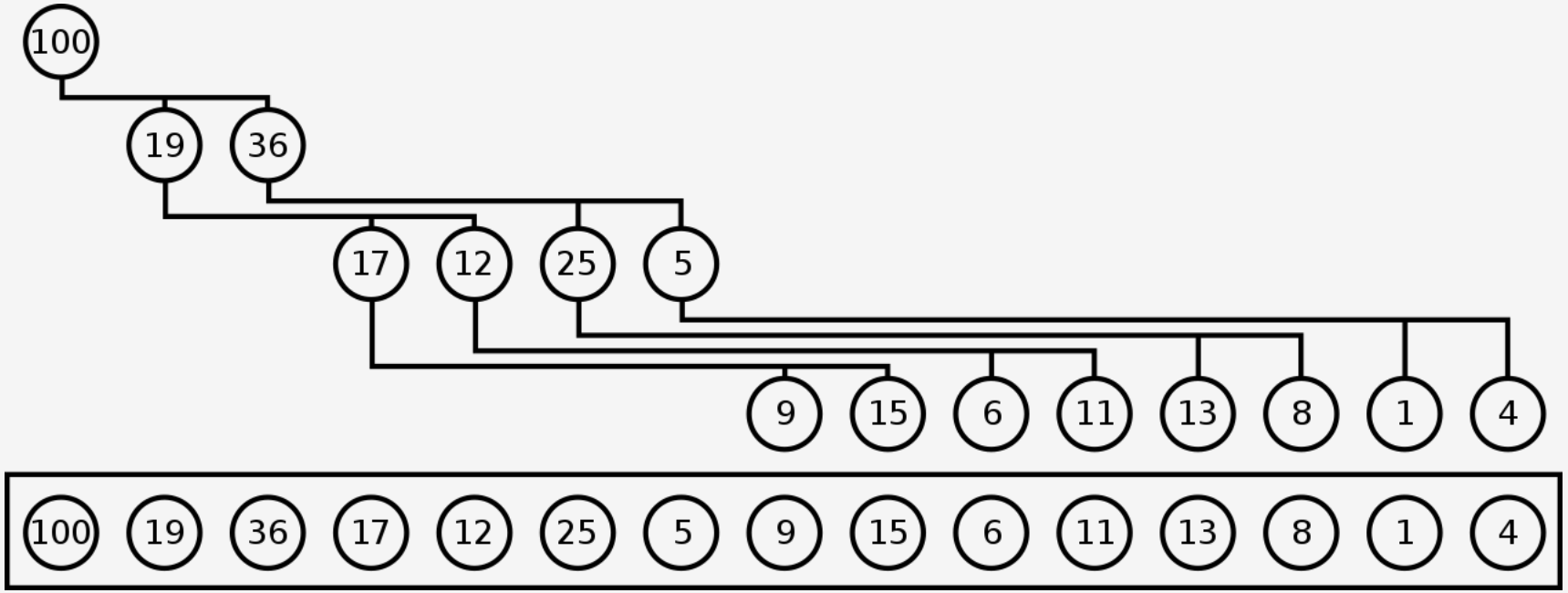
- jeśli **P** jest rodzicem **C**, to wartość **P** ≥ wartość **C** *(max heap)*
- jeśli **P** jest rodzicem **C**, to wartość **P** ≤ wartość **C** *(min heap)*

Operacje na kopcu

- stworzenie kopca z tablicy elementów
- znalezienie największego elementu *(peek)*
- dodanie elementu *(push)*
- usunięcie i zwrócenie największego elementu *(pop)*
- zwrócenie ilości elementów

Kopiec może służyć do implementacji kolejki priorytetowej *(priority queue)*.

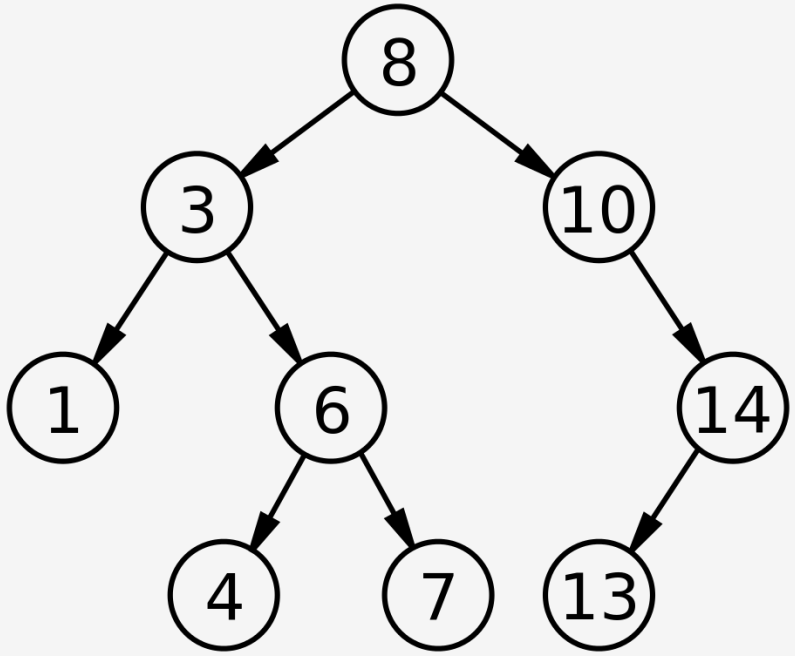
Kopiec binarny zapisany w tablicy



jeśli węzeł ma indeks ***n***, to jego dzieci mają indeksy **2*n* + 1** oraz **2*n* + 2**

jeśli węzeł ma indeks ***n***, to jego rodzic ma indeks **2 / *n***

Binarne drzewo poszukiwań (binary search tree, BST)



Specjalne drzewo, w którym:

- jeśli **L** jest lewym dzieckiem **N**, to wartość **L** ≤ wartość **N**
- jeśli **R** jest prawym dzieckiem **N**, to wartość **R** ≥ wartość **N**

Operacje na BST

- dodanie elementu
- usunięcie elementu
- wyszukanie elementu
- przechodzenie elementów w kolejności

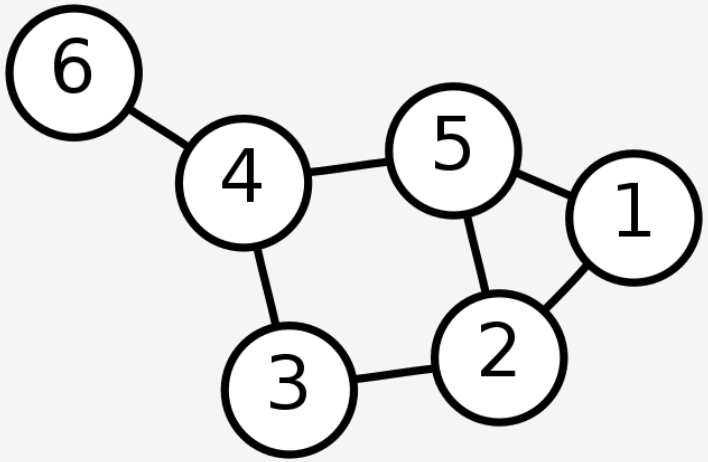
BST pozwala utrzymywać posortowaną kolekcję elementów.

Zadania!

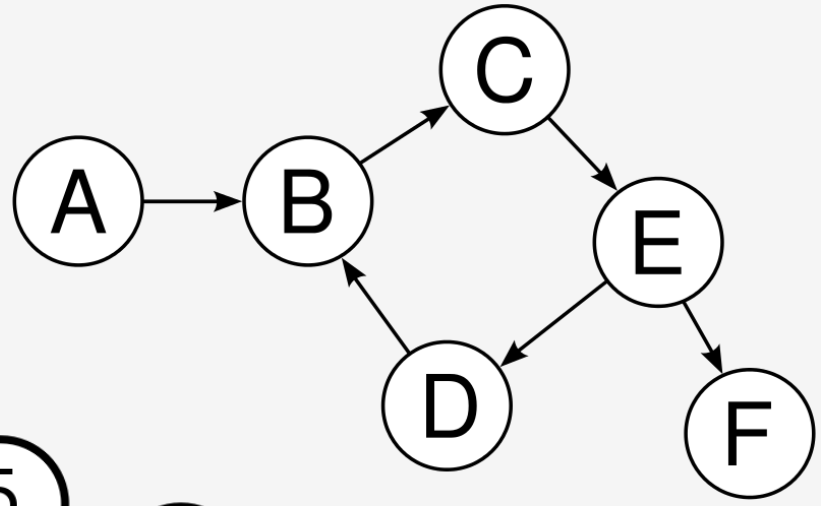
Tasks_Heap && Tasks_BST

Graf (graph)

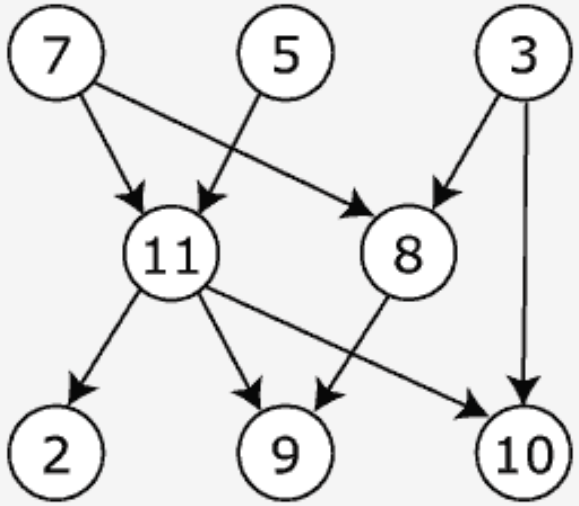
wierzchołek *(vertex)*



graf nieskierowany
(undirected graph)



graf skierowany
(directed graph)



acykliczny graf skierowany
(directed acyclic graph)

Złożoność obliczeniowa

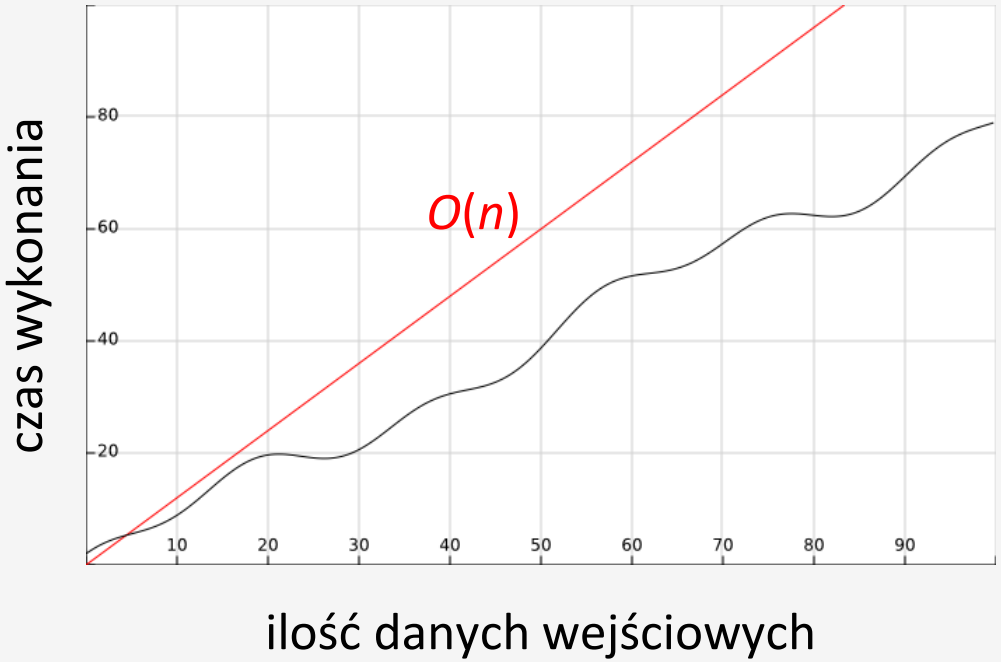
- **Teoria złożoności obliczeniowej (wg Wiki)** – dział teorii obliczeń, którego głównym celem jest określanie ilości zasobów potrzebnych do rozwiązania problemów obliczeniowych. Rozważanymi zasobami są takie wielkości jak czas, pamięć lub liczba procesorów.

Właściwości algorytmów ogólnie

- jednoznaczność
- skończoność
- uniwersalność
- poprawność
- wydajność

Złożoność obliczeniowa (computational complexity)

- złożoność czasowa *(time complexity)*
- złożoność pamięciowa *(space complexity/memory complexity)*

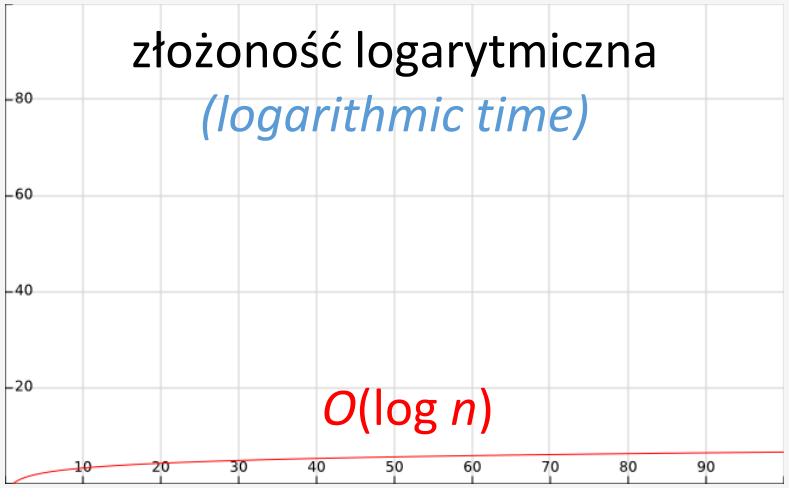


Złożoność obliczeniowa

złożoność stała
(constant time)

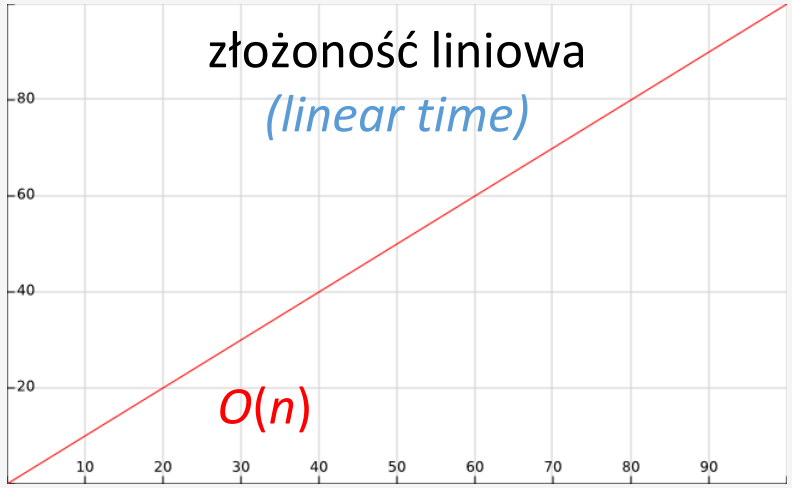
$O(1)$

złożoność logarytmiczna
(logarithmic time)



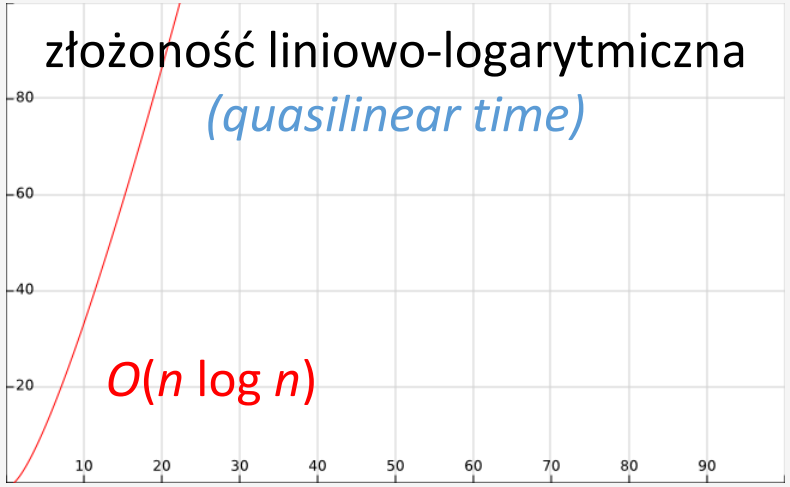
$O(\log n)$

złożoność liniowa
(linear time)



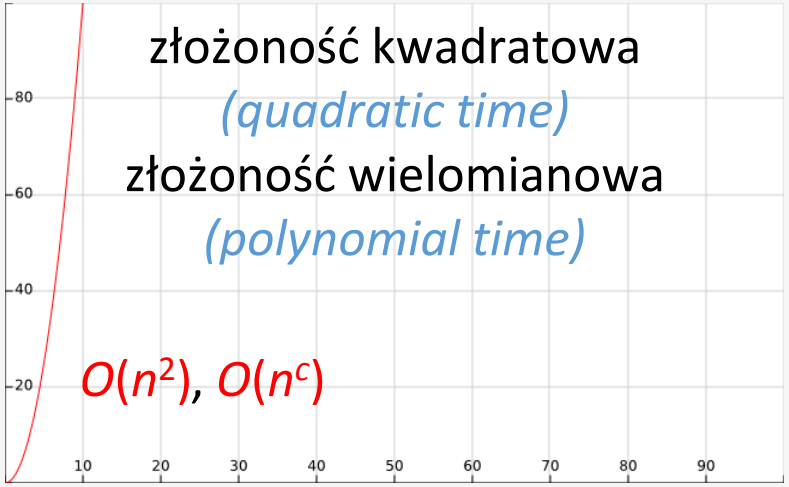
$O(n)$

złożoność liniowo-logarytmiczna
(quasilinear time)



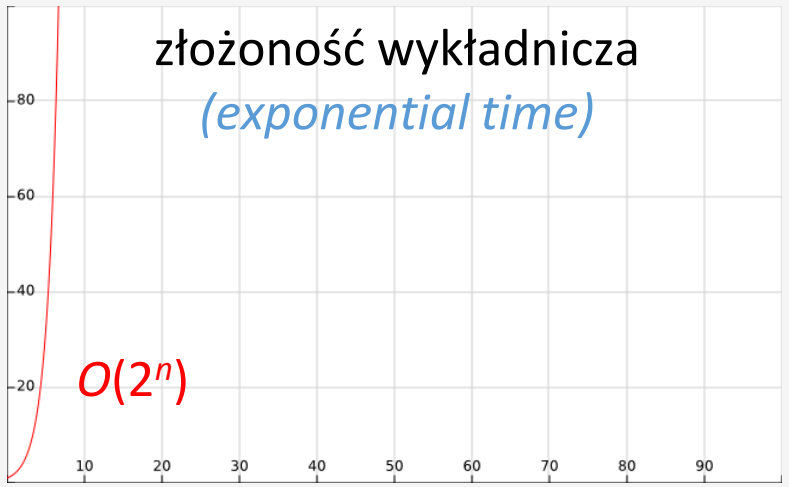
$O(n \log n)$

złożoność kwadratowa
(quadratic time)
złożoność wielomianowa
(polynomial time)



$O(n^2)$, $O(n^c)$

złożoność wykładnicza
(exponential time)



$O(2^n)$

Złożoność obliczeniowa

złożoność	nazwa	przykład
$O(1)$	stała	wyrażenie
$O(\log n)$	logarytmiczna	dzielenie na dwie części
$O(n)$	liniowa	pętla
$O(n \log n)$	liniowo-logarytmiczna	dziel i zwyciężaj
$O(n^2)$	kwadratowa	podwójna pętla (zagnieżdżona)
$O(n^c)$	wielomianowa	wielokrotnie zagnieżdżona pętla
$O(2^n)$	wykładnicza	przeszukiwanie wszystkich podzbiorów

Złożoność czasowa operacji na HashMap i HashSet

HashMap

operacja	złożoność
put	$O(1)$*
remove	$O(1)$
get	$O(1)$
size	$O(1)$
containsKey	$O(1)$
containsValue	$O(n)$
clear	$O(n)$

HashSet

operacja	złożoność
add	$O(1)$*
remove	$O(1)$
size	$O(1)$
contains	$O(1)$
clear	$O(n)$

* koszt zamortyzowany

Złożoność czasowa operacji na ArrayList

operacja	złożoność
get	$O(1)$
set	$O(1)$
size	$O(1)$
contains	$O(n)$
clear	$O(n)$

Złożoność czasowa operacji na ArrayDeque

operacja				złożoność
stos	kolejka	kolejka dwukierunkowa	lista	
push	offer	offerFist, 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				$O(n)$

* koszt zamortyzowany

Złożoność algorytmów sortowania

	czasowa	pamięciowa
selection sort	?	?
insertion sort	?	?
quicksort	?	?
merge sort	?	?

Złożoność algorytmów sortowania

	czasowa	pamięciowa
selection sort	$O(n^2)$	$O(1)$
insertion sort	$O(n^2)$	$O(1)$
quicksort	$O(n \log n)$	$O(\log n)$
merge sort	$O(n \log n)$	$O(n)$

Złożoność algorytmów sortowania

	czasowa			pamięciowa
	optymistyczna	przeciętna	pesymistyczna	
selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$
insertion sort	$O(n)$	$O(n^2)$	$O(n^2)$	$O(1)$
quicksort	$O(n \log n)$	$O(n \log n)$	$O(n^2)$	$O(\log n)$
merge sort	$O(n \log n)$	$O(n \log n)$	$O(n \log n)$	$O(n)$

Przestroga

„Przedwczesna optymalizacja jest źródłem wszelkiego zła.”

– Donald Knuth

„It is the very fact that computation has become very cheap in contrast with salaries of programmers, that squeezing the machines to yield their utmost in speed has become much less important that reliability, correctness, and organizational clarity. It is not only more urgent, but also much more costly to correct an efficient, but erroneous program, that to speed up a relatively slow, but correct program.”

– Niklaus Wirth (1974)

Przestroga

Trzy zasady optymalizacji:

1. nie optymalizuj
2. jeszcze nie optymalizuj
3. najpierw użyj profilera

ale...

Musisz wiedzieć na jakich ilościach danych będzie operować aplikacja.

https://blog.codinghorror.com/everything-is-fast-for-small-n/

Podsumowanie

Najważniejsze wnioski:

- wiedzieć jakie struktury danych mamy do dyspozycji i kiedy używać których (jakie są złożoności obliczeniowe ich operacji)
- nie wymyślać koła na nowo
- nie skupiać się na optymalizowaniu, ale mieć na uwadze złożoność obliczeniową
- rozpoznawać sytuacje, w których można wykorzystać rekurencję
- umieć czytać, analizować i implementować algorytmy