Algorytmy i struktury danych – W02

Teoria złożoności cz. 2/4,

Listy, listy wiązane, iterator dla list

Zawartość

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

Notacje asymptotyczne

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

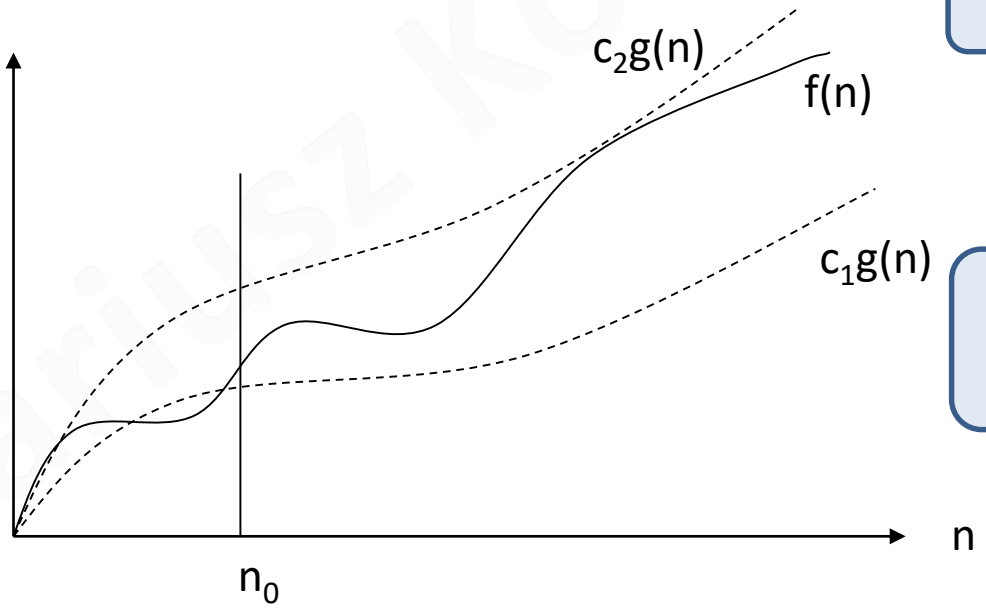
Notacja asymptotyczna – Duże Theta

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

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

Mówimy, że $f(n)$ **jest rzędu** $g(n)$

Notacja Θ wyznacza relację równoważności funkcji



Mówimy o dokładnym oszacowaniu funkcji

Duże theta - przykład

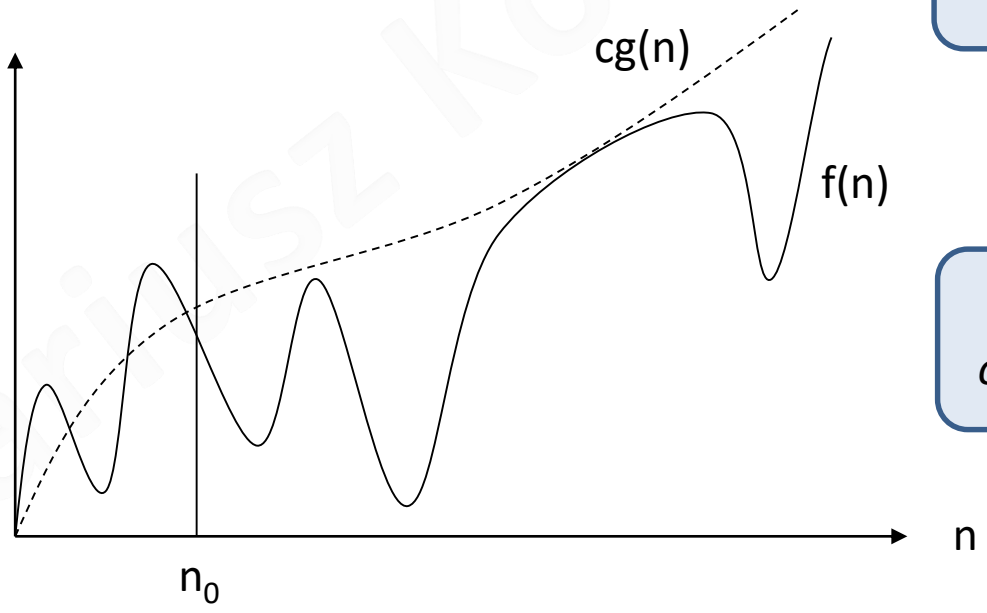
- Niech $f(n)=n^2+100n+1000$
- Zatem $f(n)= \Theta(n^2)$ *, ponieważ:*
  - *dla $n_0=1000, c_1=1, c_2=10$ oraz dla każdego $n>=n_0$:*
    - $n^2+100n+1000>n^2$
    - $n^2+100n+1000<10n^2$

      $(1.000.000+100.000+1000<10*1.000.000)$

Notacja asymptotyczna – Duże O

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

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



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

Mówimy o górnym ograniczeniu funkcji

Duże O - przykład

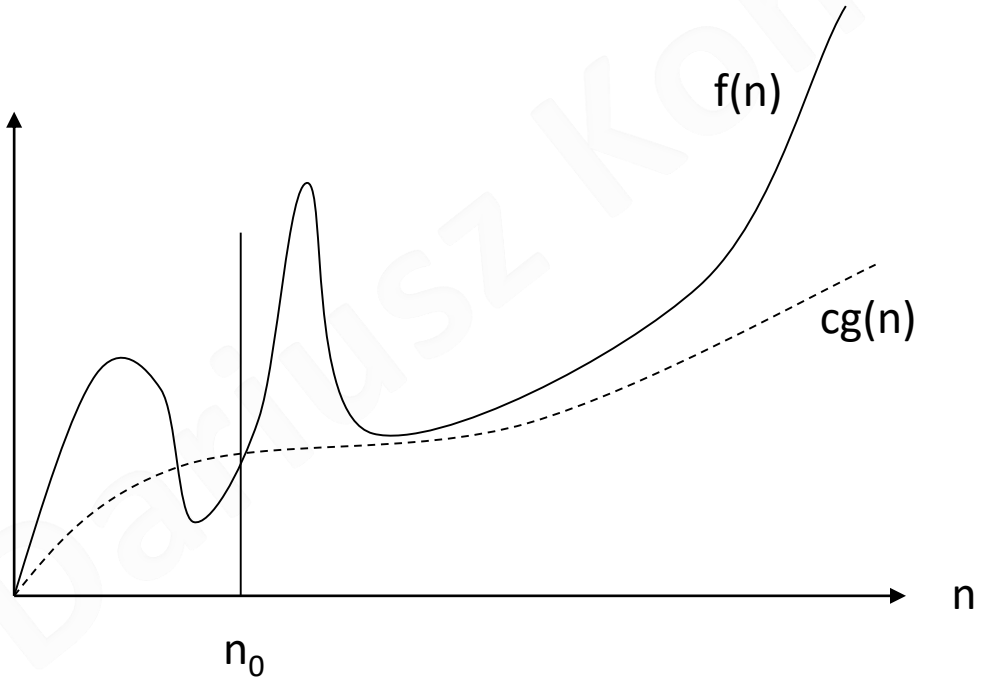
- Niech f(n)=n |sin(n)|
- Zatem f(n)= $O(n)$, ponieważ:
  - dla n_0=1,c=1, dla każdego n_0>=n zachodzi:
    - |sin(n)|<=1
    - n|sin(n)|<=n

- Również f(n)=O(n^2), jednak O(n) jest wolniej rosnącą funkcją. Notacja O(...) w teorii złożoności algorytmów jest używana do określenia ograniczenia z góry złożoności, stąd podaje się (wybiera) funkcję jak najwolniej rosnącą.

Notacja asymptotyczna – Duże omega

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

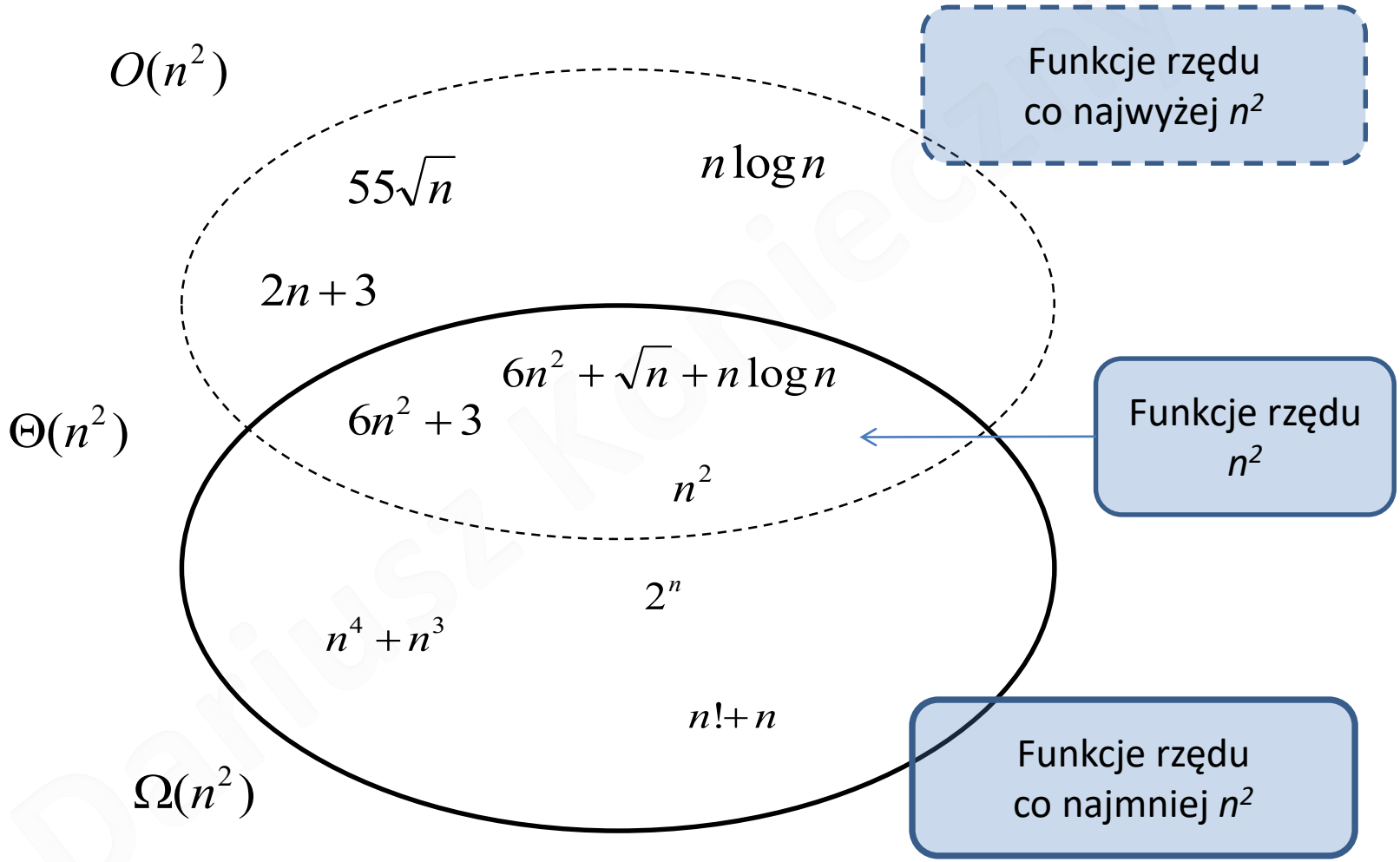
$$f(n) \geq c \cdot g(n)$$



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

Mówimy o dolnym ograniczeniu funkcji

Zależności - przykład



Przykłady liczenia złożoności

- W poniższych przykładach zamiast O(…) można by również używać notacji Θ(…)

aisd.example.ComplexityExample

```
public static int example1(int n){
  int sum=0;
  for(int i=0;i<n;i++)
    for(int j=0;j<n/2;j++)
      sum+=i+j;
  return sum;
}

public static int example2(int n){
  int sum=0;
  for(int i=0;i<2*n;i++)
    sum+=i;
  for(int j=2*n;j>0;j--)
    sum+=j;
  return sum;
}

public static int example3(int n){
  int sum=0;
  for(int i=0;i<n;i++)
    sum+=example1(i);
  return sum;
}
```

O(1)
O(1)
O(n)
$O(n^2)$
$O(n^2)$
O(1)
O(n)
O(n)
O(1)
O(n)
O(1)
$O(n^2)$
O(1)
$O(n^3)$

Złożoność problemu/algorytmu

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

Lista

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

aisd.list.IList

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

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

ListIterator<T>

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

java.util.ListIterator

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

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

Iterator dla list

- Na tym wykładzie przedstawione będzie tylko szkielet jak wyglądają operacje dla tego iteratora **w przypadku poprawnej** sekwencji operacji.
- W przypadku niepoprawnej sekwencji (np. dwa razy `remove()` bez wykonania pomiędzy `next()` lub `previous()`) iterator może zachowywać się niepoprawnie (wg dokumentacji powinien rzucać właściwym wyjątkiem)
- Uzupełnienie kodu iteratora, tak aby działał w każdym wypadku poprawnie jest ciekawym zadaniem do wykonania samodzielnie, jednak od strony algorytmiki nie dodaje istotnej wiedzy.

- Innym wyzwaniem jest używanie dwóch lub więcej iteratorów jednocześnie lub wykonywanie w czasie używania jednego iteratora metod listy modyfikującej jej strukturę (dodawanie/usuwanie), np. iterator przesuwamy na środek listy i usuwamy listę przez metodę `clear()`. Jak powinien zachować się iterator po wywołaniu `next()`?
  - Implementacja biblioteczna generuje w takich przypadkach (i wielu innych) wyjątek `IllegalStateException`.
- Obowiązuje ogólna zasada: jeśli jakiś obiekt zmodyfikował kolekcję, to wszystkie inne iteratory ulegają unieważnieniu (nie można już z nich korzystać).

Dokumentacja w kodzie Javy

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

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

aisd.util.ArrayList.InnerListIterator

iterator stoi "**pomiędzy**" elementami i tak trzeba go zaimplementować. Zaimplentowane zostaną tylko operacje *niemodyfikujące* strukturę

Press 'F2' for focus

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

Klasa `AbstractList<T>` 1/2

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

aisd.list.AbstractList

```
package aisd.util;

import java.util.Iterator;

public abstract class AbstractList<E> implements IList<E> {

  @Override
  public String toString() {
    StringBuffer buffer = new StringBuffer();
    buffer.append('[');
    if (!isEmpty()) {
      for (E item:this)
        buffer.append(item).append(", ");
      buffer.setLength(buffer.length() - 2);
    }
    buffer.append(']');
    return buffer.toString();
  }
  // ^ - bitowa różnica symetryczna
  @Override
  public int hashCode() {
    int hashCode = 0;
    for (E item:this)
      hashCode ^= item.hashCode();
    return hashCode;
  }
```

Klasa `AbstractList<T>` 2/2

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

Implementacja listy na tablicy

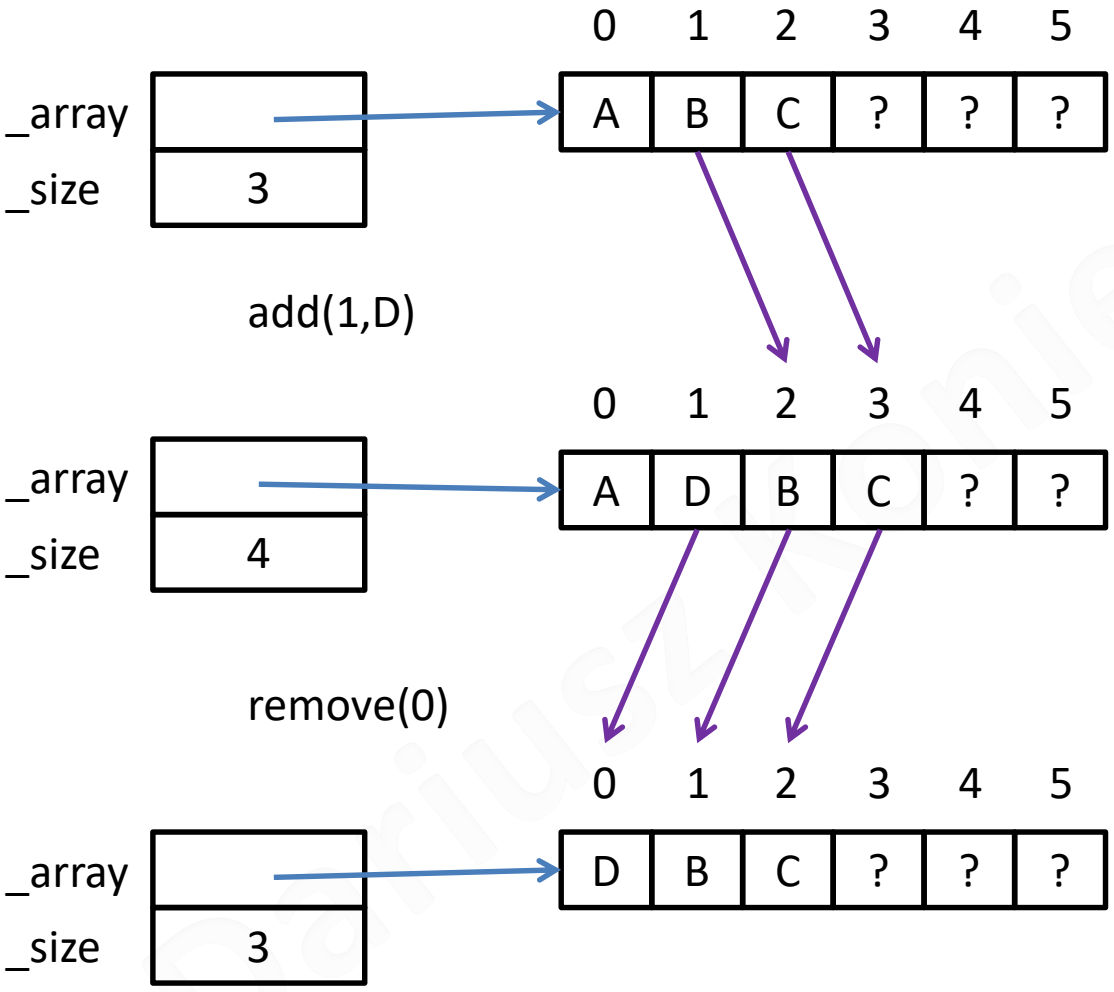
- Lista, od strony użytkownika, ma najczęściej nieograniczoną pojemność.
- Można ją zaimplementować za pomocą zwykłej tablicy, która się „rozszerza”, gdy brakuje miejsca na nowy element.
- „Rozszerzanie” polega na stworzeniu nowej, większej tablicy i przepisaniu do niej danych z poprzedniej tablicy.
- Implementacja listy będzie za pomocą klasy generycznej.

- W Javie nie można stworzyć tablicy elementów generycznych (dokładnie: tablicy elementów typu będącego parametrem klasy generycznej)
- Zamiast tego tworzona będzie tablica typu `Object` i rzutowana na tablicę elementów generycznych. Jest to dozwolone, ale kompilator ostrzega o możliwości niedopasowania typów, stąd przed funkcją z takim rzutowaniem należy dopisać annotację:
  `//@SuppressWarnings("unchecked")`
- Np.

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

aisd.list.ArrayList

Lista na tablicy - operacje

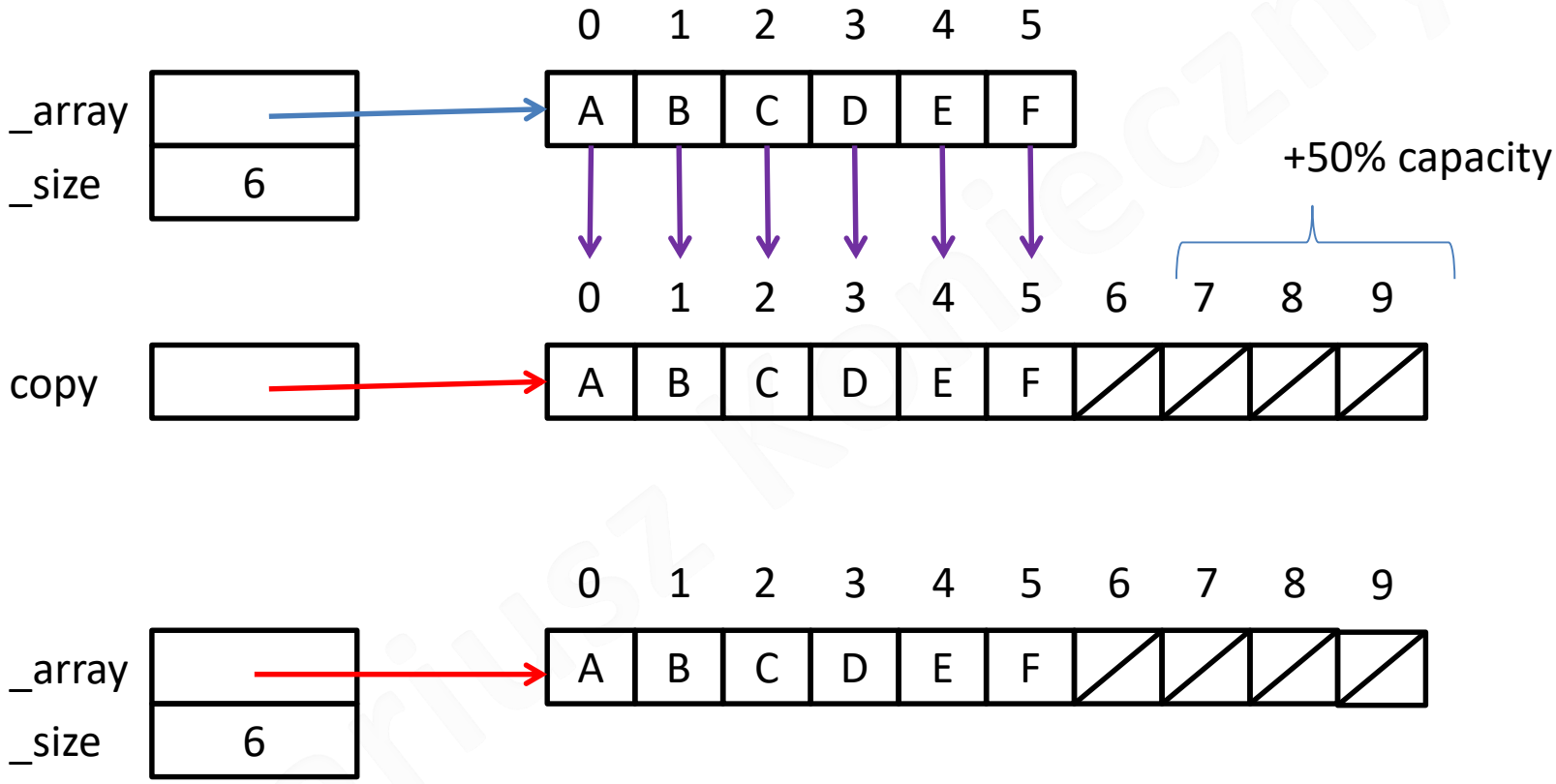


ArrayList 1/4

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

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

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

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

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

ensureCapacity

```
ensureCapacity(7)
```



ArrayList 2/4

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

ArrayList 3/4

```
@Override
public int indexOf(E value) {
  int i =0;
  while(i < _size && !value.equals(_array[i]))    ++i;
  return i<_size ? i : -1;}
@Override
public boolean contains(E value) {
  return indexOf(value) != -1;}
@Override
public E get(int index) {
  checkOutOfBounds(index);
  return _array[index];}
@Override
public E set(int index, E element) {
  checkOutOfBounds(index);
  E retValue=_array[index];
  _array[index]=element;
  return retValue;}
@Override
public E remove(int index) {
  checkOutOfBounds(index);
  E retValue = _array[index];
  int copyFrom = index + 1;
  if (copyFrom < _size) System.arraycopy(_array, copyFrom, _array, index, _size - copyFrom);
  --_size;
  return retValue;}
@Override
public boolean remove(E value) {
  int pos=0;
  while(pos<_size && !_array[pos].equals(value))
    pos++;
  if(pos<_size){
    remove(pos);
    return true;}
return false;}
```

ArrayList.InnerIterator 1/1

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

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

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

ArrayList.InnerListIterator 1/1

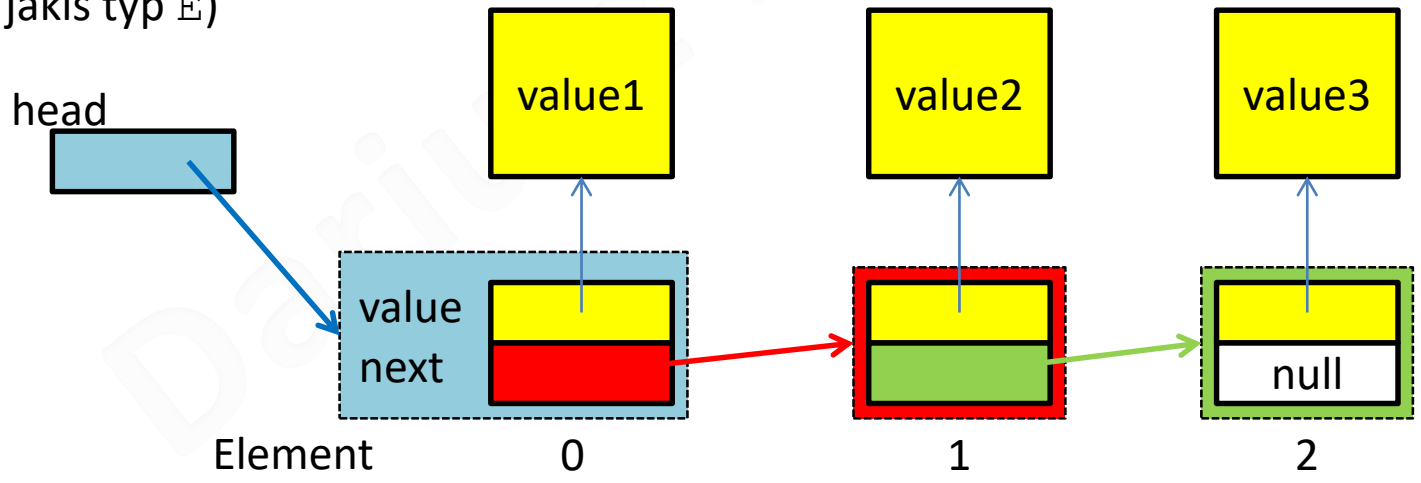
```
private class InnerListIterator implements ListIterator<E>{
  int _pos=0;
  @Override
  public void add(E Value) {
    throw new UnsupportedOperationException();}
  @Override
  public boolean hasNext() {
    return _pos<_size;}
  @Override
  public boolean hasPrevious() {
    return _pos>=0;}
  @Override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

`ArrayList, InnerListIterator` - złożoności

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

Listy za pomocą list wiązanych

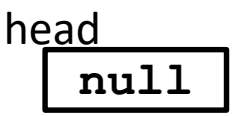
- Gdyby była potrzeba usunięcia np. co drugiego elementu na liście lub inne podobne działania (za pomocą właśnie zaimplementowanych metod), złożoność takiej operacji byłaby kwadratowa $O(n^2)$.
- Nawet gdyby zaimplementować wszystkie operacje iteratora dla list.
- Dodatkowo w `ArrayList` zdarza się,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



OneWayLinkedListWithHead.Element

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

    public E getValue() {
      return value;}

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

    public Element getNext() {
      return next;}

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

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

  Element head=null;

  public OneWayLinkedListWithHead(){}

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

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

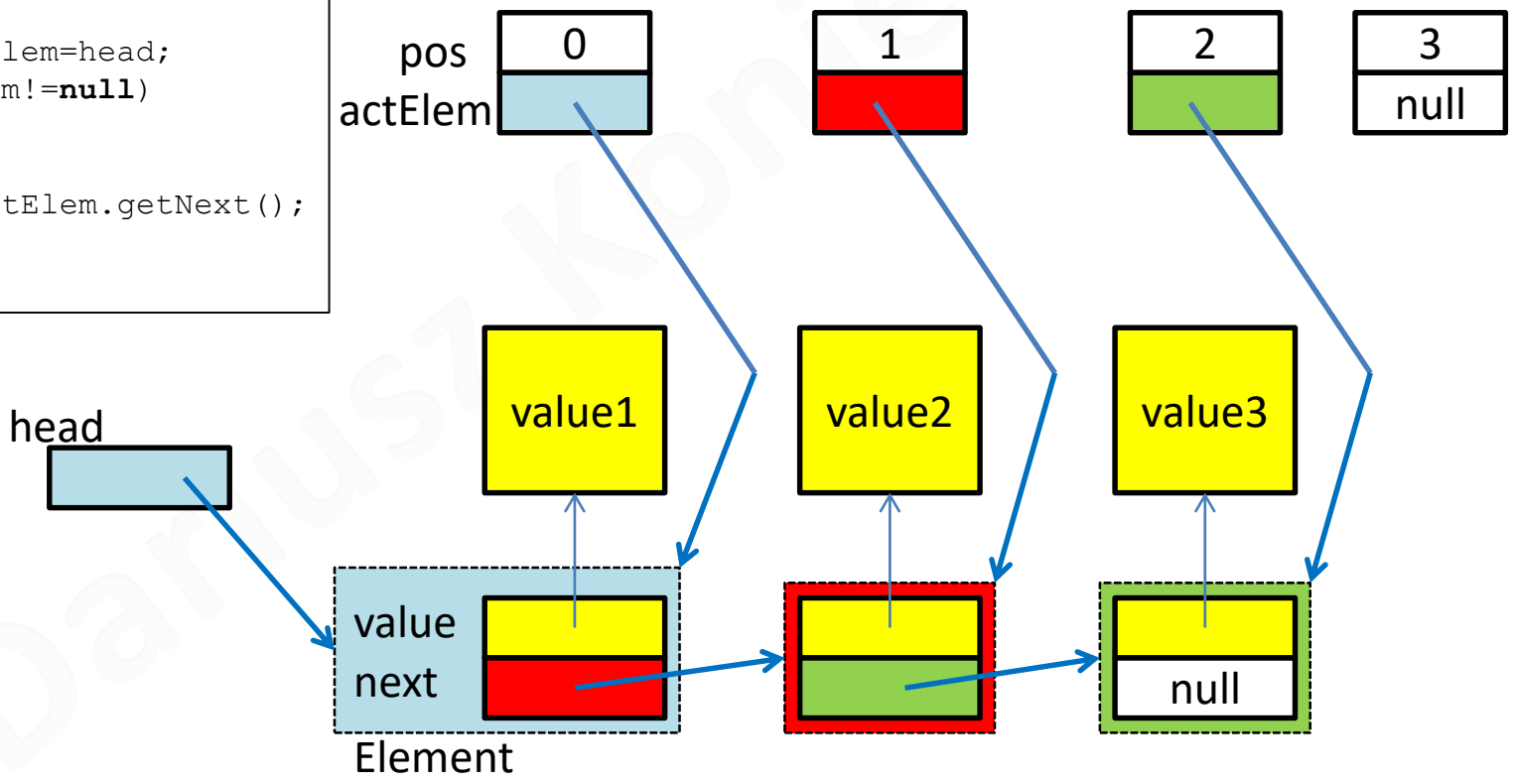
aisd.list. OneWayLinkedListWithHead



Metoda `size()`, przechodzenie po liście

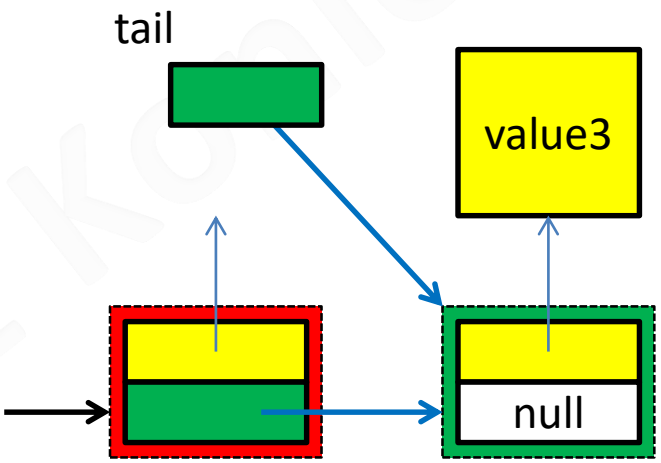
- Przechodzenie po liście powiązanej do przodu za pomocą pomocniczej referencji:
  `actElem=actElem.getNext();`

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



Metody `getElement()`, `add()`

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

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



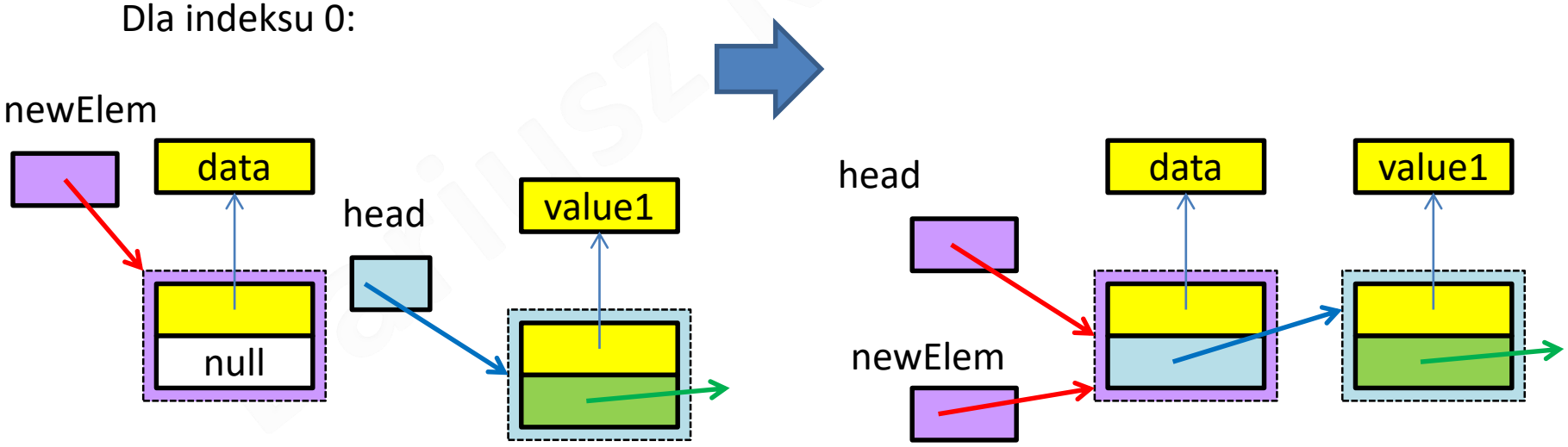
- Dodając element na koniec tej listy trzeba osobno rozpatrzeć dodawanie do pustej listy i osobno do niepustej.

Metoda `add(index, data)`

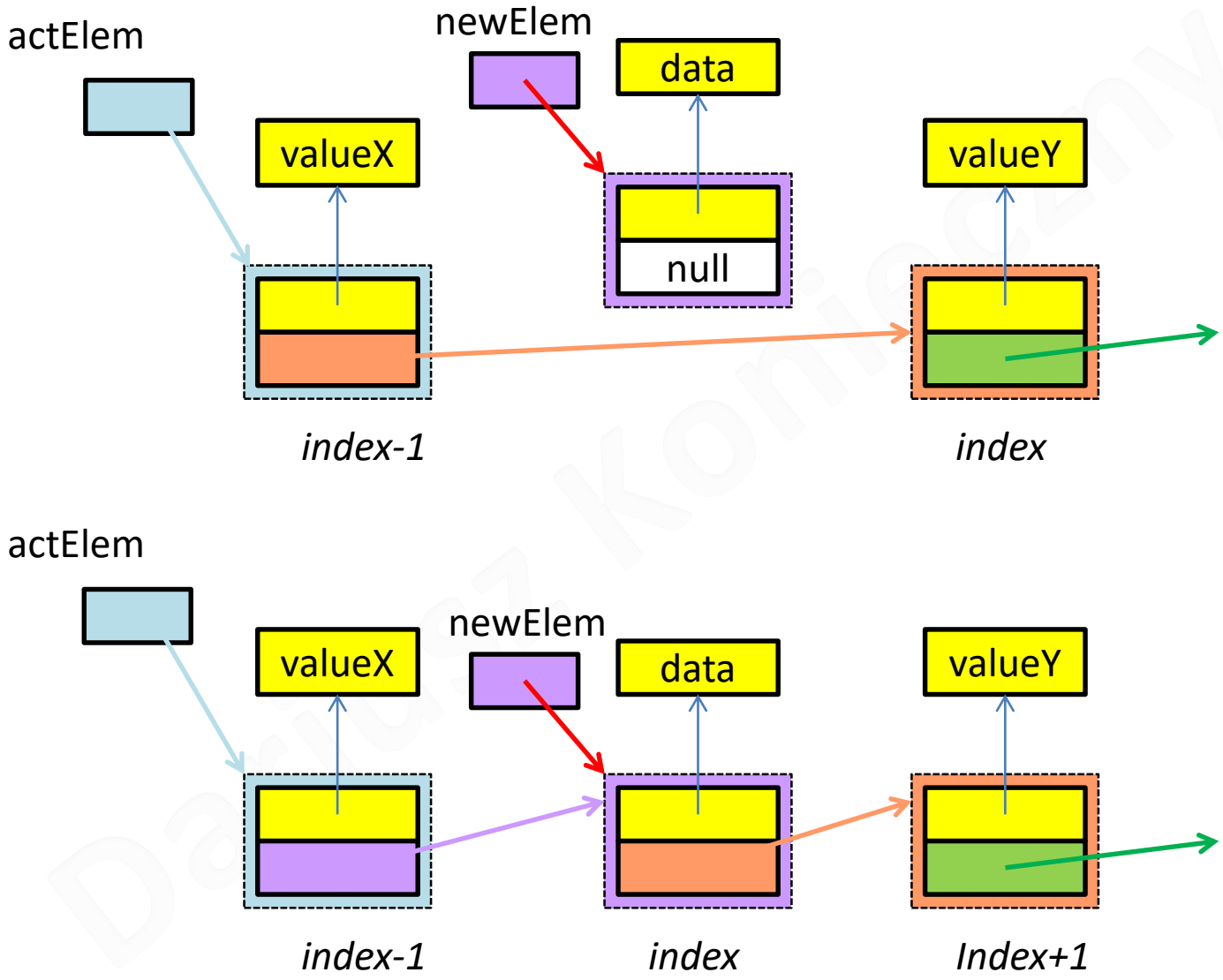
```
@Override
public boolean add(int index, E data) {
    if(index<0) throw new IndexOutOfBoundsException();
    Element newElem=new Element(data);
    if(index==0)
    {
      newElem.setNext(head);
      head=newElem;
      return true;
    }
  Element actElem=getElement(index-1);
  newElem.setNext(actElem.getNext());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

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

Dla indeksu 0:



`add(index,data)` w środku listy



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

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

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

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

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

remove() x 2

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

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

Iterator

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

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

Lista L1KPzG - złożoność

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