Algorytmy i struktury danych – W03

Teoria złożoności cz. 3/4,

stos, kolejka

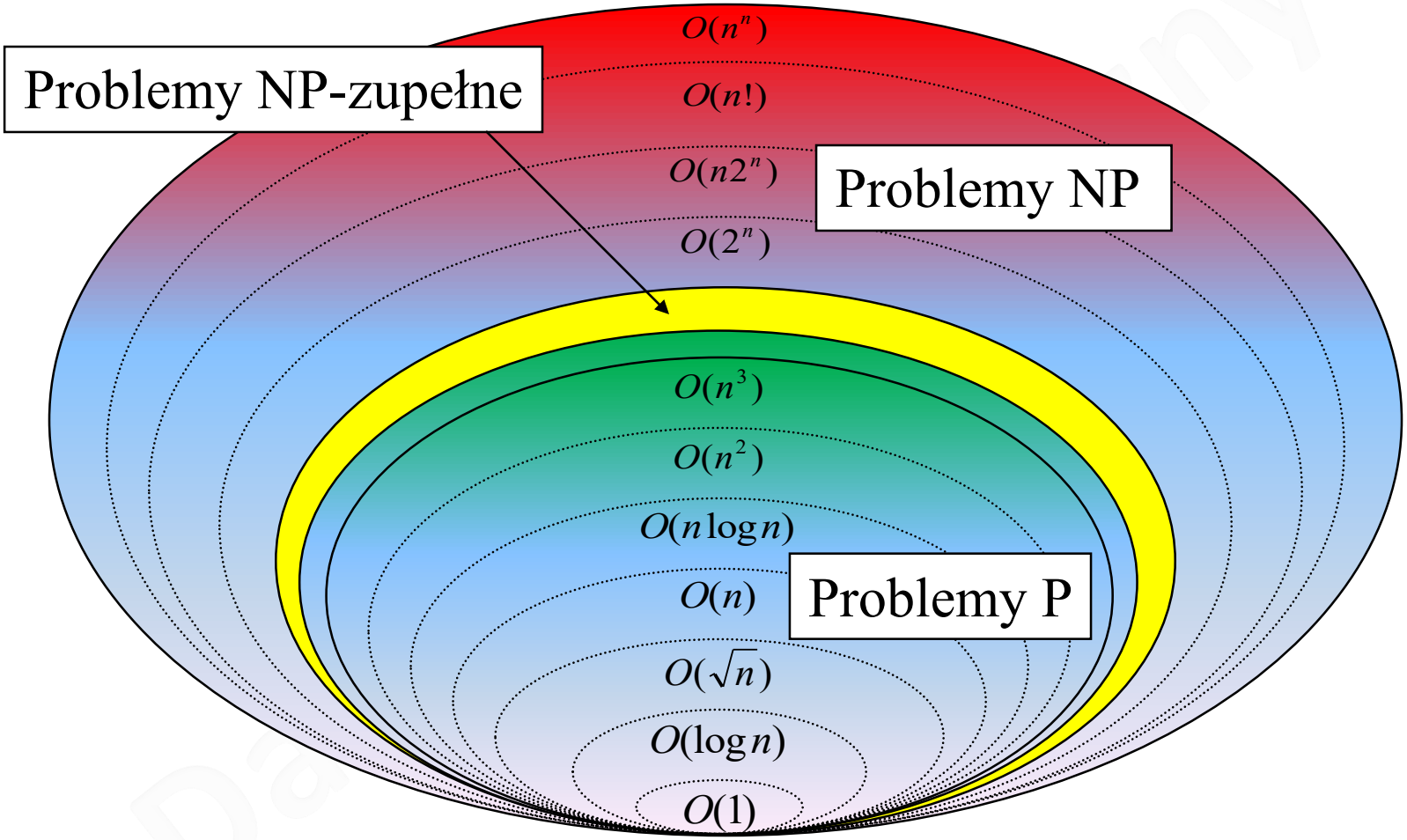
Dariusz Konieczny

Zawartość

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

Dariusz Konieczny

Hierarchia rzędów – klasy złożoności



Właściwości 1/2

- Przechodniość

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

$$f(n) = \mathrm{O}(g(n)) \text{ i } g(n) = \mathrm{O}(h(n)) \text{ implikuje } f(n) = \mathrm{O}(h(n))$$

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

- Zwrotność

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

$$f(n) = \mathrm{O}(f(n))$$

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

- Symetria

$$f(n) = \Theta(g(n)) \text{ wtw } g(n) = \Theta(f(n))$$

- Symetria przechodnia

$$f(n) = \mathrm{O}(g(n)) \text{ wtw } g(n) = \Omega(f(n))$$

Dariusz Konieczny

Właściwości 2/2

- Inne właściwości

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

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

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

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

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

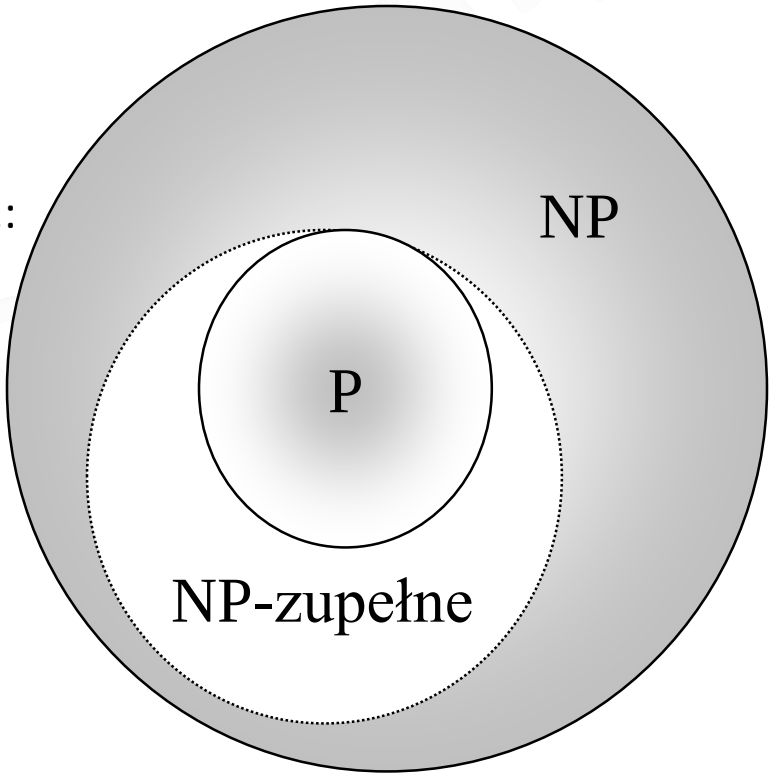
$$O(f(n) \cdot g(n)) = f(n) \cdot O(g(n))$$

- Zawsze należy pamiętać, że $O(\ldots)$, $\Theta(\ldots)$, $\Omega(\ldots)$ oznacza zbiory funkcji i znak równości jest wieloznaczny:
  - Oznacza równość zbiorów, gdy po obu stronach znaku równości są zbiory
  - Oznacza należenie funkcji do zbioru, gdy jedna ze stron jest funkcją

Dariusz Konieczny

Klasyfikacja problemów

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



Dariusz K

Porównanie złożoności 1/3

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

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

Wiek wszechświata = $13,82*10^9$ lat

Darius Wróblewski

Porównanie złożoności 2/3

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

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

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

Dariusz Wardzinski

Porównanie złożoności 3/3

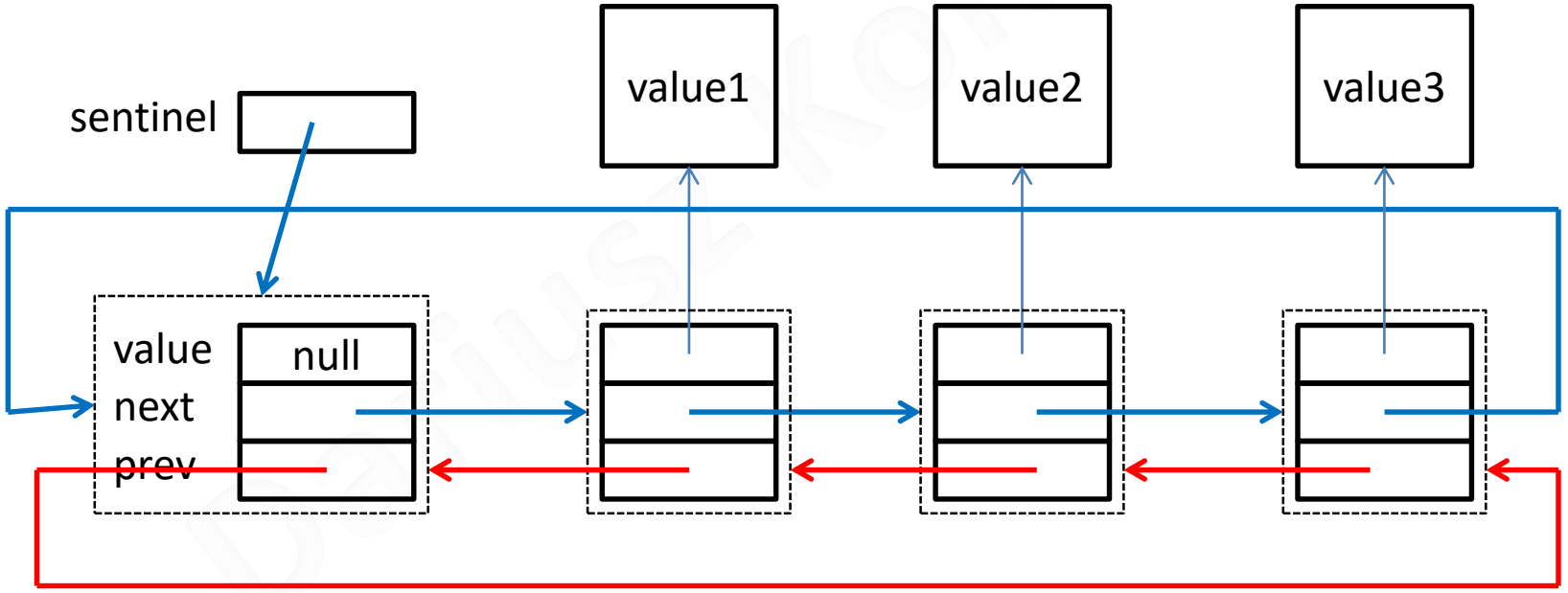
- Szybszy sprzęt obliczeniowy jest ważny, ale nie rozwiąże problemów z nieefektywny algorytmem lub złożonym obliczeniowo problemem.
- Poniżej „analiza”, o ile zwiększylibyśmy rozmiar wejściowego problemu, gdyby udało się odpowiednio przyspieszyć komputer. Jaki największy problem możemy wtedy rozwiązać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

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

Dariusz Konieczny

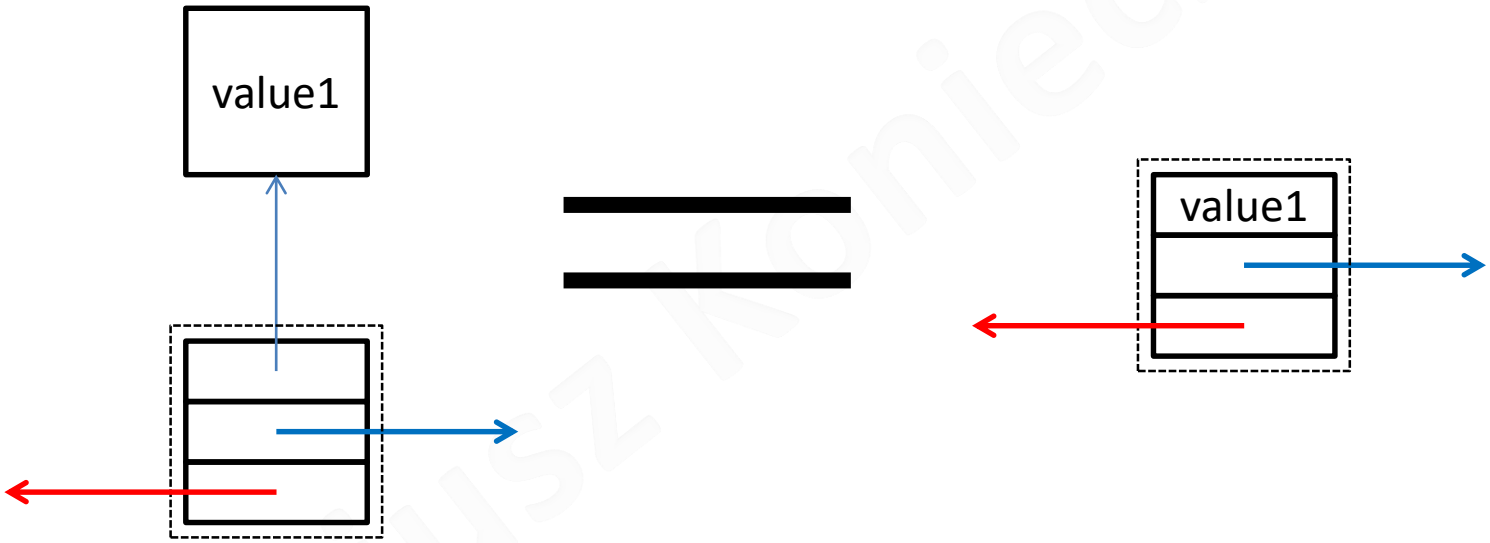
Lista L2KCzS

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



Uproszczenie graficzne

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



Dariusz Konieczny

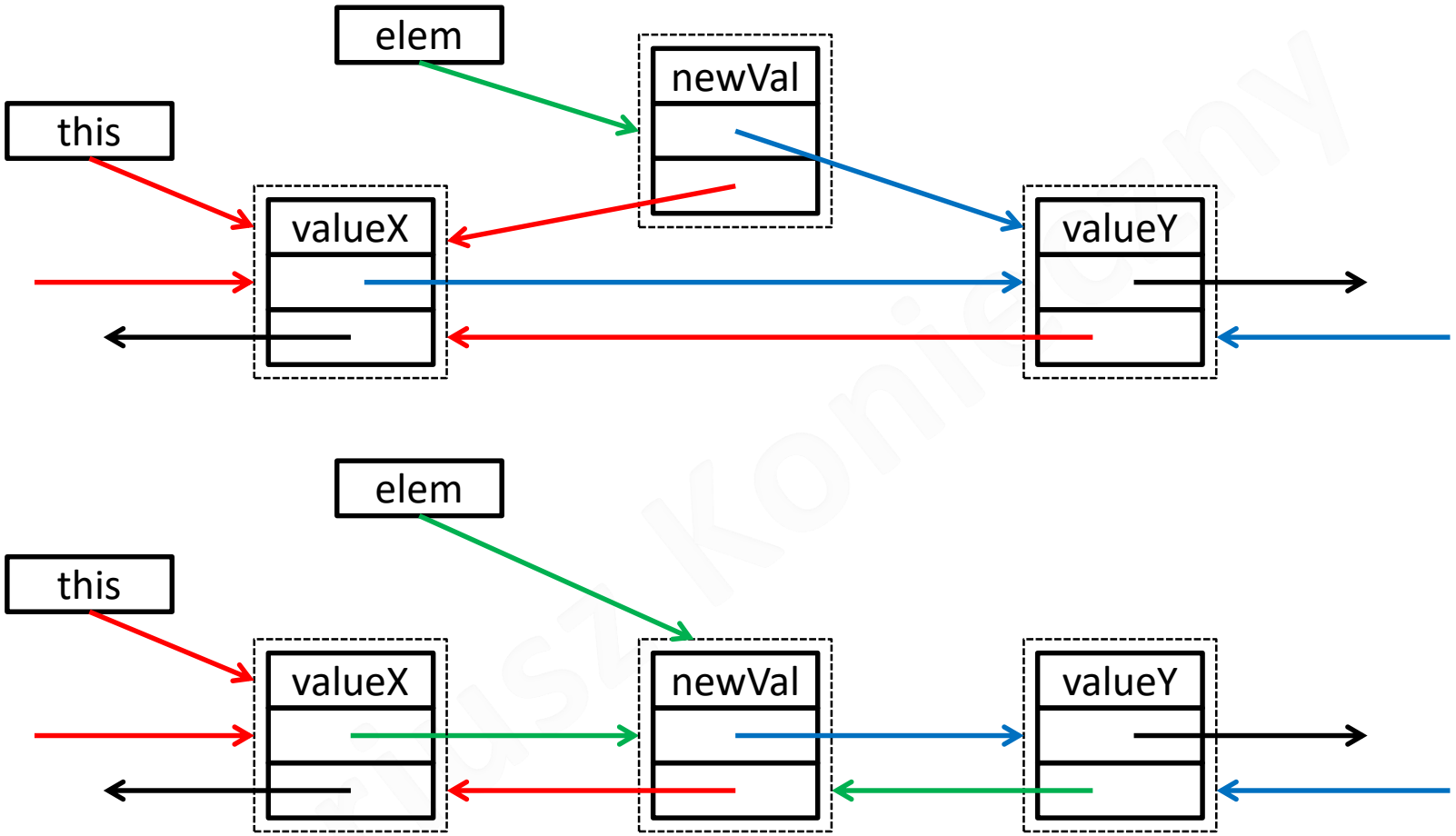
Lista L2KCzS - `Element`

aisd.list.TwoWayCycledListWithSentinel

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

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

Wersja Konieczny

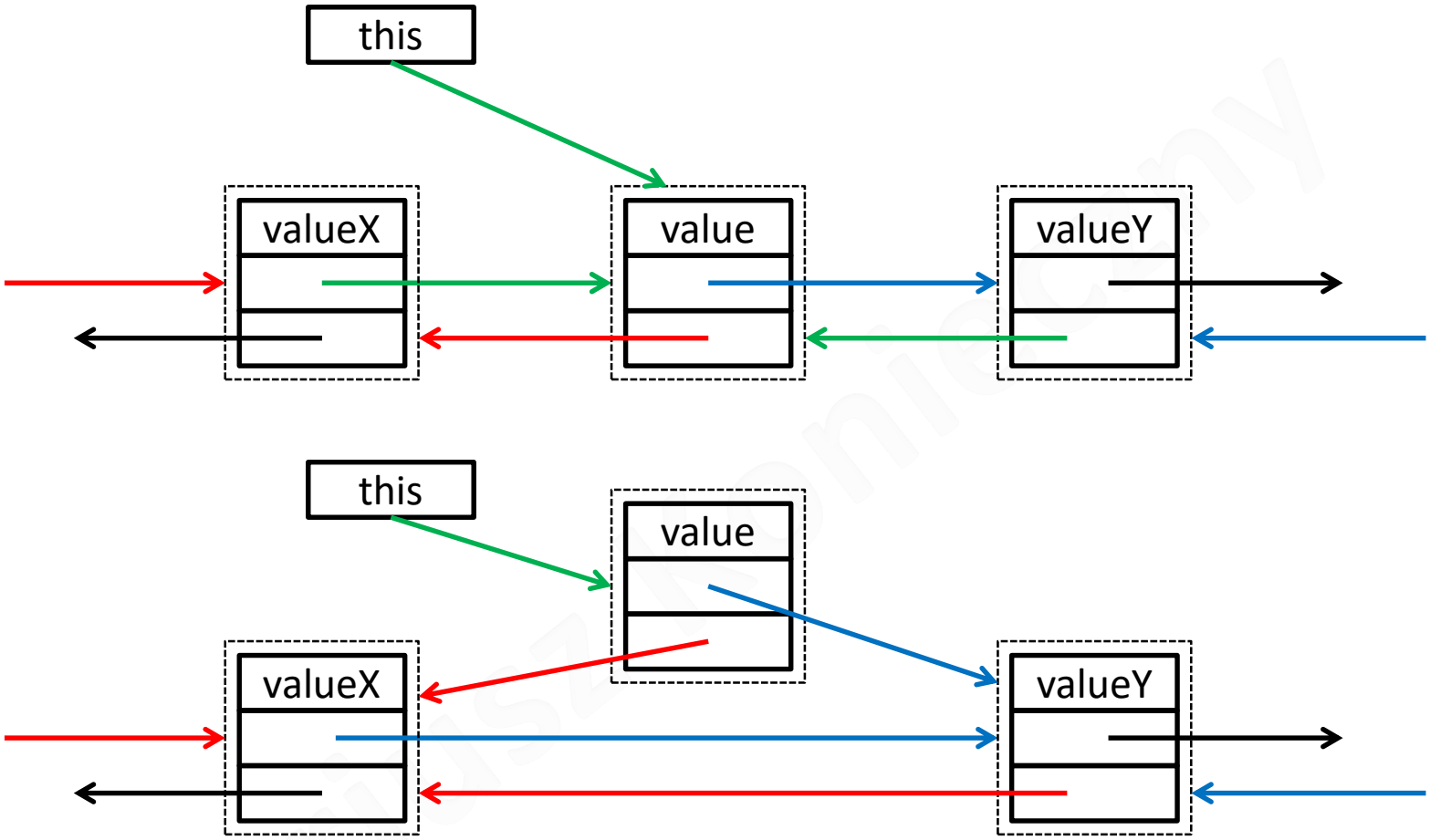
insertAfter()



- Analogicznie przebiega wykonanie metody `insertBefore()`

Dariusz Konieczny

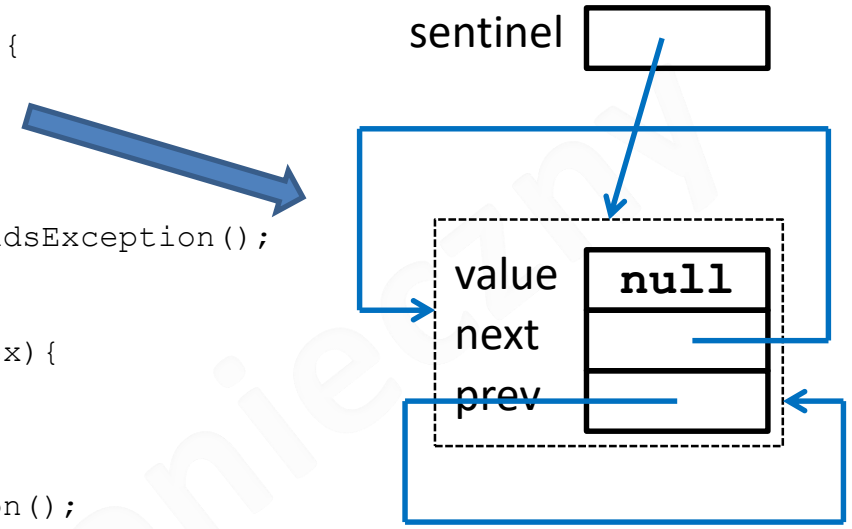
remove()



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

TwoWayCycledListWithSentinel 1/5

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



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

Dariusz Konieczny

TwoWayCycledListWithSentinel 2/5

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

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

Daniel Konieczny

TwoWayCycledListWithSentinel 3/5

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

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

Dariusz Konieczny

TwoWayCycledListWithSentinel 4/5

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

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

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

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

Daniel Konieczny

TwoWayCycledListWithSentinel 5/5

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

Dariusz Konieczny

Lista L2KCzS – złożoność

- Lista L2KCzS – złożoność pesymistyczna:
  - Konstrukcja – O(1)
  - `isEmpty()`,`clear()` – O(1)
  - `getElement()` – O(n)
    - `set()`,`get()` – O(n)
  - `add()` na końcu – O(1)
  - `add()` z indeksem – O(n) – z zerowym – O(1)
  - `indexOf()` – O(n)
    - `contains()` – O(n)
  - `remove()` x 2 – O(n) – z zerowym – O(1)
  - `size()` – O(n)
- `ListIterator` dla L2KCzS – gdyby zaimplementować wszystkie operacje - złożoność pesymistyczna:
  - `hasNext()`,`next()`,`hasPrevious()`,`previous()`,`nextIndex()`, `previousIndex()` – O(1)
  - `set()` – O(1)
  - `add()`,`remove()` – O(1)

- Lista L2KCzS nadaje się do implementacji nieograniczonej kolejki, gdzie każda operacja (oprocz size()) jest stałoczasowa - O(1).

Bartosz Nieczny

Przykład – lista studentów 1/2

- Należy dopisać porównywanie studentów

aisd.W03.Student

```
public class Student{
  int indexNo;
  double scholarship;
  public Student(int nr, double value){
    indexNo=nr;
    scholarship=value;
  }
  public void increaseScholarship(double value){
    scholarship+=value;
  }
  @Override
  public String toString(){
    return String.format("%6d %8.2f\n", indexNo, scholarship);
  }

  public boolean equals(Student stud) {
    return indexNo == stud.indexNo;
  }

  @Override
  public boolean equals(Object obj) {
    if(obj==null)
      return false;
    if (getClass() != obj.getClass())
      return false;
    return equals((Student) obj);
  }
```

Dariusz Konieczny

Przykład – lista studentów 2/2

- Należy dopisać porównywanie studentów

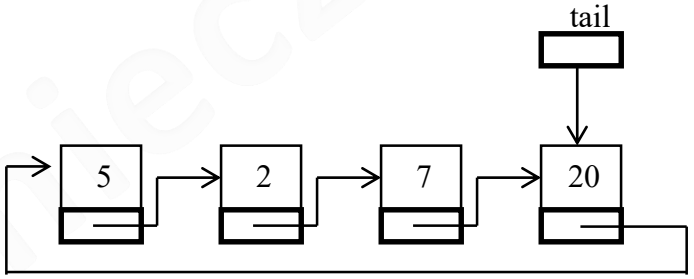
```
public static void testStudentList(){
  TwoWayCycledListWithSentinel<Student> lista=new TwoWayCycledListWithSentinel<>();
  lista.add(new Student(4,1000));
  lista.add(new Student(5,500));
  lista.add(new Student(20,0));
  lista.add(1,new Student(1, 300));
  lista.remove(0);
  lista.remove(new Student(20,10000));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

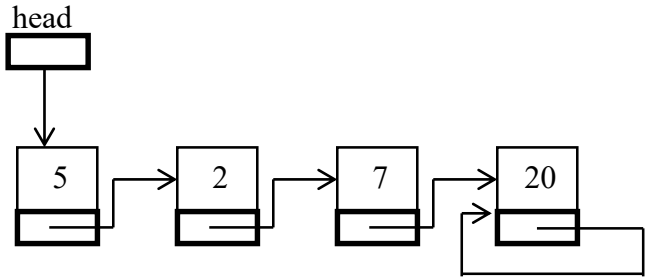
aisd.W03.AiSD_W03

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

Listy wiązane - rodzaje

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





Dariusz Konieczny

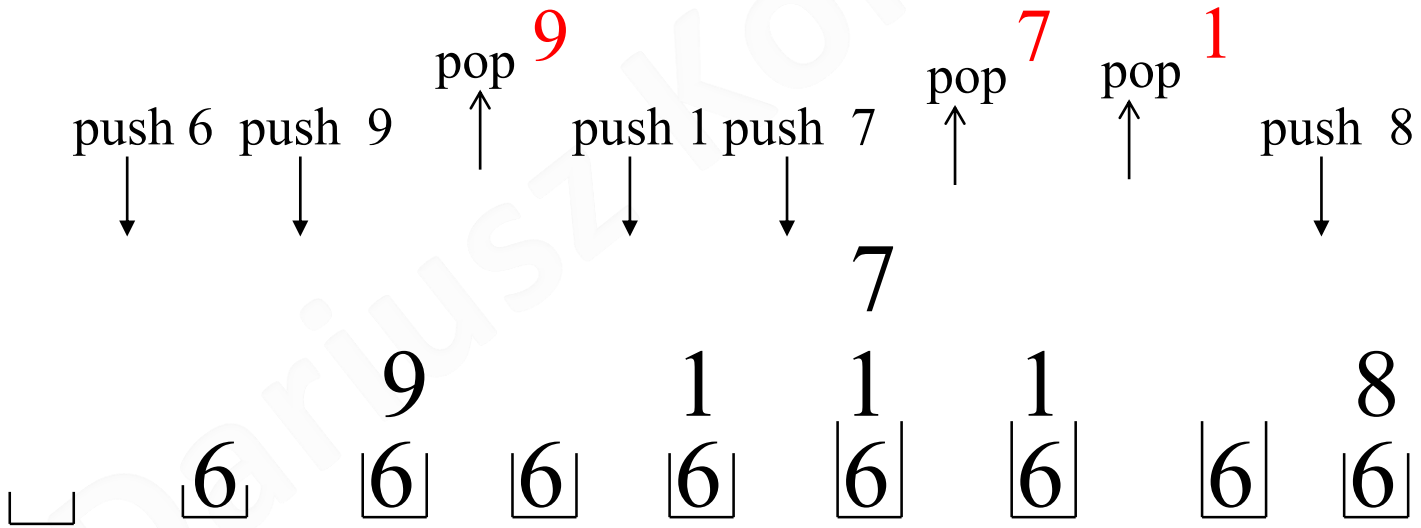
Klasa `LinkedList`

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

Dariusz Konieczny

Stos

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



Ciąg operacji na stosie

Dariusz Konieczny

Interfejs `IStack<T>`

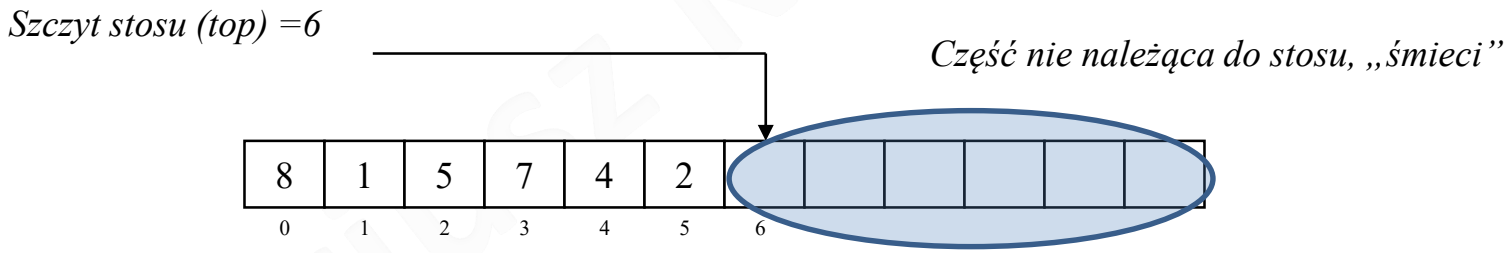
- Trzeba również stos zainicjować, ale w programowaniu obiektowym dzieje się to w konstruktorze. Po stworzeniu stos powinien być pusty.
- W bibliotekach Javy nie ma interfejsu dla stosu, jest od razu klasa stosu: `Stack<T>`.
- Stwórzmy zatem przykładowy interfejs `IStack<T>` dla stosu (z dodatkowymi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

```
public class EmptyStackException extend Exception{
}

public class FullStackException extend Exception{
}

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

Daniel Konieczny

Stos na tablicy

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



Dariusz Konieczny

Implementacja ArrayStack<T> 1/2

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

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

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

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

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

Dariusz Konieczny

Implementacja ArrayStack<T> 2/2

```
@Override
public boolean isEmpty() {
  return topIndex==0;}

@Override
public boolean isFull() {
  return topIndex==array.length;}

@Override
public T pop() throws EmptyStackException {
  if(isEmpty())
    throw new EmptyStackException();
  return array[--topIndex];}

@Override
public void push(T elem) throws FullStackException {
  if(isFull())
    throw new FullStackException();
  array[topIndex++]=elem;}

@Override
public int size() {
  return topIndex;}

@Override
public T top() throws EmptyStackException {
  if(isEmpty())
    throw new EmptyStackException();
  return array[topIndex-1];}
```

Zapis `@Override`
nie jest obowiązkowy, ale może ustrzec nas od błędów, na przykład:

```
@Override
public boolean Empty(){
…
```

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

Dariusz Kor

Złożoność operacji na stosie

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

Dariusz Konieczny

Stos za pomocą L1KPzG

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

Dariusz Konieczny

Złożoność operacji na stosie

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

Dariusz Konieczny

Stosy nieklasyczne

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

Dariusz Wardecki

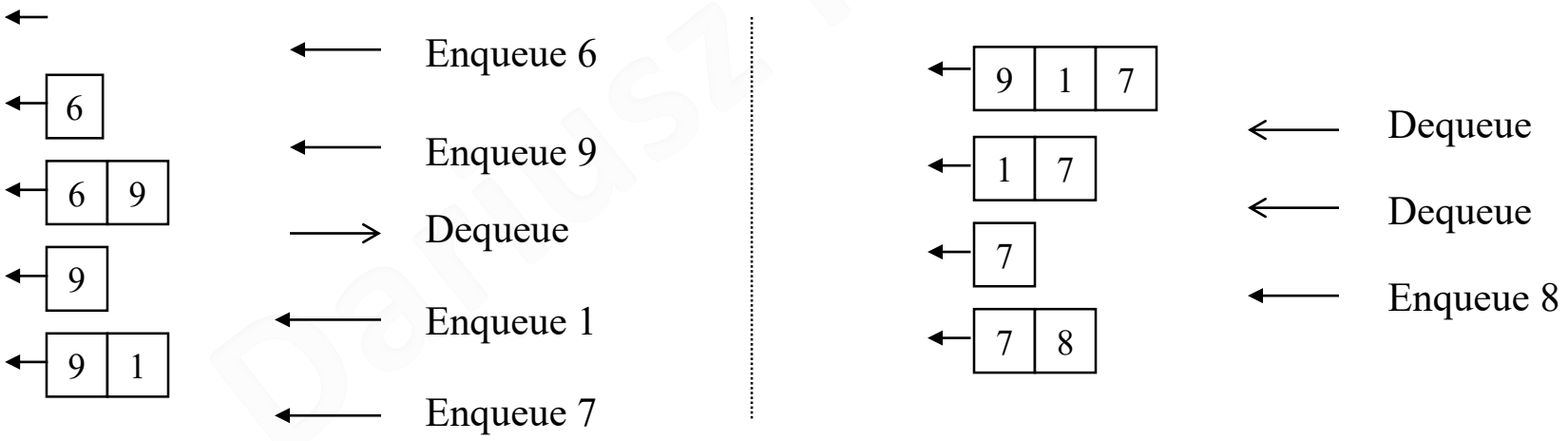
Zastosowanie stosu

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

Darmowe konie.com

Kolejka

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

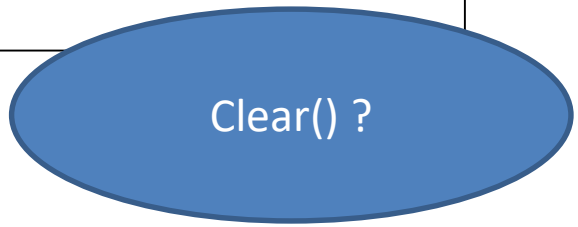


Dariusz Konieczny

Interfejs `IQueue<T>`

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

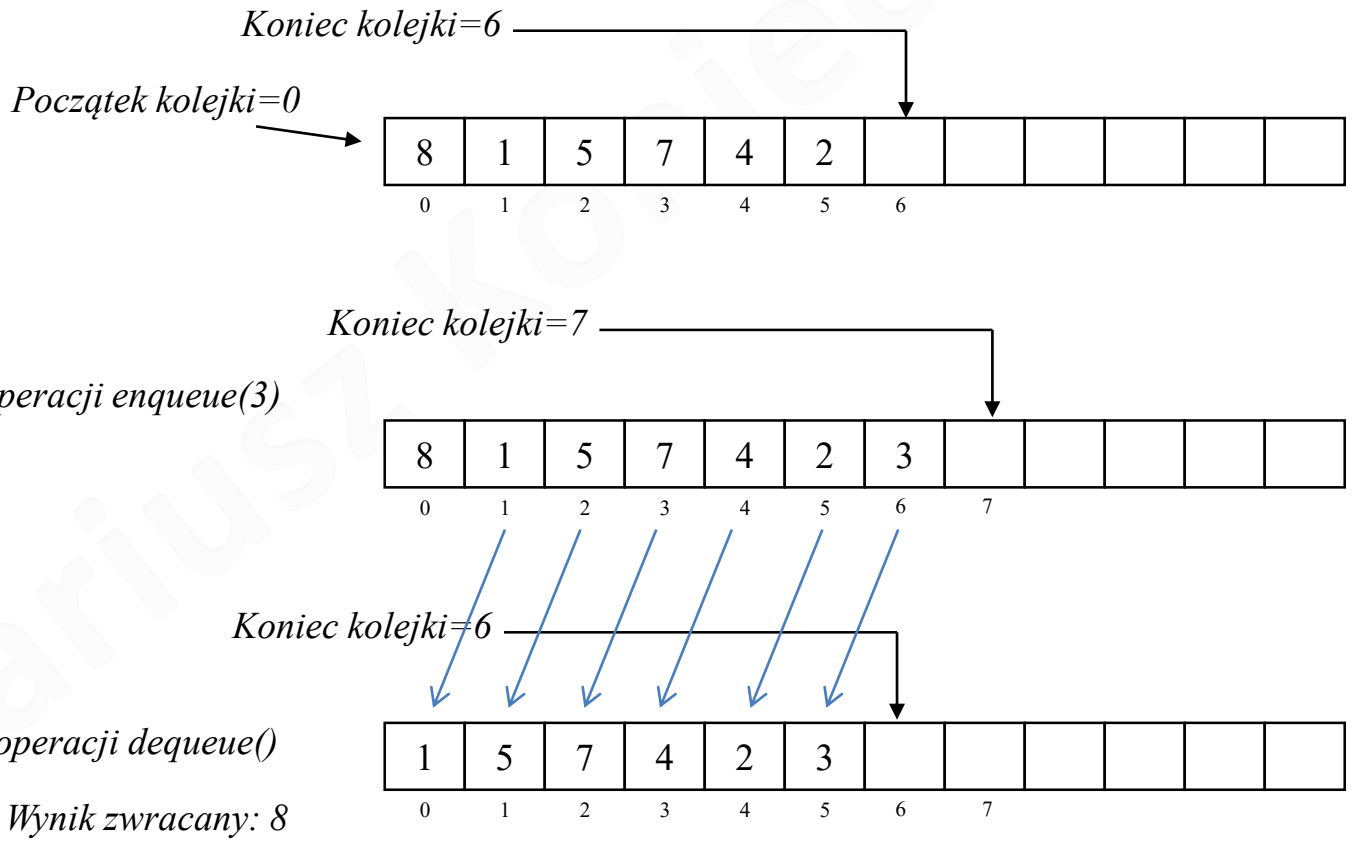
```
public class EmptyQueueException extend Exception{
}

public class FullQueueException extend Exception{
}

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



Dariusz Konieczny

Kolejka za pomocą tablicy

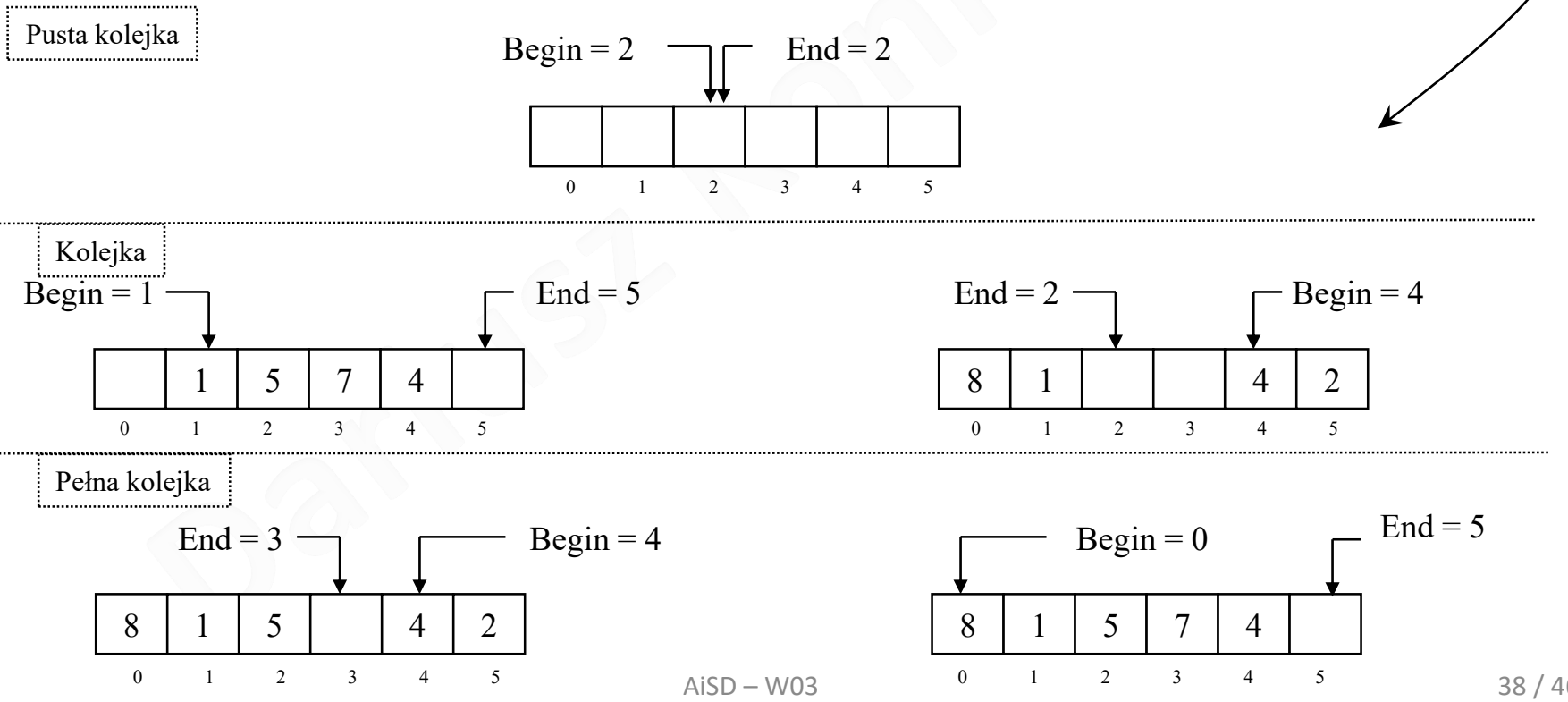
- Nieefektywne rozwiązanie:
  - zerowy indeks to zawsze początek kolejki
  - pamięta się tylko długość kolejki (jak dla stosu).
  - Przesuwanie wszystkich elementów o jedną pozycję w lewo podczas pobierania elementu z kolejki



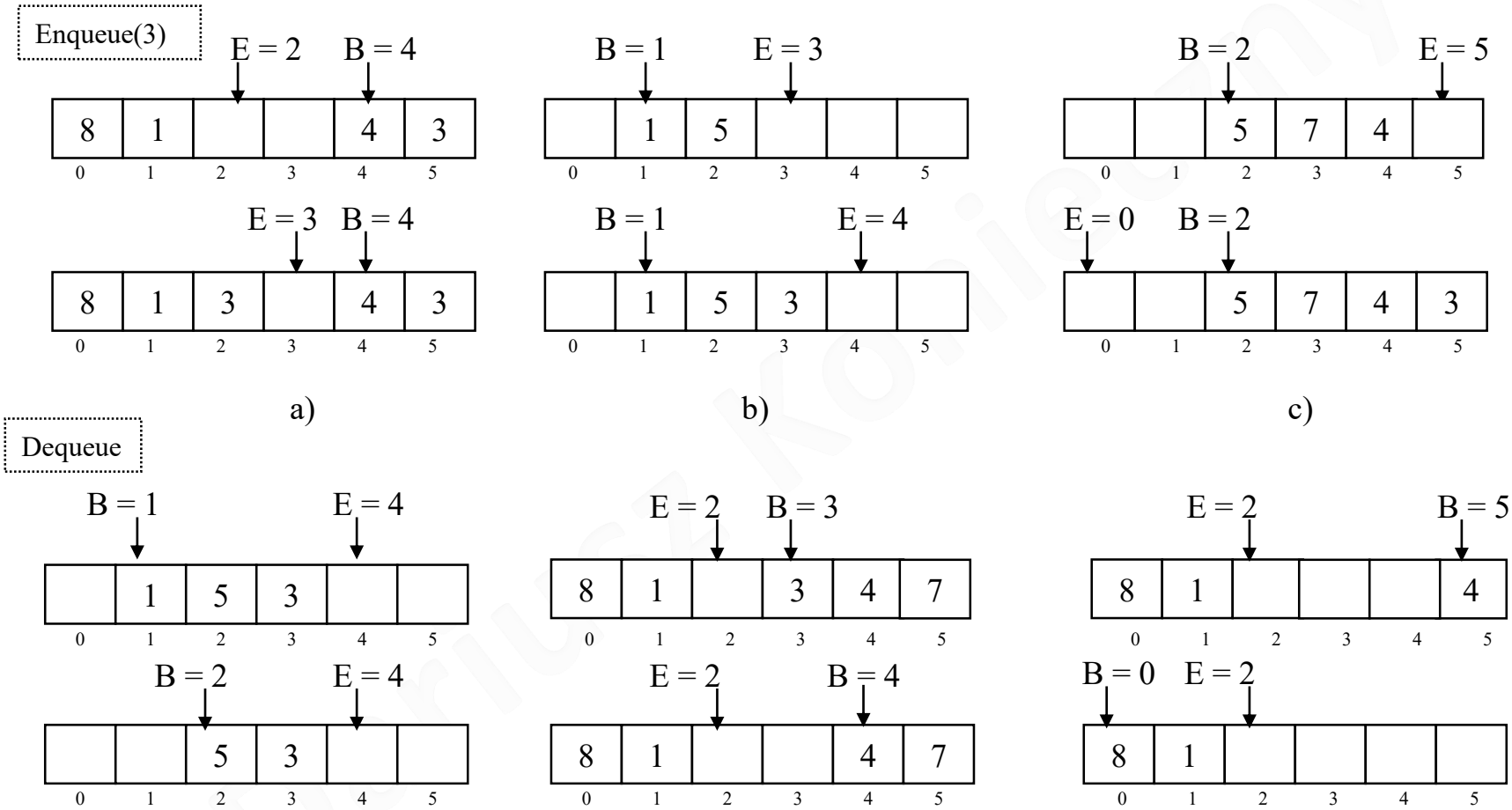
Dariusz Konieczny

Efektywna implementacja kolejki na tablicy

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



Operacje enqueue i dequeue



Bruce Romney

Kolejka – implementacja 1/2

```
public class ArrayQueue<T> implements IQueue<T> {

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

  @SuppressWarnings("unchecked")
  public ArrayQueue(int size) {
    array=(T[])new Object[size+1];
  }

  public ArrayQueue() {
    this(DEFAULT_CAPACITY);
  }


  @Override
  public boolean isEmpty() {
    return beginIndex==endIndex;
  }

  @Override
  public boolean isFull() {
    return beginIndex==(endIndex+1)%array.length;
  }
```

O(n)

O(1)

O(1)

Dariusz Konieczny

Kolejka – implementacja 2/2

```
  @Override
  public T dequeue() throws EmptyQueueException {
    if(isEmpty())
      throw new EmptyQueueException();
    T retValue=array[beginIndex++];
    beginIndex%=array.length;
    return retValue;
  }

  @Override
  public void enqueue(T elem) throws FullQueueException {
    if(isFull())
      throw new FullQueueException();
    array[endIndex++]=elem;
    endIndex%=array.length;
  }

  @Override
  public int size() {
    return (endIndex+array.length-beginIndex) % array.length;
  }

  @Override
  public T first() throws EmptyQueueException {
    if(isEmpty())
      throw new EmptyQueueException();
    return array[beginIndex];
  }
}
```

O(1)

O(1)

O(1)

O(1)

Dariusz Konieczny

Kolejka za pomocą L2KCzS

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

Dariusz Konieczny

Złożoność operacji na kolejce

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

Dariusz Konieczny

JUnit

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

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

Demonstracja działania kolejki za pomocą Junit-ów

Zastosowanie kolejki

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

- „Kolejka” to domyślnie „kolejka FIFO”. Inne rodzaje kolejek poznamy na kolejnych wykładach

Dariusz Konieczny

Zadania na ćwiczenia/laboratorium

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

- *TODO: Problemy z serwera zadań z collegiate programming.*