

# Részproblémára bontó algoritmusok (mohó, oszd-meg-és-uralkodj, dinamikus programozás), rendező algoritmusok, gráfalgoritmusok (szélességi- és mélységi keresés, minimális feszítőfák, legrövidebb utak)

---

## Részproblémára bontó algoritmusok

---

### Oszd meg és uralkodj

1. **Felosztás:** a feladatot több részfeladatra osztjuk, a részfeladatok hasonlóak az eredeti feladathoz, de kisebbek
2. **Uralkodás:** rekurzívan megoldjuk a kisebb részfeladatokat.
3. **Összevonás:** a részfeladatok megoldásait összevonjuk, és az adja a végső megoldást.

### Példa

*Felező-csúskereső algoritmus*

**Input:** egy számsorozat

**Output:** van-e benne csúcs?

**Algoritmus:** az  $n$  méretű sorozat helyett vizsgáljunk egy  $(n-1)/2$  méretűt, és ebben keressünk csúcsot,  
majd ezt folytatjuk rekurzívan

### Dinamikus programozás

**Alapgondolat:** Mi lenne, ha a már megoldott részproblémákat nem számolnánk ki újra minden egyes alkalommal  $\Rightarrow$  eltároljuk a részproblémák megoldásait. Idő tárra cserélése

- Dinamikus programozás akkor, ha a részproblémák nem függetlenek, hanem vannak közös részeik
- így minden részproblémát csak egyszer fogunk megoldani

**Iteratív megoldás:** bottom-up építkezünk, és minden lehetséges értéket megnézünk

**Rekurzív megoldás memorizálással:** top-down építkezünk, és kulcs-érték párokat nézünk (csak akkor, ha nem kell minden részmegoldás)

### Pénzváltás probléma

**Input:**  $P_1, P_2, \dots, P_n$  típusú pénzérmék,  $F$  forint

**Output:** legkevesebb hány érmével fizethető ki pontosan  $F$  forint?

Pénzváltási feladat megoldása DP-vel: minden összegre  $F$ -ig kiszámoljuk, hogy azt minimum hány pénzérmével tudjuk kifizetni

- ötlet: minden érmére megnézzük, hogy a korábbi optimális megoldás a jó, amiben nem volt benne az az érme, vagy az, ha benne van az érme
- futásidő:  $O(Fn)$

### Mohó algoritmusok

A mohó algoritmusok **lokálisan** legjobb döntést hozzák, de NEM mindig optimális megoldás az egész feladatra. Viszont, ha adható ilyen algoritmus akkor rendkívül hatékony.

Két tulajdonság, ha megadható ilyen algoritmus:

1. **Optimális részstruktúra:** A részfeladatok is optimális megoldást adnak.
2. **Mohó választás:** Lokálisan optimális választások a globális optimális megoldáshoz vezetnek

### Mohó algoritmusok helyessége:

- fogalmazzuk meg a feladatot úgy, hogy minden döntés hatására egy kisebb részprobléma keletkezzen
- bizonyítsuk be, hogy mindig van mohó választási lehetőség, tehát biztonságos
- mohó választással olyan részprobléma keletkezik, amihez hozzávéve a mohó választást, az eredeti probléma optimális megoldását kapjuk (optimális részstruktúrák)

Egy feladat optimális részstruktúrájú, ha a probléma egy opt. megoldása tartalmazza a részfeladatok optimális megoldásait is.

## Példák

### Hátizsák probléma

- adott egy hátizsák kapacitása, és  $n$  tárgy, mindegyik értékkel és súllyal megadva
- mekkora a legnagyobb összérték, amit a hátizsákba tehetünk?

*Dinamikus prog:*

Ismétléses hátizsák probléma:

Hasonlóan mint a pénzváltásnál 1D tömb, aminek az oszlopai  $0 \dots W$ -ig.

Amennyibe ismétlés nélküli:

- felvesszünk egy *kapacitástárgyak*\* száma mátrixot és minden sor egy tárgyat képvisel.
- Kiszámoljuk a legoptimálisabb értékeket
- idő = tár =  $\mathcal{O}(N * W)$ , ahol  $n$  = tárgyak,  $w$  = kapacitás

### Töredékes hátizsák probléma

- a tárgyak feldarabolhatók
- de minden tárgyból egy darab van

*Mohó algoritmus a töredékes hátizsákra:*

- számoljuk ki minden tárgyra az érték/súly arányt
- tegyük a hátizsákba a legnagyobb ilyen arányú tárgyat, az egészet ha belefér, vagy törjük, ha nem
- idő =  $\mathcal{O}(n * \log n)$ , ahol  $n$  = tárgyak
- tár =  $\mathcal{O}(1)$

### Huffman-kódolás:

input: C ábécé és gyakoriságok

kimenet: Minimális költségű prefix-fa

Algoritmus:

- Gyakoriságokat minimumos prioritási sorba rendezi
- Majd fát épít a két felső minimális elemből, ameddig csak egy fa nem lesz.
- idő =  $\mathcal{O}(n * \log n)$

# Rendező algoritmusok

*input:* n számból álló tömb

*output:* bemenő tömb elemeinek olyan sorrendje, ahol minden következő elem nagyobbegyenlő az előzőnél

**fontossága:** sok probléma triviális, ha rendezett a bemenet (pl bináris keresés, medián megállapítás)

**Stabilitás:** hogy az azonosnak ítélt elemek közötti sorrendet megőrzi-e.

- Buborék rendezés
  - Elve, hogy egy „buborékkal” haladva a tömbben több menetben előlről hátra a buborékban szereplő két elemet felcseréljük, ha azok rossz sorrendben vannak. **Stabil rendezés.**
  - Átlagos eset:  $\mathcal{O}(n^2)$
  - Legrosszabb eset:  $\mathcal{O}(n^2)$
  - Tárigénye:  $\mathcal{O}(1)$
  - Nagy adat esetén, ahol már majdnem rendezettek az elemek. Leggyorsabb, ha extrém kicsi és közel rendezett az adat. **KB csak tanító jellegű, nem éri meg soha.**
- Beszűrő rendezés
  - folyton haladunk előre a tömbben, az aktuális elemet beszűrjük a megfelelő helyre. **Stabil rendezés.**
  - Átlagos eset:  $\mathcal{O}(n^2)$
  - Legrosszabb eset:  $\mathcal{O}(n^2)$
  - Tárigénye:  $\mathcal{O}(1)$
  - Bármikor tbh. Láncolt listák esetén a leggyorsabb
- Összefésülő rendezés
  - oszd meg és uralkodj: Felbontjuk elemi részekre a tömböt, majd végighaladva összefésüljük őket megfelelő sorrendbe. Kiválaszt egy pivot elemet és ez alapján particionálja a tömböt, ami mögé a kisebbeket, elé pedig a nagyobbakat mozgatja. **Stabil rendezés.**
  - Átlagos eset:  $\mathcal{O}(n * \log n)$
  - Legrosszabb eset:  $\mathcal{O}(n * \log n)$
  - Tárigénye:  $\mathcal{O}(n)$  vagy ha láncolt lista akkor  $\mathcal{O}(1)$
- Gyorsrendezés:
  - Rekurzív algoritmus, kettéosztja a rendezendő listát egy kiemelt elemnél kisebb és nagyobb elemekre, majd a részekre külön-külön alkalmazza a gyorsrendezést. **Nem stabil rendezés**
  - **Átlagos eset:**  $\mathcal{O}(n \log n)$

- **Legrosszabb eset:**  $\mathcal{O}(n^2)$
- **Tárigénye:**  $\mathcal{O}(\log n)$
- **Leszámláló rendezés:**
  - HA az  $n$  bemeneti elem mindegyike 0 és  $k$  közötti egész szám, ahol  $k$  egy egész.
  - 1. Vezetünk egy  $C$  tömböt, amibe belerakjuk az  $i$ -edik elem előfordulásainak számát.
  - 2. Meghatározzuk minden  $i$ -re, hogy hány olyan bemeneti elem van, amelyiknek értéke  $\leq i$  (összegzés  $C$ -n)
  - 3. Minden  $j$ -re  $A[j]$ -t beletesszük  $B$  megfelelő pozíjába, amit  $C$ ből állapítunk meg
  - **Legrosszabb eset:**  $\mathcal{O}(n + k)$
  - **Tárigénye:**  $\mathcal{O}(n + k)$
- **Számjegyes rendezés (radix):**
  - Legalacsonyabb helyiértéktől a legmagasabbig megnézzük a számot a listában, majd helyére rendezzük, leszámoló rendezéshez hasonlóan, ilyen bucketeket hozunk létre 0-9-ig és ide belerakjuk az elemeket, majd kivesszük őket, és addig csináljuk ezt loopba, amíg nyilván már nincs számjegy.
  - **Legrosszabb eset:**  $\mathcal{O}(d(n + k))$ ,  $n$  darab  $d$  jegyből álló szám, ahol a számjegyek értéke legfeljebb  $k$  értéket vehetnek fel.
  - **Tárigénye:**  $\mathcal{O}(n + k)$
  - Kicsi értékek esetén

## Gráfalgoritmusok

Egy  $G = (V, E)$  struktúrát gráfnak nevezünk, ahol:

- $V$  a csúcsok halmaza
- $E \subseteq V * V$  az élek halmaza, vagyis csúcspárok
- Egy irányítatlan gráf **összefüggő**, ha bármely két csúcs között van út.
- Egy irányított gráf **erősen összefüggő**, ha bármely két csúcs között van irányított út.
- Egy gráf **transzponáltja** az élek irányának megfordítását jelenti.

## Szélességi keresés

Feladat: Járjuk be az összes csúcsot ami egy  $s$  kezdő csúcsból elérhető. Mindeközben kiszámoljuk az elérhető csúcsok távolságát  $s$ -től

**Bemenet:** Irányítatlan vagy irányított G gráf és annak egy s csúcsa

**Kimenet:** Egy szótár, ami tartalmazza az s-ből elérhető csúcsokat és azok távolságát

### **Idő- és térkomplexitás:**

Ha  $|V|$  a csúcsok és  $|E|$  a gráf éleinek száma akkor,

**Időigénye:**  $\mathcal{O}(|V| + |E|)$

**Tárigénye:**  $\mathcal{O}(b^d)$ , ahol a kezdőponttól  $d$  távolságra lévő csúcsok. A  $b$  pedig az elágazási tényező.

### **Mélységi keresés**

Amikor egy megoldást megtalálni elégséges, nincs szükség mindre/optimálisra, pl. (ki)útkeresés

Gyökércsúcsból indulva az útkeresés/bejárás során balra lefelé tartva járja be. Ha nem tud sehova lefelé menni tovább, akkor visszalép a legalsó elágazásig és a következő utat választja.

### **Idő- és térkomplexitás:**

Ha  $|V|$  a csúcsok és  $|E|$  a gráf éleinek száma akkor,

**Időigénye:**  $\mathcal{O}(|V| + |E|)$

**Tárigénye:**  $\mathcal{O}(|V|)$  VAGY  $\mathcal{O}(bd)$ , ahol a kezdőponttól  $d$  távolságra lévő csúcsok. A  $b$  pedig az elágazási tényező.

### **Erősen összefüggő komponensek**

A gráfban azok a **maximális csúcsalmazok**, amin belül bármelyik csúcsból el lehet jutni a másikba.

#### **Meghatározása:**

- Számítsuk ki MÉLYKERES algoritmussal az  $f(u)$  elhagyási értékeket
- a G transzponált gráfra alkalmazzuk a MÉLYKERES eljárást úgy, hogy az MBEJAR eljárást  $f$  szerint csökkenő sorrendbe hívjuk
- A 2. pontban az egy mélységi feszítőfába kerülő pontokat alkotnak egy erősen összefüggő komponenst.

Tehát, van egy gráf, ha irányított akkor transzponáljuk az éleit és mélységi keresést indítunk minden olyan pontból, ami még nem tartozik sehova.

### **Minimális feszítőfák**

**Feszítőfa:** Minden csúcsot érintő, összefüggő, körmentes élhalmaz.

## Kruskal algoritmus

- Minden lépésben a legkisebb, két fát összekötő élt húzzuk be (egyesítjük egyetlen fává a két fát)
- Ha a gráf összefüggő, akkor **minimális feszítőfa megalkotására** szolgál, AMÚGY meg **minimális feszítőerdőt** hoz létre.
- **Mohó algoritmus!**
- **Algoritmus:**
  - Éleket súlyok szerint növekvőbe rendezzük
  - Ezeket megvizsgáljuk, hogy melyeket vegyük be
  - Gráfok csúcsa halmazt alkot, és csak akkor kerülnek be, ha két végpontja különböző halmazban van  $\rightarrow$  halmazegyesítés.

### Idő- és térkomplexitás:

Ha  $|V|$  a csúcsok és  $|E|$  a gráf éleinek száma akkor,

**Időigénye:**  $\mathcal{O}(|E| * \log|E|)$

**Tárigénye:**  $\mathcal{O}(|V|)$

## Prim algoritmus

Összefüggő súlyozott gráf minimális feszítőfáját határozza meg.

- minden lépésben új csúcsot kötünk be a fába
  - legolcsóbb éllel elérhető csúcsot választjuk
  - **Mohó algoritmus!**
- Sűrű gráfok esetén (sok él van) Prim előnyösebb, egyébként Kruskal.

### Idő- és térkomplexitás:

Ha  $|V|$  a csúcsok és  $|E|$  a gráf éleinek száma akkor,

**Időigénye:**  $\mathcal{O}(|E| * \log|V|)$

**Tárigénye:**  $\mathcal{O}(|V| + |E|)$

## Legrövidebb utak (csúcsból kiindulva)

Bemenet: Irányított, súlyozott gráf és egy  $s$  kezdőcsúcs.

Kimenet: Minden  $V$  csúcsához a legrövidebb út  $s$  ből.

## Dijkstra algoritmus

- azokat a csúcsokat tárolja amihez már megtalálta a legrövidebb utat
- minden lépésben egyet bővíti az elért csúcsok halmazát
- legkisebb legrövidebb úttal bíró csúcsot választja
- **Mohó algoritmus!**
- nem ad helyes megoldást negatív élsúlyok esetén (beragadhat).
- **Időigény:**  $\mathcal{O}(|V| * \log|V|)$
- Minden pontból:  $\mathcal{O}(|E| * |V| * \log|V|)$

## Bellman-Ford algoritmus

- negatív súlyok esetén is működik
- **Időigény:**  $\mathcal{O}(|V| * |E|)$
- Minden pontból:  $\mathcal{O}(|V|^2 * |E|)$

## Floyd-Warshall algoritmus (legrövidebb utak minden pontpárra)

- dinamikus programozás
  - minden egyes lépésben egyre több csúcsot használhatunk
  - **Időigény:**  $\mathcal{O}(|V|^3)$
1. Ha nincsenek negatív élsúlyok és ritka a gráf akkor **Dijkstra**
  2. Ha vannak negatív élsúlyok, de nincsenek negatív összköltségű körök vagy sűrű a gráf akkor **Floyd-Warshall**
  3. Ha negatív összköltségű körök is lehetnek akkor **Ford-Bellman**

# 2. Elemi adatszerkezetek, bináris keresőfák, hasító táblázatok, gráfok és fák számítógépes reprezentációja

---

## Elemi adatszerkezetek

---

tömb, láncolt lista, sor, verem, gráf, map, kupac - saját vélemény



## Adatszerkezet

- adatok tárolására és szervezésére szolgáló módszer
- lehetővé teszi a hatékony hozzáférést és módosításokat

## Leggyakoribb műveletek

- *Módosító:*
  - beszúr
  - töröl
- *Lekérdező:*
  - keres
  - min
  - max
  - előző
  - következő

Megfelelő adatszerkezet kulcs az implementáció futásidejéhez!

## Listák

Az adatok lineárisan követik egymást.

Egy érték többször is előfordulhat.

**Műveletek:** érték, értékad, keres, beszúr, töröl

### 1. Közvetlen elérés

- minden index közvetlen elérésű, közvetlenül írható/olvasható
- érték:  $O(1)$ , keres:  $O(n)$
- beszúr és töröl esetén **változik a méret**, át kell másolni az elemeket egy új helyre
- beszúr:  $O(n)$ , töröl:  $O(n)$
- ötlet: ha tele van a tömb, **duplázzuk meg a kapacitást**
- ha negyedére csökken, **felezzük a kapacitást**
- így nem kell mindig az egész tömböt másolni

### 2. Láncolt lista

minden érték mellé mutatókat tárolunk a következő/megelőző elemre.

- **egyszeresen láncolt:** következő elemre mutat
- **kétszeresen láncolt:** előző és következőre is mutat

- **ciklikus:** az utolsó az első elemre mutat
- **őrszem:** egy nil elem, ami a lista elejére (fej) mutat

Közvetlen elérés vs Láncolt lista

- **Közvetlen elérés:** ÉRTÉK() konstans, módosító lassú
- **Láncolt lista:** ÉRTKÉ() lassú, módosító gyors, sok memória kell a mutatóknak

## Verem és sor (Stack, Queue)

Olyan listák, ahol a beszúrás és a törlés csak adott pozíción történhet

- **verem:** legutoljára beszúrt elemet vehetjük csak ki (**LIFO - Last In First Out**)
- **sor:** legkorábban beszúrt elemet vehetjük csak ki (**FIFO - First In First Out**)

Verem műveletek

- **push:** verem tetejére rakunk egy elemet
- **pop:** verem tetejéről levesszük
- $n$  verem méret esetén
  - Elérési idő:  $O(1)$ , de csak a verem tetején lévő elemet tudjuk elérni (**pop**)
  - Beszúrás:  $O(1)$ , mert mindig a tetejére pakolunk (**push**)
  - Törlés:  $O(1)$ , de csak a tetején lévő elemet tudjuk törölni (**pop**)

Sor műveletek:

- **enqueue:** sor végére rakunk
- **dequeue:** sor elejéről elveszünk
- $n$  sor méret esetén legrosszabb esetben:
  - Elérés:  $O(n)$
  - Beszúrás:  $O(1)$
  - Törlés:  $O(1)$

## Prioritási sor és kupac

### Prioritási sor

- érkezés sorrendje lényegtelen, mindig a min/max elemet akarjuk kivenni

lehet mondjuk listával megvalósítani, veremmel vagy sorral nem érdemes, mert nem számít a sorrend

Prioritási sor hatékony megvalósítása: **kupac (heap)**

## Kupac

- majdnem teljes bináris fa, minden csúcsa legalább akkora, mint a gyerekei -> max elem a gyökérben

## Bináris keresőfák

---

Keres, beszúr, töröl, min, max, következő, előző

**Mind legyen  $\mathcal{O}(\log n)$**

Bináris keresőfa

- minden csúcsnak max két gyereke van
- balra vannak a kisebb elemek
- jobbra a nagyobbak
- keresés  **$\mathcal{O}(h)$**  idejű ( $h$  a fa magassága)
- min/max is  **$\mathcal{O}(h)$** : vagy teljesen jobbra, vagy teljesen balra kell lemennünk
- következő/előző szintén  **$\mathcal{O}(h)$**  - amíg jobb/bal gyerek, addig megyünk max
- beszúr szintén  **$\mathcal{O}(h)$**  - mindig levélként szúrunk be, úgy, hogy kb megkeressük a helyét
- töröl is  **$\mathcal{O}(h)$** , levelet simán törölünk, egy gyerekeset úgy, hogy a gyereket linkeljük a szülőhöz, két gyerekeset pedig a **következővel** helyettesítjük

## EZ NEM TUDOM KELL-E SZABIVANHOZ LEHET

### AVL fák:

- AVL-fa, ha  $T$  egy nemüres bináris fa:
  - $T_L$  és  $T_R$  magasság-kiegyensúlyozottak
  - $|h_L - h_R| \leq 1$ , ahol  $h_L$  és  $h_R$  rendre a  $T$  left és right magasságai.
  - **Egyensúlyfaktora** a  $h_L - h_R$ . Ez mindegyik -1, 0 vagy 1 lehet, ellenben forgatni kell.

Minden művelete szinte:  $\mathcal{O}(\log n)$

# Hasító táblák

---

## Halmaz és szótár

### Halmaz

- egy elem legfeljebb egyszer szerepelhet benne
- keres helyett **tartalmaz-e**
- beszúr, töröl
- metszet, unió

### Szótár

- kulcs érték párok halmaza
- minden kulcs legfeljebb egyszer szerepelhet
- egy érték tetszőleges számban előfordulhat

### Asszociatív tömb

- egészek helyett bármilyen típussal indexelhetünk

### Map

- kulcs->érték leképezés

## Hasítótáblák

- Halmazok és szótárak hatékony megvalósítása
- Keres, beszúr, töröl legyen hatékony
  - Átlagos esetben:  $\mathcal{O}(1)$

Hasítótábla olyan szótár, amikor egy hash függvény segítségével állapítjuk meg, hogy melyik kulcshoz milyen érték tartozzon

példa:  **$h(k) = k \bmod m$**

ahol  $m$  a hasító tábla mérete

lehetnek ütközések! **cél: az ütközések minimalizálása**

## Ütközések minimalizálása

## Láncolt listás megoldás:

1. Az adott cellában egy láncolt listát tartunk számon
2. A rövid láncok a legjobbak
3. **Load factor:** vödörök száma / elemek száma, ha ez túl nagy akkor több vödör és újrahashelés.

## Nyílt címzés

Listák helyett tömbben "egymás után" tároljuk a megegyező hasított értékű elemeket.

→ Nincs szükség mutatókra.

### Lineáris próba:

- Addig próbálgatjuk berakni a tömbbe, amíg nem látunk üres helyet, mindig 1-et lépünk előre.
- $H(k, i, n) = H(k, n) + i \pmod{n}$ , ahol  $k$  a kulcs,  $n$  a tábla mérete,  $i$  a kipróbált hely.

### Négyzetes/Quadratikus próba:

- Két kipróbált hely között a távolságot egy másodfokú polinom adja.
- $H(k, i, n) = H(k, n) + c_1 i + c_2 i^2 \pmod{n}$ , ahol  $k$  a kulcs,  $n$  a tábla mérete,  $i$  a kipróbálás helye,  $c_1$  és  $c_2$  pedig egy valami függvényre jellemző konstans.

# Gráfok és fák számítógépes reprezentációja

---

## Gráfok reprezentációja

### 1. Szomszédsági mátrix

- minden csúcshoz hozzárendelünk egy számot
- ha  $a$  és  $b$  között van él, akkor  $\text{matrix}[a][b] = 1$  és  $\text{matrix}[b][a] = 1$
- ha nincs, akkor 0

### 2. Szomszédsági lista

- minden listaelem egy csúcs, ami szintén egy lista
- minden csúcshoz tartozó listában tároljuk a vele szomszédos csúcsokat

## Fák reprezentációja:

- Csúcsokat és éleiket reprezentáljuk
- Maga a fa egy mutató a gyökérre.

#### 1. gyerek éllista

- Kulcs
- Szülő
- Gyereklista

#### 2. Első fiú, apa, testvér

- Kulcs
- Szülő
- Első gyerek
- Testvér

#### 3. Bináris fa

- Kulcs
- Szülő
- Jobb és bal gyerek.

## 3. Hatékony visszavezetés. Nemdeterminizmus. A P és NP osztályok. NP-teljes problémák

---

### Alapvető információk:

---

**Időigény:** Adott futásidejű algoritmus adott számítási kapacitású architektúrán mekkora inputra fut le.

**Az input mérete:** Az  $a_1, \dots, a_n$  input **mérete** az  $a_i$  számok **bináris reprezentáció** hosszának összege.

**A gép időkorlátja:**  $M$  Ram gép időkorlátja az  $f(n) : \mathbb{N} \rightarrow \mathbb{N}$  függvény, ha tetszőleges  $n$  méretű inputon legfeljebb  $f(n)$  lépésben megáll.

### Hatékony visszavezetés

---

**Visszavezetésnek** nevezzük azt, mikor ha van egy problémánk, amit nem tudjuk, hogy kéne megoldanunk, és egy problémánk, amit tudjuk hogy oldjunk meg, és a nem ismert probléma inputjaiból

elkészítjük az ismert probléma egy inputját, és így oldjuk azt meg.

Hatékonyak akkor nevezhetjük, ha ez az **inputkonverzió polinomidejű**. Ezt Turing-visszavezetésnek is hívják.

### Formailag:

Legyenek  $A$  és  $B$  eldöntési problémák, azt mondjuk, hogy  $A$  (**hatékonyan**) visszavezethető  $B$ -re, ha van olyan  $f$  (**polinomidejű**) inputkonverzió, ami:

-  $A$  inputjaiból  $B$  inputjait készíti

- **Tartja a választ:**  $A(x) = B(f(x))$

**Jele:**  $A \leq_p B$  ( $B$  legalább olyan nehéz mint  $A$ )

## Nemdeterminizmus

---

Egy algoritmus *nemdeterminisztikus*, ha egy ponton úgymond szétválík a futása, és többféle eredménye is lehet a futás végére.

## P és NP osztályok

---

A  $P$  osztályban azok a problémák vannak, amelyek determinisztikusan polinomidőben megoldhatók.

Az  $NP$  osztályban azok a problémák vannak, amelyek nemdeterminisztikusan polinomidőben megoldhatók.

## NP teljes problémák

---

### Nehézség, teljesség:

$A$  egy **probléma**  $C$  pediga problémák egy **osztálya**

1. **C-nehéz:** Minden  $C$ -beli probléma visszavezethető  $A$ -ra
2. **C-teljes:**  $A$  probléma ráadásul  $C$ -ben van

Egy probléma akkor *NP*-teljes, ha *NP*-beli és *NP*-nehéz.

- **NP-beli**, ha nemdeterminisztikusan tudunk tanúkat generálni hozzá, amik igen példányai a problémának.
- **NP-nehé**, ha minden más *NP*-beli problémát hatékonyan vissza tudunk vezetni rá.
- **NP-teljes**, Vegyünk egy ismert *NP*-teljes problémát és **vezessük ezt** az új problémára vissza

## Példák

Pár képen ott van.

SAT, Hátizsák, Hamilton-út, Hamilton-kör, Euler-kör, ILP, Részletösszeg, Partíció

# 4. A PSPACE osztály. PSPACE-teljes problémák. Logaritmikus tárigényű visszavezetés. NL-teljes problémák

## Alapvető információk:

**Időigény:** Adott futásidejű algoritmus adott számítási kapacitású architektúrán mekkora inputra fut le.

**Az input mérete:** Az  $a_1, \dots, a_n$  input **mérete** az  $a_i$  számok **bináris reprezentáció** hosszának összege.

**A gép időkorlátja:**  $M$  Ram gép időkorlátja az  $f(n) : \mathbb{N} \rightarrow \mathbb{N}$  függvény, ha tetszőleges  $n$  méretű inputon legfeljebb  $f(n)$  lépésben megáll.

## Tárigény-elemzés

1. Input read-only? (**int: akárhány bites egész**)

$K$  a max értéke  $\Rightarrow \log K$  bit kell hozzá.



- Igen: Nem kell vele számolni, a hívónak kell lefoglalnia
- Nem: Akkor számolnunk kell vele.

## 2. Lokális változók?

- A lokális változók mekkora értéket vesznek fel és ezeket össze kell adni.

# PSPACE osztály = $Space(n^k)$

---

Polinom tárban (det. vagy nemdet.) eldönthető problémák osztálya.

**Savitch-tétel:** Az  $f(n)$  tárban nemdeterminisztikusan eldönthető problémák mind eldönthetők determinisztikusan,  $f^2(n)$  tárban is

Tehát:  $NSPACE(f(n))$  részhalmaza  $SPACE(f^2(n))$ -nek és mivel polinom négyzete polinom  $\rightarrow$

PSPACE = NPSPACE

Elérhetőség eldönthető  $O(\log^2 n)$  tárban (A Savitch algoritmussal!).

**Elérhetőségi módszer** használta bizonyításnak:

Szeretnénk szimulálni egy nemdet algoritmust determinisztikus módon. Ezt a RAM-gép konfigurációs gráfiával tesszük (ahol a csúcsok az állapotok, az élek a lehetséges elérhető állapotok – working regiszterek+aktuális kódsor).

Egy  $O(f(n))$  tárigényű nemdet, offline programnak egy  $n$  méretű inputon elindítva  $k^{f(n)}$  konfigurációja lehet.

Ezen lefuttatva az  $O(\log^2 n)$  tárigényű elérhetőségi algoritmust kijön, hogy determinisztikus módon szimuláltunk egy nemdet algoritmust.

A Savitch algoritmust futtatva ezen a tárigénye:

$$\log^2(k^{f(n)}) = (O(f(n) * \log k))^2 = O((f(n))^2)$$

A LÉNYEG: NEMDET ALGORITMUST SZIMULÁLUNK DET. MÓDON, A TÁRIGÉNY CSAK NÉGYZETRE EMELŐDIK. POLINOM A NÉGYZETEN STILL POLINOM SZÓVAL

PSPACE = NPSPACE

## PSPACE-teljes problémák

---

**Nehézség, teljesség:**

A egy **probléma**  $C$  pedig a problémák egy **osztálya**

1. **C-nehéz:** Minden  $C$ -beli probléma visszavezethető  $A$ -ra
2. **C-teljes:**  $A$  probléma ráadásul  $C$ -ben van

QSAT PSPACE-teljes

QSAT (kvantifikált SAT)

- *Adott:* adott egy ítéletkalkulusbeli logikai formula, változó kvantorokkal az elején (létezik, bármely, létezik, bármely stb), **magja CNF alakú, kvantormentes**
- *Kérdés:* igaz-e ez a formula?

### QSAT mint kétszemélyes játék

- input ugyanaz
- van-e az első játékosnak nyerő stratégiája abban a játékban, ahol:
  - a játékosok sorban értéket adnak a változóknak, első játékos  $x_1$ -nek, második  $x_2$ -nek stb
  - ha a formula igaz lesz, az első játékos nyert, ha hamis, akkor a második
- ez ugyanaz tkp, mint a sima QSAT, szóval ez is PSPACE-teljes

hasonlít a minimaxra

az éses csúcsoknál lévő játékos minimalizál

### Földrajzi játék

- adott egy irányított gráf és egy kijelölt kezdőcsúcs
- az első játékosnak van-e nyerő stratégiája?
  - az első játékos kezd, lerakja a bábút a kezdőcsúcsra
  - felváltva lépnek
  - egy olyan csúcsba kell húzni a bábút, ami egy lépésben elérhető, és ahol még nem voltak
  - aki először nem tud lépni, veszített

Földrajzi játék PSPACE-teljes

Adott két reguláris kifejezés, igaz-e, hogy ugyanazokra a szavakra illeszkednek?

Adott két nemdet automata, ekvivalensek-e?

Adott, egy SOKOBAN/RUSH HOUR feladvány, megoldható-e?

## Logtáras visszavezetés = $L = \text{Space}(\log n)$

---

Polinomidejű visszavezetés túl erős, ha pl P-beli problémákat akarunk egymáshoz viszonyítani, mert egy polinomidejű visszavezetés alatt már akár meg is oldhatnánk egy

P-beli problémát

Logtáras visszavezetés

Jele:  $A \leq_l B$ .

Ha  $f$  egy olyan függvény, hogy

- A inputjaiból B inputjait készíti
- választartó módon
- és logaritmikus térben kiszámítható

akkor  $f$  egy logtáras visszavezetés A-ról B-re.

## NL-teljes problémák = NSpace(log $n$ )

---

Nemdeterminisztikus logtáras problémák

Elérhetőség

1. Adott: egy  $G = (V, E)$  irányított gráf. Feltehetjük, hogy  $V = \{1, 2, \dots, n\}$ .
2. Kérdés: létezik-e 1-ből  $n$ -be vezető irányított út?

Nemdeterminisztikus módon választunk 1 és  $n$  között csúcsot és mivel az inputot olvasni kell, outputra nem írunk semmit, csak két változót tartunk számon, amibe csak  $1 \dots n$  vannak számok így logtáras lesz.

### Immerman-Szelepcsényi tétel:

Van olyan **nemdeterminisztikus** algoritmus, mely **logaritmikus térben** kiszámítja az input gráf megadott csúcsából elérhető csúcsok számát.

Ezt felhasználva egy ND program konfigurációs gráfján:

A konfigurációs gráfban először  $nd$  kiszámítjuk az elérhető konfigurációk számát, aztán ciklusban mindet nemdeterminisztikusan megpróbáljuk elérni, számoljuk, hogy mennyit sikerült. Ha annyit értünk el, amennyi csak elérhető és egyik sem elfogadó, akkor Accept, különben Reject.

Vagyis megadtuk a nemdet algoritmus komplementerét.

következmény:

- $NSPACE(f(n)) = coNSPACE(f(n))$ .
- $NL = coNL$

TEHÁT NEMDET PROBLÉMÁK KOMPLEMENTERE MEGOLDHATÓ

UGYANOLYAN TÁRBAN.

**Egyéb infók:**

$L \subseteq NL$  (részhalmaza, vagy egyenlő vele)

Elérhetőség megoldható **lineáris** tárban: Szélességi keresés algoritmus egy  $N$ -csúcsú gráf esetén két, egyenként legfeljebb  $N$  méretű csúcshalmazt tárol. Ezt pl egy  $N$  hosszú bitvektor alkalmazásával egy-egy regiszterben  $O(N)$  tárban megoldható.

## 5. Véges automata és változatai, a felismert nyelv definíciója. A reguláris nyelvtanok, a véges automaták és a reguláris kifejezések ekvivalenciája. Reguláris nyelvekre vonatkozó pumpáló lemma, alkalmazása és következményei

---

### Alapfogalmak, jelölések

---

**Ábécé:** Szimbólumok egy tetszőleges véges, nemüres halmaza jele:  $\Sigma$

- $\Sigma^*$ : Az összes szavak +  $\epsilon$
- $\Sigma^+$ : Az összes szavak, kivéve az  $\epsilon$
- $\Sigma$  **ábécé feletti szó**: egy  $a_1, \dots, a_k$  alakú sorozat.
- **Nyelv**:  $\Sigma^*$  tetszőleges  $L$  részhalmazát egy  $\Sigma$  feletti **nyelvnek** nevezzük. Ha  $L$  véges számú szóból áll, akkor **\*véges nyelvnek** nevezzük
- **Nyelvtan**: Könnyen leírható eszköz, amely alkalmas nyelvek megadására. Effektíve nyelveket tuduk vele reprezentálni.
  - PL Környezetfüggetlen:  $G = (N, \Sigma, P, S)$ , ahol
    - $N$  egy ábécé
    - $\Sigma$  egy ábécé, terminális
    - $S \in N$  kezdő szimbólum
    - $P$  pedig egy  $A \rightarrow \alpha$  alakú átírási szabály.

# Véges automata és változatai, a felismert nyelv definíciója

---

Lásd pdf

Egy nemdeterminisztikus automata determinizálása:

**Állapotai száma max  $2^n$  lesz.**

- Elindulunk a kezdőállapotból és megnézzük, hogy az első betű hatására hova megy  
-> ezeket összevonjuk egy állapottá és oda vezetjük ezzel a betűvel.
- Ezután az új összevont állapot részeit nézzük meg, hogy onnan a betűk hova mennek.

Egy  $\epsilon$  automata  $\epsilon$ -mentesítése.

- Lezártakat számolunk.
  - Azokat az állapotokat, ahova átlehet jutni  $\epsilon$  átmenettel azokat egy lezártba vesszük.

## A reguláris nyelvtanok, a véges automaták és a reguláris kifejezések ekvivalenciája

---

### Reguláris nyelvtanok

- $N$  : nemterminális abc
- $\Sigma$ : terminális abc
- $P$ : szabályok halmaza
- $S$ : eleme  $N$ , kezdő nemterminális  
Egy  $G = (N, \Sigma, P, S)$  nyelvtan reguláris (vagy jobblinéaris), ha  $P$ -ben minden szabály
  - $A \rightarrow xB$  vagy
  - $A \rightarrow x$ , alakú.

Azért jobblineáris, mert minden szabály jobb oldalán max. egy nemterminális állhat, és ez mindig a szó végén helyezkedik el. Levezetést csak  $A \rightarrow x$  alakú szabállyal fejezhetünk be, ahol  $x \in \Sigma^*$ . A reguláris nyelvtanok speciális környezetfüggetlen nyelvtanok.

Példa:  $S \rightarrow aaS|abS|baS|bbS|\epsilon$ , vagyis a páros hosszú szavakat generáló nyelvtan.

## Reguláris kifejezések

**Veszünk egy abc-t, és hozzáveszünk néhány szimbólumot, ezekből építünk reguláris kifejezéseket.**

A szigma feletti reguláris kifejezések halmaza a  $(\Sigma \cup \emptyset, \epsilon, (, ), +, *)^*$  halmaz legszűkebb olyan  $U$  részhalmaza, hogy

- $\emptyset, \epsilon$  eleme  $U$ -nak
- minden  $a$  eleme  $\Sigma$  eleme  $U$ -nak
- ha  $R_1, R_2$  eleme  $U$ , akkor  $R_1+R_2, R_1R_2, R_1^*$  is eleme  $U$ -nak

Prioritási sorrend:  $*$ , konkatenáció,  $+$

### Jelentések:

- **$|R|$ , az  $R$  által reprezentált nyelv**
- $R = \emptyset, |R| = \emptyset$ , azaz az üres nyelv
- $R = \epsilon, |R| = \{\epsilon\}$ , azaz az epszilon szimbólum önmagában, mint nyelv
- $R = a, |R| = \{a\}$ , azaz az  $a$  szimbólum önmagában, mint nyelv
- $R = R_1+R_2, |R| = |R_1| \cup |R_2|$ , azaz a két regex által generált nyelv uniója
- $R = R_1R_2, |R| = |R_1||R_2|$ , azaz a két regex által generált nyelv konkatenációja
- $R = R_1^*,$  akkor  $|R| = |R_1|^*$ , azaz a regex által generált nyelv iterációja, az összes szó összekonkatenálva egy másik nyelvbéli szóval az összes lehetséges módon

## Ekvivalencia

Tetszőleges  $L \subseteq \Sigma^*$  nyelv esetén a következő három állítás ekvivalens:

- 1.  $L$  generálható reguláris nyelvtannal
- 2.  $L$  felismerhető automatával
- 3.  $L$  reprezentálható reguláris kifejezéssel

### 3 -> 1:

*Ha  $L$  reprezentálható reguláris kifejezéssel, akkor generálható reguláris nyelvtannal.*

Bizonyítás:  $L$ -et reprezentáló  $R$  reguláris kifejezés struktúrája szerinti indukcióval.

- $R = \emptyset$ : ekkor  $L = |R| = \emptyset$ , ekkor  $L$  generálható a  $G = (N, \Sigma, \emptyset, S)$  nyelvtannal.
- $R = a$ :  $a$  eleme  $\Sigma$ , vagy  $a = \epsilon$ , ekkor  $L = |R| = \{a\}$ , ami generálható a  $G = (N, \Sigma, \{S \rightarrow A\}, S)$  nyelvtannal
- INDUKCIÓ  $R = R_1 + R_2$ , ekkor  $L = |R| = L_1 \cup L_2$ ,  $L_1 = |R_1|$ ,  $L_2 = |R_2|$ 
  - ekkor tegyük fel, hogy  $L_i$  generálható egy  $G_i = (N_i, \Sigma, P_i, S_i)$  ( $i=1,2$ ) reguláris nyelvtannal
  - ekkor  $L$  generálható egy  $G = (N_1 \cup N_2 \cup \{S\}, \Sigma, P_1 \cup P_2 \cup \{S \rightarrow S_1, S \rightarrow S_2\}, S)$  nyelvtannal, ahol  $S$  egy új szimbólum
  - aka vesszük az összes nemterminálist, az  $abc$ -t, az összes korábbi szabályt
  - továbbá egy új kezdőszimbólumot
  - és az új kezdőszimbólumból elérhető a régi kettő kezdőszimbólum, aka kiválaszthatjuk, melyik nyelvből származó szót akarjuk generálni
  - ahhoz, hogy elérhető legyen a régi két kezdőszimbólum, felveszünk két új szabályt értelemszerűen a régiek közé
- INDUKCIÓ  $R = R_1 R_2$ , ekkor  $L = |R| = L_1 L_2$ ,  $L_1 = |R_1|$ ,  $L_2 = |R_2|$ 
  - ekkor tegyük fel, hogy  $L_i$  generálható egy  $G_i = (N_i, \Sigma, P_i, S_i)$  ( $i=1,2$ ) reguláris nyelvtannal
  - Akkor  $L$  generálható a  $G = (N_1 \cup N_2, \Sigma, P, S_1)$  nyelvtannal, ahol  $P$ :
    - bele vesszük az összes szabályt az első nyelvet generáló nyelvtanból, a befejező szabályok végére odaírjuk a második nyelvtan kezdőszimbólumát
    - a második nyelvtan összes szabályát is bele vesszük
- INDUKCIÓ  $R = R_1^*$ , ekkor  $L = |R| = L_1^*$ , ahol  $L_1 = |R_1|$ 
  - ekkor tegyük fel, hogy  $L_1$  generálható egy  $G_1 = (N_1, \Sigma, P_1, S_1)$  nyelvtannal
  - ekkor  $L$  generálható egy  $G = (N_1 \cup \{S\}, \Sigma, P, S)$  nyelvtannal, ahol  $S$  egy új szimbólum
  - a szabályokat úgy módosítjuk, hogy:
    - $S$ -ből elérhető az üresszó és az eredeti kezdőszimbólum
    - a nem "befejező" szabályokat fel vesszük úgy, ahogy voltak
    - a "befejező" szabályok jobb oldalára odaírjuk az  $S$ -t

### 1 -> 2:

*Ha  $L$  nyelv reguláris, akkor felismerhető automatával.*

Bizonyítás: legyen  $L$  egy reguláris nyelv, és  $L = L(G)$ , ahol  $G$  egy reguláris nyelvtan.

Minden  $G = (N, \Sigma, P, S)$ , reguláris nyelvtanhoz megadható vele ekvivalens  $G' = (N', \Sigma, P', S)$  reguláris nyelvtan, úgy hogy  $P'$ -ben minden szabály  $A \rightarrow B$ ,  $A \rightarrow aB$  vagy  $A \rightarrow \varepsilon$  alakú, ahol  $A, B \in N'$  és  $a \in \Sigma$ .

Bizonyítás

- amelyik szabály már alaphoz ilyen alakú, azokat felvesszük  $P'$ -be
- az  $A \rightarrow a_1 a_2 a_3 \dots a_n B$  alakú szabályokat szétdaraboljuk
  - lesz belőle  $A \rightarrow a_1 A_1$ ,  $A_1 \rightarrow a_2 A_2 \rightarrow \dots \rightarrow A_{n-1} \rightarrow a_n B$
- az  $A \rightarrow a_1 a_2 a_3 \dots a_n$  szabályokat feladarboljuk
  - lesz belőle  $A \rightarrow a_1 A_1$ ,  $A_1 \rightarrow a_2 A_2 \rightarrow \dots \rightarrow A_n \rightarrow \varepsilon$

az új nemterminálisokat felvesszük  $N$ -be

Minden olyan  $G = (N, \Sigma, P, S)$  reguláris nyelvtanhoz, melynek csak  $A \rightarrow B$ ,  $A \rightarrow aB$  vagy  $A \rightarrow \varepsilon$  alakú szabályai vannak megadható olyan  $M = (Q, \Sigma, \delta, q_0, F)$  nemdeterminisztikus  $\varepsilon$ -automata, amelyre  $L(M) = L(G)$ .

Bizonyítás

- $Q = N$ , azaz a nemterminálisokból lesznek az állapotok
- $q_0 = S$ , a kezdőszimbólumból lesz a kezdőállapot
- azokból a  $B$  nemterminálisokból lesz végállapot, amikből van  $B \rightarrow \varepsilon$  szabály
- minden  $A \rightarrow aB$  kinézetű szabályból pedig legyen egy átmenet  $A$ -ból  $B$ -be a hatására

## 2 -> 3:

*Minden automatával felismert nyelv reprezentálható reguláris kifejezéssel. (Kleene tétele)*

Bizonyítás: legyen  $L = L(M)$ , ahol  $M$  egy determinisztikus automata. Megadunk egy olyan reguláris kifejezést, ami  $L$ -et reprezentálja.

Tételezzük fel, hogy  $Q = \{1, 2, 3, \dots, n\}$ , és  $q_0 = 1$ .

Minden 0 kisebbegyenlő  $k$ , azaz  $k$  darab állapot, és 0 kisebbegyenlő  $i, j$  kisebbegyenlő  $n$  esetén definiáljuk az  $L^{(k)}_{i,j}$  nyelvet a következőképpen:

Nézzük úgy az automatát, mintha az  $i$ . állapotból indulnánk, és a  $j$ -be akarnánk eljutni, de csak az  $\{1, \dots, k\}$  állapotokat érinthetjük. Az  $L^{(k)}_{i,j}$  nyelv azokat a szavakat tartalmazza, amelyeket ez a "kisebb" automata felismer.

Ezután vegyük észre azt, hogy az  $L$  nyelv tulajdonképpen úgy áll elő, hogy venni kell az összes olyan  $L^{(n)}_{1,j}$  nyelvet, ahol  $j$  egy végállapot!



Tehát vegyük az összes olyan nyelvet, ahol az első állapotból akarunk elindulni, és az utolsóba eljutni, és használhatjuk az összes állapotát  $M$ -nek (mind az  $n$  darabot). Tehát ezen  $L^{(n)}_{1,j}$  nyelvek uniója lesz  $L$ .

Ezért elég az  $L^{(n)}_{1,j}$ -ket reprezentáló reguláris kifejezéseket megadnunk ( $R^{(n)}_{1,j}$ ).

Ehhez meg kell adnunk az  $R^{(k)}_{i,j}$  reguláris kifejezéseket  $k$  szerinti indukcióval.

$k=0$  azt jelenti, hogy 0 közbülső állapotból kell eljutnunk az  $i$  állapotból a  $j$  állapotba. Ez lehet úgy, hogy valamilyen szimbólum hatására átmegyünk, vagy ha  $i=j$ , akkor hurokkellet helyben maradunk, vagy epszilonnal nem csinálunk semmit.

Az indukciós feltevésünk az, hogy minden  $i,j$ -re megadtuk az  $R^{(k)}_{i,j}$ -t

$k+1$ -hez ÉSZREVESSZÜK (ja ugye tök triviális lmao)

$L^{(k+1)}_{i,j}$  egyenlő azzal, hogy

- vagy amúgyis eljutunk  $k$  köztes állapottal is  $i$ -ből  $j$ -be
- vagy bele vesszük a  $k+1$ . állapotot is a levesbe, elmegyünk az 1-estől a  $k+1$ . állapotba, ott körözünk akármeddig, és utána  $k+1$ -ből pedig eljutunk  $j$ -be

Ekkor, mivel az  $L^{(k)}_{i,j}$  nyelvekhez az indukciós feltevés miatt tudtunk megfelelő regexet adni, ezekre elvégezve az előző azonosságot, megkapjuk az  $R^{(k+1)}_{i,j}$ -t is, ezzel pedig meg tudjuk kapni az összes regexet, ami a kezdőállapotból a végállapotokba visz, ezeket uniózva pedig az egész nyelvhez tartozó regexet.

## Reguláris nyelvekre vonatkozó pumpáló lemma, alkalmazása és következményei

---

### Pumpáló lemma

Minden  $L$  reguláris nyelv esetén megadható egy  $L$ -től függő  $k > 0$  szám, melyre minden  $w \in L$  esetén, ha  $|w| \geq k$ , akkor van olyan  $w = w_1 w_2 w_3$  felbontás, hogy

- $0 \leq |w_2|$  és  $|w_1 w_2| \leq k$
- minden  $n \geq 0$   $w_1 w_2^n w_3 \in L$

Fordítva, ha egy nyelvhez nem adható meg ilyen  $k$  szám, akkor a nyelv nem reguláris.

Kb a lényeg, hogy ha egy nyelv reguláris, akkor a k-nál hosszabb szavak felbonthatók három részre, és a középső rész ismétlődhet akármeddig

## Alkalmazása

PI bebizonyíthatjuk vele egy nyelvről, hogy nem reguláris.

$a^n b^n, n \geq 0$  nyelv nem reguláris

tegyük fel, hogy van ilyen k, amivel felbontható

a feltételek miatt  $w_2$ -ben csak a betűk vannak

ha ezt pumpáljuk, több a betű lesz benne, mint b - rossz

## Következményei

Van olyan környezetfüggetlen nyelv, amelyik nem reguláris.

1. Minden reguláris nyelvtan környezetfüggetlen. (A REG nyelvek speciális CF nyelvek-jobblinéarisak)
2. CF - REG  $\neq$  Üreshalmaz, mivel pl az  $\{a^n b^n \mid n \geq 0\}$  nyelv CF beli ( $S \rightarrow aSb \mid \epsilon$ ), viszont nem teljesül rá a pumpáló lemma  $\rightarrow$  nem REG nyelv!

# 6. A környezetfüggetlen nyelvtan és nyelv definíciója. Derivációk és derivációs fák kapcsolata. Veremautomaták és környezetfüggetlen nyelvtanok ekvivalenciája. A Bar-Hillel lemma és alkalmazása

---

## Alapfogalmak, jelölések

---

**Ábécé:** Szimbólumok egy tetszőleges véges, nemüres halmaza jele:  $\Sigma$

- $\Sigma^*$ : Az összes szavak +  $\epsilon$
- $\Sigma^+$ : Az összes szavak, kivéve az  $\epsilon$
- $\Sigma$  **ábécé feletti szó**: egy  $a_1, \dots, a_k$  alakú sorozat.
- **Nyelv**:  $\Sigma^*$  tetszőleges  $L$  részhalmazát egy  $\Sigma$  feletti **nyelvnek** nevezzük. Ha  $L$  véges számú szóból áll, akkor *\*véges nyelvnek* nevezzük
- **Nyelvtan**: Könnyen leírható eszköz, amely alkalmas nyelvek megadására. Effektíve nyelveket tuduk vele reprezentálni.
  - PL Környezetfüggetlen:  $G = (N, \Sigma, P, S)$ , ahol
    - $N$  egy ábécé
    - $\Sigma$  egy ábécé, terminális
    - $S \in N$  kezdő szimbólum
    - $P$  pedig egy  $A \rightarrow \alpha$  alakú átírási szabály.

## A környezetfüggetlen nyelvtan és nyelv definíciója

---

Egy  $G = (N, \Sigma, P, S)$  **nyelvtan**, környezetfüggetlen, ha minden szabálya  $A \rightarrow \alpha$  alakú, ahol  $\alpha$  egy **terminálisokból** és **nemterminálisokból** álló szó.

Egy nyelv **környezetfüggetlen**, ha van olyan CF nyelvtan, ami őt generálja.

## Derivációk és derivációs fák kapcsolata

---

### Korlátozás nélküli deriváció

- bármely nemterminális helyére helyettesíthetünk

### Bal/jobboldali deriváció

- csak a legbal/jobboldalabbi nemterminálisba helyettesíthetünk

pl: Bal oldali

$$\underline{K} \Rightarrow \underline{T} \Rightarrow \underline{T} * F \Rightarrow \underline{F} * F$$

Vegyes:

$$\underline{K} \Rightarrow \underline{T} \Rightarrow \underline{T} * F \Rightarrow F * \underline{F}$$

### Derivációs fák:

**Mindig csak egy gyökere van**

- vagy csak a gyökérből áll
- vagy van egy epszilon gyerek
- vagy kiindul belőle  $k$  darab él, amelyek végpontjai további derivációs fák gyökerei

**Gyökere** mindig egy **terminális-** vagy egy **nemterminális szimbólum**. Az elágazások pedig megfelelnek a nyelvtan szabályainak.

Legyen  $t$  egy derivációs fa, gyökere  $X$

$t$  magassága  $h(t)$

$t$  határa  $fr(t)$  - **határ kb a levelek balról jobbra olvasva**

### Derivációs fák kapcsolata a derivációkkal

Tetszőleges  $X$  gyökerű derivációs fára és  $\alpha$  **szóra**  $X$ -ből akkor vezethető le  $\alpha$ , ha van olyan  $X$  gyökerű derivációs fa, amelyre  $fr(t) = \alpha$

## Veremautomaták és környezetfüggetlen nyelvtanok ekvivalenciája

---

### Veremautomata

Veremautomatának nevezzük azt a  $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ , ahol

- $Q$ : állapotok halmaza
- $\Sigma$ : input abc
- $\Gamma$ : verem abc
- $q_0$  eleme  $Q$ : kezdőállapot
- $Z_0$ : verem kezdőszimbólum
- $F$ : végállapotok halmaza
- $\delta$ : átmenetfüggvény

### Az átmenet a következőképpen történik:

- ha az **automata a  $q$  állapotban van**, a szimbólum érkezik és  **$Z$  van a verem tetején**, akkor átmegy a  $q_i$  állapotba, a **veremben pedig  $Z$  helyére  $\alpha_i$  kerül**.
- Az **átmenetnél az automata kiolvas egy betűt az inputból**, leveszi  **$Z$ -t a verem tetejéről**, és tetszőleges hosszú szót odaír a helyére.

Egy szó elfogadása történhet végállapotokkal, vagy üres veremmel is. Ugyanazon automatánál általában nem egyezik meg az üres veremmel és a végállapotokkal felismert

nyelv.

## Ekvivalencia

Tétel: Minden CF nyelv felismerhető PDA-val.

**Minden környezetfüggetlen nyelvtanhoz meg lehet adni veremautomatát úgy, hogy a veremautomata (üres veremmel vagy végállapottal) ugyanazt a nyelvet ismeri fel, amit a környezetfüggetlen nyelvtan generál.**

A bizonyításhoz egy környezetfüggetlen nyelvtanhoz konstruálunk egy egyállapotos nemdeterminisztikus veremautomatát, ami **üres veremmel ismeri fel a szavakat. A verem  $abc$  legyen a nemterminálisok unió terminálisok.**

Ezzel a veremautomatával szimuláljuk a nyelvtan levezetéseit. Tudjuk továbbá, hogy az üres veremmel és a végállapottal felismert nyelvek halmaza ugyanaz, így ez az állításunk igaz lesz.

Itt pedig veremautomatához adunk meg egy környezetfüggetlen nyelvtant.

Lásd pdf 2.

## Bar-Hillel lemma és alkalmazása

---

### Tulajdonképpen pumpáló lemma CF nyelvekre

Ha  $L$  egy környezetfüggetlen nyelv ( $L \subseteq \Sigma^*$ ), akkor létezik egy nyelvtől függő  $k$  szám, amire ha egy  **$L$ -beli szó hossza nagyobb  $k$ -nál**, akkor feldarabolható 5 részre, amikre a következők teljesülnek.

Ha  $L \subseteq \Sigma^*$  nyelv környezetfüggetlen, akkor

- Megadható olyan ( $L$  től függő)  $k > 0$  egész szám,
- Hogy minden  $w \in L$  szóra, ahol  $|w| \geq k$ ,
- létezik olyan  $w = w_1 w_2 w_3 w_4 w_5$  felbontás, amelyre igazak a következő állítások:
  1.  $|w_2 w_4| > 0$  és  $|w_2 w_3 w_4| \leq k$
  2. Minden  $n \geq 0$ -ra  $w_1 w_2^n w_3 w_4^n w_5 \in L$

Alkalmazás: az  $L = a^n b^n c^n \mid n \geq 1$  nyelv nem környezetfüggetlen.

Tegyük fel, hogy igen, ekkor léteznie kell olyan  $k$  számnak, amire teljesülnek a Bar-Hillel lemmában a feltételek.

Vegyük az  $a^k b^k c^k$  szót, aminek hossza  $3k \geq k$ , tehát **jó lesz fixen**.

A lemma szerint ennek létezik  $w_1 w_2 w_3 w_4 w_5$  felbontása, melyre  $w_2 w_4$  nem epszilon, és minden  $n \geq 0$ -ra  $w_1 w_2^n w_3 w_4^n w_5$  eleme a nyelvnek.

Nézzük ekkor mi lehet **w2-ben és w4-ben**! Egyik sem tartalmazhat két betűt, mert ekkor pl ha kétszer vesszük w2-t és w4-et, akkor a betűk sorrendje nem abc lesz. Tehát biztosan csak egyféle betűt tartalmaznak. Ekkor a  $w_1 w_2 w_3 w_4 w_5$  szóbal legalább egy, és legfeljebb két betű száma több, mint a többi betűé, tehát biztos nem eleme ez a szó L-nek.

## 7. Eliminációs módszerek, mátrixok trianguláris felbontásai. Lineáris egyenletrendszerek megoldása iterációs módszerekkel. Mátrixok sajátértékeinek és sajátvektorainak numerikus meghatározása

---

### Alapvető információk

---

**Diagonális mátrix:** Főátlón kívül csak 0-ák vannak

**Egységmátrix:** Jele:  $I$ , a főátlóba 1-esek többi helyen 0-ák.

**Invertálható mátrix (nem szinguláris):** Ha létezik egy  $A$  mátrix ami csak akkor invertálható, ha van egy  $B$  mátrix amelyre igaz, hogy  $AB = I_n = A^{-1}$ .

**Szinguláris mátrix:** Olyan négyzetes mátrixok, amelyek determinánsa nulla, és nem létezik inverze.

### Eliminációs módszerek

---

A lineáris egyenletrendszerek megoldására szolgáló eljárások. ( $Ax = b$ )

## Gauss-elimináció

- $Ax = b$  alakú lineáris egyenletrendszerek megoldásához tudjuk használni
- az  $Ax = b$  egyenletrendszernek pontosan akkor van egy megoldása, ha  $\det(A) \neq 0$
- ekkor  $x = A^{-1}b$ 
  - de az inverzet kiszámolni túl lassú lenne

A Gauss-eliminációval az  $A$  mátrixot felső háromszögmátrixszá alakítjuk, és ha ez sikerül, akkor abból visszahelyettesítésekkel megkaphatjuk  $x$ -et. **Műveletigénye:**  $O(n^2/2)$ .

A felső háromszögmátrixot ún. eliminációs mátrixok segítségével kapjuk meg. Egy eliminációs mátrix dolga, hogy kinullázza az  $A$  mátrix egyik oszlopában a főátló alatti elemeket. Ha az összes ilyen eliminációs mátrixot összeszorozzuk balról egymással, akkor kapjuk az  $M$  mátrixot.

Ekkor az  $M * A$  szorzás eredménye lesz a kívánt **felső trianguláris** mátrix.

## LU felbontás

Szükséges a négyzetes mátrix

Az LU felbontás lényege, hogy az  $A$  mátrixot egy alsó és egy felső háromszögmátrixra bontjuk. A Gauss eliminációhoz nagyon hasonlít, ott az **MA szorzás eredménye egy U felső trianguláris mátrix volt**. Ha mindkét oldalt megszorozzuk balról  $M^{-1}$ -gyel, akkor azt kapjuk, hogy  $A = M^{-1}U$ . Legyen  $M^{-1} = L$ , mert  $M^{-1}$  egy **alsó trianguláris** mátrix. Ezzel elvégeztük az  $A$  mátrix LU felbontását.

Ekkor az  $Ax=b$  egyenletrendszer megoldását a következőképpen kaphatjuk:

1.  $L U x = b$
2.  $L y = b$  -  $y$  egy új mesterséges változó
3.  $U x = y$  - megoldás  $x$ -re

## Cholesky felbontás

Ha az **A mátrix**

- szimmetrikus
- pozitív definit (ha minden sajátérték pozitív)
  - Ha az átlóba **csak pozitív** van akkor biztos pozitív definit

akkor felbontható a következőképpen: (Az  $LU = x$ ,ből  $U = L^T$ )

1.  $A = LL^T$  - Ez a Cholesky alak
2.  $Ly = b$  - Az  $L^T x = y$  helyettesítésével megoldjuk y-ra
3.  $L^T x = y$  - Végül az  $y$  segítségével kifejezzük az  $x$ -et

2x olyan gyors mint az LU felbontás és **numerikusan stabilis**, szóval, ha picit változtatunk az inputon akkor kicsit változik az eredmény.

## QR felbontás

$Q$ : egy **ortogonális mátrix**, tehát  $QQ^T = Q^T Q = I$ , azaz a **transzponáltja egyben az inverze** is

$R$ : egy felső háromszögmátrix

Numerikusan stabilabb ez is.

### Megoldás:

$$1. Rx = Q^T b$$

**Tétel:** Tetszőleges  $A$  négyzetes valós reguláris mátrixnak létezik az  $A = QR$  felbontása ortogonális és felső háromszögmátrixra.

### Bizonyítás:

$A^T A$  pozitív definit, így létezik  $R^T R$  Cholesky felbontása.

Legyen ekkor  $Q$  egyenlő  $A^{R-1}$ -gyel.

Igazoljuk, hogy  $Q$  ortogonális.

$$Q^T Q = (A^{R-1})^T (A^{R-1}) = (R^{-1})^T A^T A R^{-1} = (R^{-1})^T R^T R R^{-1} = I \cdot I = I$$

behelyettesítés transzponálásos azonosság  $A^T A = R^T R$  inverzek kiütik egymás

Tehát  $Q$  valóban ortogonális



# Lineáris egyenletrendszerek megoldása iterációs módszerekkel

---

**Iterációs módszerek:** Egy kezdő állapotból, minden iteráció után egyre jobb közelítést adnak a megoldásnak.

**Nagy méretű mátrixokra,** vagy ha **eliminációs módszerek eredményei kerekítési hibával terheltek**

## Jacobi iteráció

Átrendezzük úgy az egyenletrendszert, hogy a **bal oldalon egy-egy változót kifejezünk**. Minden egyenlet esetén, úgy oldjuk meg, hogy az  $i$ -edik egyenletben az  $i$ -edik változó együtthatójával osztunk, majd az  $i$ -edik tagon kívül mindegyiket kivonjuk az egyenletből:

**Formálisan:**

$$x^{(k+1)} = -D^{-1}(A - D)x^{(k)} + D^{-1}b,$$

- $D$  egy diagonális mátrix ( $A$  főátlóbeli elemeit tartalmazza)
- $D^{-1}$ -el való szorzás pont az  $i$ -edik egyenlet elosztása az  $i$ -edik együtthatóval.
- Az  $A - D$  a jobb oldalra való átvivést jelképezi.

Választunk valami **indulóvektort**, ami ilyen kezdő megoldás kb.

A vektor elemeit behelyettesítjük a jobboldalra, és ebből kapunk egy új vektort a baloldalon, ezzel folytatjuk.

Csak akkor konvergál, ha a mátrix *szigorúan diagonálisan domináns*, vagyis az összes főátlóbeli elem abszolút értéke a legnagyobb az adott sorban.

Példa:

Egyenletrendszer:

$$5x_1 + x_2 = 7$$

$$x_1 + 2x_2 = 5$$

$$x_2 + 3x_3 = 2$$

Rendezzük át az egyenletrendszert:  $x^{(k+1)} = Bx^{(k)} + c$  alakúra.

$$x_1^{k+1} = -\frac{1}{5}x_2^{(k)} + \frac{7}{5}$$

$$x_2^{k+1} = -\frac{1}{2}x_1^{(k)} + \frac{5}{2}$$

$$x_3^{k+1} = -\frac{1}{3}x_2^{(k)} + \frac{2}{3}$$

Választunk egy kezdővektort:  $x^{(0)} = (1 \ 1 \ 1)^T$

Ezután visszairjuk mátrixos alakra.

MELLÉKLET KÉPBE (JacobiMatrix.JPG)

## Gauss-Seidel iteráció

Ugyanaz, mint a Jacobi, csak ha már **egy változó új értékét kiszámoltuk**, akkor a következő sorokban már azt az **új értéket használjuk**.

- A Gauss-Seidel gyorsabban konvergál a megoldáshoz, mint a Jacobi

## Mátrixok sajátértékeinek és sajátvektorainak numerikus meghatározása

---

### Sajátérték, sajátvektor

Legyen  $A$  egy négyzetes mátrix.

$$Ax = \lambda x$$

$x$  a **sajátvektor**,  $\lambda$  a **sajátérték**

A sajátérték olyan szám, amivel ha megszorozzuk a hozzá tartozó sajátvektort, akkor ugyanazt az eredményt kapjuk, mintha azt a vektort a mátrixszal szoroztuk volna meg.

**Meghatározása:**  $\det(A - \lambda I) = 0$

tehát, a főátló minden eleméből kivonunk lambdát, és ennek a mátrixnak keressük a determinánsát

ez egy polinomot fog eredményezni, amiben lambdák a változók, és ennek a **polinomnak a gyökei** lesznek a **sajátértékek**.

Ezt a polinomot nevezzük a mátrix **karakterisztikus polinomjának**.

 sajátérték

Valós mátrixnak is lehetnek komplex sajátértékei!

A mátrix sajátértékeinek a halmazát a mátrix *spektrumának* hívjuk.

## Hatványmódszer

A hatványmódszer a legnagyobb abszolútértékű sajátérték meghatározására szolgál. Iterációs módszer.

$$y^{(k)} = Ax^{(k)}$$

$$x^{(k+1)} = y^{(k)} / \|y^{(k)}\|$$

$$\lambda = \frac{\|y^{(k)}\|}{\|x^{(k)}\|}$$

a kiindulási  $x$  vektor ne legyen a nullvektor, és nem lehet merőleges a legnagyobb abszolútértékű sajátértékhez tartozó sajátvektorra.

A mátrixot szorozzuk jobbról az  $x$  vektorunkkal, majd a kapott  $y$  vektort **normalizáljuk**  $\Rightarrow$  egységnyi hosszúra változtatjuk, azaz leosztjuk a hosszával.

**(tehát a kapott értékeket az  $y^{(k)} = Ax^{(k)}$  ből  $\|y\| = \sqrt{y_0^2 + \dots + y_n^2}$ )**

A  $k$  betűk a kitevőben a  $k$ . iterációt jelentik, nem  $k$ . hatványt.

## Inverz hatványmódszer

$$Ay = x^k$$

$$x^{(k+1)} = y / \|y\|$$

Az inverz hatványmódszer azon a felismerésen alapul, hogy ha az  $A$  mátrix sajátértéke  $\lambda$ , és a hozzá tartozó sajátvektor  $x$ , akkor  $A^{-1}$  egy sajátértéke  $\lambda^{-1}$ , és a hozzá tartozó sajátvektor  $x$ .

# 8. Érintő, szelő, és húr módszer, a konjugált gradiens eljárás. Lagrange interpoláció. Numerikus integrálás

---

# Érintő, szelő, húrmódszer, konjugált gradiens eljárás

---

Mindegyik egyváltozós függvény zérushelyét keresi, iterációs módszerrel.

## Érintőmódszer

Más néven Newton-módszer

$f(x) = 0$  egyenlet zérushelyét keressük, ez legyen  $x^*$

Ennek egy környezetében, ha  $f(x)$  differenciálható, válasszunk ebből a környezetből egy kezdőértéket

Az iteráció, amit használunk:

$$x_{k+1} = x_k - \frac{f(x_k)}{f'(x_k)}$$

Magyarul, a következő megoldást úgy kapjuk, hogy **az előző megoldásból kivonjuk a függvény  $x_k$  helyen felvett értékének és a függvény deriváltjának az  $x_k$  pontban felvett értékének a hányadosát**. → Ezzel képezzük az adott ponthoz húzott érintőt.

Ha az  $f(x)$  függvény kétszer folytonosan differenciálható az  $x^*$  egy környezetében, akkor van olyan pont, ahonnan indulva a Newton-módszer **kvadratikusan konvergens** sorozatot ad meg, **aka gyorsan konvergál a megoldáshoz**.

$$|x^* - x_{k+1}| \leq C |x^* - x_k|^2$$

## Szelőmódszer

A Newton módszer hátránya, hogy szükség van a **deriváltak** kiszámítására → költséges.

Legyen megint  $x^*$  az  $f(x) = 0$  egyenlet egyszeres gyöke, és megint ezt keressük numerikus iterációval.

A függvény deriváltját nem mindig tudjuk, de a függvényt ki tudjuk értékelni minden helyen. Ekkor  $f'(x_k)$  helyett használhatjuk az numerikus deriváltat.

$$f'(x_k) = \frac{f(x_k) - f(x_{k-1})}{x_k - x_{k-1}}$$

Ekkor  $f'$  helyére a felső képletet behelyettesítve megkapjuk a szelőmódszer iterációs képletét:

$$x_{k+1} = x_k - \frac{f(x_k) * (x_k - x_{k-1})}{f(x_k) - f(x_{k-1})}$$

Azért szelőmódszer a neve, mert  $x_{k+1}$  az az  $(x_k, f(x_k))$  és  $(x_{k-1}, f(x_{k-1}))$  **pontokon átmenő egyenes és az x tengely metszéspontjának koordinátája.**

Olyan  $x_0, x_1$  **kezdőértékekkel szokás indítani, amelyek közrefogják a gyököt**, amit keresünk.

## Húrmódszer

A szelőmódszer egy változata.

Feltesszük, hogy a kezdeti  $x_0, x_1$  pontokban az  $f(x)$  függvény ellentétes előjelű, és  $f(x_{k+1})$  függvényében a megelőző két pontból azt választjuk, amivel ez a tulajdonság fennmarad.

## Példák a fenti módszerekre

PL:  $f(x) = x^3 - x + 1$  függvény.

**Newton:**

ennek a deriváltja:  $f'(x) = 3x^2 - 1$

Keressünk egy olyan környezetet, ahol lehet egy zérushely.

Ha megnézzük a függvényt pár helyen, akkor azt kapjuk, hogy  $f(-2) = -5$  és  $f(-1) = 1$ , szóval mivel ezek ellentétes előjelűek, válasszuk  $x_0 = -1$  értéket.

Felírjuk a képletet:

$$x_{k+1} = x_k - \frac{x_k^3 - x_k + 1}{3x_k^2 - 1}, \text{ behelyettesítve megkapjuk az } x_1\text{-et. ami itt } -1.5$$

**Szelő módszer:**

Két alappontot kell venni, a fenti függvény esetén tudjuk hogy a  $-2$  és  $-1$  közrefogja a zérushelyet, szóval legyen ez a két alappont.

Felírjuk a képletet:

$$x_{k+1} = x_k - \frac{(x_k^3 - x_k + 1)(x_k - x_{k-1})}{(x_k^3 - x_k + 1) - (x_{k-1}^3 - x_{k-1} + 1)}$$

### Húr módszer esetén:

Tudjuk, hogy a  $-2$  és  $-1$  pontoknál különböző előjelű a függvényérték, szóval ezeket használhatjuk.

Beírjuk a fenti képletbe a dolgokat. eredményként az jön ki, hogy  $f(-\frac{7}{6})$  ami közel  $0.5787$ , tehát ezt a pontot megtartjuk amelyik negatív + ezt az új pontot.

### Konvergenciájuk

A **Newton módszer gyorsabban konvergál mint a húrmódszer, a húrmódszer pedig gyorsabban mint a szelő.** Viszont nem kell nekik derivált.

### Konjugált gradiens eljárás

Az eljárás olyan lineáris ( $Ax = b$  alakú) egyenletrendszerek megoldására alkalmaz, ahol az  $A$  **együtthatómátrix szimmetrikus** ( $A = A^T$ ), **pozitív definit** ( $\forall x \neq 0 \ x^T Ax > 0$ ) és **valós** ( $A \in \mathbb{R}^{n \times n}$ ).

Pontos számolásokkal véges sok lépésben megtalálná a megoldást, de a **kerekítési hibák miatt iterációs eljárásnak veszik.**

**Gradiens:** Változók parciális deriváltjai vektorba rendezve. Van iránya és nagysága.

Ismert, hogy a többváltozós függvények gradiensvektorával ellentétes irányban csökken a leggyorsabban.

$q(x) = \frac{1}{2}x^T Ax - x^T b$  kvadratikus függvény minimumpontját keressük, mert ez ugyanaz, mint az eredeti egyenletrendszerünk megoldása, ha létezik.

Úgy keressük a következő közelítő megoldást, hogy van egy **keresési irányunk** ( $s_k$ ), és egy **lépésközünk** ( $\alpha$ ), és az aktuális pontból lépünk ebbe az irányba ekkora lépésközzel egyet.

A **negatív gradiensvektort** nevezzük **reziduális vektornak** (erre csökken a függvényünk).

**Ez lesz**  $r = b - Ax$ .

A keresési irányban ott lesz a **célfüggvény minimális ahol az új reziduális vektor merőleges az előző keresési irányra**, szóval tudjuk pontosan, hogy hova kell lépnünk az adott irányban.

**Tehát a konjugált gradiens módszer:**

- meghatározzuk a lépéshosszt
- meghatározzuk az **új közelítő megoldást** (lépünk egyet az előző megoldásból az adott irányba az új lépéshosszal ( $\alpha$ ))
- ebből kiszámoljuk az új reziduális vektort
- Kiszámolunk egy segédváltozót
- és az új keresési irányt a segédváltozóval
- és kezdjük előlről

A megállási feltételünk lehet az, hogy az utolsó néhány iterált közelítés eltérése és a reziduális vektorok eltérése bizonyos kicsi határ alatt maradtak.

## Lagrange interpoláció

**Függvényközelítési módszer.** Van pár alappontunk, és ezekre szeretnénk egy polinomot illeszteni. Ezek az **alappontok legyenek páronként különbözőek.**

Minden pontra felírunk egy egyenletet. **Ahány alappontunk van, max annyiad fokú lesz a kapott polinomunk.** Az egyenlet úgy fog kinézni, hogy ismerjük az  $x_i$  értéket, és mindenhova behelyettesítjük őket, és ezeknek az  $x_i^1, x_i^2$ , stb változóknak keressük az együtthatóját. Az egyenlet jobb oldalán pedig az  $f(x_i)$  értékek vannak.

Ebből kapunk egy lineáris egyenletrendszert, ahol az együtthatókat keressük. Ennek az egyenletrendszernek a mátrixa egy Vandermonde-mátrix lesz. Ebből következik, hogy pontosan egy polinom létezik, ami az adott pontokon áthalad.

A Lagrange-interpoláció az interpoláló polinomot a  $\sum_{i=1}^n f(x_i)L_i(x)$  alakban adja meg.

$L_i(x)$ -et úgy kapjuk, hogy egy nagy törtet veszünk - a **számlálóban összeszorozzuk** az összes  $x - x_j$ -t, ahol  $j$  nem egyenlő  $i$ -vel, tehát  $x - x_i$  szorzó kimarad belőle

A **nevezőben pedig  $x_i - x_j$ -ket szorzunk össze**, mindenhol, ahol  $j$  nem egyenlő  $i$ -vel szintén (különben nullával osztanánk).

PL:

Legyen adott 4 alappont:  $(-1, 2)$ ,  $(0, 0)$ ,  $(1, 4)$ ,  $(4, 0)$ , és ennek keressük a harmadfokú interpolációs polinomját:

<b>x</b>	<b>-1</b>	<b>0</b>	<b>1</b>	<b>4</b>
$p_3(x)$	2	0	4	0

Határozzuk meg az  $L_i(x)$  polinomokat:

$$L_1(x) = \frac{(x-0)(x-1)(x-4))}{(-1-0)(-1-1)(-1-4)}$$

$$L_2(x) = \frac{(x+1)(x-1)(x-4))}{(0+1)(0-1)(0-4)}$$

$$L_3(x) = \frac{(x+1)(x-0)(x-4))}{(1+1)(1-0)(1-4)}$$

$$L_4(x) = \frac{(x+1)(x-0)(x-1))}{(4+1)(4-0)(4-1)}$$

$p_3(x) = 2 * L_1(x) + 0 * L_2(x) + 4 * L_3(x) + 0 * L_4(x) \dots$  behelyettesítés és kiszámolás.

## Numerikus integrálás

---

**Határozatlan integrál:**

$$\int f(x) = F(x)dx,$$

ahol  $F'(x) = f(x)$  (deriválás megfordítása).  $F(x)$ -et **primitív függvénynek nevezzük**.

**Határozott integrál:** Célja, hogy egy adott  $f(x)$  függvénynek adott  $[a, b]$  intervallumon szeretnénk a **görbe alatti (előjeles) területét** kiszámítani.

$$\int_a^b f(x)dx = F(b) - F(a). \text{ (Newton-Leibniz formula)}$$

A fenti formula közelítése a cél, tehát **adott egy  $f(x)$  függvény határozott integrálját szeretnénk megközelíteni az  $[a, b]$  intervallumon**

**Kvadratúra formulák:**

$Q_n(f)$ -fel jelöljük,  $Q_n(f) = \sum_{i=1}^n w_i f(x_i)$  azaz, **az alappontokon felvett függvényérték  $w_i$  szerinti súlyozott összege**.

- Veszünk  $x_1, \dots, x_n$  alappontokat, általában feltesszük, hogy az összes  $x_i$  az  $[a, b]$  intervallumban van, ugye ebben az intervallumban keressük a határozott integrálját  $f$ -nek.
- A  $w_i$  számokat pedig súlyoknak hívjuk, amiket minden  $x_i$  alapponthoz hozzárendelünk.



### Téglalap szabály:

Amennyibe **csak egy alappontot** veszünk, az  $x_1 = \frac{a+b}{2}$  felezőpontot és a hozzárendelt  $w_i$  súly az intervallum mérete, azaz  $b - a$  lesz

**Tétel:** A  $Q_n$   $n$  alappontos kvadratúra-formula rendje legfeljebb  $2n-1$  lehet

### Interpolációs kvadratúra-formulák:

**A téglalap szabálynál veszünk egy  $x_1$  alappontot és erre illesztünk egy polinomot, és ennek a polinomnak a határozott integrálját vesszük.**

amennyiben, ha egy kvadratúra formula megkapható a következő alakban:

- Meghatározzuk a módszertől függően az  $x_1, \dots, x_n$  alappontokat,
- A kvadratúra-formula értéke az  $(x_i, f(x_i))$  pontokra illesztett Lagrange-interpolációs polinom  $[a, b]$ -n vett integrálja.

**Lagrange-interpolációs polinom:** Az  $(x_i, f(x_i))$  pontokra illesztett polinomok előállnak a következő alakban:  $\sum_{i=1}^n f(x_i) L_i(x)$ , ahol  $L_i(x)$  az  $i$ -edik Lagrange-alappolinom.

A súlyok:  $w_i = \int_a^b L_i(x) dx$

### Newton-Cotes formulák

Ha az  $[a, b]$  intervallumot elosztjuk **ekvidisztánsan** (egyforma méretű intervallumokra), és ezek végpontjait választjuk alappontoknak.

Ezek a Newton-Cotes formulák. Lehet **nyitott** és **zárt** attól függően, hogy  $a$  és  $b$  alappontok lehetnek-e.

**Nyitott esetén:**  $n+1$  egyenlő részre kell osztani az intervallumot

**Zárt esetén:**  $n-1$  egyenlő részre kell osztani az intervallumot

### Trapéz szabály:

pl: A legegyszerűbb esetben két alappontunk van és erre a két alappontra egy elsőfokú polinomot tudunk majd illeszteni.

Felvesszük a pontokat (pl:  $x_1 = a$   $x_2 = b$ ), meg a súlyt ami  $w_1 = w_2 = \frac{b-a}{2}$ , azaz

$$(f(a) + f(b)) * \frac{b-a}{2})$$

(A  $\frac{b-a}{2}$  az  $x_1, x_2$ -re illesztett Lagrange alappolinommal fog kijönni.)

## Összetett kvadratúra-szabályok

Az  $[a, b]$  intervallumokat felbontják  $n$  egyforma részre, és ezekre külön-külön csinálnak egy kvadratúra formulát.

# 10. Normálformák az elsőrendű logikában. Egyesítési algoritmus. Következtető módszerek: Alap rezolúció és elsőrendű rezolúció, ezek helyessége és teljessége

---

### Elsőrendű logika szintaxis:

Elsőrendű változók:  $x, y, z, \dots, x_1, y_5 \dots$

Függvényjelek:  $f, g, \dots, f_1, g_5 \dots$

Predikátumjelek:  $p, q, r, \dots, p_1 \dots$

Konstansok:  $a, b, c, \dots$

Konnektívák:  $\vee, \wedge, \neg, \leftrightarrow, \rightarrow$

Kvantorok:  $\forall, \exists$

Logikai konstansjelek:  $\downarrow, \uparrow$

### Egyéb fontos információk

- A változók **objektumok** egy halmazából kapnak értéket. (Pl természetes számok, stringek)
- Az objektumokat **függvények** fogják újabb objektumokba transzformálni. (összeadás, concat stb.)
- **prédikátumok** fogják igazságértékké transzformálni. (páros-e, stb.)
- Ha  $a$  formulában minden változó előfordulás kötött, akkor **mondatnak** nevezzük.

### Szintaxis:

- **Termek (Objektumértékek):**
  - Minden változó term
  - Ha  $f/n$  függvényjel,  $t_1, \dots, t_n$  pedig termék, akkor  $f(t_1, \dots, t_n)$  is term.
- **Formulák (Igazságértékek):**

- Ha  $p/n$  **predikátumjel**,  $t_1, \dots, t_n$  pedig termek, akkor  $p(t_1, \dots, t_n)$  egy atomi formula
- Ha  $F$  formula, akkor  $\neg F$  is az.
- $\uparrow, \downarrow$  is formulák
- Ha  $F$  formula és  $x$  változó, akkor  $\exists x F$  és  $\forall x F$  is formulák.

### Szemantika:

- Ahhoz, hogy **kitudjunk értékelni egy elsőrendű logikai formulát**, szükség van **strukturára**:  $A = (A, I, \phi)$ 
  - $A$  egy nemüres halmaz, az **alaphalmaz**
  - $\phi$  a változóknak egy *default* értékadása, minden  $x$  változóhoz egy  $\phi(x) \in A$  objektumot rendel.
  - $I$ , egy **interpretációs függvény**, ez rendel függvény- és predikátumjelhez szemantikát az adott struktúrában.
- A  $t$  term értéke az  $A$  struktúrában,  $A(t)$ 
  - Ha  $t = x$ , változó, akkor  $A(t) = \phi(x)$  (tehát a változók értékét a  $\phi$  szabja meg).
  - **Alapterm**: azok a termek amelyek nem tartalmaznak változókat.

**Mondat:** Ha nem szerepel benne változó szabadon.

## Normálformák predikátumkalkulusban

---

Formulákkal dolgozni tudjunk, úgy nevezett **zárt Skolem** alakra kell hozni.

Egy  $F$  formula **Skolem alakú**, ha

$$F = \forall x_1 \dots \forall x_n F^*$$

Ahol  $F^*$  formula magjában már nincs kvantor.

Ezekkel azért jó dolgozni mert,

$$F = \forall x_1 \dots \forall x_n F^* \models F^*[x_1/t_1, \dots, x_n/t_n]$$

,

tetszőlege  $t_i$  termekre. Azaz az összes magban lévő változót valami termel helyettesítem. És ez azért jó, mert az algoritmusok egyre bonyolultabb helyettesítéseket csinálnak, bízva, hogy kiütik egymást.

Nem adható meg minden formulához egy vele ekvivalens Skolem alakú formula

VISZONT:

Minden F formulához megadható egy olyan F' Skolem alakú formula, ami PONTOSAN akkor kielégíthető, ha F is kielégíthető.

(Vagyis ha F kielégíthető akkor F' is az, és ha F kielégíthetetlen akkor F' is az:

Ilyenkor F és F' "s-ekvivalensek")

Példán keresztül:

Feladat:

$$(\neg(\neg\exists yq(g(x,x),y) \vee \neg\forall zp(f(z))) \wedge (\forall z\exists y(q(c,g(z,c)) \rightarrow p(c)) \wedge \neg\forall y(q(f(y),c) \wedge q(c,z))))$$

### 1. Nyílak eliminálása

$$(\neg(\neg\exists yq(g(x,x),y) \vee \neg\forall zp(f(z))) \wedge (\forall z\exists y(\neg q(c,g(z,c)) \vee p(c)) \wedge \neg\forall y(q(f(y),c) \wedge q(c,z))))$$

### 2. Kiigazítás (Változó név ütközés elkerülés)

- Különböző kvantorok különböző változókat kötnek
- Nincs olyan változó, amely szabadon ( $\exists$ ) és kötötten ( $\forall$ ) is előfordul
- Indexelés

$$(\neg(\neg\exists y_1q(g(x,x),y_1) \vee \neg\forall z_2p(f(z_2))) \wedge (\forall z_3\exists y_4(\neg q(c,g(z_3,c)) \vee p(c)) \wedge \neg\forall y_5(q(f(y_5),c) \wedge q(c,z))))$$

### 3. Prenex alakra hozás

- Kvantorokat az elejére szervezzük. Ha volt előtte negálás alkalmazzuk rajta
- $$\exists y_1\forall z_2\forall z_3\exists y_4\exists y_5(\neg(\neg q(g(x,x),y_1) \vee \neg p(f(z_2))) \wedge ((\neg q(c,g(z_3,c)) \vee p(c)) \wedge \neg(q(f(y_5),c) \wedge q(c,z))))$$

### 4. Skolem alakra hozás

- Összes kvantor elől és mindegyik  $\forall$
- Töröljük  $\exists$  változókat (pl  $\exists x$ )
- A magbeli törölt változók helyére mindenhova  $f(x_1, \dots, x_n)$  kerül, ahol  $f$  egy új függvényjel és az előtte lévő  $\forall$  változói szerepelnek benne.

$$\forall z_2\forall z_3(\neg(\neg q(g(x,x),h_1) \vee \neg p(f(z_2))) \wedge ((\neg q(c,g(z_3,c)) \vee p(c)) \wedge \neg(q(f(h_3(z_2,z_3)),c) \wedge q(c,z))))$$

### 5. Lezárás

- Ne maradjon szabad változó-előfordulás
  - A szabad változó helyére, berakunk egy új konstans szimbólumot.
- $$\forall z_2 \forall z_3 (\neg(\neg q(g(c_3, c_3), h_1) \vee \neg p(f(z_2))) \wedge ((\neg q(c, g(z_3, c)) \vee p(c)) \wedge \neg(q(f(h_3(z_2, z_3)), c) \wedge q(c, c_5))))$$

## Egyesítési algoritmus

---

Ha  $F$  egy formula, akkor  $F[x/t]$  azt jelenti, hogy  $F$ -ben  $x$  összes előfordulását helyettesítjük  $t$ -vel.

Ha  $x_1, x_2, \dots, x_n$  **változók**, és  $t_1, \dots, t_n$  **termek**, akkor az  $[x_1/t_1], \dots, [x_n/t_n]$  helyettesítés azt jelenti, hogy először  $x_1$  helyére írunk  $t_1$ -et, aztán az eredményben  $x_2$  helyére  $t_2$ -t, stb.

Formulák halmazaira, pl klózokra is értelmezhetjük ezt.

Klóz végzett helyettesítésnél  $[x/t]$  azt jelenti, hogy minden klózra elvégezzük az  $x$  helyére  $t$  helyettesítést, és az eredményeket visszapakoljuk egy halmazba.

Ha  $C = l_1, l_2, \dots, l_n$  **literálok halmaza**, akkor  $s$  a  $c$  egyesítője, ha  $l_1 * s = \dots = l_n * s$ .  
 $C$ -re akkor mondjuk, hogy egyesíthető, ha van egyesítője.

Az  $s$  helyettesítés általánosabb az  $s'$  helyettesítésnél, ha van olyan  $s''$  helyettesítés, hogy  $s * s'' = s'$ .

### Egyesítési algoritmus:

- **input:**  $C$  klóz
- **output:**  $C$  legáltalánosabb helyettesítője, ha egyesíthető, különben azzal tér vissza, hogy nem egyesíthető
- veszünk két literált, és keressük az első eltérést
- ha az egyik helyen egy  $x$  változó áll, a másikon egy  $t$  term, amiben nincs  $x$ , akkor  $x/t$  és vissza az előző pontra
- különben return nem egyesíthető

Nem egyesíthető pl

- ha  $f(x)$  és  $c$  a különbség a két literál azonos pontján
- ha  $x$  és  $f(x)$  a különbség
- ha  $g(x)$  és  $f(x)$  a különbség

# Alap rezolúció

---

Azért ALAP mert **alap termék** (Azok a termék, amelyek nem tartalmaznak változót) vannak benne.

( $E(\Sigma)$ ): Klózok herbrand kiterjesztése)

- **input:** elsőrendű formulák egy  $\Sigma$  halmaza
- **output:** kielégíthetetlen véges sok lépésben, vagy kielégíthető véges sokban vagy végtelen ciklus
- Módszer:
  - $\Sigma$  elemeit **zárt skolem alakra** hozzuk, a formula **belsejét pedig CNF-re**, ez legyen  $\Sigma'$
  - ekkor  $E(\Sigma')$  a klózok **alap példányainak** a halmaza
  - $E(\Sigma')$ -n futtatjuk az ítéletkalkulusbeli rezolúciós algoritmust
  - általában végtelen sok alapterm van
- vegyük fel  $E(\Sigma')$  egy elemét, és rezolváljunk vele, amíg lehet
- ha kijön az üres klóz, akkor jók vagyunk, ha nem, generálunk tovább

**Helyesség és teljesség:**

$\text{üresklóz} \in \text{Res}^*(E(\Sigma'))$ , ha  $\Sigma \models \perp$ , **AZAZ, HA** letudjuk vezetni az üresklózt akkor kielégíthetetlen, és fordítva

**Bizonyításra pár bulletpoint:**

1. Zárt Skolem alakra hozás az s-ekvivalens átalakítás, azaz ha  $\Sigma$  pontosan akkor kielégíthetetlen, ha  $\Sigma'$  is
2. Herbrand-tétel következménye, hogy  $\Sigma'$  pontosan akkor kielégíthetetlen ha  $E(\Sigma')$  az

## Elsőrendű rezolúció

---

- **input:** elsőrendű formulák egy szigma halmaza
- **output:** kielégíthetetlen-e?
- $\Sigma$  zárt skolemre, mag cnfre,  $\Sigma'$
- $\Sigma'$  elemeit közvetlenül felvehetjük a listára
- ha kijön az üres klóz, kielégíthetetlen
- ha nem tudunk több klózt levezetni, kielégíthető

Rezolvensképzés:

- C1 és C2 klózat akarjuk rezolválni
- átnevezzük a változókat úgy, hogy ne legyen közös változó C1-ben és C2-ben
- kiválasztunk C1-ből és C2-ből is literálokat, az egyikből pozitívokat, a másiktól negatívokat
- ezeket pozitívan belepakoljuk egy C halmazba
- ha C egyesíthető egy s helyettesítéssel, akkor vehetjük a rezolvensét C1-nek és C2-nek
- elmentjük s-t
- vesszük C1-ből és C2-ből a maradék literálokat, és berakjuk egy halmazba
- ezen a halmazon elvégezzük az s helyettesítést, ez lesz a rezolvens

### Helyesség és teljesség:

Az elsőrendű klózatok  $\Sigma$  halmaza pontosan akkor **kielégíthetetlen**, ha  $\text{üresklóz} \in \text{Res}^*(\Sigma)$  (levezethető az üresklóz  $\Sigma$  az elsőrendű rezolúciós algoritmussal)

### Bizonyításra pár bulletpoint:

#### 1. Helyesség:

- Kijöhet az üres klóz, akkor  $\Sigma$  kielégíthetetlen, rezolvensképzés helyességéből következik.

#### 2. Teljesség:

- Ha  $\Sigma$  kielégíthetetlen, akkor az üres klóznak van egy  $C'_1, \dots, C'_n = \text{üresklóz}$  alaprezolúciós levezetése.

## 9. Normálformák az ítéletkalkulusban, Boole-függvények teljes rendszerei. Következtető módszerek: Hilbert-kalkulus és rezolúció, ezek helyessége és teljessége

---

### Alapvető információk

---

- $\text{Mod}(F)$ : Az  $F$  formula **modelljei** (olyan értékadások amelyek mellett az  $F$  igaz)

- $A \in Mod(F)$ : az A **értékkadás** F egy **modelje**
- $\models F$ : Az F formula **tautológia** (azaz minden értékkadás mellett igaz)
- $F \models G$ : Az F formulának **logikai következménye** a G formula.
  - $Mod(F) \subseteq Mod(G)$
  - Ha  $A(F) = 1$ , akkor  $A(G) = 1$  is.
- $F \equiv G$ 
  - $Mod(F) = Mod(G)$

## Ítéletkalkulus

---

- Vannak **változók** ezeket  $(p, q, r)$  szoktuk jelölni, és a 0, 1 halmazból kapnak igazságértéket.
- A formulák változókból épülnek fel ítéletlogikai összekötő jelekkel (**konnektíva**) pl  $\neg, \wedge, \vee$  stb.

## Normálformák az ítéletkalkulusban

---

Klózok éselése **Konjunkció**

Literálok vagyolása **Diszjunkció**

### DNF (Diszjunktív normálforma)

A formula olyan alakja:

- a változók pozitívan vagy negatívan szerepelhetnek benne
- a zárójelekben lévő pozitív vagy negatív változók között éselés van
- a zárójelek között vagyolás van

### Nyílmentes formula

A nyíllakat elimináljuk a formulából a következő szabályok alkalmazásával:

- $F \rightarrow G \equiv \neg F \vee G$
- $F \leftrightarrow G \equiv (F \rightarrow G) \wedge (G \rightarrow F) \equiv (\neg F \vee G) \wedge (\neg G \vee F)$



## CNF (Konjunktív normálforma)

diszjunkciók konjunkciója

A CNF alakban klózok vannak, és a klózok vannak összeéselve egymással.

Egy klózban változók vannak, negatívan vagy pozitívan, és ezek között vagyolás van. Úgy kapjuk, hogy egy már NNF-ben lévő formulában alkalmazzuk a **disztribúciós szabályt**:

- $(F \wedge G) \vee H \equiv (F \vee H) \wedge (G \vee H)$
- $(F \wedge G) \vee (H \wedge I) \equiv (F \vee H) \wedge (F \vee I) \wedge (G \vee H) \wedge (G \vee I)$

### CNF-re hozás:

1. A konnektívák eliminálása. (Fent nyilmentes formula)
2. Bevisszük a  $\neg$  jeleket a *deMorgan* azonosságokkal.
  - $\neg(F \vee G) \equiv \neg F \wedge \neg G$
  - $\neg(F \wedge G) \equiv \neg F \vee \neg G$
3. Végül a  $\vee$  jeleket visszük be disztributivitással (Fent CNF-re hozás)

## Teljes rendszerek

---

### Boole függvények

**(n-változós) Boole-függvény:** Bitvektort egy bitbe képző függvény  $f : \{0, 1\}^n \rightarrow \{0, 1\}$ .

Az  $f/n$  jelzi, hogy az  $f$  egy **n-változós** függvény.

Ezek igazából a konnektívákhoz használatos igazságtábla.

### Teljes rendszer

**Boole függvények** egy  $H$  rendszere **teljes**, ha minden  $n \geq 1$ -változós Boole-függvény előáll:

- Projekciókból
- és  $H$  elemeiből
- alkalmas **kompocízióval**

Olyan Boole függvények, amelyekkel kifejezhető az összes többi is.

Logikai műveletek (Boole függvények) egy rendszerét akkor nevezzük teljesnek, ha egy, már korábban teljesnek

ítélt rendszer minden műveletét ki tudjuk fejezni ezen műveletekkel.

$\neg, \wedge, \vee$  stb.

**A  $\{\neg, \wedge, \vee\}$  rendszer teljes**, mert minden formulát CNF alakra tudunk hozni. Ezek alapján teljes még:

- $\{\neg, \vee\}$ :
  - A negáció okés, az éselés okés, a vagyolást ki tudjuk fejezni:
  - $p \wedge q \equiv \neg(\neg p \vee \neg q)$
- $\{\neg, \wedge\}$ 
  - A negáció okés, a vagyolás okés, az éselést ki tudjuk fejezni:
  - $p \vee q \equiv \neg(\neg p \wedge \neg q)$

**A  $\{\neg, \rightarrow\}$  rendszer is teljes, mert** tudjuk, hogy **a  $\{\neg, \vee\}$  rendszer teljes**, és ki tudjuk fejezni **a műveleteit**:

- $\neg$  okés, vagyolás:
- $p \vee q \equiv (\neg p) \rightarrow q$

**A  $\{\rightarrow, \downarrow\}$  rendszer is teljes, mert** tudjuk, hogy **a  $\{\neg, \rightarrow\}$  rendszer teljes**, és ki tudjuk fejezni a műveleteit:

- $\rightarrow$  okés
- $\neg p \equiv p \rightarrow \downarrow$

## Következtető rendszerek

---

Ha  $\Sigma \models F$  pontosan akkor, ha  $\Sigma \cup \{\neg F\} \models \downarrow$ , a rezolúciós algoritmus következtetések igazolására is használhatóak a következő módon.

Helyesség és teljesség általában:

*Helyesség:* Ha azt mondom, hogy igen, akkor az tényleg legyen a válasz igen.

*Teljes:* Ha a válasz tényleg igen kellene legyen, akkor arra egyszer ráfogok jönni.

## Rezolúció

A rezolúciónál a **formuláink CNF alakban** vannak. A rezolúcióval logikai következményeket tudunk bebizonyítani, pl. hogy egy formulahalmaznak logikai következménye egy formula.

**Alapból a logikai következmény azt jelenti, hogy azoknak az értékadásoknak a halmaza, amelyek kielégítik a jobboldali formulá(ka)t, részhalmaza a jobboldali formulákat kielégítő értékadások**

halmazának. Ezzel az a baj, hogy az összes ilyen értékadást megtalálni nagyon hosszadalmas.

**Formailag:**

**input:** Klózik  $\Sigma$  halmaza

**output:** kielégíthetetlen-e  $\Sigma$ ?

**Algoritmus::**

Ezután listát vezetünk a klózikról. Egy klóz felkerülhet a listára, ha:

- eleme a  $\Sigma$ -nak
- két, korábban már a listán szereplő klóz rezolvense

**Következtető rendszer szerint formailag:**

- Input: Formulák egy  $\Sigma$  és egy  $F$  formula.
- Output: Igaz-e, hogy  $\Sigma \models F$
- Algoritmus:
  - CNF-re hozzuk a  $\Sigma$  összes elemét és a  $\neg F$  formulát is. A kapott klózik halmazát jelölje  $\Sigma'$
  - Hajtsunk végre  $\Sigma'$  rezolúciót. Ha üres halmaz eleme lesz, akkor  $\Sigma \models F$ , else nem.

Két klóznak akkor vehetjük a **rezolvenségét**, ha a **mindkettőben szerepel ugyanaz a változó**, de az **egyikben negatívan**, a **másikban pedig pozitívan**. Ekkor a **rezolvens egy olyan klóz lesz, ahol ez a változó már nem fog szerepelni, hanem csak a két klózban maradt összes többi változó**.

Ha a listára valamelyik lépésben rákerül az **üresklóz**, az azt jelenti, hogy  $\Sigma$  **kielégíthetetlen**, vagyis az eredeti logikai következmény fennáll.

**Ha sehogy sem tudjuk levezetni az üresklózt**, az azt jelenti, hogy a  $\Sigma$  **kielégíthető**, és az eredeti logikai következmény nem áll fenn.

**Helyesség:** Az algoritmus **kielégíthetetlen** válasszal áll meg, akkor az input  $\Sigma$  **valóban kielégíthetetlen**).

**Teljes:** Ha  $\Sigma$  **kielégíthetetlen**, akkor az algoritmus mindig a **kielégíthetetlen** válasszal áll meg.

Példa:

Igazoljuk rezolúcióval, hogy kielégíthetetlen:

$$(((p \rightarrow q) \wedge \neg q) \vee ((r \rightarrow \neg p) \wedge r)) \wedge s \wedge (s \rightarrow p)$$

1. CNF-re hozás

1. Nyilak eliminálása:

$$(((\neg p \vee q) \wedge \neg q) \vee ((\neg r \vee \neg p) \wedge r)) \wedge s \wedge (\neg s \vee p)$$

2. Negáció bevitele: Ez itt kész

3. Disztributivitás:

$$((\neg p \vee q \vee \neg r \vee \neg p) \wedge (\neg p \vee q \vee r) \wedge (\neg q \vee \neg r \vee \neg p) \wedge (\neg q \vee r)) \wedge s \wedge (\neg s \vee p)$$

2. Rezolúció:

$$\Sigma = \{\neg p, q, \neg r\}, \{\neg p, q, r\}, \{\neg p, \neg q, \neg r\}, \{\neg q, r\}, \{s\}, \{p, \neg s\}$$

MELLÉKLET KÉP (Rezolucio.JPG)

## Hilbert-kalkulus

**Hilbert rendszere (egy deduktív rendszer):**

- **Az input a  $\Sigma$  összes következményét lehet vele levezetni.**

Ebben a rendszerben **csak** a  $\rightarrow$  és a  $\downarrow$  logikai konstanst használhatjuk az **ítéleváltozókon kívül**

**Minden formulát ilyen alakra lehet hozni, mivel  $\{\rightarrow, \downarrow\}$  teljes rendszer.**

A Hilbert-kalkulusban Hilbert rendszerét használjuk. Az ilyen alakú formulákra is tudunk következtető rendszert építeni. A továbbiakban a formuláink mind Hilbert rendszeréből származnak.

**Hilbert rendszere: (JELE:  $\Sigma \vdash F$ )**

**Input:** Egy  $\Sigma$  formulahalmaz és egy  $F$  célformula

**Output:** Igaz-e, hogy  $\Sigma \vdash F$

**Lépések:** Listát vezetünk a formulákról. A listákra a következő elemek kerülhetnek fel:

- $\Sigma$  elemei
- Axiómapéldányok ízlés szerint.

- *Modus ponens*: ha  $F$  és  $F \rightarrow G$  is megvan a listán, akkor felvehetjük  $G$ .  
Gyakorlatilag levágjuk a nyíl nélküli.

### Háromféle axiómánk van:

- Ax1:  $(F \rightarrow (G \rightarrow H)) \rightarrow ((F \rightarrow G) \rightarrow (F \rightarrow H))$
- Ax2:  $F \rightarrow (G \rightarrow F)$
- Ax3:  $((F \rightarrow \perp) \rightarrow \perp) \rightarrow F$

Példa:

Mutassuk meg dedukcióval, hogy  $\vdash \perp \rightarrow p$

Ha alkalmazzuk a dedukciót akkor egyből az új feladat:  $\perp \vdash p$

KÉPET IDE

### Helyesség és teljesség:

- **Helyesség:**

Ha  $\Sigma \vdash F$ , akkor  $\Sigma \models F$ , AZAZ, ha valakit letudok vezetni az input  $\Sigma$ -ból akkor az következménye is a  $\Sigma$ -nak.

Tehát van valami  $F_1, \dots, F_n$  levezetés  $\Sigma$  felett, És akiket felvesszünk a listára az következménye lesz a  $\Sigma$ -nak.

- **Teljesség:**

- Azt állítjuk, hogy  $A \models F$  pontosan akkor igaz  $F$ -re, ha  $F \in \Sigma'$ , tehát az értékadás ( $A$ ), pontosan akkor fogja kielégíteni a formulát aki benne van a  $\Sigma'$ -ben

Elvileg kell a teljességhez

#### 1. dedukciós tétel:

$$\Sigma \vdash (F \rightarrow G) \Leftrightarrow \Sigma \cup \{F\} \vdash G,$$

Tehát a  $\Sigma$  formulahalmazból akkor tudunk **levezetni egy implikációt**  $(F \rightarrow G)$ , ha annak a **bal oldalát átrakjuk**  $\Sigma$ -ba  $(\Sigma \cup \{F\})$ , és ebből a **formulahalmazból le lehet vezetni a jobboldalt**  $(G)$ -t

#### 2. H-konzisztencia:

Egy  $\Sigma$  formulahalmazt H-konzisztensnek nevezünk, ha **nem** igaz, hogy  $\Sigma \vdash \perp$ .

Azaz **Hilbert rendszerben nem tudjuk bebizonyítani, hogy a formulahalmaz nem kielégíthető (van modelje).**

# 11. Keresési feladat: feladatrepresentáció, vak keresés, informált keresés, heurisztikák. Kétszemélyes zéró összegű játékok: minimax, alfa-béta eljárás. Korlátozás kielégítési feladat

---

Különbség a feladatrepresentáció és a játékok között, az **ágensek száma**.

## Keresési feladat: feladatrepresentáció, vak keresés, informált keresés, heurisztikák

---

### Feladatrepresentáció

Tekintsünk egy diszkrét, statikus, determinisztikus és teljesen megfigyelhető feladatkörnyezetet. Tegyük fel, hogy a világ tökéletesen modellezhető a következőkkel:

- **lehetséges állapotok halmaza**
- **egy kezdőállapot**
- **lehetséges cselekvések halmaza** (állapotátmenet függvény, minden állapothoz hozzárendelünk egy (cselekvés, állapot) rendezett párokból álló halmazt, tehát egy állapotban milyen cselekvések hatására milyen állapotba juthat az ágensünk)
- **állapotátmenet költségfüggvénye**, minden lehetséges állapot-cselekvés-állapot hármashoz hozzárendelünk egy költséget, azaz egy állapotból egy (másik) állapotba jutásnak mekkora a költsége
- **célállapotok halmaza**, tehát hova szeretnénk, hogy eljusson az ágensünk

Ez egy **súlyozott gráfot** definiál, ez a gráf az **állapottér**

Feltesszük továbbá, hogy az állapotok száma véges, vagy megszámlálható. Úton állapotok cselekvésekkel összekötött sorozatát értjük, ennek van egy összköltsége is.

pl: Utazástervezési feladat: útvonaltervezés,

állapotok = hely és időpont párok;

cselekvés = közlekedési eszközök aktuális állapotból való indulása

költség= idő és pénz fgv-e

### **Pl: 8-kirakó**

Kezdőállapot = maga a kezdőpálya

Állapotok = célállapotból csusztatásokkal elérhető konfigurációk

Cselekvés = üres hely mozgatása fel, le, jobbra és balra.

Költség = konstans minden cselekvésre

Célállapot = a célállapotot az ábra mutatja

KÉP HOZZÁ (KirakoMestint.JPG)

## **Vak (informálatlan) keresés**

### **Fakeresés**

Adott kezdőállapotból találjunk minimális költségű utat egy célállapotba. Az állapottér nem mindig adott explicit módon, és végtelen is lehet.

**Ötlet:** keresőfa építése, a kezdőállapotból nö vesszünk fát a szomszédos állapotok hozzávételével, amíg célállapotot nem találunk.

A keresőfa NEM azonos a feladat állapotterével, pl ha van két csúcs között oda-vissza él.

```
fakeresés():  
    perem = { újcsúcs(kezdőállapot) }  
    while perem.nemüres()  
        csúcs = perem.elsőkivesz()  
        if csúcs.célállapot() return csúcs  
        perem.beszúr(csúcs.kiterjeszt())  
    return failure
```

A csúcs.kiterjeszt() létrehozza a csúcsból elérhető összes állapotból a keresőfa csúcsot.

A perem egy prioritási sor, ettől függ a bejárési stratégia.

A hatékonyságot növelhetjük, ha úgy szűrünk be csúcsokat a perembe, hogy abban az esetben, ha a peremben található már ugyanazzal az állapottal egy másik csúcs, akkor ha az új csúcs költsége kisebb, lecseréljük a régi csúcsot az újra, különben nem tesszük bele az újat.

## **Algoritmusok vizsgálata**

Algoritmus **teljes** akkor és csak akkor, amikor létezik **véges számú állapot érintésével elérhető célállapot**, az algoritmus meg is talál egyet.

Egy algoritmus **optimális** akkor és csak akkor, ha **teljes** és minden megtalált célállapot optimális költségű.

Idő- és memóriaigény számolásához pár betű.

- **b**: szomszédok maximális száma
- **d**: a legkisebb mélységű célállapot mélysége
- **m**: A keresőfa maximális mélysége

Ahol m és d lehet megszámlálhatóan végtelen

### Szélességi keresés

Fakeresés, ahol a perem egy FIFO perem.

- **Teljes**, minden, véges számú állapot érintésével elérhető állapotot véges időben elér
- **Általában nem optimális**, de pl akkor igen, ha a költség a mélység nem csökkenő függvénye
- időigény = tárigény  $O(b^{d+1})$

### Mélységi keresés

Fakeresés, LIFO perem

- **Teljes**, ha a keresési fa véges mélységű
- **Nem optimális**
- Időigény: legrosszabb esetben  $O(b^m)$  (nagyon rossz, lehet végtelen), tárigény legrosszabb esetben  $O(bm)$  (ez egész biztató)

### Iteratívan mélyülő keresés

Mélységi keresések sorozata 1, 2, 3 stb korlátozva, amíg célállapotot nem találunk.

- Teljesség és optimalitás a szélességgel egyezik meg
- időigény  $= O(b^d)$  (akár jobb is lehet, mint a szélességi)
- tárigény  $= O(bd)$  (jobb, mint a mélységi)

**Ez a legjobb informálatlan kereső.**



## Egyenletes költségű keresés

A peremben a rendezés költség alapú, mindig először a legkisebb útköltségű csúcsot terjesztjük ki.

- **Teljes és optimális**, ha minden él költsége nagyobb  $\geq \epsilon > 0$
- (Idő és tárigény nagyban függ a **költségfüggvénytől**)

## Gráfkeresés

### Ha nem fa az állapottér!

Ha a **kezdőállapotból több út is vezet egy állapotba**, akkor a **fakeresés végtelen ciklusba eshet**

Fakeresés, de a perem mellett még tárolunk egy ún. **zárt halmazt** is. A zárt halmazba **azok a csúcsok kerülnek**, amiket **már kiterjesztettünk**. A perembe helyezés előtt minden csúcsra megnézzük, hogy már a zárt halmazban van-e. Ha igen, nem tesszük a perembe. Másrészt minden peremből kivett csúcsot a zárt halmazba teszünk. Így minden állapothoz a legelső megtalált út lesz tárolva.

## Informált keresés, heurisztikák

---

**Itt már tudjuk, hogy “hova megyünk”.**

**Heurisztika:** minden állapotból megbecsüli, hogy mekkora az optimális út költsége az adott állapotból egy célállapotba: **tehát értelmesebben tudunk következő szomszédot választani.**

Pl. légvonalbeli távolság a célig a térképen egy útvonal-tervezési problémához jó heurisztika.

$h(n)$ : optimális költség közelítése a legközelebbi célállapotba  $n$  állapotból

$g(n)$ : tényleges költség a kezdőállapotból  $n$ -be

## Mohó

Fakeresés, peremben a rendezést  $h()$  alapján csináljuk, mindig a legkisebb értékű csúcsot vesszük ki.

Ha csak annyit teszünk fel, hogy  $h(n) = 0$  ha  $n$  célállapot, akkor **fakeresés esetén:**

- Teljes, de csak ha a keresési fa véges mélységű
- Nem optimális
- időigény, tárigény  $O(b^m)$

**gráfkeresésnél** az optimalitás hiánya miatt az első megtalált út nem mindig a legjobb.

## A\*

A peremben a rendezést  $f() = h() + g()$  alapján végezzük, a legkisebb csúcsot vesszük ki.  $f()$  **a teljes út költségét becsüli a kezdőállapotból a végállapotba**. Ha  $h = 0$ , és gráfkeresést alkalmazunk, akkor a **Dijkstra-t** kapjuk.

- Egy  $h$  heurisztika **elfogadható**, ha nem ad nagyobb értéket, mint a tényleges optimális érték.  
Fakeresést feltételezve, ha  $h$  elfogadható és a keresési fa véges, akkor  $A^*$  optimális.
- Egy  $h$  heurisztika **konzisztens**, ha  $h(n) \leq$  mint a **valódi költség**  $n$  egyik bármely, plusz a szomszéd heurisztikája.  
Gráfkeresést feltételezve, ha  $h$  **konzisztens és az állapottér véges**, akkor  $A^*$  **optimális**.

Az  $A^*$  optimálisan hatékony, de a **tárigénye általában exponenciális**. és nagyon nagyban függ  $h$ -tól. Az **időigény** szintén nagyon **nagyban függ**  $h$ -tól.

PL rá: <http://www.inf.u-szeged.hu/~ihegedus/teach/a-star.pdf>

## Heurisztikák

A **jó** heurisztikus függvények előállítása fontos, lehetőleg elfogadható és konzisztens legyen.

**Relaxált probléma:** elhagyunk feltételeket az eredeti problémából.

Kombinálhatunk több heurisztikát is.

Készíthetünk mintaadatbázisokat, ahol részproblémák egzakt költségét tároljuk.

Belátható, hogy a relaxált probléma optimális költsége  $\leq$  az eredeti probléma optimális költségénél, mivel az eredeti probléma állapottere része a relaxálnak.  $\Rightarrow$  **elfogadható heurisztika**.

Sőt mivel a heurisztika a probléma egy relaxációjának tényleges költsége, ezért **konzisztens** is.

## Kétszemélyes zero összegű játékok: minimax, alfa-béta eljárás

---

### Kétszemélyes, lépésváltásos, determinisztikus, zero összegű játék

- lehetséges állapotok halmaza
- egy kezdőállapot
- lehetséges cselekvések halmaza, és egy állapotátmenet függvény
- célállapotok
- **hasznosságfüggvény**: Minden célállapothoz, hasznosságértéket rendel.

**Két ágens van**, felváltva lépnek. Az **egyik maximalizálni** akarja a hasznosságfüggvényt (MAX játékos), a **másik minimalizálni** (MIN játékos).

Konvenció szerint MAX kezd. Az első célállapot elérésekor a játéknak definíció szerint vége.

**Zero összegű játék: A MIN játékos minimalizálja a hasznosságot, ami ugyanaz, mint maximalizálni a negatív hasznosságot.** Ez a *negamax formalizmus*. Itt a két játékos nyereségének az összege a végállapotban mindig nulla, innen a zero összegű elnevezés.

(Játékelméletben: Az a játék, amelyben a játékosok csak egymás kárára növelhetik a nyereségüket.)

### Minimax algoritmus, alfa-béta vágás

Mindkét játékos ismeri a teljes játékgráfot, bármilyen komplex számítást képes elvégezni és nem hibázik (tökéletes racionalitás). A minimax algoritmus alapján lehet megvalósítani a legjobb stratégiát tökéletes racionalitás esetén.

Minimax:

```
maxÉrték(n)
1 if végállapot(n) return hasznosság(n)
2 max = -végtelen
3 for a in n szomszédai
```

```

4   max = max(max, minÉrték(a))
5   return max

minÉrték(n)
1   if végállapot(n) return hasznosság(n)
2   min = +végtelen
3   for a in n szomszédai
4       min = min(min, maxÉrték(a))
5   return min

```

Ha  $n$  végállapot, visszaadja a hasznosságát. Különben a max-nál  $n$  szomszédaira kiszámolja a maximális értéket, ami vagy az aktuális maximum, vagy nézi, hogy a másik játékos mit lépne.

Csak elméleti jelentőségű, a minimax algoritmus nem skálázódik. Az összes lehetséges állapot kiszámolása rettentő sok idő lenne pl sakknál.

### Alfa-béta vágás

Ha tudjuk, hogy pl MAX-nak már van egy olyan stratégiája, ahol biztosan egy 10 értékű hasznosságot el tud érni az adott csúcsban, akkor a csúcs további kiértékelésekor nem kell vizsgálni olyan csúcsokat, ahol MIN ki tud kényszeríteni  $\leq 10$  hasznosságot, mert ennél már MAX-nak van jobb stratégiája

minÉrték és maxÉrték hívásakor átadjuk az alfa és béta paramétereket is a függvénynek.

**Alfa jeletése:** MAXnak már felfedeztünk egy olyan stratégiát, amely alfa hasznosságot biztosít, ha ennél kisebbet találnánk, azt nem vizsgáljuk.

**Béta jelentése:** MINnek már felfedeztünk egy olyan stratégiát, amely béta hasznosságot biztosít, ha ennél nagyobbát találnánk, azt nem vizsgáljuk

A gyakorlatban a minimax és az alfa-béta vágásos algoritmusokat is csak meghatározott mélységig vizsgáljuk, illetve heurisztikákat is alkalmazhatunk. A csúcsok bejárési sorrendje is nagyon fontos, mert pl alfa béta vágásnál egy jó rendezés mellett nagyon sok csúcsot vághatunk le.

## Korlátozás kielégítési feladat

---

A feladat az állapottérrel adott keresési problémák és az optimalizálási problémák jellemzőit ötvözi. Az állapotok és célállapotok speciális alakúak.

**Lehetséges állapotok halmaza:** a feladat állapotai az  $n$  db változó lehetséges értékkombinációi.

$D = D_1 * \dots * D_n$  (\* itt most a descartes szorzat), ahol  $D_i$  ( $D$  domain értékkészletének  $i$ . értéke) az  $i$ . változó lehetséges **értékei**

**Célállapotok:** a megengedett állapotok, adottak különböző korlátozások, és azok az állapotok a célállapotok, amik minden korlátozást kielégítenek.

Az út a megoldásig lényegtelen, és gyakran célfüggvény is értelmezve van az állapotok felett, ilyenkor egy optimális célállapot megtalálása a cél.

PL: Gráfszínezési probléma.

Adott egy  $G(V, E)$  gráf, ahol  $n = |V|$ . A változók a gráf pontjai. Az  $i$  pont lehetséges színeinek halmaza a  $D_i$  és  $D_1 = \dots = D_n$ .

Minden  $e \in E$  élhez rendelünk egy  $C_e$  korlátozást, amely azokat a színezéseket engedi meg, ahol az  $e$  él két végpontja különböző színű.

## Inkrementális kereső algoritmusok

Optimalizálás helyett keresési feladatot definiálunk. **Nem az eredeti állapottér felett kell dolgozni**, hanem kikell terjeszteni ezt a teret úgy, hogy felvesszünk egy új "ismeretlen" értéket (**jele: ?**) és az összes Domainben lévő értékhez hozzáadjuk ezt a változót.

- $D_i = D_i \cup \{?\}$  vektorok lesznek az **új keresési tár állapotai**
- **Kezdeti állapot:** csupa kérdőjel ( $?, \dots, ?$ )
- **Állapotátmenet költsége** legyen konstans
- **Állapotátmenetek** valamely pontosa egy "?"-jelet lecserélnek egy adott változó másik értékére.  $\rightarrow$  sokkal kisebb fa méret.

Erre már lehet végrehajtani bármely korábban nézett informálatlan keresési algoritmus. A **mélységi keresés** elég jó, mivel kicsi a keresőfa mélysége és nem fogyaszt memóriát (backtrack)

Ennél jobb viszont az informális keresési algoritmusok bevetése:

- Válasszuk azt a változót, amihez a **legkisebb megengedett érték** maradt.
  - Ha nem egyértelmű akkor azt, amelyre a **legtöbb korlátozás** vonatkozik
- A választott változó megengedett értékeiből kezdjük azzal, amelyik a **legkevésbé korlátozza a következő lépések lehetséges számát**

# 12. Teljes együttes eloszlás tömör reprezentációja, Bayes hálók. Gépi tanulás: felügyelt tanulás problémája, döntési fák, naiv Bayes módszer, modellillesztés, mesterséges neuronhálók, k-legközelebbi szomszéd módszere

---

## Valószínűség

---

Problémák modellezésénél, megoldásánál szeretünk logikai változókat, és logikai következtetéseket használni. Ezzel problémák akadhatnak:

- Ha nem teljes a tudásunk
- Ha heurisztikai szabályokat vezetünk be, akkor a tapasztalat inkonzisztens lehet az elmélettel

Vagyis a hiányos, részleges tudás kezelésére a logika nem optimális. A **tudás tökéletlenségének** a kezelésére a valószínűséget használjuk. Ilyenkor az **ismeretlen tényeket és szabályokat véletlen hatásként kezeljük.**

Bayesi felfogásban a valószínűség a hit fokát, és nem az igazság fokát jelenti. (Szemben a fuzzy logikával, ahol pl: "ez a ház nagy" kijelentés folytonos igazságértéket vehet fel)

### Véletlen változók:

- Van neve, és értékkészlete (domainje): logikai, diszkrét, folytonos.
- **Elemi kijelentés:**  $A$  vél.változó  $D_A$  domainnel. Egy elemi kijelentés  $A$  értékének egy korlátozását fejezi ki (pl:  $A = d$ , ahol  $d \in D_A$ , amelyek a következő értékeket vehetik fel:
  - Logikai: Ekkor a Domain {Igaz, Hamis}
  - Diszkrét: Megszámlálható domain, pl {nap, eső, felhő, hó}
  - Folytonos:  $X$  véletlen változó,  $D \subseteq \mathbb{R}$
- **Elemi esemény:** Minden véletlen változóhoz értéket rendel. Ha az  $A_1 \dots A_n$  véletlen változókat definiáltuk a  $D_1 \dots D_n$  domainekkel, akkor az elemi események (lehetséges világok) halmaza a  $D_1 \times \dots \times D_n$  halmaz. Vagyis egy "lehetséges

világban” – **elemi eseményben** az  $A_i$  változó a hozzá tartozó  $D_i$  ből **pontosan egy értéket vesz fel.**

### Fenti definíciók pár következménye:

1. Minden lehetséges világot pontosan egy **elemi esemény** írja le.
2. Egy **elemi esemény** természetes módon minden lehetséges **elemi kijelentéshez igazságértéket rendel**
3. Minden **kijelentés** logikailag ekvivalens a neki nem ellentmondó elemi eseményeket leíró kijelentések halmazával.

### Valószínűség, kijelentések:

A valószínűség egy függvény, ami egy kijelentéshez egy valós számot rendel.  $P(a)$  az  $a$  kijelentés valószínűsége. Minden kijelentés elemi események egy halmazával ekvivalens.

**Egy kijelentés valószínűsége egyenlő a vele konzisztens elemi események**

**valószínűségének az összegével.** (Ehhez kell a teljes együttes eloszlás, ami megadja az összes elemi esemény valószínűségét)

### Feltételes valószínűség

$$P(A|B) = \frac{P(A \cap B)}{P(B)}$$

### Kazualitás:

Okszerűség, okozati kapcsolat

Pl: Fogfájás nem hat az időjárásra, tehát ott független, de az Időjárás hathat a Fogfájásra, ezért nem beszélhetünk mégsem függetlenségről  $\Rightarrow$  a Fogfájás ténye adhat infót az Időjárásról.

## Teljes együttes eloszlás tömör reprezentációja, Bayes hálók

---

### Teljes együttes eloszlás

Minden lehetséges eseményre tudjuk annak a valószínűségét. Pl van 3 logikai típusú véletlen változónk, akkor összesen  $2^3=8$ -féle eset lehet ezekre. A teljes együttes eloszlásnál mind a 8 esetnek tudjuk a valószínűségét.  $\rightarrow$  **az összes elemi esemény valószínűségét megadja.**

Viszont nyilván ez miatt nem skálázódik jól.

## Tömör reprezentáció

A **kijelentések függetlensége** a legfontosabb tulajdonság a teljes együttes eloszlás tömöríthetőségéhez. Van **függetlenség, és feltételes függetlenség**.

### Függetlenség

$a$  és  $b$  kijelentések **függetlenek**, ha  $P(a \wedge b) = P(a) * P(b)$

**A függetlenség struktúrát takar.** Ha pl  $n$  logikai változónk van, amik két független részhalmazra oszthatók  $m$  és  $k$  mérettel, akkor a  $2^n$  valószínűség tárolása helyett elég  $2^m + 2^k$  valószínűséget tárolni, ami sokkal kevesebb lehet.

Extrém esetben, ha pl. az  $A_1, \dots, A_n$  diszkrét változók kölcsönösen függetlenek (tetszőleges két részhalmaz független), akkor csak  $O(n)$  értéket kell tárolni, mivel ez esetben

$$P(A_1, \dots, A_n) = P(A_1) \dots P(A_n)$$

### Feltételes függetlenség

Az abszolút függetlenség ritka. Ezért használhatunk feltételesen függetlenséget.

$a$  és  $b$  kijelentések **feltételesen függetlenek**  $c$  feltevésével, akkor és csak akkor, ha  $P(a \wedge b | c) = P(a | c) * P(b | c)$ . Tipikus eset, ha  $a$  és  $b$  közös oka  $c$ .

Pl: fogfájás és a beakadás közös oka a luk.

### Naiv-Bayes modell alakja

Ha  $A$  feltevése mellett  $B_1, \dots, B_n$  kölcsönösen függetlenek, akkor

$P(B_1, \dots, B_n | A) = \prod_{i=1}^n P(B_i | A)$ . ha ez igaz, akkor ez a *Naiv-Bayes modell alakja*, ami a következőképp definiálható:

$$P(B_1, \dots, B_n, A) = P(A) \prod_{i=1}^n P(B_i | A),$$

**Ezzel  $O(n)$  tömörítés érhető el**

### Bayes szabály/tétel $a$ és $b$ kijelentésekre

$$P(a|b) = \frac{P(b|a)P(a)}{P(b)}$$

ez következik a feltételes valószínűség képletéből:

$$P(a|b) = \frac{P(a \cap b)}{P(b)}$$

alapján



$$P(a|b)P(b) = P(a \cap b) = P(b|a)P(a)$$

amiből a  $P(b)$ -vel leosztva adódik a tétel.

## Bayes hálók

A feltételes függetlenség hasznos, mert tömöríthetjük a teljes együttes eloszlást.

A teljes együttes eloszlás feltételes függetlenségeit ragadja meg és ezekből egy **speciális gráfot** definiál. Ez tömör és intuitív reprezentációt tesz lehetővé.

Bayes háló esetén alkalmazunk **láncszabályt**, ami azt jelenti, hogy a **teljes együttes eloszlást** (amit majd tömöríteni szeretnénk) **feltételes eloszlások szorzataként** jeleníti meg.

Formailag:

$$P(X_1, \dots, X_n) = \prod_{i=1}^n P(X_i | X_{i+1}, \dots, X_n)$$

Ezután az egyes feltételes valószínűségekből elhagyhatunk változókat, azaz minden  $P(X_i | X_{i+1}, \dots, X_n)$  tényezőre az  $\{X_{i+1}, \dots, X_n\}$  változókból vegyünk egy *Szülő*( $x_i$ ).

Ebből tudunk tömöríteni, mivel a Szülők halmaz minimális.

$$P(X_1, \dots, X_n) = \prod_{i=1}^n P(X_i | \text{Szülő}(X_i))$$

(ez a bayes háló)

**Ez a teljes eloszlás tömörített reprezentációja.**

Ezt lehet egy gráf formájában vizualizálni.

Például az  $X_i$  változókat fel lehet venni mint a gráf csomópontjai, a *Szülő*( $X_i$ ) halmaz elemeiből pedig éleket lehet húzni az  $X_i$  változó felé.

**Ez a gráf irányított körmentes gráf lesz.**

“Tegyük fel” hogy nem érted a fenti matekot. Józan paraszti megmagyarázása a Bayes hálónak:

A Bayes hálóval változók egy halmazát, és a köztük lévő feltételes függőségeket reprezentáljuk egy irányított, körmentes gráffal. Ideális olyan feladatra, ahol egy bekövetkezett eseményből próbáljuk meg megbecsülni az ő “okát”. Pl: Kap egy csomó tünetet, Bayes hálóval képesek vagyunk megbecsülni az őt okozható betegségeknek a valószínűségeit.

Bayes háló  $O(n * 2^k)$  tud tömöríteni. ahol  $k$  a szülők száma.  $n$  pedig a node-ok száma, ellenben az  $O(2^n)$  el szemben.

## Bayes háló konstruálás

Sok esetben definiálva vannak a **változók** és a **hatások** a változók között, és szakértőkkel kitöltjük az empirikus tudás segítségével a változókhoz tartozó **feltételes eloszlásokat**. Ilyenkor nem az **eloszlásból**, hanem az intuitív reprezentáció adott, ami definiál egy **teljes együttes eloszlést**, amit felhasználhatunk következtetésre.

Fontos, hogy a **láncszabály** esetén rögzített változósorrendtől függ a Bayes háló alakja.

## Függetlenség és Bayes háló

Ahhoz, hogy egy random node-ra megmondjuk, hogy milyey függetlenségi információjá van, a **Markov-takarót** tudjuk használni.

Pl:  $X$  markov takarója az a halmaz, amely  $X$  szülőinek,  $X$  gyerekeinek és  $X$  gyerekei szülőinek az uniója.

# Gépi tanulás: felügyelt tanulás problémája, döntési fák, naiv Bayes módszer, modellillesztés, mesterséges neuronhálók, k-legközelebbi szomszéd módszere

---

## Felügyelt tanulás problémája

Tapasztalati tények felhasználása arra, hogy egy racionális ágens teljesítményét növeljük.

## Felügyelt tanulás

Egy  $f : X \rightarrow Y$  függvényt keresünk, amely illeszkedik adott példákra. A **példák**  $((x_1, f(x_1)), \dots, (x_n, f(x_n)))$  alakban adottak.  $(x_i \in X)$

Pl.  $X$ : Emailek halmaza  $Y \{spam, -spam\}$

## Modellillesztés (rész szerintem)

Mivel az  $f$  függvényt általában nem ismerjük, ezért a feladat az, hogy **keresünk egy  $h : X \rightarrow Y$  függvényt amely  $f$ -et közelíti**

A  $h$  függvény **konzisztens** az adatokkal, ha  $h(x_i) = f(x_i)$  minden példára.

Ezt a  $h$  függvényt mindig egy  $H$  hipotézistérben keressük, azaz **egy függvényt mindig adott alakban keressük**.

Gyakorlatban elég, ha  $h$  elég közel van a példákhoz, mivel sokszor hibás, vagy zajos a tanuló példa, ezért káros lehet  $\rightarrow$  túltanulás következhet be pontos illeszkedés esetén.

Példa.

Az a  $h$ , amelyre  $h(x) = f(x)$  minden példára, egyébként  $h(x) = 0$ , az tökéletesen megtanulja a példákat, de lehető legrosszabban általánosít. Ez a **magolás**

A magolási probléma miatt **tömör reprezentációra** kell törekedni, lehetőleg tömörebb mint a példák listája. Ez az **Occam borotvája elv**: Ha más alapján nem tudunk választani, akkor a lehető legtömörebb leírást kell venni.

Tehát, hogy a fenti tulajdonságot elérjük fontos a  $H$  hipotézistér gondos meghatározása.

**A priori ismeretek fontossága:** A **tabula rasa** (tiszt a lappal történő indulás) tanulás a fentiek szerint lehetetlen. A  $H$  halmaz és az algoritmus maga a priori ismeretek alapján kerülnek megtervezésre.

## Döntési fák

### Induktív (felügyelt) tanulás konkrét példája.

Feltesszük, hogy  $x \in X$ -ben diszkrét változók egy vektora van,  $f(x) \in Y$ -ban pedig szintén valami diszkrét változó egy értéke, pl  $Y = \{igen, nem\}$

Mivel  $Y$  véges halmaz, osztályozási feladatról beszélünk, ahol  $X$  elemeit kell osztályokba sorolni, és az osztályok  $Y$  értékeinek felelnek meg. (Ha  $Y$  folytonos, akkor regresszióról beszélünk.)

*Tulajdonképpen osztályozás,  $X$  **elemeit** kell  $Y$  valamelyik **osztályába** sorolni.*

**Előnye**, hogy döntései megmagyarázhatók, mert emberileg értelmezhető lépésekben jutottunk el odáig.

### Kifejezőereje megegyezik az ítéletkalkulussal.

Mivel vannak valamilyen **ítéletek** (változó értékadások), egy **modell** (egy  $x \in X$  változóvektor), és egy **formula** (döntési fából adódoan).

**Fa  $\Rightarrow$  formula:** Vesszük az összes olyan levelet amelyen igen címke van, és az oda vezető utakban "és"-el összekötjük az éleket, és az utakat "vagy"-gyal összekötjük.

**Formula**  $\Rightarrow$  **fa**: A logikai formula igazságtábláját fel lehet írni fa alakban, ha vesszük a változók  $A_1, \dots, A_n$  felsorolását, az  $A_1$  a gyökér, értékei az élek (i/h általában), és az  $i$  edik szinten a fában minden pontban  $A_i$  van.

### Döntési fa építése:

adottak **pozitív és negatív példák felcímkézve**, tipikusan több száz (pl: Vendégek = tele, Várakozás = 10-30, Éhes=igen)

1. vegyük a **gyökérbe** azt a változót, ami a **legjobban szeparálja** a pozitív és negatív **példákat**. A legjobban szeparáló attribútumot az információtartalma, azaz entrópiája segítségével választhatjuk ki
2. Aztán ezt folytassuk **rekurzív** módon a gyerekein, tehát **nem rögzített változókból** választunk egy gyökeret a következő **részfához**

Speciális esetek amik megállítják a **rekurziót**:

- ha **csak pozitív vagy negatív példa van**, akkor **levélhez értünk, felcímkézzük** ezzel a levelet
- ha **üreshalmaz**, akkor a **szülő szerint többségi szavazattal címkézzük**
- ha **nincs több változó, de vannak negatív és pozitív példák is** (valszeg zajjal terhelt az adat), akkor szintén **többségi szavazattal címkézhetjük a levelet**

**entrópia**:  $-\sum_i p_i \log p_i$ ,

ahol  $\sum_i p_i = 1$ , amelynek minimuma 0, ami a maximális rendezettséget jelöli.

## Naiv Bayes módszer

**Statisztikai következtetési módszer**, amely adatbázisban található példák alapján ismeretlen példákat osztályoz.

Például emaileket akarunk spam vagy nem spamként osztályozni. Az emailben lévő szavakra meghatározzuk, hogy milyen valószínűséggel fordul elő egy normális üzenetben, vagy egy spam-ban. Ezután meg kell határozni, hogy milyen valószínűséggel kapunk normális üzenetet, és milyennel spam-et.

Legyen  $A$  és  $B_1, \dots, B_n$  a nyelvünk változói. (pl  $A$  lehet igaz, ha az email spam, hamis ha nem, illetve a  $B_i$  változó pedig az  $i$ . szó előfordulását jelezheti).

Tehát a feladat  $b_1, \dots, b_n$  konkrét email esetében meghatározni, hogy  $A$  mely értékekre lesz a  $P(A|b_1, \dots, b_n)$  **feltételes valószínűség maximális**

Ehhez a következő átalakításokat illetve függetlenségi feltevéseket tesszük:

$$P(A|b_1, \dots, b_n) = \alpha P(A) P(b_1, \dots, b_n|A) \approx \alpha P(A) \prod_{i=1}^n P(b_i|A).$$

Itt az első egyenlőségjel a Bayes tétel alkalmazása, ahol  $\alpha = 1/P(b_1, \dots, b_n)$ . Mivel csak  $A$  értékei közötti sorrendet keresünk, és  $\alpha$  nem függ  $A$ -tól, az  $\alpha$  értéke nem érdekes.

A második pedig a naiv Bayes feltevést fogalmazza meg. Nem biztos, hogy teljesül az egyenlőség viszont könnyen tudunk adatbázisból valószínűségeket számolni.

Fontos kiemelni, hogy a szavakat nem nézzük milyen sorrendbe írjuk fel. pl a "Dear Friend" és a "Friend Dear" is ugyanakkora értéket fog kapni. **Nem tesz fel közöttük semmilyen függőséget.**

Ha akarunk **tesztelni egy adott kérdést**, hogy pl azok a szavak az adott levélben spam/nem spam, akkor felírjuk hozzá a fenti képlettel **mindkettő esetben** a valószínűséget, és **amelyikhez közelebb van oda fog kerülni.**

Amennyiben az egyik szó nincs a másik szótárában, akkor az automatikusan 0 valószínűséget fog felvenni, ezért néha adnak hozzá minden szó előforduláshoz egy  $\alpha$  számot, tipikusan 1-et

## Modellillesztés

Sztem már leírtam fentebb szóval igazából a tanuló példákra illesztett  $h$  függvényt. Lásd felügyelet tanulás

## Mesterséges neuronhálók

A mesterséges neuron a következőképpen épül fel

- **Bemenet:**  $x = [x_1, \dots, x_n]$  vektor
- **Súlyok:**  $w = [w_1, \dots, w_n]$  súlyvektor
- $w_0$  bias weight, eltolássúly
- $x_0$  pedig fix bemenet, mindig  $-1$
- először minden bemeneti értéket megszorozza a hozzá tartozó súllyal, ezeket összeadja, majd kivonja belőle az eltolássúlyt
- majd a kapott értéken alkalmazzuk az **aktivációs függvényt**

Az aktivációs függvény célja, hogy 1-hez közeli értéket adjon, ha jó input érkezik, és 0-hoz közelít, ha rossz.

Példa aktivációs függvények:

- **küszöbfüggvény:** 0, ha az input  $\leq$  mint 0, 1, ha nagyobb (perceptron)
- **szigmoid függvény:**  $g(x) = 1/(1 + e^{-x})$
- **Rectifier aktiváció:**  $g(x) = \max(0, x)$  (ReLU)

Neuronokból hálózatokat szokás építeni. Egy hálózatnak lehet több rétege is. Van egy input, egy output és lehet több rejtett rétege is. Egy rétegen belül a neuronok között nincs kapcsolat, csak a rétegek között (előrecsatolt hálózatok).

## k-legközelebbi szomszéd módszere

Adottak  $(x_1, y_1), \dots, (x_n, y_n)$  példák.

Adott  $x$ -re az  $y$ -t az  $x$ -hez "közeli" példák alapján határozzuk meg. **Hogyan?**

1. Keressük meg  $x$  **k legközelebbi szomszédját**
  - Leghasonlóbbakat nézzük a predikálandó egyedhez (hasonlósági függvény maximuma/távolság függvény minimuma).
  - Majd a  $k$  legközelebbi szomszéd osztálycímkeiből kiválasztja a leggyakoribbat és azt predikálja.
2.  $h(x)$  értéke ezen szomszédok  $y$ -jainak átlaga (esetleg távolsággal súlyozva) ha  $y$  folytonos,  
ha diszkrét, akkor pl. többségi szavazás

A legközelebbi szomszédot többféleképpen is meglehet nézni.

1. Diszkrét esetben Hammington távolság: különböző jellemzők száma
2. Folytonos esetben euklideszi távolság, vagy manhattan távolság.

## 13. LP alapfeladat, példa, szimplex algoritmus, az LP geometriája, generálóelem választási szabályok, kétfázisú szimplex módszer, speciális esetek (ciklizáció-degeneráció, nem korlátos feladat, nincs lehetséges megoldás)

---

# LP alapfeladat

---

@kép (LP\_alapfeladat.JPG)

**LP alapfeladat:** Keressük adott lineáris célfüggvényt,  $\mathbb{R}^n$  értelmezési tartományú függvény szélsőértékét, értelmezési tartományának adott lineáris korlátokkal meghatározott részében.

**Lehetséges megoldás:** olyan  $p$  vektor, hogy  $p$ -t behelyettesítve  $x$ -be kielégíti a feladat feltételrendszerét.

**Lehetséges megoldási tartomány:** az összes lehetséges megoldás (vektor) halmaza.

**Optimális megoldás:** egy olyan lehetséges megoldás, ahol a célfüggvény felveszi a maximumát/minimumát.

## LP felírása + példa

1. Válasszuk meg a **döntési változókat** ( $x_1, x_2, \dots$ )
2. Határozzuk meg a célt és a **célfüggvényt** (max/min pl:  $\max 2x_1 + 5x_2$ )
3. Írjuk fel a **korlátozó feltételeket** (lineáris egyenlőtlenségek pl:  $2x_1 + 5x_2 \geq 10$ )
4. Határozzuk meg a **változók értelmezési tartományát** (előjel feltételek  $x_1 \geq 0$ )

Példa:

Egy cég **Tardisokat** és **Dalekeket** akar árulni. A **Tardis** darabja 25 euró míg a **Dalek** darabja 20 euró profitot jövedelmez.

A következő két héten a termék összerakására **200 munkaóra** áll rendelkezésre. Tardis és a Dalek is 5óra/db.

A **Dalekhez** szükség van egy **speciális lézerhabverőre**, amiből csak **24 darab** van raktáron.

A cég raktározási helye **320 négyzetméter**, amiből a Tardis **10 négyzetmétert** a Dalek **5 négyzetmétert** foglal el.

A cég **maximalizálni** szeretne

1.  $x_1$  az összeszerelendő Tardisok száma,  $x_2$  az összeszerelendő Dalekek száma.
2. **Célfüggvény felírása**
  - Tudjuk, hogy maximalizálni akar a cég, és tudjuk mennyi 1 darab Tardis és Dalek profitja.

- Tehát  $25x_1 + 20x_2 = z$

### 3. Korlátozó feltételek:

- **Összeszerelés időigénye:** Mivel mindkettő összeszerelése 5 óra és csak 200 óra áll rendelkezésre ezért:
  - $5x_1 + 5x_2 \leq 200$
- **A dalek lézerhabverője is fontos,** amiből 24 darab van
  - $x_2 \leq 24$
- **Raktározási feltétel is fontos,** Tardis:  $10m^2$  Dalek:  $5m^2$  és csak  $320m^2$  áll rendelkezésre.
  - $10x_1 + 5x_2 \leq 320$

Tehát:

$$\max z = 25x_1 + 20x_2$$

$$5x_1 + 5x_2 \leq 200$$

$$x_2 \leq 24$$

$$10x_1 + 5x_2 \leq 320$$

$$x_1, x_2 \geq 0$$

## Az LP geometriája

---

A lineáris programozás szoros kapcsolatban áll a **konvex geometriával**: Fogalmak, mint a bázismegoldás, lineáris feltétel, lehetséges megoldások halmaza, stb... mind megfeleltethető egy-egy geometria objektummal.

$R^n$ :  $n$ -dimenziós **lineáris tér** a valós számok felett – elemei az  $n$  elemű valós **vektorok**.

$E^n$ :  $n$ -dimenziós **euklideszi tér**: lineáris tér amin értelmezett egy **belső szorzat** és egy **távolság függvény**:

- **Belső szorzat:**  $\langle x, y \rangle = x^T y$ , ezáltal vektornorma:  $\|x\| = \sqrt{\langle x, x \rangle}$
- **Távolság:**  $d(x, y) = \|x - y\|_2$ , vagyis a koordinátakülönbség-négyzetek összegeinek a gyöke.

Így a lehetséges megoldások megadhatóak pontokként ( $n$ -dimenziós vektor) az  $E^n$  térben.

$x_1$  és  $x_2$  **közi szakasz**:  $\{x : x \in E^n, x = \lambda x_1 + (1 - \lambda)x_2\}$ , ahol  $\lambda \in [0, 1]$ .

Világos hogy ha  $\lambda = 1/2$ , akkor egy **felezőpontot** definiálunk.



**Csúcspont:** olyan pont, amely nem áll elő egyetlen ponthalmazbeli szakasz felezőpontjaként sem.

A lineáris feltételek **síkokat**, és **zárt féltereket** definiálnak az euklideszi térben hiszen:

- **n-dimenziós sík:**  $\{x : x \in E^n, a_1x_1 + a_2x_2 + \dots + a_nx_n = b\}$ , ahol  $a_1 \dots a_n, b$  rögzített számok.
- **n-dimenziós zárt féltér:**  $\{x : x \in E^n, a_1x_1 + a_2x_2 + \dots + a_nx_n \leq b\}$ , ahol  $a_1 \dots a_n, b$  rögzített számok.

Ezáltal, a **lehetséges megoldások halmaza** tulajdonképpen a feltételek által definiált **zárt félterek, és síkok metszete**, ami egy **konvex poliéder**: zárt, véges sok csúcsponttal rendelkező, konvex ponthalmaz.

### Példa:

Két változó esetén ábrázolhatjuk pl. a lehetséges megoldások halmazát koordináta rendszerben.

Minden feltétel egy egyenest határoz meg, ezeket berajzoljuk.

Ezzel valamilyen sokszöget kapunk meg, ennek a sokszögnek a csúcsainak a koordinátái lesznek a lehetséges megoldások.

@kép (LP\_Geometria\_1.JPG és LP\_Geometria\_2.JPG)

## Szimplex algoritmus

---

Ahhoz, hogy lecseréljük az **egyenlőtlenségeket egyenlőségekre** az LP alapfeladatban, adjunk hozzá minden egyenlőtlenség bal oldalához egy **mesterséges változót**.

Ezután **fejazzuk ki a mesterséges változókat az egyenlet átrendezésével**.

A kapott egyenletrendszeret hívjuk **szótár alak** nak.

### Terminológia

- **Természetes (vagy döntési) változók:** Standard alakú feladatban szereplő változók  $(x_1, x_2, \dots, x_n)$
- **Mesterséges változók:** A szótár felírásakor felvett új nemnegatív változók  $(x_{n+1}, \dots, x_{n+m})$
- **Bázisváltozók:** A szótár feltétel egyenleteinek a bal oldalán álló változók
- **Nembázis változók:** A szótár jobb oldalán álló változók.

- **Szótár bázismegoldása:** Olyan  $x$  vektor, amelyben a nembázis változók értéke 0, ezért a bázisváltozók értékei az őket tartalmazó egyenletek jobb oldali konstansai.
- **Lehetséges megoldás:** Olyan bázismegoldás, ami egyben lehetséges megoldás is, azaz a szótárra teljesül, hogy  $b_i \geq 0$

### A szimplex algoritmus:

- iteratív optimum keresés
- ismételt áttérés más szótárakra, a következő feltételek betartása mellett:
  - minden iteráció szótára ekvivalens az öt megelőzőével
  - minden iteráció bázismegoldásán a célfüggvény értéke nagyobb vagy egyenlő, mint az előző iterációban
  - minden iteráció bázismegoldása lehetséges megoldás

### Szimplex algoritmus menete:

1. A szótárban  $c_j \leq 0$  minden  $j = 1, 2, \dots, n$ -re?
  - Igen: Az aktuális bázismegoldás **optimális**
  - Nem: Folytatás 2. ponttal
2. Válasszuk a nembázis változók közül **belépőváltozónak** valamely  $x_k$ -t amelyre  $c_k > 0$  (Pozitív célfüggvény együttható)
3.  $a_{ik} \geq 0$  minden  $i = 1, 2, 3, \dots, m$ -re?
  - Igen: Az LP feladat **nem korlátos**
  - Nem: folytatás 4. ponttal
4. Legyen  $l$  valamely index, amelyre  $\left| \frac{b_l}{a_{lk}} \right|$  minimális és  $a_{lk} < 0$
5. Hajtsunk végre egy **pivot lépést** úgy, hogy  $x_k$  legyen a belépőváltozó és az  $l$ . feltétel bázisváltozója legyen a kilépő. Folytatás 1. ponttal

**Pivot lépés:** új szótár megadása egy bázis és nembázis változó szerepének felcserélésével

**Belépőváltozó:** az a nembázis változó, ami a következő szótárra áttéréskor bázisváltozóvá válik

**Kilépő változó:** az a bázisváltozó, ami a köv. szótárra áttéréskor nembázissá válik

**Szótárak ekvivalenciája:** két szótár ekvivalens, ha az általuk leírt egyenletrendszer összes lehetséges megoldása és a hozzájuk tartozó célfüggvényértékek rendre megegyeznek

Pivot lépés előtti és utáni szótárak ekvivalensek.

A szimplex algoritmus csak egy **keretalgoritmus**: nem teszi egyértelművé, hogy a 2. és a 4. pontban melyik változókat válasszuk, amennyiben többet is lehetne.

**Pivot szabály / Generálóelem választási szabály:** Olyan szabály, ami egyértelművé teszi, hogy a szimplex algoritmusban mely változók legyenek a belépő- és a kilépőváltozók, ha több változó is teljesíti az alapfeltételeket.

## Generálóelem választási szabályok

---

**Klasszikus szimplex algoritmus pivot szabály:** (Nem biztosan áll meg)

- A lehetséges belépőváltozók közül válasszuk a legnagyobb  $c_k$  értékűt. Több ilyen esetén a legkisebb indexűt.
- A lehetséges **kilépőváltozók** közül válasszuk a legkisebb  $l$  indexű egyenlet változóját, amelyre  $\frac{b_l}{a_{lk}}$  **minimális és**  $a_{lk} < 0$ .

**Bland szabály** (Biztosan megáll)

- **Oszlop:** a lehetséges belépőváltozók közül válasszuk a legkisebb indexűt
- **Sor:** A lehetséges változók közül válasszuk a legkisebb korlátot adó egyenlet (konstansokat kell nézni), ha egyenlő akkor a legkisebb indexű.

**Legnagyobb növekmény** (Nem biztosan áll meg)

- $\max(c_i * \min(|\frac{b_i}{a_{ij}}|)),$   
\*@kép (Legnagyobb\_novekmeny.JPG)

**Lexikografikus szabály** (Biztosan megáll)

- **kiegészítjük epszilonokkal mesterségesen a szótárat**
  - a lehetséges belépőváltozók közül a legnagyobb  $c_k$  értékűt válasszuk, több ilyennél a legkisebb indexűt
  - a lehetséges kilépőváltozók közül azt, amelynek  $l$  indexű egyenletére az együtthatókból álló vektor lexikografikusan a legkisebb

Véletlen pivot

- 1 valószínűséggel terminál

# Kétfázisú szimplex módszer

---

Ha van **negatív konstans**, akkor alkalmazható a kétfázisú szimplex módszer.

## Vegyünk egy segédfeladatot

- bevezetünk egy új,  **$x_0$  mesterséges segédváltozót**
- legyen  **$w$  az új célfüggvény**:  $w = -x_0$
- térjünk át szótár alakra
- vegyük a **legnegatívabb jobboldalú egyenletet**, és ebből fejezzük ki  $x_0$ -t
- a többiből a mesterséges változókat
- **ezután már egy lehetséges indulószótárt kapunk**

A standard feladatnak csak akkor létezik lehetséges megoldása, ha  $w = 0$  **a hozzá felírt segédfeladat optimuma**.

Ha a segédfeladatot megoldjuk a szimplexszel, és annak optimuma 0, akkor a megoldás utolsó szótárából könnyen felírhatunk egy olyan szótárt, amely az eredeti feladat szótára, és bázismegoldása lehetséges megoldás is egyben.

## A szótár felírásának lépései:

- az  $x_0 = 0$  feltételt elhagyjuk
- ha  $x_0$  **bázisváltozó**, akkor az egyenletének jobb oldalán lévő nem 0 együtthatójú változók egyikével végrehajtunk **egy pivot lépést**
- **elhagyjuk  $x_0$  megmaradt erőforrásait\***
- a **célfüggvény egyenletét lecseréljük az eredeti célfüggvényre**, amit átírtunk az aktuális bázisváltozóknak megfelelően

A következő fázisban pedig az átírt szótáron futtatjuk a szimplex algoritmust

## Speciális esetek

---

### Ciklizáció

**Degenerált iterációs lépés:** olyan szimplex iteráció, amelyben nem változik a bázismegoldás

Degenerált bázismegoldás: olyan bázismegoldás, amelyben egy vagy több bázisváltozó értéke is 0

Ciklizáció: ha a szimplex algoritmus valamely iterációja után egy korábbi szótárat visszakapunk, akkor az a ciklizáció

Ha a szimplex algoritmus nem áll meg, akkor ciklizál!

A ciklizáció elkerülhető megfelelő pivot szabály alkalmazásával (lexikografikus, Bland szabály)

A ciklizáció oka a degeneráció, azaz a bázisváltozók 0-vá válása a bázismegoldásban

## **Nem korlátos**

Ha az LP feladat maximalizálandó/minimalizálandó, és a célfüggvénye tetszőlegesen nagy/kicsi értéket felvehet, akkor nem korlátos a feladat.

Más szóval, ha tudunk oszlopot választani, de mikor sort választanánk minden együttható pozitív  $\rightarrow$  nem korlátos.

@kep(Nem\_korlatos.JPG)

## **Nincs lehetséges megoldás**

Ha a standard alakú LP feladatot kétfázisú szimplex módszerrel oldjuk meg, az első fázis eldönti, hogy van-e lehetséges megoldás.

Ha a felírt segédfeladatban az optimum értéke kisebb, mint nulla, akkor nincs lehetséges megoldás, ha 0, akkor van.

# **14. Primál-duál feladatpár, dualitási komplementaritási tételek, egész értékű feladatok és jellemzőik, a branch and bound módszer, a hátizsák feladat**

---

## **Primál-duál feladatpár**

---

Primál	Duál
$\max c^T x$	$\min b^T y$
$Ax \leq b$	$A^T y \geq c$
$x \geq 0$	$y \geq 0$

### A primál feladat

- maximalizálunk
- $c^T$  a célfüggvény együtthatóinak a vektora
- $A$  az együtthatók mátrixa
- $b$  a konstansok vektora

### A duál feladat

- minimalizálunk
- $b^T$  a célfüggvény együtthatóinak a vektora
- $A^T$  az együtthatók mátrixa
- $c$  a konstansok vektora
- $\leq$ -ket  $\geq$ -re cseréljük

### A duál feladat duálisa az eredeti primál feladat

Ahhoz, hogy duál feladatot megkapjunk a primálból a **következő lépéseket kell megtenni:**

- Transzponáljuk az  $A$  mátrixot
- *cseréljük fel  $b$  és  $c$  vektorok szerepét*
- cseréljük az egyenlőtlenségeket  $\geq$ -re
- Max helyett Min feladat

Gazdaági értelmezése: Tegyük fel, hogy az LP feladatunk **korlátozott erőforrások mellett maximális nyereséget célzó gyártási folyamat modellje**.

A duál feladat megoldásában az  $y_i^*$  a primál feladat  $i$  erőforrásához tartozó piaci ár, amit **marginális ár / árnyék ár**-nak nevezünk.

- Ez az **erőforrás értéke** az LP megoldójának a szemszögéből
- Az  $i$  erőforrás mennyiségén az egységnyi növelésével éppen  $y_i^*$  gal nő a nyereség.
- Viszont ha túl sok van egy erőforrásból, az nem érhet sokat.

- Továbbá  $y_i^*$ -nál többet **már nem érdemes fizetni az  $i$  erőforrásért**, míg kevesebbet igen.

## Dualitási komplementaritási tételek

---

### Gyenge dualitás

Ha  $x = (x_1, \dots, x_n)$  **lehetséges megoldása** a primál feladatnak és  $y = (y_1, \dots, y_n)$  **lehetséges megoldása** a duál feladatnak, akkor

$$c^T x \leq b^T y$$

**Vagyis a duális feladat bármely lehetséges megoldása felső korlátot ad a primál bármely lehetséges megoldására.**

### Erős dualitás

Ha  $x^* = (x_1^*, \dots, x_n^*)$  **optimális megoldása** a primál feladatnak, akkor a duál feladatnak is van **optimális megoldása**  $y^* = (y_1^*, \dots, y_n^*)$ , és a célfüggvényük megegyezik, azaz

$$c^T x^* = b^T y^*$$

Ha valamelyik  $i$ . feltétel egyenlet nem éles, azaz nem pontosan egyenlő a két oldal, akkor a kapcsolódó duál változó biztosan 0. Ha egy primál változó pozitív, akkor a kapcsolódó duális feltétel biztosan éles.

### Korlátosság és megoldhatóság

#### A korlátosság és a megoldhatóság nem függetlenek egymástól

- Ha a **primál nem korlátos**, akkor a **duálnak nincs lehetséges megoldása és fordítva**.
- Lehet, hogy egyiknek sincs lehetséges megoldása.
- Ha mindkettőnek van, akkor mindkettő korlátos
- A primál és a duál feladat egyidejű optimalitása ellenőrizhető.

### Komplementáris lazaság

Ha a primál-duál feladatpár

$\max c^T x$	$\min b^T y$
$Ax \leq b$	$A^T y \geq c$
$x \geq 0$	$y \geq 0$

Akkor azt mondjuk, hogy  $x = (x_1, \dots, x_n)$  és  $y = (y_1, \dots, y_m)$  komplementárisak, ha  $y^T(b - Ax) = 0$  és  $x^T(A^T - c) = 0$

Vagyis:

- Ha  $y_i > 0$ , akkor az  $x$ -et az  $i$ -edik egyenletbe helyettesítve egyenlőséget kapunk
- Ha  $x_i > 0$ , akkor  $y$ -t a duális feladat  $i$ -edik egyenletébe helyettesítve az egyenlőség teljesül.

### Tétel

Tegyük fel, hogy  $x$  a primál feladat optimális megoldása.

- Ha  $y$  a duál optimális megoldása, akkor  $x$  és  $y$  komplementáris.
- Ha  $y$  lehetséges megoldása a duálisnak és komplementáris  $x$ -szel, akkor  $y$  optimális megoldása a duálnak
- Létezik olyan lehetséges  $y$  megoldása a duálnak, hogy  $x$  és  $y$  komplementáris

**Ergo, ha van ilyen  $x$  és  $y$  vektor, amik a fenti "Vagyis"-ra teljesülnek, akkor az fixen optimális megoldása a primál-duál feladatpárnak**

## Egész értékű feladatok és jellemzőik

---

Tiszta egészértékű feladat (Integer Programming)

- Minden változónak egésznek kell lennie a megoldásban.

Vegyes egészértékű programozási feladat (Mixed Integer Programming)

- Csak néhány változóra követeljük meg, hogy egész legyen

0-1 IP

- minden változó értéke csak vagy 0 vagy 1 lehet



## LP lazítás

Egészértékű programozási feladat LP lazítása az az LP, amelyet úgy kapunk, hogy a változókra tett minden egészértékűségi vagy 0-1 megkötést eltörlünk.

- Bármelyik IP lehetséges megoldáshalmaz része az LP lazítás lehetséges megoldástartományának
- Maximalizálásnál az LP lazítás optimum értéke nagyobb egyenlő, mint az IP optimumértéke
- Ha az LP lazítás lehetséges megoldáshalmazának minden csúcspontja egész, akkor van egész optimális megoldása, ami az IP megoldása is egyben
- Az LP lazítás optimális megoldása bármilyen messze lehet az IP megoldásától.

## Branch and bound módszer

---

### 1. lépés

Megoldjuk az LP lazítást, ha a megoldás egészértékű, akkor done

### 2. lépés

Ha van lezáratlan részfeladatunk, akkor azt egy  $x_i$  nem egész változó szerint két részfeladatra bontjuk.

Ha  $x_i$  értéke  $x_i^*$ , akkor  $x_i \leq \text{floor}(x_i^*)$  és  $x_i \geq \text{ceil}(x_i^*)$  feltételeket vesszük hozzá egy egy részfeladatunkhoz

- a részproblémákat egy fába rendezzük
- a gyökér az első részfeladat, az LP lazítás
- a leszármazottai az ágaztatott részproblémák
- a hozzávett feltételeket az éleken adjuk meg
- a csúcsokban jegyezzük az LP-k optimális megoldásait

Lehet, hogy olyan részproblémát kapunk, aminek nincs lehetséges megoldása, ekkor ezt a levelet elhagyjuk

Találhatunk megoldásjelölteket is, ezek alsó korlátok az eredeti IP optimális értékére.

Ha találunk korábbi megoldásjelöltnél jobb megoldást, akkor a rosszabbat elvetjük.

Egy csúcs felderített/lezárt, ha

- nincs lehetséges megoldása

- megoldása egészértékű
- felderítettünk már olyan egész megoldást, ami jobb a részfeladat megoldásánál

Egy részfeladatot kizárunk, ha

- nincs lehetséges megoldása
- felderítettünk már olyan egész megoldást, ami jobb a részfeladat megoldásánál

## A hátizsák feladat

---

Egy olyan IP-t, amiben csak egy feltétel van, hátizsák feladatnak nevezünk.

Van egy hátizsákunk egy fix kapacitással, és tárgyaink, értékekkel és súlyokkal megadva.

Maximalizálni akarjuk a táskába rakott tárgyak értékét, úgy hogy a benne lévő tárgyak nem haladhatják meg a hátizsák kapacitását persze.

0-1 IP feladat, egy tárgyat viszünk vagy nem

Az LP lazítás könnyen számítható, a relatív hasznosság szerint tesszük a tárgyakat a táskába, vagyis az érték/súly hányadosuk szerint.

Branch and bound módszerrel ez is megoldható

Legrosszabb esetben  $2^n$  részfeladatot kell megoldani, NP nehéz a feladat.

Egészértékűnél még rosszabb,  $2^{M^n}$ , ahol M a lehetséges egészek száma egy változóra

## **15. Processzusok, szálak/fonalak, processzus létrehozása/befejezése, processzusok állapotai, processzus leírása. Ütemezési stratégiák és algoritmusok köteget, interaktív és valós idejű rendszerknél, ütemezési algoritmusok céljai. Kontextus-csere**

---

# Processzusok, szálak/fonalak, processzus létrehozása/befejezése, processzusok állapotai, processzus leírása

---

## Processzus

- **A végrehajtás alatt lévő program.**
- Szekvenciálisan végrehajtódó program
- **Egyidejűleg több processzus létezik:** A processzor idejét meg kell osztani az egyidejűleg létező processzusok között: időosztás (time sharing)
- Futó processzusok is létrehozhatnak processzusokat: Kooperatív folyamatok, egymással együttműködő, de amúgy független processzusok
- Az **erőforrásokat az OS-től kapják** (centralizált erőforrás kezelés)
- jogosultságokkal rendelkeznek
- Előtérben és háttérben futó folyamatok
- Processzusnak lehet **címtartománya**
  - Saját memória
  - Osztott memória

## Processus állapotok: KÉP HOZZÁ "ProcesszusAllapotok.JPG"

- **Futáskész:** készen áll a futásra, csak ideiglenesen le lett állítva, hogy egy másik processzus futhasson
- **Futó:** a proc bitrolja a CPU-t
- **Blokkolt:** bizonyos külső esemény bekövetkezéséig nem képes futni
- **Iniciális**
- **Terminális**
- **Felfüggesztett**

## Szálak/fonalak (thread)

- **Önálló végrehajtási egységként működő program, objektum, szekvenciálisan végrehajtható utasítás-sorozat**
- A proc hozza létre (akár többet is egyszerre)
- Osztozik a létrehozó proc erőforrásain
- Van saját **állapota, verme**
- Egy folyamaton belül több tevékenység végezhető párhuzamosan

- **Szálak megvalósítása:**

- A felhasználó kezeli a szálakat egy függvénykönyvtár segítségével. Ekkor a kernel (az operációs rendszer alapja (magja), amely felelős a hardver erőforrásainak kezeléséért) nem tud semmit a szálakról
- **A kernel kezeli a szálakat.** Szálak létrehozása és megszüntetése kernelhívásokkal történik

## Processzustáblázat és PCB

A proc nyilvántartására, tulajdonságainak leírására szolgáló memóriaterület.

Processusonként egy egy bejegyzés - Processzus vezérlő blokk (PCB)

### PCB tartalma:

- **azonosító:** processzus id
- processzus **állapota**
- **CPU állapot:** a kontextus cseréhez
- jogosultságok, prioritás
- birtokolt erőforrások

## Processzus létrehozása

- Futó processzusok is létrehozhatnak processzusokat: Kooperatív folyamatok, egymással együttműködő, de amúgy független processzusok
- Egyszerű esetekben megoldható, hogy minden processzus elérhető az OS elindulása után
- Általános célú rendszerekben szükség van a processzusok létrehozására és megszüntetésére
- **Processzusokat létrehozó események:**
  - Rendszer inicializálása
  - Felhasználó által kezdeményezett
  - Köteget feladat kezdeményezése
- **Az OS indulásakor sok processzus keletkezik:**
  - Felhasználókkal tartják a kapcsolatot: Előtérben futnak
  - Nincsenek felhasználóhoz rendelve:
    - Saját feladatuk van
    - Háttérben futnak
- **Lépései:**
  1. Memóriaterület foglalása a PCB számára

2. PCB kitöltése iniciális adatokkal
3. Programszöveg, adatok, verem számára memórafoglalás, betöltés
4. A PCB procok láncra fűzése, futáskész állapot. Ettől kezdve a proc osztódik a CPU-n.

## Processzus befejezése

- **Szabályos kilépés (exit(0)):** önkéntes, végzett a feladatával
- **Kilépés hiba miatt**
- **Kilépés végzetes hiba miatt:** önkéntelen, illegális utasítás, nullával osztás
- **Egy másik proc megsemmisíti:** önkéntelen, másik proc kill() utasítására
- **Lépései:**
  1. Gyermekek procok megszüntetése (rekurzívan)
  2. PCB procok láncról való levétele, terminális állapot. Ettől kezdve a proc nem osztódik a CPU-n
  3. Proc bitrokaiban lévő erőforrások felszabadítása (pl. fájlok lezárása)
  4. A memóriaterképnek (konstansok, változók, dinamikus változók) megfelelő memóriaterület felszabadítása
  5. PCB memóriaterületének felszabadítása

**Kölcsönös kizárás:** Ha egy processzus megosztott erőforrást, akkor a többi processzus tartózkodik ettől.

Kettő vagy több processzus egy-egy szakasza nem lehet átfedő, mert ilyen ez a két szakas egymásra nézve **kritikus szekciók** → ennek a megoldása az oprendszer feladata.

## Ütemezési stratégiák és algoritmusok köteget, interaktív és valós idejű rendszereknél, ütemezési algoritmusok céljai

---

### Ütemező

- Egy CPU áll rendelkezésre. Processzusok versengenek a CPU-ért
- Az OS dönti el, hogy melyik kapja meg a CPU-t
- Az **ütemező (scheduler)** hozza meg a döntést Ütemezési algoritmus

Ütemezés

- **Feladata:** Egy adott időpontban futáskész procok közül egy kiválasztása, amely a következőkben a CPU-t bitrokolni fogja
- **Mikor kell ütemezni?:** amikor egy processzus befejeződik vagy blokkolódik
- **Céljai:**
  - a CPU legyen jól kihasznált
  - az átfutási idő (proc létrejöttétől megszűnéséig eltelt idő) legyen rövid
  - egységnyi idő alatt minél több proc teljesüljön

## Ütemezés kötegelt rendszerekben

A manapság használatos op.rendszerek nem tartoznak a kötegelt rendszerek (: **Előre meghatározott sorrend szerint végrehajtandó feladatok együttese.**) világába, mégis érdemes röviden megemlíteni ezek ütemezési típusait.

- **Sorrendi ütemezés:** (First-Come First-Served)
  - Futásra kész folyamatok egy várakozó sorban helyezkednek el.
  - A sorban levő első folyamatot hajtja végre a központi egység. Ha befejeződik a folyamat végrehajtása, az ütemező a sorban következő feladatot veszi elő.
  - Új feladatok a sor végére kerülnek
  - Ha az aktuálisan futó folyamat blokkolódik, akkor a sorban következő folyamat jön, míg a blokkolt folyamat, ha újra futásra kész lesz, akkor a sor végére kerül, és majd idővel újra rá kerül a vezérlés.
- **Legrövidebb feladat először:** (Shortest Job First)
  - az a folyamat kerül először ütemezésre, melyiknek a legkisebb a futási ideje.
  - az alkalmazhatóság szempontjából nem ideális, ha nem tudjuk előre a folyamatok végrehajtási idejét.
- **Legrövidebb maradék futásidejű:**
  - Ismerni kell a folyamatok futási idejét előre.
  - Amikor új folyamat érkezik, vagy a blokkolás miatt egy következő folyamathoz kerül a vezérlés, akkor nem a teljes folyamat végrehajtási idejét, hanem csak a hátralévő időt vizsgálja az ütemező, és amelyik folyamatnak legkisebb a maradék futási ideje, az kerül ütemezésre
- **Háromszintű futásidejű:**

- A feladatok a központi memóriában vannak, közülük egyet hajt végre a központi egység. Előfordulhat, hogy a többi feladat közül ki kell rakni egyet a háttértárba, mivel a működés során elfogyhat a memória.
- Az a döntést, hogy a futásra jelentkező folyamatok milyen sorrendben kerüljenek be a memóriába, a bebocsátó ütemező hozza meg.

## Ütemezés interaktív rendszereknél

### • Round Robin

- Az ütemező beállít egy **időintervallumot** egy időzítő segítségével és amikor az **időzítő lejár megszakítást ad**.
- Megadott időközönként óramegszakítás következik be és ekkor az ütemező a következő folyamatnak adja a processzort.
- A folyamatokat egy sorban tárolja a rendszer, és amikor lejárt az időszelet, akkor az a folyamat, amelyiktől az ütemező éppen elveszi a vezérlést, a sor végére kerül
- Minden processzusnak egyforma fontosságot ad.

### • Prioritásos ütemezés

- Felmerül az igény, hogy nem feltétlenül egyformán fontos minden egyes folyamat.
- A folyamatokhoz egy fontossági mérőszámot, prioritást (prioritási osztályt) rendel hozzá
- A legmagasabb prioritású futáskész processzus kapja meg a CPU-t

## Ütemezés valós idejű rendszereknél

### Alapvető szerepe van az időnek

Ha a feladatainknak nemcsak azt szabjuk meg, hogy hajtódjanak végre valamilyen korrekt ütemezés szerint, hanem az is egy kritérium, hogy egy adott kérést valamilyen időn belül ki kell szolgálni, akkor valós idejű op.rendszerről beszélünk.

A megfelelő határidők betartása úgy valósítható meg, hogy **egy programot több folyamatra bontunk (ezek a folyamatok általában kiszámítható viselkedéssel rendelkeznek)**, és ezeknek a rövid folyamatoknak az **ütemező biztosítja a számukra előírt határidő betartását**

### • Szigorú valós idejű rendszer

- a határidő betartása kötelező

### • Toleráns valós idejű (soft real-time) rendszer

- a határidők kis mulasztása még elfogadható, tolerálható.

## Kontextus csere

Több idejűleg létező processzusok - Egyetlen processzor: a CPU váltakozva hajtja végre a procok programjait.

Kontextus csere: A CPU átvált  $P1$  procról a  $P2$  procra.

- $P1$  állapotát a CPU regisztereiből menteni kell az erre a célra fenntartott memóriaterületre. (IP, SP)
- $P2$  korábban memóriába mentett állapotát helyre kell állítani a CPU regisztereiben

**16. Processzusok kommunikációja, versenyhelyzetek, kölcsönös kizárás. Konkurens és kooperatív processzusok. Kritikus szekciók és megvalósítási módszereik: kölcsönös kizárás tevékeny várakozással (megszakítások tiltása, változók zárolása, szigorú váltogatás, Peterson megoldása, TSL utasítás). Altatás és ébresztés: termelő-fogyasztó probléma, szemaforok, mutex-ek, monitorok, Üzenet, adás, vétel. Írók és olvasók problémája. Sorompók**

---

**Processzusok kommunikációja, versenyhelyzetek, kölcsönös kizárás.**

---



Processzus: **A végrehajtás alatt lévő program a memóriában.**

## Processzusok kommunikációja

- A processzusoknak szükségük vannak a kommunikációra
  - **Adatok átadása az egyik folyamatból a másiknak** (Pipelining)
  - **Közös erőforrások használata** (memória, nyomtató, stb.)

Kettő vagy több processzus egy-egy szakasza nem lehet átfedő, azaz két szakasz egymásra nézve **kritikus szekciók**.

**Kritikus szekció:** A program az a része, ahol előfordulhat versenyhelyzet.

### Szabályok:

1. Legfeljebb egy proc lehet kritikus szekciójában
2. Kritikus szekción kívüli proc nem befolyásolhatja másik proc kritikus szekcióba lépését.
3. Véges időn belül bármely kritikus szekcióba lépni kívánó proc beléphet.
4. Processzusok sebessége közömbös

### Versenyhelyzet:

Amikor több párhuzamosan futó folyamat közös erőforrást használ. A futás eredménye függ attól, hogy az egyes folyamatok mikor és hogyan futnak.

- Kooperatív processzusok közös tárolóterületen dolgoznak (olvasnak és írnak).
- Processzusok közös adatot olvasnak és a végeredmény attól függ, hogy ki és pontosan mikor fut

**Megoldás:** Egyszerre csak egy folyamat lehet kritikus szekcióban. Amíg a folyamat kritikus szekcióban van, azt nem szabad megszakítani. Ebből a megoldásból származhatnak új problémák.

## Kölcsönös kizárás

---

- **Az a módszer, ami biztosítja, hogy ha egy folyamat használ valamilyen megosztott, közös adatot, akkor más folyamatok ebben az időben ne tudják azt elérni**
- pl.: egy adott időben csak egy processzus számára engedélyezett, hogy a nyomtatónak utasításokat küldjön

- Kölcsönös kizárás miatt előfordulható problémák:
  - **holtpont (deadlock):** processzusok egymásra befejeződésére várnak, hogy a várt erőforrás felszabaduljon
  - **éhezés (starvation):** egy processzusnak határozatlan ideig várnia kell egy erőforrás használatára

## Kritikus szekciók és megvalósítási módszereik: kölcsönös kizárás tevékeny várakozással (megszakítások tiltása, változók zárolása, szigorú váltogatás, Peterson megoldása, TSL utasítás).

---

Láthattuk, hogy a kritikus szekcióba való belépés nem feltétel nélküli. Hogyan biztosíthatjuk a kölcsönös kizárás teljesülését?

- **Hardware-es módszerek**
  - **Megszakítások tiltásával**
    - letiltjuk a megszakítást a kritikus szekcióba lépés után, majd újra engedélyezzük, mielőtt elhagyja azt, így nem fordulhat elő óramegszakítás, azaz a CPU nem fog másik processzusra váltani
    - jól használható, de általánosan nem biztos, hogy a legyszerencsebb
      - a legegyszerűbb hiba, hogy elfelejtjük újra engedélyezni a megszakítást a kritikus szekció végén
  - **TSL utasítás segítségével**
    - A mai rendszerekben a processzornak van egy „TSL reg, lock” (TSL EAX, lock) formájú utasítása (TSL – Test and Set Lock).
    - Ez az utasítás beolvassa a LOCK memóriaszó tartalmát a „reg” regiszterbe, majd egy nem nulla értéket ír a „lock” memóriacímre.
    - A CPU zárolja a memóriasínt, azaz tiltva van a memória elérés a CPU-knak a művelet befejezéséig.
    - A művelet befejezésekor 0 érték kerül a LOCK memóriaterületre
- **Software-es módszer**
  - **Változók zárolása**
    - Van egy **osztott zárolási változó**, aminek a kezdeti értéke 0.
    - Kritikus szekcióba lépés előtt a processzus **teszteli ezt a változót**.
      - Ha 0 az értéke, akkor 1-re állítja és belép a kritikus szekcióba.
      - Ha az értéke 1, akkor várakozik, amíg nem lesz 0.
  - **Szigorú váltogatás módszere**

- Folyamatosan pazarulja a CPU-t állandó tesztelése miatt.
- A kölcsönös kizárás feltételeit teljesíti, kivéve azt hogy **egyetlen kritikus szekción kívüli folyamat sem blokkolhat másik folyamatot**
- **Peterson-féle megoldás**
  - Van **két metódus a kritikus szekcióba való belépésre** (enter\_region) és **kilépésre** (leave\_region).
  - A kritikus szekcióba lépés előtt a processzus meghívja az enter\_region eljárást, kilépéskor pedig a leave\_region eljárást. Az enter\_region eljárás biztosítani fogja, hogy a másik processzus várakozik, ha szükséges.
  - **csak 2 processzus esetén működik**

## Altatás és ébresztés: termelő-fogyasztó probléma, szemaforok, mutex-ek, monitorok, Üzenet, adás, vétel.

---

### Altatás-ébresztés

Ahogy láttuk az előző, tevékeny várakozást használó versenyhelyzet-elkerülő megoldásokban a legfontosabb gond az, hogy processzoridőt pazarolnak. Ahhoz, hogy a drága processzoridőt se pazaroljuk, olyan megoldást lehet javasolni, ami vagy blokkolni tud egy folyamatot (aludni küldi), vagy fel tudja ébreszteni ebből a blokkolt állapotból.

A **tevékeny várakozás feloldására az egyik eszköz a sleep-wakeup** rendszerhívás páros. A lényege, hogy a **sleep rendszerhívás blokkolja a hívót**, azaz fel lesz függesztve, amíg egy másik processzus fel nem ébreszti. A **wakeup rendszerhívás a paraméterül kap egy processzus azonosítót, amely segítségével felébreszti az adott processzust**, tehát nem lesz blokkolva továbbá.

### Termelő-fogyasztó probléma

Két processzus osztozik egy közös, rögzített méretű tárolón. A *termelő* adatokat tesz bele, a *fogyasztó* kiveszi azokat.

Ha a tároló tele van és a gyártó elemet akar berakni, akkor elalszik, majd felébreszti a fogyasztó, ha egy elemet kivesz a tárolóból.

**Fordítva is:** ha a tároló üres, és a fogyasztó ki akar venni egy elemet, akkor elalszik, és majd felébreszti a gyártó, ha legyártott egy eleme

## Szemafor

- A szemafor 1965-ben Dijkstra hozta létre, amely **egy nagyon jelentős technika az egyidejű folyamatok kezelésére egy egyszerű egész érték használatával**
- Ez egy **megosztott egész változó**. Értéke pozitív vagy 0, és csak **várakozások** és **signal** műveleteken keresztül érhetőek el.
- Két metódusa van a *down* és az *up*. (általánosítható a *sleep* és *wakeup*-ra)
  - **down** metódus megnézi, hogy az adott folyamat a szemaforon nagyobb-e mint 0.
  - Ha igen, akkor **csökkent rajta eggyel**
  - Ha nem (tehát =0), akkor egyből alvásba rakja, nem növel rajta.
- Garantált, hogyha a szemafor elindult, akkor semelyik másik processzus nem érheti ez a szemafor, amíg a feladat le nem futott, vagy blokkoltba került.
- Az **up** metódus a szemafor elérését növeli.

## Mutex

- Olyan speciális szemafor, amelynek **csak két értéke** lehet
- Ha csak kölcsönös kizárás biztosítására kell a szemafor létrehozni, és nincs szükség annak számlálási képességére, akkor azt egy kezdőértékkel hozzuk létre.
- **Ezt a kétállapotú (értéke 0 és 1) szemafor** sok környezetben speciális névvel, az angol kölcsönös kizárás kifejezésből mutexnek nevezzük.
- Ha egy **folyamatnak zárolásra van szüksége**, a „**mutex\_lock**” eljárást hívja, míg ha **a zárolást meg akarja szüntetni**, a „**mutex\_unlock**” utasítást hívja.
- Aki másodszor (vagy harmadszor) hívja a „**mutex\_lock**” eljárást, az blokkolódik, és csak a „**mutex\_unlock**” hatására tudja folytatni a végrehajtást.

## Monitor

- Eljárások, változók és adatszerkezetek együttese egy speciális modulba összegyűjtve, hogy használható legyen a kölcsönös kizárás megvalósítására
- Legfontosabb tulajdonsága, hogy egy adott időpillanatban csak egy proc lehet aktív benne
- A processzusok bármikor hívhatják a monitorban lévő eljárásokat, de nem érhetik el a belső adatszerkezeteit (mint OOP-nál)
- **wait( c )**: alvó állapotba kerül a végrehajtó proc
- **signal( c )**: a c miatt alvó procot felébreszti

## Üzenet, adás, vétel.

- Folyamatok együttműködéshez információ cserére van szükség. Két mód:
  - közös tárterületen keresztül
  - kommunikációs csatornán keresztül (egy vagy kétirányú)
- Folyamat fommunikáció fajták:
  - Közvetlen kömmunikáció
    - csak egy csatorna létezik, és más folyamatok nem használhatják
  - Közvetett kommunikáció
    - Közbülső adatszerkezeten (pl. postaládán (mailbox)) keresztül valósul meg.
  - Aszimmetrikus
    - Adó vagy vevő megnevezi, hogy melyik folyamattal akar kommunikálni
    - A másik fél egy kaput (port) használ, ezen keresztül több folyamathoz, is kapcsolódhat.
    - Tipikus eset: a vevőhöz tartozik a kapu, az adóknak kell a vevő folyamatot és annak a kapuját megnevezni. (Pl. szerver, szolgáltató folyamat)
  - Üzenetszórás
    - A közeg több folyamatot köt össze.
- Műveletek:
  - send(cél, &üzenet)
  - receive(forrás, &üzenet)

## Írók és olvasók problémája. Sorompók.

---

### Írók és olvasók problémája

Több proc egymással versengve írja és olvassa ugyanazt az adatot. Megengedett az egyidejű olvasás, de ha egy proc írni akar, akkor más procok sem nem írhatnak se nem olvashatnak. (pl, adatbázisok, fájlok, hálózat)

Ha a folyamatos olvasók utánpótlása, az írok éheznek.

Megoldás: Érkezési sorrend betartása → csökken a hatékonyság

### Sorompók:

- Sorompó primitív

- Könyvtári eljárás
- Fázisokra osztjuk az alkalmazást
- **Szabály**
  - Egyetlen processzus sem mehet tovább a következő fázisra, amíg az összes processzus nem áll készen
- Sorompó elhelyezése mindegyik fázis végére
  - Amikor egy processzus a sorompóhoz ér, akkor addig blokkolódik ameddig az összes processzus el nem éri a sorompót
- A sorompó az utolsó processzus beérkezése után elengedi a azokat
- Nagy mátrix-okon végzett párhuzamos műveletek

# 1. Adatbázis-tervezés: A relációs adatmodell fogalma. Az egyed-kapcsolat diagram és leképezése relációs modellre, kulcsok fajtái. Funkcionális függőség, a normalizálás célja, normálformák

---

## A relációs adatmodell fogalma

---

A relációs adatmodell mind az adatokat, mind a köztük lévő kapcsolatokat kétdimenziós táblákban tárolja.

### Attribútum:

- névvel, értéktartománnyal megadott tulajdonság
- $Z$  értéktartományát  $\text{dom}(Z)$  jelöli
- **csak elemi típusú értékekből állhat** (numerikus, karakter, string)
- gyakran megadjuk az ábrázolás hosszát is

### Relációséma:

Ha  $A = \{A_1, \dots, A_n\}$  jelöli az **attribútumhalmazt** és a **séma neve**  $R$ , akkor a **relációsémát**  $R(A_1, \dots, A_n) = R(A)$  jelöli

- névvel ellátott attribútumhalmaz

- **névütközés esetén kiírhatjuk a tábla nevét is az attribútum elé**

### **A relációséma nem tárol adatot!**

Csak szerkezeti leírást jelent.

Az adatok relációkkal adhatók meg. Egy  $R(A)$  séma feletti reláció  $A$  értéktartományainak direktszorzatának egy részhalmaza (mindegyik attribútum értékei közül választunk egyet, és ezt egy vektorba pakoljuk). Egy ilyen reláció már megjeleníthető adattábla formájában, egy reláció a táblázat egy sorának felel meg.

## **Az egyed-kapcsolat diagram és leképezése relációs modellre**

---

### **EK-diagram**

Az egyed-kapcsolat modell konkrét adatmodelltől függetlenül, szemléletesen adja meg az adatbázis szerkezetét.

#### **Egyed vagy entitás**

- a valós világ egy objektuma
- szeretnénk róla információt tárolni az adatbázisban
- **egyed típus:** általánosságban jelent egy valós objektumot
- **egyed példány:** egy konkrét objektum
- **gyenge egyed:** ha az egyed nem határozza meg egyértelműen attribútumainak semmilyen részhalmaza

#### **Tulajdonság vagy attribútum**

- az egyed egy jellemzője
- **tulajdonságtípus:** pl felhasználók jelszava általánosságban
- **tulajdonságpéldány:** pl Egy konkrét jelszó
- az attribútumok egy olyan legszűkebb részhalmazát, amely egyértelműen meghatározza az egyedet, **kulcsnak** nevezzük

### **Kapcsolatok**

- egyedek között alakulhatnak ki
- **kapcsolattípus:** pl felhasználó és üzenet között

- **kapcsolatpéldány:** pl Kis József és a 69420. üzenet
- kapcsolatoknak is lehet tulajdonsága

Azt a modellt, amelyben az adatbázis a tárolandó adatokat egyedekkel, tulajdonságokkal és kapcsolatokkal írja le, **egyed-kapcsolat modellnek** nevezzük, a hozzá kapcsolódó diagramot pedig egyed-kapcsolat diagramnak.

A diagramon

- az egyedeket téglalappal
- a tulajdonságokat ellipszissel
- a kulcsot aláhúzással
- a kapcsolatokat rombusszal jelöljük.

## EK leképezése relációs adatmodellre

### Egyedek leképezése

- minden egyedhez egy relációsémát írunk fel, melynek neve az egyed neve, attribútumai pedig az egyed attribútumai, kulcsa pedig az egyed kulcsa
- **gyenge egyednél az attribútumokhoz hozzá kell venni a meghatározó kapcsolatokon keresztül csatlakozó egyedek kulcsattribútumait is**, külső kulcsként

### Összetett attr. leképezése

- összetett attribútumot helyettesítünk az őt alkotó elemi attribútumokkal

### Többértékű attribútumok leképezése

- **egyik lehetőség:**
  - eltekintünk attól, hogy többértékű, és egyszerű szöveggént tároljuk
  - hátránya, hogy nem kezelhetők külön külön az elemek
- **másik lehetőség:**
  - minden sorból annyit veszünk fel, ahány értéke van a többértékű attribútumnak
  - hátránya a sok fölösleges sor
  - kulcsok elromlanak
  - kerülendő



- **harmadik lehetőség:**

- új táblát veszünk fel, ahova kigyűjtjük, hogy melyik sorhoz milyen értékei tartoznak a többértékű attribútumnak
- akár külön kigyűjthetjük egy táblába az összes lehetséges értékét a többértékű attribútumnak, és egy kapcsolótáblával kötjük össze az egyeddel

## **Kapcsolatok leképezése**

- minden kapcsolathoz felvesszünk egy új sémát
- neve a kapcsolat neve, attribútumai a kapcsolódó egyedek kulcsattribútumai és a kapcsolat saját attribútumai
- meg kell határozni ennek a sémának is a kulcsát
- ha ez a kulcs megegyezik valamelyik kapcsolt egyed kulcsával, akkor ez a séma beolvasztható abba az egyedbe, ezt hívjuk konszolidációnak, ez a gyakorlatban egy lépésben is elvégezhető persze
- **1:1** kapcsolat esetén az egyik tetszőlegesen választott egyedbe beolvaszthatjuk a kapcsolat sémáját
- **1:N** kapcsolat esetén az N oldali egyedet bővítjük a másik egyed kulcsattribútumaival, és a kapcsolat saját attribútumaival
- **N:M** kapcsolat esetén új sémát veszünk fel

## **Specializáló kapcsolatok leképezése**

Minden megközelítésnek lehetnek hátrányai, mérlegelnünk kell

### **Első lehetőség**

- főtípus és altípus is külön sémában, és az altípus attribútumai közé felvesszük a főtípus attribútumait is
- minden egyedpéldány csak egy táblában fog szerepelni

### **Második lehetőség**

- minden altípushoz új séma, de abban csak a főtípus kulcsattribútumai jelennek meg
- minden egyedpéldány szerepel a saját altípusának táblájában és a főtípus táblájában is

### **Harmadik lehetőség**

- egy közös tábla az összes lehetséges attribútummal

- minden sorban csak a releváns cellákat töltjük ki

## Kulcsok fajtái

---

### Szuperkulcs

- egyértelműen azonosítja a tábla sorait
- $R(A)$  bármely két sora különbözik a szuperkulcson
- mivel a táblában általában nem engedünk meg ismétlődő sorokat, ezért ha az összes attribútumot vesszük, az mindig szuperkulcs

### Kulcs

- olyan szuperkulcs, amelynek egyetlen valódi részhalmaza sem szuperkulcs
- ha egyelemű, **egyszerű kulcsnak** nevezzük
- ha többelemű, **összetettnek**
- előfordulhat, hogy van több kulcs is, ekkor kiválasztunk egyet
- a kiválasztott kulcsot elsődleges kulcsnak nevezzük

### Külső kulcs

- másik, vagy ugyanazon séma elsődleges kulcsára vonatkozik

Mind az elsődleges kulcs és a külső kulcsok is a sémára vonatkozó feltételek, függetlenek az adatoktól

## Funkcionális függőség

---

P és Q attribútumhalmazok, az  $R(A)$  sémán

P-től funkcionálisan függ Q, ha bármilyen R feletti tábla esetén ha P-n megegyezik két sor, akkor Q-n is meg fog egyezni.

Triviális, ha Q részhalmaza P-nek, és nemtriviális, ha P-nek és Q-nak nincs közös attribútuma.

PI a felhasználónévtől funkcionálisan függ az email sokszor.

# Normalizálás célja, normálformák

---

Tárolhatnánk az összes adatunkat egy nagy táblában is, de ilyenkor gondok merülhetnek fel az adatbázisműveletek során, illetve nagyon redundáns lenne az adattárolás. A normalizálás célja kisebb táblák létrehozása a redundancia elkerülése érdekében.

## Normálformák

Dekompozíció segítségével megszüntetjük lépésről lépésre a redundanciát úgy, hogy a sémában lévő függőségekre egyre szigorúbb feltételeket adunk.

**Elsődleges, másodlagos attribútum:** szerepel a séma valamelyik kulcsában, ha nem akkor másodlagos

Tranzitív, közvetlen függés: Ha  $X$ -től függ  $Z$ , és van olyan  $Y$ , hogy  $X \rightarrow Y$  és  $Y \rightarrow Z$ , ellenkező esetben közvetlenül függ

1NF:

- Ha az attribútumok értéktartománya csak egyszerű adatokból áll (nincs többszörös vagy összetett attribútum)

2NF:

- Ha minden másodlagos attribútum teljesen függ bármely kulcstól

3NF:

- Minden másodlagos attribútum közvetlenül függ bármely kulcstól, azaz nincs tranzitív függés

BCNF:

- Egy relációséma Boyce-Codd normálformában van, ha bármely nemtriviális  $L \rightarrow B$  függés esetén  $L$  szuperkulcs.

## 2. Az SQL adatbázisnyelv: Az adatdefiníciós nyelv (DDL) és az

# adatmanipulációs nyelv (DML).

## Relációsémák definiálása, megszorítások típusai és létrehozásuk. Adatmanipulációs lehetőségek és lekérdezések

---

### SQL

---

Structured Query Language, egy lekérdező nyelv. Arra szolgál, hogy adatokat kezeljünk vele.

- beszúrás
- törlés
- módosítás
- lekérdezés

A nyelv elemeit két fő részre oszthatjuk.

### Az adatdefiníciós nyelv (DDL)

---

Ide tartoznak az adatbázisok, sémák, típusok definíciós utasításai, pl:

- CREATE DATABASE
- CREATE TABLE
- ALTER TABLE
- DROP TABLE
- **CREATE TRIGGER**: Nem tábla létrehozásra van

#### Triggerekről pár infó

Olyan kis programok, aktív elemek az adatbázisokban, amelyek valamilyen

- dataaktualizáló művelet vagy
  - Amelyek minden adatmanipulációs művelet esetén végrehajtódnak (Insert, update)
- rendszerszintű művelet esetén hajtódnak végre.

- PI felhasználó bejelentkezése stb.

## Az adat manipulációs nyelv (DML)

---

Ide tartoznak a beszűrő, módosító, törlő, lekérdező utasítások.

- INSERT INTO
- UPDATE
- DELETE FROM
- SELECT

Egyes irodalmak különválasztják a lekérdező utasításokat a manipulációs utasításoktól.

## Relációsémák definiálása, megszorítások típusai és létrehozásuk

---

Relációsémákat a

```
CREATE TABLE tablanev(  
    mező1 típus [oszlopfeltételek],  
    ....  
    [tablafeltételek]  
);
```

utasítással hozhatunk létre. A sémák különböznek a tábláktól, és nevével ellentétben a CREATE TABLE utasítás csak a relációsémát hozza létre. A tábla már az adatrekordok halmazát jelenti.

Relációsémákat módosítani a

Új oszlop hozzáadás:

```
ALTER TABLE táblanév ADD (uj_oszlop TÍPUS [oszlopfeltételek])
```

Oszlop módosítása:

```
ALTER TABLE táblanév MODIFY (meglevo_oszlop TÍPUS [oszlopfeltetl])
```

Oszlop törlése:

```
ALTER TABLE táblanév DROP (oszlop)
```

Adatbázis törlése:

DROP TABLE táblanév;

## Megszorítások

### Oszlopfeltételek:

Csak az adott mezőre vonatkoznak

- **PRIMARY KEY**, az elsődleges kulcs
- **UNIQUE**, kulcs, minden érték egyszer fordulhat elő az oszlopban
- **NOT NULL**, az oszlop értéke nem lehet NULL, azaz kötelező kitölteni
- **REFERENCES T(oszlop)**, a T tábla oszlop oszlopára vonatkozó külső kulcs
- **DEFAULT tartalom**, az oszlop alapértelmezett értéke tartalom lesz

### Táblafeltételek

Ha több oszlopra is vonatkoznak feltételek, azt itt tudjuk megadni.

- **PRIMARY KEY(oszloplista)**, az elsődleges kulcs/kulcsok
- **UNIQUE (oszloplista)**, kulcs, minden érték egyszer fordulhat elő az oszlopban
- **FOREIGN KEY (oszloplista) REFERENCES T(oszloplista)**, a T tábla oszloplista oszloplistájára vonatkozó külső kulcs

### Külső kulcs feltételek és szabályok

Az integritás megőrzése szempontjából a külső kulcsokhoz meghatározhatjuk azt is, hogy hogyan viselkedjenek a hivatkozott kulcs törlése vagy módosítása esetén.

### ON DELETE

- **RESTRICT**, ha van a törlendő rekord kulcsára van vonatkozó külső kulcs, megtiltjuk a törlést
- **SET NULL**, a törlendő rekord kulcsára hivatkozó külső kulcs értékét NULL-ra állítjuk
- **NO ACTION**, a törlendő rekord kulcsára vonatkozó külső kulcs értéke nem változik
- **CASCADE**, a törlendő rekord kulcsára hivatkozó külső kulcsú rekordok is törlődnek

pl:

```
felhasznalonev VARCHAR(20) REFERENCES felhasznalo(felhasznalonev) ON DELETE
```

SET NULL

## ON UPDATE

- **RESTRICT**, ha van a módosítandó rekord kulcsára van vonatkozó külső kulcs, megtiltjuk a módosítást
- **SET NULL**, a módosítandó rekord kulcsára hivatkozó külső kulcs értékét NULL-ra állítjuk
- **NO ACTION**, a módosítandó rekord kulcsára vonatkozó külső kulcs értéke nem változik
- **CASCADE**, a módosítandó rekord kulcsára hivatkozó külső kulcsú rekordok is az új értékre változnak

```
felhasznalonev VARCHAR(20) REFERENCES felhasználó(felhasznalonev) ON UPDATE CASCADE
```

## Táblákra és attribútumokra vonatkozó megszorítások

Elsődleges feladata, hogy megelőzzük az adatbeviteli hibákat, és elkerüljük a hiányzó adatokat a kötelező mezőkből.

**NOT NULL:** a cella értékét kötelező kitölteni, nem lehet NULL

**CHECK (feltétel):** ellenőrző feltétel arra, hogy milyen értékeket vehet fel az adott oszlop

**DOMAIN:** értéktartomány egy oszlop értékeire vonatkozóan

## Adatmanipulációs lehetőségek és lekérdezések

---

### Adatok beszúrása:

Ha csak adott oszlopoknak akarunk értéket adni (pl mert nem kötelező, vagy alapértelmezett érték):

```
INSERT INTO táblanév (oszloplista) VALUES (értéklista);
```

Ha minden oszlop értékét ki akarjuk tölteni:

```
INSERT INTO táblanév VALUES (értéklista);
```

Adatok módosítása:

```
UPDATE táblanév SET  
    oszlop=kifejezés  
    [oszlop2=kifejezés2]  
    [WHERE feltétel];
```

Módosítjuk egy vagy több oszlop értékét az adott táblában, azokon a sorokon, amelyek eleget tesznek a WHERE záradékban tett feltételnek.

### Adatok törlése:

```
DELETE FROM táblanév [WHERE feltétel];
```

Töröljük az összes rekordot a táblából, amelyek megfelelnek a WHERE záradékban megadott feltételnek.

### Lekérdezések:

```
SELECT oszloplista FROM tábla;
```

A megadott oszlopokat kilistázza az adott táblából. oszloplista helyére megadható \*, ha az összes oszlopot listázni akarjuk.

### Teljes szintaxisa:

```
SELECT [DISTINCT] oszloplista FROM táblalista  
    [WHERE feltétel]  
    [GROUP BY oszloplista]  
    [HAVING csoportfeltétel]  
    [ORDER BY oszloplista [DESC]];
```

**DISTINCT:** csak a különböző sorokat írja ki

**FROM táblalista:** a táblalistában megadott táblákból képez Descartes szorzatot

**WHERE:** kiválasztás a feltétel szerint

**GROUP BY:** csoportosítás az oszloplistában szereplő oszlopok szerint

**HAVING:** a csoportosítás után a csoportokra vonatkozó feltétel

**ORDER BY:** az oszloplistában szereplő adatok rendezése abc szerint növekvő vagy csökkenő sorrendben

### Összesítő függvények



Leggyakrabban a **GROUP BY-jal együtt** szoktuk használni, de enélkül is lehet. **Leginkább a SELECT utáni oszloplistában**, de a **where-ben** és a **having-ban** is használható. Az eredményoszlopokat AS kulcsszóval el is nevezhetjük.

**MIN(oszlop):** az oszlopban lévő minimumot adja vissza

**MAX(oszlop):** maxot

**AVG(oszlop):** az oszlop átlaga

**SUM(oszlop):** az oszlop összege

**COUNT ([DISTINCT] oszlop):** az eredményben szereplő (különböző) rekordok száma

### **Természetes összekapcsolás**

**SELECT \* FROM T1, T2 WHERE T1.X = T2.X;**

X az most egy oszlop, egy kulcs-külső kulcs kapcsolat.

Erre használható még SQL-ben az **INNER JOIN** kulcsszó is.

```
SELECT * FROM T1, T2 INNER JOIN T2 ON T1.X = T2.X;
```

Használható még a NATURAL JOIN kifejezés is, de ez egy picit máshogy működik. Ennek a használatához a két tábla közös attribútumhalmaza ugyanazokat az oszlopneveket tartalmazza mindkét táblában és a párosított oszlopok típusa is megegyezik. Ebből kifolyólag nem kell megadnunk a kapcsolódó, kulcs és külső kulcs oszlopokat. A közös oszlop csak egy példányban jelenik majd meg.

```
SELECT * FROM T1 NATURAL JOIN T2;
```

### **Jobboldali, baloldali és teljes külső összekapcsolás:**

Valamelyik, vagy mindkét tábla összes rekordja szerepelni fog az eredményben.

**Baloldali összekapcsolásnál (LEFT JOIN)** a baloldali tábla minden rekordja megmarad, és ezekhez a rekordokhoz párosítjuk a jobboldali tábla rekordjait.

**Jobboldalinál (RIGHT JOIN) pont fordítva.**

**Teljes összekapcsolásnál (FULL OUTER JOIN)** pedig mindkét tábla összes rekordja megmarad, és mindenhol a hiányzó helyeken NULL értékek lesznek.

### **Theta kapcsolat**

Nem feltételezünk, hogy lenne a két táblának közös kapcsolómezője. → **Descartes szorzat**

```
SELECT * FROM T1 , T2 WHERE feltétel ;
```

**Lekérdezések eredményén, amikor ugyanannyi és ugyanolyan típusú oszlopot kérünk le, használhatunk halmazműveleteket is, pl **UNION** vagy **INTERSECT**.**

## Alkérdesek

Alkérés tulajdonképpen egy **SELECT** utasítás. Leginkább a **WHERE** és **HAVING** feltételeibe szoktuk megadni.

Lehetőség van megadni őket beszűrő, módosító és törlő utasításokban.

Pl: INSERT INTO táblanév [(oszloplista)] AS (alkérés);

# 3. Simítás/szűrés képtérben (átlagoló szűrők, Gauss simítás és mediánszűrés); élek detektálása (gradiens-operátorokkal és Marr-Hildreth módszerrel)

---

**Zaj:** A képpont-intenzitások nemkívánatos változása



## Simítás/szűrés képtérben

---

### Átlagoló szűrés

Vesszük egy képpontnak egy környezetét, és vesszük ebben a környezetben az összes képpont átlagát. Ezzel az átlag lesz a képpont új értéke.

Ezt az átlagolást konvolúcióval is végezhetjük, ahol a konvolúciós maszkunkban minden érték  $\frac{1}{n^2}$ , ha  $n * n$ -es a maszk.

- minél nagyobb környezetet nézünk, annál erősebb a simító hatás
- **haszna:**
  - csökkenti a zajt
- **kára:**

- gyengíti az éleket
- homályossá teszi a képet
- **súlyozott átlagolást is lehet csinálni** - konvolúció
  - a legnagyobb súly az aktuális pontunknak legyen
  - ahogy távolodunk a ponttól, annál kisebbek legyenek a súlyok

## Gauss simítás

- ahogy távolodunk a ponttól, annál kisebbek legyenek a súlyok
- erre nagyon jó a gauss harang
- minden sűrűségfüggvény integrálja 1
  - minél nagyobb a  $\sigma$  (szigma), annál szélesebb, de annál alacsonyabb a harang
  - ezzel szépen lehet jeleket simítani
- binomiális együtthatók jól közelítik a normális eloszlás görbáját
- van 2D gauss is, harang alakú

## hogyan lehet gauss függvényt közelíteni diszkrét értékekkel?

- vegyük a binomiális együtthatókat tartalmazó sorvektort, és osszuk el minden elemet  $2^n$ -nel
- ezt szorozzuk össze a transzponáltjával, és így kapjuk a gauss görbe közelítését

Hozzájuthatunk így diszkrét gauss eloszlású  $n \times n$ -es konvolúciós maszkokhoz, és az ilyenekkel vett konvolúció a Gauss szűrés

Az élek itt is rombolódnak

Lehet olyat is, hogy csak akkor simítunk, ha az adott képpont intenzitásának környezeti átlagtól való eltérése meghalad egy  $T$  küszöbértéket

## Medián szűrés

**medián** = sorbarendezzük az értékeket, és a középsőt vesszük

$$\min \leq \text{med} \leq \max$$

medián nem lineáris

### medián szűrés:

nézzük egy környezetét a pontnak, ezt rendezzük sorba, és a középső érték legyen a

képpont új értéke

**só-bors zaj eltüntetésére szépen alkalmas**

tiszta képet kapunk, ha pl 5x5-ös környezetben nézve a 25 képpontból max 12 teljesen fekete vagy teljesen fehér képpont van

**megszünteti az egyedi, és kis kiterjedésű kiugrásokat**

**jobban megőrzi az éleket**, mint az átlagolás

**nagy kiterjedésű zajfoltoknál jel-elnyomó**

a zajt hagyja meg, és a lényeg tűnhet el

## Élek detektálása

---

**él ott van a képen, ahol az intenzitás valamilyen irányban felugrik, vagy lecsökken**

élek nagyon fontosak a látásban, **ahol markánsak az élek, azokat jól érzékeljük**

Tipikus élprofilok:

- lehet ideális/lépcsős él
- lejtős él
- tető
- vonal
- zajos

**tangens:** érintő iránytangense/meredeksége

**első derivált:** hol vannak szélsőértékek, monotonitás

derivált pozitív, nő, negatív, csökken

él ott van, ahol az intenzitásprofil első deriváltja nagy

## Gradiens operátorokkal

---

többszörös függvényeket is lehet deriválni, pl parciálisan

egyik változót lerögzítjük, és a másik szerint deriválunk

**gradiens:** első parciális deriváltakból alkotott vektor

2D-ben az érintőre merőleges vektor

ennek van két komponense (x és y szerint vett derivált)

## **gradiens nagysága - magnitúdó**

első vektornormánál a **gradienskomponensek abszolútértékének az összegét** nézzük

2D-ben a kettes vektornorma az a pitagorasz tételből jön

2D-ben van a gradiensnek iránya is  **$\arctan(y/x)$**

él iránya a gradiensre merőleges

diszkrét gradiens operátorok

## **roberts, prewitt, sobel, frei-chen**

mind a négy módszer konvolúciós maszkpárokat alkalmaz

### **1. roberts operátor**

- adott két 3x3-as mátrix, ha az egyikkel konvolválunk, akkor az x irányú parciális deriváltat közelítjük, ha a másikkal, akkor az y irányút igazából nem is kell konvolúció
- **x**: a képpont értékéből kivonjuk az északkeleti szomszédját
- **y**: a képpont értékéből kivonjuk az északnyugati szomszédját

	0		0		-1				-1		0		0	
	0		1		0				0		1		0	
	0		0		0				0		0		0	

**pro: könnyen számítható**

**kontra: zajérzékeny**

### **2. prewitt operátor**

- itt is két 3x3-as maszk van, csak kicsit más, mint az előbb
- **x**: baloldali oszlop csupa 1, jobboldali csupa -1, középen 0
- **y**: felső sor -1, alsó sor 1, középen 0
- 

	1		0		-1				-1		-1		-1	
	1		0		-1				0		0		0	
	1		0		-1				1		1		1	

### **3. sobel operátor**

- két 3x3 maszk
- ha négyzet mozaikon mintavételezett a képünk akkor ami két pixel élen osztozkodik (vízszintesen vagy függőlegesen szomszédos), akkor azok közelebb vannak egymáshoz, mintha csak csúcson érintkeznének

1	0	-1		-1	-2	-1	
2	0	-2		0	0	0	
1	0	-1		1	2	1	

#### 4. frei-chen operátor

- ugyanaz, mint a sobel, csak 2 helyett gyök(2)

#### gradiens maszk tervezése x irányban

- szimmetrikus ne húzzon el se balra, se jobbra
- asszimmetrikus ne húzzon el se fel, se le
- legyen az összege az elemeknek 0

8 irányban élt kereső gradiens operátorok **compass operátorok**

#### prewitt compass operátor

- 8 különböző maszkkal dolgozik, a 8 égtáj irányába maszkelemek összege 0

#### robinson-3 compass operátor

- 3-féle elem szerepel a maszkokban (1, -1)

#### robinson-5 compass operátor

- 5-féle elem (-2, -1, 0, 1, 2)

#### kirsch compass operátor

- 0, -3, 5 értékek szerepelnek benne

#### Marr-Hildreth módszer

**LoG:** Gauss laplace transzformáltja.

**Laplace:** Gradiens önmagával vett szorzata, amit másodfokú deriváltak közelítésére használnak. SKALÁR

1. konvolváljuk a képet egy vagy több alkalmas **LoG függvénnyel**

2. keressünk közös **nulla átmeneteket**

nulla átmenet ott van, ahol adott pont kis környezetében előfordulnak pozitív és negatív értékek is.

**eredménye** mindig egy bináris éltérkép

lehetnek fantomélek is, de ez a gyakorlatban elhanyagolható

**LoG a frekvenciatérben**

konvolúciós tétel szerint **f\*LoG** gyorsan számítható fourier-trafóval meg pontonkénti szorzással

adott szigmára előre kiszámíthatjuk a sombrero fourier trafóját  
ezt is eltárolhatjuk

## 4. Alakrepresentáció, határ- és régió-alapú alakleíró jellemzők, Fourier leírás

---

### Alakrepresentáció

---

Az alak/forma megítélésének fontos szerep jut a látásunkban.

Az alak (shape) nem bír egzakt matematikai definícióval

A **szegmentálást** követően az objektumok kontúrjaiból vagy foltjaiból (attól függően, hogy **határ-** vagy **régió-alapú** szegmentálást vetettünk-e be) számos **alakleíró jellemzőt** vonhatunk ki.

Hangsúlyozandó, hogy itt már elszakadhatunk a digitális képektől, **némelyik jellemző csak egy szám**, mások pedig **összetett struktúrák is lehetnek**.

Az alakleíró jellemzőket három osztályba soroljuk. (**Határ, Régió, Transzformációs**)

### Határ alapú alakleíró jellemzők

---

- lánckód, alakleíró szám
- kerület, terület, kompaktság, cirkularitás
- közelítés poligonnal
- parametrikus kontúr, határvonal leíró függvény
- meredekségi hisztogram
- görbület, energia
- strukturális leírás

## Lánckód

- Az alakzat határpontjait követi/láncolja az óramutató járásával ellentétes irányban.
- **Határpont:** Az alakzat olyan pontja, melynek van az alakzathoz nem tartozó 8- ill, 4-szomszédja.
- **Különböző kezdőpontból más lánckód jöhet ki!**

## Pozitívumok:

- Invariáns a forgatásra, ha a szög  $k * \pi/2$ , eltolásra
- Gyors algoritmus, eltolás-invariáns
- kompakt

## Hátrányok:

- Zaj érzékeny,
- nem skála-invariáns
- pontosság legfeljebb pixelnyi lehet

**Különbségkód:** a lánckód első deriváltja, a szomszédok elemek közötti elmozdulások száma

**Normalizálás:** Addig permutáljuk a különbségkódot, amíg a legkisebb értékű kódot kapjuk.

**Alakleíró szám:** A normalizált különbségkód (NEM FÜGG A KEZDŐPONT VÁLASZTÁSTÓL)

## Kerület, terület számítása

- A kerület és a terület két gyakran bevetett alakleíró jellemző. Mindkettő származtatható a lánckódból is.
- **8-as lánckód esetén:**




- terület =  $\text{gyök}(2) * (\text{páratlan elemek száma}) + \text{páros elemek száma}$  a lánckódban


- **4-es lánckód esetén:**

- terület = lánckód rendje (hossza)

### poligon területe 8-as lánckód esetén:

- számontartunk egy  $y$ -t, ami kezdetben 0. Ehhez ha a lánckódban lévő következő szám "felfele" mutat hozzáadunk 1-et, ha "lefele", akkor kivonunk 1-et
  - a területváltozást szintén a lánckódban következő szám iránya határozza meg ( $y$  alapján), ahogy az alábbi képen is látszik
  - a területet úgy kapjuk, hogy folyton összeadogatjuk a területváltozásokat, és a végén vesszük az abszolútértékét

 alt text

 alt text

### Kompaktság és cirkularitás, görbület

- kompaktság =  $(\text{kerület})^2 / \text{terület}$
- cirkularitás =  $\text{terület} / (\text{kerület})^2$
- **görbület:** a határhoz rajzolt érintőkör sugarának reciproka

### Parametrikus kontúr

- A parametrikus kontúr két egyváltozós függvénnyel reprezentálja a szegmenst. A kontúron végighaladva követjük az  $x$  és az  $y$  koordináták változásait.

### Leírás egyváltozós függvényekkel (Szignatúra)

Pl. A súlypontnak a határtól vett távolságát a szög függvényében fejezi ki.

Nagyban függ az alakzat méretétől és a határon vett kezdőponttól. → normalizálásra szorul.

- Csillag-szerű objektum:
  - Van olyan pontja, amelyből induló tetszőleges irányú sugár a határt egyetlen pontban metszi.

# Régió alapú alakleíró jellemzők

---

A határ-alapúakhoz hasonlóan, számos régió-alapú alakleíró jellemzőt javasoltak.

- befoglaló téglalap, rektangularitás
- főtengely, melléktengely, átmérő, excentricitás, főtengely szöge
- konvex burok, konvex kiegészítés, konkávitási fa, particionált határ,
- vetületek, törés-költség
- topológiai leírások, Euler-szám, szomszédsági fa,
- váz,
- momentumok, invariáns momentumok

## Befoglaló téglalap, rektangularitás

- **álló befoglaló téglalap:** az objektum koordinátáinak minimumai és maximumai megadják az álló befoglaló téglalap csúcsait.
- minimális befoglaló téglalap
- **rektangularitás:** Azt mondja meg, hogy az objektum „bedobozolásakor” mennyi a tárgy és a „levegő” által elfoglalt területek aránya, tehát  $\rightarrow$  **alakzat területe / minimális befoglaló téglalap**

## Főtengely, melléktengely, átmérő, excentricitás, főtengely szöge

- **főtengely:** az alakzaton belül haladó leghosszabb egyenes szakasz
- **melléktengely:** az alakzaton belüli, a főtengelyre merőleges leghosszabb egyenes szakasz
- **átmérő:** a határ két legtávolabbi pontját köti össze. A főtengely hossza általában nem egyezik meg az átmérővel (csak a konvexeknél)
- **excentricitás:** a fő- és melléktengely hosszaránya:  $\frac{d1}{d2}$
- **főtengely szöge:** a főtengely és az x-tengely által bezárt szög

## Konvex burok, konvex kiegészítés, konkávitási fa, particionált határ

- **konvex burok:** az alakzatot tartalmazó minimális konvex alakzat
- **konvex kiegészítés:** a konvex burok és az alakzat különbsége
- **konkávitási fa:** A fa gyökere a kiindulási alakzat, az első szinten a konvex különbség alakzatai helyezkednek el, melyekre a faépítést rekurzív módon folytatjuk.
- **partícionált határ:** A konvex burok határát osztja fel részekre.

## Vetületek, törés-költség

- **vetületek:** A bináris képekből képzett nem-negatív egészekből álló (1D) tömbök.
- **törés-költség:** A vetületek továbbbragozása, kiszűri a zajos képek oszlopaiban lévő „magányos” objektumpontokat.

## Topológiai leírások, Euler-szám, szomszédsági fa,

- **topológiai leírások**
  - *bináris kép:* kétféle érték lehet benne, az 1-es az alakzatot (komponenst) reprezentálja feketével, míg a 0-s a háttérrel (lyukakat) fehérrel
  - *komponens:* maximálisan összefüggő fekete halmaz
  - *üreg:* a negált kép egy véges komponense
- **Euler-féle szám:** egyetlen egész szám  $\rightarrow$  *komponensek száma - üregek száma*.
  - Rengeteg képre lehet az ugyanaz. Valamit elárul a képről, de önmagában keveset.
- **összefüggőségi-fa:** A bináris képekhez rendelt irányított gráf **@kép**  
(**Osszefuggosegi\_fa.JPG**)
  - minden egyes csúcs megfelel a kép egy (fehér vagy fekete) komponensének,
  - a gráf tartalmazza az (X,Y) éleket, ha az X komponens „körülveszi” a vele szomszédos Y komponenst

## Váz

A váz egy gyakran alkalmazott régió-alapú alakleíró jellemző, mely leírja az objektumok általános formáját.

Alapvetően 3-féleképp határozhatjuk meg:

1. a vázat az objektum azon pontjai alkotják, melyekre kettő vagy több legközelebbi határpont található.

2. Az objektum határát (minden pontjában) egyidejűleg felgyújtjuk. A váz azokból a pontokból áll, ahol a tűzfrontok találkoznak és kioltják egymást. (Feltételezzük, hogy a tűzfrontok minden irányban egyenletes sebességgel, vagyis izotropikusan terjednek.)
3. A vázat az objektumba beírható maximális (nyílt) hipergömbök középpontjai alkotják. Egy beírható hipergömb maximális, ha őt nem tartalmazza egyetlen másik beírható hipergömb sem.

**Invariáns az eltolásra, elforgatásra és az uniform skálázásra.**

## Momentumok, invariáns momentumok

**Pro:**

Egy szám

- többszintű képekre is értelmezettek, invariánsak a főbb geometriai műveletekre
  - rotálás, eltolás, skálázás, tükrözés stb...

Bizonyos (centrális) momentumoknak geometriai jelentés is tulajdonítható, illetve fontos jellemzők kifejezhetők a segítségükkel, például súlypont.

Javasoltak viszont 7 ún. invariáns momentumot is (ld. 56. dia), amelyekhez nem társíthatók különösebb jelentések, de a belőlük alkotott rendezett hetesek (vagy akár hármasok, ha nem vesszük mindet figyelembe) jól jellemzik az objektumokat.

## Transzformációs megközelítés

---

### Fourier transzformáció

**Ez egy transzformáción alapuló alakleírás**

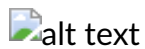
**Transzformáljuk** (szigorúan 1D Fourier transzformációt alkalmazunk) **a határ  $K$  darab mintavételezett pontjából képzett  $s$  vektort. Az eredményül kapott  $a$  vektor adja a Fourier leírást.** (vagyis tartalmazza a Fourier együtthatókat, a transzformáció bázisfüggvényeinek súlyait)

Az alakzat rekonstrukciójához az inverz Fourier-transzformációt kell végrehajtani.

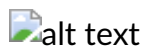
A  $K$  darab Fourier együtthatóból visszakaphatnánk torzítatlanul az eredeti mintavételezett pontokat, az alakleíráshoz viszont nem az összes súlyt, hanem csak egy részüket tartjuk meg, mindössze  $P < K$  darab együttható alapján térünk vissza a képtérbe

- ekkor a képtérben ismét  $K$  darab pontot kapunk vissza, de nem a kiindulás mintavételezettjeit.
- Az együtthatók egy részének eldobásával kapott leírás (a meghagyott együtthatók adják a jellemzést) voltaképpen egy veszteséges tömörítés: kevesebb adattal tudjuk jól-rosszul közelíteni a kiindulást.

Az alábbi kép azt mutatja, hogy a 64 kontúrponttal mintavételezett négyzetre csak sok együttható megtartásával tudunk négyzetfélét rekonstruálni.



A következő képen viszont tesztobjektum határa közel 3000 ponttal adott, és már 36 együttható is visszaadhat 3000 pontot úgy, hogy azok jól közelítik a kiindulási kontúrt.



## 5. Algoritmusok vezérlési szerkezetei és megvalósításuk C programozási nyelven. A szekvenciális, iterációs, elágazásos, és az eljárás vezérlés

### Algoritmusok vezérlési szerkezetei és megvalósításuk C nyelven

**Algoritmus:** bármilyen jól definiált számítási eljárást, amely bemenetként bizonyos értéket vagy értékeket kap és kimenetként bizonyos értéket vagy értékeket állít elő. Vizsgálhatjuk helyesség, idő- és tárigény szempontjából

**Algoritmus vezérlése:** Az az előírás, amely az algoritmus minden lépésére (részműveletére) kijelöli, hogy a lépés végrehajtása után melyik lépés végrehajtásával

folytatódjon (esetleg fejeződjék be) az algoritmus végrehajtása. Az algoritmusnak, mint műveletnek a vezérlés a legfontosabb komponense.

Négy fő vezérlési módot különböztetünk meg:

- **Szekvenciális:** Véges sok adott művelet rögzített sorrendben egymás után történő végrehajtása. (sorban egymás után)
- **Szelekciós:** Véges sok rögzített művelet közül adott feltétel alapján valamelyik végrehajtása. (if, else, if else, switch)
- **Ismétléses:** Adott művelet adott feltétel szerinti ismételt végrehajtása. (for, while, do while)
- **Eljárás:** Adott művelet alkalmazása adott argumentumokra, ami az argumentumok értékének pontosan meghatározott változását eredményezi. (void func, int func, double func, ...)

A vezérlési módok nyelvek feletti fogalmak.

A imperatív (algoritmikus) programozási nyelvekben ezek a vezérlési szerkezetek (közvetlenül vagy közvetve) megvalósíthatók.

## A szekvenciális, iterációs, elágazásos, és az eljárás vezérlés

---

### Szekvenciális vezérlés

Szekvenciális vezérlésről akkor beszélünk, amikor a P probléma megoldását úgy kapjuk, hogy a problémát  $P_1, \dots, P_n$  részproblémákra bontjuk, majd az ezekre adott megoldásokat (részalgoritmusokat) sorban egymás után hajtjuk végre.

$P_1, \dots, P_n$  lehetnek elemi műveletek, vagy nem elemi részproblémák megnevezései.

### Eljárásvezérlés

Eljárásvezérlésről akkor beszélünk, amikor egy műveletet adott argumentumokra alkalmazunk, aminek hatására az argumentumok értékei pontosan meghatározott módon változnak meg.

Az eljárásvezérlés fajtái:

- Eljárasművelet
- Függvényművelet

C-ben kicsi a különbség a kettő között.

## Függvényművelet

- A matematikai függvény fogalmának általánosítása
- Ha egy részprobléma célja egy érték kiszámítása adott értékek függvényében, akkor a megoldást megfogalmazhatjuk függvényművelettel.
- A függvényművelet specifikációja tartalmazza:
  - A művelet elnevezését
  - A paraméterek felsorolását
  - Mindegyik paraméter adattípusát
  - A művelet hatásának leírását
  - A függvényművelet eredménytípusát
- **Minden függvényben szerepelnie kell legalább egy return utasításnak**
- Ha a függvényben egy ilyen utasítást hajtunk végre, akkor a függvény értékének kiszámítása befejeződik. A hívás helyén a függvény a return által kiszámított értéket veszi fel

## Eljárasművelet

- Ha eljárást szeretnénk készíteni C nyelven, akkor egy olyan függvényt kell deklarálni, melynek eredménytípusa **void**. Ebben az esetben a függvény definíciójában nem kötelező a return utasítás, illetve ha mégis van ilyen, akkor nem adható meg utána kifejezés
- **Megvalósítás:**
  - csak bemenő módú argumentumok vannak
  - pointerekkel lehet kezelni kimenő argumentumokként is

## Szelekciós vezérlés

Szelekciós vezérlésről akkor beszélünk, amikor véges sok rögzített művelet közül véges sok feltétel alapján választjuk ki, hogy melyik művelet kerüljön végrehajtásra.

Típusai:

- Egyszerű szelekciós vezérlés
- Többszörös szelekciós vezérlés

- Esetkiválasztásos szelekciós vezérlés
- A fenti három „egyébként” ággal

### **Egyszerű szelekciós vezérlés**

- Egyszerű szelekció esetén egyetlen feltétel és egyetlen művelet van (ami persze lehet összetett).
- A vezérlés bővíthető úgy, hogy a 3. pontban üres művelet helyett egy B műveletet hajtunk végre, ekkor beszélünk egyébként ágról.

Egyszerű szelekciós utasítás megvalósítása C nyelven:

```
if(F) {
    A;
}
```

### **Többszörös szelekciós vezérlés**

- Ha több feltételünk és több műveletünk van, akkor többszörös szelekcióról beszélünk.
- A többszörös szelekció is bővíthető egyébként ággal úgy, hogy egy nemüres B műveletet hajtunk végre a 3. lépésben.
- Legyenek  $F_i$  logikai kifejezések,  $A_i$  (és B) pedig tetszőleges műveletek. Az  $F_i$  feltételekből és  $A_i$  (és B) műveletekből képzett többszörös szelekciós vezérlés a következő vezérlési előírást jelenti:
  - Az  $F_i$  feltételek sorban történő kiértékelésével adjunk választ a következő kérdésre: Van-e olyan  $i$  amelyre teljesül, hogy az  $F_i$  feltétel igaz és az összes  $F_j$  feltétel hamis?
  - Ha van ilyen  $i$ , akkor hajtunk végre az  $A_i$  műveletet és fejezzük be az összetett művelet végrehajtását.
  - Egyébként, vagyis ha minden  $F_i$  feltétel hamis, akkor (hajtunk végre B-t és) fejezzük be az összetett művelet végrehajtását.

Többszörös szelekciós utasítás megvalósítása C nyelven:

```
if(F1) {
    A1;
} else if (F2) {
    A2;
}...
```



- C nyelvben nincs külön utasítás a többszörös szelekció megvalósítására, ezért az egyszerű szelekció ismételt alkalmazásával kell azt megvalósítani.
- Ez azon az összefüggésen alapszik, hogy a többszörös szelekció levezethető egyszerű szelekciók megfelelő összetételével.

### Esetkiválasztós szelekciós vezérlés

Ha a többszörös szelekciós vezérlésben minden  $F_i$  feltételünk  $K \in H_i$  alakú, akkor esetkiválasztásos szelekcióról beszélünk.

- Legyen  $K$  egy adott típusú kifejezés, legyenek  $H_i$  ilyen típusú halmazok,  $A_i$  (és  $B$ ) pedig tetszőleges műveletek. A  $K$  szelektor kifejezésből,  $H_i$  kiválasztó halmazokból és  $A_i$  (és  $B$ ) műveletekből képzett esetkiválasztásos szelekciós vezérlés a következő vezérlési előírást jelenti:
  - Értékeljük ki a  $K$  kifejezést és folytassuk a 2.) lépéssel.
  - Adjunk választ a következő kérdésre: Van-e olyan  $i$  ( $1 \leq i \leq n$ ), amelyre teljesül, hogy a kiszámolt érték eleme a  $H_i$  halmaznak és nem eleme az összes  $H_j$  ( $1 \leq j < i$ ) halmaznak?
  - Ha van ilyen  $i$ , akkor hajtsuk végre az  $A_i$  műveletet és fejezzük be az összetett művelet végrehajtását.
  - Egyébként, vagyis ha  $K$  nem eleme egyetlen  $H_i$  halmaznak sem, akkor (hajtsuk végre  $B$ -t és) fejezzük be az összetett művelet végrehajtását.
- A kiválasztó halmazok megadása az esetkiválasztásos szelekció kritikus pontja.
- Algoritmusok tervezése során bármilyen effektív halmazmegadást használhatunk, azonban a tényleges megvalósításkor csak a választott programozási nyelv eszközeit alkalmazhatjuk.

A switch utasítás: Ha egy kifejezés értéke alapján többféle utasítás közül kell választanunk, a switch utasítást használhatjuk. Megadhatjuk, hogy hol kezdődjön és meddig tartson az utasítás-sorozat végrehajtása.

A switch utasítás szintaxisa C-ben:

```
switch(kifejezés) {
    case konstans1:
        A;
        break;
    case konstans2:
        B;
```

```
        break;
    default:
        D;
}
```

- A szelektor kifejezés és a konstansok típusának meg kell egyeznie. Egy konstans legfeljebb egy case mögött és a default kulcsszó is legfeljebb egyszer szerepelhet egy switch utasításban.
- A default címke olyan, mintha a szelektor kifejezés lehetséges értékei közül minden olyat felsorolnánk, ami nem szerepel case mögött az adott switch-ben.
- A címkék (beleértve a default-ot is) sorrendje tetszőleges lehet, az nem befolyásolja, hogy a szelektor kifejezés melyik címkét választja.
- A szelektor kifejezés értékétől csak az függ, hogy melyik helyen kezdjük el végrehajtani a switch magját. Ha a végrehajtás elkezdődik, akkor onnantól kezdve az első break (vagy return) utasításig, vagy a switch végéig sorban hajtódnak végre az utasítások. Ebben a fázisban a további case és default címkéknek már nincs jelentősége.
- A Hi halmazok elemszáma tetszőleges lehet, viszont a case-ek után csak egy-egy érték állhat.

## Ismétléses vezérlések

Ismétléses vezérlésen olyan vezérlési előírást értünk, amely adott műveletnek adott feltétel szerinti ismételt végrehajtását írja elő.

Az algoritmustervezés során a leginkább megfelelő ismétléses vezérlési formát használjuk, függetlenül attól, hogy a megvalósításra használt programozási nyelvben közvetlenül megvalósítható-e ez a vezérlési mód.

Ismétléses vezérlés képzését ciklusszervezésnek is nevezik, így az ismétlésben szereplő műveletet ciklusmagnak hívjuk.

Az ismétlési feltétel szerint ötféle ismétléses vezérlést különböztetünk meg:

- Kezdőfeltételes
- Végfeltételes
- Számlálósos
- Hurok
- Diszkrét

## Kezdőfeltételes ismétléses vezérlés

Kezdőfeltételes vezérlésről akkor beszélünk, ha a ciklusmag (ismételt) végrehajtását egy belépési (ismétlési) feltételhez kötjük.

- Legyen F logikai kifejezés, M pedig tetszőleges művelet. Az F ismétlési feltételből és az M műveletből (a ciklusmagból) képzett kezdőfeltételes ismétléses vezérlés a következő vezérlési előírást jelenti:
  - Értékeljük ki az F feltételt és folytassuk a 2.) lépéssel.
  - Ha F értéke hamis, akkor az ismétlés és ezzel együtt az összetett művelet végrehajtása befejeződött.
  - Egyébként, vagyis ha az F értéke igaz, akkor hajtsuk végre az M műveletet, majd folytassuk az 1.) lépéssel.
- A feltétel ellenőrzése a művelet előtt történik
- Ha az F értéke kezdetben hamis, az összetett művelet végrehajtása befejeződik anélkül, hogy az M művelet egyszer is végrehajtásra kerülne
- Ha az F értéke igaz, és az M művelet nincs hatással az F feltételre, akkor F igaz is marad, tehát az összetett művelet végrehajtása nem tud befejeződni. Ilyenkor végtelen ciklus végrehajtását írtuk elő.
- Fontos tehát, hogy az M művelet hatással legyen az F feltételre.

A while utasítás: Ha valamilyen műveletet mindaddig végre kell hajtani, amíg egy feltétel igaz, a while utasítás használható.

```
while(F) {  
    M;  
}
```

## Végfeltételes ismétléses vezérlés

A végfeltételes ismétléses vezérlés alapvetően abban különbözik a kezdőfeltételes ismétléses vezérléstől, hogy a ciklusmag legalább egyszer végrehajtódik.

Végfeltételes vezérlésről akkor beszélünk, ha a ciklusmag elhagyását egy kilépési feltételhez kötjük.

- Legyen F logikai kifejezés, M pedig tetszőleges művelet. Az F kilépési feltételből és az M műveletből (a ciklusmagból) képzett végfeltételes ismétléses vezérlés a következő vezérlési előírást jelenti:
  - Hajtsuk végre az M műveletet majd folytassuk a 2.) lépéssel.
  - Értékeljük ki az F feltételt és folytassuk a 3.) lépéssel.
  - Ha F értéke igaz, akkor az ismétléses vezérlés és ezzel együtt az összetett művelet végrehajtása befejeződött.
  - Egyébként, vagyis ha az F értéke hamis, akkor folytassuk az 1.) lépéssel.
- Ha az F értéke kezdetben hamis, és az M művelet nincs hatással F-re, akkor végtelen ciklust kapunk. Ha az F értéke kezdetben igaz, M legalább egyszer akkor is végrehajtásra kerül.
- A kezdő és végfeltételes vezérlések kifejezhetőek egymás segítségével.

A do while: utasítás Ha valamilyen műveletet mindaddig végre kell hajtani, amíg egy feltétel igaz, a do while utasítás használható. A művelet végrehajtása szükséges a feltétel kiértékeléséhez. A feltétel ellenőrzése a művelet után történik, így ha a feltétel kezdetben hamis volt, a műveletet akkor is legalább egyszer végrehajtjuk.

```
do {
    M;
} while (!F);
```

### **Számlálós ismétléses vezérlések**

Számlálós ismétléses vezérlésről akkor beszélünk, ha a ciklusmagot végre kell hajtani sorban minden olyan értékére (növekvő vagy csökkenő sorrendben), amely egy adott intervallumba esik.

Legyen a és b egész érték, i egész típusú változó, M pedig tetszőleges művelet, amelynek nincs hatása a, b és i értékére.

Növekvő számlálós ismétléses vezérlések:

- Az a és b határértékekből, i ciklusváltozóból és M műveletből (ciklusmagból) képzett növekvő számlálós ismétléses vezérlés az alábbi vezérlési előírást jelenti:
  - Legyen  $i = a$  és folytassuk a 2.) lépéssel.
  - Ha  $b < i$  (i nagyobb mint a intervallum végpontja), akkor az ismétlés és ezzel együtt az összetett művelet végrehajtása befejeződött.
  - Egyébként, vagyis ha  $i \leq b$ , akkor hajtsuk végre az M műveletet, majd folytassuk a 4.) lépéssel.

- Növeljük  $i$  értékét 1-gyel, és folytassuk a 2.) lépéssel.

Csökkenő számlálós ismétléses vezérlések:

- Az  $a$  és  $b$  határértékekből,  $i$  ciklusváltozóból és  $M$  műveletből (ciklusmagból) képzett csökkenő számlálós ismétléses vezérlés az alábbi vezérlési előírást jelenti:
  - Legyen  $i = b$  és folytassuk a 2.) lépéssel.
  - Ha  $i < a$ , akkor az ismétlés és ezzel együtt az összetett művelet végrehajtása befejeződött.
  - Egyébként, vagyis ha  $a \leq i$ , akkor hajtsuk végre az  $M$  műveletet, majd folytassuk a 4.) lépéssel.
  - Csökkentsük  $i$  értékét 1-gyel, és folytassuk a 2.) lépéssel.

A for utasítás: Ha valamilyen műveletet sorban több értékére is végre kell hajtani, akkor a for utasítás használható.

```
for (i = a; i <= b; i++) {
    M;
}
for (kif1; kif2; kif3) {
    M;
}
```

C-ben a for utasítás általános alakja:

- A kif1 és kif3 többnyire értékadás vagy függvényhívás, kif2 pedig relációs kifejezés.
- Bármelyik kifejezés elhagyható, de a pontosvesszőknek meg kell maradniuk
- kif2 elhagyása esetén a feltételt konstans igaznak tekintjük, ekkor a break vagy return segítségével lehet kiugrani a ciklusból.

## Hurok ismétléses vezérlés

Amikor a ciklusmag ismétlését a ciklusmagon belül vezéreljük úgy, hogy a ciklus különböző pontjain adott feltételek teljesülése esetén a ciklus végrehajtását befejezzük, hurok ismétléses vezérlésről beszélünk.

- Legyenek  $F_i$  logikai kifejezések,  $K_i$  és  $M_j$  pedig tetszőleges (akár üres) műveletek  $1 \leq i \leq n$  és  $0 \leq j \leq n$  értékekre. Az  $F_i$  kijáratí feltételekből,  $K_i$  kijáratí műveletekből és az  $M_i$  műveletekből képzett hurok ismétléses vezérlés a következő előírást jelenti:

- Az ismétléses vezérlés következő végrehajtandó egysége az  $M_0$  művelet.
- Ha a végrehajtandó egység az  $M_j$  művelet, akkor ez végrehajtódik.  $j = n$  esetén folytassuk az 1.) lépéssel, különben pedig az  $F_{j+1}$  feltétel végrehajtásával a 3.) lépésben.
- Ha a végrehajtandó egység az  $F_i$  feltétel ( $1 \leq i \leq n$ ), akkor értékeljük ki. Ha  $F_i$  igaz volt, akkor hajtsuk végre a  $K_i$  műveletet, és fejezzük be a vezérlést. Különben a végrehajtás az  $M_i$  művelettel folytatódik a 2.) lépésben.
- A kezdő- és végfeltételes ismétléses vezérlések speciális esetei a hurok ismétléses vezérlésnek.
- A C nyelvben a ciklusmag folyamatos végrehajtásának megszakítására két utasítás használható:
- break: Megszakítja a ciklust, a program végrehajtása a ciklusmag utáni első utasítással folytatódik. Használható a switch utasításban is, hatására a program végrehajtása a switch utáni első utasítással folytatódik.
- continue: Megszakítja a ciklusmag aktuális lefutását, a vezérlés a ciklus feltételének kiértékelésével (while, do while) illetve az inkrementáló kifejezés kiértékelésével (for) folytatódik.

### **Diszkrét ismétléses vezérlés:**

Diszkrét ismétléses vezérlésről akkor beszélünk, ha a ciklusmagot végre kell hajtani egy halmaz minden elemére tetszőleges sorrendben.

- Legyen  $x$  egy  $T$  típusú változó,  $H$  a  $T$  értékkészletének részhalmaza,  $M$  pedig tetszőleges művelet, amelynek nincs hatása  $x$  és  $H$  értékére. A  $H$  halmazból,  $x$  ciklusváltozóból és  $M$  műveletből (ciklusmagból) képzett diszkrét ismétléses vezérlés az alábbi vezérlési előírást jelenti:
  - Ha a  $H$  halmaz minden elemére végrehajtottuk az  $M$  műveletet, akkor vége a vezérlésnek.
  - Egyébként vegyünk a  $H$  halmaz egy olyan tetszőleges  $e$  elemét, amelyre még nem hajtottuk végre az  $M$  műveletet, és folytassuk a 3.) lépéssel.
  - Legyen  $x = e$  és hajtsuk végre az  $M$  műveletet, majd folytassuk az 1.) lépéssel.
- A  $H$  halmaz számossága határozza meg, hogy az  $M$  művelet hányszor hajtódik végre. Ha a  $H$  az üres halmaz, akkor a diszkrét ismétléses vezérlés az  $M$  művelet végrehajtása nélkül befejeződik.
- A diszkrét ismétléses vezérlésnek nincs közvetlen megvalósítása a C nyelvben.

## 6. Egyszerű adattípusok: egész, valós, logikai és karakter típusok és kifejezések. Az egyszerű típusok reprezentációja, számábrázolási tartományuk, pontosságuk, memória igényük és műveleteik. Az összetett adattípusok és a típusképzések, valamint megvalósításuk C nyelven. A pointer, a tömb, a rekord és az unió típus. Az egyes típusok szerepe, használata

---

**Egyszerű adattípusok: egész, valós, logikai és karakter típusok és kifejezések. Az egyszerű típusok reprezentációja, számábrázolási tartományuk, pontosságuk, memória igényük és műveleteik**

---

Az **adattípus** (gyakran röviden **típus**) az értékek egy halmazához rendelt név vagy címke és ezen halmaz értékein végrehajtható néhány művelet

Az elemi adattípusok értékeit nem lehet önmagukban értelmes részekre bontani.

Ha a nyelv szintaktikája szerint a program egy adott pontján típusnak kellene következnie de az hiányzik, a fordító a típus helyére automatikusan int-et helyettesít.

### **Egész típusok**

A C nyelvben az egész típus az int.

Az **int** típus értékkészlete az alábbi kulcsszavakkal módosítható:

- **signed** (1 byte): A típus előjeles értékeket fog tartalmazni (int, char).

- **unsigned** (1 byte): A típus csak előjeltelen, nemnegatív értékeket fog tartalmazni (int, char).
- **short** (2 byte): Rövidebb helyen tárolódik, így kisebb lesz az értékkészlet (int).
- **long** (4 byte): Hosszabb helyen tárolódik, így bővebb lesz az értékkészlet (int).  
Duplán is alkalmazható (**long long, ami 8 byte**).

Az egész típusok az értékkészlet határain belüli minden egész értéket pontosan ábrázolnak.

Az egyes gépeken az egyes típusok mérete más-más lehet, de minden C megvalósításban teljesülnie kell a  $\text{sizeof}(\text{short}) \leq \text{sizeof}(\text{int}) \leq \text{sizeof}(\text{long}) \leq \text{sizeof}(\text{long long})$  relációnak.

A C nyelv különféle egész adattípusai az értékhasználatukban különböznek egymástól, az értelmezett műveletükben megegyeznek

Az egész adattípusokon általában az 5 matematikai alpműveletet és az értékadás műveletét értelmezzük, de C nyelven ennél jóval többet.

Értékadó művelet jobb oldalán álló kifejezés kiértékelése független attól, hogy a bal oldalon milyen típusú változó van.

**A / művelet két egész értékre alkalmazva maradékos osztást jelent!**

#### **Tárolás:**

n bites tárterületnek  $2^n$  állapota van, vagyis egy n biten tárolt adattípusnak legfeljebb ennyi különböző értéke lehet.

**Egész típusoknál a kettes komplement** szokás használni, ha negatív értékek is szerepelhetnek az értékhasználatban.

#### **Kettes komplement:**

- van egy pozitív számunk, és annak keressük a negatív párját
- a számot kettes számrendszerben felírjuk
- invertáljuk az összes bitet
- majd hozzáadunk a végén egyet
- a kapott szám lesz a szám ellentettje

#### **Értékhasználat mérete:**

Ha negatív számok nem szerepelnek az értékhasználatban, akkor az értékhasználat a  $[0 \dots 2^n - 1]$  zárt intervallum.



Ha az értékhalmban negatív számok is szerepelnek, akkor az értékhalmban a  $[-2^{(n-1)} \dots 2^{(n-1)} - 1]$  zárt intervallum.

### Műveletei:

- bitenkénti
  - negáció
  - és
  - vagy
  - kizáró vagy
  - balra léptetés
  - jobbra léptetés

### Karakter típus

A char adattípus a C nyelv eleve definiált elemi adattípusa, értékkészlete 256 elemet tartalmaz.

A char adattípus egészként is használható, de alapvetően karakterek (betűk, számjegyek, írásjelek) tárolására való.

- Hogy melyik értékhez melyik karakter tartozik, az az alkalmazott kódtáblázattól függ.
- Bizonyos karakterek (általában a rendezés szerint első néhány) vezérlő karakternek számítanak, és nem megjeleníthetők.

### Egy C programban karakter értékeket megadhatunk:

- karakterkóddal számértékként, vagy
- aposztrófok közé írt karakterrel

A speciális karaktereket, illetve magát az aposztrófot (és végső soron tetszőleges karaktert is) escape-szekvenciákkal lehet megadni.

Az escape-szekvenciákat a \ (backslash) karakterrel kell kezdeni.

Konvertáljunk egy tetszőleges számjegy karaktert (ch) a neki megfelelő egész számmá és egy egyjegyű egészet (i) karakterré:

```
i = ch - '0';  
ch = i + '0';
```

## Valós típusok

A C nyelvben a valós adattípusok a **float** és **double**.

A **double** adattípus az alábbi kulcsszóval módosítható:

- **long**: Implementációfüggő módon 64, 80, 96 vagy 128 bites pontosságot megvalósító adattípus

A valós adattípusok az értékkészlet határain belül sem képesek minden valós értéket pontosan ábrázolni. Viszont az értékkészlet határain belüli minden a valós értéket képesek egy típusfüggő e relatív pontossággal ábrázolni, az a-hoz legközelebbi a típus által pontosan ábrázolható x valós értékkel.

- A C nyelv különféle valós adattípusai az értékalmazukban különböznek egymástól, az értelmezett műveletükben megegyeznek.
- Valós kifejezésben bármely valós vagy egész típusú tényező (akár vegyesen többféle is) szerepelhet.
- Valós konstans típusa double, vagy a szátleírásban megadott típus (f, l suffix).
- Értékadó művelet jobb oldalán álló kifejezés kiértékelése független attól, hogy a bal oldalon milyen típusú változó van.
- A típus pontatlansága miatt az == műveletet nagyon körültekintően kell használni!

### Ábrázolása:

Egy valós értéket tároló memóriaterület **három részre osztható**: az **előjelbitet**, a **törtet** és az **exponenciális kitevőt** kódoló részre.

- Az **előjelbit** 0 értéke a pozitív, 1 értéke a negatív számokat jelöli
- A számot kettes számrendszerben  $1.m \times 2^k$  alakra hozzuk, majd az  $m$  **sámjegyeit eltároljuk a törtnek**, a  $k$ -nak **egy típusfüggő  $b$  konstanssal növelt értékét pedig a kitevőnek fenntartott részen**.
- Így a **tört rész hossza az ábrázolás pontosságát** (az értékes számjegyek számát), a **kitevő pedig az értéktartomány méretét** határozza meg.
- Nagyon kicsi számokat speciálisan  $0.m \times 2^{(1-b)}$  alakban tárolhatunk, ekkor a kitevő összes bitje 0.
- Ha a kitevő összes bitje 1, az csupa 0 bitből álló tört esetén a  $\infty$ , minden más esetben NaN értéket jelent.
- A 32/64 bites float/double az 1 előjelbit mögött 8/11 biten a kitevő  $b = 127$ -tel/1023-mal növelt értékét, majd 23/52 biten a törtet tárolja.

## Logikai típus

A C nyelvnek csak a C99 szabvány óta része a logikai (\_Bool) típus (melynek értékkészlete a {0, 1} halmaz), de azért logikai értékek persze előtte is keletkeztek.

A műveletek eredményeként keletkező logikai hamis értéket a 0 egész érték reprezentálja, és a 0 egész érték logikai értéként értelmezve hamisat jelent.

A műveletek eredményeként keletkező logikai igaz értéket az 1 egész érték reprezentálja, de bármely 0-tól különböző egész érték logikai értéként értelmezve igazat jelent.

stdbool.h-ban definiált a bool típus és a true, false konstansok

Konstansként is definiálhatjuk, pl

```
#define TRUE 1
#define FALSE 0
```

## **Az összetett adattípusok és a típusképzések, valamint megvalósításuk C nyelven. A pointer, a tömb, a rekord és az unió típus. Az egyes típusok szerepe, használata**

---

### **Összetett adattípusok, típusképzések**

Az összetett adattípusok értékei tovább bonthatóak, további értelmezésük lehetséges.

A C nyelv összetett adattípusai:

- **Pointer típus**
  - Függvény típus
- **Tömb típus**
  - Sztringek
- **Rekord típus**
  - Szorzat-rekord
  - Egyesítési-rekord

### **Pointer típus**

Az eddigi tárgyalásunkban szerepelt változók statikusak abban az értelemben, hogy létezésük annak a blokknak a végrehajtásához kötött, amelyben a változó deklarálva lett. A programozónak a deklaráció helyén túl nincs befolyása a változó létesítésére és megszüntetésére.

Az olyan változókat, amelyek a blokkok aktivizálásától függetlenül létesíthetők és megszüntethetők, dinamikus változóknak nevezzük.

Dinamikus változók megvalósításának általános eszköze a pointer típus.

Egy pointer típusú változó értéke (első megközelítésben) egy meghatározott típusú dinamikus változó.

Pointer típusú változót a \* segítségével deklarálhatunk:

```
típus * változónév;
```

Az eddigiek során lényegében azonosítottuk a változóhivatkozást és a hivatkozott változót.

A dinamikus változók megértéséhez viszont világosan különbséget kell tennünk az alábbi három fogalom között:

- Változóhivatkozás
- Hivatkozott változó
- Változó értéke

A változóhivatkozás szintaktikus egység, meghatározott formai szabályok szerint képzett jelsorozat egy adott programnyelven, tehát egy kódrészlet.

A változó a program futása során a program által lefoglalt memóriaterület egy része, amelyen egy adott (elemi vagy összetett) típusú érték tárolódik.

Különböző változóhivatkozások hivatkozhatnak ugyanarra a változóra, illetve ugyanaz a változóhivatkozás a végrehajtás különböző időpontjaiban különböző változókra hivatkozhat.

Egy változóhivatkozáshoz nem biztos, hogy egy adott időben tartozik hivatkozott változó.

Műveletek:

- **NULL**
  - NULL, nem tartozik hozzá dinamikus változó
- **Létesít**
  - `x = malloc(sizeof(E))`
- **Értékadás**
  - `x = y`
- **Törlés**
  - `free(x)`
- **Dereferencia:** A pointer által mutatott dinamikus változó elérése, eredménye egy változóhivatkozás.
  - `*x`
- **Egyenlő**
  - `p == q`
- **NemEgyenlő**
  - `p != q`

**A memóriaműveletekhez szükség van az `stdlib.h` vagy a `memory.h` használatára.**

**`malloc(S)`**, lefoglal egy S méretű memóriaterületet

**`sizeof(E)`**, megmondja, hogy egy E típusú érték mekkora helyet igényel a memóriában

**`malloc(sizeof(E))`**, létrehoz egy E típusú érték tárolására is alkalmas változó

**`free(p)`**, felszabadítja a p-hez tartozó memóriaterületet, ezután a p-hez nem lesz érvényes változóhivatkozás

Linux alatt logikailag minden programnak saját memória-tartománya van, amin belül az egyes memóriacímeket egy sorszám azonosítja.

**Pointer típusú változó** 32 bites rendszereken 4 bájt, 64 bites rendszereken 8 bájt hosszban a hozzá tartozó dinamikus változóhoz foglalt memóriamező kezdőcímét (sorszámát) tartalmazza.

A pointer értéke tehát (második megközelítésben) értelmezhető egy tetszőleges memóriacímként is, amely értelmezés egybeesik a pointer megvalósításával.

Ilyen módon viszont értelmezhetjük a címképző műveletet, ami egy változó memóriabeli pozícióját, címét adja vissza.

- **Cím**
  - `p = &x`

A void\* egy speciális, úgynevezett típustalan pointer. Az ilyen típusú pointerok „csak” memóriacímek tárolására alkalmasak, a dereferencia művelet alkalmazása rájuk értelmetlen. Viszont minden típusú pointerrel kompatibilisek értékadás és összehasonlítás tekintetében.

## Tömb típus

Algoritmusok tervezésekor gyakran előfordul, hogy adatok sorozatával kell dolgozni, vagy mert az input adatok sorozatot alkotnak, vagy mert a feladat megoldásához kell.

Tegyük fel, hogy a sorozat rögzített elemszámú ( $n$ ) és mindegyik komponensük egy megadott (elemi vagy összetett) típusból ( $E$ ) való érték.

Ekkor tehát egy olyan összetett adathalmazzal van dolgunk, amelynek egy eleme  $A = (a_0, \dots, a_{n-1})$ , ahol  $a_i \in E, \forall i \in (0, \dots, n-1)$ -re.

Ha az ilyen sorozatokon a következő műveleteket értelmezzük, akkor egy (absztrakt) adattípushoz jutunk, amit Tömb típusnak nevezünk.

Jelöljük ezt a Tömb típust  $T$ -vel, a  $0, \dots, n-1$  intervallumot pedig  $I$ -vel.

## Műveletek

- *Kiolvas*
  - a sorozat  $i$ . elemének kiolvasása egy változóba
- *Módosít*
  - a sorozat  $i$ . elemének módosítása egy  $E$  típusú értékre
- *Értékadás*
  - a változó felveszi a tömb értékét

Tömb típusú változót az alábbi módon deklarálhatunk:

```
típus változónév[elemszám];
```

Tömbelem hivatkozásra a `[]` operátort használjuk.

Ez egy olyan tömbökön értelmezett művelet C-ben, ami nagyon magas precedenciával rendelkezik és balasszociatív.

Egy tömbre a tömbindexelés operátort (megfelelő index használatával) alkalmazva a tömb adott elemét változóként kapjuk vissza.

## Rekord típus

A tömb típus nagyszámú, de ugyanazon típusú adat tárolására alkalmas.

Problémák megoldása során viszont gyakran előfordul, hogy különböző típusú, de logikailag összetartozó adatelemek együttesével kell dolgozni.

Az ilyen adatok tárolására szolgálnak a rekord típusok, ezek létrehozására pedig a rekord típusképzések.

Ha az egyes típusú adatokat egyszerre kell tudnunk tárolni, szorzat-rekordról beszélünk.

Az új adattípusra a  $T = \text{Rekord}(T_1, \dots, T_k)$  jelölést használjuk és szorzat-rekordnak vagy struktúrának nevezzük.

- kiolvas
- módosít
- értékadás

```
typedef struct T {  
    T1 M1;  
    ...  
    Tk Mk;  
} T;
```

A fenti típusképzésben az  $M_1, \dots, M_k$  azonosítókat mezőazonosítóknak (tagnak, member-nek) hívjuk és lokálisak a típusképzésre nézve.

Az absztrakt típus műveletei mezőhivatkozások segítségével valósíthatóak meg.

A mezőhivatkozásra a  $.$  operátort használjuk. Ez egy olyan rekordokon értelmezett művelet C-ben, ami nagyon magas precedenciával rendelkezik és balasszociatív.

Egy rekordra a mezőkiválasztás operátort (megfelelő mezőnévvel) alkalmazva a rekord mezőjét változóként kapjuk vissza.

## Unió típus

Ha az egyes típusú adatokat nem kell egyszerre tárolni, egyesített-rekordról beszélünk

A T halmazon is a szorzat rekordhoz hasonló módon értelmezhetünk kiolvasó és módosító műveletet.

Az új adattípust a T 0 változati típusból és T 1 , . . . , T k egyesítési-tag típusokból képzett egyesített-rekord típusnak nevezzük.

```
typedef union T {  
    T1 M1;  
    ...  
    Tk Mk;  
} T;
```

A union típusú változó számára foglalt memória mérete, amely a sizeof függvénnyel lekérdezhető:

$$\text{sizeof}(T) = \max\{\text{sizeof}(T_1), \dots, \text{sizeof}(T_k)\}$$

Valamennyi változati mező ugyanazon a memóriacímen kezdődik, ami megegyezik a teljes union típusú érték címével (azaz minden mező eltolása, offset-je 0).

## Union vs Struct

	Struct	Union
<b>Méret</b>	A tagok elemei méretének az összege	A legnagyobb elemnek a mérete
<b>Memória</b>	Minden tagnak van külön memória részlete	A memórián osztoznak
<b>Tagok elérése</b>	Bármelyik tagot el lehet érni bármikor	Egyszerre csak egy tagot lehet egy időben
<b>Inicializálása</b>	Bármennyi tagot lehet inicializálni egyszerre	Csak az első tagot tudjuk inicializálni.



# 7. Objektum orientált paradigma és annak megvalósítása a JAVA és C++ nyelvekben. Az absztrakt adattípus, az osztály. Az egységbezárás, az információ elrejtés, az öröklődés, az újrafelhasználás és a polimorfizmus. A polimorfizmus feloldásának módszere

---

## Objektum orientált paradigma

---

Az objektum orientált paradigma az **objektumok** fogalmán alapuló programozási paradigma. Az objektumok egységbe foglalják az adatokat és a hozzájuk tartozó műveleteket. A program egymással kommunikáló objektumok összességéből áll melyek használják egymás műveleteit és adatait.

Az objektum-orientáltság három alapillére:

- Egységbezárás és adatelrejtés (Encapsulation & information hiding)
- Újrafelhasználás, polimorfizmus és öröklődés (Reusability, polymorphism & inheritance)
- Magasabb fokú absztrakció

### Egységbezárás és adatelrejtés

Az egységbe zárás azt fejezi ki, hogy az összetartozó adatok és függvények, eljárások együtt vannak, egy egységbe tartoznak.

További fontos fogalom az **adatelrejtés**, ami azt jelenti, hogy kívülről csak az férhető hozzá közvetlenül, amit az objektum osztálya megenged.

Ha az objektum, illetve osztály elrejt az összes adattagját, és csak bizonyos metódusokon keresztül férhetnek hozzá a kliensek, akkor az egységbe zárás az absztrakciót és információelrejtés erős formáját valósítja meg

## Az osztály és objektum

**Absztrakt adattípus:** Az adattípus leírásának legmagasabb szintje, amelyben az adattípust úgy specifikáljuk, hogy az adatok ábrázolására és a műveletek implementációjára semmilyen előírást nem adunk.

**Osztály:** Egy absztrakt adattípus. Az adattagokból és a rajta elvégezhető műveleteket zárja egy egységbe. Egészen konkrétan objektumok csoportjának leírása, amelyeknek közös az attribútumaik, operációik és szemantikus viselkedésük van. Ugyanúgy viselkedik, mint minden egyéb primitív típus, tehát pl. változó (objektum) hozható létre belőlük.

- **Létrehozás:** Java-ban és C++ban is a class kulcsszóval tudunk osztályokat definiálni. Az osztályokból tetszőleges mennyiségben létrehozhatunk példányokat, azaz objektumokat.

**Objektum:** Egy változó, melynek típusa valamely objektumosztály, vagyis az osztály egy példánya amely rendelkezik állapottal, viselkedéssel, identitással. Az objektumok gyakran megfeleltethetők a való élet objektumainak vagy egyedeinek

- **állapot:** Egy az objektum lehetséges létezési lehetőségei közül (a tulajdonságok aktuális értéke, pl: lámpaBekapcsolva true vagy false)
- **viselkedés:** Az objektum viselkedése annak leírása, hogy az objektum hogy reagál más objektumok kéréseire. (metódusok, pl: lámpa.bekapcsol())
- **identitás:** Minden objektum egyedi, még akkor is, ha éppen ugyanabban az állapotban vannak, és ugyanolyan viselkedést képesek megvalósítani.

## Információ elrejtése

A láthatóságok segítségével tudjuk szabályozni adattagok, metódusok elérését, ugyanis ezeket az objektumorientált paradigma értelmében korlátozni kell, kívülről csak és kizárólag ellenőrzött módon lehessen ezeket elérni, használni.

Az adattagok, és metódusok láthatóságának vezérléséhez vannak kulcsszavak, amelyekkel megfelelően el tudjuk rejtetni őket.

Láthatósági opciók

- **public:** mindenholnan látható

- **protected:** csak az osztály scope-ján belül, illetve a később az adott osztályból származtatott gyerekosztályokon belül lehet hivatkozni.
- **private:** csak az adott osztályon belül lehet hivatkozni rá

(**Java-ban alapértelmezetten package private** (az adott package-n belül public, egyébként private) a láthatóság, míg **C++ -ban private**)

Törekedni kell a minél nagyobb adatbiztonságra és információ elrejtésre: az adat tagok láthatósága legyen private, esetleg indokolt esetben protected.

## Öröklődés

Osztályok között értelmezett viszony, amely segítségével egy általánosabb típusból (ősosztály) egy sajátosabb típust tudunk létrehozni (utódosztály). Az utódosztály adatokat és műveleteket örököl, kiegészíti ezeket saját adatokkal és műveletekkel, illetve felülírhat bizonyos műveleteket. A kód újrafelhasználásának egyik módja.

**Megkülönböztetünk egyszeres és többszörös öröklítést.**

A hasonlóság kifejezése az ős felé az általánosítás. A különbség a gyerek felé a specializálás.

**Java:** az extends kulcsszóval tudjuk jelezni, hogy az adott osztály egy másik osztálynak a leszármazottja. Java-ban egyszeres öröklődés van, vagyis **egy osztály csak is egy ősoosztályból származhat** (viszont több interfészt implementálhat)

- **super:** segítségével gyerekosztályból hivatkozhatunk szülőosztály adattagjaira és metódusaira.

**C++:** Az osztály neve után vesszővel elválasztva lehet megadni az ősoosztályokat és velük együtt a láthatóságaikat. **Lehetőség van többszörös öröklődésre is.**

- Az öröklődés során lehetőség van az ős osztály tagjainak láthatósági opcióján változtatni. Ezt az ős osztályok felsorolásakor kell definiálni. Az változtatás csak szigorítást (korlátozást) jelenthet. Az alábbi táblázat a gyermek osztálybeli láthatóságot mutatja be az ős osztálybeli láthatóság és a módosítás függvényében:

Base class member access specifier	Type inheritance	of	inheritance
	public	protected	private

Base class member access specifier	Type inheritance	of	inheritance
Public	protected	protected	private
Protected	protected	protected	private
Private	Hidden	Hidden	Hidden

## Virtuális öröklődés

Többszörös öröklődésnél előfordulhat olyan eset, amikor egy-egy ős osztály az öröklődési hierarchia különböző pontján ismét megjelenik. Ekkor a gyermek osztályban ennek az ős osztálynak több példánya jelenhet meg. Erre néhány esetben nincs szükség, például ha az ős osztály csak egy eljárás-erőforrás, akkor minden esetben elegendő egyetlen előfordulás a gyermek osztályokban.

A virtuális ős osztályt az öröklődésnél az ős osztályok felsorolásakor **virtual módosítóval** kell jelezni.

(Ha nem adom meg a virtual módosító szót, akkor az A osztály többször fog megjelenni a D osztály példányaiban. Hivatkozásnál mindig meg kell mondani, hogy az A melyik példányáról van szó: C::A::m\_iN, B::A::m\_iN.)

## Újrafelhasználás, Polimorfizmus:

Az újrafelhasználhatóság az OOP egyik legfontosabb előnye.

Az a jelenség, hogy egy változó nem csak egyfajta típusú objektumra hivatkozhat a polimorfizmus.

A polimorfizmus lehetővé teszi számunkra, hogy egyetlen műveletet különböző módon hajtsunk végre. Más szavakkal, a polimorfizmus lehetővé teszi egy interfész definiálását és több megvalósítást. Az objektumok felcserélhetőségét biztosítja. Az objektumok ős típusai alapján kezeljük, így a kód nem függ a specifikus típusoktól.

Polimorfizmusra két lehetőség van:

- **statikus polimorfizmus (korai hozzárendelés)** - a hívott metódus nevének és címének összerendelése szerkesztéskor történik meg. A futtatható programban

már fix metóduscímek találhatók. (statikus, private, final metódusok)

- **dinamikus polimorfizmus (késői hozzárendelés)** - metódus nevének és címének hozzárendelése a hívás előtti sorban történik, futási időben

## Virtuális eljárások

Egy virtuális eljárás címének meghatározása indirekt módon, futás közben történik.

Java-ban eleve **csak virtuális eljárások** vannak (**kivéve a final metódusokat**, amelyeket nem lehet felüldefiniálni és a **private metódusokat, amelyeket nem lehet örökölni**)

C++ -ban a **virtuális függvénytábla** tartja nyilván a virtuális eljárások címeit. A VFT táblázat öröklődik, feltöltéséről a konstruktor gondoskodik. A származtatott osztály konstruktora módosítja a virtuális függvénytáblát, kijavítja az ősoosztályból örökölt metóduscímeket. Amikor a konstruálási folyamat véget ér, a VFT táblázat minden sora értéket kap, mégpedig a ténylegesen létrehozott osztálynak megfelelő metódus címeiket. A VFT táblázat sorai ezután már nem változnak meg.

- Virtuális eljárásokat a **virtual kulcsszóval** tudunk létrehozni. Az újrafelhasználás során nagy valószínűséggel módosításra kerülő eljárásokat a szülő osztályokban célszerű egyből virtuálisra megírni, mert ezzel jelentős munkát lehet megtakarítani a későbbiekben.

## Absztrakt osztály, interfész

**Java:**

**Absztrakt osztályok:**

- Az abstract kulcsszóval hozható létre.
- Egy absztrakt osztályból nem hozható létre objektum.
- Tartalmazhat absztrakt metódusokat (absztrakt metódusnak nincs implementációja, azaz törzse), illetve nem absztraktokat
- Gyerek osztályban az abstract metódusokat felül KELL definiálni, ha példányosítható osztályt szeretnénk
- Ha egy osztály rendelkezik legalább egy absztrakt metódussal, akkor osztálynak is absztraktnak kell lennie
- Lehetnek adattagjai

**Interfész**

- Az interface kulcsszóval lehet létrehozni
- Egy speciális absztrakt osztály
- Nincsenek sem megvalósított metódusok, sem adattagok. Csupán metódus deklarációkat tartalmaz (Újabb javában lehet **public static final** lesz mindegyik adattag)
- Gyerekosztályban az **implements** kulcsszóval lehet implementálni

**C++:**

### **Absztrakt osztályok:**

A törzs nélküli virtuális eljárásokat **pure virtual** eljárásoknak nevezzük (pl.: virtual int getArea() = 0;). A pure virtual eljárás egy üres (NULL) bejegyzést foglal el a VFT (Virtual Function Table) táblázatban. Ha egy osztály ilyen eljárást tartalmaz, akkor azt absztrakt osztálynak nevezzük amiatt, mert ebből az osztályból objektum példányokat létrehozni nem lehet. A gyermek osztályokban minden pure virtual eljárást megfelelő törzsszel kell ellátni, ezt a fordító ellenőrzi. Amíg egyetlen pure virtual eljárás is marad, az osztály absztrakt lesz.

Interfészeket lehet szimulálni úgy, hogy minden metódust pure virtuallá teszünk.

## **8. Objektumok életrajza, létrehozás, inicializálás, másolás, megszüntetés. Dinamikus, lokális és statikus objektumok létrehozása. A statikus adattagok és metódusok, valamint szerepük a programozásban. Operáció és operátor overloading a JAVA és C++ nyelvekben. Kivételkezelés**

---

### **Objektumok létrehozása**

---

Az objektumokat Java-ban és C++ -ban is tárolhatjuk **statikusan** (az adatszegmensben), a **veremben** (lokálisan) vagy a **heapben** (dinamikusan).

Java-ban az objektumok mindig a heap-ben keletkeznek, kivéve a primitív típusokat.

### **Objektumok inicializálása, konstruktorok**

Az osztályok konstruktora fogja inicializálni az objektumot. A konstruktor neve meg kell egyezzen az osztály nevével. A konstruktornak nincs visszatérési értéke, de paraméterei lehetnek, amelyekkel meg lehet adni, hogy hogyan inicializáljuk az objektumot.

A **new** operátor:

- Szintaxis: new Osztály(args)
- Létrehozás lépései:
  - Lefoglalja a számára szükséges memóriát
  - Meghívja az osztály konstruktorát
  - Visszaadja az objektumra mutató referenciát

Egy osztályhoz készíthetünk több konstruktort, amelyek különböző paraméterlistával rendelkeznek.

**C++ban is hasonlóan működik a konstruktor:** a konstruktor inicializálja az objektumot, azaz tölti fel az adattagjait értékekkel, több különböző paraméter listájú konstruktort lehet létrehozni egy osztályhoz, a konstruktor neve meg kell egyezzen az osztály nevével és visszaadott értéke nem lehet.

A paraméter nélküli konstruktor eljárás neve: **alapértelmezett (default) konstruktor**.

Csak ős osztályokban kötelező, akkor ha az osztályból gyermek osztályokat szeretnének létrehozni öröklődéssel. Megvalósítható oly módon is, hogy egy nem default konstruktor minden paraméteréhez default eljárás paramétereit adjuk (pl. Osztaly(int x = 1, int y = 2)).

Amennyiben egy gyermek osztály konstruálunk, akkor a konstruktor minden esetben meg kell hívja rekurzívan az ős osztály(ok) konstruktorait mielőtt elkezdené a saját eljárás törzsét végrehajtani. Java-ban ez impliciten megtörténik, ha az ősosztálynak van default konstruktora.

C++ban a **heapbeli objektumok létrehozása a new operátorral** történik, **megszüntetésük pedig a delete operátorral**. A létrehozáshoz nem elegendő a memória megfelelő méretben történő lefoglalása, hanem a konstruktor eljárást is meg kell hívni. (Ezért nem lehet objektum példányt létrehozni malloc eljárással.)

A new operátorral egyetlen objektum példányt vagy megadott méretű tömböt hozhatunk létre. A new operátor alkalmazásának eredménye mindig egy pointer a new operandusában megadott osztályra.

### **Szintaxis:**

Tömbök foglálásakor a default konstruktor hívódik meg. Megszüntetésüknél az üres [] zárójelpár használata kötelező.

C++ban az objektum megszüntetése előtti takarítást, erőforrás felszabadítást a destruktor végzi. A neve meg kell egyezzen az osztály nevével, ami elé egy ~ (tilde) jelet is kell tenni. Paramétere és visszaadott értéke nem lehet. A destruktor már nem állíthatja meg az objektum megszüntetését. Amikor a destruktor véget ér, az objektumot a rendszer a memóriából törli. Mindig a gyerek osztály destruktora hívódik meg először, és azt követi rekurzívan az ős osztályok destruktorainak a meghívása.

Java-ban nincs szükség a heap-ben létrehozott objektumok manuális törlésére. A takarítást automatikusan elvégzi a garbage collector (szemétgyűjtő). Ez biztonságosabb, a programozónak nem kell emlékeznie, hogy fel kell szabadítani az erőforrásokat, viszont sokkal lassabb. A szemétgyűjtést kézzel is el lehet indítani, de ez nem egyenlő a destruktorral, kiszámíthatatlan, hogy mikor fog végrehajtódni.

## **Objektum másolás**

Akkor beszélünk klónozásról, ha egy objektum példányt két (vagy több) példányban sokszorosítunk úgy, hogy az egyes példányok adat tagjai azonosak lesznek.

Klónozás lehetséges az „=” segítségével, viszont ilyenkor az objektumok ugyan lemásolódnak, de a referenciájuk ugyanarra a memóriaterületre fog mutatni, azaz, ha pl. az egyik másolt objektum egyik adattagját módosítjuk, az az eredeti objektumra is hatással lesz.

### **Java:**

Valódi másolást Java-ban a clone() metódussal tudunk végrehajtani. Az osztálynak, amit szeretnénk klónozzhatóvá tenni implementálnia kell a Cloneable interfészt és meg kell hívnia az ős clone() metódusát (super.clone()).

### **C++:**

A valós klónozás megvalósítására szolgál a copy konstruktor. A copy konstruktor



paramétereinek száma 1, ennek az egy paraméternek a típusa pedig a tartalmazó osztályra mutató referencia típus.

## Dinamikus, lokális és statikus objektumok létrehozása:

---

### C++:

A **statikusan létrehozott objektum** az adott kód blokk végén megszűnik, amelyikben létre lett hozva.

**Lokális objektumokat** default paraméter vagy objektumokat tartalmazó kifejezésekben használhatunk. Szokás még objektum konstansnak is nevezni őket.

**Objektumokat dinamikusan** a new operátor segítségével tudunk létrehozni, amelynek törléséről a programozónak kell gondoskodnia.

### Java:

Java-ban minden objektum dinamikusan jön létre a heap-ben.

## A statikus adattagok és metódusok

A statikus adattagok és metódusok hasonlóan működnek Java-ban és C++ban. Mindkét nyelven a static módosítóval tudjuk jelezni, hogy az adott member statikus lesz.

Az ilyen adattagok és metódusok csak egy példányban jönnek létre és az osztálynak lesznek a tagjai, amelyet az objektumok közösen használhatnak.

A statikus metódusok nem lehetnek virtuálisak, nem hivatkozhatnak az adott objektumra (this-re).

Az ilyen adattagok, metódusok példányosítás nélkül is használhatóak.

Olyan esetekben lehetnek hasznosak, amikor az adott adattag, metódus független az objektumoktól, és mindenhol megegyezne az implementáció. Például van egy statikus adattagunk, amely a diákok számát tárolja és egy statikus metódus, amely visszaadja ennek a statikus adattagnak az értékét.

## Operáció és operátor overloading

---

### Operáció kiterjesztés

Az operáció kiterjesztés mind Java, mind C++ nyelven támogatott és hasonló módon működik. A lényege, hogy azonos nevű függvények többször vannak implementálva melyek paraméterei eltérő számúak és típusúak lehetnek. Ilyenek a változó paraméterű konstruktorok is többek között.

A fordító a kiterjesztett metódusokat a paraméterlistájuk alapján különbözteti meg

```
void sum(int a,int b){ System.out.println(a+b); }  
void sum(int a,int b,int c){ System.out.println(a+b+c); }
```

## Operátor kiterjesztés

### Java-ban nincs lehetőség az operátorok kiterjesztésére.

A C++ programozási nyelv lehetőséget biztosít arra, hogy az osztályokra kiterjesszük a nyelvben definiált bináris és unáris operátorokat. A kiterjesztésre vonatkozóan több megkorlát is van, ennek ellenére ez a szolgáltatás jelentős lépés az absztrakció növelésének irányába.

- A kiterjesztés CSAK osztályok esetén lehetséges (ebben benne van a class, struct és a union), viszont nem működik tömbökre, pointerekre.
- Bizonyos elemi operátorok kiterjesztésére nincs lehetőség, ilyenek a . (member selection), :: (scope resolution), ? : (ternary), \* (pointer to member), # és ## a preprocesszorból. Kiterjeszthető viszont a (typecast) operátor!
- Az operátorok precedenciája nem változtatható meg.
- Az operátor eljárások öröklődnek (kivéve az „=” operátor).
- Az operátorok egyik operandusa osztály (vagy osztályra mutató referencia típus) kell legyen. Ettől függetlenül lehet a két operandus különböző.

## A friend osztályok és eljárások

---

Az operátor eljárások implementációjakor szükség lehet objektumok private és protected adatainak elérésére. Az adatok elérésére használhatunk segéd eljárásokat, azonban itt egy speciális esetről van szó, ahol számításba jöhet egy "barát" eljárás alkalmazása is (talán szándékaink szerint metódussal szerettük volna megvalósítani az operátor eljárást, csak valamiért ez nem volt megengedett).

- A friend eljárást az osztály belsejében kell deklarálni.
- Nemcsak eljárás lehet friend, hanem másik osztály is!

- A friend eljárások nem lesznek az osztály tagjai!
- Abban az osztályban kell őket deklarálni, amelynek a tagjait el kívánják érni.
- A friend eljárásokat a global scope-ban kell megvalósítani.

## Kivételkezelés

---

A kivétel a program futása során előálló rendellenes állapot, amely közbeavatkozás nélkül a program futásának abnormális megszakadását eredményezheti. A kivételes helyzetek kezelésének elmulasztása például a hálózati a kommunikáció, vagy az adatbázis tranzakció félbeszakadásával járhat úgy, hogy mindeközben meghatározatlan állapotba kerül a rendszer.

A lényeg, hogy amikor egy hiba megjelenik a programban, azaz egy kivételes esemény történik, a program normális végrehajtása megáll, és átadódik a vezérlés a kivételkezelő mechanizmusnak.

A Java kivételkezelése a C++ kivételkezelésére alapul.

A kivételkezelés eszköze a try és a catch utasítás, míg a manuális kivétel kiváltásra szolgál a throw utasítás. A kivételkezelő blokk végén a finally mindenképpen lefut.

A program azon részeit, ahol a kivételek keletkezhetnek, és amiket utána kivétel kezelő részek követnek, a program védett régióinak nevezzük.

A kivétel bekövetkezésekor a throwbeli kifejezés típusának megfelelő catch blokk hívódik meg, ezt a veremben visszafelé haladva keresi meg a rendszer. A verem tartalmát az adott pontig kiüríti a rendszer, végrehajtja a catch blokkot, majd a try utáni sorral folytatja a végrehajtást.

## Kivétel létrehozása

Beépített kivétel osztályok mellett létrehozhatunk sajátokat is.

Java:

Ha akarunk, akár saját kivételeket is hozhatunk létre bármely kivétel osztály specializálásával. Ekkor egy osztályt az Exception osztályból kell származtatni és meg kell hívni az ősosztály konstruktorát.

C++:

```
class MyException : public std::exception {  
    std::string _msg;
```

```
public:
    MyException(const std::string& msg) : _msg(msg){}
    virtual const char* what() const noexcept override {
        return _msg.c_str();
    }
};
```

## 9. Java és C++ programok fordítása és futtatása. Parancssori paraméterek, fordítási opciók, nagyobb projektek fordítása. Absztrakt-, interfész- és generikus osztályok, virtuális eljárások. A virtuális eljárások megvalósítása, szerepe, használata

---

### C++ fordítás, futtatás

---

#### 1. Előfordítás

Első lépésben az előfordító(preprocessor) a tényleges fordítóprogram futása előtt szövegesen átalakítja a forráskódot.

Az előfordító különböző szöveges változtatásokat hajt végre a forráskódon, előkészíti azt a tényleges fordításra.

Feladatai:

- **Header fájlok beszúrása. (\*.hpp/.h)**
  - A **forrásfájlban (\*.cpp)** fizikailag több sorban elhelyezkedő forráskód logikailag egy sorbatörténő csoportosítása (ha szükséges).
  - A kommentek helyettesítése whitespace karakterekkel.
  - Az előfordítónak a programozó által megadott feladatok végrehajtása (szimbólumokbehelyettesítése, feltételes fordítás, makrók, stb.)
- A leggyakoribb műveletei a szöveg helyettesítés (**#define**),

a szöveges állomány beépítése (**#include**) valamint a program részeinek feltételtől függő megtartása

- Az előfeldolgozó az **#include** direktíva hatására az utasításban szereplő szöveges fájl tartalmát beszúrja a programunkba, a direktíva helyére.

## 2. Fordítás

Fordításkor a forrásfájlokból az első lépésben **tárgymodulok (\*.o) keletkeznek**, önmagukban nem futóképesek. (**Assembly kódot csinál**)

Ezt követően szükség van egy szerkesztőre, ami ezeket a modulokat összeszerkeszti. Linux/Unix rendszerek esetén a fordító a **gcc**. Az alábbi módon tudjuk lefordítani a több forrásfájlból álló projektet:

**gcc -o prog main.cpp class1.cpp class2.cpp**

Felsoroljuk azokat a fájlokat (a felsorolás sorrendje lényegtelen), amiket le szeretnénk fordítani. Fontos a main.cpp megadása hiszen ez a program belépési pontja.

## 3. Linkelés

A **-o prog**, megadásakor megadhatjuk a program nevét, ekkor prog néven hozza létre az .exe fájlt. Ha nem mondunk semmit, akkor az alapértelmezett exe fájl neve a.out lesz. Célszerű használni a -o kapcsolót. Az exe kiterjesztés csak Windows esetén van, Linux esetén csak futtatási jogú fájlt kapunk.

A fordító először mindegyiket lefordítja, melyek a .o kiterjesztésű tárgymodul fájlok lesznek, majd ezek összeszerkesztésre kerülnek

## Fordítási lehetőségek

- forrásfájlokból kiindulva: gcc -o prog class1.cpp class2.cpp
  - Ekkor modulonként létrejönnek a tárgymodulok .o kiterjesztéssel.
  - Amennyiben több forrásfájl van, akkor megoldható: gcc -o prog \*.cpp -ként is.
- tárgymodul és forrásfájl megadásával: Amely modulok nem változtak meg, azokat felesleges újralfordítani, tehát megadhatjuk tárgymodul és forrásfájl megadásával
  - F: gcc -o prog class1.o class2.cpp
- tárgymodulkönyvtár és forrásfájl felhasználásával:
  - a tárgymodulkönyvtár kiterjesztése .a
  - Tárgymodulkönyvtárat létrehozni (archiver) (-cr : create): ar -cr liba.a a.o
  - F: gcc -o prog b.cpp liba.a

- csak tárgymodulok felhasználásával: ekkor a -c kapcsolóval csak fordítást végzünk, szerkesztést nem.
  - F: gcc -c a.cpp b.cpp: Ekkor a.o és b.o tárgymodulokat kapunk
  - Ezt követően az ld nevű (link editor) szerkesztőprogrammal kell összeszerkeszteni a modulokat.
  - F: ld -o prog a.o b.o

## A gcc fordító fontosabb fordítási opciói

Szintaxis: gcc [kapcsolók] forrásfájlok

- **-Ob[szint]:** A gcc fordítónak a -Ob[szint] kapcsolóval tudjuk megmondani, hogy milyen optimalizálásokat alkalmazzon, a szint maximum 3 lehet (0,1,2), inline eljárások.
- **-c:** mint compile, lefordítja és összeállítja a forrást, linkelést nem végez.
- **-o:** lehetőségünk van megadni a futtatható állomány nevét, amennyiben nem adunk meg, az alapértelmezett az a.out lesz.
- **-Wall:** A figyelmeztetéseket írja ki.
- **-g:** engedélyezi a hibakeresési információk elhelyezését a programban, ami emiatt sokkal nagyobb lesz, de nyomon lehet követni a futását például a gdb programmal.
- **-Werror:** Fordítás-idejű figyelmeztetéseket errorokká alakítja.

## C++ parancssori paraméterek

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

A C++ programok kezdő eljárása minden esetben a main() eljárás. A main függvény első két paramétere az argc, ami egy int és az argv tömb:

- az argc a parancssorban szereplő argumentumok száma,
- az argv a string alakban tárolt argumentumok címeit tároló tömb, az első argumentum címe argv[0], a másodiké argv[1], ..., az utolsó argumentum után egy NULL pointer következik. Az argv[0] a program nevét és útvonalát tartalmazza. A paraméterek valójában az 1 indextől kezdődnek.

## Java fordítás, futtatás:

---

Ahhoz, hogy Java programokat tudjunk futtatni, illetve fejleszteni, szükségünk lesz egy fordító- és/vagy futtatókörnyezetre, valamint egy fordítóprogramra. A kész programunk futtatásához mindösszesen a JRE (Java Runtime Environment) szükséges, ami biztosítja a Java alkalmazások futtatásának minimális követelményeit, mint például a JVM (Java Virtual Machine)

Azonban a fejlesztéshez szükségünk lesz a JDK-ra (Java Development Kit) is. Ez tartalmazza a Java alkalmazások futtatásához, valamint azok készítéséhez, fordításához szükséges programozói eszközöket is (tehát a JRE-t nem kell külön letölteni, a JDK tartalmazza).

A fordítás folyamata az alábbiak alapján történik:

- Először a **.java** kiterjesztésű fájlokat a Java-fordító (compiler) egy közbülső nyelvre fordítja
- **Java bájtkódot kapunk eredményül** (ez a bájtkód hordozható). A java bájtkód a számítógép számára még nem értelmezhető. (kiterjesztése .class)
- Ennek a kódnak az értelmezését és fordítását gépi kódra a JVM (Java Virtual Machine) végzi el futásidőben.

**Fordítás:** `javac filename.java`

**Futtatás:** `java filename`

### Java fordítási opciók:

- **-g:** debug információk generálása
- **-s <könyvtár>:** a generált fájlok könyvtárának megadása
- **-sourcepath <path>:** a forrásfájlok elérési útvonalát meg lehet adni
- **-Werror:** figyelmeztetés esetén megáll a fordítás
- **-O:** Optimalizálás
- **-nowarn:** Ne legyen semmi figyelmeztetés

Java parancssori paraméterek

```
public static void main(String[] args)
```

A main függvény paramétere az args string tömb, amely tartalmazza a parancssori paramétereket. Ezen a tömbön valamilyen ciklus segítségével végig iterálhatunk és a parancssori paramétereket tetszés szerint kezelhetjük.

Nagyobb projektek esetén szokás build fájlokat alkalmazni: ant, gradle, makefile, stb.

## Virtuális eljárások

---

Egy virtuális eljárás címének meghatározása indirekt módon, futás közben történik. Java-ban eleve csak virtuális eljárások vannak (kivéve a final metódusokat, amelyeket nem lehet felüldefiniálni és a private metódusokat, amelyeket nem lehet örökölni)

C++ban a virtuális függvénytábla tartja nyilván a virtuális eljárások címeit. A VFT táblázat öröklődik, feltöltéséről a konstruktor gondoskodik. A származtatott osztály konstruktora módosítja a virtuális függvénytáblát, kijavítja az őosztályból örökölt metóduscímeket. Amikor a konstruálási folyamat véget ér, a VFT táblázat minden sora értéket kap, mégpedig a ténylegesen létrehozott osztálynak megfelelő metódus címeket. A VFT táblázat sorai ezután már nem változnak meg.

- Virtuális eljárásokat a virtual kulcsszóval tudunk létrehozni. Az újrafelhasználás során nagy valószínűséggel módosításra kerülő eljárásokat a szülő osztályokban célszerű egyből virtuálisra megírni, mert ezzel jelentős munkát lehet megtakarítani a későbbiekben.

### **Java:**

#### Absztrakt osztályok

- Az abstract kulcsszóval hozható létre.
- Egy absztrakt osztályból nem hozható létre objektum.
- Tartalmazhat absztrakt metódusokat (absztrakt metódusnak nincs implementációja, azaz törzse), illetve nem absztraktokat
- Gyerek osztályban az abstract metódusokat felül KELL definiálni, ha példányosítható osztályt szeretnénk
- Ha egy osztály rendelkezik legalább egy absztrakt metódussal, akkor osztálynak is absztraktnak kell lennie
- Lehetnek adattagjai

#### Interfész

- Az interface kulcsszóval lehet létrehozni
- Egy speciális absztrakt osztály
- Nincsenek sem megvalósított metódusok, sem adattagok. Csupán metódus deklarációkat tartalmaz
- Gyerekosztályban az implements kulcsszóval lehet implementálni

### **C++:**

#### Absztrakt osztályok:

A törzs nélküli virtuális eljárásokat pure virtual eljárásoknak nevezzük (pl.: virtual int



getArea() = 0;). A pure virtual eljárás egy üres (NULL) bejegyzést foglal el a VFT (Virtual Function Table) táblázatban. Ha egy osztály ilyen eljárást tartalmaz, akkor azt absztrakt osztálynak nevezzük amiatt, mert ebből az osztályból objektum példányokat létrehozni nem lehet. A gyermek osztályokban minden pure virtual eljárást megfelelő törzzsel kell ellátni, ezt a fordító ellenőrzi. Amíg egyetlen pure virtual eljárás is marad, az osztály absztrakt lesz.

## Generikus osztályok

---

Az generikus programozás módszere a kód hatékonyságának növelése érdekében valósul meg. Az általános programozás lehetővé teszi a programozó számára, hogy általános algoritmust írjon, amely minden adattípussal működik. Nincs szükség több, különféle algoritmusok létrehozására, ha az adattípus egész szám, karakterlánc vagy karakter.

Java

Lehetőség nyílt az osztályok paraméterezésére más típusokkal.

Gyakorlatilag statikus polimorfizmusról van szó, egy típusparamétert adunk meg, mivel az osztály maga úgy lett megírva, hogy a lehető legáltalánosabb legyen, és ne kelljen külön IntegerList, StringList, AllatList, stb. osztályokat megírnunk, hanem egy általános osztályt, mint sablont használunk, és a tényleges típust a kacsacsőrök között mondjuk meg.

Primitív típusal nem lehet paraméterezni, az fordítási hibát okoz.

A típusparamétereket konvenció szerint egyetlen nagybetűvel szokás elnevezni, hogy egyértelműen megkülönböztethető legyen.

Gyakori elnevezések:

- E : Element (tárolók használatánál)
- K : Key
- N : Number
- T : Type
- V : Value

Néha szükség lehet, hogy a típusparaméterre valamilyen megszorítást tegyünk:

- `public class NaturalNumber`
- Wildcard-ok, ismeretlen típusok:
  - `public void process(List<? extends Foo> list)`

- minden olyan listára, ami vagy a Foo, vagy annak leszármazottaiból áll
- `public void addNumbers(List<? super Foo> list)`
- minden olyan listára, ami vagy a Foo, vagy annak őseiből áll

C++

C++ban generikus osztályokat sablonok (template) segítségével tudunk létrehozni.

A függvénysablonok speciális funkciók, amelyek generikus típusokkal működhetnek. Ez lehetővé teszi számunkra, hogy létrehozzunk egy függvénysablont, amelynek funkcionalitása egynél több típushoz vagy osztályhoz igazítható anélkül, hogy megismételnénk az egyes típusok teljes kódját.

## 10. A programozási nyelvek csoportosítása (paradigmák), az egyes csoportokba tartozó nyelvek legfontosabb tulajdonságai

---

### Paradigmák

---

A programozási paradigma egy osztályozási forma, amely a programozási nyelvek jellemzőin alapul.

### Imperatív

---

Utasításokat használ, hogy egy program állapotát megváltoztassa.

### Procedurális

A feladatokat felbonthatjuk elvégzendő feladatok szerint, tehát *alprogramokat* (függvény, eljárás) hozunk létre. Ezek között paraméterátadással, függvény visszatérési értékkel kommunikálnak.

Pl: C, C++,...

## Objektumorientált paradigma

Az objektum orientál paradigma az objektumok fogalmán alapuló programozási paradigma. Az objektumok **egységbe foglalják az adatokat** és a hozzájuk tartozó **műveleteket**. A program **egymással kommunikáló objektumok összességéből áll** melyek használják egymás műveleteit és adatait.

**Öröklődés** osztályok között, egyszeres vagy többszörös öröklődéssel.

Lehetséges **polimorfizmus és absztrakt/interfész osztályok létrehozására**.

**A polimorfizmus** lehetővé teszi számunkra, hogy egyetlen műveletet különböző módon hajtsunk végre. Más szavakkal, a polimorfizmus lehetővé teszi egy interfész definiálását és több megvalósítást. Az objektumok felcserélhetőségét biztosítja. Az objektumok őstípusai alapján kezeljük, így a kód nem függ a specifikus típusoktól.

**Polimorfizmusra két lehetőség van:**

- **statikus polimorfizmus (korai hozzárendelés)** - a hívott metódus nevének és címének összerendelése szerkesztéskor történik meg. A futtatható programban már fix metóduscímek találhatók. (statikus, private, final metódusok)
- **dinamikus polimorfizmus (késői hozzárendelés)** - metódus nevének és címének hozzárendelése a hívás előtti sorban történik, futási időben

A legtöbb OOP nyelv osztályalapú, azaz az objektumok osztályok példányai és típusuk az osztály.

## Smalltalk

GNU Smalltalk interpreter

Beolvas minden karaktert az első **! -ig**. A „!” jellel jelezzük, hogy végre szeretnénk hajtani az addig beírt kifejezéseket. Több kifejezés futtatása esetén itt is – mint sok más nyelven – jeleznünk kell azt, hogy hol fejeződik be egy kifejezés erre való a **„pont” (.)**

## Precedencia

Ha nem zárójelezünk – mindig balról jobbra történik, így a  $2+3*10$  értéke 50 lesz, használjunk zárójelet:  $2+(3*10)$ .

Objektumok, üzenetek

A Smalltalk nyelv egy objektumorientált nyelv **MINDENT** objektumnak tekintünk.

A programozás során üzeneteket küldünk az egyes objektumoknak. Egy objektumnak háromféle üzenetet küldhetünk:

- **Unáris:** szintaxis: 'Hello' printNI !
- **Bináris:** szintaxis: 3+5
- **Kulcsszavas:** szintaxis: tomb at:1 put: 10

**Objektumok összehasonlítása:** két objektum egyenlő, ha ugyanazt az objektumot reprezentálják és azonos, ha értékük megegyezik és egyazon objektumok.

## Objektumok másolása

- **deepCopy (unáris üzenet):** Teljes másolat készítése objektumról.
- **shallowCopy (unáris üzenet):** Felszíni másolat
- **copy (unáris üzenet):** Osztályonként változó lehet, az Object osztályban a shallowCopy-t jelenti.

## Metaosztály

Mint korábban említettük, a Smalltalkban mindent objektumnak tekintünk. **Még az osztályok is objektumok.** De ha az osztály objektum, akkor az is - *mint minden más objektum* - valamilyen osztályhoz kell tartozzon. Másképp fogalmazva minden osztály (pontosan) egy másik osztály példánya. Ezen "másik" osztályt **metaosztálynak** hívjuk

## Object osztály

Az Object osztály minden osztály közös **őse**, tehát minden objektum az Object osztály egy példánya. Ezért minden, az Object osztálynak rendelkezésre álló művelettel minden más objektum is rendelkezik.

- **class** – unáris: visszatérése az objektum osztálya
- **isMemberOf** – kulcsszavas: visszatérése logikai érték. Ha a címzett objektum példánya ezen osztálynak, akkor "true" a visszatérési érték, egyébként "false"
  - 'Hello' isMemberOf: String ! → true

## Változók

- Lokális változók:
  - |x y z| - deklarálása (2 pipeline között)
  - x := 2. (egyszeres értékadás)

- `x := y := z := 2.` (többszörös értékadás)
- Globális változók: `Smalltalk at: #valtozonev put: érték !`

## Blokkok

Más programozási nyelveken megismert programblokkok szerepével egyezik meg. Vannak paraméteres és nem paraméteres blokkok. Paraméteres blokkok rendelkeznek lokális változókkal, melyeknek a blokk kiértékelésekor adunk értéket. A változók élettartama és láthatósága korlátozódik az adott blokkra.

- `[i | i printNI ] value: 5`
- `['Hello' print . 'world' printNI] value.`

## Vezérlési szerkezetek

- **Feltételes vezérlés:** `valtozo > 10 ifTrue: ['x erteke nagyobb 10-nel' printNI] ifFalse: ['x erteke nem nagyobb 10-nel' printNI]`
- **Ismétléses vezérlés:** `[a<10] whileTrue: [a printNI . a:=a+1]`
- **For ciklus:** `1 to: 10 do: [:i | i printNI]`

## Kollekciók

- **Set:** ismétlés nélküli rendezetlen halmaz - `new, add()`
- **Bag:** olyan Set, amiben megengedjük az ismétlődést - `new, add()`
- **Dictionary:** egy asszociatív tömb (egy olyan tömb, amit nem csak számokkal, hanem (itt) tetszőleges objektummal is indexelhetünk)

## Tömb

- `tömb := Array new: 10`
- `tömb at: 1`
- `tömb at: 1 put: obj`

## A collect

- kollekció elemein lépked végig, mely minden egyes elemére végrehajtja az üzenet argumentumblokkjában található utasításokat
- `|tomb| tomb := #(10 3 43 29) collect: [:tombelem | tombelem*2]`

## Osztályok

- **példányváltozók:** minden objektum rendelkezik vele
- **osztályváltozó:** kb. statikus globális változó

## Metódusok definiálása osztályokhoz

pl.:

Beolvasás `x := stdin.nextLine.S`

Integer üzenetek

## Dekleratív programozás

---

Deklaráljuk a program elvárt működését, nem akarjuk explicit meghatározni annak mikéntjét.

## Funkcionális programozás

- Értékek, kifejezések és függvények vannak
- A program maga egy függvény
- Ciklus helyett **rekurzió**
- A funkcionális programnyelvek a programozási feladatot egy függvény kiértékelésének tekintik.
- A két fő eleme az **érték** és a **függvény**, nevét is függvények kitüntetett szerepének köszönheti.
- **Egy más megfogalmazás szerint, a funkcionális programozás során a programozó inkább azt specifikálja programban, mit kell kiszámítani, nem azt, hogy hogyan, milyen lépésekben.**
- Függvények hívásából és kiértékelésből áll a program. Nincsenek állapotok, mellékhatások (nem számít, mikor, csak az melyik függvényt hívjuk).

## Haskell

Egy tisztán funkcionális programozási nyelven írt programban nem a kifejezések egymásutánján van a hangsúly.

**Erősen vagy statikusan típusos nyelv**, így ahol a nyelv T-típust várja, csak T-típusra kiértékelődő kifejezést fogad el.

**A program egy függvényhívással hajtódik végre.**

Egy funkcionális program típus-, osztály-, és függvénydeklarációk, illetve definíciók

sorozatából és egy kezdeti kifejezés kiértékeléséből áll.

**A kiértékelést** úgy képzeljük el, mint a kezdeti kifejezésben szereplő függvények behelyettesítését.

Tehát egy program végrehajtása nem más, mint a kezdeti kifejezésből kiinduló redukciós lépések sorozata. Egy **kifejezés normál formájú**, ha már tovább nem redukálható (nem átírható) állapotban van. **Egy redukálható kifejezést redexnek hívunk.**

## Kiértékelési módok

A Haskell nyelv a **lusta kiértékelési stratégiát használja.**

**A lusta kiértékelés során** mindig a legkülső redex kerül helyettesítésre, az argumentumokat csak szükség esetén értékeli ki. Ez a **módszer mindig megtalálja a kezdeti kifejezés normál formáját.**

A mohó kiértékelés az argumentumok kiértékelésével kezdődik, csak ezután hajtja végre a függvény alkalmazásának megfelelő redukciós lépést.

## Futtatás

Elindítjuk a Haskell interpretert (hugs) és betöltjük az általunk megírt definíciós forrásállományt. Betöltés után rendelkezésre áll az összes általunk megírt függvény, melyek közül bármelyiket meghívhatjuk a függvény nevének beírásával (a megfelelő paraméterezéssel). Amennyiben módosítjuk a definíciós állományt, újra kell tölteni azt.

**Atomi típusok:** Int, Float, Bool

Függvények definiálása

A visszatérési értéket a kiértékelése határozza meg, ami lehet egy konstans érték vagy akár egy rekurzív kifejezés is

Esetvizsgálatok

Függvény paramétere függvény

Lokális definíciók függvénydefiníciókban

Típusok létrehozása

## Logikai programozás

A problémakörrel kapcsolatos **tényeket** logikai képletek formájában fejezik ki, és a programokat **következtetési szabályok** alkalmazásával hajtják végre, amíg nem találunk választ a problémára, vagy a képletek halmaza nem következetes.

## Prolog

Prolog program csak az **adatokat** és az **összefüggéseket** tartalmazza. **Kérdések** hatására a programvégrehajtást beépített **következtető-rendszer** végzi.

A logikai programok egy modellre vonatkoztatott állítások halmaza, melyek a modell tulajdonságait és azok között fellépő kapcsolatokat (relációit) írják le.

Egy adott relációt meghatározó állítások részhalmazát predikátumnak nevezzük. A **predikátumokat alkotó állítások tények vagy szabályok lehetnek**. A tényeket és szabályokat (és majd a Prolognak feltett kérdéseket is) **ponttal zárjuk le**.

Tekintsük a következő példát, mely egy család tagjai között fellépő kapcsolatot írják le.

A szülő predikátum argumentumait szándékosan írtuk kis betűkkel. A kis betűkkel írtakat a Prolog konstansként kezeli. (ka, katalin, szilvia, stb...) Minden nyomtatott nagybetűt vagy nagy kezdőbetűvel kezdődőket változónak tekint. (X, Y, Szilvia, Magdolna, stb...)

Egy prolog program csak az **adatokat** és az **összefüggéseket** tartalmazza, majd **kérdések hatására** a programvégrehajtás beépített **következtető-rendszer** végzi.

## Futtatás

- kiterjesztés **.pl**
- A Prolog egy terminálablakba beírt „sicstus” paranccsal indítható. Egy Prolog állományt a következőképpen „tölthetjük be”: (feltéve, hogy az aktuális könyvtárban létezik egy [prolog.pl](#) állomány)

## A Prolog program felépítése

- Prolog érték: **term**
  - Egyszerű term: alma, 1000,...
  - Összetett termék
    - **Lista:** nagyon hasonlít a Haskell-ben megismert listára. Itt sincsenek indexelve az elemek, rekurzióval fogjuk bejárni a listát. Példa listára: [1,2,3,4,5].  
Kiértékelés  
Kifejezések kiértékelésére a beépített, **infix is operátort**
- Relációk megadása:
  - Tények
  - Következtetés szabályok
- Kérdésfeltevés interaktív módon
  - Eldöntendő kérdés



- Általános kérdés

### Tények:

Tények fejezik ki, hogy a megadott objektumok között fennáll bizonyos reláció.  
barát(john, mary).

Ezek egy adatbázis definiálnak.

### Kérdések:

Eldöntendő kérdések ugyanúgy néznek ki, mint a tények csak más a szövegkörnyezet.  
?- barát(john, mary).

### Következtető rendszer:

- Prolog **backtracking** keresést alkalmaz a válaszok megtalálásra.
- Részcélokra bontás majd egymás után válasz keresés!
- Célokat és a tényeket illesztéssel kapcsolja össze.
- Ha nem talál rész célra válasz, akkor **visszalép** az előző rész célra és új illeszkedő elemet talál rá.

## Párhuzamos programozás

---

Egyszerre több szálon történik a végrehajtás → végrehajtási szál: **folyamat (process)**

Előnyei:

- Természetes kifejezésmód
- Sebességnövekedés

Hátrányai:

- Bonyolultabb a szekvenciálisnál

Sokféle probléma léphet fel a **közös memória** és az **osztott memória** adathozzáférés miatt.

Kezelnit kell a folyamatok létrehozását és megszüntetését és együttműködését.

Felléphet **holtpont** = Kölcsönös egymásra várakozás, vagy **éhezés**, amikor nincs holtpont mégis erőforráshoz nem jut hozzá.

## Occam

**Imperatív**, folyamatok saját memóriával rendelkeznek, üzenetküldéssel kommunikálnak.

**Occam program részei:**

- Változók
- Folyamatok
- Csatornák

**Csatornák:**

A csatorna két folyamat közti **adatátvitelre** szolgál

- Egyirányú
- Küldős és fogadó is legfeljebb egy lehet
- biztonságos
- **Szinkron:** A küldő és fogadó bevárják egymást, megtörténik az adatátvitel, majd a küldő és fogadó folytatódik.

**Folyamatok:**

Életciklus:

- Elindul
- Csinál valamit
- Befejeződik

Befejezésnél **holtpontba** kerülhet, erre odakell figyelni.

**Elemi folyamatok:**

- Üres utasítás - **SKIP**
- Beépített holtpont - **STOP**
- Értékadás -  $v := e$
- Input -  $c ? v$
- Output -  $c ! e$

Az Occam egy **párhuzamos programozási nyelv**. Ezen paradigma szerint az **egyes folyamatok párhuzamosan futnak**. Ez több processzoros gépek esetén valós párhuzamosságot jelent (egy processzor egy folyamatot dolgoz fel), de egy processzor esetén ez nyilván nem valósulhat meg, az **egyes folyamatok „időszelleteket” kapnak, az Occam a párhuzamosságot időosztással szimulálja**. Az egyes folyamatok közötti kommunikáció csatornákon keresztül valósul meg.

**A P1 és P2 folyamatok a C csatornán keresztül kommunikálnak.**

A **folyamatok közötti kommunikációt mindig csatornákkal valósítjuk meg**. A fenti példában a P1 folyamat a C csatornán keresztül valamilyen adatot küld a P2 folyamatnak. Ez a következőképpen valósul meg: ha egy folyamat elérkezik arra a pontra, ahol értéket küld [fogad], várakozik a másik folyamatra, amíg az is el nem ér a fogad [küld] pontra. Amikor mindketten készen állnak az adatcserére (azaz mindkét

folyamatban a küldés [fogadás] pontra került a vezérlés) létrejön az adatcsere, majd mindkettő folytatja a futását.

## Fordítás

- KroC, csak Linux-hoz
- `kroc -d pelda.occ`  
Fontos tudnivalók a nyelvről
- Minden, a nyelvben lefoglalt kulcsszót nagy betűvel kell írni (SEQ, PAR, PROC, stb...)
- A blokkstruktúrát indentációval jelöljük (két szóközzel beljebb kezdjük)
- Minden egyes kifejezés új sorban kezdődik (esetlegesen két szóközzel beljebb)
- Egy Occam program a következőképpen épül fel:  
<deklarációk>
- Például:

## Elemi folyamatok

A fenti példában, küldés esetében egy kifejezést ( $k + 5$ ) küldünk a C csatornára, fogadás esetén pedig a C csatornáról várunk egy értéket, amely az x változóban kerül.

A SKIP folyamat a legegyszerűbb elemi folyamat, „semmit nem csinál”. Haszontalannak tűnhet, de összetettebb programok esetében (például még nem kifejlesztett programrészek esetében) hasznos lehet. Párhuzamos folyamatok esetében fontos, hogy minden folyamat termináljon, ellenkező esetben az egész, folyamatokból álló „rendszer” leáll.

A STOP szintén „nem csinál semmit”, de ez sosem terminál – ellentétben a SKIP-el. Egy folyamatban a STOP (feltéve hogy a vezérlés odakerül), annak holtpontba jutását eredményezi. Szintén haszontalannak tűnhet, de ezzel egy folyamatot leállíthatunk más folyamatok működésének befolyásolása nélkül, ami hibakeresésnél hasznos lehet. Azt mondjuk, hogy egy folyamat holtpont állapotba került, ha az már nem képes további működésre (vezérlése leáll), és ez a leállás nem a folyamat helyes lefutásának eredménye. Párhuzamos folyamatok közül akár egy folyamat holtpont állapotba kerülése az egész program holtpont állapotba kerülését eredményezi, hiszen az összes többi folyamat várja a holtpontban levő folyamat terminálását, ami sosem fog bekövetkezni.

Blokk struktúra 2 szóközönként beljebb kell kezdeni

## Precedencia

A kifejezésekben, operátorok között precedenciát nem határozunk meg, így MINDIG zárójelezést kell használni a precedencia meghatározásához

## **Adattípusok**

Csatorna

SEQ

PAR

Az egész PAR blokk akkor terminál, ha a benne „elindított” folyamatok mindegyike terminál

PROC

A PROC egy előre definiált, névvel ellátott folyamat. Tekintheünk úgy rá, mintha egy eljárást definiálnánk

ALT

Ha egy őr engedélyezetté válik, akkor a benne megadott változó felveszi a csatornáról érkező adat értékét és „elindítja” a hozzá tartozó folyamatot

Az x változó értéke attól függ, hogy c1-re vagy c2-re érkezik előbb adat.

Mivel a program írásakor nem tudhatjuk, hogy melyik csatornáról fog adat érkezni, ezért az ALT-ot tartalmazó programok nemdeterminisztikusak

Függvény

Vezérlési szerkezetek

- Feltételes vezérlés  
Holtpont elkerülése
- Ismétléses vezérlés
- For ciklus

# **11. Szoftverfejlesztési folyamat és elemei; a folyamat különböző modelljei**

---

**A szoftverfolyamat:** Tevékenységek és kapcsolódó eredmények, amely során elkészítjük a szoftvert.

## Alapvető elemek

---

- **Szoftverspecifikáció (mit):**
  - a szoftver funkcióit és korlátait meg kell határozni
  - *Legkisebb a változás költséges*
  - Eredménye a **követelményspecifikáció**
- **Szoftvertesztelés és implementáció (hogyan):**
  - a specifikációnak megfelelően a szoftvert elő kell állítani
  - Alrendszerek meghatározása, komponens tervezés stb.
- **Szoftvervalidáció (ellenőrzés):**
  - a szoftvert ellenőrizni kell, hogy tényleg azt fejlesztettük ki, amit az ügyfél kíván.
  - **Verifikáció:** Rendszer megfelel e a specifikációnak
  - **Validáció:** Megfelel e a megrendelő elvárásainak
- **Szoftverevolúció (változás):** a szoftvert úgy alakítani, hogy megfeleljen a későbbi kívánságoknak

## A szoftverfolyamat modelljei

---

A szoftverfolyamat modellje a szoftverfolyamat absztrakt reprezentációja. Ezek a modellek egy-egy egyedi perspektívából reprezentál egy szoftverfolyamatot, de nem pontos specifikációja annak. Sokkal inkább hasznos absztrakciók, amit a szoftverfejlesztési folyamat különböző megközelítési módjainak megértéséhez használunk.

### Vízesés modell

- **Specifikáció:** rögzítjük a termék követelményeit. Mit tudjon a szoftver, és mit nem.
- **Tervezés:** szétválasztódnak a szoftver- és hardverkövetelmények. Megtervezzük a rendszer architektúráját.

- **Implementáció:** a szoftver fejlesztése, egységtesztelése. Az egységtesztelés azt a célt szolgálja, hogy a szoftver minden egyes egysége megfelel-e a specifikációnak.
- **Verifikáció:** a különálló programegységes és programok integrálása és teljes rendszerként való tesztelése, hogy a rendszer megfelel-e a specifikációnak. A tesztelés után a rendszer átadható az ügyfélnek.
- **Karbantartás:** a szoftver életciklusának leghosszabb fázisa. A karbantartásba beletartozik olyan hibák javítása is, amelyek nem merültek fel az életciklus korábbi szakaszaiban, illetve a szolgáltatások továbbfejlesztése.

A fázisok eredménye egy vagy több dokumentum, amelyek jóváhagyása megtörtént. A következő fázis nem indulhat, amíg az előző be nem fejeződött.

**Probléma:** a folyamat korai szakaszaiban állást kell foglalnunk és el kell köteleznünk magunkat, és nehéz az ügyfélhez történő alkalmazkodás. Akkor jó, ha előre ismerjük a követelményeket. Nagyobb rendszerek kisebb folyamatainál használják főleg.

## Evolúciós fejlesztés

Az evolúciós fejlesztés lényege, hogy ki kell fejleszteni egy korai implementációt, azt a felhasználókkal véleményeztetni, és finomítani a felhasználói visszajelzések alapján, amíg megfelelő rendszert el nem élünk.

Két különböző típusa ismert:

- **Feltáró fejlesztés:** a folyamat célja az hogy a megrendelővel együtt feltárjuk a követelményeket, és kialakítsuk a végleges rendszert. A végleges rendszer úgy alakul ki, hogy egyre több, az ügyfél által kért tulajdonságot társítunk a már meglévőkhöz.
- **Eldobható prototípus fejlesztése:** ekkor az evolúciós fejlesztés célja, hogy minél jobban megértsük az ügyfél követelményeit, és azokra alapozva a legpontosabban fejlesszük le a terméket.

Az evolúciós fejlesztés jobb, mint a vízesés modell, ha a lehető legpontosabban szeretnénk az ügyfél kívánságainak megfelelő szoftvert fejleszteni. Előnye, hogy a specifikáció inkrementálisan fejleszthető.

A vezetőség és a tervezők szempontjából két probléma merülhet fel:

- A folyamat nem látható. A menedzsereknek rendszeresen leszállítható eredményekre van szükségük, hogy mérhessék a fejlődést.

- A rendszerek sokszor szegényesen struktúráltak. A folyamatos változtatások rontják a szoftver struktúráját.

A várhatóan rövid élettartamú kis vagy közepes rendszerek esetén az evolúciós megközelítési mód a legcélravezetőbb.

## **Iterációs, inkrementális modell**

- Folyamat iterációja elkerülhetetlen
- ha a követelmények változnak, akkor a folyamat bizonyos részeit is változtatni kell
- ennél a modellnél minimális a specifikáció, fejlesztésben sok iteráció van, és menet közben alakul ki a végleges specifikáció
- Inkrementalitás: részfunkciókkal már működő rendszert fejlesztünk, amit minden iterációban (inkrementálisan) javítunk
- Nagy körvonalakban specifikáljuk a rendszert
  - „Inkremensek” meghatározása
  - Funkcionalitásokhoz prioritásokat rendelünk
  - Magasabbakat előbb kell biztosítani
- Architektúrát meg kell határozni
- További inkremensek pontos specifikálása menet közben történik
- Egyes inkremensek kifejlesztése történhet akár különböző folyamatokkal is - Vízésés vagy evolúciós, amelyik jobb
- Az elkészült inkremenseket akár szolgálatba is lehet állítani
- Ha határidő csúszás van kilátásban a teljes projekt nem lesz kudarcra ítélve, esetleg csak egyes inkremensek
- Megfelelő méretű inkremensek meghatározása nem triviális feladat
  - Ha túl kicsi: nem működőképes
  - Ha túl nagy: elveszítjük a modell lényegét

Bizonyos esetekben számos alapvető funkcionálisitást kell megvalósítani. Egész addig nincs működő inkremens

## **eXtreme Programming (XP)**

- Szélsőséges inkrementális modell
- Nagyon kis funkcionálisú inkremensek
- Megrendelő intenzív részvétele fontos
- Programozás csoportos tevékenység - többen ülnek a képernyő előtt

- Sok támadója van

## **RAD (Rapid Application Development)**

- Extrém rövid élekciklus
  - Működő rendszer 60-90 nap alatt
- Vízés modell „nagysebességű” adaptálása
  - Párhuzamos fe