

Univerzális programozás

Írd meg a saját programozás tankönyvedet!

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	2020. május 21.	

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	Aktualizálás, javítások.	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	3
2. Helló, Turing!	5
2.1. Végtelen ciklus	5
2.2. Lefagyott, nem fagyott, akkor most mi van?	6
2.3. Változók értékének felcserélése	8
2.4. Labdapattogás	9
2.5. Szóhossz és a Linus Torvalds féle BogomIPS	12
2.6. Helló, Google!	14
2.7. 100 éves a Brun tétel	17
2.8. A Monty Hall probléma	17
3. Helló, Chomsky!	18
3.1. Decimálisból unárisba átváltó Turing gép	18
3.2. Az $a^n b^n c^n$ nyelv nem környezetfüggetlen	18
3.3. Hivatkozási nyelv	18
3.4. Saját lexikális elemző	19
3.5. l33t.1	20
3.6. A források olvasása	22
3.7. Logikus	24
3.8. Deklaráció	24

4. Helló, Caesar!	27
4.1. double ** háromszögmátrix	27
4.2. C EXOR titkosító	28
4.3. Java EXOR titkosító	30
4.4. C EXOR törő	30
4.5. Neurális OR, AND és EXOR kapu	34
4.6. Hiba-visszaterjesztékes perceptron	34
5. Helló, Mandelbrot!	37
5.1. A Mandelbrot halmaz	37
5.2. A Mandelbrot halmaz a std::complex osztállyal	40
5.3. Biomorfok	42
5.4. A Mandelbrot halmaz CUDA megvalósítása	43
5.5. Mandelbrot nagyító és utazó C++ nyelven	43
5.6. Mandelbrot nagyító és utazó Java nyelven	43
6. Helló, Welch!	44
6.1. Első osztályom	44
6.2. LZW	46
6.3. Fabejárás	50
6.4. Tag a gyökér	54
6.5. Mutató a gyökér	57
6.6. Mozgató szemantika	58
7. Helló, Conway!	60
7.1. Hangyaszimulációk	60
7.2. Java életjáték	60
7.3. Qt C++ életjáték	60
7.4. BrainB Benchmark	61
8. Helló, Schwarzenegger!	62
8.1. Szoftmax Py MNIST	62
8.2. Szoftmax R MNIST	62
8.3. Mély MNIST	62
8.4. Deep dream	62
8.5. Robotpszichológia	63

9. Helló, Chaitin!	64
9.1. Iteratív és rekurzív faktoriális Lisp-ben	64
9.2. Weizenbaum Eliza programja	64
9.3. Gimp Scheme Script-fu: króm effekt	64
9.4. Gimp Scheme Script-fu: név mandala	64
9.5. Lambda	65
9.6. Omega	65
 III. Második felvonás	 66
10. Helló, Arroway!	68
10.1. A BPP algoritmus Java megvalósítása	68
10.2. Java osztályok a Pi-ben	68
 IV. Irodalomjegyzék	 69
10.3. Általános	70
10.4. C	70
10.5. C++	70
10.6. Lisp	70

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 alkálni 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 (hogyan 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?

1.2. Milyen doksikat olvassak el?

- Olvasgasd 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
- [[KERNIGHANRITCHIE](#)]
- [[BMECPP](#)]
- Az igazi kockák persze csemegéznek a C nyelvi szabvány [ISO/IEC 9899:2017](#) kódcsipeteiből is.

1.3. Milyen filmeket nézzek meg?

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

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!

Saját megoldás videó:https://youtu.be/2kQEw_1BLFM

Megoldás forrása:https://github.com/HernyakPisti/Prog1/tree/master/bhax/thematic_tutorials/bhax_textbook/-turing

Tanulságok, tapasztalatok, magyarázat...

```
int main() {  
    for (;;) {}  
    return 0;  
}
```

Az alábbi kód egy magot fog futtatni 100%-on. A kód fordításához a "gcc inf-100.c -o 100" parancsot használjuk majd "./100" paranccsal futtatjuk. A futtatás alatt azt szeretnénk elérni hogy egy mag állandóan 100%-on fusson vagy ahhoz nagyon közel, hogy ezt leellenőrizzük nyissunk meg egy másik terminált és a "top" paranccsal figyeljük meg hogy mi történt.

```
#include <unistd.h>  
int main() {  
    for (;;) {  
        sleep(1);  
    }  
    return 0;  
}
```

Az alábbi kód egy magot fog 0%-on futtatni. A kód fordításához a "gcc inf-0.c -o 0" parancsot használjuk majd a "./0" paranccsal futtatjuk. A futtatás alatt azt szeretnénk elérni hogy egy map állandóan 0%-on fusson. Ha ezt szeretnénk leellenőrizni a "top" paranccsal akkor sajnos nem látunk semmi változást mivel sok más program fut így a mi programunk nem jelenik meg.

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

int main() {
    #pragma omp parallel
    while(1) {
    }
    return 0;
}
```

Az utolsó kód a feladatban arra törekszik hogy minden magot 100%-on futtasson. A kód fordításához a "gcc inf-all.c -o all -fopenmp" parancsot kell használni. A futtatáshoz "./all" parancsot használjuk. A "-fopenmp" kapcsolót a "#pragma omp parallel" sor miatt használjuk ami ahhoz kell hogy az összes magot párhuzamosan tudjuk használni. Egy másik terminálban futtatott "top" paranccsal ellenőrizhetjük hogy a program minden szálát 100%-on futtat vagy ahhoz nagyon közel.

Minden kód alapja egy végtelen ciklus, mivel nem adjuk meg meddig fusson a kód ezért a ciklusunk végtelen. Erre két példát is használunk a "for(;;)" és a "while(1)". Mindkettő egy végtelen ciklus míg a while-lal írt könnyebben olvasható addig a for-ral írt egyszerűbb.

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? Sehoggy, 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 Saját megoldás videó: https://youtu.be/2kQEw_1BLFM

Megoldás forrása: https://github.com/HernyakPisti/Prog1/tree/master/bhax/thematic_tutorials/bhax_textbook/-turing

Tanulságok, tapasztalatok, magyarázat...

```
#include <stdio.h>

int main(){
    int a=10;
    int b=20;
    printf("Eredeti: a=%d, b=%d.\n", a,b);

    int temp = 0;
    temp=a;
    a=b;
    b=temp;
    printf("Segédes változás: a=%d, b=%d.\n", a,b);
    //a=20, b=10
    b=b-a; //b=10-20=-10 a=20
    a=a+b; //a=20+(-10)=10 b=-10
    b=a-b; //b=10-(-10)=20 a=10
    printf("Matoperátoros változás: a=%d, b=%d.\n", a,b);

    a = a ^ b;
    b = a ^ b;
    a = a ^ b;
    printf("XOR operátoros változás: a=%d, b=%d.\n", a,b);

    return 0;
}
```

A kód fordításához a "gcc váltcsere.c -o csere" parancsot használjuk, a futtatáshoz pedig "./csere". Ha lefordítjuk és futtatjuk a kódot akkor hirtelen végig szalad a kód és nem tudjuk mitörtént csak cserélgetünk 2 számot.

A legelső cserélés egy egyszerű segéd változóval történik. Az az létrehozunk egy temp nevű változót, amibe eltároljuk az a-t majd az a-t egyenlővé tesszük b-vel és a végén b-t egyenlővé tesszük a temp-pel. Ez egy egyszerű változó csere amit elég könnyű végig követni, hátránya az hogy külön memória hely kell a temp változónak.

A második cserélés a matematikával történik meg. Egyszerű összeadás kivonás. A b-t egyenlővé tesszük b-a-val, a-t egyenlővé tesszük az a+b összeggel majd a b-t egyenlővé tesszük a-b-vel. Hogy tudjuk vé-

gigkövetni hol mennyi a változók értékének egyszerűen kikommentelve írjuk fel hogy mikor mennyi az értékük.

A harmadik cserélés az XOR opetárt használja. Az első sorban kombináljuk az a-t és a b-t a XOR segítségével ezzel megkapjuk az a+b-t és ezt letároljuk az a-ban. A második sorban az összeget XOR-ral kombináljuk b-vel amiből kiszűrjük a b-t így megkapjuk az a-t és ezt letároljuk az b-ben. Mivel még az a-ban az összeg van letárolva így megint kombináljuk az összeget b-vel (az összeg még mindig a-ban van letárolva) így kiszűrjük az eredeti a értéket mivel az eredeti a érték már b-ben van így megkapva az eredeti b-t amit letárolunk a-ba. Ezzel kész a csere. Innen látszik hogy a XOR jó mód információ tárolásra mivel bármikor eltudjuk tüntetni az XOR-t ha még egyszer használjuk az XOR-t.

2.4. Labdapattogás

Először if-ekkel, majd bármiféle logikai utasítás vagy kifejezés használata nélkül írd 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> Saját megoldás videó: https://youtu.be/-2kQEw_1BLFM

Megoldás forrása: https://github.com/HernyakPisti/Prog1/tree/master/bhax/thematic_tutorials/bhax_textbook/-turing

Tanulságok, tapasztalatok, magyarázat...

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int sz=10,m=10,a=1,b=1;
#define SZG 50
#define MG 20
int falsz[SZG];
int falm[MG];
void palya(){
system("clear");
for (int i=0;i<(SZG+2);i++)
    printf("X");
    printf("\n");
for (int i=0;i<m;i++){
    printf("X");
        for (int j=1;j<(SZG+1);j++)
            printf(" ");
        printf("X\n");
    }
printf("X");
for (int i=0;i<sz;i++)
    printf(" ");
printf("X");
for (int i=(sz+1);i<SZG;i++)
    printf(" ");
```

```
printf("X\n");
for (int i=m+1;i<MG;i++){
    printf("X");
    for (int j=1;j<(SZG+1);j++)
        printf(" ");
    printf("X\n");
}
for (int i=0;i<(SZG+2);i++)
    printf("X");
printf("\n");
}
void mozdulj(){
sz=sz+a;
m=m+b;
a=a*falsz[sz];
b=b*falm[m]; /
}

int main(){

for (int i=1;i<SZG;i++)
    falsz[i]=1;
for (int i=1;i<MG;i++)
    falm[i]=1;
falsz[0]=-1;
falsz[SZG-1]=-1;
falm[0]=-1;
falm[MG-1]=-1;

while(1){
palya();
mozdulj();
usleep(50000);
}
}
```

A programot a "gcc pattogifnelkull.c -o pattog" paranccsal fordítjuk le, majd "./pattog" paranccsal futtatjuk. A kód összetettebb, mivel több függvény is szerepel benne. Kezdjük a main-nel:

```
int main(){

for (int i=1;i<SZG;i++)
    falsz[i]=1;
for (int i=1;i<MG;i++)
    falm[i]=1;
falsz[0]=-1;
falsz[SZG-1]=-1;
falm[0]=-1;
falm[MG-1]=-1;
```

```
while(1){
palya();
mozdulj();
usleep(50000);
}
```

A main-ben két for ciklus van amik úgy néznek ki mint amik tömböket töltenek fel csupa 1-sel. A kód ezeket használja majd hogy meghatározza a labda helyzetét. A ciklusok utáni 4 sorban pedig a falakat látjuk ahol a labdának irányt kell változtatni ezért az ott lévő értékeket átírjuk -1-re. Ezután pedig jön egy végtelen ciklus amiben 2 új függvény szerepel a palya és a mozdulj, ezekről később lesz szó. Az usleep pedig egy egyszerű késleltetés arra hogy a labdának nyomon lehessen követni az útvonalát. Térjünk át a palya függvényre és az előtte lévő sorokra:

```
#include <stdio.h>
#include <stdlib.h>
#include <unistd.h>
int sz=10,m=10,a=1,b=1;
#define SZG 50
#define MG 20
int falsz[SZG];
int falm[MG];
void palya(){
system("clear");
for (int i=0;i<(SZG+2);i++)
    printf("X");
    printf("\n");
for (int i=0;i<m;i++){
    printf("X");
        for (int j=1;j<(SZG+1);j++)
            printf(" ");
        printf("X\n");
    }
printf("X");
for (int i=0;i<sz;i++)
    printf(" ");
printf("X");
for (int i=(sz+1);i<SZG;i++)
    printf(" ");
printf("X\n");
for (int i=m+1;i<MG;i++){
    printf("X");
        for (int j=1;j<(SZG+1);j++)
            printf(" ");
        printf("X\n");
    }
for (int i=0;i<(SZG+2);i++)
    printf("X");
    printf("\n");
}
```

A kód a szokásos include-okkal kezdődik. Ezután 4 változót vezetünk be, sz ami a szélességet jelenti ahonnan a labda indul majd, m ami a magasságot jelenti ahonnan a labda indul majd, valamint van a és b változó amik a labdairányát jelölik majd. Ezután két konstans következik az SZG és az MG ezek jelölik a pálya méretét. Ezután létrehozuk a két tömböt amiket majd azt tároljuk hogy a labda falnál van-e vagy nem, erre 1 és -1 használjuk.

Ezután következik a pályafüggvény ami void típusú mivel nincs visszaadott értéke. A függvény egy "system("clear")" paranccsal indul ami letörli a terminált hogy a pálya jól látszódjon. Ezután több for ciklus is van amikkel a pályát rajzoljuk és utána a labdát majd megint a pályát. A kód úgy működik hogy a falakat "X"-ek jelölik. A falak közti terület pedig " " (egyszerű szóközök). A rajzolás pedig úgy történik hogy indulunk mindig az első sortól aztán addig írjuk az "X"-et meg a " " -ök amíg nem kell a labdát leírni. Aztán leírjuk a labdát ami "X" karakter jelöl. Majd ezután befejezzük a pályát.

A kód utolsó része pedig a mozdulj függvény:

```
void mozdulj() {  
    sz=sz+a;  
    m=m+b;  
    a=a*falsz[sz];  
    b=b*falm[m];  
}
```

Ez a függvény is void típusú mivel nincs visszatérő értéke. Ebben a függvényben használjuk az első 4 változót amit bevezettünk. Ezek fogják megszabni hogy a labda milyen X,Y koordinátán áll és milyen irányba kell haladnia. A "sz=sz+a" és "m=m+b" jelenti a koordinátát, az "a=a*falsz[sz]" és a "b=falm[m]" pedig az irányt. Az a és b értéke vagy 1 vagy -1 ha egy akkor pozitív irányba halad a labda ha -1 akkor negatív irányba.

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!

Saját megoldás videó:https://youtu.be/2kQEw_1BLFM

Megoldás forrása:https://github.com/HernyakPisti/Prog1/tree/master/bhax/thematic_tutorials/bhax_textbook/-turing

Tanulságok, tapasztalatok, magyarázat...

```
#include <time.h>  
#include <stdio.h>  
  
void  
delay (unsigned long long loops)  
{  
    for (unsigned long long i = 0; i < loops; i++);  
}  
  
int  
main (void)
```

```
{
    unsigned long long loops_per_sec = 1;
    unsigned long long ticks;

    printf ("Calibrating delay loop..");
    fflush (stdout);

    while ((loops_per_sec <= 1))
    {
        ticks = clock ();
        delay (loops_per_sec);
        ticks = clock () - ticks;

        if (ticks >= CLOCKS_PER_SEC)
        {
            loops_per_sec = (loops_per_sec / ticks) * CLOCKS_PER_SEC;

            printf ("ok - %llu.%02llu BogoMIPS\n", loops_per_sec / 500000,
                (loops_per_sec / 5000) % 100);

            return 0;
        }
    }

    printf ("failed\n");
    return -1;
}
```

Az alábbi kód Bátfa Norbert-től van egy az egyben.

A kód fordításához a "gcc bogomips.c -o bogo" parancsot használjuk a futtatáshoz pedig a "./bogo" parancsot.

A kód arra való hogy megállalítsa a számítógép bogomips értékét, ami úgy írható le hogy milyen erős is az adott számítógép, de nekünk nem is ez a fontos hanem a main függvényben lévő while ciklus fejben.

```
while ((loops_per_sec <= 1))
```

Erről a while ciklusról van szó amiben van egy bit shiftelő operátor van ami csak annyit tesz hogy az int-ben tárolt számot bitenként lépteti. Másszóval mindig a 2 hatványát adja meg.

Ezt a while ciklus fejet használjuk arra is hogy megtudjuk a bit hosszát:

```
#include <stdio.h>
int main(){
    int db=1;
    int szo=1;

    while(szo <= 1){
        db++;
        //printf("%d\n", szo);
    }
}
```

```
printf("Szóhossz: %d bit\n", db);  
return 0;  
}
```

A kódot a "gcc szohosz.c -o szo" paranccsal fordítjuk majd a "./szo" paranccsal futtatjuk.

A kód egy egyszerű main függvényből áll amiben egyből deklarálunk 2 változót a db és a szo változót. A db fogja jelölni hogy hányszor futt le a lentebb lévő while ciklus a szo pedig egy egyszerű int amit a while ciklus fejben használunk amiben léptetjük a bitet. A ciklus lefutása után pedig kiírjuk a db változó értékét, ami 32 lesz, ezzel megtudva azt hogy a bit hossza a számítógépen 32 bit.

2.6. Helló, Google!

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

Saját megoldás videó:https://youtu.be/2kQEw_1BLFM

Megoldás forrása:https://github.com/HernyakPisti/Prog1/tree/master/bhax/thematic_tutorials/bhax_textbook/-turing

Tanulságok, tapasztalatok, magyarázat...

```
#include <stdio.h>  
#include <math.h>  
  
void kiir(double tomb[], int db) {  
    for (int i=0; i<db; i++)  
        printf("%f\n", tomb[i]);  
}  
  
double tavolsag(double PR[], double PRv[], int db) {  
    double osszeg=0.0;  
    for (int i=0; i<db; i++)  
        osszeg += (PRv[i]-PR[i])*(PRv[i]-PR[i]);  
    return sqrt(osszeg);  
}  
  
int main() {  
    double L[4][4]=  
    {  
        {0.0, 0.0, 1.0/3.0, 0.0},  
        {1.0, 1.0/2.0, 1.0/3.0, 1.0},  
        {0.0, 1.0/2.0, 0.0, 0.0},  
        {0.0, 0.0, 1.0/3.0, 0.0}  
    };  
};
```

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

int i, j;

for(;;) {

    for(i=0; i<4; i++) {
        PR[i] = 0.0;
        for(j=0; j<4; j++)
            PR[i] += (L[i][j] * PRv[j]);
    }
    if (tavolsag(PR, PRv, 4) < 0.00000001)
        break;
    for(i=0; i<4; i++)
        PRv[i] = PR[i];
}
kiir(PR, 4);
return 0;
}
```

A kódot a "gcc pagerank.c -o page -lm" paranccsal fordítjuk és a "./page" paranccsal futtatjuk. A -lm kapcsoló az sqrt függvény miatt kell.

A program feladata hogy egy 4 honlapból álló hálózatra kiszámolja az adott 4 lap PageRank értékét. A PageRank még a Google alapítói is használták kezdetben. Az elgondolás az az volt hogy annál jobb egy honlap ha arra minnél több jó értékelésű honlap mutat.

A kód itt is több függvényből áll ezért kezdjük megint a main-nel:

```
int main() {

    double L[4][4] =
    {
        {0.0, 0.0, 1.0/3.0, 0.0},
        {1.0, 1.0/2.0, 1.0/3.0, 1.0},
        {0.0, 1.0/2.0, 0.0, 0.0},
        {0.0, 0.0, 1.0/3.0, 0.0}
    };

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

    int i, j;

    for(;;) {

        for(i=0; i<4; i++) {
            PR[i] = 0.0;
            for(j=0; j<4; j++)
                PR[i] += (L[i][j] * PRv[j]);
        }
    }
}
```



```
    }  
    if (tavolsag(PR, PRv, 4) < 0.00000001)  
        break;  
    for(i=0; i<4; i++)  
        PRv[i] = PR[i];  
}  
kiir(PR, 4);  
return 0;  
}
```

A mainben létrehozunk egy double típusú link mátrixot L néven. Ezután még másik 2 double típusú tömböt hozunk létre PRv és PR néven. Az eddigi adatok mind a feladatból lettek kinézve. Ezután deklarálunk 2 változót i és j-t amiket majd ciklusokban használunk. Ezután jön egy többszörösen összetett for ciklus sorozat. Az első for ciklus egy egyszerű végtelen ciklus, mivel a PageRanket nem elég egyszer kiszámolni hanem többször kell.

A második for ciklus gyorsan feltöltjük 0.0-val a PR tömböt mivel mindig üres tömb kell a számoláshoz és pár sorral lentebb a PR tömböt feltöltjük más számokkal amiket mindig le kell nullázni ezért kell mindig újra nullázni.

A harmadik for ciklusban történik egy egyszerű mátrix szorzás. Amit a végtelen ciklusból való kilépésért felelős if követ. Az if azt vizsgálja hogy a tavolsag függvény visszaadott értéke elég kicsi-e ahhoz hogy kilépjen-e a végtelen ciklusból ha igen akkor áttér a kiir függvényre és véget ér a program de ha nem akkor jön a negyedik for ciklus amia PRv-be másolja a PR értékeit, majd kezdődik az egész ciklus újra.

Nézzük meg a két függvényt is:

```
#include <stdio.h>  
#include <math.h>  
  
void kiir(double tomb[], int db) {  
  
    for (int i=0; i<db; i++)  
        printf("%f\n", tomb[i]);  
}  
  
double tavolsag(double PR[], double PRv[], int db) {  
  
    double osszeg=0.0;  
    for (int i=0; i<db; i++)  
        osszeg += (PRv[i]-PR[i]) * (PRv[i]-PR[i]);  
  
    return sqrt(osszeg);  
}
```

A kód elején egy új include van a math.h. Ez az sqrt függvény miatt szükséges. Az első függvény ami szerepel az a kiir. Ez egy void típusú függvény mivel nincs visszatérési értéke, csak arra szolgál hogyha a végtelen ciklusból kiléptünk akkor a végső értékeket kiírja egy printf-el

A második függvény egy double típusú mivel a visszatérése egy sqrt függvényből jön. Ezzel a függvénnyel vizsgáljuk azt hogy a PageRankek között elég kicsi-e már az eltérés mert ha elég kicsi lesz akkor már nem kell tovább számolni és megkaptuk a végső PageRank értékeket.

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

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

Tanulságok, tapasztalatok, magyarázat...

3. fejezet

Helló, Chomsky!

3.1. Decimálisból unárisba átváltó Turing gép

Állapotátmenet gráfiával megadva írd meg ezt a gépet!

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!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

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: https://youtu.be/S_jBmmQixQs

Megoldás forrása: https://github.com/HernyakPisti/Prog1/tree/master/bhax/thematic_tutorials/bhax_textbook/-chomsky

BNF-ben definiálva a C utasítás fogalma

Backus-Naur-forma környezetfüggetlen nyelvtanok leírására használható metaszintaxis: végeredményben formális nyelvek is leírhatók vele. A legtöbb programozási nyelv elméleti leírása és/vagy szemantikai dokumentumai általában BNF-ban vannak leírva. A BNF széles körben használatos a számítógépek programozási nyelveinek nyelvtanának leírására, ideértve az utasítás készleteket és a kommunikációs protollokat is.

```

<utasítás> ::= <címkézett_utasítás> | <kifejezés_utasítás> | < ←
    összesített_utasítás> | <kiválasztó_utasítás> | <literációs_utasítás> ←
    | <vezérlésátadó_utasítás>
<címkézett_utasítás> ::= <azonosító> ";" <utasítás> | "case" < ←
    állandó_kifejezés> ":" <utasítás> | "default :" <utasítás>
<kiválasztó_utasítás> ::= "if" "(" <kifejezés> ")" <utasítás> | "if" "(" ←
    <kifejezés> ")" <utasítás> "else" <utasítás> | "switch" "("<kifejezés ←
    >)" <utasítás>
<kifejezés_utasítás> ::= [<kifejezés>] ";"
<vezérlésátadó_utasítás> ::= "goto" <azonosító> ";" | "continue;" | " ←
    break;" | "return" [<kifejezés>] ";"
<összesített_utasítás> ::= "{" [<deklarációs_lista>] [<utasítási_lista>] ←
    "}"
<deklarációs_lista> ::= <deklaráció> | <deklarációs_lista> <deklaráció>
<literációs_utasítás> ::= "while" "(" <kifejezés> ")" <utasítás> | "do" ←
    <utasítás> "while" "(" <kifejezés> ")" <utasítás> | "for" "(" [ < ←
    kifejezés> ";" <kifejezés> ";" <kifejezés> ] ")" <utasítás>
<utasítási_lista> ::= <utasítás> | <utasítási_lista> <utasítás>

```

Tanulságok, tapasztalatok, magyarázat...

```

#include <stdio.h>
int main(){
for(int i=0;i<10;i++)
printf("%d\n",i);
return 0;
}

```

A kód fordításához használjuk elsőnek a "gcc version.c -o version -std=c99" parancsot majd futassuk a "./version" paranccsal. Ezzel a paranccsal hiba nélkül fordul és fut a kód. Azonban ha a fordításnál a parancsot máshogy írjuk, "gcc version.c -o version -std=c89" valahogy így, azonban hibát kapunk mivel a fordításnál megadott "std=c89" kapcsoló miatt a kódot C89-es szabványon fordítjuk amiben nem lehet a for ciklus ciklusmagjában deklarálni változót, azonban ez C99-es szabványban már lehet.

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!

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

Megoldás forrása: https://github.com/HernyakPisti/Prog1/tree/master/bhax/thematic_tutorials/bhax_textbook/-chomsky

Tanulságok, tapasztalatok, magyarázat...

```
%{
#include <stdio.h>
int realnum = 0; //hány változótadtunk a bementre
}%
digit [0-9]
%%
{digit}* (\.{digit}+)? {realnum++;
printf("[realnum: %s %f]", yytext, atof(yytext));
}
%%
int main(){
yylex();
printf("Number of real numbers: %d\n", realnum);
return 0;
}
```

Ez az első kódunk ami .l végződésű tehát ezt máshogy kell fordítanunk mint az eddigieket. Elsőnek is térjünk ki hogy mit takar .l végződés. A .l végződésből tudjuk hogy egy lex fájlról beszélünk amit úgy lehetne körülírni hogy adunk meg neki különböző kritériumokat és azok alapján készít nekünk egy teljesen új kódot ami a megadott kritériumok alapján dolgozik. Ezek a kritériumok közül az elsőben szerepelnek a .c részek pl include-ok, struktúrák/tömbök/változók deklarálása.. Ezt választja el az úgy nevezett szabályrendszerrel "%%" jelölés.

Ezután adjuk meg a szabályrendszert. Itt a szabályrendszerben egy sor van ami fontosabb: "{digit}* (\.{digit}+)?" Ebben a sorban írjuk le a lexernek hogy milyen alakú a valós szám. A digit jelölés pár sorral fentebb van kifejtve ami csak annyit tesz hogy digit [0-9] azaz a digit egy 0-tól 9-ig lévő bármilyen szám lehet. A * azt jelöli hogy bármennyi lehet de legalább egy kell hogy legyen. Ezután a "." rész csak egyszerűen a pontot jelöli a \ jel védi le a pontot hogy azt .-nak olvassa a lexer. ami ezután jön pedig a valós szám utolsó része a tizedes számok. Ezek is 0-9 tartományból lehetnek számok és a "+" jelöli azt hogy ebből több is lehet.

Ezután jön a lexer utolsó része ahol egyszerűen leírjuk a maint amiben a "yylex()" sor indítja el a lexer függvényt, az utána lévő sorok meg már ismerősek egyszerű kiírás és visszatérés 0-val ha minden sikerült.

Most hogy tudjuk mi az a lexer fordítsuk és futtassuk. Elsőnek kell a .l forrásból kell készíteni egy .c forrást. Ezt a "-lex real.l" paranccsal tesszük meg. Ezután az aktuális könyvtárba elkészül nekünk a .c forrás. Ezt a "gcc lex.yy.c -lfl" paranccsal fordítsuk és ezután a "./a.out" paranccsal futtassuk. Az -lfl kapcsoló a lexer miatt szükséges.

3.5. l33t.l

Lexelj össze egy l33t ciphert!

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

Megoldás forrása: https://github.com/HernyakPisti/Prog1/tree/master/bhax/thematic_tutorials/bhax_textbook/-chomsky

Tanulságok, tapasztalatok, magyarázat...

```
%{
#include <stdio.h>
#include <stdlib.h>
#include <time.h>
#include <ctype.h>
#define LEETSIZE (sizeof leetdict / sizeof (struct cypher))

struct cypher {
char c;
char* leet[4];
}
leetdict[]={
    {'a', {"4", "4", "@", "/-\\\"}},
    {'b', {"b", "8", "|3", "|"}},
    {'c', {"c", "(", "<", "{"}},
    {'d', {"d", "|)", "|", "|"}},
    {'e', {"3", "3", "3", "3"}},
    {'f', {"f", "|=", "ph", "|#"}},
    {'g', {"g", "6", "[", "+"}},
    {'h', {"h", "4", "|-|", "[-"]}},
    {'i', {"1", "1", "|", "!"}},
    {'j', {"j", "7", "_|", "_/"}},
    {'k', {"k", "|<", "1<", "|{"}},
    {'l', {"l", "1", "|", "|_"}},
    {'m', {"m", "44", "(V)", "|\\|/"}},
    {'n', {"n", "|\\|", "/\\|", "/V"}},
    {'o', {"0", "0", "()", "[]"}},
    {'p', {"p", "/o", "|D", "|o"}},
    {'q', {"q", "9", "O_", "(,)"}},
    {'r', {"r", "12", "12", "|2"}},
    {'s', {"s", "5", "$", "$"}},
    {'t', {"t", "7", "7", "'|'"}},
    {'u', {"u", "|_|", "(_)", "[_]"}},
    {'v', {"v", "\\|/", "\\|/", "\\|/"}},
    {'w', {"w", "VV", "\\|\\|/", "(/\\|)"}},
    {'x', {"x", "%", ")(", ")("}},
    {'y', {"y", "", "", ""}},
    {'z', {"z", "2", "7_", ">_"}},

    {'0', {"D", "0", "D", "0"}},
    {'1', {"I", "I", "L", "L"}},
    {'2', {"Z", "Z", "Z", "e"}},
    {'3', {"E", "E", "E", "E"}},
    {'4', {"h", "h", "A", "A"}},
    {'5', {"S", "S", "S", "S"}},
    {'6', {"b", "b", "G", "G"}},
    {'7', {"T", "T", "j", "j"}},
    {'8', {"X", "X", "X", "X"}},
    {'9', {"g", "g", "j", "j"}}
```

```
};

%}
%%
. {
int found = 0;
for (int i = 0; i < LEETSIZE; ++i){
if (leetdict[i].c == tolower(*yytext)){
int r = 1+(int) (100.0*rand()/RAND_MAX+1.0);

printf("%s", leetdict[i].leet[r%4]);
found = 1;
break;
}
}
if (!found)
printf("%c", *yytext);
}
%%

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

Ebben a feladatban is egy lexert kell használni. Itt is ugyan az a felállítás mint az előző feladatban. Készítünk egy .l forrást ami készít nekünk egy .c forrást azokkal a kritériumokkal amikkel mi készítettük el a .l forrást.

Ebben a kódban is az első részben különböző include-ok szerepelnek, konstans/tömb/stuktúra definiálás. Ezután jön megint a szabályrendszer. A szabályrendszernek az az elve ha talál a leetdict nevű tömbből egy megegyező elemet a bementről akkor azt kicseréli az azonos sorban lévő másik karakter(ekkel), és miután egyezés van akkor ki is lépünk a ciklusból hogy ne menjünk tovább hiszen egy karaktert csak egyszer kell kicserélni. A szabályrendszer után pedig jön a main, ebben is meghívjuk a lexert a "yylex()" sorral a "srand" függvény pedig a random szám miatt kell, mivel random hogy a megtalált karaktert mire cseréljük ki.

Most hogy tudjuk hogy működik a forrás fordítsuk futtassuk. Ezt a "lex leet.l" paranccsal tesszük ez elkészíti nekünk a .c forrást. Ezt a "gcc lex.yy.c -lfl" paranccsal fordítsuk és az "./a.out" paranccsal futtassuk. A "-lfl" kapcsoló itt is a lexer miatt szükséges.

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 megváránzásra, elkapja valamelyiket esetleg a splint vagy a frama?

i.

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

Hiba: Return a value ignored. Ha a SIGINT jel figyelmen kívül volt hagyva akkor maradjon úgy, azonban ha nem volt figyelmen kívül hagyva akkor a jelkezelő függvény kezelje.

ii.

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

A ciklusban deklarált i változó értékehez először hozzáadunk egyet majd visszatér a megnövelt érték.

iii.

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

A ciklusban deklarált i változó értékét először visszatér aztán megnöveli eggyel.

iv.

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

Hiba: Expression has undefined behavior (left operand uses i, modified by right operand). A programnak le kellene futnia de mivel az i változó egyszerre változik és van értéként megadva ezért hiba történik.

v.

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

Hiba: Right operand is non-boolean (int). Rosszul van a for ciklus ciklusmagja felírva, mivel a (*d++ = *s++) a ciklusmagban van felírva és itt feltételként kéne szerepelni.

vi.

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

Hibák: Argument 2 modifies a, used by argument 1 és Argument 1 modifies a, used by argument 2. A kód 2 számot fog kiírni az f függvénytől függően. Az első esetben "a" eggyel kisebb mint ++a, a második esetben viszont ++a és "a" ugyan az a szám lesz.

vii.

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

A kód 2 számot fog kiírni. Az első amit az f függvény ad vissza a második amit pedig az f-nek adtunk át elsőnek.

viii.

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

A kód 2 számot fog kiírni. Az első szám az a memória cím amit az f függvény ad meg, a második esetben pedig az a szám aminek kiírta a kód a memória címét.

Megoldás forrása: https://github.com/HernyakPisti/Prog1/tree/master/bhax/thematic_tutorials/bhax_textbook/-chomsky

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

Tanulságok, tapasztalatok, magyarázat...

Ebben a feladatban több különböző forráskódot használtunk. Volt köztük ami warningot okozott míg volt ami nem. A warningok pontosabb megértéséhez használjuk a "splint" parancsot.

A forrást pedig a "gcc splint.c -o splint" parancssal fordítsuk és a "./splint" parancssal futtassuk ami nekünk egy csomó különböző számot ír ki, de itt most nem ez a lényeg, hanem a "splint splint.c" parancs eredménye. A parancs után megjelenik a terminálban különböző warningok amiket jobban kifejtve olvashatunk el.

3.7. Logikus

Hogyan olvasod természetes nyelven az alábbi Ar nyelvű formulákat?

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

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

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

Tanulságok, tapasztalatok, magyarázat...

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;`
- `int *b = &a;`
- `int &r = a;`
- `int c[5];`
- `int (&tr)[5] = c;`
- `int *d[5];`
- `int *h ();`
- `int *(*l) ();`
- `int (*v (int c)) (int a, int b)`
- `int (*(z) (int)) (int, int);`

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

Megoldás forrása: https://github.com/HernyakPisti/Prog1/tree/master/bhax/thematic_tutorials/bhax_textbook/-chomsky

Tanulságok, tapasztalatok, magyarázat...

```
#include <stdio.h>

int* eg(); //egészre mutató mutatót visszaadó függvény

int main () {

    int a; //egész
    int* b; //egészre mutató mutató
    int* c = &a; //egész referenciája
    int d[100]; //egészek tömbje
```

```
int* e = &d[10]; //egészek tömbjének referenciája (nem az első elemé)
int* pp[100]; //egészre mutató mutatók tömbje
int* (*eg_pointer) (); //egészre mutató mutatót visszaadó függvényre mutató ←
    mutató
int (*egesz (int c3)) (int c1, int c2); //egészet visszaadó és két egészet ←
    kapó függvényre mutató mutatót visszaadó, egészet kapó függvény
int (*(egeszre) (int f3)) (int f1, int f2); //függvényt mutató egy egészet ←
    visszaadó és két egészet kapó függvényre mutató mutatót visszaadó, ←
    egészet kapó függvényre
return 0;
}
```

A programot a "gcc deklaracio.c -o dekl" paranccsal fordítsuk és a "./dekl" paranccsal futtasuk. A terminálba nem fog semmi megjelenni mivel csak lefordul a kód a benne lévő kód sorokat egy egyszerű "cat deklaracio.c" paranccsal tekinthetjük meg.

4. fejezet

Helló, Caesar!

4.1. double ** háromszögmátrix

Megoldás videó: https://youtu.be/rfLaGXYM8_4

Megoldás forrása: https://github.com/HernyakPisti/Prog1/tree/master/bhax/thematic_tutorials/bhax_textbook/-caesar

Tanulságok, tapasztalatok, magyarázat...

```
#include <stdlib.h>
#include <stdio.h>
int main() {

    double **tm;
    int nr=5;

    if((tm = (double**) malloc (nr*sizeof(double*)))==NULL)
        return -1;

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

    for (int i=0; i<nr;++i){
        for (int j=0; j<i+1;++j)
            tm [i][j] = i +j;
    }

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

```
for (int i=0; i<nr;++i){
    free(tm[i]);
}
free(tm);
return 0;
}
```

A feladatban főleg a malloc és a free használata a lényeg ezért róluk röviden pár szóban. A free/malloc páros memória kezelésre használják, a malloc foglal a free felszabadítja a memóriát. A kódban használjuk a malloc visszatérési értékét is ami NULL ha nem sikerül a memória foglalás. A malloc alap visszatérési típusa void * mi a kódban viszont double** visszatérési típust akarunk. A mallocnak meg kell adni paraméterként azt hogy mennyi memóriát kell lefoglalni. A free-nek meg kell adni paraméterként egy mutatót hogy melyik memóriát területet szabadítsa fel.

A kódot a "gcc tm.c -o tm" paranccsal fordítsuk és a "./tm" paranccsal futtassuk. Ekkor a kimeneten megjelenik az alsó háromszög mátrixunk, viszont nem látjuk azt ha van memória foglalási/felszabadítási hiba ezért futtassuk le a kódot a valgrind-el is ami kijelzi nekünk a memória változását a kód futása után. "valgrind ./tm" paranccsal látjuk azt hogy nincs memória foglalási/felszabadítási hiba.

A kód elején a szükséges include-ok szerepelnek utána pedig a változók deklarálása. A "double **tm" a mutatók tömbjére fog hivatkozni, mert lesz egy tömbünk amiben mutatók lesznek és ezek a mutatók lesznek a sor és oszlopindexekre mutatók, szóval tulajdonképpen egy két dimenziós tömb. Az "int nr=5" pedig hogy hány sort használunk.

Ezután egy hiba kezelés következik: Hogyha a tm-nek nem sikerül lefoglalni a memóriát a malloc-cal akkor az NULL értéket ad vissza és ha ez történik akkor egy egyszerű -1 értékkel kilépünk. Ilyen hiba történhet hogyha mondjuk nem tudjuk a memóriát írni avagy nincs elegendő memória vagy a mallocnak megadott méret 0.

Ezután a tm-nek elkezdjük lefoglalni a méretet egy for ciklussal. Itt (nr*sizeof(double*)) -nyi memóriát foglalunk, itt is történik hibakezelés és ez történne meg akkor felszabadítjuk a tm-et és kilépünk a programból.

Ezután a tm-et feltöltjük értékekkel két egymásba ágyazott for ciklussal. Az indexelésre figyelni kell nehogy túlindexeljük a ciklusunk.

Ezután a tm-ben lévő értékek kiírása következik két egymásba ágyazott for ciklussal. Ezután felszabadítjuk a memóriát a free használatával.

4.2. C EXOR titkosító

Írj egy EXOR titkosítót C-ben!

Megoldás videó: https://youtu.be/rfLaGXYM8_4

Megoldás forrása: https://github.com/HernyakPisti/Prog1/tree/master/bhax/thematic_tutorials/bhax_textbook/-caesar

Tanulságok, tapasztalatok, magyarázat...

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

#define MAX_KULCS 100
#define BUFFER_MERET 256

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);

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;
    }

    write(1,buffer,olvasott_bajtok);
}

return 0;
}
```

A feladatban az XOR titkosítást fogjuk használni ami bitenként történik, ennek a titkosításnak nagy előnye hogy nagyon gyors és ha még egyszer elvégezzük a titkosítást ugyan azt kapjuk vissza. Másik nagy előnye hogy alacsony szinten történik.

A kódot "gcc titok.c -o titok" paranccsal fordítsuk majd a "./titok kulcs bemeneti fájl kimeneti fájl" a mi esetünkben a kulcs méret 8 a bemeneti fájl a tiszta.txt.

A kód elején a szokásos include-ok és utána konstansok definiálása. Ezután következik a main. A mainben létrehozunk két char típusú tömböt, utána deklarálunk két int-et amik a ciklusokhoz fognak kelleni. Ezután az "int kulcs_meret = strlen(argv[1])" sorral a kulcs_meret-nek megadjuk a parancssori argumentumként érkezett string hosszát. A következő sorban a strncpy függvénnyel lemásoljuk a kulcs változóba a kulcsot.

Ezután jön a ciklus ami addig fut amíg van bemenet mivel a read függvényt használjuk beolvasásra aminek három paramétere van, honnan olvasunk (0-van megadva ami a sztender bemenet), hova olvasunk és hány bitet akarunk olvasni. A while ciklusban belül van egy for ciklus amiben történik a titkosítás. A buffer[i]-t mindig felülírjuk a titkosított karakterrel, ezután léptetjük a kulcsot is de mivel a kulcs sokkal rövidebb mint a bemenet ezért mindig maradékosan leosztjuk a kulcs_merettel így ha kell újra kezdjük a kulcs_index-et. Ezután kiírjuk a titkosított szöveget ami általában valami szemét lesz. A wrtie függvényt használjuk erre aminek ugyan úgy három paramétere van, hova írjunk (1 van megadva mivel az jelenti a sztender kimenetet),

honnan írunk és mennyi bitet írunk.

4.3. Java EXOR titkosító

Írj egy EXOR titkosítót Java-ban!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

4.4. C EXOR törő

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

Megoldás videó:https://youtu.be/rfLaGXYM8_4

Megoldás forrása: https://github.com/HernyakPisti/Prog1/tree/master/bhax/thematic_tutorials/bhax_textbook/-caesar

Tanulságok, tapasztalatok, magyarázat...

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

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

int tiszta_lehet(const char titkos[],int titkos_meret){

    double szohossz = atlagos_szohossz(titkos,titkos_meret);

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

```
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);

}

int main(){

    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;

    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 szöveg: [%s\n]",ii ←
                ,ji,ki,li,mi,ni,oi,pi,titkos);

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

    return 0;
}
```

Ebben a feladatban a XOR titkosítóval kapott szöveget fogjuk visszafejteni, ezzel is látva azt hogy ha valamin kétszer használjuk a XOR műveletet önmagát kapjuk vissza.

A kódot "gcc nem_titok.c -o nem_titok" parancssak fordítsuk majd "./nem_titok kulcs bemeneti fájl" parancssal futtasuk. A kulcs 8 karakterből áll, a bemeneti fájl pedig az a fájl amit a titok.c program generált nekünk.

A kód elején a szükséges include-ok és a konstansok definiálása. Ezután több függvény jön amikről később. Kezdjük a main függvénnyel:

```
int main() {

    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;

    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 szöveg: [%s\n]",ii ←
                ,ji,ki,li,mi,ni,oi,pi,titkos);

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

    return 0;
}
```

A mainben a szükséges tömböket deklaráljuk és egy szükséges segéd mutatót a "*p" alatt. Ezután egy while ciklus jön ami a beolvasásért felelős. A ciklusmag itt kicsit hosszabb mivel nem tudjuk hogy hány bitet kell beolvasni, hogy ne szaladjunk túl a BUFFER méretből. Egy sima if-el van megvalósítva hogyha kevesebb a beolvasott akkor ez ha több a beolvasott mint a BUFFER akkor az. Mivel a p-be olvasunk ezért a ciklusban mindig hozzáfűzzük a beolvasott biteket. A while ciklust követő for ciklus arra az esetre van hogyha lenne üres hely akkor azokat feltöltjük "0"-kal.

Ami ezután jön az a fő törés mivel itt előállítjuk az összes létező kulcs variánst, ezt nyolcs egymásba ágyazott for ciklussal érjük el. Azért nyolc mert a kulcs méret is nyolc. A ciklusok végén mindig lefuttatunk egy if-et ami ha tiszta szöveget ad nekünk akkor kiírjuk a hozzátartozó kulcsot és a kulccsal kapott tiszta szöveget. Hogy mi lehet tiszta szöveg arról majd később beszélek. Hogyha sikerült tiszta szöveget kapni az nem feltétlen a tiszta szöveg és mivel nem használunk segéd tömböt ezért kell a kiírás után megint XOR művelet végrehajtani az exor függvényvel, ezzel gátolva meg azt hogy ne az egyszer tiszta szöveget próbáljuk meg újra visszafejteni a következő ciklusban.

És akkor a maradék függvényekről pár szót:

```
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;
}

int tiszta_lehet(const char titkos[],int titkos_meret){

    double szohossz = atlagos_szohossz(titkos,titkos_meret);

    return szohossz < 9.0 && szohossz > 6.0 &&
```

```
    strcasestr (titkos, "nem") && strcasestr (titkos, "hogy") && strcasestr ( ←  
        titkos, "az") && strcasestr (titkos, "ha");  
}  
  
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);  
}
```

A függvények közül az exor_tores függvénnyel kezdem. Ennek négy paramétert adunk meg amik a XOR művelet végrehajtásához szükségesek, ebben a függvényben van másik két függvény. Az exor függvény végzi el a XOR műveletet, míg a tiszta_lehet függvény ellenőrzi ciklusonként hogy a kapott szöveg tiszta-e. Egy kapott szöveg akkor tiszta ha eleget tesz különböző kritériumoknak. Pl szerepeljen a szövegben a "nem", "hogy", "az" és a "hogy" szavak, hogy ezek a szavak szerepelnek-e a szövegben arra a strcasestr függvényt használjuk ami egyszerűen megkeresi a megadott szövegben a megadott szót. Azért ezek a szavak mert ezek nagyon gyakran fordulnak elő a szövegben azonban ha az eredeti szövegben nem szerepelnek ezek a szavak akkor semmi se fog megjelenni ezért figyelni kell hogy a bemeneti szövegben szerepeljenek. Másik kritérium az hogy a szavak átlagos hossza 6 és 9 között legyen, hogy ezt vizsgáljuk erre van az atlagos_szo_hossz függvény.

4.5. Neurális OR, AND és EXOR kapu

R

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

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

Tanulságok, tapasztalatok, magyarázat...

4.6. Hiba-visszaterjesztéses perceptron

C++

Megoldás videó: https://youtu.be/rfLaGXYM8_4

Megoldás forrása: https://github.com/HernyakPisti/Prog1/tree/master/bhax/thematic_tutorials/bhax_textbook/-caesar

Tanulságok, tapasztalatok, magyarázat...

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

int main (int argc, char **argv)
{
    png::image <png::rgb_pixel> png_image (argv[1]);

    int size = png_image.get_width() * png_image.get_height();

    Perceptron* p = new Perceptron (3, size, 256, 1);

    double* image = new double[size];

    for (int i = 0; i<png_image.get_width(); ++i)
        for (int j = 0; j<png_image.get_height(); ++j)
            image[i*png_image.get_width() + j] = png_image[i][j].red;

    double value = (*p) (image);

    std::cout << value << std::endl;

    delete p;
    delete [] image;

    return 0;
}
```

A kódot a "cpp main.cpp -o main" paranccsal fordítsuk és a "./main mandel.png" paranccsal futtasuk. A mandel.png egy könyvtárban szerepel a main.cpp-vel

A kód elején a szükséges include-ok szerepelnek, azonban a "mlp.hpp"-nek nem kacsacsőrökkel van körülzárva ebből azt tudjuk hogy a main.cpp-vel aktuális könyvtárban keresi a fordító.

A main első sorában szerepel a png::image objektum amiben megadjuk hogy a png-nk milyen típusú és mivel a mandel.png rgb típusú ezért rgb_pixel paramétert adunk meg. Ezután megadunk egy nevet és utána hogy a png-t hol találja a program. Ezután kiszámoljuk a kép méretét egy egyszerű szorzással (magasság*szélesség). Ezután elkészítjük a Perceptron objektumot, aminek foglalunk memóriát. Ezután megadjuk a Perceptronnak hogy hány rétegű legyen ezután a réteg számnak megfelelően annyi paramétert adunk meg. Elsőnek a size-t mivel megadjuk az összes pixel ezután 256 és a végén egyet.

Mivel a képet át kell alakítani egy double tömbbé ezért foglalunk annak is helyet. Ebbe fogjuk bemásolni a képet sor folytonosan. A for ciklus végzi el ezt a másolást. Azért van szükség másolásra mivel a Perceptron objektumnak double* kell átadni paraméternek. A cikluson belül pedig elkérjük a kép[i][j] elemének a vörös csatornáját ciklusonként. Ezzel a kép vektorrá lett alakítva, ezután át kell adni a vektort a Perceptronnak.

A Perceptron majd egy double értékkel fog visszatérni ezért hozunk létre egy double típusú value változót. Ebbe a változóba mentsük el a Perceptron osztály függvényhívó operátorát használva. Ezután egyszerűen kiírjuk az értéket és felszabadítjuk a memóriát amit lefoglaltunk.

DRAFT

5. fejezet

Helló, Mandelbrot!

5.1. A Mandelbrot halmaz

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

Megoldás forrása: https://github.com/HernyakPisti/Prog1/tree/master/bhax/thematic_tutorials/bhax_textbook/-mandel

Tanulságok, tapasztalatok, magyarázat...

Ez a halmaz Benoit B. Mandelbrot, lengyel származású matematikusról, a megalkotójától kapta a nevét. Úgy építhető fel ez a halmaz ha a komplex számsíkon veszünk egy C pontot és erre képezzük a következő rekurzív sorozatot:

$$Z_0 := C$$
$$Z_{i+1} := Z_i^2 + C$$

Ezzel az egyszerű rekurzióval definiált sorozatról be lehet bizonyítani, hogy bizonyos C számok választása esetében végtelenbe tart, vagy más C -k esetében pedig nullához tart. Más eset nem lehetséges.

A Mandelbrot-halmaznak azok és csak azok a C komplex számok az elemei, amelyek esetében a fenti sorozat nullához tart. Ábrázolás esetében ezeket általában feketére szokták festeni, míg a többi pontot attól függően, hogy "milyen gyorsan" tart a végtelenbe.

Most hogy megismertük a Mandelbrot halmazt, készítsük mi is el egy ábrát róla, ehhez használjuk a következő kódot.

```
#include <iostream>
#include "png++/png.hpp"
#include <sys/times.h>

#define MERET 600
#define ITER_HAT 32000

void
mandel (int keypadat[MERET][MERET]) {

    // MÉRÜNK IDŐT (PP 64)
    clock_t delta = clock ();
```

```
// MÉRÜNK IDŐT (PP 66)
struct tms tmsbuf1, tmsbuf2;
times (&tmsbuf1);

// 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, ujureZ, ujimZ;
// Hány iterációt csináltunk?
int iteracio = 0;
// Végigzongorázzuk a szélesség x magasság rácsot:
for (int j = 0; j < magassag; ++j)
{
    //sor = j;
    for (int k = 0; k < szelesseg; ++k)
    {
        // c = (reC, imC) a rács csomópontjainak
        // megfelelő komplex szám
        reC = a + k * dx;
        imC = d - j * dy;
        // z_0 = 0 = (reZ, imZ)
        reZ = 0;
        imZ = 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;
        }

        kepadat[j][k] = iteracio;
    }
}
```

```
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;
}

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

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

    int kepadat[MERET][MERET];

    mandel(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;
}
```

Az alábbi kód Bátfai Norbert-től van.

A kód fordítása a "g++ mandelpngt.c++ -o mandelpng" paranccsal történik és futtatása pedig a "./mandelpng fájlnev" a fájlnev lesz az a név amit a kimeneti png fog névként kapni.

A kód két függvényből áll, a main-ből és mandel-ből. A mandel függvény elején változókat vezetünk be amikkel majd az időt fogjuk mérni, majd ezután a szükséges változóknak megadjuk az értékeket amiket még a kód legelején definiáltunk. Ezután lebegőpontos típusú változókkal történik a számlálás.

Ezután jön mag a számítás, ezt két egymásba ágyazott for ciklussal és egy while ciklussal tesszük. A két for ciklus felel azért hogy minden rácson átmenjünk. A while ciklusban történik a rekurzív számolás, ebből a ciklusból való kilépést biztosítja az iteracio változó.

Ha befejeztük a számítás akkor a ciklus után kiszámoljuk az eltelt időt amit megjelenítünk a terminálon. Ezután következik a main függvény.

A main egyből egy hibakezeléssel indul, hogyha a felhasználó rosszul használná a programot akkor egy üzenet segít neki és a program visszatér -1 értékkel és leáll.

Ha a felhasználó jól használja a programot akkor a mainben létrejön egy kepadat nevű mátrix amit egyből át is adunk a mandel függvényben ahol megtörténik a Mandelbrot halmaz kiszámítása.

Ami után visszatér a vezérlés a mandel függvénytől a main-nek létrehozuk a kep változót amit majd legenerálunk, a képet pixelről pixelre színezzük ki két egymásba ágyazott for ciklussal, majd ha kész a kép a terminálra kiírjuk a megadott fájlnévvel a mentve-t és a program leáll.

A Mandelbrotról lehetne hosszabban is írni de hiába tanultam emelt szintű matematikát nem hiszem hogy képes lennék többet magyarázni, inkább átadom magam annak az élvezetnek hogy egyre jobban belemélyedek milyen alakzatokra képes a halmaz.

5.2. A Mandelbrot halmaz a std::complex osztállyal

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

Megoldás forrása: https://github.com/HernyakPisti/Prog1/tree/master/bhax/thematic_tutorials/bhax_textbook/-mandel

Tanulságok, tapasztalatok, magyarázat...

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

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

    int szelesseg = 1920;
    int magassag = 1080;
    int iteraciosHatar = 255;
    double a = -1.9;
    double b = 0.7;
    double c = -1.3;
    double d = 1.3;
```

```
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;
}

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;
}
```

Az alábbi kód Bátfai Norbert-től van.

Ebben a feladatban ugyan az a feladat mint az előzőben csak itt annyi a könnyítésünk hogy a kódban nem lebegőpontos változókkal dolgozunk hanem komplex számokkal ezzel elérve azt hogy az előző küdban használt külön függvényt elhagyhassuk.

A kód fordításához használjuk a "g++ 3.1.2.cpp -lpng -o 3.1.2" parancsot és a futtatásnál pedig "./3.1.2 fájlnev szélesség magasság n a b c d" parancsot. A fájlnev az a név amit a generált kép fog kapni, a többi megadott szám pedig vagy a kép méretei vagy pedig a számításhoz szükséges adatok.

A kód elején itt is egy hibakezelés van, hogyha nem megfelelő mennyiségű bemeneti adatot adunk meg akkor a program kiírja hogy mi a megfelelő bemenet és -1 értékkel leáll, azonban ha megfelelő mennyiségű bemenetet adunk meg akkor a program elkezd számítani a megadott értékkel a halmazt.

Az összetett számítás miatt a terminálon mindig olvasható a számítás állapota %-ra lebontva

A számolás itt is két egymásba ágyazott for ciklussal és egy while ciklussal történik. A for ciklusok itt is azért vannak hogy minden sor minden oszlop cellájába elmenjünk, míg a while ciklusban történik maga a számolás.

A cikluson belül közben folyamatosan készül a kép színezése is, és ha kész van minden akkor a program erről értesít minket és leáll.

Ez a feladat és az előző egy remek példa arra hogy mindenre létezik több megoldás is de vannak olyakos sokkal könnyebb utak is, ilyen másik példa mint amikor a lexert használtuk hogy helyettünk írjon meg több 100 oldalnyi kódot csupán 20-40 sor alapján.

5.3. Biomorfok

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

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

Tanulságok, tapasztalatok, magyarázat... Passz!

5.4. A Mandelbrot halmaz CUDA megvalósítása

Megoldás videó:

Megoldás forrása:

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!

Megoldás forrása: https://github.com/HernyakPisti/Prog1/tree/master/bhax/thematic_tutorials/bhax_textbook/-mandel

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

Tanulságok, tapasztalatok, magyarázat...

Ebben a feladatban egy külső program segítségével készítjük el a nekünk szükséges programokat a forrásainkból.

Ahhoz hogy működjön a programunk elsőnek telepítenünk kell a QT szoftvert. Hogyha ez megvan akkor a QT-beli qmake parancs használtavál elkészítünk egy make file-t a forrásainkból ami általl kapunk is egy kész futtatható fájlt, amit ha futtatunk felugrik egy interaktív ablak amibe kijelöléssel tudunk nagyítani.

A feladat futtatásához a terminálba a következő parancsokat kell beírni. "qmake frak.pro" (ezáltal elkészül a make file-unk), "make" (ezáltal elkészül az egész program aminek a végeredménye egy már futtatható file) "./frak" paranccsal felugrik az interaktív ablak.

Ebben az interaktív ablakban tudunk egyre jobban belenagyítani a képbe, ami minden nagyítás után újra számolódik egy kis elcsúszással a kijelöléshez képest.

5.6. Mandelbrot nagyító és utazó Java nyelven

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: <https://youtu.be/qe12vdwuYYk>

Megoldás forrása: https://github.com/HernyakPisti/Prog1/tree/master/bhax/thematic_tutorials/bhax_textbook/-welch

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!

A kód fordításához használjuk a "g++ polargen.cpp polargenteszt.cpp -o polargen" parancsot majd a futtatáshoz "./polargen".

Ebben a feladatban egy saját osztályt próbálunk ki random számok generálására. Ehhez a polargen.cpp és polargenteszt.cpp kódokat használjuk és a polargen.h headert

Kezdjük az értelmezést a polargenteszt.cpp-vel.

```
#include <iostream>
#include "polargen.h"

int
main (int argc, char **argv)
{
    PolarGen pg;

    for (int i = 0; i < 10; ++i)
        std::cout << pg.kovetkezo () << std::endl;

    return 0;
}
```

Egy egyszerűbb kódról van itt szó. A main első sorában készítünk egy PolarGen típust, amit aztán egy for ciklusban fogunk meghívni 10-el. A ciklusban mindig meghívjuk a következő tagfüggvényt amivel majd a random számokat generálni fogjuk, és azokat írjuk ki mindig.

Térjünk át a polargen.h fájlra.

```
#ifndef POLARGEN__H
#define POLARGEN__H

#include <cstdlib>
#include <cmath>
#include <ctime>

class PolarGen
{
public:
    PolarGen ()
    {
        nincsTarolt = true; //konstruktor igazra inicializál
        std::srand (std::time (NULL)); //inicializál egy random számot
    }
    ~PolarGen ()
    {
    }
    double kovetkezo ();

private:
    bool nincsTarolt;
    double tarolt;
};

#endif
```

Ez itt maga az osztály, az osztály elején szerepel egy konstruktor ami egy logikai tagot inicializál igazra és után egy másik egy random számot. Ezek után pedig meghívásra kerül a következő függvény amiben a random szám generálás fog történni.

Nézzük meg az utolsó fájlt a polargen.cpp-t

```
#include "polargen.h"

double
PolarGen::kovetkezo ()
{
    if (nincsTarolt)
    {
        double u1, u2, v1, v2, w;
        do
        {
            u1 = std::rand () / (RAND_MAX + 1.0);
            u2 = std::rand () / (RAND_MAX + 1.0);
```

```
    v1 = 2 * u1 - 1;
    v2 = 2 * u2 - 1;
    w = v1 * v1 + v2 * v2;
}
while (w > 1);

double r = std::sqrt ((-2 * std::log (w)) / w);

tarolt = r * v2;
nincsTarolt = !nincsTarolt;

return r * v1;
}
else
{
    nincsTarolt = !nincsTarolt;
    return tarolt;
}
}
```

Ebben a kód részletben szerepel a következő függvény aminek a visszatérési értéke lesz mindig az a random szám amit kifogunk írni.

A kód elején egy if szerepel ami annyit tesz hogy leellenőrzi hogy vannak-e már tárolt számok és ha nincsenek akkor készít.

6.2. LZW

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

Megoldás videó: <https://youtu.be/qe12vdwuYYk>

Megoldás forrása: https://github.com/HernyakPisti/Prog1/tree/master/bhax/thematic_tutorials/bhax_textbook/-welch

Tanulságok, tapasztalatok, magyarázat...

A feladatban használt kód Bátfai Nortbert-től származik

A kód fordításához használjuk a "gcc z.c -lm -o z" parancsot, az lm kapcsoló a math.h header miatt szükséges mivel a kódban használjuk az sqrt függvényt, a futtatáshoz pedig a "./z bemenetifájl kimeneti fájl".

Mivel a kód hosszabb terjedelmű ezért több részre bontva értelmezzük. Kezdjük a kód elejével.

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

```
BINF_PTR
uj_elem ()
{
    BINFA_PTR p;

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

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

Az include-ok után szerepel egy önhivatkozó struktúra aminek a fontosabb szerepe a mellékágakra való mutatók állítása, ezután egy hibakezelés következik ami annyit tesz hogyha nem tudunk a "p" változónak elég memóriát foglalni a szabad tárból akkor írjon ki egy hiba üzenetet vagy adja vissza a "p"-t, ezután pedig több függvényt hozunk létre amik majd a fa adatait írják ki vagy a fa kiírását végzik, az utolsó függvény pedig majd a lefoglalt memóriát szabadítja fel.

Ezután nézzük a main függvény elejét.

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

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

    while (read (0, (void *) &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;  
        }  
    }  
}
```

A main elején létrehozunk egy "b" nevű char típusú változót amivel majd a bemenetet fogjuk vizsgálni, Ezután létrehozunk a gyöker elemet aminek mind két ágát ki NULL-ozzuk és létrehozunk egy mutatót ami a gyökérre mutat. Ezután következik a beolvasás egy while ciklussal. Hogyha '0'-ás karakter jön akkor megnézzük hogy létezik-e már gyermek ezen az ágon aminek az értéke NULL és ha igen akkor itt is létrehozunk egy csomópont két mellékággal amiknek ugyan úgy NULL értéket adunk, de ha már létezik 0-ás gyermek és értéke is van akkor a fa mutatót ráállítjuk arra a csomopontra és beolvassuk a következő értéket. Ez történik akkor is ha '1'-es érték jön a bemeneten. Amint olyan pontra érkezünk ahol nincs két mellék ág ott létrehozunk két mellég ágat azoknak az értéküket NULL-ra állítjuk és a fa mutatót visszaállítjuk a gyökérre.

Nézzük meg a main második felét.

```
        printf ("\n");  
        kiir (gyoker);  
  
        extern int max_melyseg, atlagosszeg, melyseg, atlagdb;  
        extern double szorasosszeg, atlag;  
  
        printf ("melyseg=%d\n", max_melyseg-1);  
  
        atlagosszeg = 0;  
        melyseg = 0;  
        atlagdb = 0;  
        ratlag (gyoker);  
  
        atlag = ((double)atlagosszeg) / atlagdb;  
  
        atlagosszeg = 0;  
        melyseg = 0;  
        atlagdb = 0;  
        szorasosszeg = 0.0;  
  
        rszoras (gyoker);
```

```
double szoras = 0.0;

if (atlagdb - 1 > 0)
    szoras = sqrt( szorasosszeg / (atlagdb - 1));
else
    szoras = sqrt (szorasosszeg);

printf ("atlag=%f\nszoras=%f\n", atlag, szoras);

szabadit (gyoker);
}
```

Elméletben eddigre már felvan építve a fánk, márcsak kikell írunk, és az adatait kell kiszámolnunk, erre hozunk létre több változót is, amiknek ezután értékeket adunk a megfelelő számítások elvégzése után. A main utolsó soraiban szereplő "szabadit" függvény fogja felszabadítani a lefoglalt memóriát. A kód további sorai pedig főleg különböző számításokat tartalmaznak, azonban a "kiir" függvényről érdemes még pár szót ejteni.

```
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-1);

        kiir (elem->bal_nulla);

        --melyseg;
    }
}
```

Ez a függvény végzi el a fa kirajzolását. Mivel csak akkor kell kiírni egy elemet ha annak nem NULL az értéke ezért a függvény egyből ezt vizsgálja és ha az érték mégis NULL akkor át is ugorjuk az egész függvényt. Azonban ha az elem értéke más mint NULL akkor elsőnek megnöveljük a mélységet mivel minden elem mellé ()-be melléírjuk hogy melyik magassági szinten vagyunk most, és mindig vizsgáljuk hogy ez a mélység-e esetleg az eddigi legnagyobb. Ezután pedig inorder sorban kiírjuk a fa elemeit. Miután kiírtuk az elemet csökkentsük a mélységet.

6.3. Fabejárás

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

Megoldás videó: <https://youtu.be/qe12vduYYk>

Megoldás forrása: https://github.com/HernyakPisti/Prog1/tree/master/bhax/thematic_tutorials/bhax_textbook/-welch

Tanulságok, tapasztalatok, magyarázat...

A kód bemutatásához egy kisebb bemeneti fájlt használunk hogy lássuk a fát egyben.

A feladat elvégzéséhez csak egy kicsit kell a forrás kódunkat megváltoztatni, de elsőnek nézzük meg hogy milyenek a bejárások.

3 bejárás van, az előző feladatban az inordert néztük meg ami azt jelenti hogy elsőnek az egyik részfát járjuk be aztán a csomópontot majd a másik részfát, ennél a bejárásnál a kiírásnál a gyökér középtájon lesz.

```
pisti@pisti-Lenovo-ideapad-520-15IKB:~/Asztal/osszes_prog/welch$ ./z <kicsi.txt
-----0(3)
-----0(2)
-----0(4)
-----1(3)
-----0(1)
-----0(3)
-----0(5)
-----0(8)
-----0(7)
-----0(9)
-----1(8)
-----1(6)
-----1(7)
-----1(8)
-----1(4)
-----1(5)
-----1(2)
-----0(4)
-----1(3)
-----1(4)
---/(0)
-----0(3)
-----1(4)
-----0(2)
-----0(5)
-----0(4)
-----0(6)
-----1(5)
-----1(3)
-----0(5)
-----1(6)
-----1(4)
-----1(1)
-----0(3)
-----1(4)
-----1(2)
melyseg=9
atlag=5.384615
szoras=1.894662
pisti@pisti-Lenovo-ideapad-520-15IKB:~/Asztal/osszes_prog/welch$
```

Ahhoz hogy ilyen fát kapjunk csak a "kiir" függvényben kell pár sor felcserélni, de mivel alaphoz inorder volt az első feladatban a bejárás ezért itt még a függvényünk változatlan

```
void
kiir (BINFA_PTR elem)
{
```

```
if (elem != NULL)
{
    ++melyseg;

    if (melyseg > max_melyseg)
        max_melyseg = melyseg;

    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-1);

    kiir (elem->jobb_egy);

    --melyseg;
}
```

Itt látható az előző feladatból egy az egyben a "kiir" függvény, a fontosabb sorok a feladat szempontjából a for ciklus és a két kiir (elem->bal_nulla)/kiir (elem->jobb_egy) sor. Ezek a sorok határozzák meg a kiírást, hogy más bejárást érjünk el csak ezeket a sorokat fogjuk cserélgetni.

Akkor most jöjjön a preorder bejárás. Ennél a bejárásnál elsőnek a csomópontot írjuk ki majd a két részét, a kiírásnál emiatt a gyökér legfelül lesz.

```
pisti@pisti-Lenovo-ideapad-520-15IKB:~/Asztal/osszes_prog/welch$ ./z <kicsi.txt
---/(0)
-----0(1)
-----0(2)
-----0(3)
-----1(3)
-----0(4)
-----1(2)
-----0(3)
-----1(4)
-----0(5)
-----1(6)
-----0(7)
-----0(8)
-----1(8)
-----0(9)
-----1(7)
-----1(8)
-----1(5)
-----1(3)
-----0(4)
-----1(4)
-----1(1)
-----0(2)
-----0(3)
-----1(4)
-----1(3)
-----0(4)
-----0(5)
-----1(5)
-----0(6)
-----1(4)
-----0(5)
-----1(6)
-----1(2)
-----0(3)
-----1(4)
melyseg=9
atlag=5.384615
szoras=1.894662
pisti@pisti-Lenovo-ideapad-520-15IKB:~/Asztal/osszes_prog/welch$
```

Hogy ilyen fát kapjunk a "kiir" függvényt a következő képpen kell átírni:

```
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-1);

        kiir (elem->bal_nulla);

        kiir (elem->jobb_egy);
    }
}
```

```

        --melyseg;
    }
}

```

Tehát itt a részfák kirajzolását utolsónak írjuk így érjük el a preorder bejárást.

És akkor nézzük az utolsó bejárást, a postorder bejárást. Ennél a bejárásnál elsőnek a két részfat írjuk ki majd végül a gyökeret, ezért a gyökér itt legalul lesz.

```

^[[Aristi@pisti-Lenovo-ideapad-520-15IKB:~/Asztal/osszes_prog/welch$ ./z <kicsi.txt
-----0(3)
-----0(4)
-----1(3)
-----0(2)
-----0(8)
-----0(9)
-----1(8)
-----0(7)
-----1(8)
-----1(7)
-----1(6)
-----0(5)
-----1(5)
-----1(4)
-----0(3)
-----0(4)
-----1(4)
-----1(3)
-----1(2)
-----0(1)
-----1(4)
-----0(3)
-----0(5)
-----0(6)
-----1(5)
-----0(4)
-----1(6)
-----0(5)
-----1(4)
-----1(3)
-----0(2)
-----1(4)
-----0(3)
-----1(2)
-----1(1)
---/(0)
melyseg=9
atlag=5.384615
szoras=1.894662
pisti@pisti-Lenovo-ideapad-520-15IKB:~/Asztal/osszes_prog/welch$ 

```

Hogy ilyen fát kapjunk a "kiir" függvényt a következő képpen kell átírni:

```

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-1);

--melyseg;
}
```

Tehát itt a részfák kirajzolását tesszük meg elsőnek ezzel érve el a postorder bejárást és a gyökér így leg-
alulra kerül a kiírás során.

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!

Megoldás videó: <https://youtu.be/qe12vduYYk>

Megoldás forrása: https://github.com/HernyakPisti/Prog1/tree/master/bhax/thematic_tutorials/bhax_textbook/-welch

Tanulságok, tapasztalatok, magyarázat...

A feladat megoldásához használt kód Bátfai Norbert-től van.

A kód fordításához használjuk a "g++ z3a7.cpp -o z3a7" parancsot, a futtatáshoz pedig a "./z3a7 bemeneti-
fájl -o bemenetifájl" parancsot.

A kódban használt számítások ugyanazon elv mentén lesznek kiszámolva ezért ezekről most nem írnék
mégegyszer. Ez a kód legnagyobb különbsége az .c kódhoz képest hogy itt a fát nem egy struktúra se-
gítségével dolgozzuk fel hanem egy külön osztállyal. Ennek az osztály elején ugyan úgy létrehozunk egy
gyökérre mutató mutatót "fa" néven. A gyökérre mutató mutató létrehozása után itt is egyesével olvassuk
be a bemenetetről ékező karaktereket amiket a kódban a "b" változóban tárolunk. Az elv itt is az mint a
.c megvalósításban, ha a csomópont értéke NULL akkor ott létrehozunk két új mellékágot amiknek NULL
értéket adunk és a mutató visszaugrik a gyökérre, azonban ha nem NULL az értéke akkor áthelyezzük a
mutatót az új csomópontra és olvassuk a következő bemenetet.

```
class LZWBinFa
{
public:
    LZWBinFa ():fa (&gyoker)
    {
    }
    ~LZWBinFa ()
    {
        szabadit (gyoker.egyGyermek ());
    }
};
```

```
        szabadit (gyoker.nullasGyermek ());
    }
    void operator<< (char b)
    {
        if (b == '0')
        {
            if (!fa->nullasGyermek ())
            {
                Csomopont *uj = new Csomopont ('0');

                fa->ujNullasGyermek (uj);

                fa = &gyoker;
            }
            else
            {
                fa = fa->nullasGyermek ();
            }
        }
        else
        {
            if (!fa->egyenesGyermek ())
            {
                Csomopont *uj = new Csomopont ('1');
                fa->ujEgyenesGyermek (uj);
                fa = &gyoker;
            }
            else
            {
                fa = fa->egyenesGyermek ();
            }
        }
    }

    void kiir (void)
    {
        melyseg = 0;
        kiir (&gyoker, std::cout);
    }
    int getMelyseg (void);
    double getAtlag (void);
    double getSzasas (void);
    friend std::ostream & operator<< (std::ostream & os, LZWBinFa & bf)
    {
        bf.kiir (os);
        return os;
    }
    void kiir (std::ostream & os)
    {
        melyseg = 0;
```



```
        kiir (&gyoker, os);  
    }
```

Az osztály ezutáni részében egy új privát osztály hozunk létre ami a csomópont mutató mozgatásával fog foglalkozni. Ezután pedig jönnek a számítási függvények, majd pedig a main függvény.

```
int  
main (int argc, char *argv[])  
{  
    if (argc != 4)  
    {  
        usage ();  
        return -1;  
    }  
    char *inFile = *++argv;  
    if ((*++argv + 1) != 'o')  
    {  
        usage ();  
        return -2;  
    }  
    std::fstream beFile (inFile, std::ios_base::in);  
    if (!beFile)  
    {  
        std::cout << inFile << " nem letezik..." << std::endl;  
        usage ();  
        return -3;  
    }  
    std::fstream kiFile (*++argv, std::ios_base::out);  
    unsigned char b;  
    LZWBinFa binFa;  
    while (beFile.read ((char *) &b, sizeof (unsigned char)))  
        if (b == 0x0a)  
            break;  
    bool kommentben = false;  
    while (beFile.read ((char *) &b, sizeof (unsigned char)))  
    {  
        if (b == 0x3e)  
        {  
            kommentben = true;  
            continue;  
        }  
        if (b == 0x0a)  
        {  
            kommentben = false;  
            continue;  
        }  
        if (kommentben)  
            continue;  
        if (b == 0x4e)  
            continue;  
        for (int i = 0; i < 8; ++i)
```

```
{
    if (b & 0x80)
        binFa << '1';
    else
        binFa << '0';
    b <= 1;
}
}
kiFile << binFa;
kiFile << "depth = " << binFa.getMelyseg () << std::endl;
kiFile << "mean = " << binFa.getAtlag () << std::endl;
kiFile << "var = " << binFa.getSzoras () << std::endl;
kiFile.close ();
beFile.close ();
return 0;
}
```

A main legelején hibakezelések vannak, ellenőrizzük hogy 4 db parancs sori argumentum van, azok olyan alakúak amilyen alakúnak lenniük kell, hogyha valami hiba van akkor arról értesítjük a felhasználót és a program leáll. Hogyha minden rendben van a kiFile-ban elmentjük a kimeneti fájl nevét és elkezdjük olvasni a bemenetet egy while ciklussal, amit bit maszkolással mindig eltárolunk a "binFa"-ba amit majd átadunk a "kiFile"-nak ami elkészíti ez alapján a kimeneti fájlt.

6.5. Mutató a gyökér

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

Megoldás videó: <https://youtu.be/qe12vdwuYYk>

Megoldás forrása: https://github.com/HernyakPisti/Prog1/tree/master/bhax/thematic_tutorials/bhax_textbook/-welch

Tanulságok, tapasztalatok, magyarázat...

Ebben a feladatban lényegében csak egy karaktert kell beraknunk az előző forráskódba.

Csomopont gyoker;

Ebbe a sorban kell a Csomopont után egy "*" tenni ezzel a gyökeret mutatóvá tesszük, ám ha így fordítjuk a kódot akkor egy csomó hibaüzenet fogad minket mivel így jópár függvényt vagy értékadást avagy hivatkozást át kell írni. Ilyen sorok a következők.

```
LZWBinFa ()
{
    gyoker = new Csomopont ();
    fa=gyoker;
}
~LZWBinFa ()
{
    szabadit (gyoker->egyGyermek ());
```

```
szabadit (gyoker->nullasGyermekek ());  
if(gyoker)  
    delete gyoker;  
}
```

Mivel a gyökér mutató ezért kell foglalni helyet neki a szabadtárban, a szabadit függvény meghívásakor is új karaktert kell használni, és mivel a gyökérnek foglaltunk helyet ezt majd törölni is kell.

```
fa = gyoker;
```

Hasonlóan az előző kód részlethez itt is módosítani kellett a kódot, mivel már nem a címet kell átadni hanem magát a mutatót emiatt márcsak a "fa=gyoker;" kifejezés elég.

6.6. Mozgató szemantika

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

Megoldás videó: <https://youtu.be/qe12vduYYk>

Megoldás forrása: https://github.com/HernyakPisti/Prog1/tree/master/bhax/thematic_tutorials/bhax_textbook/-welch

Tanulságok, tapasztalatok, magyarázat...

Ebben a feladatban leszeretnénk másolni az eredeti fát, ehhez létrehozunk még egy binfát binFa2 néven, amivel az eredetit másoljuk le, ellenőrzés képp ki is írjuk az alábbi kód részlettel

```
LZWBinFa binFa2 (binFa);  
  
kiFile << binFa2;  
  
kiFile << "depth = " << binFa2.getMelyseg () << std::endl;  
kiFile << "mean = " << binFa2.getAtlag () << std::endl;  
kiFile << "var = " << binFa2.getSzoras () << std::endl;
```

Hogy ezt a másolást megtudjuk tenni egy másoló konstruktorra és a másoló értékadás konstruktorra lesz szükségünk ami az eredeti kódban privát osztályban van, ezért onnan ki kell hozni. A másoláshoz az eredetit adjuk meg paraméternek ahol egy másoló függvényt fogunk használni amit majd lentebb nézzük meg. Ennek a függvénynek meg kell adni az eredeti gyökér mutatóját és az eredeti fa mutatóját. A lentebbi kód részletben látjuk ezt

```
LZWBinFa (const LZWBinFa & eredeti)  
{  
    std::cout << "másoló konstruktor\n";  
    gyoker = masol(eredeti.gyoker, eredeti.fa);  
}
```

A másoló érték adásnál peding létrehozunk egy lokális binFát "temp" néven, amit a másoló konstruktorra bízunk, majd a "swap"-et használva megcseréljük a tempet az eredetivel, azonban ehhez szükség lesz a mozgásra. A lentebbi kód részletben látjuk ezt.

```
LZWBInFa & operator= (const LZWBInFa & eredeti)
{
    std::cout << "másoló értékadás\n";
    LZWBInFa temp {eredeti};
    std::swap (*this, temp);

    return *this;
}
```

A fenti kód részletekbe helyeztünk nyomkövetést hogy tudjuk a kód megfelelően működik-e, és ezért van több kiírás az előző kódhoz képest.

A mozgató értékadásnál csak swap-eljük a mutatókat. Az eredeti gyökeret a mi gyökerünkkel, ugyan így az eredeti fa mutatóval. Ezt a lentebbi kód részletben látjuk.

```
LZWBInFa & operator= (LZWBInFa && eredeti)
{
    std::cout << "mozgató értékadás\n";
    std::swap(eredeti.gyoker, gyoker);
    std::swap(eredeti.fa, fa);

    return *this;
}
```

Ezután márcsak a masol függvényt kell megírunk ami a másolást fogja elvégezni. Ez egy rekurzív függvény lesz, hasonlóan a "kiir" függvényhez. Létre kell hozni egy lokális csomópont objektumot `uj_csomopont` néven aminek kezdetben `nullptr` lesz az értéke. Ezután ellenőrizzük hogy az újjonnan létre hozott `csp` mutató értéke nem `NULL`-e és ha nem akkor foglalunk neki helyet a szabadmemóriában, amiután elkérjük a csomópont értékét amit másol, ezt a `getBetu` függvénnyel érjük el, ezután jön a rekurzió.

Ezután másoljuk a másolt csomópontnak a két mellékágát, itt hívjuk meg rekurzívan a masol függvényt.

Hogyha a csomópontunk megegyezik a fa mutatóval akkor át kell állítanunk a fa mutatót az új csomópontra, ezután vissza is térhetünk az új csomóponttal. Ezt mind a lentebbi kód részletben látjuk.

```
Csomopont* masol (Csomopont* csp, Csomopont* fa_ptr){
    Csomopont* uj_csomopont=nullptr;

    if (csp){
        uj_csomopont = new Csomopont (csp->getBetu());
        uj_csomopont->ujNullasGyermek(masol(csp->>nullasGyermek(), fa_ptr));
        uj_csomopont->ujEgyesGyermek(masol(csp->egyenesGyermek(), fa_ptr));

        if (csp==fa_ptr)
            this->fa = uj_csomopont;
    }

    return uj_csomopont;
}
```

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!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

7.3. Qt C++ életjáték

Most Qt C++-ban!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

7.4. BrainB Benchmark

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

DRAFT

8. fejezet

Helló, Schwarzenegger!

8.1. Szoftmax Py MNIST

aa 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. Robotpszichológia

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

III. rész

Második felvonás

DRAFT

**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

10. fejezet

Helló, Arroway!

10.1. A BPP algoritmus Java megvalósítása

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

10.2. Java osztályok a Pi-ben

Az előző feladat kódját fejleszd tovább: vizsgáld, hogy Vannak-e Java osztályok a Pi hexadecimális kifejtésében!

Megoldás videó:

Megoldás forrása:

Tanulságok, tapasztalatok, magyarázat...

IV. rész

Irodalomjegyzék

DRAFT

10.3. Általános

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

10.4. C

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

10.5. C++

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

10.6. 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.