

Informacje wstępne

1. Rok IZ – rok akademicki 2013/14 (studia niestacjonarne)

Algorytmy i struktury danych (INZ1645)

Prowadzący: dr inż. Zbigniew Szpunar
Wydział IZ – Instytut Informatyki
pok. 201/14 D-2,

zbigniew.szpunar@pwr.wroc.pl

konsultacje:

niedziele zjazdów, 15:00 – 17:00 (sala 2.28 B4)

piątki, 13:00 - 15:00 (pok. 201/14 D-2)

Informacje wstępne

Formy dydaktyczne, wymiar godzinowy:

Wykład: 18 godzin (2 godz./zjazd)

Ćwiczenia: 9 godzin (2 godz. co drugi zjazd)

Laboratorium: 18 godzin (2 godz./zjazd)

Wymagania wstępne

Znajomość zagadnień z kursu „Podstawy programowania”

Cel kursu

Celem kursu jest poznanie przez Słuchaczy wybranych zagadnień z zakresu technik algorytmicznych, w powiązaniu z różnymi strukturami danych. Efektem praktycznym kursu ma być umiejętność wykorzystywania poznanych technik algorytmicznych w sprawnym rozwiązywaniu zadań programowo-implementacyjnych z zakresu inżynierii oprogramowania.

Informacje wstępne

Tematyka kursu obejmuje następujące zagadnienia:

- Iteracja i iteratory,
- Złożoność obliczeniowa,
- Listy,
- Kolejki, kolejki priorytetowe,
- Stosy,
- Algorytmy sortowania,
- Binarne wyszukiwanie i wstawianie,
- Drzewa,
- Słowniki i haszowanie,
- Grafy

Informacje wstępne

Tryb oceny

Zaliczenie wykładu i ćwiczeń na wspólną ocenę (grupa kursów), określoną na podstawie [egzaminu](#), z uwzględnieniem aktywności podczas zajęć.

Data egzaminu zostanie ogłoszona w ciągu najbliższego tygodnia.

Laboratorium będzie zaliczane na podstawie ocen zbioru opracowanych indywidualnie i uruchomionych programów, przydzielonych przez prowadzącego do realizacji na poszczególnych zajęciach.

Informacje wstępne

Literatura

1. Harris S., Ross J., *Algorytmy. Od podstaw*, Wyd. Helion, Gliwice, 2006
2. Wirth N., *Algorytmy + struktury danych = programy*, WNT, Warszawa, 2004, wyd. VII
3. Knuth D., *Sztuka programowania, tom 1,2,3*, WNT, Warszawa, 2002
4. Banachowski L., Diks K., Rytter W., *Algorytmy i struktury danych*, WNT, Warszawa, 2006, wyd. 5
5. Cormen T.H., Leiserson Ch E, Rivest R.L., *Wprowadzenie do algorytmów*, WNT, Warszawa, 2007, wyd. 8
6. Aho A.V., Hopcroft J.E., Ullman J.D., *Algorytmy i struktury danych*, Helion, 2003,
7. Sedgewick R., *Algorytmy w Javie*
8. Bentley J., *Perły programowania*, WNT, Warszawa, 2001

Wykład 1. Iteracja i iteratory

Iteracja

Iteracja i rekurencja – dwie podstawowe koncepcje organizacji sterowania przetwarzaniem w programach.

Iteracja – wielokrotne powtarzanie tego samego bloku czynności.

Liczba powtórzeń może być określona jawnie (znana a priori),

```
np. i=0; s=0; do {s+=a[i]; i++;}  
    while (i<n);
```

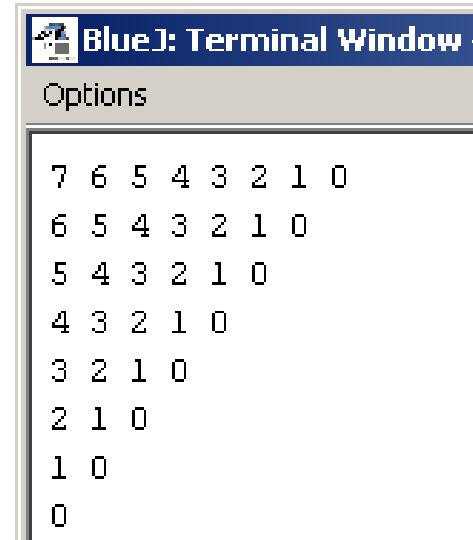
lub wynikać z wystąpienia podczas przetwarzania **określonej sytuacji** (spełnienia / nie spełnienia określonego warunku).

Rekurencja

Rekurencja – wywoływanie *metody* przez samą siebie (bezpośrednio lub pośrednio). Oznacza to realizację idei podziału problemu na podproblemy o tej samej naturze, lecz o mniejszych rozmiarach.

Przykład rekurencji:

```
void drukujCiagLiczb(int k) {  
    if (k==0) System.out.println(k);  
    else { System.out.print(k); drukujCiagLiczb(k-1); }  
}  
  
void drukujPiramide(int n) {  
    if (n==0) System.out.println(n);  
    else { drukujCiagLiczb(n); drukujPiramide(n-1); }  
}
```



```
BlueJ: Terminal Window  
Options  
7 6 5 4 3 2 1 0  
6 5 4 3 2 1 0  
5 4 3 2 1 0  
4 3 2 1 0  
3 2 1 0  
2 1 0  
1 0  
0
```


Iteracyjne przetwarzanie struktur danych - tablice

Przetwarzanie tablicy często sprowadza się do użycia **zmiennej indeksowanej**, której wartość zmienia się w pętli w jawnie określony sposób.

Np. dla następującej klasy:

```
public class Student {  
    int nrIndeksu;  
    double stypendium;  
    public Student(int nr, double kwota){nrIndeksu=nr; stypendium=kwota; }  
    public void zwiekszStypendium(double kwota){ stypendium+=kwota; }  
    public void wyswietlDane(){  
        System.out.printf("%6d  %8.2f\n",nrIndeksu,stypendium);  
    }  
}
```

Iteracyjne przetwarzanie struktur danych - tablice

Zwiększenie stypendium o stały dodatek może być zrealizowane w następujący sposób:

Iteracyjne przetwarzanie struktur danych - tablice

.....

// Załóżmy, że mamy następującą listę studentów:

```
Student [ ] s = new Student[5];
```

```
s[0]=new Student(1,500);
```

```
s[1]=new Student(2,400);
```

```
s[2]=new Student(3,0);
```

```
s[3]=new Student(4,500);
```

```
s[4]=new Student(5,700);
```

// Zwiększenie stypendium wszystkim studentom o kwotę **dodatek**:

```
for (int i = 0; i < s.length; i++) s[i].zwiększStypendium(50);
```

// Wyświetlenie listy stypendiów:

```
for (int i = 0; i < s.length; i++) s[i].wyswietlDane();
```

Iteracyjne przetwarzanie struktur danych - tablice

.....

// Załóżmy, że mamy następującą listę studentów:

```
Student [ ] s = new Student[5];
```

```
s[0]=new Student(1,500);
```

```
s[1]=new Student(2,400);
```

```
s[2]=new Student(3,0);
```

```
s[3]=new Student(4,500);
```

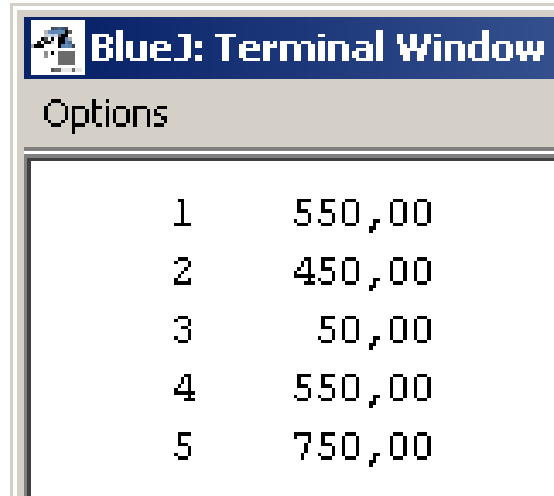
```
s[4]=new Student(5,700);
```

// Zwiększenie stypendium wszystkim studentom o kwotę **dodatek**:

```
for (int i = 0; i < s.length; i++) s[i].zwiększStypendium(50);
```

// Wyświetlenie listy stypendiów:

```
for (int i = 0; i < s.length; i++) s[i].wyswietlDane();
```



BlueJ: Terminal Window

Options

1	550,00
2	450,00
3	50,00
4	550,00
5	750,00

Iteracyjne przetwarzanie struktur danych - tablice

.....

// Załóżmy, że mamy następującą listę studentów:

```
Student [ ] s = new Student[5];
```

```
s[0]=new Student(1,500);
```

```
s[1]=new Student(2,400);
```

```
s[2]=new Student(3,0);
```

```
s[3]=new Student(4,500);
```

```
s[4]=new Student(5,700);
```

// Zwiększenie stypendium wszystkim studentom o kwotę **dodatek**:

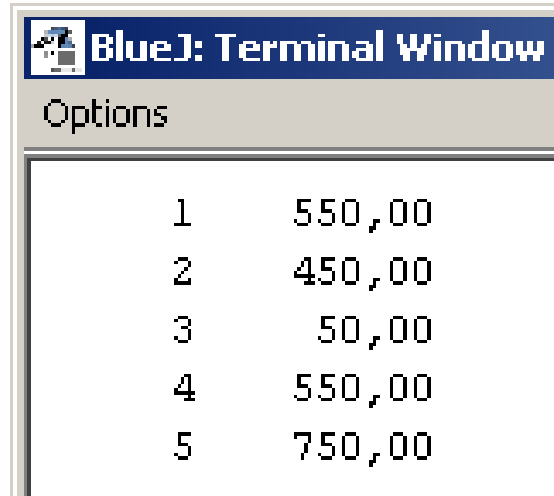
```
for (int i = 0; i < s.length; i++) s[i].zwiększStypendium(50);
```

// Wyświetlenie listy stypendiów:

```
for (int i = 0; i < s.length; i++) s[i].wyswietlDane();
```

.....

Takie przetwarzanie sekwencyjne obejmuje wszystkie kolejne elementy tablicy. Jak sprawić, by ograniczyć przetwarzanie tylko do wybranych elementów, np. **zwiększyć stypendium tylko tym studentom, którzy je mają?**



Options	
1	550,00
2	450,00
3	50,00
4	550,00
5	750,00

Jak organizować iteracje?

Uogólnienie problemu wyboru elementów iteracji:

- Jak poradzić sobie **w wielu różnych** iteracjach, w których wybór elementów dokonywany jest **w różny sposób** (według różnych kryteriów)?
- Jak ukryć szczegóły związane ze sposobem przechodzenia przez kolejne elementy przetwarzanego zbioru elementów?
- Jak poradzić sobie z przejściem na inną strukturę danych (np. z tablicy na listę, tabelę relacyjnej bazy danych, itp.)?

Takim mechanizmem wspomagającym jest **iterator (enumerator)**.

Iterator dostarcza **zestaw metod** zapewniających dostęp do danych, w postaci **interfejsu**.

Implementacja interfejsu musi być dostosowana do konkretnej struktury danych, zapewniając poprawność działania iteratora.

Iterator jest **konceptją**, a nie konkretną implementacją.

W swojej książce o wzorcach projektowych, Gamma i inni zaproponowali koncepcję iteratora, który jest bardziej uniwersalny od standardowego, wbudowanego w Javę.

Interfejs iteratora

```
// package iterators;
public interface Iterator // metody wymagające zaimplementowania
{ public void previous(); // powoduje przejście do poprzedniego elementu
  public void next();      // powoduje przejście do następnego elementu
  public void first();     // powoduje przejście do pierwszego elementu
  public void last();      // powoduje przejście do ostatniego elementu
                          // niezaimplementowanie powyższych metod
                          // wywołuje wyjątek UnsupportedOperationException
  public boolean isDone(); // true jeśli nie jest określony element bieżący,
                          // iterator jest w stanie "wyczerpany"
                          // false – jeśli jest określony element bieżący
  public Object current() throws IteratorOutOfBoundsException;
                          // udostępnia wartość bieżącego elementu
                          // jeśli bieżący element nie jest określony, występuje wyjątek
}
```

Iterator – przykłady wykorzystania

Pętla while

```
Iterator i;
```

```
// wariant "od początku do końca"
```

```
i.first();
```

```
while(!i.isDone()){
```

```
    Object object = i.current();
```

```
    ....
```

```
    i.next();
```

```
}
```

```
// wariant "od końca do początku"
```

```
i.last();
```

```
while(!i.isDone()){
```

```
    Object object = i.current();
```

```
    ....
```

```
    i.previous();
```

```
}
```


Iterator – przykłady wykorzystania

Pętla for

Iterator i;

// wariant "od początku do końca"

for (i.first(); !i.isDone(); i.next())

{ Object object = i.current();

....

}

// wariant "od końca do początku"

for (i.last(); !i.isDone(); i.previous())

{ Object object = i.current();

....

}

Iterator tablicowy

```
// package iterators;
```

```
public class IteratorTablicowy implements Iterator {  
    final Object [ ] tablica;  
    final int pierwszy;  
    final int ostatni;  
    int biezacy=-1;  
  
    public IteratorTablicowy(Object[] tab, int odElem, int liczbaElem)  
    { tablica=tab;  
      pierwszy=odElem;  
      ostatni=odElem+liczbaElem-1;  
    }  
  
    public IteratorTablicowy(Object[] tab)  
    { tablica=tab;  
      pierwszy=0;  
      ostatni=tablica.length-1;  
    }  
}
```

Iterator tablicowy

```
public void first()  
{ biezacy=pierwszy; }
```

```
public void last()  
{ biezacy=ostatni; }
```

```
public void next()  
{ ++biezacy; }
```

```
public void previous()  
{ --biezacy; }
```

```
public boolean isDone()  
{ return biezacy < pierwszy || biezacy > ostatni; }
```

```
public Object current()  
{ return tablica [ biezacy ]; }
```

```
}
```

Iterator tablicowy

```
public class GrupaStud
{
    Student [ ] s = new Student[5];
    public GrupaStud(){
        s[0]=new Student(1,500);
        s[1]=new Student(2,400);
        s[2]=new Student(3,0);
        s[3]=new Student(4,500);
        s[4]=new Student(5,700);
    }
    .....
}
```

Iterator tablicowy

```
public class GrupaStud
{
    public static void main(){
        Student [ ] s = new Student[5];
        s[0]=new Student(1,500);
        s[1]=new Student(2,400);
        s[2]=new Student(3,0);
        s[3]=new Student(4,500);
        s[4]=new Student(5,700);
        IteratorTablicowy itab = new IteratorTablicowy(s);
        // pełna lista studentów
        for (itab.first(); !itab.isDone(); itab.next()){
            Student st = (Student) itab.current();
            st.wyswietlDane();
        }
    }
}
```

Iterator tablicowy do przetwarzania odwrotnego

Do przetwarzania w odwrotnej kolejności można zbudować **iterator odwrotny** :

```
// package iterators;  
public class IteratorOdwrotny implements Iterator {  
    private final Iterator i;  
  
    public IteratorOdwrotny(Iterator iter) { i=iter; }  
    public void first() { i.last(); }  
    public void last() { i.first(); }  
    public void next() { i.previous(); }  
    public void previous() { i.next(); }  
    public boolean isDone() { return i.isDone(); }  
    public Object current() { return i.current(); }  
}
```

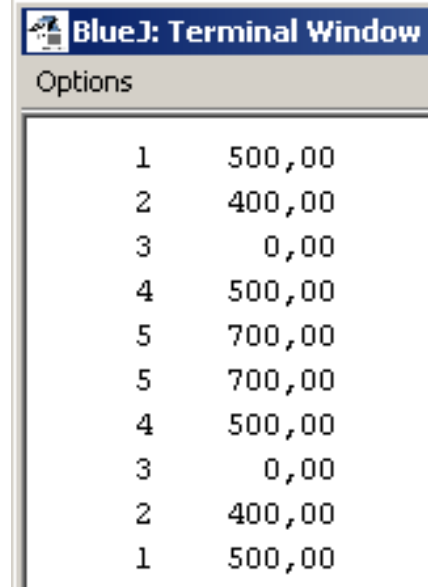
Dzięki wykorzystaniu iteratora odwrotnego nie trzeba zmieniać klasy, która go wykorzystuje - wystarczy przekazać jej albo iterator bazowy albo odwrotny.

Iterator tablicowy do przetwarzania odwrotnego

```
public class GrupaStud {  
    ....  
    public void wyswietlListe(Iterator itab){  
        for (itab.first(); !itab.isDone(); itab.next()){  
            Student st = (Student) itab.current();  
            st.wyswietlDane();  
        }  
    }  
    public static void main(){  
        ...  
        IteratorTablicowy itab = new IteratorTablicowy(s);  
        // pełna lista studentów  
        wyswietlListe(itab);  
        IteratorOdwrotny itabRev = new IteratorOdwrotny(itab);  
        // pełna lista studentów - w odwrotnej kolejności  
        wyswietlListe(itabRev);  
    }  
}
```

Iterator tablicowy do przetwarzania odwrotnego

```
public class GrupaStud {  
    ....  
    public void wyswietlListe(Iterator itab){  
        for (itab.first(); !itab.isDone(); itab.next()){  
            Student st = (Student) itab.current();  
            st.wyswietlDane();  
        }  
    }  
    public static void main(){  
        ...  
        IteratorTablicowy itab = new IteratorTablicowy(s);  
        // pełna lista studentów  
        wyswietlListe(itab);  
        IteratorOdwrotny itabRev = new IteratorOdwrotny(itab);  
        // pełna lista studentów - w odwrotnej kolejności  
        wyswietlListe(itabRev);  
    }  
}
```



A screenshot of a BlueJ terminal window titled "BlueJ: Terminal Window". It displays the output of a program, showing a list of student IDs and their corresponding fees. The output is as follows:

Options	
1	500,00
2	400,00
3	0,00
4	500,00
5	700,00
5	700,00
4	500,00
3	0,00
2	400,00
1	500,00

Iterator filtrujący

Do przetwarzania tylko wybranych (według określonego kryterium) elementów wygodnie jest użyć **iteratora filtrującego**:

```
// package iterators;
public class IteratorFiltrujacy implements Iterator {
    private final Iterator iterf;
    private final Predicate pred; // dla określenia efektu filtracji
    // Klasa Predicate zawiera tylko metodę logiczną accept() – badającą
    // spełnienie kryterium; należy ją zaimplementować w klasie definiującej
    // przetwarzane elementy.
    // package iterators;
    // public interface Predicate { public boolean accept(Object o); }
    public IteratorFiltrujacy(Iterator i, Predicate predykat)
    { iterf=i; pred=predykat; iterf.first() }

    public void filtrujDoPrzodu()
    { while (!iterf.isDone() && !pred.accept(iterf.current()))
        iterf.next(); }
    public void filtrujDoTyłu()
    { while( !iterf.isDone() && !pred.accept(iterf.current()))
        iterf.previous(); }
```

Iterator filtrujący

```
public void first()  
{ iterf.first(); filtrujDoPrzodu(); }
```

```
public void last()  
{ iterf.last(); filtrujDoTylu(); }
```

```
public void next()  
{ iterf.next(); filtrujDoPrzodu(); }
```

```
public void previous()  
{ iterf.previous(); filtrujDoTylu();}
```

```
public boolean isDone()  
{ return iterf.isDone(); }
```

```
public Object current()  
{ return iterf.current();}
```

```
}
```

Iterator filtrujący

Iterator filtrujący wykorzystuje **inny iterator** oraz **metodę klasyfikującą**: akceptującą lub odrzucającą elementy zwracane przez iterator. Metoda ta nosi nazwę **predykatora**.

W ten sposób iterator filtrujący ignoruje wszystkie te elementy, które nie spełniają warunków określonych przez predykator.