Data Structures and Algorithms – W02

Complexity part 2,
Lists, LinkedLists, Iterator for list

Contents

- Computation complexity part 2:
  - Asymptotic Notation Θ,O, Ω
- Lists:
  - Description
  - Interface `IList<T>`
  - Interface `ListIterator<T>`
  - Class `AbstractList<T>`
- Array implementation of a list: class `ArrayList<T>`
- Linked lists:
  - One-way, straight, with a head, without sentinel – L1SwH, class `OneWayLinkedListWithHead<T>`

Asymptotic Notation

- Reminder:
  - n : input size
  - $f(n)$: the function telling how many calculation steps should be performed for data of length n
- Function $f(n)$ can:
  - be in the form of a complicated mathematical formula
  - depend not only on n but on specific data
  - be non-deterministic
- Instead of giving an exact formula, the function **order** is more important. For this purpose, **asymptotic notations** are used.

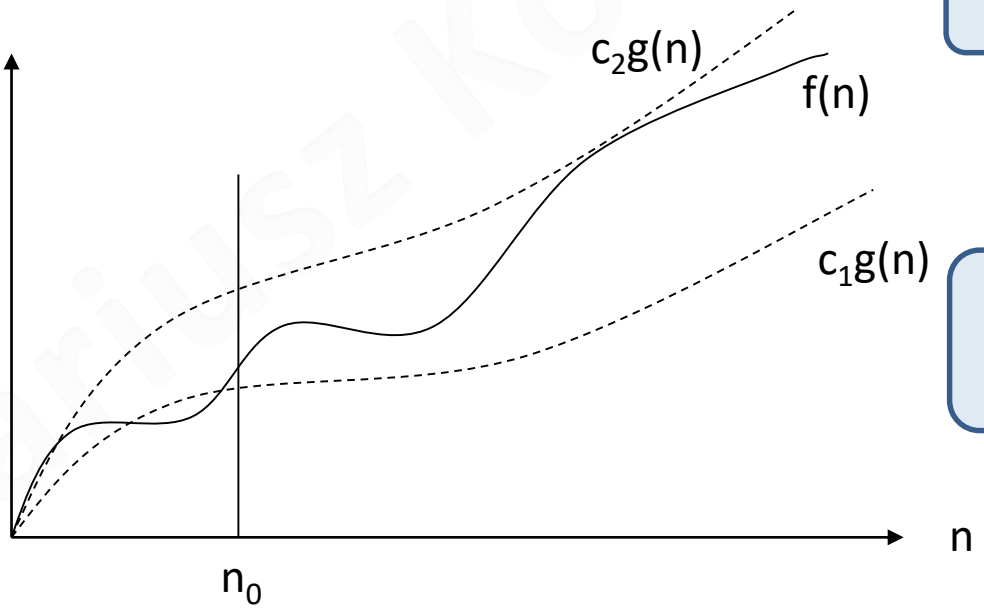
Asymptotic Notation – Big Theta

For a given function $g(n)$, we denote by $\Theta(g(n))$ the set of functions $f(n)$ having the property that there exists positive constants c_1,c_2, and n_0 such that for all n>=n_0,

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

$f(x)$ **has order** $g(x)$

Θ determines an *equivalence relation*



We're talking about the exact estimate of the function

Big Theta - example

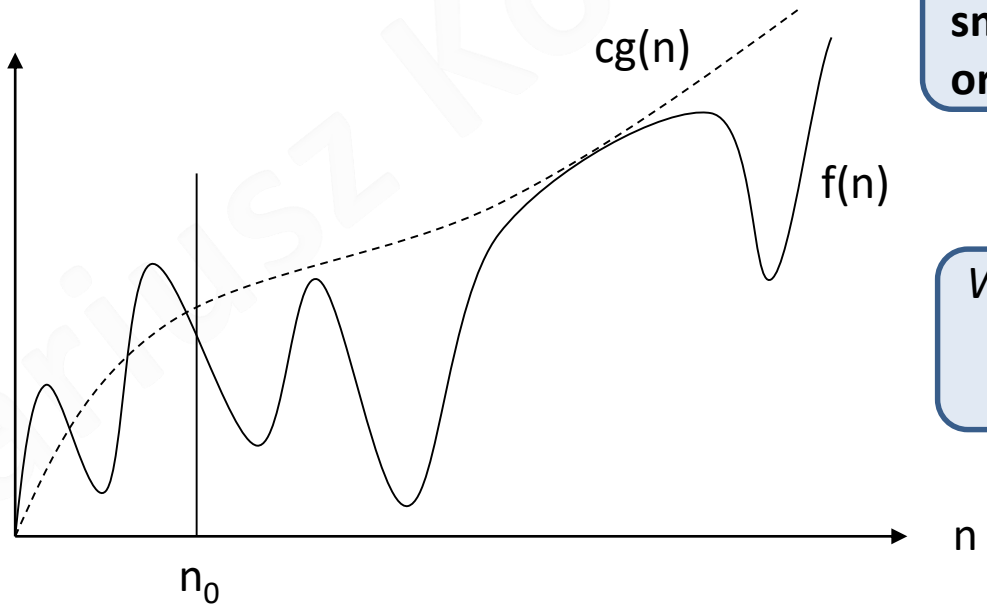
- f(n)=n^2+100n+1000
- f(n)= $\Theta(n^2)$, because:
  - for n_0=1000,c_1=1, c_2=10 we have for n>=n_0:
    - n^2+100n+1000>n^2
    - n^2+100n+1000<10n^2

      (1.000.000+100.000+1000<10*1.000.000)

Asymptotic Notation – Big Oh

For a given function *g(n)*, we denote by *O(g(n))* the set of functions *f(n)* having the property that there exists positive constants *c* and n_0 such that for all $n >= n_0$

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



We say that *f* has a **smaller or equal order** than *g*

We are talking about **asymptotic upper bounds**

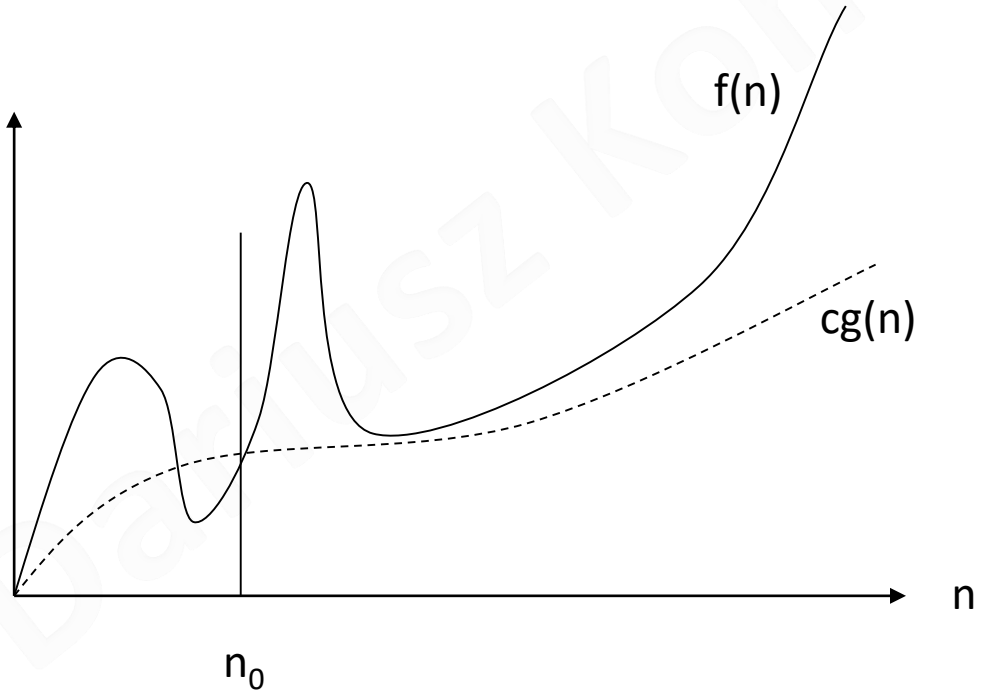
Big Oh - example

- Let f(n)=n |sin(n)|
- So f(n)= $O(n)$, because:
  - for n_0=1,c=1, we have for n_0>=n:
    - |sin(n)|<=1
    - n|sin(n)|<=n
- Of course also f(n)=O(n^2) however, O (n) is a slower growing function. Notation O (...) in the theory of algorithm complexity is used to determine the constraint of complexity, hence (chooses) the function **as slowly growing as possible**.

Asymptotic Notation – Big Omega

For a given function *g(n)*, we denote by $\Omega(g(n))$ the set of functions *f(n)* having the property that there exists positive constants *c* and n_0 such that for all $n >= n_0$,

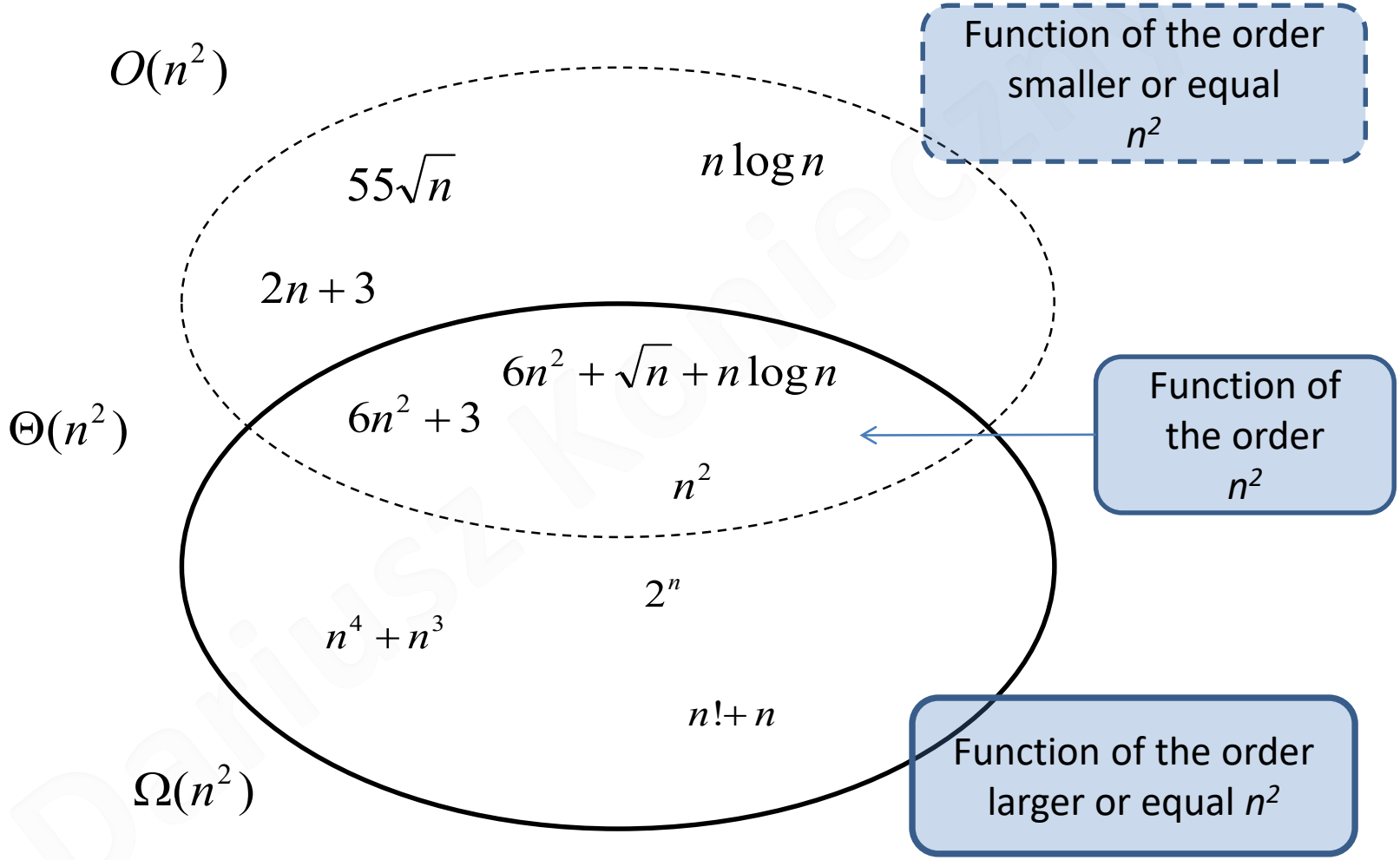
$$c \cdot g(n) \leq f(n)$$



We say that *f* has a **larger or equal order** than *g*

We are talking about **asymptotic lower bounds**

Dependences - examples

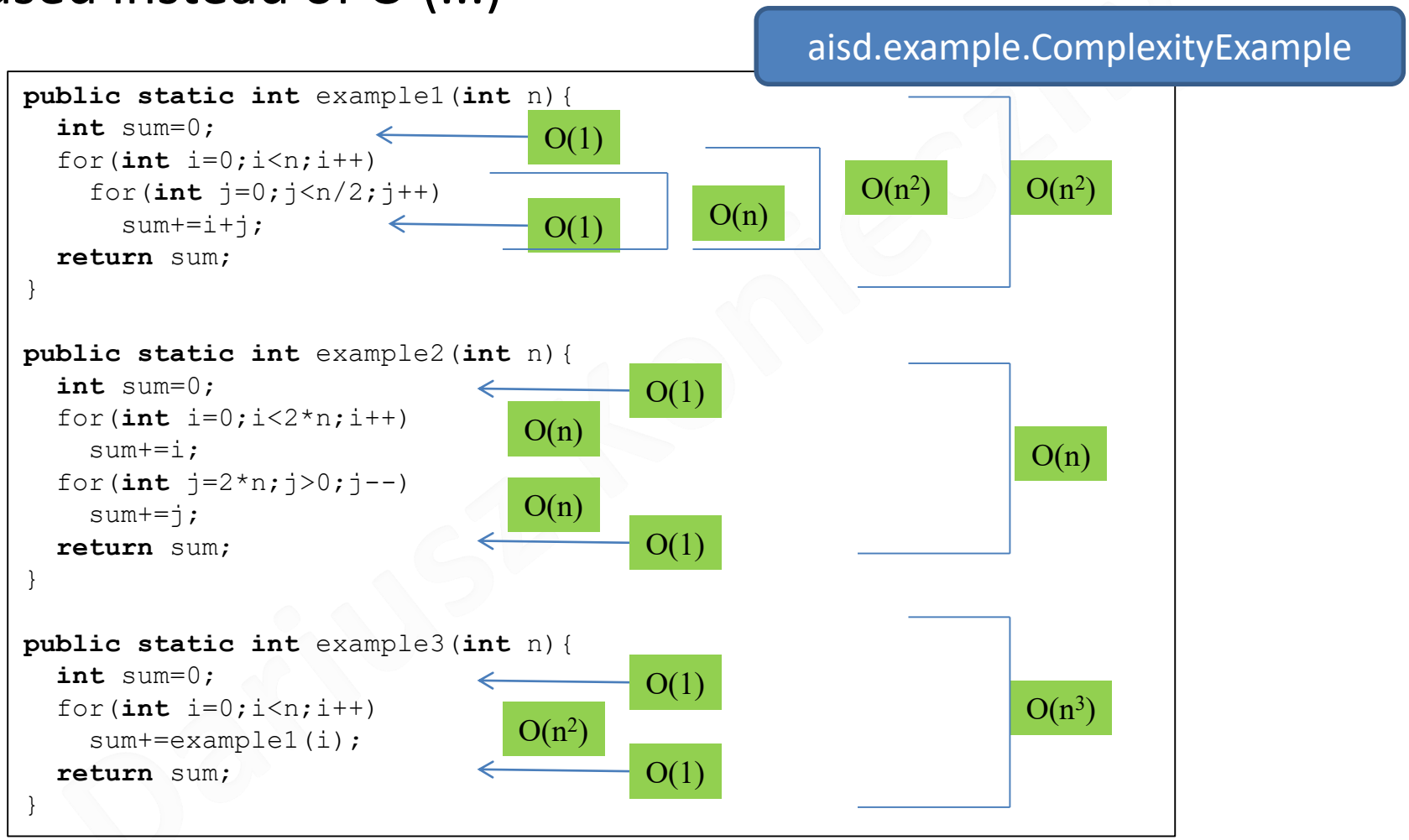


Examples of calculating complexity

- In the examples below, the notation Θ (...) could be used instead of O (...)

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;
}
```



Complexity of problem/algorithm

- **time complexity** *of a problem* is the number of *steps* that it takes to solve an instance of the problem as a function of the *size of the input*, using the *most efficient* algorithm
  - Sometimes the algorithm that is always known is not the best, hence the problem is often presented with the proof of what the minimum number of steps **must** be done
- **space complexity** *of a problem* is a related concept, that measures the *amount of space*, or memory required by the algorithm
- **time complexity** and **space complexity** can be considered for a *chosen algorithm*.

The list

- The list is **a linear structure** in which you can perform multiple operations anywhere in the structure.
- In addition to the regular iterator, the list should also include an iterator for lists, which allows you to move in two directions along the list.
- The interface for lists `List <E>` is available in the `java.lang` package. In order not to complicate the code within this lecture, a similar, simplified `IList <E>` interface will be created.

aisd.list.IList

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

public interface IList<E> extends Iterable<E> {
  boolean add(E e); // add an element on the end of the list
  void add(int index, E e); // add element on position index
  void clear(); // remove all elements
  boolean contains(E e); // check if the contains input element (equals())
  E get(int index); // get an element from position index
  E set(int index, E e); // set an element on position index
  int indexOf(E element); // find position of an element (equals())
  boolean isEmpty(); // check ig list is empty
  Iterator<E> iterator(); // return an iterator
  ListIterator<E> listIterator(); // return an listIterator
  E remove(int index); // remove element from position index
  boolean remove(E e); // remove an input element (equals())
  int size();
}
```

ListIterator<T>

- Interface to line collections, which can be moved in two directions.
- The operations that can be performed using this interface are given below.
- If an operation is difficult/ineffective for a given collection, instead of implementing it, you can throw the exception `UnsupportedOperationException`. They are marked as optional in the documentation.

java.util.ListIterator

```
public class ListIterator<T> implements Iterator<T> {
  void add(E e); // add e at current position, after iterator
  boolean hasNext();
  boolean hasPrevious();// reverse to hasNext
  E next();
  int nextIndex(); // index of element which will be returnet by next()
  E previous();
  int previousIndex(); similar like nextIndex()
  void remove(); // remove last element returned by next() or previous()
  void set(E e); set a new value on position last returned
}
```

- The abovementioned approach does not unnecessarily generate the whole family of iterators (eg iterators without adding / remove operations or one of them, etc.)
- Of course, creating your own collection and returning the iterator for it should be documented how the operations in the iterator behave.
- Iterators present in Java libraries are prepared for concurrent programming, so their implementation is more complex than what is presented in this course.

Iterator for list

- In this lecture, only the skeleton will be presented as the operations for this iterator in the case of the correct sequence of operations
- In the case of an incorrect sequence (eg, twice `remove()` without execution between `next ()` or `previous ()`), the iterator may behave incorrectly (according to the documentation should throw the right exception)
- Supplementing the iterator code so that it works in every case is an interesting task to do on its own, but it does not add significant knowledge from the algorithmic side.

- Another challenge is to use two or more iterators at the same time, or to use the list method to modify the structure (`add` / `delete`) while using the iterator, eg move the iterator to the middle of the list and delete the list by the `clear ()` method. How should the iterator behave after calling `next()`?
  - The library implementation generates an `IllegalStateException` exception in such cases (and many others).
- The general rule applies: if an object has modified a collection, all other iterators are invalidated (you can no longer use them).

Documentation in the Java code

- Java allows you to comment on the code so that the commentary allows technical documentation.
- Most development environments read such formatted comments while working on the code.
- Documenting comment should start with "/ **" instead of "/ *”
- The following lines start with "*", but this is not necessary
- Such a comment is written BEFORE the declaration of class, method, field.
- You can use part of HTML tags and other special tags in the text.

```
@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;
```

Class `AbstractList<T>` 1/2

- Implementation of certain methods (derived from the `Object` class) may look the same for each list, so it is worth creating an abstract class in which they will be implemented (from Java 9.0, you can do it within the interface):

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();
  }
  // ^ - bitwise exclusive OR
  @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;  }
}
```

Implementation of the list on the array

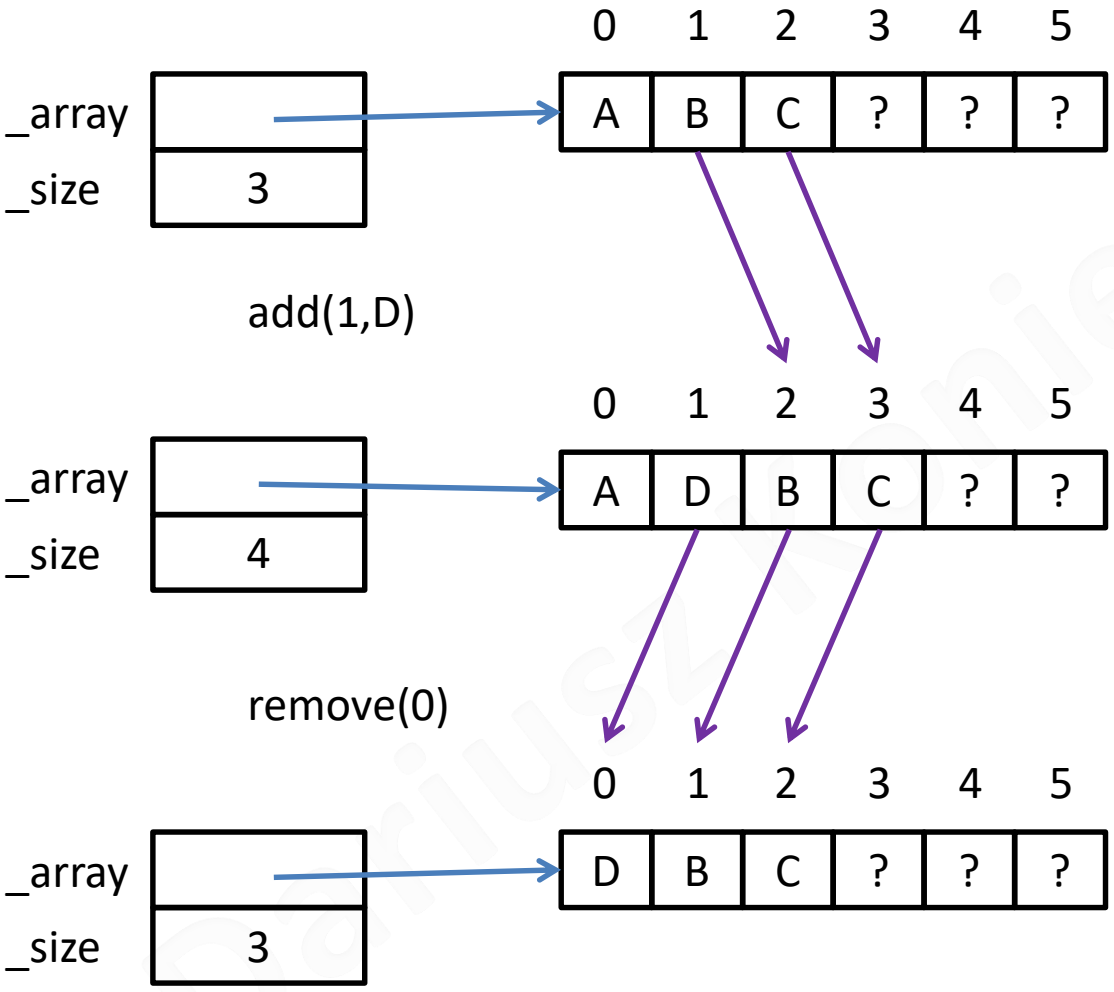
- The list, from the user's side, usually has unlimited capacity.
- It can be implemented using a regular array that "expands" when there is not enough room for a new item."Expanding" consists in creating a new, larger array and rewriting data from the previous table to it.
- The implementation of the list will be using the generic class

- In Java, you can not create an array of generic elements (exactly: an array of elements of a type being a parameter of a generic class)
- Instead, an array of `Object` type will be created and casting onto an array of generic elements. This is allowed, but the compiler warns about the possibility of mismatching types, hence an annotation should be added before the function with such casting: `//@SuppressWarnings("unchecked")`
- E.a.

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

aisd.list.ArrayList

A list on an array

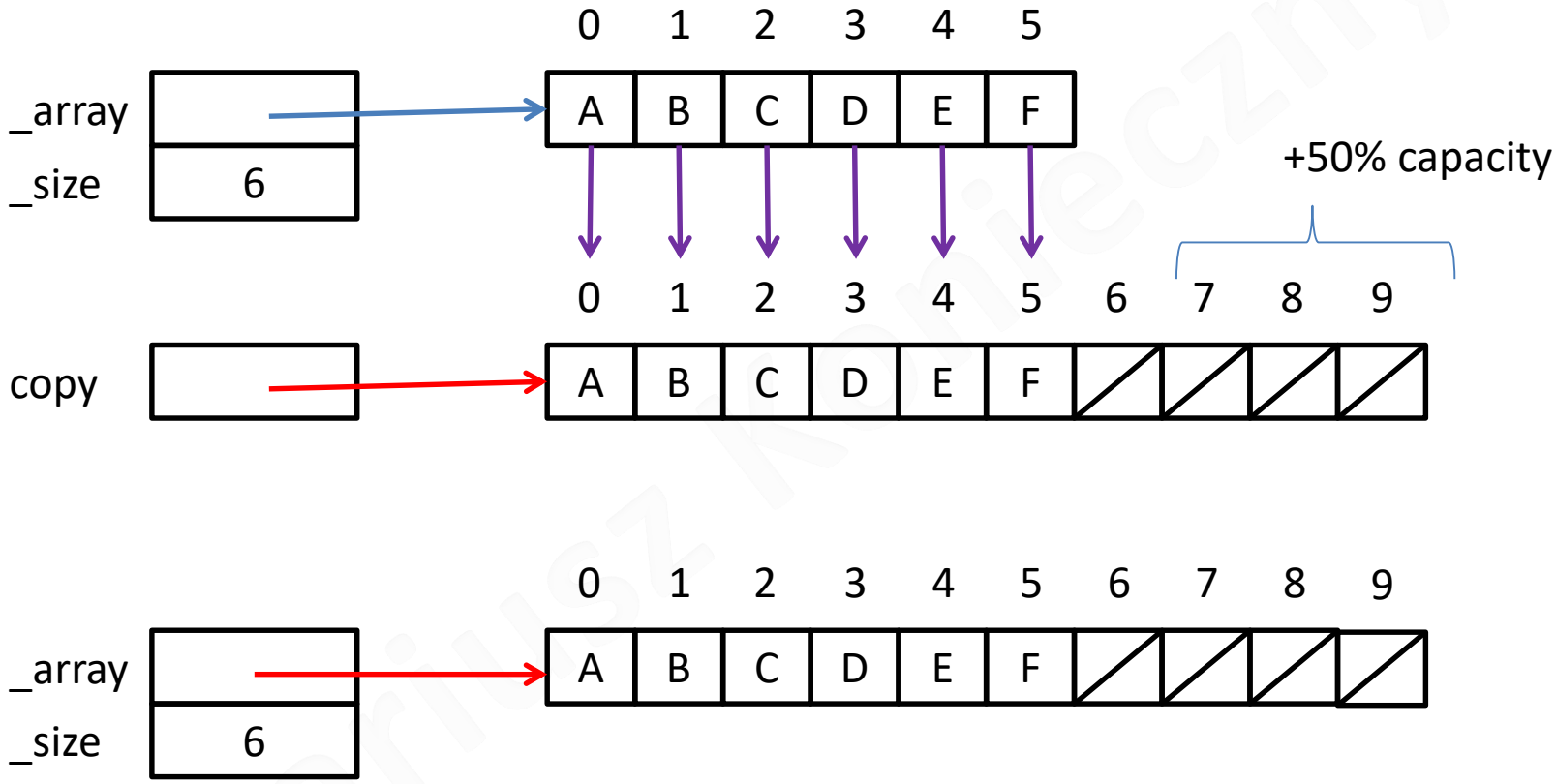


ArrayList 1/4

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

public class ArrayList<E> extends AbstractList<E> {
  /** <b> Default </b> size of an initial array. */
  private static final int DEFAULT_INITIAL_CAPACITY = 16;
  /**  <b> Initial </b> size of array. */
  private final int _initialCapacity;
  /** A reference to an array with elements */
  private E[] _array;
  /** Size of the array used for a the list.*/
  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

```
/** Extension of the array if no space in the current. */
@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;
}}
       // validation of the index
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 true;   }
```

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

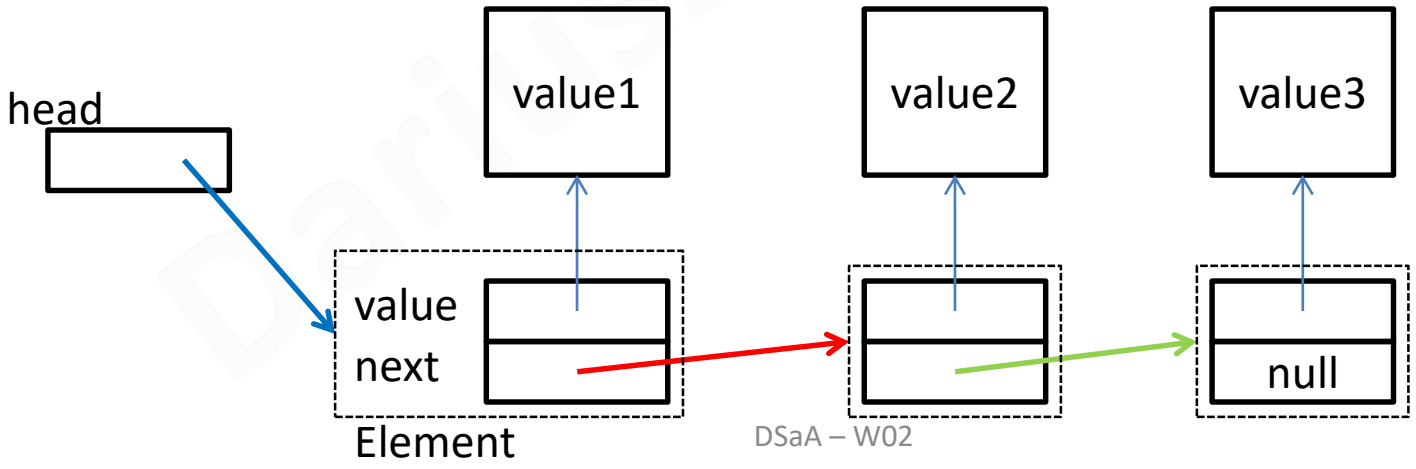
```
private class InnerListIterator implements ListIterator<E>{
  int _pos=0;
  @Override
  public void add(E Value) {
    throw new UnsupportedOperationException();}
  @Override
  public boolean hasNext() {
    return _pos<_size;}
  @Override
  public boolean hasPrevious() {
    return _pos>=0;}
  @Override
  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` - complexity

- `ArrayList` – worse-case complexity:
  - Construction – O(1)
  - `isEmpty(),size(),clear()` – O(1)
  - `set(),get()` – O(1)
  - `ensureCapacity()` – O(n)
    - `add()` on the end – O(n)
  - `add()` ith index – O(n), on the end – O(1) if without `ensureCapacity()`
  - `indexOf()` – O(n)
    - `contains()` – O(n)
  - `remove()` x 2 – O(n)
- `ArrayList` – average complexity:
  - `ensureCapacity()` – O(1)
    - `add()` on the end – O(1)
- `ArrayList.InnerListIterator` if you implement all operations – average complexity:
  - `hasNext(),next(),hasPrevious(),previous(),nextIndex(), previousIndex()` – O(1)
  - `set()` – O(1)
  - `add(),remove()` – O(n)

Lists using linked lists

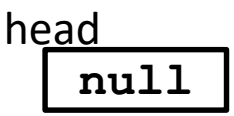
- If there was a need to remove eg every other item on the list or other similar activities (using just implemented methods), the complexity of such an operation would be square $O(n^2)$. Even if you implement all iterator operations for lists.
- Additionally, in `ArrayList`, half of the array may be unused.
- If in a given collection you will need to frequently insert / remove elements in the middle in the addition while navigating the iterator, there is a faster implementation of the list idea using the linked list of elements.
- The simplest to understand is the one-way straight list with the head (L1SwH).
  - Each element of the list has two fields: `value` and reference for the `next` element. If this item is missing, the field is **`null`**.
  - The entire list is remembered through the `head` field, which is the reference to the first element of the list.
- Because it will be a generic class, the `value` will also be a reference (for some type `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;}
```

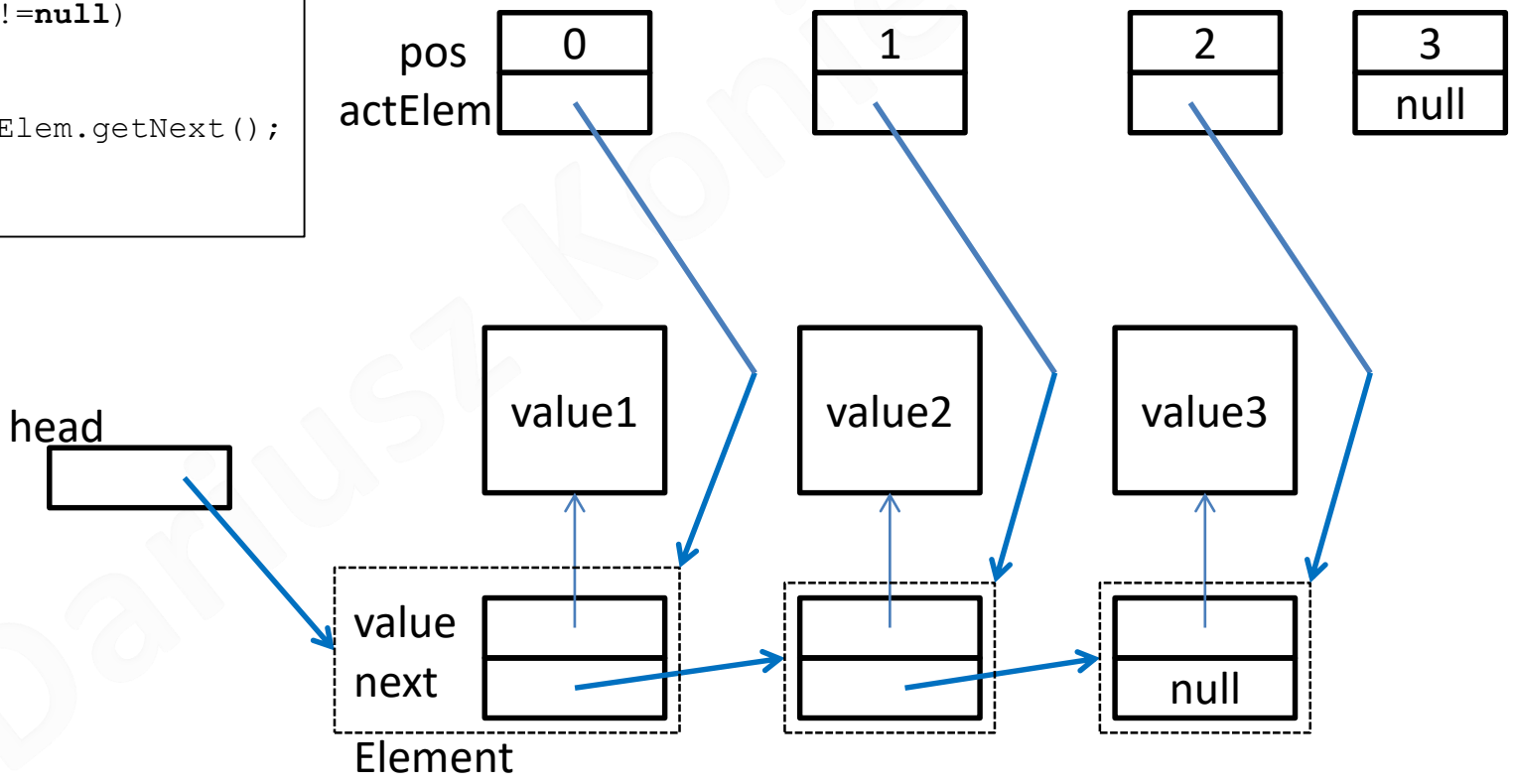
aisd.list. OneWayLinkedListWithHead



Method `size()`, moving forward

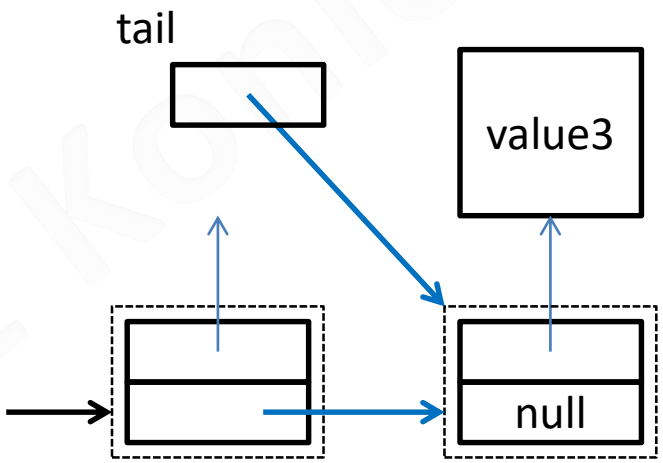
- Moving forward in alinked list using an auxiliary reference :
  `actElem=actElem.getNext();`

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



Methods `getElement()`, `add()`

```
/** Returns the reference to Element, the inner class */
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;
}
```



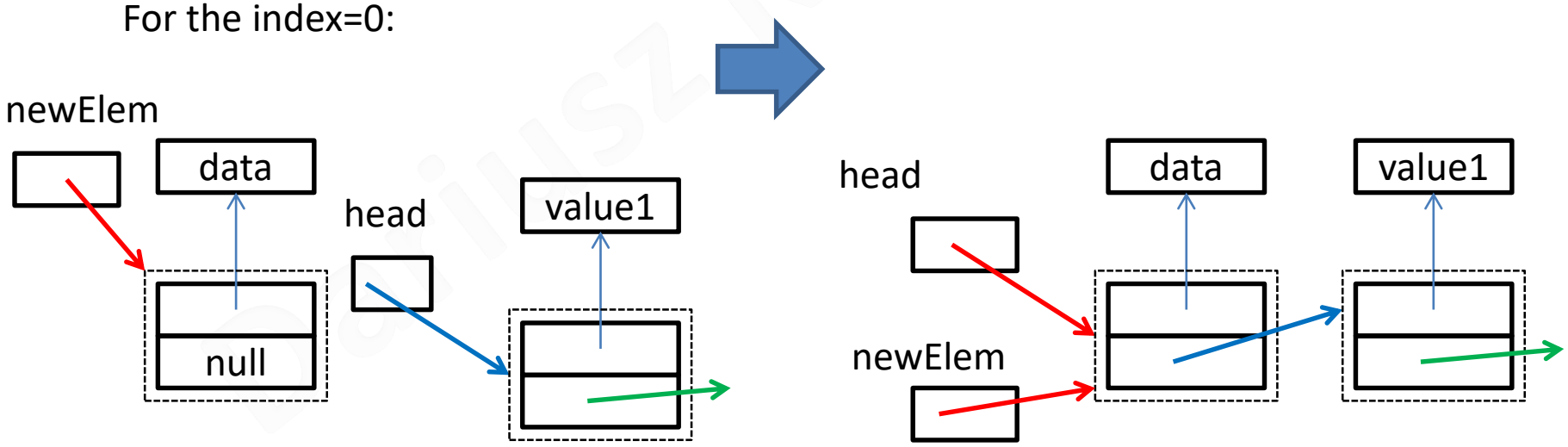
- When adding an element to the end of this list, you must separately consider adding to an empty list and separately to non-empty.

Method `add()` with an indexi

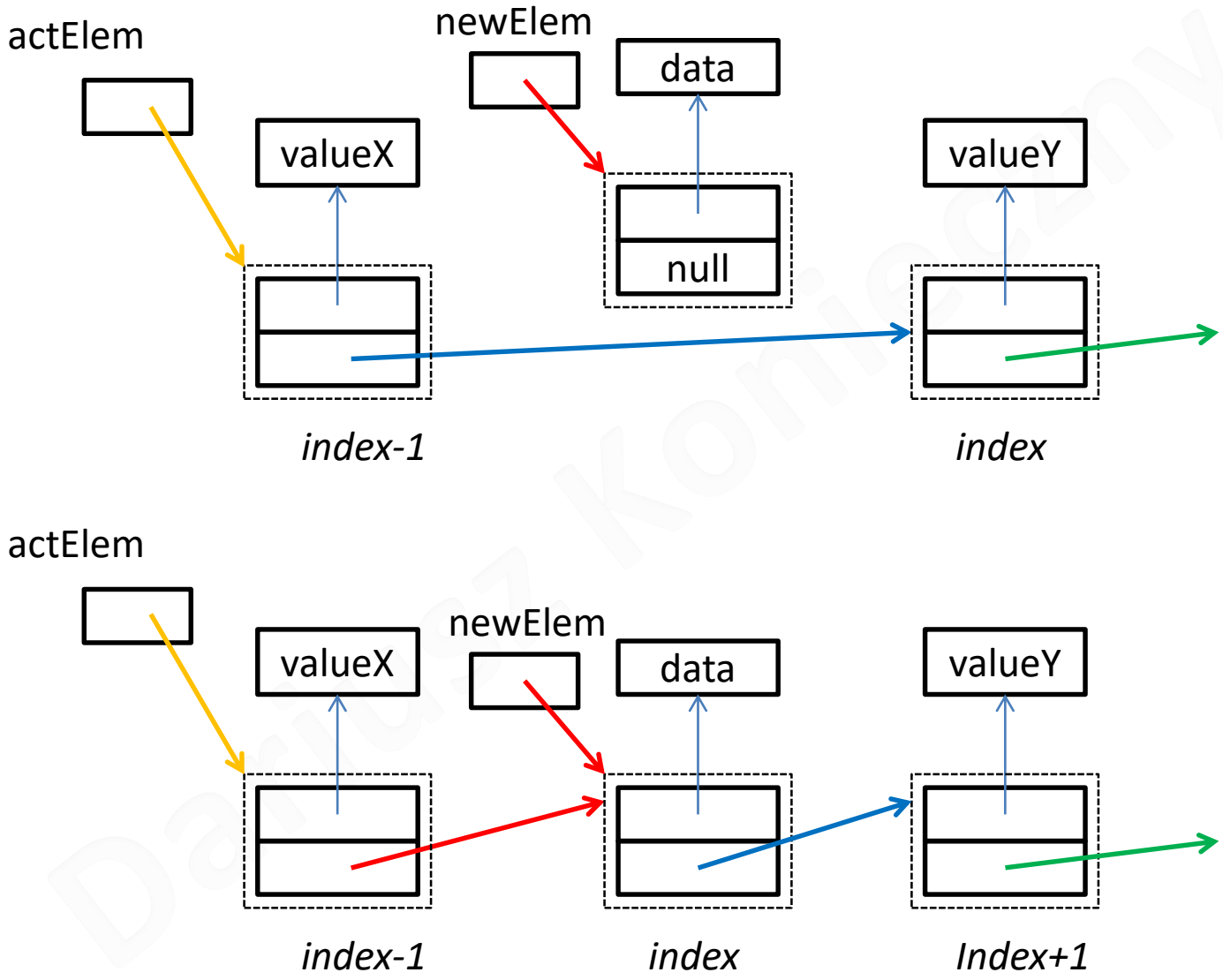
```
@Override
public boolean add(int index, E data) {
    if(index<0) throw new IndexOutOfBoundsException();
    Element newElem=new Element(data);
    if(index==0)
    {
      newElem.setNext(head);
      head=newElem;
      return true;
    }
  Element actElem=getElement(index-1);
  newElem.setNext(actElem.getNext());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

- You need to consider adding to position 0 separately
- Otherwise, you need to find the element from the (index-1) position: explanation on the next slide

For the index=0:



`add()` in the middle of the list



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

- When removing an element from the given index, just as in adding, you must stop one element earlier and make the changes inversely to the add () operation.

```
@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();
}
```

List L1SwH - complexity

- Lista L1SwH – worse-case complexity:
  - Construction – O(1)
  - `isEmpty()`,`clear()` – O(1)
  - `set()`,`get()`, `size()` – O(n)
  - `add()` on the end – O(n)
  - `add()` on begin – O(1)
  - `add()` with an index – O(n)
  - `indexOf()` – O(n)
    - `contains()` – O(n)
  - `remove()` x 2 – O(n)
  - `remove()` on the begin – O(1)
- Iterator for the L1SwH :
  - `hasNext()`,`next()` – O(1)
- The L1SwH list is quite limited in its applications, but it allows effective stack implementation