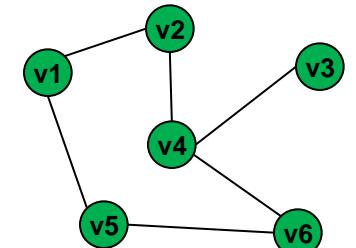
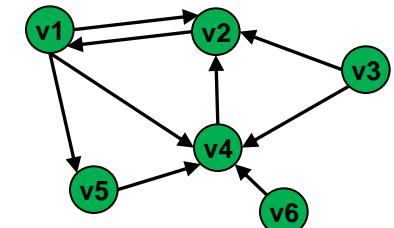

Podstawy implementacji grafów

Graf (przypomnienie)

- **Grafem G** nazywamy uporządkowaną parę $G=(V,E)$ składającą się z niepustego, skończonego zbioru **wierzchołków (węzłów)** V oraz zbioru **krawędzi** E czyli par wierzchołków ze zbioru V .
- Rozpatrywane są dwa następujące rodzaje grafów:
 - » **graf skierowany (zorientowany, digraf)**, gdy E jest podzbiorem zbioru par uporządkowanych ze zbioru V , tzn. $E \subseteq V \times V$
 - » **graf nieskierowany (niezorientowany)**, gdy E jest podzbiorem zbioru wszystkich dwuelementowych podzbiorów zbioru V
 - » **Implementacja grafu nieskierowanego nie będzie wymagana (jest bardzo podobna)**



Metody reprezentacji grafów

(przypomnienie)

- Nie będzie dany graf $G=(V,E)$ taki, że $n=|V|$ i $m=|E|$
- **Lista sąsiedztwa**
 - » Dla każdego wierzchołka $x \in V$ budujemy listę (oznaczaną przez $L[x]$) wierzchołków y będących sąsiadami X , tzn. istnieje para uporządkowana $(x,y) \in E$ (dla grafów skierowanych) lub istnieje zbiór $\{x,y\} \in E$ (dla grafów nieskierowanych)
- **Macierz sąsiedztwa**
 - » Budujemy macierz sąsiedztwa A o rozmiarze $n \times n$:

$$A[x, y] = \begin{cases} 1, & \text{dla } (x, y) \in E \\ 0, & \text{dla } (x, y) \notin E \end{cases}$$

Przykład: Lista i macierz sąsiedztwa dla grafu skierowanego

- Lista sąsiedztwa:

1: 2, 3

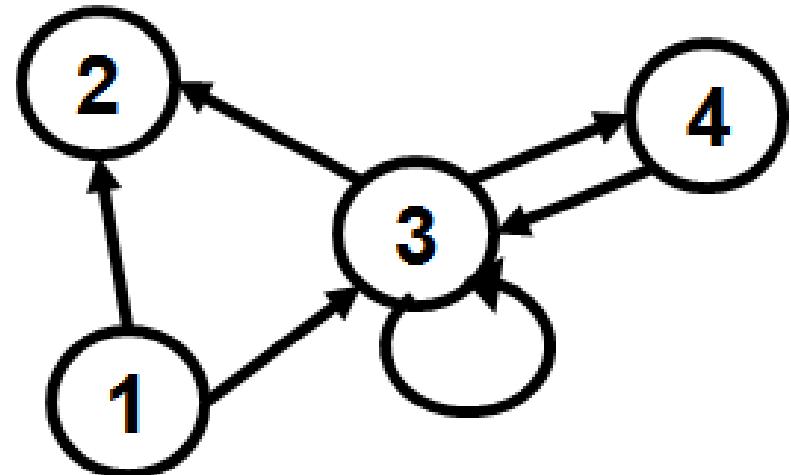
2:

3: 3, 2, 4

4: 3

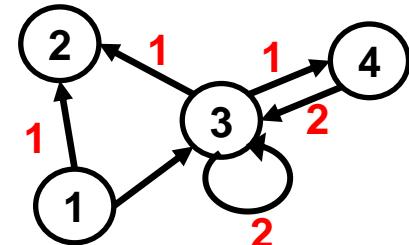
- Macierz sąsiedztwa:

	1	2	3	4
1	0	1	1	0
2	0	0	0	0
3	0	1	1	1
4	0	0	1	0



Wagi połączeń

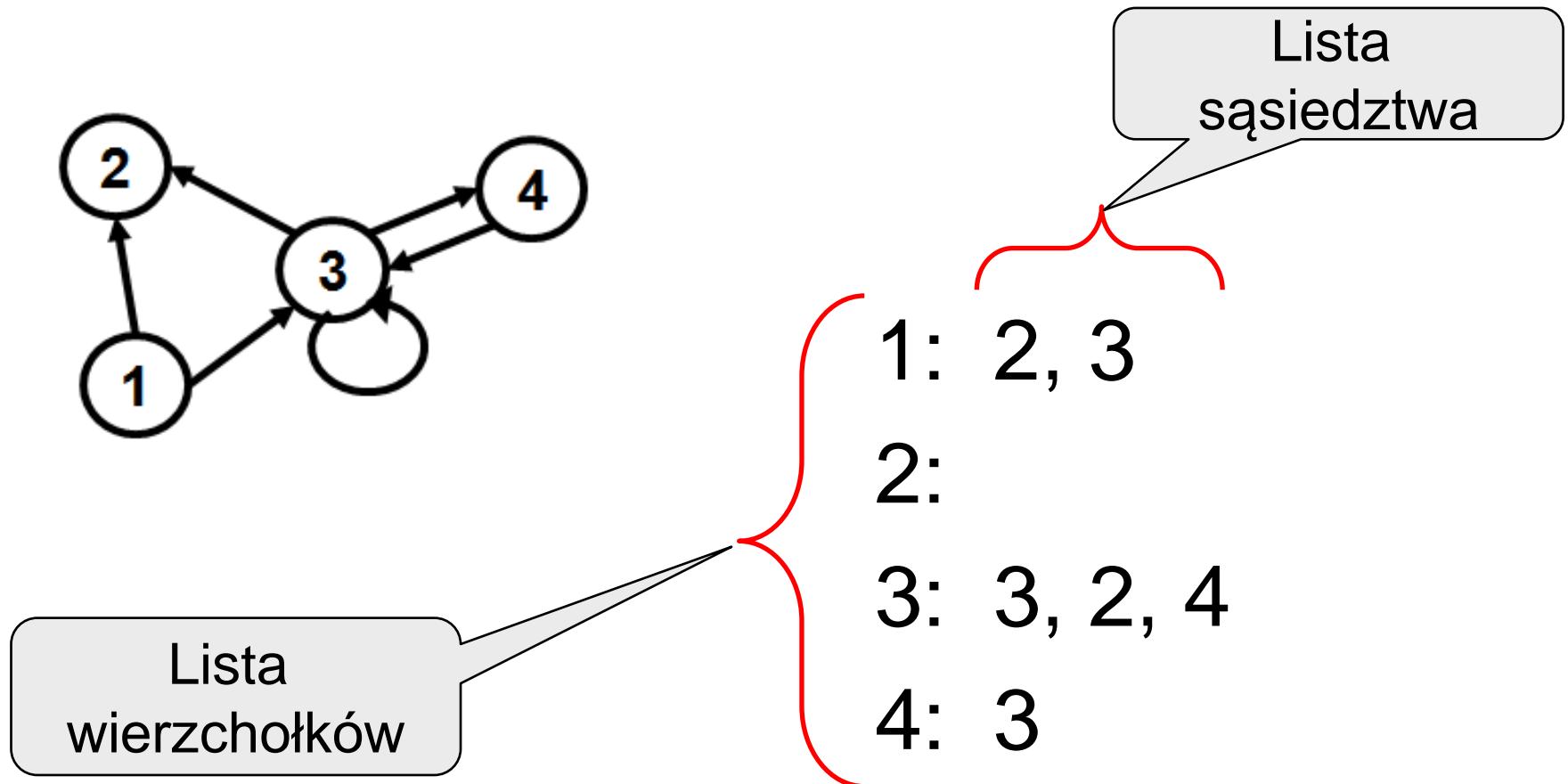
- W grafach skierowanych często etykietuje się krawędzie wagami, które symbolizują koszt przejścia po krawędzi
 - » Np. odległość między miastami, koszt przesunięcia ramienia robota itp.
- Jeśli nie podajemy wag krawędzi, to zwykle oznacza, że wagi wszystkich krawędzi są równe 1



Implementacje grafów

- Podstawową implementacją grafu jest implementacja tablicowa
 - » Tablica służy do implementacji listy lub macierzy sąsiedztwa
- Możliwa jest także implementacja dowiązaniowa
 - » Każdy wierzchołek „pamięta” wskaźniki do wierzchołków będących jego sąsiadami (dość niewygodne)

Jak implementować graf za pomocą list sąsiedztwa?



Klasa DGraph - przykładowa implementacja grafu skierowanego za pomocą list sąsiedztwa

- Zawiera w sobie listę wierzchołków (obiektów klasy DVertex), którą implementujemy za pomocą tablicy dynamicznej ArrayList

```
public class DGraph
{
    private ArrayList<DVertex> vertexList; //Lista wierzcholkow grafu

    public DGraph() // konstruktor
    {
        vertexList = new ArrayList<DVertex>();
    }

    public void addVertex(String label) //Dodanie NOWEGO wierzcholka grafu
    {
        ...
        DVertex vertex = new DVertex(label);
        vertexList.add(vertex);
    }

    ...
}
```

```

public class DVertex
{
    private String label; //Nazwa (etykieta) wierzchołka
    private ArrayList<String> vertexLabelList; //Lista sąsiedztwa w grafie
    private ArrayList<Integer> weightList; // Lista waag polaczen

    public DVertex(String label)
    {
        this.label = label;
        vertexLabelList = new ArrayList<String>();
        weightList = new ArrayList<Integer>();
    }

    public String getLabel() { return label; }

    public int getNoVertexLabel() { return vertexLabelList.size(); }

    public boolean addVertexLabel(String label,int weight)
    {
        ...
        vertexLabelList.add(label);
        weightList.add(new Integer(weight));
        return true;
    }
}

```

Klasa wierzchołka grafu: DVertex

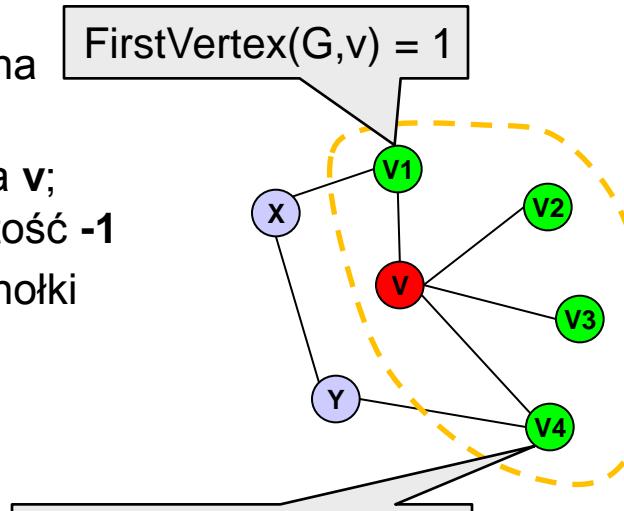
- Listy sąsiedztwa są implementowane jako listy etykiet wierzchołków do których mamy w grafie krawędzie od danego wierzchołka

Klasyczna implementacja grafu wymaga implementacji następujących operacji

- **AddVertex (G, v)** – dodanie wierzchołka v do grafu G (na razie bez krawędzi)
- **GetVertex(G, i)** – zwrócenie wierzchołka o numerze i w grafie G ; jeśli nie ma takiego wierzchołka, to zwracana jest wartość **null**
- **GetVertexIndex(G, v)** – zwrócenie numeru wierzchołka v ; jeśli nie ma takiego wierzchołka, to zwracana jest wartość **-1**
- **AddEdge(G, i_1, i_2)** – dodanie krawędzi łączącej wierzchołki o numerze i_1 i i_2 w grafie G
- **FirstVertexIndex(G, v)** – zwrócenie numeru (indeksu) pierwszego wierzchołka sąsiedniego dla wierzchołka v w grafie G ; jeśli wierzchołek v nie ma wierzchołków sąsiednich, to zwracana jest wartość **-1** (indeks wierzchołka pustego)
- **NextVertexIndex(G, v, i)** - zwrócenie numeru (indeksu) wierzchołka następnego w stosunku do wierzchołka o indeksie i wśród wierzchołków sąsiednich dla wierzchołka v w grafie G ; jeśli i jest indeksem ostatniego wierzchołka sąsiedniego dla v , to operacja zwraca wartość **-1** (indeks wierzchołka pustego)
- **RemoveVertex(G, i)** – usunięcie wierzchołka o numerze i z grafu G wraz z krawędziami łączącymi ten wierzchołek z wierzchołkami sąsiednimi
- **RemoveEdge(G, i_1, i_2)** – usunięcie krawędzi łączącej wierzchołki o numerach i_1 i i_2 (jeśli taka krawędź istnieje w grafie G)

FirstVertex(G, v) = 1

NextVertex(G, V_3) = 4
NextVertex(G, V_4) = -1



Implementacja grafu skierowanego za pomocą list sąsiedztwa

- Szczegóły implementacji grafu skierowanego w klasach pakietu:
`implabstract.graph`
w NetBeans

Przeszukiwanie grafów

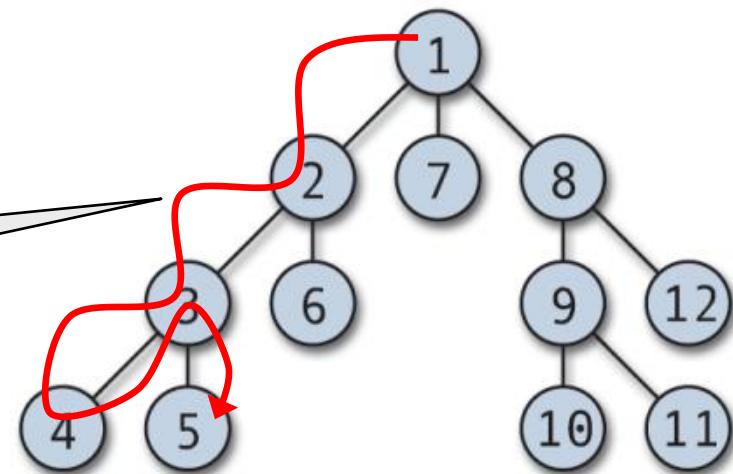
- Przeszukiwanie grafów jest podstawowym sposobem ich analizy
- Cel: Znaleźć algorytm, który zapewni nam przechodzenie grafu w taki sposób, że każdy wierzchołek zostanie przetworzony dokładnie jeden raz
- Efekt: Podczas przechodzenia grafu będziemy mogli dokonać jego analizy (np. znaleźć najkrótszą ścieżkę między wierzchołkami)
- Dwa podejścia: przeszukiwanie w głąb i przeszukiwanie wszerz

Przeszukiwanie grafu w głąb

(ang. depth-first search, w skrócie DFS)

- Polega na badaniu wszystkich krawędzi wychodzących z podanego wierzchołka.
- Po zbadaniu wszystkich krawędzi wychodzących z danego wierzchołka algorytm powraca do wierzchołka, z którego dany wierzchołek został odwiedzony
 - » Jest to zatem klasyczne przeszukiwanie z nawrotami

Kolejność przeszukiwania wierzchołków



Konstrukcja implementacji przeszukiwania w głąb

(wykorzystywany jest stos numerów wierzchołków)

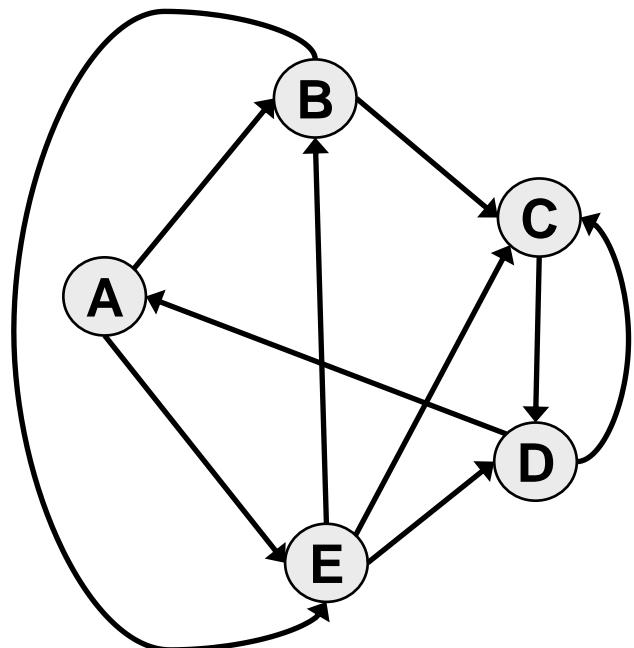
1. Utwórz stos numerów wierzchołków S (pusty).
2. Zapisz na stos wierzchołek początkowy A.
3. Oznacz wierzchołek A jako odwiedzony.
4. Dopóki stos S jest niepusty wykonuj:
 - » Podglądnij wierzchołek na szczycie stosu i nazwij go S.
 - » Jeśli wierzchołek S ma jeszcze chociaż jednego nieodwiedzonego sąsiada to:
 - Pobierz następnego nieodwiedzonego sąsiada i nazwij go N
 - Oznacz wierzchołek N jako odwiedzony.
 - Przetwórz N (np. wypisz go na ekran)
 - Zapisz na stos wierzchołek N
 - » wpp. ściągnij ze stosu wierzchołek S

Przykład przeszukiwania grafu w głąb

- Wierzchołek początkowy: A
- Kolejność odwiedzania:
 - » A B C D E

1. Utwórz stos numerów wierzchołków S (pusty).
2. Zapisz na stos wierzchołek początkowy A.
3. Oznacz wierzchołek A jako odwiedzony.
4. Przetwórz A (np. wypisz go na ekran)
5. Dopóki stos S jest niepusty wykonuj:
 - » Podglądnij wierzchołek na szczycie stosu i nazwij go S.
 - » Jeśli wierzchołek S ma jeszcze chociaż jednego nieodwiedzonego sąsiada to:
 - Pobierz następnego nieodwiedzonego sąsiada i nazwij go N
 - Oznacz wierzchołek N jako odwiedzony.
 - Przetwórz N (np. wypisz go na ekran)
 - Zapisz na stos wierzchołek N
 - » wpp. ściągnij ze stosu wierzchołek S

A: B, E
B: C, E
C: D
D: C A
E: D C B



Implementacja przeszukiwania grafu w głąb

- Szczegóły implementacji **przeszukiwania w głąb** grafu w klasie:
`implabstract.graph.DGraph`

Złożoność obliczeniowa przeszukiwania grafu w głąb

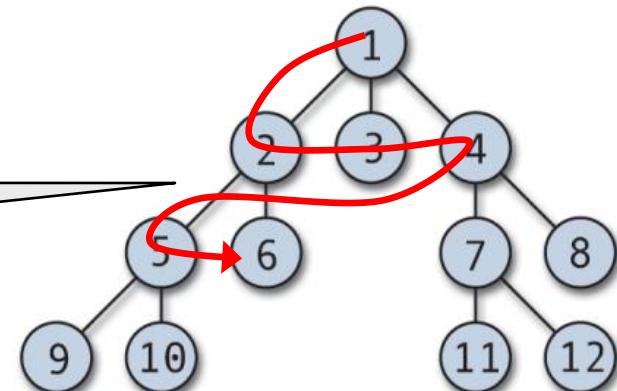
- **Złożoność czasowa:** Ponieważ w najgorszym przypadku przeszukiwanie w głąb musi przebyć wszystkie krawędzie prowadzące do wszystkich węzłów, złożoność czasowa przeszukiwania wszerz wynosi $O(|V| + |E|)$, gdzie $|V|$ to liczba węzłów, a $|E|$ to liczba krawędzi w grafie.
- **Złożoność pamięciowa:** Algorytm w każdym momencie wymaga zapamiętania tylko ścieżki od korzenia do bieżącego węzła, zatem złożoność jest rzędu $O(h)$, gdzie h jest długością najdłuższej prostej ścieżki w grafie

Przeszukiwanie grafu wszerz

(ang. Breadth-first search, w skrócie BFS)

- Przechodzenie grafu rozpoczyna się od zadanego wierzchołka i polega na odwiedzeniu wszystkich osiągalnych z niego wierzchołków
 - » Wierzchołki odwiedzane są warstwami, gdzie dana warstwa oznacza wierzchołki tak samo odległe od zadanego wierzchołka.

Kolejność przeszukiwania wierzchołków



Konstrukcja implementacji przeszukiwania grafu wszerz

(wykorzystywana jest kolejka numerów wierzchołków)

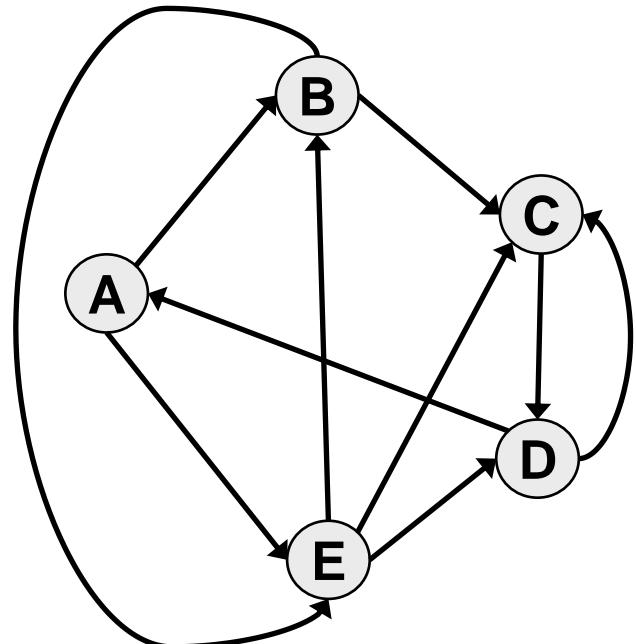
1. Utwórz kolejkę numerów wierzchołków Q (pusta).
2. Zapisz do kolejki wierzchołek początkowy A.
3. Oznacz wierzchołek A jako odwiedzony.
4. Przetwórz A (np. wypisz go na ekran)
5. Dopóki kolejka Q jest niepusta wykonuj:
 - » Pobierz z kolejki wierzchołek (pierwszy) i nazwij go S.
 - » Dla każdego wierzchołka N będącego jeszcze nieodwiedzonym sąsiadem S wykonuj:
 - Oznacz wierzchołek N jako odwiedzony.
 - Przetwórz N (np. wypisz go na ekran)
 - Wstaw N do kolejki Q

Przykład przeszukiwania grafu wszerz

- Wierzchołek początkowy: A
- Kolejność odwiedzania:
 - » A B E C D

1. Utwórz kolejkę numerów wierzchołków Q (pusta).
2. Zapisz do kolejki wierzchołek początkowy A.
3. Oznacz wierzchołek A jako odwiedzony.
4. Przetwórz A (np. wypisz go na ekran)
5. Dopóki kolejka Q jest niepusta wykonuj:
 - » Pobierz z kolejki wierzchołek (pierwszy) i nazwij go S.
 - » Dla każdego wierzchołka N będącego jeszcze nieodwiedzonym sąsiadem S wykonuj:
 - Oznacz wierzchołek N jako odwiedzony.
 - Przetwórz N (np. wypisz go na ekran)
 - Wstaw N do kolejki Q

A: B, E
B: C, E
C: D
D: C A
E: D C B



Implementacja przeszukiwania grafu wszerz

- Szczegóły implementacji **przeszukiwania wszerz** grafu w klasie:
`implabstract.graph.DGraph`

Złożoność obliczeniowa przeszukiwania grafu wszerz

- **Złożoność czasowa:** Ponieważ w najgorszym przypadku przeszukiwanie wszerz musi przebyć wszystkie krawędzie prowadzące do wszystkich węzłów, złożoność czasowa przeszukiwania wszerz wynosi $O(|V| + |E|)$, gdzie $|V|$ to liczba węzłów, a $|E|$ to liczba krawędzi w grafie.
- **Złożoność pamięciowa:** Jest uzależniona jest od tego w jaki sposób reprezentowany jest graf wejściowy.
 - » W przypadku listy sąsiedztwa dla każdego wierzchołka przechowywana jest lista wierzchołków osiągalnych bezpośrednio z niego. Dlatego w tym wypadku złożoność pamięciowa wynosi $O(|V| + |E|)$, gdzie $|V|$ to liczba węzłów, a $|E|$ to liczba krawędzi w grafie, odpowiadająca sumie wierzchołków znajdujących się na listach sąsiedztwa.
 - » W przypadku macierzy sąsiedztwa wymagane jest przechowywanie macierzy o wymiarach $|V| \times |V|$, czyli potrzebne jest $O(|V|^2)$ pamięci.

Podstawowe zastosowania przeszukiwania grafów

- Znajdowanie spójnych składowych grafu
- Znajdowanie cykli w grafie
- Sprawdzanie istnienia ścieżki pomiędzy wierzchołkami

Implementacje abstrakcyjnych struktur danych

Sposoby implementacji abstrakcyjnych struktur danych za pomocą konkretnych struktur danych

- Przez **modyfikację** istniejącej konkretnej struktury danych
- Przez **osadzenie** (wstawienie) w implementowanej klasie konkretnej struktury danych
- Przez **dziedziczenie** konkretnej struktury danych

Sposób implementacji przez modyfikację istniejącej konkretnej struktury danych

- Polega na tym że kod źródłowy struktury konkretnej jest modyfikowany celem uzyskania nazw metod i pól oraz działania poszczególnych metod zgodnego ze specyfikacją implementowanej abstrakcyjnej struktury danych
 - » **Zalety:** możliwość dokładnego dopasowania do potrzeb użytkownika
 - » **Wady:** zmiany kodu wymagają dużo czasu na samą implementację oraz na pełne przetestowanie klasy zwykle mocno zmodyfikowanej
- NIE JEST ZALECANE, dopóki nie będzie konieczne.

Przykład implementacji przez modyfikację: stos za pomocą tablicy dynamicznej

- Implementacja dostępna w klasie:
`implabstract.stack.DAStack1`

Sposób implementacji przez osadzanie istniejącej konkretnej struktury danych

- Sposób polega na tym, że definiuje się nową klasę implementowanej struktury i w tej klasie definiuje się (osadza, wstawia) obiekt (referencję do) struktury konkretnej jako zawartość zmiennej; podczas definiowania operacji implementowanej struktury wywoływane są metody na rzecz wstawionego obiektu
 - » **Zalety:** Szybkość i niezawodność implementacji
 - » **Wady:** Wszystkie metody trzeba definiować samemu (zwykle niczego nie można odziedziczyć)
- Stosowane często przy użyciu tablic dynamicznych, list powiązanych, drzew binarnych, tablic mieszających
- GODNE POLECENIA

Przykład implementacji przez osadzanie: stos za pomocą tablicy dynamicznej

- Implementacja dostępna w klasie:
`implabstract.stack.DAStack2`

Sposób implementacji przez dziedziczenie istniejącej konkretnej struktury danych

- Polega na tym, że definiuje się nową klasę, która dziedziczy po klasie struktury konkretnej; podczas definiowania operacji implementowanej struktury wywoływane są metody odziedziczonej klasy lub są one w miarę potrzeby przesłaniane
 - » **Zalety:** Duża szybkość implementacji oraz konieczność implementacji tylko przesłanianych metod (reszta jest odziedziczona)
 - » **Wady:** Nieco bardziej skomplikowane podejście techniczne z punktu widzenia OOP niż osadzanie oraz konieczność analizy wszystkich odziedziczonych metod pod kątem ich poprawności i przydatności w implementowanej strukturze danych
- Stosowane często np. przy użyciu tablic dynamicznych
- Sposób najbardziej zaawansowany z punktu widzenia OOP

Przykład implementacji przez dziedziczenie: stos za pomocą tablicy dynamicznej

- Implementacja dostępna w klasie:
`implabstract.stack.DAStack3`

Wybrane implementacje abstrakcyjnych struktur danych za pomocą tablicy dynamicznej

Implementacja stosu za pomocą tablicy dynamicznej

- Trzy przypadki implementacji stosu za pomocą tablicy dynamicznej zostały omówione wcześniej:

implabstract.stack.DAStack1

implabstract.stack.DAStack2

implabstract.stack.DAStack3

- UWAGA: Na poszczególne oceny obowiązują następujące implementacje stosu:
 - » na ocenę dostateczną – stos liczb całkowitych,
 - » na ocenę dobrą – stos wartości opakowanych (np. klasami Integer, Double, Float) lub stos obiektów typu String,
 - » na ocenę bardzo dobrą – stos złożonych obiektów zdefiniowanych przez użytkownika; np. stos osób (imię nazwisko, wiek), stos punktów ma płaszczyźnie (dwie współrzędne), stos książek (tytuł, wydawnictwo, rok wydania) itp.

Implementacja stosu za pomocą klasy `ArrayList` przez osadzanie

- Bardzo częsta sytuacja w codziennej praktyce programisty
 - » Jest oczywiście także możliwość implementacji przez odziedziczenie klasy `ArrayList`
- Implementacja dostępna w klasie:
`implabstract.stack.ArrayListStack`
- UWAGA: Na poszczególne oceny obowiązują następujące implementacje stosu:
 - » na ocenę dostateczną – obiektów typu `String`,
 - » na ocenę dobrą – stos wartości opakowanych (np. klasami `Integer`, `Double`, `Float`),
 - » na ocenę bardzo dobrą – stos złożonych obiektów zdefiniowanych przez użytkownika; np. stos osób (imię nazwisko, wiek), stos punktów ma płaszczyźnie (dwie współrzędne), stos książek (tytuł, wydawnictwo, rok wydania) itp.

Implementacja kolejki za pomocą tablicy dynamicznej

- Implementacja dostępna w klasie:
`implabstract.queue.DAQueue`
- UWAGA: Na poszczególne oceny obowiązują następujące implementacje kolejki:
 - » na ocenę dostateczną – kolejka liczb całkowitych,
 - » na ocenę dobrą – kolejka wartości opakowanych (np. klasami Integer, Double, Float) lub kolejka obiektów typu String,
 - » na ocenę bardzo dobrą – kolejka złożonych obiektów zdefiniowanych przez użytkownika; np. kolejka osób (imię nazwisko, wiek), kolejka punktów ma płaszczyźnie (dwie współrzędne), kolejka książek (tytuł, wydawnictwo, rok wydania) itp.

Implementacja kolejki za pomocą klasy ArrayList

- Implementacja dostępna w klasie:
`implabstract.queue.ArrayListQueue`
- UWAGA: Na poszczególne oceny obowiązują następujące implementacje kolejki:
 - » na ocenę dostateczną – kolejka liczb całkowitych,
 - » na ocenę dobrą – kolejka wartości opakowanych (np. klasami Integer, Double, Float) lub kolejka obiektów typu String,
 - » na ocenę bardzo dobrą – kolejka złożonych obiektów zdefiniowanych przez użytkownika; np. kolejka osób (imię nazwisko, wiek), kolejka punktów ma płaszczyźnie (dwie współrzędne), kolejka książek (tytuł, wydawnictwo, rok wydania) itp.

Implementacja kolejki priorytetowej za pomocą tablicy dynamicznej

- Implementacja dostępna w klasie:
`implabstract.queue.DAPriorityQ`
- Jest to kolejka priorytetowa znaków (tak jak na przykładzie wykładowym abstrakcyjnej kolejki priorytetowej).
 - » Podana informacyjnie (nie będzie wymagana przy zaliczeniu i egzaminie).

Implementacja listy z bezpośrednim dostępem do każdego elementu poprzez podanie jego numeru za pomocą tablicy dynamicznej

- Dla listy liczb całkowitych implementacja tego typu listy jest dostępna w klasie konkretnej:

structures.dynarray.IntDynArray

- Dla listy dowolnych obiektów tego typu listy jest dostępna w klasie konkretnej:

structures.dynarray.DynArray

- UWAGA: Na poszczególne oceny obowiązują następujące implementacje tego typu listy:

- » na ocenę dostateczną – lista liczb całkowitych,
- » na ocenę dobrą – lista wartości opakowanych (np. klasami Integer, Double, Float) lub lista obiektów typu String,
- » na ocenę bardzo dobrą – lista złożonych obiektów zdefiniowanych przez użytkownika; np. lista osób (imię nazwisko, wiek), lista punktów ma płaszczyźnie (dwie współrzędne), lista książek (tytuł, wydawnictwo, rok wydania) itp.

Implementacja zbioru za pomocą tablicy dynamicznej

- Implementacja dostępna w klasie:
`implabstract.set.DASet`

- UWAGA: Na poszczególne oceny obowiązują następujące implementacje zbioru:
 - » na ocenę dostateczną – zbiór liczb całkowitych,
 - » na ocenę dobrą – zbiór wartości opakowanych (np. klasami Integer, Double, Float) lub kolejka obiektów typu String,
 - » na ocenę bardzo dobrą – zbiór złożonych obiektów zdefiniowanych przez użytkownika; np. zbiór osób (imię nazwisko, wiek), zbiór punktów na płaszczyźnie (dwie współrzędne), zbiór książek (tytuł, wydawnictwo, rok wydania) itp.

Implementacja zbioru za pomocą klasy ArrayList przez osadzanie

- Implementacja dostępna w klasie:

`implabstract.set.ArrayListSet`

- UWAGA: Na poszczególne oceny obowiązują następujące implementacje zbioru:

- » na ocenę dostateczną – zbiór obiektów typu `String`,
- » na ocenę dobrą – zbiór wartości opakowanych (np. klasami `Integer`, `Double`, `Float`),
- » na ocenę bardzo dobrą – zbiór złożonych obiektów zdefiniowanych przez użytkownika; np. zbiór osób (imię nazwisko, wiek), zbiór punktów na płaszczyźnie (dwie współrzędne), zbiór książek (tytuł, wydawnictwo, rok wydania) itp.

Implementacja słownika (mapy) za pomocą tablicy dynamicznej

- Implementacja dostępna w klasie:
`implabstract.map.DAMapString`
- UWAGA: Na poszczególne oceny obowiązują następujące implementacje słownika:
 - » na ocenę dostateczną – słownik par: klucz całkowity + wartość typu `String`,
 - » na ocenę dobrą – słownik par: klucz typu `String` + wartość opakowana (np. klasami `Integer`, `Double`, `Float`),
 - » na ocenę bardzo dobrą – słownik par: klucz typu `String` + wartość będąca złożonym obiektem zdefiniowanym przez użytkownika (np. osoby (imię nazwisko, wiek), punkty ma płaszczyźnie (dwie współrzędne), książki (tytuł, wydawnictwo, rok wydania) itp.

Implementacja słownika (mapy) za pomocą klasy ArrayList

- Implementacja dostępna w klasie:
`implabstract.map.ArrayListMapString`
- UWAGA: Na poszczególne oceny obowiązują następujące implementacje słownika:
 - » na ocenę dostateczną – słownik par: klucz całkowity + wartość typu `String`,
 - » na ocenę dobrą – słownik par: klucz typu `String` + wartość opakowana (np. klasami `Integer`, `Double`, `Float`),
 - » na ocenę bardzo dobrą – słownik par: klucz typu `String` + wartość będąca złożonym obiektem zdefiniowanym przez użytkownika (np. osoby (imię nazwisko, wiek), punkty ma płaszczyźnie (dwie współrzędne), książki (tytuł, wydawnictwo, rok wydania) itp.

Wybrane implementacje abstrakcyjnych struktur danych za pomocą listy powiązanej

Implementacja stosu za pomocą listy powiązanej

- Dla liczb całkowitych implementacja dostępna w klasie:

`implabstract.stack.LLStack`

Implementacja stosu za pomocą klasы LinkedList przez osadzanie

- Implementacja dostępna w klasie:
`implabstact.stack.LinkedListStack`
- UWAGA: Na poszczególne oceny obowiązują następujące implementacje stosu:
 - » na ocenę dostateczną – obiektów typu String,
 - » na ocenę dobrą – stos wartości opakowanych (np. klasami Integer, Double, Float),
 - » na ocenę bardzo dobrą – stos złożonych obiektów zdefiniowanych przez użytkownika; np. stos osób (imię nazwisko, wiek), stos punktów ma płaszczyźnie (dwie współrzędne), stos książek (tytuł, wydawnictwo, rok wydania) itp.

Implementacja kolejki za pomocą listy powiązanej

- Dla liczb całkowitych implementacja dostępna w klasie:
`implabstract.queue.DAQueue`

Implementacja kolejki za pomocą klasy LinkedList

- Implementacja dostępna w klasie:
`implabstract.queue.LinkedListQueue`
- UWAGA: Na poszczególne oceny obowiązują następujące implementacje kolejki:
 - » na ocenę dostateczną – kolejka liczb całkowitych,
 - » na ocenę dobrą – kolejka wartości opakowanych (np. klasami Integer, Double, Float) lub kolejka obiektów typu String,
 - » na ocenę bardzo dobrą – kolejka złożonych obiektów zdefiniowanych przez użytkownika; np. kolejka osób (imię nazwisko, wiek), kolejka punktów ma płaszczyźnie (dwie współrzędne), kolejka książek (tytuł, wydawnictwo, rok wydania) itp.

Implementacja listy dwustronnej za pomocą listy powiązanej z klasy LinkedList

- Implementacja dostępna w klasie:

implabstract.list.LinkedListDList

- UWAGA: Na poszczególne oceny obowiązują następujące implementacje listy:

- » na ocenę dostateczną – lista liczb całkowitych,
- » na ocenę dobrą – lista wartości opakowanych (np. klasami Integer, Double, Float) lub stos obiektów typu String,
- » na ocenę bardzo dobrą – lista złożonych obiektów zdefiniowanych przez użytkownika; np. osób (imię nazwisko, wiek), punktów ma płaszczyźnie (dwie współrzędne), książek (tytuł, wydawnictwo, rok wydania) itp.

Wybrane implementacje abstrakcyjnych struktur danych za pomocą drzewa BST

Implementacja zbioru za pomocą drzewa binarnego z klasy TreeSet

- Implementacja dostępna w klasie:
`implabstract.set.TreeSetSet`
- UWAGA: Na poszczególne oceny obowiązują następujące implementacje zbioru:
 - » na ocenę dostateczną – zbiór liczb całkowitych,
 - » na ocenę dobrą – zbiór wartości opakowanych (np. klasami Integer, Double, Float) lub stos obiektów typu String,
 - » na ocenę bardzo dobrą – zbiór złożonych obiektów zdefiniowanych przez użytkownika; np. osób (imię nazwisko, wiek), punktów ma płaszczyźnie (dwie współrzędne), książki (tytuł, wydawnictwo, rok wydania) itp.

Wybrane implementacje abstrakcyjnych struktur danych za pomocą tablicy mieszającej

Implementacja zbioru za pomocą tablicy mieszającej

- Dla zbioru liczb całkowitych implementacja dostępna w klasie:

`implabstract.set.HTSet`

Implementacja zbioru za pomocą klasy HashSet

- Implementacja dla liczb całkowitych dostępna w klasie:
`implabstact.set.HashSetSet`
- UWAGA: Na poszczególne oceny obowiązują następujące implementacje zbioru:
 - » na ocenę dostateczną – zbiór obiektów typu String,
 - » na ocenę dobrą – zbiór wartości opakowanych (np. klasami Integer, Double, Float),
 - » na ocenę bardzo dobrą – zbiór złożonych obiektów zdefiniowanych przez użytkownika; np. zbiór osób (imię nazwisko, wiek), zbiór punktów na płaszczyźnie (dwie współrzędne), zbiór książek (tytuł, wydawnictwo, rok wydania) itp. (***ten przypadek wymaga zdefiniowania funkcji haszującej dla obiektów zdefiniowanej klasy***)

Zapis i odczyt zawartości struktury danych

- Zapis i odczyt zawartości struktury danych zwykle ma bardzo duże znaczenie praktyczne
- Odbywa się do pliku (najczęściej do pliku tekstowego lub binarnego) lub do bazy danych (zwykle relacyjnej, ale nie tylko)
- W przypadku zapisu do pliku tekstowego, plik ten często ma jakiś standardowy format (np. XML lub HTML).
 - » Często stosujemy także swoje własne formaty plików
- Do pliku binarnego zapisujemy dane, które są bardzo duże (zapis i odczyt z plików binarnych może być znacznie szybszy niż dla plików tekstowych) lub gdy nie zależy nam (lub wręcz nie chcemy tego), aby dane można było obejrzeć ręcznie za pomocą edytora tekstopwego
- Zapisując dane do plików często je kompresujemy „w locie” (wtedy musimy je dekompresować podczas odczytu)

Przykład zapisu i odczytu danych do/z pliku dla tablicy dynamicznej

- Implementacja dla plików tekstowych dostępna w klasie:

`implabstract.set.ArrayListSet`

Złożoność obliczeniowa i trudność problemów

Dwa rodzaje problemów obliczeniowych

- **Problem obliczeniowy** to zadanie, które może być rozwiązane za pomocą komputera lub innej maszyny liczącej.
 - » Istnieją **dwa rodzaje problemów obliczeniowych**: problemy decyzyjne i problemy optymalizacyjne.
- **Problem decyzyjny** to pewien zbiór parametrów oraz pytanie, na które odpowiedź brzmi „tak” lub „nie”
 - » Ustalając wartości tych parametrów otrzymujemy **instancję**, czyli **konkretny przypadek problemu**.
- **Problem optymalizacyjny**, to taki problem, w którym należy ekstremalizować (maksymalizować lub minimalizować) pewną funkcję celu
 - » Rozwiązaniem jest konkretna instancja, dla której zostało osiągnięte ekstremum
- **Wiele problemów może być formułowane zarówno w wersji decyzyjnej, jak i w wersji optymalizacyjnej**

Przykład sformułowania problemu jako problem decyzyjny i optymalizacyjny dla problemu plecakowego

- Dany jest skończony zbiór elementów $A=\{a_1, \dots, a_n\}$, ciąg rozmiarów tych elementów $s(a_1), \dots, s(a_n)$, ich wartości $w(a_1), \dots, w(a_n)$ oraz stałe b i t .
- **Wersja decyzyjna:** Czy istnieje podzbiór $B \subseteq A$ taki, że

$$\sum_{a_i \in B} s(a_i) \leq b \quad \text{i} \quad \sum_{a_i \in B} w(a_i) \geq t \quad ?$$

- » Innymi słowy: Czy istnieje taki podzbiór zbioru elementów A , że elementy z tego podzbioru mieszczą się do plecaka o objętości b , a ich wartość jest nie mniejsza niż t ?
- **Wersja optymalizacyjna:** Znaleźć podzbiór zbioru $B \subseteq A$ taki, że
$$\sum_{a_i \in B} s(a_i) \leq b \quad \text{i} \quad \sum_{a_i \in B} w(a_i) \quad \text{osiąga wartość maksymalną.}$$
» Innymi słowy: Znajdź taki podzbiór zbioru A , że elementy z tego podzbioru mieszczą się do plecaka o objętości b , a ich wartość jest możliwie największa.

Problem i jego instancja czyli konkretny przypadek problemu

- Rozważmy **problem plecakowy**: Dany jest skończony zbiór elementów $A=\{a_1, \dots, a_n\}$, ciąg rozmiarów tych elementów $s(a_1), \dots, s(a_n)$, ich wartości $w(a_1), \dots, w(a_n)$ oraz pojemność b plecaka i stała t .
- Ustalając powyższe parametry uzyskujemy instancję problemu
 - » **Przykład instancji**: niech zbiór A składa się z 5 elementów o rozmiarach wynoszących odpowiednio 5, 3, 2, 4, 3 i wartościach 3, 4, 2, 6, 1 oraz pojemność plecaka $b=10$ i stała $t=12$.
 - » Zauważmy, że w przypadku problemu decyzyjnego odpowiedź brzmi tak, gdyż w przypadku podzbioru $\{3, 2, 4\}$ jego sumaryczny rozmiar wynosi 9 (mieści się do plecaka) i wartość sumaryczna wynosi 12, czyli jest większa lub równa t

Rozmiar instancji problemu

- Dane instancji I (konkretnie wartości parametrów problemu) zapisuje się (koduje) za pomocą skończonego łańcucha $L(I)$ symboli należących do z góry określonego alfabetu Σ zgodnie z ustaloną regułą kodowania.
- Przez **rozmiar** instancji $N(I)$ rozumiemy **długość łańcucha** $L(I)$
 - » Np. dla problemu plecakowego ustalmy alfabet $\Sigma=\{0,1,\text{ , },\text{ . }\}$ (zero, jeden, spacja i kropka). Wszystkie liczby zapisujemy w postaci binarnej, oddzielając je spacjami, w kolejności: liczba elementów, pojemność plecaka b , ograniczenie t , rozmiary elementów oraz ich wartości. Kropka oznacza koniec łańcucha danych.
 - » Rozmiar instancji wyliczamy według wzoru:

$$N(I) = \lfloor \log_2 n \rfloor + \lfloor \log_2 b \rfloor + \lfloor \log_2 t \rfloor + 3 + \sum_{i=1}^n \lfloor \log_2 s(a_i) \rfloor + 1 + \sum_{i=1}^n \lfloor \log_2 w(a_i) \rfloor + 1 + 2n + 3$$

Wybór sposobu kodowania

- Dane można kodować także według różnych sposobów (reguł)
- Kodowanie powinno być jednoznaczne i możliwie zwięzłe (bez redundancji)
- Kodowanie powinno być także „rozsądne”, czyli nie powodować wykładniczego wzrostu rozmiaru kodowanej instancji w stosunku do innych reguł kodowania
 - » Np. kod jedynkowy w przypadku problemu plecakowego, czyli taki, że liczba k jest kodowana za pomocą k -jedynek jest nierozsądny, gdyż jego rozmiar rośnie bardzo szybko w stosunku do kodu binarnego (z poprzedniego slajdu)
 - Dla przykładu liczbę 9 w kodzie binarny kodujemy jako 1001, a w kodzie jedynkowym jako 11111111

Problemy decyzyjne a optymalizacyjne

- Z danym problemem optymalizacyjnym **można związać odpowiadający mu problem decyzyjny** i taki problem decyzyjny **jest obliczeniowo nie trudniejszy**, niż odpowiadający mu pierwotny problem optymalizacyjny (uproszczenie).
- Jeśli **problem decyzyjny jest obliczeniowo „trudny”**, to **„trudny” jest również odpowiadający mu problem optymalizacyjny** (utrudnienie).
- **Obserwacja 1:** Jeśli da się w "prosty" sposób rozwiązać problem optymalizacyjny, to można również "prosto" rozwiązać zвязany z nim problem decyzyjny.
 - Np. jeśli potrafimy rozwiązać problem optymalizacyjny plecakowy, to potrafimy rozwiązać odpowiadający mu problem decyzyjny
- **Obserwacja 2:** Jeśli problem decyzyjny jest „trudny”, to odpowiadający mu problem optymalizacyjny również jest „trudny” (pytamy o coś więcej).
 - Np. jeśli problem plecakowy w wersji decyzyjnej jest trudny, to na pewno problem plecakowy w wersji optymalizacyjnej także jest trudny

Dwa wnioski dotyczące dowodzenia „łatwości” i „trudności” problemów

- **Wniosek 1:** W celu wykazania "łatwości" problemu decyzyjnego wystarczy wykazać łatwość problemu optymalizacyjnego (np. zaproponować szybki algorytm).
- **Wniosek 2:** Dla wykazania "trudności" problemu optymalizacyjnego wystarczy wykazać "trudność" związanego z nim problemu decyzyjnego.

Uściślenie pojęcia złożoności obliczeniowej (1)

- Przyjętą miarą złożoności czasowej jest **liczba operacji jednostkowych** w zależności od rozmiaru wejścia.
 - » Pomiar rzeczywistego czasu zegarowego **jest mało użyteczny** ze względu na silną zależność od sposobu realizacji algorytmu, użytego kompilatora oraz maszyny na której algorytm wykonujemy.
 - » Dlatego w charakterze czasu wykonania rozpatruje się zwykle liczbę **operacji jednostkowych** lub nawet **dominujących**.
 - » Operacjami podstawowymi mogą być na przykład: podstawienie, porównanie lub prosta operacja arytmetyczna.
- Często pokrywa się to z poprzednim podejściem do definiowania złożoności czasowej (poprzedni semestr), które było oparte na rozmiarze charakterystycznym zadania (ale nie zawsze)
 - » Czasami rozmiar danych na wejściu jest w istotny sposób różny (np. mniejszy) od rozmiaru charakterystycznego zadania

Uściślenie pojęcia złożoności obliczeniowej (2)

- Przyjętą miarą złożoności czasowej jest zatem liczba operacji jednostkowych w zależności od rozmiaru danych wejściowych
- Rozmiar danych wejściowych często rozumiemy jako liczbę pojedynczych danych na wejściu (rozmiar charakterystyczny zadania)
- Bardziej ściśle jest to jednak rozmiar danych wejściowych wyliczony dla ustalonej metody kodowania instancji na wejściu
- Przez **funkcję złożoności obliczeniowej algorytmu A** (oznaczoną przez f_A) rozwiązującego problem P rozumiemy funkcję przyporządkowującą każdej wartości rozmiaru instancji $I \in D_P$ (liczbie $n=N(I)$) maksymalną liczbę operacji jednostkowych potrzebnych do rozwiązania problemu P za pomocą algorytmu A.
 - » Dla różnych instancji o rozmiarze n algorytm A może wykonać różne liczby operacji jednostkowych i dlatego ustalamy tutaj maksymalną liczbę

Algorytm wielomianowy i wykładniczy

- **Algorytmem wielomianowym** nazywamy algorytm, którego funkcja złożoności obliczeniowej jest rzędu $O(p(n))$, gdzie p jest pewnym wielomianem, a n jest rozmiarem rozwiązywanej instancji.
- Każdy algorytm, którego funkcja złożoności obliczeniowej nie może być tak ograniczona nazywamy **algorytmem wykładniczym**
- Uwaga: *Funkcja złożoności obliczeniowej algorytmu wykładniczego nie musi być funkcją wykładniczą*

Przykład problemu wielomianowego

- Założmy, że pewien algorytm wyszukujący minimum m-elementowego ciągu liczb naturalnych mniejszych od 10 wymaga wykonania $4*m+5$ operacji jednostkowych (prosty linowy algorytm).
- Ciąg liczb kodujemy binarnie, przeznaczając 4 bity na każdy element
- Dane na wejściu mają zatem rozmiar:
$$n = N(I) = 4 + 1 + m * 4 + m - 1 = 5m - 4$$
 (potrzebujemy 4 bity na rozmiar ciągu, 1 bit na spację, 4m bitów na elementy oraz $m-1$ spacji na oddzielenie elementów ciągu od siebie)
- Stąd $n=5m-4$ oraz $m=(n+4)/5$
- Zatem $f_A(n) = 4*m+5 = 4*(n+4)/5+5 = 4/5*n + 8 \frac{1}{5}$
- Widać zatem, że $f_A(n)$ da się ograniczyć przez wielomian liniowy, zatem algorytm jest rzędu $O(n)$

Złożoność obliczeniowa narzędziem do charakteryzacji trudności problemów

- W informatyce **problem jest trudny**, gdy nie da się go rozwiązać za pomocą algorytmu o złożoności obliczeniowej wielomianowej (nie jest znany taki algorytm go rozwiązujący, a jest znany algorytm o złożoności wykładniczej)
 - » Przykłady rzędów złożoności problemów nie będących trudnymi: $O(\log n)$, $O(n)$, $O(n \log n)$, $O(n^2)$, $O(n^{10})$
 - Problemy **nie będące trudnymi**: problem sortowania ciągu, problem wyszukania minimum zbioru, problem wyszukiwania wartości w zbiorze danych
 - » Przykłady rzędów złożoności problemów trudnych: $O(2^n)$, $O(2^{2^n})$, $O(n!)$
 - Problemy **trudne**: problem plecakowy, problem komiwojażera, problem pokrycia zbiorów
- Ponieważ dla wykazania "trudności" problemu optymalizacyjnego wystarczy wykazać "trudność" związanego z nim problemu decyzyjnego, **zwykle w teoretycznych rozważaniach dotyczących trudności problemów rozważa się wersje decyzyjne.**

Trzy kategorie problemów ze względu na ich trudność

1. Problemy, dla których **wynaleziono algorytm wielomianowy** (na pewno nie są trudne)
2. Problemy, których **trudność została udowodniona** (na pewno są trudne)
3. Problemy, których **trudność nie została udowodniona**, jednak **nie udało się wynaleźć** rozwiązujących je algorytmów wielomianowych (nie wiadomo, czy są trudne)
 - » ***Uwaga: Większość problemów rozważanych przez informatyków należy do pierwszej albo do trzeciej kategorii***

Cztery szczególne klasy problemów

- Rozważa się 4 następujące klasy problemów, które zostały utworzone z punktu widzenia złożoności obliczeniowej danego problemu
 - » problemy z klasy **P**,
 - » problemy z klasy **NP**,
 - » klasa problemów **NP-zupełnych**,
 - » klasa problemów **NP-trudnych**.

Klasa NP problemów

- Problemami klasy NP nazywamy problemy decyzyjne w których sprawdzenie poprawności określonego rozwiązania wymaga złożoności obliczeniowej wielomianowej.
- Z powyższego stwierdzenia wynika więc, że znalezienie rozwiązania dla problemów NP wymaga złożoności co najmniej wielomianowej

Problem sumy podzbioru (przykład problemu z klasy NP)

- Sformułowanie: Czy suma jakikolwiek niepustego podzbioru zadanego zbioru X liczb całkowitych (np. $X=\{-2,6,-3,72,10,-11\}$) jest równa k , gdzie k jest liczbą całkowitą?
- Trudno znaleźć rozwiązanie tego zagadnienia w czasie wielomianowym.
 - » Nasuwający się algorytm sprawdzenia wszystkich możliwych podzbiorów ma złożoność wykładniczą ze względu na liczebność zbioru.
- Nie wiadomo zatem, czy problem ten jest klasy P.
- Na pewno natomiast uzyskawszy z zewnątrz kandydata na rozwiązanie (np. $Y=\{-2,6,10,-11\}$) możemy w liniowym (a zatem wielomianowym) czasie sprawdzić, czy suma elementów takiego podzbioru jest równa k . Jest to zatem problem z klasy NP.

Klasa problemów P

- Problemami klasy P nazywamy **problemy decyzyjne** w których znalezienie rozwiązania **wymaga złożoności obliczeniowej wielomianowej**.
- Problemy P należą do zbioru problemów NP, czyli sprawdzenie poprawności rozwiązania danego problemu jest możliwe w czasie wielomianowym;
- Jednak znalezienie rozwiązania problemu z klasy P jest możliwe również w czasie wielomianowym.
- Różnica pomiędzy problemami P i NP polega więc na tym, że w przypadku P znalezienie rozwiązania ma mieć złożoność wielomianową, podczas gdy dla NP sprawdzenie podanego z zewnątrz rozwiązania ma mieć taką złożoność.

Pytanie: P=NP ?

- Każdy problem P jest NP, jednak nie wiadomo, czy każdy problem NP jest P.
- Jest to jedno z wielkich nierozerwanych dotychczas zagadnień informatyki.
 - » Najważniejsze pytanie teoretycznej informatyki i jest jednym z problemów milenijnych.

Transformacja wielomianowa problemów

- Założmy, że są dane dwa problemy decyzyjne P_1 i P_2 , które posiadają dwa zbiory instancji odpowiednio D_1 i D_2
- **Transformacja wielomianowa** problemu P_2 do problemu P_1 to taka funkcja $f: D_2 \rightarrow D_1$, która spełnia dwa następujące warunki:
 1. dla każdej instancji I_2 należącej do D_2 odpowiedź brzmi "tak" wtedy i tylko wtedy, gdy dla instancji $f(I_2)$ odpowiedź brzmi również "tak"
 2. czas deterministycznego obliczania wartości funkcji f dla każdej instancji I_2 należącej do D_2 jest ograniczony od góry przez wielomian zależny od rozmiaru instancji I_2 , czyli od $N(I_2)$
- Jeśli istnieje transformacja wielomianowa problemu P_2 do problemu P_1 to mówimy, że problem P_2 **jest redukowalny** do problemu P_1
 - » Oznacza to, że jeśli potrafimy rozwiązać wielomianowo problem P_2 , to również wielomianowo potrafimy rozwiązać problem P_1

Przykład transformacji wielomianowej problemów (1)

- Problem P1: **Problem podziału zbioru**
 - » Dany jest skończony zbiór $C=\{c_1, \dots, c_m\}$ oraz rodzina wag $v(c_1), \dots, v(c_m)$ elementów ze zbioru C
 - » Czy istnieje podzbiór $D \subseteq C$ o takiej własności, że suma wag elementów ze zbioru D jest równa sumie wag elementów ze zbioru $C \setminus D$?
- Problem P2: **Problem plecakowy**
 - » Dany jest skończony zbiór elementów (przedmiotów) $A=\{a_1, \dots, a_n\}$, ciąg rozmiarów tych elementów $s(a_1), \dots, s(a_n)$, ich wartości $w(a_1), \dots, w(a_n)$ oraz pojemność b plecaka i stała t .
 - » Czy istnieje taki podzbiór $B \subseteq A$, że elementy z tego podzbioru mieszczą się do plecaka o objętości b , a ich wartość jest nie mniejsza niż t ?
- *Pokażemy, że problem P1 jest redukowalny do problemu P2*

Przykład transformacji wielomianowej problemów (2)

- Dla udowodnienia redukowalności P_1 do P_2 należy skonstruować funkcję $f: D_1 \rightarrow D_2$ (gdzie D_1 i D_2 są zbiorami instancji problemów P_1 i P_2) taką, że:
 1. dla każdej instancji I_1 należącej do D_1 odpowiedź brzmi "tak" wtedy i tylko wtedy, gdy dla instancji $f(I_1)$ odpowiedź brzmi również "tak"
 2. czas deterministycznego obliczania wartości funkcji f dla każdej instancji I_1 należącej do D_1 jest ograniczony od góry przez wielomian zależny od rozmiaru instancji I_1 , czyli od $N(I_1)$

Przykład transformacji wielomianowej problemów (3)

- Definiujemy funkcję $f: D_1 \rightarrow D_2$ w następujący sposób.
- Założmy, że jest dana instancja $I_1 \in D_1$ w skład której wchodzi zbiór $C = \{c_1, \dots, c_m\}$ elementów z wagami $v(c_1), \dots, v(c_m)$
- Funkcja f instancji I_1 przypisuje instancję $I_2 \in D_2$ w następujący sposób:
 - » liczba elementów $n=m$
 - » rozmiary $s(a_i) = v(c_i)$, dla $i=1, \dots, n$
 - » wartości $w(a_i) = v(c_i)$, dla $i=1, \dots, n$
 - » stała $b = t = \text{połowa sumy wag } v(c_1), \dots, v(c_m)$

Problem P1: Problem podziału zbioru

- » Dany jest skończony zbiór $C = \{c_1, \dots, c_m\}$ oraz rodzina wag $v(c_1), \dots, v(c_m)$ elementów ze zbioru C
- » Czy istnieje podzbiór $D \subseteq C$ o takiej własności, że suma wag elementów ze zbioru D jest równa sumie wag elementów ze zbioru $C \setminus D$?

Problem P2: Problem plecakowy

- » Dany jest skończony zbiór elementów (przedmiotów) $A = \{a_1, \dots, a_n\}$, ciąg rozmiarów tych elementów $s(a_1), \dots, s(a_n)$, ich wartości $w(a_1), \dots, w(a_n)$ oraz pojemność b plecaka i stała t .
- » Czy istnieje taki podzbiór $B \subseteq A$, że elementy z tego podzbioru mieszczą się do plecaka o objętości b , a ich wartość jest nie mniejsza niż t ?

Przykład transformacji wielomianowej problemów (4)

- **Dowód warunku 1 (\Leftarrow) :** Jeśli odpowiedź dla I2 jest „tak”, to istnieje podzbiór B przedmiotów z A, że elementy z B mieszczą się do plecaka o objętości b , a ich wartość ich w sumie jest nie mniejsza niż t .
- Istnieje także zbiór D zawarty w C, do którego należą elementy z C odpowiadające elementom z B.
- Ponieważ $b=t$ i rozmiary przedmiotów równają się ich wartościom, suma ich rozmiarów musi być równa b.
- Z drugiej strony ta suma wartości elementów z B jest równa sumie elementów ze zbioru D, a ta z kolei jest równa połowie sumy wag elementów z problemu P1. Zatem suma elementów $C \setminus D$ także będzie równa b.
- Elementy zbioru C odpowiadające elementom zbioru B (zbior D) tworzą zatem podzbiór o zadanej własności, czyli odpowiedź dla I1 brzmi „tak”.

Przykład transformacji wielomianowej problemów (5)

- **Dowód warunku 1 (\Rightarrow) :** Niech teraz odpowiedź dla I1 jest „tak”. Oznacza to, że istnieje podzbiór D taki, że suma waga dla D jest równa połowie sumy wag dla całego C. Jeśli zatem z elementów zbioru A odpowiadających podzbiorowi D utworzymy podzbiór B, to suma wag w tego podzbioru będzie równa sumie rozmiarów oraz liczbie b i liczbie t.

$$\sum_{a_i \in B} s(a_i) = \sum_{a_i \in B} w(a_i) = b = t$$

- Elementy podzbioru B tworzą zatem podzbiór A o zadanej własności, czyli odpowiedź dla I2 brzmi „tak”.

Przykład transformacji wielomianowej problemów (6)

- **Dowód warunku 2:** Łatwo można zauważyć, że czas deterministycznego obliczania wartości funkcji f dla każdej instancji I_1 należącej do D_1 jest ograniczony od góry przez wielomian zależny od rozmiaru instancji I_1 , czyli od $N(I_1)$, gdyż procedura ta musi przepisać $2n+3$ liczb co może być zrobione w czasie wielomianowym (przypisanie wartości n , b , t oraz wag i rozmiarów)

P₁ jest redukowalny do problemu P₂ c.n.d.

Klasa problemów NP-zupełnych

- Do klasy problemów NP-zupełnych należy każdy problem decyzyjny spełniający trzy poniższe warunki:
 - » nie jest znany algorytm rozwiązujący ten problem w czasie wielomianowym,
 - » należy do klasy NP (rozwiązania problemu dla konkretnych danych dają się weryfikować w czasie wielomianowym),
 - » dowolny problem należący do NP może być do niego zredukowany w czasie wielomianowym.
- Pierwszym problemem, którego NP-zupełność wykazano, był problem SAT, czyli problem spełnialności formuł zdaniowych (Stephen Cook, 1971).
 - » SAT to pytanie rachunku zdań - czy dla danej formuły logicznej istnieje takie podstawienie (wartościowanie) zmiennych zdaniowych, żeby formuła była prawdziwa.
 - » Cook podał dowód, że problem SAT należy do NP oraz każdy problem z klasy NP jest redukowalny do tego problemu

Dowodzenie NP-zupełności problemów

- Dla wykazania NP-zupełności badanego problemu P_1 decyzyjnego wystarczy udowodnić, że ten problem należy do klasy NP oraz przetransformować do niego wielomianowo dowolny znany problem NP-zupełny.
 - Uzasadnienie: Jeśli problem P_1 jest z klasy NP i istnieje problem P_2 redukowalny do P_1 , to wszystkie problemy z klasy NP są redukowalne do P_1 . Wynika to z faktu, że wszystkie problemy były redukowalne do problemu SAT, a zatem pośrednio są one redukowalne do P_2 , a więc i do P_1 (poprzez złożenie transformacji wielomianowych, które także jest transformacją wielomianową)

Przykład dowodu NP-zupełności problemu dla decyzyjnego problemu plecakowego

- Problem plecakowy jest w klasie NP, gdyż dla danego podzbioru przedmiotów można łatwo sprawdzić, czy mieszczą się w plecaku i ich waga jest co najmniej t
- Problem podziału redukuje się do problemu plecakowego
- Problem podziału zbioru jest NP-zupełny (wiadomo z literatury)
- Wniosek: Problem plecakowy jest NP-zupełny

Klasa problemów NP-trudnych

- Do klasy problemów NP-trudnych należą problemy spełniające dwa poniższe warunki:
 - » nie jest znany algorytm rozwiązuający ten problem w czasie wielomianowym (mogą być znane rozwiązania wykładnicze),
 - » dowolny problem należący do NP może być do niego zredukowany w czasie wielomianowym.
- Każdy problem NP-zupełny jest problemem NP-trudnym
- **Istnieją problemy NP-trudne, które nie są NP-zupełne**
 - » W definicji NP-trudności nie wymaga się należenia do klasy NP, a jedynie redukowalności wszystkich problemów z klasy NP do danego problemu; ponadto, problemy w klasie NP-trudnych nie muszą być decyzyjne
- **Ponieważ dla problemu NP-trudnego, każdy problem z klasy problemów NP-zupełnych może być zredukowany (sprawdzony) do tego problemu, zatem problem NP-trudny to taki problem obliczeniowy, którego rozwiązanie jest co najmniej tak trudne jak rozwiązanie każdego problemu z klasy NP.**

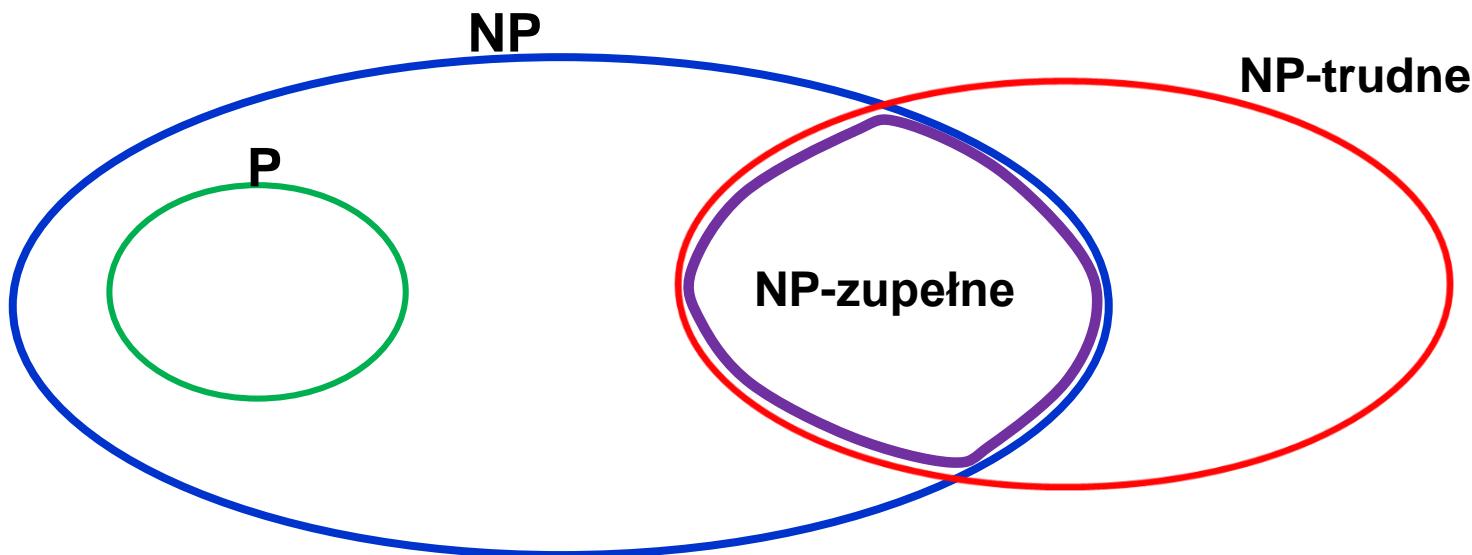
Zasady dowodzenie NP-trudności problemów

- Każdy problem optymalizacyjny, którego wersja decyzyjna jest NP-zupełna jest problemem NP-trudnym
- Dla wykazania trudności rozważanego problemu optymalizacyjnego wystarcza wykazanie NP-zupełności odpowiadającego mu problemu decyzyjnego.
 - » Mówimy wtedy, że dany problem optymalizacyjny jest NP-trudny.
 - Np: problem decyzyjny plecakowy jest NP-zupełny i dlatego problem optymalizacyjny plecakowy jest NP-trudny

Klasyfikacja problemów z punktu widzenia złożoności czasowej

Klasa problemu	Klasy złożoności czasowej	
	Sprawdzenie poprawności rozwiązania	Znalezienie rozwiązania
problem P	wielomianowa	
problem NP	wielomianowa	wielomianowa lub wykładnicza
problem NP-zupełny	wielomianowa	wykładnicza
problem NP-trudny	wielomianowa lub wykładnicza	wykładnicza

Hierarchia klas złożoności czasowej



Znaczenie klasyfikacji problemów jako uzasadnienie naszej niemocy

- Jeśli udowodnimy NP-zupełność swojego problemu to znaczy, że jesteśmy w takiej sytuacji jak wiele innych osób
- Jeśli udowodnimy NP-trudność, to sytuacja jest jeszcze gorsza, gdyż być może wychodzimy poza klasę NP

Klasy problemów z punktu widzenia złożoności czasowej

- P-SPACE – problemy w których znalezienie rozwiązania wymaga pamięci ograniczonej przez pewien wielomian $p(n)$ zależny od rozmiaru problemu n
- NP-SPACE – problemy w których zweryfikowanie poprawności rozwiązania dla konkretnych danych wymaga pamięci ograniczonej przez pewien wielomian $p(n)$ zależny od rozmiaru problemu n
- $P\text{-SPACE} \subseteq NP\text{-SPACE}$ – skoro da się rozwiązać problem w pamięci wielomianowej (co np. wymaga wczytania i przetworzenia danych wejściowych), to sama weryfikacja danych musi dać się zrobić w pamięci wielomianowej (np. w celu tej weryfikacji rozwiązujemy dany problem i wykorzystujemy rozwiązanie do weryfikacji)

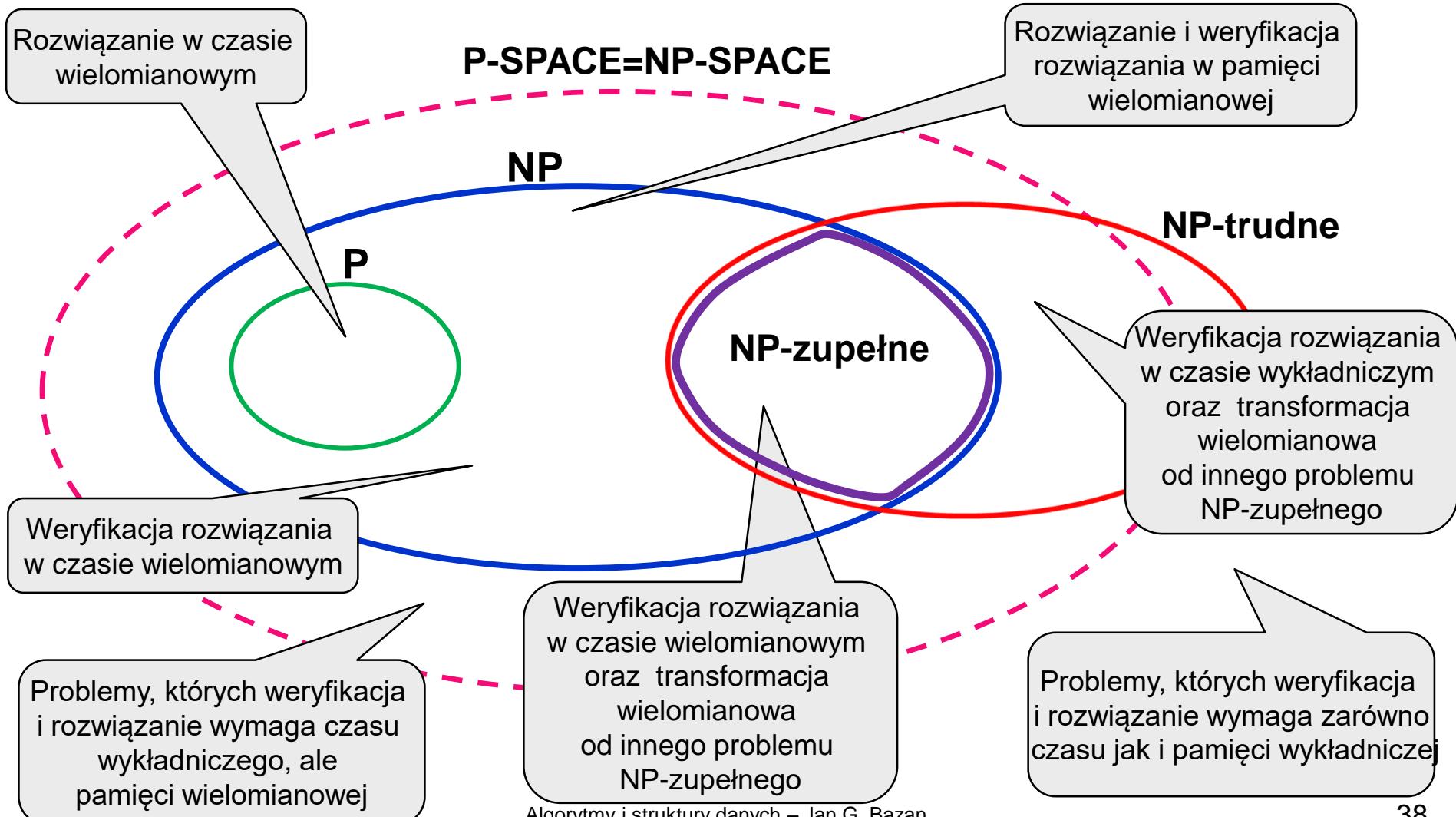
Relacja pomiędzy P i P-SPACE oraz NP i NP-SPACE

- $P \subseteq P\text{-SPACE}$ – jeśli algorytm rozwiązujący problem z klasy P wykonuje wielomianowo ograniczoną liczbę ruchów, to używa on wielomianowej liczby komórek pamięci (każdy ruch używa pewną skończoną liczbę komórek pamięci).
- $NP \subseteq NP\text{-SPACE}$ - jeśli algorytm, który potrafi zweryfikować poprawność rozwiązania dla konkretnych danych wykonuje wielomianowo ograniczoną liczbę ruchów, to używa on wielomianowej liczby komórek pamięci

Relacja pomiędzy P-SPACE i NP-SPACE

- Twierdzenie Savitcha: **P-SPACE=NP-SPACE**
 - » Dyskusyjne jest tylko: $\text{NP-SPACE} \subseteq \text{P-SPACE}$
 - » Jeśli dla danego problemu da się zweryfikować poprawność rozwiązania dla konkretnych danych w pamięci ograniczonej przez pewien wielomian $p(n)$ zależny od rozmiaru problemu n , to da się również rozwiązać ten problem w pamięci wielomianowej (dowód dotyczy maszyn Turinga)

Umiejscowienie klasy P-SPACE w hierarchii klas złożoności czasowej

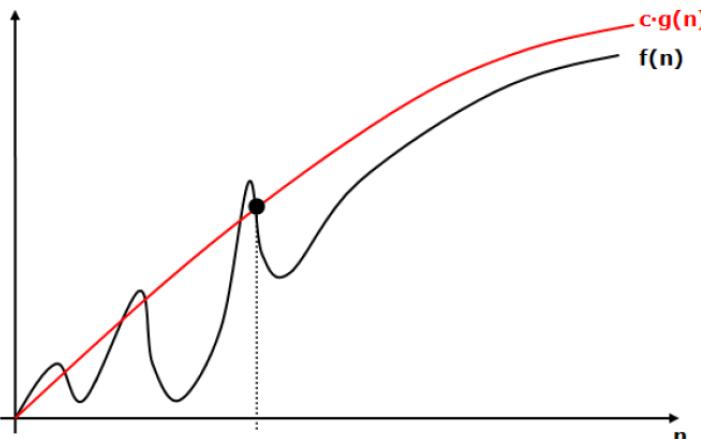


Lista znanych problemów NP-trudnych

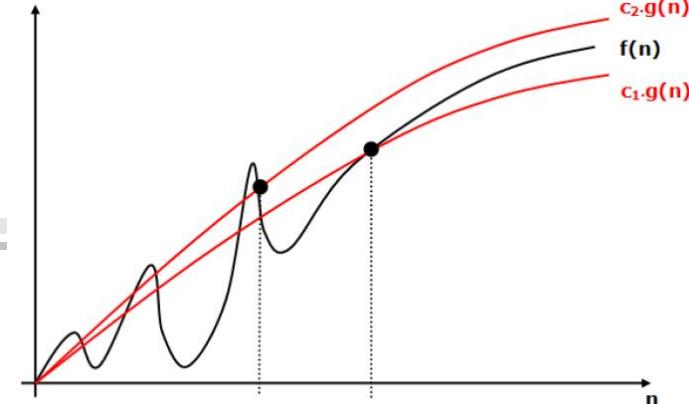
- Problem SAT
- Problem CNF-SAT
- Problem pokrycia wierzchołkowego grafu
- Problem kolorowania grafów
- Problem kliki
- Problem pokrycia zbiorami
- Problem sumy podzbioru
- Problem plecakowy
- Problem komiwojażera
- ... i wiele innych

Egzamin testowy z ASD

Pytanie do efektu W1/3



- **Grupa A:** Jeśli f i g są funkcjami określonymi w zbiorze liczb naturalnych o wartościach w zbiorze nieujemnych liczb rzeczywistych i c jest pewną stałą, to powyższy rysunek ilustruje:
 - A. ograniczanie od dołu funkcji f przez funkcję g ,
 - B. ograniczanie od góry funkcji f przez funkcję g ,
 - C. ograniczanie od dołu i od góry funkcji f przez funkcję g ,
 - D. żadna z powyższych możliwości.



- **Grupa B:** Jeśli f i g są funkcjami określonymi w zbiorze liczb naturalnych o wartościach w zbiorze nieujemnych liczb rzeczywistych oraz c_1 i c_2 są pewnymi stałymi, to powyższy rysunek ilustruje:
 - A. ograniczanie od dołu funkcji f przez funkcję g ,
 - B. ograniczanie od góry funkcji f przez funkcję g ,
 - C. ograniczanie od dołu i od góry funkcji f przez funkcję g ,
 - D. żadna z powyższych możliwości.

Pytanie do efektu W1/3

- **Grupa A:** Jeśli funkcja kosztu czasowego pewnego algorytmu iteracyjnego jest postaci $C(n)=6n^2+4n-2$ (gdzie n to rozmiar charakterystyczny zadania) i algorytm wykorzystuje zawsze pamięć w postaci 4 zmiennych, to złożoność obliczeniowa czasowa tego algorytmu jest rzędu:
 - A. $\Theta(1)$
 - B. $\Theta(n)$
 - C. $\Theta(n^2)$
 - D. $\Theta(n^3)$
- **Grupa B:** Jeśli funkcja kosztu czasowego pewnego algorytmu iteracyjnego jest postaci $C(n)=6n+3$ (gdzie n to rozmiar charakterystyczny zadania) i dla dowolnego n algorytm wykorzystuje pamięć w postaci 10 zmiennych, to złożoność obliczeniowa pamięciowa tego algorytmu jest rzędu:
 - A. $\Theta(1)$
 - B. $\Theta(n)$
 - C. $\Theta(n^2)$
 - D. $\Theta(n^3)$

Pytanie do efektu W1/3

- **Grupa A:** Która trójka poniższych określeń dotyczy tylko i wyłącznie klas złożoności obliczeniowej algorytmów:
 - A. sortujący, liniowy, kwadratowy,
 - B. zachłanny, NP-trudny, logarytmiczny,
 - C. NP-zupełny, wykładniczy, szybki,
 - D. kwadratowy, wykładniczy, NP-trudny.
- **Grupa B:** Jeśli złożoności obliczeniowe 3 algorytmów są odpowiednio rzędu: $\Theta(n^2)$, $\Theta(\log n)$, $\Theta(2^n)$, to są to algorytmy:
 - A. kwadratowy, logarytmiczny, wykładniczy,
 - B. wielomianowy, logarytmiczny, potęgowy,
 - C. kwadratowy, logarytmiczny, dwójkowy,
 - D. żadana z powyższych odpowiedzi nie jest do końca prawdziwa.

Pytanie do efektu W1/3

- **Grupa A:** Klasa problemów z punktu widzenia złożoności obliczeniowej zawierająca takie problemy decyzyjne lub optymalizacyjne, że dla każdego z nich nie jest znany algorytm rozwiązujący ten problem w czasie wielomianowym (mogą być znane rozwiązania wykładnicze) oraz dowolny problem należący do NP może być do niego zredukowany w czasie wielomianowym nazywa się:
 - klasą P problemów,
 - klasą NP problemów,
 - klasą problemów NP-trudnych,
 - klasą problemów NP-zupełnych.
- **Grupa B:** Klasa problemów z punktu widzenia złożoności obliczeniowej zawierająca problemy decyzyjne w których sprawdzenie poprawności określonego rozwiązania wymaga złożoności obliczeniowej wielomianowej nazywa się:
 - klasą P problemów,
 - klasą NP problemów,
 - klasą problemów NP-trudnych,
 - klasą problemów NP-zupełnych.

Pytanie do efektu W1/3

- **Grupa A:** Klasa problemów z punktu widzenia złożoności obliczeniowej zawierająca problemy decyzyjne, których znalezienie rozwiązania wymaga złożoności obliczeniowej wielomianowej nazywa się:
 - klasą P problemów,
 - klasą NP problemów,
 - klasą problemów NP-trudnych,
 - klasą problemów NP-zupełnych.
- **Grupa B:** Klasa problemów z punktu widzenia złożoności obliczeniowej zawierająca problemy decyzyjne, z których każdy problem należy do klasy NP, nie jest dla niego znany algorytm rozwiązujący ten problem w czasie wielomianowym oraz dowolny problem należący do NP może być do niego zredukowany w czasie wielomianowym nazywa się:
 - klasą P problemów,
 - klasą NP problemów,
 - klasą problemów NP-trudnych,
 - klasą problemów NP-zupełnych.

Pytanie do efektu W1/4

- **Grupa A:** Które z poniższych uszeregowan klas złożoności względem stopnia efektywności algorytmu od najbardziej efektywnych do najmniej efektywnych czasowo jest poprawne:
 - A. wykładniczy, wielomianowy, liniowy, kwadratowy
 - B. logarytmiczny, kwadratowy, wykładniczy
 - C. liniowy, NP-trudny, kwadratowy
 - D. liniowy, logarytmiczny, kwadratowy
- **Grupa B:** Które z poniższych uszeregowan rzędów złożoności obliczeniowej od najmniej efektywnych do najbardziej efektywnych czasowo jest poprawne:
 - A. $O(n^2)$, $O(\log n)$, $O(2^n)$
 - B. $O(2^n)$, $O(n^2)$, $O(n \log n)$
 - C. $O(n^2)$, $O(\log n)$, $O(1)$
 - D. $O(n^2)$, $O(n)$, $O(n \log n)$

Pytanie do efektu W1/4

- **Grupa A:** Który z poniższych zestawów inkluzji jest w pełni poprawny:
 - A. $NP \subset P$,
 $NP\text{-zupełne} \subset NP$
 $NP\text{-SPACE} \subset P\text{-SPACE}$
 - B. $P \subset NP$
 $NP \subset NP\text{-zupełne}$
 $P\text{-SPACE} \subset NP\text{-SPACE}$
 - C. $P \subset NP$,
 $NP\text{-zupełne} \subset NP\text{-trudne}$
 $NP \subset P\text{-SPACE}$
 - D. $P \subset P\text{-SPACE}$
 $NP\text{-trudne} \subset NP\text{-SPACE}$
 $P \subset P\text{-SPACE}$
- **Grupa B:** Który z poniższych zestawów inkluzji jest w pełni niepoprawny
 - A. $NP \subset P$,
 $NP \subset NP\text{-zupełne}$
 $P\text{-SPACE} \subset P$
 - B. $P \subset NP$
 $NP\text{-zupełne} \subset NP$
 $NP\text{-SPACE} \subset P\text{-SPACE}$
 - C. $NP \subset P\text{-SPACE}$
 $NP\text{-zupełne} \subset NP\text{-trudne}$
 $P \subset NP\text{-trudne}$
 - D. $P\text{-SPACE} \subset P$
 $NP\text{-trudne} \subset NP\text{-zupełne}$
 $NP \subset NP\text{-trudne}$

Pytanie do efektu W1/4

- **Grupa A:** Który z poniższych algorytmów jest pesymistycznie najefektywniejszy czasowo:
 - A. wyszukiwanie maksimum w nieuporządkowanej tablicy n-elementów
 - B. wyszukiwanie podanego elementu w uporządkowanej tablicy n-elementów
 - C. wypisanie wszystkich permutacji zbioru n-elementowego,
 - D. posortowanie tablicy n-elementów metodą przez wybór.
- **Grupa B:** Które z poniższych algorytmów jest pesymistycznie najmniej efektywy czasowo:
 - A. wyszukiwanie podanego elementu w nieuporządkowanej tablicy n-elementów,
 - B. wyszukiwanie podanego elementu w uporządkowanej tablicy n-elementów,
 - C. wyszukiwanie minimum w tablicy 2-wymiarowej $n \times n$ elementów,
 - D. posortowanie tablicy n-elementów metodą MergeSort.

$$\mathbf{R1: } f(n) = \begin{cases} 1 & \text{dla } n=1 \\ f(n-1) + 2n - 1 & \text{dla } n > 1 \end{cases}$$

$$\mathbf{R2: } C(n) = \begin{cases} 1 & \text{dla } n=1 \\ C\left(\frac{n}{2}\right) + n & \text{dla } n > 1 \end{cases}$$

$$\mathbf{R3: } C(n) = \begin{cases} 1 & \text{dla } n=1 \\ 2 \cdot C\left(\frac{n}{2}\right) + 1 & \text{dla } n > 1 \end{cases}$$

Pytanie do efektu W1/4

- **Grupa A:** Które z powyższych równań rekurencyjnych nie można traktować jako reprezentację złożoności obliczeniowej czasowej algorytmu z rekurencją prostą:

- A. R2
- B. R3
- C. R2 i R3
- D. wszystkie to rekurencja prosta.

- **Grupa B:** Które z powyższych równań rekurencyjnych można traktować jako reprezentację złożoności obliczeniowej pamięciowej algorytmu z rekurencją prostą:

- A. R1
- B. R1 i R2
- C. R1 i R3
- D. żadne

Pytanie do efektu W1/5

- **Grupa A:** Jeśli algorytm A1 jest algorymem iteracyjnym przeszukującym liniowo tablicę w poszukiwaniu pewnego elementu, a A2 jest algorymem z rekurencją rozgałęzioną przeszukującym tę samą tablicę w tym samym celu, to który z tych algorytmów ma wyższą złożoność obliczeniową?
 - na pewno algorytm A1,
 - na pewno algorytm A2,
 - mogą mieć taką samą złożoność,
 - trudno porównać, bo nie podano tych algorytmów.
- **Grupa B:** Czy w przypadku algorytmu rekurencyjnego przeszukiwania tablicy uporządkowanej celem znalezienie pewnego jej elementu jest sens wykorzystywać rekurencję rozgałęzioną?
 - tak, bo warto podzielić tablice np. na 2 części i osobno (rekurencyjnie) szukać elementu w obydwu częściach,
 - nie, bo po podzieleniu tablicy np. na 2 części wystarczy szukać elementu tylko w jednej z nich (rekurencja prosta),
 - w tym przypadku wybór rekurencji prostej czy rozgałęzionej nie ma znaczenia, bo złożoność obliczeniowa w obydwu przypadkach jest taka sama,
 - trudno porównać, bo nie podano o jakie konkretnie algorytmy chodzi.

Pytanie do efektu W1/5

- **Grupa A:** Jeśli dla mojego problemu znam algorytm, który ma wykładniczą złożoność obliczeniową pamięciową, to co mogę zrobić, aby móc rozwiązywać ten problem za pomocą komputera dla dużych rozmiarów problemu?
 - A. zaimplementować ten algorytm w postaci programu i go uruchomić,
 - B. dokupić jak najwięcej pamięci RAM do komputera,
 - C. użyć komputera z większą liczbą rdzeni,
 - D. **wymyśleć algorytm o niższej złożoności pamięciowej.**

- **Grupa B:** Jeśli dla mojego problemu znam algorytm, który ma wykładniczą złożoność obliczeniową czasową, to co mogę zrobić, aby móc rozwiązywać ten problem za pomocą komputera dla dużych rozmiarów problemu?
 - A. zaimplementować ten algorytm w postaci programu i go uruchomić,
 - B. użyć komputera z możliwie największą częstotliwością taktowania procesora szybkim zegarem dokupić jak najwięcej pamięci RAM do komputera,
 - C. użyć komputera z większą liczbą rdzeni,
 - D. **wymyśleć algorytm o niższej złożoności czasowej.**

Pytanie do efektu W1/5

- **Grupa A:** Jeśli algorytm A1 jest algorymem iteracyjnym przeszukującym liniowo tablicę w poszukiwaniu pewnego elementu, a A2 jest algorymem z rekurencją rozgałęzioną przeszukującym tę samą tablicę w tym samym celu, to który z tych algorytmów ma wyższą złożoność obliczeniową?
 - na pewno algorytm A1,
 - na pewno algorytm A2,
 - mogą mieć taką samą złożoność,
 - nie można o tym dyskutować, bo nie podano tych algorytmów.
- **Grupa B:** Czy w przypadku algorytmu rekurencyjnego przeszukiwania tablicy uporządkowanej celem znalezienia pewnego jej elementu jest sens wykorzystywać rekurencję rozgałęzioną?
 - tak, bo warto podzielić tablice np. na 2 części i osobno (rekurencyjnie) szukać elementu w obydwu częściach,
 - nie, bo po podzieleniu tablicy np. na 2 części wystarczy szukać elementu tylko w jednej z nich (rekurencja prosta),
 - w tym przypadku wybór rekurencji prostej czy rozgałęzionej nie ma znaczenia, bo złożoność obliczeniowa w obydwu przypadkach jest taka sama,
 - nie można o tym dyskutować, bo nie podano o jaki konkretnie algorytm chodzi.

Pytanie do efektu W2/3

- **Grupa A:** Podaj rzędy złożoności obliczeniowej następujących operacji: wstawiania do tablicy dynamicznej, wyszukiwania w tablicy haszującej, usuwania z drzewa binarnego.
 - A. $\Theta(n)$, $\Theta(\log n)$, $\Theta(\log n)$
 - B. $\Theta(n^2)$, $\Theta(n)$, $\Theta(n)$
 - C. $\Theta(1)$, $\Theta(1)$, $\Theta(\log n)$
 - D. $\Theta(1)$, $\Theta(1)$, $\Theta(n)$
- **Grupa B:** Podaj rzędy złożoności obliczeniowej następujących operacji: wstawiania do drzewa binarnego, wyszukiwania w tablicy dynamicznej uporządkowanej, usuwania z tablicy haszującej.
 - A. $\Theta(\log n)$, $\Theta(\log n)$, $\Theta(1)$
 - B. $\Theta(1)$, $\Theta(n)$, $\Theta(n)$
 - C. $\Theta(n)$, $\Theta(\log n)$, $\Theta(1)$
 - D. $\Theta(\log n)$, $\Theta(n)$, $\Theta(1)$

Pytanie do efektu W2/3

- **Grupa A:** Strukturą konkretną nadającą się do implementacji abstrakcyjnej listy jest:
 - A. tablica dynamiczna,
 - B. lista powiązana,
 - C. drzewo binarne,
 - D. tablica mieszaną.
- **Grupa B:** Strukturą konkretną nadającą się do implementacji zbioru jest:
 - A. tablica dynamiczna,
 - B. lista powiązana,
 - C. drzewo binarne,
 - D. tablica mieszaną.

Pytanie do efektu W2/3

- **Grupa A:** Metoda reprezentacji grafów polegająca na tym, że dla każdego wierzchołka grafu budujemy listę wierzchołków będących jego sąsiadami nazywa się:
 - A. metodą sąsiadów,
 - B. metodą macierzy sąsiedztwa,
 - C. metodą listy sąsiedztwa,
 - D. żadna z powyższych.
- **Grupa B:** Metoda reprezentacji grafów polegająca na tym, że budujemy macierz kwadratową, w której przechowujemy informację o ewentualnych krawędziach pomiędzy sąsiednimi wierzchołkami w grafie nazywa się:
 - A. metodą sąsiadów,
 - B. metodą macierzy sąsiedztwa,
 - C. metodą listy sąsiedztwa,
 - D. żadna z powyższych.

Pytanie do efektu W2/4

- **Grupa A:** Która z poniższych implementacji zbioru będzie miała najszybszą operację usuwania:
 - A. za pomocą tablicy dynamicznej,
 - B. za pomocą uporządkowanej tablicy dynamicznej,
 - C. za pomocą drzewa binarnego,
 - D. za pomocą tablicy mieszaną.
- **Grupa B:** Która z poniższych implementacji listy będzie miała najwolniejszą operację wyszukiwania elementów:
 - A. za pomocą tablicy mieszaną
 - B. za pomocą tablicy dynamicznej,
 - C. za pomocą uporządkowanej tablicy dynamicznej,
 - D. za pomocą drzewa binarnego.

Pytanie do efektu W2/4

- **Grupa A:** Która z poniższych implementacji listy będzie miała najszybszą operację wyszukiwania elementów:
 - A. za pomocą tablicy dynamicznej,
 - B. za pomocą uporządkowanej tablicy dynamicznej,
 - C. za pomocą listy powiązanej,
 - D. za pomocą uporzadkowanej listy powiązanej,
- **Grupa B:** Która z poniższych implementacji zbioru będzie miała naj wolniejszą operację usuwania elementów :
 - za pomocą tablicy dynamicznej,
 - za pomocą uporządkowanej tablicy dynamicznej,
 - za pomocą drzewa binarnego,
 - za pomocą tablicy mieszającej.

Pytanie do efektu W2/4

- **Grupa A:** Która z poniższych implementacji listy będzie miała najszybszą operację wstawiania elementów:
 - A. za pomocą tablicy dynamicznej,
 - B. za pomocą uporządkowanej tablicy dynamicznej,
 - C. za pomocą listy powiązanej,
 - D. za pomocą uporządkowanej listy powiązanej,
- **Grupa B:** Która z poniższych implementacji zbioru będzie miała naj wolniejszą operację wstawiania elementów:
 - za pomocą tablicy dynamicznej,
 - za pomocą uporządkowanej tablicy dynamicznej,
 - za pomocą drzewa binarnego,
 - za pomocą tablicy haszującej

Pytanie do efektu W2/5

- **Grupa A:** Jeśli do reprezentowania danych potrzeba strukturę o możliwie szybkim wyszukiwaniu i oszczędności pamięci, a szybkość wstawiania i usuwania oraz uporządkowanie elementów w strukturze nie są istotne, to do implementacji takiej struktury zalecane jest użycie:
 - A. tablicy dynamicznej,
 - B. listy powiązanej,
 - C. tablice mieszającej,
 - D. drzewa binarnego.
- **Grupa B:** Jeśli do reprezentowania danych potrzeba strukturę o możliwie szybkim wstawianiu (najlepiej rzędu $O(1)$) i oszczędności pamięci, a szybkość wyszukiwania i usuwania oraz uporządkowanie elementów w strukturze nie są istotne, to do implementacji takiej struktury zalecane jest użycie:
 - A. tablicy dynamicznej,
 - B. listy powiązanej,
 - C. tablice mieszającej,
 - D. drzewa binarnego.

Pytanie do efektu W2/5

- **Grupa A:** Jeśli do reprezentowania danych potrzeba strukturę o możliwie szybkim wstawianiu elementów i uporządkowaniu wewnętrznym elementów, a szybkość wyszukiwania i usuwania oraz oszczędność pamięci nie są istotne, to do implementacji takiej struktury zalecane jest użycie:
 - A. tablicy dynamicznej uporządkowanej,
 - B. listy powiązanej uporządkowanej,
 - C. tablicy haszującej,
 - D. drzewa binarnego.
- **Grupa B:** Jeśli do reprezentowania danych potrzeba oszczędną pamięciowo i uporządkowaną wewnętrznie strukturę, a szybkość wstawiania, wyszukiwania i usuwania nie są istotne, to do implementacji takiej struktury zalecane jest użycie:
 - A. tablicy dynamicznej uporządkowanej,
 - B. listy powiązanej uporządkowanej,
 - C. tablicy haszującej,
 - D. drzewa binarnego.

Pytanie do efektu W2/5

- **Grupa A:** Jeśli do reprezentowania danych potrzeba strukturę o możliwie szybkim wstawianiu i wyszukiwaniu (najlepiej rzędu $O(1)$), a szybkość usuwania oraz uporządkowanie elementów i oszczędność pamięci w strukturze nie są istotne, to do implementacji takiej struktury zalecane jest użycie:
 - tablice dynamicznej,
 - tablice dynamicznej uporządkowanej,
 - tablice mieszającej,**
 - drzewa binarnego.
- **Grupa B:** Jeśli do reprezentowania danych potrzeba strukturę o możliwie szybkim wstawianiu (najlepiej rzędu $O(1)$), a szybkość wyszukiwania i usuwania oraz uporządkowanie elementów w strukturze i oszczędność pamięci nie są istotne, to do implementacji takiej struktury zalecane jest użycie:
 - tablicy dynamicznej,**
 - listy powiązanej,**
 - tablice haszującej,**
 - drzewa binarnego.

Pytanie do efektu W3/3

- **Grupa A:** Która z poniższych metod konstrukcji algorytmów jest metodą dokładną:
 - A. metoda brutalnej siły
 - B. metoda wspinaczkowa,
 - C. programowanie dynamiczne,
 - D. metoda Monte Carlo.
- **Grupa B:** Która z poniższych metod konstrukcji algorytmów jest metodą aproksymacyjną:
 - A. algorytm zachłanny
 - B. metoda dziel i zwyciężaj
 - C. przeszukiwanie z nawrotami
 - D. błędzenie losowe

Pytanie do efektu W3/3

- **Grupa A:** Metoda rozwiązania problemu wyszukiwania minimalnego elementu z tablicy n-wymiarowej polegająca na wylosowaniu k elementów z tablicy i zwróceniu najmniejszego z wylosowanych jest metodą:
 - dokładną
 - aproksymacyjną,**
 - w pewnym sensie jest dokładną i aproksymacyjną
 - trudno powiedzieć
- **Grupa B:** Metoda rozwiązania problemu wyszukania elementu w tablicy n-elementowej poprzez liniowy przegląd wszystkich elementów tablicy jest metodą:
 - dokładną**
 - aproksymacyjną,
 - w pewnym sensie jest dokładną i aproksymacyjną
 - trudno powiedzieć

Pytanie do efektu W3/3

- **Grupa A:** Metoda sortowania tablic, która polega na wyszukianiu elementu mającego się znaleźć na zadanej pozycji i zamianie miejscami z tym, który jest tam obecnie, przy czym operacja ta jest wykonywana dla wszystkich indeksów sortowanej tablicy to metoda:
 - A. bąbelkowa,
 - B. **przez wybór**
 - C. przez wstawianie,
 - D. przez scalanie
- **Grupa B:** Metoda sortowania tablic, która polega na podzieleniu tablicy danych na dwie równe części, rekurencyjnym posortowaniu osobno każdej z nich oraz ich scaleniu to metoda:
 - A. bąbelkowa,
 - B. przez wybór,
 - C. przez wstawianie,
 - D. **żadna z powyższych.**

Pytanie do efektu W3/3

- **Grupa A:** Metoda sortowania tablic, która polega na wybraniu klucza osiowego, podzieleniu tablicy na dwie części (elementy nie większe oraz większe od klucza osiowego) oraz rekurencyjnie posortowanie osobno obydwu części tablicy to metoda:
 - A. szybkiego sortowania,
 - B. przez wybór
 - C. przez wstawianie,
 - D. przez scalanie
- **Grupa B:** Metoda sortowania tablic, która polega na tym, że kolejne elementy z sortowanej tablicy są ustawiane na odpowiednie miejsca docelowe, które jest wyszukiwane poprzez porównanie ustawianego elementu z kolejnymi elementami tablicy (idąc od prawej strony) póki nie napotka się elementu mniejszego lub nastąpi dojście do początku zbioru uporządkowanego to metoda:
 - A. wolnego sortowania,
 - B. przez wybór,
 - C. przez wstawianie,
 - D. żadna z powyższych.

Pytanie do efektu W3/3

- **Grupa A:** Metoda przeszukiwania grafów polegająca na sieganiu w grafie coraz dalej o ile jest to możliwe, a w przypadku braku możliwości sięgnięcia dalej, cofaniu się do wierzchołka z którego się wyszło w poprzednim kroku i dalszym sieganiu do nieodwiedzonych jeszcze wierzchołków (aż do odwiedzenia ich wszystkich) jest metodą:
 - przeszukiwania grafu wszerz,
 - przeszukiwanie grafu w głąb,**
 - programowania dynamicznego,
 - żadna z powyższych.
- **Grupa B:** Metoda przeszukiwania grafów polegająca na odwiedzeniu wszystkich wierzchołków osiągalnych z wierzchołka początkowego warstwami, gdzie dana warstwa oznacza wierzchołki tak samo odległe od zadanego wierzchołka jest metodą:
 - przeszukiwania grafu wszerz,
 - przeszukiwanie grafu w głąb,
 - programowania dynamicznego,
 - żadna z powyższych.

Pytanie do efektu W3/4

- **Grupa A:** Dolne ograniczenie na pesymistyczny czas sortowania polega na tym, że jeśli n jest rozmiarem sortowanego ciągu, to najlepszy algorytm sortowania z użyciem porównań musi mieć złożoność obliczeniową czasową rzędu:
 - $\Omega(\log n)$
 - $O(n \log n)$
 - $\Omega(n \log n)$
 - $O(n)$
- **Grupa B:** Zgodnie z twierdzeniem na temat dolnego ograniczenia na pesymistyczny czas sortowania, algorymem sortowania o optymalnej złożoności obliczeniowej jest algorytm:
 - A. szybkiego sortowania,
 - B. przez wybór
 - C. przez wstawianie,
 - D. **przez scalanie.**

Pytanie do efektu W3/4

- **Grupa A:** Które z poniższych stwierdzeń dotyczących metod konstrukcji algorytmów jest prawdziwe:
 - A. Programowanie dynamiczne jest efektywniejsze czasowo od metody dziel i zwycięzaj,
 - B. Metoda brutalnej siły daje większą pewność znalezienia optymalnego rozwiązania problemu od algorytmu zachłannego
 - C. Metoda Monte Carlo jest efektywniejsza czasowo od metody przeszukiwania z nawrotami
 - D. Zarówno błądzenie losowe, jak i Metoda Monte Carlo mogą czasem znaleźć rozwiązanie optymalne
- **Grupa B:** Które z poniższych stwierdzeń dotyczących metod konstrukcji algorytmów jest fałszywe:
 - A. jeśli algorytm zachłanny nie jest efektywny czasowo dla danego problemu, to nie ma sensu stosować metody Monte Carlo,
 - B. metoda błądzenia losowego jest rodzajem metody generowania i testowania,
 - C. oprócz metody brutalnej siły, żadna inna metoda nie potrafi znaleźć dokładnych rozwiązań,
 - D. metoda dziel i zwycięzaj jest odmianą programowania dynamicznego i jest od niego efektywniejsza pamięciowo.

Pytanie do efektu W3/4

- **Grupa A:** Jeśli do przeszukiwania grafów stosujemy metodę przeszukiwania w głąb oraz metodę przeszukiwania wszerz, to która z tych metod jest efektywniejsza czasowo (z punktu widzenia rzędu złożoności obliczeniowej):
 - przeszukiwanie grafu wszerz,
 - przeszukiwanie grafu w głąb,
 - obydwie są tak samo efektywne,**
 - trudno powiedzieć, bo to zależy od reprezentacji grafu.
- **Grupa B:** Jeśli w algorytmie przeszukiwania grafów wszerz zależy nam na tym, aby rozmiar zbioru wierzchołków miał jak najmniejszy wpływ na wzrost złożoności obliczeniowej pamięciowej, to do reprezentacji grafu używamy:
 - listy sąsiedztwa,
 - macierzy sąsiedztwa,
 - lista i macierz sąsiedztwa są tak samo efektywne,
 - reprezentacja grafu nie ma tutaj wpływu, bo o złożoności pamięciowej decyduje długość najdłuższej prostej ścieżki w grafie.

Pytanie do efektu W3/5

- **Grupa A:** Jeśli przestrzeń stanów dla danego problemu jest mała (np. do 10000 stanów) i zależy nam na rozwiązaniu dokładnym problemu, to należy stosować metody:
 - A. programowanie dynamiczne,
 - B. metodę Monte Carlo
 - C. metodę przeszukiwania z nawrotami,
 - D. algorytm zachłanny.
- **Grupa B:** Jeśli przestrzeń stanów dla danego problemu jest bardzo duża, ale nie zależy nam na rozwiązaniu dokładnym problemu, to należy stosować metody:
 - A. metodę brutalnej siły,
 - B. metodę Monte Carlo,
 - C. metodę dziel i zwyciężaj,
 - D. błędzenie losowe.

Pytanie do efektu W3/5

- **Grupa A:** Jeśli problem plecakowy decyzyjny dotyczy wyboru około 100 przedmiotów z 700 000 przedmiotów, to jaką metodę można efektywnie użyć do rozwiązania tego problemu:
 - A. programowanie dynamiczne,
 - B. metodę Monte Carlo
 - C. metodę przeszukiwania z nawrotami,
 - D. algorytm zachłanny.
- **Grupa B:** Jeśli problem komiwojażera dotyczy 1000 miast, to jaką metodę nie może być użyta do rozwiązania tego problemu:
 - A. metoda przeszukiwania z nawrotami,
 - B. metoda brutalnej siły (przeglądanie wszystkich permutacji poszczególnych miast),
 - C. algorytm zachłanny,
 - D. metoda Monte Carlo.

Pytanie do efektu W3/5

- **Grupa A:** Jaką metodę rozwiązania najlepiej zastosować w problemie wydawania reszty:
 - A. metodę Monte Carlo,
 - B. metodę przeszukiwania z nawrotami,
 - C. algorytm zachłanny,
 - D. żadna z powyższych się nie nadaje.
- **Grupa B:** Jaką metodę rozwiązania najlepiej zastosować w problemie przewiezienia wilka, kozy i kapusty:
 - A. metoda przeszukiwania z nawrotami,
 - B. metoda brutalnej siły (przeglądanie wszystkich permutacji z powtórzeniami o ustalonej długości),
 - C. metoda Monte Carlo,
 - D. żadna z powyższych się nie nadaje.

Pytanie do efektu W4/3

- **Grupa A:** Któż z poniższych własności nie należy dowodzić dla udowodnienia, że dany algorytm jest semantycznie poprawny:
 - A. własność symetri danych wejściowych i wyjściowych,
 - B. własność częściowej poprawności,
 - C. własność określoności,
 - D. własność stopu.
- **Grupa B:** Któż z poniższych metod jest metodą dowodzenia częściowej poprawności algorytmów:
 - A. metoda liczników iteracji,
 - B. metoda malejących wielkości,
 - C. metoda niezmienników iteracji,
 - D. żadna z powyższych.

Pytanie do efektu W4/3

- **Grupa A:** Warunek logiczny, który opisuje własność poprawnych danych na wejściu algorytmu nazywa się:
 - A. warunkiem rozpoczęcia,
 - B. **warunkiem początkowym,**
 - C. warunkiem końcowym,
 - D. warunkiem określoności.
- **Grupa B:** Warunek logiczny, który opisuje własność poprawnych wyników algorytmu oraz ich związek z danymi nazywa się:
 - A. warunkiem rozpoczęcia,
 - B. warunkiem początkowym,
 - C. **warunkiem końcowym,**
 - D. warunkiem stopu.

Pytanie do efektu W4/3

- **Grupa A:** Własność algorytmu polegająca na tym, że dla każdych danych wejściowych spełniających warunek początkowy, obliczenie algorytmu nie jest przerwane, gdyż wszystkie jego operacje są dobrze określone nosi nazwę:
 - A. warunku początkowego,
 - B. warunku określoności,
 - C. warunku stopu,
 - D. warunku końcowego.
- **Grupa B:** Własność algorytmu polegająca na tym, że dla każdych danych wejściowych spełniających warunek początkowy obliczenie jest skończone nosi nazwę:
 - A. warunku początkowego,
 - B. warunku określoności,
 - C. warunku końcowego,
 - D. żadne z powyższych.

Pytanie do efektu W4/4

- **Grupa A:** Metoda dowodu własności stopu algorytmu z pętlą polegająca na wyliczeniu ile razy zostanie wykona iteracja danej pętli i stwierdzeniu, że skoro jest to liczba skończona, to iteracje pętli kiedyś się zakończą, nazywa się:
 - A. metodą liczników iteracji,
 - B. metodą malejących wielkości
 - C. metodą niemienników iteracji,
 - D. żadna z powyższych.
- **Grupa B:** Metoda dowodu własności stopu algorytmu z pętlą opierająca się na założeniu, że jeśli w kolejnych iteracjach pętli dana wielkość jest coraz mniejsza i dodatkowo jest ograniczona od dołu, to pętla kiedyś się zakończy, co oznacza, że obliczenie będzie skończone, nazywa się:
 - A. metodą liczników iteracji,
 - B. metodą malejących wielkości
 - C. metodą niemienników iteracji,
 - D. żadna z powyższych.

Pytanie do efektu W4/4

- **Grupa A:** Która z poniższych trójek operacji zawiera tylko operacje, które zagrażają włączeniu określoności algorytmu (bez uwzględniania problemu przekroczenia zakresu wartości typu liczbowego):
 - A. dodawanie, odejmowanie, mnożenie,
 - B. mnożenie, dzielenie, pierwiastkowanie,
 - C. pierwiastkowanie, dzielenie, logarytmowanie,
 - D. logarytmowanie, mnożenie, dodawanie.
- **Grupa B:** Która z poniższych trójek operacji zawiera tylko operacje, które nie zagrażają włączeniu określoności algorytmu (bez uwzględniania problemu przekroczenia zakresu wartości typu liczbowego):
 - A. dodawanie, odejmowanie, mnożenie,
 - B. mnożenie, dzielenie, pierwiastkowanie,
 - C. pierwiastkowanie, dzielenie, logarytmowanie,
 - D. logarytmowanie, mnożenie, dodawanie.

Pytanie do efektu W4/5

- **Grupa A:** Jeśli program, który powinien wyliczać sumę elementów pewnego ciągu liczb całkowitych policzył iloczyn tych elementów, to znaczy że nie sprawdzono dla tego programu własności:
 - częściowej poprawności,
 - stopu,
 - określoności
 - żadnej z powyższych.
- **Grupa B:** Jeśli program, który sprawdza występowanie pewnego elementu w tablicy zawiesił się, to znaczy że nie sprawdzono dla tego programu własności:
 - częściowej poprawności,
 - stopu,
 - określoności
 - żadnej z powyższych

Pytanie do efektu W4/5

- **Grupa A:** Jeśli w programie komputerowym zdarzył się błąd polegający na próbie wyliczania pierwiastka kwadratowego z liczby ujemnej to znaczy że nie sprawdzono dla tego programu własności:
 - A. częściowej poprawności,
 - B. stopu,
 - C. określoności
 - D. żadnej z powyższych
- **Grupa B:** Jeśli program, który ma wyliczać maksimum elementów pewnego zbioru liczb całkowitych policzył minimum tego zbioru, to znaczy że nie sprawdzono dla tego programu własności:
 - A. częściowej poprawności,
 - B. stopu,
 - C. określoności
 - D. żadnej z powyższych

Pytanie do efektu W4/5

- **Grupa A:** Jeśli przy wyliczeniu sumy elementów ciągu program zwiesił się to znaczy, że nie sprawdzono dla tego programu własności:
 - częściowej poprawności,
 - stopu,
 - określoności
 - żadnej z powyższych
- **Grupa B:** Jeśli zdarzył się błąd dzielenia przez zero w programie komputerowym, to znaczy że nie sprawdzono dla tego programu własności:
 - częściowej poprawności,
 - stopu,
 - określoności,
 - żadnej z powyższych.

Sortowanie tablic

Sortowanie

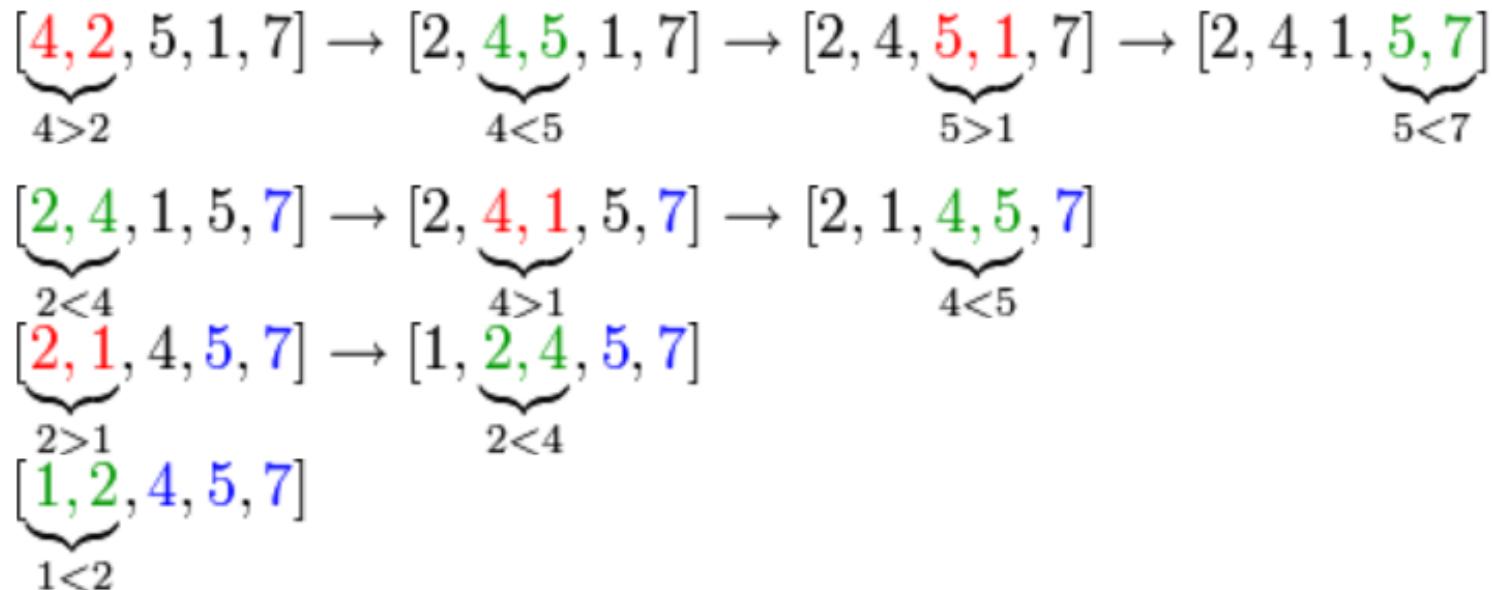
- Jeden z podstawowych problemów informatyki.
- Polega na uporządkowaniu zbioru danych względem pewnych cech charakterystycznych każdego elementu tego zbioru.
 - » Szczególnym przypadkiem jest sortowanie względem wartości każdego elementu, np. sortowanie liczb, słów itp.
- Algorytmy sortowania są stosowane w celu uporządkowania danych, umożliwienia stosowania wydajniejszych algorytmów
 - » Np. wyszukiwania i prezentacji danych w sposób czytelniejszy dla człowieka.

Sortowanie bąbelkowe

(ang. bubble sort)

- Polega na porównywaniu dwóch kolejnych elementów i zamianie ich kolejności, jeżeli zaburza ona porządek, w jakim się sortuje tablice.
 - » Takie przejście tablicy z porównywaniem i zamianą elementów wykonuje się od lewej do prawej strony tablicy.
 - » W wyniku całego przejścia jedna liczba na końcu tablicy (po prawej stronie) jest już posortowana (jest to analogia do bąbelków, które lecą do góry np. w akwariu).
- Następnie wykonywane są kolejne przejścia tablicy z porównywaniem i zamiana elementów, w wyniku których kolejne elementy stają się posortowane (każde następne przejście nie dotyczy elementów posortowanych w poprzednich przejściach).
- Sortowanie kończy się, gdy podczas kolejnego przejścia nie dokonano żadnej zmiany.
- Inspiracja naturą: *Jeśli w kolumnie z wodą wpuszcza się powietrze w pobliżu dna, to pęcherzyki powietrza mające mniejszą gęstość od wody, unoszone są do góry wskutek działania siły wyporu. Jeśli dodatkowo kolumna z wodą jest zamknięta od góry (powietrze nie może się wydostać), to powietrze stopniowo będzie wypełniać jej górną część.*

Przykład sortowania metodą bąbelkową ciągu [4, 2, 5, 1, 7]



- Każdy wiersz symbolizuje wypchnięcie kolejnego największego elementu na koniec ("wypłynięcie największego bąbelka").
- Czerwonym kolorem oznaczono pary, w których zamieniano elementy
- Zielonym kolorem oznaczono pary, w których nie zamieniano elementów („zamiana wypływającego bąbelka”)
- Niebieskim kolorem oznaczono końcówkę ciągu już posortowanego.

Klasa IntDynArrayWithSort jako „poligon” sortowania

- Implementacje metod sortowania zostały umieszczone w klasie IntDynArrayWithSort w pakiecie sort
- Jest to sortowanie w tablicy dynamicznej liczb całkowitych (takiej jak IntDynArray)
- Metody sortowania sortują wewnętrzną tablicę int [] table przy wykorzystaniu pola nElems przechowującego liczbę elementów tablicy

Procedura sortowania bąbelkowego

```
//Zamiana elementow tablicy  
private void swap(int one, int two)  
{  
    int temp = table[one];  
    table[one] = table[two];  
    table[two] = temp;  
}
```

```
public void bubbleSort() //Sortowanie bąbelkowe  
{  
    for (int i = nElems - 1; i > 0; i--) //petla zewnętrzna (malejaca)  
    {  
        for (int j = 0; j < i; j++) //petla wewnętrzna (rosnaca)  
        {  
            if (table[j] > table[j + 1]) //czy zla kolejnosc?  
            {  
                swap(j, j + 1); // no to zamiana  
            }  
        }  
    }  
}
```

Przykład dla ciągu: 4, 2, 5, 1, 7
i=4 Przed: 4 2 5 1 7 Po: 2 4 1 5 7
i=3 Przed: 2 4 1 5 7 Po: 2 1 4 5 7
i=2 Przed: 2 1 4 5 7 Po: 1 2 4 5 7
i=1 Przed: 1 2 4 5 7 Po: 1 2 4 5 7

Złożoność obliczeniowa sortowania bąbelkowego

- Algorytm wykonuje $n-1$ pętli wewnętrznych przejść, a w każdym przejściu wykonuje $n-k$ porównań (gdzie $k=1,2..n-1$ to numer bieżącego przejścia) wraz z ewentualnymi zamianami, przez co jego teoretyczna złożoność czasowa jest rzędu $O(n^2)$.
 - » Zadanie domowe: Udowodnić ten fakt rachunkowo (w dowodzie założyć pesymistycznie, że za każdym razem trzeba wykonać przestawienie elementów).
- Ze względu na to, że podczas działania algorytmu wykorzystywana jest stała liczba zmiennych, złożoność pamięciowa jest rzędu $O(1)$ (rzędu stałej).

Sortowanie przez wybór

- Jedna z prostszych metod sortowania, która polega na wyszukiwaniu elementu mającego się znaleźć na zadanej pozycji i zamianie miejscami z tym, który jest tam obecnie. Operacja jest wykonywana dla wszystkich indeksów sortowanej tablicy.
- Algorytm przedstawia się następująco:
 - » Wyszukaj minimalną wartość z tablicy spośród elementów od $i+1$ do końca tablicy
 - » Zamień wartość minimalną, z elementem na pozycji i

Procedura sortowania przez wybór

```
public void selectionSort() //Sortowanie przez wybór
{
    for (int i = 0; i < nElems - 1; i++)
    {
        int minPos = i;
        for (int j = i + 1; j < nElems; j++) //Wyszukanie elementu najmniejszego
        {
            if (table[j] < table[minPos]) minPos = j; // ...mamy nowe minimum
        }
        //Zamiana najmniejszego elementu z elementem na pozycji i
        swap(i, minPos);
    }
}
```

Przykład dla ciągu: 4, 2, 5, 1, 7
i=0 Przed: 4 2 5 1 7 Po: 1 2 5 4 7
i=1 Przed: 1 2 5 4 7 Po: 1 2 5 4 7
i=2 Przed: 1 2 5 4 7 Po: 1 2 4 5 7
i=3 Przed: 1 2 4 5 7 Po: 1 2 4 5 7

Przykład sortowania przez wybór dla ciągu [4, 2, 5, 1, 7]

i=0	Przed: 4 2 5 1 7	Po: 1 2 5 4 7
i=1	Przed: 1 2 5 4 7	Po: 1 2 5 4 7
i=2	Przed: 1 2 5 4 7	Po: 1 2 4 5 7
i=3	Przed: 1 2 4 5 7	Po: 1 2 4 5 7

- Każdy wiersz symbolizuje wyszukanie i wstawienie kolejnego minimalnego elementu
- W lewej części czerwonym kolorem oznaczono część nieposortowaną, a zielonym część posortowaną ciągu
- W prawej części zielonym kolorem oznaczono liczby właśnie wstawione, a czerwonym liczby, które posłużyły do wstawienia (przestawienie)

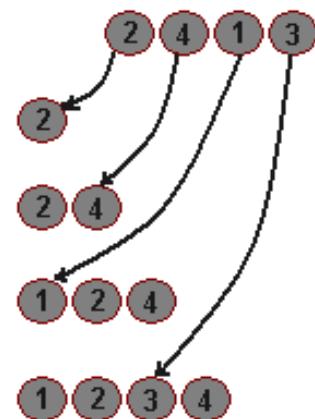
Złożoność obliczeniowa sortowania przez wybór

- Algorytm wykonuje $n-1$ wewnętrznych pętli połączonych z końcowym przestawieniem elementów, a w każdej z tych pętli wykonuje $n-k$ porównań (gdzie $k=1,2..n-1$ to numer bieżącej pętli wewnętrznej) wraz z ewentualnymi zamianami, przez co jego teoretyczna złożoność czasowa jest rzędu $O(n^2)$.
 - » Zadanie domowe: Udowodnić ten fakt rachunkowo (w dowodzie założyć pesymistycznie, że za każdym razem trzeba wykonać przestawienie elementów).
 - » Koszt czasowy tej metody sortowania jest niższy od kosztu sortowania bąbelkowego i przez wstawianie, choć rząd złożoności czasowej pozostaje taki sam
- Ze względu na to, że podczas działania algorytmu wykorzystywana jest stała liczba zmiennych, złożoność pamięciowa jest rzędu $O(1)$ (rzędu stałej).

Sortowanie przez wstawianie

(ang. Insert Sort, Insertion Sort)

- Jeden z najprostszych algorytmów sortowania, którego zasada działania odzwierciedla sposób w jaki ludzie ustawiają karty do gry
 - » Kolejne elementy wejściowe są ustawiane na odpowiednie miejsca docelowe.
- Przebieg algorytmu:
 1. Weź pierwszy element e (od lewej) ze zbioru nieposortowanego
 2. Wyciągnięty element porównuj z kolejnymi elementami zbioru posortowanego (idąc od prawej strony) póki nie napotkasz elementu mniejszego lub nie znajdziesz się na początku zbioru uporządkowanego. Ponadto, podczas porównywania przesuwaj elementy w prawo tak, aby zrobić miejsce dla elementu e .
 3. Wyciągnięty element wstaw w miejsce, gdzie skończyłeś porównywać.
 4. Jeśli jeszcze są elementy nieposortowane to przejdź do kroku 1, wpp. zakończ



Procedura sortowania przez wstawianie

```
public void insertionSort() //Sortowanie przez wstawianie
{
    for (int i = 1; i < nElems; i++) // i-pozycja pierwszego nieposortowanego
    {
        int e = table[i]; // kopiujemy wybrany element
        int j = i;          // zaczynamy przesuwanie od i
        while (j > 0 && table[j - 1] >= temp) // dopóki elementy są większe niż temp
        {
            table[j] = table[j - 1]; //przesuwamy elementy w prawo
            j--;                  // przesuwamy się w lewo
        }
        table[j] = e;      // wstawiamy wybrany element w nowym miejscu
    }
}
```

Przykład dla ciągu: 4, 2, 5, 1, 7
i=1 temp=2 Przed: 4 2 5 1 7 Po: 2 4 5 1 7
i=2 temp=5 Przed: 2 4 5 1 7 Po: 2 4 5 1 7
i=3 temp=1 Przed: 2 4 5 1 7 Po: 1 2 4 5 7
i=4 temp=7 Przed: 1 2 4 5 7 Po: 1 2 4 5 7

Przykład sortowania przez wstawianie dla ciągu [4, 2, 5, 1, 7]

i=1	e=2	Przed:	4 2 5 1 7	Po:	2 4 5 1 7
i=2	e=5	Przed:	2 4 5 1 7	Po:	2 4 5 1 7
i=3	e=1	Przed:	2 4 5 1 7	Po:	1 2 4 5 7
i=4	e=7	Przed:	1 2 4 5 7	Po:	1 2 4 5 7

- Każdy wiersz symbolizuje wstawienie kolejnego elementu nieposortowanego na swoje miejsce w części lewej posortowanej
- Czerwonym kolorem oznaczono część nieposortowaną, a zielonym część posortowaną ciągu

Złożoność obliczeniowa sortowania przez wstawianie

- Algorytm wykonuje $n-1$ pętli wewnętrznych przesunięć połączonych z końcowym wstawieniem, a w każdej z tych pętli wykonuje $k-1$ porównań (gdzie $k=1,2..n-1$ to numer bieżącej pętli wewnętrznej) wraz z ewentualnymi zamianami, przez co jego teoretyczna złożoność czasowa jest rzędu $O(n^2)$.
 - » Zadanie domowe: Udowodnić ten fakt rachunkowo (w dowodzie założyć pesymistycznie, że za każdym razem trzeba wykonać przestawienie elementów).
 - » Koszt czasowy tej metody sortowania jest niższy od kosztu sortowania bąbelkowego, choć rząd złożoności czasowej pozostaje taki sam
- Ze względu na to, że podczas działania algorytmu wykorzystywana jest stała liczba zmiennych, złożoność pamięciowa jest rzędu $O(1)$ (rzędu stałej).

Porównanie prostych algorytmów sortowania

- **Sortowanie bąbelkowe** jest bardzo nieefektywne czasowo (choć proste) i nie powinno być używane
- **Sortowanie przez wybór** minimalizuje liczbę zmian, ale liczba porównań jest taka sama jak w sortowaniu bąbelkowym (można stosować, gdy zamiana elementów jest dużo bardziej kosztowna od ich porównywania).
- **Sortowanie przez wstawianie** jest najlepszą propozycją z tych algorytmów, gdyż ma najmniejszy koszt czasowy (przegrywa jednak z bardziej złożonymi algorytmami – patrz dalej).

Sortowanie szybkie (ang. quicksort)

- Jeden z popularnych algorytmów sortowania działających z użyciem metody "dziel i zwyciężaj"
 - » Wynaleziony w 1962 przez Tony'ego Hoare'a
- Wyróżnić można trzy podstawowe kroki:
 1. Wybranie elementu rozdzielającego (tzw. klucza osiowego)
 2. Podzielenie tablicy na dwa fragmenty: do początkowego (pierwszego) przenoszone są wszystkie elementy nie większe od klucza osiowego, do końcowego (drugiego) wszystkie większe.
 3. Rekurencyjnie posortowanie osobno części początkowej i końcowej tablicy.
- Rekursja kończy się, gdy kolejny fragment uzyskany z podziału zawiera pojedynczy element, jako że jednoelementowa tablica nie wymaga sortowania.

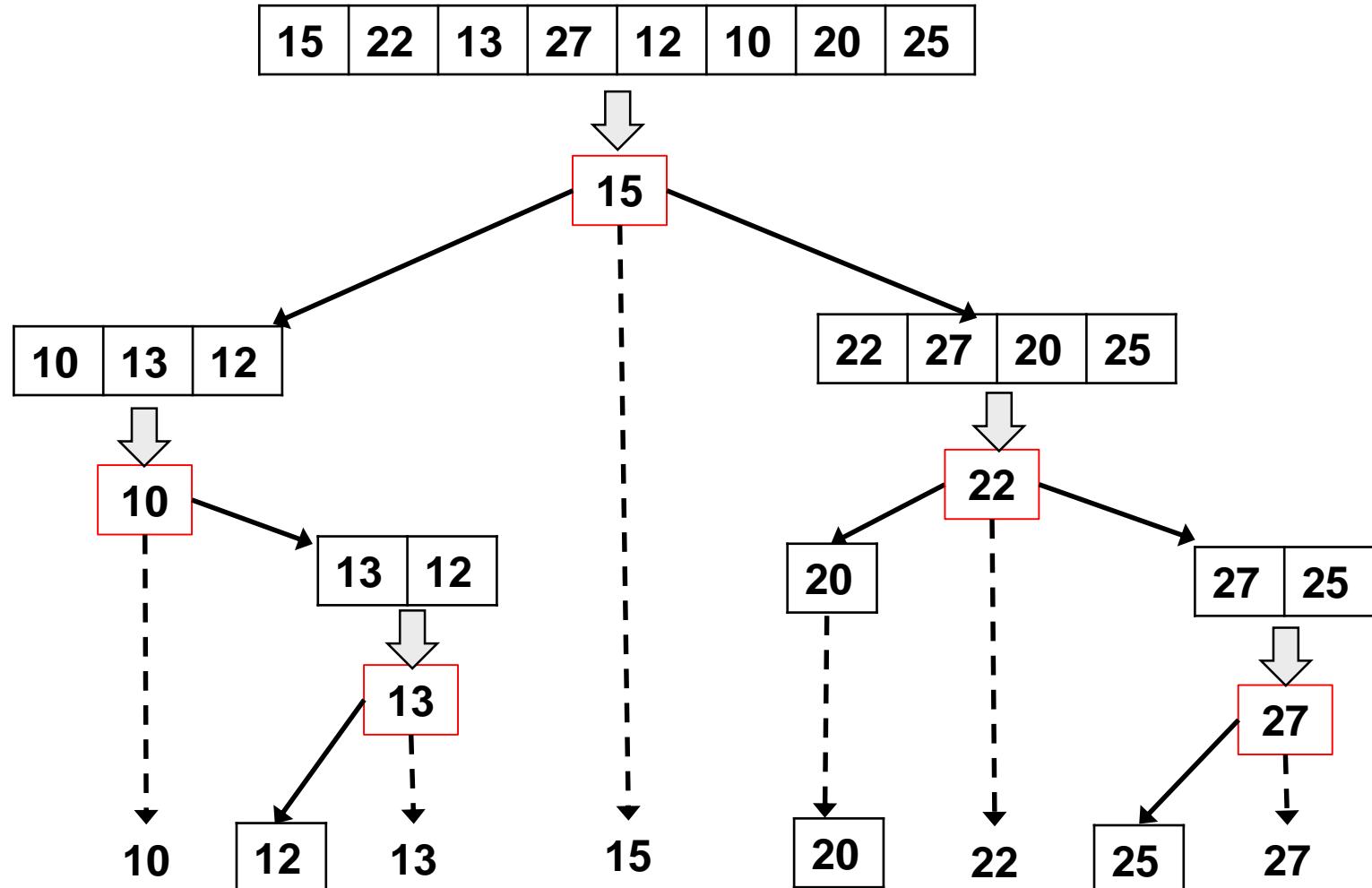
Procedura szybkiego sortowania

```
public void quickSort(int left,int right)
{
    if (left<right)
    {
        int p = table[left]; //Ustalenie klucza osiowego (jedna z metod)
        int s=left;
        for (int i=left+1; i<=right; i++)//Zbieranie elem. mniejszych od klucza
        {
            if (table[i]<p) { s=s+1; swap(s,i); }
        }
        swap(left,s); //Zamiana klucza osiowego z elem. wysunietym na prawo
        quickSort(left,s-1); //Posortowanie mniejszych od klucza osiowego
        quickSort(s+1,right); //Posortowanie wiekszych od klucza osiowego
    }
}
```

Na początku:

- » $\text{left}=0$ (numer lewego skrajnego elementu w tablicy)
- » $\text{right} = \text{rozmiar tablicy}-1$ (numer prawego skrajnego elementu w tablicy)

Przykład szybkiego sortowania tablicy 8-elementowej



Metody wyboru elementu osiowego

- Dwa sprzeczne cele: chęć podziału *na dwie równoliczne części* oraz oczekiwanie *bardzo dużej szybkości* wyznaczenia elementu osiowego.
- Rozważa się 4 następujące metody:
 - » Wybieranie **zawsze skrajnego elementu tablicy** (co dla danych już uporządkowanych daje katastrofalną złożoność $O(n^2)$).
 - » Wybór **losowego elementu tablicy** (sprawdza prawdopodobieństwo zajścia najgorszego przypadku do wartości zaniedbywalnie małych)
 - » Wybór **środkowego elementu z trzech liczb**: skrajny lewy, element tablicy, środkowy element tablicy i skrajny prawy element tablicy.
 - » Wybór **medianę elementów tablicy** (liczba o takiej własności, że powyżej i poniżej znajduje się jednakowa liczba elementów ciągu – rozwiążanie niepraktyczne ze względu na dużą złożoność wyznaczenia mediany)

Złożoność obliczeniowa szybkiego sortowania

- W przypadku typowym algorytm ten jest najszybszym algorytmem sortującym z klasy złożoności obliczeniowej czasowej (średnia złożoność obliczeniowa czasowa jest rzędu $n \log n$) - stąd pochodzi jego popularność w zastosowaniach.
 - » Trzeba jednak pamiętać, iż w pewnych sytuacjach (zależnych od sposobu wyboru klucza osiowego oraz niekorzystnego ułożenia danych wejściowych) klasa złożoności obliczeniowej tego algorytmu może się degradować do n^2 (pesymistyczna złożoność obliczeniowa czasowa tego algorytmu jest $O(n^2)$), co więcej, poziom wywołań rekurencyjnych może spowodować przepełnienie stosu i przerwanie programu.
 - Dla pewności, można zatem rozważyć inny algorytm, który daje pewność złożoności czasowej $O(n \log n)$, choć zwiększa złożoność pamięciową (np. sortowanie przez scalanie – patrz dalej)
- Ze względu na wywołanie rekurencji, złożoność pamięciowa jest rzędu $O(\log n)$ (istnieją wersje nierekurencyjne)

Sortowanie przez scalanie

(ang. merge sort)

- Rekurencyjny algorytm sortowania danych, stosujący metodę dziel i zwycięża, którego odkrycie algorytmu przypisuje się Johnowi von Neumannowi.
- Wyróżnić można trzy podstawowe kroki:
 - » Podziel zestaw danych na dwie równe części.
 - » Zastosuj sortowanie przez scalanie dla każdej z nich oddziennie, chyba że pozostał już tylko jeden element;
 - » Połącz posortowane podciągi w jeden.
- Algorytm jest szczególnie jest przydatny zwłaszcza przy danych dostępnych sekwencyjnie (po kolei, jeden element na raz), na przykład w postaci listy jednokierunkowej (tj. łączonej jednostronnie) albo pliku sekwencyjnego

Procedura sortowania przez scalanie

```
public void mergeSort(int left, int right)
{
    if (left < right)
    {
        int mid = (left + right) / 2; //Wyliczenie srodka
        mergeSort(left, mid); //Sortowanie lewej czesci
        mergeSort(mid + 1, right); //Sortowanie prawej czesci
        merge(left, mid, right); //Scalanie posortowanych czesci
    }
}
```

Na początku:

- $\text{left}=0$ (numer lewego skrajnego elementu w tablicy)
- $\text{right} = \text{rozmiar tablicy}-1$ (numer prawego skrajnego elementu w tablicy)

Scalanie

- Procedura scalania dwóch ciągów A[1..n] i B[1..m] do ciągu C[1..m+n]
- Utwórz zmienne iterujące i,j oraz ustaw je na początki ciągów A i B, tj:
 $i=1, j=1$
- Powtarzaj aż wszystkie wyrazy z A i B trafią do C:
 - » Jeżeli ciąg A wyczerpany ($i>n$), dołącz pozostałe elementy ciągu B do C i zakończ pracę.
 - » Jeżeli ciąg B wyczerpany ($j>m$), dołącz pozostałe elementy ciągu A do C i zakończ pracę.
 - » Jeżeli $A[i] \leq B[j]$ dołącz $A[i]$ do C i zwiększ i o jeden, w przeciwnym przypadku dołącz $B[j]$ do C i zwiększ j o jeden
- Scalenie wymaga $O(n+m)$ operacji porównań elementów i wstawienia ich do tablicy wynikowej (jedna pętla).

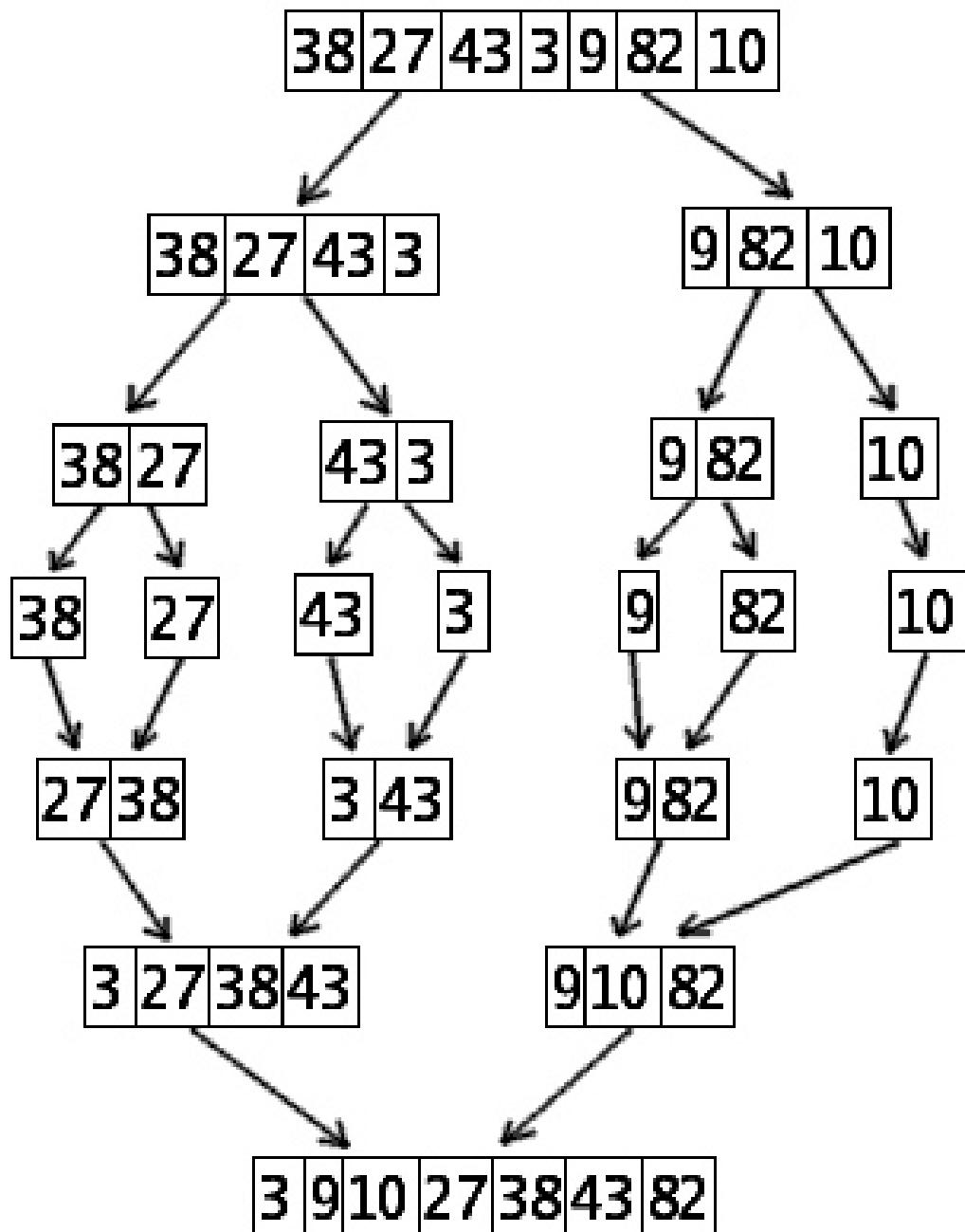
```

void merge(int left,int mid,int right) //łącz fragment od l do mid z fragm. od mid+1 do right
{
    int [] tab = new int[right-left+1]; //Tworz tablice pomocnicza
    int k = 0; //Indeks w scalanej tablicy
    int i = left; //Indeks w pierwszym fragmencie do scalenia
    int j = mid+1; //Indeks w drugim fragmencie do scalenia
    while ((i<mid+1) || (j<right+1)) //...az do konca obydwu scalanych fragmentow
    {
        if (i==mid+1) //Jesli pierwszy fragment juz jest scalony...
        {
            tab[k] = table[j]; k++; j++; //...kopiuj kolejna liczbe z drugiego fragmentu
        }
        else if (j == right + 1) //Jesli drugi fragment juz jest scalony...
        {
            tab[k] = table[i]; k++; i++; //...kopiuj kolejna liczbe z pierwszego fragmentu
        }
        else //Scalanie elementow z obydwu fragmentow
        {
            if (table[i] <= table[j]) //Element z pierwszego fragmentu
            {
                tab[k] = table[i]; i++; //... i jest scalany jako pierwszy
            }
            else // table[j]<table[i], czyli element z drugiego fragmentu jest mniejszy
            {
                tab[k] = table[j]; j++; //... i jest scalany jako pierwszy
            }
            k++; //Przechodzimy do nastepnej pozycji scalanej tablicy
        }
    }
    int l = left; //Przepisanie zawartosci tablicy pomocniczej do tablicy table
    for (i=0; i<tab.length; i++) { table[l] = tab[i]; l++; }
}

```

Procedura scalania dwóch fragmentów tablicy

Przykład sortowania przez scalanie tablicy 7-elementowej



Złożoność obliczeniowa sortowania przez scalanie

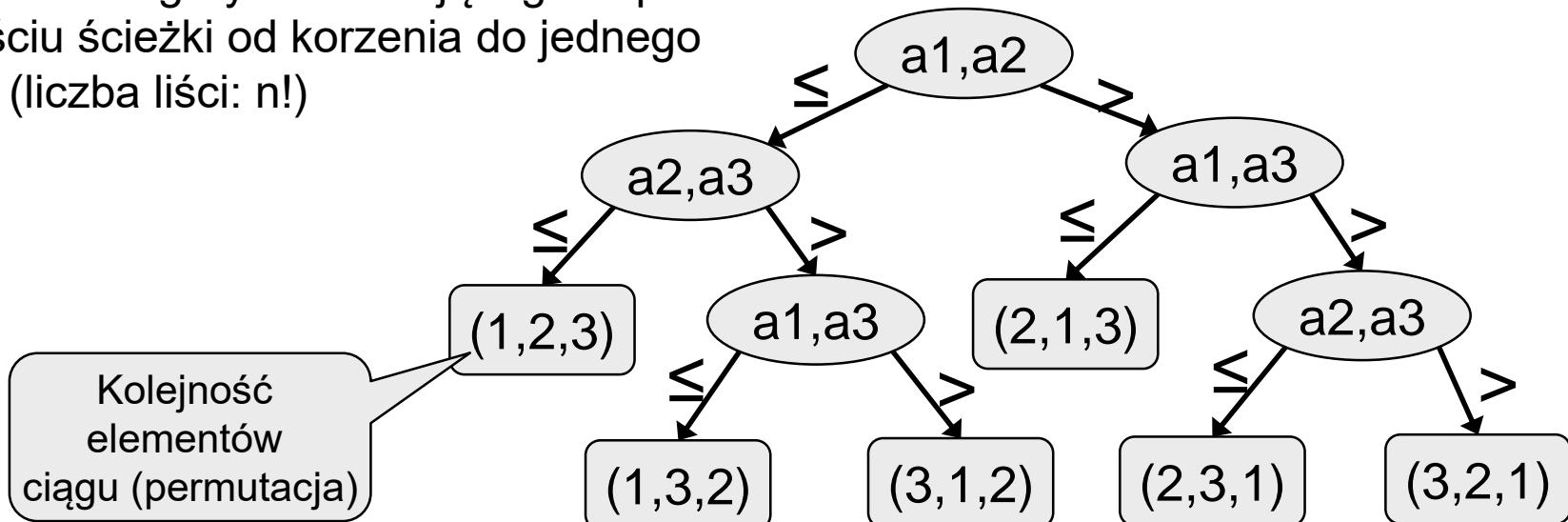
- Zakładając bez straty ogólności, że długość ciągu, który mamy posortować jest potęgą liczby 2 ($n=2^k$), złożoność czasową sortowania przez scalanie możemy formalnie opisać za pomocą równania rekurencyjnego:

$$T(n) = \begin{cases} 1 & \text{dla } n = 1 \\ 2T\left(\frac{n}{2}\right) + n & \text{wpp.} \end{cases}$$

- Rozwiązaniem tego równania jest: $T(n) = 2n \log n$
- Zatem złożoność czasowa jest rzędu: $O(n \log n)$
- Ze względu na istnienie tablicy pomocniczej przy scalaniu ciągów oraz użycie rekurencji, złożoność obliczeniowa pamięciowa jest także rzędu $O(n \log n)$

Dolne ograniczenie na pesymistyczny czas sortowania (1)

- Każdy algorytm sortujący ciąg za pomocą porównań można reprezentować w postaci drzewa decyzyjnego, które przedstawia porównania wykonywane przez algorytm sortujący dla danego ustalonego rozmiaru ciągu.
 - » Przykład drzewa decyzyjnego dla pewnej metody sortowania ciągu 3-elementowego: (a₁,a₂,a₃) (użyto 5 porównań)
- Z każdym liściem tego drzewa jest związana permutacja elementów ciągu
- Wykonanie algorytmu sortującego odpowiada przejściu ścieżki od korzenia do jednego z liści (liczba liści: n!)



Dolne ograniczenie na pesymistyczny czas sortowania (2)

- **Twierdzenie:** Każde drzewo odpowiadające algorytmowi poprawnie sortującemu n elementów, ma wysokość rzędu $\Omega(n \log n)$

Dowód: Niech D jest drzewem decyzyjnym o wysokości h odpowiadającym algorytmowi sortującemu n elementów.

Liczba permutacji n elementów wynosi $n!$, a każda permutacja odpowiada jednoznacznie pewnemu uporządkowaniu elementów.

Zatem drzewo ma co najmniej $n!$ liści. Z drugiej strony takie drzewo nie może mieć więcej jak 2^h liści. Czyli $n! \leq 2^h$, a po zlogarytmowaniu:

$h \geq \log(n!)$. Tymczasem ze wzoru Stirlinga wiadomo, że:

$$n! > \left(\frac{n}{e}\right)^n, \text{ gdzie } e=2.71828 \text{ (liczba Eulera). Zatem:}$$

$$h \geq \log\left(\frac{n}{e}\right)^n = n \log n - n \log e = \Omega(n \log n) \quad \square$$

Wnioski

- Najlepszy algorytm sortowania z użyciem porównań musi mieć złożoność obliczeniową czasową co najmniej rzędu $\Omega(n \log n)$, gdzie n jest rozmiarem sortowanego ciągu
- Algorytm sortowania przez scalanie, posiadający złożoność obliczeniową czasową rzędu $O(n \log n)$ jest przedstawicielem algorytmów sortowania o optymalnej złożoności obliczeniowej (nie istnieją algorytmy o niższym rzędzie złożoności obliczeniowej czasowej)

Sortowanie w czasie liniowym

- Jest możliwe, jeśli wiemy coś więcej o sortowanym ciągu.
- Przypadek klasyczny: Załóżmy, że elementami sortowanej tablicy T o rozmiarze n są liczby całkowite z przedziału $0..k-1$ dla pewnego naturalnego k .

Sortowanie kubełkowe

- Sortujemy tablicę T o rozmiarze n , której elementy są liczbami całkowitymi z przedziału $0..k-1$ dla pewnego naturalnego k .
- Bierzemy k kubełków $B[0], \dots, B[k-1]$.
- Do kubełka $B[i]$ wrzucamy wszystkie elementy z tablicy T o wartości równej i .
- Następnie wypisujemy zawartości kubełków, poczynając od kubełka $B[0]$ i skończywszy na kubełku $B[k-1]$.
- Kubełki reprezentujemy jako listy jednokierunkowe.
- Zarówno wrzucenie elementów do kubełków jak i końcowe wypisanie elementów ma złożoność liniową względem n .
- Złożoność pamięciowa także jest liniowa względem n , gdyż wszystkie elementy tablicy T muszą znaleźć swoje miejsce w kubełkach.

Sortowanie przez zliczanie

- Sortujemy tablicę T o rozmiarze n, której elementy są liczbami całkowitymi z przedziału 0..k-1 dla pewnego naturalnego k.
- Sortowanie przez zliczanie polega na zliczeniu, ile razy dana liczba występuje w tablicy, którą mamy posortować.
 - » Następnie wystarczy utworzyć nowy ciąg, korzystając z danych zebranych wcześniej
- Do sortowania jest potrzebna dodatkowa pamięć M w postaci k-elementowej tablicy, gdzie wartość zapisana pod danym indeksem i w tablicy M odpowiada liczbie wystąpień w sortowanej tablicy elementu i ($0 < i < k - 1$).
- Ponieważ dla każdego elementu z tablicy T wiemy pod jakim indeksem znajduje się w tablicy M komórka zliczająca liczbę jego wystąpień, zliczanie wystąpienia elementów tablicy T ma złożoność liniową względem n. Utworzenie nowego ciągu w oparciu o dane zebrane w tablicy M także jest liniowe.
- Złożoność pamięciowa również jest liniowa, ale względem k.

Sortowanie tablicy w praktyce programisty języka Java

- W praktyce codziennej najczęściej sortuje się dane umieszczone obiekcie klasy `ArrayList`
- Dla posortowania takich danych należy zdefiniować specjalną klasę komparatora, która dostarcza metody porównującej `compare` (zwraca 1, -1 lub 0)
- Sortowanie odbywa się z użyciem metody `sort` zdefiniowanej w klasie standardowej `Collections`
- W wyniku sortowania dane są przestawione bezpośrednio w tablicy dynamicznej

Klasa komparatora dla liczb całkowitych (porządek malejący)

```
import java.util.Comparator;

public class NumberComparator implements Comparator<Integer>
{
    public int compare(Integer i1, Integer i2)
    {
        if (i1.intValue() < i2.intValue()) return 1;
        else
            if (i1.intValue() > i2.intValue()) return -1;
        else
            return 0;
    }
}
```

Sortowanie zawartości obiektu klasy ArrayList

```
public static void main(String[] args)
{
    ArrayList<Integer> numberList = new ArrayList<Integer>();
    numberList.add(new Integer(4));
    numberList.add(new Integer(2));
    numberList.add(new Integer(5));
    numberList.add(new Integer(1));
    numberList.add(new Integer(7));

    for (int i = 0; i < numberList.size(); i++)
        System.out.print(numberList.get(i).toString() + " ");
    System.out.println();

    //Sortowanie metoda QuickSort
    Collections.sort(numberList, new NumberComparator());

    for (int i = 0; i < numberList.size(); i++)
        System.out.print(numberList.get(i).toString() + " ");
    System.out.println();
}
```

Dostępne
w pakiecie:
sort

4 2 5 1 7
7 5 4 2 1

Abstrakcyjne struktury danych

Czym są struktury danych?

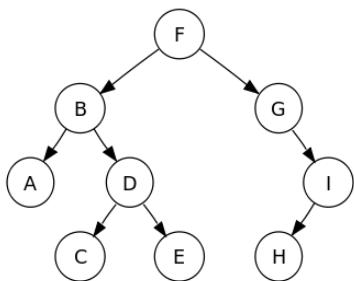
- Struktura danych (ang. data structure) to sposób uporządkowania informacji (danych).
 - » Na strukturach danych operują algorytmy.
- Dwa sposoby spojrzenia na struktury danych
 - » **abstrakcyjne struktury danych** – koncepcyjne (twórcze, myślowe) ułożenie danych (informacji) w celu rozwiązania jakiegoś problemu algorytmicznego
 - Przykłady: stos, kolejka, lista, słownik, drzewo, graf
 - » **konkretnie struktury danych** – ułożenie danych w pamięci komputera z użyciem zmiennych konkretnych typów danych oraz rozmaitych technik programowania (np. reprezentowanie danych przy wykorzystaniu złożonych typów danych jak tablice, rekordy lub klasy, realizacja powiązań pomiędzy danymi przy zastosowaniu wskaźników lub referencji do porcji danych)
 - Przykłady: tablica dynamiczna, lista powiązana, drzewo binarne, tablica haszująca
- Konkretnie struktury danych służą do implementacji abstrakcyjnych struktur danych

Rodzaje abstrakcyjnych struktur danych

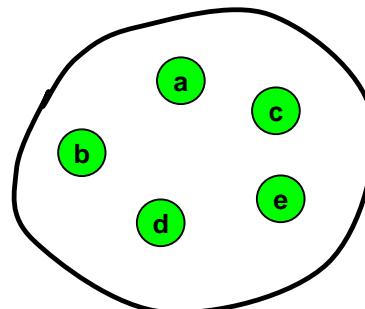
Lista



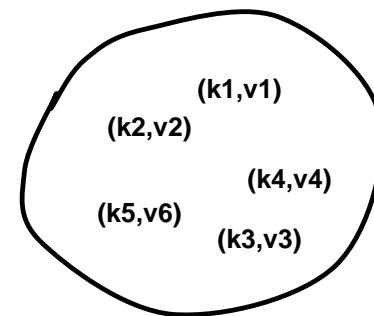
Drzewo



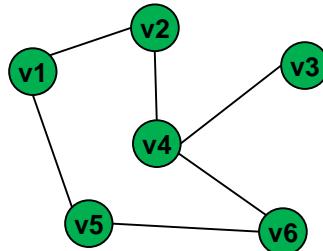
Zbiór



Słownik



Graf



Lista

- Lista w sensie abstrakcyjnym, to ciąg ponumerowanych elementów $L=[x_1, \dots, x_n]$.
- Skrajne elementy listy x_1 i x_n nazywają się **końcami listy** (lewy i prawy).
- Liczba $|L|=n$ jest **długością (rozmiarem) listy L**.
- Szczególnym przypadkiem listy jest **lista pusta**: $L=()$

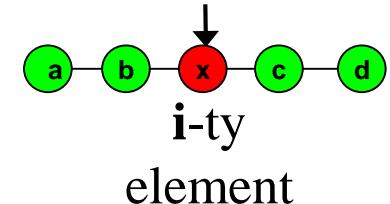
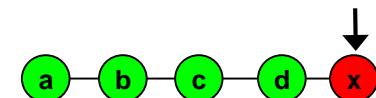
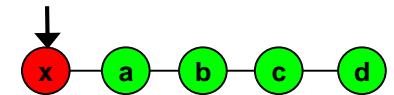


Operacje na listach

- Wstawianie elementu na listę
- Usuwanie elementu z listy
- Wyszukiwanie elementu na liście
- Przeglądanie elementów z listy

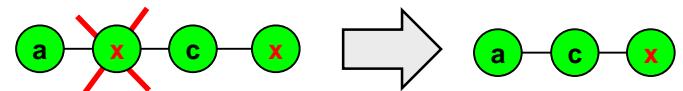
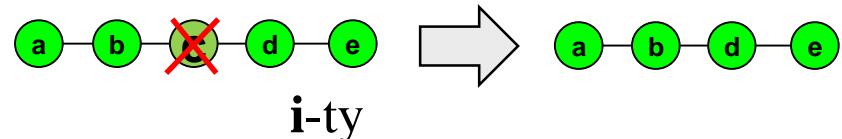
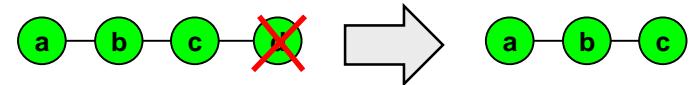
Typowe operacje wstawiania elementu na listę

- **AddFirst(L,x)** – wstawienie elementu **x** na lewy koniec listy L (na początek listy)
- **AddLast(L,x)** – wstawienie elementu **x** na prawy koniec listy L (na koniec listy)
- **Add(L,x,i)** – wstawienie elementu **x** do listy L na miejsce elementu o numerze i (dotychczasowe elementy od pozycji i do końca listy są przesuwane w prawo)



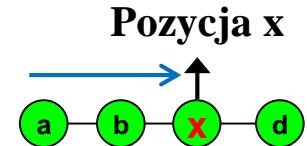
Typowe operacje usuwania elementu z listy

- **RemoveFirst(L,x)** – usunięcie elementu na lewym końcu listy L
- **RemoveLast(L,x)** – usunięcie elementu na prawym końcu listy L
- **Remove(L,i)** – usunięcie elementu o numerze i z listy L (dotychczasowe elementy od pozycji i do końca listy są przesuwane w lewo)
- **Remove(L,x)** – usunięcie elementu x z listy L (chodzi o pierwsze wystąpienie elementu x).
- **UWAGA:** Często usunięty element może być zwracany jako wynik operacji, ale nie jest to konieczne.



Wyszukiwanie elementu na liście

- **Find(L, x)** – wyszukanie elementu x na liście L
 - » Zwracany jest pozycja (numer) elementu x na liście albo liczba -1 symbolizująca brak elementu na liście
- **Przeglądanie listy a wyszukiwanie elementu na liście**
 - » Przeglądanie listy wymaga dotarcia do każdego elementu listy celem jego przetworzenia (np. wypisania na ekran)
 - » Wyszukiwanie **nie musi oznaczać przeglądania całej listy** (np. wyszukiwanie binarne)



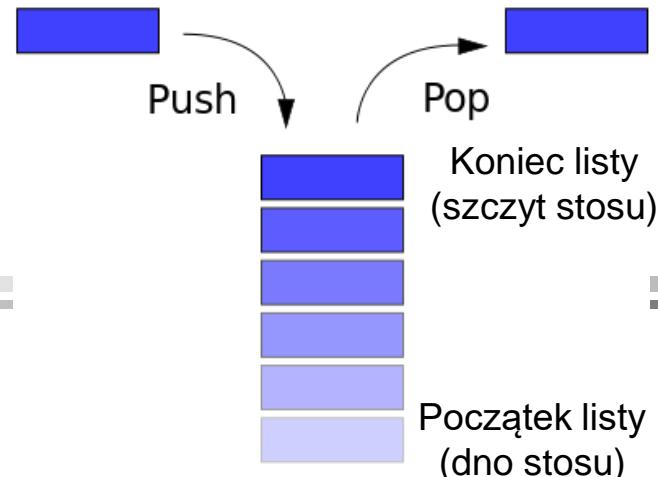
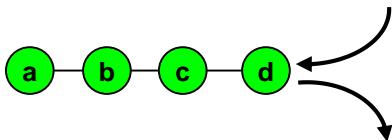
Typowe operacje potrzebne do przeglądania elementów na liście różnymi sposobami (dla różnych typów list)

- **IsEmpty(L)** – sprawdzenie czy lista jest pusta
 - » Zwracane jest **true** gdy lista jest pusta, wpp. zwracane jest **false**
- **Size(L)** – pobranie rozmiaru listy L
- **GetFirst(L)** – pobranie elementu pierwszego listy
 - » Zwracany jest element pierwszy lub wartość NULL jeśli lista jest pusta
- **GetLast(L)** – pobranie elementu ostatniego listy
 - » Zwracany jest element ostatni lub wartość NULL jeśli lista jest pusta
- **Get(L,i)** – pobranie elementu listy o numerze i
 - » Zwracany jest element lub wartość NULL, jeśli brak jest elementu o numerze i

Trzy typy list ze względu na różne możliwości dostępu do elementów i przeglądania listy

- **Listy bez możliwości przeglądania listy** z dostępem tylko do **pierwszego** i/lub **ostatniego** elementu listy (przeglądanie listy wymaga usuwania elementów z listy)
 - » Np. stos, kolejka, kolejka priorytetowa
- Listy z bezpośrednim dostępem tylko do **pierwszego** i/lub **ostatniego** elementu listy, ale **z możliwością liniowego przeglądania listy** za pomocą operacji GetNext lub/i GetPrevious (dostęp do wszystkich elementów podczas przeglądania listy)
 - » Np. lista jednokierunkowa (jednostronna i dwustronna) i lista dwukierunkowa
- **Listy z bezpośredniem dostępem do każdego elementu listy** poprzez podanie jego numeru
 - » Np. lista tablicowa

Stos



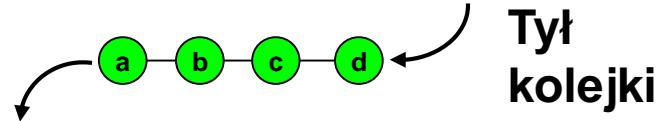
- Stos (ang. stack) – rodzaj listy w której nowe dane dopisywane są na końcu listy oraz z końca listy dane są pobierane do dalszego przetwarzania
 - » Koniec listy (tam gdzie wstawiono ostatnio element) jest tutaj nazywany **szczytem stosu**
 - » Idę stosu danych można zilustrować jako stos położonych jedna na drugiej książek – nowy egzemplarz kładzie się na wierzch stosu i z wierzchu stosu zdejmuje się kolejne egzemplarze.
 - » Elementy stosu poniżej szczytu można obejrzeć przez ściągnięcie elementów nad nimi.
- Operacje na stosie:
 - » **IsEmpty(L)** – sprawdzenie, czy lista (stos) jest pusta
 - » **AddLast(L,x)** – wstawienie elementu **x** na prawy koniec listy
 - » Operacja zwyczajowo zwana dla stosu: **push** (wypchnąć na stos)
 - » **GetLast(L)** – pobranie informacji o ostatnim elemencie listy
 - » Operacja zwyczajowo zwana dla stosu: **peek** (zerknąć na stos)
 - » **RemoveLast(L,x)** – usunięcie elementu na prawym końcu listy
 - » Operacja zwyczajowo zwana dla stosu: **pop**

Zastosowania stosu

- Stos jest używany w systemach komputerowych na wszystkich poziomach funkcjonowania systemów informatycznych.
- Stosowany jest przez procesory do chwilowego zapamiętywania rejestrów procesora
- Do przechowywania zmiennych lokalnych
- Do konstrukcji algorytmów.

Kolejka

Przód
kolejki



- Kolejka (ang. queue) – rodzaj listy, w której nowe dane dopisywane są na końcu listy, a z początku listy pobierane są dane do dalszego przetwarzania.
 - » Koniec listy (tam gdzie wstawiono ostatnio element, prawy koniec listy) jest tutaj nazywany **tyłem kolejki** (ang. back)
 - » Początek listy (tam gdzie pobierane są elementy) jest tutaj nazywany **przodem kolejki** (ang. front)
 - » Element kolejki za przodem kolejki obejrzeć przez usunięcie z kolejki elementów przed nim.
- Operacje na kolejce:
 - » **IsEmpty(L)** – sprawdzenie, czy lista (kolejka) jest pusta
 - » **AddLast(L,x)** – wstawienie elementu **x** do listy na tył kolejki (na prawy koniec listy)
 - » Operacja zwyczajowo zwana dla kolejki: **insert**
 - » **GetFirst(L)** – pobranie informacji o pierwszym elemencie listy, który jest na początku kolejki
 - » Operacja zwyczajowo zwana dla kolejki: **peek**
 - » **RemoveFirst(L,x)** – usunięcie pierwszego elementu listy (na początku kolejki)
 - » Operacja zwyczajowo zwana dla kolejki : **remove**

Kolejka priorytetowa

(specyficzny rodzaj kolejki)

- Specjalną modyfikacją kolejki jest kolejka priorytetowa
- Każdy element ma przypisany priorytet, który modyfikuje kolejność późniejszego pobierania elementów z kolejki
 - » Przy wstawianiu elementów do kolejki elementu muszą zostać wstawione na miejsce o odpowiednim priorytecie (szybkie pobieranie elementów z kolejki)
 - » **albo**
 - » przy pobieraniu elementów z kolejki wyszukiwany jest element o najwyższym priorytecie (szybkie wstawianie elementów do kolejki)
- Przy pobieraniu elementów na wyjściu kolejki niekoniecznie pojawią się te elementy, które w kolejce oczekują najdłużej, lecz te o największym priorytecie.

Kolejki priorytetowej opartej na idei uporządkowania elementów na liście względem priorytetu

- Jeden ze sposobów abstrakcyjnej realizacji kolejek priorytetowych
- Operacje na kolejce priorytetowej opartej na idei uporządkowania elementów względem priorytetu :
 - » **IsEmpty(L)** – sprawdzenie, czy lista (kolejka) jest pusta
 - » **AddByPriority(L,x)** – wstawienie elementu **x** do listy w odpowiednie miejsce zgodnie z ustalonym priorytetem dla **x**
 - Operacja zwyczajowo zwana dla kolejki: **insertByPriority**
 - » **GetFirst(L)** – pobranie informacji o pierwszym elemencie listy, który jest na początku kolejki
 - Operacja zwyczajowo zwana dla kolejki: **peek**
 - » **RemoveFirst(L,x)** – usunięcie pierwszego elementu listy (na początku kolejki)
 - » Operacja zwyczajowo zwana dla kolejki : **remove**

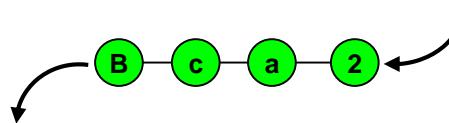
Przykład kolejki priorytetowej opartej na idei uporządkowania elementów względem priorytetu

Różne priorytety:

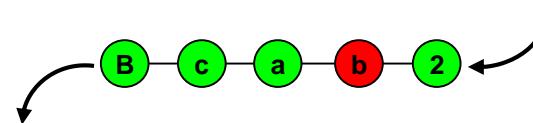
- Litery A, B, C – **priorytet 1** (najwyższy)
- Litery a, b, c – **priorytet 2** (średni)
- Cyfry 1,2,3 – **priorytet 3** (najniższy)

• *Do kolejki wstawiamy: b, 1, A*

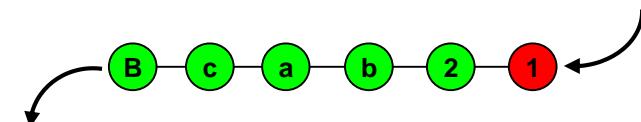
Kolejka priorytetowa na początku



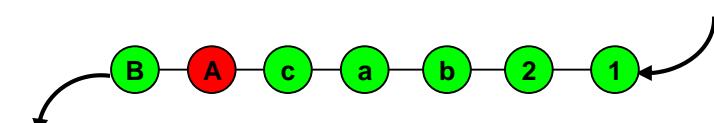
Kolejka priorytetowa po wstawieniu elementu b
(przed 2, bo litera b ma wyższy priorytet niż każda cyfra)



Kolejka priorytetowa po wstawieniu elementu 1
(na samym końcu, bo litery mają najniższy priorytet)



Kolejka priorytetowa po wstawieniu elementu A
(przed c, bo litera A ma wyższy priorytet niż każda mała litera i każda cyfra)



Przeróbka listy na kolejkę priorytetową

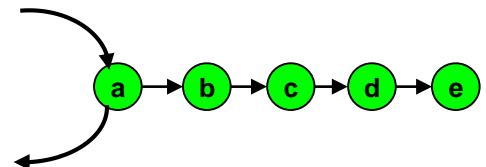
- Zwykła listę (jednokierunkową lub dwukierunkową) można łatwo zamienić na kolejkę priorytetową
 - » *Warunkiem jest, aby można było przeglądać tę listę*
- Zmiana polega na tym, że procedury pobrania informacji i usunięcia elementu pierwszego w kolejce muszą przeglądać całą kolejkę celem znalezienia elementu o najwyższym priorytecie
- Operacje na kolejce:
 - » **IsEmpty(L)** – sprawdzenie, czy lista (kolejka) jest pusta
 - » **AddLast(L,x)** – wstawienie elementu **x** do listy na tył kolejki (na prawy koniec listy)
 - » Operacja zwyczajowo zwana dla kolejki: **insert**
 - » **GetFirstByPriority(L)** – pobranie informacji o elemencie listy, który ma najwyższy priorytet
 - » Operacja zwana dla kolejki priorytetowej: **peekByPriority**
 - » **RemoveFirstByPriority(L,x)** – usunięcie elementu listy , który ma najwyższy priorytet
 - » Operacja zwana dla kolejki priorytetowej: **removeByPriority**

Przykładowe zastosowania kolejek

- Systemy symulacyjne, w których dane kolejki mogą odpowiadać czasom wystąpienia zdarzeń przeznaczonych do chronologicznego przetwarzania
 - » Np. w grach komputerowych zdarzenia są obsługiwane w kolejności pojawiania się na platformie gry lub z uwzględnieniem priorytetów
- Planowanie zadań w systemach komputerowych - dane kolejki mogą oznaczać priorytety wskazujące, którzy użytkownicy mają być obsługiwani w pierwszej kolejności
 - » Np. kolejka wydruków na drukarce sieciowej

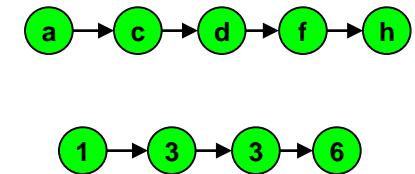
Lista jednokierunkowa, jednostronna

- Lista jednokierunkowa – rodzaj listy z możliwością przeglądania danych od lewego do prawego końca listy; dane dopisywane są na lewym końcu oraz z lewego końca listy są usuwane;
 - » Dzięki możliwości przeglądania danych dostępne jest wyszukiwanie elementów oraz ich usuwanie i wstawianie na całej długości listy.
- Główne operacje listy jednokierunkowej:
 - » **IsEmpty(L)** – sprawdzenie, czy lista jest pusta
 - » **AddFirst(L,x)** - wstawienie elementu x do listy L na początku listy (po lewej stronie)
 - Operacja zwyczajowo zwana: **insert**
 - » **RemoveFirst(L)** - usunięcie elementu na lewym końcu listy
 - Operacja zwyczajowo zwana: **delete**
 - » **GetFirst(L)** - pobranie informacji o pierwszym elemencie listy,
- Wybrane dodatkowe możliwości przy wykorzystaniu przeglądania listy:
 - » **Find(L,x), Remove(L,x), Add(L,x,i), Remove(L,i), Get(L,i)**



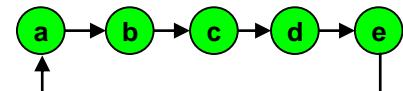
Lista jednokierunkowa, jednostronna, uporządkowana

- Lista jednokierunkowa, jednostronna i uporządkowana to taka lista, że elementy tej listy cały czas są uporządkowane względem jakiejś relacji porządku; w związku z koniecznością wstawiania elementów w ustalonym porządku wstawianie elementów jest liniowe (wymaga przeglądania listy)
- Główne operacje listy jednokierunkowej, jednostronnej i uporządkowanej:
 - » **isEmpty(L)** – sprawdzenie, czy lista jest pusta
 - » **AddOrdered(L,x)** - wstawienie elementu **x** do listy L w odpowiednim miejscu celem zachowania porządku elementów
 - Operacja zwyczajowo zwana: **insert**
 - » **RemoveFirst(L)** - usunięcie elementu na lewym końcu listy
 - Operacja zwyczajowo zwana: **delete**
 - » **GetFirst(L)** - pobranie informacji o pierwszym elemencie listy,
- Wybrane dodatkowe możliwości przy wykorzystaniu przeglądania listy:
 - » **Find(L,x), Remove(L,x), Add(L,x,i), Remove(L,i), Get(L,i)**



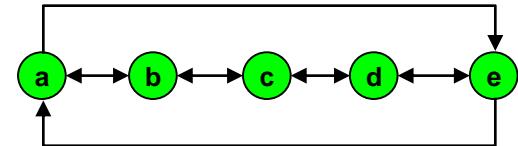
Lista jednokierunkowa, dwustronna

- Lista jednokierunkowa, dwustronna – rodzaj listy z możliwością przeglądania danych od lewego do prawego końca listy (tak jak w liście jednokierunkowej), ale dodatkowy dostęp do prawego elementu listy pozwala na **cykliczne przeglądanie** elementów
- Dane dopisywane mogą być zarówno do lewego jak i prawego końca listy oraz z lewego końca listy są usuwane
 - » Dodatkowo, dzięki możliwości przeglądania danych dostępne jest wyszukiwanie elementów oraz ich usuwanie i wstawianie na całej długości listy.
- Główne operacje listy jednokierunkowej, dwustronnej:
 - » **isEmpty(L)** – sprawdzenie, czy lista jest pusta
 - » **AddFirst(L,x)** - wstawienie elementu x do listy L na początku listy (po lewej stronie)
 - Operacja zwyczajowo zwana: **insertFirst**
 - » **AddLast(L,x)** - wstawienie elementu x do listy L na końcu listy (po prawej stronie)
 - Operacja zwyczajowo zwana: **insertLast**
 - » **RemoveFirst(L)** - usunięcie elementu na lewym końcu listy
 - Operacja zwyczajowo zwana: **delete**
 - » **GetFirst(L)** - pobranie informacji o pierwszym elemencie listy,
 - » **GetLast(L)** - pobranie informacji o ostatnim elemencie listy
- Wybrane dodatkowe możliwości przy wykorzystaniu przeglądania listy:
 - » **Find(L,x), Remove(L,x), Add(L,x,i), Remove(L,i), Get(L,i)**



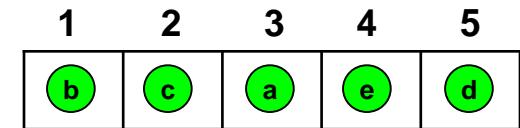
Lista dwukierunkowa

- Lista dwukierunkowa – rodzaj listy z możliwością przeglądania danych zarówno od lewego do prawego końca listy, jak i od prawego do lewego końca listy (swobodny dostęp do pierwszego i ostatniego elementu listy pozwala na cykliczne przeglądanie elementów w obydwie strony)
- Dane dopisywane są na lewym lub prawym końcu oraz z obydwu stron mogą być usuwane
- Dzięki możliwości przeglądania danych dostępne jest wyszukiwanie elementów oraz ich usuwanie i wstawianie na całej długości listy.
- Główne operacje listy dwukierunkowej:
 - » **isEmpty(L)** – sprawdzenie, czy lista jest pusta
 - » **AddFirst(L,x)** - wstawienie elementu x do listy L na początku listy (po lewej stronie)
 - Operacja zwyczajowo zwana: **insertFirst**
 - » **AddLast(L,x)** - wstawienie elementu x do listy L na końcu listy (po prawej stronie)
 - Operacja zwyczajowo zwana: **insertLast**
 - » **RemoveFirst(L)** - usunięcie elementu na lewym końcu listy
 - Operacja zwyczajowo zwana: **deleteFirst**
 - » **RemoveLast(L)** - usunięcie elementu na prawym końcu listy
 - Operacja zwyczajowo zwana: **deleteLast**
 - » **GetFirst(L)** - pobranie informacji o pierwszym elemencie listy
 - » **GetLast(L)** - pobranie informacji o ostatnim elemencie listy
- Wybrane dodatkowe możliwości przy wykorzystaniu przeglądania listy:
 - » **Find(L,x), Remove(L,x), Add(L,x,i), Remove(L,i), Get(L,i)**



Lista z bezpośrednim dostępem do każdego elementu listy poprzez podanie jego numeru

- Lista z bezpośredniem dostępem do elementów – rodzaj listy z możliwością swobodnego dostępu do wszystkich elementów listy za pomocą numeru elementu
- Dane dopisywane są zwykle na prawym końcu listy i tam są też usuwane
- Dzięki możliwości przeglądania danych dostępne jest wyszukiwanie liniowe elementów na całej długości listy.
 - Typowe operacje na liście z bezpośredniem dostępem do elementów na podstawie numeru :
 - AddLast(L,x)**
 - RemoveLast(L)**
 - Get(L,i)**
 - Find(L,x)**



Lista uporządkowana z bezpośrednim dostępem do każdego elementu listy poprzez podanie jego numeru

- Lista uporządkowana z bezpośredniim dostępem do elementów – rodzaj listy z możliwością swobodnego dostępu do wszystkich elementów listy za pomocą numeru elementu; dane dopisywane są w taki sposób, aby elementy były uporządkowane; dane usuwane są na prawym końcu listy; jednak dzięki możliwości do danych w oparciu o numer elementu oraz ich uporządkowaniu dostępne jest wyszukiwanie binarne elementów na całej długości listy.

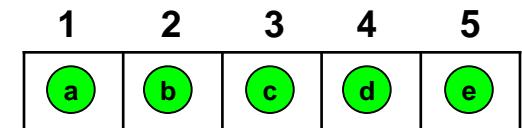
- Możliwości listy uporządkowanej z bezpośredniim dostępem do elementów:

Add(L,x,i)

RemoveLast(L)

Get(L,i)

Find(L,x) (binarnie)



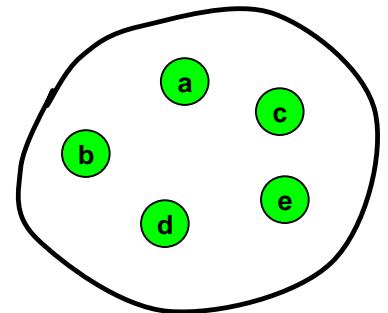
Implementacje listy

- **Tablicowa** – elementy listy są reprezentowane w *tablicy dynamicznej*
- **Dowiązaniowa** – elementy listy oraz struktura powiązań między nimi są przechowywane w pamięci za pomocą zmiennych wskaźnikowych (z użyciem *listy powiązanej*)

Zbiór

- Zbiór w sensie abstrakcyjnym, to kolekcja elementów $S=\{x_1, \dots, x_n\}$, które nie są podane w żadnym ustalonym porządku.
- Liczba $|S|=n$ jest **rozmiarem zbioru S**.
- Szczególnym przypadkiem zbioru jest **zbiór pusty**: $S=\{\}$

Zbiór $S=\{a,b,c,d,e\}$



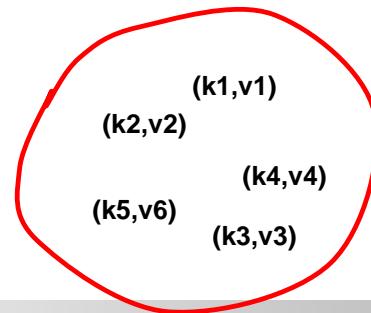
Podstawowe operacje na zbiorach

- **Insert(x,S)** – wstawienie elementu x do zbioru S
- **Delete(x,S)** – usunięcie elementu x ze zbioru S
- **Member(x,S)** – sprawdzenie czy element x należy do zbioru S (wynikiem jest wartość true, gdy x należy do S i false wpp.)
- **Union(A,B)** – obliczenie sumy zbiorów A i S2; zwraca nowy zbiór będący sumą A i B
- **Intersection(A,B)** – obliczenie iloczynu zbiorów A i B; zwraca nowy zbiór będący iloczynem A i B
- **Difference(A,B)** – obliczenie różnicy zbiorów A i B; zwraca nowy zbiór będący różnicą A i B

Implementacje zbioru

- Tablicowa – elementy zbioru są reprezentowane w tablicy dynamicznej
- Dowiązaniowa – elementy listy oraz struktura powiązań między nimi są przechowywane w pamięci za pomocą zmiennych wskaźnikowych (np. z użyciem **listy powiązanej** lub **drzewa binarnego**)
- Za pomocą tablicy haszującej (podejście najefektywniejsze)

Słownik



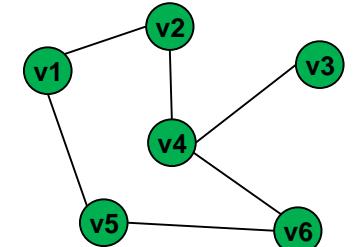
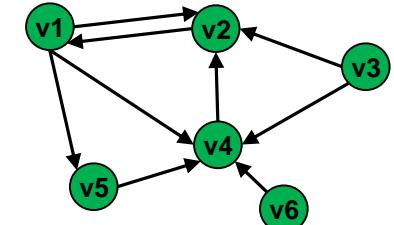
- **Słownik (mapa)** – abstrakcyjny typ danych, który jest zbiorem par: (unikatowy klucz, wartość) i umożliwia dostęp do wartości poprzez podanie klucza.
- Podstawowe operacje na słowniku:
 - » **Put(M,k,v)** – wstawienie do słownika M wartości v z kluczem k. tj. pary (k,v)
 - » **Get(M,k)** – pobranie wartości dla klucza k
 - Zwraca wartość dla klucza k lub wartość NULL, gdy nie ma takiej wartości
 - » **ContainsKey(M,k)** – sprawdzenie czy w słowniku jest wartość dla klucza k
 - Zwraca true, gdy jest taka wartość lub false wpp.
 - » **ContainsValue(M,v)** - sprawdzenie czy w słowniku jest jeden lub więcej kluczy dla wartości v
 - Zwraca true, gdy są takie klucze, wpp. zwraca false
 - » **Remove(M,k)** – usunięcie wszystkich par w których kluczem jest k
- Implementacje słownika: za pomocą tablicy dynamicznej, listy powiązanej, drzewa binarnego lub tablicy haszowanej

Przykład słownika: *Słownik angielsko-polski*

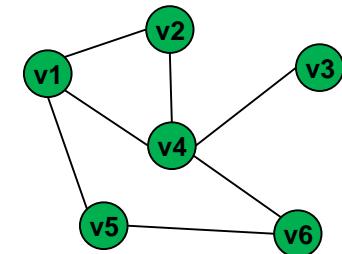
- Do słownika wstawiamy parę zawierającą w sobie klucz będący słowem angielskim i wartość będącą tłumaczeniem tego słowa na język polski
 - » np: **(book, książka)**
- Taki słownik pozwala na wyszukiwanie tłumaczeń słów angielskich na język polski pod warunkiem, że wcześniej były wstawione do słownika
 - » Drugi element pary może być kolekcją słów (wiele tłumaczeń danego słowa)
 - » np: **(book,{książka, bloczek, karnet, rezerwować, rejestrować})**

Graf

- **Grafem G** nazywamy uporządkowaną parę $G=(V,E)$ składającą się z niepustego, skończonego zbioru **wierzchołków (węzłów)** V oraz zbioru **krawędzi** E czyli par wierzchołków ze zbioru V .
- Rozpatrywane są dwa następujące rodzaje grafów:
 - » **graf skierowany (zorientowany, digraf)**, gdy E jest podzbiorem zbioru par uporządkowanych ze zbioru V , tzn. $E \subseteq V \times V$
 - » **graf nieskierowany (niezorientowany)**, gdy E jest podzbiorem zbioru wszystkich dwuelementowych podzbiorów zbioru V



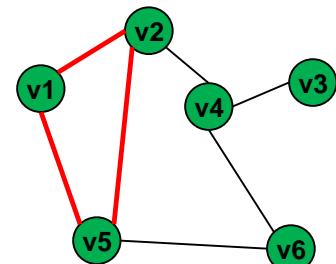
Podstawowe nazewnictwo dotyczące grafów



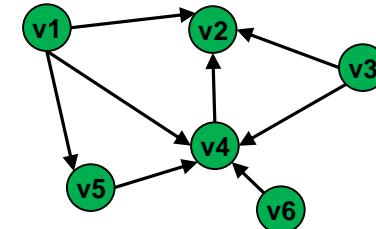
- Niech będzie dany n-wierzchołkowy graf $G=(V,E)$ (skierowany lub nieskierowany).
 - » **Drogą (ścieżką)** w grafie G jest każdy skończony ciąg wierzchołków v_0, v_1, \dots, v_k taki, że dla każdego $i=0, \dots, k-1$ krawędź pomiędzy v_i i v_{i+1} jest krawędzią grafu G .
 - » Liczba k jest **długością ścieżki**
 - » O ścieżce v_0, v_1, \dots, v_k mówimy, że **łączy wierzchołki v_0 i v_k** (lub **prowadzi od v_0 do v_k** , gdy graf jest skierowany)
 - » Wierzchołki v_0 i v_k nazywamy odpowiednio **początkiem i końcem** ścieżki
 - » O krawędzi łączącej dwa kolejne wierzchołki na ścieżce mówimy, że **należy do tej ścieżki**
 - » Długość najkrótszej ścieżki łączącej dwa wierzchołki v i w nazywamy ich **odległością** w grafie

Cykle w grafie i acykliczność

- O ścieżce v_0, v_1, \dots, v_k mówimy że jest **cyklem**, jeżeli pierwszy wierzchołek i ostatni są takie same ($v_0=v_k$).
- Jeżeli wszystkie wierzchołki w cyklu z wyjątkiem v_0 i v_k są parami różne i $k>2$ ($k>1$ dla grafów skierowanych) to cykl nazywamy **prostym**.
- **Acykliczność grafu** – oznacza brak cykli prostych w grafie



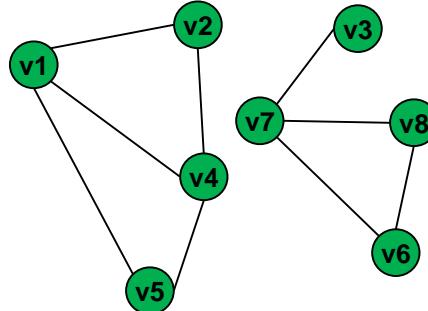
Graf acykliczny



Spójność grafu

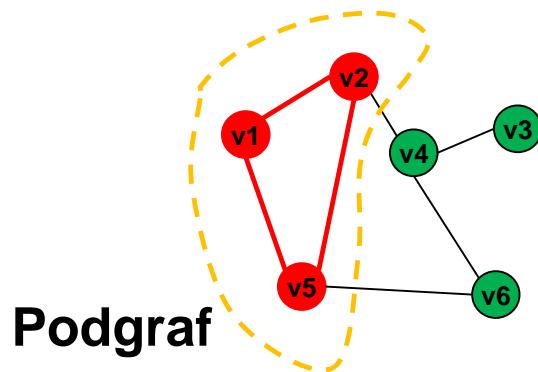
- Spójność – oznacza, że każde dwa wierzchołki grafu są połączone ścieżką utworzoną z krawędzi grafu (nie ma rozłącznych fragmentów)

**Graf
niespójny**



Podgraf

- Graf $G_1=(V_1, E_1)$, w którym V_1 jest podzbiorem zbioru wierzchołków V i E_1 jest podzbiorem zbioru krawędzi E , łączących tylko wierzchołki z V_1 nazywamy **podgrafem** G



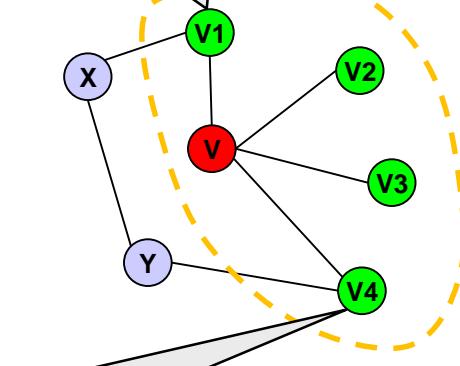
Operacje na grafie

- **AddVertex (G, v)** – dodanie wierzchołka v do grafu G (na razie bez krawędzi)
- **GetVertex(G, i)** – zwrócenie wierzchołka o numerze i w grafie G ; jeśli nie ma takiego wierzchołka, to zwracana jest wartość **null**
- **GetVertexIndex(G, v)** – zwrócenie numeru wierzchołka v ; jeśli nie ma takiego wierzchołka, to zwracana jest wartość **-1**
- **AddEdge(G, i_1, i_2)** – dodanie krawędzi łączącej wierzchołki o numerze i_1 i i_2 w grafie G
- **FirstVertexIndex(G, v)** – zwrócenie numeru (indeksu) pierwszego wierzchołka sąsiedniego dla wierzchołka v w grafie G ; jeśli wierzchołek v nie ma wierzchołków sąsiednich, to zwracana jest wartość **-1** (indeks wierzchołka pustego)
- **NextVertexIndex(G, v, i)** - zwrócenie numeru (indeksu) wierzchołka następnego w stosunku do wierzchołka o indeksie i wśród wierzchołków sąsiednich dla wierzchołka v w grafie G ; jeśli i jest indeksem ostatniego wierzchołka sąsiedniego dla v , to operacja zwraca wartość **-1** (indeks wierzchołka pustego)
- **RemoveVertex(G, i)** – usunięcie wierzchołka o numerze i z grafu G wraz z krawędziami łączącymi ten wierzchołek z wierzchołkami sąsiednimi
- **RemoveEdge(G, i_1, i_2)** – usunięcie krawędzi łączącej wierzchołki o numerach i_1 i i_2 (jeśli taka krawędź istnieje w grafie G)

FirstVertex(G, v) = 1

NextVertex(G, V_3) = 4

NextVertex(G, V_4) = -1



Metody reprezentacji grafów

- Nie będzie dany graf $G=(V,E)$ taki, że $n=|V|$ i $m=|E|$
- **Lista sąsiedztwa**
 - » Dla każdego wierzchołka $x \in V$ budujemy listę (oznaczaną przez $L[x]$) wierzchołków y będących sąsiadami X , tzn. istnieje para uporządkowana $(x,y) \in E$ (dla grafów skierowanych) lub istnieje zbiór $\{x,y\} \in E$ (dla grafów nieskierowanych)
- **Macierz sąsiedztwa**
 - » Budujemy macierz sąsiedztwa A o rozmiarze $n \times n$:

$$A[x, y] = \begin{cases} 1, & \text{dla } (x, y) \in E \\ 0, & \text{dla } (x, y) \notin E \end{cases}$$

Przykład: Lista i macierz sąsiedztwa dla grafu nieskierowanego

- Lista sąsiedztwa:

1: 2, 3

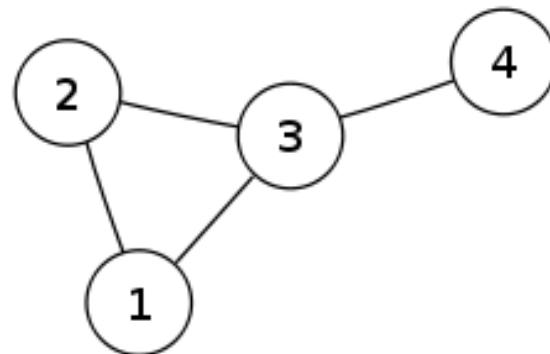
2: 1, 3

3: 1, 2, 4

4: 3

- Macierz sąsiedztwa:

	1	2	3	4
1	0	1	1	0
2	1	0	1	0
3	1	1	0	1
4	0	0	1	0



Przykład: Lista i macierz sąsiedztwa dla grafu skierowanego

- Lista sąsiedztwa:

1: 2, 3

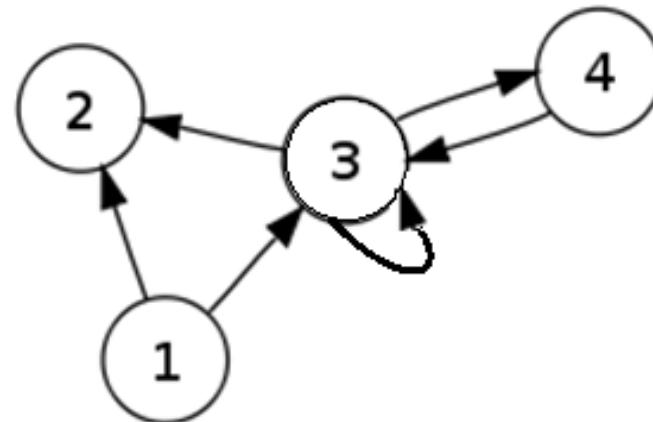
2:

3: 3, 2, 4

4: 3

- Macierz sąsiedztwa:

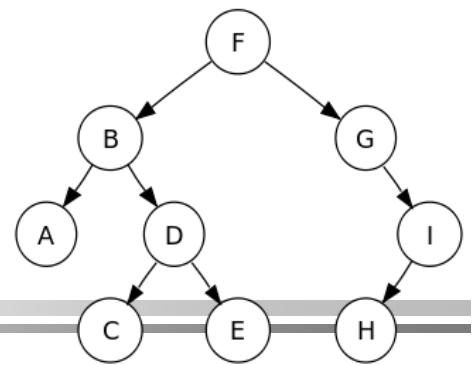
	1	2	3	4
1	0	1	1	0
2	0	0	0	0
3	0	1	1	1
4	0	0	1	0



Implementacje grafu

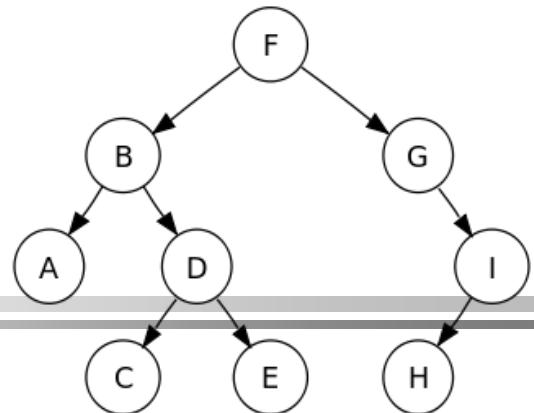
- Podstawową implementacją grafu jest implementacja tablicowa
 - » Tablica służy do implementacji listy lub macierzy sąsiedztwa
- Możliwa jest także implementacja dowiązaniowa
 - » Każdy wierzchołek „pamięta” wskaźniki do wierzchołków będących jego sąsiadami (dość niewygodne)

Drzewo



- Drzewo to dowolny, **nieskierowany graf spójny i acykliczny** składający się z **wierzchołków (węzłów)** oraz łączących je **krawędzi**.
 - » Jeśli drzewo nie jest puste, tzn. liczba wierzchołków jest większa od zera, jeden z nich jest wyróżniony i nazywany **korzeniem drzewa** (na rysunku jest to F)
- Ciąg krawędzi łączących węzły nazywa się **ścieżką**.
 - » Istnieje dokładnie jedna ścieżka łącząca korzeń z wszystkimi pozostałymi wierzchołkami.
- Liczba krawędzi w ścieżce od korzenia do węzła jest nazywana **długością ścieżki** i liczba ta określa poziom węzła.
 - » Np. korzeń znajduje się na 0 poziomie, węzły A, D i I na poziomie 2
- Wysokością drzewa jest liczba poziomów istniejących w drzewie
 - Np. drzewo z rysunku ma wysokość 3

Podstawowe pojęcia dotyczące drzew



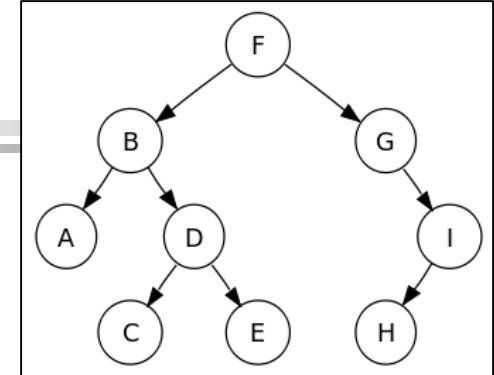
- Wierzchołek jest **rodzicem** dla każdego swojego dziecka.
 - » Każdy węzeł ma dokładnie jednego rodzica, wyjątkiem jest korzeń drzewa, który nie ma rodzica.
- Wszystkie wierzchołki połączone z danym wierzchołkiem, a leżące na następnym poziomie są nazywane **dziećmi** tego węzła
 - » Np. dziećmi wierzchołka F są B i G, natomiast wierzchołka B: A i D.
- Wierzchołek może mieć dowolną liczbę dzieci, jeśli nie ma ich wcale nazywany jest **liściem**
 - » Liśćmi w przykładowym drzewie są A, C, E, H.

Drzewa binarne i wyższych rzędów

- Specjalne znaczenie w informatyce mają **drzewa binarne**, w których liczba dzieci ograniczona jest do dwóch.
 - » Szczególnie popularne są BST i ich różne odmiany, np. drzewa AVL, drzewa czerwono-czarne.
- Drzewa które posiadają więcej niż dwoje dzieci są nazywane **drzewami wyższych rzędów**, np. drzewo trie, drzewo trójkowe, B-drzewo.

Podstawowe operacje na drzewach

- **GetRoot(T)** – zwraca korzeń drzewa T; jeśli drzewo jest puste to zwracany jest NULL
- **InsertNode (T,u,v)** – dodanie wierzchołka v do drzewa T jako dziecko wierzchołka u
- **DeleteNode (T,v)** – usunięcie wierzchołka v z drzewa T z odpowiednim połączeniem krawędzi prowadzących do dzieci wierzchołka v
- **ParentNode (T,v)** – wyszukanie w drzewie T wierzchołka będącego rodzicem wierzchołka v; jeśli v jest korzeniem zwracana jest wartość NULL
- **LeftChildNode(T,v)**, – zwrócenie pierwszego od lewej wierzchołka będącego dzieckiem wierzchołka v w drzewie T; jeśli wierzchołek v nie ma dzieci (jest liściem), to zwracana jest wartość NULL
- **NextChildNode(T,v,u)** – zwrócenie wierzchołka następnego w stosunku do wierzchołka u wśród wierzchołków-dzieci dla wierzchołka v w drzewie T; jeśli u jest najbardziej prawym węzłem wierzchołka-dziecka dla v, to operacja zwraca wartość NULL



Implementacja drzewa

- Podstawową implementacją drzewa jest implementacja dowiązaniowa
 - Każdy wierzchołek „pamięta” wskaźniki do wierzchołków będących jego dziećmi
- Możliwa jest także implementacja tablicowa tak jak w przypadku typowego grafu

Konkretne struktury danych

Konkretne struktury danych

- Konkretne struktury danych zapewniają ułożenie danych w pamięci komputera z użyciem zmiennych konkretnych typów danych oraz rozmaitych technik programowania
 - Np. reprezentowanie danych przy wykorzystaniu złożonych typów danych jak tablice, rekordy lub obiekty (egzemplarze klas), realizacja powiązań pomiędzy danymi przy zastosowaniu wskaźników lub referencji do poszczególnych porcji danych
- Najczęściej wyróżnia się następujące konkretne struktury danych:
 - » Nieuporządkowana i uporządkowana tablica dynamiczna
 - » Nieuporządkowana i uporządkowana lista powiązana
 - » Drzewo binarne (szczególnie drzewo poszukiwań binarnych)
 - » Tablica mieszająca

Na początek: Iterator

- Specjalny obiekt (egzemplarz klasy) do przeglądania konkretnej struktury danych
- Zwykle iteratory są konstruowane według pewnego standardu:
 - » Obiekt iteratora danej struktury danych jest tworzony (dostarczony) przez metodę tej struktury o nazwie `iterator()`
 - » Iterator posiada w swoim wnętrzu rodzaj wskaźnika na element struktury danych, dzięki któremu w danym momencie wskazuje na jakiś element struktury
 - Po utworzeniu, iterator wskazuje na pierwszy element struktury zgodnie z pewnym ustalonym porządkiem (zależnym od struktury danych)
 - » Mechanizmem interatora pobierającym element struktury danych do przeglądnienia na który aktualnie wskazuje iterator jest metoda `next()`
 - Po jej wykonaniu wskaźnik iteratora przesuwa się na następny element struktury
 - » Mechanizmem pozwalającym na stwierdzenie, czy jest jeszcze jakiś element do przeglądnienia jest metoda `hasNext()`
 - » W dowolnym momencie można zresetować iterator za pomocą metody `reset()`, dzięki czemu będzie on wskazywał na pierwszy element przeglądanej struktury danych

Schemat typowego zastosowania iteratora do przeglądania struktury danych

- Dane są:
 - » obiekt struk pewnej struktury danych zdefiniowanej przez klasę Struk, przy czym w strukturze tej przechowywane są obiekty klasy Elem.
 - » Do klasy Struk zdefiniowano klasę iteratora StrukIterator
 - » Przeglądanie struktury struk za pomocą iteratora odbywa się zwykle tak jak poniżej:

```
StrukIterator iterator = struk.iterator();
while (iterator.hasNext())
{
    Elem elem = iterator.next();
    elem.print(); //Jakaś akcja dla elem (np. wypisanie na ekran)
}
```

Czy zawsze do przeglądania struktur danych używamy interatorów?

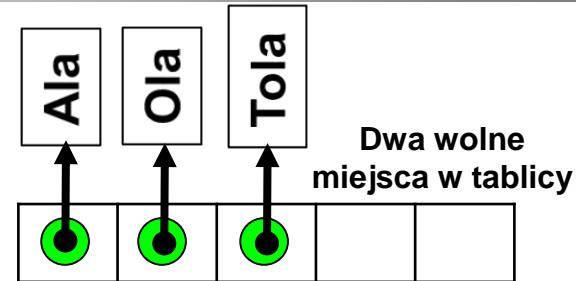
- Jeśli w strukturze danych jest łatwy dostęp do elementów (np. przez podanie indeksów – tablica dynamiczna) **to nie ma potrzeby używania iteratora.**

Tablice dynamiczne

Tablica dynamiczna (zwana np. DynArray)



nElems 3
Zmienna przechowująca aktualną liczbę elementów w tablicy



- Konkretna struktura danych implementowana jako klasa oparta na tablicy, w której wprost przechowywane są elementy lub referencje (adresy) przechowywanych w strukturze elementów
 - » Pierwszy przypadek dotyczy sytuacji gdy elementy są wartościami typów konkretnych, tj. int, double, float, byte, char itd
 - » Drugi przypadek dotyczy sytuacji, gdy przechowywane elementy są wartościami typów obiektowych, czyli są egzemplarzami klas, np. String
- W klasie jest prywatna zmienna tablicowa np. **table**, prywatna zmienna o nazwie np. **nElems** przechowująca aktualną liczbę elementów wstawionych do tablicy oraz publiczna metoda (np. **size()**), która zwraca wartość tej zmiennej
- Konstruktor klasy zwykle określa maksymalną liczbę elementów które mogą zmieścić się w tablicy
- W klasie jest metoda np. **add**, która wstawia element będący jej argumentem do tablicy (jednocześnie metoda ta zwiększa wartość licznika elementów)
- W klasie jest metoda np. **get**, która zwraca element tablicy o podanym numerze (indeksie), przy czym numery elementów są od 0 do nElems-1
- W klasie jest metoda np. **remove**, która usuwa element o podanym numerze; jednocześnie metoda ta zmniejsza wartość licznika elementów

Przykład tablicy dynamicznej przechowującej liczby całkowite

```
public class IntDynArray
{
    private int [] table; //Referencja do tablicy
    private int nElems; //Aktualna liczba elementow w tablicy

    public IntDynArray(int maxSize) // Konstruktor
    {
        table = new int[maxSize]; nElems = 0;
    }
    public void add(int value) // Wstawienie elementu do tablicy
    {
        if (nElems >= table.length) throw new IndexOutOfBoundsException("Za duzo elementow!");
        table[nElems] = value; nElems++; // Wstawiamy element
    }
    public int get(int index) {return table[index]; } // Pozyskanie elementu o danym indeksie

    public int size() {return nElems; } // Zwraca liczbe elementow w tablicy

    public boolean remove(int index) // Usuniecie elementu o danym indeksie
    {
        if (nElems==0) return false; //Jesli nie ma elementow nic nie usuwamy
        for (int j = index; j < nElems-1; j++) // Przesuwamy pozostale elementy w lewo
        {
            table[j] = table[j + 1];
        }
        nElems--; // Zmniejszamy licznik elementow
        return true;
    }

    public void print() //Wypisanie elementow tablicy dynamicznej
    {
        for (int i = 0; i < nElems; i++) System.out.print(get(i)+" "); System.out.println();
    }
}
```

Test tablicy dynamicznej przechowujacej liczby całkowite

```
public static void main(String[] args)
{
    int maxSize = 100;
    IntDynArray dynArray = new IntDynArray(maxSize);
    dynArray.add(11);    // Wstawiamy 4 elementy
    dynArray.array.add(33);
    dynArray.add(22);
    dynArray.add(44);
    dynArray.print();   // Wypisujemy zawartosc tablicy
    dynArray.remove(1); // Usuwamy 2 elementy
    dynArray.remove(2);
    dynArray.print();   // Wypisujemy zawartosc tablicy
    dynArray.add(55);   // Wstawiamy 2 elementy
    dynArray.add(66);
    dynArray.print();   // Wypisujemy zawartosc tablicy
}
```

```
11 33 22 44
11 22
11 22 55 66
```

Relokacja tablicy dynamicznej



- Jeśli do tablicy dynamicznej wstawione jest tyle elementów, że zapełniona jest cała tablica wewnętrzna, to wstawienie dalszych elementów nie jest możliwe
 - » Potrzeba przewidywania maksymalnej liczby elementów – niewygodne
- Rozwiązaniem problemu jest relokacja tablicy dynamicznej
 - » Utworzenie wewnętrz tablicy dynamicznej nowej tablicy wewnętrznej o znaczaco większym rozmiarze, przepisanie elementów z dotychczasowej tablicy wewnętrznej do nowej (większej) tablicy wewnętrznej, podmienienie starej tablicy wewnętrznej na nową
- Typowym sposobem na zwiększenie tablicy wewnętrznej jest podwojenie jej wielkości

Relokacja tablicy IntDynArray

- Relokacja ma miejsce przy wstawianiu elementów, gdy zabraknie miejsca na elementy

```
public void add(int value)           // Wstawienie elementu do tablicy
{
    if (nElems >= table.length) //Sprawdzenie czy brak miejsca
    {
        int [] locTable = new int[table.length*2]; //Utworzenie nowej tablicy
                                                       //wewnętrznej
        for (int i=0; i<table.length; i++)
            locTable[i]=table[i]; //Przepisanie starej tablicy do nowej
        table = locTable; //Zamiana starej tablicy na nowa
    }
    table[nElems] = value;          // Wstawienie elementu
    nElems++;                      // Zwiększenie licznika elementow
}
```

Wyszukiwanie elementu w tablicy dynamicznej (nieuporządkowanej) elementów typu prostego (np. int)

```
public int find(int searchElem) // Szukanie określonego elementu
{
    for (int j = 0; j < nElems; j++)
    {
        if (table[j] == searchElem) return j; //Element znaleziono
    }
    return -1; // Elementu nie znaleziono
}
```

- Typowe wyszukiwanie liniowe.

Złożoność obliczeniowa operacji na tablicy dynamicznej

- Niech n będzie liczbą elementów w tablicy dynamicznej

Wstawienie elementu	Wyszukanie elementu	Usunięcie elementu
$O(1)$	$O(n)$	$O(n)$

- Wstawienie elementu odbywa się na końcu tablicy
- Wyszukiwanie jest liniowe
- Usunięcie wymaga wyszukania elementu i przesunięcia elementów

Przykład tablicy dynamicznej przechowującej obiekty dowolnej klasy

```
public class DynArray
{
    private Object [] table; //Referencja do tablicy
    private int nElems; //Aktualna liczba elementow w tablicy

    public DynArray(int maxSize) // Konstruktor
    {
        table = new Object[maxSize]; nElems = 0;
    }

    public void add(Object value)
    {
        if (nElems >= table.length)
        {
            Object [] locTable = new Object[table.length*2];
            for (int i=0; i<table.length; i++)
                locTable[i]=table[i];
            table = locTable;
        }
        table[nElems] = value; nElems++;
    }

    public Object get(int index) {return table[index];}

    public int size() { return nElems; }

    public boolean remove(int index)
    {
        if (nElems==0) return false;
        for (int j = index; j < nElems-1; j++)
        {
            table[j] = table[j + 1];
        }
        nElems--; return true;
    }
}
```

Ala Ola Tola
Ala Tola
Ala Tola Lolek

```
public void print()
{
    for (int i = 0; i < nElems; i++)
        System.out.print(get(i).toString()+" ");
    System.out.println();
}

public static void main(String[] args)
{
    int maxSize = 5;
    DynArray array = new DynArray(maxSize);
    array.add("Ala");
    array.add("Ola");
    array.add("Tola");
    array.print();
    array.remove(1);
    array.print();
    array.add("Lolek");
    array.print();
}
```

Wyszukiwanie elementu w tablicy dynamicznej (nieuporządkowanej) zawierającej obiekty (egzemplarze klas)

```
public int find(Object searchElem) // Szukanie określonego elementu
{
    for (int j = 0; j < nElems; j++)
    {
        if (table[j].equals(searchElem)) return j; //Element znaleziono
    }
    return -1; // Elementu nie znaleziono
}
```

- Typowe wyszukiwanie liniowe
- Wymaga przełonięcia w klasie przechowywanego w tablicy elementu metody equals

Klasa `ArrayList` jako standardowa kolekcja języka Java implementująca tablicę dynamiczną elementów parametryzowanego typu oznaczanego przez `<typ>` (wybrane możliwości)

- Konstruktory:
 - » `ArrayList<typ>()` – bezparametrowy konstruktor,
 - » `ArrayList<typ>(int initialCapacity)` – konstruktor z początkowym rozmiarem tablicy,
- `void add(<typ> elem)` – wstawienie elementu do struktury
- `int size()` – pobranie informacji o aktualnej liczbie wstawionych elementów
- `<typ> get(int index)` – pobranie elementu o podanym numerze
- `void remove(int index)` – usunięcie elementu o podanym numerze,
- `void clear()` – usunięcie wszystkich elementów kolekcji
- `<typ> set(int index, <typ> elem)` – zastąpienie elementu będącego na miejscu `index` elementem `elem` (zwraca poprzedni element)

Przykład zastosowania klasy ArrayList

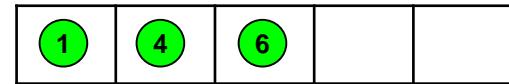
```
import java.util.ArrayList;
public class TestArrayList
{
    public static void print(ArrayList<String> arrayList)
    {
        for (int i=0; i<arrayList.size(); i++)
            System.out.print(arrayList.get(i)+" ");
        System.out.println();
    }

    public static void main(String[] args)
    {
        ArrayList<String> arrayList = new ArrayList<String>();
        arrayList.add("Ala");
        arrayList.add("Ola");
        arrayList.add("Lolek");
        print(arrayList);
        arrayList.remove(1);
        print(arrayList);
        arrayList.add("Bolek");
        arrayList.set(1,"Karol");
        print(arrayList);
    }
}
```

Ala Ola Lolek
Ala Lolek
Ala Karol Bolek

Uporządkowana tablica dynamiczna (zwana np. OrdDynArray)

- Elementy uporządkowanej tablicy dynamicznej zawsze są ułożone względem jakiegoś porządku
- W związku z tym metoda **add** musi wstawiać elementy w odpowiednim miejscu tablicy co zajmuje czas liniowy względem liczby elementów w tablicy
- Dzięki uporządkowaniu elementów metoda **find** może wykorzystać wyszukiwanie binarne



Wstawianie elementu do uporządkowanej tablicy dynamicznej liczb całkowitych

```
public void add(int value)           // Wstawienie elementu do tablicy
{
    if (nElems >= table.length) //Potrzeba relokacji
    {
        int [] locTable = new int[table.length*2];
        for (int i=0; i<table.length; i++) locTable[i]=table[i];
        table = locTable;
    }
    int j;
    for(j=0; j<nElems; j++) // Znajdujemy miejsce dla elementu
        if(table[j] > value) break;

    for(int k=nElems; k>j; k--) // Przesuwamy większe elementy
        table[k] = table[k-1];

    table[j] = value;           // Wstawiamy element
    nElems++;                  // Zwiększamy licznik elementow
}
```

- Wyszukanie miejsca i przesunięcie elementów jest liniowe

Wyszukanie elementu w uporządkowanej tablicy dynamicznej liczb całkowitych

```
public int find(int searchElem) // Szukanie określonego elementu
{
    int left = 0;           // ograniczenie lewe
    int right = nElems - 1; // ograniczenie prawe
    int currIndex; // aktualnie sprawdzany indeks

    while (true)
    {
        currIndex = (left + right) / 2;
        if (table[currIndex] == searchElem) return currIndex; //Element znaleziony
        else
            if (left > right) return -1; //Brak elementu
            else
                {
                    if (table[currIndex]<searchElem) left = currIndex + 1; //Jest w górnej połowie
                    else right = currIndex - 1; // Jest w dolnej połowie tablicy
                }
    }
}
```

- Wyszukiwanie binarne

Złożoność obliczeniowa operacji na uporządkowanej tablicy dynamicznej

- Niech n będzie liczbą elementów w tablicy dynamicznej

Wstawienie elementu	Wyszukanie elementu	Usunięcie elementu
$O(n)$	$O(\log n)$	$O(n)$

- Wstawienie elementu odbywa się za pomocą wyszukania liniowego i wymaga przekładania elementów
- Wyszukiwanie jest binarne
- Usunięcie wymaga wyszukania elementu i przesunięcia elementów (przesunięcie powoduje, że pesymistyczna złożoność jest liniowa)

Wyszukiwanie elementu w uporządkowanej tablicy dynamicznej zawierającej obiekty typu String

```
public int find(String searchElem) // Szukanie określonego elementu
{
    int left = 0;                  // ograniczenie lewe
    int right = nElems - 1; // ograniczenie prawe
    int currIndex;   // aktualnie sprawdzany indeks

    while (true)
    {
        currIndex = (left + right) / 2;
        if (table[currIndex].equals(searchElem))
        {
            return currIndex; //Element znaleziony
        }
        else if (left > right) return -1; //Brak elementu
        else
        {
            if (table[currIndex].compareTo(searchElem) < 0) // table[currIndex]<searchElem
            {
                left = currIndex + 1; // Jest w górnej połowie tablicy
            }
            else
            {
                right = currIndex - 1; // Jest w dolnej połowie tablicy
            }
        }
    }
}
```

- Wyszukiwanie binarne
- Wymaga zdefiniowania w klasie metody `compareTo`
- Implementacja w klasie `OrdStringDynArray`

Podsumowanie zalet i wady tablicy dynamicznej

- **Zalety:**
 - » Łatwe do zaprogramowania
 - » Dostęp swobodny (w czasie stałym) do elementów
 - » Możliwość przeglądania uporządkowanych danych, ale tylko dla tablicy uporządkowanej
 - » Dosyć szybkie wyszukiwanie danych dla tablic uporządkowanych
- **Wady:**
 - » Utrudnione usuwanie elementów
 - » Potrzeba przewidywania wielkości tablicy lub zgoda na czasochłonną relokację
 - » Nadmiar użytej pamięci (na nieużywany jeszcze fragment wewnętrznej tablicy)
 - » Utrudnione wstawianie elementów dla tablicy uporządkowanej
 - » Wolne wyszukiwanie danych dla tablic nieuporządkowanych

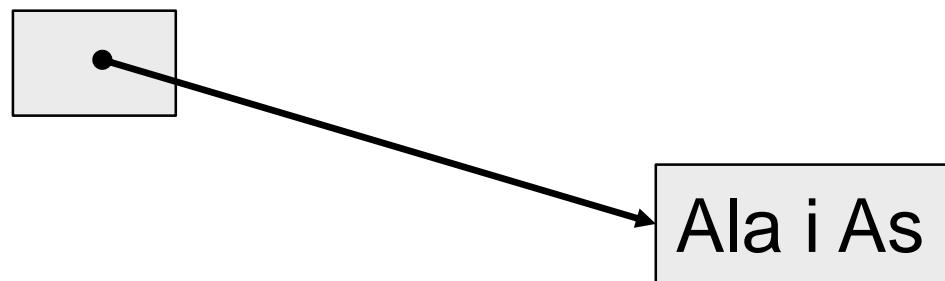
Listy powiązane

Jak w języku Java organizowany jest dostęp do obiektów?

- Obiekty są tworzone jako egzemplarze klas
- Referencje (adresy) do tak utworzonych obiektów są umieszczane w zmiennych typu tej klasy
- Np:

```
String tekst = new String("Ala i As");
```

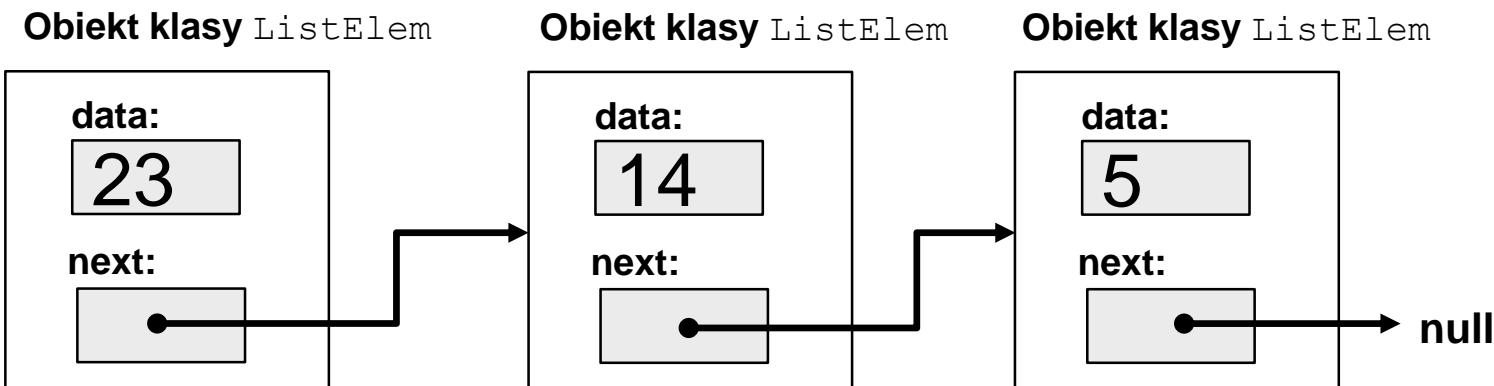
Zmienna: tekst



Połączenia pomiędzy obiektami jako sposób na reprezentację list

- Każda klasa może mieć dodatkowe pole typu tej właśnie klasy
- W obiektach klasy takie pole może przechowywać referencję do innego obiektu tej samej klasy
- Dzięki takim połączeniom można tworzyć w pamięci komputera listy elementów będących obiektami tej klasy

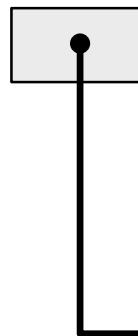
```
public class ListElem  
{  
    public int data;  
    public ListElem next;  
}
```



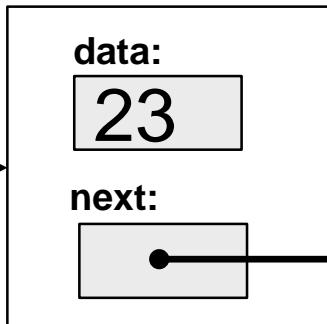
Lista powiązana jednostronna (LinkedList)

- Dla zapewnienia dostępu do tego typu list, w programie musi być jeszcze jedna zmienna typu tej samej klasy, która przechowuje referencję do pierwszego elementu listy (zmienna **first**)
 - » Z tej racji, że ostatni element listy nie wskazuje już na żaden obiekt, wprowadzono specjalną wartość na którą zmienna **next** w tym obiekcie powinna wskazywać (wartość **null**)
- Tego typu listy nazywamy **listami powiązanymi**

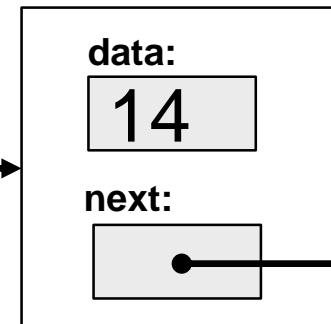
first:



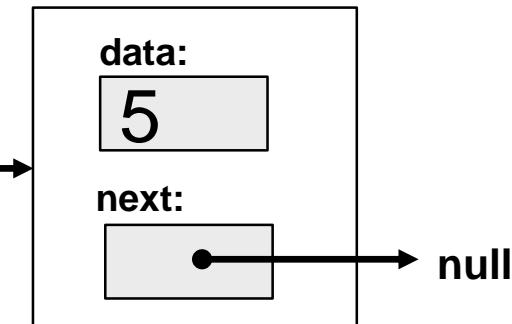
Obiekt klasy ListElem



Obiekt klasy ListElem



Obiekt klasy ListElem



Przykład klasy elementu listy powiązanej

```
public class ListElem
{
    public int iData; // Jakas dana w elemencie listy (liczba calkowita)
    public ListElem next; // Referencja do nastepnego elementu listy

    public ListElem(int iData) // konstruktor
    {
        this.iData = iData; // Inicjalizacja danych
        next = null; // Pole next ma na poczatku wartosc null
    }

    public void printElem() // Wypisuje element listy
    {
        System.out.print(iData);
    }
}
```

```

public class LinkedList
{
    private ListElem first; // Referencja do pierwszego elem.

    public LinkedList() { first = null; } // Konstruktor
    public boolean isEmpty() { return (first == null); }

    // Wstawienie elementu na początek listy
    public void insertFirst(int value)
    {
        ListElem newElem = new ListElem(value);
        newElem.next = first;
        first = newElem;
    }

    // Wyszukiwanie elementu
    public ListElem find(int elem)
    {
        if (isEmpty()) return null;
        ListElem current = first;
        while (current.iData != elem)
        {
            if (current.next == null) return null;
            else current = current.next;
        }
        return current;
    }

    // Usunięcie pierwszego elementu listy
    public ListElem deleteFirst()
    {
        if (isEmpty()) return null;
        ListElem temp = first;
        first = first.next;
        return temp;
    }
}

```

Przykład listy powiązanej przechowującej liczby całkowite

```

    //Usuwanie wskazanego elementu
    public ListElem delete(int elem)
    {
        if (isEmpty()) return null;
        ListElem current = first;
        ListElem previous = null;
        while (current.iData != elem)
        {
            if (current.next == null) return null;
            else
            {
                previous = current;
                current = current.next;
            }
        }
        if (previous == null)
        { first = first.next; }
        else
        {
            previous.next = current.next;
        }
        return current;
    }
}

```

Wyszukiwanie elementu na liście powiązanej

```
public ListElem find(int elem)      // Wyszukiwanie elementu
{
    if (isEmpty()) return null;

    ListElem current = first; // Rozpoczynamy od pierwszego elementu
    while (current.iData != elem)
    {
        if (current.next == null) return null;
        else
            current = current.next;
    }
    return current;
}
```

Test listy powiązanej przechowującej liczby całkowite

```
Lista (początek-->koniec): 88 66 44 22
Znaleziono element 44
Usunięto element o kluczu 66
Lista (początek-->koniec): 88 44 22
```

```
public static void main(String[] args)
{
    LinkedList theList = new LinkedList(); //Tworzymy listę
    theList.insertFirst(22); //Wstawiamy 4 elementy
    theList.insertFirst(44);
    theList.insertFirst(66);
    theList.insertFirst(88);
    theList.printList(); //Wypisujemy zawartość listy
    int liczba = 44;
    ListElem fElem = theList.find(liczba); // Szukamy elementu: 44
    if (fElem != null) System.out.println("Znaleziono " + fElem.iData);
    else System.out.println("Nie znaleziono "+liczba);
    ListElem dElem = theList.delete(66); // Usuwamy element
    if (dElem != null) System.out.println("Usunięto " + dElem.iData);
    else System.out.println("Nie można usunąć elementu");
    theList.printList(); // Wypisujemy listę
}
```

Złożoność obliczeniowa operacji na liście powiązanej

- Niech n będzie liczbą elementów na liście powiązanej

Wstawienie elementu	Wyszukanie elementu	Usunięcie elementu
$O(1)$	$O(n)$	$O(n)$

- Wstawienie elementu odbywa się na początku listy
- Wyszukiwanie jest liniowe
- Usunięcie wymaga wyszukania elementu

Uporządkowana lista powiązana jednostronna (SortedLinkedList)

- Elementy uporządkowanej listy powiązanej zawsze są ułożone względem jakiegoś porządku
- W związku z tym metoda **insert** musi wstawiać elementy w odpowiednim miejscu listy co zajmuje czas liniowy względem liczby elementów na liście

Wstawianie elementu do uporządkowanej listy powiązanej liczb całkowitych

```
public void insert(int value) // Wstawianie z zachowaniem porządku
{
    ListElem newElem = new ListElem(value);
    ListElem previous = null;
    ListElem current = first;      //Rozpoczynamy od początku listy

    // Dopóki nie koniec listy i elementy sa mniejsze niz value
    while (current != null && newElem.iData > current.iData)
    {
        previous = current;
        current = current.next; // Przechodzimy do następnego elementu
    }

    if (previous == null) first = newElem; // Znalazl na początku listy
    else previous.next = newElem; //... nie na początku...
    newElem.next = current;
}
```

- Wyszukanie miejsca jest liniowe

Złożoność obliczeniowa operacji na uporządkowanej liście powiązanej

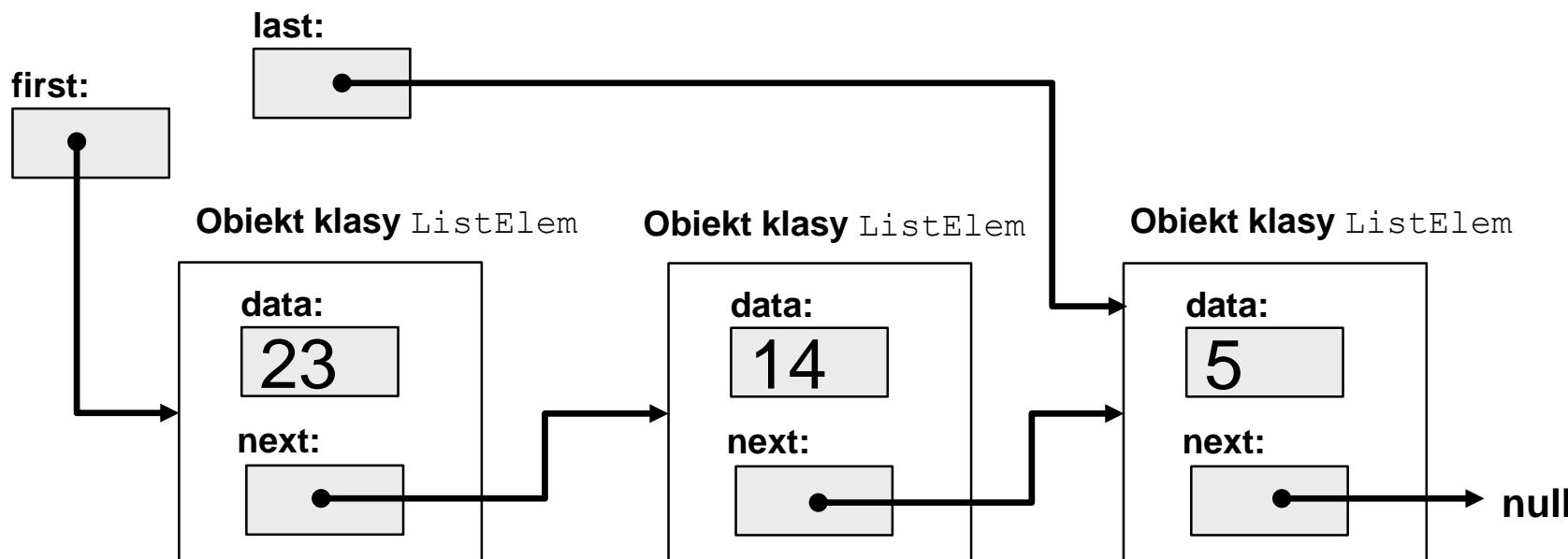
- Niech n będzie liczbą elementów na liście powiązanej

Wstawienie elementu	Wyszukanie elementu	Usunięcie elementu
$O(n)$	$O(n)$	$O(n)$

- Wstawienie elementu wymaga wyszukania miejsca na ten element
- Wyszukiwanie jest liniowe
- Usunięcie wymaga wyszukania elementu

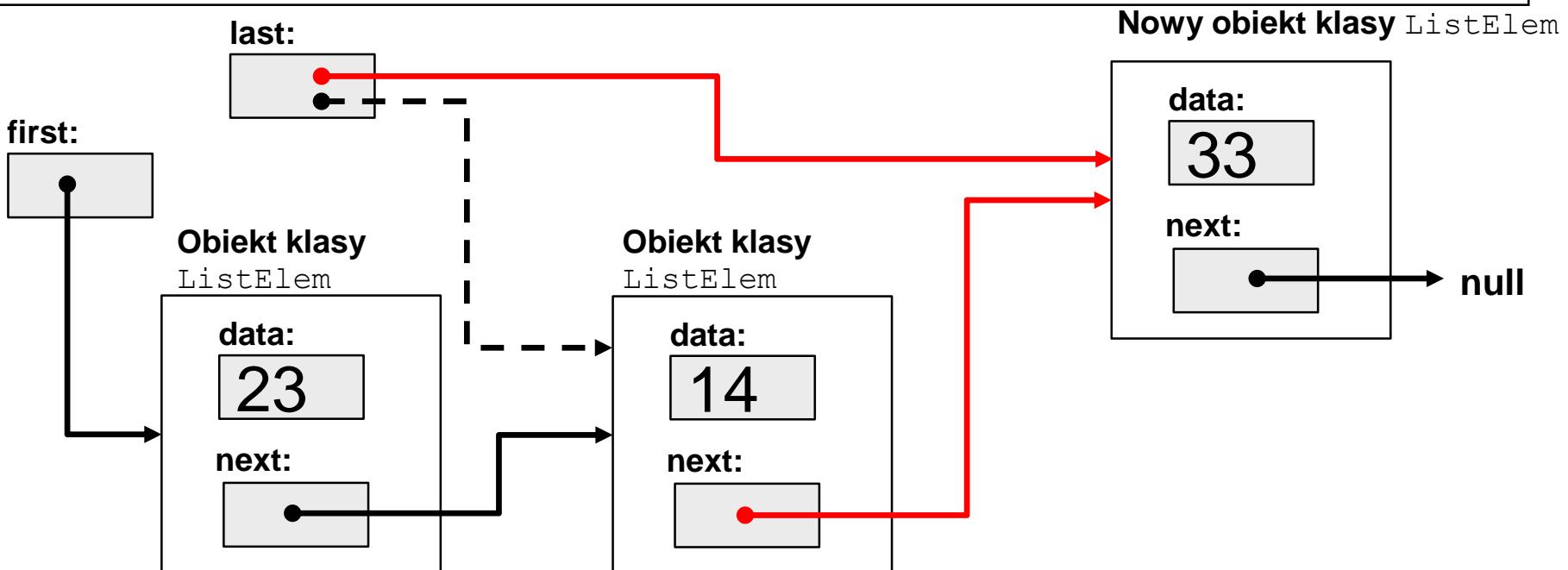
Lista powiązana dwustronna (FirstLastLinkedList)

- Na liście powiązanej dwustronnej istnieje dostęp zarówno do pierwszego, jak i do ostatniego elementu listy
- Dla zapewnienia takiego dostępu, w programie musi być jeszcze jedna zmienna typu tej samej klasy, która przechowuje referencję do ostatniego elementu listy (zmienna **last**)
- Dzięki dostępowi do ostatniego elementu listy możliwe jest szybkie wstawianie (nie usuwanie!) elementu na końcu listy



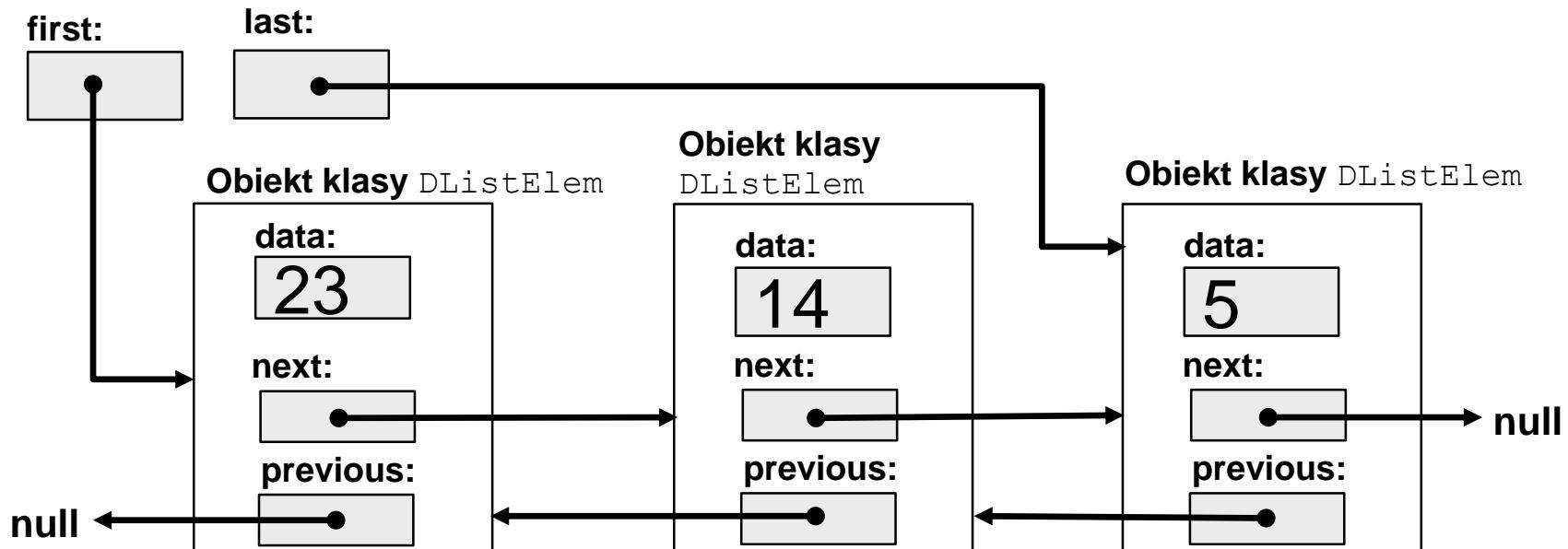
Wstawianie elementu na końcu do listy powiązanej dwustronnej liczb całkowitych

```
public void insertLast(int value) //Wstawienie na koniec listy
{
    ListElem newElem = new ListElem(value);
    if (isEmpty()) first = newElem;
    else last.next = newElem;
    last = newElem;
}
```



Lista powiązana dwukierunkowa (DoublyLinkedList)

- Na liście powiązanej dwukierunkowej istnieje dostęp zarówno do pierwszego, jak i do ostatniego elementu listy
- Ponadto istnieją powiązania elementów w obydwie strony
 - » Dla zapewnienia takiego dostępu w klasie elementu musi być jeszcze jedna zmienna typu tej samej klasy (zmienna **previous**)
- Dzięki dostępowi do ostatniego elementu listy możliwe jest szybkie wstawianie i usuwanie elementu na końcu listy



Przykład klasy elementu dla listy powiązanej dwukierunkowej

```
public class DListElem
{
    public int iData;          // Dana w elemencie listy (liczba całkowita)
    public DListElem next;     // Referencja do następnego elementu listy
    public DListElem previous; // Referencja do poprzedniego elementu listy

    public DListElem(int iData) // konstruktor
    {
        this.iData = iData;    // Inicjalizacja danych
        next = null;
        previous = null;
    }

    public void printElem()   // Wypisuje liczbę zawartą w elemencie listy
    {
        System.out.print(iData);
    }
}
```

Wstawianie elementu na początku i na końcu do listy powiązanej dwukierunkowej liczb całkowitych

```
public void insertFirst(int value) // Wstawienie na poczatek listy
{
    DListElem newElem = new DListElem(value);

    if (isEmpty()) last = newElem; //Wstawiamy pierwszy element listy
    else first.previous = newElem; //Poprzedni pierwszego to teraz nowy element

    newElem.next = first; //Nastepnym elementem wstawionego jest byly pierwszy
    first = newElem; //Wstawiony staje sie pierwszym
}

public void insertLast(int value) // Wstawienie na koniec listy
{
    DListElem newElem = new DListElem(value);
    if (isEmpty()) first = newElem; //Wstawiamy pierwszy element listy
    else
    {
        last.next = newElem; //Nastepny ostatniego to nowy element
        newElem.previous = last; //Poprzedni wstawionego to byly ostatni
    }
    last = newElem; //Wstawiony staje sie ostatnim
}
```

Usuwanie elementu na początku i końcu listy powiązanej dwukierunkowej liczb całkowitych

```
public DListElem deleteFirst()      // Usunięcie pierwszego elementu listy
{
    if (isEmpty()) return null;
    DListElem temp = first;
    if (first.next == null) last = null; //Był tylko jeden element na liście
    else first.previous = null; //Drugi element nie będzie miał poprzednika
    first = first.next; //Pierwszym elementem będzie był drugi element
    return temp;
}

public DListElem deleteLast()      // Usunięcie ostatniego elementu listy
{
    if (isEmpty()) return null;
    DListElem temp = last;
    if (first.next == null) first = null; //Był tylko jeden element na liście
    else last.previous.next = null; //Przedostatni element nie będzie miał następnika
    last = last.previous; //Ostatnim elementem będzie był przedostatni element
    return temp;
}
```

```

public static void main(String[] args) //test listy powiazanej
{
    LinkedList<String> list = new LinkedList<String>();
    list.addLast("Ala");
    list.addLast("Ola");
    list.addLast("Tola");
    list.addLast("Bolek");
    list.addLast("Lolek");
    list.addLast("Krzysiek");
    print(list);
    list.removeFirst();
    list.removeLast();
    print(list);
    list.remove("Tola");
    print(list);
    boolean resultTola = list.contains("Tola");
    System.out.println("Wynik dla Tola: "+resultTola);
    boolean resultBolek = list.contains("Bolek");
    System.out.println("Wynik dla Bolek: "+resultBolek);
    list.addLast("Czesiek");
    print(list);
    String first = list.getFirst();
    String last = list.getLast();
    System.out.println("Pierwszy: "+first);
    System.out.println("Ostatni: "+last);
}

```

```

public static void print(LinkedList<String> list)
{
    //Wypisywanie za pomoca iteratatora
    Iterator<String> iterator = list.iterator();
    while (iterator.hasNext())
    {
        String tekst = iterator.next();
        System.out.print(tekst+" ");
    }
    System.out.println();
}

```

Ala Ola Tola Bolek Lolek Krzysiek
Ola Tola Bolek Lolek
Ola Bolek Lolek
Wynik wyszukiwania dla Tola: false
Wynik wyszukiwania dla Bolek: true
Ola Bolek Lolek Czesiek
Pierwszy: Ola
Ostatni: Czesiek

**Przykład zastosowania
klasy LinkedList**
 (lista dwukierunkowa)

Podsumowanie wad i zalet list powiązanych

- **Zalety:**

- » Wykorzystuje pamięć tylko w takim stopniu, jak to jest konieczne (nie ma nadmiaru wykorzystania pamięci jak w przypadku tablic dynamicznych)
- » Możliwość przeglądania uporządkowanych danych, ale tylko dla listy uporządkowanej

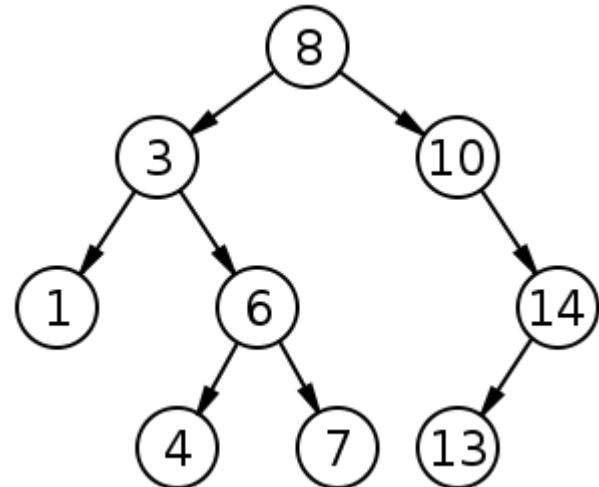
- **Wady:**

- » Trudniejsze programowanie niż dla tablic dynamicznych
- » Szybkość wyszukiwania i usuwania jest tylko liniowa
- » Dla listy uporządkowanej wzrasta czas wstawiania elementów

Drzewa poszukiwań binarnych

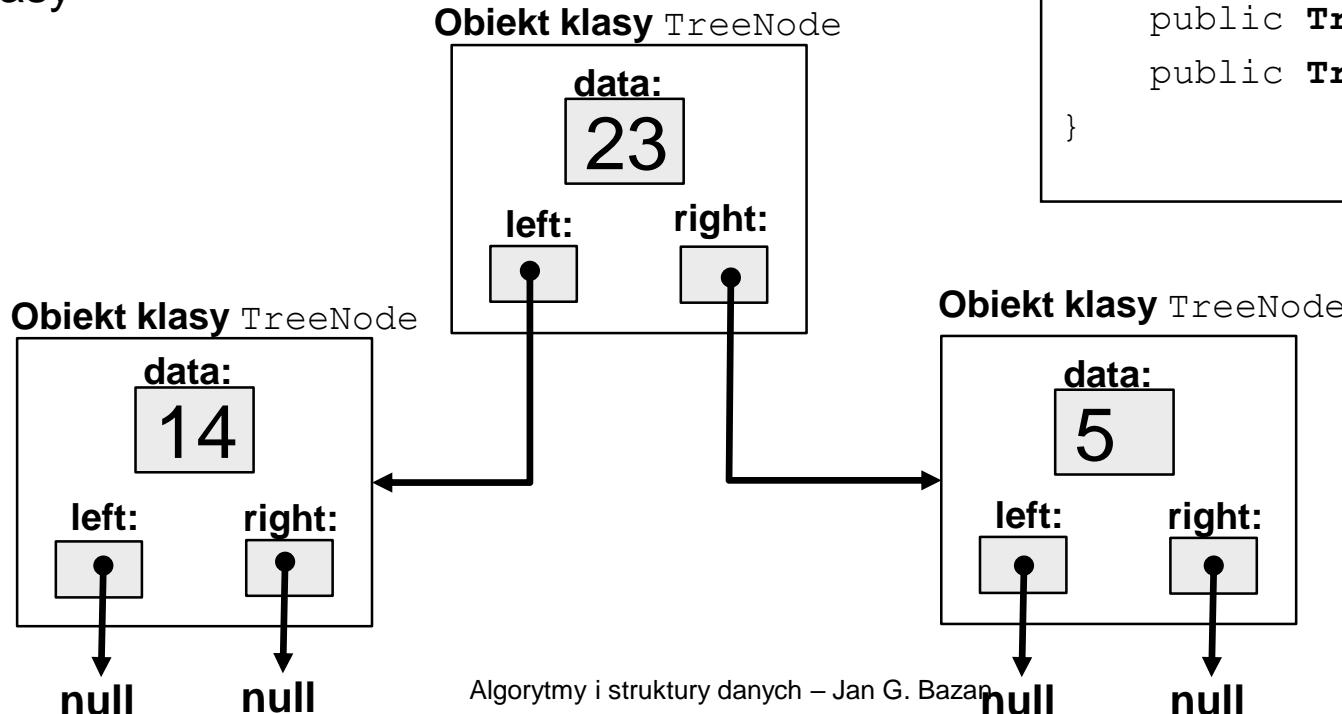
Drzewo poszukiwań binarnych

- Zwane pod nazwą drzewa BST (ang. Binary Search Tree)
 - » Struktura danych będąca drzewem binarnym, tzn. takim drzewem, że każdy węzeł drzewa ma co najwyżej dwa węzły będące jego dziećmi (synami)
- Z każdym węzłem wiąże się wartość klucza, która służy do konstrukcji struktury drzewa, ale także jest daną węzła
 - » W węźle mogą być także inne dane. np. jakieś teksty lub liczby ułamkowe
- Ponadto, w BST lewe poddrzewo każdego węzła zawiera wyłącznie elementy o kluczach nie większych niż klucz tego węzła, a prawe poddrzewo zawiera wyłącznie elementy o kluczach nie mniejszych niż klucz węzła.
- Węzły, oprócz klucza i ewentualnych danych, przechowują wskaźniki (adresy) na swojego lewego i prawego syna oraz na swojego ojca.



Połączenia pomiędzy obiektami jako sposób na reprezentację drzew binarnych

- Każda klasa może mieć dodatkowe dwa pola typu tej właśnie klasy
 - » W obiektach takie pola mogą przechowywać referencję do dwóch innych obiektów tej samej klasy
- Dzięki takim połączeniom można tworzyć w pamięci komputera drzewa elementów będących obiektami tej klasy



```
public class TreeNode  
{  
    public int data;  
    public TreeNode left;  
    public TreeNode right;  
}
```

Klasa TreeNode (węzeł drzewa BST)

Dodatkowy element w BST

```
public class TreeNode
{
    public int iData;                      // Element danych (klucz)
    public TreeNode parent;                 // Rodzic węzła
    public TreeNode left;                   // Lewy potomek węzła
    public TreeNode right;                  // Prawy lewy potomek węzła

    public TreeNode() //Konstruktor
    {
        iData = 0;
        parent = null;
        left = null;
        right = null;
    }
}
```

Klasa BSTree (początek)

```
public class TreeNode
{
    public int iData;          // Element danych (klucz)
    public TreeNode parent;    // Rodzic węzła
    public TreeNode left;      // Lewy potomek węzła
    public TreeNode right;     // Prawy lewy potomek węzła

    public TreeNode() //Konstruktor
    {
        iData = 0;
        parent = null;
        left = null;
        right = null;
    }
}
```

```
public class BSTree
{
    private TreeNode root; // pierwszy węzeł drzewa (korzeń)

    public BSTree() // konstruktor
    {
        root = null;
    }

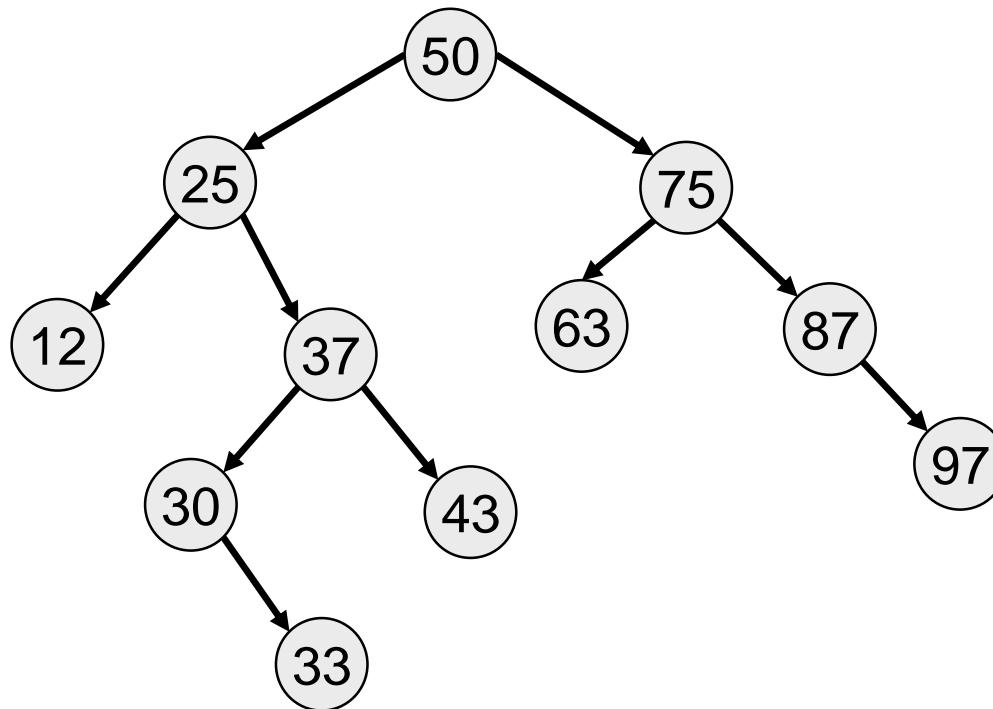
    ...
}
```

Wstawianie elementów do drzewa BST

- Nowe elementy wstawiane są jako liście drzewa w odpowiednim miejscu
- Wstawianie rozpoczyna się od korzenia drzewa
- Później przechodzi się po drzewie aż do węzła, w który jest wolne miejsce z lewej lub z prawej strony (w zależności od klucza wstawianego węzła), aby wstawić nowy węzeł
 - » Nowy węzeł wstawiany jest po lewej stronie bieżącego węzła, gdy jego klucz jest mniejszy od klucza bieżącego węzła
 - » Nowy węzeł wstawiany jest po prawej stronie bieżącego węzła, gdy jego klucz jest większy od klucza bieżącego węzła

Przykład tworzenia drzewa BST

- Utworzyć drzewo BST dla zbioru liczb całkowitych: {50, 25, 75, 12, 37, 43, 30, 33, 87, 63, 97}



```

public void insert(int elem)
{
    TreeNode newNode = new TreeNode();      // tworzymy nowy węzeł
    newNode.iData = elem;                  // zapisujemy w nim dane

    if (root == null) root = newNode; // Jesli drzewo jest puste
    else //Drzewo ma jednak korzen
    {
        TreeNode current = root;      // zaczynamy poszukiwania miejsca od korzenia
        while (true)                  // (zakończenie pętli wewnątrz za pomocą return)
        {
            TreeNode parent = current;
            if (elem < current.iData) // czy idziemy w lewo?
            {
                current = current.left;
                if (current == null) // jeśli nie ma lewego,
                {
                    parent.left = newNode; // wstawiamy po lewej stronie
                    newNode.parent = parent;
                    return;
                }
            }
            else // idziemy na prawo
            {
                current = current.right;
                if (current == null) // jesli nie ma prawego
                {
                    parent.right = newNode; // wstawiamy po
                    newNode.parent = parent; // prawej stronie
                    return;
                }
            }
        }
    }
}

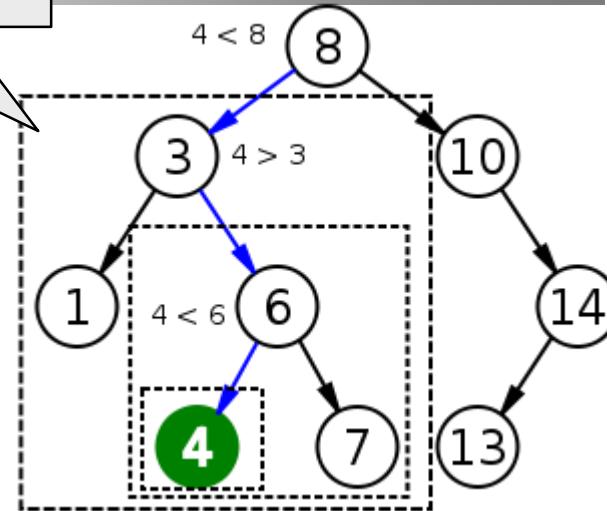
```

Wstawianie węzła do BST

Wyszukiwanie klucza w drzewie

Wyszukiwanie klucza 4

- Wyszukiwanie klucza rozpoczyna się od korzenia
- Następnie klucz każdego napotkanego węzła jest porównywany z poszukiwanym kluczem:
 - » jeżeli obie wartości są równe, to zwracana jest wartość **true** lub adres węzła ze znalezionym kluczem,
 - » jeżeli wartość poszukiwanego klucza jest mniejsza niż wartość klucza w porównywanym węźle to dalsze poszukiwania prowadzone są tylko w lewym poddrzewie;
 - » analogicznie, jeżeli wartość poszukiwanego klucza jest większa niż wartość klucza w porównywanym węźle to dalsze poszukiwania prowadzone są tylko w prawym poddrzewie.
- Jeżeli przeszukane zostaną wszystkie węzły i w żadnym nie będzie znaleziony klucz, to zwracana jest wartość **false**



Metoda find wyszukująca klucz w BST

```
public boolean find(int elem)      // sprawdzenie czy w zbiorze jest element: elem
{
    if (root==null) return false;      // puste drzewo

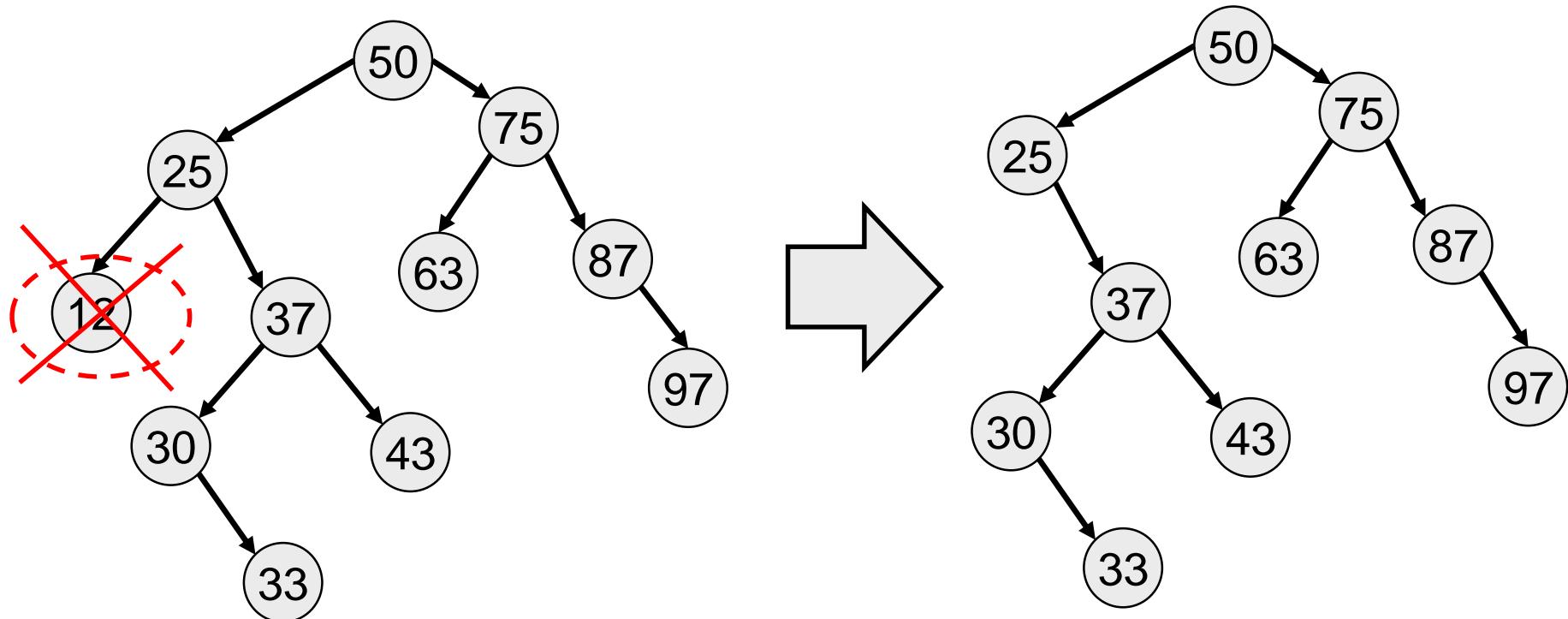
    TreeNode current = root;          // zaczynamy od korzenia drzewa
    while (current.iData != elem)      // dopóki nie odnaleziono elementu
    {
        if (elem < current.iData)      // czy przechodzimy na lewo?
        {
            current = current.left;
        }
        else // na prawo
        {
            current = current.right;
        }
        if (current == null)           // jeśli brak potomka,
        {
            return false;              // nie odnaleziono elementu
        }
    }
    return true; // odnaleziono element
}
```

Usuwanie węzła drzewa BST

- Usuwanie węzła jest procedurą bardziej skomplikowaną niż jego wstawianie, gdyż podczas wykonywania procedury należy rozważyć trzy przypadki:
 - » Usuwany węzeł nie ma synów (jest liściem)
 - » Usuwany węzeł ma jednego syna
 - » Usuwany węzeł ma dwóch synów

Usuwanie węzła, gdy węzeł nie ma synów (jest liściem)

- Wskaźnik do węzła w jego ojcu zastępowany jest wskaźnikiem do węzła pustego (poniżej usuwanie węzła z kluczem 12)

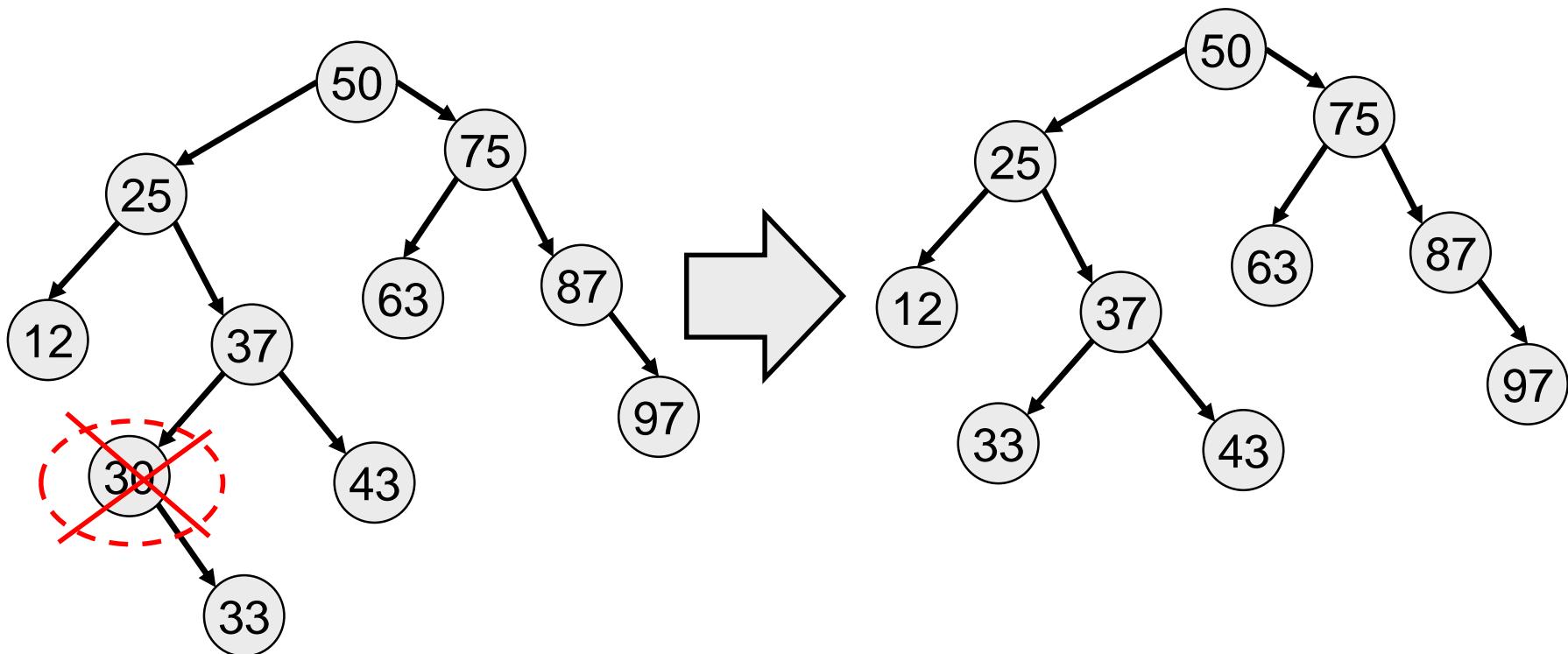


Metoda usuwająca węzeł bez potomków w BST

```
void deleteWithoutChilds(TreeNode current) //Usuniecie wezla bez potomkow
{
    if (current == root)                      // jeśli to korzeń,
    {
        root = null;                         // opróżniamy drzewo
    }
    else
    {
        if (current.parent.left == current) //usuwany element był lewym synem ojca
        {
            current.parent.left = null;     // odłączamy usuwany węzeł
        }
        else    //był prawym synem ojca
        {
            current.parent.right = null;   // odłączamy usuwany węzeł
        }
    }
}
```

Usuwanie węzła, gdy usuwany węzeł ma jednego syna

- Dany węzeł usuwamy a jego syna podstawiamy w miejsce usuniętego węzła (poniżej usuwanie węzła z kluczem 30)



Metoda usuwająca węzeł z jednym lewym potomkiem w BST

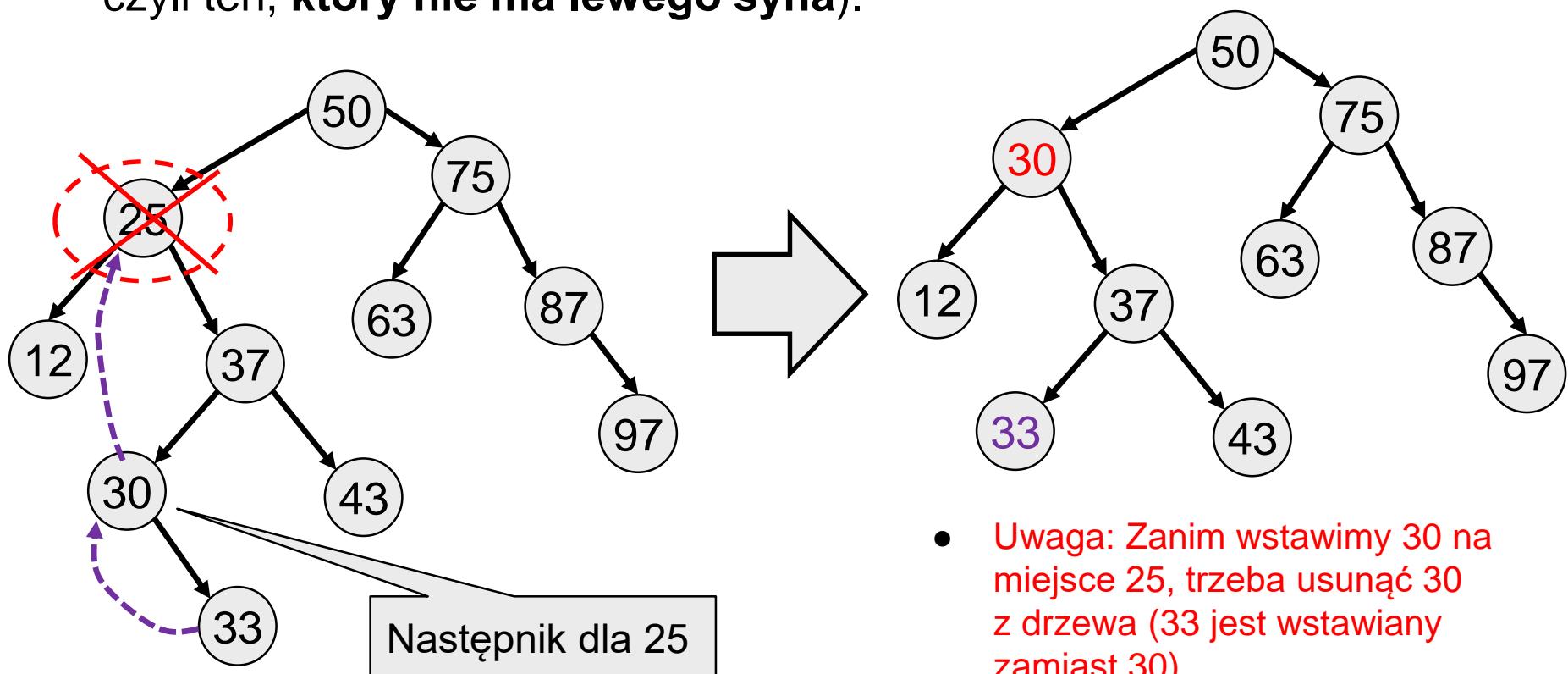
```
void deleteWithOneLeftChild(TreeNode current)
{
    if (current == root)
    {
        root = current.left;
    }
    else
        if (current.parent.left == current) //usuwany był lewym synem ojca
        {
            current.parent.left = current.left;
        }
        else //był prawym synem ojca
        {
            current.parent.right = current.left;
        }
}
```

Metoda usuwająca węzeł z jednym prawym potomkiem w BST

```
void deleteWithOneRightChild(TreeNode current)
{
    if (current == root)
    {
        root = current.right;
    }
    else
        if (current.parent.left == current) //usuwany był lewym synem ojca
        {
            current.parent.left = current.right;
        }
        else //był prawym synem ojca
        {
            current.parent.right = current.right;
        }
}
```

Usuwanie węzła, gdy usuwany węzeł ma dwóch synów

- Dany węzeł usuwamy a jego miejsce wstawiamy węzeł, który jest jego **następnikiem** (najmniejszy element większy od usuwanego, czyli ten, który **nie ma lewego syna**).



Metoda usuwająca węzeł z dwoma potomkami w BST

```
void deleteWithTwoChilds(TreeNode current)
{
    TreeNode successor = getSuccessor(current); //Wyszukanie następnika usuwanego wezla

    if (successor.left==null && successor.right==null) //nastepnik nie ma potomkow
    {
        deleteWithoutChilds(successor);
    }
    else //nastepnik ma jednego prawego potomka (nie moze miec lewego)
    {
        deleteWithOneRightChild(successor);
    }

    TreeNode parentCurr = current.parent;

    //Wymiana usuwanego wezla na jego nastepnik
    if (parentCurr.left==current) parentCurr.left = successor;
    else parentCurr.right = successor;

    successor.left = current.left;
    successor.right = current.right;
}
```

Wyszukanie następnika w BST

```
TreeNode getSuccessor(TreeNode node) //Wyszukanie nastepnika wezla
{
    TreeNode current = node.right;
    while (current.left != null) //Idziemy w lewy dolny lisc podrzewa
    {
        current = current.left;
    }
    return current;
}
```

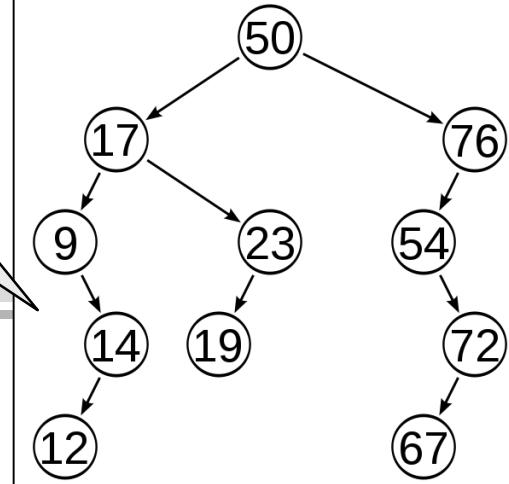
Usuwanie węzła drzewa BST

1. Wyszukanie węzła do usunięcia.
2. Stwierdzenie rodzaju węzła (3 rodzaje: bez potomków, jeden potomek, dwóch potomków)
3. Zastosowanie stosowanej procedury usunięcia (poprzednie slajdy)

Szczegóły dostępne w klasie *BSTree*.

Wyważanie drzewa

Drzewo BST
niezrównoważone



- Średni czas wykonywania operacji na drzewach BST zależy od średniej wysokości drzewa (tj. długości najdłuższej ścieżki od korzenia do liści).
- Najlepiej, gdy wynosi ona w przybliżeniu $\log_2 n$, gdzie n to liczba węzłów w drzewie.
 - » Powoduje to że zarówno w lewym i prawym poddrzewie jest mniej więcej tyle samo węzłów, a tym samym dojście do każdego liścia zajmuje mniej więcej tyle samo kroków.
- Tak jest w przypadku **drzewa zrównoważonego** - różnica wysokości lewego i prawego poddrzewa każdego z węzłów wynosi co najwyżej 1.
- Drzewo jest **doskonale zrównoważone**, gdy dodatkowo wszystkie liście znajdują się na najwyższej dwóch poziomach.
- Struktury gwarantujące zrównoważenie to drzewa AVL i drzewa czerwono-czarne.
 - » Posiadają one bardziej skomplikowane reguły wstawiania i usuwania, jak również przechowują dodatkowe wartości pomocnicze w węzłach drzewa.

Złożoność obliczeniowa operacji na drzewie poszukiwań binarnych (BST)

- Niech n będzie liczbą elementów wstawionych do BST

Wstawienie elementu	Wyszukanie elementu	Usunięcie elementu
$O(\log n)$	$O(\log n)$	$O(\log n)$

- Wyszukiwanie elementu jest po ścieżce
- Wstawienie elementu wymaga wyszukania miejsca na ten element
- Usunięcie elementu wymaga wyszukania elementu i wykonania algorytmu usunięcia w czasie stałym lub w czasie $\log n$ (wyszukanie następnika)

Podsumowanie wad i zalet drzew binarnych

- **Zalety:**
 - » Szybkie wstawianie, wyszukiwanie i usuwanie elementów (znacznie szybsze niż w przypadku tablic dynamicznych i list powiązanych), szczególnie do danych o charakterze losowym
 - » Możliwość przeglądania uporządkowanych danych
- **Wady:**
 - » Drzewo powinno być zrównoważone; jeśli tak nie będzie, to może bardzo wzrosnąć czas wyszukania i usuwania elementów (nawet do złożoności $O(n)$)

Tablice mieszające

Tablice mieszające

- **Tablica mieszająca lub tablica z haszowaniem** (ang. *hash table*) to struktura danych, która jest jednym z najważniejszych sposobów przechowywania informacji, w taki sposób aby możliwy był do nich bardzo szybki dostęp
- Odwołania do przechowywanych elementów struktury dokonywane są na podstawie **klucza**, który dany element identyfikuje
- Tablice mieszające opierają się na **zwyczajnych tablicach indeksowanych liczbami**
- Dostęp do danych jest bardzo szybki i nie zależy od rozmiaru tablicy ani położenia elementu (w czasie stałym)

Funkcja mieszająca

- W tablicy mieszającej stosuje się funkcję mieszającą, która dla danego klucza wyznacza indeks komórki w tablicy, gdzie dany element powinien się znajdować
 - » Innymi słowy funkcja mieszająca przekształca klucz w liczbę z zadanego zakresu.
- Funkcje mieszające są zwykle nieskomplikowane, tak aby czas ich wykonywania nie dominował w operacjach na tablicy.
- Funkcję mieszającą dobiera się do typu klucza
 - » Przykład dla liczb całkowitych
 - Dzielenie modulo przez wielkość tablicy mieszającej –patrz dalej
 - » Przykład dla tablic przechowujących nazwiska (dowolnie długie łańcuchy znaków)
 - Poprzez złożenie kodów dwójkowych poszczególnych znaków oraz wykonanie dzielenia modulo powstałej liczby przez wielkość tablicy mieszającej,

Przykład funkcji mieszającej dla klucza będącego liczbą naturalną

- Najprostszym przykładem funkcji mieszającej jest funkcja postaci: $h(x) = x \bmod m$, gdzie x jest liczbą naturalną, a m rozmiarem tablicy mieszającej
- Przykłady obliczeń funkcji mieszającej dla $m=7$

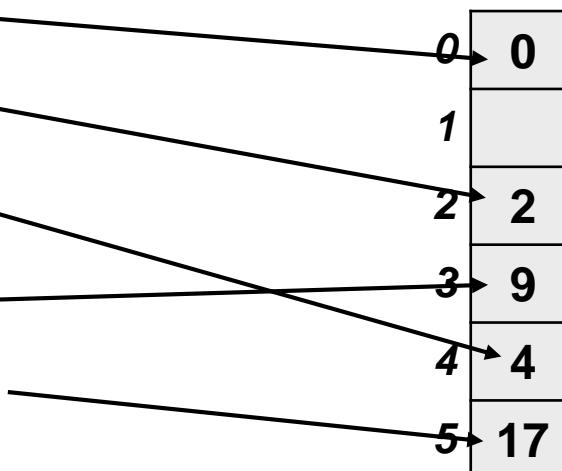
» $x=0: h(x) = 0 \bmod 6 = 0$

» $x=2: h(x) = 2 \bmod 6 = 2$

» $x=4: h(x) = 4 \bmod 6 = 4$

» $x=9: h(x) = 9 \bmod 6 = 3$

» $x=17: h(x) = 17 \bmod 6 = 5$



Wyznaczanie numeru komórki dla klucza

Jak obliczać funkcje mieszające dla wartości różnych typów?

- Każda klasa opakowująca (np. Integer, Float, Double, Long) oraz klasa String mają funkcję mieszającą `int hashCode()`
- Zaleca się korzystanie z tych funkcji zarówno w przypadku pojedynczych wartości opakowywanych typów, ale także w przypadku wartości o bardziej złożonej strukturze
 - » Np. dla par wartości można liczyć sumę funkcji mieszających
- W przypadku złożonych wartości definiujemy klasę, której obiekty reprezentują taką złożoną wartość i w tej klasie definiujemy metodę `int hashCode()`
 - » Zawsze należy jednak zapewnić, aby zakres wartości funkcji mieszającej nie przekraczał rozmiarów tablicy
 - » UWAGA: Najlepiej jest, aby w takim przypadku funkcja mieszająca rozrzucała złożone wartości w miarę równomiernie do wszystkich list wewnętrznych w tablicy mieszającej

Wyszukiwanie elementów oraz kolizja

- W najprostszym przypadku wartość funkcji mieszającej, obliczona dla danego klucza (klucz reprezentuje element), wyznacza dokładnie indeks szukanej informacji w tablicy.
- Jeżeli miejsce wskazywane przez obliczony indeks jest puste, to poszukiwanej informacji nie ma w tablicy.
 - » Wyszukiwanie elementu ma zatem złożoność czasową $O(1)$
- Problem pojawia się wówczas, gdy dla dwóch różnych kluczy funkcja mieszająca ma taką samą wartość
 - » Taką sytuację określany mianem **kolizji**
 - $x=0: h(x) = 2 \text{ mod } 6 = 2$
 - $x=2: h(x) = 8 \text{ mod } 6 = 2$
- **Pytanie:** Gdzie umieścić drugi element, gdy pierwszy zajął już miejsce wyznaczone przez funkcję mieszającą?

Metody rozwiązania problemu kolizji: *Zastąpienie lub rezygnacja ze wstawienia*

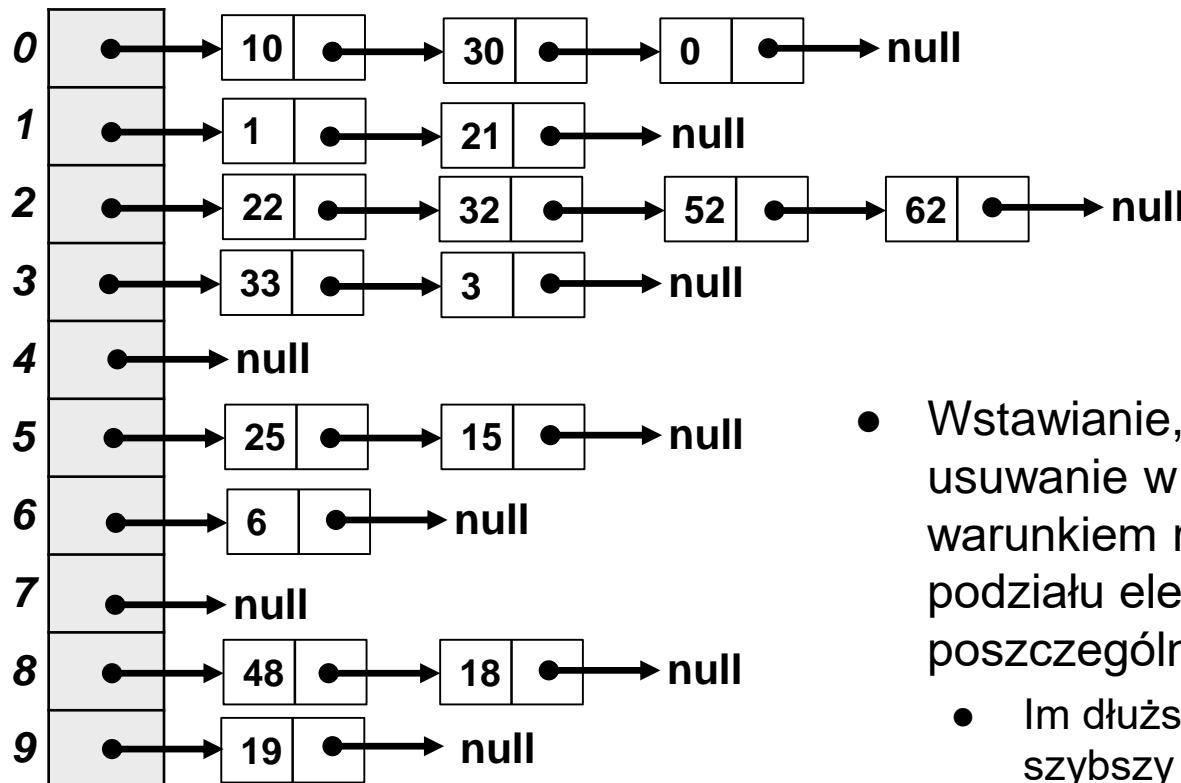
- Najprostszym sposobem jest **zastąpienie elementu** znajdującego się w tablicy przez nowy element (poprzedni element jest tracony)
- Drugi sposób polega na **rezygnacji z wstawiania nowego elementu** (drugi element nie będzie wstawiony do tablicy)
- Ponieważ zwykle wymagane jest, aby oba elementy znalazły się w tablicy, powyższe dwie metody **nie mają praktycznego znaczenia**

Metody rozwiązania problemu kolizji: Metoda łańcuchowa

- **Metoda łańcuchowa** (zwana także **adresowanie zamknięte**) polega na przechowywaniu elementów nie bezpośrednio w tablicy, lecz na liście związanego z danym indeksem.
 - » Wstawiane elementy dołączają się do jednego z końców listy.
- Średnia złożoność wyszukiwania jest złożonością liniowym wyszukiwaniem elementu na liście i zależy od współczynnika wypełnienia listy, czyli stosunku liczby elementów do wielkości tablicy.
- Ponieważ złożoność pesymistyczna wyszukiwania wynosi $O(n)$, czasami zamiast list stosuje się drzewa binarne.
- Zaletą metody łańcuchowej jest szybkość i prostota usuwania elementów z listy.
- Wadą pesymistyczny czas wyszukiwania i usuwania elementów z listy - $O(n)$

Przykład zastosowania metody łańcuchowej do przechowywania liczb naturalnych przy funkcji mieszającej postaci: $h(x) = x \bmod 10$

- Zbiór: {10, 30, 0, 1, 21, 22, 32, 52, ,62, 33, 3, 25, 15, 6, 48, 18, 19}



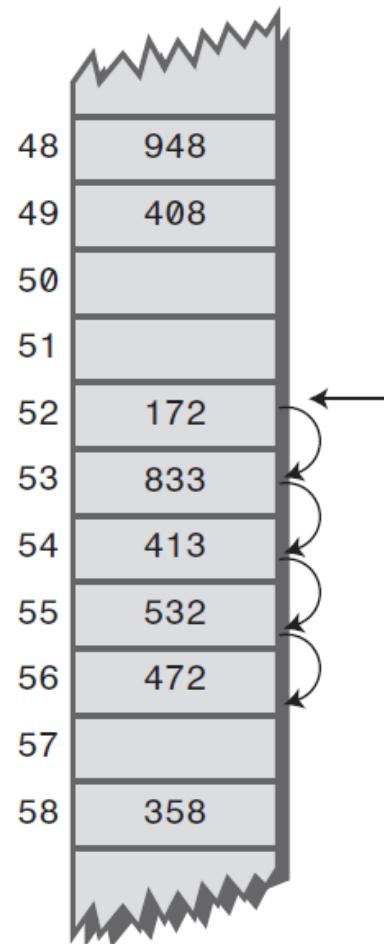
- Wstawianie, wyszukiwanie i usuwanie w czasie $O(1)$ pod warunkiem równomiernego podziału elementów na poszczególne listy
 - Im dłuższa tablica, tym szybszy czas dostępu

Metody rozwiązania problemu kolizji: *Adresowanie otwarte*

- W podejściu tym, w przypadku kolizji, nowy element wstawia się w innym miejscu niż wynikałoby to z wartości funkcji mieszającej.
 - » Nowa lokalizacja określana jest przez dodanie do wartości funkcji mieszającej wartości tzw. funkcji przyrostu $p(i)$, gdzie i oznacza numer próby (to znaczy ile razy wstawienie się nie powiodło ze względu na kolizję); otrzymaną liczbę należy jeszcze podzielić modulo przez rozmiar tablicy m .
- Ze względu na rodzaj funkcji przyrostu wyróżnia się następujące metody adresowania otwartego:
 - » **szukanie (sondowanie) liniowe**, dla funkcji przyrostu postaci $p(i)=i$
 - jeśli $h(x)$ jest wartością funkcji mieszającej, to kolejno sprawdzane są komórki o numerach $h(x)$, $(h(x)+1) \bmod m$, $(h(x)+2) \bmod m$, $(h(x)+3) \bmod m$ itd.
 - » **szukanie (sondowanie) kwadratowe**, dla $p(i)=i^2$
 - jeśli $h(x)$ jest wartością funkcji mieszającej, to kolejno sprawdzane są komórki o numerach $h(x)$, $(h(x)+1) \bmod m$, $(h(x)+4) \bmod m$, $(h(x)+16) \bmod m$ itd.
 - » **mieszanie podwójne**, dla $p(i)=i * h' (K)$, gdzie h' jest dodatkową funkcją mieszającą

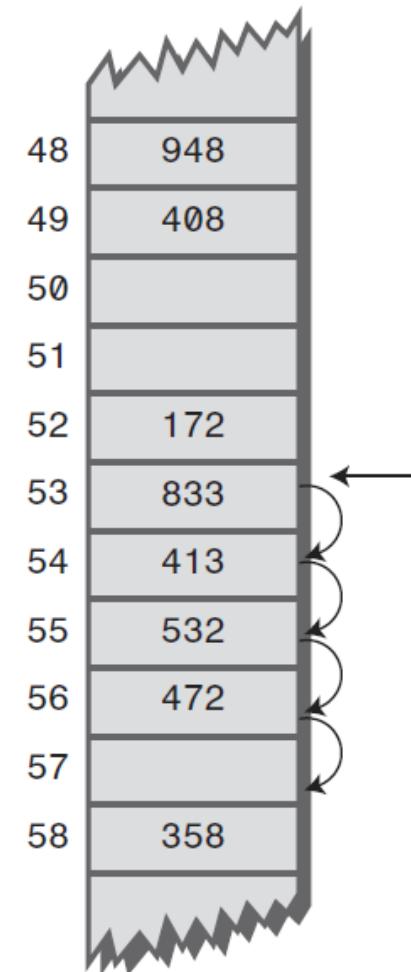
Przykład szukania liniowego dla istniejącego w tablicy klucza

- Funkcja mieszającej postaci: $h(x) = x \bmod 60$
- Szukamy miejsca dla klucza $x=472$
- $h(x)=472 \bmod 60 = 52$
- Ponieważ na miejscu numer 52 jest już inny element, sondowane są kolejno miejsca numer $52+1=53$, $52+2=54$, $52+3=55$ i $52+4=56$ (pomijam mod 60)
- Na miejscu numer 56 zostaje znaleziony klucz 472, co potwierdza istnienie klucza w tablicy



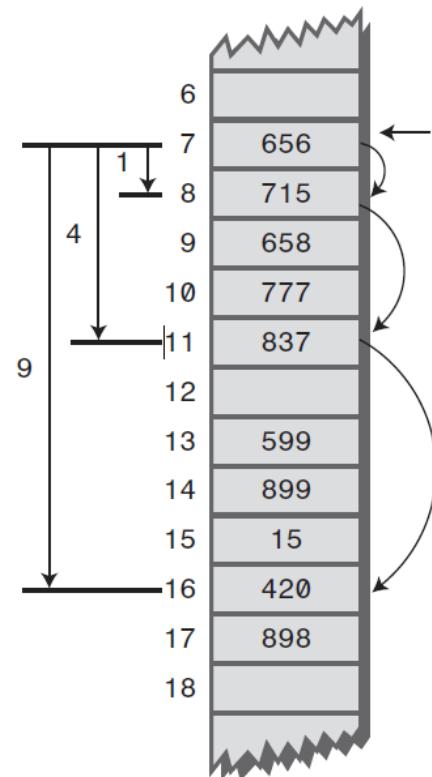
Przykład szukania liniowego dla nie istniejącego w tablicy klucza

- Funkcja mieszającej postaci: $h(x) = x \bmod 60$
- Szukamy miejsca dla klucza $x=893$
- $h(x)=893 \bmod 60 = 53$
- Ponieważ na miejscu numer 53 jest już inny element, sondowane są kolejno miejsca numer 54, 55, 56 i 57
- Na miejscu numer 57 jest puste miejsce, co oznacza, że klucza 893 nie ma w tablicy



Przykład szukania kwadratowego dla istniejącego w tablicy klucza

- Funkcja mieszającej postaci: $h(x) = x \bmod 59$
 - » 59 jest liczbą pierwszą i tak powinno być, bo inaczej sondowanie może się zapętlić
- Szukamy miejsca dla klucza $x=420$
 - » $h(x)=420 \bmod 59 = 7$
- Ponieważ na miejscu numer 7 jest już inny element, sondowane są kolejno miejsca numer $7+1*1=8$, $7+2*2=11$ i $7+3*3=16$ (pomijam mod 59)
- Na miejscu numer 16 zostaje znaleziony klucz 420, co potwierdza istnienie klucza w tablicy



Problemy nierównomiernego rozmieszczania elementów w tablicy i usuwania elementów

- W przypadku szukania liniowego może pojawić się **problem nierównomiernego rozmieszczania elementów w tablicy**, to znaczy koncentracji miejsc zajętych w pewnych zakresach indeksów przy małej zajętości innych obszarów tablicy.
- Problem ten jest w znacznym stopniu **zredukowany w przypadku szukania kwadratowego**, chociaż w tej metodzie występuje analogiczny problem wtórny, natomiast praktycznie wyeliminowany on jest dla mierzania podwójnego.
- Innym kłopotem jest **skomplikowanie procesu usuwania** elementu, w sytuacji gdy w tablicy znajdują się inne elementy, o tej samej wartości funkcji mieszanającej.
 - » Wymusza to rozróżnianie trzech stanów elementów tablicy: **zajęta, wolna, wolna po usunięciu**.

Współczynnik wypełniania tablicy mieszającej

- Współczynnik wypełniania tablicy mieszającej (ang. *load factor*) jest definiowany jako iloraz liczby elementów zapisanych w tablicy mieszającej (m) do fizycznego rozmiaru tablicy (n).
- Jeśli rozkład prawdopodobieństwa wartości funkcji skrótu jest jednostajny, wówczas przeciętnie dla $\alpha=m/n$ elementów wystąpią kolizje.
- Przy inicjalizacji tablicy mieszającej podaje się początkowy rozmiar n (lub jest ona niejawnie określona przez implementację).
- Natomiast podczas pracy z tablicą sprawdzany jest aktualny współczynnik wypełnienia i gdy jest za duży, rozmiar fizyczny tablicy zostaje odpowiednio korygowany.
- W praktyce przyjmuje się wartość współczynnika wypełnienia na poziomie $\alpha = 0.7 \dots 0.8$.

Implementacja tablicy mieszającej metodą łańcuchową

- Pokaz klasy

structures.hashtable.IHashTable
w NetBeans

Złożoność obliczeniowa operacji na tablicy mieszającej

- Niech n będzie liczbą elementów w tablicy mieszającej

Wstawienie elementu	Wyszukanie elementu	Usunięcie elementu
$O(1)$	$O(1)$	$O(1)$

- Wszystkie trzy operacje wymagają jedynie obliczenia wartości funkcji mieszającej, co jest wykonywane w czasie stałym

Klasa HashSet jako standardowa kolekcja języka Java implementująca tablicę mieszającą elementów parametryzowanego typu oznaczanego przez <typ> (wybrane możliwości)

- Konstruktory:
 - » `HashSet<typ>()` – bezparametrowy konstruktor,
 - » `HashSet<typ>(int initialCapacity)` – konstruktor z początkowym rozmiarem tablicy mieszajacej,
- `void add(<typ> elem)` – wstawienie elementu do struktury
- `int size()` – pobranie informacji o aktualnej liczbie wstawionych elementów
- `boolean contains(<typ> elem)` – pobranie informacji, czy podany element występuje w strukturze (wartości true i false)
- `boolean remove(<typ> elem)` – usunięcie podanego elementu (zwrota true, gdy został rzeczywiście usunięty),
- `void clear()` – usunięcie wszystkich elementów kolekcji

Podsumowanie wad i zalet tablic mieszających

- **Zalety:**
 - » Bardzo szybkie wstawianie, wyszukiwanie i usuwanie elementów
- **Wady:**
 - » Kolejność danych jest zupełnie nieprzewidywalna (nie nadaje się dla sytuacji, gdy chcemy mieć dane uporządkowane).
 - » Dane powinny być równomiernie rozproszone w tablicy mieszającej za pomocą funkcji mieszającej; jeśli tak nie będzie, to może bardzo wzrosnąć czas wyszukania i usuwania elementów (nawet do złożoności $O(n)$, gdy wszystkie elementy pójdą na tę samą listę)

Porównanie efektywności konkretnych struktur danych

Struktura danych	Wyszukiwanie	Wstawianie	Usuwanie	Przeglądanie według porządku
Tablica dynamiczna	$O(n)$	$O(1)$	$O(n)$	NIE
Tablica dynamiczna uporządkowana	$O(\log n)$	$O(n)$	$O(n)$	TAK
Lista powiązana	$O(n)$	$O(1)$	$O(n)$	NIE
Lista powiązana uporządkowana	$O(n)$	$O(n)$	$O(n)$	TAK
Drzewo binarne średnio	$O(\log n)$	$O(\log n)$	$O(\log n)$	TAK
Tablica mieszająca	$O(1)$	$O(1)$	$O(1)$	NIE

Poprawność algorytmów

Poprawność algorytmu

- Poprawność algorytmu jest to jego cecha wyrażająca się spełnianiem przez ten algorytm narzuconych mu wymagań (specyfikacji)
 - » Poprawny algorytm daje takie odpowiedzi, jakich oczekujemy
- Algorytm musi być poprawny, aby miało sens rozpatrywanie jego złożoności.

Algorytmy niepoprawne

- Algorytm niepoprawny to taki, który nie spełnia zadanej specyfikacji
- Niezgodność ze specyfikacją wynika z popełnionych błędów podczas konstrukcji algorytmu
- Ze względu na moment zauważenia błędu rozróżnia się następujące rodzaje błędów w algorytmie i w programie:
 - » Błędy składniowe
 - » Błędy wykonania
 - » Błędy logiczne

Błędy składniowe

(syntaktyczne, gramatyczne)

- Powstają w wyniku naruszenia reguł składni języka programowania, którego używamy do zapisania algorytmu
- Zgłasza je kompilator podczas translacji i są zwykle łatwe do usunięcia
- Jeśli występują, program nie może być wykonany

Błędy wykonania

- Zgłaszane przez środowisko uruchomieniowe w czasie wykonania programu
- Nie są łatwe do znalezienia i usunięcia
- Komunikaty o nich mogą być mylące

Przykład:

Dzielenie przez zero – jeśli program chce wykonać dzielenie przez zero, to jego działanie jest przerywane z odpowiednim komunikatem.

Błędy logiczne

Jeśli program kompliluje się i wykonuje bez komunikatów o błędach, nie znaczy to jeszcze, że jest on napisany poprawnie!

- Merytoryczne błędy w projekcie programu, w stosowanych algorytmach lub ich implementacji
- Kod źródłowy kompliluje się bez błędów oraz
 - » brak komunikatów o błędach podczas wykonania
 - » brak niewłaściwych zdarzeń lub danych podczas uruchamiania
- Najtrudniejsze do znalezienia
- Wykrycie ich wymaga często mozolnego śledzenia zachowania się programu lub nawet powtórnej analizy jego merytorycznej poprawności

Przykłady błędów logicznych

- Błąd algorytmu:

- » sredniaPieciuWartosci = suma/2
(powinno być dzielenie przez 5)

- Błąd implementacji:

- » błędnie wprowadzone w kodzie źródłowym
suma = a - b;
(powinno być suma = a + b;)

Spektakularne skutki błędów w programach komputerowych

- **Załamanie projektu Mars Climate Orbiter**, gdyż parametry silników rakietowych zostały błędnie obliczone i sonda wkroczyła do atmosfery Marsa pod złym kątem i spłonęła.
 - » Przyczyną było błędne określenie jednostek, w jakich podawane były parametry fizyczne - jedna grupa inżynierów pracowała w jednostkach angielskich, druga w metrycznych.
- **Katastrofa rakiety Ariane** już po 40 sekundach od startu.
 - » Przyczyną był błąd konwersji danych między 64-bitowymi danymi zmiennoprzecinkowymi a 16-bitowymi całkowitymi. Skutkiem było zejście rakiety z kursu i eksplozja.
- **Awaria w oprogramowaniu sterującym akceleratorem Therac-25** przeznaczonym do radioterapii. Pacjenci mogli być narażeni na dawki wielokrotnie większe od zaplanowanego napromienowania leczonego narządu.
- W 1983 r. wiele błędów w niedostatecznie przetestowanym oprogramowaniu kontroli przestrzeni powietrznej poskutkowało **fałszywym alarmem radzieckiego systemu obrony przeciwraziowej** (nieprawdziwa informacja o kilku pociskach wystrzelonych przez USA).



Dowodzenie poprawności algorytmów

- Sposobem na wyeliminowanie błędów w algorytmach jest dowodzenie ich poprawności
- Dowód poprawności algorytmu jest rozumowaniem matematycznym prowadzącym do formalnego wykazania, że dany algorytm przy poprawnych danych wejściowych da nam wynik spełniający wyspecyfikowane przez nas wymagania
- Do dowodzenia poprawności algorytmów wykorzystywane są zazwyczaj pewne formalizmy matematyczne, z których większość opiera się o logikę Hoare'a
 - » Przy tym podejściu dowód poprawności algorytmu polega na udowodnieniu jego tzw. semantycznej poprawności

Warunek początkowy i końcowy - pojęcia wprowadzające do semantycznej poprawności

- Z dowolnym algorytmem A wiążemy dwa następujące warunki logiczne: **warunek początkowy α** oraz **warunek końcowy β**
 - » Przez warunek logiczny rozumiemy pewną formułę logiczną, która może mieć dwie wartości: prawda (true) i fałsz (false)
- **Warunek początkowy α** - opisuje własność poprawnych danych na wejściu algorytmu A (mówi jakich danych trzeba użyć dla uruchomienia algorytmu A)
- **Warunek końcowy β** - opisuje własność poprawnych wyników algorytmu oraz ich związek z danymi

Algorytm semantycznie poprawny

- Mówimy, że algorytm A jest **semantycznie (znaczeniowo) poprawny** wtedy i tylko wtedy, gdy dla każdych danych wejściowych spełniających warunek początkowy α obliczenie algorytmu **dochodzi do końca** i jego **wynik spełnia warunek końcowy β**
- Aby wykazać, że algorytm jest semantycznie poprawny należy udowodnić trzy następujące jego własności:
 1. własność częściowej poprawności względem warunku początkowego α i końcowego β ,
 2. własność określoności,
 3. własność stopu.

Własność częściowej poprawności algorytmu

- Mówimy, że algorytm A **jest częściowo poprawny** względem warunku początkowego α i końcowego β wtedy i tylko wtedy, gdy dla konkretnych danych wejściowych spełniających α jeżeli algorytm dochodzi do końca obliczeń, to to jego wynik spełnia warunek β .
- **Wskazówka:** Do udowodnienia tego faktu używamy **metodę niezmienników iteracji**, która polega na sprawdzaniu w określonych miejscach algorytmu (zwykle po obliczeniu istotnej wartości danego zadania w danej iteracji pętli) czy spełniona jest pewna funkcja zdaniowa F zależna od wartości m , gdzie przez m rozumiemy m -tą iterację algorytmu (taka funkcję F nazywamy **niezmiennikiem**).
 - » Dowód przeprowadzamy indukcyjnie ze względu na liczbę iteracji pętli.
 - » Niezmiennik pętli traktowany jest jako założenie indukcyjne, na podstawie którego wykazuje się prawdziwość kroku indukcyjnego.

Własność określoności

- Mówimy, że algorytm A **posiada własność określoności**, gdy dla każdych danych wejściowych spełniających α , obliczenie algorytmu nie jest przerwane, gdyż wszystkie jego operacje są dobrze określone.
- **Wskazówka:** Dla dowodu własności określoności algorytmu trzeba pokazać, że wszystkie operacje w algorytmie są dobrze określone i zawsze dają się wyliczyć.
 - » Jeśli w algorytmie brak jest obliczeń funkcji, które mają ograniczoną dziedzinę (np. operacja dzielenia, logarytmowanie, pierwiastkowanie itp.), można powiezieć, że własność określoności jest trywialnie spełniona.
 - » Jeśli jednak są takie funkcje, to trzeba przeprowadzić analizę, czy nie zagrożą one własności określoności.
 - Np. jeśli w algorytmie jest dzielenie, to należy pokazać, że nigdy nie będzie dzielenia przez 0.

Własność stopu

- Mówimy, że algorytm **posiada własność stopu** wtedy i tylko wtedy, gdy dla każdych danych wejściowych spełniających α obliczenie jest skończone
 - » Program zatrzymuje się dla wszystkich możliwych danych
- **Wskazówka:** Własność stopu dowodzimy jedną z dwóch poniższych metod:
 - » metoda liczników iteracji,
 - » metoda malejących wielkości
 - » **Obydwie metody dotyczą sytuacji, gdy w algorytmie mamy pętlę.**

Metody dowodzenia własności stopu

- **Metoda liczników iteracji** - polega na tym, że staramy się wyliczyć ile razy zostanie wykona iteracja danej pętli.
 - » Jeśli nam się to uda, to wnioskujemy, że iteracje pętli kiedyś się zakończą, co oznacza, że obliczenie będzie skończone.
 - Np. w programie szukającym maksimum z tablicy liczb całkowitych, liczba iteracji będzie równa liczbie komórek tablicy, a zatem będzie skończona
- **Metoda malejących wielkości** - opiera się na założeniu, że jeśli w kolejnych iteracjach pętli dana wielkość jest coraz mniejsza i dodatkowo jest ograniczona od dołu, to pętla kiedyś się zakończy, co oznacza, że obliczenie będzie skończone.
 - » Np. w programie wyszukującym binarnie liczbę w tablicy uporządkowanej, w każdej iteracji pętli dzielimy dany fragment tablicy (w której aktualnie szukamy) na dwie w miarę równe części, by w kolejnej iteracji szukać liczby tylko w jednej z tych dwóch części
 - Zatem pętla musi się zakończyć, bo dzielić tablicę nie można w nieskończoność; w końcu po podziale pozostało do przeszukania jednoelementowy fragment tablicy i po jego sprawdzeniu pętla się zakończy

Nierozstrzygalność problemu stopu

- Problem stopu dla ustalonego algorytmu bywa trudny do rozstrzygnięcia
- **Twierdzenie: Nie istnieje uniwersalny algorytm rozstrzygający problem stopu dla wszystkich algorytmów.**

- Dowód „nie wprost”: Załóżmy, że istnieje taki program o nazwie stop, który działa zgodnie z pseudokodem:

- Korzystając z procedury stop można by jednak utworzyć nowy program test, który dla dowolnego programu program zatrzymuje się wtedy i tylko wtedy, kiedy program zapętla się na swoim własnym kodzie podanym jako dane wejściowe. Jego pseudokod miałby postać

```
procedura stop(program, dane):  
    jeżeli program(dane) zatrzymuje się, to  
        zwróć tak,  
    w przeciwnym przypadku  
        zwróć nie.
```

- Powstaje pytanie: czy program test zatrzymuje się po otrzymaniu swojego własnego kodu jako danych wejściowych?
- Jeżeli test zatrzymuje się na danych test, to stop zwraca tak dla programu test uruchomionego z danymi test, czyli test zapętla się dla danych test, co jest sprzeczne z założeniem o skończości działania stop.
- Z kolei jeżeli test zapętla się dla danych test, to stop zwraca nie dla programu test z danymi test, czyli test się zatrzymuje dla danych test, co stoi w sprzeczności z założeniem o zapętleniu się kombinacji tego programu i danych.
- W ten sposób założenie o istnieniu programu stop prowadzi do sprzeczności, skąd wynika, iż problem stopu jest nierozstrzygalny.

```
procedura test(program):  
    jeżeli stop(program, program) = tak, to  
        zapętl się.
```

Przykład: Semantyczna poprawność algorytmu obliczającego sumę kwadratów pierwszych n liczb naturalnych

Rozwiązanie w osobnym dokumencie PDF.

Zadania domowe

- Napisać algorytm w pseudojęzyku, który oblicza silnię pierwszych n liczb naturalnych dla ustalonej liczby naturalnej n oraz udowodnić poprawność semantyczną tego algorytmu.
- Napisać algorytm w pseudojęzyku, który wyznacza minimum danego ciągu liczb całkowitych (założyć, że cały ciąg pojawia się na wejściu algorytmu w wyniku wykonania jednej instrukcji `read(tab)`, gdzie `tab` jest tablicą do której są czytane elementy ciągu). Udowodnić poprawność semantyczną tego algorytmu.
- Napisać algorytm w pseudojęzyku, który sprawdza, czy w danym ciągu liczb całkowitych znajduje się podana liczba (wyszukiwanie liniowe). Założyć, że cały ciąg pojawia się na wejściu algorytmu w wyniku wykonania jednej instrukcji `read(tab)`, gdzie `tab` jest tablicą do której są czytane elementy ciągu. Poszukiwana liczba także jest wczytywana za pomocą instrukcji `read`. Udowodnić poprawność semantyczną tego algorytmu.

Praktyka

- W praktyce opracowywania programów komputerowych pomija się (niestety) etap badania całkowitej poprawności algorytmu i bada się „produkt końcowy”, czyli sam program:
 - » testowanie na licznych zestawach danych wejściowych (dla takiego zestawu powinien być znany wynik końcowy)
 - » uruchamianie w obecności narzędzi diagnostycznych (badanie stanów pośrednich i końcowych)
- Narzędzia programistyczne: asercje, testy aplikacji

Asercje

- **Asercja** (ang. assertion) to **predykat** (forma zdaniowa w danym języku), która **zwraca prawdę lub fałsz**
 - » Umieszczony w pewnym miejscu w kodzie.
- Wskazuje, że programista zakłada, że predykat ów jest w danym miejscu **prawdziwy**.
 - » W przypadku gdy predykat jest fałszywy (czyli nie spełnione są warunki postawione przez programistę) asercja powoduje przerwanie wykonania programu (rzucenie wyjątku).
- Asercja ma szczególne zastosowanie w trakcie testowania tworzonego oprogramowania, np. **sprawdzenie luk** lub **odporności na błędy**
 - » Zaletą jej stosowania jest możliwość sprawdzenia, w którym fragmencie kodu źródłowego programu nastąpił błąd.
 - » Pozwala na upewnienie się, że pewne założenia co do określonych wyrażeń są prawdziwe.

```
package asercje;

import java.util.Scanner;

public class ProstyPrzyklad
{

    public static void main(String args[])
    {
        Scanner skaner = new Scanner(System.in);

        //assert false : "Tutaj nie moze byc sterowanie";

        System.out.print("Wprowadz liczbe pomiedzy 0 i 20: ");

        int liczba = skaner.nextInt();

        assert (liczba >= 0 && liczba <= 20) : "Nieporownana liczba: " + liczba;

        /*
        if (liczba<0 || liczba>20)
        {
            throw new IllegalArgumentException("Liczba poza zakresem: " + liczba);
        }
        */

        System.out.println("Wprowadziles:"+ liczba);
    }
}
```

Przykład

Testowanie aplikacji

(dwie grupy testów)

- Testy "czarnoskrzynkowe".

- » Tester otrzymuje aplikację lub jej moduł i stara się ją doprowadzić do niepoprawnego działania
- » Nie wie jak testowany element jest zbudowany, a tylko wie jaką logikę powinien spełniać.
- » Zaleta: Możliwość wyszukania nietypowych zachowań użytkownika.
 - Testy integracyjne, testy interfejsu użytkownika, testy akceptacyjne.
- » Wada: niska automatyzacja procesu testowego

- Testy "białoskrzynkowe".

- » Zazwyczaj są przeprowadzane wewnętrznie zespołu programistów.
- » Mogą one obejmować zarówno całe moduły i być traktowane jako testy obciążeniowe/wydajnościowe, ale mogą też dotyczyć niewielkich fragmentów kodu takich jak pojedyncze klasy czy metody (testy jednostkowe).

Przeznaczenie testów jednostkowych

- Test jednostkowy służy do sprawdzenia pojedynczej jednostki kodu
 - » Jednostką taką zazwyczaj jest metoda.
- Testy organizowane są w zestawy, które obejmują pojedyncze klasy.

Trzy sposoby przeprowadzania testów „białoskrzynkowych”

- „**Słabe debuggowanie**” - wypisywanie na standardowe wyjście stanu systemu w pewnych miejscach (`System.out.println()`).
 - » Szeroko stosowane przez osoby początkujące jak i profesjonalistów
 - » Wady: konieczność komplikacji kodu za każdym razem gdy wykonujemy test oraz po zakończeniu testu i usunięciu metod wypisujących (komplikacja kodu po zakończeniu testów oznacza, że może się on zmienić!).
- **Debugger** - brak możliwości automatyzacji wyników, duży nakład pracy przy projektowaniu samego śledzenia.
- **Mini aplikacja** – wywołująca kilkukrotnie testowany kod i następnie weryfikująca wyniki.
 - » Podejście jest zdecydowanie lepsze ponieważ pozwala na automatyzację całego procesu testów.
 - » Zamiast samemu pisać kod można użyć biblioteki JUnit, która zawiera już odpowiednie mechanizmy związane z przygotowaniem, uruchamianiem i weryfikowaniem wyników testu (będzie na programowaniu zespołowym).
 - » **Imitacje obiektów** – obiekty prowizorycznych klas testowych, które zastępują obiekty rzeczywiste na czas uruchamiania i testowania kodu
 - » **Atrapy obiektów** – mechanizm naśladujący klasy testowe bez ich tworzenia (zasada magnetowidu, który nagrywa co jest potrzebne do testowania, a później zapis jest odczytywany i podawany na testowany moduł)

Przypadki testowania

- Pisząc testy należy przyjąć pewne założenia.
 - » Jeżeli wiemy co dana aplikacja ma robić należy określić przypadki testowania aplikacji
- **Przypadki typowe** - testowanie przypadków typowego użycia aplikacji
 - » Np. średnia kilku liczb różnych od zera: {3, 4, 5, 3}
- **Przypadki niepoprawne** (błędne dane, dane nieobsługiwane przez aplikację)
 - » Np. średnia zbioru {}, średnia zbioru {1, J, 3, s}
- **Przypadki graniczne i specyficzne** – mogą zostać zinterpretowane jako prawidłowe albo nieprawidłowe
 - » Np. średnie zbiorów: {2}, {0}, {2, 5, 0}, {3, 5}