Politechnika Wrocławska, Wydział Informatyki i Zarządzania
Zbigniew Szpunar

Wykład 3.
Listy – struktury, implementacja, przetwarzanie

Listy

Lista – jedna z najbardziej uniwersalnych struktur danych.

Uporządkowana kolekcja elementów zapewniająca dostęp do dowolnego elementu (pobrania elementu po jego zidentyfikowaniu).

Przykłady, w których lista występuje jako naturalna struktura danych:

- lista studentów zapisanych na kurs,
- lista dokumentów w segregatorze,
- lista *grup produktów* w katalogu, wraz z *listami produktów* w każdej grupie,
- lista obecności pracowników i zadań do wykonania przez każdego pracownika.

Elementy listy są obiektami, które podlegają przetwarzaniu.

Lista jest, w pewnym sensie, alternatywą dla tablic.

Politechnika Wrocławska, Wydział Informatyki i Zarządzania
Zbigniew Szpunar

Listy

W wielu przypadkach lista jest rozszerzeniem możliwości tablicy (np. nie występuje tu ograniczenie rozmiaru, charakterystyczne dla tablicy).

Są przypadki, w których tablice wykazują przewagę (np. tablice zapewniają bezpośredni dostęp indeksowy do elementów).

Rodzaje list :

- proste i cykliczne,
- jednokierunkowe, dwukierunkowe i wielokierunkowe,
- uporządkowane i nieuporządkowane.

Listy

Lista musi implementować co najmniej 4 operacje:

- size – zwraca rozmiar listy (liczbę elementów listy),
- insert – wstawia element na wskazaną pozycję listy (rozmiar listy wzrasta o 1).
- delete – usuwa element ze wskazanej pozycji listy (rozmiar listy maleje o 1) oraz zwraca wartość usuniętego elementu.
- get – zwraca wartość elementu znajdującego się na wskazanej pozycji listy.

Jeśli wskazana pozycja listy w operacjach: insert, delete, get wykracza poza rozmiar listy, generowany jest wyjątek *IndexOutOfBoundsException*.

Politechnika Wrocławska, Wydział Informatyki i Zarządzania
Zbigniew Szpunar

Przykładowe rozszerzenie operacji listowych

W praktyce wymagana jest większa liczba operacji, np.:

- set – zmień (zamiast: delete i insert), zwróć wartość przed zmianą,
- add – dołącz element na koniec listy (rozmiar zwiększa się o 1),
- delete (wartość) – usuwa element o podanej wartości, zwracając true (jeśli taki element jest) lub nie zmienia listy, zwracając "false" (gdy brak takiego elementu).

  "Element o wskazanej wartości" oznacza element, którego metoda equals() stwierdza zgodność z wartością poszukiwaną.
- contains – sprawdza, czy w liście występuje element o danej wartości,
- indexOf – zwraca pozycję (indeks) pierwszego wystąpienia danej wartości; jeśli brak wartości – zwraca wartość –1,
- isEmpty – zwraca true, jeśli lista jest pusta (czyli size()=0) lub false dla listy niepustej,
- iterator – zwraca iterator dla listy,
- clear – usuwa wszystkie elementy z listy.

Politechnika Wrocławska, Wydział Informatyki i Zarządzania
Zbigniew Szpunar

Przykładowy interfejs listowy

```
public interface List extends Iterable {
   public int size();                          // zwraca rozmiar listy (liczbę elementów listy)
   public void insert(int index, Object value) throws IndexOutOfBoundsException;
                                               // wstawia element na pozycję index
   public Object delete(int index) throws IndexOutOfBoundsException;
                                  // usuwa element z pozycji index, zwraca wartość usuniętego elementu

   public boolean delete(Object value);     // usuwa element o podanej wartości, zwracając
         // true (jeśli taki element jest) lub nie zmienia listy, zwracając false (gdy brak elementu)

   public Object get(int index) throws IndexOutOfBoundsException;
                                                  // zwraca wartość elementu z pozycji index

   public Object set(int index, Object value) throws IndexOutOfBoundsException;
            // zmień wartość elementu z pozycji index, zwróć wartość elementu sprzed zmiany

   public void add(Object value);      // dołącza element na koniec listy
   public boolean contains (Object value);      // sprawdza, czy w liście występuje element
                                  // o danej wartości; zwraca true – jeśli jest, false – jeśli nie ma

   public int indexOf(Object value); // zwraca indeks pierwszego wystąpienia danej
                                                  // wartości; jeśli brak wartości - zwraca "–1"
   public boolean isEmpty(); // zwraca true, jeśli lista jest pusta lub false dla listy niepustej,
   public void clear(); // usuwa wszystkie elementy z listy
}
```

Przedstawiony interfejs List jest specjalizacją (podklasą) interfejsu Iterable, zawierającą metodę iterator();

```
public interface Iterable {
  public Iterator iterator();
}
```

co oznacza, że dla listy może być określony iterator umożliwiający nawigowanie po elementach listy.

Przykład (z zapisami studentów na kurs):

1.Z użyciem tablicy do implementacji listy:

```
String [ ] zapisy = ... ;
zapisy[0] = "Kowalski";
zapisy[1] = "Nowakowski";
zapisy[2] = "Zaręba";
...
for (int i=0; i<zapisy.length; i++) {System.out.println( zapisy[i] );}
```

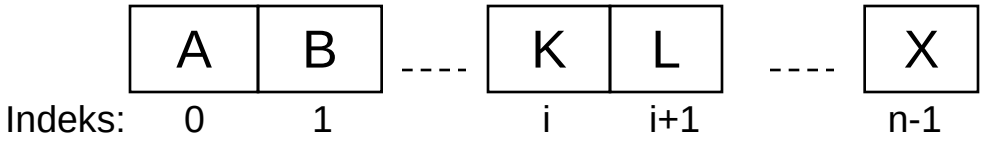
2. Z użyciem interfejsu List (bez ustalenia konkretnej implementacji listy)

```
List zapisy = ... ;
zapisy.add("Kowalski”);
zapisy.add("Nowakowski”);
zapisy.add("Zaręba”);
...
Iterator i = zapisy.iterator();
for (i.first(); !i.isDone(); i.next()) {System.out.println( i.current() );}
```

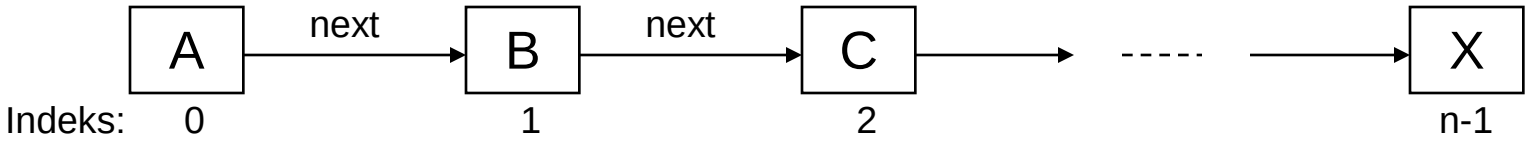
Politechnika Wrocławska, Wydział Informatyki i Zarządzania
Zbigniew Szpunar

Implementowanie list

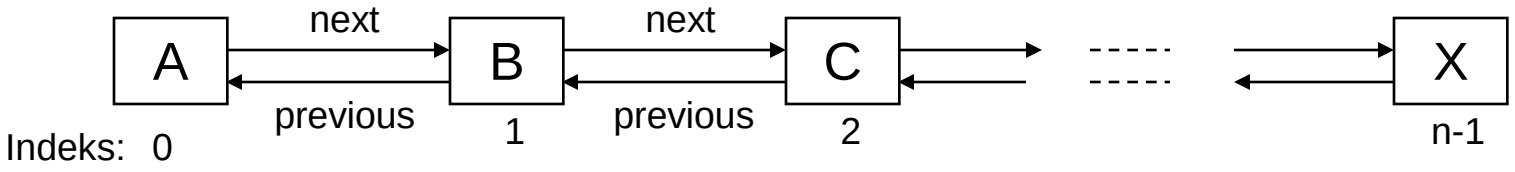
Zazwyczaj implementuje się listy (bez elementów o wartościach pustych (null):
- z użyciem tablic (elementy listy ei są elementami pewnej tablicy),



- z użyciem list wiązanych (elementy ei są powiązane ze sobą za pomocą referencji – odwołań, wskaźników); każdy element zawiera odwołanie do elementu następnego (lista pojedynczo wiązana lub jednokierunkowa):



lub do elementu następnego oraz poprzedniego (tzw. lista podwójnie wiązana lub dwukierunkowa):



Politechnika Wrocławska, Wydział Informatyki i Zarządzania
Zbigniew Szpunar

Implementowanie listy

1. Kolejne elementy listy są w kolejnych elementach tablicy.
2. Dostęp do elementów jest sekwencyjny; odczyt (pobranie) i zapis (modyfikacja) elementu odbywa się przy użyciu indeksu (wówczas taka implementacja jest najbardziej efektywna).
3. Bardzo nieefektywne jest:
   - wstawianie elementów listy (wymaga inkrementacji "zajętości" tablicy lub "zwiększenia rozmiaru tablicy" oraz wymaga "przesunięcia w prawo" od miejsca określonego indeksem wstawianego elementu, by zrobić miejsce na ten element),
   - usuwanie elementów listy (wymaga dekrementacji "zajętości" tablicy oraz "przesunięcia w lewo" elementów dla zlikwidowania "dziury" po usuniętym elemencie).

   W rzeczywistości operacje przesuwania oznaczają fizyczne kopiowanie elementów.
4. "Zwiększenie rozmiaru tablicy" - oznacza utworzenie nowej, większej tablicy oraz przepisanie do niej elementów listy.

Politechnika Wrocławska, Wydział Informatyki i Zarządzania
Zbigniew Szpunar

Implementowanie listy w tablicy

```
public class ArrayList implements List
{  private static final int DEFAULT_INITIAL_CAPACITY = 16; // domyślny rozmiar pocz. tablicy
   private final int _initialCapacity;      // bieżąca wielkość tablicy
   private Object [ ] _array;               // tablica na elementy listy
   private int _size;                       // bieżąca długość listy

   public ArrayList(int initialCapacity) // konstruktor listy kreujący tablicę o danym rozmiarze
    {  assert initialCapacity >0 : "Inicjalny rozmiar tablicy musi być dodatni";
        _initialCapacity = initialCapacity;
        clear();
    }
   public ArrayList()                       // konstruktor listy kreujący tablicę o domyślnym rozmiarze
    { this(DEFAULT_INITIAL_CAPACITY); }

   public ArrayList(Object [ ] array) // konstruktor listy z wypełnieniem listy elementami tablicy
    {  _initialCapacity = array.length;
        clear();
        System.arraycopy(array, 0, _array, 0, array.length);
        _size = array.length;
    }
     public void clear() {
        _array = new Object[_initialCapacity];
        _size = 0;
    }  // c.d.n.
```

Politechnika Wrocławska, Wydział Informatyki i Zarządzania
Zbigniew Szpunar

Implementowanie listy w tablicy – c.d.

```
// c.d.
 public void insert(int index, Object value) throws IndexOutOfBoundsException {
   if (index<0 || index>_size) throw new IndexOutOfBoundException();
   ensureCapacity(_size + 1);
   System.arraycopy(_array, index, _array, index + 1, _size - index);
   _array[index] = value;
   ++_size;
 }

 private void ensureCapacity(int capacity) {
   if (_array.length < capacity) {                    //strategia tworzenia tablicy z 50% zapasem
      Object [ ] copy = new Object[capacity + capacity / 2];
      System.arraycopy(_array, 0, copy, 0, _size);
      _array = copy;   // następuje dynamiczna zmiana rozmiaru tablicy _array z elementami listy
   }
 }

 public void add(Object value) {
    insert(size(), value);      // add() to szczególny przypadek insert(): wstawienie elementu
                                // na końcu listy
 }
// c.d.n.
```

Implementowanie listy w tablicy – c.d.

```
// c.d.
 public Object get(int index) throws IndexOutOfBoundsException {
   checkOutOfBounds(index);
   return _array[index];
 }

 public Object set(int index, Object value) throws IndexOutOfBoundsException {
   checkOutOfBounds(index);
   Object oldValue = _array[index];
   _array[index] = value;
   return oldValue;
 }

 private void checkOutOfBounds(int index) throws IndexOutOfBoundsException {
   if (index < 0 || index >= size())  // kontrola zakresu indeksu dla listy (nie dla tablicy!)
       throw new IndexOutOfBoundsException();
 }
 // c.d.n.
```

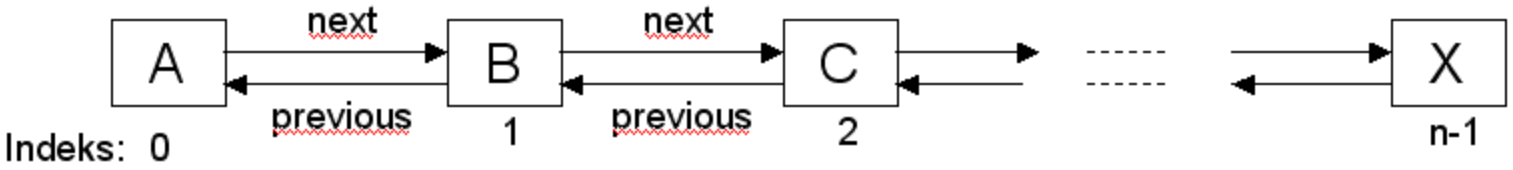
Implementowanie listy w tablicy – c.d.

```
// c.d.
 public int indexOf(Object value) {
   int i =0;
   while(i < _size && !value.equals(_array[i]))   ++i;
   return i<_size ? i : -1;
 }

 public Iterator iterator() {
   return new ArrayIterator(_array, 0, _size);  // w treści wykładu 2. Jest: "IteratorTablicowy"
 }

public boolean contains(Object value) {
   return indexOf(value) != -1;
 }

 public boolean isEmpty() {
   return _size == 0;      // lub: return size()==0
 }

 public int size() {
   return _size;
 }

// c.d.n.
```

Implementowanie listy w tablicy – c.d.

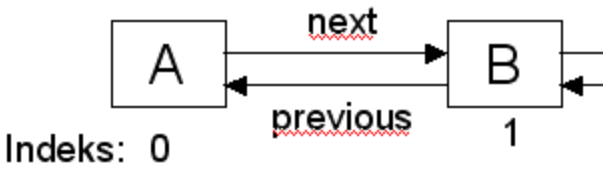
```
// c.d.
 public Object delete(int index) throws IndexOutOfBoundsException {
     checkOutOfBounds(index);
     Object value = _array[index];
     int copyFrom = index + 1;
     if (copyFrom < _size)
          System.arraycopy(_array, copyFrom,
                                   _array, index,
                                   _size - copyFrom);
     --_size;
     return value;
 }

 public boolean delete(Object value) {
    int index = indexOf(value);            // element "przeliczony" na wartość indeksu
    if (index != -1)
         delete(index);
    return index != -1;
 }
}
```

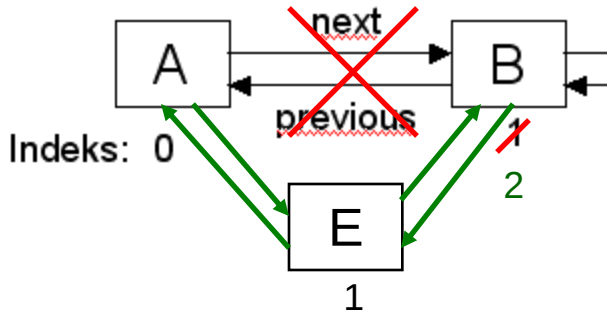
Implementowanie listy jako listy podwójnie wiązanej



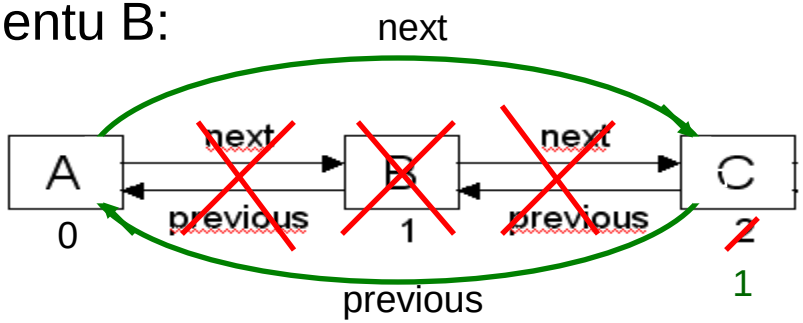
1. Operacje wstawiania i usuwania elementów nie wymagają tu kosztownego przesuwania (kopiowania) elementów, lecz jedynie aktualizacji kilku łączników (referencji, wskaźników).
2. Dostęp do wskazanej pozycji w dużych listach jest kosztowny.

- wstawienie elementu E na pozycję 1 (czyli pomiędzy A i B):





- usunięcie elementu B:



Politechnika Wrocławska, Wydział Informatyki i Zarządzania
Zbigniew Szpunar

Implementowanie listy jako listy podwójnie wiązanej

Z listą podwójnie wiązaną używane są zazwyczaj dwa dodatkowe łączniki:
- do pierwszego elementu (tzw. głowa, ang. *head*),
- do ostatniego elementu (tzw. ogon, ang. *tail*).

Zwane też początkiem i końcem listy. Umożliwiają one bezpośredni dostęp do obydwu krańców listy. Element organizacyjny, reprezentujący początek i koniec listy jest nazywany wartownikiem lub obiektem pustym.

```
// element listy jest zdefiniowany jako wewnętrzna prywatna klasa Element

public class LinkedList implements List {
    private final Element _headAndTail = new Element(null); // wartownik
    private int _size;    // rozmiar listy, długość listy

    public LinkedList() {
       clear();           // konstruktor domyślny; sprowadza listę do stanu początkowego
    }
// c.d.n.
```

Implementowanie listy jako listy podwójnie wiązanej

```
// c.d.

private static final class Element {   // definiowanie klasy elementu listy
    private Object _value;
    private Element _previous;
    private Element _next;

    public Element(Object value)
      { setValue(value); }

    public void setValue(Object value)
      { _value = value;  }

    public Object getValue()
      { return _value;   }

    public Element getPrevious()
      { return _previous;}

    public void setPrevious(Element previous) {
       assert previous != null : "Wskaźnik na element poprzedni nie może być pusty";
       _previous = previous;
    }
// c.d.n.
```

Implementowanie listy jako listy podwójnie wiązanej

```
// c.d. definiowanie klasy elementu

   public Element getNext()
     { return _next; }

    public void setNext(Element next) {
      assert next != null : "Wskaźnik na element następny nie może być pusty";
      _next = next;
    }

// wstawienie danego (this) elementu przed element next
   public void attachBefore(Element next) {
       Element previous = next.getPrevious();
       setNext(next);
       setPrevious(previous);
       next.setPrevious(this);
       previous.setNext(this);
    }
    public void detach() {   // usunięcie elementu
        _previous.setNext(_next);
        _next.setPrevious(_previous);
    }
}    // koniec definiowania klasy elementu
// c.d.n.
```

Implementowanie listy jako listy podwójnie wiązanej

```
// c.d.  - implementacja interfejsu
 public void insert(int index, Object value) throws IndexOutOfBoundsException {
     if (index<0 || index>_size) throw new IndexOutOfBoundsException();
     Element element = new Element(value);
     element.attachBefore(getElement(index));
     ++_size;
 }
// pobieranie elementu znajdującego się na pozycji index
// wybór kierunku przeszukiwania listy przyspiesza działanie
    private Element getElement(int index) {
        return index< _size/2 ? getElementForwards(index) : getElementBackwards(index);
    }
    private Element getElementForwards(int index) {// dojście do podanego indexu "w przód"
        Element element = _headAndTail.getNext();
        for (int i = index; i > 0; --i)
            element = element.getNext();
        return element;
    }
    private Element getElementBackwards(int index) {// dojście do podanego indexu "do tyłu"
        Element element = _headAndTail;
        for (int i = _size - index; i > 0; --i)
            element = element.getPrevious();
        return element;
    } // c.d.n.
```

Implementowanie listy jako listy podwójnie wiązanej

```
// c.d.  - implementacja interfejsu listy
 private void checkOutOfBounds(int index) throws IndexOutOfBoundsException {
    if (index < 0 || index >= size()) throw new IndexOutOfBoundsException();
 }
 public void add(Object value)
    { insert(size(), value); }

public int size()
    { return _size; }

 public Object get(int index) throws IndexOutOfBoundsException {
    checkOutOfBounds(index);
    return getElement(index).getValue();
 }

 public Object set(int index, Object value) throws IndexOutOfBoundsException {
    checkOutOfBounds(index);
    Element element = getElement(index);
    Object oldValue = element.getValue();
    element.setValue(value);
    return oldValue;
}
// c.d.n.
```

Politechnika Wrocławska, Wydział Informatyki i Zarządzania
Zbigniew Szpunar

Implementowanie listy jako listy podwójnie wiązanej

```
// c.d.  - implementacja interfejsu listy
 public Object delete(int index) throws IndexOutOfBoundsException {
     checkOutOfBounds(index);
     Element element = getElement(index);
     element.detach();
     --_size;
     return element.getValue();
 }
 public boolean delete(Object value) {
     Element e = _headAndTail.getNext();
     while (e != _headAndTail && ! value.equals(e.getValue()))
              e = e.getNext();
     if (e != _headAndTail)
             { e.detach(); --_size; return true; }
       else return false;
 }
 public boolean contains(Object value)
    { return indexOf(value) != -1; }

 public void clear() {
    _headAndTail.setPrevious(_headAndTail);
    _headAndTail.setNext(_headAndTail);
    _size = 0;
 }  // c.d.n.
```

Implementowanie listy jako listy podwójnie wiązanej

```
// c.d.  - implementacja interfejsu listy

 public int indexOf(Object value) {
    int index = 0;
    Element e = _headAndTail.getNext();
    while( e != _headAndTail && ! value.equals(e.getValue()))
           { e = e.getNext(); ++index; }
    return e!=_headAndTail ? index : -1;
 }

 public boolean isEmpty()
    { return _size == 0; }

 public Iterator iterator()
    { return new ValueIterator(); }

// koniec implementacji interfejsu listy
// c.d.n.
```

Implementowanie listy jako listy podwójnie wiązanej

```
// c.d.
// iterator po wartościach elementów listy

 private final class ValueIterator implements Iterator {
    private Element _current = _headAndTail;
    public void first()
     { _current = _headAndTail.getNext(); }
    public void last()
     { _current = _headAndTail.getPrevious(); }
    public boolean isDone()
     { return _current == _headAndTail; }
    public void next()
     { _current = _current.getNext(); }
    public void previous()
     { _current = _current.getPrevious(); }
    public Object current() throws IndexOutOfBoundsException {
      if (isDone())
              throw new IndexOutOfBoundsException();
      return _current.getValue();
     }
 }
}
```

Implementowanie listy – wybór sposobu implementacji

Wybór sposobu implementacji listy zależy od konkretnego zestawu operacji, jakie na danej liście wykonuje się najczęściej.
Przykład działań z listą, z wykorzystaniem jej iteratora
(klasę Student zdefiniowano w zadaniu z iteratorami, po poprzednim wykładzie)

```
public class GrupaStud
{ List lista = new ArrayList();
  public GrupaStud(){  }
  private class StudentZZaliczeniem implements Predicate {
        public boolean accept(Object s){
          return ((Student)s).ocena>2;
       }
  }
  public void wyswietlListe(Iterator itab) {
       System.out.println("     NrInd  Nazwisko        Imię              Ocena");
         for (itab.first(); !itab.isDone(); itab.next()) {
            Student st = (Student) itab.current();
            st.wyswietlDane();
         }
         System.out.println("== koniec wykazu ==");
       }
    }
```

Przykład działań z listą, z wykorzystaniem jej iteratora
(klasę Student zdefiniowano przy omawianiu iteratorów, na/po wykładzie 1.)

```
public class GrupaStud {
    LinkedList grupa = new LinkedList();
    .....
 public void main(String[] args){
     GrupaStud g = new GrupaStud();
     LinkedList grupa = new LinkedList();
     grupa.add(new Student(100,"Kowalski","Jan",3.5));
     grupa.add(new Student(200,"Nowak","Anna",4.5));
     grupa.add(new Student(300,"Jankowski","Adam",4.0));
     g.wyswietlListe(grupa.iterator());
     grupa.insert(2, new Student(500,"Adamski","Lech",5.0));
     g.wyswietlListe(grupa.iterator());
     grupa.delete(1);
     g.wyswietlListe(grupa.iterator());
     grupa.add(new Student(400,"Maliniak","Stefan",2.0));
     grupa.add(new Student(600,"Anioł","Stanisław",4.0));
     g.wyswietlListe(grupa.iterator());
     StudentBezZaliczenia studBezZal = new StudentBezZaliczenia();
     IteratorFiltrujacy itbezZal = new IteratorFiltrujacy(grupa.iterator(),studBezZal);
     wyswietlListe(itbezZal);
    }
}
```

Przykład działań z listą, z wykorzystaniem jej iteratora
(klasę Student zdefiniowano przy omawianiu iteratorów, na/po wykładzie 1.)

```
public class GrupaStud {
    LinkedList grupa = new LinkedList();
    .....
 public void main(String[] args){
     GrupaStud g = new GrupaStud();
     LinkedList grupa = new LinkedList();
     grupa.add(new Student(100,"Kowalski","Jan",3.5))
     grupa.add(new Student(200,"Nowak","Anna",4.5))
     grupa.add(new Student(300,"Jankowski","Adam",
     g.wyswietlListe(grupa.iterator());
     grupa.insert(2, new Student(500,"Adamski","Lech"
     g.wyswietlListe(grupa.iterator());
     grupa.delete(1);
     g.wyswietlListe(grupa.iterator());
     grupa.add(new Student(400,"Maliniak","Stefan",2.
     grupa.add(new Student(600,"Anioł","Stanisław",4.
     g.wyswietlListe(grupa.iterator());
     StudentBezZaliczenia studBezZal = new StudentB
     IteratorFiltrujacy itbezZal = new IteratorFiltrujacy(g
     wyswietlListe(itbezZal);
    }
}
```

BlueJ: Terminal Window - Listy

Options

```
     NrInd  Nazwisko        Imię            Ocena
     100    Kowalski        Jan              3,50
     200    Nowak           Anna             4,50
     300    Jankowski       Adam             4,00
== koniec wykazu ==
     NrInd  Nazwisko        Imię            Ocena
     100    Kowalski        Jan              3,50
     200    Nowak           Anna             4,50
     500    Adamski         Lech             5,00
     300    Jankowski       Adam             4,00
== koniec wykazu ==
     NrInd  Nazwisko        Imię            Ocena
     100    Kowalski        Jan              3,50
     500    Adamski         Lech             5,00
     300    Jankowski       Adam             4,00
== koniec wykazu ==
     NrInd  Nazwisko        Imię            Ocena
     100    Kowalski        Jan              3,50
     500    Adamski         Lech             5,00
     300    Jankowski       Adam             4,00
     400    Maliniak        Stefan           2,00
     600    Anioł           Stanisław        4,00
== koniec wykazu ==
     NrInd  Nazwisko        Imię            Ocena
     400    Maliniak        Stefan           2,00
== koniec wykazu ==
```

Politechnika Wrocławska, Wydział Informatyki i Zarządzania
Zbigniew Szpunar

Implementowanie listy – wybór sposobu implementacji

Jak już wcześniej zaznaczono, wybór sposobu implementacji listy zależy od konkretnego zestawu operacji, jakie na danej liście wykonuje się najczęściej.

Należy przeanalizować wszystkie okoliczności, które przemawiają za i przeciw każdemu ze sposobów implementacji i następnie wybrać ten, który zapewni największą wydajność przetwarzania.