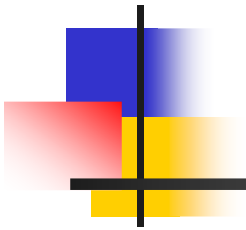
Universitatea “Lucian Blaga” din Sibiu,
Catedra de Calculatoare și Automatizări

Analiza şi proiectarea algoritmilor

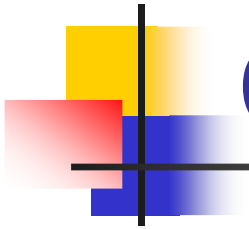


Curs, anul II – Calculatoare, Tehnologia informaţiei, Ingineria sistemelor multimedia

Puncte de credit: 5

Autor: Dr. Árpád GELLÉRT

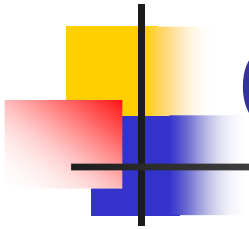
Web: http://webspace.ulbsibiu.ro/arpad.gellert
E-mail: arpad.gellert@ulbsibiu.ro



Cuprins (1)

Partea I – Limbajul Java

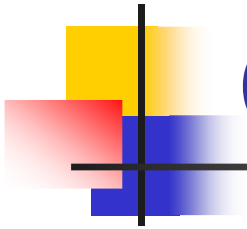
- Structura lexicală și instrucțiuni
- Variabile și constante
- Tablouri și matrici
- Tratarea excepțiilor
- Operații I/O
- Crearea claselor
- Interfețe grafice
- Fire de execuție
- Colecții



Cuprins (2)

Partea II – Analiza algoritmilor

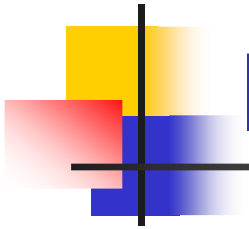
- Complexitate, notații asimptotice
- Recurențe
- Algoritmi de căutare și sortare
- Tabele de dispersie
- Arbori binari
- Heap-uri
- Grafuri



Cuprins (3)

Partea III – Proiectarea algoritmilor

- Divide et impera
- Greedy
- Programare dinamică
- Backtracking
- Algoritmi genetici
- Rețele neuronale

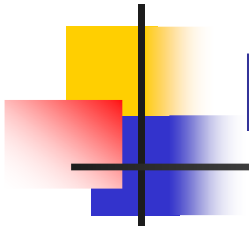


Nota finală

Nota laborator (NL) = 0,5*T + 0,5*C

Nota finală = 0,4*NL + 0,6*E

- T = Test susținut la laborator din aplicațiile propuse la laborator în cadrul capitolului *Limbajul Java*;
- C = Colocviu de laborator din aplicațiile propuse la laborator în cadrul capitolelor *Analiza algoritmilor* respectiv *Proiectarea algoritmilor*;
- E = Examen susținut din capitolele *Analiza algoritmilor* și *Proiectarea algoritmilor*.



Bibliografie de bază

[Knu00] Knuth D., *Arta programării calculatoarelor*, Teora, 2000.

[Cor00] Cormen T., Leiserson C., Rivest R., *Introducere în algoritmi*, Agora, 2000.

[Giu04] Giumale C., *Introducere în analiza algoritmilor*, Polirom, 2004.

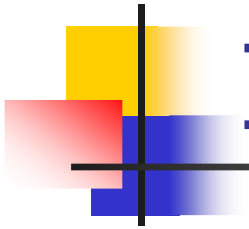
[Wai01] Waite M., Lafore R., *Structuri de date și algoritmi în Java*, Teora, 2001.

[Log07] Logofătu D., *Algoritmi fundamentali în Java*, Polirom, 2007.

[Tan07] Tanasă Ș., Andrei Ș., Olaru C., *Java de la 0 la expert*, Polirom, 2007.

Introducere (1)

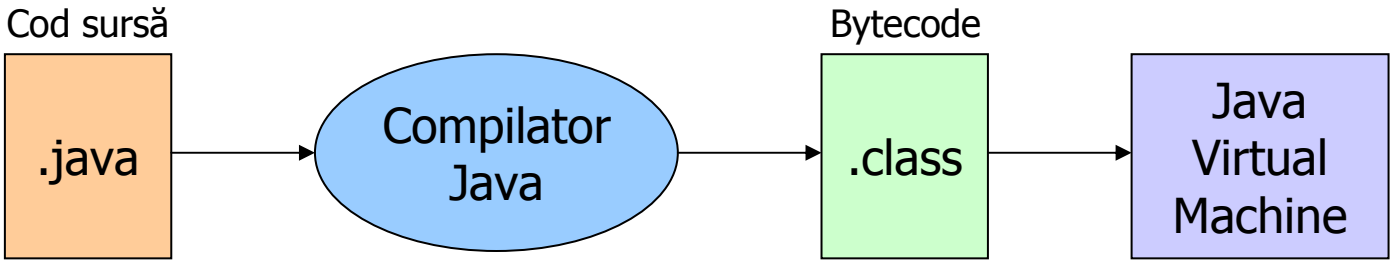
- **Geneza cuvântului algoritm:** provine din latinizarea numelui savantului Al-Khwarizmi (algoritmi) – matematician, geograf, astronom şi astrolog persan (780-850), considerat și părintele algebrei moderne.
- **Algoritmul** este o secvenţă de operaţii care transformă mulţimea datelor de intrare în datele de ieşire.
- **Proiectarea algoritmului** constă în două etape:
  - Descrierea algoritmului printr-un pseudolimbaj (schemă logică, pseudocod);
  - Demonstrarea corectitudinii rezultatelor în raport cu datele de intrare.
- **Analiza algoritmului** se referă la evaluarea performanţelor acestuia (timp de execuţie, spaţiu de memorie).



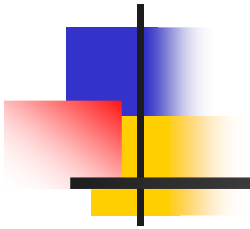
Introducere (2)

- **Imlementarea algoritmilor** se va face în limbajul Java. Medii de dezvoltare:
  - Java Development Kit (JDK) – http://www.oracle.com
  - NetBeans – https://netbeans.org
  - IntelliJ – https://www.jetbrains.com
  - Eclipse – http://www.eclipse.org
  - Android Studio – https://developer.android.com/studio/index.html
- **Principalele caracteristici** ale limbajului Java:
  - Simplitate – elimină supraîncărcarea operatorilor, moştenirea multiplă, etc.;
  - Complet orientat pe obiecte;
  - Portabilitate – Java este un limbaj compilat şi interpretat, fiind astfel independent de platforma de lucru (Write Once, Run Enywhere);
  - Oferă posibilitatea, prin JNI (Java Native Interface), de a implementa aplicații mixte (ex. Java/C++), prețul folosirii codului nativ fiind însă dependența de platformă [Gor98];
  - Permite programarea cu fire de execuţie.

Introducere (3)

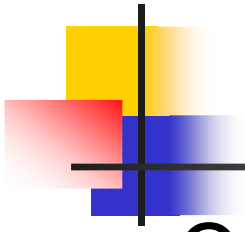


- În urma compilării codului sursă, se obţine codul de octeţi (bytecode) care este apoi interpretat de maşina virtuală Java (JVM). Astfel, aplicaţia poate fi rulată pe orice platformă care foloseşte mediul de execuţie Java.
- Pe lângă aplicaţii obişnuite, pot fi implementate aplicaţii web (applet), aplicaţii server (servlet, JavaServer Pages, etc.), aplicaţii în reţea, telefonie mobilă (midlet), aplicaţii distribuite RMI (Remote Method Invocation), etc.
- Compilarea programelor (javac.exe):
  *javac *.java*
- Rularea programelor (java.exe):
  *java **



Partea I

Limbajul Java

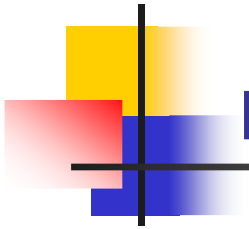


Limbajul Java

O aplicaţie simplă:

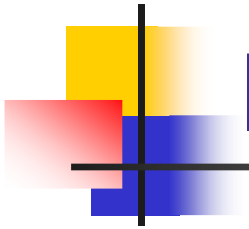
```
class HelloWorld{
    public static void main(String args[]){
        System.out.println("Hello world");
    }
}
```

- Aplicaţia HelloWorld afişează la consolă mesajul “Hello world”.
- În orice aplicaţie trebuie să existe o clasă primară, care conţine metoda *main* de la care începe execuţia în momentul rulării. Parametrul *args* al metodei *main* este un tablou de şiruri de caractere şi reprezintă argumentele din linie de comandă.
- Fişierul care conţine clasa primară trebuie să aibă numele clasei (case sensitive!).
- Java permite o singură clasă publică într-un fișier.
- Compilarea programului: *javac HelloWorld.java*
- Rularea programului: *java HelloWorld*



Limbajul Java – convenţii de scriere a codului [Web01]

- Principalele segmentele ale codului Java apar în următoarea ordine:
  - Declaraţia de pachet (*package*);
  - Directivele de *import*;
  - Declaraţia de clasă sau interfaţă;
  - Declaraţiile variabilelor membru ale clasei (statice) sau ale instanţei;
  - Constructori şi metode;
- Convenţii de nume
  - Numele de clase, implicit de constructori, încep cu majusculă (ex.: NumeClasa);
  - Numele de metode încep cu literă mică (ex.: numeMetoda);
  - Numele de variabile şi obiecte încep cu literă mică (ex.: numeObiect);
  - Numele constantelor se scriu cu majuscule, cuvintele fiind despărţite prin "_" (ex.: NUME_CONSTANTA);
  - Numele trebuie să fie cât mai sugestive.



Limbajul Java – Structura lexicală

Structura lexicală a limbajului Java este foarte asemănătoare cu cea a limbajului C++.

- Comentariile se fac ca în C++.
- Operatorii sunt în mare parte la fel ca în C++.
- Instrucțiunile sunt și ele foarte asemănătoare cu cele din C++. O diferență importantă constă în faptul că expresiile care apar în instrucțiunile condiționale (*if*, *for*, *while*, *do*) sunt strict de tip *boolean*, în timp ce în C++ pot fi de tip *int*.
- Tipurile de date se clasifică în două categorii:
  - Tipuri de date primitive: *byte* (1 octet), *short* (2 octeți), *int* (4 octeți), *long* (8 octeți), *char* (2 octeți – unicode), *float* (4 octeți), *double* (8 octeți), *boolean* (1 bit);
  - Tipuri de date referință: clase, interfețe, tablouri. Există și variantele referință (clase wrapper) ale tipurilor de date primitive: *Byte*, *Short*, *Integer*, *Long*, *Character*, *Float*, *Double*, *Boolean*. Variantele referință ale tipurilor de date primitive sunt necesare pentru că în Java nu există operatorul & pentru definiția unei referințe. Pentru șiruri de caractere, pe lângă *char[]*, există și tipul referință String.

Limbajul Java – variabile

- Variabilele primitive se pot crea prin declarație. Exemple:
  - int i = 10;
  - float f = 3.14f;
  - double d = 3.14;
- Variabilele referință se pot crea doar cu operatorul *new* (care returnează o referință). Variabilele de tip String și începând cu JDK 1.5, cele de tip wrapper care se pot crea și prin declarație (făr *new*). Exemple:
  - Integer i = new Integer(10);
  - Integer j = 20;
  - String h = new String("Hello");
  - String w = "world";
- În Java nu este admisă supraîncărcarea operatorilor, excepție fiind doar operatorul de adunare pentru clasa *String*, care permite concatenarea șirurilor. Exemplu:

```
String h = "Hello";
String w = "world";
String s = h + " " + w;
```

- Concatenarea unui *String* cu o variabilă primitivă determină conversia implicită a acesteia în *String*. Exemplu:

```
String t = "Two";
int i = 2;
boolean b = true;
String s = t + " = " + i  + " is " + b;     //Rezultat: "Two = 2 is true"
```

Limbajul Java – constante

- Pentru crearea unei constante, declarația de variabilă trebuie precedată de cuvântul cheie *final*.
- Nu este permisă modificarea valorii unei constante.
- O clasă declarată *final* nu poate fi derivată (v. moștenirea).
- Exemple

```
public class Main {
    public static void main(String[] args) {
        final String APA = "Analiza si proiectarea algoritmilor";
        final int TEN = 10;
        TEN++;                                                    //Eroare!
        APA = "Analiza, proiectarea si implementarea algoritmilor";  //Eroare!
        int eleven = TEN + 1;
    }
}
```

Limbajul Java – clasa String (1)

Principalele operații:

- **substring** cu parametrii *index0* și *indexf* – returneaza subșirul care începe la *index0* și se termină la *indexf-1*, lungimea fiind *indexf-index0*. Exemplu:

  String str = "Hello world";
  String substr = str.substring(0, 5); //Rezultat: "Hello"

- **charAt** – returnează caracterul de pe o anumită poziție din șir. Exemplu:

  String str = "Hello world";
  char c = str.charAt(6); //Rezultat: 'w'

- **length** – retunează lungimea șirului în număr de caractere. Exemplu:

  String str = "Hello world";
  int len = str.length(); //Rezultat: 11

- **equals / equalsIgnoreCase** – permite comparația unui șir cu alt șir. Returnează true dacă cele două șiruri sunt egale, respectiv false dacă ele nu sunt egale. Exemplu:

  String hello = "Hello";
  if(hello.equals("Hello")) //Rezultat: true
  System.out.println("equal"); //Rezultat: "equal"

Limbajul Java – clasa String (2)

- **compareTo / compareToIgnoreCase** – compară două șiruri. Returnează valoarea 0 dacă cele două șiruri sunt lexicografic egale, o valoare negativă dacă parametrul este un șir lexicografic mai mare decât șirul curent și o valoare pozitivă dacă parametrul este un șir lexicografic mai mic decât șirul curent. Exemplu:

```
String hello = "Hello";
if(hello.compareTo("Hello") == 0)          //Rezultat: 0
        System.out.println("equal");       //Rezultat: "equal"
```

- **trim** – elimină eventualele spații de la începutul și sfârșitul șirului. Exemplu:

```
String hello = " Hello ";
hello = hello.trim();                      //hello="Hello"
```

- **toLowerCase / toUpperCase** – transformă toate literele din șir în litere mici / mari. Exemplu:

```
String str = "Hello world";
str = str.toUpperCase();                   //Rezultat: "HELLO WORLD"
```

- **split** – separarea unui șir pe baza unui delimitator. Exemplu:

```
String str = "Hello world";
String t[] = str.split(" ");               //Rezultat: {"Hello", "world"};
```

Limbajul Java – clasa StringTokenizer

- Permite separarea unui șir de caractere în simboluri;
- Se instanțiază un obiect *StringTokenizer*, specificând șirul de caractere și setul de delimitatori;
- Următoarea secvență de program afișează pe ecran simbolurile (cuvintele) șirului delimitate prin caracterele spațiu și virgulă:

```
String str = "Analiza, proiectarea si implementarea algoritmilor";
StringTokenizer st = new StringTokenizer(str, " ,");
while(st.hasMoreTokens())
    System.out.println(st.nextToken());
```

- Setul de delimitatori poate fi specificat și prin funcția *nextToken*:

```
while(st.hasMoreTokens())
    System.out.println(st.nextToken(" ,"));
```

Limbajul Java – tablouri

- Posibilități de declarare a tablourilor:
  - int[] fibo = new int[10];
  - int fibo[] = new int[10];
  - int n = 10;
    int fibo[] = new int[n];
  - int fibo[] = {0, 1, 1, 2, 3, 5, 8, 13, 21, 34};
- Tablou cu elemente primitive vs. referință:
  - int pTab[] = new int[5]; //tablou (referinta!) cu 5 elem. primitive

    for(int i=0; i<pTab.length; i++)

        pTab[i] = i+1;
  - Integer rTab[] = new Integer[5]; //tablou (referinta!) cu 5 elem. referinta

    for(int i=0; i<rTab.length; i++)

        rTab[i] = new Integer(i+1);
- Afișarea elementelor unui tablou:
  - for(int i=0; i<fibo.length; i++)

        System.out.println(fibo[i]);
  - for(int i : fibo) //incepand cu JDK 1.5

        System.out.println(i);

Limbajul Java – clasa Arrays

Principalele operații:

- **fill** – permite umplerea tabloului sau a unei zone din tablou cu o anumită valoare:

```
int tab[] = new int[5];
Arrays.fill(tab, 7);                    //Rezultat: tab={7,7,7,7,7}
Arrays.fill(tab, 1, 3, 8);              //Rezultat: tab={7,8,8,7,7}
```

- **equals** – compară două tablouri returnând true dacă au toate elementele egale:

```
int a[] = {1,2,3,4};
int b[] = {1,2,3,4};
int c[] = {1,2,4,8};
System.out.println(Arrays.equals(a, b) ? "a==b" : "a!=b");   //Rezultat: "a==b"
System.out.println(Arrays.equals(a, c) ? "a==c" : "a!=c");   //Rezultat: "a!=c"
```

- **sort** – sortează elementele tabloului în ordine crescătoare folosind algoritmul Quicksort pentru tipuri primitive respectiv Mergesort pentru tipuri referință:

```
int pTab[] = {9,6,2,1,5,3};
Arrays.sort(pTab,1,4);                  //Rezultat: tab={9,1,2,6,5,3}
Arrays.sort(pTab);                      //Rezultat: tab={1,2,3,5,6,9}

Integer rTab[] = new Integer[5];
for(int i=0, k=rTab.length; i<rTab.length; i++, k--)   //Rezultat : tab={5,4,3,2,1}
    rTab[i] = new Integer(k);
Arrays.sort(rTab);                      //Rezultat : tab={1,2,3,4,5}
```

- **binarySearch** – returnează poziția elementului căutat în tablou sau o valoare negativă dacă valoarea căutată nu este găsită. Algoritmul folosit este căutarea binară, de aceea tabloul trebuie sortat înainte de apelul acestei metode. Exemplu:

```
int tab[] = {5,4,1,7,3};                //tablou nesortat
int v = Arrays.binarySearch(tab, 4);    //Rezultat greșit: -4
Arrays.sort(tab);                       //Rezultat: tab={1,3,4,5,7}
v = Arrays.binarySearch(tab, 4);        //Rezultat corect: 2
```

Limbajul Java – matrici

- Posibilități de declarare a matricilor:
  - int m[][] = new int[3][4];
  - int nRow=3, nCol=4;
    int m[][] = new int[nRow][nCol];
  - int m[][] = new int[3][];
    for(int i=0; i<3; i++)
        m[i] = new int[4];
  - int m[][] = {{1,2,3,4},{5,6,7,8},{9,10,11,12}};

- Tablou multidimensional cu număr variabil de coloane. Exemplu de alocare, inițializare și afișare:

```
int m[][] = new int[3][];
for(int i=0, k=1; i<3; i++){
    m[i] = new int[4+i];
    for(int j=0; j<4+i; j++, k++)
        m[i][j] = k;
}
for(int i=0; i<3; i++){                    //Rezultat afișat:
    for(int j=0; j<4+i; j++)               //1   2   3   4
        System.out.print(m[i][j]+"\t");    //5   6   7   8   9
    System.out.println();                  //10  11  12  13  14  15
}
```

- O matrice *m* are *m.length* linii și *m[0].length* coloane.

Limbajul Java – conversii (1)

- Conversia variabilelor primitive
  - Conversie implicită. Exemple:

```
int a = 3, b = 2;
double c = a/b;          //Conversie din int in double. Rezultat: 1.0
int d = 5;
float f = d;             //Conversie din int in float. Rezultat: 5.0
```

  - Conversie explicită (casting). Exemplu:

```
double pi = 3.14;
int i = (int)pi;         //Conversie din double in int. Rezultat: 3
```

- Conversia variabilelor primitive în variabile referință. Exemple:

```
int x = 10;
Integer y = new Integer(x);        //Conversie din int in Integer
float m = 3.14f;
Float n = new Float(m);            //Conversie din float in Float
String str10 = String.valueOf(x);  //Conversie din int in String
```

Limbajul Java – conversii (2)

- Conversia variabilelor referință în variabile primitive. Exemple:

```
Integer zece = new Integer(10);
int z = zece.intValue();                    //Conversie din Integer in int
Double p = new Double(3.14);
double q = p.doubleValue();        //Conversie din Double in double
String str20 = "20";
int d = Integer.parseInt(str20);           //Conversie din String in int
String strPi = "3.14";
double pi = Double.parseDouble(strPi); //Conversie String - double
```

- Autoboxing și inboxing (începând cu JDK 1.5). Exemple:

```
int z = 10;
Integer zece = z;                           //autoboxing
Double pi = new Double(3.14);
double p = pi;                              //unboxing
```

Limbajul Java – comparații (1)

- Variabile primitive (comparație)

```
int a = 5;
int b = 5;
System.out.println(a==b ? "a==b" : "a!=b");          //Rezultat : "a==b"
```

- Variabile referință. Dacă cel puțin una din variabilele referință s-a creat cu new, operatorii "==", "!=", "<", "<=", ">" și ">=" nu se pot folosi pentru comparații, fiind necesar apelul metodelor corespunzătoare existente în clasele wrapper.

  Exemple:

  - Comparație greșită (se compară adrese!):

```
Integer a = new Integer(5);
Integer b = new Integer(5);
System.out.println(a==b ? "a==b" : "a!=b");          //Rezultat: "a!=b"
```

  - Comparaţie greşită (se compară o adresă cu o valoare!):

```
Integer a = new Integer(5);
Integer b = 5;
System.out.println(a==b ? "a==b" : "a!=b");          //"a!=b"
```

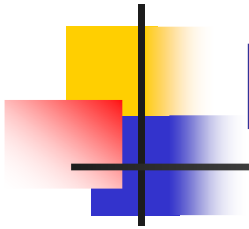
Limbajul Java – comparații (2)

- Comparaţie corectă (se compară valori!):
```
Integer a = 5;
Integer b = 5;
System.out.println(a==b ? "a==b" : "a!=b");                    //"a==b"
```

- Comparație corectă (se compară valori!):
```
Integer a = new Integer(5);
Integer b = new Integer(5);
System.out.println(a.equals(b) ? "a==b" : "a!=b");             //Rezultat : "a==b"
```

- Comparație corectă (se compară valori!):
```
Integer a = new Integer(5);
Integer b = new Integer(5);
System.out.println(a.compareTo(b)==0 ? "a==b" : "a!=b");    //Rezultat: "a==b"
```

- Comparație corectă (se compară valori!):
```
Integer a = new Integer(5);
Integer b = new Integer(5);
System.out.println(a.intValue()==b.intValue() ? "a==b" : "a!=b"); //Rezultat: "a==b"
```



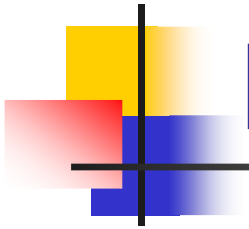
Limbajul Java – clasa File (1)

Clasa *File* oferă informații despre fișiere și directoare și permite redenumirea respectiv ștergerea acestora. Principalele operații:

- **length** – returnează dimensiunea în octeți a fișierului;
- **getName** – returnează numele fișierului;
- **getPath / getAbsolutePath / getCanonicalPath** – returnează calea fișierului;
- **getAbsoluteFile / getCanonicalFile** – returnează un obiect File aferent fișierului;
- **isFile / isDirectory** – returnează *true* dacă e fișier / director;
- **canRead / canWrite** – returnează *true* dacă aplicația poate citi / modifica fișierul;
- **exists** – verifică dacă fișierul există;
- **delete** – șterge fișierul sau directorul;
- **deleteOnExit** – șterge fișierul sau directorul la închiderea mașinii virtuale;
- **list** – apelată pentru un director returnează un tablou de tip String cu toate fișierele și subdirectoarele din acel director. Exemplu:

```
File file = new File("e:\\algoritmi_curs");
String str[] = file.list();
for(int i=0; i<str.length; i++)
    System.out.println(str[i]);
```

- **listFile** – apelată pentru un director returnează un tablou de tip File cu toate fișierele și subdirectoarele din acel director;
- **listRoots** – returnează un tablou de tip File reprezentând rădăcinile sistemelor de fișiere disponibile în sistem (ex.: "C:\\", "D:\\", "E:\\");



Limbajul Java – clasa File (2)

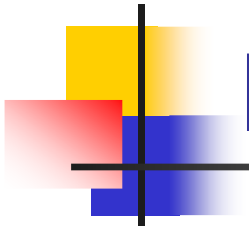
- **mkdir** – creează un director nou, returnând *true* în caz de succes. Exemplu:

```
File file = new File("algoritmi");
System.out.println(file.mkdir());
```

- **mkdirs** – creează un director nou, inclusiv directoarele inexistente care apar în cale
- **renameTo** – permite redenumirea fișierelor, returnând *true* în caz de succes. Exemplu:

```
File file = new File("Algoritmi.pdf");
System.out.println(file.renameTo(new File("algoritmi_curs.pdf")));
```

- **separator / separatorChar** – returnează un String / char reprezentând separatorul dependent de sistem: '/' în UNIX respectiv '\' în Win32.



Limbajul Java – clasa System

Principalele operații

- **currentTimeMillis** – returnează un *long* reprezentând timpul în milisecunde. Poate fi folosit ca parametru în constructorul clasei *Date* pentru obținerea datei și orei în formatul dorit.
- **exit(0)** – provoacă ieșirea din aplicație.
- **gc** – rulează mecanismul de *garbage collector* pentru eliberarea zonelor de memorie ocupate de obiecte neutilizate.
- **getProperties** – returnează un obiect *Properties* din care se pot extrage diferite informații referitoare la sistemul de operare.

Limbajul Java – clasa Math

Câteva exemple

```
public class Main {
    public static void main(String[] args) {
        double pi = Math.PI;
        System.out.println(pi);                    //3.141592653589793
        System.out.println(Math.round(pi));        //3 - rotunjire la cel mai apropiat intreg
        System.out.println(Math.floor(pi));        //3.0 - rotunjire in jos
        System.out.println(Math.ceil(pi));         //4.0 - rotunjire in sus
        System.out.println(Math.pow(2, 3));        //8.0 - ridicare la putere
        System.out.println(Math.sqrt(9));          //3.0 - radical
        System.out.println(Math.exp(1));           //2.7182818284590455 - valoarea exponentiala
        double e = Math.E;
        System.out.println(e);                                  //2.718281828459045
        System.out.println(Math.min(2, 8));        //2 - minimul
        System.out.println(Math.max(2, 8));        //8 - maximul
        System.out.println(Math.abs(-5));                       //5 - valoarea absoluta
        System.out.println(Math.log(e));           //1.0 - logaritmul natural
        System.out.println(Math.random());         //valoare aleatoare subunitara
        System.out.println(Math.sin(pi/2));        //1.0 - sinusul, exista si celelalte functii
        System.out.println(Math.toDegrees(pi/2));  //90.0 - conversie in grade
        System.out.println(Math.toRadians(180));   //3.141592653589793 - conv. in rad.
    }
}
```

Aplicații propuse

1. Căutarea minimului și a maximului într-un tablou de *n* elemente;
2. Generarea și afișarea unui tablou cu numerele lui Fibonacci.
3. Căutarea minimului și a maximului într-o matrice;
4. Adunarea a două matrici;
5. Înmulțirea a două matrici.
6. Să se scrie un program care, prin intermediul clasei *File*, să permită navigarea în structura de directoare din sistem.

Limbajul Java – tratarea excepțiilor (1)

- Excepțiile sunt erorile care apar în timpul execuției programelor.
- O instrucțiune sau o secvență de instrucțiuni trebuie inclusă într-un bloc *try* în vederea tratării eventualelor excepții pe care le poate genera. Blocul *try* este urmat de unul sau mai multe blocuri *catch* prin care sunt detectate diferitele excepții. În blocul opțional *finally* se introduc zonele de cod care trebuie executate indiferent că apare sau nu o excepție. Forma generală:

```
try{
    //instructiuni care pot genera Exceptia1 si Exceptia2
}
catch(Exceptia1 e1){
    //instructiunile care se executa daca apare Exceptia1
}
catch(Exceptia2 e2){
    //instructiunile care se executa daca apare Exceptia2
}
finally{
    //instructiunile care se executa indiferent ca apare sau nu o exceptie
}
```

- Dacă nu apar excepții, blocul *try* execută toate instrucțiunile. Dacă o instrucțiune din blocul *try* generează o excepție, se caută blocul *catch* corespunzător, trecând peste restul instrucțiunilor din *try*. Dacă există un *catch* corespunzător, se execută instrucțiunile din acel bloc *catch*. Dacă nu este găsit un bloc *catch* corespunzător, excepția este transmisă mai departe în ierarhia apelantă. Dacă excepția ajunge în vârful ierarhiei fără să fie tratată, programul afișează un mesaj de eroare și se întrerupe.

Limbajul Java – tratarea excepțiilor (2)

- Este posibilă ignorarea excepțiilor și transmiterea lor mai sus prin *throws* în ierarhia apelantă.
- În exemplul următor, dacă în metoda apare *Exceptia1* sau *Exceptia2*, ea este ignorată, transmisă mai departe și tratată în metoda apelantă *main* (ca în exemplul precedent):

```
public void metoda() throws Exceptia1, Exceptia2 {
    //instructuni care pot genera Exceptia1 si Exceptia2
}
public static void main(String[] args) {
    try{
        metoda();
    }
    catch(Exceptia1 e1){
        //instructiunile care se executa daca apare Exceptia1
    }
    catch(Exceptia2 e2){
        //instructiunile care se executa daca apare Exceptia2
    }
    finally{
        //instructiunile care se executa indiferent ca apare sau nu o exceptie
    }
}
```

Limbajul Java – tratarea excepțiilor (3)

Exemplul 1

- Implementați și apoi rulați următoarea secvență de cod:

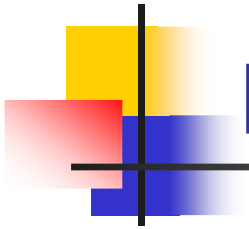
```
String str = "I'm a String!";
int i = Integer.parseInt(str);
System.out.println("Program execution finished...");
```

Conversia din *String* în *int* determină eșuarea execuției programului datorită conținutului nenumeric al variabilei *str*. Astfel nu se mai ajunge la afișarea mesajului "Program execution finished...". Excepția semnalată este *NumberFormatException*.

- Tratarea excepției:

```
String str = "I'm a String!";
try {
    int i = Integer.parseInt(str);
}
catch (NumberFormatException nfe) {
    System.out.println("Conversia nu a fost posibila!");
}
```

Excepția fiind tratată, execuția programului nu va mai eșua.



Limbajul Java – tratarea excepțiilor (4)

Exemplul 2

- Implementați și apoi rulați următoarea secvență de cod:

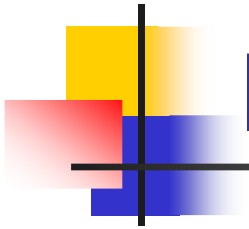
```
int tab[] = new int[10];
tab[10] = 10;
```

Accesarea unui element inexistent, al 11-lea element, nealocat, determină o excepție ArrayIndexOutOfBoundsException.

- Tratarea excepției:

```
int tab[] = new int[10];
try {
    tab[10] = 10;
}
catch (ArrayIndexOutOfBoundsException aioobe) {
    System.out.println("Ati accesat un element inexistent!");
}
```

- Evident, excepția nu apare dacă indexul folosit pentru accesarea tabloului este între 0 și 9.



Limbajul Java – operații I/O (1)

Citirea șirurilor de caractere de la tastatură

- Fluxuri de caractere:

```
BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
String str = null;
try {
    str = br.readLine();
}
catch (IOException ioe) {
    ioe.printStackTrace();
}
```

- Fluxuri de octeți:

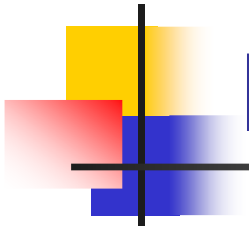
```
DataInputStream dis = new DataInputStream(System.in);
String str = null;
try {
    str = dis.readLine();
}
catch (IOException ioe) {
    ioe.printStackTrace();
}
```

Limbajul Java – operații I/O (2)

Citirea valorilor de la tastatură

- Clasa *DataInputStream* pe lângă metoda *readLine* (folosită pe pagina anterioară pentru introducerea șirurilor de caractere) dispune și de funcții pentru citirea valorilor (*readInt*, *readDouble*, etc.). Dar aceste metode sunt funcționale doar pentru valori scrise prin interfața DataOutput (*writeInt*, *writeDouble*, etc.).
- Citirea valorilor poate fi efectuată prin citire de șiruri de caractere urmată de conversia acestora:

```
DataInputStream dis = new DataInputStream(System.in);
String str = null;
try {
    str = dis.readLine();
}
catch (IOException ioe) {
    ioe.printStackTrace();
}
int i = Integer.parseInt(str);
```



Limbajul Java – operații I/O (3)

Citirea din fișiere

Următoarea secvență de cod afișează pe ecran toate liniile citite din fișierul *input.txt*. Citirea se face prin aceeași clasă DataInputStream care în loc de un InputStream va primi ca parametru un FileInputStream:

```
FileInputStream fis = null;
try{
    fis = new FileInputStream("input.txt");
}
catch(FileNotFoundException fnfe){
    fnfe.printStackTrace();
}
DataInputStream dis = new DataInputStream(fis);
String str = null;
try{
    while((str = dis.readLine()) != null)
        System.out.println(str);
    dis.close();
    System.in.read();
}
catch(IOException ioe){
    ioe.printStackTrace();
}
```

Limbajul Java – operații I/O (4)

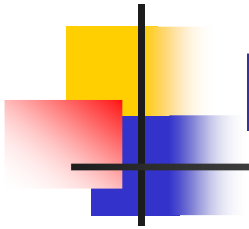
Scrierea în fișiere

- Următoarea secvență de program scrie în fișierul *data.txt* întregul 10 și valoarea float 3.14

```
try{
    FileOutputStream fos = new FileOutputStream("data.txt");
    DataOutputStream dos = new DataOutputStream(fos);
    dos.writeInt(10);
    dos.writeFloat(3.14f);
    dos.close();
}
catch(IOException ioe){
    ioe.printStackTrace();
}
```

- Fiind scrise prin metodele *writeInt* și *writeFloat*, valorile pot fi citite din fișier folosind metodele *readInt* respectiv *readFloat* ale clasei DataInputStream

```
try{
    FileInputStream fis = new FileInputStream("data.txt");
    DataInputStream dis = new DataInputStream(fis);
    System.out.println(dis.readInt());
    System.out.println(dis.readFloat());
    dis.close();
}
catch(IOException ioe){
    ioe.printStackTrace();
}
```



Limbajul Java – operații I/O (5)

Clasa Scanner

- Citire de la tastatură (începând cu JDK 1.5):

```
Scanner sc = new Scanner(System.in);
String str = sc.nextLine();
int n = sc.nextInt();
double d = sc.nextDouble();
System.out.println(str + " " + n + " " + d);
sc.close();
```

- Citire din FileInputStream (începând cu JDK 1.5):

```
FileInputStream f = null;
try {
    f = new FileInputStream("data.txt");
} catch (FileNotFoundException ex) {
    ex.printStackTrace();
}
Scanner sc = new Scanner(f);
String str = sc.nextLine();
int n = sc.nextInt();
double d = sc.nextDouble();
System.out.println(str + " " + n + " " + d);
sc.close();
```

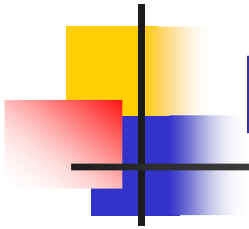
Limbajul Java – operații I/O (6)

- Citire din File (începând cu JDK 1.5):

```
File f = new File("data.txt");
Scanner sc = null;
try {
   sc = new Scanner(f);
} catch (FileNotFoundException ex) {
   ex.printStackTrace();
}
String str = sc.nextLine();
int n = sc.nextInt();
double d = sc.nextDouble();
System.out.println(str + " " + n + " " + d);
sc.close();
```

- Citirea unui fișier linie cu linie (începând cu JDK 1.5): :

```
FileInputStream f = null;
try {
   f = new FileInputStream("input.txt");
} catch (FileNotFoundException ex) {
   ex.printStackTrace();
}
Scanner sc = new Scanner(f);
while(sc.hasNext())
   System.out.println(sc.nextLine());
sc.close();
```

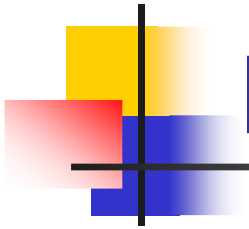


Limbajul Java – operații I/O (7)

Arhivare ZIP

Arhivarea unui fișier constă în citirea datelor din acel fișier prin *FileInputStream* și scrierea lor în arhivă prin *ZipOutputStream*. Pentru viteză mai mare, transferul datelor se face printr-un tablou de tip *byte*:

```
String fisier = "Algoritmi.pdf";              //fisierul care se arhiveaza
String zip = "Algoritmi.zip";                 //numele arhivei
byte buffer[] = new byte[1024];
try{
    FileInputStream fileIn = new FileInputStream(fisier);
    FileOutputStream f = new FileOutputStream(zip);
    ZipOutputStream zipOut = new ZipOutputStream(f);
    zipOut.putNextEntry(new ZipEntry(fisier));
    int nBytes;
    while((nBytes = fileIn.read(buffer, 0, 1024)) != -1)
        zipOut.write(buffer, 0, nBytes);
    zipOut.close();
    fileIn.close();
    f.close();
}
catch(ZipException ze){
    System.out.println(ze.toString());
}
catch(FileNotFoundException fnfe){
    fnfe.printStackTrace();
}
catch(IOException ioe){
    ioe.printStackTrace();
}
```

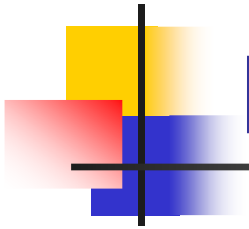


Limbajul Java – operații I/O (8)

Dezarhivare ZIP

Dezarhivarea unui fișier constă în citirea datelor din arhivă prin *ZipInputStream* și scrierea lor prin *FileOutputStream* în fișierul destinație. Pentru eficiență, transferul datelor se face printr-un tablou de tip *byte*:

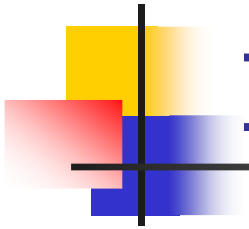
```
String fisier = "Algoritmi.pdf";              //numele fisierului destinatie
String zip = "Algoritmi.zip";                 //numele arhivei
byte buffer[] = new byte[1024];
try{
    FileInputStream fileIn = new FileInputStream(zip);
    FileOutputStream fileO = new FileOutputStream(fisier);
    ZipInputStream zipIn = new ZipInputStream(fileIn);
    zipIn.getNextEntry();
    int nBytes;
    while((nBytes = zipIn.read(buffer, 0, 1024)) != -1)
        fileO.write(buffer, 0, nBytes);
    fileIn.close();
    fileO.close();
}
catch(ZipException ze){
    System.out.println(ze.toString());
}
catch(IOException ioe){
    ioe.printStackTrace();
}
```



Limbajul Java – operații I/O (9)

Aplicații propuse

1. Să se scrie un program care cere introducerea numelui utilizatorului și afișează pe ecran mesajul *Hello* urmat de numele introdus (afișare-citire-afișare).
2. Citirea unui întreg până la introducerea unei valori valide (tratând excepția *NumberFormatException*).
3. Să se implementeze un program care citește de la tastatură lungimea unui tablou de tip *String* și elementele acestuia (numele studenților din semigrupă). Sortați în ordine alfabetică tabloul (v. metoda *sort* a clasei *Arrays*). Atenție, șirurile trebuie transformate după citire astfel încât toate literele să fie mici sau toate mari (v. *toLowerCase* sau *toUpperCase* din *String*).
4. Modificați prima aplicasie propusă astfel încât să citească și vârsta utilizatorului. Dacă vârsta introdusă depășește 100 să afișeze mesajul "Ești bătrân!", altfel să afișeze "Ești tânăr!";
5. Să se implementeze un program care citește de la tastatură lungimea unui tablou de valori întregi și elementele acestuia. Să se sorteze tabloul folosind metoda *sort* a clasei *Arrays*. Să se afișeze pe ecran tabloul sortat.
6. Să se implementeze un program care citește dintr-un fișier într-un tablou de tip *String* numele studenților din semigrupă. Sortați în ordine alfabetică tabloul (v. metoda *sort* a clasei *Arrays*). Atenție, șirurile trebuie transformate după citire astfel încât toate literele să fie mici sau toate mari (v. *toLowerCase* sau *toUpperCase* din *String*).
7. Să se citească dintr-un fișier un tablou de valori întregi și să se afișeze pe ecran media lor aritmetică. Separarea valorilor de pe linii se va face prin clasa *StringTokenizer* prezentată anterior.
8. Studiați clasa *RandomAccessFile* care, spre deosebire de FileInputStream și *FileOutputStream* (acces secvențial), permite citirea respectiv scrierea unei anumite locații din fișier.



Internaționalizare și naționalizare

Java oferă suportul prin numeroase clase pentru internaționalizarea și naționalizarea aplicațiilor

- **Clasa Locale**

```
Locale locale = Locale.getDefault();
System.out.println(locale.getCountry());                          //RO
System.out.println(locale.getDisplayCountry());                   //România
System.out.println(locale.getDisplayCountry(Locale.FRANCE));      //Roumanie
System.out.println(locale.getDisplayLanguage());                  //română
System.out.println(locale.getDisplayLanguage(Locale.FRANCE));     //roumain
System.out.println(locale.getDisplayName());                      //română (România)
System.out.println(locale.getLanguage());                         //ro
Locale g = Locale.GERMANY;
System.out.println(g.getLanguage());                              //de
```

- **Clasa NumberFormat**

```
System.out.println(NumberFormat.getInstance(locale).format(12.3));          //12,3
System.out.println(NumberFormat.getCurrencyInstance(locale).format(6.8));   //6,80 LEI
```

- **Clasa DateFormat**

```
Date date = new Date(2009-1900, 8-1, 31);                                   //Y-1900, M-1, D
System.out.println(DateFormat.getDateInstance(0, locale).format(date));     //31 august 2009
System.out.println(DateFormat.getDateInstance(2, locale).format(date));     //31.08.2009
```

Limbajul Java – crearea claselor (1)

- Definiția unei clase trebuie să conțină cuvântul cheie *class*, numele clasei și corpul clasei. Opțional, înainte de cuvântul *class* pot fi folosiți modificatorii *public* (clasa este vizibilă în toate pachetele), *abstract* (v. clase abstracte) sau *final* (clasa nu poate fi derivată). Exemplu:

```
class Person {
    private String name;
    public void setName(String name){
        this.name = name;
    }
    public String getName(){
        return name;
    }
}
```

- Clasele pot conține:
  - Variabile membru;
  - Constructori;
  - Metode.
- Tipuri de acces: *private*, *protected*, *public*
  - Variabilele și metodele private pot fi accesate doar în cadrul clasei în care s-au declarat;
  - Variabilele și metodele protejate pot fi accesate în interiorul clasei, al subclaselor (v. moștenirea) sau în pachetul (*package*) în care au fost declarate;
  - Variabilele și metodele publice pot fi accesate oriunde;
  - Tipul de acces este implicit protejat (dacă nu se precizează).
- Recomandare (încapsulare, v. cursul POO):
  - Variabile private;
  - Set de metode publice care să permită accesarea variabilelor private.

Limbajul Java – crearea claselor (2)

Exemplul 1 (un singur pachet)

```
package person;

public class Person {
    private String name;
    private String address;
    private int age;
    private void setName(String name){
        this.name = name;
    }
    private String getName(){
        return name;
    }
    protected void setAddress(String address){
        this.address = address;
    }
    protected String getAddress(){
        return address;
    }
    public void setAge(int age){
        this.age = age;
    }
    int getAge(){
        return age;
    }
}
```

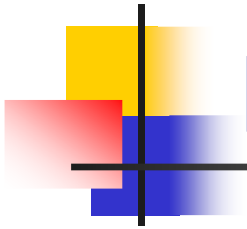
```
package person;

public class Main {
    public static void main(String[] args) {
        Person p = new Person();
        p.name = "Popescu";                                  //Eroare!
        p.setName("Popescu");                                //Eroare!
        p.setAddress("Str. N. Balcescu, Nr. 5");
        p.setAge(103);
        System.out.println("Numele: " + p.getName());       //Eroare!
        System.out.println("Adresa: " + p.getAddress());
        System.out.println("Varsta: " + p.getAge());
    }
}
```

Limbajul Java – crearea claselor (3)

Exemplul 2 (pachete diferite, clasa de bază nepublică)

```
package person;

class Person {
    private String name;
    private String address;
    private int age;
    private void setName(String name){
        this.name = name;
    }
    private String getName(){
        return name;
    }
    protected void setAddress(String address){
        this.address = address;
    }
    protected String getAddress(){
        return address;
    }
    public void setAge(int age){
        this.age = age;
    }
    int getAge(){
        return age;
    }
}
```

```
package main;
import person.Person;

public class Main {
    public static void main(String[] args) {
        Person p = new Person();                    //Eroare!
    }
}
```

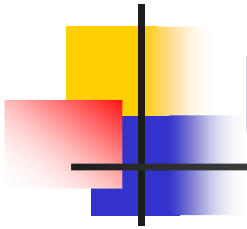


Limbajul Java – crearea claselor (4)

Exemplul 3 (pachete diferite, clasa de bază publică)

```
package person;

public class Person {
    private String name;
    private String address;
    private int age;
    private void setName(String name){
        this.name = name;
    }
    private String getName(){
        return name;
    }
    protected void setAddress(String address){
        this.address = address;
    }
    protected String getAddress(){
        return address;
    }
    public void setAge(int age){
        this.age = age;
    }
    int getAge(){
        return age;
    }
}
```

```
package main;
import person.Person;

public class Main {
    public static void main(String[] args) {
        Person p = new Person();
        p.name = "Popescu";                                  //Eroare!
        p.setName("Popescu");                                //Eroare!
        p.setAddress("Str. N. Balcescu, Nr. 5");             //Eroare!
        p.setAge(103);
        System.out.println("Numele: " + p.getName());       //Eroare!
        System.out.println("Adresa: " + p.getAddress());    //Eroare!
        System.out.println("Varsta: " + p.getAge());        //Eroare!
    }
}
```

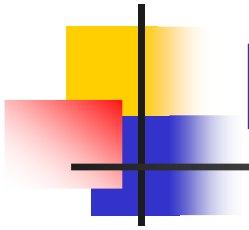


Limbajul Java – crearea claselor (5)

Exemplul 4 (o variantă corectă)

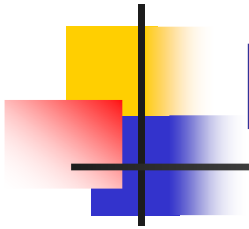
```
package person;

public class Person {
    private String name;
    private String address;
    private int age;
    public void setName(String name){
        this.name = name;
    }
    public String getName(){
        return name;
    }
    public void setAddress(String address){
        this.address = address;
    }
    public String getAddress(){
        return address;
    }
    public void setAge(int age){
        this.age = age;
    }
    public int getAge(){
        return age;
    }
}
```

```
package person;

public class Main {
    public static void main(String[] args) {
        Person p = new Person();
        p.setName("Popescu");
        p.setAddress("Str. N. Balcescu, Nr. 5");
        p.setAge(103);
        System.out.println("Numele: " + p.getName());
        System.out.println("Adresa: " + p.getAddress());
        System.out.println("Varsta: " + p.getAge());
    }
}
```



Limbajul Java – constructori (1)

- Este o metodă publică fără tip, având același nume cu cel al clasei;
- Principalul rol al constructorului este acela de a inițializa obiectul pentru care s-a apelat;
- Dacă într-o clasă nu se declară un constructor, atunci compilatorul generează unul implicit, fără parametri;
- Într-o clasă pot coexista mai mulți constructori cu parametri diferiți (tip, număr);
- La instanțierea unui obiect se apelează constructorul corespunzător parametrilor de apel;
- În Java nu există destructori. Sistemul apelează periodic mecanismul *garbage collector*.

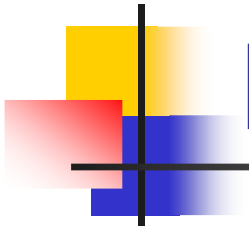


Limbajul Java – constructori (2)

Exemplu

```
public class Person {
    private String name;
    private String address;
    private int age;
    public Person(String name, String address, int age) {
        this.name = name;
        this.address = address;
        this.age = age;
    }
    public void setName(String name){
        this.name = name;
    }
    public String getName(){
        return name;
    }
    public void setAddress(String address){
        this.address = address;
    }
    public String getAddress(){
        return address;
    }
    public void setAge(int age){
        this.age = age;
    }
    public int getAge(){
        return age;
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Person p = new Person("Pop", "Sibiu", 103);
        System.out.println("Numele: " + p.getName());     //Pop
        System.out.println("Adresa: " + p.getAddress());  //Sibiu
        System.out.println("Varsta: " + p.getAge());      //103
        p.setName("Rus");
        p.setAddress("Avrig");
        p.setAge(98);
        System.out.println("Numele: " + p.getName());     //Rus
        System.out.println("Adresa: " + p.getAddress());  //Avrig
        System.out.println("Varsta: " + p.getAge());      //98
    }
}
```



Limbajul Java – moștenirea (1)

- Unul dintre marile avantaje ale programării orientate-obiect (POO) constă în reutilizarea codului;
- Toate clasele Java, cu excepția interfețelor, sunt subclase ale clasei rădăcină *Object*;
- Procesul prin care o clasă nouă refolosește o clasă veche se numește moștenire;
- O clasă, numită clasă derivată, poate moșteni variabilele și metodele unei alte clase, numită clasă de bază;
- Dacă apare aceeași metodă cu aceiași parametri atât în clasa de bază cât și în cea derivată (prin redefinire) și se apelează metoda instanței clasei derivate, se execută metoda clasei derivate;
- Derivarea unei clase în Java se face prin cuvântul cheie *extends* urmat de numele clasei de bază;
- O clasă declarată *final* nu poate fi derivată;
- Java nu permite moștenire multiplă: o clasă poate deriva o singură clasă de bază. Moștenirea multiplă este permisă doar folosind interfețele (vor fi prezentate);
- Constructorii clasei derivate pot folosi printr-un apel *super* constructorii clasei de bază pentru inițializarea variabilelor moștenite;
- În exemplul prezentat în continuare, clasele Student și Teacher moștenesc clasa Person;

Limbajul Java – moștenirea (2)

Exemplu

```
public class Person {
    private String name;
    private String address;
    private int age;
    public Person(String name, String address, int age) {
        this.name = name;
        this.address = address;
        this.age = age;
    }
    public void setName(String name){
        this.name = name;
    }
    public String getName(){
        return name;
    }
    public void setAddress(String address){
        this.address = address;
    }
    public String getAddress(){
        return address;
    }
    public void setAge(int age){
        this.age = age;
    }
    public int getAge(){
        return age;
    }
}
```

```
public class Student extends Person {
    private int grade;                    //medie
    public Student(String name, String address, int age, int grade) {
        super(name, address, age);    //trebuie sa fie prima instructiune!
        this.grade = grade;
    }
    public void setGrade(int grade){
        this.grade = grade;
    }
    public int getGrade(){
        return grade;
    }
}

public class Teacher extends Person {
    private int courses;                  //nr. cursuri
    public Teacher(String name, String address, int age, int courses) {
        super(name, address, age);    //trebuie sa fie prima instructiune!
        this.courses = courses;
    }
    public void setCourses(int courses){
        this.courses = courses;
    }
    public int getCourses(){
        return courses;
    }
}

public class Main {
    public static void main(String[] args) {
        Student s = new Student("Rus", "Avrig", 98, 4);
        System.out.println(s.getAge());                    //98
    }
}
```

Limbajul Java – moștenirea (3)

- Dacă un constructor al clasei derivate nu apelează prin *super* un constructor al clasei de bază, compilatorul va apela implicit constructorul fără parametri al clasei de bază. Dacă însă clasa de bază are doar constructori cu parametri, compilatorul nu va genera constructorul implicit. În acest caz, în clasa de bază trebuie declarat și un constructor fără parametri.

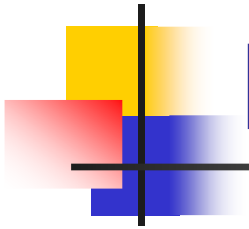
- Exemplu:

```
public class Base {
    public Base(){
    }
    public Base(int p) {
    }
}

public class Sub extends Base{
    public Sub() {
    }
}
```

Limbajul Java – moștenirea (4)

Supraîncărcarea metodelor

```
public class Base {
    public void method(int i) {
        System.out.println("Base integer is " + i);
    }
    public void method(String s){
        System.out.println("Base string is " + s);
    }
}


public class Sub extends Base {
    public void method(int j){
        System.out.println("Sub integer is " + j);
    }
    public static void main(String[] args) {
        Base b1 = new Base();
        Base b2 = new Sub();
        Sub s = new Sub();
        b1.method(1);                   //"Base integer is 1"
        b2.method(2);                   //"Sub integer is 2"
        s.method(3);                    //"Sub integer is 3"
        s.method("4");                  //"Base string is 4"
    }
}
```



Limbajul Java – moștenirea (5)

Aplicații

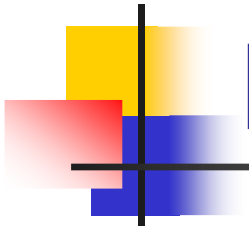
1. Să se implementeze o aplicație care să permită introducerea de la tastatură a numărului de studenți din semigrupă urmat de datele lor (într-un tablou) și a numărului de profesori din acest semestru urmat de datele lor (într-un alt tablou). Căutați și afișați studentul cu media cea mai mare respectiv profesorul cu cele mai multe cursuri.
2. Definiți clasa **Employee** (angajat) derivată din clasa **Person** prezentată, având câmpul suplimentar **salary**. Introduceți de la tastatură 5 obiecte de tip **Employee** într-un tablou. Căutați și afișați angajatul cu salariul cel mai mare.
3. Definiți clasa **Product** cu câmpul **price** (preț). Definiți clasa **Book** (carte) derivată din clasa **Product** având câmpurile suplimentare **title**, **author** și **publisher** (editor). Introduceți de la tastatură 5 obiecte de tip **Book** într-un tablou. Căutați și afișați cartea cu prețul cel mai mic.
4. Definiți clasa **Book** (carte) cu câmpurile **title**, **author** și **publisher** (editor). Definiți clasa **LibraryCard** (fișă de bibliotecă) derivată din clasa **Book** având câmpul suplimentar **copies** (nr. exemplare). Introduceți de la tastatură 5 obiecte de tip **LibraryCard** într-un tablou. Căutați și afișați cartea cu numărul cel mai mare de exemplare.
5. Definiți clasa **Vehicle** cu câmpul **manufacturer** (fabricant). Definiți clasa **Car** derivată din clasa **Vehicle** având câmpurile suplimentare **model**, **length** (lungime), **maxSpeed** (viteză maximă), **price** (preț). Introduceți de la tastatură 5 obiecte de tip **Car** într-un tablou. Căutați și afișați mașina cu prețul cel mai mic.

Limbajul Java – clase abstracte

- O clasă trebuie declarată abstractă dacă conține cel puțin o metodă abstractă.
- Metodele abstracte sunt declarate fără implementare urmând ca ele să fie implementate în clasele derivate. O clasă abstractă nu poate fi instanțiată.
- Exemplu:

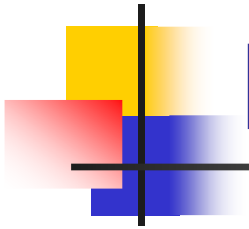
```
public abstract class Product {
    private double price;
    public Product(double price) {
        this.price = price;
    }
    public abstract double computeFinalPrice();
    public void setPrice(double price){
        this.price = price;
    }
    public double getPrice(){
        return price;
    }
}

public class Book extends Product {
    public Book(double price) {
        super(price);
    }
    public double computeFinalPrice(){          //TVA=9%
        return getPrice() + (9*getPrice())/100;
    }
}
```

```
public class Car extends Product{
    public Car(double price) {
        super(price);
    }
    public double computeFinalPrice(){          //TVA=19%
        return getPrice() + (19*getPrice())/100;
    }
}

public class Main {
    public static void main(String[] args) {
        Product p = new Product(10);            //Eroare!
        Book b = new Book(100);
        System.out.println(b.computeFinalPrice());
        Car c = new Car(100000);
        System.out.println(c.computeFinalPrice());
    }
}
```



Limbajul Java – interfețe (1)

- Interfețele oferă avantajele moștenirii multiple, evitând complicațiile acesteia.
- Toate metodele dintr-o interfață sunt implicit publice și abstracte. O interfață furnizează numele metodelor fără implementarea lor. P interfață care conține o singură metodă se numește funcțională.
- Interfețele nu pot fi instanțiate.
- Clasa care folosește o interfață trebuie să implementeze toate metodele acesteia.
- Se poate deriva o interfață din alte interfețe prin același mecanism care a fost prezentat la clase (v. moștenirea).
- În continuare vor fi prezentate câteva exemple:
  - O interfață *Price* folosită de clasele *Product*, *Book* și *Car*.
  - Folosirea interfețelor *Comparator* sau *Comparable* pentru sortarea unui tablou de obiecte prin metoda *sort* din clasa *Arrays*. În cazul interfeței *Comparator* trebuie implementată metoda *compare*.
  - Interfața *Serializable* pentru serializarea obiectelor.



Limbajul Java – interfețe (2)

Exemplul 1 (interfața *Price*, implementare *computeFinalPrice*)

```
public interface Price {
    public double computeFinalPrice();
}

public abstract class Product implements Price{
    private double price;
    public Product(double price) {
        this.price = price;
    }
    public void setPrice(double price){
        this.price = price;
    }
    public double getPrice(){
        return price;
    }
}

public class Book extends Product {
    public Book(double price) {
        super(price);
    }
    public double computeFinalPrice(){          //TVA=9%
        return getPrice() + (9*getPrice())/100;
    }
}
```

```
public class Car extends Product{
    public Car(double price) {
        super(price);
    }
    public double computeFinalPrice(){          //TVA=19%
        return getPrice() + (19*getPrice())/100;
    }
}

public class Main {
    public static void main(String[] args) {
        Product p = new Product(10);            //Eroare!
        Book b = new Book(100);
        System.out.println(b.computeFinalPrice());
        Car c = new Car(100000);
        System.out.println(c.computeFinalPrice());
    }
}
```

Limbajul Java – interfețe (3)

Exemplul 2 (interfața *Comparator*, implementare *compare*)

```
public class Item {
    String str;
    int num;
    Item(String str, int num) {
        this.str = str;
        this.num = num;
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Item t[] = new Item[5];
        t[0] = new Item("c", 2);
        t[1] = new Item("a", 1);
        t[2] = new Item("e", 4);
        t[3] = new Item("b", 5);
        t[4] = new Item("d", 3);
        //Sortare dupa campul num
        java.util.Arrays.sort(t, new Comparator() {
            public int compare(Object o1, Object o2) {
                //return ((Item)o1).num-((Item)o2).num;
                if(((Item)o1).num<((Item)o2).num) return -1;
                if(((Item)o1).num>((Item)o2).num) return 1;
                return 0;
            }
        });
        //Sortare dupa campul str
        java.util.Arrays.sort(t, new Comparator() {
            public int compare(Object o1, Object o2) {
                return ((Item)o1).str.compareTo(((Item)o2).str);
            }
        });
    }
}
```

Limbajul Java – interfețe (4)

Exemplul 3 (interfața *Comparator*, implementare *compare*)

```
public class Item implements Comparator {
    String str;
    int num;
    Item() {}
    Item(String str, int num) {
        this.str = str;
        this.num = num;
    }
    public int compare(Object o1, Object o2) {
       return ((Item)o1).num-((Item)o2).num;
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Item t[] = new Item[5];
        t[0] = new Item("c", 2);
        t[1] = new Item("a", 1);
        t[2] = new Item("e", 4);
        t[3] = new Item("b", 5);
        t[4] = new Item("d", 3);
        //Sortare dupa campul num
        java.util.Arrays.sort(t, new Item());
        //Afisare
        for (int i=0; i<t.length; i++)
           System.out.println(t[i].str + " - " + t[i].num);
        //Sortare dupa campul str
        java.util.Arrays.sort(t, new Comparator() {
            public int compare(Object o1, Object o2) {
                return ((Item)o1).str.compareTo(((Item)o2).str);
            }
        });
        //Afisare
        for (int i=0; i<t.length; i++)
           System.out.println(t[i].str + " - " + t[i].num);
    }
}
```

Limbajul Java – interfețe (5)

Exemplul 4 (interfața *Comparable*, implementare *compareTo*)

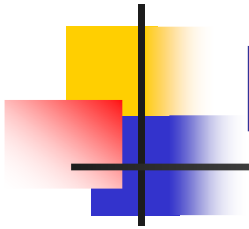
```
public class Item implements Comparable {
    String str;
    int num;
    Item(String str, int num) {
        this.str = str;
        this.num = num;
    }
    public int compareTo(Object o) {
       return num-((Item)o).num;
       //return str.compareTo(((Item)o).str);
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Item t[] = new Item[5];
        t[0] = new Item("c", 2);
        t[1] = new Item("a", 1);
        t[2] = new Item("e", 4);
        t[3] = new Item("b", 5);
        t[4] = new Item("d", 3);
        //Sortare dupa campul num
        java.util.Arrays.sort(t);
        //Afisare
        for (int i=0; i<t.length; i++)
           System.out.println(t[i].str + " - " + t[i].num);
        //Sortare dupa campul str
        java.util.Arrays.sort(t, new Comparator() {
            public int compare(Object o1, Object o2) {
                return ((Item)o1).str.compareTo(((Item)o2).str);
            }
        });
        //Afisare
        for (int i=0; i<t.length; i++)
           System.out.println(t[i].str + " - " + t[i].num);
    }
}
```

Limbajul Java – interfețe (6)

Exemplul 5 (interfața *Serializable* – permite serializarea obiectelor)

```
public class Item implements Serializable {
    String str;
    int num;
    Item(String str, int num) {
        this.str = str;
        this.num = num;
    }
}


public class Main {
    public static void main(String[] args) {
        FileOutputStream fos = null;
        FileInputStream fis = null;
        ObjectOutputStream oos = null;
        ObjectInputStream ois = null;
        try{
            fos = new FileOutputStream("data.txt");
            fis = new FileInputStream("data.txt");
        }
        catch(FileNotFoundException e){
            e.printStackTrace();
        }
        try{
            oos = new ObjectOutputStream(fos);
            ois = new ObjectInputStream(fis);
        }
        catch(IOException e){
            e.printStackTrace();
        }
        try{
            oos.writeObject(new Item("c", 2));   //Scrierea obiectului in fisier
            oos.close(); fos.close();
        }
        catch(IOException e){
            e.printStackTrace();
        }
        try{
            Item item=(Item)ois.readObject();   //Citirea obiectului din fisier
            System.out.println(item.str + " - " + item.num);
            ois.close(); fis.close();
        }
        catch(IOException e){
            e.printStackTrace();
        }
        catch(ClassNotFoundException e){
            e.printStackTrace();
        }
    }
}
```



Limbajul Java – interfețe (7)

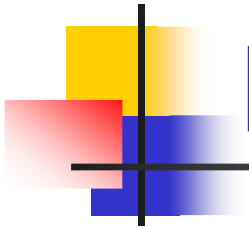
Expresii lambda

- Disponibile din JDK 1.8, expresiile lambda pot fi folosite pentru a crea instanțe ale unor interfețe funcționale [Blo17].
- Pentru expresiile lambda s-a introdus noul operator -> în Java.
- Expresiile lambda sunt compuse din două părți. Partea din stânga operatorului -> specifică lista parametrilor separați prin virgulă, iar partea din dreapta este corpul lambda care conține operațiile efectuate de expresia lambda. Spre exemplu, expresia lambda

```
(a,b)->a+b
```

- calculează suma parametrilor a și b. În cazul în care în partea dreaptă a expresiei lambda este un bloc de instrucțiuni, acesta trebuie furnizat între acolade:

```
(a,b)->{return a+b;}
```



Limbajul Java – interfețe (8)

Exemplul 1 (interfața *Arithmetic*, aplicare operații aritmetice)

```
public interface Arithmetic {
    int operation(int a, int b);
}


public class Main {
    public static void main(String[] args) {
        // TODO code application logic here
        Arithmetic addition = (a,b)->a+b;
        System.out.println(addition.operation(6,2)); //8
        Arithmetic substraction = (a,b)->a-b;
        System.out.println(substraction.operation(6,2)); //4
        Arithmetic multiplication = (a,b)->a*b;
        System.out.println(multiplication.operation(6,2));//12
        Arithmetic division = (a,b)->a/b;
        System.out.println(division.operation(6,2)); //3
    }
}
```

Limbajul Java – interfețe (9)

Exemplul 2 (interfața *Comparator*, sortare folosind expresii lambda)

```
public class Item {
    String str;
    int num;
    Item(String str, int num) {
        this.str = str;
        this.num = num;
    }
}
public class Comparisons {
     public int compareByStr(Item a, Item b){
         return a.str.compareTo(b.str);
     }
     public int compareByNum(Item a, Item b){
         return a.num-b.num;
     }
}
public class Main {
    public static void main(String[] args) {
        Item t[] = new Item[5]; Comparisons c = new Comparisons();
        t[0] = new Item("c", 2); t[1] = new Item("a", 1);
        t[2] = new Item("e", 4); t[3] = new Item("b", 5); t[4] = new Item("d", 3);
        java.util.Arrays.sort(t,(o1,o2) -> c.compareByNum(o1, o2)); //Sortare dupa campul num
        java.util.Arrays.sort(t,(o1,o2) -> c.compareByStr(o1, o2));   //Sortare dupa campul str
    }
}
```

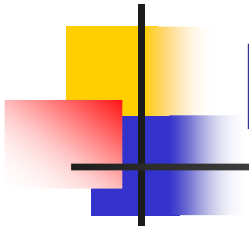
Limbajul Java – interfețe (10)

Exemplul 3 (Intarfața *Comparator*, fără clasa Comparisons din exemplul 2)

```
public class Item {
    String str;
    int num;
    Item(String str, int num) {
        this.str = str;
        this.num = num;
    }
}
public class Main {
    public static void main(String[] args) {
        Item t[] = new Item[5];
        t[0] = new Item("c", 2); t[1] = new Item("a", 1);
        t[2] = new Item("e", 4); t[3] = new Item("b", 5); t[4] = new Item("d", 3);
        //Sortare dupa campul num
        java.util.Arrays.sort(t,(o1,o2) -> {return o1.num-o2.num;});
        //Sortare dupa campul str
        java.util.Arrays.sort(t,(o1,o2) -> {return o1.str.compareTo(o2.str);});
    }
}
```

Afișarea se poate face cu ajutorul buclei:

```
for(Item i:t)
    System.out.println(i.str + " " + i.num);
```



Limbajul Java – interfețe (11)

Aplicații

1. Definiți clasa **Person** cu câmpurile **name**, **address** și **age**. Definiți clasa **Student** derivată din clasa **Person** având câmpul suplimentar **grade** (medie). Introduceți dintr-un fișier *student.txt* numărul de studenți din semigrupă urmat de datele lor, într-un tablou. Sortați și afișați studenții în ordinea crescătoare a mediei.
2. Definiți clasa **Person** cu câmpurile **name**, **address** și **age**. Definiți clasa **Teacher** derivată din clasa **Person** având câmpul suplimentar **courses** (nr. cursuri). Introduceți dintr-un fișier *teacher.txt* numărul de profesori din acest semestru urmat de datele lor, într-un tablou. Sortați și afișați profesorii în ordinea crescătoare a numărului de cursuri predate.
3. Definiți clasa **Person** cu câmpurile **name**, **address** și **age**. Definiți clasa **Employee** (angajat) derivată din clasa **Person** având câmpul suplimentar **salary** (salariu). Introduceți dintr-un fișier *employee.txt* numărul de angajați urmat de datele lor, într-un tablou. Sortați și afișați angajații în ordinea crescătoare a salariilor.
4. Definiți clasa **Product** cu câmpul **price** (preț). Definiți clasa **Book** (carte) derivată din clasa **Product** având câmpurile suplimentare **title**, **author** și **publisher** (editor). Introduceți dintr-un fișier *book.txt* numărul de cărți urmat de datele lor, într-un tablou. Sortați și afișați cărțile în ordinea crescătoare a prețului.
5. Definiți clasa **Book** (carte) cu câmpurile **title**, **author** și **publisher** (editor). Definiți clasa **LibraryCard** (fișă de bibliotecă) derivată din clasa **Book** având câmpul suplimentar **copies** (nr. exemplare). Introduceți dintr-un fișier *librarycard.txt* numărul de fișe urmat de datele acestora, într-un tablou. Sortați și afișați cărțile în ordinea crescătoare a numărului de exemplare disponibile.
6. Definiți clasa **Vehicle** cu câmpul **manufacturer** (fabricant). Definiți clasa **Car** derivată din clasa **Vehicle** având câmpurile suplimentare **model**, **length** (lungime), **maxSpeed** (viteză maximă), **price** (preț). Introduceți dintr-un fișier *car.txt* numărul de mașini, urmat de datele lor, într-un tablou. Sortați și afișați mașinile în ordinea crescătoare a prețului.

Limbajul Java – partajarea datelor (1)

- Partajarea datelor de către obiectele unei clase poate fi realizată prin intermediul membrilor statici ai clasei respective.
- În timp ce variabilele obișnuite (non-statice) aparțin instanțelor clasei (obiectelor), variabilele declarate statice aparțin clasei și sunt partajate de către toate obiectele acesteia.
- Unei variabile statice i se alocă memorie o singură dată, la prima instanțiere a clasei. La următoarele instanțieri ale clasei variabilei statice nu i se mai alocă memorie, dar toate obiectele clasei pot accesa aceeași variabilă statică, alocată la prima instanțiere.
- Metodele statice pot fi apelate fără instanțierea clasei.
- Metodele statice nu pot utiliza variabile și metode non-statice.
- Un exemplu de utilizare a unei variabile statice este contorizarea obiectelor instanțiate dintr-o clasă:

```
public class Person {
    private String name;
    static int counter;
    public Person(String name) {
        this.name = name;
        counter++;
        System.out.println(counter + " persoane.");
    }
    public void setName(String name){
        this.name = name;
    }
    public String getName(){
        return name;
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Person p1 = new Person("Popescu");      //"1 persoane."
        Person p2 = new Person("Ionescu");      //"2 persoane."
    }
}
```

Limbajul Java – partajarea datelor (2)

Exemple de folosire a metodelor statice:

```
public class Person {
    private String name;
    private int age;
    static int counter;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
        increaseCounter();
    }
    public static void increaseCounter(){
        counter++;
        System.out.println(counter + " persoane.");
    }
    public static void increaseAge(){
        age++;                          //Eroare!
    }
    public void setName(String name){
        this.name = name;
    }
    public String getName(){
        return name;
    }
    public void setAge(int age){
        this.age = age;
    }
    public int getAge(){
        return age;
    }
}
```

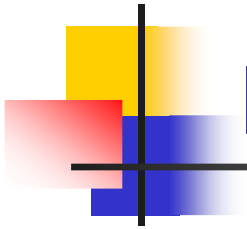
```
public class Main {
    public static void main(String[] args) {
        Person p1 = new Person("Popescu", 61);      //"1 persoane."
        Person p2 = new Person("Ionescu", 72);      //"2 persoane."
        Person.increaseCounter();                   //"3 persoane."
        p1 = new Person("Petrescu", 53);            //"4 persoane."
    }
}
```

Limbajul Java – partajarea datelor (3)

Alte exemple de folosire a variabilelor și metodelor statice:

```
public class Person {
    private String name;
    private int age;
    static int counter;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
        increaseCounter();
    }
    public static void increaseCounter(){
        counter++;
        System.out.println(counter + " persoane.");
    }
    public void printCounter(){
       System.out.println(counter);
    }
    public void setName(String name){
        this.name = name;
    }
    public String getName(){
        return name;
    }
    public void setAge(int age){
        this.age = age;
    }
    public int getAge(){
        return age;
    }
```

```
    public static void main(String[] args) {
        Person p1 = new Person("Popescu", 61);      //"1 persoane."
        Person p2 = new Person("Ionescu", 72);      //"2 persoane."
        Person.increaseCounter() ;                  //"3 persoane."
        p1 = new Person("Petrescu", 53);            //"4 persoane."
        printCounter();                             //Eroare!
        p2.printCounter();                          //"4"
    }
}
```



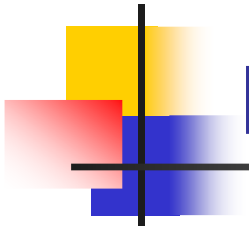
Limbajul Java – atribuire vs. clonare (1)

Folosirea operatorului de atribuire

În cazul obiectelor operatorul de atribuire determină atribuirea referințelor (adreselor)

```
public class Person {
    private String name;
    private int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public int getAge() {
        return age;
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Person p1 = new Person("Pop", 61);
        Person p2 = p1;
        System.out.println(p1.getName() + ", " + p1.getAge());      //"Pop, 61"
        System.out.println(p2.getName() + ", " + p2.getAge());      //"Pop, 61"
        p1.setAge(p1.getAge() + 1);
        System.out.println(p1.getName() + ", " + p1.getAge());      //"Pop, 62"
        System.out.println(p2.getName() + ", " + p2.getAge());      //"Pop, 62"
        p2.setName("Rus");
        System.out.println(p1.getName() + ", " + p1.getAge());      //"Rus, 62"
        System.out.println(p2.getName() + ", " + p2.getAge());      //"Rus, 62"
        p1 = new Person("Lup", 53);
        System.out.println(p1.getName() + ", " + p1.getAge());      //"Lup, 53"
        System.out.println(p2.getName() + ", " + p2.getAge());      //"Rus, 62"
        p1.setAge(p1.getAge() + 1);
        System.out.println(p1.getName() + ", " + p1.getAge());      //"Lup, 54"
        System.out.println(p2.getName() + ", " + p2.getAge());      //"Rus, 62"
    }
}
```

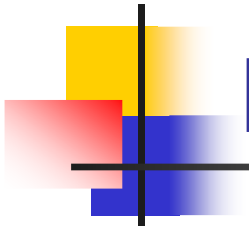


Limbajul Java – atribuire vs. clonare (2)

Clonarea obiectelor

```
public class Person implements Cloneable {
    private String name;
    private int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public Object clone() {  //protected in Object
        Object obj = null;
        try {
            obj = super.clone();
        }
        catch (CloneNotSupportedException ex) {
        }
        return obj;
    }
    public void setName(String name) {
        this.name = name;
    }
    public String getName() {
        return name;
    }
    public void setAge(int age) {
        this.age = age;
    }
    public int getAge() {
        return age;
    }
}
```

```
public class Main {
    public static void main(String[] args) {
        Person p1 = new Person("Pop", 61);
        Person p2 = (Person)p1.clone();
        System.out.println(p1.getName() + ", " + p1.getAge());      //"Pop, 61"
        System.out.println(p2.getName() + ", " + p2.getAge());      //"Pop, 61"
        p1.setAge(p1.getAge() + 1);
        System.out.println(p1.getName() + ", " + p1.getAge());      //"Pop, 62"
        System.out.println(p2.getName() + ", " + p2.getAge());      //"Pop, 61"
        p2.setName("Rus");
        System.out.println(p1.getName() + ", " + p1.getAge());      //"Pop, 62"
        System.out.println(p2.getName() + ", " + p2.getAge());      //"Rus, 61"
    }
}
```

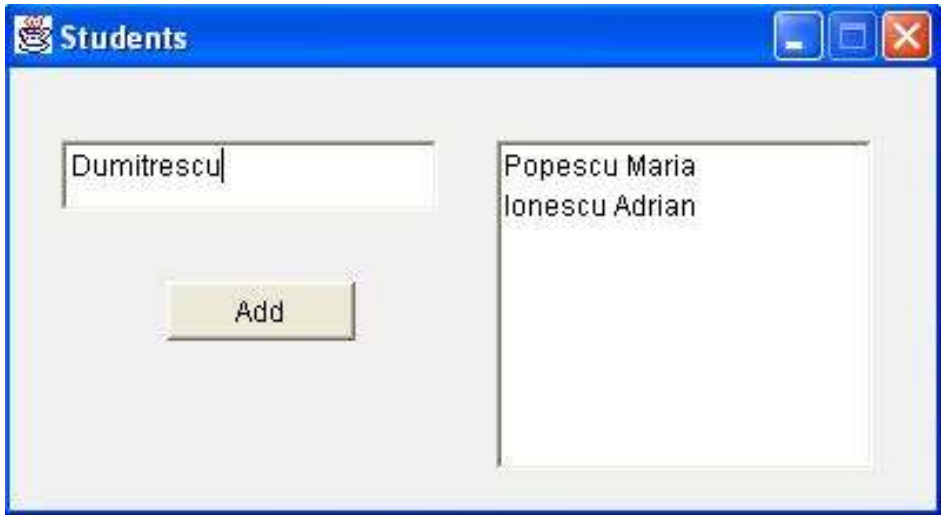


Limbajul Java – interfață grafică (1)

- O componentă foarte importantă a aplicațiilor moderne este interfața grafică prin care acestea comunică cu utilizatorul: se citesc datele și se afișează rezultatele.
- Mediile de dezvoltare Java oferă numeroase pachete cu diverse componente vizuale.
- În următoarea aplicație vom folosi componente din pachetul AWT (Advanced Windowing Toolkit) pentru citirea studenților într-o listă
- Fereastra neredimensionabilă a aplicației este prezentată pe pagina următoare.

Limbajul Java – interfață grafică (2)

Fereastra aplicației

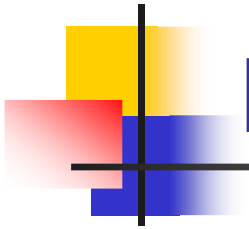


- Numele studentului se introduce printr-un *TextField*;
- Acționarea butonului *Add* determină adăugarea textului din TextField în componenta *List* respectiv ștergerea din *TextField*;
- Adăugarea în componenta *List* determină apariția unui scroll atunci când numărul de linii introduse trece de limita de vizualizare.

Limbajul Java – interfață grafică (3)

Codul sursă al aplicației

```
public class StudentFrame extends Frame {
  private List studentList = new List();
  private TextField studentTextField = new TextField();
  private Button addButton = new Button();

  public StudentFrame() {
    try {
      jbInit();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
    show();
  }

  public static void main(String[] args) {
    StudentFrame studentFrame = new StudentFrame();
  }

  private void jbInit() throws Exception {
    this.addWindowListener(new java.awt.event.WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        this_windowClosing(e);
      }
    });
    studentList.setBounds(new Rectangle(210, 60, 160, 140));
    studentTextField.setBounds(new Rectangle(25, 60, 160, 30));
    addButton.setLabel("Add");
    addButton.setBounds(new Rectangle(70, 120, 80, 25));
    addButton.addActionListener(new java.awt.event.ActionListener() {
      public void actionPerformed(ActionEvent e) {
        addButton_actionPerformed(e);
      }
    });
    this.setSize(400, 220);
    this.setResizable(false);
    this.setLayout(null);  //componentele pot fi asezate cu mouse-ul
    this.setTitle("Students");
    this.setBackground(new Color(240, 240, 240));
    this.add(addButton, null);
    this.add(studentTextField, null);
    this.add(studentList, null);
  }

  void this_windowClosing(WindowEvent e) {
    System.exit(0);      //inchiderea aplicatiei
  }

  void addButton_actionPerformed(ActionEvent e) {
    studentList.add(studentTextField.getText());
    studentTextField.setText("");       //stergere
  }
}
```



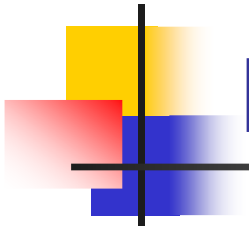
Limbajul Java – interfață grafică (4)

Tratarea evenimentelor

- Evenimentele sunt generate de acțiunile utilizatorului asupra componentelor interfeței grafice.
- Pentru tratarea unui anumit eveniment într-o clasă, trebuie implementată interfața *Listener* corespunzătoare care să prelucreze evenimentul respectiv. De exemplu, pentru tratarea evenimentelor *ActionEvent*, se implementează interfața *ActionListener*.
- Astfel, evenimentele sunt recepționate de obiecte de tip *Listener* care apelează metode de tratare a evenimentelor.
- Aplicația anterioară tratează evenimentele de închidere a ferestrei și de click pe butonul *Add*.
- În continuare vom extinde aplicația astfel încât să răspundă la apăsarea tastei ENTER, tratând evenimentul *KeyEvent.* Pentru asta se poate folosi interfața *KeyListener* implementând toate metodele (*keyPressed*, *keyReleased*, *keyTyped*) sau clasa abstractă *KeyAdapter* care ne permite să implementăm doar metodele necesare (în cazul nostru *keyPressed*).

Limbajul Java – interfață grafică (5)

```
public class StudentFrame extends Frame {
  private List studentList = new List();
  private TextField studentTextField = new TextField();
  private Button addButton = new Button();

  public StudentFrame() {
    try {
      jbInit();
    }
    catch(Exception e) {
      e.printStackTrace();
    }
    show();
    studentTextField.requestFocus();
  }

  public static void main(String[] args) {
    StudentFrame studentFrame = new StudentFrame();
  }

  private void jbInit() throws Exception {
    this.addWindowListener(new java.awt.event.WindowAdapter() {
      public void windowClosing(WindowEvent e) {
        this_windowClosing(e);
      }
    });
    studentList.setBounds(new Rectangle(210, 60, 160, 140));
    studentTextField.setBounds(new Rectangle(25, 60, 160, 30));
    studentTextField.addKeyListener(new java.awt.event.KeyAdapter() {
      public void keyPressed(KeyEvent e) {
        studentTextField_keyPressed(e);
      }
    });
    addButton.setLabel("Add");
    addButton.setBounds(new Rectangle(70, 120, 80, 25));
    addButton.addActionListener(new java.awt.event.ActionListener() {
      public void actionPerformed(ActionEvent e) {
        addButton_actionPerformed(e);
      }
    });
    this.setSize(400, 220);
    this.setResizable(false);
    this.setLayout(null);  //componentele pot fi asezate cu mouse-ul
    this.setTitle("Students");
    this.setBackground(new Color(240, 240, 240));
    this.add(addButton, null);
    this.add(studentTextField, null);
    this.add(studentList, null);
  }

  void this_windowClosing(WindowEvent e) {
    System.exit(0);         //inchiderea aplicatiei
  }

  void addButton_actionPerformed(ActionEvent e) {
    studentList.add(studentTextField.getText());
    studentTextField.setText("");             //stergere
    studentTextField.requestFocus();
  }

  void studentTextField_keyPressed(KeyEvent e) {
    if(e.getKeyCode()==e.VK_ENTER)
      addButton_actionPerformed(new ActionEvent(this, 0, null));
  }
}
```



Limbajul Java – interfață grafică (6)

Layout manager (aranjarea componentelor)

- Toate obiectele grafice de tip container (*Frame*, *Panel*, etc.) au o proprietate layout prin care se specifică cum să fie așezate și dimensionate componentele: *null*, *XYLayout*, *BorderLayout*, *FlowLayout*, *GridLayout*, *GridBagLayout*, etc.
- Un layout *null* (folosit și în aplicația prezentată) sau *XYLayout* păstrează pozițiile și dimensiunile originale ale componentelor. Se pot folosi în cazul containerelor neredimensionabile.
- *BorderLayout*, *FlowLayout*, *GridLayout* și *GridBagLayout* repoziționează și/sau rescalează obiectele în cazul redimensionării containerului.

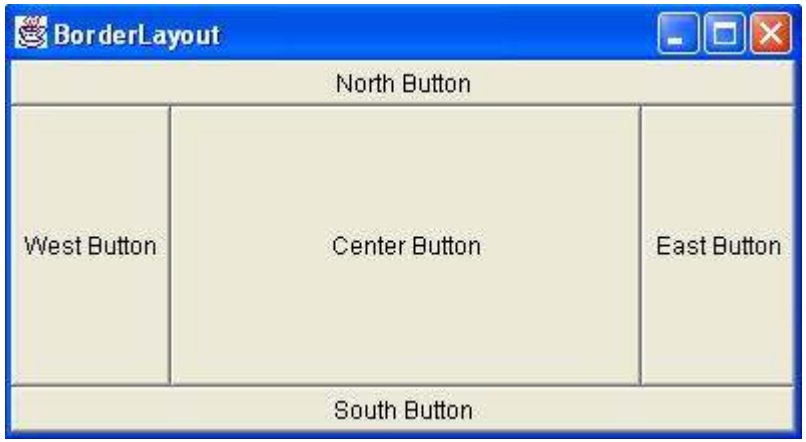
Limbajul Java – interfață grafică (7)

BorderLayout permite aranjarea componentelor din container în 5 zone: nord, sud, est, vest și centru. Componentele din nord și sud își păstrează înălțimea originală și sunt scalate la lățimea containerului. Componentele din est și vest își păstrează lățimea originală și sunt scalate vertical astfel încât să acopere spațiul dintre nord și sud. Componenta din centru se expandează pentru acoperirea întreg spațiului rămas.

Constructori:

BorderLayout() – fără distanţă între componente;

BorderLayout(int hgap, int vgap) – primeşte ca parametri distanţele orizontale şi verticale dintre componente.




```
public class BorderFrame extends Frame {
  private BorderLayout bLayout = new BorderLayout();
  private Button northButton = new Button("North Button");
  private Button southButton = new Button("South Button");
  private Button westButton = new Button("West Button");
  private Button eastButton = new Button("East Button");
  private Button centerButton = new Button("Center Button");

  public BorderFrame() {
    this.setSize(400, 220);
    this.setTitle("BorderLayout");
    this.setLayout(bLayout);
    this.setBackground(new Color(240, 240, 240));
    this.add(northButton, BorderLayout.NORTH);
    this.add(southButton, BorderLayout.SOUTH);
    this.add(westButton, BorderLayout.WEST);
    this.add(eastButton, BorderLayout.EAST);
    this.add(centerButton, BorderLayout.CENTER);
    this.show();
  }

  public static void main(String[] args) {
    new BorderFrame();
  }
}
```

Limbajul Java – interfață grafică (8)

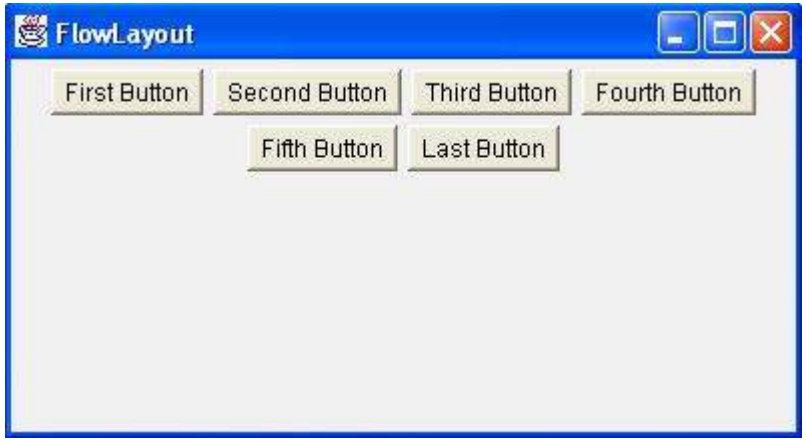
FlowLayout aranjează componentele pe linii păstrând dimensiunile. Aliniază componentele care încap într-o linie și dacă e nevoie continuă pe linia următoare. Se folosește de obicei pentru aranjarea butoanelor într-un *Panel*.

Constructori:

FlowLayout() – aliniere centrată şi distanţă implicită de 5 între componente atât pe orizontală cât şi pe verticală;

FlowLayout(int align) – primeşte ca parametru tipul de aliniere (LEFT, RIGHT, CENTER din FlowLayout), distanţa dintre componente fiind implicit 5;

FlowLayout(int align, int hgap, int vgap) – primeşte ca parametri tipul de aliniere (LEFT, RIGHT, CENTER din FlowLayout) şi distanţele orizontale şi verticale dintre componente.




```
public class FlowFrame extends Frame {
  private FlowLayout fLayout = new FlowLayout();
  private Button firstButton = new Button("First Button");
  private Button secondButton = new Button("Second Button");
  private Button thirdButton = new Button("Third Button");
  private Button fourthButton = new Button("Fourth Button");
  private Button fifthButton = new Button("Fifth Button");
  private Button lastButton = new Button("Last Button");

  public FlowFrame() {
    this.setSize(400, 220);
    this.setTitle("FlowLayout");
    this.setLayout(fLayout);
    this.setBackground(new Color(240, 240, 240));
    this.add(firstButton);
    this.add(secondButton);
    this.add(thirdButton);
    this.add(fourthButton);
    this.add(fifthButton);
    this.add(lastButton);
    this.show();
  }

  public static void main(String[] args) {
    new FlowFrame();
  }
}
```

Limbajul Java – interfață grafică (9)

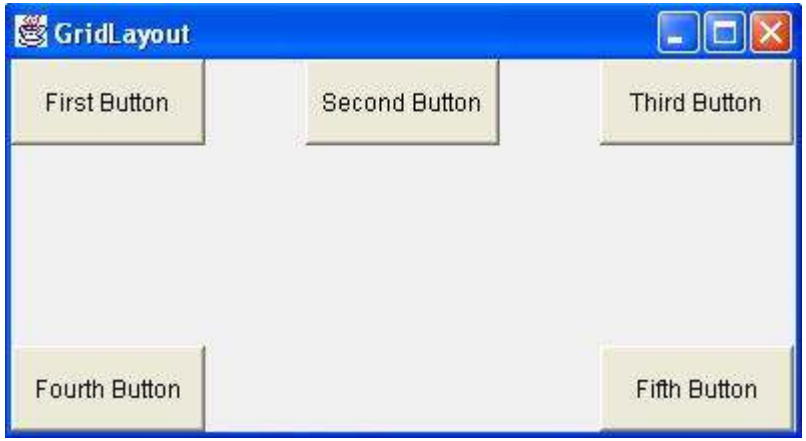
GridLayout împarte containerul în linii și coloane și păstrează componentele în celulele astfel obținute. Toate celulele au aceeași dimensiune. Fiecare componentă este expandată la dimensiunea celulei. În cazul unor interfețe complexe, se pot introduce în celule containere *Panel* cu subcomponente aranjate tot prin *GridLayout*. Pentru obținerea unor spații libere se pot introduce componente *Panel* goale în celulele corespunzătoare.

Constructori:

GridLayout() – creează o singură linie cu câte o coloană pentru fiecare componentă adăugată, fără distanţă între componente;

GridLayout(int rows, int cols) – primeşte ca parametri numărul de linii şi coloane;

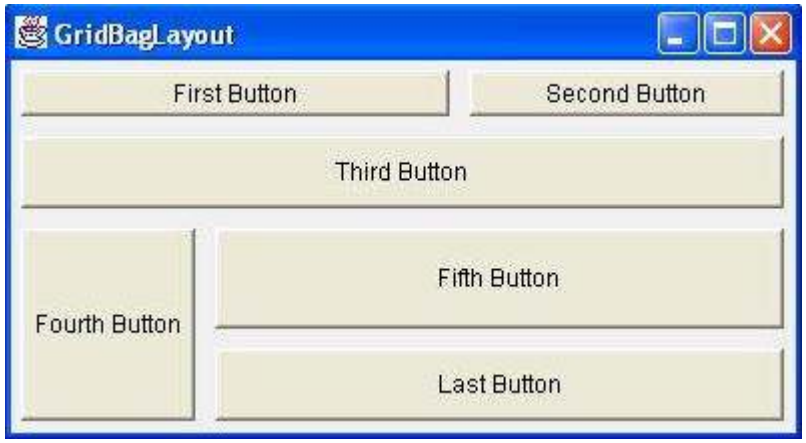
GridLayout(int rows, int cols, int hgap, int vgap) – primeşte ca parametri numărul de linii şi coloane respectiv distanţele orizontale şi verticale dintre componente.

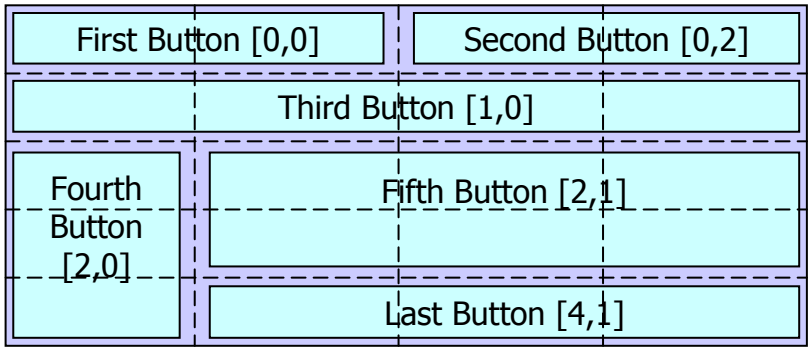



```
public class GridFrame extends Frame {
  private GridLayout gLayout = new GridLayout(2, 3, 50, 100);
  private Button firstButton = new Button("First Button");
  private Button secondButton = new Button("Second Button");
  private Button thirdButton = new Button("Third Button");
  private Button fourthButton = new Button("Fourth Button");
  private Button fifthButton = new Button("Fifth Button");
  private Panel emptyPanel = new Panel();

  public GridFrame() {
    this.setSize(400, 220);
    this.setTitle("GridLayout");
    this.setLayout(gLayout);
    this.setBackground(new Color(240, 240, 240));
    this.add(firstButton);
    this.add(secondButton);
    this.add(thirdButton);
    this.add(fourthButton);
    this.add(emptyPanel);
    this.add(fifthButton);
    show();
  }

  public static void main(String[] args) {
    new GridFrame();
  }
}
```

Limbajul Java – interfață grafică (10)

GridBagLayout împarte containerul în celule care pot avea dimensiuni diferite. O componentă poate ocupa mai multe celule. Exemplu:





Alinierea componentelor:

First Button:
gridy = 0, gridx = 0 (celula [0,0])
gridwidth = 2, gridheight = 1 (două coloane, o linie)
weightx = 0.6 (0.6 din extraspațiul orizontal)
weighty = 0.0 (0.0 din extraspațiul vertical)

Second Button:
gridy = 0, gridx = 2 (celula [0,2])
gridwidth = 2, gridheight = 1 (două coloane, o linie)
weightx = 0.3 (0.3 din extraspațiul orizontal)
weighty = 0.0 (0.0 din extraspațiul vertical)

Third Button:
gridy = 1, gridx = 0 (celula [1,0])
gridwidth = 4, gridheight = 1 (patru coloane, o linie)
weightx = 0.0 (0.0 din extraspațiul orizontal)
weighty = 0.3 (0.3 din extraspațiul vertical)

Fourth Button:
gridy = 2, gridx = 0 (celula [2,0])
gridwidth = 1, gridheight = 3 (o coloană, trei linii)
weightx = 0.0 (0.0 din extraspațiul orizontal)
weighty = 0.3 (0.3 din extraspațiul vertical)

Fifth Button:
gridy = 2, gridx = 1 (celula [2,1])
gridwidth = 3, gridheight = 2 (trei coloane, două linii)
weightx = 0.6 (0.6 din extraspațiul orizontal)
weighty = 0.6 (0.6 din extraspațiul vertical)

Last Button:
gridy = 4, gridx = 1 (celula [4,1])
gridwidth = 3, gridheight = 1 (trei coloane, o linie)
weightx = 0.6 (0.6 din extraspațiul orizontal)
weighty = 0.3 (0.3 din extraspațiul vertical)

Limbajul Java – interfață grafică (11)

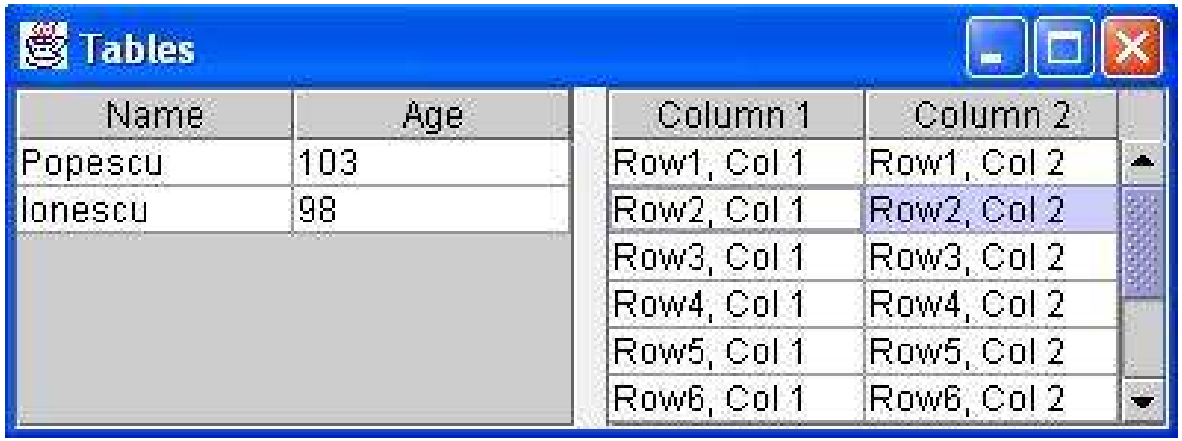
GridBagLayout – codul sursă

```
public class GridBagFrame extends Frame {
  private GridBagLayout gbLayout = new GridBagLayout();
  private GridBagConstraints gbc = new GridBagConstraints();
  private Button firstButton = new Button("First Button");
  private Button secondButton = new Button("Second Button");
  private Button thirdButton = new Button("Third Button");
  private Button fourthButton = new Button("Fourth Button");
  private Button fifthButton = new Button("Fifth Button");
  private Button lastButton = new Button("Last Button");

  public GridBagFrame() {
    this.setSize(400, 220);
    this.setTitle("GridBagLayout");
    this.setLayout(gbLayout);
    this.setBackground(new Color(240, 240, 240));
    gbc.fill = GridBagConstraints.BOTH; //directii de extindere
    gbc.insets = new Insets(5, 5, 5, 5); //dist. fata de marginile celulei
    gbc.gridy = 0; //linia 0
    gbc.gridx = 0; //coloana 0
    gbc.gridwidth = 2; //ocupa doua coloane
    gbc.weightx = 0.6; //0.6 din extraspatiul orizontal
    gbLayout.setConstraints(firstButton, gbc);
    this.add(firstButton);
    gbc.gridy = 0; //linia 0
    gbc.gridx = 2; //coloana 2
    gbc.weightx = 0.3; //0.3 din extraspatiul orizontal
    gbLayout.setConstraints(secondButton, gbc);
    this.add(secondButton);
    gbc.gridy = 1; //linia 1
    gbc.gridx = 0; //coloana 0
    gbc.gridwidth = 4; //ocupa 4 coloane
    gbc.weightx = 0.0; //resetare
    gbc.weighty = 0.3; //0.3 din extraspatiul vertical
    gbLayout.setConstraints(thirdButton, gbc);
    this.add(thirdButton);
    gbc.gridy = 2; //linia 2
    gbc.gridx = 0; //coloana 0
    gbc.gridwidth = 1; //resetare
    gbc.gridheight = 3; //ocupa trei linii
    gbLayout.setConstraints(fourthButton, gbc);
    this.add(fourthButton);
    gbc.gridy = 2; //linia 2
    gbc.gridx = 1; //coloana 1
    gbc.gridwidth = 3; //ocupa trei coloane
    gbc.gridheight = 2; //ocupa doua linii
    gbc.weighty = 0.6; //0.6 din extraspatiul vertical
    gbc.weightx = 0.6; //0.6 din extraspatiul orizontal
    gbLayout.setConstraints(fifthButton, gbc);
    this.add(fifthButton);
    gbc.gridy = 4; //linia 4
    gbc.gridx = 1; //coloana 1
    gbc.gridheight = 1; //resetare
    gbc.weighty = 0.3; //0.3 din extraspatiul vertical
    gbLayout.setConstraints(lastButton, gbc);
    this.add(lastButton);
    show();
  }

  public static void main(String[] args) {
    new GridBagFrame();
  }
}
```

Limbajul Java – interfață grafică (12)

Crearea tabelelor

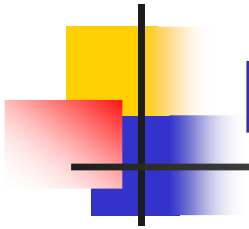
- O clasă care permite crearea tabelelor este *JTable*;
- Gestionarea componentelor *JTable* se poate face prin clasa *DefaultTableModel*;
- Pentru o funcționare corectă cu scroll, obiectul *JTable* trebuie adăugat pe un *JScrollPane* și nu pe *ScrollPane*!
- Aplicația următoare folosește două tabele, unul preîncărcat, iar celălalt cu încărcare dinamică a datelor:



Limbajul Java – interfață grafică (13)

Codul sursă al aplicației cu tabele

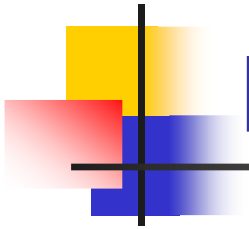
```
public class TableFrame extends Frame {
  private GridLayout gLayout = new GridLayout(1, 2, 10, 10);
  private JScrollPane leftScrollPane = new JScrollPane();
  private JScrollPane rightScrollPane = new JScrollPane();
  private String leftTableColumns[] = {"Name", "Age"}; //header
  private Object leftTableData[][] = {{"Popescu", "103"}, {"Ionescu", "98"}}; //continut
  private JTable leftTable = new JTable(leftTableData, leftTableColumns); //tabel cu date preincarcate
  private JTable rightTable = new JTable(); //tabel gol
  private String rightTableColumns[] = {"Column 1", "Column 2"}; //header
  private DefaultTableModel rightTableModel = new DefaultTableModel(rightTableColumns, 0);

  public TableFrame() {
    this.setSize(400, 150);
    this.setLayout(gLayout);
    this.setTitle("Tables");
    this.setBackground(new Color(240, 240, 240));
    this.add(leftScrollPane);
    this.add(rightScrollPane);
    leftScrollPane.getViewport().add(leftTable, null);
    rightScrollPane.getViewport().add(rightTable, null);
    rightTable.setModel(rightTableModel);
    for(int i=1; i<=10; i++){
      String row[] = {"Row" + i + ", Col 1", "Row" + i + ", Col 2"};
      rightTableModel.addRow(row);
    }
    show();
  }

  public static void main(String[] args) {
    new TableFrame();
  }
}
```



Limbajul Java – interfață grafică (14)

Aplicații

1. Introduceți în fereastra aplicației care gestionează lista de studenți un buton *Clear* care să permită ștergerea conținutului listei. Ştergerea conţinutului componentei *List* (din pachetul AWT) se poate face prin metodele *clear* sau *removeAll*.
2. Modificați funcția de adăugare în listă astfel încât aceasta să păstreze lista sortată la introducerea unui nou student. Preluarea conţinutului componentei *List* (din AWT) într-un tablou de tip *String* se poate efectua prin metoda *getItems*, iar preluarea liniei de pe o anumită poziţie se poate face prin metoda *getItem* cu un parametru de tip *int*, reprezentând poziţia în listă. Inserarea pe o poziţie în listă se poate efectua prin metoda *add* cu un parametru *String* şi unul de tip *int*, reprezentând poziţia. De asemenea, *getItemCount* returnează numărul de linii din listă.
3. Înlocuiţi lista din aplicaţia prezentată cu un tabel.
4. Dezvoltaţi aplicaţia astfel încât, pe lângă nume, să se introducă în tabel adresa, vârsta şi media studentului.
5. Rearanjați componentele pe fereastră prin *GridLayout* sau *GridBagLayout* și setați fereastra redimensionabilă.



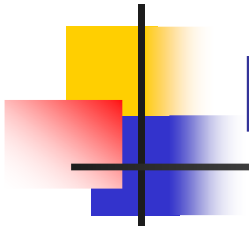
Limbajul Java – fire de execuție (1)

- Un fir de execuție (*thread*) este o secvență de instrucțiuni executată de un program.
- Firul de execuție primar este metoda *main* și atunci când acesta se termină, se încheie și programul.
- Un program poate utiliza fire multiple care sunt executate concurențial.
- Utilizarea firelor multiple este utilă în cazul programelor complexe care efectuează seturi de activități multiple.
- Fiecare fir de execuție are o prioritate care poate lua valori de la 1 (prioritate mică) la 10 (prioritate maximă). Firele cu prioritate mare sunt avantajate la comutare, ele primesc mai des controlul procesorului.
- Pentru implementarea unui fir de execuţie în Java, se poate extinde clasa *Thread*. Deoarece Java nu acceptă moştenirea multiplă, în cazul în care a fost deja extinsă o clasă, pentru crearea unui fir de execuţie trebuie implementată interfaţa *Runnable*. Indiferent de metoda utilizată, se suprascrie metoda *run* care trebuie să conţină instrucţiunile firului.
- Aplicaţia următoare porneşte două fire de execuţie: unul pentru afişarea numerelor şi celălalt pentru afişarea literelor. Pentru a observa diferenţele dintre cele două metode de implementare, firul de execuţie *Numbers* extinde clasa *Thread*, în timp ce *Letters* implementează interfaţa *Runnable*.

Limbajul Java – fire de execuție (2)

Afișare concurențială de numere și litere:

```
public class Numbers extends Thread {          //extinde clasa Thread
  public void run(){
    for(int i=0; i<1000; i++){
      System.out.println(i);
    }
  }
}

public class Letters implements Runnable {     //implementeaza interfata Runnable
  char a = 'a';
  public void run(){
    for(int i=0; i<1000; i++){
      int c = a + i%26;
      System.out.println((char)c);
    }
  }
}

public class Main {
  public static void main(String[] args) {
    Numbers numbers = new Numbers();
    Thread  letters = new Thread(new Letters());
    letters.start();
    numbers.start();
  }
}
```



Limbajul Java – fire de execuție (3)

- Principalele operații care pot fi efectuate după crearea unui fir de execuție:
  - **start** – pornește firul de execuție
  - **setPriority** – setează prioritatea firului de execuție (valoare între 1 și 10)
  - **getPriority** – returnează prioritatea firului de execuție (valoare între 1 și 10)
  - **sleep** – suspendă execuția unui fir pentru o perioadă precizată în milisecunde.
  - **yield** – suspendă temporar execuția firului curent în favoarea altor fire de execuție.
- Se recomandă oprirea firului prin încheierea execuției metodei **run**.
- Următoarea aplicație folosește un fir de execuție pentru deplasarea unei mingi pe orizontală.

Limbajul Java – fire de execuție (4)

```
public class Ball extends Thread {
    private int px = 0;              //pozitia x
    private int py = 0;              //pozitia y
    private int size = 0;
    private Color color = null;
    private MyFrame parent = null;

    public Ball(MyFrame parent, int px, int py, int size, Color color) {
        this.parent = parent;        //referinta la fereastra
        this.px = px;
        this.py = py;
        this.size = size;
        this.color = color;
    }

    public int getPX(){
        return px;
    }

    public int getPY(){
        return py;
    }

    public int getSize(){
        return size;
    }

    public Color getColor(){
        return color;
    }

    public void run(){
        while(px < parent.getSize().width){          //iesire din fereastra
            px++;
            parent.paint();
        }
    }
}
```

Limbajul Java – fire de execuție (5)

```
public class MyFrame extends Frame {
    Ball ball = null;
    Image buffer = null;

    public MyFrame() {
        this.setSize(new Dimension(400, 300));
        this.setTitle("Balls");
        this.addWindowListener(new java.awt.event.WindowAdapter() {          //detectia evenimentului de inchidere a ferestrei
            public void windowClosing(WindowEvent e) {
                this_windowClosing(e);
            }
        });
        setVisible(true);
        buffer = createImage(getSize().width, getSize().height);
        ball = new Ball(this, 20, 50, 20, Color.red);
        ball.start();
    }

    void paint(){
        Graphics gbuffer = buffer.getGraphics();
        //se deseneaza mai intai in buffer (tehnica Double Buffering)
        gbuffer.setColor(Color.white);
        gbuffer.fillRect(0, 0, getSize().width, getSize().height);
        gbuffer.setColor(ball.getColor());
        gbuffer.fillOval(ball.getPX(), ball.getPY(), ball.getSize(), ball.getSize());
        paint(gbuffer);
        //se copiaza imaginea din buffer pe fereastra (tehnica Double Buffering)
        Graphics g = getGraphics();
        g.drawImage(buffer, 0, 0, getSize().width, getSize().height, 0, 0, getSize().width, getSize().height, this);
    }

    void this_windowClosing(WindowEvent e) {                                 //evenimentul de inchidere a ferestrei
        System.exit(0);
    }

    public static void main(String[] args){
        new MyFrame();
    }
}
```

Limbajul Java – fire de execuție (6)

Sincronizarea firelor de execuție

- Dacă în cadrul unui program Java există un fir de execuţie care creează (produce) date şi un al doilea fir de execuţie care le prelucrează (consumă), de regulă se declară un bloc *synchronized*, ceea ce permite ca un singur fir să aibă acces la resurse (metode, date) la un moment dat.
- Atunci când un fir de execuţie apelează *wait* în cadrul unui bloc de cod *synchronized*, alt fir poate accesa codul.
- Iar atunci când un fir de execuţie încheie procesarea codului *synchronized*, el apelează metoda *notify* pentru a anunţa alte fire de execuţie să înceteze aşteptarea.
- Reluăm în continuare aplicația care afișează numere și litere sincronizând cele două fire de execuție.

Limbajul Java – fire de execuție (7)

```
public class Numbers extends Thread {
  Main m;
  public Numbers(Main m){
    this.m = m;
  }
  public void run(){
    m.numbers();
  }
}

public class Letters implements Runnable {
 Main m;
 public Letters(Main m){
   this.m = m;
 }
 public void run(){
   m.letters();
 }
}

public class Main {
  public Main(){
    Numbers numbers = new Numbers(this);
    Thread  letters = new Thread(new Letters(this));
    letters.start();
    numbers.start();
  }
```

```
public synchronized void numbers(){
   for(int i=0; i<1000; i++){
     if(i%26==0){
     try {
       this.notify();
       this.wait();
     }
     catch (InterruptedException ex) { }
     }
     System.out.println(i);
   }
   this.notify();
 }

public synchronized void letters(){
   char a = 'a';
   for(int i=0; i<1000; i++){
     if(i%26==0){
       try {
         this.notify();
         this.wait();
       }
       catch (InterruptedException ex) { }
     }
     int c = a + i%26;
     System.out.println((char)c);
   }
  this.notify();
 }

 public static void main(String[] args) {
  new Main();
 }
}
```

Limbajul Java – fire de execuție (8)

Aplicații propuse:

1. Introducerea unei întârzieri de 10 ms în algoritmul de mișcare a bilei din aplicația prezentată.
2. Modificarea aplicației astfel încât să permită pornirea mai multor mingi simultan.
3. Să se înlocuiască algoritmul de deplasare a mingii pe orizontală cu următorul algoritm de mișcare:

```
dx = 1;
dy = 1;
while (true){
   px += dx;
   py += dy;
   if((px <= 0) || (px >= frame_width))
      dx = -dx;
   if((py <= 0) || (py >= frame_height))
      dy = -dy;
   paint();
}
```

Un alt algoritm de mișcare:

```
gravy = 1;
speed = -30
speedy = -30;
speedx = 0;
while(true){
   speedy += gravy;
   py += speedy;
   px += speedx;
   if(py > frameheight){
      speedy = speed;
      speed += 3;
   }
   if(speed == 0) break;
   paint();
}
```

Limbajul Java – colecții (1)

O colecție grupează date într-un singur obiect și permite manipularea acestora.

Clasele Vector și ArrayList

- Permit implementarea listelor redimensionabile cu elemente referință de orice tip, inclusiv *null*.
- O diferență importantă: *Vector* este sincronizat, iar *ArrayList* este nesincronizat, ceea ce-l face mai rapid. Dacă un *ArrayList* este accesat și modificat de mai multe fire de execuție concurențial, necesită sincronizare externă (a obiectului care încapsulează lista). O altă soluție constă în crearea unei instanțe sincronizate, prevenind astfel eventualele accese nesincronizate:

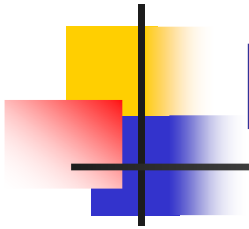
  List list = Collections.synchronizedList(new ArrayList(...));
- Clasa Vector s-a păstrat pentru compatibilitate cu versiuni JDK mai vechi de 1.2.
- Exemple

```
Vector vector = new Vector();                    //vector gol
vector.addElement(new Integer(1));               //vector = {1}
vector.add(new Integer(3));                      //vector = {1, 3}
vector.insertElementAt(new Integer(5), 1);       //vector = {1, 5, 3}
vector.setElementAt(new Integer(2), 1);          //vector = {1, 2, 3}

ArrayList list = new ArrayList();                //lista goala
list.add(new Integer(1));                        //list = {1}
list.add(new Integer(3));                        //list = {1, 3}
list.add(1, new Integer(5));                     //list = {1, 5, 3}
list.set(1, new Integer(2));                     //list = {1, 2, 3}
```

Limbajul Java – colecții (2)

- Începând cu versiunea JDK 1.8, listele au metodă *sort*. În exemplul următor criteriul de sortare este furnizat printr-o expresie lambda. Afișarea poate fi și ea realizată cu ajutorul unei expresii lambda, apelând metoda *forEach* a listei, disponibilă tot de la JDK 1.8.

```
public class Person{
    private String name;
    private int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public static void main(String[] args) {
        ArrayList list = new ArrayList();
        list.add(new Person("Popescu", 103));
        list.add(new Person("Ionescu", 98));
        list.add(new Person("Petrescu", 49));
        list.sort((o1,o2)->((Person)o1).name.compareTo(((Person)o2).name));
        //{Ionescu-98, Petrescu-49, Popescu-103}
        list.forEach((o)->System.out.println(((Person)o).name+" "+((Person)o).age));
    }
}
```



Limbajul Java – colecții (3)

- Pentru evitarea conversiei repetate din *Object* în Person, la crearea colecției se poate preciza tipul componentelor. Astfel, exemplul anterior s-ar rescrie în felul următor:

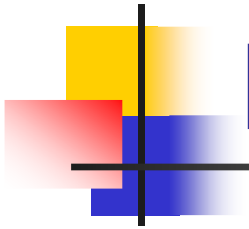
```
public class Person{
    private String name;
    private int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public static void main(String[] args) {
        ArrayList<Person> list = new ArrayList();
        //ArrayList<Person> list = new ArrayList<Person>();
        list.add(new Person("Popescu", 103));
        list.add(new Person("Ionescu", 98));
        list.add(new Person("Petrescu", 49));
        list.sort((o1,o2)->o1.name.compareTo(o2.name));
        //{Ionescu-98, Petrescu-49, Popescu-103}
        list.forEach((o)->System.out.println(o.name+" "+o.age));
    }
}
```

Limbajul Java – colecții (4)

Clasa Stack

- Permite implementarea sincronizată a stivelor
- Extinde clasa Vector cu operațiile specifice stivei:
  - push – introduce un element în vârful stivei;
  - pop – scoate elementul din vârful stivei;
  - peek – preia (citește) elementul din vârful stivei, fără să-l scoată;
  - empty – verifică dacă stiva e goală;
  - search – returnează poziția unui obiect din stivă față de vârf.
- Exemple:

```
Stack stack = new Stack();                              //stiva goala
System.out.println(stack.empty());                      //true
stack.push(new Integer(1));                             //stack = {1}
stack.push(new Integer(2));                             //stack = {1, 2}
stack.push(new Integer(3));                             //stack = {1, 2, 3}
System.out.println(stack.search(new Integer(3)));       //1
System.out.println(stack.search(new Integer(2)));       //2
System.out.println(stack.empty());                      //false
System.out.println(stack.pop());                        //3, stack = {1, 2}
System.out.println(stack.peek());                       //2, stack = {1, 2}
System.out.println(stack.pop());                        //2, stack = {1}
System.out.println(stack.pop());                        //1, stiva goala
System.out.println(stack.empty());                      //true
```



Limbajul Java – colecții (5)

Clasa LinkedList

- Permite implementarea stivelor și a cozilor.
- Diferența față de Stack: LinkedList este nesincronizat (deci mai rapid). Nefiind sincronizat, LinkedList necesită sincronizare externă în caz de accesare concurențială multifir.
- Exemple:

```
LinkedList stack = new LinkedList();                //stiva goala
stack.addLast(new Integer(1));                      //stack = {1}
stack.addLast(new Integer(2));                      //stack = {1, 2}
stack.addLast(new Integer(3));                      //stack = {1, 2, 3}
System.out.println(stack.removeLast());             //3, stack = {1, 2}
System.out.println(stack.getLast());                //2, stack = {1, 2}
System.out.println(stack.removeLast());             //2, stack = {1}
System.out.println(stack.removeLast());             //1, stiva goala

LinkedList queue = new LinkedList();                //coada goala
queue.addFirst(new Integer (1));                    //queue = {1}
queue.addFirst(new Integer(2));                     //queue = {2, 1}
queue.addFirst(new Integer(3));                     //queue = {3, 2, 1}
System.out.println(queue.removeLast());             //1, queue = {3, 2}
System.out.println(queue.removeLast());             //2, queue = {3}
System.out.println(queue.removeLast());             //3, coada goala
```

Limbajul Java – colecții (6)

Clasele HashSet, TreeSet și LinkedHashSet

- Interfața Set permite folosirea mulțimilor – colecții de date în care elementele sunt unice.
- Clasa **HashSet** păstrează elementele adăugate în ordinea stabilită de o funcție de dispersie, asigurând astfel operații de introducere și preluare a unui element rapide, de complexitate O(1). Funcția de dispersie poate fi schimbată prin redefinirea metodei *hashCode()*.
- Clasa **TreeSet** păstrează elementele sortate în ordine crescătoare, ceea ce determină operații de introducere și preluare a unui element mai lente, de complexitate O(log n).
- Clasa **LinkedHashSet** (disponibilă începând cu versiunea JDK 1.4) păstrează elementele în ordinea în care au fost introduse asigurând operații de accesare a unui element de complexitate O(1).
- Exemple

```
HashSet hs = new HashSet();                  //hs = {}
hs.add(new Integer(1));                      //hs = {1}
hs.add(new Integer(2));                      //hs = {2, 1}
hs.add(new Integer(4));                      //hs = {2, 4, 1}
hs.add(new Integer(2));                      //hs = {2, 4, 1}

TreeSet ts = new TreeSet(hs);                //ts = {1, 2, 4}
ts.add(new Integer(3));                      //ts = {1, 2, 3, 4}

LinkedHashSet lhs = new LinkedHashSet();     //lhs = {}
lhs.add(new Integer(1));                     //lhs = {1}
lhs.add(new Integer(2));                     //lhs = {1, 2}
lhs.add(new Integer(4));                     //lhs = {1, 2, 4}
lhs.add(new Integer(3));                     //lhs = {1, 2, 4, 3}
```

Limbajul Java – colecții (7)

Implementarea interfeței Comparable pentru obiectele încărcate în TreeSet

- Obiectele încărcate în TreeSet sunt sortate prin intermediul funcției *compareTo.* De aceea, clasa obiectelor încărcate trebuie să implementeze interfața Comparable.
- Proprietatea de set (mulțime) se aplică câmpului după care se face comparația în *compareTo*. Dacă valoarea acelui câmp este aceeași în două obiecte, chiar dacă celelalte câmpuri diferă, în set rămâne doar unul din obiecte. De aceea, dacă în *compareTo* valorile câmpului comparat sunt egale se poate ține cont de celelalte câmpuri, aplicând astfel proprietatea de set pe întregul obiect.
- Exemplu:

```
public class Person implements Comparable{
  private String name;
  private int age;

  public Person(String name, int age) {
    this.name = name;
    this.age = age;
  }

  public int compareTo(Object p){
    return this.name.compareTo(((Person)p).name);
  }
```

```
  public static void main(String[] args) {
    TreeSet ts = new TreeSet();
    ts.add(new Person("Popescu", 103));
    //{Popescu-103}
    ts.add(new Person("Ionescu", 98));
    //{Ionescu-98, Popescu-103}
    ts.add(new Person("Petrescu", 49));
    //{Ionescu-98, Petrescu-49, Popescu-103}
  }
}
```

Limbajul Java – colecții (8)

- Criteriul de comparație poate fi furnizat și printr-o expresie lambda, caz în care clasa Person nu mai trebuie să implementeze interfața *Comparable*:

```
public class Person{
    private String name;
    private int age;
    public Person(String name, int age) {
        this.name = name;
        this.age = age;
    }
    public static Comparator alphabeticNames = (p1,p2) -> {
        return ((Person)p1).name.compareTo(((Person)p2).name);
    };
    public static void main(String[] args) {
        TreeSet ts = new TreeSet(Person.alphabeticNames);
        ts.add(new Person("Popescu", 103));
        ts.add(new Person("Ionescu", 98));
        ts.add(new Person("Petrescu", 49));
    }
}
```

Limbajul Java – colecții (9)

- Conținutul colecției *ts* de tip *TreeSet* din aplicația prezentată mai sus, se poate afișa printr-o buclă *for* îmbunătățită, disponibilă începând cu JDK 1.5:

```
TreeSet<Person> ts = new TreeSet(Person.alphabeticNames);
ts.add(new Person("Popescu", 103));
ts.add(new Person("Ionescu", 98));
ts.add(new Person("Petrescu", 49));
//{Ionescu-98, Petrescu-49, Popescu-103}
for(Person o: ts)
    System.out.println(o.name + " " + o.age);
```

- O altă variantă de afișare a conținutului colecției constă în apelul metodei *forEach* (disponibilă începând cu JDK 1.8), cu o expresie lambda:

```
TreeSet<Person> ts = new TreeSet(Person.alphabeticNames);
ts.add(new Person("Popescu", 103));
ts.add(new Person("Ionescu", 98));
ts.add(new Person("Petrescu", 49));
//{Ionescu-98, Petrescu-49, Popescu-103}
ts.forEach((o)->System.out.println(o.name + " " + o.age));
```

Limbajul Java – colecții (10)

Iteratori

- Iteratorii permit traversarea în ordine a colecțiilor de date.
- Exemplul 1 (aplicația anterioară):

```
TreeSet<Person> ts = new TreeSet(Person.alphabeticNames);
ts.add(new Person("Popescu", 103));
ts.add(new Person("Ionescu", 98));
ts.add(new Person("Petrescu", 49));
//{Ionescu-98, Petrescu-49, Popescu-103}
Iterator<Person> i = ts.iterator();
while(i.hasNext()){
    Person o = i.next();
    System.out.println(o.name + " " + o.age);
}
```

- Exemplul 2 (colecție de tip String):

```
TreeSet ts = new TreeSet();
ts.add("Romania");
ts.add("Anglia");
ts.add("Germania");
ts.add("Franta");
Iterator i = ts.iterator();
while(i.hasNext()) System.out.println(i.next());  //{Anglia, Franta, Germania, Romania}
```

Limbajul Java – colecții (11)

Clasele HashMap, TreeMap și LinkedHashMap

- Interfața Map permite păstrarea de perechi cheie-valoare, ambele de tip referință, cu chei unice. Principalele operații:
  - put(k, v) – asociază cheii k valoarea v. Dacă există deja cheia k atunci vechea valoare se înlocuiește cu v.
  - get(k) – returnează valoarea corespunzătoare cheii k.
- Clasa **HashMap** păstrează cheile în ordinea stabilită de o funcție de dispersie asigurând astfel accesări rapide, de complexitate O(1).
- Clasa **TreeMap** păstrează cheile ordonate crescător, accesarea fiind astfel mai lentă, de complexitate O(log n).
- Clasa **LinkedHashMap** (disponibilă începând cu JDK 1.4) păstrează cheile în ordinea în care au fost introduse asigurând accesări de complexitate O(1).
- Exemple

```
HashMap hm = new HashMap();
hm.put("Romania", "Bucuresti");
hm.put("Franta", "Paris");
hm.put("Germania", "Bonn");
hm.put("Germania", "Berlin");
System.out.println(hm);                    //{Romania=Bucuresti, Germania=Berlin, Franta=Paris}
System.out.println(hm.get("Germania"));    //Berlin

TreeMap tm = new TreeMap(hm);
System.out.println(tm);                    //{Franta=Paris, Germania=Berlin, Romania=Bucuresti}

LinkedHashMap lhm = new LinkedHashMap();
lhm.put("Romania", "Bucuresti");
lhm.put("Franta", "Paris");
lhm.put("Germania", "Berlin");
System.out.println(lhm);                   //{Romania=Bucuresti, Franta=Paris, Germania=Berlin}
```

Limbajul Java – colecții (12)

Implementarea interfeţei Comparable pentru cheile încărcate în TreeMap

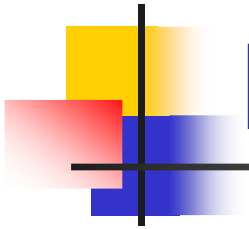
- Perechile cheie-valoare încărcate în TreeMap sunt sortate în ordinea crescătoare a cheilor, prin intermediul funcției *compareTo.* De aceea, clasa obiectelor cheie trebuie să implementeze interfața Comparable. Exemplu:

```
public class Person implements Comparable{
  private String name;
  private int age;

  public Person(String name, int age) {
    this.name = name;
    this.age = age;
  }

  public int compareTo(Object p){
    return this.name.compareTo(((Person)p).name);
  }

  public static void main(String[] args) {
    TreeMap tm = new TreeMap();
    tm.put(new Person("Pop", 54), "Romania");          //tm={Pop-54-Romania}
    tm.put(new Person("Brown", 98), "Anglia");         //tm={Brown-98-Anglia, Pop-54-Romania}
    tm.put(new Person("Schmitt", 49), "Germania");     //tm={Brown-98-Anglia, Pop-54-Romania, Schmitt-49-Germania}
  }
}
```

Limbajul Java – colecții (13)

- Pentru evitarea conversiei repetate a cheilor din *Object* în Person respectiv a valorilor din Object în String, la crearea colecției se poate preciza tipul cheilor și a valorilor. Astfel, exemplul anterior, cu afișarea inclusă, s-ar rescrie în felul următor:

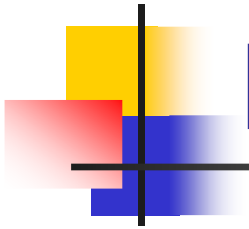
```
TreeMap<Person, String> tm = new TreeMap();
//TreeMap<Person, String> tm = new TreeMap<Person, String>();
Person p = new Person("Pop", 54);
tm.put(p, "Romania");
//tm={Pop-54-Romania}
Person q = new Person("Brown", 98);
tm.put(q, "Anglia");
//tm={Brown-98-Anglia, Pop-54-Romania}
Person r = new Person("Schmitt", 49);
tm.put(r, "Germania");
//tm={Brown-98-Anglia, Pop-54-Romania, Schmitt-49-Germania}
for(Map.Entry<Person,String> entry : tm.entrySet()) {
    Person k = entry.getKey();
    String v = entry.getValue();
    System.out.println(k.getName()+ "-" +k.getAge()+ "-" +v);
}
```



Limbajul Java – colecții (14)

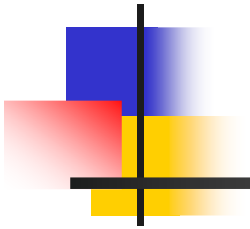
Aplicații propuse

1. Să se modifice programele prezentate la TreeSet și TreeMap astfel încât sortarea persoanelor să se facă după vârstă în cazul în care au același nume.
2. Definiți clasa **Course** cu câmpurile **title**, **teacher** (profesor), **credits** (nr. credite) și **year** (an de studiu). Introduceţi de la tastatură cinci obiecte de tip **Course** într-un **TreeSet** şi afişaţi cursul cu cel mai mic număr de credite.
3. Definiți clasa **Course** cu câmpurile **title**, **teacher** (profesor), **credits** (nr. credite) și **year** (an de studiu). Introduceţi de la tastatură cinci obiecte de tip **Course** într-un **TreeSet** şi afişaţi cursul cu cel mai mare număr de credite.
4. Definiți clasa **Course** cu câmpurile **title**, **teacher** (profesor), **credits** (nr. credite) și **year** (an de studiu). Introduceţi de la tastatură cinci obiecte de tip **Course** într-un **TreeSet** şi afişaţi-le în ordinea descrescătoare a numărului de credite.
5. Definiți clasa **Course** cu câmpurile **title**, **teacher** (profesor), **credits** (nr. credite) și **year** (an de studiu). Introduceţi de la tastatură cinci obiecte de tip **Course** într-un **TreeSet** şi afişaţi-le în ordinea crescătoare a anului de studiu.
6. Definiți clasa **Course** cu câmpurile **title**, **teacher** (profesor), **credits** (nr. credite) și **year** (an de studiu). Introduceţi de la tastatură cinci obiecte de tip **Course** într-un **TreeSet** şi afişaţi-le în ordinea alfabetică a titlului.
7. Definiți clasa **Course** cu câmpurile **title**, **teacher** (profesor), **credits** (nr. credite) și **year** (an de studiu). Introduceţi de la tastatură cinci obiecte de tip **Course** într-un **TreeSet** şi afişaţi-le în ordinea alfabetică a profesorului.



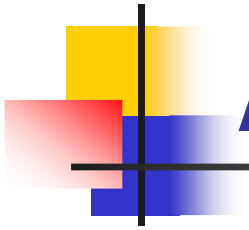
Limbajul Java – test

Test (T) susținut la laborator din aplicațiile propuse la laborator în cadrul capitolului *Limbajul Java* – se va implementa și rula pe calculator o aplicație care implică o moștenire respectiv folosirea unor colecții de date.



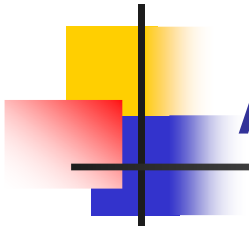
Partea II

Analiza algoritmilor



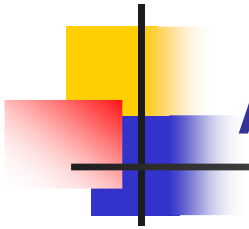
Analiza algoritmilor

- Algoritmul este o secvenţă de operaţii care transformă mulţimea datelor de intrare în datele de ieşire.
- Analiza algoritmului se referă la evaluarea performanţelor acestuia (timp de execuţie, spaţiu de memorie). Scopul este acela de a găsi algoritmul cel mai eficient pentru o anumită problemă.
- La analiza algoritmilor trebuie avut în vedere modelul de implementare (secvențială, paralelă) și arhitectura mașinii de calcul (superscalar, multithreading, multicore) respectiv dacă aceasta permite sau nu execuția speculativă a instrucțiunilor.



Analiza algoritmilor – mașina Turing

- Conjectura Church-Turing postulează că pentru orice algoritm există o mașină Turing echivalentă. Cu alte cuvinte, nicio procedură de calcul nu este algoritm dacă nu poate fi executată de o mașină Turing. În esență, mașina Turing reprezintă ceea ce astăzi numim algoritm. Așadar, o mașină Turing este echivalentă cu noțiunea de algoritm.
- O mașină Turing constă din:
  - O bandă împărțită în celule, fiecare conținând un simbol (inclusiv cel vid) dintr-un alfabet finit.
  - Un cap care poate scrie și citi și care se poate deplasa la stânga sau la dreapta.
  - Un registru de stare care stochează starea mașinii Turing. Numărul stărilor este finit și există o stare inițială.
  - O funcție de tranziție (tabelă de acțiuni) care, în funcție de starea curentă și simbolul citit, determină ce simbol să se scrie, cum să se deplaseze capul (la stânga sau la dreapta) și care va fi noua stare.
- Mașina Turing se oprește dacă se ajunge într-o stare finală sau dacă nu există combinația de simbol-stare în tabela de acțiuni.



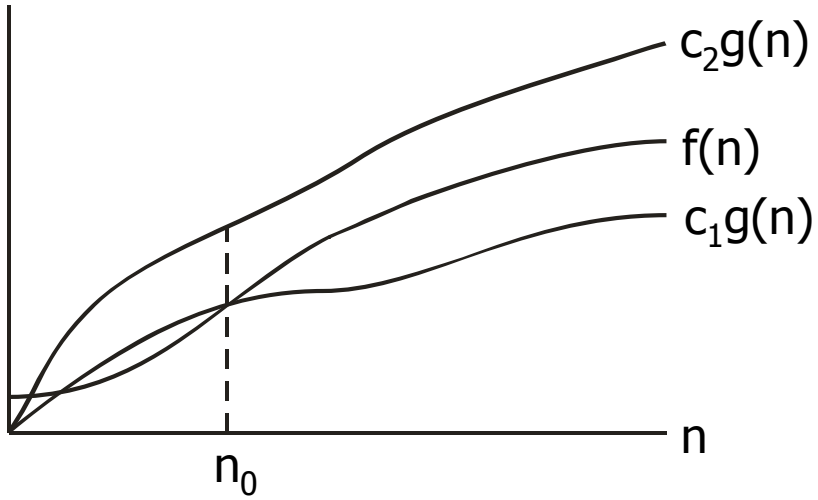
Analiza algoritmilor – complexitate (1)

- Complexitatea unui algoritm, exprimată în funcție de dimensiunea datelor de intrare (*n*), are următoarele componente:
  - Complexitate temporală – numărul de operații necesare pentru rezolvarea unei probleme
  - Complexitate spațială – spațiul de memorie utilizat
- Complexitatea temporală este indicatorul principal al performanței unui algoritm.
- Complexitatea temporală a unui algoritm poate fi determinată experimental (prin măsurarea timpului de rulare) sau prin analiză teoretică (prin aproximarea numărului de operații primitive).
- Complexitatea unui algoritm este de regulă exprimată (aproximată) prin notații asimptotice [Cor00] care reține doar termenul dominant al expresiei, ceilalți termeni devenind neglijabili pentru valori mari ale lui n. Exemple:
  - Dacă execuția unui algoritm necesită 2^n+n^5+8 operații, atunci complexitatea temporală este 2^n;
  - Un algoritm cu $7n^4+10n^2+1000n$ operații elementare are complexitate n^4.

Analiza algoritmilor – complexitate (2)

Θ-notația

$$\Theta(g(n)) = \{f(n) \mid \exists c_1, c_2, n_0 > 0 \ \ a.\hat{i}. \ 0 \le c_1 \cdot g(n) \le f(n) \le c_2 \cdot g(n), \ \forall \ n \ge n_0\}$$

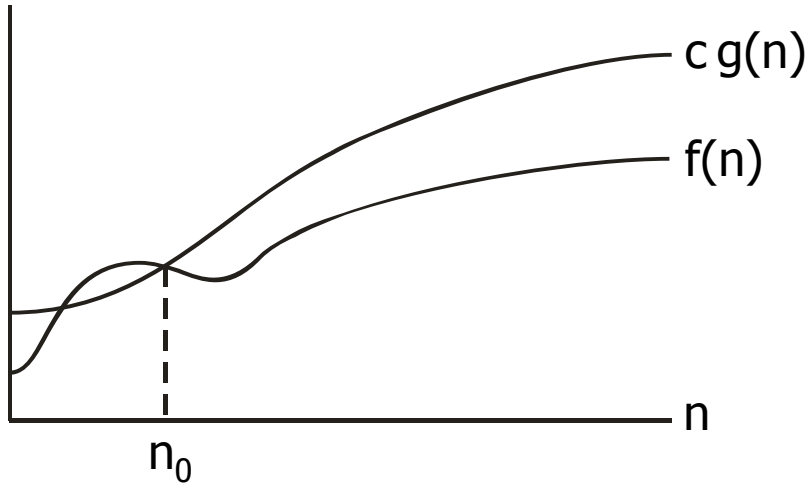


Θ-notația delimitează o funcție asimptotic inferior și superior, g(n) este o margine asimptotic tare pentru f(n).

Analiza algoritmilor – complexitate (3)

O-notația

$$O(g(n)) = \{f(n) \mid \exists c, n_0 > 0 \ \ a.î.\ 0 \le f(n) \le c \cdot g(n),\ \forall\ n \ge n_0\}$$

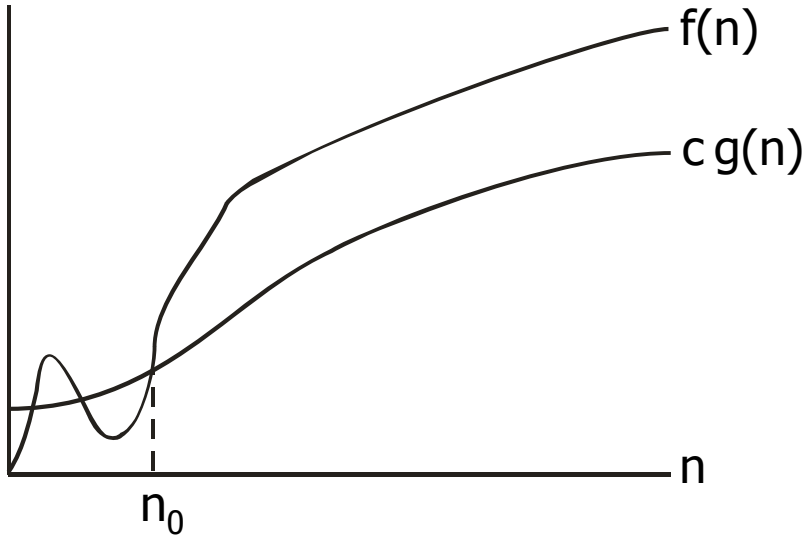


O-notația delimitează o funcție asimptotic superior, g(n) este o margine asimptotică superioară.

Analiza algoritmilor – complexitate (4)

Ω-notația

$$\Omega(g(n)) = \{f(n) \mid \exists c, n_0 > 0 \ \ a.î. \ 0 \le c \cdot g(n) \le f(n), \ \forall \ n \ge n_0\}$$



Ω-notația delimitează o funcție asimptotic inferior, g(n) este o margine asimptotică inferioară.

Analiza algoritmilor – complexitate (5)

- **o-notația:** Delimitarea asimptotică superioară dată de notația O poate sau nu să fie o delimitare asimptotic strânsă. De exemplu, $2n^2=O(n^2)$ este o delimitare asimptotic strânsă, pe când $2n^2=O(n^3)$ nu este. Vom folosi o-notația pentru a desemna o delimitare superioară care nu este asimptotic strânsă: $2n^2=o(n^3)$.
$$o(g(n)) = \{f(n) \mid \forall c > 0, \exists n_0 > 0 \ \ a.î. \ 0 \le f(n) < c \cdot g(n), \forall \ n \ge n_0\}$$

- **ω-notația** desemnează o delimitare asimptotică inferioară care nu este asimptotic strânsă: $2n^2=\omega(n)$.
$$\omega(g(n)) = \{f(n) \mid \forall c > 0, \exists n_0 > 0 \ \ a.î. \ 0 \le c \cdot g(n) < f(n), \forall \ n \ge n_0\}$$

- **Teoremă:** Pentru orice două funcții f(n) și g(n), avem $f(n)=\Theta(g(n))$ dacă și numai dacă $f(n)=O(g(n))$ și $f(n)=\Omega(g(n))$.

- **Convenție:** În orice ecuație în care se folosesc notații asimptotice membrul drept oferă un nivel de detaliu mai scăzut decât membrul stâng. Exemplu: $2n^2+3n+1 = 2n^2+\Theta(n) = \Theta(n^2)$. Putem scrie $2n^2+3n+1 = \Theta(n^2)$, dar nu scriem niciodată $\Theta(n^2) = 2n^2+3n+1$.

Analiza algoritmilor – complexitate (6)

Alte exemple [Knu00]

- Știm că

$$1^2 + 2^2 + \ldots + n^2 = \frac{n(n+1)(2n+1)}{6} = \frac{1}{3}n^3 + \frac{1}{2}n^2 + \frac{1}{6}n \qquad (1)$$

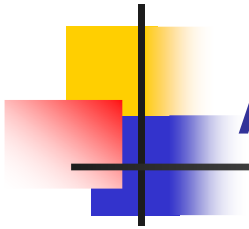
- Rezultă că

$$1^2 + 2^2 + \ldots + n^2 = O(n^4) \qquad (2)$$

$$1^2 + 2^2 + \ldots + n^2 = O(n^3) \qquad (3)$$

$$1^2 + 2^2 + \ldots + n^2 = \frac{1}{3}n^3 + O(n^2) \qquad (4)$$

- Ecuația (2) este destul de nerafinată, dar nu incorectă; ecuația (3) este mai puternică, iar ecuația (4) și mai puternică.



Analiza algoritmilor – complexitate (7)

Ordinul de complexitate

Constantă	O(1)
Logaritmică	O(log n)
Liniară	O(n)
Liniar logaritmică	O(n log n)
Pătratică	$O(n^2)$
Polinomială (p>2)	$O(n^p)$
Exponențială	$O(a^n)$

Analiza algoritmilor – complexitate (8)

Exprimați următoarele complexități cu ajutorul notației Θ:

1) $n^3 + 100n^2$

2) $2^n + 5n + 3$

3) $n^{2^n} + 4 \cdot 2^n$

4) $2 \cdot n^2 + n \cdot \ln n$

5) $n^k + n + n^k \cdot \ln n$

6) $1 + 2 + \ldots + n$

7) $1^2 + 2^2 + \ldots + n^2$

8) $1^3 + 2^3 + \ldots + n^3$

9) $1 \cdot 2 + 2 \cdot 3 + \ldots + n \cdot (n+1)$

10) $1^2 + 3^2 + \ldots + (2n-1)^2$

Analiza algoritmilor – logaritmi (1)

- Vom utiliza următoarele notații:

$$\lg n = \log_2 n$$
$$\ln n = \log_e n$$
$$\lg^k n = (\lg n)^k$$
$$\lg \lg n = \lg(\lg n)$$

- Deoarece schimbarea bazei unui logaritm de la o constantă la alta schimbă valoarea logaritmului doar printr-un factor constant, vom folosi notația *lg n* atunci când factorii constanți nu vor fi importanți.

Analiza algoritmilor – logaritmi (2)

Pentru orice numere reale a>0, b>0, c>0 și n,

$$a = b^{\log_b a}$$

$$\log_c (ab) = \log_c a + \log_c b$$

$$\log_b a^n = n \log_b a$$

$$\log_b a = \frac{\log_c a}{\log_c b}$$

$$\log_b a = \frac{1}{\log_a b}$$

$$\log_b a \cdot \log_a b = 1$$

$$\log_b a = \log_b c \cdot \log_c a$$

$$\log_b \frac{c}{a} = \log_b c - \log_b a$$

$$\log_b \frac{1}{a} = -\log_b a$$

$$a^{\log_b n} = n^{\log_b a}$$

Analiza algoritmilor – recurențe (1)

O recurență este o ecuație care descrie o funcție exprimând valoarea sa pentru argumente mai mici. Când un algoritm conține o apelare recursivă la el însuși, timpul său de execuție poate fi descris printr-o recurență. Recurențele pot fi rezolvate prin metoda substituției, metoda iterației sau metoda master [Cor00].

Metoda master oferă o soluție simplă pentru rezolvarea recurențelor de forma:

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

unde a≥1, b>1, iar f(n) este o funcție dată.

Teorema master. Fie a≥1 și b>1 constante, f(n) o funcție și T(n) definită pe întregii nenegativi prin recurența

$$T(n) = a \cdot T\left(\frac{n}{b}\right) + f(n)$$

atunci T(n) poate fi delimitată asimptotic după cum urmează:

1. $dacă\ \ f(n) = O(n^{\log_b a - \varepsilon}),\ \varepsilon > 0 \ \Rightarrow\ T(n) = \Theta(n^{\log_b a})$

2. $dacă\ \ f(n) = \Theta(n^{\log_b a}) \ \Rightarrow\ T(n) = \Theta(n^{\log_b a} \lg n)$

3. $dacă\ \ f(n) = \Omega(n^{\log_b a + \varepsilon}),\ \varepsilon > 0\ şi\ af\left(\frac{n}{b}\right) \le cf(n),\ c < 1\ şi\ n\ suficient\ de\ mari \Rightarrow T(n) = \Theta(f(n)).$

Analiza algoritmilor – recurențe (2)

Exemplu de utilizare a metodei master

Să considerăm $T(n) = 9 \cdot T\left(\frac{n}{3}\right) + n$

Pentru această recurență a=9, b=3, f(n)=n, astfel

$$n^{\log_b a} = n^{\log_3 9} = n^2$$

Deoarece $f(n) = O(n^{\log_3 9 - \varepsilon}) = O(n),\ \varepsilon = 1,$
aplicând cazul 1 al teoremei master,

$$T(n) = \Theta(n^{\log_3 9}) = \Theta(n^2)$$

Analiza algoritmilor – recurențe (3)

Exemplu de neaplicabilitate a metodei master

Să considerăm următoarea recurență:

$$T(n) = 2T(n/2) + n\lg n$$

Pentru această recurență a=2, b=2, f(n)=n lg n. Astfel,

$$n^{\log_b a} = n^{\log_2 2} = n^1 = n$$

Se poate observa că niciunul din cazurile teoremei master nu se poate aplica:

1. $n\lg n \neq O(n^{1-\varepsilon}),\ \forall\ \varepsilon > 0$
2. $n\lg n \neq \Theta(n)$
3. $n\lg n \neq \Omega(n^{1+\varepsilon}),\ \forall\ \varepsilon > 0\ (ex.\ \varepsilon = 0.01,\ n^{1.01} > n\lg n)$

Vom folosi în continuare metoda iterației pentru rezolvarea acestei recurențe.

Analiza algoritmilor – recurențe (4)

Rezolvare prin metoda iterației a recurenței

$$T(n) = 2T(n/2) + n\lg n$$

Iterăm recurența după cum urmează:

$$T(n) = 2T\left(\frac{n}{2}\right) + n\lg n$$

$$2T\left(\frac{n}{2}\right) = 2\left[2T\left(\frac{\frac{n}{2}}{2}\right) + \frac{n}{2}\lg\frac{n}{2}\right] = 2^2 T\left(\frac{n}{2^2}\right) + n\lg\frac{n}{2}$$

$$2^2 T\left(\frac{n}{2^2}\right) = 2^2\left[2T\left(\frac{n}{2^3}\right) + \frac{n}{2^2}\lg\frac{n}{2^2}\right] = 2^3 T\left(\frac{n}{2^3}\right) + n\lg\frac{n}{2^2}$$

$$2^{q-1} T\left(\frac{n}{2^{q-1}}\right) = 2^q T\left(\frac{n}{2^q}\right) + n\lg\frac{n}{2^{q-1}}$$

Considerând condiția la limită T(1)=Θ(1), recurența se termină pentru

$$\frac{n}{2^q} = 1 \Rightarrow 2^q = n \Rightarrow q = \lg n$$

Analiza algoritmilor – recurențe (5)

Am arătat că la limita T(1)=Θ(1), q=lg n. Obținem:

$$T(n) = 2^{\lg n} T(1) + \sum_{k=0}^{\lg n - 1} n \lg \frac{n}{2^k}$$

$$T(n) = 2^{\lg n} \Theta(1) + n \sum_{k=0}^{\lg n - 1} \left(\lg n - \lg 2^k \right)$$

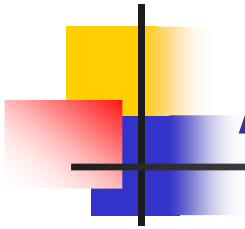
$$T(n) = n^{\lg 2} \Theta(1) + n \cdot \lg n \cdot \lg n - n \sum_{k=0}^{\lg n - 1} \lg 2^k$$

$$T(n) = n^1 \cdot \Theta(1) + n \cdot \lg^2 n - n \sum_{k=0}^{\lg n - 1} k$$

$$\sum_{k=0}^{n-1} k = \frac{n(n-1)}{2} \Rightarrow \sum_{k=0}^{\lg n - 1} k = \frac{\lg n \cdot (\lg n - 1)}{2}$$

$$T(n) = \Theta\left(n + n \lg^2 n - \frac{n \cdot \lg n \cdot (\lg n - 1)}{2} \right)$$

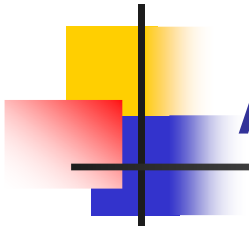
$$T(n) = \Theta\left(n \lg^2 n \right)$$



Analiza algoritmilor – recurențe (6)

Exerciții. Dați o delimitare asimptotică strânsă pentru următoarele recurențe:

1) $T(n) = T(2n/3) + 1$
2) $T(n) = 3T(n/4) + n \lg n$
3) $T(n) = 3T(n/3) + n/2$
4) $T(n) = 2T(n/2) + n$
5) $T(n) = 16T(n/4) + n$
6) $T(n) = 4T(n/2) + n^2$
7) $T(n) = 4T(n/2) + n^3$



Analiza algoritmilor – tipuri de analiză

- Principalele tipuri de analiză
  - Analiza cazului cel mai favorabil – cel mai scurt timp de execuție posibil pe date de intrare de dimensiune constantă *n*;
  - Analiza cazului cel mai defavorabil – cel mai mare timp de execuție posibil relativ la orice date de intrare de dimensiune constantă *n*;
  - Analiza cazului mediu – timpul mediu de execuție al unui algoritm considerând toate datele de intrare de dimensiune constantă *n*. Există însă probleme la care nu e prea clar care sunt datele de intrare de complexitate medie.
- Urmează câteva exemple de analiză a unor algoritmi: căutare, sortare, etc.

Analiza algoritmilor – căutarea maximului (1)

Se dă un tablou X de *n* elemente. Să se găsească *m* astfel încât $m = \max_{1 \le k \le n} X[k]$, considerând cel mai mare indice *k* care satisface această relație.

1. k ← n-1, m ← X[n].

2. Dacă k=0, algoritmul se încheie.

3. Dacă X[k]≤m, mergeți la pasul **5.**

4. m ← X[k].

5. Decrementați k cu 1 și reveniți la pasul **2.**

Pasul	Cost	Nr. execuții
1.	c1	1
2.	c2	n
3.	c3	n-1
4.	c4	A
5.	c5	n-1

Pentru ca analiza să fie completă trebuie să determinăm mărimea A, care reprezintă numărul de modificări ale valorii maxime curente.

Analiza algoritmilor – căutarea maximului (2)

Determinarea mărimii A

- Cazul cel mai favorabil: A=0, se obține atunci când maximul este X[n];
- Cazul cel mai defavorabil: A=n-1, se obține atunci când maximul este X[1];
- Cazul mediu: presupunând că cele n valori sunt distincte, contează doar ordinea lor relativă. Dacă n=3, există următoarele posibilități echiprobabile:

Situație	A
X[1]>X[2]>X[3]	2
X[1]>X[3]>X[2]	1
X[2]>X[1]>X[3]	1
X[2]>X[3]>X[1]	1
X[3]>X[1]>X[2]	0
X[3]>X[2]>X[1]	0

Analiza algoritmilor – căutarea maximului (3)

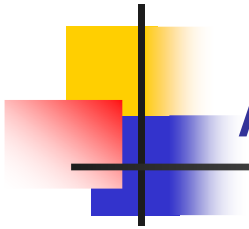
Determinarea valorii A pentru cazul mediu

- Probabilitatea ca *A* să aibă valoarea *k* va fi p_{nk}=(nr. situații în care *A=k*)*/n!*
- În exemplul anterior $p_{30}=1/3$, $p_{31}=1/2$, $p_{32}=1/6$
- Valoarea medie a lui A este definită astfel:

$$A_n = \sum_k k \cdot p_{nk}$$

A_n reprezintă de fapt media ponderată a valorilor lui *A*.

- În exemplul anterior valoarea medie a lui *A* pentru *n*=3 va fi 5/6. Knuth arată că pentru *n* mare A_n=ln *n*.



Analiza algoritmilor – căutarea maximului (4)

Determinarea complexității

- Cazul cel mai favorabil:
  $c_1 + c_2n + c_3(n-1) + 0 + c_5(n-1) =$
  $(c_2+c_3+c_5)n + (c_1-c_3-c_5) = an+b = \Theta(n)$
- Cazul cel mai defavorabil:
  $c_1 + c_2n + c_3(n-1) + c_4(n-1) + c_5(n-1) =$

  $(c_2+c_3+c_4+c_5)n + (c_1-c_3-c_4-c_5) = an+b = \Theta(n)$
- Cazul mediu:
  $c_1 + c_2n + c_3(n-1) + c_4 \ln n + c_5(n-1) =$
  $(c_2+c_3+c_5)n + c_4 \ln n + (c_1-c_3-c_5) = an + b \ln n + c = \Theta(n)$

Analiza algoritmilor – căutarea maximului (5)

Punând problema mai simplu,

	Cost	Execuții
m ← X[n]	c_1	1
for k ← n-1 downto 1 do	c_2	n
if X[k]>m then m ← X[k]	c_3	n-1

considerând operație elementară fiecare iterație a ciclului, e ușor de observat că bucla este parcursă în întregime, deci algoritmul are aceeași complexitate

$$(c_2+c_3)n + (c_1-c_3) = an+b = \Theta(n)$$

indiferent de ordinea relativă a celor *n* valori.

Analiza algoritmilor – căutarea secvențială (1)

Se dă un tablou X de n elemente. Se parcurge secvențial tabloul X până la primul element care are valoarea k.

	Cost	Execuții
1. i ← 1.	c_1	1
2. Dacă X[i] = k, algoritmul se încheie cu succes.	c_2	C
3. i ← i+1.	c_3	C-S
4. Dacă i≤n se revine la pasul **2.**	c_4	C-S
În caz contrar, algoritmul se încheie fără succes.		

unde C este numărul de comparații de chei și S este 1 în caz de succes respectiv 0 în caz de căutare fără succes. Complexitatea algoritmului este:

$$c_1 + c_2C + c_3(C-S) + c_4(C-S)$$

În cazul în care valoarea nu este găsită, avem C=n, S=0, cu o complexitate

$$(c_2+c_3+c_4)n + c_1 = an+b = \Theta(n)$$

În caz de succes, cu X[i]=k, avem C=i, S=1, cu o complexitate

$$(c_2+c_3+c_4)i + (c_1-c_3-c_4)$$

Considerând că valorile din tablou sunt distincte și echiprobabile, valoarea medie a lui C este

$$\frac{1+2+\ldots+n}{n} = \frac{n+1}{2}$$

deci complexitatea în cazul mediu este tot Θ(n).

Analiza algoritmilor – căutarea secvențială (2)

- Punând problema mai simplu, algoritmul arată în felul următor:

```
CAUTARE_SECV(X, k)
    for i ← 1 to n do
        if x[i] = k then
            return i
    return 0
```

- Algoritmul constă în execuția unei bucle: o singura iterație în cazul cel mai favorabil, *n* iterații în cazul cel mai defavorabil, respectiv (1+2+...+n)/n = (*n*+1)/2 iterații în cazul mediu. Așadar, complexitatea este Θ(1) în cazul cel mai favorabil, Θ(n) în cazul cel mai defavorabil și tot Θ(n) în cazul mediu.

Analiza algoritmilor – căutarea binară (1)

Se dă un tablou X de n elemente ordonate crescător. Valoarea căutată k se compară cu elementul din mijlocul tabloului, iar rezultatul acestei comparații ne arată în care jumătate trebuie continuată căutarea.

	Cost	**Execuții**
1. s ← 1, d ← n	c_1	1
2. Dacă d<s alg. se încheie fără succes. Altfel, i ← (s+d)/2.	c_2	C+1-S
3. Dacă k<X[i] se trece la pasul **4.** Dacă k>X[i] se trece la pasul **5.** Dacă k=X[i] alg. se încheie cu succes.	c_3	C
4. d ← i-1 și se revine la pasul **2.**	c_{45}	C-S
5. s ← i+1 și se revine la pasul **2.**		

unde C este numărul de comparații de chei și S este 1 în caz de succes respectiv 0 în caz de căutare fără succes. Complexitatea algoritmului este:

$$c_1 + c_2(C+1-S) + c_3C + c_{45}(C-S)$$

Analiza algoritmilor – căutarea binară (2)

Rescris într-o formă mai apropiată de limbajele de nivel înalt, algoritmul arată în felul următor:

```
CAUTARE_BINARA(X, k)                    Cost    Execuții
    st ← 1                              c1      1
    dr ← X.size                         c2      1
    while st≤dr do                      c3      C
            i ← (st+dr)/2
            if k=X[i] then
                    return i
            if k<X[i] then
                    dr ← i-1
            if k>X[i] then
                    st ← i+1
    return 0                            c4      1-S
```

Complexitatea algoritmului este:

$$c1 + c2 + c3C + c4(1-S)$$

Complexitatea temporală a algoritmului de căutare binară depinde astfel doar de numărul de comparații C.

Analiza algoritmilor – căutarea binară (3)

Determinarea valorii C pentru cazul cel mai favorabil

Valoarea căutată este găsită la prima comparație, astfel C=1, deci complexitatea temporală este Θ(1).

Determinarea valorii C pentru cazul cel mai defavorabil

Valoarea maximă a lui C (numărul comparațiilor de chei) poate fi exprimată prin următoarea recurență

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(1)$$

care descrie înjumătățirea recurentă a spațiului de căutare cu un cost constant Θ(1).

Aplicând terorema master, avem a=1 (căutarea se continuă pe o singură ramură st./dr.), b=2 (spațiul de căutare se reduce la jumătate) și f(n)=Θ(1). Astfel,

$$T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(n^0 \lg n) = \Theta(\lg n)$$

deci putem aplica cazul 2 al teoremei master. Prin urmare,

$$n^{\log_b a} = n^{\log_2 1} = n^0 = 1$$

Astfel, în cazul cel mai defavorabil, C=Θ(lg n) iar complexitatea temporală este Θ(lg n).

Analiza algoritmilor – căutarea binară (4)

Determinarea valorii C pentru cazul mediu

Considerând că valorile din tablou sunt distincte și echiprobabile, în cazul mediu valoarea căutată este găsită la jumătatea procesului de căutare, ceea ce în număr de iterații este echivalent cu parcurgere totală dar divizarea recurentă a spațiului de căutare la un sfert:

$$T(n) = T\left(\frac{n}{4}\right) + \Theta(1)$$

Aplicând teorema master, avem a=1, b=4 și f(n)=Θ(1). Astfel,

$$n^{\log_b a} = n^{\log_4 1} = n^0 = 1$$

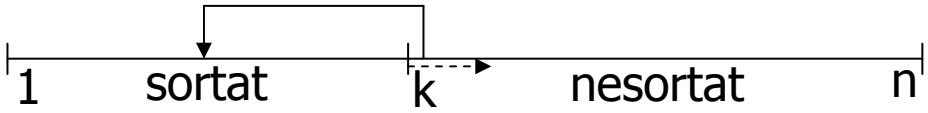
deci putem aplica cazul 2 al teoremei master. Prin urmare,

$$T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(n^0 \lg n) = \Theta(\lg n)$$

Astfel, în cazul mediu, la fel ca în cazul cel mai defavorabil, C=Θ(lg n), deci complexitatea temporală este tot Θ(lg n).

Analiza algoritmilor – sortare prin inserție (1)

Se dă un tablou X de n elemente. Se consideră pe rând vectorii formați din primele 2, 3, ..., n elemente din tablou, ordonate prin aducerea noului element (al 2-lea, al 3-lea, ..., al n-lea) pe poziția corespunzătoare valorii sale. Aceasta presupune deplasarea la dreapta a elementelor cu valori mai mari, astfel încât noul element să poată fi inserat înaintea lor:



	Cost	**Execuții**
for j ← 2 to n do	c_1	n
k ← X[j]	c_2	n-1
i ← j-1	c_3	n-1
while i>0 and X[i]>k do	c_4	$\sum_{j=2}^{n} t_j$
X[i+1] ← X[i]	c_5	$\sum_{j=2}^{n} (t_j - 1)$
i ← i-1	c_6	$\sum_{j=2}^{n} (t_j - 1)$
X[i+1] ← k	c_7	n-1

unde am notat cu t_j numărul de execuții ale testului while

Analiza algoritmilor – sortare prin inserție (2)

Determinarea complexității în cazul cel mai favorabil

- Vectorul de intrare este deja sortat, deci bucla *while* nu se execută niciodată (se verifică doar condițiile o singură dată în fiecare iterație *for*), astfel $t_j=1$, rezultă:
  - $\sum_{j=2}^{n} t_j = n-1$
  - $\sum_{j=2}^{n} (t_j - 1) = 0$
- $c_1 n + c_2(n-1) + c_3(n-1) + c_4(n-1) + c_7(n-1) =$

  $(c_1 + c_2 + c_3 + c_4 + c_7)n - (c_2 + c_3 + c_4 + c_7) = an + b = \Theta(n)$

Analiza algoritmilor – sortare prin inserție (3)

Determinarea complexității în cazul cel mai defavorabil

- Vectorul de intrare este sortat în ordine inversă, deci bucla *while* se execută de 1, 2, ..., n-1 ori (condițiile fiind verificate de 2, 3, ..., n ori) pentru j=2, 3, ...,n, astfel t_j=j,
  - $\sum_{j=2}^{n} t_j = \sum_{j=2}^{n} j = \frac{n(n+1)}{2} - 1$
  - $\sum_{j=2}^{n} (t_j - 1) = \frac{n(n+1)}{2} - 1 - (n-1) = \frac{n(n-1)}{2}$
- $c_1 n + c_2(n-1) + c_3(n-1) + c_4\left(\frac{n(n+1)}{2} - 1\right) + c_5 \cdot \frac{n(n-1)}{2} + c_6 \cdot \frac{n(n-1)}{2} + c_7(n-1) =$

  $\left(\frac{c_4}{2} + \frac{c_5}{2} + \frac{c_6}{2}\right)n^2 + \left(c_1 + c_2 + c_3 + \frac{c_4}{2} - \frac{c_5}{2} - \frac{c_6}{2} + c_7\right)n - (c_2 + c_3 + c_4 + c_7) =$

  $an^2 + bn + c = \Theta(n^2)$

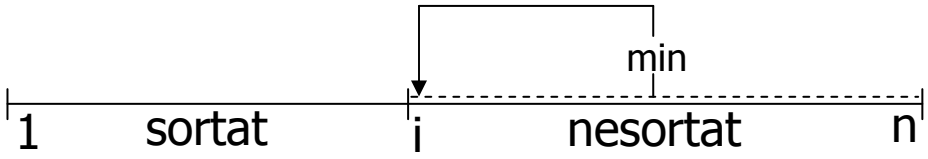
Analiza algoritmilor – sortare prin inserție (4)

Determinarea complexității în cazul mediu

- Să presupunem că alegem la întâmplare n numere distincte și aplicăm sortarea prin inserție.
- Câte iterații sunt necesare pentru inserarea elementului X[j] în subvectorul X[1...j-1]? În medie jumătate din elementele subvectorului X[1...j-1] sunt mai mici decât X[j] și jumătate sunt mai mari. Prin urmare, în medie, trebuie verificate jumătate din elementele subvectorului X[1...j-1], deci $t_j=j/2$,
  - $$\sum_{j=2}^{n} t_j = \frac{1}{2} \cdot \sum_{j=2}^{n} j = \frac{1}{2}\left(\frac{n(n+1)}{2} - 1\right) = \frac{n^2+n-2}{4}$$
  - $$\sum_{j=2}^{n} (t_j - 1) = \frac{n^2+n-2}{4} - (n-1) = \frac{n^2-3n+2}{4}$$
- Astfel, complexitatea în cazul mediu va fi tot $\Theta(n^2)$, la fel ca în cazul cel mai defavorabil.

Analiza algoritmilor – sortare prin selecție (1)

Se dă un tablou X de n elemente. Se consideră pe rând subtablourile formate din elementele i, ..., n (cu i=1, 2, ..., n) și se aduce prin interschimbare elementul minim al fiecărui subtablou pe poziția i:




```
SELSORT1(X)
    for i ← 1 to n-1 do
        min ← i
        for j ← i+1 to n do
            if X[j] < X[min] then
                min ← j
        X[i] ←→ X[min]
```

Analiza algoritmilor – sortare prin selecție (2)

O altă variantă a sortării prin selecție:

```
SELSORT2(X)
    for i ← 1 to n-1 do
        for j ← i+1 to n do
            if X[j] < X[i] then
                X[i] ←→ X[j]
```

Complexitatea algoritmului este:

$$C = (n-1) + (n-2) + \ldots + 1 = \frac{n(n-1)}{2} = \Theta(n^2)$$

Analiza algoritmilor – Bubblesort (1)

Se dă un tablou X de n elemente. Algoritmul Bubblesort parcurge repetat tabloul X interschimbând dacă e cazul elementele consecutive:

```
BUBBLESORT1(X)
    n ← X.size
    for i ← 1 to n do
        for j ← 1 to n-1 do
            if X[j] > X[j+1] then
                X[j] ←→ X[j+1]
```

Complexitatea algoritmului este:

$C = n(n-1) = \Theta(n^2)$

Analiza algoritmilor – Bubblesort (2)

Elementul de pe ultima poziție a iterației curente poate fi exclus din iterația următoare:

```
BUBBLESORT2(X)
    n ← X.size
    for i ← 1 to n-1 do
        for j ← 1 to n-i do
            if X[j] > X[j+1] then
                X[j] ←→ X[j+1]
```

Complexitatea algoritmului este:

$$C = (n-1) + (n-2) + \ldots + 1 = \frac{n(n-1)}{2} = \Theta(n^2)$$

Analiza algoritmilor – Bubblesort (3)

O altă variantă în care elementul de pe prima poziție a iterației curente poate fi exclus din iterația următoare:

```
BUBBLESORT3(X)
    n ← X.size
    for i ← 1 to n-1 do
        for j ← n-1 downto i do
            if X[j] > X[j+1] then
                X[j] ←→ X[j+1]
```

Complexitatea algoritmului este:

$$C = (n-1) + (n-2) + \ldots + 1 = \frac{n(n-1)}{2} = \Theta(n^2)$$

Analiza algoritmilor – Bubblesort (4)

Algoritmul poate fi îmbunătățit reducând numărul de iterații de la n la câte sunt necesare până la obținerea tabloului sortat. Și în acest caz elementul de pe ultima poziție a iterației curente poate fi exclus din iterația următoare:

```
BUBBLESORT4(X)
    sortat ← false
    n ← X.size-1
    while not sortat do
        sortat ← true
        for i ← 1 to n do
            if X[i] > X[i+1] then
                X[i] ←→ X[i+1]
                sortat ← false
         n←n-1
```

Complexitatea algoritmului în cazul cel mai favorabil (o singură parcurgere) este **Θ(n)** iar în cazul cel mai defavorabil rămâne:

$$C = (n-1) + (n-2) + \ldots + 1 = \frac{n(n-1)}{2} = \Theta(n^2)$$

Analiza algoritmilor – Quicksort (1)

Algoritmul de sortare rapidă se bazează pe paradigma *divide et impera*. Tabloul este rearanjat în două subtablouri astfel încât fiecare element al primului subtablou este mai mic sau egal cu orice element din al doilea subtablou. Cele două subtablouri sunt sortate prin apeluri recursive ale algoritmului de sortare rapidă.

```
QUICKSORT(X, primul, ultimul)
        i ← primul
        j ← ultimul
        pivot ← X[primul]
        while i<j do
            while X[i]<pivot do
                i ← i+1
            while X[j]>pivot do
                j ← j-1
            if i<j then
                X[i] ←→ X[j]
            if i≤j then
                i ← i+1
                j ← j-1
        if primul<j then QUICKSORT(X, primul, j)
        if i<ultimul then QUICKSORT(X, i, ultimul)
```

Analiza algoritmilor – Quicksort (2)

Analiza cazului cel mai defavorabil

- Cazul cel mai defavorabil este acela în care toate partiționările sunt total dezechilibrate: un subtablou de un element și unul de n-1 elemente. Astfel, cazul cel mai defavorabil poate fi descris prin următoarea recurență:

$$T(n) = T(1) + T(n-1) + \Theta(n)$$

- Unde Θ(n) este timpul de partiționare. Deoarece T(1)=Θ(1), se ajunge la recurența

$$T(n) = T(n-1) + \Theta(n)$$

- Apoi, iterând recurența, obținem:

$$T(n) = \sum_{k=1}^{n} \Theta(k) = \Theta\left(\sum_{k=1}^{n} k\right) = \Theta\left(\frac{n(n+1)}{2}\right) = \Theta(n^2)$$

Analiza algoritmilor – Quicksort (3)

Analiza cazului cel mai favorabil

- Atunci când partițiile sunt aproximativ egale se ajunge la următoarea recurență:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

- Folosind teorema master avem a=2 (sortarea se continuă pe ambele partiții), b=2 (vectorul de sortat se înjumătățește) și f(n)=Θ(n) (timpul de partiționare). Astfel,

$$n^{\log_b a} = n^{\log_2 2} = n^1 = n$$

- Deci putem aplica cazul 2 al teoremei master. Prin urmare, avem:

$$T(n) = \Theta(n^{\log_b a} \lg n) = \Theta(n^{\log_2 2} \lg n) = \Theta(n \lg n)$$

Analiza algoritmilor – Quicksort (4)

Analiza cazului mediu

- Atunci când partiționările echilibrate și dezechilibrate alternează, combinarea unei partiționări defavorabile și a uneia favorabile produce trei subvectori de dimensiuni 1, (n-1)/2 și (n-1)/2 cu un cost total:

  n + n-1 = 2n-1 = Θ(n)

- Se poate observa că o partiționare defavorabilă de un cost Θ(n) poate fi absorbită de o partiționare echilibrată de cost Θ(n), și partiționarea rezultată este favorabilă.
- Astfel, complexitatea medie este aceeași ca în cazul cel mai favorabil, $O(n \lg n)$, doar constanta din notația O este mai mare. Alternanța partiționărilor dezechilibrate cu cele echilibrate poate fi descrisă prin recurența

$$T(n) = \Theta(n) + T(1) + T(n-1) = \Theta(n) + T(1) + \Theta(n) + T\left(\frac{n-1}{2}\right) + T\left(\frac{n-1}{2}\right)$$

- unde T(1)=Θ(1), iar Θ(n)+Θ(1)+Θ(n)=Θ(n), deci obținem recurența

$$T(n) = 2T\left(\frac{n-1}{2}\right) + \Theta(n)$$

Analiza algoritmilor – Quicksort (5)

Rezolvarea recurenței cazului mediu

Metoda master nu e aplicabilă pentru recurența

$$T(n) = 2T\left(\frac{n-1}{2}\right) + \Theta(n)$$

Prin urmare, vom folosi metoda iterației:

$$T(n) = 2T\left(\frac{n-1}{2}\right) + \Theta(n)$$

$$2T\left(\frac{n-1}{2}\right) = 2\left[2T\left(\frac{\frac{n-1}{2}-1}{2}\right) + \Theta\left(\frac{n-1}{2}\right)\right] = 2^2 T\left(\frac{n-3}{2^2}\right) + 2^1\Theta\left(\frac{n-1}{2^1}\right)$$

$$2^2 T\left(\frac{n-3}{2^2}\right) = 2^2\left[2T\left(\frac{\frac{n-3}{2^2}-1}{2}\right) + \Theta\left(\frac{n-3}{2^2}\right)\right] = 2^3 T\left(\frac{n-7}{2^3}\right) + 2^2\Theta\left(\frac{n-3}{2^2}\right)$$

$$2^{q-1} T\left(\frac{n-2^{q-1}+1}{2^{q-1}}\right) = 2^q T\left(\frac{n-2^q+1}{2^q}\right) + 2^{q-1}\Theta\left(\frac{n-2^{q-1}+1}{2^{q-1}}\right)$$

Analiza algoritmilor – Quicksort (6)

Considerând condiția la limită T(1)=Θ(1), recurența se termină pentru

$$\frac{n-2^q+1}{2^q}=1 \Rightarrow 2^{q+1}=n+1 \Rightarrow q+1=\lg(n+1) \Rightarrow q=\lg(n+1)-1$$

Astfel, obținem:

$$T(n)=2^{\lg(n+1)-1}T(1)+\sum_{k=1}^{\lg(n+1)-1}\left(2^{k-1}\Theta\left(\frac{n-2^{k-1}+1}{2^{k-1}}\right)\right)$$

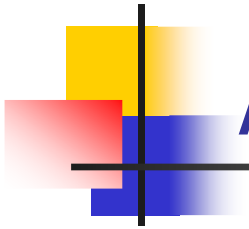
$$T(n)=2^{\lg(n+1)-1}\Theta(1)+\sum_{k=1}^{\lg(n+1)-1}\Theta(n)$$

$$T(n)=\Theta(n)+\left(\lg(n+1)-1\right)\cdot\Theta(n)$$

$$T(n)=\Theta\left(n+n\lg(n+1)-n\right)$$

$$T(n)=\Theta(n\lg n)$$

Analiza algoritmilor – Mergesort (1)



- Algoritmul de sortare prin interclasare (*Mergesort*), bazat pe paradigma *divide et impera*, a fost introdus de John von Neumann în 1945.
- Funcția principală a algoritmului, *MERGESORT*, realizează următoarele operații:
  - determină mijlocul tabloului;
  - sortează prin câte un autoapel (recursiv) cele două subtablouri;
  - apelează funcția *MERGE* care interclasează cele două subtablouri (sortate), obținând astfel tabloul sortat.
- Divizarea are loc recursiv până se ajunge la subtablouri formate dintr-un singur element, considerate deci sortate.
- Urmează în continuare pseudocodul algoritmului.

Analiza algoritmilor – Mergesort (2)

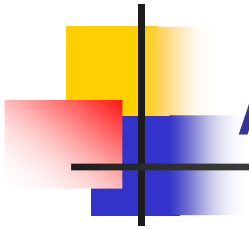
```
MERGE(X, primul, pivot, ultimul)
    k ← primul
    i ← primul
    j ← pivot+1;
    while i≤pivot and j≤ultimul do
        if X[i]<X[j] then
            T[k] ← X[i]
            k ← k+1
            i ← i+1
        else
            T[k] ← X[j]
            k ← k+1
            j ← j+1
    if j>ultimul then
        for j ← i to pivot do
            T[k] ← X[j]
            k ← k+1
    if i>pivot then
        for i ← j to ultimul do
            T[k] ← X[i]
            k ← k+1
    for i ← primul to ultimul do
        X[i] ← T[i]
```

```
MERGESORT(X, primul, ultimul)
    if primul<ultimul then
        pivot ← (primul+ultimul)/2
        MERGESORT(X, primul, pivot)
        MERGESORT(X, pivot+1, ultimul)
        MERGE(X, primul, pivot, ultimul)
```

Având în vedere divizarea în două subtablouri de aproximativ aceeași dimensiune și apoi interclasarea, complexitatea temporală a algoritmului poate fi descrisă prin recurența:

$$T(n) = 2T\left(\frac{n}{2}\right) + \Theta(n)$$

rezolvată la Quicksort, cu soluția Θ(n lg n).



Analiza algoritmilor – căutare și sortare

Aplicații

1. Să se rezolve recurența aferentă cazului cel mai defavorabil al algoritmului de căutare binară folosind metoda iterației.
2. Să se rezolve recurenţa care descrie complexitatea temporală a algoritmului *Mergesort* folosind metoda iteraţiei.
3. Să se implementeze algoritmii de sortare prezentați.
4. Comparați algoritmii de sortare, pe un tablou mare, măsurând timpii de execuție.

Funcții de dispersie

Mapează o cheie k într-una din cele m locații ale tabelei de dispersie

- Metoda diviziunii

  $h(k) = k \bmod m$

- Metoda înmulțirii

  $h(k) = \lfloor m(kA \bmod 1) \rfloor,\ 0 < A < 1$

- Dispersia universală

  $h(k) = \sum_{i=0}^{r} a_i k_i \bmod m,$

  unde cheia k s-a descompus în r+1 octeți, astfel încât

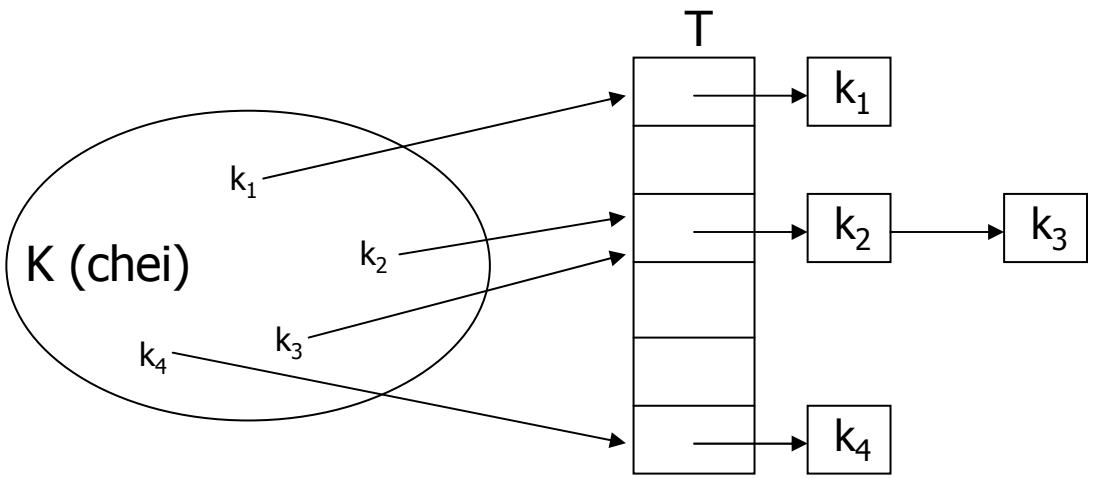
  $k=\{k_0, k_1, ..., k_r\},\ k_i<m$

  $a=\{a_0, a_1, ..., a_r\}$ valori alese aleator din mulțimea $\{0, 1, ..., m-1\}$

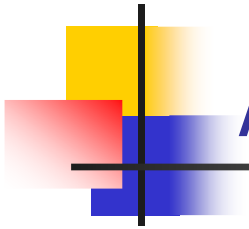
Analiza algoritmilor – tabele de dispersie (2)

Rezolvarea coliziunilor prin înlănțuire

- Elementele care se dispersează în aceeași locație se încarcă într-o listă înlănțuită.



- Locația j conține un pointer către capul listei elementelor care se dispersează în locația j, sau NULL dacă nu există astfel de elemente.
- Operația de căutare, în cazul cel mai defavorabil (când toate cele n chei se dispersează în aceeași locație), este Θ(n).



Analiza algoritmilor – tabele de dispersie (3)

Rezolvarea coliziunilor prin adresare deschisă

- Toate elementele sunt memorate în interiorul tabelei de dispersie.
- Locația j conține elementul dispersat în locația j sau NULL dacă nu există un astfel de element.
- Pentru inserare se examinează succesiv tabela de dispersie, începând cu locația indexată, până la găsirea unei locații libere.
- Pentru căutarea unei chei se examinează secvența de locații folosită de algoritmul de inserare. Căutarea se termină cu succes dacă se găsește cheia k sau fără succes dacă se întâlnește o locație liberă.
- La ștergerea unei chei nu poate fi marcată locația ca fiind liberă pentru că ar face imposibilă accesarea cheilor a căror inserare a găsit această locație ocupată. Prin urmare, locația se marchează cu o valoare specială (ȘTERS, -1, etc.). Evident că la inserare locațiile marcate astfel se consideră libere.

Analiza algoritmilor – tabele de dispersie (4)

Verificarea liniară

- Fiind dată o funcție de dispersie h', metoda verificării liniare folosește funcția de dispersie

$$h(k,i) = \left(h'(k) + i\right) \bmod m$$

pentru i=0, 1, ..., m-1.

- În cazul accesării tabelei de dispersie cu o cheie k, secvența locațiilor examinate este:

T[h'(k)], T[h'(k)+1], ..., T[m-1], T[0], T[1], ..., T[h'(k)-1].

- Verificarea liniară poate genera grupări primare (șiruri lungi de locații ocupate care tind să se lungească) crescând timpul mediu de căutare.

Analiza algoritmilor – tabele de dispersie (5)

Verificarea pătratică

- Fiind dată funcția de dispersie h', verificarea pătratică folosește o funcție de dispersie de forma

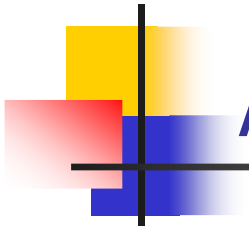
$$h(k,i) = \left(h'(k) + c_1 i + c_2 i^2\right) \bmod m,$$

unde c_1, $c_2 \neq 0$ sunt constante auxiliare și $i = 0, 1, \ldots, m-1$.

- Locația verificată inițial este T[h'(k)], următoarele locații examinate depinzând într-o manieră pătratică de i.
- Dacă două chei au aceeași poziție de start a verificării, atunci secvențele locațiilor verificate coincid:

$$h(k_1,0) = h(k_2,0) \Rightarrow h(k_1,i) = h(k_2,i)$$

Această situație poate conduce la o formă mai ușoară de grupare, numită grupare secundară.



Analiza algoritmilor – tabele de dispersie (6)

Dispersia dublă

- Se folosește o funcție de dispersie de forma:

$$h(k,i) = \left(h'(k) + i \cdot h''(k)\right) \bmod m,$$

unde *h′* și *h″* sunt funcții de dispersie auxiliare

- Spre deosebire de verificarea liniară sau pătratică, secvența locațiilor examinate depinde prin două funcții de dispersie de cheia k.
- În cazul dispersiei duble, când cheia variază, poziția inițială a verificării h’(k) și decalajul h”(k) pot varia independent, evitând astfel grupările primare și secundare.

Analiza algoritmilor – tabele de dispersie (7)

Exerciții [Cor00]

1. Se consideră o tabelă de dispersie de dimensiune m=1000 și funcția de dispersie

$$h(k) = \lfloor m(kA \bmod 1) \rfloor,\ A = \frac{\sqrt{5}-1}{2}$$

Calculați locațiile în care se pun cheile 61, 62, 63, 64 și 65.

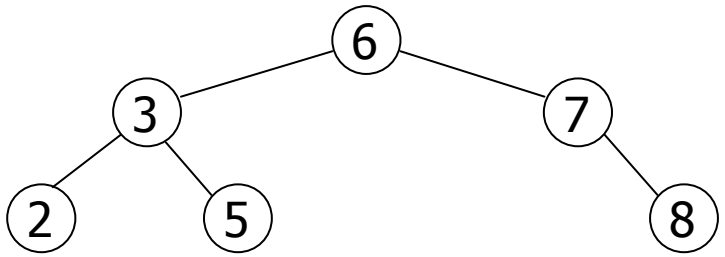
2. Se consideră că se inserează cheile 10, 22, 31, 4, 15, 28, 17, 88, 59 într-o tabelă de dispersie cu m=11 locații folosind adresarea deschisă cu funcția primară de dispersie $h'=k \bmod m$. Ilustrați rezultatul inserării acestor chei folosind verificarea liniară, verificarea pătratică cu c_1=1 și c_2=3 respectiv dispersia dublă cu $h''(k)=1+(k \bmod (m-1))$.

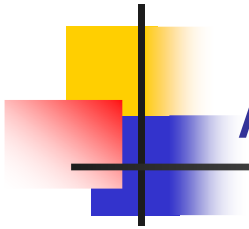
Analiza algoritmilor – arbori binari de căutare (1)

- **Arborele** este o particularizare a unei structuri de date mai generale, graful, care va fi prezentat în capitolul următor.
- Utilizarea unei **structuri arborescente binare** face posibilă inserarea și ștergerea rapidă, precum și căutarea eficientă a datelor.
- Dacă orice nod din arbore poate avea cel mult doi fii, atunci arborele se numește **arbore binar**.
- Pe lângă un câmp *cheie* și date adiționale, fiecare nod al arborelui binar conține câmpurile *st*, *dr* și *p* – referințe spre nodurile corespunzătoare fiului stâng, fiului drept și părintelui.
- Într-un **arbore binar de căutare** cheile sunt întotdeauna astfel memorate încât ele satisfac următoarea proprietate:

  Fie x un nod dintr-un arbore binar de căutare. Dacă y este un nod din subarborele stâng al lui x, atunci y.cheie≤x.cheie. Dacă y este un nod din subarborele drept al lui x, atunci y.cheie≥x.cheie.

- Exemplu:





Analiza algoritmilor – arbori binari de căutare (2)

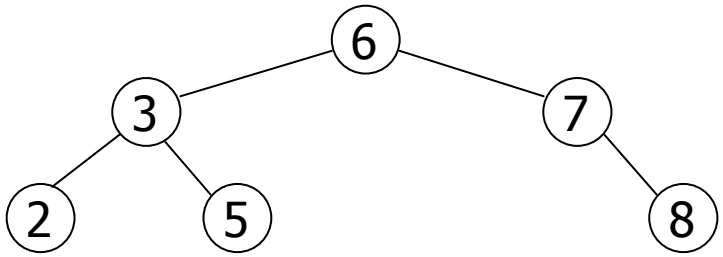
Implementarea clasei Node în Java

```
public class Node {
  int key;                  //cheia
  float data;               //alte informatii
  Node leftChild;           //fiul stang
  Node rightChild;          //fiul drept
  Node parent;              //parintele

  public Node(int k){  //constructorul
    key = k;                //initializarea cheii
  }
}
```

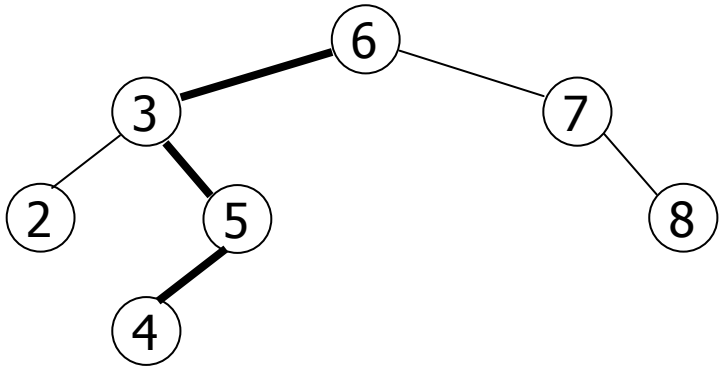
Analiza algoritmilor – arbori binari de căutare (3)

Inserarea – constă în inserarea nodului *z* pe poziția corespunzătoare în arborele binar de căutare cu rădăcina *r*.

```
INSEREAZA(z)
    y ← NULL
    x ← r
    while x≠NULL do
        y ← x
        if z.cheie<x.cheie then
            x ← x.st
        else
            x ← x.dr
    z.p ← y
    if y=NULL then
        r ← z
    else if z.cheie<y.cheie then
        y.st ← z
    else
        y.dr ← z
```



Inserarea nodului 4:



Analiza algoritmilor – arbori binari de căutare (4)

Implementarea inserării în Java

```
public class BinaryTree {

  Node root = null;                          //nodul radacina

  public BinaryTree() {
    insert(new Node(7));                     //inserarea unui nod cu cheia 7
    insert(new Node(1));                     //inserarea unui nod cu cheia 1
    insert(new Node(5));                     //inserarea unui nod cu cheia 5
  }

  public void insert(Node z){
   Node y = null;                            //parintele nodului curent x
   Node x = root;                            //nodul curent
   while(x!=null){                           //deplasarea in jos in arbore
     y=x;
     if(z.key<x.key) x=x.leftChild;          //deplasarea in jos pe subarborele stang
     else x=x.rightChild;                    //deplasarea in jos pe subarborele drept
   }
   z.parent=y;                               //x==null, z este inserat pe aceasta pozitie null, cu y parinte
   if(y==null) root=z;                       //daca parintele este null, z devine radacina arborelui
   else if(z.key<y.key) y.leftChild=z;       //setarea nodului z ca fiu stang al lui y
   else y.rightChild=z;                      //setarea nodului z ca fiu drept al lui y
  }

public static void main(String[] args) {
    BinaryTree binaryTree1 = new BinaryTree();
  }
}
```

Analiza algoritmilor – arbori binari de căutare (5)

- **Traversarea arborelui în inordine** – vizitează nodurile în ordinea crescătoare a cheilor. Rădăcina se vizitează între nodurile din subarborele său stâng și cele din subarborele său drept. În exemplul anterior: 2, 3, 5, 6, 7, 8.

```
INORDINE(x)
        if x≠NULL then
                INORDINE(x.st)
                AFISEAZA(x.cheie)
                INORDINE(x.dr)
```

- **Traversarea arborelui în preordine** – vizitează rădăcina înaintea nodurilor din subarbori. În exemplul anterior: 6, 3, 2, 5, 7, 8.

```
PREORDINE(x)
        if x≠NULL then
                AFISEAZA(x.cheie)
                PREORDINE(x.st)
                PREORDINE(x.dr)
```

- **Traversarea arborelui în postordine** – vizitează rădăcina după nodurile din subarbori. În exemplul anterior: 2, 5, 3, 8, 7, 6.

```
POSTORDINE(x)
        if x≠NULL then
                POSTORDINE(x.st)
                POSTORDINE(x.dr)
                AFISEAZA(x.cheie)
```

Analiza algoritmilor – arbori binari de căutare (6)

Căutarea unei chei

- Căutarea returnează nodul având cheia *k* dacă există sau NULL în caz contrar.
- **Căutarea recursivă**

```
CAUTA_REC(x, k)
    if x=NULL or k=x.cheie then
        return x
    if k<x.cheie then
        return CAUTA_REC(x.st, k)
    else
        return CAUTA_REC(x.dr, k)
```

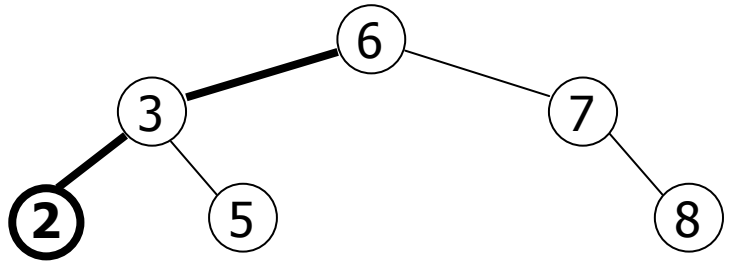
- **Căutarea iterativă** (de obicei mai eficientă decât varianta recursivă)

```
CAUTA_IT(x, k)
    while x≠NULL and k≠x.cheie do
        if k<x.cheie then
            x ← x.st
        else
            x ← x.dr
    return x
```

Analiza algoritmilor – arbori binari de căutare (7)

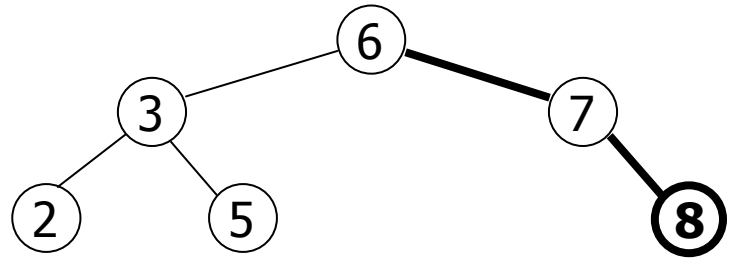
- **Minimul** – determinarea nodului cu cheia minimă dintr-un arbore binar de căutare se realizează prin deplasare în jos pe subarborele stâng până când se ajunge la un nod frunză.

```
MINIM(x)
    while x.st≠NULL do
        x ← x.st
    return x
```



- **Maximul** – este nodul cu cheia maximă și se determină prin deplasare în jos pe subarborele drept până când se ajunge la un nod frunză.

```
MAXIM(x)
    while x.dr≠NULL do
        x ← x.dr
    return x
```



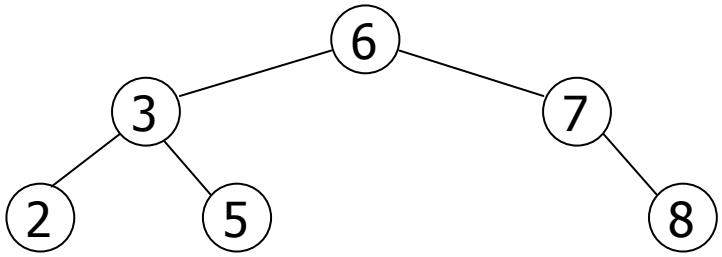
Analiza algoritmilor – arbori binari de căutare (8)

Succesorul unui nod *x* este nodul având cea mai mică cheie mai mare decât cheia lui *x*, sau NULL, dacă *x* are cea mai mare cheie din arborele binar de căutare. Trebuie tratate două alternative:

- Dacă *x* are subarbore drept, atunci succesorul lui *x* este nodul cu cheia minimă din acest subarbore. În exemplul anterior, succesorul lui 3 este 5.
- Dacă *x* nu are fiu drept, atunci succesorul lui *x* se determină traversând arborele binar de căutare de la *x* în sus până când se întâlneşte un nod care este fiu stâng, părintele acelui nod fiind succesorul lui *x*. În exemplul anterior, succesorul lui 5 este 6.

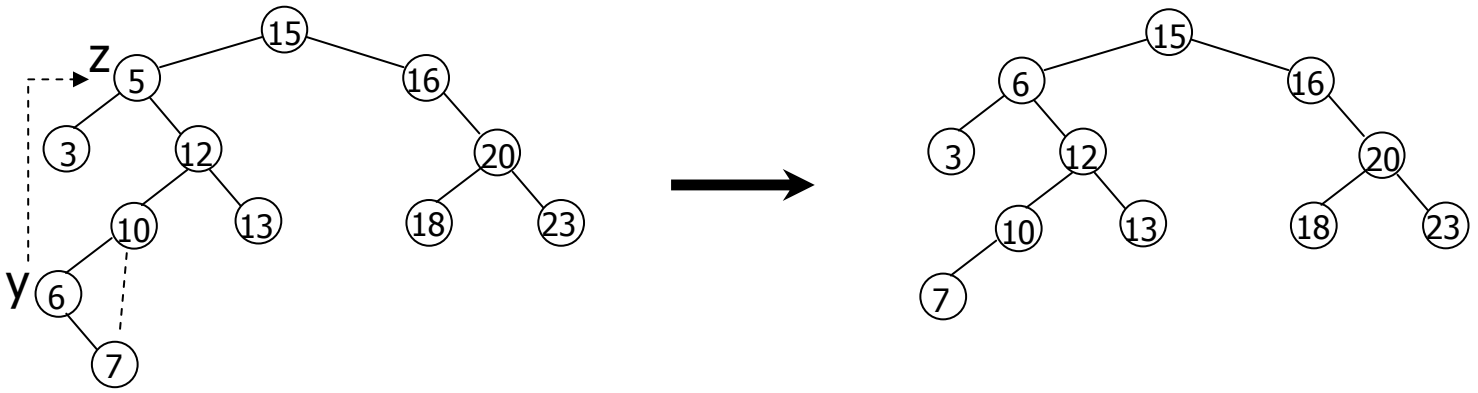
```
SUCCESOR(x)
    if x.dr≠NULL then
        return MINIM(x.dr)
    y ← x.p
    while y≠NULL and x=y.dr do
        x ← y
        y ← y.p
    return y
```

Succesorul lui 3 este 5
Succesorul lui 5 este 6



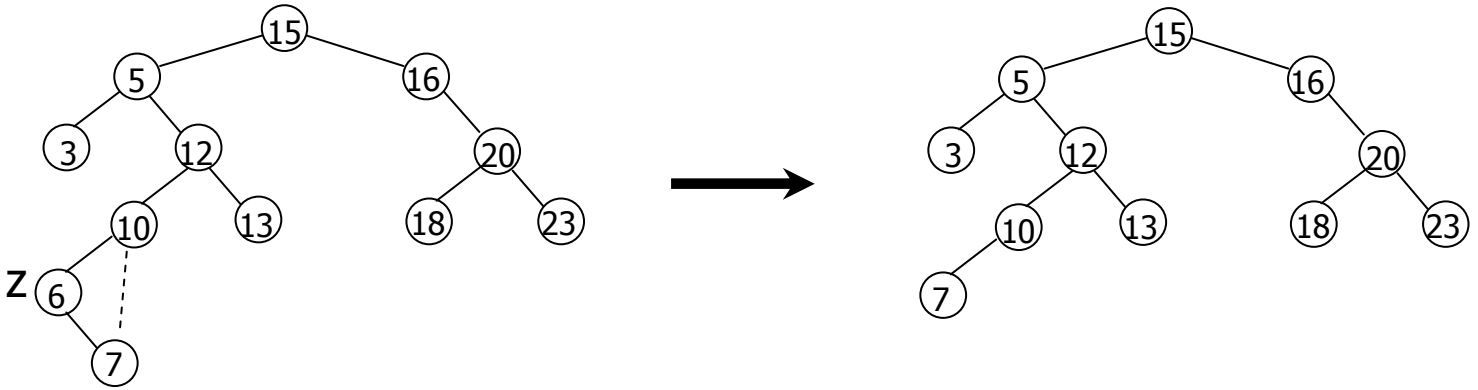
Analiza algoritmilor – arbori binari de căutare (9)

- **Ștergerea** unui nod *z* constă în tratarea următoarelor trei situații:
  - Dacă *z* nu are fii, se va modifica părintele său pentru a-i înlocui fiul *z* cu NULL.
  - Dacă *z* are un fiu, *z* va fi eliminat prin setarea unei legături de la părintele lui *z* la fiul lui *z*.
  - Dacă *z* are doi fii, se va elimina din arbore succesorul *y* al lui *z* și apoi se vor înlocui cheia și informațiile adiționale ale lui *z* cu cele ale lui *y*.
- Exemplu în care *z* are doi fii:

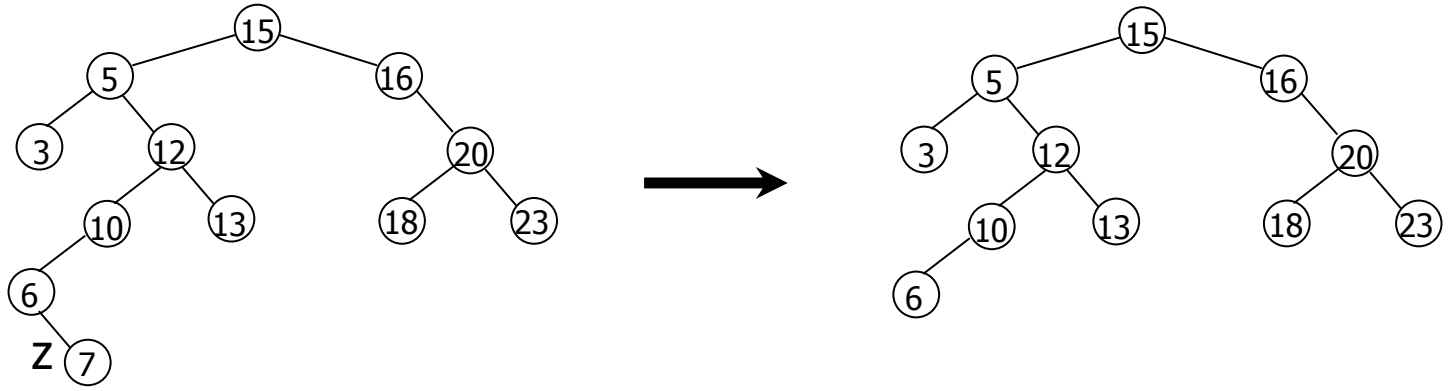


Analiza algoritmilor – arbori binari de căutare (10)

- Exemplu în care z are un singur fiu:



- Exemplu în care z nu are fii:



Analiza algoritmilor – arbori binari de căutare (11)

```
STERGE(z)
    if z.st=NULL and z.dr=NULL then
        y ← z
        if y.p=NULL then
            radacina ← NULL
        else if y=y.p.st then
            y.p.st ← NULL
        else
            y.p.dr ← NULL
    else if z.st=NULL or z.dr=NULL then
        y ← z
        if y.st≠NULL then
            x ← y.st
        else
            x ← y.dr
        if x≠NULL then
            x.p ← y.p
        if y.p=NULL then
            radacina ← x
        else if y=y.p.st then
            y.p.st ← x
        else
            y.p.dr ← x
    else
        y ← SUCCESOR(z)
        if y.st≠NULL then
            x ← y.st
        else
            x ← y.dr
        if x≠NULL then
            x.p ← y.p
        if y.p=NULL then
            radacina ← x
        else if y=y.p.st then
            y.p.st ← x
        else
            y.p.dr ← x
        if y≠z then
            z.cheie ← y.cheie
```

Analiza algoritmilor – arbori binari de căutare (12)

În continuare este prezentată o variantă de ștergere care organizează compact cele trei situații. Algoritmul determină nodul y care se șterge și reface legăturile (în al 3-lea caz chiar și cheia) nodurilor implicate.

```
STERGE(z)
    if z.st=NULL or z.dr=NULL then
        y ← z
    else
        y ← SUCCESOR(z)
    if y.st≠NULL then
        x ← y.st
    else
        x ← y.dr
    if x≠NULL then
        x.p ← y.p
    if y.p=NULL then
        radacina ← x
    else if y=y.p.st then
        y.p.st ← x
    else
        y.p.dr ← x
    if y≠z then
        z.cheie ← y.cheie
    return y
```

Analiza algoritmilor – arbori binari de căutare (13)

Complexitatea operațiilor

- Operațiile de bază pe arborii binari INSEREAZA, CAUTA, MINIM, MAXIM, SUCCESOR și STERGE consumă un timp proporțional cu înălțimea arborelui.
- Pentru un arbore binar relativ echilibrat cu n noduri, aceste operații se execută în cazul cel mai defavorabil într-un timp Θ(lg n). În acest caz, complexitatea temporală poate fi exprimată prin recurența

$$T(n) = T\left(\frac{n}{2}\right) + \Theta(1)$$

  rezolvată deja pentru algoritmul de căutare binară.
- Dacă arborele binar este degenerat (ex. lanț liniar de n noduri), atunci timpul consumat în cazul cel mai defavorabil este Θ(n).
- Operațiile INORDINE, PREORDINE și POSTORDINE au nevoie de un timp Θ(n) pentru traversarea unui arbore binar de căutare cu n noduri.

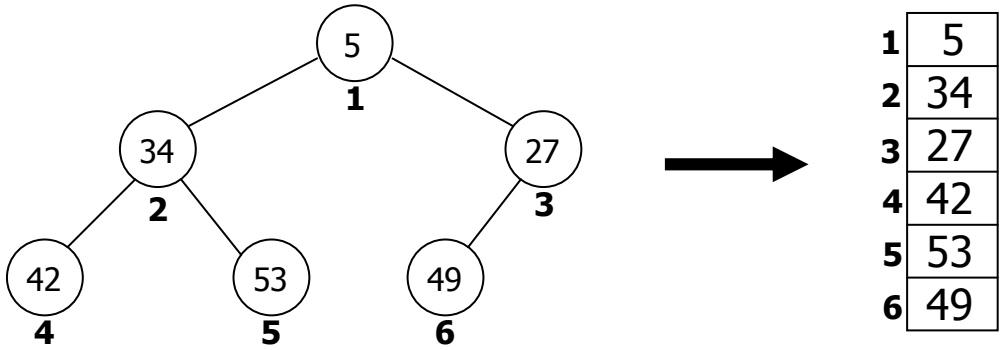
Analiza algoritmilor – arbori binari de căutare (14)

Aplicații

1. Fie secvența de valori: 4, 7, 2, 5, 3, 1, 6.
   a) Reprezentați grafic arborele binar de căutare construit prin inserarea secvenței de valori de mai sus.
   b) În ce ordine se afișează valorile printr-o traversare în preordine? Dar printr-o traversare în postordine?
   c) Care este succesorul nodului cu cheia 4? Dar succesorul nodului cu cheia 6?
   d) Reprezentați grafic arborele binar de căutare obținut după ștergerea nodului 4.
2. Să se completeze clasa BinaryTree prezentată cu traversarea arborelui binar de căutare în inordine, preordine și postordine.
3. Să se implementeze în Java (în clasa BinaryTree) căutarea recursivă și căutarea iterativă a unei chei în arborele binar de căutare.
4. Să se introducă în clasa BinaryTree metode pentru determinarea cheii minime și maxime în arborele binar de căutare.
5. Să se implementeze în clasa BinaryTree o metodă care să identifice succesorul unui nod în arborele binar de căutare.
6. Să se implementeze în clasa BinaryTree algoritmul de ștergere a unui nod din arborele binar de căutare.
7. Să se modifice clasa Node și operațiile implementate în clasa BinaryTree astfel încât cheia să fie numele studentului.
8. Să se rezolve prin metoda iterației recurența aferentă operațiilor de bază (inserare, căutare, minim, maxim, succesor și ștergere) pe un arbore binar de căutare echilibrat cu n noduri.

Analiza algoritmilor – heap-uri (1)

- Heap-ul (movila) este un arbore binar complet (până la ultimul nivel) cu următoarea proprietate:
  - orice nod dintr-un **heap minimizant** trebuie să aibă o cheie mai mică (sau egală) decât oricare dintre fiii săi.
  - orice nod dintr-un **heap maximizant** trebuie să aibă o cheie mai mare (sau egală) decât oricare dintre fiii săi.
- Reprezentarea heap-urilor se face prin tablouri. Într-un tablou indexat de la 1, fiii nodului k au indicii 2k și 2k+1, iar părintele nodului k are indicele k/2. Într-un tablou indexat de la 0 (ca în Java), fiii nodului k au indicii 2k+1 și 2k+2, iar părintele nodului k are indicele (k-1)/2. Exemplu de heap minimizant:

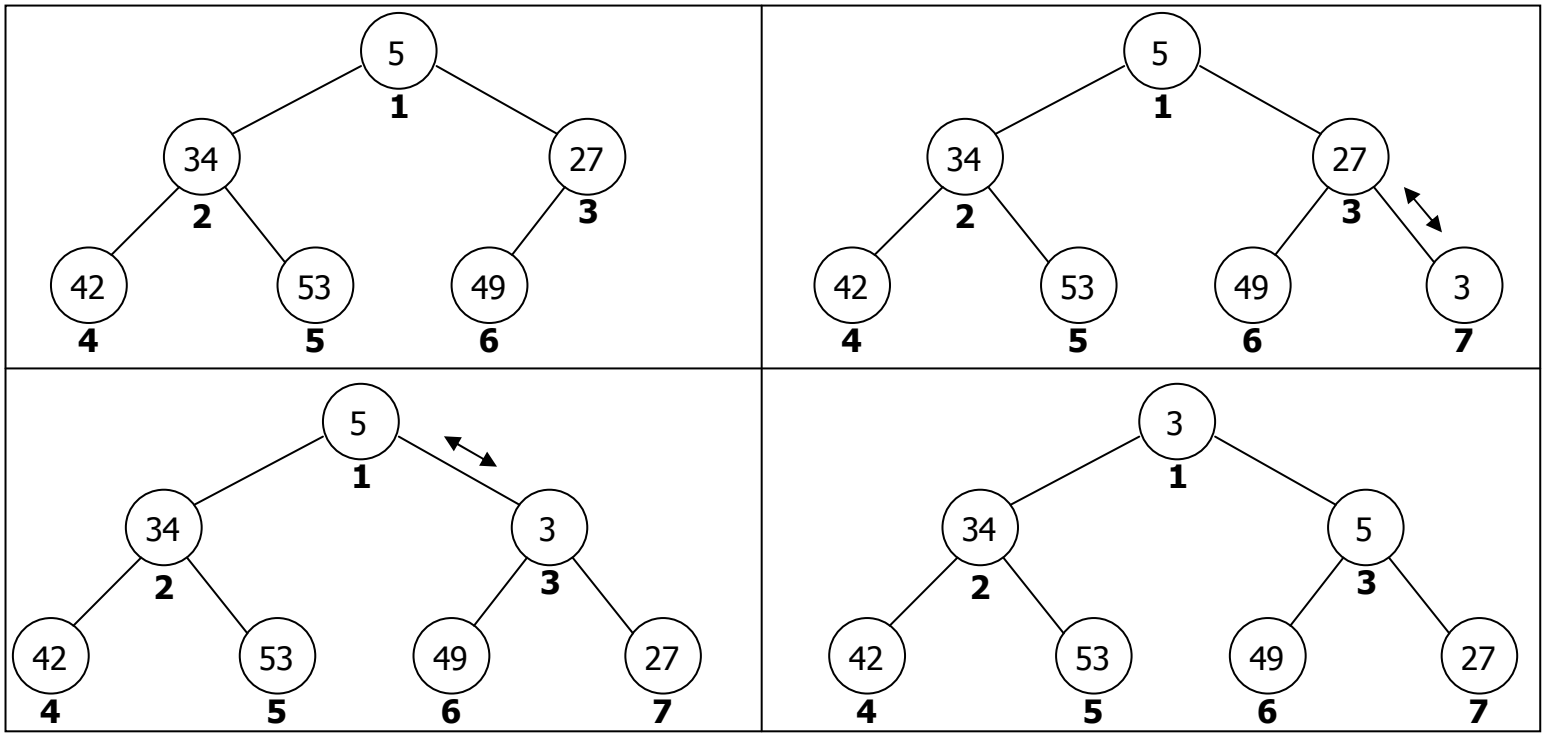


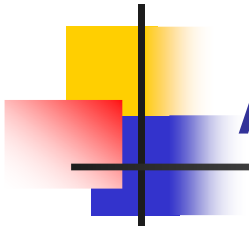
- Heap-ul permite selecția rapidă a elementului cu cheie minimă sau maximă (care este chiar rădăcina), operație realizată în O(1).
- Heap-ul binar este slab ordonat în raport cu arborele binar de căutare, de aceea, traversarea nodurilor în ordine este dificilă.
- Pentru o structură heap A notăm cu **A.size** dimensiunea totală a tabloului respectiv **A.heapsize** dimensiunea heap-ului.

Analiza algoritmilor – heap-uri (2)

Inserarea unui nod

- Constă în două operații:
  - Adăugarea nodului pe ultimul nivel, sau pe un nou nivel dacă arborele este complet. În cazul reprezentării prin tablouri, noul element se adaugă în prima poziție disponibilă.
  - Propagarea în sus (spre rădăcină) a nodului inserat până la îndeplinirea condiției de heap.
- Exemplu de inserare a valorii 3 într-un heap minimizant





Analiza algoritmilor – heap-uri (3)

Determinarea părintelui sau fiilor nodului cu indicele *i* într-un heap reprezentat prin tablou.

```
PARINTE(i)
    return i/2

STANGA(i)
    return 2i

DREAPTA(i)
    return 2i+1
```

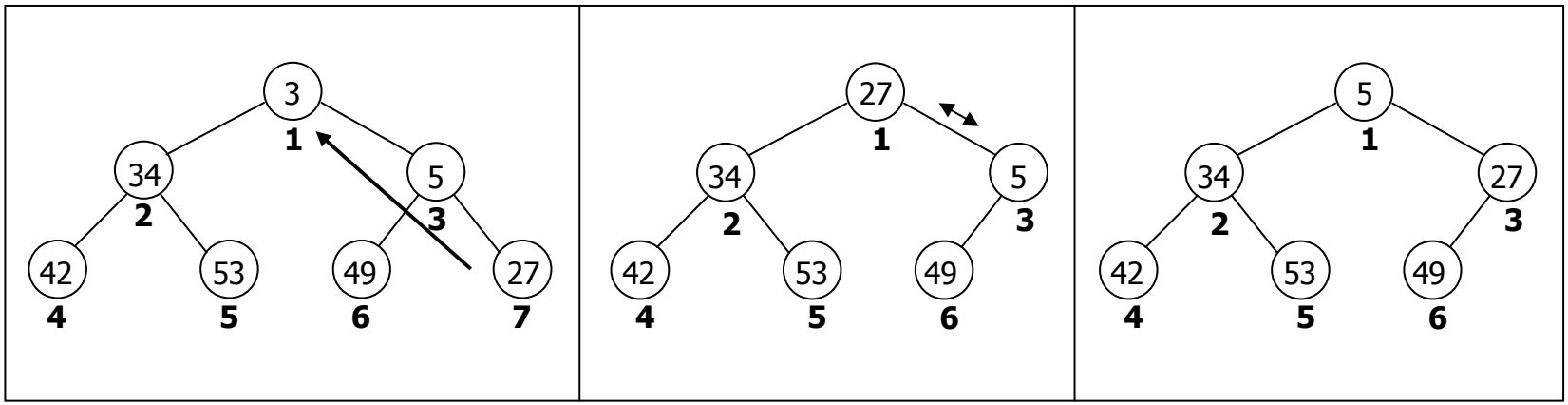
Inserarea nodului z în heap-ul minimizant A:

```
INSEREAZA(A, z)
    A.heapsize ← A.heapsize+1
    i ← A.heapsize
    while i>1 and A[PARINTE(i)]>z do
        A[i] ← A[PARINTE(i)]
        i ← PARINTE(i)
    A[i] ← z
```

Analiza algoritmilor – heap-uri (4)

Extragerea rădăcinii

- Constă în două operații:
  - Copierea ultimului nod în rădăcină.
  - Propagarea în jos a rădăcinii temporare până la îndeplinirea condiției de heap (reconstituirea proprietății de heap). Propagarea se face spre fiul cu cheia cea mai mică într-un heap minimizant respectiv spre fiul cu cheia mai mare într-un heap maximizant.
- Exemplu de extragere a rădăcinii dintr-un heap minimizant

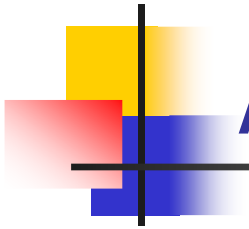


Analiza algoritmilor – heap-uri (5)

La reconstituirea heap-ului se consideră că subarborii care au ca rădăcină STANGA(i) respectiv DREAPTA (i) sunt heap-uri.

```
RECONSTITUIE(A, i)
    s ← STANGA(i)
    d ← DREAPTA(i)
    if s≤A.heapsize and A[s]<A[i] then
        min ← s
    else
        min ← i
    if d≤A.heapsize and A[d]<A[min] then
        min ← d
    if min≠i then
        A[i] ←→ A[min]
        RECONSTITUIE(A, min)
```

```
EXTRAGE(A)
    min ← A[1]
    A[1] ← A[A.heapsize]
    A.heapsize ← A.heapsize-1
    RECONSTITUIE(A, 1)
    return min
```



Analiza algoritmilor – heap-uri (6)

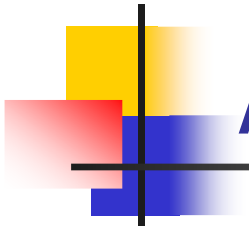
Construcția heap-urilor

- Prin reconstituirea recursivă a proprietății de heap de la nodul n/2 în jos – nodurile n/2+1...n sunt frunze și îndeplinesc condiția de heap:

```
CONSTRUIESTE(A)
    A.heapsize ← A.size
    for i ← A.size/2 downto 1 do
        RECONSTITUIE(A, i)
```

- Prin inserare repetată în heap considerând că primul element al tabloului formează deja un heap:

```
CONSTRUIESTE(A)
    A.heapsize ← 1
    for i ← 2 to A.size do
        INSEREAZA(A, A[i])
```



Analiza algoritmilor – heap-uri (7)

Heapsort

Algoritmul heapsort începe cu apelul procedurii CONSTRUIESTE prin care transformă vectorul de intrare A în heap. Sortarea se face prin interschimbarea repetată a rădăcinii cu ultimul element din heap urmată de excluderea lui din heap. Sortarea este descrescătoare cu un heap minimizant respectiv crescătoare cu heap maximizant.

```
HEAPSORT(A)
    CONSTRUIESTE(A)
    for i ← A.size downto 2 do
        A[1] ←→ A[i]
        A.heapsize ← A.heapsize-1
        RECONSTITUIE(A, 1)
```

Analiza algoritmilor – heap-uri (8)

Complexitatea operațiilor în heap

- Operațiile INSEREAZA, EXTRAGE și RECONSTITUIE sunt de complexitate O(lg n), la fel ca operațiile într-un arbore binar de căutare (v. capitolul precedent).
- Operația de construire prin inserare repetată apelează de O(n) ori INSEREAZA de complexitate O(lg n), deci, timpul total de execuție este O(n lg n).
- În cazul operației de construire prin reconstituire recursivă se observă că nodurile de pe ultimul nivel (aproximativ jumătate) îndeplinesc condiția de heap, deci nu sunt propagate deloc. Nodurile de pe nivelul deasupra frunzelor (aproximativ un sfert) sunt propagate cel mult un nivel. Nodul aflat în rădăcină este propagat cel mult h-1 nivele, ($\lg n$ rotunjit în jos). Astfel, numărul de propagări este

$$\frac{n}{2}\cdot 0+\frac{n}{4}\cdot 1+\frac{n}{8}\cdot 2+\ldots+\frac{n}{2^{k+1}}\cdot k=\sum_{k=0}^{\lfloor \lg n\rfloor}\frac{n}{2^{k+1}}\cdot O(k)=O\left(n\sum_{k=0}^{\lfloor \lg n\rfloor}\frac{k}{2^{k+1}}\right)$$

Știm că pentru orice x subunitar $\sum_{k=0}^{\infty}k\cdot x^k=\frac{x}{(1-x)^2}$

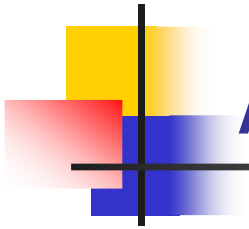
Astfel rezultă că $O\left(n\sum_{k=0}^{\lfloor \lg n\rfloor}\frac{k}{2^{k+1}}\right)=O\left(\frac{n}{2}\cdot\sum_{k=0}^{\infty}k\cdot\left(\frac{1}{2}\right)^k\right)=O\left(\frac{n}{2}\cdot\frac{\frac{1}{2}}{\left(1-\frac{1}{2}\right)^2}\right)=O(n)$

- Timpul de execuție al algoritmului heapsort este O(n lg n) deoarece CONSTRUIESTE se execută în O(n) iar operația RECONSTITUIE de O(lg n) este apelată de n-1 ori.

Analiza algoritmilor – heap-uri (9)

Aplicații

1. Ilustrați modul de funcționare al procedurii CONSTRUIESTE, prin reconstituire heap minimizant, pentru vectorul A={5,3,17,10,84,19,6,22,9}.
2. Ilustrați modul de funcționare al procedurii CONSTRUIESTE, prin inserare în heap minimizant, pentru vectorul A={5,3,17,10,84,19,6,22,9}.
3. Ilustrați modul de funcționare al procedurii CONSTRUIESTE, prin reconstituire heap maximizant, pentru vectorul A={5,3,17,10,84,19,6,22,9}.
4. Ilustrați modul de funcționare al procedurii CONSTRUIESTE, prin inserare în heap maximizant, pentru vectorul A={5,3,17,10,84,19,6,22,9}.
5. Ilustrați modul de funcționare al procedurii HEAPSORT pentru vectorul A={5,3,17,10,84,19,6,22,9}.
6. Să se implementeze în Java operațiile de construire, inserare, extragere și reconstituire.
7. Să se implementeze în Java algoritmul heapsort.
8. Să se rescrie funcția RECONSTITUIE pentru un heap maximizant.
9. Să se implementeze în Java algoritmul heapsort folosind un heap maximizant, astfel încât sortarea să se facă în ordine crescătoare.
10. Să se adapteze și să se reutilizeze clasa Node de la arbori binari de căutare (cu informație suplimentară nume student) pentru heap-uri.

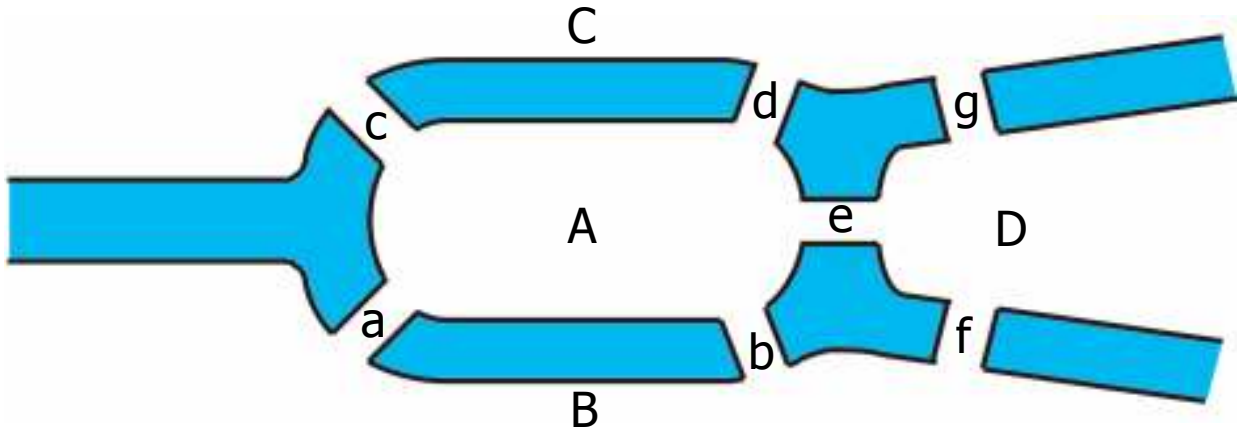


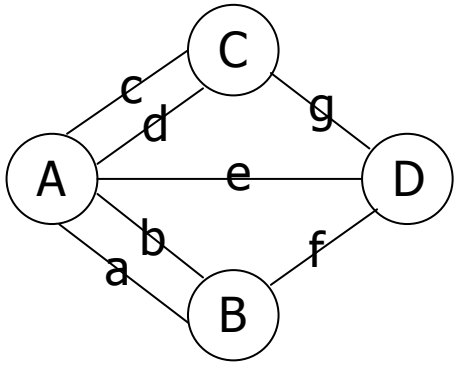
Analiza algoritmilor – grafuri (1)

- Grafurile au o formă determinată de o problemă concretă.
- Nodurile grafului se numesc **vârfuri** și ele pot fi conectate prin **muchii**. Notăm un graf G=(V,E), unde V este mulțimea vârfurilor și E este mulțimea muchiilor. Notăm cu |V| numărul vârfurilor și cu |E| numărul muchiilor.
- Fiecare vârf conține câmpurile i (index), c (color), d (distance/discovered), p (predecessor), f (finalized).
- Două vârfuri se numesc **adiacente** dacă sunt conectate direct printr-o muchie.
- Un **drum** reprezintă o secvență de muchii.
- Un graf se numește **conex** dacă există cel puțin un drum între toate vârfurile.
- Un graf în care muchiile nu au o direcție (deplasarea se poate face în orice sens) se numește **neorientat**. Un graf în care muchiile au o singură direcție (marcată prin săgeată) se numește **orientat**.
- Un graf în care muchiile au asociate ponderi (costul drumurilor dintre vârfuri) se numește **ponderat**.
- **Complexitatea** unui algoritm pe un graf G=(V,E) se exprimă în funcție de dimensiunea intrării descrisă prin doi parametri: numărul de vârfuri |V| și numărul de muchii |E|. În cadrul notației asimptotice simbolul V înseamnă |V|, iar simbolul E înseamnă |E|.

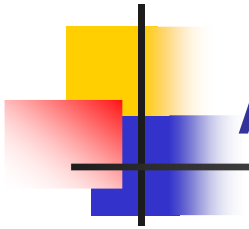
Analiza algoritmilor – grafuri (2)

Podurile din Königsberg





Unul dintre primii matematicieni care a studiat grafurile a fost **Leonhard Euler**, în secolul al XVIII-lea. El a rezolvat celebra problemă a podurilor din Königsberg, demonstrând că nu există un drum care să cuprindă toate cele șapte poduri, fără a traversa de două ori același pod.



Analiza algoritmilor – grafuri (3)

Implementarea clasei Vertex în Java

Într-o implementare abstractă vârfurile pot fi valori întregi. În majoritatea aplicațiilor însă pentru fiecare vârf trebuie păstrate mai multe informații, ca în clasa *Vertex* de mai jos:

```
package graphs;
import java.awt.*;

public class Vertex {

  Color color;
  Vertex predecessor;
  int distance;
  int discovered; //for DFS
  int finalized; //for DFS
  char label;
  int index;
  static int infinite = 100;

  public Vertex(int i) {
    index = i;
    color = Color.white;
    distance = infinite;
    predecessor = null;
  }
}
```

Analiza algoritmilor – grafuri (4)

Reprezentarea grafurilor prin matrice de adiacență

- Matricea de adiacență este un tablou bidimensional, în care elementele indică prezența unei muchii între două vârfuri. Dacă graful are |V| vârfuri, numerotate cu 1,2,...,|V| în mod arbitrar, matricea de adiacență este un tablou $A=(a_{ij})$ cu |V|x|V| elemente, unde

$$a_{ij} = \begin{cases} 1 \ dacă \ (i,j) \in E, \\ 0 \ altfel. \end{cases}$$

- Matricea de adiacență poate fi folosită și pentru grafuri ponderate. În acest caz, costul w(i,j) al unei muchii (i,j) ∈ E este memorat ca element din linia *i* și coloana *j* a matricei de adiacență:

$$a_{ij} = \begin{cases} w(i,j) \ dacă \ (i,j) \in E, \\ 0 \ altfel. \end{cases}$$

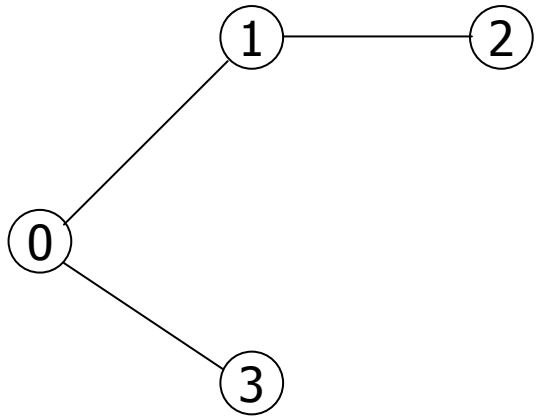
- Avantajul constă în verificarea rapidă a adiacențelor. Dezavantajul constă în faptul că necesarul de memorie pentru matricea de adiacență a unui graf este $\Theta(V^2)$ și nu depinde de numărul de muchii. Astfel, reprezentarea prin matrice de adiacență este indicată în cazul grafurilor cu un număr relativ mic de vârfuri sau atunci când graful este dens (|E| aproximativ egal cu $|V|^2$).

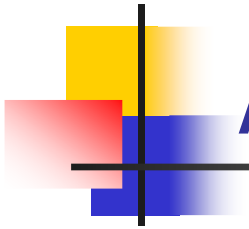
Analiza algoritmilor – grafuri (5)

Implementarea în Java a unui graf neponderat reprezentat prin matrice de adiacență

```
package graphs;
import java.awt.*;
import java.util.*;

public class Graph {
  int maxVerts = 20;
  int nVerts = 0;
  ArrayList<Vertex> vertexList = new ArrayList();
  int A[][] = new int[maxVerts][maxVerts];

  public Graph() {
    for(int i=0; i<maxVerts; i++)
      for(int j=0; j<maxVerts; j++)
        A[i][j] = 0;
  }

  public void addVertex(){
    vertexList.add(new Vertex(nVerts++));
  }

  public void addEdge(int v1, int v2){
    A[v1][v2] = 1;
    A[v2][v1] = 1;
  }
```

```
  public static void main(String[] args) {
    Graph graph = new Graph();
    graph.addVertex(); //0
    graph.addVertex(); //1
    graph.addVertex(); //2
    graph.addVertex(); //3
    graph.addEdge(0, 1);
    graph.addEdge(1, 2);
    graph.addEdge(0, 3);
  }
}
```





Analiza algoritmilor – grafuri (6)

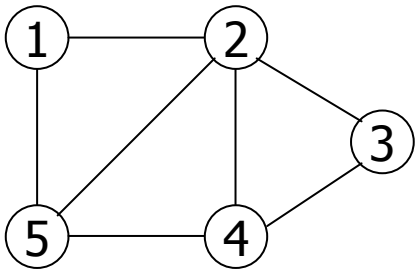
Reprezentarea grafurilor prin liste de adiacență

- Se folosește un tablou A cu |V| liste, câte una pentru fiecare vârf din V. Pentru fiecare i ϵ V, lista de adiacență A[i] conține toate vârfurile j pentru care există o muchie (i,j) ϵ E.
- Dacă graful este orientat, suma lungimilor tuturor listelor de adiacență este |E|. Dacă graful este neorientat, lungimea totală a listelor de adiacență este 2|E| deoarece, dacă (i,j) este o muchie, atunci *i* apare în lista de adiacență a lui *j* dar și *j* apare în lista de adiacență a lui *i*.
- Dacă graful este ponderat, costul w(i,j) al muchiei (i,j) ϵ E este memorat împreună cu vârful *j* în lista de adiacență a lui *i*.
- Avantajul constă în necesarul de memorie O(max(V,E)). Reprezentarea prin liste de adiacență este preferată pentru că oferă un mod compact de reprezentare a grafurilor rare (cu |E| mult mai mic decât $|V|^2$). Dezavantajul constă în faptul că verificarea adiacențelor implică operații de căutare în liste.

Analiza algoritmilor – grafuri (7)

Reprezentarea grafurilor neponderate. Exemple

Graf neorientat



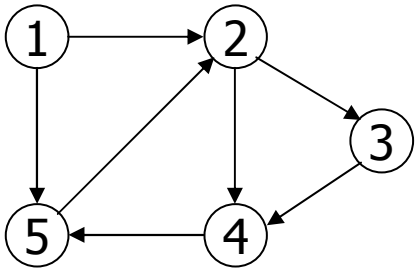
Listele de adiacență

1 → 2 → 5
2 → 1 → 3 → 4 → 5
3 → 2 → 4
4 → 2 → 3 → 5
5 → 1 → 2 → 4

Matricea de adiacență

	1	**2**	**3**	**4**	**5**
1	0	1	0	0	1
2	1	0	1	1	1
3	0	1	0	1	0
4	0	1	1	0	1
5	1	1	0	1	0

Graf orientat



Listele de adiacență

1 → 2 → 5
2 → 3 → 4
3 → 4
4 → 5
5 → 2

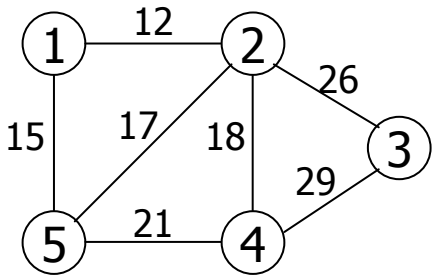
Matricea de adiacență

	1	**2**	**3**	**4**	**5**
1	0	1	0	0	1
2	0	0	1	1	0
3	0	0	0	1	0
4	0	0	0	0	1
5	0	1	0	0	0

Analiza algoritmilor – grafuri (8)

Reprezentarea grafurilor ponderate. Exemple

Graf neorientat



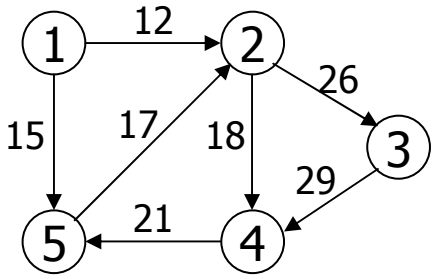
Listele de adiacență

1 → 2/12 → 5/15
2 → 1/12 → 3/26 → 4/18 → 5/17
3 → 2/26 → 4/29
4 → 2/18 → 3/29 → 5/21
5 → 1/15 → 2/17 → 4/21

Matricea de adiacență

	1	**2**	**3**	**4**	**5**
1	0	12	0	0	15
2	12	0	26	18	17
3	0	26	0	29	0
4	0	18	29	0	21
5	15	17	0	21	0

Graf orientat

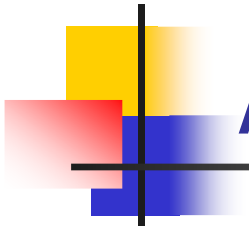


Listele de adiacență

1 → 2/12 → 5/15
2 → 3/26 → 4/18
3 → 4/29
4 → 5/21
5 → 2/17

Matricea de adiacență

	1	**2**	**3**	**4**	**5**
1	0	12	0	0	15
2	0	0	26	18	0
3	0	0	0	29	0
4	0	0	0	0	21
5	0	17	0	0	0



Analiza algoritmilor – grafuri (9)

Parcurgerea grafurilor în lățime (breadth-first search)

- Parcurgerea în lățime a unui graf G=(V,E) pornește de la un vârf sursă *s* și explorează sistematic graful descoperind toate vârfurile accesibile din *s*. Algoritmul calculează distanța de la *s* la toate aceste vârfuri accesibile.
- Parcurgerea în lățime lărgește uniform frontiera dintre vârfurile descoperite și cele nedescoperite. Astfel, algoritmul descoperă toate vârfurile aflate la distanța k față de s înainte de cele aflate la distanța k+1.
- Acest algoritm este implementat cu ajutorul unei cozi Q.
- Pentru a ține evidența avansării, parcurgerea în lățime colorează fiecare vârf cu alb (nedescoperit), gri (descoperit, cu lista de adiacență în curs de explorare) sau negru (descoperit, cu lista de adiacență explorată).
- Algoritmul păstrează pentru fiecare vârf v ∈ V culoarea în v.c, predecesorul în v.p și distanța de la sursa s în câmpul v.d.
- Complexitatea algoritmului constă în operațiile efectuate cu coada (fiecare vârf este inserat și scos o singură dată) cu un cost O(V) și în examinarea listelor de adiacență cu un cost O(E), astfel timpul total de execuție este O(V+E).

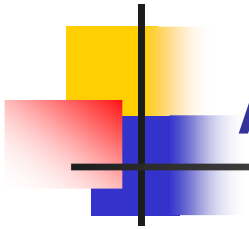
Analiza algoritmilor – grafuri (10)

```
BFS(G, s)
    foreach u ϵ V do
        u.c ← ALB
        u.d ← ∞
        u.p ← NULL
    s.c ← GRI
    s.d ← 0
    Q.INSERT(s)
    while Q≠Ø do
        u ← Q.HEAD
        PRINT(u.i)
        foreach v ϵ A[u] do
            if v.c=ALB then
                v.c ← GRI
                v.d ← u.d+1
                v.p ← u
                Q.INSERT(v)
        Q.DELETE(Q.HEAD)
        u.c ← NEGRU
```

Analiza algoritmilor – grafuri (11)

Implementarea în Java a parcurgerii grafurilor în lățime

```
public void BFS(int s){
  for(int u=0; u<nVerts; u++){
    vertexList.get(u).color = Color.white;
    vertexList.get(u).distance = Vertex.infinite;
    vertexList.get(u).predecessor = null;
  }
  vertexList.get(s).color = Color.gray;
  vertexList.get(s).distance = 0;
  LinkedList<Vertex> queue = new LinkedList();
  queue.addFirst(vertexList.get(s));
  while(!queue.isEmpty()){
    Vertex u = queue.getLast();
    System.out.println(u.index);
    for(int v=0; v<nVerts; v++)
      if(A[v][u.index]==1 && vertexList.get(v).color==Color.white){
        vertexList.get(v).color = Color.gray;
        vertexList.get(v).distance = u.distance+1;
        vertexList.get(v).predecessor = u;
        queue.addFirst(vertexList.get(v));
      }
    queue.removeLast();
    u.color = Color.black;
  }
}
```



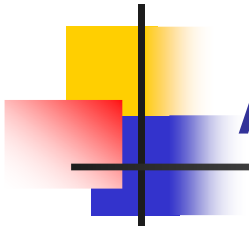
Analiza algoritmilor – grafuri (12)

Parcurgerea grafurilor în adâncime (depth-first search)

- Algoritmul explorează în adâncime subgraful succesor al celui mai recent descoperit vârf *v* dintr-un graf G=(V,E). Când toate muchiile care pleacă din *v* au fost explorate, algoritmul revine pentru explorarea următoarelor muchii care pleacă din predecesorul lui *v*.
- Parcurgerea în adâncime se implementează cu ajutorul stivei, fiind astfel naturală o abordare recursivă.
- Pentru a ține evidența avansării, parcurgerea în adâncime colorează fiecare vârf v ∈ V cu alb (nedescoperit), gri (descoperit, cu zona de graf accesibilă din *v* în curs de explorare) sau negru (descoperit, cu zona de graf accesibilă din *v* explorată).
- Algoritmul păstrează pentru fiecare vârf v ∈ V culoarea în v.c și predecesorul în v.p. De asemenea, algoritmul folosește un ceas *t* al parcurgerii care crește atunci când un nod își schimbă culoarea. Algoritmul creează marcaje de timp pentru fiecare vârf v ∈ V: momentul când v este descoperit (devine gri) este păstrat în v.d, iar momentul în care explorarea zonei grafului accesibilă din *v* a fost finalizată este păstrat în v.f.
- Marcajele de timp precum și predecesorii se pot folosi pentru o serie extinsă de prelucrări.
- Complexitatea algoritmului constă în explorarea vârfurilor Θ(V) și în examinarea listelor de adiacență Θ(E), astfel timpul total de execuție este Θ(V+E).

Analiza algoritmilor – grafuri (13)

```
DFS(G)
    foreach u ϵ V do
        u.c ← ALB
        u.p ← NULL
    t ← 0
    foreach u ϵ V do
        if u.c=ALB then
            EXPLOREAZA(u)


EXPLOREAZA(u)
    u.c ← GRI
    PRINT(u.i)
    u.d ← t ← t+1
    foreach v ϵ A[u] do
        if v.c=ALB then
            v.p ← u
            EXPLOREAZA(v)
    u.c ← NEGRU
    u.f ← t ← t+1
```



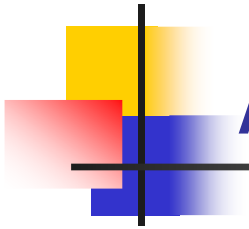
Analiza algoritmilor – grafuri (14)

Drumuri minime. Algoritmul Dijkstra

- Algoritmul de căutare în lățime determină drumurile minime în grafuri neponderate. Algoritmul Dijkstra permite determinarea drumurilor minime de la vârful sursă *s* către toate vârfurile unui graf ponderat cu costuri nenegative. Astfel, vom presupune că $w(u,v) \geq 0$ pentru fiecare muchie $(u,v) \in E$.
- Pentru fiecare vârf $v \in V$ algoritmul păstrează în v.d estimarea drumului minim față de *s*, iar în v.p păstrează predecesorul. Algoritmul folosește o coadă de priorități Q inițializată cu V (conține toate vârfurile grafului).
- La fiecare iterație *while*, se extrage din Q vârful *u* cu drumul minim. Pentru fiecare vârf *v* adiacent cu *u* se reactualizează v.d și v.p în cazul în care drumul minim către *v* poate fi ameliorat dacă trece prin *u* (operație de relaxare a muchiilor).
- Complexitatea algoritmului constă în extragerea minimului (operație O(V) efectuată de |V| ori) cu un cost $O(V^2)$ și în examinarea listelor de adiacență (fiecare muchie este examinată o singură dată) cu un cost O(E), astfel timpul total de execuție este $O(V^2+E)=O(V^2)$.

Analiza algoritmilor – grafuri (15)

```
DIJKSTRA(G, w, s)
    foreach u ϵ V do
        u.d ← ∞
        u.p ← NULL
    s.d ← 0
    Q ← V
    while Q≠Ø do
        u ← MIN(Q, d)
        Q ← Q-{u}
        foreach v ϵ A[u] do
            if v.d>u.d+w(u,v) then
                v.d ← u.d+w(u,v)
                v.p ← u
```



Analiza algoritmilor – grafuri (16)

Căutare în spațiul stărilor: algoritmul A*

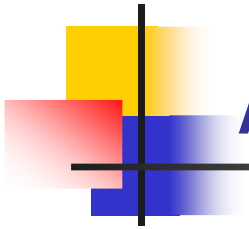
- Algoritmul A*, introdus în 1968 de Hart, Nilsson şi Raphael, este o extensie şi în acelaşi timp o variantă îmbunătăţită a algoritmului Dijkstra care permite căutarea unei căi eficiente între două vârfuri ale unui graf.

- Pe lângă estimarea drumului minim faţă de vârful sursă *s* algoritmul foloseşte şi o funcţie euristică pentru estimarea distanţei faţă de vârful ţintă *t*.

- La fiecare iteraţie algoritmul extrage din coada de priorităţi Q vârful *u* estimat cu distanţa *s-u-t* minimă.

Analiza algoritmilor – grafuri (17)

```
A*(G, w, s, t)                          s=sursă, t=țintă/target
    foreach u ϵ V do
        u.d ← ∞
        u.h ← ∞
        u.f ← ∞
        u.p ← NULL
    s.d ← 0
    s.h ← H(s, t)                       H=heuristic
    s.f ← s.h
    Q ← {s}                             Q=openset
    C ← Ø                               C=closedset
    while Q≠Ø do
        u ← MIN(Q, f)
        if u=t then return true
        Q ← Q - {u}
        C ← C ∪ {u}
        foreach v ϵ A[u] do
            if v ϵ C then continue
            if v ∉ Q then Q ← Q ∪ {v}
            if v.d>u.d+w(u,v) then
                v.d ← u.d+w(u,v)
                v.h ← H(v, t)
                v.f ← v.d+v.h
                v.p ← u
    return false
```



Analiza algoritmilor – grafuri (18)

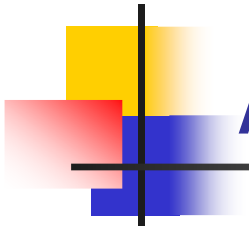
Algoritmul Kruskal

- Algoritml Kruskal este folosit pentru determinarea arborelui parțial de cost minim G*=(V, Q) dintr-un graf conex ponderat G=(V,E), adică arborele care să conțină toate vârfurile și care să fie minimizat din punct de vedere al ponderilor.
- Inițial Q este vid.
- Muchiile din E sunt sortate crescător după ponderi.
- Aceste muchii sunt adăugate apoi în Q, cu condiția să nu formeze un ciclu.
- La final se returnează Q conținând muchiile arborelui parțial de cost minim.

Analiza algoritmilor – grafuri (19)

```
KRUSKAL(G)
    Q ← Ø
    foreach u ∈ V do
            CREEAZA_MULTIME(v)
    SORTEAZA(E)
    foreach (u,v) ∈ E do
            if GASESTE(u) ≠ GASESTE(v) then
                    Q ← Q ∪ (u,v)
                    REUNESTE(u, v)
    return Q
```

Etapa cea mai costisitoare din algoritm este sortarea setului de muchii E. Astfel, complexitatea algoritmului este O(E log E).



Analiza algoritmilor – grafuri (20)

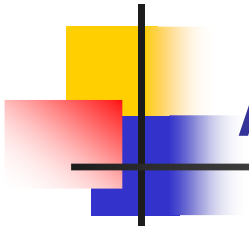
Algoritmul Prim

- Algoritmul Prim determină arborele parțial de cost minim G*=(V, Q) dintr-un graf ponderat G=(V,E) pornind de la un vârf sursă *s*, care este adăugat în U.
- Și în acest algoritm Q este inițial vid.
- Se formează două seturi de vârfuri: U conținând vârfurile vizitate și V-U cu vârfurile nevizitate.
- Vârfurile se mută pe rând din V-U în U considerând muchiile cu cost minim, care sunt și ele adăugate în Q.

Analiza algoritmilor – grafuri (21)

```
PRIM(G, s)
    Q ← Ø
    U ← {s}
    while U ≠ V
        GASESTE_MIN(u, v), u ∈ U, v ∈ V-U
        Q ← Q ∪ (u,v)
        U ← U ∪ {v}
    return Q
```

Într-o implementare cu matrice de adiacență, complexitatea algoritmului constă în parcurgerea vârfurilor și, pentru fiecare din ele, găsirea acelui vârf cu care muchia este de cost minim, așadar implică $O(V^2)$ operații.



Analiza algoritmilor – grafuri (22)

Aplicații

1. Să se modifice clasa Graph, inclusiv metoda BFS, astfel încât grafurile să fie reprezentate prin liste de adiacență în loc de matrice de adiacență.

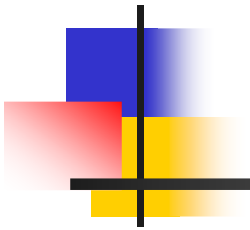
2. Să se implementeze în clasa Graph algoritmul de parcurgere a grafurilor în adâncime.

3. Să se implementeze în clasa Graph algoritmul Dijkstra care să afișeze drumurile minime într-un graf ponderat. Ponderile notate cu w(u,v) pot fi păstrate în matricea de adiacență A(u,v) – pentru asta e necesară adăugarea unei noi metode *addEdge* în clasa Graph.

4. Să se implementeze în clasa Graph algoritmul Kruskal.

5. Să se implementeze în clasa *Graph* algoritmul Prim.

6. Să se implementeze un algoritm de colorare a grafurilor



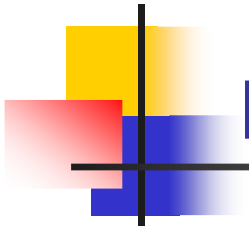
Partea III

Proiectarea algoritmilor

Proiectarea algoritmilor – divide et impera (1)

- **Divide et impera** este o metodă de construire a algoritmilor. Rezolvarea unei probleme prin această metodă constă în:
  - împărțirea problemei în două sau mai multe subprobleme de dimensiune mai mică.
  - rezolvarea subproblemelor.
  - combinarea rezultatelor pentru obținerea soluției problemei inițiale.
- Divizarea problemei se poate face recursiv până când subproblemele devin elementare și pot fi rezolvate direct.
- Forma generală a algoritmului de rezolvare a problemei P de dimensiune *n* cu soluția S:

```
DIVIZARE(P, n, S)
    if n≤n0 then
        SOLUTIE(P, n, S)
    else
        SEPARARE (P, n) în (P1, n1), ..., (Pk, nk)
        DIVIZARE(P1, n1, S1)
        ...
        DIVIZARE(Pk, nk, Sk)
        COMBINARE (S1, ..., Sk) în S
```



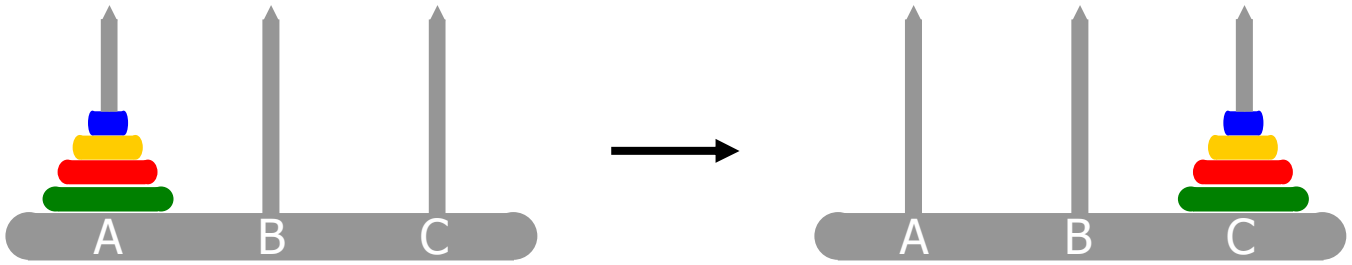
Proiectarea algoritmilor – divide et impera (2)

Turnurile din Hanoi

- Problema turnurilor din Hanoi a fost introdusă în 1883 de matematicianul francez Édouard Lucas. Problema are la bază o legendă hindusă conform căreia zeul Brahma a așezat în templul din Benares o stivă de 64 de discuri din aur cu diametre diferite, pe o tijă, în ordinea descrescătoare a diametrelor. Călugării templului au sarcina de a muta toate discurile pe o altă tijă, folosind și o tijă intermediară, astfel încât:
  - La un moment dat doar un disc, aflat în vârful unei tije, poate fi mutat pe o altă tijă;
  - Nu este permisă așezarea unui disc de dimensiune mai mare peste un disc de dimensiune mai mică.
- Conform legendei, lumea se va sfârși atunci când călugării își vor termina treaba.

Proiectarea algoritmilor – divide et impera (3)

Algoritmul HANOI

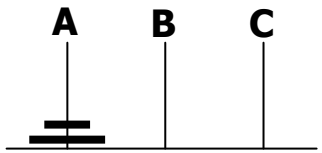


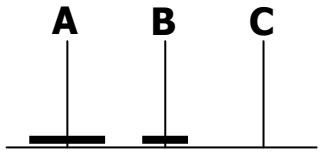
Algoritmul mută n discuri de pe tija A pe tija C folosind tija B. Pentru asta, trebuie mutate n-1 discuri de pe A pe B prin C, ultimul disc de pe A pe C și apoi n-1 discuri de pe B pe C prin A.

```
HANOI(n, A, C, B)
    if n≥1 then
        HANOI(n-1, A, B, C)
        PRINT(A → C)
        HANOI(n-1, B, C, A)
```

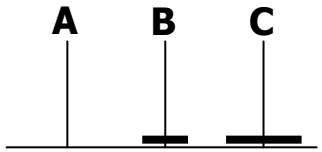
Proiectarea algoritmilor – divide et impera (4)

Exemplu (n=2)

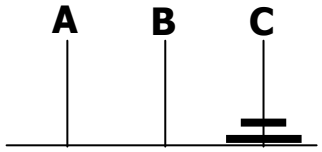




A → B



A → C



B → C

Complexitatea algoritmului HANOI

Numărul de mutări poate fi descris prin recurența

$$T(n) = T(n-1) + 1 + T(n-1) = 2T(n-1) + 1$$

Aplicăm metoda iterației:

$$T(n) = 2^1 \cdot T(n-1) + 2^0$$
$$2T(n-1) = 2 \cdot \left(2T(n-2) + 1\right) = 2^2 \cdot T(n-2) + 2^1$$
$$2^2 \cdot T(n-2) = 2^2 \cdot \left(2T(n-3) + 1\right) = 2^3 \cdot T(n-3) + 2^2$$
$$2^{q-1} \cdot T(n-q+1) = 2^q \cdot T(n-q) + 2^{q-1}$$

Considerând condiția la limită T(0)=0, recurența se termină pentru

$$n - q = 0 \Rightarrow q = n$$

Proiectarea algoritmilor – divide et impera (6)

Am arătat că la limită T(0)=0, q=n. Obținem

$$T(n) = 2^n \cdot T(0) + \sum_{k=0}^{n-1} 2^k$$

$$T(n) = 2^n \cdot 0 + \sum_{k=0}^{n-1} 2^k = \sum_{k=0}^{n-1} 2^k$$

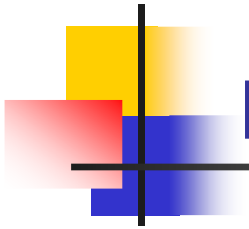
Știm că

$$\sum_{k=0}^{n} x^k = \frac{x^{n+1} - 1}{x - 1}$$

Rezultă astfel că

$$T(n) = \frac{2^n - 1}{2 - 1} = 2^n - 1$$

Complexitatea algoritmului este $O(2^n)$, deci călugării mai au mult de lucru...



Proiectarea algoritmilor – divide et impera (7)

Aplicații

1. Care din algoritmii prezentați anterior au la bază metoda divide et impera?
2. Să se implementeze în Java algoritmul de rezolvare a problemei turnurilor din Hanoi.
3. Să se implementeze în Java varianta problemei turnurilor din Hanoi cu 4 tije.
4. Se dă un tablou de valori întregi. Să se determine cel mai mare divizor comun al valorilor din tablou prin metoda divide et impera. Este evident că putem calcula separat cel mai mare divizor comun pentru cele două jumătăți ale tabloului, iar soluția este cel mai mare divizor comun al celor două. Procesul de divizare continuă până când se ajunge la subtablouri de un element.

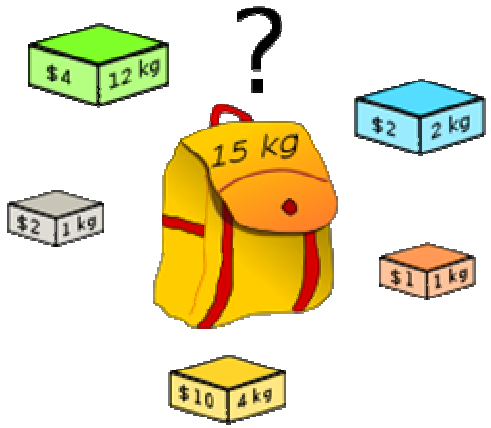
Proiectarea algoritmilor – greedy (1)

Algoritmii greedy aleg la fiecare moment cel mai bun candidat posibil. Metoda greedy face optimizări locale, cu speranța că acestea vor conduce la un optim global. Acești algoritmi sunt de obicei rapizi și furnizează o soluție relativ bună, dar nu întotdeauna cea mai bună. Forma generală a unui algoritm greedy:

```
GREEDY(C)
    S ← Ø
    while C≠Ø do
        x ← BEST(C)
        C ← C-{x}
        if FEASIBLE(S ∪ {x}) then
            S ← S ∪ {x}
```

Proiectarea algoritmilor – greedy (2)

Problema fracționară a rucsacului



Se consideră că dispunem de un rucsac cu o anumită capacitate *G* (greutate maximă) și de *n* obiecte, definite prin greutate *g* și preț *p*. Să se umple rucsacul cu obiecte, astfel încât valoarea totală să fie maximă. Pot fi introduse și fracțiuni din obiecte.

Proiectarea algoritmilor – greedy (3)

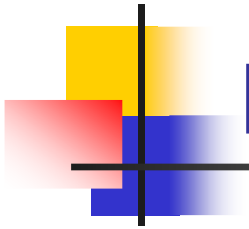
- Algoritmul greedy pentru rezolvarea problemei fracționare a rucsacului constă în următorii pași:

  1. v[i] ← p[i]/g[i], $i = \overline{1,n}$;
  2. sortează vectorii v, p, g în ordinea descrescătoare a valorilor din v;
  3. caută k astfel încât

$$\sum_{i=1}^{k} g_i \le G < \sum_{j=1}^{k+1} g_j$$

soluția fiind $\begin{cases} g_i, \ pentru \ \ i = \overline{1,k} \\ G - \sum_{i=1}^{k} g_i, \ pentru \ k+1 \end{cases}$

- Timpul de execuție al algoritmului este O(n lg n), deoarece sortarea se execută în O(n lg n), iar complexitatea operației de căutare din pasul 3 este O(n).



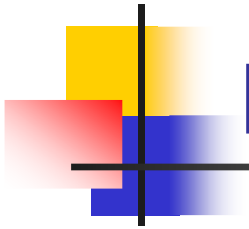
Proiectarea algoritmilor – greedy (4)

Coduri Huffman

- Codificarea Huffman este o tehnică foarte utilă pentru compactarea datelor. Algoritmul greedy propus de David Albert Huffman oferă o modalitate optimă de reprezentare a caracterelor prin coduri binare unice.
- Fie un fișier cu 100 de caractere care conține doar literele a-f, cu următoarele frecvențe:

Caracter	a	b	c	d	e	f
Frecvență	45	13	12	16	9	5
Cod cu lungime fixă	000	001	010	011	100	101
Cod cu lungime variabilă	0	101	100	111	1101	1100

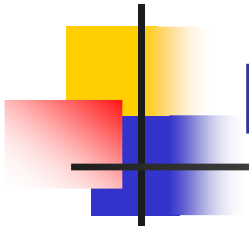
- Dacă folosim un cod binar de lungime fixă, avem nevoie de trei biți pentru a reprezenta șase caractere (a=000, b=001, ..., f=101). Această metodă necesită 300 de biți pentru codificarea celor 100 de caractere din fișier.
- O codificare cu lungime variabilă, care atribuie coduri scurte caracterelor frecvente și coduri lungi caracterelor cu frecvență redusă, poate codifica fișierul prin 224 de biți (45x1+13x3+12x3+16x3+9x4+5x4), economisind 25% din spațiu.
- O codificare optimă pentru un fișier este reprezentată printr-un arbore binar complet.



Proiectarea algoritmilor – greedy (5)

Decodificarea

- Codurile și lungimile acestora se generează de algoritm astfel încât distincția simbolurilor este asigurată.
- În exemplul anterior
  - a = 0
  - b = 101
  - c = 100
  - d = 111
  - e = 1101
  - f = 1100
- Astfel, dacă un cod începe cu 0 e format dintr-un singur bit și este "a". Dacă începe cu 1, sigur e format din cel puțin trei biți, iar dacă primii trei biți sunt 110 înseamnă că există și un al patrulea bit, etc.
- De exemplu, decodificând pas cu pas secvența binară 1000110011010, veți obține cuvântul "cafea".



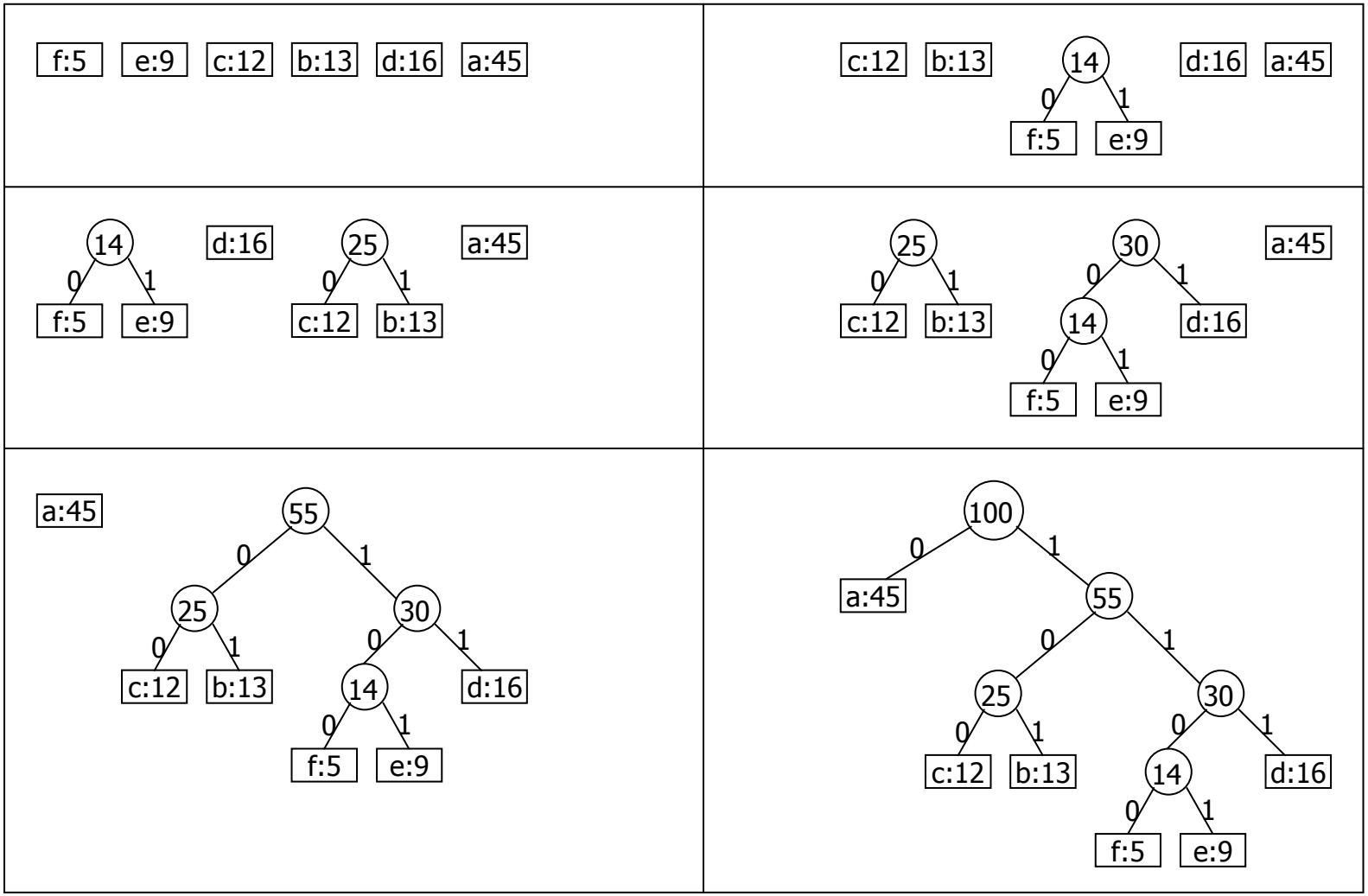
Proiectarea algoritmilor – greedy (6)

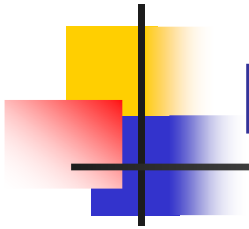
Construcția unui cod Huffman

- Algoritmul Huffman pornește de la |C| frunze, fiecare reprezentând câte un caracter c ∈ C cu o frecvență dată f[c].
- Se realizează |C-1| operații de fuzionare, până la obținerea arborelui final.
- La fiecare pas, se aleg doi arbori (inițial frunze) cu frecvența cea mai redusă și se înlocuiesc cu arborele obținut prin fuzionarea lor. Frecvența noului arbore este suma frecvențelor celor doi arbori care au fuzionat.
- Prin fuzionarea a doi arbori A_1 și A_2 se obține un nou arbore binar, în care arborii A_1 și A_2 sunt fii stâng respectiv drept al rădăcinii.
- În arborele final frunzele sunt caracterele și interpretăm codul binar pentru un caracter ca fiind drumul de la rădacină până la frunza corespunzătoare, unde o deplasare pe fiul stâng este reprezentată prin 0, iar o deplasare pe fiul drept prin 1.

Proiectarea algoritmilor – greedy (7)

Construcția unui cod Huffman

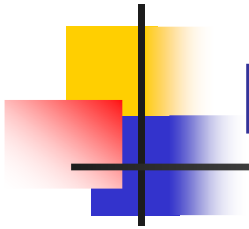




Proiectarea algoritmilor – greedy (8)

Algoritmul lui Huffman

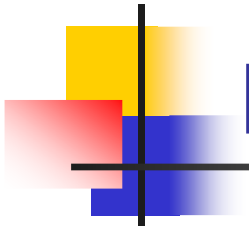
```
HUFFMAN(C)
    n ← |C|
    Q ← C
    for i ← 1 to n-1 do
        z ← CREATENODE()
        x ← z.st ← MIN(Q)
        Q ← Q-{x}
        y ← z.dr ← MIN(Q)
        Q ← Q-{y}
        z.cheie ← x.cheie + y.cheie
        Q.INSERT(z)
    return MIN(Q)
```



Proiectarea algoritmilor – greedy (9)

Complexitatea algoritmului Huffman

- Presupunem implementarea structurii Q sub forma unui heap binar.
- Astfel, construirea lui Q prin reconstituire heap, poate fi realizată în O(n).
- Bucla for are n-1 iterații în care apelează operații heap de complexitate O(lg n).
- Rezultă un timp total de execuție, pe o mulțime de n caractere, de O(n lg n).



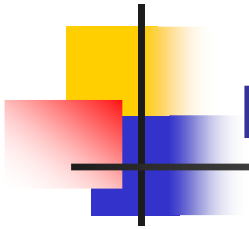
Proiectarea algoritmilor – greedy (10)

Aplicații

1. Să se ilustreze construcția arborelui Huffman pentru următoarele caractere și frecvențe:
   C={p:100, q:17, r:2, x:58, y:80, z:5}

   Decodificați apoi secvența binară 1111011001.
2. Stabiliți o codificare Huffman pentru următoarea secvență de frecvențe formată din primele numere ale șirului Fibonacci:

   C={a:1, b:1, c:2, d:3, e:5, f:8, g:13, h:21}
3. Desenați arborele Huffman pentru cuvântul ABRACADABRA.
4. Care din algoritmii prezentați anterior au la bază strategia greedy?
5. Implementați în Java algoritmul Huffman.
6. Implementați în Java problema fracționară a rucsacului.



Proiectarea algoritmilor – programare dinamică (1)

- Conceptul de programare dinamică a fost introdus în 1957 de matematicianul american Richard Bellman.
- Programarea dinamică, asemenea metodei divide et impera, rezolvă probleme combinând soluțiile unor subprobleme.
- Programarea dinamică este aplicabilă atunci când subproblemele au în comun sub-subprobleme.
- Spre deosebire de metoda divide et impera, un algoritm bazat pe programare dinamică rezolvă fiecare sub-subproblemă o singură dată, memorează soluția, evitând astfel recalcularea.

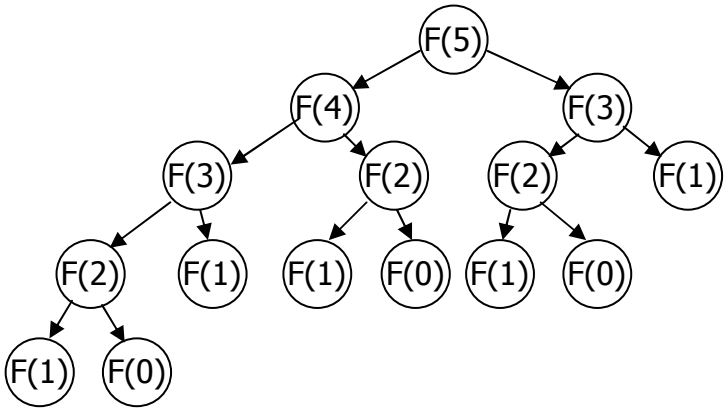
Proiectarea algoritmilor – programare dinamică (2)

Un prim exemplu: șirul lui Fibonacci

- Șirul lui Fibonacci este definit astfel:

$$F(n) = \begin{cases} 0, & n = 0 \\ 1, & n = 1 \\ F(n-1) + F(n-2), & n > 1 \end{cases}$$

- Arborele recursiv pentru calculul F(5) este următorul:



- În cazul implementării recursive, termenii sunt determinați de mai multe ori: F(2) de trei ori și F(3) de două ori, pentru calculul F(5). Astfel, varianta recursivă are complexitate exponențială: $O(2^n)$.
- Varianta iterativă memorează tot timpul rezultatele, deci are la bază programarea dinamică. Calculând o singură dată termenii, complexitatea este liniară: O(n).

Proiectarea algoritmilor – programare dinamică (3)

- Pseudocodul algoritmului iterativ:

```
FIBONACCI(n)
    if n≤1 then
        return n
    a ← 0
    b ← 1
    for i ← 2 to n do
        f ← a + b
        a ← b
        b ← f
    return f
```

- Există o variantă de calcul și mai rapidă, care calculează direct un anumit termen al șirului lui Fibonacci (deci cu o complexitate constantă O(1)), introdusă de Binet în 1843:

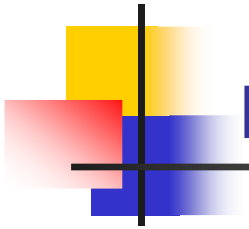
$$F_n = \frac{\Phi^n - (-\Phi)^{-n}}{\sqrt{5}}$$

unde Φ este secțiunea de aur, având valoarea:

$$\Phi = \frac{1+\sqrt{5}}{2}$$

Cel mai lung subșir crescător

- Fie A un șir de *n* numere întregi. Se cere determinarea subșirului crescător de lungime maximă, nu neapărat contiguu, în șirul A.
- Pentru rezolvarea problemei construim vectorii:
  - L[i] = lungimea subșirului crescător care începe pe poziția i și are ca prim element A[i].
  - S[i] = k, succesorul elementului A[i] în subșirul crescător, cu L[k]=max{L[j]| j>i și A[j]>A[i]}, iar S[i]=-1 dacă nu există un astfel de succesor.
- Cel mai lung subșir crescător va începe pe acea poziție i în A pentru care L[i] este maxim, iar afișarea se poate face cu ajutorul vectorului S.



Proiectarea algoritmilor – programare dinamică (5)

Algoritmul de determinare a subșirului crescător maximal

```
CMLSC(A)
    imax ← n
    for i ← n downto 1 do
        S[i] ← -1
        L[i] ← 1
        for j ← i+1 to n do
            if A[j]>A[i] and L[j]+1>L[i] then
                L[i] ← L[j]+1
                S[i] ← j
        if L[i]>L[imax] then
            imax ← i
    i ← imax
    while i≠-1 do
        PRINT(A[i])
        i ← S[i]
```

Exemplu:

- Fie șirul de întregi:
  A=3, 5, 76, 1, 45, 2, 68, 52, 90, 0, 4, 15
- Aplicând algoritmul CMLSC, obținem

	1	2	3	4	5	6	7	8	9	10	11	12
A	3	5	76	1	45	2	68	52	90	0	4	15
L	**5**	4	2	4	3	3	2	2	1	3	2	1
S	2	5	9	5	7	7	9	9	-1	11	12	-1

- Astfel, cel mai lung subșir crescător este:
  3, 5, 45, 68, 90

Complexitatea algoritmului CMLSC

- Operațiile din bucla *for* exterioară, de cost c_1, sunt apelate de n ori.
- Numărul de apeluri ale operațiilor din bucla *for* interioară, de cost c_2, este

$$1+2+\ldots+n-1=\sum_{k=1}^{n-1}k=\frac{n(n-1)}{2}$$

- Operațiile din bucla *while*, de cost c_3, sunt executate de cel mult n ori.
- Astfel, complexitatea algoritmului este

$$c_1 n + c_2 \frac{n(n-1)}{2} + c_3 n = an^2 + bn + c = O(n^2)$$

Proiectarea algoritmilor – programare dinamică (8)

Aplicații propuse

1. Se consideră un triunghi de numere, reprezentat într-o matrice A, ca în exemplul de mai jos:

10						
82	**81**					
4	6	**10**				
2	14	**35**	7			
41	3	**52**	26	15		
32	90	11	**87**	56	23	
54	65	89	32	**71**	9	31

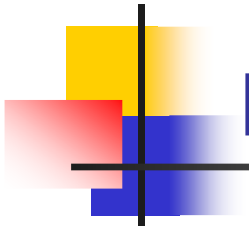
și o bilă care cade în jos sau spre dreapta-jos. Să se determine drumul prin care bila ar strânge numărul maxim de puncte. Se va folosi o matrice P de predecesori (-1=nu există, 1=pas în jos, 0=pas spre dreapta-jos) și o matrice S pentru sumele maxime.

2. Se dau pozițiile de start și de final respectiv dimensiunea tablei de șah. Se cere determinarea celui mai scurt drum al unui cal între cele două poziții. Se va folosi o matrice L care să păstreze lungimea minimă din fiecare poziție a tablei de șah și un tablou P de predecesori.

Proiectarea algoritmilor – backtracking (1)

- Backtracking este o metodă de căutare sistematică cu încercări repetate și reveniri în caz de nereușită.
- Soluția este construită în mod progresiv, prin adăugarea câte unei componente S_{k+1} la o soluție parțială $(S_1,S_2,...,S_k)$ care reprezintă o selecție din primele k oferte din totalul celor n, astfel încât $(S_1,S_2,...S_k,S_{k+1})$ formează o nouă soluție parțială.
- Soluția finală se obține în momentul în care selecția cuprinde toate cele n oferte, deci k=n.
- Metoda backtracking este în general ineficientă, având complexitate exponențială. De aceea, se folosește doar atunci când nu există metode mai eficiente de rezolvare a problemei.
- Forma generală a algoritmului backtracking recursiv:

```
BACKTRACKING(k)
    for i ← 1 to n do
        if ACCEPT(k, i) then
            S[k] ← i
            if FINAL(k) then
                return S
            else BACKTRACKING(k+1)
```



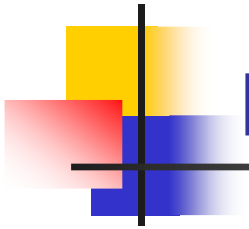
Proiectarea algoritmilor – backtracking (2)

Problema celor n regine

- Fie o tablă de șah de dimensiune n x n. Determinați toate posibilitățile de a amplasa n regine pe această tablă, astfel încât ele să nu se atace reciproc. Două regine se atacă reciproc dacă se află pe aceeași linie, coloană sau diagonală.
- Prima observație este că pe fiecare linie poate fi amplasată o singură regină. Astfel, pentru o tablă de șah de dimensiune n x n, problema celor n regine poate fi reprezentată printr-un tablou de n elemente, unde S[k]=i semnifică o regină amplasată pe poziția (k,i) a tablei de șah.
- Condiția ca două regine i și j să nu fie pe aceeași coloană este S[i]≠S[j].
- Condiția ca două regine i și j să nu fie pe aceeași diagonală este |j-i|≠|S[j]-S[i]|.
- Pentru n=2 și 3 nu există soluții, pentru n=4 sunt două soluții, etc.

Proiectarea algoritmilor – backtracking (3)

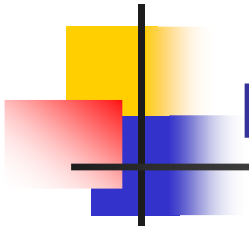
```
QUEENS(k)
    for i ← 1 to n do
        if ACCEPT(k, i) then
            S[k] ← i
            if k=n then
                PRINT(S)
            else QUEENS(k+1)


ACCEPT(k, i)
    for j ← 1 to k-1 do
        if S[j]=i then
            return FALSE
        if ABS(j-k)=ABS(S[j]-i) then
            return FALSE
    return TRUE
```



Proiectarea algoritmilor – backtracking (4)

Aplicații propuse

1. Să se implementeze în Java algoritmul prezentat pentru rezolvarea problemei celor n regine.
2. Fie o tablă de șah de dimensiune n x n. Determinați toate posibilitățile de a amplasa n ture pe această tablă, astfel încât ele să nu se atace reciproc. Două ture se atacă reciproc dacă se află pe aceeași linie sau coloană.
3. Fie o tablă de șah de dimensiune n x n și poziția de start a calului. Să se determine toate drumurile calului care să acopere toată tabla de șah și să treacă o singură dată prin toate pozițiile.



Proiectarea algoritmilor – algoritmi genetici (1)

- Un algoritm genetic este o procedură iterativă de căutare globală.
- Algoritmul procesează o populație de soluții potențiale (cromozomi) distribuite peste întreg spațiul de căutare.
- Pentru rezolvarea unei probleme folosind un algoritm genetic este necesară definirea unei funcții F care să evalueze performanța fiecărui cromozom.
- În fiecare iterație se creează o nouă populație P, numită generație. Toate generațiile au același număr de indivizi. În general, noua generație constă din indivizi mai performanți.
- Evoluția spre optimul global este asigurată prin recombinarea (încrucișarea) și modificarea (mutația) cromozomilor.
- Condiția de oprire se referă, de regulă, la atingerea unui anumit număr de generații ng.
- Forma generală a algoritmului genetic fundamental:

$P_1 \leftarrow$ RANDOM()
for t $\leftarrow$ 1 to ng do
    EVALUARE(P_t)
    P = SELECTIE(P_t)
    P' $\leftarrow$ RECOMBINARE(P)
    $P_{t+1} \leftarrow$ MODIFICARE(P')

Proiectarea algoritmilor – algoritmi genetici (2)

- Evaluarea populației de cromozomi

$$v_i = F(c_i), \quad i = \overline{1, n_c}$$

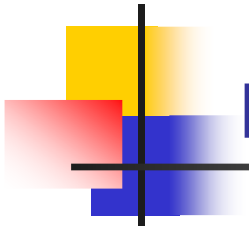
- Selecția
  - Se calculează suma valorilor obținute în urma evaluării

$$S = \sum_{i=1}^{n_c} v_i$$

  - Se determină probabilitățile cumulative de selecție

$$p_j = \sum_{i=1}^{j} \frac{v_i}{S}, \quad j = \overline{1, n_c}, \qquad p_{n_c} = 1$$

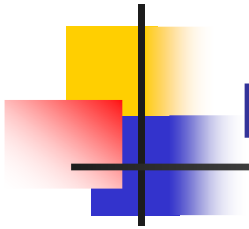
  - Pentru selecția noii populații se generează aleator n_c valori subunitare. Fiecare valoare r_i generată aleator va selecta acel cromozom c_k pentru care $r_i \in [p_k, p_{k+1}]$.
- Recombinarea a doi cromozomi constă în interschimbarea unor gene (biți).
- Modificarea unui cromozom constă în inversarea unor gene (biți).



Proiectarea algoritmilor – algoritmi genetici (3)

Determinarea maximului unei funcții

- Se va determina maximul unei funcții F(x), definită în exemplul nostru SIN(x). Maximul se caută în intervalul [li, ls], deci x∈[li, ls].
- Funcția X(v) normalizează valoarea v, aducând-o în intervalul [li, ls].
- Funcțiile GETBIT(v, i), SETBIT(v, i) și RESETBIT(v, i) preiau, setează respectiv resetează bitul de pe poziția i din v.
- Funcția INCRUCISARE(C, p1, p2) recombină biții cromozomilor p1 și p2.
- Funcția MUTATIE(C, p) modifică cromozomul p.
- Funcția INITIALIZARE inițializează populația de cromozomi cu valori generate aleator.
- Funcția SELECTIE evaluează populația de cromozomi și generează o populație nouă păstrând doar cromozomii performanți.
- Funcția RECOMBINARE realizează o selecție în vederea încrucișării.
- Funcția MODIFICARE realizează operația de mutație a populației de cromozomi.
- Funcția MAX reprezintă algoritmul genetic în sine, care determină maximul funcției F(x), cu x∈[li, ls]. Funcția folosește o populație C de nc cromozomi. Căutarea se termină după ng=50 de generații.



Proiectarea algoritmilor – algoritmi genetici (4)

```
li ← 0
ls ← 2
nc ← 500
ng ← 50
pincrucisare ← 0.3
pmutatie ← 0.1

X(v)
    return li+v/65535*(ls-li)

F(x)
    return SIN(x)

GETBIT(v, i)
    return v>>i&1

SETBIT(v, i)
    return v|(1<<i)

RESETBIT(v, i)
    return v&~(1<<i)
```

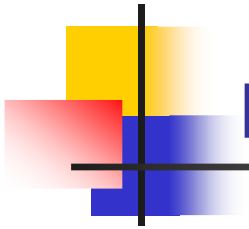
Proiectarea algoritmilor – algoritmi genetici (5)

```
INCRUCISARE(C, p1, p2)
    v1 ← C[p1]
    v2 ← C[p2]
    r ← RANDOM(0, 16)
    for j ← r to 16 do
        if GETBIT(v2, j)=1 then
            SETBIT(C[p1], j)
        else RESETBIT(C[p1], j)
        if GETBIT(v1, j)=1 then
            SETBIT(C[p2], j)
        else RESETBIT(C[p2], j)
```

```
MUTATIE(C, p)
    cp ← C[p]
    for j ← 1 to 16 do
        if pmutatie>RANDOM(0, 1) then
            if GETBIT(cp, j)=1 then
                RESETBIT(cp, j)
            else SETBIT(cp, j)
    if F(X(cp))>F(X(C[p])) then
        C[p] ← cp
```

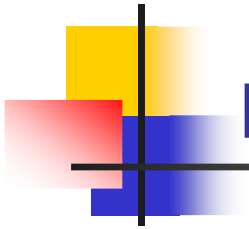
Proiectarea algoritmilor – algoritmi genetici (6)

```
INITIALIZARE(C)
    for i ← 1 to nc do
        C[i] ← RANDOM(0, 65536)
    xmax ← X(C[1])
    fmax ← F(xmax)


SELECTIE(C)
    s ← 0
    for i ← 1 to nc do
        V[i] ← F(X(C[i]))
        s ← s+V[i]
    P[1] ← V[1]/s
    for i ← 2 to nc do
        P[i] ← P[i-1]+V[i]/s
    for i ← 1 to nc do
        r ← RANDOM(0, 1)
        for j ← 1 to nc do
            if r>P[j] and r≤P[j+1] then
                C'[i] ← C[j]
    for i ← 1 to nc do
        C[i] ← C'[i]
```



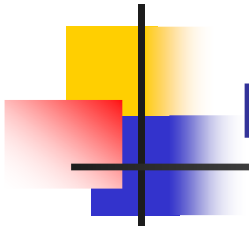
Proiectarea algoritmilor – algoritmi genetici (7)

```
RECOMBINARE(C)
    primul ← TRUE
    for i ← 1 to nc do
        if RANDOM(0, 1)<pincrucisare then
            if primul then
                primul ← FALSE
                p1 ← i
            else
                primul ← TRUE
                p2 ← i
                INCRUCISARE(C, p1, p2)


MODIFICARE(C)
    for i ← 1 to nc do
        MUTATIE(C, i)
```



Proiectarea algoritmilor – algoritmi genetici (8)

```
MAX(ng)
    INITIALIZARE(C)
    for t ← 1 to ng do
        SELECTIE(C)
        RECOMBINARE(C)
        MODIFICARE(C)
        maxiteri ← 1
        maxiterf ← F(X(C[1]))
        for i ← 2 to nc do
            if F(X(C[i]))>maxiterf then
                maxiteri ← i
                maxiterf ← F(X(C[i]))
        if maxiterf>fmax then
            fmax ← maxiterf
            xmax ← X(C[maxiteri])
    PRINT(xmax, fmax)
```



Proiectarea algoritmilor – algoritmi genetici (9)

Aplicații propuse

1. Să se implementeze în Java algoritmul genetic prezentat.
2. Să se modifice algoritmul genetic prezentat astfel încât să determine minimul unei funcții. Să se implementeze algoritmul modificat în Java.
3. Să se implementeze un algoritm genetic pentru rezolvarea problemei comis-voiajorului (*traveling salesman problem*). Se consideră n orașe și se cunosc distanțele dintre oricare două orașe. Un comis-voiajor trebuie să treacă prin toate cele n orașe. Se cere să se determine drumul minim care să pornească dintr-un oraș oarecare, să treacă o singură dată prin toate orașele celelalte și să revină în orașul inițial.

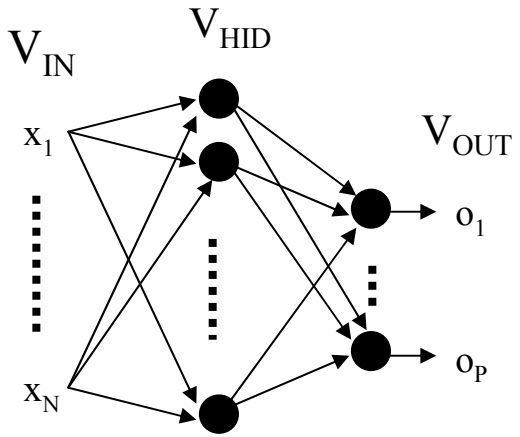
Proiectarea algoritmilor – rețele neuronale (1)

- Rețelele neuronale sunt folosite pentru rezolvarea unor probleme dificile, cum sunt cele de estimare, identificare și predicție.
- Există probleme practice de mare complexitate pentru care elaborarea unui algoritm este dificilă sau chiar imposibilă.
- Pornind de la o mulțime de exemple, rețelele neuronale sunt capabile să sintetizeze un model al problemei. Un mare avantaj al rețelelor neuronale constă în capacitatea lor de a învăța din exemple și apoi să trateze cazuri similare.
- Rețelele neuronale sunt formate dintr-o multitudine de neuroni, elemente simple de calcul care operează în paralel.
- Modelul de neuron a fost propus de McCulloch și Pitts în 1943. Hebb a introdus în 1949 un mecanism de învățare prin modificarea conexiunilor sinaptice dintre neuroni. Rosenblatt a propus apoi în 1957 primul model de rețea neuronală numită perceptron, care interconectează o mulțime de neuroni.

Proiectarea algoritmilor – rețele neuronale (2)

Perceptronul multistrat

- Structura perceptronului multistrat cu un singur strat ascuns este prezentată în figura următoare:



- Fiecare intrare x_i într-un neuron are asociată o pondere w_i. Ieșirea neuronului se obține prin însumarea ponderată a intrărilor,

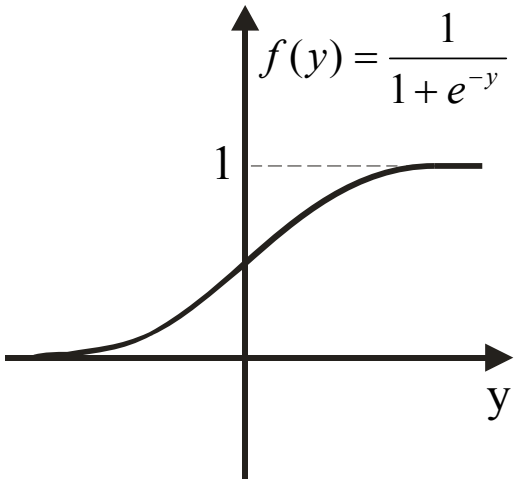
$$y = \sum_{i=1}^{n} w_i \cdot x_i$$

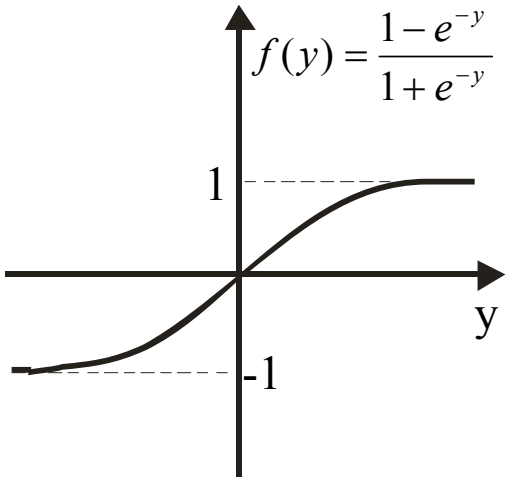
asupra căreia se aplică o funcție de activare.

Proiectarea algoritmilor – rețele neuronale (3)

Funcții de activare

- Funcția de activare are rolul de a restrânge domeniul de variație al ieșirii neuronului. Cele mai uzuale funcții de activare sunt următoarele:





- Vom folosi în continuare funcția de activare

$$f(y) = \frac{1-e^{-y}}{1+e^{-y}}$$

care restrânge ieșirea neuronului la intervalul (-1, 1).

Proiectarea algoritmilor – rețele neuronale (4)

Algoritmul de învățare backpropagation [Mit97]

- Algoritmul *backpropagation* ajustează ponderile rețelei neuronale astfel încât eroarea să scadă
- Algoritmul de învățare backpropagation, cu funcția de activare

$$f(y) = \frac{1 - e^{-y}}{1 + e^{-y}}$$

și intrări codificate cu -1 și 1, constă în următorii pași:

1. Se creează o rețea neuronală cu N intrări, M unități ascunse și P unități de ieșire.

2. Se inițializează ponderile

$$w_{ij}^{1};\ i = \overline{1,\ N};\ \ j = \overline{1,\ M}$$

$$w_{jk}^{2};\ \ j = \overline{1,\ M};\ \ k = \overline{1,\ P}$$

cu valori aleatoare mici [Mit97] din intervalul (-0.05, 0.05).

Proiectarea algoritmilor – rețele neuronale (5)

3. Cât timp $E(\overline{W}) = \frac{1}{2} \cdot \sum_{k=1}^{P} \left(t_k - o_k^2\right)^2 > T$ repetă

3.1. Se aplică rețelei intrarea $\overline{X^1}$ și se calculează ieșirea $\overline{O^2}$

$$net_j^1 = \sum_{i=1}^{N} w_{ij}^1 \cdot x_i^1, \quad j = 1,...,M$$

$$x_j^2 = o_j^1 = f(net_j^1), \quad j = 1,...,M$$

$$net_k^2 = \sum_{j=1}^{M} w_{jk}^2 \cdot x_j^2, \quad k = 1,...,P$$

$$o_k^2 = f(net_k^2), \quad k = 1,...,P$$

3.2. Pentru fiecare neuron din stratul de ieșire k, $k = \overline{1, P}$ se calculează eroarea δ_k^2 față de valoarea corectă t_k

$$\delta_k^2 = \left(t_k - o_k^2\right) \cdot f'(net_k^2) = \frac{1}{2} \cdot \left(t_k - o_k^2\right) \cdot \left(1 - o_k^2 \cdot o_k^2\right)$$

Proiectarea algoritmilor – rețele neuronale (6)

3.3. Pentru fiecare neuron din stratul ascuns j, $j = \overline{1, M}$ se calculează eroarea δ_j^1

$$\delta_j^1 = \sum_{k=1}^{P} \delta_k^2 \cdot w_{jk}^2 \cdot f'(net_j^1) = \frac{1}{2} \cdot \left(1 - o_j^1 \cdot o_j^1\right) \cdot \sum_{k=1}^{P} \delta_k^2 \cdot w_{jk}^2$$

3.4. Se actualizează toate ponderile rețelei neuronale

$$w_{jk}^2 = w_{jk}^2 + \alpha \cdot \delta_k^2 \cdot x_j^2, \quad j = \overline{1,\ M}, \quad k = \overline{1,\ P}$$

$$w_{ij}^1 = w_{ij}^1 + \alpha \cdot \delta_j^1 \cdot x_i^1, \quad i = \overline{1,\ N}, \quad j = \overline{1,\ M}$$

unde α este rata de învățare.

Proiectarea algoritmilor – rețele neuronale (7)

- În continuare este prezentat pseudocodul algoritmului backpropagation, considerând N intrări, M unități ascunse și P ieșiri.
- Am notat cu:
  - alpha - rata de învățare
  - input - vectorul de intrare (cu valorile codificate cu -1 și 1)
  - neth - vectorul cu valorile stratului ascuns înainte de activare
  - whin - matricea ponderilor dintre stratul ascuns și de intrare
  - bhin - vectorul bias aferent stratului ascuns
  - hidd - vectorul cu valorile stratului ascuns după activare
  - neto - vectorul cu valorile de ieșire înainte de activare
  - wohi - matricea ponderilor dintre stratul de ieșire și ascuns
  - bohi - vectorul bias aferent stratului de ieșire
  - out - vectorul cu valorile de ieșire după activare
  - deltaout - vectorul de erori aferent stratului de ieșire
  - deltain - vectorul de erori aferent stratului ascuns
  - target - vectorul cu ieșirile corecte (folosit în etapa de învățare).

Proiectarea algoritmilor – rețele neuronale (8)

```
alpha ← 0.3
input[N]
neth[M]
whin[M,N]
bhin[M]
hidd[M]
neto[P]
wohi[P,M]
bohi[P]
out[P]
deltaout[P]
deltain[M]

GENERARE_PONDERI()
    wi ← 4.0/N
    hwi ← 2.0/N
    for i ← 1 to M do
        bhin[i] ← (RANDOM(0, 10000) % FLOOR(wi*100))/100.0-hwi
        for j ← 1 to N do
            whin[i,j] ←(RANDOM(0, 10000) % FLOOR(wi*100))/100.0-hwi
    for i ← 1 to P do
        bohi[i] ← (RANDOM(0, 10000) % FLOOR(wi*100))/100.0-hwi
        for(j ← 1 to M do
            wohi[i,j] ←(RANDOM(0, 10000) % FLOOR(wi*100))/100.0-hwi

F(x)
    return (1-EXP(-1*x))/(1+EXP(-1*x))
```

Proiectarea algoritmilor – rețele neuronale (9)

```
dF(x)
    return (1-(F(x)*F(x)))/2

FORWARD(input[])
    for i ← 1 to M do
        neth[i] ← bhin[i]
        for j ← 1 to N do
            neth[i] ← neth[i] + whin[i,j]*input[j]
        hidd[i] ← F(neth[i])
    for i ← 1 to P do
        neto[i] ← bohi[i]
        for j ← 1 to M do
            neto[i] ← neo[i] + wohi[i,j]*hidd[j]
        out[i] ← F(neto[i])

BACKWARD(target[], input[])
    for i ← 1 to P do
        for j ← 1 to M do
            deltaout[i] ← (target[i]-out[i]) * dF(neto[i])
            wohi[i,j] ← wohi[i,j] + alpha*deltaout[i]*hidd[j]
            bohi[i] ← bohi[i] + alpha*deltaout[i]
    for i ← 1 to M do
        for j ← 1 to N do
            deltain[i] ← 0
            for k ← 1 to P do
                deltain[i]←deltain[i]+deltaout[k]*wohi[k,i]*dF(neth[i])
            whin[i,j] ← whin[i,j] + alpha*deltain[i]*input[j]
            bhin[i] ← bhin[i] + alpha*deltain[i]
```

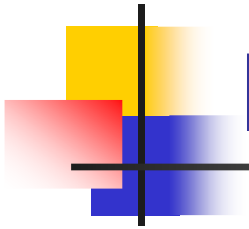
Aplicații propuse

1. Să se rescrie algoritmul *backpropagation* pentru funcţia de activare

$$f(y) = \frac{1}{1 + e^{-y}}$$

2. Să se implementeze în Java o rețea neuronală cu un strat ascuns care, pe baza algoritmului *backpropagation* prezentat, să învețe cifrele 0-9.

3. Să se implementeze în Java o reţea neuronală cu un strat ascuns care, folosind algoritmul *backpropagation*, să înveţe literele alfabetului limbii engleze.

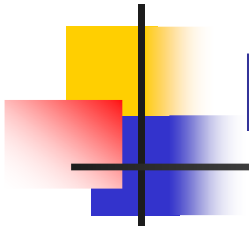


Nota finală

Nota laborator (NL) = 0,5*T + 0,5*C

Nota finală = 0,4*NL + 0,6*E

- T = Test susținut la laborator din aplicațiile propuse la laborator în cadrul capitolului *Limbajul Java*;
- C = Colocviu de laborator din aplicațiile propuse la laborator în cadrul capitolelor *Analiza algoritmilor* respectiv *Proiectarea algoritmilor*;
- E = Examen susținut din capitolele *Analiza algoritmilor* și *Proiectarea algoritmilor*.



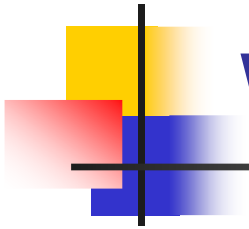
Bibliografie

Algoritmi

[Knu00] Knuth D., *Arta programării calculatoarelor*, Vol. 1 – *Algoritmi fundamentali*, Teora, 2000.
[Knu02] Knuth D., *Arta programării calculatoarelor*, Vol. 3 – *Sortare și căutare*, Teora, 2002.
[Cor00] Cormen T., Leiserson C., Rivest R., *Introducere în algoritmi*, Agora, 2000.
[Giu04] Giumale C., *Introducere în analiza algoritmilor*, Polirom, 2004.
[Log07] Logofătu D., *Algoritmi fundamentali în Java*, Polirom, 2007.
[Wai01] Waite M., Lafore R., *Structuri de date și algoritmi în Java*, Teora, 2001.
[Cri98] Cristea V., Athanasiu I., Kalisz E., Iorga V., *Tehnici de programare*, Teora 1998.
[Mit97] Mitchell T., *Machine Learning*, McGraw-Hill, 1997.

Programare în Java

[Blo17] Bloch J., Effective Java, Third Edition, Addison-Wesley Professional, 2017.
[Rob00] Roberts S., Heller P., Ernest M., *Complete Java 2 Certification*, Second Edition, SYBEX, USA, 2000.
[Cha01] Chan M., Griffith S., Iasi A., *Java 1001 secrete pentru programatori*, Teora, 2000.
[Tan07] Tanasă Ș., Andrei Ș., Olaru C., *Java de la 0 la expert*, Polirom, 2007.
[Hun01] Hunter J., Crawford W., *Java Servlet Programming*, Second Edition, O'Reilly, USA, 2001.
[Gea01] Geary D., *Advanced JavaServer Pages*, Prentice Hall, USA, 2001.
[Gor98] Gordon R., *Java Native Interface*, Prentice Hall, USA, 1998.



Webliografie

[Web01] http://java.sun.com/docs/codeconv/html/CodeConvTOC.doc.html
[Web02] http://www.javapassion.com/javaintro/
[Web03] http://thor.info.uaic.ro/~acf/java/curs/cursuri.html
[Web04] http://labs.cs.utt.ro/labs/sdaa/html/LabSDA.html
[Web05] http://www.personal.kent.edu/~rmuhamma/Algorithms/algorithm.html