

Szegedi Tudományegyetem
Programtervező informatikus (BSc.)

Záróvizsga tételek kidolgozása

2020/2021 II. - Szeles László



Tartalomjegyzék

Algoritmusok és adatszerkezetek I - 1.)	7
Mohó algoritmus	7
Oszd-meg-és-uralkodj (D&C)	8
Dinamikus programozás (DP)	8
Rendező algoritmusok	9
Gráfalgoritmusok	15
Összegzés	18
Algoritmusok és adatszerkezetek I - 2.)	20
Elemi adatszerkezetek	20
Bináris keresőfák	22
Hasító táblázatok	24
Gráfok és fák számítógépes reprezentációja	26
Összegzés	27
Bonyolultságelmélet - 1.)	29
Hatókony visszavezetés	29
Nemdeterminizmus	32
A P és NP osztályok	32
NP teljes problémák	33
Összegzés	36
Bonyolultságelmélet - 2.)	38
A PSPACE osztály	38
PSPACE teljes problémák	38
Logaritmikus tárígényű visszavezetés	38
NL-teljes problémák	38
Összegzés	39
Formális nyelvek - 1.)	40
Véges automaták és a felismert nyelv	41
A reguláris nyelvtanok	43
Véges automaták és reguláris kifejezések ekvivalenciája	43
Reguláris nyelvekre vonatkozó pumpáló lemma	44
Összegzés	45
Formális nyelvek - 2.)	46
Környezetfüggetlen nyelvtan és nyelv	46
Derivációk és derivációs fák	46
Veremautomaták és környezetfüggetlen nyelvtanok ekvivalenciája	47
A Bar-Hillel lemma	48
Összegzés	50

Közelítő-, és szimbolikus számítások - 1.)	52
Eliminációs módszerek	52
Mátrixok trianguláris felbontásai	52
Lineáris egyenletrendszerek megoldása iterációs módszerekkel	52
Mátrixok sajátértékeinek és sajátvektorainak numerikus meghatározása	55
Összegzés	56
Közelítő-, és szimbolikus számítások - 2.)	57
Érintő-, szelő-, és húrmódszer	57
A konjugált gradiens eljárás	57
Lagrange interpoláció	57
Numerikus integrálás	57
Összegzés	58
Logika és informatikai alkalmazásai - 1.)	59
Normálformák az ítéletkalkulusban	59
Teljes rendszerek	61
Hilbert-kalkulus	62
Összegzés	64
Logika és informatikai alkalmazásai - 2.)	66
Normálformák a predikátumkalkulusban	66
Egyesítési algoritmus	68
Következtető módszerek	69
Alap rezolúció (ground rezolúció)	69
Elsőrendű rezolúció	69
Összegzés	71
Mesterséges intelligencia I - 1.)	73
Keresési feladat	73
Informálatlan (vak) keresés	73
Informált keresés és heurisztikák	74
Kétszemélyes, lépésváltós, determinisztikus, zéró összegű játékok	75
Korlátozás kielégítési feladat	78
Összegzés	79
Mesterséges intelligencia I - 2.)	82
Teljes együttes eloszlás tömör reprezentációja, Bayes hálók	82
Gépi tanulás: felügyelt tanulás problémája	84
Döntési fák	84
Naiv Bayes módszer	84
Modellillesztés	84
Mesterséges neuronhálók	84
K-legközelebbi szomszéd módszere	84
Összegzés	85

Operációkutatás I - 1.)	86
LP alapfeladata	86
Az LP geometriája	86
Generálóelem választási szabályok	87
Kétfázisú szimplex módszer	87
Speciális esetek	87
Összegzés	88
Operációkutatás I - 2.)	89
Primál-duál feladatpár	89
Dualitási komplementáris feltételek	89
Egészértékű feladatok és jellemzőik (ILP)	89
A Branch-and-Bound módszer	89
A hátizsák feladat	89
Összegzés	90
Operációs rendszerek - 1.)	91
Processzusok	91
Ütemezési stratégiák és algoritmusok	93
Ütemezési algoritmusok céljai	94
Stratégiák kötegelt rendszerekben	94
Stratégiák interaktív rendszerekben	95
Stratégiák valós idejű rendszerekben	97
Kontextus-csere	97
Összegzés	98
Operációs rendszerek - 2.)	103
Processzusok kommunikációja, vészhelyzetek, kölcsönös kizárási	103
Konkurens és kooperatív processzusok	103
Kritikus szekciók és megvalósítási módszereik	103
Altatás és ébresztés	103
Sorompók	103
Összegzés	104
Adatbázisok - 1.)	105
Az egyed-kapcsolat (ER) diagram	105
A relációs adatmodell	106
Az EK diagram leképezése relációs modellre	106
Funkcionális függőség	107
Normálformák	107
Összegzés	108
Adatbázisok - 2.)	109
Az SQL adatbázisnyelv	109
Relációsémák definiálásai, megszorítások típusai és létrehozásuk	109
Adatmanipulációs lehetőségek és lekérdezések	109
Összegzés	110

Digitális képfeldolgozás - 1.)	111
Simítás/szűrés képtérben	111
Élek detektálása	111
Összegzés	112
Digitális képfeldolgozás - 2.)	113
Alakreprezentáció	113
Határ- és régió alapú alakleíró jellemzők	113
Fourier leírás	113
Összegzés	114
Programozás alapjai - 1.)	115
Algoritmusok vezérlési szerkezetei	115
Összegzés	118
Programozás alapjai - 2.)	119
Elemi adattípusok	119
Összetett adattípusok	121
Összegzés	124
Programozás I, II - 1.)	126
Objektum orientált paradigma	126
Az absztrakt adattípus, az osztály	126
Az egységbázárás, információ elrejtése	127
Az öröklődés	127
Újrafelhasználás és polimorfizmus	128
Összegzés	130
Programozás I, II - 2.)	131
Objektumok életciklusa	131
Dinamikus, lokális és statikus objektumok létrehozása	131
A statikus adattagok, metódusok	131
Operáció és operator overloading	131
Kivételkezelés	131
Összegzés	132
Programozás I, II - 3.)	133
Java és C++ programok fordítása és futtatása	133
Parancssori paraméterek, fordítási opciók, nagyobb projektek fordítása	133
Absztrakt-, interfész- és generikus osztályok, virtuális eljárások	133
A virtuális eljárások megvalósítása, szerepe, használata	133
Összegzés	134
Programozási nyelvek - 1.)	135
A programozási nyelvek csoportosítása (paradigmák)	135
Összegzés	137

Rendszerfejlesztés I - 1.)	139
Szoftverfejlesztési folyamat	139
Szoftverfejlesztési folyamat modelljei	140
Összegzés	143
Rendszerfejlesztés I - 2.)	146
Projektmenedzsment	146
Költségbecslés	148
Szoftvermérés	149
Összegzés	150
Számítógép-hálózatok - 1.)	153
Számítógép-hálózati architektúrák, szabványosítók	153
Összegzés	157
Számítógép-hálózatok - 2.)	160
Kiemelt fontosságú kommunikációs protokollok	160
Összegzés	162
Számítógép architektúra - 1.)	163
Neumann-elvű gép egységei	163
CPU	163
Korszerű számítógépek tervezési elvei	163
RISC és CISC architektúrák és jellemzőik	163
Összefoglalás	164
Számítógép architektúra - 2.)	165
Számítógép perifériák	165
Telekommunikációs berendezések	165

Algoritmusok és adatszerkezetek I - 1.)

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

Algoritmusnak nevezünk 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

Adatszerkezet: adatok tárolására és szervezésére szolgáló módszer, amely lehetővé teszi a hozzáférést és módosításokat.

Futási idő: egy bizonyos bemenetre a végrehajtott (gépfüggetlen) alapműveletek vagy "lépések" száma.

Időigény: Egy algoritmus időigénye $T(n)$, ha az algoritmus tetszőleges n méretű inputon $T(n)$ időben megáll.

Mohó algoritmus

A mohó algoritmus alapelve az, hogy minden adott lépésben az optimálisnak látszó választást teszi meg. Habár ez egy elég egyszerű megközelítésnek tűnik, nem minden problémára adható ilyen megoldás, viszont ha mégis, akkor az elég hatékony. Részproblémára bontáskor az a cél, hogy a mohó választás egyetlen részproblémát eredményezzen, amelynek optimális megoldásából következik az eredeti probléma optimális megoldása.

Példa: adott egy hátizsák kapacitása, és n darab tárgy, mindegyik értékkel és súlyval megadva. Mekkora a legnagyobb összérték, amit a hátizsákba tehetünk? (Közismertebb nevén ez a **hátizsák-probléma** specifikációja, speciális esete az, ha a tárgyak feldarabolhatóak, ebben az esetben már **töredékes hátizsák problémáról beszélünk**).

Bemenet:

- S (a hátizsák kapacitása),
- n darab tárgy, aminek a súlyát rendre $S_i, i \in \{0, 1, \dots, n\}$, értékét pedig $E_i, i \in \{0, 1, \dots, n\}$ jelöli.

Kimenet:

- Mi a legnagyobb $\sum E$, ami legfeljebb S kapacitásba belefér?

A hátizsák probléma egyik mohó megoldása az lehet, hogy az összes tárgy közül fogjuk a legnagyobb értékkel rendelkezőt, ami még befér a táskába, és beletesszük. Ebben az esetben nem vesszük figyelembe azt, hogy a táskába több tárgyat is bele kell helyeznünk, hanem csak egyszerűen egyesével vizsgáljuk, hogy éppen akkor melyik tárgy éri a legtöbbet, aztán ezt addig ismételjük rekurzívan, amíg a táska meg nem telik, vagy nincs olyan (feladatspecifikációtól függően) tárgy, ami beleférne a táskába. Így, hogy a legnagyobbat választottuk ki, lényegében a rekurzív hívások lehetséges számát a lehető legkevesebbre csökkentjük: a mohó részproblémára osztásánál ugyanis arra törekszünk, hogy egy, vagy legalábbis a legkevesebb részt kelljen megoldanunk az uralkodás alatt.

Ezzel ellentétben mondjuk a globális maximum keresés nem feltétlenül megoldható mohó algoritmussal. Maximumnak azt az inflexiós pontot (a függvény előjelet vált) tekintjük ugye egy függvényen, ahol az értékek növekedés után csökkenni kezdenek. A mohó algoritmus az első ilyen kritériumnak megfelelő pontot globális maximumnak tekintheti, ugyanis nincs körülötte más, ennél nagyobb pont. Ettől függetlenül tudjuk, hogy egy lokális maximum érték nem feltétlenül globális => tehát a mohó algoritmus globális maximum keresésére nem feltétlenül produkál jó megoldást.

Oszd-meg-és-uralkodj (D&C)

Az oszd-meg-és-uralkodj (divide-and-conquer) fogalom a programozásban egy problémamegoldó megközelítést takar. Eszerint a beérkező feladatot felosztjuk kisebb **diszjunkt** részekre (**felosztás**), majd azokat rekurzívan megoldjuk. Ha a részfeladatok elég kicsik, akkor azokat nem osztjuk tovább kisebb részekre, rögtön megoldjuk (**uralkodás**) őket, és végül a részeredményeket összevonjuk az eredeti feladat megoldásává (**összevonás**).

Példa: adott egy ábrázolt függvény, keressünk csúcsot rajta! (Felező-csúcskereső algoritmus)

Bemenet:

- Egy $f(x)$ függvény vagy számsorozat, amin csúcsot keresünk

Kimenet:

- A függvény egy pontja, ahol csúcs található

A problémát először a felosztással kezdjük: nem kell az egész függvényt vizsgálnunk, vegyük csak a felét (mivel logikus, ha a függvény egy szakaszán találunk csúcsot, akkor nyilván az a függvényben is csúcs lesz). Ezután vesszük a szakasznak a közepét, és vizsgáljuk annak a környezetét: ha a középső érték kisebb, mint a balról mellette lévő, akkor folytatjuk a keresést a középtől balra, ellenkező esetben pedig jobbra keresünk tovább.

Elvileg minden divide-and-conquer algoritmust meg lehet valósítani iteratívan is és rekurzívan is. A rekurzió előnyei:

- olvashatóbb a kód
- intuitív (önmagát megmagyarázó) megvalósítás

Hátrányai:

- memória allokáció
- nehezebb debuggolni

Dinamikus programozás (DP)

Példa: adott n típusú pénzérme (amelyek tetszőlegesen akár többször is felhasználhatóak), a legkevesebbet felhasználva hogyan fizethető ki F forint (ha nem fizethető ki pontosan, a kimenet legyen -1, **pénz felváltási feladat**)?

Ennek a feladatnak van több megoldása is, amit a korábban ismertetett módszerekkel tudunk megközelíteni:

Mohó megvalósítás:

$P1=1, P2=5, P3=10, P4=20, P5=25, P6=50$

$F=40$

Egyszerű algoritmus kimenete: 25+10+5

(Nem helyes, ugyanis 2x felhasználva a 20-as pénzérmét megkapjuk a 40-et, és az egy optimális)

Nyers erő:

Felsorolunk minden létező kombinációt, és veszünk egyet abból, ami optimális. Helyes megoldást produkál, viszont az időigénye nagyon lassú: $O(F^n)$.

Részproblémákra osztás:

$P_1=1, P_2=5, P_3=6$

$F=9$

Párhuzamosan egyesével feltételezzük, hogy "ha bármelyik érmét kiválasztjuk, akkor a hátralévő részösszeget legkevesebb mennyi érméből tudjuk kirakni?". Ez formálisan (a +1 azért van, mertazzal jelöljük, hogy már egy érmét kiválasztottunk):

$$\minPenz(9) = \min \begin{aligned} & \minPenz(8) + 1 \\ & \minPenz(4) + 1 \\ & \minPenz(3) + 1 \end{aligned}$$

Ez az egész elég ismerős lehet, mivel eléggé hasonlít az oszd-meg-és-uralkodj módszerhez, viszont a lényeges különböző az, hogy az itt található rész problémák **nem diszjunktak**, tehát közös részproblémák fordulhatnak elő. Az elképzelés az, hogy ezek közül a már megoldottakat memorizáljuk, és ha még egyszer kell használni, felidézzük. A DP minden egyes részfeladatot, és annak részfeladatit (és így tovább) pontosan egyszer old meg, **az eredményt egy táblázatban tárolja**, és ezáltal elkerüli az ismételt számítást (futási idő: $O(F^n)$).

Rendező algoritmusok

Bemenet: Egy n számból álló tömb,

Kimenet: A bemenő tömb elemeinek olyan a_1, a_2, \dots, a_n permutációja (sorbarendezése), amelyre igaz, hogy $a_1 \leq a_2 \leq \dots \leq a_n$

Beszúró rendezés: Ennél a rendezésnél fogjuk a bemeneti tömböt, a második elemtől végig iterálunk rajta. Létrehozunk egy i index változót, és megnézzük, hogy az azon az indexen szereplő szám nagyobb-e, mint a beszúrni kívánt kulcs. Ha igen, akkor eggyel jobbra shifteljük a jelenlegi értéket, és csökkentjük i -t eggyel. Ha véget ér ez a ciklus, akkor megtaláltuk a beszúrandó szám helyét, és beillesztjük. **Algoritmus:**

```

for  $j \leftarrow 2$  to  $hossz[A]$ 
  do kulcs  $\leftarrow A[j]$ 
     $\triangleleft A[j]$  beszúrása az  $A[1 \dots j - 1]$  rendezett sorozatba.
     $i \leftarrow j - 1$ 
    while  $i > 0$  és  $A[i] > \text{kulcs}$ 
      do  $A[i + 1] \leftarrow A[i]$ 
         $i \leftarrow i - 1$ 
     $A[i + 1] \leftarrow \text{kulcs}$ 
  
```

Példa: BeszuroRendez([5, 8, 2, 9, 3])

I.) [5, **8**, 2, 9, 3]

=> először a 8-ast vizsgáljuk: mivel 5 nem nagyobb, mint 8, ezért nincs változás, nézzük a következő számot

II.) [5, 8, **2**, 9, 3]

=> nézzük a kettest: a kettesnél nagyobb a 8-as tehát a 8-as eggyel jobbra shiftel.

=> a következő szám amit nézünk az az 5-ös, az is hasonló okok miatt kerül át.

=> mivel az összes nála kisebb számot megvizsgáltuk, az algoritmus terminál, a kapott sorozat: [2, 5, 8, 9, 3]

III.) [2, 5, 8, **9**, 3]

=> a következő indexen lévő számot vizsgáljuk, a 9-est. Mivel tőle balra nincs nála nagyobb szám, ezért az algoritmus nem változtat semmin, terminál pár lépés után.

IV.) [2, 5, 8, 9, **3**]

=> végül vizsgáljuk a 3-ast. Ameddig a tömbben tőle balra nagyobb számot találunk a tömbben, addig azokat eggyel jobbra shifteljük, majd amikor elérünk az első nála kisebb számhoz (jelen esetben a ketteshez), akkor az után beszúrjuk az elemet.

=> a végső tömb pedig: [2, 3, 5, 8, 9]

Az algoritmus futási ideje $O(n^2)$ (ez itt egy ordó), ugyanis legrosszabb esetben egy elemünk sincs a helyén, és minden elemet össze kell hasonlítani nagyából minden elemmel (ergo n elemet n -szer kell összehasonlítani => n^2).

Összefésűlő rendezés: egy D&C megoldás: az n elemű rendezendő tömböt felosztjuk két részre, azokat rekurzívan összefésűlő rendezéssel rendezzük, majd a két részsorozatot összefésülik.
Algoritmus (jó csúnya, de azért tessék):

ÖSSZEFÉSÜL(A, p, q, r)

```

1   $n_1 \leftarrow q - p + 1$ 
2   $n_2 \leftarrow r - q$ 
3  az  $L[1 \dots n_1 + 1]$  és  $R[1 \dots n_2 + 1]$  tömbök létrehozása
4  for  $i \leftarrow 1$  to  $n_1$ 
5    do  $L[i] \leftarrow A[p + i - 1]$ 
6  for  $j \leftarrow 1$  to  $n_2$ 
7    do  $R[j] \leftarrow A[q + j]$ 
8   $L[n_1 + 1] \leftarrow \infty$ 
9   $R[n_2 + 1] \leftarrow \infty$ 
10  $i \leftarrow 1$ 
11  $j \leftarrow 1$ 
12 for  $k \leftarrow p$  to  $r$ 
13   do if  $L[i] \leq R[j]$ 
14     then  $A[k] \leftarrow L[i]$ 
15        $i \leftarrow i + 1$ 
16     else  $A[k] \leftarrow R[j]$ 
17        $j \leftarrow j + 1$ 
```



Példa: OsszefesuloRendez([5, 8, 2, 9, 3])

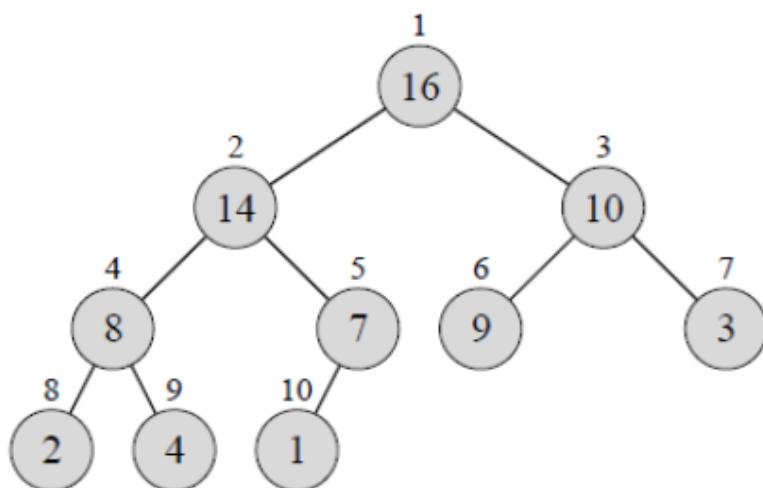
- I.) Először levesszük az első két elemet (5, 8), azokat sorbarendezzük, majd visszatesszük.
- II.) Ezután vesszük a következő két elemet (2, 9), amik szintén nem változnak.
- III.) Ezt követően vesszük a két rendezett részt, és azokat rendezzük => [2, 5, 8, 9, 3]
- IV.) Mivel már csak a 3-as nincs a helyén, ezért összefésüljük az előző, rendezett résszel, és így kijön a [2, 3, 5, 8, 9] rendezett sor.

Futásidje $\Theta(n * \log(n))$ (az ott egy theta betű a gyengébbek kedvéért, mint jómagam). Ez egy sokkal lassabban növő függvény, mint az előző $O(n^2)$. Tegyük fel, hogy $O(n)$ időt töltünk a bemenet nagyából két részre osztásával, rekurzívan rendezzük az összes darabot, majd $\Theta(n)$ idő alatt ezeket a rendezett töredékeket összekombináljuk.

Ebből az következik, hogy ha folytatjuk a D&C megközelítést, az összes leosztás együtt lineáris időben megtörténik, mivel ahogy növekszik az osztandó részek száma, úgy csökken az osztott részek mérete.

A teljes szükséges időt megkapjuk úgy, hogy ha megszorozzuk a leosztási szakaszok költségét a leosztási szakaszok számával. Mivel ugye minden alkalommal felezzük a tömböt, ebből jön a $\log_2 n$ leosztási szakasz, és minden leosztási szakasz költsége n . Így jön ki az $n * \log(n)$.

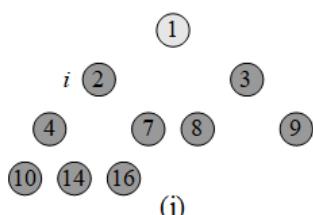
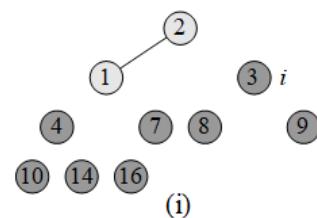
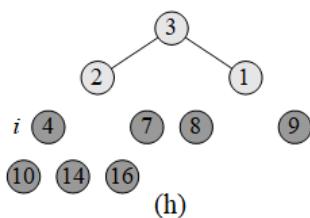
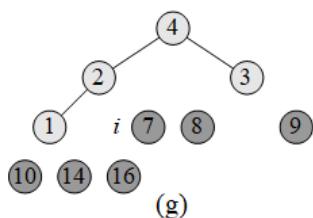
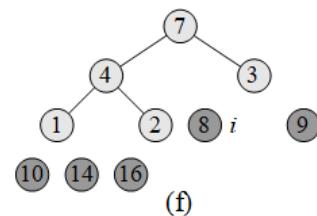
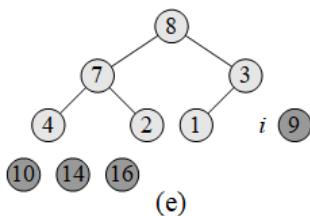
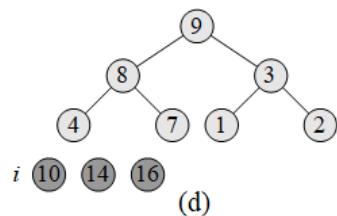
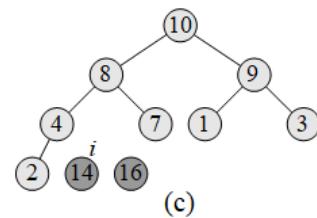
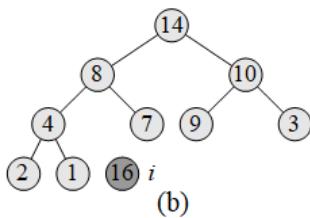
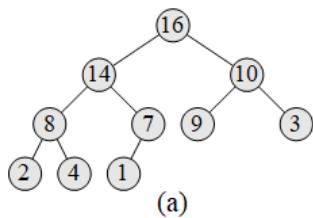
Kupacrendezés: Az alábbi képen látható, hogy a kupac valójában egy bináris fa, amely minden szintjén teljesen kitöltött, kivéve a legalacsonyabb szintet, ahol balról jobbra haladva egy adott csúcsig vannak elemek. Leveleit balról jobbra indexeljünk, és így reprezentálhatunk egy tömböt.



Az algoritmus első lépésében építünk egy maximum kupacot az inputból - ekkor a legnagyobb elem maga a gyökérelem lesz. Kicseréljük ezt a kupac utolsó elemével, és meghívjuk az eljárást az eggyel kisebb méretű kupacra. Ha sérül a korábban említett kupactulajdonság, akkor állítsuk helyre. Ezeket a lépéseket addig ismétljük, amíg a kupac mérete nem lesz 1. Ez a rendezés nem stabil, mivel helyben rendez. Futásidje $O(n\log(n))$.

Na de hogyan is állítsuk helyre a kupactulajdonságot: összehasonlítjuk a gyökér és a két gyereke értékét, ha a gyökér maximum, nincs további teendőnk. Ha viszont valamelyik gyerek értéke nagyobb, akkor cseréljük meg a gyökér és az ő értékét, majd rekurzívan állítsuk helyre a hozzá tartozó részfát.

Példa:

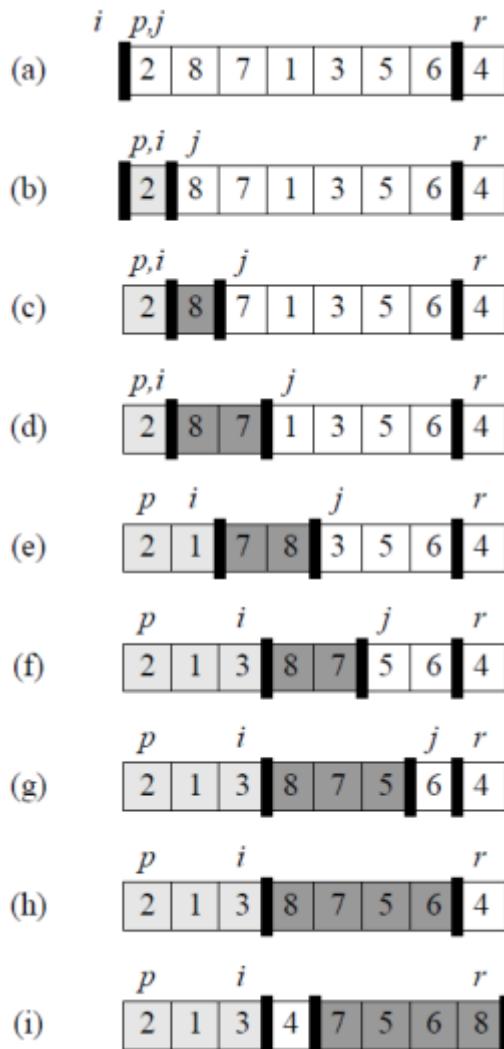


<i>A</i>	[1 2 3 4 7 8 9 10 14 16]
----------	--

(k)

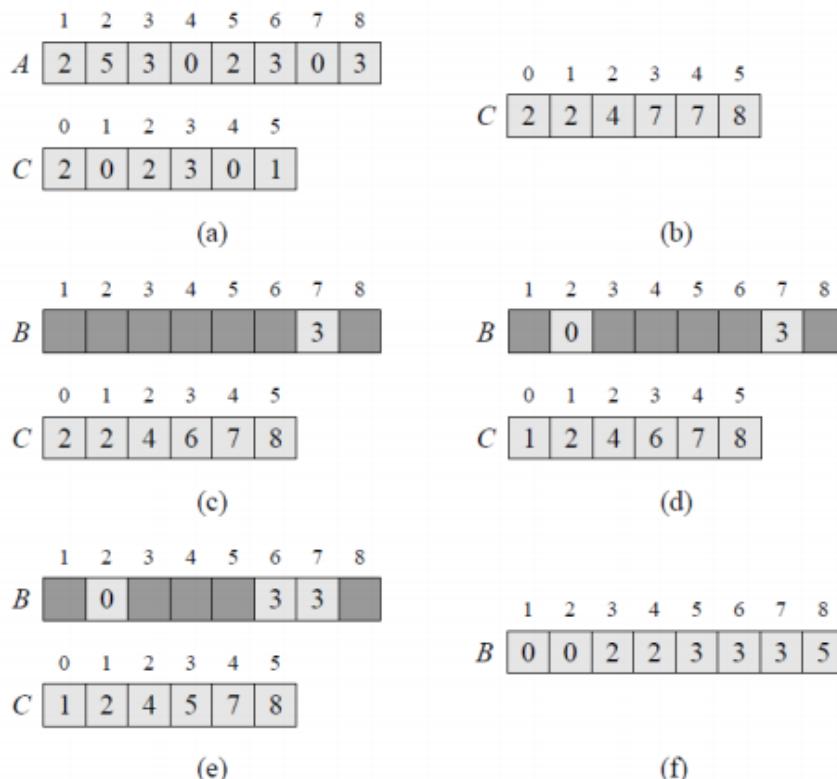
Gyorsrendezés: Szintén egy D&C elven alapuló rendezés. Az input tömböt felosztjuk 2 részre egy K kulcs segítségével úgy, hogy az első rész tartalmazza a K -nál kisebb elemeket, a másik tömb pedig a nála nagyobbakat. A kulccsal egyenlő értékek kerülhetnek bármelyik tömbbe, csak ne minden kettőbe egyszerre. Ezt követően a kapott résztömbököt gyorsrendezés rekurzív hívásával rendezzük, és a helyben rendezés miatt az összevonás lépére nincs szükség. Időigénye ($O(n^2)$) legrosszabb esetben, átlagosan $O(n \log(n))$. Tárigénye $O(\log(n))$.

Példa:



Leszámláló rendezés: Akkor használjuk, ha a tömbben legfeljebb k -félé érték szerepel. Első lépésként létrehozunk egy k elemű tömböt, és számoljuk bele össze, hogy melyik elemből mennyi van. Ezután módosítsuk úgy az értékeket, hogy minden tömbelem az őt megelőző elemek összegét tartalmazza. Végezetül az input minden elemét a helyére teszünk úgy, hogy visszafelé végig iterálunk a tömbön, és a segédben található indexre tesszük az elemet, majd csökkentjük a számlálóját. Időigénye $O(n + k)$ - ebből következik, hogy csak akkor érdemes használni, ha $k=O(n)$, mert akkor a futásidő $O(n)$. Stabil, külső rendezés.

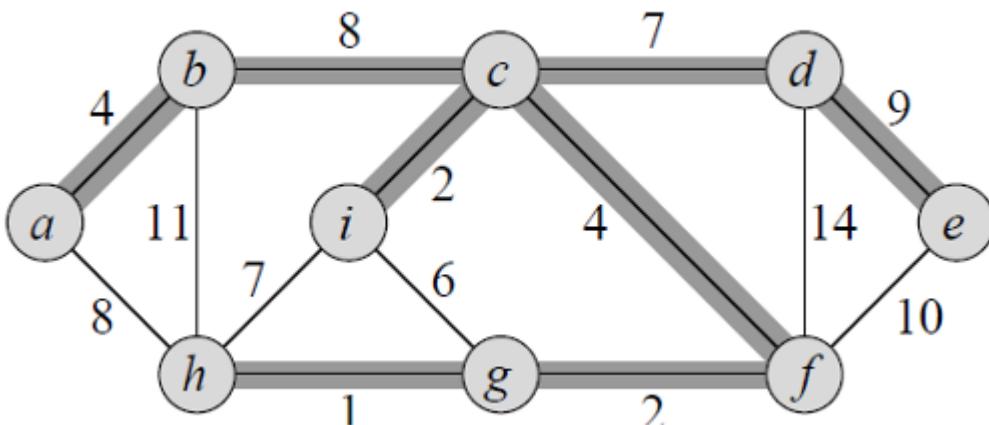
Példa:



8.2. ábra. A LESZÁMLÁLÓ-RENDEZÉS működése egy olyan $A[1..8]$ bemeneti tömbön, amelynek elemei $k = 5$ -nél nem nagyobb nemnegatív egészek. (a) Az A tömb és a C segédtömb a 4. sor után. (b) A C tömb a 7. sor után. (c)–(e) A B kimeneti tömb és a C segédtömb a 9–11. sorokban levő ciklus első, második, illetve harmadik iterációs lépése után. A B tömbnek csak a világos elemei kaptak értéket. (f) A végső, rendezett B kimeneti tömb.

Gráfalgoritmusok

Minimális feszítőfa probléma: bemenete egy összefüggő, irányíthatlan, súlyozott $G=(V, E)$ gráf, ahol V a csúcsok halmaza, E pedig az éleké, és egy $w(u, v)$ súly, ami az (u, v) él költségét fejezi ki. Ebből egy olyan feszítőfát (minden csúcsot érintő, összefüggő, körmentes élhalmaz) kellene képezni, amiben az élek költségeinek összege minimális.



Ezt el tudjuk érni például a **Kruskal algoritmussal**, ami a következő **mohó** elven alapul: minimális fák erdejét (csoportját) tároljuk. Kezdetben minden pont egy külön fa, majd szépen lassan azokat összekötögetjük, így csökken majd a fák száma az erdőben. minden lépésben a legkisebb, két fát összekötő élet húzzuk be (ezáltal egyesítjük őket egy favá). Az algoritmus véges lépésben terminál, és eredményül egy minden V csúcsot érintő, körmentes, összefüggő, minimális költségű élhalmazt kapunk $O(E \log(E))$ idő alatt, ahol E a bemeneti gráf éleinek száma.

Ezen felül választhatjuk még a szintén **mohó**, **Prím algoritmust** is, amiben egy tetszőleges gyökérpontból kiindulva fát szerkesztünk úgy, hogy az iteráció minden lépésében egy új csúcsot kötünk be, az aktuális pozícióból legkisebb költséggel rendelkező élen keresztül.

Adott csúcsból induló legrövidebb utak problémája: itt a bemenet egy irányított, súlyozott $G=(V, E)$ gráf, és egy s kezdőcsúcs. A súly szintén a $w(u, v)$ által van meghatározva. Kimenetben meg kellene adni minden V csúcshoz vezető legrövidebb utat s -ből indulva.

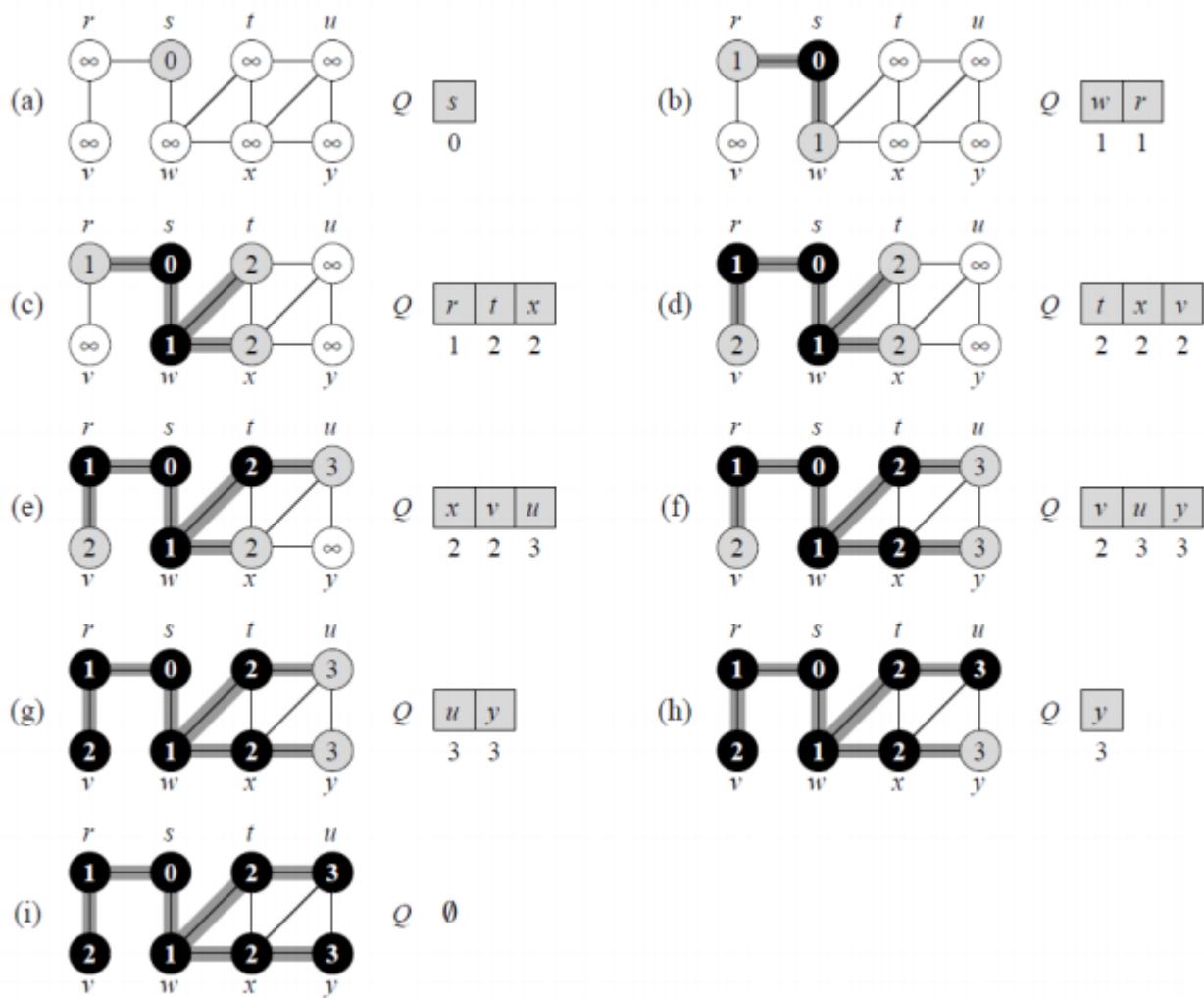
Erre megoldást jelenthet **Dijkstra algoritmusa**, ami azokat a csúcsokat tárolja, amihez már megtalálta a legrövidebb utat. minden lépésben bővíti az elérte csúcsok halmazát úgy, hogy a legrövidebb úttal bíró csúcsot választja. Ez is egy **mohó** algoritmus, aminek futásidje $O(E \log(V))$. Mivel a legrövidebb út részütje is legrövidebb út, ezért az algoritmus helyes (egy $s \Rightarrow p_1 \Rightarrow p_2 \Rightarrow p_3 \Rightarrow p_4$ út megadja az s csúcsból indulva a p_1, p_2, p_3 és p_4 pontokba vezető legrövidebb utakat is).

A súlyozott gráfok algoritmusainál érdemes vizsgálni, hogy hogyan viselkednek körök és negatív élsúlyok esetén. A Dijkstra a körökkel még boldogul, viszont a negatív összsúlyú körök már problémát jelentenek. Továbbá negatív élsúlyok esetén sem ad az algoritmus helyes megoldást.

A Dijkstra hibáira megoldást nyújthat a **Bellman-Ford algoritmus**, ami már negatív élsúlyok mellett is működik, valamint észleli, ha a kezdőcsúcsból elérhető negatív kör. Itt összesen $V-1$ iterációban próbálunk minden élen javítani $O(VE)$ idő alatt.

Szélességi keresés gráfokban: járjuk be az összes csúcsot, ami egy s kezdőcsúcsból elérhető, miközben kiszámoljuk a távolságukat s-től (távolság = legkevesebb élt tartalmazó út észáma). Bemenetben érkezik egy irányított **VAGY** irányítatlan gráf, és annak egy s csúcsa, kimenetben egy szótárat kell megadnunk, ami tartalmazza az s-ből elérhető csúcsokat és azok távolságát.

A bejárás pillanatnyi állapotát a csúcsok fehér, szürke, illetve fekete színezésével tartja számon (sor adatszerkezetet felhasználva). Kezdetben minden csúcs fehér és később szürkére vagy feketére változhat. Egy csúcs elérté válik, amikor először rátalálunk a keresés során és ezután a színe nem lehet fehér. Egy fekete csúcs összes szomszédja elért csúcs, a szürke csúcsoknak lehetnek fehér szomszédjaik, ezek alkotják a már elért és még felfedezetlen csúcsok közti határt.



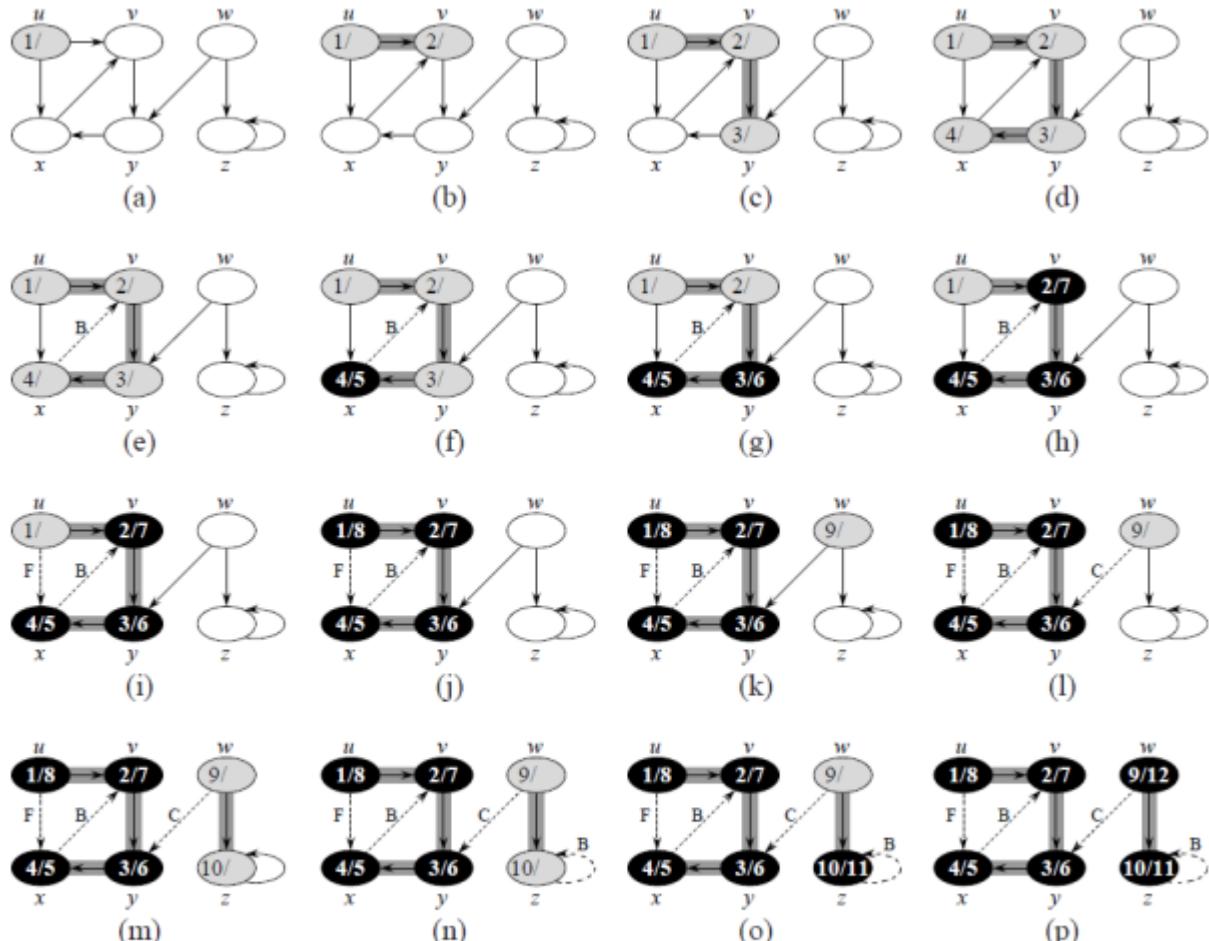
Ezáltal kaphatunk egy a példában is jól látható ún. **szélességi fát**, ami függ a kimenő élek bejárási sorrendjétől, de a távolságok egyértelműek.

Mélységi keresés gráfokban: akkor használjuk, amikor egy megoldást megtalálni elégséges, nem szükséges az összes, vagy hogy a megtalált megoldás optimális legyen (pl van-e út u és v csúcs között?). Bemenete ugyanaz, mint a szélességi keresésnek. Eredményül egy úgynevezett **mélységi feszítőerdőt** (MFE) kapunk, amiben az éleket az alábbi módon osztályozzuk:

Erősen összefüggő komponensek: a gráfban azok a maximális csúcshalmazok, amin belül bármelyik csúcsból el lehet jutni bármely másikba.

- Faél: (u,v) faél, ha bekerül a MFE élei közé, azaz $\pi(v) = u$.
- Visszaél: (u,v) visszaél, ha u leszármazottja v-nek a MFE-ben.
- Előreél: (u,v) előreél, ha v leszármazottja u-nak a MFE-ben és nem faél.
- Keresztél: minden más esetben (u,v) keresztél

A szélességi kereséshez hasonlóan a csúcsok állapotát itt is színekkel különböztetjük meg. Kezdetben minden csúcs fehér, amikor egy csúcsot elérünk akkor szürkére színezzük, és befeketítjük, ha elhagytuk, azaz amikor a szomszédsági listájának minden elemét megvizsgáltuk.



22.4. ábra. MK algoritmus működésének szemléltetése egy irányított gráfon. Az algoritmusban az adott pontig megvizsgált éleket megvastagítottuk, ha fa élek, egyébként szaggatott vonallal jelöljük ezeket. Azokat az éleket, amelyek nem tartoznak fához, B, C vagy F címkékkel látta el, annak megfelelően, hogy visszamatató, kereszt vagy előremutató élek. A csúcsokban feltüntettük az elérési/elhagyási időpontokat.

Összegzés

Mohó algoritmus:

- hártság probléma (töredékes hártság probléma, ismétléses hártság probléma)
- az algoritmus minden aktuális lépésben az optimálisnak tűnő lépést teszi meg
- nem minden problémára adható mohó algoritmus, ha mégis, az hatékony
- felosztásnál igyekszünk 1 részproblémát kapni, amiből megkapjuk az eredményt

Oszd-meg-és-uralkodj (D&C):

- csúcskeresés
- felosztás - uralkodás - összevonás
- több diszjunkt, az eredetihez hasonló részre osztjuk a problémát
- azokat rekurzívan megoldjuk
- a kapott részeredményeket összevonjuk az eredeti probléma megoldásává

Dinamikus programozás (DP):

- pénzváltás probléma
- "nyers erő" megoldások (minden lehetséges megoldás felsorolása, megfelelő kiválasztása)
- hasonló mint a D&C, annyival, hogy itt előfordulhatnak azonos részproblémák
- ezeknek a problémáknak a megoldásait tároljuk, ha újra előfordulnak, a tárolt megoldást újra felhasználjuk

Rendező algoritmusok:

- bemenete egy n elemű tömb, kimenete a tömb elemeivel növekvő sorrendben
- BESZÚRÓ RENDEZÉS
 - végig iterálunk a bemeneti tömbben a második elemtől kezdve
 - megvizsgáljuk a tőle balra lévő elemeket
 - ameddig tőle nagyobbakat találunk, azokat eggyel jobbra toljuk
 - amint találunk egy kisebbet, beszűrjuk az elemet
 - időigénye $O(n^2)$
- ÖSSZEFÉSÜLŐ RENDEZÉS
 - felezgetjük az inputot, és a kapott töredékeket sorbarendezzük
 - majd ezeket a töredékeket egymással összekombináljuk és rendezzük
 - időigénye $\Theta(n \log(n))$
- KUPACRENDEZÉS
 - kupac: egy bináris fa, amely minden szintjén teljesen kitöltött, kivéve a legalacsonyabb szintet, ahol balról jobbra haladva egy adott csúcsig vannak elemek
 - kupaccal (heap) reprezentáljuk a bemeneti tömböt,
 - először maximum kupacot képzünk,
 - kivesszük a gyökerelementet, és betesszük a rendezett tömb elejére,
 - kupactulajdonság helyreállítása (ha a gyökerelemnél bármelyik gyereke nagyobb, akkor azokat kicseréljük, részfáikat hasonlóképpen, rekurzívan rendezzük), ha kell
 - időigénye ($O(n^2)$) legrosszabb esetben, átlagosan $O(n \log(n))$
- GYORSRENDEZÉS
 - D&C elven alapul,
 - egy kulcs segítségével két részre osztjuk a bemeneti tömböt, egyik részben a kulcsnál nagyobb, a másikban a nála kisebb értékeket tároljuk,
 - a kapott résztömböket rekurzívan gyorsrendezzük
 - eredményül egy rendezett tömböt kapunk
 - időigénye ($O(n^2)$) legrosszabb esetben, átlagosan $O(n \log(n))$

- LESZÁMOLÓ RENDEZÉS
 - bemeneti tömbben legfeljebb k-féle érték,
 - k elemű tömb létrehozása, megszámoljuk, hogy az értékek hányszor szerepelnek,
 - minden tömbeemet az őt megelőző elemek összegére módosítunk
 - majd a tömböt hátulról populáljuk úgy, hogy a segédben található indexre tesszük az elemet
 - időigénye $O(n + k)$, $k=O(n)$ esetén ez $O(n)$

Gráfalgoritmusok:

- SZÉLESSÉGI KERESÉS
 - bemenete egy irányított vagy irányítatlan, súlyozott gráf és egy s csúcs
 - kimenetben egy szótárat kapunk, amiben minden s csúcsból elérhető csúcshoz a hozzá tartozó távolságot rendeljük
 - fehér, szürke, fekete színezés (kezdetben minden fehér, s szürke; első lépésben s szomszédai legyenek szürkék, s maga meg fekete, majd ezt ismételjük addig, ameddig van legalább egy szürke csúcs)
 - szélességi fa (csúcsok bejárási sorrendjétől függ, de jól látható a távolság)
 - időigénye $O(V + E)$
- MÉLYSÉGI KERESÉS
 - bemenete egy irányított vagy irányítatlan, súlyozott gráf és egy s csúcs
 - kimenete egy **mélységi feszítőerdő**
 - faél, visszaél, előreél, keresztél
 - erősen összefüggő komponensek
 - időigénye $O(V + E)$
- MINIMÁLIS FESZÍTŐFÁK
 - bemenete egy összefüggő, irányítatlan, súlyozott gráf
 - kimenete egy olyan T feszítőfa (minden csúcsot érintő, összefüggő, körmentes élhalmaz), amelyben az élek költségének összege minimális
 - Kruskal algoritmus (minimális fák tárolása, kezdetben minden pont egy fa, minden alkalommal a legkisebb, két fát összekötő élt húzzuk be - **mohó**, $O(E \log(E))$ idő)
 - Prím algoritmus (egyetlen fa, minden lépésben új csúcsot kötünk be a fába a legolcsóbb élen keresztül - **mohó**, $O(E \log(V))$ idő, ha $V < E$)
 - Sűrű gráfok esetén (sok az él benne) Prím erősebb, egyébként Kruskal, egyszerűbb adatszerkezet miatt
- LEGRÖVIDEBB UTAK
 - bemenete egy irányított, súlyozott gráf és egy s kezdőcsúcs
 - kimenete az s csúcsból induló összes legrövidebb út a gráf többi csúcsába
 - Díjsktra algoritmus (minden lépésben bővíjtük az elért csúcsok halmazát, ha egy csúcsba már felvettünk élet, de találunk "olcsóbbat", akkor azt lecseréljük. Mindig a legrövidebb úttal bíró csúcsot választja, nem ad jó megoldást negatív körök vagy élsúlyok esetén - **mohó**, $O(E \log(V))$ idő)
 - Bellman-Ford algoritmus (negatív élsúlyokra is működik, negatív köröket észreveszi. Egy iterációban próbálunk minden élen javítani, V-1 iterációra van szükség)

Algoritmusok és adatszerkezetek I - 2.)

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

Elemi adatszerkezetek

Halmaz: szerepe ugyanúgy alapvető a számítástudományban, mint a matematikában. Viszont még a matematikában egy halmazt időben változatlannak tekintünk, addig egy olyan halmaz, amelyen egy algoritmus végez műveleteket, bővülhet, zsugorodhat, vagy feldolgozás során másként módosulhat. Az ilyen halmazokat **dinamikusnak** nevezzük. A dinamikus halmazokon értelmezett műveletek 2 csoportba sorolhatók: lekérdező műveletek (információt szolgáltatnak a halmazról), és módosító műveletek (megváltoztatják a halmazt). A leggyakoribb műveletek a következők:

- **Keres(H, k):** lekérdező művelet, mely egy H halmazban k kulcsú elemet keres, és egy olyan x elem mutatóját adja vissza, amelyre $\text{kulcs}[x] = k$, illetve NIL-t ad vissza, ha nincs ilyen elem a H halmazban
- **Beszür(H, x):** módosító művelet, mely bővíti a H halmazt az x által mutatott elemmel. Általában feltesszük, hogy előzőleg az x elem minden mező értékét kaptak, amelyre az adott implementációban szükség van
- **Töröl(H, x):** módosító művelet, amely eltávolítja a H halmazból az x által mutatott elemet. (Fontos, hogy x egy elemre mutató pointer, tehát nem kulcs szerinti törlésről van szó.)
- **Minimum(H) / Maximum(H):** lekérdező művelet, mely a teljesen rendezett H halmazban szereplő legkisebb / legnagyobb kulcsértékű elem mutatóját adja vissza.
- **Következő(H, x) / Előző(H, x):** lekérdező művelet, amely annak a H halmazbeli elemnek a mutatóját adja vissza, amelynek a kulcsértéke közvetlenül az x elem kulcsértéke után / előtt helyezkedik el a teljes rendezés szerint, illetve NIL-t ad vissza, ha x a legnagyobb / legkisebb kulcsú elem a halmazban.

Verem: felfogható úgy is, mint egy speciális halmaz, aminek felül van írva a Beszür és a Töröl művelete annak érdekében, hogy a **LIFO (Last In First Out)** tulajdonságát megtartsa. Hasonlóan működik mint egy földbe ásott verem: csak a tetején lehet dolgokat betenni, és kivenni.

Egy legfeljebb n elemet tartalmazó verem megvalósítható egy V[1..n] tömbbel. A tömb $\text{tető}[V]$ attribútuma a verembe legutoljára betett elemének az indexe. V[1] a legalsó elem, V[tető[V]] pedig a legfelső. Ha $\text{tető}[V] = 0$, a verem nem tartalmaz elemet, tehát üres. Ha üres veremből próbálunk elemet kivenni, akkor alulcsordul, ha $\text{tető}[V]$ értéke meghaladja n-t, akkor túlcsordul. A verem műveletei jobbról láthatók az ábrán (mindhárom művelet O(1) idejű)

ÜRES-VEREM(V)

```

1 if  $\text{tető}[V] = 0$ 
2 then return IGAZ
3 else return HAMIS

```

VEREMBE(V, x)

```

1  $\text{tető}[V] \leftarrow \text{tető}[V] + 1$ 
2  $V[\text{tető}[V]] \leftarrow x$ 

```

VEREMBŐL(V)

```

1 if ÜRES-VEREM( $V$ )
2 then error „alulcsordulás”
3 else  $\text{tető}[V] \leftarrow \text{tető}[V] - 1$ 
4 return  $V[\text{tető}[V] + 1]$ 

```

Sor: A sorokon értelmezett Beszür művelet neve Sorba, a Töröl művelet pedig a Sorból, amelyeknek a veremhez hasonlóan nincs argumentuma. A sor első elemét fejnek, az utolsó elemét végének nevezzük. Amikor egy elemet behelyezünk a sorba, akkor az a végére kerül, amikor egy elemet kiveszünk, akkor azt az elejéről vesszük ki.

Egy legfeljebb $n - 1$ elemű sort megvalósíthatunk egy $S[1..n]$ tömb felhasználásával. A $fej[S]$ attribútum a sor első elemét indexeli, a $vége[S]$ pedig annak a helynek az indexe, ahova az új elem majd kerül. A sor ciklikus elrendezésű, tehát a tömb végére érve, az n -edik hely után az első hely következik. Ha $fej[S] = vége[S]$, akkor a sor üres. Ha üres sorból próbálunk elemet kivenni az alulcsordulást okoz, ha pedig $fej[S] = vége[S] + 1$, akkor a sor tele van és új elem beszúrása túlcsorduláshoz vezet. A Sorba és Sorból műveletek $O(1)$ idejűek.

Láncolt lista: A láncolt lista olyan adatszerkezet, melyben az objektumok lineáris sorrendben követik egymást. A tömbökönél a lineáris sorrendet a tömbindexek határozzák meg, ezzel szemben a láncolt listákban mutatók valósítják meg az elemek lineáris elrendezettségét: a lista minden objektuma tartalmaz egy mutatót, ami a következő elemre mutat. A láncolt lista egyszerű és rugalmas eszköz a dinamikus halmazok ábrázolására. A lista egy attribútuma az első elemre mutató $fej[L]$ pointer, ha $fej[L] = NIL$, akkor a lista üres.

LISTÁBAN-KERES(L, k)

- 1 $x \leftarrow fej[L]$
- 2 **while** $x \neq NIL$ és $kulcs[x] \neq k$
- 3 **do** $x \leftarrow köv[x]$
- 4 **return** x

$\Theta(n)$ a futásidő legrosszabb esetben.

Számos változata van, egy lista lehet egyszeresen vagy kétszeresen láncolt, rendezett vagy nem rendezett, és lehet ciklikus vagy nem ciklikus. Ha a lista kétszeresen láncolt, akkor az elemek a kulcs mező mellett két mutatót is tartalmaznak, ezek a köv. és az előző (egyszeresen láncolt lista esetében csak köv. van), értelemszerűen az első a következő elemre mutat, a második pedig az előző elemre. Ha egy lista rendezett, akkor a mutatók által meghatározott sorrend megegyezik az elemekben tárolt kulcsmezők növekvő sorrendjével: a legkisebb kulcsérték a lista fejében található, a legnagyobb pedig a végében. Ha egy lista rendezetlen, akkor az elemeinek sorrendje tetszőleges lehet. A ciklikus listát az jellemzi, hogy a fej előző mutatója a lista végére, a vége köv. pointere pedig a fejre mutat. A feltüntetett műveletek kétszeresen láncolt, rendezetlen listákra vonatkoznak:

SORBA(S, x)

- 1 $S[vége[S]] \leftarrow x$
- 2 **if** $vége[S] = hossz[S]$
- 3 **then** $vége[S] \leftarrow 1$
- 4 **else** $vége[S] \leftarrow vége[S] + 1$

SORBÓL(S)

- 1 $x \leftarrow S[fej[S]]$
- 2 **if** $fej[S] = hossz[S]$
- 3 **then** $fej[S] \leftarrow 1$
- 4 **else** $fej[S] \leftarrow fej[S] + 1$
- 5 **return** x

LISTÁBA-BESZÚR(L, x)

```

1  köv[x] ← fej[L]
2  if fej[L] ≠ NIL
3    then előző[fej[L]] ← x
4  fej[L] ← x
5  előző[x] ← NIL

```

Futási ideje O(1).

LISTÁBÓL-TÖRÖL(L, x)

```

1  if előző[x] ≠ NIL
2    then köv[e előző[x]] ← köv[x]
3    else fej[L] ← köv[x]
4  if köv[x] ≠ NIL
5    then előző[köv[x]] ← előző[x]

```

Futási ideje O(1), ha azonban kulcs szerint akarunk törölni, akkor ehhez a legrosszabb esetben $\Theta(n)$ idő szükséges, mivel előbb végre kell hajtani a Listában-keres eljárást.

Bináris keresőfák

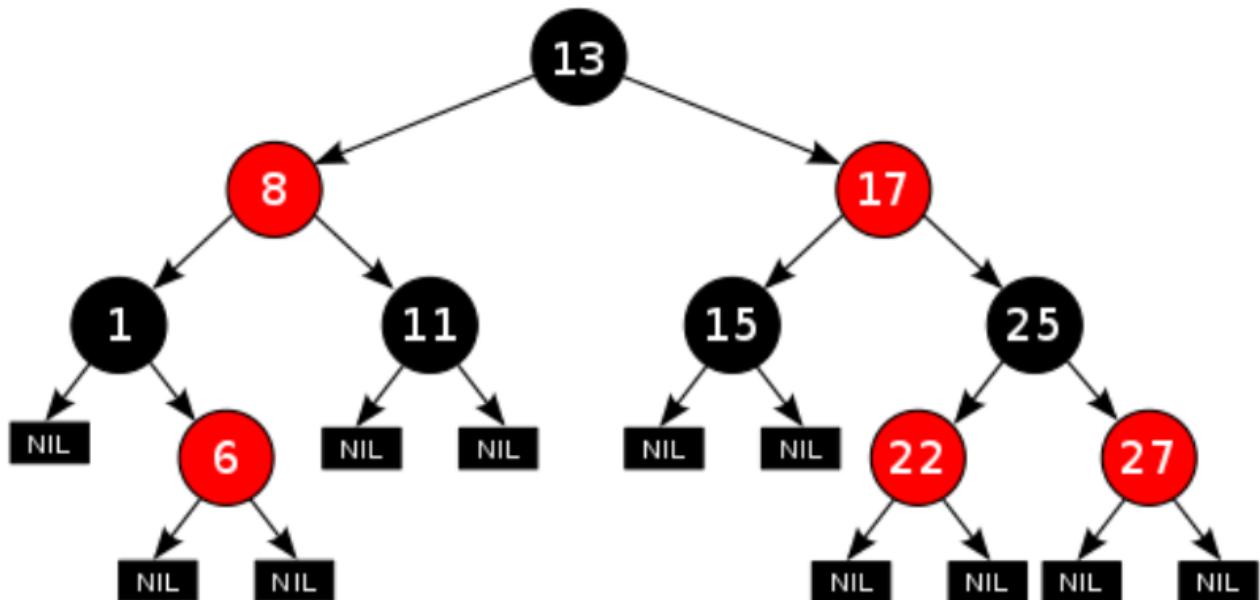
A bináris keresőfák elsődleges elképzelése az, hogy az összes művelet (keres, beszúr, törlő és társai) mind legyenek $O(\log(n))$ idejűek. Ezen felül még annyit kell tudnia egy bináris fának, hogy minden csúcsának maximum 2 gyereke van, valamint a bal gyerek értéke minden esetben kisebb a szülő értékénél, a jobb gyereké pedig nagyobb.

A fa magassága (h) legrosszabb esetben könnyen belátható, hogy akár n is lehet, ahol n a beszúrt elemek száma. Mivel a fa szerkezete erősen függ a beszúrás sorrendjétől, ezért ugyanazokat az elemeket számtalan keresőfában tárolhatjuk.

Kiegyensúlyozott keresőfáról akkor beszélünk, ha a annak magassága a lehető legkisebb, így az azon végrehajtott műveletek időigénye csak a magasságtól függ. Erre egy példa a piros-fekete fa (ami megközelítőleg kiegyensúlyozott), aminek magassága legfeljebb $2 \log_2(n+1)$. Ezeket a következőképpen alkotjuk:

- minden csúcs színe piros, vagy fekete,
- a gyökér fekete
- minden levél (NIL) színe fekete
- minden piros csúcsnak minden gyereke fekete
- bármely csúcsból bármely levélre vezető úton ugyanannyi fekete csúcs van

Látható, hogy ezek a tulajdonságok módosítások során fent kell tartani az 5 piros-fekete tulajdonságot, ami egész könnyen sérülhet. Ez nem jelent akkora bajt, ugyanis $O(\log(n))$ idő alatt helyre tudjuk állítani.



Egyfajta kiegyensúlyozott piros-fekete fa

Visszatérve a bináris keresőfákra

Keresésnél a gyökérkcsúsból indulva a következőképpen járunk el:

- megnézzük, hogy az aktuális elem a keresett-e: ha igen, visszatérünk
 - ha nem, akkor megnézzük, hogy a kulcs értéke kisebb vagy nagyobb a jelenleginél
 - ha kisebb, akkor a bal gyereken hajtjuk végre ezt, ha nagyobb, akkor nyilván a balon
- Könnyen belátható, hogy ez simán egy konstans idejű futást eredményez, amit $O(h)$ -val jelölünk, ahol h a keresőfa magassága.

Min/Max esetében még könnyebb az eljárás, ugyanis ameddig találunk bal/jobb gyereket, addig mászunk lefelé a fában, és amikor eljutunk oda, ahol már nincs kisebb / nagyobb elem (ergo nincs már gyerek a megfelelő irányban), visszatérünk az értékkel. Elég triviális, hogy a futásidő $O(h)$.

A **Következő/Előző** esetében már kicsit cselesebben kell eljarnunk, ugyanis tudjuk, hogy az adott csúcs jobb gyerekének még lehetnek bal gyerekei, ami a kettejük által bezárt intervallumba eshetnek, ezért például x -re következő úgy találjuk meg, hogy megkeressük magát x -et, és ha van jobb gyereke, akkor a belőle képzett részfában minimumot keresünk, és visszaadjuk azt, mint következőt. Abban az esetben, ha nincs jobb gyereke x -nek akkor addig fejtjük vissza a szülőket, ameddig nem találjuk meg az első olyan számot, ami nagyobb, mint x . Az előző esetében ugyanígy járunk el, csak tükrözni kell. A futásidő itt is $O(h)$.

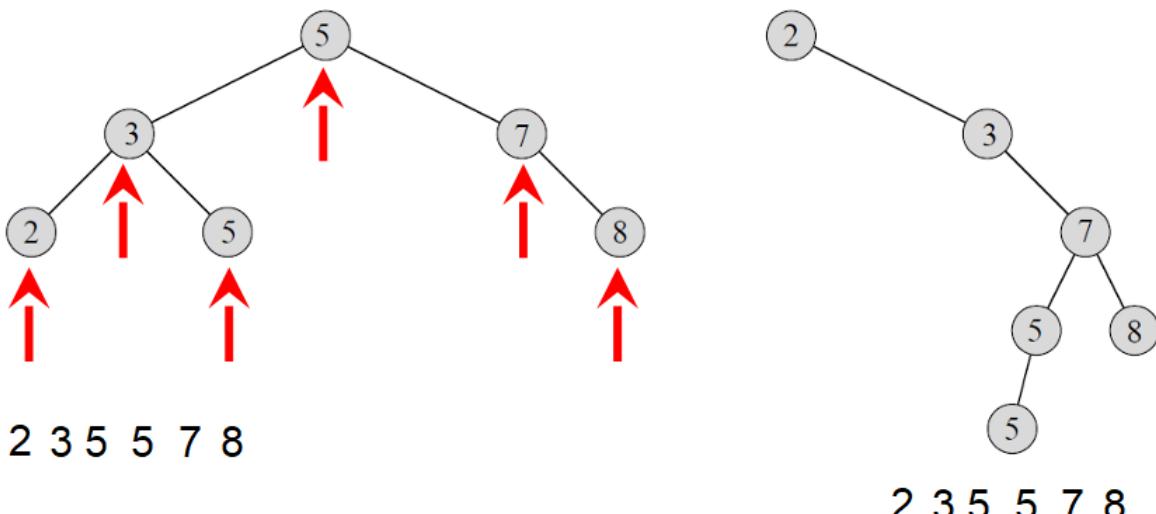
Beszúrásnál csak meg kell találjuk azt a helyet, ahova beszúrható maga az elem: tehát ha az aktuális csúcsnál nagyobb az érték, akkor megnézzük, hogy a jobb gyereke szabad-e, ha igen, akkor beszúrjuk, ha nem, akkor a jobb gyerekén folytatjuk ezt az eljárást. Itt is látszik azért, hogy ez csak a fa magasságától fog függni, tehát szintén $O(h)$ a futásidő.

A **Törlés** talán egy kicsit trükkösebb, mert itt gondolom nem kell mondani se, hogy simán előfordulhat olyan eset, ahol olyan csúcsokat törlünk, aminek vannak gyerekei. Itt tehát azt kell csináljuk, hogy ha a töröld csúcs:

- levél: egyszerűen kitöröljük a fából
- egy gyerekkel rendelkezik: egyszerűen összekötjük a szülőt a törölt csúcs gyerekével
- két gyerekkel rendelkezik: a bal gyerekének legnagyobb gyerekét tesszük a helyére

Értelemszerűen ha két gyerekkel rendelkező csúcs esetén nincs a bal gyereknek jobb leszármazottja, akkor egyszerűen magát a csúcsot adjuk vissza.

Inorder fabejárás során x elemből először bejárjuk annak bal gyerekéből képzett részfát, majd a jobb gyerekéből képzettet is.



Hasító táblázatok

A hasító táblázat hatékony adatszerkezet szótárak megvalósítására. Bár egy elem hasító táblázatban való keresésének ideje ugyanolyan hosszú lehet, mint láncolt listák esetében – a legrosszabb esetben $\Theta(n)$ –, a gyakorlatban a hasítás nagyon jól működik. Ésszerű feltételek mellett egy elem hasító táblázatban való keresésének várható ideje $O(1)$.

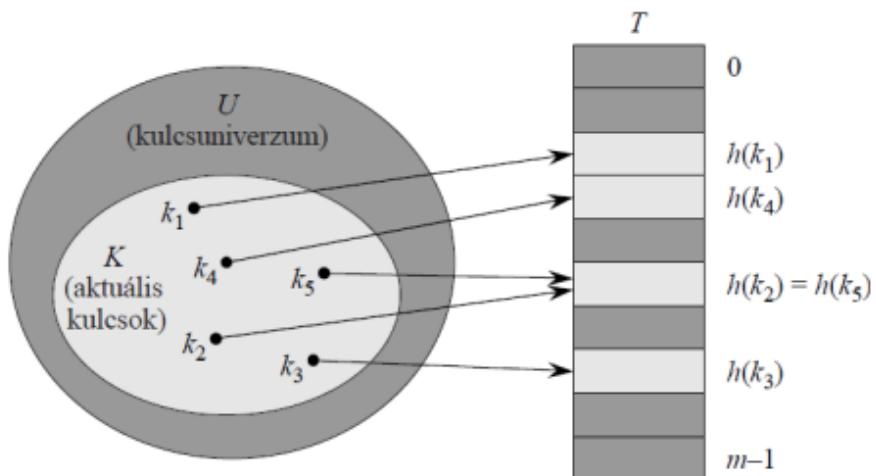
A közvetlen címzés egyik problémája, hogy ha az U kulcsuniverzum nagy, akkor egy $|U|$ méretű T táblázat tárolása nem célszerű, vagy egyenesen lehetetlen. Fennáll továbbá, hogy az aktuálisan tárolt elemek K halmaza az U-hoz képest olyan kicsi is lehet, hogy a T által elfoglalt hely legnagyobb része kihasználatlan.

Amikor a szótárban tárolt kulcsok K halmaza sokkal kisebb, mint a lehetséges kulcsok U univerzuma, akkor a hasító táblázatnak jóval kevesebb memóriára van szüksége, mint a közvetlen címzésű táblázatnak. A memóriaigény leszorítható $\Theta(|K|)$ -ra, ugyanakkor egy elem hasító táblában való keresésének ideje továbbra is $O(1)$ marad.

Közvetlen címzés esetében egy k kulcsú elem a k-adik résben tárolódik. A hasítás alkalmazása esetén ez az elem a $h(k)$ helyre kerül, vagyis egy h hasító függvényt használunk arra, hogy a rést a k kulcsból meghatározzuk. Itt h a kulcsok U univerzumát képezi le a $T[0..m - 1]$ hasító táblázat réseire: $h : U \rightarrow \{0, 1, \dots, m - 1\}$. Úgy mondjuk, hogy $h(k)$ a k kulcs hasított értéke. A hasító

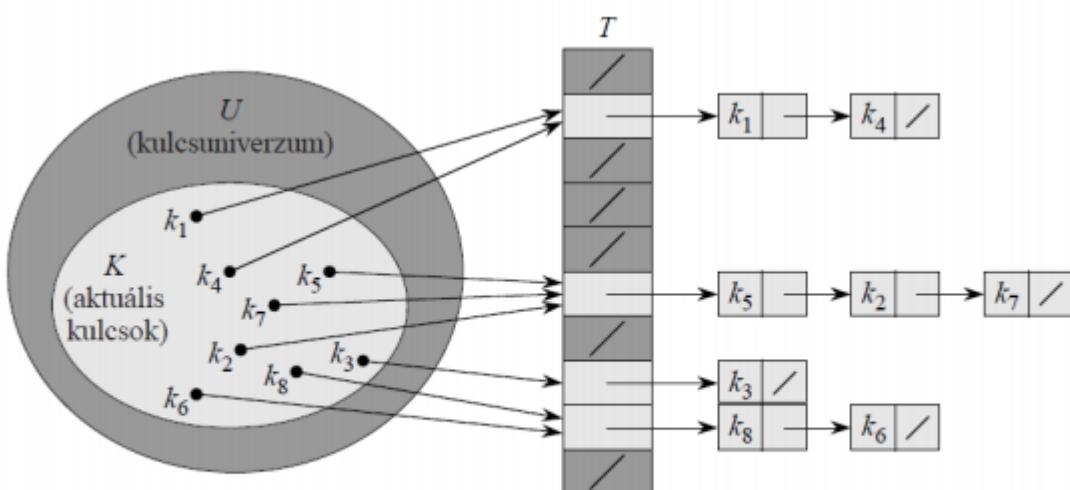
függvény célja az, hogy csökkentse a szükséges tömbindexek tartományát. $|U|$ számú index helyett most csak m -re van szükség. A memóriaigény is ennek megfelelően csökken.

Megtörténhet, hogy két kulcs ugyanarra a résre képződik le. Ezt a helyzetet ütközésnek nevezik. Szerencsére vannak hatékony módszerek az ütközések nyomán keletkező konfliktusok feloldására.



11.2. ábra. A h hasító függvény felhasználása a kulcsoknak egy hasító táblázat réseire való leképezésére. A k_4 és k_5 kulcsok ugyanarra a résre képződnek le, tehát ütköznek.

Ütközésfeloldás láncolással: az ugyanarra a résre képződő elemeket összefogjuk egy láncolt listába. A j -edik rés egy mutatót tartalmaz, mely a j címre leképződő elemek listájának fejére mutat. Amennyiben ilyen elemek nincsenek, a j -edik rés a NIL-t tartalmazza.

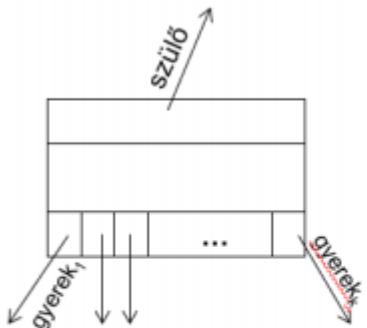


11.3. ábra. Ütközésfeloldás láncolással. A hasító táblázat egy $T[j]$ rése azoknak a kulcsoknak a láncolt listáját tartalmazza, melyek hasított értéke pontosan j . Például $h(k_1) = h(k_4)$ és $h(k_5) = h(k_2) = h(k_7)$.

Gráfok és fák számítógépes reprezentációja

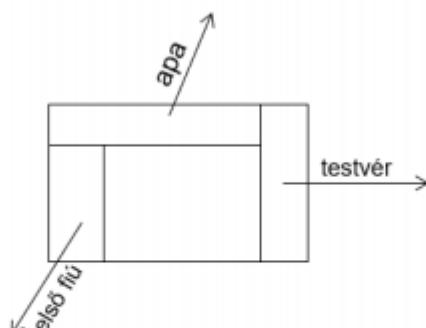
Fák reprezentációjánál a csúcsokat és éleket kell reprezentáljuk (lényegében maga a fa egy mutató a gyökérre), és erre több lehetőségünk is van:

gyerek éllista



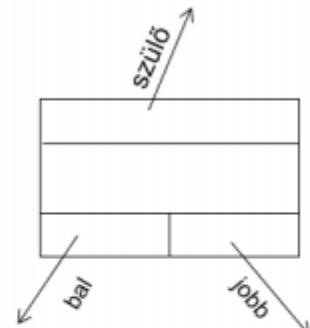
```
class Node{
    Object key;
    Node parent;
    List<Node> children;
}
```

első fiú, apa, testvér



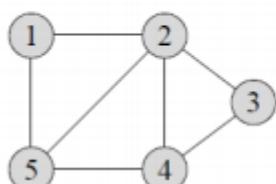
```
class Node{
    Object key;
    Node parent;
    Node first_child;
    Node brother;
}
```

bináris fa

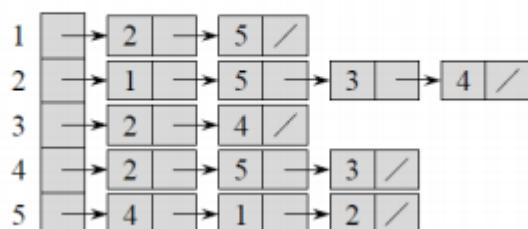


```
class Node{
    Object key;
    Node parent;
    Node left;
    Node right;
}
```

Gráfok ábrázolásánál meg kell különböztetnünk az irányított és irányítatlan gráfok ábrázolását. Mindkét esetben szomszédsági listával, vagy csúcsmátrixszal tudjuk őket reprezentálni, a különbség az, hogy más szabályok szerint vesszük fel az értékeket.



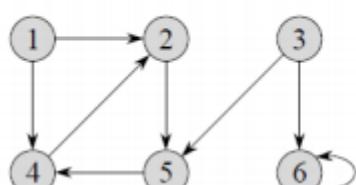
Irányítatlan gráf



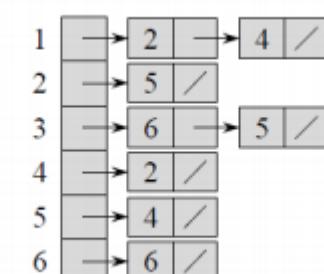
Szomszédsági lista

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

Csúcsmátrix



Irányított gráf



Szomszédsági listája

1	2	3	4	5	6
1	0	1	0	1	0
2	0	0	0	0	1
3	0	0	0	0	1
4	0	1	0	0	0
5	0	0	0	1	0
6	0	0	0	0	1

Csúcsmátrix

Összegzés

Elemi adatszerkezetek:

- HALMAZ
 - alapvető szerep, matematikával ellentétben a számítástechnikai halmazok időben változhatnak (**dinamikus halmazok**)
 - lekérdező műveletek (keres, minimum, maximum, előző, következő)
 - módosító műveletek (beszúr, töröl)
- VEREM
 - egy n elemű verem egy n elemű tömbbel megvalósítható
 - LIFO (Last In First Out) szerkezet
 - Beszúr = Verembe, Töröl = Veremből
 - beszúráskor a sor végére kerül az elem, kivételkor pedig a legutóbb beszúrt elemet vesszük ki
 - olyan, mint egy földbe ásott verem: csak a tetején lehet kivenni és betenni
- SOR
 - FIFO (First In First Out) szerkezet
 - Beszúr = Sorba, Töröl = Sorból
 - beszúráskor a sor végére kerül az elem, kivételkor pedig azt az elemet vesszük ki, ami a legrégebb óta a sorban van már (tehát beérkezési sorrendben)
- LÁNCOLT LISTA
 - egyes elemei tartalmaznak referenciát a rájuk következő elemre (opcionálisan tartalmazhatnak az előzőre is)
 - több változata létezik:
 - **egyszeresen láncolt**: csak a következő elemre tartalmaz hivatkozást
 - **kétszeresen láncolt**: az előző elemre is tartalmaz hivatkozást
 - **rendezett**: valamilyen prioritás szerint rendezve van
 - **nem rendezett**: az elemek között nincs prioritási sorrend
 - **ciklikus**: az első elem előzője az utolsó, aminek következője az első
 - **nem ciklikus**: az első elem előzője és az utolsó elem következője is NIL

Bináris keresőfák

- minden tárgyalt művelet futásideje legyen $O(\log(n))$
- **bináris keresőfa tulajdonság**: x tetszőleges csúcsra ha vesszük T_x bal/jobb oldali részfáját, akkor T_x minden eleme kisebb/nagyobb (vagy egyenlő), mint x.
- a fa magassága legrosszabb esetben lehet akár n is. Hogy a műveletek futásigénye ne az elemek számától, hanem a fa magasságától függjön, próbálunk **kiegyensúlyozott keresőfákkal** dolgozni (piros-fekete fa pl.).
- **Keresés**: gyökérelemtől kezdve megvizsgáljuk, hogy megtaláltuk-e a keresett elemet, és ha nem, akkor a keresett elem és az aktuális csúcs viszonyától függően tovább keresünk.
- **Min/Max**: ameddig az adott csúcsnak van bal/jobb gyereke, addig azon keresztül másunk lefelé a fában, egészen addig, ameddig meg nem találjuk azt az elemet, aminek nincs.
- **Következő/Előző**: a következő kifejezhető úgy, mintha x jobb gyerekére szerkesztett részfában minimumot keresnékn (az előző nyilván a bal részfában maximum lesz).
- **Beszúrás**: addig lépegetünk jobbra-balra a fában a beszúrandó érték és a jelenlegi csúcs relációjától függően, ameddig meg nem találjuk azt a pontot, ahol az érték pl nagyobb, mint az aktuális csúcs, és annak nincs jobb oldali gyereke

- **Törlés:** ha az eltávolítandó elemek nincs gyereke, azonnal töröljük; ha egy gyereke van, akkor kivágjuk (tehát töröljük, és a gyerekét csatoljuk a törlött elem szülőjéhez); ha két gyereke van, akkor az ő bal részfájában található, megelőző elemet tesszük a helyére
- ezeknél a műveleteknél figyelni kell, hogy a bináris-keresőfa tulajdonság megmaradjon
- **Inorder fabejárás** során ki tudjuk venni növekvő sorrendben a fából az elemeket, először a gyökér bal részfáját járjuk be rekurzívan, majd ezt követően a jobbat is.

Hasító táblázatok

- hatékony adatszerkezet szótárak megvalósítására
- keresés ideje olyan lehet, mint a láncolt listáké (legrosszabb esetben $\Theta(n)$, átlagosan $O(1)$)
- ha egy U kulcsuniverzum mérete nagy, akkor egy $|U|$ méretű táblázat tárolása nem célravezető (akár lehetetlen)
- közvetlen címzés esetében egy k kulcsú elem a k -adik résben tárolódik
- hasítás alkalmazása esetén egy elem a $h(k)$ helyre kerül, vagyis egy h hasító függvényt használunk arra, hogy a rést a k kulcsból meghatározzuk.
- a h hasítófüggvény a kulcsok U univerzumát képezi le a $T[0..m - 1]$ hasító táblázat réseire: $h : U \rightarrow \{0, 1, \dots, m - 1\}$ ($h(k)$ a k kulcs hasított értéke).
- a hasító függvény célja az, hogy csökkentse a szükséges tömbindexek tartományát. A memóriaigény is ennek megfelelően csökken.

Gráfok és fák számítógépes reprezentációja

- **FÁK ÁBRÁZOLÁSA:**
 - gyerek éllista: tartalmazza magát a kulcsot, a szülőt, és a leszármazottait listában
 - első fiú, apa, testvér: tartalmazza a kulcsot, a szülőt, egy gyereket, és egy testvérét
 - bináris fa: tartalmazza a szülőt, a kulcsot, a jobb és a bal gyerekeket
- **GRÁFOK ÁBRÁZOLÁSA:**
 - irányított és irányítatlan gráfokat ugyanúgy ábrázolunk, csak más szabályok szerint
 - szomszédsági lista: ha két csúcs között van él, akkor azt felvesszük (irányítatlan); ha az adott csúcsba vezet él, akkor azt felvesszük (irányított)
 - csúcsmátrix esetén egy főátlóra szimmetrikus mátrixot kapunk (ugyanis ha x csúcsból vezet él y csúcsba, akkor irányítatlan esetén y csúcsból is vezet x csúcsba), vagy irányított gráfoknál ha a egy csúcsból vezet a másikba, akkor beállítjuk a mátrix megfelelő indexét 1-re

Bonyolultságelmélet - 1.)

Hatókony visszavezetés. Nemdeterminizmus. A P és NP osztályok. NP-teljes problémák.

Hatókony visszavezetés

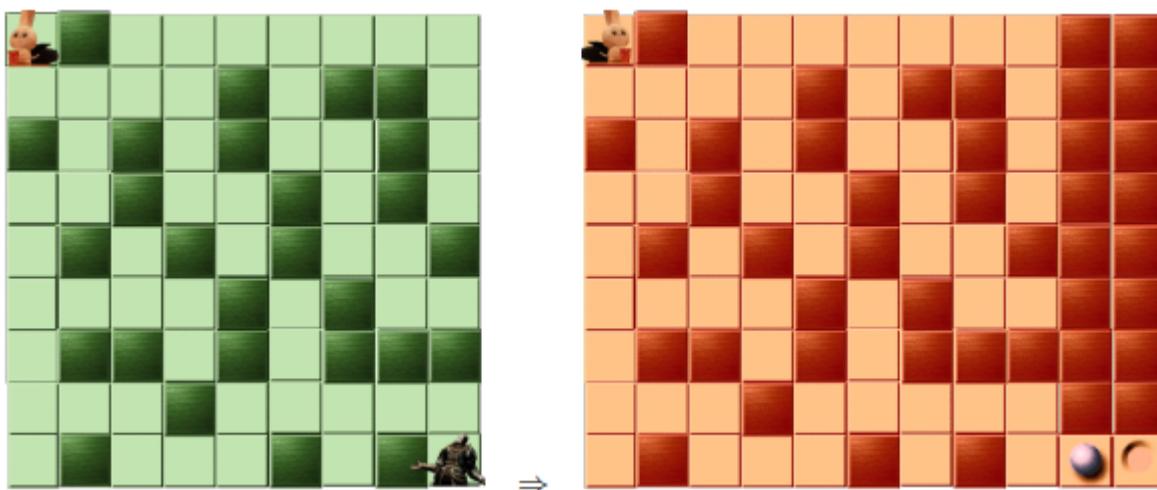
Definíció szerint az A eldöntési probléma **hatékonyan visszavezethető** (vagy „polinomidőben visszavezethető”) a B eldöntési problémára, jelben $A \leq_p B$, ha van olyan f **polinomidőben** kiszámítható függvény, amely A inputjaiból B inputjait készítí **választartó** módon.

Ebből következtethetünk arra, hogy ha A polinomidőben visszavezethető B-re, és tudjuk, hogy maga B polinomidőben eldönthető, akkor A is eldönthető polinomidőben, valamint azt is tudjuk, hogy ha A-ra nincs polinomidejű algoritmus, akkor B-re sincs. **Bizonyítsuk is ezt be!**

Abból, hogy **két polinomidejű algoritmus kompozíciója is polinomidejű** kikövetkeztethető, hogy ha B-t polinomidőben megoldja egy *B* algoritmus, és *f* az A-ról egy hatékony visszavezetés B-re, akkor A-t az $A(x) := B(f(x))$ algoritmus **polinomidőben eldönti (a választartás miatt)**.

Ennek példájára vegyük például a **Maze** (labirintus bal felső sarkából jussunk le a jobb alsóba mondjuk) és **Sokoban** (szintén labirintus, némely mezőkön golyók találhatók, máshol pedig lyukak, a cél az lenne, hogy az összes golyót juttassunk el egy lyukba), kezdetben egyáltalán nem hasonlónak tűnő problémákat. Első hallásra talán a Sokoban bonyolultabbnak tűnhet, viszont próbáljuk meg visszavezetni rá a Maze problémát!

Egy lehetséges megoldás lehet az, ha az alábbi képen látható módon kibővítjük a táblát (levesszük a batman-t, mellette kibővítjük a táblát 2 új sorral, aminek csak a cél melletti mezője legyen elérhető úgy, hogy a cél pozíció mellé közvetlenül helyezünk egy golyót, és a golyó másik oldalára - tehát nem arról, amerről a cél volt, hogy bele lehessen lökni - egy lyukat).

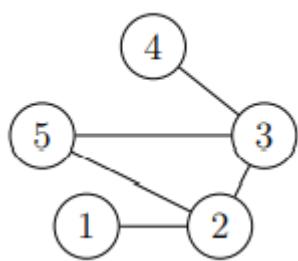


A választartás nem kérdés, ugyanis ha az első ábra a Maze egy IGEN példánya, akkor a második ábrán sem fog gondot okozni ugyanazon az úton végigmenni és belökni a golyót a helyére. Most a kérdés az, hogy ez a visszavezetés hatékony-e (azaz polinomidejű). Mivel csak levettük a batmant, és hozzáadtunk a labirintushoz két oszlopot (mindössze két mezőt), ez lineáris időben megvan, tehát **ez a Maze Sokoban visszavezetés hatékony**.

Itt most van még egy zsák érdekes és hasznos példa, ilyen például még a **Hamilton-út** (adott egy G gráf, van-e benne minden csúcsot pontosan egyszer érintő út) **visszavezetése a TSP(E)-re** ($TSP(E)$ = Traveling Salesman Problem, adott N darab város és a köztük lévő távolság, valamint egy $C \geq 0$ célérték, van-e olyan út, ami minden városon pontosan egyszer megy át, és összköltsége legfeljebb C), ahol megmutatjuk, hogy a $TSP(E)$ legalább olyan nehéz, mint a Hamilton-út.

Mivel még csak a bemenetek sem egyeznek meg, ezért első lépésként adnunk kell egy olyan inputkonverziót, ami a Hamilton-út egy példányát transzformálja át a $TSP(E)$ egy példányára, és mivel hatékony visszavezetést szeretnénk, ezért jó lenne, ha ez választató lenne (tehát ha az input Hamilton-út IGEN példányát hozza ki, akkor az konvertált inputtal meglött $TSP(E)$ is IGEN példányt produkáljon).

Ezt a legkönyebben úgy érhetjük el, hogy felírjük a G gráfhoz tartozó távolság mátrixot úgy, hogy ha **G-ben két csúcs között volt él, akkor legyen a távolságuk 1, ha nem volt, akkor legyen 2** (önmagától mindenki nyilván 0 távolságra van).



<i>d</i>	1	2	3	4	5
1	0	1	2	2	2
2	1	0	1	2	1
3	2	1	0	1	1
4	2	2	1	0	2
5	2	1	1	2	0

Mivel a generált $TSP(E)$ példányban minden távolság vagy 1, vagy 2, így az optimális körút költsége valahol N és $2N$ között lesz, ahol N a csúcsok száma.

Ha van Hamilton-út a gráfban, akkor választva egy Hamilton-utat és azon végigmenve összesen $N-1$ élen haladtunk, eddig az út költsége $N-1$, aztán a Hamilton-út végpontjából visszaugorva a kezdőpontba vagy 1, vagy 2 lesz a távolság értéke (attól függően, hogy volt-e ott él vagy sem), így ilyenkor a $TSP(E)$ példányban az optimum értéke legfeljebb $N + 1$.

Ha pedig a generált $TSP(E)$ példányban van legfeljebb $N + 1$ költségű körút, akkor abban van N él, mindenek a súlya vagy 1, vagy 2. Tehát ha legfeljebb egy olyan lépést tehet, ahol a távolság 2, és ezen felül csupa 1 súlyút, akkor az az eredeti gráfban egy Hamilton-kör (akkor van Hamilton-út is). Ha ebből kihagyjuk a 2 súlyú lépést az eredeti gráfban található Hamilton-utat kapjuk, tehát ilyenkor mindenképp van Hamilton-út G -ben.

Azt kaptuk tehát, hogy ha G -ból a fenti módszerrel előállítjuk a DG távolság mátrixot (ez könnyű átalakítás, csak végi kell menni a G szomszédsági mátrixán és az 1-esekből 1-est, a főátlóbeli értékekből 0-t, a többi 0-ból pedig 2-est csinálni), és célszámnak beállítjuk a $C = N + 1$ értéket, akkor **G-ben pontosan akkor van Hamilton-út, ha DG-ben van legfeljebb C költségű körút**.

Tehát az átalakítás hatékony és tartja a választ, azaz visszavezetés Hamilton-útról TSP(E)-re.

Itt is látható volt, hogy a hatékony visszavezetés általában 3 fő részre osztható:

1. megadunk egy olyan f átalakítást, ami az A probléma inputjaiból a B probléma inputját készíti
2. megmutatjuk, hogy ha A -nak egy IGEN példányából indulunk, akkor B -nek egy igen példányába érkezünk
3. megmutatjuk, hogy ha B -nek egy IGEN példányába érkezünk, akkor A -nak egy igen példányából kellett induljunk.

Sokszor a három közül a harmadik pont lesz a legnehezebb; általában az első két pont teljesülni szokott, de a harmadik nem mindig, ilyenkor fordulnak elő „hamis találatok” (mikor nem példányból készül igen példány, ekkor persze a konverzió **nem** visszavezetés).

Végső példa a **Párosítás** (adott egy G gráf, van-e G -ben teljes párosítás - tehát található-e benne olyan független élhalmaz, ami az összes csúcsot lefedi, de bármely két él nem rendelkezik közös csúccsal) **visszavezetése a SAT** (adott egy konjunktív normálformájú (CNF) formula, kérdés, hogy kielégíthető-e) **problémára**.

Kezdésnek tehát megint meg kell adnunk egy olyan inputkonverziót, amit a **Párosítás** inputjából (tehát egy G gráfból) elkészíti a SAT inputját (ami egy CNF) úgy, hogy az választartó legyen - ergo G -ben pont akkor legyen teljes párosítás, ha a CNF kielégíthető. Na de hogy is csináljuk ezt?

Ugye van eredetileg egy gráfunk, amiből formula kell, tehát könnyen ki lehet következtetni, hogy itt valamiből változókat kell majd csinálnunk: nem túl nagy a választék, csúcsok vagy élek lehetnek itt változók. Mivel a párosítás inkább az élekkel operál, ezért **vegyünk fel egy változót a gráf összes élére, és a CNF formulánkban állítsuk igazra azokat, amiket a párosítás során felveszünk, ellenkező esetben hamis**.

Ezzel a párosítás egyik fele megvan: bekerülnek azok az élek, amiket ilyenkor felvennénk. Igen ám, de nem kell nagyon belegondolni, hogy ez így simán IGEN példányt csinálna olyan inputokról is, amik amúgy a Párosítás NEM példányai, mivel itt még nem ellenőriztük le azt, hogy minden csúcsra pontosan egy él illeszkedik-e. Ha ezt összeéseljük az előző megállapításra csúcsonként, akkor jók vagyunk, mivel CNF-ek éselése továbbra is CNF lesz. A “pontosan egy” kijelentés azt jelenti, hogy “legalább egy” és “legfeljebb egy”: ugye ez megint szuper, mert megint maradtunk a CNF vonzáskörzetében.

Azt könnyű még felírni, hogy u csúcsra legalább egy kiválasztott él illeszkedik (összevagyoljuk a változókat, ha legalább az egyik igaz, az igazza állítja a klózt), a nehezebb rész ott jön, amikor a “legfeljebb egy” részt vizsgáljuk: ott azt kell néznünk, hogy “nem ez a kettő egyszerre, és nem az a kettő egyszerre” az összes párra. Ez “matekul” úgy néz ki, hogy $\neg(x_i \wedge x_j)$, ami DeMorgan azonossággal megfeleltető a $(\neg x_i \vee \neg x_j)$ CNF-be illő klóznak. Aggodalomra adhat okot, ugyanis látható, hogy ebből jó sok lesz - igen ám, de ez a jó sok is csak négyzetes sok, tehát polinom \Rightarrow polinomidőben előállítható.

Ha ezt a (nagy) CNF-et létrehozzuk, az pont **hatékony visszavezetés** (ezt most onnan látjuk, hogy elmagyaráztuk, mit jelent a formula és látszik, hogy pontosan akkor lesz igaz, ha a változóértékkedás olyan élhalmaz kiválasztásának felel meg, aminek a gráf minden csúcsára pontosan egy-egy eleme illeszkedik).

Nemdeterminizmus

A nemdeterminisztikus algoritmusok sajátossága az, hogy nincs meghatározva, hogy mi szerint adunk értéket a változóinknak (például a 3-színezés problémánál fogunk egy random csúcsot, és a választott 3 szín közül beszínezzük valamelyikre - itt azért erősen látszik a "random" hatás, ami mégsem teljesen random, csak "valahol el kell kezdeni" elven az ilyen problémáknál így járunk el). RAM programok esetén ez felfogható úgy, hogy egy adott változó lehet 0 is és 1 is, és ezekkel a gép párhuzamosan, két külön szalon számolat, és megnézi minden esetet.

Számos problémára adható nemdeterminisztikus algoritmus, ilyen például a **Hamilton-út** is:

- 1.) a bemeneti gráfra generálunk egy N elemű tömböt, aminek minden elemére generálunk egy $1 + \lceil \log(N) \rceil$ bites számot (ezt azért csináljuk, mert szeretnénk a gráf egy csúcsát eltárolni, és ennyi biten tudunk eltárolni egy $1\dots N$ számot). Ezt követően mondjuk egy 5 csúcsú gráfra kaphatunk egy ilyen tömböt (az egyszerűség kedvéért nem biteket használva): [1, 5, 2, 4, 3].
- 2.) Miután legeneráltuk ezt a csodát megnézzük, hogy **valóban a bemeneti gráf csúcsainak egy permutációját (sorbarendezését) kaptuk-e** - validáció
- 3.) Ellenőrizzük, hogy a két egymást követő csúcsok szomszédosak-e: ha igen, akkor tök egyszerűen találtunk egy Hamilton-utat, adjunk vissza TRUE-t, ha nem, akkor ez épp nem volt jó, és ez a szál visszaad egy FALSE-t, és értékelődik ki a többi.

Itt a lépéseket egyenként polinomidőben el tudjuk végezni, és ha ezek kompozícióját vesszük, akkor az még mindig egy valamilyen polinomot produkál, ergo ez az egész algoritmus polinomidő alatt eldönthető (ez azért fontos, mert az ilyen **nemdeterminisztikusan polinomidőben eldönthető** problémák tartoznak az NP osztályba).

A korábban említett **3-színezés (adott egy G gráf, kiszínezhetőek-e a csúccai három színnel úgy, hogy két szomszédos csúcs ne legyen azonos színű)** problémát is így tudjuk hatékonyan megoldani: létrehozunk egy N elemű tömböt, mindegyikhez rendelünk egy kétbites számot (ha 11-et sikerült generálni, akkor az alapból eldobjuk, mert az már a negyedik "szín" lenne), majd leellenőrizzük, hogy a gráf minden (i, j) élre $T[i] \neq T[j]$. Ha mindegyikre igaz, akkor true, különben false. Könnyű belátni, hogy ez lineáris időigény alatt lefut.

A P és NP osztályok

A P és az NP bonyolultsági osztályok egész közel állnak egymáshoz, egészen annyira, hogy egyesek szerint a két osztály azonos (bár erre még nincs bizonyítás).

P-ben található az összes olyan eldöntési probléma (outputja két bites - igen/nem, accept/reject), amire létezik $O(n^k)$ - POLINOM időigényű eldöntő algoritmus, valamelyen konstans k -ra. A Cobham-Edmonds tézis szerint ezeket tartjuk hatékonyan megoldható problémáknak.

NP jelöli a nemdeterminisztikus algoritmussal polinomidőben eldönthető problémák osztályát. Ebből következik, hogy P-nek részhalmaza NP, mivel egy (determinisztikus) polinomidejű algoritmust felfoghatunk úgy is, mint egy olyan nemdeterminisztikus algoritmust, ami nem generál egyszer sem nemdeterminisztikusan bitet

NP teljes problémák

Egy problémára akkor mondjuk, hogy **nehéz** egy adott bonyolultsági osztályra, ha az osztály bármely problémája visszavezethető rá. Ugyanez a probléma akkor **teljes**, ha még ő maga is benne van az adott osztályban.

SAT (Cook-tétele)

Input: egy CNF

Output: kielégíthető-e (azaz létezik-e olyan értékkombináció a változóinak, amire a formula IGAZ)?

Ez egy nagyon hosszú Touring-gépes bizonyítás, sokkal egyszerűbb magából a jegyzetből elolvasható megérteni (<https://www.inf.u-szeged.hu/~szabivan/download/bonyelm/jegyzet.pdf>)

FormSAT (általánosabb SAT)

Input: egy ítéletkalkulusbeli formula (tehát a VAGY és ÉS műveleteken kívül is megengedettek)

Output: kielégíthető-e?

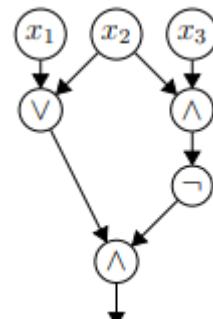
Olyan dolgok ismeretében, mint a DeMorgan azonosság, és hasonló csalafintaságok, ez könnyen, lineáris időben átalakítható CNF-fé: tehát az inputkonverzió után - mivel az ítéletkalkulus nem változik a két probléma között - a bizonyítás ugyanaz, mint a SAT esetében.

Hálózat-kielégíthetőség

Input: egy hálózat

Output: kielégíthető-e?

A képen látható, hogy egy hálózat hogyan is akar kinézni: megfigyelhetjük, hogy a tetején vannak a változók (amik 0 vagy 1 értéket vehetnek fel), aztán vannak 1 és 2 változós műveletek, amik aztán adnak valamilyen részeredményt, és a legvégén közösen, hogy maga a hálózat milyen eredményt produkált: lényegében ezek a hálózatok segítségével le tudunk képezni egy tetszőleges formulát. Mivel itt több művelet van, mint a SAT CNF-jei esetében, ezért próbáljuk meg inkább a FormSAT-ot (amire már amúgy a SAT-ot visszavezettük, és azt meg bebizonyítottuk, hogy NP-teljes) visszavezetni erre.



Itt igazából annyit csinálunk, hogy a logikán tanult ítéletkalkulusbeli azonosságokat alkalmazva a FormSAT-ba beérkező formulát átalakítjuk egy hálózattá: ha találunk egy új literált, akkor arra felvesszük a hozzá tartozó kaput, a NEGÁL-ást, VAGY-olást és ÉS-elést megtartjuk, az $(x \rightarrow y)$ implikációt kapukkal felírjuk mint $\neg(x \vee y)$, és az $(x \leftrightarrow y)$ ekvivalenciát, mint $(x \rightarrow y) \wedge (y \rightarrow x)$ - ezt nyilván tovább visszük, és ráküldjük az implikációt, amivel egy ilyen negálós csodát kapunk megint. Ebből ugye megint kijön, hogy egy beérkező FormSAT formulát simán tudunk hálózattá alakítani úgy, hogy a kettő ekvivalens legyen, szóval mivel a működésük ugyanaz (ergo mindenkorban ugyanazt fogja eredményezni pl a logikai ÉS), ezért ez egy visszavezetése a FormSAT-ra, tehát ez is NP-teljes.

3SAT (kSAT egy specifikus formája)

Input: egy olyan CNF, amelyben minden klóz összesen 3 literált tartalmaz

Output: kielégíthető-e?

Vissza fogjuk vezetni az előző, Hálózat-kielégíthetőség problémát erre, a kérdés már csak az, hogy hogyan alakítjuk át az inputot. Adott inputként egy hálózatot, amiben lehet mindenféle logikai művelet: erre adnunk kell egy konverziót a kapukra, mivel a literálokra nem kell felvennünk klózt.

- **ÉS kapu:** a CNF-be bekerül a $(\neg g \vee g_1 \vee g_2) \wedge (\neg g \vee g_2 \vee g_3) \wedge (\neg g_1 \vee \neg g_2 \vee g)$ formula
- **VAGY kapu:** $(\neg g \vee g_1 \vee g_2) \wedge (\neg g_1 \vee g \vee g) \wedge (\neg g_2 \vee g \vee g)$ kerül be
- **NEGÁLÁS kapu:** $(\neg g \vee \neg g_1 \vee \neg g_2) \wedge (g_1 \vee g \vee g)$

Ezekkel a csoda átalakításokkal tudunk kapni egy hálózatból egy vele ekvivalens CNF-et, ami nyilván választartó, tehát visszavezettük a hálózat-kielégíthetőséget a 3SAT-ra, úgyhogy ez is NP-teljes.

Módosított SAT (nem ez a hivatalos neve, csak talán így könnyebb megjegyezni)

Input: egy formula, melynek minden klózában vagy legfeljebb két literál szerepel, vagy csupa negatív literálból álló háromelemű klóz.

Output: kielégíthető-e?

Visszavezetjük erre a problémára a 3SAT-ot: Bevezetjük minden x_i változóhoz annak a kalapos változatát (\hat{x}_i), és minden értékadásban x_i és \hat{x}_i ellentétes értéket vesznek fel ($x_i \leftrightarrow \neg \hat{x}_i$ formula CNF-jével: $(x_i \vee \hat{x}_i) \wedge (\neg x_i \vee \neg \hat{x}_i)$ érjük el). A készített formulába tehát minden i-re felveszük ezt a két bináris klózt, ezek megfelelnek az új probléma input szintaxisának, továbbá, az eredeti formulában minden pozitívan előforduló x_i változót $\neg \hat{x}_i$ -re cserélünk, így ezekben a klózokban pedig csupa negatív literál szerepel, ami szintén megfelel az új probléma szintaxisának. Mivel ekvivalens átalakításokat használtunk, világos, hogy ami az eredeti 3SAT problémát kielégíti, az ezt is ki fogja => visszavezettük a 3SAT-ra => ez is NP-teljes.

Független csúcshalmaz

Input: egy G (irányítatlan) gráf és egy K szám.

Output: van-e G-ben K darab független (azaz páronként nem szomszédos) csúcs

Ahogy korábban, itt is megpróbálunk egy már bizonyítottan NP-teljes problémát erre visszavezetni: jelen esetben itt megint a 3SAT-ot választjuk. Létrehozunk a 3SAT bemenetéből egy olyan G gráfot, amely:

- csúcsai a CNF-beli literálelőfordulások (tehát pontosan 3n darab csúcs lesz)
- minden egyes klóz literáljait összekötjük éellel (tehát most n darab háromszögünk lett)
- az ellentétes literálokat tartalmazó klózokat szintén összekötjük (tehát ha az egyikben van x, egy másikban $\neg x$, a hozzájuk tartozó literálokat összekötjük)
- K célszám pedig legyen a klózok száma

Innen kapunk n darab háromszöget, ha azokon belül ki tudunk választani alakzatonként egy csúcsot úgy, hogy azok egymással ne legyenek összekötve, akkor létezik a gráfnak K elemű független csúcshalmaza. Ezt hatékonyabban úgy is megkaphatjuk, hogy ha találunk a gráfból felírt CNF-re kielégítő értékadást, akkor konkrétan megkapjuk azt is, hogy melyik pontokat kell kiválasztanunk.

Klikk

Input: egy G (irányítatlan) gráf és egy K szám.

Output: van-e G-ben K darab páronként szomszédos csúcs (azaz K elemű klikk)

Erre most a Független csúcshalmaz problémát fogjuk visszavezetni: vesszük a gráf komplementerét (ergo a komplementer gráfban akkor veszünk fel éleket, ha az adott két csúcs között az eredetiben nem volt), meg az eredeti K számot. Ez egy simán polinomidejű inputkonverzió, tehát nem csak hogy visszavezettük rá az említett problémát, hanem még hatékonyan is tettük meg azt.

Hamilton-út (mivel visszavezethető rá, ezért TSP(E) is)**Input:** egy G gráf**Output:** van-e G-ben Hamilton-út (minden csúcsot pontosan egyszer érintő út)

Visszavezetjük rá a 3SAT problémát:

3-Színezés**Input:** egy G gráf**Output:** G csúcsai kiszínezhetőek-e úgy, hogy két szomszédos csúcs ne legyen egy színű

Visszavezetjük rá a 3SAT problémát:

Hármasítás**Input:** két egyforma méretű (mondjuk diszfunkciós) halmaz, A, B és C, valamint egy $R \subseteq A \times B \times C$ reláció**Output:** van-e olyan $M \subseteq R$ részhalmaza a megengedett hármasoknak, melyben minden $A \cup B \cup C$ -beli elem pontosan egyszer van fedve?

Visszavezetjük rá a 3SAT problémát:

Pontos Lefedés Hármasokkal**Input:** egy U 3m-elemű halmaz és háromelemű részhalmazainak egy $S_1, \dots, S_n \subseteq U$ rendszere**Output:** Van-e az S_i -k között m , amiknek uniója U ?**Halmazlefedés****Input:** egy U halmaz, részhalmazainak egy $S_1, \dots, S_n \subseteq U$ rendszere és egy K szám**Output:** Van-e az S_i -k között K darab, amiknek uniója U ?**Halmazpakolás****Input:** egy U halmaz, részhalmazainak egy $S_1, \dots, S_n \subseteq U$ rendszere és egy K szám**Output:** Van-e az S_i -k között K darab páronként diszfunkciós?**Egész Értékű Programozás****Input:** egy $Ax \leq b$ egyenlőtlenség-rendszer, A-ban és b-ben egész számok szerepelnek**Output:** van-e egész koordinátájú x vektor, amely kielégíti az egyenlőtlenségeket?

Visszavezetjük rá a 3SAT problémát:

Részletösszeg**Input:** pozitív egészek egy a_1, \dots, a_k sorozata**Output:** van-e ezeknek egy olyan részhalmaza, amelynek összege épp K ?

Visszavezetjük rá a 3SAT problémát:

Hátizsák**Input:** i darab tárgy w_i súlyjal és c_i értékkal, egy W összkapacitás és egy C célérték**Output:** Van-e a tárgyaknak olyan részhalmaza, amelynek összsúlya legfeljebb W , összértéke pedig legalább C ?

Visszavezetjük rá a Részletösszeg problémát:

Összegzés

Hatókony visszavezetés

- az A eldöntési probléma **hatékonyan (polinomidőben) visszavezethető** a B eldöntési problémára ($A \leq_P B$) ha van olyan f **polinomidőben** kiszámítható függvény, amely A inputjaiból B inputjait készíti **választartó (IGEN példány IGEN marad)** módon.
- ha A polinomidőben visszavezethető B-re, és tudjuk, hogy maga B polinomidőben eldönthető, akkor A is eldönthető polinomidőben, valamint azt is tudjuk, hogy ha A-ra nincs polinomidejű algoritmus, akkor B-re sincs (bizonyítás a 29. oldalon)
- fő lépések:
 - inputkonverzió
 - bizonyítás: ha A IGEN példányából indulunk, akkor B IGEN példányát kapjuk
 - bizonyítás: ha B IGEN példányát kaptuk, akkor A IGEN példányából kellett indulni
- **Maze \leq_p Sokoban:** a cél mellé két új mező, egyikbe golyó, másikba lyuk
- **Hamilton-út \leq_p TSP(E):** városok súlya megkapható módosított szomszédsági gráf segítségével, 0: önmaga, 1: volt a két csúcs között él, 2: egyébként; TSP(E) második paramétere $N + 1$
- **Párosítás \leq_p SAT:** élekre felveszünk egy változót, igazra állítjuk, ha a párosításban benne volt, különben hamis; ezt követően leellenőrizzük, hogy a csúcsokra pontosan egy él illeszkedik.

Nemdeterminizmus

- egyes lépésekben nincs meghatározva, mi szerint adunk értéket a változóknak
- ugyanez formálisan: a változók értékkadása **nemdeterminisztikus**,
- amit egy algoritmus elvégez, majd determinisztikusan kiértékel

A P és az NP osztályok

- Két nagyon közel álló bonyolultsági osztály (egészen annyira közel áll, hogy egyesek szerint a kettő egyenlő - viszont ez még nem bizonyított)
- A P osztályba tartoznak azok a problémák, amelyet egy algoritmus **polinomidőben el tud dönten**. Ilyenek például:
 - Elérhetőség
 - TSP (Traveling Salesman Problem)
 - TSP(E)
 - Maze
 - Sokoban
 - Hamilton-út
 - SAT
 - Párosítás
- Az NP osztályba tartoznak azok a problémák, amelyre létezik **nemdeterminisztikus algoritmus, ami azt polinomidőben el tudja dönten**.
- Ebből a két tulajdonságból látszik, hogy a két halmaz elégé hasonlít, ha nem is egyenlők, azt biztosan tudjuk, hogy **P pont NP és coNP (NP komplementere) metszetében található**.

NP teljes problémák

- egy probléma akkor **C-nehéz**, ha C minden problémája visszavezethető rá
- egy probléma akkor **C-teljes**, ha C-nehéz, és ő maga is benne van C-ben
- **SAT** (Cook tétele)
- **FormSAT** ($\text{SAT} \leq_P \text{FormSAT}$)
- **Módosított SAT** (nem ez a hivatalos neve, $\text{FormSAT} \leq_P \text{Módosított SAT}$)
- **Hálózat-kielégíthetőség** ($\text{FormSAT} \leq_P \text{Hálózat-kielégíthetőség}$)
- **3SAT** (Hálózat-kielégíthetőség \leq_P 3SAT)
- **Független csúcshalmaz** (3SAT \leq_P Független csúcshalmaz)
- **Klikk** (Független csúcshalmaz \leq_P Klikk)
- **Hamilton-út** (párszor visszavezettük már rá, ezért TSP(E) is)
- **3-Színezés** (3SAT \leq_P 3-színezés)
- **Hármasítás** (3SAT \leq_P Hármasítás)
- **Pontos Lefedés Hármasokkal** (Hármasítás \leq_P Pontos Lefedés Hármasokkal)
- **Halmazlefedés** (Pontos Lefedés Hármasokkal \leq_P Halmazlefedés)
- **Halmazpakolás** (Pontos Lefedés Hármasokkal \leq_P Halmazpakolás)
- **Egész Értékű Programozás** (Integer Logical Programming / **ILP**, 3SAT \leq_P ILP)
- **Részletösszeg** (3SAT \leq_P Részletösszeg)
- **Partíció** (Részletösszeg \leq_P Partíció)
- **Hátizsák** (Részletösszeg \leq_P Hátizsák)
- Őszintén, ezeket jobb átolvasni a jegyzetből: lényegében annyiról van szó, hogy valami speci módon átalakítgatjuk az adott problémánk inputját egy NP teljes probléma inputjára ekvivalensen, és így megkapjuk, hogy a problémánk is NP teljes.

Bonyolultságelmélet - 2.)

A PSPACE osztály. PSPACE-teljes problémák. Logaritmikus tárígényű visszavezetés. NL-teljes problémák.

Egy adott program tárígénye az $f(n): N \rightarrow N$ függvény, ha bármely legfeljebb n méretű inputon $f(n)$ tárat használ.

A PSPACE osztály

Mielőtt rátérünk a PSPACE osztályra, elemezzük a SPACE osztályt. Ez a bonyolultsági osztály azon problémákat tartalmazza, amelyekre létezik egy $O(f(n))$ lineáris tárígényű program. Nem ismert, hogy NP és SPACE között mi az összefüggés, annyit tudunk biztosan, hogy ezek nem egyenlők. Ezen felül tudjuk még azt is, hogy NP zárt a visszavezetésre, SPACE pedig nem az.

Alapvető különbség a tár és idő között, hogy a tár újrafelhasználható. Egy nemdeterminisztikus program időbonyolultságát a leghosszabb szál hosszaként definiáltuk. Ehhez hasonlóan a tárígényt úgy definiáljuk, mint a legtöbb tárat használó tárígény. Tehát ebből következik, hogy egy nemdeterminisztikus program tárbonyolultsága akkor lineáris $f(n)$, ha tetszőleges, legfeljebb n méretű inputon legfeljebb lineáris $f(n)$ tárat használ. Azok a problémák, amelyek előírhatók nemdeterminisztikus programokkal $O(f(n))$ tárban, az **NSPACE** osztályban vannak.

Látva, hogy mik tartoznak az NSPACE osztályba, már feltételezhetjük, a **PSPACE** osztályba tartoznak azok a problémák, amelyek **polinom tárban előírhatók** (tehát a legtöbb tárat igénylő szál tárígénye egy polinom). További nevesített bonyolultsági osztályok:

R, RE - előírható illetve felismerhető problémák

TIME($f(n)$), **NTIME($f(n)$)** – det./nemdet. $O(f(n))$ időben előírható problémák

SPACE($f(n)$), **NSPACE($f(n)$)** – det/nemdet. $O(f(n))$ tárban előírható problémák

P = TIME(n^k) =TIME(n^k) – polinomidőben előírható problémák

NP = NTIME(n^k) – nemdet. polinomidőben előírható problémák

L = SPACE(log(n)) – logaritmikus tárban előírható problémák

NL = NSPACE(log(n)) – nemdet. logtárban előírhatóek

PSPACE = SPACE(n^k) – polinom tárban előírhatóek

NPSPACE = NSPACE(n^k) – nemdet. polinom tárban...

EXP = TIME(2^{n^k}) - exponenciális idő alatt előírható problémák.

Ezeknek sorrendje: $L \subseteq NL \subseteq P \subseteq NP \subseteq PSPACE \subseteq EXP$

Problémaosztálybeli teljesség, nehézség, hatékony visszavezetés

Ha C egy problémaosztály, A pedig egy előírtendő probléma, akkor:

- A **C-nehéz**, ha C-ben található összes probléma visszavezethető A-ra
- A **C-teljes**, ha Ő maga is C-ben van

Definíció szerint az A előírtési probléma **hatékonyan visszavezethető** a B előírtési problémára ($A \leq_P B$), ha van olyan f polinomidőben kiszámítható függvény, amely A inputjaiból B inputjait készíti **választartó** módon.

PSPACE teljes problémák

QSAT

Input: Q_1x_1, \dots, Q_mx_m alakú (predikátumkalkulusbeli) zárt (minden változót köt kvantor) formula

Kimenet: a formula értéke

```
function QSAT(F)
    if F ==  $\exists x F'$  then
        return QSAT(F' [x/0]) or QSAT(F' [x/1])
    else if F ==  $\forall x F'$ 
        return QSAT(F' [x/0]) and QSAT(F' [x/1])
    else
        return EVAL(F)
```

Itt nem kell megérteni, hogy miért oldja meg ez a probléma a QSAT problémát, egyszerűen kezdésnek csak fogadjuk el, hogy ez egy öt eldöntő program. Határozzuk meg valahogy a tárigényét!

Itt az EVAL sorral nem kell foglalkozni tárigény szempontjából, így nézzük először, hogy tárolunk-e változókat. Látható, hogy nem, csak rekurzív hívások történnek. Mindkét alkalommal ki kell számoljuk, hogy a beérkező formula milyen értéket vesz fel, ha az első kvantor által kötött változót rögzítjük 0-ra, majd 1-re. Ebből következik, hogy ha n darab változónk van az F formulában, akkor n-szer kell megvizsgálnunk 0-ra az értékeket, és n-szer kell megvizsgáljuk 1-re is őket. Ezeknek kiértékelésére egyesével lineáris tár elég. Mivel tudjuk, hogy lineáris (n) * lineáris (n) az gyakran polinom, ezért megkapjuk, hogy itt a tárigény $O(n^2)$.

Ide tartoznak még a Helyben elfogadás, Reguláris kifejezések ekvivalenciája, Véges automaták ekvivalenciája, Földrajzi játék, GÓ, Hex, Reversi, Dáma, Amőba (kétszemélyesek), Sokoban (egyszemélyes), Rush Hour, Életjáték

Logaritmikus tárigényű visszavezetés

Az f függvény az A eldöntési problémának a B eldöntési problémára való **logaritmikus tárigényű visszavezetése**, ha f kiszámítható logaritmikus tárban, és tetszőleges I inputra $A(I) = B(f(I))$ - tehát választartó.

A logaritmikus tárigényű (tehát mindenképp offline) algoritmusoknak polinom sok konfigurációja lehet adott input esetén, és mivel a visszavezetés nem eshet végtelen ciklusba, ezért az polinomidőben meg is áll. Ebből az következik, hogy ha egy A problémára egy B probléma logtárasan visszavezethető, akkor A-ra B polinomidőben is visszavezethető (nem ismert, hogy a két visszavezetés egybeesik-e, ha igen, akkor $L = P$).

A teljesség és nehézség fogalmát kicsit variálni kell. Vegyük az A problémát és egy C bonyolultsági osztályt. Ekkor A akkor lesz C-nehéz a logtáras visszavezetést nézve, ha minden probléma logtárasan visszavezethető A-ra. Továbbá akkor lesz A C-teljes, ha ő maga is benne van az adott bonyolultsági osztályban.

Azt mondjuk, hogy a logtáras visszavezetés tranzitív, ugyanis ha f az A-nak B-re, g pedig B-nek C-re való logtáras visszavezetése, akkor f és g kompozíciójával kapott összetett függvény pontosan A-nak C-re való logtáras visszavezetése.

NL-teljes problémák

Ide tartoznak azok az eldöntendő problémák, amelyekre adható egy olyan logtáras, nemdeterminisztikus program, amely azt eldönti. Ezen felül NL összes problémája logtárban visszavezethető rá.

Mivel P, ezért NL bármely két nemtriviális problémája hatékonyan visszavezethető egymásra, így a polinomidejű visszavezetésre nézve ezen osztályokon belül a "teljesség" fogalma értelmetlenne válik, ugyanis a bennük található problémák bármelyike visszavezethető lesz rájuk, ha ők is eldönthetőek polinomidőben. Azt mondjuk, hogy **egy probléma NL/P-teljes/nehéz, ha a logtáras visszavezetésre nézve az**.

2SAT

Input: Q_1x_1, \dots, Q_mx_m alakú (predikátumkalkulusbeli) zárt (minden változót köt kvantor) formula
Kimenet: a formula értéke

Ide tartozik még például az ELÉRHETŐSÉG probléma is.

Összegzés

Formális nyelvek - 1.)

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

Formális nyelv: a matematika, a logika és az informatika számára egy véges ábécéből generálható, véges hosszúságú szavak (például karakter stringek, jelsorozatok) halmaza

Mivel foglalkozik a formális nyelvek?

- A formális nyelvek elmélete szimbólumsorok halmazaival foglalkozik
- Célja: véges, tömör, leírást adni az ilyen halmazoknak

Jelölések:

- | | |
|--------------------------------|--|
| - a, b, c, d | Σ elemei |
| - A, B, C, D és S | N elemei |
| - U, V, W, X, Y, Z | N \cup Σ elemei |
| - $\alpha \beta \gamma \delta$ | (N \cup Σ) [*] elemei |
| - u, v, w, x, y, z | Σ^* elemei |

Ábécé: Szimbólumok véges nem üres halmaza (jele: Σ), pl. $\Sigma = \{a, b, c\}$

Szó: Egy Σ ábécé elemeiből képzett véges sorozat. 0 hosszú szó az üres szó (jele: ϵ)

- pl. a, aa, aabba
- Összes szó halmaza
- $\Sigma^* = \{\epsilon, a, b, aa, ab, aba\}$
- $\Sigma^+ = \{a, b, aa, ab, aba\}$

Nyelv: Σ^* tetszőleges részhalmazát Σ feletti nyelvnek nevezzük (jele: L). Ha L véges számú szóból áll, akkor véges nyelv. Szavak egy véges vagy végtelen halmaza.

- pl. legyen $\Sigma = \{a, b\}$ ábécé
- $L = \{a, b, ab\}$ véges nyelv
- $L = \{a^i b^j \mid 0 \leq i\}$ végtelen nyelv

Nyelvtan: Olyan végesen specifikált eszköz, mellyel nyelvek megadására van lehetőség. (a nyelv és nyelvtan két különböző fogalom, Egy nyelv szavak halmaza, nyelvtan egy végesen specifikált eszköz nyelvek generálására)

Véges automaták és a felismert nyelv

A determinisztikus automaták olyan véges automaták, ahol egy adott állapotból egy bizonyos betű előfordulására az automata csak egy állapotba kerülhet át (ergo nincsenek benne ugyanarra a betűre elágazások), amelyet az $M = (Q, \Sigma, \delta, q_0, F)$ elemekkel jelölünk, ahol:

- Q : az állapotok nem üres, véges halmaza
- Σ : inputábécé
- δ : egy $Q \times \Sigma \rightarrow Q$ leképezés (átmenetfüggvény)
- q_0 : kezdőállapot
- F : végállapotok halmaza (Q részhalmaza)

Egy automata megadható táblázat formában is: A kezdő állapotot a táblázat első sorába írjuk, a végállapotokat pedig megjelöljük.

Minden lépésben az automata beolvassa a következő input szimbólumot, és átmegy egy olyan állapotba, amelyet az átmeneti függvény meghatároz az adott aktuális input szimbólumra, és azon állapotra vonatkozóan, amelyben az automata az adott pillanatban van.

Az input szó felfogható úgy, mint egy cellákra osztott szalag, aminek minden helye egy szimbólumot tartalmaz. Kezdetben az A véges automata a q_0 kezdőállapotban van, és az olvasófej az input szalagon lévő u szó első betűjét dolgozza fel. Ezután a véges automata lépések sorozatát hajtja vére, így olvassa végig az u szót, betűről betűre haladva.

Átmeneti reláció:

$(q, w), (q', w') \in C$ esetén $(q, w) \vdash_M (q', w')$ ha $w = aw'$, valamely $a \in \Sigma$ -ra és $\delta(q, a) = q'$.

$(q, w) \vdash_M (q', w')$, egy lépés

$(q, w) \vdash_M^n (q', w')$, $n \geq 0$ lépés

$(q, w) \vdash_M^+ (q', w')$, legalább egy lépés

$(q, w) \vdash_M^* (q', w')$, valamennyi (esetleg 0) lépés

Az $M = (Q, \Sigma, \delta, q_0, F)$ automata által felismert nyelven az

$$L(M) = \{w \in \Sigma^* \mid (q_0, w) \vdash_M^* (q, \epsilon) \text{ és } q \in F\}$$

nyelvet értjük.

Szavakkal: q_0 -ból a w hatására valamelyik $q \in F$ végállapotba jutunk.

A nemdeterminisztikus automaták olyan véges automaták, ahol egy adott állapotból egy input szimbólum hatására több állapotba is átkerülhet (nem tudjuk determinisztikusan eldönteni, hogy melyik állapotba kellene továbbhaladnunk, így párhuzamosan vezetjük tovább a lehetséges utat), amelyet az $M = (Q, \Sigma, \delta, q_0, F)$ elemekkel jelölünk, ahol:

- Q : az állapotok nem üres, véges halmaza
- Σ : inputábécé
- δ : egy $Q \times \Sigma \rightarrow P(Q)$ leképezés
- q_0 : kezdőállapot
- F : végállapotok halmaza (Q részhalmaza)

Az $M = (Q, \Sigma, \delta, q_0, F)$ automata által felismert nyelven az

$$L(M) = \{w \in \Sigma^* \mid (q_0, w) \vdash_M^* (q, \varepsilon) \text{ valamely } q \in F\text{-re}\}$$

nyelvet értjük.

Szavakkal: q_0 -ból a w hatására elérhető valamely $q \in F$ végállapot (ugyanakkor esetleg nem végállapotok is elérhetők).

Az $M = (Q, \Sigma, \delta, q_0, F)$ nemdeterminisztikus automata **teljesen definiált (teljes)**, ha minden állapot ($q \in Q$) és betű ($a \in \Sigma$) esetén az átmeneti függvény által generált állapothalmaz legalább egyelemű (vagyis nem tudnak elakadni).

TÉTEL: Tetszőleges $M = (Q, \Sigma, \delta, q_0, F)$ nemdeterminisztikus automatához megadható egy olyan $M = (Q, \Sigma, \delta, q_0, F)$ teljesen definiált automata, amely ugyanazt a nyelvet ismeri fel (ergo minden nemdeterminisztikus automata teljesre hozható).

TÉTEL: Egy nyelv akkor és csak akkor ismerhető fel nemdeterminisztikus automatával, ha a felismerhető determinisztikus automatával (ergo a nemdeterminisztikus automaták által felismert nyelvek a determinisztikus automatával felismert nyelvek részhalmazát képezik).

A nemdeterminisztikus ε -automaták olyan véges automaták, ahol egy adott állapotból egy input szimbólum hatására akár több állapotba is átkerülhetünk, és olyan átmenet is megengedett, amelyik "nem fogyasztja az inputot". Ezt is az $M = (Q, \Sigma, \delta, q_0, F)$ elemekkel jelölünk, ahol:

- Q : az állapotok nem üres, véges halmaza
- Σ : inputábécé
- δ : egy $Q \times (\Sigma \cup \varepsilon) \rightarrow P(Q)$ leképezés, tehát az inputábécé kibővítve az üres epszilonnal
- q_0 : kezdőállapot
- F : végállapotok halmaza (Q részhalmaza)

Egy ε -átmenet



A felismert nyelv definíciója ugyanaz, mint a nemdeterminisztikus esetben: az M automata felismeri a w szót, ha q_0 -ból w hatására elérhető valamelyik $q \in F$ végállapot (esetleg ϵ -átmenetek segítségével).

TÉTEL: Egy nyelv akkor és csak akkor ismerhető fel nemdeterminisztikus ϵ -automatával, ha a felismerhető nemdeterminisztikus automatával (még szűkebb halmaz, mint a nemdeterminisztikus)

Egy nemdeterminisztikus ϵ -automatához megadható egy olyan nemdeterminisztikus automata, amely ugyanazt a nyelvet ismeri fel, ehhez ki kell számoljuk az **állapotok ϵ -lezárását az eredeti automatában**. Ezt úgy kapjuk meg, hogy egy $\{q\}$ halmazból kiindulva hozzávesszük a belőle egy ϵ -átmenettel elérhető állapotokat, és ezt az eljárást addig folytatjuk, ameddig a halmaz bővíthető.

A reguláris nyelvtanok

Egy $G = (N, \Sigma, P, S)$ nyelvtanra azt mondjuk, hogy az **reguláris** (jobblineáris), ha P-ben minden szabály $A \rightarrow xB$, vagy $A \rightarrow x$ alakú. Ez szavakkal leírva azt jelenti, hogy minden szabály jobb oldalán legfeljebb egy nemterminális van, és ha van, akkor az a szó jobb oldali végén van. Ez érvényes minden olyan szóra is, amelyet az S kezdőszimbólumból vezetünk le. Ezeket a levezetéseket az $A \rightarrow x$ alakú szabállyal fejezzük be.

Ezen elindulva egy L nyelvet akkor nevezünk **reguláris nyelvnek**, ha létezik olyan G reguláris nyelvtan, melyre $L = L(G)$. Ezt szavakkal úgy is mondhatjuk, hogy egy nyelv akkor reguláris, ha létezik olyan reguláris nyelvtan, amellyel az generálható. Az reguláris nyelvek halmazát REG-gel jelöljük. minden reguláris nyelvtan környezetfüggetlen, így ebből következik az is, hogy minden reguláris nyelv is az.

Véges automaták és reguláris kifejezések ekvivalenciája

Veszünk egy ábécét, és hozzáveszünk néhány segédszimbólumot. Ezekből reguláris kifejezéseket építünk fel bizonyos szabályok szerint. minden reguláris kifejezés meghatároz egy nyelvet. Egy Σ ábécé feletti reguláris kifejezések halmaza a $(\Sigma \cup \{\emptyset, \epsilon, (,), +, *\})^*$ halmaz **legsűkebb** olyan U részhalmaza, amelyre teljesül, hogy:

- \emptyset eleme U-nak,
- ϵ eleme U-nak,
- Σ minden eleme U-nak
- ha vesszük U két elemét, akkor azok kombinációja (+: vagy, *: valamennyiszer, konkatenáció) is eleme lesz U-nak

$|R|$ az R reguláris kifejezés által meghatározott nyelv. Egy L nyelv akkor reprezentálható reguláris kifejezéssel, ha van L szimbólumai felett egy olyan R reguláris kifejezés, amely meghatározza L-t. Példa: $L = \{uabav \mid u, v \in \Sigma^*\} =$ a szóban előfordul az aba részszó. Ezt az L nyelvet az alábbi reguláris kifejezéssel reprezentálhatjuk: $L = |(a+b)^*aba(a+b)^*| =$ a vagy b betű található a szó elején valahányszor (akár 0-szor), utána valahol előkerül az aba részszó, majd megint a vagy b valahányszor a szó végén. Látható, hogy az automatával felismerhető nyelvek és a reguláris kifejezések elég szoros kapcsolatban állnak egymással, bizonyítsuk be, hogy:

L generálható reguláris nyelvtannal (1) = L felismerhető automatával (2) = L reprezentálható reguláris kifejezással (3)

1. Lemma: (3) \Rightarrow (1)

2. Lemma: (1) \Rightarrow (2)

3. Lemma: (2) \Rightarrow (3)

(1) \Leftrightarrow (2) \Leftrightarrow (3).

1. Lemma: a reprezentálható nyelvek regulárisak

Ezt be tudjuk bizonyítani az L-et reprezentáló R reguláris kifejezés struktúrája szerinti indukcióval (következetéssel). Lényegében a bizonyos reguláris kifejezésekhez megadunk egy olyan nyelvtant, ami kigenerálja az általuk reprezentált nyelvet. Ha:

- $R = \emptyset$, akkor az általa reprezentált nyelv generálható a $G = (\{S\}, \Sigma, \emptyset, S)$ nyelvtanból
- $R = a, \epsilon$, az R által reprezentált nyelv generálható a $G = (\{S\}, \Sigma, \{S \rightarrow a\}, S)$ nyelvtanból
- $R = (R1) + (R2)$, a nyelvtan $G = (N_1 \cup N_2 \cup \{S\}, \Sigma, P_1 \cup P_2 \cup \{S \rightarrow S_1, S \rightarrow S_2\}, S)$
- $R = (R1)(R2)$ esetén a $G = (N_1 \cup N_2, \Sigma, P, S_1)$ nyelvtan generálja a nyelvet
- $R = (R1)^*$ kifejezéshez tartozó nyelvtan pedig $G = (N_1 \cup \{S\}, \Sigma, P, S)$

2. Lemma: a reguláris nyelvek felismerhetők automatával

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

Az ilyen jobblineáris nyelvtanokhoz megadható egy nemdeterminisztikus ϵ -automata, amely felismeri a nyelvtan által meghatározott nyelvet.

3. Lemma: az automatával felismerhető nyelvek reprezentálhatók

S. C. Kleene tétele szerint minden automatával felismerhető nyelv reprezentálható reguláris kifejezéssel. A bizonyításhoz veszünk egy nemdeterminisztikus automatát, ami felismeri az L nyelvet, majd megadunk egy L-et reprezentáló reguláris kifejezést.

Reguláris nyelvekre vonatkozó pumpáló lemma

Minden L reguláris nyelv esetén megadható egy (L -től függő) $k > 0$ egész szám úgy, hogy minden szóra a nyelvből teljesül, hogy ha k -nál hosszabb, akkor létezik olyan $w = w_1w_2w_3$ felbontás, melyre $0 < |w_2|$, és $|w_1w_2| \leq k$, valamint w_2 -t tetszőlegesen sokszor egymás mellé írva még mindig L-beli szót kapunk. Ebből következik, hogy ha egy adott nyelvhez nem adható meg egy ilyen k index, ami teljesítené a fent említett feltételeket, akkor a nyelv nem reguláris. Ezért a pumpáló lemma egyik alkalmazása a nyelvek regularitásának bizonyítása.

Az $L = \{a^n b^n \mid n \geq 0\}$ nyelv olyan szavakat tartalmaz, amik n darab a betűvel kezdődnek, majd azokat n darab b betű követi (például aaabbb), keressük a fent említett k indexet. Vizsgáljuk például az $a^k b^k$ szót, melynek hossza láthatóan $2k$. Találnunk kell egy olyan felbontást, amely tartja a pumpáló lemma szabályait. Mivel tudjuk, hogy $|w_1w_2| \leq k$, ezért az első és középső szó is csak a betűket tartalmazhat, így akármekkora is lesz ebből az első szó, a második szót többszörözve csak a betűket adunk hozzá, így azok már nem lesznek benne a nyelvben, tehát L nem reguláris.

A pumpáló lemma egyik következményeként megtudtuk, hogy van olyan környezetfüggetlen nyelv, amelyik nem reguláris.

Összegzés

Véges automata és változatai, a felismert nyelv definíciója

- a véges automata a legegyszerűbb gép minták felismerésére
- az általuk felismert nyelv olyan szók halmaza, amelyek hatására a q_0 kezdőállapotból indulva egy végállapotba jutunk (végállapatos felismerés)
- determinisztikus véges automata: minden állapotból egy adott szimbólumra csak egy út vezet ki
- nemdeterminisztikus véges automata: előfordulhat, hogy némely állapotból egy adott szimbólumra több út is vezet ki
- nemdeterminisztikus ϵ -automata: léteznek üres (ϵ) átmenetek bizonyos állapotokból

Reguláris nyelvtanok

- egy nyelvtan reguláris (jobblineáris), ha minden szabálya $A \rightarrow xB$, vagy $A \rightarrow x$ alakú
- minden szabály jobb oldalán legfeljebb egy nemterminális van, és ha van, akkor az a szó jobb oldali végén van.
- érvényes minden olyan szóra is, amelyet az S kezdőszimbólumból vezetünk le
- egy L nyelv reguláris, ha létezik olyan G reguláris nyelvtan, melyre $L = L(G)$
- egy nyelv akkor reguláris, ha létezik olyan reguláris nyelvtan, amellyel az a nyelv generálható.
- reguláris nyelvek halmaza REG
- minden reguláris nyelvtan környezetfüggetlen, így ebből következik az is, hogy minden reguláris nyelv is az.

Véges automaták és reguláris kifejezések ekvivalenciája

- L generálható reguláris nyelvtannal (1)
 - bizonyítható az L -et reprezentáló R reguláris kifejezés struktúrája szerinti indukcióval (következtetéssel)
 - a reguláris kifejezésekhez megadunk egy olyan nyelvtant, ami kigenerálja az általuk reprezentált nyelvet
- L felismerhető automatával (2)
 - minden reguláris nyelvtanhoz megadható egy vele ekvivalens másik reguláris nyelvtan úgy, hogy minden szabály $A \rightarrow B$, $A \rightarrow aB$ vagy $A \rightarrow \epsilon$ alakú legyen
 - Az ilyen jobblineáris nyelvtanokhoz megadható egy nemdeterminisztikus ϵ -automata, amely felismeri a nyelvtan által meghatározott nyelvet.
- L reprezentálható reguláris kifejezéssel (3)
 - S. C. Kleene tétele szerint minden automatával felismerhető nyelv reprezentálható reguláris kifejezéssel

Reguláris nyelvekre vonatkozó pumpáló lemma

- a reguláris nyelvekre megadható egy olyan $k > 0$ egész szám,
- hogy minden szóra a nyelvből ha az k -nál hosszabb,
- akkor van olyan $w = w_1w_2w_3$ felbontás, melyre $0 < |w_2|$, és $|w_1w_2| \leq k$,
- és w_2 tetszőleges többszörözése (pumpálása) után is a nyelvbe illő szót kapunk
- nyelvek regularitásának bizonyítására használható
- bizonyítható vele, hogy van olyan környezetfüggetlen nyelv, amely nem reguláris
- következményeként megkaptuk, hogy $REG \subset CF$
- a reguláris nyelvek valódi részét képezik a környezetfüggetlen nyelveknek

Formális nyelvek - 2.)

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

Környezetfüggetlen nyelvtan és nyelv

Egy környezetfüggetlen nyelvtan megadható az alábbi módon: $G = (N, \Sigma, P, S)$, ahol:

- N a nemterminálisok ábécéje
- Σ egy terminális (befejező, végső) ábécé, nincs közös eleme az N halmazzal
- S a kezdő szimbólum (start)
- P pedig $A \rightarrow a$ alakú átírási szabályok véges halmaza úgy, hogy $A \in N$ és $a \in (N \cup \Sigma)^*$

Akkor mondjuk egy nyelvtanra, hogy környezetfüggetlen, ha P összes szabálya $A \rightarrow a$ alakú.

Nyelvtanok segítségével tudunk nyelveket generálni. Egy ilyen nyelvbe pontosan azok a szavak fognak bekerülni, amiket S kezdőszimbólumaiból G nyelvvel levezethetünk. Egy nyelvet általában nem csak egy nyelvtannal lehet generálni, éppen ezért ha két nyelvtan ugyanazt a nyelvet generálja, akkor ők **ekvivalensek**. Ha egy L nyelvhez meg tudunk adni egy környezetfüggetlen nyelvtant, akkor L környezetfüggetlen nyelv (ennek a kontrapozíciója, hogy L akkor és csak akkor környezetfüggetlen, ha van olyan G környezetfüggetlen nyelvtan, amely generálja őt - $L = L(G)$). Az ilyen nyelvek halmazát **CF**-fel jelöljük

Derivációk és derivációs fák

Levezetések (derivációk):

- $\gamma \xrightarrow{G} \delta$ egy lépés (= közvetlen levezetés)
- $\gamma \xRightarrow[G]{n} \delta$ $n \geq 0$ lépés, ugyanis ha 0 lenne, akkor γ és δ egyenlő kellene, hogy legyen
- $\gamma \xRightarrow[G]{+} \delta$ legalább egy lépés
- $\gamma \xRightarrow[G]{*} \delta$ valamennyi (esetleg 0) lépés

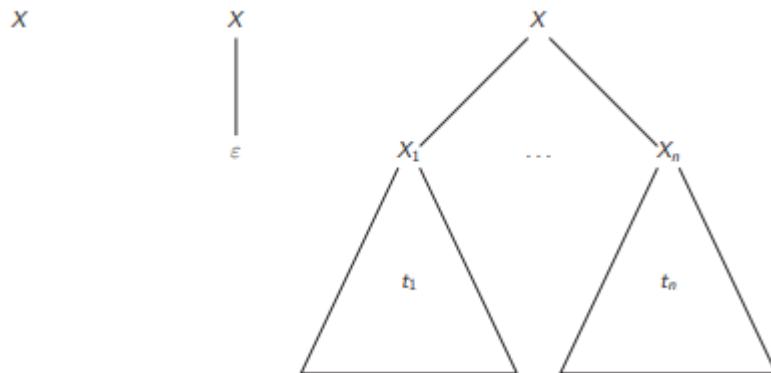
Azt mondjuk, hogy a v szó közvetlenül (vagy egy lépéssben) levezethető az u szóból G -ben, ha teljesül rájuk az alábbi formula

$$u \xrightarrow{G} v$$

Általában azok a levezetések az érdekesek, amelyek a kezdőszimbólumból indulnak ki, és a terminális ábécé valamelyik elemében végződnek.

Az $X \in (N \cup \Sigma)$ gyökerű derivációs fák halmaza a legszűkebb olyan D_X halmaz, amelyre:

- csak egyetlen X csúcspontja (a gyökere) van, eleme D_X halmaznak
- ha $X \rightarrow \varepsilon \in P$ akkor a gyökér az X , egyetlen leszármazottja pedig az ε , eleme D_X -nek
- Ha $X \rightarrow X_1 \dots X_n \in P$, továbbá $t_1 \in D_{X_1}, \dots, t_n \in D_{X_n}$, akkor az a fa, amelynek gyökere X , a gyökeréből n él indul rendre a t_1, \dots, t_n fák gyökeréhez, eleme D_X -nek



Legyen t egy X gyökerű derivációs fa. Ekkor t magasságát $h(t)$ -vel, a határát pedig $fr(t)$ -vel jelöljük és az alábbi módon definiáljuk:

- Ha t az egyetlen X szög pontból álló fa, akkor $h(t) = 0$ és $fr(t) = X$
- Ha t gyökere X , aminek egyetlen leszármazottja ε , akkor $h(t) = 1$ és $fr(t) = \varepsilon$.
- Ha t gyökere X , amiből n él vezet rendre a t_1, \dots, t_n közvetlen leszármazott részfák gyökeréhez, akkor $h(t) = 1 + \max\{h(t_i) \mid 1 \leq i \leq n\}$ és $fr(t) = fr(t_1) \dots fr(t_n)$.

Informálisan: $h(t)$ a t -ben lévő olyan utak hosszának maximuma, amelyek t gyökerétől annak valamely leveléhez vezetnek. Továbbá, $fr(t)$ azon $(N \cup \Sigma)^*$ -beli szó, amelyet t leveleinek balról jobbra (vagy: preorder) bejárásával kapunk.

Veremautomaták és környezetfüggetlen nyelvtanok ekvivalenciája

Veremautomatának (vagy pushdown automatának, PDA) nevezzük azt a $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$ rendszert, ahol:

- Q egy véges halmaz, az állapotok halmaza
- Σ az input ábécé;
- Γ a verem ábécé;
- $q_0 \in Q$ a kezdőállapot;
- $Z_0 \in \Gamma$ a verem kezdő szimbóluma;
- $F \subseteq Q$ a végállapotok halmaza;
- $\delta : Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma \rightarrow Pf(Q \times \Gamma^*)$ az átmenet függvény

Alapvetően ez egy nemdeterminisztikus modell, valamint még az ε -átmenetek is megengedettek (nem minden lépésben "fogyasztjuk" az inputot). Ha esetleg egy ilyen átmeneten lépnénk tovább, azt ε -lépéseknek nevezzük.

A $C = Q \times \Sigma^* \times \Gamma^*$ halmazt a P konfigurációi halmazának nevezzük. Egy $(q, w, y) \in C$ konfiguráció jelentése az, hogy P a $q \in Q$ állapotban van, a $w \in \Sigma^*$ input szót kapja és vermények tartalma y .

A vermet egy szónak tekintjük. A veremben felülről számított első, második, stb betű a szó első, második, stb betűje. Például: AAZ_0 az a verem, melyben három betű van, a legfelső betű A .

CF nyelvtanok és PDA-k ekvivalenciája

Tétel: minden L környezetfüggetlen nyelvhez van olyan P veremautomata melyre $L(P) = L$ (ugyanazt a nyelvet ismerik fel)

- Legyen $G=(N, \Sigma, P, S)$ egy környezetfüggetlen nyelvtan, ami generálja L nyelvet. Az ötlet az, hogy a veremben készítjük a vezetést és amikor ezzel sikerül előállítani a bemeneti szó következő karakterét, akkor ezt a karaktert kiveszük a veremből és a szón is továbblépünk a következő karakterre.
- Ehhez legyen $\Gamma = N \cup \Sigma \cup \{Z_0\}$, $Q = \{q_0, q, q_e\}$, q_e az elfogadó állapotba
- Először a nyelvtan kezdő változóját berakjuk a verembe: $\delta(q_0, \epsilon, Z) = \{(q, SZ)\}$.
- A q állapotban, ha a verem tetején a nyelvtan egy A változója van, akkor helyettesítsük ezt egy A -hoz tartozó szabály jobb oldalával, azaz $\delta(q, \epsilon, A)$ az olyan (q, a) párokból áll, melyekre a nyelvtanban van $A \rightarrow a$ szabály
- Amikor a q állapotban a verem tetején $a \in \Sigma$ karakter áll és ez megegyezik a szó következő karakterével, akkor kidobjuk a veremből. Amennyiben nem a megfelelő karakter jelenik meg a veremben, akkor a számítás elakad, $\delta(q, a, a) = \{(q, \epsilon)\}$.
- Az elfogadáshoz kell még egy $\delta(q, \epsilon, Z) = \{(q_e, Z)\}$ átmenet, amivel átlépünk az elfogadó állapotba.

Tétel: minden veremautomata által felismert nyelv környezetfüggetlen

A Bar-Hillel lemma

A Bar-Hillel lemma szerint minden környezetfüggetlen nyelvhez megadható egy olyan $k > 0$ egész szám, ami a nyelvben található minden szóra képes egy olyan 5 részre történő felbontást csinálni, amely középső három részének szimbólumszáma kisebb, vagy egyenlő mint a k index, második és negyedik része nem üres, és ezeket tetszőlegesen sokszorosítva még mindig a nyelvbe illő szót kapunk. Formálisan:

- megadható egy olyan $k > 0$ egész szám, hogy
- minden $w \in L$ esetén,
- ha $|w| \geq k$, akkor van olyan $w = w_1 w_2 w_3 w_4 w_5$ felbontás, melyre
 - ha $|w_2 w_3 w_4| \leq k$
 - és $w_2 w_4 \neq \epsilon$
 - akkor minden $n \geq 0$ -ra $w_1 w_2^n w_3 w_4^n w_5 \in L$

Bizonyítás. Legyen $L = L(G)$, ahol $G = (N, \Sigma, P, S)$
Chomsky-normálformában lévő cf nyelvtan.

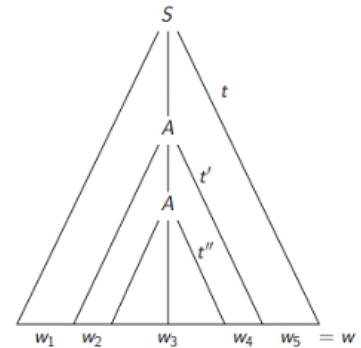
Legyen $k = 2^m$, ahol $m = ||N||$. Megmutatjuk, hogy ez a k megfelelő lesz.

Vegyük egy $w \in L$ szót, melyre $|w| \geq k$. Akkor van olyan $t \in D_S$ derivációs fa, melyre $fr(t) = w$ és $h(t) \geq m + 1$.

Ha ugyanis $h(t) \leq m$ lenne, akkor a t derivációs fa mindegyik útján legfeljebb m nemterminális lenne (a legalsó csúcsponton terminális van).

Mivel (Chomsky-normálforma miatt) a fa minden nemterminális csúcspontban legfeljebb kétfelé ágazik, a legalsó nemterminálisnál pedig már nem ágazik el, a w szó hossza legfeljebb 2^{m-1} lenne.

Következésképpen t -ben van olyan, S -től valamely levélhez vezető út melynek hossza legalább $m+1$ és amelyen ezért (egy terminális és) legalább $m+1$ nemterminális szimbólum szerepel.



Mivel $m = ||N||$, van olyan nemterminális, ami ezen az úton legalább kétszer előfordul. Jelölje A azt a nemterminálist, amelyik a terminális levéltől indulva a gyökér felé legelőször megismétlődik.

Definiáljuk w_1, w_2, w_3, w_4 és w_5 szavakat az ábrán látható módon. Ekkor nyilván $w = w_1 w_2 w_3 w_4 w_5$, mert t határát osztottuk fel. Továbbá:

- 1) $|w_2 w_3 w_4| \leq k$, mert a legelső ismétlődést vettük,
- 2) $w_2 w_4 \neq \epsilon$, mert az A "felső" előfordulásánál a fa kétfelé ágazik és egyik ágon sem vezethető le ϵ (Chomsky-normálforma!),
- 3) minden $n \geq 0$ -ra, $w_1 w_2^n w_3 w_4^n w_5 \in L$, mert a t' fa iterálható.

Összegzés

Környezetfüggetlen nyelv és nyelvtanok

- egy nyelvtan környezetfüggetlen, ha P összes szabálya $A \rightarrow \alpha$ alakú.
- egy nyelvbé pontosan azok a szavak tartoznak, amiket S kezdőszimbólumaiból G nyelvtannal levezethetünk
- egy nyelvet nem csak egy nyelvtannal lehet generálni
- ha két nyelvtan ugyanazt a nyelvet generálja, akkor ők **ekvivalensek**
- L akkor és csak akkor környezetfüggetlen, ha van olyan G környezetfüggetlen nyelvtan, amely generálja őt - $L = L(G)$.
- környezetfüggetlen nyelvek halmazát CF-fel jelöljük

Derivációk és derivációs fák

- Levezetések (derivációk):
 - $\gamma \Rightarrow_G \delta$ egy lépés (= közvetlen levezetés)
 - $\gamma \Rightarrow_G^n \delta$ $n \geq 0$ lépés, (ha 0, akkor γ és δ egyenlő)
 - $\gamma \Rightarrow_G^+ \delta$ legalább egy lépés
 - $\gamma \Rightarrow_G^* \delta$ valamennyi (esetleg 0) lépés
- a v szó közvetlenül levezethető az u szóból G -ben, ha $u \Rightarrow_G v$ teljesül
- az a fontos, amikor a levezetés a kezdőszimbólumból indul, és a terminális ábécé valamelyik elemében végződik
- Az $X \in (N \cup \Sigma)$ gyökerű derivációs fák halmaza a legszűkebb olyan D_X halmaz, amelyre:
 - ha csak egyetlen X csúcspontja (a gyökere) van, eleme DX halmaznak
 - ha $X \rightarrow \epsilon \in P$ akkor a gyökér az X , egyetlen leszármazottja pedig az ϵ , eleme DX -nek
 - ha $X \rightarrow X_1 \dots X_n \in P$, továbbá $t_1 \in D_{X_1}, \dots, t_n \in D_{X_n}$, akkor az a fa, amelynek gyökere X , a gyökeréből n él indul rendre a t_1, \dots, t_n fák gyökeréhez, eleme D_X -nek
- a fa magasságát $h(t)$ -vel, a határát pedig $fr(t)$ -vel jelöljük
 - ha t az egyetlen X szögpontból álló fa, akkor $h(t) = 0$ és $fr(t) = X$
 - ha t gyökere X , aminek egyetlen leszármazottja ϵ , akkor $h(t) = 1$ és $fr(t) = \epsilon$.
 - Ha t gyökere X , amiből n él vezet rendre a t_1, \dots, t_n közvetlen leszármazott részfák gyökeréhez, akkor $h(t) = 1 + \max\{h(t_i) \mid 1 \leq i \leq n\}$ és $fr(t) = fr(t_1) \dots fr(t_n)$.
- $h(t)$ a t -ben lévő olyan utak hosszának maximuma, amelyek t gyökerétől annak valamely leveléhez vezetnek
- $fr(t)$ azon $(N \cup \Sigma)^*$ -beli szó, amelyet t leveleinek preorder bejárással történő leolvasásával kapunk

Veremautomaták és környezetfüggetlen nyelvtanok ekvivalenciája

- egy nemdeterminisztikus modell
- ϵ -átmenetek is megengedettek (nem minden lépésben "fogyasztjuk" az inputot), ilyen átmenetek esetén ϵ -lépésről beszélünk
- a $C = Q \times \Sigma^* \times \Gamma^*$ halmazt a P konfigurációi halmazának nevezzük
- a $(q, w, \gamma) \in C$ konfiguráció jelentése az, hogy P a $q \in Q$ állapotban van, a $w \in \Sigma^*$ input szót kapja és vermények tartalma γ .
- a vermet is egy szónak tekintjük
- ha q állapotban a w szó következő a szimbóluma esetén a veremautomata kiveszi az a szimbólumot, és beteszi a verembe a következőt, és átkerül egy másik állapotra

- elfogadás történhet üres veremmel és végállapottal is
- minden L környezetfüggetlen nyelvhez egy őt felismerő veremautomata

A Bar-Hillel lemma

- minden környezetfüggetlen nyelvhez megadható egy olyan $k > 0$ egész szám
- a nyelvben található minden szóra
- képes egy olyan 5 részre történő felbontást csinálni,
- amely középső három részének szimbólumszáma kisebb, vagy egyenlő mint k
- második és negyedik része nem üres, és ezeket tetszőlegesen sokszorosítva még mindig a nyelvbe illő szót kapunk

Közelítő-, és szimbolikus számítások - 1.)

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

Általános fogalmak

- 1.) Egy A mátrix **diagonális**, ha csak a főátlóban tartalmaz elemeket
- 2.) Egy A mátrix **egységmátrix** (I), ha olyan diagonális mátrix, amely csak egyeseket tartalmaz a főátlóban
- 3.) Egy A mátrix **szimmetrikus**, ha $A = A^T$, azaz megegyezik a transzponáltjával
- 4.) Egy A mátrix **pozitív definit**, ha szimmetrikus, és minden nem 0 vektorra $x^T A x > 0$.
- 5.) Egy A mátrix **i-edik főminorja** az A első i sorából és i oszlopából képzett A_i mátrix determinánsa
- 6.) Egy A mátrix **háromszögmátrix**, ha csak a főátlóban és az alatta/felette lévő helyeken tartalmaz értéket, a többi érték 0.
- 7.) Egy A mátrix **szinguláris, vagy degenerált**, ha determinánsa 0, nem invertálható.
- 8.) Egy A mátrix **invertálható**, ha létezik olyan B mátrix, mellyel öt megszorozva egységmátrixot kapunk.
- 9.) Egy $A_{n \times m}$ mátrix **transzponáltja** az az $A^T_{n \times m}$ mátrix, ahol A sorai A^T oszlopai, és A oszlopai A^T sorai
- 10.) Az A mátrix **ortogonális**, ha a mátrix inverze megegyezik a transzponáltjával ($Q Q^T = I$)
- 11.) Az A mátrix **diagonálisan domináns**, ha minden főátlóbeli elemének abszolút értéke nagyobb vagy egyenlő az öt tartalmazó sor összes többi eleménél
- 12.) Az A mátrix **szigorúan diagonálisan domináns**, ha minden főátlóbeli elemének abszolút értéke nagyobb az öt tartalmazó sor összes többi eleménél

Eliminációs módszerek

Ha fogunk egy tetszőleges a vektort, akkor erre viszonylag egyszerű megadni egy olyan M_k mátrixot, amellyel azt balról beszorozva a k-adik elem alatti együtthatók eltűnjenek (tehát legyen az értékük 0):

$$M_k a = \begin{pmatrix} 1 & & & & \\ & \ddots & & & \\ & & 1 & & \\ & & & -a_{k+1}/a_k & \\ & & & \vdots & \ddots \\ & & & -a_n/a_k & 1 \end{pmatrix} \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_k \\ a_{k+1} \\ \vdots \\ a_n \end{pmatrix} = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_k \\ 0 \\ \vdots \\ 0 \end{pmatrix}$$

Az M_k mátrix ki nem írt értékei 0, a főátlójában pedig csak 1-esek szerepelnek. A vektorban található a_k együtthatót **generálóelemnek**, a mátrixot pedig **eliminációs mátrixnak**, vagy **gauss-transzformációknak** nevezzük. Az eliminációs mátrixok fő tulajdonságai:

- reguláris (nem szinguláris - van inverze, amivel ha beszorozzuk, egyet kapunk)
- $M_k = I - m e_k^T$ (I : egységmátrix, m : egy olyan vektor, ami a generálóelem indexével bezárólag 0-kat tartalmaz, utána meg az előző érték és generálóelem hányadosát, e_k pedig az egységmátrix k-adik oszlopa), **főátlón kívüli elemei csak az előjükben különböznek**. Példa: $a = (1, 2, -3) \Rightarrow$

$$M_1 a = \begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 3 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 \\ 2 \\ -3 \end{pmatrix} = \begin{pmatrix} 1 \\ 0 \\ 0 \end{pmatrix}$$

Mátrixok trianguláris felbontásai

LL^T felbontás (Cholesky): Ha a Gauss elimináció során vizsgált mátrix pozitív definit, akkor létezik $U = L^T$ felbontás, tehát $A = LU = LL^T$. A pozitív definit tulajdonságban először a szimmetriát, majd végül a másik feltételt vizsgáljuk meg. A szimmetria gyakran ránézésre eldönthető, a második feltételt gyakorlatban ezt úgy vizsgáljuk meg, hogy a mátrixra képzett összes főminor pozitív. Ha ez a két feltétel teljesül rá, akkor alkalmazható rá ez a felbontás. Ezután elkezdjük magát a felbontást. Szeretnénk tehát egy olyan mátrixot kapni, amit fel tudunk bontani két triangulárisra úgy, hogy az egyik a másik transzponáltja legyen. Erre írunk fel egy egyenletet:

$$\begin{pmatrix} 9 & -6 & 3 \\ -6 & 20 & 18 \\ 3 & 18 & 62 \end{pmatrix} = \begin{pmatrix} l_{11} & 0 & 0 \\ l_{21} & l_{22} & 0 \\ l_{31} & l_{32} & l_{33} \end{pmatrix} \begin{pmatrix} l_{11} & l_{21} & l_{31} \\ 0 & l_{22} & l_{32} \\ 0 & 0 & l_{33} \end{pmatrix}$$

Majd végezzük el a mátrixszorzást a jobb oldalon

$$\begin{pmatrix} 9 & -6 & 3 \\ -6 & 20 & 18 \\ 3 & 18 & 62 \end{pmatrix} = \begin{pmatrix} l_{11}^2 & l_{11}l_{21} & l_{11}l_{31} \\ l_{11}l_{21} & l_{21}^2 + l_{22}^2 & l_{21}l_{31} + l_{22}l_{32} \\ l_{11}l_{31} & l_{21}l_{31} + l_{22}l_{32} & l_{31}^2 + l_{32}^2 + l_{33}^2 \end{pmatrix}$$

Ezután csak behelyettesítjük a megfelelő értékeket a jobb oldali mátrixba, és kiszámoljuk a kapott egyenleteket

$$l_{11} = \sqrt{l_{11}^2} = \sqrt{9} = 3$$

$$l_{11}l_{21} = -6 \Rightarrow 3l_{21} = -6 \Rightarrow l_{21} = -2$$

$$l_{11}l_{31} = 3 \Rightarrow 3l_{31} = 3 \Rightarrow l_{31} = 1$$

$$l_{21}^2 + l_{22}^2 = 20 \Rightarrow 4 + l_{22}^2 = 20 \Rightarrow l_{22}^2 = 16 \Rightarrow l_{22} = 4$$

$$l_{21}l_{31} + l_{22}l_{32} = 18 \Rightarrow -2 + 4l_{32} = 18 \Rightarrow 4l_{32} = 20 \Rightarrow l_{32} = 5$$

$$l_{31}^2 + l_{32}^2 + l_{33}^2 = 62 \Rightarrow 1^2 + 5^2 + l_{33}^2 = 62 \Rightarrow 26 + l_{33}^2 = 62 \Rightarrow l_{33}^2 = 36 \Rightarrow l_{33} = 6$$

Majd ezt visszahelyettesítve megkapjuk tehát a felbontás L háromszögmátrixát

$$L = \begin{pmatrix} 3 & 0 & 0 \\ -2 & 4 & 0 \\ 1 & 5 & 6 \end{pmatrix}$$

Ellenőrzés:

$$LL^T = \begin{pmatrix} 3 & 0 & 0 \\ -2 & 4 & 0 \\ 1 & 5 & 6 \end{pmatrix} \begin{pmatrix} 3 & -2 & 1 \\ 0 & 4 & 5 \\ 0 & 0 & 6 \end{pmatrix} = \begin{pmatrix} 9 & -6 & 3 \\ -6 & 20 & 18 \\ 3 & 18 & 62 \end{pmatrix} = A$$

LU felbontás: lineáris egyenletrendszerek megoldásához az $Ax = b$ alakú egyenletrendszerből eliminációs mátrixokkal olyat szeretnénk kialakítani, amelynek bal oldalán egy felső háromszögmátrix van, így a behelyettesítést egyszerű leolvasással végre tudjuk hajtani.

$$\begin{pmatrix} \times & \times & \times \\ \times & \times & \times \\ \times & \times & \times \end{pmatrix} M_1 \rightarrow \begin{pmatrix} \times & \times & \times \\ 0 & x & x \\ 0 & x & x \end{pmatrix} M_2 \rightarrow \begin{pmatrix} \times & \times & \times \\ 0 & \times & \times \\ 0 & 0 & x \end{pmatrix} M_2 M_1 A$$

$$A \qquad \qquad \qquad M_1 A \qquad \qquad \qquad M_2 M_1 A$$

Itt \times a mátrix szorzás előtti elemeit jelöli, x pedig az utolsó lépésben megváltozott, nem feltétlenül 0 együtthatókat. Ezzel az eljárással kapjuk az LU felbontást (ne tessék aggódni, ha itt még ködös, a példánál minden világos lesz).

Gauss iteráció egy lineáris egyenletrendszerek megoldására szolgáló, Gauss eliminación alapuló iterációs módszer, ami formálisan a következő lépésekkel áll:

- A bemeneti mátrixot LU bontjuk,
- $Ly = b$ megoldása y -ra
- $Ux = y$ megoldása x -re

(matlabban ezt olyan csoda parancsokkal tudjuk megcsinálni, mint az LTriSol, UTriSol, triil, triu)

Példa

Vegyük a következő lineáris egyenletrendszert:

$$\begin{aligned} 2x_1 + 4x_2 - 2x_3 &= 2, \\ 4x_1 + 9x_2 - 3x_3 &= 8, \\ -2x_1 - 3x_2 + 7x_3 &= 10. \end{aligned}$$

Először ezt felírjuk **mátrix formájú alakra**

$$\begin{pmatrix} 2 & 4 & -2 \\ 4 & 9 & -3 \\ -2 & -3 & 7 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 2 \\ 8 \\ 10 \end{pmatrix}$$

Ugye mivel felső háromszög mátrixra szeretnénk bontani a bemeneti mátrixot, ezért jó lenne, ha a főátló alatt csak 0 értékek szerepelnének. Tudván azt, hogy a lineáris egyenletrendszerek megoldásánál a sorok konstansszoros addíciójával / redukciójával az egyenletrendszer helyességén nem változunk, induljunk el ezen az elven: **próbáljuk a generálóelem sorának (pozitív vagy negatív) konstansszorosát hozzáadni az alatta lévő sorokhoz!** Könnyen észrevehető, hogy az első sor generálóelemét (a kettest) $-2x$ kell hozzáadni a második sorhoz, és $1x$ a harmadikhoz, hogy kijöjenek a várt 0-k.

$$M_1 A = \begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 2 & 4 & -2 \\ 4 & 9 & -3 \\ -2 & -3 & 7 \end{pmatrix} = \begin{pmatrix} 2 & 4 & -2 \\ 0 & 1 & 1 \\ 0 & 1 & 5 \end{pmatrix}$$

$$M_1 b = \begin{pmatrix} 1 & 0 & 0 \\ -2 & 1 & 0 \\ 1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 2 \\ 8 \\ 10 \end{pmatrix} = \begin{pmatrix} 2 \\ 4 \\ 12 \end{pmatrix}$$

Láthatjuk, hogy a sor többi eleme is módosult, és eredményül az első oszlopunk már kezd hasonlítani egy trianguláris mátrix első oszlopára. Folytassuk az iterációt a második oszlopon a második elemmel: vonjuk ki a harmadik sorból a második sort!

$$M_2 M_1 A = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} 2 & 4 & -2 \\ 0 & 1 & 1 \\ 0 & 1 & 5 \end{pmatrix} = \begin{pmatrix} 2 & 4 & -2 \\ 0 & 1 & 1 \\ 0 & 0 & 4 \end{pmatrix}$$

$$M_2 M_1 b = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & -1 & 1 \end{pmatrix} \begin{pmatrix} 2 \\ 4 \\ 12 \end{pmatrix} = \begin{pmatrix} 2 \\ 4 \\ 8 \end{pmatrix}$$

Láthatjuk, hogy megkaptunk az eredeti lineáris egyenletrendszerrel ekvivalens felső háromszögmátrixos alakot:

$$\begin{pmatrix} 2 & 4 & -2 \\ 0 & 1 & 1 \\ 0 & 0 & 4 \end{pmatrix} \begin{pmatrix} x_1 \\ x_2 \\ x_3 \end{pmatrix} = \begin{pmatrix} 2 \\ 4 \\ 8 \end{pmatrix}$$

És ebből már tudjuk, hogy:

$$4x_3 = 8 \Rightarrow x_3 = 2$$

$$x_2 + x_3 = 4 \Rightarrow x_2 = 4 - x_3 = 4 - 2 = 2$$

$$2x_1 + 4x_2 - 2x_3 = 2 \Rightarrow x_1 = \frac{2 - 4x_2 + 2x_3}{2} = \frac{2 - 8 + 4}{2} = \frac{-2}{2} = -1$$

tehát $x = (-1, 2, 2)^T$, az LU felbontás:

$$L = L_1 L_2 = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & 0 & 1 \end{pmatrix} \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & 1 & 1 \end{pmatrix}$$

Ezzel maga az A mátrix felbontása:

$$A = \begin{pmatrix} 2 & 4 & -2 \\ 4 & 9 & -3 \\ -2 & -3 & 7 \end{pmatrix} = \begin{pmatrix} 1 & 0 & 0 \\ 2 & 1 & 0 \\ -1 & 1 & 1 \end{pmatrix} \begin{pmatrix} 2 & 4 & -2 \\ 0 & 1 & 1 \\ 0 & 0 & 4 \end{pmatrix} = LU$$

QR felbontás: egy $R^{n \times n}$ reguláris mátrixnak létezik egy olyan QR felbontása, ahol Q egy ortogonális mátrix, R pedig egy felső háromszögmátrix.

Egyenletrendszerek megoldása iterációs módszerekkel

Jacobi iteráció: Ezeket a lineáris egyenletrendszereket más, talán kicsit lényegretörőbb módszerekkel is meg tudjuk oldani, ilyen lehet például a **Jacobi iteráció**. Lépései:

- Fejezzük ki az egyes változókat külön - külön sorokból, és kerüljenek ők egy oldalra
- Vegyük egy indulóvektort, és határozzuk meg így az egyenlet jobb oldali értékét

- majd fogjuk az így kapott vektort, és a következő iterációban használjuk őt indulóvektorként
Vannak esetek, amikor a Jacobi-iteráció nem konvergál (azaz a kapott eredmények nem közelítenek a valós megoldáshoz). Igazából egy egyenletrendszert annyival is el tudunk szűrni, hogy más sorokból fejezzük ki a bizonyos változókat.

Gauss-Seidel iteráció: A Jacobi iteráció sebessége javítható úgy, ha a vektorkomponensek meghatározása után az iterációs képlet jobb oldalán már az új értékeket használjuk (tehát amit már egyszer kifejeztünk, később inkább őt használjuk, ne az eredeti változatát).

$$\begin{aligned}x_1^{k+1} &= \frac{4 + x_2^k - x_3^k}{4} \\x_2^{k+1} &= \frac{1 + 2x_1^{k+1} + x_3^k}{4} \\x_3^{k+1} &= \frac{4 + 2x_1^{k+1} - x_2^{k+1}}{5}\end{aligned}$$

Ez gyorsabb, mint a Jacobi-iteráció: általában kevesebb lépésben kihozza az eredményt.

Az iterációs módszerek egy fontos tulajdonsága, hogy konvergensek-e, azaz kellően sok iterációt végrehajtva közelítjük-e a keresett megoldást. Akkor mondjuk, hogy egy iterációs sorozat **globálisan konvergens**, ha tetszőleges kezdővektorból indulva a módszer a megoldáshoz konvergál.

Mindkét iterációs eljárás pontosan akkor konvergens, ha a $p(B)$ spektrálsugárra (azaz a legnagyobb abszolútértékű sajátérték abszolútértékére) teljesül, hogy $p(B) < 1$.

Mátrixok sajátértékeinek és sajátvektorainak numerikus meghatározása

Egy A mátrix sajátértéke (λ) és sajátvektorára (v) igaz kell, hogy legyen az alábbi állítás (amit egyébként az A mátrix **sajátérték egyenletének** hívunk):

$$Av = \lambda v$$

...amiből következik, hogy

$$Av - \lambda v = 0$$

$$v(A - \lambda I) = 0$$

A második homogén egyenletben (homogén egyenlet: egy olyan egyenlőség, melynek jobb oldalán a 0 vektor található) egy olyan formát kaptunk, amelyik nem helyes, ugyanis a sajátérték egy skalár (ha jobban tetszik, csak egy sima szám, nem egy mátrix), és ez nem vonható ki közvetlen a mátrixból. Át kell rendezni úgy, hogy a sajátérték is valamilyen mátrixos alakú legyen, és ezt megtehetjük úgy, hogy beszorozzuk egy egységmátrixszal:

$$v(A - \lambda I) = 0$$

Ez csak akkor lehet igaz, ha $\det(A - \lambda I) = 0$. Az itt kapott egyenletet **karakterisztikus polinomnak** nevezzük. Ennek kiszámításával megkapjuk az A mátrix sajátértékét.

$$A \text{ mátrix a következő: } \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}$$

$$\begin{aligned} \det(A - \lambda I) &= \det \left(\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} - \begin{pmatrix} \lambda & 0 \\ 0 & \lambda \end{pmatrix} \right) = \det \left(\begin{pmatrix} -\lambda & 1 \\ 1 & -\lambda \end{pmatrix} \right) = \\ &= \lambda^2 - 1 = 0 \implies \lambda_1 = 1 \text{ és } \lambda_2 = -1. \end{aligned}$$

Így már a második egyenletbe történő visszahelyettesítéssel megkapható a sajátvektor értéke is:

$$\begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix} - \begin{pmatrix} \lambda_i & 0 \\ 0 & \lambda_i \end{pmatrix} \begin{pmatrix} x_1^{(i)} \\ x_2^{(i)} \end{pmatrix} = 0 \implies \begin{pmatrix} -\lambda_i & 1 \\ 1 & -\lambda_i \end{pmatrix} \begin{pmatrix} x_1^{(i)} \\ x_2^{(i)} \end{pmatrix} = 0$$

Lambda behelyettesítése...

$$\begin{pmatrix} -1 & 1 \\ 1 & -1 \end{pmatrix} \begin{pmatrix} x_1^{(1)} \\ x_2^{(1)} \end{pmatrix} = 0$$

Egyenletrendszeres forma...

$$\begin{aligned} -x_1^{(1)} + x_2^{(1)} &= 0 \\ x_1^{(1)} - x_2^{(1)} &= 0 \end{aligned}$$

Ebből a két lehetséges sajátvektor értéke...

$$\begin{pmatrix} x_1^{(1)} \\ x_2^{(1)} \end{pmatrix} = \begin{pmatrix} 1 \\ 1 \end{pmatrix} \quad \begin{pmatrix} x_1^{(2)} \\ -x_2^{(2)} \end{pmatrix} = \begin{pmatrix} -1 \\ 1 \end{pmatrix}$$

Összegzés

Közelítő-, és szimbolikus számítások - 2.)

Érintő-, szelő-, és húrmódszer, a konjugált gradiens eljárás. Lagrange interpoláció. Numerikus integrálás.

Alapvető szükséges fogalmak

Egy $f(x)$ függvény **deriváltja** az x pontban található érintő meredeksége

Egy $f(x)$ függvény **differenciálható**, ha az x pontjában jól közelíthető lineáris függvénnel

Az x^* **egyszeres zérushelye** f -nek, ha $f'(x^*) \neq 0$, tehát a függvény deriváltjában x nem zérushely

Az x^* **izolált zérushelye** f -nek, ha $f(x^*) = 0$, és van olyan $\varepsilon > 0$ küszöbszám, hogy az $(x^* - \varepsilon, x^* + \varepsilon)$ intervallumban x^* az egyetlen zérushely.

Egy $f(x)$ függvény **zérushelye** az a pont, ahol metszi az x tengelyt.

A **függvényosztályok** lehetnek például polinomiálisak, lineárisak, exponenciálisak, stb.

A **Vandermonde-mátrix** egy speciális alakú mátrix, amelyben a sorok mértani sorozatot alkotnak (bármelyik tag és az azt megelőző tag hányadosa állandó)

$$V = \begin{bmatrix} 1 & x_1 & x_1^2 & \dots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \dots & x_2^{n-1} \\ 1 & x_3 & x_3^2 & \dots & x_3^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_m & x_m^2 & \dots & x_m^{n-1} \end{bmatrix}$$

A Π szimbólum egy halmaz felett értelmezett összeszorzást jelenti, tehát:

$$\prod_{k=3}^7 k = (3)(4)(5)(6)(7)$$

Newton- (érintő-), szelő-, és húrmódszer

Az egyik legjobb módszer valós függvények zérushelyeinek (gyökeinek) közelítésére. Gyorsan konvergál, ha az iteráció egy gyökhöz elég közelről indul. Tegyük fel, hogy az $f(x) = 0$ egyenlet x^* egyszeres, izolált zérushelyét akarjuk meghatározni, és hogy ennek környezetében $f(x)$ differenciálható. Válasszunk ki a kapott környezetből egy x_0 kezdőértéket, majd képezzük az

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

iterációs sorozatot. Ezt az eljárást nevezzük **érintő-**, vagy **Newton módszernek**. Az iterációs képlet geometriai értelmezése az, hogy az aktuális x_k pontban meghatározzuk a függvénynek és annak deriváltjának értékét, ezekkel képzünk az adott ponthoz húzott érintőt, és a következő iterációs pontnak az **érintő zérushelyét** vesszük.

Egy adott $P(x_0, y_0)$ ponton áthaladó m meredekségű egyenes egyenletét az

$$y - y_0 = m(x - x_0)$$

írja le. Ha éppen a k-adik iterációban járunk, akkor az $(x_k, f(x_k))$ ponthoz tartozó érintőt és annak zérushelyét akarjuk meghatározni. Ezt a következő egyenlettel írjuk le:

$$y - f(x_k) = m(x - x_k)$$

Azt tudjuk, hogy az adott pontba húzott érintő meredeksége az pont a függvény adott pontban értelmezett deriváltjának értéke, tehát $m = f'(x_k)$. Ez alapján az előbbi képlet átírva:

$$y - f(x_k) = f'(x_k)(x - x_k)$$

Mivel az x tengellyel való metszéspontot (zérushelyet) keressük, ezért $y = 0$, tehát elhagyható. Ennek függvényében az előbbi egyenlet átírható az alábbi alakra, amit átrendezve megkapjuk a Newton-módszer képletét (átvisszük $-f(x_k)t$, leosztunk $f'(x_k)$ val, majd kifejezzük x-et):

$$-f(x_k) = f'(x_k)(x - x_k)$$

A Newton-módszer egyik hátránya, hogy szükség van a deriváltak kiszámítására, ami költséges művelet, és ha a függvény egyébként nem ismert (csak behelyettesíteni tudunk, de nincs rá explicit képletünk, vagy csak valami közelítő algoritmusunk van), akkor ki se tudjuk számolni a deriváltat. Ilyenkor alkalmas inkább **érintő módszert** használni.

Ennél a módszernél két kezdőértéket kell választani, legyenek ezek x_0 és x_1 . Az x_{k+1} nem lesz más, mint az $(x_k, f(x_k))$ és $(x_{k-1}, f(x_{k-1}))$ pontokon áthaladó egyenes x tengellyel vett metszéspontjának x koordinátája. Érdemes ezt a módszert tehát egy olyan kezdőértékkel indítani, amelyek a keresett x^* gyököt közrefogják.

A szelőmódszer azon változatát, ahol a kezdeti x_0 és x_1 pontokban az $f(x)$ függvény ellentétes előjelű, **húrmódszernek** nevezzük. minden iterációban azt a x_{k+1} pontot választjuk, amelyikkel ez a tulajdonság fennmarad.

A konjugált gradiens eljárás

Ismert, hogy egy többváltozós függvény a gradiensvektorával ellentétes irányban csökken a leggyorsabban.

Lagrange interpoláció

Azt a feladatot, melynek célja adott (x_i, y_i) , $i = 1, 2, \dots, m$ pontpár sorozatához előállítani egy olyan $f(x)$ függvényt, amely egy előre adott függvényosztályba tartozik, és az x_i alappontokban a hozzájuk tartozó y_i értékeket veszi fel, **interpolációt** nevezünk (ergo pontokra illesztünk egy függvényt). Az interpoláció másik jelentése az, hogy a közelítő függvény értékét becsüljük a legkisebb és legnagyobb x_i által meghatározott intervallumon. Ha az adott pont nincs benne az intervallumban, **extrapolációról** beszélünk.

A **Lagrange interpolációt** akkor használjuk, ha polinommal szeretnénk közelíteni az adott függvényt. Ennél a módszernél feltesszük, hogy az alappontok páronként különböznek (ami valljuk be, jogos, mert minél akarnánk 5 egyenlő pontra közelíteni). Ilyenkor a függvény nem mehet át két $y = f(x)$ értékhez tartozó ponton (tehát ugyanazon a ponton kétszer).

Az interpolációs feltételek egy lineáris egyenletrendszer határoznak meg a keresett a_k együtthatókra vonatkozóan, és pont egy Vandermonde-mátrixszal írhatók le:

$$\begin{pmatrix} 1 & x_1 & x_1^2 & \cdots & x_1^{n-1} \\ 1 & x_2 & x_2^2 & \cdots & x_2^{n-1} \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 1 & x_n & x_n^2 & \cdots & x_n^{n-1} \end{pmatrix} \begin{pmatrix} a_0 \\ a_1 \\ \vdots \\ a_{n-1} \end{pmatrix} = \begin{pmatrix} y_1 \\ y_2 \\ \vdots \\ y_n \end{pmatrix}$$

A Vandermonde-mátrixok determinánsa megkapható az alábbi formulából:

$$\prod_{i>j} (x_i - x_j)$$

Ebből adódik, hogy amennyiben az interpolációs alappontok páronként különböznek (amit egyébként feltételeként már kikötöttünk a legelején), akkor ez pontosan egy **n-1-edfokú interpolációs polinomot fog létrehozni** az alábbi módon:

$$L_i(x) = \prod_{j=1, j \neq i}^n \frac{x - x_j}{x_i - x_j} = \frac{(x - x_1)(x - x_2) \cdots (x - x_{i-1})(x - x_{i+1}) \cdots (x - x_n)}{(x_i - x_1)(x_i - x_2) \cdots (x_i - x_{i-1})(x_i - x_{i+1}) \cdots (x_i - x_n)}$$

Numerikus integrálás

A Numerikus integrálás során a feladat egy f függvény határozott integráljának közelítése az $[a, b]$ intervallumon. Abból indulunk ki, hogy az f függvény határozatlan integrálját (ha van neki egyáltalán) nem ismerjük, viszont adott x -re az $f(x)$ függvényértéket ki tudjuk (bár közelítően) számolni.

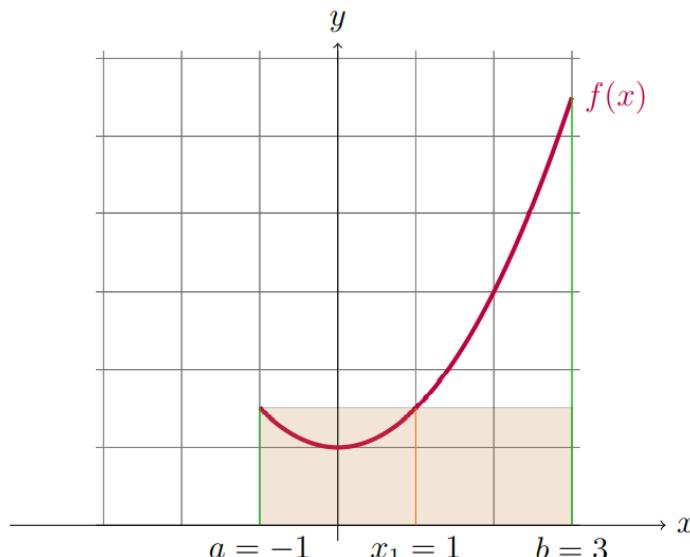
Egy **kvadratúra-formula** az f függvény határozott integráljára az $[a, b]$ intervallumon a következőképpen néz ki:

- Kiszámítunk x_1, \dots, x_n alappontokat, melyek mindegyike az adott intervallumba esik
- Meghatározunk minden x_i alapponthoz egy w_i súlyt,
- Ekkor a kvadratúra-formula értéke az alappontokon felvett függvényértékek w_i szerinti súlyozott összege lesz. Formálisan:

$$Q_n(f) = \sum_{i=1}^n w_i f(x_i)$$

Ez egy lehetőség, hogy csak egy alappontot veszünk, mégpedig az intervallum felezőpontját, és a hozzá rendelt w_1 súlyt az intervallum mérete, tehát $b - a$ lesz. Az ebből előállított formula ábrázolása:

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



Ez a formula a grafikus ábrázoláson látható színes téglalap területét adja, ugyanis ez egyenlő lesz a téglalap hossza ($b - a$) szorozva téglalap magassága ($f((a+b)/2)$). Ennek a módszernek a neve a **téglalap szabály**, és ez a legegyszerűbb kvadratúra-formula.

Észrevehetjük, hogy a téglalap-szabály alkalmazásakor veszünk egy x_1 alappontot a megadott intervallumban, majd arra illesztünk egy polinomot, és ennek a polinomnak vesszük a határozott integrálját.

Ha egy kvadratúra-formula megkapható a következő alakban:

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

Összegzés

Logika és informatikai alkalmazásai - 1.)

Normálformák az ítéletkalkulusban, teljes rendszerek. Következtető módszerek: Hilbert-kalkulus és rezolúció.

Normálformák az ítéletkalkulusban

Hogy ne legyen az egyes algoritmusokra sok eset, ezért általában azokat valamilyen normálformában lévő formulára szoktuk hozni. A **Konjunktív normálforma** az egyik leggyakrabban alkalmazott normálforma:

- Az ítéletváltozókat és negáltjaikat literáloknak nevezzük (pl.: p , $\neg q$, $\neg r$, r)
- Véges sok literál diszjunckióját (logikai „vagy”) klóznak nevezzük
- pl.: $(p \vee \neg q)$, $(\neg p \vee \neg q \vee r)$, p
- Véges sok klóz konjunkcióját pedig konjunktív normálformának, azaz CNF-nek nevezünk

CNF-re hozás: minden formula egy vele ekvivalens CNF-re hozható. Lépések:

- Először a \rightarrow és \leftrightarrow konnektívákat elimináljuk a formulából a következő ekvivalenciákkal:
 - $F \rightarrow G \equiv \neg F \vee G$
 - $F \leftrightarrow G \equiv (\neg F \vee G) \wedge (F \vee \neg G)$
- Majd a \neg jeleket visszük le a változók mellé a deMorgan azonosságokkal:
 - $\neg(F \vee G) \equiv \neg F \wedge \neg G$
 - $\neg(F \wedge G) \equiv \neg F \vee \neg G$
 - $\neg\neg F \equiv F$

Ekkor a formula **negációs normálformában**, azaz **NNF-ben** van.

- Végül a \vee jeleket visszük be a \wedge jelek alá a disztributivitás alkalmazásával:
 - $F \vee (G \wedge H) \equiv (F \vee G) \wedge (F \vee H)$
 - $(F \wedge G) \vee H \equiv (F \vee H) \wedge (G \vee H)$

Egy CNF-eken értelmezett ismert probléma a SAT (inputban kap egy CNF-et, outputban pedig legenerálja, hogy az kielégíthető-e, vagy sem. Egy formula akkor kielégíthető, ha létezik a literáljainak olyan értékkombinációja, amihez a formula igazra értékelődik ki). Egy lehetséges megoldás erre az, hogy fogjuk a formulát, és felírjuk a klózainak halmazaként, a klózokat meg a literáljainak halmazaként. Itt fontos megemlíteni még, hogy ha az inputban még nincs is, az algoritmusok simán generálhatnak egyes esetekben üres klózt: ez minden értékkombináció mellett **hamis**, jele \square . Ezzel ellentétben az üres CNF minden értékkombináció mellett **igaz**, jele \emptyset (később a rezolúciónál ez lesz a kiinduló gondolatunk).

Ha ℓ egy literál, akkor $\bar{\ell}$ jelöli ℓ komplementerét: $\bar{p} = \neg p$ és $\bar{\neg p} = p$.

CNF-ek megszorítására is van lehetőség, akár csak a Boole-függvényeknél. Ezt az alábbi eljárással tudjuk megtenni:

Ha Σ klózok egy halmaza és ℓ egy literál, akkor a $\Sigma|_{\ell=1}$ is egy klózhalmaz, mégpedig:

- Σ -ból elhagyjuk az ℓ -t tartalmazó klózokat,
- az eredmény klózaiból pedig elhagyjuk az $\bar{\ell}$ -eket.

Legyen \mathcal{A} értékkadás, Σ CNF és ℓ literál.

$\mathcal{A} \models \Sigma$ pontosan akkor igaz, ha

- $\mathcal{A}(\ell) = 1$ és $\mathcal{A} \models \Sigma|_{\ell=1}$
- vagy $\mathcal{A}(\ell) = 0$ és $\mathcal{A} \models \Sigma|_{\ell=0}$.

Röviden, ha $\mathcal{A} \models \Sigma|_{\ell=\mathcal{A}(\ell)}$.

Mert

Ha $\mathcal{A}(\ell) = 1$, akkor \mathcal{A} kielégíti az összes Σ -beli klózt, melyekben ℓ szerepel.

A többi klózból a $\bar{\ell}$ literál értéke \mathcal{A} mellett hamis, elhagyható belőlük.

A kapott klózhalmaz épp $\Sigma|_{\ell=1}$.

(Az $\ell = 0$ eset is kész ezzel, hiszen az ugyanaz, mint az $\bar{\ell} = 1$ eset.)

pl: Ha $\Sigma = \{\{p\}, \{p, q\}, \{\neg p, \neg q\}, \{p, \neg p, r\}\}$ és $I = \neg p$, akkor $\Sigma|_{\neg p=1} = \{\square, \{q\}\}$.

A keresési tér vágása:

Unit propagation:

Ha van egysékklóz a CNF-ben, az kényszeríti a benne lévő változók értékét, azaz ha van Σ -ban egy $\{l\}$ egységgklóz, akkor Σ minden A modelljében $A(l) = 1$.

Tehát ekkor pontosan akkor kielégíthető Σ , ha $\Sigma|_{l=1}$ az, mivel $\Sigma|_{l=0}$ -ban lesz egy üres klóz, így kielégíthetetlen.

Pure literal elimination:

Ha l olyan literál, melynek komplementere nem fordul elő Σ -ban, akkor Σ pontosan akkor kielégíthető, ha $\Sigma|_{l=1}$ az.

Tehát, ha l komplementere nem szerepel Σ -ban, akkor l -t biztonsággal 1-re állíthatjuk.

DPLL algoritmus

Ha az input formula üres CNF, akkor return **true**.

Ha az input formulában van üres klóz, akkor return **false**.

Különben:

- Ha van egységgklóz, akkor azt 1-re állítjuk
- Ha van l literál, amelynek komplementere nem szerepel a formulában, akkor $l=1$
- Ha egyik sem a fentiek közül, akkor választunk egy p változót és rekurzívan kiértékeljük $p=0$ -ra és $p=1$ -re.

Teljes rendszerek

Boole-függvények egy H rendszere teljes vagy adekvát, ha minden n -változós függvény előáll

- a projekciókból (π_i jelölte az i -edik változó kiválasztását)
- és H elemeiből
- alkalmas kompozícióval

Kompozíció: ha f/n és $g_1/k, \dots, g_n/k$ Boole-függvények, akkor az $f \circ \langle g_1, \dots, g_n \rangle$ az a k -változós Boole-függvény, melyre

$$(f \circ \langle g_1, \dots, g_n \rangle)(x_1, \dots, x_k) = f(g_1(x_1, \dots, x_k), \dots, g_n(x_1, \dots, x_k))$$

Hogy H teljes, az pont azt jelenti, hogy ha vesszük azt az O operátort, melyben a projekciók és H elemei szerepelnek, akkor erre $O^*(\emptyset)$ tartalmazza az összes Boole-függvényt. Tehát megint egy műveletcsaláddal definiált lezárási operátorról van szó.

Ahogy erről korábban is szó esett, a Boole-függvényekre is tudunk megszorításokat kikötni. Ha $b \in \{0, 1\}$ igazságérték, akkor $f|_{x_n=b}$ jelöli azt az $(n-1)$ -változós Boole-függvényt, amelyet úgy kapunk, hogy f inputjában x_n értékét b -re rögzítjük.

Formálisan

$$f|_{x_n=b}(x_1, \dots, x_{n-1}) := f(x_1, \dots, x_{n-1}, b).$$

Például

- $\vee|_{x_2=1}$ a konstans 1 függvény: $\vee|_{x_2=1}(x_1) = \vee(x_1, 1) = 1$.
- $\wedge|_{x_2=0}$ a konstans 0 függvény
- $\wedge|_{x_2=1}$ az identikus ($x_1 \mapsto x_1$) függvény

stb.

Shannon expanzió szerint minden Boole-függvény előáll a projekciók és a $\{\neg, \vee, \wedge\}$ Boole-függvények alkalmas kompozíciójaként. Tehát minden Boole-függvény indukálható olyan formulával, melyben csak a $\{\neg, \vee, \wedge\}$ konnektívák szerepelnek, és ez n szerinti teljes indukcióval bizonyítható:

- Ha $n = 0$, akkor $f/0$ az vagy konstans 0, vagy konstans 1
- Ha $n > 0$, akkor az indukciós feltevés szerint az $f|_{x_n=b}(x_1, \dots, x_{n-1})$ Boole-függvények előállnak ilyen alakban, és a Shannon expanzióban szintén csak ezt a három műveletet használjuk

Fontos megjegyezni, hogy a $\{\neg, \vee, \wedge\}$ rendszer például teljes.

Hilbert-kalkulus

Hilbert rendszere egy deduktív (következtető) rendszer: az input Σ összes következményét (és csak azokat) lehet vele levezetni. Ebben a rendszerben csak a \rightarrow konnektívát és a \downarrow (azonosan hamis) logikai konstanst használjuk az ítéletváltozókon kívül. minden formula ilyen alakra hozható, ugyanis a $\{\rightarrow, \downarrow\}$ rendszer teljes. Három **axiómát** tartalmaz a rendszer, amelyek mind tautológiák:

$$\text{Ax1: } (F \rightarrow (G \rightarrow H)) \rightarrow ((F \rightarrow G) \rightarrow (F \rightarrow H))$$

$$\text{Ax2: } F \rightarrow (G \rightarrow F)$$

$$\text{Ax3: } ((F \rightarrow \downarrow) \rightarrow \downarrow) \rightarrow F$$

Mivel \rightarrow nem asszociatív, ezért fontos a sorrend (ergo ha kicsit átrendezzük a formulákat, akkor azok már nem feltétlenül lesznek tautológiák)! Ezeknek az axiómáknak megadható **példány** is, amit úgy képzünk, hogy **F**, **G**, **H** helyére tetszőleges formulát írunk.

A **leválasztási következtetés**, vagy **modus ponens** egy helyes következtetési szabály, amelynek segítségében le tudunk vezetni formulákat Hilbert rendszerében.

$$\{F, F \rightarrow G\} \vdash G$$

Azt mondjuk, hogy egy **F** formula levezethető Σ -ból Hilbert rendszerében ($\Sigma \vdash F$), ha létezik olyan F_1, F_2, \dots, F_n formula-sorozat, melynek minden eleme

- Σ -beli, vagy
- axiómapéldány, vagy
- előáll két korábbiból modus ponenssel

Példa: $\{p \rightarrow q\} \vdash p \rightarrow (r \rightarrow q)$

1. $(p \rightarrow (q \rightarrow (r \rightarrow q))) \rightarrow ((p \rightarrow q) \rightarrow (p \rightarrow (r \rightarrow q)))$	Ax1[F/p, G/q, H/(r \rightarrow q)]
2. $(q \rightarrow (r \rightarrow q))$	Ax2[F/q, G/r]
3. $(q \rightarrow (r \rightarrow q)) \rightarrow (p \rightarrow (q \rightarrow (r \rightarrow q)))$	Ax2[F/(q \rightarrow (r \rightarrow q)), G/p]
4. $p \rightarrow (q \rightarrow (r \rightarrow q))$	MP(2, 3)
5. $(p \rightarrow q) \rightarrow (p \rightarrow (r \rightarrow q))$	MP(1, 4)
6. $p \rightarrow q$	$\in \Sigma$
7. $p \rightarrow (r \rightarrow q)$	MP(5, 6)

A visszavezetés lényege az, hogy axiómapéldányokkal és a levezetés bal oldalán található formulák segítségével állítsuk elő a jobb oldalon látható formulát.

Rezolúció

Input: klózok Σ halmaza.

Output: kielégíthetetlen-e Σ ?

Algoritmus:

- Listát vezetünk a klózokról
- Egy klózt felvehetünk, ha
 - Σ -beli vagy
 - két, a listán már szereplő klóz rezolvense
- Ha az üres klóz kerül a listára, Σ kielégíthetetlen
- Ha már nem tudunk új klózt felvenni, és nincs köztük \square , Σ kielégíthető

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

1. $\{p, q, r\}$	$\in \Sigma$
2. $\{\neg p, r\}$	$\in \Sigma$
3. $\{q, r\}$	Res(1, 2)
4. $\{\neg q, s\}$	$\in \Sigma$
5. $\{r, s\}$	Res (3, 4)
6. $\{\neg r, s\}$	$\in \Sigma$
7. $\{s\}$	Res (5, 6)
8. $\{\neg s\}$	$\in \Sigma$
9. \square	Res(7, 8)

Σ kielégíthetetlen, mivel az üres klóz kihozható.

A rezolúciós algoritmus **helyes**, ami azt jelenti, hogy ha kielégíthetetlen válasszal áll meg, akkor az input Σ tényleg kielégíthetetlen.

A rezolúciós algoritmus **teljes**, ami azt jelenti, hogy ha Σ kielégíthetetlen, akkor minden "kielégíthetetlen" választ ad.

Egy lépésben **egyszerre csak egy literál mentén rezolválunk**, ugyanis ha több literál mentén tennénk ezt, az nem helyes következtetés!

Ennek egy másik módja a **lineáris rezolúció**, ahol minden előző lépésben felvett klózhöz veszünk hozzá egyet a Σ klózai közül, és azokat rezolváljuk, a többi minden ugyanaz.

Egy formulára akkor mondhatjuk, hogy Horn-formula, ha benne minden klóz Horn-klóz. Egy klóz akkor lesz Horn-klóz, ha benne legfeljebb egy pozitív literál szerepel. Itt rezolúció közben minden úgy kell felvennünk a klózokat, hogy minden képzésben az egyik résztvevő klóz pozitív egységekklóz legyen.

Végezetül létezik még egy SLD (Selective Linear Definite) rezolúció is, ami lényegében egy olyan lineáris rezolúció, ahol Horn formulából indulunk ki és nem rezolválunk a listán korábban szereplő klózokkal.

Összegzés

Normálformák az ítéletkalkulusban

- konjunktív normálforma az egyik leggyakrabban alkalmazott normálforma
- az ítéletváltozókat és negáltjaikat literáloknak nevezzük (pl.: p, $\neg q$, $\neg p$, r)
- véges sok literál diszjunckióját (logikai „vagy”) klóznak nevezzük pl.: $(p \vee \neg q)$, $(\neg p \vee \neg q \vee r)$
- véges sok klóz konjunkcióját pedig konjunktív normálformának, azaz CNF-nek nevezünk
- CNF-re hozás: minden formula egy vele ekvivalens CNF-re hozható
- először a \rightarrow és \leftrightarrow konnektívákat elimináljuk a formulából a következő ekvivalenciákkal:
 - $F \rightarrow G \equiv \neg F \vee G$
 - $F \leftrightarrow G \equiv (\neg F \vee G) \wedge (F \vee \neg G)$
- majd a \neg jeleket visszük le a változók mellé a deMorgan azonosságokkal:
 - $\neg(F \vee G) \equiv \neg F \wedge \neg G$
 - $\neg(F \wedge G) \equiv \neg F \vee \neg G$
 - $\neg\neg F \equiv F$
- ekkor a formula **negációs normálformában**, azaz NNF-ben van.
- Végül a \vee jeleket visszük be a \wedge jelek alá a disztributivitás alkalmazásával:
 - $F \vee (G \wedge H) \equiv (F \vee G) \wedge (F \vee H)$
 - $(F \wedge G) \vee H \equiv (F \vee H) \wedge (G \vee H)$
- SAT (inputban kap egy CNF-ét, outputban pedig generálja, hogy az kielégíthető-e)
- ha az inputban még nincs is, az algoritmusok simán generálhatnak egyes esetekben üres klózt: ez minden értékadás mellett hamis, jele \square .
- az üres CNF minden értékadás mellett igaz, jele \emptyset
- CNF-ek megszorítására is van lehetőség, akár csak a Boole-függvényeknél

Teljes rendszerek

- Boole-függvények egy H rendszere teljes (adekvát), ha minden n-változós függvény előáll
 - a projekciókból
 - és H elemeiből
 - alkalmas kompozícióval
- ha H teljes, az pont azt jelenti, hogy ha vesszük azt az O operátort, melyben a projekciók és H elemei szerepelnek, akkor erre $O^*(\emptyset)$ tartalmazza az összes Boole-függvényt
- a Boole-függvényekre is tudunk megszorításokat kikötni. Ha $b \in \{0, 1\}$ igazságérték, akkor $f|_{X_{n=b}}$ jelöli azt az $(n-1)$ -változós Boole-függvényt, amelyet úgy kapunk, hogy f inputjában x_n értékét b-re rögzítjük.
- Shannon expanzió minden Boole-függvény indukálható olyan formulával, melyben csak a $\{\neg, \vee, \wedge\}$ konnektívák szerepelnek, és ez n szerinti teljes indukcióval bizonyítható
- a $\{\neg, \vee, \wedge\}$ rendszer teljes

Hilbert rendszere

- egy deduktív (következtető) rendszer
- az input Σ összes következményét (és csak azokat) lehet vele levezetni
- csak a \rightarrow konnektívát és a \downarrow (azonosan hamis) logikai konstanst használjuk
- minden formula ilyen alakra hozható, ugyanis a $\{\rightarrow, \downarrow\}$ rendszer teljes
 - Ax1: $(F \rightarrow (G \rightarrow H)) \rightarrow ((F \rightarrow G) \rightarrow (F \rightarrow H))$
 - Ax2: $F \rightarrow (G \rightarrow F)$
 - Ax3: $((F \rightarrow \downarrow) \rightarrow \downarrow) \rightarrow F$
- \rightarrow nem asszociatív, ezért fontos a sorrend

- az axiómáknak megadható példány is, amit úgy képzünk, hogy F, G, H helyére tetszőleges formulát írunk
- leválasztási következtetés (modus ponens) egy helyes következtetési szabály, amelynek segítségében le tudunk vezetni formulákat Hilbert rendszerében.
- F formula levezethető Σ -ból Hilbert rendszerében, ha létezik olyan formula-sorozat, melynek minden eleme Σ -beli, axiómapéldány, vagy előáll ezekből modus ponenssel

Rezolúció

- Input: klózok Σ halmaza.
- Output: kielégíthetetlen-e Σ ?
- halmazt vezetünk a klózokról
- Egy klózt felvehetünk, ha
 - Σ -beli vagy
 - két, a listán már szereplő klóz rezolvense
- Ha az üres klóz kerül a listára, Σ kielégíthetetlen
- Ha már nem tudunk új klózt felvenni, és nincs köztük \square , Σ kielégíthető
- **helyes:** ha kielégíthetetlen válasszal áll meg, akkor az input Σ tényleg kielégíthetetlen
- **teljes:** ha Σ kielégíthetetlen, akkor minden esetben "kielégíthetetlen" választ ad
- egy lépésben egyszerre csak egy literál mentén rezolválunk,
- lineáris rezolúció: minden lépésben felvett klózhoz veszünk hozzá egyet a Σ klózai közül, és azokat rezolváljuk
- Horn-formula, minden klóz Horn-klóz
- Horn-klóz, legfeljebb egy pozitív literál van benne
- rezolúció közben minden képzésben az egyik résztvevő klóz pozitív egységekkel legyen
- SLD (Selective Linear Definite) rezolúció: egy olyan lineáris rezolúció, ahol minden lépésben csak egy literál mentén rezolválunk a listán korábban szereplő klózokkal.

Logika és informatikai alkalmazásai - 2.)

Normálformák a predikátumkalkulusban. Egyesítési algoritmus. Következtető módszerek: Alap rezolúció, elsőrendű rezolúció.

Normálformák a predikátumkalkulusban

Első lépésként tisztázzuk, hogy mi is a **predikátumkalkulus (elsőrendű logika)**. A **változók** objektumok, amik egy **univerzumából vagy halmazából kapnak értékeket** (stringek, természetes számok). A konnektívákon kívül **kvantorok is használhatóak** (mint \forall vagy \exists). Az **objektumokat függvények transzformálhatják más objektumokba** (összeadás, stringek összefűzése), és ezekből **az objektumokból predikátumok képeznek igazságértéket** (az adott szám páros-e).

Szintaxis:

- elsőrendű változók: $x, y, z, \dots, x_1, y_5, \dots$
- függvényjelek: f, g, \dots
- predikátumjelek: p, q, r, \dots
- konnektívák: $\wedge, \vee, \rightarrow, \leftrightarrow, \neg$
- kvantorok: \forall, \exists
- logikai konstansjelek: \uparrow, \downarrow

Minden függvényjelnek és predikátumjelnek van egy **aritása**, avagy **rangja** és változószáma. Ha f egy n -változós jel, ezt f/n jelöli. A 0-arithás függvényjeleket konstansjelnek nevezzük, a 0-arithás predikátumjeleket pedig logikai változóknak.

Elsőrendű logikában két szintaktikus kategória van: a termek és a formulák halmaza. Kiértékeléskor a termek vesznek fel objektumot értékként, a formulák pedig igazságértéket.

Termek: minden változó term; ha f/n függvényjel, t_1, \dots, t_n pedig termek, akkor $f(t_1, \dots, t_n)$ is term.

Egy term értékét egy meghatározott struktúrában úgy kapjuk meg, hogy ha a term egy változó, akkor annak értékét φ szabja meg. Ha a term egy függvény, akkor a struktúrájában lévő termeket előbb rekurzívan kiértékeljük, és a kapott értékeket visszahelyettesítjük.

Formulák: 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. Ha F és G formulák, akkor $F \vee G, F \wedge G, F \rightarrow G, F \leftrightarrow G$ is az. \downarrow és \uparrow is formulák. Ha F formula és x változó, akkor $\exists x F$ és $\forall x F$ is formulák. Egy formulát mondatnak vagy zárt formulának nevezünk, ha nem szerepel benne szabadon változó (tehát olyan változó, amit nem köt legalább egy kvantor).

Struktúrák: egy elsőrendű formulát egy struktúrában értékelünk ki, ami egy olyan (A, I, φ) számhármas, ahol:

- A egy nemüres halmaz, az univerzum, az alaphalmaz
- φ a változók default értékadása, minden x változóhoz egy $\varphi(x) \in A$ objektumot rendel
- I az interpretációs függvény, ez rendel a függvény-, és predikátumjelekhez szemantikát az adott struktúrában

Például: ha $+/2, \times/2, /1$ függvényjelek, $0/0$ és $1/0$ konstansjelek, x és y pedig változók, akkor egy struktúra lehet pl. $\mathcal{A} = (\mathbb{N}_0, I, \varphi)$, ahol

- $\mathbb{N}_0 = \{0, 1, 2, \dots\}$ a természetes számok halmaza;
- $\varphi(x) = 2, \varphi(y) = 3, \varphi(z) = 6, \dots$
- $I(+), I(\times)$ és $I(/')$ rendre az összeadás, szorzás és az $n \mapsto n + 1$ rákövetkezés függvények,
- $I(0) = 0$ és $I(1) = 1$.

Most, hogy már megismerkedtünk a predikátumkalkulussal, térjünk is rá a normálformákra:

Zárt Skolem alak: Hogy egy formulával dolgozni tudunk, zárt Skolem alakra kell hozni. Ezt öt lépésekben érhetjük el:

- Nyilak eliminálása (a szokásos módon)
- Kiigazítás (ne legyen változónév-ütközés)
- Prenex alakra hozás (összes kvantor előre kerül)
- Skolem alakra hozás (akkor az összes kvantor elől lesz, és minden univerzális)
- Lezáras (ne maradjon szabad változó)

Nyilak eliminálása: az implikáció (\rightarrow) és az ekvivalencia (\leftrightarrow) műveletek ekvivalens átalakítása:

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

Kiigazítás: meg kell keresnünk, hogy a formula tartalmaz-e olyan változókat, amelyeket több kvantor is köti. Egy kvantor (\forall , \exists) akkor köti egy változót, ha az utána következik bárhol a formulában. Ezt úgy érjük el, hogy ha egy változóra ez előfordul, akkor azokat indexeléssel egyedivé tesszük.

- $\exists x p(x, y) \wedge \forall x q(f(x), x) \wedge \exists y p(y, f(y))$ nem kiigazított, ugyanis
 - x-et két kvantor is köti
 - y van kötötten és kötetlenül is
- $\exists x_1 p(x_1, y) \wedge \forall x_2 q(f(x_2), x_2) \wedge \exists y_1 p(y_1, f(y_1))$ viszont már jó lesz

Prenex alak: egy formula akkor van prenex alakban, ha a benne lévő összes kvantor elől van. Ez igazából nem egy rocket science, mivel kiigazítottuk a formulánkat, minden kvantor ki tud lépkedni az alábbi ekvivalenciák alapján a formula elejére különösebb probléma nélkül:

$\neg \exists x F \equiv \forall x \neg F$	
$\neg \forall x F \equiv \exists x \neg F$	
$\exists x F \vee G \equiv \exists x (F \vee G)$	ha x nem szerepel G-ben szabadon
$\exists x F \wedge G \equiv \exists x (F \wedge G)$	ha x nem szerepel G-ben szabadon
$\forall x F \vee G \equiv \forall x (F \vee G)$	ha x nem szerepel G-ben szabadon
$\forall x F \wedge G \equiv \forall x (F \wedge G)$	ha x nem szerepel G-ben szabadon
$F \vee \exists x G \equiv \exists x (F \vee G)$	ha x nem szerepel F-ben szabadon
$F \wedge \exists x G \equiv \exists x (F \wedge G)$	ha x nem szerepel F-ben szabadon
$F \vee \forall x G \equiv \forall x (F \vee G)$	ha x nem szerepel F-ben szabadon
$F \wedge \forall x G \equiv \forall x (F \wedge G)$	ha x nem szerepel F-ben szabadon

Skolem alak: egy olyan prenex alak, amelyben nincs már egzisztenciális kuantor. Ezt úgy érjük el, hogy az ekkor már formula elején szereplő $\exists x$ -eket töröljük, és a magban a kötött változó helyére (x) egy új függvényjelet vezetünk be, ami paramétereit az őt megelőző univerzális kuantorok által kötött változók lesznek. Tehát:

- $\exists x \forall y \forall z \exists v \exists w (p(x, y, f(z)) \wedge \neg q(x, f(v), w) \wedge p(c, v))$
- $\forall y \forall z (p(d, y, f(z)) \wedge \neg q(d, f(g(y, z)), h(y, z)) \wedge p(c, g(y, z)))$

Habár nem minden F formulához lehet megadni skolem alakot, minden F formulához tudunk egy olyan F' skolem alakú formulát generálni, ami pontosan akkor kielégíthető, ha F is az (tehát F és F' **sekvivalensek**).

Lezárási: az előző lépésekben megkaptuk a skolem alakot, ezt úgy zárjuk le, hogy minden szabad x változó előfordulása helyett (tehát azok helyett, amiket véletlenül egy kvantor se köt) egy új c_x konstansjelet vezetünk be úgy, hogy minden formulában az összes szabad x helyére ugyanazt a c_x -et írjuk.

Egyesítési algoritmus

Mi is az egyesítés? Ha $C = \{l_1, \dots, l_n\}$ literálok egy halmaza (tehát C egy klóz), akkor C egyesíthető, ha létezik olyan s egyesítője, amelyre $l_1s = \dots = l_ns$ (tehát ha tudunk a változóknak olyan értékeket adni, hogy azok egyenlőek legyenek). **Például**:

$C = \{p(x, f(y)), p(g(y), z)\}$, itt látható, hogy a két literál: $p(x, f(y))$
 $p(g(y), z)$

Tehát mondjuk itt ha x helyére behelyettesítjük $g(y)$ -t, és z helyére $f(y)$ -t, akkor a két literál láthatóan pont ugyanaz lesz, ezért egy lehetséges egyesítő lehet az $s = \{x / g(y), z / f(y)\}$.

Algoritmus:

Input: egy C klóz

Output: ha egyesíthető, akkor a legáltalánosabb egyesítőjét adja vissza, különben mondja meg, hogy nem egyesíthető

Erre adható egy kellemes rekurzív megvalósítás:

Incializáljuk s -t üresre ($s := []$);

Kilépési feltétel: C -ben már nincs több vizsgálandó literál ($|C| \leq 1$), ekkor adjuk vissza s -t

Minden iterációban vegyük két különböző literált C -ból, és keressük meg az első eltérő pozíciójukat. Ha itt az egyik literálban egy x változó áll, a másikban pedig egy t term, **melyben nincs x** , akkor az x változó helyére írjuk be a t termet, és adjuk azt hozzá az s -hez, különben nem egyesíthető.

Példa:

$C = \{\neg p(f(z, g(a, y)), h(z)), \neg p(f(f(u, v), w), h(f(a, b)))\}$

$s = []$

Az első eltérő pozíció z és $f(u, v)$: ez teljesen oké, z olyan változó, ami nincs $f(u, v)$ -ban

$s = [z / f(u, v)] \rightarrow C = \{\neg p(f(f(u, v), g(a, y)), h(f(u, v))), \neg p(f(f(u, v), w), h(f(a, b)))\}$

A következő eltérő pozíció a $g(a, y)$ és w lesz: ezzel sincs baj, $g(a, y)$ nem tartalmazza w -t

$s = [z / f(u, v), w / g(a, y)] \rightarrow C = \{\neg p(f(f(u, v), g(a, y)), h(f(u, v))), \neg p(f(f(u, v), g(a, y)), h(f(a, b)))\}$

Az utolsó eltérő pozíció $f(u, v)$ és $f(a, b)$: iss sincs baj, u -t és v -t be tudjuk kötni a -ra és b -re

$s = [z / f(u, v), w / g(a, y), u / a, v / b] \rightarrow$

$C = \{\neg p(f(f(a, b), g(a, y)), h(f(a, b))), \neg p(f(f(a, b), g(a, y)), h(f(a, b)))\}$

Következtető módszerek

Input: elsőrendű formulák egy Σ halmaza

Output: ha Σ kielégíthetetlen, az algoritmus azt véges sok lépéssben levezeti. Ha kielégíthető, akkor vagy levezeti ezt, vagy végtelen ciklusba esik.

Alap rezolúció (ground rezolúció)

Σ elemeit zárt Skolem alakra hozzuk, a kapott formulág magját CNF-re (legyen ez Σ'). Ekkor legyen $E(\Sigma')$ a klózok alap példányainak halmaza, amin futtatjuk az ítéletkalkulusbeli rezolúciós algoritmust.

Példa: $F = \forall x p(x) \wedge \exists y (p(y) \rightarrow q(f(y))) \wedge \forall z \neg q(z)$

Skolem alakra hozzuk:

1. Nyílak elhagyása: $\forall x p(x) \wedge \exists y (\neg p(y) \vee q(f(y))) \wedge \forall z \neg q(z)$
2. Kiigazítás: nem kell, egy változót sem köt több kvantor
3. Prenex: $\forall x p(x) \wedge \exists y \forall z ((\neg p(y) \vee q(f(y))) \wedge \neg q(z)) \rightarrow \forall x \exists y \forall z (p(x) \wedge (\neg p(y) \vee q(f(y))) \wedge \neg q(z))$
4. Skolem $\forall x \forall z (p(x) \wedge (\neg p(g(x)) \vee q(f(g(x)))) \wedge \neg q(z))$
5. Lezárás: nem kell, nincs kötetlen változó

A	B	C
$\Sigma = \{ \{p(x)\}, \{\neg p(g(x)), q(f(g(x)))\}, \{\neg q(z)\} \}$		
$T_0 = \{ c, f(c), g(c), f(f(c)), f(g(c)) \dots \}$		
1. $\{\neg p(g(x)), q(f(g(x)))\}$	$B[x / c]$	
2. $\{p(g(c))\}$	$A[x / g(c)]$	
3. $\{q(f(g(c)))\}$	Res(1, 2)	
4. $\{\neg q(f(g(c)))\}$	$C[z / f(g(c))]$	
5. \square	Res(3, 4)	

Nagy a keresési tér, mivel minden változót helyettesítünk egy-egy alaptermmel. Az alap rezolúciós algoritmus helyes és teljes, tehát pontosan akkor vezethető le egy Σ klózhalmazból az üres klóz, ha az kielégíthetetlen.

Elsőrendű rezolúció

Listát vezetünk a klózokról. Egy klózt felveszünk, ha az Σ -beli, vagy két, már a listán szereplő klóz rezolvense (ez ismerős lehet már a predikátumkalkulus-beli rezolúcióból). Ha az üres klóz kerül a listára, akkor Σ kielégíthetetlen, különben ha már nem tudunk több klózt felvenni, Σ kielégíthető.

Két elsőrendű logikai klóz rezolvensét így kapjuk:

- Átnevezzük a klózban a változókat úgy, hogy a kapott klózok ne tartalmazzanak közös változót
- Kiválasztunk mindenből legalább egy literált,
- Futtatjuk az egyesítési algoritmust az elsőből érkező literálokon, és a másodikból érkezők komplementerén

- Ha C egyesíthető az s legáltalánosabb egyesítővel, akkor végrehajtjuk az egyesítést a nem kiválasztott literálok halmazán, és a kapott klöz lesz a rezolvens

Csak akkor tudunk a kiválasztott literálok mentén egyesíteni, ha a megfelelő literálokból képzett klóz egyesíthető. Ehhez az mindenkiépp kell, hogy az összes kiválasztott literálban ugyanaz legyen a predikátumjel. Ezen felül az is kell, hogy a C-be kerülő literálok előjele megegyezzen. Tehát kiválasztáskor érdemes

- választanunk egy p predikátumjelet
 - az első klózból a pozitív p-s literálokat választani, legalább egyet
 - a második klózból a negatív p-s literálokat választani, legalább egyet
 - és ezeknek az előjel nélküli változatát megpróbálni egyesíteni

Ebben a formában ha összehasonlítjuk az ítéletkalkulus-beli rezolvensképzéssel, ott nem kellett átneveznünk változókat (leginkább azért, mert a klózban nincsenek elsőrendű változók). Kiválasztva egy p predikátumjelet mindenklőzben csak a p és a $\neg p$ szerepelhet p -s literálként, így csak úgy tudunk rezolvenst képezni, ha ha az egyik klózban van p , a másikban meg $\neg p$.

Példa: (itt most kihagyjuk a CNF-es zárt skolem alakra hozást, viszont ehhez is szükséges)

$$\Sigma = \{ \{p(f(x)), \neg q(z), p(z)\}, \{\neg p(x), r(q(x)), a\} \}$$

Itt először látjuk, hogy az mindenbeli klózban van x , tehát nevezzük át pl C_2 -ben az x -eket y -re
 $C_1s_1 = \{p(f(x)), \neg q(z), p(z)\}$ $C_2s_2 = \{\neg p(y), r(q(y), a)\}$

Válasszuk ki mondjuk C_{1s_1} -ből $p(f(x))$ -et és $p(z)$ -t, C_{2s_2} -ből pedig $\neg p(y)$ -t, így kapiuk

$$C = \{p(f(x)), p(z)-t, p(y)-t\}$$

Amit **egyesítenünk** kell. Ha ráküldjük az algoritmust, akkor látjuk, hogy z és y változók helyére simán beírható $f(x)$, tehát akkor a legáltalánosabb egyesítőnk az $s = [z / f(x), y / f(x)]$ lesz. Ezt követően **végrehajtjuk a nem kiválasztott literálokon az s szerinti egyesítést**:

$$\{\neg q(z), r(q(y)), a\} \models [z / f(x), y / f(x)] = \{\neg q(f(x)), r(q(f(x))), a\}$$

Tehát a két klóz rezolvense $\{\neg q(f(x)), r(q(f(x))), a\}$.

Összegzés

Predikátumkalkulus

- elsőrendű változók: $x, y, z, \dots, x_1, y_5, \dots$
- függvényjelek: f, g, \dots
- predikátumjelek: p, q, r, \dots
- konnektívák: $\wedge, \vee, \rightarrow, \leftrightarrow, \neg$
- kvantorok: \forall, \exists
- logikai konstansjelek: \uparrow, \downarrow
- minden változó term; ha f/n függvényjel, t_1, \dots, t_n pedig termek, akkor $f(t_1, \dots, t_n)$ is term
- 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
- ha F és G formulák, akkor $F \vee G, F \wedge G, F \rightarrow G, F \leftrightarrow G$ is az
- \downarrow és \uparrow is formulák
- ha F formula és x változó, akkor $\exists x F$ és $\forall x F$ is formulák
- egy elsőrendű formulát struktúrában értékelünk ki, ami egy (A, I, φ) számhármas, ahol:
 - A egy nemüres halmaz, az univerzum, az alaphalmaz
 - φ a változók default értékadása, minden x változóhoz egy $\varphi(x) \in A$ objektumot rendel
 - I az **interpretációs függvény**, ez rendel a függvény-, és predikátumjelekhez szemantikát az adott struktúrában

Normálformák a predikátumkalkulusban

- Zárt skolem alakra hozás lépései
 - nyilak eliminálása (a szokásos módon)
 - $F \rightarrow G \equiv \neg F \vee G$
 - $F \leftrightarrow G \equiv (\neg F \vee G) \wedge (F \vee \neg G)$
 - kiigazítás (ne legyen változónév-ütközés)
 - többszörösen kötött változók indexelése
 - prenex alakra hozás (összes kuantor előre kerül)
 - skolem alakra hozás (akkor az összes kuantor elől lesz, és minden univerzális)
 - $\exists x$ -eket töröljük, és a magban a kötött változó helyére egy új függvényjelet vezetünk be, ami paramétere az öt megelőző univerzális kuantorok által kötött változók lesznek
 - lezárás
 - ha esetleg maradt olyan változó, amit nem köt egy kuantor sem, akkor azokat cseréljük le egy új konstansra
- Prenex alak: minden kuantor a formula elején van
- Skolem alak: olyan prenex, aminek csak egzisztenciális kuantorai vannak
- Zárt skolem alak: olyan skolem, aminek nincsenek kötetlen változói

Egyesítési algoritmus

- input: egy (vagy több) klóz
- output: legálthatóságban ellenőrzött, vagy az, hogy az input nem egyesíthető
- vegyük két különböző literált C-ből
- keressük meg az első eltérő pozíójukat
- ha az egyik literálban egy x változó áll, a másikban pedig egy t term, melyben nincs x , akkor az x változó helyére írjuk be a t termet, és adjuk azt hozzá az s-hez
- különben nem egyesíthető.

Alaprezolúció

- input elsőrendű formulák egy Σ halmaza
- output Σ kielégíthetetlen, az algoritmus azt véges sok lépésben levezeti. Ha kielégíthető, akkor vagy levezeti ezt, vagy végtelen ciklusba esik.
- Σ elemeit zárt Skolem alakra hozzuk
- a kapott formula magját CNF-re (legyen ez Σ')
- $E(\Sigma')$ a klózok alap példányainak halmaza, amin futtatjuk az ítéletkalkulusbeli rezolúciós algoritmust
- nagy a keresési tér, mivel minden változót helyettesítünk egy-egy alaptermmel
- helyes és teljes: pontosan akkor vezethető le egy Σ klózhalmazból az üres klóz, ha az kielégíthetetlen

Elsőrendű rezolúció

- Két elsőrendű logikai klóz rezolvensét így kapjuk:
 - Átnevezzük a klózban a változókat úgy, hogy a kapott klózok ne tartalmazzanak közös változót
 - Kiválasztunk mindenből legalább egy literált,
 - Futtatjuk az egyesítési algoritmust az elsőből érkező literálokon, és a második klózból az első klóz literáljainak komplementerén
 - Ha C egyesíthető az s legáltalánosabb egyesítővel, akkor végrehajtjuk az egyesítést az összes nem kiválasztott literálon, és a kapott klóz lesz a rezolvens
- helyes és teljes, mindenből piszoknagy bizonyítással
- lift lemma: ha két klóz alap példányainak van R' rezolvense, akkor maguknak a klózoknak létezik olyan elsőrendű rezolvensje, melynek R' alap példánya

Mesterséges intelligencia I - 1.)

Keresési feladat: feladatreprézentáció, vak keresés, informált keresés, heurisztikák. Kétszemélyes zéró összegű játékok: minimax, alfa-béta eljárás. Korlátos kielégítési feladat.

Keresési feladat

Keresési feladatok esetén a feladatot egy **súlyozott gráffal reprezentáljuk**, amit a következő adatok birtokában tudunk felépíteni:

- Lehetséges állapotok halmaza (ezek lesznek a gráf csúcsai)
- Egy kezdőállapot
- Lehetséges cselekvések halmaza, és egy állapotátmenet függvény, amely minden állapothoz hozzárendel egy (cselekvés, állapot) típusú, rendezett párokból álló halmazt (ezek lesznek az élek)
- Állapotátmenet költségfüggvénye, amely minden lehetséges állapot-cselekvés-állapot hármashoz egy $c(x, a, y)$ valós nemnegatív költséget rendel (ezek lesznek a súlyok)
- Célállapotok halmaza (lehetséges állapotok részhalmaza)

Ezt a gráfot állapottérnek nevezzük, továbbá a feladatok reprezentálásakor feltételezzük, hogy az állapotok számossága véges vagy megszámlálható és, hogy egy állapotnak legfeljebb véges számú szomszédja lehet.

Informálatlan (vak) keresés

Ezen stratégiáknak semmilyen információjuk nincs az állapotokról a probléma definíciójában megadott információn kívül. Működésük során mindenkor annyit tudunk tenni, hogy megnézzük a lehetséges következő állapotokat, vagy a rendelkezésünkre álló állapotokat a célállapottal (tehát annyit azért tudunk, hogy ha megtaláltuk a célállapotot, akkor azt felismerjük).

Fakeresés: a kezdőállapotból növesszünk egy fát a szomszédos állapotok hozzávételével, amíg célállapotot nem találunk. Ha ügyesek vagyunk, optimális is lesz.

fakeresés

```

1 perem = { újcsúcs (kezdőállapot) }
2 while perem.nemüres ()
3     csúcs = perem.elsőkivesz ()
4     if csúcs.célállapot () return csúcs
5     perem.beszúr (csúcs.kiterjeszt ())
6 return failure

```

Gráfkeresés: abban az esetben, ha nem fa az állapottér, hanem mondjuk egy gráf, amiben van kör, a fakereséssel könnyen tudunk végtelen ciklusba kerülni. Ennek érdekében vezeti be a gráfkeresés a **zárt halmaz** fogalmát, ahol a már egyszer kiterjesztett csúcokat tároljuk, ha ezekkel találkozunk a későbbiekben, őket már nem vesszük fel újra a verembe.

Szélességi keresés: fogjuk a kezdőcsúcsot, betesszük egy verembe. Ezt követően minden kivesszük a következő elemet a veremből (FIFO szerint), kiterjesztjük, és betesszük a gyerekeit a verembe. Addig keresünk, ameddig van csúcs a veremben, vagy nem találunk célállapotot.

Mélységi keresés: effektíve ugyanazt csináljuk, mint a szélességi keresésnél annyi különbséggel, hogy itt sort használunk a verem helyett. Szintén addig keresünk, ameddig van elem a sorban, vagy ameddig nem találunk végállapotot.

Iteratívan mélyülő keresés: olyan mélységi keresések sorozata, amelyeket lekorlátozzuk egy bizonyos mélységgel. Igaz, hogy az egymást követő iterációkban az első szinteket többször bejárjuk, mégis javítunk a futásidőn. Ez a legjobb (vak) kereső.

Egyenletes költségű keresés: a peremben költség alapú prioritással rendezzük az elemeket, minden iterációban a legkisebb költségű elemet vesszük ki. Az idő és tárigénye nagyban függ a költségfüggvénytől.

Informált keresés és heurisztikák

Az informálatlan keresésnél csak azt tudtuk honnan jövünk, de arról semmit, hogy hová megyünk (ezért vak keresés), így minden szomszéd egyformán jött szóba. Az informált keresésnél a tényleges út hosszán kívül bevezetünk egy új függvényt, a **heurisztikát**. A heurisztika minden állapotból megbecsüli, hogy mekkora az optimális út költsége az adott állapotból egy célállapotba. (pl.: legrövidebb út kiszámolásánál légvonalbeli távolság).

- $h(n)$: optimális költség közelítése n állapotból a legközelebbi célállapotba
- $g(n)$: tényleges költség a kezdőállapotból n-be.

Habár a legrövidebb út heuristikájának előállítására könnyű példát adni, ez a feladat sokszor nem triviális, ugyanakkor nagyon fontos lenne a heurisztika minősége, lehetőleg az legyen elfogadható, és konzisztens, és a heurisztika alulról becsülje az optimális értéket. Hogyan kaphatunk ilyet?

Egy könnyen belátható jó módszer lehet az eredeti probléma **relaxálása**. Egy relaxált probléma optimális megoldása $h()$. Vegyük itt példának a tologató 8-kirakót, és adjunk meg rá pár heurisztikát (csak szomszédos üres helyre tudunk tolni négyzeteket):

$h_1()$: hagyjuk el mondjuk az első megszorítást: bármilyen szomszédos mezőre tudjuk tenni a négyzeteket. Könnyen belátható, hogy ilyenkor minden az éppen elérhető mezőre tesszük a megfelelő négyzetet, tehát az így kapott heurisztika erre az ábrára $h_1() = 8$.

$h_2()$: hagyjuk el a második megszorítást: akkor is tudjuk az adott négyzetet a szomszédos mezőkre tolni, ha azok nem üresek. Ilyenkor elég kiszámolni a rossz helyen lévő négyzetek Manhattan-távolságát (szükséges függőleges és vízszintes lépések száma) az ő megfelelő helyüktől, és a kapott értékeket összeadni, erre az ábrára ez $h_2() = 18$.

7	2	4
5		6
8	3	1

Egyébként az ábrán látható kirakós optimális heurisztikája $h_{\text{opt}}() = 26$.

Igazából itt könnyen észrevehető, hogy az első relaxációt tetszőleges kirakósra minden kisebb értéket fog generálni, mint a második (nyilván a szomszédos mezőre rakás jobban megnehezíti ezt a feladatot, mint a négyzet üressége), ezért azt mondjuk, hogy **h_2 dominálja h_1 -et**. Belátható még az is, hogy a relaxált probléma optimális költsége kisebb, vagy egyenlő lesz az eredeti probléma optimális költségénél, ugyanis az eredeti probléma állapotterre része a relaxáltnak. Ennek a feltételnek is megfelelünk, tehát elfogadható a heurisztika.

Mohó algoritmus: a peremben a rendezést csak a $h()$ heurisztika alapján végezzük el, és minden iterációban a legkisebb értékű csúcsot vesszük ki (hasonló a mélységi kereséshez). Nem optimális, exponenciális idő és tárigénye van és gráfkeresénél az első megtalált út nem feltétlenül optimális. A legrosszabb eset nagyon rossz, de jó heurisztikával javítható.

Az A* algoritmus: a peremben a rendezést nem csak a heurisztika alapján rendezzük, hanem az adott csúcsba eljutáshoz szükséges költség alapján is, tehát $f() = h() + g()$ alapján. Ezt úgy is mondhatjuk, hogy minden állapotra kiszámoljuk a rajta keresztül vezető teljes út lehetséges költségét. Ha a heurisztika konstans 0, akkor magát a Dijkstra algoritmust kapjuk. Ez egy hatékony algoritmus, ugyanis csak azuokat a csúcsokat terjesztjük ki, amelyek kisebbek az optimális költségnél (tehát olyan csúcsokkal nem foglalkozunk, amelyeken keresztül nem is kaphatunk jó megoldást).

Egyszerűsített, memóriakorlátozott A*: adunk egy memóriakorlátot az A*-ra, és addig futtatjuk, ameddig van memória. Ha elfogyott töröljük a legrosszabb értékű utat (több egyenlő esetén a legrégebbit), és a szülőjében eltároljuk, hogy belőle vezetett egy adott költséggel út, és szükség esetén azt újra kiterjesztjük. Érthető, hogy ha az optimális megoldás tárolásához több memóriára van szükség, mint amekkora a korlátozásba beleférne, akkor soha nem fogjuk azt megtalálni. Továbbá hátrány még az is, hogy sok hasonló költségű alternatív út esetén folyton eldobálja, majd újraépíti őket, ha egyszerre nem férnek bele a memóriába.

Kétszemélyes, lépésváltós, determinisztikus, zéró összegű játékok

Kétszemélyes: Az ilyen problémák esetén két ágenst különböztetünk meg egymástól, az egyiket MAX-nak, a másikat MIN-nek nevezzük annak révén, hogy az egyik a hasznosságfüggvény értékét maximalizálni szeretné, a másik pedig minimalizálni.

Lépésváltós: Konvenció szerint a MAX kezd, és amint kiválasztotta, hogy melyik lépést teszi, MIN következik. Ez egészen addig ismétlődik, ameddig egy célállapotot el nem érnek, ekkor a játéknak vége

Determinisztikus: Az ágensek egy meghatározott **stratégia**¹ szerint döntik el, hogy az aktuális állapotból melyik a számukra optimális lépés. A stratégiákból több is létezik, ezekről bővebben később.

Zéró összegű: Az egyik ágens pontosan annyit nyer, mint amennyit a másik veszít.

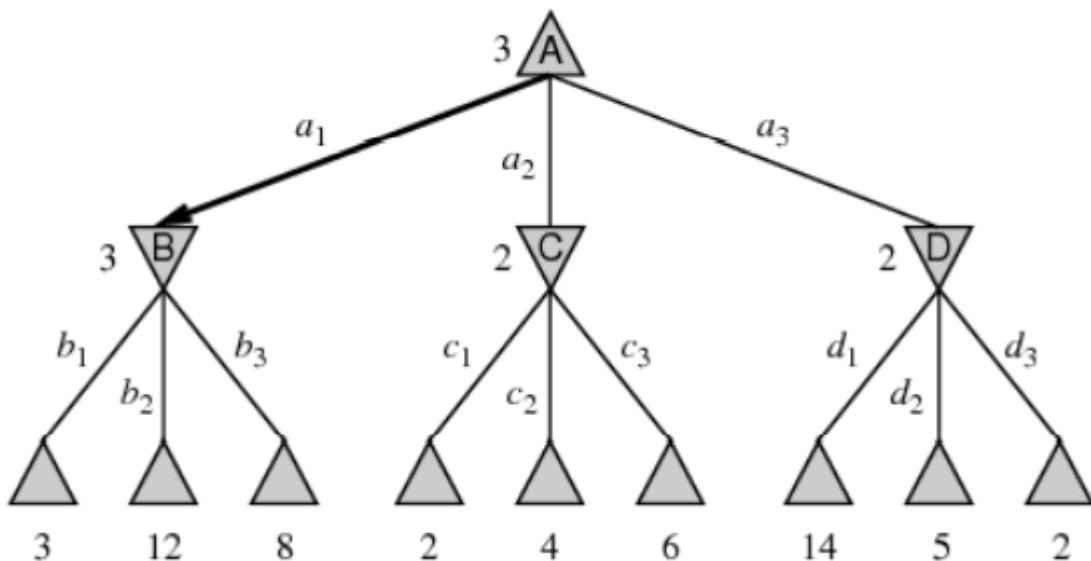
¹ Stratégia: egy olyan algoritmus (eljárás), amely konkrétan meghatározza, hogy a rendelkezésre álló lépések közül az adott ágensnek melyik az optimális, tehát melyiket fogja választani.

MINIMAX algoritmus: egy játéket, és annak bármilyen elérhető állapotát tudjuk gráfokkal reprezentálni. Ennél az algoritmusnál feltesszük, hogy minden játékos ismeri a **teljes** játékgráfot, tetszőlegesen komplex számításokat tud elvégezni, és nem hibázik (érezhető, hogy ezek elég erős feltevések). Ezt más néven **tökéletes racionálitás hipotézisének** is nevezzük. Abban az esetben, ha a tökéletes racionálitás feltehető, minden játékos számára a lehető legjobb stratégiát a MINIMAX algoritmus alapján lehet megvalósítani.

Az algoritmus során minden n csúcsra rekurzívan kiszámoljuk:

$$\text{minimax}(n) = \begin{cases} \text{hasznosság}(n) & \text{ha végállapot} \\ \max_{\{a \text{ } n \text{ szomszédja}\}} \text{minimax}(a) & \text{ha MAX jön} \\ \min_{\{a \text{ } n \text{ szomszédja}\}} \text{minimax}(a) & \text{ha MIN jön} \end{cases}$$

Ha a játékgráfban van kör, akkor nem terminál az algoritmus (mivel ez fakeresés, és simán tudunk olyan csúcsokat vizsgálni, amiket már egyszer megvizsgáltunk), de ez nem igazán jelent problémát, ugyanis a játékszabályok gyakran kizárnák a köröket a végtelenségig folytatódó játszmák megelőzése érdekében.



Ugye tudjuk, hogy MAX kezdi a játékot az A csúcsban (a fa gyökerében). Neki az a lényeg, hogy a rendelkezésre álló lépések közül azt a részfát válassza ki, amin keresztül a lehető legnagyobb értékű hasznosságfüggvény érhető el. Ebben a lépésben ez az a_1 élen keresztül lesz elérhető, mert a másik két csúcson 2-t érhetünk el mindössze.

Ekkor következik MIN, aki a B állapotból el kell, hogy döntse, hogy melyik az az állapot, amelyben a hasznosságfüggvény a lehető legkisebb értéket veszi fel. Ahogy az a képen is látható, megvizsgálja a b_1 , b_2 és b_3 útvonalakon elérhető állapotokat, és észreveszi, hogy a számára legjobb érték a b_1 úton keresztül érhető el (mivel $3 < 8 < 12$).

Ez így, hogy a fa fel van címkézve, könnyen meghatározható, viszont feladatként az is előfordulhat (vizsgán báris volt), hogy címezzük fel az adott játékfát a MINIMAX algoritmus által szerezhető hasznosságfüggvény értékekkel. Ilyen esetben a csúcsokon nyerhető összegeket fogjuk tudni (érthető, hogy ezek nélkül ez nem meghatározható), és a fában felfelé kell tudnunk meghatározni az értékeket.

Először meg kell nézni a fa mélységét: páros mélység esetén először a levelek közül a legkisebbet vesszük, ellenkező esetben a legnagyobbat. Fogjuk ezt az értéket, és betesszük a szülőbe. Ezután ezt megismételjük annyi különbséggel, hogy a következő lépében a másik játékos számára optimális érték kerül be a szülőbe (addig, ameddig nem jutunk el a gyökérig). Érthető, hogy kb ez is úgy működik, mint maga az algoritmus, csak fordítva.

Habár ez elég könnyen kiszámolható, sajnos érezhető, hogy nem skálázódik valami jól, ugyanis a sakkból már 10^{154} csúcs van, ebből 10^{40} különböző (a világegyetem atomjait körülbelül 10^{80} köré számolják). Itt felmerül a kérdés, hogy hogyan lehet hatékonyan MINIMAX-ot számolni? Ha nem hatékonyan, bár közelítőleg? Kell egyáltalán MINIMAX-ot számolni?

Alfa-béta eljárás: a játékot szintén gráffal reprezentáljuk mint a MINIMAX esetében. A példában látható volt, hogy rengeteg olyan érték ki lett számolva, ami az eljárás során nem volt használva, erre nyújt megoldást az **alfa-béta vágás**.

Ha tudjuk, hogy például MAX egy adott csúcs rekurzív kiértékelése során talált egy olyan stratégiát, amellyel ki tud kényszeríteni egy 10 értékű hasznosságot, akkor a további kiértékelés során érezhetően felesleges azokat az állapotokat, ahol MIN ki tud kényszeríteni ≤ 10 értékeket, ugyanis MAX sosem fog tudni ide jutni (ugyanis ennél van már jobb startégiája, és MIN tökéletesen racionális).

Ha MAXnak már felfedeztünk egy olyan stratégiát, amely nem generál megfelelő hasznosságot, **alfa vágás** segítségével levágjuk az onnan elérhető részfát, és nem számoljuk ki. Nyilván MIN esetében ezt **béta vágásnak** nevezzük. Érezhető a hasonlóság a MINIMAX-hoz képest, csak itt próbálhatunk alfát és bétát, és ha kell, vágunk.

Érezhető, hogy a futásidőt sokban befolyásolja a szomszédok bejárási sorrendje, ugyanis ha minden csúcs esetében elsőre megtaláljuk a legjobb lépést (optimális eset), akár megfelezhetjük az időigényt. Ennek érdekében használhatunk rendezési heurisztikát, aminek segítségével elég közel tudunk kerülni az optimális esethez.

Tekintettel arra, hogy a játékgráfban is sok ismétlődés lehet (más lépések hatására is elő tudjuk idézni, hogy ugyanabba az állapotba kerüljünk), ezen tovább tudunk optimalizálni, ha a gráfkereséshez hasonlóan tároljuk a már látott állapotokat (mint a zárt halmazban). Itt a hagyományos elnevezése a zárt halmaznak a **transzponációs tábla**.

Korlátozás kielégítési feladat

A feladat az állapottérrel adott **keresési problémák** és az **optimalizálási problémák** jellemzőit ötvözi (igazából csak gondolunk bele: keresünk egy olyan állapotot, amely kielégíti a korlátozásokat - optimális), ahol az állapotok és célállapotok speciális alakúak.

Lehetséges állapotok halmaza: $D = D_1 \times \dots \times D_n$, ahol D_i az i. változó lehetséges értékei, azaz a feladat állapotai az n db változó lehetséges értékkombinációi.

Célállapotok: a megengedett állapotok, amelyek definíciója a következő: adottak C_1, \dots, C_m korlátozások, $C_i \subseteq D$. A megengedett vagy konzisztens állapotok halmaza a $C_1 \cap \dots \cap C_m$ olyan állapotokat tartalmaz, amelyek minden korlátozást kielégítenek.

Ugyanakkor, akár az optimalizálási problémáknál, az út a megoldásig lényegtelen, és gyakran célfüggvény is értelmezve van az állapotok felett, ugyanis itt **egy optimális célállapot megtalálása a cél**.

Vegyük például a Gráfsínezés problémát (bemenet egy gráf, kiszínezhetőek-e a csúcsai valamennyi színnel úgy, hogy két szomszédos csúcs ne legyen egyszínű). Érhető, hogy ebben az esetben D_i egy olyan értékadás lesz a gráf összes csúcsára, amely valamilyen színt felvesz. Ehhez minden csúcsra felírunk egy korlátozást, amely a szomszédos csúcsokra páronként megszorítja, hogy azok nem lehetnek egyszerre egyszínűek. Ezekre a változópárokra gráfokat tudunk felírni, amiket **kényszergráfnak** hívunk. Könnyen belátható, hogy ilyen gráffal minden kettőnél több változót érintő korlátozást fel tudunk írni, ha szükség esetén bevezethetünk segédváltozókat.

Inkrementális kereső algoritmusok: ugye említettük, hogy ezek a feladatok a keresési problémák és az optimalizálási problémák jellemzőit ötvözik, első nekifutásra álljunk hozzájuk úgy, mint a keresési problémákhoz. Ekkor az állapottér a következő:

- minden változóhoz felveszünk egy új ismeretlen értéket, amit ?-el jelölünk, a kezdőállapot nyilván így (?,...,?) lesz
- Az állapotátmenet függvény minden állapothoz hozzárendeli azokat az állapotokat, amelyekben egygel kevesebb ? van, és még megengedett
- A költség minden állapotra lehet konstans

Onnan jön az elnevezés, hogy **minden inkrementációban próbáljuk csökkenteni az ismeretlen értékek számát, ezáltal közelítünk a megfelelő állapothoz**, érhető, hogy nem skálázódik jól nagy problémákra. Ehhez a kereséshez használhatunk bármilyen informálatlan algoritmust.

Optimalizáló algoritmusok (lokális keresők): a másik megközelítés nyilván az optimalizálási problémákat definiált megközelítés lesz. Itt mondjuk a célfüggvényt definiáljuk úgy, hogy az a megsértett korlátozások számát adja (nyilván ezt szeretnénk minimalizálni, egészen 0-ra). Ha eredetileg volt célfüggvényünk, akkor értelemszerűen össze kell vele kombinálnunk. Az operátorokat definiálhatjuk sokféleképpen, például valamelyik változó értékének megváltoztatásaként. Ehhez használható az összes korábban látott lokális kereső, genetikus algoritmus, stb.

Ezek nagyon gyorsak és sikeresek, de nem mindig találnak (optimális) megoldást.

Összegzés

Keresési feladat

- súlyozott gráffal reprezentáljuk
- lehetséges állapotok halmaza (ezek lesznek a gráf csúcsai)
- kezdőállapot
- lehetséges cselekvések halmaza, és egy állapotátmenet függvény, amely minden állapothoz hozzárendel egy (cselekvés, állapot) típusú, rendezett párokból álló halmazt (ezek lesznek az élek)
- állapotátmenet költségfüggvénye, amely minden lehetséges állapot-cselekvés-állapot hármashoz egy $c(x, a, y)$ valós nemnegatív költséget rendel (ezek lesznek a súlyok)
- célállapotok halmaza
- ezt a gráfot állapottérnek nevezzük,
- az állapotok számossága véges vagy megszámlálható
- egy állapotnak legfeljebb véges számú szomszédja van

Informálatlan (vak) keresés

- az állapottér egy súlyozott gráf, ahol a csúcsok az állapotok, az élek a cselekvések, a súlyok pedig a költségek
- véges vagy megszámlálható állapot, legfeljebb véges számú szomszéd
- semmilyen információjuk nincs az állapotokról a probléma definíciójában megadott információk kívül
- annyit tudunk tenni, hogy megnézzük a lehetséges következő állapotokat, vagy a rendelkezésünkre álló állapotokat összehasonlítjuk a célállappal
- FAKERESÉS
 - a kezdőállapotot betesszük a perembe
 - minden iterációban kivesszük a perem első elemét
 - megnézzük, hogy célállapot-e
 - ha nem, kiterjesztjük (a gyerekeit bevesszük az perembe)
 - végtelen ciklusba juthat még akkor is, ha véges számú állapot van (gráfokban kör)
- GRÁFKERESÉS
 - fakeresés, csak tároljuk az egyszer már kiterjesztett csúcsokat (zárt halmaz)
 - azokat a csúcsokat nem vizsgáljuk, amiket már kiterjesztettünk
- SZÉLESSÉGI KERESÉS
 - FIFO (first-in-first-out) perem - verem
 - kis mélység mellett jó, nem skálázódik jól
- MÉLYSÉGI KERESÉS
 - LIFO (last-in-first-out) perem - sor
 - nem optimális
- ITERATÍVAN MÉLYÜLŐ KERESÉS
 - mélységi keresések sorozata
 - korlátozott mélységre, ameddig nem találunk célállapotot
 - legjobb informálatlan kereső
- EGYENLETES KÖLTSÉGŰ KERESÉS
 - perem rendezése költség alapú

Informált keresés

- a tényleges út hosszán kívül bevezetünk egy új függvényt, a heurisztikát
- a heurisztika minden állapotból megbecsüli, hogy mekkora lehet az optimális út költsége az adott állapotból egy célállapotba
- $h(n)$: optimális költség közelítése n állapotból a legközelebbi célállapotba
- $g(n)$: tényleges költség a kezdőállapotból n-be
- egy $h()$ heurisztika elfogadható, ha nem ad nagyobb értéket, mint az optimális érték
- MOHÓ ALGORITMUS:
 - a peremben a rendezést $h()$ alapján végezzük
 - legkisebb értékű csúcsot vesszük ki
 - "legjobb először"
- AZ A* ALGORITMUS:
 - a permet $h() + g()$ által kapott érték alapján rendezzük
 - a legkisebb értékű csúcsot vesszük ki
 - ha $h()$ konziszens és az állapottér véges, akkor A* optimális
- EGYSZERŰSÍTETT, MEMÓRIAKORLÁTOZOTT A*:
 - futtatjuk A*-ot ameddig van memória
 - ha elfogyott, töröljük a legrosszabb levelet a fában (egyenlő esetén a legrégebbit)
 - törölt csúcs szülőjében jegyezzük meg az innen elérhető ismert legjobb költséget
 - problémája az, hogy sok hasonló költségű út esetén ugrálhat az utak között, eldobja és újraépíti őket
 - ha az optimális út memóriaigénye nagyobb, mint amennyit a korlátozással tárolni tudunk, akkor nincs mit tenni, nem is találjuk meg az optimális megoldást

Kétszemélyes, lépésváltásos, zéró összegű, determinisztikus játékok

- kétszemélyes: MIN és MAX ágens, egyik minimalizálni, a másik maximalizálni szeretné a hasznosságfüggvényt
- lépésváltásos, tehát minden két játékos egy lépést hajt végre, majd a másik jön
- zéró összegű, tehát pontosan annyit nyer az egyik játékos, mint amennyit a másik veszít
- determinisztikus, tehát egy meghatározott stratégia alapján döntenek
- állapotteret gráffal reprezentáljuk
- MINIMAX ALGORITMUS
 - tökéletes racionálitás hipotézise
 - minden két játékos ismeri a teljes játékgráfot
 - tetszőlegesen komplex számításokat tud elvégezni
 - nem hibázik
 - ha a tökéletes racionálitás feltehető, minden két játékos számára a lehető legjobb stratégiát a MINIMAX algoritmus alapján lehet megvalósítani.
 - minden csúcsra rekurzívan a levelektől kezdve kiszámoljuk
 - a hasznosságot, ha végállapot
 - maximumot, ha a következő lépésben MAX jön
 - minimumot, ha a következő lépésben MIN jön
- nem skálázódik jól, túl sokat számol

- ALFA-BÉTA ELJÁRÁS

- olyan mint a MINIMAX
- ha MAX számításakor találtunk egy értéket, ami MIN determinisztikus lépései miatt nem lesz elérhető, akkor azt a részfát levágjuk
- ha MAX vág, az alfa vágás, ha MIN vág, az béta vágás
- a futásidőt befolyásolja a szomszédok bejárási sorrendje, használhatunk rendezési heurisztikát
- a gráfkereséshez hasonlóan tárolhatjuk a már ismert állapotokat transzponációs táblában

Korlátozás kielégítési feladat

- a keresési problémák és az optimalizálási problémák jellemzőit ötvözi
- keresünk egy olyan állapotot, amely kielégíti a korlátozásokat - optimális
- állapotok halmaza: a feladat állapotai az n db változó lehetséges értékadási kombinációi
- célállapotok: olyan állapotokat tartalmaz, amelyek minden korlátozást kielégítenek.
- az út a megoldásig lényegtelen, a célfüggvény is értelmezve van az állapotok felett
- egy optimális célállapot megtalálása a cél.
- Gráfszínezés probléma (bemenet egy gráf, kiszínezhetőek-e a csúcsai valamennyi színnel úgy, hogy két szomszédos csúcs ne legyen egyszínű).
- az állapottér egy csúcsa egy olyan értékadást fog reprezentálni, ami a bemeneti gráf összes csúcsához valamilyen színt rendel
- minden csúcsra felírunk egy korlátozást, amely a szomszédos csúcsokra páronként megszorítja, hogy azok nem lehetnek egyszínűek
- a változópárokra gráfokat tudunk felírni (kényszergráfak) hívunk
- ilyen gráffal minden kettőnél több változót érintő korlátozást fel tudunk írni, ha bevezethetünk segédváltozókat
- INKREMENTÁLIS KERESŐ ALGORITMUSOK
 - állapottérbeli keresési problémaként formalizáljuk
 - minden változóhoz felveszünk egy új ismeretlen értéket, amit ?-el jelölünk
 - a kezdőállapot (?,...,?) lesz
 - az állapotátmenet függvény minden állapothoz hozzárendeli azokat az állapotokat, amelyekben egygel kevesebb ? van, és még megengedett
 - a költség minden állapotra konstans
 - minden inkrementációban próbáljuk csökkenteni az ismeretlen értékek számát, ezáltal közelítünk a megfelelő állapothoz
 - nem skálázódik jól nagy problémákra.
- OPTIMALIZÁLÓ ALGORITMUSOK (LOKÁLIS KERESŐK):
 - optimalizálási problémaként formalizáljuk
 - a célfüggvényt definiáljuk úgy, hogy az a megsértett korlátozások számát adja
 - ezt szeretnénk minimalizálni, egészen 0-ra
 - ha eredetileg volt célfüggvényünk, akkor össze kell vele kombinálnunk
 - az operátorokat definiálhatjuk sokféleképpen
 - ehhez használható az összes korábban látott lokális kereső pl
 - gyorsak és sikeresek, de nem mindenkorának (optimális) megoldást.

Mesterséges intelligencia I - 2.)

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

Teljes együttes eloszlás tömör reprezentációja, Bayes hálók

A logika igaz/hamis világgal és logikai következtetésekkel problémák vannak:

- ha nem teljes a tudás (tények vagy szabályok), akkor nem minden tudunk logikai levezetéseket gyártani fontos kérdésekhez, döntésképtelenek leszünk
- ha heurisztikus (=rávezető, tapasztalatra épülő) szabályokat vezetünk be, akkor a tapasztalat inkonzisztens (=ellenmondó) lehet az elméettel, tehát a logikai levezetés megint csak nem működik

Tehát a hiányos, részleges tudás kezelésére a logika nem optimális.

Valószínűség: a tudás tökéletlenségének (ismeretlen tények és szabályok) véletlen hatásaként való kezelése. pl: fogfájás → lyuk 80%-ban igaz

- Egy állítás/esemény valószínűsége változik annak függvényében, hogy mit tapasztalunk. (pl ha húzunk egy kártyalapot, más a valószínűsége annak, hogy pikk ász mielőtt és miután megnéztük)

A feladat az, hogy megfigyelésekkel (tényekből) és valószínűségi tudásból számoljuk ki az ágens számára fontos események valószínűségét.

Véletlen változók: egy véletlen változónak van neve és lehetséges értékei (értékkészlet, domain). Legyen A egy véletlen változó, D_A domainnel. A D_A típusának függvényében képezhetünk elemi kijelentéseket, amelyek az A értékének egy korlátozását fejezik ki (pl. A = d, ahol $d \in D_A$). Domain alapján a következő típusok vannak:

- **logikai:** ekkor a domain {igaz, hamis}, pl Fogfájás=igaz egy elemi kijelentés (röviden fogfájás kisbetűvel)
- **diszkrét:** megszámlálható domain. Pl. Idő, ahol a domain pl. {nap, eső, felhő, hó}. röviden az Idő=nap helyett lehet az hogy nap
- **folytonos:** pl. X véletlen változó $D \subseteq R$ (valós számok). Pl. $X < 3.2$ egy elemi kijelentés.

Komplex kijelentéseket képezhetünk logikai operátorokkal (\wedge , \vee , \neg , ...).

Elemi esemény (a lehetséges világok analógja): 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 a $D_1 \times \dots \times D_n$ halmaz (azaz az összes domain direkt szorzatával megkaphatjuk az elemi eseményeket).

Egy lehetséges világban (azaz elemi eseményben) az A_i véletlen változó a hozzá tartozó D_i -ből pontosan egy értéket vesz fel. Például ha két logikai véletlen változónk van (Luk és Fogfájás), akkor négy elemi esemény van:

$$\text{luk} \wedge \text{fogfájás}, \text{luk} \wedge \neg \text{fogfájás}, \neg \text{luk} \wedge \text{fogfájás}, \neg \text{luk} \wedge \neg \text{fogfájás}$$

A valószínűség egy függvény, amely egy kijelentéshez egy valós számot rendel.

Feltételes valószínűség: $P(a|b)$ kijelentés azt jelenti, hogy "mennyi a valószínűsége az a eseménynek, ha b-t tudjuk", például $P(\text{luk}|\text{fogfájás}) = 0.8$, mennyi a valószínűsége hogy lyukas a fog feltéve hogy fáj. Definíció szerint: $P(a|b) = P(a \wedge b) / P(b)$, ($P(b) > 0$ nem osztunk nullával).

A szorzatszabály: $P(a \wedge b) = P(a|b)P(b) = P(b|a)P(a)$.

Teljes együttes eloszlás: megadja az összes elemi esemény valószínűségét, amelyből kiszámítható bármely kijelentés valószínűsége. Például legyenek a véletlen változók Luk, Fogfájás, Beakad, mind logikai $\{0, 1\}$ típusú. Ekkor a teljes együttes eloszlást egy táblázat tartalmazza.

Luk	Fogfájás	Beakad	$P()$
nem	nem	nem	0.576
nem	nem	igen	0.144
nem	igen	nem	0.064
nem	igen	igen	0.016
igen	nem	nem	0.008
igen	nem	igen	0.072
igen	igen	nem	0.012
igen	igen	igen	0.108

Ebből pl. $P(\text{luk} \vee \text{fogfájás}) = 0.108 + 0.012 + 0.072 + 0.008 + 0.016 + 0.064$.

Ha ismert a teljes együttes eloszlás táblázat nagy lehet (pl. ha minden változó logikai, és n db változó van, akkor ez 2^n méretű, tehát ez nem skálázódik) viszont reprezentálhatjuk tömbben is a **függetlenség** és a **feltételes függetlenség** segítségével, úgy már elviselhetőbb.

Függetlenség: Az a és b kijelentések függetlenek akkor és csak akkor, ha $P(a \wedge b) = P(a)P(b)$. Az A és B véletlen változók (vagy változóhalmazok) függetlenek akkor és csak akkor, ha $P(A,B) = P(A)P(B)$, vagy ekvivalensen $P(A|B) = P(A)$ és $P(A|B) = P(B)$. Azaz két változó akkor függetlenek, ha az egyik nem tartalmaz információt a másikról.

Feltételes függetlenség: Az a és b kijelentések feltételesen függetlenek c feltevésével akkor és csak akkor, ha $P(a \wedge b|c) = P(a|c)P(b|c)$. Ekkor a és b nem feltétlenül független. Tipikusan a és b közös oka c. Pl. a fogfájás és a beakadás közös oka a luk, a fogfájás és a beakadás nem független, de ha feltesszük, hogy van luk, akkor már igen.

Az A és B véletlen változók feltételesen függetlenek C feltevésével akkor és csak akkor, ha $P(A, B|C) = P(A|C)P(B|C)$, vagy ekvivalensen $P(A, B|C) = P(A|C)$ és $P(A, B|C) = P(B|C)$.

Az eloszlás reprezentációjának tömörítése a feltételes függetlenség segítségével: Tegyük fel, hogy a következő egyenlőség igaz:

$$P(A, B, C) = P(A, B|C)P(C) = P(A|C)P(B|C)P(C),$$

Ahol A, B és C jelöli változók diszjunkt (nincs közös elemük) halmazát. Itt az első egyenlőség a szorzat szabály, a második a egyenlőség fogalmazza meg a feltételes függetlenség feltevését. Ha A feltevése mellett teljesül, hogy B_1, \dots, B_n kölcsönösen függetlenek.

Bayes hálók: valószínűségi modellek leírására használják. A valószínűségi modellek egyik legkézenfekvőbb megadási módja a teljes együttes eloszlás megadása lenne, azonban n db logikai változó esetén 2^n db elemi esemény valószínűségét kellene megadni és tárolni.

Hogy egyszerűbb dolgunk legyen, érdemes a valószínűségi változók közötti feltételes függetlenségeket kihasználni, mert segítségükkel tömöríthetjük a teljes együttes eloszlás reprezentációját. A Bayes-hálók a teljes együttes eloszlás feltételes függetlenségeit ragadják meg egy meghatározott módon tömör és intuitív reprezentációt (vagy közelítést) tesznek lehetővé.

A Bayes-háló **egy olyan irányított, körmentes gráf**, ahol:

- a háló csomópontjai valószínűségi változók
- az irányított élek az ok okozati függőségek: ha X -ből vezet Y -ba él, akkor X függ Y -től
- minden X_i csomóponthoz tartozik egy $P(X_i | \text{Szülök}(X_i))$ feltételes valószínűség eloszlás, ami számszerűen megadja a szülők hatását a csomóponti változóra

Bayes-hálók esetén ha X_1, \dots, X_n a véletlen változók egy felsorolása, akkor a láncszabály:

$$\begin{aligned} P(X_1, \dots, X_n) &= P(X_1|X_2, \dots, X_n)P(X_2, \dots, X_n) = \\ &= P(X_1|X_2, \dots, X_n)P(X_2|X_3, \dots, X_n)P(X_3|X_4, \dots, X_n) \cdots P(X_n) = \\ &= \prod_{i=1}^n P(X_i|X_{i+1}, \dots, X_n) \end{aligned}$$

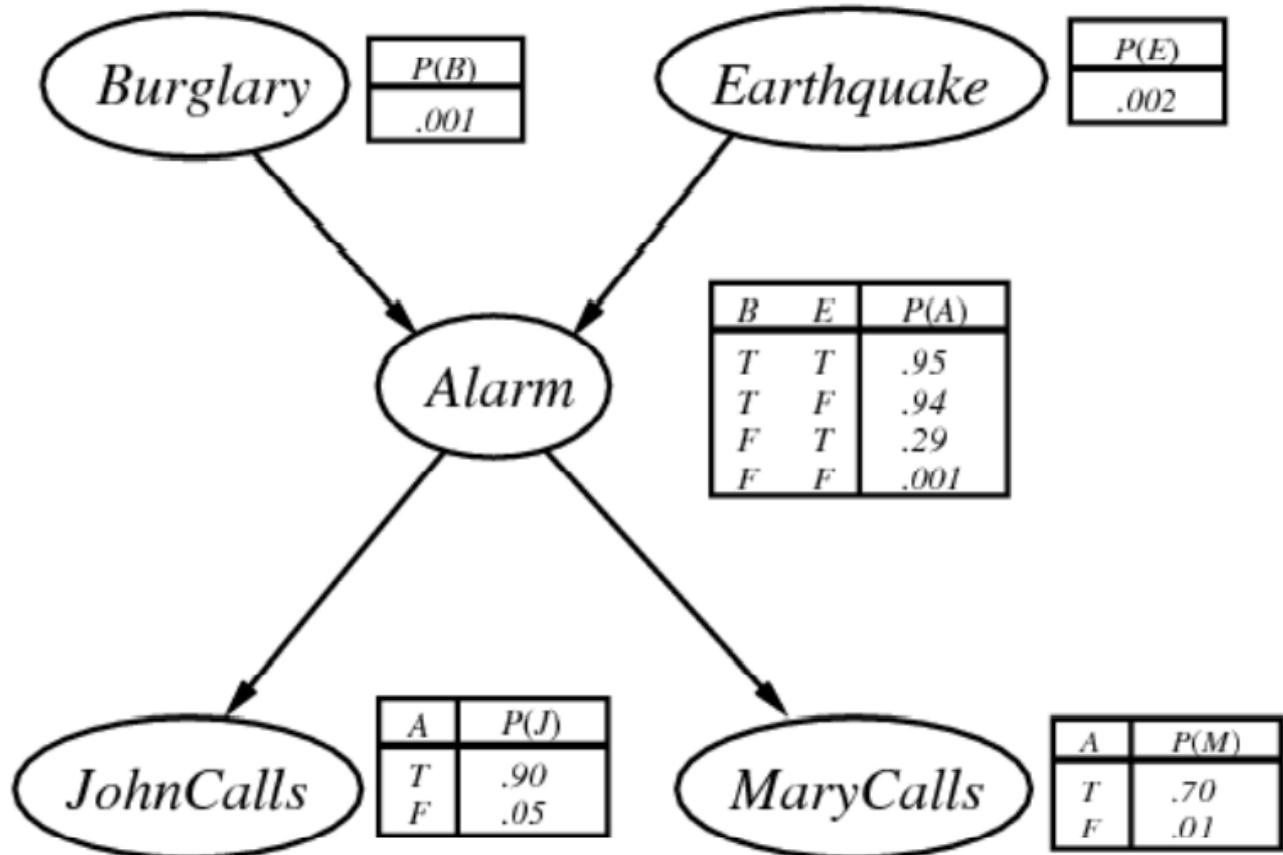
Naiv Bayes algoritmus: statisztikai következtetési módszer, amely adatbázisban található példák alapján ismeretlen példákat osztályoz. Pl. a feladat lehet üzenetek osztályozása spam és nem spam kategóriákba. Ekkor az adatbázis példa emaileket tartalmaz, amelyekhez ismert az osztályozás eredménye (tudjuk róluk, hogy spam vagy nem spam).

Legyen A és B_1, \dots, B_n a nyelvünk változói. Pl. A lehet igaz, ha egy email spam, hamis, ha nem, illetve B_i logikai változó pedig az i. szó előfordulását jelezheti (igaz, ha az email tartalmazza az i. szót, hamis, ha nem). A feladat tehát az, hogy adott konkrét b_1, \dots, b_n email esetén meghatározzuk, hogy A mely értékére 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 α nem függ A -től, az α értéke nem érdekes. A második közelítő egyenlőségjel fogalmazza meg a **naiv Bayes feltevést**. Ez csak közelítés, mivel nem tudjuk biztosan, hogy az egyenlőség teljesül-e. Általában valószínűbb, hogy nem teljesül, de abban bízunk, hogy elfogadható közelítést ad. A pontatlanságért cserébe $P(A)$ és $P(b_i|A)$ könnyen közelíthető az adatbázisban található példák segítségével, így a képlet a gyakorlatban kiszámolható A minden lehetséges értékére, és a nagysági sorrend meghatározható.

Példa Bayes-hálóra:



Ebben egy lehetséges valószínűség pl:

$$\begin{aligned}
 & P(B \wedge \neg E \wedge A \wedge J \wedge \neg M) = \\
 & = P(B)P(\neg E)P(A|B, \neg E)P(J|A)P(\neg M|A) = \\
 & = 0.001 \cdot (1 - 0.002) \cdot 0.94 \cdot 0.9 \cdot (1 - 0.7)
 \end{aligned}$$

Gépi tanulás

A tanulás tapasztalati (megfigyelt) tények felhasználása arra, hogy egy racionális ágens teljesítményét növeljük. Több fajtája van (X az állapotok, Y pedig a cselekvések halmaza):

Felügyelt (induktív) 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 ($x_i \in X$). Például X : emailek halmaza, $Y = \{\text{spam}, \text{nem spam}\}$, a példák pedig kézzel osztályozott emailek. Ez lehet egy spam szűrő tanítása.

Felügyelet nélküli tanulás: a példák csak x_1, \dots, x_n alakban adottak ($x_i \in X$), és nem függvényt kell keresni, hanem mintázatokat (pl. eloszlást).

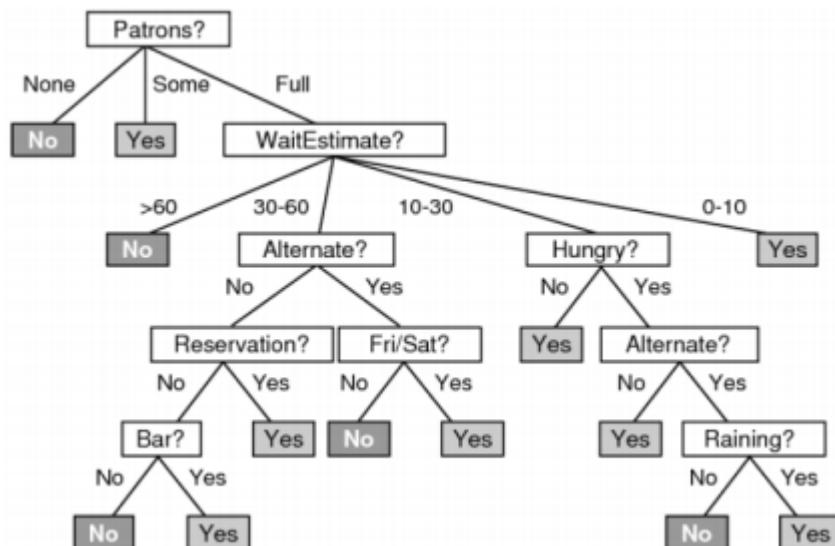
Megerősítéses tanulás: itt is létezik az $X \rightarrow Y$ függvény, de tanuló példák nem adottak direkt módon. A feladat itt az, hogy egy állapottérben az ágens megtanulja az optimális stratégiát úgy, hogy a jövőben érintett állapotokban összegyűjtött jutalmak maximalizálását érje el. Számos finomabb megkülönböztetés van, pl. felügyelt tanulás esetén a példák folyamatosan gyűlnek tanulás közben, vagy akár az ágens kérdéseket tehet fel (aktív tanulás) stb.

Reprezentáció: Az X és Y halmazok tetszőleges objektumokat írhatnak le, és fontos, hogy ezeket hogyan reprezentáljuk. Pl. szövegeknek sokfajta reprezentációja lehet, a tartalmazott szavak listájától a folytonos szemantikus beágyazásokig. Az Y halmaz jellemzően diszkrét osztálycímeket tartalmaz (pl. spam vagy nem spam), illetve lehet folytonos halmaz is.

Döntési fák

A felügyelt tanulás egy konkrét példája. Feltesszük, hogy $x \in X$ diszkrét változók értékeinek vektora, és $f(x) \in Y$ egy diszkrét változó egy értéke (pl. $Y = \{\text{igen, nem}\}$). 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).

Példa: az a feladat, hogy döntsük el, hogy érdemes-e asztalra várni egy étteremben. Ez egy 2 osztályos (igen, nem) osztályozási feladat, vagyis **döntési feladat**. Az X halmaz a változók (Vendék, Éhes stb.) értékeinek a vektora, minden változó diszkrét.



A döntési fa előnye, hogy a döntései megmagyarázhatóak, mert emberileg értelmezhető lépésekben jutunk el a döntésig. Ez a mesterséges neurális hálókra sajnos már nem igaz.

Modellillesztés: adottak a pozitív és negatív példák, ahol minden változó értékét megadjuk, a példa címkéjével együtt, pl:

- Vendégek = tele
- Várakozás = 10-30
- Éhes = igen
- VanMásHely = nem
- Esik = igen
- Foglalás = nincs
- Péntek/Szombat = igen
- VanBár = igen
- -> IGEN (pozitív példa)

Ilyen példákból tipikusan adott minimum több száz. A példákat „bemagolni” könnyű: pl. tekintsük a példákat az igazságtábla ismert sorainak. A változók sorba rendezésével építhetünk fát, ahol az ismeretlen igazságtábla sorokhoz véletlen igazságértéket írunk.

Ez egy olyan fa lesz, amely konzisztens a példákkal. Révén, hogy ez magolás, nem fog általánosítani, ezért a gyökérbe vegyük azt a változót, amelyik a legtöbb információt hordozza (legjobban szeparálja a pozitív és negatív példákat).

Ezután a gyökér által minden rögzített változó értékére és a hozzá tartozó példa-részhalmastra rekurzívan tegyük meg ugyanezt! Adott részhalmazon a még nem rögzített változók közül gyökeret választunk a részhalmazt leíró részfához, majd megismételjük a lépést.

Vannak sajnos olyan speciális esetek, amik megállítják a rekurziót:

- Ha csak pozitív vagy negatív példa van, akkor levélhez értünk, megcímkezzük megfelelő módon
- Ha üres halmafról van szó: alapértelmezett érték, pl. a szülőben a többségi döntés
- Ha pozitív és negatív példa is van, de nincs több változó: ekkor a többségi szavazattal címkezhetjük a levelet.

Zajszűrés: a magolás problémájához hasonló a túlillesztés problémája, ahol „túlságosan pontosan” illesztjük az adatokra a modellt, ami akkor fordul elő, ha túl általános a modellünk, vagy ha túl kevés a példa. Ilyenkor már általában a zajt reprezentáljuk, nem a lényeget, és az általánosítási képesség ezért csökken. Például, ha dobókockával dobjuk a példákat és feljegyzünk irreleváns attribútumokat, pl. a kocka színét, a dobás idejét stb. Az irreleváns attribútumok információnyeresége elméletben nulla. A gyakorlatban viszont nem, tehát felépül egy döntési fa, ami értelmetlen – túlillesztettük. Zajt szűrhetünk úgy, hogy megnézzük, hogy az információnyereség statisztikailag szignifikáns-e.

Mesterséges neuronhálók

K-legközelebbi szomszéd módszere

Összegzés

Operációkutatás I - 1.)

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 alapfeladata

Keressük meg az adott lineáris, R^n (valós számok halmaza) értelmezési tartományú célfüggvény szélsőértékét, (minimumát/maximumát) értelmezési tartományának adott lineáris korlátozókkal (feltételekkel) meghatározott részében. LP alapfeladatának felírása:

$\begin{array}{ll} \text{Max} & c_1x_1 + c_2x_2 + \dots + c_nx_n = z \\ \text{Felt.} & a_{11}x_1 + a_{12}x_2 + \dots + a_{1n}x_n \leq b_1 \\ & a_{21}x_1 + a_{22}x_2 + \dots + a_{2n}x_n \leq b_2 \\ & \vdots \\ & a_{m1}x_1 + a_{m2}x_2 + \dots + a_{mn}x_n \leq b_m \\ & x_1, \dots, x_n \geq 0 \end{array}$	<div style="display: flex; justify-content: space-between;"> <div style="flex: 1;"> <p>célfüggvény</p> <p>feltételek</p> </div> <div style="flex: 1; text-align: right;"> <p>változókra vonatkozó korlátozások, ezt minden meg kell adni, ugyanis nem lehetnek negatívak!</p> </div> </div>
--	---

Az LP geometriája

A lineáris programozás szoros kapcsolatban áll a konvex geometriával, mivel az LP kapcsán használt fogalmaknak (mint pl. bázismegoldás, lineáris feltétel stb.) megfeleltethető egy-egy geometriai objektum:

- **R^n :** n-dimenziós lineáris tér a valós számok felett, elemei az n elemű valós vektorok
- **E^n :** n-dimenziós Euklideszi tér, ami egy olyan lineáris tér, amelyben értelmezett egy belső szorzat és egy távolságfüggvény. Transzponálással, normával mutatható be.
- **Pont:** $x \in E^n$ vektor
- **Lehetséges megoldások:** pontok ebben az n-dimenziós Euklideszi térben
- **Lineáris feltételek:** zárt félterek és síkok
- **Lehetséges megoldások halmaza:** zárt félterek és síkok metszete, egy konvex poliéder
- **Feltételek:** poliéder lapok (egy lineáris cfgv valamelyik csúcsában veszi fel a szélsőértékét)
- **Bázismegoldások:** poliéder csúcsok

Ha a lehetséges megoldások halmaza korlátos, akkor van az Ip-nek optimális megoldása- Ebből következik, hogy az optimális megoldások halmaza konvex. Fontos geometriai fogalmak:

- Poliéder: zárt, véges sok csúcsponttal rendelkező ponthalmaz
- Konvex ponthalmaz: egy olyan ponthalmaz, amelyből két tetszőleges pontját összekötő szakasz összes pontja is a halmaz eleme
- Zárt ponthalmaz: tartalmazza a pontjaiból képezhető tetszőleges konvergens sorozat határértékét is

Generálóelem választási szabályok

Első lépésként felírjuk a megadott korlátozásokat és a célfüggvényt. Ezután fel kell vennünk minden sorba egy **mesterséges változót**. Ezt úgy tudjuk megtenni, hogy minden korlátozáshoz hozzáadunk egy, még nem használt változót.

Ezt követően **szótár alakra** kell hozni a feladatot úgy, hogy az első lépésben a korlátozások sorában kifejezzük a mesterséges változókat, később ezek a ki-, és belépő változók segítségével átkerülhetnek másik helyekre. Ebben az alakban bármikor leolvasható egy **lehetséges bázismegoldás** úgy, hogy leolvassuk a kifejezett változókat és a mellettük lévő konstansokat. Egy bázismegoldás akkor **optimális** ha a célfüggvényben már nincs pozitív együtthatós változó.

Ha nem találtuk meg az optimális megoldást, akkor **generáló elemet** kell választanunk. Ez lényegében egy adott sor (feltétel) adott oszlopában (változó) található érték lesz. A **belépő** változók meghatározásához elég sok esetben csak a célfüggvényt vizsgálni, viszont a kilépő változóhoz gyakran **hányadostesztet** kell végeznünk.

Hányadosteszt során vesszük azokat a korlátokat, ahol a belépőváltozó **negatívan** szerepel. Ezekben az egyenletekben a konstans értéket elosztjuk a belépőváltozó szorzójának abszolút értékével, majd rögzítjük, hogy az adott értékre mekkora értéket kaptunk. Mikor az összes feltételre kiszámoltuk ezt, megnézzük, hogy melyik sorból kaptunk a legkisebb értéket, és megkapjuk a kilépő változót.

Generáló elemet tehát a következő módszerek alapján tudunk választani az alábbi feladatra:

$$\begin{array}{rcl} x_4 = 4 & - & x_1 + 2x_3 \\ x_5 = 8 & + & 3x_2 + x_3 \\ x_6 = 7 & + & 2x_1 - x_2 + 2x_3 \\ \hline Z(x) = 0 & + & 5x_1 + 8x_2 + 2x_3 \end{array}$$

Klasszikus

- **belépőváltozó:** a lehetséges belépőváltozók közül (pozitív előjelű a célfüggvényben) válasszuk ki azt, amely a „legpozitívabb” konstansú, több ilyen esetén válasszuk a legelsőt
- **kilépőváltozó:** hányadosteszt, ütközés esetén válasszuk a legkisebb indexű egyenlet bázisváltozóját (tehát a felsorolásban szereplő elsőt)

Bland (AZ ALGORITMUS GARANTÁLTAN VÉGES SZÁMÚ LÉPÉSBEN VÉGET FOG ÉRNI!)

- **belépőváltozó:** a lehetséges belépőváltozók közül kiválasztjuk a legkisebb indexűt
- **kilépőváltozó:** hányadosteszt, ütközés esetén kiválasszuk a legkisebb indexű egyenlet bázisváltozóját

Legnagyobb növekmény (a változás a célfüggvény értékét legjobban növelte!)

- hányadostesztünk az összes lehetséges belépőváltozóra, és az összes változóra megnézzük, hogy mennyi lenne náluk a növekmény, és vesszük a legnagyobbat. Ezáltal meghatározzuk egyszerre a be-, és kilépő változókat is.

Kétfázisú szimplex módszer

Kétfázisú szimplex módszerről akkor beszélünk, amikor a szótár eredeti alakja nem lehetséges, tehát valamelyik korlát jobb oldalán negatív szám szerepel. Ebben az esetben először át kell alakítanunk a szótárt olyan alakra, amelyen alkalmazni tudjuk a korábban ismertetett algoritmust.

Első lépésként felírjuk a megszokott szótár alakot a mesterséges változókkal. Ezt követően felveszünk minden egyenlethez egy $+x_0$ változót. Ezt követően a $Z(x)$ maximalizálási feladathoz felveszük azt a $W(x)$ minimalizálási segédfeladatot, amit úgy kapunk, hogy a $Z(x)$ -ben található előjeleket megváltoztatjuk.

Ezt követően addig választunk generálóelemet a kiegészített feladatban, ameddig a segédfüggvényre nem tudjuk kihozni a 0 értéket. Ha ezzel megvagyunk, akkor a legnegatívabb korlátból kifejezzük x_0 értékét, behelyettesítjük a többi korlátba az ő értékét, és ezáltal megkapunk egy olyan szótár alakot, amire már tudjuk alkalmazni a szimplex algoritmust.

A standard feladatnak akkor, és csak akkor van lehetséges megoldása, ha a hozzá felírt segédfeladat optimuma 0. Ha ez nem elérhető, vagy nincs megoldása, akkor a standard feladat sem megoldható.

Speciális esetek

Ciklizáció-degeneráció: abban az esetben, ha a egy bázismegoldás változói közül legalább az egyik értéke 0, akkor **degenerált bázismegoldásról** beszélünk. Ezen felül **degenerált iterációs lépések** nevezzük azt, amikor az iterációs lépést követően nem változik a bázismegoldás értéke. Az ilyen anomáliák vezethetnek ciklizációhoz, amikor ugyanis visszakapunk egy olyan szótárat, amelyet a szimplex algoritmus során már egyszer megkaptunk.

Nem korlátos feladat: A lineáris programozás egyik alaptétele az az állítás, hogy ha a feladatnak nincs optimális megoldása, akkor az vagy nem korlátos, vagy **nincs lehetséges megoldása**. Ez utóbbit úgy tudjuk megvizsgálni egy adott feladaton belül, hogy megnézzük a célfüggvényben lévő lehetséges belépőváltozókat. Megnézzük a felette lévő szótárban, hogy a hozzá tartozó nem bázisváltozók milyen előjelűek, és ha azt tapasztaljuk, hogy azok is pozitívak, akkor a feladatunk nem korlátos, mivel így igazából ezek tetszőlegesen nagy értéket vehetnek fel. Csak úgy lehet tovább számolni, ha azok a nem bázisváltozók negatívak.

Ha a lehetséges megoldások halmaza korlátos, akkor az LP-nek van optimális megoldása.

Összegzés

Operációkutatás I - 2.)

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

Primál-duál feladatpár

Azt az állítást, miszerint „egy LP feladatnak ha van optimális megoldása, akkor van optimális bázismegoldása is, a **dualitás tételeivel** tudjuk belátni. A dualitás rugalmas hozzállást tesz lehetővé a LP feladatok megoldására.

A szimplex algoritmus iterációsáma közelítőleg a feltételek számával arányos, így ha az eredeti – primál feladatban sok a feltétel, és kevés a változó, akkor érdemes áttérni annak a duálisára. Ugyanakkor ha az első esetben szükség lenne kétfázisú szimplexre, de a duálban nem, akkor is érdemes áttérni.

Menet közben esetleg új feltétel(ek)et kell hozzávennünk a primál feladathoz, akkor a duál feladattal dolgozva ezek csak egy új, nembázisváltozó(k)ént szerepelnek(én)e(k), tehát az algoritmus gyorsaságát nem igazán befolyásolná(k). Egyszerűen hozzávesszük az eredeti feladathoz, és folytatjuk a feladatmegoldást. A duál a standard alakú feladatból egyszerűen megkapható:

- transzponáljuk az A mátrixot
- b és c vektorok szerepét felcseréljük
- maximalizálás helyett minimalizálunk

Egyenletrendszer általános alakja: $Ax + b = c$, ahol A a változők együtthatóit tartalmazó mátrix, b az egyenletekhez tartozó konstansokat tartalmazó vektor, és c az eredmény.

A primál feladat:

$$\begin{array}{rcl} \text{Max} & c^T x & = z \\ & Ax & \leq b \\ & x & \geq 0 \end{array}$$

A duál feladat:

$$\begin{array}{rcl} \text{Min} & b^T y & = w \\ & A^T y & \geq c \\ & y & \geq 0 \end{array}$$

Dualitási komplementáris feltételek

A duál duálisa az eredeti primál. Ez abból ered, hogy ha duplán transzponálunk egy tetszőleges A mátrixot, akkor visszakapjuk magát az A mátrixot. A b és c vektorok esetén meg csak oda-vissza cserélgettük őket, szóval az állítás helyes.

A **gyenge dualitás tétele** szerint, ha az x vektor egy lehetséges bázismegoldása a primál feladatnak, és y vektor pedig egy lehetséges megoldása a duálnak, akkor $c^T x \leq b^T y$. Továbbá ez a tételes kimondja azt is, hogy ha a primál nem korlátos, akkor a duálnak nincs lehetséges megoldása, valamint ha a duál nem korlátos, akkor a primálnak nincs lehetséges megoldása.

Az **erős dualitás tétele** szerint ha az x vektor egy lehetséges bázismegoldása a primál feladatnak, és y vektor pedig egy lehetséges megoldása a duálnak, akkor $c^T x = b^T y$. Ide tartozik még az az

igazság is, hogy ha a primálnak és duálnak is van lehetséges megoldása, akkor az optimumok megegyeznek, valamint a priál természetes változóinak a duál mesterséges változók célfüggvénybeli együtthatói felelnek meg, ellentétes előjellel.

Egészértékű feladatok és jellemzőik (ILP)

A Branch-and-Bound módszer

A hátizsák feladat

Összegzés

Operációs rendszerek - 1.)

Processzusok, szálak/fonalak, processzus létrehozása/befejezése, processzusok állapotai, processzus leírása. Ütemezési stratégiák és algoritmusok kötegelt, interaktív és valós idejű rendszereknél, ütemezési algoritmusok céljai. Kontextus-csere.

Processzusok

A **Processzus** lényegében egy **végrehajtás alatt lévő program**. minden processzushoz **tartozik** egy saját **címtartomány**, azaz a memória egy minimális és egy maximális című helye közötti szelet, amelyen belül a processzus olvashat és írhat. A **címtartomány tartalmazza a végrehajtandó programot**, annak adatait és vermét. minden processzushoz **tartozik** még egy **regiszterkészlet**, beleértve az utasításszámlálót, veremmutatót, egyéb hardverregisztereket és a program futásához szükséges egyéb információkat.

Egy hagyományos operációs rendszerben minden egyes processzus saját címtartománnyal és **egyetlen vezérlési szállal rendelkezik**. Gyakran előfordul olyan helyzet, amikor kívánatos lenne több kvázi párhuzamosan futó vezérlési szál használata egy címtartományon belül úgy, mintha különálló processzusok lennének.

A processzus által felhasznált erőforrások lehetnek a megnyitott fájlok, gyermekprocesszusok, függőben lévő ébresztők, szignálkezelők, elszámolási információk stb. Ezeken kívül a processzusnak van még egy **végrehajtási szála**, amit rendszerint **szálnak** rövidítenek.

A szál **rendelkezik utasításszámlálóval**, amely nyilvántartja, hogy melyik utasítás végrehajtása következik. **Regiszterei** vannak, melyek az aktuális munka változót tárolják. **Rendelkezik veremmel**, amely a végrehajtás eseményeit rögzíti, egy-egy kerettel minden meghívott eljáráshoz, amelyből nem tért még vissza a vezérlés. Bár a szálat egy processzuson belül kell végrehajtani, a szál és annak processzusa különböző fogalmak, így külön kezelhetők.

A processzusok az erőforrások csoportosításai, a szálak pedig azok az egyedek, amelyeket CPU-n való végrehajtásra ütemeznek. A szálak segítségével ugyanazon a processzuson belül több végrehajtást eszközölhetünk, amelyek egymástól nagymértékben függetlenek.

Ha ugyanazon a címtartományon több szál van, akkor egy különálló száltáblázatra van szükség szálankénti bejegyzésekkel. A szálankénti bejegyzések között lesz az **utasításszámláló**, a **regiszterek** és az **állapot**.

Az utasításszámláló azért szükséges, mert a szálakat (hasonlóan a processzusokhoz) felfüggeszthetjük és folytathatjuk. A regiszterek azért kellenek, mert amikor a szálakat felfüggesztjük, a regisztereiket el kell menteni. Végül a szálak, a processzusokhoz hasonlóan futó, futáskész vagy blokkolt állapotban lehetnek.

A **processzusokat leírhatjuk** a hozzájuk tartozó **processzustáblázat** alapján, ami a processzusok nyilvántartására, annak tulajdonságainak leírására szolgáló memóriaterület. Ennek a memóriarésznek **egyes bejegyzéseit processzus vezérlő blokknak** (PCB, Process Control Block) **nevezzük**, ami tartalmazza a processzus azonosítóját, állapotát, a CPU állapotát az esetleges kontextus cseréhez, a jogosultságokat és a birtokolt erőforrások listáját.

Azt már tisztáztuk tehát, hogy minden processzus egy önálló egység saját utasításszámlálóval, veremmel, nyitott fájlokkal, ébresztőkkel és egyéb belső állapotokkal, viszont gyakran szükségük van interakcióra egymás között, ezért állapotuk kommunikációjával tudnak jelezni egymásnak.

Ezek az állapotok alapvetően 3 értéket vehetnek fel:

- 1.) **Futó** (az adott pillanatban éppen használja a CPU-t)
- 2.) **Futáskész** (ideiglenesen leállították egy másik processzor futása érdekében)
- 3.) **Blokkolt** (bizonyos külső esemény bekövetkeztéig nem képes futni)

Az egyszerűbb rendszerekben, megoldható, hogy minden szükséges processzus, a rendszer indulásakor elérhető legyen. Általános célú rendszerek esetében azonban szükség van processzusok létrehozására és megszüntetésére működés közben is. Egy processzus létrehozását 4 fő esemény okozhatja:

1. A rendszer inicializálása.
2. A processzus által meghívott processzust létrehozó rendszerhívás végrehajtása.
3. A felhasználó egy processzus létrehozását kéri.
4. Kötegelt feladat kezdeményezése.

Egy operációs rendszer indulásakor gyakran számos processzus keletkezik, sok közülük az előtérben fut. Ezek azok a processzusok, amelyek a felhasználókkal tartják a kapcsolatot, vagy számukra munkát végeznek.

Mások háttérprocesszusok, amelyek nincsenek egy bizonyos felhasználóhoz hozzárendelve, ehelyett valamilyen sajátos feladataik van (démonoknak is hívjuk őket, pl weboldalak kéréseinek kezelése, nyomtatás).

Egy futó processzus is adhat ki rendszerhívást új processzus létrehozására. Ez olyankor különösen hasznos, ha az elvégzendő munka könnyen megfogalmazható néhány együttműködő, de egyébként egymástól független processzussal (pl. fordításkor make – C fordító – install).

A létrehozás lépései:

1. Memóriaterület foglalása a PCB (Process Control Block), vagyis folyamatvezérlő tábla számára.
2. PCB kitöltése iniciális adatokkal (pl. használható erőforrások, jogosultságok a létrehozó jogosultságaiból stb.).
3. Memória foglalás (programszöveg, adatok, verem számára).
4. A PCB processzusok láncára fűzése (az állapot beállítása futáskészre), ettől kezdve a processzus osztozik a CPU-n.

A processzus létrehozása után elkezd dolgozni, teszi a dolgát, de előbb vagy utóbb befejeződik, rendszerint a következő körülmények között:

1. Szabályos kilépés (önkéntes).
2. Kilépés hiba miatt (önkéntes).
3. Kilépés végzetes hiba miatt (önkéntelen).
4. Egy másik processzus megsemmisíti (önkéntelen).

A legtöbb processzus azért fejeződik be, mert végzett a feladatával, illetve például az is, amikor a fordítóprogram lefordította a neki adott programot. Ekkor végrehajt egy rendszerhívást, amivel közli az operációs rendszer felé, hogy elkészült.

A befejezés második indoka lehet, hogy a processzus végzetes hibát fedezett fel, például, ha egy olyan fájlt akarunk lefordítani, ami nem létezik, a fordítóprogram egyszerűen kilép.

A befejezés harmadik oka a processzus által okozott hiba, esetleg egy hibás programsor miatt. Erre példa többek között egy illegális utasítás végrehajtása, nem létező memóriaterületre való hivatkozás, vagy nullával való osztás. Ilyen esetekben az operációs rendszer egy szignált küld a processzusnak és megszakítja a működését.

A processzusbefejezés negyedik oka az, hogy egy processzus végrehajt egy olyan rendszerhívást, amely azt közli az operációs rendszerrel, hogy semmisítsen meg egy másik processzust. Néhány OS esetében ez a kill hívás. A megsemmisítőnek rendelkeznie kell a szükséges jogosultsággal a kérés végrehajtásához. Vannak operációs rendszerek, ahol egy processzus befejeződése maga után vonja az általa létrehozott processzusok megsemmisítését is.

A befejezés lépései:

- 1.) A gyermek processzusok megszüntetése.
- 2.) A PCB processzusok láncáról való levétele (állapot beállítása terminálisra), ettől kezdve a processzus nem osztozik a CPU-n.
- 3.) A processzus birtokában lévő erőforrások felszabadítása (PCB-beli nyilvántartás szerint, pl. nyitott fájlok lezárása).
- 4.) A lefoglalt memóriaterületek felszabadítása (memóriatérkép alapján).
- 5.) A PCB memóriaterületének felszabadítása.

Ütemezési stratégiák és algoritmusok

Az operációs rendszer azon részét, amely eldönti, hogy melyik processzus fusson először, **ütemezőnek (scheduler)** nevezünk, valamint az erre a célra használt algoritmust pedig **ütemezési algoritmusnak**.

Az időosztásos rendszerekben az ütemezés bonyolult, hiszen gyakran több felhasználó vár a kiszolgálásra, és még kötegelt feladatsorok is lehetnek, vagy háttérfeladatok hajtanak végre különböző műveleteket. **Ütemezésre tehát akkor kerül sor, ha:**

- 1.) egy processzus befejeződik
- 2.) egy processzus blokkolódik egy I/O művelet vagy szemafor miatt
- 3.) új processzus jön létre
- 4.) I/O megszakítás történik
- 5.) időzítőmegszakítás történik

Az ütemezési algoritmusok két csoportba oszthatóak az időzítőmegszakítások kezelésének vonatkozásában. **Nem megszakítható ütemezés** esetében az ütemező a kiválasztott processzust addig engedi futni, amíg az blokkolódik, vagy amíg önszántából le nem mond a processzorról. Ezzel ellentétben **megszakítható ütemezés** esetében a kiválasztott processzus csak **legfeljebb** egy előre meghatározott ideig futhat.

Ütemezési algoritmusok céljai

Minden rendszer

- Páratlanság – minden processzusnak megfelelő hozzáférést biztosítani a CPU-hoz.
- Elvek betartása – a meghatározott elvek szerinti működés biztosítása.
- Egyensúly – a rendszer minden részének egyenletes terhelése.

Kötegelt rendszerek

- Áteresztőképesség – maximalizálni az időegységenként végrehajtott feladatok számát.
- Áthaladási idő – minimalizálni a feladat-végrehajtás kezdeményezése és a befejezés között eltelt időt.
- CPU-kihasználtság – a CPU soha nem állhat tétlenül.

Interaktív rendszerek

- Válaszidő – a kérésekre gyors válasz biztosítása.
- Arányosság – a felhasználók elvárásainak való megfelelés.

Valós idejű rendszerek

- Határidők betartása – adatvesztés elkerülése.
- Előrejelezhetőség – minőségromlás elkerülése multimédia-rendszerekben

Stratégiák kötegelt rendszerekben

A sorrendi ütemezés az egyik legegyszerűbb algoritmus, ami olyan sorrendben osztja ki a CPU-t a processzusoknak, amilyen sorrendben azok kérik. A futásra kész processzusok egyetlen várakozó soron állnak készenlétben. Amikor az első feladat belép a rendszerbe azonnal indulhat, és addig futhat, ameddig akar. Ha további feladatok érkeznek, azok a sor végére kerülnek. Amikor egy futó processzus blokkolódik, a sor elején álló processzus futhat helyette. Ha az előző blokkolt processzus újra futásra kész, akkor beáll a sor végére. Nagy erőssége az algoritmusnak, hogy könnyű megérteni és leprogramozni. Hátránya az, hogy ha nem szakít meg egyes processzusokat, akkor azok nagyon hosszú ideig is futahtnak.

A legrövidebb feladatot először algoritmus feltételezi, hogy a futási idők előre ismertek. Amikor több egyformán fontos feladat van futásra készen, akkor az ütemezőnek a legrövidebb feladatot először elvet kell követnie. Ez az algoritmus optimális abban az esetben, ha a feladatok egyszerre rendelkezésre állnak, és ismerjük a futásidőt. Előfordulhat ugyanis, hogy nem egyszerre érkeznek, ilyenkor általában egyből elkezdjük a futtatásukat és ha először egy sokáig tartó feladat jött, megtörténhet, hogy jobban jártunk volna, ha várunk egy kicsit és a kevesebb futásidőjű feladatot indítjuk el elsőnek.

A legrövidebb maradék algoritmus egy megszakítható változata az előző algoritmusnak. Ennél az ütemező minden időt a processzust választja, amelynek a legkevesebb a befejeződésig még megmaradt ideje (ismerni kell a futásidőt). Amikor új feladat érkezik, a teljes idő összehasonlításra kerül az éppen futó processzus még hátralévő idejével. Ha az új feladat kevesebb időt igényel, mint az aktuális processzus, akkor az aktuális lecseréli az újra. Ezzel a megoldással az új, rövid feladatok jó kiszolgálásban részesülnek.

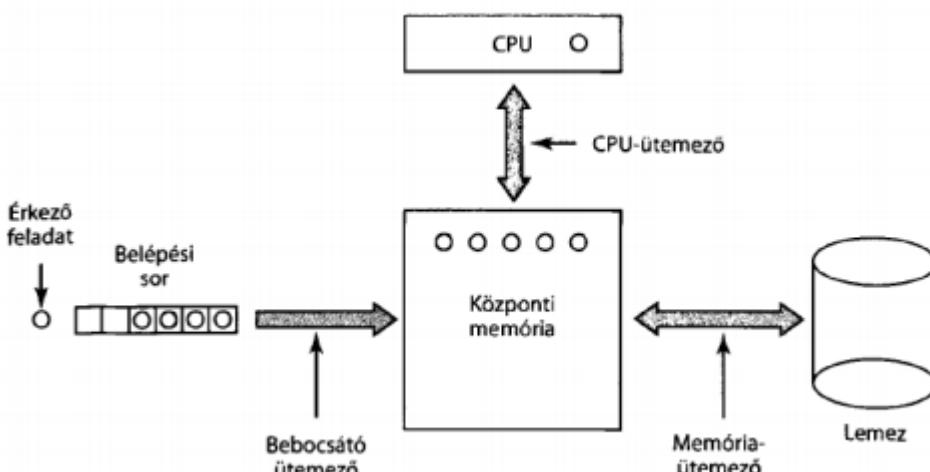
Háromszintű ütemezés során az újonnan érkező feladatok először egy lemezen tárolt belépései várakozó sorba kerülnek. A bebocsátó ütemező dönti el, hogy mely feladatok léphetnek be a rendszerbe. A bebocsátást vezérlő tipikus algoritmus egy megfelelő számításigényes és I/O

igényes keverék előállítását kísérelheti meg. Másik lehetőség, hogy a rövid feladatokat hamar beengedi, a hosszabbaknak várniuk kell. Ha egy feladat már belépett a rendszerbe, akkor létre lehet hozni számára egy processzust, és elkezdhet vételkedni a CPU-ért. De az is megtörténhet, hogy olyan sok processzus van, hogy nem férnek a memóriába. Ebben az esetben néhány processzust ki kell helyezni a lemezre.

Az ütemezés második szintje azt dönti el, hogy melyik processzus maradjon a memóriában, és melyik kerüljön ki a lemezre. Az a komponens, amely ezt a döntést meghozza, **memóriaütemezőnek** nevezik. A döntéseket bizonyos időnként felül kell vizsgálni, hogy a lemezen tárolt feladatoknak is legyen esélyük a bekerülésre. A memória ütemező rendszeres időközönként átnézi a lemezen tárolt processzusokat az alábbi szempontok alapján:

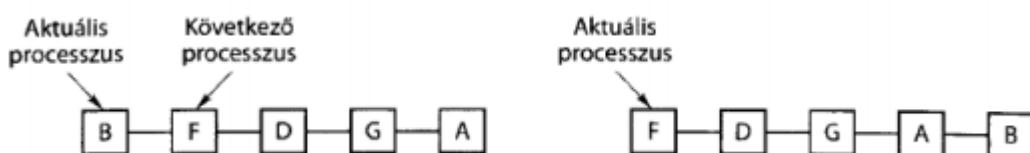
- Mennyi idő telt el a processzus lemezre vitele óta?
- Mennyi CPU időt használt fel a processzus nemrégiben?
- Milyen nagy a processzus?
- Mennyire fontos a processzus?

Az ütemezés harmadik szintje választja ki valójában, hogy a futásra kész processzusok közül melyik fussen következőnek. Ezt gyakran CPU-ütemezőnek nevezik, és az emberek általában erre gondolnak, mikor az ütemezőről beszélnek.

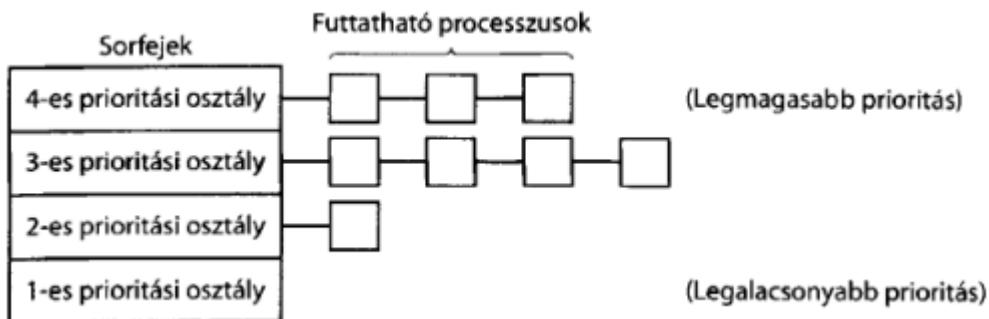


Stratégiák interaktív rendszerekben

A legegyszerűbb és legszélesebb körben használt algoritmus a **Round robin ütemezés**. minden processzusnak ki van osztva egy időintervallum (időszelet), ami alatt engedélyezett a futása. Az ütemezőnek mindenkorra egy listát kell karbantartania a futtatandó processzusokról. Amikor a processzus felhasználja az időszeletét, akkor a lista végére kerül.. Az időszelet túl kicsire állítása túl sok processzus átkapcsolást okoz, és csökken a CPU hatékonysága, viszont ha túl nagy esetén rövid interaktív kérésekre gyenge válaszidőt eredményezhet. Az időszelet 20-50 ezred másodperc körül értéke gyakran ésszerű kompromisszum.



Prioritásos ütemezés a round robin nem figyel a processzusok prioritására, viszont például többfelhasználós gépeken ez eltérő lehet. Az alapötlet az, hogy minden processzushoz rendeljünk egy prioritást, és a legmagasabb prioritású futásra kész processzusnak engedjük meg, hogy fussen. Annak megelőzésére, hogy a magas prioritású processzusok végtelen ideig fussanak, az ütemező minden óraütemben csökkentheti az éppen futó processzus prioritását. Egy másik megoldás lehet, hogy minden processzushoz hozzárendelünk egy maximális időszeletet, ami alatt futhat. A prioritásokat a processzusokhoz statikusan vagy dinamikusan lehet hozzárendelni. Sokszor szoktak prioritási osztályokat felállítani és egy osztályon belül round robin ütemezést használni.



Többszörös sorok megpróbál a CPU igényes processzusoknak inkább nagyobb időszeletet adni, mint gyakran kicsit, mivel ekkor kevesebb sorokba kell a lemezre írni. Másrészt viszont, ha minden processzusnak nagy időszeletet adunk, az gyenge válaszidőt jelent. A megoldás itt is prioritási osztályok felállítása. A legmagasabb osztályban lévő processzusok 1 időszeletig futnak, a következő legmagasabb osztályban lévők 2 időszeletig, a következőben lévők 4 időszeletig, és így tovább. Valahányszor egy processzus elhasználja az összes, számára biztosított időt, egy osztálytalálkozásban. Ez az elv bevezetése azokat a processzusokat védi meg az örökös büntetéstől, amelyek induláskor hosszú időt igényeltek, de később interaktívvá váltak.

Legrövidebb processzus következzen algoritmus becslésekét végez a múltbeli viselkedés alapján, és a processzust a legkisebb becsült futásidejű alapján futtatjuk. A sorozat következő elemét becsülhetjük úgy, hogy vesszük az éppen mért értékek és az előző becslések súlyozott átlagát, ezt a technikát néha öregedésnek nevezzük. Ez sok olyan esetben alkalmazható, amikor a becslést az előző értékekre alapozva kell elvégezni.

A garantált ütemezés ígéreteket tesz a felhasználónak a teljesítménnyel kapcsolatban. Egy reális ígéret, amit könnyű betartani, a következő: ha n felhasználó van bejelentkezve, akkor a CPU teljesítményének körülbelül $1/n$ -ed részét fogod megkapni. Hogy betartsuk az ígéretet, a rendszernek nyomon kell követnie, hogy egy processzus mennyi CPU-időt kapott létrehozása óta és hogy eddig mennyi használt el. Ezután minden felhasználóhoz kiszámítja a neki járó mennyiséget, majd minden processzushoz kiszámolja az aktuálisan felhasznált és a neki járó CPU-idő arányát, és a legkisebb arányszámmal rendelkező processzust fogja futtatni.

Sorsjáték-ütemezés is ígéretet tesz a felhasználónak, de a megvalósítása sokkal könnyebb. Az alapötlet az, hogy minden processzusnak sorsjegyet adunk a különböző erőforrásokhoz. Ha ütemezési döntést kell hozni, egy sorsjegyet véletlenszerűen kiválasztunk, és az a processzus kapja meg az erőforrást, amelynél a sorsjegy van. A fontosabb processzusok többlet sorsjegyeket kaphatnak, hogy növeljék a nyerési esélyeiket. Ezzel az ütemezéssel nagyon jó a válaszidő.

Az arányos ütemezés figyelembe veszi, hogy ki a processzus tulajdonosa. Ebben a modellben minden felhasználó kap valamekkora hányadot a CPU-időből, és az ütemező úgy választja ki a folyamatokat, hogy ezt kikényszerítse, tehát ha két felhasználónak a CPU 50-50%-a lett előirányozva, akkor ennyit fognak kapni attól függetlenül, hogy hánny processzust futtatnak. Pl. az előző esetben, ha az 1-es felhasználónak 4 processzusa van: A, B, C és D, a 2-es felhasználónak csak 1: E, akkor round robin ütemezéssel egy lehetséges ütemezési sorozat, amelyik figyelembe veszi a keretfeltételeket:

A E B E C E D E A E B E C E D E ...

Másrészről, ha az 1-es felhasználónak kétszer annyi CPU-idő jár, mint a 2-esnek, akkor ezt kaphatjuk:

A B E C D E A B E C D E ...

Stratégiák valós idejű rendszerekben

Egy valós idejű rendszerben az idő alapvető szerepet játszik. Jellemzően egy vagy több külső fizikai eszköz ingert küld a számítógép felé, amire annak megfelelően reagálnia kell egy adott időn belül. Olyan esetekben, ahol az ilyen rendszereket használják egy túl későn kapott jó válasz gyakran ugyanolyan rossz, mintha egyáltalán nem kaptunk volna semmit (pl. robotpilóta).

A valós idejű rendszereket általában 2 csoportba soroljuk: **szigorú valós idejű rendszerek** (abszolút határidők vannak, amelyeket kötelező betartani), és **toleráns valós idejű rendszerek** (néha egy-egy határidő elmulasztása nem kívánatos ugyan, de tolerálható).

Az események, amelyekre egy valós idejű rendszernek válaszolnia kell tovább csoportosíthatók: **periodikusak** (rendszeres intervallumonként fordulnak elő) és **aperiodikusak** (megjósolhatatlan az előfordulásuk). Attól függően, hogy mennyi idő szükséges az egyes események feldolgozásához, előfordulhat, hogy nem tudja minden kezelní.

Azokat a valós idejű rendszereket, amelyek ezt a feltételt teljesítik, ütemezhetőnek nevezzük. A valós idejű ütemezési algoritmusok dinamikusak vagy statikusak lehetnek. Az előbbi az ütemezési döntéseket futás közben hozza, az utóbbi a rendszer futásának megkezdése előtt. A statikus ütemezés csak akkor működik, ha előre teljes információk van az elvégzendő feladatokról és a határidőkről is. A dinamikus algoritmusok esetében nincsenek ilyen korlátozások.

Kontextus-csere

Ha egyetlen CPU van és több egyidejűleg létező processzus, akkor a CPU váltakozva hajtja végre a processzusokat. A kontextus csere az, amikor a CPU átvált a P1 processzusról a P2 processzusra. Ehhez P1 állapotát a CPU regisztereiből el kell menteni az erre fenntartott memóriaterületre, majd P2 korábban elmentett állapotát helyre kell állítani a CPU regisztereiben. A kontextus cserére többprocesszoros rendszerekben is szükség van.

Összegzés

Processzusok

- végrehajtás alatt lévő program
- minden processzushoz tartozik:
 - saját címtartomány, ami tartalmazza a végrehajtandó programot, adatait és vermét
 - regiszterkészlet, beleértve az utasításszámlálót, veremmutatót, egyéb hardverregisztereket és a program futásához szükséges egyéb információkat
 - felhasznált erőforrások, amik lehetnek a megnyitott fájlok, gyermekprocesszusok, függőben lévő ébresztők, szignálkezelők, elszámolási információk
 - végrehajtási szál, amit rendszerint szálnak rövidítenek.
- a processzusokat leírhatjuk a hozzájuk tartozó processzustáblázat alapján
- ennek a memóriarésznek egyes bejegyzéseit processzus vezérő blokknak (PCB, Process Control Block) nevezzük, ami tartalmazza
 - a processzus azonosítóját,
 - állapotát,
 - a CPU állapotát (kontextus cseréhez),
 - a jogosultságokat
 - a birtokolt erőforrások listáját.
- a processzusoknak alapvetően 3 állapotuk lehet:
 - Futó (az adott pillanatban éppen használja a CPU-t)
 - Futáskész (ideiglenesen leállították egy másik processzor futása érdekében)
 - Blokkolt (bizonyos külső esemény bekövetkeztéig nem képes futni)
- egy processzus létrehozását 4 fő esemény okozhatja:
 - a rendszer inicializálása
 - a processzus által meghívott processzust létrehozó rendszerhívás végrehajtása
 - a felhasználó egy processzus létrehozását kéri
 - kötegelt feladat kezdeményezése
- előtérben futó processzusok a felhasználókkal tartják a kapcsolatot, vagy számukra munkát végeznek
- háttérprocesszusok: nincsenek egy bizonyos felhasználóhoz rendelve, ehelyett valamilyen sajátos feladatuk van (démonoknak)
- egy futó processzus adhat ki rendszerhívást új processzus létrehozására (pl. fordításkor make – C fordító – install).
- a létrehozás lépései:
 - Memóriaterület foglalása a PCB (Process Control Block) számára
 - PCB kitöltése iniciális adatokkal (pl. használható erőforrások, jogosultságok a létrehozó jogosultságaiból stb.)
 - Memória foglalás (programszöveg, adatok, verem számára)
 - A PCB processzusok láncára fűzése (az állapot beállítása futáskészre), ettől kezdve a processzus osztozik a CPU-n.
- A processzus létrehozása után elkezd dolgozni, és előbb vagy utóbb befejeződik, rendszerint a következő körülmenyek között:
 - Szabályos kilépés (önkéntes).
 - Kilépés hiba miatt (önkéntes).
 - Kilépés végzetes hiba miatt (önkéntelen).
 - Egy másik processzus megsemmisíti (önkéntelen).

- A befejezés lépései:
 - a gyermek processzusok megszüntetése.
 - a PCB processzusok láncáról való levétele (állapot beállítása terminálisra), ettől kezdve a processzus nem osztozik a CPU-n.
 - a processzus birtokában lévő erőforrások felszabadítása (PCB-beli nyilvántartás szerint, pl. nyitott fájlok lezárása).
 - a lefoglalt memóriaterületek felszabadítása (memóriatérkép alapján).
 - a PCB memóriaterületének felszabadítása.

Szálak/fonalak

- minden processzus saját címtartománnal és vezérlési szállal rendelkezik
- gyakran szükséges egy processzuson belül több párhuzamosan futó vezérlési szál használata úgy, mintha különálló processzusok lennének.
- a szál rendelkezik
 - utasításszámlálóval, (nyilvántartja, hogy melyik utasítás végrehajtása következik)
 - regiszterekkel (az aktuális munka változót tárolják)
 - veremmel (a végrehajtás eseményeit rögzíti, egy-egy kerettel minden meghívott eljáráshoz, amelyből nem tért még vissza a vezérlés)
- a szál és annak processzusa különböző fogalmak, így külön kezelhetők.
- szálak segítségével ugyanazon a processzuson belül több végrehajtást eszközölhetünk, amelyek egymástól nagymértékben függetlenek
- ha egy címtartományon több szál van, akkor egy különálló száltáblázatra van szükség
- a szálakat felfüggeszthetjük és folytathatjuk.
- amikor a szálakat felfüggesztjük, a regisztereiket el kell menteni
- a processzusokhoz hasonlóan futó, futáskész vagy blokkolt állapotban lehetnek.

Ütemezési stratégiák és algoritmusok

- CÉLOK
 - minden rendszer
 - Páratlanság – minden processzusnak megfelelő hozzáférést biztosítani a CPU-hoz.
 - Elvek betartása – a meghatározott elvek szerinti működés biztosítása.
 - Egyensúly – a rendszer minden részének egyenletes terhelése.
 - Kötegelt rendszerek
 - Áteresztőképesség – maximalizálni az időegységenként végrehajtott feladatok számát.
 - Áthaladási idő – minimalizálni a feladat-végrehajtás kezdeményezése és a befejezés között eltelt időt.
 - CPU-kihasználtság – a CPU soha nem állhat tétlenül.
 - Interaktív rendszerek
 - Válaszidő – a kérésekre gyors válasz biztosítása.
 - Arányosság – a felhasználók elvárásainak való megfelelés.
 - Valós idejű rendszerek
 - Határidők betartása – adatvesztés elkerülése.
 - Előrejelezhetőség – minőségromlás elkerülése multimédia-rendszerekben
- KÖTEGELETT RENDSZEREK
 - SORRENDEI ÜTEMEZÉS
 - egyik legegyszerűbb algoritmus

- olyan sorrendben osztja ki a CPU-t a processzusoknak, amilyen sorrendben azok kérik
 - a kész processzusok egyetlen várakozó soron állnak készenlétben
 - amikor az első feladat belép a rendszerbe azonnal indulhat, és addig futhat, ameddig akar
 - ha további feladatok érkeznek, azok a sor végére kerülnek.
 - ha egy futó processzus blokkolódik, a sor elején álló processzus futhat helyette, majd ha újra futáskész lesz, beáll a sor végére
 - könnyű megérteni és leprogramozni
 - ha nem szakít meg processzusokat, akkor nagyon hosszú ideig futahtnak.
- LEGRÖVIDEBB FELADAT ELŐSZÖR
 - feltételezi, hogy a futási idők előre ismertek
 - az ütemezőnek a legrövidebb feladatnak adja oda a CPU-t először
 - optimális abban az esetben, ha a feladatok egyszerre rendelkezésre állnak, és ismerjük a futásidőt
 - Előfordulhat hogy először egy sokáig tartó feladatnak adtuk oda a CPU-t, de később jött egy gyorsabb, és jobban jártunk volna, ha várunk egy kicsit
 - LEGRÖVIDEBB MARADÉK
 - megszakítható változata az előző algoritmusnak
 - az ütemező mindig azt a processzust választja, amelynek a legkevesebb a befejeződésig még megmaradt ideje (ismerni kell a futásidőt).
 - amikor új feladat érkezik, a teljes idő összehasonlításra kerül az éppen futó processzus még hátralévő idejével.
 - ha az új feladat kevesebb időt igényel, akkor az aktuálist lecseréli az újra
 - az új, rövid feladatok jó kiszolgálásban részesülnek.
 - HÁROMSZINTŰ ÜTEMEZÉS
 - az újonnan érkező feladatok először egy lemezen tárolt belépési várakozó sorba kerülnek.
 - a **bebocsátó ütemező** dönti el, hogy mely feladatok léphetnek be a rendszerbe
 - ha egy feladat már belépett a rendszerbe, akkor létre lehet hozni számára egy processzust, és elkezdhet vetélkedni a CPU-ért
 - megtörténhet, hogy olyan sok processzus van, hogy nem férnek a memoriába
 - néhány processzust ki kell helyezni a lemezre
 - az ütemezés **második szintje** azt dönti el, hogy melyik processzus maradjon a memoriában, és melyik kerüljön ki a lemezre (memóriaütemező dönti el)
 - a döntéseket bizonyos időnként felül kell vizsgálni, hogy a lemezen tárolt feladatoknak is legyen esélyük a bekerülésre.
 - rendszeres időközönként átnézi a lemezen tárolt processzusokat:
 - Mennyi idő telt el a processzus lemezre vitele óta?
 - Mennyi CPU időt használt fel a processzus nemrégiben?
 - Milyen nagy a processzus?
 - Mennyire fontos a processzus?
 - az ütemezés **harmadik szintje** választja ki valójában, hogy a futásra kész processzusok közül melyik fusson következőnek (CPU ütemező)

- INTERAKTÍV RENDSZEREK

- ROUND ROBIN

- minden processzusnak ki van osztva egy időintervallum (időszelet), ami alatt engedélyezett a futása
 - az ütemezőnek minden processzusnak meg kell adnia a futtatandó processzusokról
 - ha egy processzus felhasználja az időszeletét, akkor a lista végére kerül.
 - ha az időszelet túl kicsire állítása túl sok processzus átkapcsolást okoz, és csökken a CPU hatékonysága,
 - túl nagy rövid interaktív kérésekre gyenge válaszidőt eredményezhet.
 - 20-50 ezred másodperc körüli értéke ésszerű kompromisszum.

- PRIORITÁSOS ÜTEMEZÉS

- mindegyik processzushoz rendeljünk egy prioritást, és a legmagasabb prioritású futásra kész processzusnak engedjük meg, hogy fussen
 - hogy a magas prioritású processzusok ne fussanak végtelen ideig, az ütemező minden óraütemben csökkentheti az éppen futó processzus prioritását.
 - a prioritásokat statikusan vagy dinamikusan lehet megadni
 - prioritási osztályokat felállítása, azon belül round robin alkalmazása

- TÖBBSZÖRÖS SOROK

- a CPU igényes processzusoknak inkább nagyobb időszeletet adni, mint gyakran kicsit,
 - prioritási osztályok felállítása
 - a magasabb osztályban lévő processzusok több időszeletet kapnak
 - valahányszor egy processzus elhasználja számára biztosított időt, egy osztállyal lejebb kerül
 - jó olyan processzusoknak, amelyek induláskor hosszú időt igényeltek, de később interaktívvá váltak.

- LEGRÖVIDEBB PROCESSZUS KÖVETKEZZEN

- becsléseket végez a múltbeli viselkedés alapján
 - a legkisebb becsült futásidő alapján választunk processzort
 - a következő elemet úgy becsüljük, hogy vesszük az éppen mért értékek és az előző becslések súlyozott átlagát (öregedésnek)
 - olyan esetben alkalmazható, amikor a becslést az előző értékekre alapozva kell elvégezni.

- GARANTÁLT ÜTEMEZÉS

- ígéreteket tesz a felhasználónak a teljesítménnyel kapcsolatban
 - a rendszernek nyomon kell követnie, hogy egy processzus mennyi CPU-időt kapott létrehozása óta és hogy eddig mennyit használt el
 - mindenikhez kiszámítja a neki járó mennyiséget, majd mindenik processzushoz kiszámolja az aktuálisan felhasznált és a neki járó CPU-idő arányát, és a lelkisebb arányszámmal rendelkező processzust fogja futtatni.

- SORSJÁTÉK ÜTEMEZÉS
 - ígéretet tesz a felhasználónak,
 - könnyebb a megvalósítása
 - minden processzusnak sorsjegyet adunk a különböző erőforrásokhoz
 - ha ütemezési döntést kell hozni, egy sorsjegyet véletlenszerűen kiválasztunk, és az a processzus kapja meg az erőforrást, amelynél a sorsjegy van
 - a fontosabb processzusok több sorsjegyeket kaphatnak, hogy növeljék a nyerési esélyeiket. Ezzel az ütemezéssel nagyon jó a válaszidő.
- ARÁNYOS ÜTEMEZÉS
 - figyelembe veszi, hogy ki a processzus tulajdonosa
 - minden felhasználó kap valamekkora hányadot a CPU-időből, és az ütemező úgy választja ki a folyamatokat, hogy ezt kikényszerítse
- VALÓS IDEJŰ RENDSZEREK
 - egy vagy több külső fizikai eszköz ingert küld a számítógép felé, amire annak megfelelően reagálnia kell egy adott időn belül
 - egy túl későn kapott jó válasz gyakran ugyanolyan rossz, mintha egyáltalán nem kaptunk volna semmit (pl. robotpilóta).
 - szigorú valós idejű rendszerek (abszolút határidők vannak, amelyeket kötelező betartani),
 - toleráns valós idejű rendszerek (néha egy-egy határidő elmulasztása nem kívánatos ugyan, de tolerálható).
 - periodikus események (rendszeres intervallumonként fordulnak elő)
 - aperiodikus események (megjósolhatatlan az előfordulásuk).
 - előfordulhat, hogy nem tudja az összes eseményt kezelni
 - a valós idejű ütemezési algoritmusok dinamikusak (futás közben hoz döntést) vagy statikusak (futás megkezdése előtt hoz döntést) lehetnek

Kontextus-csere

- ha egyetlen CPU van és több egyidejűleg létező processzus, akkor a CPU váltakozva hajtja végre a processzusokat
- a kontextus csere az, amikor a CPU átvált a P1 processzusról a P2 processzusra
- P1 állapotát a CPU regisztereiből el kell menteni az erre fenntartott memóriaterületre
- P2 korábban elmentett állapotát helyre kell állítani a CPU regisztereiben
- többprocesszoros rendszerekben is szükség van rá

Operációs rendszerek - 2.)

Processzusok kommunikációja, versenyhelyzetek, kölcsönös kizárási. Konkurens és kooperatív processzusok. Kritikus szekciók és megvalósítási módszereik: kölcsönös kizárási 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, vészszituációk, kölcsönös kizárási

A processzusoknak gyakran szükségük van az egymással való kommunikációra (IPC – InterProcess Communication). Például egy parancsértelemező adatcsőben, amikor az első processzus kimenő adatait át kell adni a második processzusnak, és sorban így tovább. Ezért szükséges a processzusok közti kommunikáció, előnyben részesítve egy megszakítások nélküli, jól strukturált módot.

A kommunikációnak három fő területe van:

- Hogyan tud egy processzus információt küldeni egy másiknak?
- Hogyan tudjuk biztosítani, hogy kettő vagy több processzus ne keresztezze egymás útját, amikor kritikus tevékenységekbe kezdenek?
- Függőségek esetén hogyan állítsuk sorrendbe a processzusokat? (Például, ha az A processzus adatokat állít elő, és a B processzus kinyomtatja azt, akkor B-nek várnia kell, míg az A néhány adatot előkészít, a nyomtatás csak ez után kezdődhet.)

Vannak operációs rendszerek, ahol az együtt dolgozó processzusok közös tárolóterületen osztozhatnak. A megosztott tároló lehet a főmemóriában, vagy egy megosztott fájlban, ez nem változtat a kommunikáció természetén vagy a felmerülő problémákon.

Azokat az eseteket, amikor kettő vagy több processzus olvas vagy ír megosztott adatokat, és a végeredmény attól függ, hogy ki és pontosan mikor fut, versenyhelyzeteknek nevezzük. A versenyhelyzeteket tartalmazó programokat igen nehéz nyomon követni, tesztelés során néha előfordulnak furcsa és látszólag megmagyarázhatatlan dolgok.

Általános probléma a háttérnyomtatás. A kliens nyomtatáskor beteszi a fájl nevét egy háttérkatalógusba. A háttérkatalógus fájlnevek tárolására alkalmas, sorszámmal azonosított rekeszeket tartalmaz. A nyomtató démon rendszeresen ellenőrzi, hogy kell-e nyomtatni. Ha kell, akkor kinyomtatja az adott dolgot, majd kitörli a nevét a katalógusból. Tegyük fel, hogy van 2 megosztott változónk, az out, mely a következő nyomtatandó fájlra mutat, és in, ami pedig a katalógus első szabad rekeszére. Ekkor megtörténhet, hogy az A processzus olvassa az in értékét és eltárolja lokális változóban. Ezután kontextus csere történik, és B eltárolja az állományt, majd az in-t frissíti az új értékkal. A folytatja a futását, felülírja a korábbi rekesz tartalmát – kitörli a B által beírt állomány nevét, majd frissíti az in értékét.

A kölcsönös kizárási egy mód a versenyhelyzetek elkerülésére. Itt is és sok más esetben is, ahol megosztott memória, megosztott fájlok, vagy bármi más megosztott dolog szerepel, a baj megelőzésének kulcsa az, hogy találunk valamilyen módot annak megtiltására, hogy egy időben egynél több processzus olvassa és írja a megosztott adatokat.

Bármely operációs rendszer egyik fő tervezési szempontja, hogy megválasszuk az alkalmas primitív műveleteket a kölcsönös kizárás eléréséhez. A konkurens processzusok küzdenek a CPU-ért egymással.

A konkurens processzusok küzdenek a CPU-ért egymással, a kooperatív processzusok pedig bizonyos időnként lemondanak a CPU használatáról, átengedve azt egy másiknak. Semmi sem kötelezi azonban arra, hogy ezt megtegye, tehát tetszőleges ideig lefoglalhatja az erőforrást, a többi folyamatot várakozásra kényszerítve.

Kritikus szekciók és megvalósítási módszereik

Az idő egy részében a processzus belső számolási is egyéb olyan tevékenységekkel van elfoglalva, amelyek nem vezetnek versenyhelyzetekhez, azonban néha a processzus megosztott memóriához vagy fájlokhoz nyúl. **A programnak azt a részét, amelyben a megosztott memóriát használja, kritikus területnek vagy kritikus szekciónak nevezzük.**

Ha úgy tudnánk rendezni a dolgokat, hogy soha ne legyen azonos időben két processzus a kritikus szekciójában, akkor elkerülhetnék a versenyhelyzeteket. Bár ez a követelmény megőv a versenyhelyzetektől, mégsem elegendő ahhoz, hogy korrektan együttműködő párhuzamos processzusaink legyenek, és azok hatékonyan használják a megosztott adatokat. A jó megoldáshoz négy feltételt kell betartani:

- Ne legyen két processzus egyszerre a saját kritikus szekciójában.
- Semmilyen előfeltétel ne legyen a sebességekről vagy a CPU-k számáról.
- Egyetlen, a kritikus szekcióján kívül futó processzus sem blokkolhat más processzusokat.
- Egyetlen processzusnak se kelljen örökké arra várni, hogy belépjen a kritikus szekciójába.

Kölcsönös kizárás A kölcsönös kizárás megvalósítására számos különböző módszer létezik. A kölcsönös kizárás röviden összefoglalva annyit jelent, hogy miközben egy processzus azzal van elfoglalva, hogy a kritikus szekciójában a megosztott memóriát aktualizálja, ne legyen más olyan processzus, amely belép az Ő saját kritikus szekciójába ezzel bajt okozva.

Azt, amikor folyamatosan tesztelünk egy változót egy bizonyos érték megjelenéséig, **tevékeny várakozásnak** nevezzük. Általában tartózkodni kellene ettől, mert pazarolja a CPU időt. Csak akkor használjuk a tevékeny várakozást, ha ésszerűen elvárható, hogy a várakozás rövid lesz. A tevékeny várakozást használó zárolásokat aktív várakozásnak hívjuk.

Altatás és ébresztés

Termelő-fogyasztó probléma: két processzus osztottak egy közös, rögzített méretű tárolón. Az egyikük, a gyártó, adatokat helyez el benne, a másik, a fogyasztó, kiveszi azokat. A nehézség akkor jelentkezik, amikor a gyártó új elemet kíván a tárolóba tenni, de az már tele van. A megoldás a gyártó számára az, hogy elalszik, és felébresztik, amikor a fogyasztó egy vagy több elemet kivett. Hasonlóképpen, ha a fogyasztó szeretne egy elemet kivenni a tárolóból és látja, hogy a tároló üres, akkor elalszik, amíg a gyártó tesz valamit a tárolóba és felébreszti őt.

A Szemafor egy változótípus, amiben számolhatjuk az ébresztéseket későbbi felhasználás céljából. A szemafor értéke lehet 0, jelezve, hogy nincs elmentett ébresztés, vagy valamilyen

pozitív érték, ha egy vagy több ébresztés van függőben. Két alapvető művelet van, a **down** és az **up** (rendre a **sleep** és a **wakeup** általánosításai).

A **down** művelet megvizsgálja, hogy a szemafor értéke nagyobb-e, mint 0. Ha igen, csökkenti az értékét (vagyis felhasznál egy tárolt ébresztést), és azonnal folytatja. Ha az érték 0, akkor a processzust elaltatja, mielőtt a **down** befejeződne. Az érték ellenőrzése, cseréje és a lehetséges elalvás együttesen oszthatatlan elemi műveletként hajtódik végre. Ez garantálja, hogy ha egy szemafor művelet elkezdődik, más processzus nem tudja elérni a szemafort mindaddig, amíg a művelet be nem fejeződik vagy nem blokkolódik. Az elemi művelet bevezetése nagyon lényeges a szinkronizációs problémák megoldásához és a versenyhelyzetek elkerüléséhez.

Az **up** művelet a megadott szemafor értékét növeli. Ha egy vagy több processzus aludna ezen a szemaforon, mivel képtelen volt befejezni egy korábbi down műveletet, akkor közülük az egyiket kiválasztja a rendszer, és megengedi neki, hogy befejezze a **down** műveletét. Így olyan szemaforon végrehajtva az **up** műveletet, amelyen processzusok aludtak, a szemafor még mindig 0 lesz, de eggyel kevesebb processzus fog rajta aludni. A szemafor növelésének és egy processzus felébresztésének művelete szintén nem választható szét. Egy **up** műveletet végrehajtó processzus nem blokkolható, mint ahogy az előző modellben a **wakeup** végrehajtása sem volt az.

Az **up** és a **down** rendszerhívásként kerül megvalósításra, amelyben az operációs rendszer egyszerűen tilt minden megszakítást, mialatt vizsgálja és aktualizálja a szemafort, valamint elaltatja a processzust, ha kell. Ha több CPU-t használunk, minden szemafort védeni kell egy zárolásváltozóval, a TSL utasítást használva, annak érdekében, hogy egy időben csak egy CPU vizsgálja a szemafort.

A Mutex-ek a szemafor egyszerűsített változatai, csak bizonyos erőforrások vagy kódrészkek kölcsönös kizárásnak kezelésére alkalmasak. Ez egy olyan változó, amely kétféle állapotban lehet: nem zárt vagy zárt. Ennek következtében egyetlen bit is elegendő a reprezentálásához, de a gyakorlatban gyakran egy egész értéket használnak, ahol a 0 jelenti a nem zárt, és bármilyen más érték a zárt állapotot.

Két eljárás használatos a mutexek esetében. Amikor egy processzus (vagy szál) hozzá szeretne férni a kritikus szekcióhoz, meghívja a **mutex_lock** eljárást. Ha a mutex pillanatnyilag nem zárt, akkor a hívás sikeres, és a hívó szál szabadon beléphet a kritikus szekcióba. Ha a mutex már zárt állapotban van, akkor a hívó blokkolódik, amíg a kritikus szekcióban lévő processzus nem végez, és meg nem hívja a **mutex_unlock** eljárást. Ekkor, ha több processzus is blokkolódik a mutexen, közülük az egyik véletlenszerűen kiválasztott szerezheti meg a zárolást.

Az olyan helyzeteket, ahol minden lehetséges processzus blokkolt, hol pontnak nevezik. A **monitor** eljárások, változók és adatszerkezetek együttese, mindezek egy speciális fajta modulba vagy csomagba összegyűjtve. A processzusok bármikor hívhatják a monitorban lévő eljárásokat, de nem érhetik el közvetlenül a monitor belső adatszerkezeteit a monitoron kívül deklarált eljárásokból. minden időpillanatban csak egy processzus lehet aktív egy monitorban. A monitorok programozási nyelvi konstrukciók, ezért a fordítóprogram tudja, hogy ezek speciálisak, és képes a monitoreljárás-hívásokat másképpen kezelni, mint az egyéb eljáráshívásokat. Ha nem használja másik processzus a monitort, akkor a hívó beléphet. Általános módszer a mutexek vagy bináris szemaforok használata a monitorok belépési pontjainál. Mivel fordítóprogram és nem a programozó intézi el a kölcsönös kizárást, kisebb a valószínűsége, hogy valami elromlik.

Üzenetek, adás, vétel

Az írók és olvasók problémája egy adatbázis elérését modellezi. Az elfogadható, hogy több processzus egyidejűleg olvasson az adatbázisból, de ha egy processzus aktualizálja (írja) az adatbázist, akkor azt más processzusoknak nem szabad elérniük, még az olvasóknak sem. A kérdés az, hogy hogyan programozzuk az olvasókat és az írókat.

Tegyük fel, hogy mialatt egy olvasó használja az adatbázist, egy másik olvasó érkezik. Mivel nem probléma, ha két olvasó van egy időben, ezért bebocsátjuk. A harmadik és további olvasókat is bebocsátjuk, ha érkeznek.

Most tegyük fel, hogy egy író érkezik. Az írót nem engedhetjük be az adatbázisba, mert az íróknak kizárolagos hozzáférésre van szükségük, így az író felfüggesztődik. Később további olvasók jelennek meg. Amíg van legalább egy aktív olvasó, további olvasók jöhetsznek. Ennek a stratégiának az a következménye, hogy ha az olvasóknak folyamatos utánpótlása van, akkor azok megérkezésük után azonnal bejuthatnak.

Az író mindenkorában felfüggesztett állapotban marad, amíg az olvasók el nem fogynak. Ha mondjuk 2 másodpercenként jön egy olvasó, és minden olvasónak 5 másodperces munkája van, az író soha nem kerül be. Van ennek a megoldásnak viszont egy olyan változata is, ami az írónak ad prioritást.

A **sorompó** egy primitív könyvtári eljárás, lényege, hogy fázisokra osztjuk az alkalmazást. A szabály az, hogy egyetlen processzus sem mehet tovább a következő fázisra, amíg az összes processzus készen nem áll. Mindegyik fázis végére elhelyezünk egy sorompót. Amikor egy processzus a sorompóhoz ér, akkor addig blokkolódik, ameddig az összes többi processzus el nem éri a sorompót. A sorompó az utolsó processzus beérkezése után elengedi azokat. Hasznos például nagy mátrixokon végezhető párhuzamos műveletek esetében.

Összegzés

Adatbázisok - 1.)

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

Adatok típusai:

- Egyszerű (atomi) adat: szám, string, dátum, logikai érték.
- Összetett adat: egyszerű adatokból képezhető. Változatai:
 - halmaz: egynemű elemek halmaza. Példa: egy vállalat osztályai.
 - lista: egynemű elemek rendezett sorozata. Példa: könyv szerzői.
 - struktúra: különféle elemek rendezett sorozata (lakcím = (helység, utca, házszám))
 - a fentiek kombinációi.
- NULL: definiálatlan adat. (Nem azonos a nulla értékkel!)

Elnévezések:

- Adatbázis (DB = database): adott formátum és rendszer szerint tárolt adatok együttese.
- Adatbázis-kezelő rendszer (DBMS = Database Management System): az adatbázist kezelő szoftver. Pl.: MySQL, Oracle, Access, DBeaver, PGAdmin
- Rekord (feljegyzés): az adatbázis alapvető adategysége. Általában struktúra felépítésű.

Egy adatbázis-alkalmazásnál az alábbi szinteket különböztethetjük meg:

- Felhasználói felület
- Célalkalmazásként készített program
- Adatmodell (logikai adatstruktúra)
- DBMS
- Fizikai adatstruktúra

Az egyed-kapcsolat (ER) diagram

Az egyed-kapcsolat diagram, az adatok logikai modelljét, egymáshoz viszonyított relációját mutatja meg. A valós világ jelenségeit egyedekkel, tulajdonságokkal és kapcsolatokkal leíró modellt egyed-kapcsolat modellnek, az ezt ábrázoló diagramot egyed-kapcsolat diagramnak nevezik. Az adatbázis logikai modelljének elkészítésekor szem előtt kell tartanunk, hogy miiről szeretnénk milyen adatokat tárolni, és ezek hogyan viszonyulnak egymáshoz. Elemei:

Egyed (entity): valós világban létező doleg, amit tulajdonságokkal akarunk leírni (jele: téglalap)

Attribútum: az egyed egy jellemzője (jele: ellipszis)

Kapcsolat: két vagy több egyed között határoz meg relációt (jele: rombusz)

Összetett attribútum: maga is attribútumokkal rendelkezik

Többértékű attribútum: aktuális értéke halmaz vagy lista lehet (jele: kettős ellipszis)

Gyenge egyed: az attribútumai nem határozzák meg egyértelműen, csak a kapcsolatai révén lesz meghatározott (jele: kettős téglalap)

Meghatározó kapcsolat: gyenge entitást meghatározó kapcsolat (jele: kettős rombusz)

Specializáló kapcsolatok: olyan kapcsolat, amely hierarchiát jelöl az egyedek között (jele: háromszög, amelynek csúcsa a főtípus felé mutat)

A relációs adatmodell

Lényege, hogy az egyedeket, tulajdonságokat és kapcsolatokat egyaránt táblázatok (adattáblák) segítségével határozzuk meg. Az adattábla sorokból és oszlopokból áll. Egy sorát rekordnak nevezzük, amely annyi mezőből áll, ahány oszlopa van a táblának.

Relációsémának nevezünk egy attribútumhalmazt, amelyhez azonosító nevet rendelünk.

- $R(A_1, \dots, A_n)$: ahol R a relációséma neve, A_1, \dots, A_n , pedig az attribútumhalmaza
- minden attribútumhoz tartozik egy értékkészlet, amelyből felveheti értékeit.

Attribútumnak nevezünk egy tulajdonságot, amelyet a megnevezésével azonosítunk, és értéktartományt rendelünk hozzá. A Z attribútum értéktartományát a domain szó rövidítésével jelöljük: $\text{dom}(Z)$.

Szuperkulcs: Azt a $K \subseteq \{A_1, \dots, A_n\}$ attribútumhalmazt, amelyen a T tábla minden sora különbözik szuperkulcsnak nevezzük (egyértelműen azonosítja a tábla sorait).

Kulcs: Az A attribútumhalmaz K részhalmazát kulcsnak nevezzük, ha minimális szuperkulcs, vagyis egyetlen valódi részhalmaza sem szuperkulcs. Ha K egyetlen attribútumból áll, akkor egyszerű, egyébként összetett kulcsról beszélünk.

Elsődleges kulcs: ha több kulcs is meghatározható a sémában, akkor kijelöljük az egyiket, amelyet elsődleges kulcsnak nevezünk. Egy relációsémában tehát mindig csak egy elsődleges kulcs lehet. Az elsődleges kulcsot alkotó attribútumokat aláhúzással szokás jelölni.

Külső kulcs: Egy relációséma attribútumainak valamely részhalmaza külső kulcs (másnéven idegen kulcs, angolul foreign key), ha egy másik séma elsődleges kulcsára hivatkozik. A külső kulcs értéke a hivatkozott táblában előforduló kulcsérték vagy NULL lehet. A külső kulcsot dőlt betűvel, vagy a hivatkozott kulcsra mutató nyíllal jelöljük.

Ha egy adatbázis valamennyi táblájának sémáját felírjuk a kulcsok és külső kulcsok jelölésével együtt, akkor **relációs adatbázissémát** kapunk.

Az EK diagram leképezése relációs modellre

Egyedek leképezése: minden egyedhez felírunk egy relációsémát, amelynek neve az egyed neve, attribútumai az egyed attribútumai, elsődleges kulcsa az egyed kulcs- attribútuma(i).

Személy (személyi szám, név, születési dátum)

Gyenge entitások leképezése: a gyenge entitás relációsémáját bővíteni kell a meghatározó kapcsolat(ok)ban szereplő egyed(ek) kulcsával. Pl.: Ha egy tulajdonosnak több, azonos gépe lehet, akkor ezeket egy sorszám attribútummal különböztetjük meg:

Tulajdonos (személyiszám, név, lakcím)

Számítógép (processzor, memória, merevlemez, személyiszám, sorszám)

Összetett attribútumok leképezése: Az összetett attribútumot a részattribútumaival helyettesítjük a leképezés során.

Személy (személyi szám, név, születési dátum)

Többértékű attribútum leképezése: A többértékű attribútum számára egy külön relációsémát hozunk létre, amelyben feltüntetjük az attribútumot és az egyed kulcsát

Könyv (ISBN, cím, kiadási év)

Szerző (ISBN, szerző)

Funkcionális függőség

Legyen $R(A_1, \dots, A_n)$ egy relációséma, és P, Q az $\{A_1, \dots, A_n\}$ attribútumhalmaz részhalmazai. P -től funkcionálisan függ Q (jelölésben $P \rightarrow Q$), ha bármely R feletti T tábla esetén valahányszor két sor megegyezik P -n, akkor megegyezik Q -n is.

{osztálykód} → {osztálynév, vezAdószám}

{adószám} → {név, lakcím, osztálykód, osztálynév, vezAdószám}

Ebben a példában az adószám funkcionálisan függ az osztálykódtól, ugyanis feltehetően ha az adószám esetében megegyezik egy rekordban az osztálykód és a vezAdószám értéke, akkor az osztálykód esetében is meg fog egyezni.

Normálformák

Ha az egyed-kapcsolat modellt helyesen írjuk fel, akkor általában optimális (redundanciamentes) relációs adatbázis sémát kapunk. Semmi garancia nincs azonban arra, hogy az E-K modell optimális, ezért szükség van a relációsémák formális vizsgálatára, amely a redundanciákat detektálja és az optimalizálást, **normalizálást** tesz lehetővé.

Először is tekintsük meg azt, hogy mit is jelent, ha egy **adattábla redundáns**. Hogy egyszerűbb legyen ezt megérteni, vegyük az alábbi sémát, és a hozzá tartozó táblát:

DOLGOZÓ (név, adószám, cím, osztálykód, osztálynév, vezAdószám)

Név	Adószám	Cím	Osztálykód	Osztálynév	VezAdószám
Kovács	1111	Pécs, Vár u.5.	2	Tervezési	8888
Tóth	2222	Tata, Tó u.2.	1	Munkaügyi	3333
Kovács	3333	Vác, Róka u.1.	1	Munkaügyi	3333
Török	8888	Pécs, Sas u.8.	2	Tervezési	8888
Kiss	4444	Pápa, Kő tér 2.	3	Kutatási	4444
Takács	5555	Győr, Pap u. 7.	1	Munkaügyi	3333
Fekete	6666	Pécs, Hegy u.5.	3	Kutatási	4444
Nagy	7777	Pécs, Cső u.25.	3	Kutatási	4444

A táblában lévő **redundancia** aktualizálási anomáliákat okozhat, például:

- 1.) Módosítás esetén, ha egy osztály neve, vagy vezetője megváltozik, akkor több helyen kell a módosítást elvégezni.
- 2.) Új rekord felvétele esetén simán előfordulhat, hogy az osztály nevét rosszul adják meg (mondjuk tervezési, vagy Tervező), vagy ha új osztályt szeretnénk létrehozni, akkor be kell

szúrunk egy minden oszloban NULL-t tartalmazó rekordot az új osztály nevével, ami később feleslegessé válik

3.) Törlés esetén ha egy osztály összes dolgozóját töröljük, maga az osztály is elveszik

Mi lehet erre a megoldás? Bontsuk fel két sémára (dekompozíció)!

DOLGOZÓ (név, adószám, cím, osztálykód)

OSZTÁLY (osztálykód, osztálynév, vezAdószám)

<i>Név</i>	<i>Adószám</i>	<i>Cím</i>	<i>Osztálykód</i>
Kovács	1111	Pécs, Vár u.5.	2
Tóth	2222	Tata, Tó u.2.	1
Kovács	3333	Vác, Róka u.1.	1
Török	8888	Pécs, Sas u.8.	2
Kiss	4444	Pápa, Kő tér 2.	3
Takács	5555	Győr, Pap u. 7.	1
Fekete	6666	Pécs, Hegy u.5.	3
Nagy	7777	Pécs, Cső u.25.	3

<i>Osztálykód</i>	<i>Osztálynév</i>	<i>VezAdószám</i>
1	Munkaügyi	3333
2	Tervezési	8888
3	Kutatási	4444

Egy helyesen felírt EK diagram leképezése során eredetileg ehhez kellett volna jutnunk. A továbbiakban azt vizsgáljuk, hogy mikor van egy táblában redundancia, és hogyan kell ezt a tábla felbontásával megszüntetni.

1. NORMÁLFORMA (1NF)

Egy relációséma 1NF-ben van, ha az attribútumok értéktartománya csak egyszerű (atomi) adatokból áll (nem tartalmaz például listát vagy struktúrát). Mivel az 1NF feltétel teljesülését már a relációs modell definíciójánál kikötöttük, ezért minden sémát eleve 1NF-nek tételezünk fel.

2. NORMÁLFORMA (2NF)

Legyen

- X,
- $Y \subseteq A$ (Y valódi részhalmaza A -nak),
- és $X \rightarrow Y$ (X funkcionálisan függ Y -től)

Azt mondjuk, hogy X -től **teljesen függ** Y , ha X -ből bármely attribútumot elhagyva a függőség már nem teljesül, vagyis bármely $X_1 \subset X$ esetén $X_1 \rightarrow Y$ már nem igaz (X bármely valódi részhalmazával a funkcionális függőség már nem áll fenn). Ha egy séma nincs 2NF-ben, akkor már előfordulhat redundancia.

Egy attribútumot **elsődleges attribútumnak** nevezünk, ha szerepel a relációséma valamely kulcsában, ellenkező esetben másodlagos attribútum.

Egy relációséma akkor van 2NF-ben, ha minden másodlagos attribútum teljesen függ bármely kulcstól, tehát ha minden kulcs egy attribútumból áll, vagy nincsenek másodlagos attribútumok, akkor a séma 2NF-ben van.

Vegyük egy példát, azonnal érhető lesz (a tábla a következő oldalon látható):

DOLGPROJ (Adószám, Név, Projektkód, Óra, Projektnév, Projekthely)

<i>Adószám</i>	<i>Név</i>	<i>Projektkód</i>	<i>Óra</i>	<i>Projektnév</i>	<i>Projekthely</i>
1111	Kovács	P2	4	Adatmodell	Veszprém
2222	Tóth	P1	6	Hardware	Budapest
4444	Kiss	P1	5	Hardware	Budapest
1111	Kovács	P1	2	Hardware	Budapest
1111	Kovács	P5	8	Teszt	Szeged
8888	Török	P2	12	Adatmodell	Veszprém
5555	Takács	P5	3	Teszt	Szeged
6666	Fekete	P5	4	Teszt	Szeged
8888	Török	P3	4	Software	Veszprém
7777	Nagy	P3	14	Software	Veszprém

Itt az alábbi függőségeket figyelhetjük meg:

Adószám → Név

Projektkód → {Projektnév, Projekthely}

{Adószám, Projektkód} → Óra

A sémban {Adószám, Projektkód} kulcs, mivel ettől minden attribútum függ, ugyanakkor akár Adószámot, akár Projektkódot elhagyva ez már nem teljesül. A séma nincs 2NF-ben, mert pl. Név csak Adószám-tól függ, vagyis nem függ teljesen a kulcstól.

Hozzuk akkor 2NF-re! Dekompozícióval válasszuk le az Adószám → Név függőséget:

DOLGOZÓ (Adószám, Név)

PROJ (Adószám, Projektkód, Óra, Projektnév, Projekthely)

A második séma a Projektkód → {Projektnév, Projekthely} függőség miatt nincs 2NF-ben.

DOLG (Adószám, Név)

PROJ (Projektkód, Projektnév, Projekthely)

DP (Adószám, Projektkód, Óra)

<i>Adószám</i>	<i>Név</i>	<i>Adószám</i>	<i>Projektkód</i>	<i>Óra</i>
1111	Kovács	1111	P2	4
2222	Tóth	2222	P1	6
4444	Kiss	4444	P1	5
8888	Török	1111	P1	2
5555	Takács	1111	P5	8
6666	Fekete	8888	P2	12
7777	Nagy	5555	P5	3
		6666	P5	4
		8888	P3	4
		8888	P3	4
<i>Projektkód</i>	<i>Projektnév</i>	<i>Projekthely</i>		
P1	Hardware	Budapest		
P2	Adatmodell	Veszprém		
P3	Software	Veszprém		
P5	Teszt	Szeged		

3. NORMÁLFORMA (3NF)

Legyen:

- X ,
- $Z \subseteq A$ (Z valódi részhalmaza A -nak)
- és $X \rightarrow Z$ (X funkcionálisan függ Z -től)

Azt mondjuk, hogy X -től **tranzitívan függ** Z , ha van olyan $Y \subseteq A$, amelyre $X \rightarrow Y$ és $Y \rightarrow Z$, de X nem függ Y -tól, és az $Y \rightarrow Z$ függés teljesen nem triviális. Ellenkező esetben Z közvetlenül függ X -től.

Egy relációséma 3NF-ben van, ha minden másodlagos attribútuma közvetlenül függ bármely kulcstól, vagy ha a sémában nincs másodlagos attribútum.

Összegzés

Adatbázisok - 2.)

Az SQL adatbázisnyelv: Az adatdefiníciós nyelv (DDL) és az adatmanipulációs nyelv (DML). Relációsémák definiálása, megszorítások típusai és létrehozásuk. Adatmanipulációs lehetőségek és lekérdezések.

Az SQL adatbázisnyelv

SQL = Structured Query Language (strukturált lekérdező nyelv) a relációs adatbázis-kezelés szabványos nyelve. Nem algoritmikus nyelv, de algoritmikus nyelvekbe beépíthető (beágyazott SQL). Nagyon sok típusa van, melyek minden saját bővítményeket adnak hozzá a nyelv alapvető elemeihez (ilyen például a PostgreSQL, PLSQL, stb). Az SQL utasításait két fő csoportba szokták sorolni:

- DDL-Adatdefiníciós nyelv (Data Definition Language): adatstruktúra definiáló utasítások.
- DML-Adatmanipulációs nyelv (Data Manipulation Language): adatokon műveletet végző utasítások.

Relációsémák definiálásai, megszorítások típusai és létrehozásuk

Relációséma létrehozására a **CREATE TABLE** utasítás szolgál, amely egyben egy üres táblát is létrehoz a sémához. Az attribútumok definiálása mellett a kulcsok és külső kulcsok megadására is lehetőséget nyújt:

Adattípusok

- **CHAR(n)**: n hosszúságú karaktersorozat
- **VARCHAR(n)**: legfeljebb n hosszúságú karaktersorozat
- **INTEGER**: egész szám (röviden INT)
- **REAL**: valós (lebegőpontos) szám, másnéven FLOAT
- **DECIMAL(n[,d])**: n jegyű decimális szám, ebből d tizedesjegy
- **DATE**: dátum (év, hónap, nap)
- **TIME**: idő (óra, perc, másodperc)

Feltételek (egy adott oszlopra vonatkoznak):

- **PRIMARY KEY**: elsődleges kulcs (csak egy lehet)
- **UNIQUE**: kulcs (több is lehet)
- **REFERENCES tábla(oszlop) [ON-feltételek]**: külső kulcs

Táblafeltételek (az egész táblára vonatkoznak):

- **PRIMARY KEY (oszloplista)**: elsődleges kulcs
- **UNIQUE (oszloplista)**: kulcs
- **FOREIGN KEY (oszloplista) REFERENCES tábla(oszloplista) [ON-feltételek]**: külső kulcs. ON-feltételek megadásával szabályozhatjuk a rendszer viselkedését (T1 a hivatkozó, T2 a hivatkozott tábla):
- **ON UPDATE/DELETE CASCADE**: ha T2 egy sorában változik a kulcs értéke, akkor a rá való T1-beli hivatkozások is megfelelően módosulnak (módosítás továbbgyűrűzése).
- **ON UPDATE/DELETE SET NULL**: ha T2 egy sorában változik a kulcs értéke, akkor T1-ben a rá való külső kulcs hivatkozások értéke NULL lesz.
- **DROP TABLE**: Hatására a séma és a hozzá tartozó adattábla törlődik.
- **ALTER TABLE**: Relációséma módosítása

Adatmanipulációs lehetőségek és lekérdezések

- **INSERT INTO táblanév [(oszloplista)] VALUES (értéklista):** értékek beszúrása a megadott tábla megadott oszlopaiba. Ha oszloplista nem szerepel, akkor valamennyi oszlop értéket kap a CREATE TABLE-ben megadott sorrendben
- **UPDATE táblanév SET oszlop = kifejezés, ..., oszlop = kifejezés [WHERE feltétel]:** értékek aktualizálása. Az értékkedés minden olyan soron végrehajtódik, amely eleget tesz a WHERE feltételnek. Ha WHERE feltétel nem szerepel, akkor az értékkedés az összes sorra megtörténik.
- **DELETE FROM táblanév [WHERE feltétel]:** értékek törlése. Hatására azok a sorok törlődnek, amelyek eleget tesznek a WHERE feltételnek. Ha a WHERE feltételt elhagyjuk, akkor az összes sor törlődik

Lekérdezésre a SELECT utasítás szolgál, amely egy vagy több adattáblából egy eredménytáblát állít elő. Az eredménytábla a képernyőn listázásra kerül, vagy más módon használható fel.

SELECT [DISTINCT] oszloplista FROM táblanevek [WHERE feltétel];

Ha oszloplista helyére * karaktert írunk, ez az összes oszlop felsorolásával egyenértékű.

A **DISTINCT** opciót akkor kell kiírni, ha az eredménytáblában csak a különböző értékeket szerethnénk megjeleníteni. A **SELECT** után megadott oszloplista valójában nem csak oszlopneveket, hanem tetszőleges kifejezéseket is tartalmazhat (mint például COUNT, *, +), és az eredménytábla oszlopainak elnevezésére alias (SELECT table AS t) neveket adhatunk meg. Vannak függvények (pl.: ABS, TO_CHAR, TO_DATE ...) és összesítő függvények is (pl.: AVG, SUM, MAX, MIN, COUNT).

SELECT AVG(fizetés) FROM Dolgozó

Csoportosítás (GROUP BY, HAVING): Ha a tábla sorait csoportonként szeretnénk összesíteni, akkor a SELECT utasítás a **GROUP BY** oszloplista alparancsal bővíteni. Egy csoportba azok a sorok tartoznak, melyeknél oszloplista értéke azonos. Az eredménytáblában egy csoportból egy rekord lesz. Az összesítő függvények csoportonként hajtódnak végre.

SELECT osztkód, AVG(fizetés) FROM Dolgozó GROUP BY osztkód;

Csoportosítási szabály: A SELECT után összesítő függvényen kívül csak olyan oszlopnév tüntethető fel, amely a GROUP BY-ban is szerepel.

HAVING: segítségével A GROUP BY által képezett csoportok közül válogathatunk, hogy a talált értékek közül csak a megadott feltételnek eleget tevő sorok kerüljenek összesítésre.

SELECT osztkód, AVG(fizetés)

FROM Dolgozó

GROUP BY osztkód HAVING AVG(fizetés) > 180000

Eredménytábla rendezése: ORDER BY oszlopnév [DESC], ..., oszlopnév [DESC]: ha valamilyen rendezettségen kívánjuk látni az eredményt. A SELECT utasítás végére helyezhető, és az eredménytáblának a megadott oszlopok szerinti rendezését írja elő.

SELECT utasítás általános alakja:

SELECT [DISTINCT] oszloplista	projekció
FROM táblanévlista	Descartes-szorzat
[JOIN táblanév ON t.oszlop = t1.oszlop]	értékkészlet esetleges módosítása
[WHERE feltétel]	szelekció
[GROUP BY oszloplista	csoportonként összevonás
[HAVING feltétel]]	csoport-szelekció
[ORDER BY oszloplista];	rendezés

Ha egy **SELECT** utasítás **WHERE** vagy **HAVING** feltételében olyan logikai kifejezés szerepel, amely SELECT utasítást tartalmaz, ezt **alkérdésnek** vagy belső SELECT-nek is nevezik.

SELECT név, fizetés FROM Dolgozó WHERE fizetés < (SELECT AVG(fizetés) FROM dolgozó)

Nézettáblák (virtuális tábla, view) nem tárolnak adatokat. Egy transzformációs formula, ami segítségével a tárolt táblák egy meghatározott lekérdezésre adott eredményét tudjuk megkapni.

CREATE VIEW táblanév [(oszloplista)] AS alkérdés;

Nézettáblák alkalmazási lehetőségei:

- Származtatott adattáblák létrehozása, amelyek a törzsadatok módosításakor automatikusan módosulnak (pl. összegzőtáblák).
- Bizonyos adatok elrejtése egyes felhasználók elől (adatbiztonság vagy egyszerűsítés céljából).

Az SQL lehetőségeivel nem oldható meg minden adatbázis kezelési feladat. SQL-ben például nem használhatók változók és vezérlési szerkezetek, így az adatbázis algoritmikus kezelése sem lehetséges. Ezért az SQL utasításokat általában egy hagyományos algoritmikus programnyelv (C, Java, stb.) utasításaival keverten használjuk, és az SQL utasításokban felhasználhatók a befogadó programnyelv változói is. Ezt a megoldást nevezzük beágyazott (embedded) SQL-nek.

ODBC (Open DataBase Connection): normál C nyelvű programot írhatunk, amelynél egy függvénykönyvtár segítségével érjük el az adatbázist, alapvetően SQL utasításokat küldhetünk a DBMS-nek. Ezzel bizonyos rendszerfüggetlenség érhető el. A program elején includeolni kell header fájlokat, mely hatására az alábbi ODBC függvények, típusok, konstansok használhatók:

- SQLDriverConnectdbc, ...);
- SQLPreparestmt, "SELECT cikkszám, egységár FROM Raktár", SQL_NTS);
- SQLExecutestmt);

JDBC (Java DataBase Connection): Az ODBC-hez hasonló, de a Java objektumorientált jellegének felel meg.

- DriverManager.getConnection(url, user, password)
- CreateStatementSQL lekérdezés vagy utasítás létrehozása)
- executeQuerySQL lekérdezés vagy utasítás végrehajtása)

Összegzés

Digitális képfeldolgozás - 1.)

Simítás/szűrés képtérben (átlagoló szűrők, Gauss simítás és mediánszűrés); élek detektálása (gradiens-operátorokkal és Marr-Hildreth módszerrel).

Simítás/szűrés képtérben

Zaj: a képpont-intenzitások nemkívánatos változása, símítás és szűrés során ezeket szeretnénk valahogy javítani, csökkenteni. Több típusa létezik, mint például a Gauss-zaj (az intenzitásváltozás normál eloszlást követ), vagy só-bors zaj (fehér és fekete pontok kvázi random előfordulása).

Átlagoló szűrők: adott területen az intenzitások átlagát számoljuk, és mindegyiknek azt adjuk értékül. Ehhez használhatunk több féle maszkot, ami lehet 3x3-as, 5x5-ös négyzet, vagy 1.5, 2-es sugarú kör. A lényeg az, hogy kicsit normalizálni próbáljuk a képpontok intenzitása által meghatározott függvényt, kicsit tompítani, hogy ne legyenek akkora kiugrások.



eredeti kép

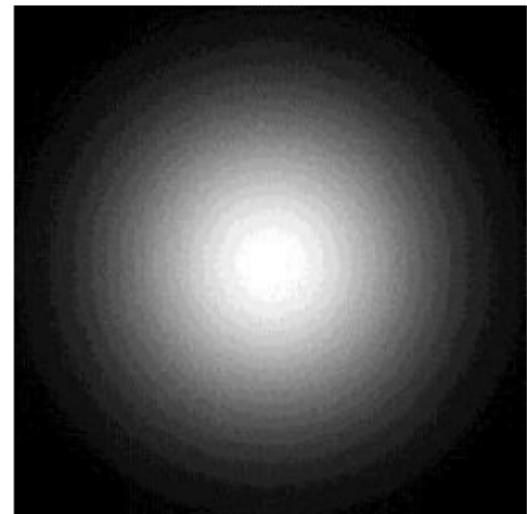
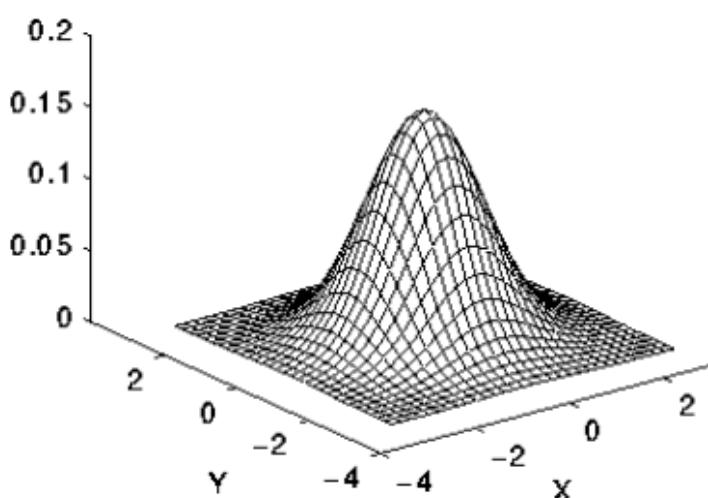
átlagolás
5x5-ös maszkkal

átlagolás
11x11-es maszkkal

Elmondhatjuk tehát, hogy az átlag-szűrő hatására a képpontok felveszik a környezetük átlagát, így a szűrt kép intenzitásértékei továbbra is a kiindulási kép intenzitástartományában maradnak. A zaj csökkentésére megfelelő eszköz lehet, viszont gyengíti az éleket, homályossá teszi a képet.

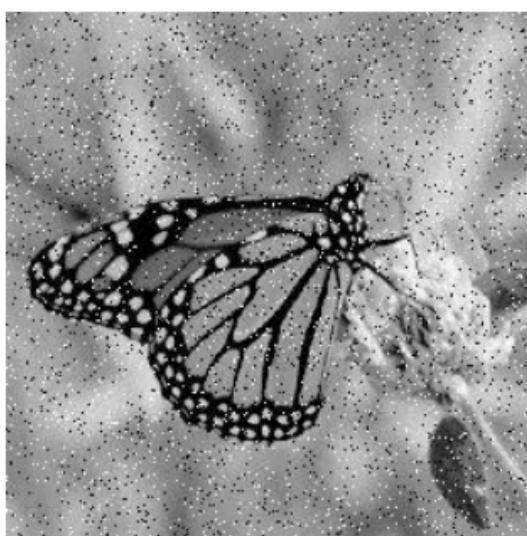
Használhatunk az átlagoláshoz súlyokat is, amik általában a képponttól mért távolsággal arányosan csökkennek.

Gauss simítás: Gauss-simítás során valamelyen Gauss-görbét követő intenzitású maszkkal simítjuk a képet. Egy ilyen görbe így néz ki 2D-ben:



Medián: Az $[a_1, a_2, \dots, a_{2n+1}]$ (páratlan elemszámú) számtömb mediánja a nagyság szerint rendezett tömb középső, ($n+1$ -edik) eleme. Jelölése $\text{med}\{a_1, a_2, \dots, a_{2n+1}\}$. Montható úgy is, mint egy középérték.

Mediánszűrés: A mediánszűrést úgy kapjuk, hogy fogjuk az adott képpont környezetét, azoknak intenzitását sorbarendezzük, majd a képpont új intenzitása a medián érték lesz. Ez ideális lehet só-bors zaj szűrésére.



só-bors zajos



medián szűrt

Segítségével meg tudjuk szüntetni a kis kiterjedésű kiugrásokat. Ez jobban megőrzi az éleket, mint az átlagolás, nagy kiterjedésű zajfoltoknál fel-elenyomó.

Élek detektálása

Pontok detektálása során a lokális környezetben lévő képpontokétől eltérő intenzitású pontokat szeretnénk észrevenni. A vonalak lényegében összefüggő pontok, detektálásuk során az egységnyi vastagságú görbéket szeretnénk kijelölni.

Élekről akkor beszélünk ha a képfüggvény valamely irány mentén hirtelen változik. Ideális esetben az élek lépcsősek, de sajnos a gyakorlatban inkább lejtős élekkel tudunk találkozni.

A **gradiens** a többváltozós függvények deriválásának általánosítása. Ennek egy vektormező az eredménye, ami azt mutatja meg, hogy hogyan változik a függvény (merre növekszik, mennyire növekszik, stb).

Ha az intenzitás profilt nézzük, akkor az **elsőrendű derivált** segítségével **egy lejtős intenzitásprofil** alapján **éleket** tudunk **csinálni**.

Roberts-operator zajérzékeny, viszont könnyen számítható.

Gradiens-operátorokkal

Marr-Hildreth módszerrel

Összegzés

Digitális képfeldolgozás - 2.)

Alakreprezentáció, határ- és régió-alapú alakleíró jellemzők, Fourier leírás.

Alakreprezentáció

Az alakzat pontok olyan összefüggő rendszere. A való életben számos példát ismerünk rájuk, a kérdés itt az, hogy a számítógép hogyan reprezentálja, vagy ismeri fel ezeket. Az alakreprezentáció a gépi látás általános modelljének harmadik, a szegmentálást megelőző lépése.

Az alakreprezentációt több módszer szerint is végezhetjük:

- az objektumot körülvevő **határ** leírásával
- az objektum által elfoglalt **régió** leírásával
- **transzformációs** megközelítésekkel

A **határvonal alapú alakreprezentáció** fő tulajdonságai:

Fourier leírás



Összegzés

Programozás alapjai - 1.)

Algoritmusok vezérlési szerkezetei és megvalósításuk C programozási nyelven. A szekvenciális, iterációs, elágazásos, és az eljárás vezérlés.

Algoritmusok vezérlési szerkezetei

Az algoritmusnak, mint műveletnek a legfontosabb része az, hogy megadjuk annak vezérlését, tehát az előírást, amely az algoritmus minden lépéssére kijelöli, hogy a lépés végrehajtása során melyik lépés végrehajtása következik. Ezeket leírhatjuk többféle módon:

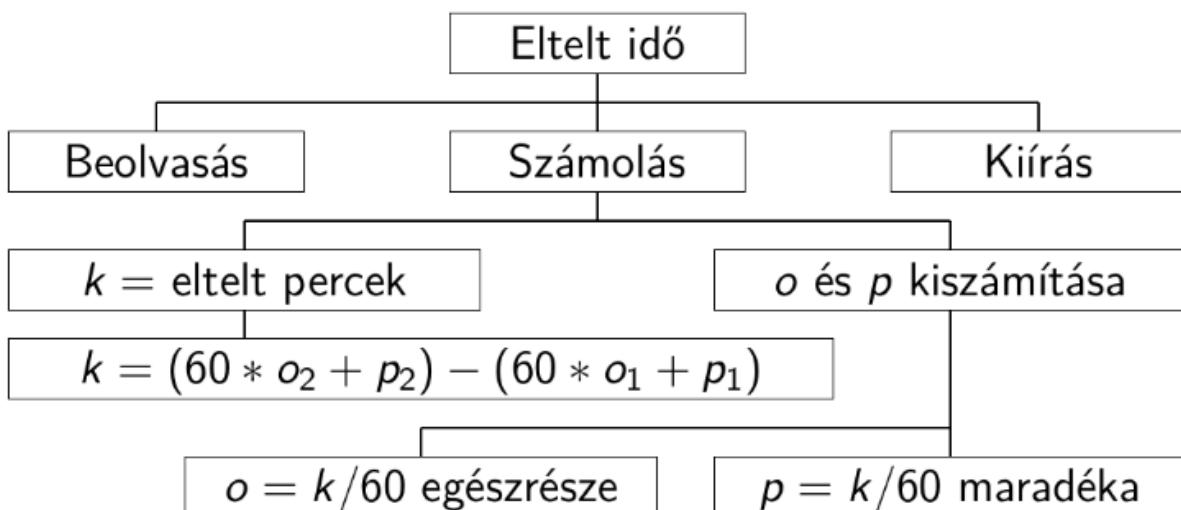
- **Természetes nyelvi leírás:** szövegesen, mondatokba foglalva írja le az algoritmust (sejthető, hogy ez azért jó távol állhat a megvalósítástól)
- **Pszeudokód:** egy programozási nyelvhez hasonló strukturált nyelv, csak szabadabb
- **Folyamatábra:** grafikus, kevésbé strukturált gráf a végrehajtásra, a folyamatra koncentrál
- **Szerkezeti ábra:** grafikus, strukturált leírása az algoritmus felépítésének, amely leírja a működési folyamatot is.

Szekvenciális vezérlés: 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 végrehajtjuk.

Példa - eltelt idő kiszámítása

Probléma: adott két időpont (órában és percben megadva), keressük a kettő között a különbséget percben (is).

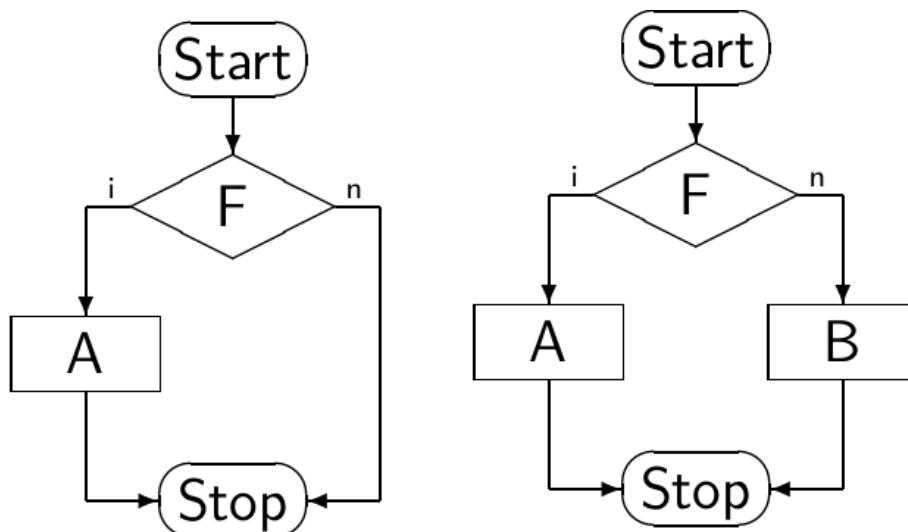
Megoldás: Vesszük a megfelelő megszorítások közé eső paramétereket (értelem szerűen az óra ne legyen már nagyobb, mint 24, sem kisebb, mint 1 - a percekre is ugyan ez essen 1 és 60 közé, valamint lehetőség szerint minden a második időpont következzen az első után), kiszámoljuk a két paraméter különbségét, majd kiírjuk. Szerkezeti ábra, és C megvalósítás a következő oldalon.



Szelekciós vezérlés (elágazásos): véges sok rögzített művelet közül véges sok feltétel alapján választjuk ki, hogy melyik művelet kerüljön végrehajtásra. Attól függően, hogy a kiválasztás milyen módszerrel történik meg, különböző altípusai vannak ennek a vezérlésnek.

Szelekciós vezérlés típusai:

- egyszerű szelekciós vezérlés (if - then - else)
- többszörös szelekciós vezérlés (if - then - else if - then - else)
- esetkiválasztásos szelekció (switch - case)
- az említettek kiegészítése egyéb ágakkal

**Példa - háromszögek osztályozása**

Probléma: adottak a háromszög oldalai, döntsük el róla, hogy az milyen háromszög.

Megoldás: le kell ellenőriznünk, hogy pozitív hosszúak-e a kapott élek, valamint a hosszabb oldal nagyobb, vagy egyenlő kell, hogy legyen a másik két oldal összegénél, előfordulhat, hogy minden oldal egyenlő hosszúságú, valamint pitagorasz tételere is rálesünk, hogy derékszögű-e a háromszög.

Eljárásvezérlés: egy műveletet adott argumentumokra alkalmazunk, aminek hatására az argumentumok értékei pontosan meghatározott módon változnak meg.

Két fajtája van: az eljárásművelet és a függvényművelet. Ez a két művelet bizonyos programozási nyelvek esetén teljesen meg vannak különböztetve, bizonyos nyelvek esetén szinte nincs különbség a kettő között. C nyelv esetében minimális a különbség a kettő között.

Függvényművelet

A C-ben használt függvényeket tekinthetjük a matematikai függvények általánosításának. Ha egy részprobléma célja egy érték kiszámítása adott értékek függvényében, akkor a megoldást megfogalmazhatjuk függvényművelettel. (FÜGGVÉNY = adott paraméterekből számoljunk ki valamit)

A függvényművelet specifikációja tartalmazza:

- a művelet elnevezését,
- a paraméterek felsorolását,
- a paraméterek típusát,
- a műveletek hatásának leírását,
- a függvényművelet eredménytípusát

Eljárásművelet

Alkalmazása adott argumentumokra az argumentumok értékének pontosan meghatározott megváltozását eredményezi. minden eljárásműveletnek rögzített számú paramétere van, és minden paraméter rögzített adattípusú. **ELJÁRÁS** = parancsok csoportja, aminek hatására változhatnak a paraméterek

Amikor meghívódik egy függvény, az átadott változók, mint aktuális paraméterek kerülnek felhasználásra: értékei bemásolódnak (jobbról balra) a verembe, ahol a függvény formális paramétere által hivatkozhatóakká válnak.

A függvény a végrehajtása során memóriát allokál a saját lokális változói számára a veremben. A függvény végrehajtása addig tart, amíg el nem jutunk egy return utasításig. Ekkor a függvényhívás kifejezés felveszi a függvényben kiszámolt értéket, és a vezérlés végrehajtása visszakerül a hívó függvényhez. Ezután a hívó gondoskodik a veremmutató visszaállításáról, hogy oda mutasson, ahova a függvényhívás és a paraméterek bepakolása előtt mutatott.

Függvények mellékhatásán azt értjük, hogy a függvényhívás hatására nem csak a függvényérték számítódik ki (és a paraméterek változhatnak meg), hanem megváltozhat egy globális változó értéke is (vagy egyéb műveletek is végrehajtódnak, pl. kiíratás)

Előnyök:

- többszörös felhasználás: részproblémák megoldására elég egy függvényt készíteni
- memóriaigény csökkentése: a lokális változók csak a függvény végrehajtásáig vannak a memóriában
- karbantarthatóság: áttekinthetőbb kód
- modularitás: bizonyos eszmei egységek jól elkülöníthetők
- programhelyesség: könnyebb bizonyítás, tesztelés, hibakeresés, javítás

Ismétléses (iterációs) vezérlés: egy adott feltétel szerint egy műveletsort végrehajtását többször megismételjük. Fajtái:

- kezdőfeltételes (while),
- végfeltételes (do-while),
- számlálásos (fori),
- hurok (while, bizonyos feltételek beteljesülése során kilépünk belőle),
- diszkrét (foreach)

Kezdőfeltételes: a ciklusmag (ismételt) végrehajtását egy belépési feltételhez kötjük. Ameddig a feltétel teljesül, újra és újra végrehajtjuk, amint beteljesül, kilépünk a ciklusból, és hello.

Végfeltételes: a ciklusmag végrehajtását egy kilépési feltételhez kötjük. Ha a ciklusmag végrehajtásának végén a feltétel teljesül, továbbadjuk a vezérlést.

Számlálásos: a ciklusmagot végre kell hajtani egy változó sorban minden olyan értékére (növekvő vagy csökkenő sorrendben), amely egy adott intervallumba esik.

Hurok: az ismétlését a ciklusmagon belül vezéreljük úgy, hogy különböző pontokon adott feltételek teljesülése esetén a ciklus végrehajtását befejezzük.

Diszkrét: a ciklusmagot végre kell hajtani egy halmaz minden elemére tetszőleges sorrendben.

Összegzés

Vezérlési szerkezetek

- a műveleteknek fontos része a vezérlés megadása (előírás, amely az algoritmus minden lépéssére kijelöli, hogy a lépés végrehajtása során melyik lépés végrehajtása következik)
- SZEKVENCIÁLIS VEZÉRLÉS
 - sorban értékelődik ki minden p_i utasítás, egymás után
 - ez nagyjából olyan, mint amikor egymás után írunk páros parancsot a kódban
- ITERÁCIÓS (ISMÉTLÉSES) VEZÉRLÉS
 - a ciklusmag egy bizonyos feltétel alapján többször, ismételten végrehajtódik
 - kezdőfeltéles (while), végfeltéles (do-while), számlálásos (fori), hurok (while-if), diszkrét (foreach)
- ELÁGAZÁSOS VEZÉRLÉS
 - a vezérlés egy bizonyos feltétel teljesülése esetén tovább halad, vagy bizonyos részeket kihagy
 - egyszerű szelekciós vezérlés (if-then), többszörös szelekciós vezérlés (if-then-else), esetkiválasztós vezérlés (switch-case), és ezek kombinációja
- ELJÁRÁSVEZÉRLÉS
 - egy műveletet adott argumentumokra alkalmazunk, aminek hatására azok értékei pontosan meghatározott módon változnak meg
 - a függvények specifikációja a következő elemeket tartalmazza:
 - a művelet elnevezését,
 - a paraméterek felsorolását,
 - a paraméterek típusát,
 - a műveletek hatásának leírását,
 - a függvényművelet eredménytípusát
 - FÜGGVÉNYMŰVELET
 - a C függvények a matematikai függvények általánosításai
 - adott paraméterekből számoljunk ki valamit
 - Függvények mellékhatása: a függvényhívás hatására nem csak a függvényérték számítódik ki, hanem pl megváltozhat egy globális változó értéke is
 - ELJÁRÁSMŰVELET
 - rögzített számú paraméter, és minden paraméter rögzített adattípusú
 - parancsok csoportja, amik operálnak a paraméterekkel
 - Függvények előnyei:
 - újrafelhasználás
 - memóriaigény csökkentése
 - karbantarthatóság
 - modularitás
 - programhelyesség

Programozás alapjai - 2.)

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.

Az adattípus olyan egysége a programnak, amely annak értékhalmaza és az értékhalmazán végezhető műveletek alapján pontosan meghatározható. minden adattípus vagy elemi, vagy más adattípusokból képzett összetett adattípus.

Elemi adattípusok

A felsorolt adattípusokon kívül még a felsorolás (enum) is elemi adattípusnak számít. Ezek azért elemiek, mert nem lehet önmagukban értelmes részekre bontani. Az elemi adattípusok statikusak (nem úgy, mint a static kulcsszóval), létezésük a létrehozás blokkjához kötött.

KARAKTER

Típus	Értékkészlet	Memória
char, signed char	[-128, 127]	8 bit
unsigned char	[0, 255]	8 bit

Pontosság: túl-, és alulcsordulás előfordulhat, ha a kereten kívül esik a reprezentált érték

Műveletek: feltételezhetően ugyanaz, mint az egészek esetében, bitenkénti ÉS-eléssel és VAGY-olással kibővíve.

A gép legkisebb címzethető egysége, egy karakter (betű, szám, írásjel) ábrázolására való. Ábrázolás ugyanaz, mint a 8 bites egészeknél. Értékadás karakterkóddal számként, vagy aposztrófok közé írt karakterrel.

LOGIKAI

Típus	Értékkészlet	Memória
bool	[true, false]	8 bit

Pontosság: elég pontos, ilyen bináris értéket érezhetően nem könnyű elrontani

Műveletek:

- negálás (!)
- egyenlő (==)
- nem egyenlő (!=)
- logikai éselés (&&)
- logikai vagyolás (||)

Eredetileg a C nyelvben nem definiált adattípus, a C⁹⁹-es szabvánnyal került be, használatához szükséges az stdbool.h header file-t behúzni.

EGÉSZ

Típus	Értékkészlet	Memória
short, short int, signed short, signed short int	[-32.767, 32.767]	16 bit
unsigned short, unsigned short ing	[0, 65.535]	16 bit
int, signed, signed int	[-32.767, 32.767]	16 bit
unsigned, unsigned int	[0, 65.535]	16 bit
long, long int, signed long, signed long int	[-2 milliárd, 2 milliárd]	32 bit
unsigned long, unsigned long int	[0, 4 milliárd]	32 bit
long long, long long int, signed long long, signed long long int	[-sok, sok]	64 bit
unsigned long long, unsigned long long int	[0, rengeteg]	64 bit

Pontosság: túl-, és alulcsordulás előfordulhat, ha a kereten kívül esik a reprezentált érték

Műveletek:

- előjelváltás (-int)
- összeadás (int+int)
- kivonás(int-int)
- szorzás (int*int)
- egész osztás (int/int)
- maradékképzés (int%int)
- egyenlő (==)
- nem egyenlő (!=)
- kisebb (<), kisebb vagy egyenlő (<=)
- nagyobb (>), nagyobb vagy egyenlő (>=)

C programokon belül a szélsőértékek eléréséhez be kell húzni a limits.h header file-t. Látható, hogy unsigned típusok esetén ha az értelmezési tartomány negatív részét (tehát a 0-nál kisebb számokat) levágjuk belőle, és a pozitív értékeket kitoljuk annyival, mint amennyit így szerezünk. Ez azért van így, mert az előjeles számok esetében az első bitet elhasználjuk az előjel tárolására, és a többivel fejezzük ki magát a számot, tehát ha például van egy 8 biten tárolt számunk, azzal alapesetben maximálisan 2^8 nagyságú számot tudunk tárolni (példa: unsigned char, [0, 255]), viszont ha az első biten tároljuk a szám negativitását, akkor már csak 2^7 bitünk marad a számra.

VALÓS

Típus	Értékkészlet	Memória
float (egyszeres pontosság)	1 előjelbit, 8 bit egész érték, 23 bit tört	32 bit
double (kettős pontosság)	1 előjelbit, 11 bit egész érték, 52 bit tört	64 bit
long double	1 előjelbit, 15 bit egész érték, 112 bit tört	128 bit

Műveletek: ugyanaz, mint az egészknél, figyelni kell a logikai ==, != műveletekre, mert float és double összehasonlításánál a pontosság miatt lehet eltérés! Ezen felül használható még math.h header több metódusa is (mint például a cos, sin, log, exp)

Összetett adattípusok

POINTER

Az elemi adattípusoknál szó esett a statikus változókról, nézzük meg mi történik abban az esetben, ha egy változó nem esik ebbe a kategóriába! Ha egy változó nem függ az őt tartalmazó blokk aktivitásától, **dinamikus változóról** beszélünk. Egy pointer típusú változó értéke egy meghatározott típusú dinamikus változó. C-ben a pointer típusú változókat * segítségével tudunk létrehozni, például:

```
int *p
```

A fenti sorról az alábbiak állapíthatóak meg:

- A *p egy int típusú (dinamikus) változó
- A p egy int* típusú (tehát egy int-re mutató) változó

Ennek tudatában nézzük meg a következő sort!

```
int *p, q
```

Ebben az esetben csak ***p egy int-re mutató változó lesz, míg q egy int értéket tárol!** Ezért szokás a csillagot inkább a változó neve mellé tenni, és nem a típus mellé. **Változóhivatkozásokor** meghatározott formai szabályok szerint képzett jelsorozat segítségével elérhetjük a pointer által mutatott dinamikus változó értékét. Ebben az esetben a dinamikus változót **hivatkozott változónak** hívjuk, a visszakapott értéket pedig a **változó értékének**.

Műveletek:

- **NULL**: amikor a pointerhez nem tartozik dinamikus változó
- **Létesít (malloc)**: új, dinamikus változó létesítése
- **Törlés (free)**: dinamikus változó törlése, a lefoglal memória felszabadítása
- **Dereferencia (*)**: lehivatkozzuk a pointer által mutatott dinamikus változót, így érjük el annak értékét - **magas precedencia**
- **(Nem)Egyenlő (==, !=)**: két pointer értéke megegyezik-e
- **Cím lekérése (&)**: megmondja, hogy p melyik már létrehozott változóra mutasson - **magas precedencia**

Használatukhoz szükséges az stdlib.h vagy memory.h header fileok egyikét behúzni a kódba. A malloc(S) (memory allocation) lefoglal egy S méretű memóriaterületet. Ezt gyakran a sizeOf(E) metódussal párból használjuk, ami visszaadja, hogy az E típus mekkora helyet foglal a memoriában, tehát kombinálva kapunk egy malloc(sizeOf(E)) utasítást, ami létrehoz egy új E típusú érték tárolására (is) alkalmas változót, amit p értéke lesz. Linuxos rendszerekben a pointer a hozzá tartozó dinamikus változó kezdőcímét tartalmazza, tehát felfogható egy tétszőleges memóriacímként is.

```
int i, j, *p;
p = &i;           /* p i-re mutat */
j = *p;           /* hatása ua., mint j = i; */
*p = 2;           /* hatása ua., mint i = 2; */
j = *p + 1;       /* hatása ua., mint j = i + 1; */
*p += 1;          /* hatása ua., mint i += 1; */
++*p;             /* hatása ua., mint ++i; */
(*p)++;           /* hatása ua., mint i++; */
```

TÖMB

Algoritmusok tervezésekor gyakran előfordul, hogy adatok sorozatával kell dolgozni, vagy mert az input adatok sorozatot alkotnak, vagy mert a feladat megoldásához ez kell. Tegyük fel, hogy a sorozat rögzített elemszámú (n) és minden egyik komponensük egy megadott (elemi vagy összetett) típusú 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.
- Deklarálás: típus változónév[elemszám];
- C nyelvben a tömb indexelése minden esetben 0-val kezdődik

Műveletek

- **Kiolvas**($\rightarrow A:T; \rightarrow i:I; \leftarrow x:E$): Adott $i \in I$ -re az A sorozat i . komponensének kiolvasása adott x, E típusú változóba, pl.: $x = A[i]$
- **Módosít**($\leftarrow A:T; \rightarrow i:I; \rightarrow x:E$): Adott $i \in I$ -re az A sorozat i . komponensének módosítása adott x, E típusú értékre, pl.: $A[i] = x$
- **Értékkopírozás**($\leftarrow A:T; \rightarrow X:T$) Az A változó felveszi az X, T típusú kifejezés értékét, pl.: $A = X$ nem alkalmazható, helyette: `memcpy(A, X, sizeof(T)); memcpy_s(A, sizeof(A), X, sizeof(T))`;

A Kiolvas és a Módosít műveletek megvalósítása a tömbelem-hivatkozással történik. A 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.

Tárolás

Tömb típusú változó számára történő helyfoglalás azt jelenti, hogy minden tömbelem, mint változó számára memóriát kell foglalni. Feltehetjük, hogy egy adott tömb változóhoz a tömbelemek számára foglalt tárterület összefüggő mezőt alkot. A tömbök használata nagy körültekintést igényel, mert a program végrehajtása közben nincs indexhatár-ellenőrzés (tehát simán hivatkozhatunk olyan indexre is véletlenül, ami nem a tömb által lefoglalt területen belül esik).

Egy függvény paramétere lehet tömb típusú is. Ilyen esetben azonban nem történik teljes értékmásolás, hanem csak a tömb címe (vagyis egy pointer) kerül átadásra, így a tömb elemein végzett bármely módosítás az eredeti tömbön kerül végrehajtásra.

A pointerek és a tömbök kapcsolata

A C nyelvben szoros kapcsolat van a mutatók és a tömbök között: valamennyi művelet, amely tömbindexeléssel végrehajtható, mutatók használatával éppúgy elvégezhető. Gyakorlatilag a tömbre való hivatkozást a fordító a tömb kezdetét megcímző mutatóvá alakítja át.

Ha p_a egy tömb adott elemére mutat, akkor definíció szerint p_{a+1} a tömb következő elemére fog mutatni, így ha p_a az $a[0]$ -ra mutat ($*p_a = &a[0]$), akkor $*(p_{a+1})$ a $a[1]$ tartalmát szolgáltatja, p_{a+i} az $a[i]$ elem címe, és $*(p_{a+i})$ az $a[i]$ elem értéke (Röviden: bármilyen tömb vagy indexkifejezés leírható, mint egy mutató plusz egy eltolás).

REKORD

Gyakran előfordul, hogy különböző típusú, de logikailag összefüggő adatelemekkel kell dolgozni: az ilyen adatok tárolására szolgál a **szorzat-rekord** típus. Tegyük fel például, hogy el kell tároljuk egy könyv szerzőjét, és címét is, ez legyen két string. A típusok értékkészletének direkt szorzata képezi az új rekord típust, amit **struktúrának** is hívunk.

Műveletek:

- **Kiolvas** (\rightarrow): visszaadja az adott komponens értékét
- **Módosít** (\leftrightarrow): módosítja az adott komponens értékét
- **Értékkadás** (\leftarrow): értéket ad az adott komponenst értékét

A képen látható kódrészlethez ha csinálunk egy SzemelyTip x változót, teljesen valid hivatkozás lenne pl az x.nev.csaladi vagy az x.szulido.ev. A struktúrák lefoglalt memóriaterület a bennük tárolt összes komponens területének **összegével** egyenlő.

UNIÓ

Sokban hasonlít a struktúrához (ezért hívjuk őket **egyesített-rekordoknak** is), a kiolvas és a módosít műveleteket hasonlóan is értelmezzük rajtuk. minden értelmezett műveletet szintén mezőhivatkozások (. operátor) segítségével valósítjuk meg, mint a struct esetében, viszont itt bármikor hivatkozhatunk bármelyik mezőre, függetlenül attól, hogy az unió éppen melyik mező értékét tárolja. Legfontosabb különbség közük talán az, hogy az unió az általa felvehető értékek közül a legnagyobbnak megfelelő méretet foglal le a memoriában, és ezek közül csak egyet vehet fel (kicsit olyan, mint az enum, csak na: értük a különbséget).

```
typedef struct DatumTip {
    short ev;
    char ho;
    char nap;
} DatumTip;

typedef char Szoveg20[21];

typedef struct CimTip {
    Szoveg20 varos, utca;
    short hazszam;
    short iranyitoSz;
} CimTip;

typedef struct SzemelyTip {
    struct {
        Szoveg20 csaladi, uto;
    } nev;
    int szemelyiSzam;
    Szoveg20 szulHely;
    DatumTip szulido;
    CimTip lakcim;
} SzemelyTip;
```

```
typedef enum { kor, haromszog, negyszog } Sikidom;

typedef union Szam {
    double Valos;
    long int Egesz;
} Szam;

typedef struct Idom {
    Sikidom Fajta;
    union {
        double Sugar;
        struct { double A, B, C; } U1;
        struct { double D1, D2, D3, D4; } U2;
    } UU;
} Idom;

typedef struct Alakzat {
    double x, y;
    Sikidom forma;
    union {
        double sugar;
        struct { double alfa, oldali, oldal2; };
        struct { double hossz, szel; };
    };
} Alakzat;
```

Ahogy az a képen is látható, az uniók és a rekordok tetszőlegesen egymásba ágyazhatók. Itt például az Idom struktúrának megadunk egy Sikidom típusú változót: ez az elején létrehozott enumerációs típus. Ezen felül látható még az is, hogy az unióban eltároljuk az aktuális sikidom specifikus tulajdonságait: kör esetén a sugarát, háromszög vagy négyzet esetén az oldalait.

Habár itt a példában nem látszott, megengedett még a & művelet használata is, így a struct és a union típusú változók cím szerinti paraméterátadással is kezelhetők.

Összegzés

Egyszerű adattípusok

- az egyszerű adattípusok a hozzájuk szükséges tárhely által maximálisan reprezentálható számmal egyenlő értéket tudnak mutatni (tehát egy 8 bites szám 2^8 számú értéket tárolhat: ez signed esetén $[-2^7, 2^7]$, mivel kell 1 bit az előjelnek, unsigned esetén pedig $[0, 2^8]$).
- KARAKTER
 - 8 bit (1 byte)
 - egy számot, vagy aposztrófok között megadott betű/szám/írásjel számkódját tárolja (C esetében ASCII)
 - lehet signed és unsigned és signed is-
- LOGIKAI
 - 8 bit (1 byte)
 - logikai értékeket tárol (igaz / hamis)
 - értékkészlete [true, false]
- EGÉSZ
 - 16 bit (2 byte) / 32 bit (4 byte) / 64 bit (8 byte)
 - (signed) short, int / (signed) long / (signed) long long
 - sok művelet értelmezett rajtuk, körülbelül az összes aritmetikai, és logikai műveletek
 - túl-, és alulcsordulás, ha az értéken kívül megyünk
 - limits.h header segítségével kapunk olyan menő konstansokat, mint pl INT_MIN
- VALÓS
 - float (egyszeres pontosság), 32 biten (1 előjel bit, 8 bit karakterisztika, 23 bit mantissa)
 - double (kétszeres pontosság), 64 biten (1 előjel bit, 11 bit karakterisztika, 52 bit mantissa)
 - long double (nagy double), 128 biten (1 előjel bit, 15 bit karakterisztika, 112 bit mantissa)

Összetett adattípusok

- TÖMB
 - ha sorozat rögzített elemszámú, és minden egyik elem egy megadott típusú érték, érdemes tömböt használni a megvalósításhoz
 - Kiolas, Módosít, Értékkadás
 - C-ben A = X nem alkalmazható, helyette: memcpy(A, X, sizeof(T))
- POINTER
 - **dinamikus változó:** a változó nem függ az őt tartalmazó blokk aktivitásától
 - egy pointer típusú változó értéke egy meghatározott típusú dinamikus változó
 - NULL, Létesít, Töröl, Dereferencia, (Nem)Egyenlő, Cím lekérése
 - szükséges az stdlib.h vagy memory.h header fileok egyike
 - E *p = malloc(sizeof(E)) létrehoz egy E-re mutató pointert, és lefoglal neki a memoriában egy E tárolására elegendő memóriaméretet
- REKORD
 - különböző típusú, de logikailag összefüggő adatalemekkel kell dolgozni
 - szorzat-rekord: a memoriában lefoglalt terület a mezői számára szükséges memória összege
 - Kiolas, Módosít, Értékkadás
 - C-ben typedef struct A {} A; paranccsal hozzuk létre

- UNIÓ

- ha egy adattag több közül egy értéket vehet fel,
- egyesített rekord: a memóriában lefoglalt terület egyenlő lesz az általa legnagyobb területet igénylő mező méretével
- Kiolvas, Módosít, Értékkadás
- C-ben typedef union A {} A; paranccsal hozzuk létre

Programozás I, II - 1.)

Objektum orientált paradigmája és annak megvalósítása a JAVA és C++ nyelvekben. Az absztrakt adattípus, az osztály. Az egységbázis, az információ elrejtése, az öröklődés, az újrafelhasználás és a polimorfizmus. A polimorfizmus feloldásának módszere.

Objektum orientált paradigmája

Az objektumorientált (**object-oriented programming, röviden OOP**) paradigmája az objektumok fogalmán alapuló programozási elv. Az objektumok logikai egységbe foglalják az adatokat és a hozzájuk tartozó műveleteket. Az adatokat ismerjük mezők, attribútumok, tulajdonságok néven, a műveleteket metódusokként szokták emlegetni. Az objektum által tartalmazott adatokon általában az objektum metódusai végeznek műveletet. Egy ilyen program egymással kommunikáló objektumok összességéből áll. A legtöbb objektumorientált nyelv osztály alapú, azaz az objektumok osztályok példányai, és típusuk az osztály.

Például egy hétköznapi fogalom, a "kutya" felfogható egy osztály (a kutyák osztálya) tagjaként, annak egyik objektumaként. minden kutya objektum rendelkezik a kutyára jellemző tulajdonságokkal (például szín, méret stb) és cselekvési képességekkel (például futás, ugatás).

A legtöbb széles körben alkalmazott nyelv többek között az objektumorientált programozást is támogatja, tipikusan az imperatív, procedurális programozással együtt.

Az absztrakt adattípus, az osztály

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, amely az adattagjait és a rajta elvégezhető műveleteket zárja egy logikai 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: Javában és C++-ban is a **class** kulcsszóval tudunk osztályokat definiálni. Az osztályokból tetszőleges mennyiségben létrehozhatunk példányokat, azaz objektumokat.

Objektum: egy változó, melynek típusa valamely osztály (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: bekapcsolva true vagy false)
- **Viselkedés:** az objektum viselkedése annak leírása, hogy az objektum hogyan 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.

Az egységbezárás, információ elrejtése

A láthatóságok segítségével tudjuk szabályozni adattagok, metódusok elérését, ugyanis ezeket az objektumorientált paradigmában értelmében korlátozni kell, kívülről csak és kizárolag ellenőrzött módon lehessen ezeket elérni, használni. A láthatóság vezérléséhez az alábbi kulcsszavakat használhatjuk:

- **public**: minden honnan látható
- **protected**: csak az osztály scope-jában, és a leszármazottaiban látható
- **private**: csak az adott osztályban érhető el

Ezen felül Java-ban van még egy **package private** láthatóság is (ez az alapértelmezett). A nevéből következtethető, hogy ez a tartalmazó csomagon belül elérhető bárhonnan, azon kívül nem. C++-ban a láthatóság mindig **private**.

Érhető, hogy a programozás során nem feltétlenül szeretnénk az osztályok minden példányának minden adattagját kiszolgáltatni a programot futtató környezetnek (például nem feltétlenül szeretnénk egy felhasználó kódját is kiszolgáltatni egy weboldalon). Az egységbezárás (**encapsulation**) ezen eszközök segítségével segít elrejteni az objektumok változóinak értékét úgy, hogy az őket leíró osztályokban a mezőket **private** láthatóságra állítjuk. Ezeknek az értékét természetesen elérhetjük **getterek**, és módosíthatjuk **setterek** segítségével, amiket **public** láthatóságra teszünk. Ezeket természetesen tetszőlegesen specifikus működéssel is elláthatjuk (beállítandó érték ellenőrzése, annak esetleges módosítása).

Az öröklődés

Öröklődésnek nevezük az osztályok között értelmezett viszonyt, amely segítségével egy általánosabb típusból (ősosztály) egy sajátosabb típust tudunk létrehozni (utódosztály). Az utódosztály adatokat és műveleteket örököli, kiegészíti ezeket saját adatokkal és műveletekkel, illetve felülírhat bizonyos műveleteket. A kód újrafelhasználásának egyik módja. Megkülönböztetünk egyszeres és többszörös öröklőést. A hasonlóság kifejezése az ős felé az **általánosítás**, különbség a gyerek felé a **specializálás**.

Java esetében 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 ősosztályból származhat (viszont több interfész implementálhat) **super** segítségével gyerekosztályból hivatkozhatunk szülőosztály adattagjaira és megótudaira (super() meghívásával pedig a konstruktoraira).

C++ esetében az osztály neve után egy kettőspontot követően vesszővel elválasztva lehet megadni az ősosztályokat és velük együtt a láthatóságaikat. Ebből a megfogalmazásból már következtethető, hogy a C++-ban még megengedett a többszörös öröklődés.

Az öröklődés során lehetőség van az ősosztály tagjainak láthatóságán változtatni. Ezt az ősosztályok felsorolásakor kell definiálni. Az változtatás csak szigorítást (korlátozást) jelenthet, tehát egy **private** metódust nem lehetünk leszármaztatás során **public**-ká. Ez alól mondjuk kivételt képeznek Java-ban a konstruktork, ahol meg tudjuk adni, hogy a leszármaztatott osztály konstruktora az ősével ellentétben **private** helyett **public** legyen.

Többszörös öröklődés esetén előfordulhat olyan eset, amikor egy-egy ős osztály az öröklődési hierarchia különböző pontján ismét megjelenik (például gyémánt öröklődés). Ekkor a gyermek osztályban ennek az ős osztálynak több példánya jelenhet meg. Erre néhány esetben nincs szükség, például ha az ősosztály csak egy eljárás-erőforrás, akkor minden esetben elegendő egyetlen előfordulás a gyermek osztályokban. Ebben az esetben egy virtuális ősosztály hozunk létre, amit az öröklődésnél az ősosztályok felsorolásakor **virtual** módosítóval kell jelezni.

Újrafelhasználás és polimorfizmus

Az újrafelhasználhatóság az objektum orientált paradigmára egyik legfontosabb előnye. Az a jelenség, amikor egy változóra nem csak egyfajta típusú objektumként hivatkozhatunk. 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 típusuktól. Polimorfizmusról tehát akkor beszélünk, ha egy metódust egy ősosztály és egy gyermekosztály is tartalmaz (tehát felül lett írva), és a leszármazott egy példányán meghívjuk a metódust. Ekkor el kell döntenünk, hogy melyik osztály metódusát hajtsuk végre.

Polimorfizmusra két lehetőség van:

- **statikus polimorfizmus (korai hozzárendelés)** - a hívott metódus nevénk és címének összerendelése szerkesztéskor történik meg. A futtatható programban már fix metóduscímek találhatók. (staticus, private, final metódusok)
- **dinamikus polimorfizmus (késői hozzárendelés)** - metódus nevénk és címének hozzárendelése a hívás előtti sorban történik, futási időben

Egy virtuális eljárás címének meghatározása indirekt módon, futás közben történik (mondhatnánk úgy is, hogy késői hozzárendeléssel). Javaban 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 (VFT) tartja nyilván a virtuális eljárások címeit. A VFT táblázat öröklődik, feltöltéséről a konstruktur gondoskodik. A származtatott osztály konstruktora módosítja a virtuális függvénytáblát, kijavítja az ősosztályból örökölt metóduscímeket. Amikor a konstruálási folyamat véget ér, a 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 későbbiekbén (tehát ha egy metódusról tudjuk, hogy azt a leszármazott osztályokban felül szeretnénk írni, akkor az legyen virtual).

Javaban **absztrakt osztályok és interfészek (interface)** segítségével tudjuk a polimorfizmust megvalósítani. Az absztrakt osztályok különlegessége az, hogy azok tartalmazhatnak eljárásokat, amik szükség szerint lehetnek akár **absztraktok** is (ezekhez nem adunk megvalósítást, majd a leszármazott osztályokban megvalósítjuk) és adattagokat, viszont példányt belőlük sosem hozhatunk létre. Belőlük - akárcsak a többi osztályból - az **extends** kulcsszóval származtatunk le.

Az **interface**-ek olyan speciális osztályok, amelyek **metódusok deklarációját tartalmazzák csak** (a Java újabb verzióiban már léteznek **funkcionális interface**-k is, amelyek tartalmazhatnak **default** megvalósítást egyes metódusokra). Belőlük az **implements** kulcsszóval származtatunk le, és egy adott osztály tetszőlegesen sok interfész valósíthat meg. Öröklés során minden metódust **meg kell valósítanunk az összes interfésből**. Akárcsak az absztrakt osztályok esetében, az **interfészek sem példányosíthatók**.

A gyakorlatban a polimorfizmust akkor láthatjuk, ha például létrehozunk egy "I interfész típusú" X változót. Ez valójában azt fogja jelezni a programban, hogy X egy olyan tetszőleges objektum lesz, ami megvalósítja I-t. Ebben az esetben látható a többalakúság, ugyanis X-re egyszerre hivatkozhatunk úgy is, mint X, és úgy is, mint egy I-t megvalósító példány. Értelemszerűen ez öröklődésnél hasonlóan működik.

Ez egy hasznos eszköze az objektum orientált programozásnak, viszont **sokszor szeretnénk elkerülni (vagy feloldani) a többalakúságot**. Erre egy általános módszer a **kasztolás (type cast)**, ahol **meghatározzuk, hogy egy változóra milyen típusú objektumként fogunk a későbbiekbén hivatkozni**. Javaban rendelkezésünkre áll még az **instanceof** kulcsszó, ami megmondja, hogy az adott változó a meghatározott osztály példánya-e. Erre rákérdezhetünk úgy is, hogy az **X.getClass().equals(Object.class)** metódussal direkt rákérdezünk arra, hogy X az adott osztály típusú-e (primitívek esetén ott vannak a wrapper osztályok, mint például int → Integer).

A típuskasztoláson kívül **C++-ban** is vannak olyan metódusok, amivel meg tudjuk határozni egy változó osztályát. Az std névterben található egy **is_base_of** nevű metódus, ami meghatározza, hogy a második paraméterként átadott változó az első paraméterben található osztály egy példánya-e.

Összegzés

Az objektumorientált paradigmá

- az objektumorientált (object-oriented programming, röviden OOP) paradigmá az objektumok fogalmán alapuló programozási elv
- az objektumok logikai egysége foglalják az adatokat és a hozzájuk tartozó műveleteket.
- az adatokon általában az objektum metódusai végeznek műveletet
- egy OOP program egymással kommunikáló objektumok összességéből áll

Az absztrakt adattípus

- az adattípus leírásának legmagasabb szintje
- az adatok ábrázolására és a műveletek implementációjára semmilyen előírást nem adunk

Az osztály

- egy absztrakt adattípus, amely az adattagjait és a rajta elvégezhető műveleteket zárja egy logikai egységre
- objektumok csoportjának leírása, amelyeknek közösek az attribútumai, operációik és szemantikus viselkedésük
- úgy viselkedik, mint egy primitív típus, tehát pl. változó (objektum) hozható létre belőlük
- class kulcsszóval tudunk osztályokat definiálni
- tetszőleges mennyiségben létrehozhatunk példányokat (objektumokat)
- az objektum egy változó, melynek típusa valamely osztály, ami rendelkezik
 - állapottal: a tulajdonságok aktuális értéke, pl: bekapcsolva true vagy false
 - viselkedéssel: az objektum viselkedése (metódusok, pl: lámpa.bekapcsol())
 - identitással: minden objektum egyedi, még akkor is, ha éppen ugyanabban az állapotban vannak, és ugyanolyan viselkedést képesek megvalósítani

Öröklődés

- az osztályok között értelmezett viszony, amely segítségével egy általánosabb típusból (ősosztály) egy sajátosabb típust tudunk létrehozni (utódosztály)
- egyszeres és többszörös öröklődést
- Java
 - extends kulcsszóval tudjuk jelezni az öröklődést
 - egy osztály csak egy osztályból származhat, de több interfész implementálhat
 - super segítségével gyerekosztályból hivatkozhatunk ősosztályra
- C++
 - az osztály neve után egy kettőspontot követően vesszővel elválasztva lehet megadni az ősosztályokat és velük együtt a láthatóságaikat
 - megengedett a többszörös öröklődés
- lehetőség van az ősosztály tagjainak láthatóságán szigorítani (korlátozni)
- többszörös öröklődés esetén egy-egy ős osztály az öröklődési hierarchia különböző pontján ismét megjelenik (például gyémánt öröklődés)
- ebben az esetben egy virtuális ősosztály hozunk létre, amit az öröklődésnél az ősosztályok felsorolásakor virtual módon kell jelezni.

Újrafelhasználás és polimorfizmus

- az objektum orientált paradigmá egyik legfontosabb előnye
- egy változóra nem csak egyfajta típusú objektumként hivatkozhatunk
- Polimorfizmusról beszélünk, ha egy metódust egy ősosztály és egy gyermekosztály is tartalmaz (tehát felül lett írva), és a leszármazott egy példányán meghívjuk a metódust. Ekkor el kell döntenünk, hogy melyik osztály metódusát hajtsuk végre.

- **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 virtuális eljárások címének meghatározása indirekt módon, futás közben történik
- **Javaban csak virtuális eljárások vannak** (kivéve a final és a private metódusokat)
- **C++-ban a virtuális függvénytábla (VFT)** tartja nyilván a virtuális eljárások címeit
- **a VFT táblázat öröklődik, feltöltéséről a konstruktur gondoskodik**
- amikor a konstruálási folyamat véget ér, a táblázat minden sora értéket kap a létrehozott osztálynak megfelelő metódusok címeit.
- a VFT táblázat sorai ezután már nem változnak meg
- virtuális eljárásokat a virtual kulcsszóval tudunk létrehozni
- ha egy metódusról tudjuk, hogy azt a leszármazott osztályokban felül szeretnénk írni, akkor az legyen virtual
- **Java**
 - az absztrakt osztályok tartalmazhatnak (absztrakt) eljárásokat, adattagokat, viszont példányt belőlük nem hozhatunk létre.
 - az extends kulcsszóval származtatunk le
 - az interface-ek olyan osztályok, amelyek metódusok deklarációját tartalmazzák
 - funkcionális interface-k tartalmazhatnak default megvalósítást egyes metódusokra
 - az implements kulcsszóval származtatunk le
 - egy adott osztály tetszőlegesen sok interfész valósíthat meg
 - minden metódust meg kell valósítanunk az összes interfésből
 - az interfések sem példányosíthatók
- a polimorfizmust akkor láthatjuk, ha létrehozunk egy I interfész típusú X változót
- ez azt fogja jelezni a programban, hogy X egy olyan tetszőleges objektum lesz, ami megvalósítja I-t
- látható a többalakúság, ugyanis X-re egyszerre hivatkozhatunk úgy is, mint X, és úgy is, mint egy I-t megvalósító példány
- sokszor szeretnénk elkerülni (vagy feloldani) a többalakúságot
- egy általános módszer a **kasztolás (type cast)**, ahol meghatározzuk, hogy egy változóra milyen típusú objektumként fogunk a későbbiekbén hivatkozni
- **Java**
 - instanceof kulcsszó, ami megmondja, hogy az adott változó a meghatározott osztály példánya-e
 - rákérdezhetünk az X.getClass().equals(Object.class) metódussal is
 - primitívek esetén ott vannak a wrapper osztályok, mint például int → Integer
- **C++**
 - az std névtérben található egy is_base_of nevű metódus egy változó meghatározott osztály példánya-e

Programozás I, II - 2.)

Objektumok életciklusa, 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 életciklusa

Az objektumok tárolására több lehetőségünk is van: **statikusan** (az adatszegmensen, nem flexibilis, de gyors), **a lokálisan** (veremben, automatikus és gyors, de nem minden megfelelő), vagy **a dinamikusan** (heapben, dinamikus, futásidő közben, csak lassabb). Amennyiben egy leszármaztatott osztályt példányosítunk, akkor előbb meghívja az ősosztály(ok) konstruktorait, mielőtt elkezdené a saját eljárástörzsét végrehajtani. Javaban ez implicit megtörténik, ha az ősnek van default (alapértelmezett) konstruktora².

Java esetén az objektumok **mindig a heap-ben keletkeznek** (tehát az összes objektumot dinamikusan tároljuk), kivéve a primitív típusokat, azokat a **veremben** tároljuk (lokálisan). **Minden objektumot az adott osztály konstruktora inicializálja amikor példányosítjuk a new operátorral.** A létrehozás lépései:

- 1.) lefoglalja a szükséges memóriát
- 2.) meghívja az osztály konstruktörét
- 3.) visszaadja az objektumra mutató referenciát

Objektumokat tudunk másolni úgynevezett **copy konstruktorknak** létrehozásával, vagy a **Cloneable** interfész megvalósításával. Az első megközelítés során létrehozunk egy olyan konstruktort, amely egy, a saját magával azonos típusú objektumot vár, aminek kiveszi az értéket, és a saját megfelelő változóit is beállítja arra az értékre. Az utóbbi során viszont meg kell valósítanunk az interfészben található **.clone()** metódust, aminek keretein belül általában az ősosztály **.clone()** metódusának megfelelő értékre parse-olt változatát adjuk vissza.

Szerencsére itt nincs szükség az objektumok manuális törlésére, ugyanis a takarítást automatikusan elvégzi a virtuális gép **garbage collectorja** (GC, szemetgyűjtő). Ez biztonságosabb, a programozónak nem kell emlékeznie, hogy fel kell szabadítani az erőforrásokat, viszont sokkal lassabb. A szemetgyűjtést kézzel is el lehet indítani, de ez nem egyenlő a destruktörral, kiszámíthatatlan, hogy mikor fog végrehajtódni.

A **C++-ban** is hasonlóan működik a konstruktur: inicializálja az objektumot (feltölti az adattagjait a konstruktornak meghatározott értékekkel) és feltölti az objektumhoz tartozó **VFT-t** (virtuális függvénytáblázatot) a benne lévő metódusok megfelelő címével, majd visszaadja a létrehozott objektum referenciáját.

Az ilyen módon létrehozott objektumok a Javahez hasonlóan szintén a **heopen** jönnek létre, valamint az összes blokk szintű változó **statikusan (az adatszegmensen)** lesz létrehozva. Lokális objektumokat default paraméter vagy objektumokat tartalmazó kifejezésekben hozhatunk létre. Szokás még objektum konstansnak is nevezni őket.

² **Minden osztálynak van default konstruktora, ugyanis öröklik az Object osztályból, amiből származik minden objektum.** Pont ezért alapértelmezett a neve, nem feltétlenül kell felülírjuk.

C++-ban a heapbeli objektumok létrehozása szintén a new operátorral történik, viszont itt a megszüntetésükre is figyelni kell, amit a delete operátorral lehetünk meg. A létrehozáshoz nem elegendő a memória megfelelő méretben történő lefoglalása, hanem a konstruktur eljárását 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. Amikor tömböket foglalunk le, akkor minden a default konstruktur hívódik meg.

Objektumok klónozására szintén a copy konstruktoros megközelítés használható.

Az objektum megszüntetése előtti takarítást a destruktur végzi. A neve meg kell egyezzen az osztály nevével, ami elé egy ~ (tilde) jelet is kell tenni. Paramétere és visszaadott értéke nem lehet. A destruktur már nem állíthatja meg az objektum megszüntetését. Amikor a destruktur véget ér, az objektumot a rendszer a memoriából törli. Mindig a gyerek osztály destruktora hívódik meg először, és azt követi rekurzívan az ős osztályok destruktoraik a meghívása.

A statikusan létrehozott objektumok az adott blokk végén megszűnnék, amelyikben létre lett hozva. Lokális objektumokat default paraméter vagy objektumokat tartalmazó kifejezésekben használhatunk. Szokás még objektum konstansnak is nevezni őket.

A statikus adattagok, 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), valamint a statikus metódusok keretein belül csak és kizárolag más statikus memberekre hivatkozhatunk (gyakori fordítási hiba Java esetén a "Non-static reference"). Az ilyen adattagok, metódusok példányosítás nélkül is használhatóak.

Olyan esetekben lehetnek hasznosak, amikor az adott adattag, metódus független az objektumuktól, és mindenhol megegyezik az implementáció. Gyakori alkalmazása lehet az osztályokon belül egy statikus .convert() metódus létrehozása, ami paraméterül kap egy objektumot, és mindenből függetlenül, minden híváskor ugyanazon szabályok szerint alakítja át azt egy megfelelő típusba. Ezen felül még ismert lehet a Java programok belépései pontjául szolgáló main metódus deklarációjából is (`public static void main(String[] args)`) - érhető, hogy ebből sem kell több.

Operáció és operator overloading

Operációk kiterjesztéséről (overloading) akkor beszélünk, ha egy azonos nevű függvényt többször használunk különböző paraméterlistával. Ilyenkor a futtató környezet a paraméterek alapján fogja eldönteni, hogy az aktuális lépésben melyik végrehajtást kell választania. Erre van lehetőség C++ és Java esetében is.

Az operátor kiterjesztés viszont Java-ban már nem elérhető. Ez egy olyan folyamat, amikor felül szeretnénk írni, hogy hogyan reagálnak a létrehozott objektumaink bizonyos operátorok (például aritmetikai szimbólumok) előfordulása esetén. Lényegében ez felfogható úgy is, mintha a + jel által generált összeadás operációt írnánk felül.

Ki tudunk terjeszteni operátorokat osztályok (class) rekordok (struct) és uniók (union) esetén is, viszont tömbökre, pointerekre már nem működik. Számos operátort felül tudunk írni (mint például a +, -, (), [], /, stb), viszont vannak bizonyos, a rendszer által már lefoglalt operátorok, amiket nem lehet (., ::, ?, :, *, #, ##). Sejthető, hogy az operátorok precedenciája (kiértékelési sorrendje) nem változtatható meg. Ezek a felülírt operátor műveletek hasonlóan öröklődnek, mint az operációk, kivéve az =.

Példa operátor kiterjesztésére:

```
Complex operator+ (const Complex& obj) {
    Complex temp;
    temp.real = real + obj.real;
    temp.imag = imag + obj.imag;
    return temp;
}
```

Az operátor eljárások implementációjakor szükség lehet más osztályok **private** és **protected** adattagjainak elérésére. Erre használhatunk eljárásokat is, azonban előfordulhat egy olyan speciális eset, ahol számításba jöhet egy "barát" eljárás alkalmazása. Ezt a **friend** eljárást az osztály belsejében kell deklarálni. Nemcsak az eljárás lehet friend, hanem egy egész osztály is, így annak minden private vagy protected adattagját, metódusát elérjük. Lényegében ezzel meghatározhatjuk azt, hogy egy metódus habár az osztályon kívül van, az el tudja érni az ő egyébként elérhetetlen adattagjait.

Kivételkezelés

Előfordulhat, hogy a programok futásuk során valamilyen hibára futnak, és ilyenkor kivételeket dobhatnak. **Java** esetében ezeket két nagy csoportba osztjuk: **ellenőrzött**, és **nem ellenőrzött (futásidéjű)** kivételek csoportjába. Ahogy azt a nevük is mondja, az első csoportot le kell kezeljük (ellenőrizzük) abban az esetben, ha a futás közben előfordul, míg a másikat nem feltétlenül szükséges. De hogyan is kezeljük a kivételeket?

Alapvetően a beépített kivételek mellett szükség esetén létre tudunk hozni saját kivétel típusokat is úgy, hogy létrehozunk egy olyan új osztályt, ami a megfelelő kivétel típusból (Exception / RuntimeException) származik.

Ha megtaláltuk, hogy melyik az a kivétel, amit felügyelni szeretnénk, akkor azt a kódrészst, amelyben a kivétel előfordulhat, egy **try-catch-finally** struktúrába szervezzük. Itt ugyanis a vezérlés megpróbálja elvégezni a **try** részben megadott parancsokat, és ha a **catch** ágakban megadott hibák egyike előfordul, akkor kilép a végrehajtásból, és a megfelelő catch ágon fut tovább. A **finally** ág opcionális, ez a rész minden esetben le fog futni: ha találtunk hibát, ha nem. Ennek a struktúrának még egy változata a **try-with-resource** szerkezet, amiben a **try** blokk kezdete előtt meg tudunk adni olyan blokk szintű változókat, amelyek a végrehajtás végén mindenképp megszűnnék. Eredetileg ez volt a finally feladata (pl adatbázis kapcsolatok lezárása), viszont később ez az újítás bizonyos értelemben átvette a szerepét.

Előfordulhat, hogy nem feltétlenül szeretnénk egy hibát lekezelni: ebben az esetben a kockázatos kódrészletet tartalmazó metódus deklarációja után egy **throws** kulcsszóval jelezük, hogy a metódus futása során egy adott kivétel keletkezhet. Ehhez közel áll a **throw** utasítás, ami segítségével manuálisan dobhatunk kivételeket a program bármely pontján.

C++ esetén a kivételkezelés nagyon hasonló, annyi különbséggel, hogy az öröklési eljárás különbségből adódóan máshogyan hozunk létre saját kivételeket:

```
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();
    }
};
```

Összegzés

Objektumok életciklusa

- JAVA:
 - objektumok **heapben**, primitívek **veremben**
 - new operátorral példányosítunk
 - lefoglalja a szükséges memóriát
 - meghívja az osztály konstruktörét
 - visszaadja az objektumra mutató referenciát
 - másolás pl copy konstruktőrrel, vagy a Cloneable interfész megvalósításával
 - copy konstruktur esetén egy, az osztállyal azonos típusú objektumot várunk paraméterül, aminek értékeit beállítgatjuk a megfelelő változók értékére
 - meg kell valósítanunk az interfészben található .clone() metódust, aminek keretein belül általában az ősosztály .clone() metódusának megfelelő értékre parse-olt változatát adjuk vissza.
- C++:
 - a heapen szintén new operátorral hozunk létre objektumokat
 - az összes blokk szintű változó statikusan (az adatszegmensen) lesz tárolva
 - lokális objektumokat default paraméter vagy objektumokat tartalmazó kifejezésben hozhatunk létre (objektum konstans)
 - megszüntetés delete operátorral
 - létrehozáshoz nem elegendő a memória megfelelő méretben történő lefoglalása, hanem a konstruktur eljárását 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.
 - amikor tömböket foglalunk le, akkor minden a default konstruktur hívódik meg.
 - objektumok klónozására copy konstruktőrrel
 - az objektum megszüntetése előtti takarítást a destruktur végzi ~ (tilde), paramétere és visszaadott értéke nincs
 - a destruktur már nem állíthatja meg az objektum megszüntetését
 - ha a destruktur véget ér, az objektumot a rendszer a memoriából törli
 - minden a gyerek osztály destruktora hívódik meg először, és azt követi rekurzívan az ős osztályok destruktoraainak a meghívása.
 - a statikusan létrehozott objektumok az adott blokk végén megszűnnék

Statikus adattagok, metódusok

- static kulcsszó segítségével
- az ilyen adattagok és metódusok csak egy példányban jönnek létre
- a statikus metódusok nem lehetnek virtuálisak
- nem hivatkozhatnak az adott objektumra (this-re)
- nem hivatkozhatnak nem statikus memberre ("Non-static reference")
- az ilyen adattagok, metódusok példányosítás nélkül is használhatóak.
- hasznosak, amikor az adott adattag, metódus független az objektumuktól
- mindenhol megegyezik az implementáció
- `public static void main(String[] args)`

Operáció és operátor overloading

- operáció kiterjesztésről akkor beszélünk, ha azonos nevű függvényeket más paraméter kombinációkkal ismételten megadunk (ez van Javaban is és C++ban is)
- operátor kiterjesztésről akkor beszélünk, ha megszabjuk az osztályoknak, hogy azok példányai bizonyos szimbólumok előfordulásakor hogyan működjenek (ez C++ban van)
- némely operátor (., ::, ?, *, #, ##) nem felülírható
- sok (+, ++, -, --, =, (), []) viszont igen
- az = kivételével az operátor kiterjesztések öröklődnek
- osztályokra, rekordokra, uniókra kiterjeszthetők, tömbökre, pointerekre nem

Kivételkezelés

- try-catch-finally
- throws, throw
- try-with-resource
- ellenőrzött (checked) és nem ellenőrzött (unchecked, futásidéjű, runtime) kivételek
- `class MyException : public std::exception`

Programozás I, II - 3.)

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.

Java és C++ programok fordítása és futtatása

Alapvetően erősen különbözik a két nyelv fordítása, ugyanis a Java-t egy virtuális gép futtatja az operációs rendszer felett (JVM, vagy Java Virtual Machine), míg a C++ fileok esetén gépi kódra fordítunk.

Ahhoz, hogy Java programokat tudunk futtatni, illetve fejleszteni, szükségünk lesz egy fordító- és/vagy futtatókörnyezetre, valamint egy fordítóprogramra. A kész programunk futtatásához mindenkorábban a JRE (Java Runtime Environment) szükséges, ami biztosítja a Java alkalmazások futtatásának minimális követelményeit, mint például a JVM (Java Virtual Machine). A fejlesztéshez azonban szükségünk lesz 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:

- a .java kiterjesztésű fájlokat a Java-fordító (compiler) egy közönséges 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.

Fordítási és futtatási parancsok:

```
> javac pelda.java  
> java pelda
```

A fordításhoz megadhatunk még pár opciót kapcsolók segítségével:

- **-g:** debug információk generálása
- **-s <könyvtár>:** a generált fájlok könyvtárának megadása
- **-sourcepath <path>:** a forrásfájlok elérési útvonalát meg tudjuk így megadni
- **-Werror:** figyelmeztetés esetén megáll a fordítás

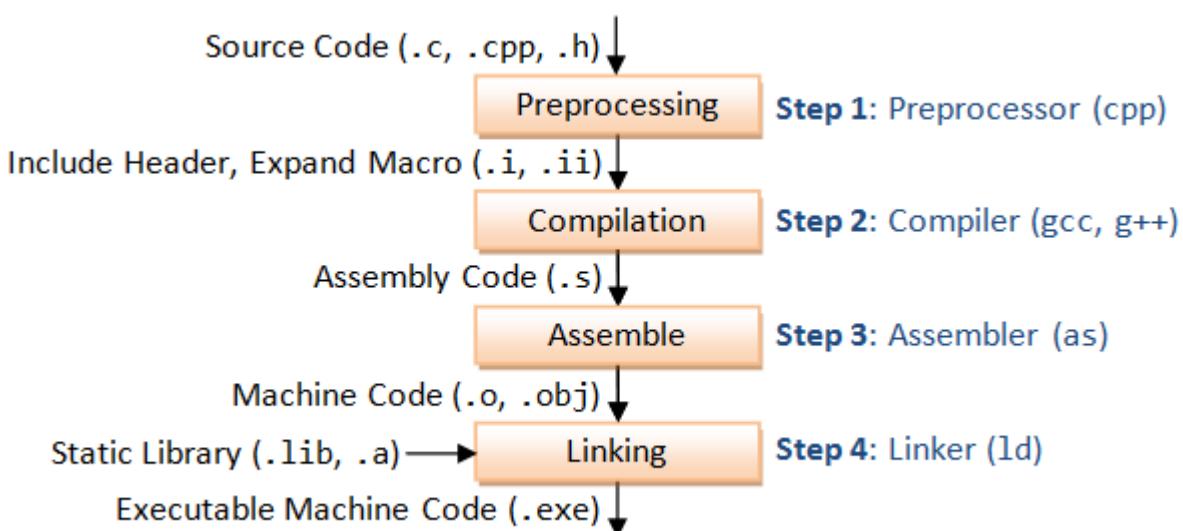
Egy valamire való fejlesztőkörnyezet (IDE, Integrated Development Environment) lehetővé teszi azt is, hogy futtatási konfigurációkat hozunk létre, így akár egy kattintással is tudjuk futtatni a programjainkat. A Java programok belépési pontja a **public static void main(String[] args)** függvény, ahol a String[] args paraméter tartalmazza a program parancssori argumentumait. Ezeket úgy tudjuk a programnak megadni, hogy futtatás után a program után beírjuk ezeket, amit a main törzsében kedvünkre használhatunk.

Nagyobb projektek esetén érhető okokból nem skálázódik túl jól ez a fordítási módszer (ugyanis sok osztály esetén azokat egyesével kell megadni, vagy *.java megadása esetén az összes mappát fel kell keresni). Ezért jelentek meg az úgynevezett **build toolok**, mint például az **Ant**, **Gradle**, **Maven**, **Makefile** és társai. Értelemszerűen ezek minden rendelkeznek saját kapcsolókkal.

C++ kódok fordításánál a C-hez hasonló **tool chain-t** használunk. Első lépésben az előfordító (**preprocessor**) a tényleges fordítóprogram futása előtt szövegesen átalakítja a forráskódot. Elsődleges feladatai:

- header fileok behúzása, #include direktívák értelmezése
- rendezi a program sorait
- kommentek helyettesítése whitespace karakterekkel
- makrók (mint például #define) helyettesítése

Ezt követően következik a fordító (**compiler**), amikor a C++ forráskódból Assembly kódot készítünk. Ezután következik az **Assembler**, ami ebből tárgymodulokat (.o) készít. Ezek önmagukban még mindig nem futtatható állományok, viszont utolsó lépésben a **Linker** ezeket összeszerkeszti, és futtatható programot készít belőlük.



Linux/Unix alapú gépek esetén az alapértelmezett fordító a **gcc**. Az alábbi módon tudjuk lefordítani az akár több forrásfájlból álló projektet:

```
> gcc -o prog main.cpp class1.cpp class2.cpp
> ./prog
```

Ahogy az látható, a belépési pontot tartalmazó fájlon kívül felsoroltuk még a többi fordítandó fájt is, valamint a -o (output) kapcsoló segítségével megadtuk a fordítás kimenetének nevét. További kapcsolók:

- **Ob[szint]**: optimalizálások alkalmazása, a szint maximum 3 lehet (0,1,2), inline eljárások.
- **-c**: mint compile, lefordítja és összeállítja a forrást, linkelést nem végez
- **-o**: mint output, megadhatjuk a futtatható állomány nevét (alapértelmezetten a.out lesz)
- **-Wall**: további információkat ad fordítás során
- **-g**: mint debug, hibakeresési információk elhelyezése a programban gdb (GNU debugger) részére

Ahogy Javaban, C-ben is van a programnak belépési pontja az `int main(int argc, char* argv[])` függvény keretein belül. Az első paraméter (argc) az argumentumok számát, míg a második (argv) az argumentumokat tartalmazza karakterláncként. Itt argv[0] a program nevét és útvonalát tartalmazza, tehát az érdemleges paraméterek valójában argv[1]-től kezdődnek.

Absztrakt-, interfész- és generikus osztályok, virtuális eljárások

Java esetében az **absztrakt osztályok** olyan osztályok, amelyek ugyan tartalmazhatnak **adattagokat**, **metódus deklarációkat** (absztrakt metódusok) vagy implementációkat, sosem lehet őket példányosítani. Akkor használjuk őket, ha általánosítani szeretnénk a program egy logikai egységét. Az **abstract** kulcsszó segítségével tudjuk őket a gyakorlatban létrehozni. Ha egy metódus elő tesszük ezt, akkor az nem tartalmazhat megvalósítást (törzset), és a leszármazott osztályokban meg kell őket valósítani. Ha az osztálynak van legalább egy absztrakt metódusa, akkor az osztálynak is annak kell lennie.

Hasonló működéssel rendelkeznek az **interfészek** is. Ezek olyan különleges absztrakt osztályok, amelyek csak **metódusok deklarációját tartalmazhatják**, azok megvalósítását nem. Java 11-ben érkezett egy újítás, amiben bevezették a **funkcionális interfészeket**, amik már tartalmazhatnak **default** (alapértelmezett) megvalósításokat bizonyos függvényekhez. Az **interface** kulcsszóval lehet őket létrehozni úgy, mint egy osztályt.

Fontos lehet itt megjegyezni, hogy egy tetszőleges osztály minden osztályból származhat le (legyen az absztrakt, vagy sem), viszont több interfészt is megvalósíthat.

C++ban annyi a különbözőség, hogy **interfészek létrehozására nincs lehetőség**, viszont **absztrakt osztályokkal** helyettesíthetjük őket. A bennük található, törzs nélküli virtuális eljárásokat **pure virtual** eljárásoknak nevezünk (`virtual int getArea() = 0`). Ezek egy üres bejegyzést foglalnak el a VFT-ben. Ha egy osztály ilyen eljárásot tartalmaz, akkor azt absztrakt osztálynak tekintjük, mivel ebből az osztályból példányokat nem lehet létrehozni. Értelemszerűen a gyermekosztályokban minden pure virtual eljárás meg kell valósítani, és ezt a fordító ellenőrzi is.

A **generikus** osztályok alapelve azonos a két programnyelvben, csupán csak megvalósításban különböznek. A generikus programozás lényege az, a lehető legáltalánosabb leírását adjuk meg adott kód részleteknek, és így a fordítóra bízzuk, hogy a nem specifikusan meghatározott részeket létrehozza (**generáljon**). Ha tudjuk például, hogy egy algoritmus adattípusról függetlenül ugyanúgy működik, akkor érdemes azt egy generikus osztályban tárolni.

Javában gyakorlatilag statikus polimorfizmusról van szó, ami a gyakorlatban úgy néz ki, hogy az osztály deklarációjában meg kell adnunk egy (vagy több) **generikus paramétert** <> között, amit később a metódusok megadásánál újra felhasználhatunk. Vegyük példának a ArrayList generikus osztályt. Ez megvalósítja a List generikus interfészt, ami jelenleg nem is annyira fontos. Amilyen értéket megadunk az ArrayList típus meghatározása után, az a lista olyan típusú értékeket fog tárolni. Például `ArrayList<String>` egy karakterláncokból álló listát fog képezni, és a metódusai, mint például az `.add()` egy ilyen típusú paramétert fog a későbbiekbén erre a példányra várni. Primitív típusokat nem lehet generikus paraméterként megadni, erre vannak a **wrapper osztályok** (mint például `int → Integer`).

Lehetőségünk van még a típusparaméterek megszorítására is, ugyanis kiköthetjük, hogy az egyes paraméterek melyik osztályból származzonak le, vagy melyik interfészket valósítsák meg. Ekkor a fejlesztés során más lehetőségeket is elérhetővé teszünk. Létre tudunk hozni generikus metódusokat generikus osztályokon kívül is **wildcardok használatával**. Ezzel lényegében egy **capture** blokkot hozunk létre, ami az adott típusokat fogadja el (például `List<? extends A> list`).

Gyakori típus elnevezések: **E** (element), **K** (key), **N** (number), **T** (type), **V** (value)

C++ban a generikus osztályokat sablonok (**template**) segítségével tudunk létrehozni. Ezek olyan speciális funkciók, amelyek generikus típusokkal működnek. Lényegében a funkcionalitásuk ugyanaz: több típusra működnek anélkül, hogy a kód bármekkora részét is újraírnánk.

Példa generikus osztályokra:

Java	C++
<pre>public class Storage<T> { private List<T> items; public Storage() { this.items = new ArrayList(); } public void addItem(T item) { this.items.add(item) } }</pre>	<pre>template <typename T> class Storage { private: T* items; int i = 0; public: Storage(int s) { this.items = new T[s]; } void addItem(T item) { this.items[i] = item; i++; } }</pre>

A virtuális eljárások megvalósítása, szerepe, használata

Egy virtuális eljárás címének meghatározása indirekt módon, futás közben történik. Javaban eleve csak virtuális eljárások vannak (kvíve a final és a private metódusokat). C++-ban a virtuális függvénytábla tartja nyilván a virtuális eljárások címeit. A VFT táblázat öröklődik, feltöltéséről a konstruktur gondoskodik. A származtatott osztály konstruktora módosítja a virtuális függvénytáblát, kijavítja az ősosztályból 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.

Meghíváskor akkor tud a megfelelő eljárás aktiválódni, ha az objektumpéldányok megőrzik a létrehozási típusukat. Mivel a példányok címét tartalmazó pointerek típusát a program futása során igény szerint megváltoztathatjuk ős vagy gyerekosztály típusra, ezért meg kell különböztetnünk **static** és **dynamic** típusokat egymástól.

A **static** type a forráskód alapján, a változók típusából kikövetkeztethető típus. Ennek nem sok köze van a létrehozás előtt a típushoz, attól lényegesen eltérhet, viszont a fordítóprogram ebből indul ki. Értelemszerűen ha a forráskódban valami CType* típussal szerepel, akkor a fordító automatikusan a CType osztály egy példányára mutató pointerként fogja kezelni.

A **dynamic** type akkor határozható meg, ha az objektumpéldányok tárolják, és megőrzik a létrehozás előtt a típushoz, attól lényegesen eltérhet, viszont a fordítóprogram ebből indul ki. Amennyiben ez alapján működő kódot képes a fordító előállítani, akkor a program intelligensebb módon, a mindenkor létrehozási típus alapján, futás közben, a hívás előtt határozza meg a meghívandó virtuális eljárás címét.

Összegzés

Java és C++ programok fordítása és futtatása

- JAVA
 - java fejlesztéshez kell JDK
 - futtatáshoz JRE
 - java programokat JVM futtatja
 - fordítás folyamata:
 - a .java kiterjesztésű fájlokat a Java-fordító 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 és futtatás: javac és java parancsokkal
 - javac kapcsolók: -g, -s <könyvtár>, -sourcepath, -Werror
 - fejlesztői környezet (IDE) lehetővé, hogy futtatási konfigurációkat hozzunk létre
 - belépési pont a public static void main(String[] args)
 - String[] args paraméter tartalmazza a program parancssori argumentumait
 - nagyobb projektek esetén érdemes build toolokat használni (Ant, Gradle, Maven)
- C++
 - C-hez hasonló tool chain
 - preprocessor - compiler - assembler - linker
 - Linux rendszereken gcc
 - megadható olyan kapcsoló neki, amivel a fordítás csak némely részét végzi el
 - kapcsolók: Ob[szint], -c, -o, -Wall, -g
 - belépési pont az int main(int argc, char* argv[]) függvény
 - argc az argumentumok száma
 - argv az argumentumok
 - argv[0] a program nevét és útvonalát tartalmazza

Absztrakt-, interfész- és generikus osztályok

- ABSZTRAKT OSZTÁLYOK
 - tartalmazhatnak adattagokat
 - metódusokat, vagy csak metódus deklarációkat (javaban absztrakt, C++-ban pure virtual metódusok)
 - ha van legalább egy deklaráció, akkor az osztály absztrakt
 - nem példányosíthatók
- INTERFÉSZEK
 - speciális absztrakt osztályok
 - Javaban az interface kulcsszó segítségével létrehozható
 - csak metódusdeklarációkat tartalmaz
 - funkcionális interfések tartalmazhatnak alapértelmezett (default) megvalósítást
- GENERIKUS OSZTÁLYOK
 - a lehető legáltalánosabb leírását adjuk meg adott problémák megoldására
 - a fordítóra bízzuk, hogy a nem meghatározott részeket létrehozza (generálja)
 - ha egy algoritmus adattípusról függetlenül működik, akkor kell ezt használni
 - Javaban statikus polimorfizmusról
 - primitív típusok nem lehetnek generikus paraméterek (wrapperek, int → Integer).

- típusparaméterek megszorítása: kiköthetjük, hogy az egyes paraméterek melyik osztályból származzonak le, vagy melyik interfészket valósítsák meg
- wildcardok használata, például `List<? extends A>` list
- gyakori elnevezések: E (element), K (key), N (number), T (type), V (value)
- C++ban a generikus osztályokat sablonok (template) segítségével tudjuk létrehozni

Virtuális eljárások megvalósítása, szerepe, használata

- virtuális eljárás: futás közben határozódik meg a memóriacíme
- Javaban minden metódus virtuális
- C++-ban lehet statikus és dinamikus virtuális metódus
- static a forráskód alapján, a változók típusából kikövetkeztethető típus
- dynamic akkor határozható meg, ha az objektumpéldányok tárolják, és megőrzik a létrehozáskor kapott típusukat

Programozási nyelvek - 1.)

A programozási nyelvek csoportosítása (paradigmák), az egyes csoportokba tartozó nyelvek legfontosabb tulajdonságai.

A programozási nyelvek csoportosítása (paradigmák)

A programozási **paradigma** egy olyan osztályozási forma, amely a programozási nyelvek jellemzőin alapul. Nyelvcsoportok:

- Imperatív / procedurális (C, C++, Pascal)
- Objektumorientált (C++, Java, Smalltalk)
- Applikatív, funkcionális (Haskell, ML)
- Szabály alapú, logikai (Prolog, HASL)
- Párhuzaos (Occam, PVM, MPI)

Az imperatív / procedurális paradigmája: alapköt a kifejezések képezik, ami megváltoztatják a program állapotát (asemblynél például a mov parancs beletesz egy értéket az adott regiszterbe, és tovább tolja a PC - program counter - értékét, ergo változtat a program állapotán). Egy imperatív program olyan parancsokból áll, amelyet a számítógépnek el kell végeznie. A procedurális programozás ezt annyival bővíti ki, hogy ezeket az utasításokat procedúráakra (angolul subroutine, functions) csoportosítja. Ezekre a nyelvekre jellemző a **pointer** típus, segítségével hatékonyan tudunk bizonyos elemekre hivatkozni, mert azokról nem kell másolatot készítenünk, hanem közvetlenül tudjuk őket módosítani. Felfogható úgy is, mint a memóriacímek absztrakciója (ha nem vagyunk biztosak a pointerek kezelésében, használhatunk helyettük **referenciákat**, amik nagyjából ugyanazt csinálják, C-ben például ezt & segítségével tudjuk megcsinálni).

Az objektumorientált paradigmája: ez a paradigmája az objektumok fogalmán alapuló programozási paradigmája. Az objektumok olyan absztrakt adattípusok, amelyek egysége foglalják a logikailag összefüggő adatokat és a hozzájuk tartozó műveleteket. A program egymással kommunikáló objektumok összességeből áll, amelyek használják egymás műveleteit és adatait. A legfontosabb tulajdonság az ilyen nyelvknél talán az **örökölés**, amely során a leszármaztatott osztályok megkapják az őseinek változóit és eljárásait, amiket esetlegesen akár felül is írhatnak, kiegészíthetnek, stb. Ezen felül még sok fontos tulajdonsággal bír az OOP, ezekre a Programozás I., II. tételek részletesebben kitérnék.

A funkcionális paradigmája: értékek, (matematikai) kifejezések és függvények kombinációból épül fel a kód. Maga a program is egy függvény: egy meghatározott belépései pontba kerülünk, ha egy ilyen nyelven írt programot futtatunk. Ciklusok helyett **rekurziót** kell használni. Az ilyen jellegű programokkal **nehéz** interakcióba lépni (input/output értékeket átadni, kivenni), viszont helyességük ellenőrzése sokkal könnyebb.

A logikai paradigmája: egy logikai program csak az adatoka, és az azok közötti összefüggéseket tartalmazza. Kérdések hatására a végrehajtást **egy beépített következetető rendszer** végzi el. Prologban például a programozás úgy néz ki, hogy megadjuk az objektumokat, az azokon értelmezett relációkat, és a relációkkal kapcsolatos kérdéseket. Ahogy logikában is, itt is az értékeket **termnek** nevezzük, például egy term lehet *alma*, *1000*, vagy akár *mary* is. Ezt követően meg kell adnunk azt, hogy ezek között milyen kapcsolat (reláció) áll fenn, például *likes(mary, alma)*, ami annyit tesz, hogy Mary szereti az almát.

Miután az összes ilyen relációt felvettük, "kérdéseket tehetünk fel a programnak interaktív módon", amik lehetnek **eldöntendő** (Mary szereti az almát?), vagy **általános kérdések** (ki szereti az almát?). Az ilyen kérdésekre válaszul **tényeket** kapunk, amik a megadott objektumok közötti relációkat reprezentálják, így képezve egy adatbázist.

Az ilyen logikai programok esetében könnyen meg tudjuk különböztetni az általános és eldöntendő kérdéseket úgy, hogy az **általános kérdések tartalmaznak egy változót**, míg az **eldöntendők konkrét értékekre vannak értelmezve**. Hogyan is néz ki ez a gyakorlatban?

```

alma;                      // alma term letrehozasa
mary;                       // mary term letrehozasa
joe;                        // joe term letrehozasa
likes(mary, alma);         // relacio letrehozasa a ket term kozott

?- likes(mary, alma);     // kerdes feltetele a programnak
> True.                    // az eldontorendszer valasza

?- likes(joe, alma);      // ujabb kerdes
> False.                   // false, mivel nem talalt ilyen kapcsolatot a
                           // rendszer
?- likes(X, alma);        // ki szereti az almat? kerdes feltetele
> mary.                    // az adatbazis vizsgalata utan erre a dantesre
                           // jutott a rendszer

```

A párhuzamos paradigmája: számos párhuzamos programozási modell létezik, amelyek **osztoznak** **pár közös problémán**. Ilyenek például a közös hozzáférés (**közös memória van**, amin a tárolt adatot egyszerre több végrehajtás is módosíthatja), a folyamatok létrehozása, megszüntetése, kezelése és együttműködése (**független**, **versengenek az erőforrásért**, **együttműködő**). Itt még gondot okozhat az is, hogy az ilyen programok elégé **kiszámíthatatlanok**, ugyanis a folyamatok relatív (egymáshoz viszonyított) **sebessége minden futáskor más lehet**. Előfordulhatnak olyan esetek is, hogy ugyanarra az inputra más outputot generálnak, esetleg **holppontba (deadlock)** juthatnak (kölcsönös egymásra várakozás két külön futó, párhuzamos eljárásnál), vagy akár egyszerűen egy folyamat **éhezik (starvation)**, amikor nincs holppont, mégsem jut hozzá az erőforráshoz.

Egy ilyen paradigmán alapuló nyelv az **Occam**, ami **imperatív folyamatokat használ**, **saját memoriával**, amik **üzenetküldéssel kommunikálnak egymással**. Egy occam program változókból, folyamatokból és csatornákból áll, amiket szigorú formai követelményekkel rendszerez. Rendelkezik pár egyszerű (bool, byte, int, int16, int32..., real32, real64)-, és egy összetett (tömb) típussal, amiknek így adhatunk értéket: INT x := 5.

Említésre kerültek a **csatornák**: ezeknek a **fő feladata a folyamatok közötti adatátvitel**. Ezek **egyirányú, biztonságos, szinkron** (a küldő és fogadó bevárják egymás, ergo a fogadó nem kér addig, ameddig a küldő nem küldött neki semmit) csatornák, amiket a CHAN OF INT c: parancssal hozhatunk például létre. Egy folyamat egyszerre egy küldő, és egy fogadó csatornával rendelkezhet. Ha már így szó esett róluk, a folyamatok életciklusa három részből áll: fogja magát, **elindul, csinál valamit, aztán befejeződik** (terminál). Nyilván ha a középső részen holppontba kerültünk, akkor az utolsó rész nem következik be, erre nem árt odafigyelni.

Összegzés

Az imperatív / procedurális paradigmá

- C, C++, Pascal
- kifejezések alapul
- egy ilyen program olyan parancsokból áll, amit a számítógépnek el kell végeznie
- procedurális programozás esetén ezeket a parancsokat eljárásokba (procedúrákba) csoportosítjuk
- előnye a pointer típus (biztonságosabb a referencia), felfogható, mint a memóriacímek absztrakciója
- hatékony adatcímzés, közvetlen értékmódosítás

A objektumorientált paradigmá

- C++, C#, Java, Pascal
- az objektumok fogalmán alapul, amikből olyan adattípusokat képzünk, amelyek egységbe foglalják a logikailag összefüggő értékeket és a hozzájuk tartozó műveleteket
- számos jó tulajdonsággal rendelkeznek, ilyenek például az öröklődés, overloading és sok más
- Java és C# esetében például nem túl hatékony a memória elérése, ugyanis ezeket egy virtuális gép futtatja, ami nem ad hozzáférést közvetlenül a memoriához

A funkcionális paradigmá

- Haskell
- értékek, (matematikai) kifejezések és függvények kombinációja
- az ilyen program is egy függvény: egy meghatározott belépési pontba kerülünk, ha futtatjuk
- ciklusok helyett rekurziót kell használni
- az ilyen jellegű programokkal nehéz interakcióba lépni (input/output)
- helyességük ellenőrzése nagyon könnyű (rekurzió néha áttekinthetőbb, mint egy ciklus)

A logikai paradigmá

- Prolog
- csak az adatokat, és az azok közötti összefüggéseket tartalmazza
- kérdések hatására a végrehajtást egy beépített következtető rendszer végzi el
- az értékeket termnek nevezzük (alma, 1000, 10, mary, stb)
- meg kell adnunk azt, hogy ezek között milyen kapcsolat (reláció) áll fenn (likes(mary, alma))
- miután az összes ilyen relációt felvettük, "kérdéseket tehetünk fel a programnak interaktív módon"
- ezek lehetnek eldöntendő (Mary szereti az almát?), vagy általános kérdések (ki szereti az almát?).
- a kérdésekre válaszul tényeket kapunk, amik a megadott objektumok közötti relációkat reprezentálják (a relációk egy adatbázist képeznek)
- az általános kérdések tartalmaznak egy változót, míg az eldöntendők konkrét értékekre vannak értelmezve.

A párhuzamos paradigmá

- Occam
- PROBLÉMÁK
 - közös hozzáférés (közös memória van, amin a tárolt adatot egyszerre több végrehajtás is módosíthatja),
 - folyamatok létrehozása, megszüntetése, kezelése és együttműködése (függetlenek, versengenek az erőforrásért, együttműködnek)
 - kiszámíthatatlanok, a folyamatok relatív (egymáshoz viszonyított) sebessége minden futáskor más lehet
 - ugyanarra az inputra más outputot generálhatnak
 - holtpontba (deadlock) juthatnak (kölcsönös egymásra várakozás két külön futó, párhuzamos eljárásnál),
 - egy folyamat éhezik (starvation), amikor nincs holtpont, mégsem jut hozzá az erőforráshoz.
- Occam imperatív folyamatokat használ saját memoriával, amik üzenetküldéssel (csatornák segítségével) kommunikálnak egymással
- változókból, folyamatokból és csatornákból áll
- szigorú formai követelmények
- pár egyszerű (bool, byte, int, int16, int32, real32, real64)-, és egy összetett (tömb) típus
- a csatornák egyirányú, biztonságos, szinkron tulajdonságokkal rendelkeznek
- egy folyamatnak egy bemeneti és egy kimeneti csatornája lehet
- két folyamat követheti egymás sorban (SEQ), vagy párhuzamosan (PAR) futhat mellette

Rendszerfejlesztés I - 1.)

Szoftverfejlesztési folyamat és elemei; a folyamat különböző modelljei.

Szoftverfejlesztési folyamat

Szoftverfejlesztési folyamatnak nevezzük egy bizonyos modell szerinti tevékenységek összességét, amelynek eredménye a szoftvertermék. Ez több komponensből összességeből áll:

- fejlesztési folyamat
- menedzsment
- technikai módszerek
- eszközök használata

Az elkészült termék esetében nem csak az elvárt funkcionális és teljesítmény a fontos, hanem az üzembiztonság, jó használhatóság és karbantarthatóság is. Mielőtt még a fejlesztés kezdetét venné, részletesen meg kell tervezni a lépéseket:

- 1.) Kezdetben van egy terv, ami valószínűleg az idő műlásával módosulni fog a célok és körülmények változásából eredően
- 2.) Ezután összegezni kell a megszorításokat (mennyi fejlesztő lesz elérhető, azok naponta mennyi órát tudnak dolgozni, a kliensnek mikorra kellenek a termék bizonyos részei)
- 3.) Ezen felül még a paramétereket (szoftver mérete, esetleges változtatások nagysága) is mérlegelni, becsülni kell,
- 4.) Majd elkezdődhet ennek ismeretében a mérföldkövek, és átadandók terve.
- 5.) Miután ez elkészült, egy ütemtervet kell létrehozni, amely megmondja, hogy a fejlesztésben résztvevő felek min mikor fognak dolgozni a mérföldek között
- 6.) és zárásképp minden tervből meghatározássá kell alakítani

A projekt ütemezése egy komoly feladat, ahol több szempontot is figyelembe kell venni. Ilyen például a tevékenységekhez szükséges idő, aminek becslése talán a legnehezebb (kezdetben érdemesebb pozitívnak lenni: adjunk rá 50%-ot). Ezen felül szem előtt kell tartani azt is, hogy a rendelkezésünkre álló munkaerőt optimálisan használjuk ki a fejlesztők igényei függvényében (személy-nap/személy-hónap alakulása szabadságok miatt, váratlan kimeradások, stb).

Fő elemei:

- feladatak (termékek)
- átadandók (határidők)

Kiegészítő elemek:

- Projektmenedzsment (Project Manager)
- Konfigurációmenedzsment (Business Analyst - BA)
- Dokumentáció (Developers - DEV)
- Minőségbiztosítás (Quality Assurance - QA - tester), kockázatmenedzsment
- Mérés (Testers)

Fázisai:

- Specifikáció (mit csinálunk)
- Fejlesztés (hogyan csinálunk)
- Validáció, verifikáció (jól csináltuk-e, ellenőrzés)
- Evolúció (karbantartás, felügyelés, "3RD level support")

Specifikáció: meghatározzuk a termék funkcionalitását és az esetleges hozzá tartozó megszorításokat (eldöntjük milyen funkciókat, szolgáltatásokat követelünk meg a rendszertől). Fontos kellő időt eltölteni a tervezéssel, ugyanis **itt a legkisebb az esetleges változások költsége** (mivel itt még nem kezdtünk el fejleszteni semmit, értelemszerűen így nem lesznek "kidobott" munkaórák). Eredménye egy dokumentum (esetleg spike-ok / prototípusok), amik később a fejlesztők munkáját segítik.

Fejlesztés: a specifikáció megvalósítása. Mielőtt még kezdetét venné a tárgyilagos fejlesztés, ezen fázis keretein belül egy **újabb tervezés veszi kezdetét**, ahol a termék struktúráját, adatszerkezetét és alrendszerének hierarchiáját (interfészek, osztályok, stb) **tervezik meg elsősorban a fejlesztők** (esetlegesen készülhetnek el még EK és más diagramok is). Fontosabb tervezési modellek:

- Structured Design (SD),
- SSADM,
- OMT (objektumorientált),
- UML (objektumorientált).

Miután a tervezés véget ért, kezdetét veheti a fejlesztés, ami magába foglalja magát a programozást, és a nyomonkövetést (debugging).

Validáció: ebben a fázisban **ellenőrzik**, hogy a program megfelel-e az előírt specifikációknak, és a megrendelő elvárásainak. Ide tartozik a **szoftver manuális-, és kóddal történő tesztelése is**, ez utóbbi típusai a következők lehetnek:

- Egységesztelés (**unit test**): a komponenst magában, a többiből függetlenül
- Modul tesztelés (**integration test**): a függő és kapcsolódó komponenseket együtt
- Alrendszer tesztelése: több modul közös illeszkedésének vizsgálata (általában egy független csapat által)
- Rendszer tesztelése: a módosított alrendszeret közösen teszteljük előre nem várt kölcsönhatásokat keresve
- Átviteli tesztelés: a megrendelő adataival valós környezetben történik (TEST-ENV)
- Béta tesztelés: olyan hibák keresése, amiket a fejlesztők nem láthattak (PRODUCTION)

Evolúció: a szoftver működésének nyomon követése, esetleges hibák kezelése (meggyezés alapján javítása - 3RD level support). Az **itt felmerülő problémák** sajnos már költségesek lehetnek a megrendelőnek, viszont az átvételt követően ezeknek kiküszöbölése sokszor plusz kiadással jár.

Szoftverfejlesztési folyamat modelljei

Vízesés modell: egy **szekvenciális** modell, ahol a fázisok lépcsőzetesen követik egymást. **Minden egyes fázis befejezésekor annak teljesen el kell készülni**, mielőtt a következő lépésre áttérnénk (az esetleges hibákat összegyűjtik a fázisok végén, majd a folyamat legvégére érve azokat javítják). Látható, hogy ez a modell **csak akkor működhet jól, ha a specifikáció nagyon jó**, viszont azokat **nagyon nehéz pontosan átlátni a projekt legelején**, ezért érhető okokból ritkán van egyszerű, lineáris fejlesztés. Ezen felül hátrányt jelent az is, hogy a **megrendelő a termékkel csak a fejlesztés végén találkozik a termékkel**, ahol ugyanis már csak nagy költséggel lehet orvosolni a problémákat.

Iterációs (inkrementális) modell: ez már egy kicsit kötetlenebb model, ahol a fejlesztési folyamat iterációja elkerülhetetlen. Ha a követelmények változnak, akkor a folyamat bizonyos részei is vele változnak, ezért azokat csak nagy körvonalakban kell specifikálni - a végső követelmények úgyis menet közben alakulnak ki. Általában akkor beszélünk ilyen modellről, amikor egy már működő rendszert fejlesztünk tovább, azaz ha elkészül egy részeredmény, azt megvizsgálják, és kezdődik minden előről - tehát **probálunk a terméken minden ilyen iterációban javítani**. Az iterációkban tetszőlegesen választható fejlesztési modell, az előzőben alkalmazottól függetlenül. minden inkremens akár szolgálatba is állítható, ezért ha esetleg határidő csúszás alakulna ki, akkor nem vész kudarcba az egész projekt, hanem csak bizonyos inkremensek. Egy elégé érezhető problémája az, hogy az inkremensek meghatározása nem triviális feladat, ugyanis a túl kicsi inkremensek nem feltétlenül lesznek működőképesek, a túl nagy inkremensek miatt viszont elveszítjük a modell lényegét.

Evolúciós (prototípusos) modell: lényegében egy szélsőséges iterációs modell, ahol egy durva specifikációra a csapat gyorsan fejleszt egy prototípust, amit a megrendelő kiértékel, majd újra ír követelményeket az akkori működés alapján. Sejthető, hogy ez a gyakorlatban **sok iterációt vesz igénybe**, mire a termék felveszi végső állapotát, habár a prototípusok száma csökkenhető, ha olyan előre megírt komponenseket használunk, amiket a kliens már korábban elfogadott. Ezen tovább ronthat az a tény, miszerint még csak nem is biztos, hogy a prototípus lesz a végtermék. Ebből következik, hogy ez a modell **sok belefektetett időt igényel a megrendelő részéről is, ugyanis előfordulhat, hogy sok prototípust kell megvizsgálni**. Ebből ered sokszor az a probléma is, ahol a megrendelő egy prototípusról azt gondolja, hogy már kész rendszer, és nehezen tud ellenállni, hogy ne használja. Érthető, hogy itt még a termék nem biztos, hogy készen áll, és a gyors fejlesztés miatt sokszor nem a legoptimálisabb megoldások kerülnek be egy prototípusba.

Extreme Programming (XP): egy elég szélsőséges modell, ahol **meghatározott idő alatt kell kis funkcionálitású inkremeneseket megvalósítani**. Ebben az esetben fontos a megrendelő intenzív részvételle, hogy gyorsan tudjon reagálni az esetlegesen felmerülő kérdésekre. Gyakori ilyen esetekben még **a cooperative programming is, aminek során több fejlesztő ül egy képernyő előtt, és közösen fejlesztenek**. Sokak szerint ez nem egy hatékony fejlesztési modell, érthető okokból (ahogy a mondás is tartja, a jó munkához idő kell).

Rapid Application Development (RAD): jellemzően **extrém rövid életciklus alatt kell (általában) a semmiből működő rendszert létrehozni (körülbelül 60-90 nap alatt végbemegy ez a történet)**. Felfogható úgy, mint egy "turbó", nagy sebességű vízesés modell. Rendszerint több emberből álló fejlesztőegységek dolgoznak párhuzamosan, különböző komponenseken a hatékonyság érdekében. Itt szintén fontos a megrendelő intenzív együttműködése, hasonló okokból, mint az XP esetében. Bár egyszerűnek tűnik ez a modell, mégsem minden típusú fejlesztésnél alkalmazható.

Spirális modell: a **prototípus modell és a vízesés modell kombinációja**, olyan, mint egy általánosabb inkrementációs modell. **Nincsenek rögzített fázisai: a fejlesztés egy spirálal ábrázolható**, amely körei egy-egy fázist reprezentálnak, melyeknek kimenete egy "release". Ezeknek a köröknek a célja lehet például a megvalósíthatóság, a követelmények meghatározása, tervezés, javítás, karbantartás, stb. Általában ezek a körök 3-6 szektorra oszthatók.

WINWIN spirális modell: alapelve az, hogy a fejlesztők is, és a megrendelő is nyerjen. **Sok tárgyalást vezet be az eredeti spirális modellbe**, amelyek során az érdekeltek ismertetik nyerő feltételeiket, majd kompromisszumot kötnek.

V-modell: ez nem csak egy fejlesztési modell, hanem egy módszertan is egyben, amit a német védelmi minisztérium dolgozott ki. Elsődleges szempontja a biztonság, ám a benne lévő sorrendet nem szokták szigorúan venni (egyes fázisok kihagyhatók). **Követelmények meghatározása → kockázatelemzés → specifikálás → architektúrális tervezés → modul tervezés → modul előállítása, tesztelése → integrálás, tervezés → verifikálás → bizonyságlalás → üzemeltetés.** Látható, hogy sok tervezés, és sok ellenőrzés van a lépések között.

Tisztaszoba (cleanroom): egy nagyon magas színvonalú, hitelesíthető szoftverek fejlesztésénél használt eljárás (például banki alkalmazások). Sokszor a fejlesztők a fejlesztett rendszer nagyon kis töredékét látják, az egész applikáció olyan minimális részhalmazát, ami a fejlesztéshez éppen szükséges. A hangsúly a hibák kijavítása helyett azok megelőzésén van. Az egész csapat ellenőrzi, hogy a tervezés a helyes úton jár-e, miközben fokozatosan, iteratívan fejlesztenek, aminek haladását előre meghatározott szabványokban mérik. Ha egy bizonyos rész elkészül, akkor azt nagyjából mint egy statisztikai eljárást hajtják végre (sokkal jobban odafigyelnek a lefedettségi százalékokra, a tesztek minőségére, stb). Látható, hogy ez egy nagyon formális fejlesztési modell.

Rational Unified Process (RUP): a Rational Software Corporation fejlesztette, amit később az IBM felvásárolt. Több folyamatmodell ötvözete, legfőbb jellemzői: konkrét célkitűzéseknek (mire kell, mire nem kell figyelni) megfelelően komponensek iteratív fejlesztése, amelyet egy becsült idő alatt kell végrehajtani. Általában biztosítani szokták még a program magas minőségét, és ezt átadás utáni karbantartással / kezeléssel szokták alátámasztani.

SCRUM: egy nagyon fontos agilis projektmenedzsment módszertan. Egy termék elkészítését körülbelül kéthetes fázisokra (sprintekre) osztjuk, aminek elején és végén értekeznek a fejlesztők, hogy mit is sikerült megvalósítani, mit nem, és mi az, ami még hátra van. Ezen felül a SCRUM van még egy rövid, helyzetjelentő reggeli stand-up meeting is, ahol a "mit csináltam tegnap?", "mit csinálok ma?" és a "van-e valami, ami akadályoz a munkában?" kérdésekre adnak választ a többi fejlesztőnek. Ezeket a megbeszéléseket egy Scrum master vezeti. Rajta kívül (és a fejlesztőkön kívül) fontos megemlíteni még a terméktulajdonost (Product Owner), aki a kliens oldalán felel a fejlesztésért, az üzleti szereplőket (Stakeholders), akik a kliens oldalán támogatják a terméktulajdonost a döntéseiben, valamint a menedzsmentet (Managers), akik a fejlesztő csapat üzleti oldalát képviselik.

Kanban: legfontosabb eleme egy tábla (lehet fizikai vagy digitális), amelyen oszlopokba rendezve megtaláljuk a tennivalókat teendő (TODO), folyamatban lévő (IN PROGRESS) és kész (DONE) feladatakat. Ezek az oszlopok nyilván tetszés szerint tovább oszthatóak, általában esetben ezeket szoktuk megkülönböztetni. Ezt a táblát a fejlesztés során folyamatosan frissíteni kell, és így tudjuk jól reprezentálni a fejlesztés aktuális státuszát (a SCRUMBAN a SCRUM módszertan kanban táblás bővítése).

Összegzés

Szoftverfejlesztési folyamat

- fejlesztés = folyamat + menedzsment + technikai módszerek + eszközök használata
- nem csak a funkcionális és a teljesítmény fontos, hanem a megbízhatóság, jó használhatóság, karbantarthatóság, és a továbbfejleszthetőség is
- TERVEZÉSI FOLYAMAT:
 - terv (módosulhat, itt még olcsó)
 - megszorítások összegzése
 - paraméterek becslése
 - mérföldkövek és átadandók tervezése
 - ütemterv készítése
 - rögzítés, átadandók és határidők meghatározása
- ELEMEI:
 - feladatok, termékek (fő)
 - határidők, átadandók (fő)
 - projektmenedzsment (kiegészítő)
 - konfigurációmenedzsment (kiegészítő)
 - dokumentáció (kiegészítő)
 - minőségbiztosítás, kockázatelemzés (kiegészítő)
 - mérés (kiegészítő)
- FÁZISAI:
 - specifikáció (mit csinálunk)
 - fejlesztés (hogyan csináljuk)
 - validáció/verifikáció (jól csináltuk-e)
 - evolúció (karbantartás, felügyelés, "3RD level support")

Szoftverfejlesztési modellek

- VÍZESÉS MODELL
 - szekvenciális modell, a fázisok egymás után követik egymást, és nincs visszalépés
 - esetleges hibákat a fázisok legvégén javítják
 - nagyon jó specifikáció szükséges - kezdetben nehéz
 - a megrendelő a projekt végéig nem találkozik a termékkel: itt már drága a javítás
- ITERÁCIÓS (INKREMENTÁLIS) MODELL
 - a fejlesztési folyamat iterál
 - a specifikáció csak nagy vonalakban meghatározott
 - a végső követelmények a menet közben alakulnak ki
 - próbálunk a terméken minden iterációban javítani
 - egyes iterációkban tetszőlegesen választhatunk fejlesztési modellt
 - minden inkremens működésbe helyezhető
 - esetleges határidő csúszás esetén nem lesz oda az egész projekt, csak bizonyos inkremensek
 - az inkremensek meghatározása nem triviális feladat (túl kicsi: nem működőképes, túl nagy: elveszítjük a modell lényegét)
 - számos funkcionálitást kell megvalósítani, egész addig nincs működő inkremens

- **EVOLÚCIÓS (PROTOTÍPUSOS) MODELL**
 - az iterációs modell egy szélsőséges verziója
 - nincs részletes specifikáció
 - gyorsan fejlesztünk egy prototípust
 - a megrendelő azt kiértékeli, és újabb specifikációkat állít elő
 - intenzív kapcsolat szükséges hozzá a megrendelővel
 - nem feltétlenül a prototípusból lesz a végleges termék (van, hogy az egészet el kell dobni)
 - a prototípusok számát csökkentheti, ha olyan kész komponenseket használunk, amiket a megrendelő már elfogadott
 - PROTOTÍPUS ≠ KÉSZ RENDSZER!!!
 - gyors fejlesztés miatt a minőség romolhat
 - hibrid megoldások kellenek a vízesés modellel
- **EXTREME PROGRAMMING (XP)**
 - kis funkcionálisú inkremensek megvalósítása
 - megrendelő intenzív részvételle szükséges
 - csoportos fejlesztés (cooperative programming)
- **RAPID APPLICATION DEVELOPMENT (RAD)**
 - extrém rövid életciklus
 - gyakran a semmiből kell 60-90 nap alatt működő rendszert csinálni
 - "nagysebességű" vízesés modell
 - több csapat párhuzamosan fejleszt különböző komponenseket
 - nagy emberi erőforrásigény
 - intenzív együttműködés szükséges
 - nem minden alkalmazható
- **FÁZISAI**
 - Üzleti modellezés: milyen adatok áramlanak az egységek között
 - Adatmodellezés: hogyan reprezentáljuk azokat az adatokat adatszerkezettel
 - Adatfolyam processzus: adatmodell megvalósítása
 - Alkalmazás generálás: automatikus generálás, komponensek használata
 - Tesztelés: csak komponens tesztelés
- **SPIRÁLIS MODELL**
 - a prototípus modell és a vízesés modell kombinációja,
 - egy általánosabb inkrementációs modell
 - nincsenek rögzített fázisai: a fejlesztés egy spirállal ábrázolható, amely körei egy-egy fázist reprezentálnak
 - a körök kimenete egy "release",.. aminek a célja lehet például a megvalósítás, a követelmények meghatározása, tervezés, javítás, karbantartás, stb.
 - általában egy kör 3-6 szektorra osztható
- **WINWIN SPIRÁLIS MODELL**
 - WINWIN = mindenki nyer (megrendelő és fejlesztő is)
 - sok tárgyalás a felek között
 - nyerő feltételek ismertetése
 - kompromisszumkötés
- **V-MODELL**
 - egy szekvenciális modell, kötetlen lépésekkel (egyes fázisok kihagyhatók)
 - sook tervezés és ellenőrzés (verifikáció) van a folyamat keretein belül

- TISZTASZÓBA

- nagyon magas színvonalú fejlesztések során (banki alkalmazások pl)
- a fejlesztéshez a szükséges adatok legkisebb részhalmazát kapják meg a fejlesztők
- a problémák javítása helyett azok megelőzésén van a hangsúly
- fokozatosan ellenőrzött tervezés
- iteratív megvalósítás
- haladás szabványos mérése
- statisztikailag ellenőrzött tesztelés

- RATIONAL UNIFIED PROCESS (RAD):

- több folyamatmodell ötvözete
- konkrét célkitűzések (mire kell, mire nem kell figyelni)
- komponensek iteratív fejlesztése,
- jól megbecsült idő alatt
- a program magas minőségét biztosított
- átadás utáni karbantartás / hibák kezelése

Agilis szoftverfejlesztési modellek**- SCRUM**

- projektmenedzselési módszertan
- fejlesztés két hetes szakaszokra van osztva (sprintek)
- naponta meeting, ahol megbeszélik, mit csináltak, mit fognak, és van-e blokkoló
- sprintek elején és végén megbeszélés, hogy mit sikerült, mit nem sikerült megcsinálni, és mi van még hátra (retrospective, planning, grooming, stb)
- scrum master, product owner, team, stakeholders, management

- KANBAN

- tábla, amin követni tudjuk a feladatokat
- általában teendő (todo), folyamatban (in progress) és kész (done) oszlopok, amik tetszőlegesen tovább oszthatók
- SCRUMBAN: olyan scrum, amelyben használnak kanban táblát
- JIRA, Git issue tracker, és társai

Rendszerfejlesztés I - 2.)

Projektmenedzsment. Költségbecslés, szoftvermérés.

Projektmenedzsment

A projektmenedzsment alapvetően több részből áll:

- az emberek menedzselése
- minőségellenőrzés, és biztosítás
- folyamat továbbfejlesztése
- konfigurációkezelés
- rendszerépítés
- hibamenedzsment

Egy projekt esetleges sikertelenségének okai lehetnek:

- A szükséges ráfordítások alulbecslése
- Technikai nehézségek
- A projekt csapatban nem megfelelő a kommunikáció
- A projektmenedzsment hibái

Az emberek menedzselése: egy szoftverfejlesztő szervezet legnagyobb vagyonát az ott dolgozó emberek képezik. Sok projekt bukásának legfőbb oka lehet a rossz humánmenedzsment (kevés pozitív visszajelzés, a munkások igényeinek folytonos figyelmen kívül hagyása, alacsony fizetés / megbecsülés), ezért erre hatványozottan figyelni kell! Törekedni kell a csapaton belül a hatékony együttműködésre és az erős csapatszellel kialakítására, ugyanis bizonyított tény, hogy egy csapat sokkal jobban teljesít, ha közösen megoldandó problémaként tekint a feladatokra. Nagyon fontos továbbá a csapaton belül jó kommunikációt kialakítani, hogy az információ a lehető legjobban áradjon a tagok között.

A csapatot több szempont alapján építhetjük fel, ilyenek lehetnek a megoldandó probléma nehézsége, mérete, számít még az is, hogy a csoport mennyi ideig fog együttműködni (hosszabb időre nyilván sokkal fontosabb a csapat összeszoktatása). Fontos lehet a feladat modularizálhatósága, a minőségi és megbízhatósági követelmények, kommunikációs igény, határidők szigorúsága...

Minőségellenőrzés, és biztosítás: A projekt összes tagjának közös célja a termék (vagy szolgáltatás) magas szinten tartása. Ezalatt azt értjük, hogy azon felül, hogy a projekt megfelel a specifikációknak, az legyen karbantartható, áttekinthető (clean-code), továbbfejleszthető, és feleljen meg a fejlesztők és a megrendelő belső igényeinek. A gond viszont az, hogy egyes jellemzőket (mint például az átláthatóság) nem könnyű specifikálni, viszont sokszor mégis fontos a személyes preferenciáknak is megfelelni.

Folyamat továbbfejlesztése: CMM(I) (Capability Maturity Model (Integration)) a szoftver folyamat mérése, aminek célja a szoftverfejlesztési folyamat hatékonyságának valamilyen számokkal történő reprezentálása. Egy adott szervezet szervezet megkaphatja valamely szintű minősítését

Besorolási szintjei (nagyobb szintek esetén a kisebb szintek követelményei is teljesülnek):

- 1.) **Kezdeti:** csak néhány folyamat definiált, a többségük esetleges
- 2.) **Reprodukálható:** a projektmenedzsment alapvető folyamat definiált: költségütemezés, funkcionális kezelése (konfigurációs menedzsment, szoftverminőség biztosítása, alvállalkozói menedzsment, projekt tervezés (planning), követelmény menedzsment, projekt követés / felügyelet)
- 3.) **Definiált:** a menedzsment és a fejlesztés folyamatai is dokumentáltak és szabványosítottak az egész szervezetre. (alapos átnézések (review), csoportok közötti koordináció, termék menedzsment, integrált szoftver menedzsment, tréning programok)
- 4.) **Ellenőrzött:** a szoftver folyamat és termék minőségének részletes mérése, ellenőrzése.
- 5.) **Optimalizált:** a folyamatok folytonos javítása az új technológiák ellenőrzött bevezetésével

A szoftver folyamat javításán állandóan dolgozni kell, aminek alapvető célja a minőség és a hatékonyság növelése. Ehhez használhatunk metrikákat, hogy könnyebben tudjuk követni. Ehhez a részhez tartozik még a hibák analizálása (hibák eredetének kategorizálása, azok javítási költségének felmérése) is.

Konfigurációkezelés: a rendszer változásainak kezelése, és azok felügyelettel követése, a szoftverek evolúciója miatt kiemelkedő fontossággal bír. A minőség kezelésének, biztosításának szerves része. A szoftvereket különböző változatokban tudjuk értelmezni (verziók, kiadások / release, alapvonal / baseline, mainline, trunk, fejlesztési ágak / branch), amiket a megrendelő is nyomon tud követni. Ehhez ajánlott egy verziókövető rendszer használata, mint pl a Git vagy a BitBucket. Egy valamirevaló konfigurációs adatbázis minden tárol (forráskód, dokumentumok, építési folyamat, szkriptek, hibák adatbázisa, változások története, verziók), amit egy adott konfigurációról tudni kell.

Rendszerépítés: a kód struktúrája nyilván projektenként más, ezért a benne lévő komponensek fordítása, szerkesztése, és lokális futtatása nem minden triviális. Ezek a file-ok gyakran egymás közti építési függőségekkel rendelkeznek, ami nagyobb projektek esetében elég komplex szerkezetet is jelenthet. Érthető okokból a fordítás ilyen esetekben igen hosszú ideig is tarthat, ezért kell inkrementálisan végezni (apró inkremensek megvalósítása, majd ellenőrzés, és kezdjük előről). Sokszor ezt a folyamatot nem árt automatizálni, viszont nem minden, hogy milyen eszközzel. Szerencsére erre manapság már több lehetőségünk is van: nagyon sok IDE (Integrated Development Environment) rendelkezik beépített (akár alapértelmezett) futtatási konfigurációkkal, valamint léteznek CI (Continuous Integration) felületek is (például Jenkins), amik meghatározott szkripteket futtathnak egy előre konfigurált szerveren, ezáltal megbízható visszajelzést kaphatunk az írt kód fordíthatóságáról.

Hibamenedzsment: a termék fejlesztése során fontos a jelentkező hibák követése, mert sajnos sok lehet belőlük. Ezeket érdemes kategorizálni, prioritási sorba rendezni, majd követni, hogy mi történik velük a későbbiekben. A követésre készítenünk kell egy hibaadatbázist, ahol minden hibához egyedi azonosítót rendelünk, rögzítjük a bejelentő nevét, bejelentés idejét, és egy rövid összegzést magáról a hibáról. Ezen felül érdemes még feljegyezni a probléma súlyosságát (triviális, kicsi, nagy), az előidézéshez szükséges platformot, operációs rendszert, és ideális esetben magát a terméket is (lehetőleg verziószámmal). Érthető, hogy egy hiba más hibákat is eredményezhet, ilyen esetben ezt a kapcsolatot is jelezünk kell. Fontos a hiba életútját rögzíteni, hogy tudjuk jelenteni, ha annak javítása elkészült, vagy esetleg a későbbiekben meg tudunk magyarázni esetleges kellemetlenségeket.

Költségbecslés

Egy projekt költsége szoros kapcsolatban áll az ahhoz szükséges munk-, idő- és pénzköltséggel, amikre lehet, és kell is becslést adni. Egy tetszőleges projekt összköltségét megkaphatjuk úgy, hogy vesszük a hardveres és szoftveres költségeket, és azoknak karbantartásának költségét, valamint az utazási-, és képzési költségeket, és végezetül természetesen magát a munkaköltséget. Ha ezeket kiderítettük, tudunk becslést adni arra, hogy mennyi pénzre, mennyi rövidítésre és mennyi időre lenne szükség a projekt elkészítéséhez. Jórészt ezeket könnyen meg tudjuk becsülni, viszont a munka költségének kiszámítása nem feltétlenül triviális, ugyanis ebbe az értékbe beleszámoljuk a fejlesztők fizetését, az őket kisegítő személyzet fizetését, az iroda bérleti díját, az infrastruktúra (pl. hálózat) használatát, járulékokat, adókat... egyszóval **minent**, ami akármilyen közvetett, vagy közvetlen módon kapcsolatban áll a fejlesztéssel.

Többféle módszer áll rendelkezésünkre becsléshez:

- hasonló projektek alapján
- **egyszerű dekompozíciós technika**
 - **szoftver mérete alapján** (approximációs döntési lépések alapján, szabványos komponenesméretek használata - általában létező komponenseket módosítunk)
 - **probléma alapú becslés** (funkcióra bontás, "baseline" metrikák, LOC / Lines of Code becslés - dekompozíció funkcióra, FP / Function Point becslés - dekompozíció jellemzőkre)
 - **folyamat alapú becslés** (meghatározzuk a funkciókat és azok végrehajtandó feladatokat, majd becsüljük magukat a feladatokat).
- algoritmikus módszerek
- tapasztalati modellek
- szakértői vélemények
- Parkinson törvénye
- nyerő ár (a legnagyobb összeg, amit a vevő fizetni képes)

COCOMO (Constructive Cost Modell): objektumpontok segítségével számolja ki a költségeket az emberhónap és termelékenység függvényében.

Object type	Complexity weight		
	Simple	Medium	Difficult
Screen	1	2	3
Report	2	5	8
3GL component			10

Developer's experience/capability	Very low	Low	Nominal	High	Very high
Environment maturity/capability	Very low	Low	Nominal	High	Very high
PROD	4	7	13	25	50

Szoftvermérés

A termék vagy folyamat valamely jellemzőjét próbáljuk numerikusan kifejezni (metrika). A kapott értékekből következtetések vonhatók le a minőségre vonatkozóan. Alapvetően két nagy csoportra oszthatjuk őket: **vezérlési-** (folyamattal kapcsolatosak, pl. egy hiba javításához szükséges átlagos idő), **és prediktor** (termékkel kapcsolatosak, pl. LOC - Lines of Code, ciklomatikus komplexitás, osztály metódusainak száma) metrikák.

LOC = Lines Of Code mérésére például több technikát is alkalmazhatunk: számolhatjuk csak a nem üres sorokat vagy csak a végrehajtható sorokat is, mindenkorrel egy reprezentációt kapunk, ami nem feltétlenül releváns bizonyos esetekben, mivel nyilván assemblyben sokkal több sorból áll egy egyszerű kiíratás is, ami manapság egy fejlett programnyelvben egy sor.

A mérési folyamat elején fontos meghatároznunk, hogy milyen mérést is szeretnénk végezni, a termék melyik komponensén. Csak ezek előntése után érdemes megkezdeni magát a mérést - azaz a metrikák kiszámítását.

A termék metrikáit a következő csoportokba osztjuk:

- Dinamikus: szoros kapcsolat a minőségi jellemzőkkel (teljesítmény, hibák száma)
- Statikus: közvetett kapcsolat, sok konkrét metrikát javasoltak már (méret, komplexitás, kohézió / összetartozás, objektumorientáltsággal kapcsolatos metrikák)

Méret alapú metrikák: sokan használják őket, de alkalmazásuk és helyességük körül sok vita van

- Hibák / KLOC (KLOC = 1000 Lines of Code)
- Defekt / KLOC
- Költség / LOC
- Dokumentációs oldalak / KLOC
- Hibák / emberhónap
- LOC / emberhónap
- Költség / dokumentációs oldal

Funkció alapú metrikák:

- Felhasználói inputok száma (alkalmazáshoz szükséges adatok)
- Felhasználói outputok szám riportok, képernyőképek, hibaüzenetek
- Felhasználói kérdések száma - on-line input és output
- Fájlok száma- adatok logikai csoportja

3D mérték:

- Számítás: $\text{Index} = I + O + Q + F + E + T + R$ ($I=\text{input}$, $O=\text{output}$, $Q=\text{lekérdezés}$, $F=\text{fájlok}$, $E=\text{külső interfész}$, $T=\text{transzformáció}$, $R=\text{átmenetek}$)

Minőség mérése:

- **Integritás:** külső támadások elleni védelemű
- **Fényegetettség:** annak valószínűsége, hogy egy adott típusú támadás bekövetkezik egy adott időszakban
- **Biztonság:** annak valószínűsége, hogy egy adott típusú támadást visszaver a rendszer
- **Integritás** = $\Sigma [1 - (\text{fenyegetettség} \times (1 - \text{biztonság}))]$ (Összegzés a különböző támadás típusokra történik)
- DRE (defect removal efficiency), $DRE = E/(E+D)$, ahol E olyan hibák száma, amelyeket még az átadás előtt felfedezünk, D pedig az átadás után a felhasználó által észlelt hiányosságok száma

Összegzés

Projektmenedzsment

- **EMBEREK MENEDZSELÉSE**
 - egy szoftverfejlesztő szervezet legnagyobb vagyona az ott dolgozó emberek
 - a projekt bukásának egyik oka lehet a rossz humánmenedzsment
 - fontos a hatékony együttműködés és kommunikáció kialakítása
- **MINŐSÉGELLENŐRZÉS, ÉS BIZTOSÍTÁS**
 - a projekt összes tagjának közös célja a termék magas szinten tartása
 - azon felül, hogy a projekt megfelel a specifikációknak, feleljen meg a fejlesztők és a megrendelő belső igényeinek is
 - karbantarthatóság, áttekinthetőség (clean-code), továbbfejleszthetőség
- **FOLYAMAT TOVÁBBFEJLESZTÉSE**
 - CMM(I) (Capability Maturity Model (Integration)) célja a szoftverfejlesztési folyamat hatékonyságának valamelyen reprezentálása
 - **Besorolási szintek**
 - **Kezdeti:** csak néhány folyamat definiált, a többségük esetleges
 - **Reprodukálható:** a projektmenedzsment alapvető folyamat definiált: költségütemezés, funkcionális kezelése
 - **Definiált:** a menedzsment és a fejlesztés folyamatai dokumentáltak, szabványosítottak
 - **Ellenőrzött:** a termék minőségének részletes mérése, ellenőrzése.
 - **Optimalizált:** folytonos javítás, új technológiák ellenőrzött bevezetése
 - szoftver folyamat javításán állandóan dolgozni kell
 - célja a minőség és a hatékonyság növelése
 - használunk metrikákat
 - hibák analizálása (hibák eredetének kategorizálása, azok javítási költségének felmérése)
- **KONFIGURÁCIÓKEZELÉS**
 - a rendszer változásainak kezelése, és azok felügyelettel követése
 - a szoftverek evolúciója miatt fontos.
 - minőségbiztosítás része
 - szoftverek változatai: verziók, kiadások / release, alapvonal / baseline, mainline, trunk, fejlesztési ágak / branch
 - verziókövető rendszer használata, mint pl a Git vagy a BitBucket
 - konfigurációs adatbázis minden tárol: forráskód, dokumentumok, építési folyamat, szkriptek, hibák adatbázisa, változások története, verziók
- **RENDSZERÉPÍTÉS**
 - a projekt komponenseinek fordítása, szerkesztése, és lokális futtatása nem minden triviális
 - gyakran építési függőségek vannak a fileok között, ami nagyobb projektek esetében elég komplex szerkezetet is jelenthet
 - nagy projektek esetén a fordítás igen hosszú ideig is tarthat
 - inkrementálisan kell a fejlesztést végezni (apró inkremensek megvalósítása, majd ellenőrzés, és kezdjük előről)
 - nem árt automatizálni ezt a folyamatot, viszont nem mindegy, hogy milyen eszközök használunk

- nagyon sok IDE (Integrated Development Environment) rendelkezik beépített futtatási konfigurációkkal,
- CI (Continuous Integration) felületek (Jenkins), meghatározott szkripteket futtatnak egy előre konfigurált szerveren
- **HIBAMENEDZSMENT**
 - fontos a jelentkező hibák követése
 - sok hiba lehet
 - érdemes kategorizálni, prioritási sorba rendezni, majd követni, hogy mi történik velük a későbbiekben.
 - hibaadatbázis: minden hibához egyedi azonosító, bejelentő neve, bejelentés ideje, rövid összegzés, probléma súlyossága (triviális, kicsi, nagy), az előidézéshez szükséges platform, operációs rendszer, termék (verziószámmal).
 - más hibákkal kapcsolatban állás esetén a kapcsolatot is jelezünk kell.
 - a hiba életútját rögzíteni kell

Költségbecslés

- szoros kapcsolatban áll a munka-, idő- és pénzköltséggel,
- lehet, és kell is becslést adni
- az összköltség meghatározható a hardveres és szoftveres költségek, karbantartási költségek, utazási-, és képzési költségek, és a munkaköltség összegeként
- meg kell becsülni, hogy mennyi pénzre, mennyi ráfordításra és mennyi időre lenne szükség
- a munka költségének kiszámítása nem feltétlenül triviális
- ebbe az értékbe beleszámolunk **mindent**, ami akármilyen közvetett, vagy közvetlen módon kapcsolatban áll a fejlesztéssel.
- **COCOMO** (Constructive Cost Modell), objektumpontok segítségével számolja ki a költségeket az emberhónap és termelékenység függvényében

Szoftvermérés

- a termék vagy folyamat valamely jellemzőjét próbáljuk numerikusan kifejezni (metrika).
- a kapott értékekkel következtetések vonhatók le a minőségre vonatkozóan.
- vezérlési- (folyamattal kapcsolatos), és prediktor (termékkel kapcsolatos) metrikák.
- LOC = Lines Of Code mérése több technikával: nem üres sorok, végrehajtható sorok
- nem feltétlenül releváns nem összehasonlítható programozási nyelvek.
- meg kell határozni, hogy milyen mérést végzünk a termék melyik komponensén
- **DINAMIKUS METRIKÁK**
 - szoros kapcsolat a minőségi jellemzőkkel
 - teljesítmény, hibák száma
- **STATIKUS METRIKÁK:**
 - közvetett kapcsolat,
 - sok metrikát javasoltak már
 - méret, komplexitás, kohézió, objektumorientáltsággal kapcsolatos metrikák
- **Méret alapú metrikák:** alkalmazásuk kérdéses
 - Hibák / KLOC (KLOC = 1000 Lines of Code)
 - Defekt / KLOC
 - Költség / LOC
 - Dokumentációs oldalak / KLOC
 - Hibák / emberhónap
 - LOC / emberhónap
 - Költség / dokumentációs oldal

- **Funkció alapú metrikák:**
 - Felhasználói inputok száma (alkalmazáshoz szükséges adatok)
 - Felhasználói outputok szám riportok, képernyőképek, hibaüzenetek
 - Felhasználói kérdések száma - on-line input és output
 - Fájlok száma- adatok logikai csoportja
- **3D mérték:**
 - $\text{Index} = I + O + Q + F + E + T + R$, ahol I=input, O=output, Q=lekérdezés, F=fájlok, E=külső interfész, T=transzformáció, R=átmenetek
- **Minőség mérése:**
 - Integritás: külső támadások elleni védelemű
 - Fenyegetettség: annak valószínűsége, hogy egy adott típusú támadás bekövetkezik egy adott időszakban
 - Biztonság: annak valószínűsége, hogy egy adott típusú támadást visszaver a rendszer
 - Integritás = $\Sigma [1 - (\text{fenyegetettség} \times (1 - \text{biztonság}))]$ (Összegzés a különböző támadás típusokra történik)
 - DRE (defect removal efficiency), DRE = $E/(E+D)$, ahol E az átadás előtt felfedezett hibák száma, D az átadás után a felhasználó által észlelt hiányosságok száma

Számítógép-hálózatok - 1.)

Számítógép-hálózati architektúrák, szabványosítók (ISO/OSI, Internet, ITU, IEEE).

Számítógép-hálózati architektúrák, szabványosítók

Számos hálózatépítéssel és hálózatüzemeltetéssel foglalkozó cég létezik, és mindegyiknek saját elképzelése van arról, hogy hogyan kell a dolgokat csinálni. Koordináció nélkül teljes káosz uralkodna, és a felhasználók semmihez nem jutnának hozzá. Az egyetlen kiút a hálózatok szabványosítása.

A szabványok nemcsak a különböző számítógépek közötti kommunikációt teszik lehetővé, hanem bővílik a szabványhoz kapcsolódó termékek piacát is, ami végül tömegtermeléshez, gazdaságosabb gyártáshoz, és további olyan előnyökkel jár, amelyek az árakat csökkentik, a szabvány elfogadhatóságát pedig növelik.

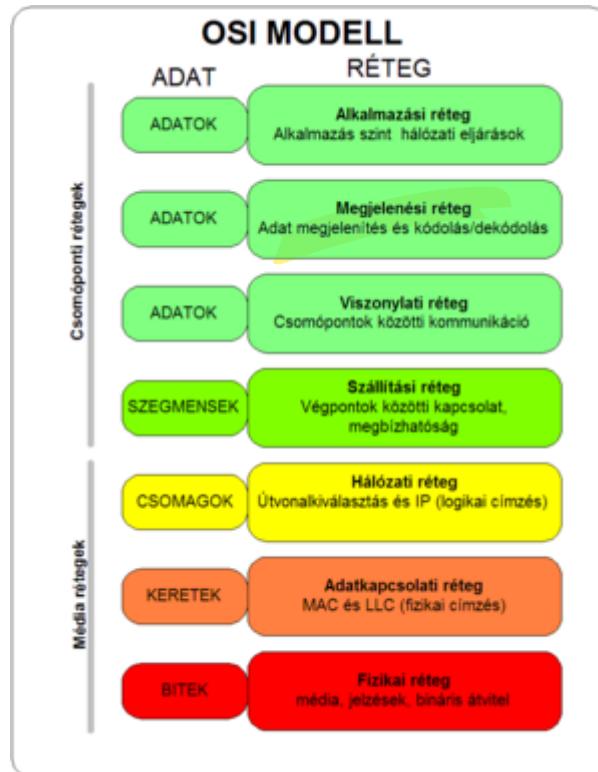
A **de facto** (latinul „tényleges”) szabványok azok a szabványok, amelyek **hivatalos leírás nélkül maguktól** alakultak ki. Az IBM PC és leszármazottai például a személyi számítógépek de facto szabványa, ugyanis több tucat gyártó kezdte el igencsak hűen másolni az IBM gépeit.

A **de jure** (latinul „törvényes”) szabványok ezzel szemben **olyan hivatalos szabványok, amelyeket bizonyos szabványosítási szervezetek elfogadtak**. A nemzetközi szabványosítási szervezeteket általában két nagy csoportra oszthatjuk. Az egyik csoportba azok tartoznak, amelyek **államközi szerződések** útján jöttek létre, a másikba pedig az önkéntesen, nem szerződéses alapon létrehozott szervezetek. A számítógép hálózatok szabványosításának világában minden típusú szervezetből van jó néhány.

ISO/OSI: A nemzetközi szabványokat a Nemzetközi Szabványügyi Szervezet (**International Standards Organization, ISO**) adja ki, amely egy 1946ban alakult, önkéntes, **nem államközi szerződéseken alapuló szervezet**. Az ISO tagságát 89 tagállam nemzeti szabványügyi szervezete alkotja. A tagok között megtalálható az ANSI (Egyesült Államok), a BSI (Nagy Britannia), az AFNOR (Franciaország), a DIN (Németország) és még további 85 szervezet.

Az ISO a legkülönbözőbb témaikban ad ki szabványokat, a csavaroktól és a csavaranyáktól (szó szerint) kezdve a telefonpóznák bevonatáig minden ideértve. Több mint 5000 szabványt adtak eddig ki, beleértve az **OSI** szabványokat is. Az ISOnak közel 200 Műszaki Bizottsága (Technical Committee) van, amelyeket a megalakulásuk sorrendjében számoztak be. Ezek mindegyike külön szakterülettel foglalkozik. A Műszaki Bizottságok albizottságokra (subcommittee, SC), azok pedig munkacsoportokra (working group, WG) vannak felosztva.

Az Open Systems Interconnection (OSI) Reference Model (Nyílt rendszerek összekapcsolása referenciamodellel) egy rétegekbe szervezett rendszer absztrakt leírása, amely a számítógépek kommunikációjához szükséges hálózati protokollt határozza meg. Gyakran az OSI hétrétegű modellje néven is emlegetik.



Fizikai réteg: Feladata, hogy továbbítja a biteket a kommunikációs csatornán. A rétegnak biztosítania kell, hogy az egyik oldalon elküldött 1-es a másik oldalon is 1-esként érkezzenek meg. Ez a réteg tipikusan olyan kérdésekkel foglalkozik, hogy mekkora feszültséget kell használni a logikai 0 és mekkorát a logikai 1 reprezentálásához, mennyi ideig tart egy bit továbbítása, megvalósítható-e az átvitel minden oldalon, miként jön létre, stb.

Adatkapsolati réteg: Fő feladata, hogy a fizikai réteg szerény adottságait egy olyan vonallá alakítsa, amely a hálózati réteg számára felderítetlen. Ezt úgy oldja meg, hogy az átvendő adatokat a küldő fél oldalán adatkeretekbe tördeli, és ezeket sorrendben továbbítja. Ha a szolgáltatás megbízható a fogadó fél egy nyugtató kerettel nyugtázza minden egyes keret helyes vételét. Forgalomszabályozást és hibakezelést is ez a réteg valósítja meg.

Hálózati réteg: Az alhálózat működését irányítja. A legfontosabb kérdés itt az, hogy milyen útvonalon kell a csomagokat a forrástól a célig eljuttatni. Az útvonalak meghatározása történhet statikus táblázatok felhasználásával, vagy dinamikus, amikor ugyanis minden csomag számára egyenként kerül kijelölésre az útvonal.

Szállítási réteg: Feladatai közé tartozik például a forgalomszabályozás, hibajavítás, multiplexelés. Megbízhatóság: csomagok elveszhetnek hálózati feltörődés vagy hibák miatt. Egy hibajavító kód segítségével, mint például az ellenőrző összeg, a szállítási protokoll ellenőrizheti, hogy az adat sérült-e, és ezt megerősítheti egy ACK vagy NACK üzenet küldésével a küldőnek.

Viszony réteg: A réteg lehetővé teszi, hogy két számítógép felhasználói kapcsolatot létesítsenek egymással. A viszony réteg segítségével egy felhasználó állományokat mozgathat számítógépek között. Jellegzetes feladata a logikai kapcsolat felépítése és bontása, párbeszéd szervezése.

Megjelenítési réteg: Az átvitt információ szintaktikájával illetve szemantikájával foglalkozik. Annak érdekében, hogy a különböző adatábrázolást használó gépek kommunikálni tudjanak, a párbeszéd során használt adatszerkezeteket és szabványos kódolást absztrakt módon kell definiálni. Ez a réteg ezekkel az absztrakt adatszerkezetekkel foglalkozik. Lehetővé teszi a magasabb szintű adat-szerkezetek definiálását és átvitelét.

Alkalmazási réteg: Protokollok sokasága, mint például a HTTP. További alkalmazási protokollok léteznek adatátvitelre (FTP), e-levelezésre (SMTP).

Internet: Az Internet az összekapcsolt számítógépes hálózatok globális rendszere, amely az Internet Protocol Suite (TCP / IP) protokollt használja a hálózatok és az eszközök közötti kommunikációhoz. Ez hálózatok olyan hálózata, amely magán-, állami, tudományos, üzleti és kormányzati hálózatokból áll, lokális és globális kiterjedésűek, összekapcsolva őket az elektronikus, vezeték nélküli és optikai hálózati technológiák széles skálájával. Ezek a hálózatok közös protokollokat használnak és közös szolgáltatásokat nyújtanak.

Az internetnek nincs egyetlen központosított irányítása sem a technológiai megvalósításban, sem a hozzáférésre és használatra vonatkozó politikákban. Ez egy nem szokványos rendszer abból a szempontból, hogy senki sem tervezte meg, és senki sem felügyeli.

Az elődje, az ARPANET (az első szélesebb körben kiépített nagy kiterjedésű csomag-kapcsolt hálózat), eredetileg a regionális tudományos és katonai hálózatok összekapcsolásának gerincét szolgáltatta a 70'-es években. Miután 1983. január 1-jén a TCP/IP lett az egyetlen hivatalos protokoll az ARPANET-en, a hozzá csatlakozó hálózatok, gépek és felhasználók száma gyorsan nőtt. Amikor a NSFNET-et és az ARPANET-et összekapcsolták, a növekedés sebessége exponenciálissá vált. Sok területi hálózat csatlakozott és kapcsolatok épültek Kanada, Európa és különféle csendes-óceáni szigetek hálózatai felé is. Valamikor az 1980-as évek közepén az emberek kezdtek úgy tekinteni erre a hálózatcsoportra mint egy internetre, majd később mint az Internetre. A mi definíciót az, hogy egy gép akkor van rajta az Interneten, ha az a **TCP/IP-protokollkészletet használja, rendelkezik saját IP-címmel és tud más gépeknek IP-csomagokat küldeni az Interneten át** (legfőbb alkalmazásai az E-levél, hírek, távoli bejelentkezés és a fájlküldés volt).

A modem egy kártya a számítógépben, amely a számítógép által előállított digitális jeleket analóg jelekké alakítja annak érdekében, hogy azok akadály nélkül átmehessenek a telefonhálózaton. Ezeket a jeleket az ISP POP-jéhez (Point of Presence, szolgáltatási pont) továbbítják, ahol ezeket a jeleket eltávolítják a telefonhálózatból, és az ISP (Internet Service Provider) területi hálózatára játsszák át. Ettől a ponttól a rendszer teljesen digitális és csomagkapcsolt. Amennyiben az ISP nem más, mint a helyi telefontársaság, a POP valószínűleg abban a telefonközpontban található, ahová a felhasználó telefon vezetéke befut. Ha az ISP nem a helyi telefontársaság, akkor a POP néhány kapcsoló központtal arrébb is lehet.

Az ISP területi hálózata az ISP által kiszolgált különféle városokban elhelyezett, egymással összekötött routerekből áll. Ha a csomag célja egy olyan hoszt, amelyet ugyanaz az ISP szolgál ki, akkor a csomag egyenesen a cél hoszthoz kerül. Egyéb esetekben a csomagot az ISP a saját gerinchálózati szolgáltatójának adja át.

Az, hogy egy számítógép képes az elektronikus levelek küldésére és fogadására még nem elégges, hiszen az e-levelek sok Interneten kívüli hálózatba is eljutnak az átjárókon keresztül. Ez a kép már azért sem teljesen tiszta, mert modernen keresztül több millió személyi számítógép tudja feltárcsázni internetszolgáltatóját, ahol ezek ideiglenes IP címet kapnak, és IP-csomagokat küldenek más, Interneten lévő hosztoknak. A józan ész azt diktálja, hogy ezeket a számítógépeket is Internetre kapcsolt gépeknek tekintsük arra az időre, amíg a szolgáltató routerével kapcsolatban állnak.

ITU: Nyilvánvalóan szükség van világméretű kompatibilitásra annak érdekében, hogy a különböző országokban élő emberek (illetve számítógépek) kapcsolatba kerülhessenek egymással. Ez az igény tulajdonképpen már régóta létezik. Az ITU feladata az volt, hogy szabványosítsa a nemzetközi távközlést, amely akkoriban még csak a távírást jelentette, azonban már akkor is nyilvánvaló volt, hogy problémához fog vezetni az, ha az országok egyik fele a Morse-kódot használja, a másik fele meg valami mást. Amikor a telefon nemzetközi szintű szolgáltatássá vált, az ITU magára vállalta a telefonrendszer szabványosítását is. Három fő ágazata van:

- Rádiókommunikációs ágazat (ITU-R).
- Távközlési szabványosítási ágazat (ITU-T)
- Fejlesztési ágazat (ITU-D)

ITU-R: Az 1927-ben Nemzetközi Rádió Tanácsadó Bizottság vagy CCIR néven (francia névén Comité consultatif international pour la radio) alapított ágazat kezeli a nemzetközi rádiófrekvenciás spektrum- és műholdpálya-erőforrásokat. 1992-ben a CCIR lett az ITU-R. Feladata a rádiófrekvenciák kiosztása a világszerte egymással versengő csoportoknak

ITU-T: A szabványosítás az ITU kezdetektől fogva eredeti célja volt. Az 1956-ban Nemzetközi Telefon- és Távirati Tanácsadó Bizottságként vagy CCITT-ként (Comité consultatif international téléphonique et télégraphique) elnevezéssel létrehozott ágazat egységesíti a globális távközlést (a rádió kivételével).

Az ITU-T feladata az, hogy műszaki javaslatokat tegyen a telefonok, a távírók és az adatkommunikáció interfészeire. Ezek gyakran válnak nemzetközileg elfogadott szabványokká. Fontos megjegyezni, hogy az ITU-T ajánlásai csak műszaki javaslatokat tartalmaznak. Az, hogy azokat egy ország elfogadja-e vagy sem, csak az adott országon múlik.

ITU-D: Az 1992-ben létrehozott ágazat hozzájárul az információs és kommunikációs technológiákhoz (IKT) való igazságos, fenntartható és megfizethető hozzáférés terjesztéséhez

IEEE: A szabványosítás világának egy másik fontos szereplője az IEEE (Institute of Electrical and Electronics Engineers, Villamos- és Elektronikai Mérnökök Intézete), amely a világ legnagyobb szakmai szervezete. Azonkívül, hogy rengeteg folyóiratot ad ki, és több száz konferenciát rendez meg évente, IEEE szabványokat is dolgoz ki a villamosmérnöki tudományok és az informatika területén.

Az IEEE 802-es bizottsága sok LAN-fajtát szabványosított. A sikeres munkacsoportok aránya a megalakulás óta alacsony; egy 802.x szám kiosztása még nem garantálja a sikert is. De a sikertörténetek (főként a 802.3 - ethernet és a 802.11 - wireless LAN) hatása óriási volt. Céljai az elektromos és elektronikai mérnöki, telekommunikációs, számítógépes mérnöki és rokon tudományágak oktatási és műszaki fejlődése.

Összegzés

Számítógép-hálózati architektúrák

- sok hálózatépítéssel és üzemeltetéssel foglalkozó cég létezik, mindegyiknek saját elköpzelése van arról, hogy hogyan kell a dolgokat csinálni
- ezeket koordinálni kell, amire a megoldás a hálózatok szabványosítása
- több szabványhoz tartozó termék növekvő piaci értéket jelent, és elfogadottabb szabványt
- **de facto** (tényleges): hivatalos leírás nélkül, maguktól alakultak ki (IBM PC)
- **de jure** (törvényes): hivatalos szabványok, melyeket szabványosítási szervek elfogadtak.
- a nemzetközi szabványosítási szervezeteket két csoportra oszthatjuk: államközi szerződések útján létrejöttek, és az önkéntesen, nem szerződéses alapon létrehozott szervezetek

ISO szervezet és szabványai

- a nemzetközi szabványokat a Nemzetközi Szabványügyi Szervezet (International Standards Organization, ISO) adja ki,
- önkéntes, nem államközi szerződésekben alapuló szervezet
- 89 tagállam alkotja (pl Egyesült Államok, Nagy Britannia, Franciaország, Németország)
- Az ISO a legkülönbözőbb témákban ad ki szabványokat a csavaroktól kezdve a telefonpóznák bevonatáig.
- több mint 5000 kiadott szabvány, beleértve az OSI szabványokat is.
- közel 200 Műszaki Bizottság (Technical Committee), ezek mindegyike, melyeket albizottságokra (subcommittee, SC), azokat pedig munkacsoportokra (working group, WG) osztottak.

Az OSI modell

- Az Open Systems Interconnection (OSI) Reference Model (Nyílt rendszerek összekapcsolása referenciamodellje) egy rétegekbe szervezett rendszer absztrakt leírása
- számítógépek kommunikációjához szükséges hálózati protokollt határozza meg
- összesen 7 réteg van
- **Fizikai réteg:** bitek továbbítása
- **Adatkapcsolati réteg:** bitek adatkeretekbe tördelése a titkosítás érdekében (packetek)
- **Hálózati réteg:** az alhálózat működését irányítja, feladata az útvonal meghatározása. Ez lehet statikus (minden csomag ugyanazon az úton megy végig), vagy dinamikus (minden csomag részére saját utat jelöl ki)
- **Szállítási réteg:** feladatai a forgalomszabályozás, hibajavítás és multiplexelés. Néha a csomagok elveszhetnek pl hálózati feltorlódás miatt, amit egy hibajavító kód segítségével (ellenőrző összeg) ellenőrizhetünk, és sérült adat esetén ezt jelzi NACK üzenet küldésével (különben ACK).
- **Viszony réteg:** lehetővé teszi, hogy két számítógép felhasználói kapcsolatot létesítsen. Segítségével egy felhasználó állományokat mozgathat számítógépek között. Jellegzetes feladata a logikai kapcsolat felépítése és bontása
- **Megjelenítési réteg:** az átvitt információ szintaktikájával illetve szemantikájával foglalkozik. Lehetővé teszi a magasabb szintű adatszerkezetek definiálását és átvitelét.
- **Alkalmazási réteg:** protokollok sokasága (HTTP, WebSocket)

Internet

- összekapcsolt számítógépes hálózatok globális rendszere, (TCP / IP) protokollt használja
- magán-, állami, tudományos, üzleti és kormányzati hálózatokból áll,
- lokális és globális kiterjedésűek
- közös protokollokat használnak és közös szolgáltatásokat nyújtanak
- nincs központosított irányítása
- nem szokványos rendszer, senki sem tervezte meg, és senki sem felügyeli
- elődje, az ARPANET (szélesebb körben kiépített csomag-kapcsolt hálózat),
- eredetileg a regionális tudományos és katonai hálózatok összekapcsolására
- mikor az NSFNET-et és az ARPANET-et összekapcsolták, felgyorsult a növekedés
- definíció szerint egy gép akkor van az Interneten, ha az a TCP/IP-protokollkészletet használja, rendelkezik saját IP-címmel és tud más gépeknek IP-csomagokat küldeni
- legfőbb alkalmazásai az E-levél, hírek, távoli bejelentkezés és a fájlküldés
- a modem egy kártya a számítógépben, amely a számítógép által előállított digitális jeleket analóg jelekké alakítja, hogy azok akadály nélkül átmehessenek a telefonhálózaton
- a jeleket az ISP (Internet Service Provider) POP-jéhez (szolgáltatási pont) továbbítják,
- ott ezeket a jeleket eltávolítják a telefonhálózatból, és a területi hálózatára játszik át
- ettől a ponttól a rendszer teljesen digitális és csomagkapcsolt

ITU

- elsődleges feladata az volt, hogy szabványosítsa a nemzetközi távközlést
- már akkor is nyilvánvaló volt, hogy problémához fog vezetni az, ha az országok egyik fele a Morse-kódot használja, a másik fele meg valami mást
- amikor a telefon nemzetközi szintű szolgáltatássá vált, az ITU magára vállalta a telefonrendserek szabványosítását is.
- RÁDIÓKOMMUNIKÁCIÓS ÁGAZAT (ITU-R)
 - Nemzetközi Rádió Tanácsadó Bizottság (CCIR) utódja
 - kezeli a nemzetközi rádiófrekvenciás spektrum- és műholdpálya-erőforrásokat
 - feladata a rádiófrekvenciák kiosztása az egymással versengő csoportoknak
- TÁVKÖLÉSI SZABVÁNYOSÍTÁSI ÁGAZAT (ITU-T)
 - Nemzetközi Telefon- és Távirati Tanácsadó Bizottság (CCITT)
 - eredeti célja a szabványosítás
 - egységesít a globális távközlést
 - műszaki javaslatokat tesz a telefonok, távírók és az adatkommunikáció interfészeire
 - az országokon múlik, hogy ezeket a javaslatokat elfogadják-e
- FEJLESZTÉSI ÁGAZAT (ITU-D)
 - hozzájárul az információs és kommunikációs technológiákhoz (IKT) való igazságos, fenntartható és megfizethető hozzáférés terjesztéséhez

IEEE

- Institute of Electrical and Electronics Engineers,
- Villamos- és Elektronikai Mérnökök Intézete
- a világ legnagyobb szakmai szervezete
- IEEE szabványokat dolgoz ki a villamosmérnöki tudományok és az informatika területén
- IEEE 802-es bizottsága sok LAN-fajtát szabványosított
- 802.3 - ethernet
- 802.11 - wireless LAN
- célja az elektromos és elektronikai mérnöki, telekommunikációs, számítógépes mérnöki és rokon tudományágak oktatási és műszaki fejlődése

Számítógép-hálózatok - 2.)

Kiemelt fontosságú kommunikációs protokollok (PPP, Ethernet, IP, TCP, HTTP, RSA).

Kiemelt fontosságú kommunikációs protokollok

PPP (Point-to-Point Protocol): egy olyan magasszintű, többprotokolos keretezési eljárás, amely alkalmas modem, HDLC bit-soros vonal és más fizikai rétegek feletti használatra. Támogatja a hibajelzést, fejléctömörítést, és akár a megbízható átvitelt is.

Szolgáltatásai: Olyan keretezési módszert vezet be, mely egyértelműen ábrázolja a keret végét és a következő keret elejét. A keretformátum egyúttal megoldja a hibajelzést is. Adatkapcsolat-vezérlő protokollt tartalmaz (LCP - Link Control Protocol) a vonalak felélesztésére, tesztelésére és bontására. Különböző hálózati vezérlő protokollokat (NCP - Network Control Protocol) tartalmaz mindegyik támogatott hálózati réteghez.

Keretformátuma: kialakításakor szempont volt, hogy minél jobban hasonlítsa a HDLC keretformátumra, viszont ellentétben a HDLC-vel, a PPP karakteralapú és nem bitalapú. minden PPP keret a szabványos **jelző** bajttal (01111110) kezdődik. Ezután következik a **Cím** mező, mely minden PPP keretben az 11111111 értékre van állítva annak jelzésére, hogy minden állomásnak vennie kell a keretet. A harmadik mező a **Vezérlő** mező, mely alapesetben a 00000011 értékre van állítva, mely a számosztlan keret jelzésére szolgál. Alapesetben a PPP nem biztosít megbízható átvitelt, de zajos környezetben (pl. vezeték nélküli hálózat) a megbízható átvitel megoldható a számosztott mód használatával (sorszámok és nyugták alkalmazásával). A következő mező a **Protokoll** mező, mely azt jelzi, milyen csomag van az adat mezőben (pl. LCP, NCP, IP, IPX, stb.). Következik az **Adat** mező, mely változó hosszúságú, de alapértelmezésként maximum 1500 bajt. És végül jön az **Ellenőrző** összeg mező, mely általában 2 bajt, de lehet 4 is.

Frame

Flag	Address	Control	Protocol	Data	FCS
1 byte	1 byte	1 byte	2 bytes	variable	2 or 4 bytes

Ethernet: Az Ethernet egy számítógépes hálózati technológiák családja, amelyet helyi hálózatban (LAN), városi hálózatokban (MAN) és nagy kiterjedésű hálózatokban (WAN) használnak. Először 1983-ban szabványosították IEEE 802.3 néven. Az Ethernet-et azóta finomították, hogy támogassa a nagyobb bitsebességet, a nagyobb csomópontok számát és a nagyobb összeköttetési távolságokat. Kábelezés:

Table : Types of 10 Mbps Ethernets								
Name	Cable	Max. segment	Nodes /segment	Signaling technique	Topology used	Cable diameter	Access Method	advantage
10Base5	Thick coax	500 m	100	Baseband (Manchester)	Bus	10	CSMA/CD	Good For back bones
10Base2	Thin coax	200 m	30	Baseband (Manchester)	Bus	5	CSMA/CD	Cheapest system
10Base-T	Twisted pair	100 m	1024	Based band (Manchester)	Star	0.4 to 0.6	CSMA/CD	Easy maintenance
10Base-F	Fiber optics	2000m	1024	Manchester/on-off	Star	62.5/125 μ m	CSMA/CD	Best between buildings

IP (Internet Protocol):

TCP (Transmission Control Protocol):

HTTP (HyperText Transfer Protocol):

RSA (Rivest–Shamir–Adleman) titkosítás:

Összegzés

Számítógép architektúra - 1.)

Neumann-elvű gép egységei. CPU, adatút, utasítás-végrehajtás, utasítás- és processzorszintű párhuzamosság. Korszerű számítógépek tervezési elvei. Példák RISC (UltraSPARC) és CISC (Pentium 4) architektúrákra, jellemzőik.

Neumann-elvű gép egységei

Neumann-architektúra mára a tárolt programú számítógép fogalmává vált. A számítógép működését tárolt program vezérli (Turing), amit vezérlés-folyam (control-flow) segítségével lehet leírni. Az aritmetikai és logikai műveletek (programutasítások) végrehajtását önálló részegység (ALU) végzi. 2-es (bináris) számrendszer alkalmaznak, és öt fő funkcionális egységből áll:

- **központi memória:** a program kódját és adatait tárolja számokként
- **központi feldolgozóegység (CPU):** a központi memoriában tárolt program utasításait beolvassa és végrehajtja
- **külső sín:** a részegységeket köti össze, adatokat, címeket, vezérlőjeleket továbbít
- **belső sín:** CPU részegységei közötti kommunikációt hozza létre (vezérlőegység (ALU) regiszterek)
- **beviteli/kiviteli eszközök:** kapcsolatot teremt a felhasználóval, adatot tárol a háttértáron, nyomtat, stb.
- **működést biztosító járulékos eszközök:** például gépház, tápellátás, hűtés...

A vezérlőegység (CU, control unit) utasításokat olvas be a memóriából, és vezérli az ALU és a regiszterek működését.

Az aritmetikai-logikai egység (ALU, Arithmetic Logic Unit) Egy tipikus Neumann-féle CPU belső szerkezetének részében az ALU végzi az összeadást, a kivonást és más egyszerű (aritmetikai) műveleteket az inputjain, így adva át az eredményt az output regiszternek, azaz a kimeneten ez fog megjelenni.

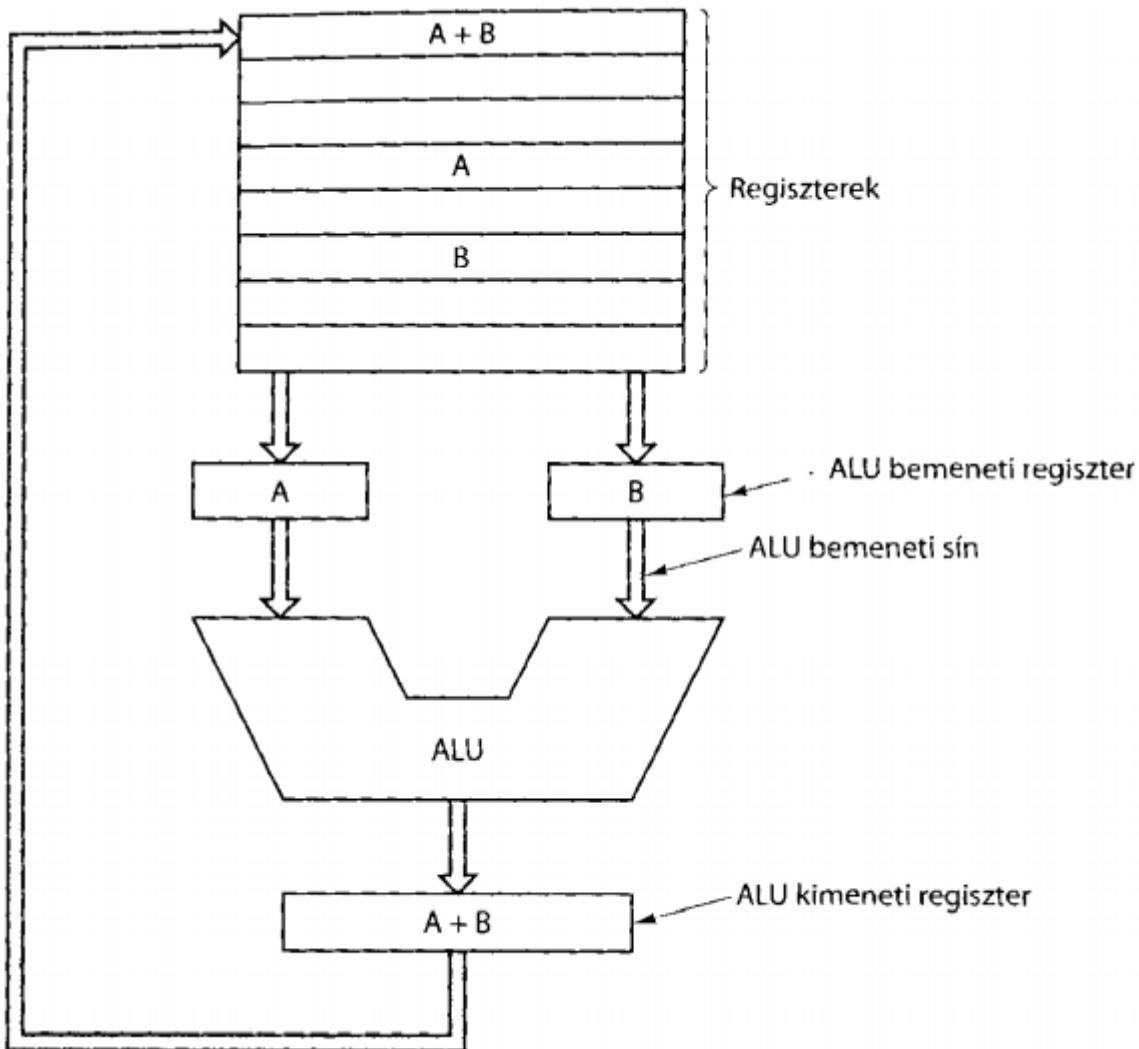
- az utasítások végrehajtásához szükséges aritmetikai és logikai műveleteket végzi el
- Aritmetikai operátorok: +, -, *, / (alapműveletek)
- Logikai operátorok: NOT, AND, OR, NAND, NOR, XOR, NXOR (EQ)

A regiszterek kis méretű, gyorsan címezhető és olvasható memóriarekeszek, amelyek részeredményeket és vezérlőinformációkat tárolnak. A számítógépek központi feldolgozó egységeinek (CPU), illetve mikroprocesszorainak gyorsan írható-olvasható, ideiglenes tartalmú, és általában egyszerre csak 1 gépi szó feldolgozására alkalmas tárolóegységei.

Az adatút az adatok áramlásának útja, alapfeladata, hogy kiválasszon egy vagy két regisztert, az ALU-val műveletet végeztessen el rajtuk (összeadás, kivonás...), az eredményt pedig valamelyik regiszterben tárolja. Egyes gépeken az adatút működését mikrogramm vezérli, másutt a vezérlés közvetlenül a hardver feladata. Folyamata:

- A regiszter készletből feltöltődik az ALU két bemenő regisztere (A és B)
- Az eredmény az ALU kimenő regiszterébe kerül
- Az ALU kimenő regiszteréből a kijelölt regiszterbe kerül az eredmény

Két operandusnak az ALU-n történő átfutásából és az eredmény regiszterbe tárolásából álló folyamatot **adatútciklusnak** nevezzük.



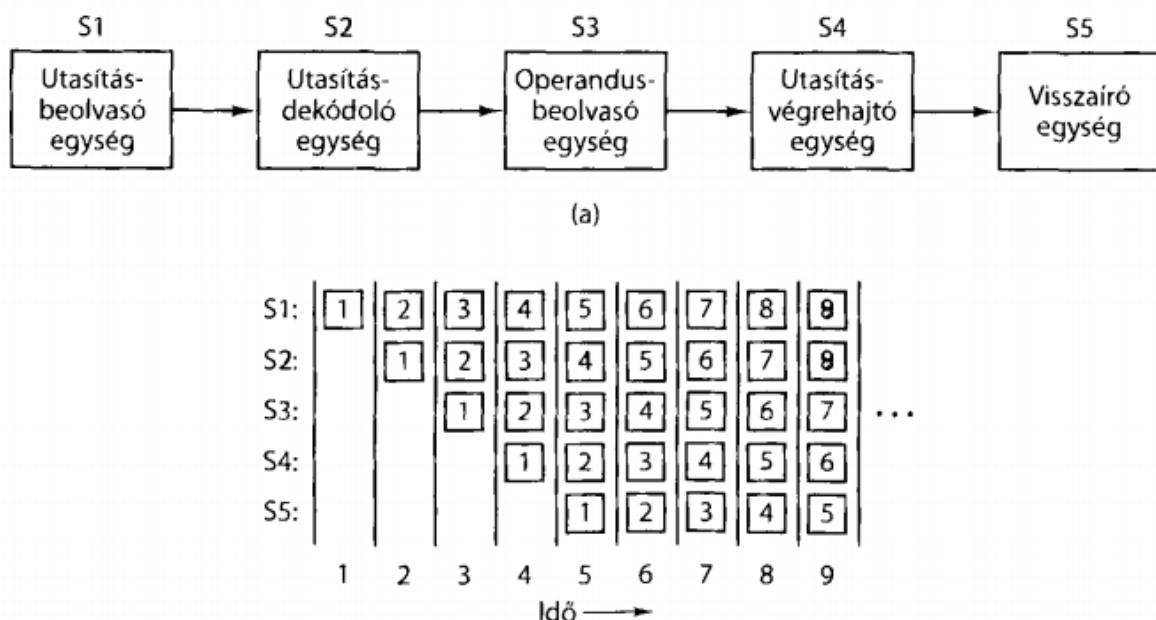
Utasítás-véghajtás: A mikroprocesszor 1-1 utasítása úgynevezett gépi ciklusok sorozatából áll, vagyis 1 utasítás egy vagy több gépi ciklusból tevődik össze, amit a CPU apró lépések sorozataként hajt végre a következőképpen:

- A soron következő utasítás beolvasása a memóriából az utasításregiszterbe
- Az utasításszámláló beállítása a következő utasítás címére
- A beolvasott utasítás típusának meghatározása
- Ha az utasítás memóriabeli szót használ, a szó helyének megállapítása
- Ha szükséges, a szó beolvasása a CPU egy regiszterébe
- Az utasítás véghajtása
 - Vissza az 1. pontra, a következő utasítás véghajtásának megkezdése

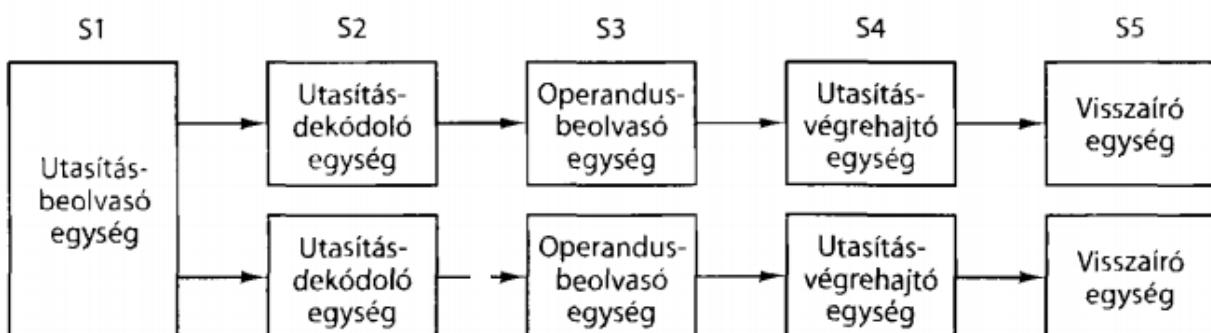
Ezt a lépéssorozatot gyakran nevezik **betöltő-dekódoló-véghajtó ciklusnak**, és központi szerepet tölt be minden számítógép működésében. Nagy probléma a számítógépeknél, hogy a memória olvasása lassú, ezért az utasítás és az adatok beolvasása közben a CPU több része kihasználatlan. A gyorsítás egyik módja az órajel frekvenciájának növelése, de ez korlátozott. Emiatt a legtöbb tervező a **párhuzamosság** kiaknázásában lát lehetőséget. Itt fontos még megemlíteni két fogalmat: **késleltetés**: utasítás véghajtásának időigénye, **áteresztőképesség**: MIPS (millió utasítás mp-enként).

Utasításszintű párhuzamosság: Az utasítások végrehajtásának gyorsítása érdekében előre be lehet olvasni az utasításokat, hogy azok rendelkezésre álljanak, amikor szükség van rájuk. Ezeket az utasításokat egy előolvasási puffer (prefetch buffer) elnevezésű regiszterkészlet tárolja. Ilyen módon a soron következő utasítást általában az előolvasási pufferből lehet venni ahelyett, hogy egy memóriaolvasás befejeződésére kellene várni.

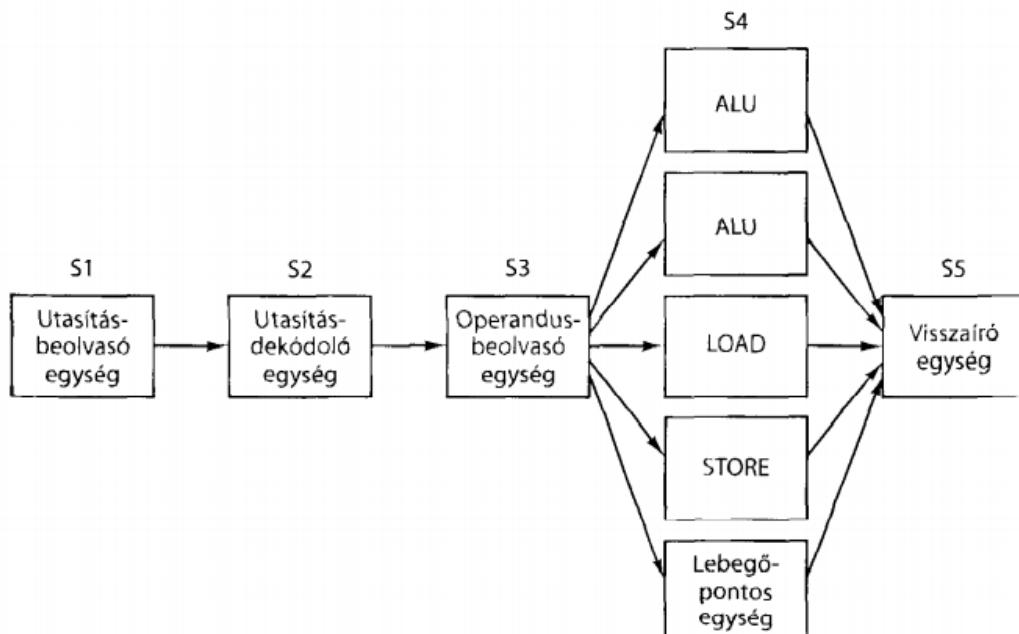
Csővezeték: Lényegében az előolvasás az utasítás végrehajtását két részre osztja: beolvasás és tulajdonképpeni végrehajtás. A csővezeték ezt a stratégiát viszi sokkal tovább. Az utasítás végrehajtását kettő helyett több részre osztja, minden részt külön hardverelem kezel, amelyek minden egyszerre működhettek. Lehetővé teszi, hogy kompromisszumot kössünk késleltetés (meddig tart egy utasítás végrehajtása) és áteresztőképesség (hány MIPS a processzor) között.



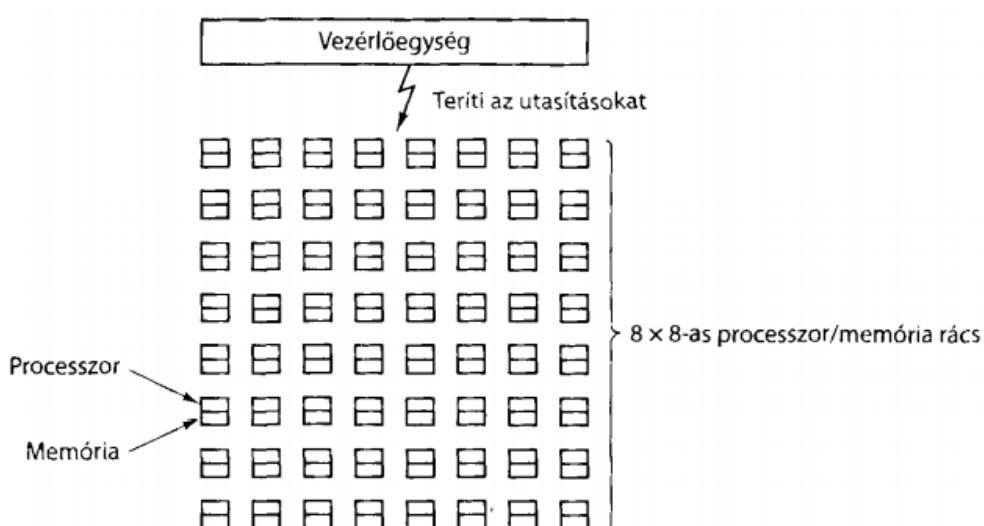
Párhuzamos csővezeték: Az előolvasó egység két utasítást olvas be egyszerre, majd ezeket az egyik, illetve a másik csővezetékre teszi. A csővezetékeknek saját ALU-juk van, így párhuzamosan tudnak működni, feltéve, hogy a két utasítás nem használja ugyanazt az erőforrást, és egyik sem használja fel a másik eredményét. Ugyanúgy, mint egyetlen csővezeték esetén, a feltételek betartását vagy a fordítóprogramnak kell garantálnia, vagy a konfliktusokat egy kiegészítő hardvernek kell a végrehajtás során felismernie és kiküszöbölnie.



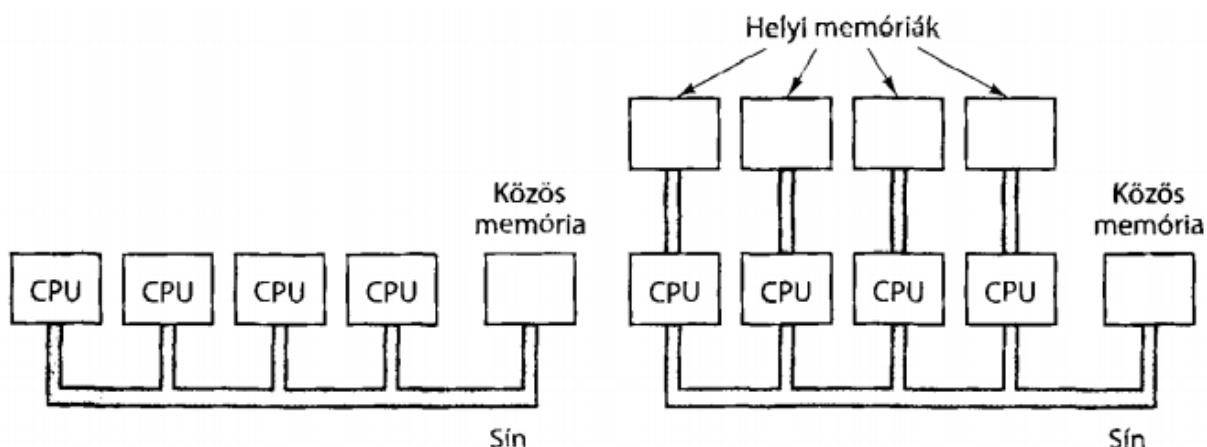
Szuperskaláris architektúra: Itt egy csővezetéket használnak, több funkcionális egységgel. Ezek olyan processzorok, amelyek több – gyakran négy vagy hat – utasítás végrehajtását kezdik el egyetlen órajel alatt. Természetesen egy szuperskaláris CPU-nak több funkcionális egységének kell lennie, amelyek kezelik mindezeket az utasításokat. Az utasítások megkezdését sokkal nagyobb ütemben végezik, mint amilyen ütemben azokat végre lehet hajtani, így a terhelés megszlik a funkcionális egységek között. A megfelelő fázis lényegesen gyorsabban tudja előkészíteni az utasításokat, mint ahogy a rákövetkező fázis képes azokat végrehajtani. Ez a fázis funkcionális egységeinek többsége egy órajelnél jóval több időt igényel feladata elvégzéséhez – a memoriához fordulók vagy a lebegőpontos műveleteket végzők tuti. Több ALU-t is tartalmazhat.



A processzorszintű párhuzamosság egyik megvalósítása a **Tömb processzorok**. Egy ilyen processzor nagyszámú egyforma processzorból áll, ugyanazon műveleteket egyszerre végzik különböző adathalmazokon. A feladatok szabályossága és szerkezete különösen megfelelővé teszi ezeket párhuzamos feldolgozásra. Olyan utasításokat hajthatnak végre, mint amilyen például két vektor elemeinek páronkénti összeadása. Időnként **SIMD** (Single Instruction-stream Multiple Data stream) processzorként is hivatkoznak rá.



Multiprocesszorokban több teljes CPU van, amelyek egy közös memóriát használnak. Amikor két vagy több CPU rendelkezik azzal a képességgel, hogy szorosan együttműködjenek, akkor azokat szorosan kapcsoltaknak nevezik. A legegyszerűbb, ha egyetlen sín van, amelyhez csatlakoztatjuk a memóriát és az összes processzort. Ha sok gyors processzor próbálja állandóan elérni a memóriát a közös sínen keresztül, az konfliktusokhoz vezet. Az egyik megoldás, hogy minden processzornak biztosítunk valamekkora saját lokális memóriát, amelyet a többiek nem érhetnek el. Így csökken a közös sín forgalma. Jellemzően maximum pár száz CPU-t építenek össze.



Multiszámitógépek: Nehéz sok processzort és memóriát összekötni. Ezért gyakran sok összekapcsolt számítógépből álló rendszereket építnek, amelyeknek csak saját memóriájuk van. A multiszámitógépek CPU-it lazán kapcsoltaknak nevezik. A multiszámitógép processzorai üzenetek küldésével kommunikálnak egymással. Nagy rendszerekben nem célszerű minden számítógépet minden másikkal összekötni, ezért 2 és 3 dimenziós rácsot, fákat és gyűrűket használnak. Ennek következtében egy gép valamelyik másikhoz küldött üzeneteinek gyakran egy vagy több közbenső gépen vagy csomópontron kell áthaladniuk ahhoz, hogy a kiindulási helyükön elérjenek a céljukhoz. Néhány mikroszekundumos nagyságrendű üzenetküldési idők nagyobb nehézség nélkül elérhetők. 10 000 processzort tartalmazó multiszámitógépeket is építettek már.

Korszerű számítógépek tervezési elvei

Minden utasítást közvetlenül a hardver hajtson végre

- A gyakran használtakat mindenkorban
- Interpretált mikrutasítások elkerülése

Maximalizálni kell az utasítások kiadási ütemét

- Párhuzamos utasításkiadásra törekedni

Az utasítások könnyen dekódolhatók legyenek

- Kevés mezőből álljanak, szabályosak, egyformá hosszúak legyenek, ...

Csak a betöltő és a tároló utasítások hivatkozzanak a memóriára

- Egyszerűbb utasításforma, párhuzamosítást segíti

Sok regiszter legyen

- Számítások során ne kelljen a lassú memóriába írni

RISC és CISC architektúrák és jellemzőik

UltraSPARC (RISC): Azt a szempontot tartották szem előtt, hogy a processzor kevés alapvető utasítást tudjon végrehajtani, de azokat rendkívül gyorsan (jellemzően 1 órajelciklus alatt). Ezek a RISC (Reduced Instruction Set Computer - redukált utasításkészletű) processzorok.

Itt az összetettebb funkciókat több utasítás kombinációjával lehet megvalósítani. A RISC mikroprocesszorokba számos belső regiszter kerül integrálásra, ezáltal is csökkentve a memoriához való fordulás gyakoriságát és gyorsítva a működést.

Ugyancsak sajátja ezen processzoroknak a - később ismertetett - ún. pipeline architektúra. Ennek lényege az, hogy a műveleteket részműveletekké bontják szét, és e részműveleteket időben párhuzamosítják. A RISC processzorok az utolsó 10 évben - első sorban a nagyobb teljesítményt igénylő rendszereknél (pl. munkaállomások) nyertek teret.

Nagyon kevés utasítással rendelkeznek, tipikusan 50 körül. Az adatút egyszeri bejárásával végrehajthatók ezek az utasítások, tehát egy órajel alatt. Nem használ mikroprogram interpretálást, ezért sokkal gyorsabb, mint a CISC.

Pentium 4 (CISC): Azok a processzorok tartoznak ide, amelyek utasításkészlete lehetőleg minden programozói igényt ki próbál elégíteni, vagyis komplex utasításkészletet alkot. Ezeket nevezzük CISC (Complex Instruction Set Computer = komplex utasításkészletű számítógép) processzoroknak. Markáns elemei az Intel processzorok. A CISC törekvésnek az egyik mozgatórugója, hogy megpróbálják a magasabb szintű nyelvek lehetőségeit közelíteni, vagyis, hogy a programozás "munkaigényes" alacsony szintjét, gépközeli voltát így is ellensúlyozzák.

Interpretálást használ, ezért sokkal összetettebb utasításai vannak, mint egy RISC gépnek. Több száz ilyen utasítása lehet. Az interpretálás miatt lassabb a végrehajtás.

Összefoglalás

Neumann elvű gép egységei

- központi memória: a program kódját és adatait tárolja számokként
- központi feldolgozóegység: memóriában lévő utasítások beolvasása és végrehajtása
- külső sín: a részegységeket köti össze, adatokat, címeket, vezérlőjeleket továbbít
- belső sín: CPU részegységei közötti kommunikáció (vezérlőegység-ALU-regiszterek)
- beviteli/kiviteli eszközök: kapcsolat a felhasználóval, adattárolás (háttértáron), stb.
- működést biztosító járulékos eszközök: például gépház, tápellátás, hűtés...

Adatút

- az adatok áramlásának útja
- alapfeladata, hogy kiválasszon egy vagy két regisztert, az ALU-val műveletet végezzen el rajtuk (összeadás, kivonás...), az eredményt pedig valamelyik regiszterben tárolja
- mikroprogram vezérelheti, de ez lehet közvetlenül a hardver feladata is
 - a regiszter készletből feltöltődik az ALU két bemenő regisztere (A és B)
 - az eredmény az ALU kimenő regiszterébe kerül
 - az ALU kimenő regiszteréből a kijelölt regiszterbe kerül az eredmény
- ezt a folyamatot adatútciklusnak nevezzük

Utasításvégrehajtás

- az utasítások gépi ciklusok sorozatából állnak
- egy utasítás végrehajtás egy vagy több gépi ciklusból tevődik össze, amit a CPU lépések sorozataként hajt végre
 - a soron következő utasítás beolvasása a memóriából az utasításregiszterbe
 - az utasításszámláló beállítása a következő utasítás címére.
 - a beolvasott utasítás típusának meghatározása.
 - ha az utasítás memóriabeli szót használ, a szó helyének megállapítása.
 - ha szükséges, a szó beolvasása a CPU egy regiszterébe.
 - az utasítás végrehajtása.
 - vissza az 1. pontra, a következő utasítás végrehajtásának megkezdése.
- betöltő-dekódoló-végrehajtó ciklus, központi szerepet tölt be minden számítógépen
- a memória olvasása lassú, az utasítás adatai beolvasása közben a CPU kihasználatlan
- gyorsítás egyik módja az órajel frekvenciájának növelése, de ez korlátozott
- a legtöbb tervező a párhuzamosságban lát lehetőséget.
- késleltetés: utasítás végrehajtásának időigénye
- áteresztőképesség: MIPS (millió utasítás mp-enként).

Utasításszintű párhuzamosság

- CSŐVEZETÉK
 - az utasítás végrehajtását több részre osztja
 - minden részt külön hardverelem kezel, amelyek minden egyszerre működhetnek
 - kompromisszumot tudunk kötni késleltetés és áteresztőképesség között
- PÁRHUZAMOS CSŐVEZETÉK
 - két utasítást olvas be egyszerre,
 - ezeket az egyik, illetve a másik csővezetékre teszi
 - saját ALU-juk van, párhuzamosan tudnak működni, feltéve, hogy a két utasítás nem használja ugyanazt az erőforrást, és egyik sem használja fel a másik eredményét
 - a feltételek betartását vagy a fordítóprogramnak, vagy a konfliktusokat egy kiegészítő hardvernek kell a végrehajtás során felismernie és kiküszöbölnie.

- SZUPERSKALÁRIS ARCHITEKTÚRA
 - egy csővezetéket használnak, több funkcionális egységgel
 - olyan processzorok, amelyek több utasítás végrehajtását kezdik el egy órajel alatt
 - sokkal gyorsabban kezdik meg az utasításokat, mint ahogy azokat végrehajtják
 - megeszik a terhelés az ALU-k között
 - a memóriához fordulók vagy a lebegőpontos műveleteket végzők sokáig tartanak

Processzorszintű párhuzamosság

- TÖMB PROCESSZOROK
 - nagyszámú egyforma processzorból áll
 - ugyanazon műveleteket egyszerre végzik különböző adathalmazokon
 - a feladatok szabályossága és szerkezete miatt különösen hatékony párhuzamosítás
 - például két vektor elemeinek páronkénti összeadása.
 - **SIMD** (Single Instruction-stream Multiple Data stream)
- MULTIPROCESSZOROK
 - több teljes CPU van
 - közös memória
 - ha két vagy több CPU tud szorosan együttműködni akkor azokat szorosan kapcsoltaknak nevezik
 - egyetlen sín van, amelyhez csatlakoztatjuk a memóriát és az összes processzort
 - ha sok gyors processzor próbálja állandóan elérni a memóriát a közös sínen keresztül, az konfliktusokhoz vezet
 - megoldás: minden processzornak biztosítunk valamekkora saját lokális memóriát
 - jellemzően maximum pár száz CPU-t építenek össze
- MULTISZÁMÍTÓGÉPEK
 - nehéz sok processzort és memóriát összekötni
 - sok összekapcsolt számítógépből álló rendszereket építnek, amelyeknek csak saját memóriájuk van
 - a multiszámítógépek CPU-it lazán kapcsoltaknak nevezik
 - processzorai üzenetek küldésével kommunikálnak egymással
 - nagy rendszerekben nem célszerű minden számítógépet minden másikkal összekötni, ezért 2 és 3 dimenziós rácsot, fákat és gyűrűket használnak
 - egy gép valamelyik másikhoz küldött üzeneteinek gyakran egy vagy több közbenső gépen vagy csomóponton kell áthaladniuk.
 - Néhány mikroszekundumos nagyságrendű üzenetküldési idők nagyobb nehézség nélkül elérhetők
 - 10 000 processzort tartalmazó multiszámítógépeket is építettek már.

Korszerű számítógépek fejlesztési elvei

- minden utasítást közvetlenül a hardver hajtson végre
- Maximalizálni kell az utasítások kiadási ütemét
- Az utasítások könnyen dekódolhatók legyenek
- Csak a betöltő és a tároló utasítások hivatkozzanak a memóriára
- Sok regiszter legyen

RISC

- Reduced Instruction Set Computer (csökkentett utasításkészletű számítógép)
- az összetettebb funkciókat több utasítás kombinációjával lehet megvalósítani
- számos belső regiszter → kevesebb memóriaművelet
- pipeline architektúra, lényege az, hogy a műveleteket részműveletekké bontják szét, és e részműveleteket időben párhuzamosítják
- kevés utasítással rendelkeznek (50) körül, amik az adatút egyszeri bejárásával végrehajthatók (egy órajel alatt)
- nincs mikroprogram interpretálás, sokkal gyorsabb, mint a CISC.
- UltraSPARC

CISC

- Complex Instruction Set Computer (összetett utasításkészletű számítógép)
- utasításkészlete lehetőleg minden programozói igényt ki próbál elégíteni
- Intel processzorok
- próbálják a magasabb szintű nyelvek lehetőségeit közelíteni
- a programozás "munkaigényes" alacsony szintjét, gépközeli voltát így is ellensúlyozzák
- Interpretálást használ, ezért sokkal összetettebb utasításai vannak, mint egy RISC gépnek
- több száz utasítása is lehet
- az interpretálás miatt lassabb a végrehajtás
- Pentium 4

Számítógép architektúra - 2.)

Számítógép perifériák: Mágneses és optikai adattárolás alapelvei, működésük (merevlemez, Audio CD, CD-ROM, CD-R, CD-RW, DVD, Bluray). SCSI, RAID. Nyomtatók, egér, billentyűzet. Telekommunikációs berendezések (modem, ADSL, KábelTV-s internet).

Számítógép perifériák

A számítógéphez különböző perifériák kapcsolhatók, melyek segítségével a felhasználók kommunikálni tudnak a géppel. Ezek egy része beviteli, vagy kiviteli eszköz, amely az adatok bevitelére, vagy kiírására szolgál. A háttértárolók feladata az adatok és programok hosszabb ideig tartó tárolása. Tartalmuk a számítógép kikapcsolása után is megmarad. A megnevezést általában azokra az eszközökre használják, amelyek külsőleg csatlakoznak a géphez, tipikusan egy számítógépes buszon keresztül, mint például az USB.

Mágneses adattárolás esetén mágneslemezen tárolunk adatokat (merevlemez). Ezek egy vagy több mágnesezhető bevonattal ellátott alumíniumkorongból állnak. Egy indukciós fej lebeg a lemez felszíne felett egy vékony légpárnán. Ha pozitív vagy negatív áram folyik az indukciós tekercsben, a fej alatt a lemez magnetizálódik, és ahogy a korong forog a fej alatt, így bitsorozatot lehet felírni, amiket később vissza lehet olvasni.

Egy teljes körülfordulás alatt felírt bitsorozat a sáv. minden sáv rögzített, tipikusan 512 bájt méretű szektorokra van osztva, amelyeket egy fejléc előz meg, lehetővé téve a fej szinkronizálását írás és olvasás előtt. Az adatok után hibajavító kód helyezkedik el (Hamming vagy Reed-Solomon).

Minden lemeznek vannak mozgatható karjai, melyek a forgástengelytől sugárirányban ki-be tudnak mozogni. minden sugárirányú pozícióban egy-egy sáv írható fel, tehát a sávok forgástengely középpontú koncentrikus körök.

Egy lemezegység több, egymás felett elhelyezett korongból áll. minden felülethez tartozik egy fej és egy mozgatókar. A karok rögzítve vannak egymáshoz, így a fejek minden ugyanarra a sugárirányú pozícióra állnak be. Egy adott sugárirányú pozícióhoz tartozó sávok összességét cilindereknek nevezzük. Általában 6-12 korong található egymás felett.

Egy szektor beolvasásához vagy kiírásához először a fejet a megfelelő sugárirányú pozícióba kell állítani, ezt keresésnek (seek) hívják. A fej kívánt sugárirányú pozícióba való beállása után van egy kis szünet, az ún. forgási késleltetés, amíg a keresett szektor a fej alá fordul. A külső sávok hosszabbak, mint a belsők, a lemezek pedig a fejek pozíciójától függetlenül állandó szögsebességgel forognak, ezért ez problémát vet fel.

Megoldásképp a cilindereket zónákba osztják, és a külső zónákban több szektort tesznek egy sávba. minden szektor mérete egyforma. minden lemezhez tartozik egy lemezvezérlő, egy lapka, amely vezéri a meghajtót.

Az optikai adattárolók olyan kör alakú lemezek, amelyek felületén helyezkedik el az adattárolásra alkalmas réteg. A lemezek írása és olvasása a nevükön adódóan optikai eljárással történik lézersugárral a lemez forgatása közben. A lemezen történő adatrögzítéskor apró mélyedéseket hozunk létre spirál alakú vonalban, így tárolva a digitális adatot, olvasáshoz azonos hullámhosszú lézersugár ellenőrzi, hogy a sugár visszatükröződik, vagy szétszóródik a lemez felületéről.

Az optikai tárolókat több tulajdonságuk is jelentősen megkülönbözteti a mágneses táraktól: az optikai tárolás elméletben sokkal nagyobb adatsűrűséget enged meg, mivel a fény sokkal kisebb területre fókuszálható, mint a mágneses adattárolókban az elemi mágnesezhető részecskék mérete. Továbbá, a megfelelő minőségű és megfelelően kezelt optikai lemezek élettartama évszázadokban mérhető, ezenkívül nem érzékenyek az elektromágneses behatásokra sem.

A felületen elhelyezkedő mélyedéseket üregnek (**pit**), az üregek közötti érintetlen területeket pedig szintnek (**land**) hívják. Az tűnik a legegyszerűbbnek, hogy üreget használunk a 0, szintet az 1 tárolásához, ennél azonban megbízhatóbb, ha az üreg/szint vagy a szint/üreg átmenetet használjuk az 1-hez, az átmenet hiányát pedig a 0-hoz.

Az adattárolók részletes tulajdonságai az összegzésben találhatók!

A **SCSI** (Small Computer System Interface) olyan szabványegyüttettség, melyet számítógépek és perifériák közötti adatátvitelre terveztek. A SCSI szabványok definiálják a parancsokat, protokollokat, valamint az elektromos és az optikai csatolófelületeket.

Leggyakoribb felhasználási területe a merevlemezek és mágnesszalag-meghajtók, de sok más periférián is alkalmazzák (pl. szkennerek, CD-meghajtók).

Az SCSI-lemezek nem különböznek az IDE-lemezeiktől abban a tekintetben, hogy ezek is csatlakoznak a szabványhoz, sávokra és szektorokra osztva, de más az interfészük, és sokkal nagyobb az adatátviteli sebességük. Az 5 MHz-estől a 160 MHz-ig nagyon sok változatot kifejlesztettek.

A SCSI több egy merevlemez-interfésznel. Ez egy sín, amelyre egy SCSI-vezérlő és legfeljebb hétközött csatlakoztatható. Ezek között lehet egy vagy több SCSI-merevlemez, CD-ROM, CD-író, szkanner, szalagegység és más SCSI-periféria.

A SCSI-vezérlők és perifériák kezdeményező és fogadó üzemmódban működhetnek. Általában a kezdeményezőként működő vezérlő adja ki a parancsokat a fogadóként viselkedő lemezegységeknek és egyéb perifériáknak.

A szabvány megengedi, hogy az összes eszköz egyszerre működjön, így nagyban növelhető a hatékonyság több folyamatot futtató környezetben.

A **RAID** egy tárolási technológia, mely segítségével az adatok elosztása vagy replikálása több fizikailag független merevlemezen, egy logikai lemez létrehozásával lehetséges. minden RAID szint alapjában véve vagy az adatbiztonság növelését vagy az adatátviteli sebesség növelését szolgálja.

Azon túl, hogy a RAID szoftveres szempontból egyetlen lemeznek látszik, az adatok szét vannak osztva a meghajtók között, lehetővé téve a párhuzamos működést. Alapötlete a lemezegységek csíkokra (**stripes**) bontása, amik **nem azonosak a lemez fizikai sávjaival**.

RAID-0 (összefűzés vagy csíkozás): Lemezek egyszerű összefűzését jelenti, viszont semmilyen redundanciát nem ad, így nem biztosít hibatűrést. Egyetlen meghajtó mehibásodása az egész tömb hibáját okozza. Az írási és olvasási műveletek párhuzamosítva történnek, ideális esetben a sebesség az egyes lemezek sebességének összege lesz, így a módszer a RAID szintek közül a legjobb teljesítményt nyújtja (a többi módszernél a redundancia kezelése lassítja a rendszert)

RAID-1 (tükrözés): alapja az adatok tükrözése, azaz az információk egyidejű tárolása a tömb minden elemén. Az adatok olvasása párhuzamosan történik a diszkekéről, felgyorsítván az olvasás sebességét; az írás normál sebességgel, párhuzamosan történik a meghajtókon. Igen jó védelmet biztosít, bármely meghajtó meghibásodása esetén folytatódhat a működés.

RAID-2: Egyes meghajtókat hibajavító tárolására tartanak fenn. A hibajavító kód lényege, hogy az adatbitekből valamilyen matematikai művelet segítségével redundáns biteket képeznek. A használt eljárástól függően a kapott kód akár több bithiba észlelésére, illetve javítására alkalmas. A védelem ára a megnövekedett adatmennyiség. A módszer esetleges lemezhiba esetén képes annak detektálására, illetve kijavítására.

RAID-3: Felépítése hasonlít a RAID 2-re, viszont nem a teljes hibajavító kód, hanem csak egy lemeznyi paritásinformáció tárolódik. Egy adott paritáscsík a különböző lemezen azonos pozícióban elhelyezkedő csíkokból XOR művelet segítségével kapható meg. A rendszerben egy meghajtó kiesése nem okoz problémát, mivel a rajta lévő információ a többi meghajtó XOR-aként megkapható.

RAID-4: felépítése a RAID 3-mal megegyezik. Az egyetlen különbség, hogy itt nagyméretű csíkokat definiálnak, így egy rekord egy meghajtón helyezkedik el, lehetővé téve egyszerre több rekord párhuzamos írását, olvasását. Problémát okoz viszont, hogy a paritás-meghajtó adott csíkját minden egyes íráskor frissíteni kell, aminek következtében párhuzamos íráskor a paritásmeghajtó a rendszer szűk keresztmetszetévé válik.

RAID-5: a paritás információt nem egy kitüntetett meghajtón, hanem „körbeforgó paritás” használatával, egyenletesen az összes meghajtón elosztva tárolja, ezzel megoldást adva a paritás-meghajtó jelentette szűk keresztmetszetet. Mind az írási, mind az olvasási műveletek párhuzamosan végezhetőek. Egy meghajtó meghibásodása esetén az adatok sértetlenül visszaolvashatóak, a hibás meghajtó adatait a vezérlő a többi meghajtóról ki tudja számolni.

A nyomtatók nyomtatófejében apró tük vannak (általában 9 vagy 24 db). A papír előtt egy kifeszített festékszalag mozog, amelyre a tük ráütnek, és létrehoznak a papíron egy pontot. A képekből a pontokból fog állni. A tüköt elektromágneses tér mozgatja, és rugóerő húzza vissza eredeti helyükre. Ezzel az eljárással nem csak karakterek, hanem képek, rajzok is nyomtathatóak. A nyomtatott képek felbontása gyenge, a nyomtató lassú viszont olcsók és nagyon megbízhatók.

A tintasugaras nyomtatók tintapatronok segítségével tintacseppeket juttatnak a papírlapra. A patronban van egy porlasztó, ez megfelelő méretű tintacseppekre alakítja a tintát, és a papírlapra juttatja azt. A színes tintasugaras nyomtató színes tintapatronokat használ, általában négy alapsín használatával keveri ki a megfelelő árnyalatokat: ciánkék, bíborvörös, sárga és fekete színek használatával. **Minden tintasugaras nyomtató porlasztással juttatja a tintacseppeket a papírlapra, de a porlasztás módszere változik.** Ez történhet piezoelektronos úton, elektrosztatikusan, vagy gózbuborékok segítségével.

A tintasugaras nyomtatók egy-egy karaktert sokkal több képpontból állítanak össze mint például a mátrixnyomtatók, ezért sokkal szébb képet is adnak annál: megfelelő tintasugaras nyomtatóval igen jó minőségű, színes képek, akár fotók is nyomtathatók.

A lézernyomtatók esetében a készülék szíve egy fényérzékeny anyaggal bevont forgó henger. Egy-egy lap nyomtatása előtt elektromosan feltöltődik, majd egy lézer fénye pásztázza végig a hosszában, amelyet egy nyolcszögletű tükrrel irányítanak a hengerre. A fényt modulálják, hogy világos és sötét pontokat kapjanak. Azok a pontok, ahol fény éri a hengert, elveszítik elektromos töltésüket. Miután egy pontkból álló sor elkészült, a henger elfordul és elkezdődhet a következő sor előállítása. Később az első sor eléri a tonerkazettát, amely elektrosztatikusan érzékeny fekete port tartalmaz. A por hozzátapad a még feltöltött pontokhoz, így láthatóvá válik a sor. Tovább fordulva a bevont henger hozzányomódik a papírhoz, átadva a papírnak a festéket. A papír ezután felmelegített görgők között halad el, ezáltal a festék véglegesen hozzátapad a papírhoz. A lézernyomtatók kiváló minőségű képet készítenek, nagy a rugalmasságuk, sebességük és elfogadható a költség.

Az egér egy grafikus felületen való mutató mozgatására szolgáló periféria. Az egéren egy, kettő vagy akár több nyomógomb van, illetve egy görgő is lehet rajta. Belsejében található érzékelő felismeri és továbbítja a számítógép felé az egér mozgását egy sima felületen.

Az optikai egér a mozgásokat egy optikai szenzor segítségével ismeri fel, mely egy fénykibocsátó diódát használt a megvilágításhoz. Az első optikai egereket csak egy speciális fémes egérapadon lehetett használni, melyre kék és szürke vonalak hálója volt felfestve. Miután a számítógépes eszközök egyre olcsóbbak lettek, lehetőség nyílt egy sokkal pontosabb képelemző chip beépítésére is az egérbe, melynek segítségével az egér mozgását már szinte bármilyen felületen érzékelni lehetett, így többé nem volt szükség speciális egérapadra. Ez a fejlesztés megnyitotta a lehetőséget az optikai egerek elterjedése előtt.

A modern optikai egerek egy reflexszenzor segítségével sorozatos képeket készítenek az egér alatti területről. A képek közötti eltérést egy képelemző chip dolgozza fel, és az eredményt a két tengelyhez viszonyított elmozdulássá alakítja.

Mechanikus egér esetében egy golyó két egymásra merőleges tengelyt forgat, melyek továbbítják a mozgását a fényáteresztő résekkel rendelkező korongoknak. Végeredményben az egér elmozdulása fényimpulzusok sorozatává változik, mégpedig annál több fényimpulzus keletkezik, minél nagyobb az egér által megtett út. A fényérzékeny szenzorok érzékelik az impulzusokat és elektromos jelekké alakítják.

A billentyűzet gombjai kábelezés szempontjából egy ún. billentyűzet-mátrixban vannak elhelyezve. Egy meghatározott billentyű lenyomásának vagy felengedésének észlelésekor a belső mikroprocesszor egy, az adott billentyűt egyértelműen azonosító ún. scan-kódot küld a számítógép felé. Ugyanezen billentyű felengedésekor a mikroprocesszor egy másik, felengedési scan-kódot továbbít a billentyűzet-illesztő áramkör felé. Ezáltal részint kiküszöbölné a több billentyű közel egyidejű lenyomásából adódó jelenség, a karakterek "elvesztése". A megfelelő gomb vagy kombinációk értelmezése és feldolgozása így teljesen a számítógép billentyűzetkezelő rutinjának feladata.

Telekommunikációs berendezések

A **modem** egy olyan berendezés, ami egy vivőhullám modulációjával a digitális jelet analóg információvá, illetve a másik oldalon ennek demodulációjával újra digitális információvá alakítja. Az eljárás célja, hogy a digitális adatot analóg módon átvihetővé tegye.

A moduláció különféle eljárások csoportja, amelyek biztosítják, hogy egy tipikusan szinuszos jel - a vivő - képes legyen információ hordozására. A szinuszos jel három fő paraméterét, az amplitúdóját, a fázisát vagy a frekvenciáját módosíthatja a modulációs eljárás, azért, hogy a vivő információt hordozhasson.

A modem egy másik modemmel működik párból, ezek az átviteli közeg két végén vannak. Szigorú értelemben véve a két modem két adatátviteli berendezést köt össze, azonban a másik végberendezés tovább csatlakozhat az internet felé.

Az **ADSL** (Asymmetric Digital Subscriber Line, aszimmetrikus digitális előfizetői vonal) valójában egy olyan kommunikációs technológia, amely egy csavart rézérpárú telefonkábelben keresztül juttat el adatot A pontból B pontba. A technológia segítségével a hagyományos modemekhez képest gyorsabb digitális adatátvitel érhető el, ezért igazi áttörés volt megjelenése az internetszolgáltatás piacán.

Az ADSL jellemzője, hogy a letöltési és a feltöltési sávszélesség aránya nem egyenlő (vagyis a vonal aszimmetrikus), amely az otthoni felhasználóknak kedvezve a letöltés sebességét helyezi előnybe a feltöltéssel szemben.

Mind technikai, mind üzleti okai vannak az ADSL gyors elterjedésének. A technikai előnyt az adja, hogy a zajelnyomási lehetőségeket kihasználva lehetővé teszi nagyobb távolságon is a gyors adatátvitelt a felhasználó lakása és a DSLAM eszköz között (amely a telefonközpontokban helyezkedik el, Digital Subscriber Line Access Multiplexer).

KábelTV-s internet szolgáltatói minden városban fő telephellyel rendelkeznek, valamint rengeteg, elektronikával zsúfolt dobozzal szerte a működési területükön, amelyeket fejállomásoknak neveznek.

A fejállomások nagy sávszélességű kábelekkel vagy üvegkábelekkel kapcsolódnak a fő telephelyhez. minden fejállomásról egy vagy több kábel indul el, otthonok és irodák százain halad keresztül. minden előfizető a rajta keresztülhaladó kábelhez csatlakozik. így a felhasználók osztoznak egy fejállomáshoz vezető kábelen, ezért a kiszolgálás sebessége attól függ, hogy pillanatnyilag hányan használják az adott vezetéket. A kábelek sávszélessége 750 MHz.

Összegzés

Mágneses és optikai adattárolás

- MÁGNESES TÁROLÁS
 - mágneslemezen tárolunk adatokat
 - egy vagy több mágnesezhető bevonattal ellátott alumínium korong
 - egy indukciós fej lebeg a lemez felszíne felett egy vékony légpárnán
 - ha áram folyik az indukciós tekercsben, a fej alatt a lemez magnetizálódik
 - ahogyan a korong forog a fej alatt, bitsorozatot lehet írni és olvasni
 - egy teljes körülfordulás alatt felírt bitsorozat a sáv
 - minden sáv rögzített (512 bájt) méretű szektorokra van osztva
 - szektorok előtt fejléc, lehetővé téve a fej szinkronizálását
 - az adatok után hibajavító kód helyezkedik el (Hamming vagy Reed-Solomon)
 - minden lemeznek vannak mozgatható karjai
 - a forgástengelytől sugárirányban ki-be tudnak mozogni
 - minden sugárirányú pozícióban egy-egy sáv írható fel (sávok koncentrikus körök)
 - egy lemezegység több, egymás felett elhelyezett korongból áll, saját fejjel
 - a karok rögzítve vannak egymáshoz,
 - egy adott sugárirányú pozícióhoz tartozó sávok összessége a cilinder
 - egy szektor beolvasásához vagy kiírásához először a fejet a megfelelő pozícióba kell állítani (seek)
 - a fej kívánt sugárirányú pozícióba való beállása utáni kis szünet (forgási késleltetés)
 - külső sávok hosszabbak, mint a belsők, állandó sebesség mellett ez probléma
 - a cilindereket zónákba osztják
 - a külső zónákban több szektort tesznek egy sávba
- OPTIKAI TÁROLÁS
 - kör alakú lemezek
 - felületén helyezkedik el az adattárolásra alkalmas réteg
 - írása és olvasása lézersugárral történik a lemez forgatása közben
 - apró mélyedéseket hozunk létre spirál alakú vonalban
 - olvasáshoz azonos hullámhosszú lézersugár ellenőrzi, hogy a sugár tükröződik, vagy szétszóródik
 - sokkal nagyobb adatsűrűséget enged meg, mivel a fény sokkal kisebb területre fókuszálható,
 - megfelelő minőségű és megfelelően kezelt optikai lemezek élettartama nagyobb
 - nem érzékenyek az elektromágneses behatásokra sem.
 - mélyedések üregek (pit), az üregek közötti érintetlen területek pedig szintnek (land)
 - üreg/szint vagy a szint/üreg átmenetet használjuk az 1-hez, az átmenet hiányát pedig a 0-hoz.
- HDD
 - mágneses
 - 500 GB - 12 TB
 - írási és olvasási sebessége függ:
 - a lemezek forgási sebességétől (5400, 7200, 1000 vagy 15000 RPM),
 - az adatsűrűségtől

- AUDIO CD
 - a jel sűrűsége állandó a spirál mentén
 - 74 percnyi anyag fér rá (Beethoven IX. szimfóniája kiadható legyen)
 - állandó kerületi sebesség, változó forgási sebesség (120 cm/mp)
 - nincs hibajavítás, nem gond, ha néhány bit elvész az audio anyagból
- CD-ROM
 - univerzális adathordozó, illetve médialemez
 - csak olvasható (véglegesített, read-only memory)
 - népszerűen használták szoftverek és adatok terjesztésére
 - kereskedelmi forgalomban hozzák létre, létrehozásuk után nem menthető rájuk adat
 - 650 MB tárolható
- CD-R
 - író berendezéssel rögzíthető az adat (1x)
 - újdonság:
 - író lézernyaláb
 - alumínium helyett arany felület
 - üregek és szintek helyett festékréteg alkalmazása
 - kezdetben átlátszó a festékréteg (cianin (zöld) vagy ftalocianin (sárgás))
 - 700 MB tárolható
- CD-RW
 - újraírható optikai lemez (read-write)
 - újdonság:
 - más adattároló réteg:
 - ezüst, indium, antimon és tellúr ötvözeti
 - kétféle stabil állapot: kristályos és amorf (más fényvisszaverő képesség)
 - 3 eltérő energiájú lézer:
 - legmagasabb energia: megolvad az ötvözeti → amorf
 - közepes energia: megolvad → kristályos állapot
 - alacsony energia: anyag állapotnak érzékelése, de meg nem változik
 - 700 MB tárolható
- DVD
 - nagy kapacitású optikai tároló
 - leginkább mozgókép és jó minőségű hang, valamint adat tárolására használatos
 - általában akkora, mint a CD, vagyis 120 mm átmérőjű.
 - egységek/kétrétegek illetve egyoldalas/kétoldalas lemez (4,5 GB – 17 GB)
 - nagyobb jelsűrűség, mert
 - kisebbek az üregek (0,4 µm (CD: 0,8 µm))
 - szorosabb spirálok
 - vörös lézert használtak
- BLURAY
 - a DVD technológia továbbfejlesztése, a Blu-Ray disc
 - kék lézer használata írásra és olvasásra a vörös helyett
 - rövidebb hullámhossz, jobban fókuszálható, kisebb mélyedések
 - 25 GB (egyoldalas) és 50 GB (kétoldalas) adattárolási képesség

SCSI

- Small Computer System Interface
- olyan szabványegyüttes, melyet számítógépek és perifériák közötti adatátvitelre terveztek
- leggyakoribb felhasználási területe a merevlemezek és mágnesszalag-meghajtók
- az SCSI-lemezek nem különböznek is cilinderekre, sávokra és szektorokra vannak osztva
- más az interfészük, ezért sokkal nagyobb az adatátviteli sebességük
- 5 MHz-estől a 160 MHz-ig sok változat.
- SCSI egy sín, amelyre egy SCSI-vezérlő és legfeljebb hét eszköz csatlakoztatható
- lehet SCSI-merevlemez, CD-ROM, CD-író, szalagegység és más SCSI-periféria
- az SCSI-vezérlők és perifériák kezdeményező és fogadó üzemmódban működhetnek
- a kezdeményezőként működő vezérlő adja ki a parancsokat a fogadóként viselkedő lemezegységeknek és egyéb perifériáknak.
- az összes eszköz egyszerre működhet, nagyban növelhető a hatékonyság (párhuzamos)

RAID

- tárolási technológia
- segítségével az adatok elosztása vagy replikálása több fizikailag független merevlemezen, egy logikai lemez létrehozásával lehetséges
- minden RAID szint vagy az adatbiztonság vagy az adatátviteli sebesség növelésére szolgál
- az adatok szét vannak osztva a meghajtók között, lehetővé téve a párhuzamos működést
- lemezegységek csíkokra (stripes) bontása, amik nem azonosak a lemez fizikai sávjaival
- RAID-0 (összefűzés vagy csíkozás):
 - lemezek egyszerű összefűzése
 - semmilyen redundanciát nem ad, így nem biztosít hibatűrést
 - egy meghajtó meghibásodása az egész tömb hibáját okozza
 - írási és olvasási műveletek párhuzamosítva történnek
 - a sebesség az egyes lemezek sebességének összege
 - a RAID szintek közül a legjobb teljesítményt nyújtja
- RAID-1 (tükrözés):
 - alapja az adatok tükrözése
 - az információk egyidejű tárolása a tömb minden elemén
 - az olvasás párhuzamosan történik a diszkekéről, felgyorsítva a sebességet
 - az írás normál sebességgel, párhuzamosan történik a meghajtókon
 - jó védelmet biztosít, egy meghajtó meghibásodása esetén folytatódhat a működés
- RAID-2:
 - egyes meghajtókat hibajavító tárolására tartanak fenn
 - a hibajavító kód lényege, hogy az adatból valamelyen matematikai művelet segítségével redundáns biteket képeznek
 - több bithiba észlelésére, illetve javítására alkalmas
 - megnövekedett adatmennyiség
 - esetleges lemezhiba esetén képes annak detektálására, javítására
- RAID-3:
 - mint a RAID 2, csak nem a teljes hibajavító kód, hanem csak egy lemeznyi paritásinformáció tárolódik
 - egy adott paritáscsík a különböző lemezeken azonos pozícióban elhelyezkedő csíkokból XOR művelet segítségével kapható meg
 - egy meghajtó kiesése nem okoz problémát

- RAID-4:
 - mint a RAID 3, csak nagyméretű csíkok miatt egy rekord egy meghajtón van
 - egyszerre több rekord párhuzamos írása, olvasása
 - a paritás-meghajtó adott csíkját minden egyes íráskor frissíteni kell
 - párhuzamos íráskor a paritásmeghajtó a rendszer szűk keresztmetszetévé válik
- RAID-5:
 - nem egy kitüntetett meghajtón, hanem „körbeforgó paritás” használatával tárolja
 - paritás elosztva az összes meghajtón
 - az írási és olvasási műveletek párhuzamosan végezhetőek
 - mehibásodás esetén az adatok sértetlenül visszaolvashatóak, a hibás meghajtó adatait a vezérlő a többi meghajtóról ki tudja számolni

Perifériák

- NYOMTATÓK
 - nyomtatófejében apró tük vannak (általában 9 vagy 24 db)
 - a papír előtt egy kifeszített festékszalag mozog, amelyre a tük ráütnek, és létrehoznak a papíron egy pontot
 - a kép ezekből a pontokból fog állni
 - a tüket elektromágneses tér mozgatja, és rugóerő húzza vissza eredeti helyükre
 - ezzel az eljárással nem csak karakterek, hanem képek, rajzok is nyomtathatóak
 - gyenge felbontás, a nyomtató lassú viszont olcsók és nagyon megbízhatók
 - TINTASUGARAS
 - tintapatronok segítségével tintacseppeket juttatnak a papírlapra
 - a patronban van egy porlasztó, ez megfelelő méretű tintacseppekre alakítja a tintát, és a papírlapra juttatja azt
 - a színes tintasugaras nyomtatót színes tintapatronokat használ, általában négy alapszín használatával (ciánkék, bíborvörös, sárga és fekete)
 - különböző tintasugaras nyomtatóknál a porlasztás módszere változó lehet
 - sokkal szébb kép a mátrixnyomtatónál
 - tintasugaras nyomtatóval jó minőségű képek, akár fotók is nyomtathatók
 - LÉZERES
 - szíve egy fényérzékeny anyaggal bevont forgó henger
 - nyomtatás előtt elektromosan feltöltődik, majd egy lézer fénye pásztázza végig a hosszában, amelyet egy nyolcszögletű tükkorrel irányítanak
 - a fénnyt modulálják, hogy világos és sötét pontokat kapjanak
 - ha egy sor elkészült, a henger elfordul és elkezdődhet a következő
 - később az első sor eléri a tonerkazettát, amely elektrosztatikusan érzékeny fekete port tartalmaz, ami hozzátapad a még feltöltött pontokhoz, így láthatóvá válik a sor
 - a papír végül felmelegített görgők között halad el, ezáltal a festék véglegesen hozzátapad
 - minőséges képek, nagy a rugalmasságuk, sebességük és elfogadható a költség.
 - EGÉR
 - grafikus felületen való mutató mozgatására szolgáló periféria
 - nyomógombok találhatók rajta, illetve egy görgő is lehet
 - érzékelő ismeri fel és továbbítja a számítógép felé az egér mozgását egy felületen

- OPTIKAI
 - egér a mozgásokat egy optikai szenzor segítségével ismeri fel
 - fénykibocsátó diódát használt a megvilágításhoz
 - kezdetben korlátozott használat (képelemző chip segített rajta)
 - a modern optikai egerek egy reflexszenzor segítségével sorozatos képeket készítenek az egér alatti területről, amit egy képelemző chip dolgoz fel
 - a mozgást a két kép viszonyított elmozdulása adja
- MECHANIKUS
 - egy golyó két egymásra merőleges tengelyt forgat, melyek továbbítják a mozgását a fényáteresztő résekkel rendelkező korongoknak.
 - az egér elmozdulása fényimpulzusok sorozatával változik,
 - annál több fényimpulzus keletkezik, minél nagyobb az egér által megtett út
 - fényérzékeny szenzorok érzékelik az impulzusokat és elektromos jelekkel alakítják
- BILLentyűZET
 - gombjai kábelezés szempontjából egy billentyűzet-mátrixban vannak elhelyezve
 - egy billentyű lenyomásának vagy észlelésekor a belső mikroprocesszor egy, az adott billentyűt egyértelműen azonosító ún. scan-kódot küld a számítógép felé
 - felengedésekor a mikroprocesszor egy másik, felengedési scan-kódot továbbít a billentyűzet-illesztő áramkör felé
 - a megfelelő gomb vagy kombinációk értelmezése és feldolgozása a számítógép billentyűzetkezelő rutinjának feladata

Telekommunikációs berendezések

- MODEM
 - digitális jel → (moduláció) → analóg jel → (demoduláció) → digitális jel
 - célja, hogy a digitális adatot analóg módon átvihetővé tegye
 - moduláció különféle eljárások csoportja, amelyek biztosítják, hogy egy tipikusan szinuszos jel képes legyen információ hordozására
 - amplitúdót, a fázist vagy frekvenciát módosíthatja a modulációs eljárás, azért, hogy az információt hordozhasson.
 - egy másik modemmel működik párban az átviteli közeg két végén
 - két modem két adatátviteli berendezést köt össze (másik csatlakozhat az internetre)
- ADSL
 - Asymmetric Digital Subscriber Line, aszimmetrikus digitális előfizetői vonal
 - csavart rézérpárú telefonkábelben keresztül juttat el adatot A pontból B pontba
 - a hagyományos modemekhez képest gyorsabb digitális adatátvitel
 - a letöltési és a feltöltési sávszélesség különbözik (vonal aszimmetrikus)
 - a zajelnyomás miatt nagyobb távolságon is gyors adatátvitel a felhasználó és a DSLAM (telefonközpontokban) eszköz között
- KÁBELTV-S INTERNET
 - szolgáltatói minden városban fő telephellyel rendelkeznek
 - fejállomások nagy sávszélességű kábelekkel vagy üvegkábelekkel kapcsolódnak a fő telephelyhez
 - minden fejállomásról több kábel indulhat el, több otthonon és irodán keresztül
 - minden előfizető a rajta keresztülhaladó kábelhez csatlakozik
 - a felhasználók osztognak a kábelben, a kiszolgálás sebessége függ a használattól
 - sávszélessége 750 MHz

Mindenkinék, aki ebből a jegyzetből készül fel a vizsgájára, akár most (2021), vagy akár később, a lehető legnagyobb szerencsét kívánom! Nagyon fontos az alapfogalmakkal tisztában lenni, a vizsga során inkább széltében járják be a tételeket, mintsem mélységében. Egyes tanárokra érdemes lehet még e-mailben rákérdezni, hogy az adott tárgyakból mely részeket tartják fontosnak, és esetleg annak részletesebben utána nézni.

Ha már eddig sikerült eljutni, ez is meglesz!

Mantra: a kettesnél nincs jobb jegy, csak szebb!