

Algorytmy i struktury danych – W01

Iteratory
Złożoności obliczeniowa cz. 1

Zawartość

- Uwagi ogólne nt. kolorystyki
- Iteracja
- Rekursja/rekurencja
- Iteratory
 - Interfejs Iterable<X>
 - Iterator na tablice
 - FilterIterator, Predicate
 - Iterator a pętla **foreach**
- Złożoność obliczeniowa cz. 1
 - Pojęcie algorytmu
 - Notacja i nazewnictwo
 - Przykład usprawniania algorytmu - potęgowanie

Uwagi ogólne

- Kolorы:
 - Brak grubej ramki – zawartość bezpośrednio związana z kursem
 - Niebieska gruba ramka – informacje programistyczne związane z praktyką lub środowiskami programistycznymi.
 - Fioletowa gruba ramka – inne informacje.
- Kody programów:
 - Pełne wersje znajdują się na ePortalu
 - Na wykładzie prezentowane kody będą tak formatowane, aby zajęły mniej miejsca oraz pewne metody mogą być pominięte.

Iteracja

- **Iteracja** - wielokrotne powtarzanie tego samego bloku czynności.
- **Liczba powtórzeń** może być określona jawnie (znana a priori)

```
i=0; s=0;  
do {  
    s+=a[i]; i++;  
} while (i<n);
```

- Lub wynikać z wystąpienia podczas przetwarzania **określonej sytuacji** (spełnienie / nie spełnienie określonego warunku)

Rekurencja

- **Rekurencja** – wywoływanie metody **przez samą siebie** (bezpośrednio lub pośrednio). Oznacza to realizację idei podziału problemu na podproblemy o tej samej naturze, lecz o mniejszych rozmiarach

```
public static void drawNumberSequence(int k) {  
    if (k==0) System.out.println(k);  
    else {  
        System.out.print(k+" ");  
        drawNumberSequence(k-1);  
    }  
}  
  
public static void drawPyramid(int n) {  
    if (n==0) System.out.println(n);  
    else {  
        drawNumberSequence(n);  
        drawPyramid(n-1);  
    }  
}  
  
public static void main(String[] args) {  
    drawPyramid(7);  
}
```

7	6	5	4	3	2	1	0
6	5	4	3	2	1	0	
5	4	3	2	1	0		
4	3	2	1	0			
3	2	1	0				
2	1	0					
1	0						
0							

aisd.W01a.RekurencjaDemo

Iteracyjne przetwarzanie tablic 1/3

- Przetwarzanie tablicy często sprowadza się do użycia zmiennej indeksowanej, której wartość zmienia się w pętli w jawnie określony sposób.

```
public class Student {  
    int indexNo;  
    double scholarship;  
    public Student(int no, double amount) {  
        indexNo=no;  
        scholarship=amount;  
    }  
    public void increaseScholarship(double amount) {  
        scholarship+=amount;  
    }  
    public void showData() {  
        System.out.printf("%6d %8.2f\n", indexNo, scholarship);  
    }  
}
```

aisd.W01a.Student

C.d.

- Operacje iteracyjne na tablicy studentów można zrealizować następująco:

```
// Załóżmy, że mamy następującą listę studentów:  
Student [] s = new Student[5];  
s[0]=new Student(1,500);  
s[1]=new Student(2,400);  
s[2]=new Student(3,0);  
s[3]=new Student(4,500);  
s[4]=new Student(5,700);  
// Zwiększenie stypendium wszystkim studentom o kwotę dodatek:  
for (int i = 0; i< s.length; i++)  
    s[i]. increaseScholarship(50);  
// Wyświetlenie listy stypendiów:  
for (int i = 0; i< s.length; i++)  
    s[i]. showData();
```

aisd.W01a.StudentDemo

1	550,0
2	450,0
3	50,0
4	550,0
5	750,0

Iteratory

- **Iterator** to obiekt klasy, który służy do przechodzenia po **elementach** jakiejś **kolekcji** w sposób **uporządkowany** tak, aby każdy element odwiedzić **dokładnie raz**.
- To znaczy, że widzimy kolekcję jako **strukturę liniową**.
- Jest to *wzorzec projektowy*.
- Ma na celu ukrycie wewnętrznej struktury kolekcji.
- Używanie iteratorów (zamiast dostępu bezpośredniego do elementów kolekcji metodami specyficznymi dla tej kolekcji) pozwala w przyszłości zamienić kolekcję na inną bez potrzeby zmiany kodu używającego iteratory.
- W bardziej złożonych kolekcjach (drzewa, grafy itd.) znalezienie „następnego” elementu może być trudne dla użytkownika kolekcji, stąd udostępnienie przez twórcę kolekcji iteratora ułatwia korzystanie z takiej kolekcji
- Może istnieć wiele sposobów przechodzenia po kolekcji, wtedy dla każdego sposobu należy stworzyć odpowiedni iterator.

Iteratory

- W teorii wzorców projektowych Gramma i inni zaproponowano, aby iterator pozwalał na:
 - Przejście na **początek kolekcji** (*first*)
 - **Odczytanie bieżącego** elementu kolekcji (*current*)
 - **Przejście do przodu** do kolejnego elementu (*next*)
 - Przejście na **koniec kolekcji** (*last*)
 - **Przejście do tyłu** do poprzedniego elementu (*previous*)
 - **Sprawdzenie**, czy iterator wyszedł **poza kolekcję** (*isDone*)
- Rozbudowane iteratory mogą również np. usuwać element, na który wskazują
- W praktyce okazuje się, że zdecydowana większość użycia iteratorów polega na jednokrotnym przejściu po kolekcji od pierwszego elementu do ostatniego bez potrzeby wielokrotnego odczytywania tego samego elementu.
- Dodatkowo dla pewnych struktur przejście do tyłu, rozpoczęcie od końca, czy ponownie od nowa jest bardzo utrudnione i nieefektywne.
- Twórcy Javy stworzyli zatem iteratory jednokierunkowe jednokrotnego użycia jako iterator podstawowy. Ma on następujące cechy:
 - Po stworzeniu iterator jest ustawiany na początku kolekcji
 - Poruszamy się tylko do przodu
 - Możemy tylko raz odczytać bieżącą pozycję
 - Nie możemy wrócić na początek
 - Gdy dojdziemy do końca kolekcji iterator jest w zasadzie bezużyteczny.

Iteratory w Javie

- W Javie jest interfejs `Iterator` w dwóch wariantach:
 - Jako ogólny interfejs
 - Jako interfejs generyczny
- Ich definicje znajdują się pakiecie `java.util`
- Posiada tylko trzy metody oraz założenie, że **konstruktor** dla konkretnej kolekcji ma go stawiać **PRZED** pierwszym elementem.
- Metoda `hasNext()` zwraca informację, czy istnieje kolejny element.
- Metoda `next()` „przeskakuje” nad elementem, zwraca go i staje **PRZED** kolejnym elementem. Jeśli „bieżącego” elementu nie ma, kończy się wyjątkiem `java.util.NoSuchElementException`.
- Metoda `remove()` usuwa dopiero co przeskoczony element. Ponowne użycie bez uruchomienia `next()` kończy się wyjątkiem `java.lang.IllegalStateException`.
- Jeżeli metoda `remove()` nie może być wykonana (bo np. kolekcja jest niemodyfikowalna) kończy się wyjątkiem `java.lang.UnsupportedOperationException`

```
interface Iterator{  
    boolean hasNext();  
    Object next();  
    void remove();  
}
```

```
interface Iterator<T>{  
    boolean hasNext();  
    T next();  
    void remove();  
}
```

Interfejs Iterable<X>

- Aby w pełni wykorzystać mechanizm iteratorów kolekcje muszą zapewniać swoje poprawne iteratory. W tym celu klasa kolekcji musi zapewniać implementację interfejsu Iterable<X>.
- Znajduje się on w pakiecie `java.util`
- Jest interfejsem ogólnym lub generycznym
- Ma tylko jedną metodę:
 - `Iterator iterator();`
 - lub
 - `Iterator<X> iterator();`
- Dzięki implementacji tego interfejsu, kolekcję będzie można używać również w nowej formie pętli **for** (patrz kilka slajdów dalej)

Diagram operacji z użyciem iteratorów

- Przykład działania iteratora dla cztero-elementowej kolekcji liniowej i poniższej sekwencji instrukcji:

```
SomeCollection<X> col=...; // kolekcja zawiera wartości  
// Val0,Val1,Val2,Val3  
Iterator iter=col.iterator(); // pobranie iteratora  
// z kolekcji  
iter.hasNext(); //true  
X value=iter.next(); // value=Val0  
iter.hasNext(); //true  
X value=iter.next(); // value=Val1  
iter.remove(); // usunięto Val1  
iter.hasNext(); //true  
X value=iter.next(); // value=Val2  
iter.hasNext(); //true  
X value=iter.next(); // value=Val3  
iter.hasNext(); //false
```

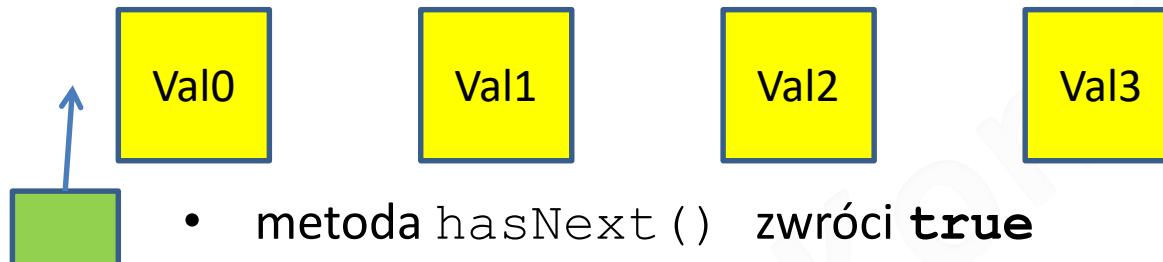
aisd.W01a.IteratorDemo

Działanie iteratorów

- Stan abstrakcyjnej kolekcji liniowej

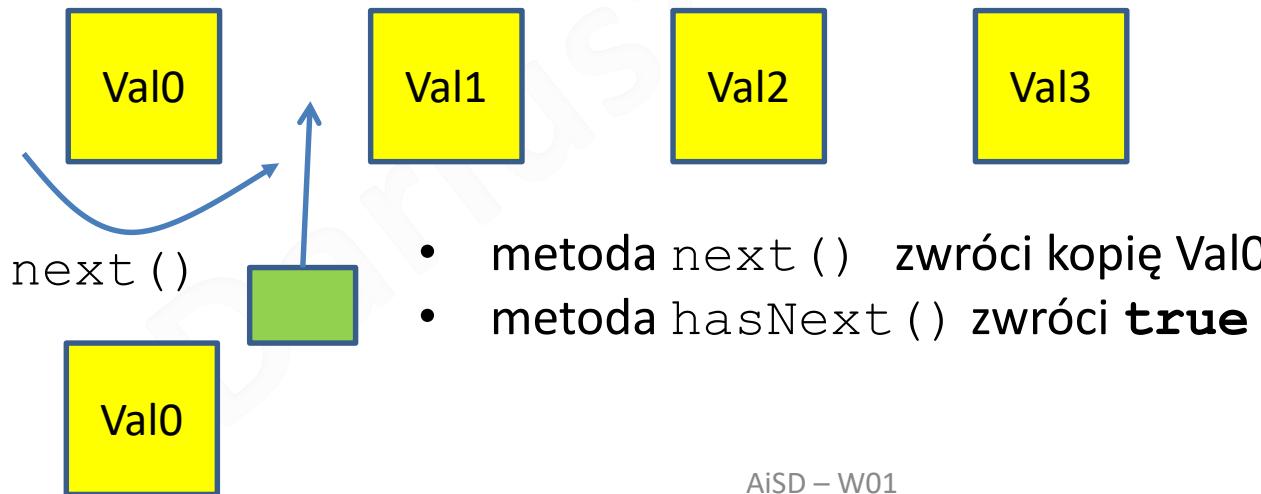


- Stan iteratora po skonstruowaniu



- metoda `hasNext ()` zwróci **true**

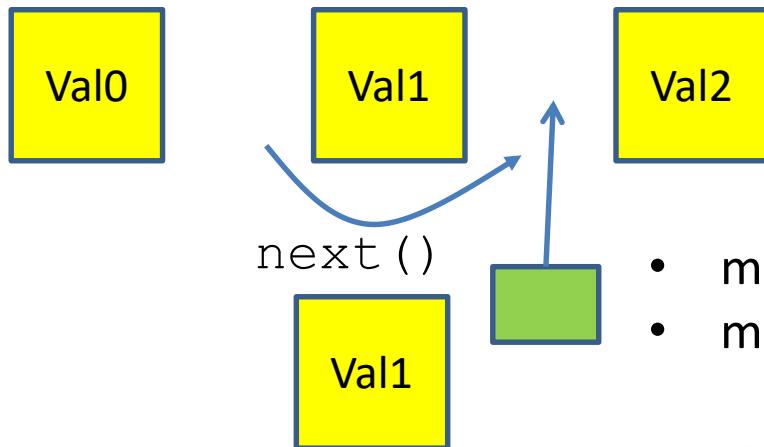
- Stan iteratora po wykonaniu `next ()`



- metoda `next ()` zwróci kopię `Val0`
- metoda `hasNext ()` zwróci **true**

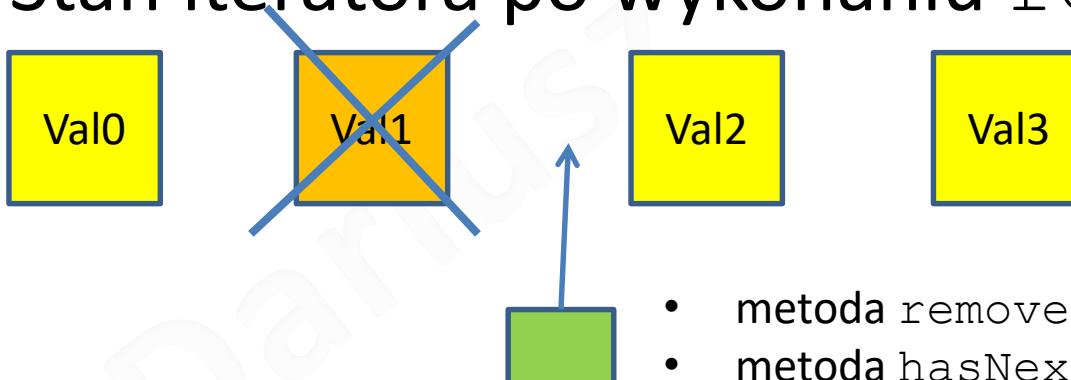
Działanie iteratorów

- Stan iteratora po wykonaniu kolejnego next ()



- metoda `next ()` zwróci kopię `Val1`
- metoda `hasNext ()` zwróci `true`

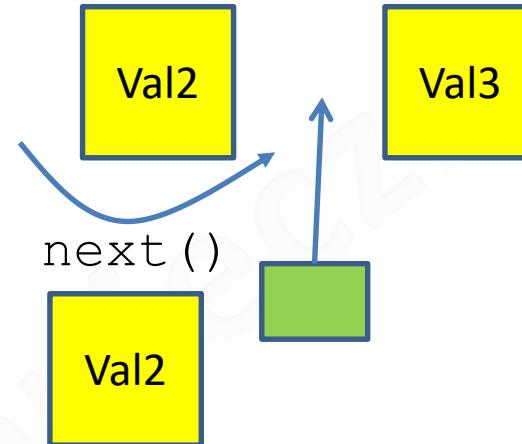
- Stan iteratora po wykonaniu remove ()



- metoda `remove ()` usunie `Val1` z kolekcji
- metoda `hasNext ()` zwróci `true`

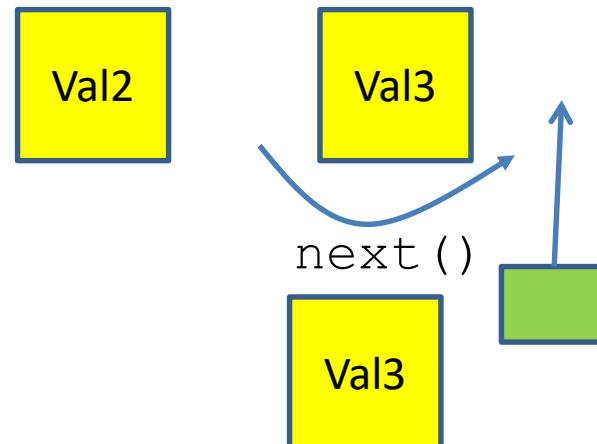
Działanie iteratorów

- Stan iteratora po wykonaniu kolejnego next ()



- metoda `next()` zwróci kopię `Val2`
- metoda `hasNext()` zwróci `true`

- Stan iteratora po wykonaniu kolejnego next ()



- metoda `next()` zwróci kopię `Val3`
- metoda `hasNext()` zwróci `false`

Iterator dla zwykłej tablicy.

- Dla zwykłej tablicy nie ma WPROST iteratorów.
- Jak mogłyby wyglądać implementacja takiego iteratora.

```
public class ArrayIterator<T> implements Iterator<T> {  
    private T array[];  
    private int pos = 0;  
  
    public ArrayIterator(T anArray[]) {  
        array = anArray;  
    }  
    public boolean hasNext() {  
        return pos < array.length;  
    }  
    public T next() throws NoSuchElementException {  
        if (hasNext())  
            return array[pos++];  
        else  
            throw new NoSuchElementException();  
    }  
    public void remove() {  
        throw new UnsupportedOperationException();  
    }  
}
```

aisd.util.ArrayIterator

Iterator odwrotny

- W pomyśle Gramma iterator był dwukierunkowy, miał symetryczne metody dla operacji w odwrotnym kierunku. Dzięki temu łatwo było napisać **ogólny** iterator odwrotny.
- W przypadku bibliotecznego iteratora Javy jest to niemożliwe, ale część kolekcji może zwracać iterator odwrotny jako zwykły iterator.
- Dla tablicy mogłoby to wyglądać jak poniżej:

```
public class ArrayReverseIterator<T> implements Iterator<T> {  
    private T array[];  
    private int pos;  
  
    public ArrayIterator(T anArray[]) {  
        array = anArray;  
        pos = array.length;  
    }  
    public boolean hasNext() {  
        return pos > 0;  
    }  
    public T next() throws NoSuchElementException {  
        if (hasNext())  
            return array[--pos];  
        else  
            throw new NoSuchElementException();  
    }  
    public void remove() {  
        throw new UnsupportedOperationException();  
    }  
}
```

aisd.util.ArrayReverselterator

FilterIterator, Predicate

- Gdy chcemy przejść po kolekcji i wykonać pewną operację dla wybranych (pewnym warunkiem logicznym) elementów, warto użyć filtra.
- Filtr taki musi posiadać metodę sprawdzającą czy kolejny element spełnia nasz warunek (predykat).
- W Javie nie ma standardowo zaimplementowanego takiego iteratora.
Implementacja mogłaby wyglądać następująco:

```
public interface Predicate<T>{  
    boolean accept(T arg);  
}
```

aisd.util.Predicate

```
public final class FilterIterator<T> implements Iterator<T>{  
  
    private Iterator<T> iterator;  
    private Predicate<T> predicate;  
  
    private T elemNext=null;  
    private boolean bHasNext=true;  
  
    public FilterIterator(Iterator<T> iterator, Predicate<T> predicate) {  
        super();  
        this.iterator = iterator; // nie może być null  
        this.predicate = predicate; //nie może być null  
        findNextValid();  
    }  
}
```

aisd.util.FilterIterator

FilterIterator 2/2

```
private void findNextValid() {  
    while (iterator.hasNext()) {  
        elemNext = iterator.next();  
        if (predicate.accept(elemNext)) {  
            return;  
        }  
    }  
    bHasNext=false;  
    elemNext=null;  
}  
@Override  
public boolean hasNext() {  
    return bHasNext;  
}  
  
@Override  
public T next() {  
    T nextValue = elemNext;  
    findNextValid();  
    return nextValue;  
}  
}
```

aisd.util.FilterIterator

Pętla foreach

- Od Javy 5.0 jest dostępna tzw. pętla foreach:

```
for ({deklaracja_zmiennej_pętli} : {kolekcja lub tablica})  
    {ciało_pętli}
```

- Wykonanie tej pętli oznacza przyporządkowanie zmiennej pętli kolejnych wartości z kolekcji (tablicy) i wykonywanie dla niej ciała pętli
- Aby to jednak wykonać kolekcja musi zwracać iterator, czyli musi zapewniać interfejs Iterable<X>.
 - Wyjątek – **zwykła** tablica nie ma iteratora, ale pętla **foreach** działa również dla niej
- Nie jest to nowy mechanizm tylko skrót notacyjny dla szczególnej postaci pętli for:

```
for (TypX x:collection)  
{  
    ciało_pętli  
}
```

```
for (Iterator<TypX> iter=collection.iterator(); iter.hasNext();) {  
    {  
        TypX x=iter.next();  
        ciało_pętli  
    }  
}
```

Iterator a kolekcja

- Co się stanie, gdy zmodyfikujemy kolekcję w trakcie przechodzenia po niej iteratorem?
 - Możemy skasować pozycję, przed którą stoi iterator
 - Możemy skasować pozycję, za którą stoi iterator
 - Możemy dodać pozycję, przed lub za iteratorem
 - Dodać element na końcu, gdy iterator doszedł do końca
 - Kilka powyższych modyfikacji!
 - Programując współbieżnie modyfikacje może robić inny proces!
 - itd. itp.
- Modyfikacja może zostać dokonana przez metody dla tej kolekcji lub metodę INNEGO iteratora.
- Wybrano najprostsze rozwiązanie: jeśli struktura kolekcji ulegnie modyfikacji, próba użycia iteratora spowoduje wyjątek `ConcurrentModificationException`.
- Oczywiście nie dotyczy to wywołania funkcji `remove()` rozważanego iteratora.
- W ramach tego wykładu ten aspekt działania iteratorów będzie pomijany, żeby nie komplikować przykładów.

Co powinna zwracać operacja next ()

- Pytanie jest: jeśli operacja next () zwraca referencję na obiekt, to czy to jest:
 - obiekt oryginalny z kolekcji (czyli otrzymujemy kopię REFERENCJI)
 - kopia obiektu z kolekcji ?
- Odpowiedź: **kopia referencji**, czyli mamy bezpośredni dostęp do elementu
- To samo pytanie dla kolekcji typów prostych? – odpowiedź: **kopię wartości**
- W przypadku obiektów w kolekcjach UPORZĄDKOWANYCH zmiana **zawartości** obiektu powoduje niebezpieczeństwo zniszczenia porządku elementów.

Przykład działania iteratorów 1

- Tablica obiektów klasy Student – iteratory i pętle **foreach**

```
// Załóżmy, że mamy listę studentów z poprzednich slajdów:  
Student [] s = new Student[5];  
...  
// Zwiększenie stypendium wszystkim studentom o kwotę 50:  
Iterator<Student> iterStud=new ArrayIterator<Student>(s);  
while(iterStud.hasNext())  
    iterStud.next().increaseScholarship(50);  
// Wyświetlenie listy stypendiów:  
iterStud=new ArrayIterator<Student>(s);  
while(iterStud.hasNext())  
    iterStud.next().showData();
```

aisd.W01a.StudentDemo

```
// Załóżmy, że mamy listę studentów z poprzednich slajdów:  
Student [] s = new Student[5];  
...  
// Zwiększenie stypendium wszystkim studentom o kwotę 50:  
for (Student student:s)  
    student.increaseScholarship(50);  
// Wyświetlenie listy stypendiów:  
for (Student student:s)  
    student.showData();
```

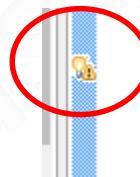
Modyfikacja w tablicy int?

- Tablica int
- Próba modyfikacji elementów w pętli
- Modyfikowana jest wartość lokalnej kopii! – kompilator o tym ostrzega!
- Porównanie ze stroną 20 „Pętla **foreach**”

```
int [] array={1,2,3,4,5,6};  
for(int elem:array)  
    System.out.print(elem+",");  
System.out.println();  
  
for(int elem:array)  
    elem+=10;  
  
for(int elem:array)  
    System.out.print(elem+",");  
System.out.println();
```

```
1,2,3,4,5,6,  
1,2,3,4,5,6,
```

aisd.W01a.ArrayIntModificationDemo



```
for(int elem:array)  
    elem+=10;
```



The value of the local variable elem is not used
elem+=10;

Przykład działania iteratorów 3

- Modyfikacja zawartości obiektów klasy Student dokonuje się (przykład dwa slajdy wcześniej)
- Czy można „podmienić” studenta/studentów na nowych?

aisd.W01a.StudentModificationDemo

```
// Załóżmy, że mamy listę studentów z poprzednich slajdów:  
Student [] s = new Student[5];  
...  
for (Student student:s)           // to samo ostrzeżenie jak  
    student=new Student(10,1000);   // dla tablicy int-ów  
for (Student student:s)  
    student. showData();
```

```
// Załóżmy, że mamy listę studentów z poprzednich slajdów:  
Student [] s = new Student[5];  
...  
Iterator<Student> iter=new ArrayIterator<Student>(s);  
Student stud=iter.next();           // tu widać wprost, że operujemy  
stud=new Student(10,1000);          // na lokalnej zmiennej  
for (Student student:s)  
    student.showData();
```

- Zmian w tablicy nie będzie, bo działamy na lokalnych **kopiach** referencji

ZŁOŻONOŚĆ OBliczeniowa cz. 1

Algorytm

- Nie ma jednej ustalonej definicji:
 - Klasyczna: **jednoznaczny** przepis obliczenia w skończonym czasie pewnych danych *wejściowych* do pewnych danych *wynikowych*
 - PWN: skończony ciąg jasno zdefiniowanych czynności, koniecznych do wykonania pewnego rodzaju zadań. Sposób postępowania prowadzący do rozwiązania problemu.
 - ...
- Cechy poprawnego algorytmu:
 - Otrzymuje **dane wejściowe**
 - Generuje **dane wyjściowe**
 - **Wykonalność** – polecenia zawarte w algorytmie są wykonywalne, tzn. dostępne, a pisząc algorytm wystarczy się nimi tylko posłużyć.
 - **Jednoznaczność** – dla tych samych danych wejściowych otrzymamy te same dane wyjściowe
 - **Determinizm** – dane pośrednie zależą od danych wejściowych i poprzednio wykonanych kroków
 - **Skończoność** – zatrzymuje się po wykonaniu skończonej liczby instrukcji
 - **Poprawność** – wynik jest poprawny
 - **Ogólność** – można go zastosować dla szerokiego zbioru danych wejściowych

Algorytmy

- Każdy program komputerowy jest w zasadzie algorytmem. Jednak algorytmem są też:
 - Instrukcje użycia narzędzi i maszyn
 - Przepisy kulinarne, leków itp.
 - Opis procesu technologicznego
 - Proces obliczeń inżynierskich
 - itd.
- Algorytm może być wyrażony:
 - W języku potocznym
 - W kodzie w znanym języku programowania
 - W pseudokodzie
 - Za pomocą diagramów
 - Mieszanina powyższych
- Podział problemów:
 - Rozwiązywalne, nierozwiązywalne
 - Problemy decyzyjne, problemy obliczeniowe

Notacja i nazewnictwo

- **Krok** – operacja elementarna (zależy od problemu/algorytmu):
 - Operacja algebraiczna (dodawanie, odejmowanie, ...)
 - Porównanie ($=$, \leq , ...)
 - Kopiowanie, przypisanie ($=$)
- **Rozmiar problemu (n)** – liczba danych wejściowych:
 - Teoria: liczba bitów
 - Realna ilość: liczba danych do posortowania itp.
 - Użyteczne: rozmiar macierzy kwadratowej
 - Więcej niż jedna liczba: np. dla grafów
- **Złożoność (obliczeniowa) problemu** wyrażana jest jako funkcja od n , np. $f(n)$ oznaczającą **liczbę kroków** do wykonania dla **danych wejściowych rozmiaru n** .

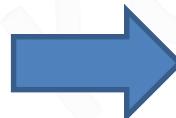
Fakt

- **Najprostszy algorytm** rozwiązania problemu często **nie jest najbardziej wydajny**. Dlatego podczas projektowania algorytmu **nie zadowalaj się każdym działającym algorytmem**.

Jak obliczyć k^n ? (n liczba naturalna)

Proste rozwiązanie – z definicji matematycznej:

$$k^0=1$$
$$k^n=k \cdot k^{(n-1)} \text{ dla } n>=1$$



W najgorszym przypadku
 n mnożeń

Co, gdy n ma 100 cyfr (jest rzędu 10^{100})?
Tak jest w problemach kryptografii

```
Power (k, n)
{
    Result=1;
    while (n>0)
    {
        Result=Result*k;
        n--;
    }
    return Result;
}
```

k^n – lepszy algorytm (1)

- Można użyć matematycznych własności potęgowania:

$$k^{2n} = (k^n)^2$$

$$k^{2n+1} = k(k^n)^2$$

- Np. :

$$k^{89} = k(k^{44})^2$$

$$k^{44} = (k^{22})^2$$

$$k^{22} = (k^{11})^2$$

$$k^{11} = k(k^5)^2$$

$$k^5 = k(k^2)^2$$

$$k^2 = k * k$$

```
Power(k, n)
```

```
{
```

```
    if(n==0) return 1;  
    if(n==1) return k;  
    result=Power(k,n/2);  
    if(n%2==0)  
        return result*result;  
    else  
        return k*result*result;
```

```
}
```

W najgorszym przypadku
 $2 * \log(n)$ mnożeń

Rekurencja !

Np. na kartach graficznych
nie było możliwości rekurencji

k^n – jeszcze lepszy algorytm (2)

- Przedstawienie potęgi w reprezentacji binarnej i rozbicie na składniki o potęgach będących potęgami dwójki.

k	1
k^2	0
k^4	0
k^8	1
k^{16}	1
k^{32}	0
k^{64}	1

$$k^{89} = k^{64} * k^{16} * k^8 * k^1$$

$$k^{89} = k^{1011001}$$

```
Power (k, n)
{
    Result=1;
    pow=k;
    while (n>0)
    {
        if ((n%2)==1)
            Result*=pow;
        pow*=pow;
        n/=2;
    }
    return Result;
}
```

W najgorszym przypadku
 $2 \cdot \log(n)$ mnożeń

Brak rekurencji !

Algorytmy i struktury danych – W02

Teoria złożoności cz. 2/4,
Listy, listy wiązane, iterator dla list

Zawartość

- Złożoność cz. 2:
 - Notacja asymptotyczna Θ, O, Ω
- Listy:
 - Opis
 - Interfejs `IList<T>`
 - Interfejs `ListIterator<T>`
 - Klasa `AbstractList<T>`
- Implementacja listy na tablicy: klasa `ArrayList<T>`
- Listy wiązane:
 - Jednokierunkowa, prosta, z głową, bez strażnika – L1KzG, klasa `OneWayLinkedListWithHead<T>`

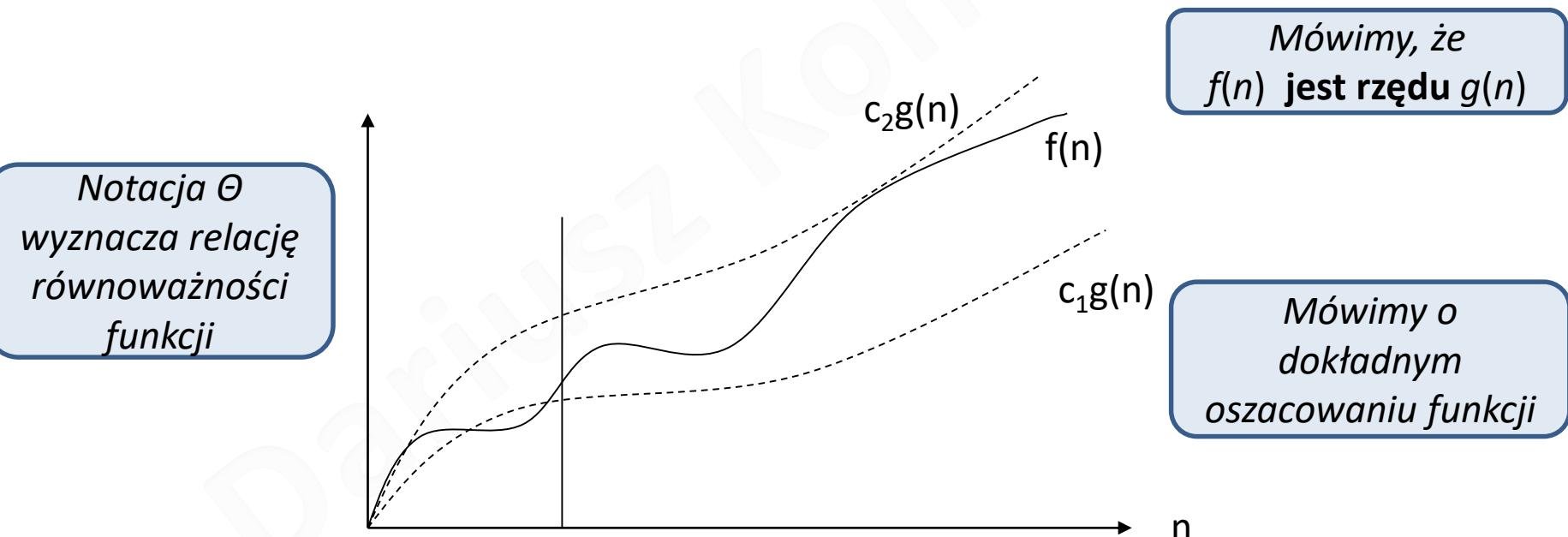
Notacje asymptotyczne

- Przypomnienie:
 - n : liczba danych wejściowych
 - $f(n)$: funkcja mówiąca ile kroków obliczeniowych należy wykonać dla danych o długości n
- Funkcja $f(n)$ może:
 - mieć postać skomplikowanego wzoru matematycznego
 - Zależeć nie tylko od n , ale od konkretnych danych
 - Może być niedeterministyczna
- Zamiast podawać dokładny wzór, ważniejszy jest rząd funkcji. Do tego celu wykorzystuje się notacje asymptotyczne.

Notacja asymptotyczna – Duże Theta

- Dla danej funkcji $g(n)$ oznaczamy jako $\Theta(g(n))$ zbiór wszystkich funkcji $f(n)$ posiadających właściwość, że istnieją takie dodatnie wartości c_1 i c_2 oraz n_0 , że dla każdego $n \geq n_0$ zachodzi:

$$c_1 g(n) \leq f(n) \leq c_2 g(n)$$



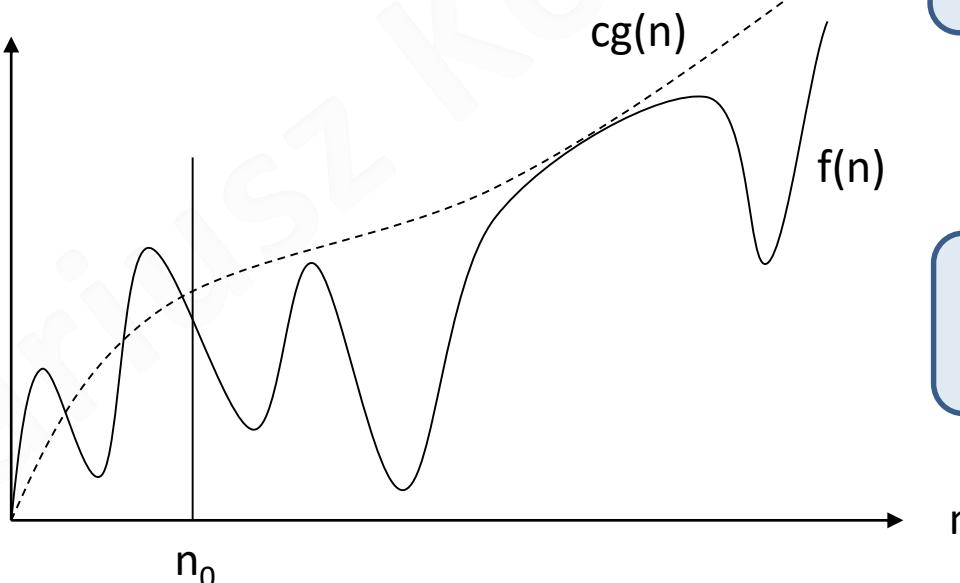
Duże theta - przykład

- Niech $f(n)=n^2+100n+1000$
- Zatem $f(n)=\Theta(n^2)$, ponieważ:
 - dla $n_0=1000, c_1=1, c_2=10$ oraz dla każdego $n \geq n_0$:
 - $n^2+100n+1000 > n^2$
 - $n^2+100n+1000 < 10n^2$
 $(1.000.000+100.000+1000 < 10 * 1.000.000)$

Notacja asymptotyczna – Duże O

- Dla danej funkcji $g(n)$ oznaczamy jako $O(g(n))$ zbiór wszystkich funkcji $f(n)$ posiadających właściwość, że istnieją takie dodatnie wartości c oraz n_0 , że dla każdego $n \geq n_0$ zachodzi:

$$f(n) \leq c \cdot g(n)$$



Mówimy, że
 $f(n)$ jest rzędu
co najwyżej $g(n)$

Mówimy o górnym
ograniczeniu funkcji

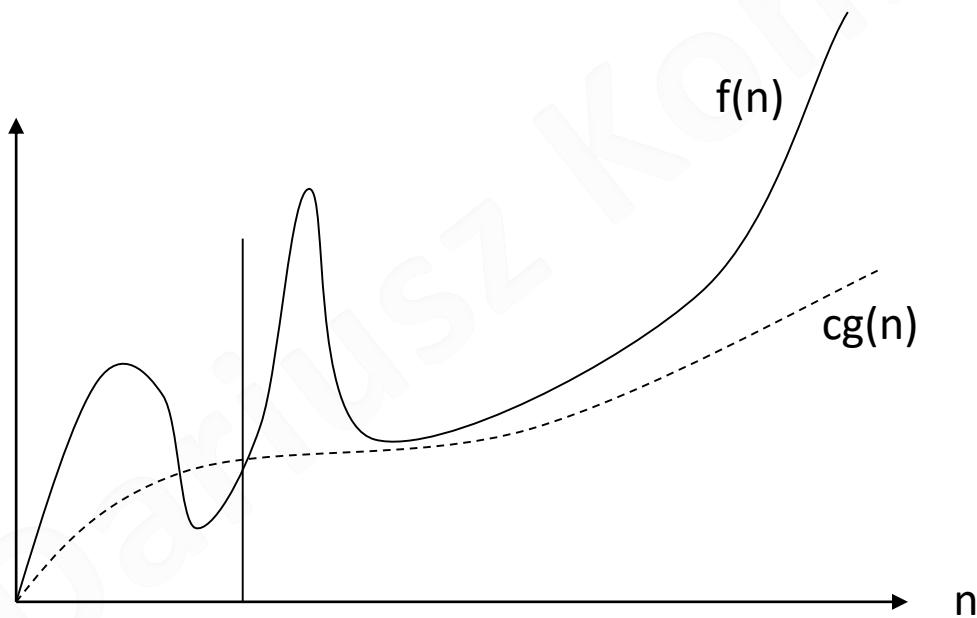
Duże O - przykład

- Niech $f(n)=n |\sin(n)|$
- Zatem $f(n)=O(n)$, ponieważ:
 - dla $n_0=1, c=1$, dla każdego $n_0>=n$ zachodzi:
 - $|\sin(n)|<=1$
 - $n |\sin(n)|<=n$
- Również $f(n)=O(n^2)$, jednak $O(n)$ jest wolniej rosnącą funkcją. Notacja $O(\dots)$ w teorii złożoności algorytmów jest używana do określenia ograniczenia z góry złożoności, stąd podaje się (wybiera) funkcję jak najwolniej rosnącą.

Notacja asymptotyczna – Duże Ω

- Dla danej funkcji $g(n)$ oznaczamy jako $\Omega(g(n))$ zbiór wszystkich funkcji $f(n)$ posiadających właściwość, że istnieją takie dodatnie wartości c oraz n_0 , że dla każdego $n \geq n_0$ zachodzi:

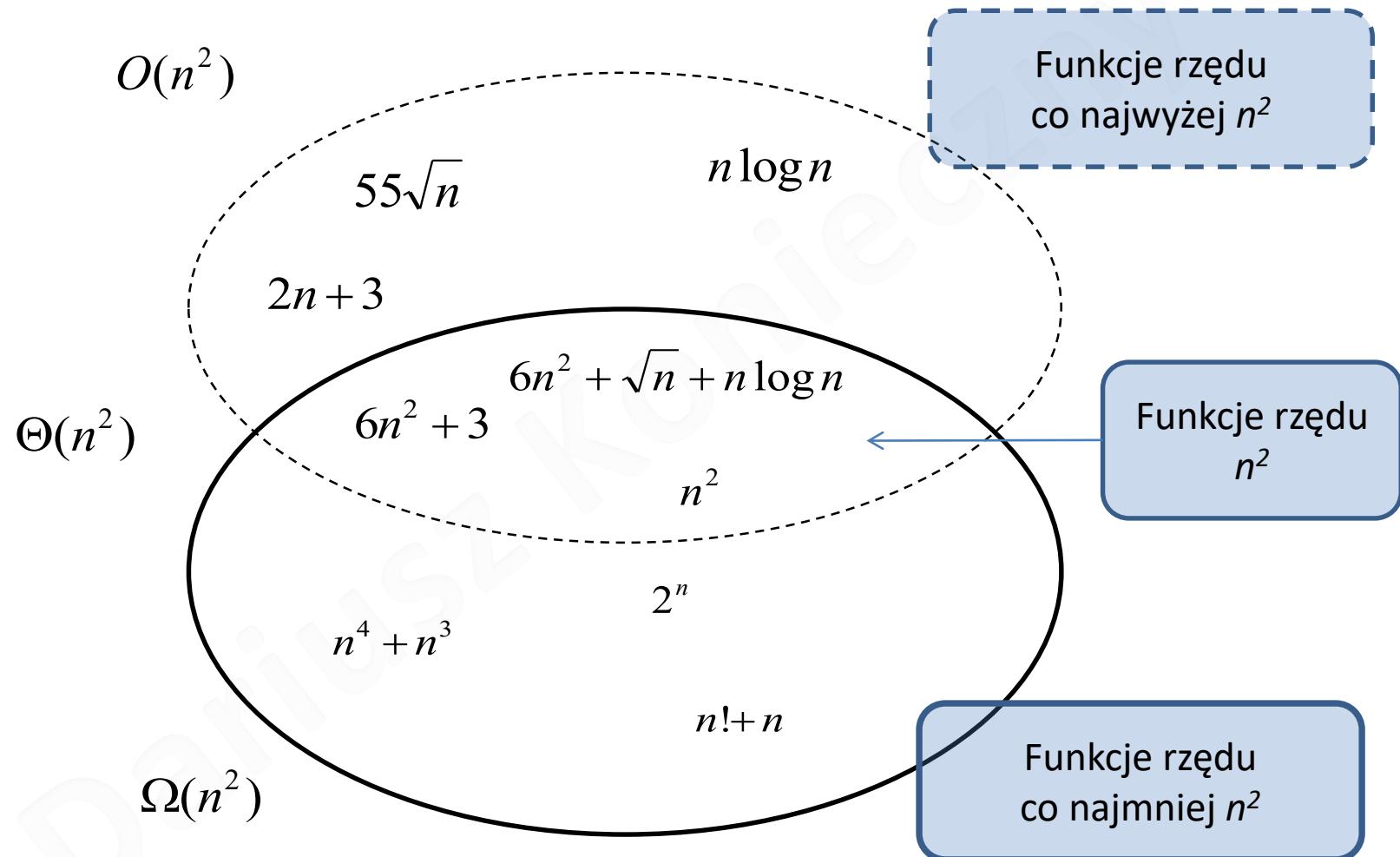
$$f(n) \geq c \cdot g(n)$$



Mówimy, że
 $f(x)$ jest rzędu
co najmniej $g(x)$

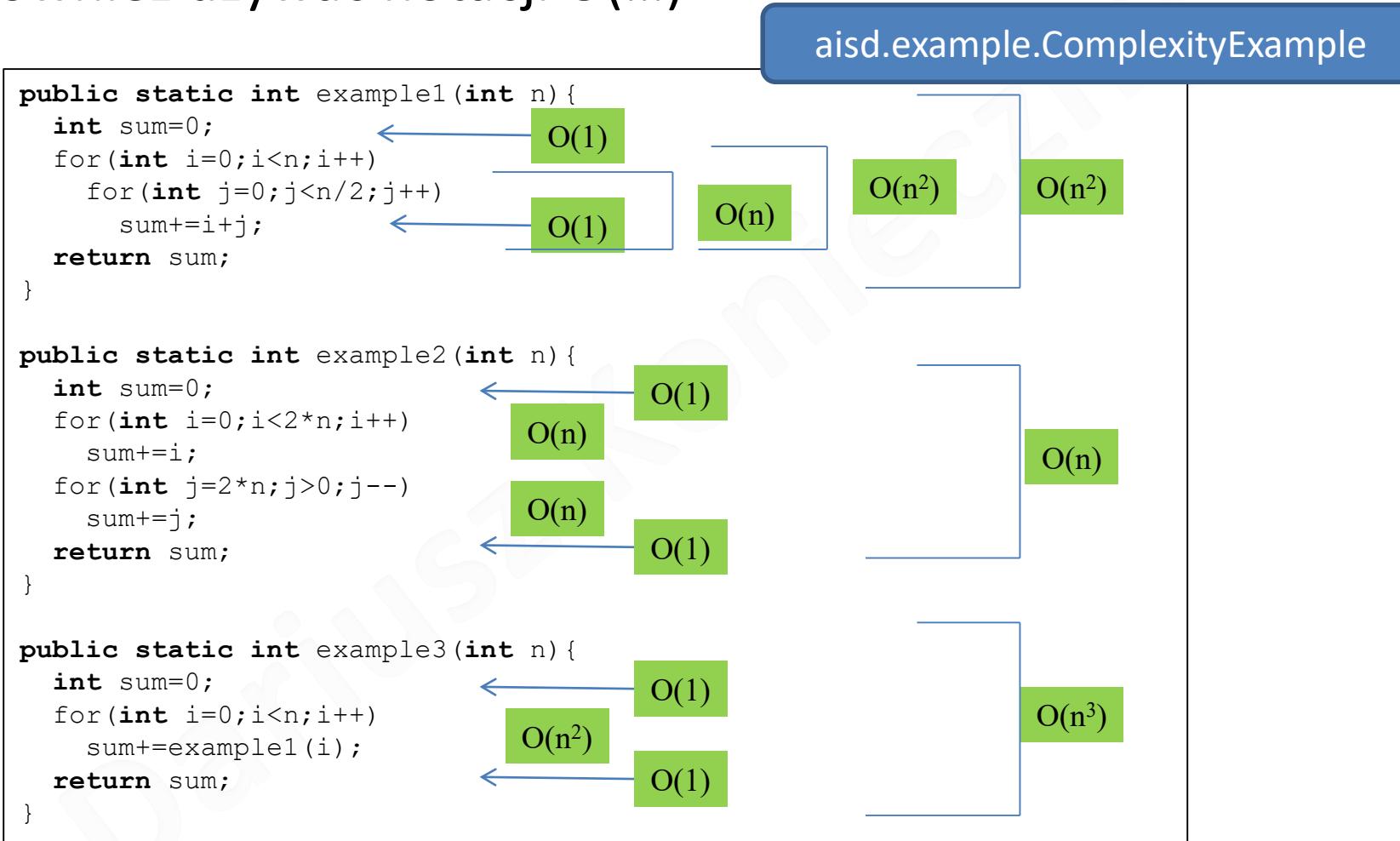
Mówimy o dolnym
ograniczeniu funkcji

Zależności - przykład



Przykłady liczenia złożoności

- W poniższych przykładach zamiast $O(\dots)$ można by również używać notacji $\Theta(\dots)$



Złożoność problemu/algorytmu

- **Złożoność czasowa** problemu to liczba kroków potrzebna do rozwiązania instancji problemu, jako funkcja rozmiaru danych wejściowych, używając najbardziej efektywnego algorytmu.
 - Nie zawsze dotychczas znany algorytm jest najlepszy, stąd dla problemu przedstawia się często dowód jaka musi być minimalna liczba kroków
- **Złożoność pamięciowa** problemu jest analogicznym pojęciem, które mierzy ilość pamięci potrzebnej przez algorytm
- **Złożoność czasowa** oraz **złożoność pamięciowa** może być rozważana w odniesieniu do **wybranego algorytmu**.

Lista

- Lista jest strukturą liniową, w której można wykonywać wiele operacji w dowolnym miejscu tej struktury.
- Lista oprócz zwykłego iteratora powinna też udostępniać iterator dla list, który umożliwia poruszanie się w dwóch kierunkach po liście.
- W pakiecie `java.lang` dostępny jest interfejs dla list `List<E>`. Aby nie komplikować kodu w ramach tego wykładu stworzony będzie podobny, uproszczony interfejs `IList<E>`.

aisd.list.IList

```
import java.util.Iterator;
import java.util.ListIterator;

public interface IList<E> extends Iterable<E> {
    boolean add(E e); // dodanie elementu na koniec listy
    void add(int index, E element); // dodanie elementu na podanej pozycji
    void clear(); // skasowanie wszystkich elementów
    boolean contains(E element); // czy lista zawiera podany element (equals())
    E get(int index); // pobranie elementu z podanej pozycji
    E set(int index, E element); // ustawienie nowej wartości na pozycji
    int indexOf(E element); // pozycja szukanego elementu (equals())
    boolean isEmpty(); // czy lista jest pusta
    Iterator<E> iterator(); // zwraca iterator przed pierwszą pozycją
    ListIterator<E> listIterator(); // j.w. dla ListIterator
    E remove(int index); // usuwa element z podanej pozycji
    boolean remove(Element e); // usuwa element (equals())
    int size(); // rozmiar listy
}
```

ListIterator<T>

- Interfejs do kolekcji liniowych, po których można się przemieszczać w dwóch kierunkach.
- Operacje, jakie można wykonywać za pomocą tego interfejsu podano poniżej.
- Jeśli jakaś operacja jest trudna/nieefektywna dla danej kolekcji, zamiast ją implementować można rzucać wyjątkiem `UnsupportedOperationException`. W dokumentacji są one zaznaczone jako opcjonalne.

java.util.ListIterator

```
public class ListIterator<T> implements Iterator<T> {  
    void add(E e); // dodanie e w bieżącej pozycji, ZA kurSOR  
    boolean hasNext();  
    boolean hasPrevious(); // jak hasNext, ale w przeciwnym kierunku  
    E next();  
    int nextIndex(); // indeks elementu, który byłby zwrocony przez next()  
    E previous(); jak next(), ale w przeciwnym kierunku  
    int previousIndex(); jak nextIndex(), ale w przeciwnym kierunku  
    void remove(); // usuwa ostatnio zwrocony element przez next() lub previous()  
    void set(E e); wstawia wartosc e do kolekcji pod ostatnio zwrocony element  
}
```

- Wspomniane wyżej podejście nie generuje niepotrzebnie całej rodziny iteratorów (np. iteratory bez operacji add/remove lub jednej z nich itd.)
- Oczywiście tworząc własną kolekcję i zwracając iterator dla niej należy udokumentować, jak się zachowują poszczególne operacje w iteratorze.
- Iteratory obecne w bibliotekach języka Java są przygotowane do programowania współbieżnego, więc ich implementacja jest bardziej złożona niż to co jest przedstawiane na tym kursie.

Iterator dla list

- Na tym wykładzie przedstawione będzie tylko szkielet jak wyglądają operacje dla tego iteratora w przypadku poprawnej sekwencji operacji
 - W przypadku niepoprawnej sekwencji (np. dwa razy `remove()` bez wykonania pomiędzy `next()` lub `previous()`) iterator może zachowywać się niepoprawnie (wg dokumentacji powinien rzucać właściwym wyjątkiem)
 - Uzupełnienie kodu iteratora, tak aby działał w każdym wypadku poprawnie jest ciekawym zadaniem do wykonania samodzielnie, jednak od strony algorytmiki nie dodaje istotnej wiedzy.
-
- Innym wyzwaniem jest używanie dwóch lub więcej iteratorów jednocześnie lub wykonywanie w czasie używania iteratora metod listy modyfikującej jej strukturę (dodawanie/usuwanie), np. iterator przesuwamy na środek listy i usuwamy listę przez metodę `clear()`. Jak powinien zachować się iterator po wywołaniu `next()`?
 - Implementacja biblioteczna generuje w takich przypadkach (i wielu innych) wyjątek `IllegalStateException`.
 - Obowiązuje ogólna zasada: jeśli jakiś obiekt zmodyfikował kolekcję, to wszystkie inne iteratory ulegają unieważnieniu (nie można już z nich korzystać).

Dokumentacja w kodzie Javy

- Java pozwala komentować kod tak, aby komentarz pozwalał na dokumentację techniczną.
- Większość środowisk deweloperskich odczytuje tak sformatowane komentarze w trakcie pracy nad kodem.
- Komentarz dokumentujący powinien zaczynać się od „/**” zamiast od „/*”
- Kolejne linii zaczynają się od „*”, ale nie jest to konieczne
- Komentarz taki pisze się PRZED deklaracją klasy, metody, pola.
- W tekście można używać część tagów HTML oraz innych specjalnych.

```
@Override
public ListIterator<E> listIterator() {
    return new InnerListIterator();
}

@ aisd.util.ArrayList.InnerListIterator

    iterator stoi "pomiędzy" elementami i tak trzeba go zaimplementować. Zaimplementowane zostaną tylko operacje
    niemodyfikujące strukturę
Press 'F2' for focus

/** iterator stoi "pomiędzy" elementami i tak trzeba go zaimplementować.
 * Zaimplementowane zostaną tylko operacje niemodyfikujące strukturę */
private class InnerListIterator implements ListIterator<E>{
    int _pos=0;
```

Klasa AbstractList<T> 1/2

- Implementacja pewnych metod (pochodzących z klasy Object) może wyglądać dla każdej listy tak samo, zatem warto stworzyć abstrakcyjną klasę, w której zostaną one zrealizowane (od Javy 9.0 można to zrobić w ramach interfejsu):

```
package aisd.util;  
  
import java.util.Iterator;  
  
public abstract class AbstractList<E> implements IList<E> {  
  
    @Override  
    public String toString() {  
        StringBuffer buffer = new StringBuffer();  
        buffer.append('[');  
        if (!isEmpty()) {  
            for (E item:this)  
                buffer.append(item).append(", ");  
            buffer.setLength(buffer.length() - 2);  
        }  
        buffer.append(']');  
        return buffer.toString();  
    }  
    // ^ - bitowa różnica symetryczna  
    @Override  
    public int hashCode() {  
        int hashCode = 0;  
        for (E item:this)  
            hashCode ^= item.hashCode();  
        return hashCode;  
    }  
}
```

aisd.list.AbstractList

Klasa AbstractList<T> 2/2

```
@SuppressWarnings("unchecked")
@Override
public boolean equals(Object object) {
    if(object==null)
        return false;
    if (getClass() != object.getClass())
        return false;
    return equals((IList<E>) object);
}
public boolean equals(IList<E> other) {
    if (other == null || size() != other.size())
        return false;
    else {
        Iterator<E> i = iterator();
        Iterator<E> j = other.iterator();
        boolean has1=i.hasNext(),has2=j.hasNext();
        for(;has1 && has2 && i.next().equals(j.next()));
        {
            has1=i.hasNext();
            has2=j.hasNext();
        }
    return !has1 && !has2;  }
}
```

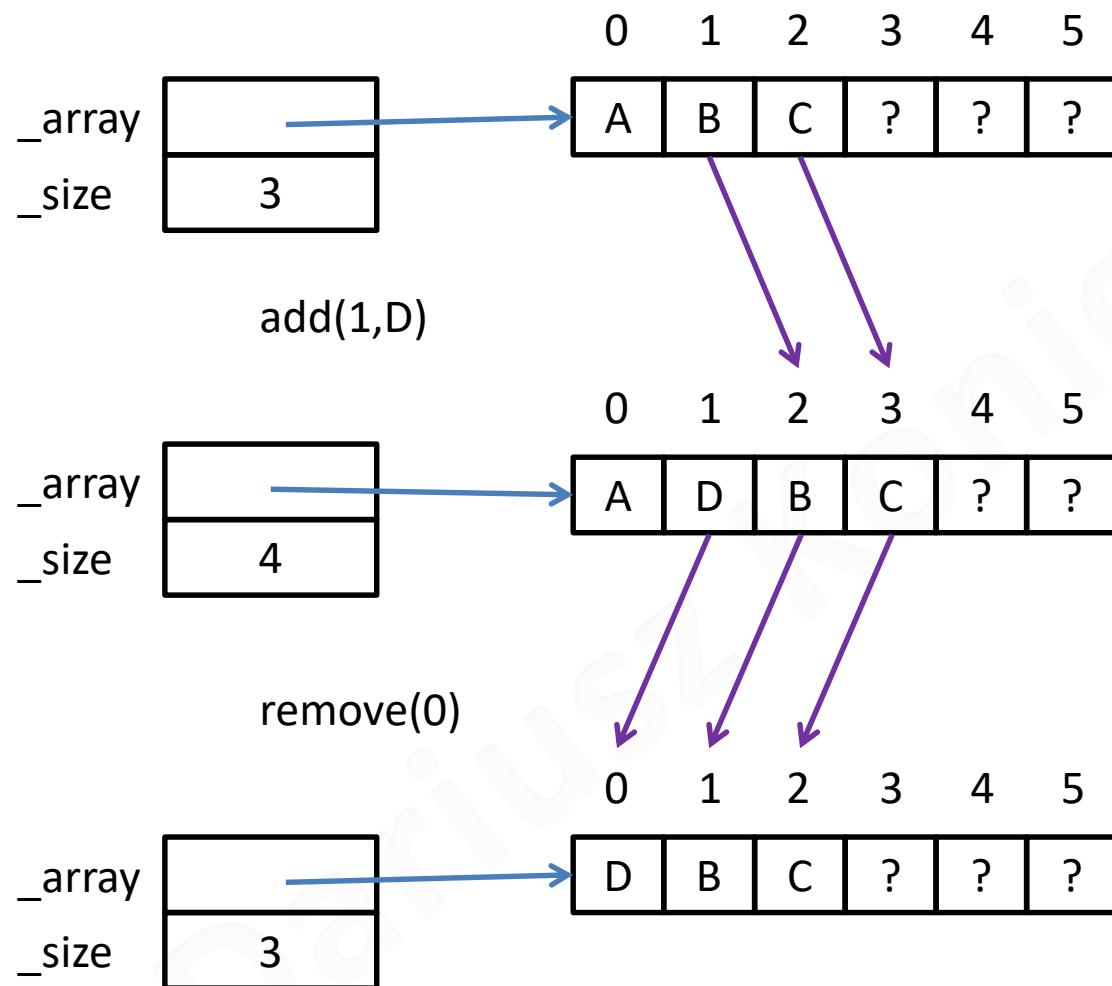
Implementacja listy na tablicy

- Lista, od strony użytkownika, ma najczęściej nieograniczoną pojemność.
 - Można ją zaimplementować za pomocą zwykłej tablicy, która się „rozszerza”, gdy brakuje miejsca na nowy element.
 - „Rozszerzanie” polega na stworzeniu nowej, większej tablicy i przepisaniu do niej danych z poprzedniej tablicy.
 - Implementacja listy będzie za pomocą klasy generycznej
-
- W Javie nie można stworzyć tablicy elementów generycznych (dokładnie: tablicy elementów typu będącego parametrem klasy generycznej)
 - Zamiast tego tworzona będzie tablica typu Object i rzutowana na tablicę elementów generycznych. Jest to dozwolone, ale kompilator ostrzega o możliwości niedopasowania typów, stąd przed funkcją z takim rzutowaniem należy dodać annotację:
 `//@SuppressWarnings ("unchecked")`
 - Np.

```
public class ArrayList<E> extends AbstractList<E> {  
    ...  
    private E[] _array;  
    ...  
    @SuppressWarnings ("unchecked")  
    public ArrayList(int capacity) {  
        ...  
        _array=(E[]) (new Object[capacity]);  
        ...  
    }
```

aisd.list.ArrayList

Lista na tablicy



ArrayList 1/4

```
package aisd.util;
import java.util.Iterator;
import java.util.ListIterator;

public class ArrayList<E> extends AbstractList<E> {

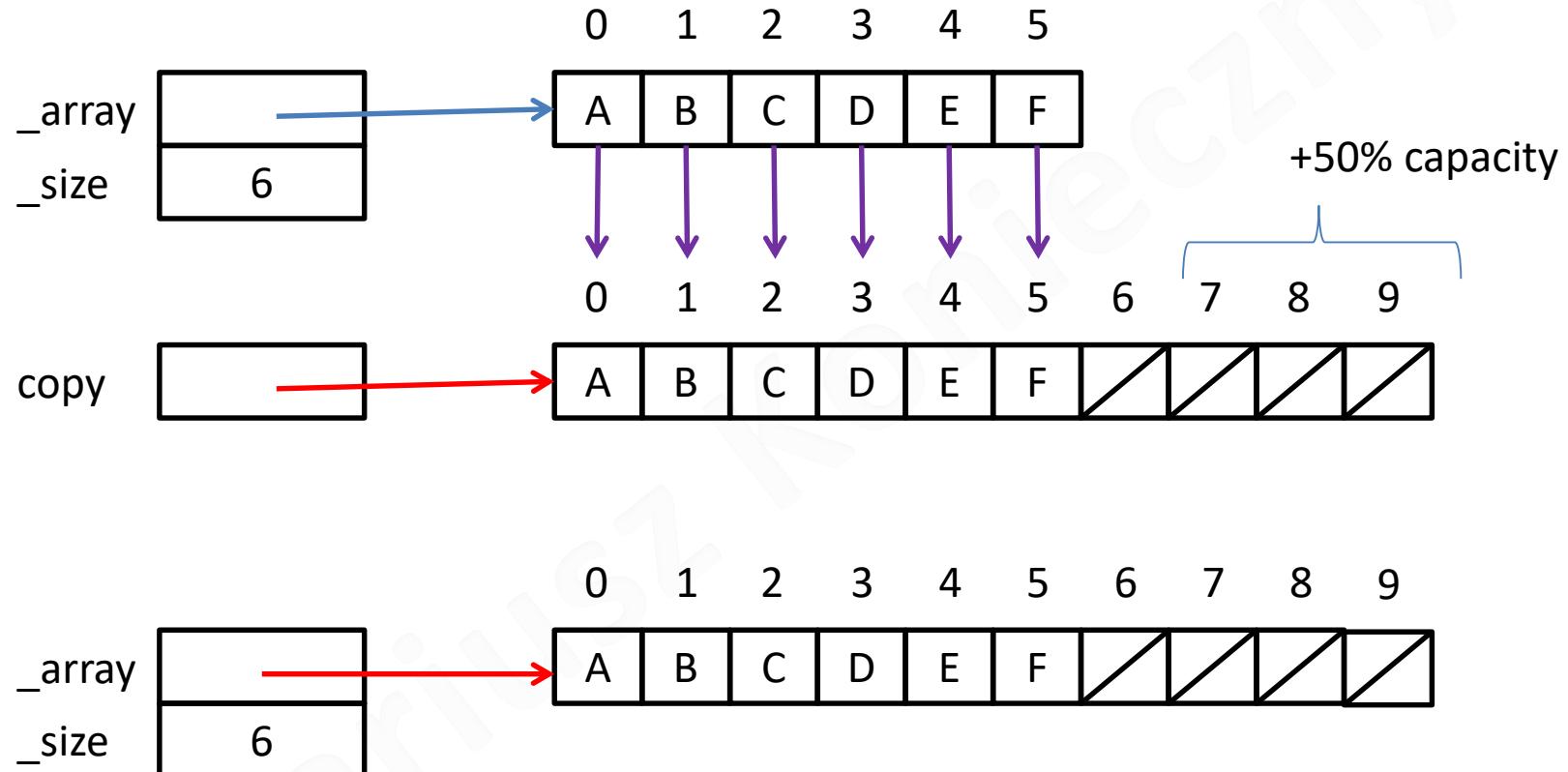
    /** <b> domyślna </b> wielkość początkowa tablicy */
    private static final int DEFAULT_INITIAL_CAPACITY = 16;
    /** <b> Początkowa </b> wielkość tablicy. */
    private final int _initialCapacity;
    /** referencja na tablicę zawierającą elementy */
    private E[] _array;
    /** rozmiar tablicy traktowanej jako lista */
    private int _size;

    // @SuppressWarnings("unchecked")
    public ArrayList(int capacity) {
        if(capacity<=0)
            capacity=DEFAULT_INITIAL_CAPACITY;
        _initialCapacity=capacity;
        _array=(E[]) (new Object[capacity]);
        _size=0;
    }

    public ArrayList(){
        this(DEFAULT_INITIAL_CAPACITY);
    }
    @Override
    public boolean isEmpty() {
        return _size==0;
    }
    @Override
    public int size() {
        return _size;
    }
}
```

ensureCapacity

ensureCapacity(7)



ArrayList 2/4

```
/** rozszerzenie tablicy jeśli za mało miejsca w obecnej */
@SuppressWarnings("unchecked")
private void ensureCapacity(int capacity) {
    if (_array.length < capacity) {
        E[] copy = (E[]) (new Object[capacity + capacity / 2]);
        System.arraycopy(_array, 0, copy, 0, _size);
        _array = copy;
    }
    // sprawdzenie poprawności indeksu
private void checkOutOfBounds(int index) throws IndexOutOfBoundsException {
    if(index<0 || index>=_size) throw new IndexOutOfBoundsException();
}
@SuppressWarnings("unchecked")
@Override
public void clear() {
    _array=(E[]) (new Object[_initialCapacity]);
    _size=0;
}
@Override
public boolean add(E value) {
    ensureCapacity(_size+1);
    _array[_size]=value;
    _size++;
    return true;
}
@Override
public boolean add(int index, E value) {
    if(index<0 || index>_size) throw new IndexOutOfBoundsException();
    ensureCapacity(_size+1);
    if(index!=_size)
        System.arraycopy(_array, index, _array, index+1, _size - index);
    _array[index]=value;
    _size++;
    return false;
}
```

ArrayList 3/4

```
@Override
public int indexOf(E value) {
    int i = 0;
    while(i < _size && !value.equals(_array[i]))    ++i;
    return i<_size ? i : -1;
}

@Override
public boolean contains(E value) {
    return indexOf(value) != -1;
}

@Override
public E get(int index) {
    checkOutOfBounds(index);
    return _array[index];
}

@Override
public E set(int index, E element) {
    checkOutOfBounds(index);
    E retValue=_array[index];
    _array[index]=element;
    return retValue;
}

@Override
public E remove(int index) {
    checkOutOfBounds(index);
    E retValue = _array[index];
    int copyFrom = index + 1;
    if (copyFrom < _size) System.arraycopy(_array, copyFrom, _array, index, _size - copyFrom);
    --_size;
    return retValue;
}

@Override
public boolean remove(E value) {
    int pos=0;
    while(pos<_size && !_array[pos].equals(value))
        pos++;
    if(pos<_size){
        remove(pos);
        return true;
    }
    return false;
}
```

ArrayList.InnerIterator 1/1

```
@Override
public Iterator<E> iterator() {
    return new InnerIterator();
}

@Override
public ListIterator<E> listIterator() {
    return new InnerListIterator();
}

private class InnerIterator implements Iterator<E>{
    int _pos=0;
    @Override
    public boolean hasNext() {
        return _pos<_size;
    }

    @Override
    public E next() {
        return _array[_pos++];
    }
}
```

ArrayList.InnerListIterator 1/1

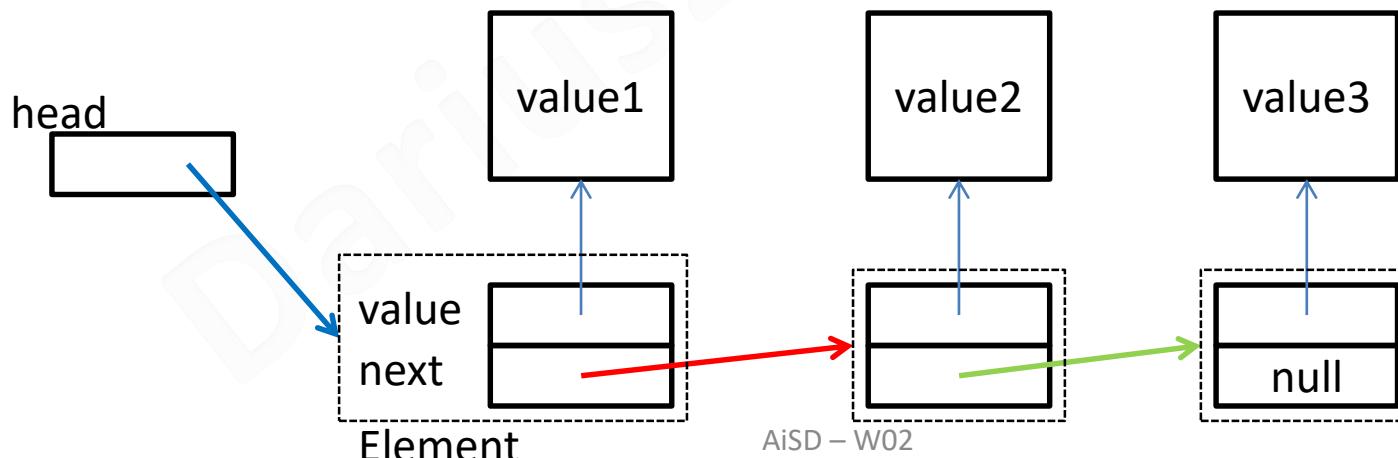
```
private class InnerListIterator implements ListIterator<E>{
    int _pos=0;
    @Override
    public void add(E Value) {
        throw new UnsupportedOperationException();
    }
    @Override
    public boolean hasNext() {
        return _pos<_size;
    }
    @Override
    public boolean hasPrevious() {
        return _pos>=0;
    }
    @Override
    public E next() {
        return _array[_pos++];
    }
    @Override
    public int nextIndex() {
        return _pos;
    }
    @Override
    public E previous() {
        return _array[--_pos];
    }
    @Override
    public int previousIndex() {
        return _pos-1;
    }
    @Override
    public void remove() {
        throw new UnsupportedOperationException();
    }
    @Override
    public void set(E e) {
        throw new UnsupportedOperationException();
    }
}
```

ArrayList, InnerListIterator - złożoności

- **ArrayList – złożoność pesymistyczna:**
 - Konstrukcja – $O(1)$
 - isEmpty(), size(), clear() – $O(1)$
 - set(), get() – $O(1)$
 - ensureCapacity() – $O(n)$
 - add() na końcu – $O(n)$
 - add() z indeksem – $O(n)$, na końcu – $O(1)$ bez ensureCapacity()
 - indexOf() – $O(n)$
 - contains() – $O(n)$
 - remove() x 2 – $O(n)$
- **ArrayList – złożoność średnia:**
 - ensureCapacity() – $O(1)$
 - add() na końcu – $O(1)$
- **ArrayList.InnerListIterator, gdyby zaimplementować wszystkie operacje – złożoność średnia:**
 - hasNext(), next(), hasPrevious(), previous(), nextIndex(), previousIndex() – $O(1)$
 - set() – $O(1)$
 - add(), remove() – $O(n)$

Listy za pomocą list wiązanych

- Gdyby była potrzeba usunięcia np. co drugiego elementu na liście lub inne podobne działania (za pomocą właśnie zaimplementowanych metod), złożoność takiej operacji byłaby kwadratowa $O(n^2)$.
- Nawet gdyby zaimplementować wszystkie operacje iteratora dla list.
- Dodatkowo w `ArrayList` zdarza się, że połowa tablicy jest nieużywana.
- Jeśli w danej kolekcji będzie potrzeba częstego wstawiania/usuwania elementów w środku w dodatku poruszając się iteratorem, istnieje szybsza implementacja listy za pomocą listy elementów wiązanych.
- Najprostszą do zrozumienia jest lista jednokierunkowa prosta z głową (L1KPzG).
 - Każdy element listy ma dwa pola: wartość oraz referencję na kolejny element. Jeśli tego elementu nie ma, to pole ma wartość **null**.
 - Cała lista jest pamiętana poprzez pole `head` (głowa), które jest referencją na pierwszy element listy.
- Ponieważ będzie to klasa generyczna, zatem wartość będzie również referencją (na jakiś typ `E`)



OneWayLinkedListWithHead.Element

```
public class OneWayLinkedListWithHead<E> extends AbstractList<E>{
    private class Element{
        private E value;
        private Element next;

        public E getValue() {
            return value;
        }

        public void setValue(E value) {
            this.value = value;
        }

        public Element getNext() {
            return next;
        }

        public void setNext(Element next) {
            this.next = next;
        }

        Element(E data){
            this.value=data;
        }
    }

    Element head=null;

    public OneWayLinkedListWithHead() {}

    public boolean isEmpty(){
        return head==null;
    }

    @Override
    public void clear() {
        head=null;
    }
}
```

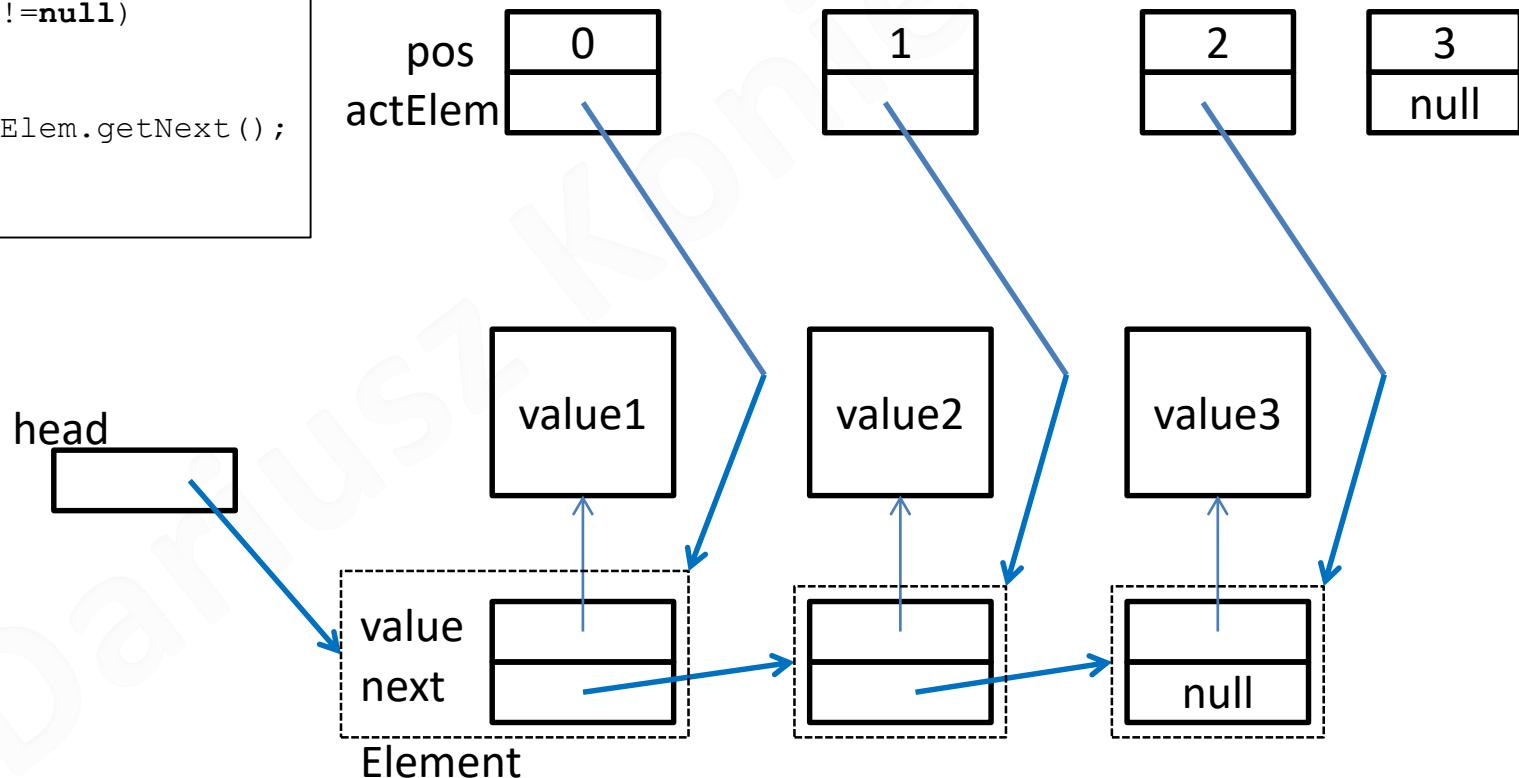
aisd.list. OneWayLinkedListWithHead

head
null

Metoda size(), przehodzenie po liście

- Przechodzenie po liście powiązanej do przodu za pomocą pomocniczej referencji:
actElem=actElem.getNext();

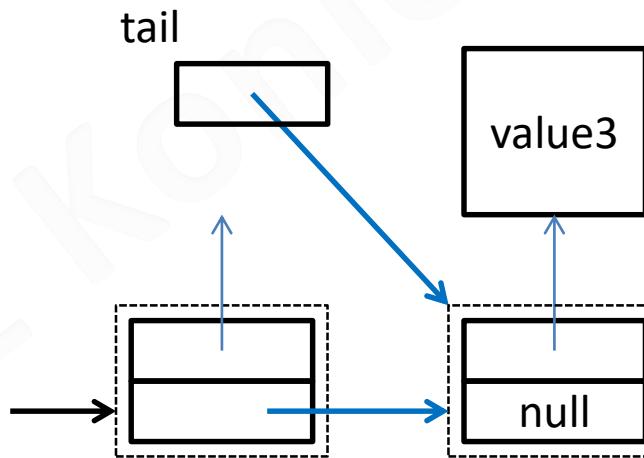
```
@Override  
public int size() {  
    int pos=0;  
    Element actElem=head;  
    while(actElem!=null)  
    {  
        pos++;  
        actElem=actElem.getNext();  
    }  
    return pos; }
```



Metody getElement(), add()

```
/** zwraca referencję na Element, wewnętrzną klasę */
private Element getElement(int index) {
    if(index<0) throw new IndexOutOfBoundsException();
    Element actElem=head;
    while(index>0 && actElem!=null) {
        index--;
        actElem=actElem.getNext();
    }
    if (actElem==null)
        throw new IndexOutOfBoundsException();
    return actElem;
}

@Override
public boolean add(E e) {
    Element newElem=new Element(e);
    if(head==null) {
        head=newElem;
        return true;
    }
    Element tail=head;
    while(tail.getNext()!=null)
        tail=tail.getNext();
    tail.setNext(newElem);
    return true;
}
```



- Dodając element na koniec tej listy trzeba osobno rozpatrzyć dodawanie do pustej listy i osobno do niepustej.

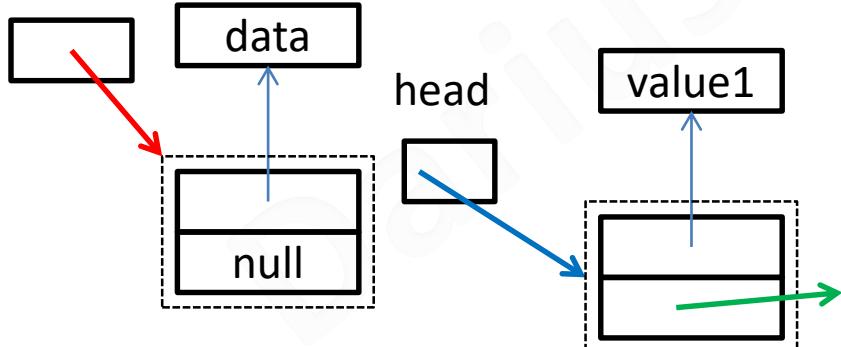
Metoda add () na pozycji

```
@Override  
public boolean add(int index, E data) {  
    if(index<0) throw new IndexOutOfBoundsException();  
    Element newElem=new Element(data);  
    if(index==0)  
    {  
        newElem.setNext(head);  
        head=newElem;  
        return true;  
    }  
    Element actElem=getElement(index-1);  
    newElem.setNext(actElem.getNext());  
    actElem.setNext(newElem);  
    return true;}  
}
```

- Osobno trzeba rozpatrzyć dodawanie na pozycję 0
- W p.p. trzeba znaleźć element z pozycji (index-1) : wytłumaczenie na kolejnym slajdzie

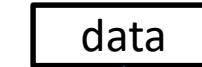
Dla indeksu 0:

newElem

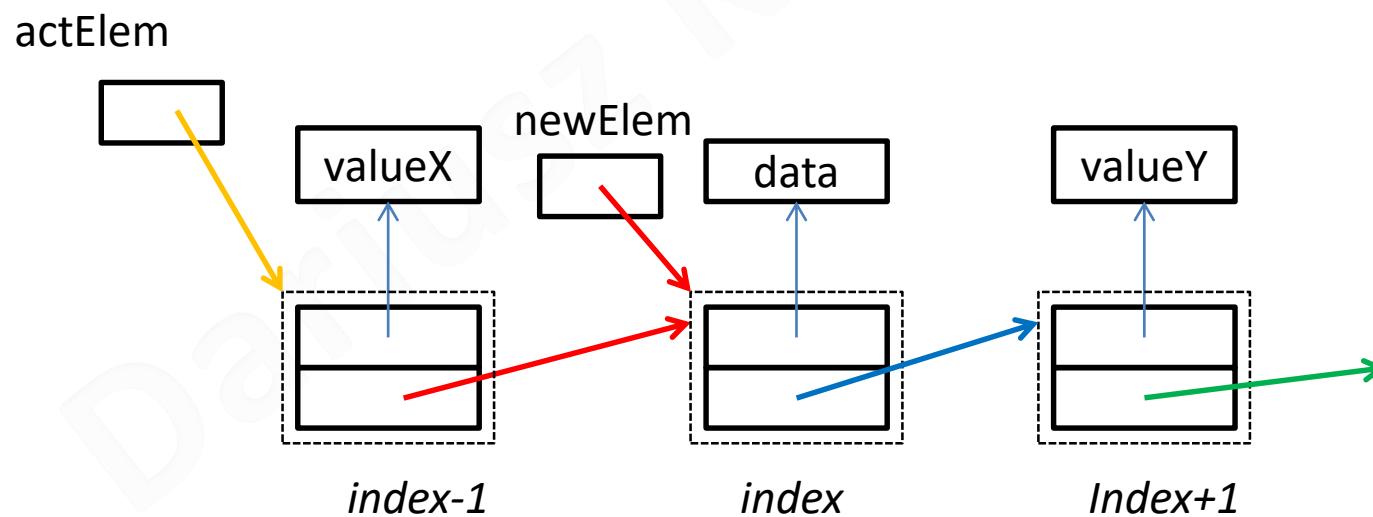
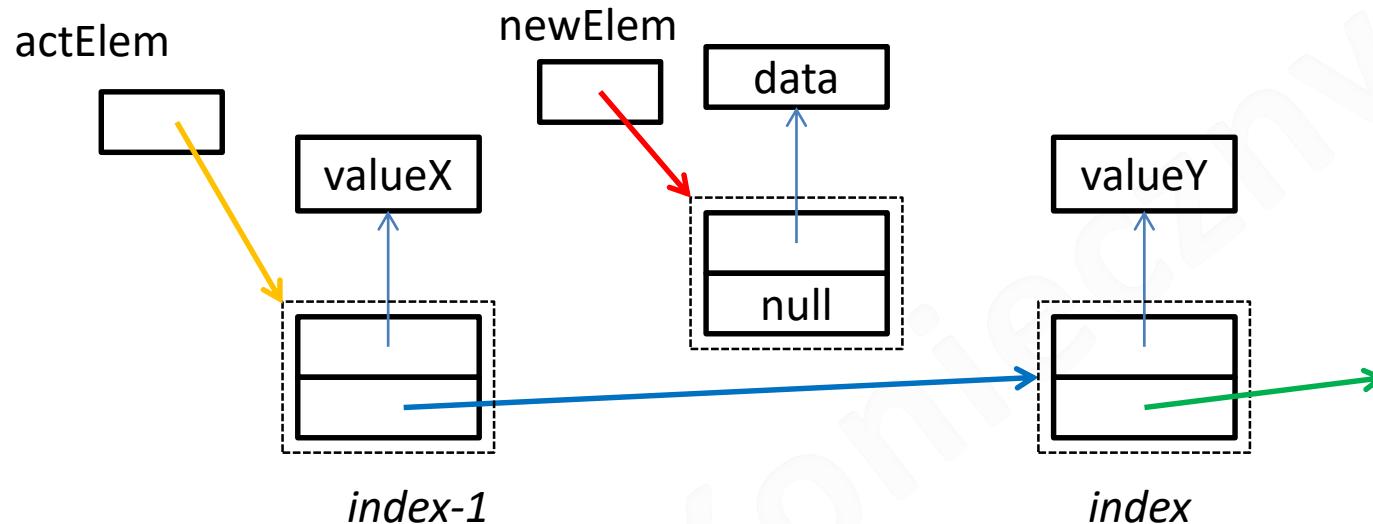


head

newElem



add() w środku listy



indexOf(), contains(), get(), set()

```
@Override
public int indexOf(E data) {
    int pos=0;
    Element actElem=head;
    while(actElem!=null)
    {
        if(actElem.getValue().equals(data))
            return pos;
        pos++;
        actElem=actElem.getNext();
    }
    return -1;
}

@Override
public boolean contains(E data) {
    return indexOf(data)>=0;

}

@Override
public E get(int index) {
    Element actElem=getElement(index);
    return actElem.getValue();

}

@Override
public E set(int index, E data) {
    Element actElem=getElement(index);
    E elemData=actElem.getValue();
    actElem.setValue(data);
    return elemData;
}
```

remove() x 2

- Usuwając element z podanego indeksu, podobnie jak w dodawaniu, trzeba się zatrzymać jeden element wcześniej i wykonać zmiany odwrotnie do operacji add().

```
@Override
public E remove(int index) {
    if(index<0 || head==null) throw new IndexOutOfBoundsException();
    if(index==0) {
        E retValue=head.getValue();
        head=head.getNext();
        return retValue;
    }
    Element actElem=getElement(index-1);
    if(actElem.getNext()==null)
        throw new IndexOutOfBoundsException();
    E retValue=actElem.getNext().getValue();
    actElem.setNext(actElem.getNext().getNext());
    return retValue;
}

@Override
public boolean remove(E value) {
    if(head==null)
        return false;
    if(head.getValue().equals(value)) {
        head=head.getNext();
        return true;
    }
    Element actElem=head;
    while(actElem.getNext()!=null && !actElem.getNext().getValue().equals(value))
        actElem=actElem.getNext();
    if(actElem.getNext()==null)
        return false;
    actElem.setNext(actElem.getNext().getNext());
    return true;
}
```

Iterator

```
private class InnerIterator implements Iterator<E>{
    Element actElem;
    public InnerIterator() {
        actElem=head;
    }
    @Override
    public boolean hasNext() {
        return actElem!=null;
    }
    @Override
    public E next() {
        E value=actElem.getValue();
        actElem=actElem.getNext();
        return value;
    }
}
@Override
public Iterator<E> iterator() {
    return new InnerIterator();
}
```

Lista L1KPzG - złożoność

- Lista L1KPzG – złożoność pesymistyczna:
 - Konstrukcja – $O(1)$
 - `isEmpty()`, `clear()` – $O(1)$
 - `set()`, `get()`, `size()` – $O(n)$
 - `add()` na końcu – $O(n)$
 - `add()` na początku – $O(1)$
 - `add()` z indeksem – $O(n)$
 - `indexOf()` – $O(n)$
 - `contains()` – $O(n)$
 - `remove()` x 2 – $O(n)$
 - `remove()` na początku – $O(1)$
- Iterator dla listy L1KPzG :
 - `hasNext()`, `next()` – $O(1)$
- Lista L1KPzG jest dość ograniczona w swoich zastosowaniach, pozwala jednak na efektywną implementację stosu

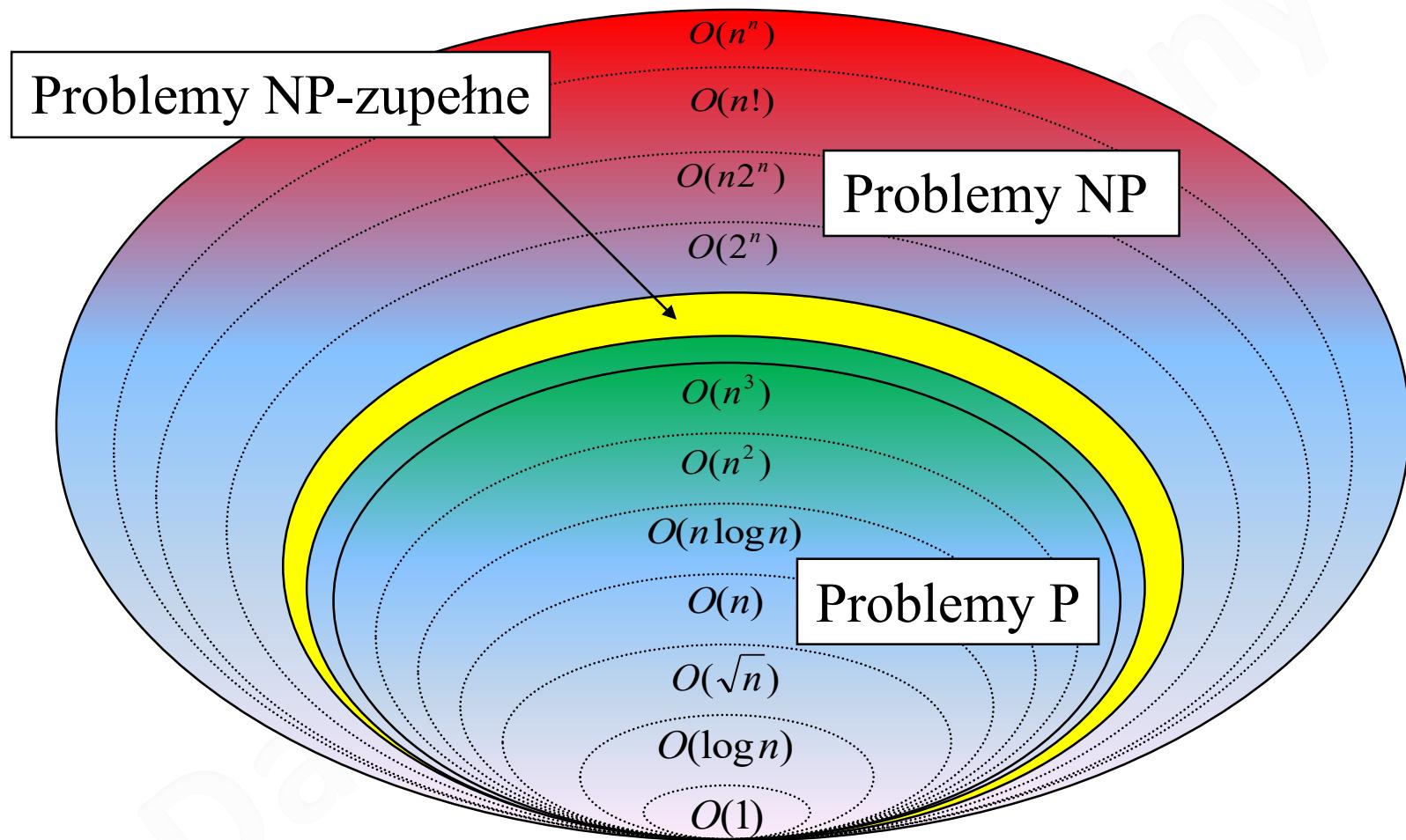
Algorytmy i struktury danych – W03

Teoria złożoności cz. 3/4,
stos, kolejka

Zawartość

- Złożoność cz. 3:
 - Hierarchia złożoności
 - Porównanie złożoności
- Lista dwukierunkowa, cykliczna, ze strażnikiem – L2KCzS, klasa TwoWayCycledListWithSentinel<E>
- Typy list
- Lista LinkedList w Javie
- Stos:
 - Opis
 - Implementacja za pomocą tablicy
 - Implementacja za pomocą listy
- Kolejka:
 - Opis
 - Implementacja za pomocą tablicy z jednym wolnym miejscem
 - Implementacja za pomocą listy

Hierarchia rzędów – klasy złożoności



Właściwości 1/2

- **Przechodniość**

$f(n) = \Theta(g(n))$ i $g(n) = \Theta(h(n))$ implikuje $f(n) = \Theta(h(n))$

$f(n) = O(g(n))$ i $g(n) = O(h(n))$ implikuje $f(n) = O(h(n))$

$f(n) = \Omega(g(n))$ i $g(n) = \Omega(h(n))$ implikuje $f(n) = \Omega(h(n))$

- **Zwrotność**

$f(n) = \Theta(f(n))$

$f(n) = O(f(n))$

$f(n) = \Omega(f(n))$

- **Symetria**

$f(n) = \Theta(g(n))$ wtw $g(n) = \Theta(f(n))$

- **Symetria przechodnia**

$f(n) = O(g(n))$ wtw $g(n) = \Omega(f(n))$

Właściwości 2/2

- Inne właściwości

$$n^m = O(n^k), \quad \text{gdzie } m \leq k$$

$$O(f(n)) + O(g(n)) = O(|f(n)| + |g(n)|)$$

$$c \cdot O(f(n)) = O(f(n)), \quad \text{gdzie } c \text{ jest stałe}$$

$$O(O(f(n))) = O(f(n))$$

$$O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$$

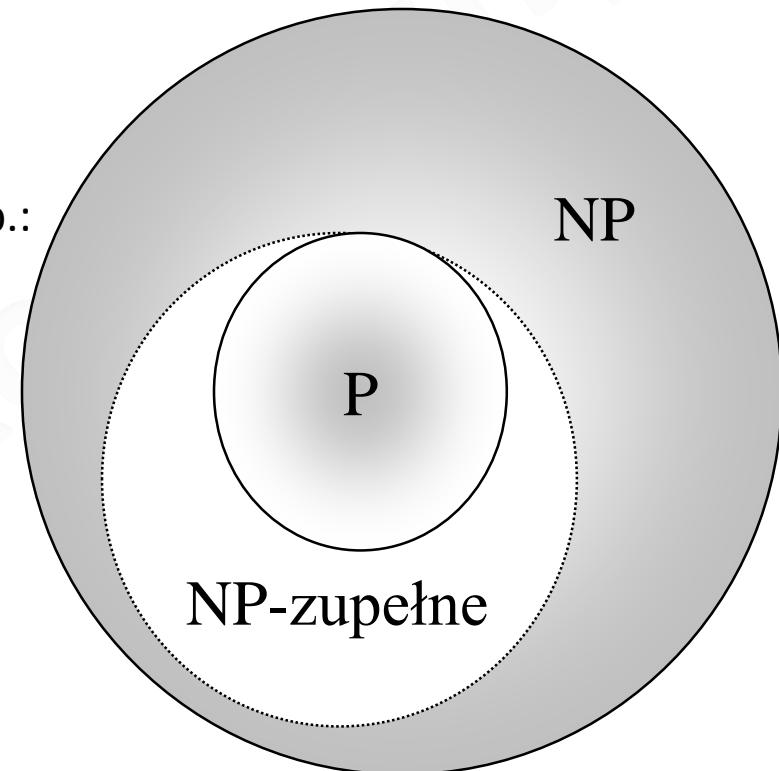
$$O(f(n) \cdot g(n)) = f(n) \cdot O(g(n))$$

- Zawsze należy pamiętać, że $O(\dots)$, $\Theta(\dots)$, $\Omega(\dots)$ oznacza zbiory funkcji i znak równości jest wieloznaczny:

- Oznacza równość zbiorów, gdy po obu stronach znaku równości są zbiory
- Oznacza należenie funkcji do zbioru, gdy jedna ze stron jest funkcją

Klasyfikacja problemów

- Problemy P:
 - Złożoność obliczeniowa co najwyżej wielomianowa np.:
 - Sortowania
 - Mnożenie macierzy
 - Szukanie najkrótszej ścieżki w grafie
- Problemy NP:
 - Złożoność co najmniej wykładnicza np.:
 - Metoda simplex
 - Problem małpiej układanki
- Problemy NP-zupełne
 - Złożoność nieznana, między P a NP
 - Problem spełnialności
 - Cykl Hamiltona
 - Podział zbioru



Porównanie złożoności 1/3

- Założenie: komputer jest w stanie wykonać miliard (10^9) kroków algorytmu w ciągu jednej sekundy. Czyli 1000 MHz kroków (a nie taktów zegara procesora) – nieosiągalne obecnie dla domowych komputerów

Funkcja złożoności	Rozmiar n					
	10	20	30	40	50	60
n	0,00000001s	0,00000002s	0,00000003s	0,00000004s	0,00000005s	0,00000006s
n^2	0,0000001s	0,0000004s	0,0000009s	0,0000016s	0,0000025s	0,0000036s
n^5	0,0001s	0,0032s	0,0243s	0,1024s	0,3125s	0,7776s
n^{10}	10s	2,8h	6,8 dni	121,4 dni	3,1 lat	19,2 lat
2^n	0,0000001s	0,001s	1s	18,3 min	13 dni	36,6 lat
3^n	0,000006s	3,5s	2,4 dni	385 lat	$2 \cdot 10^7$ lat	$1,3 \cdot 10^{12}$ lat
$n!$	0,003s	77 lat	$8 \cdot 10^{15}$ lat	$2 \cdot 10^{32}$ lat	$9 \cdot 10^{47}$ lat	$2 \cdot 10^{65}$ lat

Wiek wszechświata = $13,82 \cdot 10^9$ lat

Porównanie złożoności 2/3

- W ramach tego kursu złożoności obliczeniowe algorytmów nie przekroczą $O(n^4)$.
- Warto sprawdzić, co daje zmniejszenie wykładnika n o jeden, czyli polepszenie algorytmu o rzęd wielkości (tabelka poniżej).
- W API do różnych klas/interfejsów znajdują się informacje o złożoności danej metody lub oczekiwanej złożoności.

Funkcja złożoności	Rozmiar n					
	10	100	1000	10000	100000	1000000
n	0,00000001s	0,0000001s	0,000001s	0,00001s	0,0001s	0,001s
$n \log_2 n$	0,00000003s	0,0000006s	0,00001s	0,0001s	0,0016s	0,02s
n^2	0,0000001s	0,00001s	0,001s	0,1s	10s	16,6min
n^3	0,000001s	0,001s	1s	16,6min	11,5 dni	31,7 lat

Złożoność	Nazwa
$O(1)$	Stała
$O(\log_2 n)$	Logarytmiczna
$O(n)$	Liniowa
$O(n \log_2 n)$	Liniowo-logarytmiczna
$O(n^2)$	Kwadratowa
$O(n^3)$	Sześcienna

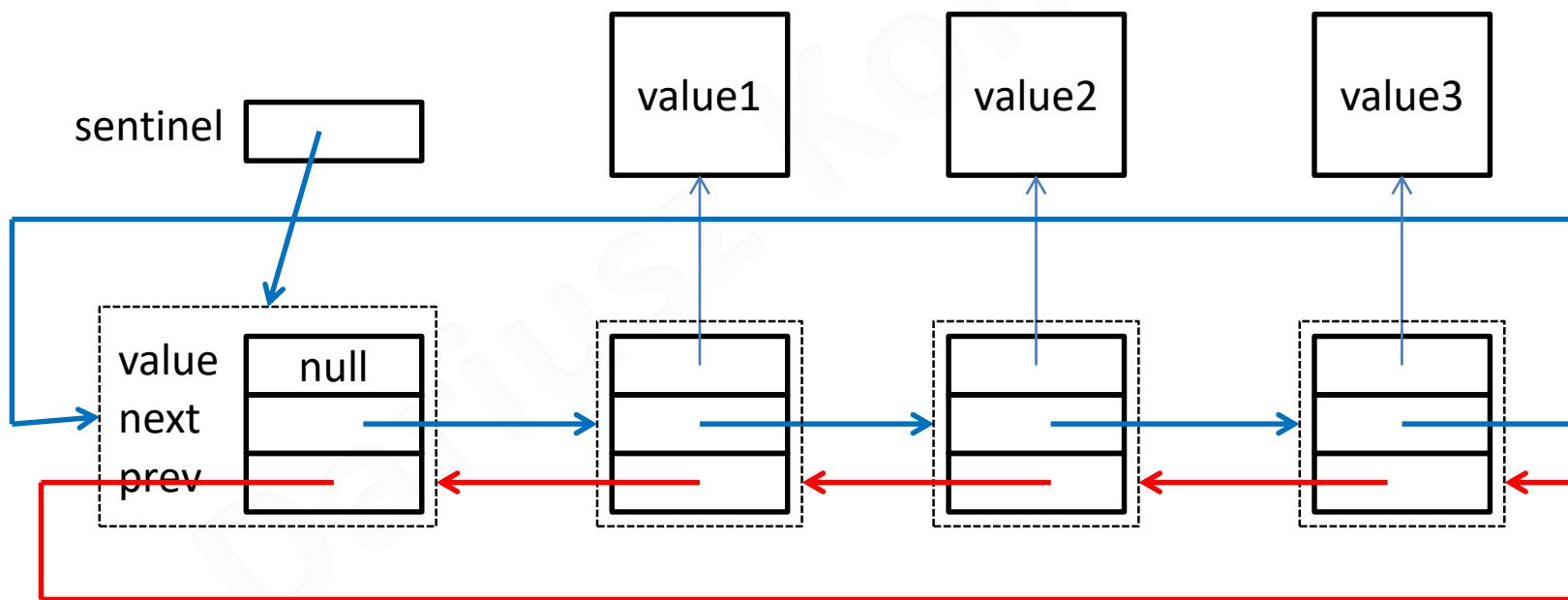
Porównanie złożoności 3/3

- Szybszy sprzęt obliczeniowy jest ważny, ale nie rozwiąże problemów z nieefektywnym algorytmem lub złożonym obliczeniowo problemem.
- Poniżej „analiza”, o ile zwiększylibyśmy rozmiar wejściowego problemu, gdyby udało się odpowiednio przyspieszyć komputer. Jaki największy problem możemy wtedy rozwiązać w ciągu jednej godziny?

Funkcja złożoności	Rozmiar największego problemu rozwiązywanego w ciągu 1h		
	Przez obecny komputer	Przez komputer 100 razy szybszy	Przez komputer 1000 razy szybszy
n	N_1 (10^{12})	$100N_1$ (10^{14})	$1000N_1$ (10^{15})
n^2	N_2 (10^6)	$10N_2$ (10^7)	$31,6N_2$ $(3 \cdot 10^7)$
n^3	N_3 $(10\ 000)$	$4,64N_3$ $(46\ 400)$	$10N_3$ $(100\ 000)$
n^5	N_4 (251)	$2,5N_4$ (631)	$3,98N_4$ (1000)
2^n	N_5 (40)	$N_5+6,64$ (47)	$N_5+9,97$ (50)
3^n	N_6 (25)	$N_6+4,19$ (29)	$N_6+6,29$ (31)

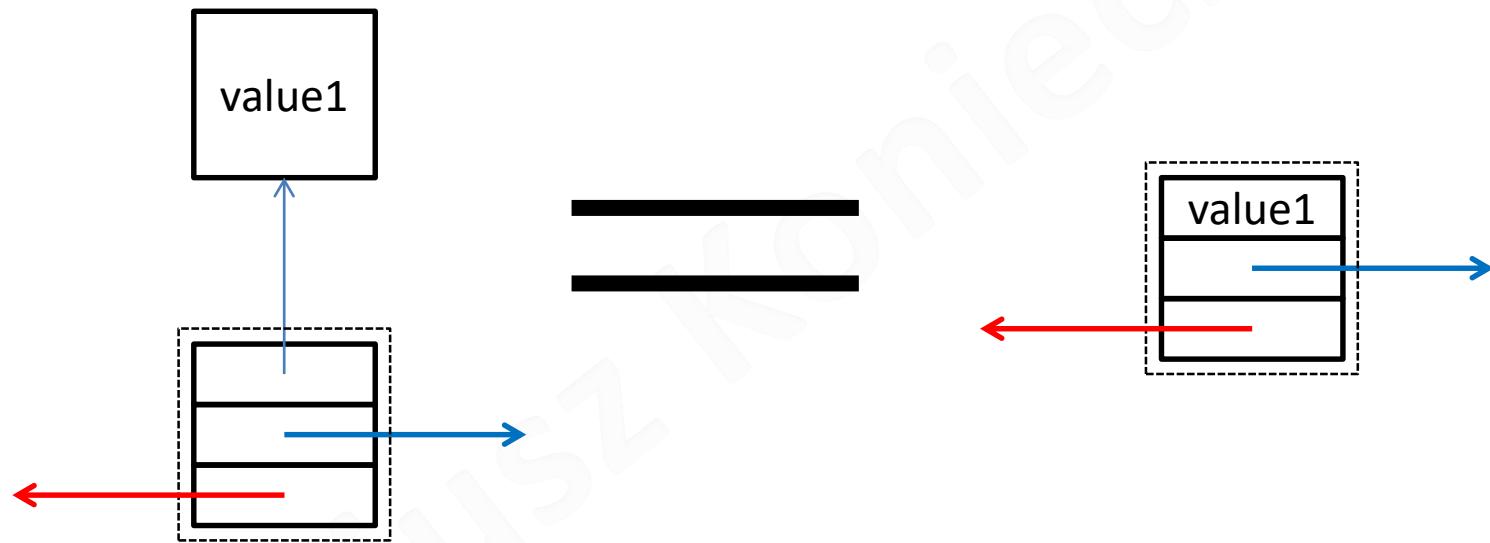
Lista L2KCzS

- Lista dwukierunkowa ma elementy, które mają dowiązanie do elementu poprzedniego i następnego w liście.
- Lista cykliczna zamiast wartości `null` w dowiązaniach w pierwszym i ostatnim elemencie ma dowiązania pomiędzy tymi elementami.
- Lista ze strażnikiem ma jeden element, który nie zawiera danych użytkownika listy ale służy uproszczeniu implementacji, gdyż zawsze w liście jest strażnik (nawet w pustej liście). Najczęściej w miejscu na referencję na dane jest wartość `null`.
- Lista dwukierunkowa cykliczna ze strażnikiem posiada wszystkie powyżej wymienione cechy.



Uproszczenie graficzne

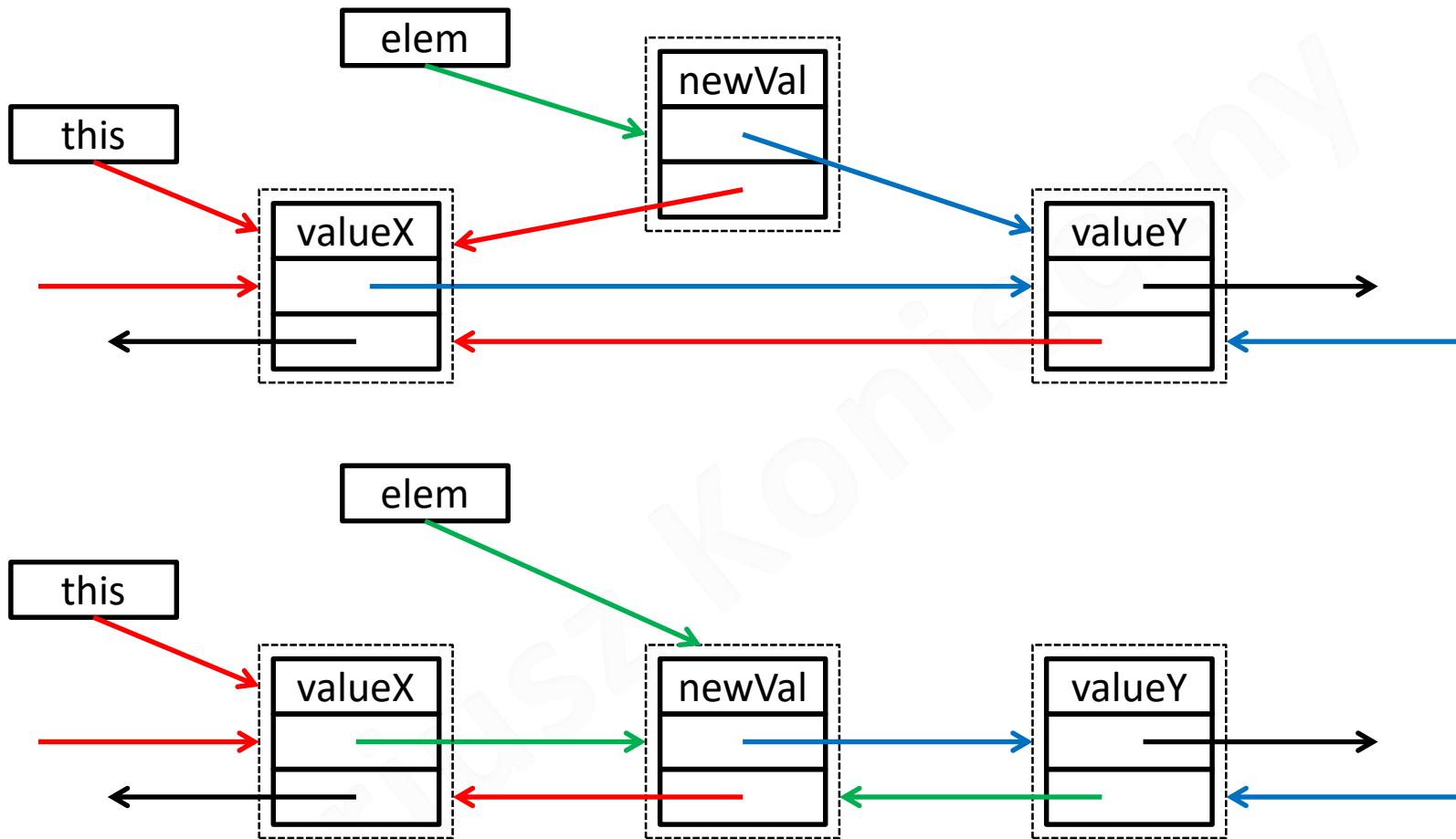
- Dla uproszczenia rysunków zamiast referencji na wartość rysowana będzie odpowiednia wartość w prostokącie.



Lista L2KCzS - Element

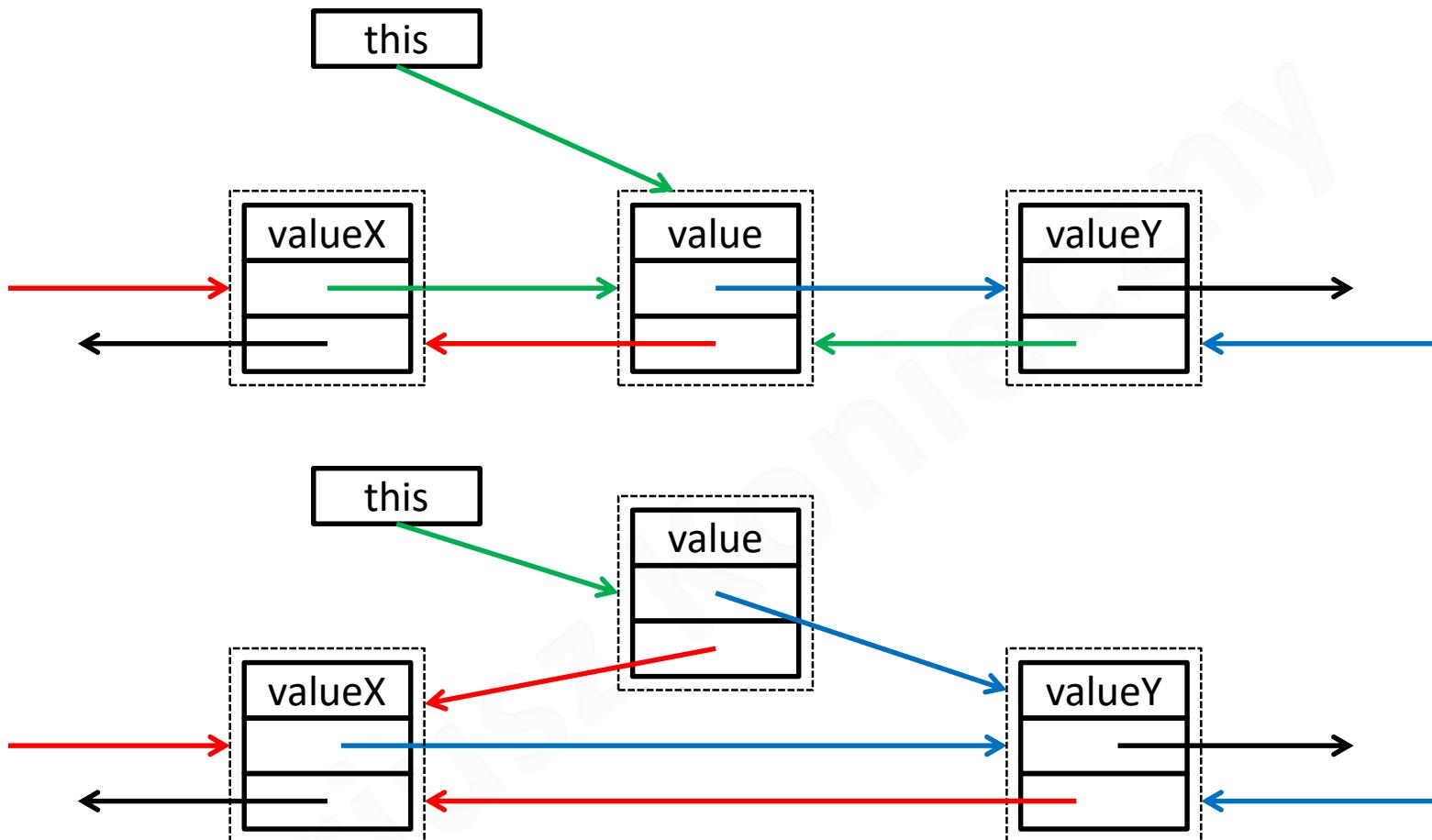
```
public class TwoWayCycledListWithSentinel<E> extends AbstractList<E> {  
    private class Element{  
        private E value;  
        private Element next;  
        private Element prev;  
  
        public E getValue() { return value; }  
        public void setValue(E value) { this.value = value; }  
        public Element getNext() {return next;}  
        public void setNext(Element next) {this.next = next;}  
        public Element getPrev() {return prev;}  
        public void setPrev(Element prev) {this.prev = prev;}  
        Element(E data){this.value=data;}  
        /** elem będzie stawiony <b> za this </b> */  
        public void insertAfter(Element elem){  
            elem.setNext(this.getNext());  
            elem.setPrev(this);  
            this.getNext().setPrev(elem);  
            this.setNext(elem);}  
        /** elem będzie stawiany <b> przed this </b> */  
        public void insertBefore(Element elem){  
            elem.setNext(this);  
            elem.setPrev(this.getPrev());  
            this.getPrev().setNext(elem);  
            this.setPrev(elem);}  
        /** elem będzie usuwany z listy w której jest <p>  
        * <b>Założenie:</b> element jest już umieszczony w liście i nie jest to sentinel */  
        public void remove(){  
            this.getNext().setPrev(this.getPrev());  
            this.getPrev().setNext(this.getNext());}}}
```

insertAfter ()



- Analogicznie przebiega wykonanie metody `insertBefore()`

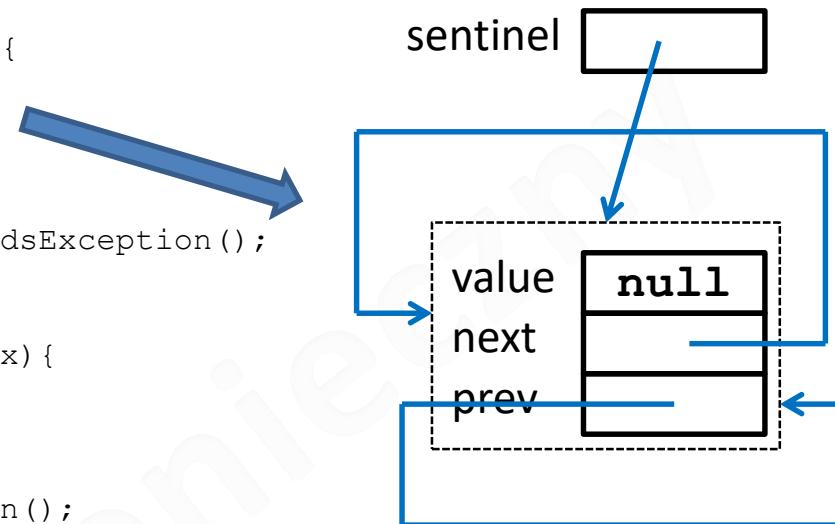
remove ()



- Po zakończeniu usuwania element zostanie w pamięci, jednak nie będzie dostępny (nie powinien być). Realnie usunięty z pamięci zostanie w czasie uruchomienia odśmieczača pamięci.

TwoWayCycledListWithSentinel 1/5

```
Element sentinel=null;
public TwoWayCycledListWithSentinel() {
    sentinel=new Element(null);
    sentinel.setNext(sentinel);
    sentinel.setPrev(sentinel);}
private Element getElement(int index) {
    if(index<0) throw new IndexOutOfBoundsException();
    Element elem=sentinel.getNext();
    int counter=0;
    while(elem!=sentinel && counter<index) {
        counter++;
        elem=elem.getNext();}
    if(elem==sentinel)
        throw new IndexOutOfBoundsException();
    return elem;}
private Element getElement(E value) {
    Element elem=sentinel.getNext();
    while(elem!=sentinel && !value.equals(elem.getValue())){
        elem=elem.getNext();}
    if(elem==sentinel)
        return null;
    return elem;}
```



- Zasada działania metod `getElement()` x2 jest w zasadzie taka sama jak dla listy jednokierukowej. Są to prywatne metody operujące na obiektach klasy `Element`, niewidocznej dla użytkownika listy.

TwoWayCycledListWithSentinel 2/5

```
public boolean isEmpty() {  
    return sentinel.getNext()==sentinel;  
}  
public void clear() {  
    sentinel.setNext(sentinel);  
    sentinel.setPrev(sentinel);}  
public boolean contains(E value) {  
    return indexOf(value)!=-1;}  
public E get(int index) {  
    Element elem=getElement(index);  
    return elem.getValue();}  
public E set(int index, E value) {  
    Element elem=getElement(index);  
    E retValue=elem.getValue();  
    elem.setValue(value);  
    return retValue;}  
public boolean add(E value) {  
    Element newElem=new Element(value);  
    sentinel.insertBefore(newElem);  
    return true;}  
public boolean add(int index, E value) {  
    Element newElem=new Element(value);  
    if(index==0) sentinel.insertAfter(newElem);  
    else{  
        Element elem=getElement(index-1);  
        elem.insertAfter(newElem);}  
    return true;}
```

- Metody add () używają metody insertAfter () i insertBefore () klasy wewnętrznej Element

TwoWayCycledListWithSentinel 3/5

```
public int indexOf(E value) {  
    Element elem=sentinel.getNext();  
    int counter=0;  
    while(elem!=sentinel && !elem.getValue().equals(value)) {  
        counter++;  
        elem=elem.getNext();}  
    if(elem==sentinel)  
        return -1;  
    return counter;}  
public E remove(int index) {  
    Element elem=getElement(index);  
    elem.remove();  
    return elem.getValue();}  
public boolean remove(E value) {  
    Element elem=getElement(value);  
    if(elem==null) return false;  
    elem.remove();  
    return true;}  
public int size() {  
    Element elem=sentinel.getNext();  
    int counter=0;  
    while(elem!=sentinel){  
        counter++;  
        elem=elem.getNext();}  
    return counter;}  
public Iterator<E> iterator() {  
    return new TWCIterator();}
```

- Metody `remove()` używają metodę `remove()` z klasy wewnętrznej `Element`

TwoWayCycledListWithSentinel 4/5

```
private class TWCIterator implements Iterator<E>{
    Element _current=sentinel;
    public boolean hasNext() {
        return _current.getNext()!=sentinel;
    }
    public E next() {
        _current=_current.getNext();
        return _current.getValue();
    }
}

public ListIterator<E> listIterator() {
    return new TWCLListIterator();
}

private class TWCLListIterator implements ListIterator<E>{
    boolean wasNext=false;
    boolean wasPrevious=false;
    /** strażnik */
    Element _current=sentinel;
    public boolean hasNext() {
        return _current.getNext()!=sentinel;
    }
    public boolean hasPrevious() {
        return _current!=sentinel;
    }
    public int nextIndex() {
        throw new UnsupportedOperationException();
    }
    public int previousIndex() {
        throw new UnsupportedOperationException();
    }
}
```

Do poprawnej implementacji usuwania trzeba wiedzieć, w którą stronę przesuwaliśmy się po liście: do przodu, czy do tyłu.

TwoWayCycledListWithSentinel 5/5

```
public E next() {  
    wasNext=true;  
    wasPrevious=false;  
    _current=_current.getNext();  
    return _current.getValue();}  
public E previous() {  
    wasNext=false;  
    wasPrevious=true;  
    E retValue=_current.getValue();  
    _current=_current.getPrev();  
    return retValue;}  
public void remove() {  
    if(wasNext){  
        Element curr=_current.getPrev();  
        _current.remove();  
        _current=curr;  
        wasNext=false;}  
    if(wasPrevious){  
        _current.getNext().remove();  
        wasPrevious=false;}}  
public void add(E data) {  
    Element newElem=new Element(data);  
    _current.insertAfter(newElem);  
    _current=_current.getNext();}  
public void set(E data) {  
    if(wasNext){  
        _current.setValue(data);  
        wasNext=false;}  
    if(wasPrevious){  
        _current.getNext().setValue(data);  
        wasNext=false;}}}
```

Lista L2KCzS – złożoność

- Lista L2KCzS – złożoność pesymistyczna:
 - Konstrukcja – $O(1)$
 - `isEmpty()`, `clear()` – $O(1)$
 - `getElement()` – $O(n)$
 - `set()`, `get()` – $O(n)$
 - `add()` na końcu – $O(1)$
 - `add()` z indeksem – $O(n)$ – z zerowym – $O(1)$
 - `indexOf()` – $O(n)$
 - `contains()` – $O(n)$
 - `remove()` $\times 2$ – $O(n)$ – z zerowym – $O(1)$
 - `size()` – $O(n)$
- `ListIterator` dla L2KCzS – gdyby zaimplementować wszystkie operacje - złożoność pesymistyczna:
 - `hasNext()`, `next()`, `hasPrevious()`, `previous()`, `nextIndex()`, `previousIndex()` – $O(1)$
 - `set()` – $O(1)$
 - `add()`, `remove()` – $O(1)$
- Lista L2KCzS nadaje się do implementacji nieograniczonej kolejki, gdzie każda operacja (oprócz `size()`) jest stałoczasowa - $O(1)$.

Przykład – lista studentów 1/2

- Należy dopisać porównywanie studentów

```
public class Student{  
    int indexNo;  
    double scholarship;  
    public Student(int nr, double value){  
        indexNo=nr;  
        scholarship=value;  
    }  
    public void increaseScholarship(double value){  
        scholarship+=value;  
    }  
    @Override  
    public String toString(){  
        return String.format("%6d %8.2f\n", indexNo, scholarship);  
    }  
  
    public boolean equals(Student stud) {  
        return indexNo == stud.indexNo;  
    }  
  
    @Override  
    public boolean equals(Object obj) {  
        if(obj==null)  
            return false;  
        if (getClass() != obj.getClass())  
            return false;  
        return equals((Student) obj);  
    }  
}
```

aisd.W03.Student

Przykład – lista studentów 2/2

- Należy dopisać porównywanie studentów

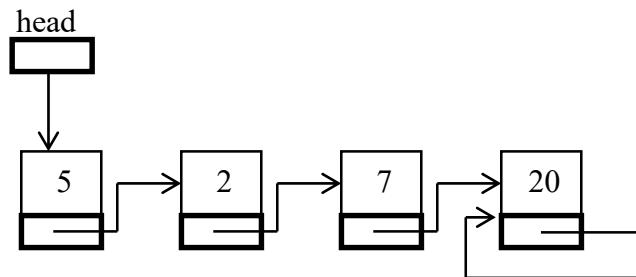
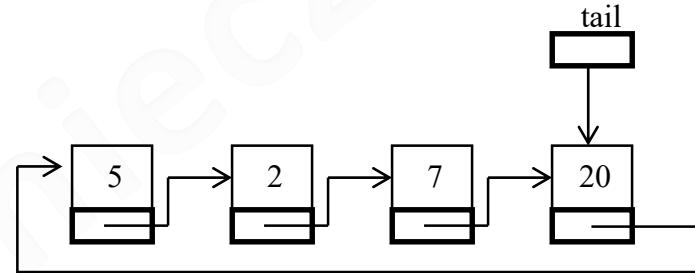
```
public static void testStudentList() {
    TwoWayCycledListWithSentinel<Student> lista=new TwoWayCycledListWithSentinel<>();
    lista.add(new Student(4,1000));
    lista.add(new Student(5,500));
    lista.add(new Student(20,0));
    lista.add(1,new Student(1, 300));
    lista.remove(0);
    lista.remove(new Student(20,10000));
    System.out.println(lista);
    ListIterator<Student> iter=lista.listIterator();
    iter.add(new Student(100,400));
    iter.next();
    iter.add(new Student(101, 600));
    iter.next();
    iter.remove();
    System.out.println(iter.hasNext());
    System.out.println(lista);
    iter.previous();
    iter.remove();
    iter.add(new Student(102,800));
    iter.previous();
    iter.previous();
    iter.previous();
    System.out.println(iter.hasPrevious());
    iter.add(new Student(103,1200));
    System.out.println(iter.hasPrevious());
    System.out.println(lista);
}
```

aisd.W03.AiSD_W03

```
[ 1 300,00
, 5 500,00
]
false
[ 100 400,00
, 1 300,00
, 101 600,00
]
false
true
[ 103 1200,00
, 100 400,00
, 1 300,00
, 102 800,00
]
```

Listy wiązane - rodzaje

- Ze względu na możliwość poruszania się w czasie stałym:
 - Jednokierunkowe
 - Dwukierunkowe
- Ze względu na punkt dostępu do listy:
 - Z głową
 - Z ogonem
 - Z głową i ogonem
- Ze względu na nadmiarowe elementy:
 - Ze strażnikiem
 - Bez strażnika
- Ze względu na strukturę wewnętrzną:
 - Proste
 - Cykliczne
- Specjalne:
 - Lista jednokierunkowa zapętlone na ostatnim elemencie.

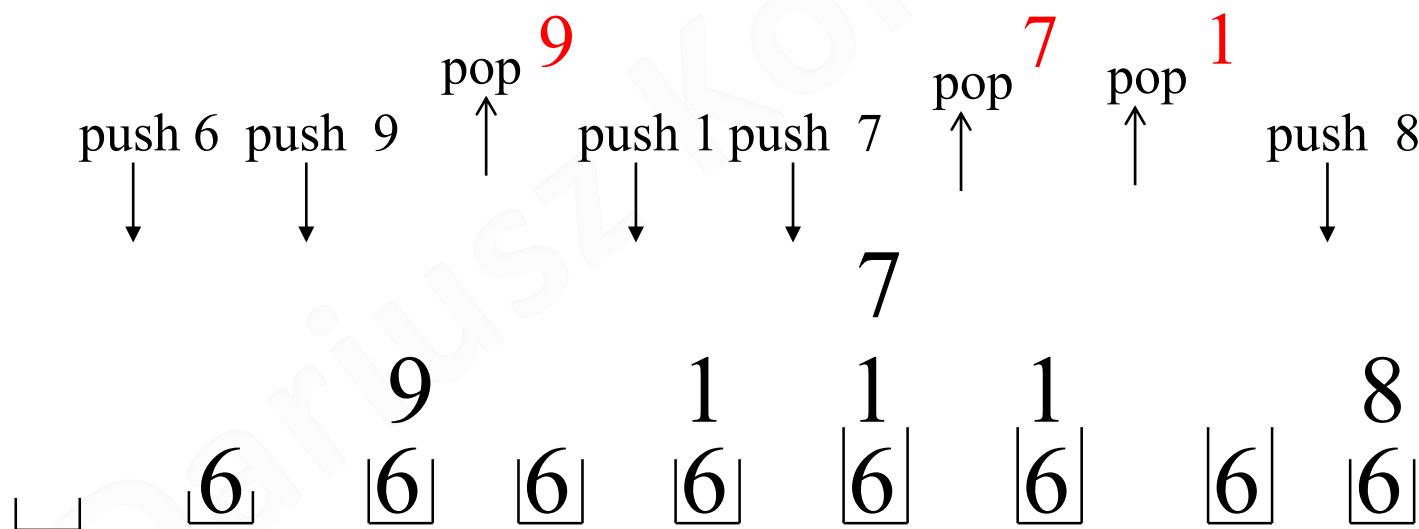


Klasa LinkedList

- W pakiecie `java.util` znajduje się klasa `LinkedList`
- Jest to lista dwukierunkowa
- Prawdopodobnie cykliczna
- Pozwala na wstawianie jako elementu wartości `null`
- Nie jest synchronizowana
- W bibliotece Javy są metody opakowujące do jej synchronizacji.
- Jej metody lub metody iteratorów rzucać mogą wyjątkiem `ConcurrentModificationException` .

Stos

- Klasyczny stos jest strukturą liniową LIFO (ang. Last In First Out), do której mamy dostęp tylko na jednym końcu. Możemy element dodać (operacja push) lub usunąć (operacja pop) z tego końca, zwracając jako wynik. Dodatkowo można sprawdzić, czy stos nie jest pusty lub pełny.



Ciąg operacji na stosie

Interfejs IStack<T>

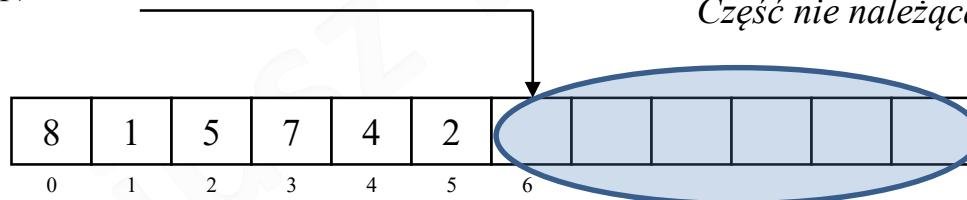
- Trzeba również stos zainicjować, ale w programowaniu obiektowym dzieje się to w konstruktorze. Po stworzeniu stos powinien być pusty.
- W bibliotekach Javy nie ma interfejsu dla stosu, jest od razu klasa stosu: Stack<T>.
- Stwórzmy zatem przykładowy interfejs IStack<T> dla stosu (z dodatkowymi operacjami):

```
public class EmptyStackException extend Exception{  
}  
  
public class FullStackException extend Exception{  
}  
  
public interface IStack<T>{  
    boolean isEmpty();  
    boolean isFull();  
    T pop() throws EmptyStackException;  
    void push(T elem) throws FullStackException;  
    int size(); // zwraca liczbę elementów na stosie  
    T top() throws EmptyStackException;  
        // zwraca element ze szczytu stosu bez usuwania go  
}
```

Stos na tablicy

- Realizacja stosu na tablicy:
 - o ograniczonej pojemności
 - Wystarczy pamiętać, jaka początkowa część tablicy jest elementami stosu. Realizowane to jest za pomocą jednego pola całkowitoliczbowego.
 - o nieograniczonej pojemności
 - Podobnie, jednak przy przekroczeniu pojemności tablicy, następuje rezerwacja nowej, większej i przeniesienie elementów do nowej tablicy.

Szczyt stosu (top) = 6



Implementacja ArrayStack<T> 1/2

- Nie można stworzyć obiektu lub tablicy obiektów klasy T parametru klasy generycznej ArrayStack<T>. Zamiast tego należy stworzyć tablicę obiektów klasy Object i rzutować na klasę tablicy obiektów typu T.
- Pojawia się wówczas ostrzeżenie, które można zignorować dopisując przed metodą dyrektywą kompilatora:
 @SuppressWarnings ("unchecked")

```
public class ArrayStack<T> implements IStack<T> {

    private static final int DEFAULT_CAPACITY = 16;
    T array[];
    int topIndex;

    // klasy generyczne w zasadzie są typu Object
    // pozwala ją jednak już na etapie kompilacji sprawdzać poprawność typów
    @SuppressWarnings("unchecked")
    public ArrayStack (int initialSize){
        array=(T[]) (new Object[initialSize]);
        topIndex=0;
    }

    public ArrayStack () {
        this(DEFAULT_CAPACITY);
    }
}
```

Implementacja ArrayStack<T> 2/2

```
@Override  
public boolean isEmpty() {  
    return topIndex==0; }  
  
@Override  
public boolean isFull() {  
    return topIndex==array.length; }  
  
@Override  
public T pop() throws EmptyStackException {  
    if(isEmpty())  
        throw new EmptyStackException();  
    return array[--topIndex]; }  
  
@Override  
public void push(T elem) throws FullStackException {  
    if(isFull())  
        throw new FullStackException();  
    array[topIndex++]=elem; }  
  
@Override  
public int size() {  
    return topIndex; }  
  
@Override  
public T top() throws EmptyStackException {  
    if(isEmpty())  
        throw new EmptyStackException();  
    return array[topIndex-1]; }
```

Zapis @Override

nie jest obowiązkowy, ale może ustrzec nas od błędów, na przykład:

```
@Override  
public boolean Empty() {  
    ...}
```

Wygeneruje błąd kompilacji, bo w interfejsie nie ma metody Empty()

Złożoność operacji na stosie

- Dla implementacji na tablicy:
 - Konstrukcja: $O(n)$
 - isEmpty: $O(1)$
 - isFull: $O(1)$
 - pop: $O(1)$
 - push: $O(1)$
 - size: $O(1)$
 - top: $O(1)$

Stos za pomocą L1KPzG

```
public class ListStack<E> implements IStack<E> {
    IList<E> _list;
    public ListStack() {
        _list = new OneWayLinkedListWithHead<E>();
    }
    @Override
    public boolean isEmpty() {
        return _list.isEmpty();
    }
    @Override
    public boolean isFull() {
        return false;
    }
    @Override
    public E pop() throws EmptyStackException {
        E value=_list.remove(0);
        if(value==null) throw new EmptyStackException();
        return value;
    }
    @Override
    public void push(E elem) throws FullStackException {
        _list.add(0,elem);
    }
    @Override
    public int size() {
        return _list.size();
    }
    @Override
    public E top() throws EmptyStackException {
        E value=_list.get(0);
        if(value==null) throw new EmptyStackException();
        return value;
    }
}
```

Złożoność operacji na stosie

- Dla implementacji na L1KPzG:
 - Konstrukcja: $O(1)$
 - isEmpty: $O(1)$
 - isFull: $O(1)$
 - pop: $O(1)$
 - push: $O(1)$
 - size: $O(1)$ – z polem `_size` ($O(n)$ – bez)
 - top: $O(1)$

Stosy nieklasyczne

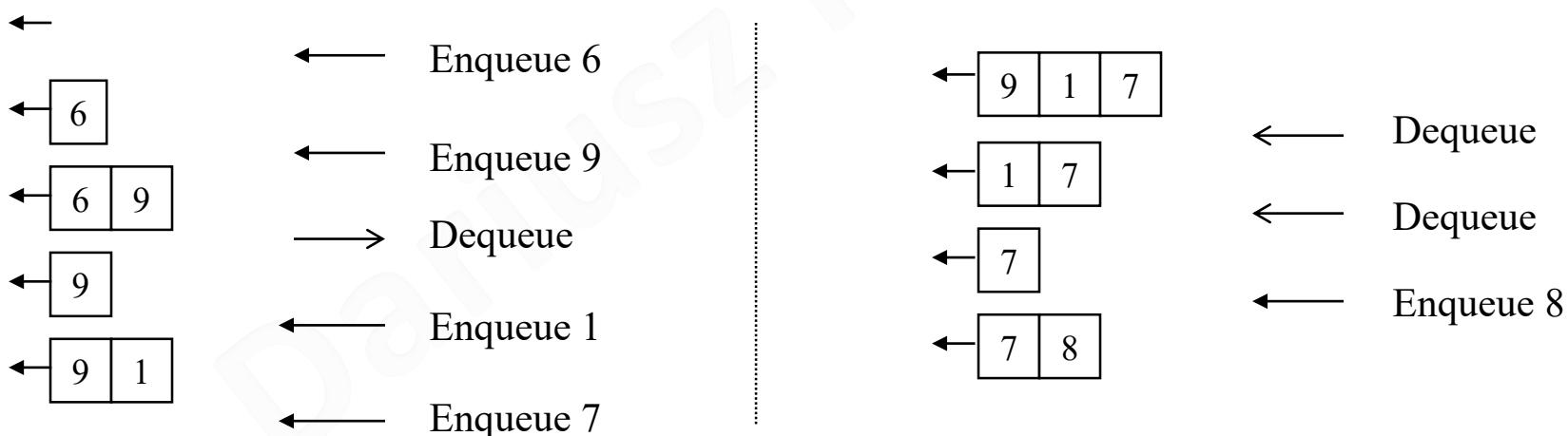
- Stos tonący:
 - Stos o ograniczonej pojemności, w momencie wstawiania elementu do pełnego stosu następuje odrzucenie elementu na dnie stosu („tonie”), aby zrobić miejsce na szczycie dla nowego elementu
- Dodatkowe operacje na stosie:
 - Szukanie elementu (zwraca głębokość elementu względem szczytu stosu)
 - Maksymalny rozmiar stosu
 - Możliwość usuwania dowolnego elementu ze stosu – rzadko
 - Iterator do poruszania się po stosie bez jego zmiany

Zastosowanie stosu

- Podstawowa struktura dynamiczna, występuje w wielu algorytmach (wiele takich algorytmów będzie na tym kursie)
- Programy podczas uruchamiania metod/funkcji/procedur przekazują informacje przez stos programu.
 - Adres powrotu z wywołania funkcji
 - Parametry wywołania funkcji
 - Zwracany rezultat
 - Dzięki temu mechanizmowi możliwe są metody rekurencyjne
- Zarządzanie ekranami aplikacji (stos okienek)
- Zarządzanie ciągiem zmian (np. w edytorze tekstu) z możliwością ich wycofywania i ponawiania (undo/redo)
- Odwracanie kolejności wyrazów w ciągu
- Itd., itd.

Kolejka

- Kolejka FIFO (ang. First In First Out), w skrócie nazywana po prostu kolejką, to struktura liniowa w której użytkownik ma dostęp do obydwóch końców, z tym, że do jednego końca (koniec kolejki, ogon kolejki) może dodawać elementy, a z drugiego (początek kolejki, głowa kolejki) pobierać. Rozwinięcie skrótu FIFO oznacza, że element pierwszy wstawiony do kolejki będzie pierwszym wyciągniętym, zasada dla kolejnych elementów jest analogiczna.
- Z powyższego opisu wynika, że potrzeba następujących operacji dla kolejki:
 - Stworzenie kolejki
 - Wstawienie elementu do kolejki (enqueue)
 - Pobranie elementu z kolejki (dequeue)
 - Sprawdzenie, czy kolejka jest pusta (isEmpty)
 - Sprawdzenie, czy kolejka jest pełna (isFull)



Interfejs IQueue<T>

- Tworzenie kolejki będzie w konstruktorze. Po stworzeniu kolejka powinna być pusta.
- W bibliotekach Javy interfejs dla kolejki jest przygotowany do wspomagania programowania wielowątkowego, więc jest zbyt rozbudowany dla tego wykładu.
- Stwórzmy zatem przykładowy interfejs IQueue<T> dla kolejki (z dodatkowymi operacjami):

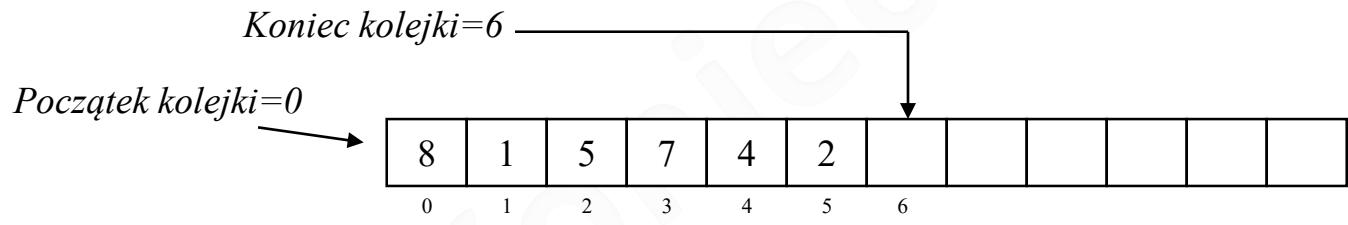
```
public class EmptyQueueException extends Exception{  
}  
  
public class FullQueueException extends Exception{  
}  
  
public interface IQueue<T>{  
    boolean isEmpty();  
    boolean isFull();  
    T dequeue() throws EmptyQueueException;  
    void enqueue(T elem) throws FullQueueException;  
    int size(); // zwraca liczbę elementów w kolejce  
    T first() throws EmptyQueueException;  
        // zwraca element z początku kolejki bez usuwania go  
}
```

Clear() ?

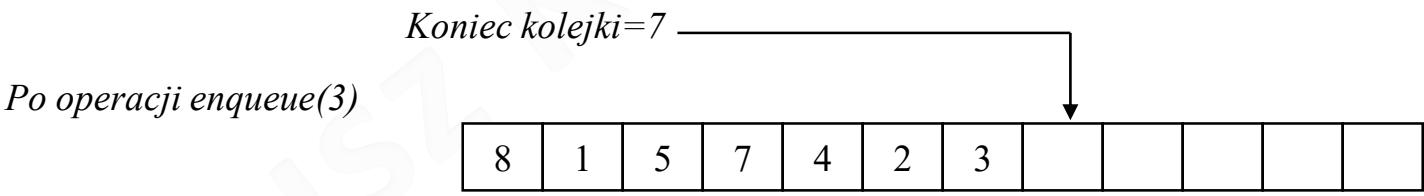
Kolejka za pomocą tablicy

- Nieefektywne rozwiązanie:

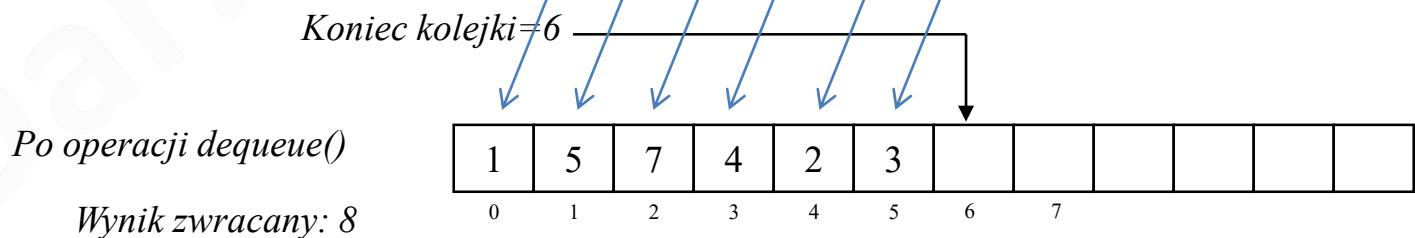
- zerowy indeks to zawsze początek kolejki
- pamięta się tylko długość kolejki (jak dla stosu).
- Przesuwanie wszystkich elementów o jedną pozycję w lewo podczas pobierania elementu z kolejki



O(1)



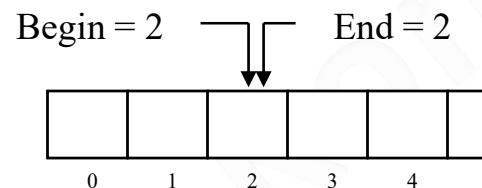
O(n)



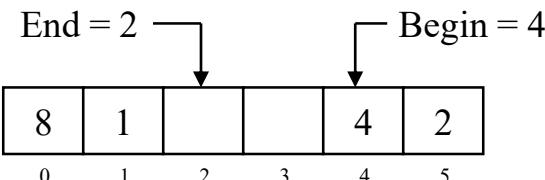
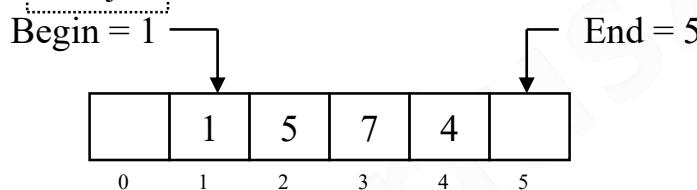
Efektywna implementacja kolejki na tablicy

- Efektywna realizacja kolejki za pomocą tablicy, wykorzystujący elementy tablicy w sposób cykliczny:
 - o ograniczonej pojemności
 - Najbardziej eleganckie rozwiązanie tworzy tablice z o jeden większym rozmiarze niż podany w konstruktorze. Oprócz tego w obiekcie są dwa indeksy wskazujące początek kolejki oraz miejsce za końcem kolejki
 - o nieograniczonej pojemności
 - Podobnie, jednak przy przekroczeniu pojemności tablicy, następuje rezerwacja nowej, większej i przeniesienie elementów do nowej tablicy.

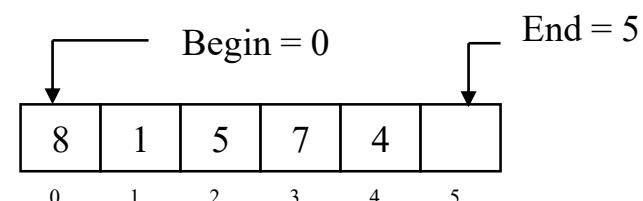
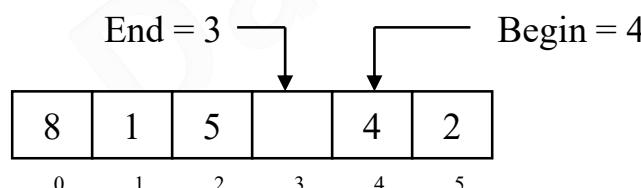
Pusta kolejka



Kolejka

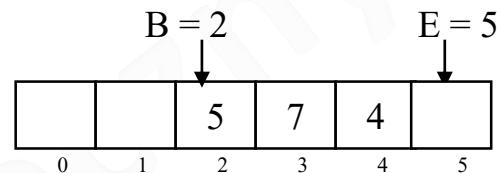
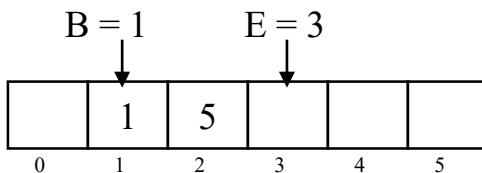
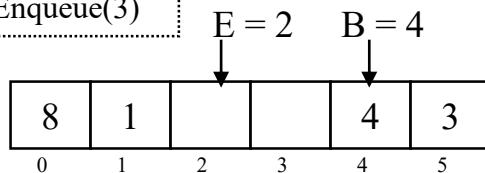


Pełna kolejka



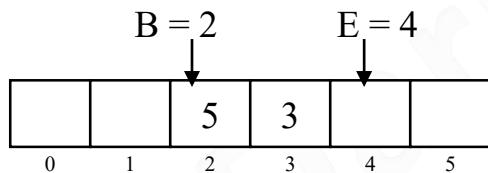
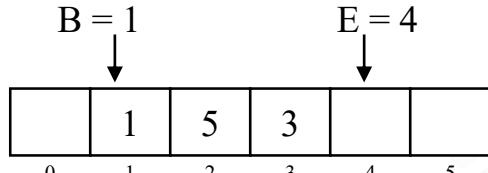
Operacje enqueue i dequeue

Enqueue(3)

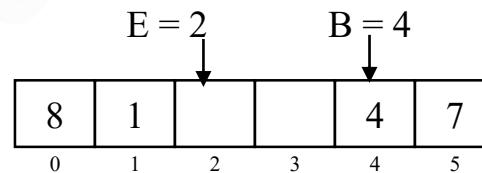
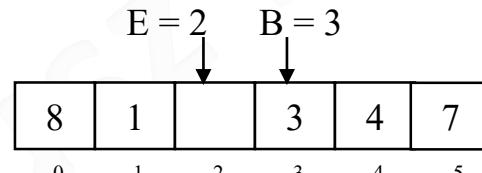


a)

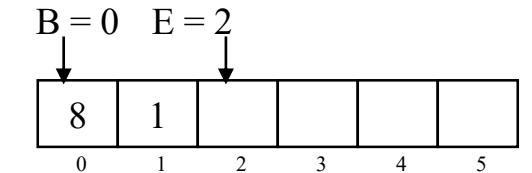
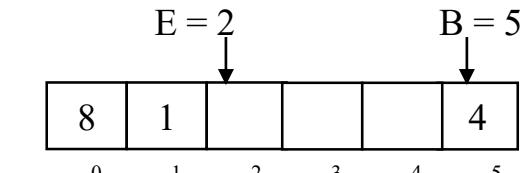
Dequeue



b)



c)



Kolejka – implementacja 1/2

```
public class ArrayQueue<T> implements IQueue<T> {  
  
    private static final int DEFAULT_CAPACITY = 16;  
    T array[];  
    int beginIndex;  
    int endIndex;  
  
    @SuppressWarnings("unchecked")  
    public ArrayQueue(int size) {  
        array=(T[])new Object[size+1];  
    }  
  
    public ArrayQueue() {  
        this(DEFAULT_CAPACITY);  
    }  
  
    @Override  
    public boolean isEmpty() {  
        return beginIndex==endIndex;  
    }  
  
    @Override  
    public boolean isFull() {  
        return beginIndex==(endIndex+1)%array.length;  
    }  
}
```

O(n)

O(1)

O(1)

Kolejka – implementacja 2/2

```
@Override  
public T dequeue() throws EmptyQueueException {  
    if (isEmpty())  
        throw new EmptyQueueException();  
    T retValue=array[beginIndex++];  
    beginIndex%=array.length;  
    return retValue;  
}  
  
@Override  
public void enqueue(T elem) throws FullQueueException {  
    if (isFull())  
        throw new FullQueueException();  
    array[endIndex++]=elem;  
    endIndex%=array.length;  
}  
  
@Override  
public int size() {  
    return (endIndex+array.length-beginIndex) % array.length;  
}  
  
@Override  
public T first() throws EmptyQueueException {  
    if (isEmpty())  
        throw new EmptyQueueException();  
    return array[beginIndex];  
}
```

O(1)

O(1)

O(1)

O(1)

Kolejka za pomocą L2KCzS

```
public class ListQueue<E> implements IQueue<E>{
    TwoWayCycledListWithSentinel<E> _list;
    public ListQueue() {
        _list=new TwoWayCycledListWithSentinel<E>();
    }
    @Override
    public boolean isEmpty() {
        return _list.isEmpty();
    }
    @Override
    public boolean isFull() {
        return false;
    }
    @Override
    public E dequeue() throws EmptyQueueException {
        E value=_list.remove(0);
        if(value==null) throw new EmptyQueueException();
        return value;
    }
    @Override
    public void enqueue(E elem) throws FullQueueException {
        _list.add(elem);
    }
    @Override
    public int size() {
        return _list.size();
    }
    @Override
    public E first() throws EmptyQueueException {
        E value=_list.get(0);
        if(value==null) throw new EmptyQueueException();
        return value;
    }
}
```

Złożoność operacji na kolejce

- Dla implementacji na L2KCzS :
 - Konstrukcja: $O(1)$
 - isEmpty: $O(1)$
 - isFull: $O(1)$
 - enqueue: $O(1)$ (na koniec kolejki)
 - dequeue: $O(1)$
 - size: $O(1)$ – z polem `_size` ($O(n)$ – bez)
 - first: $O(1)$

JUnit

- Biblioteka do **automatycznych** testów jednostkowych.
- Są różne, kolejne wersje. Na tym wykładzie wykorzystywana będzie wersja 4.
- Wiele środowisk programistycznych (np. Eclipse) posiada wsparcie wizualne do prezentowania przebiegu i wyników testów, raportowanie itp.
- Tworzy się niezależną klasę do testowania (nie zaśmieca się testowanej klasy)
- Liczba testów nieograniczona.
- Metody `setUp()`, `tearDown()` ... uruchamiane przed i po każdym teście.

Zestaw testów szczególnie przydatny przy modyfikacji metod – teoretyczne usprawnienie może zepsuć funkcjonalność klasy.

Demonstracja działania kolejki za pomocą Junit-ów

Zastosowanie kolejki

- Podstawowa struktura dynamiczna, występuje w wielu algorytmach (wiele takich algorytmów będzie na tym kursie)
- Zarządzanie żądaniami do serwera (WWW, bazy danych itd.) w architekturze klient-serwer
- Bufor podczas przetwarzania w problemie producenci-konsumenci
- Buforowanie odczytu z urządzeń fizycznych
- Wszelkie strumienie (danych, rozkazów, multimedialne) to w zasadzie kolejki.
- „Kolejka” to domyślnie „kolejka FIFO”. Inne rodzaje kolejek poznamy na kolejnych wykładach

Zadania na ćwiczenia/laboratorium

- Zaimplementować wyliczanie wartości wyrażenia będącego już w postaci ONP.
- Zaimplementować stos tonący
- Veloso's Traversable Stack jest to stos, który poza zwykłymi operacjami ma możliwość nieniszczącego odczytu z pozycji wskazywanej przez kurSOR (peek). KurSOR można ustawić na wierzchołek stosu (top) i przesuwać o jedną pozycję w dół stosu (down - potrzebna jest sygnalizacja osiągnięcia dna stosu). Normalne operacje (push i pop) automatycznie ustawiają kurSOR na wierzchołek. Zaimplementuj VTS jako rozszerzenie zwykłego stosu.
- Zaimplementować stos i kolejkę o ograniczonym rozmiarze jako macierz dwuwymiarową $n*n$.
- *TODO: Problemy z serwera zadań z collegiate programming.*

Algorytmy i struktury danych – W04

Teoria złożoności cz. 4/4

Komparator

Proste algorytmy sortowania

Zawartość

- Złożoność cz. 3 - analiza złożoności:
 - Rodzaje złożoności:
 - Pesymistyczna
 - Średnia
- Sortowanie – definicja
- Interfejsy porównujące obiekty - komparatory:
 - Comparable (porządek naturalny)
 - Comparator
- Komparator odwrotny
- Komparator złożony
- Interfejs RandomAccess
- Proste algorytmy sortowania - $O(n^2)$
 - Sortowanie przez wstawianie
 - Sortowanie przez wybór
 - Sortowanie bąbelkowe

Notacja

- Wejściem I dla algorytmu są dane takie jak liczby, ciąg znaków, rekordy na który algorytm wykonuje operacje przetwarzające
- Niech \mathcal{F}_n oznacza wszystkich wejść do algorytmu o rozmiarze n .
- Dla $I \in \mathcal{F}_n$ niech $\tau(I)$ oznacza liczbę elementarnych kroków, które wykonuje algorytm dla wejścia I
- τ – czyta się *tau*

Złożoność pesymistyczna

- Złożoność pesymistyczna (najgorszego przypadku, ang. *Worst-case complexity*) algorytmu jest funkcją $W(n)$ taką, że $W(n)$ równa się maksymalnej wartości $\tau(I)$, gdzie I przyjmuje wartości spośród wszystkich możliwych wejść o rozmiarze n . Czyli,

$$W(n) = \max \left\{ \tau(I) \mid I \in \mathcal{F}_n \right\}$$

Złożoność pesymistyczna - przykład

Posortuj n elementów poprzez wygenerowanie wszystkich permutacji elementów, sprawdzając, czy kolejne permutacja jest posortowana. Jeśli jest posortowana, zakończ generowanie permutacji i cały algorytm.

Założenia:

- następna generacja jest wykonywana w jednym kroku elementarnym
- Sprawdzanie, czy permutacja jest rozwiązaniem wykonywane jest w jednym kroku

Przy tych założeniach:

$$W(n) = 2n! = O(n!)$$

Złożoność średnia/oczekiwana

Niech $p(I)$ będzie prawdopodobieństwem, że wejście I będzie daną wejściową

Średnia/oczekiwana złożoność (ang. average (expected) complexity) $A(n)$ algorytmu ze skończonym zbiorem wejść F_n jest definiowana jako:

$$A(n) = \sum_{I \in F_n} \tau(I) p(I) = E[\tau]$$

Formuła II

Niech p_i oznacza prawdopodobieństwo, że algorytm wykona *dokładnie i kroków elementarnych*; czyli $p_i = P(\tau=i)$.

Wówczas:

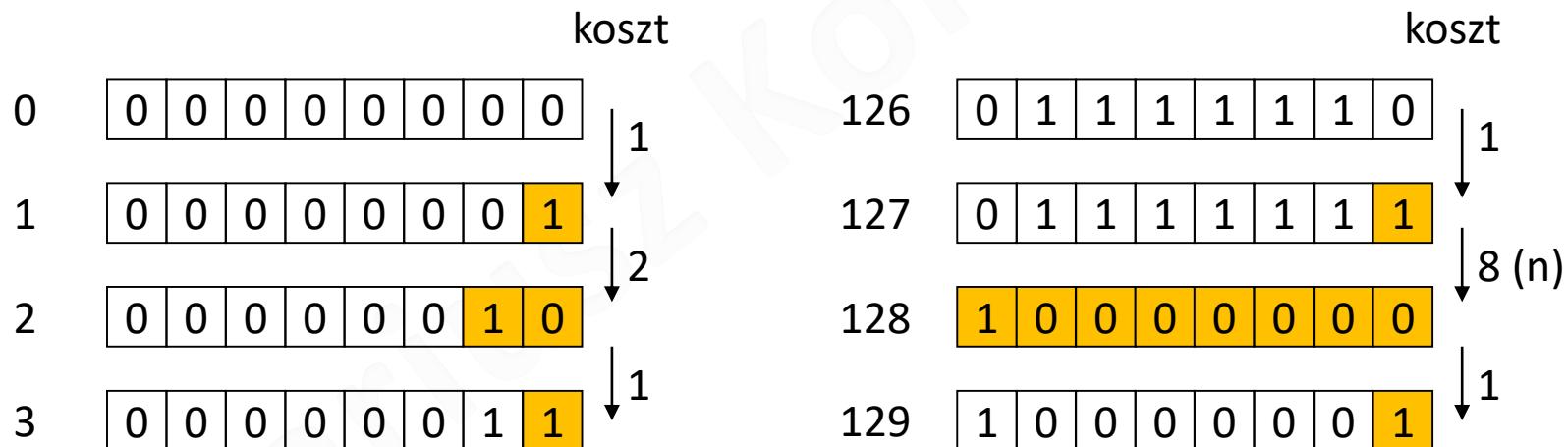
$$A(n) = E[\tau] = \sum_{i=1}^{W(n)} ip_i$$

Koszt zamortyzowany

W analizie kosztu zamortyzowanego pokazuje się, że dla wszystkich n , ciąg n operacji zajmuje pesymistycznie czas całkowity $T(n)$. W pesymistyczny przypadek **średni koszt**, lub **zamortyzowany koszt**, dla jednej operacji jest zatem $T(n)/n$. Należy zauważyć, że koszt zamortyzowany odnosi się do każdej operacji, również wtedy, gdy w ciągu operacji były one różnych typów.

Licznik n -bitowy

- Mamy n -bitowy binarny licznik, w którym wartość możemy tylko zwiększać o jeden (n – rozmiar danych wejściowych).
- Operacja elementarna (koszt) – zmiana **jednego** bitu (z 0 na 1 oraz z 1 na 0)



Analiza metodą kosztu zamortyzowanego

- Złożoność pesymistyczna: $W(n)=O(n)$
- Złożoność średnia? Może $A(n)=O(n/2)$? Nie!

$$\begin{aligned}\frac{2^n}{2} \cdot 1 + \frac{2^n}{4} \cdot 2 + \frac{2^n}{8} \cdot 3 + \dots + \frac{2^n}{2^n} \cdot n &= 1 \cdot 2^{n-1} + 2 \cdot 2^{n-2} + 3 \cdot 2^{n-3} + \dots + n \cdot 2^{n-n} \\&= (2^{n-1} + 2^{n-2} + \dots + 2^0) + 1 \cdot 2^{n-2} + 2 \cdot 2^{n-3} + \dots + (n-1) \cdot 2^0 \\&= (2^n - 1) + (2^{n-2} + \dots + 2^0) + 1 \cdot 2^{n-3} + \dots + (n-2) \cdot 2^0 \\&= (2^n - 1) + (2^{n-1} - 1) + \dots + (2^1 - 1) + (2^0 - 1) \\&= 2^{n+1} - 1 - n\end{aligned}$$

$$A(n) < \frac{2^{n+1}}{2^n} = 2 \quad \Rightarrow \quad A(n) = O(1)$$

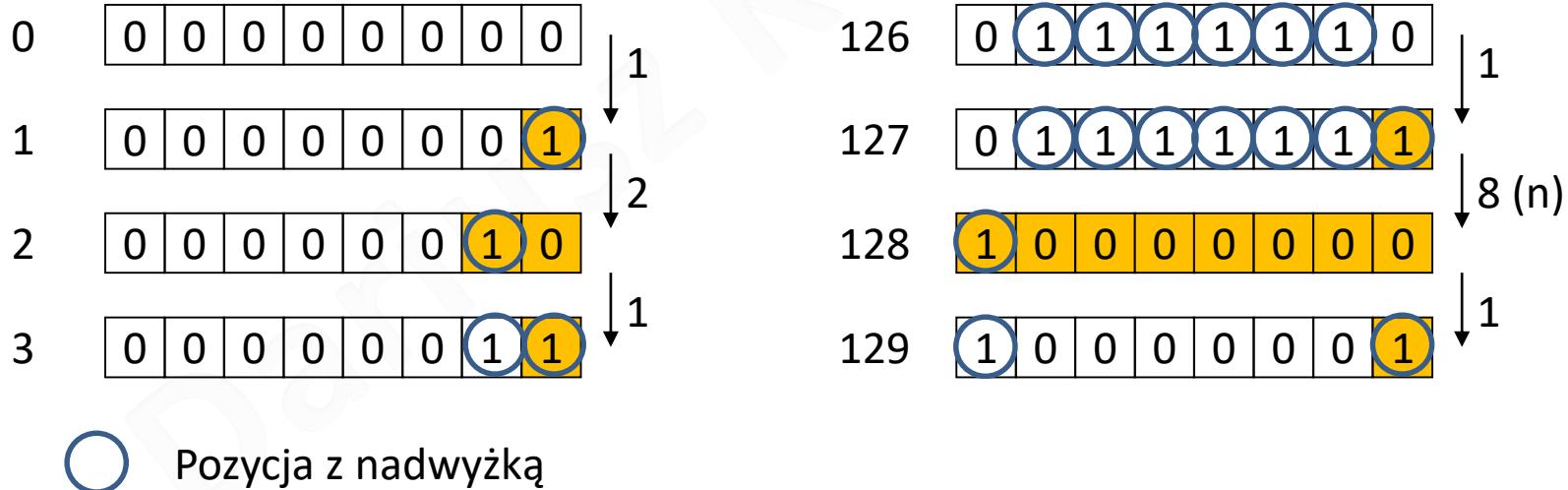
Metoda księgowania

- W metodzie księgowania kosztu zamortyzowanego, do każdej operacji przyporządkowujemy koszt, który może być mniejszy lub większy od rzeczywistego kosztu. Jednak koszt rzeczywisty nigdy nie może być większy niż powstały zamortyzowany koszt. Nadwyżka ponad rzeczywisty koszt ciągu operacji nazywamy kredytem. Kredyt może być użyty do opłacenia tych operacji, których koszt rzeczywisty jest większy niż ustalony koszt zamortyzowany.
- Najczęściej mamy operacje, które odbywają się wcześniej i których koszt ustalamy na większy niż w rzeczywistości, natomiast operacje późniejsze obciążamy mniejszym kosztem niż koszt rzeczywisty. Natomiast możemy pokazać, że wcześniej zebrany kredyt NA PEWNO uzupełni braki.

Licznik n -bitowy – met. księgowania

Używając metody księgowania dla n -bitowego licznika, obciążmy operację zmiany bitu na 1 opłatą równą 2. Gdy ustawiamy bit na 1, jedną z tych dwóch jednostek płacimy za rzeczywisty koszt zmiany bitu, pozostała jednostkę kładziemy na tym biecie, aby użyć ją przy powrotnej zmianie na 0. W każdym momencie wszystkie 1 w liczniku mają na siebie to jednostkę, więc zmieniając nawet wiele jedynek na zero, mamy wystarczającą nadwyżkę na taką zamianę. Oczywiście płacąc z nadwyżki, zdejmujemy tą jednostkę z danego bitu.

- Ponieważ w każdej sytuacji tylko 0 jest zmieniane na 1, zatem w każdym kroku płacimy 2 jednostki za zmianę, czyli jest to dokładnie średni zamortyzowany koszt operacji zwiększenia o jeden na tym liczniku.



Sortowanie - definicja

- **Sortowanie:** proces, w którym kolekcję elementów ustawiamy w określonym porządku
 - Operacje porządkujące dane:
 - Porównanie
 - Zamiana lub przypisanie
 - Porządkowanie jest najczęściej wg pewnego klucza
 - Bardziej uniwersalnym podejściem jest używanie **komparatora**

Definicja formalna:

Dla początkowej tablicy S , składającej się z N elementów stworzyć tablicę wynikową S' taką, że:

- 1) $S'_{i+1} \leq S'_i$, dla $0 < i < N$ (elementy są posortowane) oraz
- 2) S' jest permutacją S .

Cechy sortowania 1/2

- Metody używania elementów
 - Przez porównanie elementów – komparator
 - Bez bezpośredniego porównywania elementów
- Stabilność - czy elementy o jednakowych kluczach zachowają pierwotną kolejność :
 - **Stabilny**
 - Niestabilny
- Odległość między elementami porównywanyimi:
 - **Bliskie**
 - Dalekie
- Możliwość zastosowania do częściowo posortowanej kolekcji:
 - Sortowanie zawsze całej kolekcji
 - „**Dosortowanie” nowych elementów**

(klucz, dana dodatkowa)

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

Stabilność sortowania:



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

Cechy sortowania 2/2

- Złożoność obliczeniowa:
 - Proste: $O(n^2)$
 - Pośrednie: $O(n^{3/2})$ i inne
 - Szybkie: $O(n \log n)$
 - Specjalnego zastosowania: $O(n)$
- Potrzeba dodatkowej pamięci:
 - w miejscu - bez dodatkowej pamięci większej niż $O(1)$
 - Potrzebna pamięć większa niż $O(1)$
- Stopień skomplikowania implementacji
- Miejsce lub struktura danych wejściowych:
 - Pamięć RAM
 - Plik
 - Tablica, lista wiązana
 - Strumień
- Liczba porównywania lub przepisywania elementów (w C++ można przepisywać całe elementy, w Javie – raczej referencje)
- Wrażliwość na przypadki danych wejściowych:
 - dane losowe (najczęstsza sytuacja)
 - dane posortowane (w większości posortowane)
 - dane posortowane odwrotnie
 - dane wszystkie równe (lub wiele równych)
 - inne

Komparator - idea

- Komparator składa się w zasadzie z jednej funkcji z dwoma parametrami A i B typu X, która zwraca informację, czy element A powinien być w ustalonym porządku przed elementem B, czy powinien być za elementem B, czy też są to te same elementy (ze względu na ustalony porządek, co nie znaczy, że są to elementy identyczne).
- Funkcja ta powinna być określona dla wszystkich możliwych par danego typu X oraz wyznaczać liniowy porządek.
- Liniowy porządek (\leq) ma właściwości (dla dowolnych A,B,C typu X):
 - **Zwrotność:** $(A \leq A)$
 - **Spójność:** $(A \leq B) \vee (B \leq A)$
 - **Antysymetryczność:** $((A \leq B) \text{ oraz } (B \leq A)) \Rightarrow (A = B)$
 - **Przechodniość:** $((A \leq B) \text{ oraz } (B \leq C)) \Rightarrow (A \leq C)$
- Komparator musi zwracać w sumie 3 rodzaje wyniki:



Interfejs Comparable

- Posiada tylko jedną funkcję
- Wyznacza w kolekcji tzw. **porządek naturalny** elementów.
- Wynik powinien być zgodny z poprzednim slajdem.
- Porównuje bieżący obiekt (`this`) z argumentem `other`: porównaj (`this, other`).
- Wada: w ten sposób obiekt może udostępniać tylko jeden porządek.

```
interface Comparable<T>{
    int compareTo(T other);
}
```

```
public class Student implements Comparable<Student>{
    ...
    @Override
    public int compareTo(Student other) {
    }
}
```

- Ciekawostka: dla klasy `String` wartość zwracana przez metodę `compareTo()` oznacza różnicę kodów znaków na pierwszym miejscu, gdzie ciągi się różnią lub różnicę długości, gdy jeden jest prefiksem drugiego.

Klasa Student i naturalny porządek

```
public class Student implements Comparable<Student>{
    String nazwisko;
    int nrIndeksu;
    double stypendium;
    public Student(String nazw, int nr, double kwota) {
        nazwisko=nazw;
        nrIndeksu=nr;
        stypendium=kwota; }
    public void zwięksStypendium(double kwota) {
        stypendium+=kwota; }
    @Override
    public String toString() {
        return String.format("%6d %8.2f\n",nrIndeksu,stypendium); }
    public boolean equals(Student stud) {
        return nazwisko.equals(stud.nazwisko); }
    @Override
    public boolean equals(Object obj) {
        if(obj==null)
            return false;
        if (getClass() != obj.getClass())
            return false;
        return equals((Student) obj); }
    @Override
    public int compareTo(Student o) {
        return nazwisko.compareTo(o.nazwisko); }
}
```

Interfejs Comparator

- W Javie istnieje inny interfejs: Comparator używany w wielu kolekcjach i metodach.
- Posiada tylko jedną funkcję
- Można tworzyć wiele takich interfejsów, bo obydwa argumenty otrzymuje jako parametry.
- Każda implementacja takiego interfejsu to nowa klasa.
- Aby nie tworzyć nazw dla klas z takim komparatorem można tworzyć klasy anonimowe.
- Zamiast w metodach/strukturach korzystających z komparator tworzyć dwie implementacje dla dwóch rodzajów interfejsów, interfejs Comparable przekształca się do interfejsu Comparator

```
interface Comparator<T>{
    int compare(T left, T right);
}
```

```
class XYZ<T> {
    Comparator<T> comparator;

    public XYZ(Comparator<T> comp) {
        comparator=comp;    }
    public XYZ() {
        comparator=new Comparator(){      // klasa anonimowa
            public int compare(T first, T other) {
                return first.compareTo(other);
            }
        }
    }
}
```

To jest szkielet kodu,
niekompilowalny

Komparator porządku naturalnego

- Można stworzyć klasę generyczną to tworzenia komparatorów naturalnych porównujących wg porządku naturalnego podanej w parametrze klasy

```
public class NaturalComparator<T extends Comparable<? super T>> implements Comparator<T> {  
    @Override  
    public int compare(T o1, T o2) {  
        return o1.compareTo(o2);  
    }  
}
```

Komparator odwrotny

- Można stworzyć klasę, która umożliwia, z zadanego komparatora wejściowego, stworzenie komparatora odwrotnego

```
public class ReverseComparator<T extends Comparable<? super T>> implements  
    Comparator<T> {  
    // podstawowy komparator  
    private final Comparator<T> _comparator;  
    public ReverseComparator(Comparator<T> comparator)  
    {  
        _comparator = comparator;  
    }  
    @Override  
    public int compare(T left, T right)  
    {  
        return _comparator.compare(right, left);  
    }  
}
```

- Poprawny zapis **obiektowy** w języku Java dla takich klas wymagałby dodatkowego wytłumaczenia, stąd w dalszych materiałach zostanie on porzucony, celem skupienia się na algorytmice, a nie specyfikacji klas generycznych języka Java.
- Zamiast tego używane będzie wyłączanie ostrzeżeń o możliwych kłopotach z niepoprawnym rzutowaniem.

Komparator złożony

- Zamiast tworzyć za każdym razem nowe, skomplikowane komparatory można tworzyć komparator poprzez złożenie prostszych komparatorów.
- W takim przypadku porównanie elementów nastąpi najpierw poprzez użycie pierwszego komparatora. Gdy on uzna, że elementy są równe, uruchamia się drugi komparator itd.
- Komparatory przechowywane będą w liście/tablicy.

```
public class CompoundComparator<T> implements Comparator<T> {  
    //tablica komparatorów ; od najważniejszego  
    private final IList<Object> _comparators =new ArrayList<Object>();  
    public void addComparator(Comparator<T> comparator)  
    {  
        _comparators.add(comparator);  
    }  
    @SuppressWarnings("unchecked")  
    public int compare(T left, T right) throws ClassCastException {  
        int result = 0;  
        for (Object obj:_comparators){  
            Comparator<T> comp=(Comparator<T>) obj;  
            result=comp.compare(left, right);  
            if(result!=0) break;  
        }  
        return result;  
    }  
}
```

Testowanie komparatorów

```
public static void main(String[] args) {  
    Student stud1=new Student("Nowak",1234, 1200.0);  
    Student stud2=new Student("Nowak",1230, 1000.0);  
    Student stud3=new Student("Kowalski",1111, 2000.0);  
    System.out.println("porzadek naturalny:");  
    System.out.println(stud1.compareTo(stud2));  
    System.out.println(stud1.compareTo(stud3));  
    System.out.println(stud2.compareTo(stud3));  
    System.out.println(stud3.compareTo(stud2));  
    Comparator<Student> revComp=new ReverseComparator<Student>()  
        new NaturalComparator<Student>());  
    System.out.println("komparator odwrocony:");  
    System.out.println(revComp.compare(stud2,stud3));  
    CompoundComparator<Student> complexComp=new CompoundComparator<Student>();  
    complexComp.addComparator(new Comparator<Student>() {  
        public int compare(Student o1, Student o2) {  
            return o1.nazwisko.compareTo(o2.nazwisko);}  
    });  
    complexComp.addComparator(new Comparator<Student>() {  
        public int compare(Student o1, Student o2) {  
            return o1.nrIndeksu-o2.nrIndeksu;}  
    });  
    complexComp.addComparator(new Comparator<Student>() {  
        public int compare(Student o1, Student o2) {  
            if (o1.stypendium<o2.stypendium) return -1000;  
            else if(o1.stypendium>o2.stypendium) return 1000;  
            else return 0;}  
    });  
    System.out.println("komparator złożony:");  
    System.out.println(complexComp.compare(stud1,stud2));  
}
```

porzadek naturalny:

0

3

3

-3

komparator odwrocony:

-3

komparator złożony:

4

Interfejs RandomAccess

- Jeden z kilku interfejsów **bez metod!**
- Służy jako **oznaczenie** kolekcji jako takiej, w której metody `get(int index)` i `set(int index, T element)` są stałoczasowe, czyli $O(1)$.
- Np. metody z klasy `Collections` sprawdzają, czy kolekcja implementuje ten interfejs czy nie i uruchamia odpowiednie wewnętrzne działanie.
- Klasa `ArrayList` **zapewnia** interfejs `RandomAccess`.
- Klasa `LinkedList` **nie zapewnia** interfejsu `RandomAccess`.

```
interface RandomAccess {  
}
```

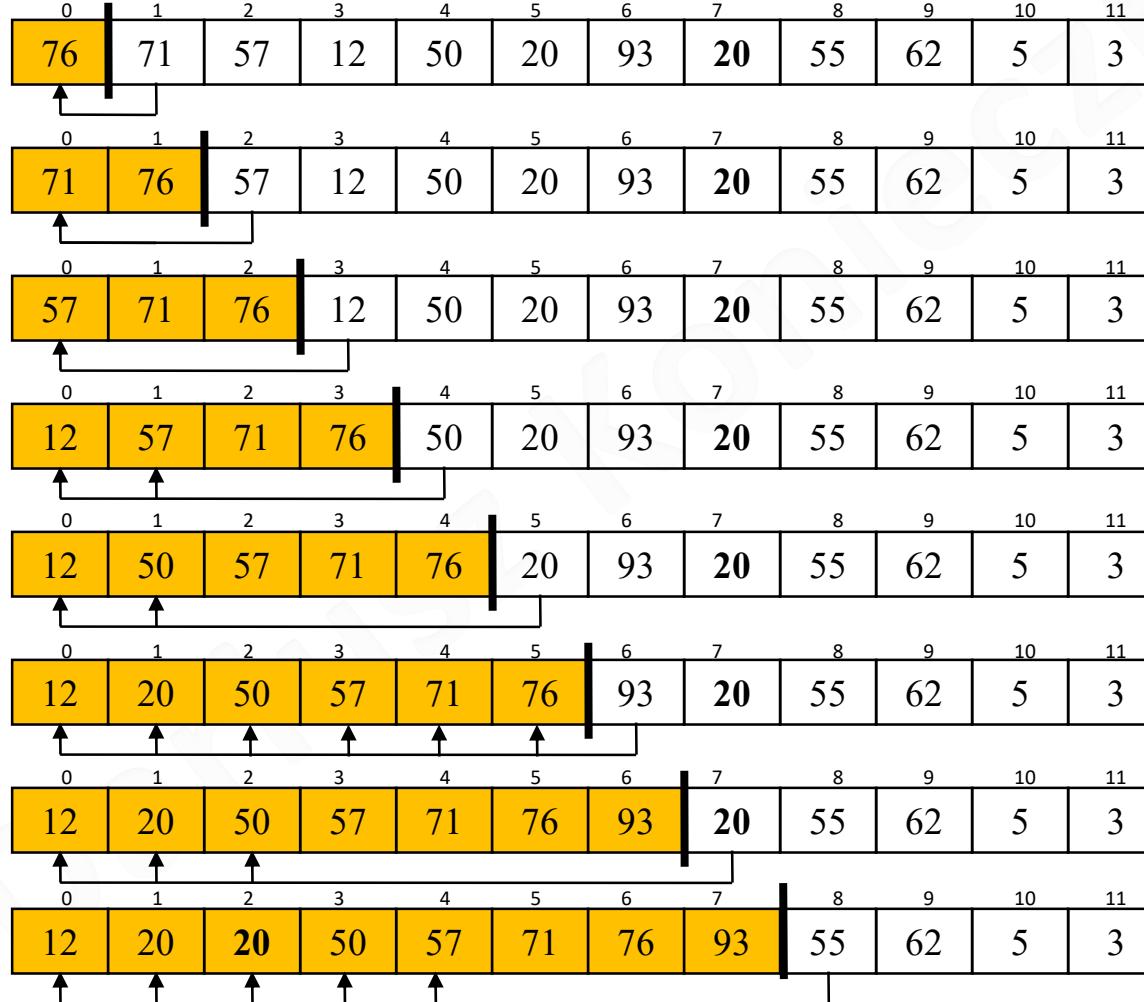
Interfejs ListSorter

- Dla celów dydaktycznych stwórzmy interfejs dla klas sortujących ListSorter<T>.
- Każda klasa z algorytmem sortowania będzie musiała implementować ten interfejs.
- W większości metod prezentowanych na wykładzie będzie założenie, że lista wejściowa dostarczają interfejsu RandomAccess.
- Implementacja prezentowanych algorytmów w wersji dla list wiązanych jest dobrym ćwiczeniem praktycznym.

```
public interface ListSorter<T> {  
    public IList<T> sort(IList<T> list);  
}
```

Sortowanie przez wstawianie

- Jak układanie kart do gry po rozdaniu graczom
- Idea: do części już posortowanej dokładany kolejny element, szukając dla niego poprawnej pozycji.



S. przez wstawianie - kod

- Wersja z poszukiwaniem miejsca do wstawienia od prawej strony
- Podczas poszukiwania miejsca – przesuwanie elementów.

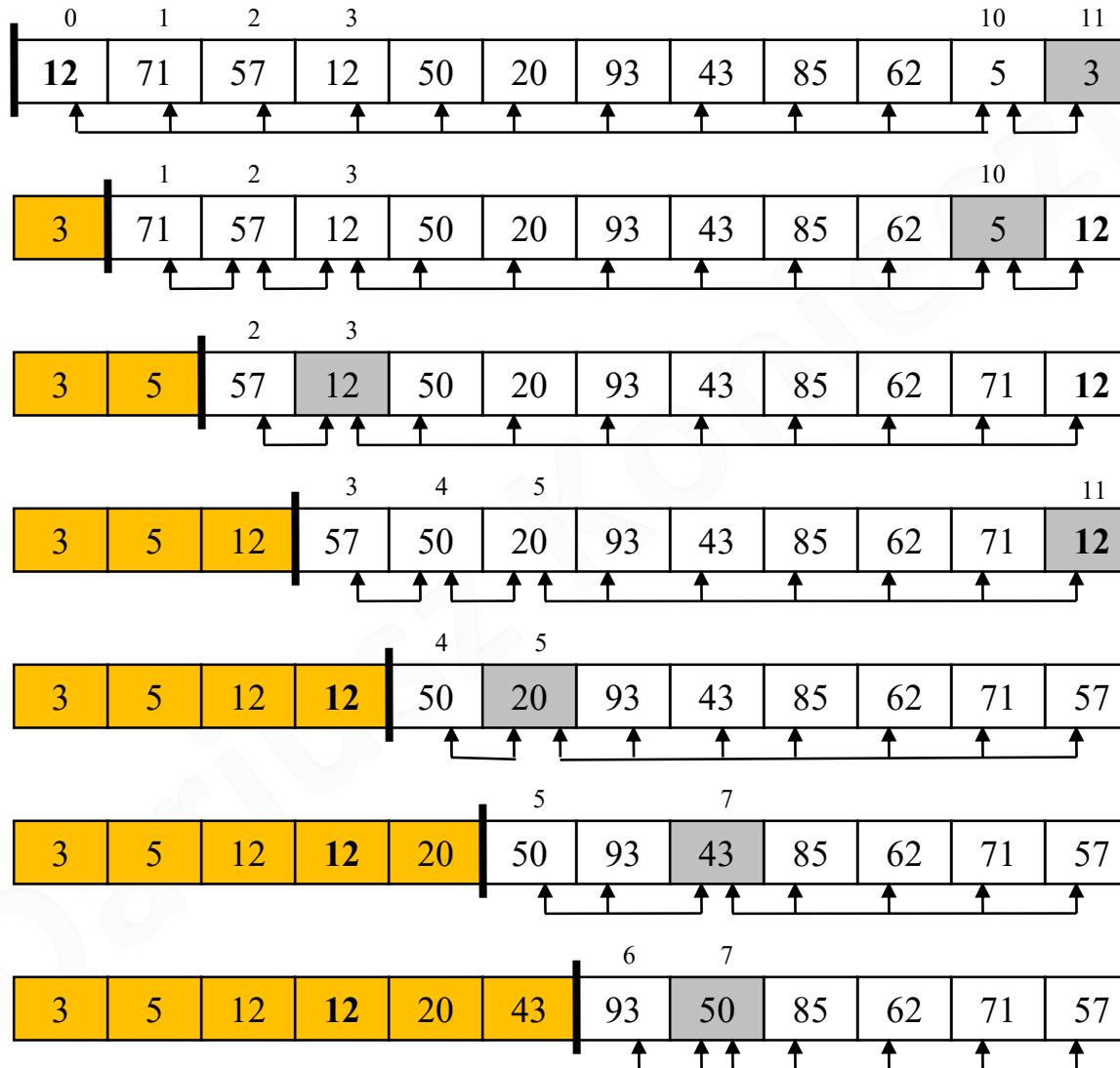
```
public class InsertSort<T> implements ListSorter<T> {  
    private final Comparator<T> _comparator;  
    public InsertSort(Comparator<T> comparator)  
    {  
        _comparator = comparator;  
    }  
    public IList<T> sort(IList<T> list) {  
        for (int i = 1; i < list.size(); ++i) {  
            T value = list.get(i), temp;  
            int j; // będzie wykorzystywane poza pętla  
            for (j = i; j > 0 && _comparator.compare(value, temp=list.get(j - 1)) < 0; --j)  
                list.set(j, temp);  
            list.set(j, value);  
        }  
        return list;  
    }  
}
```

S. przez wstawianie - analiza

- Złożoność obliczeniowa (pesymistyczna, średnia) – $O(n^2)$
- Sortowanie w miejscu
- Sortowanie stabilne
- Działa szybko dla małych list – algorytm progowy dla algorytmu szybkiego
- Idea sortowania przez wstawianie używana w listach wiązanych posortowanych
- Warianty:
 - Część posortowana narasta od lewej/prawej strony
 - Szukamy miejsca do wstawienia od lewej/prawej strony
- Możliwość usprawnień:
 - Szukać miejsca wstawienia używając poszukiwania binarnego – czas $O(\log n)$
 - Szukać miejsca od prawej strony – szczególnie gdy kolekcja jest w dużych fragmentach już posortowana:
1,3,5,6,2,7,8,10,11,9,13

Sortowanie przez wybór

- Idea: szukamy kolejnej wartości minimalnej (maksymalnej) oraz dostawienie jej do wcześniej znalezionych wartości.



S. przez wybór - kod

- Wybór minimum
- Wydzielona operacja swap

```
public class SelectSort<T> implements ListSorter<T> {  
    private final Comparator<T> _comparator;  
    public SelectSort(Comparator<T> comparator) {  
        _comparator = comparator;  
    }  
    public IList<T> sort(IList<T> list) {  
        int size = list.size();  
        for (int slot = 0; slot < size - 1; ++slot) {  
            int smallest = slot; // pozycja wartości minimalnej  
            for (int check = slot + 1; check < size; ++check)  
                if (_comparator.compare(list.get(check), list.get(smallest)) < 0)  
                    smallest = check;  
            swap(list, smallest, slot);  
        }  
        return list;  
    }  
    private void swap(IList<T> list, int left, int right) {  
        if (left != right) {  
            T temp = list.get(left);  
            list.set(left, list.get(right));  
            list.set(right, temp);  
        }  
    }  
}
```

S. przez wstawianie - analiza

- Złożoność obliczeniowa pesymistyczna i średnia – $O(n^2)$
- Sortowanie w miejscu
- Sortowanie niestabilne! – dodanie w komparatorze porównywania indeksu przed sortowaniem wprowadza stabilność
- Łatwe do implementacji, szczególnie, gdy już jest zaimplementowanie poszukiwanie pozycji minimalnego elementu
- Dużo porównań $O(n^2/2)$, ale tylko $O(n)$ przepisań/zamian
- Usprawnienia:
 - Szukać jednocześnie wartości minimalnej i maksymalnej – mniej porównań i podstawień elementów.

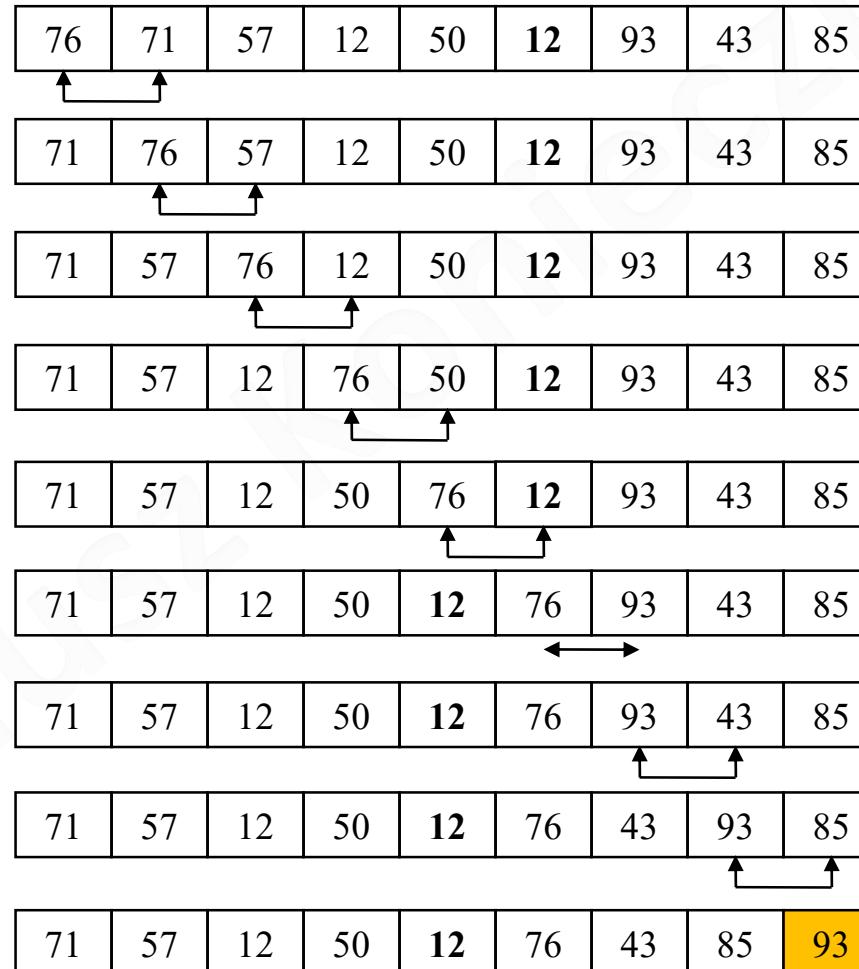
Sortowanie bąbelkowe

- Idea: lokalne zamiany między sąsiednimi elementami, jeśli nie są we właściwej kolejności. Zamiany robi się od lewej do prawej od zerowego elementu, potem od pierwsze go itd.

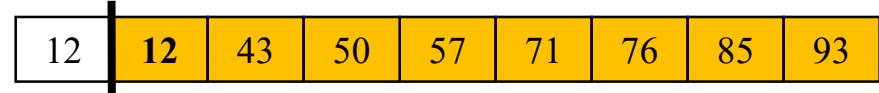
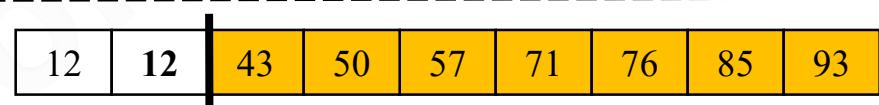
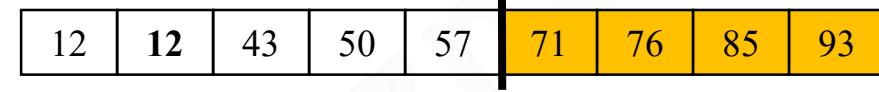
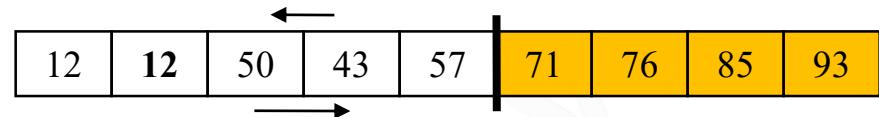
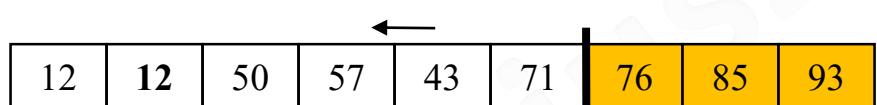
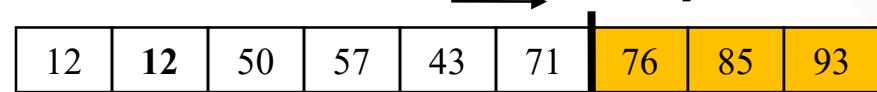
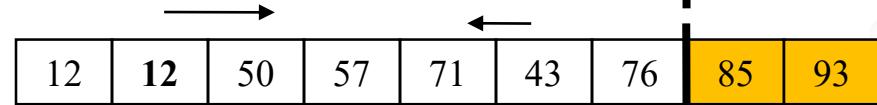
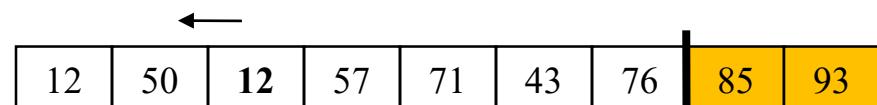
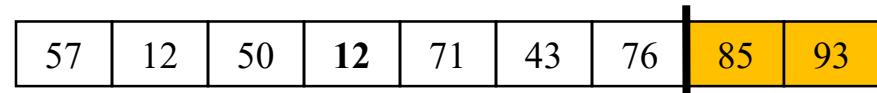
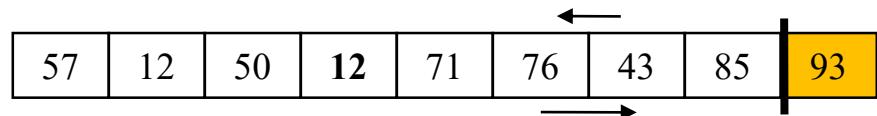
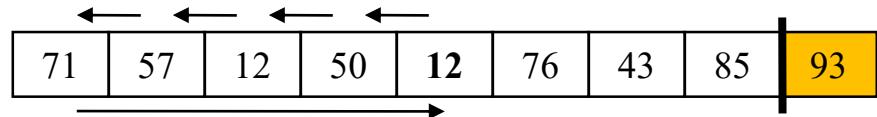
Jeden duży krok algorytmu

↑↑ zamiana

↔↔ bez zamiany



Sortowanie bąbelkowe – c.d.



Sortowanie bąbelkowe - kod

```
public class BubbleSort<T> implements ListSorter<T> {
    private final Comparator<T> _comparator;
    public BubbleSort(Comparator<T> comparator)
        { _comparator = comparator; }
    // the result is a sorted primary list
    // najbardziej prymitywna wersja
    public IList<T> sort(IList<T> list) {
        int size = list.size();
        for (int pass = 1; pass < size; ++pass) {
            for (int left = 0; left < (size - pass); ++left) {
                int right = left + 1;
                if (_comparator.compare(list.get(left), list.get(right)) > 0)
                    swap(list, left, right);
            }
        }
        return list;
    }
    private void swap(IList<T> list, int left, int right) {
        T temp = list.get(left);
        list.set(left, list.get(right));
        list.set(right, temp);
    }
}
```

S. bąbelkowe - analiza

- Złożoność obliczeniowa pesymistyczna i średnia – $O(n^2)$
 - Wolne działanie tego algorytmu wynika z faktu że pojedyncza zamiana sąsiednich elementów zmienia stopień nieuporządkowania ciągu (liczony jako liczba inwersji - ile razy wartość większa występuje przed mniejszą) tylko o 1.
- Bardzo łatwy do implementacji
- Sortowanie stabilne
- Sortowanie w miejscu
- Bliska odległość między porównywanyymi elementami
- Możliwe usprawnienia:
 - Sprawdzać, czy ciąg nie jest już posortowany (nie było żadnej zamiany)
 - Pamiętać miejsce ostatniej zmiany (dalej ciąg jest już posortowany i nie ulegnie zmianie)
 - Zamiast ciągu lokalnych zamian (operacja swap) robić przesunięcia w lewo – 3 razy mniej podstawień.
 - Poruszać się raz w prawo, raz w lewo (ShakerSort) – dobre dla ciągów w większości posortowanych (z pierwszym usprawnieniem)
- Usprawnienia mogą pogorszyć czas wykonania gdy ciąg jest losowy, a porównanie elementów szybkie – dominuje czas potrzebny na dodatkowe operacje.

S. bąbelkowe – kod z usprawnieniami

- Kod dla 3 pierwszych usprawnień:

```
public class BubbleSortBetter<T> implements ListSorter<T> {  
    private final Comparator<T> _comparator;  
    public BubbleSortBetter(Comparator<T> comparator)  
    {  
        _comparator = comparator;  
    }  
    // wynikiem jest posortowana lista pierwotna  
    // wersja ulepszona wykrywająca wcześniejsze uporządkowanie  
    public IList<T> sort(IList<T> list) {  
        int lastSwap = list.size()-1; //pozycja ostatniej zamiany  
        while(lastSwap>0){  
            int end=lastSwap;  
            lastSwap=0;  
            for (int left = 0; left < end; ++left) {  
                if (_comparator.compare(list.get(left), list.get(left+1)) > 0)  
                { //ciąg zamian jest zastąpiony ciągiem przepisań  
                    T temp=list.get(left);  
                    while(left<end && _comparator.compare(temp, list.get(left+1)) > 0)  
                    { list.set(left, list.get(left+1)); left++; }  
                    lastSwap=left;  
                    list.set(left,temp);  
                }  
            }  
        }  
        return list;  
    }  
}
```

Podsumowanie

- Sortowania proste o złożoności $O(n^2)$ są nieefektywne dla dużych kolekcji
- Dla małych kolekcji potrafią być szybsze od algorytmów o złożoności $O(n \log n)$, które będą przedstawione na kolejnym wykładzie
 - Używane są jako algorytmy progowe, gdy kolekcje są małe
- Istnieje algorytm używający algorytmu bąbelkowego jako wewnętrzny – ShellSort – o złożoności $O(n^{3/2})$.
- Prawie wszystkie przedstawione implementacje używają metod `get` / `set` z indeksem i zakładają, że operacje te są o złożoności $O(1)$ (interfejs `RandomAccess`).
 - Jeśli (np. dla `LinkedList`), operacje te są złożoności $O(n)$, to ostateczna złożoność przedstawionych sortowań będzie $O(n^3)$!
- Część z tych algorytmów można przekształcić na wersję używającą iteratorów, wtedy byłaby użyteczna dla list wiązanych
 - ale mogłaby być mniej efektywna dla tablic

Algorytmy i struktury danych – W05

Efektywne algorytmy sortowania

Zawartość

Sortowania efektywne – $O(n \log n)$

- Sortowanie przez scalanie
 - Wersja rekurencyjna
 - Wersja iteracyjna
- Sortowanie szybkie
 - Pesymistyczne $O(n^2)$
 - Średnie $O(n \log n)$
- Sortowanie przez kopcowanie (sort. stogowe)
- Inne sortowania
 - Przez zliczanie - $O(n)$
 - Pozycyjne – nawet $O(n)$
 - Kubekowe – nawet $O(n)$
- Sortowanie w bibliotece standardowej Javy
 - z RandomAccess
 - **bez** RandomAccess

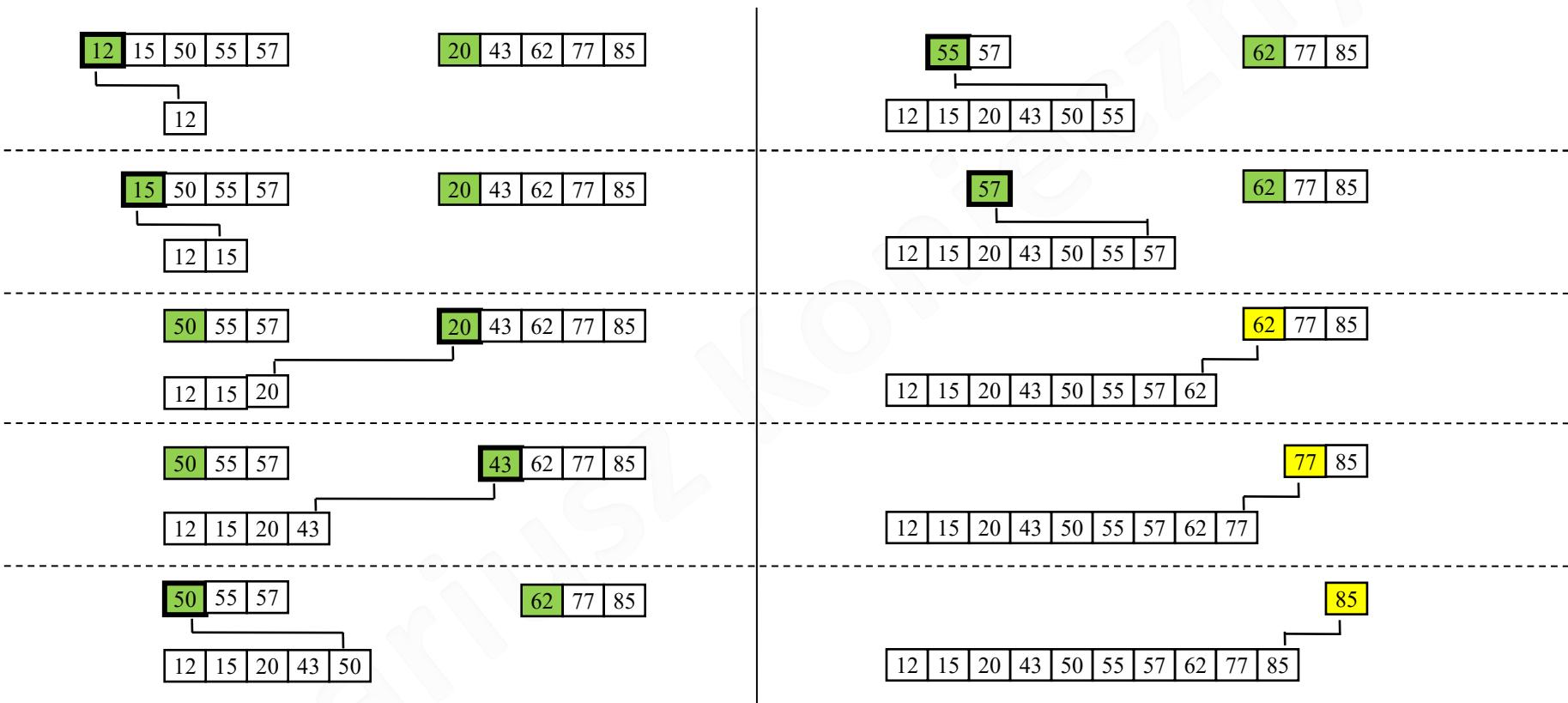
Sortowanie przez scalanie

- Idea:
 - Łączenie dwóch posortowanych ciągów w jeden posortowany można wykonać w czasie $O(n)$
 - Ciągi jednoelementowe są posortowane
 - Za pomocą techniki „dziel i rządź” dzielimy串 na dwa podciągi równe (lub prawie równe), sortujemy każdy (stosując rekurencyjnie ta samą metodę sortowania), łączymy wyniki w串 posortowany

-
- [12] Element porównywany, **wybrany** do danych wyjściowych
 - [62] Element porównywany, nie wybrany do danych wyjściowych
 - [62] Element przepisywany do danych wyjściowych

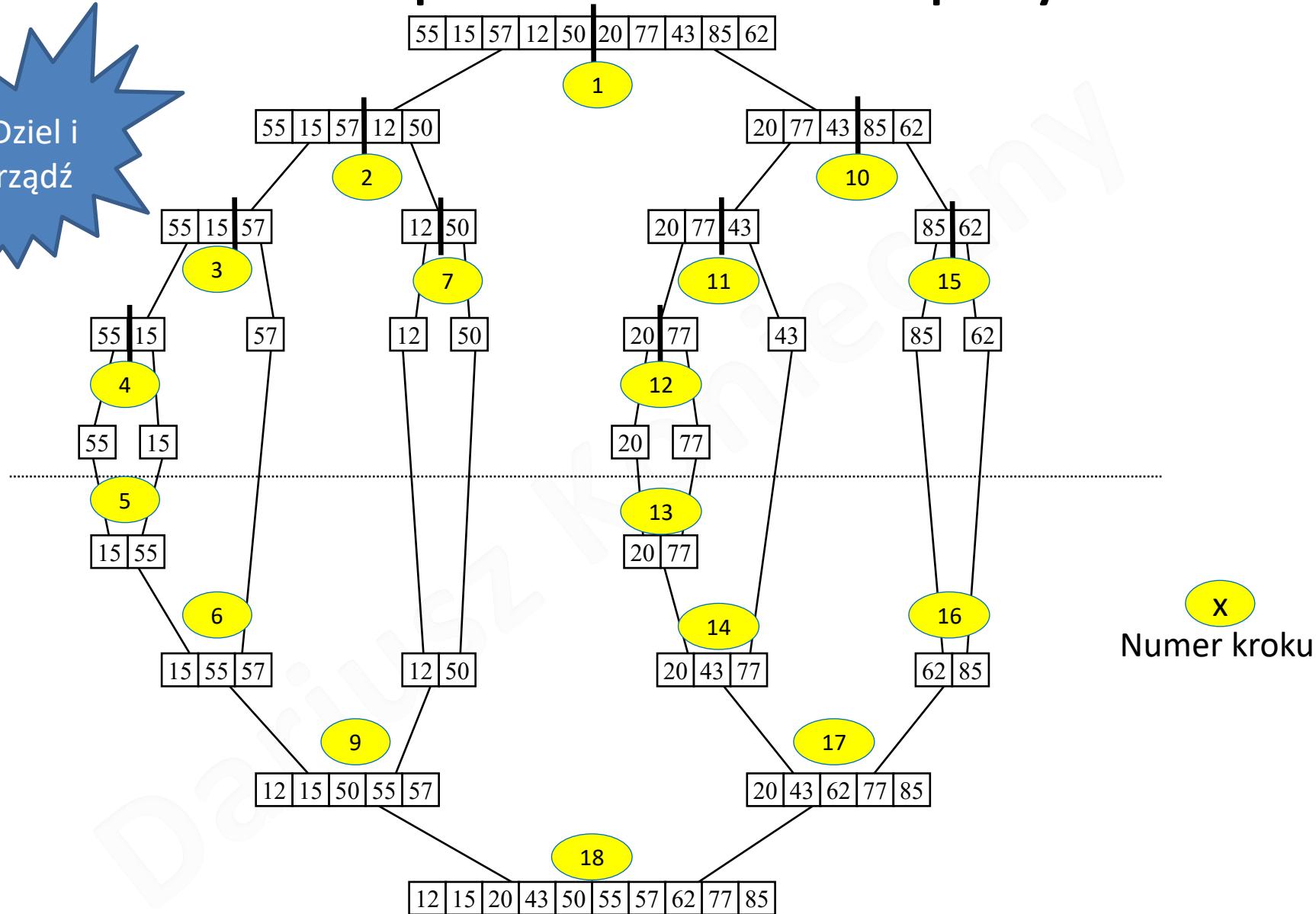
Scalanie w czasie $O(n)$

- W sposób zachłanny do wyniku dodaje kolejny element z dwóch wejściowych list wybierając zawsze mniejszą wartość (gdy są równe – lewy).



Sortowanie przez scalanie - przykład

Dziel i
rządź



Sortowanie przez scalanie – kod 1/2

```
// wynikiem jest nowa lista
public IList<T> sort(IList<T> list)
{
    return mergesort(list, 0, list.size() - 1);
}

@SuppressWarnings("unchecked")
private IList<T> mergesort(IList<T> list, int startIndex, int endIndex) {
    if (startIndex == endIndex) {
        IList<T> result = (IList<T>) (new ArrayList<Object>());
        result.add(list.get(startIndex));
        return result;
    }
    int splitIndex = startIndex + (endIndex - startIndex) / 2;
    return merge(mergesort(list, startIndex, splitIndex),
                  mergesort(list, splitIndex + 1, endIndex));
}
```

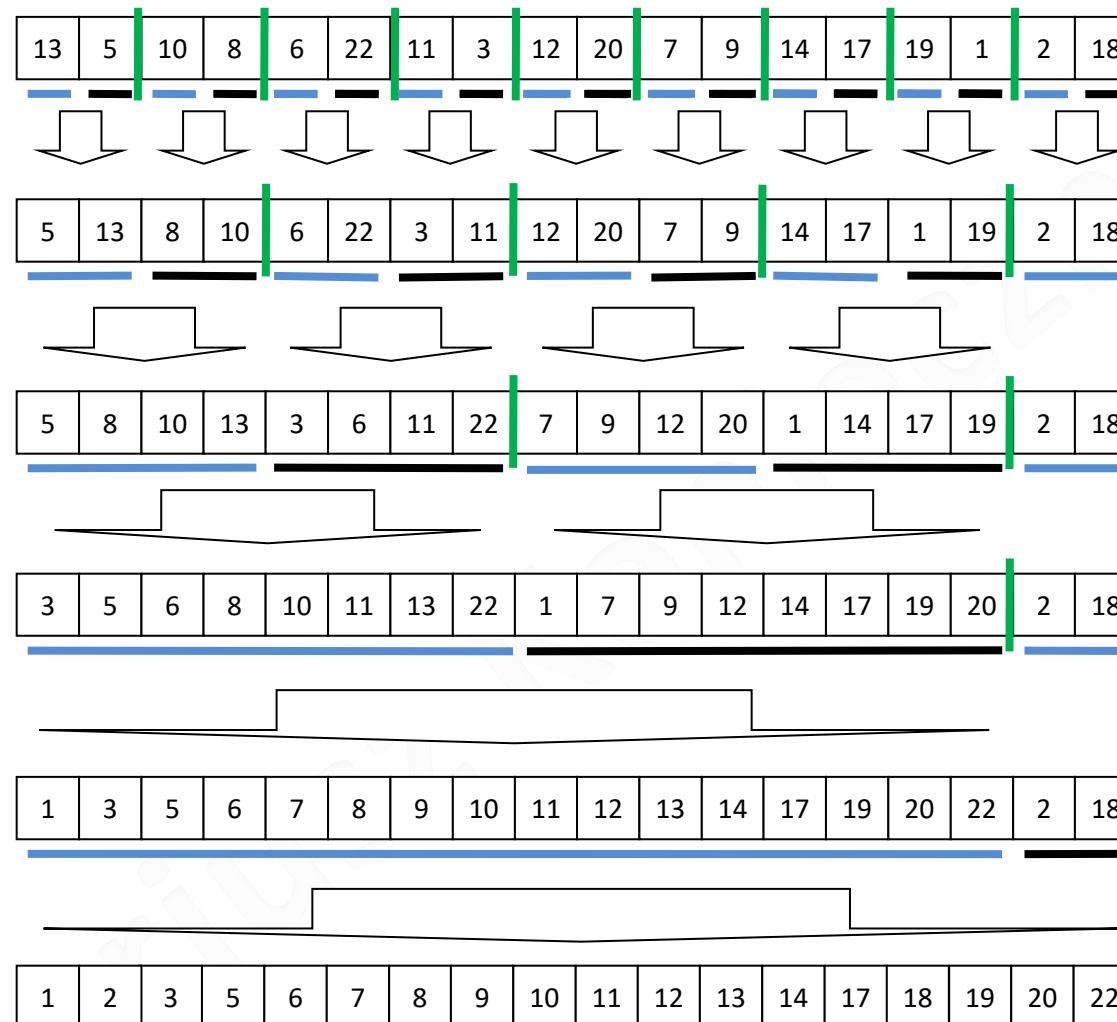
Sortowanie przez scalanie – kod 2/2

```
@SuppressWarnings("unchecked")
private IList<T> merge(IList<T> left, IList<T> right) {
    // mimo wszystko musimy się zdecydować na konkretną implementację listy
    IList<T> result = (IList<T>) new ArrayList<Object>();
    Iterator<T> l = left.iterator();
    Iterator<T> r = right.iterator();
    T elemL=null, elemR=null;
    // musimy opóźnić wychodzenie z pętli do czasu dodania do wyniku
    // ostatniego elementu jednego z ciągów
    boolean contL, contR;
    if(contL=l.hasNext()) elemL=l.next();
    if(contR=r.hasNext()) elemR=r.next();
    while (contL && contR) {
        if (_comparator.compare(elemL, elemR) <= 0) {
            result.add(elemL);
            if(contL=l.hasNext()) elemL=l.next();
            else result.add(elemR); } //już odczytany element drugiej listy do wyniku
        else {
            result.add(elemR);
            if(contR=r.hasNext()) elemR=r.next();
            else result.add(elemL); } //już odczytany element pierwszej listy do wyniku
    }
    while(l.hasNext()) result.add(l.next());
    while(r.hasNext()) result.add(r.next());
    return result; }
```

Sortowanie przez scalanie - analiza

- Złożoność obliczeniowa:
 - Pesymistyczna, średnia: $O(n \log n)$
- Złożoność dodatkowej pamięci: $\Omega(n)$
- Stabilny
- Łatwy do zrównoleglania
- Można zaimplementować ten kod w wersji iteracyjnej:
 - Ze stosem list/tablic:
 - Przeróbka wersji rekurencyjnej
 - Z kolejką list/tablic:
 - Do kolejki najpierw trafiają listy jednoelementowe powstałe z każdego elementu wejściowej listy.
 - Wyciągamy z kolejki dwie listy, scalamy, wynik na koniec kolejki
 - Bez stosów/kolejek dla list o dostępie swobodnym:
 - Bez dokonywania podziału traktujemy fragmenty listy jako podlisty o długości 1, potem 2, 4, 8, 16 itd. analogicznie jak dla rozwiązania z kolejką.

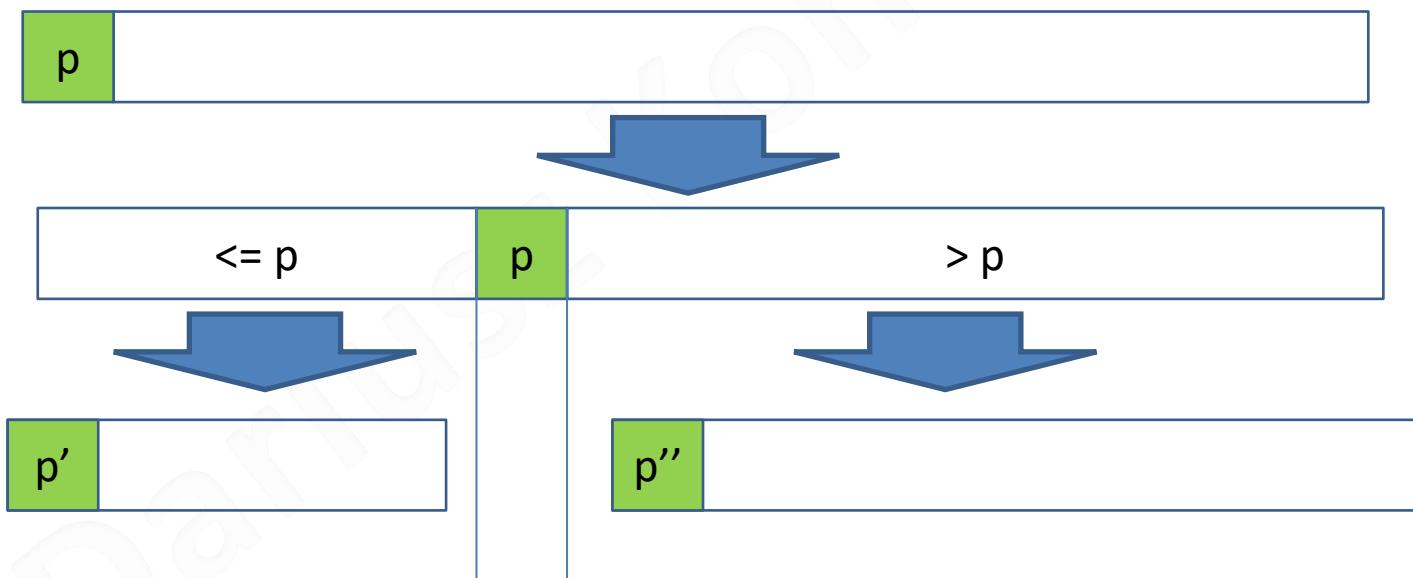
Sortowanie przez scalanie iteracyjne



Programowanie
dynamiczne

Sortowanie szybkie

- ang. **quicksort**
- Idea: Z tablicy wybiera się element rozdzielający (ang. pivot). Do jednej tablicy wstawia się elementy mniejsze równe pivotowi, do drugiej – większe od pivota. Element rozdzielający wstawiamy pomiędzy tabele i jest to jego ostateczne miejsce. Z każdą tablicą rekurencyjnie robimy to samo. Oczywiście tablice jednoelementowe są posortowane i następuje powrót z rekurencji.
- Podział na te dwie tablice można zrobić „w miejscu” w czasie $O(n)$.



Quicksort - pseudokod

```
// dla tablicy od pozycji p do pozycji r-1
quicksort(A,p,r)
{
    // jeżeli podtablica posiada więcej niż 1 element
    if(p<r-1)
    {
        // używając wybranego pivota podzieli tablicę na dwie części:
        // lewą z z elementami mniejszymi lub równymi pivotowi
        // i prawą z elementami większymi
        // q - pozycja podziału
        // i miejsce pivota po podziale
        q=partitioning(A,p,r);
        // posortuj lewą stronę w miejscu
        quicksort(A,p,q);
        // posortują prawą stronę w miejscu
        quicksort(A,q+1,r);
        // teraz część od p do r jest już posortowana
    }
}
```

45

Pivot w danym kroku

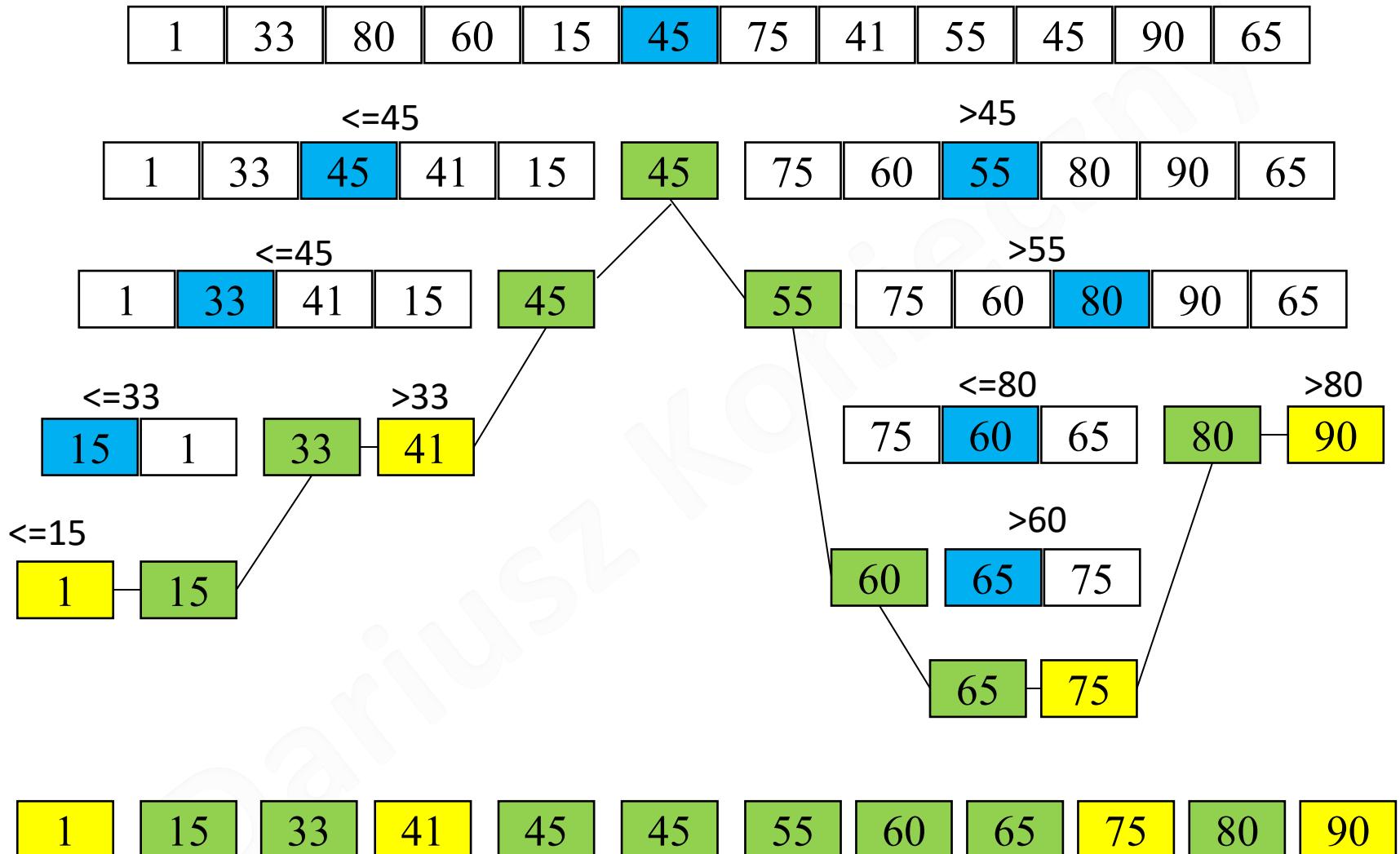
41

Podtablica jednoelementowa

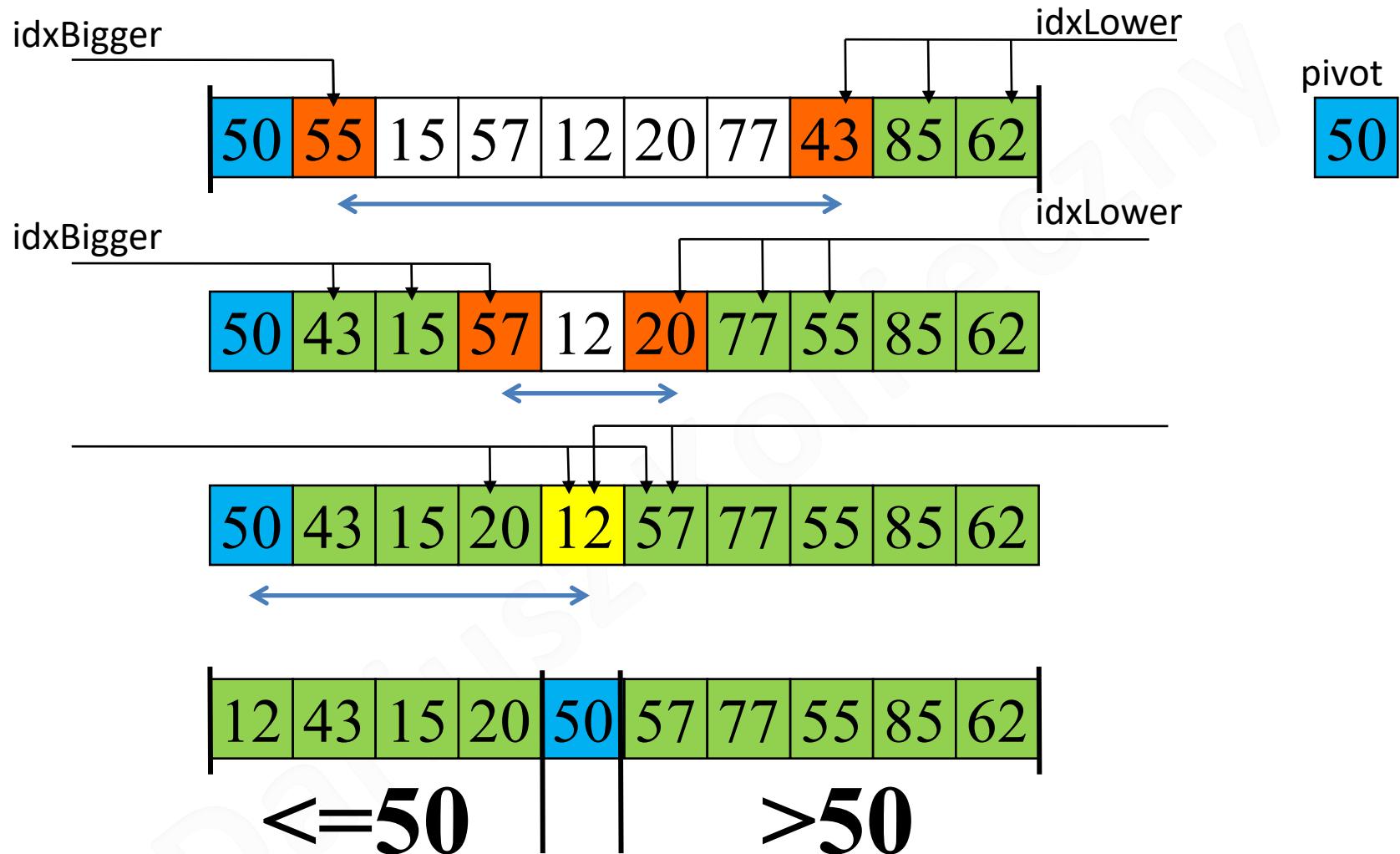
33

Pivot z poprzedniego kroku już na finalnej pozycji

Kroki w quicksort



Podział w quicksort



Quicksort – kod 1/2

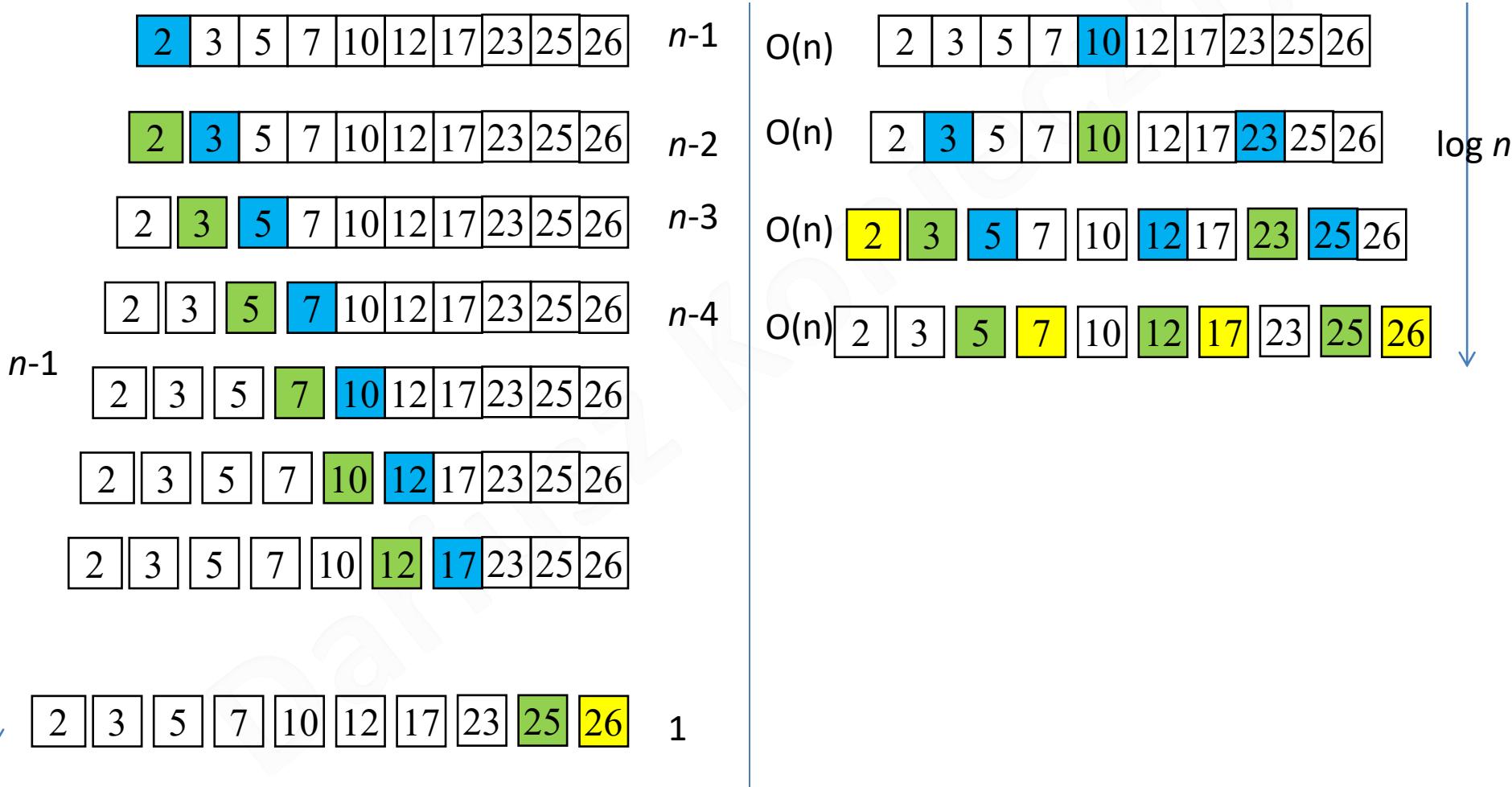
```
public IList<T> sort(IList<T> list) {  
    quicksort(list, 0, list.size());  
    return list;  
}  
private void quicksort(IList<T> list, int startIndex, int endIndex) {  
if (endIndex - startIndex > 1) {  
    int partition = partition(list, startIndex, endIndex);  
    quicksort(list, startIndex, partition );  
    quicksort(list, partition + 1, endIndex);}  
private int partition(IList<T> list, int nFrom, int nTo) {  
    //jako element dzielący bierzemy losowy  
    int rnd=nFrom+random.nextInt(nTo-nFrom);  
    swap(list,nFrom,rnd);  
    T value=list.get(nFrom);  
    int idxBigger=nFrom+1, idxLower=nTo-1;  
    do{  
        while(idxBigger<=idxLower && _comparator.compare(list.get(idxBigger),value)<=0)  
            idxBigger++;  
        while(_comparator.compare(list.get(idxLower),value)>0)  
            idxLower--;  
        if(idxBigger<idxLower)  
            swap(list,idxBigger,idxLower);  
    }while(idxBigger<idxLower);  
    swap(list,idxLower,nFrom);  
    return idxLower;  
}  
private void swap(IList<T> list, int left, int right) {  
    if (left != right) {  
        T temp = list.get(left);  
        list.set(left, list.get(right));  
        list.set(right, temp);}  
}
```

Quicksort – kod 2/2

```
// wynikiem jest posortowana oryginalna lista
public IList<T> sort(IList<T> list) {
    quicksort(list, 0, list.size() - 1);
    return list;
}
private void quicksort(IList<T> list, int startIndex, int endIndex) {
    if (endIndex > startIndex) {
        int partition = partition(list, startIndex, endIndex);
        quicksort(list, startIndex, partition );
        quicksort(list, partition + 1, endIndex);}
}
// podział według Lomuto
private int partition(IList<T> list, int left, int right) {
    //jako element dzielący bierzemy ostatni
    T value=list.get(right);
    int i=left-1;
    while (left <= right){
        if( _comparator.compare(list.get(left), value) <= 0)
            swap(list, ++i,left);
        ++left;}
    return i<right ? i :i-1;
}
private void swap(IList<T> list, int left, int right) {
    if (left != right) {
        T temp = list.get(left);
        list.set(left, list.get(right));
        list.set(right, temp);}
}
```

Quicksort – analiza 1/2

- Podział zależy od pivota i rozkładu liczb
- Najlepszy i najgorszy przypadek



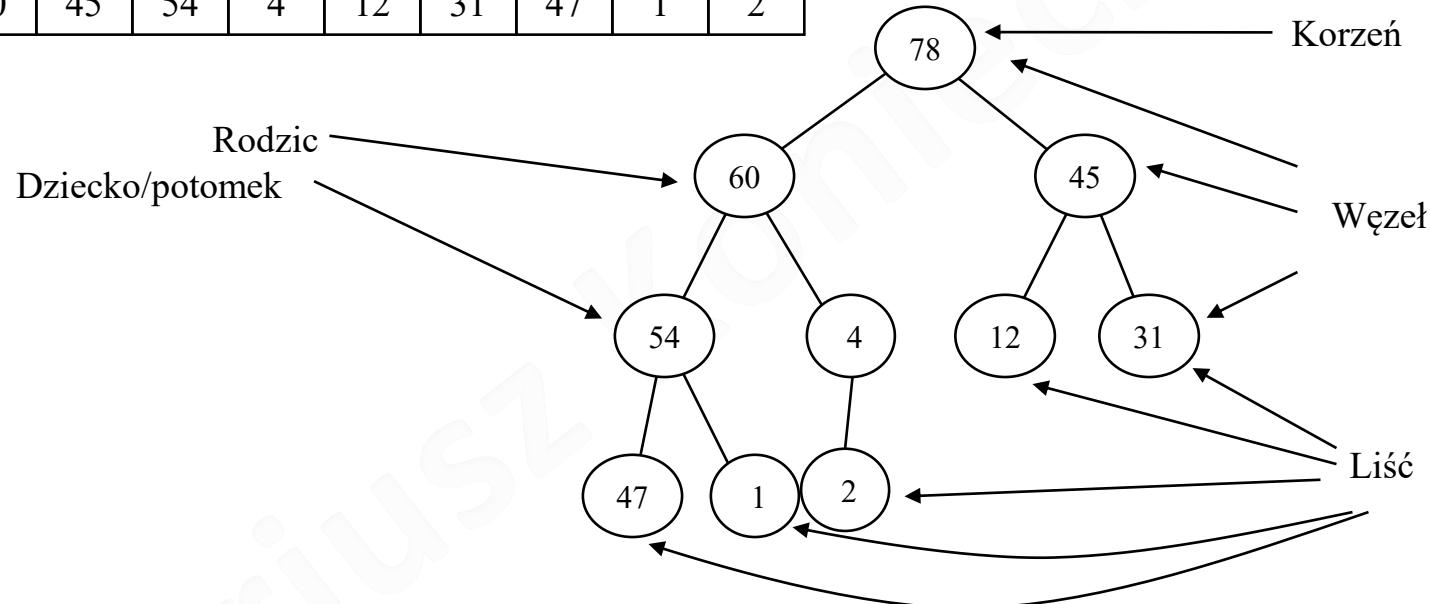
QuickSort – analiza 2/2

- Pesymistyczna złożoność – $O(n^2)$
- W literaturze można znaleźć dowód, że średnia oczekiwana złożoność czasowa wynosi $O(n \log n)$.
- Teoretycznie w miejscu, ale potrzeba miejsca na stosie wywołań funkcji rzędu $\Omega(\log n)$.
- Niestabilny.
- Wybór pivota:
 - Niepoprawne: Pierwszy element, środkowy element, ostatni element
 - Dobre: losowy element
 - Dla dużych n :
 - wybrać losowo 3 elementy i „średni” jest pivotem
 - wybrać k ($k << n$) losowych elementów, posortować algorytmem prostym, wybrać medianę posortowanego ciągu
- Dla odpowiednio małych n zaprzestać rekurencji na rzecz algorytmu prostego.
- Usprawnić dla ciągów równych wartości lub wielu równych.

Sortowanie stogowe

- Inaczej: sortowanie przez kopcowanie (ang. heapsort)
- Kopiec jest strukturą widzianą jako drzewo prawie pełne:
 - Wszystkie poziomy drzewa są wypełnione (z wyjątkiem ewentualnie najniższego poziomu)
 - Na ostatnim poziomie drzewo jest spójnie wypełnione od lewej
- W kopcu wartość w węźle nie może przekraczać wartości w jego rodzicu.

0	1	2	3	4	5	6	7	8	9
78	60	45	54	4	12	31	47	1	2



Kopiec to tablica, gdzie:

1) Węzeł z indeksem i posiada dwójkę potomków z indeksami $2*i+1$ oraz $2*i+2$

2) Węzeł z indeksem i ma rodzica z indeksem $\left\lfloor \frac{i-1}{2} \right\rfloor$

Tworzenie kopca

- Z dowolnej tablicy zrobić poprawny kopiec.
- Można wykorzystać oryginalną tablicę.
- Idea polega na naprawianiu kopca od dołu (pierwszego poziomu za liśćmi).
- Analizuje się węzłów oraz jego potomków. Gdy okazuje się, że rodzic nie posiada największej z wartości – następuje zamiana z większą wartością potomka. Zamiana ta może złamać zasadę na niższym poziomie, więc rekurencyjnie naprawić trzeba ten fragment kopca nieraz aż do liścia.
- Ponieważ kopiec ma wysokość $\log n$, zatem dla każdego z n węzłów trzeba wykonać maksymalnie $\log n$ kroków, zatem w tworzenie kopca wymaga $O(n \log n)$ czasu.
 - Dokładniejsza analiza algorytmu pozwala stwierdzić, że metoda budowania kopca wykonywana jest w czasie $O(n)$.



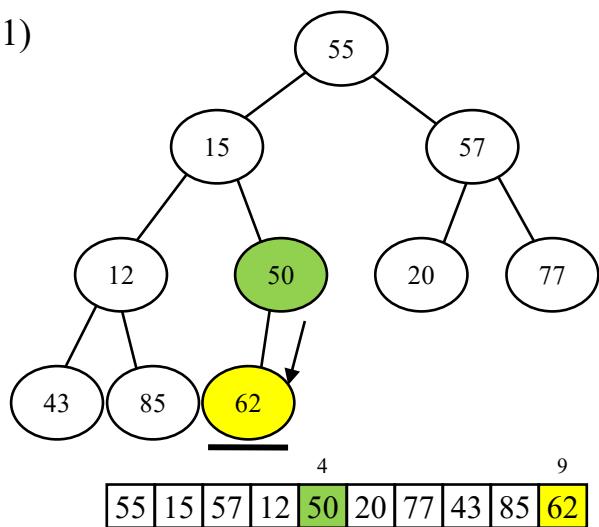
badany węzeł



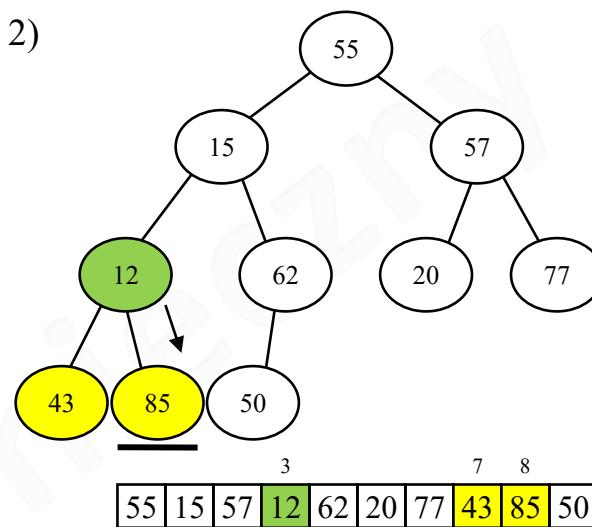
potomek badanego węzła

Tworzenie kopca – przykład 1/2

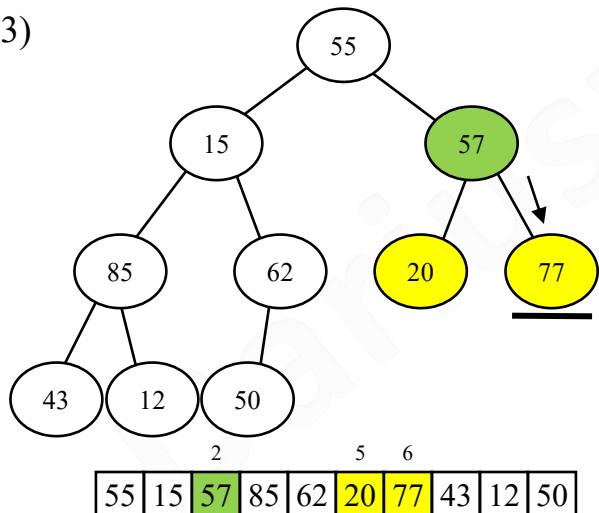
1)



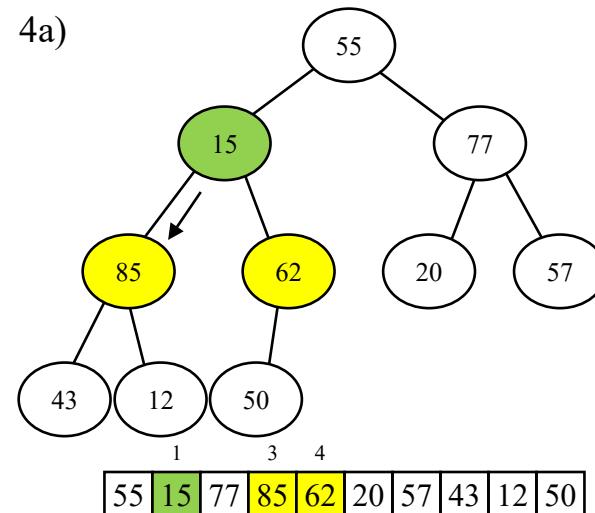
2)



3)

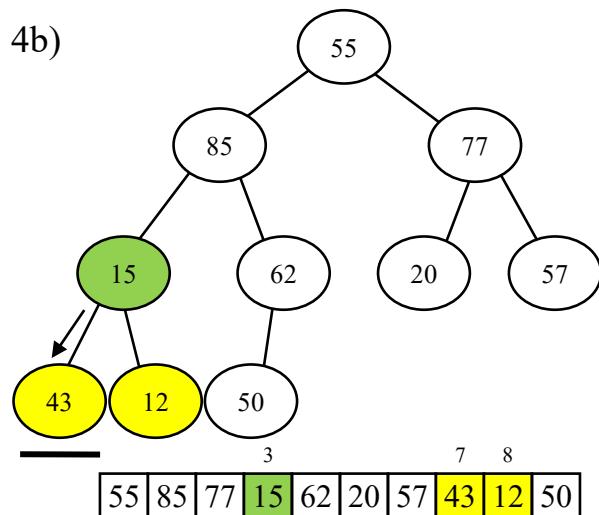


4a)

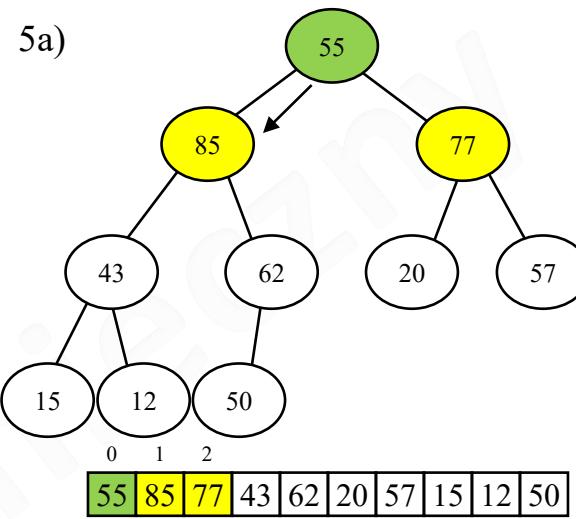


Tworzenie kopca – przykład 2/2

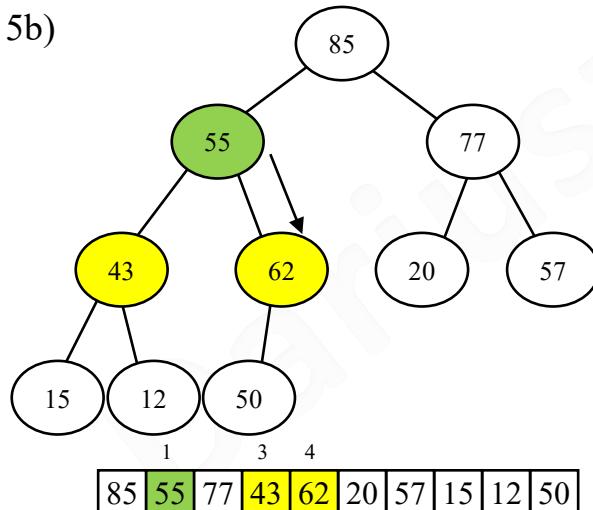
4b)



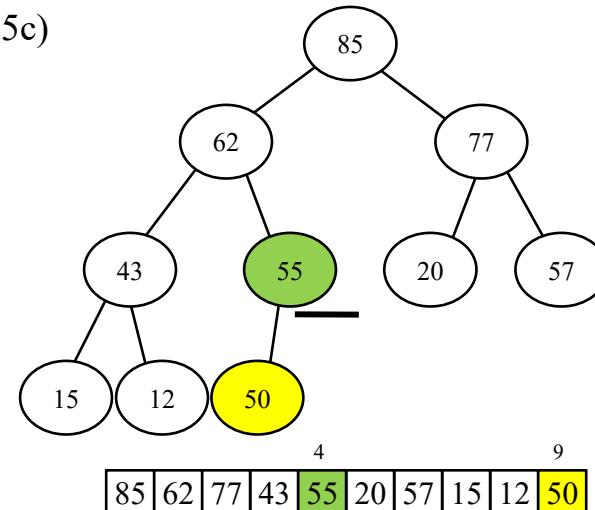
5a)



5b)



5c)



Tworzenie kopca - kod

```
/** założenie: left != right */
private void swap(IList<T> list, int left, int right) {
    T temp = list.get(left);
    list.set(left, list.get(right));
    list.set(right, temp);
}

public void sink(IList<T> heap, int idx, int n) {
    int idxOfBigger=2*idx+1;
    if(idxOfBigger<n) {
        if(idxOfBigger+1<n &&
            _comparator.compare(heap.get(idxOfBigger), heap.get(idxOfBigger+1))<0)
            idxOfBigger++;
        if(_comparator.compare(heap.get(idx), heap.get(idxOfBigger))<0) {
            swap(heap, idx, idxOfBigger);
            sink(heap, idxOfBigger, n);
        }
    }
}

void heapAdjustment(IList<T> heap, int n)
{
    for(int i=(n-1)/2; i>=0; i--)
        sink(heap, i, n);
}
```

Sortowanie przez kopcowanie

- 1) Stworzenie kopca
- 2) Kopiec -> tablica posortowana: zamiana korzenia z ostatnim elementem. Po zamianie ostatni element tablicy nie należy już do kopca. Natomiast trzeba naprawić kopiec od korzenia. Kolejne kroki wyglądają analogicznie.
- Dla każdego elementu należy zrobić $O(\log n)$ kroków, czyli w sumie $O(n \log n)$ kroków.



Element ulegający wymianie



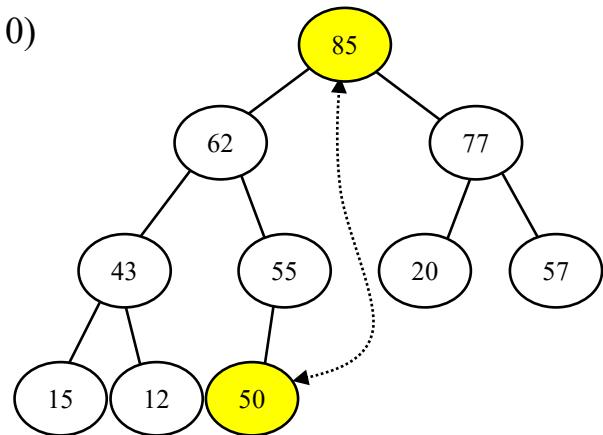
Element posortowany,
nie należy już do kopca



Elementy przesuwane przy naprawianiu kopca

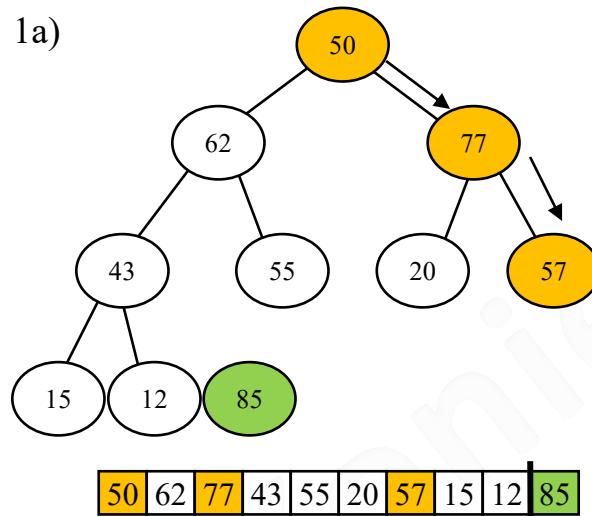
Heapsort – przykład

0)



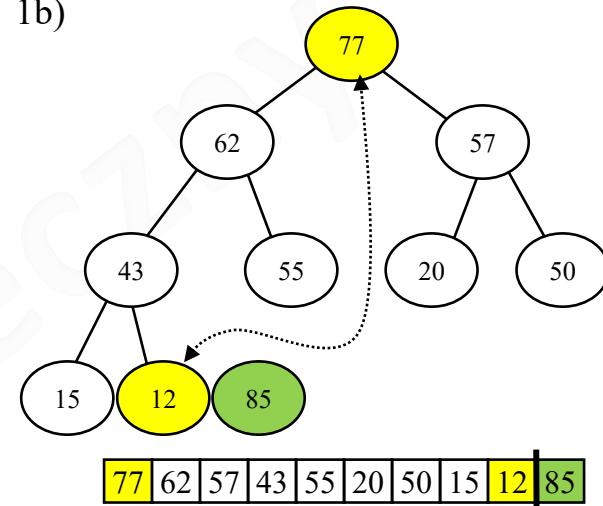
85 | 62 | 77 | 43 | 55 | 20 | 57 | 15 | 12 | 50

1a)



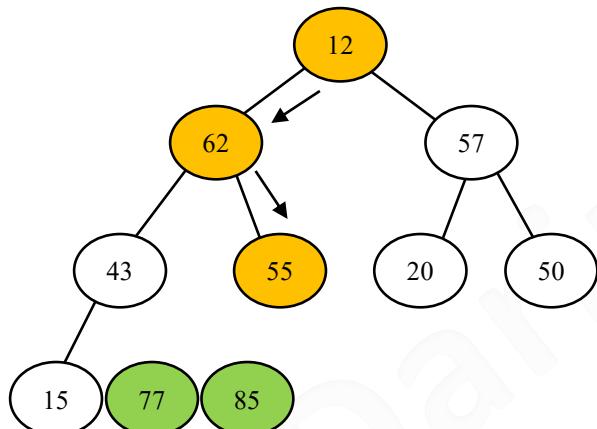
50 | 62 | 77 | 43 | 55 | 20 | 57 | 15 | 12 | 85

1b)



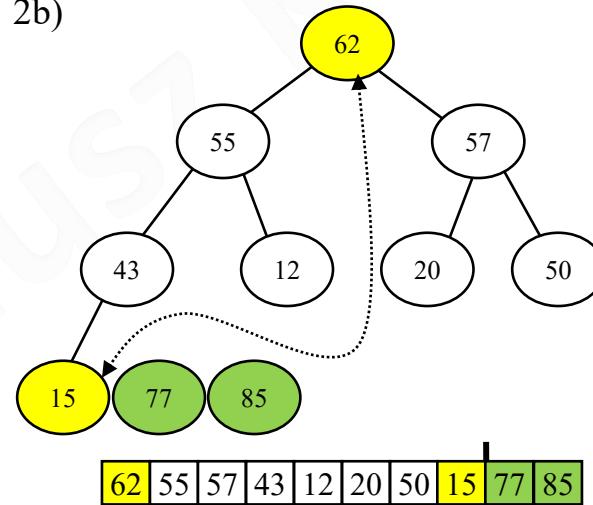
77 | 62 | 57 | 43 | 55 | 20 | 50 | 15 | 12 | 85

2b)



12 | 62 | 57 | 43 | 55 | 20 | 50 | 15 | 77 | 85

3a)



15 | 55 | 57 | 43 | 12 | 20 | 50 | 62 | 77 | 85

Heapsort – kod, właściwości

```
// wynikiem jest oryginalna lista/tablica
@Override
public IList<T> sort(IList<T> list) {
    heapsort(list, list.size());
    return list;
}

private void heapsort(IList<T> heap, int n) {
    heapAdjustment(heap, n);
    for(int i=n-1;i>0;i--) {
        swap(heap,i,0);
        sink(heap,0,i);
    }
}
```

- Złożoność czasowa:
 - Średnia, pesymistyczna: $O(n \log n)$
 - Jednak wolniejszy od quicksort i mergesort
 - Dla danych równych działa w czasie $O(n)$
- Złożoność pamięciowa: $O(1)$ – nietrudno napisać wersję iteracyjną funkcji `sink()`.
- Właściwości:
 - W miejscu
 - Niestabilny
 - Kopiec może być użyty do kolejek priorytetowych (patrz. kolejne wykłady)

Przykładowe czasy wykonania

- $n=10.000.000$ losowych elementów typu Integer
- Czasy wykonania:

MergeSort	:	18826 milliseconds
IterMergeSort	:	11482 milliseconds
QuickSort	:	11841 milliseconds
QuickSortBetter	:	11939 milliseconds
HeapSort	:	40564 milliseconds

MergeSort	:	18995 milliseconds
IterMergeSort	:	15083 milliseconds
QuickSort	:	12817 milliseconds
QuickSortBetter	:	10354 milliseconds
HeapSort	:	31106 milliseconds

- $n=10.000$ losowych elementów typu Integer
- Czasy wykonania:

BubbleSort	:	640 milliseconds
InsertSort	:	453 milliseconds
SelectSort	:	312 milliseconds
MergeSort	:	32 milliseconds
Iter.MergeSort	:	16 milliseconds
QuickSort	:	32 milliseconds
QuickSortBetter	:	31 milliseconds
HeapSort	:	31 milliseconds

- $n=50.000$ losowych elementów typu Integer
- Czasy wykonania:

BubbleSort	:	20059 milliseconds
InsertSort	:	9970 milliseconds
SelectSort	:	8860 milliseconds
MergeSort	:	156 milliseconds
Iter.MergeSort	:	109 milliseconds
QuickSort	:	156 milliseconds
QuickSortBetter	:	78 milliseconds
HeapSort	:	110 milliseconds

Inne sortowania

- Sortowania bez porównywania elementów ze sobą:
 - najczęściej dla kluczy, które są liczbami
 - Można je traktować jako liczby naturalne z pewnego zakresu
 - Lub wręcz muszą być liczbami w systemie pozycyjnym
 - Lub znamy rozkład/przedział liczb-kluczy
- Złożoność obliczeniowa (przy odpowiednich założeniach): $O(n)$

Sortowanie przez zliczanie

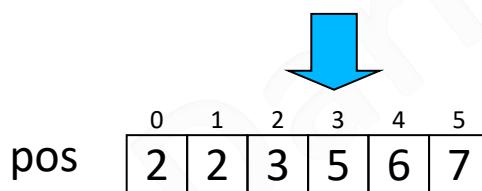
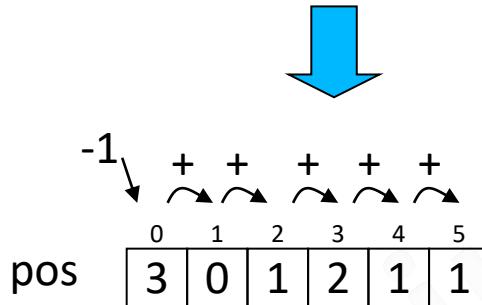
- ang. counting sort
- Niech każda z n wejściowych danych będzie liczbą z zakresu od 0 do k , dla pewnego ustalonego k .
- Jeśli $k=O(n)$ to sortowanie działa w czasie $O(n)$.
- Idea: wyznaczyć dla każdego wejściowego elementu x liczbę elementów mniejszych od x . Ta informacja może zostać użyta do wstawienia elementu x bezpośrednio na właściwą pozycję tablicy wynikowej
- Złożoność czasowa: $\Theta(n+k)$, czyli $O(n)$
- Złożoność dodatkowej pamięci: $\Theta(n+k)$

$n=8, k=5$

arr	0	4	0	2	3	3	0	5
-----	---	---	---	---	---	---	---	---

result	0	1	2	3	4	5	6	7
			0			3		5

pos	0	1	2	3	4	5	6	7
	1	2	3	4	6	6		



result	0	1	2	3	4	5	6	7
			0			3		5

pos	0	1	2	3	6	6		
	0	1	2	3	6	6		

result	0	1	2	3	4	5	6	7
	0	0	2	3	3	4	5	

pos	0	1	2	3	4	5	6	7
	-1	1	3	3	5	6		

Sortowanie przez zliczanie - kod

```
public static void countingSort(int arr[], int k)
{
    k++;
    int n=arr.length;
    int pos=new int[k];
    int result=new int[n];
    int i,j;
    for(i=0;i<k;i++)
        pos[i]=0;
    for(j=0;j<n;j++)
        pos[arr[j]]++;
    pos[0]--;
    for(i=1;i<k;i++)
        pos[i]+=pos[i-1];
    for(j=n-1;j>=0;j--)
    {
        result[pos[arr[j]]]=arr[j];
        pos[arr[j]]--;
    }
    for(j=0;j<n;j++)
        arr[j]=result[j];
}
```

Sortowanie pozycyjne

- ang. radix sort
- dla liczb nieujemnych całkowitych
- pseudokod:

```
RadixSort( $A, d$ ) // $d$  – liczba cyfr
```

```
for  $i$  from 0 to  $d-1$ 
```

użyj stabilnego sortowania A względem cyfry i

293	231	3	3
495	22	22	22
329	293	329	195
248	3	231	231
22	495	235	235
3	195	248	248
256	235	256	256
195	256	293	293
231	248	495	329
235	329	195	495

Można użyć sortowania
przez zliczanie!

Złożoność: $\Omega(d*(n+k))$

Jeśli d jest stałe: $O(n)$

Sortowanie kubełkowe

- ang. bucket sort
- Założenia:
 - Znamy zakres wartości klucza liczbowego (może być zmiennopozycyjny) - X
 - Wartości klucza są równomiernie rozłożone
- Idea:
 - Podział zakresu na m równych części (kubełki)
 - Przydzielanie danych wg klucza do odpowiednich kubełków w czasie $O(1)$
 - Posortowanie kubełków
- Złożoność czasowa (przy pewnych założeniach): $O(n)$

```
for (i=0; i<n; i++) { // pseudokod wstawiania do kubełka/kolejki
    pr=floor (el[i]/X*M)
    DoKolejki(Q[pr],el[i]) } // np. do kolejki priorytetowej
```

[0, 25)

[25, 50)

[50, 75)

[75, 100)

23	31	64	16	89	70	41	79	15	66	99	89	14	48	68	24	98	50	70	10	59	40
----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

itd.

23	16	15	14	24	10
----	----	----	----	----	----

31	41	48	40
----	----	----	----

64	70	66	68	50	70	59
----	----	----	----	----	----	----

89	79	99	89	98
----	----	----	----	----

Porównanie prosty algorytmów

Algorytm	Stabilność	Porównań	Podstawień	Złożoność czasowa
insertSort - tablica	Tak	$O(n \log n)$	$O(n^2)$	$O(n^2)$
insertSort – lista wiązana	Tak	$O(n \log n)$	$O(n^2)$	$O(n^2)$
selectSort - tablica	Tak	$O(n^2)$	$O(n)$	$O(n^2)$
selectSort – lista wiązana	Tak	$O(n^2)$	$O(n)$	$O(n^2)$
bubbleSort – tablica	Tak	$O(n^2)$	$O(n^2)$	$O(n^2)$
bubbleSort – lista wiązana	Tak	$O(n^2)$	$O(n^2)$	$O(n^2)$

Porównanie efektywnych algorytmów

Algorytm	Stabilność	Złożoność czasowa	Złożoność pamięciowa
mergeSort - tablica	Tak	$O(n \log n)$	$O(n)$ Operacja łączenia
mergeSort – lista wiązana	Tak	$O(n \log n)$	$O(n)$ kolejka
quickSort - tablica	Nie	$O(n \log n)$	$O(\log n)$ rekurencja
quickSort – lista wiązana	Nie	$O(n \log n)$	$O(\log n)$ rekurencja
heapSort - tablica	Nie	$O(n \log n)$	$O(1)$
heapSort – lista wiązana	-	-	-

Porównanie algorytmów specjalnych

Algorytm	Stabilność	Złożoność czasowa	Złożoność pamięciowa
Counting sort - tablica	Tak	$O(n+k)$	$O(n+k)$
Counting sort – lista wiązana	Tak	$O(n)$	$O(k)$
RadixSort – tablica	Tak	$O(d*n)$	$O(k)$
RadixSort – linked list	Tak	$O(d*n)$	$O(k)$
BucketSort - tablica	Tak	$O(n)$	$O(n)$
BucketSort – lista wiązana	Tak	$O(n)$	$O(k)$

Sortowania w bibliotece Javy

- Klasa `Arrays` z metodą `sort (...)` dla tablic:
 - Tylko tablica – sortuje całą tablicę
 - Tablica i zakres indeksów – sortuje fragment tablicy
 - Używa quicksort-a
- Klasa `Collections` z metodą `sort (...)` dla list:
 - Używa mergesort-a
- Powyższe metody używają albo wbudowanego porównywania albo komparatora naturalnego, można je wywołać z dodatkowym parametrem – komparatorem.
- Istnieją klasy i metody przekształcające dowolne kolekcje na tablice lub listę.
- Dla innych kolekcji (np. mapy, tablice mieszające) istnieją specjalne metody sortującego.
- Od Javy 8.0 można używać wyrażeń lambda do krótkiego zapisu komparatora
- Od Javy 8.0 w przetwarzaniu strumieniowym (leniwym) można zamiast złożonego komparatora używać metody klasy `Comparator` `comparing (...)` oraz `thenComparing (...)`

Algorytmy i struktury danych – W06

Wyszukiwanie liniowe i binarne

Kolejki priorytetowe

Tablice mieszające

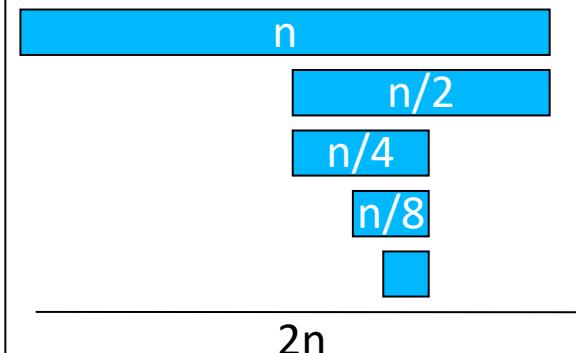
Zawartość

- Szukanie mediany i i-tego elementu
- Wyszukiwanie liniowe
- Wyszukiwanie binarne
- Kolejki priorytetowe:
 - Za pomocą list wiązanych
 - Za pomocą tablic
 - Za pomocą kopca
- Tablice mieszające (haszujące)
 - Adresowanie otwarte
 - Tablica list

Szukanie mediany lub i-tego elementu

- Szukanie minimum/maksimum to fragment algorytmu selectSort o złożoności $O(n)$
- Jak znaleźć medianę (środkową wartość) lub i-ty element posortowanego ciągu. Najpierw sortując? Zatem złożoność: $O(n \log n)$
- Można to wykonać szybciej stosując ideę z quicksort.

```
// A – tablica, p, r – zakres poszukiwania
// i - szukana pozycja
RANDOMIZED-SELECT(A, p, r, i)
if p = r
    return A[p]
q=RANDOMIZED-PARTITION(A, p, r)
k=q - p + 1
if i = k // wartość pivota jest szukaną wartością
    return A[q]
else if i < k
    return RANDOMIZED-SELECT(A, p, q - 1, i)
else
    return RANDOMIZED-SELECT(A, q + 1, r, i - k)
```



Średnia złożoność: $O(n)$

Wyszukiwanie liniowe (WL)

- W liniowej kolekcji nieuporządkowanej w celu wyszukania elementu o danej zawartości należy sprawdzić każdy element poruszając się od początku do końca kolekcji. Jest to **wyszukiwanie liniowe**.
- Złożoność pesymistyczna, średnia: $O(n)$
- W celu porównania elementów z poszukiwanym wzorcem można się posłużyć:
 - metodą `equals()` - ale może być tylko jedna taka metoda
 - komparatorem

```
public class Searching<T> {  
    /** zwraca pozycję znalezioneego elementu jeśli elementu nie ma zwraca -1 */  
    public int linearSearch(IList<T> list, Comparator<T> comp, T what) {  
        int pos=0;  
        for(T elem:list) {  
            if(comp.compare(what,elem)==0)  
                return pos;  
            else  
                pos++;  
        }  
        return -1;  
    }  
}
```

Wyszukiwanie binarne (WB)

- Jeśli mamy kolekcję liniową o **dostępie swobodnym posortowaną** (wg wartości, którą szukamy), możemy użyć **szukania binarnego**.
- Szukaną wartość porównujemy ze środkowym elementem. Jeśli wartość szukana jest mniejsza – znajduje się na lewo od pozycji środkowej. Jeśli większa – na prawo. Jeśli równa – koniec szukania. Z częścią, w której może być szukana wartość postępujemy dokładnie tak samo jak z początkową kolekcją. Jeśli część kolekcji stanie się pusta – wartości w ogóle nie ma w kolekcji
- Złożoność pesymistyczna i średnia: $O(\log n)$

```
public int binarySearch(IList<T> list, Comparator<T> comp, T what ) {  
    int left=0;  
    int right=list.size()-1;  
    int middle;  
    while(left<=right) {  
        middle=(left+right)/2;  
        int compValue=comp.compare(what,list.get(middle));  
        if(compValue==0)  
            return middle;  
        if(compValue<0)  
            right=middle-1;  
        else  
            left=middle+1;  
    }  
    return -1;  
}
```

Posortowana lista wiązana

- Struktura liniowa bez dostępu swobodnego a wyszukiwanie binarne:
 - Implementacja przedstawiona spowoduje złożoność $O(n \log n)$ ze względu na operacje get
 - Używając iteratorów dla list można poruszać się do przodu i do tyłu zamiast używać operacji get, co ograniczy złożoność poruszania się do $O(n)$, chociaż może być nawet dwa razy więcej kroków niż przy przeszukiwaniu liniowym.
 - Ograniczona zostanie liczba porównań do $O(\log n)$.
- Wniosek: poprawny algorytm wyszukiwania binarnego dla list ma sens, gdy czas porównywania jest zdecydowanie większy niż czas przesuwania się po liście.

Kolejki priorytetowe (KP)

- Struktura danych, w której potrzebujemy dwóch podstawowych operacji:
 - Dodaj element do kolekcji
 - Pobierz i usuń element kolekcji o najwyższym priorytecie.
- Dodatkowo może być oczekiwana operacja:
 - Pobierz największy element kolekcji (bez usuwania)
- Ponieważ nazwy operacji są takie same jak dla kolejki FIFO (choć mają inne znaczenie), można użyć interfejsu `IQueue` jako interfejsu dla kolejek priorytetowych.

Kolejki priorytetowe – listy wiązane

- Realizacja kolejki priorytetowej za pomocą listy wiązanej:
 - Nieuporządkowanej:
 - Wstawianie elementu (na koniec lub na początek): $O(1)$
 - Pobranie i usunięcie elementu – najpierw trzeba znaleźć element za pomocą wyszukiwania liniowego i potem „wyjąć” z listy: $O(n)+O(1) = O(n)$
 - Uporządkowanej w/g priorytetu:
 - Wstawienie elementu (na podstawie priorytetu): $O(n)$
 - Pobranie i usunięcie elementu z początku listy: $O(1)$

KP – listy wiązane - kod

```
public class UnsortedListPriorityQueue<T> implements IQueue<T> {
    private final TwoWayCycledListWithSentinel<T> _list;
    private final Comparator<T> _comparator;

    public UnsortedListPriorityQueue(Comparator<T> comp) {
        _comparator=comp;
        _list=new TwoWayCycledListWithSentinel<T>();
    }

    public boolean isEmpty() {
        return _list.isEmpty();
    }

    public boolean isFull() {
        return false;
    }
    public int size() {
        return _list.size();
    }
    public T first() throws EmptyQueueException {
        if(_list.isEmpty())
            throw new EmptyQueueException();
        return _list.get(getIndexOfLargestElement());
    }
}
```

KP – listy wiązane - kod

```
public void enqueue(T elem) throws FullQueueException {
    _list.add(elem);
}

private int getIndexOfLargestElement() {
    if(_list.isEmpty())
        return -1;
    Iterator<T> iter=_list.iterator();
    int counter=0, maxPos=0;
    T value=iter.next();
    T elem=null;
    while(iter.hasNext()) {
        counter++;
        if(_comparator.compare(elem=iter.next(), value)>0) {
            maxPos=counter;
            value=elem;
        }
    }
    return maxPos;
}

public T dequeue() throws EmptyQueueException {
    if(_list.isEmpty())
        throw new EmptyQueueException();
    return _list.remove(getIndexOfLargestElement());
}
```

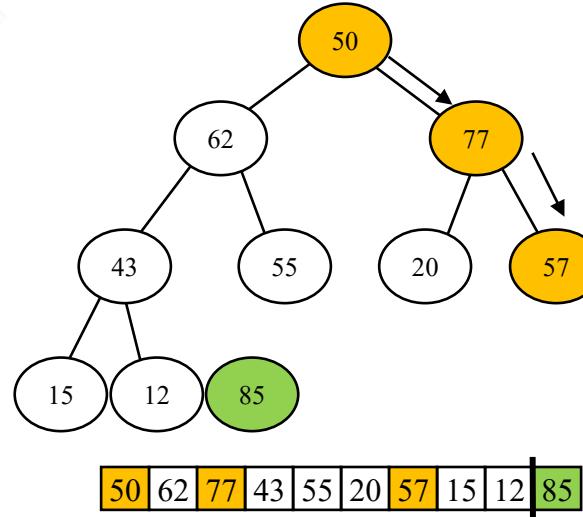
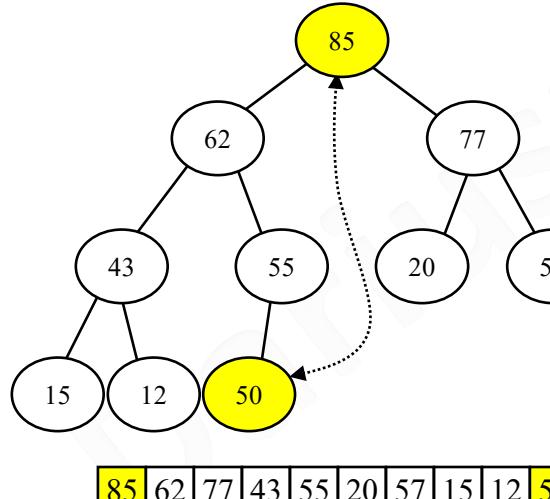
Kolejki priorytetowe - tablice

- Realizacja kolejki priorytetowej za pomocą tablicy:
 - Nieuporządkowanej:
 - Wstawianie elementu (na koniec), złożoność zamortyzowana: $O(1)$
 - Pobranie i usunięcie elementu – najpierw trzeba znaleźć element za pomocą wyszukiwania liniowego i potem przesunąć elementy w lewo w tablicy : $O(n)$
 - Uporządkowanej w/g priorytetu (**odwrotnie**):
 - Wstawienie elementu – znajdowanie pozycji za pomocą wyszukiwania binarnego, następnie przesunięcie elementów w prawo w tablicy: $O(\log n) + O(n) = O(n)$
 - Pobranie i usunięcie elementu – znajduje się **na końcu** zatem $O(1)$

Kolejki priorytetowe - kopiec

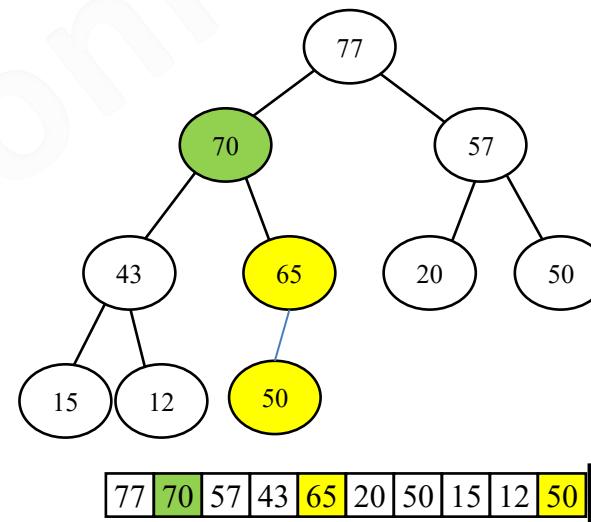
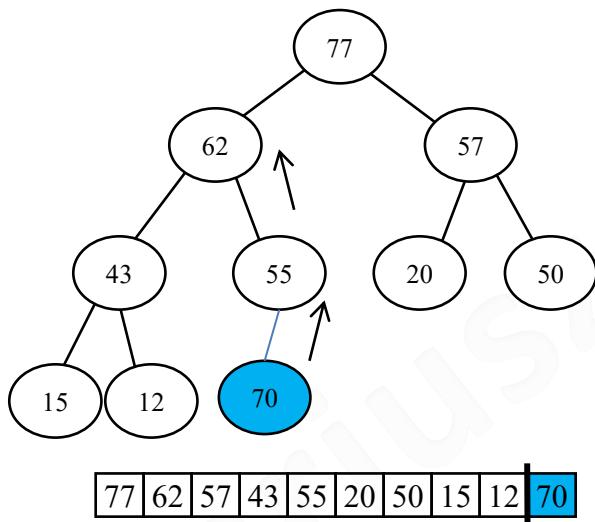
- Kopiec – struktura z poprzedniego wykładu – może być użyta do obsługi operacji kolejki priorytetowej:
 - Operacja pobrania elementu jest częścią jednego kroku algorytmu sortowania przez kopcowanie – gdy mamy poprawny kopiec należy zamienić korzeń z ostatnim elementem tablicy i naprawić kopiec: $O(\log n)$
 - Naprawa kopca to „zatopienie” elementu (sink)

```
swap(heap, n-1, 0);  
sink(heap, 0, n-1);
```



Kopiec – dodanie elementu

- Dodanie nowego elementu do kopca:
 - Dodajemy na kolejnej wolnej pozycji w tablicy
 - Traktujemy go jako nowy liść, który jednak może łamać zasadę poprawnego kopca.
 - Naprawiamy kopiec idąc od wstawionego liścia w górę (swim).
- Złożoność: $O(\log n)$



Kopiec jako KP – kod 1/2

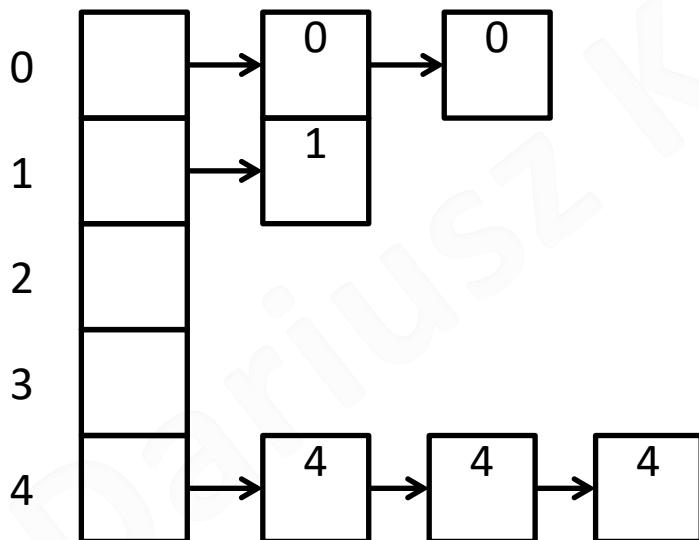
```
public class HeapPriorityQueue<T> implements IQueue<T> {
    private final IList<T> _list;
    private final Comparator<T> _comparator;
    public HeapPriorityQueue(Comparator<T> comparator) {
        _comparator = comparator;
        _list = new ArrayList<T>();
    }
    public void enqueue(T value) {
        _list.add(value);
        swim(_list.size() - 1);
    }
    public void clear() {
        _list.clear();
    }
    public int size() {
        return _list.size();
    }
    public boolean isEmpty() {
        return _list.isEmpty();
    }
    public boolean isFull() {
        return false;
    }
    public T first() throws EmptyQueueException {
        if (isEmpty()) throw new EmptyQueueException();
        return _list.get(0);
    }
    public T dequeue() throws EmptyQueueException {
        if (isEmpty()) throw new EmptyQueueException();
        T result = _list.get(0);
        if (_list.size() > 1) {
            _list.set(0, _list.get(_list.size() - 1));
            sink(0);
        }
        _list.remove(_list.size() - 1);
        return result;
    }
}
```

Kopiec jako KP – kod 2/2

```
private void swap(int index1, int index2) {  
    T temp = _list.get(index1);  
    _list.set(index1, _list.get(index2));  
    _list.set(index2, temp);  
}  
  
// wynoszenie elementu w górę, wersja iteracyjna  
private void swim(int index) {  
    int parent;  
    while(index != 0 &&  
        _comparator.compare(_list.get(index), _list.get(parent= (index - 1) / 2)) > 0){  
        swap(index, parent);  
        index=parent;  
    }  
}  
  
// opuszczanie elementu w dół stogu, wersja iteracyjna  
private void sink(int index) {  
    boolean isDone=false;  
    int child;  
    while(!isDone && (child=2*index+ 1 ) < _list.size()) {  
        if (child < _list.size()- 1 &&  
            _comparator.compare(_list.get(child), _list.get(child+1)) < 0)  
            ++child;  
        if (_comparator.compare(_list.get(index), _list.get(child)) < 0){  
            swap(index, child);  
            index=child;  
        }  
        else isDone=true;  
    }  
}
```

KP – inne implementacje

- Jeśli priorytetów jest niewiele, załóżmy k wartości, najprościej jest zrealizować KP jako tablicę (ew. listę) list dla **każdego priorytetu oddziennie**. Dla uproszczenia załóżmy, że są to liczby całkowite z zakresu od 0 do $k-1$
 - Wstawienie elementu dla takiej tablicy list to $O(1)$
 - Pobranie elementu – w najgorszym przypadku trzeba sprawdzać po kolej, która lista nie jest pusta – $O(k)=O(1)$



Porównanie złożoności realizacji KP

Realizacja	enqueue()	dequeue()	first()
Tablica nieuporządkowana	O(1)	O(n)	O(n)
Tablica uporządkowana	O(n)	O(1)	O(1)
Lista wiązana nieuporządkowana	O(1)	O(n)	O(n)
Lista wiązana uporządkowana	O(n)	O(1)	O(1)
Kopiec na tablicy	O(log n)	O(log n)	O(1)
Tablica list	O(1)	O(k)	O(k)

Słownik

- Słownik, to dynamiczna (najczęściej) struktura danych do pamiętania:
 - klucz lub
 - par klucz-wartość.
- Słownik zwany jest również tablicą asocjacyjną, tablicą skojarzeniową, tablicą indeksowaną kluczem, odwzorowaniem, mapą.
- Słownik musi umożliwiać dwie podstawowe operacje, które muszą być jak najbardziej efektywne:
 - Znalezienie elementu na podstawie klucza – najczęstsza operacja!
 - Dodanie elementu
- W słowniku kluczy zwraca jest tylko informacja logiczna, czy klucz jest, czy go nie ma. W słowniku par klucz-wartość: wartość powiązana z kluczem (lub **null**).
- Słownik statyczny ma tylko jedną operację:
 - znajdowanie elementu wg klucza.
- Słownik może umożliwiać również inne operacje (lub ich podzbiór):
 - Usuwanie elementu wg klucza
 - Przegląd słownika w sposób posortowany wg klucza
 - Łączenie słowników w nowy słownik

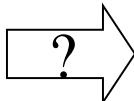
Słownik za pomocą tablic/list

- Lista nieuporządkowana:
 - **Znajdowanie elementu** – wyszukiwanie liniowe $O(n)$
 - Dodawanie elementu – na koniec $O(1)$
- Lista uporządkowana na liście wiązanej:
 - **Znajdowanie elementu**: wyszukiwanie liniowe $O(n)$, średnio połowa elementów zostanie porównana
 - Dodawanie elementu: wyszukiwanie liniowe $O(n)$
- Lista uporządkowana na tablicy:
 - **Znajdowanie elementu**: wyszukiwanie binarne $O(\log n)$.
 - Dodawanie elementu: $O(n)$ ze względu na przesunięcie elementów w tablicy.

Tablice mieszające (HT)

- Czy jest możliwe szybsze niż w czasie $O(\log n)$ znajdowanie klucza? Tak!
- Tablica mieszającą zwaną z angielskiego tablicą haszującą (ang. hash table).
- Idea:
 - Tablica o ustalonej długości może pod konkretnym indeksem zawierać element lub wartość „pustą” (np. `null`).
 - Z tablicą związana jest funkcja mieszająca (hash function), która dla każdego klucza zwraca pewną wartość całkowitą dodatnią.
 - Wstawianie elementu: wartość pozycji klucza (pary klucz-wartość) wyznacza się biorąc wartość funkcji mieszającej modulo rozmiar tablicy.
 - Znajdowanie elementu: klucz jest argumentem funkcji mieszającej, która zwraca pozycję w tablicy. Sprawdzamy czy w danej pozycji jest zapisany szukany klucz:
 - Jeśli TAK – klucz jest w zbiorze (lub zwracamy wartość z pary klucz-wartość)
 - Jeśli NIE – klucza nie ma w zbiorze (lub zwracamy wartość „pustą”)

Kowalski Jan
Adamska Jolanta
Nowak Piotr
Ciesielski Stanisław
Laskowik Darek
Plis Beata



0	
1	
2	
3	
4	
5	

HT - przykład

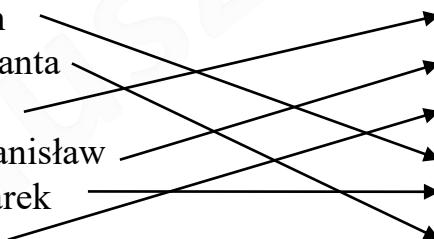
K – klucz (tutaj nazwisko)

$h(K) = K[0]$ – funkcją mieszającą jest kod ASCII pierwszej litery nazwiska
 $\text{pos}(K) = h(K) \bmod 6$ - generacja pozycji modulo długość tablicy

$$\begin{aligned}h(\text{,,Kowalski"}) &= (\text{,,K"}) \bmod 6 = 75 \bmod 6 = 3 \\h(\text{,,Adamska"}) &= (\text{,,A"}) \bmod 6 = 65 \bmod 6 = 5 \\h(\text{,,Nowak"}) &= (\text{,,N"}) \bmod 6 = 78 \bmod 6 = 0 \\h(\text{,,Ciesielski"}) &= (\text{,,C"}) \bmod 6 = 67 \bmod 6 = 1 \\h(\text{,,Laskowik"}) &= (\text{,,L"}) \bmod 6 = 76 \bmod 6 = 4 \\h(\text{,,Plis"}) &= (\text{,,P"}) \bmod 6 = 80 \bmod 6 = 2\end{aligned}$$

A - 65	G - 71	M - 77
B - 66	H - 72	N - 78
C - 67	I - 73	O - 79
D - 68	J - 74	P - 80
E - 69	K - 75	Q - 81
F - 70	L - 76	R - 82

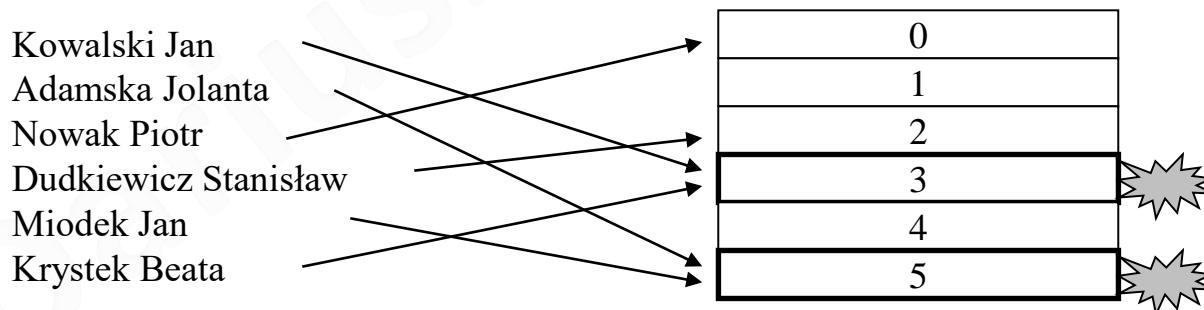
Kowalski Jan
Adamska Jolanta
Nowak Piotr
Ciesielski Stanisław
Laskowik Darek
Plis Beata



0	Nowak Piotr
1	Ciesielski Stanisław
2	Plis Beata
3	Kowalski Jan
4	Laskowik Darek
5	Adamska Jolanta

HT – nie zawsze idealnie

- Perfekcyjna funkcja mieszająca (ang. perfect hash function) to funkcja, która dla różnych kluczy generuje zawsze różne indeksy tablicy
 - W zasadzie możliwe tylko dla stałego zbioru kluczy (statyczna tablica mieszająca):
 - Słowa kluczowe danego języka programowania
 - Zbiór obserwowanych wyrazów w strumieniu
 - itp.
 - Specjalne techniki szukania takich funkcji
- Gdy zbiór kluczy jest dynamiczny (np. zwiększa się) lub znalezienie perfekcyjnej funkcji jest trudne pojawiają się konflikty:



HT – adresowanie otwarte

- Rozwiązywanie kolizji w adresowaniu otwartym:
 - Oprócz funkcji mieszającej $h(K)$ istnieje funkcja przyrostowa do wskazywania kolejnej pozycji do wstawienia w i-tej próbie: $p(i)$
 - W przypadku kolizji sprawdzane są kolejno pozycje $(h(K)+p(1))\%size$, $(h(K)+p(2))\%size$ itd.
 - Analogiczne pozycje musza być również sprawdzane przy znajdowaniu elementu

HT – próbkowanie liniowe

- $p(i)=i$ – najprostsza funkcja przyrostowa
- Oznacza, że w i-tej próbie sprawdza się pozycję $(h(K)+i) \% \text{size}$

A_5 – klucz, dla którego
wartość funkcji
mieszającej wynosi 5

A_5	A_2	A_3	0	
			1	
			2	A_2
			3	A_3
			4	
			5	A_5
			6	
			7	
			8	
			9	

B_5	A_9	B_2	0	
			1	
			2	A_2
			3	A_3
			4	B_2
			5	A_5
			6	B_5
			7	
			8	
			9	A_9

B_9	C_2	0	B_9
		1	
		2	A_2
		3	A_3
		4	B_2
		5	A_5
		6	B_5
		7	C_2
		8	
		9	A_9

- Próbkowanie liniowe jest łatwe do zaimplementowania, jednak powoduje problem zwany klastrowaniem pierwotnym.
- Procedura poszukiwania A_2, C_2
- Procedura poszukiwania A_4
- Procedura usuwania na przykładzie B_2

HT – próbkowanie kwadratowe, podwójne mieszanie

- $p(i) = c_1 i + c_2 i^2$
- lub $p(i) = (-1)^{i-1}((i+1)/2)^2$, czyli $(+1, -1, +4, -4, +9, -9, \dots)$

A ₅	A ₂	A ₃	0	
			1	
			2	A ₂
			3	A ₃
			4	
			5	A ₅
			6	
			7	
			8	
			9	

B ₅	A ₉	B ₂	0	
			1	B ₂
			2	A ₂
			3	A ₃
			4	
			5	A ₅
			6	B ₅
			7	
			8	
			9	A ₉

B ₉	C ₂	0	B ₉
		1	B ₂
		2	A ₂
		3	A ₃
		4	
		5	A ₅
		6	B ₅
		7	
		8	C ₂
		9	A ₉

- Próbkowanie kwadratowe powoduje problem zwany klastrowaniem wtórnym.

Rozwiążanie problemu klastrowania elementów:

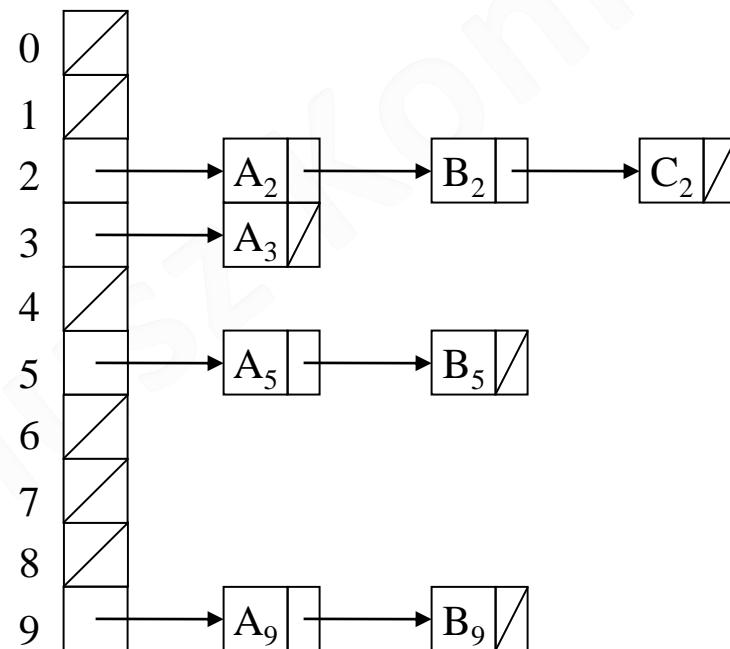
Podwójne mieszanie (funkcja przyrostowa również zależy od klucza): $p(i, K) = i * h_2(K)$

Adresowanie otwarte - usuwanie

- Usuwanie elementu z podanym kluczem:
 - Może w ogóle nie występować
 - Jeśli jest rzadkie, możemy postępować, jak zostało wytłumaczone na tym wykładzie
 - Jeśli może być częste można dodać znacznik do każdej pozycji tablicy, czy została usunięta (raz na jakiś czas robi się rzeczywiste usunięcie-reorganizację tablicy)
 - W pozycję ze znacznikiem można wstawić nowy element
 - Przy poszukiwaniu elementu pozycję ze znacznikiem traktuje się jak zajętą.

HT jako tablica list wiązanych

- Zamiast stosowania adresowania otwartego lepiej użyć tablicy list (lepsza wydajność kosztem pamięci).
 - Każda pozycja tablicy to lista wiązana (np. dwukierunkowa cykliczna ze strażnikiem)
 - Zamiast konfliktów mamy wstawianie do listy wskazanej przez funkcję mieszającą (np. na koniec w czasie $O(1)$) .
 - Usuwanie jest usuwaniem z listy wiązanej
 - Poszukiwanie to poszukiwanie na liście wiązanej
 - Ostatnie dwie operacje zależą od długości listy, ale zakładając dobrą funkcję mieszającą i mały stopień wypełnienia tablicy, listy te będą bardzo krótkie rzędu 2-3 elementów.



Funkcja mieszająca

- Funkcja mieszająca – pierwsza istotna składowa tablic mieszających:
 - Dla **tego samego klucza** musi zawsze zwracać **tą samą** wartość
 - Dla **podobnych** kluczy powinna dawać bardzo **różne** wartości
 - Powinna być szybka do wyliczenia

Stopień wypełnienia.

- Oznaczana jako α - jest to liczba elementów w tablicy do wszystkich pozycji tablicy. Stąd zawsze jest w przedziale $[0, 1]$.
- Im mniejsza wartość α , tym mniejsza szansa na kolizję.

HT – Złożoność $O(f(\alpha))$

- Złożoność zależy od stopnia wypełnienia (oczywiście dla dużych tablic) oraz obecności (lub nie) elementu w słowniku

algorytm	Liczba prób dla $el \in \text{słownik}$	Liczba prób dla $el \notin \text{słownik}$
Tablica list	$1+\alpha/2$	$1+\alpha$
Adresowanie liniowe	$0,5+1/(2*(1-\alpha))$	$0,5+1/(2*(1-\alpha)^2)$
Mieszanie podwójne	$-\ln(1-\alpha)/\alpha$	$1/(1-\alpha)$

HT – Złożoność-przykłady

Algorytm- sytuacja	Stopień wypełnienia tablicy - α			
	1/2	2/3	3/4	9/10
Tablica list - trafione	1,25	1,33	1,38	1,45
Tablica list - chybione	1,5	1,67	1,75	1,9
Próbkowanie liniowe - trafione	1,5	2,0	3,0	5,5
Próbkowanie liniowe - chybione	2,5	5,0	8,5	55,5
Podwójne mieszanie - trafione	1,4	1,6	1,8	2,6
Podwójne mieszanie - chybione	2,0	3,0	4	9

HT - podsumowanie

- Tablice mieszające są najlepsze jako słowniki.
 - Nie zapewniają jednak żadnego uporządkowania.
 - Stałe tablice mieszające wykorzystywane są np. w kompilatorach.
 - Dla poprawnej złożoności trzeba dobrać dobrą funkcję mieszającą
-
- Uwagi techniczne:
 - Java już w klasie `Object` ma metodę `hashCode()` do generowania wartości po zahaszowaniu obiektu:
 - Metoda ta standardowo haszuje referencję, co nie jest tym czego oczekujemy. Jeśli chcemy korzystać z tablic mieszających należy ją nadpisać własną wersją.
 - Wiele klas ma właściwą wersję tej metody np. `String`. W tworzeniu własnej metody haszującej warto używać wartości z metody `hashCode()` pól własnej klasy, łącząc je poprzez operację bitową XOR. Np. tak zostało to zaimplementowane w klasie `AbstractList<T>`:

```
// ^ - bitowa różnica symetryczna
@Override
public int hashCode() {
    int hashCode = 0;
    for (E item:this)
        hashCode ^= item.hashCode();
    return hashCode;
}
```

Algorytmy i struktury danych – W07

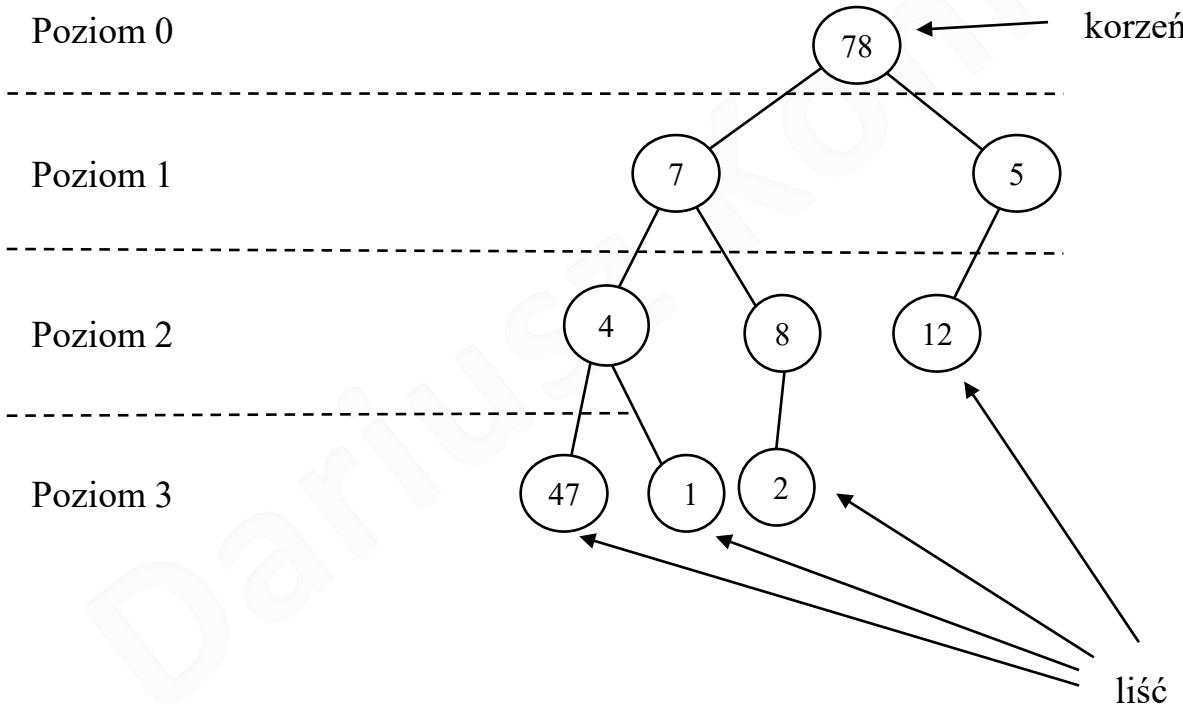
BST - Binarne drzewo poszukiwań

Zawartość

- Drzewo binarne – definicja
- BST - definicja
- BST: realizacja z referencją do rodzica - pseudokod
 - Podstawowe operacje na BST: wyszukiwanie, przegląd, min, max, następnik, wstawianie, usuwanie
- BST: realizacje bez referencji na rodzica - Java
 - Podstawowe operacje na BST : wyszukiwanie, przegląd, min, max, następnik, wstawianie, usuwanie
- Ogólny schemat metod na drzewie binarnym
- Drzewo niewyważone
- Podsumowanie

Drzewo binarne

- Drzewo binarne to taka spójna struktura węzłów, w której każdy węzeł może mieć co najwyżej dwa bezpośrednie pod-węzły (zwane **dziećmi/potomkami**).
- Nad-węzeł danego węzła nazywany jest **rodzicem/przodkiem**.
- Jest jeden węzeł bez rodzica – nazywany jest **korzeniem**.
- Węzły bez potomków nazywamy **liśćmi**.
- **Poziom/głębokość węzła** to liczba krawędzi od korzenia w dół do węzła.
- **Wysokość drzewa** to maksimum z głębokości liści
- Dla pustego drzewa wysokość jest ustalona na -1.

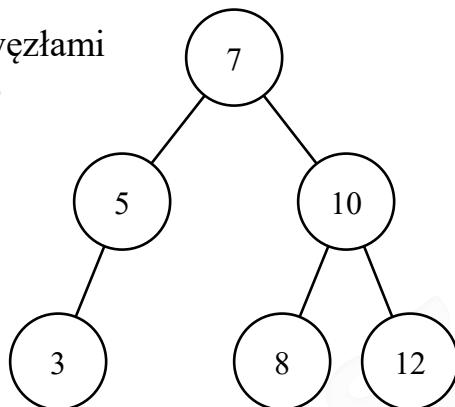


BST

- **Drzewo BST** (ang. binary search tree), czyli drzewo poszukiwań binarnych (lub binarne drzewo poszukiwań) jest drzewem binarnym z dodatkową właściwością:

Niech x będzie węzłem w drzewie BST. Jeśli y jest węzłem w lewym poddrzewie x , to $\text{key}[y] \leq \text{key}[x]$. Jeśli y jest węzłem w prawym poddrzewie x , to $\text{key}[y] \geq \text{key}[x]$.

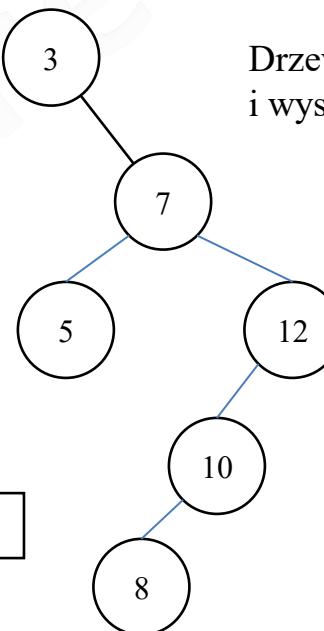
Drzewo z $n = 6$ węzłami i wysokości $h = 3$



$$\log n \leq h \leq n$$

Drzewo z $n = 6$ węzłami i wysokości $h = 5$

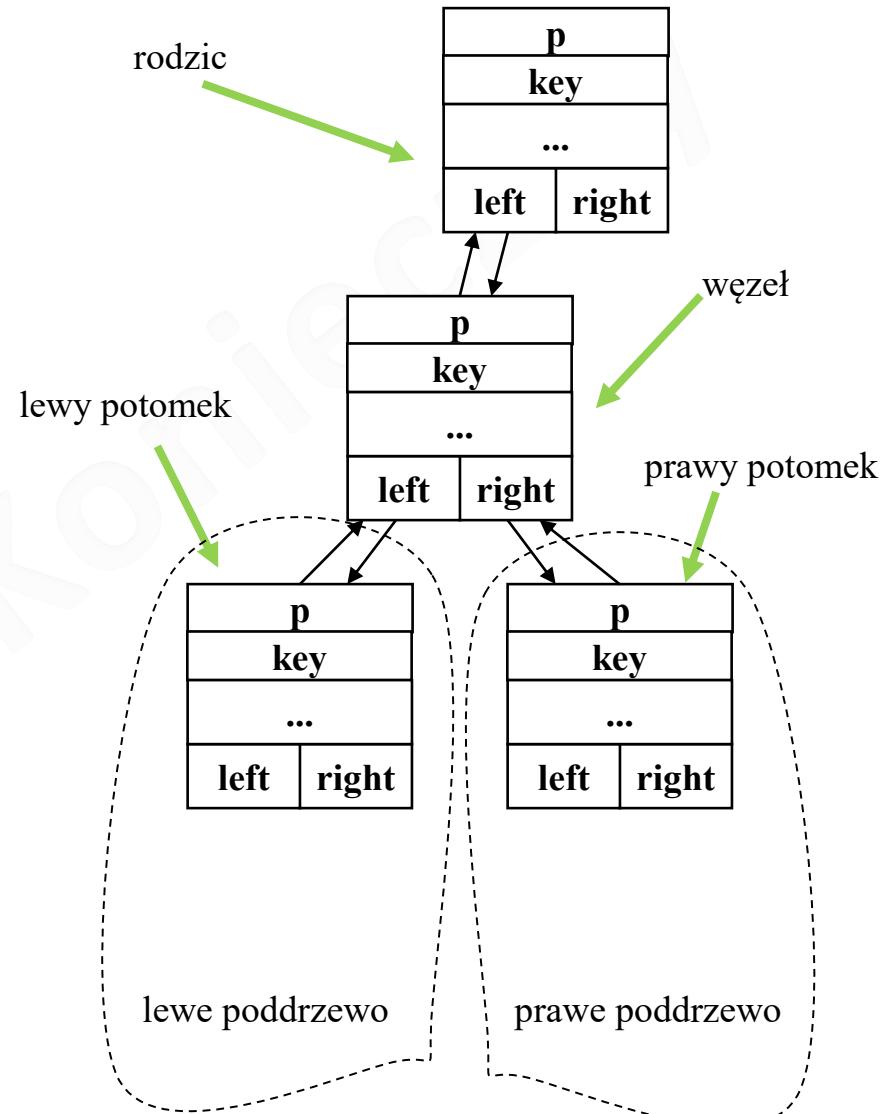
$$\text{Średnia wysokość } h = O(\log n)$$



- W niektórych definicjach zakłada się, że **klucze w drzewie nie mogą się powtarzać**, stąd w definicji pojawiają się ostre nierówności. Warunek ten będzie **dodany w realizacjach drzewa na tym wykładzie**.

BST – realizacja 1

- Pierwsza realizacja:
 - Każdy węzeł posiada pole key, left, right i p do pamiętania odpowiednio: klucza, wiązanie do lewego i prawego potomka, wiązanie do rodzica. Może też posiadać jakiś inne dodatkowe pola.
 - Korzeń w polu rodzica ma wartość **null**, analogicznie liście w polach left i right mają wartość **null**.
- Prawy potomek wraz z wszystkimi podwężłami aż do liści nazywany jest prawym poddrzewem, analogicznie definiuje się lewe poddrzewo.



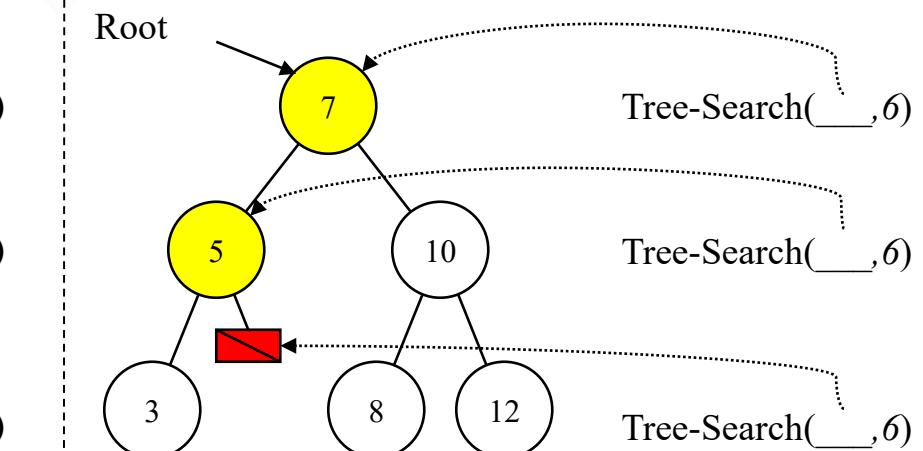
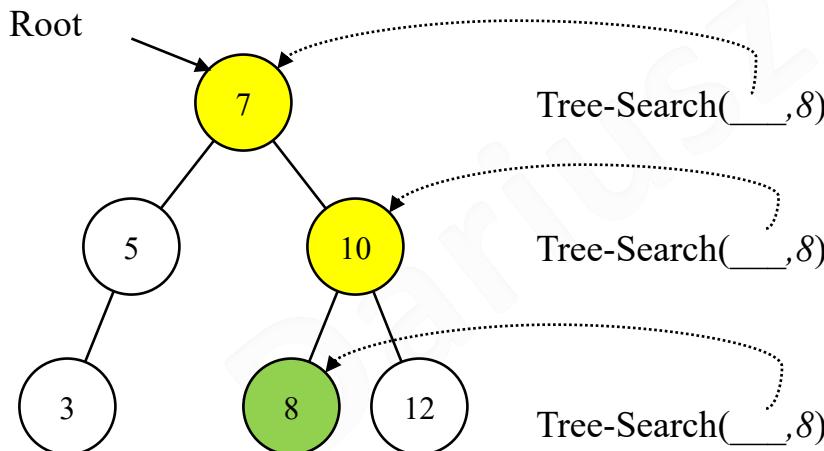
Wyszukiwanie węzła w BST

- Parametrem jest klucz poszukiwanego elementu. Wynikiem – węzeł ze znalezionym kluczem lub wartość **null**.
- Korzystamy z własności BST.
- Idea: Zaczynając od korzenia porównujemy poszukiwany klucz z kluczem w danym węźle. Równe – znaleźliśmy węzeł. Poszukiwany klucz jest mniejszy – szukamy w lewym poddrzewie, większy – w prawym poddrzewie.

```
Tree-Search(x,k)
{1} if (x = null) or (k = key[x])
    then return x
{2} if k<key[x]
    then return Tree-Search(left[x],k)
{3} else return Tree-Search(right[x],k)
```

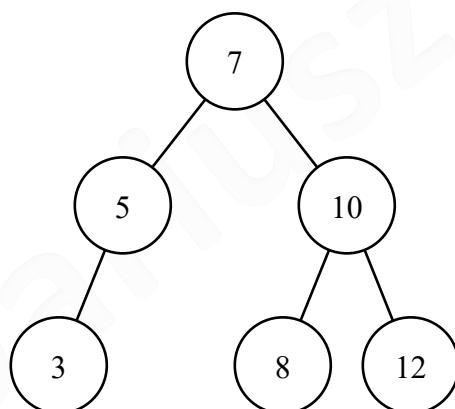
Tree-Search (root, k)

złożoność: $O(h)$



Przegląd drzewa w BST

- Przeglądanie drzewa (ang. *tree-walk*) polega na odwiedzeniu wszystkich wierzchołków dokładnie raz (w usystematyzowany sposób).
- Odwiedzenie wierzchołka (np. wypisywanie do strumienia) może być wykonane na następujące sposoby:
 - Zanim odwiedzimy jego poddrzewa (*preorder-walk*)
 - Po odwiedzeniu poddrzew (*postorder-walk*)
 - Pomiędzy odwiedzaniem poddrzew (*inorder-walk*).
- Powyższe sposoby mają dwie wersje:
 - najpierw lewe poddrzewo, potem prawe (ten wykład)
 - najpierw prawe poddrzewo, potem lewe



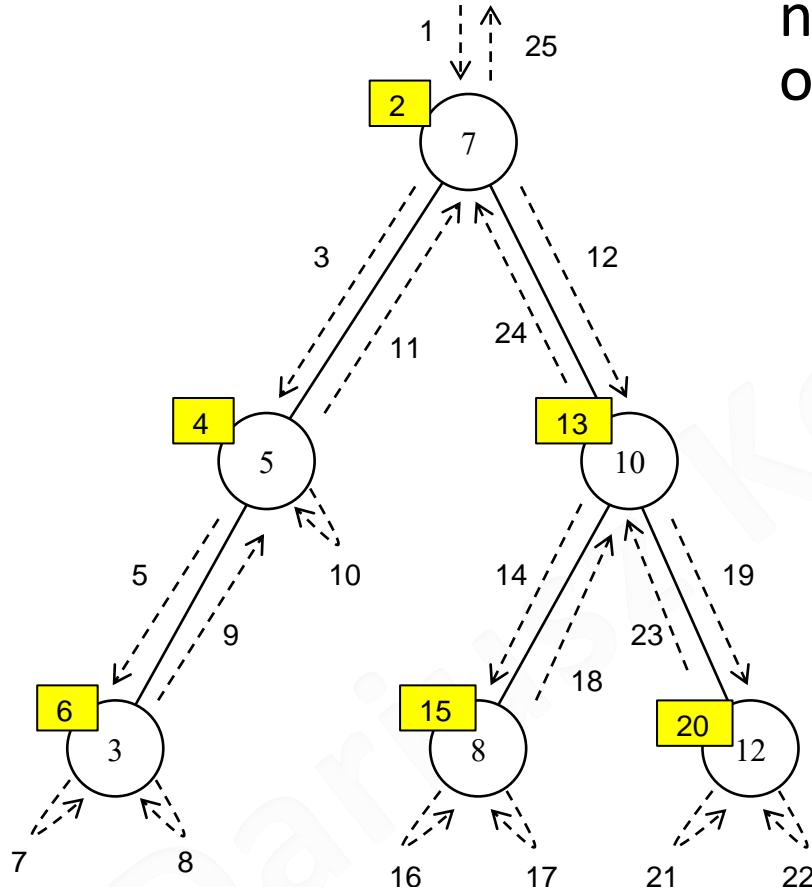
Preorder-Walk: 7,5,3,10,8,12

Postorder-Walk: 3,5,8,12,10,7

Inorder-Walk: 3,5,7,8,10,12

Przegląd pre-order

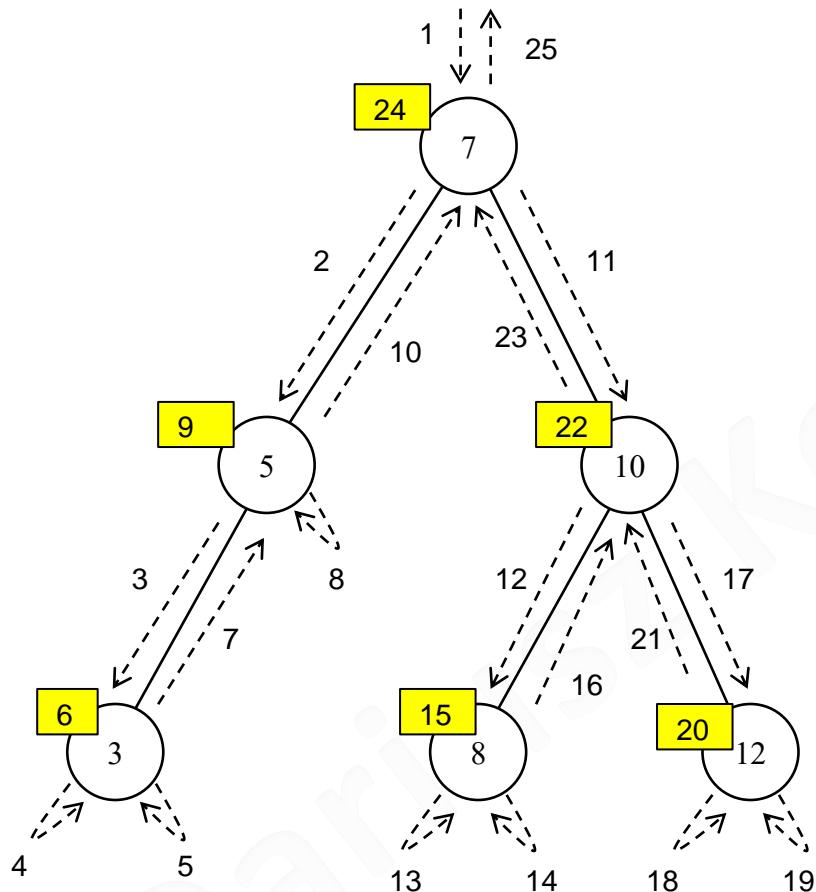
- Odwiedzenie wierzchołka następuje **przed** odwiedzeniem poddrzew



Preorder-Walk: 7,5,3,10,8,12

Przegląd post-order

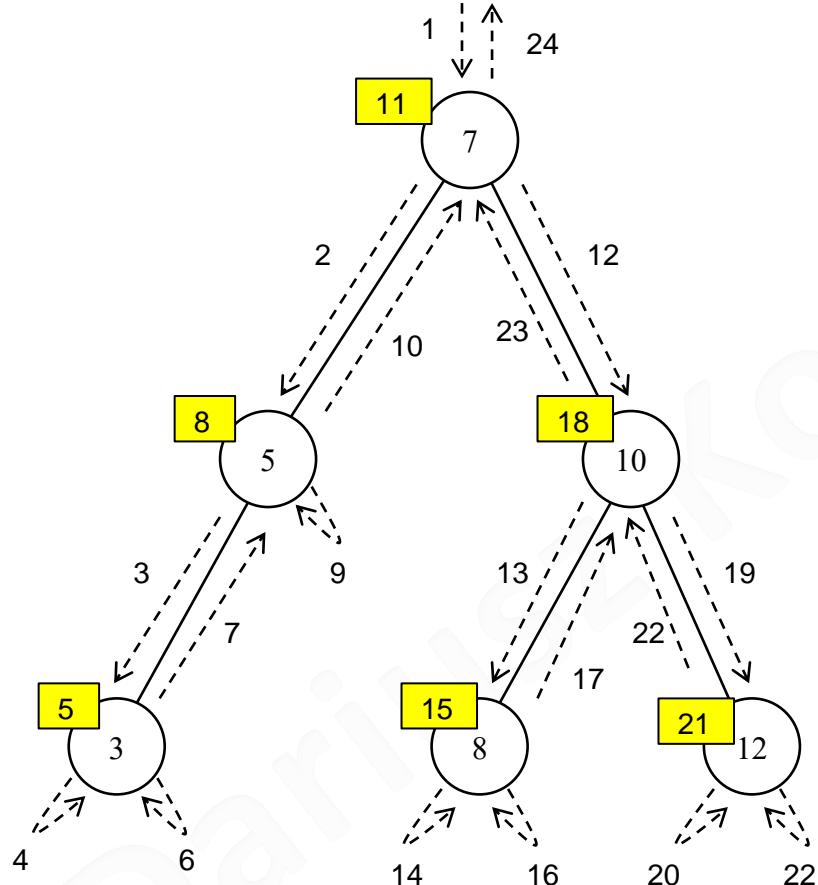
- Odwiedzenie węzła następuje **po** odwiedzeniu poddrzew



Postorder-Walk: 3,5,8,12,10,7

Przegląd in-order

- Odwiedzenie węzła następuje **pomiędzy** odwiedzaniem poddrzew



Inorder-Walk: 3,5,7,8,10,12

Przegląd in-order – kod i analiza

```
Tree-Inorder-Walk(x)
{ 1}   if x <> null then
{ 2}     Tree-Inorder-Walk(left[x])
{ 3}     show key[x]
{ 4}     Tree-Inorder-Walk(right[x])
```

- Wywołanie `Tree-Inorder-Walk(root)`
- Częsty schemat w działaniu na drzewie: dwie metody:
 - jedna rekurencyjna działa dla węzła podanego w argumencie wywołania
 - druga wywołuje pierwszą z korzeniem jako argument
- Pozostałe przeglądy różnią się tylko miejscem wywołania {3} linii kodu
- Przegląd In-order – przegląd w porządku posortowanych kluczy, stąd najczęściej używany
- Ciekawy problem: stworzenie iteratorów dla każdego ze sposobów przeglądu
- Złożoność: $\Theta(n)$
- Głębokość rekurencji: $\Theta(h)$

Wyszukiwanie min i max w BST

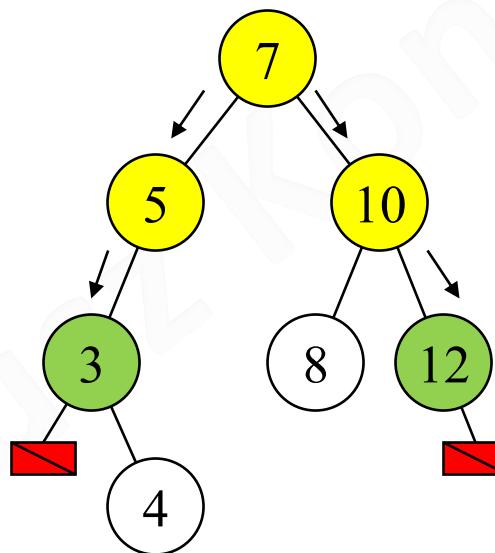
- Poszukiwanie minimum w drzewie BST polega na przehodzeniu w lewo aż do węzła, który nie posiada lewego potomka
- Operacja poszukiwania maksimum przebiega analogicznie z użyciem prawego potomka.

```
{ 1}  
{ 2}  
{ 3}
```

```
Tree-Minimum(x)  
while left[x] <> null do  
    x := left[x]  
return x
```

```
{ 1}  
{ 2}  
{ 3}
```

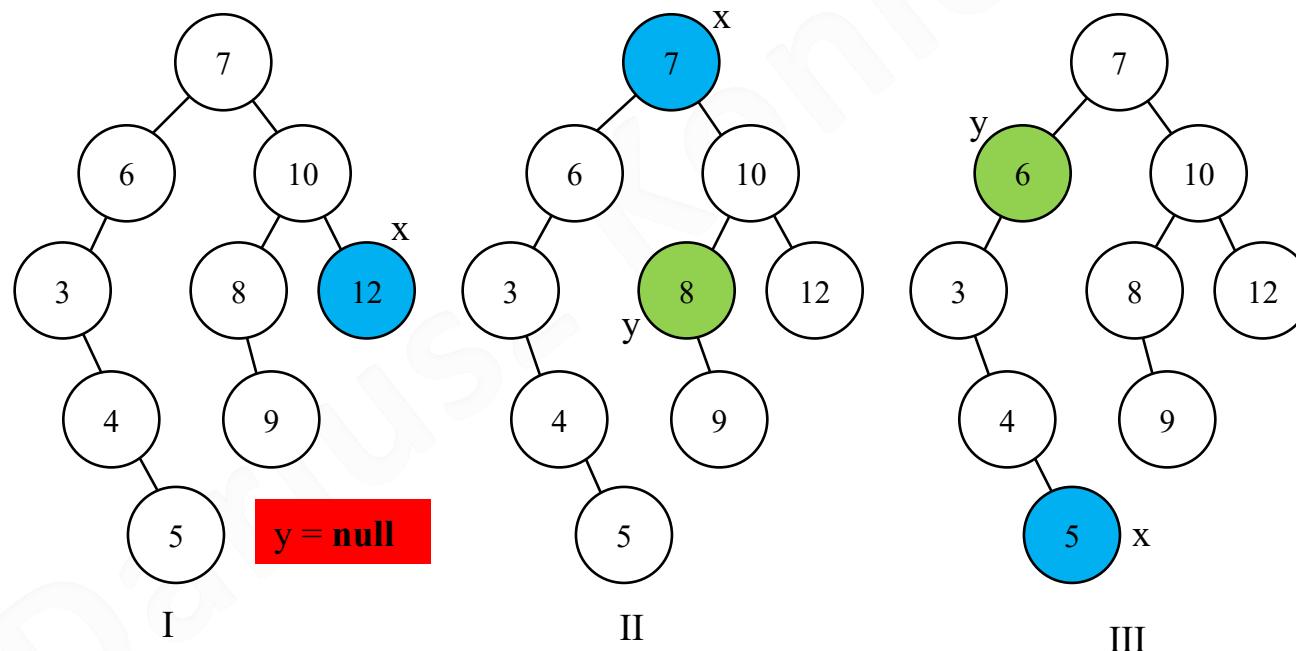
```
Tree-Maksimum(x)  
while right[x] <> null do  
    x := right[x]  
return x
```



złożoność: $O(h)$

Następnik i poprzednik w BST

- Dla wybranego węzła w drzewie BST nierzaz potrzeba znaleźć jego następnik, czyli węzeł odwiedzany jako kolejny przy przeglądzie in-order.
 - Poprzednika definiuje się analogicznie.
 - Są możliwe 3 przypadki przy szukaniu następnika:
 - I) Węzeł nie ma następnika
 - II) Następnik tego węzła znajduje się w jego prawym poddrzewie
 - III) Następnik tego węzła znajduje się wyżej, na ścieżce do korzenia



Wyszukiwanie następnika w BST

- W przypadku II) trzeba znaleźć minimum w jego prawym poddrzewie
- W przypadkach I) i III) należy zmierzać w kierunku korzenia aż dojdziemy do węzła od strony lewego dziecka. Jeśli takiego węzła nie znajdziemy – brak następnika.

```
Tree-Successor(x)
{ 1}   if right[x] <> null then
{ 2}     return Tree-Minimum(right[x])
{ 3}   y := p[x]
{ 4}   while (y <> null) and (x = right[y]) do
{ 5}     x := y
{ 6}     y := p[y]
{ 7}   return y
```

Złożoność: $O(h)$

Wstawianie elementu w BST 1/2

- Wstawienie nowego węzła o kluczu v polega na znalezieniu dla niego miejsca jako liść i przebiega bardzo podobnie do wyszukiwania.
- W tym pseudokodzie zakłada się, że węzeł został już stworzony oraz wypełniony tak, że $\text{key}[z] = v$, $\text{left}[z] = \text{null}$, $\text{right}[z] = \text{null}$, oraz, że klucza v nie ma jeszcze w drzewie.

```
{ 1} Tree-Insert(root, z)
{ 2} y := null
{ 3} x := root
{ 4} while x <> null do
{ 5}     y := x
{ 6}     if key[z] < key[x]
{ 7}         then x := left[x]
{ 8}         else x := right[x]
{ 9}     p[z] := y
{10}    if y = null
{11}        then root := z
{12}        else if key[z] < key [y]
{13}            then left[y] := z
{14}            else right[y] := z
```

Drzewo po wstawieniu kluczy:

5, 3, 7, 11, 4, 2, 12, 10

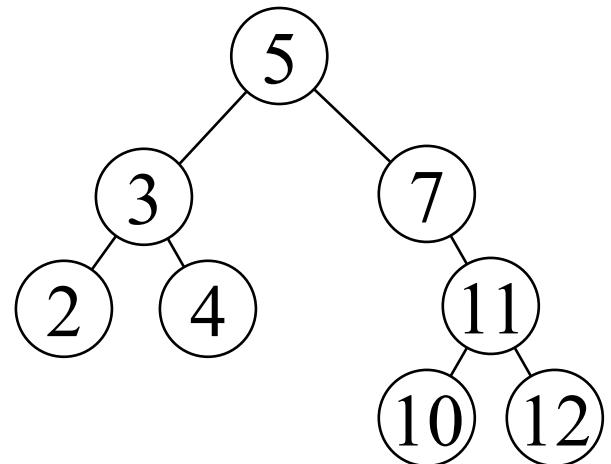
Wstawianie elementu w BST 1/2

- Wstawienie nowego węzła o kluczu v polega na znalezieniu dla niego miejsca jako liść i przebiega bardzo podobnie do wyszukiwania.
- W tym pseudokodzie zakłada się, że węzeł został już stworzony oraz wypełniony tak, że $\text{key}[z] = v$, $\text{left}[z] = \text{null}$, $\text{right}[z] = \text{null}$, oraz, że klucza v nie ma jeszcze w drzewie.

```
{ 1} Tree-Insert(root, z)
{ 2} y := null
{ 3} x := root
{ 4} while x <> null do
{ 5}     y := x
{ 6}     if key[z] < key[x]
{ 7}         then x := left[x]
{ 8}         else x := right[x]
{ 9}     p[z] := y
{10}    if y = null
{11}        then root := z
{12}        else if key[z] < key [y]
{13}            then left[y] := z
{14}            else right[y] := z
```

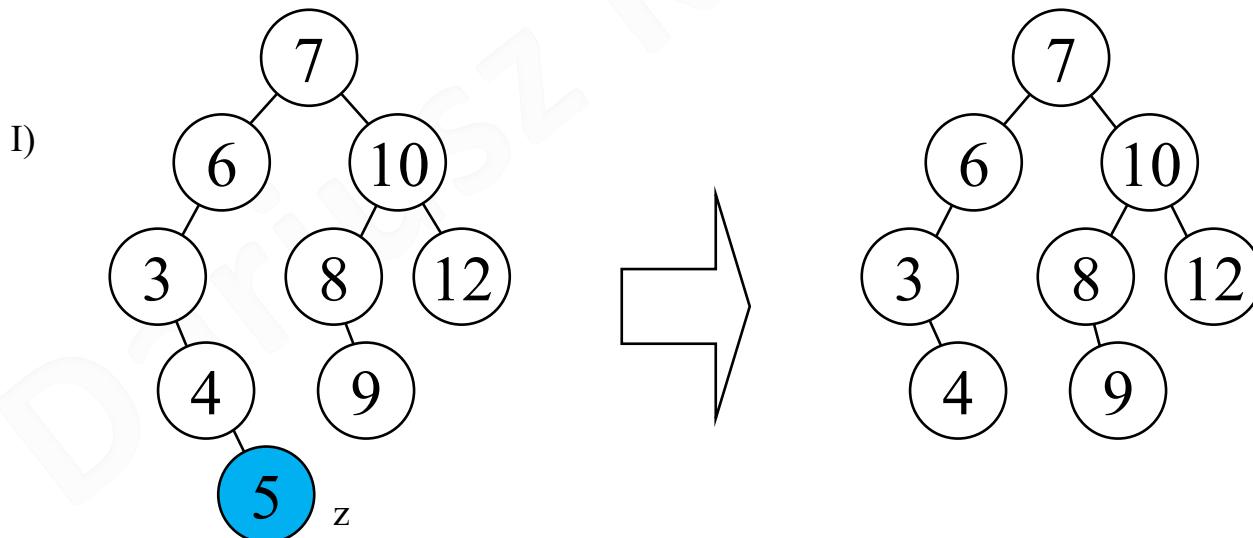
Drzewo po wstawieniu kluczy:

5, 3, 7, 11, 4, 2, 12, 10

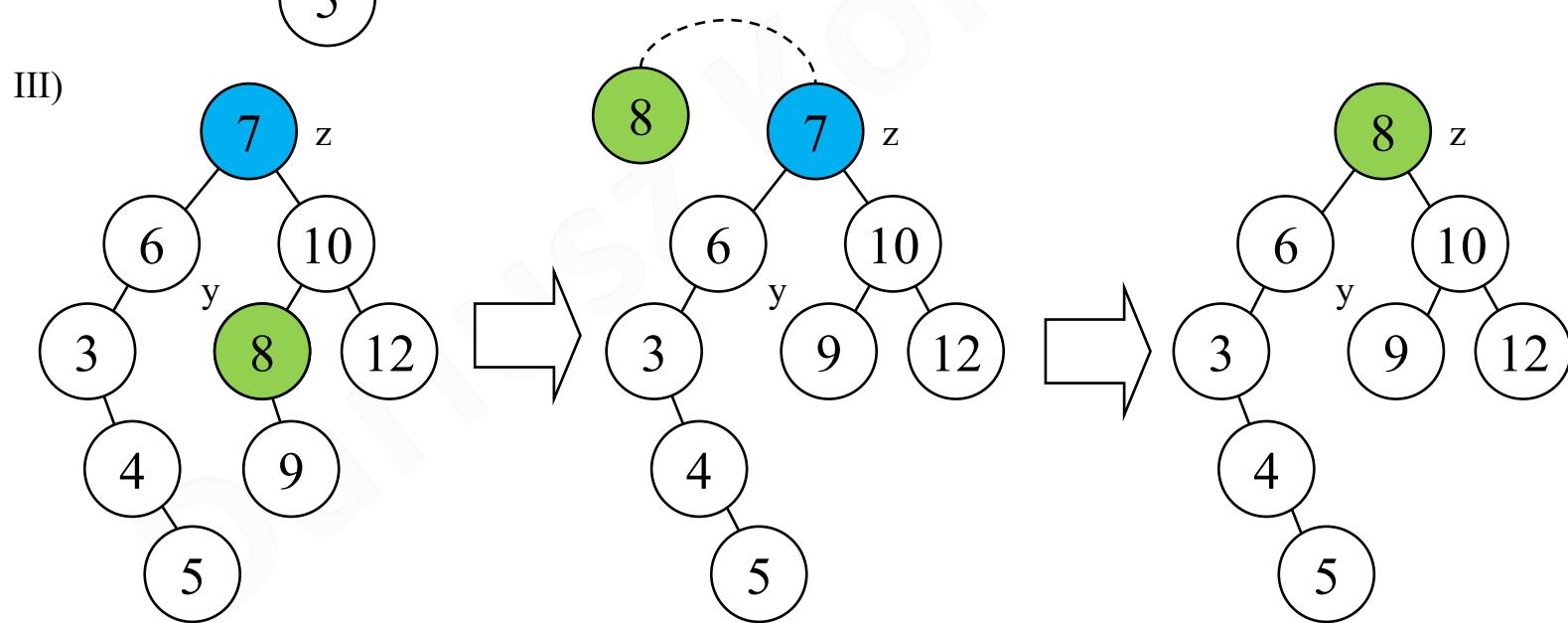
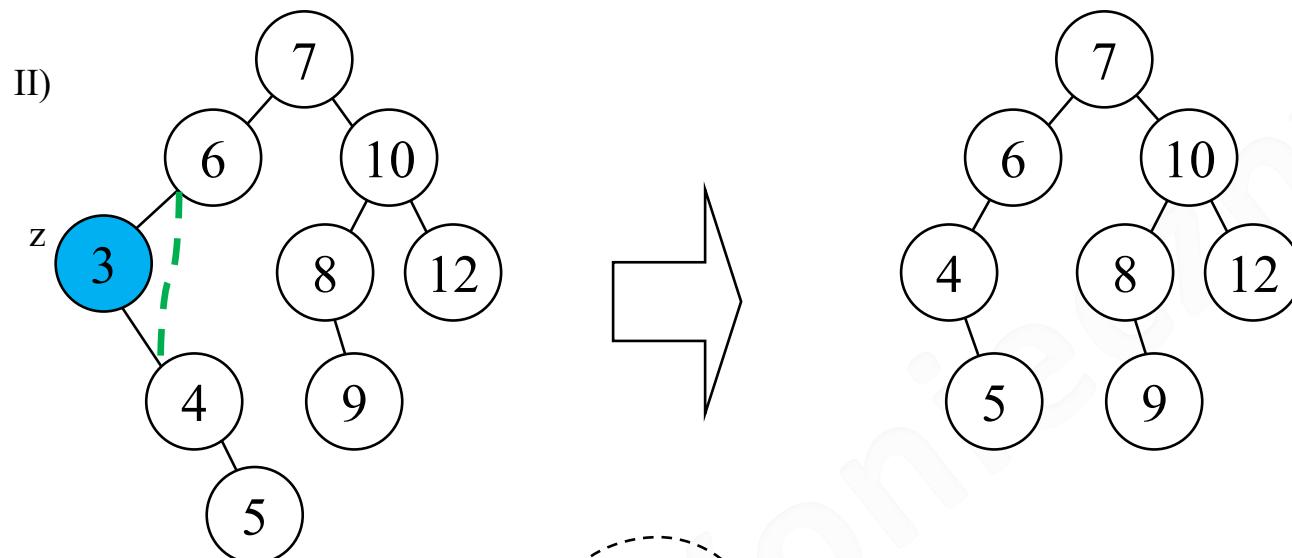


Usuwanie elementu w BST 1/2

- Usuwanie elementu/węzła z to w zasadzie naprawianie drzewa w minimalnej ilości kroków, aby nadal było BST.
- Istnieją 3 przypadki podczas usuwania węzła z z drzewa BST:
 - I) węzeł z nie ma potomków
 - II) węzeł z ma dokładnie jednego potomka
 - III) węzeł z ma dwóch potomków
- Co należy zrobić:
 - I) w rodzicu węzła z zmodyfikować odpowiednie pole potomka na wartość **null** (ewentualnie ustawić korzeń na **null**)
 - II) w rodzicu węzła z zmodyfikować odpowiednie pole potomka na wartość jedynego potomka węzła z .
 - III) znaleźć następnik węzła z (nazwijmy go y). Ten następnik na pewno nie ma dwóch potomków. Wymienić pola klucza i ewentualnie inne dodatkowe dane między węzłami z i y . Następnie usunąć z drzewa węzeł y (czyli przypadek I lub II).



Usuwanie elementu w BST 2/2



Usuwanie elementu w BST - kod

- W tej realizacji łatwo dodatkowo zwrócić usunięty węzeł

```
Tree-Delete(root, z)
{ 1}   if (left[z] = null) or (right[z] = null)
{ 2}     then y:= z
{ 3}   else y := Tree-Successor(z)
{ 4}   if left[y] <> null
{ 5}     then x := left[y]
{ 6}     else x := right[y]
{ 7}   if x <> null
{ 8}     then p[x] = p[y]
{ 9}   if p[y] = null
{10}    then root := x
{11}   else if y = left[p[y]]
{12}     then left[p[y]] := x
{13}     else right[p[y]] := x
{14}   if y <> z
{15}     then swap(key[z], key[y])
{16}     { jeśli węzeł posiada inne pola – również je wymienić}
{17}   return y
```

złożoność: $O(h)$

BST – realizacja 2

- Większość operacji dla BST nie wymaga wiązania w górę (do rodzica)
- Można zrealizować węzeł w drzewie BST bez wiązania do rodzica.
- Korzystając z OOP zamiast klucza lepiej użyć komparatora
- Wewnętrzna wartość (`value`) będzie nazywana **elementem**.

```
public class BST<T> {  
    class Node{  
        T value; // element  
        Node left;  
        Node right;  
        Node(T obj) {  
            value=obj; }  
        Node(T obj, Node leftNode,Node rightNode) {  
            value=obj;  
            left=leftNode;  
            right=rightNode; }  
    }  
    private final Comparator<T> _comparator;  
    private Node _root;  
    public BST(Comparator<T> comp) {  
        _comparator=comp;  
        _root=null; }  
}
```

Wyszukiwanie elementu

- Praktycznie nie różni się od pierwszej realizacji.
- Tym razem realizacja iteracyjna
- Prywatna funkcja zwracająca referencję na znaleziony węzeł (a nie tylko element)

```
public T find(T elem) {
    Node node=search(elem);
    return node==null?null:node.value;
}

private Node search(T elem) {
    Node node=_root;
    int cmp=0;
    while(node!=null && (cmp=_comparator.compare(elem, node.value))!=0)
        node=cmp<0? node.left:node.right;
    return node;
}
```

Przegląd inorder z egzekutorem 1/2

- Wykorzystanie egzekutora do lepszego wykorzystania przeglądu drzewa – zamiast show może być dowolna metoda.
- Przedstawiany egzekutor będzie jednokrotnego użycia.

```
package aisd.executor;
public interface IExecutor<T, R> {
    void execute(T elem);
    R getResult();
}
```

```
public <R> void inOrderWalk(IExecutor<T, R> exec) {
    inOrderWalk(_root, exec);
}
private <R> void inOrderWalk(Node node, IExecutor<T, R> exec) {
    if(node!=null) {
        inOrderWalk(node.left, exec);
        exec.execute(node.value);
        inOrderWalk(node.right, exec);
    }
}
```

- Trudniejszy problem – stworzenie iteratorów dla przeglądów w tej realizacji – dla optymalnej złożoności obliczeniowej należy zastosować stos wewnątrz iteratora pamiętający węzły na ścieżce do korzenia wraz z tym korzeniem.

Przegląd inorder z egzekutorem 2/2

- Egzekutor z buforem do pamiętania ciągu znaków

```
class IntegerToStringExec implements IExecutor<Integer, String>{  
    StringBuffer line=new StringBuffer();  
    @Override  
    public void execute(Integer elem) {  
        line.append(elem+"; ");}  
    @Override  
    public String getResult(){  
        line.delete(line.length()-2, line.length());  
        return line.toString();}}
```

```
public static void main(String[] args) {  
    BST<Integer> tree=new BST<Integer>(new Comparator<Integer>() {  
        public int compare(Integer o1, Integer o2) {  
            return o1-o2; }});  
    tree.insert(7);  
    tree.insert(5);  
    tree.insert(2);  
    tree.insert(10);  
    tree.insert(12);  
    IntegerToStringExec exec=new IntegerToStringExec();  
    tree.inOrderWalk(exec);  
    System.out.println(exec.getResult());}
```

2; 5; 7; 10; 12

Wyszukiwanie min i max

- W praktyce OOP warto rozbić na prywatną metodę szukającą węzła i publiczną zwracającą element przechowywany w węźle.

```
public T getMin() {  
    if (_root==null) throw new NoSuchElementException();  
    Node node=getMin(_root);  
    return node.value;  
}  
  
public T getMax() {  
    if (_root==null) throw new NoSuchElementException();  
    Node node=getMax(_root);  
    return node.value;  
}  
  
private Node getMin(Node node) {  
    assert(node!=null);  
    while (node.left!=null)  
        node=node.left;  
    return node;  
}  
  
private Node getMax(Node node) {  
    assert(node!=null);  
    while (node.right!=null)  
        node=node.right;  
    return node;  
}
```

Wyszukiwania następnika

- Szukanie następnika, mając inny węzeł jako argument uniemożliwia użycia wprost algorytmu z pierwszej realizacji
 - Nie możemy poruszać się od węzła w kierunku korzenia
- Będziemy szukać następnika elementu. Stąd najpierw poszukamy węzła z elementem, a potem wracając można użyć ścieżki powrotnej dla przypadku I i III (slajd 13)

```
public T successor(T elem) {  
    Node succNode=successorNode(_root, elem);  
    return succNode==null?null:succNode.value;}  
  
private Node successorNode(Node node, T elem) {  
    if(node==null) throw new NoSuchElementException();  
    int cmp=_comparator.compare(elem, node.value);  
    if(cmp==0) {  
        if(node.right!=null)  
            return getMin(node.right);  
        else return null;  
    } else if(cmp<0) {  
        Node retNode=successorNode(node.left, elem);  
        return retNode==null?node:retNode;  
    } else { // cmp>0  
        return successorNode(node.right, elem);  
    }  
}
```

Wstawianie elementu

- Wersja iteracyjna jako ćwiczenie – analogicznie jak kod pierwszej realizacji, bez linii {8} (slajd 15), z tworzeniem elementu tuż przed wstawieniem.
- Wersja rekurencyjna – Java przekazuje argumenty do metod przez wartość, więc w celu wstawienia nowej wartości trzeba ponownie dowiązywać poddrzewa, w których zostanie dodany nowy element

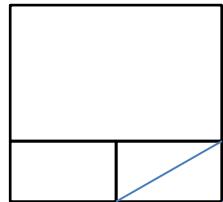
```
public void insert(T elem) {
    _root=_insert(_root,elem); }

private Node insert(Node node, T elem) {
    if(node==null)
        node=new Node(elem);
    else{
        int cmp=_comparator.compare(elem, node.value);
        if(cmp<0)
            node.left=_insert(node.left,elem);
        else if(cmp>0)
            node.right=_insert(node.right,elem);
        else
            throw new DuplicateElementException(elem.toString());
    }
    return node;
}
```

Wstawianie elementu

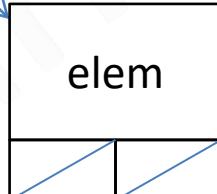
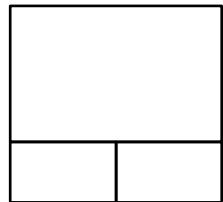
- Dojście do miejsca wstawienia

node



```
node.right=insert(node.right,elem)  
if(node==null)  
    node=new Node(elem);  
else{ ...  
}  
return node;
```

node



Wstawianie elementu

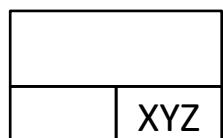
- Pozostałe wywołania rekurencyjne

node

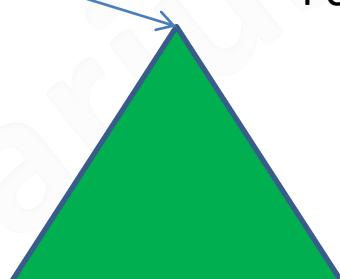
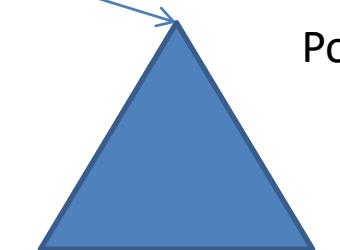


Poddrzewo przed wstawieniem elementu

node



Poddrzewo po wstawieniu elementu



Usuwanie elementu

- Nie można użyć wersji z pierwszej realizacji wprost
 - Należałoby uzupełnić wersję szukania węzła tak, aby zwracał również węzeł rodzica.
- Druga koncepcja to realizacja rekurencyjna, która wracając dowiązuje powrotnie elementy do swojego rodzica (podobny pomysł jak przy dodawaniu)
- Dla ułatwienia metoda publiczna będzie typu **void**.
- Jeśli schodząc w dół dojdziemy do przypadku I lub II, rozwiążemy go bez problemu. W przypadku III – dwójka dzieci – osobna metoda usunie minimum z prawego poddrzewa, wcześniej wartość tego węzła wstawiając do węzła z dwójką dzieci.

```
public void delete(T elem) {  
    _root=_root.delete(elem,_root);  
}
```

//... kontynuacja na następnym slajdzie

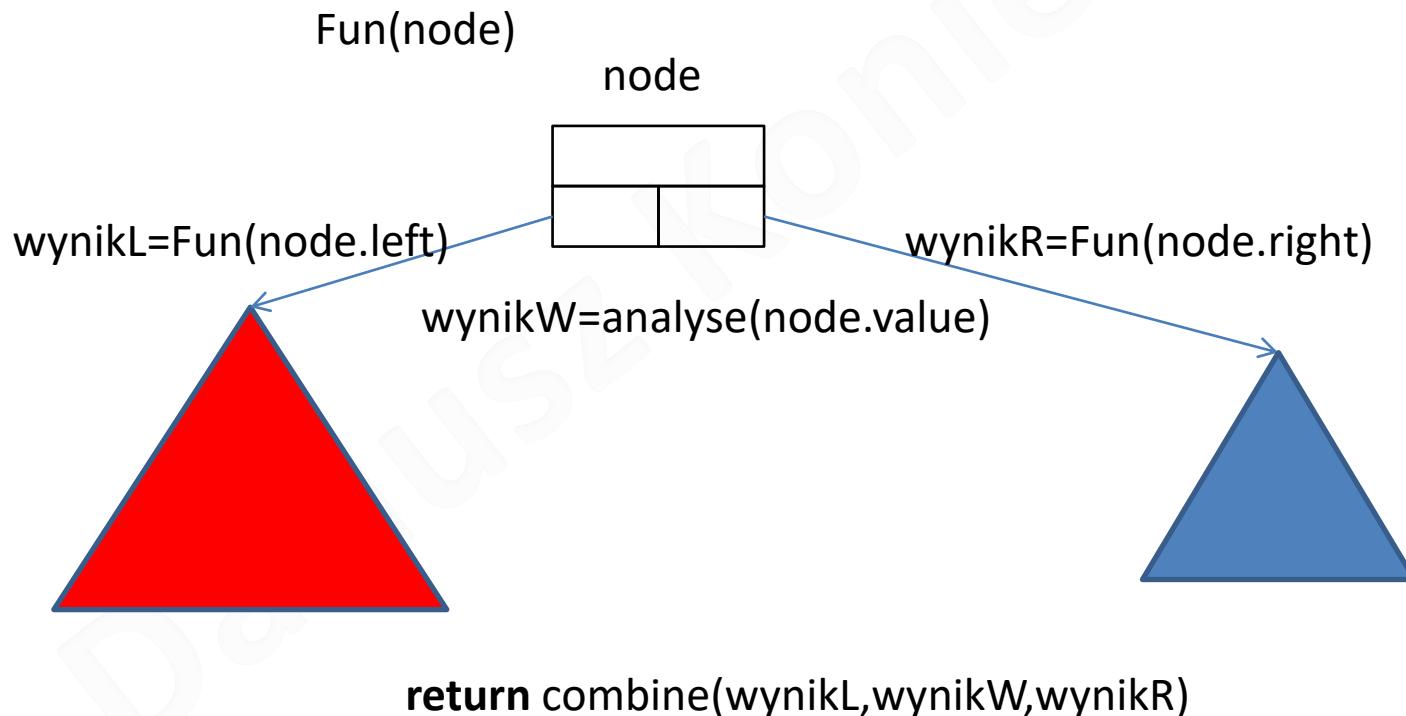
Usuwanie elementu

```
protected Node delete(T elem, Node node) {
    if(node==null) throw new NoSuchElementException();
    else {
        int cmp=_comparator.compare(elem, node.value);
        if(cmp<0)
            node.left=delete(elem, node.left);
        else if(cmp>0)
            node.right=delete(elem, node.right);
        else if(node.left!=null &&node.right!=null)
            node.right=detachMin(node, node.right);
        else node = (node.left != null) ? node.left : node.right;
    }
    return node;
}

private Node detachMin(Node del, Node node) {
    if(node.left!=null) node.left=detachMin(del, node.left);
    else {
        del.value=node.value;
        node=node.right;
    }
    return node;
}
```

Schemat metod dla drzewa binarnego

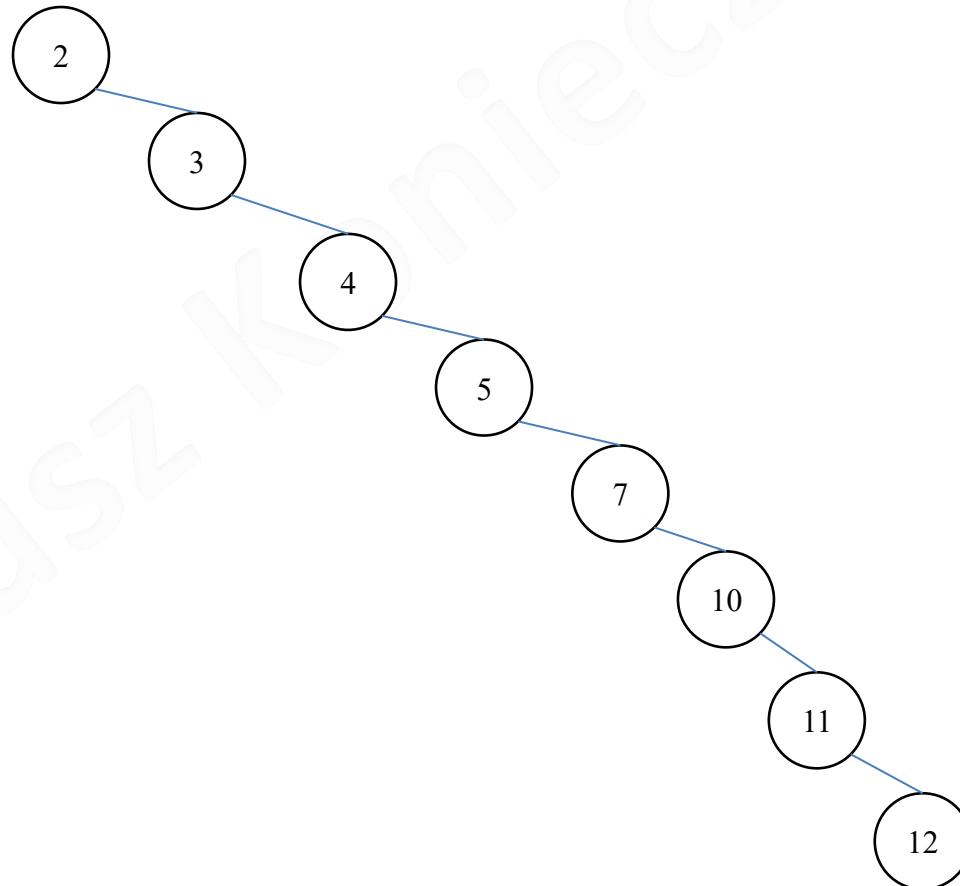
- Duża część metod niemodyfikujących drzewo binarne, implementowanych rekurencyjnie, działa wg poniższego schematu („dziel i rządź”):
 - Wykonaj metodę dla lewego dziecka (i otrzymaj wynikL)
 - Wykonaj metodę dla prawego dziecka (i otrzymaj wynikP)
 - Zanalizuj element wejściowego węzła (i otrzymaj wynikW)
 - Połącz wszystkie wyniki, aby otrzymać wynik ostateczny dla poddrzewa zakorzenionego w wejściowym węźle



Niewyważone drzewa

- Proste drzewo BST podczas tworzenia/usuwania może stać się mocno niewyważone.
- Paradoksalnie najgorszy ciąg wartości to np. ciąg posortowany – tworzy zdegenerowane drzewo. Zdegenerowane do listy wiązanej ze złożonością liniową większości operacji

Drzewo po wstawieniu kluczy:
2, 3, 4, 5, 7, 10, 11, 12



BST - podsumowanie

- Złożoność większości operacji na BST zależy od wysokości drzewa: $O(h)$
- Proste drzewo BST może być bardzo niewyważone: $h=O(n)$
- Drzewo BST można implementować z wiązaniem do rodzica i bez
 - Zalety i wady
- Wiele operacji można zaimplementować rekurencyjnie lub iteracyjnie:
 - Część operacji zaimplementowanych iteracyjnie wymaga wewnętrznego stosu, część – nie

- Informacja techniczna:
- W bibliotece standardowej Javy nie ma implementacji prostego drzewa BST.
 - Nie zawsze optymalna struktura

Algorytmy i struktury danych – W08

Drzewa wyważone: czerwono-czarne
i B-drzewa

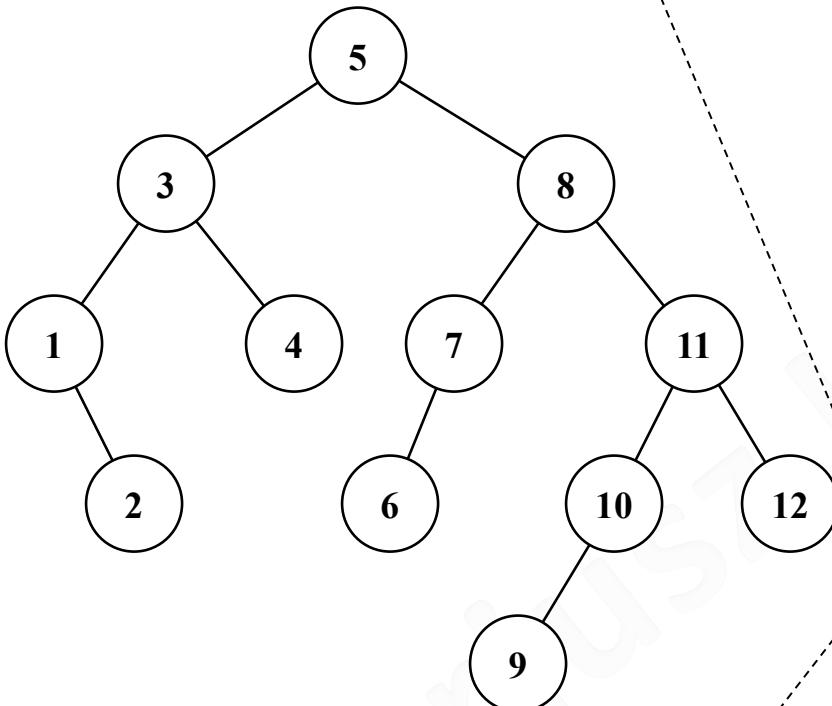
Zawartość

- Drzewa samorównoważące się
- Rotacja w drzewie BST
- Drzewo AVL – podstawowe informacje
- Drzewo czerwono-czarne:
 - Definicja
 - Czarna wysokość
 - Wstawianie węzła
 - Usuwanie węzła i naprawa po usunięciu
 - Podsumowanie, złożoność operacji
- B-drzewo:
 - Definicja
 - Operacje pomocnicze
 - Operacja wstawienia węzła
 - Opis usuwania węzła
 - Podsumowanie, złożoność operacji
- Dodatek: drzewo AVL w szczegółach

Drzewa binarne

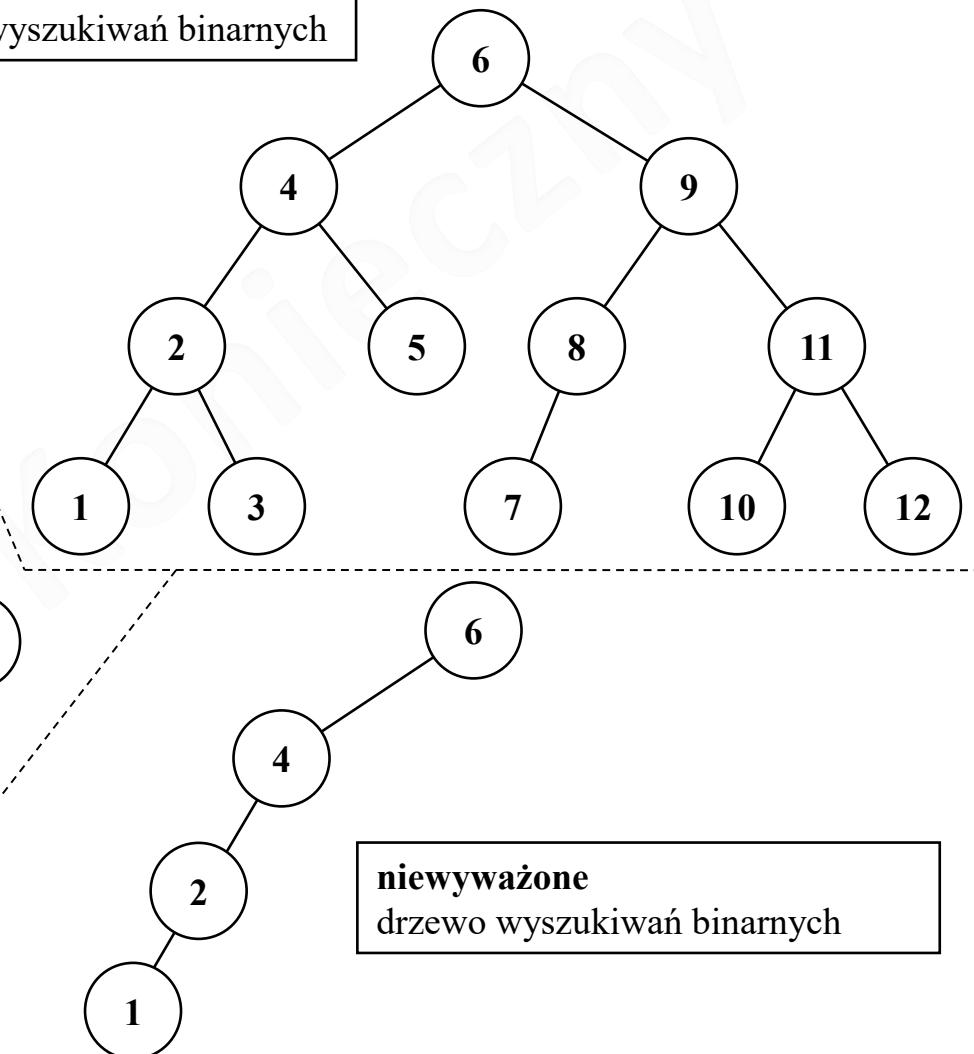
wyważone

drzewo wyszukiwań binarnych



pełne

drzewo wyszukiwań binarnych



niewyważone

drzewo wyszukiwań binarnych

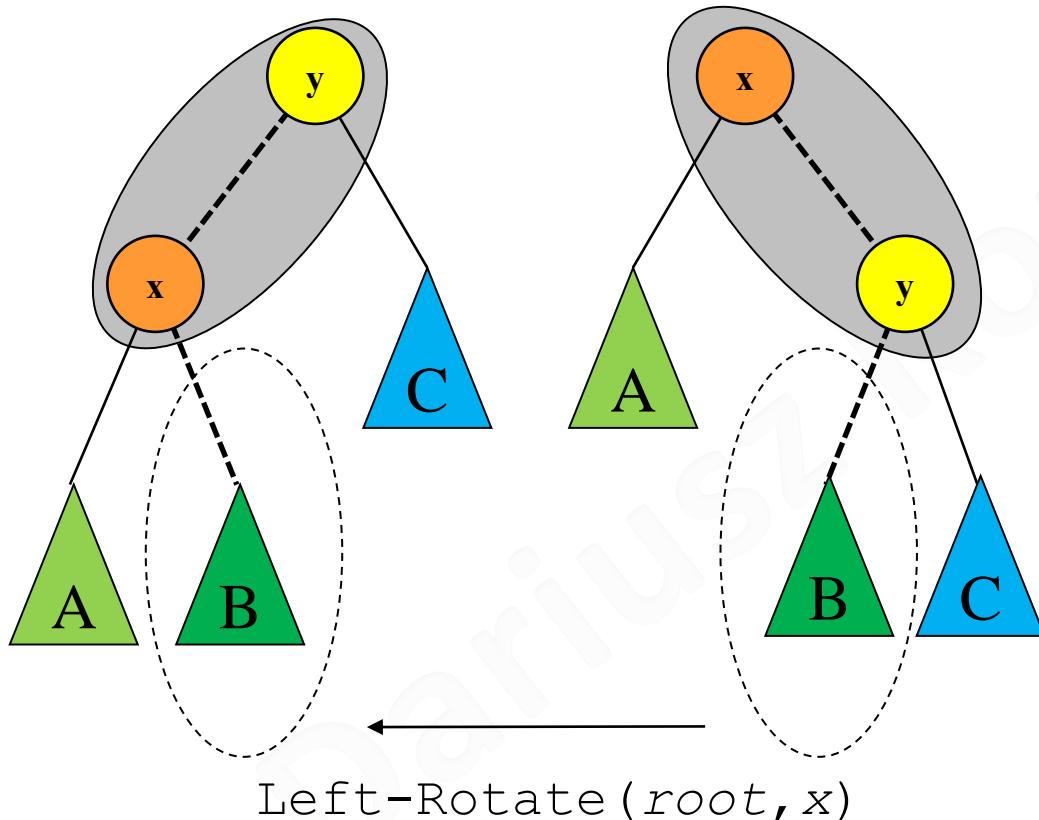
Drzewa samorównoważące się

- Większość operacji w drzewie binarnym **zależy od wysokości drzewa**, zatem pożądane jest utrzymanie małej wysokości drzewa.
- Podstawowe drzewo BST ma tą wadę, że w wielu przypadkach (np. dla wstawiania danych posortowanych) degeneruje się do listy wiązanej co powoduje wzrost kosztów operacji do wartości $O(n)$.
- **Samorównoważące się** drzewa rozwiążają ten problem poprzez ciąg modyfikacji drzewa (np. rotację) w kluczowych miejscach w celu redukcji wysokości drzewa. Chociaż powoduje to pewien narzut czasowy, jest to usprawiedliwione, gdyż skraca czas kolejnych operacji.
- Wysokość drzewa musi być co najmniej $\log n$, gdyż na poziomie k mieści się co najwyżej 2^k węzłów; dla pełnego drzewa jest to dokładnie taka liczba węzłów. Wyważone drzewo nie jest tak precyzyjnie wypełnione, gdyż utrzymywanie takiego zapełnienia po każdej operacji jest kosztowne ($O(n)$). Zamiast tego drzewa samorównoważące utrzymują wysokość drzewa na poziomie $O(\log n)$, czyli $c * \log n$, gdzie $1 < c \leq 2$. Powoduje to, że wszystkie podstawowe operacje (wyszukiwanie, wstawianie, usuwanie) są wykonywalne w czasie $O(\log n)$.

Rotacja w drzewie

- Rotacja w drzewie binarnym jest operacją zmieniającą strukturę drzewa **bez wpływu** na kolejność **in-order**.

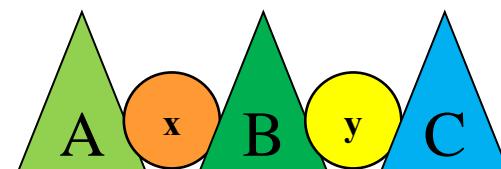
Right-Rotate(*root*, *y*)



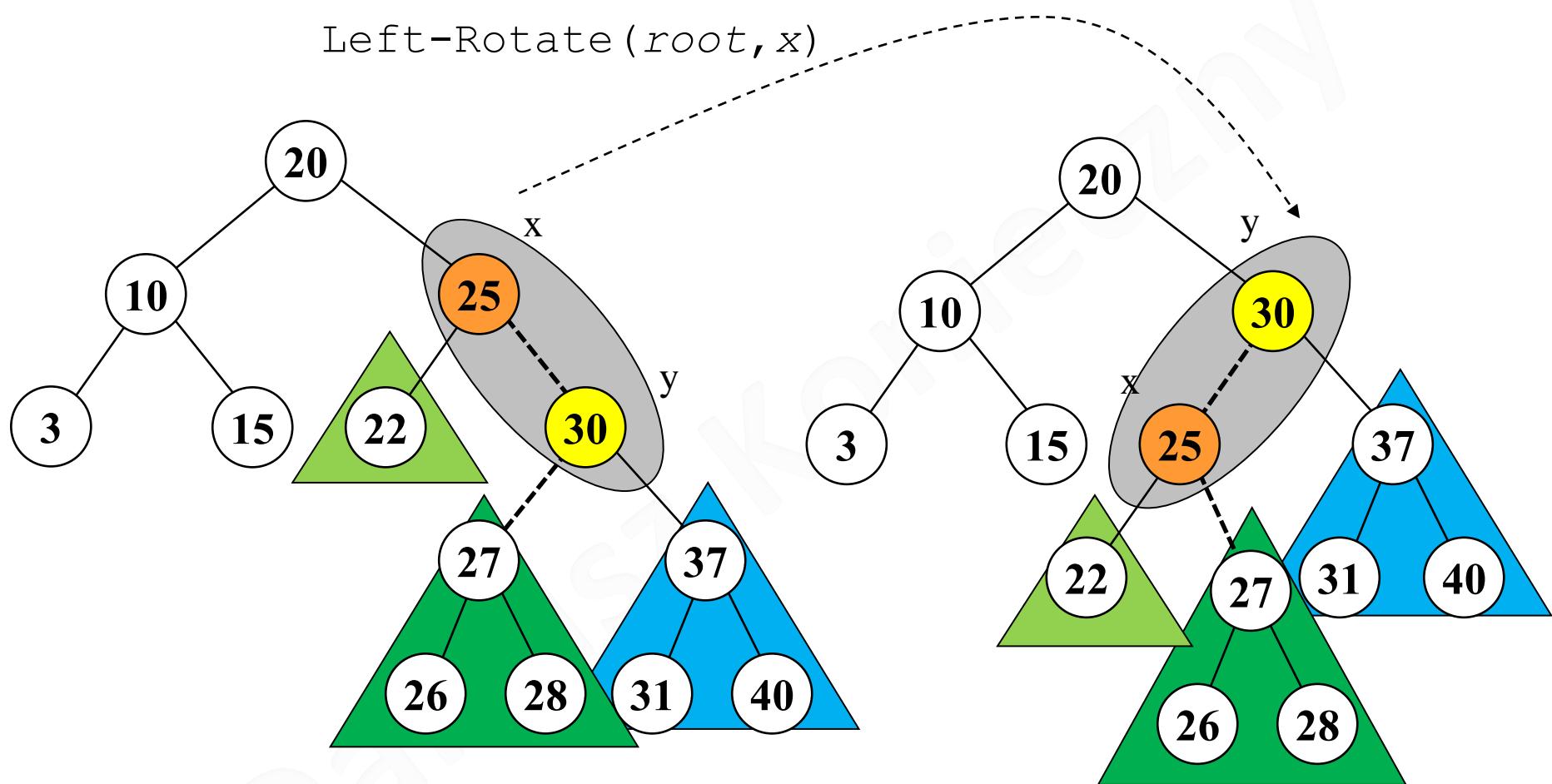
```
{ 1} Left-Rotate(root, x)
{ 2}   y := right[x]
{ 3}   right[x] := left[y]
{ 4}   if left[y] <> null
{ 5}     then p[left[y]] := x
{ 6}   p[y] := p[x]
{ 7}   if p[x] = null
{ 8}     then root := y
{ 9}   else if x = left[p[x]]
{10}     then left[p[x]] := y
{11}     else right[p[x]] := y
{12}   left[y] := x
      p[x] := y
```

in-order lewy

in-order prawy



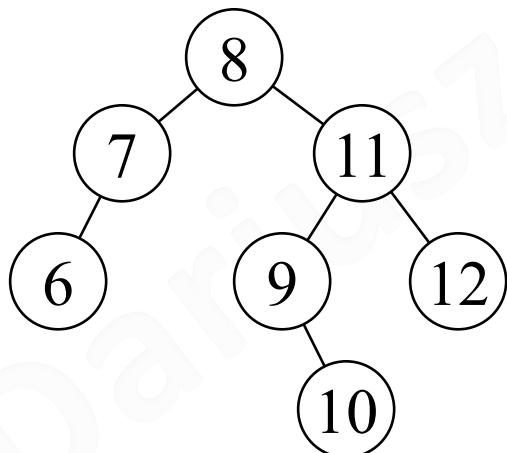
Rotacja - przykład



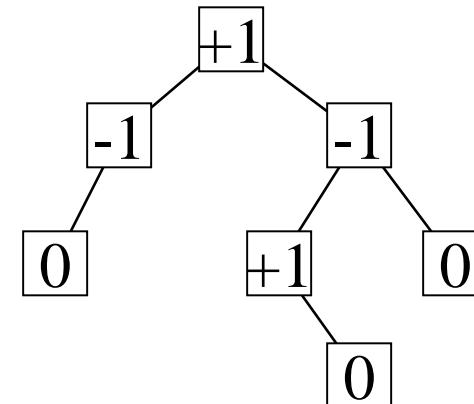
Drzewo AVL

- Nazwa AVL pochodzi o nazwisk jego twórców: G.M. Adelson-Velsky oraz E.M. Landis. Jest to drzewo BST z dodatkową informacją pomagającą w wyważaniu drzewa.
- W drzewie tym w każdym węźle pamiętamy **współczynnik wyważenia**, który jest różnicą między wysokością prawego poddrzewa i lewego poddrzewa tego węzła. Jeśli ten współczynnik jest **-1, 0, 1** poddrzewo w tym węźle uznajemy za **wyważone**. Jeśli podczas modyfikacji drzewa współczynnik ten będzie miał wartość -2 lub 2 drzewo takie wymaga wyważenia za pomocą rotacji. W celu otrzymania efektywnej implementacji, współczynnik wyważenia powinien być przechowywany w węźle.

Drzewo BST



Współczynniki wyważenia
przechowywane w węzłach



Drzewo AVL - analiza

Minimalna liczba węzłów w drzewie AVL wyraża się poniższą formułą rekurencyjną:

$$AVL_h = AVL_{h-1} + AVL_{h-2} + 1$$

gdzie:

$$AVL_0 = 0$$

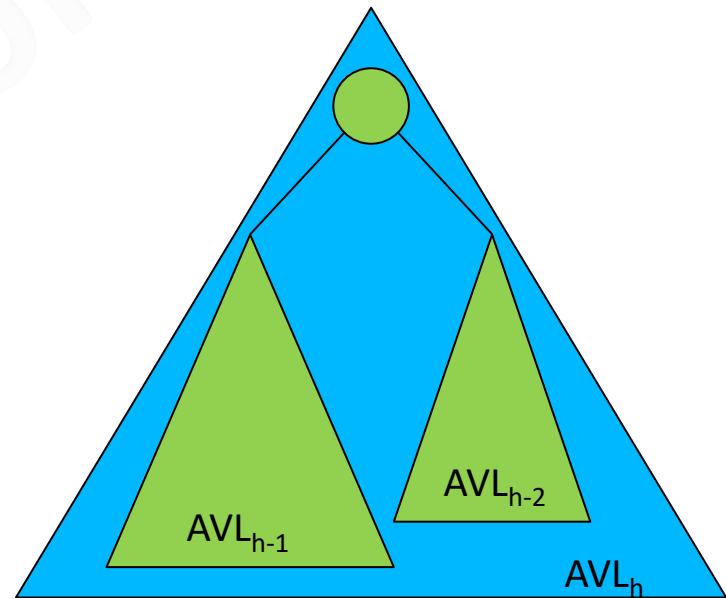
$$AVL_1 = 1$$

Stąd ograniczenie na wysokość h drzewa AVL wyraża się przez formułę (udowodnioną przez autorów) :

$$\lg(n+1) \leq h < 1,44 \lg(n+2) - 0,328$$

Czyli:

$$h = O(\lg n)$$



Drzewo AVL - podsumowanie

- Drzewo AVL potrzebuje przechowywać dodatkowo pięć różnych wartości w węźle: -2, -1, 0, 1, 2.
- Wysokość drzewa jest nie większa niż $1,5 * \log n$.
- Dzięki rotacjom wykonywanym podczas dodawania lub usuwania zachowuje poprawne współczynniki wyważenia.
- Naprawianie następuje na poziomach: od poziomu modyfikacji drzewa w kierunku korzenia. Na każdym poziomie wykonywane są maksymalnie dwie rotacje, czyli złożoność tych operacji jest $O(\log n)$.
- W bibliotekach standardowych wielu języków programowania zostało zastąpione przez drzewo czerwono-czarne, które mimo teoretycznie większej wysokości jest pesymistycznej, w praktyce działa szybciej.
- Na końcu tego wykładu znajdują się slajdy przedstawiające algorytmy dodawania i usuwania węzła w drzewie AVL w języku angielskim – **materiał nieobowiązujący na egzaminie**, dla chętnych.

Red-Black Tree (RB-tree)

Drzewo czerwono-czarne (ang. ***red-black tree, RB-Tree***) jest drzewem binarnym z jednym bitem w każdym węźle na dodatkową informację: **kolor** tego węzła, który może być **CZERWONY** (RED) lub **CZARNY** (BLACK). Przez dodanie ograniczenia na kolorowanie ścieżki od dowolnego korzenia do jego liści zapewnione zostanie, że **żadna ścieżka nie jest więcej niż dwukrotnie większa od każdej innej ścieżki**. Stąd można przyjąć, że drzewo jest zrównoważone.

W pierwszej realizacji każdy węzeł drzewa posiada pola *color*, *key*, *left*, *right*, and *p*. Jeśli dla danego węzła nie istnieje rodzic lub potomek, odpowiednie pole ma wartość **null**. To drzewo zakłada, że liśćmi są wartości **null** stąd traktuje się je jako zewnętrzne węzły, gdy węzły zawierające klucze traktuje się jako wewnętrzne węzły tegoż drzewa.

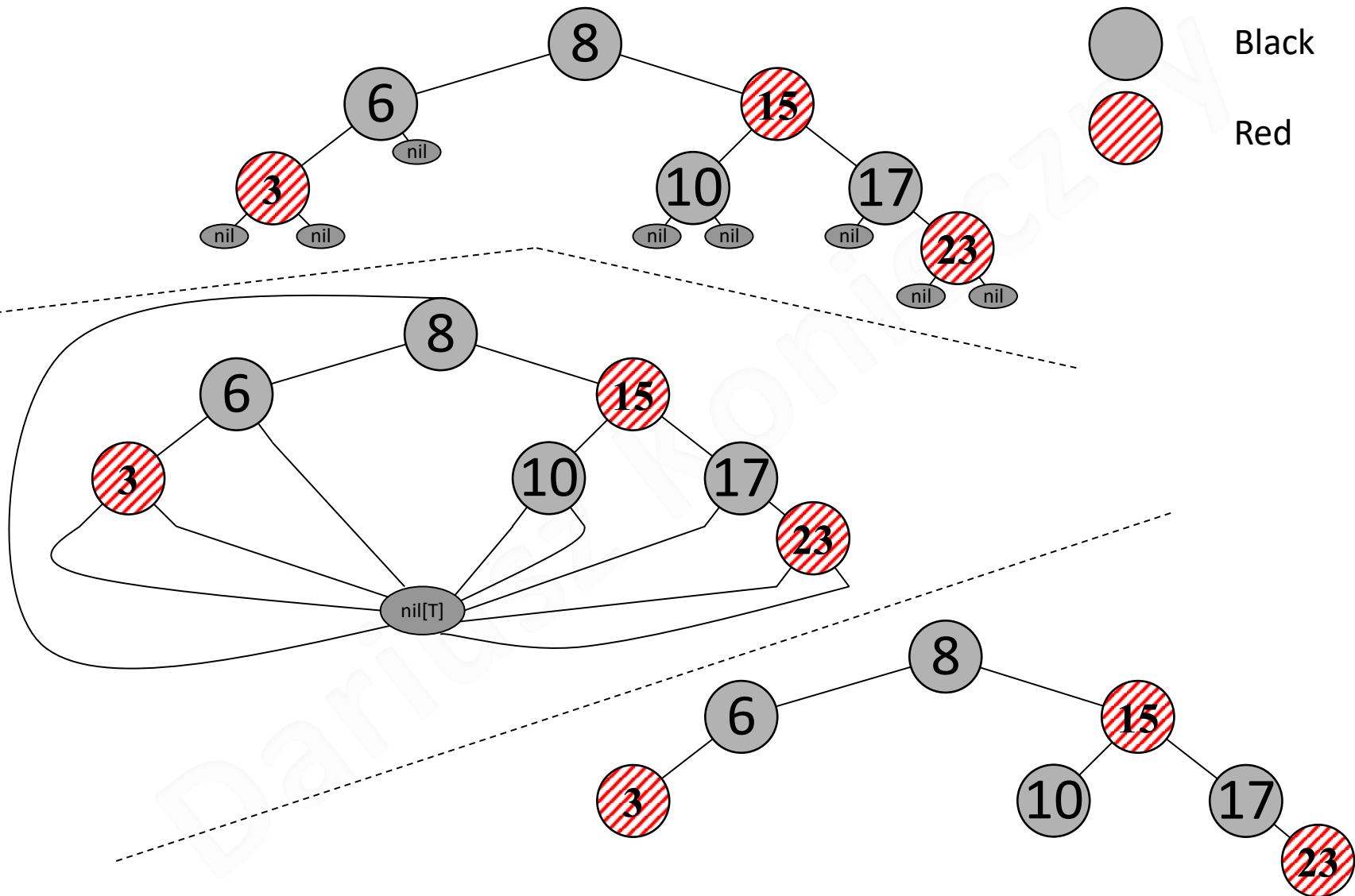
Drzewo binarne jest poprawnym drzewem czerwono-czarnym, jeśli spełnia poniższe **właściwości czerwono-czarne**:

- 1) Każdy węzeł jest albo czerwony, albo czarny.
- 2) Korzeń jest czarny.
- 3) Każdy liść (**null**) jest czarny.
- 4) Jeśli węzeł jest czerwony, jego obydwa potomkowie są czarni.
- 5) Dla każdego węzła, wszystkie ścieżki od tego węzła do liści w jego poddrzewie zawierają taką samą liczbę czarnych węzłów.

RB-Tree – czarna wysokość (bh)

- Liczbę czarnych węzłów na dowolnej ścieżce z węzła x (wykluczając węzeł x) do liścia nazywamy **czarną wysokością węzła**, która oznaczamy jako $bh(x)$.
- Przez czarną wysokość RB-drzewa będziemy rozumieć czarną wysokość jego korzenia.
- Fakt: Drzewo czerwono-czarne o n węzłach wewnętrznych ma wysokość co najwyżej $2\lg(n+1)$.
- Na kolejnych slajdach bh oznaczać będzie czarną wysokość do danego miejsca w drzewie.

RB-Tree - reprezentacji

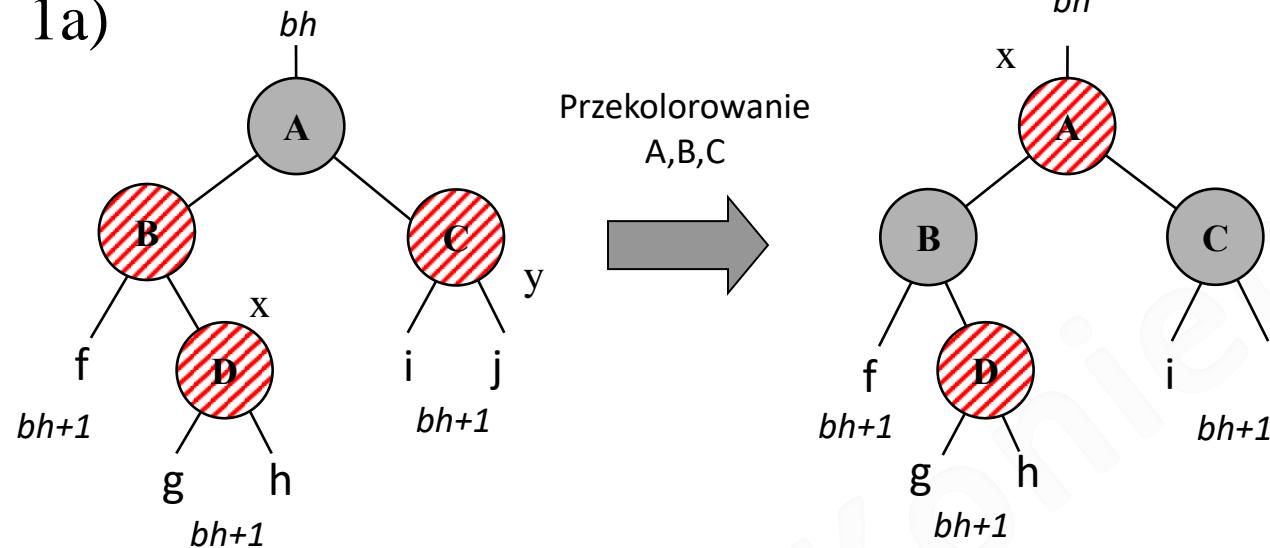


Wstawianie w RB-Tree 1/3

- Wstawianie polega na wstawieniu nowego węzła **jak do zwykłego drzewa binarnego** jako liść i zakolorowaniu węzła **na czerwono** (nie jest wtedy łamana własność 5, ale może być złamana własność 4).
- W przypadku złamania własności 4 następuje naprawienie drzewa od miejsca wstawienia poprzez przekolorowywanie i ewentualnie rotacje w drzewie.
- Reguły naprawiające na kolejnych slajdach prezentują przypadki, gdy węzeł x aktualnie analizowany jest w **lewy** poddrzewie swojego **dziadka**. Analogicznie będzie w symetrycznych przypadkach.

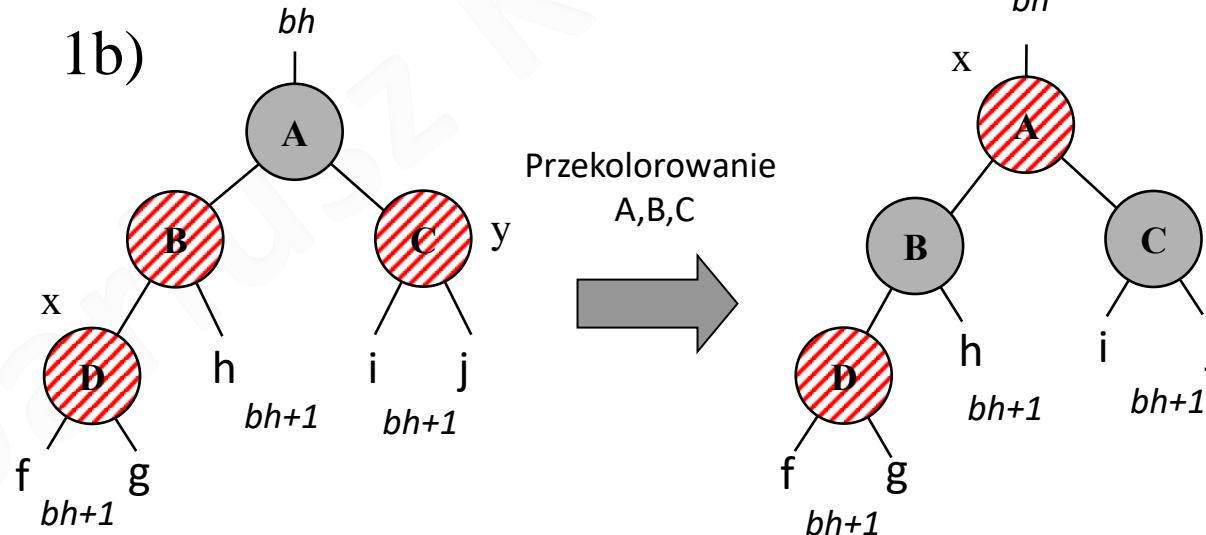
Wstawianie w RB-Tree 2/3

1a)



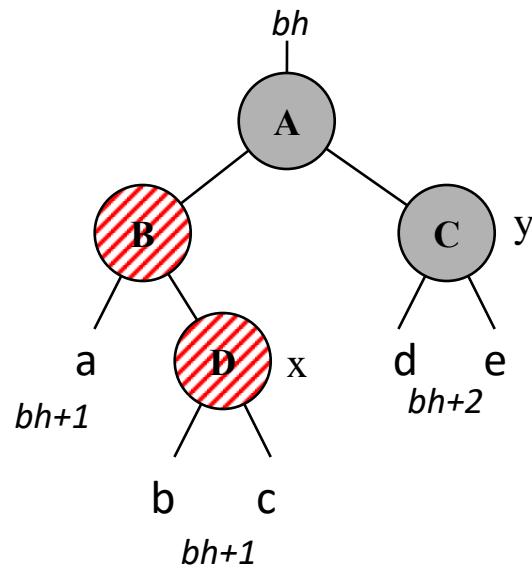
Przekolorowanie
A,B,C

1b)



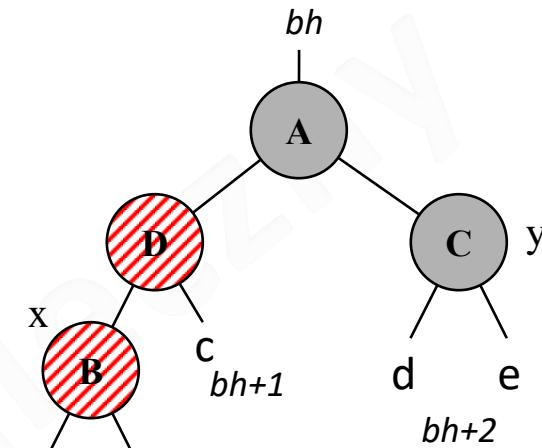
Wstawianie w RB-Tree 3/3

II)

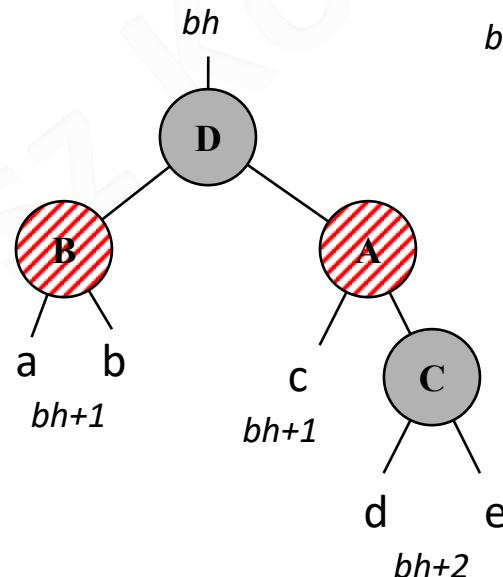


III)

Left-Rotate(B)



Right-Rotate(A),
Przekolorowanie A,D

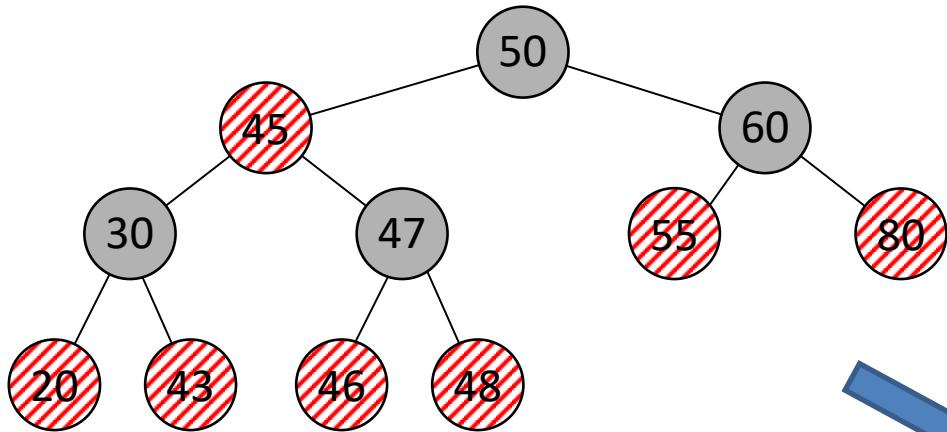


Stop

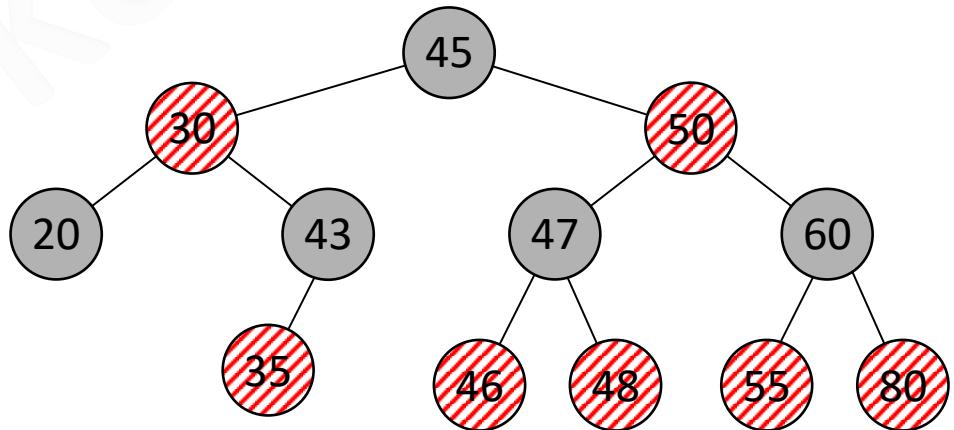
Wstawianie w RB-Tree - pseudokod

```
{ 1} RB-Insert(root, x)
{ 2}     Tree-Insert(root, x)
{ 3}     color[x] := RED
{ 4}     while x <> root and color[p[x]] = RED do
{ 5}         if p[x] = left[p[p[x]]] then
{ 6}             y := right[p[p[x]]]
{ 7}             if color[y] = RED then           // przypadek 1
{ 8}                 color[p[x]] := BLACK      // przypadek 1
{ 9}                 color[y] := BLACK        // przypadek 1
{10}                color[p[p[x]]] := RED       // przypadek 1
{11}                x := p[p[x]]
{12}            else
{13}                if x = right[p[x]] then
{14}                    x := p[x]           // przypadek 2
{15}                    Left-Rotate(root, x) // przypadek 2
{16}                    color[p[x]] := BLACK    // przypadek 3
{17}                    color[p[p[x]]] := RED      // przypadek 3
{18}                    Right-Rotate(root, p[p[x]]) { case 3}
{19}            else .... { to samo co po then z zamianą „right” oraz
{20}                         „left”}
{21}     color[root] := BLACK
```

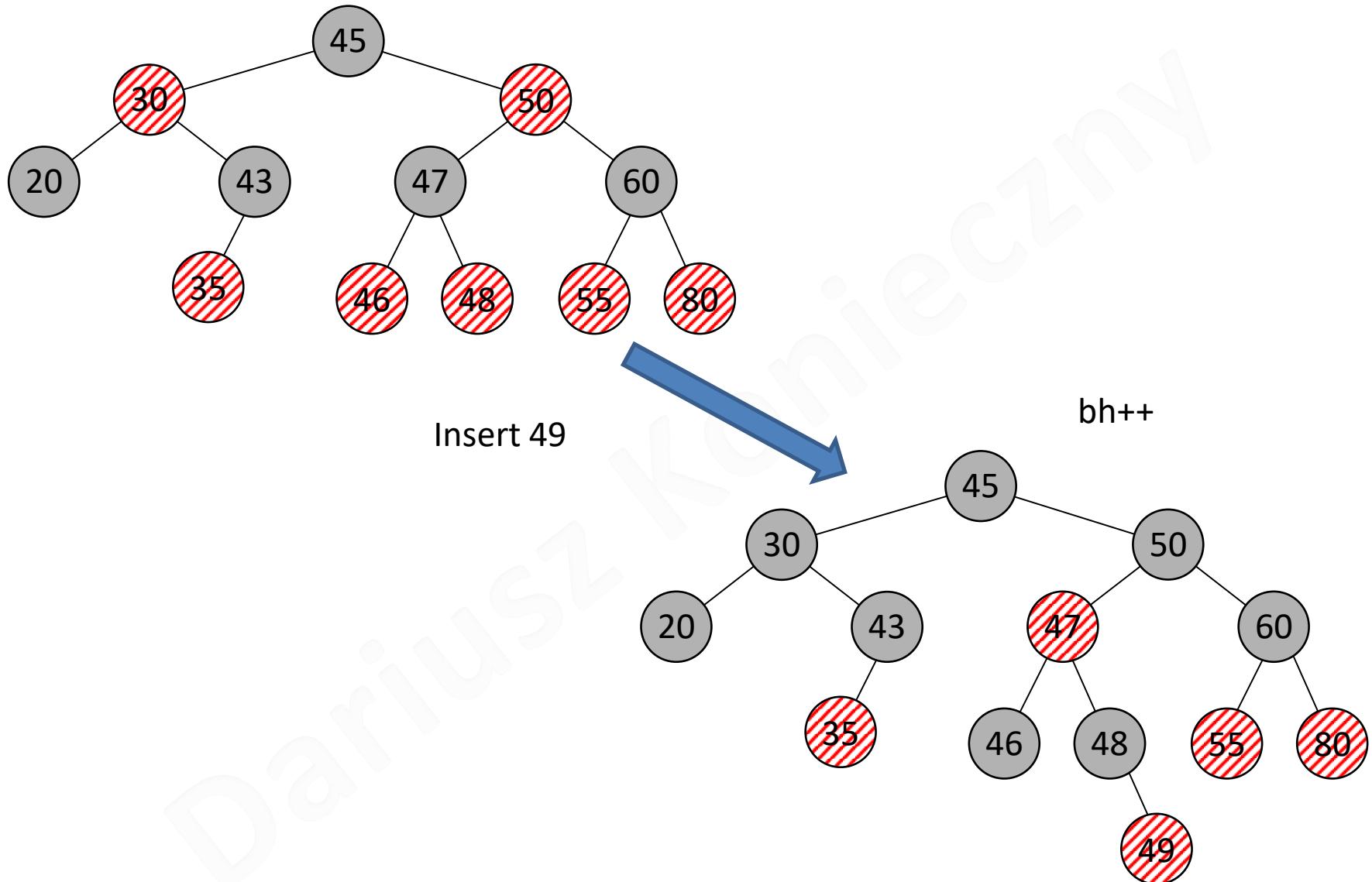
Wstawianie do RB-Tree – przykład 1/3



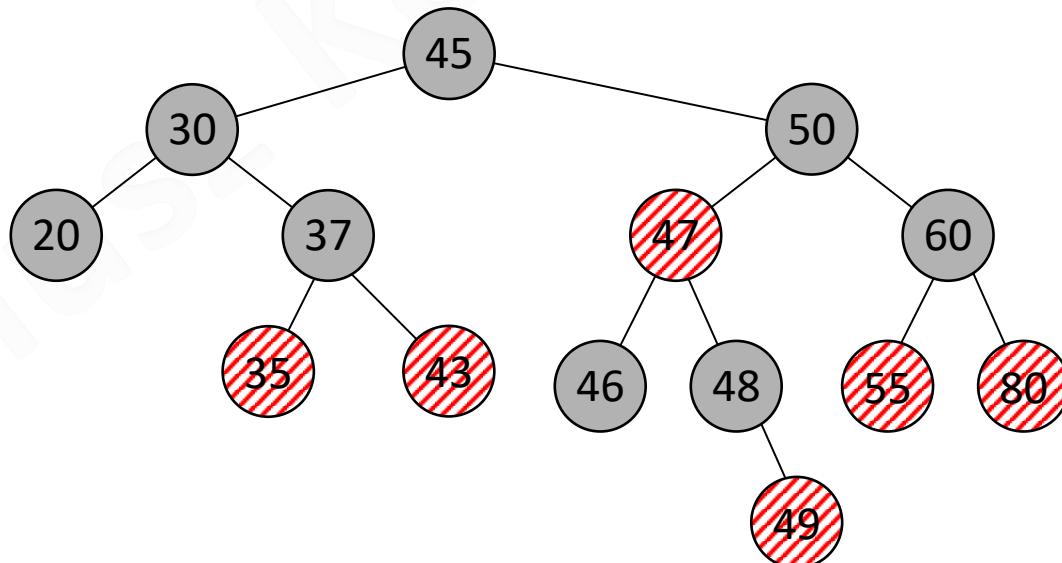
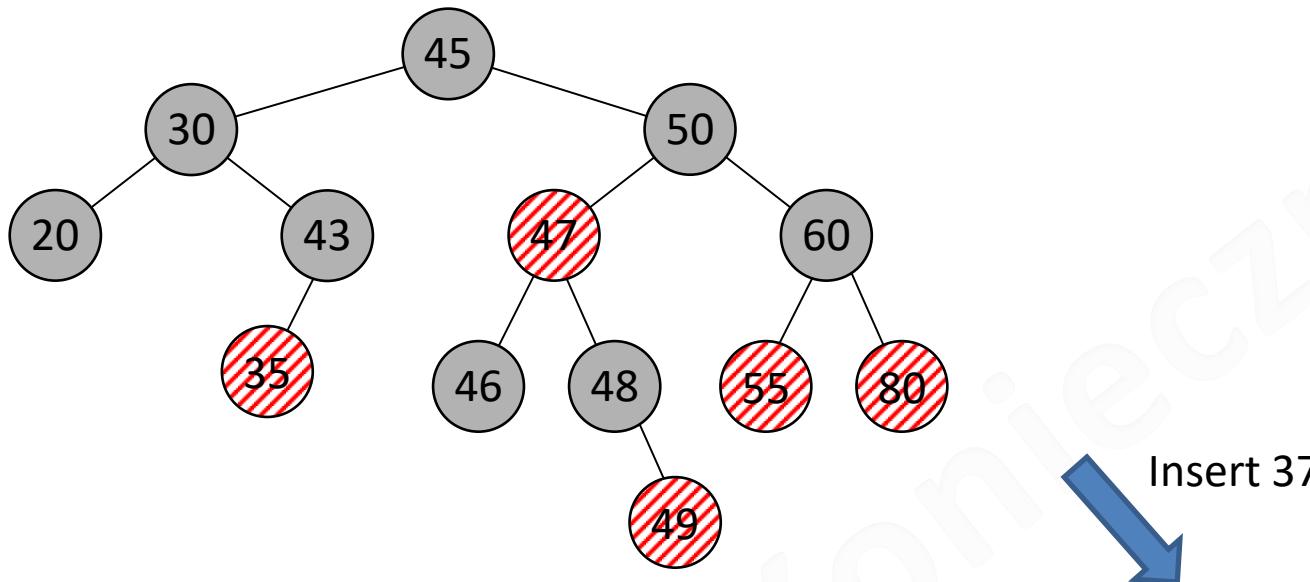
Insert 35



Wstawianie do RB-Tree – przykład 2/3



Wstawianie do RB-Tree – przykład 3/3



Usuwanie z RB-Tree

- Procedura RB-Delete jest podobna do Tree-Delete, ale:
 - Wszystkie referencje do **null** są zastąpione przez referencję do strażnika $\text{NULL}[T]$
 - Sprawdzenie, czy x jest **null** jest usunięte i przyporządkowanie $p[x] := p[y]$ jest wykonywane zawsze. Zatem, jeśli x jest strażnikiem $\text{NULL}[T]$, jego referencja na rodzica wskazuje na rodzica usuniętego węzła y .
 - Po usunięciu, jeśli y jest **czarny**, wykonywane jest wywołanie procedury RB-Delete-Fixup, ponieważ pewne właściwości czerwono-czarne mogą być naruszone.

Usuwanie z RB-Tree - pseudokod

```
RB-Delete(T, z)
{ 1}   if (left[z] = nil[T]) or (right[z] = nil[T])
{ 2}     then y := z
{ 3}     else y := Tree-Successor(z)
{ 4}   if left[y] <> nil[T]
{ 5}     then x := left[y]
{ 6}     else x := right[y]
{ 7}   p[x] = p[y]
{ 8}   if p[y] = nil[T]
{ 9}     then root[T] := x
{10}    else if y = left[p[y]]
{11}      then left[p[y]] := x
{12}      else right[p[y]] := x
{13}   if y <> z
{14}     then swap(key[z], key[y])
{15}       { if node y has another fields, copy them here }
{16}   if color[y] = BLACK
{17}     then RB-Delete-Fixup(T, x)
return y
```

Naprawianie po usunięciu

W momencie wypinania węzła y w RB-Delete, mogą się pojawić trzy problemy:

- 1) Węzeł y był korzeniem (`root`) i czerwone dziecko y zostało nowym korzeniem (właściwość 2 jest naruszona)
- 2) Obydwa węzły x oraz $p[y]$ (teraz również $p[x]$) są czerwone (właściwość 4 jest naruszona)
- 3) usunięcie węzła y spowodowało, że każda ścieżka, która poprzednio zawierała y ma obecnie o jeden za mało czarnych węzłów (właściwość 5 is naruszona)

Możemy naprawić problem 3) poprzez stwierdzenie, że węzeł x ma „dodatkowy” czarny token: CZERWONY x jest **CZERWONY-I-CZARNY**, CZARNY x jest **PODWÓJNIE-CZARNY**.

Przemerzamy (w specyficzny sposób) drzewo w kierunku korzenia poszukując czerwonego węzła, którego możemy przekolorować na czarny.

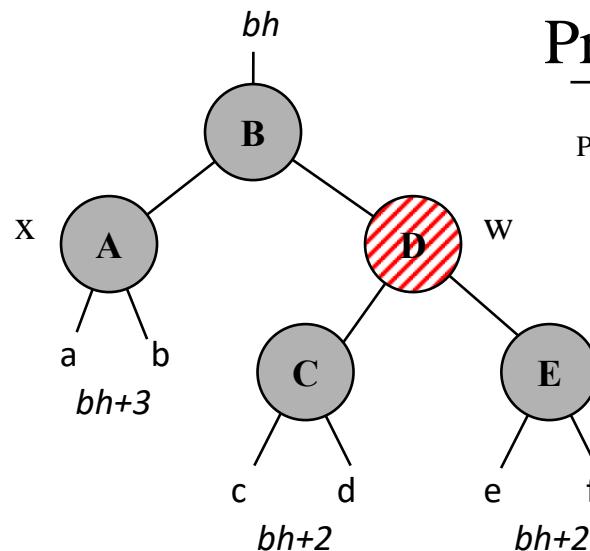
Sposób działania będzie zależał **od koloru brata** analizowanego węzła oraz od kolorów jego dzieci.

x – węzeł analizowany z „dodatkowym” czarnym tokenem

w – brat węzła x

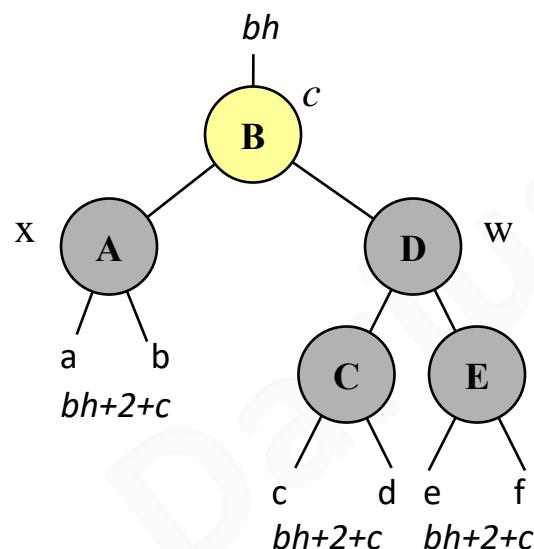
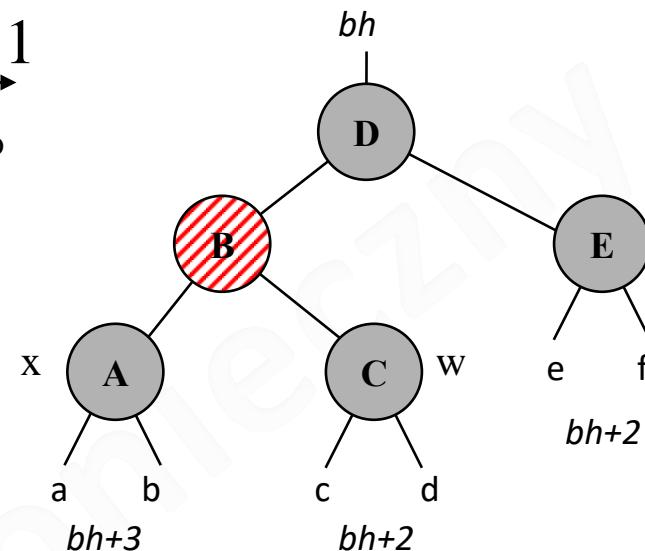
Przedstawione tylko przypadki, gdy x jest w lewym poddrzewie swojego ojca.

Naprawianie RB-Tree – przypadek 1 i 2



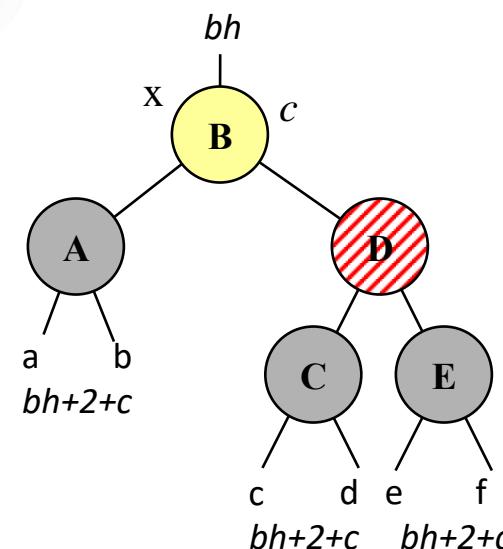
Przypadek 1

Left_rotate(B),
Przekolorowanie B,D

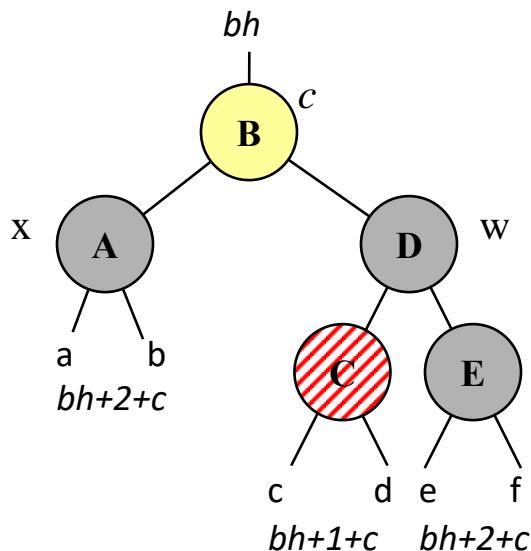


Przypadek 2

Przekolorowanie D

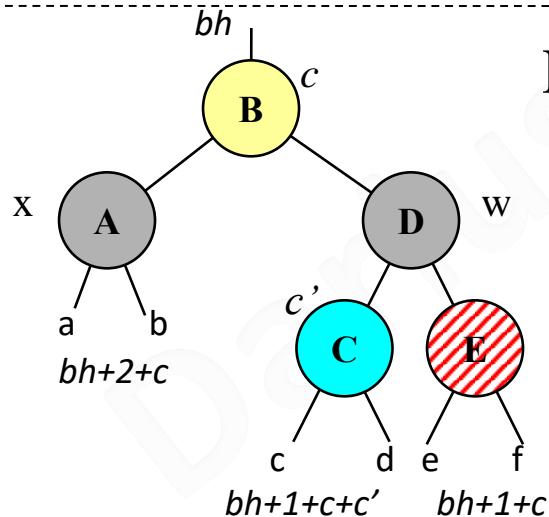
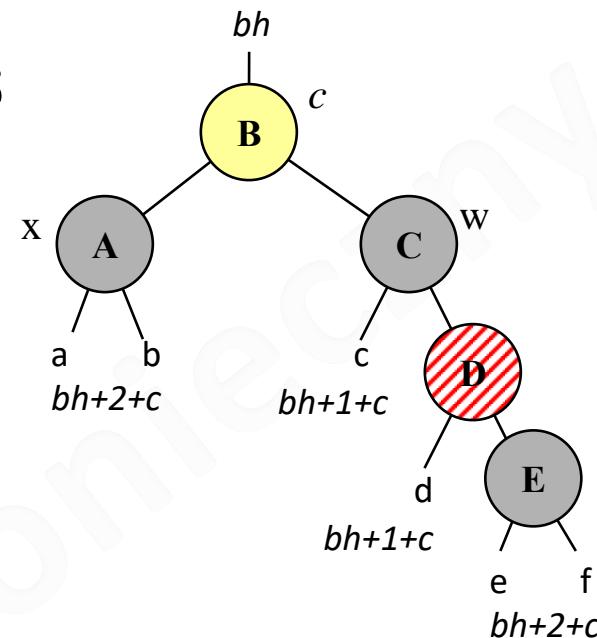


Naprawianie RB-Tree – przypadek 3 i 4



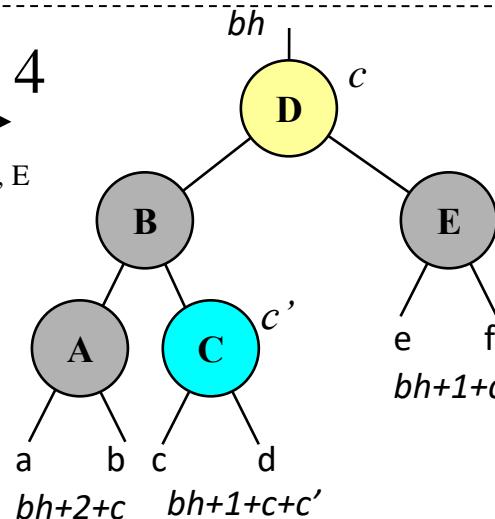
Przypadek 3

Right_rotate(D),
Przekolorowanie C,D



Przypadek 4

Left_rotate(B),
Przekolorowanie B,D, E

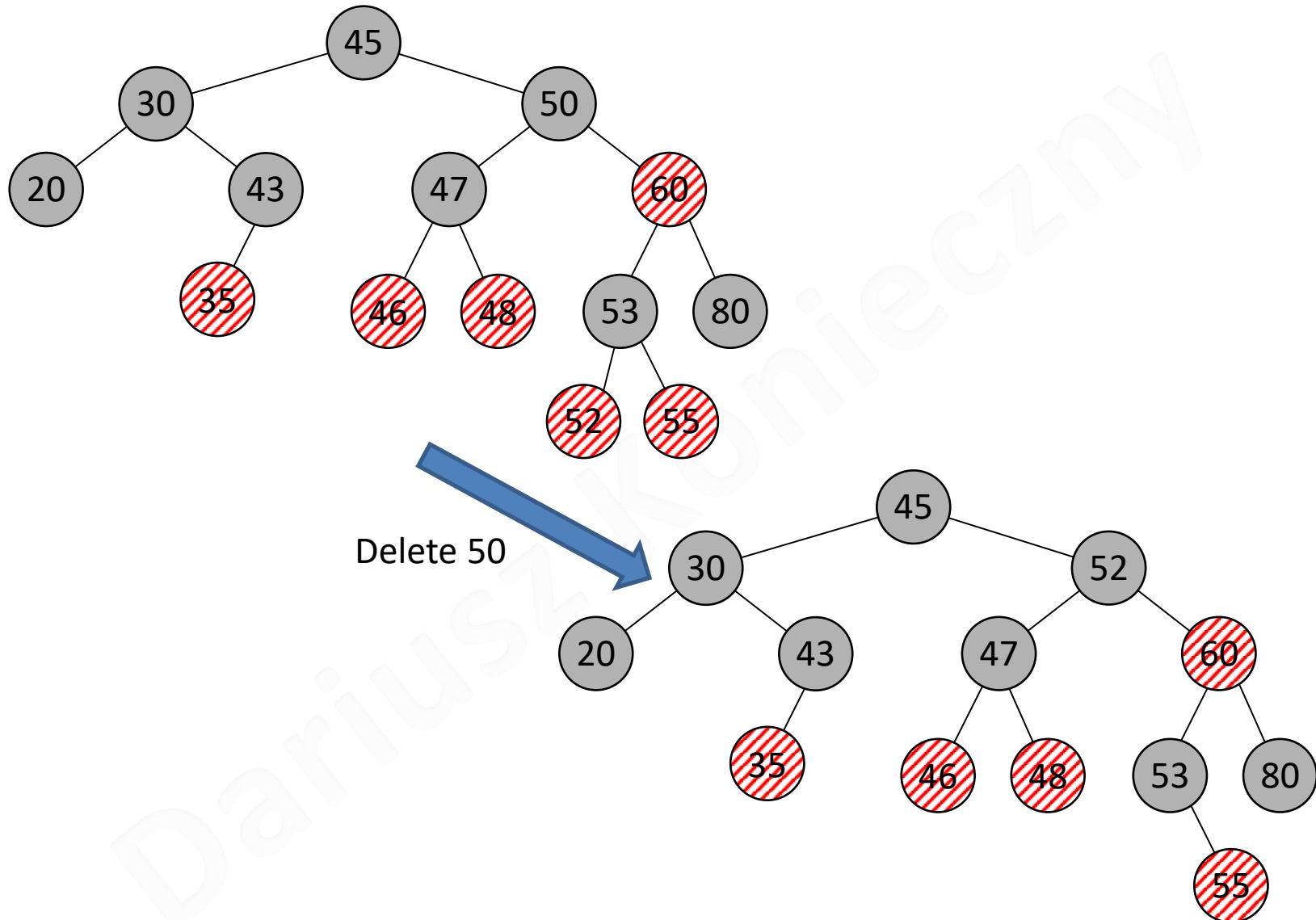


x = root [T]

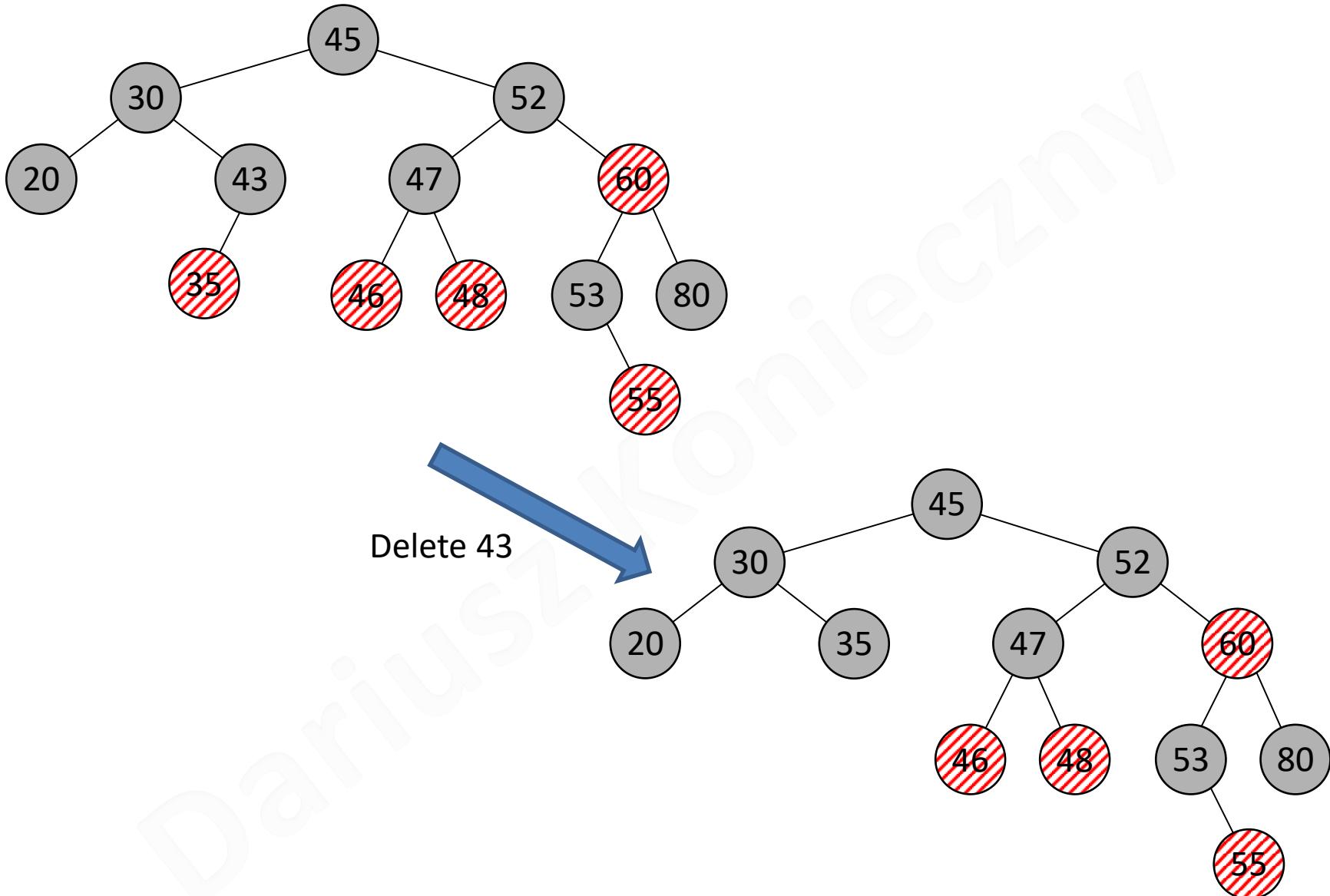
Naprawianie RB-Tree - pseudokod

```
{ 1} RB-Delete-Fixup(T, x)
{ 2}   while x!= root[T] and color[x]=BLACK
{ 3}     do if x=left[p[x]] then
{ 4}       w:= right[p[x]]
{ 5}       if color[w]=RED then          // przypadek 1
{ 6}         color[w]:=BLACK           // przypadek 1
{ 7}         color[p[x]]:=RED          // przypadek 1
{ 8}         Left-Rotate(T,p[x])      // przypadek 1
{ 9}         w:=right[p[x]]          // przypadek 1
{10}       if color[left[w]]=BLACK and color[right[w]]=BLACK then
{11}         color[w]:=RED           // przypadek 2
{12}         x:=p[x]                 // przypadek 2
{13}       else if color[right[w]]=BLACK then
{14}         color[left[w]]:=BLACK    // przypadek 3
{15}         color[w]:=RED           // przypadek 3
{16}         Right-Rotate(T,w)       // przypadek 3
{17}         w:=right[p[x]]          // przypadek 3
{18}         color[w]:=color[p[x]]    // przypadek 4
{19}         color[p[x]]:=BLACK       // przypadek 4
{20}         color[right[w]]:=BLACK    // przypadek 4
{21}         Left-Rotate(T,p[x])      // przypadek 4
{22}         x:=root[T]              // przypadek 4
{23}   else /* */
{24}   color[x]=BLACK
```

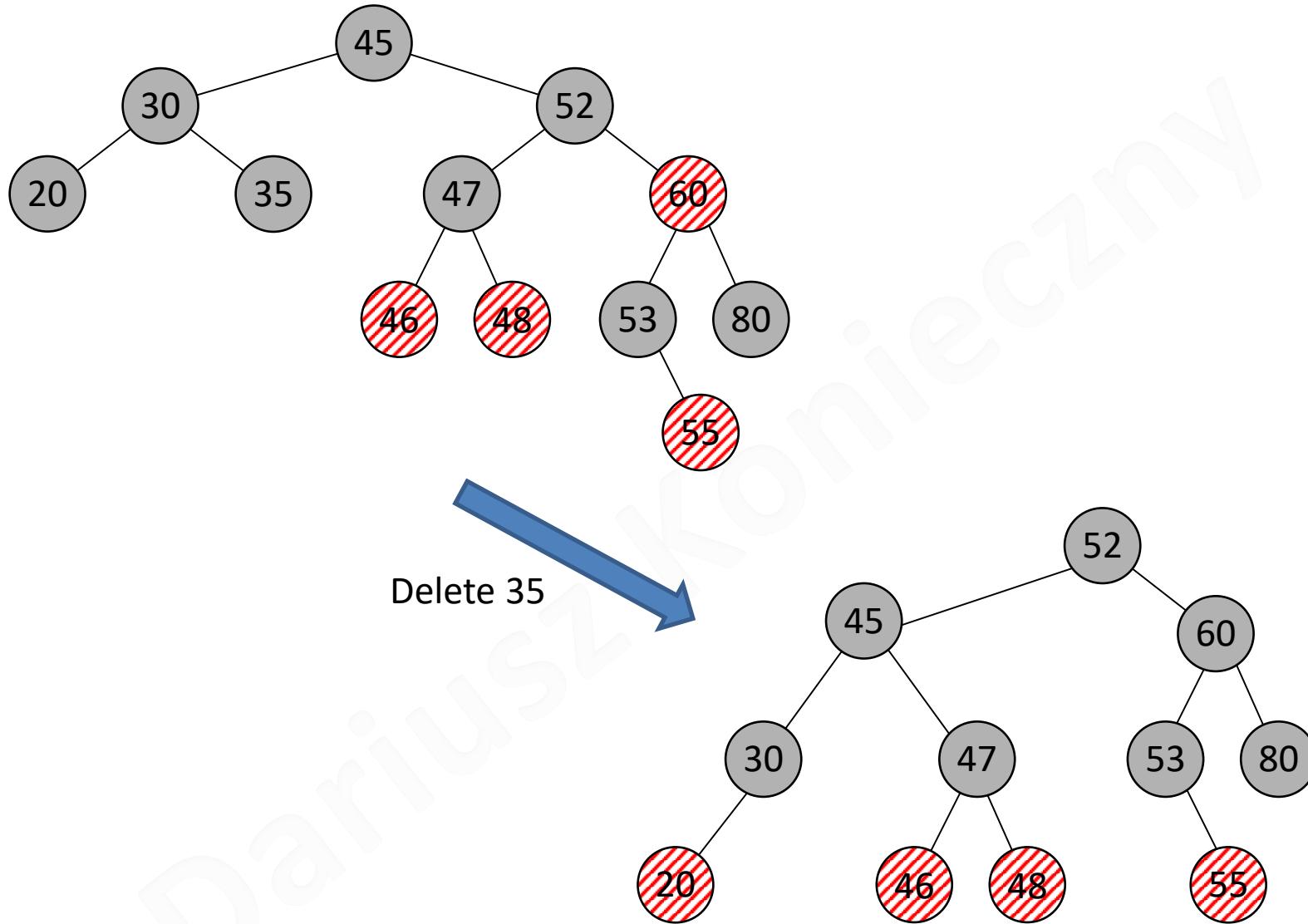
Usuwanie z RB-Tree – przykład 1/3



Usuwanie z RB-Tree – przykład 2/3



Usuwanie z RB-Tree – przykład 3/3

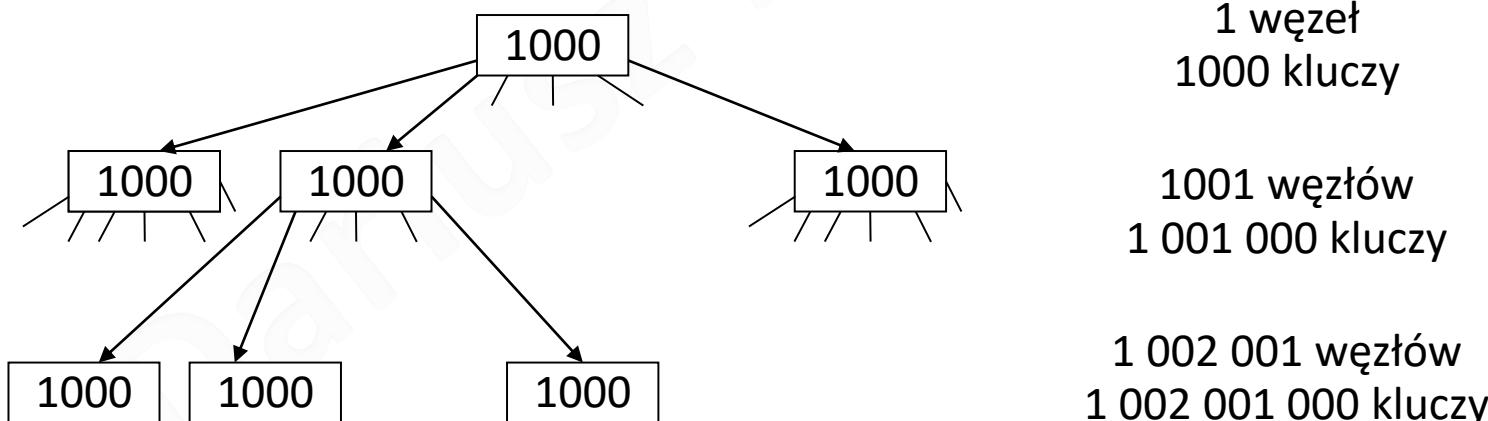
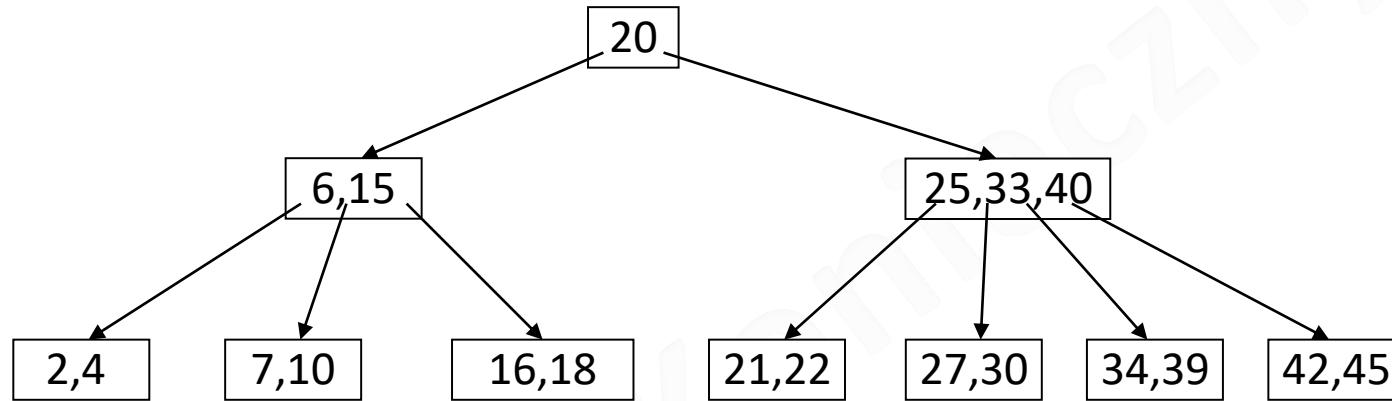


RB-Tree - złożoność

- Wstawianie:
 - Wstawianie jak do BST: $O(h)$
 - Naprawa - $O(h)$ przekolorowań i 2 rotacje: $O(h)$
 - Złożoność pesymistyczna: $O(h)=O(\log n)$
- Usuwanie:
 - Usuwanie jak z BST: $O(h)$
 - Naprawa – $O(h)$ przekolorowań i 2 rotacje: $O(h)$
 - Złożoność pesymistyczna: $O(h)=O(\log n)$
- Wszystkie inne operacje zależące od wysokości drzewa – złożoność $O(\log n)$
- Klasa `TreeMap<K, V>` jest zaimplementowana z użyciem drzew czerwono-czarnych wg algorytmu Cormena i in. przedstawionego na tym wykładzie.

B-Drzewo – idea, przykład

- Rozwińmy ideę drzewa binarnego przeszukiwań do drzew, które będą miały więcej niż dwójkę potomków, jednak nadal mają być, w miarę możliwości, regularne.



B-Drzewo – gdzie/kiedy/po co

- Mała wysokość
- Duża liczba kluczy
- Pamięć zewnętrza: dysk jest podzielony na sektory
 - Operacje o długim czasie działania (długoczasowe):
 - Odczyt z dysku: DISK-READ
 - Zapis na dysk: DISK-WRITE
 - Operacje o szybkim czasie działania(krótkoczasowe):
 - Operacja na CPU takie jak porównanie kluczy, przypisanie wartości itp.

B-drzewo - definicja

B-tree T jest drzewem ukorzenionym (którego korzeniem jest $\text{root}[T]$) posiadające następujące właściwości:

1) Każdy węzeł x posiada następujące pola:

- a) $n[x]$ – liczba kluczy przechowywanych aktualnie w węźle x ,
- b) $n[x]$ pól kluczy, przechowywane w porządku niemalejącym, czyli $\text{key}_1[x] \leq \text{key}_2[x] \leq \dots \leq \text{key}_{n[x]}[x]$,
- c) $\text{leaf}[x]$ – wartość logiczna równa TRUE jeśli x jest liściem oraz FALSE jeśli x jest wewnętrznym węzłem.

2) Każdy wewnętrzny węzeł x zawiera również $n[x]+1$ wskaźników $c_1[x], c_2[x], \dots, c_{n[x]+1}[x]$ do swoich potomków. Węzeł liść nie posiada potomków, więc jego pola c_i są nieokreślone.

3) Klucze $\text{key}_i[x]$ oddzielają zakresy kluczy, które są przechowywane w każdym poddrzewie: jeśli k_i jest dowolnym kluczem przechowywanym w poddrzewie zakorzenionym w $c_i[x]$, to

$$k_1 \leq \text{key}_1[x] \leq k_2 \leq \text{key}_2[x] \leq \dots \leq \text{key}_{n[x]}[x] \leq k_{n[x]+1}.$$

4) Wszystkie liście mają tą samą głębokość, która jest wysokością drzewa, czyli h .

5) Istnieje dolne i górne ograniczenie na liczbę kluczy przechowywanych w węźle. To ograniczenie można wyrazić z użyciem stałej całkowitej $t \geq 2$ nazywanej **minimalnym stopniem** B-drzewa:

- a) Każdy węzeł inny niż korzeń musi posiadać co najmniej $t - 1$ kluczy. Stąd każdy wewnętrzny węzeł oprócz korzenia ma co najmniej t potomków. Jeżeli drzewo nie jest puste, korzeń musi mieć co najmniej jeden klucz.
- b) Każdy węzeł może posiadać co najwyżej $2t - 1$ kluczy. Stąd każdy wewnętrzny węzeł może posiadać co najwyżej $2t$ dzieci. Mówimy, że węzeł jest **pełny**, jeśli zawiera dokładnie $2t - 1$ kluczy.

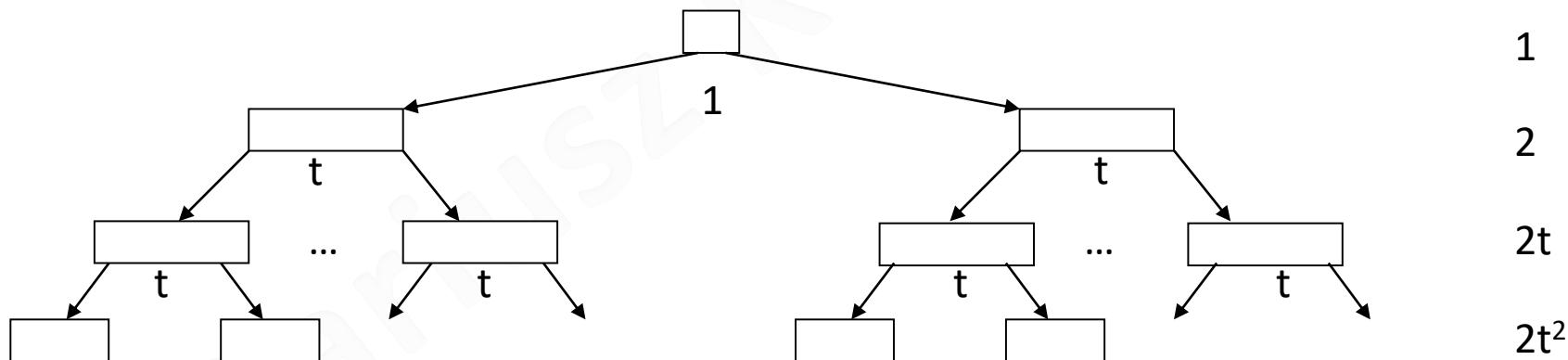
Jeżeli $t=2$ (minimalny stopień), takie drzewo nazywamy **2-3-4 drzewem**.

B-drzewo – właściwość wysokości

- Liczba operacji dyskowych potrzebnych dla większości operacji na B-drzewie jest proporcjonalna do wysokości B-drzewa.

Jeśli $n \geq 1$, to dla dowolnego B-drzewa T z n kluczami o wysokości h i minimalnym stopniu $t \geq 2$, zachodzi:

$$h \leq \lg_t \frac{n+1}{2} \quad \Rightarrow \quad h = O(\lg_t n)$$



B-drzewo - przeszukiwanie

- Idea: podobna do BST, tylko w każdym węźle jest więcej kluczy do porównania
- Procedura zwraca parę: węzeł i indeks w ramach tego węzła, gdzie znajduje się szukany klucz. W przypadku niepowodzenia zwraca **null**.

```
{ 1} B-Tree-Search (x, k)
{ 2} i:=1
{ 3} while i ≤ n[x] and k > keyi[x] do
{ 4}   i:=i + 1
{ 5} if i != n[x] and k = keyi[x] then
{ 6}   return (x, i)
{ 7} if leaf [x] then
{ 8}   return null
{ 9} else
{10}   DISK-READ (ci [x])
        return B-TREE-SEARCH (ci [x], k)
```

Dostęp do stron dysku: $\Theta(h) = \Theta(\lg_t n)$

Czas CPU: $O(th) = \Theta(t \lg_t n)$

B-tree - Tworzenie

- Budując B-drzewo T , najpierw uruchamiane jest B_Tree_Create w celu stworzenia pustego korzenia a następnie będzie uruchomione B_Tree_Insert aby dodawać nowe klucze. Obydwie procedury używają pomocniczej procedury $Allocate_Node$, która alokuje jedną stronę dyskową, która zostanie użyta jako nowy węzeł, w czasie $O(1)$. Można założyć, że tworzenie węzła przez $Allocate_Node$ nie wymaga operacji $DISK_READ$, gdyż nie było wcześniej żadnej użytecznej informacji nt. wezła przechowywanej na dysku.

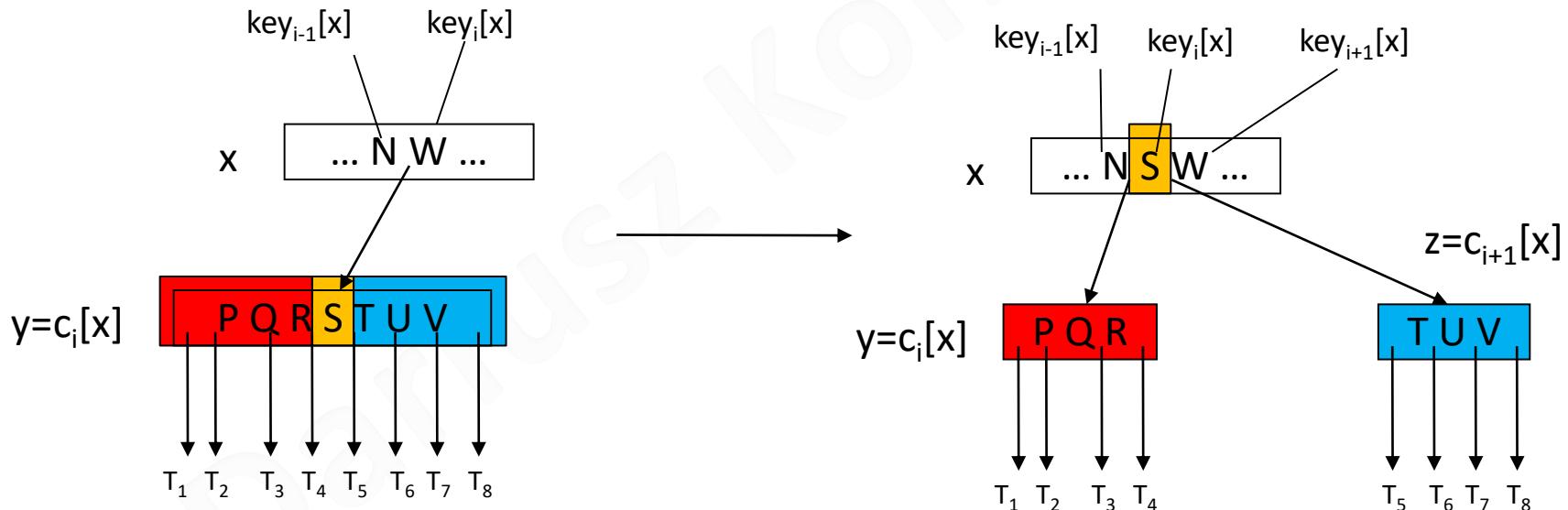
```
{ 1} B-Tree-Create (T)
{ 2}   x := Allocate_Node ()
{ 3}   leaf[x]=true
{ 4}   n[x]=0
{ 5}   DISK_WRITE (x)
{ }   root[T] := x
```

Dostęp do stron dysku : $O(1)$

Czas CPU: $O(1)$

B-drzewo – Podział potomka

- Procedura B-TREE-SPLIT-CHILD otrzymuje na wejściu *niepełny* węzeł wewnętrzny x (z założenia znajduje się on już w pamięci), indeks i , oraz węzeł y (z założenia znajduje się on również już w pamięci) taki, że $y = c_i[x]$ jest *pełnym* potomkiem x . Procedura następnie dzieli tego potomka na dwie części, dostosowując x aby zawierał dane tego dodatkowego potomka.



B-drzewo – Podział potomka - kod

```
B_Tree_Split_Child(x,i,y)
{ 1} z := Allocate_Node()
{ 2} leaf[z]:=leaf[y]
{ 3} n[z]:=t - 1
{ 4} for j:=1 to t - 1 do
{ 5}     keyj[z] := keyj+t[y]
{ 6} if not leaf [y] then
{ 7}     for j:=1 to t do
{ 8}         cj[z]:=cj+t[y]
n[y]:=t - 1
{ 9} for j:=n[x] + 1 downto i + 1 do
{10}    cj+1[x]:=cj[x]
{11} ci+1[x]:=z
{12} for j:=n[x] downto i do
{13}     keyj+1[x]:=keyj[x]
{14} keyi[x]:=keyt[y]
{15} n[x]:=n[x] + 1
{16} DISK-WRITE(y)
{17} DISK-WRITE(z)
{18} DISK-WRITE(x)
```

Dostęp do stron dysku : O(1)

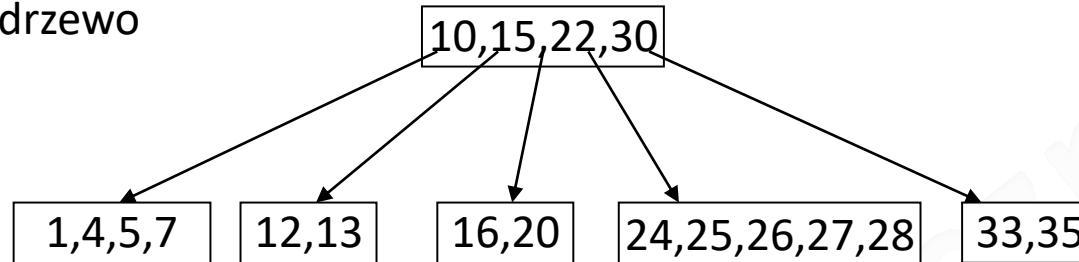
Czas CPU: $\Theta(t)$

B-drzewo – wstawianie – przykład 1/2

Początkowe drzewo

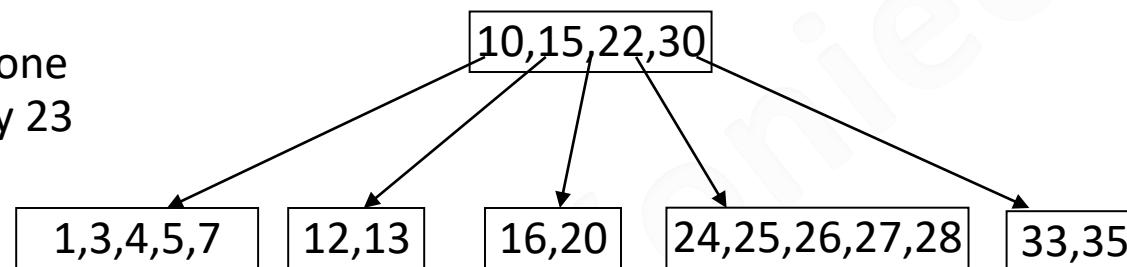
$t=3$

Wstawmy 3



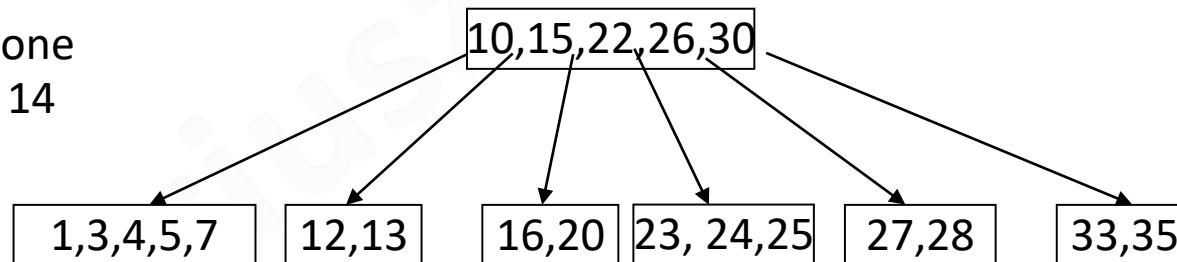
3 wstawione

Wstawmy 23



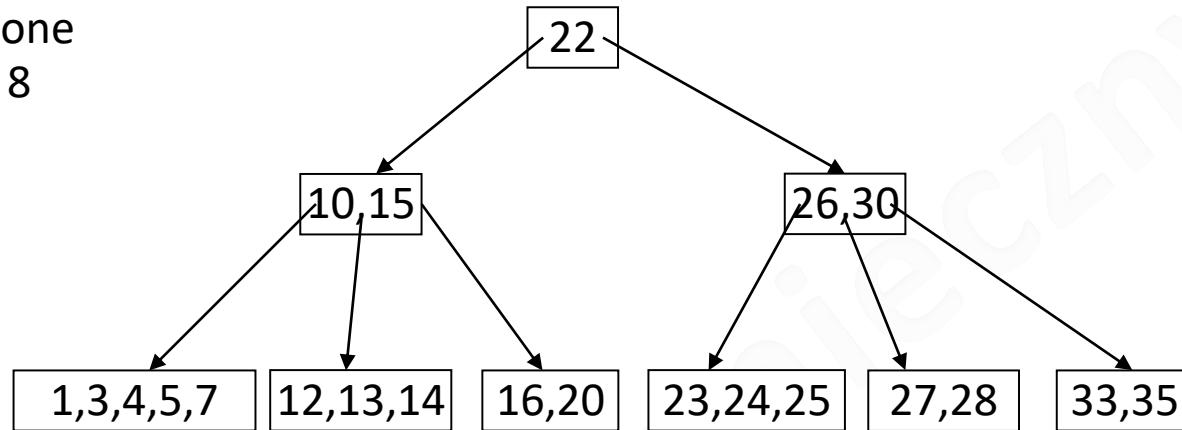
23 wstawione

Wstawmy 14

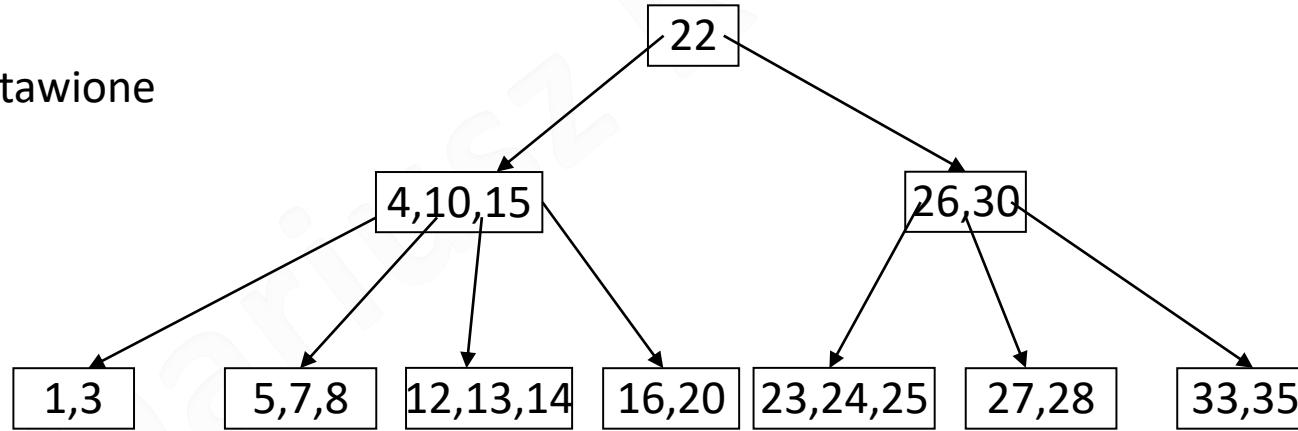


B-drzewo – wstawianie – przykład 2/2

14 wstawione
Wstawmy 8



8 wstawione



B-drzewo - wstawianie

- W celu podziału korzenia najpierw staje się on dzieckiem nowego pustego korzenia, aby można było wywołać procedurę B-TREE-SPLITCHILD. Drzewo rośnie zatem o jeden; podział jest jedynym środkiem, dzięki któremu drzewo rośnie.

```
{ 1} B_Tree_Insert(T, k)
{ 2} r := root[T]
{ 3} if n[r]=2*t-1 then
{ 4}   s := Allocate_Node()
{ 5}   root[T] := s
{ 6}   leaf[s] := false
{ 7}   n[s] := 0
{ 8}   c1[s] := r
{ 9}   B_Tree_Split_Child(s,1,r)
{10}  B_Tree_Insert_Nonfull(s,k)
{11} else
{12}   B_Tree_Insert_Nonfull(r,k)
```

B-drzewo – wstawianie do niepełnego węzła

```
B_Tree_Insert_Nonfull(x, k)
{ 1} i := n[x]
{ 2} if leaf[x] then
{ 3}   while i >= 1 and k < keyi[x] do
{ 4}     keyi+1[x] := keyi[x]
{ 5}     i := i-1
{ 6}   keyi+1[x] := k
{ 7}   n[x] := n[x]+1
{ 8}   DISK_WRITE(x)
{ 9} else
{10}   while i >= 1 and k < keyi[x] do
{11}     i := i-1
{12}   i := i+1
{13}   DISK_READ(ci[x])
{14}   if n[ci[x]] = 2*t-1 then
{15}     B_Tree_Split_Child(x, i, ci[x])
{16}     if k > keyi[x] then
{17}       i := i+1
{18}   B_Tree_Insert_nonfull(ci[x], k)
```

Dostęp do stron dysku : O(h)

Czas CPU: $\Theta(\text{th})$

B-drzewo - usuwanie

- Usuwanie z B-drzewa jest wykonywane analogicznie do wstawiania, ale dużo bardziej skomplikowane, gdyż klucz może być usuwany z wewnętrznego węzła (nie tylko liścia), co powoduje potrzebę poprawienia jego potomków.
- Gdy w przypadku dodawania należy uważać, żeby węzeł nie był „za duży”, w przypadku usuwania możemy naruszyć zasady B-drzewa, tworząc węzeł ze zbyt małą liczbą kluczy. Stąd pojawia się potrzeba łączenia węzłów.
- Szczegółowe zasady i możliwe przypadki w książce Cormena i in. zajmują prawie 2 strony tekstu. Do ewentualnego doczytania i przeanalizowania samodzielnego.

B-drzewo - Podsumowanie

- Wyszukiwanie :
 - Dostęp do stron dysku: $O(h) = O(\lg_t n)$
 - Czas CPU: $O(th) = O(t \lg_t n)$
- Wstawianie:
 - Dostęp do stron dysku: $O(h) = O(\lg_t n)$
 - Czas CPU: $O(th) = O(t \lg_t n)$
- Usuwanie:
 - Dostęp do stron dysku: $O(h) = O(\lg_t n)$
 - Czas CPU: $O(th) = O(t \lg_t n)$
- Popularniejsze w zastosowaniach bazodanowych i systemach plików są B+drzewa, które są szczególnym przypadkiem B-drzew, przechowującym dane tylko w liściach.
 - Btrfs (ang. B-tree File System) – system plików dla systemu Linux.

Drzewo AVL

Materiał dodatkowy – w języku angielskim

Insertion into AVL tree

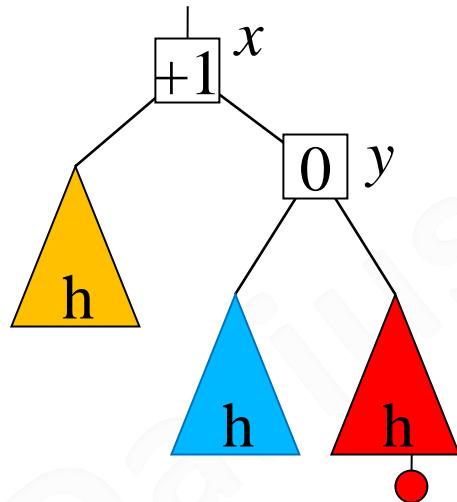
Insertion into an AVL tree may be carried out by inserting the given value into the tree as if it were an unbalanced binary search tree, and then retracing one's steps toward the root updating the balance factor of the nodes.

- If the balance factor becomes -1, 0, or 1 then the tree is still in AVL form, and no rotations are necessary. If the balance factor becomes 0, we end the insertion. Otherwise we have to go one step toward the root.
- If the balance factor becomes 2 or -2 then the tree rooted at this node is unbalanced, and a tree rotation (1 or 2) is needed.

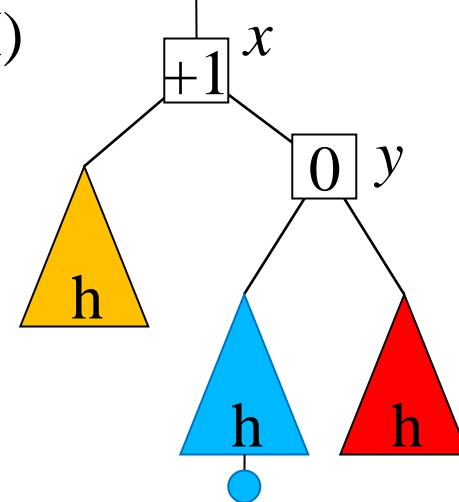
Let's consider a situation the balance factor +2 is found.

before updating up to +2

I)

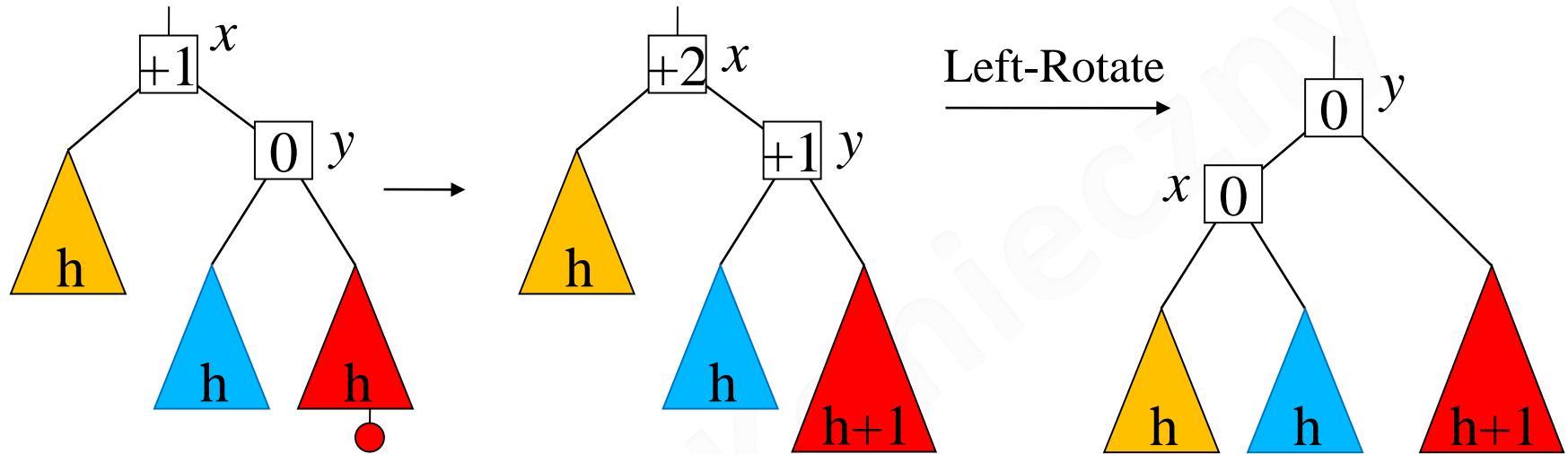


II)

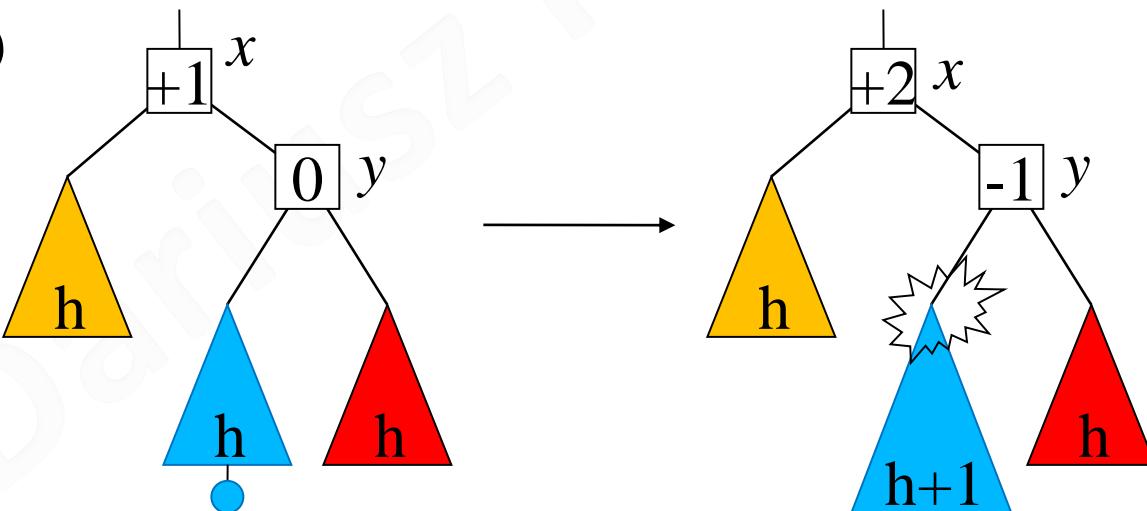


Insertion into AVL tree

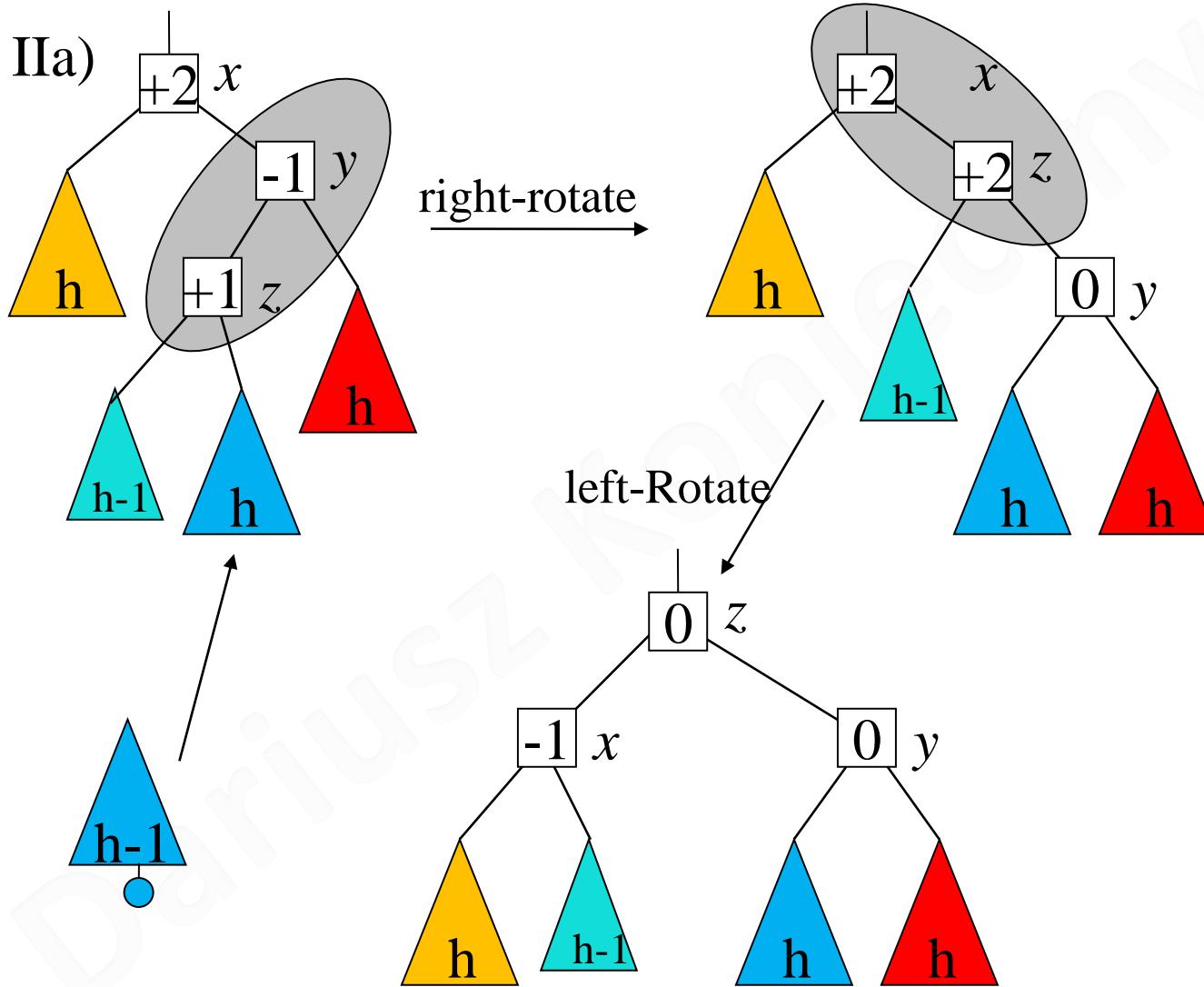
I)



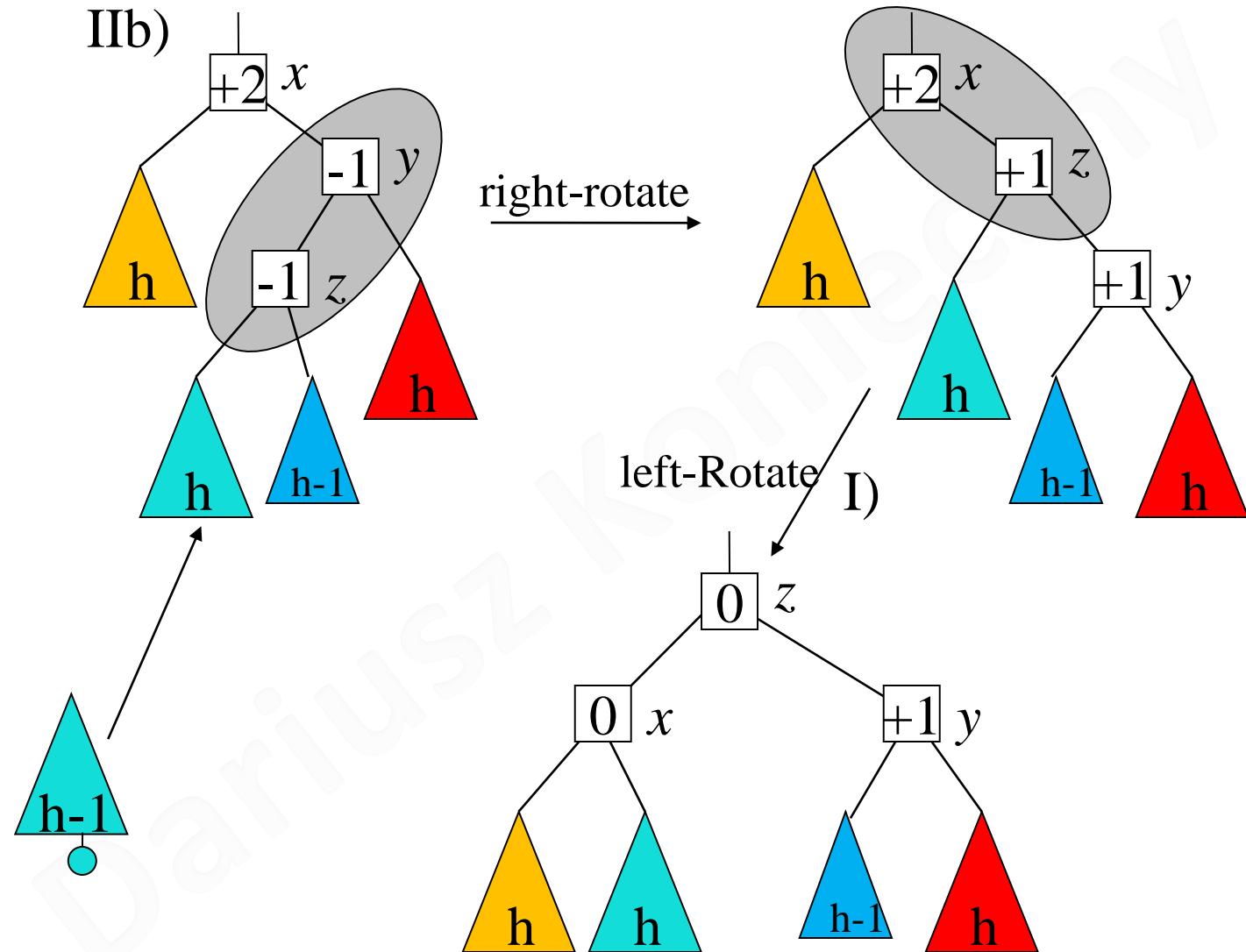
II)



Insertion into AVL tree



Insertion into AVL tree

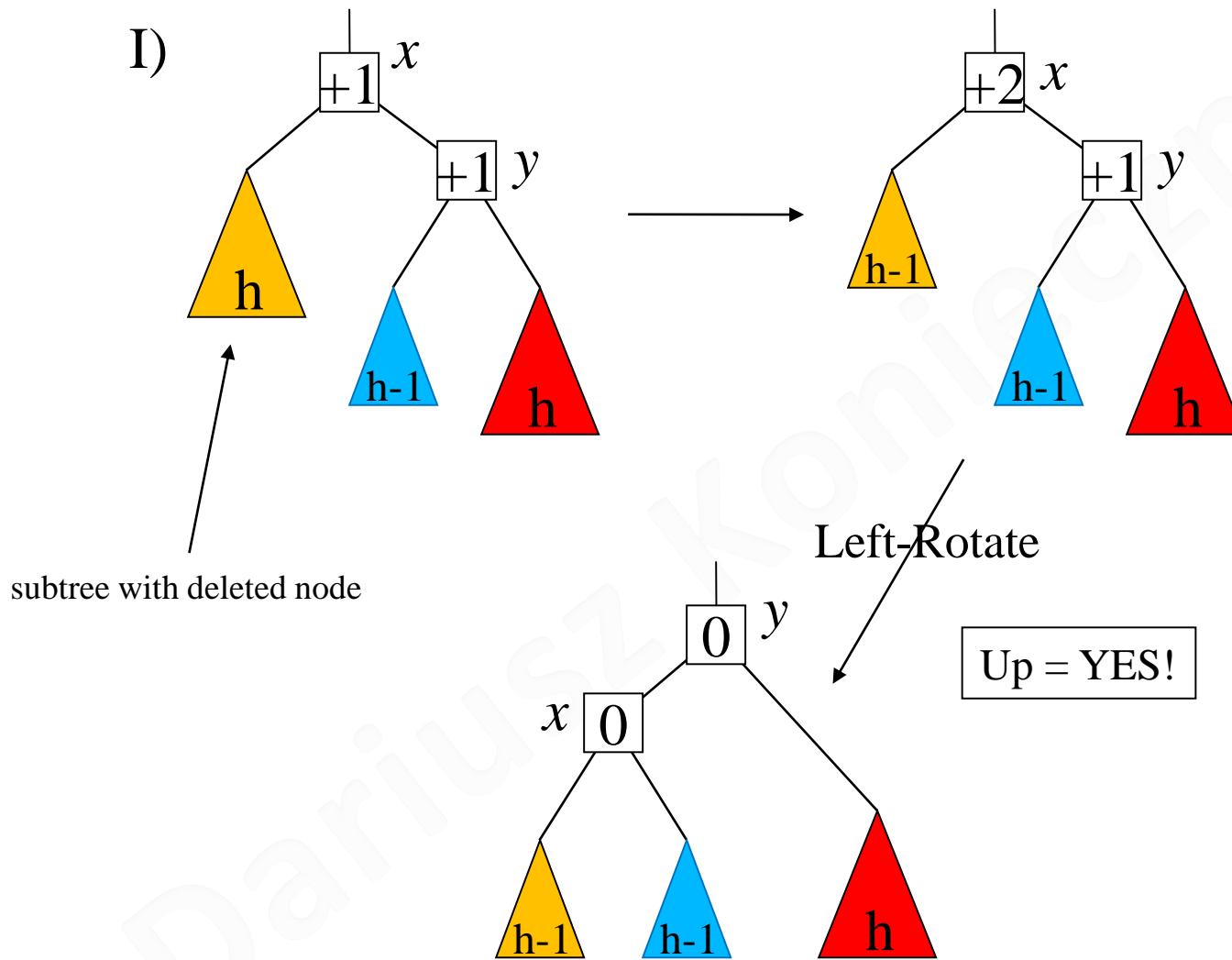


Deletion from AVL tree

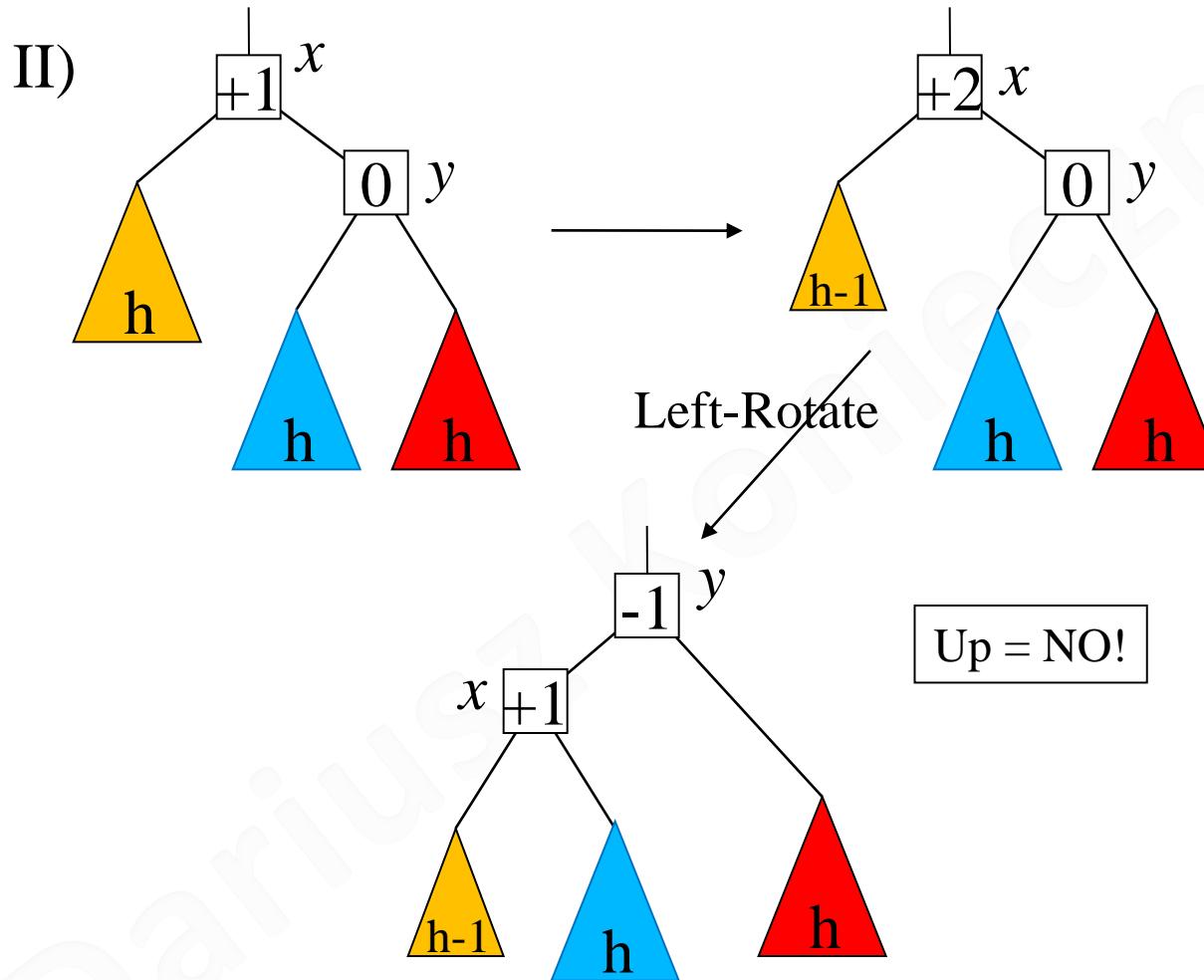
Deletion from an AVL tree may be carried out by deleting the given value from the tree as if it were an unbalanced binary search tree, and then retracing one's steps toward the root updating the balance factor of the nodes.

- If the balance factor becomes -1, 0, or 1 then the tree is still in AVL form, and no rotations are necessary. If the balance factor becomes -1 or 1, we end the deletion. Otherwise we have to go one step toward the root.
- If the balance factor becomes 2 or -2 then the tree rooted at this node is unbalanced, and a tree rotation (1 or 2) is needed. If the rotation leaves the subtree's balance factor at 0 then the retracing towards the root must continue since the height of this subtree has decreased by one.

Deletion from AVL tree

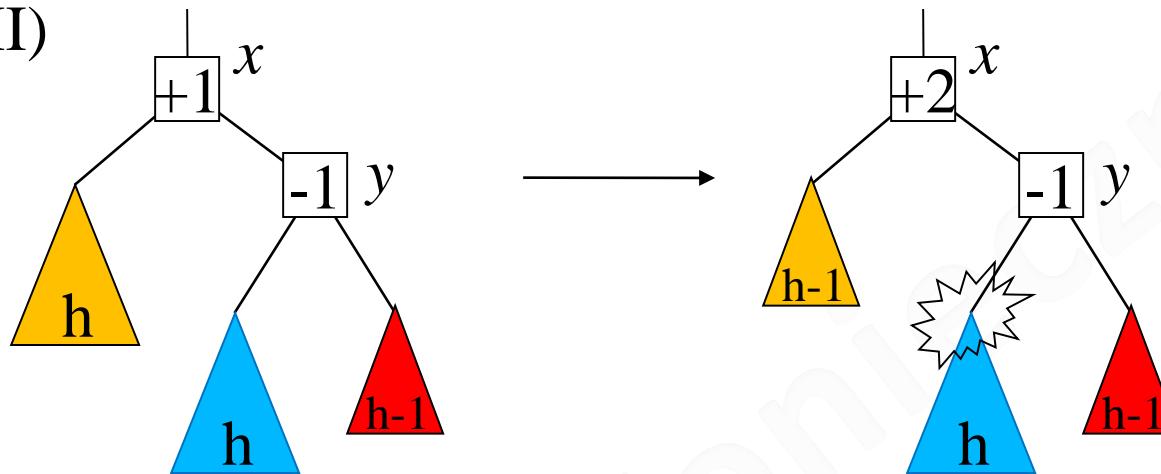


Deletion from AVL tree

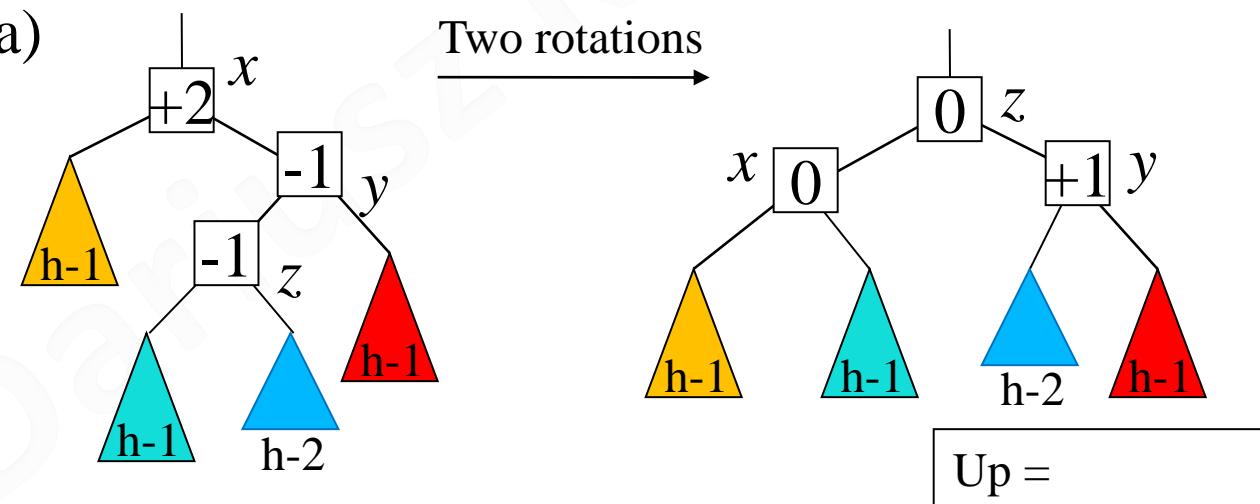


Deletion from AVL tree

III)

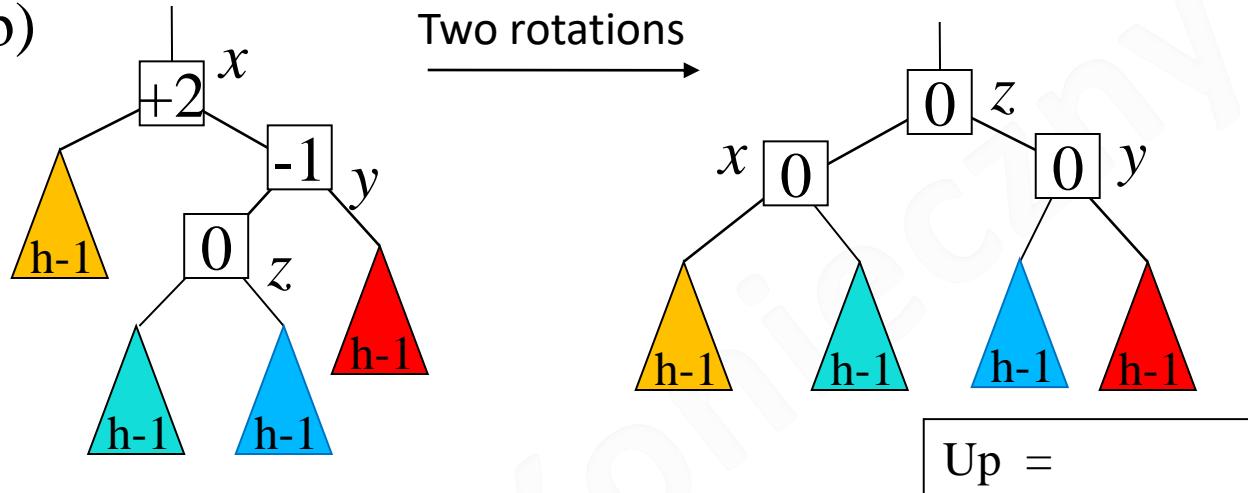


IIIa)

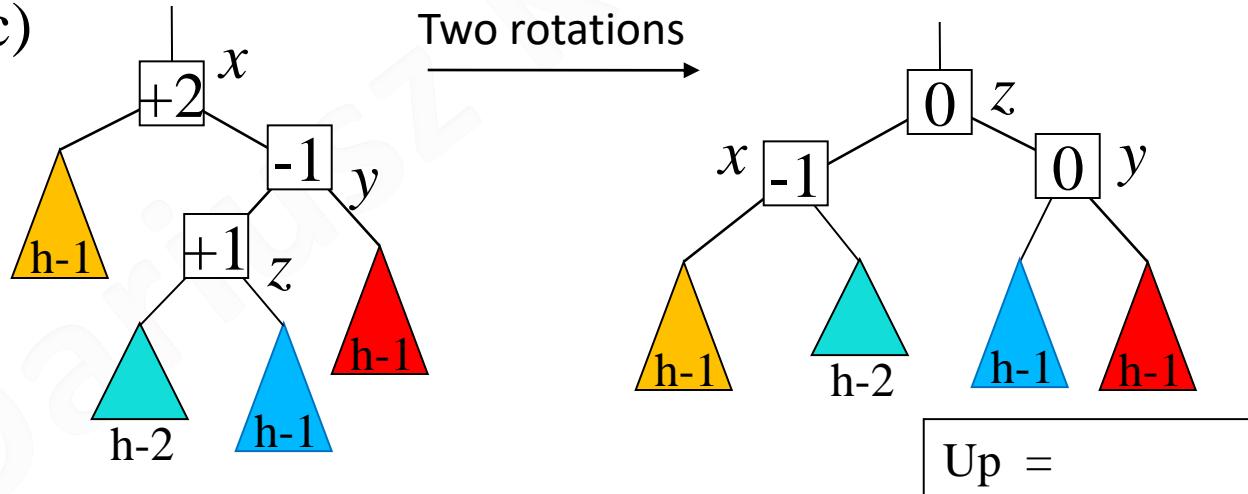


Deletion from AVL tree

IIIb)



IIIc)



Insertion and deletion - complexity

- Insertion:
 - insertion like into BST: $O(h)$
 - steps up to root and max two rotation: $O(h) + 2$
 - complexity: $O(h)=O(\log n)$
- Deletion:
 - deletion like from BST: $O(h)$
 - steps up to root and rotations: $O(2*h)=O(h)$
 - complexity: $O(h)=O(\log n)$

Algorytmy i struktury danych – W09

Drzewa przedziałowe

Kopce dwumianowe

Las zbiorów rozłącznych

Zawartość

- Drzewa przedziałowe
 - Przykład dostosowywania drzew czerwono-czarnych
- Kopce złączalne
 - Kopce dwumianowe: wszystkie operacje
 - Kopce Fibonacciego – krótka informacja
- Struktury danych dla zbiorów rozłącznych:
 - Zmodyfikowana lista wiązana
 - Las zbiorów rozłącznych

Zbiór przedziałów

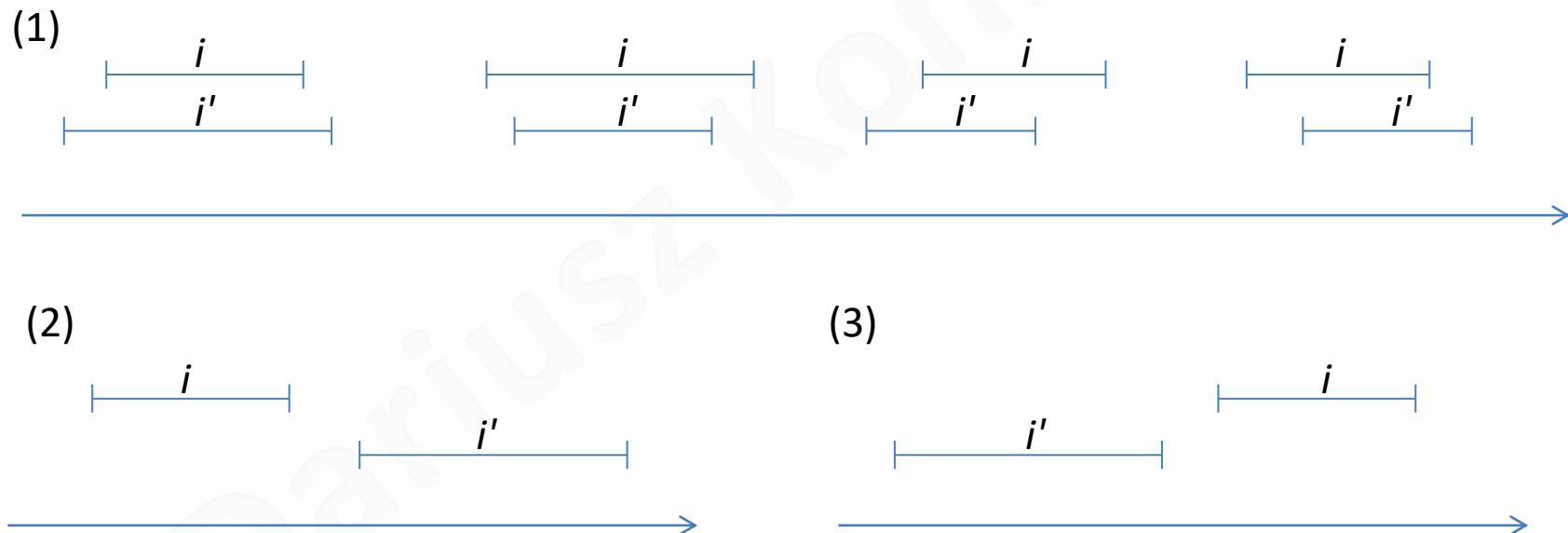
- Chcemy zapamiętać dynamiczny zbiór przedziałów.
- Zbiór ten mam umówić:
 - Dodanie nowego przedziału
 - Usunięcie przedziału ze zbioru
 - Znalezienie dowolnego przedziału, który zachodzi na inny, podany przedział

Definicja przedział

- **Przedział domknięty** jest **uporządkowaną parą** liczb rzeczywistych $[t_1, t_2]$, gdzie $t_1 \leq t_2$.
- Przedział $[t_1, t_2]$ reprezentuje zbiór $\{t \in R : t_1 \leq t \leq t_2\}$
- Dla uproszczenia rozważań będą rozpatrywane tylko przedziały domknięte.
- Przedział $[t_1, t_2]$ można reprezentować jako obiekt i o polach $low[i] = t_1$ (**lewy koniec**) oraz $high[i] = t_2$ (**prawy koniec**)

Zachodzenie na siebie

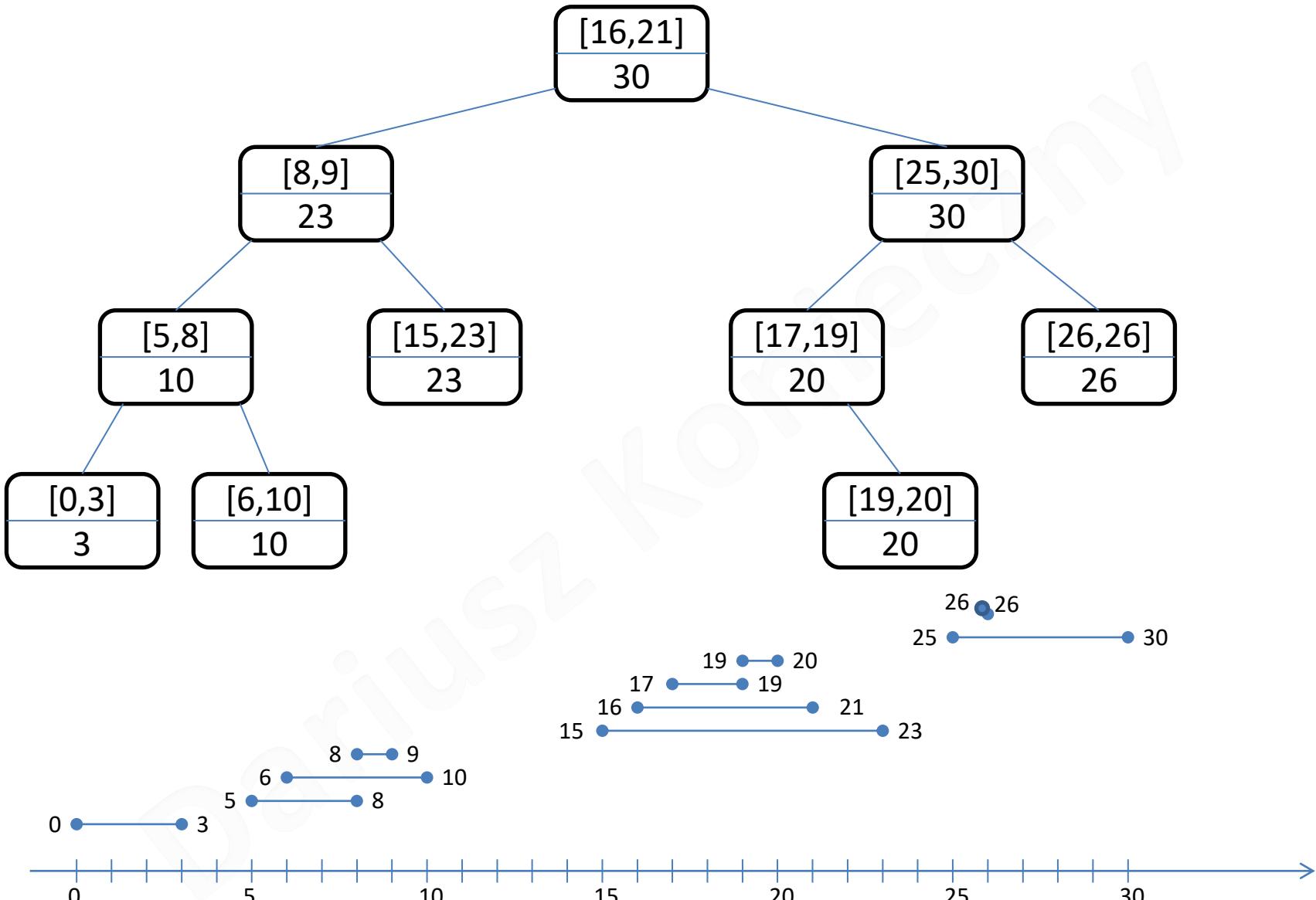
- Dwa przedziały i oraz i' zachodzą na siebie, jeśli $i \cap i' \neq \emptyset$, tj. jeśli $low[i] \leq high[i']$ oraz $low[i'] \leq high[i]$.
- Trychotomia przedziałowa: Każde dwa przedziały i oraz i' spełniają jeden z trzech następujących warunków:
 - (1) i oraz i' zachodzą na siebie;
 - (2) $high[i] < low[i']$
 - (3) $high[i'] < low[i]$



Drzewo przedziałowe

- **Drzewo przedziałowe** jest drzewem czerwono-czarnym, w którym każdy węzeł x zawiera przedział $int[x]$.
- **Kluczem** węzła x jest **lewy koniec** przedziału tj. $low[int[x]]$.
 - Jeśli dopusczamy różne przedziały, ale o tym samym lewy końcu wartość $high[int[x]]$ jest kluczem drugiego stopnia.
 - Jeśli dopusczamy powtarzanie się przedziałów, najlepiej dodać licznik.
- Dodatkowo każdy węzeł posiada atrybut $max[x]$, który jest **największą wartością końca przedziałów** spośród przedziałów znajdujących się **w poddrzewie o korzeniu w x** .
- Podczas wstawiania i usuwania węzła z takiego drzewa należy uaktualniać wartość pola $max[x]$ posuwając się w kierunku korzenia zgodnie z ogólną regułą:
$$max[x] = \max(high[int[x]], max[left[x]], max[right[x]])$$
- Również po wykonaniu rotacji należy skorzystać z powyższej reguły.

Drzewo przedziałowe - przykład



Operacja Interval-Search

- Operacja poszukiwania konkretnego przedziału przebiega jak w drzewie BST
- Istnieje inna operacja, $\text{Interval-Search}(T, i)$, która wyznacza wskaźnik do takiego elementu x drzewa przedziałowego T , że przedziały $\text{int}[x]$ oraz i **zachodzą na siebie**, lub stałą **null**, jeśli nie ma takiego przedziału w zbiorze reprezentowanym przez drzewo T .

```
Interval-Search(T, i)
{ 1}   x := root[T]
{ 2}   while x!= null and (i nie zachodzi na int[x]) do
{ 3}     if left[x]!=null and max[left[x]]>=low[i]
{ 4}       then x=left[x]
{ 5}       else x=right[x]
{ 6}   return x
```

- Przykład poszukiwania przedziałów:
 - $i=[22,25]$
 - $i=[11,14]$
- Dowód poprawności algorytmu można znaleźć w Cormen i in. „Wprowadzenie do algorytmów”

Drzewa przedziałowe inne

- Istnieje jeszcze wiele innych drzew przedziałowych:
 - Umożliwiających szybkie znalezienie **wszystkich przedziałów** zachodzących na dany przedział
 - Dla liczb naturalnych - oparte na tablicy.
 - Inne
- Przedstawione drzewo przedziałowe pozwala rozwiązać poniższy problem:
 - Mamy zbiór n prostokątów na płaszczyźnie o bokach równoległych do osi układu współrzędnych. W czasie $O(n \log n)$ wykryć, czy dowolne dwa prostokąty nachodzą na siebie.

Kopce złączalne

- Kopce złączalne (ang. mergeable heaps) umożliwiają wykonanie następujących operacji:
 - **Make-Heap()** tworzy i zwraca nowy, pusty kopiec.
 - **Insert(H, x)** wstawia do kopca H węzeł x o danym kluczu $\text{key}[x]$.
 - **Minimum(H)** zwraca wskaźnik do węzła w kopcu H , który zawiera minimalny klucz.
 - **Extract-Min(H)** usuwa z kopca H węzeł o minimalnym kluczu, zwracając jednocześnie wskaźnik do tego węzła.
 - **Union(H_1, H_2)** tworzy i zwraca nowy kopiec, który zawiera wszystkie węzły z kopców H_1 i H_2 . Kopce H_1 i H_2 są w tej operacji „niszczone”.
- Dodatkowo przedstawiona struktura danych umożliwia wykonanie następujących operacji:
 - **Decrease-Key(H, x, k)** nadaje kluczowi w węźle x z kopca H nową wartość k , o której zakłada się, że jest nie większa od bieżącej wartości klucza.
 - **Delete(H, x)** usuwa węzeł x z kopca H .

Porównanie kopków złączalnych

Procedura	Kopiec binarny (koszt pesymistyczny)	Kopiec dwumianowy (koszt pesymistyczny)	Kopiec Fibonacciego (koszt zamortyzowany)
Make-Heap	$\Theta(1)$	$\Theta(1)$	$\Theta(1)$
Insert	$\Theta(\lg n)$	$O(\lg n)$	$\Theta(1)$
Minimum	$\Theta(1)$	$O(\lg n)$	$\Theta(1)$
Extract-Min	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$
Union	$\Theta(n)$	$O(\lg n)$	$\Theta(1)$
Decrease-Key	$\Theta(\lg n)$	$\Theta(\lg n)$	$\Theta(1)$
Delete	$\Theta(\lg n)$	$\Theta(\lg n)$	$O(\lg n)$

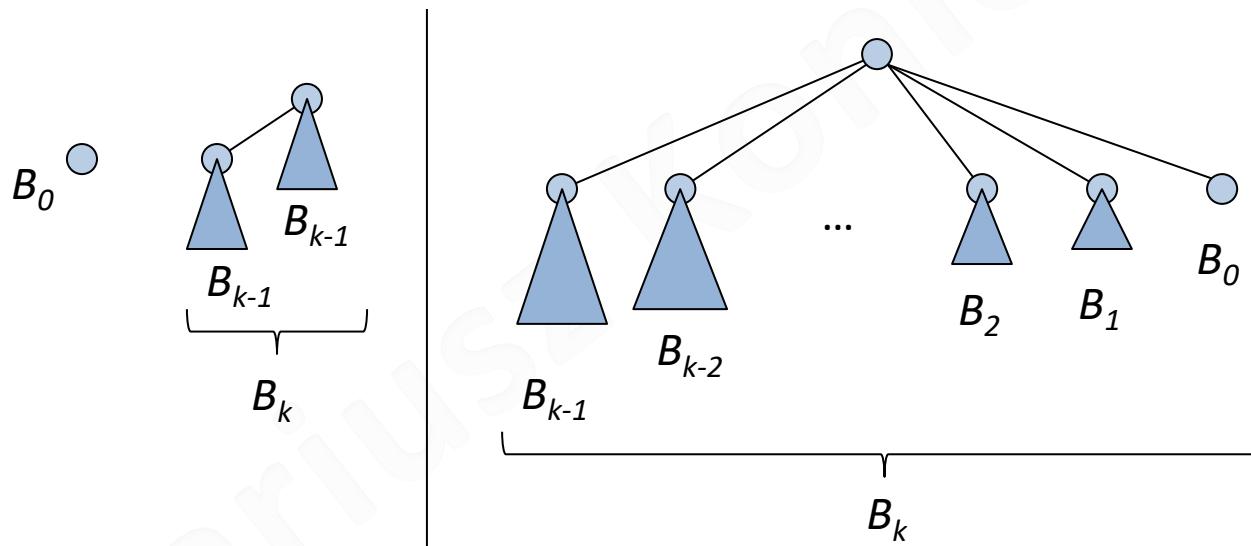
Uwagi do porównania

- Podstawowy problem kopca dwumianowego to liniowy czas łączenia kopków
- Dla kopca Fibonacciego podano czas zamortyzowany, więc może się zdarzyć, że wykonanie pewnej operacji w danym momencie będzie miało czas $O(n)$.
- Wszystkie kopce są nieefektywne jeśli chodzi o operację Search poszukującą dowolnego klucza. Z tego powodu operacje Decrease-Key i Delete wymagają podania wskaźnika do węzła. W wielu zastosowaniach nie stanowi to problemu.

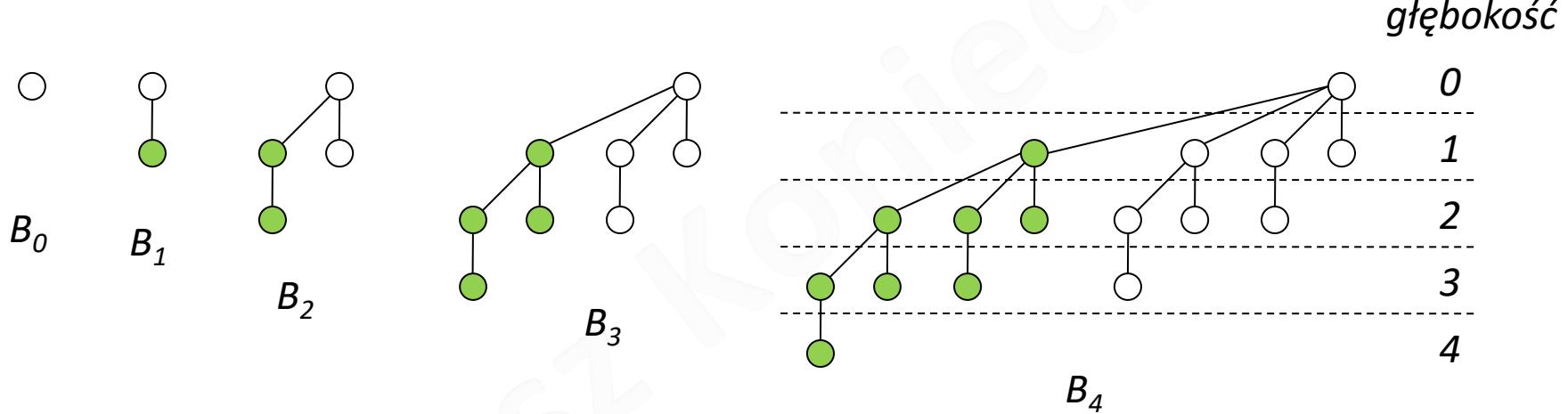
Drzewo dwumianowe

Drzewo dwumianowe (ang. binomial tree) B_k jest drzewem uporządkowanym zdefiniowanym rekurencyjnie:

- Drzewo dwumianowe B_0 składa się z pojedynczego węzła.
- Drzewo dwumianowe B_k składa się z dwóch drzew B_{k-1} które są razem powiązane: korzeń jednego drzewa jest skrajnie lewym synem drugiego.



Drzewa dwumianowe - przykłady



Drzewa dwumianowe - właściwości

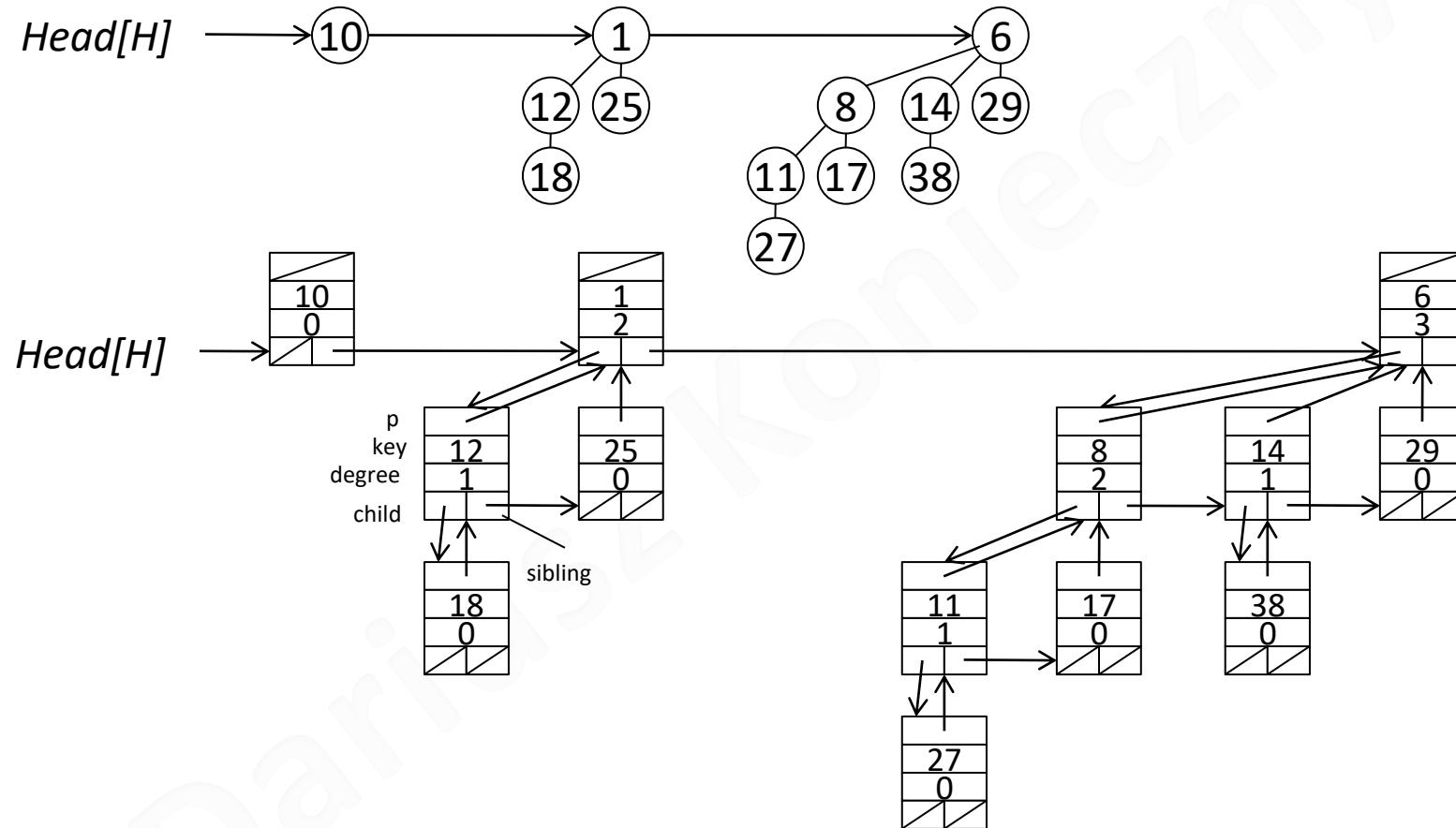
- W drzewie dwumianowym B_k :
 - 1) jest 2^k węzłów;
 - 2) wysokość drzewa wynosi k ;
 - 3) dokładnie $\binom{k}{i}$ węzłów znajduje się na głębokości i , dla $i = 0, 1, \dots, k$;
 - 4) stopień korzenia wynosi k i jest większy od stopnia każdego innego węzła; ponadto, jeżeli synów korzenia ponumerujemy kolejno, od lewego do prawego $k-1, k-2, \dots, 0$, to syn o numerze i jest korzeniem poddrzewa B_i .
- Wniosek:
maksymalny stopień węzła w n -węzłowym drzewie dwumianowym wynosi $\lg n$.

Kopiec dwumianowy - KD

- **Kopiec dwumianowy H jest zbiorem drzew dwumianowych, które mają następujące właściwości kopca dwumianowego:**
 - (1) Każde drzewo dwumianowe w kopcu H jest **uporządkowane kopcowo**: klucz w węźle jest nie mniejszy niż klucz ojca.
 - (2) Dla każdego $d \geq 0$ istnieje w H co najwyżej jedno drzewo dwumianowe, którego korzeń ma stopień d .
- Właściwość (1) mówi, że klucz w korzeniu drzewa z porządkiem kopcowym jest najmniejszym kluczem w drzewie.
- Z właściwości (2) wynika, że n -węzłowy kopiec dwumianowy H składa się z co najwyżej $\lfloor \lg n \rfloor + 1$ drzew dwumianowych. Ponieważ binarna reprezentacja n zawiera $\lfloor \lg n \rfloor + 1$ bitów, niech będzie to $\langle b_{\lfloor \lg n \rfloor}, b_{\lfloor \lg n \rfloor - 1}, \dots, b_0 \rangle$. Drzewo dwumianowe B_i występuje w kopcu H wtedy i tylko wtedy, gdy bit $b_i = 1$.

Kopiec dwumianowy - reprezentacja

- Przykład



Operacje na KD – tworzenie, znajd. min. klucza

- Tworzenie nowego kopca dwumianowego:
 - `Make-Binomial-Heap()` – rezerwacja pamięci dla obiektu H , ustawienie `head[H]=null` oraz zwrócenie wskaźnika na H
- Znajdowanie minimalnego klucza:
 - Procedura `Binomial-Heap-Minimum` zwraca wskaźnik do węzła z minimalnym kluczem w n -węzłowym kopcu dwumianowym H . W tej realizacji zakłada się, że w kopcu nie ma kluczy o wartości ∞ .

```
Binomial-Heap-Minimum(H)
{ 1} y:=null
{ 2} x:=head[H]
{ 3} min:= $\infty$ 
{ 4} while x ≠ null do
{ 5}   if key[x] < min then
{ 6}     min:=key[x]
{ 7}     y:=x
{ 8}     x:=sibling[x]
{ 9} return y
```

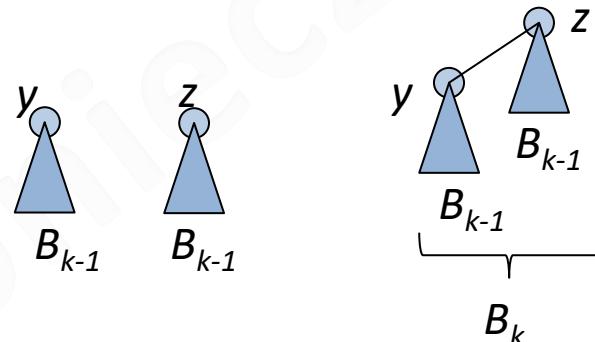
Złożoność czasowa $\Theta(\lg n)$

Łączenie dwóch drzew dwumianowych

- Następująca procedura dowiązuje **drzewo** B_{k-1} o korzeniu w węźle y do **drzewa** B_{k-1} o korzeniu w węźle z ; z staje się ojcem y . Węzeł z zostaje w ten sposób korzeniem drzewa B_k .

```
Binomial-Link(y, z)
{ 1} p[y]:=z
{ 2} sibling[y]:=child[z]
{ 3} child[z]:=y
{ 4} degree[z]:=degree[z]+1
```

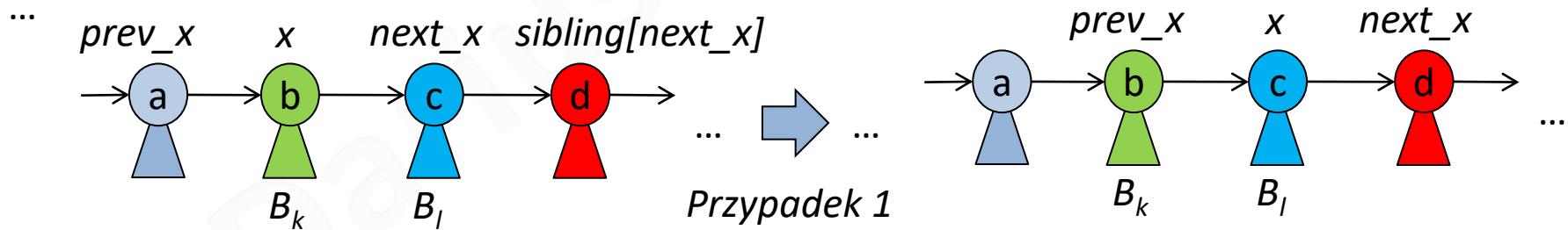
Złożoność $\Theta(1)$



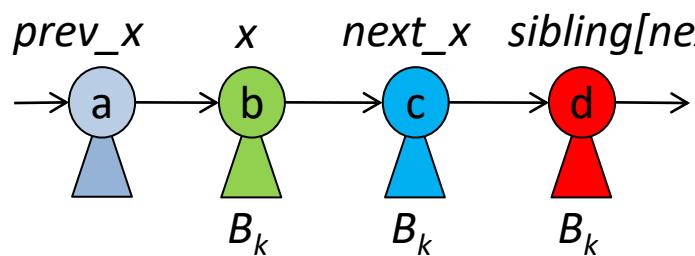
- Operacja łączenia dwóch **kopców** dwumianowych jest używana jako podprogram w większości pozostałych operacji. Działanie Binomial-Heap-Union polega na łączeniu drzew dwumianowych, których korzenie mają te same stopnie (używając Binomial-Link).

Łączenie dwóch KD 1/2

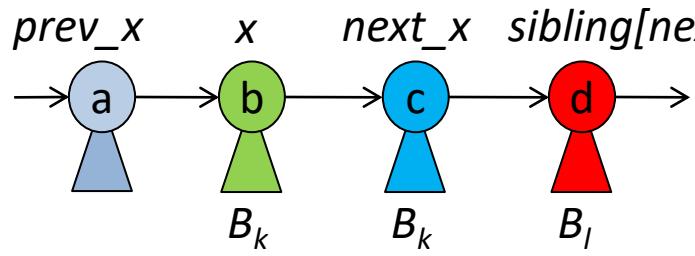
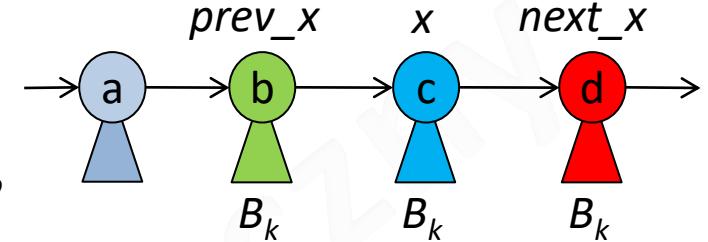
- Procedura Binomial-Heap-Union:
 - Łączy dwa kopce H_1 i H_2 w jeden kopiec
 - Kopce H_1 i H_2 są niszczone
 - Używa pomocniczej procedury Binomial-Heap-Merge, która scalą listy korzeni kopców H_1 i H_2 w pojedynczą listę **uporządkowaną niemalejąco wg stopni**. (jak operacja Merge dla list posortowanych).
 - Po scaleniu podczas przechodzenia po liście mogą wystąpić 4 sytuacje w zależności do stopni w x , $next_x$, $sibling[next_x]$
 - W każdym przypadku zakładamy, że $k \neq l$



Łączenie dwóch KD 2/2

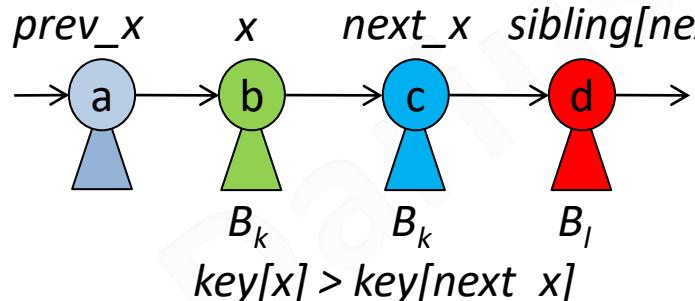
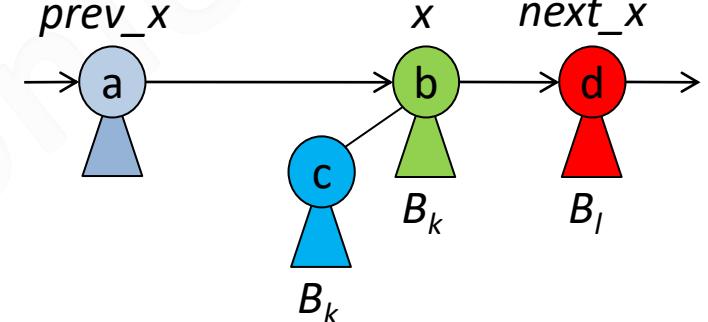


Przypadek 2



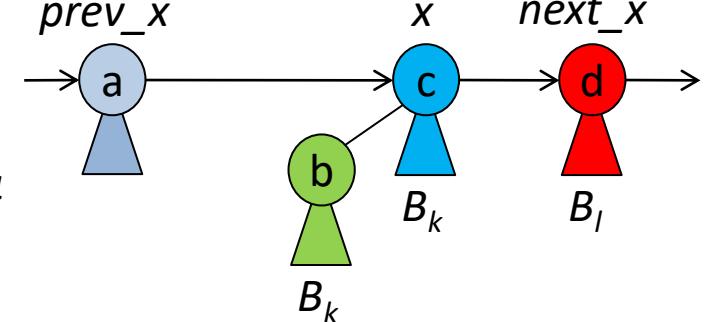
Przypadek 3

$key[x] \leq key[next_x]$

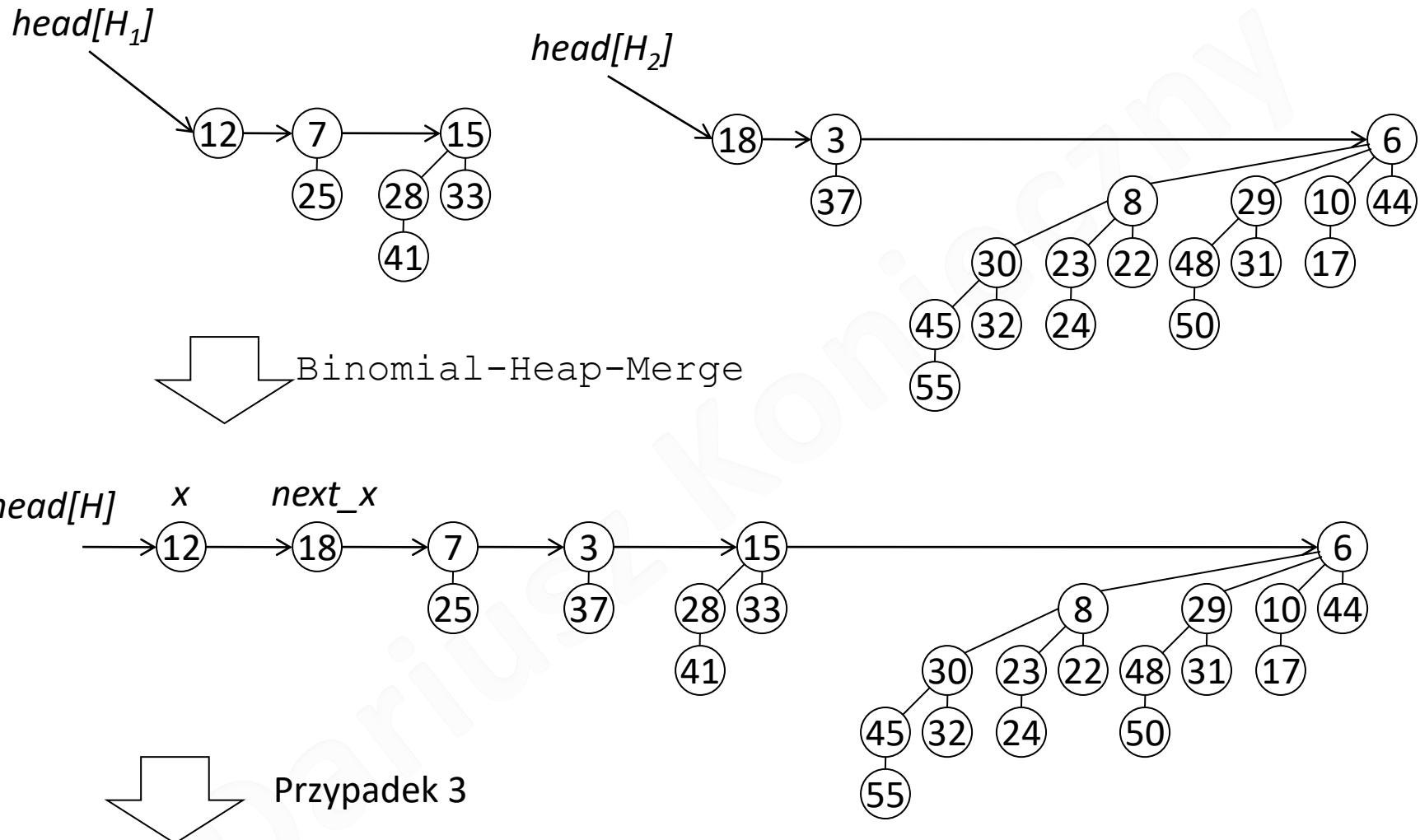


Przypadek 4

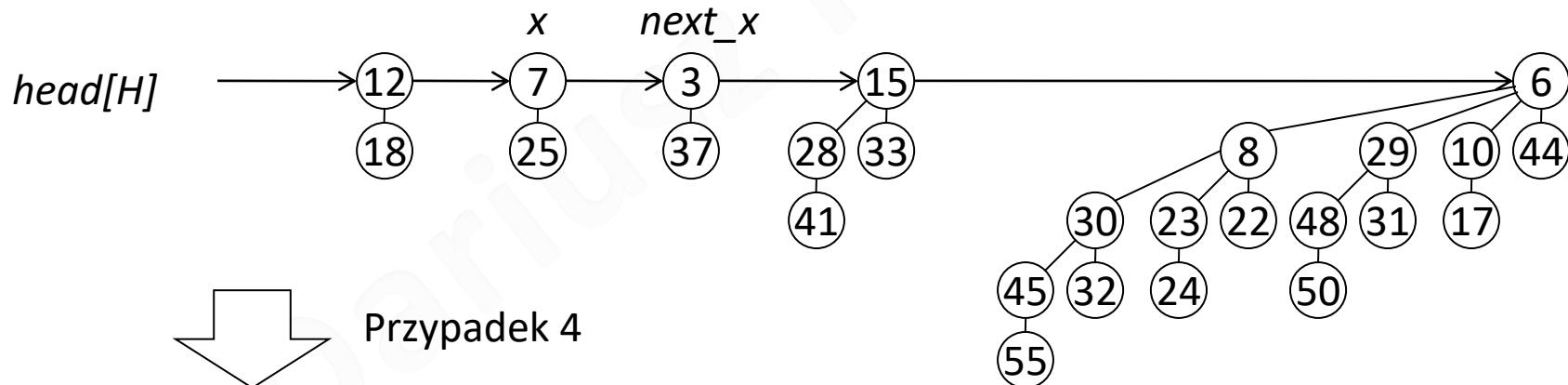
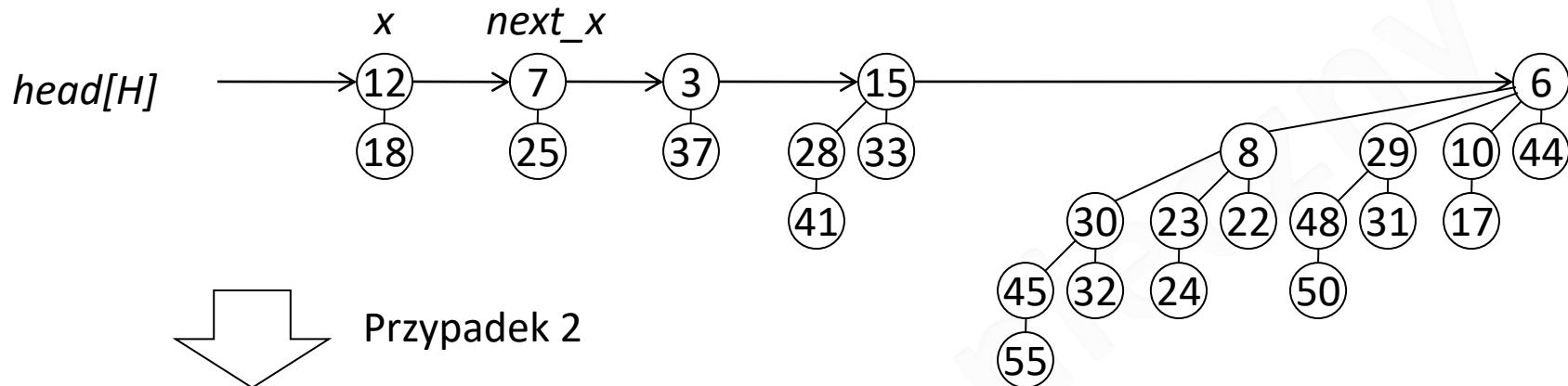
$key[x] > key[next_x]$



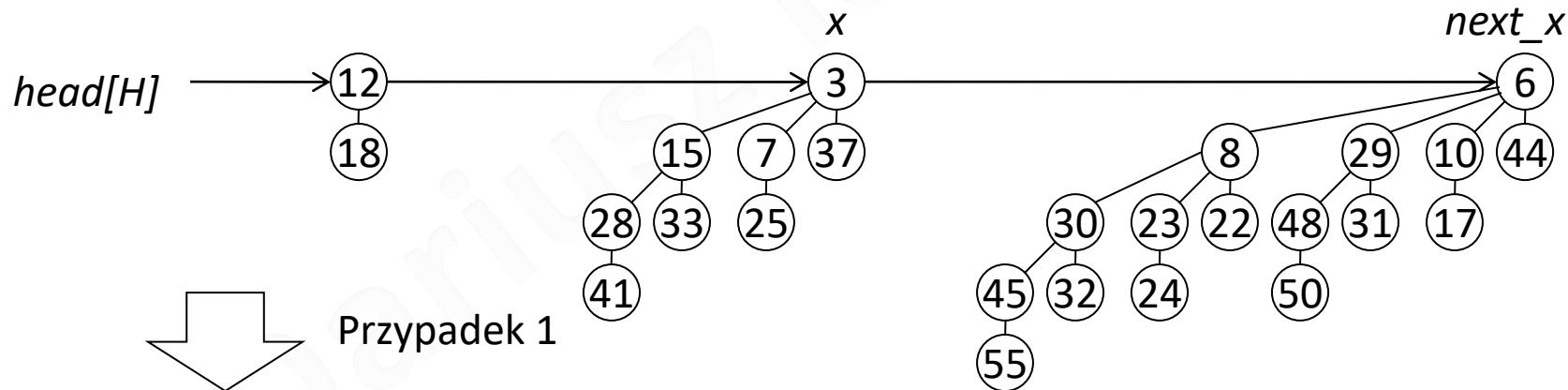
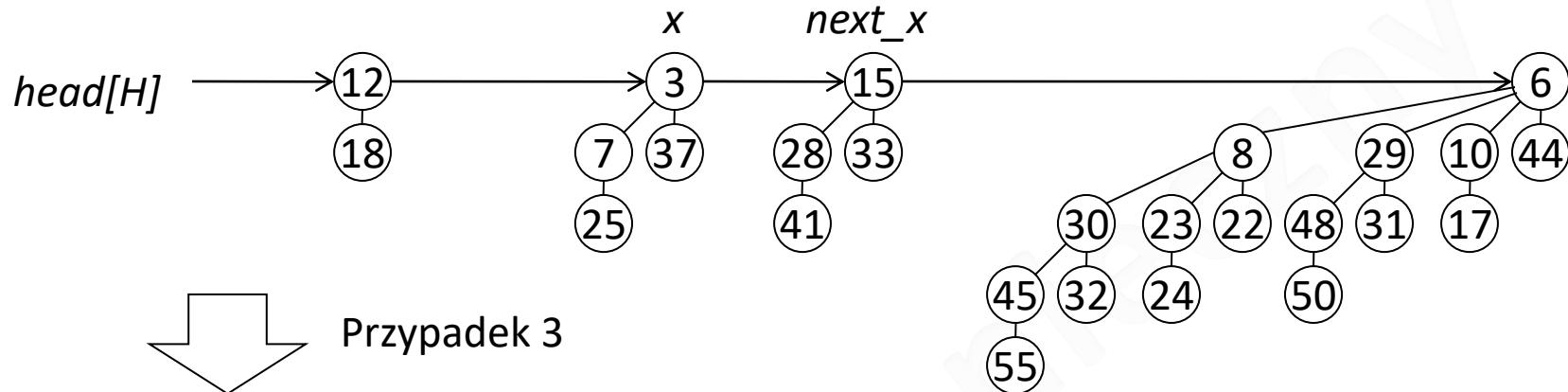
Łączenie dwóch KD – przykład 1/3



Łączenie dwóch KD – przykład 2/3



Łączenie dwóch KD – przykład 3/3



Łączenie dwóch KD - kod

```
Binomial-Heap-Union( $H_1, H_2$ )
{ 1}  $H := \text{Make-Binomial-Heap}()$ 
{ 2}  $\text{head}[H] := \text{Binomial-Heap-Merge}(H_1, H_2)$ 
{ 3} Zwolnij obiekty  $H_1$  and  $H_2$ , ale nie listy, na które wskazują
{ 4} if  $\text{head}[H] = \text{null}$  then
{ 5}     return  $H$ 
{ 6}  $\text{prev\_x} := \text{null}$ 
{ 7}  $x := \text{head}[H]$ 
{ 8}  $\text{next\_x} := \text{sibling}[x]$ 
{ 9} while  $\text{next\_x} \neq \text{NIL}$  do
{10}     if ( $\text{degree}[x] \neq \text{degree}[\text{next\_x}]$ ) or ( $\text{sibling}[\text{next\_x}] \neq \text{null}$  and
{11}          $\text{degree}[\text{sibling}[\text{next\_x}]] = \text{degree}[x]$ ) then
{12}          $\text{prev\_x} = x$                                 // przypadek 1 i 2
{13}          $x = \text{next\_x}$                             // przypadek 1 i 2
{14}     else
{15}         if  $\text{key}[x] \leq \text{key}[\text{next\_x}]$  then
{16}              $\text{sibling}[x] := \text{sibling}[\text{next\_x}]$         // przypadek 3
{17}              $\text{Binomial-Link}(\text{next\_x}, x)$                 // przypadek 3
{18}         else
{19}             if  $\text{prev\_x} = \text{null}$  then                                // przypadek 4
{20}                  $\text{head}[H] := \text{next\_x}$                       // przypadek 4
{21}             else
{22}                  $\text{sibling}[\text{prev\_x}] := \text{next\_x}$           // przypadek 4
{23}                  $\text{Binomial-Link}(x, \text{next\_x})$             // przypadek 4
{24}                  $x := \text{next\_x}$                             // przypadek 4
{25}                  $\text{next\_x} := \text{sibling}[x]$ 
return  $H$ 
```

Złożoność $\Theta(\lg n)$

Wstawianie węzła z kluczem

```
Binomial-Heap-Insert (H, x)
{ 1} H' := Make-Binomial-Heap ()
{ 2} p[x] := NIL
{ 3} child[x] := NIL
{ 4} sibling[x] := NIL
{ 5} degree[x] := 0
{ 6} head[H'] = x
{ 7} H := Binomial-Heap-Union (H, H')
```

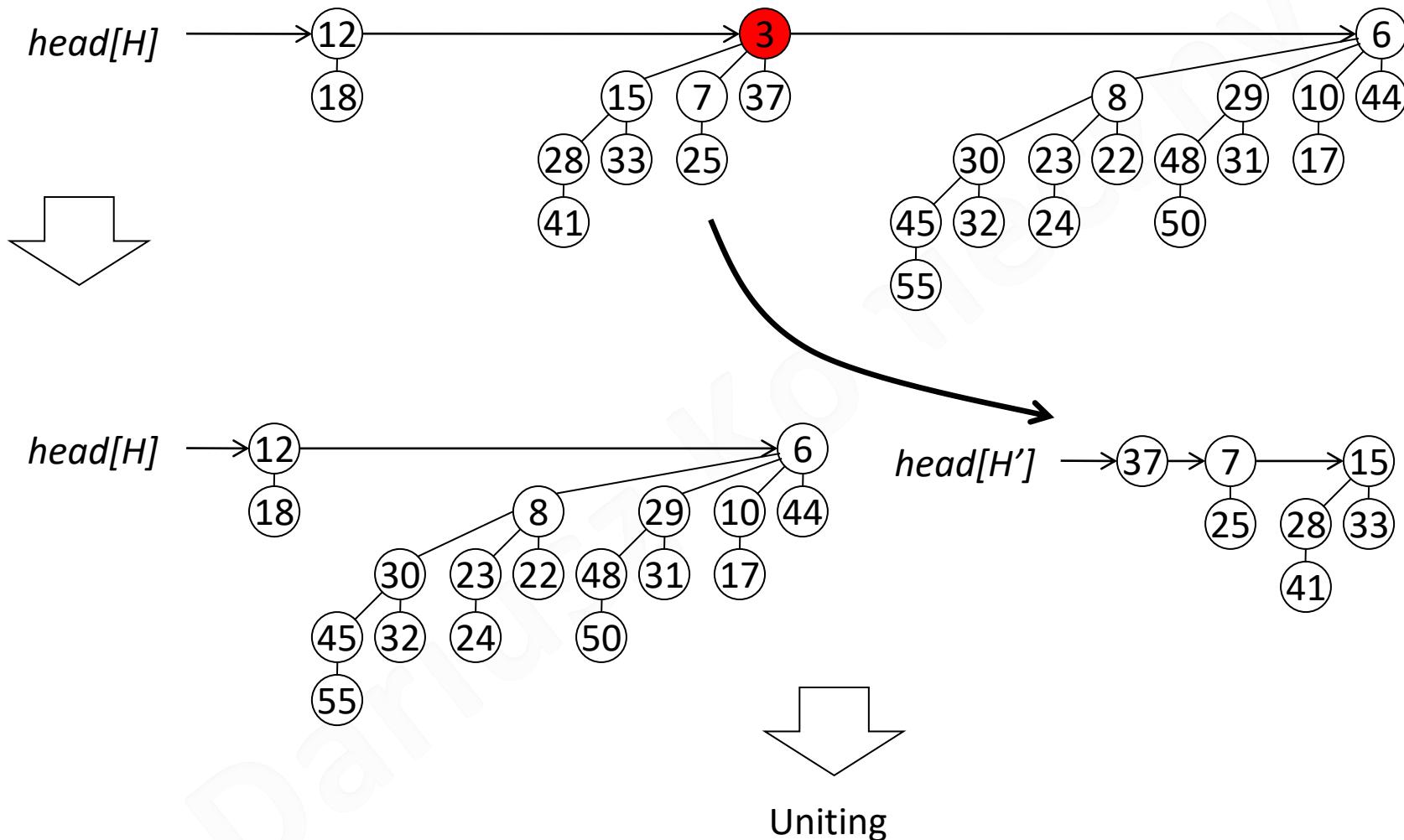
Złożoność $\Theta(\lg n)$

Usuwanie węzła z minimalnym kluczem

```
Binomial-Heap-Extract-Min (H)
{ 1} Znajdź korzeń x z minimalnym kluczem na liście korzeni H
    i usuń x z listy korzeni H
{ 2} H' := Make-Binomial-Heap ()
{ 3} Odwróć kolejność elementów na liście synów węzła x i zapamiętaj
    wskaźnik do głowy wynikowej listy w zmiennej head[H']
{ 4} H := Binomial-Heap-Union (H, H')
{ 5} return x
```

Złożoność $\Theta(\lg n)$

Binomial-Heap-Extract-Min – przykład



Zmniejszenie wartości klucza

```
Binomial-Heap-Decrease-Key(H, x, k)
{ 1} if k > key[x] then
{ 2}   error „new key is greater than current key”
{ 3} key[x] := k
{ 4} y := x
{ 5} z := p[y]
{ 6} while z ≠ NIL and key[y] < key[z] do
{ 7}   exchange key[y] and key[z]
{ 8}   if y and z have satellite fields, exchange them, too
{ 9}   y := z
{10}  z := p[y]
```

Złożoność $\Theta(\lg n)$

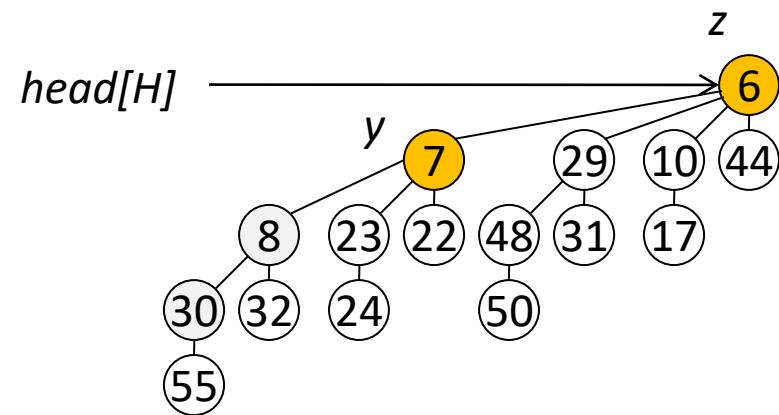
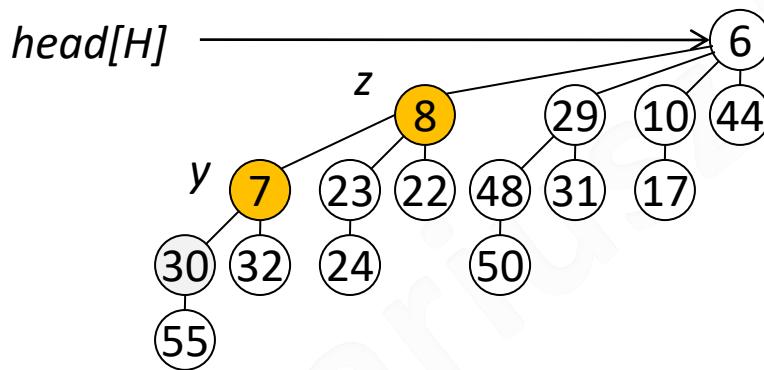
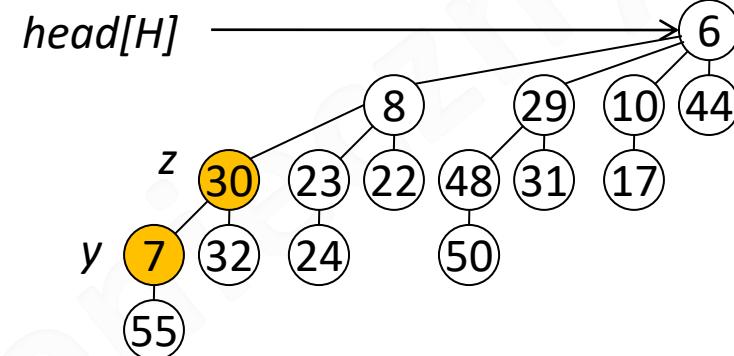
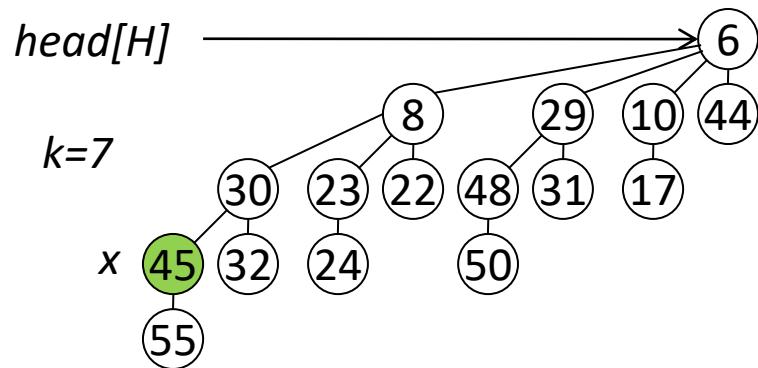
Usunięcie węzła/klucza

```
Binomial-Heap-Delete(H, x)
{ 1} Binomial-Heap-Decrease-Key(H, x, -∞)
{ 2} Binomial-Heap-Extract-Min(H)
```

Złożoność $\Theta(\lg n)$

- Powyższe operacje zakładają, że nie tracimy czas na poszukiwanie węzła

Zmniejszanie wartości klucza - przykład



Kopce - podsumowanie

- Kopiec dwumianowy:
 - Większość operacji używa procedury łączenia dwóch kopków, co powoduje ich złożoność logarytmiczną
- Kopiec Fibonacciego:
 - Idea podobna do kopca dwumianowego
 - Inna reprezentacja: nieuporządkowane, dwukierunkowe listy
 - Naprawianie struktury odkładane w czasie, gdy jest to absolutnie niezbędne
 - Dużo operacji wykonywalne w zamortyzowanym czasie stałym.

Str. danych dla zbiorów rozłącznych

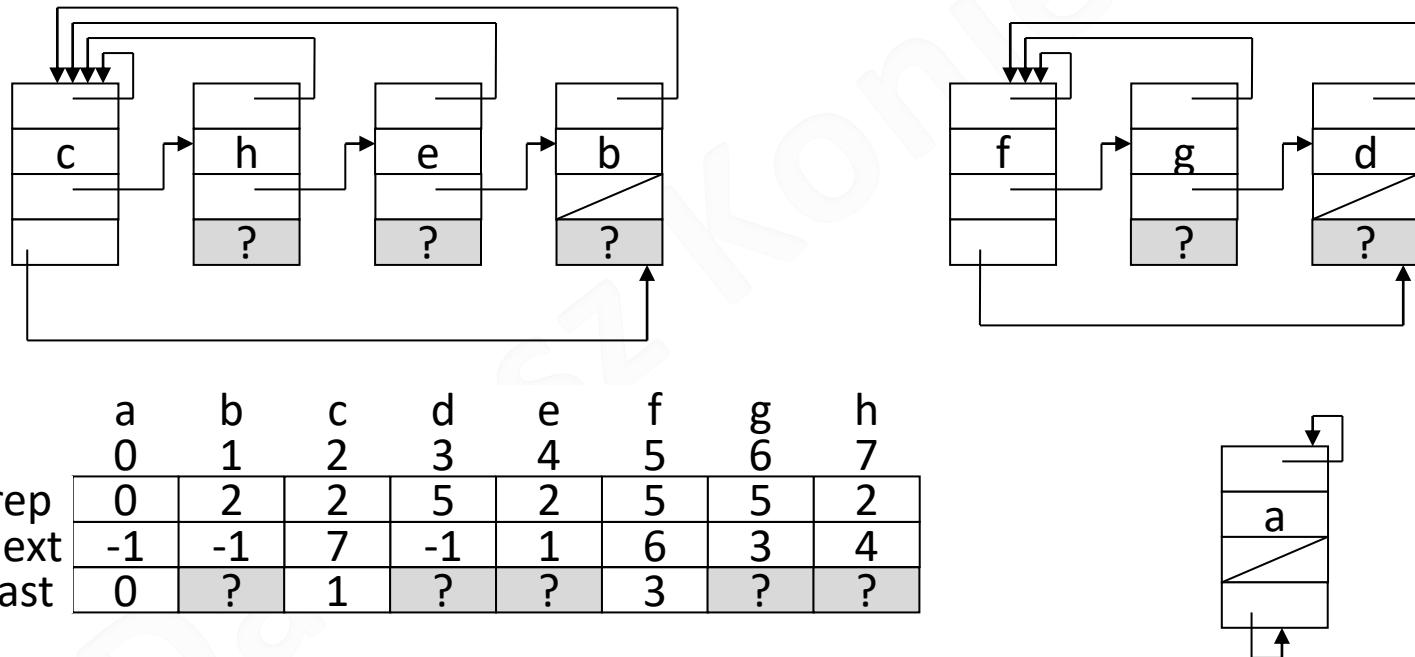
- Struktury danych dla zbiorów rozłącznych:
 - Zmodyfikowana lista wiązana
 - Las zbiorów rozłącznych

Operacje na zbiorach rozłącznych

- Struktura danych dla zbiorów rozłącznych umożliwia zarządzanie rodziną $S=\{S_1, S_2, \dots, S_k\}$ rozłącznych zbiorów dynamicznych.
- Każdy zbiór identyfikowany jest przez swojego **reprezentanta** będącego pewnym elementem tego zbioru.
 - W wielu zastosowaniach nie ma znaczenia, którego elementu użyjemy jako reprezentanta
 - W innych może być określona zasada wyboru, np. z wartością minimalną
- Każdy element zbioru reprezentowany jest przez pewien obiekt.
- Dla danego obiektu x (i ewentualnie y) dynamiczna struktura musi umożliwiać następujące operacje:
 - **Make-Set (x)** – tworzy nowy zbiór którego jedyny element (a więc i reprezentant) jest wskazywany przez x . Ponieważ zbory są rozłączne, x nie może być elementem innego zbioru.
 - **Union (x, y)** – łączy dwa zbory dynamiczne zawierające odpowiednio x i y , powiedzmy S_x i S_y , w nowy zbiór będący ich sumą.
 - S_x i S_y są rozłączne
 - Reprezentantem sumy jest najczęściej albo reprezentant S_x albo reprezentant S_y
 - Zbory S_x i S_y są „niszczone” i usuwane z rodziny zbiorów S .
 - **Find-Set (x)** – zwraca wskaźnik do reprezentanta zbioru zawierającego x . Jest to najczęstsza operacja!

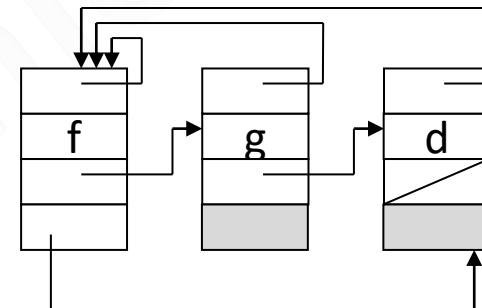
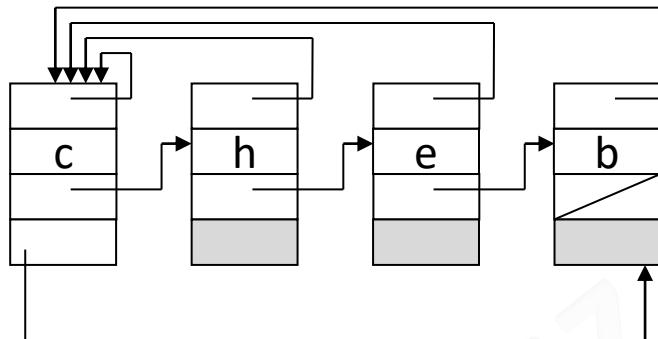
Implementacja za pomocą listy wiązanej

- Lista jednokierunkowa prosta z modyfikacjami
- Każdy element ma wiązanie do reprezentanta, którym jest pierwszy element listy (głowa), aby operacja Find-Set była wykonywana w czasie $O(1)$
- Każdy element ma wiązanie do ostatniego elementu, ale można używać to pole tylko w reprezentancie. Pola to przyspieszenia wykonanie operacji Union.
- W praktyce liczba wszystkich elementów jest znana zanim skorzystamy ze struktury, stąd elementy mogą być pamiętane w tablicy indeksowanej liczbami naturalnymi od zera, a zamiast wskaźników używać indeksów.

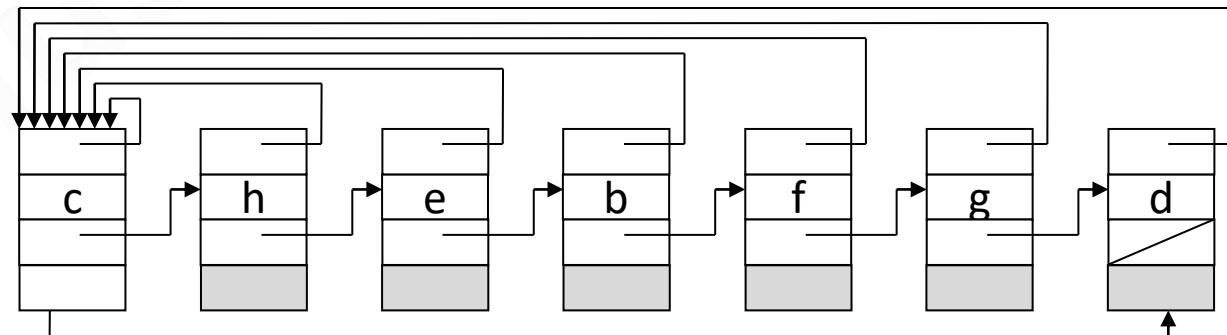


Operacja Union(x,y)

- Znajdujemy reprezentanta x, potem ostatni element listy. Za ostatnim elementem dołączamy reprezentanta y.
- W drugiej liście musimy uaktualnić pola wskazujące na reprezentanta.
- Czas wykonania zależny od długości drugiej listy.



Union(e,g)



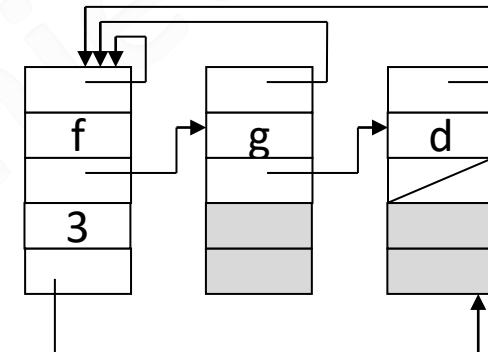
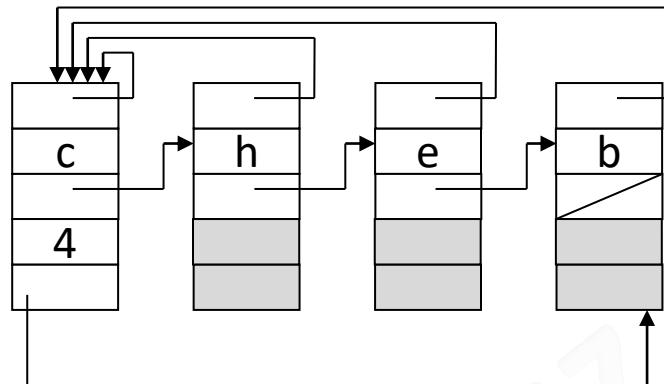
Prosta implementacja - analiza

- Make-Set () : $O(1)$
- Find-Set () : $O(1)$
- Union () : czas zamortyzowany = $\Theta(m)$ dla każdej z $m=\Theta(n)$ operacji

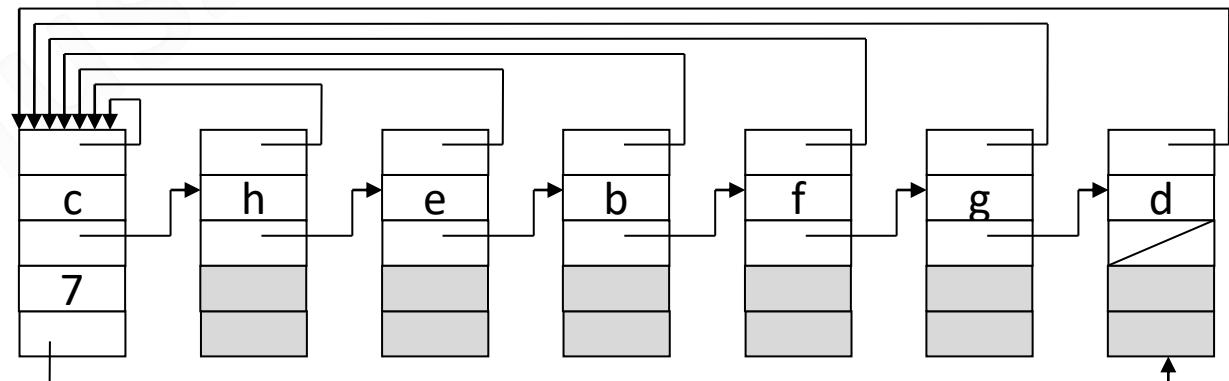
operacja	liczba zaktualizowanych obiektów		
Make_Set (x_1)	1		
Make_Set (x_2)	1		
...	...		
Make_Set (x_n)	1		$\Theta(n + q^2)$
Union (x_2, x_1)	1		
Union (x_3, x_2)	2		
Union (x_4, x_3)	3		
...	...		
Union (x_q, x_{q-1})	$q-1$		
		m operacji	$\Theta(m^2)$
		1 operacja	$\Theta(m)$

Operacja Union z wyważaniem

- Każdy reprezentant zawiera dodatkowo informację o długości listy (która łatwo uaktualniać) i że zawszełączamy krótszą listę do dłuższej (lub równej).
- Czas wykonania pojedynczej operacji Union przy użyciu tej prostej **heurystyki łączenia z wyważaniem** po analizie kosztu zamortyzowanego wynosi $O(\lg n)$.

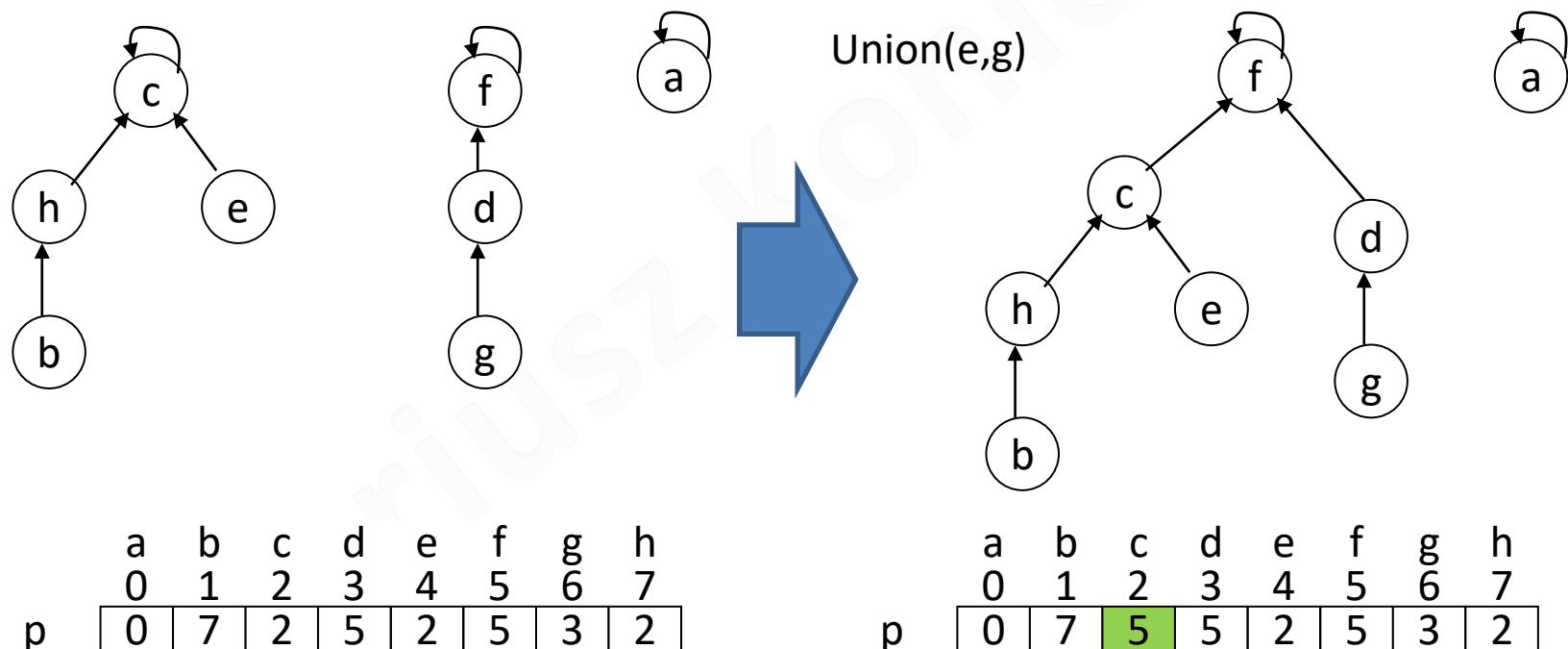


Union(g,e)



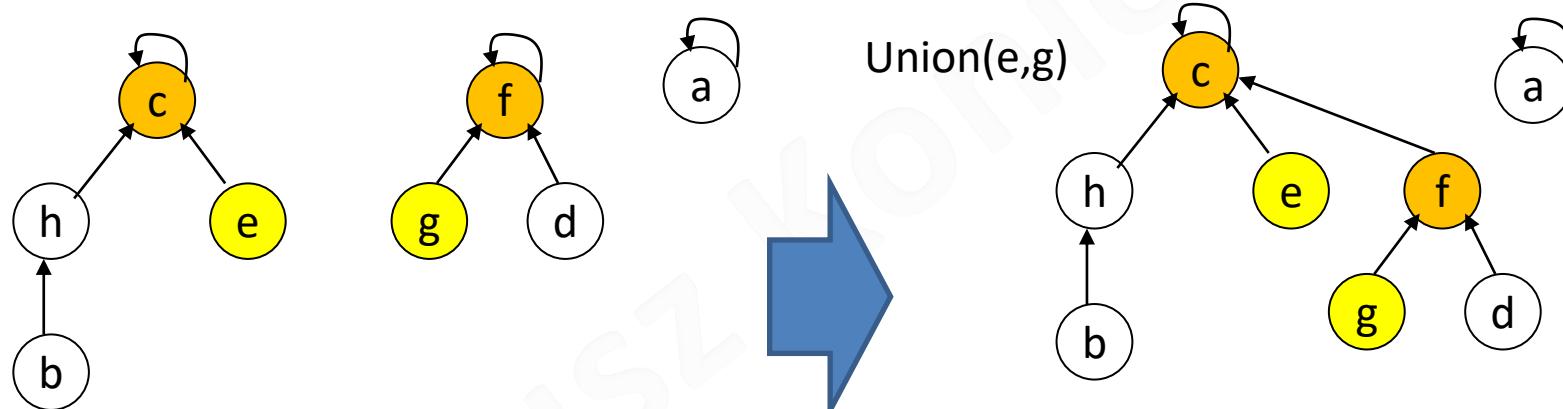
Las zbiorów rozłącznych

- W lesie zbiorów rozłącznych każdy element zawiera jedynie wskaźnik do swojego ojca w drzewie.
- **Korzeń** każdego drzewa zawiera **reprezentanta**, a jego wskaźnik do ojca wskazuje na niego samego.
- **W praktyce** las zbiorów rozłącznych jest również (jak poprzednia implementacja) realizowany na **zwykłej tablicy obiektów**.



Heurystyki 1/2

- Łączenie wg rangi (potencjalnej maksymalnej wysokości) – analogicznie jak dla poprzedniej reprezentacji



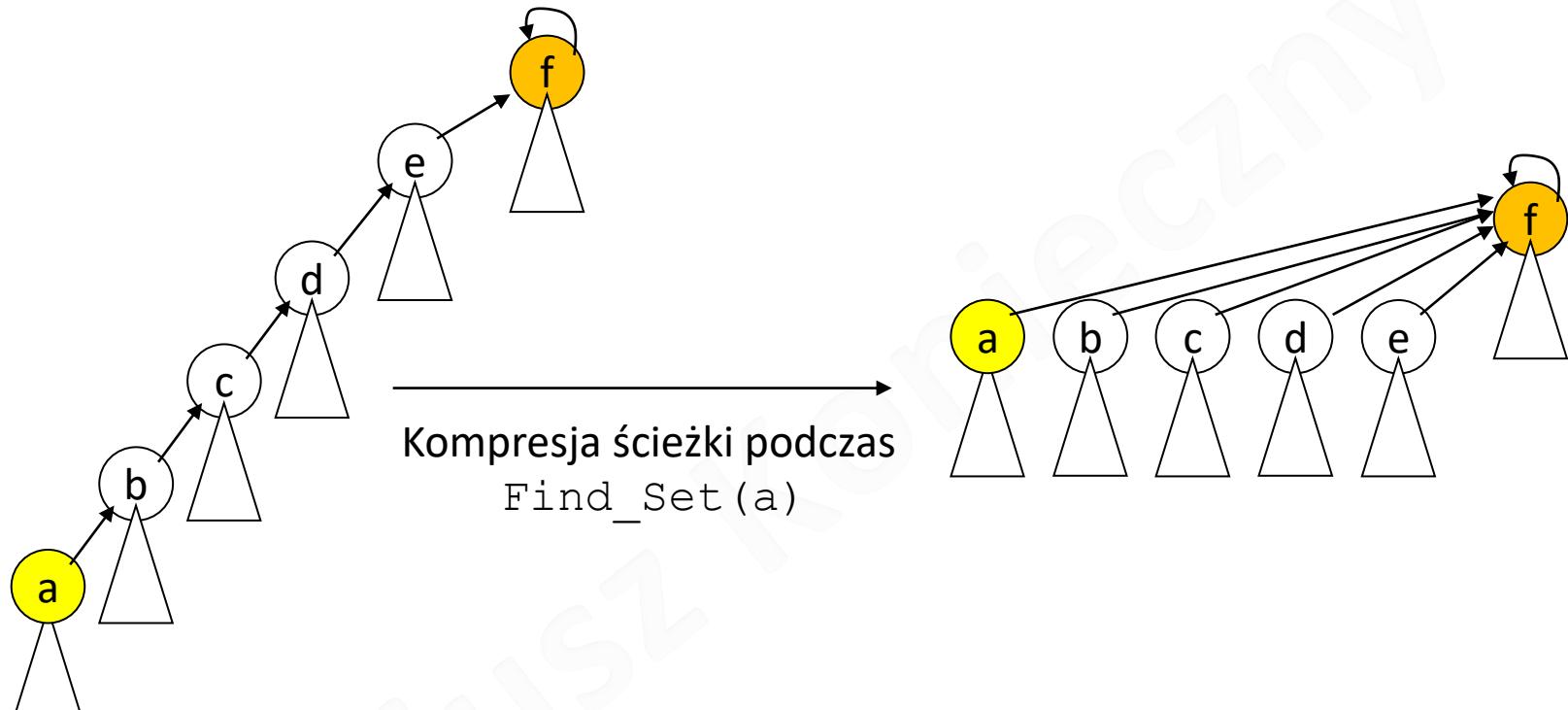
a	b	c	d	e	f	g	h
0	1	2	3	4	5	6	7
p	0	7	2	5	2	5	2
rank	0	0	2	0	0	1	0



a	b	c	d	e	f	g	h
0	1	2	3	4	5	6	7
p	0	7	2	5	2	2	5
rank	0	0	2	0	0	1	0

Heurystyki 2/2

- Kompresja ścieżki podczas operacji Find-Set
 - Przy powrocie z rekurencji aktualnia wskaźnik na rodzica wartością reprezentanta.



	a	b	c	d	e	f	g	h
0	0	1	2	3	4	5	6	7
p	1	2	3	4	5	5	?	?
rank	?	?	?	?	?	?	?	?

Find_Set(a)

→

	a	b	c	d	e	f	g	h
0	0	1	2	3	4	5	6	7
p	5	5	5	5	5	5	?	?
rank	?	?	?	?	?	?	?	?

Las zbiorów rozłącznych - kod

```
Make-Set (x)
{ 1} p[x] :=x
{ 2} rank[x] :=0
```

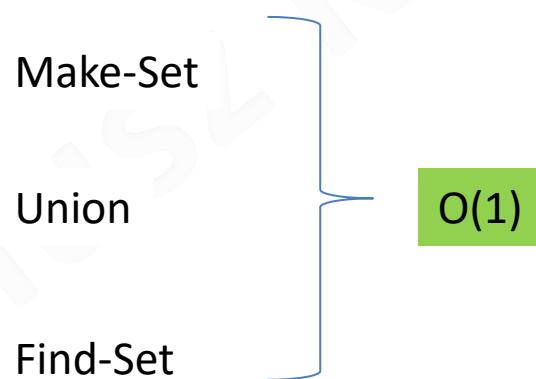
```
Union (x, y)
{ 1} Link(Find-Set(x), Find-Set(y))
```

```
Link (x, y)
{ 1} if rank[x]>rank[y] then
{ 2}     p[y]:=x
{ 3} else
{ 4}     p[x]:=y
{ 5}     if rank[x]=rank[y] then
{ 6}         rank[y]:=rank[y]+1
```

```
Find-Set (x)
{ 1} if x!=p[x] then
{ 2}     p[x] := Find-Set(p[x])
{ 3} return p[x]
```

Las zbiorów rozłącznych - złożoność

- Jeśli zastosujemy zarówno łączenie według rangi, jak i kompresję ścieżki, to czas działania dla m operacji w najgorszym przypadku wynosi $O(m \alpha(m,n))$, gdzie $\alpha(m,n)$ jest **bardzo wolno rosnącą** funkcją będącą odwrotnością funkcji Ackermanna. W każdym wyobrażalnym zastosowaniu struktury danych dla zbiorów rozłącznych $\alpha(m,n) \leq 4$ (gdzie np. n to liczba atomów we wszechświecie równa 10^{80}); możemy zatem traktować czas wykonania m operacji jako liniowy względem m we wszystkich praktycznych sytuacjach.
- Stąd złożoność czasową każdej pojedynczej operacji uznajemy za $O(1)$.



Algorytmy i struktury danych – W10

Teoria grafów cz.1

Wstęp – definicje, reprezentacja

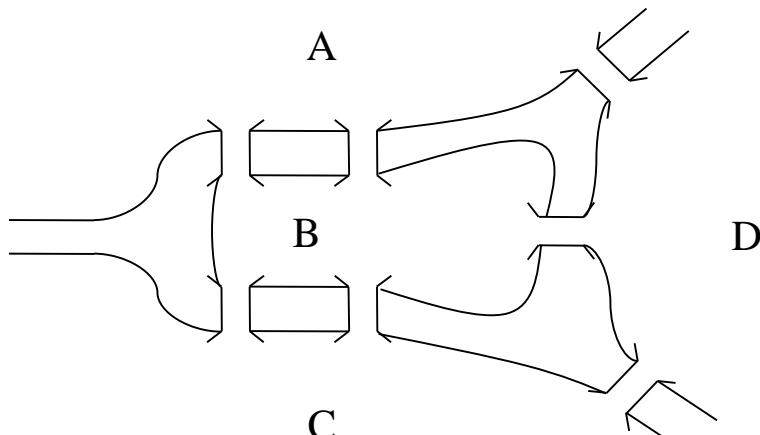
Składowe spójne

Przeszukiwanie grafu wszerz i w głąb

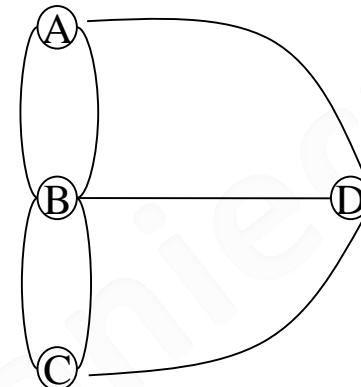
Zawartość

- Wstęp do teorii grafów
 - Definicje grafu, digrafu
 - Definicje z teorii grafów: ścieżki, cyklu itd.
- Reprezentacje w pamięci komputera
 - Macierz sąsiedztwa
 - Tablica list sąsiedztwa
 - Macierz incydencji
- Algorytm dla znajdowania komponentów spójnych
- Algorytm przeszukiwania wszerz (BFS)
- Algorytm przeszukiwania w głąb (DFS)

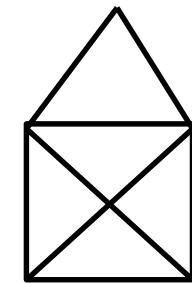
Siedem mostów w Królewcu (1736)



Dashed vertical line



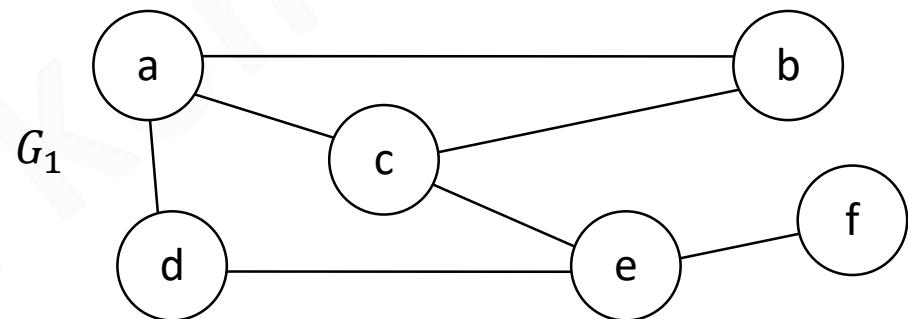
Stopień(A) = 3
Stopień(B) = 5
Stopień(C) = 3
Stopień(D) = 3



- Cyklem Eulera w grafie jest cykl, który przechodzi po każdej krawędzi dokładnie raz (choć może odwiedzać wierzchołki wielokrotnie).
- Graf spójny posiada cykl Eulera jeśli stopień(v) jest parzysty dla każdego wierzchołka v .
- Droga Eulera – ścieżka w grafie, która używa każdej krawędzi dokładnie raz.
- Droga Eulera w grafie spójnym istnieje jeśli co najwyżej dwa wierzchołki mają nieparzysty stopień.

Graf - definicja

- **Graf G** (zwany również **grafem nieskierowanym**) jest uporządkowaną parą:
 $G=(V,E)$
gdzie:
V – skończony zbiór punktów zwanych **wierzchołkami** (ang. *vertices*)
E – skończony zbiór **krawędzi** (ang. *edges*), z których każda łączy parę wierzchołków. Wierzchołki należące do krawędzi nazywamy jego jej końcami.
 $e \in E$ jest **nieuporządkowaną** parą (u, v) , gdzie $u, v \in V$ (wierzchołki u i v są połączone)



- Przykład:
 $G_1=(V_1,E_1)$
 $V_1=\{a,b,c,d,e,f\}$
 $E_1=\{(a,c),(b,a),(a,d),(d,e),(c,e),(e,f),(c,b)\}$

- W teorii istnieją również grafy **nieskończone**, jednak nie będziemy się nimi zajmować na wykładzie

Digraf (graf skierowany) - definicja

- **Digraf (graf skierowany – ang. *digraph, directed graph*)** G jest uporządkowaną parą:

$$G = (V, A)$$

gdzie:

V – skończony zbiór punktów zwanych **wierzchołkami** (ang. *vertices*)

A – skończony zbiór **krawędzi skierowanych** lub **łuków** (ang. *arcs*) łączących parę wierzchołków.

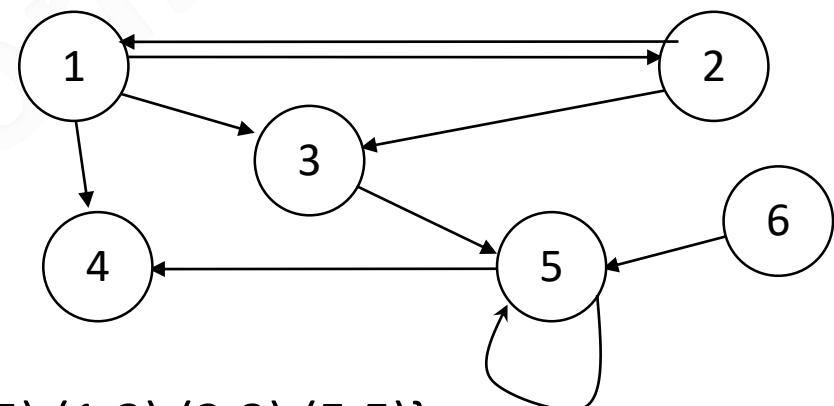
$e \in A$ jest **uporządkowaną** parą (u, v) , gdzie $u, v \in V$ (istnieje połączenie **z u do v**)

- Przykład:

$$G_2 = (V_2, E_2)$$

$$V_2 = \{1, 2, 3, 4, 5, 6\}$$

$$E_2 = \{(1, 4), (5, 4), (1, 2), (6, 5), (2, 1), (3, 5), (1, 3), (2, 3), (5, 5)\}$$



Rozwinięcia definicji grafu

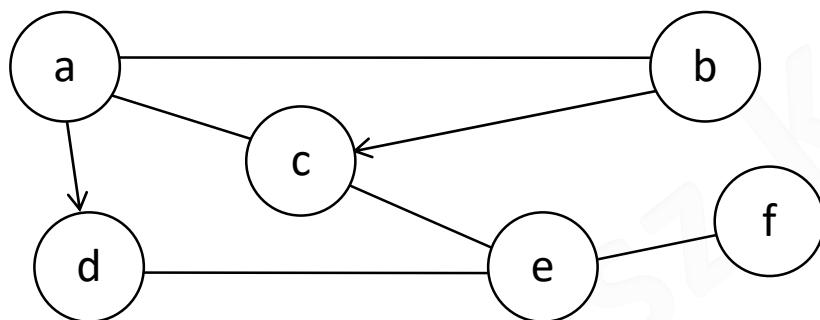
- I) Graf mieszany:

$$G=(V,E,A)$$

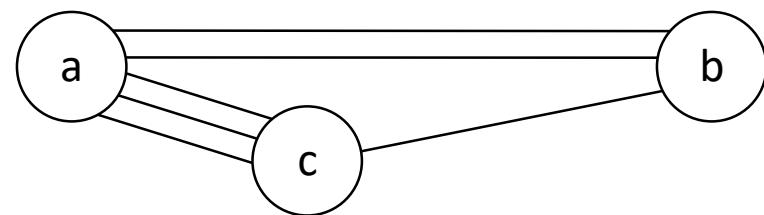
gdzie:

V, E, A – definicje jak wcześniej

- II) Multigraf (pseudograf), gdzie mogą występować krawędzie wielokrotne



I)

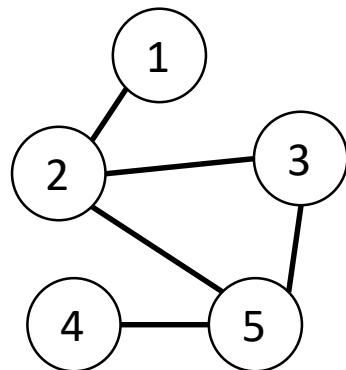


II)

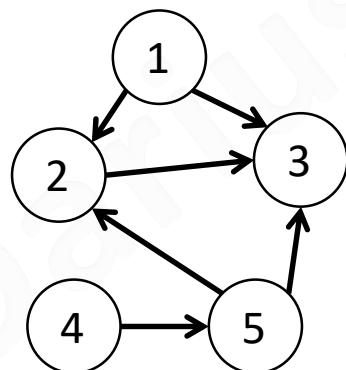
Reprezentacja grafu – macierz sąsiedztwa

- **Macierzą sąsiedztwa** grafu $G = (V, E)$ jest macierz $n \times n$ (gdzie $n=|V|$) $A = (a_{i,j})$, gdzie elementy zdefiniowane są następująco:

$$a_{i,j} = \begin{cases} 1 & (v_i, v_j) \in E \\ 0 & w \quad p.p. \end{cases}$$



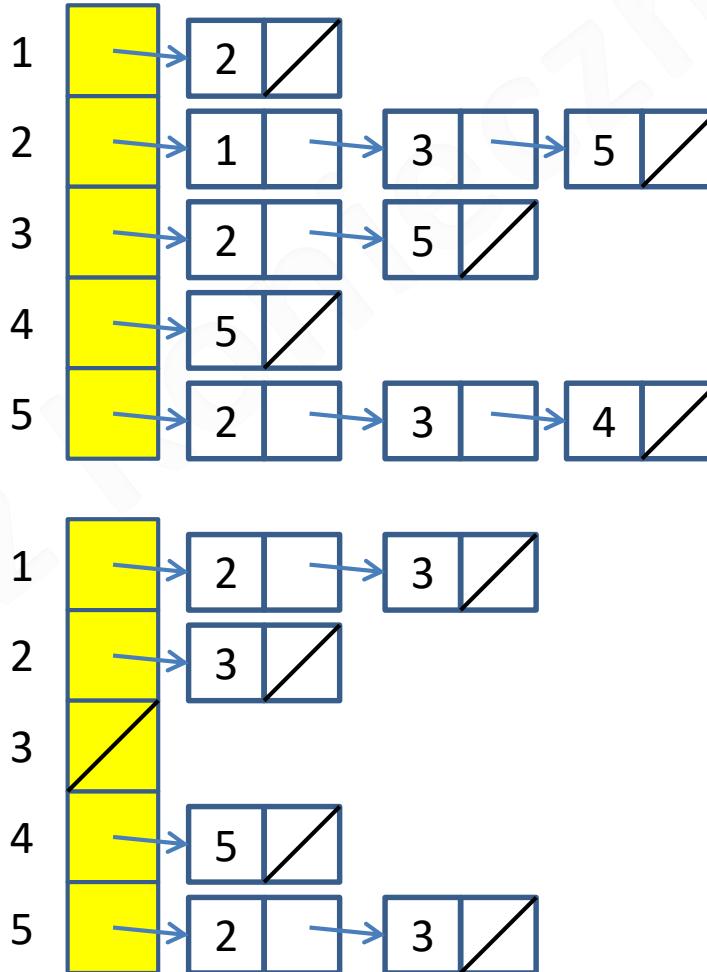
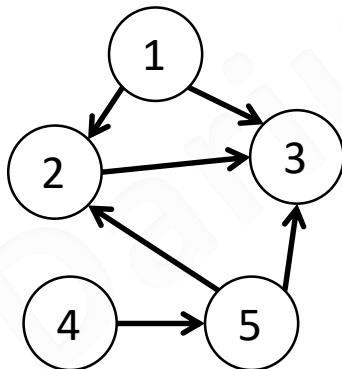
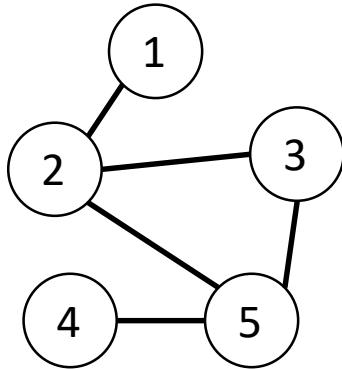
$$A = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 0 & 1 & 0 & 0 & 0 \\ 1 & 0 & 1 & 0 & 1 \\ 0 & 1 & 0 & 0 & 1 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 1 & 0 \end{bmatrix}$$



$$A = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 & 0 \end{bmatrix}$$

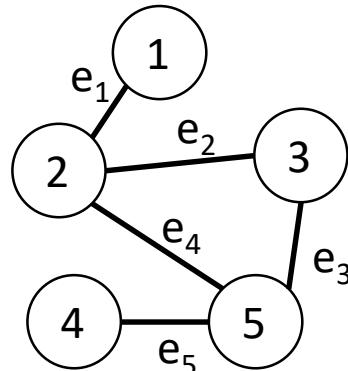
Reprezentacja grafu – lista sąsiedztwa

- Listą sąsiedztwa grafu $G = (V, E)$ jest tablica list $\text{Adj}[1..|V|]$. Dla każdego wierzchołka $v \in V$, $\text{Adj}[v]$ jest listą wiązaną wierzchołków sąsiadujących z wierzchołkiem v .

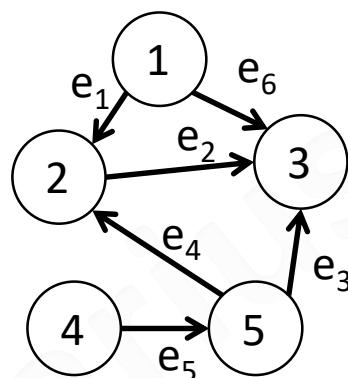


Reprezentacja grafu – macierz incydencji

- Mniej popularna, tylko do części algorytmów przydatna
- Wymaga dużo pamięci $O(|V|^*|E|)$
- Każda kolumna opisuje **jedną krawędź**: wartości 1 lub -1 są w wierszach oznaczających końce krawędzi



$$A = \begin{bmatrix} & \text{krawędzie} \\ & e_1 \quad e_2 \quad e_3 \quad e_4 \quad e_5 \\ \text{krawędzie} \\ 1 & 1 \quad 0 \quad 0 \quad 0 \quad 0 \\ 2 & 1 \quad 1 \quad 0 \quad 1 \quad 0 \\ 3 & 0 \quad 1 \quad 1 \quad 0 \quad 0 \\ 4 & 0 \quad 0 \quad 0 \quad 0 \quad 1 \\ 5 & 0 \quad 0 \quad 1 \quad 1 \quad 1 \end{bmatrix}$$

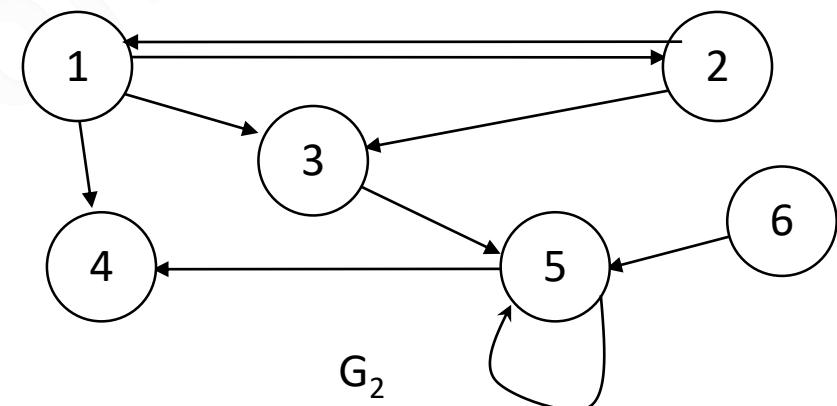
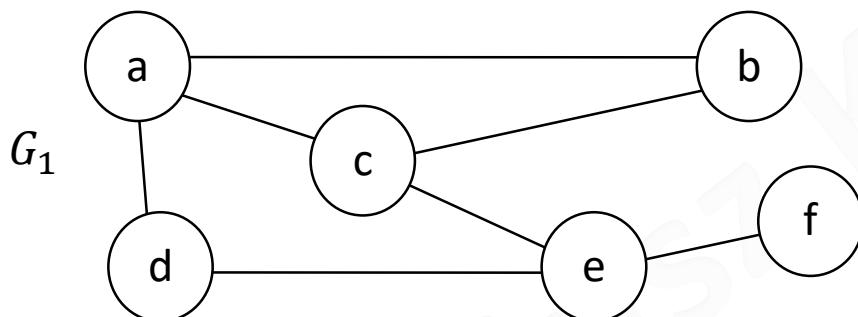


$$A = \begin{bmatrix} & \text{krawędzie} \\ & e_1 \quad e_2 \quad e_3 \quad e_4 \quad e_5 \quad e_6 \\ \text{krawędzie} \\ 1 & -1 \quad 0 \quad 0 \quad 0 \quad 0 \quad -1 \\ 2 & 1 \quad -1 \quad 0 \quad 1 \quad 0 \quad 0 \\ 3 & 0 \quad 1 \quad 1 \quad 0 \quad 0 \quad 1 \\ 4 & 0 \quad 0 \quad 0 \quad 0 \quad -1 \quad 0 \\ 5 & 0 \quad 0 \quad -1 \quad -1 \quad 1 \quad 0 \end{bmatrix}$$

- Istnieje wiele innych reprezentacji grafu w pamięci komputera przystosowanych do konkretnych grafów, sytuacji, algorytmów.

Definicje 1/9

- **Uwaga ogólna:** ile książek tyle definicji !
- **Stopień wierzchołka** v to liczba $\deg(v)$ krawędzi, do których należy wierzchołek v
 - W grafach skierowanych wierzchołek ma:
 - **stopień wychodzący** $\degOut(v)$ – czyli liczba krawędzi, gdzie wierzchołek v jest na pierwszym miejscu w krawędzi
 - **stopień wchodzący** $\degIn(v)$ – czyli liczba krawędzi, gdzie wierzchołek v jest na drugim miejscu w krawędzi
- **Wierzchołek izolowany** – wierzchołek o stopniu 0

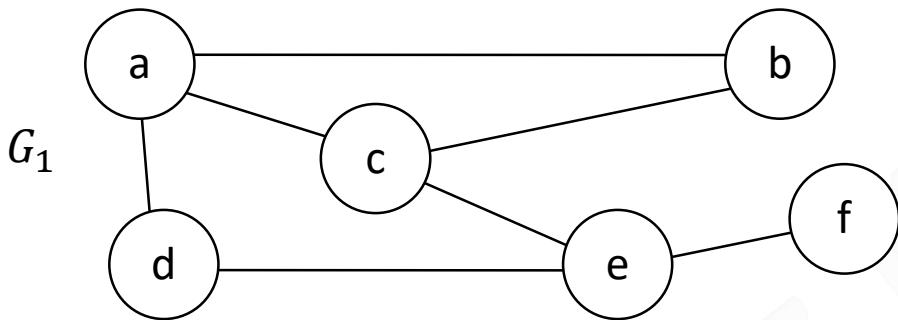


np.
 $\deg(c)=3$
 $\deg(f)=1$

np.
 $\degOut(3)=1$
 $\degIn(3)=2$

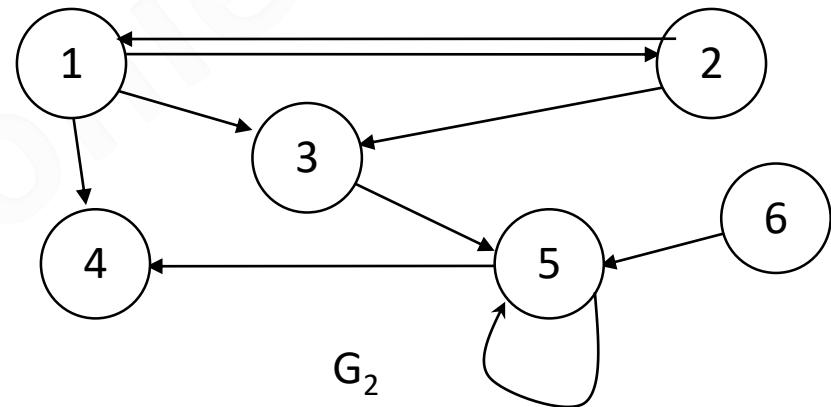
Definicje 2/9

- **Wierzchołek v jest incydentny z krawędzią e**, jeśli jest jednym z jej końców.
- Mówimy, że **wierzchołki sąsiadują ze sobą**, jeśli łączy je krawędź.
- Dwie **krawędzie są incydentne/sąsiednie**, gdy łączy je wierzchołek.
 - W grafach skierowanych łączący wierzchołek dla jednej krawędzi musi być początkowym, dla drugiej końcowym wierzchołkiem



Np.

a jest incydentny z (a,c)
c sąsiaduje z e
(d,e) jest incydentne z (e,f)

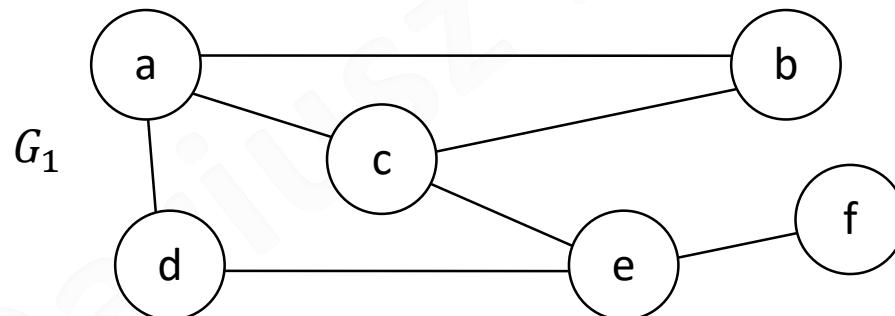


np.

4 jest incydentny z (5,4)
(2,1) jest incydentny z (1,4)
(2,1) nie jest incydentny z (2,3)

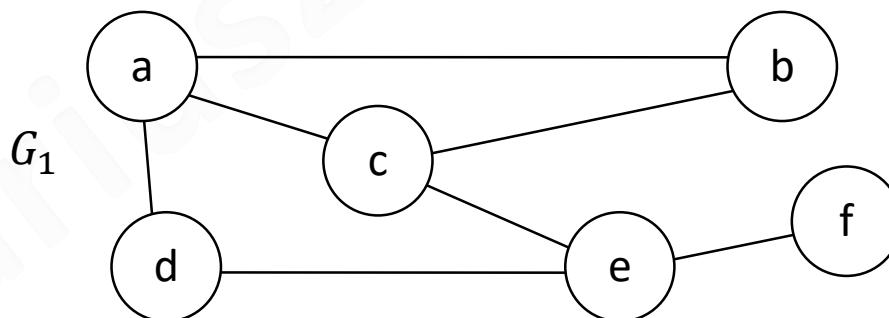
Definicje 3/9

- **Ścieżką z wierzchołka v do wierzchołka u** nazywamy taki ciąg $(v_0, v_1, v_2, \dots, v_k)$, że:
 $v_0=v$, $v_k=u$ oraz $(v_i, v_{i+1}) \in E$ dla każdego $i = 0, 1, \dots, k-1$
oraz żadne kolejne krawędzie na ścieżce nie są identyczne
 - Np. dla grafu G_1 ścieżka $p_1=(a,c,e,d,a,b)$
 - Np. dla grafu G_1 ścieżka nie jest $p_2=(b,c,d,e)$ bo nie istnieje krawędź (c,d)
- **Długość ścieżki** to liczba wystąpień krawędzi należących do ścieżki
 - Długość ścieżki p_1 wynosi 5



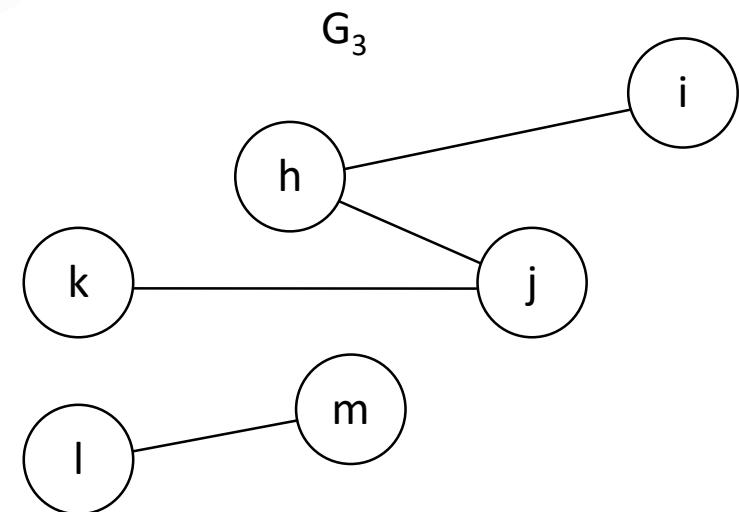
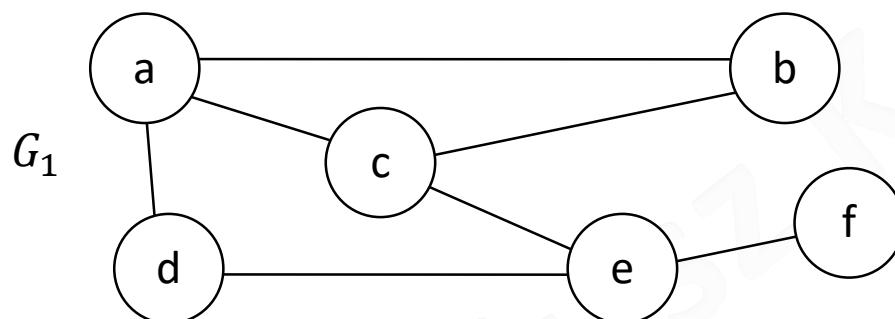
Definicje 4/9

- **Ścieżka prosta** to ścieżka bez powtarzających się krawędzi
- **Droga** to ścieżka bez powtarzających się (oprócz ewentualnie pierwszego i ostatniego) **wierzchołków** na ścieżce.
 - W grafie G_1 ścieżka $p_1=(a,c,e,d,a,b)$ nie jest drogą, drogą jest $p_2=(b,c,e,d)$
- **Cykl** to ścieżka zamknięta, gdzie pierwszy i ostatni wierzchołek na ścieżce jest ten sam.
- **Cykl prosty** to cykl będący drogą.
 - Przykładowy cykl w grafie G_1 to $p_3=(c,e,d,e)$
- **Cykl własny/pętla własna** – krawędź z wierzchołka do tego samego wierzchołka (dopuszczalny raczej tylko w digrafie lub multigrafie)
- Mówimy, że graf jest **acykliczny**, gdy nie zawiera żadnego cyklu.



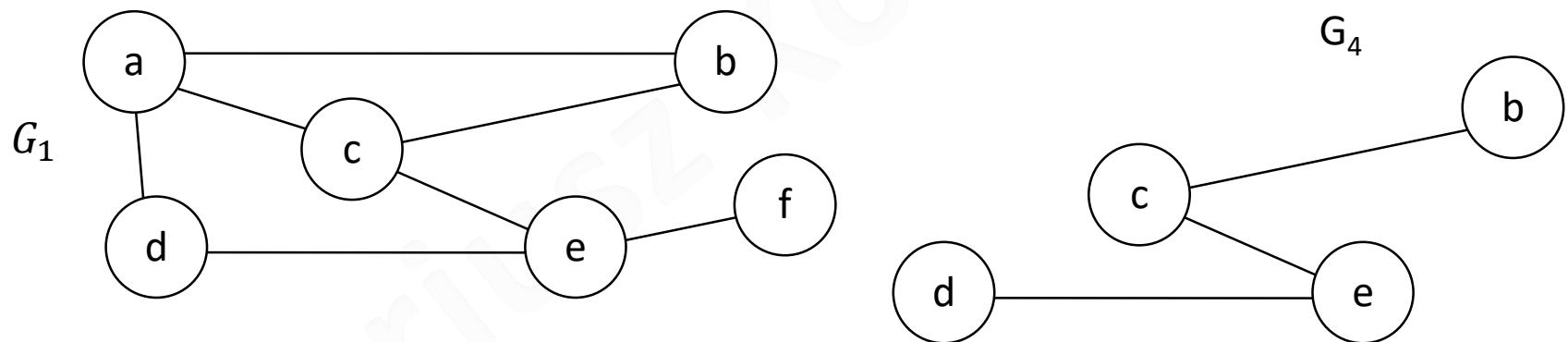
Definicje 5/9

- Graf nazywamy **spójnym**, gdy każda para wierzchołków jest połączona ścieżką.
 - Dla grafu nieskierowanego definicję stosuje się bezpośrednio
 - Dla grafu skierowanego przekształca się go na graf podstawowy (bez kierunków na krawędziach)
- Graf G_1 jest spójny, graf G_3 nie jest spójny.



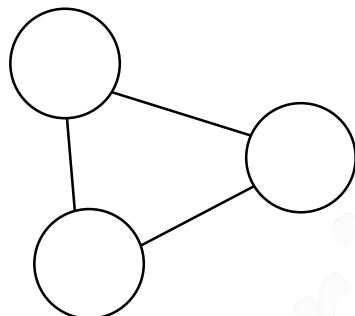
Definicje 6/9

- Graf $G_I = (V_I, E_I)$ jest **podgrafem** grafu $G = (V, E)$ jeśli $V_I \subseteq V$ oraz $E_I \subseteq E$.
- **Podgraf** grafu G **indukowany przez** zbiór wierzchołków $V_I \subseteq V$ jest grafem $G_I = (V_I, E_I)$, gdzie $E_I = \{ (u, v) \in E \mid u, v \in V_I \}$
 - Podgraf grafu G_1 indukowany zbiorem wierzchołków $\{b, c, d, e\}$, to graf G_4

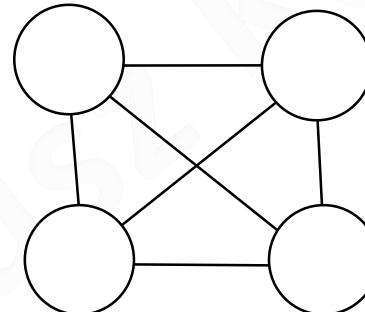


Definicje 7/9

- Graf nazywały **pełnym**, jeśli każda para wierzchołków jest połączona krawędzią. Nieskierowany graf pełny ma $|E| = |V| * (|V| - 1)/2$ krawędzi.
- **Gęstość** grafu – stosunek liczby krawędzi do największej możliwej liczby krawędzi.
 - Graf nazywamy **gęstym**, gdy liczba jego krawędzi jest rzędu liczby krawędzi grafu pełnego, czyli $|E| = O(|V|^2)$
 - Graf nazywamy **rzadkim** w przeciwnym wypadku
 - Najczęściej zakładamy, że liczba krawędzi w grafie rzadkim $|E| = O(|V|)$

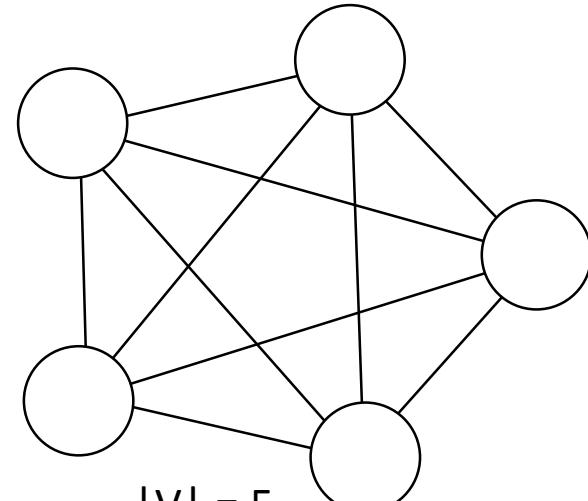


$$|V| = 3$$
$$|E| = 3$$



$$|V| = 4$$
$$|E| = 6$$

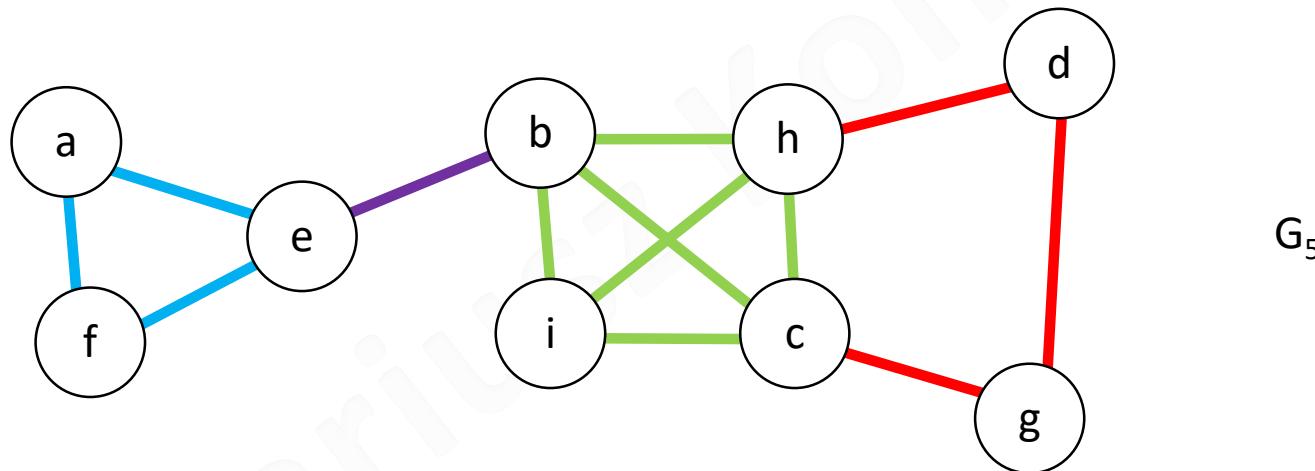
Grafy pełne



$$|V| = 5$$
$$|E| = 10$$

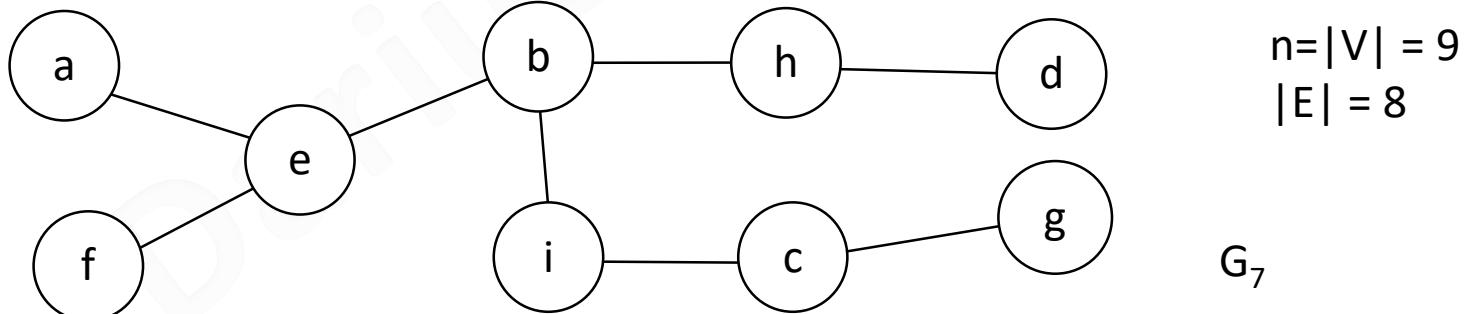
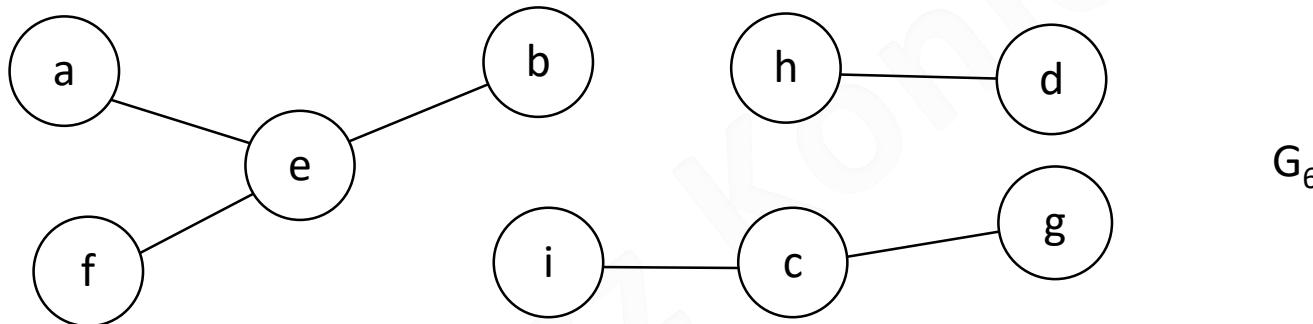
Definicje 8/9

- **Klika** – podgraf będący grafem pełnym
 - Podgraf grafu G_5 indukowany wierzchołkami $\{b,c,h,i\}$ jest kliką
 - Podgraf grafu G_5 indukowany wierzchołkami $\{a,e,f\}$ jest kliką
 - Podgraf grafu G_5 indukowany wierzchołkami $\{e,b\}$ jest kliką
 - Podgraf grafu G_5 indukowany wierzchołkami $\{c,d,g,h\}$ **nie** jest kliką



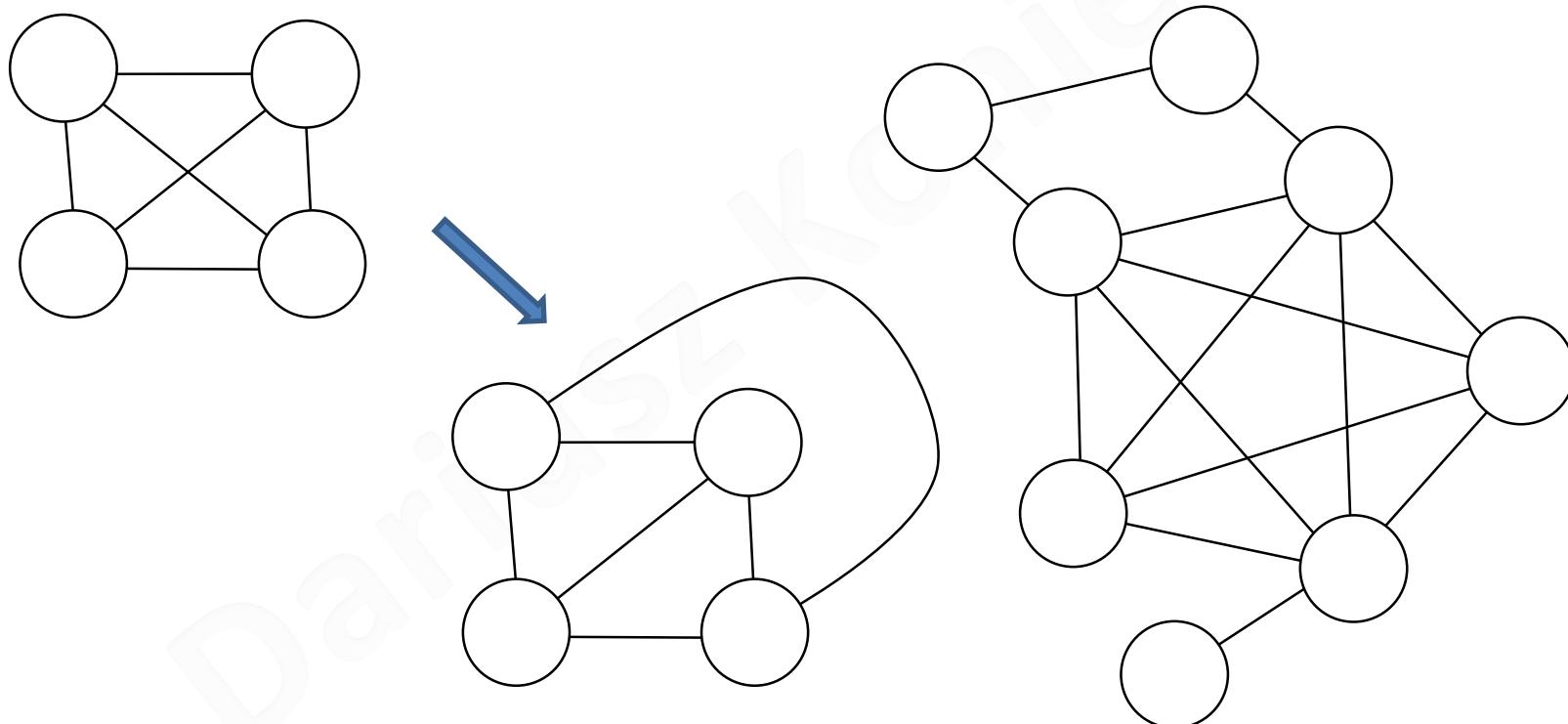
Definicje 9/9

- **Las** – inna nazwa grafu acyklicznego
 - Grafy G_6 i G_7 są lasami
- **Drzewo** – acykliczny spójny graf
 - Graf G_7 (ale nie G_6) jest drzewem
- Jeśli $G = (V, E)$ jest **drzewem**, to $|E| = |V| - 1$
- Często w rozważaniach nt. złożoności obliczeniowej/pamięciowej jako rozmiar przyjmuje się liczbę wierzchołków, czyli $n = |V|$



Inne pojęcia i problemy teorii grafów 1/4

- **Graf planarny** – możliwy do narysowania na płaszczyźnie bez przecinania się krawędzi.
 - Problem: Czy dany graf jest planarny?
 - Graf pełny o 4 wierzchołkach jest planarny, ale każdy graf pełny o większej liczbie wierzchołków już nie jest planarny.



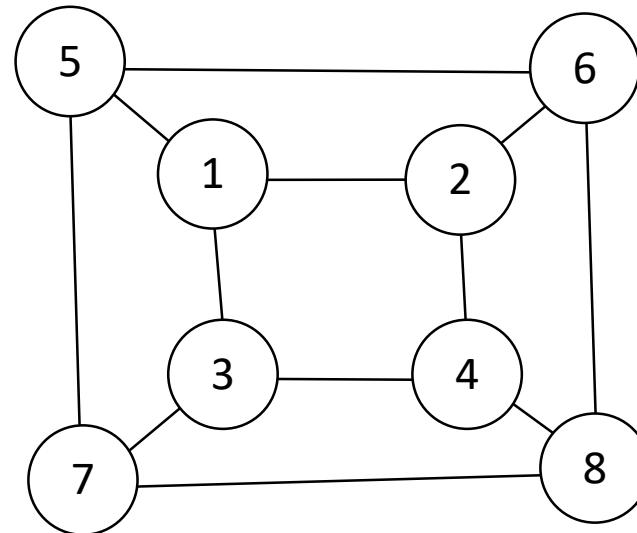
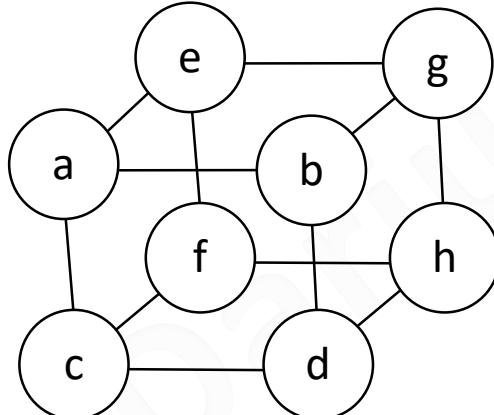
Inne pojęcia i problemy teorii grafów 2/4

- **Kolorowanie grafu:** nadanie każdemu wierzchołkowi koloru tak, aby żadne sąsiadujące ze sobą wierzchołki nie były pokolorowane tym samym kolorem
 - Problem: minimalna liczba kolorów do pokolorowania danego grafu.
- **Graf eulerowski** – posiada ścieżkę przechodzącą przez każdą krawędź dokładnie raz
 - Problem: czy dany graf jest grafem eulerowskim
- **Graf hamiltonowski** – posiada ścieżkę przechodzącą przez każdy wierzchołek dokładnie raz
 - Problem: czy dany graf jest grafem hamiltonowskim

Inne pojęcia i problemy teorii grafów 3/4

- **Izomorfizm grafów**

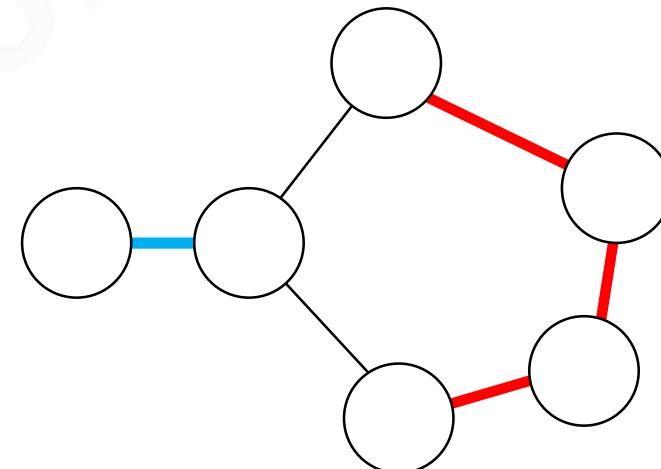
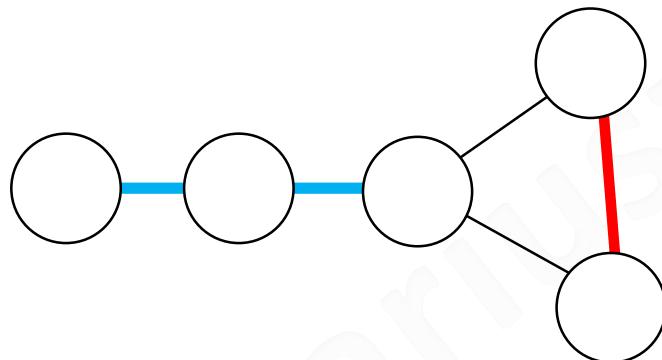
- Graficzna reprezentacja grafów (w postaci kropek i łączących je krzywych) jest tylko sposobem przedstawienia relacji zachodzących między wierzchołkami. Dla każdego grafu istnieje nieskończoność wielu przedstawiających go jednoznacznie wykresów (*rysunków*). Właściwości grafów są niezależne od sposobu numerowania wierzchołków, kolejności ich rysowania itp. Grafy różniące się tylko sposobem ich przedstawienia, lub indeksami nadanymi wierzchołkom, nazywamy izomorficznymi.
- Problem: czy dane dwa grafy są izomorficzne?



Inne pojęcia i problemy teorii grafów 4/4

- **Homeomorfizm grafów**

- Dwa grafy są *homeomorficzne*, jeśli z jednego grafu można otrzymać drugi zastępując wybrane krawędzie łańcuchami prostymi lub łańcuchy proste pojedynczymi krawędziami ("dorysowywanie" na krawędziach dowolnej liczby wierzchołków bądź wymazywanie ich).
- Problem: czy dane dwa grafy są homeomorficzne?



Graf ważony

- **Ważony graf** (lub digraf) G jest trójką uporządkowaną (V, E, w) , gdzie:
 V – zbiór wierzchołków,
 E – zbiór krawędzi,
natomiast
 w jest **funkcją** $w: E \rightarrow \mathbb{R}$ z wartościami rzeczywistymi zdefiniowanymi na zbiorze krawędzi.

– Zamiast zapisu $w((v_1, v_2))$ będziemy pisać w skrócie $w(v_1, v_2)$

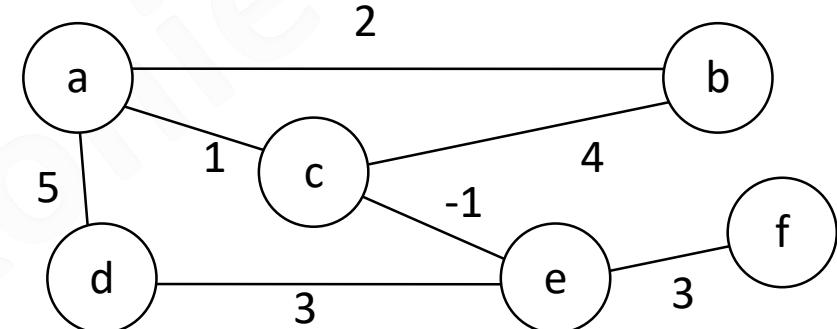
- **Wagą grafu** jest suma wag jego krawędzi
- **Wagą ścieżki** jest sumą wag jej krawędzi.

- Przykład:

$$G_1 = (V_1, E_1, w)$$

$$V_1 = \{a, b, c, d, e, f\}$$

$$E_1 = \{(a,c), (b,a), (a,d), (d,e), (c,e), (e,f), (c,b)\}$$



$e \in E_1$	(a,c)	(b,a)	(a,d)	(d,e)	(c,e)	(e,f)	(c,b)
$w(e)$	1	2	5	3	-1	3	4

- Waga grafu G_1 wynosi 17
- Waga ścieżki $p_1 = (a,c,e,d,a,b)$ wynosi 10

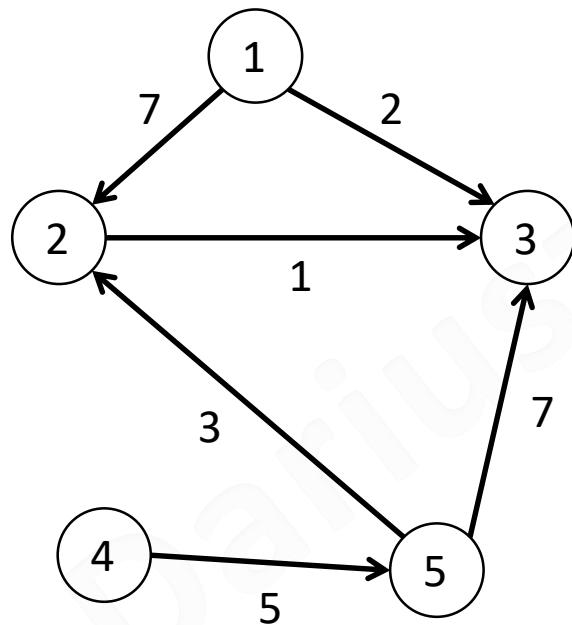
Grafy - zastosowania

- Za pomocą grafu można prawie wszystko zamodelować:
 - Sieci
 - Komputerowa
 - Drogowa
 - Ulic
 - ...
 - Zależności międzyludzkie
 - Procesy:
 - Technologiczne
 - Komputerowe
 - Przepływ pracy
 - Połączenia międzycząsteczkowe
 - Mapy
 - Relacje w bazie danych
 - ...

Macierz sąsiedztwa dla grafów ważonych

- Macierzą sąsiedztwa grafu **ważonego** $G = (V, E, W)$ jest macierz $n \times n$ (gdzie $n=|V|$) $A = (a_{i,j})$, gdzie elementy zdefiniowane są następująco:

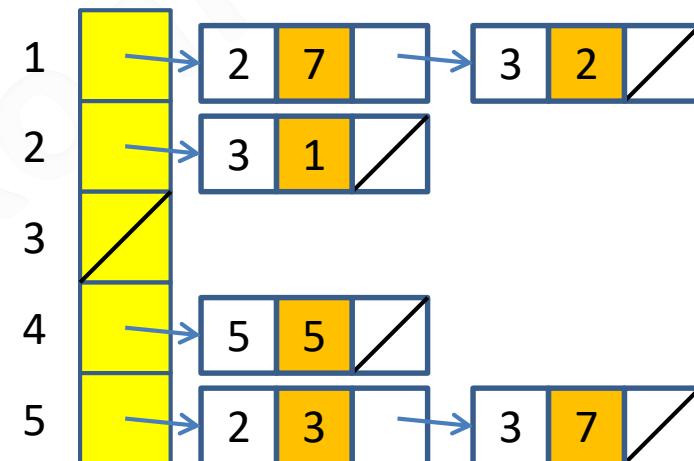
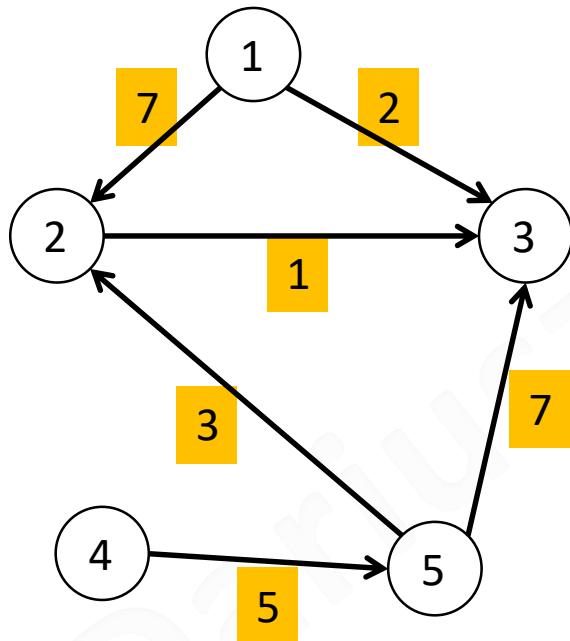
$$a_{i,j} = \begin{cases} w(v_i, v_j) & \text{gdy } (v_i, v_j) \in E \\ 0 & \text{gdy } i = j \\ \infty & \text{w p.p.} \end{cases}$$



$$A = \begin{bmatrix} 1 & 2 & 3 & 4 & 5 \\ 1 & 0 & 7 & 2 & \infty & \infty \\ 2 & \infty & 0 & 1 & \infty & \infty \\ 3 & \infty & \infty & 0 & \infty & \infty \\ 4 & \infty & \infty & \infty & 0 & 5 \\ 5 & \infty & 3 & 7 & \infty & 0 \end{bmatrix}$$

Lista sąsiedztwa dla grafów ważonych

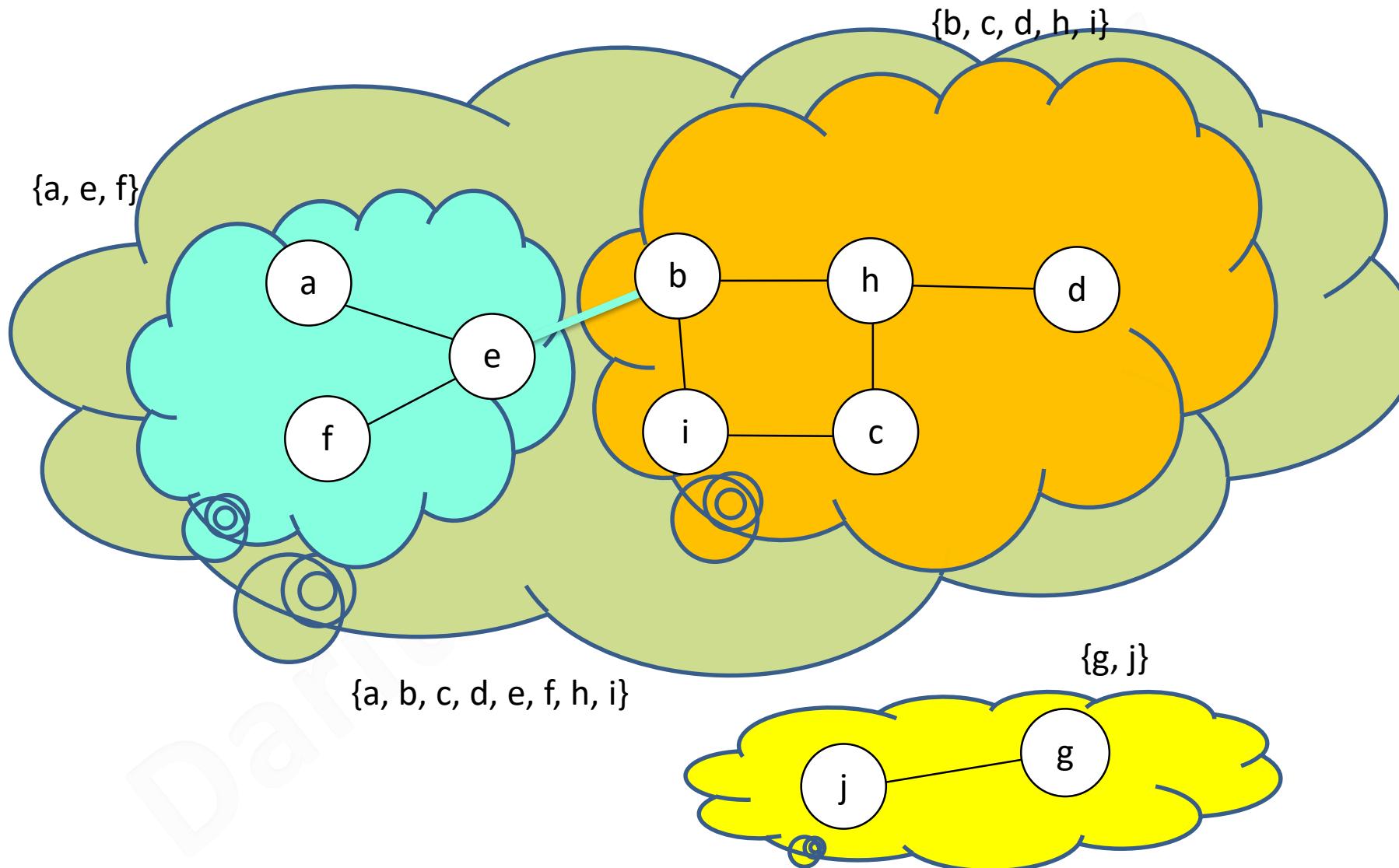
- Analogicznie jak dla grafów nieważonych, dodatkowo w każdym elemencie listy pamiętamy wagę krawędzi



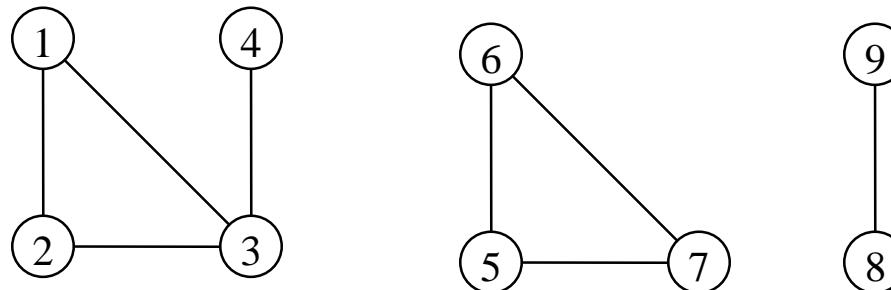
Składowe spójne

- **Spójna składowa** (lub po prostu **składowa** – ang. *connected component*) grafu nieskierowanego G – spójny podgraf grafu G nie zawarty w większym podgrafie spójnym grafu G.
- Dwa wierzchołki należą do tej samej składowej wtedy i tylko wtedy, gdy istnieje ścieżka pomiędzy nimi.
- Problem: **podział grafu na składowe spójne** (dla grafu nieskierowanego)
- **Idea algorytmu:** Nie trzeba szukać dokładnej ścieżki między dwoma wierzchołkami.
 - Analizujemy kolejne krawędzie (w dowolnej kolejności, każda krawędź dokładnie raz)
 - Wystarczy pamiętać w danej chwili algorytmu między którymi wierzchołkami istnieje jakaś ścieżka jako zbiór wierzchołków między którymi istnieją ścieżki. Zbiory takich ścieżek są rozłączne. Na początku sa to zbiory jednoelementowe (dla każdego wierzchołka jeden zbiór)
 - Gdy analizuje krawędź między wierzchołkami należącymi do różnych zbiorów, np. A i B, to znaczy, że używając jej można znaleźć (przez nią) ścieżkę z każdego wierzchołka ze zbioru A do każdego wierzchołka ze zbioru B. Zatem musimy połączyć te dwa zbiory.
 - Jeśli krawędź łączy wierzchołki z tego samego zbioru, do znaleźliśmy inną ścieżkę (zapewne też cykl), ale nic to nie zmienia w podziale wierzchołków na zbiory

Przykładowy krok algorytmu - analiza krawędzi (b,e)



Składowe spójne – przykład i algorytm



Graf z trzema składowymi spójnymi

- Sposób rozwiązania wskazuje na użycie struktury do pamiętania rozłącznych zbiorów
 - Czyli lasu zbiorów rozłącznych (DSF)

```
ConnectedComponents ( $G$ )
{ 1 }   for dla każdego wierzchołka  $u \in V[G]$  do
{ 2 }     MakeSet( $u$ )
{ 3 }   for dla każdej krawędzi  $(u, v) \in E$  do
{ 4 }     if FindSet( $u$ )  $\neq$  FindSet( $v$ ) then
{ 5 }       Union( $u, v$ )
```

Złożoność $\Theta(|E| + |V|)$

- Po skończeniu algorytmu reprezentant zbioru wyznacza składową spójną

Składowe spójne - przykład

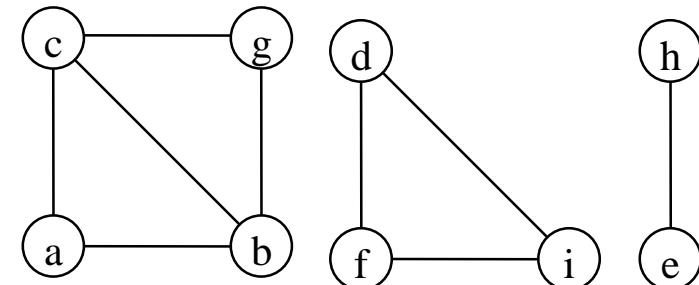
$$V = \{ a, b, c, d, e, f, g, h, i \}$$

$$E = \{ (a,c), (d,f), (d,i), (f,i), (e,h), (b,g), (a,b), (b,c), (c,g) \}$$

{a}	{a,c}	{a,c}	{a,c}	{a,c}	{a,c}	{a,c}	{a,c,b,g}
{b}	{b}	{b}	{b}	{b}	{b}	{b,g}	
{c}							
{d}	{d}	{d,f}	{d,f,i}	{d,f,i}	{d,f,i}	{d,f,i}	{d,f,i}
{e}	{e}	{e}	{e}	{e}	{e,h}	{e,h}	{e,h}
{f}	{f}						
{g}	{g}	{g}	{g}	{g}	{g}		
{h}	{h}	{h}	{h}	{h}	{h}		
{i}	{i}	{i}					

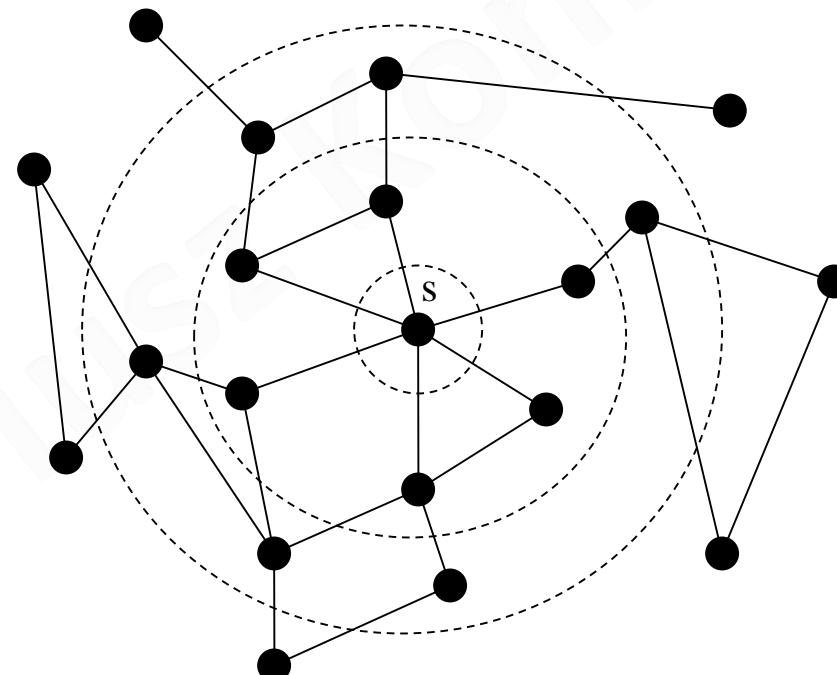
Złożoność (lista sąsiedztwa): $O(|V| + |E|)$

Złożoność (macierz sąsiedztwa): $O(|V|^2)$



Przeszukiwanie wszerz (BFS)

- **Przeszukiwanie grafu** – odwiedzenie wszystkich (możliwych do odwiedzenia) wierzchołków grafu w systematyczny sposób.
- Dany jest graf $G = (V, E)$ oraz wyróżniony wierzchołek początkowy (zwany źródłem) s , w **przeszukiwaniu wszerz** (ang. breadth first search - BFS) chcemy aby granica między wierzchołkami odwiedzonymi i nieodwiedzonymi przekraczana była jednocześnie na całej szerokości. Tzn. wierzchołki w odległości k od źródła są odwiedzane przed wierzchołkami w odległości $k+1$.
- Algorytm będzie budował **drzewo** przeszukiwań wszerz



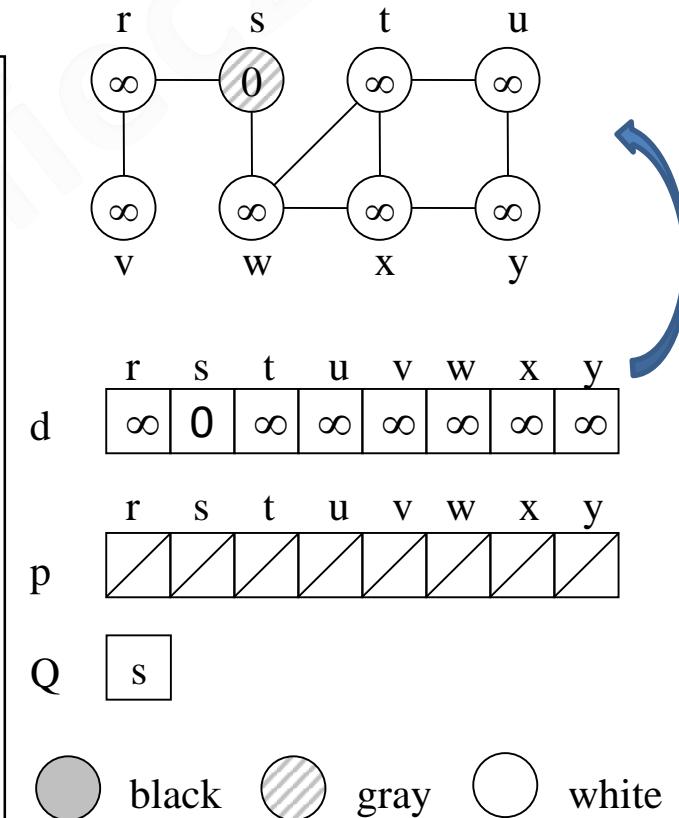
BFS - kod

- BFS koloruje każdy wierzchołek na biało, szaro lub czarno (tablica `color`).
- BFS tworzy drzewo przeszukiwań wszerz początkowo zawierające tylko korzeń, którym jest wierzchołek źródłowy s
- Poprzednik na ścieżce do korzenia wierzchołka u jest przechowywany w pozycji tablicy $p[u]$
- Odległość (mierzona w liczbie krawędzi) od źródła s do wierzchołka u obliczana przez algorytm jest przechowywana w tablicy pod pozycją $d[u]$.

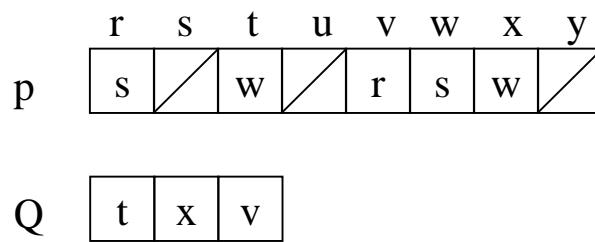
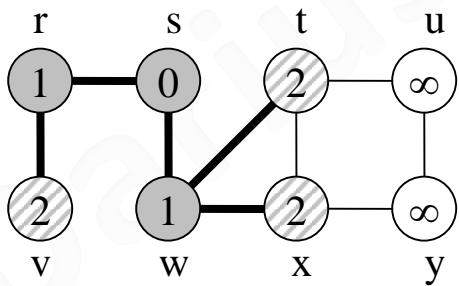
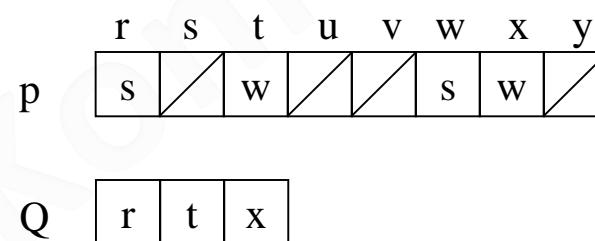
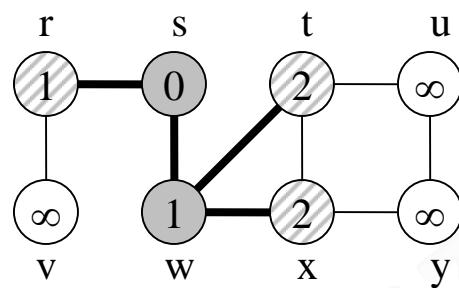
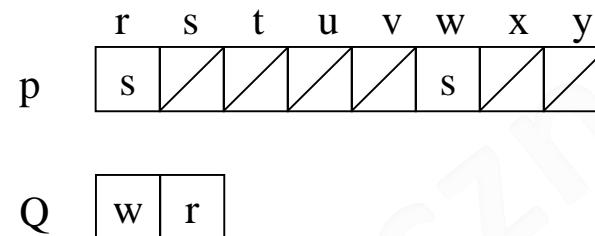
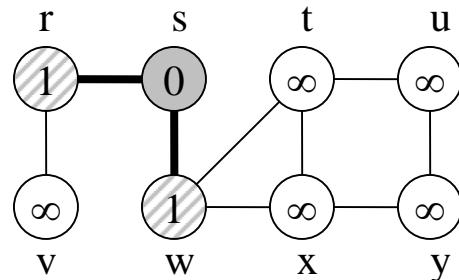
```

BFS( $G, s$ )
{ 1} for każdy wierzchołek  $u \in V[G] - \{s\}$ 
{ 2}   do  $color[u] := \text{WHITE}$ 
{ 3}      $d[u] := \infty$ 
{ 4}      $p[u] := \text{NIL}$ 
{ 5}  $color[s] := \text{GREY}$ 
{ 6}  $d[s] := 0$ 
{ 7}  $p[s] := \text{NIL}$ 
{ 8} Enqueue( $\mathcal{Q}, s$ )
{ 9} while not Empty( $\mathcal{Q}$ )
{10}   do  $u := \text{Dequeue}(\mathcal{Q})$ 
{11}     for każdy  $v \in \text{Adj}[u]$ 
{12}       do if  $color[v] = \text{WHITE}$ 
{13}         then  $color[v] := \text{GREY}$ 
{14}            $d[v] := d[u] + 1$ 
{15}            $p[v] := u$ 
{16}           Enqueue( $\mathcal{Q}, v$ )
{17}      $color[u] := \text{BLACK}$ 

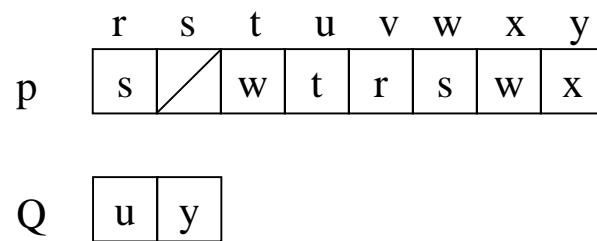
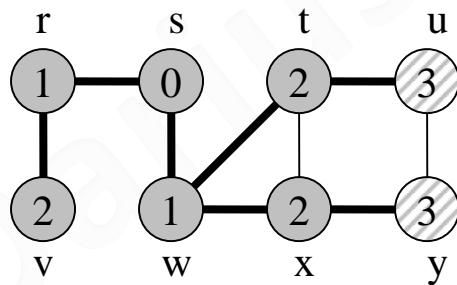
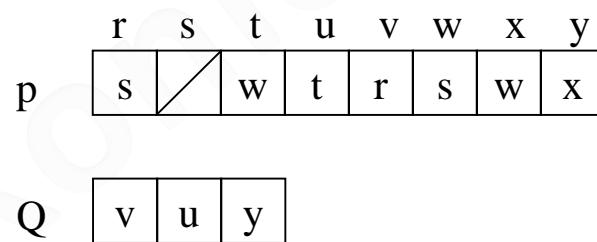
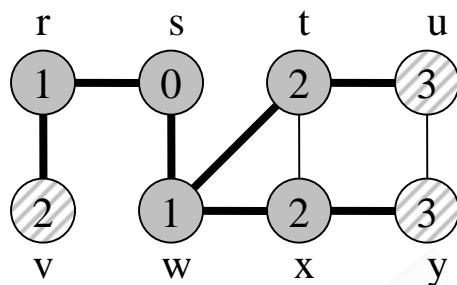
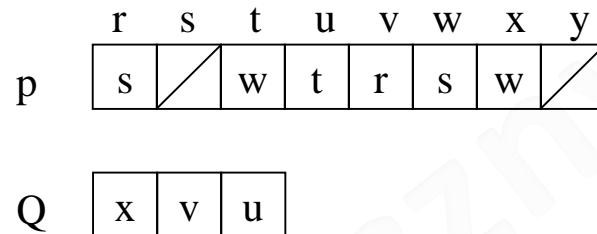
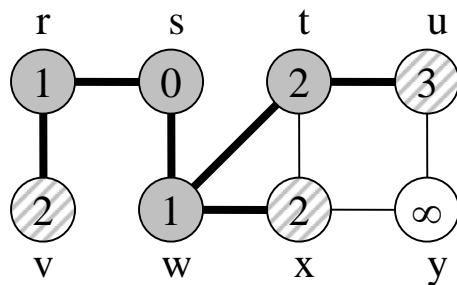
```



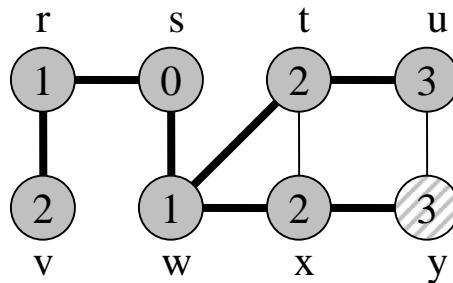
BFS – przykład 1/3



BFS – przykład 2/3

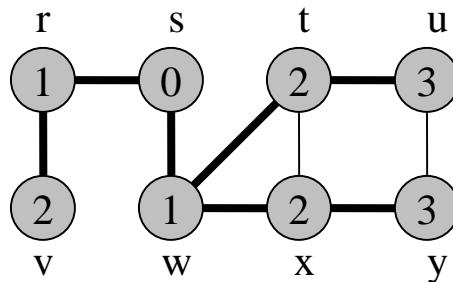


BFS – przykład 3/3



p	r	s	t	u	v	w	x	y
	s	/	w	t	r	s	w	x

Q y



p	r	s	t	u	v	w	x	y
	s	/	w	t	r	s	w	x

Q

```
PrintPath(G, s, v)
{ 1} if v = s
{ 2}   then print s
{ 3} else if p[v] = null
{ 4}   then print „brak ścieżki z” s „do” v „exists”
{ 5}   else PrintPath(G, s, p[v])
{ 6}   print v
```

Złożoność (lista sąsiedztwa): $O(|V|+|E|)$

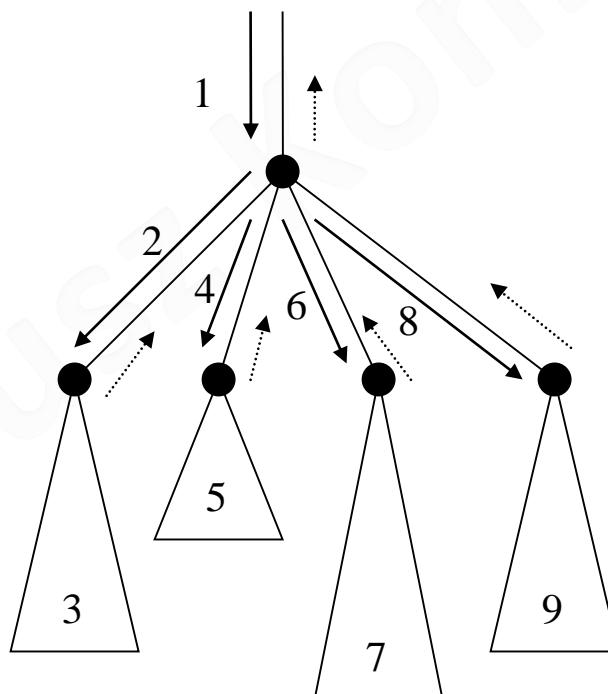
Złożoność (macierz sąsiedztwa): $O(|V|^2)$

BFS - zastosowanie

- Powstaje drzewo minimalnych ścieżek (jako liczba krawędzi)
- Może być wstępem do rozwiązania wielu innych problemów z teorii grafów.
 - W sieciach przepływowych poszukiwanie ścieżki rozszerzającej
- Używane przy rozwiązywaniu problemów, gdy graf budowany jest w trakcie przeszukiwania (przestrzeń stanów/ drzewa gry)
 - Gry logiczne – szachy itp.

Przeszukiwanie w głąb

- Ogólna zasada: sięganie głębiej jeśli tylko się da.
- Badanie krawędzi jeszcze nie odwiedzonej i przejście po tej krawędzi do kolejnego wierzchołka, jeśli jeszcze nie był odwiedzony.
- Jeśli nie ma nieodwiedzonych wierzchołków sąsiadujących – powrót wierzchołka poprzedzającego na ścieżce.
- Powstaje las rozpinający
- Algorytm
 - koloruje wierzchołki (jak BSF): biały, szary, czarny
 - spamiętuje poprzedników – tablica p
 - Zapisuje etykiety czasowe : czas wejścia do wierzchołka w tablicy t oraz opuszczenia wierzchołka w tablicy f.

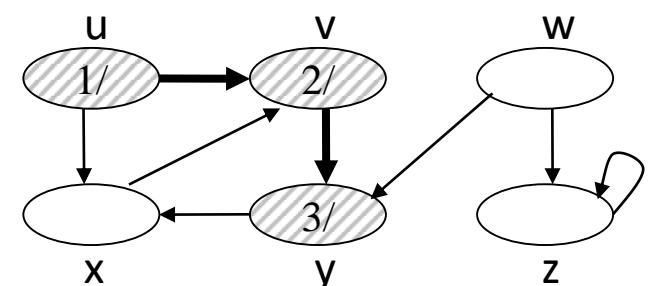
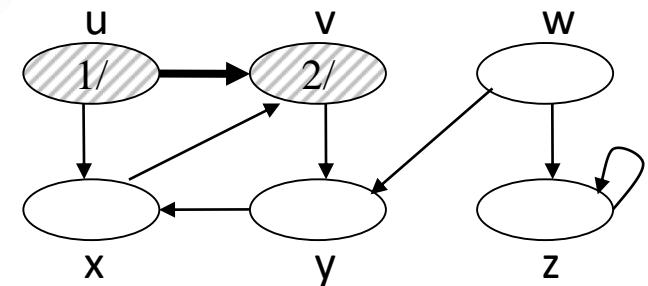
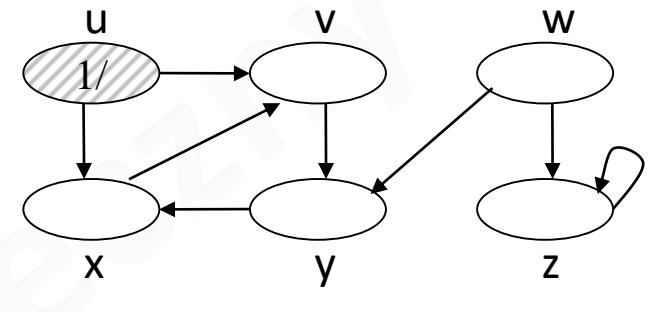


DFS - kod

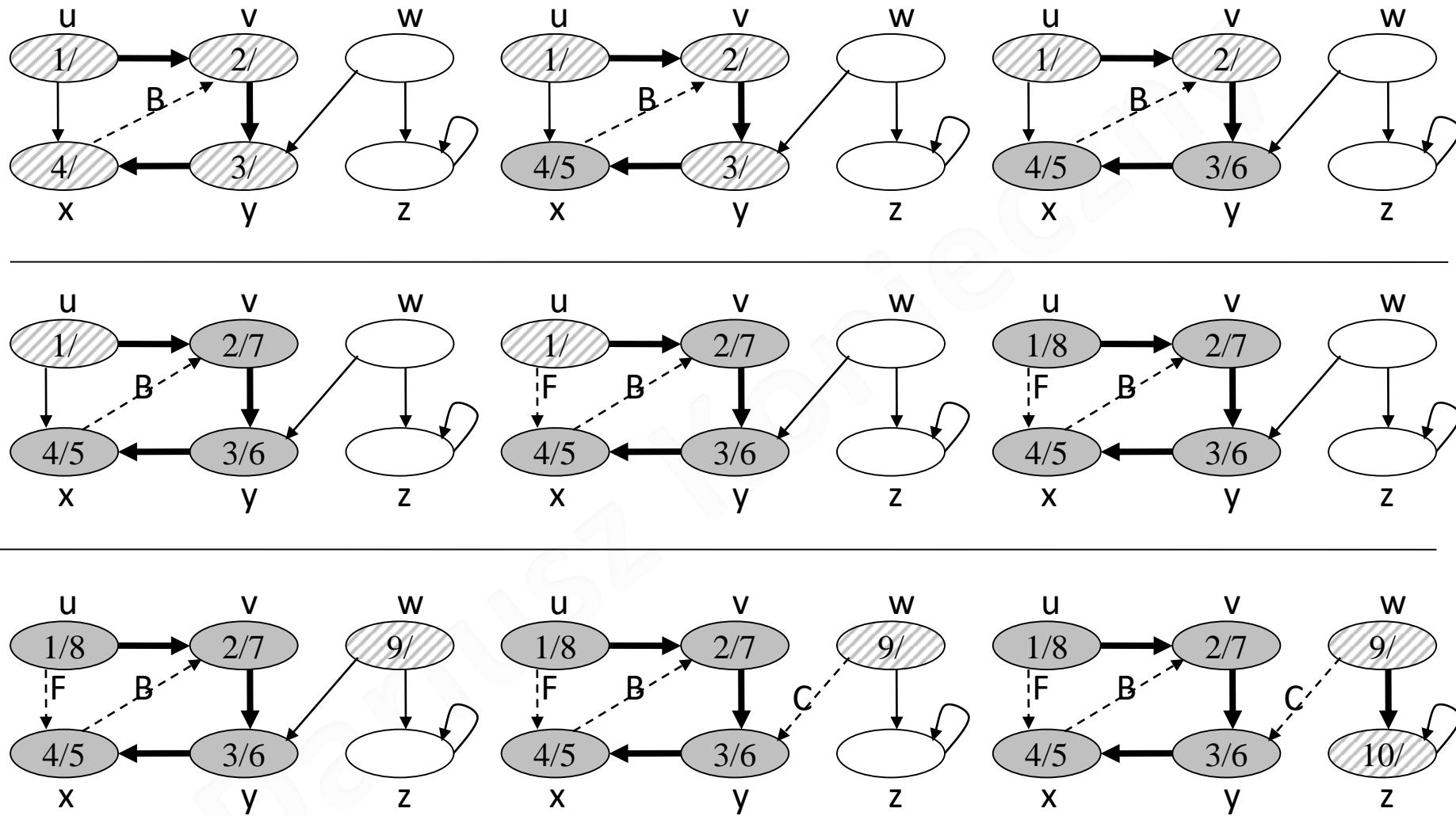
- Wersja rekurencyjna, „globalny” czas time

```
DFS(G)
{ 1}   for każdy wierzchołek  $u \in V[G]$  do
{ 2}       color[u] := WHITE
{ 3}       p[u] := NIL
{ 4}   time := 0
{ 5}   for każdy wierzchołek  $u \in V[G]$  do
{ 6}       if color[u] = WHITE then
{ 7}           DFS_Visit(u)
```

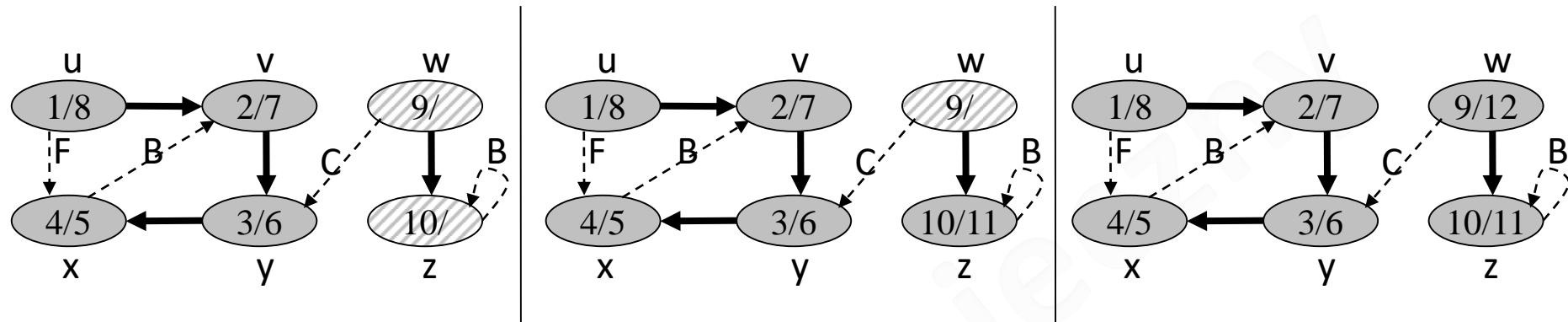
```
DFS_Visit(u)
{ 1}   color[u] := GREY
{ 2}   time := time + 1
{ 3}   t[u] := time
{ 4}   for każda krawędź  $v \in \text{Adj}[u]$  do
{ 5}       if color[v] = WHITE then
{ 6}           p[v] := u
{ 7}           DFS_Visit(v)
{ 8}   color[u] := BLACK
{ 9}   f[u] := time := time + 1;
```



DFS – przykład 1/2



DFS – przykład 2/2



B – krawędź powrotna (ang. *back edge*)

F – krawędź w przód (ang. *forward edge*)

C – krawędź poprzeczna (ang. *cross edge*)

Złożoność (lista sąsiedztwa): $O(|V|+|E|)$

Złożoność (macierz sąsiedztwa): $O(|V|^2)$

DFS - zastosowanie

- Składowa innych algorytmów z teorii grafów
 - Poszukiwanie mostów, czyli krawędzi dzielących graf na dwie składowe spójne
- Przeszukiwanie przestrzeni stanów, gdy BFS tworzy zbyt dużo stanów potrzebnych do zapamiętania, a przestrzeń stanów ma postać drzewa:
 - DFS musi pamiętać tylko ścieżkę od korzenia
 - Różne odmiany: min-max, alpha-beta, A* itp.

Algorytmy i struktury danych – W11

Teoria grafów cz.2

Minimalne drzewo rozpinające

Najkrótsze ścieżki z wybranego źródła

Najkrótsze ścieżki wszystkich par wierzchołków

Maksymalny przepływ

Zawartość

- Minimalne drzewo rozpinające
 - Algorytm Kruskala
- Najkrótsze ścieżki z wybranego źródła
 - Algorytm Dijkstry
- Minimalne drzewo rozpinające
 - Algorytm Prima
- Minimalne ścieżki między wszystkimi parami wierzchołków
 - Algorytm Floyda
- Sieci przepływowne
 - Maksymalny przepływ
 - Metoda Forda-Fulkersona
 - Algorytm Edmondsa-Karpa
 - Maksymalnie skojarzenie w grafie dwudzielnym

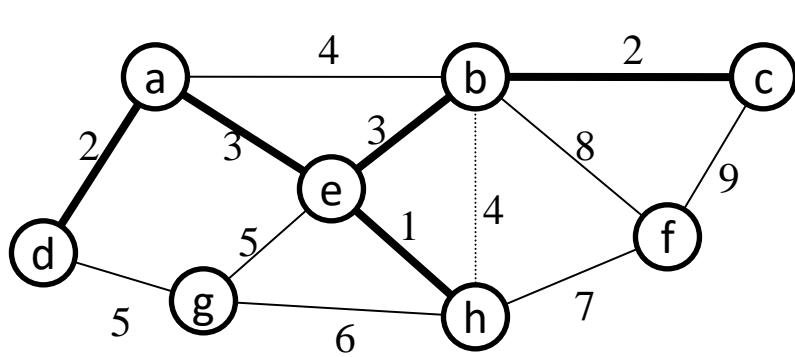
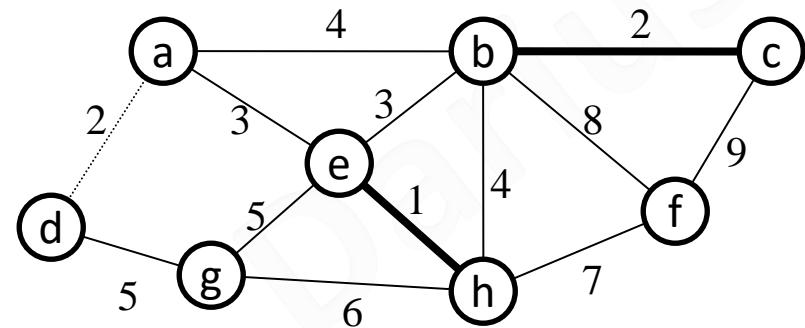
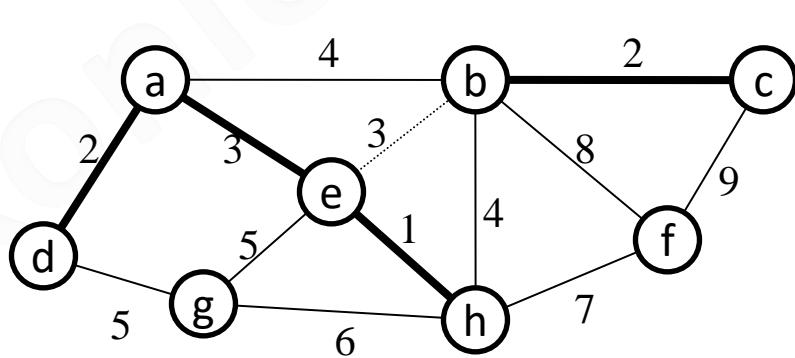
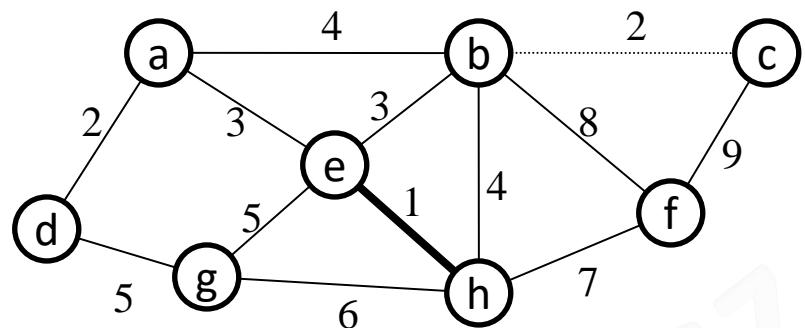
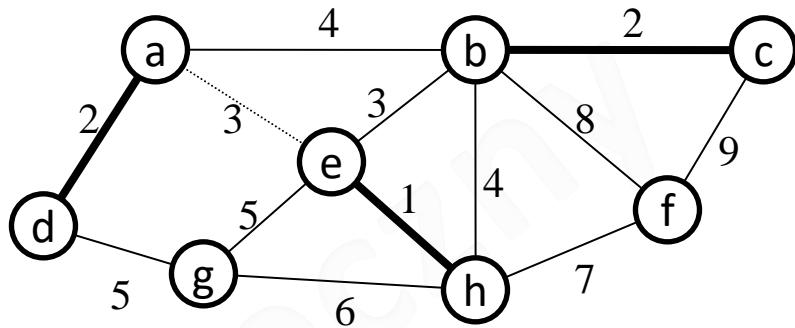
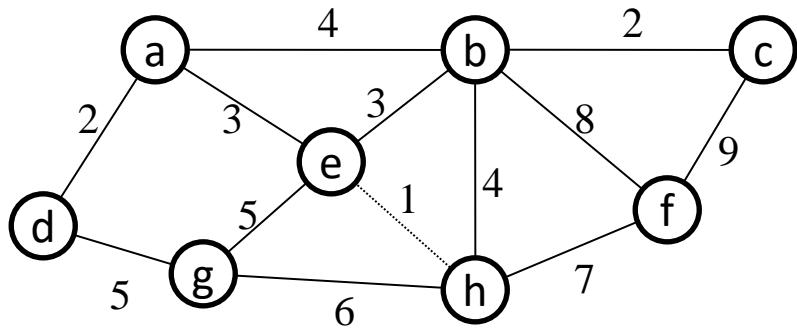
Minimalne drzewo rozpinające - MST

- Mają dany spójny, nieskierowany graf, **drzewo rozpinające** takiego grafu to taki podgraf, który jest drzewem i łączy ze sobą wszystkie wierzchołki.
- **Minimalne drzewo rozpinające** (lub **drzewo rozpinające z minimalną wagą**, ang. *minimum spanning tree*) jest drzewem rozpinającym z wagą nie większą niż waga każdego innego drzewa rozpinającego.

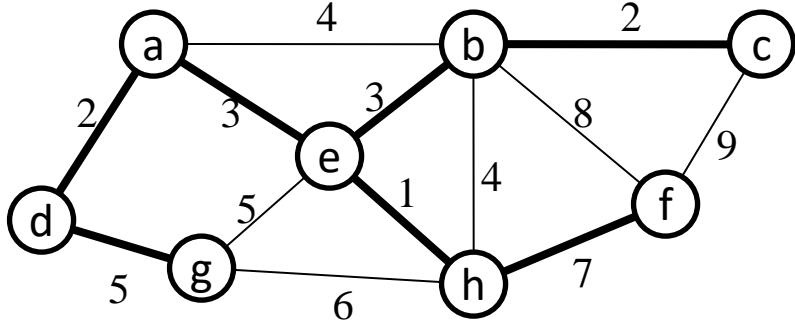
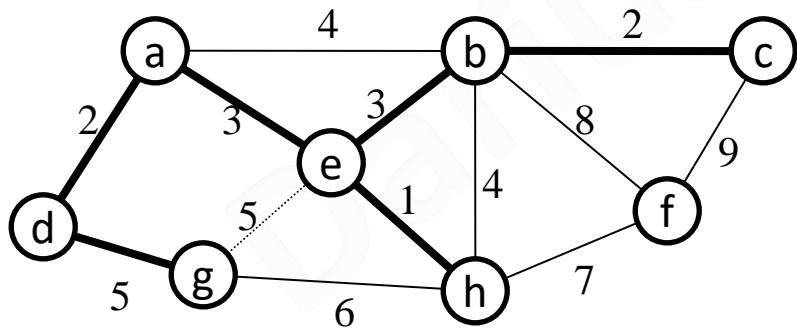
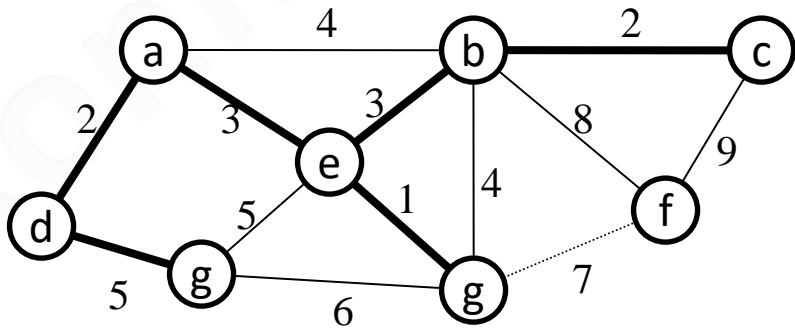
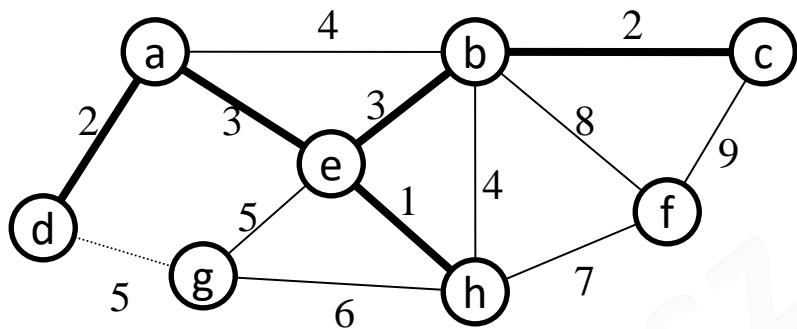
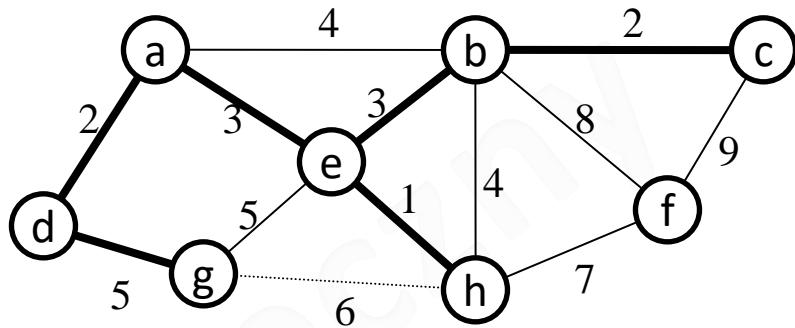
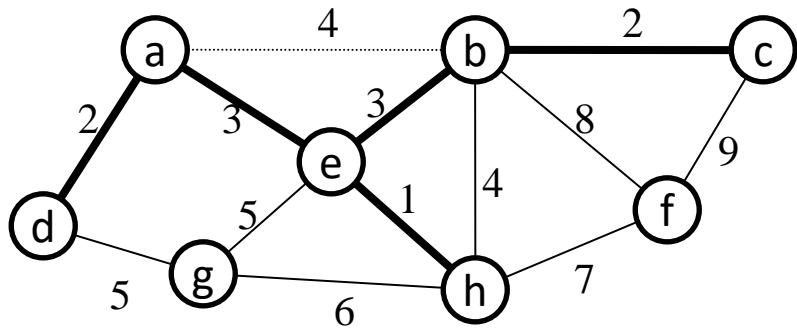
Algorytm Kruskala

1. Stwórz las F (zbiór drzew), gdzie każdy wierzchołek w grafie jest izolowanym wierzchołkiem.
2. Stwórz zbiór S zawierający wszystkie krawędzie w grafie
3. Dopóki S nie jest pusty
 - usuń z S krawędź o minimalnej wadze
 - jeśli krawędź łączy dwa różne drzewa, dodaj krawędź do drzewa F , łącząc te dwa drzewa w jedno
 - w p.p. odrzuć krawędź

Algorytm Kruskala – przykład 1/2



Algorytm Kruskala – przykład 2/2



Algorytm Kruskala - kod

```
MST( $G, w$ )
{ 1 }   A :=  $\emptyset$ 
{ 2 }   for każdego wierzchołka  $u \in V[G]$ 
{ 3 }     do MakeSet( $v$ )
{ 4 }   Posortuj krawędzie ze zbioru  $E$  w porządku niemalejącym wg wagi  $w$ 
{ 5 }   for każda krawędź  $(u, v) \in E$ , pobranej w kolejności posortowanej
{ 6 }     do if FindSet( $u$ ) ≠ FindSet( $v$ )
{ 7 }       then  $A := A \cup \{(u, v)\}$ 
{ 8 }       Union( $u, v$ )
{ 9 }   return  $A$ 
```

Złożoność (lista sąsiedztwa): $O(|E| \log |E|)$

Najkrótsze ścieżki z jednym źródłem

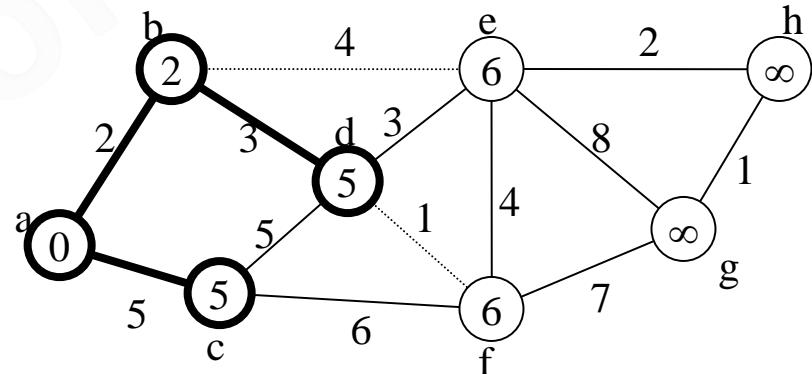
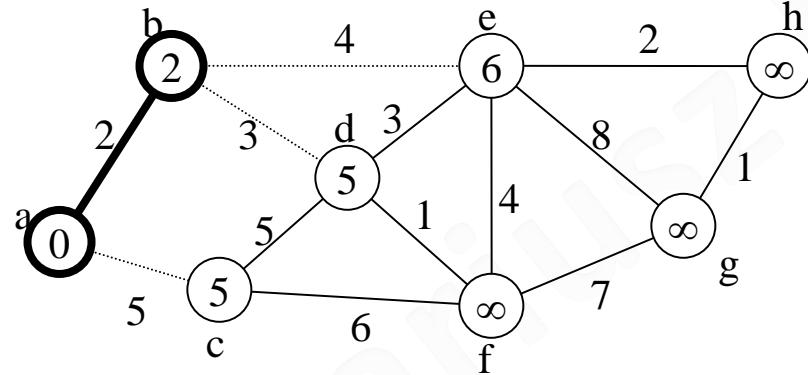
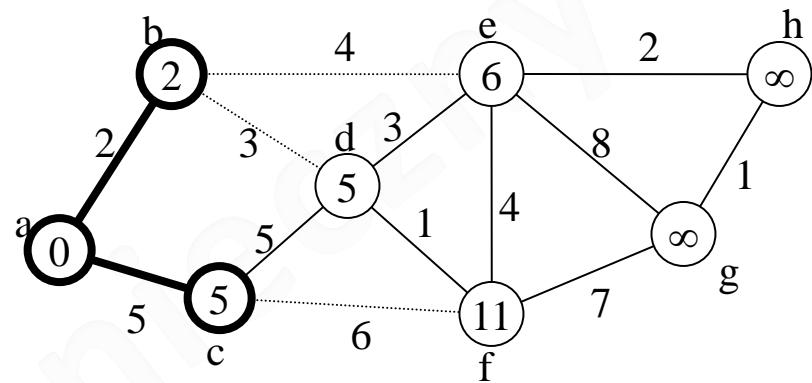
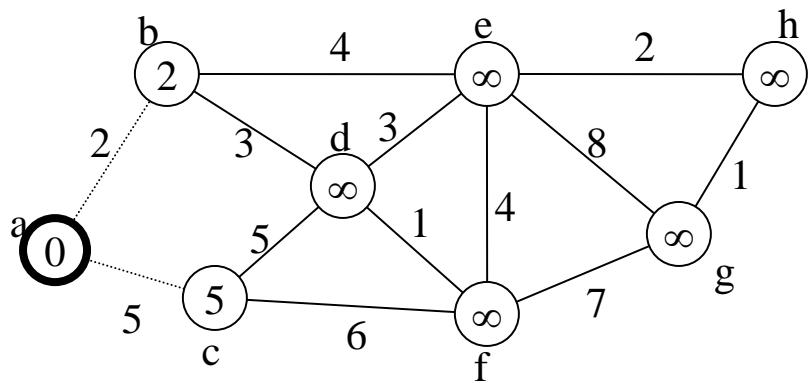
- Problem **najkrótszej ścieżki** to problem znalezienia najkrótszej ścieżki między dwoma wierzchołkami, czyli aby waga takiej ścieżki była jak najmniejsza.
- **Najkrótsze ścieżki z jednego źródła** (ang. *single source shortest paths*) jest bardziej ogólnym problemem, w którym musimy znaleźć najkrótsze ścieżki z podanego wierzchołka źródłowego do wszystkich innych wierzchołków w grafie

```
1. procedure Dijkstra_Single_Source_SP( $V, E, w, s$ )
2. begin
3.    $V_T := \{s\}$ ;
4.   for wszystkich  $v \in (V - V_T)$  do
5.     if krawędź( $s, v$ ) istnieje then  $I[v] := w(s, v)$ ;
6.     else  $I[v] := \infty$ ;
7.   while  $V_T \neq V$  do
8.     begin
9.       Znajdź wierzchołek  $u$  taki, że  $I[u] = \min\{ I[v] \mid v \in (V - V_T)\}$ ;
10.       $V_T := V_T \cup \{u\}$ ;
11.      for wszystkich  $v \in (V - V_T)$  do
12.         $I[v] = \min\{ I[v], I[u] + w(u, v) \}$ ;
13.      endwhile
14.    end Dijkstra_Single_Source_SP
```

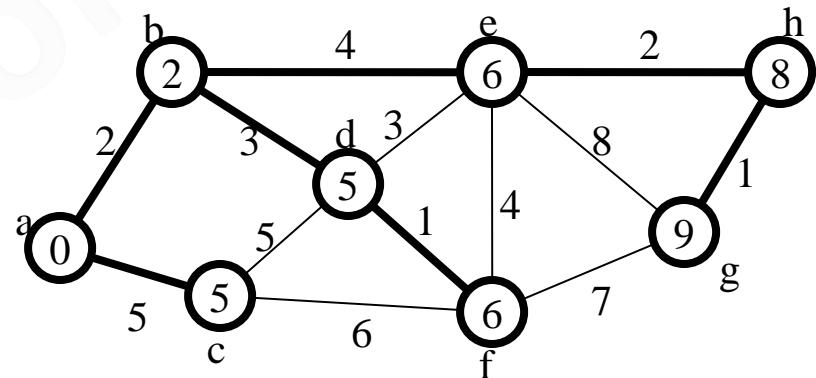
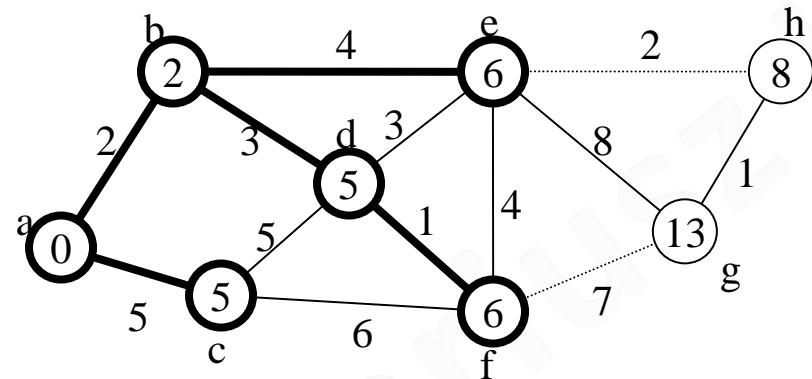
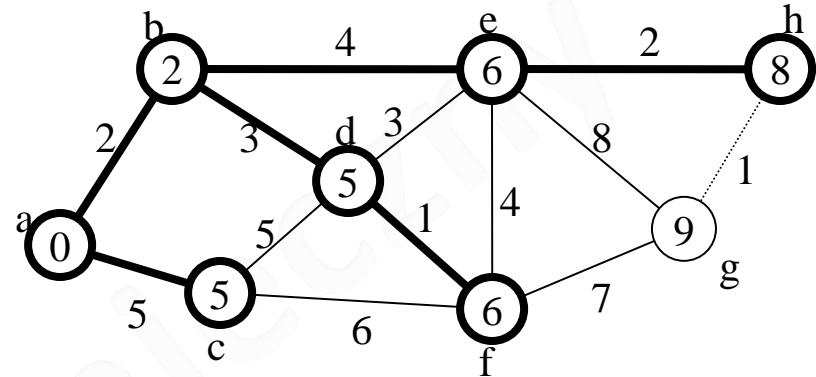
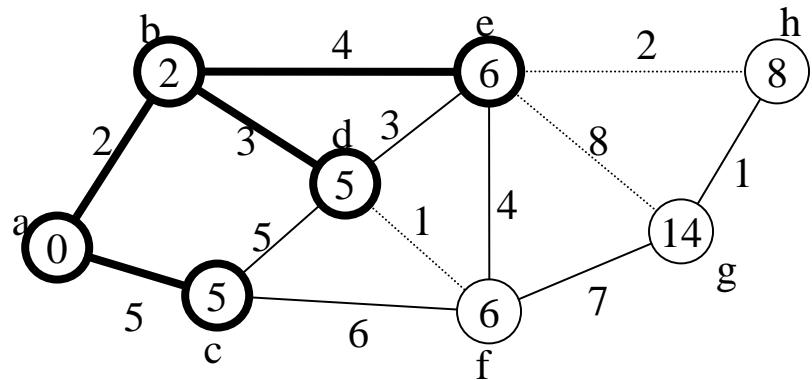


Algorytm
zachłanny

SSSP – przykład 1/2



SSSP – przykład 2/2



Złożoność (kopiec binarny): $O(V \log V + E \log V)$

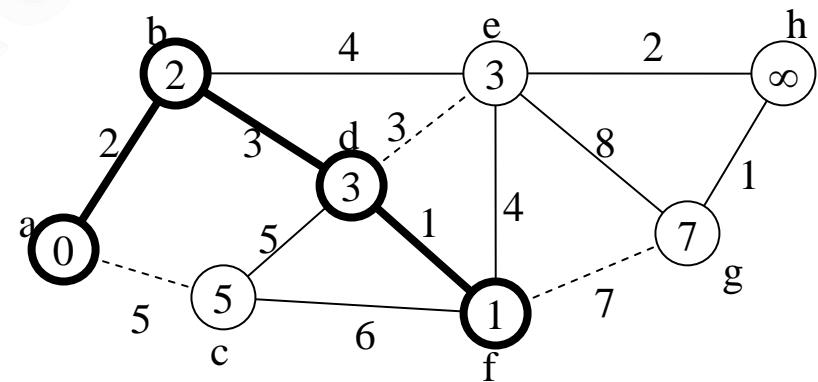
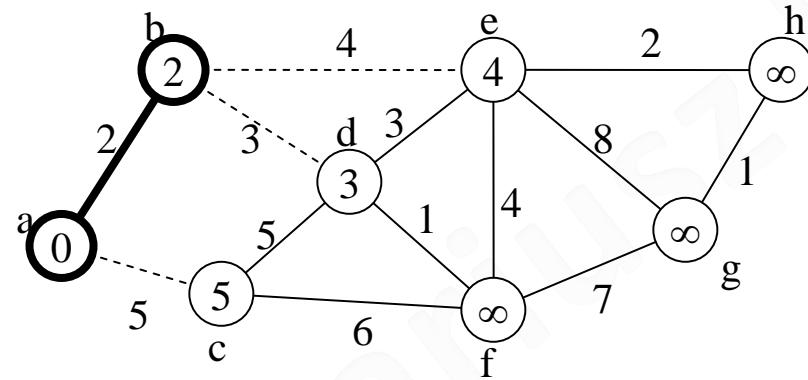
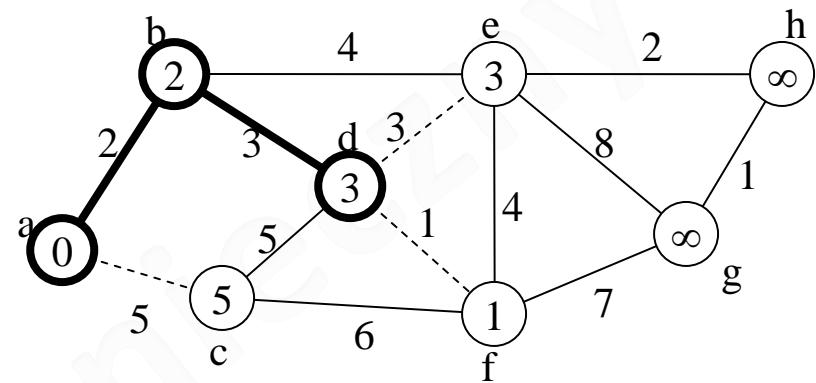
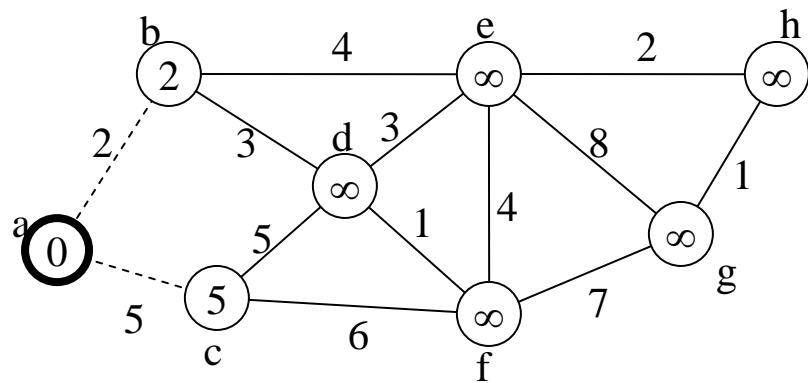
Złożoność (kopiec Fibonacciego): $O(V \log V + E)$

MST – algorytm Prima

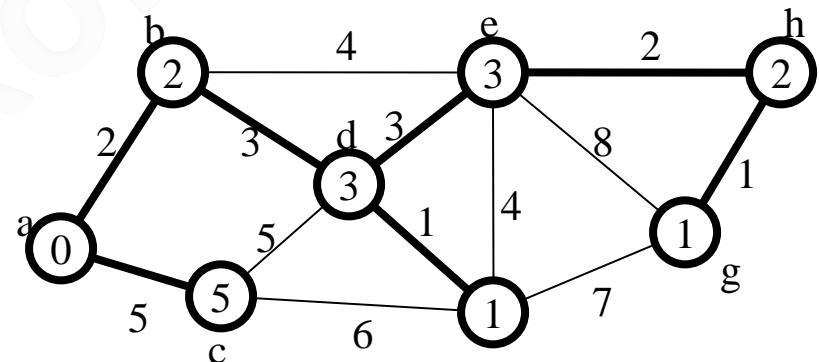
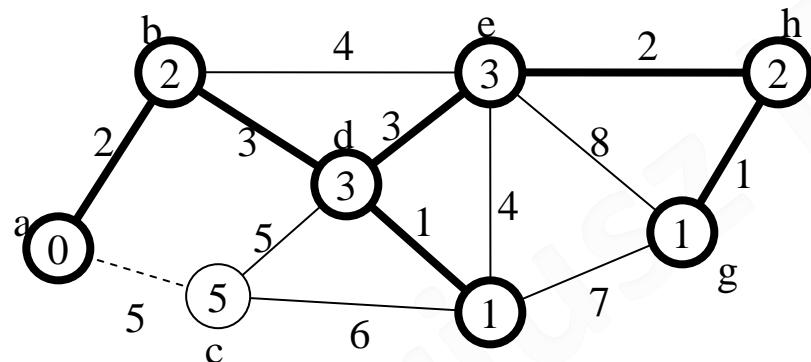
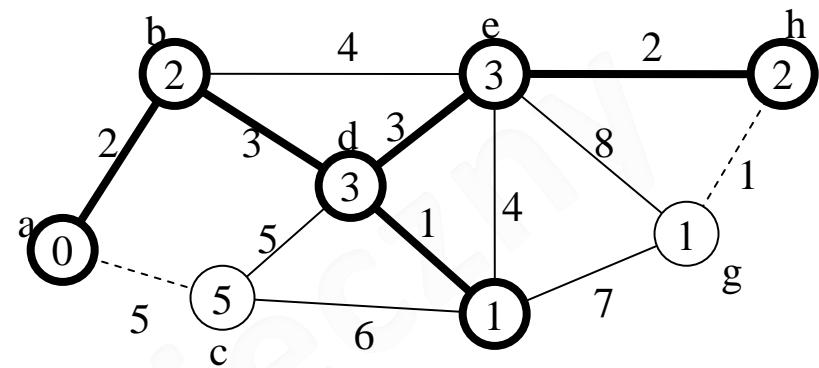
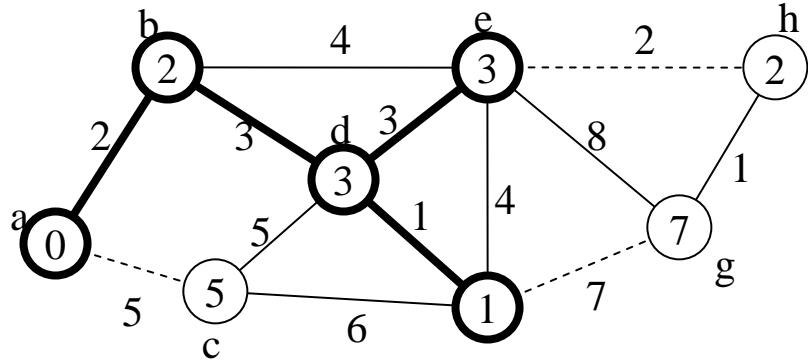
- Idea wzięta z Dijkstra_Single_Source_SP
- Różnica jest tylko w linii 12, w której ten algorytm bierze pod uwagę tylko wagę krawędzi (a nie sumę z wartością pamiętaną dla wierzchołka)

```
1. procedure Prim_MST( $V, E, w, s$ )
2. begin
3.    $V_T := \{s\}$ ;
4.   for wszystkich  $v \in (V - V_T)$  do
5.     If krawędź( $s, v$ ) istnieje then  $d[v] := w(s, v)$ ;
6.     else  $d[v] := \infty$ ;
7.   while  $V_T \neq V$  do
8.     begin
9.       znajdź wierzchołek  $u$  taki, że  $d[u] = \min\{ d[v] \mid v \in (V - V_T) \}$ ;
10.       $V_T := V_T \cup \{u\}$ ;
11.      for każdego  $v \in (V - V_T)$  do
12.         $d[v] = \min\{ d[v], w(u, v) \}$ ;
13.      endwhile
14.    end Prim_MST
```

Alg. Prima – przykład 1/2



Alg. Prima – przykład 2/2



Złożoność (kopiec dwumianowy): $O(|V| \log |V| + |E| \log |V|)$

Złożoność (kopiec Fibonacciego): $O(|V| \log |V| + |E|)$

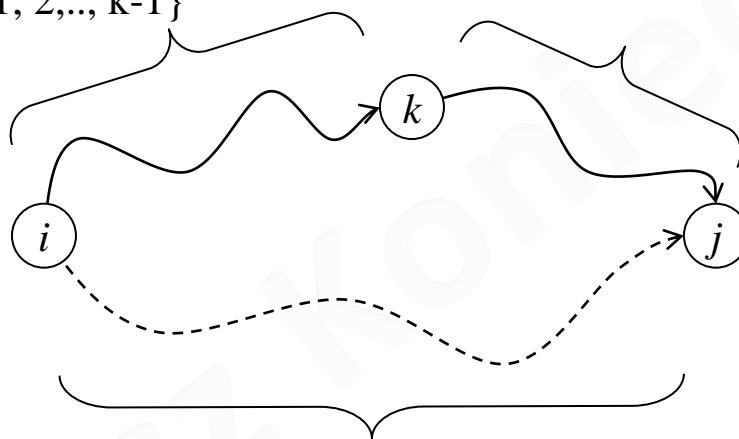
Najkrótsze ścieżki między **wszystkimi** parami wierzchołków

- Wykonać algorytm najkrótszych ścieżek z jednego źródła dla każdego wierzchołka
 - Złożoność algorytmu Dijkstry dla najkrótszych ścieżek z jednego źródła: $O(|V| \log |V| + |E|)$
 - Złożoność dla $|V|$ wierzchołków:
 $O(|V|^2 \log |V| + |V| * |E|)$
- Jeśli mamy graf z ujemnymi wagami (ale bez ujemnych cykli):
 - Można użyć algorytmu Bellmana-Forda dla najkrótszych ścieżek z jednego źródła. Złożoność: $O(|V| * |E|)$
 - Złożoność dla $|V|$ wierzchołków : $O(|V|^2 |E|)$

Algorytm Floyda-Warshalla

Niech $p_{ij}^{(k)}$ oznacza najkrótszą ścieżkę od wierzchołka i do wierzchołka j z wierzchołka mi pośrednimi ze zbioru $\{1, 2, \dots, k\}$. Niech $d_{ij}^{(k)}$ oznacza wagę takiej ścieżki.

Wszystkie pośrednie wierzchołki
ze zbioru $\{1, 2, \dots, k-1\}$



Wszystkie pośrednie wierzchołki
ze zbioru $\{1, 2, \dots, k-1\}$

Wszystkie pośrednie wierzchołki
ze zbioru $\{1, 2, \dots, k-1\}$

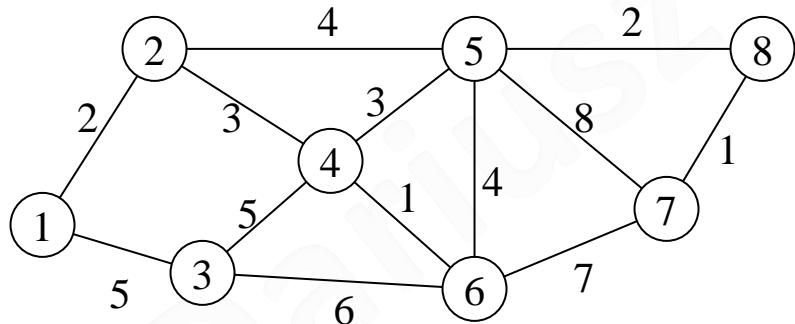
$$d_{ij}^{(k)} = \begin{cases} w_{ij} & \text{dla } k = 0 \\ \min(d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}) & \text{dla } k \geq 1 \end{cases}$$

Programowanie
dynamiczne

Algorytm F.-W. – kod, przykład 1/3

```
1. procedure FLOYD_ALL_PAIRS_SP(A)
2. begin
3.      $D^{(0)} = A;$ 
4.     for  $k = 1$  to  $n$  do
5.         for  $i = 1$  to  $n$  do
6.             for  $j = 1$  to  $n$  do
7.                  $d^{(k)}_{i,j} := \min(d^{(k-1)}_{i,j}, d^{(k-1)}_{i,k} + d^{(k-1)}_{k,j});$ 
8. end FLOYD_ALL_PAIRS_SP
```

Złożoność:
 $O(|V|^3)$



0	2	5	∞	∞	∞	∞	∞
2	0	∞	3	4	∞	∞	∞
5	∞	0	5	∞	6	∞	∞
∞	3	5	0	3	1	∞	∞
∞	4	∞	3	0	4	8	2
∞	∞	6	1	4	0	7	∞
∞	∞	∞	8	7	0	1	1
∞	∞	∞	2	∞	1	0	0

Algorytm F.-W. – przykład 2/3

$$D^{(1)} = \left\{ \begin{array}{|c|c|cccccc|} \hline 0 & 2 & 5 & \infty & \infty & \infty & \infty & \infty \\ \hline 2 & 0 & 7 & 3 & 4 & \infty & \infty & \infty \\ \hline 5 & 7 & 0 & 5 & \infty & 6 & \infty & \infty \\ \hline \infty & 3 & 5 & 0 & 3 & 1 & \infty & \infty \\ \hline \infty & 4 & \infty & 3 & 0 & 4 & 8 & 2 \\ \hline \infty & \infty & 6 & 1 & 4 & 0 & 7 & \infty \\ \hline \infty & \infty & \infty & \infty & 8 & 7 & 0 & 1 \\ \hline \infty & \infty & \infty & \infty & 2 & \infty & 1 & 0 \\ \hline \end{array} \right\}$$

$$D^{(2)} = \left\{ \begin{array}{|c|c|cccccc|} \hline 0 & 2 & 5 & 5 & 6 & \infty & \infty & \infty \\ \hline 2 & 0 & 7 & 3 & 4 & \infty & \infty & \infty \\ \hline 5 & 7 & 0 & 5 & 11 & 6 & \infty & \infty \\ \hline 5 & 3 & 5 & 0 & 3 & 1 & \infty & \infty \\ \hline 6 & 4 & 11 & 3 & 0 & 4 & 8 & 2 \\ \hline \infty & \infty & 6 & 1 & 4 & 0 & 7 & \infty \\ \hline \infty & \infty & \infty & \infty & 8 & 7 & 0 & 1 \\ \hline \infty & \infty & \infty & \infty & 2 & \infty & 1 & 0 \\ \hline \end{array} \right\}$$

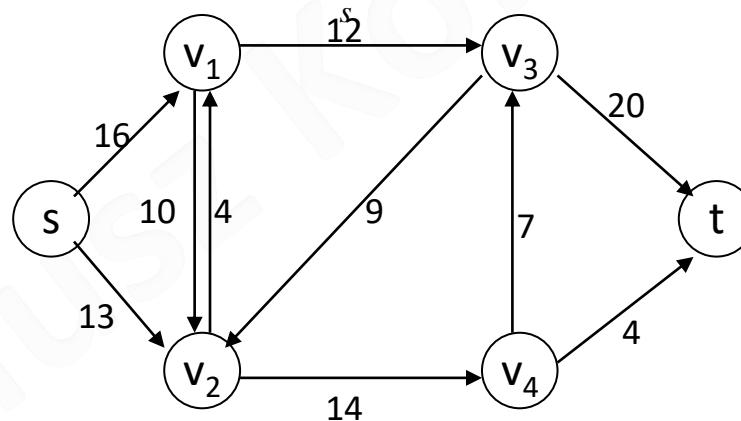
$$D^{(3)} = \left\{ \begin{array}{cccccccc} - & - & - & - & - & - & - & - \\ - & - & - & - & - & - & - & - \\ - & - & - & - & - & - & - & - \\ - & - & - & - & - & - & - & - \\ - & - & - & - & - & - & - & - \\ - & - & - & - & - & - & - & - \\ - & - & - & - & - & - & - & - \\ - & - & - & - & - & - & - & - \end{array} \right\}$$

$$D^{(4)} = \left\{ \begin{array}{cccccccc} - & - & - & - & - & - & - & - \\ - & - & - & - & - & - & - & - \\ - & - & - & - & - & - & - & - \\ - & - & - & - & - & - & - & - \\ - & - & - & - & - & - & - & - \\ - & - & - & - & - & - & - & - \\ - & - & - & - & - & - & - & - \\ - & - & - & - & - & - & - & - \end{array} \right\}$$

Algorytm F.-W. – przykład 3/3

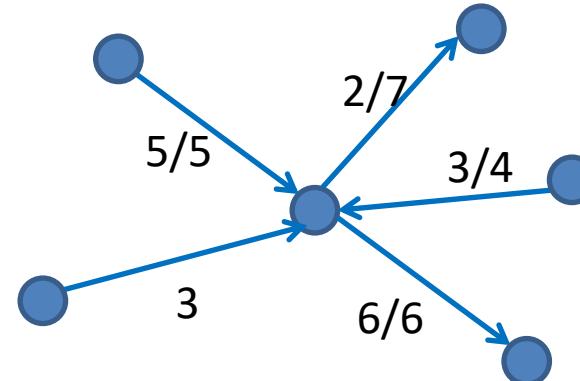
Sieci przepływowne

- **Sieć przepływowa** (ang. *flow network*) $G=(V,E)$ jest skierowanym grafem, w którym każda krawędź $(u,v) \in E$ ma nieujemną przepustowość $c(u,v) \geq 0$. Jeśli $(u,v) \notin E$, to przyjmujemy, że $c(u,v)=0$. W sieci wyróżniamy dwa wierzchołki: **źródło** (ang. *source*) s oraz **ujście** (ang. *sink*) t . Dla wygody przyjmujemy, że każdy wierzchołek leży na pewnej ścieżce ze źródła do ujścia. Oznacza to, że dla każdego wierzchołka $v \in V$, istnieje ścieżka $s \rightsquigarrow v \rightsquigarrow t$. Graf jest zatem spójny, stąd $|E| \geq |V| - 1$

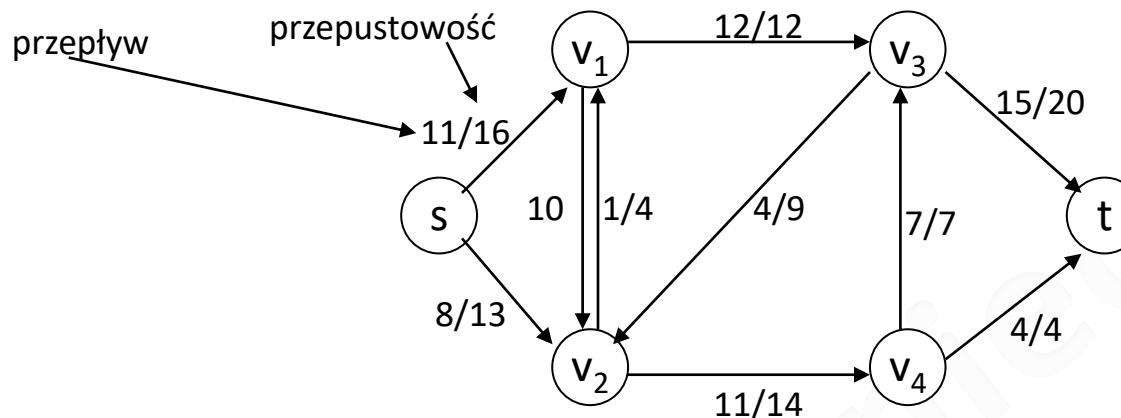


Przepływy

- Niech $G = (V, E)$ będzie siecią przepływową z funkcją **przepustowości** c . Niech s będzie źródłem w sieci, a t ujściem. **Przepływem** w G nazywamy każdą funkcję o wartościach rzeczywistych $f : V \times V \rightarrow \mathbb{R}$ spełniającą poniższe trzy warunki:
 - **Warunek przepustowości:** Dla wszystkich $u, v \in V$, zachodzi $f(u, v) \leq c(u, v)$.
 - **Warunek skośnej symetrii:** Dla wszystkich $u, v \in V$, zachodzi $f(u, v) = -f(v, u)$.
 - **Warunek zachowania przepływu:** Dla wszystkich $u \in V - \{s, t\}$, zachodzi:
$$\sum_{v \in V} f(u, v) = 0$$



Przepływy - przykład



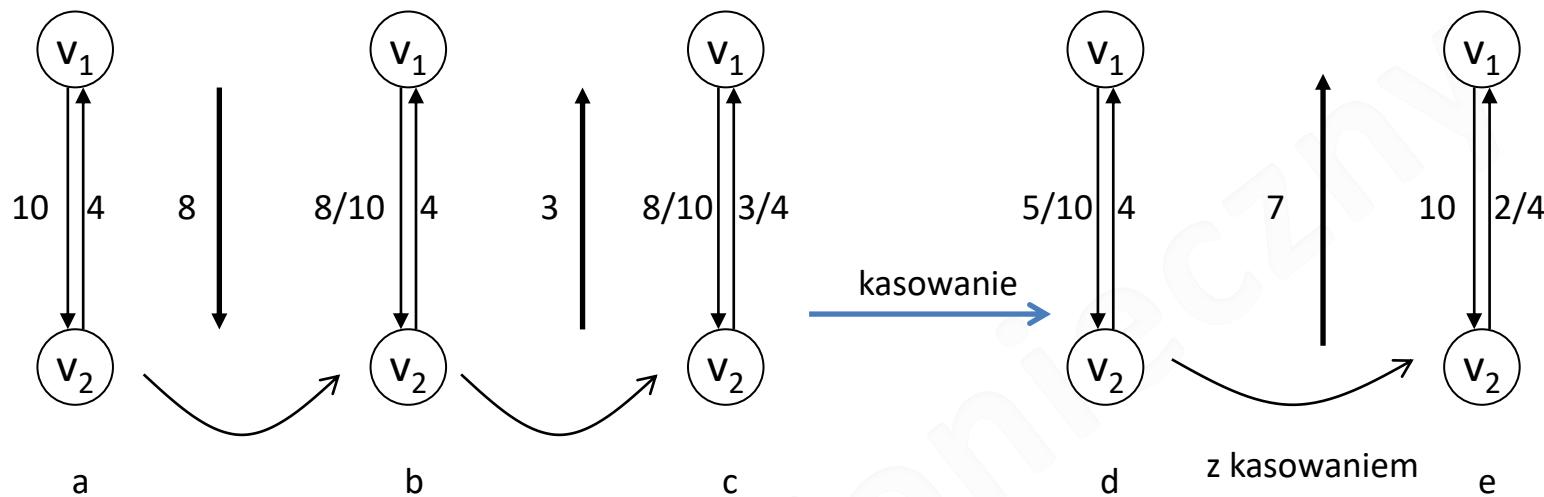
- Wielkość $f(u, v)$, która może być dodatnia, ujemna lub zero nazywamy **przepływem netto** z wierzchołka u do wierzchołka v . Wartość przepływu f definiujemy jako:

$$|f| = \sum_{v \in V} f(s, v)$$

Oznacza to, że wartością przepływu jest łączny przepływ netto opuszczający źródło.
(Tutaj, zapis $|\cdot|$ oznacza wartość przepływu, a nie wartość bezwzględną lub moc zbioru)

W problemie maksymalnego przepływu dana jest sieć G oraz źródło s oraz ujście t . Należy znaleźć przepływ o maksymalnej wartości z s do t .

Kasowanie

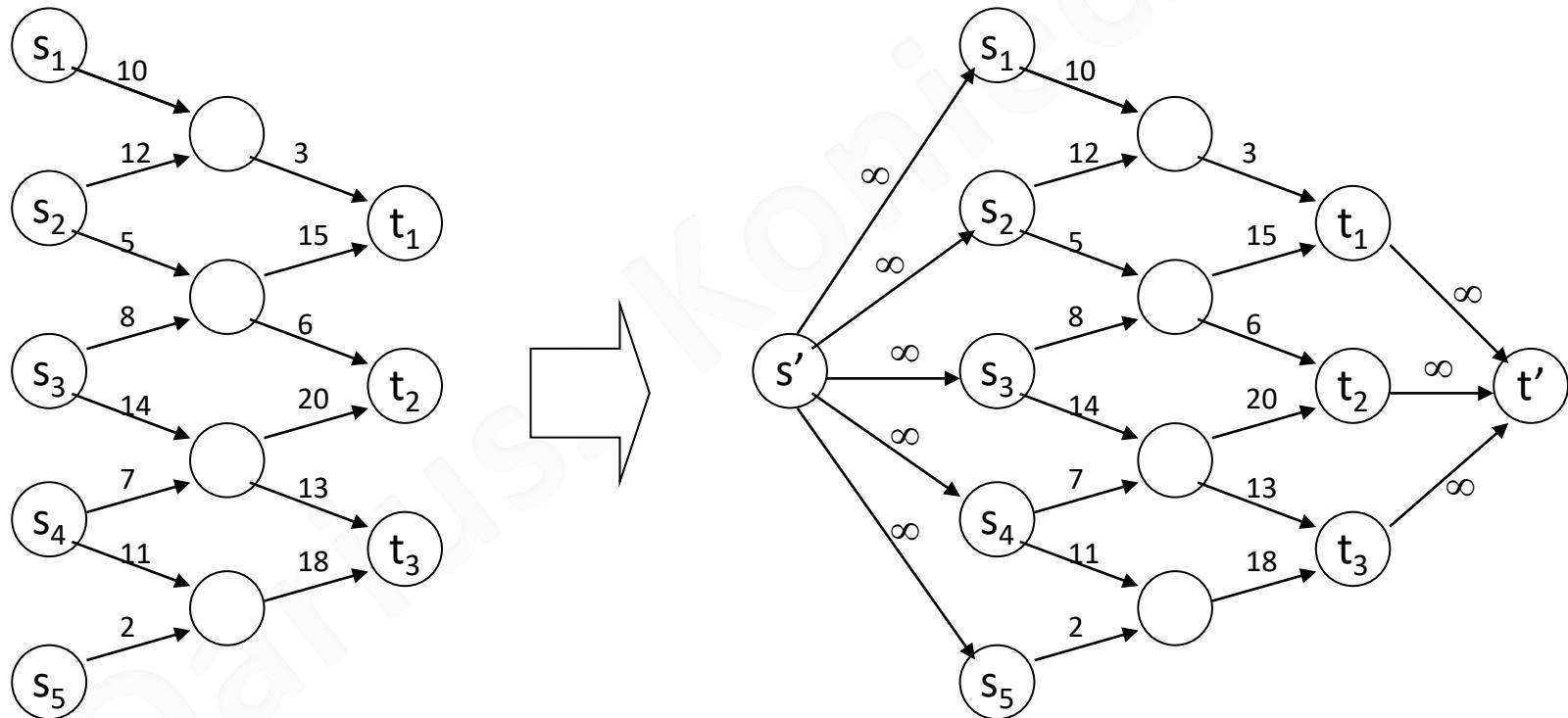


- a) stan początkowy z przepustowościami
- b) Początkowo wysyłamy z v_1 do v_2 8 kontenerów na dzień (k/dz).
- c) Po nowym zamówieniu musimy przesłać z v_2 do v_1 3 k/dz.
- d) Kasujemy przepływu w przeciwnych kierunkach rzędu 3 k/dz.
- e) W kolejnym zamówieniu mamy przesłać 7 k/dz z v_1 do v_2 . przepływu już po dokonaniu skasowania.

Operacja kasowania nie narusza żaden z warunków poprawnego przepływu

Sieć z wieloma źródłami i ujściami

- Problem ten możemy zredukować do problemu maksymalnego przepływu dodając **superźródło** i **superujście** połączone odpowiednio ze źródłami i ujściami krawędziami o nieskończonej przepustowości.



Metoda Forda-Fulkersona

- Metoda Ford-Fulkerson jest metodą iteracyjną.
- Startujemy z $f(u, v) = 0$ dla wszystkich $u, v \in V$, czyli początkowa wartość przepływu wynosi 0
- W każdej iteracji zwiększamy przepływ „ścieżką powiększającą”. Ścieżka powiększająca to taka ścieżka od źródła s do ujścia t , po której możemy przesłać większy przepływ. Powtarzamy ten proces dopóty, dopóki nie można już znaleźć ścieżki powiększającej.

```
FORD-FULKERSON-METHOD ( $G, s, t$ )
{ 1} inicjowanie  $f$  na 0
{ 2} while istnieje ścieżka powiększająca  $p$  do
{ 3}         powiększ przepływ  $f$  wzduż  $p$ 
{ 4} return  $f$ 
```

Sieć residualna

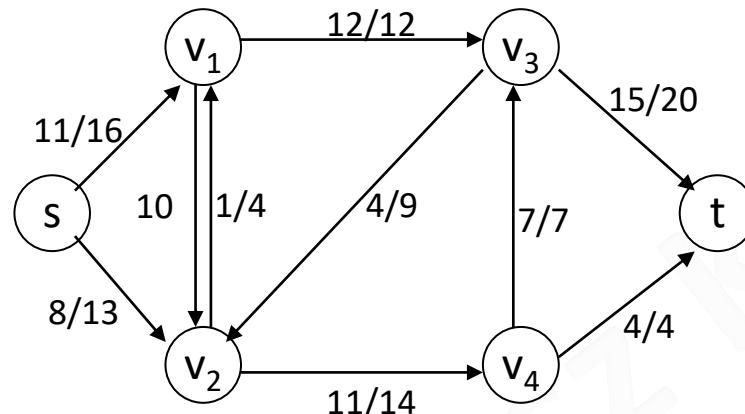
Intuicyjnie , dla danej sieci przepływowej i pewnego przepływu sieć residualna składa się z krawędzi , które dopuszczają większy przepływ netto. Bardziej formalnie założymy, że dana jest sieć przepływowa $G = (V, E)$ ze źródłem s i ujściem t . Niech f będzie pewnym przepływem w G . Rozważmy parę wierzchołków $u, v \in V$. Dodatkowy przepływ z u do v nie przekraczający przepustowości $c(u, v)$ nazywamy **przepustowością residualną** dla (u, v) , którą definiujemy następująco:

$$c_f(u, v) = c(u, v) - f(u, v)$$

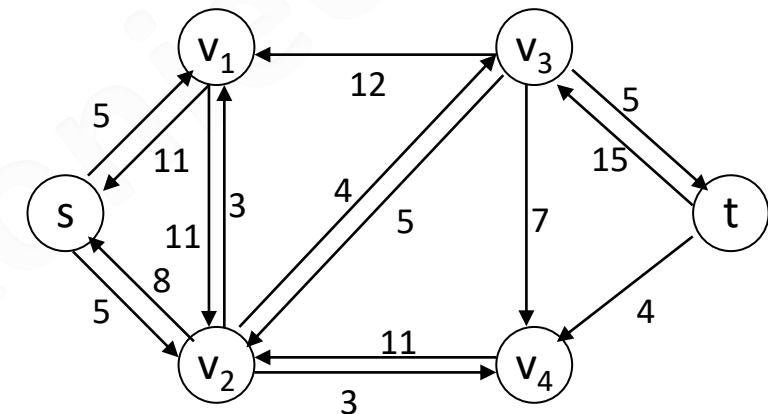
Sieć residualna

Dana jest sieć $G = (V, E)$ oraz przepływ f . Siecią residualną dla sieci G indukowaną przez przepływ f nazywamy sieć $G_f = (V, E_f)$, w której

$$E_f = \{(u, v) \in V \times V : c_f(u, v) > 0\}$$



przepływy

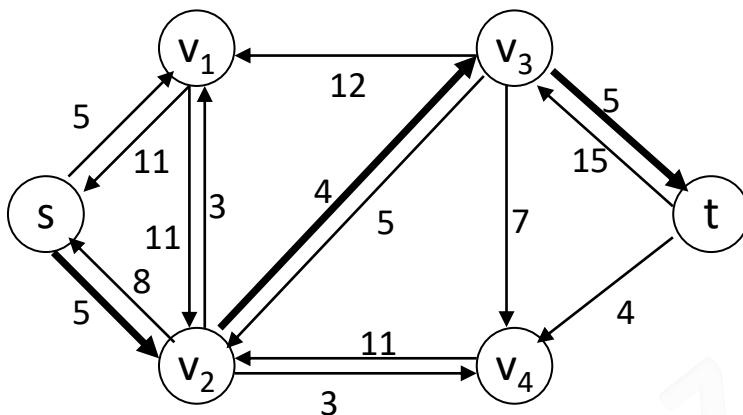


sieć residualna

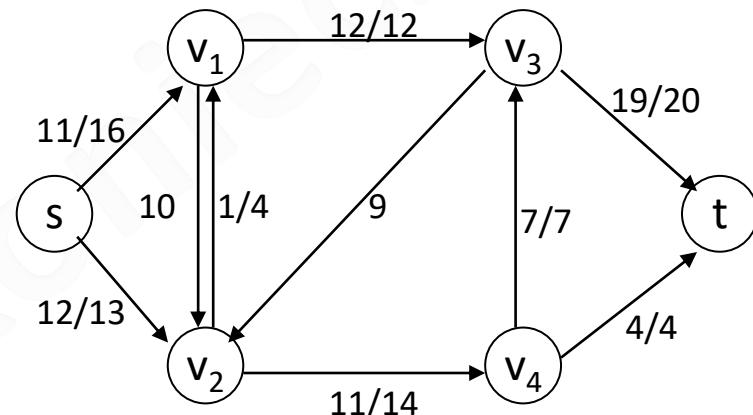
Niech $G = (V, E)$ będzie siecią ze źródłem s i ujściem t , i niech f będzie przepływem w G . Niech G_f będzie siecią residualną sieci G indukowaną przez f , natomiast niech f' będzie przepływem w G_f . Wówczas suma przepływów $f + f'$ zdefiniowana jest przez wartość $|f + f'| = |f| + |f'|$.

Ścieżka powiększająca

- Dla danej sieci $G = (V, E)$ i przepływu f , ścieżką powiększającą p nazywamy każdą ścieżką ze źródła s do ujścia t w sieci residualnej G_f . Zgodnie z definicją sieci residualnej każda krawędź (u, v) na ścieżce powiększającej umożliwia pewien dodatni przepływ netto między wierzchołkami u i v bez naruszenia warunku przepustowości dla tej krawędzi.



Sieć residualna
ze ścieżką powiększającą



Nowy przepływ w sieci powiększony wzdłuż
ścieżki o jej przepustowość residualną

Największy możliwy przepływ netto po krawędziach ścieżki powiększającej p nazywamy jej **przepustowością residualną** i definiujemy następująco:

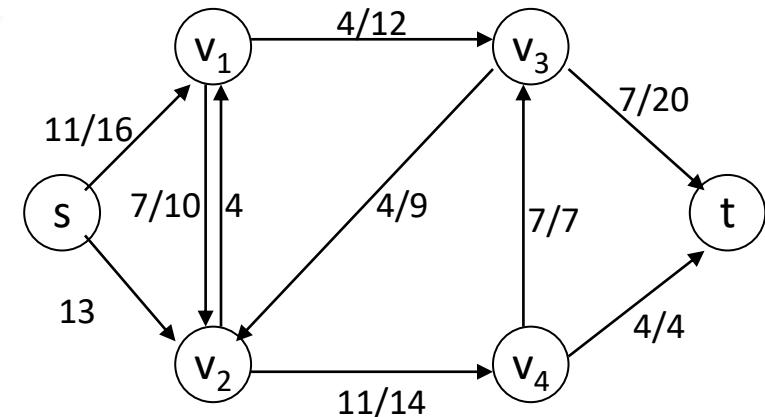
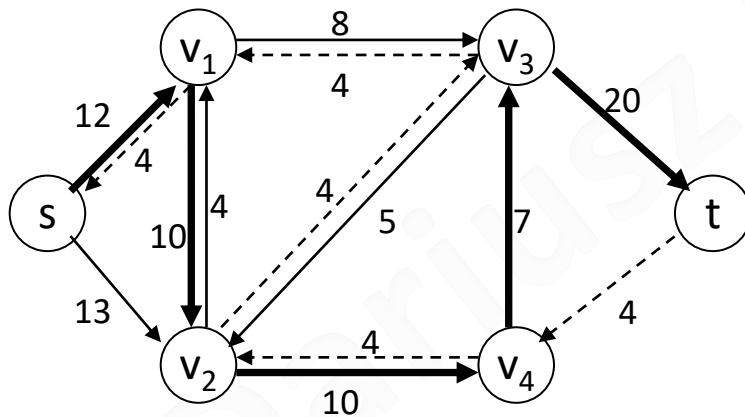
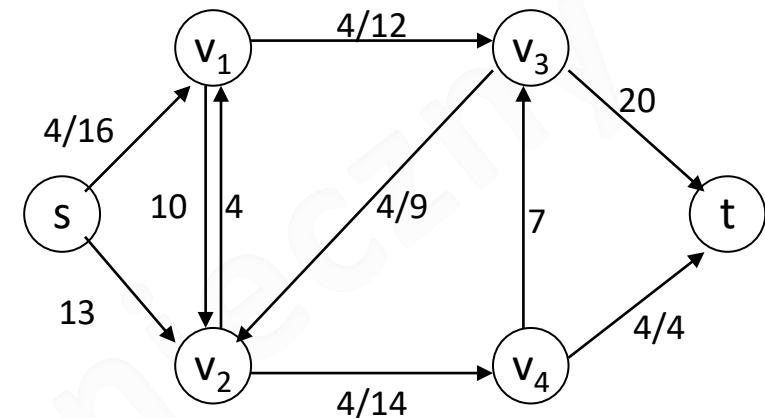
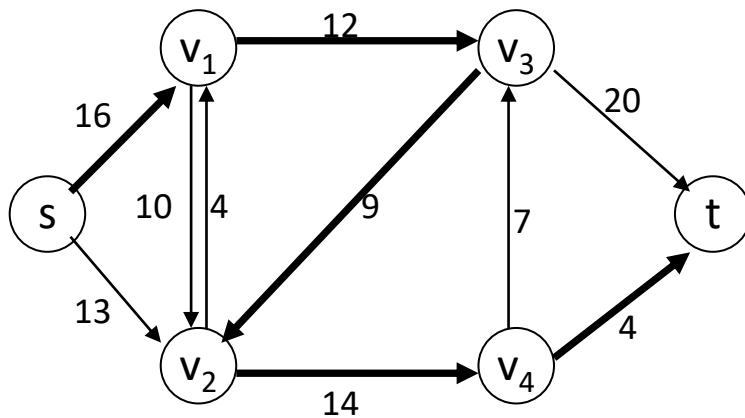
$$c_f(p) = \min \{c_f(u, v) : (u, v) \text{ jest na } p\}$$

Podstawowy algorytm Forda-Fulkersona

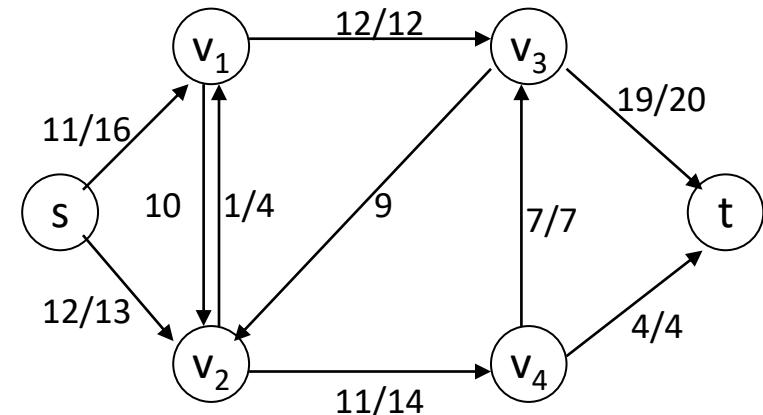
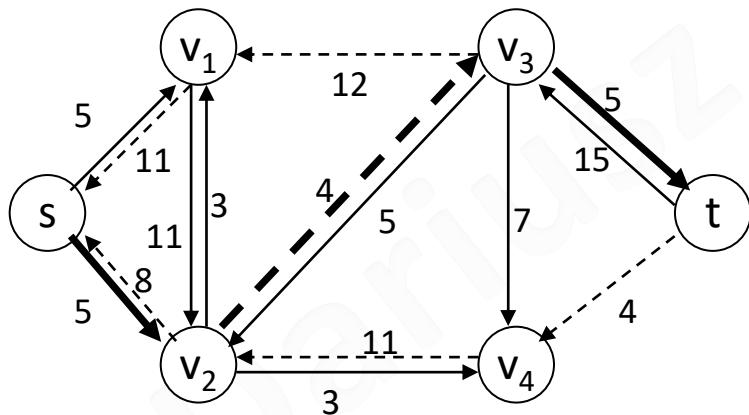
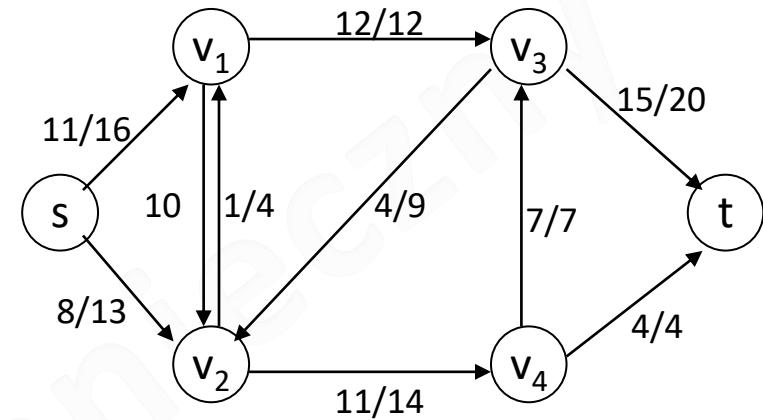
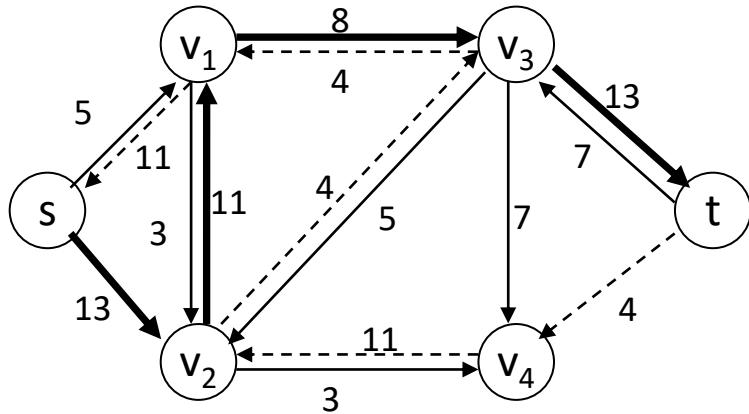
- W metodzie Forda-Fulkersona w każdej iteracji znajdujemy dowolną ścieżkę powiększającą p i zwiększamy przepływ f wzdłuż p o przepustowość residualną $c_f(p)$.
- Metoda wyboru ścieżki – dowolna:
 - Dla przepustowości wyrażonych liczbami rzeczywistymi można stworzyć sieć, gdzie ciągle będzie zwiększany przepływ, ale nigdy nie osiągniemy maksymalnego przepływu
 - Dla przepustowości wyrażonych liczbami całkowitymi zawsze znajdzie się maksymalny przepływ, ale może być to czasochłonne (patrz dalej)

```
FORD-FULKERSON ( $G, s, t$ )
{ 1} for każda krawędź  $(u, v) \in E[G]$  do
{ 2}    $f[u, v] := 0$ 
{ 3}    $f[v, u] := 0$ 
{ 4} while istnieje ścieżka  $p$  z  $s$  do  $t$  w sieci residualnej  $G_f$  do
{ 5}    $c_f(p) := \min \{c_f(u, v) : (u, v) \text{ is in } p\}$ 
{ 6}   for każda krawędź  $(u, v)$  w  $p$  do
{ 7}      $f[u, v] := f[u, v] + c_f(p)$ 
{ 8}      $f[v, u] := -f[u, v]$ 
```

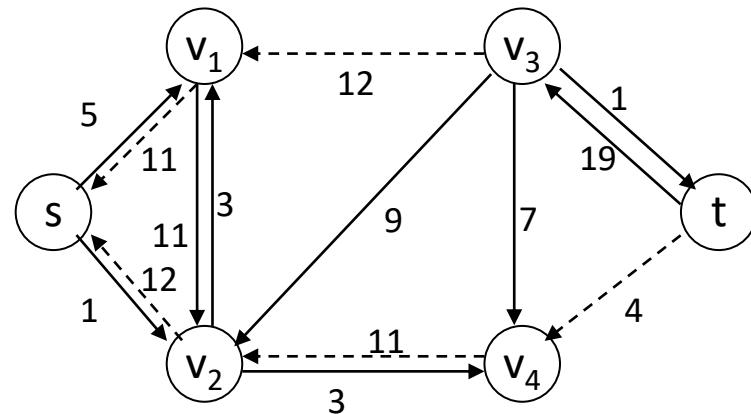
Przykład 1/2



Przykład 2/2

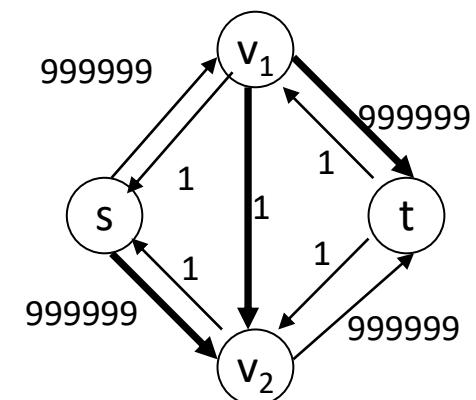
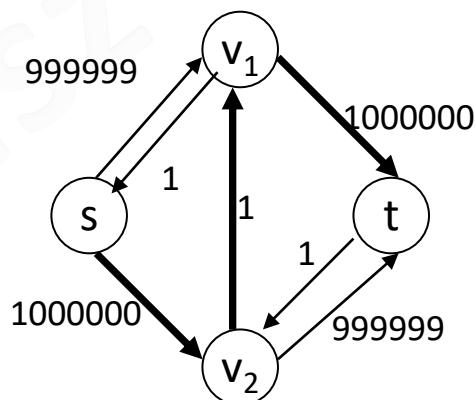
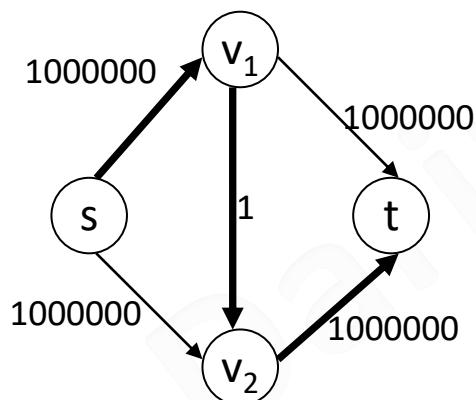


Analiza



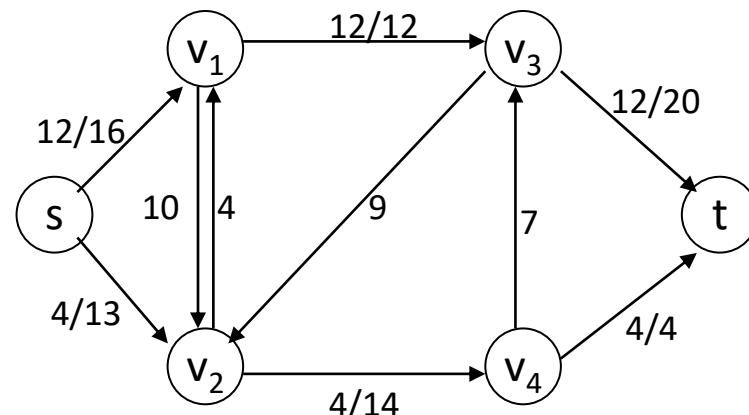
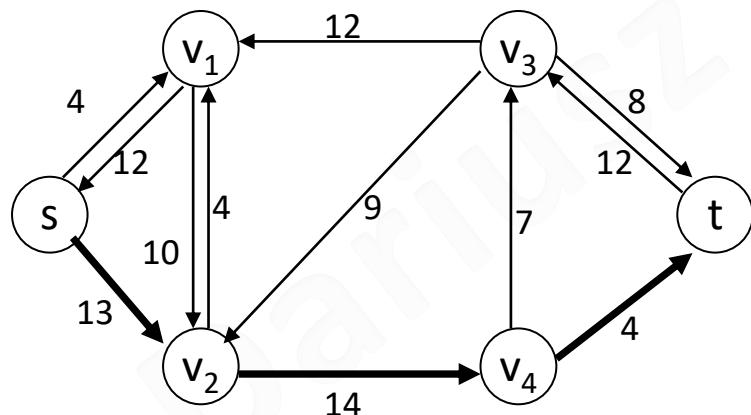
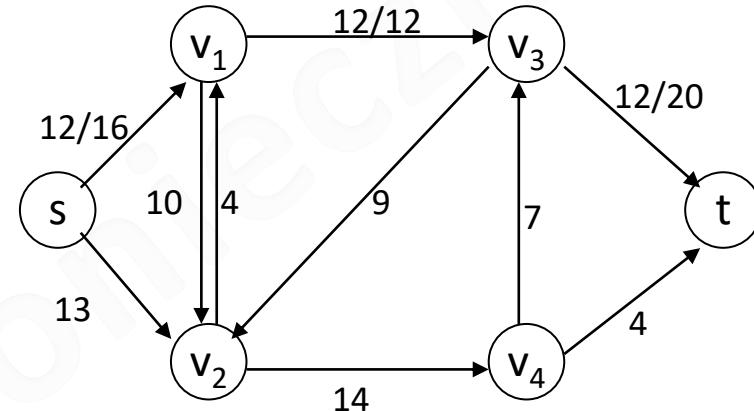
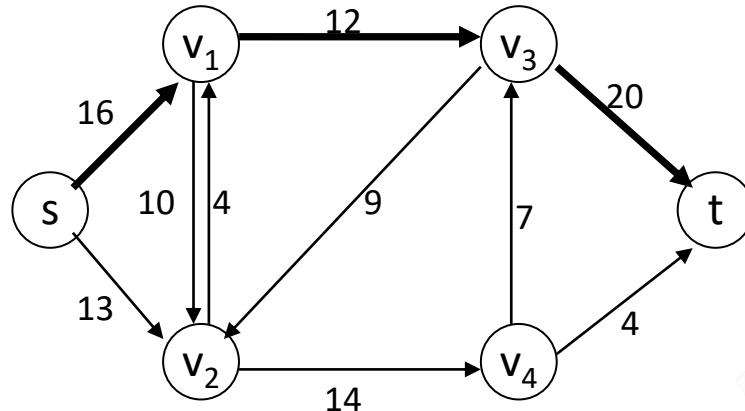
- złożoność:
 - $O(|f|V)$

Wartość przepływu

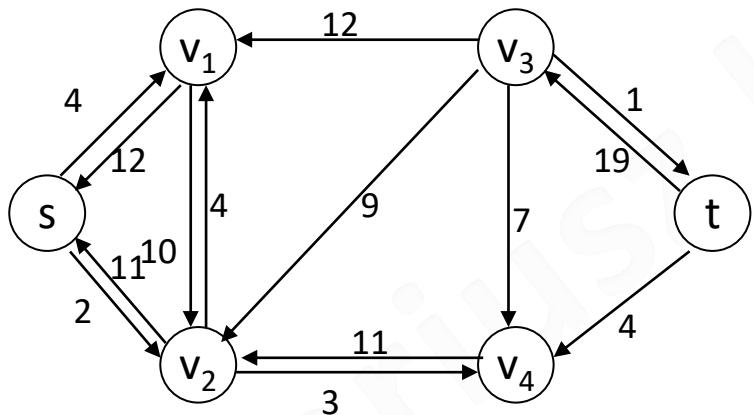
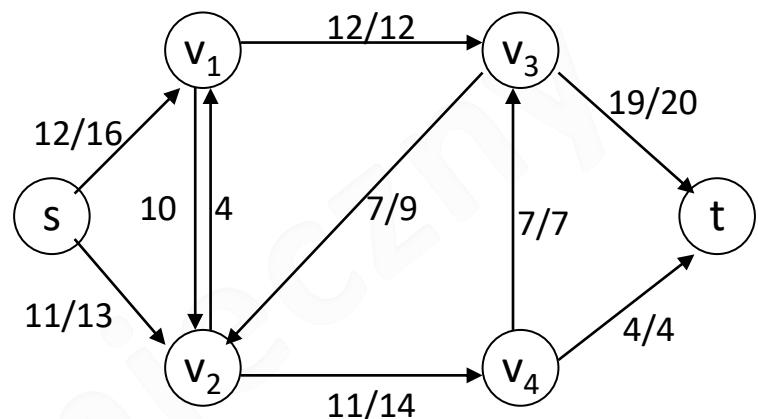
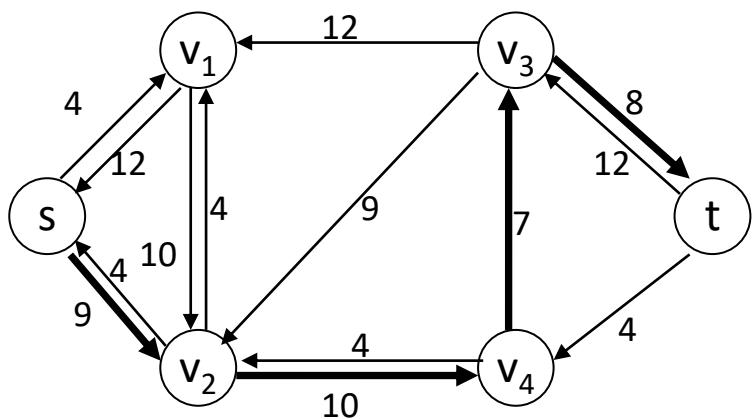


Algorytm Edmonda-Karpa, przykład 1/2

- Zamiast szukać dowolnej ścieżki, szukamy ścieżki o najmniejszej długości z użyciem przeglądania grafu wszerz.



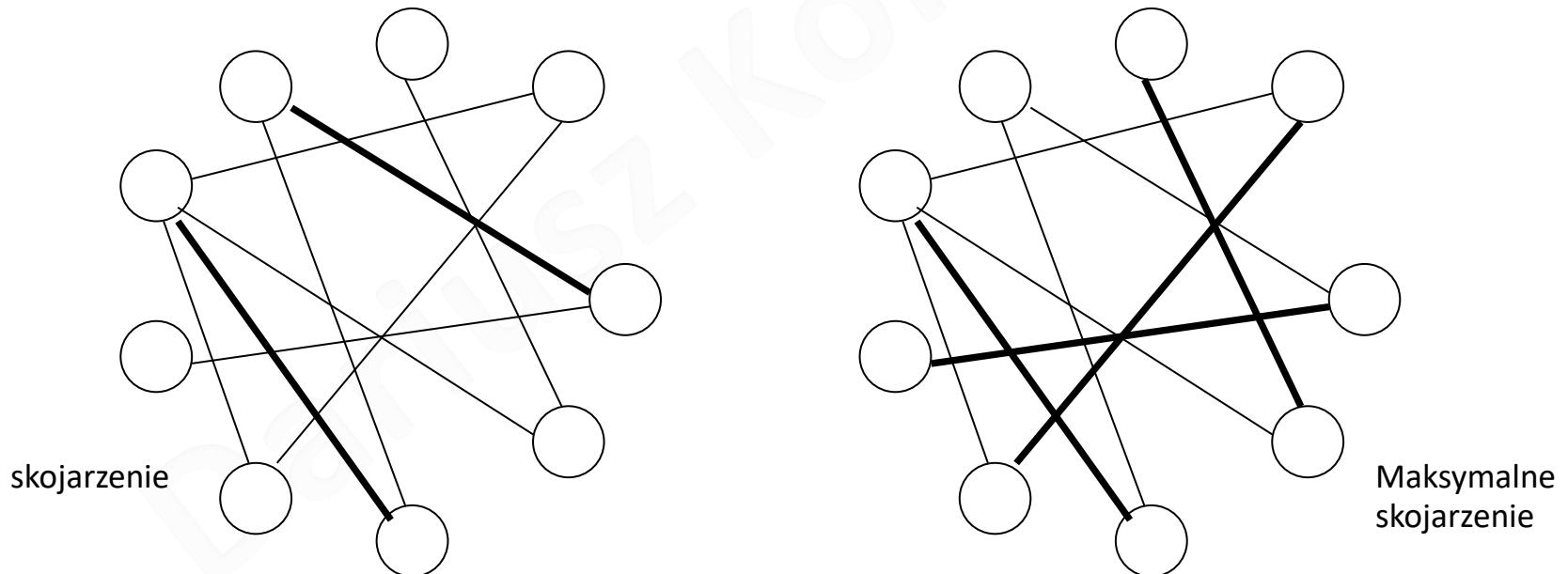
Przykład – 2/2, analiza



- złożoność:
 - $O(VE^2)$

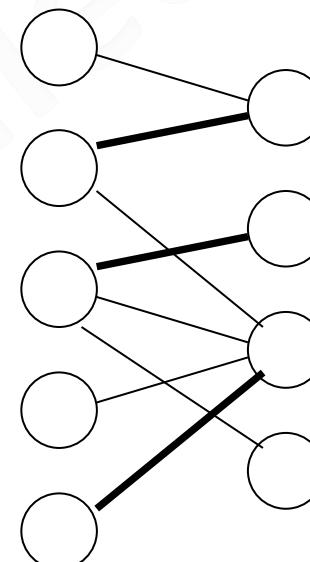
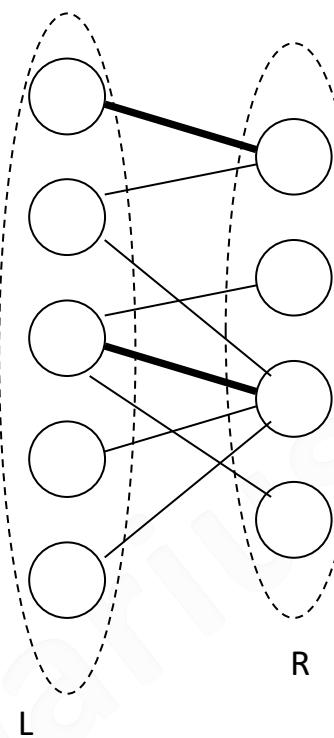
Skojarzenie w grafie

- Niech $G = (V, E)$ będzie grafem nieskierowanym. **Skojarzeniem** w grafie G nazywamy każdy podzbiór krawędzi $M \subseteq E$ taki, że dla wszystkich wierzchołków $v \in V$, co najwyżej jedna krawędź z podzbioru M jest incydentna z v . Mówimy, że wierzchołek $v \in V$ jest **skojarzony** w podzbiorze M jeśli pewna krawędź z M jest incydentna z wierzchołkiem v ; w przeciwnym razie mówimy, że wierzchołek v jest wolny. **Maksymalnym skojarzeniem** nazywamy każde skojarzenie o maksymalnej liczności, czyli takie skojarzenie M , że dla każdego innego skojarzenia M' , mamy $|M| \geq |M'|$



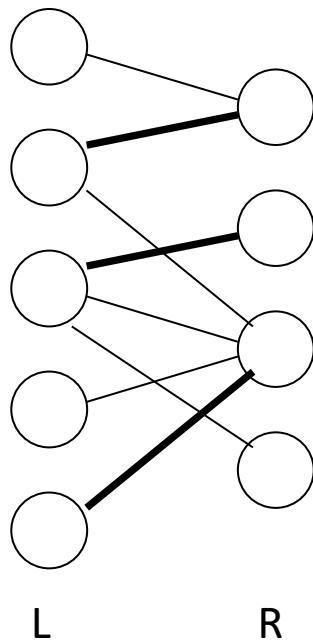
Maks. skojarzenie w gr. dwudzielnych

- Założymy, że zbiór wierzchołków można podzielić na $V = L \cup R$, gdzie L i R są rozłączne, a wszystkie krawędzie z E prowadzą miedzy L a R .



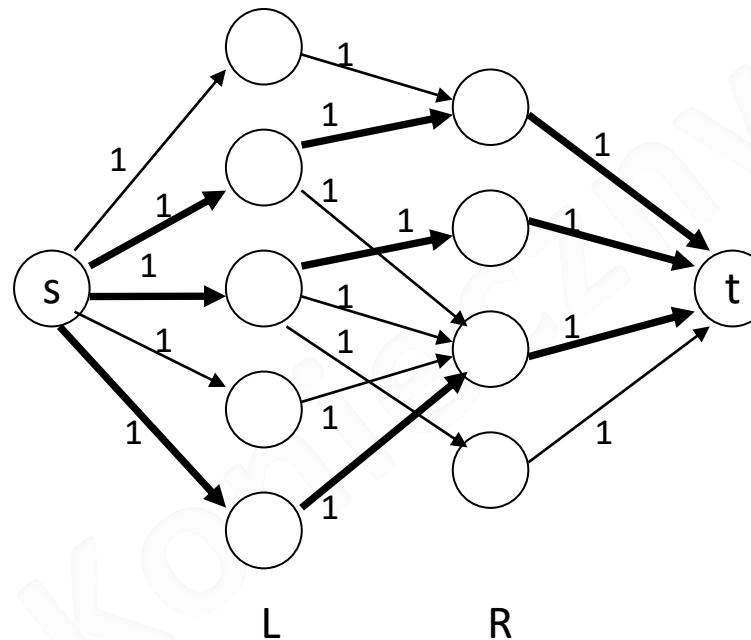
Maksymalne skojarzenie

Odpowiadająca sieć przepływową



L

R



L

R

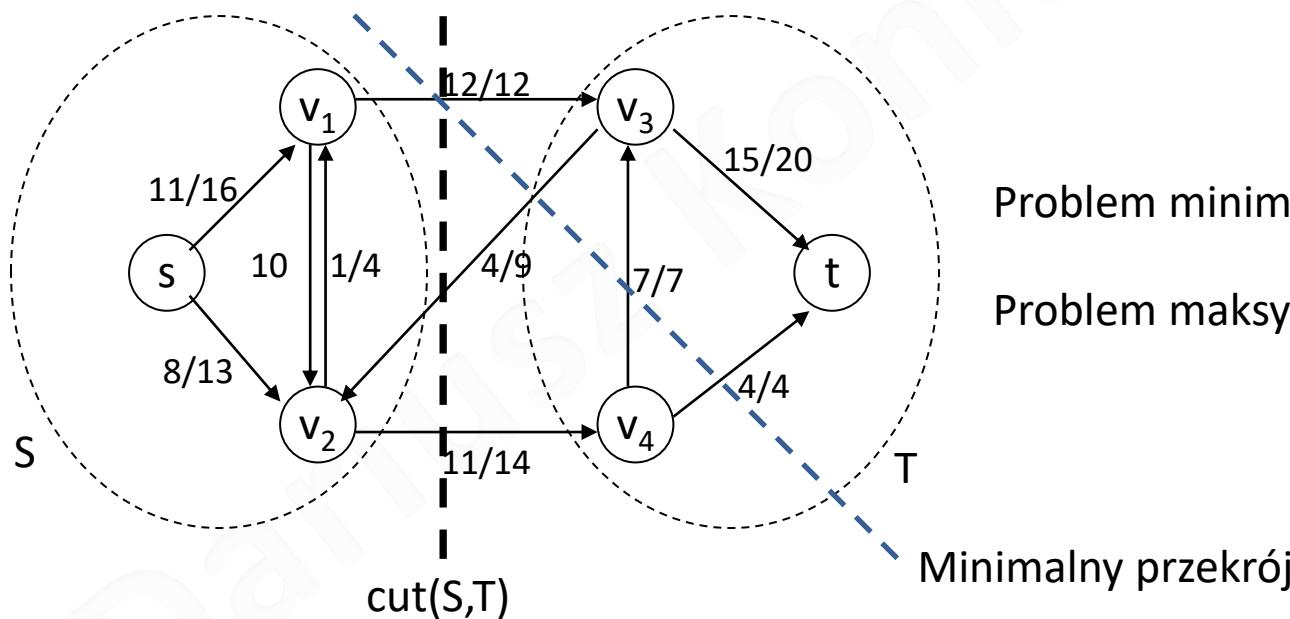
- Dla dwudzielnego grafu G konstrujemy odpowiadającą mu sieć przepływową $G' = (V', E')$ w następujący sposób. Niech źródło s i ujście t będzie nowymi wierzchołkami nie należącymi do V , i niech $V' = V \cup \{s, t\}$. Jeśli $V = L \cup R$ jest podziałem zbioru wierzchołków G , to zbiorem krawędzi skierowanych w sieci G' jest:

$$E' = \{(s, u) : u \in L\} \cup \{(u, v) : u \in L \wedge v \in R \wedge (u, v) \in E\} \cup \{(v, t) : v \in R\}$$

Złożoność: O(VE)

Przekroje w sieciach

- **Przekrojem** (S, T) sieci $G = (V, E)$ jest podział V na zbiory S i $T = V - S$ takie, że $s \in S$ oraz $t \in T$. Jeśli f jest przepływem, to **przepływ netto przez przekrój** (S, T) definiujemy jako $f(S, T)$. **Przepustowośćą przekroju** (S, T) jest $c(S, T)$. Minimalnym przekrojem jest przekrój, którego przepustowość jest najmniejsza z wszystkich możliwych w tej sieci.



$$f(S, T) = f(\{s, v_1, v_2\}, \{v_3, v_4, t\}) = 12 - 4 + 11 = 19$$

$$c(S, T) = 12 + 14 = 26$$

Problem minimalnego przekroju
||
Problem maksymalnego przepływu

Minimalny przekrój

Podsumowanie

- Problemy teorii grafów można rozwiązać na wiele sposob
- Istnieją podobieństwa między algorytmami rozwiązującymi różne problemy
- Ulepszając wewnętrzny krok algorytmu można otrzymać algorytm o lepszej złożoności:
 - a nawet zapewnić zbieżność algorytmu
- Znając algorytm znajdowania największego przepływu można rozwiązać wiele innych problemów odpowiednio przekształcając go na sieć przepływu.

Algorytmy i struktury danych – W12

Podstawowe techniki rozwiązywania
problemów

Zawartość

- Podstawowe techniki rozwiązywania problemów
 - „Dziel i rządź”
 - Programowanie dynamiczne:
 - Technika zachłanna
 - Inne techniki
- Ciekawe zadania

Techniki

- Podstawowe techniki rozwiązywania problemów:
 - Dziel i rządź
 - Programowanie dynamiczne
 - Algorytmy zachłanne

Dziel i rządź

- Technika „dziel i rządź” (ang. Divide and conquer, łac. Divide et impera): problem dzieli się **rekurencyjnie na dwa lub więcej mniejszych podproblemów tego samego (lub podobnego) typu** tak długo, aż fragmenty staną się wystarczająco proste do bezpośredniego rozwiązymania. Z kolei rozwiązania otrzymane dla podproblemów **scala się**, uzyskując rozwiązanie całego zadania
- Rozwiązanie bezpośrednie stosowane jest, gdy rozmiar problemu osiągnie pewną ustaloną wcześniej wartość progową.
- Podproblemy muszą być niezależne od siebie!

Dziel i rządź – pseudokod

```
procedure Dziel_i_rzadz(I, J)
Input: I (dane wejściowe problemu)
output: J (rozwiązanie problemu dla danych I)
if I Jest_Znany then
    Przyporządkuj znane rozwiązanie do J
else
    Dziel(I, I1, ..., Im) // m może zależeć od wejścia I
    for i=1 to m do
        Dziel_i_rzadz(Ii, Ji)
    endfor
    Polacz(J1, ..., Jm, J)
endif
end Dziel_i_rzadz
```

Dziel i rządź - mergesort

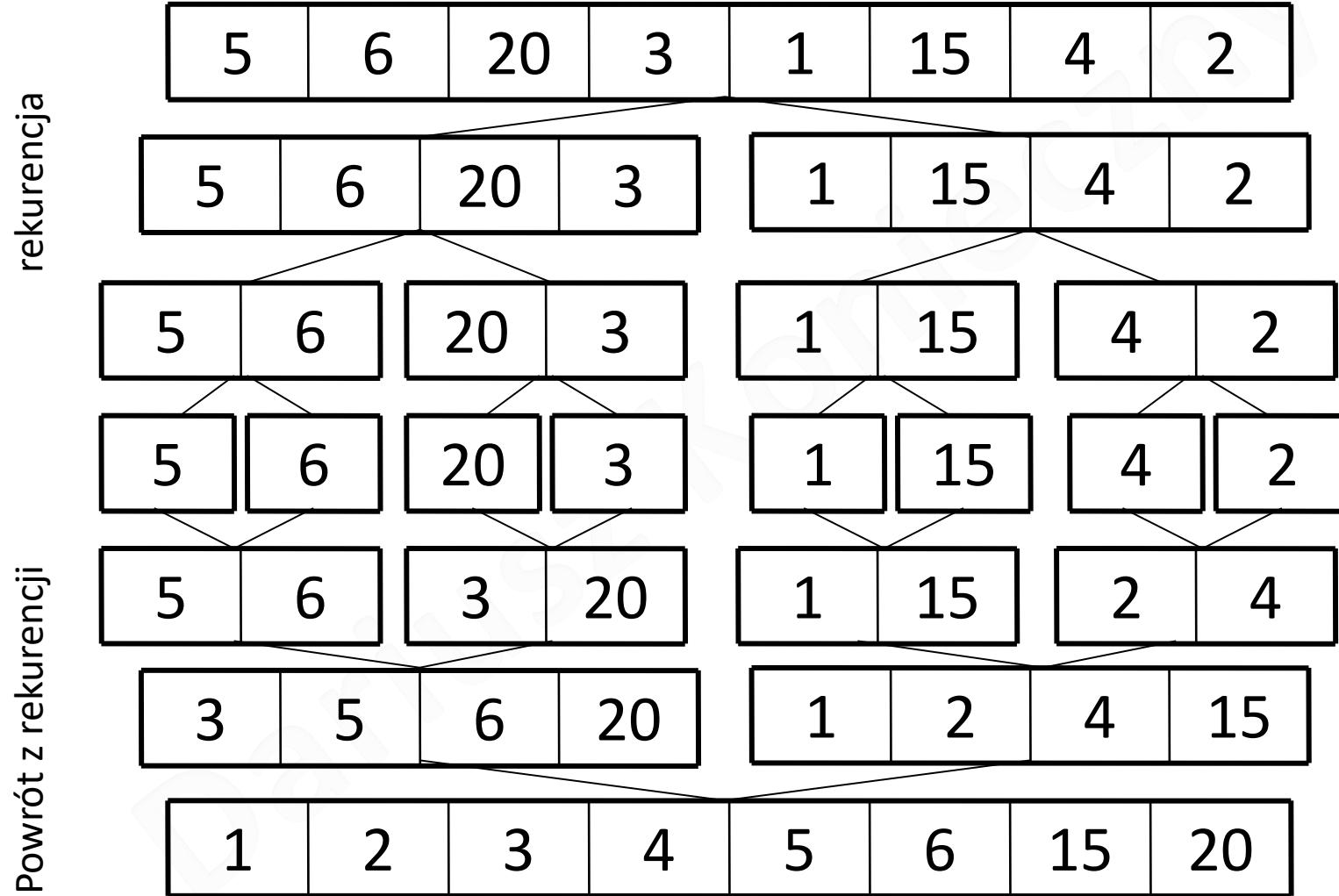
- MergeSort – sortowanie przez scalanie wykorzystuje technikę „dziel i rządź”.

MergeSort:

1. Podziel dane wejściowe na dwie równe (lub prawie równe) części A i B
 2. Posortuje A (za pomocą mergesort)
 3. Posortuje B (za pomocą mergesort)
 4. Połącz (ang. merge) części A i B, wiedząc, że te dwie części są już posortowane
- Zatrzymaj rekurencję jeśli rozmiar części jest 1, gdyż tablica z jednym elementem jest już posortowana.

MergeSort - przykład

- Na wykładzie z sortowania będzie pokazane, jak scalić dwie posortowane części w posortowaną całość w czasie liniowym



Programowanie dynamiczne

- Programowanie dynamiczne (PD) jest **podobne** do techniki „dziel i rządź” w tym sensie, że bazuje również na rekurencyjnym **podziale problemu na podproblemy** o mniejszym rozmiarze lub prostsze. Aczkolwiek, gdy w technice „dziel i rządź” używa się podejścia „**top-down**”, programowanie dynamiczne rozwiązuje problem w sposób „**bottom-up**”: zaczyna rozwiązywać od najprostszych problemów, potem składa je w rozwiązania większych problemów, aż dojdzie do wejściowego problemu.
- W przeciwnieństwie do „dziel i rządź” podproblemy mogą współdzielić cząstkowe rozwiązania podpodproblemów.

PD – problem LCS

Najdłuższy wspólny podciąg (ang. *longest common subsequence* - LCS)

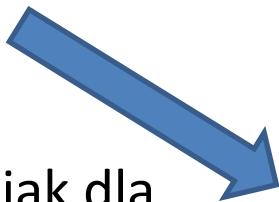
- Niech A będzie ciągiem znaków $A=a_0a_1\dots a_{n-1}$. Podciągiem ciągu A jest ciąg
$$T = a_{i_0}a_{i_1}\dots a_{i_{k-1}} \text{ gdzie } 0 \leq i_0 < i_1 < \dots < i_{k-1} < n$$
- Na przykład: „samples” -> „sms”, „ss”, „mp”
- Dla dwóch ciągów $A=a_0a_1\dots a_{n-1}$ oraz $B=b_0b_1\dots b_{m-1}$ szukamy takiego najdłuższego ciągu C, że będzie podciągiem zarówno ciągu A jak i ciągu B.

LCS - rozwiązańe

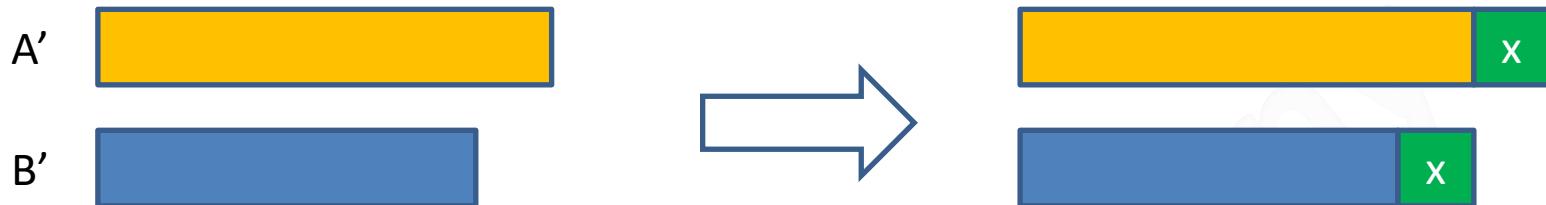
- Dla uproszczenia chcemy obliczyć **długość** LCS. Niech $LCS[i,j]$ będzie długością najdłuższego wspólnego podcięgu dla ciągów $A'=a_0a_1\dots a_{i-1}$ oraz $B'=b_0b_1\dots b_{j-1}$

$$LCS[i, j] = \begin{cases} 0 & \text{dla } i = 0 \text{ lub } j = 0 \\ LCS[i-1, j-1] + 1 & \text{dla } a_{i-1} = b_{j-1} \\ \max(LCS[i, j-1], LCS[i-1, j]) & \text{w.p.p.} \end{cases}$$

- Teoretycznie można to równanie rozwiązywać rekurencyjnie (jak dla techniki „dziel i rządź”), jednak lepiej użyć dwuwymiarowej tablicy do policzenia wartości LCS wiersz po wierszu od 0 do $m-1$ oraz dla każdego wiersza w komórkach od 0 do $n-1$

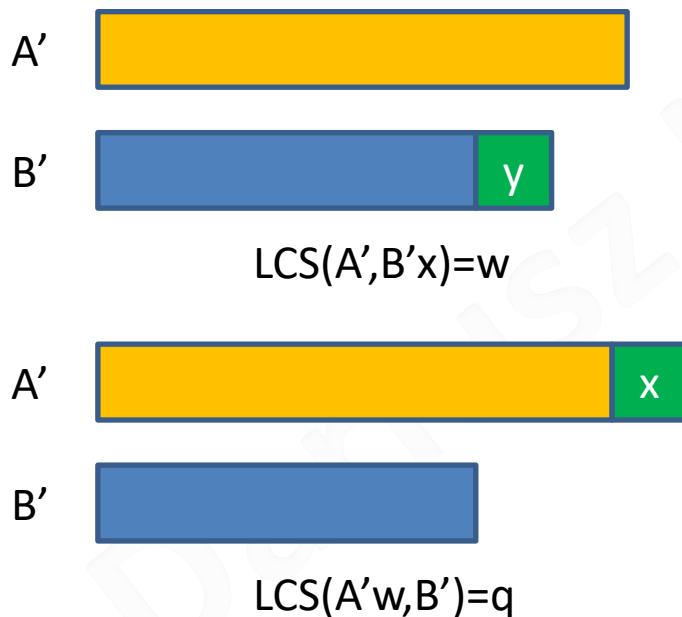


LCS - rozwiązańe



$$\text{LCS}(A', B') = k$$

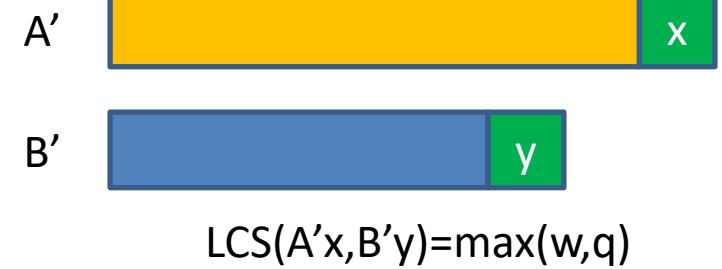
$$\text{LCS}(A'x, B'x) = k+1$$



$$\text{LCS}(A', B'x) = w$$



$$\text{LCS}(A'w, B') = q$$



$$\text{LCS}(A'x, B'y) = \max(w, q)$$

LCS – przykład 1/2

- A=„abbaa”
- B=„bababab”
- n=5
- m=7
- $\text{LCS}[n,m]=$

	b	a	b	a	b	a	b	a	b
0	0	1	2	3	4	5	6	7	
a	1	0	0	1	1	1	1	1	1
b	2	0	1	1					
b	3								
a	4								
a	5								

LCS – przykład 2/2

- A=„abbaa”
 - B=„bababab”
 - n=5
 - m=7
-
- $\text{LCS}[n,m]=4$

$O(n*m)$
 $O(n^2)$

		b	a	b	a	b	a	b	
		0	1	2	3	4	5	6	7
a	0	0	0	0	0	0	0	0	0
	1	0	0	1	1	1	1	1	1
b	2	0	1	1	2	2	2	2	2
	3	0	1	1	2	2	3	3	3
a	4	0	1	2	2	3	3	4	4
	5	0	1	2	2	3	3	4	4

Algorytm zachłanny

Metoda zachłanna stosowana do rozwiązywania problemów optymalizacyjnych bazuje na filozofii zachłanego **maksymalizowania** (lub **minimalizowania**) **krótkoterminowego zysku** mając nadzieję na najlepsze rozwiązanie bez względu na długoterminowe konsekwencje.

Podejmowanie decyzji opartej na krótkoterminowym zysku **może nie prowadzić** do rozwiązania **optymalnego**. Zatem zawsze **należy udowodnić**, że rozwiązanie zachłanne **rzeczywiście jest optymalne**.

Zaleta: algorytmy oparte na metodzie zachłannej są zwykle **bardzo proste, łatwe do zakodowania i efektywne**

Wada: Zdarza się jednak, że tworząc algorytm bazujący na metodzie zachłannej możemy **często kończyć działanie na wyniku gorszym od optymalnego**.

Zaleta: Dla wielu istotnych problemów metoda zachłanna dostarcza optymalnego wyniku (i zostało to dowiedzione)

Zaleta: Dla wielu istotnych problemów, chociaż metoda zachłanna nie tworzy optymalnego wyniku, to w pewnym sensie otrzymujemy dobre przybliżenie do wyniku optymalnego.

Algorytm zachłanny - zarys

```
procedure Greedy(S,Solution)
input: S (base set)
output: Solution
    PartialSolution = Ø      // początkowe rozwiązanie jest puste
    R=S
    while PartialSolution is not a solution and R!=Ø do
        x=GreedySelect(R)          // wybór zachłanny
        R=R\{x}
        if PartialSolution U {x} is feasible then // dopuszczalne rozwiązanie?
            PartialSolution= PartialSolution ∪ {x} // zatem dodaj do rozwiązania
        endif
    endwhile
    if PartialSolution is a solution then
        Solution=PartialSolution           // jest rozwiązanie!
    else
        write(„Greedy fails to produce a solution”) // nie udało się
    endif
end Greedy
```

Alg. zachłanny – wydawanie reszty

Wydawanie reszty – przypuśćmy, że właśnie kupiliśmy coś i sprzedawca chce nam wydać dokładną resztę używając **najmniejszą liczbę** monet. Zakładamy, że jest wystarczająca liczba monet każdej denominacji.

- Algorytm zachłanny wydaje resztę używając jak największej monet najwyższej denominacji, potem bierze pod uwagę drugą, pod względem wielkości, denominację itd.

Wydawanie reszty – przykład 1/2

Możliwe monety: 1gr, 2gr, 5gr, 10gr, 20gr, 50gr

- Reszta do wydania =97gr
- wydaj 50gr, wydaj =47gr
- wydaj 20gr, zostało =27gr
- wydaj 20gr, zostało =7gr
- wydaj 5gr, zostało =2gr
- wydaj 2gr, zostało =0gr
- Liczba monet = 5

Można udowodnić, że dla tego zestawu denominacji algorytm zachłanny generuje zawsze optymalny wynik

Wydawanie reszty – przykład 2/2

Możliwe monety : 1gr, 4gr, 5gr, 10gr, 40gr, 50gr

- Reszta do wydania = 88gr
 - wydaj 50gr, wydaj =38gr
 - wydaj 10gr, wydaj =28gr
 - wydaj 10gr, wydaj =18gr
 - wydaj 10gr, wydaj =8gr
 - wydaj 5gr, wydaj =3gr
 - wydaj 1gr, wydaj =2gr
 - wydaj 1gr, wydaj =1gr
 - wydaj 1gr, wydaj =0gr
 - Liczba monet = 8
-
- Jednak najlepsze rozwiązanie to 40gr+40gr+4gr+4gr, liczba monet = 4

Gdyby pod krótkoterminowością w tym przypadku rozumieć 3 kolejne denominacje i skomplikować algorytm, można by otrzymywać optymalne rozwiązanie

Czy to byłby nadal algorytm zachłanny?

Inne techniki

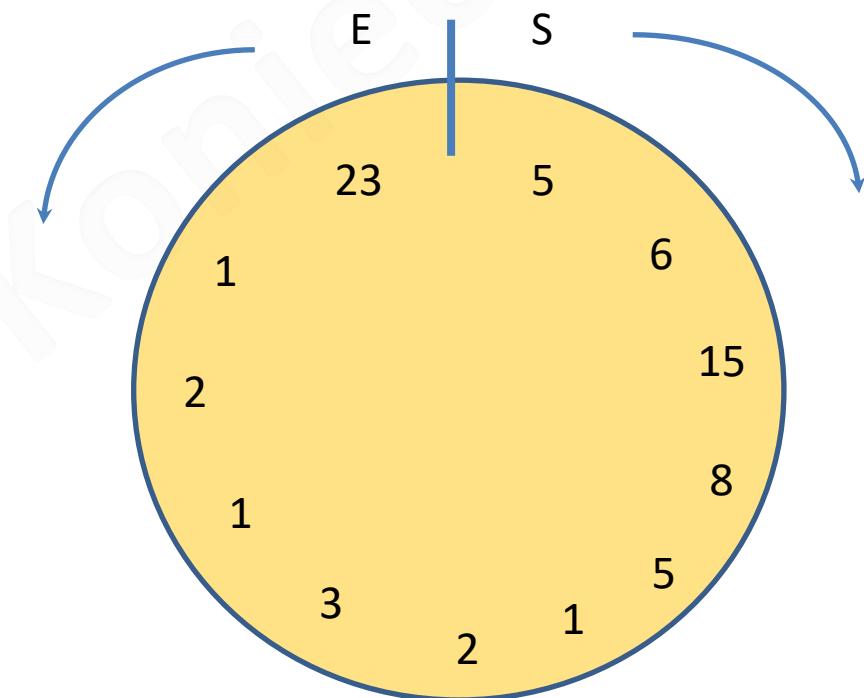
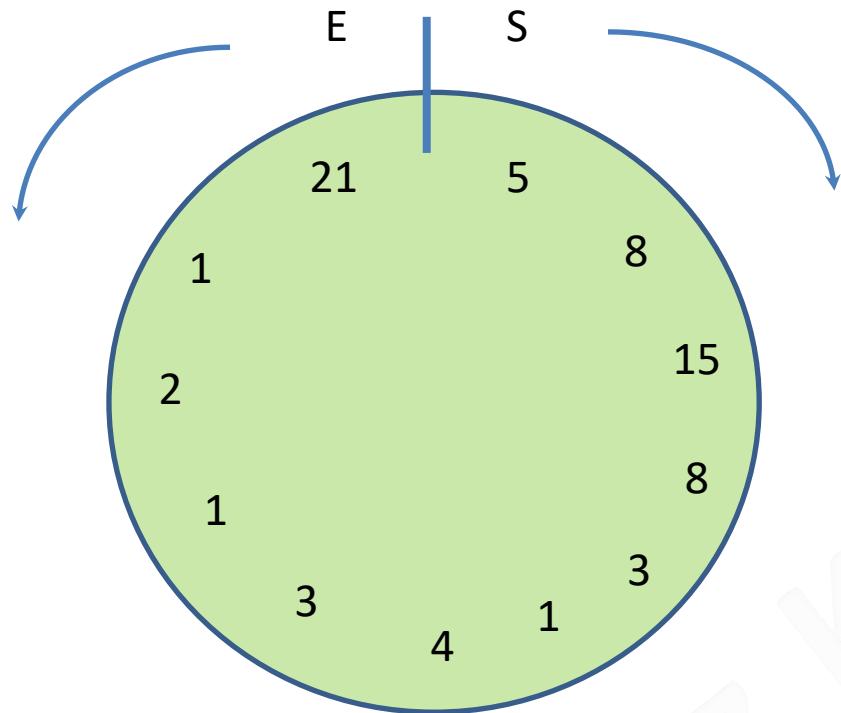
- Brute force – przegląd zupełny
 - Wygenerowanie wszystkich rozwiązań, odrzucenie niedopuszczalnych i spośród pozostałych wybranie optymalnego.
 - Jeśli dopiero poszukujemy algorytmu, przydatne do stworzenia poprawnych testów.
- Przeszukiwawcza przestrzeni stanów
 - Podobna do przeglądu zupełnego, jednak potrafi odrzucić zbiory rozwiązań, które nigdy nie doprowadzą do poprawnego rozwiązania.
- Technika zamiatania (miotły)
 - Raczej w algorytmach graficznych, po wstępnym przetworzeniu danych zbiera się informacje do rozwiązania dodając np. w kolejności: od lewej do prawej, wierszami od dołu do góry
- Itd.

Przykładowe zadania

- <http://livearchive.onlinejudge.org/>
- <https://icpcarchive.ecs.baylor.edu/>
- [->Browse Problems->ICPC Archive Volumes](#)
 - **2535 - Magnificent Meatballs**
 - **2122 - Recognizing S Expressions**
 - **2487 - Lollies**
 - **3390 - Pascal's Travels**

Wspólne rozwiązywanie problemów w czasie wykładu.

Magnificent Meatballs - przykłady



S-expression przykłady

- Które to S-wyrażenia?

- t
- #
- tt
- t,t
- (t,t)
- ((a,b))
- (t,a,b)
- ((A,a),b)
- ((a,b),c
- ((a,b),(c,g))
- (((t,u),w),h)
- (q,(w,(e,r)))
- [x,y]

Lollies - przykłady

dzień	lizaki	czekaj	rozw.
1	3	4	
2	5	8	
3	4	6	
4	2	3	
5	3	7	
6	4	3	
7	4	3	
8	3	1	
9	2	3	

Algorytmy i struktury danych – W13

Wyszukiwanie wzorca w tekście

Algorytm unifikacji

Zawartość

- Wyszukiwanie wzorca w tekście:
 - Definicje
 - Algorytm naiwny – analiza
 - Algorytm Rabina-Karpa
 - Algorytm z wykorzystaniem automatów skończonych
 - Algorytm Knutha-Morrisa-Pratta (KMP)
- Unifikacje:
 - Definicje
 - Algorytm unifikacji

ϵ – epsilon

δ – delta

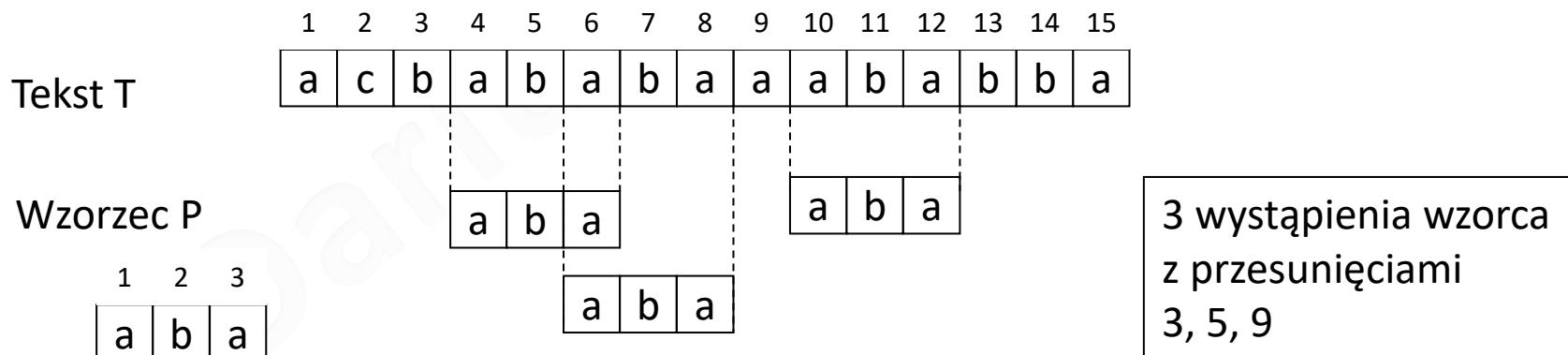
ϕ – fi

σ – sigma

π - pi

Problem wyszukiwania wzorca

- Zakładamy, że:
 - Tekst jest tablicą $T[1..n]$ długości n
 - Wzorzec jest tablicą $P[1..m]$ o długości $m \leq n$.
 - Elementy z P oraz T są symbolami z pewnego skończonego alfabetu Σ .
 - Np. $\Sigma = \{0, 1\}$ lub $\Sigma = \{a, b, \dots, z\}$.
- Teksty P oraz T są często nazywane **słownymi**.
- Mówimy, że wzorzec P **występuje z przesunięciem s** w tekście T (lub, równoważnie, że wzorzec P **występuje, począwszy od pozycji $s+1$**), gdy $0 \leq s \leq n - m$ oraz $T[s + 1..s + m] = P[1..m]$
 - Tzn. gdy $T[s + j] = P[j]$, dla $1 \leq j \leq m$.
- Jeśli P występuje z przesunięciem s w tekście T , to s nazywamy **poprawnym przesunięciem**. W p.p. s nazywamy **niepoprawnym przesunięciem**. Problem wyszukiwania wzorca polega na znajdowaniu wszystkich poprawnych przesunięć dla danego wzorca P w ustalonym tekście T .



Notacja i terminologia 1/2

- Przez Σ^* (czyt. "sigma-gwiazdka") oznaczamy zbiór wszystkich tekstów (słów) utworzonych z symboli alfabetu Σ . Rozważamy tylko słowa o skończonej długości.
- Słowo o długości zero, nazywane słowem pustym i oznaczane przez ϵ , również należy do Σ^* .
- Długość słowa x oznaczamy jako $|x|$.
- Konkatenacja (złożenie) dwóch słów x i y , oznaczane jako xy , jest słowem o długości $|x| + |y|$ składającym się z symboli x , za którymi występują symbole y .
- Słowo w jest **prefiksem** słowa x (co oznaczamy $w \sqsubset x$), gdy $x = wy$ dla pewnego słowa $y \in \Sigma^*$.
 - Jeśli $w \sqsubset x$, to $|w| \leq |x|$.
- Słowo w jest **sufiksem** słowa x (co oznaczamy $w \sqsupset x$), gdy $x = yw$ dla pewnego słowa $y \in \Sigma^*$.
 - Jeśli $w \sqsupset x$, to $|w| \leq |x|$.
- Słowo puste ϵ jest jednocześnie prefiksem i sufiksem każdego słowa.
- Przykład: $ab \sqsubset abcca$ oraz $cca \sqsupset abcca$.
- Dla każdych słów x i y oraz każdego symbolu a , mamy $x \sqsupset y$ wtedy i tylko wtedy, gdy $xa \sqsupset ya$.
- Relacje \sqsubset oraz \sqsupset są przechodnie.

Notacja i terminologia 2/2

- **Lemat o nierozłączności sufiksów:**

Założmy, że x, y oraz z są słowami takimi, że $x \sqsupseteq z$ i $y \sqsupseteq z$.

Jeśli $|x| \leq |y|$, to $x \sqsupseteq y$. Jeśli $|x| \geq |y|$, to $y \sqsupseteq x$. Jeśli $|x| = |y|$, to $x = y$.

Dla zwięzłości zapisu:

- k -symbolowy prefiks $P[1..k]$ wzorca $P[1..m]$ zapiszemy jako P_k .
 - zatem $P_0 = \varepsilon$ and $P_m = P = P[1..m]$.

- k -symbolowy prefiks tekstu T zapiszemy jako T_k .

Używając tej notacji możemy wyrazić problem wyszukiwania wzorca jako problem znajdowania wszystkich przesunięć s w przedziale $0 \leq s \leq n-m$ takich, że $P \sqsupseteq T_{s+m}$.

Test " $x = y$ " potrzebuje (z definicji) czasu $\Omega(t + 1)$, gdzie t jest długością najdłuższego słowa z takiego, że $z \sqsubseteq x$ i $z \sqsubseteq y$.

Problem poszukiwania wzorca występuje przy poszukiwaniu genów z ciągu DNA to poszukiwanie w tekście długości około $3 \cdot 10^8$ symboli (zasady A/T/G/C).

Algorytm naiwny - kod

```
Naive_String_Matcher(T, P)
{ 1} n := length[T]
{ 2} m := length[P]
{ 3} for s:=0 to n-m do
{ 4}   if P[1..m]=T[s+1..s+m] then
{ 5}     print „wzorzec występuje z przesunięciem ” s
```

$T=a^n, P=a^m$



$(n-m+1)m$ porównań symboli

$T=a^n, P=a^{m-1}b$



Złożoność pesymistyczna: $O((n-m+1)m)$

$T=„losowy”, P=„losowy”$



Średnio $\leq 2(n-m+1)$
porównań symboli



Średnia złożoność: $O(n-m+1)$

Algorytm Rabina-Karpa - idea

- Dla uproszczenia załóżmy, że $\Sigma = \{0, 1, 2, \dots, 9\}$,
 - Ogólnie symbol jest cyfrą w systemie o podstawie d , gdzie $d = |\Sigma|$. Każde słowo z k symbolami to inaczej k-cyfrowa liczba w systemie d.
 - Np. tekst „31415” można interpretować jako liczbę dziesiętną 31 415.
- Dla danego wzorca $P[1..m]$, niech p oznacza odpowiadającą mu wartość dziesiętną
- Podobnie dla tekstu $T[1..n]$, niech t_s wartość dziesiętną podsłowa $T[s + 1..s + m]$ o długości m , dla $s = 0, 1, \dots, n-m$.
- Oczywiście $t_s = p$ wtedy i tylko wtedy, gdy $T[s+1..s+m] = P[1..m]$; zatem s jest poprawnym przesunięciem wtedy i tylko wtedy, gdy $t_s = p$. Jeśli możemy policzyć p w czasie $\Omega(m)$ oraz wszystkie wartości t_i w sumie w czasie $\Omega(n)$, to całe obliczenia zajmą $\Omega(n)$.
- Może obliczyć p w czasie $\Omega(m)$ używając reguły Hornera:
$$p = P[m] + 10(P[m - 1] + 10(P[m - 2] + \dots + 10(P[2] + 10P[1]))).$$
- Podobnie obliczymy t_0 z ciągu $T[1..m]$ w czasie $\Omega(m)$. W celu wyliczenia pozostałych wartości t_1, t_2, \dots, t_{n-m} w czasie $\Omega(n - m)$, wystarczy zauważyc, że t_{s+1} możemy otrzymać z t_s w stałym czasie, ponieważ

$$t_{s+1} = 10(t_s - 10^{m-1}T[s+1]) + T[s+m+1]$$

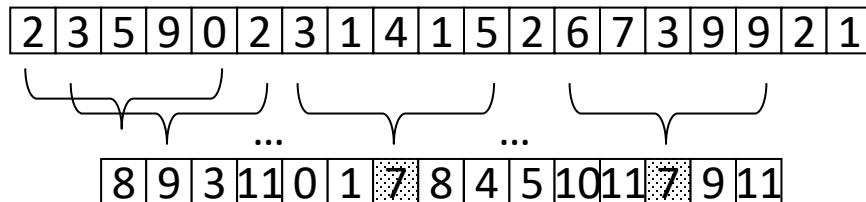
$T=..314152.., m=5, t_s=31415$

$$t_{s+1}=10(31415-10000*3)+2=14152$$

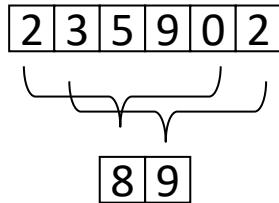
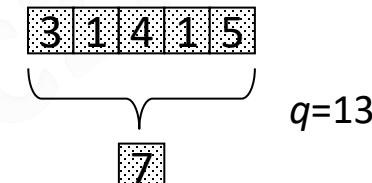
Problem! : p oraz t_s mogą być za duże, aby te operacje matematyczne
Można wykonać w stałym czasie

Algorytm Rabina-Karpa

- Możemy obliczać p oraz t_s modulo odpowiednio dobranej wartości q .



Poprawne
przesunięcie Niepoprawne
przesunięcie



$$\begin{aligned} 35902 &\equiv (23590 - 2 \cdot 10000) * 10 + 2 \pmod{13} \\ &\equiv (8 - 2 \cdot 3) * 10 + 2 \pmod{13} \\ &\equiv 9 \end{aligned}$$

$$t_{s+1} = (d(t_s - T[s+1]h) + T[s+m+1]) \pmod{q} \quad \text{gdzie} \quad h \equiv d^{m-1} \pmod{q}$$

Algorytm Rabina-Karpa - kod

```
Rabin_Karp_Matcher(T, P)
{ 1} n := length[T]
{ 2} m := length[P]
{ 3} h := dm-1 mod q
{ 4} p := 0
{ 5} t0 := 0
{ 6} for i:=1 to m do
{ 7}     p := (d*p+P[i]) mod q
{ 8}     to := (d*t0+T[i]) mod q
{ 9} for s:=0 to n-m do
{10}    if p=ts then
{11}        if P[1..m]=T[s+1..s+m] then
{12}            print „wzorzec występuje z przesunięciem ” s
{13}        if s < n-m then
{14}            ts+1 := (d*(ts-T[s+1]*h)+T[s+m+1]) mod q
```

Wartość modułu q jest dobierana najczęściej tak, aby dq mieściła się w pamięci komputera (w słowie maszynowym)

Algorytm Rabina-Karpa - analiza

- Analizę złożoności algorytmów często dzieli się na dwie fazy:
 - **Preprocessing:** wykonywany aby przygotować dane do działania. Jeśli pewne dane się powtórzą w kolejnej instancji problemu, nie trzeba tych obliczeń wykonywać (tutaj są to obliczenia na wzorcu)
 - **Właściwy algorytm:** używa danych z preprocessingu i pozostałych danych
- Procedura RABIN-KARP-MATCHER potrzebuje $\Omega(m)$ czasu na preprocessing (obliczenie wartości p).
- W pesymistycznym przypadku porównywanie zajmie czas $\Omega((n - m + 1)m)$
 - Np. dla $P = a^m$ oraz $T = a^n$ wszystkie przesunięcia są poprawne i zawsze trzeba robić porównywanie ciągów w czasie $\Omega(m)$
- W wielu aplikacjach spodziewamy się, że liczba wystąpień wzorca jest niewielka, można założyć, że rzędu $O(1)$. Wówczas, po analizie heurystycznej, okazuje się, że czas przeanalizowania całego tekstu będzie rzędu $O(n)$, co da całociowy czas równy $O(n)$.

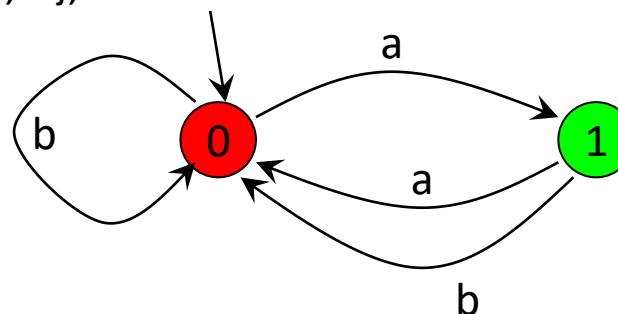
Automat skończony 1/2

- **Automatem skończonym M nazywamy uporządkowaną piątkę $(Q, q_0, A, \Sigma, \delta)$,** gdzie:
 - Q jest skończonym zbiorem **stanów** automatu,
 - $q_0 \in Q$ jest **stanem początkowym**,
 - $A \subseteq Q$ jest zbiorem **stanów akceptujących**,
 - Σ jest **alfabetem wejściowym**,
 - δ jest funkcją z $Q \times \Sigma \rightarrow Q$, zwana **funkcją przejść** automatu M .

Automat skończony rozpoczyna działanie w stanie q_0 , a następnie wczytuje kolejne symbole słowa wejściowego. Jeśli automat jest w stanie q i czyta symbol wejściowy a , to przechodzi od stanu q do stanu $\delta(q, a)$. Jeśli bieżący stan q jest elementem A , to mówimy, że automat M **akceptuje** wczytany dotychczas tekst. W p.p. mówimy, że **odrzuca** wczytany tekst.

$$Q = \{0, 1\}, q_0=0, A=\{1\} \Sigma = \{ a, b\},$$

		symbol	
		a	b
stan	0	1	0
	1	0	0



Automat skończony 2/2

- Dla automatu M oznaczmy przez $\phi: \Sigma^* \rightarrow Q$ tak zwaną **rozszerzoną funkcję przejść**, zdefiniowaną w ten sposób, że $\phi(w)$ jest stanem, do którego przechodzi M po wczytaniu tekstu w . Zatem M akceptuje w wtedy i tylko wtedy, gdy $\phi(w) \in A$. Funkcja ϕ jest zdefiniowana rekurencyjnie:

$$\phi(\epsilon) = q_0$$

$$\phi(wa) = \delta(\phi(w), a), \quad \text{for } w \in \Sigma^*, a \in \Sigma$$

Definiujemy pomocniczą funkcję σ , nazywaną **funkcją sufiksową** wzorca P . Funkcja σ jest odwzorowaniem z Σ^* w zbiór $\{0, 1, \dots, m\}$ takim, że $\sigma(x)$ jest długością największego prefiksu P , który jest jednocześnie sufiksem x :

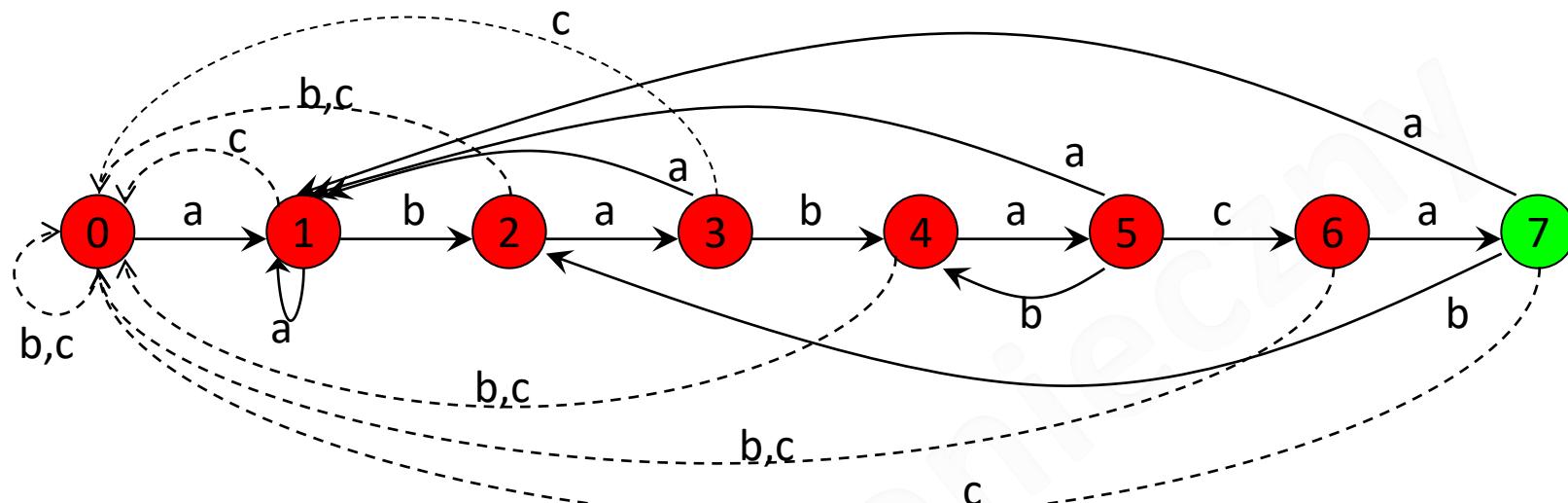
$$\sigma(x) = \max \{ k: P_k \sqsupseteq x \}$$

Automat wyszukiwania danego wzorca $P[1..m]$ definiujemy następująco:

- Zbiorem stanów jest $Q = \{0, 1, \dots, m\}$. Stanem początkowym jest $q_0 = 0$, a stan m jest jedynym stanem akceptującym.
- Funkcja przejść δ dla każdego stanu q i symbolu a jest zdefiniowana równaniem:

$$\delta(q, a) = \sigma(P_q a)$$

Automat skończony - przykład



δ	a	b	c	P
0	1	0	0	a
1	1	2	0	b
2	3	0	0	a
3	1	4	0	b
4	5	0	0	a
5	1	4	6	c
6	7	0	0	a
7	1	2	0	

i
T[i]
stan $\phi(T[i])$

-	1	2	3	4	5	6	7	8	9	10	11
-	a	b	a	b	a	b	a	c	a	b	a
0	1	2	3	4	5	4	5	6	7	2	3

Algorytm z automatem skończ.

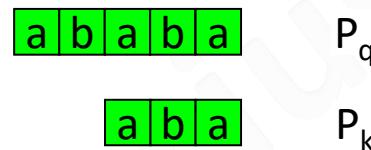
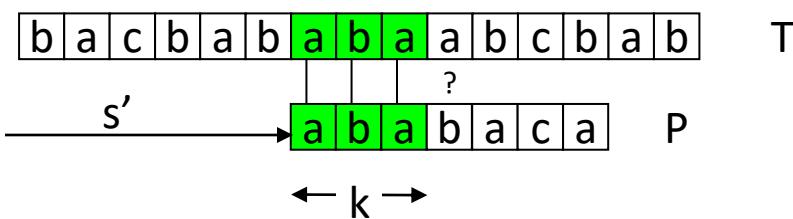
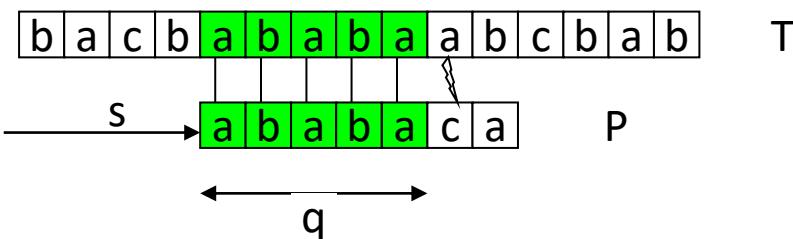
```
Finite_Automaton_Matcher(T, P)
{ 1} n := length[T]
{ 2} q := 0
{ 3} for i:=1 to n do
{ 4}     q := δ(q, T[i])
{ 5}     if q = m then
{ 6}         s := i - m
{ 7}         print „wzorzec występuje z przesunięciem ” s
```

```
Compute_Transition_Function(P, Σ)
{ 1} m := length[P]
{ 2} for q:=0 to m do
{ 3}     for a∈Σ do
{ 4}         k := min(m+1, q+2)
{ 5}         repeat
{ 6}             k := k - 1
{ 7}         until Pk ⊢ Pqa
{ 8}         δ(q, a) := k
{ 9} return δ
```

Alg. z aut. skończonym - analiza

- Czas wykonania procedury COMPUTE-TRANSITION-FUNCTION jest $O(m^3|\Sigma|)$:
 - Zewnętrzne pętle dają współczynnik $m|\Sigma|$
 - Wewnętrzna **repeat** uruchamia się co najwyżej $m + 1$ razy
 - Test $P_k \sqsupseteq P_q a$ w linii 6 może wymagać nawet m porównań symboli.
- Istnieje szybsza procedura usprawniająca wewnętrzne pętle do tego stopnia, że po analizie czasu zamortyzowanego otrzyma się czas $O(m|\Sigma|)$.
- Oznacza to, że usprawniony algorytm obliczania δ znajduje wszystkie wystąpienia wzorca długości m w tekście długości n nad alfabetem Σ w czasie: preprocessing - $O(m|\Sigma|)$ czas porównywania z tekstem - $\Omega(n)$.

Idea Knutha-Morrissa-Pratta



- Dla danego wzorca $P[1..m]$, tworzona jest funkcja prefiksowa dla wzorca P jako funkcja $\pi : \{1, 2, \dots, m\} \rightarrow \{0, 1, \dots, m - 1\}$ taka, że

$$\pi[q] = \max\{k : k < q \text{ oraz } P_k \sqsupseteq P_q\}$$

- To znaczy, że $\pi[q]$ jest długością najbliższego prefiksu P , który jest właściwym sufiksem P_q .

i	1	2	3	4	5	6	7	8	9	10
$P[i]$	a	b	a	b	a	b	a	b	c	a
$\pi[i]$	0	0	1	2	3	4	5	6	0	1

KMP - Kod

```
KMP_Matcher(T,P)
{ 1} n := length[T]
{ 2} m := length[P]
{ 3} π := Compute_Prefix_Function(P)
{ 4} q := 0
{ 5} for i:=1 to n do
{ 6}   while q>0 and P[q+1]≠T[i] do
{ 7}     q := π[q]
{ 8}   if P[q+1] = T[i] then
{ 9}     q := q+1
{10}   if q=m then
{11}     print „wzorzec występuje z przesunięciem “ i-m
{12}     q := π[q]
```

```
Compute_Prefix_Function(P)
{ 1} m := length[P]
{ 2} π[1] := 0
{ 3} k := 0
{ 4} for q:=2 to m do
{ 5}   while k>0 and P[k+1]≠P[q] do
{ 6}     k := π[k]
{ 7}   if P[k+1]=P[q] then
{ 8}     k := k+1
{ 9}   π[q] := k
{10} return π
```

i	1	2	3	4	5	6	7	8	9	10
P[i]	a	b	a	b	a	b	a	b	c	a
π[i]	0	0	1	2	3	4	5	6	0	1

KMP - Analiza

- Używając metody potencjału w analizie zamortyzowanego czasu procedura COMPUTE-PREFIX-FUNCTION wykonuje się w czasie $\Omega(m)$:
 - W pętli **for** (linia 4) możemy tylko raz podnieść wartość k o jeden w linii 8
 - Pętla **while** (linia 5) może tylko obniżyć wartość k, ale nigdy poniżej 0. Czyli sumarycznie może się wykonać tyle razy ile razy k zostanie zwiększone.
 - Pozostałe instrukcje są stałoczasowe $O(1)$.
 - Zewnętrzna pętla wykonuje się $\Omega(m)$ razy, stąd całociowy czas procedura COMPUTE-PREFIX-FUNCTION ma złożoność $\Omega(m)$.
- Analogiczna analiza działania procedury KMP-MATCHER doprowadzi do wniosku, że jej złożoność wynosi $\Omega(n)$.

Złożoność - porównanie

Algorytm	Czas preprocessingu	Pesymistyczny czas porównywania
Naiwny	0	$O((n-m+1)m)$
Rabin-Karp	$\Omega(m)$	$O((n-m+1)m)$
Automat skończony	$O(m \Sigma)$	$\Omega(n)$
Knutt-Morris-Pratt	$\Omega(m)$	$\Omega(n)$

Algorytm unifikacji - definicje

- Unifikacja – Czy można wykonać jakieś podstawienie za zmienne, aby pewne dwa wyrażenia były sobie równe?
 - Np. $p(f(x),g(y))$ oraz $p(f(f(a)),g(z))$?
 - Podstawienie $\{x \leftarrow f(a), y \leftarrow f(g(a)), z \leftarrow f(g(a))\}$
 - Otrzymujemy: $p(f(f(a)),g(f(g(a))))$ oraz $p(f(f(a)),g(f(g(a))))$, czyli te same termy
 - Można użyć podstawienie $\{x \leftarrow f(a), y \leftarrow a, z \leftarrow a\}$
 - Albo wręcz $\{x \leftarrow f(a), z \leftarrow y\}$
 - Ostatnie podstawienie to najbardziej ogólne podstawienie uzgadniające (ang. most general unifier; MGU)
 - W niektórych przypadkach nie da się znaleźć MGU np. $g(x)$ i $f(y)$

Unifikacja - definicje

Dla uproszczenia ograniczmy definicję termu do czterech zasad:

- term ::= x gdzie x to zmienna
- term ::= $f(\text{lista_termów})$ gdzie f to symbol funkcyjny
- lista_termów ::= term
- lista_termów ::= term, lista_termów

Przykład:

- $f(g(a,b),h(c),g(c,c))$
- f,g,h – symbole funkcyjne
- a,b,c - zmienne

Unifikacja – ogólny algorytm

- Wejście – zbiór równań termów
- Wyjście - zbiór równań w postaci MGU lub odpowiedź, że nie można znaleźć MGU.

Przekształcenia (x – jakaś zmienna):

Reg1. Przekształć równanie $t=x$, gdzie t nie jest zmienną, do $x=t$.

Reg2. Usuń równanie postaci $x=x$

Reg3. Niech $t'=t''$ nie są zmiennymi. Jeśli główne symbole funkcyjne termów t' oraz t'' są różne – zakończ algorytm z negatywną odpowiedzią. W p.p. zastąp równanie $f(t'_1, \dots, t'_k) = f(t''_1, \dots, t''_k)$ k równaniami postaci $t'_1 = t''_1, \dots, t'_k = t''_k$.

Reg4. Niech $x=t$ będzie równaniem takim, że zmienna x występuje w zbiorze równań nie tylko po lewej stronie tego równania. Jeśli zmienna x występuje w t – zakończ algorytm z negatywną odpowiedzią. W p.p. zastąp wszystkie wystąpienia zmiennej x w innych równaniach termem t.

Algorytm niedeterministyczny wykonuje te przekształcenia jak długo się da. Jeśli możliwe przekształcenia się skończą – zbiór równań stanowi MGU.

Unifikacja – algorytm deterministyczny

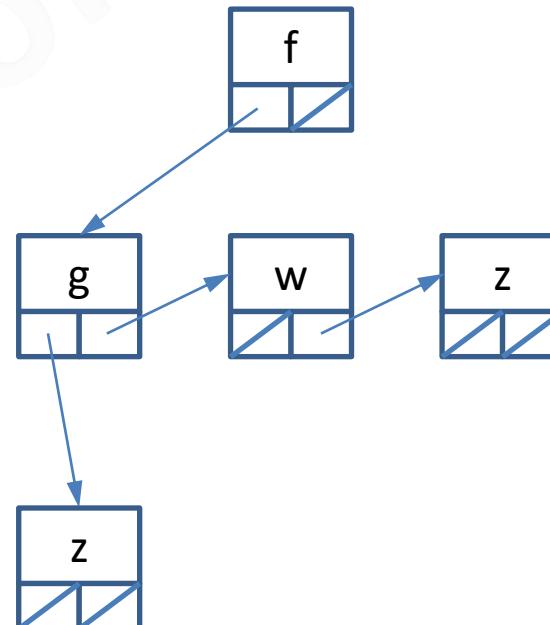
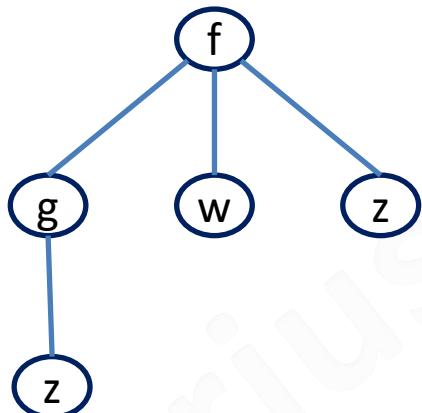
1. Ustaw równania w kolejkę
2. Pobierz równanie z kolejki
3. Jeśli można wykonać przekształcenie **Reg2**, wróć do kroku 2
4. Jeśli można wykonać przekształcenie **Reg1**, równanie wstaw na początek kolejki. Wróć do kroku 2.
5. Jeśli można wykonać przekształczenia **Reg3**, wykonaj je, a powstałe równania wstaw na początek kolejki. Wróć do kroku 2.
6. Jeśli można wykonać przekształcenie **Reg4**, wykonaj je, równanie wstaw na koniec kolejki, wróć do kroku 2
7. Jeśli wszystkie równania były już analizowane - równania w kolejce stanowią MGU, koniec algorytmu. Jeśli nie, wstaw równanie na koniec i wróć do kroku 2.
(w krokach 5 i 6 algorytm dla przekształceń **Reg3** i **Reg4** może zakończyć się odpowiedzią negatywną).

Unifikacja - przykład

- { $g(y)=x, f(x,h(x),y)=f(g(z),w,z)$ } Reg1
- { $x=g(y), f(x,h(x),y)=f(g(z),w,z)$ } Reg4
- { $f(g(y), h(g(y)),y)=f(g(z),w,z), x=g(y)$ } Reg3
- { $g(y)=g(z), h(g(y))=w, y=z, x=g(y)$ } Reg3
- { $y=z, h(g(y))=w, y=z, x=g(y)$ } Reg4
- { $h(g(z))=w, z=z, x=g(z), y=z$ } Reg1
- { $z=z, x=g(z), y=z, w=h(g(z))$ } Reg2
- { $x=g(z), y=z, w=h(g(z))$ }
- Koniec algorytmu: ostatni zbiór stanowi MGU

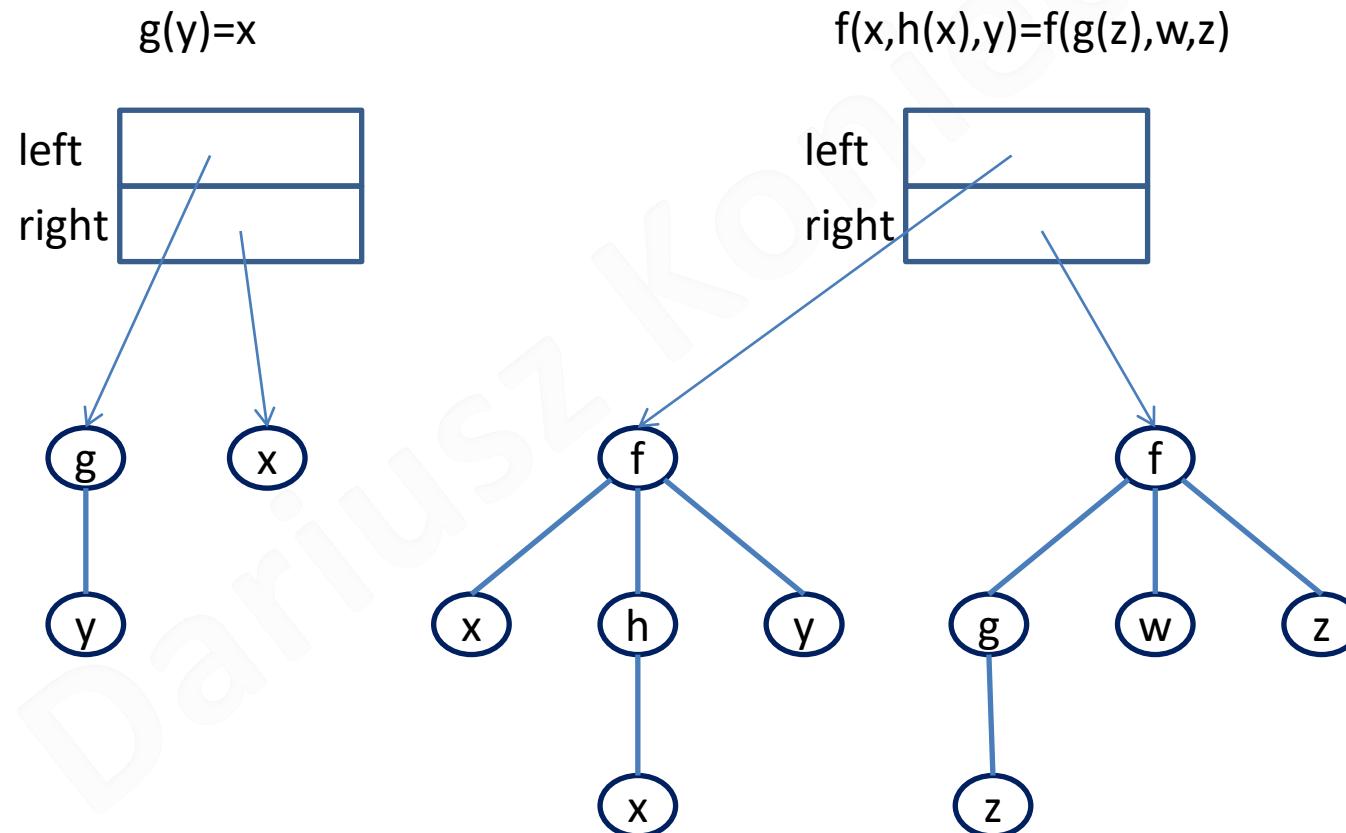
Drzewa termów - implementacja

- Pamiętanie termu jako ciąg znaków jest niewygodne, nieefektywne.
- Drzewa dla termów można zaimplementować jak drzewa z dowolną liczbą dzieci – podobnie jak w kopcu dwumianowym: referencja na najbardziej lewe dziecko oraz na rodzeństwo.
- Dla algorytmu unifikacji niepotrzebna jest referencja na rodzica pamiętana w węźle.
- Węzły wewnętrzne zawierają symbol funkcji
- Liście zawierają symbol zmiennej



Drzewa termów w algorytmie unifikacji

- Jedno równanie to para takich drzew.
- Łatwa analiza i wykonanie operacji związanych z przekształceniами Reg1, Reg2, Reg3, Reg4.



Algorytmy i struktury danych – W14

Kody Huffmana

Problemy plecakowe

Wypukła otoczka

Zawartość

- Kompresowanie danych
 - Kody prefiksowe
 - Kody Huffmana, algorytm tworzenia
- Problem plecakowy:
 - Ciągły problem plecakowy i rozwiązanie
 - Dyskretny problem plecakowy i rozwiązanie
- Wypukła otoczka:
 - Definicje
 - Problem wypukłej otoczki
 - Algorytm Grahama – technika zamiatania i miotły

Kompresja danych

- Założmy, że mamy plik z danymi w postaci 100 000 symboli (znaków), który chcemy skompresować
- Częstości wystąpień symboli jest podana poniższą tabelką.
- W pliku występuje tylko te 6 symboli

	a	b	c	d	e	f
Częstość (w tysiącach)	45	13	12	16	9	5
Słowa kodowe o ustalonej długości	000	001	010	011	100	101
Słowa kodowe o zmiennej długości	0	101	100	111	1101	1100

Kodowanie za pomocą słów o ustalonej długości potrzebuje bitów:
 $3 \cdot 100,000 = 300,000$

Kodowanie za pomocą słów o zmiennej długości potrzebuje bitów:
 $(45 \cdot 1 + 13 \cdot 3 + 12 \cdot 3 + 16 \cdot 3 + 9 \cdot 4 + 5 \cdot 4) \cdot 1,000 = 224,000$
czyli mniej o ok. 25%

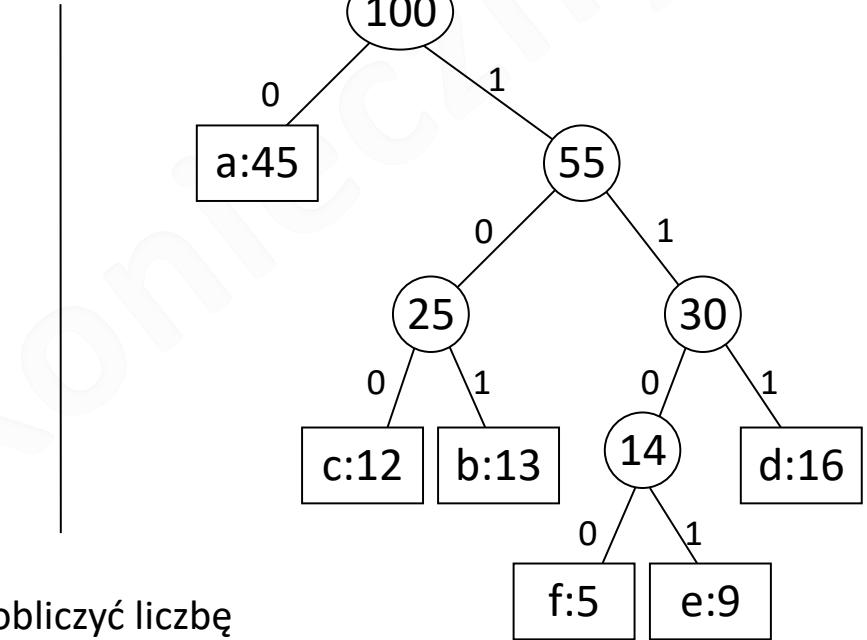
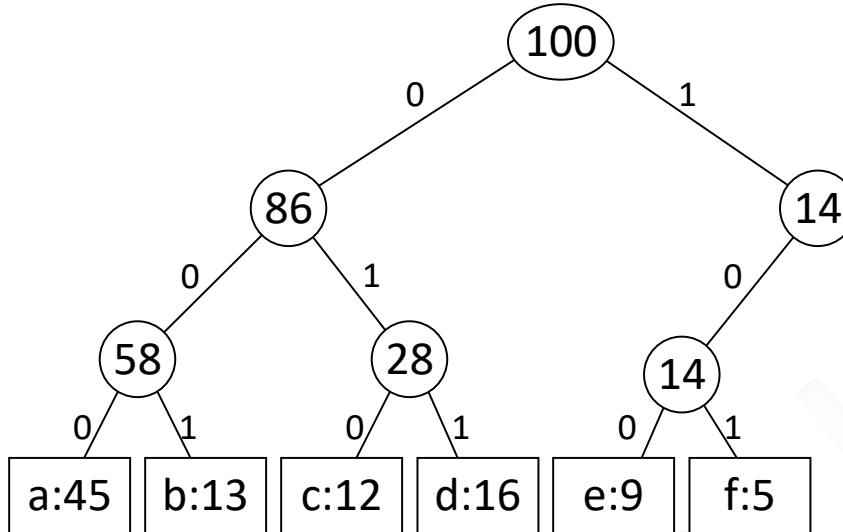
Kody prefiksowe

- Kod żadnego symbolu/znaku nie jest prefiksem kodu innego znaku – **kody prefiksowe**.
- Kodowanie to zamiana symboli na kody bitowe w kolejności jak w pliku wejściowym oraz ich konkatenacja. Np. 3-znakowy plik zawierający ciąg „abc” kodujemy jako $0 \cdot 101 \cdot 100 = 0101100$, gdzie jako symbol konkatenacji używamy ‘·’.
- Dekodowanie (w kodach prefiksowych) polega na znalezieniu od początku ciągu bitów pierwszego kodu. Może być tylko jeden taki kod. „Odcinamy” ten kod z początku ciągu i z pozostałą resztą robimy dokładnie to samo. Np. dla zakodowanego ciągu 001011101 rozkładamy go jednoznacznie na $0 \cdot 0 \cdot 101 \cdot 1101$, co daje słowo „aabe”.
- W praktyce w pliku będzie nagłówek przedstawiający użyte kodowanie.

	a	b	c	d	e	f
Kody prefiksowe	0	101	100	111	1101	1100

Drzewo kodowania

- Liście reprezentują kodowane symbole
- Ścieżka od korzenia do liścia reprezentuje sposób kodowanie symbolu, gdzie droga do lewego dziecka oznacza 0, natomiast do prawego 1



- Mając dane drzewo kodowania T , łatwo jest obliczyć liczbę bitów potrzebnych do zakodowania pliku. Dla każdego znaku c w alfabetie C , niech $f(c)$ oznacza częstość c w pliku oraz niech $d_T(c)$ oznacza głębokość liścia z c w drzewie (jest to również długość słowa kodowego). Liczba bitów potrzebnych do zakodowania pliku jest zatem równa $B(T)$ zwana kosztem drzewa T .

$$B(T) = \sum_{c \in C} f(c) d_T(c)$$

Kody Huffmana - kod

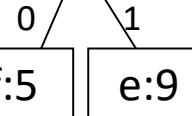
- C to zbiór n symboli.
- Każdy symbol reprezentowany jest przez obiekt posiadający częstotliwość $f[c]$.
- Algorytm buduje drzewo T optymalnego kodowania od dołu: Zaczyna z $|C|$ liśćmi oraz wykonuje ciąg $|C| - 1$ operacji "łączenia" aby otrzymać końcowe drzewo.
- Kolejka priorytetowa Q z atrybutami f jest używana do wyznaczania dwóch obiektów o najmniejszej częstotliwości.
- Wynikiem scalenia dwóch obiektów jest nowy obiekt o częstotliwości równej sumie częstotliwości dwóch jego składowych (jeden z nich staje się lewym, a drugi prawym dzieckiem nowego obiektu)

```
Huffman(C)
{ 1}   n := |C|
{ 2}   Q := C
{ 3}   for i:=1 to n-1 do
{ 4}     z := Allocate_Node()
{ 5}     x := left[z] := Extract_Min(Q)
{ 6}     y := right[z] := Extract_Min(Q)
{ 7}     f[z] := f[x] + f[y]
{ 8}     Insert(Q,z)
{ 9}   return Extract_Min(Q)
```

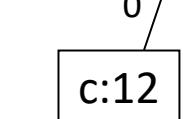
Kody Huffmana - przykład

f:5 e:9 c:12 b:13 d:16 a:45

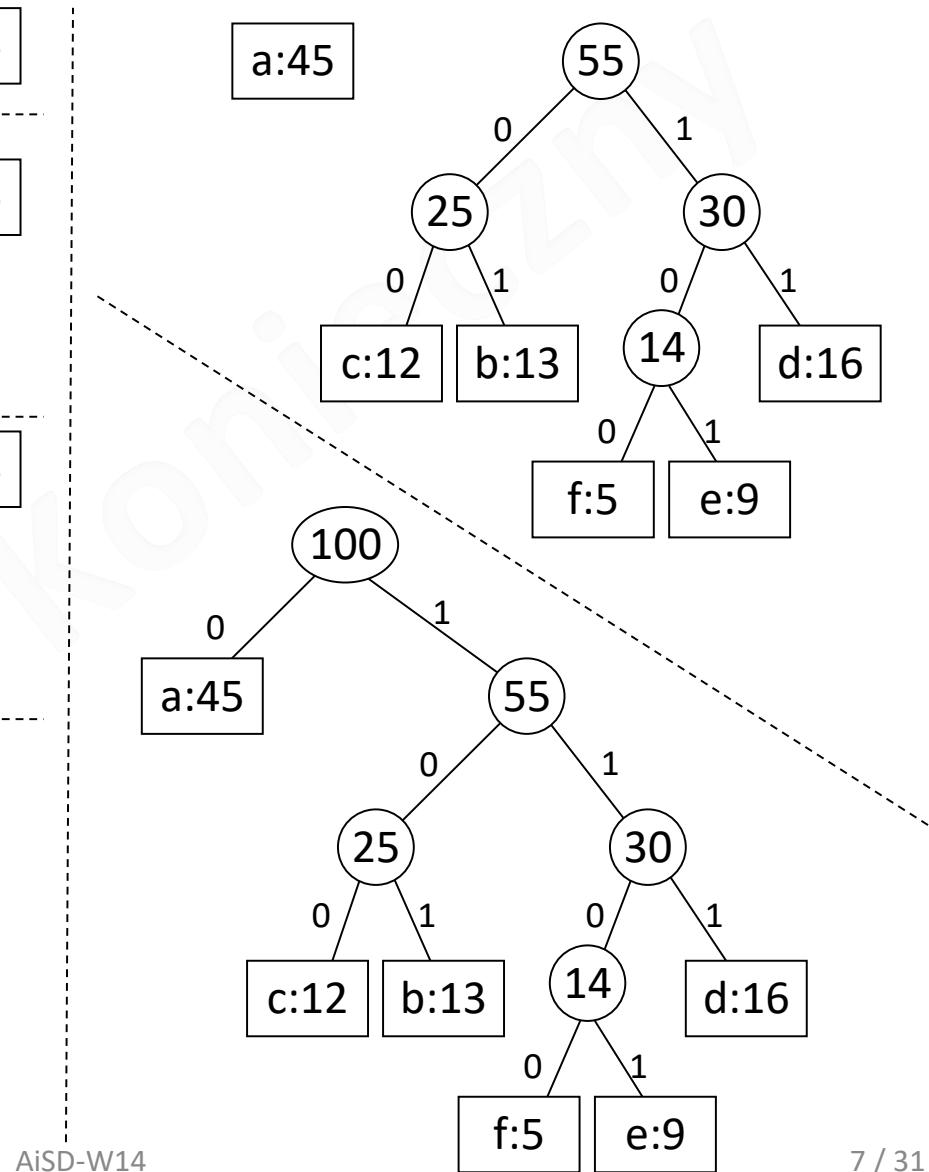
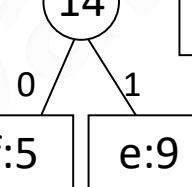
c:12 b:13 14 d:16 a:45



14 d:16 25 a:45
f:5 e:9 c:12 b:13



25 30 a:45
c:12 b:13 14 d:16
f:5 e:9



Kody Huffmana - analiza

- Kolejkę Q jest zaimplementowana jako binarny minheap.
- Dla zbioru C z n znakami budowanie kopca zajmuje czas $O(n)$ time z użyciem procedury BUILD-MIN-HEAP.
- Pętla **for** w liniach 3-8 wykonywana jest dokładnie $n-1$ razy
 - Każda operacja na kopcu wymaga czasu $O(\lg n)$,
 - Czas wykonania pętli to $O(n \lg n)$.
- Całkowita złożoność procedury HUFFMAN na zbiorze n znakowym to $O(n \lg n)$.

Ciekawe zestawy tworzące pochylone drzewa:

a:1 b:1 c:2 d:3 e:5 f:8 g:13 h:21

a:1 b:1 c:2 d:4 e:8 f:16 g:32 h:64

Problemy plecakowe

- Problemy plecakowe:
 - **Dyskretny problem plecakowy** (ang. *0-1 knapsack problem*):
 - Dane: mamy do dyspozycji plecak o maksymalnej pojemności W oraz zbiór n elementów $\{x_1, x_2, \dots, x_j, \dots, x_n\}$ przy czym każdy element ma określoną wartość v_j oraz wielkość w_j
 - Cel: wybrać podzbiór przedmiotów, których sumaryczna wielkość nie przekracza pojemności W oraz ich sumaryczna wartość jest największa możliwa.
 - Założenie: Można wybrać tylko cały przedmiot.
 - **Ciągły problem plecakowy** (ang. *fractional knapsack problem*), takie same dane i cel, ale:
 - Założenie: można wybrać dowolny fragment przedmiotów (dowolną część ułamkową).

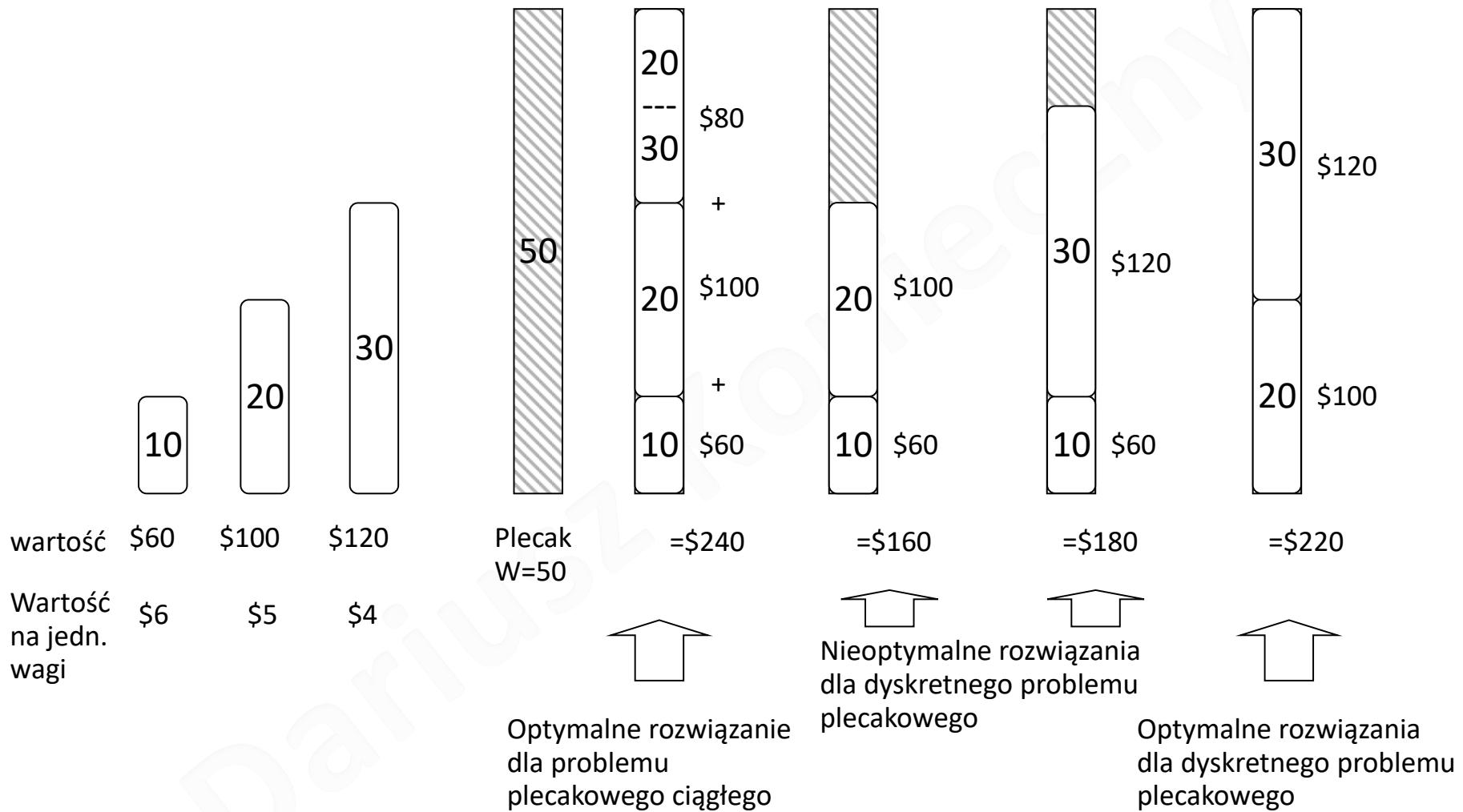
Rozwiążanie 1

Ciągły problem plecakowy – rozwiązanie: (*algorytm zachłanny*):

- Obliczyć wartość na jednostkę wagi v_i/w_i każdego elementu
- Weź jak najwięcej elementu z największą wartością na jednostkę wagi. Jeśli wykorzystałeś cały element, weź najwięcej drugiego w kolejności itd. aż zapełnisz cały plecak
- Z powodu sortowania elementów wg wartości na jednostkę wagi złożoność tego algorytmu wynosi $O(n \lg n)$.

Dla **dyskretnego problemu plecakowego** strategia zachłanna nie działa!

Przykład 1

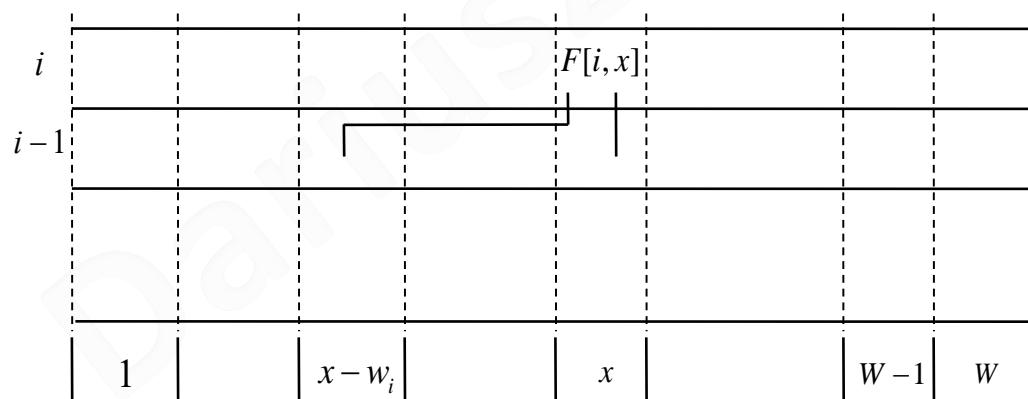


Problem plecakowy dyskr. - rozwiązańie

- **Dyskretny problem plecakowy** – rozwiązanie: *programowanie dynamiczne*
- Znając najlepsze rozwiązanie problemu plecakowego dla przedmiotów $\{v_1, v_2, \dots, v_i\}$ i plecaków o pojemności 1 to W , znajdujemy formułę znajdująca najlepsze rozwiązanie dla zbioru przedmiotów $\{v_1, v_2, \dots, v_i, v_{i+1}\}$

$$\begin{aligned} \sum_{i=1}^n w_i t_i &\leq W & t_i \in \{0,1\} \\ \sum_{i=1}^n v_i t_i &\rightarrow \max \\ F[i, x] = &\begin{cases} 0 & x \geq 0, i = 0 \\ -\infty & x < 0, i = 0 \\ \max\{F[i-1, x], (F[i-1, x - w_i] + v_i)\} & 1 \leq i \leq n \end{cases} \end{aligned}$$

Złożoność:
 $O(nW)$



Przykład 2 – 1/4

	1	2	3	4	5	6	7
wielkość, w_i	3	10	8	6	2	9	
wartość, v_i	5	12	10	7	1	11	

Pojemność plecaka W=20

Przykład 2 – 2/4

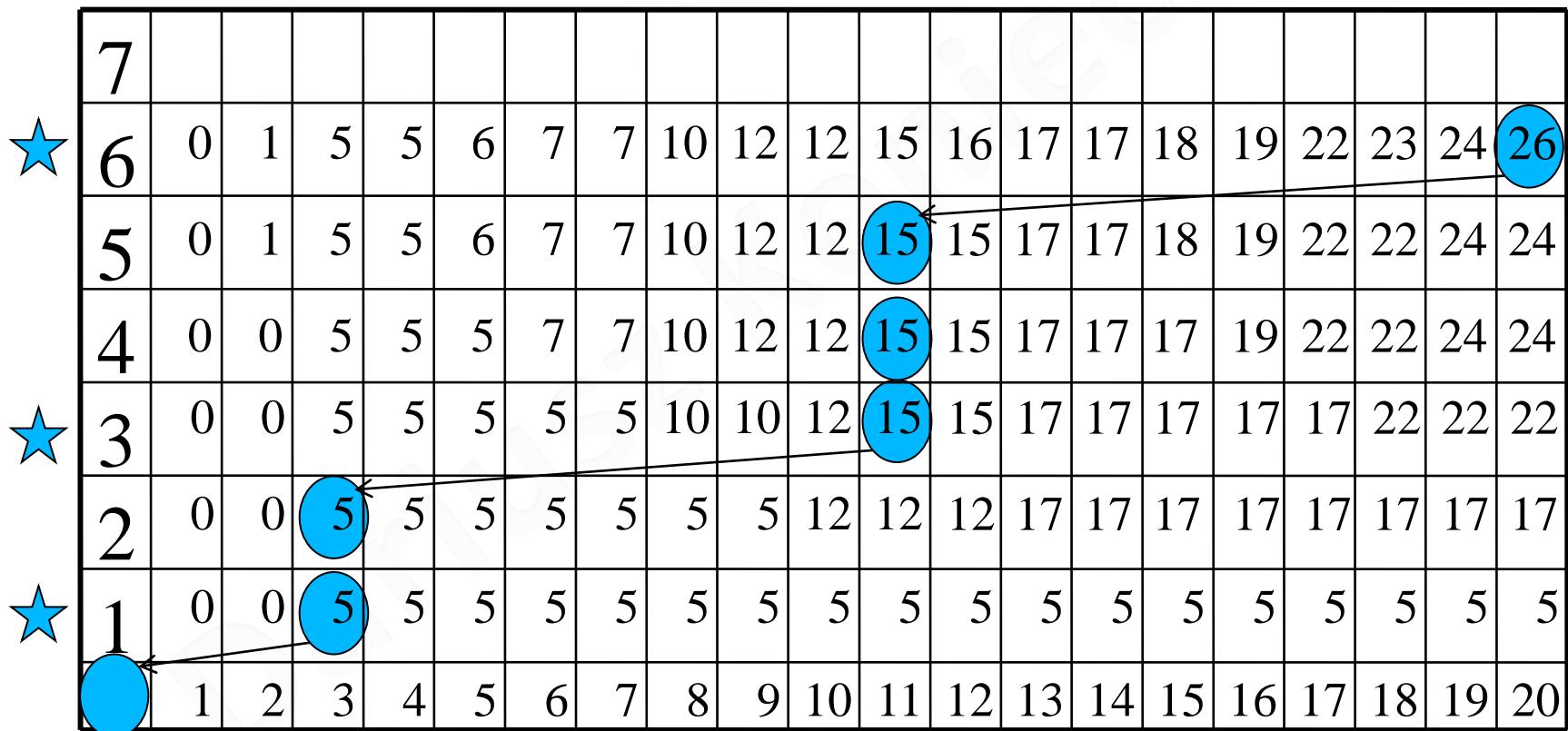
	1	2	3	4	5	6	7
wielkość, w_i	3	10	8	6	2	9	
wartość, v_i	5	12	10	7	1	11	

Pojemność plecaka W=20

Przykład 2 – 3/4

	1	2	3	4	5	6	7
wielkość, w_i	3	10	8	6	2	9	
wartość, v_i	5	12	10	7	1	11	

Pojemność plecaka $W=20$



Przykład 2 – 4/4

	1	2	3	4	5	6	7
wielkość, w_i	3	10	8	6	2	9	
wartość, v_i	5	12	10	7	1	11	

Pojemność plecaka $W=14$

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

Algorytmy geometryczne – pojęcia i problemy

- Podstawowe obiekty geometryczne:
 - **Punkt** p , reprezentowany przez parę współrzędnych (x_p, y_p) w układzie XOY ,
 - **Odcinek** $p-q$, reprezentowany przez parę punktów p oraz q ,
 - **Wektor** o początku w p a końcu w q , oznaczany jako $p \rightarrow q$
 - **Prosta**, reprezentowana przez dowolną parę różnych punktów leżących na niej.
- Przykłady problemów:
 - względne położenie punktów,
 - po której stronie wektora $p \rightarrow q$ leży punkt r
 - czy punkty x i y leżą po tej samej stronie prostej $p-q$
 - czy punkt r należy do odcinka $p-q$
 - czy odcinki $p-q$ oraz $r-s$ przecinają się
 - czy punkt p leży wewnątrz wielokąta W ,
 - znajdowanie otoczki wypukłej zbioru punktów
 - czy w zbiorze odcinków istnieją dwa odcinki przecinające się
 - wyznaczanie najmniejszej odległości w zbiorze punktów.
 - ...

Względne położenie punktów 1/2

- Dopuszczalne operacje arytmetyczne to **dodawanie, odejmowanie i mnożenie**. Pozostałe powodują zaokrąglenia i niedokładne obliczenia!
 - Wyznaczanie współczynnika nachylenia prostej – NIE!
- Atomową operacją używaną w algorytmach jest operacja wyznaczania względnego położenia trzech punktów:

$$p = (x_p, y_p), q = (x_q, y_q), r = (x_r, y_r)$$

- Konstruujemy wyznacznik ·

$$\det(p, q, r) = \det \begin{bmatrix} x_p & y_p & 1 \\ x_q & y_q & 1 \\ x_r & y_r & 1 \end{bmatrix}$$

- Znak wyznacznika jest równy znakowi sinusa kąta między wektorami $p \rightarrow r$ oraz $p \rightarrow q$.

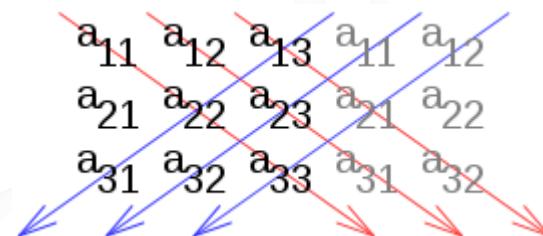
Wyznacznik – metoda Sarrusa

Obliczanie wyznacznika macierzy 3-stopnia (reguła Sarrusa) :

- Aby obliczyć wyznacznik:

$$\begin{vmatrix} a_{11} & a_{12} & a_{13} \\ a_{21} & a_{22} & a_{23} \\ a_{31} & a_{32} & a_{33} \end{vmatrix}$$

- dopisuje się z prawej strony dwie pierwsze kolumny:

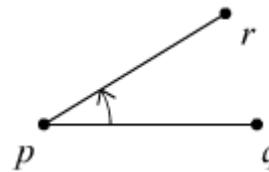


- a następnie oblicza się sumę iloczynów wzdłuż **czerwonych strzałek** i odejmuje od niej sumę iloczynów wzdłuż **niebieskich strzałek**. Ogólny wzór ma postać następującą:

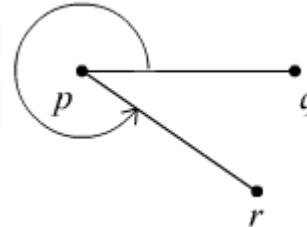
$$(a_{11} \cdot a_{22} \cdot a_{33} + a_{12} \cdot a_{23} \cdot a_{31} + a_{13} \cdot a_{21} \cdot a_{32}) - (a_{13} \cdot a_{22} \cdot a_{31} + a_{11} \cdot a_{23} \cdot a_{32} + a_{12} \cdot a_{21} \cdot a_{33})$$

Względne położenie punktów 2/2

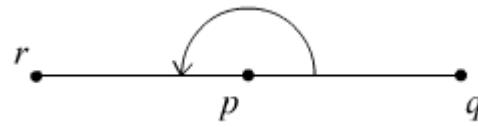
- Punkt r leży po lewej stronie wektora $p \rightarrow q$, jeżeli $\det(p, q, r) > 0$.



- Punkt r leży po prawej stronie wektora $p \rightarrow q$, jeżeli $\det(p, q, r) < 0$.



- Jeżeli $\det(p, q, r) = 0$ to powiemy, że punkty są współliniowe.

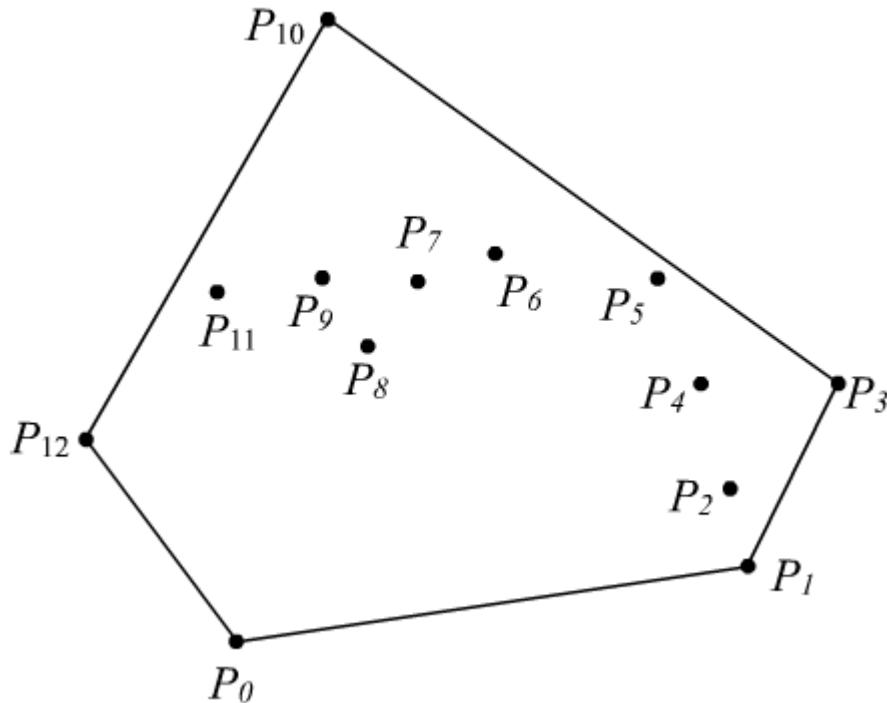


Metoda miotły i zamiatanie

- Oprócz wcześniej poznanych technik rozwiązywania problemów, w przypadku problemów geometrycznych istnieje jeszcze jedna podstawowa technika: **metoda miotły i zamiatania**.
- **Zamiatanie:** Zaczynamy od posortowania punktów zgodnie z jedną z ich współrzędnych, np. współrzędną x. Następnie przeglądamy punkty, przesuwając pionową prostą (tzw. **miotłę**) od lewej do prawej. W miotle pamiętamy informację o obiektach ją przecinających.
 - Przykład: konstrukcja algorytmu sprawdzającego, czy w zbiorze prostokątów z bokami równoległymi do osi układu dowolne dwa prostokąty się pokrywają
- **Zamiatanie polarne (biegunowe):** Najpierw wybieramy jeden z punktów i porządkujemy resztę obiektów zgodnie z ich współrzędną polarną (kątową) względem tego punktu. Następnie przeglądamy punkty zgodnie z ich uporządkowaniem.
 - Przykład: konstrukcja algorytmu znajdowania otoczki wypukłej zbioru punktów.

Problem wypukłej otoczki

- **Otoczka wypukła** $O(S)$ skońzonego zbioru punktów S to najmniejszy wypukły wielokąt P zawierający punkty zbioru S .
- W problemie otoczki wypukłej mamy dany zbiór punktów i chcemy wyznaczyć wierzchołki otoczki wypukłej w kolejności ich występowania na jej obwodzie.



$$S = \{P_0, P_1, P_2, P_3, P_4, P_5, P_6, P_7, P_8, P_9, P_{10}, P_{11}, P_{12}\}$$

$$O(S) = \{P_0, P_1, P_3, P_{10}, P_{12}\}$$

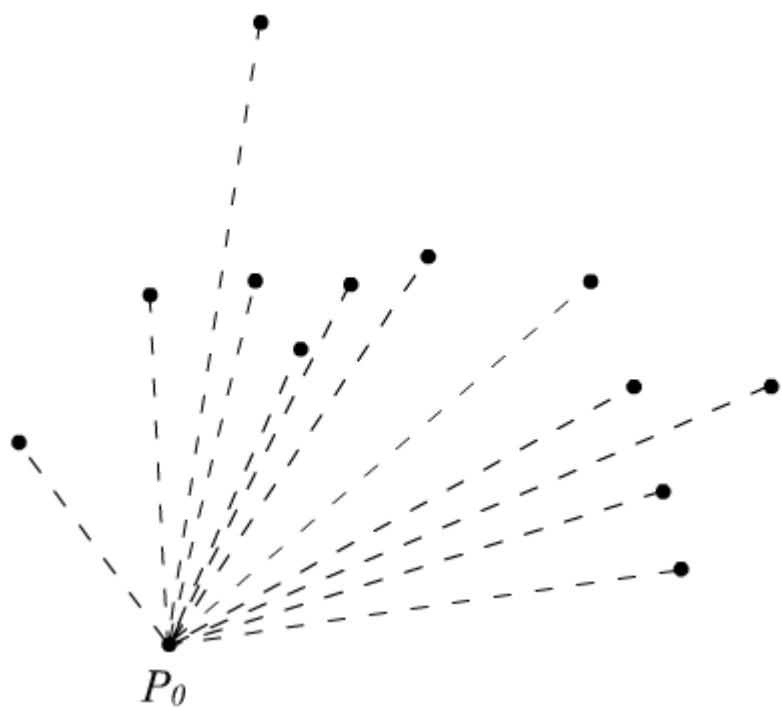
Algorytm Grahama - idea

- **Idea algorytmu:** W algorytmie Grahama problem wypukłej otoczki jest rozwiązywany z użyciem stosu S , który zawiera kandydatów na wierzchołki otoczki. Każdy punkt z wejściowego zbioru Q jest raz wkładany na stos, natomiast punkty nie będące wierzchołkami otoczki są ze stosu zdejmowane. W momencie zakończenia działania algorytmu stos S zawiera punkty występujące na otoczce w kolejności odwrotnej do ruchu wskazówek zegara.
- Danymi wejściowymi do procedury GRAHAM jest co najmniej 3-elementowy zbiór punktów Q .
- Procedura ta używa funkcji:
 - $\text{top}(S)$ zwracającej wierzchołek stosu S ,
 - $\text{nextToTop}(S)$ zwracającej drugi wierzchołek na stosie,
 - $\text{pop}(S)$ zdejmującej ze stosu element znajdujący się na szczycie stosu,
 - $\text{push}(p, S)$ kładącej wierzchołek p na szczyt stosu .
- Algorytm wymaga na wstępie wybranie ze zbioru Q punktu „startowego” oraz posortowaniu pozostałych punktów względem ich współrzędnej polarnej (biegunowej).

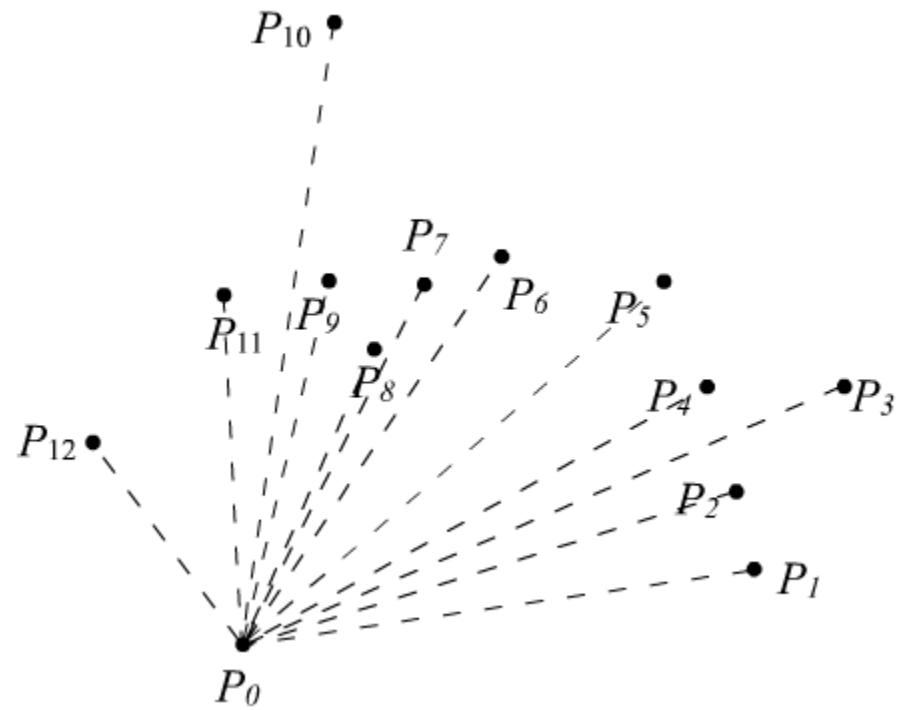
Algorytm Grahama

```
Graham(Q)
{ 1} niech  $p_0$  będzie punktem w Q o najmniejszej współrzędnej y; jeżeli
     jest kilka takich punktów, to tym najbardziej na lewo spośród nich.
{ 2} posortować pozostałe punkty ze zbioru Q rosnąco względem ich
     współrzędnych polarnych, w odniesieniu do  $p_0$ . Jeśli dwa albo więcej
     mają taką wspólną kątową, zostaw tylko ten najdalszy od  $p_0$ .
     Niech  $\{p_1, p_2, \dots, p_m\}$  ( $m \leq n$ ) będzie tym posortowanym ciągiem.
{ 3} Push( $p_0, S$ )
{ 4} Push( $p_1, S$ )
{ 5} Push( $p_2, S$ )
{ 6} for  $i=3$  to  $m$  do
{ 7}   while kąt utworzony przez punkty  $\text{NextToTop}(S), \text{Top}(S)$  i  $p_i$ 
{ 8}     nie oznacza skrętu w lewo do
{ 9}     Pop(S)
{10}    Push(S,  $p_i$ )
{11} return S
```

Przykład 1/6

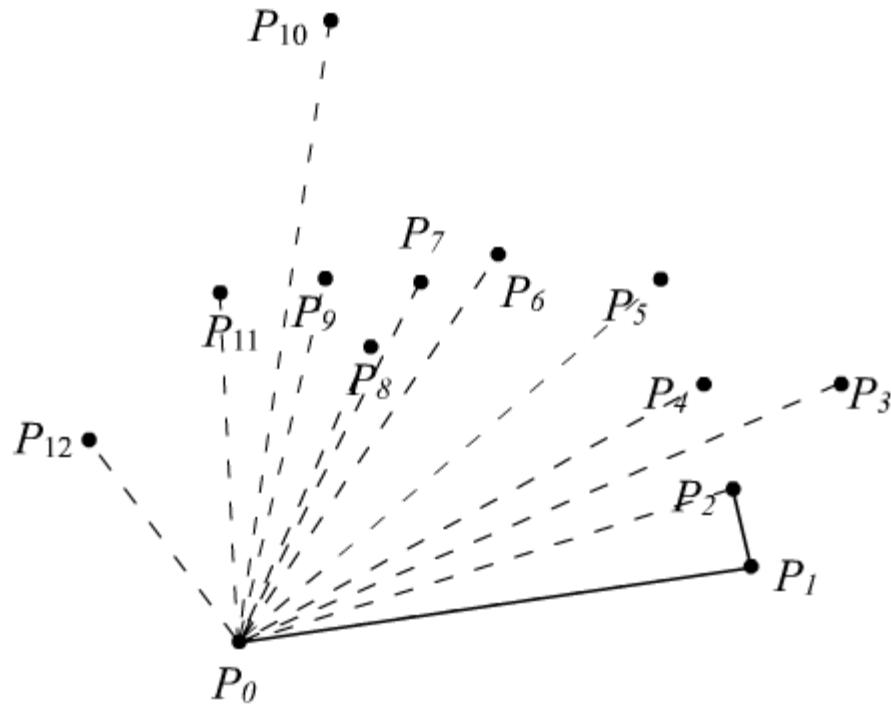


Wybieramy punkt P_0 jako leżący najniżej.
Jeśli jest więcej takich punktów,
wybieramy ten najbardziej z lewej strony.

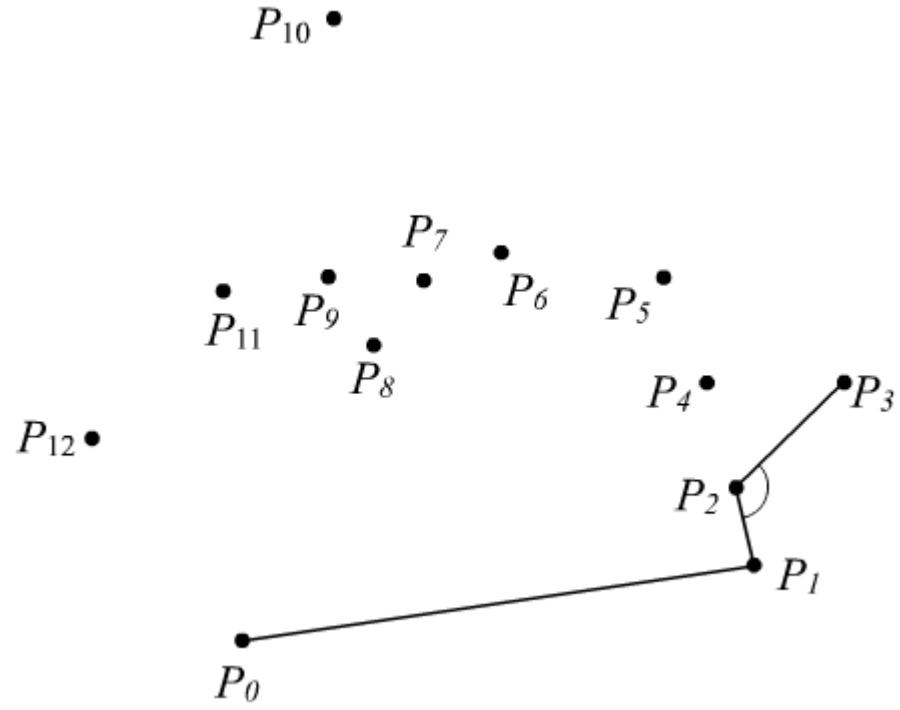


Sortujemy punkty w porządku rosnącej
współrzędnej polarnej (w jakim będą
przetwarzane).

Przykład 2/6

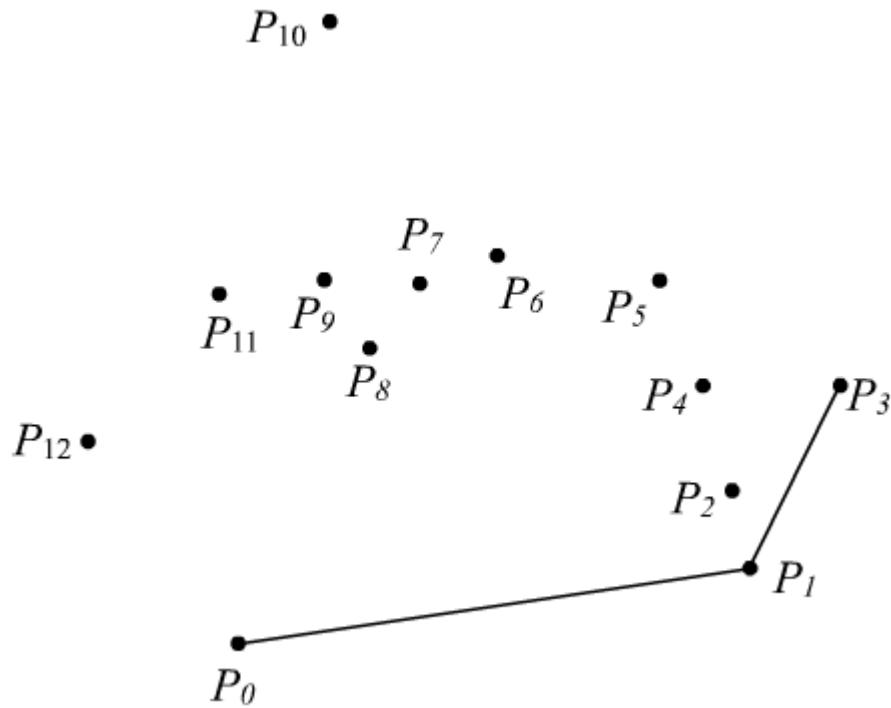


Konstruujemy otoczkę wypukłą złożoną z trzech pierwszych punktów: $\{P_0, P_1, P_2\}$. W kolejnych krokach będziemy konstruować otoczkę wypukłą dla coraz większej liczby punktów.

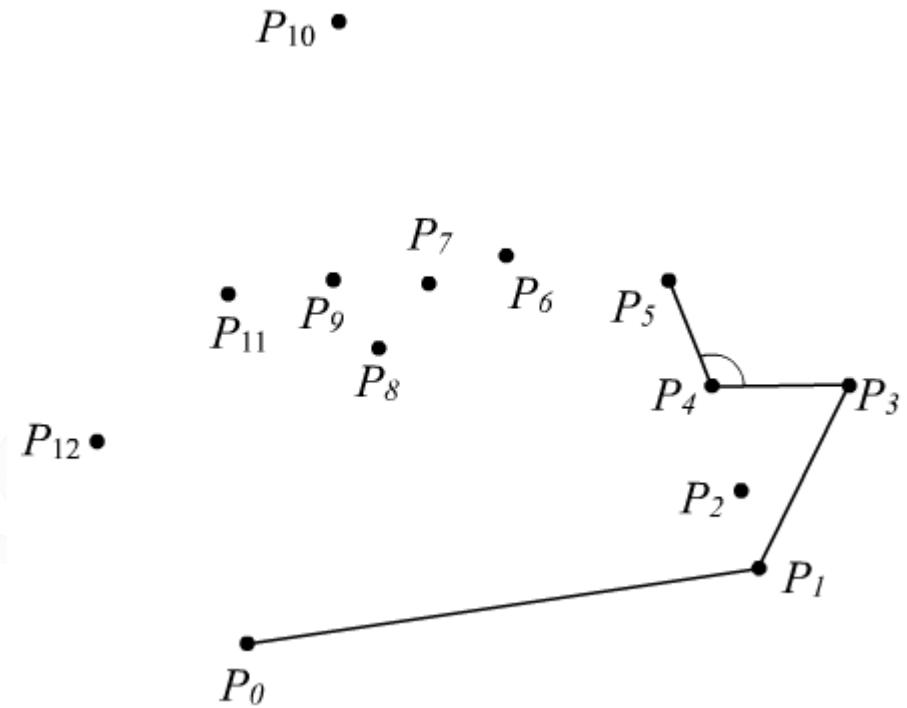


Dodajemy połączenie otoczki z kolejnym punktem (P_3). Otoczka $\{P_0, P_1, P_2, P_3\}$ przestała być wypukła (punkt P_3 leży na lewo od wektora $P_2 \rightarrow P_1$).

Przykład 3/6

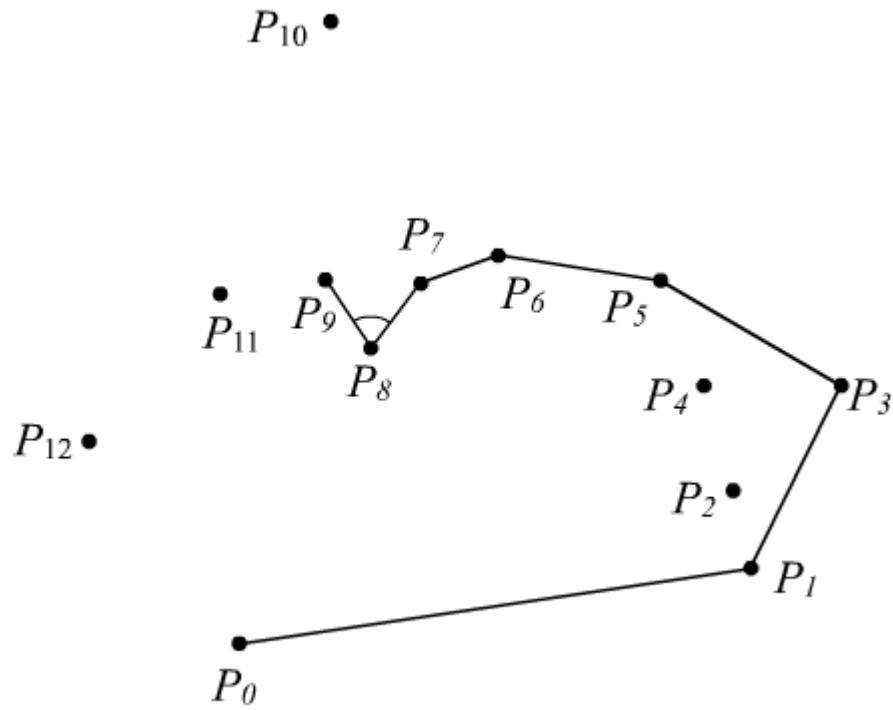


Usuwany punkt P_2 z otoczki (zastępujemy kąt wklęsty odcinkiem P_1-P_3).

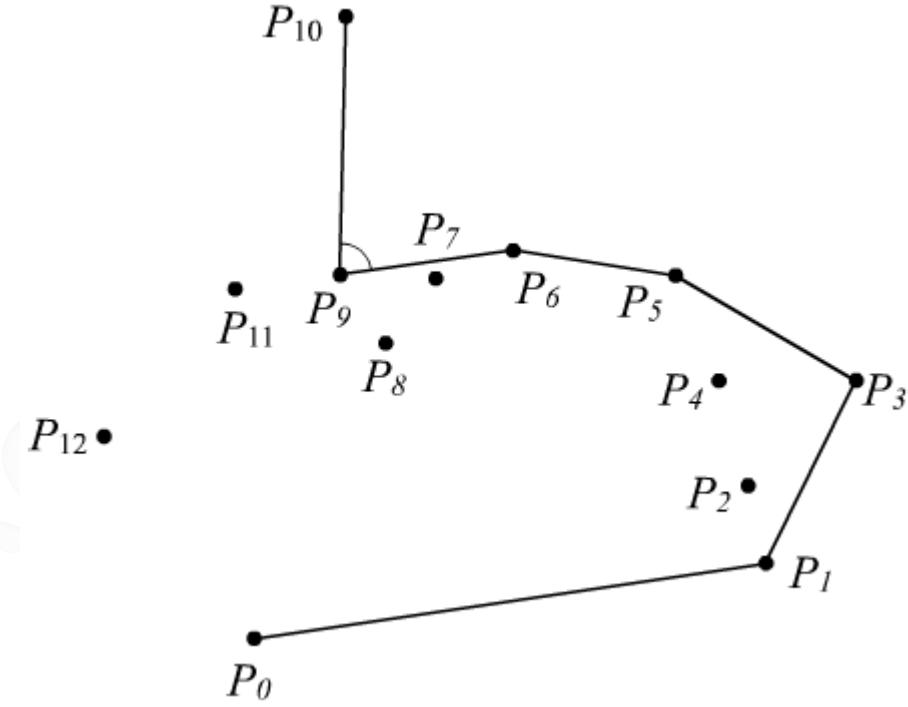


Po kolejnych krokach...

Przykład 4/6

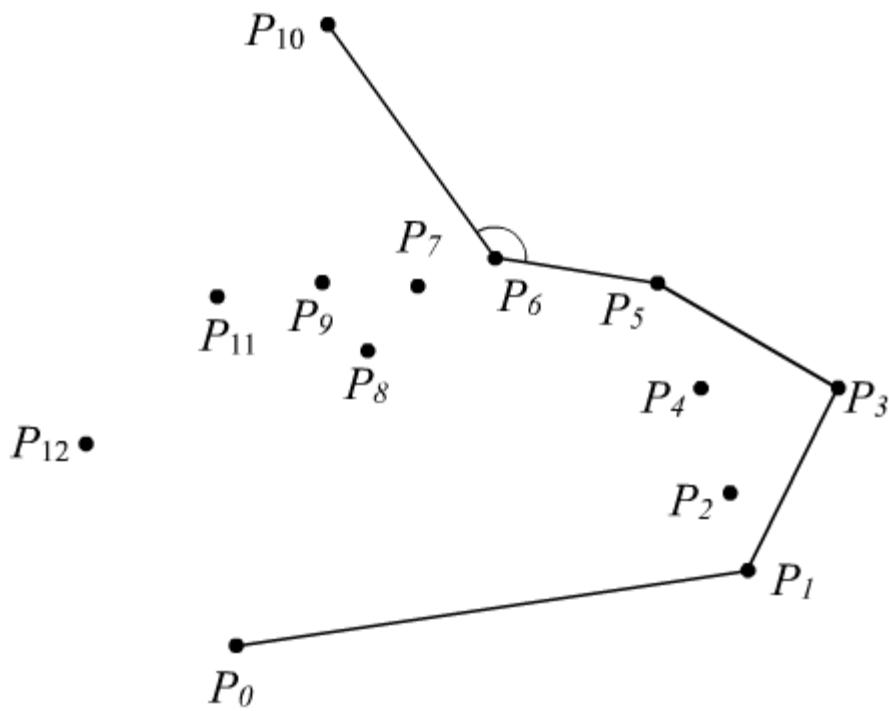


Po kilku kolejnych krokach...

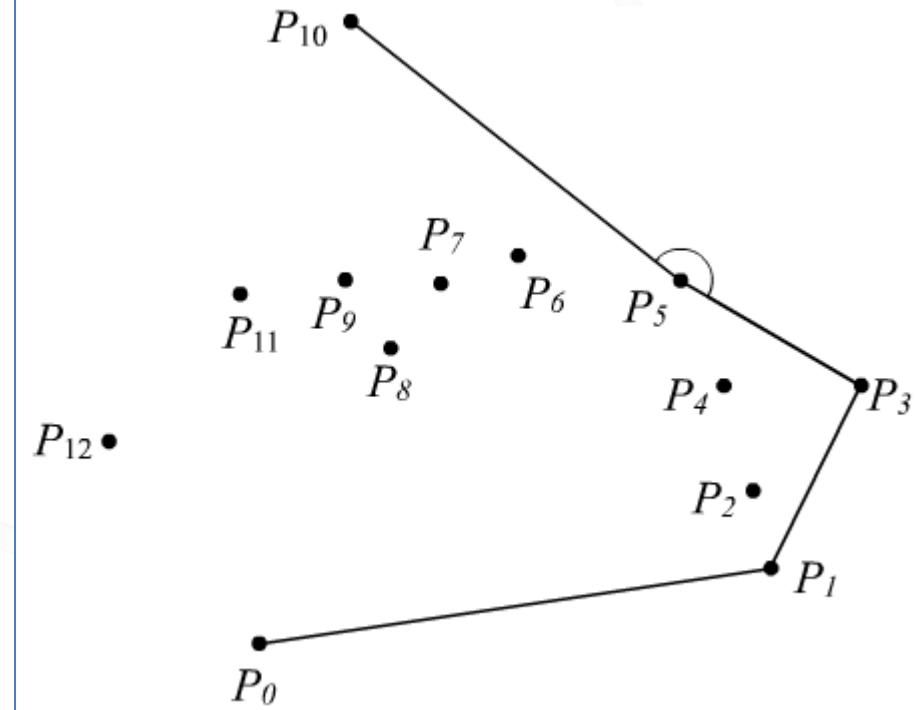


Po kilku kolejnych krokach...

Przykład 5/6



Po kilku kolejnych krokach...

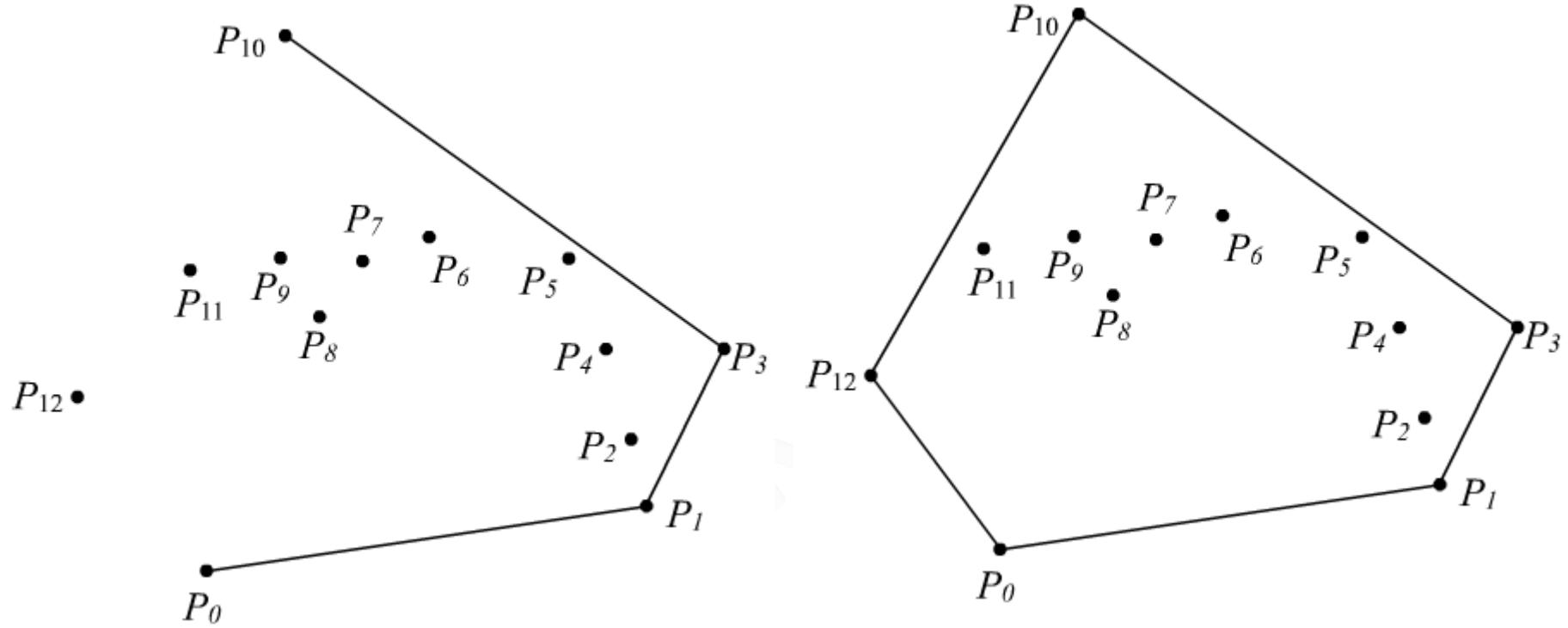


Po kilku kolejnych krokach...

Otoczka = $\{P_0, P_1, P_3, P_5\}$

Sprawdzanie punktu P_5 , który zostanie odrzucony, natomiast P_{10} dodany

Przykład 6/6



Po poprzednim kroku:

Otoczka = $\{P_0, P_1, P_3, P_{10}\}$

Po kilku kolejnych krokach finalnie:

$$\text{Otoczka} = \{P_0, P_1, P_3, P_{10}, P_{12}\}$$

Algorytm Grahama - analiza

- Złożoność kroku 2, w którym sortuje się elementy wynosi $O(n \log n)$
- Liczbę wykonań pętli **for** równa się n
- Całkowita liczba wykonań pętli **while** nie może przekroczyć $O(n)$, bo zdjąć element ze stosu można po wcześniejszym włożeniu go na stos
 - Można udowodnić, że zawsze na stosie będą co najmniej 2 pierwsze elementy (p_0 i p_1)
- W jednym wykonaniu pętli **for** wykonamy jedną operację Push.
- Wniosek: całkowita złożoność algorytmu Grahama wynosi $O(n \log n)$.

Algorytmy i struktury danych – W15

Problemy
P, NP, NP-zupełne, NP-trudne

Zawartość

- Problemy nierzestrzygalne
- Problemy optymalizacyjne a decyzyjne
- Notacja formalna dla klas problemów
 - Za pomocą teorii języków
- Klasy problemów
 - Klasa P
 - Klasa NP
 - Klasa NP-trudne
 - Klasa NPC (problemy NP-zupełne)
- Przykłady problemów NP-zupełnych
- Techniki rozwiązywania problemów NP-zupełnych (i nie tylko)

Nierozstrzygalne problemy

- **Problem stopu** jest problemem decyzyjnym dotyczącym właściwości programów komputerowych na ustalonym modelu obliczeń Turinga.
- W tej abstrakcyjnej strukturze nie ma ograniczeń zasobów pamięci ani czasu na wykonanie programu; może to potrwać dowolnie długo i wykorzystać dowolnie dużo miejsca do przechowywania, zanim się zatrzyma.
- Pytanie brzmi, biorąc pod uwagę program i dane wejściowe do programu, czy program ostatecznie zatrzyma się, gdy uruchomi się z tymi danymi wejściowymi.
- Problem stop był jednym z pierwszych problemów, które okazały się **nierozstrzygalne**, co oznacza, że nie ma programu komputerowego zdolnego do prawidłowego udzielenia odpowiedzi na pytanie o wszystkie możliwe dane wejściowe (czyli dowolny program komputerowy i dowolne dopuszczalne dane wejściowe).
- Nie potrafimy nawet rozstrzygnąć to dla „prostego” problemu Collatza (inaczej zwanego „problemem $3n+1$ ”)

Problem Collatza

- Pytanie: czy istnieje taka wartość całkowita dodatnia dla której poniższy program się **nie** zatrzyma?

```
Collatz(n) {  
    while(n>1) {  
        if(n mod 2 == 0)  
            n:=n/2;  
        else  
            n:=3*n+1;  
    }  
}
```

- Przykładowa sekwencja wartości zaczynając od n=27:
 - 27, 82, 41, 124, 62, 31, 94, 47, 142, 71, 214, 107, 322, 161, 484, 242, 121, 364, 182, 91, 274, 137, 412, 206, 103, 310, 155, 466, 233, 700, 350, 175, 526, 263, 790, 395, 1186, 593, 1780, 890, 445, 1336, 668, 334, 167, 502, 251, 754, 377, 1132, 566, 283, 850, 425, 1276, 638, 319, 958, 479, 1438, 719, 2158, 1079, 3238, 1619, 4858, 2429, 7288, 3644, 1822, 911, 2734, 1367, 4102, 2051, 6154, 3077, 9232, 4616, 2308, 1154, 577, 1732, 866, 433, 1300, 650, 325, 976, 488, 244, 122, 61, 184, 92, 46, 23, 70, 35, 106, 53, 160, 80, 40, 20, 10, 5, 16, 8, 4, 2, 1

Problemy optymalizacyjne a decyzyjne

- **Problemy optymalizacyjne:** każde dopuszczalne rozwiązanie ma przypisaną wartość i chcemy znaleźć dopuszczalne rozwiązanie o najlepszej wartości
 - Problem SHORTEST-PATH: mamy dany nieskierowany graf G oraz wierzchołki u i v , i chcemy znaleźć ścieżkę z u do v z użyciem najmniejszej liczby krawędzi
- **Problemy decyzyjne:** odpowiedź brzmi po prostu "tak" lub "nie" (1 lub 0). NP-zupełność odnosi się bezpośrednio do problemów decyzyjnych.
 - Problem PATH, mamy dany graf nieskierowany G , wierzchołki u i v oraz liczbę całkowitą k , czy istnieje ścieżka z u do v składającą się z co najwyżej k krawędzi
- Jeśli problem z optymalizacją jest łatwy, to związany z tym problem decyzyjny również jest łatwy.
- Jeśli problem decyzyjny jest trudny, jego powiązany problem optymalizacyjny również jest trudny.

Problem abstrakcyjny

- Abstrakcyjny problem Q będący relacją binarną na zbiorze I instancji problemowych i zbioru S rozwiązań problemowych.
- Na przykład:
 - instancją dla SHORTEST-PATH jest trójka składająca się z grafu i dwóch wierzchołków.
 - rozwiązaniem jest sekwencja wierzchołków w grafie, ewentualnie pusta sekwencja oznaczającą brak ścieżki.
 - problem SHORTEST-PATH jest relacją, która wiąże każdąinstancję grafu i dwóch wierzchołków z najkrótszą ścieżką w grafie, która łączy dwa wierzchołki. Ponieważ najkrótsze ścieżki niekoniecznie są unikalne, dana instancja problemu może mieć więcej niż jedno rozwiązanie.

Abstrakcyjny problem decyzyjny

- Problemy z rozwiązaniem tak/nie. W tym przypadku możemy widzieć abstrakcyjny problem decyzyjny jako funkcję odwzorowującą zestaw instancji I na zestaw rozwiązań $\{0, 1\}$.
- Na przykład:
 - problemem decyzyjnym związanym z SHORTEST-PATH jest problem PATH: jeśli $i = \langle G, u, v, k \rangle$ jest instancją problemu decyzyjnego PATH, to $\text{PATH}(i) = 1$ (tak), jeśli najkrótsza ścieżka z u do v ma najwyżej k krawędzi, a $\text{PATH}(i) = 0$ (nie) w przeciwnym razie.
- Abstrakcyjny problem to problem matematyczny. W przypadku rozwiązywania na komputerze trzeba skonkretyzować jak będziemy reprezentować problem i jego rozwiązanie.

Kodowanie

- **Kodowanie** zestawu S obiektów abstrakcyjnych to odwzorowanie e z S do **zbioru łańcuchów binarnych**.
 - kodowanie liczb naturalnych $N = \{0, 1, 2, 3, 4, \dots\}$ jako ciągów $\{0, 1, 10, 11, 100, \dots\}$
 - Nawet obiekt złożony można zakodować jako łańcuch dwójkowy, łącząc reprezentacje jego części składowych. Wieloboki, grafy, funkcje, uporządkowane pary, programy - wszystkie mogą być zakodowane jako ciągi binarne
- Unarne kodowanie - "drogie" kodowania
 - e.a $\{1, 11, 111, 1111, \dots\}$
- Inne kodowanie z inną bazą niż dwa, jest wielomianowo równoważne

Definicje

- Nazywamy problem, którego zestaw instancji jest zbiorem ciągów binarnych **konkretnym problemem**.
- Mówimy, że algorytm **rozwiązuje** konkretny problem **w czasie** $O(T(n))$, jeśli, gdy zostanie dostarczona instancja problemu o długości $n = |i|$, algorytm może wytworzyć rozwiązanie w czasie $O(T(n))$. Konkretny problem jest **rozwiązywalny w czasie wielomianowym**, jeśli istnieje algorytm do rozwiązania w czasie $O(n^k)$ dla pewnej stałej k .
- **Klasa złożoności P** to zbiór konkretnych problemów decyzyjnych, które są **rozwiązywalne wielomianowo**.
- Wszystkie dotychczas prezentowane na tym kursie problemy należą do klasy złożoności P.

Język formalny 1/2

- **Alfabet** S to skończony zestaw symboli.
- Język L nad S jest dowolnym zbiorem ciągów złożonych z symboli z S .
 - Na przykład, jeśli $S = \{0, 1\}$, zbiór $L = \{10, 11, 101, 111, 1011, 1101, 10001, \dots\}$ jest językiem binarnych reprezentacji liczb pierwszych.
- Oznaczamy pusty łańcuch przez ϵ , a pusty język przez \emptyset .
- Język wszystkich łańcuchów nad S jest oznaczony jako S^* .
 - Na przykład, jeśli $S = \{0, 1\}$, to $S^* = \{\epsilon, 0, 1, 00, 01, 10, 11, 000, \dots\}$ jest zbiorem wszystkich ciągów binarnych.
- Każdy język L nad S jest podzbiorem S^*
- ...

Język formalny 2/2

- Z punktu widzenia teorii języków zbiór instancji dowolnego problemu decyzyjnego Q to po prostu zbiór S^* , gdzie $S = \{0, 1\}$.
- Ponieważ problem Q jest w pełni określony przez te jego instancje, dla których odpowiedzią jest 1 (tak) możemy traktować Q jako język L nad $S = \{0, 1\}$, gdzie:

$$L = \{ x \in S^* : Q(x) = 1 \}$$

- Mówimy, że algorytm A **akceptuje** słowo $x \in \{0, 1\}^*$, jeśli dla danych wejściowych x algorytm oblicza $A(x) = 1$. Język **akceptowany** przez algorytm A to zbiór słów $L = \{x \in \{0, 1\}^* : A(x) = 1\}$, czyli zbiór tych słów, które algorytm akceptuje.
- Algorytm A **odrzuca** słowo x , jeśli $A(x) = 0$

Akceptowanie/rozstrzyganie języka

- Nawet jeśli język L jest akceptowany przez algorytm A , algorytm nie musi koniecznie odrzucać danego mu na wejściu słowa $x \notin L$. Algorytm może się na przykład zapętlić.
- Język L jest **rozstrzygalny** przez algorytm A jeśli każdy ciąg binarny jest albo akceptowany, albo odrzucany.
- Język L jest **akceptowalny w czasie wielomianowym** przez algorytm A , jeśli dla dowolnego słowa $x \in L$ o długości n algorytm akceptuje x w czasie $O(n^k)$ dla pewnej stałej k .
- Język L jest **rozstrzygalny w czasie wielomianowym** przez algorytm A , jeśli dla dowolnego słowa $x \in \{0, 1\}^*$ o długości n algorytm rozstrzyga przynależność x do L w czasie $O(n^k)$ dla pewnej stałej k .
- Alternatywna definicja klasy złożoności P :
$$P = \{L \subseteq \{0, 1\}^* : \text{istnieje algorytm } A \text{ rozstrzygający o } L \text{ w czasie wielomianowym}\}$$

Algorytm weryfikacji

- **Algorytm weryfikacyjny** definiujemy jako algorytm A o dwóch parametrach, z których jeden jest zwykłym ciągiem x , a drugi to ciąg binarny y , zwany **świadectwem (certyfikatem)**
- Dwuparametrowy algorytm A **weryfikuje** ciąg wejściowy x , jeśli istnieje świadectwo y tak, że $A(x,y)=1$. **Język weryfikowany** przez algorytm A to:
$$L = \{x \in \{0, 1\}^* : \text{istnieje } y \in \{0, 1\}^* \text{ takie, że } A(x, y) = 1\}.$$
- Algorytm A **weryfikuje** język L , jeśli dla każdego słowa $x \in L$, istnieje świadectwo y , którego A może użyć w celu wykazania, że $x \in L$.
 - Cyklem Hamiltona dla nieskierowanego grafu $G=(V, E)$ jest cyk prosty, który przechodzi przez każdy wierzchołek. Graf jest **hamiltonowski**, jeśli posiada taki cykl, lub **nie jest hamiltonowski** w p.p.
 - W problemie cyklu Hamiltona świadectwem jest lista wierzchołków cyklu. Algorytm weryfikacji zbada cykl i sprawdzi, czy wszystkie krawędzie istnieją i czy przechodzi przez wszystkie wierzchołki. Jeśli graf jest hamiltonowski, to można taką listę znaleźć. Przeciwnie, jeśli graf nie jest hamiltonowski, to żaden taki ciąg nie oszuka algorytmu weryfikującego.

Klasa złożoności NP

- **Klasa złożoności NP** to klasa języków, które można weryfikować za pomocą algorytmu wielomianowego.
- Język L należy do klasy NP wtedy i tylko wtedy, gdy istnieje dwuparametryowy wielomianowy algorytm A i stała c takie, że

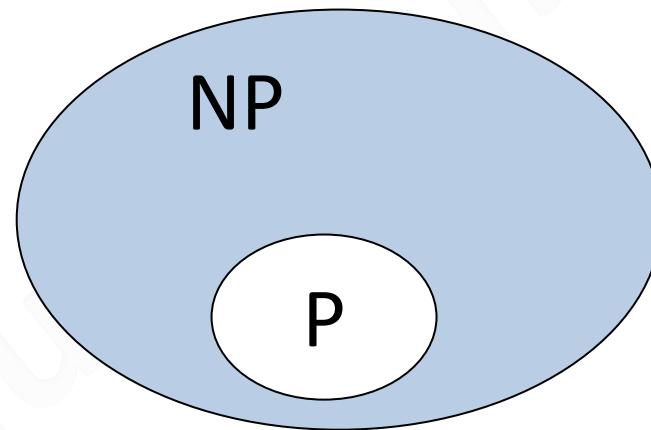
$$L = \{x \in \{0, 1\}^*: \text{istnieje świadectwo } y, \\ \text{gdzie } |y| = O(|x|^c), \text{ że } A(x, y) = 1\}.$$

- Mówimy, że algorytm A **weryfikuje język L w czasie wielomianowym**

Pytanie

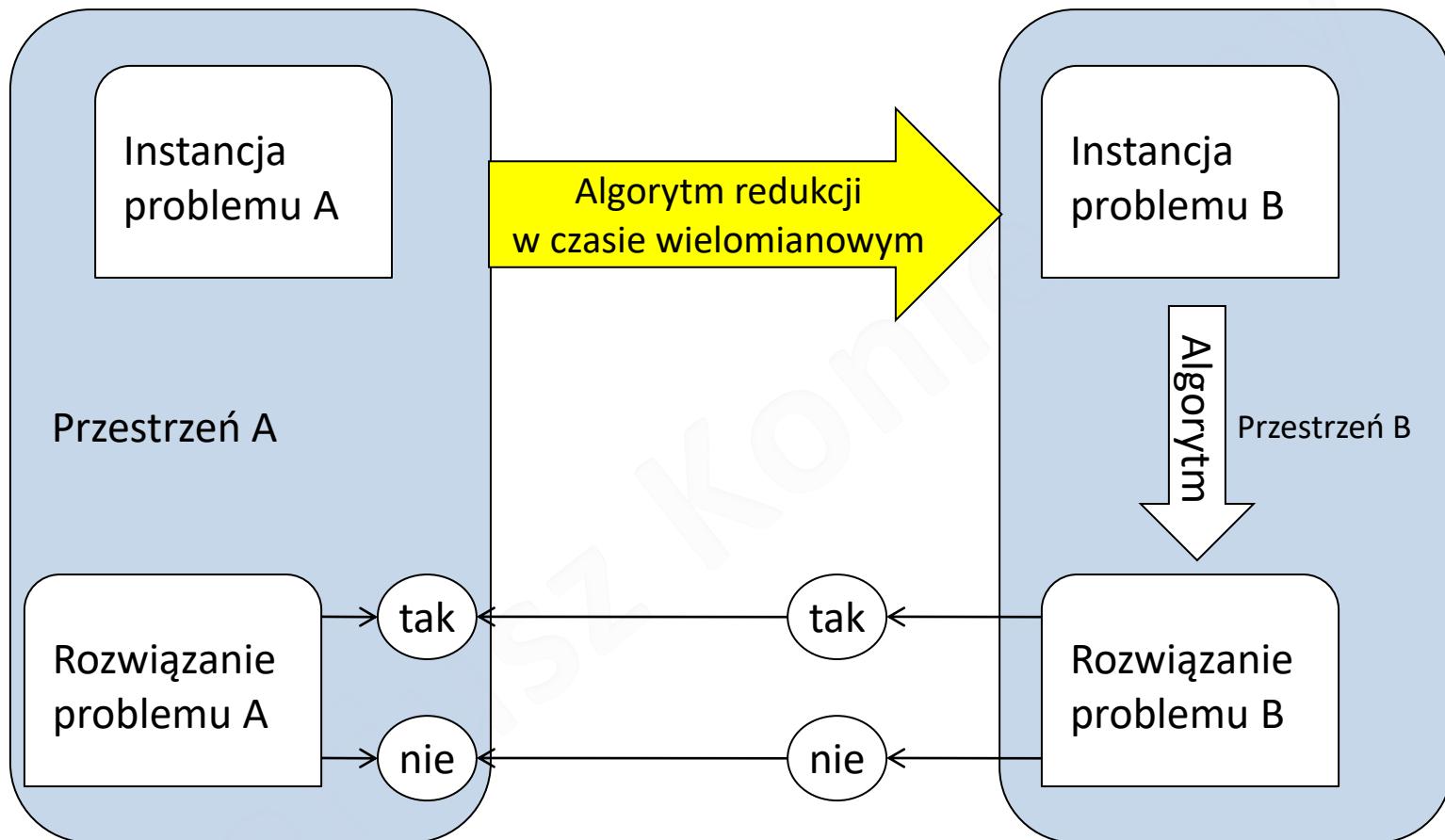
P = NP ?

(Prawdopodobnie $P \neq NP$)



Milion dolarów nagrody za rozwiązywanie tej kwestii!

Redukowalność 1/2



Redukowalność 2/2

- Język L_1 jest **redukowalny w czasie wielomianowym** do języka L_2 , co zapiszemy jako $L_1 \leq_p L_2$, jeśli istnieje obliczalna w czasie wielomianowym funkcja $f: \{0,1\}^* \rightarrow \{0,1\}^*$ taka, że dla każdego $x \in \{0,1\}^*$,
$$x \in L_1 \text{ wtedy i tylko wtedy, gdy } f(x) \in L_2$$
- Funkcję f nazywamy **funkcją redukcji**, a obliczający ją wielomianowy algorytm F – **algorytmem redukcji**.

Lemat

Jeśli $L_1, L_2 \in \{0,1\}^*$ są językami takimi, że $L_1 \leq_p L_2$, to z tego, że $L_2 \in P$ wynika, że $L_1 \in P$

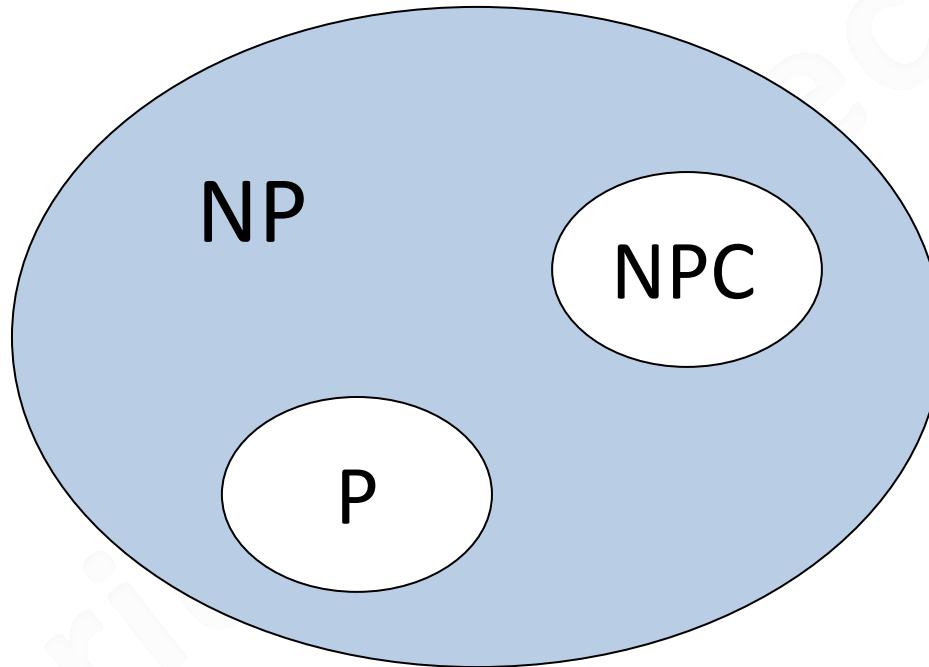
NP-zupełność

- Język $L \in \{0, 1\}^*$ jest **NP-zupełny** jeśli
 1. $L \in \text{NP}$, oraz
 2. $L' \leq_P L$ dla każdego $L' \in \text{NP}$.
- Jeśli język L spełnia własność 2, ale niekoniecznie własność 1, mówimy, że L jest **NP-trudny**.
- Definiujemy NPC jako klasę języków NP-zupełnych.

Twierdzenie

Jeśli jakikolwiek problem NP-zupełny jest rozwiązywalny w czasie wielomianowym, to $P = \text{NP}$. Jeśli jakikolwiek problem w NP nie jest rozwiązywalny w czasie wielomianowym, to żaden problem NP-zupełny nie jest rozwiązywalny w czasie wielomianowym.

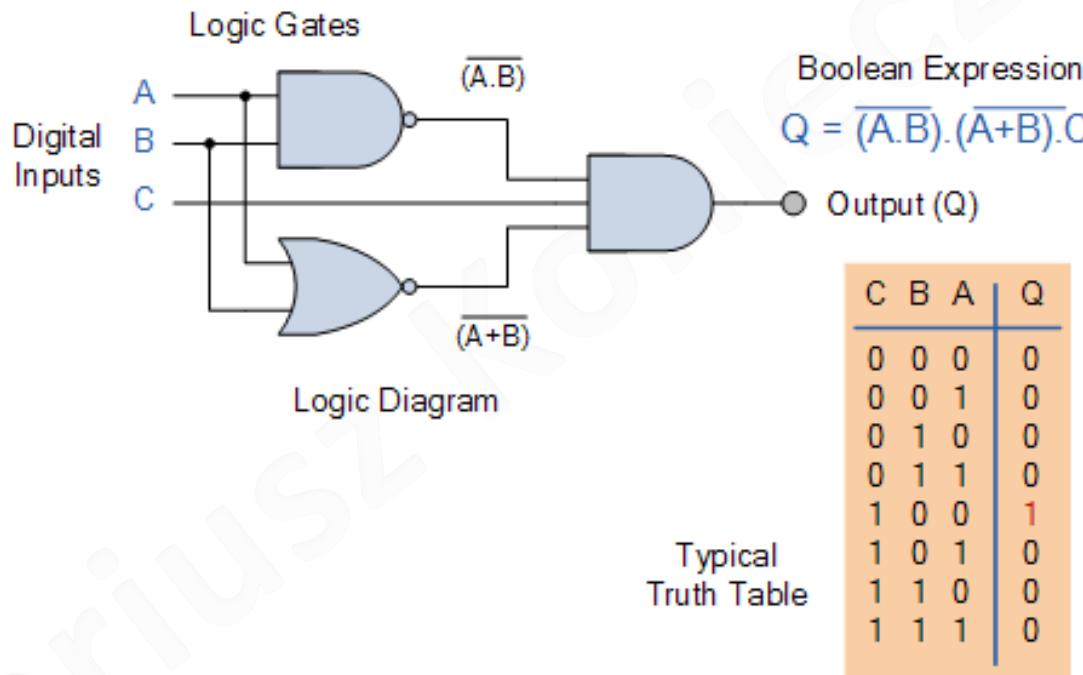
Prawdopodobna relacja



Problemy NP-zupełne 1/8

- Spełnialność układów logicznych

- Czy dany układ logiczny, zbudowany z n wejść i z bramek AND, OR i NOT jest spełnialny?
- W książce Cormena i in. – dowód, że ten problem należy do klasy NP-zupełnej. Kolejne problemy dzięki redukowalności można do niego sprowadzić.

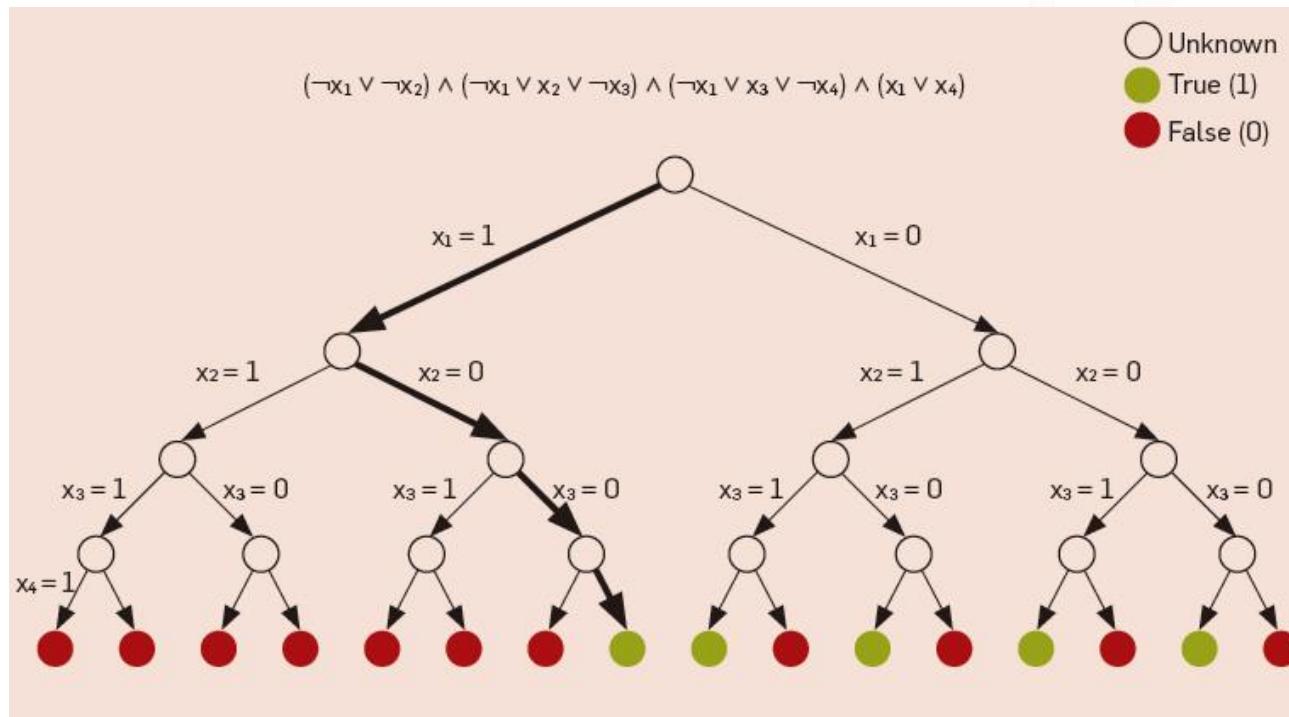


https://www.electronics-tutorials.ws/combination/comb_1.html

Problemy NP-zupełne 2/8

- Spełnialność formuł (*ang. problem SAT*)

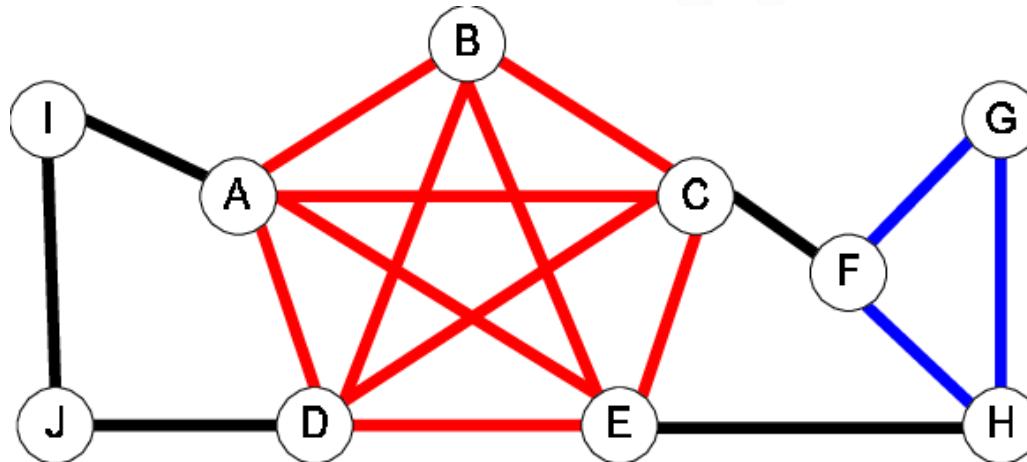
- Czy formuła logiczna składająca się ze zmiennych logicznych, spójników logicznych i nawiasów jest spełnialna?
- To pierwszy problem, którego NP-zupełność udowodniono (Stephen Cook w 1971 roku)



<https://cacm.acm.org/magazines/2009/8/34498-boolean-satisfiability-from-theoretical-hardness-to-practical-success/fulltext>

Problemy NP-zupełne 3/8

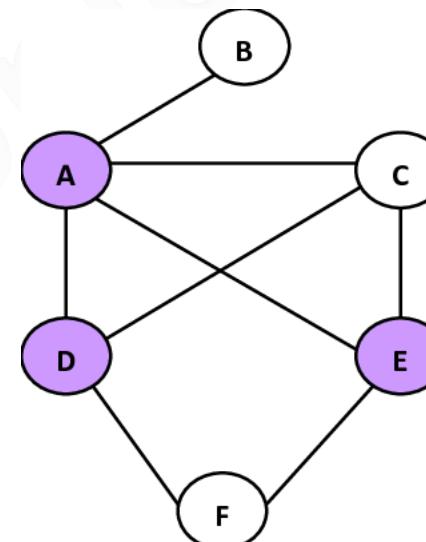
- Problem kliki
 - Kliką w grafie nieskierowanym $G = (V, E)$ jest podzbiór wierzchołków $V' \subseteq V$, taki, że każda para wierzchołków jest połączona krawędzią z E . **Rozmiar** kliki to liczba wierzchołków, które zawiera. **Problem kliki** to optymalizacyjny problem znalezienia w grafie kliki maksymalnego rozmiaru. W wersji decyzyjnej pytamy, czy istnieje w grafie klika danego rozmiaru k .



https://www.researchgate.net/figure/Graph-containing-a-clique-of-size-3-FGH-and-a-maximum-clique-of-size-5-ABCDE_fig2_221429651

Problemy NP-zupełne 4/8

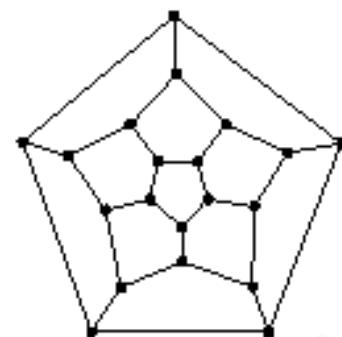
- Problem pokrycia wierzchołkowego
 - **Pokrycie wierzchołkowe** grafu nieskierowanego $G = (V, E)$ to podzbiór $V' \subseteq V$ taki, że jeśli $(u, v) \in E$, to $u \in V'$ lub $v \in V'$ (lub obydwa). Inaczej mówiąc, każdy wierzchołek „pokrywa” incydentne z nim krawędzie, a pokryciem wierzchołkowym G jest zbiór wierzchołków pokrywając wszystkie krawędzie zbioru E . **Rozmiar** pokrycia wierzchołkowego to liczba składających się na nie wierzchołków. Problem pokrycia wierzchołkowego polega na znalezieniu w danym grafie pokrycia wierzchołkowego minimalnego rozmiaru.



https://www.researchgate.net/figure/An-undirected-graph-G-the-shadowed-nodes-represent-the-vertex-cover_fig1_263008838

Problemy NP-zupełne 5/8

- Problem cyklu Hamiltona – cykl, w którym każdy węzeł odwiedzamy dokładnie raz



INPUT

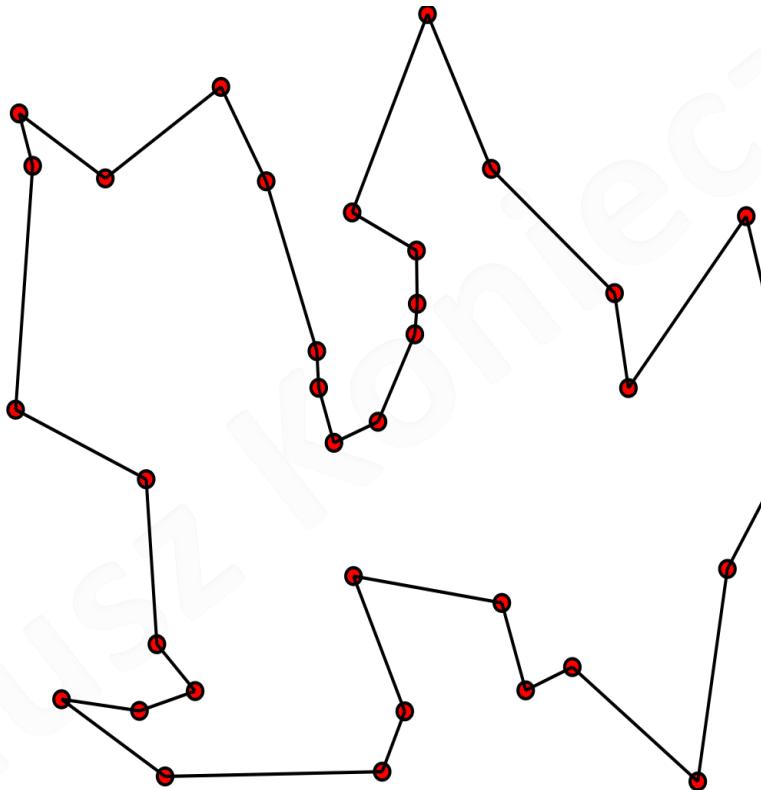


OUTPUT

[https://www8.cs.umu.se/kurser/TDBA77/VT06
/algorithms/BOOK/BOOK4/NODE176.HTM](https://www8.cs.umu.se/kurser/TDBA77/VT06/algorithms/BOOK/BOOK4/NODE176.HTM)

Problemy NP-zupełne 6/8

- Problem komiwojażera (podobny do cyklu Hamiltona, ale na grafie ważonym znaleźć cykl o minimalnej wadze)

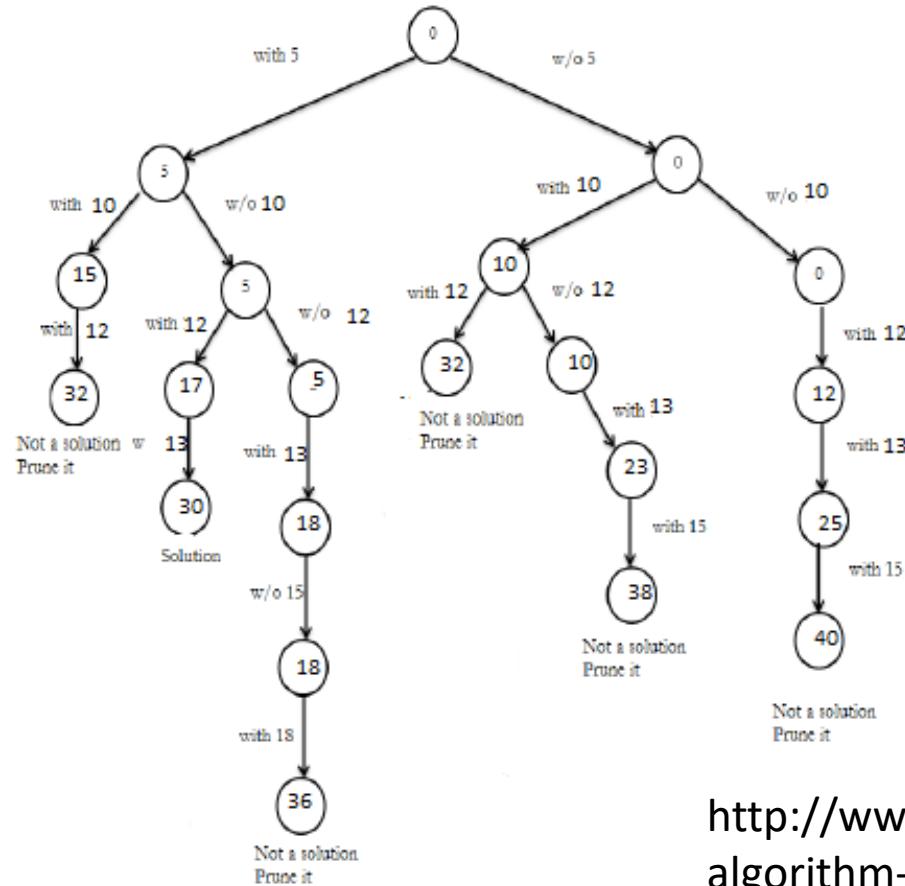


https://en.wikipedia.org/wiki/Travelling_salesman_problem

Problemy NP-zupełne 7/8

- Problem sumy podzbioru

- W **problemie sumy podzbioru** mamy dany zbiór skończony $S \subseteq N$ i wartość docelową $t \in N$. Pytamy, czy istnieje podzbiór $S' \subseteq S$, którego suma elementów wynosi t .



$$S = \{5, 10, 12, 13, 15, 18\}$$
$$t = 30$$

<http://www.ques10.com/p/17687/write-an-algorithm-for-sum-of-subsets-solve-the-fo/>

Problemy NP-zupełne 8/8

- Istnieje wiele innych problemów należących do klasy NP-zupełnych.
- Źródła do rozpoczęcia poszukiwań:
 - Wikipedia List of NP-complete problems
https://en.wikipedia.org/wiki/List_of_NP-complete_problems
 - A compendium of NP optimization problems:
<http://www.nada.kth.se/~viggo/wwwcompendium/wwwcompendium.html>
 - Graph of NP-Complete Problems:
<https://adriann.github.io/npc/npc.html>

Podział algorytmów

- Efektywne:
 - Algorytmy o złożoności wielomianowej
- Nieefektywne:
 - Algorytmy o złożoności ponad-wielowymiarowej

Rozwiązywanie problemów NP-zupełnych

- Jak dotąd wszystkie znane algorytmy dla problemów NP-zupełnym wymagają **ponadwielomianowego** czasu i nie wiadomo, czy kiedykolwiek zostaną znalezione lepsze.
- Poniższe techniki mogą być stosowane do rozwiązywania problemów obliczeniowych w ogóle, i często powodują powstanie znacznie szybszych algorytmów:
 - **Aproksymacja:** Zamiast szukać optymalnego rozwiązania, szukaj "prawie" optymalnego rozwiązania.
 - **Losowość:** Użyj losowości, aby uzyskać szybszy średni czas pracy, i pozwól algorytmowi zawodzić z niewielkim prawdopodobieństwem.
 - **Ograniczenie:** Ograniczając strukturę danych wejściowych (np. do grafów planarnych), zwykle możliwe są szybsze algorytmy.
 - **Parametryzacja:** Często istnieją szybkie algorytmy, jeśli pewne parametry wejścia są ustalone.
 - **Heurystyczny:** algorytm, który działa "dość dobrze" w wielu przypadkach, ale dla którego nie ma dowodu, że jest on zawsze szybki i zawsze daje dobry wynik.