European Funds
Knowledge Education Development

Wrocław University
of Science and Technology

European Union
European Social Fund

ZPR PWr – Zintegrowany Program Rozwoju Politechniki Wrocławskiej

Data Structures and Algorithms – W03

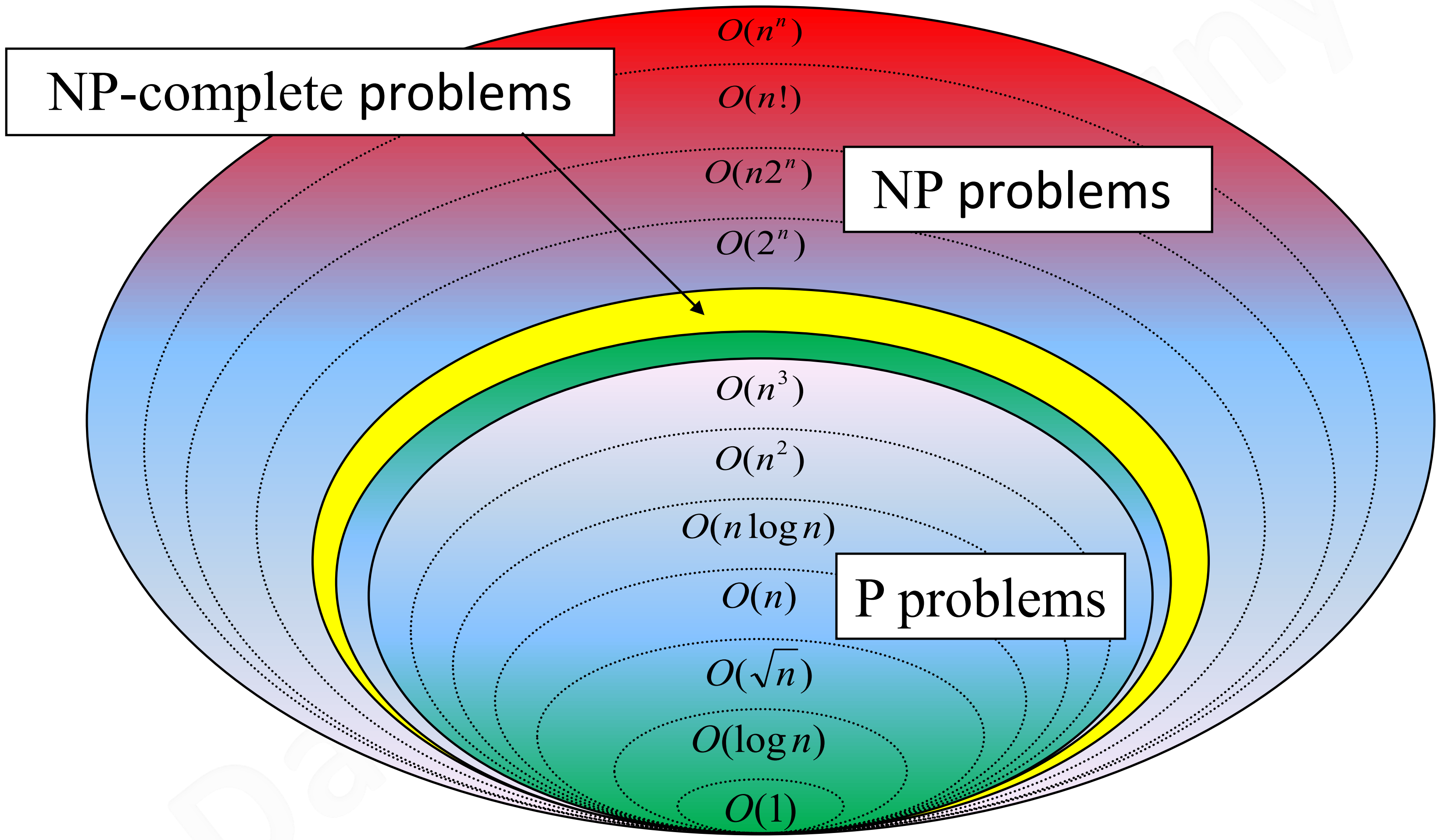
Complexity - part 3 of 4,

Two-way, circular list with a sentinel

Stack, queue

Contents

Hierarchy of orders – complexity classes



Relationships 1/2

- transitivity

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

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

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

- reflexivity

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

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

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

- symmetry

$$f(n) = \Theta(g(n)) \text{ if and only if } g(n) = \Theta(f(n))$$

- transitive symmetry

$$f(n) = O(g(n)) \text{ if and only if } g(n) = \Omega(f(n))$$

Relationships 2/2

- Others property

$$n^m = O(n^k), \qquad \text{where } m \le k$$

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

$$c \cdot O(f(n)) = O(f(n)), \qquad \text{where } c \text{ is constant}$$

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

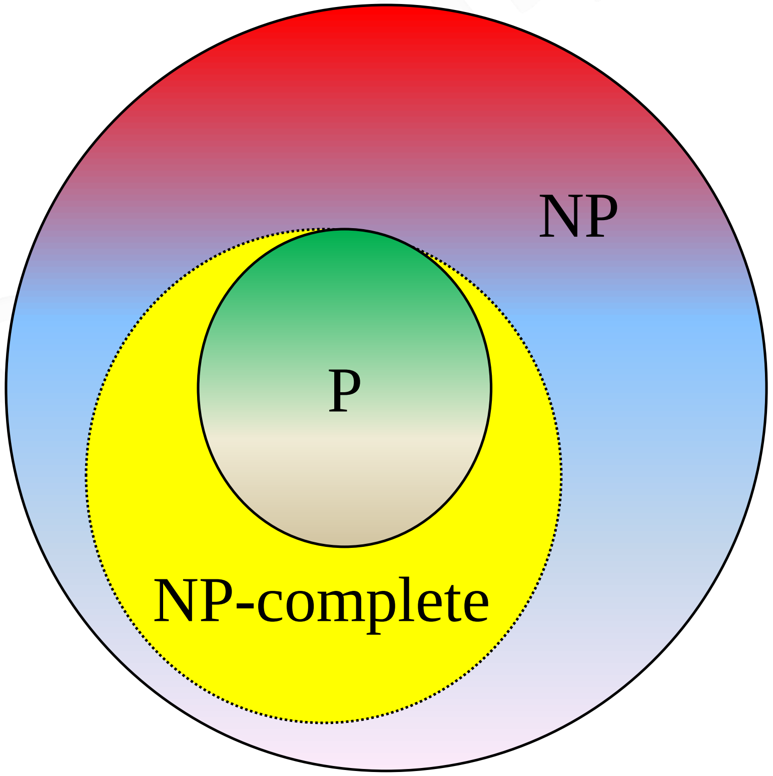
$$O(f(n)) \cdot O(g(n)) = O(f(n) \cdot g(n))$$

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

- Always remember that O(…), Θ(…), Ω(…) means the set of function and the equals sign is ambiguous:
  - It means the equality of sets when sets are equal on both sides of the equal sign
  - Indicates that the function belongs to the set when one of the side is a function

Problems classification

- P problems
  - Computation complexity is at most polynomial
    - sorting
    - matrix multiplication
    - shortest path
- NP problems
  - Computation complexity is at least exponential
    - simplex method
    - monk’s puzzle problem
- NP-complete problems
    - boolean satisfiability problem (SAT)
    - Hamilton’s cycle
    - set division



Complexity comparison 1/3

- Assumption: the computer is able to perform one billion (10^9) steps of the algorithm in one second. That is 1000 MHz steps (and not CPU clock cycles) - not currently available for home computers

Complexity function	Size 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 days	121,4 days	3,1 years	19,2 years
2^n	0,0000001s	0,001s	1s	18,3 min	13 days	36,6 years
3^n	0,000006s	3,5s	2,4 days	385 years	$2*10^7$ years	$1,3*10^{12}$ years
n!	0,003s	77 years	$8*10^{15}$ years	$2*10^{32}$ years	$9*10^{47}$ years	$2*10^{65}$ years

age of universe = $13,82*10^9$ years

Complexity comparison 2/3

- In this course, the computational complexity of the algorithms will not exceed O(n^4).
- It is worth checking what is the reduction of the exponent *n* by one, i.e. improving the algorithm by an order of magnitude (see table below).
- The API for various classes / interfaces contains information about the complexity of a given method or the expected complexity.

Complexity function	Size 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 days	31,7 years

Complexity	**Name**
O(1)	Constant
O($\log_2 n$)	Logarithmic
O(n)	Linear
O($n \log_2 n$)	Linear-Logarithmic
O(n^2)	Quadratic
O(n^3)	Cubic

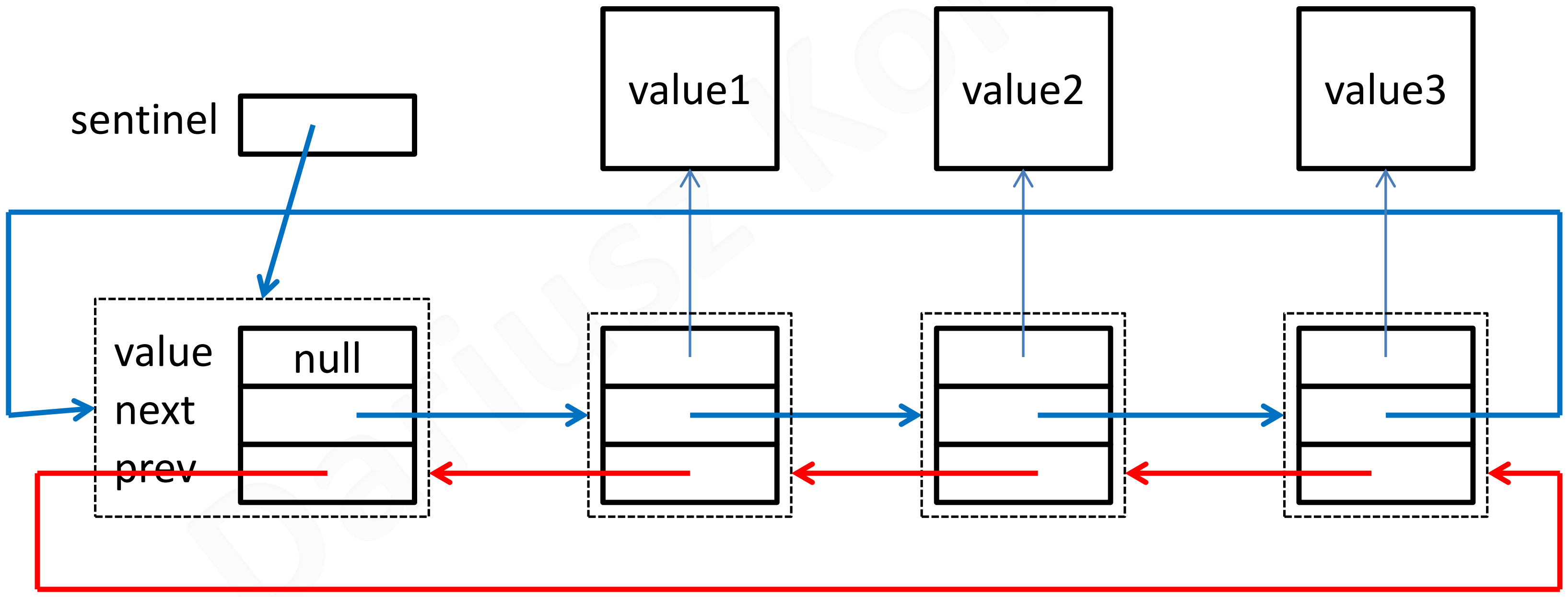
Complexity comparison 3/3

- Faster computing equipment is important, but it will not solve problems with an ineffective algorithm or a computationally complex problem.
- Below, "analysis", how we would increase the size of the input problem, if we could properly speed up the computer. What is the biggest problem we can solve in one hour?

Complexity function	Size of the biggest problem solved during 1h		
	By current computer	By computer 100 times faster	By computer 1000 times faster
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)

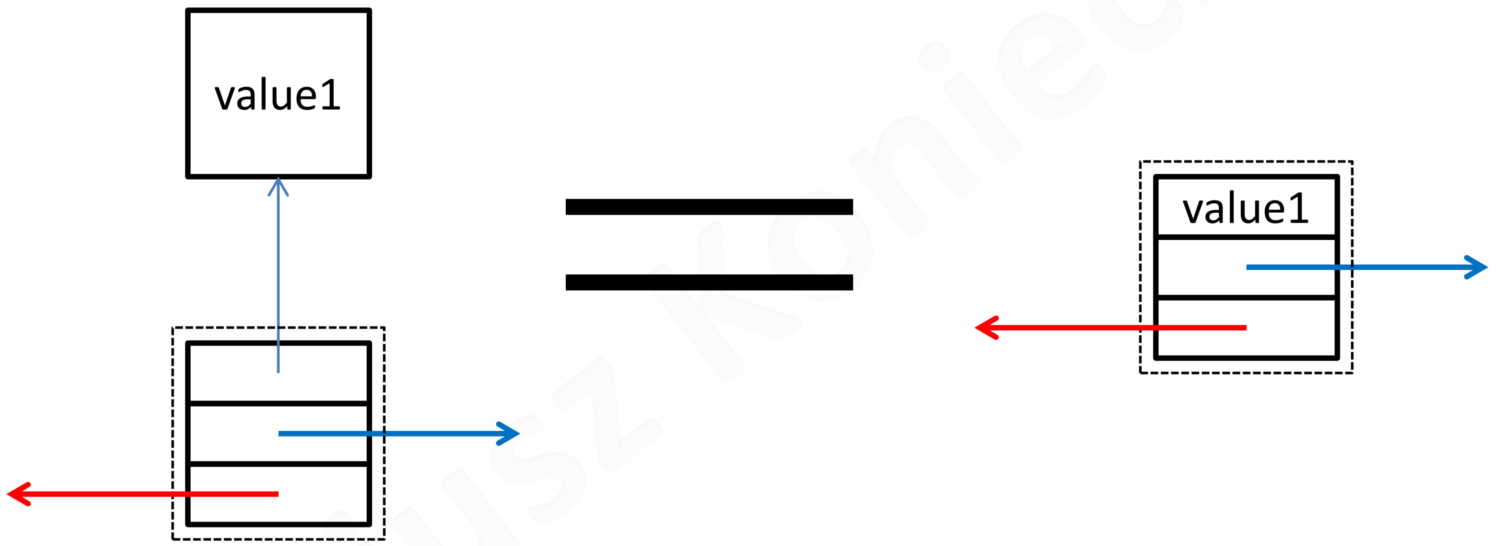
The list L2WCwS

- A two-way (bidirectional) list has elements that have a link to the previous and next item in the list.
- A cyclic list instead of **`null`** in links in the first and last element has links between these elements.
- The list with the sentinel has one element, which does not contain user data of the list but it is used to simplify the implementation, because there is always a guard in the list (even in an empty list). Most often the value reference for data is **`null`**.
- A two-way circular list with a sentinel has all of the above-mentioned features.



Graphic simplification

- For the sake of simplifying the drawings, instead of a reference to the value, a corresponding value in the rectangle will be drawn.

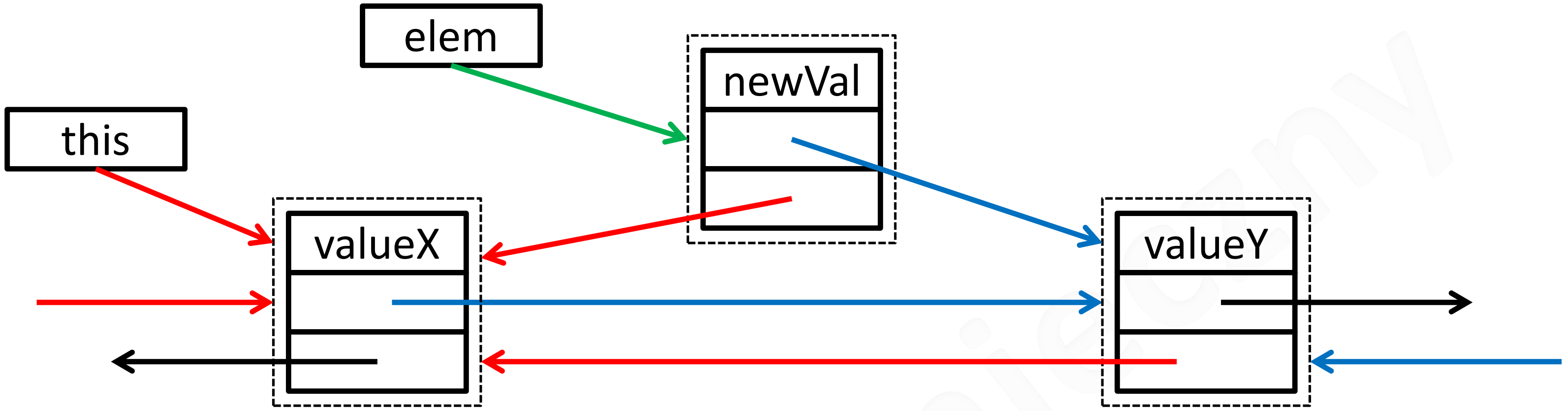


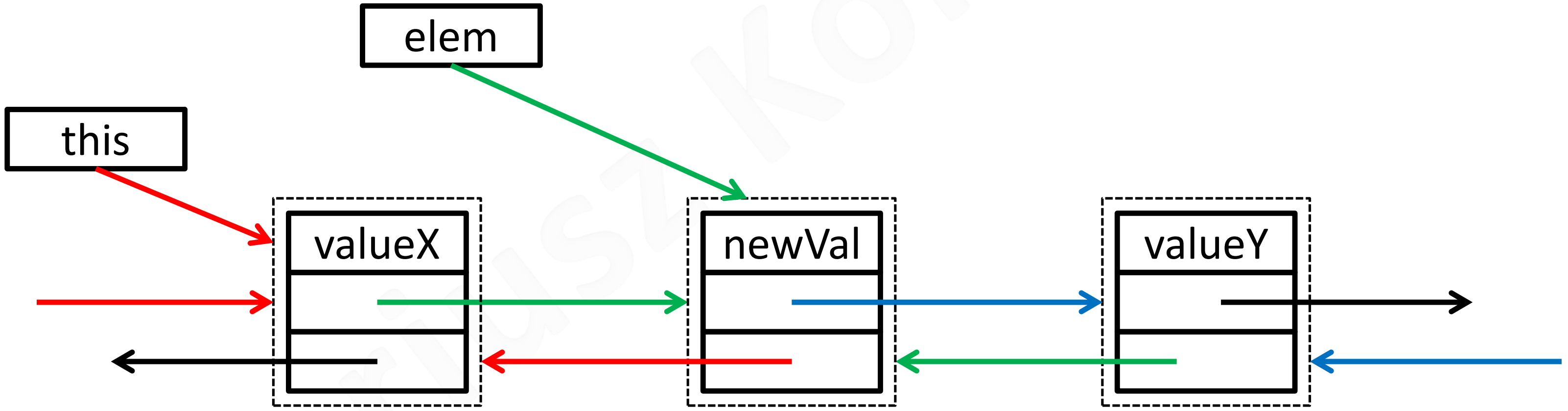
Lista L2KCzS - `Element`

aisd.list.TwoWayCycledListWithSentinel

```java
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 will be inserted <b> after this </b>*/
    public void insertAfter(Element elem){
      elem.setNext(this.getNext());
      elem.setPrev(this);
      this.getNext().setPrev(elem);
      this.setNext(elem);}
    /** elem will be inserted <b> before this </b>*/
    public void insertBefore(Element elem){
      elem.setNext(this);
      elem.setPrev(this.getPrev());
      this.getPrev().setNext(elem);
      this.setPrev(elem);}
    /** elem will be removed from a list it is inside <p>
   * <b>Assumption:</b> element is in the list and it is not a sentinel */
    public void remove(){
      this.getNext().setPrev(this.getPrev());
      this.getPrev().setNext(this.getNext());}}
```

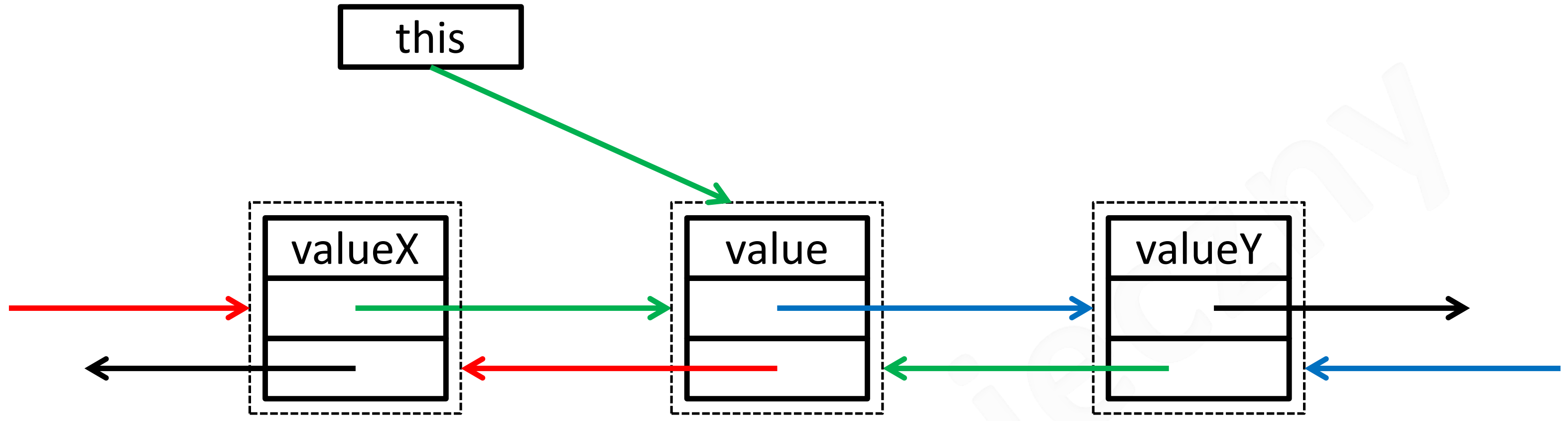
`insertAfter()`

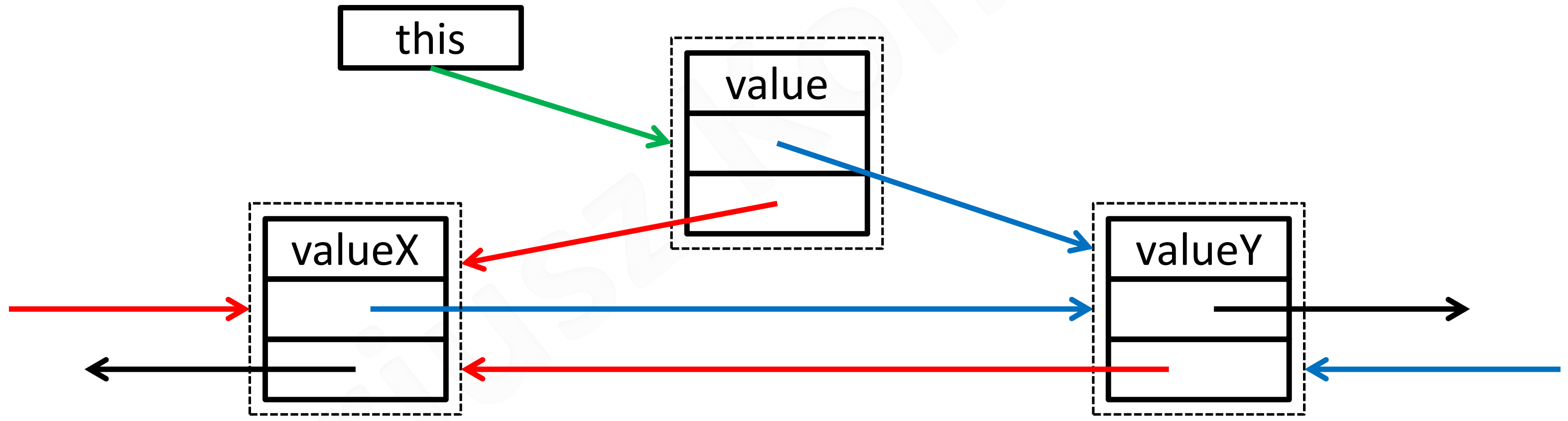




- The implementation of the method `insertBefore()` is analogous

remove()

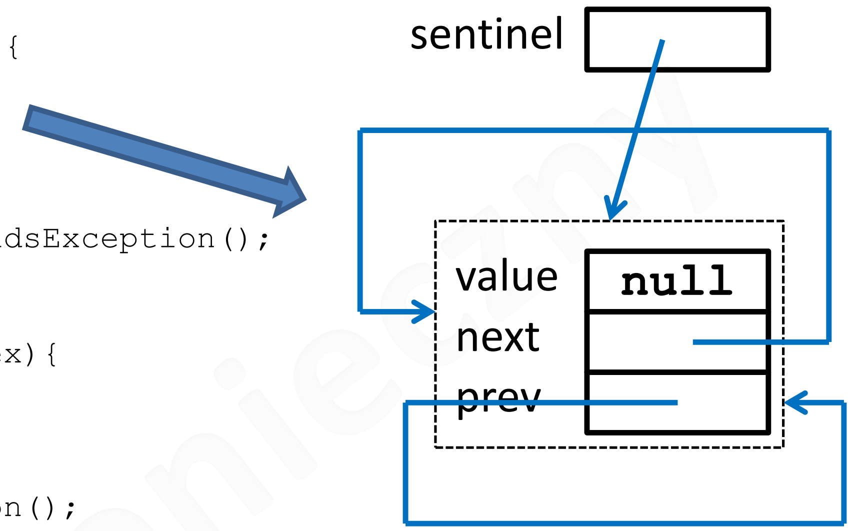




- After the removal, the item will remain in memory, but it will not be accessible (it should not be). A real remove from the memory will be during the start of the Carbage Collector.

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;}
```



- The principle of methods of the `getElement()` x2 methods is basically the same as for the one-way list. These are private methods that operate on `Element` objects, which are invisible to the user.

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;}
```

- The method `add()` uses the method `insertAfter()` and `insertBefore()` of inner class `Element`

TwoWayCycledListWithSentinel 3/5

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

- The methods `remove()` use methods `remove()` from inner class `Element`

TwoWayCycledListWithSentinel 4/5

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

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

private class TWCListIterator implements ListIterator<E>{
  boolean wasNext=false;
  boolean wasPrevious=false;
  /** sentinel */
  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();}
```

For the correct implementation of the remove methods in the iterator, you need to know which way we were moving on the list: forward or backward.

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(new Elem);
  _current=_current.getNext();}
public void set(E data) {
  if(wasNext){
    _current.setValue(data);
    wasNext=false;}
  if(wasPrevious){
    _current.getNext().setValue(data);
    wasNext=false;}}}
```

The list L2WCwS – complexity

- List L2KCzS – the worse complexity:
  - Construction – O(1)
  - `isEmpty(),clear()` – O(1)
  - `getElement()` – O(n)
    - `set(),get()` – O(n)
  - `add()` on the end – O(1)
  - `add()` with an index – O(n) – on zero position – O(1)
  - `indexOf()` – O(n)
    - `contains()` – O(n)
  - `remove()` x 2 – O(n) – on zero position – O(1)
  - `size()` – O(n)
- `ListIterator` for L2WCwS – if you implement all operations - pessimistic complexity:
  - `hasNext(),next(),hasPrevious(),previous(),nextIndex(), previousIndex()` – O(1)
  - `set()` – O(1)
  - `add(),remove()` – O(1)

- The L2WCwS list is suitable for the implementation of an unlimited queue, where each operation (except `size ()`) is constant - O(1).

Example – list od students 1/2

- Student comparison (`equals()`) should be added

aisd.W03.Student

```java
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);
  }
```

Example – list od students 2/2

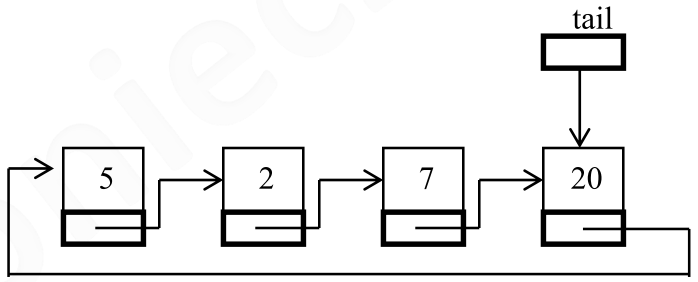
```java
public static void testStudentList(){
  TwoWayCycledListWithSentinel<Student> lista=new TwoWayCycledListWithSentinel<>();
  lista.add(new Student(4,1000));
  lista.add(new Student(5,500));
  lista.add(new Student(20,0));
  lista.add(1,new Student(1, 300));
  lista.remove(0);
  lista.remove(new 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);
}
```

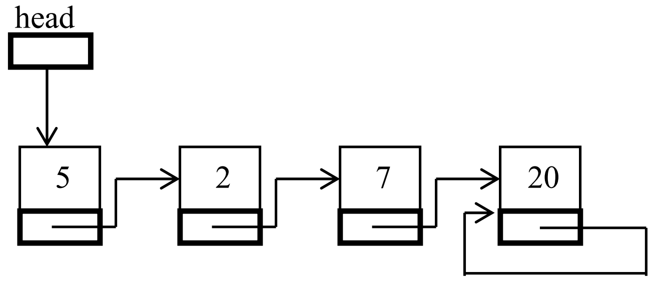
aisd.W03.AiSD_W03

```
[    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
]
```

Linked list - types

- List directions:
  - singly-linked – one-way linked
  - doubly-linked – two-way linked
- List order:
  - ordered
  - unordered
- List inner organisation:
  - with head
  - with tail
  - with head and tail
- List end:
  - ordinary-linked - straight
  - circularly-linked - circular
- Extra element in a lists:
  - without sentinel
  - with sentinel
- Special property:
  - cycled on the last element





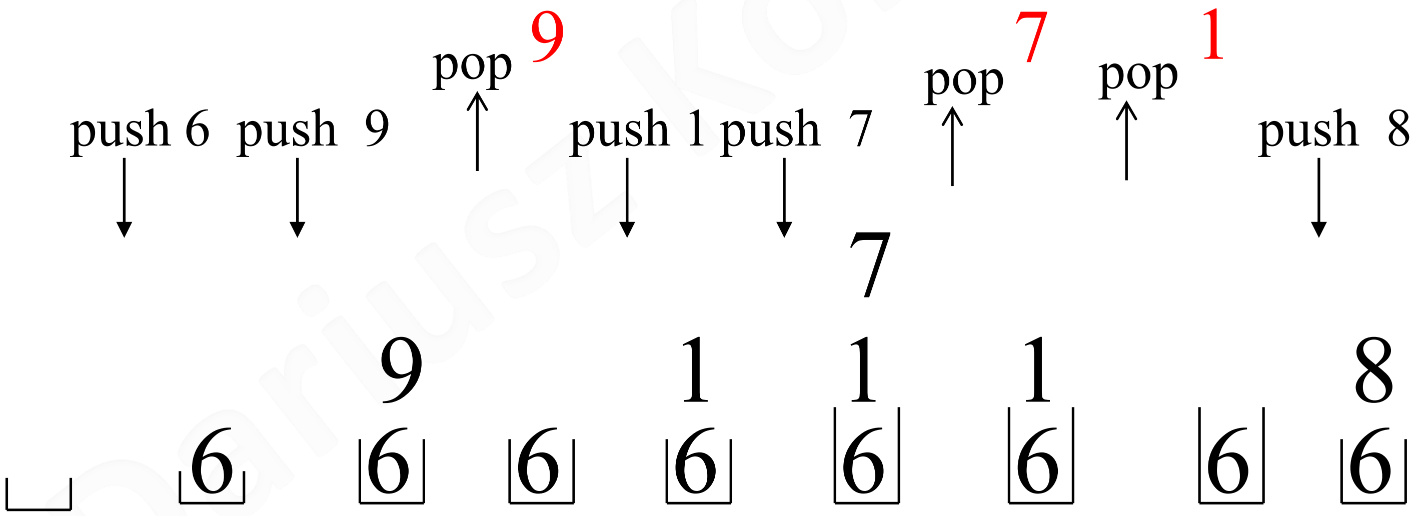
Class `LinkedList`

- The package `java.util` has a class `LinkedList`
- This is a two-way list
- Probably cyrcular
- Allows you to insert **`null`** as an element
- It is not synchronized
- In the Java library there are wrapping methods for its synchronization.
- Its methods or methods of iterators can throw exception `ConcurrentModificationException` .

Stack

- ***Stack*** is an abstract data structure, in which an element can be inserted and removed only on one end. Stack is a LIFO structure (*last in, first out*), it means last inserted element will be the first removed element.
- We can add an element (`push` operation) or delete (`pop` operation) from this end, returning as a result. In addition, you can check whether the stack is empty or full.

Basic operations on a stack:
•*Construction* – empties, or preparing the structure to work
•Checking if a stack is empty (isEmpty)
•Checking if a stack is full (isFull)
•Inserting an element on a stack (push
•Taking an element from a stack(pop)



Sequence of operations on a stack

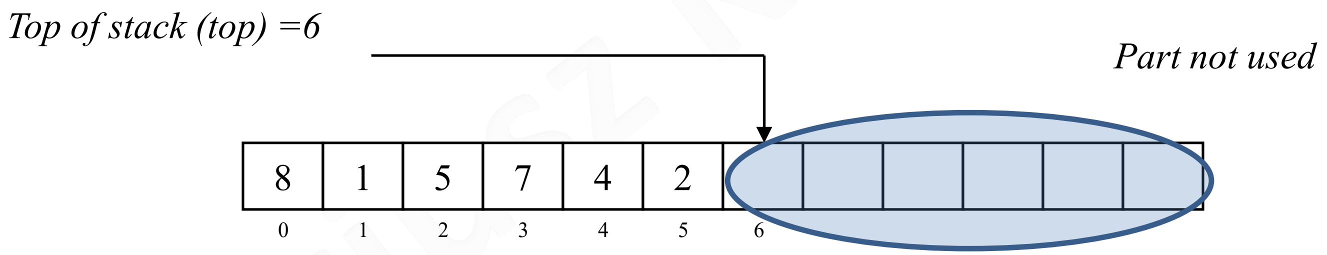
Interface IStack<T>

- You also need to initialize the stack, but in object-oriented programming it happens in the constructor. Once created, the stack should be empty.
- In Java libraries there is no interface for the stack, it is simple a class `Stack<T>`.
- Let's create an example of the `IStack<T>` interface for the stack (with additional operations):

```
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();  // return numer of elements on the stack
  T top() throws EmptyStackException;
     // return element without removing it from stack
}
```

Stack on an array

- Stack realization with an array:
  - With finite capacity
    - Just remember what the initial part of the array is the elements of the stack. This is accomplished with one integer field.
  - With infinite capacity
    - Similarly, however, when the capacity of the board is exceeded, new and larger reservations are made and the elements are transferred to the new board. (idea from `ArrayList` class)



Implementation `ArrayStack<T>` 1/2

- You can not create an object or array of objects of class `T` of the generic parameter `ArrayStack <T>`. Instead, create an array of `Object` class objects and project them onto the `T`-type object array.
- A warning appears, which can be ignored by adding a compiler directive before the method:
  `@SuppressWarnings("unchecked")`

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

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

  // generic class are in real of the class Object
  // but it allows to check types during compilation phase
  @SuppressWarnings("unchecked")
  public ArrayStack (int initialSize){
    array=(T[])(new Object[initialSize]);
    topIndex=0;
  }

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

Implementation `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];}
```

A notation `@Override` is not mandatory, but it can protect us from errors, for example::

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

It will generate a compile error because the `Empty()` method is not present in the interface

Complexity of methods of Stack

- Implementation with an array:
  - Construction : O(*n*)
  - `isEmpty`: O(1)
  - `isFull`: O(1)
  - `pop`: O(1)
  - `push`: O(1)
  - `size`: O(1)
  - `top`: O(1)

Stack with a linked list L1SwH

```
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;}
  }
```

O(1)

O(1)

O(1)

O(1)

O(1)

O(n) O(1)

O(1)

Complexity of methods of Stack

- Implementation with a linked list L1SwH :
  - construction: O(1)
  - `isEmpty`: O(1)
  - `isFull`: O(1)
  - `pop`: O(1)
  - `push`: O(1)
  - `size`: O(1) – with field `_size` (O(*n*) – without)
  - `top`: O(1)

Non-typical stacks

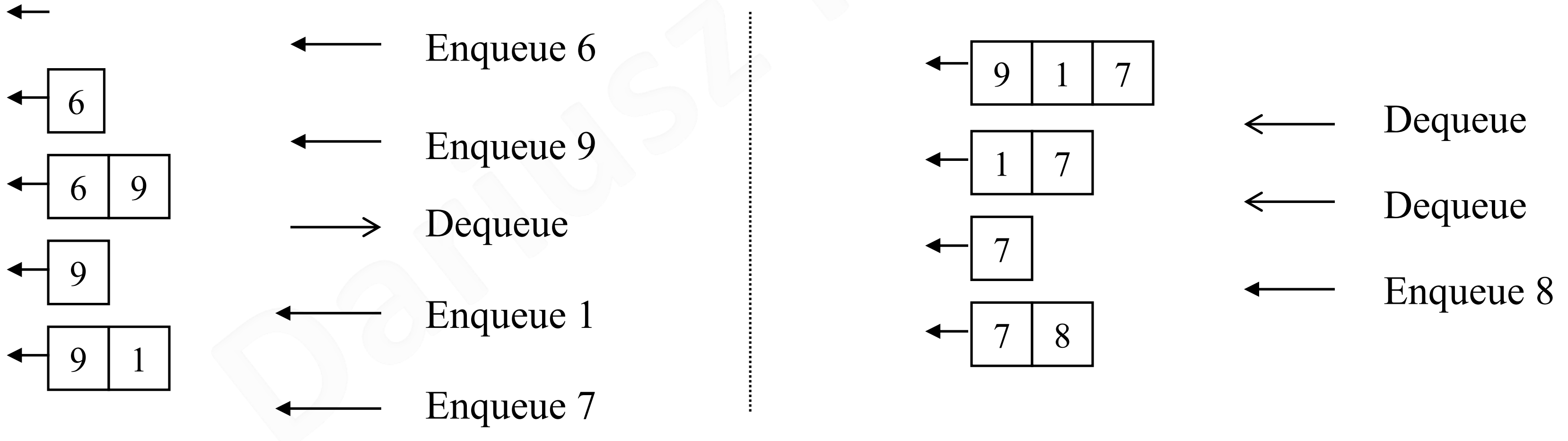
- Sinking stack:
  - A stack with limited capacity, at the time of inserting an element into a full stack, removes the element at the bottom of the stack ("sink") to make space at the top for the new element
- Additional operations on the stack:
  - Finding an element (returns the depth of an element relative to the top of the stack)
  - The maximum stack size
  - The ability to remove any element from the stack – rarely
  - Iterator to move around the stack without changing it

Stack application

- Basic dynamic structure, occurs in many algorithms (many such algorithms will be on this course)
- When programs / functions / routines are run, programs pass information through the program stack.
  - Return address from the function call
  - Parameters of the function call
  - Returned result
  - Thanks to this mechanism, recursive methods are possible
- Application screen management (window stack)
- Managing a sequence of changes (eg in a word processor) with the option of undoing and redoing (undo / redo)
- Reversing the order of words in a string
- Etc.

Queue

- The FIFO (First In First Out) queue, in short simply called a queue, is a linear structure in which the user has access to both ends, except that to one end (the end of the queue, the queue's tail) can add elements, and from the second (start of the queue, queue head) drawn. Expanding the FIFO shortcut means that the first element inserted in the queue will be the first one drawn, the rule for the subsequent elements is analogous.
- The above description shows that the following operations are needed for the queue:
  - Creation of the queue
  - Inserting an element into the queue (enqueue)
  - Get an item from the queue (dequeue)
  - Checking if the queue is empty (isEmpty)
  - Checking if the queue is full (isFull)



Interface `IQueue<T>`

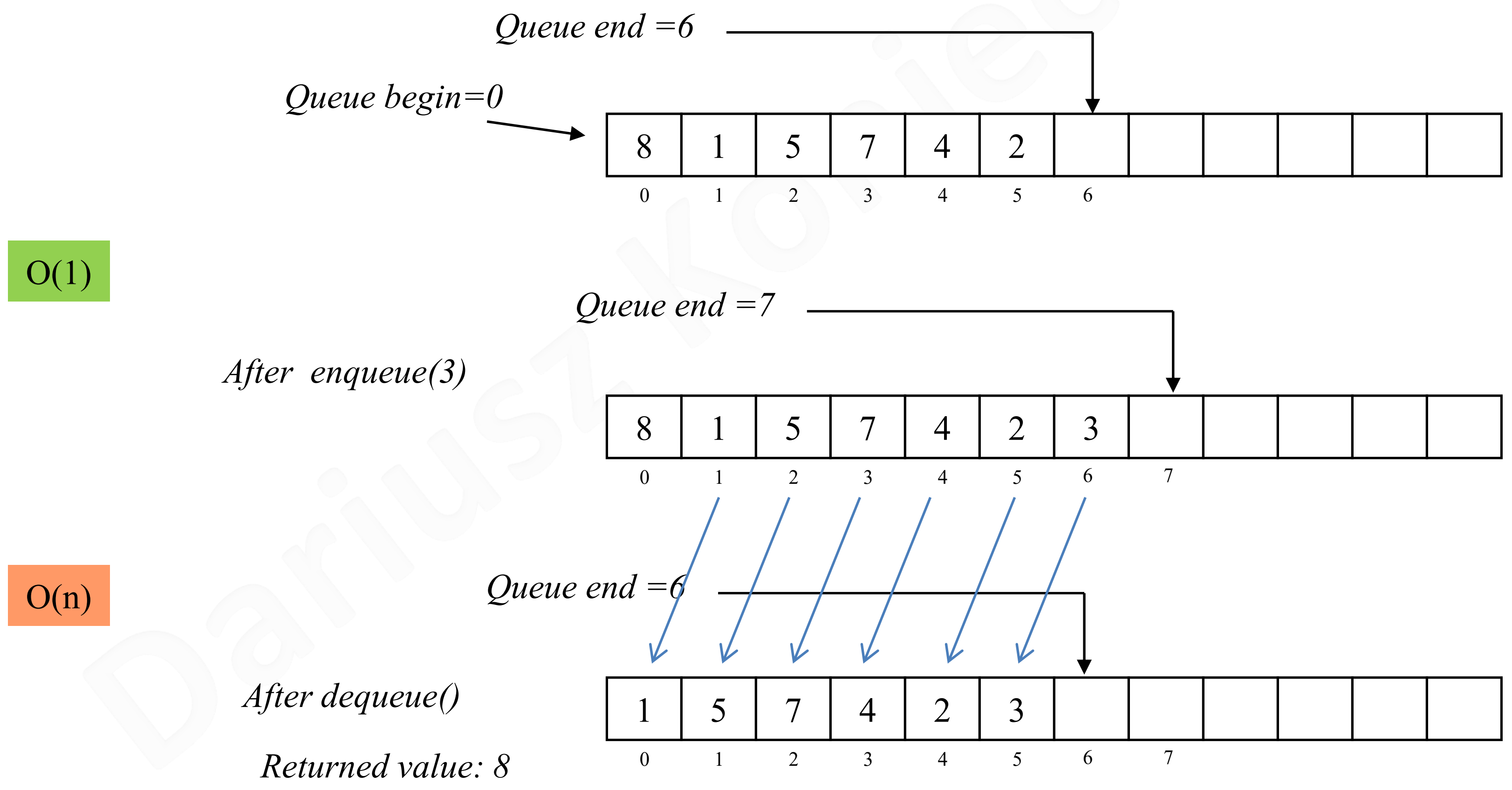
- The creation of the queue will be in the constructor. Once created, the queue should be empty.
- In Java libraries the interface for the queue is prepared to support multithreaded programming, so it is too extensive for this lecture.
- Let's create an example of the `IQueue <T>` interface for the queue (with additional operations):

```
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();  // return the number of element in a queue
  T first() throws EmptyQueueException;
     // return first element without removing it
}
```

Clear() ?

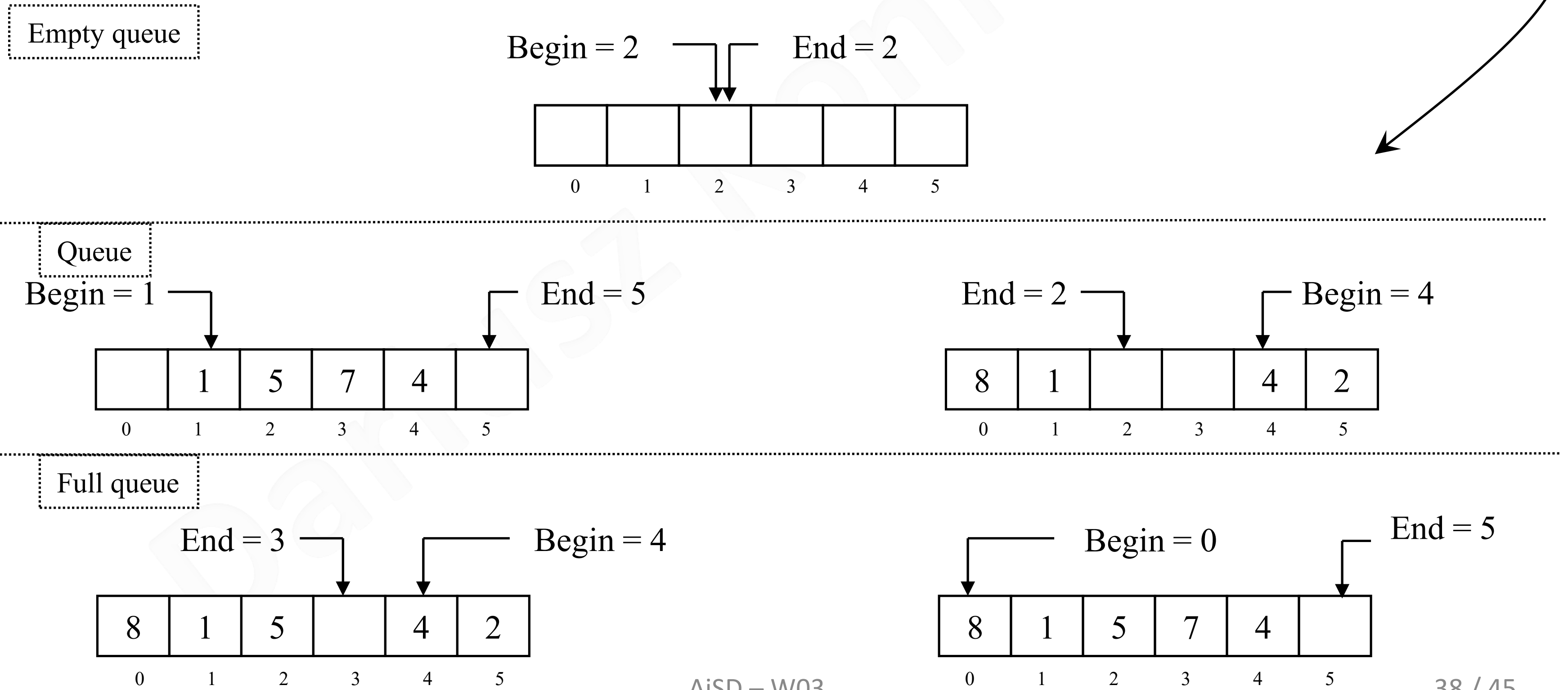
Queue with an array

- Inefficient solution:
  - a zero index is always the beginning of the queue
  - only the length of the queue is remembered (as for the stack).
  - shifting all elements one position to the left when removing an item from the queue



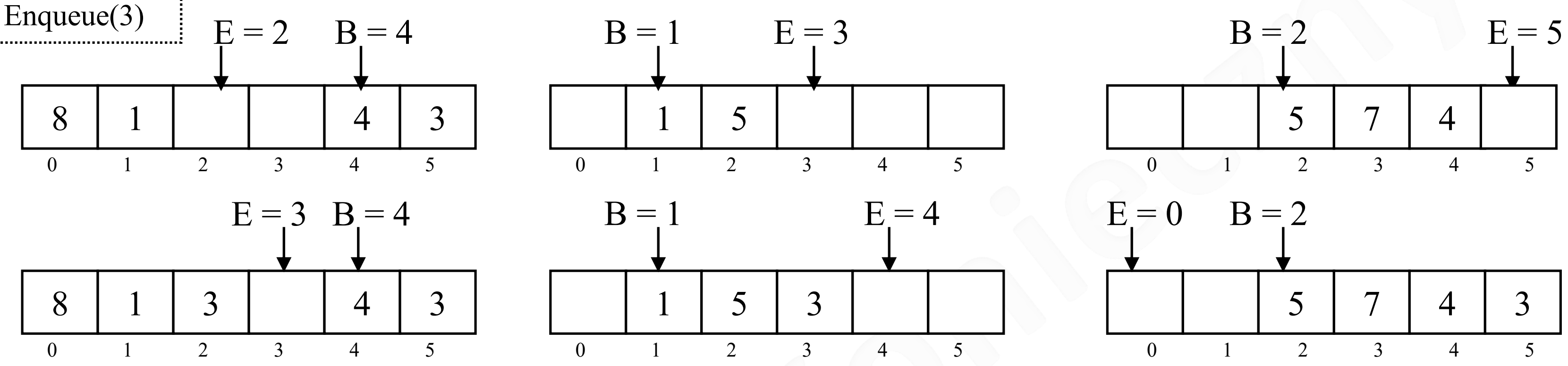
Effective implementation of the queue on the array

- Effective implementation of the queue using the array, using the elements of the table in a circular way:
  - with limited capacity
    - The most elegant solution creates array with one larger size than the one given in the constructor. In addition, the object has two indexes indicating the beginning of the queue and the place after the end of the queue
  - with unlimited capacity
    - Similarly, however, when the capacity of the array is exceeded, new and larger reservations are made and the elements are transferred to the new array.



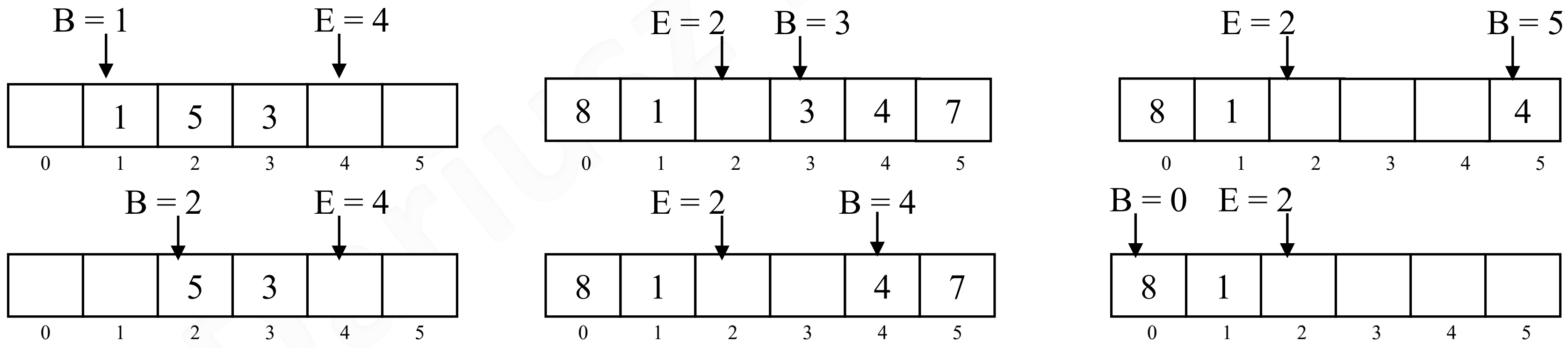
Operations `enqueue` and `dequeue`

Enqueue(3)



a) b) c)

Dequeue



Queue – implementation 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)

Queue – implementation 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)

Queue with a list L2CwS

```
public class ListQueue<E> implements IQueue<E>{
  TwoWayCycledListWithSentinel<E> _list;
  public ListQueue() {
    _list=new TwoWayCycledListWithSentinel<E>();}                O(1)
  @Override
  public boolean isEmpty() {
    return _list.isEmpty();}                                     O(1)
  @Override
  public boolean isFull() {
    return false;}                                               O(1)
  @Override
  public E dequeue() throws EmptyQueueException {
    E value=_list.remove(0);
    if(value==null) throw new EmptyQueueException();             O(1)
    return value;}
  @Override
  public void enqueue(E elem) throws FullQueueException {
    _list.add(elem);}                                            O(1)
  @Override
  public int size() {
    return _list.size();}                                        O(n)     O(1)
  @Override
  public E first() throws EmptyQueueException {
    E value=_list.get(0);
    if(value==null) throw new EmptyQueueException();             O(1)
    return value;}
}
```

Complexity of methods of queue

- Queue with list L2WCwS :
  - construction: O(1)
  - `isEmpty`: O(1)
  - `isFull`: O(1)
  - `enqueu`: O(1) (on the end of a list)
  - `dequeu`: O(1)
  - `size`: O(1) – with field `_size` (O(n) – without)
  - `first`: O(1)

Technical information - JUnit

- Library for automatic unit tests.
- There are different, subsequent versions. This lecture will use version 4 (or 5).
- Many programming environments (eg Eclipse) have visual support to present the course and process of testing, reporting, etc.
- An independent test class is created (no clutter of the tested class)
- The number of unlimited tests.
- The methods `setUp()`, `tearDown()` ... run before and after each test.
- A set of tests particularly useful when modifying methods - theoretical improvement can spoil the functionality of the class.

Demonstration of the queue using the JUnits

Queue applications

- Basic dynamic structure, occurs in many algorithms (many such algorithms will be on this course)
- Management of requests to the server (WWW, databases, etc.) in the client-server architecture
- Buffer during processing in the problem of producers-consumers
- Buffering reading from physical devices
- All streams (data, orders, multimedia) are basically queues.

"Queue" is the "FIFO queue" by default. We will get to know other types of queues at the next lectures