

Univerzális programozás

Így neveld a programozód!

Ed. BHAX, DEBRECEN,
2019. február 19, v. 0.0.4

Copyright © 2019 Dr. Bátfai Norbert

Copyright (C) 2019, Norbert Bátfai Ph.D., batfai.norbert@inf.unideb.hu, nbatfai@gmail.com,

Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".

<https://www.gnu.org/licenses/fdl.html>

Engedélyt adunk Önnek a jelen dokumentum sokszorosítására, terjesztésére és/vagy módosítására a Free Software Foundation által kiadott GNU FDL 1.3-as, vagy bármely azt követő verziójának feltételei alapján. Nincs Nem Változtatható szakasz, nincs Címlapszöveg, nincs Hátlapszöveg.

<http://gnu.hu/fdl.html>

COLLABORATORS

	<i>TITLE :</i> Univerzális programozás		
<i>ACTION</i>	<i>NAME</i>	<i>DATE</i>	<i>SIGNATURE</i>
WRITTEN BY	Bátfai, Norbert, Bátfai, Mátyás, Bátfai, Nándor, Ács Bátfai, Margaréta	2019. szeptember 18.	

REVISION HISTORY

NUMBER	DATE	DESCRIPTION	NAME
0.0.1	2019-02-12	Az iniciális dokumentum szerkezetének kialakítása.	nbatfai
0.0.2	2019-02-14	Inciális feladatlisták összeállítása.	nbatfai
0.0.3	2019-02-16	Feladatlisták folytatása. Feltöltés a BHAX csatorna https://gitlab.com/nbatfai/bhax repójába.	nbatfai
0.0.4	2019-02-19	A Brun tételes feladat kidolgozása.	nbatfai

Ajánlás

„To me, you understand something only if you can program it. (You, not someone else!) Otherwise you don't really understand it, you only think you understand it.”

—Gregory Chaitin, *META MATH! The Quest for Omega*, [METAMATH]

Tartalomjegyzék

I. Bevezetés	1
1. Vízió	2
1.1. Mi a programozás?	2
1.2. Milyen doksikat olvassak el?	2
1.3. Milyen filmeket nézzek meg?	2
II. Tematikus feladatok	4
2. Helló, Turing!	6
2.1. Végtelen ciklus	6
2.2. Lefagyott, nem fagyott, akkor most mi van?	7
2.3. Változók értékének felcserélése	9
2.4. Labdapattogás	10
2.5. Szóhossz és a Linus Torvalds féle BogomIPS	12
2.6. Helló, Google!	14
2.7. 100 éves a Brun tétel	16
2.8. A Monty Hall probléma	17
3. Helló, Chomsky!	19
3.1. Decimálisból unárisba átváltó Turing gép	19
3.2. Az $a^n b^n c^n$ nyelv nem környezetfüggetlen	19
3.3. Hivatkozási nyelv	21
3.4. Saját lexikális elemző	22
3.5. l33t.1	23
3.6. A források olvasása	24
3.7. Logikus	26
3.8. Deklaráció	26

4. Helló, Caesar!	29
4.1. int *** háromszögmátrix	29
4.2. C EXOR titkosító	30
4.3. Java EXOR titkosító	31
4.4. C EXOR törő	33
4.5. Neurális OR, AND és EXOR kapu	35
4.6. Hiba-visszaterjesztéssel perceptron	41
5. Helló, Mandelbrot!	42
5.1. A Mandelbrot halmaz	42
5.2. A Mandelbrot halmaz a std::complex osztállyal	44
5.3. Biomorfok	46
5.4. A Mandelbrot halmaz CUDA megvalósítása	50
5.5. Mandelbrot nagyító és utazó C++ nyelven	54
5.6. Mandelbrot nagyító és utazó Java nyelven	54
6. Helló, Welch!	61
6.1. Első osztályom	61
6.2. LZW	61
6.3. Fabejárás	65
6.4. Tag a gyökér	68
6.5. Mutató a gyökér	72
6.6. Mozgató szemantika	73
7. Helló, Conway!	75
7.1. Hangyaszimulációk	75
7.2. Java életjáték	75
7.3. Qt C++ életjáték	83
7.4. BrainB Benchmark	90
8. Helló, Schwarzenegger!	91
8.1. Szoftmax Py MNIST	91
8.2. Szoftmax R MNIST	91
8.3. Mély MNIST	91
8.4. Deep dream	91
8.5. Minecraft-MALMÖ	92

9. Helló, Chaitin!	93
9.1. Iteratív és rekurzív faktoriális Lisp-ben	93
9.2. Weizenbaum Eliza programja	93
9.3. Gimp Scheme Script-fu: króm effekt	93
9.4. Gimp Scheme Script-fu: név mandala	93
9.5. Lambda	94
9.6. Omega	94
10. Helló, Gutenberg!	95
10.1. Juhász István, Magasszintű programozási nyelvek I.	95
10.2. Programozás bevezetés	102
10.3. Programozás	102
III. Második felvonás	105
11. Helló, Berners-Lee!	107
11.1. C++ és Java nyelvek összehasonlítása	107
11.2. Python	108
11.3.	108
11.4.	108
11.5.	108
12. Helló, Arroway!	110
12.1. OO szemlélet	110
12.2. Homokozó	110
12.3. "Gagyí"	110
12.4. Yoda	110
12.5. Kódolás from scratch	111
13. Helló, Liskov!	112
13.1. Liskov helyettesítés sértése	112
13.2. Szülő-gyerek	112
13.3. Anti OO	112
13.4. Hello, Android!	112
13.5. Ciklomatikus komplexitás	113

14. Helló, Mandelbrot!	114
14.1. Reverse engineering UML osztálydiagram	114
14.2. Forward engineering UML osztálydiagram	114
14.3. Egy esettan	114
14.4. BPMN	114
14.5. TeX UML	115
15. Helló, Chomsky!	116
15.1. Encoding	116
15.2. OOCWC lexer	116
15.3. Paszigráfia Rapszódia OpenGL full screen vizualizáció	116
15.4. Paszigráfia Rapszódia LuaLaTeX vizualizáció	117
15.5. Perceptron osztály	117
16. Helló, !	118
16.1.	118
16.2.	118
16.3.	118
16.4.	118
16.5.	119
17. Helló, !	120
17.1.	120
17.2.	120
17.3.	120
17.4.	120
17.5.	121
18. Helló, !	122
18.1.	122
18.2.	122
18.3.	122
18.4.	122
18.5.	123

19. Helló, !	124
19.1.	124
19.2.	124
19.3.	124
19.4.	124
19.5.	125
 IV. Irodalomjegyzék	 126
19.6. Általános	127
19.7. C	127
19.8. C++	127
19.9. Lisp	127

Ábrák jegyzéke

2.1. PageRank 4 honlapra felrajzolva	14
2.2. Táblázatban kiszámolva	15
2.3. A B_2 konstans közelítése	17
4.1. Lokális memória nélküli neuron (perceptron)	36
4.2. OR logikai kapu	37
4.3. OR-AND logikai kapu	38
4.4. EXOR logikai kapu rejtett neuron nélkül	39
4.5. EXOR logikai kapu rejtett neuronokkal	40
5.1. Mandelbrot kétdimenziós koordináta-rendszerben való ábrázolása	42
5.2. CUDA-s Mandelbrot halmaz váza, 600 x 600-as kép esetén:	50
5.3. CUDA-s kép kiszámítása ennyivel gyorsabb	53
5.4. Mandelbrot nagyítása JAVA környezetben	60
6.1. Binfa felépítése:	62
6.2. Emlékeztetőül az Inorder bejárásra:	65
6.3. Példa Postorder bejárásra:	67

Előszó

Amikor programozónak terveztem állni, ellenezték a környezetemben, mondván, hogy kell szövegszerkesztő meg táblázatkezelő, de az már van... nem lesz programozói munka.

Tévedtek. Hogy egy generáció múlva kell-e még tömegesen hús-vér programozó vagy olcsóbb lesz allokálni igény szerint pár robot programozót a felhőből? A programozók dolgozók lesznek vagy papok? Ki tudhatná ma.

Mindenesetre a programozás a teoretikus kultúra csúcsa. A GNU mozgalomban látom annak garanciáját, hogy ebben a szellemi kalandban a gyerekeim is részt vehessenek majd. Ezért programozunk.

Hogyan forgasd

A könyv célja egy stabil programozási szemlélet kialakítása az olvasóban. Módszere, hogy hetekre bontva ad egy tematikus feladatcsokrot. Minden feladathoz megadja a megoldás forráskódját és forrásokat feldolgozó videókat. Az olvasó feladata, hogy ezek tanulmányozása után maga adja meg a feladat megoldásának lényegi magyarázatát, avagy írja meg a könyvet.

Miért univerzális? Mert az olvasótól (kvázi az írótól) függ, hogy kinek szól a könyv. Alapértelmezésben gyerekeknek, mert velük készítem az iniciális változatot. Ám tervezem felhasználását az egyetemi programozás oktatásban is. Ahogy szélesedni tudna a felhasználók köre, akkor lehetne kiadása különböző korosztályú gyerekeknek, családoknak, szakköröknek, programozás kurzusoknak, felnőtt és továbbképzési műhelyeknek és sorolhatnánk...

Milyen nyelven nyomjuk?

C (mutatók), C++ (másoló és mozgató szemantika) és Java (lebutított C++) nyelvekből kell egy jó alap, ezt kell kiegészíteni pár R (vektoros szemlélet), Python (gépi tanulás bevezető), Lisp és Prolog (hogy lássuk mást is) példával.

Hogyan nyomjuk?

Ránts le a <https://gitlab.com/nbatfai/bhax> git repót, vagy méginkább forkolj belőle magadnak egy sajátot a GitLabon, ha már saját könyvön dolgozol!

Ha megvannak a könyv DocBook XML forrásai, akkor az alább látható **make** parancs ellenőrzi, hogy „jól formázottak” és „érvényesek-e” ezek az XML források, majd elkészíti a dlatex programmal a könyved pdf változatát, íme:

```
batfai@entropy:~$ cd glrepos/bhax/thematic_tutorials/bhax_textbook/
batfai@entropy:~/glrepos/bhax/thematic_tutorials/bhax_textbook$ make
rm -f bhax-textbook-fdl.pdf
xmllint --xinclude bhax-textbook-fdl.xml --output output.xml
xmllint --relaxng http://docbook.org/xml/5.0/rng/docbookxi.rng output.xml  ←
--noout
output.xml validates
rm -f output.xml
dlatex bhax-textbook-fdl.xml -p bhax-textbook.xls
Build the book set list...
Build the listings...
XSLT stylesheets DocBook - LaTeX 2e (0.3.10)
=====
Stripping NS from DocBook 5/NG document.
Processing stripped document.
Image 'dlatex' not found
Build bhax-textbook-fdl.pdf
'bhax-textbook-fdl.pdf' successfully built
```

Ha minden igaz, akkor most éppen ezt a legenerált `bhax-textbook-fdl.pdf` fájlt olvasod.



A DocBook XML 5.1 új neked?

Ez esetben forgasd a <https://tdg.docbook.org/tdg/5.1/> könyvet, a végén találsz az informatikai szövegek jelölésére használható gazdag „API” elemenkénti bemutatását.

I. rész

Bevezetés

1. fejezet

Vízió

1.1. Mi a programozás?

Ne cifrázzuk: programok írása. Mik akkor a programok? Mit jelent az írásuk?

1.2. Milyen doksikat olvassak el?

- Kezd ezzel: <http://esr.fsf.hu/hacker-howto.html>!
- Olvasgasd aztán a kézikönyv lapjait, kezd a **man man** parancs kiadásával. A C programozásban a 3-as szintű lapokat fogod nézegetni, például az első feladat kapcsán ezt a **man 3 sleep** lapot
- C kapcsán a [**KERNIGHANRITCHIE**] könyv adott részei.
- C++ kapcsán a [**BMECPP**] könyv adott részei.
- Az igazi kockák persze csemegéznek a C nyelvi szabvány **ISO/IEC 9899:2017** kódcsipeteiből is.
- Amiből viszont a legeslegjobban lehet tanulni, az a **The GNU C Reference Manual**, mert gcc specifikus és programozókra van hangolva: szinte csak 1-2 lényegi mondat és apró, lényegi kódcsipetek! Aki pdf-ben jobban szereti olvasni: <https://www.gnu.org/software/gnu-c-manual/gnu-c-manual.pdf>
- Az R kódok olvasása kis általános tapasztalat után automatikusan, erőfeszítés nélkül menni fog. A Python nincs ennyire a spektrum magától értetődő végén, ezért ahhoz olvasd el a [**BMECPP**] könyv - 20 oldalas gyorstalpaló részét.

1.3. Milyen filmeket nézzek meg?

- 21 - Las Vegas ostroma, <https://www.imdb.com/title/tt0478087/>, benne a **Monty Hall probléma** bemutatása.
- Kódjátzsma, <https://www.imdb.com/title/tt2084970>, benne a **kódtörő feladat** élménye.

- , , benne a bemutatása.
- , , benne a bemutatása.
- , , benne a bemutatása.
- , , benne a bemutatása.
- , , benne a bemutatása.
- , , benne a bemutatása.

DRAFT

II. rész

Tematikus feladatok

**Bátf41 Haxor Stream**

A feladatokkal kapcsolatos élő adásokat sugároz a <https://www.twitch.tv/nbatfai> csatorna, melynek permanens archívuma a <https://www.youtube.com/c/nbatfai> csatornán található.

DRAFT

2. fejezet

Helló, Turing!

2.1. Végtelen ciklus

Írj olyan C végtelen ciklusokat, amelyek 0 illetve 100 százalékban dolgoztatnak egy magot és egy olyat, amely 100 százalékban minden magot!

A feladatok közül a legegyszerűbb, amikor 100 százalékban dolgoztatunk egy magot. A kivitelezéséhez nem kell más, mint egy végtelen ciklus. Ezt kétféle képpen is megcsinálom most, mégpedig for és while ciklusokkal is. Miközben a ciklusok futnak, közben egy mag próbálja befejezni a futásukat, így az adott mag pörög ahogy csak bír.

For ciklussal:

```
int main()
{
    for(;;);
}
```

While ciklussal:

```
int main()
{
    while(1);
}
```

Következő, hogy 0 százalékban dolgoztassuk a magot. Egy egyszerű sleep function-nal lehet megoldani ezt a feladatot.

```
#include <unistd.h>

int main()
{
    for (;;)
        sleep(1);

    return 0;
}
```

Mikor minden magot akarunk egyidőben 100 százalékosan futtatni, akkor a következő megoldás az egyik lehetséges megoldás. Egy végtelen ciklust ráeresztünk minden magra. A `#pragma omp parallel` segítségével eresztjük rá az összes szálra. Ha csak adott mennyiségű szálon akarjuk futtatni egyszerre, akkor `#pragma omp parallel num_threads(adott_mennyiség)` beírásával tudjuk kivitelezni. A program fordításakor egy `-fopenmp` kapcsolót is bele kell írni.

```
#include <unistd.h>
#include <omp.h>

int main()
{
    #pragma omp parallel

    for(;;);

    return 0;
}
```

2.2. Lefagyott, nem fagyott, akkor most mi van?

Mutasd meg, hogy nem lehet olyan programot írni, amely bármely más programról eldönti, hogy le fog-e fagyni vagy sem!

Megoldás videó:

Megoldás forrása: tegyük fel, hogy akkora haxorok vagyunk, hogy meg tudjuk írni a `Lefagy` függvényt, amely tetszőleges programról el tudja dönteni, hogy van-e benne végtelen ciklus:

```
Program T100
{
    boolean Lefagy(Program P)
    {
        if(P-ben van végtelen ciklus)
            return true;
        else
            return false;
    }

    main(Input Q)
    {
        Lefagy(Q)
    }
}
```

A program futtatása, például akár az előző v. c ilyen pszeudókódjára:

```
T100(t.c.pseudo)
true
```

akár önmagára

```
T100 (T100)
false
```

ezt a kimenetet adja.

A T100-as programot felhasználva készítsük most el az alábbi T1000-set, amelyben a Lefagy-ra építő Lefagy2 már nem tartalmaz feltételezett, csak konkrét kódot:

```
Program T1000
{
    boolean Lefagy(Program P)
    {
        if(P-ben van végtelen ciklus)
            return true;
        else
            return false;
    }

    boolean Lefagy2(Program P)
    {
        if(Lefagy(P))
            return true;
        else
            for(;;);
    }

    main(Input Q)
    {
        Lefagy2(Q)
    }
}
```

Mit for kiírni erre a T1000 (T1000) futtatásra?

- Ha T1000 lefagyó, akkor nem fog lefagyni, kiírja, hogy true
- Ha T1000 nem fagyó, akkor pedig le fog fagyni...

akkor most hogy fog működni? Sehogy, mert ilyen Lefagy függvényt, azaz a T100 program nem is létezik.

Tanulságok, tapasztalatok, magyarázat...

2.3. Változók értékének felcserélése

Írj olyan C programot, amely felcseréli két változó értékét, bármiféle logikai utasítás vagy kifejezés használata nélkül!

Megoldás videó: https://bhaxor.blog.hu/2018/08/28/10_begin_goto_20_avagy_elindulunk

Változók felcserélése segédváltozó felhasználása és logikai utasítás nélkül kétféleképp lehetséges C nyelven.

Egyik megoldás, hogy a "kizáró vagy", EXOR vagy XOR művelet segítségével cseréljük meg az adott változókat. Fontos megjegyezni, hogy ez csak számoknál fog működni, karaktereknél és stringeknél nem.

```
#include <stdio.h>

int main()
{
    int a = 2;
    int b = 9;
    printf("a = %d\nb = %d\n", a , b);

    a = a ^ b;
    b = a ^ b;
    a = a ^ b;
    printf("Csere:\na = %d\nb = %d\n", a , b);

    return 0;
}
```

A megadott értékeket a \wedge operátor bitenként fogja összeadni. Azaz a számoknak a kettes számrendszerbeli értékeikkel fogja a műveletet végrehajtani. Különböző bitértékek esetén 1-et, azonos értékek esetén 0-t ad vissza.

```
-----|a = 0 0 1 0| /*eredeti a*/
-----|b = 1 0 0 1| /*eredeti b*/

a = a ^ b ---->>|a = 1 0 1 1| /*össze exoroztuk az a-t és b-t*/
b = a ^ b ---->>|b = 0 0 1 0| /*megcserélt b*/
a = a ^ b ---->>|a = 1 0 0 1| /*megcserélt a*/
```

A második módszer a C++ beépített swap függvényének segítségével hívása

```
#include <iostream>

using namespace std;

int main()
{
    int a = 2;
    int b = 9;
    cout << "a = " << a << endl;
    cout << "b = " << b << endl;
```

```
swap( a,b );

cout << "a = " << a << endl;
cout << "b = " << b << endl;

return 0;
}
```

2.4. Labdapattogás

Először if-ekkel, majd bármiféle logikai utasítás vagy kifejezés használata nélkül írj egy olyan programot, ami egy labdát pattogtat a karakteres konzolon! (Hogy mit értek pattogtatás alatt, alább láthatod a videókon.)

Megoldás videó: <https://bhaxor.blog.hu/2018/08/28/labdapattogas>

Megoldás if-et használva:

```
#include <stdio.h>
#include <curses.h>
#include <unistd.h>

int
main ( void )
{
    WINDOW *ablak; /*mutató az aktuális konzolablakra*/
    ablak = initscr (); /*amíg a program fut, ezt az ablakot használja*/

    int x = 0;
    int y = 0; /*labda kezdőpozíciók*/

    int xnov = 1;
    int ynov = 1; /*a léptetés mértéke*/

    int mx;
    int my; /*az ablak mérete*/

    for (;;)
    {
        getmaxyx ( ablak, my , mx ); /*az ablak aktuális méretét hívja be*/

        mvprintw ( y, x, "O" ); /*a labda kirajzoltatása*/

        refresh (); /*valós időben frissítjük az ablakot*/
        usleep ( 100000 ); /*mikrosec-ben mérve, altatjuk a programot*/
        /*a labda sebességét is ez határozza meg*/
        clear (); /*csak az aktuális labda jelenjen meg*/
    }
}
```

```
x = x + xnov;
y = y + ynov; /*mozgatjuk a labdát a koordináták növelésével*/

if ( x>=mx-1 ) /*ha elértük a konzol jobb oldalát, visszafordulunk ↵
*/
{
    xnov = xnov * -1;
}
if ( x<=0 ) /*ha elértük a konzol bal oldalát, visszafordulunk*/
{
    xnov = xnov * -1;
}
if ( y<=0 ) /*ha elértük a konzol tetejét, visszafordulunk*/
{
    ynov = ynov * -1;
}
if ( y>=my-1 ) /*ha elértük a konzol alját, visszafordulunk*/
{
    ynov = ynov * -1;
}
}
return 0;
}
```

Megoldás if nélkül:

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>

static void gotoxy(int x, int y)                /*kurzor pozicionálása*/
{
    int i;
    for(i=0; i<y; i++) printf("\n");           /*lefelé tolás*/
    for(i=0; i<x; i++) printf(" ");           /*jobbra tolás*/
    printf("o\n");                             /*labda ikonja*/
}

void usleep(int);
int main(void)
{
    int egyx=1;
    int egyy=-1;
    int i;
    int x=10;                                   /*a labda kezdeti pozíciója*/
    int y=20;
    int ty[23]; //magasság                    /*a pálya mérete*/
    int tx[80]; //szélesség
```

```
/*pálya széleinek meghatározása*/

for(i=0; i<23; i++)
    ty[i]=1;

ty[1]=-1;
ty[22]=-1;

for(i=0; i<79; i++)
    tx[i]=1;

tx[1]=-1;
tx[79]=-1;


for(;;)
{
    (void) gotoxy(x,y);
    /*printf("o\n"); Áthelyezve a gotoxy függvényve*/

    x+=egy x;
    y+=egy y;

    egy x*=tx[x];
    egy y*=ty[y];

    usleep (100000);
    (void) system("clear");
}
}
```

2.5. Szóhossz és a Linus Torvalds féle BogoMIPS

Írj egy programot, ami megnézi, hogy hány bites a szó a gépeden, azaz mekkora az int mérete. Használd ugyanazt a while ciklus fejet, amit Linus Torvalds a BogoMIPS rutinjában!

Tisztázzuk először, hogy mi is az a BogoMIPS?

A MIPS jelentés "Millions of Instructions Per Second", azaz feladatok milliói, amik egy másodperc alatt vannak végrehajtva. A "bogo" szó pedig arra utal, hogy ez nem egy valódi dolog, sokkal inkább egy kitaláció. Magyarul sajnos elég rosszul hangzik. Az ötlet Linus Torvalds fejéből pattant ki, mert kellett neki egy időzített loop, ami a processzor sebességét nézi meg bootolásnál. Először az 1.0-ás Linux kernelben jelent meg.

```
static void calibrate_delay(void)
{
```



```
int ticks;

printk("Calibrating delay loop.. ");
while (loops_per_sec <= 1) {
    ticks = jiffies;
    __delay(loops_per_sec);
    ticks = jiffies - ticks;
    if (ticks >= HZ) {
        __asm__("mull %1 ; divl %2"
            : "=a" (loops_per_sec)
            : "d" (HZ),
              "r" (ticks),
              "0" (loops_per_sec)
            : "dx");
        printk("ok - %lu.%02lu BogoMips\n",
            loops_per_sec/500000,
            (loops_per_sec/5000) % 100);
        return;
    }
}
printk("failed\n");
}
```

Most pedig nézzük meg, hogy milyen hosszú lehet egy gépi szó.

```
#include <stdio.h>

int main(void)
{
    int s=0;
    int c=1;
    while(c != 0)
    {
        c <= 1;
        ++s;
    }
    printf("Egy gépi szó maximális hossza %d karakter.\n", s );
}
```

Az `s` változóban fogjuk számolni a lépéseket 0-tól indulva. Tudjuk, hogy a számítógép kettes számrendszerben gondolkodik. Így képzeljük el, hogy `c` változóbeli értékünk kettes számrendszerben van ábrázolva. A kérdés az az, hogy egy `int` változó mekkora felületű, azaz hány bit értékű? Egy számot úgy tárol a gép, hogy a szám kettes számrendszerbeli értékét veszi és azt teszi a hely végére. Az elejét pedig feltölti megfelelő mennyiségű nullával. A program akkor fog jól működni, ha olyan számot adunk meg aminek a kettes számrendszer szerinti legkisebb helyiértékű száma 1-es. Tehát páratlan számokkal fog működni helyesen.

A `while` ciklusban addig megyünk, amíg el nem érjük a `c` változónk kettes számrendszerbeli utolsó 1-es karakterét. Ekkor minden bit 0 értéket vesz fel, tehát a változónk értéke is 0 lesz. Amíg ezt el nem érjük, a léptetés közben egyesével növeljük az `s` változónkat. Így a végén megkapjuk a maximális szóhosszt.

Kimenetünk a következő: Egy gépi szó maximális hossza 32 karakter. Tehát 32 hosszúságú egy int változó. Így legnagyobb számunk a $(2^{32})-1$ lehet.

Az előbb int változó hosszát néztük meg, most nézzük meg a long int változó hosszát

```
#include <stdio.h>

int main(void)
{
    int s=0;
    long int c=1;
    while(c != 0)
    {
        c <= 1;
        ++s;
    }
    printf("Egy gépi szó maximális hossza %d karakter.\n", s );
}
```

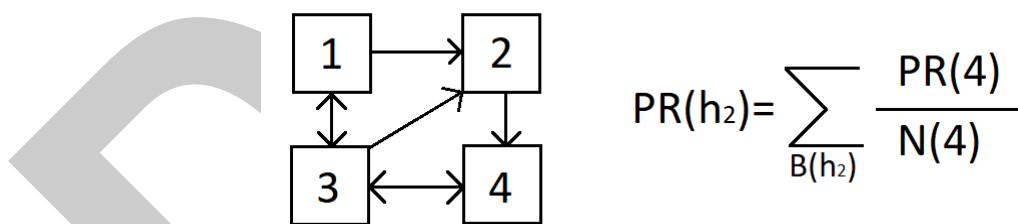
Kimenetünk a következő: Egy gépi szó maximális hossza 64 karakter. Tehát 64 hosszúságú egy long int változó, azaz kétszer hosszabb mint egy int. Így legnagyobb számunk a $(2^{64})-1$ lehet.

2.6. Helló, Google!

Írj olyan C programot, amely egy 4 honlapból álló hálózatra kiszámolja a négy lap Page-Rank értékét!

A Larry Page által kifejlesztett PageRank algoritmus nem csinál mást, mint megmondja nekünk, hogy egy honlap milyen minőségű, a rá mutató honlapok számának és minőségének függvényében.

Megoldás Móricka-rajz:



$$PR(h_2) = \sum_{B(h_2)} \frac{PR(4)}{N(4)}$$

2.1. ábra. PageRank 4 honlapra felrajzolva

A felvázolt állapot táblázatos formában az algoritmussal kiszámolva:

	A	i(1)	i(2)	PR	
1	1/4	1/12	1.5/12	1	
2	1/4	2.5/12	2/12	2	A: alaphelyzet
3	1/4	4.5/12	4.5/12	3	i: iterációk
4	1/4	4/12	4/12	4	PR: Page Rank

2.2. ábra. Táblázatban kiszámolva

Az alábbi C programkódot futtatva számolhatjuk ki az értékeket:

```
#include <stdio.h>
#include <math.h>
#include <stdlib.h>

void
kiir (double tomb[], int db)
{
    int i;
    for (i=0; i<db; i++)
        printf("PageRank [%d]: %lf\n", i, tomb[i]);
}

double tavolsag(double pagerank[], double pagerank_temp[], int db)
{
    double tav = 0.0;
    int i;
    for(i=0; i<db; i++)
        tav +=abs(pagerank[i] - pagerank_temp[i]);
    return tav;
}

int main(void)
{
    double L[4][4] = {
        {0.0, 0.0, 1.0 / 3.0, 0.0},
        {1.0 / 2.0, 0.0, 1.0 / 3.0, 0.0},
        {1.0 / 2.0, 0.0, 0.0, 1.0},
        {0.0, 1.0, 1.0 / 3.0, 0.0}
    };

    double PR[4] = {0.0, 0.0, 0.0, 0.0};
    double PRv[4] = {1.0 / 4.0, 1.0 / 4.0, 1.0 / 4.0, 1.0 / 4.0};

    long int i, j, h;
    i=0; j=0; h=5;

    for (;;)
    {
        for(i=0; i<4; i++)
```

```
    PR[i] = PRv[i];
    for (i=0; i<4; i++)
    {
        double temp=0;
        for (j=0; j<4; j++)
            temp+=L[i][j]*PR[j];
        PRv[i]=temp;
    }

    if ( tavolsag(PR,PRv, 4) < 0.00001)
        break;
}
kiir (PR,4);
return 0;
}
```

Forrás 1: [bhax/attention_raising/PageRank/pagerank.c](https://github.com/bhax/attention_raising/PageRank/pagerank.c)

2.7. 100 éves a Brun tétel

Írj R szimulációt a Brun tétel demonstrálására!

Megoldás videó: <https://youtu.be/xbYhp9G6VqQ>

Megoldás forrása: https://gitlab.com/nbatfai/bhax/blob/master/attention_raising/Primek_R

Mi is a Brun tétel?

A Brun tétel szerint ezeknek az ikerprímeknek a reciprokösszegük egy konstanshoz konvergálnak(közelítenek).

Prímszám az a természetes szám, amely csak önmagával és eggyel osztható. Eukleidész görög matematikus már Krisztus előtt tudta, hogy végtelen sok prímszám van, de ma sem tudja senki, hogy végtelen sok ikerprím van-e. Két prím ikerprím, ha különbségük 2.

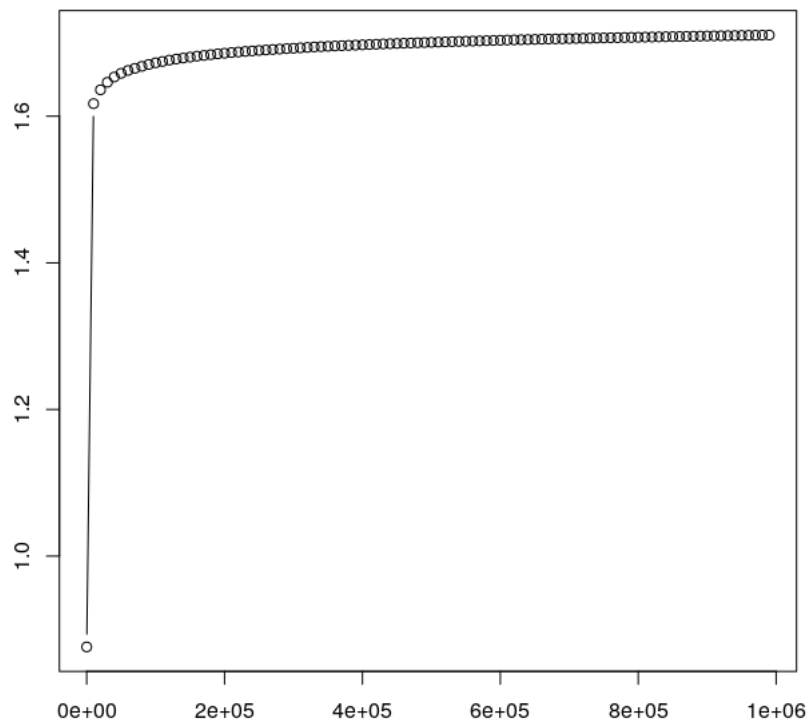
```
library(matlab)

stp <- function(x) {

    primes = primes(x)
    diff = primes[2:length(primes)]-primes[1:length(primes)-1]
    idx = which(diff==2)
    t1primes = primes[idx]
    t2primes = primes[idx]+2
    rt1plust2 = 1/t1primes+1/t2primes
    return(sum(rt1plust2))
}

x=seq(13, 1000000, by=10000)
y=sapply(x, FUN = stp)
plot(x,y,type="b")
```

A következő ábra azt mutatja, hogy a szumma értéke, hogyan nő, egy határértékhez tart, a B_2 Brun konstanshoz. Ezt ezzel a csipettel rajzoltuk ki, ahol először a fenti számítást 13-ig végezzük, majd 10013, majd 20013-ig, egészen 990013-ig, azaz közel 1 millióig. Vegyük észre, hogy az ábra első köre, a 13 értékhez tartozó 1.044023.



2.3. ábra. A B_2 konstans közelítése

2.8. A Monty Hall probléma

Írj R szimulációt a Monty Hall problémára!

Megoldás videó: https://bhaxor.blog.hu/2019/01/03/erdos_pal_mit_keresett_a_nagykonyvben_a_monty_hall-paradoxon_kapcsan

Megoldás forrása: https://gitlab.com/nbatfai/bhax/tree/master/attention_raising/MontyHall_R

Let's Make a Deal című showban a műsorvezető Monty Hall döntések elé állította a show résztvevőit. 3 ajtó közül kellett választaniuk egyet. Két ajtó mögött egy-egy kecske, míg az egyik mögött viszont a főnyeremény volt. A játékosnak egy ajtót kellett kiválasztania.

Miután a játékos választott egy ajtót a műsorvezető kinyitott egy ajtót, ami mögött kecske lapult és nem a játékos által választott ajtót nyitotta ki természetesen. Ekkor a játékos előtt két zárt ajtó maradt és ezek közül újra választhatott. Választhatta az elsőként választott ajtót, de választhatta amásik ajtót ami ezen kívül még zárva volt.

A Monty Hall paradoxon szerint ha ekkor (a második körben, amikor már csak két ajtó közül kellett választani) a játékos megváltoztatja a döntését és az eredetileg választott ajtó helyett a másikat nyitja ki, akkor az esélyei megduplázódnak.

A műsorvezetőről elnevezett paradoxon is ezen kérdések egyikén alapszik.

A paradoxon bizonyításához szükséges R kód a következő:

```
kiserletek_szama=10000000
kiserlet = sample(1:3, kiserletek_szama, replace=T)
jatekos = sample(1:3, kiserletek_szama, replace=T)
musorvezeto=vector(length = kiserletek_szama)

for (i in 1:kiserletek_szama) {

  if(kiserlet[i]==jatekos[i]){

    mibol=setdiff(c(1,2,3), kiserlet[i])

  }else{

    mibol=setdiff(c(1,2,3), c(kiserlet[i], jatekos[i]))

  }

  musorvezeto[i] = mibol[sample(1:length(mibol),1)]

}

nemvaltoztatesnyer= which(kiserlet==jatekos)
valtoztat=vector(length = kiserletek_szama)

for (i in 1:kiserletek_szama) {

  holvalt = setdiff(c(1,2,3), c(musorvezeto[i], jatekos[i]))
  valtoztat[i] = holvalt[sample(1:length(holvalt),1)]

}

valtoztatesnyer = which(kiserlet==valtoztat)

sprintf("Kiserletek szama: %i", kiserletek_szama)
length(nemvaltoztatesnyer)
length(valtoztatesnyer)
length(nemvaltoztatesnyer)/length(valtoztatesnyer)
length(nemvaltoztatesnyer)+length(valtoztatesnyer)
```

3. fejezet

Helló, Chomsky!

3.1. Decimálisból unárisba átváltó Turing gép

Állapotátmenet gráfiájával megadva írd meg ezt a gépet!

Az inputnál elkezdjük beolvasni a decimális értékeket. 0 és üres karakter esetén üreset írunk vissza és lépünk jobbra addig, amíg el nem érjük az első "valódi" karaktert. A-tól B-ig.

A beolvasott karaktereket ugyanúgy visszaírjuk, majd lépünk jobbra, addig ímg el nem érjük a karakterlánc végét (itt az "=") jelentő karaktert. B-től C-ig

Itt visszalépünk és eggyel csökkentjük az utolsó decimális karaktert (kivéve, ha 0-ra végződik, mert akkor az utolsó 2-t csökkentjük). majd az olvasott értéket visszaírjuk változatlanul és jobbra lépünk. C-ben jelezve

Jobbra léptetve olvasunk és írunk változatlanul, amíg üres helyhez nem érünk. C-ből D-be

Üres helyre 1-et helyettesítünk és balra lépünk. D-től E-ig

Változatlanul olvasunk-írunk balra haladva. Ismételjük az első üres helyig. visszaírjuk azt is, majd jobbra léptetünk. E-től F-ig

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

3.2. Az $a^n b^n c^n$ nyelv nem környezetfüggetlen

Mutass be legalább két környezetfüggő generatív grammatikát, amely ezt a nyelvet generálja!

A formális nyelvtan: A matematikában, informatikában és a nyelvészetben egy formális nyelv olyan szavakból, áll, melyeket egy abc-ből vett, jól definiált, bizonyos szabályoknak megfelelő betűk alkotnak. Egy formális nyelv abc-je szimbólumokból, betűkből és tokenekből állhat, melyek strinegekké állnak össze.

A formális nyelvtan 2 nagy csoportja, az analitikus és generatív nyelvtan.

Analiktikus nyelvtan esetében, a nyelvtani szabályokat alkalmazva "csupán" azt tudjuk eldönteni, hogy a kiértékelni kívánt karaktersorozat eleme-e a nyelvnek, vagy sem. (Igaz vagy Hamis értéket kapunk vissza)

A generatív nyelvtan, olyan nyelvészeti szabályok összessége, amely a nyelvtant olyan szabályrendszernek tekinti, amely pontosan azon a szavak kombinációit generálja, amelyek az adott nyelvben grammatikus mondatokat alkotnak. A kifejezést Noam Chomsky használta először az 50-es évek végén.

A Chomsky-féle nyelvi hierarchia a következőképpen alakul:

- 3. típusú nyelvek: reguláris
- 2. típusú nyelvek: környezetfüggetlen
- 1. típusú nyelvek: környezetfüggő
- 0. típusú nyelvek: rekurzívan felsorolható

1. Példa $L = \{ a^n b^n c^n \mid N > 0 \}$:

Szabályok:

```
S -> abc vagy aSQ
bQc -> bbcc
cQ -> cc
cc -> Qc
```

1. lépés. Állítsuk elő a megfelelő számú a-t:

```
S --> aSQ --> aaSQQ --> aaabcQQ -->
```

2. lépés. Alkalmazzuk az átalakítások szabályait:

```
--> aaabccQ --> aaabQcQ
aaabQcc --> aaabbccc --> aaabbQcc --> aaabbbccc
```

2. Példa $L = \{ a^n b^n c^n \mid N > 0 \}$:

- Minden "A"-t átalakítunk "a"-ra.
- Minden "B"-t átalakítunk "b"-re.
- Minden "C"-t átalakítunk "c"-re.

Szabályok:

```
S -> aSBC
S -> β (a béta karakter üres stringet fog visszaadni)
```

változók cseréje, ha szükségünk lesz rá:

```
CB -> HB
HB -> HC
HC -> BC
```

```
aB -> ab
bB -> bb
bC -> bc
cC -> cc
```


1. lépés. Állítsuk elő a megfelelő számú a-t, B-t és C-t. (még egyelőre nem lesznek jó sorrendben)

```
S-->aSBC-->aaSBCBC-->aaaSBCBCBC-->aaaßBCBCBC-->
```

2. lépés. Elérjük, hogy az összes B a C-k elé kerüljön.

```
aaaBCBCBC-->aaaBBBCCC-->
```

3. lépés. Végül átalakítjuk az összes változót konstanssá.

```
aaaBBBCCC-->aaabbbccc-->
```

Megoldás forrása: https://arato.inf.unideb.hu/batfai.norbert/UDPROG/deprecated/Prog1_1.pdf

3.3. Hivatkozási nyelv

A [KERNIGHANRITCHIE] könyv C referencia-kézikönyv/Utasítások melléklete alapján definiáld BNF-ben a C utasítás fogalmát! Majd mutass be olyan kódcsipeteket, amelyek adott szabvánnyal nem fordulnak (például C89), mással (például C99) igen.

Megoldás videó:

Megoldás forrása:

Rövid kódok, amik C89-es szabvánnyal nem fordulnak, de c99-el már igen.

C99-nél már lehetséges dupla '/' jel után kommentet írni, még c89-nél ez hibát jelent.

```
#include <stdio.h>

int main(void) //na itt a bibi
{
    printf("Hello World\n");
    return 0;
}
```

A másik ilyen hiba lehet, ha for ciklus utasításán belül próbálunk meg változót deklarálni c89 szabvánnyal:

```
#include <stdio.h>

int main()
{
    for (int i = 0; i < 5; ++i)
    {
        printf("%d\n", i );
    }
    return 0;
}
```

Viszont ez nem jelenthet hibát, ha c99 szabvánnyal fordítunk

Fordítás a következőképpen működik: `gcc programnev.c -o programnev` ehhez pedig hozzá fűzni még egy `-std=c89` vagy `-std=c99` kapcsolót.

3.4. Saját lexikális elemző

Írj olyan programot, ami számolja a bemenetén megjelenő valós számokat! Nem elfogadható olyan megoldás, amely maga olvassa betűnként a bemenetet, a feladat lényege, hogy lexert használjunk, azaz óriások vállán álljunk és ne kispályázzunk!

Lexikális analízisnek hívjuk azokat a folyamatokat, amikor az inputon lévő karaktersorozatokat tokenek sorozatává alakítjuk át. Lexernek hívjuk azokat a programokat, amik az előbb leírt lexikális analízist hajtják végre.

Az alább leírt L kódunkból az általunk használt lexer egy C kódot fog készíteni, amit lefordítunk és futtatunk.

A kód első részét a Lexer beleteszi az általa készített C kódba. Itt számoltatjuk meg a szövegben talált valós számokat. Ezen felül itt adjuk meg, hogy mit is keresünk pontosan. Esetünkben a 0 és 9 közötti számokat keressük.

```
%{
#include <stdio.h>
int valos_szamok = 0;
%}
szam [0-9]
```

A második rész a fordítási szabályokat tartalmazza. Az általunk keresett karakter lánc a szabályok szerint a következőképpen néz ki:

A karakterlánc elejét tetszőleges számú (lehet 0 darab is '*') egyjegyű szám képezi. Majd ezt követően a lánc második felén, egy minimum egy hosszúságú szám sorozat jöhet. A második fele vagy létezik vagy nem, ezt a ? operátorral fogalmazzuk meg és mindenféle képpen '.'-tal kell kezdődnie, ha létezik.

```
%%
{szam}* (\.{szam}+)? {++valos_szamok;
    printf("[valossz=%s %f]", yytext, atof(yytext));}
%%
```

A program utolsó részével hívjuk meg a lexikális elemzést. Miután ez lefut, kiíratjuk az inputon beérkezett valós számok darabszámát.

```
int main()
{
    yylex ();
    printf("Valós számok darabszáma: %d\n", valos_szamok);
    return 0;
}
```

Telepítsd a lexert a következőképpen: `sudo apt install flex`

Majd a következő parancsokkal fordítsd le, végül futtasd a kapott C fájlt:

```
$ lex -o realnumber.c realnumber.l
$ gcc real.c -o lex -lfl
$ ./lex
```

Megoldás videó: https://www.youtube.com/watch?v=9KnMqrkj_kU

Megoldás forrása: https://arato.inf.unideb.hu/batfai.norbert/UDPROG/deprecated/Prog1_1.pdf

3.5. l33t.l

Lexelj össze egy l33t ciphert!

A cipher algoritmusokat kódolásra és dekódolásra használják. Kettéoszthatjuk őket azerint, hogy a dekódolási-kódolási folyamat során ugyanazt a titkos kulcsot használja-e az algoritmus, vagy két különbözőt a két folyamathoz.

Az l33t röviden és tömören egy szleng. A neve a működéséből adódóan jöhetett létre. Lényege, hogy a szöveget alkotó karaktereket lecserélik, formailag hasonló, de kissé szokatlan karakterekre, ezzel minimálisan torzítva, esetleg rövidítve az eddig megszokott abc-t és nyelvtant.

Az előző feladathoz hasonlóan, most is 3 nagyobb részből épül fel a programunk.

Létrehozunk egy konstanst L337SIZE néven. Ez a szám az l337dlc7 tömb méretét osztva a cipher struktúra bájtban megadott méretével lesz egyenlő. Ezek után létrehozunk egy struktúra típust. Ebben fogjuk beolvasni a karaktereket. Továbbá itt fogjuk definiálni a beolvasott karakterekhez rendelhető karakterkészletet is. Egy karaktert egy stringgel fogunk behelyettesíteni.

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <ctype.h>

#define L337SIZE (sizeof l337dlc7 / sizeof (struct cipher))

struct cipher {
    char c;
    char *leet[4];
} l337dlc7 [] = {
/*itt lesz 36 sornyi definíció*/
};
%}
```

A 36 sornyi definíció alapjául a következő Wikipedia oldal szolgál: [Wikipedia leet karakterek helyettesítése](#)

A második részben, az L337SIZE tömb sorait járjuk végig, hogy megtaláljuk a behelyettesíteni kívánt karakter sorát. Ha megtaláltuk a kívánt karaktert, akkor egy véletlenszerű szám segítségével program eldönti, hogy a 4 lehetséges string helyett, melyiket helyettesítse be.

```
%%
. {

    int found = 0;
    for(int i=0; i<L337SIZE; ++i)
    {
```

```
        if(l337d1c7[i].c == tolower(*yytext))
        {

            int r = 1+(int) (100.0*rand()/(RAND_MAX+1.0));

            if(r<70)
                printf("%s", l337d1c7[i].leet[0]);
            else if(r<80)
                printf("%s", l337d1c7[i].leet[1]);
            else if(r<90)
                printf("%s", l337d1c7[i].leet[2]);
            else
                printf("%s", l337d1c7[i].leet[3]);

            found = 1;
            break;
        }

    }

    if(!found)
        printf("%c", *yytext);

}

%%
```

Az utolsó rész sorai a C forráskódot fogják alkotni. Itt inicializáljuk a véletlen-szám generátort, indítjuk a lexikális elemzést. A program a standard inputot fogja olvasni.

```
int
main()
{
    srand(time(NULL)+getpid());
    yylex();
    return 0;
}
```

A programot ugyan úgy fogjuk futtatni, mint az előző feladatban, azaz:

```
$ lex -o leet.c leet.l
$ gcc leet.c -o leet -lfl
$ ./leet
```

Megoldás videó: https://www.youtube.com/watch?v=06C_PqDpD_k

3.6. A források olvasása

Hogyan olvasod, hogyan értelmezed természetes nyelven az alábbi kódcsipeteket? Például

```
if(signal(SIGINT, jelkezelő)==SIG_IGN)
    signal(SIGINT, SIG_IGN);
```

Ha a SIGINT jel kezelése figyelmen kívül volt hagyva, akkor ezen túl is legyen figyelmen kívül hagyva, ha nem volt figyelmen kívül hagyva, akkor a jelkezelő függvény kezelje. (Miután a **man 7 signal** lapon megismertem a SIGINT jelet, a **man 2 signal** lapon pedig a használt rendszerhívást.)



Bugok

Vigyázz, sok csipet kerülendő, mert bugokat visz a kódba! Melyek ezek és miért? Ha nem megyránézésre, elkapja valamelyiket esetleg a splint vagy a frama?

i.

```
if(signal(SIGINT, SIG_IGN)!=SIG_IGN)
    signal(SIGINT, jelkezelő);
```

Ha a SIGINT jel kezelése nem lett figyelmen kívül hagyva, akkor a jelkezelő függvénynek kell kezelnie azt.

ii.

```
for(i=0; i<5; ++i)
```

Ez egy for ciklus, amit 0-ról indítunk és 5-ször fut le, eggyel növelve az i értékét. Az ++i egy preinkrementáló, mert i értékét még akkor növeli eggyel, mielőtt belépne a ciklusmagba.

iii.

```
for(i=0; i<5; i++)
```

Ebben a for ciklusban egy post inkrementációt használunk, ami annyit jelent, hogy az i változónk értéke a ciklusmag végén történik meg.

iv.

```
for(i=0; i<5; tomb[i] = i++)
```

Ebben a for ciklusban egy tomb elemeinek sorszámát növelnénk 1-el. Legalábbis talán ez lenne a célja a költőnek. Ha a ciklusmagban kiíratjuk mindig az i értékét, akkor láthatjuk hogy az első érték mindig valamilyen memóriaszemét lesz. Kerülendő implementáció.

v.

```
for(i=0; i<n && (*d++ = *s++); ++i)
```

Ebben a for ciklusban az i változónkat n-1-ig növeljük, de csak akkor, ha teljesül (*d++ = *s++) a feltétel is. A splinter kiabál, hogy a logikai operátor után nem boolean típusú érték jön.

vi.

```
printf("%d %d", f(a, ++a), f(++a, a));
```

Két decimális számot az f függvény segítségével ki akarunk írni. A bökkenő, hogy a függvény argumentumainak sorrendje nincs előre meghatározva.

vii.

```
printf("%d %d", f(a), a);
```

Két decimális számot íratunk ki. Az első az f függvény argumentuma lesz, az így kapott értéket kapjuk. A második pedig egyszerűen csak az a értékét kapjuk.

viii.

```
printf("%d %d", f(&a), a);
```

A kiíratásnál az első értékünk alakulása a következő: az f függvény megkapja az a változó címét, majd a függvény az erre a címre mutató pointer alatti értéket fogja vissza adni. Míg a második érték pedig az a lesz önmaga.

3.7. Logikus

Hogyan olvasod természetes nyelven az alábbi Ar nyelvű formulákat?

```
$(\texttt{\textbackslash forall } x \texttt{\textbackslash exists } y ((x < y) \texttt{\textbackslash wedge } (y \texttt{\textbackslash text } \{ \text{prím} \})))$
```

```
$(\texttt{\textbackslash forall } x \texttt{\textbackslash exists } y ((x < y) \texttt{\textbackslash wedge } (y \texttt{\textbackslash text } \{ \text{prím} \}) \texttt{\textbackslash wedge } (S y \texttt{\textbackslash text } \{ \text{prím} \}))) \leftrightarrow
```

```
)$
```

```
$(\texttt{\textbackslash exists } y \texttt{\textbackslash forall } x (x \texttt{\textbackslash text } \{ \text{prím} \}) \texttt{\textbackslash supset } (x < y))$
```

```
$(\texttt{\textbackslash exists } y \texttt{\textbackslash forall } x (y < x) \texttt{\textbackslash supset } \texttt{\textbackslash neg } (x \texttt{\textbackslash text } \{ \text{prím} \})))$
```

I. Végtelen sok prímszám van.

```
$(\texttt{\textbackslash forall } x \texttt{\textbackslash exists } y ((x < y) \texttt{\textbackslash wedge } (y \texttt{\textbackslash text } \{ \text{prím} \})))$
```

II. Végtelen sok iker-prímszám van.

```
$(\texttt{\textbackslash forall } x \texttt{\textbackslash exists } y ((x < y) \texttt{\textbackslash wedge } (y \texttt{\textbackslash text } \{ \text{prím} \}) \texttt{\textbackslash wedge } (S y \texttt{\textbackslash text } \{ \text{prím} \}))) \leftrightarrow
```

```
)$
```

III. Véges sok prímszám van.

```
$(\texttt{\textbackslash exists } y \texttt{\textbackslash forall } x (x \texttt{\textbackslash text } \{ \text{prím} \}) \texttt{\textbackslash supset } (x < y))$
```

IV. Véges sok prímszám van.

```
$(\texttt{\textbackslash exists } y \texttt{\textbackslash forall } x (y < x) \texttt{\textbackslash supset } \texttt{\textbackslash neg } (x \texttt{\textbackslash text } \{ \text{prím} \})))$
```

Megoldás forrása: https://gitlab.com/nbatfai/bhax/blob/master/attention_raising/MatLog_LaTeX

Megoldás videó: <https://youtu.be/ZexiPy3ZxsA>, https://youtu.be/AJSXOQFF_wk

3.8. Deklaráció

Vezesd be egy programba (forduljon le) a következőket:

- egész
- egészre mutató mutató

- egész referenciája
- egészek tömbje
- egészek tömbjének referenciája (nem az első elemé)
- egészre mutató mutatók tömbje
- egészre mutató mutatót visszaadó függvény
- egészre mutató mutatót visszaadó függvényre mutató mutató
- egészet visszaadó és két egészet kapó függvényre mutató mutatót visszaadó, egészet kapó függvény
- függvénymutató egy egészet visszaadó és két egészet kapó függvényre mutató mutatót visszaadó, egészet kapó függvényre

Mit vezetnek be a programba a következő nevek?

- ```
int a;
```

Egy int típusú a változót hozunk létre
- ```
int *b = &a;
```

Egy b mutató, ami a-nak a memóriacímét tartalmazza
- ```
int &r = a;
```

Egy referenciaváltozó, ami a-nak az értékét kapja meg
- ```
int c[5];
```

Egy 5 elemű, inteket tartalmazó tömb
- ```
int (&tr)[5] = c;
```

Itt a tr megkapja a c tömb elemeit
- ```
int *d[5];
```

EgészekEgy függvény, ami egészekre mutató mutatót ad visszat tároló, tömbre mutató tömb
- ```
int *h ();
```

Egy függvény, ami egészekre mutató mutatót ad vissza
- ```
int *(*l) ();
```

Egy függvény mutató, ami egészekre mutató mutatót ad vissza

- ```
int (*v (int c)) (int a, int b)
```

Egy egészszel visszatérő 2 egészet váró függvényre mutató mutatóval visszatérő 2 egészet váró függvény

- ```
int ((*z) (int)) (int, int);
```

Függvényt mutató egy egészet visszaadó és két egészet kapó függvényre mutató mutatót visszaadó, egészet kapó függvényre mutató mutató.

4. fejezet

Helló, Caesar!

4.1. int *** háromszögmátrix

Részletenként haladva járjuk be a programot

```
#include <stdio.h>
#include <stdlib.h>

int
main ()
{
    int sorok_szama = 5;
    double **tm;
```

Elsőként megadjuk, hogy hány soros lesz a mátrixunk. Másodszor pedig deklarálunk egy 8 bájt méretű double típusú pointert, amin keresztül hivatkozni fogunk a mátrix soraiban található elemekre.

```
    if((tm = (double **) malloc (sorok_szama * sizeof (double *))) == NULL)
    {
        return -1;
    }
```

A mátrix minden sorához lefoglalunk egy-egy mutatót a memóriában, a malloc függvénnyel. A függvény által visszaadott mutatót double típusúra kényszerítjük. Ha ez nem jön össze, akkor kilépünk a programból a visszakapott null pointer miatt.

```
    for (int i = 0; i < sorok_szama; ++i)
    {
        if((tm[i] = (double *) malloc ((i + 1) * sizeof (double))) == NULL)
        {
            return -1;
        }
    }
```

Majd a háromszög-mátrix struktúrájának alapján, minden sorban soronként egyel több helyet foglalunk le a memóriában. Itt már nem mutatók, hanem a mátrixunk értékei lesznek tárolva. Ha nem jön össze, itt is kilépünk a programból.

```
for (int i = 0; i < sorok_szama; ++i)
    for (int j = 0; j < i + 1; ++j)
        tm[i][j] = i * (i + 1) / 2 + j;

for (int i = 0; i < sorok_szama; ++i)
{
    for (int j = 0; j < i + 1; ++j)
        printf ("%f, ", tm[i][j]);
    printf ("\n");
}
```

Egyesével feltöltjük a mátrixunkat lebegőpontos számokkal. Soronként, azon belül oszloponként haladva.

```
for (int i = 0; i < sorok_szama; ++i)
    free (tm[i]);

free (tm);

return 0;
}
```

Legvégül pedig ismét végiglépkedünk a korábban lefoglalt memóriaterületeken és felszabadítunk mindent. Ezzel megakadályozva, hogy elfolyjon a memóriánk.

Megoldás videó 1: <https://www.youtube.com/watch?v=1MRTuKwRsB0>

Megoldás videó 2: <https://www.youtube.com/watch?v=RKbX5-EWpzA>

4.2. C EXOR titkosító

Írj egy EXOR titkosítót C-ben!

Lépésről-lépésre, kódcsipetről-kódcsipetre.

```
#include <stdio.h>
#include <unistd.h>
#include <string.h>

#define MAX_KULCS 100
#define BUFFER_MERET 256
```

Beállítjuk a kódoláshoz megadott kulcs és a buffer maximális méretét.

```
int
main (int argc, char **argv)
{
```

```
char kulcs[MAX_KULCS];
char buffer[BUFFER_MERET];
int kulcs_index = 0;
int olvasott_bajtok = 0;
int kulcs_meret = strlen (argv[1]);
strncpy (kulcs, argv[1], MAX_KULCS);
```

A main-be létrehozunk két char típusú tömböt, amibe a kulcs és a buffer értékeit tároljuk. Majd inicializálunk egy kulcs_index változót, amibe majd a kulcs aktuális elemét tároljuk, karakterről-karakterre léptetve a kódolás közben, míg az olvasott_bajtok változóba, a beolvasott bájtok számát fogjuk tárolni. Az strlen() függvénnyel a kulcs méretét kapjuk meg. Azután az strncpy()-val bemásoljuk az előbb megadott parancssori argumentumban tárolt sztringet a kulcs tömbünkbe. A függvény 3. paramétere megadjanekünk, hogy max. mennyi karaktert másolhatunk át.

```
while ((olvasott_bajtok = read (0, (void *) buffer, BUFFER_MERET))
{
    for (int i = 0; i < olvasott_bajtok; ++i)
    {
        buffer[i] = buffer[i] ^ kulcs[kulcs_index];

        kulcs_index = (kulcs_index + 1) % kulcs_meret;
    }
}
```

Addig olvasunk a bemenetről, amíg van mit és tároljuk a beolvasott bájtokat a bufferben. Ha már nincs mit olvasni, a read függvény 0-val tér vissza. Minden egyes beolvasott bájtot össze EXOR-ozunk a kulcs soron következő karakterével. Végül pedig kiíratjuk az eredményt.

```
write (1, buffer, olvasott_bajtok);

    }
}
```

Megoldás forrása: https://progpater.blog.hu/2011/02/15/felvetelt_hirdet_a_cia

4.3. Java EXOR titkosító

Írj egy EXOR titkosítót Java-ban!

A titkosításhoz használt algoritmust egy külön osztályban hoztuk létre.

```
public class exor {

    public exor(String keyString,
        java.io.InputStream inputStream,
```

```
        java.io.OutputStream outputStream)
        throws java.io.IOException {

    byte [] kulcs = keyString.getBytes();
    byte [] buffer = new byte[256];
    int keyIndex = 0;
    int readBytes = 0;

    while((readBytes =
        inputStream.read(buffer)) != -1) {

        for(int i=0; i<readBytes; ++i) {

            buffer[i] = (byte)(buffer[i] ^ kulcs[keyIndex]);
            keyIndex = (keyIndex+1) % kulcs.length;

        }

        outputStream.write(buffer, 0, readBytes);

    }

}

public static void main(String[] args) {

    try {

        new exor(args[0], System.in, System.out);

    } catch(java.io.IOException e) {

        e.printStackTrace();

    }

}

}
```

A paraméterként átadott kulcsszöveget tároljuk egy stringben, majd létrehozunk egy bejövő-kimenő csatornát. Beolvassuk a kulcsot egy byte típusú tömbbe. Inicializáljuk a kulcsindex és olvasott bájtok változókat. Majd egy while ciklus a bemenetet olvassa tömbönként buffer méret szerint. Ezután egy for ciklus segítségével bejárjuk az olvasott_bájtok változót és exorozzuk a buffer tartalmát a kulccsal. Majd kulcs_index-t növeljük a kulcs méret eléréséig. A main()-be használjuk a trycatch-et, ami a hibák elkapására használatos. A try tartalmazza az utasítást, ha valami nem stimmel hibát dob, a catch "elkapja" és kapunk egy hibaüzenetet.

Megoldás forrása: <https://www.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat/ch01.html>

4.4. C EXOR törő

Írj egy olyan C programot, amely megtöri az első feladatban előállított titkos szövegeket!

```
#define MAX_TITKOS 4096
#define OLVASAS_BUFFER 256
#define KULCS_MERET 8
#define _GNU_SOURCE

#include <stdio.h>
#include <unistd.h>
#include <string.h>

double
atlagos_szohossz (const char *titkos, int titkos_meret)
{
    int sz = 0;
    for (int i = 0; i < titkos_meret; ++i)
        if (titkos[i] == ' ')
            ++sz;

    return (double) titkos_meret / sz;
}
```

Kiszámítjuk az átlagos szóhosszt, mégpedig a következőképpen: a szöveg karakterszámát elosztjuk a szövegben található szóközök számával

```
int
tisztalehet (const char *titkos, int titkos_meret)
{
    double szohossz = atlagos_szohossz (titkos, titkos_meret);

    return szohossz > 6.0 && szohossz < 9.0
        && strcasestr (titkos, "hogy") && strcasestr (titkos, "nem")
        && strcasestr (titkos, "az") && strcasestr (titkos, "ha");
}
```

Az alap koncepció, hogy az eredeti szöveg biztosan tartalmazza a leggyakoribb magyar szavakat (hogy, nem, az, ha). Az átlagos szóhossz azért kell, hogy a törések számát csökkenthessük.

```
void
exor (const char kulcs[], int kulcs_meret, char titkos[], int titkos_meret)
{
    int kulcs_index = 0;

    for (int i = 0; i < titkos_meret; ++i)
```

```
{  
  
    titkos[i] = titkos[i] ^ kulcs[kulcs_index];  
    kulcs_index = (kulcs_index + 1) % kulcs_meret;  
  
}  
  
}  
  
int  
exor_tores (const char kulcs[], int kulcs_meret, char titkos[], int ←  
    titkos_meret)  
{  
  
    exor (kulcs, kulcs_meret, titkos, titkos_meret);  
  
    return tiszta_lehet (titkos, titkos_meret);  
  
}
```

Bájtónként végrehajtjuk az EXOR-t. A % segítségével a kulcs akkor is aktuális marad, ha a szöveg hosszabb mint a keresett kulcs.

```
int  
main (void)  
{  
  
    char kulcs[KULCS_MERET];  
    char titkos[MAX_TITKOS];  
    char *p = titkos;  
    int olvasott_bajtok;  
  
    while ((olvasott_bajtok =  
        read (0, (void *) p,  
            (p - titkos + OLVASAS_BUFFER <  
                MAX_TITKOS) ? OLVASAS_BUFFER : titkos + MAX_TITKOS - p)))  
        p += olvasott_bajtok;  
  
    // maradek hely nullazasa a titkos bufferben  
    for (int i = 0; i < MAX_TITKOS - (p - titkos); ++i)  
        titkos[p - titkos + i] = '\\0';  
  
    for (int ii = '0'; ii <= '9'; ++ii)  
        for (int ji = '0'; ji <= '9'; ++ji)  
            for (int ki = '0'; ki <= '9'; ++ki)  
                for (int li = '0'; li <= '9'; ++li)  
                    for (int mi = '0'; mi <= '9'; ++mi)  
                        for (int ni = '0'; ni <= '9'; ++ni)  
                            for (int oi = '0'; oi <= '9'; ++oi)  
                                for (int pi = '0'; pi <= '9'; ++pi)  
                                {
```

```
    kulcs[0] = ii;
    kulcs[1] = ji;
    kulcs[2] = ki;
    kulcs[3] = li;
    kulcs[4] = mi;
    kulcs[5] = ni;
    kulcs[6] = oi;
    kulcs[7] = pi;

    if (exor_tores (kulcs, KULCS_MERET, titkos, p - titkos))
        printf
        ("Kulcs: [%c%c%c%c%c%c%c%c]\nTiszta szoveg: [%s]\n",
         ii, ji, ki, li, mi, ni, oi, pi, titkos);

    exor (kulcs, KULCS_MERET, titkos, p - titkos);
}

return 0;
}
```

A while ciklus addig fog tartani, amíg van mit olvasni a bemenetről. Egymásba ágyazott for ciklusokkal előállítjuk az összes lehetséges kulcsot és kipróbáljuk mindet. A végén újra exor-ozunk, így nem lesz szükségünk egy második bufferre.

4.5. Neurális OR, AND és EXOR kapu

R

Mi is az a neurális hálózat?

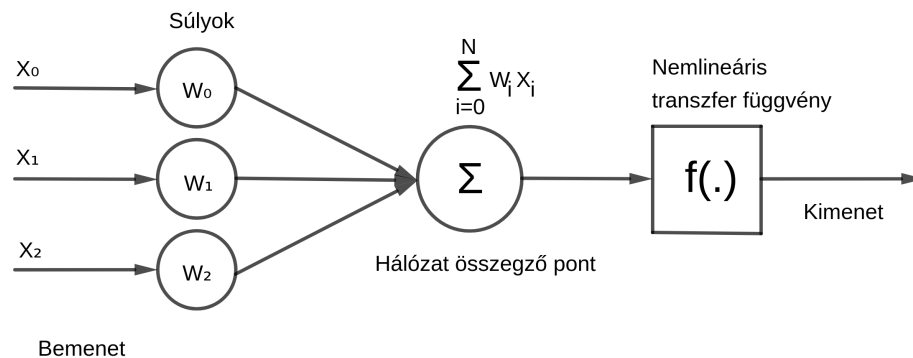
Egy olyan eszköz (legyen hardveres, vagy szoftveres), amely képes párhuzamosan feldolgozni a beérkező információt (esetünkben inputot) , illetve:

- neuronok (feldolgozást végző elemek) összekapcsolt rendszeréből áll
- minta alaján való tanulásra képes, tanulási algoritmussal rendelkezik
- előhívási algoritmus segítségével képes a korábban megtanult információ felhasználására/előhívására

Neurális hálózatok működésének két fő fázisa:

- A tanulási folyamat kezdete, a hálózat kialakítása. Egy lassabb folyamat, mely potenciálisan magában hordozhatja a sikertelen tanulási szakaszokat is.
- Az előhívási fázis, az előzővel ellentétben már jóval gyorsabb folyamat. Az előhívási algoritmus meghívása.

Legfontosabb építőeleme a neuron (feldolgozó elem), mely több bemenettel és egy kimenettel rendelkező eszköz, ami a bemenet és kimenet között egy nemlineáris leképezést hoz létre a megfelelő (transzfer) függvény segítségével.



4.1. ábra. Lokális memória nélküli neuron (perceptron)

Neuronok 3 fő típusa:

- **Bemeneti neuron:** Egyetlen bemenetük (a hálózat bemenete) és egyetlen kimenetük (a hálózat kimenete) van. Nincs jelfeldolgozó szerepük.
- **Kimeneti neuron:** a hálózatból a környezetbe továbbítja az információt.
- **Rejtett neuron:** bemenetük és kimenetük is a többi neuronhoz kapcsolódik.

Egy neuron akkor fog aktiválódni, ha a bemenetek súlyozott összege meghalad egy bizonyos értéket. A megfelelő bemenettel és eltolássúllyal rendelkező neuronok képesek lesznek logikai kapuként funkcionálni. Ezekkel a kapukkal képesek vagyunk az alapvető logikai függvényeket megjeleníteni.

OR logikai kapu (R implementáció):

```
library(neuralnet)

a1 <- c(0, 1, 0, 1)
a2 <- c(0, 0, 1, 1)
```



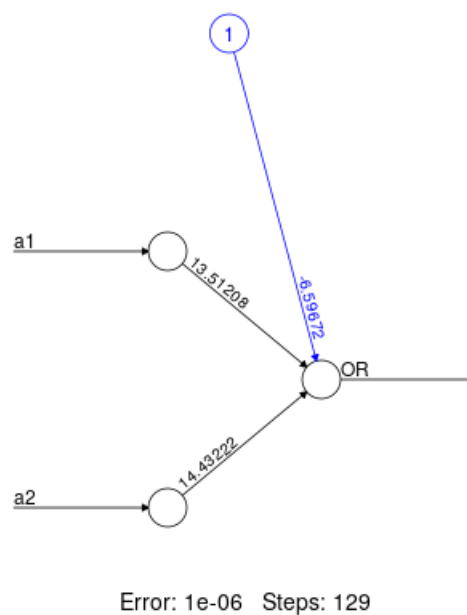
```
OR <- c(0,1,1,1)

or.data <- data.frame(a1, a2, OR)

nn.or <- neuralnet(OR~a1+a2, or.data, hidden=0, linear.output=FALSE, ←
  stepmax = 1e+07, threshold = 0.000001)

plot(nn.or)

compute(nn.or, or.data[,1:2])
```



4.2. ábra. OR logikai kapu

Eredmény:

```
$net.result
      [,1]
[1,] 0.001362976
[2,] 0.999008555
[3,] 0.999604714
[4,] 0.999999999
```

OR-AND logikai kapu (R implementáció):

```
a1 <- c(0,1,0,1)
a2 <- c(0,0,1,1)
```

```

OR    <- c(0,1,1,1)
AND   <- c(0,0,0,1)

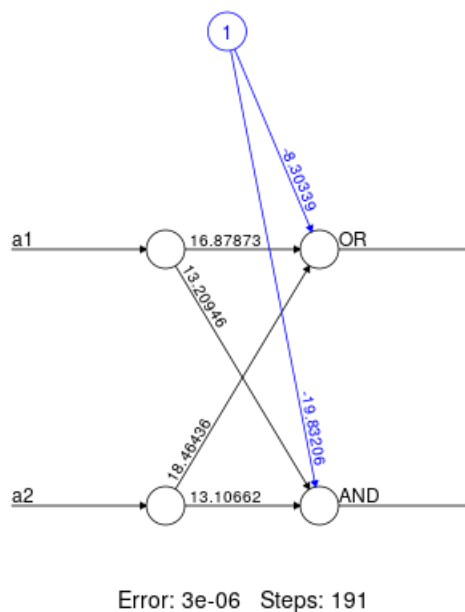
orand.data <- data.frame(a1, a2, OR, AND)

nn.orand <- neuralnet(OR+AND~a1+a2, orand.data, hidden=0, linear.output= <-
  FALSE, stepmax = 1e+07, threshold = 0.000001)

plot(nn.orand)

compute(nn.orand, orand.data[,1:2])

```



4.3. ábra. OR-AND logikai kapu

Eredmény:

```

$net.result
      [,1]      [,2]
[1,] 0.0002476137 2.438077e-09
[2,] 0.9998113326 1.328202e-03
[3,] 0.9999613515 1.198563e-03
[4,] 1.0000000000 9.984747e-01

```

EXOR logikai kapu rejtett neuron nélkül (R implementáció):

```
a1 <- c(0,1,0,1)
```

```

a2      <- c(0,0,1,1)
EXOR    <- c(0,1,1,0)

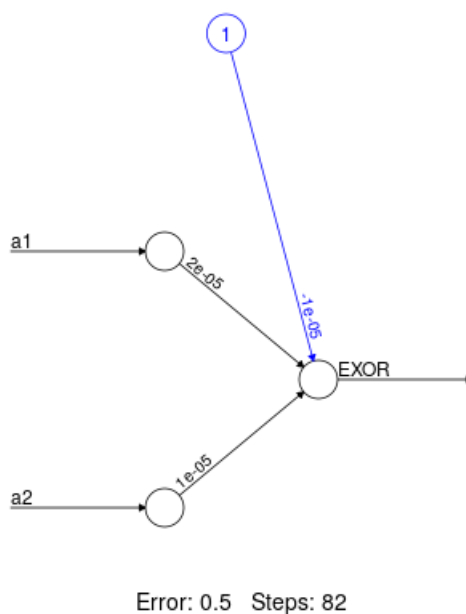
exor.data <- data.frame(a1, a2, EXOR)

nn.exor <- neuralnet(EXOR~a1+a2, exor.data, hidden=0, linear.output=FALSE, ←
  stepmax = 1e+07, threshold = 0.000001)

plot(nn.exor)

compute(nn.exor, exor.data[,1:2])

```



4.4. ábra. EXOR logikai kapu rejtett neuron nélkül

Eredmény:

```

$net.result
      [,1]
[1,] 0.4999972
[2,] 0.5000010
[3,] 0.4999987
[4,] 0.5000025

```

EXOR logikai kapu rejtett neuronokkal (R implementáció):

```

a1      <- c(0,1,0,1)

```

```

a2      <- c(0,0,1,1)
EXOR    <- c(0,1,1,0)

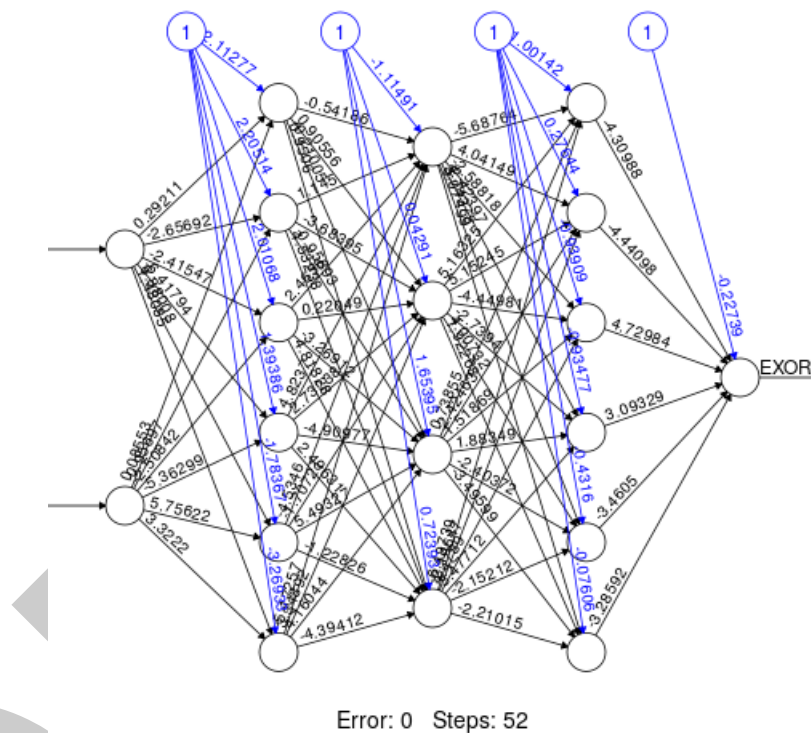
exor.data <- data.frame(a1, a2, EXOR)

nn.exor <- neuralnet(EXOR~a1+a2, exor.data, hidden=c(6, 4, 6), linear. <-
  output=FALSE, stepmax = 1e+07, threshold = 0.000001)

plot(nn.exor)

compute(nn.exor, exor.data[,1:2])

```



4.5. ábra. EXOR logikai kapu rejtett neuronokkal

Eredmény:

```

$net.result
      [,1]
[1,] 0.0001283067
[2,] 0.9993446185
[3,] 0.9993605209
[4,] 0.0001806955

```

Forrás 1: [bhax/attention_raising/NN_R/nn.r](https://github.com/bhax/attention_raising/NN_R/nn.r)

Megoldás videó: <https://youtu.be/Koyw6IH5ScQ>

Megoldás forrása 1: https://gitlab.com/nbatfai/bhax/tree/master/attention_raising/NN_R

Megoldás forrása 2: https://www.tankonyvtar.hu/hu/tartalom/tamop425/0026_neuralis_4_4/ch01s04.html#id4925

4.6. Hiba-visszaterjesztéses perceptron

C++

Egy többrétegű perceptron tanítása:

- Definiáljuk a kezdeti súlyokat.
- Az input végighalad a hálózaton, a súlyok változatlanok maradnak.
- Az ezáltal kapott kimeneti jelet összehasonlítjuk a tényleges kimeneti jellel.
- A hibát visszaküldjük a hálózat neuronjain keresztül és változtatunk a súlyokon, oly módon, hogy a hibák száma minimálisra csökkenjen

A kérdés viszont az, hogy milyen módon változtassuk meg a súlyokat a lehető legkevesebb hiba elérésének érdekében?

Ez a hiba-visszaterjesztés (back-propagation) algoritmussal történik a kimeneti réteg(ek)ből a rejtett rétegekbe.

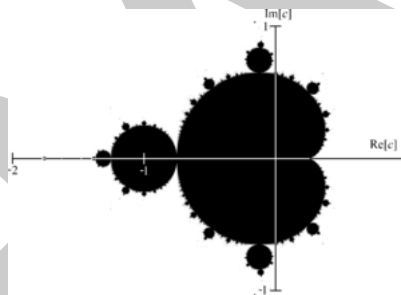
- A kimeneti rétegre meghatározzuk a hibák számát.
- Az így kapott hibaértékeket visszaterjesztjük a kimeneti réteg előtti, rejtett rétegre.
- A kapott hibákat egyre korábbi rétegekre terjesztjük oly módon, hogy a hiba skálázódik, az aktuális és az őt megelőző súlyok értékeinek függvényében.
- Az algoritmust addig folytatjuk, amíg el nem érjük a bemeneti réteget.

5. fejezet

Helló, Mandelbrot!

5.1. A Mandelbrot halmaz

Egy "örökké létező önmagát újra és újra reprodukáló, kaotikus, infláló világegyetemet ábrázol." A minta megfigyelése csak a legújabb technológiai vívmányok segítségével vált lehetségessé. Szabad szemmel ugyanis csak villózó fényeket láthatnánk. Pedig ezek valójában végtelenszer ismétlődő Mandelbrot halmazok.



5.1. ábra. Mandelbrot kétdimenziós koordinátarendszerben való ábrázolása

A Mandelbrot halmaz, egy, a komplex számsíkon vizuálisan ábrázolható fraktál-minta. Ahhoz, hogy értelmezhesük ezt a mondatot, haladjuk szépen sorban. Mi is az a komplex szám? Valószínűleg mindenki megtudja mondani, hogy mennyi 4-nek a négyzetgyöke(2), vagy 16-nak (4). De tudjuk-e, hogy mennyi -1-nek? A válasz pedig i , egy kitalált szám, melyet négyzetre emelve -1-et kapunk. Ha összekapcsolunk egy valós számot, pl.: 3-at az i -vel, akkor az így kapott $3i$ egy komplex szám lesz. A 3 a valós rész, az i pedig a képzetes (imaginárius) rész.

Ahhoz, hogy a komplex számokat vizualizálni tudjuk, szükség van egy Descartes-féle derékszögű koordináta rendszerhez hasonló 2 dimenziós térre. Ez lesz a komplex számsík. A vízszintes tengely lesz a valós tengely, a függőleges tengely pedig a képzetes tengely. Ezen a számsíkon képesek vagyunk ábrázolni minden komplex számot. Az így ábrázolt elemek talán legfontosabb eleme az origótól való távolsága. Ez által sokkal egyszerűbben vizualizálhatjuk nem csak magukat a számokat, de a velük elvégezhető műveleteket is (összeadás, kivonás). De hogy jutunk el innen, a Mandelbrot halmazig?

```
#include <png++/png.hpp>
```

```
#define N 800
#define M 800
#define MAXX 0.7
#define MINX -2.0
#define MAXY 1.35
#define MINY -1.35
```

A `png++` csomag segítségével fogjuk kirajzoltatni egy fájlba az eredményt. Definiáljuk az elkészíteni kívánt kép méreteit, illetve a koordináta-rendszeren a tartományt, amin belül szeretnénk, hogy a program dolgozzon.

```
void GeneratePNG( int tomb[N][M] )
{
    png::image< png::rgb_pixel > image(N, M);
    for (int x = 0; x < N; x++)
    {
        for (int y = 0; y < M; y++)
        {
            image[x][y] = png::rgb_pixel(tomb[x][y], tomb[x][y], tomb[x][y] ←
                );
        }
    }
    image.write("mandelbrot1.png");
}
```

Az ez után létrehozott függvény fogja nekünk kiírni a halmaz képét soronként és oszloponként haladva egy png fájlba.

```
struct Komplex
{
    double re, im;
};
```

A komplex számok ábrázolásához létrehozott struktúra valós és imaginárius értékekkel.

```
int main()
{
    int tomb[N][M];

    int i, j, k;

    double dx = (MAXX - MINX) / N;
    double dy = (MAXY - MINY) / M;

    struct Komplex C, Z, Zuj;

    int iteracio;

    for (i = 0; i < M; i++)
    {
```

```
for (j = 0; j < N; j++)
{
    C.re = MINX + j * dx;
    C.im = MAXY - i * dy;

    Z.re = 0;
    Z.im = 0;
    iteracio = 0;

    while(Z.re * Z.re + Z.im * Z.im < 4 && iteracio++ < 255)
    {
        Zuj.re = Z.re * Z.re - Z.im * Z.im + C.re;
        Zuj.im = 2 * Z.re * Z.im + C.im;
        Z.re = Zuj.re;
        Z.im = Zuj.im;
    }

    tomb[i][j] = 256 - iteracio;
}

GeneratePNG(tomb);

return 0;
}
```

Számolni kezdjük a $f_c(z)=z^2+c$ iterációit. Végiglépkedünk a rácspontokon. C.re (valós rész) és C.im (imaginárius rész) a háló rácspontjainak megfelelő komplex számot alkotják. Z.re és Z.im iterációs komplex szám részei. Amennyiben az iterációk során a c távolsága a Z_0 ponttól (azaz az origótól) nagyobb lesz mint 2, vagy eléjük az iterációs maximumot, akkor az érték nem lesz eleme a mandelbrot halmaznak.

Megoldás forrása: https://www.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat/apbs02.html#mandelbrot_halm

5.2. A Mandelbrot halmaz a `std::complex` osztállyal

Az `complex` osztály beemelésével lehetőségünk lesz olyan komplex típusokat deklarálni, amelyek egy változónak a valós és imaginárius részét is el tudják tárolni. Így az algoritmus komplex számait egy egyszerűen lehet kezelni.

```
#include <iostream>
#include "png++/png.hpp"
#include <complex>

int
main ( int argc, char *argv[] )
{
    int szelesseg = 1366;
    int magassag = 1024;
```



```
int iteraciosHatar = 255;
double a = -2;
double b = 0.7;
double c = -1.35;
double d = 1.35;
```

Definiáljuk az elkészíteni kívánt kép méreteit, illetve a koordinátarendszeren a tartományt, amin belül szeretnénk, hogy a program dolgozzon.

```
if ( argc == 9 )
{
    szelesseg = atoi ( argv[2] );
    magassag =  atoi ( argv[3] );
    iteraciosHatar =  atoi ( argv[4] );
    a = atof ( argv[5] );
    b = atof ( argv[6] );
    c = atof ( argv[7] );
    d = atof ( argv[8] );
}
else
{
    std::cout << "Hasznalat: ./3.1.2 fajlnev szelesseg magassag n a b c d ↵" << std::endl;
    return -1;
}
```

A program futtatásánál megadható argumentumokat vizsgáljuk. Minden egyes argumentumot beolvasáskor egy egészzé vagy egy lebegőpontos számmá alakítunk át, majd elhelyezzük őket egy tömbben. Így állíthatjuk a képméretet és a valós-számrendszer beli tartományt.

```
png::image < png::rgb_pixel > kep ( szelesseg, magassag );

double dx = ( b - a ) / szelesseg;
double dy = ( d - c ) / magassag;
double reC, imC, reZ, imZ;
int iteracio = 0;

std::cout << "Szamitas\n";

// j megy a sorokon
for ( int j = 0; j < magassag; ++j )
{
    // k megy az oszlopokon

    for ( int k = 0; k < szelesseg; ++k )
    {

        // c = (reC, imC) a halo racspontjainak
        // megfelelo komplex szam

        reC = a + k * dx;
```

```

    imC = d - j * dy;
    std::complex<double> c ( reC, imC );

    std::complex<double> z_n ( 0, 0 );
    iteracio = 0;

    while ( std::abs ( z_n ) < 4 && iteracio < iteraciosHatar )
    {
        z_n = z_n * z_n + c;

        ++iteracio;
    }

    kep.set_pixel ( k, j,
                    png::rgb_pixel ( iteracio%255, (iteracio*iteracio <=
                    )%255, 0 ) );
}

int szazalek = ( double ) j / ( double ) magassag * 100.0;
std::cout << "\r" << szazalek << "%" << std::flush;
}

kep.write ( argv[1] );
std::cout << "\r" << argv[1] << " mentve." << std::endl;
}

```

A kép, mint 2 dimenziós objektum pixeleit egyesével kezdjük el vizsgálni, hogy része lesz-e a Mandelbrot halmazunknak. A while ciklus belseje az előző Mandelbrot halmaz pontban vázolt algoritmushoz képest nagyon leegyszerűsödik, hála a `std::complex` osztálynak.

Megoldás videó: <https://youtu.be/gvaqijHIRUs>

5.3. Biomorfok

Mik is a biomorfok? A biomorfok olyan alakzatok, melyek nagyon hasonlítanak egy természetes organizmus mikroszkópikus képére, viszont matematikai és nem biológiai eredetűek. Mi esetünkben egy kétdimenziós számsíkon ábrázolt fraktólokról van szó. Legismertebb fraktálok: "Julia-halmazok", "Mandelbrot-halmaz" és a véletlenül felfedezett "Biomorfok".

A különbség a Mandelbrot halmaz és a Julia halmazok között az, hogy a Mandelbrot-halmaz komplex iterációban a `C` változó:

```

// j megy a sorokon
for ( int j = 0; j < magassag; ++j )
{
    for ( int k = 0; k < szelesseg; ++k )
    {

```

```
// c = (reC, imC) a halo racspontjainak
// megfelelo komplex szam

reC = a + k * dx;
imC = d - j * dy;
std::complex<double> c ( reC, imC );

std::complex<double> z_n ( 0, 0 );
iteracio = 0;

while ( std::abs ( z_n ) < 4 && iteracio < iteraciosHatar )
{
    z_n = z_n * z_n + c;

    ++iteracio;
}
```

Míg a Julia-halmaz ábrázolásánál a c konstans lesz. Minden vizsgált z rácspontra ugyanaz:

```
// j megy a sorokon
for ( int j = 0; j < magassag; ++j )
{
    // k megy az oszlopokon
    for ( int k = 0; k < szelesseg; ++k )
    {
        double reZ = a + k * dx;
        double imZ = d - j * dy;
        std::complex<double> z_n ( reZ, imZ );

        int iteracio = 0;
        for (int i=0; i < iteraciosHatar; ++i)
        {
            z_n = std::pow(z_n, 3) + cc;
            if(std::real ( z_n ) > R || std::imag ( z_n ) > R)
            {
                iteracio = i;
                break;
            }
        }
    }
}
```

A korábbi Mandelbrot halmazt kiszámoló forráskódot módosítva:

```
int
main ( int argc, char *argv[] )
{

    int szelesseg = 1920;
    int magassag = 1080;
```

```
int iteraciosHatar = 255;
double xmin = -1.9;
double xmax = 0.7;
double ymin = -1.3;
double ymax = 1.3;
double reC = .285, imC = 0;
double R = 10.0;

if ( argc == 12 )
{
    szelesseg = atoi ( argv[2] );
    magassag = atoi ( argv[3] );
    iteraciosHatar = atoi ( argv[4] );
    xmin = atof ( argv[5] );
    xmax = atof ( argv[6] );
    ymin = atof ( argv[7] );
    ymax = atof ( argv[8] );
    reC = atof ( argv[9] );
    imC = atof ( argv[10] );
    R = atof ( argv[11] );
}
else
{
    std::cout << "Hasznalat: ./3.1.2 fajlnev szelesseg magassag n a b c ↵
        d reC imC R" << std::endl;
    return -1;
}

png::image < png::rgb_pixel > kep ( szelesseg, magassag );

double dx = ( xmax - xmin ) / szelesseg;
double dy = ( ymax - ymin ) / magassag;

std::complex<double> cc ( reC, imC );

std::cout << "Szamitas\n";

// j megy a sorokon
for ( int y = 0; y < magassag; ++y )
{
    // k megy az oszlopokon

    for ( int x = 0; x < szelesseg; ++x )
    {

        double reZ = xmin + x * dx;
        double imZ = ymax - y * dy;
        std::complex<double> z_n ( reZ, imZ );
```

```
int iteracio = 0;
for (int i=0; i < iteraciosHatar; ++i)
{
    z_n = std::pow(z_n, 3) + cc;
    //z_n = std::pow(z_n, 2) + std::sin(z_n) + cc;
    if(std::real ( z_n ) > R || std::imag ( z_n ) > R)
    {
        iteracio = i;
        break;
    }
}

kep.set_pixel ( x, y,
                png::rgb_pixel ( (iteracio*20)%255, (iteracio *
                *40)%255, (iteracio*60)%255 ));
}

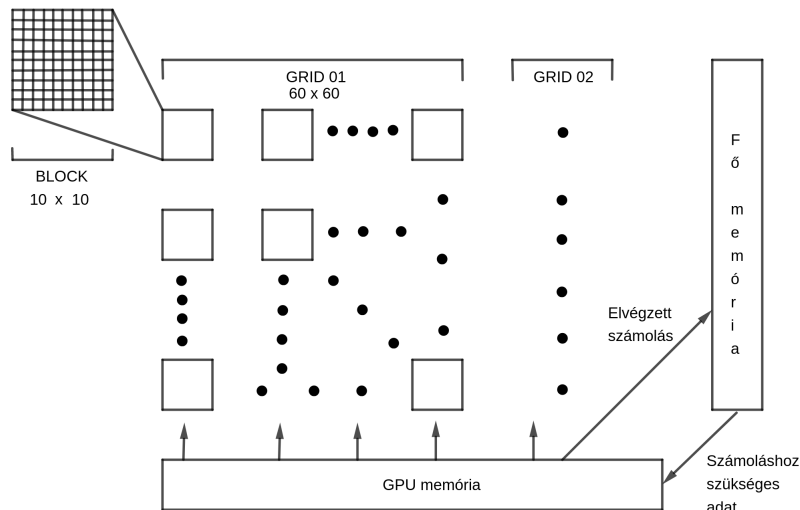
int szazalek = ( double ) y / ( double ) magassag * 100.0;
std::cout << "\r" << szazalek << "%" << std::flush;
}

kep.write ( argv[1] );
std::cout << "\r" << argv[1] << " mentve." << std::endl;
}
```

Megoldás videó: <https://youtu.be/IJMbgRzY76E>

Megoldás forrása: https://gitlab.com/nbatfai/bhax/tree/master/attention_raising/Biomorf

5.4. A Mandelbrot halmaz CUDA megvalósítása



5.2. ábra. CUDA-s Mandelbrot halmaz váza, 600 x 600-as kép esetén:

```
#include <png++/image.hpp>
#include <png++/rgb_pixel.hpp>

#include <sys/times.h>
#include <iostream>

#define MERET 600
#define ITER_HAT 32000

__device__ int
mandel (int k, int j)
{
    // Végigzongorázza a CUDA a szélesség x magasság rácsot:
    // most éppen a j. sor k. oszlopában vagyunk

    // számítás adatai
    float a = -2.0, b = .7, c = -1.35, d = 1.35;
    int szelesseg = MERET, magassag = MERET, iteraciosHatar = ITER_HAT;

    // a számítás
    float dx = (b - a) / szelesseg;
    float dy = (d - c) / magassag;
    float reC, imC, reZ, imZ, ujreZ, ujimZ;
    // Hány iterációt csináltunk?
    int iteracio = 0;
```

```
// c = (reC, imC) a rács csomópontjainak
// megfelelő komplex szám
reC = a + k * dx;
imC = d - j * dy;
// z_0 = 0 = (reZ, imZ)
reZ = 0.0;
imZ = 0.0;
iteracio = 0;
// z_{n+1} = z_n * z_n + c iterációk
// számítása, amíg |z_n| < 2 vagy még
// nem értük el a 255 iterációt, ha
// viszont elértük, akkor úgy vesszük,
// hogy a kiindulási c komplex számra
// az iteráció konvergens, azaz a c a
// Mandelbrot halmaz eleme
while (reZ * reZ + imZ * imZ < 4 && iteracio < iteraciosHatar)
{
    // z_{n+1} = z_n * z_n + c
    ujureZ = reZ * reZ - imZ * imZ + reC;
    ujimZ = 2 * reZ * imZ + imC;
    reZ = ujureZ;
    imZ = ujimZ;

    ++iteracio;
}
return iteracio;
}

/*
__global__ void
mandelkernel (int *kepadat)
{
    int j = blockIdx.x;
    int k = blockIdx.y;

    kepadat[j + k * MERET] = mandel (j, k);
}
*/

__global__ void
mandelkernel (int *kepadat)
{
    int tj = threadIdx.x;
    int tk = threadIdx.y;
```

```
int j = blockIdx.x * 10 + tj;
int k = blockIdx.y * 10 + tk;

kepadat[j + k * MERET] = mandel (j, k);
}

void
cudamandel (int kepadat[MERET][MERET])
{
    int *device_kepadat;
    cudaMalloc ((void **) &device_kepadat, MERET * MERET * sizeof (int));

    // dim3 grid (MERET, MERET);
    // mandelkernel <<< grid, 1 >>> (device_kepadat);

    dim3 grid (MERET / 10, MERET / 10);
    dim3 tgrid (10, 10);
    mandelkernel <<< grid, tgrid >>> (device_kepadat);

    cudaMemcpy (kepadat, device_kepadat,
                MERET * MERET * sizeof (int), cudaMemcpyDeviceToHost);
    cudaFree (device_kepadat);
}

int
main (int argc, char *argv[])
{
    // Mérünk időt (PP 64)
    clock_t delta = clock ();
    // Mérünk időt (PP 66)
    struct tms tmsbuf1, tmsbuf2;
    times (&tmsbuf1);

    if (argc != 2)
    {
        std::cout << "Hasznalat: ./mandelpngc fajlnev";
        return -1;
    }

    int kepadat[MERET][MERET];

    cudamandel (kepadat);

    png::image < png::rgb_pixel > kep (MERET, MERET);
```

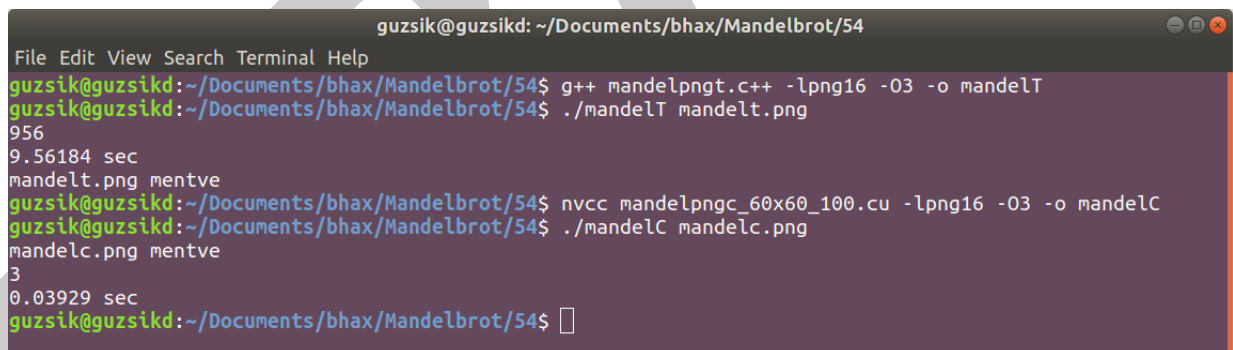


```
for (int j = 0; j < MERET; ++j)
{
    //sor = j;
    for (int k = 0; k < MERET; ++k)
    {
        kep.set_pixel (k, j,
            png::rgb_pixel (255 -
                (255 * kepadat[j][k]) / ITER_HAT,
                255 -
                (255 * kepadat[j][k]) / ITER_HAT,
                255 -
                (255 * kepadat[j][k]) / ITER_HAT));
    }
}
kep.write (argv[1]);

std::cout << argv[1] << " mentve" << std::endl;

times (&tmsbuf2);
std::cout << tmsbuf2.tms_utime - tmsbuf1.tms_utime
    + tmsbuf2.tms_stime - tmsbuf1.tms_stime << std::endl;

delta = clock () - delta;
std::cout << (float) delta / CLOCKS_PER_SEC << " sec" << std::endl;
}
```



```
guzsik@guzsikd: ~/Documents/bhax/Mandelbrot/54
File Edit View Search Terminal Help
guzsik@guzsikd:~/Documents/bhax/Mandelbrot/54$ g++ mandelpngt.cpp -lpng16 -O3 -o mandelT
guzsik@guzsikd:~/Documents/bhax/Mandelbrot/54$ ./mandelT mandelt.png
956
9.56184 sec
mandelt.png mentve
guzsik@guzsikd:~/Documents/bhax/Mandelbrot/54$ nvcc mandelpngc_60x60_100.cu -lpng16 -O3 -o mandelC
guzsik@guzsikd:~/Documents/bhax/Mandelbrot/54$ ./mandelC mandelc.png
mandelc.png mentve
3
0.03929 sec
guzsik@guzsikd:~/Documents/bhax/Mandelbrot/54$
```

5.3. ábra. CUDA-s kép kiszámítása ennyivel gyorsabb

Megoldás videó: <https://www.youtube.com/watch?v=gvaqijHIRUs>

Megoldás forrása: https://www.tankonyvtar.hu/hu/tartalom/tamop412A/2011-0063_01_parhuzamos_prog_linux/adatok.html

5.5. Mandelbrot nagyító és utazó C++ nyelven

Építs GUI-t a Mandelbrot algoritmusra, lehessen egérrel nagyítani egy területet, illetve egy pontot egérrel kiválasztva vizualizálja onnan a komplex iteráció bejárta z_n komplex számokat!

Szükséges források:

[Programozó Páternoszter](#)

[UDPROG közösség](#)

5.6. Mandelbrot nagyító és utazó Java nyelven

A projektet a [Javát tanítok](#) oldalán található útmutató alapján készítettem el.

```
public class MandelbrotHalmaz extends java.awt.Frame implements Runnable {

    protected double a, b, c, d;

    protected int szélesség, magasság;

    protected java.awt.image.BufferedImage kép;

    protected int iterációsHatár = 255;

    protected boolean számításFut = false;

    protected int sor = 0;

    protected static int pillanatfelvételSzámláló = 0;

    public MandelbrotHalmaz(double a, double b, double c, double d,
        int szélesség, int iterációsHatár) {
        this.a = a;
        this.b = b;
        this.c = c;
        this.d = d;
        this.szélesség = szélesség;
        this.iterációsHatár = iterációsHatár;

        this.magasság = (int)(szélesség * ((d-c)/(b-a)));

        kép = new java.awt.image.BufferedImage(szélesség, magasság,
            java.awt.image.BufferedImage.TYPE_INT_RGB);

        addWindowListener(new java.awt.event.WindowAdapter() {
            public void windowClosing(java.awt.event.WindowEvent e) {
                setVisible(false);
                System.exit(0);
            }
        });
    }
}
```

```
    }  
    });
```

Deklaráljuk a komplex sík tartományát, illetve a síkra vetített hálót és adatait.

```
addKeyListener(new java.awt.event.KeyAdapter() {  
  
    public void keyPressed(java.awt.event.KeyEvent e) {  
        if(e.getKeyCode() == java.awt.event.KeyEvent.VK_S)  
            pillanatfelvétel();  
  
        else if(e.getKeyCode() == java.awt.event.KeyEvent.VK_N) {  
            if(számításFut == false) {  
                MandelbrotHalmaz.this.iterációsHatár += 256;  
  
                számításFut = true;  
                new Thread(MandelbrotHalmaz.this).start();  
            }  
  
        } else if(e.getKeyCode() == java.awt.event.KeyEvent.VK_M) {  
            if(számításFut == false) {  
                MandelbrotHalmaz.this.iterációsHatár += 10*256;  
  
                számításFut = true;  
                new Thread(MandelbrotHalmaz.this).start();  
            }  
        }  
    }  
});
```

Figyeljük a billentyűzetről érkező inputokat.

- S lenyomására pillanatkép készül.
- N lenyomására újraszámoljuk és kicsivel pontosabb számolást végzünk.
- M lenyomására újraszámoljuk és sokkal pontosabb számolást végzünk.

```
setTitle("A Mandelbrot halmaz");  
setResizable(false);  
setSize(szélesség, magasság);  
setVisible(true);  
  
számításFut = true;  
new Thread(this).start();  
}  
  
public void paint(java.awt.Graphics g) {  
  
    g.drawImage(kép, 0, 0, this);
```

```
        if(számításFut) {
            g.setColor(java.awt.Color.RED);
            g.drawLine(0, sor, getWidth(), sor);
        }
    }
```

A megjeleníteni kívánt ablak beállításai. A számítást futás közben egy vörös szakasszal jelöljük.

```
public void pillanatfelvétel() {

    java.awt.image.BufferedImage mentKép =
        new java.awt.image.BufferedImage(szélesség, magasság,
            java.awt.image.BufferedImage.TYPE_INT_RGB);
    java.awt.Graphics g = mentKép.getGraphics();
    g.drawImage(kép, 0, 0, this);
    g.setColor(java.awt.Color.BLUE);
    g.drawString("a=" + a, 10, 15);
    g.drawString("b=" + b, 10, 30);
    g.drawString("c=" + c, 10, 45);
    g.drawString("d=" + d, 10, 60);
    g.drawString("n=" + iterációsHatár, 10, 75);
    g.dispose();

    StringBuffer sb = new StringBuffer();
    sb = sb.delete(0, sb.length());
    sb.append("MandelbrotHalmaz_");
    sb.append(++pillanatfelvételSzámláló);
    sb.append("_");

    sb.append(a);
    sb.append("_");
    sb.append(b);
    sb.append("_");
    sb.append(c);
    sb.append("_");
    sb.append(d);
    sb.append(".png");

    try {
        javax.imageio.ImageIO.write(mentKép, "png",
            new java.io.File(sb.toString()));
    } catch (java.io.IOException e) {
        e.printStackTrace();
    }
}
```

A pillanatkép elmentése, a kép adatainak beállítása, illetve az aktuális adatok képre mentése.

```
public void run() {
```

```
double dx = (b-a)/szélesség;
double dy = (d-c)/magasság;
double reC, imC, reZ, imZ, ujreZ, ujimZ;
int rgb;

int iteráció = 0;

for(int j=0; j<magasság; ++j) {
    sor = j;
    for(int k=0; k<szélesség; ++k) {

        reC = a+k*dx;
        imC = d-j*dy;

        reZ = 0;
        imZ = 0;
        iteráció = 0;

        while(reZ*reZ + imZ*imZ < 4 && iteráció < iterációsHatár) {

            ujreZ = reZ*reZ - imZ*imZ + reC;
            ujimZ = 2*reZ*imZ + imC;
            reZ = ujreZ;
            imZ = ujimZ;

            ++iteráció;

        }

        iteráció %= 256;

        rgb = (255-iteráció)|
            ((255-iteráció) << 8) |
            ((255-iteráció) << 16);

        kép.setRGB(k, j, rgb);
    }
    repaint();
}
számításFut = false;

public static void main(String[] args) {

    new MandelbrotHalmaz(-2.0, .7, -1.35, 1.35, 600, 255);

}
```

A program lelke, ami a halmaz számítását végzi. a Dupla for ciklussal végigmegyünk a szélesség x magasság dimenzióin. A $c = (reC, imC)$ a rácpontoknak megfelelő komplex szám. $z_{\{n+1\}} = z_n *$

$z_n + c$ -t iteráljuk amíg el nem érjük a 2-t, vagy az iterációs határt. Ha elérjük a határt, akkor a vizsgált pont a halmaz része, tehát az iteráció konvergens.

A nagyító programot ennek a programnak a továbbfejlesztésével hozhatjuk létre. Ehhez létrehozunk egy újabb osztályt.

```
public class MandelbrotHalmazNagyító extends MandelbrotHalmaz {  
  
    private int x, y;  
  
    private int mx, my;
```

Itt megjegyezzük a nagyítani kívánt terület bal felső sarkát, valamint a terület szélességét és magasságát.

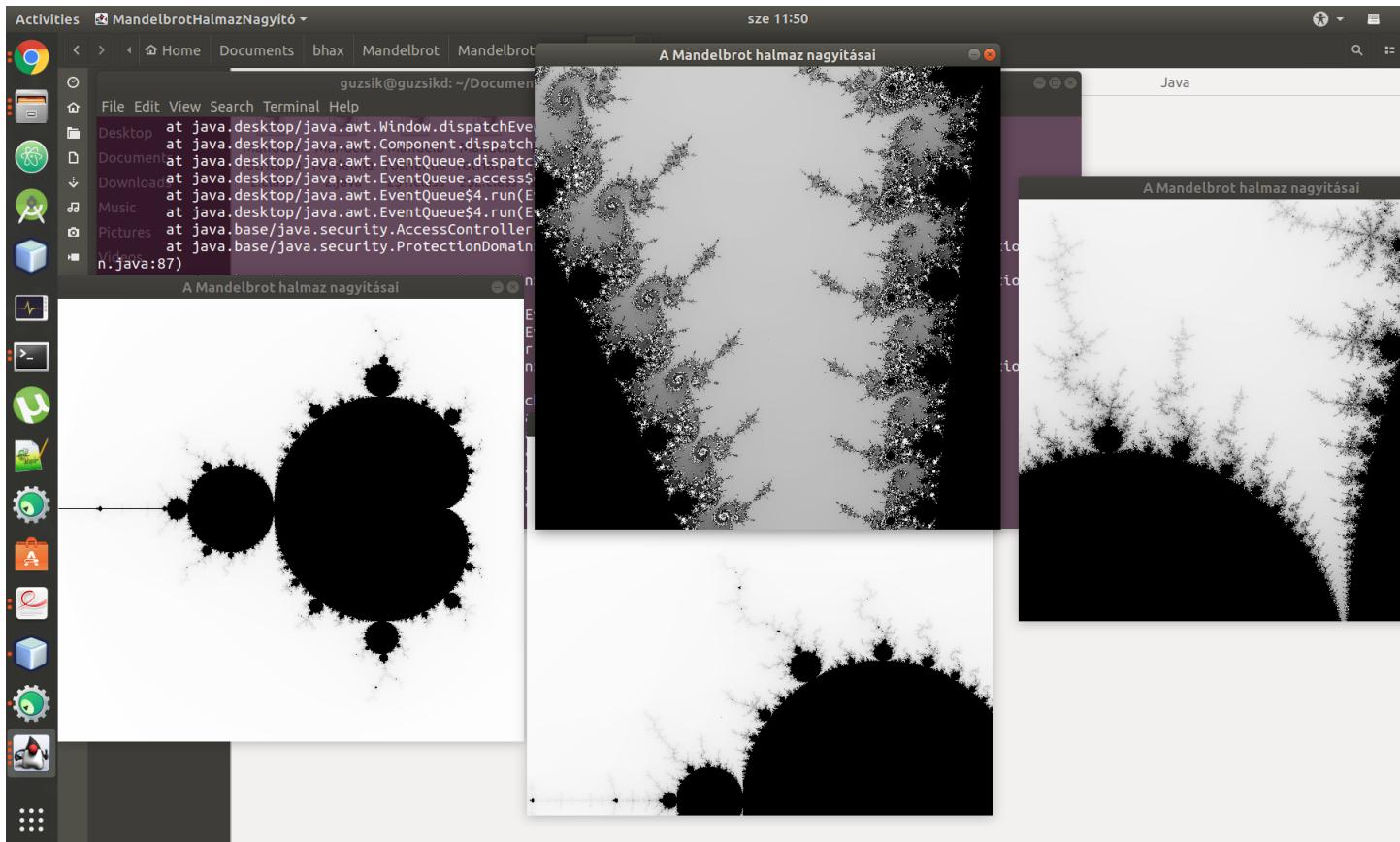
```
    public MandelbrotHalmazNagyító(double a, double b, double c, double d,  
        int szélesség, int iterációsHatár) {  
  
        super(a, b, c, d, szélesség, iterációsHatár);  
        setTitle("A Mandelbrot halmaz nagyításai");  
  
        addMouseListener(new java.awt.event.MouseAdapter() {  
  
            public void mousePressed(java.awt.event.MouseEvent m) {  
  
                x = m.getX();  
                y = m.getY();  
                mx = 0;  
                my = 0;  
                repaint();  
            }  
  
            public void mouseReleased(java.awt.event.MouseEvent m) {  
                double dx = (MandelbrotHalmazNagyító.this.b  
                    - MandelbrotHalmazNagyító.this.a)  
                    /MandelbrotHalmazNagyító.this.szélesség;  
                double dy = (MandelbrotHalmazNagyító.this.d  
                    - MandelbrotHalmazNagyító.this.c)  
                    /MandelbrotHalmazNagyító.this.magasság;  
  
                new MandelbrotHalmazNagyító(MandelbrotHalmazNagyító.this.a+ ←  
                    x*dx,  
                        MandelbrotHalmazNagyító.this.a+x*dx+mx*dx,  
                        MandelbrotHalmazNagyító.this.d-y*dy-my*dy,  
                        MandelbrotHalmazNagyító.this.d-y*dy,  
                        600,  
                        MandelbrotHalmazNagyító.this.iterációsHatár);  
            }  
        });  
  
        addMouseMotionListener(new java.awt.event.MouseMotionAdapter() {  
  
            public void mouseDragged(java.awt.event.MouseEvent m) {
```

```
        mx = m.getX() - x;  
        my = m.getY() - y;  
        repaint();  
    }  
});  
}
```

Egérkattintással és nyomvatartással kijelöljük a nagyítani kívánt területet. Ha felengedjük az egeret, akkor az újonnan kijelölt terület ismét kiszámításra kerül.

```
public void paint(java.awt.Graphics g) {  
  
    g.drawImage(kép, 0, 0, this);  
  
    if(számításFut) {  
        g.setColor(java.awt.Color.RED);  
        g.drawLine(0, sor, getWidth(), sor);  
    }  
  
    g.setColor(java.awt.Color.GREEN);  
    g.drawRect(x, y, mx, my);  
}  
  
public static void main(String[] args) {  
  
    new MandelbrotHalmazNagyító(-2.0, .7, -1.35, 1.35, 600, 255);  
}  
}
```

A nagyítani kívánt területet újrarajzoltatjuk, ha éppen számol a program, akkor egy vörös szakasszal jelöljük, hogy hol tart éppen.



5.4. ábra. Mandelbrot nagyítása JAVA környezetben

6. fejezet

Helló, Welch!

6.1. Első osztályom

Valósítsd meg C++-ban és Java-ban az módosított polártranszformációs algoritmust! A matek háttér teljesen irreleváns, csak annyiban érdekes, hogy az algoritmus egy számítása során két normálist számol ki, az egyiket elspájzolod és egy további logikai taggal az osztályban jelzed, hogy van vagy nincs eltérő kiszámolt szám.

Megoldás videó:

Megoldás forrása:

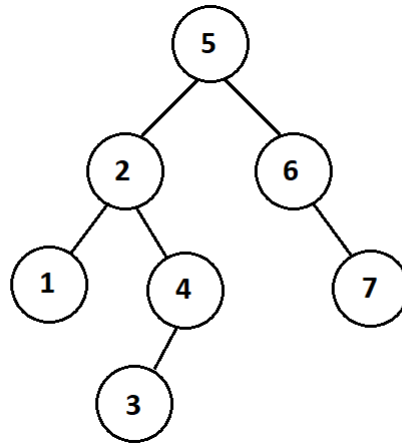
Tanulságok, tapasztalatok, magyarázat... térj ki arra is, hogy a JDK forrásaiban a Sun programozói pont úgy csinálták meg ahogyan te is, azaz az OO nemhogy nem nehéz, hanem éppen természetes neked!

6.2. LZW

Valósítsd meg C-ben az LZW algoritmus fa-építését!

Az Lempel–Ziv–Welch (LZW) algoritmust az 1980-as évek közepén, Abraham Lempel és Jacob Ziv már létező algoritmusát továbbfejlesztve, Terry Welch publikálta. Az algoritmus a UNIX alapú rendszerek fájl-tömörítő segédprogramja által terjedt el leginkább, továbbá GIF kiterjesztésű képek és PDF fájlok veszteségmentes tömörítéséhez is használják. Az USA-ban 2003-ban, a világ többi részén 2004-ben az algoritmus szabadalma lejárt, ezért azóta alkalmazása a háttérbe szorult.

A bináris fát úgy építjük föl, hogy a bemenetre érkező nullákat és egyeseket olvassuk. Ha olyan egységet olvasunk be, ami már korábban volt, akkor olvassuk tovább. Az így kapott új egységet fogjuk a fa gyökerétől kezdve "lelépkedni". Ha az adott egység ábrázolva van, visszaugrunk a gyökérbe és olvassuk tovább az inputot.



6.1. ábra. Binfa felépítése:

A program készítéséhez [Programozó Páternoszter](#)-en található útmutatót hívtam segítségül.

A kód bemutatása lépésről lépésre:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>

typedef struct binfa
{
    int ertek;
    struct binfa *bal_nulla;
    struct binfa *jobb_egy;
} BINFA, *BINFA_PTR;
```

Itt definiáljuk a binfa struktúrát.

```
BINFA_PTR
uj_elem ()
{
    BINFA_PTR p;

    if ((p = (BINFA_PTR) malloc (sizeof (BINFA))) == NULL)
    {
        perror ("memoria");
        exit (EXIT_FAILURE);
    }
    return p;
}
```

Az új elem létrehozásakor memóriát szabadítunk majd fel, hiba esetén viszont kilépünk a programból.

```
extern void kiir (BINFA_PTR elem);
extern void szabadit (BINFA_PTR elem);
```

Deklaráljuk, de még nem definiáljuk a fa kiírásához és a lefoglalt memória felszabadításához használt függvényeket. (Egyelőre nem foglalunk le nekik helyet a memóriában.)

```
int
main (int argc, char **argv)
{
    char b;

    BINFA_PTR gyoker = uj_elem ();
    gyoker->ertek = '/';
    BINFA_PTR fa = gyoker;

    while (read (0, (void *) &b, 1))
    {
        write (1, &b, 1);
        if (b == '0')
        {
            if (fa->bal_nulla == NULL)
            {
                fa->bal_nulla = uj_elem ();
                fa->bal_nulla->ertek = 0;
                fa->bal_nulla->bal_nulla = fa->bal_nulla->jobb_egy = NULL;
                fa = gyoker;
            }
            else
            {
                fa = fa->bal_nulla;
            }
        }
        else
        {
            if (fa->jobb_egy == NULL)
            {
                fa->jobb_egy = uj_elem ();
                fa->jobb_egy->ertek = 1;
                fa->jobb_egy->bal_nulla = fa->jobb_egy->jobb_egy = NULL;
                fa = gyoker;
            }
            else
            {
                fa = fa->jobb_egy;
            }
        }
    }

    printf ("\n");
    kiir (gyoker);
    extern int max_melyseg;
    printf ("melyseg=%d\n", max_melyseg);
    szabadit (gyoker);
}
```

```
}
```

Először létrehozunk a gyökeret és ráállítjuk a fa pointert. Olvassuk az inputon érkező 0-kat és 1-eseket.

- Ha 0-t olvasunk és az aktuális nodenak (amire a fa pointerünk jelenleg mutat) nincs nullás gyermeke, akkor az `uj_elem` függvénnyel létrehozunk neki egyet, majd megkapja értékül a 0-t. Ezután ennek az újonnan létrehozott gyermeknek beállítjuk a bal és jobb gyermekeit NULL pointerekre és visszaugrunk a pointerrel a gyökerre. Ha a beolvasáskor már volt nullás gyermeke az aktuális nodenak, akkor a mutatót ráállítjuk.
- Ha 1-t olvasunk és az aktuális nodenak (amire a fa pointerünk jelenleg mutat) nincs egyes gyermeke, akkor az `uj_elem` függvénnyel létrehozunk neki egyet, majd megkapja értékül az 1-t. Ezután ennek az újonnan létrehozott gyermeknek beállítjuk a bal és jobb gyermekeit NULL pointerekre és visszaugrunk a pointerrel a gyökerre. Ha a beolvasáskor már volt egyes gyermeke az aktuális nodenak, akkor a mutatót ráállítjuk.

```
static int melyseg = 0;
int max_melyseg = 0;

void
kiir (BINFA_PTR elem)
{
    if (elem != NULL)
    {
        ++melyseg;
        if (melyseg > max_melyseg)
            max_melyseg = melyseg;
        kiir (elem->jobb_egy);
        for (int i = 0; i < melyseg; ++i)
            printf ("---");
        printf ("%c(%d)\n", elem->ertek < 2 ? '0' + elem->ertek : elem->↵
            ertek,
            melyseg);
        kiir (elem->bal_nulla);
        --melyseg;
    }
}
```

A `kiir` függvénnyel jelenítjük meg magát a fát a parancssorban. Mivel 90 fokkal el van forgatva az eredmény balra és fentről lefelé írjuk ki az ágakat ezért inorder bejárás esetén először a jobb oldali ágat, majd a gyökeret, majd végül a bal oldali ágat rajzoltatjuk ki. Közben számljuk a fa mélységét is.

```
void
szabadit (BINFA_PTR elem)
{
    if (elem != NULL)
    {
        szabadit (elem->jobb_egy);
        szabadit (elem->bal_nulla);
    }
}
```

```
    free (elem);  
}  
}
```

Rekurzívan hívjuk a függvényt, amivel felszabadítjuk a korábban lefoglalt elemeket.

A programot fordítva és futtatva tetszőleges inputtal:



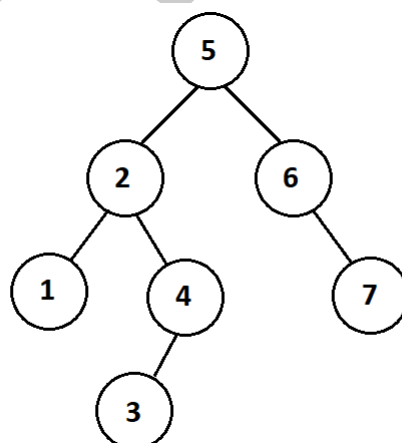
```
guzsik@guzsikd: ~/Documents/bhax/Welch/Binfa  
File Edit View Search Terminal Help  
guzsik@guzsikd:~/Documents/bhax/Welch/Binfa$ ./lzw <fa.txt  
01111001001001000111  
  
-----1(4)  
-----1(3)  
-----1(2)  
-----0(3)  
-----0(4)  
-----0(5)  
---/(1)  
-----1(3)  
-----0(2)  
-----0(3)  
melyseg=5  
guzsik@guzsikd:~/Documents/bhax/Welch/Binfa$
```

Megoldás forrása: https://progpater.blog.hu/2011/02/19/gyonyor_a_tomor/

6.3. Fabejárás

Járd be az előző (inorder bejárású) fát pre- és posztorder is!

Az előző LZW bináris fa feladatban a fát Inorder módon jártuk be. Azaz a bal oldali részfát dolgozzuk fel, majd a gyökeret és végül a jobb oldali részfát.



6.2. ábra. Emlékeztetőül az Inorder bejárásra:

A forráskódon belül, ezt a `kiír` függvényenél láthattuk.

```

static int melyseg = 0;
int max_melyseg = 0;

void
kiir (BINFA_PTR elem)
{
    if (elem != NULL)
    {
        ++melyseg;
        if (melyseg > max_melyseg)
            max_melyseg = melyseg;
        kiir (elem->jobb_egy);
        for (int i = 0; i < melyseg; ++i)
            printf ("---");
        printf ("%c(%d)\n", elem->ertek < 2 ? '0' + elem->ertek : elem-> ←
            ertek,
            melyseg);
        kiir (elem->bal_nulla);
        --melyseg;
    }
}

```

A program a következő fát rajzolta ki nekünk:

```

guzsik@guzsikd: ~/Documents/bhax/Welch/Binfa
File Edit View Search Terminal Help
guzsik@guzsikd:~/Documents/bhax/Welch/Binfa$ ./lzw <fa.txt
01111001001001000111
-----1(4)
-----1(3)
-----1(2)
-----0(3)
-----0(4)
-----0(5)
---/(1)
-----1(3)
-----0(2)
-----0(3)
melyseg=5
guzsik@guzsikd:~/Documents/bhax/Welch/Binfa$ 

```

A preorder bejárás a következőképpen fog alakulni. Először mindig a gyökeret dolgozzuk fel, majd a bal oldali részfat (ott is először a gyökeret) és végül a jobb oldali részfat (ott is a gyökeret).

A forráskódon a következő apró módosítást hajtjuk végre:

```

void
kiir (BINFA_PTR elem)
{
    if (elem != NULL)
    {
        ++melyseg;

```

```

    if (melyseg > max_melyseg)
        max_melyseg = melyseg;

    kiir (elem->jobb_egy);
    kiir (elem->bal_nulla);
    for (int i = 0; i < melyseg; ++i)
        printf ("---");
    printf ("%c(%d)\n", elem->ertek < 2 ? '0' + elem->ertek : elem->ertek <=
        , melyseg);

    --melyseg;
}
}

```

A program a következő fát rajzolta ki nekünk:

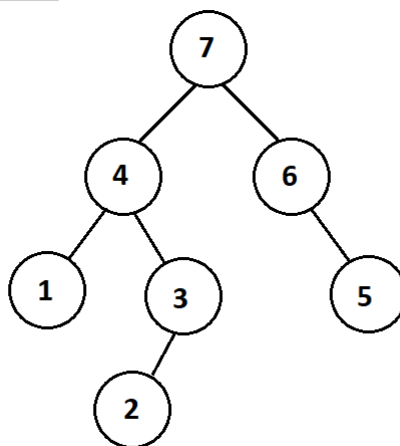
```

guzsik@guzsikd: ~/Documents/bhax/Welch/Binfapre
File Edit View Search Terminal Help
guzsik@guzsikd:~/Documents/bhax/Welch/Binfapre$ ./lzwpre <fa.txt
01111001001001000111

-----1(4)
-----1(3)
-----0(5)
-----0(4)
-----0(3)
-----1(2)
-----1(3)
-----0(3)
-----0(2)
---/(1)
melyseg=5
guzsik@guzsikd:~/Documents/bhax/Welch/Binfapre$ 

```

A postorder bejárásnál először mindig a bal oldali részfát (ott is először a bal részfát) majd a jobb oldali részfát (ott is a jobb részfát) dolgozzuk fel. A legvégére marad a gyökér.



6.3. ábra. Példa Postorder bejárásra:

A forráskódon a következő apró módosítást hajtjuk végre:

```
void
kiir (BINFA_PTR elem)
{
    if (elem != NULL)
    {
        ++melyseg;
        if (melyseg > max_melyseg)
            max_melyseg = melyseg;

        for (int i = 0; i < melyseg; ++i)
            printf ("---");
        printf ("%c(%d)\n", elem->ertek < 2 ? '0' + elem->ertek : elem->ertek ->
            , melyseg);
        kiir (elem->jobb_egy);
        kiir (elem->bal_nulla);
        --melyseg;
    }
}
```

A program a következő fát rajzolta ki nekünk:

```
guzsik@guzsikd: ~/Documents/bhax/Welch/Binfapost
File Edit View Search Terminal Help
guzsik@guzsikd:~/Documents/bhax/Welch/Binfapost$ ./lzwpost <fa.txt
01111001001001000111

---/(1)
-----1(2)
-----1(3)
-----1(4)
-----0(3)
-----0(4)
-----0(5)
-----0(2)
-----1(3)
-----0(3)
melyseg=5
guzsik@guzsikd:~/Documents/bhax/Welch/Binfapost$
```

6.4. Tag a gyökér

Az LZW algoritmust ültess át egy C++ osztályba, legyen egy Tree és egy beágyazott Node osztálya. A gyökér csomópont legyen kompozícióban a fával!

A bináris fa építését az újonnan létrehozott LZWTree osztályban szeretnénk megvalósítani. Ide kellene még beágyaznunk a fa csomópontjának (Node) a jellemzését is. Azért fogjuk beágyazni, mert egyelőre semmilyen különleges szerepet nem kap. Szimplán csak a fa részeként tekintünk rá.

```
#include <iostream>
```



```
class LZWTree
{
public:
    LZWTree (): fa(&gyoker){}

    ~LZWTree ()
    {
        szabadit (gyoker.egyenesGyermekek ());
        szabadit (gyoker.nullasGyermekek ());
    }

    void operator<<(char b)
    {
        if (b == '0')
        {
            if (!fa->nullasGyermekek ())
            {
                Node *uj = new Node ('0');
                fa->ujNullasGyermekek (uj);
                fa = &gyoker;
            }
            else
            {
                fa = fa->nullasGyermekek ();
            }
        }
        else
        {
            if (!fa->egyenesGyermekek ())
            {
                Node *uj = new Node ('1');
                fa->ujEgyenesGyermekek (uj);
                fa = &gyoker;
            }
            else
            {
                fa = fa->egyenesGyermekek ();
            }
        }
    }

    void kiir (void)
    {
        melyseg = 0;
        kiir (&gyoker);
    }

    void szabadit (void)
    {
        szabadit (gyoker.jobbEgy);
        szabadit (gyoker.balNulla);
    }
}
```

```
}
```

A fa konstruktora és destruktora után az LZWTree osztályon belül definiáljuk a balra eltoló bitshift operátor túlterhelését. Ez segíteni fog az inputról érkező karaktereket "beletolni" a LZWTree objektumba. Így fog felépülni a fa.

```
private:

class Node
{
public:
    Node (char b = '/') : betu (b), balNulla (0), jobbEgy (0) {};
    ~Node () {};
    Node *nullasGyermekek () {
        return balNulla;
    }
    Node *egyegyGyermekek ()
    {
        return jobbEgy;
    }
    void ujNullasGyermekek (Node * gy)
    {
        balNulla = gy;
    }
    void ujEgyegyGyermekek (Node * gy)
    {
        jobbEgy = gy;
    }

private:
    friend class LZWTree;
    char betu;
    Node *balNulla;
    Node *jobbEgy;
    Node (const Node &);
    Node & operator=(const Node &);
};
```

Amennyiben paraméter nélküli a Node konstruktor, akkor az alapértelmezett '/'-jellel fogja azt létrehozni. Egyébként a meghívó karakter kerül a betű helyére. A bal és jobb gyermekekre mutató mutatókat nullára állítjuk. Az aktuális Node mindig meg tudja mondani, hogy mi a bal illetve jobb gyermeke. Meg is mondhatjuk neki, hogy melyik csomópont legyen ezentúl az új gyermek. Barátjaként deklaráljuk az LZWTree osztályt, hogy az dolgozhasson a csomópontokkal.

```
Node gyoker;
Node *fa;
int melyseg;

LZWTree (const LZWTree &);
LZWTree & operator=(const LZWTree &);
```

```
void kiir (Node* elem)
{
    if (elem != NULL)
    {
        ++melyseg;
        kiir (elem->jobbEgy);

        for (int i = 0; i < melyseg; ++i)
            std::cout << "---";
        std::cout << elem->betu << "(" << melyseg - 1 << ")" << std::endl;
        kiir (elem->balNulla);
        --melyseg;
    }
}

void szabadit (Node * elem)
{
    if (elem != NULL)
    {
        szabadit (elem->jobbEgy);
        szabadit (elem->balNulla);
        delete elem;
    }
}

};
```

Mindig az aktuális csomóponttra mutatunk. A `kiir` függvénnyel jelenítjük meg magát a fát. Felszabadítjuk a két gyermeket, nehogy elfolyjon a memóriánk.

```
int
main ()
{
    char b;
    LZWTree binFa;

    while (std::cin >> b)
    {
        binFa << b;
    }

    binFa.kiir ();
    binFa.szabadit ();

    return 0;
}
```

Bitenként olvassuk a bemenetet, de a fát már karakterenként építjük fel.

Fordítás és futtatás után:

```
$ g++ tree.cpp -o tree
$ ./tree < teszt.txt
-----1 (3)
-----1 (2)
-----1 (1)
-----0 (2)
-----0 (3)
-----0 (4)
---/ (0)
-----1 (2)
-----0 (1)
-----0 (2)
```

Megoldás forrása:https://progater.blog.hu/2011/04/01/imadni_fogjak_a_c_t_egy_emberkent_tiszta_szivbol_2

Továbbá:https://progater.blog.hu/2011/04/14/egyutt_tamadjuk_meg

6.5. Mutató a gyökér

Írd át az előző forrást, hogy a gyökér csomópont ne kompozícióban, csak aggregációban legyen a fával!

Eddig a LZWTree-ben a fa gyökere mindig egy objektum volt. A `fa` mutatót kezdetben ráállítottuk és a gyökér referencia értékét adta vissza. Mostantól a gyökér is és a fa is egy-egy mutató lesz. Tehát nem a referenciát adjuk át a fa mutatónak, hanem magát a gyökér mutatót.

```
class LZWTree
{
public:
    LZWTree ()
    {
        gyoker = new Node();
        fa = gyoker;
    }

    ~LZWTree ()
    {
        szabadit (gyoker->egyesGyermekek ());
        szabadit (gyoker->nullasGyermekek ());
        delete gyoker;
    }
}
```

A konstruktorban a gyökeret új csomópontra mutató mutatóként hozzuk létre, a destruktorban pedig ugyanígy szabadítjuk fel. Az eddig a gyökér előtt álló referenciajeleket mindenhol kitörölgetjük.

```
Node *gyoker;
Node *fa;
int melyseg;
```

A csomópontban a védett tagok között is objektumként szerepelt eddig, ott is át kell írni pointerre.

Láassuk hogy fordul-e és fut-e a program a módosítások után?

```
$ g++ pointer.cpp -o pointer
$ ./pointer < teszt.txt
-----1 (3)
-----1 (2)
-----1 (1)
-----0 (2)
-----0 (3)
-----0 (4)
---/ (0)
-----1 (2)
-----0 (1)
-----0 (2)
```

6.6. Mozgató szemantika

Írj az előző programhoz mozgató konstruktort és értékadást, a mozgató konstruktor legyen a mozgató értékadásra alapozva!

A feladatot az [UDPROG közösség](#) repója segítségével készítettem el.

Mozgató konstruktor helyett itt egy másoló konstruktor:

```
LZWTree& operator= (LZWTree& copy) //másoló értékadás
{
    szabadit(gyoker->egyGyermek ()); //régi értéket törlöm
    szabadit(gyoker->nullasGyermek ());

    bejar(gyoker, copy.gyoker); //rekurzívan bejárom a fákat és ↵
        átmásolom az értékeket

    fa = gyoker; //mindkét fában visszaugrok a gyökérhez
    copy.fa = copy.gyoker;
}

void bejar (Node * masolat, Node * eredeti) //rekurzív famásolás, ↵
        másoló értékadáshoz
{
    if (eredeti != NULL) //ha létezik a másolandó fa
    {
        if ( !eredeti->nullasGyermek() ) //ha nem létezik az ↵
            eredeti nullasgyermek
        {
            masolat->ujNullasGyermek(NULL);
        }
        else //ha létezik az eredeti nullásgyermek
        {

```

```
        //létrehozni a masolat nullasgyermeket és meghívni újra a ↵  
        bejart  
        Node* uj = new Node ('0');  
        masolat->ujNullasGyermek (uj);  
        bejar(masolat->>nullasGyermek(), eredeti->>nullasGyermek());  
    }  
  
    if ( !eredeti->egyesGyermek() ) //ha nem létezik az eredeti ↵  
        egyesgyermeke  
    {  
        masolat->ujEgyesGyermek (NULL);  
    }  
    else //ha létezik az eredeti egyesgyermeke  
    {  
        //létrehozni a masolat egyesgyermeket és meghívni újra a ↵  
        bejart  
        Node *uj = new Node ('1');  
        masolat->ujEgyesGyermek (uj);  
        bejar(masolat->egyesGyermek(), eredeti->egyesGyermek());  
    }  
}  
else //ha nem létezik a másolandó fa  
{  
    masolat = NULL;  
}  
}
```

7. fejezet

Helló, Conway!

7.1. Hangyaszimulációk

Írj Qt C++-ban egy hangyaszimulációs programot, a forrásaidról utólag reverse engineering jelleggel készíts UML osztálydiagramot is!

Megoldás videó: <https://bhaxor.blog.hu/2018/10/10/myrmecologist>

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

7.2. Java életjáték

Írd meg Java-ban a John Horton Conway-féle életjátékot, valósítsa meg a sikló-kilövőt!

Életfolyamatok szimulálása

A Conway-féle sejtautomata szabályai zseniálisan egyszerűek. Az élettér egy négyzetháló. Minden cellában egy sejt élhet. Minden sejtet nyolc szomszédos cella vesz körül. Minden sejt a szomszédjainak létszámától függően életben marad, vagy meghal. Bizonyos esetekben egy üres cellában új sejt születik. Ha egy generációban egy sejtnek kettő vagy három élő szomszédja van, akkor a sejt élni fog a következő generációban is, minden más esetben a sejt kihal (túlnépesedés, vagy elszigeteltség miatt). Ha egy üres cellának pontosan három élő sejt van a szomszédjában, akkor ott új sejt születik. Az alakzatok viselkedése nagyon hasonlít élő szervezetek változásaihoz (szaporodás, kihalás, fejlődés, visszafejlődés, stb), ezért megilleti a "szimulációs játék" elismerő jelző - egy játék, amely utánozza a valós életfolyamatokat.

Röviden:

- Egy élő sejt meghal, szomszédok száma kisebb, mint 2 vagy nagyobb, mint 3
- Egy sejt életben marad, ha szomszédok száma 2 vagy 3
- Létre jön egy sejt ott, ahol a szomszédok száma pontosan 3

Sikló-kilövőről röviden:

Conway 50 dollárt ígért egy bizonyítottan végtelenül növekedő formáció kiindulási állapotáért. Ekkor az egyetemeken, bankokban, hivatalokban beindult a formációgyártás. Matematikus és laikus, fizikus és méla filzóf próbálkozott, hogy felállítsa azt a formációt, amelyik végtelen terjeszkedik, nekifeszülve a virtuális tér bitzónáinak.

A díjat egyébként a Massachusetts Egyetem Mesterséges Intelligencia Csoportjának két tagja nyerte. Egy valóban meglepő felfedezést tettek: találtak egy "sikló kilövőt"! Az induló alakzat "ágyúvá" alakul, amely először a 40-ik lépésben lő ki egy "siklót", majd ütemesen ismétlődve, minden további 30-ik periódusban egy-egy továbbit. Mivel minden "sikló" születésekor öt újabb sejt kerül a táblára, a népesség nyilvánvalóan korlátlanul növekszik.

A kód a [Javát tanítok](#) oldalán található Sejtautomata szimuláció alapján készült.

```
public class Sejtautomata extends java.awt.Frame implements Runnable {

    public static final boolean ÉLŐ = true;

    public static final boolean HALOTT = false;

    protected boolean [][][] rácsek = new boolean [2][][];

    protected boolean [][] rác;

    protected int rácIndex = 0;

    protected int cellaSzélesség = 20;
    protected int cellaMagasság = 20;

    protected int szélesség = 20;
    protected int magasság = 10;

    protected int várakozás = 10;

    private java.awt.Robot robot;
```

Boolean típusokban deklaráljuk, hogy egy sejt lehet élő vagy halott. A rácindex mutatja majd az aktuális rácset. Cellák szélessége és magassága pixelben megadva. A sebességet levettem egészen 10-re, hogy gyorsabban érjük el majd a káoszt.

```
public Sejtautomata(int szélesség, int magasság)
{
    this.szélesség = szélesség;
    this.magasság = magasság;

    rácsek[0] = new boolean[magasság][szélesség];
    rácsek[1] = new boolean[magasság][szélesség];
    rácIndex = 0;
    rác = rácsek[rácIndex];

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



```
        for(int j=0; j<rács[0].length; ++j)
            rács[i][j] = HALOTT;

        siklóKilövő(rács, 5, 60);

        addWindowListener(new java.awt.event.WindowAdapter()
        {
            public void windowClosing(java.awt.event.WindowEvent e)
            {
                setVisible(false);
                System.exit(0);
            }
        });
```

A kiinduló rácsmező minden sejtje halott lesz kezdetben. Ezekre fogunk alakzatokat helyezni. Ha az ablakot bezárjuk, a program kilép.

```
        addKeyListener(new java.awt.event.KeyAdapter()
        {
            public void keyPressed(java.awt.event.KeyEvent e)
            {
                if(e.getKeyCode() == java.awt.event.KeyEvent.VK_K)
                {
                    cellaSzélesség /= 2;
                    cellaMagasság /= 2;
                    setSize(Sejtautomata.this.szélesség*cellaSzélesség,
                        Sejtautomata.this.magasság*cellaMagasság);
                    validate();
                }
                else if(e.getKeyCode() == java.awt.event.KeyEvent.VK_N)
                {
                    cellaSzélesség *= 2;
                    cellaMagasság *= 2;
                    setSize(Sejtautomata.this.szélesség*cellaSzélesség,
                        Sejtautomata.this.magasság*cellaMagasság);
                    validate();
                }

                else if(e.getKeyCode() == java.awt.event.KeyEvent.VK_G)
                    várakozás /= 2;
                else if(e.getKeyCode() == java.awt.event.KeyEvent.VK_L)
                    várakozás *= 2;
                repaint();
            }
        });
```

A billentyűzetről érkező inputot figyeljük.

- "K" lenyomása esetén a kirajzolt cellák méretét felezzük.
- "N" lenyomása esetén a kirajzolt cellák méretét duplázzuk.
- "G" lenyomása esetén gyorsítunk.
- "L" lenyomása esetén lassítunk.

```
addMouseListener(new java.awt.event.MouseAdapter()
{
    public void mousePressed(java.awt.event.MouseEvent m)
    {
        int x = m.getX()/cellaSzélesség;
        int y = m.getY()/cellaMagasság;
        rácsok[rácsIndex][y][x] = !rácsok[rácsIndex][y][x];
        repaint();
    }
});

addMouseMotionListener(new java.awt.event.MouseMotionAdapter()
{
    public void mouseDragged(java.awt.event.MouseEvent m)
    {
        int x = m.getX()/cellaSzélesség;
        int y = m.getY()/cellaMagasság;
        rácsok[rácsIndex][y][x] = ÉLŐ;
        repaint();
    }
});
```

Az egérről érkező inputot figyeljük. Kattintással és egérhúzással jelölünk ki új sejteket.

```
cellaSzélesség = 10;
cellaMagasság = 10;

try
{
    robot = new java.awt.Robot(
        java.awt.GraphicsEnvironment.
            getLocalGraphicsEnvironment().
            getDefaultScreenDevice());
}
catch(java.awt.AWTException e)
{
    e.printStackTrace();
}

setTitle("Sejtautomata");
```

```
setResizable(false);
setSize(szélesség*cellaSzélesség,
        magasság*cellaMagasság);
setVisible(true);
```

A program indulásakor megjelenő ablak adatai. Az iteráció elindítása.

```
public void paint(java.awt.Graphics g)
{
    boolean [][] rács = rácsok[rácsIndex];

    for(int i=0; i<rács.length; ++i)
    {
        for(int j=0; j<rács[0].length; ++j)
        {
            if(rács[i][j] == ÉLŐ)
                g.setColor(java.awt.Color.BLACK);
            else
                g.setColor(java.awt.Color.WHITE);
            g.fillRect(j*cellaSzélesség, i*cellaMagasság,
                      cellaSzélesség, cellaMagasság);

            g.setColor(java.awt.Color.LIGHT_GRAY);
            g.drawRect(j*cellaSzélesség, i*cellaMagasság,
                      cellaSzélesség, cellaMagasság);
        }
    }
}
```

Kirajzoltatjuk az aktuális sejteket ábrázoló teret. ha élő a sejt akkor feketére színezzük, ha halott, akkor fehérre. A rácsokhoz szürke színt használunk.

```
public int szomszédokSzáma(boolean [][] rács,
                           int sor, int oszlop, boolean állapot)
{
    int állapotúSzomszéd = 0;

    for(int i=-1; i<2; ++i)
        for(int j=-1; j<2; ++j)

            if(!((i==0) && (j==0)))
            {
                int o = oszlop + j;
                if(o < 0)
                    o = szélesség-1;
                else if(o >= szélesség)
                    o = 0;
```

```
        int s = sor + i;
        if(s < 0)
            s = magasság-1;
        else if(s >= magasság)
            s = 0;

        if(rács[s][o] == állapot)
            ++állapotúSzomszéd;
    }

    return állapotúSzomszéd;
}
```

Az aktuális sejt nyolc szomszédját számoljuk végig, magát a sejtet kihagyjuk.

```
public void időFejlődés()
{

    boolean [][] rácsElőtte = rácsok[rácsIndex];
    boolean [][] rácsUtána = rácsok[(rácsIndex+1)%2];

    for(int i=0; i<rácsElőtte.length; ++i)
    {
        for(int j=0; j<rácsElőtte[0].length; ++j)
        {

            int élők = szomszédokSzama(rácsElőtte, i, j, ÉLŐ);

            if(rácsElőtte[i][j] == ÉLŐ)
            {

                if(élők==2 || élők==3)
                    rácsUtána[i][j] = ÉLŐ;
                else
                    rácsUtána[i][j] = HALOTT;
            }
            else
            {

                if(élők==3)
                    rácsUtána[i][j] = ÉLŐ;
                else
                    rácsUtána[i][j] = HALOTT;
            }
        }
    }

    rácsIndex = (rácsIndex+1)%2;
}
```

A sejtek életére vonatkozó (fentebb már említett) 4 szabályt vizsgáljuk itt végig. Ezzel számoljuk ki, hogy a következő körben az adott sejt élő vagy halott lesz.

```
public void run()
{
    while(true)
    {
        try
        {
            Thread.sleep(várakozás);
        }

        catch (InterruptedException e) {}

        időFejlődés();
        repaint();
    }
}
```

A sejtter fejlődése az idő függvényében.

```
public void sikló(boolean [][] rács, int x, int y)
{
    rács[y+ 0][x+ 2] = ÉLŐ;
    rács[y+ 1][x+ 1] = ÉLŐ;
    rács[y+ 2][x+ 1] = ÉLŐ;
    rács[y+ 2][x+ 2] = ÉLŐ;
    rács[y+ 2][x+ 3] = ÉLŐ;
}

public void siklóKilövő(boolean [][] rács, int x, int y)
{
    rács[y+ 6][x+ 0] = ÉLŐ;
    rács[y+ 6][x+ 1] = ÉLŐ;
    rács[y+ 7][x+ 0] = ÉLŐ;
    rács[y+ 7][x+ 1] = ÉLŐ;

    rács[y+ 3][x+ 13] = ÉLŐ;

    rács[y+ 4][x+ 12] = ÉLŐ;
    rács[y+ 4][x+ 14] = ÉLŐ;

    rács[y+ 5][x+ 11] = ÉLŐ;
    rács[y+ 5][x+ 15] = ÉLŐ;
    rács[y+ 5][x+ 16] = ÉLŐ;
    rács[y+ 5][x+ 25] = ÉLŐ;

    rács[y+ 6][x+ 11] = ÉLŐ;
    rács[y+ 6][x+ 15] = ÉLŐ;
```

```
rács[y+ 6][x+ 16] = ÉLŐ;  
rács[y+ 6][x+ 22] = ÉLŐ;  
rács[y+ 6][x+ 23] = ÉLŐ;  
rács[y+ 6][x+ 24] = ÉLŐ;  
rács[y+ 6][x+ 25] = ÉLŐ;  
  
rács[y+ 7][x+ 11] = ÉLŐ;  
rács[y+ 7][x+ 15] = ÉLŐ;  
rács[y+ 7][x+ 16] = ÉLŐ;  
rács[y+ 7][x+ 21] = ÉLŐ;  
rács[y+ 7][x+ 22] = ÉLŐ;  
rács[y+ 7][x+ 23] = ÉLŐ;  
rács[y+ 7][x+ 24] = ÉLŐ;  
  
rács[y+ 8][x+ 12] = ÉLŐ;  
rács[y+ 8][x+ 14] = ÉLŐ;  
rács[y+ 8][x+ 21] = ÉLŐ;  
rács[y+ 8][x+ 24] = ÉLŐ;  
rács[y+ 8][x+ 34] = ÉLŐ;  
rács[y+ 8][x+ 35] = ÉLŐ;  
  
rács[y+ 9][x+ 13] = ÉLŐ;  
rács[y+ 9][x+ 21] = ÉLŐ;  
rács[y+ 9][x+ 22] = ÉLŐ;  
rács[y+ 9][x+ 23] = ÉLŐ;  
rács[y+ 9][x+ 24] = ÉLŐ;  
rács[y+ 9][x+ 34] = ÉLŐ;  
rács[y+ 9][x+ 35] = ÉLŐ;  
  
rács[y+ 10][x+ 22] = ÉLŐ;  
rács[y+ 10][x+ 23] = ÉLŐ;  
rács[y+ 10][x+ 24] = ÉLŐ;  
rács[y+ 10][x+ 25] = ÉLŐ;  
  
rács[y+ 11][x+ 25] = ÉLŐ;
```

```
}
```

A glider és a glider-gun elhelyezése a sejttérben.

```
public void update(java.awt.Graphics g)  
{  
    paint(g);  
}  
  
public static void main(String[] args)  
{  
  
    new Sejtautomata(100, 75);  
}
```

```
}
```

Végül teszünk róla, hogy ne villogjon az ablak és egy 100 x 75-ös mérettel példányosítjuk a létrehozott objektumot.

Megoldás forrása: <https://www.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat/apb.html#id564903>

Javát tanítok: <https://www.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat/apb.html#id564903>

7.3. Qt C++ életjáték

Most Qt C++-ban!

A main.cpp felépítése:

```
#include <QApplication>
#include "sejtablak.h"

int main(int argc, char *argv[])
{
    QApplication a(argc, argv);
    SejtAblak w(100, 75);
    w.show();

    return a.exec();
}
```

A beincludált sejtablak.h:

```
#ifndef SEJTABLAK_H
#define SEJTABLAK_H

#include <QMainWindow>
#include <QPainter>
#include "sejtszal.h"

class SejtSzal;

class SejtAblak : public QMainWindow
{
    Q_OBJECT

public:
    SejtAblak(int szelesseg = 100, int magassag = 75, QWidget *parent = 0);
    ~SejtAblak();

    static const bool ELO = true;

    static const bool HALOTT = false;
    void vissza(int racsIndex);
}
```

```
protected:

    bool ***racsok;
    bool **racs;
    int racsIndex;
    int cellaSzelesseg;
    int cellaMagassag;
    int szelesseg;
    int magassag;
    void paintEvent (QPaintEvent*);
    void siklo(bool **racs, int x, int y);
    void sikloKilovo(bool **racs, int x, int y);

private:
    SejtSzal* életjatek;

};

#endif
```

Itt található az élő vagy halott sejtek boolean változója. A sejtter mérete. A rács többdimenziós tömbjeire mutató mutatók.

sejtablak.cpp:

```
#include "sejtablak.h"

SejtAblak::SejtAblak(int szelesseg, int magassag, QWidget *parent)
    : QMainWindow(parent)
{
    setWindowTitle("A John Horton Conway-féle életjáték");

    this->magassag = magassag;
    this->szelesseg = szelesseg;

    cellaSzelesseg = 6;
    cellaMagassag = 6;

    setFixedSize(QSize(szelesseg*cellaSzelesseg, magassag*cellaMagassag));

    racsok = new bool**[2];
    racsok[0] = new bool*[magassag];
    for(int i=0; i<magassag; ++i)
        racsok[0][i] = new bool [szelesseg];
    racsok[1] = new bool*[magassag];
    for(int i=0; i<magassag; ++i)
        racsok[1][i] = new bool [szelesseg];

    racsIndex = 0;
    racs = racsok[racsIndex];
```



```
for(int i=0; i<magassag; ++i)
    for(int j=0; j<szelesseg; ++j)
        racs[i][j] = HALOTT;

sikloKilovo(racs, 5, 60);

eletjatek = new SejtSzal(racsok, szelesseg, magassag, 120, this);
eletjatek->start();
}

void SejtAblak::paintEvent(QPaintEvent*) {
    QPainter qpainter(this);

    bool **racs = racsok[racsIndex];

    for(int i=0; i<magassag; ++i)
    {
        for(int j=0; j<szelesseg; ++j)
        {
            if(racs[i][j] == ELO)
                qpainter.fillRect(j*cellaSzelesseg, i*cellaMagassag,
                                   cellaSzelesseg, cellaMagassag, Qt::black) ←
                ;
            else
                qpainter.fillRect(j*cellaSzelesseg, i*cellaMagassag,
                                   cellaSzelesseg, cellaMagassag, Qt::white) ←
                ;
            qpainter.setPen(QPen(Qt::gray, 1));
            qpainter.drawRect(j*cellaSzelesseg, i*cellaMagassag,
                               cellaSzelesseg, cellaMagassag);
        }
    }

    qpainter.end();
}

SejtAblak::~SejtAblak()
{
    delete eletjatek;

    for(int i=0; i<magassag; ++i)
    {
        delete[] racsok[0][i];
        delete[] racsok[1][i];
    }
}
```

```
    }

    delete[] racsok[0];
    delete[] racsok[1];
    delete[] racsok;

}

void SejtAblak::vissza(int racsIndex)
{
    this->racsIndex = racsIndex;
    update();
}

void SejtAblak::siklo(bool **racs, int x, int y)
{
    racs[y+ 0][x+ 2] = ELO;
    racs[y+ 1][x+ 1] = ELO;
    racs[y+ 2][x+ 1] = ELO;
    racs[y+ 2][x+ 2] = ELO;
    racs[y+ 2][x+ 3] = ELO;

}

void SejtAblak::sikloKilovo(bool **racs, int x, int y)
{
    racs[y+ 6][x+ 0] = ELO;
    racs[y+ 6][x+ 1] = ELO;
    racs[y+ 7][x+ 0] = ELO;
    racs[y+ 7][x+ 1] = ELO;

    racs[y+ 3][x+ 13] = ELO;

    racs[y+ 4][x+ 12] = ELO;
    racs[y+ 4][x+ 14] = ELO;

    racs[y+ 5][x+ 11] = ELO;
    racs[y+ 5][x+ 15] = ELO;
    racs[y+ 5][x+ 16] = ELO;
    racs[y+ 5][x+ 25] = ELO;

    racs[y+ 6][x+ 11] = ELO;
    racs[y+ 6][x+ 15] = ELO;
    racs[y+ 6][x+ 16] = ELO;
    racs[y+ 6][x+ 22] = ELO;
    racs[y+ 6][x+ 23] = ELO;
    racs[y+ 6][x+ 24] = ELO;
```

```
    racs[y+ 6][x+ 25] = ELO;

    racs[y+ 7][x+ 11] = ELO;
    racs[y+ 7][x+ 15] = ELO;
    racs[y+ 7][x+ 16] = ELO;
    racs[y+ 7][x+ 21] = ELO;
    racs[y+ 7][x+ 22] = ELO;
    racs[y+ 7][x+ 23] = ELO;
    racs[y+ 7][x+ 24] = ELO;

    racs[y+ 8][x+ 12] = ELO;
    racs[y+ 8][x+ 14] = ELO;
    racs[y+ 8][x+ 21] = ELO;
    racs[y+ 8][x+ 24] = ELO;
    racs[y+ 8][x+ 34] = ELO;
    racs[y+ 8][x+ 35] = ELO;

    racs[y+ 9][x+ 13] = ELO;
    racs[y+ 9][x+ 21] = ELO;
    racs[y+ 9][x+ 22] = ELO;
    racs[y+ 9][x+ 23] = ELO;
    racs[y+ 9][x+ 24] = ELO;
    racs[y+ 9][x+ 34] = ELO;
    racs[y+ 9][x+ 35] = ELO;

    racs[y+ 10][x+ 22] = ELO;
    racs[y+ 10][x+ 23] = ELO;
    racs[y+ 10][x+ 24] = ELO;
    racs[y+ 10][x+ 25] = ELO;

    racs[y+ 11][x+ 25] = ELO;

}
```

sejtszal.h:

```
#ifndef SEJTSZAL_H
#define SEJTSZAL_H

#include <QThread>
#include "sejtablak.h"

class SejtAblak;

class SejtSzal : public QThread
{
    Q_OBJECT

public:
    SejtSzal(bool ***racso, int szelesseg, int magassag,
             int varakozas, SejtAblak *sejtAblak);
```

```
~SejtSzal();
void run();

protected:
    bool ***racsok;
    int szelesseg, magassag;
    int racsIndex;
    int varakozas;
    void idoFejlodes();
    int szomszedokSzama(bool **racs,
                        int sor, int oszlop, bool allapot);
    SejtAblak* sejtAblak;

};

#endif
```

sejtszal.cpp:

```
#include "sejtszal.h"

SejtSzal::SejtSzal(bool ***racsok, int szelesseg, int magassag, int ←
    varakozas, SejtAblak *sejtAblak)
{
    this->racsok = racsok;
    this->szelesseg = szelesseg;
    this->magassag = magassag;
    this->varakozas = varakozas;
    this->sejtAblak = sejtAblak;

    racsIndex = 0;
}

int SejtSzal::szomszedokSzama(bool **racs,
                              int sor, int oszlop, bool allapot)
{
    int allapotuSzomszed = 0;

    for(int i=-1; i<2; ++i)
        for(int j=-1; j<2; ++j)

            if(!((i==0) && (j==0)))
            {

                int o = oszlop + j;
                if(o < 0)
                    o = szelesseg-1;
                else if(o >= szelesseg)
                    o = 0;

                int s = sor + i;
```

```
        if(s < 0)
            s = magassag-1;
        else if(s >= magassag)
            s = 0;

        if(racs[s][0] == allapot)
            ++allapotuSzomszed;
    }

    return allapotuSzomszed;
}

void SejtSzal::idoFejlodes()
{

    bool **racsElotte = racsok[racsIndex];
    bool **racsUtana = racsok[(racsIndex+1)%2];

    for(int i=0; i<magassag; ++i)
    {
        for(int j=0; j<szelesseg; ++j)
        {

            int elok = szomszedokSzama(racsElotte, i, j, SejtAblak::ELO);

            if(racsElotte[i][j] == SejtAblak::ELO)
            {

                if(elok==2 || elok==3)
                    racsUtana[i][j] = SejtAblak::ELO;
                else
                    racsUtana[i][j] = SejtAblak::HALOTT;
            }
            else
            {

                if(elok==3)
                    racsUtana[i][j] = SejtAblak::ELO;
                else
                    racsUtana[i][j] = SejtAblak::HALOTT;
            }
        }
    }
    racsIndex = (racsIndex+1)%2;
}

void SejtSzal::run()
{
```

```
while(true) {  
    QThread::msleep(varakozas);  
    idoFejlodes();  
    sejtAblak->vissza(racsIndex);  
}  
  
}  
  
SejtSzal::~~SejtSzal()  
{  
}
```

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

7.4. BrainB Benchmark

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

8. fejezet

Helló, Schwarzenegger!

8.1. Szoftmax Py MNIST

Python

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

8.2. Szoftmax R MNIST

R

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

8.3. Mély MNIST

Python

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

8.4. Deep dream

Keras

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

8.5. Minecraft-MALMÖ

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

DRAFT

9. fejezet

Helló, Chaitin!

9.1. Iteratív és rekurzív faktoriális Lisp-ben

Megoldás videó:

Megoldás forrása:

9.2. Weizenbaum Eliza programja

Éleszd fel Weizenbaum Eliza programját!

Megoldás videó:

Megoldás forrása:

9.3. Gimp Scheme Script-fu: króm effekt

Írj olyan script-fu kiterjesztést a GIMP programhoz, amely megvalósítja a króm effektet egy bemenő szövegre!

Megoldás videó: https://youtu.be/OKdAkI_c7Sc

Megoldás forrása: https://gitlab.com/nbatfai/bhax/tree/master/attention_raising/GIMP_Lisp/Chrome

Tanulságok, tapasztalatok, magyarázat...

9.4. Gimp Scheme Script-fu: név mandala

Írj olyan script-fu kiterjesztést a GIMP programhoz, amely név-mandalát készít a bemenő szövegből!

Megoldás videó: https://bhaxor.blog.hu/2019/01/10/a_gimp_lisp_hackelese_a_scheme_programozasi_nyelv

Megoldás forrása: https://gitlab.com/nbatfai/bhax/tree/master/attention_raising/GIMP_Lisp/Mandala

Tanulságok, tapasztalatok, magyarázat...

9.5. Lambda

Hasonlítsd össze a következő programokat!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

9.6. Omega

Megoldás videó:

Megoldás forrása:

DRAFT

10. fejezet

Helló, Gutenberg!

10.1. Juhász István, Magasszintű programozási nyelvek I.

1. Alapfogalmak

A kialakult programozási nyelvek három szintre oszthatóak. Legalul találhatók (a hardverhez legközelebb) a gépi nyelvek. Őket követik az assembly szintű nyelvek. Majd legfelül a (programozóhoz/felhasználóhoz legközelebb álló, "emberi fogyasztásra is alkalmas") magas szintű programozási nyelvek. Egy magas szintű nyelven létrehozott programot forrásprogramnak/forráskódnak nevezhetünk. Egy program írása során be kell tartani a szintaktikai (a nyelvre vonatkozó formalitások és nyelvtan) valamint szemantikai (tartalom, logikai összetettség, értelmezhetőség) szabályokat is. Az elkészült forrásprogramot valamilyen módszerrel gépi kóddá kell alakítani, hogy a processzor értelmezni és futtatni tudja. Erre lesz segítségünkre kétféle technika, a fordítóprogram és az interpreter.

A fordítóprogram négy lépésben alakítja át forrásprogramot gépi kóddá. Először végrehajt egy lexikális elemzést, amely kiszűri az esetleges szintaktikai hibákat. Ellenőrzi hogy szemantikailag helyes-e a program, majd megindul a kódgenerálás. Az interpreteres megvalósítás annyiban tér el a fordítóprogramtól, hogy itt nem jön létre a folyamat legvégén a tárgyprogram generálása. A forrásprogramot utasításoként, azonnal értelmezi majd végre is hajtja. A kívánt program egyből lefut.

A programozási nyelvek saját hivatkozási nyelvei magukba foglalják az adott nyelvhez definiált szintaktikai és szemantikai szabályokat. Sajnos a programok implementációk közötti hordozhatósága a mai napig nem lett megoldva. Tehát egy X processzoron és Y operációs rendszeren megírt programot egy másik, Z processzoron és Q operációs rendszeren futtatva eltérő eredményt kaphatunk.

2. Programnyelvek osztályozása: Imperatív és Dekleratív nyelvek

2.1 Egyik nagy csoport az imperatív nyelvek csoportja, algoritmikus nyelv, ahol a kód nem más, mint utasítások sorozata. Legtöbbször változókkal végrehajtott műveletek és utasítások. Két alcsoportja: eljárásorientált nyelvek és objektumorientált nyelvek.

2.2 Másik csoportja a deklaratív nyelvek. Ezek nem algoritmikus nyelvek. A programozó itt csak egy problémát ad meg, a megoldás a nyelv implementációiban található. Itt nincs lehetőség memóriaműveletekre. Két alcsoportja: funkcionális és logikai nyelvek.

3. Lexikális egységek

A lexikális elemzés során, a fordító által felismert egységek: többkarakteres szimbólum, szimbolikus név, címke, megjegyzés, literál.

3.1 Többkarakteres szimbólumok, csak az adott nyelvhez köthetőek.

3.2 Szimbolikus nevek.

3.2.1 Azonosító: Betűvel kezdődő, betűvel vagy számjeggyel folytatódó karaktersorozat. A programozó saját eszközeit nevezi meg velük és hivatkozik rájuk.

3.2.2 Kulcsszó: Az adott programnyelvben kötött, foglalt szavak, melyhez maga a nyelv társít jelentést. A programozó által nem változtatható meg.

3.2.3 Standard azonosító: Az adott programnyelvben kötött, foglalt szavak, melyhez maga a nyelv társít jelentést. De ezeket a programozó megváltoztathatja. Pl.: függvények nevei.

3.3 Címke: speciális karaktersorozat, ami utasítások jelölésére szolgál. Lehet előjelnélküli egész, vagy azonosító.

3.4 Megjegyzés: a fordító által teljesen ignorált szövegrész a programon belül, amely csupán a program megértését teszi könnyebbé más programozók/programot olvasók számára.

3.5 Literál (konstans): típus + érték komponensekből felépülő fix érték.

4. Adattípusok

Egy absztrakt eszköz a programozásban, amit a tartománya, vele végezhető műveletek és a reprezentációja határoz meg. A tartomány a programozási eszköz által felvehető értékeket tartalmazza. A műveletek végrehajthatóak a tartomány elemein. A reprezentáció a típusok tárban való megjelenését határozza meg. Minden nyelv rendelkezik standard típusokkal. Saját típust is létrehozhatunk, még hozzá úgy, hogy megadjuk annak tartományát, műveleteit és a reprezentációját. Az adattípusok két nagy csoportja, a skalár és a strukturált.

A skalár (egyszerű) adattípus atomi értékeket tartalmaz, nyelvi eszközökkel tovább nem bontható. A strukturált (összetett) adattípusok elemei maguk is típussal rendelkeznek. egy-egy értékcsoporthat képviselnek. Explicit módon kell őket megadni.

4.1 Egyszerű típusok

- Egész típus (short, int, long), ábrázolásuk fixpontos, méretük az ábrázoláshoz szükséges bájtok számában tér el.(numerikus típus)
- Valós típus (float, double), ábrázolásuk lebegőpontos. (numerikus típus)
- Karakteres (karakterlánc, string), ábrázolásuk karakteres. Karakterenként egy-két bájt.
- Logikai típus (boolean), ábrázolásuk logikai (igaz-hamis).
- Felsorolásos típus, saját típusként kell definiálni.

4.2 Összetett típusok

A tömb típus statikus (a tartomány elemeinek száma azonos) és homogén (az elemek azonos típusúak). A tömböt meghatározza: dimenzióinak száma, indexkészletének típusa és tartománya, és elemeinek a típusa. (A C nyelv nem ismeri a többdimenziós tömböt. De létrehozható egydimenziós tömbökből (mint elemekből) felépített egydimenziós tömb.)

Egy tömb elemei skalár vagy összetett típusúak lehetnek. Egy tömb indexelése egész típussal történhet. Indextartomány megadásánál a tömbben lévő elemek darabszámát kell megadni. Az alsó és felső határt konstans kifejezésekkel kell megadni.

A rekord típus: a rekord megjelenése típus szinten, minden esetben heterogén. A rekordon belüli elemeket mezőnek nevezzük. Minden mezőnek saját neve és típusa van. C-ben a rekord típus statikus (a mezők száma értékcsoporthoz tartozó).

4.3 Mutató típus

Olyan egyszerű típus, aminek elemei tárcímek. indirekt címzésre használható. Speciális eleme a NULL mutató, amely nem mutat sehova.

5. Nevesített konstans

A nevesített konstansnak három komponense van: neve, típusa, értéke. Egy konstanst használat előtt mindig deklarálni kell. A programban mindig nevével jelenik meg és értékét jelenti. Igyekezünk "beszélő" nevet választani neki. Névváltoztatás esetén elég a deklarációban megváltoztatni.

6. A változó

A változónak négy komponense van: név, attribútum, cím, érték. A név, mint azonosító, mindig ezzel jelenik meg a programkódban. Az attribútum meghatározza a változó viselkedését programfuttatás közben. Deklaráció fajtái: explicit, implicit, automatikus. Egy változó élettartama az az időszak, amíg a változó rendelkezik címkomponenssel. A változóhoz címet rendelhetünk: statikus, dinamikus vagy programozó által vezérelt tárhelyszámmal (abszolút vagy relatív címmel).

7. Alapelemek

Az aritmetikai típusok egyszerűek elemeivel műveletek végezhetők. Integrális típusok: egész, karakter, felsorolás. Valós típusok: float, double, long double.

Származtatott típusok: tömb, függvény, mutató, struktúra, union, void típus.

Explicit deklarációs utasítás: [CONST] típusleírás eszközazonosítás [= kifejezés]

- azonosító : típusleírás típusú változó
- (azonosító): típusleírás függvényt címző mutató típusú változó
- *azonosító : típusleírás típusú eszközt címző mutató típusú változó
- azonosító(): típusleírás visszatérési típusú függvény
- azonosító[]: típusleírás típusú elemeket tartalmazó tömb típusú változó
- Saját típus definiálása: `typedef típusleírás név [, típusleírás név]...;`
- Struktúra deklarálása: `STRUCT [struktúratispús_név] {mező_deklarációk} [változólista]`
- Union deklarálása: `UNION [uniontípus_név] {mező_deklarációk} [változólista];`

8. Kifejezések

Két komponensű (érték és típus komponens) szintaktikai eszközök. operandusokból (értékeket képvisel), operátorokból (műveleti jeleket képvisel) és kerek zárójelekből (műveleti sorrendet határoz meg) áll. Az operátorok műveletvégzéséhez szükséges operandusok számától függően beszélhetünk egyoperandusú (unáris), kétoperandusú (bináris), vagy háromoperandusú (ternáris) operátorokról.

Kétoperandusú operátoroknál az operandusok és az operátor sorrendje lehet:

- prefix (az operátor az operandusok előtt áll)
- infix (az operátor az operandusok között áll)
- postfix (az operátor az operandusok mögött áll)

A fordítóprogramok az infix kifejezésekből postfix kifejezéseket állítanak elő, majd a műveletvégzés aszerint történik.

Vannak típusegyenértékűséget és vannak a típuskényszerítést használó nyelvek. A C a numerikus típusoknál megengedi a típuskényszerítést. itt beszélhetünk bővítésről és szűkítésről. Bővítésnél az átalakítás értékvesztés nélkül végrehajtható. A szűkítés ennek az ellenkezője, itt az átalakításnál értékcsökkentés, esetleg kerekítés történhet. Konstans kifejezés az a kifejezés, aminek értéke fordítási időben eldől.

A C kifejezésorientált nyelv. A típuskényszerítés elvét vallja.

Precedenciatablázat a C nyelvben (fentről-lefelé erősebbtől-gyengébb felé haladva):

Függvényoperátor és műveleti sorrendet változtató zárójel	()
Tömboperátor	[]
Minősítő operátor	.
Mutatóval minősítő operátor (balról jobbra kötnek)	->

csillag/mutató típusú operátor	*
Tárcímet megadó operátor	&
Plusz előjel	+
Mínusz előjel	-
Ha az operandus értéke nem nulla, akkor az eredmény nulla, egyébként 1	!
Komplement operátor	~
Operandus értékét egyel növelő	++
Operandus értékét egyel csökkentő	--
A típus ábrázolási hosszát adja bájtban (jobbról balra kötnek)	sizeof (típus)

Szorzás	*
Osztás	/
Maradékképzés (balról jobbra kötnek)	%

Összeadás	+
Kivonás (balról jobbra kötnek)	-

Jobbra léptető bitshift	>>
Balra léptető bitshift (balról jobbra kötnek)	<<

Kisebb	<
Nagyobb	>
Kisebb vagy egyenlő	<=

Nagyobb vagy egyenlő (balról jobbra kötnek)	>=

Egyenlő	==
Nem egyenlő (balról jobbra kötnek)	!=

Bitenkénti és (balról jobbra köt)	&

Bitenkénti kizáró vagy (balról jobbra köt)	^

Bitenkénti vagy (balról jobbra köt)	

Logikai és (balról jobbra köt)	&&

Bitenkénti vagy (balról jobbra köt)	

Háromoperandusú operátor (balról jobbra köt)	?:

Értékadó operátorok	= += -= *= /= %= >>= <<= &= ^= =
(jobbról balra kötnek)	

9. Utasítások

Utasítások alkotják az eljárásorientált nyelveken megírt programok azon egységeit, amelyekkel az algoritmusok egyes lépéseit megadhatjuk, illetve a fordítóprogram ezek segítségével generálja a tárgyprogramot. Két csoportja van: a deklarációs és a végrehajtható utasítások. A deklarációs utasítások mögött nem áll tárgykód. Ezek a fordítóprogramnak szólnak. A fordítóprogram végrehajtható utasításokból generálja a tárgykódot.

9.1. Az értékadó utasítás feladata beállítani vagy módosítani a változók értékkomponensét.

9.2. Az üres utasítás hatására a processzor egy üres gépi utasítást hajt végre. A két utasítás-végjel között nem áll semmi.

9.3. Az ugró utasítás segítségével a program egy adott pontjáról egy adott címkével ellátott végrehajtható utasításra adhatjuk át a vezérlést.

9.4. A kétirányú elágaztató utasítás arra szolgál, hogy a program egy adott pontján két tevékenység közül válasszunk, illetve, hogy az adott tevékenységet végrehajtsuk-e vagy sem.

9.5. A többirányú elágaztató utasítás arra szolgál, hogy a program egy adott pontján egymást kölcsönösen kizáró akárhány tevékenység közül egyet végrehajtsunk. C-ben erre használható a switch utasítás.

9.6. A ciklusszervező utasítások lehetővé teszik, hogy a program egy adott pontján egy bizonyos tevékenységet akárhányszor megismételjünk.

9.7. Feltételes ciklusnál az ismétlődést egy feltétel igaz vagy hamis értéke szabályozza. A feltétel maga vagy a fejben vagy a végben szerepel. Kezdőfeltételes ciklusnál kiértékelődik a feltétel. Ha igaz akkor végrehajtódik a ciklusmag. Végfeltételes ciklusnál először végrehajtódik a mag, majd ezután értékelődik ki a feltétel.

9.8. Előírt lépésszámú ciklusnál az ismétlődésre vonatkozó információk a fejben vannak. Minden esetben van egy ciklusváltozó. A változó által felvett értékekre fut le a ciklusmag. Az előírt lépésszámú ciklus lehet előletesztelő, vagy hátulatesztelő.

9.9. A felsorolós ciklus egy olyan előírt lépésszámú ciklus, aminek van ciklusváltozója és a ciklusváltozó által felvett érték mellett lefut a mag.

9.10. A végtelen ciklus az a ciklusfajta, ahol sem a fejben, sem a végben nincs információ az ismétlődésre vonatkozóan.

9.11. Az összetett ciklus ciklusfejében tetszőlegesen sok ismétlődésre vonatkozó információ sorolható föl.

9.12. Vezérlő utasítások: A CONTINUE, a ciklusmag hátralévő utasításait nem hajtja végre, hanem az ismétlődés feltételei szerint vagy újabb cikluslépésbe kezd, vagy befejezi a ciklust. A BREAK a ciklust szabályosan befejezi és kilép a többszörös elágaztató utasításból. A RETURN szabályosan befejezi a függvényt és visszaadja a vezérlést a hívónak.

10. A programok szerkezete

Egyben kell-e lefordítanunk a program teljes szövegét, vagy fel kell osztanunk önállóan fordítható részekre?

- Van olyan programnyelv melyben a program önálló részekből áll, ezek akár külön is fordíthatóak. Nem strukturáltak.
- Vannak nyelvek, ahol a program csak egy nagy egységként fordítható. A programegységek nem függetlenek.
- Létezhet az első kettő kombinációja is. A nyelvben van független programegység, de strukturával rendelkezik.

Az eljárásorientált nyelvek egységei: alprogram, blokk, csomag, task.

10.1. Alprogramok

Az eljárásorientált nyelvek paradigmáit meghatározza. Újrafelhasználható, ha a program különböző részein a programrész megismétlődik. Egyszer írjuk meg, majd később hivatkozunk rá, újra meghívjuk. Az alprogram felépül fejből (specifikáció), törzsből (implementáció), végből. Négy komponense van: név, formális paraméter lista, törzs, környezet. A név a fejben szereplő azonosító. A formális paraméterlistában a paraméterek nevei és a futás közben szabályozó információik szerepelnek. Ha a paraméter lista üres, akkor paraméter nélküli az alprogram. A törzsben kapnak helyet a deklarációs és végrehajtó utasítások. Az alprogramon belül deklarált eszközök és nevek az alprogram lokális eszközei és nevei. Ezek az alprogramon kívülről láthatatlanok. Ellenkezői a globális nevek, amiket az alprogramon kívül deklarálunk. A globális változók együttese a környezet. A függvény egyetlen értéket meghatározó alprogram. A függvény neve mindig megadja a visszatérési értékét is. Az eljárás akkor fejeződik be szabályosan, ha elérjük a végét, vagy külön utasítást adunk ki a befejezésre. Minden nyelvben minden programban lennie kell egy főprogramnak. Ez felügyeli az összes többi alprogram működését. A főprogram szabályos lefutása után a vezérlés visszakérül az operációs rendszerhez.

10.2. Hívási lánc , Rekurzio

A hívási lánc, ha egy programegység meghív egy másik programegységet, majd az is meghív egy újabb programegységet, stb. A hívási lánc első eleme mindig a főprogram, minden tag aktív, de csak a legutólag meghívott egység fog működni. Az aktív program újbóli meghívását rekurzióknak hívjuk. Közvetlen rekurzió esetén az alprogram önmagát hívja meg. Közvetlen rekurzió esetén már korábban meghívott alprogram kerül meghívásra.

10.3. A blokk

Csak egy programegység belsejében állhat. Van kezdete, törzse, vége. A törzsben vannak a végrehajtó utasítások. A blokk kezdete előtt álló címke a blokk neve. Ahhoz hogy a blokk aktív legyen meg kell várni, hogy rá kerüljön a vezérlés, vagy GOTO utasítással a kezdetére ugorhatunk. Ha a blokk eléri a végét, vagy GOTO-val kiugrunk, akkor a blokk működése befejeződik.

11. Paraméterkiértékelés

A paraméterkiértékelés nem más, mint egy olyan folyamat, ahol a formális és aktuális paraméterek egymáshoz rendelődnek, egy alprogram hívásánál. A paraméterátadás kommunikációját meghatározó információk jönnek létre. Elsődleges és egyedi lesz a formális paraméterlista. Az aktuális paraméterlisták száma az alprogram hívásainak számával egyenlő. Mindig a formálishoz rendeljük az aktuálisat. Beszélhetünk sorrendi vagy név szerinti kötésről.

Sorrendi kötés: Az aktuális paraméterek a felsorolás sorrendjében lesznek hozzárendelve a formális paraméterekhez.

Név szerinti kötés: itt a formális paraméter nevét adjuk meg és mellette az aktuális paramétert, valamilyen szintaktikával. A formális paraméterek sorrendje itt nem számít.

Ha a formális paraméterek száma rögzített, akkor az aktuális paraméterek számának ezzel megegyezőnek kell lennie, VAGY az aktuális paraméterek száma lehet kevesebb is (csak érték szerinti paraméterátadásnál), ekkor amihez nem tartozik aktuális paraméter, majd alapértelmezett módon lesz érték rendelve. Ha a formális paraméterek száma tetszőleges, akkor az aktuális paraméterek száma is tetszőleges lesz. Az aktuális paraméter típusa kovertálható a formális paraméter típusára.

12. Paraméterátadás

Ez az alprogramok és programegységek közötti kommunikáció. Hívó és hívott részből áll. Létrejöhet érték, cím, eredmény, érték-eredmény, név vagy szöveg szerint.

Érték szerinti paraméterátadásnál a formális paraméterek címkomponenssel (hívott oldal), az aktuális paraméterek értékkomponenssel rendelkeznek (hívó oldal). Az alprogram a formális paraméter kezdőértékével dolgozik. Az információ a hívótól a hívott felé áramlik. Az aktuális paraméter kifejezés.

Cím szerinti paraméterátadásnál formális paraméter nem, az aktuális paraméter viszont rendelkezik címkomponenssel. Az alprogram a hívó területen fog dolgozni. Kétirányú információáramlás, az alprogram átvehet és írhat értéket a hívó területre. Az aktuális paraméter változó.

Eredmény szerinti paraméterátadás aza aktuális paraméter rendelkezik címkomponenssel, a formális paraméter rendelkezik címkomponenssel a hívott területen. Az alprogram a saját területén dolgozik és nem használja az aktuális paraméter címét. A működés végén átmásolja a formális paraméter értékét a címkomponensére. Kommunikáció iránya a hívottól a hívó felé. Az aktuális paraméter változó.

Érték-eredmény szerinti paraméterátadás esetén a formális paraméternek van címkomponense, az aktuális paraméternek van érték és címkomponense is. Az aktuális paraméter értéke és címe a hívotthoz kerül. Az alprogram saját területén dolgozik, majd az eredmény az aktuális paraméter címére másolódik. Kétirányú kommunikáció. Az aktuális paraméter változó.

Név szerinti paraméterátadásnál az aktuális paraméter egy szimbólumsorozat. A szimbólumsorozat felülírja az alprogram szövegében előforduló formális paraméter minden előfordulását, majd ezután kezdi meg a futást az alprogram.

Szöveg szerinti paraméterátadás annyiban különbözik a név szerintitől, hogy a formális paraméter felülírása a formális paraméter első előfordulása után következik be, futás közben.

Formális paraméterek három csoportja: input paraméter (az alprogram információt kap a hívótól), output paraméter (a hívott információt ad át a hívónak) és input-output paraméter (kétirányú információátadás).

10.2. Programozás bevezetés

[KERNIGHANRITCHIE]

Vezérlési szerkezetek

A vezérlési szerkezetek a számítások végrehajtásának sorrendjét határozzák meg. A kifejezések utasítássá válnak, ha pontosvessző követi őket. A kapcsos zárójelek felhasználásával a deklarációkat és utasításokat egyetlen blokkba foghatjuk össze.

Az if-else utasítással döntést, választást írunk le de az else rész nem kötelező. A gép a kifejezés kiértékelése után, ha annak értéke igaz akkor az 1. utasítást, ha értéke hamis, akkor a 2. utasítást hajtja végre. A gép sorban kiértékeli a kifejezéseket. Ha egy kifejezés igaz, akkor a hozzá tartozó utasítást a gép végrehajtja, ezzel a vizsgálat lezárul. Else-if esetén, ha a vezérlés ide kerül, egyetlen korábbi feltétel sem teljesült. Néha ilyenkor semmit sem kell csinálni, így a záró else utasítás elhagyható. A switch utasítás a többirányú programelágaztatásnál használható. Kifejezések értékét hasonlítja össze az állandó értékekkel és ennek megfelelő ugrást hajt végre. A switch kiértékeli a zárójelek közötti kifejezést és összehasonlítja az összes case értékével. A default case-re akkor kerül a vezérlés, ha a többi case egyike sem teljesül. A break utasítás hatására a vezérlés azonnal kilép a switchből. Ugyancsak break utasítással lehet kilépni a while, for és do ciklusokból is. A for (kifejezés1; kifejezés2; kifejezés3) {utasítás;} átírható while ciklusra. kifejezés1 while (kifejezés2){utasítás kifejezés3;} Végtelen ciklusból return vagy break paranccsal lehet kugrani. A harmadik C-beli ciklusfajta, a do-while, a vizsgálatot a ciklus végén, a ciklustörzs végrehajtása után végzi el és a törzs legalább egyszer mindenképpen végrehajtódik. A break utasítással a vizsgálat előtt is ki lehet ugrani a for, while és do ciklusokból, csakúgy, mint a switch-ből. A break utasítás hatására a vezérlés azonnal kilép a legbelső zárt ciklusból. A continue utasítás a continue-t tartalmazó ciklus (for, while, do) következő iterációját kezdi meg.

Megoldás videó: <https://youtu.be/zmfT9miB-jY>

10.3. Programozás

[BMECPP]

A C nyelv nem objektumorientált újdonságai (1-16. oldal)

A C++ nyelv, elődjének, a C nyelvnek a továbbfejlesztése. C-ben üres paraméterlistával definiált függvény tetszőleges számú paraméterrel hívható, míg C++-ben az üres paraméterlista ugyanaz, mintha void paramétert adtunk volna meg. A C++ nyelven került bevezetésre a bool típus, ami true vagy false értéket vehet fel. Alkalmazása olvashatóbb kódot eredményez. Beépített típus lett a wchar_t Unicode karakterek.

A C++-ban minden olyan helyen állhat deklaráció, ahol utasítás is állhat. A függvényeket nevük és argumentumlistájuk azonosít együttesen, így létrehozhatunk azonos nevű, de eltérő argumentumlistával rendelkező függvényeket. (Függvény túlterhelés) A függvények argumentumainak megadhatunk alapértelmezett értékeket és így a függvény a megadott argumentum alapértelmezett értékével kerül meghívásra. Az alapértelmezett argumentumokat a függvénydeklarációnál érdemes megadni. De vigyázzunk, mert alapértelmezett argumentumok és függvénynév-túlterhelés egyidejű alkalmazása fordítási hibát okozhat.

C++-ban a paraméterek átadása referenciatípussal történik. A referenciát inicializálnunk kell a referencia típusának megfelelő változóval. Közvetlenül a függvény visszatérése után a függvényparaméterek és a lokális változók felszabadulnak. Ezekre nem szabad referenciával (sem pointerrel) visszatérni, mert érvénytelen memóriaterületen elhelyezkedő (korábban felszabadított) változóra fogunk hivatkozni. Nagyméretű argumentumok (pl. struktúrák) esetén teljesítménynövekedést érhetünk el, ha csak az argumentumok címér adjuk át.

Az objektumorientáltság alapelvei (17-59. oldal)

Az újabb és újabb feladatok bonyolultságával a programok bonyolultsága is nő. Az objektumorientált programozás a 90-es évekre terjedt el igazán. Fő alapelve az egységbezárás. Osztálynak nevezzük az egységbe záró struktúrát. Az osztály példányai az objektumok. Az osztály és az objektum fogalma egységbe zárja a tulajdonságokat és a rajtuk elvégzett műveleteket. Adatrejtésnek hívjuk az objektum azon mechanizmusát, amely biztosítja, hogy a program többi része nem férhessen hozzá az objektum belsejéhez. Az általánosításnak (specializációnak) nevezett kapcsolattípusban a speciálisabb osztály rendelkezhet az általánosabb osztály tulajdonságaival, vagy műveleteivel. Ezt öröklésnek nevezzük. Egy másik tulajdonság a helyettesíthetőség, azaz egy speciális osztály objektuma bárhol helyettesíteni tudja az általánosabb osztály objektumát.

Már ne tartozik az OO programozás alapelvei közé, de a C++-ban mégis megtalálható a típusmozgatás. Azaz a user által definiált típusok ugyanúgy viselkednek, mint a beépített típusok. Az OO programozás alap gondolata, hogy a rendszer funkcióit egymással együttműködő objektumok valósítják meg.

Egységbezárás C++-ban

Egy struktúrának nem csak tagváltozói (attribútumai), de tagfüggvényei (metódusai/műveletei) is lehetnek.

Tagváltozók a struktúra adattagjai. A `->` és a `.` operátorokkal tagváltozókra hivatkozhatunk. Tagfüggvényeket kétféleképpen adhatunk meg: osztálydefinícióban, vagy a struktúra definíción kívül is. Ha két azonosnevű osztálynak van azonos nevű és paraméterlistájú tagfüggvénye, akkor a hatókör operátort (`::`) használjuk, hogy elkerüljük a névütközést és a fordító tudja, hogy melyik függvény melyik osztály tagja.

Adatrejtés

Az egységbe zárás áttekinthetővé teszi a kódot és lehetőséget ad az adatrejtésre is. Eddig minden adat publikus volt minden függvény számára. Azonban a struktúra definíciójában elhelyezett `private` kulcsszó után szereplő tagváltozók és függvények csak az osztályon belül láthatóak. ha az osztályból fel szeretnénk használni, akkor egy változót kell deklarálni, ezt nevezzük példányosításnak. Az így létrejött osztálypéldány objektumnak nevezzük.

Konstruktorok, destruktorok

A konstruktor feladata, hogy egy objektum létrejöttkor inicializálja azt. Speciális tagfüggvény, neve megegyezik az osztály nevével és automatikusan hívódik meg. A destruktor a korábban lefoglalt erőforrások felszabadításáért felelős. `~Osztálynév()` alakban. Nincs paramétere.

Dinamikus adattagot tartalmazó osztályok

Dinamikus memóriakezelésre használt függvények a `malloc` és `free` függvények. A `malloc` csak a lefoglalni kívánt tárterület méretét ismeri bájtokban. A `new` operátor egy lefoglalt típusra mutató pointert ad vissza. A `delete` operátor a felszabadítani kívánt objektum destruktort hívja meg. Tömbök lefoglalására alkalmas a `new []`, felszabadításukra a `delete []` operátor. A korábban lefoglalt memóriaterületeket mindig fel kell szabadítani, különben elfolyik a memóriánk.

Másolókonstruktor

A másolókonstruktor az osztály típusával megegyező referenciát vár. Az újonnan létrehozott objektumot egy már meglévő objektum alapján hozza létre (inicializálja). Ha a másolókonstruktor érték szerint adunk át egy paramétert akkor az adott változó lemásolódik, felhasználjuk a függvénytörzsben, majd a függvényből való kilépés után felszabadul. ha referenciát adunk át, akkor az átadott érték változhat. A fordító bitenként fogja másolni az objektumot, hacsak nem írunk neki másolókonstruktor. A bitenkénti másolás a sekély másolás, a dinamikus adattagok másolása a mély másolás.

Friend függvények és osztályok

A `friend` szó használatával egy osztály feljogosít globális függvényeket és más osztályok tagfüggvényeit, hogy hozzáférhessenek a védett tagjaikhoz. Friend osztályok esetében az osztály egy másik osztályt jogot fel, hogy hozzáférhessen a védett tagjaihoz. A `friend` tulajdonság nem öröklődik, és nem is tranzitív. Az inicializálás egy újonnan létrehozott adatszerkezet kezdőértékének beállítását jelenti. Míg az értékadás egy már meglévő adatszerkezetnek ad új értéket.

A konstruktor inicializálási listájába lehetn a tagváltozókat inicializálni. `Konstruktor(argumentumlista) lista`

Statikus tagok

Osztályok esetében definiálhatunk statikus tagváltozókat, amik az adott osztályhoz (nem az osztály objektumaihoz) tartoznak. Ezeket osztályváltozóknak is nevezhetjük, mert az osztály minden objektumára vonatkozó közös értéket vettek fel. Deklarációjuk az osztályban a `static` kulcsszóval történik, de a statikus tagváltozót ugyanekkor definiálni is kell az osztályon kívül. Ez a hatókör operátorral (`::`) történhet meg, hogy tudjuk melyik osztályhoz tartozik az adott statikus változó. lehetőség van statikus tagfüggvények definiálására is. A `static` kulcsszót kell megadni az adott tagfüggvényre vonatkozóan. Statikus tagfüggvényekből a nem statikus tagfüggvények és tagváltozók nem érhetőek el. A `this` mutató nem értelmezhető statikus függvények törzsében. Akkor érdemes statikus tagváltozókat használni, ha olyan változóra van szükség, amely az osztály minden változójára közös. A statikus tagváltozók mindig a globális változókkal együtt inicializálódnak, a `main` függvénybe lépés előtt.

A beágyazott definíciók: az enumeráció, osztály-, struktúra- és típusdefiníciók osztálydefiníción belüli megadása. A privát részben található definíciók csak az őt tartalmazó osztály tagfüggvényei számára érhetőek el, a publikus részben találhatóak mindenki számára elérhetőek. A beágyazott osztály nem biztosít speciális jogokat az őt tartalmazó osztálynak és ez visszafelé is ugyanígy igaz.

Operátorok és túlterhelésük (93-96. oldal)

Az operátorok (`+`, `-`, `*`, `/`, `%`, stb.) argumentumokkal végeznek műveleteket, az így kapott eredmények a visszatérési értékek. Egy operátor mellékhatásának nevezzük, ha az operátor megváltoztatja az argumentum értékét. Az operátorkiértékelések sorrendjét (erősségét) egy precedenciátáblázat tartalmazza. Az operátortúlterhelés saját operátorok definiálását jelenti. A függvények és operátorok közötti különbség csupán a kiértékelési szabályrendszer. Az operátorok tulajdonképpen speciális nevű függvények és kiértékelésük speciális szabályrendszert vesz alapul. célunk, hogy az általunk definiált típusokra is megadhassunk operátor működéseket. Így az argumentumtípusok közül legalább az egyik nem beépített típus lesz. Az operátort tagfüggvényként érdemes definiálni, ha első paramétere olyan típus, amit mi írunk.

III. rész

Második felvonás

**Bátf41 Haxor Stream**

A feladatokkal kapcsolatos élő adásokat sugároz a <https://www.twitch.tv/nbatfai> csatorna, melynek permanens archívuma a <https://www.youtube.com/c/nbatfai> csatornán található.

DRAFT

11. fejezet

Helló, Berners-Lee!

11.1. C++ és Java nyelvek összehasonlítása

Felhasznált könyvek a két nyelv összehasonlításához:

- C++: Benedek Zoltán, Levendovszky Tihamér Szoftverfejlesztés C++ nyelven
- Java: Nyékyné Dr. Gaizler Judit et al. Java 2 útikalauz programozóknak 5.0 I-II.

A Java nyelv nagyon hasonlít a C és a C++ nyelvekhez, hiszen szintaxisa ezen nyelvekből származtatható. Egy alapvető különbség a Java és a C++ nyelvek között már a fordítás során előjön. Míg a C++ egy natív nyelv (azaz fordítás során egy az egybe gépi kódot állít elő), addig a Java egy interpreteres nyelv, azaz a fordítás során egy bájtkódot (Javabytecode) állít elő. Ez a Javabytecode a Java Virtual Machine számára értelmezhető kód. Bármilyen rendszeren amin rajta van a JVM, az előbb említett bájtkód értelmezhető lesz. Egyszerre előnye és hátránya is ez a Java-nak. Mivel a bájtkód előállítás majd annak tovább fordítása időigényes, ezért ez nagy hátrány. Viszont ha azt nézzük, hogy a JVM által egy adott Java kódot bármilyen rendszeren futtathatjuk nagy előnyre teszünk szert. Hiszen egy blackPanther rendszeren megírt C++ programot nem biztos, hogy tudunk futtatni SUSE operációs rendszer alatt.

Szintaxis:

A Java és a C++ nyelv szintaxisa lényegében megegyezik. Viszont

Kifejezések:

Már tudjuk, hogy a C/C++ nyelveknél a részkifejezések kiértékelődési sorrendjére semmilyen szabály nem vonatkozik, magyarul: nincs megszabva melyik fog eloször, melyik következ ~ ore kiértékel ~ odni. Ez már Java-ban nem így van, Java a kiértékelési sorrend balról jobbra történik.

Típuskonverziók

Mint a C++ esetén, a Javában is van automatikus/kézi típuskonverzió, sőt a Java annyira típusos nyelv, hogy minden kifejezésben megvizsgálja a hatósági program, hogy a kifejezésben található típusok "összeférnek" egymással. Például jelez, ha lebegőpontos típusú változót szeretnénk egészen konvertálni, ekkor ugye adtavesztés megy végbe.

Objektumok

A Java programozási nyelv alapvető eleme az objektum. Az ilyen nyelveket objektum orientált (OO) programozási nyelveknek nevezzük. Az objektum az adott feladat szempontjából fontos, a valódi világ valamilyen elemének a rá jellemző tulajdonságai és viselkedései által modellezett eleme. Az objektumokkal kapcsolatban valamilyen feladatokat szeretnénk megoldani. A nyelv tervezésekor fontos szempont volt az, hogy az objektumok többé-kevésbé állandóak, de a hozzájuk tartozó feladatok nem, ezért az objektumok kapnak nagyobb hangsúlyt. A mai programok nagyon sok, egymással kölcsönhatásban álló elemből állnak, így nem is igazán programokról, hanem programrendszerekről beszélhetünk.

11.2. Python

A Python egy könnyen tanulható, de hatékony programozási nyelv. Magasszintű adatstruktúrái, az objektum-orientáltság egyszerű megközelítése, elegáns szintaxisa, dinamikus típusossága és interpreteres mivolta ideális script-nyelvvé teszi. Kiválóan alkalmas gyors fejlesztői munkákra, nagyobb projektek összefogására. Tulajdonképpen a Python nem a kifejezés szigorú értelmében vett interpreter: a Python byte-kódot fordít és azt futtatja. A szintaxisból adódóan nagyon gyors fordítót lehet írni, ezért "néz ki úgy", mintha interpretált lenne a nyelv.

Nagy előnye a nyelvnek, hogy nincs szükség a fordítási fázisra hiszen a Python forrás önmagában elegendő az értelmező számára. Emellett az interpreter könnyen bővíthető C,C++ vagy más C-ből hívható nyelven megírt függvényekkel és adattípusokkal. Segítségével rövid programok írása gyors, egyszerű és jól áttekinthető. Feltűnően gyorsabb programokat hozhatunk létre Python nyelven, mint például C/C++ nyelven.

Típusok

Python nyelvben nincs szükség explicit módon meghatározni a változók típusát. Futási idő alatt meghatározza minden változó típusát.

11.3.

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

11.4.

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

11.5.

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

DRAFT

12. fejezet

Helló, Arroway!

12.1. OO szemlélet

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

12.2. Homokozó

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

12.3. "Gagyi"

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

12.4. Yoda

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

12.5. Kódolás from scratch

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

DRAFT

13. fejezet

Helló, Liskov!

13.1. Liskov helyettesítés sértése

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

13.2. Szülő-gyerek

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

13.3. Anti OO

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

13.4. Hello, Android!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

13.5. Ciklomatikus komplexitás

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

DRAFT

14. fejezet

Helló, Mandelbrot!

14.1. Reverse engineering UML osztálydiagram

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

14.2. Forward engineering UML osztálydiagram

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

14.3. Egy esettan

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

14.4. BPMN

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

14.5. TeX UML

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

DRAFT

15. fejezet

Helló, Chomsky!

15.1. Encoding

Fordítsuk le és futtassuk a Javat tanítók könyv MandelbrotHalmazNagyító.java forrását úgy, hogy a fájl nevekben és a forrásokban is meghagyjuk az ékezetes betűket! <https://www.tankonyvtar.hu/hu/tartalom/tkt/javat-tanitok-javat/adatok.html>

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

15.2. OOCWC lexer

Izzítsuk be az OOCWC-t és vázoljuk a <https://github.com/nbatfai/robocar-emulator/blob/master/justine/rcemu/src/lexer> és kapcsolását a programunk OO struktúrájába!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

15.3. Paszigráfia Rapszódia OpenGL full screen vizualizáció

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

15.4. Paszigráfia Rapszódia LuaLaTeX vizualizáció

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

15.5. Perceptron osztály

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

DRAFT

16. fejezet

Helló, !

16.1.

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

16.2.

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

16.3.

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

16.4.

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

16.5.

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

DRAFT

17. fejezet

Helló, !

17.1.

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

17.2.

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

17.3.

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

17.4.

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

17.5.

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

DRAFT

18. fejezet

Helló, !

18.1.

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

18.2.

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

18.3.

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

18.4.

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

18.5.

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

DRAFT

19. fejezet

Helló, !

19.1.

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

19.2.

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

19.3.

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

19.4.

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

19.5.

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

DRAFT

IV. rész

Irodalomjegyzék

DRAFT

19.6. Általános

[MARX] Marx, György, *Gyorsuló idő*, Typotex , 2005.

19.7. C

[KERNIGHANRITCHIE] Kernighan, Brian W. & Ritchie, Dennis M., *A C programozási nyelv*, Bp., Műszaki, 1993.

19.8. C++

[BMECPP] Benedek, Zoltán & Levendovszky, Tihamér, *Szoftverfejlesztés C++ nyelven*, Bp., Szak Kiadó, 2013.

19.9. Lisp

[METAMATH] Chaitin, Gregory, *META MATH! The Quest for Omega*, http://arxiv.org/PS_cache/math/pdf/0404/0404335v7.pdf , 2004.

Köszönet illeti a NEMESPOR, <https://groups.google.com/forum/#!forum/nemespor>, az UDPROG tanulószoba, <https://www.facebook.com/groups/udprog>, a DEAC-Hackers előszoba, <https://www.facebook.com/groups/DEACHackers> (illetve egyéb alkalmi szerveződésű szakmai csoportok) tagjait inspiráló érdeklődésükért és hasznos észrevételeikért.

Ezen túl kiemelt köszönet illeti az említett UDPROG közösséget, mely a Debreceni Egyetem reguláris programozás oktatása tartalmi szervezését támogatja. Sok példa eleve ebben a közösségben született, vagy itt került említésre és adott esetekben szerepet kapott, mint oktatási példa.