Záróvizsga-tételek

Mérnökinformatikus szak

GDF

Jelen jegyzethalmaz **2021**-et tekintve aktuális (a tartalom már nem annyira…), a kidolgozások ténybeli és helyesírási hibákat tartalmazhatnak (sőt)…

Tartalomjegyzék

A tételsor

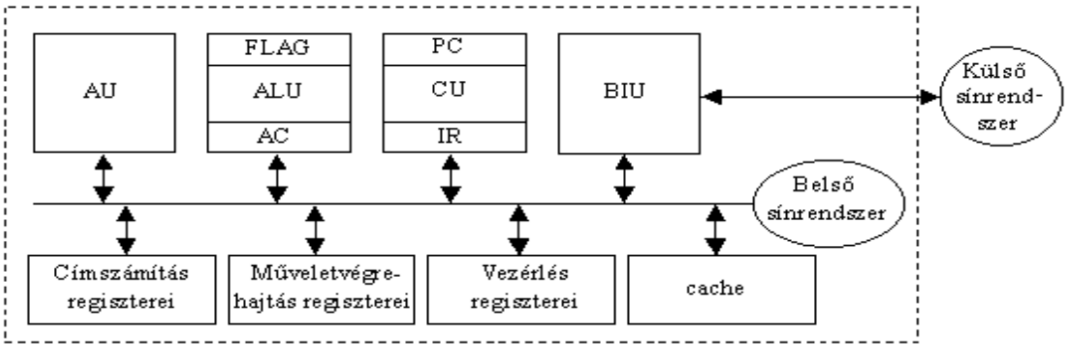
FőbbTárgyak()
{

- Informatikai alapok,
- Számítógép-architektúrák I.,
- Operációs rendszerek,
- Számítógép-hálózatok;

}

1. A processzor felépítése, utasításkészlete. Utasítások szerkezete, címzési módok. Utasításszámláló- és utasítás-regiszter. Az utasítás-feldolgozás elemi lépései

A processzor a számítógép központi feldolgozó egysége (CPU). A processzor egyik fő része az **ALU** (aritmetikai és logikai egység), ez az egység felelős a numerikus (pl. összeadás, kivonás) és logikai műveletek (pl. ÉS, VAGY) elvégzésért. A **CPU vezérlőegysége**, a **CU** vezérli a számítógép részegységeit (pl. perifériák megszakításkérelme). A CPU meghatározó részei még a **regiszterek**, amelyek kisméretű, gyors elérésű, írható-olvasható, állapottárolásokra használt memóriaáramkörök. Ezekből több is található a processzoron belül, nagyságrendileg akár 1000 darab általános célú regisztert is tartalmazhat. A processzor fontos része még a **cache-memória**, ami kis tárkapacitású és szintén nagyon gyors átmeneti tároló. Az egységek a belső-sínrendszert használják. A **buszillesztő** (buszinterfész) biztosítja a processzor kapcsolódását a külső sínrendszerhez. (*1. ábra*)



1. ábra – a processzorfelépítés logikai sémája

Utasításkészlet az elemi szintű, gépi kódú utasítások összessége, amelyek végrehajtására a processzor hardver szinten képes, és amelyre a fordítóprogram a programokat lefordítja. A komplex utasításkészlettel rendelkező számítógépek (**CISC**) sok olyan, specializált utasítással bírnak, melyeket a programok csak ritkán használnak. A csökkentett utasításkészletű számítógépeket (**RISC**) úgy egyszerűsítették le, hogy csak azokat az utasításokat valósították meg bennük, melyek gyakran előfordulnak a programokban.

Utasításszerkezet alatt a gépi kódú, a processzor számára közvetlenül értelmezhető utasítások felépítését értjük. Megadja azt, hogy az utasítás egyes részeit hogyan kell a processzornak értelmezni. Az utasításszerkezet a ***műveleti részből*** (meghatározza a művelet típusát), az ***operandushivatkozásokból*** (*címrész*; milyen operandusokkal, *kifejezéstagokkal* kell a műveletet végrehajtani és ezek hol találhatók) és a ***kiegészítő részből*** áll.

A **címzési mód** lehet ***közvetlen*** (direkt) címzés, amely esetében az utasításban maga a tárolóhelycím található, ezen belül megkülönböztetünk *abszolút* (tényleges címet tartalmaz) és *relatív* címzési módot (valamilyen alapcímhez viszonyított cím) is; a ***közvetett*** (indirekt) címzés, amelynél az utasításban megcímzett tárolóhelyen nem az operandus található, hanem annak a címe. A literális vagy álcímzés (literal,

pseudo, immediate), azaz a ***közvetlen adatcímzés*** esetében magában az utasításban található az operandus. Létezik még az ***indexelt*** címzésátalakítás is (az adatsorozatok helye a végzett műveletek előtt „beolvasásra", indexelésre kerül). A **veremcímzés** esetén valamilyen verem kerül megcímzésre.

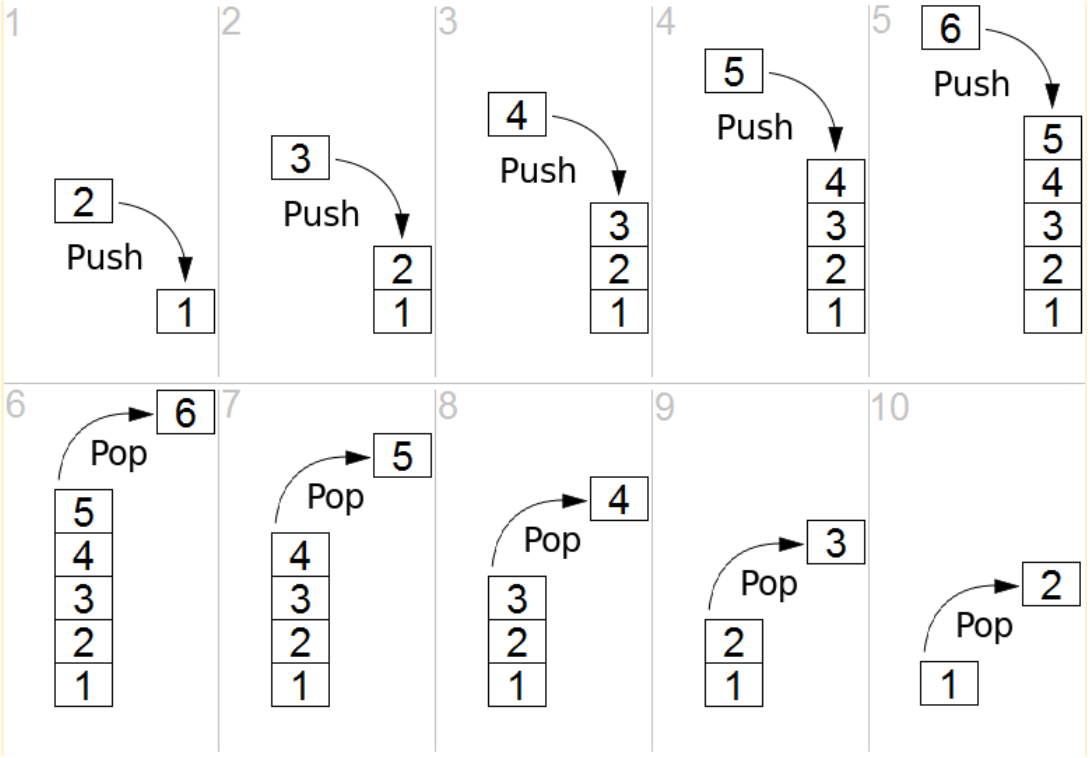
Az **utasításfeldolgozás**, utasításvégrehajtás normál gépi ciklusban történik A processzor a CU-n belüli **utasításszámláló (PC)** tartalma alapján kikeresi az utasítást, és átviszi a vezérlőegység **utasításregiszterébe (IR)**. Ezután a processzor meghatározza az operandus címét, előkészíti az adatokat a kijelölt művelethez. Ennek során tehát a CPU kiolvassa az utasítást, dekódolja, órajelre végrehajtja, majd az eredményt beírja a memóriába. A processzor órajelre hajtja végre az utasításokat, pl., ha 2GHz-es egy processzor, az azt jelenti, hogy másodpercenként 2 milliárd elemi műveletet képes végrehajtani.

2. A verem fogalma és működése, a veremmutató regiszter. A vermet kezelő utasítások. A verem alkalmazása szubrutinok kezelésében. A szubrutinra vonatkozó utasítások

A verem valamilyen **véges számú azonos típusú elem tárolására alkalmas**. A verem a számítástechnikában **lehet szoftveres**, pl. egy programozáskor használt **adatszerkezet** az adatok tárolására, **de létezik hardveres, konkrét megvalósítása is**. Különleges jellemzője, hogy a benne tárolt adatokat nem lehet csak úgy tetszőlegesen kezelni, vagy csak a verem tetején vagy alján lévő adatot lehet elérni.

Alapvetően két féle verem-megoldás létezik, a **LIFO** esetében (*last in, first out*) az utoljára a verembe helyezett elemet lehet elsőként kivenni. A **FIFO**-verem esetén (*first in, first out*) pedig ahhoz az elemhez lehet elsőként hozzáférni, amit legelsőként tettünk a verembe. A szemléltetés és egy ócska példa kedvéért, ha veszünk egy üres *Pilóta kekszes*

zacskót, amit csak az egyik végén vágtunk ki, majd egyesével belerakjuk a kekszeket akkor az kvázi LIFO verem lesz, mert az utoljára beletett kekszet tudjuk elsőként kivenni.



2. ábra – A verem működése

Veremkezelő utasításból kettőt ismerünk: van a **push** és a **pop**. LIFO-verem esetén push utasításkor új elem kerül a verem tetejére, pop utasítás esetén pedig kivesszük a verem legfelső elemét.

A verem a memóriában helyezkedik el, éppen ezért a címe, mérete változó, ezért szükség van egy **veremmutató regiszterre**, ami egy pointerként mutat a verem aktuális memóriacímére, hogy mindig el tudjuk érni a verem tartalmát.

A veremnek a **szubrutinok feldolgozásakor nagy szerepe van**, a processzor azokat a **memóriacímeket menti el** verembe, ahova az

egyes eljárások befejeztével visszatér, illetve a szubrutinok kezdetén elmenti azokat a **regisztereket, amelyeknek az értéke meg fog változni** a végrehajtás során.

Szubrutinokra vonatkozó utasítások a **CALL** (hívás) és **RET** (visszatérés).

3. A Neumann-elvek. Utasítás- és adatfolyam (SISD-, SIMD-, MISD- és MIMD-architektúrák). Adatok számítógépes ábrázolása (fixpontos, lebegőpontos, BCD, vektoros adatok, karakterek)

3.1. Neumann-elvek

Egyszerűbb felsorolás:

- A számítógép működését **tárolt program** vezérli;
- A vezérlés **control-flow** (vezérlésáramlásos) rendszerű, ami azt jelenti, hogy a gép a program utasításait egymás után sorban hajtja végre;
- A gép **belső tároló**jában a program utasításai és a végrehajtásukhoz szükséges adatok egyaránt megtalálhatók;
- Az aritmetikai és logikai műveletek (programutasítások) végrehajtását **önálló részegység** (ALU = Arithmetic Logic Unit) végzi;
- Az adatok és programok beolvasására, tárolására és megjelenítésére önálló egységek (**perifériák**) szolgálnak

Általános, bővebb felsorolás:

1) A számítógépben legyen **soros az utasítás-végrehajtás** (control-flow; vezérlésáramlásos rendszer). Ez azt jelenti, hogy a gépnek a műveleteket egyenként, egymás után kell végrehajtania.

2) A számítógép a **kettes számrendszert** használja. Ezt a számrendszert és az ezen értelmezett logikai és aritmetikai műveleteket rendkívül egyszerű ábrázolni a digitális áramkörökön.
3) A számítógép legyen **teljesen elektronikus**.
4) Legyen **univerzális**, úgynevezett Turing-gép. Alan M. Turing, angol matematikus publikációja bebizonyította, hogy amennyiben egy számítógép el tud végezni néhány alapműveletet, akkor tulajdonképpen bármilyen bonyolult feladat számítását meg tudja oldani.
5) (**Tárolt program elve** –) a számítógép az adatokat és a programokat egy helyen, a belső memóriában (operatív tár) tárolja. A központi egység innen tölti be az adatokat, majd feldolgozás után az eredményeket visszaírja.

Egy alapvető számítógéparchitektúrának az alábbiakat kell tartalmaznia:

- **ALU** (aritmetikai és logikai egység) és regiszterek;
- **vezérlőegység**, ami tartalmazza a *programszámlálót* (PC) és az *utasításregisztert* (IR);
- **operatív tár** az adatok és az utasítások tárolására;
- **háttértár** és a **perifériákhoz** tartozó *be- és kiviteli* mechanizmusok

(Másfajta felosztás Dr. Petrik Olivér lejegyzése nyomán:)

- 1. Az utasításoknak az adatokkal azonos módon (azaz egyetlen nagy kapacitású memóriában), numerikus kódok formájában való tárolása.
- 2. A kettes számrendszer alkalmazása.
- 3. Szükséges egy vezérlőegység, amely különbséget tud tenni utasítás és adat között, majd önműködően végrehajtatja az utasításokat.

- 4. Teljesen elektronikus (tehát nagyon gyors) működés.
- 5. A számítógép tartalmazzon olyan számológépet, amely képes elvégezni az alapvető logikai műveleteket is.
- 6. Szükség van olyan ki/bemeneti egységekre, amelyek áthidalják a bizonyos szempontból lassú emberi kommunikációs csatorna és a gyors számítógép közti sebességkülönbséget.
- 7. Megbízhatatlan alkatrészekből is lehet megbízható rendszert konstruálni.

A **SISD, SIMD, MISD és MIMD** rövidítések arra vonatkozóan, hogy egyidejűleg *hány utasítás- és adatfolyamot* képes kezelni az architektúra.

SISD = single instruction, single data stream = egy utasítás, egy adatfolyam

MIMD = multiple instruction, multiple data stream = több utasítás, több adatfolyam A rövidítések magát magyarázzák, ha megjegyzed, mi mit rövidít…

A **SIMD** (Single Instruction Stream Multiple Data Stream) típusú számítógép egyetlen utasításfolyammal többszörös adatfolyamot dolgoz fel. Ezek a számítógépek több párhuzamos, egy idejű működésre képes műveletvégző egységet (ALU) tartalmaznak. Vektorműveleteket képesek végrehajtani gépiutasítás-szinten.

A **MISD** (Multiple Instruction Stream Single Data Stream) típusú számítógép több utasításfolyammal egyetlen adatfolyamot dolgoz fel, gyakorlatban ilyen gépek nemléteznek.

Leegyszerűsítve **a fixpontos számábrázolást** egész számok tárolására használjuk. Azért fix, mert a tizedespont helye fixen a szám végén van, pl: 123.

A **lebegőpontos** (float) számot törtszámok, valós számok tárolására is alkalmasak. Azért lebegőpontos, mert a tizedespont "lebeg", a számértéken bárhova kerülhet, pl: 1.23 12.3 123.

A BCD-kódolás (**binárisan kódolt decimális** – Binary-Coded Decimal) a decimális (10-es számrendszer-beli) számok kódolási formája, minden számjegyet egy bitsorozat ábrázol, pl: 82: 1000 0010.

Vektoros adatok, például egy grafika tárolásakor geomatriai primitíveket: pontokat, egyeneseket, görbéket, sokszögeket használunk a kép leírására. Például egy egyenest vektorgrafikusan úgy tárolunk, hogy a kezdő és a végpontot tároljuk, csak mert ebből bármikor előállítható az egyenes bármekkora méretben. Ennek ellentéte a rasztargrafika, ahol pixelekkel írjuk le az egész egyenest és ebből adódan ha jól belenagyítunk akkor jó "pixeles", szemcsézett lesz a képünk. A vektor-processzorok (ha egy processzor utasításkészlete tartalmaz ilyet) nagyon meg tudnak gyorsítani egyes munkafolyamatokat. Ma gyakorlatilag összes korszerű processzor tudja ezt...

Karakterek kódolására többféle szabvány is létezik, Legelterjedtebb kettő az **ASCII** és az **Unicode** (pl: UTF-8 kódolás). Az ASCII mindössze 128db különböző karaktert tud reprezentálni, ez arra elég volt, hogy lefedje az angol ABC-t, a fennmaradó karakterek pedig vezérlő funkciót töltenek be (pl: DEL = delete, BS = backspace).

Az Unicode több mint 100.000 karaktert tartalmaz, amellyel gyakorlatilag a valaha volt összes modern és történelmi abc-t lefedi. Azért Unicode a neve, mert univerzális kódolási ipari szabványként hozták létre. Szoftverfejlesztői oldalról nézve rengeteg szívástól megkíméli a fejlesztőket az Unicode. Használatával kivédhetőek a bosszantó karakterkódolási hibák, pl. tipikusan az 'ő' 'ű' betűnél szétcsúszik egy rakás másik kódolási szabvány, és lehetne sorolni…

4. Az utasítás-feldolgozás gyorsítása párhuzamosítással. A pipelining lényege, szuperskalár processzorok. Fellépő problémák és kezelésük

A *párhuzamos utasításfeldolgozásnak* az a nagy előnye, hogy az arra alkalmas programoknak a *végrehajtását* ***meggyorsítja***. Sajnos a legtöbb alkalmazás nem tudja kihasználni a párhuzamosítás előnyeit. Például ha van egy 2 magos és egy 8 magos processzor, akkor a 8 magoson nem fognak 4x gyorsabban végrehajtódni a programok, ez csak egy elméleti maximum...

Körülbelül a 70-es években jöttek rá, hogy egy feladat megoldását nem csak szekvenciális módon lehet meghatározni. A soros feldolgozásnál időben egymás után történik a feldolgozás. Párhuzamos feldolgozás esetén időben egyszerre több szálon is folyhat a végrehajtás. Általánosan elmondható, hogy bonyolultabb feladatok esetén azok komplexitását párhuzamos feldolgozással lehet hatékonyabban megoldani.

A **pipelining** (futószalagos feldolgozás) neve onnan adódik, hogy a gyárban futószalagos termelésnél minden munkás ***csak egy-egy részletmunkát végezhet***, a szakmunka betanított munkává alakul, gépszerű, egyszerű műveletekből áll. A processzorban is pont ugyanez történik, csak itt a futószalag egy adott végrehajtó-egység, például egy fixpontos vagy lebegőpontos művelet-végrehajtó egység. Az egyes részműveleteket ***időben párhuzamosan*** hajtjuk végre. Ha az adott hardveregység felszabadul, akkor ezt igénybe vehetjük egy következő utasítás elemi lépésének végrehajtására.

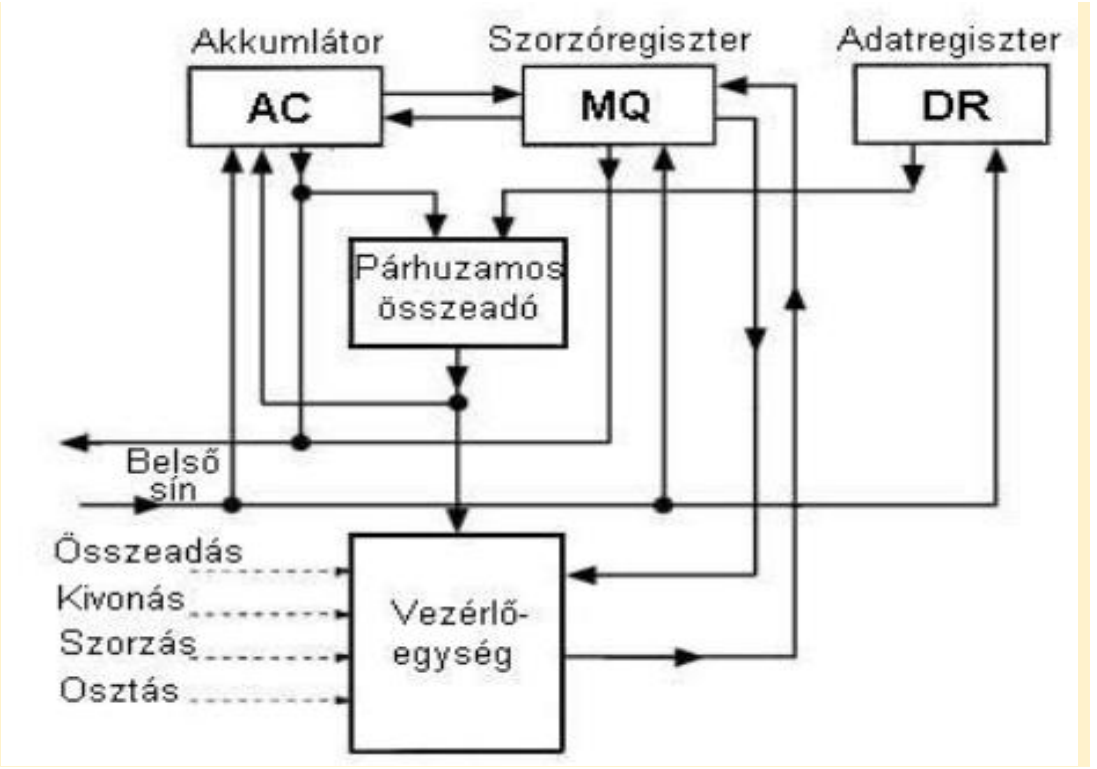
Ha **egy gépi ciklus alatt** egy processzor **több utasítást** is képes végrehajtani, azt ***szuperskalár*** processzornak nevezzük. Mivel több szálon történik a végrehajtás, nagyon fontos ezek szinkronizációja. (Ennek megvalósítása nagy kihívás hardveresen és szoftveresen programozás közben is, ha szálkezeléssel foglalkozunk...)

Egy tipikusan fellépő probléma: vannak olyan típusú utasítások, amiknek teljesen végig kell haladnia a futószalagon, mielőtt a végrehajtás folytatódhat. Különösen feltételes elágazásoknak kell tudniuk egy előzetes utasítás eredményét, így eldönthető, hogy melyik ágat kell választani. Például, egy olyan utasításban, ami azt mondja „ha x nagyobb mint 5, akkor tedd ezt, különben tedd azt" először meg kell várni, amíg az x változó konkrét értéket kap, mielőtt eldönthető lenne, hogy ezután melyik ágon kell folytatni a végrehajtást.

A megoldás, vagy az egyik lehetséges megoldás a **spekulatív végrehajtás**, ami *elágazásbecslésként* van nyilvántartva. Valóságban az elágazás egyik oldala sokkal gyakrabban lesz meghívva, mint a másik, így sok esetben helyes, ha egyszerűen azt mondjuk: „x valószínűleg kisebb lesz ötnél, kezdd el annak a végrehajtását". Ha a jóslatról kiderül, hogy helyes, hatalmas mennyiségű időt meg tudunk spórolni. A modern processzorkialakításokban már meglehetősen összetett jóslóelőrejelző rendszereket alkalmaznak, amik figyelembe veszik a korábban végrehajtott elágazások eredményeit, hogy nagyobb pontossággal jósolják meg a jövőt. Gyakorlatilag amíg a processzor vár az adatra, az üresjáratban előre elkezd kiszámolni olyan ágakat, amiken majd feltételezhetően folytatódni fog a végrehajtás.

5. Az aritmetikai-logikai egység és regiszterei (akkumulátor, flag). Fixpontos és lebegőpontos műveletek, ezek végrehajtásának egységei. Logikai műveletek

Az **ALU** (Arithmetic and Logic Unit) a processzor egy részegysége. Ez az egység felelős a numerikus (pl. összeadás, kivonás) és logikai műveletek (pl. ÉS, VAGY) elvégzéséért. Az összeadás-kivonás mellett képes fixpontos és lebegőpontos szorzásra és osztásra is.



3. ábra – A fixpontos ALU felépítése

Az ALU a következő logikai áramkörökből épül fel: ***összeadó-egységekből***, amelyek alapeleme két 1 bites fixpontos adatelem összeadását végrehajtó áramkör); ***léptető-áramkörökből*** (az operandusokat tároló hardverregiszterek jobbra vagy balra léptetését hajtják végre); ***logikai*** műveleteket végrehajtó áramkörökből; az operandusokat, illetve eredményeket tároló átmeneti tárolókból, amelyek közül legfontosabb az ***akkumulátor*** (AC) nevű regiszter.

Leegyszerűsítve, a fixpontos számábrázolást egész számok tárolására használjuk. Azért fix, mert a tizedespont helye fixen a szám végén van, pl: 123. A lebegőpontos (float) számok törtszámok, valós számok tárolására is alkalmasak. Azért lebegőpontos, mert a tizedespont "lebeg", a számon belül bárhova kerülhet, pl: 1.23 12.3 123. Két fixpontos vagy lebegőpontos szám között végezhető műveletek (összeadás,

kivonás, szorzás, osztás) a kettes számrendszer szabályai szerint bitenként történik.

Az ALU ***flagjei*** 1 biten 0 vagy 1 értéket vehetnek fel. Az *Overflow* flag (túlcsordulás) akkor billen 1-be, ha például egy szorzás után az eredmény nagyobb, mint a maximális tárolható érték. A *Sign* flag (avagy *Negative* flag) az *előjelet* tárolja, és akkor billen 1-be, ha negatív a végrehajtás eredménye. Ezen kívül ismerünk még *Zero*, *Carry* stb... flageket.

A *logikai műveletek* eredménye mindig vagy IGAZ vagy HAMIS. Logikai kapu például az AND, OR, NOT (negálás), XOR (kizáró vagy). Igazságtáblán egyszerűen ábrázolhatóak... Ezeknek a kapuknak hardveres megvalósítása tranzisztorokkal lehetséges; manapság már elképesztően parányi, nagyságrendileg nanométeres (a *mm* egymilliárdod része) méretekben.

6. A vezérlőegység feladata és jelei, vezérlési pontok. Huzalozott és mikroprogramozott műveleti vezérlés. CISC és RISC processzorok

A **vezérlőegység** (CU – control unit) a processzor egyik egysége, feladata *a teljes számítógép részegységeinek* (aritmetikai egység, memória, kommunikációs eszközök, háttértár és perifériavezérlések) *irányítása*, összehangolása.

A vezérlőegység jelei gyakorlatilag ***négyszögjelek***, lehet a processzor belső vezérlőjele, ami a processzor részegységeit koordinálja, illetve vannak a külső vezérlőjelek, amik a memória-, I/O-kezelést hangolják össze.

Huzalozott műveleti vezérlés esetén egy *kombinációs hálózatból* (különféle kapuk összekapcsolva egymással egy nagy hálózatban)

alakítják ki fizikailag a vezérlést. Ennek az az előnye, hogy nagyon gyors, viszont gyakorlatilag karbantarthatatlan a fix architektúra miatt. Egy kis módosítási igény is nagy kihatással lehet az egész kombinációs hálózat logikájára, szélsőséges esetben akár újra kell tervezni az egész kapu-hálózatot. Ez a vezérlés már a múlté… Helyette amit napjainkban is használunk, az a ***mikroprogramozott műveleti vezérlés***. Alacsony szintű szoftverkód felel a műveleti vezérlésért, ezt könnyebb megtervezni, implementálni és karbantartani is.

A processzoroknál alapvetően két különböző felépítést különböztetünk meg: a ***CISC***- (Complex Instruction Set Computer = összetett utasításkészlet) és a ***RISC-architektúrát*** (Reduced Instruction Set Computer = csökkentett utasításkészlet); a két technológiával készült processzoroknak különbözik az utasításkészletük.

CISC – nagyszámú utasítást tartalmazó utasításkészlettel rendelkeznek, az utasítások szerkezete bonyolult, többfajta memóriacímzési módot tesznek lehetővé, az utasításvégrehajtás mikroprogramvezérelt.

- ❖ Változó hosszúságú utasítások;
- ❖ Sokféle utasítás és címzési mód;
- ❖ Bonyolult mikroprogram, egyszerű fordítóprogram.
- ❖ Kisszámú regiszter.

RISC – utasításkészlete csökkentett, egyszerűsített, a memóriahozzáférés csak két utasítással: memóriából való *adatbetöltés* (LOAD) és memóriába való *adatírás* (STORE) történhet, a műveleti vezérlés huzalozott (azaz beépített áramkörökkel végrehajtott) vagy horizontális (egyszerűsített) mikroprogram-vezérelt. A csökkentett utasításkészletű számítógépeket (RISC) úgy egyszerűsítették le, hogy csak azokat az utasításokat valósították meg bennük, melyek gyakran előfordulnak a programokban.

- ❖ Egyszerű utasítások, amelyek végrehajtása 1 gépi ciklust igényel;
- ❖ Erőteljes futószalag (pipelining) feldolgozás;
- ❖ Rögzített utasításhossz;
- ❖ Nagyméretű regisztertár;
- ❖ Bonyolult fordítóprogram, egyszerű mikroprogram

7. A központi tár szerepe, áramköri megvalósítása. ROM és RAM áramkörök típusai. Dinamikus RAM belső felépítése. Átlapolt memóriakezelés

Háttértárat (mass storage) nagy mennyiségű adatállomány és az utasítások-programok tárolására használjuk. Áramkörileg úgy van kialakítva, hogy akkor is megőrzi az adatot, amikor az eszköz nincs áram alatt (pl. mágneses tár: merevlemez). Napjainkban tipikusan merevlemezt (HDD) vagy SSD-meghajtót használnak erre a célra.

Ezen kívül van az **operatív tár** (más néven főtár vagy központi tár), amit egyszerűen csak memóriának is szoktak becézni. Azokat az utasításokat és adatokat, amikkel éppen dolgozik a processzor, itt tároljuk, hogy az könnyen és gyorsan el tudja érni az éppen használat alatt levő adatokat. Attól függően, hogy a memóriát milyen feladat megoldására használjuk, van több fő típusa:

A **RAM** memória írható-olvasható gyors memória, a számítógép kikapcsolásakor a RAM tartalma elvész. Itt tipikusan csak az aktuálisan futó programok és az azok által feldolgozandó adat tárolódik. Altípusai a statikus SRAM (tipikusan cache memóriákhoz használt, processzoron belül) és a dinamikus DRAM, ami kvázi egy tipikus számítógép DDR3 vagy DDR4 memória-moduljának felel meg. A dinamikus RAM-ban 1 bit tárolására szolgáló cella 1 kondenzátorból és 1 tranzisztorból épül fel. A töltéseket apró kondenzátorok tárolják.

A **ROM** memória csak olvasható memória, például ROM-memórián tárolja az alaplap a *firmware*-t (BIOS-t, korszerűbb gépeken UEFI-t). ROM-ból léteznak különféle altípusok: a PROM egy egyszer programozható memória, utána a tartalma nem változtatható. Vagy van az EPROM, ami egy törölhető (erase) és újraprogramozható memória (elekronikusan törölhető: EEPROM).

Az **átlapolt memóriakezelés** ötlete a relatív lassú DRAM-ok miatt jelent meg. A memória függetlenül címezhető kis egységekből, *memory bank*-okból (memóriatömbökből) áll, ezek mindegyikét külön-külön a többitől függetlenül el lehet érni. A 0. memóriabankból kiolvasott adat hozzáférése alatt az 1. memóriabankban, ***a következő címen lévő adat már megcímezhető***. Ez kissé leegyszerűsítve azt jelenti, hogy cím folytonos olvasás esetén az adatok kiolvasása kb. kétszeres sebességgel történhet.

8. Gyorsítótárak (cache) feladata és működési elve. Cache-tárak felépítése és típusai. Helyettesítési és adataktualizálási stratégiák

A modern (értsd: az elmúlt bő húsz évben készült) processzorok esetében komoly kihívás a végrehajtóegységek folyamatos „etetése" adattal. A rendszermemória ugyanis egyszerűen túl lassú, az adatok bekérésétől számítva mintegy 100 órajelciklusra van szükség ahhoz, hogy azok meg is érkezzenek, vagyis a processzor szinte mindig üresben járna, ha erre várna. Hogy ezt elkerüljék, a processzorgyártók komplex ***prediktív*** (előbecslő) ***algoritmusokat*** fejlesztettek, amelyek igyekeznek a potenciálisan szükséges adatokat a gyorsítótárba, avagy cache memóriába tölteni.

Általában *több szinten* valósítják meg a **cache**-t: van L1 (azaz level 1), illetve L2 másodszintű (meg L3 stb.) cache. Az L2 cache méretében nagyobb, de lassabb is, mint az L1-es. A cache-memória *drága*. Míg

a DRAM mérete korszerű gépeken nagyságrendileg több GB, addig a cache memória mérete tipikusan csak néhány MB méretű. A cache memóriapuffert tipikusan a processzorba tokozzák bele, ez nem egy külön megvásárolható egység, mint a DRAM. A cache belső felépítését úgy lehet elképzelni, mint sorokat, ahol minden sor áll a *címrészből*, a *vezérlő-részből* és az *adatrészből*. A processzor az adott címrész alapján az *adatrészbe* írja ki azt az adatot, ami ***megváltozott*** a műveletvégzés során.

A **helyettesítési stratégiák** lényege az, hogy a legrégebben használt cache blokkok helyére írjuk be az új adatot. Erre azért van szükség, mert a cache mérete kicsi, könnyen telítődik. **Demand fetching** esetén csak a processzor igényére kér be új adatot a cachebe, **prefetching** esetén pedig egy blokk betöltése esetén előre az utána lévő blokkokat is automatikusan becacheli, hátha kelleni fog…

Az **adataktualizálási stratégiák elve** arról szól, hogy ha a processzor a műveletvégzés során megváltoztat egy adatot a cacheben, akkor azt mihamarabb *ki kell írni a főtárba is*, mielőtt az adott cache-blokk *felülíródna*. Tipikusan a gyorsítótárban módosított adat csak akkor kerül visszamásolásra a főtárba, ha a cache-nek *a módosított adatot* tartalmazó sorát felül kell írni a főtárból *bemásolandó* újabb blokkal.

9. A virtuális tárkezelés fogalma és legfontosabb eljárásai (lapozás és szegmentálás, a virtuális cím leképezése, TLB, lapcsere stratégiák)

Egy program végrehajtásához a megfelelő programrésznek és az általa feldolgozott adatoknak a memóriában (DRAM) kell lenniük. A mai korszerű számítógépek memória mérete valahol 4–32GB között mozog. Ezeknek a gyors memóriamoduloknak az ára 1GB-ra leosztva sokkal drágább, mint mondjuk a háttértárak árai, ezért az ár-érték-

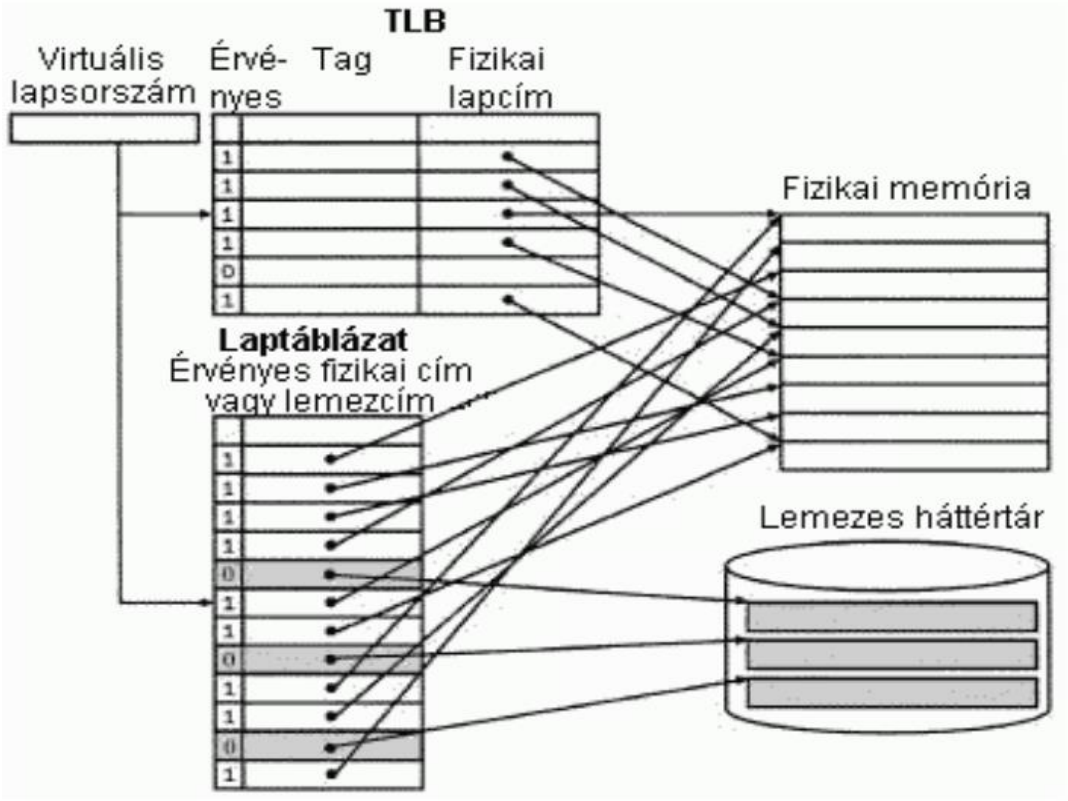
arány miatt nem praktikus a memória mennyiségének eszetlen növelése. A **virtuális tárkezelés** (lapozótár, **lapozófájl**) lényege, hogy a relatív lassú, de nagy tárhely-méretű háttértár (HDD, SSD) egy elkülönített részén tároljuk az aktuálisan nem futtatott programokat és a hozzájuk tartozó állományokat, és csak akkor töltjük be a központi memóriába, ha szükség van rájuk. Pl. egy nagyon erőforrásigényes műveletnél: videóvágáskor, ha elindítjuk a renderelést, akkor sokszor nincs az a memóriamennyiség, amit ne tudna megtölteni az operációs rendszer. Ilyenkor a virtuális tárra írja ki azokat a blokkokat, amik a memóriába már nem férnek bele.

A virtuális tárkezelésből a felhasználó semmit nem vesz észre, ez a háttérben teljesen rejtett módon történik, hogy a memória fizikai korlátai észrevétlenek legyenek.

Lapozás-eljárás esetén a virtuális tár rögzített méretű, *nem átlapolható* blokkokból áll, ezeket **lapoknak** nevezzük. Ezzel gyakorlatilag egy olyan struktúrát alakítunk ki, mint ahogy az adatok a memóriában is tárolódnak (a fix méretű blokkokban). Mivel a virtuális táron hasonló struktúrában tároljuk az adatokat, azokat olcsóbb művelet visszamozgatni a tényleges memóriába.

A **szegmentált eljárás** esetén ***a blokkok mérete nem rögzített***, ilyenkor a blokkokat **szegmenseknek** nevezzük. A szegmensek átlapolódóan is megadhatók, azaz ***ugyanaz az adat két különböző szegmensen belül is megcímezhető***. Ez felveti a töredezettség (fragmentáció) problémáját, amikor üres helyek, illetve duplikációk jönnek létre a háttértáron.

A **TLB** (translation lookaside buffer) egy **memóripuffer**, a legfrissebben használt lapok címfordításhoz szükséges adatait tartalmazza, tehát lényegében egy *címfordító cache*. A TLB a processzor és a cache tároló között helyezkedik el, mivel a programok *virtuális*, a cache pedig *fizikai címeket* tartalmaznak.



4. ábra – A TLB felépítése

A processzornak a műveletvégrehajtás során a memória *valódi*, vagy másképpen nevezve **fizikai** címeire van szüksége, tehát szükség van a virtuális címek fizikai címre való leképzésére. Mai gépekben ezt egy címleképezési áramköröket tartalmazó hardver egység végzi.

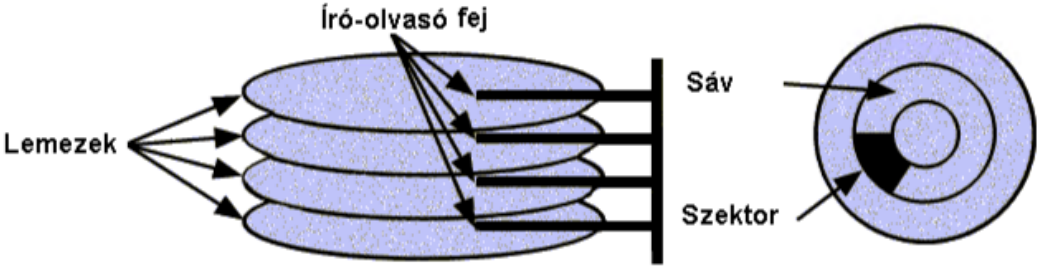
Különféle **lapcsere-stratégiák** állnak rendelkezésre, ha új lapot szeretnénk beilleszteni. A lapot beszúrhatjuk a memória első szabad helyére, vagy a legutoljára betöltött lap utáni következő szabad helyre, vagy a legjobb helyre, hogy a beszúrás után a legkevesebb szabad tárterület maradjon utána.

10. Az adatrögzítés elve a mágneses háttértárolókon. A merevlemez fizikai felépítése (szektor, sáv, cilinder) és logikai felépítése (klaszter, FAT, bootszektor). A merevlemezes egység teljesítményjellemzői (elérési idő, adatátviteli sebesség)

A merevlemez **mágneses elven** működő tárolóeszköz. Tartós tár, tehát a számítógép kikapcsolása, áramtalanítása után is megőrzi a rá mentett adatot. A számítástechnikában a merevlemez (HDD) az egyik legolcsóbb tár, ha az 1GB-ra eső árát nézzük. A winchester fémházának belsejében *korong alakú lemezeket* találunk egy tengelyre fűzve. Ezeken a lemezeken egységnyi mágnesezhető részecskék vannak. A tengelyre fűzött lemezeket egy motor forgatja nagy sebességgel (asztali gépekben tipikusan 7200 fordulat percenként), miközben egy író-/olvasó-fej az egyes mágnesezhető részegységek fölé közel mozogva elektromágneses gerjesztéssel megváltoztatja az egységek mágnesességének irányát vagy egyik, vagy a másik irányban. Ez ***a kétféle mágnesállapot*** (true/false) megalapozza a bináris adattárolás lehetőségét.

A HDD-beli *lemezeket azonos központú, különböző sugarú körök tagolják*, ezeket **sávoknak** nevezzük. Azokat a sávokat, *melyek egymás alatt* helyezkednek el, **cilindernek** nevezzük. A kör-sávokat tovább lehet bontani ún. **szektorokra**. A sávokat, cilindereket és szektorokat *számozzuk*, ez alapján vannak nyilvántartva. A HDD-meghajtó formázása (particionálása) közben *csoportosíthatjuk* a szektorokat, így alakíthatunk ki helyfoglalási egységeket, azaz **clustereket** (szektorcsoportokat). Ezek *a legkisebb címezhető egységek* a HDD-ben, legkisebb mérete 512byte lehet (1db szektor) a legnagyobb pedig 64kb (128db szektor).

Az adott merevlemezen levő **logikai** felépítés a rajta (többek közt partícionálással, formázással) kialakított ***fájlrendszertől*** függ.



5. ábra – A merevlemez logikai elrendezése

A *FAT* (file allocation table) egy szabványos fájlrendszercsaládnak a neve. a FAT16 és a FAT32 is elavult manapság, szűkül a felhasználási köre, de ennek ellenére széles körben támogatott az operációs rendszerek között. Manapság az *NTFS*, az exFat, HFS+, APFS tekinthető korszerű fájlrendszerek. FAT32-re formázott pendrivera pl. nem lehet rámásolni 4GB-nál nagyobb méretű fájlt, pl. filmeket mert (maga a partícióstruktúra) képtelen tárolni ekkora fájlt.

Egy bootolható (rendszerinditó) partíción belül az első szektor(oka)t rendszerbetöltő, másnéven ***bootszektornak*** nevezik.

(**Sebesség**:) A merevlemez esetében – mivel egy mechanikus eszköz – különböző ***késleltetésekkel*** lehet számolni. Ezek a fejmozgatási idő, forgatási idő, adatátviteli idő, stb… Nagyságrendileg néhány *ms* hosszúságúak ezek a késleltetések. Modern merevlemezek nagyjából 80-130MB/s sebességgel képesek adatokat írni-olvasni SATA 3-as csatlakozón. A jövő egyértelműen az SSD meghajtóké, ahogy egyre olcsóbbak lesznek…

11. A megszakítási rendszer (megszakítások típusai, a megszakítás kiszolgálása, vektortáblázat) és alkalmazásai. A megszakításvezérlő feladatai

A **programmegszakítás** (interrupt) azt jelenti, hogy a program futása egy ***külső vagy belső esemény bekövetkezte miatt megszakad***, és majd csak *a progammegszakítás kiszolgálását végző kód lefutása után* tér vissza a CPU az eredeti program folytatásához.

Megszakítás történhet akkor, amikor egy periféria jelzi, hogy egy input-/output-műveletet befejezett, tehát *egy meghatározott művelet befejezésekor*. De megszakítás az is, amikor valamilyen géphiba miatt egy áramköri jelzést küld a gép, vagy ha külső forrásként megnyomjuk az újraindítás gombot a számítógépen.

A megszakításkezelés ***aszinkron*** feldolgozású, mert időben *nem megjósolható*, hogy egy megszakítást kiváltó esemény (például egy billentyű leütése) mikor fog megtörténni.

A ***vektortáblázat*** a megszakításokat kiszolgáló *rutinok* kezdőcímeit tartalmazza. A megszakítás kiszolgálása úgy zajlik, hogy a megszakításkérelemre a processzor a vektortáblázatból megkapja *a megszakítási elem sorszámát*, majd ***elmenti az utasításszámláló és állapotregiszterek*** aktuális tartalmát. Végrehajtja a megszakítást kiszolgáló rutint, majd a processzor visszatölti az elmentett regiszter-tartalmakat, és folytatja a megszakított program végrehajtását.

A ***megszakításvezérlő*** feladatai – többek között –, hogy fogadja a megszakítási kérelmeket, vizsgálja, hogy nincs-e *maszkolással tiltva*, ***priorizálja*** a megszakítás fontosságát, valamint *közli* a megszakításkérést a processzorral az INT-vezetéken. Ha a processzor kész a kérés fogadására (*IACK* - Interrupt Acknowlegment) akkor a megszakításvezérlő *átadja a processzornak a megszakításvektor címét*.

12. Az I/O adatátvitel típusai. A közvetlen memória-hozzáférés (DMA) lényege és végrehajtása. A DMA-vezérlő regiszterei és működése

Az I/O- (input-/output-)adatátvitelnél megkülönböztetünk több típust az alapján, hogy az adott *input-/output-kérést* hogyan dolgozza fel a számítógép.

- **Polling** esetén a processzor minden meghatározott órajelciklusban (lehetőleg néhány ms-onként) ***ellenőrzi***, hogy volt-e I/O művelet. Ez költséges megoldás, mert állandóan – sok esetben feleslegesen – terheli a processzort.
- **Megszakításos I/O-átvitel** esetén egy *megszakítás-esemény* váltódik ki, például arról, hogy lenyomtam egy billentyűt, és ezt a megszakítás vezérlő jelzi a processzornak. Azért hívjuk megszakításnak, mert ilyenkor a processzor megszakítja az éppen végrehajtott programot a megszakítás feldolgozásának idejére, majd tovább folytatja a megszakított programot.
- **DMA** (direct memory access), közvetlen memória hozzáférés esetén az I/O-eszköz és a memória (DRAM) közötti adatátvitelben *a processzor nem vesz részt*. Ezt egy külön egység, a ***DMA-vezérlő*** végzi. Ennek az az előnye, hogy a processzor felszabadul az I/O-műveletek terhe alól. A DMA-adatátvitel végrehajtása abból áll, hogy ***ellenőrizzük*** a perifériát, ***jelezzük*** a buszfoglalási kérelmet, majd a DMA-vezérlő *masterként* ***lefoglalja*** a buszt. Ha kész az adatátvitel, akkor a DMA-vezérlő jelzi a processzornak, majd ***megszűnik*** a buszengedélyezés.

A DMA vezérlő regiszterei a ***címregiszter***, ami az aktuális forrás-/célmemóriacímet tárolja, a ***számlálóregiszter*** a még átvitelre váró adatokat tárolja, a ***módregiszter*** az átvitel irányát tárolja (írás/olvasás),

a ***maszkregiszter*** értelemszerűen a maszkot tárolja (ha van), valamint az állapotregiszter az átvitel státuszát tárolja, például hogy befejeződött-e az átvitel.

13. A sín (busz) feladata, logikai felépítése, típusai. Sínvezérlés (szinkron, aszinkron). Master- és slave-eszközök. Buszarbitráció (soros és párhuzamos sínfoglalás)

A számítógép részegységei közötti kapcsolatokat a ***sínrendszer*** (avagy *buszrendszer*) biztosítja. Ez gyakorlatilag egy *összetett mikrovezeték-hálózat*, ami különböző *sínvezérlő* áramköröket és *aktív* (pl. tranzisztor) és *passzív* (pl. ellenállás) áramköri elemeket is tartalmaz. A sínen címek, adatok és vezérlőjelek utaznak, ezek jellemzően négyszögjelek formájában.

Logikailag szétbontható a ***címsínre***, az ***adatsínre*** és a ***vezérlősínre***, ahol például a megszakítások vezérlőjelei is végighaladnak.

A ***belső sínrendszer*** a *processzoron belül* annak különböző részeit kapcsolja össze, a ***külső*** sínrendszer *a processzort köti össze* a különböző részegységekkel, jellemzően az alaplapon alakítják ki ezt a típusú sínrendszert.

A **szinkron sínvezérlés** onnan kapta a nevét, hogy a sínen kommunikáló eszközök egyszerre, azonos órajelen, azaz *szinkronban ütemezettek*. Az adás-vétel *mindig azonos sebességgel* történik. Problémája az, hogy közös órajelet kell biztosítani az összes sínre kapcsolt eszköz számára.

Aszinkron sínvezérlés esetén az *események tetszőleges időben bekövetkezhetnek*, itt nincs szükség közös órajelre, emiatt nagyon eltérő sebességű eszközök kiszolgálását is lehetővé teszi. Az átvitel során

szükség van egy úgynevezett handshake-re hogy az adás-vétel megtörtént-e (visszaigazolás).

A sínt egy időben csak egy eszközpár használhatja, az aktív kezdeményező fél a ***master***, a másik fél pedig a ***slave*** (egyenes fordításban "szolga") tehát a kommunikációt semmiképpen nem irányítja. A master tipikusan egy DMA-vezérlő, vagy valamilyen processzor szokott lenni, a slave pedig például a memória vagy egy periféria.

Amikor a sín használatát egyszerre több master eszköz is igényli, akkor el kell dönteni, hogy melyik eszköz kapja meg a használatot. Ez az eljárás a **buszarbitráció**.

- ***Soros sínfoglalás*** esetén egy "várólista" alapján egészen egyszerűen sorba állítjuk a sínfoglalási igényeket.
- ***Párhuzamos*** sínfoglalás esetén minden sínhasználatért váró eszköz önálló kérő és engedélyező vezérlővonallal rendelkezik, és a ***buszarbiter*** prioritás szerint engedélyezi a sín igénybevételét.

14. Az I/O eszközvezérlők, interfészek feladata, regiszterei, címzése. Soros és párhuzamos port és adatátvitel. Az adó és vevő szinkronizálása

Az ***I/O-eszköz-vezérlők*** kapcsolják össze a háttértárakat és perifériákat a számítógép *I/O-sínrendszerével*. Ha egy program egy adatot akar pl. beolvasni, akkor az erre vonatkozó parancs az eszközvezérlőn megy keresztül, majd ez vezérli az I/O-eszköz, például a winchester tevékenységét.

Az ***I/O-interfészeket*** legtöbbször az I/O-sínbe behelyezett, például *PCI-Express-kártyaként* (hangkártya, tuner kártya, stb..) valósítják meg. Emellett elhelyezhetők az alaplapon különféle alapvető funkciót

betöltő integrált I/O-interfészek (Ethernet, hang, stb...) amik annyira alapvető dolgok, hogy ezeket az interfészeket már eleve minden PC-alaplapra integrálják, legyen az asztali gép vagy laptop.

Az *I/O-interfészek* **regiszterei** a *parancsregiszter*, az *állapotregiszter* és a *pufferregiszter*. Ezeknek a beszédes neve elárulja, hogy mit tárolnak a műveletvégzés során…

- Soros adatátvitel (pl. USB - Universal Serial Bus) esetén az interfész és a periféria között az adatokat bitenként sorba egymás után visszük át. Mindig két eszköz vesz részt, egyik az adó, a másik a vevő szerepkörét tölti be. A soros adatfolyam értelmezéséhez a vevőnek fel kell ismernie ***az adatbitek határait***, ehhez előírhatunk *egy speciális bitsorozatot*, amit határként ismer fel a vevő. Így biztosítható az adó és a vevő összehangolt működése.
- Párhuzamos adatátvitel (pl. LPT - Linear Print Terminal) esetén az interfész és a periféria között az adatokat bitcsoportonként egyszerre visszük át, ezt párhuzamos adatátvitelnek nevezzük. (Pl. IDE-PATA-csatoló.)

Külön vezeték(ek) szükséges(ek) az adó-vevő-szinkronizmus megvalósítására is.

15. Monitorok típusai, paraméterei, működési elve. A monitorvezérlő interfész feladata, felépítése, jellemzői (felbontás, színmélység, képmemória mérete) és működése

A **katódsugárcsöves monitor** (CRT) mára abszolút elavult technológia. A képernyőre eső *elektronsugár felvillantja a* katódsugárcső előtt elhelyezkedő, foszforréteggel bevont *felület* egy pontját. Ezekből a fénypontokból áll össze a kép.

A **folyadékkristályos monitor** (LCD) esetén *két átlátszó lap közé* szorítanak folyadékkristályt. A feszültség a kijelző minden cellájára külön vezérelhető, aminek hatására a folyadékkristály-molekulák egy bizonyos irányba állnak be, és így több vagy kevesebb fényt eresztenek át.

TFT-monitornál az *aktív mátrixos* technológiát használják. Minden egyes képpontja ***egy saját tranzisztorból*** áll, ami a cella ki- és bekapcsolását gyorsítja.

A ***plazmamonitor*** nem más, mint piciny fénycsövek alkotta mátrix. A fénycsöveg közé semleges gázt (neon, argon, xenon) töltenek. Ha egy cellához tartozó elektródákra magas feszültséget kapcsolnak, akkor a kisülés hatására a cellában lévő gáz átmegy plazmába.

A fenti monitortípusok javarészt *elavultnak* számítanak, a legújabb technológiák a *LED*-háttérvilágítású ***LCD***-, illetve az ***OLED***-es monitorok. Gyakorlatilag már csak ilyen technológiájú TV-készülékeket forgalmaznak... A háttérvilágítás, illetve OLED esetén az adott képpont ***LED-technológiája hosszabb élettartamot és kis fogyasztást*** biztosít. Az OLED-kijelzők jellemzően csak kisebb, például okostelefonokban kapnak helyet, monitornak vagy TV-nek felhasználva az ilyen készülékek egyelőre drágábbak. Az OLED-kijelző rendkívül vékony (akár fél milliméter is lehet), könnyű és hajlékony változatai is léteznek, ami viselhető eszközöknél lehet praktikus. Nem használ háttérvilágítást, ezért a ***fekete*** szín itt tényleg vaksötét! Emellett nagyobb fényerőt tud kibocsátani, mint egy LCD-s monitor és energiatakarékosabb is annál.

A legújabb monitorok esetén fontos paraméter-értékek lehetnek a felbontás, a kontrasztarány, a késleltetés stb. adatai. A HDR (nagy dinamikus tartomány) megléte is számíthat.

A **monitorvezérlő-interfész** (magyarul videókártya) feladata *a monitoron megjelenítendő kép előállítása*. Ez a képpontokból tevődik össze,

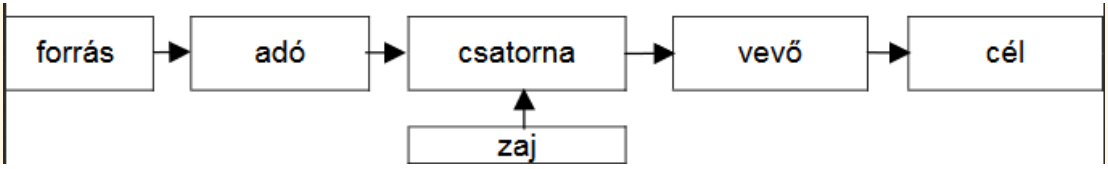
amelyeket ***pixeleknek*** neveznek. A megjelenítésre előkészített kép digitális változatát a kártya (grafikus) RAM-jában tárolják, amely így egy ***framebuffer*** (mondhatni „képpponthalmaz-tároló") szerepét tölti be. A videómemóriában minden pixelhez tároljuk a színmélységét is.

A videókártya lényeges paramétere a max. kimeneti felbontás, amely a maximális megjeleníthető pixelek számát adja meg (pl: 1920x1080 Full HD).

A ***videókártya összetevői*** maga a grafikus processzor (GPU), a videómemória, a ROM-BIOS (VBIOS), valamint – ha a kártyán van analóg (VGA) kimenet – akkor az ennek kezeléséhez szükséges digitális–analóg-konverter (DAC-)egység. Mára egyre jobban tűnik el a régi analóg tűs VGA csatlakozó, mert a DVI, HDMI és DP fizikai csatolófelületek ***digitális jelátvitelt*** biztosítanak, így nincs szükség jelkonverzióra.

16. Hálózati átviteli közegek. Vonalak megosztásának módszerei. Digitális jelek kódolása. A paritásbit és a CRC. Modemek feladata. ISDN, ATM, DSL technológiák

Ahhoz, hogy a csatornán információt lehessen átvinni, az adónak meg kell változtatnia a csatorna fizikai közegének valamilyen tulajdonságát, ami a közegen továbbterjed, és a vevő ezt a fizikaiközeg-változást érzékeli. Egy vezetékben változhat az átfolyó áram vagy a feszültség, ha antennával kisugárzott elektromágneses hullámot használunk, akkor a hullám amplitúdója, frekvenciája vagy kezdeti fázisszöge stb.



6. ábra – A kommunikáció folyamatának "szerkezete", adatátviteli modell

A modern hálózatok általában *három különböző közeget* használnak az eszközök összekapcsolására, illetve az adatok továbbítására. Ezek a közegek a következők:

- fémvezetékek a kábelekben,
- üveg- vagy műanyag-szálak (optikai kábelek),
- vezeték nélküli átvitel.

A jel továbbításához szükséges kódolás minden átviteli közegen különböző. A fémvezetékeken az adatokat speciális mintáknak megfelelő elektromos impulzusokkal kódolják. Optikai átvitelnél az infravörös vagy a látható fény tartományában használt fény-impulzusok adják a jeleket. A vezeték nélküli adatátvitelnél pedig az elektromágneses hullámok segítségével biztosítják a különböző biteket.

16.1. Vezetékes átviteli közegek

Csavart érpár (UTP, STP) – A csavart, vagy más néven sodrott érpár (Unshielded Twisted Pair = UTP) ***két szigetelt, egymásra spirálisan felcsavart rézvezeték***. Ha ezt a sodrott érpárat kívülről egy árnyékoló fémszövet burokkal is körbevesszük, akkor ***árnyékolt sodrott érpárról*** (Shielded Twisted Pair = STP) beszélünk. A csavarás a két ér egymásra hatását küszöböli ki, jelkisugárzás nem lép fel. A kategóriák közti különbség a ***felcsavarás sűrűsége***.

Koaxiális kábelek – A belső fém ***huzalt és az árnyékolást*** (ez külső vezetőként is szolgálhat) ***egy szigetelés*** (dielektrikum) *választja el*; az árnyékolást pedig külső köpeny borítja. Három igen lényeges jellemzője van: a ***hullámellenállása*** (Z0), a hosszegységre eső ***késleltetési ideje*** és ***csillapítása***. Régen volt *vékony* és *vastag* koax.

Optikai kábelek – Az üvegszálas kábelekben az információ ***fényimpulzusok*** formájában terjed egy *fényvezető közegben*, praktikusan egy üvegszálon. Az átvitel három elem segítségével valósul meg: ***fény-***

forrás (LED, dióda) – ***átviteli közeg*** – ***fényérzékelő*** (fotodióda, -tranzisztor). **Többmódusú** kábelnél *a teljes fényvisszaverődés* elvét használják a fényinformáció továküldésére. Az **egymódusú** kábel esetében az üvegszál átmérője megegyezik a fény hullámhosszával, így az nem verődik oda-vissza.

16.2. Vezetéknélküli hálózati átviteli közegek

Létezik az infravörös (lézeres), rádióhullámos és műholdas átvitel.

- Az **infravörös** (lézeres) átvitelnél a kommunikáció teljesen digitális. Jól irányítható, viszont kültéren a köd és eső zavarokat okozhat.
- A **rádióhullám** (mikrohullám) nagyobb távolságok áthidalására szolgál. A frekvenciasávok kiosztását a hatóság felügyeli, hogy egy frekvenciasávon csak egy, pl. rádióadó sugározzon.
- A **műholdas átvitel** esetén az űrbe, alacsony pályára állított műholdaknak felküldik a mikrohullámú jelet, majd azok felerősítve visszasugározzák. Óriási területen tudnak így szórni, amit a Földön antennával lehet fogni.
- A **Bluetooth** szórt spektrumú rádiófrekvenciás jelátvitel (ISM-sávú, azaz ipari-orvosi célra fenntartott sávban található) az alapja, rövid hatótávval, egyszerre 8 eszköz számára.
- **Mobilhálózatok** – A mobil-szélessáv a mobiltelefonos hálózaton szolgáltatott vezeték nélküli internethozzáférést jelenti számítógépek, mobiltelefonok és egyéb digitális készülékek számára. A mobilhálózatot használni képes eszközök rádióhullámokon kommunikálnak a hozzájuk közel eső adótoronnyal. A mobil-szélessávú hozzáférésre többféle szabvány létezik:
  - 1G (analóg)
  - 2G (GSM, CDMA, TDMA),

- 3G (UMTS, CDMA2000, EDGE, HSPA+)
- 4G (WiMAX, LTE).
- 5G (új technológiai standard. Az 5G cellás technológia távadói nagyon érzékeny érzékelők, amelyek egyaránt vesznek és továbbítanak is)

16.3. Vonalak megosztása

A hírközlési ***vonalak megosztására*** a legáltalánosabban a **multiplexelést** találták ki, azaz a (fizikai) vonalat felosztják (elemi) adatcsatornákra.

- Lehet ***frekvenciaosztásos*** multiplexelés, ilyenkor az adó oldalon a csatornák jeleit egy-egy vivőfrekvenciára ültetik.
- ***Időosztásos*** multiplexelés esetén a vonalat *időben* osztják fel több elemi adatcsatornára.
- Ezek ***kombinációja*** is működhet.

Másik lehetőség a vonalak maximális kihasználására, az átviendő információ ***kisebb adagok****ra bontása*, átvitele és összerakása (**üzenet**- és **csomagkapcsolás**i megoldások; üzenetkapcsolásnál egy csomagban átmegy az üzenet, csomagkapcsolásnál kisebb csomagokra darabolás történik).

Harmadik lehetőség a **vonalkapcsolás** (pl. telefonhívás), amikor a ***kommunikáció létrejöttekor keletkezik a kapcsolat is*** (nem az adó és a vevő közt van állandó összeköttetés), ami a kommunikáció befejezésekor megszűnik.

16.4. Digitális jelkódolás

A digitális jelkezelés esetében minden információ továbbítása **bitek formájában** történik, függetlenül az információ fajtájától.

2 érték fordulhat elő, pl. a 0-át és az 1-et reprezentáló feszültség- vagy áram-érték, és ez viszonylag nagy zaj mellet is jól megkülönböztethető megfelelő áramköri megoldásokkal. Másrész megfelelő kódolással az előforduló hibákat jelezni, sőt javítani is lehet.

A legáltalánosabb eljárás a **Manchester-kódolás**: 0 és 1 esetén is van jelváltozás, így teljes a szinkronizáció.

A **Nullára vissza nem térő** (Non-Return to Zero, NRZ-) kódolásnál mindig az a feszültségszint van a vonalon, amelyet az adott bit meghatároz. Ez nagyon egyszerűen megvalósítható kódolás. Sok váltást tartalmazó csomagoknál jó megoldás, azonban ha a sok egyforma bit van egymás után, akkor a vonal állapota is azonos szinten marad. Ez a szinkronizációt nagyon megnehezítheti.

A **hibakezelésnél** használatos fogalom a ***paritásbit*** és a *CRC*. A vevő-oldalon a pariásvizsgálat abból áll, hogy *a kódszóban lévő 1-esek számát* megnézzük, hogy ***páros vagy páratlan***. A ***CRC*** gyakorlatilag egy ***hibadetektáló*** *kód*. A küldő-oldalon a küldendő információhoz hozzáfűznek egy rövid, az üzenetből számított ellenőrző-értéket. Amikor a fogadó fél megkapja az üzenetet, akkor ő is elvégzi az ellenőrző-érték számítását a fogadott adatok alapján. Ha *nem ugyanaz* az ellenőrző-kód jön ki a kapott adatokból, akkor az azt jelenti, hogy az üzenet *nem jött át pontosan*, tehát valamilyen hiba volt az átvitelben.

16.5. Modem

A **modem** (a modulátor és demodulátor szavakból összetett szó) egy olyan berendezés, ami ***a digitális jelet analóg információvá***, illetve a másik oldalon ***az analóg jelet digitális információvá alakítja*** (amplitúdómoduláció, frekvenciamoduláció vagy fázismoduláció segítségével). Az eljárás célja, hogy a digitális adatot analóg módon átvihetővé tegye, azaz a régebben az egész világot behálózó telefonrendszert használtuk fel gépeink kapcsolatához úgy, hogy az analóggá át-

alakított jelet ráültettük egy szinuszos vivő-hullámra. Példaként említhető a rádiós és mikrohullámú modem, kábelmodem, telefonos modem.

Az **ISDN** telefonrendszer az összes ***analóg szolgáltatást digitálisra váltja fel***. Az ISDN-ben a jelek digitálisak az otthoni telefontól vagy számítógéptől kezdve az egész hálózaton át. Az ISDN-*nek nincs szüksége modemre*.

Az **ADSL** (aszimmetrikus digitális előfizetői vonal) egy kommunikációs technológia, ami a hagyományos modemeknél gyorsabb ***digitális adatátvitelt tesz lehetővé a csavart rézérpárú telefonkábelen***. Az ADSL jellemzője a DSL megoldásokon belül, 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, általában 8:1 arányban.

Az **ATM** egy ***aszinkron telekommunikációs átviteli mód***. Az adatot kis, fix csomagméretű cellákban továbbítja, emiatt gyors.

17. A számítógép-hálózatok architektúrája, az OSI-modell (rétegek, rétegszolgálatok). A TCP/IP-modell (feladata, rétegei, protokollok, információ-áramlás, címzés, útválasztás)

A rétegek és protokollok, kapcsolati szabályok halmazát nevezzük *hálózati architektúrá*nak. Annak érdekében, hogy csökkentsék a hálózatok bonyolultságát, a legtöbb hálózatot rétegekbe (*layers*) vagy szintekbe (*levels*) szervezik. Minden réteg vagy szint az alatta levőre épül.

Szempontok az adatáramlás irányai (szimplex, félduplex, duplex); funkcionális részek; hibajelzések, -védelem; adatátvitel-szabályozás (flow-control, folyamirányítás); korlátozások kivitelezése; útvonalkiválasztások stb.

17.1. Hálózati struktúra

(Ez nem biztos, hogy kérdezik, de azért nem árthat, ha itt van, önszorgalomból elolvasható:)

A ***hálózati struktúra*** a hálózat elemeinek funkcionális, azaz működésbeli elrendezését jelenti, azaz az egyes elemek működését és kapcsolatát, viszonyát írja le. A számítógépek a „***hosztok***”, és ezeket ***kommunikációs alhálózatok*** kötik össze. Az alhálózatok ***csatornákból***, azaz *vonalakból* és (aktív vagy passzív) ***kapcsolóelemekből*** állnak. (A ***topológia*** az *összekapcsolások strukturális leírása.*) 2 csoport van:

- ***Két pont közötti*** *alhálózatok* (***pont-pont összeköttetés***, ***busztopológia***) – a hálózat 2 pontját egy vezetékkel kötik össze; a hálózat többi tagja közvetetten kommunikál egymással csomagok segítségével. Amikor egy vevő *megkap egy csomagot*, és az *nem neki szól*, akkor *azt* ***továbbadja*** a szomszédnak egy következő pont–pont-összeköttetésen keresztül. Ilyenek a ***csillag***-, ***fa***-, ***háló***- és *teljes*-topológiájú hálózatok. A mai LAN-rendszerek többsége *csillag*-topológiájú.
- ***Közös csatornát használó (üzenetszórásos)*** *alhálózatok* – Ilyen típusú hálózatoknál ténylegesen ***egy kommunikációs csatorna*** van, és *ezen az egy csatornán osztozik* az összes, hálózatba kapcsolt számítógép. A küldött csomagokat a hálózat minden állomása veszi, és azt, hogy *a csomag kinek szól*, a csomagban elhelyezett egyedi – gépet címző – ***címinformáció*** hordozza. Így épülnek fel a ***busz***- és ***gyűrű***-topológiájú hálózatok.
- ***Hibrid topológiák*** – lényegében bármely elemi topológia előfordulhat egy teljes hálózat részeként, leggyakoribb mégis ***a busz és a csillag*** együttes alkalmazása. (A napjainkban használt hálózatkiépítési technika **fizikailag** ugyan **csillag**-topológia, mivel minden hálózati végpont össze van kötve egy külön

vezeték segítségével [a HUB-bal vagy] a switch-csel, ám **logikailag busz**-topológiát valósítanak meg, mivel üzenetszórásos technikával minden hálózati végpont megkapja az üzenetet.)

17.2. Hálózatszabványosítás – OSI-modell

Az **OSI-modell** a számítógépek kommunikációjához szükséges ***hálózati protokollok elméletiszabvány-rendszerét*** határozza meg. A különböző protokollok által nyújtott funkciókat ***egymásra épülő rétegekbe*** sorolja.

Hálózati kapcsolatnál az egyik gép ***k***.-adik rétege a másik gép ***ugyanilyen szintű rétegével*** kommunikál. Ezt olyan módon teszi, hogy minden egyes réteg az alatta elhelyezkedő rétegnek ***vezérlőinformációkat és adatokat*** ad át egészen a *legalsó rétegig*, ami már a kapcsolatot megvalósító *fizikai közeghez* kapcsolódik.

A kommunikációnál használt szabályok és megállapodások összességét ***protokollnak*** (*protocol*) nevezzük.

A szomszédos rétegek között egy ***réteginterfész*** húzódik, amely az alsóbb réteg által *a felsőnek nyújtott elemi műveleteket és szolgálatokat* határozza meg. A legfontosabb, hogy ez az interfész minden réteg között tiszta legyen olyan értelemben, hogy *az egyes rétegek egyértelműen definiált funkcióhalmazból álljanak*. Ez egyszerűvé teszi az adott réteg különböző megoldásainak a cseréjét, hiszen a megoldások az előbbiek alapján ***ugyanazt a szolgáltatást nyújtják*** a felettük levő rétegnek, segítve a nyílt (és egymással csereszabatos) rendszerek kialakítását.

Minden réteg csak és kizárólag *az alatta lévő rétegek* ***által nyújtott funkciókra*** támaszkodhat, és az általa megvalósított funkciókat pedig *csak a felette lévő réteg számára nyújthatja*. Szintén minden rétegnek rendelkeznie kell *a kapcsolat felépítését*, illetve annak *lebontását* biztosító eljárással; *adatátviteli szabályokkal* (egyirányú (szimplex),

váltakozóan két irányú (fél-duplex) vagy egyszerre két irányú (duplex)); *hibavédelmi* és *hibajelzési* tulajdonságokkal stb.

Ma a teljes OSI-modell egy *részhalmazát* használják csak.

17.3. OSI-modell rétegei – lentről felfelé haladva

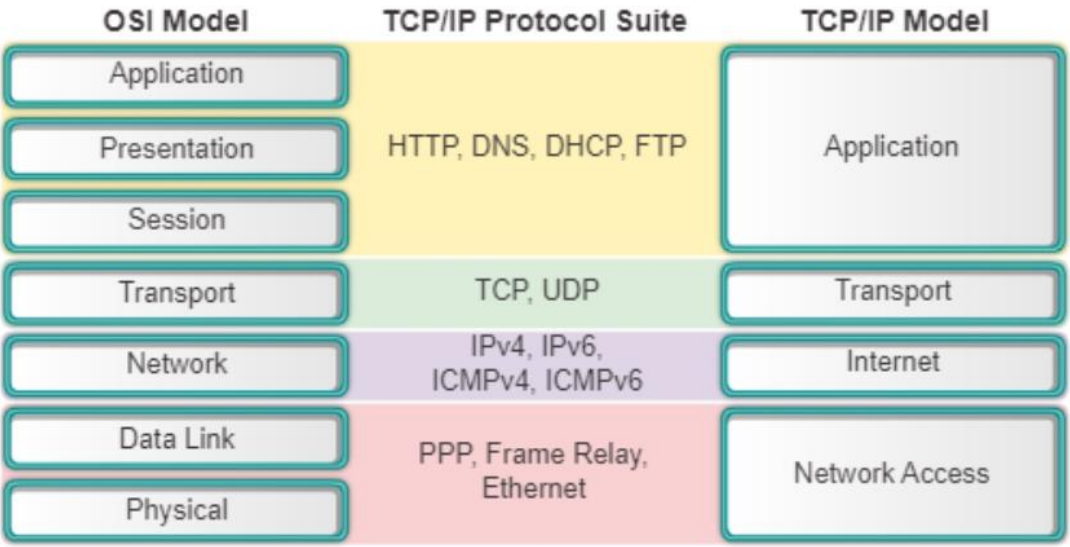
- **Fizikai réteg** felel *a fizikai átviteli közegen keresztül* a bitfolyam átviteléért. A fizikai réteg feladata a bitek kommunikációs csatornára való juttatása; a továbbítandó 0-khoz és 1-esekhez hozzárendeli a közegen továbbítható jeleket (feszültség, áram, fényimpulzusok stb.). Meghatározza az eszközökkel kapcsolatos fizikai és elektromos specifikációt, az érintkezők kiosztását, a használatos feszültség szinteket és a kábel-specifikációkat.
- **Adatkapcsolati réteg** felelős a *fizikai címzésért* (MAC). Csomópontból csomópontba való megbízható adatátviteli szolgáltatásért felel, védi a felsőbb rétegeket a fizikai rétegtől. ***Adatkeretekre*** tördeli az átvitt információmennyiséget.
- **Hálózati réteg** feladata az *útvonalválasztás* és IP (***logikai címzés***), hogy az adatkeretek milyen útvonal(ak)on haladjanak. Felelős két hálózati felhasználó közötti hálózati összeköttetési útvonal kiválasztásáért, létesítéséért, megszüntetéséért és az összeköttetésen keresztüli adatátvitelért. Statikus és dinamikus útvonalválasztási módszereket használhat.
- **Szállítási réteg** felel a végpontok közötti kapcsolatért és *a megbízhatóságért*. Feladata, hogy a viszonyrétegtől kapott adatfolyamot kisebb darabokra vágja szét, ha szükséges, akkor a túloldalon ezeket sorrendhelyesen visszaállítsa. Tulajdonképpen ez a réteg „burkolja el”, okkultálja azt a kommunikációs felek elől, hogy a köztük zajló kommunikáció útvonalainak milyen fizikai jellegzetességei vannak. Biztosítania kell a hibamentes adatátvitelt. ***Csomagokkal*** dolgozik.

- **Viszonyréteg** lehetővé teszi a *csomóponti kommunikációt* és azt, hogy a kommunikációs felek között ***milyen viszony van***, ki kezdeményez mi a kommunikáció jellege (adatfolyam, terminálablak) stb. Egy viszony például alkalmas arra, hogy állományokat továbbítson két gép között. A viszonyok egyidőben egy- és kétirányú adatáramlást is lehetővé tehetnek. A viszonyréteg egy másik szolgáltatása a *szinkronizáció*.
- **Megjelenítési réteg** az információk *megjelenítéséért,* a megfelelő ***adatábrázolásért***, -átalakításért felel, hogy egységes legyen a továbbítandó adatstruktúra. Ide tartozik például a kódátalakítás, fordítás, tömörítés stb.
- **Alkalmazási réteg** széles körben igényelt protokollokat tartalmaz. Például több száz inkompatibilis terminháltípus létezik ma a világon, ezért definiálni kell egy hálózati virtuális terminált, és a többi programot ezt felhasználva kell megírni. Az alkalmazásréteg tipikus feladatai az állománytovábbítás, elektronikus levelezés, távoli bejelentkezés stb. lefolytatásnak segítése.

A rétegek közötti kommunikáció úgynevezett ***rétegszolgálatok*** (funkcionális elemek, entitások) segítségével valósul meg. Ezek mindig két szomszédos réteg között találhatók. Lényegében a két réteg közötti kommunikáció ténylegesen ezeken a pontokon keresztül valósul meg.

17.4. A TCP/IP-protokollkészlet

Az Internet kisebb kiterjedésű számítógépes hálózatok (LAN-ok) összekapcsolásából álló globális számítógépes rendszer; ezen összekapcsolt hálózatstruktúra elemei a **TCP/IP protokoll** (protocol stack, protokollkészlet) egyik rétege, az ***IP*** segítségével kommunikálnak. A



7. ábra – A TCP/IP- és az OSI-modell összehasonlítása

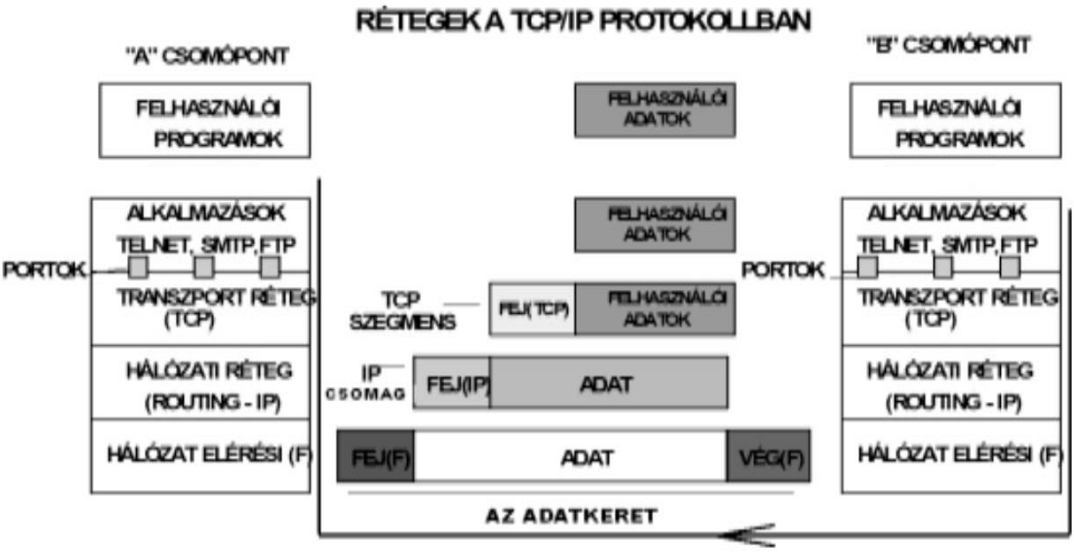
TCP/IP *nem követi* az OSI hétrétegű felépítését, mivel – a '60-as években épült ki – megelőzte azt („*de facto*-szabvány", az ARPANET-projektben fejlesztették ki '69-ben, az USÁban).

17.5. A TCP/IP-modell négy rétege

- **Hálózatelérési** (***Network Interface***) réteg: Az OSI-modell két alsó szintjének (fizikai adatkapcsolati) felel meg, és ez biztosítja a kapcsolatot a csomópontok között. Megvalósítása a vezetékes Ethernet és a WIFI. Egyéb Protokollok: *ARP* (címfeloldás), Token -ring, Token Bus, MAC stb.
- **Hálózatok közötti** (***Internet***-) réteg: Az OSI-modell hálózati rétegének felel meg, ez a réteg végzi a csomagok útvonalkijelölését és továbbítását a hálózatok között. Ennek a rétegnek a protokollja az *Internet Protocol* (**IP**), az üzenetvezérlő protokoll, a csomagok továbbítását *a hálózati címek* alapján végzi. A rétegben előforduló események és hibák jelzésére szolgál az *Internet*

Message Protocol (**ICMP**), az Internet-réteg vezérlőüzenet-protokollja.

- Hoszt-hoszt réteg (**Transport**) – Az OSI modell szállítási és hálózati rétegének felel meg. A létesített és fennálló kapcsolat fenntartását és működését biztosítja. Két rétegprotokollból áll: az egyik a *Transmission Control Protocol* (**TCP**) azaz a továbbítást szabályozó eljárás, a másik az *összekötetésmentes szállítási protokoll*, a User Datagram Protocol (**UDP**).
- ***Alkalmazási szint*** (**Application**) – Itt vannak a felhasználói és a hálózati kapcsolatot biztosító programok; a felhasználó által indított program és a szállítási réteg között teremt kapcsolatot. Jellemző kialakítás a kliens-szerver-kapcsolat. A kliens egy alkalmazás, ami valamilyen információt igényel (weboldal, IP cím, e-mail) a szervertől. A szerver is egy alkalmazás, amely folyamatosan figyeli a kliensektől érkező kéréseket, és végrehajtja. Alkalmazási protokollok: FTP, HTTP, SMTP, DHCP, SSH stb…



8. ábra – A TCPI/IP-protokoll felépítése

17.6. Címzés

Az Ethernet logikai topológiája egy ***többes hozzáférésű sín***. Minden hálózati eszköz ugyanahhoz ***megosztott közeghez*** csatlakozik és minden csomópont megkapja a közegen továbbított összes keretet.

Az összes keret feldolgozásából adódó túlzott mértékű többletterhelés megakadályozása érdekében (a *Media Access Protocol*-ból adódóan) egy **MAC-címnek** nevezett egyedi azonosítót hoztak létre, hogy a tényleges forrás-és célcsomópontokat azonosítani lehessen egy Ethernet hálózaton belül. Az OSI modell alsóbb szintjén tehát a MAC-címzés azonosítja az eszközt. Ez egy 48 bites bináris érték, amit 12 hexadecimális számjeggyel írunk le (egy hexadecimális számjegy 4 bitet jelöl), és a ***hosztokat ezzel azonosítjuk*** egyedileg.

A TCP/IP ***címrendszerében*** az IP-címek szigorú szabályok alapján vannak kialakítva. A hálózat minden gépéhez hozzá kell rendelni (legalább) egy egyedi számot a hálózati környezetben való tájékozódási lehetőség biztosításához. Az IP(v4)-címeket négy darab, ponttal elválasztott számmal írjuk le, pl. 192.168.0.10. A **127.0.0.1 = localhost**, egy spéci cím (loopback), ami az adott gép saját IP címére mutat. Ez hosztnak lehet több „neve” (***alias*** – azaz több hivatkozási alak) is az egy címhez.

Az egyes elemek 0 és 255 közötti értéket vehetnek fel, tehát 4db 8 bites számról van szó. A hálózaton ***alhálózatokat*** hozhatunk létre *címmaszkok* (alhálózati maszkok) használatával, pl. 255.255.255.0. Egy IP-címben úgy lehet megállapítani, hogy mi az alhálózat és mi a hoszt (cél-gép) címe, hogy ÉS-kapcsolatba hozzuk az IP-címet az alhálózati maszkkal.

Példa alhálózati maszk használatára:

- IP-cím:196.225.15.4
- Alhálózati maszk: 225.225.255.0

Kettes számrendszerben:

- IP-cím:11000100 11100001 00001111 00000100
- Maszk: 11111111 11111111 11111111 00000000

A két szám között az ÉS (AND) műveletet bitenként elvégezve a hálózatcímét kapjuk:11000100 11100001 00001111 00000000 (tízes számrendszerben: 196.225.15.0).

(Egyébként kiadható IPv4-címek száma folyamatosan csökken, folyamatos az áttérés az IPv6-címek használatára.)

Az **ARP-protokoll** (címmeghatározási protokoll) két alapvető funkciót biztosít:

- IPv4-címek összerendelése MAC-címekkel.
- Az összerendelési táblázat kialakítása

17.7. Információáramlás – beágyazásokkal (Encapsulation); útválasztás

A hálózaton minden kommunikáció ***egy forrás és egy cél között jön létre***. A hálózaton át küldött információt adatnak vagy adatcsomagnak nevezzük. Ha egy gép adatot akar küldeni egy másik számítógépnek, akkor a küldendő adatokat ***be kell csomagolnia***. Ezt a folyamatot **beágyazásnak** nevezzük. A referenciamodellek közül bármelyikre elmondható, hogy a hálózat rétegei a következő öt lépéssel ***ágyazzák be és továbbítják*** az adatokat (a vevőnél levő rétegek pedig az alábbi lépéseket ***visszafelé*** végzi el):

1. **Adattá** alakítják a képeket és a szövegeket.
2. **Szegmensekbe** csomagolják az adatokat.
3. A forrás- és a célállomás címét tartalmazó **csomagba** ágyazzák az *adatszegmenst*.

4. A következő közvetlenül csatlakozó készülék **MAC-címét tartalmazó keret**be ágyazzák a csomagot.

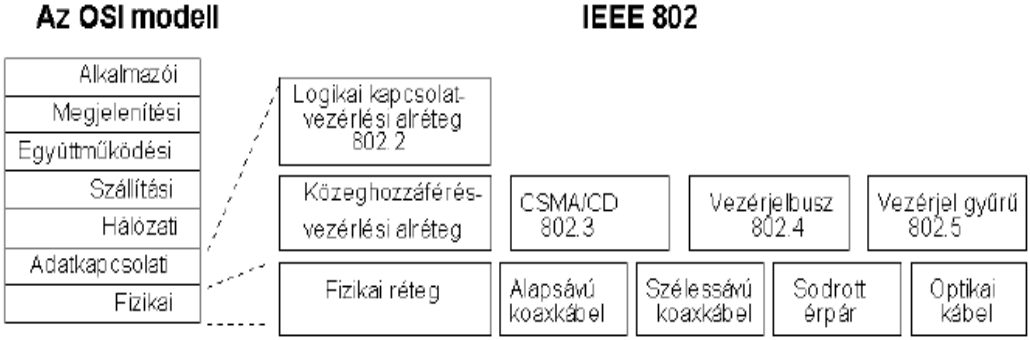
5. Átalakítják a keretet az átviteli közegen továbbítható **egyesek és nullák (bitek)** sorozatává.

A csomagkapcsolt rendszerekben az **útválasztás** (*routing*) azt a folyamatot jelöli, amivel kiválasztjuk az útvonalat, amin a csomagot továbbküldjük. Egy csomag egy rakás hálózati eszközön (routerek, switchek, bridgek stb…) is áthaladhat, amíg elér a fogadó félhez. Az útvonalat az útválasztási tábla (*routing table*) alapján határozza meg a router.

18. Lokális hálózatok szabványos megvalósítása (Ethernet, vezérjeles sín, vezérjeles gyűrű): protokollok, közeg-hozzáférési módszerek, átviteli közegek, fizikai egységek

18.1. Szabványok

Az *Institute of Electrical and Electronics Engineers* (villamosmérnökök nemzetközi szervezete, **IEEE**) egy szakmai szervezet, amely az elekt-



9. ábra – Az OSI-modell és a TCP/IP összehasonlítása

ronikai, távközlési szabványokat felügyeli. A helyi hálózat adatkapcsolati–hálózati közegeinek leírása a **802**-es szabványcsaládba tartozik (azt definiálja, hogyan férhet hozzá a hálózati közeghez egy időben több számítógép anélkül, hogy egymást zavarnák a kommunikációban). Ennek keretében három szabványt fogadtak el, amelyekre együttesen az IEEE 802-es szabvány részeként hivatkoznak.

Az IEEE 802 szerinti szolgáltatások és protokollok a hétrétegű *OSI hálózati referenciamodell* **alsó két rétegébe** (adatkapcsolati- és fizikai réteg) tartoznak. Tény, hogy az IEEE 802 az OSI adatkapcsolati rétegét két al-rétegre osztotta, amelyeket ***logikaikapcsolat-vezérlésnek*** (LLC) és ***közeghozzáférés-vezérlésnek*** (MAC) nevezett el, így a rétegek a következők lettek:

- Adatkapcsolati réteg (Data link layer)
  - LLC Sublayer (Logical Link Control)
  - MAC Sublayer (Media Access Control)
- Fizikai réteg (Physical layer)

A szabványokat további részekre osztották (ez a zöldített rész inkább egy kis önszorgalmi plusz…):

- A 802.1-es szabvány a szabványhalmaz alapjait írja le, és az interfészprimitíveket definiálja.
- A 802.2-es az adatkapcsolati réteg felső részét, az ún. LLC (Logical Link Control - logikai kapcsolatvezérlés) alréteget definiálja. Sokáig vita volt arról, hogy az eltérő módszerek miatt hová tartozzon a közeghozzáférés: a fizikai réteghez vagy az adatkapcsolati réteghez. A vita lezárásaként az adatkapcsolati réteget osztották két részre: a közeg-hozzáférési alrétegre (MAC - Media Access Control - közegelérés vezérlés) és az LLC-re.
- A 802.3-as szabvány a CSMA/CD (Ethernet) leírása. Nagyon fontos itt megjegyezni, hogy a 802.3 szabvány és az Ethernet

nem azonos fogalmak. Az Ethernet egy termék, azaz a 802.3-as szabvány megvalósítása.

- A 802.4-es szabvány a vezérjeles sín, és a
- A 802.5-as szabvány a vezérjeles gyűrű leírása.
- A 802.11 a WIFI-t tartalmazza. A **Wifi6** a 802.11ax.

18.2. Röviden a lokális hálózat funkcionális szervezéséről

A számítógép-hálózat kialakításának célja, hogy a kapcsolatban álló gépek használhassák egymás erőforrásait (perifériáit); továbbá hogy egyszerű adatközlés küldő- és/vagy fogadó-oldalán szerepelhessenek. E feladatok, szerepek és a munkamegosztás tekintetében alapvetően két modellt különböztetünk meg:

- **Egyenrangú** (peer to peer) hálózat (nincs kitüntetett, központi gép)
- **Kliens-szerver**- (ügyfél-kiszolgáló-) hálózat (van egy kitüntetett, központi gép, a szerver, melye(ke)n lehet(nek) fájl-, adatbázis-, nyomtató-kiszolgálási stb. szerepkörök, illetve egyéb szolgáltatások)

18.3. Ethernet

Az **Ethernet** elnevezés a számítógépes hálózatok egy specifikációját és annak kiterjesztéseit takarja, amelyek a LAN (helyi hálózat), MAN (városi hálózat) és WAN (nagy kiterjedésű hálózat) kialakításának műszaki tartalmát írja le. Manapság az elnevezés már összeforrt a számítógépes helyi hálózattal, így legtöbbször már ennek a szinonimájaként használatos. Az eredeti változat a DEC, Intel és Xerox cégek által kidolgozott alapsávú LAN-ra vonatkozó specifikáció volt. Az IEEE 802.3 szabvány a napjainkban is használatos megoldások alapjának tekint-

hető, amit azóta az igényeknek megfelelően időről-időre újabb kiegészítésekkel bővítenek. A régebbi Ethernet-szabványú hálózatok az ütközések feloldására a CSMA/CD-t használták.

Az Ethernet fejlődése:

- Klasszikus Ethernet (kábelszabányok: *10Base*-5; 2, T, F)
  - Vastag Ethernet
  - Vékony Ethernet
- **Kapcsolt Ethernet**
  - Gyors Ethernet (802.3u; 100Base-T4, TX, FX)
  - **Gigabites Ethernet** (802.3.*z*; 1000Base-T, SX, LX, CX)

A gigabites Ethernet – eltérően a klasszikus Ethernet-től – **pont-pont felépítésű**. A duplex és félduplex működést is támogatja. "Normális" esetnek a duplex módot tekintik, a forgalom ***mindkét irányban egy időben*** folyhat. Ezt akkor használják, ha egy ***központi kapcsolót*** (*switch*-et) vagy a periférián lévő *gépekkel*, vagy *más kapcsolókkal* kötnek össze. Ekkor minden adatot pufferelnek, így bármelyik gép és kapcsoló ***tetszés szerinti időben küldheti el az adatait*** (kereteit). Az adónak nem kell figyelnie a csatorna forgalmát, mert a versengés kizárt (így ekkor a CSMA/CD vagy bármely ütközésfigyelés használata felesleges). Ha a számítógépek nem kapcsolóhoz, hanem *elosztóhoz* (hubhoz) kapcsolódnak, akkor a *félduplex* módot használják.

18.4. A közeghozzáférés fajtái

Régebben az Ethernet-hálózatokban inkább az **üzenetszórás**t megvalósító eljárást alkalmazzák, minden hoszt ***a közösen használt csatorna*** segítségével próbál kommunikálni a másik hoszttal. Az rengeteg hoszt kommunikációjának összekeveredését és érthetetlenné válását megakadályozzuk, a hozzáférést biztosítani kell.

(Még egy kis önszorgalmi plusz - az alábbi hozzáférési módok kerültek használatba az idők során, ezek közül ma már nem mindegyik használatos:)

- ***Véletlen vezérlésű közeghozzáférés***
  - ALOHA
  - Ütközést figyelő, ütközést jelző (CSMA/CD)
  - Réselt gyűrű
  - Regiszterbeszúrásos gyűrű
- ***Osztott vezérlésű közeghozzáférés***
  - Vezérjeles gyűrű
  - Vezérjeles sín
  - Ütközést figyelő, ütközést elkerülő (CSMA/CA)
- ***Központosított vezérlésű közeghozzáférés***
  - Lekérdezéses eljárás
  - Vonalkapcsolásos eljárás

Aloha – Az eljárás lényege, hogy az adók korlátozás nélkül adhatják üzeneteiket. Bármelyik hoszt bármikor, bármekkora adást kezdeményezhet. Ha az adás pillanatában már van üzenet a csatornán, akkor természetesen ütközés keletkezik, és mindkét üzenet elvész. Ekkor az adók újrakezdik a „műsorszórást".

Réselt ALOHA – a felhasználóknak meg kell egyezniük abban, hogy mekkora időközönként lehet adást kezdeményezni. A meghatározott időközöket szokták ***időrésnek*** nevezni. (Pl. fél másodperces időrés azt jelenti, hogy félmásodpercenként kezdeményezhetnek adást, adás így fél másodperc hosszú is lehet.)

Réselt gyűrű – Gyűrű-topológia esetén alkalmazható protokoll. A réselt ALOHÁ-hoz hasonlóan itt is egységnyi részekre (keretekre) kell feldarabolni az átküldendő adatokat. Csak akkor adhat az állomás, ha a ***szinkronjel*** a rés kezdetét jelzi, és a keret foglaltságát jelző ***marker***

(jelzőbit) szabadnak jelzi a csatornát. Az adás megkezdésekor a marker foglalt állásba kerül.

Regiszterbeszúrásos gyűrű – gyűrű-topológiájú hálózatok léptető- és tárolóregiszterei vannak felhasználva a csatorna „figyelésére”, illetve adatközlésre; ha ***regiszterek*** üresek, szabad a csatorna.

18.5. Vezérjeles gyűrű és sín

Gyűrű-topológia esetén alkalmazható protokoll a ***vezérjeles gyűrű***. Lényege, hogy a legelső adást kezdeményező hoszt előállít egy ***tokent*** (***vezérjelet***), melyet adás után továbbad a szomszéd hosztnak. Az vagy adást kezdeményez, vagy adási szándék hiányában egyszerűen továbbadja a tokent. Az elv megengedi a ***prioritás*** létrehozását, ebben az esetben a prioritással rendelkező állomásnak *nem kell rögtön* tovább adnia a tokent, hanem csak meghatározott számú adás után. Gond lehet, ha a token elvész, vagy leblokkol egy állomásnál. Ennek az elkerülésére ***felügyelő-állomást*** neveznek ki, amelynek joga van a token *újrakiosztására*.

Vezérjeles sín – Ugyanaz az elv, mint a gyűrűnél, azzal a különbséggel, hogy a fizikai síntopológiát **logikai gyűrűvé** alakítják.

18.6. (CSMA/CD –) Ütközést figyelő, ütközést jelző protokoll

Carrier Sense Multiple Access with Collision Detection, azaz ***vivőjel-érzékeléses többszörös hozzáférés ütközésérzékeléssel***. Az ***Ethernet-hálózatok ezt használják*** (IEEE 802.3). Itt van feltétele az adás megkezdésének. Csak akkor kezdeményezhet adást a hoszt, ***ha a csatorna üres***, így nem ütközhet bele a már csatornán lévő adatba (előtte „belehallgat” a csatornába). Ha 2 adó egyszerre kezd „sugározni”, de észlelik, hogy már van adat, akkor leállnak, és elölről kezdik a folyamatot; a perzisztencia (kitartás; 1, 0 és p értékű lehet) határozza meg a várakozás hosszát.

A hosztoknak ***3 állapota*** lehet: ***versengéses*** (versenyez, hogy minél előbb adást kezdhessen), ***átviteli*** (éppen adatátvitelt folytat) és ***tétlen*** (nincs adatközlés).

18.7. (CSMA/CA –) Ütközést figyelő, ütközést elkerülő protokoll

Carrier Sense Multiple Access with Collision Avoidance, azaz ***vivőjel-érzékeléses többszörös hozzáférés ütközéselkerüléssel***. A CSMA/DC protokoll nagy számú hoszt esetében már nem működik kielégítően. Túlságosan ***sok hoszt kerül versengési állapotba***, és így *kénytelen lesz várakozni*. A probléma egyik megoldása, hogy szétválasztjuk a teljes hálózatot *alhálózatokra*. Ekkor viszont az alhálózatok *összeköttetése* lesz a szűk keresztmetszet. Sokkal jobb megoldás az **ütközés lehetőségének elkerülése**. Az ütközés elkerülését úgy oldják meg a CSMA/CA-típusú protokollok, hogy az adás befejezése után a hosztok egy **logikai listában** elfoglalt *helyüktől függő ideig várakoznak*, és csak utána kezdik meg az adást. (A *WIFI-hálózatok ezt használják*.)

18.8. Központosított vezérlés

Egy kitüntetett állomás vezérli a hálózatot, az állomásoknak jogokat ad és vesz el, szabályozza az állomások csatornahasználatát.

- **Lekérdezéses eljárás** – Az állomások között létezik egy kitüntetett, ***master*** (fő) ***állomás***, ami szabályozza a mellékállomások működését. A főállomás *egymás után szólítja fel a mellékállomásokat* adásra. Csak az az állomás adhat, amelyiket a főállomás felkért adásra. Ha nincs szüksége a csatornára az állomásnak, akkor ezt tudatja a főállomással, amely a következőt kérheti fel adásra.

- **Vonalkapcsolásos eljárás** – Itt is létezik főállomás és mellékállomások. Ám ebben az esetben a főállomás egy kapcsolóközpontként köti össze a kommunikálni kívánó mellékállomásokat.

18.9. Átviteli közegek (LAN)

- **Sodrott érpár** – két rézhuzalból áll, amely spirálisan feltekerve kiküszöböli a zaj jelentős részét. Erősítés nélkül is több km-ig használható. Ha több érpár fut együtt, akkor azokat kötegelik. Analóg és digitális átvitelre is alkalmas. Pl: STP-, UTP-kábelek (Cat 1-7).
- **Koaxiális kábel**: Leginkább digitális jelátvitelre alkalmas. A kábel felépítése úgy néz ki, hogy közepén fut a központi vezeték, ekörül vastag műanyag szigetelés van és ezt még réz-fonattal borítjuk körbe az árnyékolás miatt. Remek sávszélesség és kitűnő zajvédelem jellemzi.
- **Optikai szál**: az adatok átvitele fényimpulzussal történik (van fény - 1, nincs fény - 0). Három része: fényforrás - lézerdióda -, átviteli közeg - üvegszál -, fényérzékelő - fotodióda -. Nagy távolságokat képes áthidalni (pl. tengerek alatti kábelek), gyakorlatilag zaj immunis mert nem elektromágneses impulzusokat küldünk.
- **Vezeték nélküli hálózat** - WLAN, WIFI (Kábel nélküli, rádiófrekvenciás LAN); az ún. szórt spektrumú, kábelnélküli (wireless-) rendszerek a felhasználó szintjén adnak megoldást számítógépek, hálózatok összekapcsolására, közepes távolságig (10-20 m). A IEE 802.11-es szabványcsalád (kiegészítései) részletezi(k) ennek fajtáit és specifikációit.

18.10. Fizikai egységek a hálózatban

Az **adapterkártya** tartalmazza a logikai kapcsolatvezérlést, és a közeghozzáférést vezérlő funkciókat megvalósító hardvert és förmvert (firmware).

Kábelrendszer (média) azt a kábelt, ill. vezetéket és csatlakozószerelvényeket jelenti, amelyet a hálózatban lévő eszközök összekapcsolására használnak.

(Régi, manapság nem használatos eszközök az önálló *transceiver* [jeladó-vevő], illetve a *repeater*, jelismétlő.)

A leginkább használt (hálózati) aktív eszközök:

Elosztó (HUB) – A HUB (angol szó, jelentése: középpont) a csillag-topológia középpontjának feladatát ellátó eszköz. Az intelligensek csomagkapcsolást vagy forgalomirányítást is végeznek. Az elosztók az OSI-modell **fizikai rétegéhez** tartoznak. Ezek is kiszorultak manapság…

Híd (*Bridge*) – A hálózati híd ***a hálózati szegmensek között biztosít adatforgalmat*** intelligens módon. Az érkező adatcsomagból kiolvassa a feladó és a címzett hardvercímét, és tárolt áthidalási táblázata segítségével eldönti, hogy továbbítsa-e a csomagot, vagy ne. Ha a küldő és a címzett ugyanabban a szegmensben van, akkor azt nem jeleníti meg a többi kimenetén (a többi szegmensben), ha különbözőben, akkor csak a címzett szegmensébe továbbítja. Csak a MAC-címeket figyel, mivel az OSI **adatkapcsolati rétegében** tevékenykedik. (A magasabb szintű címek (pl. IP) átirányítására nem képes.)

Az **útválasztók** (*router*) a hidakhoz hasonló filozófiával működnek, csak éppen az OSI-modell **hálózati rétegében**. A csomagok ***hálózati címét vizsgálják*** és elvetik a csomagot, ha ugyanabba a szegmensbe van címezve, egyébként pedig csak arra a szegmensre küldik, ame-

lyikben a címzett található. Különböző architektúrák, illetve eltérő hálózatiközeg-hozzáférési metódust használó szegmensek között is meg tudják teremteni a kapcsolatot.

Kapcsoló (*Switch*) – A hálózati kapcsolók **az adatkapcsolati rétegben** tevékenykednek, ezért ***speciális hidak halmazának*** tekinthetjük őket. Az egyik oldaluk egy *közös belső szegmenshez*, buszhoz; másik oldaluk pedig valamilyen *hálózati csomóponthoz* vagy egy további szegmenshez kapcsolódik. Működésük egyik módja az, amikor már ismertté vált a csomag címe: ilyenkor *a csomag egyből továbbításra kerül.* A másik esetben *a teljes csomag begyűjtése után* történik csak a *cím dekódolása*, amely alapján a továbbítás megtörténhet. Ez utóbbi esetben ***speciális szűrőfeltételek*** is megfogalmazhatók, továbbá ***virtuális hálózatok*** kialakítására is lehetőség nyílik. A kapcsolókat elláthatják *útválasztó-képességekkel* is, így alkalmassá válnak – egy berendezés alkalmazásával – hálózati szintű címek alapján megvalósuló kapcsolásra is.

Átjáró (*Gateway*) – Az átjárók elsődlegesen az **alkalmazási rétegben** működnek, de kaphatnak feladatot például a viszonyrétegben is. Protokollok, illetve ***adatformátumok közötti konverziót hivatottak ellátni***. Protokollok közötti fordítás történik akkor, ha a küldő például IPX/SPX-et használ, a fogadó pedig TCP/IP-t: ekkor az átjáró a csomag adatait átalakítja a célprotokoll keretformátumára. A híd is hasonlót csinál, de az nem alakítja át a keretformátumot, hanem becsomagolja az egyiket a másikba. Az átjárók különböző technológiát használó szegmensek közötti kapcsolat megvalósítását is szolgálják.

Szünetmentes tápegység (*UPS*) – Ez a berendezés nem a hálózati közeghez illeszkedik közvetlenül (ellentétben a fenti berendezésekkel), és nem is kimondottan csak hálózat esetén lehet szerepe, hanem kliens-szerver hálózat esetén fokozott jelentőséggel bír. A kiszolgáló működési zavara megbéníthatja a teljes hálózati munkát, hirtelen leállása adatvesztést eredményezhet. Ilyen zavart okozhat az elektromos

hálózat feszültségkimaradása, -ingadozása. A szervergépeket – központi szerepük miatt – indokolt ellátni szünetmentes áramforrásokkal. Az UPS-ek jellemző adata, hogy mennyi ideig képesek tovább működtetni a rájuk kapcsolt számítógépet (és egyéb eszközöket).

19. Az operációs rendszer erőforrás-kezelőjének feladata. A holtpont és kezelésének stratégiái. Biztonságos állapot. A szemafor használata a termelő-fogyasztó folyamatok esetében

Az **erőforrás-kezelő** a rendszermag (***kernel***) azon része, amely az erőforrások elosztásáért és lefoglalásáért felelős. Ha egy folyamat erőforrást igényel, az erőforrás kezelő dönti el, hogy a kérés kielégíthető-e.

Az erőforráskezelő gondoskodik a számítógép *erőforrásainak* hatékony, gazdaságos ***elosztásáról***, illetve az erőforrások használatáért vívott ***versenyhelyzetek*** *kezeléséről*.

- **Hardver-erőforrások**: pl. processzor, memória, nyomtató és az egyéb perifériák.
- **Szoftver-erőforrások**: a különböző közösen használható programok, adatállományok, adatbázisok.

Ha bármely folyamat, amely erőforrást igényel, feltétel nélkül megkapja azt, hamar **holtpont** alakul ki. Holtpontnak nevezzük, amikor *több folyamat ugyanannak az erőforrásnak a felszabadulására vár*, amit csak egy ugyancsak várakozó folyamat tudna előidézni. Ennek elkerüléséhez ***megelőzésre*** vagy folyamatos figyelésre, majd ***felszámolásra*** van szükség.

Ha megtiltjuk, hogy egyszerre több folyamat is rendelkezzen erőforrással, elszaporodhatnak a várokozó folyamatok és **kiéheztetés** lesz a végeredmény. A kiéheztetés az erőforráskezelő-stratégia miatt jöhet

létre. Ilyenkor összesen *ugyan van elegendő erőforrás*, de szerencsétlen esetben egyes folyamatok mégis "*éheznek*". Az erőforrás-kezelő ***dönt, hogy melyik folyamat jut erőforráshoz***, és lehet, hogy egy folyamat elé mindig bekerül egy másik, így az a folyamat beláthatatlan ideig nem jut erőforráshoz.

Egyetlen foglalási lehetőség stratégiája: Csak az a folyamat foglalhat erőforrást, amelyik még egyetleneggyel sem rendelkezik.

Rangsor szerinti foglalás stratégiája: Az erőforrásokhoz sorszámot rendelünk, és a leggyakoribbak kapják a legkisebbet. Igény esetén egy folyamat csak ***a birtokoltnál magasabb sorszámú erőforrásokat*** igényelhet.

Biztonságos állapot: Egy rendszer állapota akkor biztonságos, ha létezik egy olyan sorrend, amely szerint ***a folyamatok erőforrásigényei kielégíthetőek***.

A **szemafor** használata a termelő-fogyasztó folyamatok esetében azt jelenti, hogy a termelő és a fogyasztó közös memóriaterülethez fér hozzá, de azt ***nem használhatják egyszerre***. Hogy kizárjuk az egyidejű erőforrás-használatot, használhatunk szemafort, mivel a szemafor megmutatja, *ha egy másik folyamat* ***éppen használja*** a kívánt erőforrást.

20. A magas, közbenső és alacsony szintű ütemezők feladata egy operációs rendszerben. A folyamatok állapotai. Ütemezési algoritmusok

Az idővel való gazdálkodást ütemezésnek (***scheduling***) nevezzük. Az ütemezés során a folyamatok állapota megváltozik. Attól függően, hogy milyen állapotok között történik váltás, az ütemezők több szintjét definiálhatjuk.

A magas szintű ütemező választja ki a háttértár programok közül azt, amelyik az operációs rendszer felügyelete alá kerülhet.

Közbenső ütemező folyamatosan figyeli a rendszer állapotát (terhelését) és ha *túlságosan sok folyamat* kerül futásra kész állapotba, és egyiknek sem jut elég *processzoridő*, akkor egyes folyamatokat ***felfüggeszt***, illetve ***prioritásukat átrendezi*** a rendszer hatékony működésének érdekében.

Az **alacsony ütemező** feladata, hogy a processzor (és egyéb hardverek) erőforrásait a futásra kész folyamatok között *igazságosan és hatékonyan ossza el.* Legfőbb követelmény vele szemben a gyorsaság.

A folyamatok ***állapota*** lehet:

- **Futásra kész**: CPU-n kívül az erőforrások rendelkezésre állnak.
- **Fut**: Végrehajtás alatt.
- **Várakozik**: Foglalt az erőforrás, amire szüksége van.
- (**Megszakad**): Végrehajtása megszakítva egy interrupt (megszakításkérés) miatt.

20.1. Ütemezési algoritmusok

FCFS (First Come First Served): A folyamatok ***érkezési sorrendben*** kapják meg a processzoridőt lefutásukig.

SJF (Shortest Job First): A ***legrövidebb processzoridőt igénylő folyamatot*** részesíti előnyben. A legrövidebb várakozási időt adja, viszont a *hosszabb futást* igénylő folyamatokkal „mostohán" bánik.

RR (Round Robin): Minden egyes folyamatnak egy meghatározott processzoridőt biztosít, azután megszakítja, és ***a várakozási sor végére teszi***. Előnye, hogy a legrövidebb válaszidőt produkálja és a folyamatok között „*demokratikusan*" osztja el a CPU-t, viszont jelentős *adminisztrációt* igényel.

21. Többfeladatos (multitasking) operációs rendszerek feladatai, felépítése. A tárvédelem feladata és megvalósítása (privilégiumi szintek, jogosultságok, szegmensek, deszkriptorok, kapuk)

Többfeladatos egy OS, ha egy időben több folyamat végrehajtását végzi el. Ilyen folyamatok pl.: processzor-idő, a különböző erőforrások kezelése, a képernyő el-/felosztása stb. Továbbiak:

- **Eszközkezelés**: perifériák különbözőségeinek elfedése a programok elől.
- **Megszakításkezelés**
- **Rendszerhívás**: az OS magjának úgy kell kiszolgálnia a felhasználói alkalmazások igényét, hogy azok ne vegyék észre, hogy ***nem közvetlenül*** használják a perifériákat.
- **Erőforrás-kezelés**: közös, egymással ütköző erőforrás-használat megelőzése vagy a bekövetkezéskor annak feloldása.
- **CPU-ütemezés**: CPU idejének elosztása valamilyen *stratégia* alapján és munkák közti *átkapcsolási folyamatok* vezérlése.
- **Memóriakezelés**: felosztja a RAM-ot úgy, hogy a folyamatok se egymást, se az OS-t ne zavarják.
- **Állomány- és háttértárkezelés**: Rendet tart az állományok között.

21.1. Felhasználói felület

A ***tárvédelem*** feladata a programok és adatok védelme. Ez azért különösen fontos, mert az OS és a felhasználói programok fizikailag ***ugyanabban a memóriában*** vannak tárolva. Meg kell megvalósítania az operációs rendszer védelmét a felhasználói programoktól. A felhasználói folyamatokat védeni kell egymástól – de biztosítani kell kommunikációjukat. Biztosítani kell az adatokhoz, programokhoz való hozzáférési jogok ellenőrzését.

Privilégiumok és jogosultságok figyelembevétele esetén egy program csak a vele azonos vagy magasabb szintű programot hívhatja meg, és nála alacsonyabb szinten lévő adatot nem érhet el pl. *iOS* esetén a ***sandboxing***: minden app a saját „homokozójában", a saját adataiban „matathat" csak. A rendszerhez vagy más appokhoz nem fér hozzá.

Két privilegizált szintet különböztetünk meg:

- **Magas** jogosultsági szint az OS ***rendszer-programjai*** számára
- **Alacsony** szintet a ***felhasználói*** programoknak

Az ***azonos*** privilegizálási szinten futó taszkok *memóriaterületét védeni kell egymástól*, ezért minden taszkhoz **védelmi táblák** kerülnek felépítésre. A táblákban található **deszkriptorok** (infók) határozzák meg a taszkok hozzáférési jogosultságait (olvasási/írási/végrehajtási).

A program és a memória logikai egységekre, **szegmensekre** van osztva. *A szegmensek egymás memóriaterületeit nem zavarhatják*. A folyamatokat úgy védjük, hogy minden egyes folyamathoz ***szegmensleíró táblát*** rendelünk.

A vezérlés privilegizálási szintje mindig ***ellenőrzésre*** kerül a ***kapukon*** történő áthaladása során. Kapu-típusok: *call*-kapu (taszkok közötti paraméterátadásnál), *megszakítási* kapu, *trap*-kapu, *task*-kapu.

B tételsor

FőbbTárgyak()
{

- Programozási alapok,
- Adatbázisok,
- Programozási technológia,
- Objektumorientált szoftverfejlesztés,
- Az informatikai biztonság alapjai;

}

1. Az algoritmus és a program fogalma, jellemzői. Az algoritmustervezés helye és szerepe a szoftverfejlesztésben. Algoritmusok építő elemei. Algoritmuslépések és programutasítások kapcsolata. Programvezérlési szerkezetek egy választott programozási nyelvben

1.1. Az algoritmus és a program fogalma, jellemzői

Algoritmus: egy feladat megoldását eredményező ***véges számú, egyértelmű szabályokkal megfogalmazható lépések*** sorozata.

Jellemzői:

- ❖ véges számú lépésekből (elemi tevékenységekből, instrukciókból, utasításokból) áll
- ❖ minden lépésnek egyértelműen végrehajthatónak kell lennie (a végrehajtó egység. minden lépés után eldönti, mi lesz a következő lépés)

- ❖ hivatkozhatunk összetett lépésekre is (külön megadhatjuk)
- ❖ a végrehajtandó instrukciónak valamilyen célja van. (végrehajtás során valamilyen változás következik be. Általában megváltoznak az adatok értékei)
- ❖ általában vannak bemenő (input) adatai, melyeket felhasznál.
- ❖ legalább 1 kimenő (output) adatot eredményeznie kell.
- ❖ használhat ciklusokat, rekurziót a tömörség végett
- ❖ általános legyen, ne csak egyetlen adatra használható
- ❖ nem függ programnyelvtől, platformtól
- ❖ elronthatatlannak kell lennie

Program: Az algoritmus **megfogalmazása** a számítógépek vagy a *fordító-programok* ***számára érthető*** *nyelven*, ennek *elemi lépéseit* nevezzük utasításoknak.

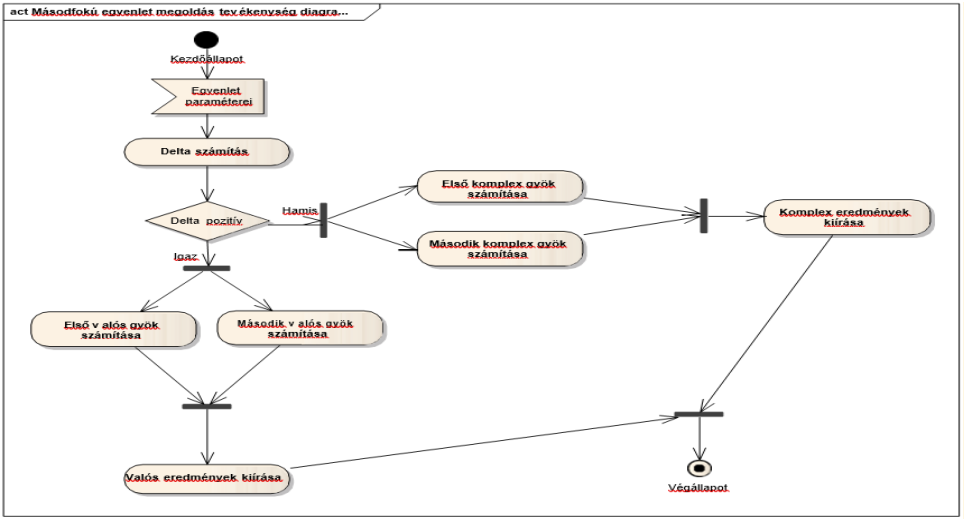
Jellemzői:

- ❖ összetett lépésekből (algoritmus) áll.
- ❖ az utasítás. végrehajtásának mindig van tárgya. Ezeket a tárgyakat a programozásban adatoknak nevezzük.
- ❖ mindig van célja.
- ❖ felhasználóbarátnak kell lennie

1.2. Algoritmustervezés helye és szerepe.

Az algoritmus kigondolása után az ***grafikus megjelenítési formát ölt*** a tevékenységdiagramban. Itt mindenekelőtt ügyelni kell az átláthatóságra. A következő fázis lehet a *pszeudokód*.

Tevékenység-diagram: algoritmus leírására szolgáló grafikus jelölésrendszer. Segítségével a program dinamikus viselkedését tudjuk ábrázolni. A probléma megoldásának a lépéseit szemlélteti, a párhuzamosan zajló vezérlési folyamatokkal együtt.



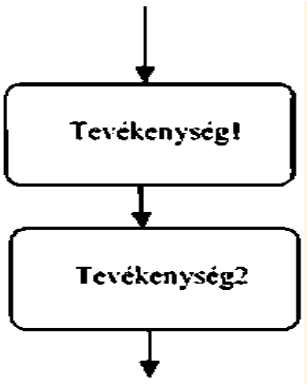
10. ábra – Algoritmus tevékenység-diagramja

1.3. Az algoritmusok építőelemei:

Vezérlőszerkezetek:

- **Szekvencia**: lépések, utasítások egymás utáni végrehajtása. A szekvenciát a nyilak irányában felsorolt egymás utáni tevékenységek alkotják.

```
Tevékenység1;
Tevékenység2;
```

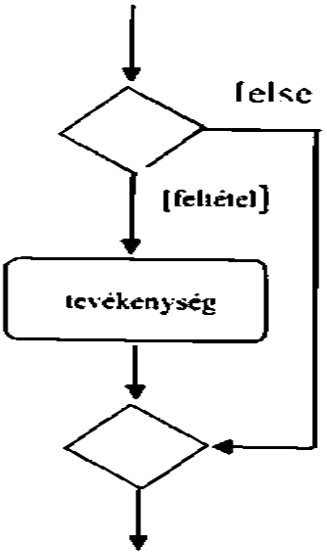


11. ábra – Szekvencia, sorrendi végrehajtás

- **Elágazás** (szelekció): programelágazást jelent, egy adott ponton a tevékenységek végrehajtása feltételektől függ. Szelekció esetén a tevékenységből vagy az elágazási pontból kifelé vezető nyilak mindegyikén szerepel egy feltétel. Ha igaz egy adott feltétel, akkor a feltétel vonalának irányában halad tovább a vezérlés, a megirányzott tevékenység hajtódik végre. Megadható egy „egyébként" ág (*else*), ha a többi feltétel nem teljesül, akkor erre az ágra kerül a vezérlés.

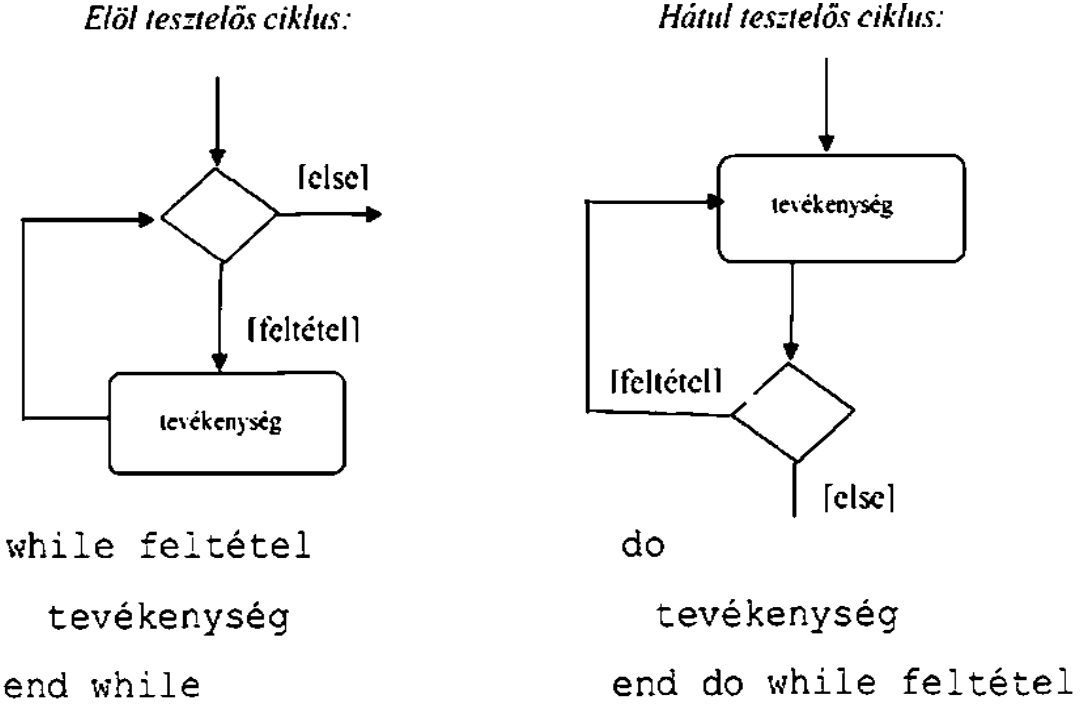
```
if feltétel
     tevékenység
end if

if feltétel1
     tevékenység1
else if feltétel2
     tevékenység2
end if
```



12. ábra – Feltételes szerkezet, elágazás

- **Ciklus** (iteráció): Bizonyos tevékenységek ismételt végrehajtása. Az ismétlendő tevékenységet ***ciklusmagnak*** nevezzük. Iteráció esetén a vezérlés ismételten visszatér *a ciklusmag elé*, de az újbóli végrehajtás előtt egy **elágazási pont** található, mely megengedi a ciklus elhagyását. Ha a vezérlés nem tudja elhagyni a ciklust, ***végtelen ciklusról*** van szó. Két fajtája létezik:
  - **Elől-tesztelő**: A *ciklusmag előtt* tesztel. A ciklusmag akkor kerül végrehajtásra, ***ha a feltétel teljesül***. Elképzelhető, hogy a ciklusmag sohasem hajtódik végre!
  - **Hátul-tesztelő**: a ciklusmag után vizsgálja a feltételt, és a ciklusmag ***egyszer*** mindenképpen végrehajtódik! („`do while`” utasítás; a „*while*” szó jelentése: amíg).



13. ábra – Elöl- és hátul-tesztelő ciklusok

- **Feltétel nélküli ugrás** (GO TO): vezérlés átadása az algoritmus egy megadott pontjára (nem minden nyelvben használható)

A csak szekvencia, szelekció, iterációból építkező programokat strukturált programnak nevezzük.

Változók: olyan ***memóriaterületek***, amelyek ***értékeket vehetnek fel***.

Típus: Minden változónak jól meghatározott típusa van. A változó csak a típusának megfelelő értéket vehet fel.

Értékmegadás: Az algoritmus változóit deklarálni kell, megadni neveiket és típusaikat.

1.4. Algoritmuslépések és a programutasítások kapcsolata.

Az algoritmusainkat *szekvenciákból*, *elágazásokból*, ill. *iterációból* építjük fel. Ha program utasításait egyszerűen *egymás után* kell végrehajtanunk, akkor **szekvenciát** írunk. Előfordulhat, hogy *egy feltételtől függ* az, hogy egy utasítást *végre kell-e hajtanunk*, vagy hogy melyik utasítást kell végrehajtanunk. Ekkor alkalmazzuk a **szelekciót** (elágazás). Valamint elképzelhető, hogy egy tevékenységet (programutasítás sorozatot) *többször kell ismételnünk*, ekkor **iterációt**, vagy ciklust alkalmazunk.

1.5. Programvezérlési szerkezetek egy kiválasztott programozási nyelvben (C#)

Deklarálás (létrehozás, értékadás) **és szekvencia** (sorrendi végrehajtás; az utasítások végét *pontosvesszővel* jelezzük):

```
byte szam1 = 254 + 2;
byte szam2 = 3 * 4;
```

Elágazás - *If*

```
// Rövid alak, ha csak egy-egy utasítás van
if (feltétel) utasítás1;
```

```
[else utasítás2];

// Ha nem csak egy-egy utasítás van
if (feltétel){
     utasítások1
     }
[else{
     utasítások2
     }]
```

Elágazás - *Else if* – A nyelvben nincs elif vagy elseif kifejezés, ezt így oldhatjuk meg (lényegében itt egymásba ágyazott elágazásokról van szó):

```
if (feltétel){
utasitasok1
     }
else if(feltétel1){
     utasitasok2
     }
else if(feltétel2){
     utasitasok2
     }

// ...
else if(feltétel***n***){
     utasítások***n***
     }
[else {
     utasítások2
     }]
```

A ***switch***-szerkezettel ***többszörös elágazás*** valósítható meg, ha a kifejezés *egész* vagy *string* típusú. A futás ahhoz a *case*-blokkhoz kerül, ahol a kifejezés *megegyezik* a megadott *konstans-kifejezéssel*. A blokk végrehajtása végén *ugró utasítás* meghatározza a következő blokk

végrehajtását. Ezt mindig meg kell adni, ami vagy egy másik *case-blokk*, vagy a *default* ág.

```
switch (kifejezés){
     case konstanskif: utasítások; ugró_utasítás;
     [default: utasítások; ugró_utasítás;]
}
```

Ciklus – *For* – A ***for*** utasítással lehet a ciklust megvalósítani. Az ***inicializációs*** *részben* kell a változó(ka)t (vesszővel elválasztva, ha több van) *kezdeti értékre* állítani, a ***léptetés***-részben kell változtatni az értéküket. A ciklus addig tart, amíg a kifejezés értéke *hamis* nem lesz. Futtatási sorrend: *init_rész*, *kifejezés-kiértékelés*: ha igaz, utasítás/blokk, majd *léptetés*, és ismétlés a kifejezés-kiértékeléstől.

```
for([init_rész],[kifejezés],[léptetés]) utasítás/blokk;
```

Ciklus – *foreach* – Újdonság a `foreach(típus azonosító in collection)` törzs alakú szerkezet, mellyel egy olyan objektum elemeit járhatjuk be, amelyek az ***IEnumerable iterface***-t valósítják meg. Az azonosítóval jelölt változó tartalma a *végrehajtás során éppen aktuális elemet* tartalmazza, azaz egy iterátor. Fontos, hogy ez a változó nem módosítható, illetve nem adható át *ref*-, valamint *out*-paraméterként.

```
double[,] values = { {1.2, 2.3, 3.4, 4.5}, {5.6, 6.7, 7.8, 8.9}};

foreach (double elementValue in values)
{
     Console.Write("{0} ", elementValue); // -> 1.2 2.3 3.4 4.5 5.6
     6.7 7.8 8.9
}
```

A *foreach* vezérlési szerkezete valójában úgy működik, hogy meghívja a végigjárandó osztály ***GetEnumator()*** függvényét, ami visszaad egy

bejárót. Tehát a *foreach* nem más, mint egy ***while*-ciklus**, csak ügyesen szervezve.

```
// Implementáció
ArrayList list = new ArrayList();

// ...
foreach(object obj in list)
     {
               DoSomething(obj);
     }

// A fordított átmeneti kód
Enumerator e = list.GetEnumerator();
     while(e.MoveNext())
     {
               object obj = e.Current;
               DoSomething(obj);
     }
```

A C# 2.0-ban az iterátor már nem ilyen erőltetetten van megvalósítva. A foreach-nak a feltétele csupán egy ***GetEnumerator()***-függvény megléte. Igaz, ez egy speciális függvény, mivel a visszatérési értékének **IEnumerator<>**-nak kell lennie, a <> között pedig az értékek típusával. A függvényben a *visszatérési kulcsszót* (return) a ***yield*** kulcsszó helyettesíti. A kulcsszó az értékeket ismétlődően adja vissza. A fordító ezekből egy jóldefiniált felsorolási típust készít. Példa:

```
public class Names
     {
     private List<string> names = new List<string>();
               public Names()
               {
               this.names.Add("John");
               this.names.Add("Smith");
               this.names.Add("Franklin");
               }
```

```
    public IEnumerator<string> GetEnumerator()
              {
              for(int i = 0; i < this.names.Count; i++)
                   {
                   yield return this.names[i];
                   }
              }
    }
```

Ciklus – *while* – A **while** utasítással lehet az elől-tesztelős ciklust megvalósítani. Hasonlóan a for ciklushoz, ha nem igaz a feltétel, akkor előfordulhat, hogy egyszer sem fut le. Ellenben a do-while ciklus egyszer mindenképp lefut.

```
while(feltétel)
     {
     // utasítások
     }

// Rövid alak
while(feltétel)
     EgyUtasítás;
```

Ciklus – *do* – A ***do-while*** utasításokkal lehet ***hátul-tesztelős ciklust*** írni. Specialitása, hogy egyszer mindenképp lefut.

```
do
     {
     // utasítások
     }
while(feltétel);
```

2. A típus és a változó fogalma. Egyszerű és összetett adattípusok. Adatok láthatósága az objektumokban. Közvetlen és közvetett hivatkozású (referencia/dinamikus) változók. Az SQL adattípusai

2.1. A típus és a változó fogalma

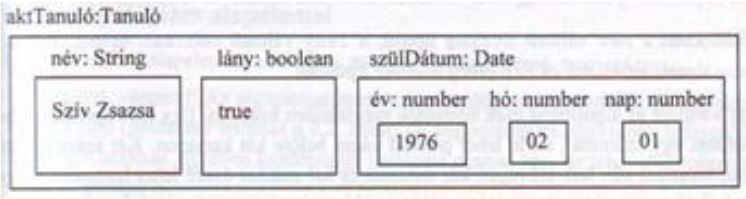
Típus: a változó ***tulajdonsága***, amely meghatározza a lefoglalt memóriaterület nagyságát (van 1 byte-os, 2 byte-os, 4 byte-os stb.), másrészt meghatározza, hogy az adatot hogyan lehet kezelni, egész számként, valós számként, karakterkódként stb., harmadrészt meghatározza, hogy *milyen műveletek* végezhetők az adattal. Vannak beépített (standard) típusok, valamint mi is létrehozhatunk újabbakat. Az új típusok definiálása a - `type` - paranccsal történik. Minden változónak van egy jól meghatározott típusa. A változó csak a típusának megfelelő értékeket vehet föl.

- *Number* (szám), egyszerű (primitív) adat; pl. 45; 99.9; 10000.
- *Boolean* (logikai) egyszerő (primitív) adat; értéke true (igaz) vagy false (hamis).
- *String* (szöveg) összetett típus; pl. Szív Zsazsa, Egri Kata.
- *Date* (dátum), összetett típus (év, hó, nap); pl. 1978,06,20

Változó: olyan memóriaterületek, amelyek különböző értéket vehetnek fel. Jellemzői az *azonosítója*, adott méretű *memóriahelye*, *típusa* és *aktuális értéke*.

Deklarálás: Az algoritmus változóit deklarálni kell, meg kell adni *neveiket* és *típusaikat*.

Inicializálás: amikor egy változó definiálásával egyidejűleg ***értéket is adunk*** a változónak.



A program által használt összes változót deklarálni kell, vagyis meg kell adni a nevét és a típusát! Először megadunk egy típust, aztán felsorolunk egy vagy több azonosítót, vesszővel elválasztva: `double fejadag, osszesen`. Amikor a változónak kezdeti, induló-értéket adunk, akkor a változót inicializáljuk! `double fejadag=0.4, osszesen;`

Konstans: Ez egy megváltoztathatatlan változó.

2.2. Egyszerű és összetett adattípusok

Egyszerű (primitív) adattípusok: Memóriaterülete oszthatatlan, egy „élettelen", viselkedés nélküli tulajdonságot tárol.

- *Egész típusok:*

Típus neve:	Foglalt memória:	Legkisebb érték:	Legnagyobb érték:
byte	1 bájt	- 128	127
short	2 bájt	- 32.768	32.767
int	4 bájt	- 2147483648	2147483647
long	8 bájt	-10^{19}	$\sim 10^{19}$

14. ábra – Egyszerű adattípusok

Összetett (referencia) adattípusok: Olyan mutató, mely egy objektum hivatkozását tartalmazza. Az objektum összetett memóriaterület, több primitív típusú változót, valamint további referenciákat is tartalmazhat. Tömb, Interfész, Osztály

2.3. Adatok láthatósága az objektumokban

Láthatóság: Az osztály deklarációi a következő láthatósággal (hozzáférési móddal) rendelkezhetnek:

- **Nyilvános** (public): minden kapcsolatban álló kliens eléri és használhatja, UML jelölése +
- **Védett** (protected): hozzáférés csak öröklésen keresztül lehetséges, UML jelölése #
- **Privát** (private):az osztály privát deklarációja, csak az osztály saját metódusai férhetnek hozzá, UML jelölése -

Az **objektum** a valós világ **egy egyedét** reprezentálja, attribútumokból és módszerekből áll, egyértelműen azonosítható, rendelkezik az *egységbezárás, elrejtés, üzenetküldés, öröklés* és *polimorfizmus* tulajdonságaival. Az azonos felépítésű objektumok egy osztályba tartoznak. Minden művelet, amit egy objektumon végre kell hajtani, egy metódust képvisel. Minden objektumnak van egy állapota, amelyet az attribútumok pillanatnyi értéke határozz meg. Az objektumok szerkezetüket tekintve zártak, azaz az attribútumok és a törzsét alkotó utasítások közvetlenül nem érhetők el egyetlen másik objektumból sem, de egy objektum egy másik objektum metódusait üzenetküldéssel aktivizálhatja.

Bezárás (zárójelezés): A bezárás az adatok és metódusok összezárását, egységbezárását jelenti (Egységbezárás elve!). Ily módon az objektum zárt és sérthetetlen lesz, tehát bizonyos metódusok meghívásával nem tud elromolni az objektum egysége. Más szóval információkat rejtünk el a kliens elől, amelyeket *csak az interfészen* keresztül lehet megközelíteni. Fontosabb szabályok:

- Az objektumnak van egy interfésze, amely a programozó által kijelölt metódusok összessége.
- Az objektumot csak az interfészen lehet megközelíteni.
- Az adatok csak metódusokon keresztül érhetők el
- Az objektum interfész része a lehető legkisebb

2.4. Közvetlen és közvetett hivatkozású (referencia/dinamikus) változók

Az objektumorientált programozásban alapvetően két típus létezik: primitív- és referencia-típus.

- **Primitív-típus**: egy primitív típusú változó azonosítójával közvetlenül hivatkozhatunk a változó memóriahelyére. Ezt a helyet a rendszer a deklaráció utasítás végrehajtásakor foglalja le. A programozó nem definiálhat primitív-típust.
- **Referencia-típus**: a referencia-típusú változók *objektumokra mutatnak*. Egy referencia típusú változó azonosítójával az objektum memóriahelyére közvetve hivatkozhatunk egy referencián (hivatkozáson) keresztül.

2.5. Az SQL adattípusai.

Az 1. szabvány SQL-nyelv elemei:

- Alapelemek:
  - Az SQL **táblákat** (relációkat) **kezel**. Elnevezésük: *table*. A tábla azonosítója (neve) betűvel kezdődik.
  - A tábla **attribútumait** *oszlopoknak* nevezzük, amelyek oszlopazonosítóját, típusát és hosszát a tábla definiálásakor adjuk meg.
  - A *táblák névvel ellátott együttese* az ***adatbázis*** (database).
- Táblafajták
  - **Adattábla**: a valódi relációk, melyek a karbantartott adatokat tartalmazzák (A-tábla).
  - **Eredménytábla**: adat- vagy eredménytáblából lekérdezéssel nyert tábla, elmenthető (E-tábla).
  - **Nézettábla**. Virtuális tábla, amelyről csak a definíciója tárolódik (V-tábla)

- **Index**: automatikus a használata, tehát csak a létrehozásáról kell gondoskodni (I-tábla).
- **Szinonima**: létező adat- vagy nézettábla átnevezése (S-tábla).

❖ Adattípusok
- SMALLINT: 6 számjegyű egész szám (az előjelet is beleszámítva): -12 345 – 123 456
- INTEGER: 11 számjegyű egész szám (az előjelet is beleszámítva): -1 234 567 890 – 12 345 678 901
- DECIMAL(x,y) x számjegyű, fixpontos decimális szám, y tizedesjeggyel: x= 1-19, y=0-18
- NUMERIC(x,y) Előjellel és a tizedes jellel együtt x számjegyű, fixpontos decimális szám, y tizedesjeggyel: x=1-20, y=0-18
- FLOAT(x,y): Előjellel és a tizedes jellel együtt x számjegyű lebegőpontos szám, y tizedesjeggyel: x=1-20, y=0-18
- CHAR(x): karakterlánc(string): x=1-254
- DATE: dátum
- LOGICAL: logikai érték, t az igaz érték(true), n a hamis érték (false)

Az SQL ***befogadó**-típusú* (host) nyelv, azaz más nyelvbe való beépülésre készült, így az SQL-nyelvben *nem lehet változókat definiálni*, csak a behívó nyelv változóit (vendégváltozók, *host-variable*) kezeli.

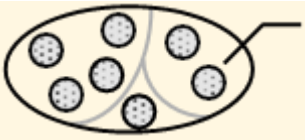
3. Adatszerkezetek (tömb, verem, sor, lista, kollekció-keretrendszer, tábla, gráf, fa). Létrehozásuk, feldolgozásuk, bejárásuk, adattárolás lehetséges módszerei, indexelés)

Az adatszerkezet egymással kapcsolatban álló adatok, objektumok összessége. Feladata az adatok hatékony, szervezett tárolása és kezelése. A kapcsolatokban részt vevő struktúraelemeket csomóponti

adatoknak nevezzük. A csomópontokat körökkel, azok közötti kapcsolatokat pedig nyilakkal ábrázoljuk. Az adatmodell működését az objektumok közötti relációk határozzák meg. A kapcsolatok alapján az adatszerkezetek négy fő csoportba sorolhatók:

3.1.1 Asszociatív adatszerkezetek

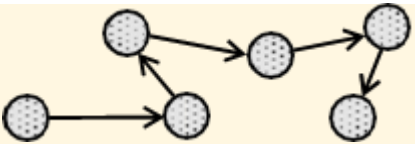
A struktúraelemek közötti kapcsolatokat az elemek azonos tulajdonságértékei létesítik. Az elemeket e tulajdonságok alapján csoportosítjuk. A kapcsolatok az asszociatív (csoportosítható) adatszerkezetekben a leglazábbak. Asszociatív adatszerkezet memóriában a tömb, a ritka a mátrix és a különböző táblák, külső tárolóeszközön pedig a direkt szervezésű állomány.



15. ábra – Adatszerkezetek (több ábra is van!)

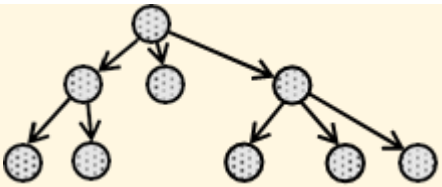
3.1.2 Szekvenciális adatszerkezetek

Az egyes struktúraelemek egymás után helyezkednek el. Mindig van egy kezdő elem, és minden elemet a struktúra egy jól meghatározott eleme követ. A kapcsolat egy–egy-jellegű: minden elem csak egy helyről látható, és minden elem csak egy elemet lát. A memóriában megvalósítható szekvenciális adatszerkezetek a jelsorozat, a verem és a sor, külső tárolóeszközön ilyenek a szekvenciális és láncolt adatállományok.



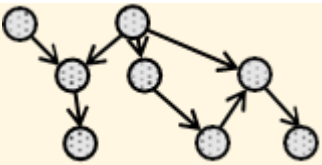
3.1.3 Hierarchikus adatszerkezetek

A struktúraelemek hierarchikusan egymás alá vannak rendelve. A kapcsolatok jellege egy-sok: minden csomópont csak egy helyről látható, egy csomópontból viszont sok csomópont látható. A hierarchikus adatszerkezeteket a belső tárban fának, a külső tárolókon hierarchiaállománynak nevezik.



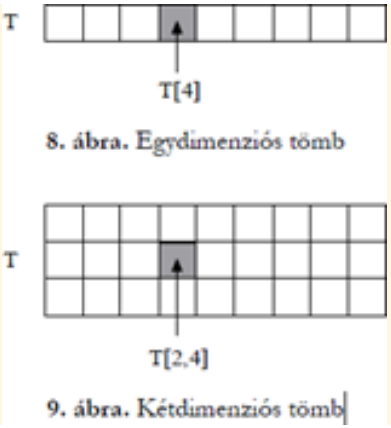
3.1.4 Hálós adatszerkezetek

Hálós adatszerkezetek esetén bármelyik csomópont bármelyik csomóponttal kapcsolatban állhat. A kapcsolatok sok-sok típusúak. A hálós adatszerkezeteket a belső tárban irányított gráfnak, ill. hálózatnak, a külső tárolókon sémának nevezik.



3.2. Tömb

A tömb egy asszociatív adatszarkezet, lényege, hogy elemeit indexeken keresztül érjük el. A tömb használatának hátránya, hogy méretét előre meg kell adni és az indexhatárok dimenziónként kötöttek. Az egydimenziós tömb egy adatsor (egy index) (8. ábra), a kétdimenziós egy téglalapalakú adattáblázat egy sor és egy oszlopindex) (9. ábra). A tömbelemek típusa bármilyen lehet, akár összetett is. A tömb adatcsoport jellemző kezelési módja az elemenkénti feldolgozás, tehát a műveletekben a tömbelemek az operandusok.



16. ábra – Tömbök

Tömbök tárolása: Sorfolytonosan: ez lehetővé teszi, hogy bármely elem helyét az indexek ismeretében kiszámíthassuk, s így közvetlenül elérhessük őket. A tömb méretétől függetlenül bár-melyik elemét ugyanannyi idő alatt érhetjük el. Elemei indexeléssel közvetlenül címezhetők

Tömb megszüntetése: Java-ban a tömb által lefoglalt memóriaterületet csak az automatikus szemétgyűjtő algoritmus szabadítja fel. Kikényszerítés: `System.gc();`. Destruktor egyébként minden programnyelvben van, kivéve a Java-ban. Tehát a tömb megszüntetéséről nekünk kell gondoskodni.

Tömb deklarálása, ekkor meg kell adnunk a tömb:

- nevét
- elemeinek típusát
- indextartományát
- dimenzióját.

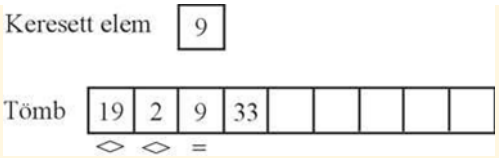
Csoportosítás tömbdimenziók száma alapján:

- Egydimenziós tömb: vektor, sorozat.
- Kétdimenziós tömb: mátrix, táblázat.

3.2.1 Tömbök feldolgozása

- A tömbök feldolgozása gyakran jelenti ***ugyanazon műveletek ismétlését*** az egyes elemekre, ez pedig tipikusan valamilyen ciklus alkalmazásához vezet.
- Különösen ***a léptető (for-) ciklus*** kötődik szorosan a tömbökhöz, mert a ciklusváltozó jól használható a tömbelemek egymás utáni megcímzésére.

Keresés tömbben: Egyenként meg kell vizsgálnunk a tömb elemeit. A vizsgálat akkor érhet véget, ha megtaláltuk az értéket, vagy már nincs több vizsgálandó elem. A vizsgálatot a leggyakoribb és legegyszerűbb módon, növekvő indexsorrendben végezzük.



17. ábra – Keresés a tömbben

Ritka Mátrix:

- Olyan többdimenziós tömb, ahol a tömb nagy része kihasználatlan
- Tárolás inkább láncolt listában ajánlott.

3.3. Tábla

A tábla egy **asszociatív adatszerkezet**, melynek elemei kulcs és adat párok, ahol a kulcsok egyediek, és bármely elem ***a kulcsán*** keresztül érhető el. A táblával kapcsolatos műveletek központi kérdése az adott kulcshoz tartozó adatok minél rövidebb idő alatt történő megkeresése, és a tábla karbantartása. A tábla tárolása vektorban és listában is megvalósítható. A táblára érvényes szabályok két csoportra bonthatók:

- A **logikai** szabályok minden táblára igazak fizikai megvalósítástól függetlenül (pl. ugyanaz a kulcs nem vehető fel kétszer).
- A **fizikai** szabályok az adott tábla fizikai megvalósításának korlátai (pl. nem vihető fel több adat, ha a tábla betelt).

Műveletek: keresés, beszúrás, törlés és szekvenciális (folyamatos, sorrendi) elérés

- Asszociatív adatszerkezet
- Egy elem: egyedi kulcs + adat
- Tárolás: egydimenziós tömbben vagy láncolt listában

adat	adat	adat	adat	adat	adat	adat
kulcs	kulcs	kulcs	kulcs	kulcs	kulcs	kulcs

„Kérem azt az adatot, melynek kulcsa <...>” ↑

Az oszlopok különböző típusúak tudnak lenni (pl. dátum, egész, tört, adatfolyam, ujjlenyomat képe, titkosított jelszó stb.).

3.4. Verem

A **verem** (*stack*) egy ***szekvenciális adatszerkezet***, melynek *mindig csak* ***a legutoljára betett*** *elemét lehet látni*, illetve kivenni (LIFO). Kell egy mutató, mely a mindenkori verem tetejére, illetve az első szabad helyre mutat. A veremről meg kell tudni állapítani, hogy az *üres vagy tele van*, hiszen üres veremből nincs értelme kivenni, tele verembe pedig nincs értelme betenni elemet.

Műveletek:

- PUSH – elem betétele a verembe, mindig a tetejére
- POP – elem kivétele a veremből, mindig a legfelsőt
- TOP – a legfelső elem lekérdezése, a verem változatlan marad

Az elemek sorrendjét a legkönnyebben verem alkalmazásával fordíthatjuk meg. Ugyanígy nagyon jól használható a verem különböző

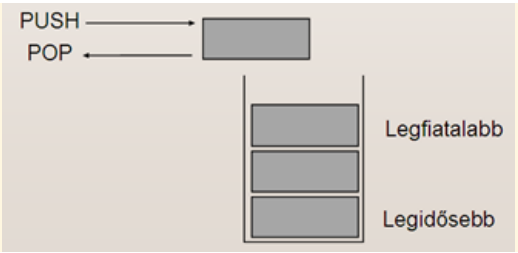
visszatérési utak megjegyzésére is. Tárolása általában *egydimenziós tömbben* vagy egy *irányban-láncolt listában* történik.

A ***tömbnél*** a verem tetején *a tömb legnagyobb indexű* (**utolsó**) eleme van, az aktuális **elemszám** mint *veremmutató* (a verem tetejére mutató index) funkcionál.

- Ráhelyezésnél a tömb aktuális elemszáma eggyel nő, az új elem az utolsó (legfelső) lesz.
- Levételnél az utolsó elemet vesszük le, az aktuális elemszám eggyel csökken

A (***láncolt***) ***listánál*** a verem tetején ***a lista kezdőeleme*** van, a kezdőmutató tölti be a veremmutató szerepét:

- Ráhelyezésnél az új elem a lista elejére kerül.
- Levételnél az első elemet vesszük le.



18. ábra – A verem (lista) működése

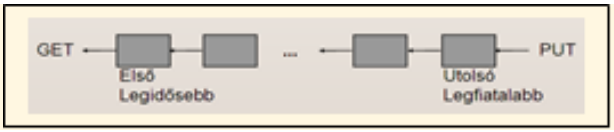
3.5. Sor

A sor (***queue***) egy szekvenciális adatszerkezet, melyből mindig ***a legelsőnek betett elemet lehet kivenni*** (FIFO). Tárolása vektorban és listában egyaránt megvalósítható.

Műveletek:

- PUT – elem betétele a sorba, mindig a sor végére
- GET – elem eltávolítása a sorból, mindig a sor elejéről
- FIRST – az első elem lekérdezése, a sor változatlan marad

Sorokkal tipikusan olyan feladatok oldhatók meg, melyekben az elemek feldolgozása ***érkezési sorrendben*** történik. Sorként működnek a pufferek, mint például a billentyűzet- vagy a nyomtatópuffer.

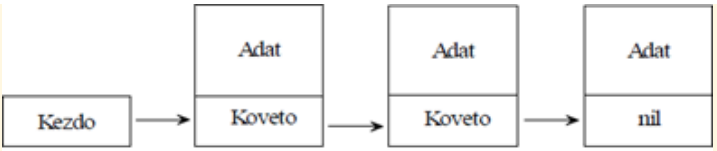


19. ábra – A sor működése

3.6. Lista

A láncolt listák **szekvenciális adatszerkezetek**, a sorrendi kapcsolatot (előző, követő) ***a lista elemeiben elhelyezett mutatók*** hozzák létre. A legegyszerűbb láncolt lista az egy-irányban láncolt (vagy röviden ***egyirányú***) lista.

- Gazdaságos memóriafoglalás, egyszerű karbantartási műveletek (törlés, beszúrás).
- Szekvenciális adatszerkezet.
- Listafej: a lista *első* elemére mutat.
- Végjel (null): speciális mutatóérték, az utolsó elem mutatórészében állva *jelzi a lánc végét*.



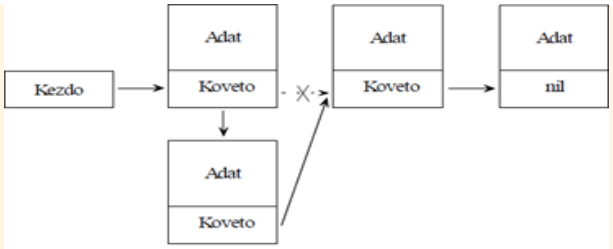
20. ábra – Egy irányban láncolt lista

További láncoltlista-változatok:

- **rendezett** láncolt lista: nem utólag rendezzük a listát, hanem az új elem felvitele a rendezettség megtartása mellett történik;
- **ciklikusan** láncolt lista: az utolsó elem mutatója az első elemre mutat;
- **két irányban láncolt** lista: egy listaelemben két mutató;

- **többszörösen láncolt** lista: több láncolat mentén is bejárható, azaz több szempont szerint is rendezett, egy elemben több mutató.

A láncolt listáknál **nincs indexe** az elemnek, a hozzáférés alapvetően soros, vagyis, ha egy elemet el akarunk érni, ahhoz végig kell járnunk a megelőző elemeket is.



21. ábra – Beszúrás egy 1 irányban láncolt listába

A láncolt listák előnyös tulajdonságai leginkább a ***módosító–karbantartó-jellegű*** feladatoknál mutatkoznak meg. Míg a tömböknél egy elem beszúrása vagy a törlése szükségszerűen ***az elem helyétől függő mennyiségű adatmozgatással jár*** (gondoljuk meg pl., hogy az első tömbelem törléséhez az összes többit „át kell pakolni"), addig a láncolt listáknál a beszúrás és a törlés az elem helyétől függetlenül, bármely elemre nézve is csak néhány műveletet igényel, hiszen egy-két mutató átírásával megvalósítható (29. és 30. ábra).

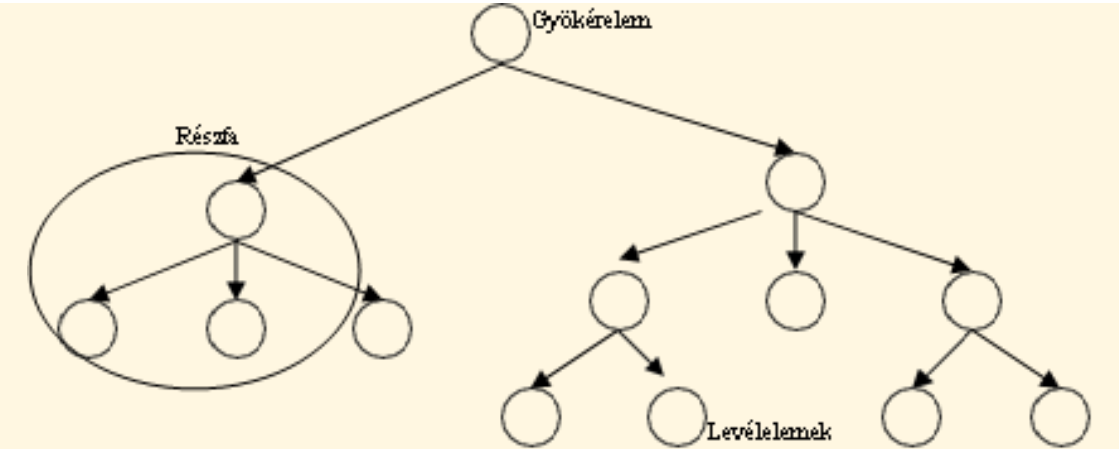
3.7. Fa

A fa egy **hierarchikus adatszerkezet**, amelyben egy elemnek *akárhány rákövetkezője*, de minden elemnek csak ***egyetlen megelőzője*** létezik. A fa egy olyan dinamikus, homogén adatszerkezet, amelyben minden elem megmondja a rákövetkezőjét. Elnevezések:

- **Gyökérelem**: a fa azon eleme, amelynek *nincs megelőzője*. A legegyszerűbb fa *egyetlen gyökérből* áll. Mindig csak egy gyökérelem van, de az kötelezően, kivétel az üres fa, ahol egy sincs.

- **Levélelemek**: a fa azon elemei, amelyeknek *nincs rákövetkező-jük* (leszármzottjuk).
- **Közbenső elem**: a fa nem gyökér-, illetve levélelemei, hanem az összes többi. Megelőző-eleme és rákövetkező-elemei is vannak.
- **Út**: egy olyan szekvenciális (folyamatos, sorrendi) adatelem-sorozat, lista, amely a gyökérelemtől kiinduló, különböző szinteken átmenő, és levélelemben véget érő, ***egymáshoz kapcsolódó él-sorozat***. Az út hosszán az adott útban található ***élek számát*** értjük. Minden levélelem a gyökérelemtől kiindulva pontosan *egy úton* érhető el
- **szülő**: ha A csomópontból él mutat B csomópontba, akkor ***A szülője B***-nek;
- **gyerek**: ha A csomópontból él mutat B csomópontba, akkor ***B gyereke A***-nak;
- **testvér**: egy szülőhöz tartozó csomópontok.

A fákkal kapcsolatban beszélhetünk szintekről, egy elem szintje megegyezik a gyökérelemtől vett távolságával. A ***0. szinten a gyökérelem*** van, az első szinten a gyökérelem „leszármazottai" vannak stb. A maximális szintszámot a fa magasságának vagy mélységének nevezzük. Minden közbenső elem egy részfa gyökereként tekinthető, így a fa részfákra bontható.



Bináris fa: minden csomópontnak, elemnek ***max***. 2 gyereke van.

Szigorúan bináris fa: a levélelemek kivételével minden csomópontnak ***pontosan két gyereke*** van. A bináris fa megvalósítása:

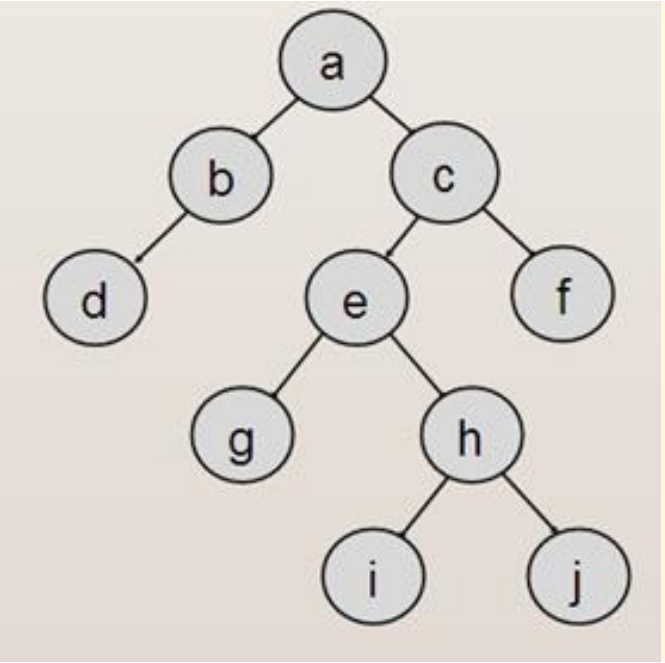
- Tárbeli megvalósítása a láncolt listában alkalmazott adatszerkezet bővítésével: minden elemnek két rákövetkezője lehet, ezért két mutatót alkalmazunk a csomópontokban, az egyik a baloldali, a másik a jobboldali részfa gyökerére mutat.
- Ha valamely mutató értéke a VégJel (null), akkor ebben az irányban nincs folytatása a fának.
- A levélelemek mindkét mutatója VégJel.

Műveletek: *nincs indexelés*

- **Lekérdező** műveletek:
  - Üres-e a fa-struktúra
  - Gyökérelem lekérdezése
  - Meghatározott elem megkeresése, az arra vonatkozó referencia (hivatkozás) visszaadása
- **Módosító** műveletek:
  - Üres fa létrehozása (*_konstruktor*)
  - Új elem beszúrása
  - Meghatározott elem kitörlése
  - Összes elem törlése
  - Egy részfa törlése
  - Részfák kicserélése egymással
  - Gyökér megváltoztatása

3.7.1 Fák bejárása

Bejárásnak nevezzük azt a folyamatot, amikor *a fa minden elemét pontosan egyszer érintve* ***feldolgozzuk***.

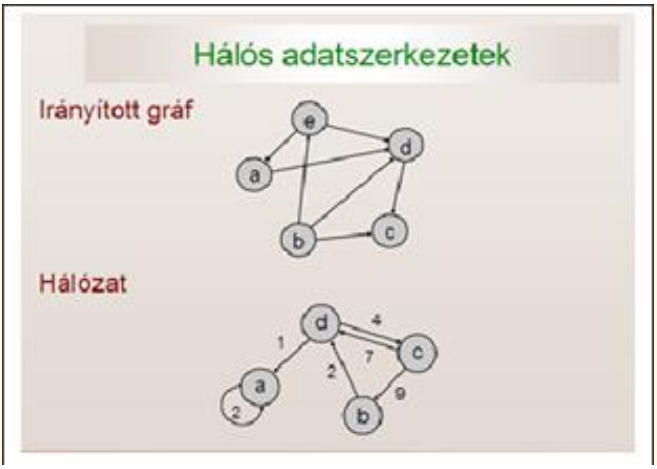


22. ábra – Fa-struktúra bejárása

- **preorder** (Gyökérkezdő) bejárás: a b d c e g h i j f
- **postorder** (Gyökérvégző) bejárás: b c a , d b e f c a , d b g h e f c a , d b i j g h e f c a Bináris fa bejárása:
- **preorder** (Gyökérkezdő) bejárás: a b d c e g h i j f
  - gyökérelem feldolgozása;
  - baloldali részfa preorder bejárása;
  - jobboldali részfa preorder bejárása;
- **inorder** (Gyökérközepű) bejárás: d b a g e i h j c f
  - baloldali részfa inorder bejárása;
  - gyökérelem feldolgozása;
  - jobboldali részfa inorder bejárása;
- **postorder** (Gyökérvégző) bejárás: d b g i j h e f c a
  - baloldali részfa postorder bejárása;
  - jobboldali részfa postorder bejárása;
  - gyökérelem feldolgozása;

3.8. Gráf

A gráf egy hálós adatszerkezet, adatelemei között a kapcsolat sok-sok jellegű: bármelyik adatelemre több helyről is eljuthatunk, és bármelyik adatelemtől elvileg több irányban is mehetünk tovább.



23. ábra – Gráf

A gráf egy köröket is tartalmazó fa, nincs kitüntetett gyökéreleme, tárolása többszörösen láncolt listában. A gráf bejárása:

- **vak** keresés
  - szélességben először keresés
  - mélységben először keresés (nem teljes, nem optimális)
  - mélységben először keresés, mélységi korláttal (nem teljes)
  - mélységben először keresés, fokozatosan növelt mélységi korláttal
- **heurisztikus** keresés (valamilyen szempont szerint osztályozzuk az egyes csomópontokat és éleket) a már meglátogatott elemeket (körök, hurkok) nem járjuk be újra.

3.9. Kollekció-keretrendszer (Collections Framework)

A ***kollekciók*** (konténerek) olyan objektumok, melyek célja egy vagy több típusba tartozó objektumok memóriában történő ***összefoglaló jellegű tárolása***, manipulálása és lekérdezése. Olyan, mint a könyvespolc vagy a ruhásszekrény. Mindkettő képes objektumok (könyvek, ruhák) tárolására, azonban jelentős különbség van közöttük például az elemek elérésében. Egy szépen elrendezett könyvespolcról bármelyik könyvet gyorsan ki lehet emelni, ráadásul ha ABC-sorrendben vannak felrakva a könyvek, még a keresés is elég gyors. Egy megfelelően mély ruhásszekrény abban különbözik a könyvespolctól, hogy abból ***csak a legutoljára berakott ruha-oszlopot*** tudjuk kivenni, a mögötte levőket nem. Ha a legbelső oszlopra van szükségünk, akkor *ki kell venni az összes előtte levőt*, hogy elérhessük az utolsó elemet.

Az egyféle objektum tárolására kihegyezett kollekciókat ***típusos kollekcióknak*** nevezzük. A típusosság előnye az, hogy az adatok berakásakor *típusellenőrzés* történik, így nem köt ki a zokni a könyvespolcon. Az adatok kiolvasásánál is biztosak lehetünk a kapott elem típusában, így a ruhásszekrénybe nyúlva nem ragadunk meg egy szakácskönyvet.

A *nem típusos* (általános) kollekciókból kiolvasott elemek használatához a kiolvasott típust legtöbbször vissza kell ***kasztolni*** (típus-konvertálni) a tényleges típusra.

A kollekció-keretrendszer egy egységes architektúra, ami a kollekciók megvalósítására, kezelésére szolgál.

Elemei:

- ***Interfészek***: absztrakt adattípusok, amelyek a kollekciókat reprezentálják. Lehetővé teszik a kollekciók implementáció független kezelését.

- ***Implementációk***: a kollekció interfészek konkrét implementációi.
- ***Algoritmusok***: azok a metódusok, amelyek hasznos műveleteket valósítanak meg, mint például keresés, rendezés különböző kollekciókon.

Előnyei:

- Minimálisra csökkenti a programozás-ráfordítást.
- Nem a „csináld magad" szemlélet az uralkodó.
- Növeli a kód minőségét és a sebességét.
- Elősegíti a kód-újrafelhasználást.

3.10. Az adatszerkezetek minősítése

Egy adatszerkezet megválasztásánál, minősítésénél számítástechnikai szempontból három fő tényező veendő tekintetbe:

- A ***tárkihasználás*** és az operatív tárban való ***tárolhatóság***. Számításigényes feladatoknál kiemelt fontosságú, hiszen a háttértár használata nagyságrendekkel lassítja a számításokat.
- A ***karbantarthatóság***, vagyis a változások átvezetésének műveletigénye. Fontossága attól függ, milyen gyakoriak a változások.
- A ***lekérdezhetőség***, vagyis az információkinyerés műveletigénye.

4. Az adatmodell alapelemei. Adatmodell típusok és jellemzőik. A relációs adatmodell fogalma, kulcsok kategóriái, kapcsolatok felállítása. Az adatmodellek és a szakterületi modellek kapcsolata, összefüggése

4.1. Az adatmodell alapelemei

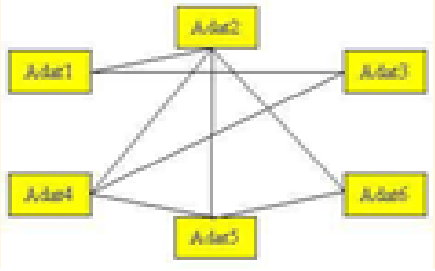
- **Adatmodell**: A valós világ számítógépes leképzésére kidolgozott koncepció, amely modellezési alapelemek, integritási *kényszerek*, *megszorítások* és az alapelemekkel végezhető *műveletek* együttese. A modellezési alapelemek biztosítják az adatok tárolását. Az alapelemek szerkezetének kialakítását az adatok közötti összefüggésnek és kapcsolatoknak megfelelően lehet kialakítani. Az adatmodellek olyan feltételeket is előírnak, melyeket az adatbázisban tárolt adatoknak mindenkor ki kell elégíteni. Ezeket nevezzük megszorításoknak. A megszorítások közül kifejezetten fontosak azok, melyeket az adatok egyértelmű visszakereshetősége és a kapcsolatok hibátlan tükrözése érdekében írunk elő – ezek az integritási kényszerek. Az adatmodell Egy séma, melyben megadjuk mely tulajdonságok határozzák meg az egyedeket, mely egyedek szerepelnek a sémában, és ezek közt milyen kapcsolatok vannak.
- **Egyed**: Egyednek hívunk minden olyan tetszőleges dolgot, objektumot, ami minden más dologtól (objektumtól) megkülönböztethető és amiről adatokat tárolunk. Pl. dolgozó, autó stb.
- **Tulajdonság**: Az egyedeket tulajdonságokkal (attribútumokkal) írjuk le. A tulajdonság az egyed egy jellemzője, ami megadja, meghatározza az egyed egy részletét. Pl. a dolgozó egyednek tulajdonsága lehet: név, fizetés stb.
- **Kapcsolat**: Kapcsolatnak nevezzük az egyedek közötti viszonyt. A kapcsolat mindig valóságos objektumok közötti viszonyt fejez ki.

- *Az 1-1 típusú kapcsolat (1:1)*: Az egyik egyedhalmaz mindegyik eleméhez a másik egyedhalmaznak pontosan 1 eleme kapcsolódik.
- *Az 1-több típusú kapcsolat (1:N)*: Az „A" egyedhalmaz mindegyik eleméhez a B egyedhalmaznak több eleme is tartozik. Pl. VEVŐ:RENDELÉS. 1 vevőhöz több rendelés is tartozhat, míg fordítva nem igaz: 1 rendelés kizárólag 1 vevőtől jön.
- *A több-több típusú kapcsolat (N:M)*: Az „A" egyedhalmaz minden eleméhez a B egyedhalmaz több eleme tartozhat, és fordítva. Pl. TERMÉK:ALKATRÉSZ; Ha végiggondoljuk, 1 termék több alkotóból állhat, és 1 alkatrész is több terméknek lehet az alkotója.

4.2. Adatmodell-típusok és jellemzőik

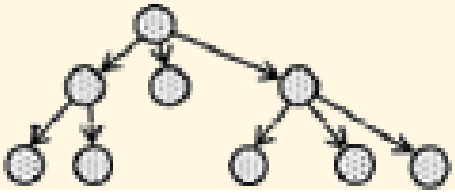
Három adatmodell létezik az alapelemek ***fizikai tárolásától*** függően.

A **hálós modellben** gráffal ábrázolható az adatbázis: a gráf csúcsain az egyed-előfordulások, az éleken a köztük lévő kapcsolat helyezkedik el. Tetszőleges két csomópont között, akkor van kapcsolat, ha őket él köti össze. Egy csomópontból tetszőleges számú él indulhat ke, de egy él csak két csomópontot köthet össze. Azaz minden elem tetszőleges számú másik elemmel lehet összefüggésben, 1:N és N:M-típusú adatkapcsolatok is megvalósíthatóak.



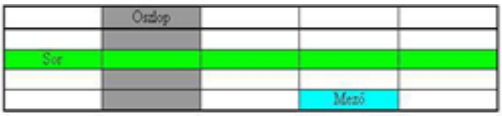
24. ábra – Hálós adatmodell

A **hierarchikus adatmodell** az elemek fölé és alárendeltségi viszonyban vannak, úgynevezett fa elrendezést (struktúrát) valósítanak meg. A fa csomópontjaiban és leveleiben helyezkednek el az adatok. A csomópontok az egyedeket, míg a kapcsolatokat az élek reprezentálják A csomópontok között levő kapcsolat, szülő gyermek kapcsolatnak felel meg. Így csak 1:N-típusú kapcsolatok képezhetők le segítségével. Az 1:N-kapcsolat azt jelenti, hogy az adatszerkezet egyik típusú egyede a hierarchiában alatta elhelyezkedő egy vagy több más egyeddel áll kapcsolatban.



25. ábra – Hierarchikus adatmodell

Relációs adatmodell (asszociatív) – A relációs adatszerkezetben az egyedek közötti kapcsolatokat az egyedek azonos tulajdonság értékei valósítják meg. Az egyedeket a tulajdonságok alapján csoportosítjuk. A relációt (reláció dolgok viszonyát jelenti) táblázat formájában adjuk meg. A táblázat oszlopait a tulajdonságok (attribútumok) neveivel azonosítjuk, melyeknek a reláción belül egyedieknek kell lenniük. A reláció soraiban tárolódnak a logikailag összetartozó adatok az úgynevezett egyed előfordulások. Az előfordulások sorrendje közömbös, de 2 teljesen azonos sor (rekord), nem lehet a táblázatban. A sor és oszlop metszésében található elemet mezőnek nevezzük, a mezők tartalmazzák a tulajdonságok értékeit. A mezők oszloponként különböző típusú (numerikus, szöveges stb.) mennyiségek lehetnek a tulajdonságot leginkább meghatározó érték alapján.



26. ábra – Relciós adatmodell

4.3. A relációs adatmodell fogalma, kulcsok kategóriái, kapcsolatok felállítása.

A korai modellekkel szemben a relációs modellt egy magas szintű lekérdező nyelv támogatja, amely nemcsak a lekérdezést, hanem az adatdefiniálást és az adatmanipulálást is ellátja. Ez a nyelv az **SQL**.

4.3.1 Relációs adatbázisok (oszlopokba szedett adatok összegsége)

A reláció az adatelemek megnevezett, összetartozó csoportjából kialakított **olyan kétdimenziós táblázat, amelyik sorokból és oszlopokból áll**, és ahol az ***oszlopok*** egy-egy **tulajdonságot** írtnak le, a ***sorok*** adják az egyedhalmaz/reláció/táblázat **egyedeit**. Ahhoz, hogy egy táblázatot relációnak lehessen tekinteni, a következő feltételeket kell kielégítenie:

- nem lehet két egyforma sora,
- minden oszlopnak egyedi neve van,
- a sorok és oszlopok sorrendje tetszőleges.

A relációs adatbázisok általában nem 1, hanem ***több***, logikailag összekapcsolható relációból (táblázatból) állnak. A reláció oszlopainak (attribútumainak) számát a reláció fokszámának, sorainak számát a reláció kardinalitásának nevezik.

27. ábra – Adattábla oszlopai, sorai, szerkezete

4.3.2 Kulcsok kategóriái

Elsődleges kulcsnak a reláció azon kulcsát nevezzük, amely ***egyértelműen azonosítja*** a táblázat egy sorát.

Külső kulcsnak vagy *idegen kulcsnak* nevezzük egy relációnak azokat az attribútumait, amelyek ***egy másik relációban kulcsot alkotnak***.

4.3.3 A kulcsok csoportosítása

Egyszerű kulcs: egyetlen attribútumból áll. A reláció kulcs a reláció egy sorát azonosítja egyértelműen. A reláció – definíció szerint – ***nem tartalmazhat két azonos sort***, ezért minden relációban létezik kulcs. A reláció kulcsnak a következő feltételeket kell teljesítenie:

- az attribútumok egy olyan csoportja, melyek csak egy sort azonosítanak (***egyértelműség***)
- a kulcsban szereplő attribútumok egyetlen részhalmaza sem alkot kulcsot
- a kulcsban szereplő attribútumok értéke nem lehet definiálatlan (NULL)

```
SZEMÉLY_ADATOK=({ SZEMÉLYI_SZÁM, SZÜL_ÉV, NÉV})
```

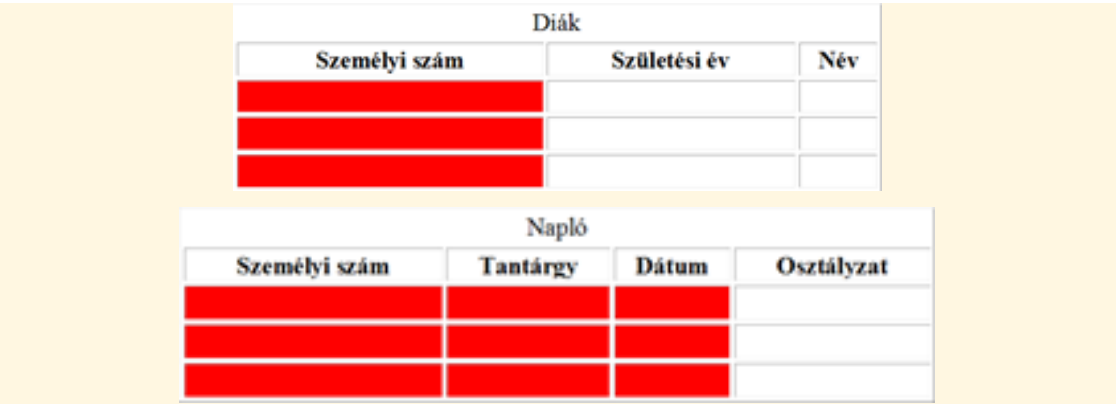
Személyi szám	Születési év	Név

28. ábra – Adattábla kulcsa

A SZEMÉLY_ADATOK relációban a SZEMÉLYI_SZÁM attribútum kulcs, mert nem lehet az adatok között *2 különböző személy azonos személyi számmal*. A születési év vagy a név nem azonosítja egyértelműen a reláció egy sorát, mivel ugyanazon a napon is született tanulók vagy azonos nevűek is lehetnek az osztályban.

Összetett kulcs: Előfordulnak olyan relációk is, melyekben a kulcs ***több attribútum-érték összekapcsolásával*** állítható elő. Készítsünk nyilvántartást a diákok különböző tantárgyakból szerzett osztályzatairól az alábbi relációval:

```
NAPLÓ=({SZEMÉLYI_SZÁM, TANTÁRGY, DÁTUM, OSZTÁLYZAT)}
```



29. ábra – 2 adattábla összefüggései

A NAPLÓ relációban a SZEMÉLYI_SZÁM nem azonosít egy sort, mivel egy diáknak több osztályzata is lehet akár ugyanabból a tantárgyból is. Ezért még a SZEMÉLYI_SZÁM és a TANTÁRGY sem alkot kulcsot. A SZEMÉLYI_SZÁM, TANTÁRGY és a DÁTUM is csak akkor alkot kulcsot, ha kizárjuk annak lehetőségét, hogy ugyanazon a napon ugyanabból a tantárgyból egy diák 2 osztályzatot kaphat. Abban az esetben, ha ez a feltételezés nem tartható (ennek a rendszer analíziséből kell kiderülnie!), akkor nem csak az osztályzat megszerzésének dátumát, hanem annak időpontját is tárolni kell. Ilyenkor természetesen a NAPLÓ relációt ezzel az új oszloppal ki kell bővíteni. Kapcsolatok felállítása:

- **Egy–egy-típusú kapcsolat**: Az egyik egyedhalmaz mindegyik eleméhez a másik egyedhalmaz pontosan egy eleme kapcsolódik.

- **Egy–több-típusú kapcsolat**: Az egyik egyedhalmaz mindegyik eleméhez a másik egyedhalmaz több eleme is kapcsolódik.
- **Több–több-típusú kapcsolat**: Az A egyedhalmaz minden eleméhez a B egyedhalmaz több eleme tartozhat és fordítva egy B egyedhalmaz-beli elemhez is több A egyedhalmaz-beli elem is tartozhat. Pl. *házak–színek* – egy ház lehet több színű is, és egy szín lehet több házon is.

DOLGOZÓ tábla – AZONOSÍTÓ a kulcsa a relációnak

AZONOSÍTÓ	NÉV	FIZETÉS
001	Nagy	100000
002	Kiss	110000

PRÉMIUM tábla – AZONOSÍTÓ oszlop külső kulcs a táblában

AZONOSÍTÓ	ÖSSZEG	DÁTUM
001	45000	99.01.12
002	50000	99.01.13

30. ábra – Táblák kulcsa(i)

4.4. Az adatmodellek és a szakterületi modellek kapcsolata, összefüggése.

Szakterületi modellen a szakterület objektumait és kapcsolatait tüntetjük fel. Az objektumok az egyedek.

A Szakterületi modell az a közös nyelv, amelynek segítségével a megrendelői és fejlesztői oldal kommunikációja egyértelmű. A Szakterületi modell letisztult tudás a szoftver alkalmazási területéről.

A Szakterületi modell és az implementáció közvetlen kapcsolatban van. A Szakterületi modellt a szakterületi szakértő és a szoftverfejlesztő közösen hozzák létre. (Döntően tábla mellett!)

A Szakterületi modell objektummodell, UML-nyelven megfogalmazva. A Szakterületi modell a szakterületi tudást olyan formában tükrözi vissza, amely érthető és implementálható a szoftverfejlesztő számára. A Szakterületi modell a tudást a hagyományos, szabad szöveges leírásnál formalizáltabban, zártabb, szigorúbb szabályok szerint írja le.

A modell az „üzlet” (***üzleti logika***) által használt fogalmakat, ezek kapcsolatait és hozzájuk kötődő szabályokat írja le. Tehát természetes módon, érthető a megrendelői oldal számára. Kellően formalizált, precíz ahhoz, hogy ennek alapján szoftver működhessen.

A modell direkt módon meghatározza az implementációt, ezért a fejlesztő képes a fejlesztés során felvetődő kérdéseit ***a megrendelő nyelvén*** feltenni.

5. Rutin, metódus, eljárás és függvény fogalma, jellemzőik. Paraméterátadás. Példány és osztálymetódusok. Eseménykezelő metódusok. Függvények az SQL-ben

5.1. Rutin, metódus, eljárás és függvény fogalma, jellemzőik

Rutin: A strukturált programozás alapfogalma, egy programból bizonyos szempontok alapján ***elkülönített forráskódrészlet***, melynek ***nevet*** *adunk* s ezzel a névvel *hivatkozunk rá* a későbbiekben.

Metódus: A objektumorientált programozás alapfogalma, ***utasítások*** (tevékenységek) ***összessége***, amelyet meghívhatunk *a metódus nevére* való hivatkozással. ***Olyan rutin, mely objektum adatain dolgozik***. Az objektumorientált programozásban az *objektumokat* le lehet írni úgy, mint egy ***tudáshalmaz*** (a benne tárolt adatok), illetve az ***azokon végzett műveletek*** (az objektumhoz tartozó rutinok) ***összessége***, ezek lesznek a metódusok. A két fogalom között részhalmaz-viszony, reláció van, azaz ***minden metódus rutin***, de csak az a rutin metódus, ami objektumhoz van kötve.

Egy rutin attól függően, hogy van-e visszatérítési értéke vagy nincs, ***eljárásnak*** vagy ***függvénynek*** szokás nevezni.

Tehát sima procedurális programozásnál mondjuk van egy `'rajzol'` rutinunk, objektumorientált programozásnál vannak objektumaink, amelyeknek meg lehet hívni a `'rajzol'` metódusát. (*Procedurális prog-*

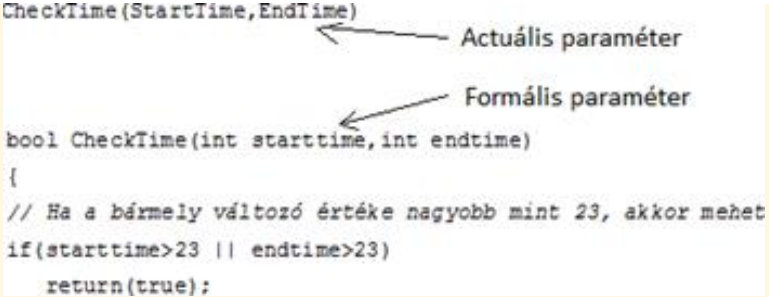
ramozásról beszélünk, ha a programozási feladat megoldását egymástól többé kevésbé független alprogramokból (procedure) építjük fel.)

Eljárás: nincsen visszatérési értéke (***void*** = semleges, üres). Utasítások összessége, melyet egyszerűen végrehajtunk az eljárás nevére való hivatkozással. Végrehajtás után a *progi* azzal az utasítással folytatódik, amelyik követi a metódushívó utasítást Pl: `System.out.println („ezt a println eljárás írta ki”)`; a ***System*** osztály ***println*** *eljárását* használtuk, erre hivatkoztunk.

Függvény: szintén utasítások összessége, melyet a nevére való hivatkozással hajtunk végre, de ***értéket ad vissza***, melyet a függvény neve képvisel. A visszatérési érték típusa a függvény ***visszatérési típusával*** egyezik meg.

5.2. Paraméterátadás

A ***szubrutinok*** a paramétereken keresztül képesek kommunikálni az őt meghívóval. Egy szubrutin nem láthatja az őt hívó szubrutin változóit, hiszen azok hatásköre csak saját utasításblokkjukra korlátozódik (vagy olyan globális változókkal, amelyek mindkettőjük számára látható).



31. ábra – Paraméterek fajtái

Az eljárások, függvények *fejlécében felsorolt paramétereket* **FORMÁLIS paramétereknek** nevezzük. Azokat a paramétereket pedig, amelyekkel az eljárást, vagy függvényt *meghívjuk*, **AKTUÁLIS paramétereknek** nevezzük. A formális és aktuális paraméterek darabszámának egyeznie kell, és páronként típusának kompatibilisnek kell lennie.

5.2.1 A paraméterátadás fajtái

Legtöbb programozási nyelvben (C, C++, Java stb.) csak az úgynevezett értékszerinti paraméterátadás létezik, míg más nyelvekben (pl. pascal, visual basic) létezik címszerinti paraméterátadás is.

- **Értékszerinti paraméterátadás**: az aktuális paraméterek tetszőleges kifejezések lehetnek és értékeiket rendre (első az elsőnek, második a másodiknak) *átadják a formális paramétereknek*. A hívott függvény nem képes megváltoztatni az aktuális paraméterek értékeit, hiszen számára azok nem láthatóak. Megoldható viszont, hogy a hívott megváltoztasson mégis egy hívó-beli értéket (azaz eredményt adjon vissza) *a mutatók vagy a referenciák használatával*. (A **mutató** egy olyan változó, amely egy memóriacímet képes tárolni, valamint képes az adott címen lévő adatot manipulálni.) Ha a formális paraméter egy mutató és a hívó ennek a mutatónak, illetve annak egy változójának a címét adja át, akkor ***a hívott képes megváltoztatni*** a hívó egy változójának értékét. Ha a formális paraméter egy *referenciaváltozó*, akkor képes felvenni egy változó referenciáját, azaz minden műveletben, amelyben szerepel, ***maga helyett a hivatkozott*** (akinek a referenciáját felvette) ***változó*** fog szerepelni, azaz úgy fog viselkedni, mintha a hivatkozott változónak egy másik neve lenne.
- **Címszerinti paraméterátadás**: Egyes nyelvek (pl. pascal, visual basic) ismerik a címszerinti paraméterátadást is. Ennél a paraméterátadásnál a *formális paraméter számára ugyanaz a*

memória terület lesz kijelölve, mint amelyet az párjaként szereplő *aktuális paraméter* használ. Így bármilyen műveletet végzünk a formális paraméterrel, az *kihatással van az aktuális paraméterre*, azaz ha az előállított eredményt egy cím szerint átadott paraméterbe pakoljuk, akkor ***azt a hívó is megkapja*** az aktuális paraméteren keresztül. A szubrutin lefutásakor a cím szerint átadott formális paraméter területe nem szabadul fel, *csak az azonosító* szűnik meg.

5.3. Példány- és osztálymetódusokok

Egy osztály metódusának a végrehajtását úgy kérhetjük, hogy *a metódus* ***nevére*** *hivatkozunk*. A metódus neve után a „( )” karakterpár szerepel, benne az esetleges paraméterekkel.

```
class TFoiskola
{
    public static void FelevMegnyitasa( int tanev, int felev)
    {
      Console.WriteLine("A foiskolan a {0}. év {1}. tanévét megnyitom.",
                        tanev,felev);
      OrarendekTorlese();
      IndexekLezarasa();
      GazdasagiEsemenyekNullazasa();
    }
}
```

A metódust meghivása az osztálynév.metódusnév formában történhet:

```
public static void Main()
{
 TFoiskola.FelevMegnyitasa( 2006, 2 );
}
```

32. ábra – Metódushívás

A metódus neve előtt megadható egy osztály vagy egy objektum neve, attól függően, hogy mit akarunk megszólítani:

- **Osztálymetódus** hívása: `Osztály.metódus(paraméterek)`
- **Példánymetódus** hívása: `objektum.metódus(paraméterek)`

Ha a metódust a ***saját*** osztályából hívjuk, akkor nem kell minősíteni, ***csak a metódus nevét*** kell leírni: `metódus(paraméterek)`.

Osztálymetódus: Vannak adatok, amelyek nem egy konkrét példányra, hanem az egész osztályra jellemzőek, ezek az *osztályadatok*, *osztályváltozók*. Tehát az osztályváltozó értéke az osztály összes példányára ugyanaz, vagyis *felesleges példányonként eltárolni*. Az osztálymetódus az osztályváltozókat manipulálja, a példányokat nem éri el, tehát objektumok nélkül is tud dolgozni Az osztály-szintű metódusok hasonlóan az osztály-szintű mezőkhöz static kulcsszóval vannak megjelölve. Az osztályszintű metódusok akkor is meghívhatóak, ha az adott, tartalmazó osztályból egyetlen példányt sem készítettek, ezért csak osztályszintű mezőkre, és konstansokra szabad hivatkozni.

Példánymetódus: Értelemszerűen *a példányváltozókon dolgozik*. Minden példányban megtalálhatjuk az osztály leírásában szereplő összes adatot, mivel azok példányonként más-más értéket vehetnek fel. Azonban metódusokat nem kell minden példányban tárolni, ezt elegendő egyszer az osztályban letárolni. A példányszintű metódusok meghívásához példányra van szükség, a metódushívás szintaktikája `példánynév.metódusnév`.

5.4. Eseménykezelő metódusok

Valamilyen **esemény** (Event) **hatására** (kattintás, görgetés, tabulátor stb.) bekövetkező metódusok.

Alacsony szintű esemény – az operációs rendszer szintjén történő elemi esemény, amely forrása csak valamilyen komponens lehet. (*Komponens-, Konténer-, Fókusz-, Ablak-, Billentyűzet-, Egér-esemény*). Pl. megváltozott a komponens mérete; fókuszba került a komponens; egy komponenst hozzáadtak a konténerhez; becsuktak egy ablakot; lenyomtak egy billentyű- vagy egérgombot.

Magas szintű esemény – ez általában logikai esemény (a progi állítja össze), forrása nem feltétlenül komponens. Magas szintő események lehetnek: ActionEvent – AbstractButton leszármazottja lehet a forrása; ItemEvent – kiválasztás esemény (1 tétel kiválasztása +változott);

AdjustmentEvent – igazítási esemény 1 gördítő-sávon; ListSelectionEvent – listakiválasztás esemény.

Pl. Java **JButton** esemény – lenyomtam 1 gombot:

```
public void actionPerformed(ActionEvent ev){
System.out.println(„lenyomtak egy gombot”);
}
```

Esemény – lenyomtam a gombot.

Eseményforrás – maga a gomb, az esemény itt keletkezik, előzetesen felfőztük az esemény figyelőre.

Eseményfigyelő – az eseményt figyelő és lekezelő objektum; az osztálynak implementálni kell az ezt figyelő interfészt – ez esetben az *ActionListener*-t. Az osztályban létezik az eseményt kezelő metódus (`actionPerformed(ActionEvent ev)`), a paraméter az eseményforrás azonosítója.

5.5. Függvények az SQL-ben

A különféle *SQL-megvalósulások sok függvényt* tartalmaznak. Az SQL általában kezeli a behívó nyelv függvényeit. A függvény-szerkezet:

```
függvénynév(argumentumok)
```

Van néhány függvény, amely alapértelmezés szerint ***be van építve*** az SQL-be, ezeket **beépített függvényeknek** nevezzük:

5.5.1 Összesítő (aggregáló) függvények

A lekérdezés eredményeként előálló táblák egyes oszlopaiban lévő értékeken végrehajthatunk bizonyos összesítő műveleteket, amelyek egyetlen értéket állítanak elő.

- **COUNT([DISTINCT] <kifejezés>)**: a kifejezés által meghatározott oszlopokban lévő rekordok száma. (DISTINCT esetén csak

a különböző értékeket számlálja, az azonosakból csak egyet vesz figyelembe)

- **SUM([DISTINCT] <aritmetikai kifejezés>)**: az aritmetikai kifejezés által meghatározott oszlopértékek összege.
- **AVG([DISTINCT] <aritmetikai kifejezés>)**: a kifejezés által meghatározott oszlopértékek átlaga.
- **MAX([DISTINCT] <kifejezés>)**: a kifejezés által meghatározott oszlopokból a legnagyobb értékeket adja vissza.
- **MIN([DISTINCT] <kifejezés>)**: a kifejezés által meghatározott oszlopokból a legkisebb értékeket adja vissza.

5.5.2 Logikai típusú vagy predikátum-függvények

- **BETWEEN**: `kif1 BETWEEN kif2 AND kif3` – Igaz értéket vesz fel, ha: kif2 <= kif1 <= kif3 (Tagadható is: NOT BETWEEN)
- **IN**: `oszlopnév IN (értéklista)` – Igaz, ha az oszlop értéke ***eleme*** a listának (valamelyikkel megegyezik).
- **LIKE**: `oszlopnév LIKE érték` – Igaz, ha az oszlop értéke ***megegyezik*** a like utáni értékkel/***hasonlít*** a like utáni értékre (ugyanis *maszkolható*: `_`; `%`). A `%` jel bármilyen karakter 0 vagy nagyobb hosszúságú sorozatát, az `_` jel egyetlen bármilyen karaktert helyettesít.

6. A kifejezés fogalma. Kifejezések kiértékelése, a műveletek precedenciája. egy választott programozási nyelv aritmetikai, logikai és relációs műveletei. Kifejezések SQL-ben

6.1. Kifejezések kiértékelése, a műveletek precedenciája

Kifejezés: **Operandusokból** (konstans, változó) és **Operátorokból** (műveletekből) **áll**. A kifejezésben szerepelhet egy vagy több Operandus, bármelyik Operandus lehet maga is egy kifejezés. (a+5 → „a” és

„5” operandus, „+” operátor) A kifejezés állhat egyetlen operandusból is, és bármelyik operandus lehet egy újabb kifejezés

6.1.1 Kifejezések lehetnek

- **Aritmetikai**: számtani alapműveletek
- **Logikai**: logikai alapműveletek
- **Karakteres**: karaktereken, sztringeken végzett műveletek, nem minden proginyelv támogatja.

A kifejezések kiértékelési **sorrendjét *a zárójelek és az Operátorok*** határozzák meg a következő szabályok szerint:

- Elsőként ***a zárójelben található*** kifejezések értékelődnek ki,
- Ezen belül ***előbb mindig a magasabb*** precedencia-szintű művelet hajtódik végre,
- Az azonos precedencia-szintű operátorok között ***a leírás sorrendisége dönt***, akkor a művelet ***asszociativitásától*** (csoportosítási, kiérrtékelési irányától) függően jobbról balra vagy balról jobbra (→) történik a kiértékelés.

Operátor	Asszociativitás
() []	Balról jobbra
! - (előjelváltás) ++ -- ~	Jobbról balra
& | ^ << >>	Balról jobbra
* / %	Balról jobbra
+ -	Balról jobbra
< <= > >= == !=	Balról jobbra
||	Balról jobbra
&&	Balról jobbra
= += -= *= /= %= >>= <<= %= != ^=	Jobbról balra
, (vessző)	Balról jobbra

33. ábra – Műveletek sorrendje, asszociációja

6.2. Választott programozási nyelv aritmetikai, logikai és relációs műveletei (C#)

A C#-ban az alábbi műveletek elvégzése támogatott.

6.2.1 Aritmetikai (számtani) műveletek

Művelet	Leírás	Példa
	A=10; B=20	
+	Két változó összeadása	A + B = 30
-	Kivonja a második változót az elsőből	A - B = -10
*	Változók szorzása	A * B = 200
/	Változók osztása	B / A = 2
%	Maradékos osztás („bennfoglalás")	B % A = 0
++	1-gyel növeli a változót	A++ = 11
- -	1-gyel csökkenti a változót	A-- = 9

6.2.2 Relációs (viszonyítási) műveletek

Művelet	Leírás	Példa
	A=10; B=20	
=	Ellenőrzi, hogy a 2 operandus (változó, tag) *egyenlő*-e; ha ***igen***, akkor a kifejezés ***igaz***.	(A == B) → HAMIS
!=	Ellenőrzi, hogy a 2 operandus egyenlő-e; ha ***nem***, akkor a kifejezés ***igaz***.	(A != B) → IGAZ
>	Ellenőrzi, hogy a *bal* oldali operandus **nagyobb**-e, mint a *jobb* oldali. Ha igen, a kifejezés ***igaz***.	(A > B) → HAMIS
<	Ellenőrzi, hogy a *bal* oldali operandus **kisebb**-e, mint a *jobb* oldali. Ha igen, a kifejezés ***igaz***.	(A < B) → IGAZ
>=	Ellenőrzi, hogy a *bal* oldali operandus **nagyobb vagy-egyenlő**-e, mint a *jobb* oldali. Ha igen, a kifejezés ***igaz***.	(A >= B) → HAMIS
<=	Ellenőrzi, hogy a *bal* oldali operandus **kisebb-vagy-egyenlő**-e, mint a *jobb* oldali. Ha igen, a kifejezés ***igaz***.	(A <= B) → IGAZ

6.2.3 Logikai műveletek

Művelet	Leírás	Példa
	A→Igaz; B→Hamis	
&&	**Logikai-ÉS-művelet** a neve; ha ***egyik*** operandus (logikai értéke) ***sem*** **HAMIS**, a kifejezés **igaz**.	(A && B) → HAMIS
||	**Logikai-VAGY-művelet** a neve; ha akárcsak az **egyik** operandus **IGAZ**, a teljes kifejezés **igaz**.	(A || B) → IGAZ
!	**Negáció**, ellentett-művelet; Egy kifejezés vagy operandus ***logikai értékének*** az ellentettjét adja (***megfordítja***).	!(A && B) → IGAZ

6.2.4 Bitenkénti műveletek

A bitenkénti műveleteket bitekre (pontosabban egész-számok bitjeire) tudjuk végrehajtatni (ki gondolta volna…?);

p	q	p & q	p | q	p ^ q
0	0	0	0	0
0	1	0	1	1
1	1	1	1	0
1	0	0	1	1

$A = 0011\ 1100_2 = 60_{10}$

$B = 0000\ 1101_2 = 13_{10}$

- A&B = 0000 1100
- A|B = 0011 1101
- A^B = 0011 0001
- ~A = 1100 0011

Művelet	Leírás	Példa
	A=60; B=13	
&	**Bitenkénti ÉS**-művelet – ha ***azonos helyiértéken ugyanaz*** az érték szerepel, akkor ***az jelenik meg*** a végeredményben	(A & B) = 12, vagyis 0000 1100

Művelet	Leírás	Példa
|	**Bitenkénti VAGY**-művelet – ha ***bármelyik*** operandusban ***1-es*** (bit) jelenik meg, az szerepelni fog a végeredményben	(A | B) = 61, vagyis 0011 1101
^	**Bitenkénti XOR**-művelet (kizáró-vagy) – csak ***akkor lesz 1-es*** a végeredményben, ha az ***csak az egyik*** operandusban (*műveleti tagban*, változóban) szerepelt; ha mindkettőben jelen van, akkor helyette 0 lesz…	(A ^ B) = 49, vagyis 0011 0001
~	**Bitenkénti egyes-komplemens**-művelet – egyváltozós művelet, az operandus bitjei invertálódnak, **ellentettjükre** változnak.	(~A) = -61, 1100 0011, az „A” kettes-komplemense
<<	Bitenkénti **balra-eltolás**-művelet – A bal oldali operandus ***1-esei*** a ***jobb*** oldali operandusba írt ***szám*** *szerint* ***lépkednek balra***…	A << 2 = 240, vagyis 1111 0000
>>	Bitenkénti **jobbra-eltolás**-művelet – A bal oldali operandus ***1-esei*** a ***jobb*** oldali operandusba írt ***szám*** *szerint* ***lépkednek jobbra***…	A >> 2 = 15, vagyis 0000 1111

6.2.5 Értékmegadási műveletek

Művelet	Leírás	Példa
=	***Egyszerű értékmegadás***, a bal oldalon a jobb oldali operandusok *összesített értéke* szerepel.	C = A + B
+=	C operandus ***megnövelése*** A-val	C += A, azaz C = C + A
-=	C operandus ***csökkentése*** A-val	C -= A, azaz C = C - A
*=	C operandus ***megszorzása*** A-val	C *= A, azaz C = C * A
/=	C operandus ***osztása*** A-val	C /= A, azaz C = C / A
%=	C operandus ***maradékos osztása*** A-val	C %= A, azaz C = C % A
<<=	C operandus 1-es bitjeinek ***eltolása balra*** 2-vel	C <<= 2, azaz C = C << 2
>>=	C operandus 1-es bitjeinek ***eltolása jobbra*** 2-vel	C >>= 2, azaz C = C >> 2
&=	A C és a „2” értékű változók ***bitenkénti ÉS***-művelete	C &= 2, azaz C = C & 2
^=	A C és a „2” értékű változók ***bitenkénti XOR***-művelete	C ^= 2, azaz C = C ^ 2
|=	A C és a „2” értékű változók ***bitenkénti VAGY***-művelete	C |= 2, azaz C = C | 2

6.2.6 Műveleti sorrend

Az „elsőbbségi rend” fentről lefelé halad.

Kategória	Műveletek	Asszociativitás
Postfix	() [] -> . ++ - -	Balról jobbra (→)
Egyváltozósak	+ - ! ~ ++ - -	Jobbról balra (←)
Sokszorosítás	* / %	Balról jobbra (→)
Összegzés	+ -	Balról jobbra (→)
Eltolásos	<< >>	Balról jobbra (→)
Relációs	< <= > >=	Balról jobbra (→)
Egyenlőség	== !=	Balról jobbra (→)
Bitenkénti ÉS	&	Balról jobbra (→)
Bitenkénti XOR	^	Balról jobbra (→)
Bitenkénti VAGY	|	Balról jobbra (→)
Logikai ÉS	&&	Balról jobbra (→)
Logikai VAGY	||	Balról jobbra (→)
Feltételes	?:	Jobbról balra (←)
Értékadás	= += -= *= /= %=>>= <<= &= ^= |=	Jobbról balra (←)
Vessző	,	Balról jobbra (→)

6.3. Kifejezések SQL-ben

Az SQL a kifejezések szerkezete és tartalma tekintetében megegyezik más nyelvekkel.

- **Aritmetikai kifejezések**: Numerikus vagy dátum-típusú oszlopnevekből, változókból, konstansokból, műveleti jelekből (+, -, *, /, **) és zárójelekből állnak. Szerepelhet bennük aritmetikai függvény is.
- **Karakter-kifejezések**: Karakter-típusú oszlopnevekből, változókból, szöveg konstansokból, műveleti jelből (a „+” a konkatenáció jele, pl: „ab”+”lak”=”ablak”) és zárójelekből állnak. A szöveg-konstansokat idézőjelek vagy aposztrófok közé tesszük.

❖ **Logikai kifejezések**: Logikai típusú oszlopnevekből, változókból, konstansokból, műveleti jelekből (AND/OR/NOT) és zárójelből áll. A logikai kifejezésekben szerepelhetnek a relációs operátorokok (< , > , =…).

7. Programozási tételek I. Elemi programozási tételek: sorozatszámitás, kiválasztás, eldöntés, lineáris keresés, megszámolás, maximum-kiválasztás (adatszerkezet nélkül, tömbbel, kollekciókkal, állományokkal)

A programozási tételek gyakran használt algoritmusokat takarnak. Ezeket az algoritmusokat általában tömbökön hajtjuk végre. Az adatok persze jöhetnek billentyűzetről, fájlból vagy adatbázisból is.

Az itt található feladatok `t[ ]` tömbön kerülnek végrehajtásra, a `t[ ]` tömbnek pedig *n* darab eleme van. Ahol több tömbbel dolgozunk, ott az első tömb `a[ ]`, amelynek *n* eleme van, a második tömb `b[ ]`, amelynek *m* eleme van. Harmadik tömb neve például `c[ ]`. Az `a[ ]` tömb ***ciklusváltozója*** szokásosan `i`, a `b[ ]` tömbé `j`, a harmadik `c[ ]` tömbé `k`.

A tömbök indexelése 0-val kezdődik. Az *értékadást* egy darab egyenlőségjel (=) jelenti, az *egyenlőségvizsgálatot* két egyenlőségjel (==) jelzi, a *nem-egyenlőt* egy kisebb-mint- és egy nagyobb-mint-jel jelzi (<>).

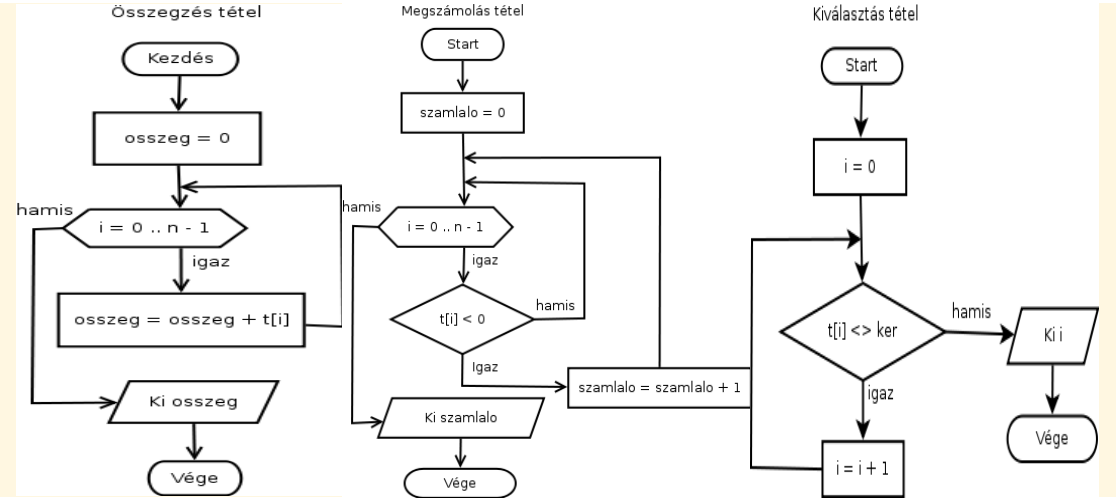
További infó: SZIT.hu

7.1. Sorozatszámítás (összegzés)

Az összegzés tétele **összeadja vagy összeszorozza** tömbben lévő számokat. Szorzásnál az S 1 kezdőértéket vesz fel.

```
osszeg = 0
ciklus i = 0 ... n -1
```

```
    osszeg = osszeg + t[i]
ciklus vége
ki osszeg
```



34. ábra – Néhány programozási tétel (Összegzés, Megszámolás, Kiválasztás)

7.2. A kiválasztás tétele

Ebben az esetben tudjuk, hogy létezik az adott tulajdonságú elem, a legelső ilyen elem sorszámát kapjuk eredményül. A kiválasztás-tételt akkor használjuk, ha tudjuk, hogy a keresett értéket tartalmazza a tömb. Ezért az nem vizsgáljuk, hogy vége van-e a tömbnek. A példában a `ker` változó tartalmazza a keresett értéket.

```
i = 0
ciklus, amíg tomb[i] <> ker
    i = i + 1
ciklus vége
ki i + 1
```

7.3. Az eldöntés tétele

Az eldöntés tétele megállapítja, hogy a tömbben van-e a feltételnek megfelelő elem.

```
van = 0
  ciklus i = 0 .. n-1
    ha tomb[i] = keresett_ertek akkor
        van = 1
    ha vége
  ciklus vége
```

A fenti megoldásnak az a hátránya, ha keresett értéket megtaláltuk, a ciklus *akkor is tovább megy*. Erre megoldást ad, ha a olyan ciklus állítunk munkába, amelyet akkor szoktunk használni, ha nem tudjuk meddig kell menni. Itt pedig ezzel van, dolgunk. Hogy hol lesz a keresett érték nem tudjuk, de ha meg van, le kell állni. Kell egy feltétel, hogy a ciklus addig menjen amíg nincs meg. Egy másik pedig, miszerint a ciklus addig menjen amíg van adat (nincs vége a tömbnek).A következő algoritmus megvalósítja ezt:

```
i = 0
ciklus amíg i<n és t[i]<> ker
    i=i+1
ciklus vége

Ha i<n akkor
    kiír "Van ilyen"
különben
    kiír "A keresett érték nem található"
ha vége
```

7.4. Lineáris keresés

A lineáris keresésben megállapítjuk, hogy létezik-e a tömbben az adott tulajdonságú elem, ha igen, megadjuk az elsőt. A tétel az *eldöntés*- és *kiválasztás*-tétel együttese.

```
ker = 30
i = 0
ciklus amíg i<n és t[i]<>ker
    i = i + 1
```

```
ciklus vége

Ha i<n akkor
    ki "Van ilyen"
    ki: "Indexe: ", i
különben
    ki: "A keresett érték nem található"
ha vége
```

7.5. A megszámolás tétele

Megszámoljuk, hogy hány adott tulajdonságú elem van a tömbben.

- X: A tömb.
- N: A tömb elemszáma, egész.
- DB: Az eredmény, egész típus.

```
DB:=0
     Ciklus I=1-től N-ig
               Ha T(X(I))
               akkor DB:=DB+1
     Ciklus vége
Eljárás vége...
```

7.6. Maximumkiválasztás

Megkeressük az adott tömbben a legnagyobb elemet. Eredményül a sorszámot és magát az értéket kaphatjuk. Hasonló feladat - csak a relációt kell megfordítani - a minimumkiválasztás.

```
max = t[0]
ciklus i = 1 .. n - 1
    ha t[i]> max akkor
        max = t[i]
    ha vége
ciklus vége
ki max
```

A tömböket csak a tárgyalás egyszerűsége miatt alkalmaztuk, azok lehetnek azonos típusú rekordokból álló fájlok vagy listák is.

Ennek megfelelően a programozási tételek megfogalmazhatók azonos hosszúságú rekordokból álló file-okra és listákra is. Ilyen esetekben az alábbi műveleteket kell kicserélni az algoritmusokban.

Tömb	File	Lista
Egy tömbelem vizsgálata	Egy rekord beolvasása egy változóba, majd a megfelelő mező vizsgálata	A listaelem beolvasása és a megfelelő mező vizsgálata
Iteráció egy ciklusban	Lépés a file következő rekordiára	Lépés a következő listaelemre
A tömb végének vizsgálata	End Of File vizsgálata	A listamutató 0-e vizsgálat
Egy tömb elem értékének az átírása	Ugrás az adott sorszámú rekordra, majd írás a rekordba, írás, után a rekordmutató! eggyel visszaállítjuk	A Listaelem kiírása
tömbindex	rekordsorszám	Egy számláló típusú érték, a listafej esetén 0 és minden iteráció esetén növekedik eggyel.

8. Programozási tételek II. Összetett programozási tételek: másolás, kiválogatás, szétválogatás, metszet, egyesítés, összefuttatás (tömbbel, kollekciókkal, halmazzal, állományokkal)

(A **7. tétel** bevezetőjében írtak itt is érvényesek valamennyire…)

8.1. Másolás

Egy sorozathoz egy másik sorozatot rendelő feladattípusokhoz. Adott egy N elemű sorozat. A feladat ezen sorozat lemásolása, s közben egy (elemre vonatkozó) átalakítást lehet végezni. Az eredmény mindig ugyanannyi elemszámú.

```
ciklus i = 1 .. n
    b[i] = művelet(a[i]) //valamilyen művelet a[i]-vel
ciklus vége
```

A példa kedvéért: Legyen például egy mondat, amelynek szavai szóközzel vannak tagolva. Szeretnénk a szóközöket alsóvonásra cserélni.

8.2. Kiválogatás

A tömb elemeit egy másik tömbbe rakjuk, valami feltételhez kötve. Például: Adott `a` és `b` tömb. Az a tömb egész számokat tartalmaz. Az `a` tömbből az 5-nél kisebb számokat átrakjuk `b` tömbbe.

```
j = 0
ciklus i = 0 .. n - 1
    ha a[i] < 5
        b[j] = a[i]
        j = j + 1
    ha vége
ciklus vége
```

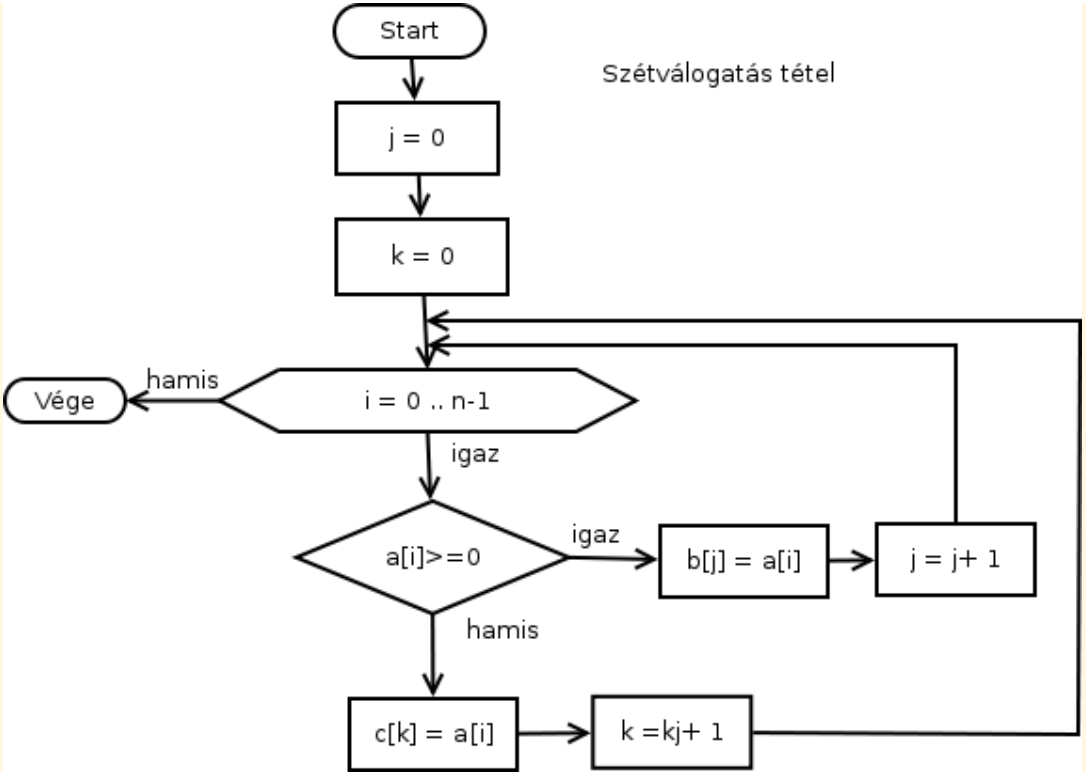
8.3. Szétválogatás

Egy sorozathoz két sorozatot rendelő feladattípusokhoz való. Rendelkezésünkre áll egy N elemű sorozat és egy, a sorozat elemein értelmezett T tulajdonság. Az a feladat, hogy a T tulajdonsággal rendelkező és a T tulajdonsággal nem rendelkező elemeket meghatározzuk egy-egy külön sorozatban.

Két tömbbe válogatjuk szét egy tömb elemeit.

```
a = {3, 2, 7, 1, 4, 8, 15, 17}
b = {}
c = {}
```

Adott „a” tömb, amely egész számokat tartalmaz. A „b” és „c” tömb pedig üres. Az „a” elemeit „b” tömbbe rakjuk, ha kisebbek 5-él, különben c-ben tároljuk.



35. ábra – A Szétválogatás tétele

```
j = 0
k = 0
ciklus i = 0 .. n-1
    ha a[i] < 5
        b[j] = a[i]
        j = j + 1
    különben
```

```
        c[k] = a[i]
        k = k + 1
    ha vége
ciklus vége
```

8.4. Metszet

Több sorozathoz egy sorozatot rendelő feladatokhoz. Rendelkezésünkre áll két sorozat. Az a feladat, hogy egy harmadik sorozatba azokat az elemeket válasszuk ki, amelyek mindkét sorozatban megtalálhatók.

Adott például A és B tömb:

- A `5, 3, 6, 2, 1`
- B `6, 2, 7, 8, 9`

A *közös elemeket* szeretnénk C tömbben:

- C `6, 2`

```
k = 0
ciklus i = 0 .. n-1
    j = 0
    ciklus amíg j<m és b[j]<>a[i]
        j = j + 1
    ciklus vége
    ha j<m akkor
        c[k] = a[i]
        k = k + 1
    ha vége
ciklus vége
```

8.5. Egyesítés (unió)

Több sorozathoz egy sorozatot rendelő feladatokhoz. Rendelkezésünkre áll két sorozat. Az a feladat, hogy egy harmadik sorozatba azokat az elemeket válasszuk ki, amelyek legalább az egyik sorozatban megtalálhatók.

A és B tömb *minden elemét szeretnénk C tömbbe* tenni.

Adott például A és B tömb:

- A `5, 3, 6, 2, 1`
- B `6, 2, 7, 8, 9`

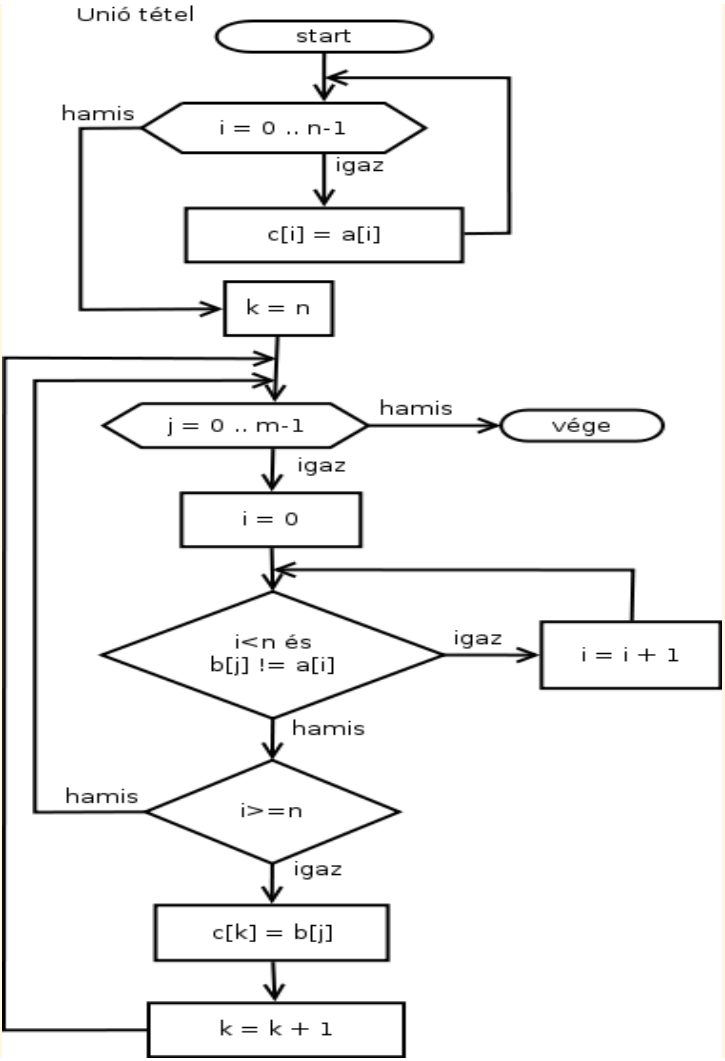
Az elemek *uniója* a C tömbben:

- C `5, 3, 6, 2, 1, 7, 8, 9`

Először C-be tesszük az A összes elemét, majd, ami nem szerepel A-ban, azt beletesszük C-be.

```
ciklus i = 0 .. n-1
    c[i] = a[i]
ciklus vége
k = n
ciklus j = 0 .. m-1
    i = 0
    ciklus amíg i<n és b[j]<>a[i]
        i = i + 1
    ciklus vége
    ha i>=n akkor
        c[k] = b[j]
        k = k + 1
    ha vége
ciklus vége
```

(Kép is van ám...:)



36. ábra – Unió tétele

8.6. Összefuttatás tétele

Adott két rendezett sorozat, képezzük a sorozatok egyesítését! A feladat megoldását a közönséges *Unió* feladatához képest most a sorozatok ***rendezettsége*** egyszerűsíti. A megoldás során sokkal kevesebb

lépést kell tennünk, ha ezt a tulajdonságot kihasználjuk. A rendezettségből adódik, hogy mindig el tudjuk dönteni, hogy melyik sorozatból kell vennünk a következő elemet.

```
i := 0
j := 0
k := -1
  ciklus amíg (i< n) és (j<m)
      k := k + 1

      ha a[i] < b[j] akkor
          c[k] := a[i]
          i := i + 1
      ellenben
      ha a[i] = b[j] akkor
          c[k] := a[i]
          i := i + 1
          j := j + 1
      ellenben
      ha a[i]> b[j] akkor
          c[k] := b[j]
          j := j + 1
      ha vége
  ciklus vége

  ciklus amíg i < n
      k := k + 1
      c[k] := a[i]
      i := i + 1
  ciklus vége

  ciklus amíg j < m
      k := k + 1
      c[k] := b[j]
      j := j + 1
  ciklus vége
```

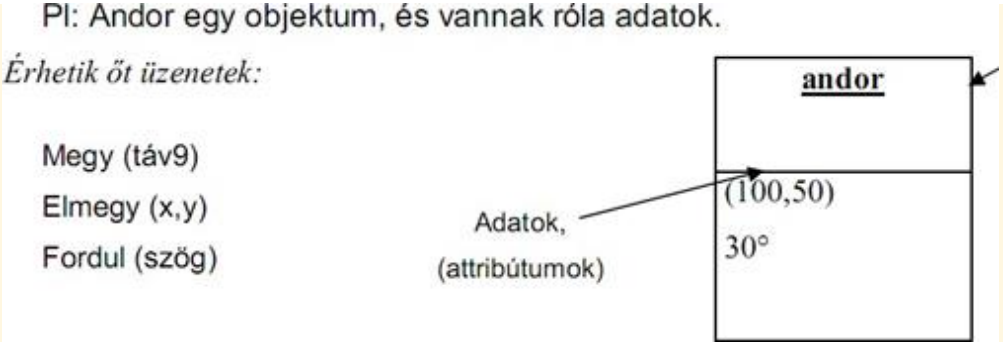
123. oldal (– „**B**” tételsor)

9. Osztály és objektum fogalma. Egységbezárás. Osztály definiálása egy választott fejlesztő környezetben. Jellemzők (properties). Az osztálymodell kapcsolata az adatbázis-modellel

9.1. Osztály és objektum fogalma

9.1.1 Objektum

A hétköznapi életben *az **objektum** egy tárgy, egy dolog*, nincs ez másképp az OOP-világban sem. Az objektumnak ***vannak adatai*** (tulajdonságai) és van valamilyen ***viselkedésmódja***. Az objektum információt tárol, kérésre feladatokat hajt végre. Az objektum felel feladatainak korrekt elvégzéséért. Az objektum ***logikailag összetartozó adatok és rajtuk dolgozó algoritmusok*** (rutin, metódus, progikód) *összessége*.



37. ábra – Objektum

Egy objektumorientált program ***egymással kommunikáló objektumok összessége***, melyben minden objektumnak megvan a jól meghatározott feladatköre. Pl. adott 1 család: Laci, Erzsi és 2 főzni tudó leány. Lacinak ebédre húslevest kell főznie.

```
„laci.elkészít(húsleves)”
```

Üzenetküldéskor ***a megszólított objektum és az üzenet*** neve közé '.'-ot (*pontot*) teszünk. Az üzenetnek lehetnek *paraméterei*. Az objektumokat üzeneteken keresztül kérhetjük meg különböző feladatok elvégzésére. Az üzenet nem más, mint egy, ***az objektumba beprogramozott rutin hívása***. Az OO-paradigmában *a rutint* ***metódusnak*** *nevezzük*. Egy objektumnak vannak **adatai és metódusai**:

- **Adatok**: az objektum az információt adatok és attribútumok formájában tárolja.
- **Metódusok**: a metódus ***olyan rutin, amely az objektum adatain dolgozik***. Az objektumokat a feladatokra *üzenetek* által lehet megkérni. Egy üzenet hatására végrehajtásra kerül az objektumnak egy, az üzenettel *azonos nevű* metódusa, s ezáltal ***az objektum adatai megváltoznak***.
- **Metódustúlterhelés**: Több, ***ugyanolyan nevű*** metódus is lehet, csak az a követelmény, hogy a paraméterlistájuk különbözzön:

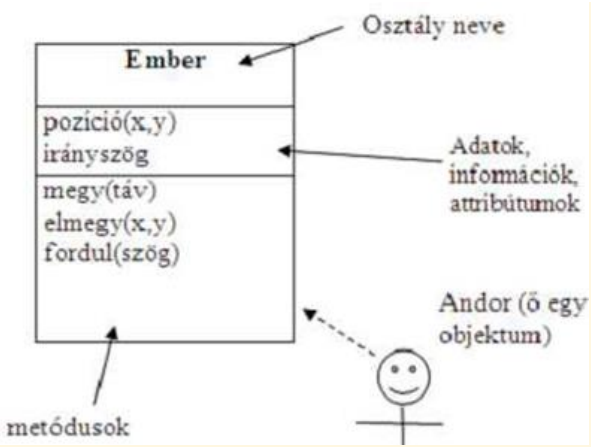
```
static double osszeg(double x, double y ){ }
static double osszeg(int x, double y ){ }
```

Az objektumnak mindig van valamilyen ***állapota*** (state) – ez megfelel az adatok *pillanatnyi* értékeinek. Egy feladat elvégzése után az objektum állapota megváltozhat. Az objektum *mindig emlékszik* az állapotára, tehát megjegyzi azt! Két azonos osztályhoz tartozó objektumnak akkor és ***csak akkor ugyanaz*** az állapota, ha az adatok ***értékei*** rendre megegyeznek. Az objektumok egyértelműen azonosíthatók. Az objektum azonossága független a tárolt értékektől.

9.1.2 Osztály

Az ***osztályozás*** a természetes emberi gondolkodás szerves része. Az ***ugyanolyan adatokat*** tartalmazó és ugyanolyan ***viselkedés-leírással jellemezhető objektumokat*** *egy osztályba* soroljuk.

Az osztály olyan objektumminta vagy típus, ***amelynek alapján példányokat*** (objektumokat) ***hozhatunk létre***. Minden objektum egy jól meghatározott osztályhoz tartozik. Minden objektum *egy osztály példánya*.



38. ábra – Osztály

Az osztályban definiáljuk:

- az objektum adatait (hogy az objektumok milyen adatokat jegyeznek meg),
- az objektum által elvégzendő műveleteket (metódusokat). A metódus tulajdonképpen rutin (eljárás, függvény), mely az adott objektum adatain dolgozik. Az üzenet pedig egy rutin hívása.

9.1.3 UML és jelölések

Az **UML**-t (Unified Modeling Language), az ***egységesített modellező nyelvet*** használjuk a valóság objektumorientált-paradigma szerinti lemodellezésére. Ehhez grafikus jelölésrendszer(eke)t használunk.

- Az **osztályt** egy 3 részre osztott téglalappal jelöljük:
  - a felső részbe írjuk az osztály nevét középre, kiemelten;
  - a középsőbe kerülnek az osztály adatai; az adat nevét és kezdőértékét így adjuk meg:

```
változónév:Típus=érték
```

- a legalsóba tesszük az osztályban szereplő metódusokat így:

```
metódusnév(paraméter:Típus,[…]):Típus
```

- a **példányt** (objektumot)szintén téglalappal jelöljük (ez kettéosztott):
- a példányba beírjuk annak azonosítóját aláhúzva (és esetleg osztályát kettősponttal elválasztva)
- szükség esetén megadhatjuk a második részben az objektum állapotát vagy bármit, amit fontosnak vélünk.
- A példányt egy szaggatott nyíllal az osztályhoz kötjük. A *nyíl* **mindig az osztály felé** irányul: ***a példány függ az osztálytól***.

Egy objektum születésekor *annak osztálya egyértelműen meg van határozva.* Az egyes objektumok üzenetek révén szólítják meg egymást. A feladatot kiadó objektumot kliensnek (***aktor***), a feladatot elvégzőt pedig szervernek (***agent***) hívjuk. Az objektumnak *életciklusa* van. (születés, élet, halál). Az objektumot létre kell hozni, majd azt követően **inicializálni** kell:

- be kell állítani kezdeti adatait
- végre kell hajtani azokat a tevékenységeket, amelyek az objektum működéséhez szükségesek. Az inicializálást végző metódust **konstruktornak** nevezzük. Pl. hozzuk létre az Ember osztályhoz tartozó Katit.

```
Kati = new Ember(100,50)
```

9.1.4 Példányváltozó, példánymetódus

Az objektum (példány) mindig a ***saját adatain*** dolgozik, a metódusokat pedig *az osztály leírásából* nézi ki. A (példányokhoz tartozó) metódusokat (ezek az osztályban vannak) elegendő ***csak egyszer az***

osztályban tárolni, azok majd *a megfelelő objektum adataitól* (állapotától) függően fognak működni. A *példányonként helyet foglaló változók a* ***példányváltozók*** (példányadatok). Az osztály azon metódusait, amelyek a *példányadatokon* (példányváltozókon) *dolgoznak*, ***példánymetódusoknak*** nevezzük (a példánymetódusok az osztályban helyezkednek el).

9.1.5 Osztályváltozó, osztálymetódus

Vannak adatok, amelyek *nem egy konkrét példányra*, hanem az ***egész osztályra*** jellemzőek. Az ilyen közös, *osztályonként egyszer előforduló változót* **osztályváltozónak** (osztályadatnak) nevezzük. Az osztályváltozó értéke az osztály összes példányára (objektumára) ***ugyanaz***, teljesen felesleges lenne ezt az értéket példányonként eltárolni. ***Osztálymetódusnak*** nevezzük az olyan metódust, amely *objektumok nélkül is* tud dolgozni. Az osztálymetódus a példányadatokat nem éri el, csak az ***osztályváltozókat manipulálja***. Az osztályváltozó, illetve osztálymetódus nevét az UML-ben aláhúzzuk!

9.2. Egységbezárás

Az objektum egyik legnagyobb ereje abban áll, hogy ***zárt és sérthetetlen***, azaz *az adatokat és a hozzájuk metódusokat összezárja* és a kliens számára *felesleges információkat elrejti*. Ez úgy oldható meg, hogy csak bizonyos, *a programozó által kijelölt metódusokat* – amelyeket ***interfésznek*** hívunk – használhatja a kliens, és az adatokat csak ezeken keresztül érheti el. A kliens így nem okozhat bajt, az objektumok állapotának egysége, logikája, konzisztenciája a kliens beavatkozása során is biztosított. Másképp: infókat rejtünk el a kliens elől, melyeket csak az interfészen keresztül lehet megközelíteni. Fontosabb szabályok:

- az objektumnak van egy **interfésze**, amely *a programozó által* ***kijelölt*** *metódusok összessége*.

- Az objektumot *csak az interfészen* keresztül lehet megközelíteni.
- Az adatok csak metódusokon keresztül érhetők el.
- Az objektum interfész-része a lehető legkisebb

A kliens üzen a szervernek – egy objektum felkérhet más objektumokat különböző feladatok elvégzésére (üzenetküldés, felkérés). Szerepkörök:

- *kliens* – a feladatot elvégeztető objektum. (ügyfél, kérő, üzenetküldő);
- *szerver* – a feladatot elvégző objektum (kiszolgáló, végrehajtó, választ adó, üzenetet fogadó)

9.2.1 Hozzáférési mód, láthatóság

- **Nyilvános** (+): minden vele kapcsolatban álló kliens eléri és használhatja,
- **Védett** (#): hozzáférés csak az osztályból és annak leszármazottaiból lehetséges,
- **Privát**: csak az osztály saját metódusai férhetnek hozzá.

9.3. Osztály definiálása egy választott fejlesztő környezetben (java)

Az Object osztály – a Javában minden osztály az **Objektum** osztályból származik. Mondhatjuk, bármely osztályból létrehozott példány az egy objektum, így azt minden olyan feladatra meg lehet kérni, mely már az objektumosztályban is definiálva van. Az objektumosztály olyan metódusokat tartalmaz, amelyek a Java minden objektumára jellemzőek. Nézzünk meg néhányat:

- *boolean equals(Object obj)*: összehasonlítja a megszólított objektumot a paraméterében megadott objektummal. A visszaadott érték true ha a 2 obj. egyenlő.
- *String toString()*: az obj. szöveges reprezentációját adja vissza.

- *Class getClass()*: visszaadja az objektum osztályát.

Egy osztály definiálása Javában egy fejből és egy blokkból áll. Az osztály fejét a **class** kulcsszó jelzi. Az osztály blokkja pedig az osztályra jellemző adatokat és metódusodeklarációkat foglalja magába.

Objektum létrehozása, deklarálása (példányosítás): Egy osztály példányait a ***new*** (új) operátorral hozhatjuk létre. A new után meg kell adni a létrehozandó objektum osztályát, ezt a konstruktor aktuális paraméterlistája követi:

```
new <OsztályAzonosító>(<aktuális paraméterlista>)
```

9.3.1 A new operátor feladatai:

- Létrehoz egy új OsztályAzonosító osztályú objektumot; lefoglalja számára a szükséges memóriát;
- Meghívja az osztálynak azt a konstruktorát, amelynek szignatúrájára ráhúzható az aktuális paraméterlista;
- Visszaadja az újdonsült objektum referenciáját (azonosítóját). Az objektum osztálya az lesz, amit a *new* után megadtunk.

A konstruktor beállítja az objektum kezdeti állapotát (kezdetei értéket ad az adatoknak és esetleges kapcsolatoknak). A konstruktor neve megegyezik az osztály nevével. Egy osztálynak több konstruktora is lehetséges. Az objektum egész élete során a *new* után megadott osztályhoz fog tartozni, ***osztályát megváltoztatni nem lehet***.

Minden referencia-típusú változót (mint ahogyan a primitív típusúakat is) deklarálni kell! Deklaráláskor csak a referencia részére következik be tárfoglalás: `<Osztályazonosító> objektum`;

Az obj. létrehozásáról a programozónak kell gondoskodnia. A *new* operátor által visszaadott referencia értékül adható a referencia-típusú változónak:

```
objektum = new <OsztályAzonosító>(<aktuális paraméterlista>);
```

vagy

```
<OsztályAzonosító> objektum = new <OsztályAzonosító> (<aktuális paraméterlista>);
```

Az objektumot a referencia-típusú változón keresztül szólíthatjuk meg: `objektum.metódus()`.

9.3.2 Konstruktorok

Amikor Egy objektumot a new operátorral létrehozunk, akkor azt ***inicializálni*** kell. A konstruktor beállítja az objektum kezdeti állapotát (kezdő-értéket ad az adatainak és kapcsolatainak, majd „felépíti"). A *konstruktor* ***neve*** *megegyezik az osztály nevével.* Egy osztálynak több konstruktora is lehet. Az objektum egész élete során a new után megadott osztályhoz fog tartozni, osztályát megváltoztatni nem lehet.

```
(<módosítók>) <OsztályAzonosító>((<formális paraméterlista>))

{
    <konstruktor blokkja>
}
```

39. ábra – Konstruktorblokk

9.3.3 A konstruktor általános szintakszisa

A konstruktor hasonlít a metódushoz – a következő szabályok érvényesek rá:

- A konstruktor neve kötelezően megegyezik az osztály nevével.
- Csak a ***new*** operátorral hívható. Egy konstruktorral nem lehet újrainicializálni egy objektumot.
- A módosítók közül csak a hozzáférési (láthatósági) módosítók használhatók.
- A konstruktor túlterhelhető
- A konstruktornak nincs visszatérési értéke, és nem *void*.
- A konstruktor nem öröklődik.

9.3.4 Alapértelmezés szerinti konstruktor

Ha egy osztályban nem adunk meg explicit módon konstruktort, akkor az osztálynak lesz egy alapértelmezés szerinti (default), paraméter nélküli konstruktora. Ha tehát az osztályban nem adtak meg konstruktort, akkor a példány létrehozásakor a rendszer ezt *az alapértelmezés szerinti konstruktort* hívja meg. Az objektum adatai ekkor az ***alapértelmezések*** szerint lesznek beállítva. Ha az osztályban létezik egy akármilyen *explicit* konstruktor (akár paraméteres, akár paraméter nélküli), akkor az osztálynak nem lesz ***implicit***, alapértelmezés szerinti konstruktora.

Konstruktor túlterhelése – A konstruktorok ugyanúgy túlterhelhetők, mint más metódusok. Egy objektum ***így többféleképpen is inicializálható***. Ha egy osztály több konstruktort definiál, akkor az egyik konstruktorból – annak első utsításaként – *meghívható egy másik konstruktor* a ***this*** ekkor eljárásként hajtódik végre:

```
this(<paraméterek>)
```

9.3.5 Jellemzők (Properties)

Az objektum beállítható [`setXY()`] és lekérdezhető [`getXY()`] tulajdonságát jellemzőnek nevezzünk(„*property*”).

9.4. Az osztálymodell kapcsolata az adatbázis-modellel.

Ha az osztálymodellünket jól terveztük meg, akkor a modellben lévő ***„entitás”-jellegű*** osztályok (entity) az adatbázismodellben ***egy-egy konkrét táblának*** felelnek majd meg. Ez az ún. ***objektum-relációs leképezés*** („OR mapping”) amelynek egyik mai tipikus képviselője a Javában lévő „EJB” technológia. Egy ilyen Java entitás-osztály főbb jellemzői:

- Implementálja a „Serializable” interfészt.

- Kötelező egy elsődlegeskulcs-attribútum („oszlop”, mező).
- A konkrét adattábla egy ***oszlopának*** (mezőjének) az entitás egy konkrét „***tulajdonsága***” (property) felel meg.

Az osztály és adatbázismodell kapcsolata pedig inkább UML-es megközelítést kíván. Ha készítünk egy alapvetően adatbázis használatára épülő rendszert, az osztályaink elnevezései és tulajdonságai nagy mértékben fognak hasonlítani a jól normalizált adatbázis logikai adatmodellére.

Persze e kettő nem lehet azonos. Adatbázisoknál nincs értelme többalakúságról vagy öröklésről beszélni, legalábbis adatbázis-modell jelölésrendszerében semmiképp. Viszont az ***egységbezárás*** és ***újrahasznosítás*** értelmezhető fogalmak.

9.4.1 Osztály-objektum kontra adattábla-rekord

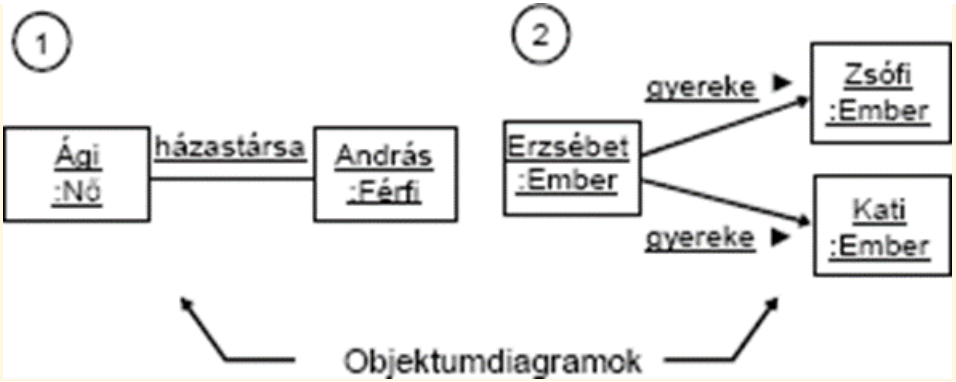
Tehát a *logikai adatmodell* és az *osztálydiagram* jelentős ***hasonlóságot*** mutat. Adattábláink mezőnevei többnyire osztályaink tulajdonságaival egyeznek meg.

10. Objektumok és osztályok közötti kapcsolatok. A kapcsolatok implementálása. Öröklődés, polimorfizmus, virtualitás

10.1. Objektumok és osztályok közötti kapcsolatok, kapcsolatok implementálása:

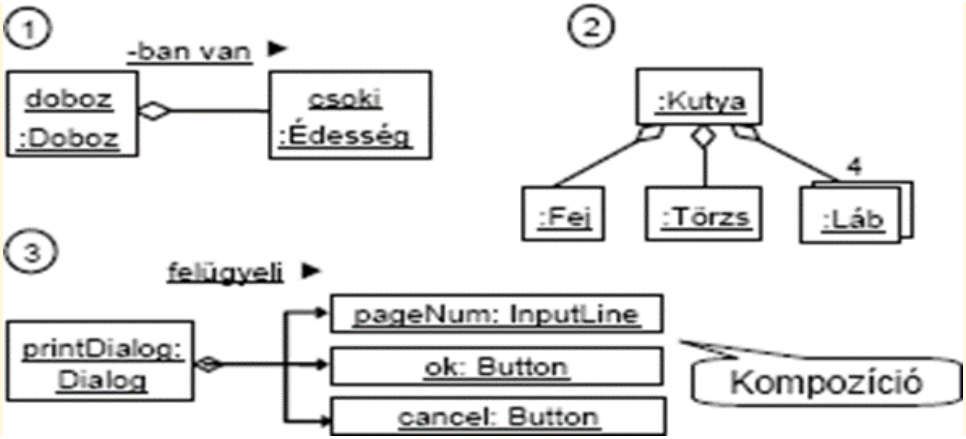
Az objektumok csak úgy tudnak együttműködni, ha azok társítási kapcsolatban állnak egymással. Ha a kliens-objektum üzenetet akar küldeni egy másik objektumnak, akkor tartalmaznia kell a megszólítani kívánt szerver objektumazonosítóját. **Kétféle társítási** kapcsolat létezik:

- **Ismeretségi kapcsolat** van két objektum között, ha egyik léte sem függ a másikétól és legalább az egyik ismeri, illetve használja a másikat. A kapcsolat nevét a vonal fölött aláhúzva írhatjuk, illetve a kapcsolat irányát a nyíl jelzi.



40. ábra – Ismeretségi kapcsolat

- **Tartalmazási kapcsolat** van két objektum között, ha az egyik objektum ***fizikailag tartalmazza vagy birtokolja*** a másik objektumot. A tartalmazó objektum az összetett vagy *egész* objektum, a benne levő pedig a ***részobjektum***. A részobjektum léte az egész objektumtól függ. Ha az egész objektumot megszüntetjük, vele együtt pusztul a rész is.
  - **Gyenge** tartalmazás, ha a rész kivehető az egészből (1),
  - **Erős** tartalmazás, ha a rész nem vehető ki az egészből
  - **Kompozíciónak** nevezzük azt a tartalmazást, amelyben *az egész objektumot* ***erős*** *tartalmazási kapcsolat köti* össze valamennyi részével. A teli rombusz (2) erős, az üres gyenge tartalmazás.



41. ábra – Tartalmazási kapcsolat

10.1.1 Objektumkapcsolatok

Az UML-ben úgy jelöljük a kapcsolatokat két objektum között, hogy az objektumokat összekötjük egy vonallal. A vonal végén lévő nyíl *a navigálás irányát* mutatja. Információs jelleggel a vonal fölé vagy alá írhatjuk *a kapcsolat nevét és irányát*. A nevet itt *aláhúzzuk*, mert objektumok közötti kapcsolatról van szó. Az objektumokat és azok kapcsolatait ábrázoló diagramot *objektumdiagramnak* nevezzük.

- **Függőség** (dependency): logikai kapcsolat. Az egyik (***független***) ***dolog változása*** maga után vonja a másik (***függő***) ***dolog változását***.
- **Általánosítás** (generalization) – öröklés: osztályszerű elemek közötti strukturális kapcsolat.
- **Megvalósítás** (realization): egy dolog megvalósít (realizál, implementál) egy másikat. Logikai kapcsolat, mely ***az általánosítás és a függőség*** keveréke. Csak ***osztályszerű*** elemek között lehetséges.

Osztályok között ugyanúgy értelmezünk ***ismeretségi***, illetve ***tartalmazási*** kapcsolatot, mint objektumok között. Egy feladat kapcsán is általában nem konkrét objektumokról beszélünk, hanem osztályok

közötti kapcsolatokról, hiszen a szabályokat legtöbbször általánosan kell meghatározni. Az osztályok közötti társítási kapcsolatot osztálydiagramon szokták ábrázolni és segítségével kifejezhető, hogy az egyik osztály egy-egy objektuma hány objektummal állhat kapcsolatba egy másik osztályból és milyen lehet az a kapcsolat.

42. ábra – Objektumkapcsolat egyik fajtája

- **Egy–egy-kapcsolat**: az egyik osztály egy példánya a másik osztály legfeljebb egy (0..1) példányával állhat kapcsolatban. A másik osztályra ugyanez vonatkozik. (Pl. Férfi és Nő házastársi viszonya).
- **Egy–sok-kapcsolat**: az egyik osztály egy példánya a másik osztály sok példányával állhat kapcsolatban, a másik osztály egy példánya viszont legfeljebb egy példánnyal állhat kapcsolatban az egyik osztályból. (Anya-Gyerek, Ország-Város).
- **Sok–sok-kapcsolat**: mindkét osztály akármelyik példánya a másik osztály sok példányával állhat kapcsolatban. (Pl. Tanfolyam-Hallgató, Hallgató- Hallgató (*)).
- ***Társítási-kapcsolat* megvalósítása**: a kliens (kérő) objektumnak ismernie kell a szerver (kiszolgáló) objektumot, különben nem tudja azt megszólítani. A program készítésekor a kliens objektumban felveszünk egy szerverre vonatkozó referenciát.
- ***Egy–egy-kapcsolat* megvalósítása**: az egyik osztályban felveszünk egy referencia tulajdonságot a másik osztályra.

- ***Egy–sok-kapcsolat* megvalósítása**: konténerobjektumokkal (pl. Vector) valósítjuk meg, de megvalósítható több referenciatulajdonság definiálásával is.
    - Tartalmazási kapcsolat esetén a kliens fizikailag tartalmazhatja a szervert, ekkor tehát nyugodtan felvehetünk a kliens osztályában egy (szerver) obj. típusú változót.
    - Ismeretségi kapcsolat esetén a kliens nem birtokolhatja a szervert, mert akkor azt más nem használhatná. Ekkor a kliensből 1 mutatót irányítunk a szerverre.
    - Ha 2 objekt 1:1 kapcsolatban áll egymással, akkor a kliens-objektumnak tartalmaznia kell 1 mutatót a szerverre, hogy megszólíthassa azt.
    - Az 1:N-kapcsolatok megvalósítására olyan konténereket használunk, amelyekbe szükség szerint akárhány obj. bedobható. Ahhoz, hogy a Kliens különböző karbantartási és keresési műveleteket hajtson végre a kapcsolódó objektumokon, valamilyen módon fizikailag hozzá kell kötözni őket. Kliens nem mutathat külön minden 1es szerverre, hiszen a szerverek száma elvileg végtelen lehet. Legyen a kliens objektumnak ***1 konténere*** (tárolója), amelybe elvileg akárhány szerver-objektum betehető. Amelyik szervernek a Kliens üzenni akar, annak *a mutatóját egyszerűen elkéri* a konténertől Az 1–sok-kapcsolat megvalósítása konténer-objektummal lehetséges.

10.2. Öröklődés, polimorfizmus, virtualitás.

10.2.1 Öröklődés

Ősosztálynak nevezzük, ***amiből örökítünk***, illetve leszármazott vagy *specializált osztály* a létrejövő utód. Az öröklődés hierarchiájának

mélysége tetszőleges (bár 10 felett már átláthatatlan), így beszélhetünk **alaposztályról** is, amely a hierarchia legfelső szintjén áll. Az osztály örökítésekor három lehetőségünk van:

- Új változókat adunk az ősosztályhoz,
- Új metódusokat adunk az ősosztályhoz,
- Az ősosztály metódusait átírjuk, azaz módosítjuk.

*Megjegyzés: az ősosztály **adatait** nem lehet átírni! Új adatok létrehozásával még nem érünk el semmit, szükséges új metódusokat is létrehozni.*

Szabályok:

- a hierarchia mélysége tetszőleges, de csak 10 alatti szám ajánlott,
- az öröklés tranzitív: ha A örökli B –t, és B örökli C –t, akkor ***A örökli C*** –t,
- példányadatok öröklése: az utód-osztály példányainak adatai = ***ősadatok+ saját***
- példánymetódusok öröklése: bármely metódust felül lehet írni.
- Az utódosztály az ősosztály ***kapcsolatait*** is örökli.
- ***Egyszeres öröklésről*** akkor beszélünk, ha egy osztálynak csak egy őse lehet.
- ***Többszörös öröklés***: egy osztálynak több őse is lehet.

Megjegyzés: a Java –ban a Pascalban és a Smalltalkban csak egyszeres öröklés van.

10.2.2 Láthatóság (hozzáférési mód)

Már volt szó arról, hogy az adatokat nem szabad kívülről manipulálni, és vannak olyan metódusok is, amelyeket a külső felhasználók elől el kell zárni. (lásd: *Bezárás*). A módszerek a következők:

- **Nyilvános** (public): minden kapcsolatban álló kliens eléri és használhatja. UML-jelölése: +

- **Védett** (protected): hozzáférés csak öröklésen keresztül lehetséges. UML-jelölése: #
- **Privát** (private): az osztály privát deklarációja, csak az osztály saját metódusai férhetnek hozzá. UML jelölése: -

Egy kész osztályt ***kétféleképpen lehet használni***:

- az osztályból ***példányt hozunk létre***: egy objektumnak kizárólag csak a publikus deklarációit lehet elérni. A privát és védett deklarációkat az objektum megszólításával nem lehet használni.
- az osztályból ***örökítéssel új osztályt*** hozunk létre: az örökítéskor az új utód osztályban felhasználjuk a már meglévő osztály adatait és metódusait. A privát deklarációkat csak az osztály programozója érheti el, ahhoz még öröklés révén sem lehet hozzáférni. A nyilvános és védett deklarációkat az utód osztály használhatja, hivatkozhat rájuk.

Szabályok:

- A láthatóságot *nem kötelező* megadni. A láthatóság alapértelmezése az ún. ***csomag-szintű láthatóság***, ez azt jelenti, hogy a deklaráció *az aktuális csomagban nyilvános*.
- Egy osztálynak is van láthatósága: a publikus osztály más csomagokból is látható; alapértelmezésként *egy osztály csak a saját csomagjában látható*.

10.2.3 Polimorfizmus (többalakúság)

Azt jelenti, hogy ***ugyanarra az üzenetre különböző objektumok különbözőképpen reagálhatnak***; minden objektum a saját, az üzenetnek megfelelő metódusával. (Az üzenet küldőjének nem kell tudnia a fogadó objektum osztályát).

10.2.4 Virtualitás

Virtuális metódus: Az objektumokkal történő munka során szükség lehet arra, hogy az utód osztály metódusait megváltoztassuk. Erre ad lehetőséget a virtuális metódusok használata. A virtuális metódusokkal átdefiniálható az ősosztály azonos nevű metódusa, így csak a futás közben dől el, hogy éppen melyik metódust kell használni.

Virtuális Metódusok Táblázata (VMT) – Minden egyes, virtuális metódusokat tartalmazó osztályhoz tartozik egy VMT. A virtuális metódusok címét a program futásakor ebből a táblázatból veszi. Az objektumpéldány egy VMT-mezőt tartalmaz, mely az osztály VMT-jének relatív címét tartalmazza (mérete 2 bájt). A példány VMT-hozzárendelését a konstruktor végzi a példány létrehozásakor, ill. inicializálásakor. Ha az osztály használ virtuális metódust, akkor van egy VMT-mezője, mely a virtuális metódus tábla címét tartalmazza.

Másképp:

Az ***@Override*** annotáció lényege, hogy egy ilyenformán jelölt metódusnak felül kell definiálnia egy azonos nevű és tulajdonságú metódust az ősosztályban. Pl.:

```
public class MyClass {

public MyClass() {
     }

@Override
public String toString() {
     return "Ez egy virtuális metódus";
     }

public static void main(String[] args) {
     new MyClass().toString();
     }
}
```

A fenti kódrészletben egy osztályban, annak ősosztályában (pl. Object) már meglévő metódust (`toString()`) írtunk felül. Ez a metódus az Object osztályban is pontosan így néz ki (visszatérési típus, név, paraméterek). A @Override annotáció elsősorban a fordítónak jelent nagy segítséget, amikor kinyomozza, hogy mely konkrét osztály metódusát is kell majd végrehajtani. (Jelen esetben ***nem az ősosztály*** (Object) `toString()`-je fog végrehajtódni, hanem ***az adott osztály*** (MyClass) `toString()`-je).

11. Algoritmusvezérelt és eseményvezérelt programozás összehasonlítása (vezérlés elve, működési mód és felhasználóval való kommunikáció alapján)

A programvezérlési módszereket osztályozhatjuk aszerint, hogy a felhasználó hogyan avatkozik be a program menetébe. Egy program kétféle lehet aszerint, hogy ***lehet-e*** vele „beszélgetni”:

- **Kötegelt**: (batch) programnak az aktor (az ügyfél) megadhat indítási paramétereket, de *a futó program működésébe* már *nem szólhat bele*.
- **Interaktív**: (*párbeszédes*) programmal a felhasználó a program *futása közben* is kommunikálhat.

Egy interaktív program a ***párbeszéd módjától*** függően kétféle lehet:

- **Algoritmusvezérelt**: az interakciót a program irányítja. Az aktor csak válaszol a program által feltett kérdésekre. A felhasználó pontosan akkor és azt mondhatja, amikor és amit a program kérdez tőle.
- **Eseményvezérelt**: egy tökéletesen eseményvezérelt program reagál a felhasználó által feltett kérésekre, kérdésekre. Az aktor azt mondhatja, amit ő akar, és akkor, amikor akarja!

Az eseményvezéreltség együtt szokott járni a grafikus felhasználói felülettel, de ez nem szükségszerű követelmény.

Az **algoritmusvezérelt program**: a kommunikációt ***a program irányítja***. Beolvassa az adatokat, elvégzi a számításokat, majd közli az eredményeket és befejezi a működést. Csak akkor van lehetőség a vezérlésre, ha a program erre külön rákérdez, vagy újra elindítjuk. Az algoritmusvezérelt program futtatása egy folyamat: a ***beolvasás, feldolgozás, eredményközlés*** folyamata.

Az **eseményvezérelt program**: a kommunikációt a felhasználó irányítja, bármikor beavatkozhat a program futásába és közölheti a kívánságait a programmal, tetszőlegesen módosíthatja az adatokat. A program fogadja és feldolgozza a beavatkozásokat, végrehajtja az utasításokat. A program futása is a felhasználó döntésére fejeződik be.

Az eseményvezérelt programozás lényege, hogy a program ill. egyes részei, ágai nem szekvenciális és előre meghatározható sorrendben futnak le, hanem a vezérlés lefutását bizonyos külső ill. belső események határozzák meg. A program egésze nem más, mint ***események bekövetkezésére válaszul végrehajtott szubrutinok*** (ún. eseménykezelők) ***laza halmaza***, amelyek nagyrészt egymástól függetlenül dolgoznak és működtetik a program egészét. Az eseményvezérelt modellben külső események alatt tipikusan a felhasználói bevitelt (billentyű lenyomása, egérművelet), valamint a hardvertől származó eseményeket (pl. az időzítőáramkor vagy perifériafia által kiváltott megszakítás, visszahívás) szokás érteni, míg a belső eseményeket a program más részeitől – vagy akár más programoktól – származó üzenetek képezik.

Eseményvezérelt program szinte bármilyen programozási nyelvben készíthető, de különösen könnyű és célszerű olyan nyelvet használni, amely támogatja az *objektumorientált programozást*, mert a ***többalakúság*** rendkívül egyszerűvé teszi az eseménykezelő architektúrák és hívási láncok kialakítását. Az eseményvezérelt programozás olyan

programozás, amely egy eseménybegyűjtő és szétosztó mechanizmuson alapszik. Az objektumok a hozzájuk rendelt eseményeket, eseménykezelő metódusok segítségével lekezelik. A metódusok ugyanolyan szabályok szerint kerülnek meghívásra, mint egy algoritmusvezérelt programban. Van azonban egy lényeges különbség: egy eseményvezérelt program fő szála mellett működik egy ***eseményelosztó szál***, benne egy eseményelosztó ciklus, amely folyamatosan figyeli, hogy bekövetkezett-e valamilyen esemény (*event*). Ha igen, akkor végignézi, hogy melyik komponenst (objektumot) illeti az esemény lekezelése – *billentyűzetlenyomás* esetén azt, amelyik éppen fókuszban van (aktív), *egéresemény* esetén azt, amelyiken rajta van az egér. Az eseményelosztó odaadja az esemény a „jogos" tulajdonosának, hogy az lekezelhesse. Ezt az eseménykezelő metódusok végzik, a programozó dolga az, hogy megírja ezeket a lekezelő metódusokat, és közölje a rendszerrel, hogy ezen az objektumon milyen eseményt figyeljen.

11.1. Egyéb csoportosítások

11.1.1 Vezérlés elve

- **Algoritmusvezérelt**: előre meghatározott sorrend szerint, szekvenciálisan;
- **Eseményvezérelt**: az események bármikor bekövetkezhetnek, nem mindre kell reagálni

11.1.2 Működési mód

Algoritmusvezérelt: a processzor jól meghatározott feladatokat hajt végre, nincs szükség beavatkozásra. Csak azok az interakciók lehetségesek, amiket a programozó lekódolt.

Eseményvezérelt: a program akár percekig "áll", az operációs rendszer által küldött üzenetekre bármikor válaszol. Az esemény mindenképpen bekövetkezik, legfeljebb nincs azt lekezelő rutin.

11.1.3 Felhasználóval való kommunikáció alapján

Algoritmusvezérelt: a felhasználó akkor kommunikálhat, ha azt a program megengedi

Eseményvezérelt: a felhasználó (szinte) bármikor bármilyen üzenetet küldhet

11.1.4 Példák

- **Algoritmusvezérelt**

```
print, polling, DMA-vezérlő, shell script
```

- **Eseményvezérelt**

```
IRQ, interrupt-masking, exception
```

12. Egy vizuális fejlesztő eszköz bemutatása: a fejlesztőkörnyezet elemei, szolgáltatásai, osztályhierarchia, látható és nem látható komponensek, adateléréshez kötődő komponensek

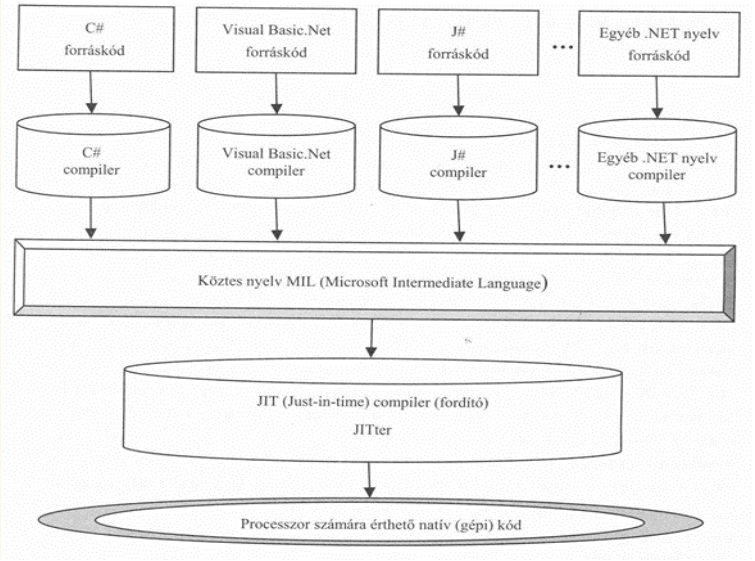
(Bizonyos részek elavulhattak...)

A Microsoft Visual Studio integrált fejlesztői környezettel a hozzá kapcsolódó .Net-keretrendszerrel *gyors alkalmazásfejlesztést* tesz lehetővé. Segítségével a következő programozási nyelvekkel tudunk alkalmazásokat készíteni:

Visual C#, Visual J#, Visual Basic, Visual C++ és lehetőség van szerver oldali programozásra ASP.NET-alapon.

A .NET egy alkalmazás keretrendszer, amiben egy csomó programrészlet beépítetten kész van. Így gyorsan lehet alkalmazásokat fejleszteni. Mivel sok minden gyárilag jól van megírva, a program eleve kevesebb hibalehetőséggel készül el. Ezen felül megvan a rendszer azon

előnye, hogy platformtól független. Hasonlóan a Java-hoz, a bináris fájlok egy virtuális, csak specifikáció-szinten létező gépre készülnek. De a program nem virtuális gépen fut, hanem ***indításkor fordítódik le*** a használt gép által értelmezhető kódra. Ennek köszönhetően egy alkalmazás futhat 32 bites rendszeren, 64 bitesen és akár Tableten is. Bizonyos körülmények teljesülése esetén Linuxon és OS-X-en is.



43. ábra – A .NET-keretrendszer felépítése

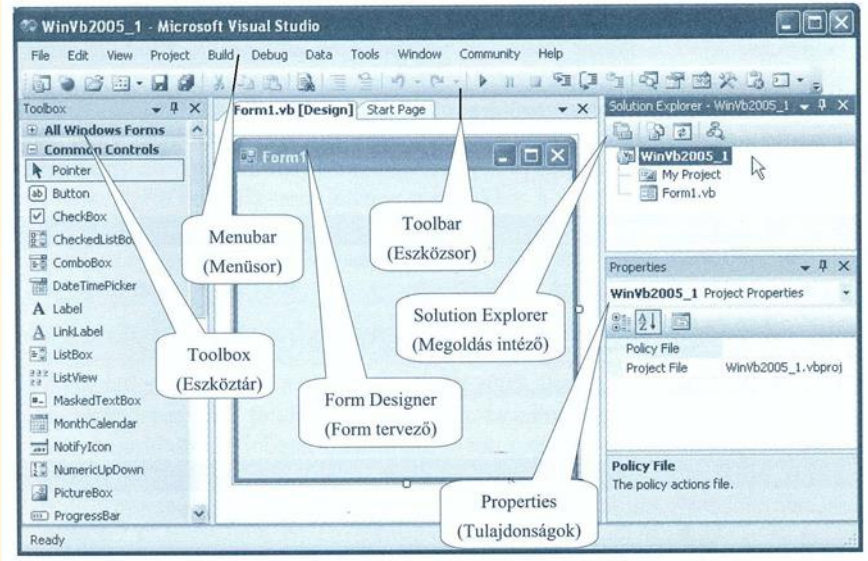
A .NET keretrendszer hivatott arra, hogy kapcsolatot teremtsen az alkalmazás és az operációs rendszer között. A CLR (Common Language Runtime) ***közös nyelvi futtatómodul*** felel a .NET-alkalmazások futtatásáért, de ehhez egy általa emészthető nyelvre van szükség. Minden .NET-nyelv rendelkezik egy saját ***fordítóval*** (*compiler*), amelynek bemenete az adott nyelv szintaktikájának megfelelő kód, a kimenete pedig már a közös nyelvi futtatómodul számára érthető *köztes* nyelv. Amikor a kiválasztott .NET nyelvű környezetben létrehozunk egy alkalmazást, annak kódját a nyelv fordítója a közös nyelvi futtatómodul számára már érthető köztes nyelvű ***MIL*** (*Microsoft Intermediate*

Language) kódra fordítja le. A közös nyelvi futtatómodul tulajdonképpen egy kiokosított virtuális gépnek is tekinthető, amely képes feltérképezni azt a számítógépes környezetet, amelyben fut, és annak megfelelően futtatja az alkalmazást. Ez a feltérképezés kiterjed mind a hardverkörnyezetre (pl. többprocesszoros alaplap), mind pedig a szoftverkörnyezetre (pl. DOS- vagy NT-alapú operációs rendszer). Mivel minden tekintetben a CLR felügyeli a kód és az általa elérhető erőforrások biztonságos felhasználását, ezért ebben a közös nyelvi futtatókörnyezetben futó kódot felügyelt kódnak hívjuk. Ezt a köztes nyelvű kódot már könnyen az egyes platformoknak leginkább megfelelő gépi kódra lehet fordítani. Ennek megfelelően a CLR tartalmaz egy futás előtti JIT (Just-in-time) fordítót (JITter), amely a processzor számára érthető gépi kódot hoz létre.

A .NET keretrendszer alatti alkalmazásfejlesztés valójában az osztályok kezelésén alapul. Habár az osztályok nagy részét készen kapjuk a keretrendszerrel, mégis előfordulhat, hogy nem találunk köztük megfelelőt. Ekkor lehetőségünk van más fejlesztők osztályait kölcsön venni, sőt mi is létrehozhatunk saját fejlesztésű osztályokat. Amikor valamilyen .NET-kompatibilis nyelven fejlesztünk, akkor csak ki kell keresni a szükséges funkciót ellátó osztályt, így csökkentve mind a fejlesztésre fordított időt, mind a programkódunk méretét. A .NET keretrendszer osztálykönyvtárának osztályait a köztük lévő eligazodás érdekében valamilyen hierarchikus struktúrába kellett szervezni. Ezt a jól meghatározott logika szerint felépített elrendezést névtér struktúrának hívják, amire tekinthetünk akár úgy is, mint az osztálykönyvtár tartalomjegyzékére. Az osztálykönyvtár ilyen módon való névterekbe szervezésével egyrészt elkerülhetők az elnevezésekből fakadó ütközések, másrészt a névterek közötti eligazodást is megkönnyíti. Ez a fajta csoportszervezés megengedi, hogy egy bizonyos osztálynevet többször is felhasználjunk, ha az nem ugyanabban a névtérben található.

12.1. A fejlesztőkörnyezet programja (VS)

A **Visual Studio** fejlesztőrendszerben minden olyan eszköz a rendelkezésünkre áll, amely a különböző típusú alkalmazások, illetve szolgáltatások létrehozásához szükséges. A tervezési, futtatási, hibakeresési és más funkcionális részek egyetlen helyről, az integrált fejlesztői környezetből IDE érhetők el. Az IDE a legtöbb windowsos fejlesztőrendszerhez hasonlóan rendelkezik menüsorral, eszközsorokkal, eszköztárral, grafikus tervezői felülettel, projektintézővel, tulajdonságok ablakkal stb.



44. ábra – Visual Studio régebbi változatának ablaka

Az eszköztár alapvetően a felhasználói felületek kialakításához használható controlok (pl.: button, Label) kollekcióját tartalmazza, amely a választott projekt (windowsos, webes, konzolalkalmazás) típusának, illetve az elvégzendő feladatnak megfelelően más-más képet mutat.

A **megoldás-intéző** ablakban az adott feladat megoldásához tartozó projekteket találjuk. Minden egyes projekt magában foglalhat több Formot, illetve több egyéb modult is, de egy Formot, vagy egy modult

minimálisan tartalmaznia kell. Az egyes projektek tartalmát a könnyebb áttekinthetőség érdekében fastruktúrába rendezve láthatjuk.

A **grafikusfelület-tervező** segítségével lehet az alkalmazás felhasználói interfészét elkészíteni. Windowsos alkalmazás esetén ez egy Form, amely futás közben egyablakként jelenik meg. A Form az alkalmazás elsődleges grafikus felhasználói felületeként jelenik meg, amely tartalmazhat controlokat, különféle grafikákat és képeket a tervező esztétikai érzékének és a megvalósítandó funkcióknak megfelelően.

A **tulajdonságok** ablak tartalmazza a kiválasztott objektumhoz rendelt tulajdonságoknak a listáját. Ezen tulajdonságoknak a készlete objektumfüggő, hiszen minden egyes objektumnak más és más a funkciója. A Tulajdonságok ablakot leggyakrabban a Formok, illetve a rajta elhelyezett controlok tulajdonságainak beállítására használjuk.

Az **Objektumböngésző** (Object Browser) dialógusablakban egy kiválasztott objektum minden jellemzőjét (események, tulajdonságok, tagfüggvények, stb) megnézhetjük.

Az események kezelését (interaktivitást) kódszinten kell megoldani, ami a Kódszerkesztőben (Code Editor) történik. Nemcsak az alkalmazáshoz felhasznált objektumok eseményeihez tartozó kód, hanem minden egyéb eljárás kódjának megírása itt történik.

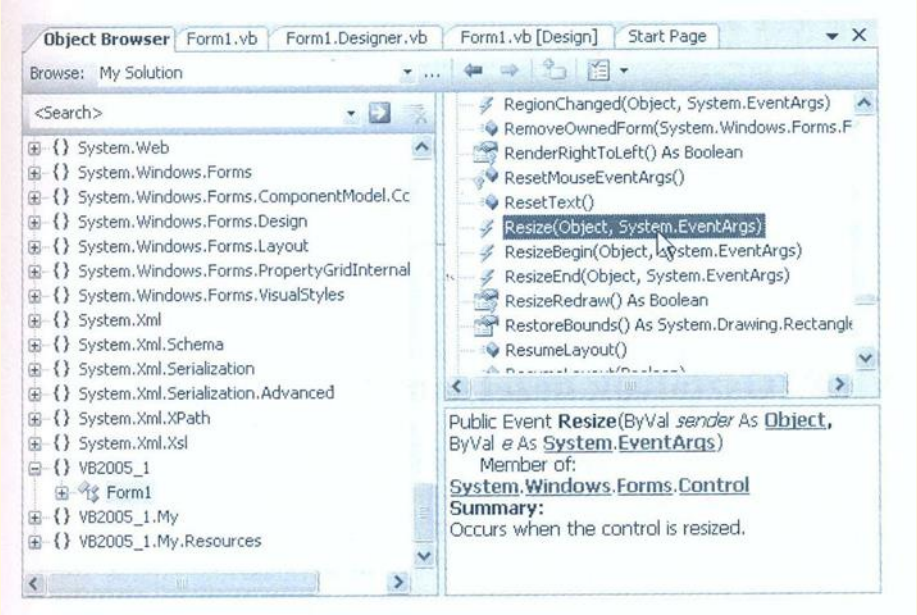
Az **Osztályok** nézet (Class View) ablakban a projektek osztályait és azok tagjait láthatjuk. Az itt kiválasztott eljárás sorára duplán kattintva, a Kódszerkesztő megfelelő helyére kerülünk.

A fejlesztőrendszernek a Professional Edition verziója a hagyományos súgó mellett egy Dinamikus súgó (Dynamic Help) ablakot is tartalmaz. Ebben az ablakban mindig a kiválasztott elemhez aktuálisan hozzárendelhető témakörök listája található.

(Megjegyzendő, hogy a(z) – többek közt a 2019-es verzió – *ingyenes* kiadásai csak magánszemélyként használhatóak, *vállalati környezetben nem*...)

12.2. A fejlesztőkörnyezet komponensei

A komponens egy olyan bináris (lefordított) szoftver-egység, amely jól definiáltan megvalósít egy funkciót. Azokat a komponenseket, melyeken keresztül a felhasználó kezelni tudja az alkalmazást, vezérlőknek (controls) nevezzük. Ilyenek a gomb, a menü, a szövegdoboz, kép stb. Léteznek olyan komponensek is, mint például az időzítő (Timer), melyek futtatás közben nem jelennek meg a képernyőn. Ezek nem tartoznak a vezérlők közé.



- ***látható*** (visible): A közönséges vezérlők (common controls), ilyen pl: a button, label, menü …)
- ***nem látható*** (logical): Pl.: Időzítő, EventLog, DirectoryEntry, Process, ...

Osztályhierarchia: A projekt osztálydiagramját megtekinthetjük, ha az adott projekten jobb egérgombbal kattintunk, és kiválasztjuk a „View Class Diagram” lehetőséget. A programnyelvhez adott osztályokat csoportosították, ezeket online és *offline* (feltelepített CD-n) kereshetővé, elérhetővé tették.

12.3. Adateléréshez kötődő komponensek

A .NET-alkalmazás keretrendszer része az adatbázis-elérés biztosító ADO.NET. Az ADO.NET komponensei segítségével elkülöníthető egymástól az adatokkal történő manipuláció, és azok megjelenítése. A komponensek ugyanis komplex objektumok, melyekkel elérhetjük a tetszőleges típusú adathalmazokat, elvégezhetjük az adatok módosítását, törlését, valamint visszakaphatjuk a lekérdezések eredményhalmazait. A kapott adathalmazokat ezt követően adatmegjelenítő objektumokkal tesszük láthatóvá. Egy adatbázistábla sorainak eléréséhez a DataGrid-vezérlőt használhatjuk, Az SqlConnection-objektumokkal SQL Server-adatbázisokhoz kapcsolódhatunk, míg az SqlDataAdapter-objektumokkal sorokat vihetünk át az SQL Server-adatbázis és egy DataSet-objektum között. A Query Builder-t használjuk az SQL-utasítások megadására, a lekérdezést beírhatjuk, de grafikusan is felépíthetjük őket.

13. Relációs adatbázisok. Funkcionális függőség fogalma, speciális függőségek szerepe. Normálformák, a normalizálás célja. A normalizálás lépéseinek szemléltetése példán. Az adatbázis-terv dokumentációja

13.1. Relációs adatbázisok

Relációs adatbázisok (oszlopokba szedett adatok összessége): A reláció az adatelemek megnevezett, összetartozó csoportjából kialakított olyan ***kétdimenziós táblázat***, amelyik *sorokból és oszlopokból áll*, és ahol az ***oszlopok*** (mezők) 1-1 ***tulajdonságot*** írtnak le, a ***sorok*** adják az egyedhalmaz/reláció/táblázat ***egyedeit***. Ahhoz, hogy egy táblázatot **relációnak** lehessen tekinteni, a következő feltételeket kell kielégítenie:

- nem lehet **két egyforma** sora,
- minden oszlopnak **egyedi neve** van,
- a sorok és oszlopok **sorrendje tetszőleges**.

A relációs adatbázisok általában nem egy, hanem *több, logikailag összekapcsolható relációból* (táblázatból) állnak. A reláció *oszlopainak* (attribútumainak) számát a ***reláció fokszámának***, *sorainak* számát a ***reláció kardinalitásának*** nevezik.

Kulcs - amennyiben egy tulajdonság vagy tulajdonságok egy csoportja ***egyértelműen meghatározza***, hogy az egyed ***melyik*** értékéről, ***előfordulásáról*** van szó, akkor *ezeket a tulajdonságokat* együtt *kulcsnak* nevezzük.

13.2. Funkcionális függőség fogalma, speciális függőségek szerepe.

A függőség fogalmának segítségével a táblázatok *belső szerkezetét* tárhatjuk fel. A *funkcionális* és *többértékű függőség* a táblák ***oszlopai között lévő összefüggéseket*** írja le.

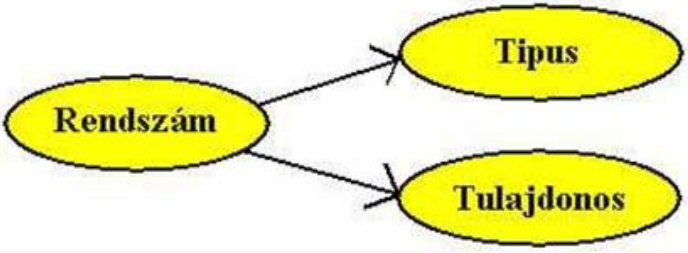
151. oldal (– „**B**” tételsor)

13.2.1 Funkcionális függőség

Adatok között akkor áll fenn funkcionális kapcsolat, ha egy vagy több adat konkrét értékéből más adatok egyértelműen következnek. Például a személyi szám és a név között funkcionális kapcsolat áll fenn, mivel minden embernek különböző személyi száma van. Ezt a SZEMÉLYI_SZÁM -> NÉV kifejezéssel jelöljük, vagy pedig egy diagrammal.

A funkcionális függőség bal oldalát a függőség meghatározójának nevezzük. Nem áll fenn funkcionális függőség akkor, ha a meghatározó egy értékét több attribútum értékkel hozhatjuk kapcsolatba. Például a `NÉV -> SZÜLETÉSI_ÉV` állítás nem igaz, mert több személynek lehet azonos neve, akik különböző időpontokban születtek.

A funkcionális függőség jobb oldalán több attribútum is állhat. Például az `AUTÓ_RENDSZÁM -> TIPUS, TULAJDONOS` funkcionális függőség azt fejezi ki, hogy az autó rendszámából következik a típusa és a tulajdonos neve, mivel minden autónak különböző a rendszáma, minden autónak egy tulajdonosa és típusa van. Ezt diagrammal is ábrázolhatjuk.



45. ábra – Függőségek az adattáblákban (több ábra...)

Az is előfordulhat, hogy két attribútum ***kölcsönösen függ*** egymástól. Ez a helyzet például a házastársak esetén:

152. oldal (– „**B**” tételsor)

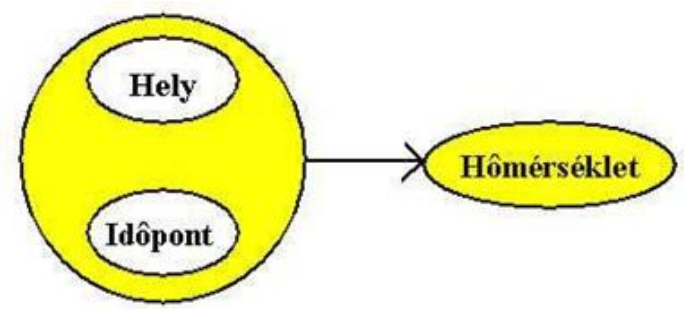
```
FÉRJ_SZEM_SZÁMA -> FELESÉG_SZEM_SZÁMA
FELESÉG_SZEM_SZÁMA <- FÉRJ_SZEM_SZÁMA
```

Mindkét funkcionális kapcsolat igaz és ezt a `FÉRJ_SZEM_SZÁMA <-> FELESÉG_SZEM_SZÁMA` jelöléssel fejezzük ki.

A funkcionális függőség bal oldalán ***több attribútum*** is megjelenhet, melyek **együttesen** határozzák meg a jobb oldalon szereplő attribútum értékét. Például hőmérsékletet mérünk különböző helyeken és időben úgy, hogy a helyszínek között azonosak is lehetnek. Ebben az esetben a következő funkcionális függőség áll fenn az attribútumok között:

```
HELY, IDŐPONT -> HŐMÉRSÉKLET
```

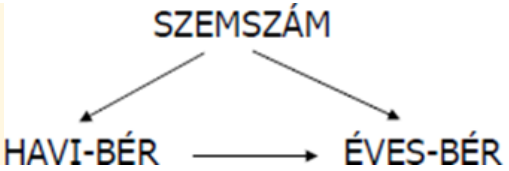
A fenti összefüggést az alábbi diagrammal is jelölhetjük:

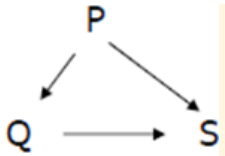


13.2.2 Speciális függőségek

Teljes funkcionális függőség: Erről akkor beszélhetünk, ha a ***meghatározó*** oldalon ***nincsen felesleges attribútum***. Például a `RENDSZÁM, TÍPUS -> SZÍN` funkcionális függőség nem teljes funkcionális függőség, mivel a rendszám már egyértelműen meghatározza a kocsi színét, ehhez *nincs szükség a típusra* is.

A **tranzitív függőség** gyakorlatilag azt jelenti, hogy két funkcionálisan függő attribútumhalmaz mellé található egy harmadik attribútumhalmaz a relációban, amely a két halmaz közötti függőséget átviszi.





A **funkcionális függőségek tulajdonságai**:

- **Reflexivitás** – Ha $Q \subseteq P \subseteq A$, akkor $P \rightarrow Q$ teljesül, azaz egy attribútum halmaz a benne levő részhalmazt meghatározza. egy tulajdonságtípus *bármely értéke* ***meghatározza önmagát***.
- **Additív** (egyesítési) szabály: $A \rightarrow B$, $A \rightarrow C \Rightarrow A \rightarrow (B+C)$ ha A tul. típus 1 B és 1 C tul. típust határomeg, úgy meghatározza a (B+C) tulajdonság típus sort is.
- Tranzitivitás: $A \rightarrow B$ és $B \rightarrow C \Rightarrow A \rightarrow C$, ha 1 A tulajdonság típus meghatároz egy B tulajdonság típust, ami C -t határozza meg, a C az A -tól is függ.
- Pszeudo-tranzitivitási szabály: $A \rightarrow B$ és $(B+x) \rightarrow C \Rightarrow (A+x) \rightarrow C$ ha egy A tulajdonság típus meghatároz 1 B -t, amely pedig bármilyen bővítményével (B+X) meghatároz 1 C -t, akkor az A megfelelő bővítménye (A+X) is meghatározza a C -t.
- Bővítés: Ha $P \rightarrow Q$ teljesül és $S \subseteq A$ egy tetszőleges attribútumhalmaz az A - ból, akkor $PUS \rightarrow QUS$
- Dekompozició: Ha $P \rightarrow Q$ teljesül és $S \subseteq Q$, akkor $P \rightarrow S$ is teljesül.

1. **Reflexivitás**
   Ha $Q \subseteq P \subseteq A$, akkor $P \rightarrow Q$
2. **Bővítés**
   Ha $P \rightarrow Q$ és $S \subseteq A$, akkor $P \cup S \rightarrow Q \cup S$
3. **Tranzitivitás**
   Ha $P \rightarrow Q$, $Q \rightarrow S$, akkor $P \rightarrow S$
4. **Egyesítési szabály**
   Ha $P \rightarrow Q$, $P \rightarrow S$, akkor $P \rightarrow Q \cup S$
5. **Pszeudotranzitivitási szabály**
   Ha $P \rightarrow Q, T \cup Q \rightarrow S$, akkor $P \cup T \rightarrow S$
6. **Dekompozíciós szabály**
   Ha $P \rightarrow Q$ és $S \subseteq Q$, akkor $P \rightarrow S$

46. ábra – Funkcionális függőségek tulajdonságai

13.3. Normálformák, a normalizálás célja. A normalizálás lépéseinek szemléltetése.

A normalizálás segítségével ***minimalizáljuk a tárolási helyet***, ***megszüntetjük*** *az adatok* ***többszörös tárolását***, a fölösleges redundanciát, továbbá a beírási, törlési és módosítási anomáliákat, amelyeket az okoz, hogy túl sok adatot tárolunk egy táblázatban (illetve az összes, duplán letárolt adat frissítése elhúzódhat...).

Normalizálás során ***induláskor egyetlen táblázatot alakítunk ki***, amelyben minden **szükséges** tulajdonságot elhelyezünk. Ezt követően feltárjuk a táblázat *belső szerkezetét*, azaz meghatározzuk az oszlopok *függőségi viszonyait*, majd megvizsgáljuk, hogy a feltárt függőségek ***eleget tesznek-e azoknak a követelményeknek***, amelyet *normálformának* nevezünk. A normálformákat *megsértő függőségeket* úgy szüntethetjük meg, hogy *a táblázatot* egy meghatározott szabály szerint, több lépésben **további táblázatokra bontjuk**. A normálformáknak való megfelelés ellenőrzését a szétbontással keletkezett új

táblázatoknál előről kezdjük. Napjainkban *hat normálformát* definiáltak, de a gyakorlatban elég az első **3 normálformával** dolgozni.

13.3.1 Első normálforma

Egy táblázat akkor van normálformában, ha **minden sorában pontosan egy attribútum érték áll**. Ha egy reláció nincs 1NF-ben, akkor kétféleképpen alakítható át ilyen típusúvá:

1. A több attribútumértéket tartalmazó sort *annyi sorra bontjuk*, *ahány* ***nem elemi attribútumérték*** szerepelt benne.
2. Az eredeti relációt ***több 1NF-es relációra*** bontjuk úgy, hogy az egyikben a reláció kulcsának értékei mellé írjuk az egyszeres attribútumértékeket, a másik relációban pedig a kulcshoz rendelt külső kulcs mellé annyi sort írunk, ahányszoros attribútumértékek szerepelnek a többszörös attribútumokban.

13.3.2 Második normálforma

Egy táblázat akkor és csak akkor van második normálformában, ha ***minden másodlagos attribútum teljesen függ a kulcstól***. (A másodlagos attribútumok ***nem függhetnek a kulcs részeitől***.)

A definícióból következik két egyszerű kritérium, mely megkönnyítheti a 2NF típusú relációk felismerését:

- Ha ***a kulcs egyetlen attribútumból áll*** (vagyis egyszerű), akkor a reláció 2NF típusú.
- Ha a relációban *nincsenek másodlagos attribútumok* (tehát ***minden attribútum része a kulcsnak***), akkor a reláció 2NF típusú.

Ha egy reláció nincs 2NF-ben, akkor alakítható át ilyen típusúvá:

Az 1NF-jú táblázatból olyan táblázatokat kell létrehoznunk, amelyekben ***a kulcs minimális***, azaz a kulcs-tulajdonságokból és a másodlagos tulajdonságokból ***önálló relációkat*** hozunk létre.

1. Kiemeljük a kulcsból azokat az attribútumokat (vagy attribútumhalmazokat), amelyek ***önállóan is*** *meghatározzák a másodlagos attribútumokat.*
2. Az 1. pontban kiválasztott *elsődleges* és az 1. pont szerint hozzájuk tartozó *másodlagos attribútumokból* ***egy relációt*** állítunk elő.
3. Azokat a másodlagos attribútumokat, amelyek ***csak a kulcstól függenek*** (a részeitől nem), tehát teljes függőségben vannak a kulccsal, a ***kulcsban szereplő elsődleges attribútumokkal együtt*** *egy táblába* fogjuk össze.

13.3.3 Harmadik normálforma

A reláció akkor és csak akkor van *harmadik normálformában*, ha ***2NF****-ben van*, és ***a másodlagos*** attribútumok között ***nincsen funkcionális*** *függőség.* Ha egy reláció nincs 3NF-ben, akkor alakítható át ilyen típusúvá:

Egy reláció 3NF-re hozható, ha ***megszüntetjük a tranzitív függőségeket*** úgy, hogy a tranzitív függőségben résztvevő attribútumhalmazok felhasználásával új relációkat készítünk.

13.3.4 Példa

Egy kórházban a betegeknek ***kartonjuk*** van. Ezen a kartonon szerepel a nevük, címük, más személyi adataik, továbbá betegségeik. A betegségükből következik, hogy *melyik osztályon fekszenek*, valamint az, hogy *milyen gyógyszereket* kapnak. Képzeljük el, hogy ezek a kartonok mind a kórház adminisztrációs irodájában vannak. Tegyük fel, hogy valakinek kimutatást kell készítenie arról, hogy egy bizonyos gyógyszert hány beteg szed. Az illetőnek az összes kartont végig kell lapoznia, a rengeteg adatból ki kell keresnie a gyógyszerek nevét. Ki kell választania a kartonok közül azoknak a kartonjait, akik az adott

gyógyszert szedik. Akár több napot is el lehetne ezzel a munkával tölteni. Bemutatásra kerül, hogy a *normalizáció eredményeként* kapott relációs modell segítségével ez mennyivel egyszerűbb feladat lesz.

Tegyük fel, hogy a kartonon a következő adatok szerepelnek a betegekről:

- Beteg azonosító (***B_azon***)
- Beteg neve (***B_név***)
- Beteg címe (***B_cím***)
- *Betegség*
- Osztály azonosító (***Oszt_az***)
- Osztály név (***Oszt_név***)
- *Főorvos*
- *Gyógyszer*

Képzeljünk el néhány rekord-előfordulást, ami segít az elsődleges kulcs kiválasztásában is (célszerű elképzelésünket táblázatba is foglalni, nehogy elkerülje figyelmünket valami lényeges összefüggés):

B_azon	*B_név*	*B_cím*	*Betegség*	*Oszt_az*	*Oszt_név*	*Főorvos*	*Gyógyszer*
444	Kiss Cili	Ajka	sérv	01	sebészet	Dr. Doktor	Algopyrin Semicillin
444	Kiss Cili	Ajka	tyúkszem	02	szemészet	Dr. Joó	Semicillin
333	Nagy Pál	Veszprém	tyúkszem	02	szemészet	Dr. Joó	Sumetrolim Demalgon

Ezzel a táblázattal még nem könnyítettünk sokat a munkánkon, mert az ismétlődések (tárolási redundancia) és az ebből származó anomáliák miatt ***nehéz karbantartani***. Ráadásul első normálformában (*1NF*) sincs, hiszen vannak benne ***többértékű*** mezők (*Betegség* és *Gyógyszer*), amit a relációs modell nem visel el. Továbbá hiányzik még az elsődleges kulcs is! A normalizálás *első lépése az 1NF* kialakítása. Mindenekelőtt ***a többértékű mezőket kell megszüntetni***. A sorok ismételt leírásával elérhetjük ezt a célt.

B_azon	B_név	B_cím	Betegség	Oszt_az	Oszt_név	Főorvos	Gyógyszer
444	Kiss Cili	Ajka	sérv	01	sebészet	Dr. Doktor	Algopyrin
444	Kiss Cili	Ajka	sérv	01	sebészet	Dr. Doktor	Semicillin
444	Kiss Cili	Ajka	tyúkszem	02	szemészet	Dr. Joó	Semicillin
333	Nagy Pál	Veszprém	tyúkszem	02	szemészet	Dr. Joó	Sumetrolim
333	Nagy Pál	Veszprém	tyúkszem	02	szemészet	Dr. Joó	Demalgon
444	Kiss Cili	Ajka	sérv	01	sebészet	Dr. Doktor	Algopyrin

Mivel a relációban azonos sorok nem fordulhatnak elő, a *sorismétlés* ***összetett*** *kulcsot fog eredményezni.* Látható, hogy a B_azon *önmagában nem elég* a rekordok azonosítására. Szükség van a *Betegség*, sőt a Gyógyszer mezőre is. A három érték együttesen már egyértelműen azonosítja a táblázat sorait. Feltételezzük, hogy egy beteg egy bizonyos betegséget csak egyszer kap meg, valamint egy időben csak egy betegséggel kezelik.

B_azon	B_név	B_cím	Betegség	Oszt_az	Oszt_név	Főorvos	Gyógyszer
444	Kiss Cili	Ajka	sérv	01	sebészet	Dr. Doktor	Algopyrin
444	Kiss Cili	Ajka	sérv	01	sebészet	Dr. Doktor	Semicillin
444	Kiss Cili	Ajka	tyúkszem	02	szemészet	Dr. Joó	Semicillin
333	Nagy Pál	Veszprém	tyúkszem	02	szemészet	Dr. Joó	Sumetrolim
333	Nagy Pál	Veszprém	tyúkszem	02	szemészet	Dr. Joó	Demalgon
444	Kiss Cili	Ajka	sérv	01	sebészet	Dr. Doktor	Algopyrin

Felületes szemlélődés után esetleg a ***B_azon*** és a ***Gyógyszer*** értékeket is alkalmasnak találnánk az azonosításra. A táblázat második és harmadik sorában azonban ***az elsődleges kulcs megegyezne*** (444, Semicillin), ami nem engedhető meg egy 1NF-ban lévő relációban. Ha a ***Betegséget*** komponensként bevonjuk a kulcsba, akkor az azonosítás már egyértelműnek látszik (ha az előfordulási adatok általánosíthatók).

Ez a táblázat már ***első normálformában*** (1NF) ***van***, mert:

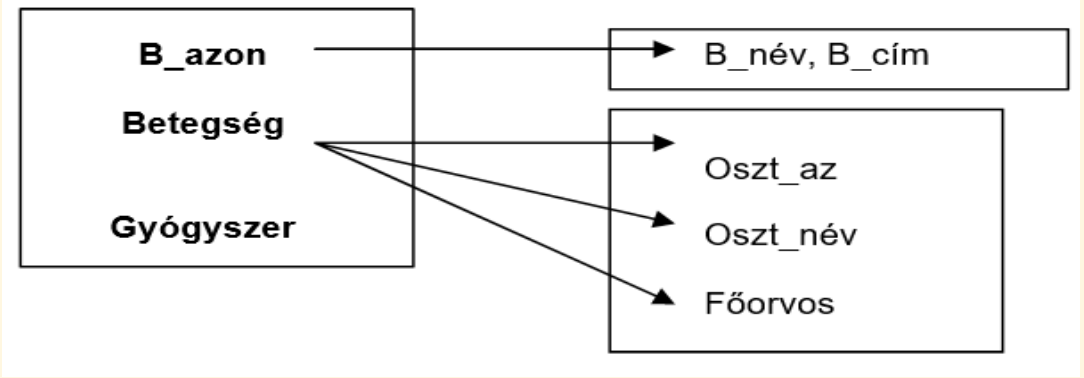
- *minden sora különböző*

- az *oszlopok száma és sorrendje minden sorban azonos*
- minden oszlop *csak egy attribútumértéket* vesz fel
- minden sorhoz *egy egyedi kulcs* tartozik, amitől az összes többi attribútum *funkcionálisan* függ.

A **második normalizálási** lépéshez meg kell vizsgálni, melyek azok az attribútumok, amelyeknek ***az egyes összetevőktől egyértelműen függenek*** és melyek azok, ***amelyeket az összetett kulcs határoz meg***. Ezt követően hozzunk létre olyan táblázatokat, amelyekben *az összes* ***nem-kulcs-attribútum teljesen függ*** *az elsődleges kulcstól*. A ***teljesen*** azt jelenti, hogy *az összetevők száma* nem csökkenthető, vagyis *a kulcs minimális*. Ha a kulcs *egyszerű*, akkor a teljes függés ***ugyanazt jelenti***, mint a funkcionális függés.

- a beteg azonosítójától teljesen függ a beteg neve és címe
- a betegségtől függ az osztály azonosítója és neve, valamint a főorvos
- a harmadik összetevőtől (gyógyszer) nem függ semmi más adat, de szükséges a sorok megkülönböztethetősége céljából.

A leírt összefüggéseket rajzzal is ábrázolhatjuk:



47. ábra – Normalizálás, 2. lépés

A rajz alapján a józan ész azt diktálja, hogy *három táblázatra* célszerű bontani 1NF táblázatunkat.

Beteg *reláció (adattábla)*

B_azon	B_név	B_cím
444	Kiss Cili	Ajka
333	Nagy Pál	Veszprém

Ki_Mire_Mit_Szed

B_azon	Betegség	Gyógyszer
444	sérv	Algopyrin
444	sérv	Semicillin
444	tyúkszem	Semicillin
333	tyúkszem	Sumetrolim
333	tyúkszem	Demalgon

Osztály

Betegség	Oszt_az	Oszt_név	Főorvos
sérv	01	sebészet	Dr. Doktor
tyúkszem	02	szemészet	Dr Joó

A három táblázat (reláció) már ***második normálformában van***, mert

- *1NF*-ben van (előfeltétel) és
- a nem-kulcs-attribútumok ***funkcionálisan teljesen függenek*** az elsődleges kulcstól

Ebben a lépésben már megjelennek az **idegen kulcsok** is. A *Beteg* relációban a ***B_azon*** az elsődleges kulcs, mert ez határozza meg a többi tulajdonságot. A *Ki_Mire_Mit_Szed* relációban elsődleges kulcs a ***B_azon***, a *Betegség* és a *Gyógyszer* attribútumok ***kombinációja***, mert ***ez határozza meg az adott sort***. *Idegen kulcsok* a ***B_azon*** és a ***Betegség*** mezők, mert a másik két táblázatra ezekkel a kulcsokkal hivatkozhatunk. Az *Osztály* relációban az *elsődleges* kulcs a ***Betegség***.

161. oldal (– „**B**” tételsor)

Figyelemre méltó azonban az adatok számának csökkenése. Az 1NF táblázat összesen 40 adatot tartalmazott, míg itt a 2NF három relációja összesen 29 adatot tartalmaz csupán! Itt azonban még nem kell megállnunk. Ha az Osztály táblázatot jól megnézzük, észre lehet venni, hogy vannak mezők, amik ***közvetve is függenek*** az elsődleges kulcstól: A Betegség meghatározza az ***Oszt_az*** osztályazonosítót és az osztály azonosítója meghatározza az ***Osz_név*** osztálynevet és a főorvost. Emiatt még mindig maradtak rendellenességek.

Tegyük fel, hogy megjelenik egy új betegség. Ekkor nemcsak a betegség nevét és az osztály azonosítóját kell bevinni, hanem az osztály nevét és főorvosát is. Lássuk, mi történik, ha megjelenik a vakbél(-gyulladás), mint új betegség:

Osztály

Betegség	Oszt_az	Oszt_név	Főorvos
sérv	01	sebészet	Dr. Doktor
tyúkszem	02	szemészet	Dr Joó
vakbél	01	sebészet	Dr. Doktor

Az első és a harmadik sorban ***sok a közös adat***. Ahhoz, hogy tudjuk, milyen betegséggel melyik osztályon fekszik a beteg, szükségünk van az osztály azonosítójára, de teljesen *felesleges újra tárolni az osztály és a főorvos nevét*. Ez sok – ugyanazon osztályra tartozó betegség esetén – fölösleges tárolást jelentene. Pl., ha 5 új betegség jelenne ugyanazon az osztályon, az 10 db feleslegesen felvitt adatot jelent!

Emiatt a táblázatot ***érdemes két további táblázatra*** bontani, megszüntetve ezzel a ***közvetett összefüggéseket***:

Betegség

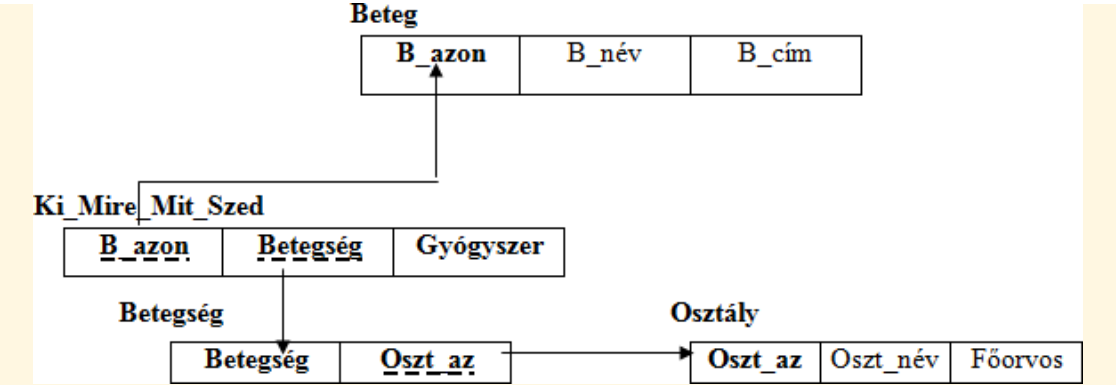
Betegség	Oszt_az
sérv	01
tyúkszem	02

vakbél	01

Osztály

Oszt_az	Oszt_név	Főorvos
01	sebészet	Dr. Doktor
02	szemészet	Dr. Joó

A *Betegség* reláció elsődleges kulcsa a ***Betegség***, *idegen-kulcsa* az ***Oszt_az***. Ezzel hivatkozhatunk az *Osztály* reláció megfelelő rekordjára. Az ***Oszt_az*** az *Osztály* elsődleges kulcsa.



48. ábra – 3. normálformában levő adattáblák

Ábrázolva a relációk közötti összefüggéseket – az így kapott reláció mindegyike harmadik normál formában van, mert

- 2NF-ben van
- **funkcionális** függés ***csak az elsődleges*** *kulcsból indul ki*; vagyis *megszüntettük* ***a közvetett (tranzitív)*** *függéseket.*

Ha most tesszük fel a korábbi kérdéseket, látható, mennyivel egyszerűbben és gyorsabban kapható rájuk válasz. A gyógyszerszámláláshoz elegendő egy kisebb táblázatot végignézni, egy esetleges főorvosváltás pedig az *Osztály* táblázat ***egyetlen*** rekordjának módosítását jelenti.

13.4. Az adatbázis-terv dokumentációja, az adatbázis-tervezés lépései

13.4.1 Az igények összegyűjtése, elemzése

Fel kell deríteni a fő alkalmazási területeket, tanulmányozni kell az adott területtel rokon, már +lévő alkalmazásokat és azok dokumentációit. A felhasználói igények, elvárások összegyűjtése érdekében célszerű a mostani és későbbi felhasználókkal elbeszélgetni. Olyan specifikációt kell készíteni, mely tartalmazza a felhasználói igényeket kielégítő tárolandó adatokat, valamint a feldolgozási műveleteket, tranzakciókat, lekérdezéseket.

13.4.2 Koncepcionális terv elkészítése

A terv készítésének folyamán kell a magas szintű modellt kialakítani. A modell segítségével + kell fogalmazni az előre tervezhető lekérdezéseket és tranzakciókat. Segítségével közérthető formában mutatja az adatbázis szerkezetét, az adatcsoportokat és azok kapcsolatait, +szorításait és segíti a felhasználó és a programozó közötti párbeszédet. A koncepcionális terv elkészítéséhez leggyakrabban az egyed-kapcsolat modellt vagy a relációs modellt alkalmazzák.

13.4.3 Adatbázis-kezelő rendszer kiválasztása

Az adatbázis-kezelő rendszer kiválasztásában igen sok tényező játszhat szerepet, mint pl.: gazdaságossági megfontolások, a rendszer szolgáltatásai, felhasználóbarát felület, több progi is használja ugyanazt az adatbázist, az adatok sűrűn változnak, nagy az adatbázis, a különböző adattípusok között bonyolult kapcsolatrendszer áll fent, az adatbiztonságra nagy az igény.

13.4.4 Adatbázis-kezelő rendszertől függő leképezés

A leképezés lényegében a logikai adatmodelltől függő szabályok alkalmazása.

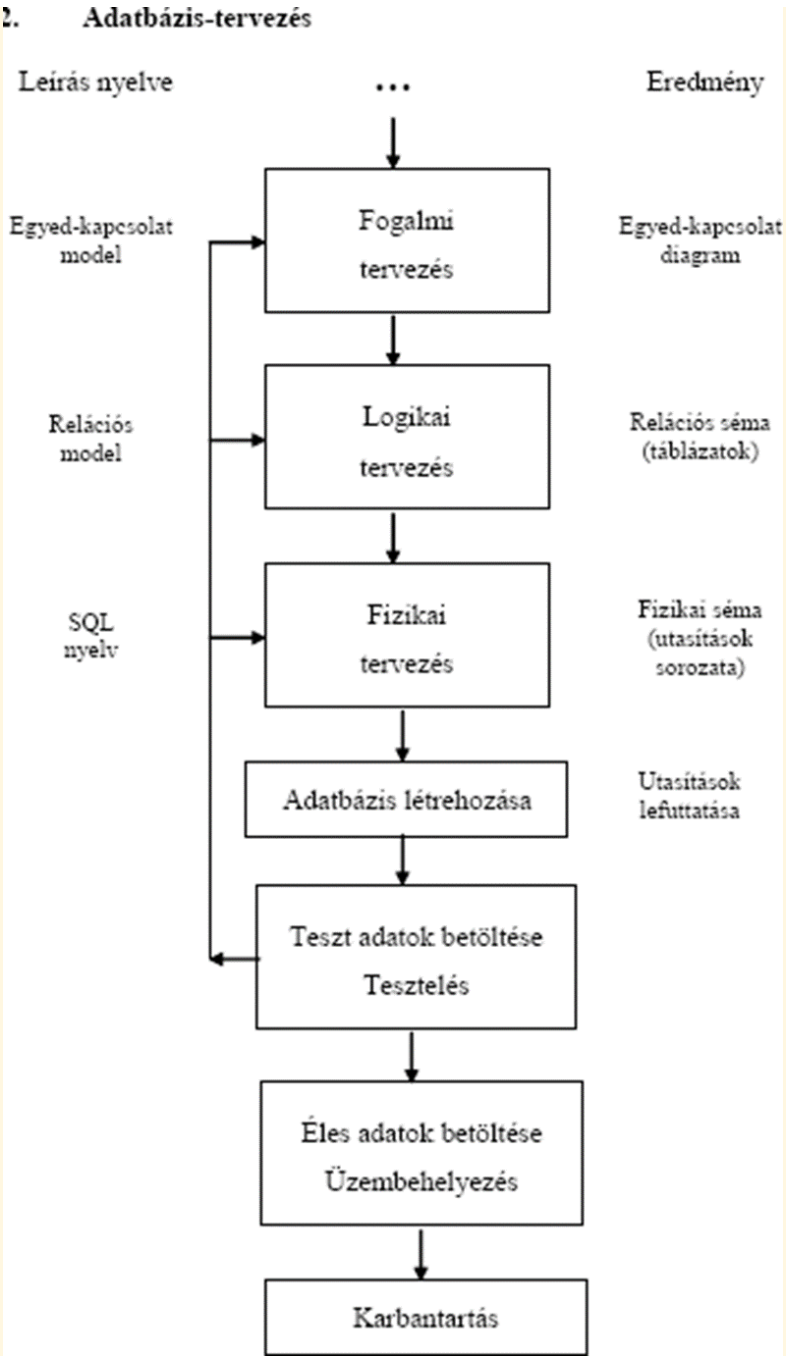
13.4.5 Fizikai tervezés

Itt kell dönteni a tárolási szerkezetről és a hozzáférési módokról, melybe az adatbázis-kezelő rendszeren kívül az operációs rendszer is beleszólhat. Relációs adatmodell használata esetén a fizikai tervesben fontos szerepet játszik az indexelés. Döntő fontosságú lehet a lekérdező és aktualizáló – vagyis a beszúró-, törlő- és módosító- – műveletek gyakorisága és egymáshoz való viszonya. Az indexelésnél azt is figyelembe kell venni, hogy az adatok elérésének gyorsítása mellett ez többlet helyfoglalással jár.

13.4.6 Megvalósítás

Az adatleíró nyelven írt sémák alapján létrejön az adatbázis szerkezete és az üres fájlok. Az így kapott adatbázist feltölthetjük adatokkal. Készülhetnek űrlapok, lekérdezések stb. Valódi adatok felvitele előtt célszerű a rendszert mintaadatokkal kipróbálni, h. ne túl későn derüljenek ki az esetleges hibák. A rendszer +valósítása után használat közben felmerülhetnek problémák, amiket orvosolni kell.

(Az adatbázis-terv dokumentációjával kapcsolatban nem igazán található példa. Itt szokták megadni a progi-kiválasztást; relációs adatmodellnél a normalizálás során létrejött táblákból a fizikai táblák adatait, esetleg külön ábrázolva a kapcsolatokat is; képernyőtervek – lekérdezések – űrlapok.)



49. ábra – Adatbázis-tervezés

14. SQL-adatbázis, adattábla, index, nézet létrehozása és törlése. Adattábla szerkezetének módosítása. Kulcsok, külső kulcsok megadása, kapcsolatok beállítása. További megszorítások elhelyezése

14.1. SQL-adatbázis, adattábla, index, nézet létrehozása és törlése, Adattábla adatszerkezetének módosítása

SQL-adatbázis: Az SQL **relációs adatmodellt** kezel, igy a nyelv fő objektuma a **reláció**, amit az SQL-ben ***táblának*** nevezünk. ***Táblák névvel ellátott csoportja alkot egy adatbázist***. Az adatbázis olyan adatoknak a halmaza, melyeket együtt kezelünk, az adatok kapcsolataikkal együtt történő ábrázolását, tárolását jelenti.

14.1.1 Az adatbázis létrehozása

```
CREATE DATABASE <adatbázisnév>;
```

Regisztrálásra kerül az új adatbázis és létrejön róla mindennemű bejegyzés a rendszer-katalógustáblákban.

14.1.2 Az adatbázis törlése, egyéb műveletek

```
DROP DATABASE <adatbázisnév>;
```

A `START DATABASE <adatbázisnév>;` megnyitja a létező adatbázist (egyszerre csak egy aktív).

`STOP DATABASE;` – lezárja az aktív adatbázist.

`SHOW DATABASE;` – információ a létező adatbázisokról.

14.1.3 Adattábla

Az adatokat tartalmazó relációt nevezzük *adattáblának*. A tábláknak ***oszlopai*** a *tulajdonságok* (attribútumok), melyek ***típusát*** meg kell adni az adattábla definiálásakor.

A *tábla nevének* az egész adatbázisban ***egyedinek*** kell lennie (max. 8 karakter, betűvel kezdődik, folytatódhat betűvel, számmal, „_”-karakterrel). Az *oszlop nevének* egy ***táblán belül egyedinek*** kell lennie (max. 10 karakter, betűvel kezdődik, folytatódhat betűvel, számmal, „_”-karakterrel). Adattábla ***definiálása***:

```
CREATE TABLE <táblanév> (<oszlopnév> adattípus(adathossz),
[<oszlopnév> adattípus(adathossz)], [...]);
```

SZEMELY {kod,nev,nem,anya,apa} *reláció*

```
CREATE TABLE szemely
(kod char(3),
nev char(12),
nem logical,
anya char(3),
apa char(3));
```

A tábla létrehozásánál a *névadást* követően azonnal a *szerkezet* megadása következik: a felsorolt oszlopoknak ***azonosítójuk***, ***típusuk*** és ***hosszuk*** van.

14.1.4 A tábla szerkezetének módosítása

A *tábla* ***törlése***:

```
DROP TABLE táblanév;
```

A tábla kitörlése az adatbázisból (a megfelelő katalógus-beli bejegyzések karbantartásával).

Az SQL-ben nincsen lehetőség *igazi módosításra*, csak ***új oszlopok felvételére***. (egyes verziókban módosítani is lehet – MODIFY) Minden más változtatást úgy tudunk elvégezni, hogy létrehozunk egy új táblát, és azt a már meglévő táblázatból töltjük fel.

Oszlop ***hozzáadása***:

```
ALTER TABLE táblanév ADD | MODIFY(oszlopnév adattípus(adathossz)
[,oszlopnév adattípus (adathossz)]...);
```

Index: Az indexállomány egy adott táblából néhány, kiemelt, rendezett oszlopból áll, a rendezés lehet *növekvő és csökkenő*) Ha az indexelt tábla értékei alapján keresünk, akkor a keresés gyorsabb lesz.

Indextáblát a következő alakú parancs hozza létre:

```
CREATE [UNIQUE] INDEX indextábla -név ON táblanév (oszlopnév
[[ASC|DESC][,oszlopnév[ASC|DESC]]..);
```

A parancs hatása: Az **ON** után adott tábla *felsorolt oszlopait rendezi* (növekedően ASC és csökkenően DESC esetén) és belőlük egy, az **INDEX** szó után megadott nevű táblát készít. Az **UNIQUE** azt jelenti, hogy ***az oszlop értékei egyediek***, s ha ez esetben ismétlődő értékek is vannak az oszlopban, a rendszer hibát jelez. *Nézettáblát nem* lehet indexelni.

Indexálás ***törlése***:

```
DROP INDEX Indextábla-név;
```

14.1.5 Nézet létrehozása és törlése

Nézettábla létrehozásakor ***egy külön táblába*** definiáljuk az adatbázis azon részeit, amelyre szükségünk lesz. A virtuális vagy *nézettábla* ***fizikailag nem jön létre***, mégis táblaként kezelhető, ***a rendszer csak a definícióját őrzi*** a katalógustáblázatban. A nézettáblát valódi

táblákból származtatjuk oly módon, hogy a kijelölt táblá(k)ra ***lekérdezést írunk***. Valahányszor a nézettáblahoz fordulunk, a rendszer az alaptáblát kérdezi le. Nézettábla lehet további nézet alaptáblája is.

Nézet-tábla definiálása:

```
CREATE VIEW Nézetábla-név [oszlopnév-lista] AS SELECT... [WITH
CHECK OPTION]
```

```
CREATE VIEW nézet-táblaneve
AS
SELECT oszlop-neve FROM melyik táblákból –(*) összes
WHERE feltétel megadása;
```

- ugyanúgy le lehet kérdezni, mintha *adattábla lenne*
- ha *egyszerű* a definíciója, akkor a sor bekerül/törlődik (származtatásában nem tartalmaz DISTINCT-, GROUP BY-záradékot, SELECT-beli kifejezést, ill. alaptáblája nem nézet)
- *adatismétlődés csökkentése* miatt hasznos
- az adatbiztonság megszervezésének legegyszerűbb módja
- bizonyos felhasználók elől el kell rejteni egy tábla különböző részeit, de bonyolultabb esetben helyesebb *ideiglenes fizikai táblát* létrehozni

A *nézettábla* ***törlése***:

```
DROP VIEW <Nézettábla-név>;
```

14.2. Kulcsok, külső kulcsok megadása, kapcsolatok beállítása.

14.2.1 Kulcsok

Egy tulajdonságot (attribútum) vagy tulajdonságok egy csoportját kulcsnak nevezzük, ***ha a tábla mindegyik sorát egyértelműen meghatározza***, de ha bármely attribútumot elhagynánk a kulcsból,

akkor *az így megmaradt oszlopok kombinációja már nem* tudná egyértelműen meghatározni azt. Vagyis a kulcsban ***az attribútumok értékei különböznek***, nincs két olyan sor, amelyben megegyeznének. Ha a kulcs ***egyetlen*** attribútumból áll, akkor a kulcsot ***egyszerű kulcsnak*** nevezzük, ha két vagy ***több oszlop*** kombinációja alkotja, akkor ***összetett kulcsnak***.

- **Elsődleges kulcs** (primary key): az a kulcs, melyet a kulcsjelöltek közül választunk ki, és kulcsként használjuk. A ki nem választott kulcsjelölteket alternatív kulcsnak nevezzük. Az elsődleges kulcsnak ***nem lehet NULL*** az értéke.
- **Külső kulcs** (idegen-kulcs, foreign key): A tábla azon attribútumai, ***amelyek egy másik táblában kulcsot alkotnak***. A külső kulcs *nem azonosítja a rekordokat*, nem valódi kulcs, csak a táblák (sorai) közti kapcsolatot fejezi ki.

Ez azt jelenti, hogy *relációs modellben* – amikor a redundancia megszüntetése miatt *több kisebb táblára* fogjuk bontani az adatbázisunkat – az egyes táblák közt fennálló kapcsolatokat úgy érjük el, hogy az ***egyik tábla kulcsát bevisszük a másik tábla oszlopai közé***. Azt is mondhatnánk, hogy a külső kulcs értékei *egyértelműen fognak rámutatni* egy másik tábla valamely sorára a kulcs alapján.

Gyakran hívjuk ***gyerektáblának*** azt, ***ahol külső kulcsként*** szerepel egy másik tábla - az ő ***szülőtáblájának*** - kulcsa.

14.2.2 Elsődleges kulcs megadása

```
CREATE TABLE táblanév (attrib1 típus(n) ... PRIMARY KEY (attrib1,
...));
```

PRIMARY KEY kulcsszóval az elsődleges kulcsot, vagy UNIQUE kulcsszóval egyedi kulcsot hozunk létre. Csak *egy elsődleges*, de akár több *egyedi kulcsa* lehet a táblának! Ezek a kulcsok természetesen nem vehetnek fel *ismeretlen* értéket (NOT NULL).

Amikor a kulcs egyszerű, akkor az őt alkotó attribútum mögött is megadhatjuk megszorításként, de összetett kulcs esetén az attribútumok felsorolása után sorolhatjuk fel ***az elsődleges kulcs összetevőit zárójelek között***. Az adatbázis-kezelőtől függ, hogy az egyedi kulcsok definiálásakor létrehozza-e az indexet; az elsődleges kulcshoz biztosan létrehozza.

14.2.3 Idegen-kulcs definiálása

```
CREATE TABLE gyerek-tábla (attrib1 típus(n)... FOREIGN KEY (attribútumok) REFERENCES (szülő-tábla) );
```

A **hivatkozási épség** fenntartása érdekében adjuk meg a külső kulcsokat. A külső kulcs szerkezetének ***azonosnak kell lennie azon elsődleges kulcséval***, melyre hivatkozik (a kulcsot alkotó *oszlopnevek egyezése nincs* kikötve). Természetesen lehet több idegen kulcs is a táblában.

14.2.4 Kapcsolatok beállításai

Kapcsolat (relationship): Az ***egyedek*** (vagy tulajdonságaik) ***közötti viszony***. A kapcsolatokat megkülönböztethetjük annak megfelelően, hogy az *egyedhalmazok közötti* viszonyt vizsgáljuk, vagy az egyes egyedek *tulajdonsághalmazai közötti* viszonnyal foglalkozunk. Az egyedhalmazok közötti kapcsolat, a táblák (relációk) közötti kapcsolatban fog megjelenni. Az egyedhalmaz tulajdonsághalmazai közötti kapcsolatokat pedig a relációs modellnél vizsgáljuk, amikor meghatározzuk a funkcionális függőséget. A következőkben **T** a hivatkozó, **S** a hivatkozott táblát jelenti.

- **REFERENCES** esetén **ON**-feltételek megadásával szabályozhatjuk a rendszer viselkedését.
  - A **megszorítások** típusa (RESTRICT, SET NULL, CASCADE)
  - **Műveletek** (UPDATE, DELETE)

- **Alapértelmezés** (ha nincs ON-feltétel): A hivatkozó táblában nem megengedett olyan beszúrás és módosítás, amely ***a hivatkozott táblában nem létező kulcs értékre hivatkozna***, továbbá a hivatkozott táblában nem megengedett olyan kulcs módosítása vagy sor törlése, *amelyre a hivatkozó tábla hivatkozik*.

❖ ON UPDATE CASCADE: Ha a hivatkozott táblában ***változik a kulcs értéke***, akkor a hivatkozó tábláében is.

❖ ON DELETE CASCADE: Ha a hivatkozott táblában *törlünk* egy sort, akkor ***a hivatkozó táblában is törlődnek*** a kérdéses sorok.

❖ ON UPDATE SET NULL: Ha a hivatkozott táblában változik a kulcs értéke, akkor ***a hivatkozó táblában NULL*** lesz.

❖ ON DELETE SET NULL: Ha a hivatkozott táblában törlünk egy sort, akkor a hivatkozó táblában NULL lesz.

```
CREATE TABLE Dolgozó
     (adószám  INT(10) PRIMARY KEY,
     név       CHAR(30),
     nem       CHAR(1) CHECK (nem IN ('F', 'N')),
     osztálykód CHAR(10) REFERENCES Osztály(osztálykód)
               ON UPDATE CASCADE
               ON DELETE SET NULL );
```

14.3. További megszorítások elhelyezése.

Attribútum-értékekre vonatkozó korlátozás: A CREATE TABLE attribútumai mögött NOT NULL CHECK (feltétel-)záradékkal gondoskodhatunk az oszlopérték helyességéről. DEFAULT érték definiálásával pedig az alapértelmezés szerinti érték megadásáról gondoskodhatunk.

14.3.1 Egy oszlopra, mezőre vonatkozó megszorítások

A NULL az attribútum definíciójában arra utal, hogy az adat megadása nem kötelező, ez az alapértelmezés, ezért a legritkább esetben írják ki.

NOT NULL az attribútum definíciójában arra utal, ***hogy az adat megadása kötelező***, azaz nem vihető be olyan sor a relációban, ahol az így definiált adat nincs kitöltve.

PRIMARY KEY – ez az oszlop a tábla elsődleges kulcsa.

UNIQUE – ez az oszlop a tábla kulcsa.

`CHECK(feltétel)` – csak a feltételt kielégítő értékek kerülhetnek be az oszlopba.

`[FOREIGN KEY] REFERENCES relációnév [(oszlop_név)]`, ez az oszlop külső kulcs

14.3.2 Több oszlopra vonatkozó megszorítások

- `PRIMARY KEY(oszlop1[, oszlop2, ...])` – ezek az oszlopok együtt alkotják az elsődleges kulcsot.
- `UNIQUE(oszlop1[,oszlop2, ...])` ezek az oszlopok együtt kulcsot alkotnak.
- `CHECK(feltétel)` – csak feltételt kielégítő sorok kerülhetnek be a táblába.
- `FOREIGN KEY (oszlop1[, oszlop2, ...]) REFERENCES reláció(oszlop1[, oszlop2, ...])`, az oszlopok külső kulcsot alkotnak a megadott tábla oszlopaihoz.

14.3.3 Önálló megszorítások

Az attribútumra és sorra vonatkozó megszorításokat a rendszer csak akkor ellenőrzi, ha az attribútum vagy reláció, melyre a feltétel vonatkozik, ***beszúrás vagy módosítás hatására változik meg***. Ezért,

hogy a több sort/ táblát érintő feltétel minden esetben igaz maradjon, önálló megszorításként adjuk meg.

```
CREATE ASSERTION <megszorítás neve> CHECK (<feltétel>)
```

Ezek az önálló feltételek elkülönülnek a táblák definíciójától, és ***egy vagy több tábla összefüggéseit*** szabályozzák. E feltételeket a rendszer *mindannyiszor megvizsgálja*, valahányszor beszúrás/módosítás/törlés történik az érintett táblák bármelyikében.

```
DROP ASSERTION megszorítás neve
```

Kitörli az adott nevű önálló megszorítást, majd újra definiálható.

Példák

```
CREATE TABLE osztaly (
  oszt_azon      INT2 PRIMARY KEY,
  nev            VARCHAR(14) NOT NULL,
  varos          VARCHAR(13) NOT NULL,
  CONSTRAINT  unev UNIQUE(nev));

CREATE TABLE alkalmazott (
  alk_azon     INT2 PRIMARY KEY,
  nev          VARCHAR(10) NOT NULL,
  beosztas     VARCHAR(9) NOT NULL,
  fonok        INT2 CONSTRAINT fonok_korl REFERENCES alkalmazott (alk_azon),
  belepes      DATE NOT NULL,
  fizetes      FLOAT4 NOT NULL,
  jutalom      FLOAT4,
  oszt_azon    INT2 NOT NULL,
  CONSTRAINT alk_kulso_kulcs FOREIGN KEY (oszt_azon) REFERENCES
               osztaly (oszt_azon));

CREATE TABLE fiz_oszt (
  f_oszt         INT2 PRIMARY KEY,
  min            FLOAT4 NOT NULL,
  max            FLOAT4 NOT NULL,
  CONSTRAINT  min_max CHECK (min < max));
```

50. ábra – Példák a megszorításokra

14.3.4 Triggerek

Fentebb már említettük, hogy ezek ***különleges tárolt eljárások***, mert az adatbázisban ***egy feltétel bekövetkezésétől függően „indítódik”*** a trigger. A trigger tehát alkalmas adathibák, hivatkozási referenciák megsértésének kiszűrésére. A megszorítások megsértésének megakadályozásán túl más célok is megvalósíthatók: adott feltételtől függően végrehajtódjanak a benne levő adatbázis-műveletek, vagy épp semmi se történjen.

```
CREATE TRIGGER tr_név
```

15. SQL adattábla sorainak felvitele, módosítása, törlés. Megszorítások figyelembevétele felvitel/módosítás/törlés esetén. Jogok kiosztása és visszavételezése

15.1. SQL adattábla sorainak felvitele, módosítása, törlése

Sorok bevitele:

```
INSERT INTO táblanév [(oszlopnév-lista)] VALUES (értéklista);
```

Ha az oszlopnév-lista elmarad, akkor a tábla definiálásakor megadott oszlopsorrendben kell az értéklista értékeit megadni. Az új sor fizikailag a tábla ***utolsó*** *sora utáni sorban* tárolódik.

15.1.1 A tábla rekordjainak (sorainak) módosítása

```
UPDATE táblanév SET oszlopnév=kfejezés[,oszlopnév=kfejezés,…]
[WHERE logikai kifejezés];
```

A felsorolt oszlopok értékeit a megadott kifejezéssel módosítja, de csak azokra a sorokra***, amelyek eleget tesznek a WHERE-ben szereplő***

logikai kifejezésnek. (Ha a WHERE elmarad, akkor az oszlop ***minden sorát*** módosítja)

Rekordok ***törlése***:

```
DELETE FROM táblanév [WHERE feltétel];
```

Törli a táblából azokat a sorokat, amelyekre teljesül a feltétel. Ha a WHERE elmarad, akkor az oszlop *minden sorát módosítja*.

15.2. Megszorítások figyelembevétele felvitel/módosítás/törlés esetén.

A **megszorítások** olyan *előírások, korlátozások*, amelyekkel megadhatjuk az ***adatbázis tartalmára*** vonatkozó kívánságainkat. A megszorításokat az adatbázisrendszer minden olyan akció során ellenőrzi, amely eredményeként az adatbázis tartalma módosul. A megszorítások lényege a ***hivatkozási épség*** (kulcsok, külső kulcsok) és az ***adatösszefüggések*** (pl. nem lehet NULL, értéktartományon belül kell legyenek bizonyos értékek stb.) *felügyelete* minden adatkezelés alkalmával. A megszorítások megadásuktól kezdve érvényesülnek, *nincs visszamenőleges hatásuk*. Késleltetett ellenőrzés végrehajtása lehetséges a DEFFERRED kulcsszóval.

15.2.1 A megszorítás-típusok

- **Kulcsok**: megszorításként azt jelenti, hogy ellenőrizze, hogy a táblában ne legyen olyan sor, amelyben a kulcsattribútomok (kulcsmezők, -oszlopok) értéke azonos lenne. Idegen kulcsok hivatkozási épségének biztosítása.
- **Attribútumértékekre vonatkozó** megszorítások: Az attribútum lehetséges értékeinek korlátozása. (Hibás adat bevitelének, valamilyen érték hibás adatra módosításának megakadályozása, értéktartomány megadása (NOT NULL, CHECK)).
- **Önálló megszorítások**: bármilyen feltétel ellenőrzése.

15.3. Jogok kiosztása és visszavételezése.

Az SQL kiköti a ***felhasználói nevek*** létezését, amiket fel lehet ruházni különféle jogokkal. Ilyen jogok az adattáblákra vagy nézettáblákra vonatkozó lekérdezés és szerkezeti módosítás, a ***3 karbantartó utasítás*** (felvitel, módosítás, törlés), illetve valamely táblára való hivatkozás, megszorító feltételek kezelése vagy az *indexelés*. Minden, SQL-ben létrehozott objektum tulajdonosa megadhatja, ***mely felhasználók milyen műveleteket végezhetnek*** az adott objektumon. Az így megadott jogok *visszavonhatók*. A jog tárgyát képező objektum megszüntetése maga után vonja ***az összes jogosultság*** *megszűntét*.

15.3.1 A táblákra vonatkozó jogosultság adományozása

A parancs formája:

```
GRANT ALL [PRIVILEGES] | <jogosultséglista> ON [TABLE] táblalista
TO PUBLIC|<felhasználólista> [WITH GRANT OPTION];
```

A parancs minden jogot (ALL PRIVILEGES) vagy a jogosultséglistában szereplő műveletekre való ***jogot*** *adja* ***a táblalistában szereplő táblákra*** mindenkinek (PUBLIC esetén) vagy a felhasználólistában szereplő személyeknek. Amennyiben a WITH GRANT OPTION szerepel, akkor az ***e jogokat kapók át is adhatják*** ezeket a jogokat másoknak.

A ***jogosultséglista*** elemeit a következő táblázatban foglalhatjuk össze:

A jogosultság neve:	A jogosultság jelentése:
ALTER	Jogosultság a tábla módosítására
DELETE	Jogosultság a tábla törlésére
INDEX	Jogosultság indextábla létrehozására
INSERT	Jogosultság új sor felvételére a táblázatba
SELECT	Jogosultság lekérdezésre
UPDATE	Jogosultság a tábla módosítására

51. ábra – Tábla-jogosultságok az SQL-ben

15.3.2 Jogosultság adományozása az adatbázison végzett műveletekre

A parancs formája:

```
GRANT adatbázisjog TO PUBLIC/<felhasználólista>;
```

A parancs ***jogosultságot ad az adatbázisra*** vonatkozóan vagy mindenkinek (PUBLIC) vagy adott felhasználóknak a felhasználólista szerint. Az adatbázisjogokat a következő táblázatban foglalhatjuk összsze:

A jog neve:	A jog jelentése:
CONNECT	- Hozzáférés a teljes adatbázishoz - Jog arra, hogy SELECT, INSERT, DELETE, UPDATE műveleteket végezzen más felhasználók tábláin, ha ilyen jogosultságot kapott a táblákra vonatkozó GRANT-tal - Jog nézettáblák és szinonim táblák létrehozására.
RESOURCE	- Minden CONNECT jogosultság - Jogosultság táblák és indextáblák létrehozására, jogosultságok adományozása ezekre a táblákra
DBA	-Teljes adatbázis-adminisztrátori jogkör

52. ábra – Adatbázis-jogok az SQL-ben

15.3.3 Táblákra vonatkozó jogosultság visszavonása

A parancs formája:

```
REVOKE ALL [PRIVILEGES] | jogosultságlista ON [TABLE] táblalista TO
PUBLIC |felhasználólista;
```

A parancs ***hatása***: Az összes jogosultságot (ALL PRIVILEGES vagy csak a jogosultságlistában felsoroltokat a megadott ***táblákra*** vonatkozóan mindenkitől (PUBLIC) vagy csak a listában szereplő felhasználóktól ***visszavonja***.

15.3.4 Adatbázis-jogosultságok visszavonása

A parancs formája:

```
REVOKE adatbázisjog FROM PUBLIC | felhasználólista;
```

A parancs az ***adatbázisjogokat*** mindenkitől (PUBLIC) vagy a listában szereplőktől ***visszavonja***.

16. Adattáblák lekérdezése (egy táblában nyilvántartott adatok, kapcsolatban lévő adattáblák, adattáblákban fennálló korlátozások, adattáblák hagyományos lekérdezése, adattáblák lekérdezése szabványos lekérdező mondattal)

16.1. Lekérdezés összeállítása és végrehajtása az SQL-ben.

A lekérdezés hatására egy úgynevezett **eredménytábla** (E-tábla) **jön létre**, amelynek egy vagy több oszlopa, és egy vagy több sora lehet. Az E-tábla csak ***ideiglenesen jön létre***, elmentéséről gondoskodni kell. A SELECT-paranccsal kiadható lekérdezések típusát az ***alparancsok***, valamint az *alparancsok* ***operandusai*** adják meg, szerkezete kötött, az alparancsok csak megfelelő sorrendben írhatóak:

```
SELECT[…]
INTO            Az E-tábla egy sorának tárolása
FROM            Descartes-szorzat
WHERE           szelekció, sorok kiválasztása
GROUP BY        csoportosítás
HAVING          csoportok közötti választás
UNION           Eredmény-táblák (E-táblák) összefűzése (unió-művelet)
ORDER BY        E-tábla rendezése
SAVE TO TEMP    E-tábla megőrzése, elmentése

SELECT [ALL/DISTINCT] oszlopnév[,oszlopnév…]|*
     FROM táblanév[,táblanév…]
     WHERE feltétel
     GROUP BY oszlopnév [,oszlopnév…]
     HAVING feltétel
UNION
     SELECT [ALL/DISTINCT] oszlopnévlista|*
     FROM táblalista
     WHERE feltétel
     GROUP BY oszlopnév [,oszlopnév…]
     HAVING feltétel
UNION
     ORDER BY oszlopnév [ASC|DESC];
```

A SELECT után ***mindig kell FROM***, ugyanis az utána lévő *táblalistában szereplő táblákból* történik az oszloplistában lévő oszlopok kiemelése (ha az oszloplista helyén * van, akkor az ***összes*** oszlopon). A DISTINCT hatása: az E(redmény)-tábla ***a duplikált sorokat csak egyszer tartalmazza***. Az ALL esetén pedig az azonosakat is annyiszor tartalmazza, ahányszor szerepelnek (ez az alapértelmezett).

Ha a ***sorok közül*** kell válogatni, akkor a **WHERE** mögött adjuk meg a szelekció feltételét, ami egy logikai kifejezés, tehát azok a sorok kerülnek majd megjelenítésre, amikre a feltétel igaz.

A GROUP BY alparancs: oszlopok értékei alapján **csoportosításokat** képez (egy csoportba az ***azonos értékűek*** tartoznak).

A HAVING alparancs: A GROUP BY csoportjaiból ***csak azok a sorok kerülnek*** az E- táblába, amelyek eleget tesznek a feltételnek.

Az UNION alparancs (művelet): A SELECT-ekkel létrehozott táblákat **összefűzi** (egymás alá teszi). Azok a sorok, amelyek ***közösek***, *csak egyszer* jelennek meg (unió, egyesítés). Az E-tábláknak ***kompatibilisnek*** kell lenniük: ***azonos oszlopszámok, azonos oszlop-típusok*** stb. (az oszlopneveknek nem kell megegyezniük).

A ORDER BY alparancs: Megadott oszlop vagy oszlopok szerint ***rendezi az Eredmény****(-tábla)****t***, növekvően (ASC-alapértelmezett) vagy csökkenően (DESC).

16.2. Belső (beágyazott) lekérdezés lehetősége.

A WHERE és a HAVING feltételében ***SELECT parancs*** is szerepelhet, amit ***belső*** vagy *beágyazott*, illetve ***allekérdezésnek*** nevezünk, de nem tartalmazhat ORDER BY és UNION parancsokat, és a GRUOP BY, SAVE TO TEMP alparancsok is *csak egyszer* fordulhatnak elő a teljes SELECT-ben. A külső SELECT ***a belső E-táblájából*** *hozza létre* az ő saját E-tábláját. A belső SELECT is tartalmazhat belső SELECT-et, vagyis **a SELECT-ek egymásba ágyazhatóak**. A belső (vagy al-) SELECT-et mindig ***zárójelpárban*** kell megadni. A kiértékelés menete:

- a belső SELECT kiértékelődik, és egy vagy több sort, esetleg oszlopértéket ***átad*** a külső SELECT-nek
- a külső SELECT ezen értékek alapján összeállítja az eredményt.

Egyetlen értéket tartalmazó belső E-tábla: Ha a belső E-tábla ***egyetlen értéket*** tartalmaz, akkor a WHERE-parancsrészben ***egyszerű összehasonlításokat*** végezhetünk.

Jelenítsük meg azoknak a dolgozóknak a nevét, fizetését, akiknek a fizetése kisebb az átlagfizetésnél:

```
SELECT vnev+knev, fiz
FROM dolgozó
WHERE fiz<(SELECT AVG(fiz) FROM dolgozo);
```

Több értéket tartalmazó belső E-tábla: Amikor a belső SELECT-nek több értéke van, akkor a WHERE utasításrészben ***4 féle „halmazos" logikai feltétel*** szerepelhet:

- IN predikátum(tulajdonság): **Igaz** értéket ad vissza, ha az ***egyenlőség*** valamelyik belső-E-táblabeli értékre igaz.
- ANY predikátum: **Igaz**, ha a megadott összehasonlítás ***valamelyik*** belső-E-táblabeli értékre igaz.
- ALL predikátum: **Igaz** értéket ad vissza, ha a megadott összehasonlítás **valamennyi** belső-E-táblabeli értékre igaz.
- EXISTS predikátum: Kiválasztja azon sorokat, amelyekhez a belső-E-táblában ***egy vagy több sor tartozik*** (tehát „létezik").

17. A kliensoldali programozás alapelemei az Internetes alkalmazások fejlesztésénél. A kapcsolódó technológiák rövid bemutatása: HTML, XHTML, XML, CSS, XSL. A kliensoldali script nyelvek használata.

17.1. A kliensoldali programozás alapelemei az Internetes alkalmazások fejlesztésénél.

A tisztán kliensoldali megoldások esetén a kibővített (pl. térinformatikai) funkcionalitást a kliens-számítógépen futó alkalmazás biztosítja. Ezt a megoldást vastagkliens-megoldásnak is nevezik, mivel a munka nagyobbik része a kliens-számítógépen folyik, a hálózati sávszélesség mellett a kliens-számítógép teljesítménye a meghatározó. A webszerver az adatokat, esetleg a letöltendő programokat szolgáltatja. Ezek a rendszerek az aktív képernyőkezelés megvalósítása érdekében igyekeznek kihasználni a korszerű WEB-böngészők dinamikus-HTML-lehetőségeit. A kliensoldali VBscript-, illetve JavaScript-technológia lehetővé teszi számos funkció elvégzését a kliensoldalon. Az Internetes alkalmazások kliensoldali programozásának elemei: HTML- és XHTML-weblapok, CSS-lapok, illetve kliensoldali scriptek (VBScript, JavaScript).

17.2. A kapcsolódó technológiák rövid bemutatása

17.2.1 HTML

Hyper Text Markup Language olyan szöveges dokumentum, amely magában hordozza a dokumentum megjelenítéséhez szükséges, formázást leíró paramétereket. Így a dokumentumokat olyan szöveges formában lehet tárolni, amit kis sávszélességű hálózaton is rövid idő alatt lehet továbbítani és a kliens számítógépén – bármilyen operációs rendszerrel is üzemel az – a dokumentum megjeleníthető lesz.

A HTML nyelv *alapeleme* a ***címke***, vagy angol nyelven a ***tag*** („teg”). Ezek nyitó (<) és záró (>) „kacsacsőr-jelek” közé zárt egyszavas angol nyelvű dokumentum-elemnevek, vagy azok rövidítései (például <body> címke jelzi, hogy itt a dokumentum törzse kezdődik).

A **<head>-elem**:

- a HTML-dokumentum kísérőadatainak tárolására szolgál
- nincs attribútuma
- további elemek tárolója
  - <title> a címsorban megjelenő szöveg
  - <base> a dokumentum alapértelmezett helye
  - <link> kapcsolat más dokumentumokkal
  - <meta> információ a dokumentumról kulcsszavak leírás stb.
  - <script> JavaScript- vagy VBScript-kód helye
  - <style> beágyazott (vagy linkelt) stíluslap

A **<body> elem**:

- eredetileg a dokumentum törzsének határait jelöli;
- kiterjesztették a dokumentum alap-háttér- és szövegszíneinek beállítására (amíg nem használtak CSS-t…);
  - háttérszín: <body bgcolor=”[szín]”>
  - szövegszín: <body text=”[szín]”>
  - hyperlinkek színe: <body link=”[szín]”>
  - meglátogatott link: <body vlink=”[szín]”>
  - aktív linkek színe: <body alink=”[szín]”>
- oldalbetöltésekhez kapcsolható *eseményindítók* is itt helyezhetők el

17.2.2 XHTML

Az XHTML (EXtensible HyperText Markup Language – kiterjeszthető HTML) a HTML pontosabb és „tisztább” verziója. A HTML 4.01 leváltására hozták létre. Az XHTML az XML egy alkalmazása, vagyis XML

eszközökkel is feldolgozható. Az XHTML felülről kompatibilis a HTML 4.01-el. Gyakorlatilag nincs jelentős eltérés a két nyelv között, csak a formai követelmények lettek szigorúbbak:

- Mindent kisbetűvel kell írni!
- Minden elemet le kell zárni, az üres elemeket önmagukban egy szóközzel és egy „/”-jellel:
.
- Az elemeket csak egymásba ágyazva lehet használni! Ez nem jó:

```
<b><i>Szöveg</b></i> Jó: <b><i>szöveg</i></b>
```

- Az attribútumokat ***idézőjelek*** közé írjuk! <cimszo magyar="alma”>
- Az attribútumoknak *legyen értéke*!

17.2.3 XML

XML (eXtensible Markup Language), magyarul ***bővíthető jelölőnyelv***. Az XML egy rendkívül hasznos technológia, ***de nem*** tekinthető egy ***önálló*** programozási nyelvnek. Az XML-dokumentumban lényegében egy „nyelvtant” képezünk le az adatszerkezeteink megadására, illetve leírására. Vagyis az XML használatával egyedi leíró-nyelveket alkothatunk, amelyek segítségével gyakorlatilag bármilyen információ leírható. A HTML-hez képest a változás az, hogy saját tag-eket is definiálhatunk. Nincsenek előre definiált tag-ek, azok minden egyes XML-alkalmazás esetén egyedileg definiálhatók. Az XML-ben tárolt adatok hardver-, és szoftverfüggetlen adatmegosztást tesznek lehetővé. Sokkal könnyebbé teszi olyan adatok létrehozását, amelyekkel különböző alkalmazások is dolgozni tudnak. Egy jól formázott XML-dokumentum esetén:

- A címkék neveiben a kis-és nagybetűk különbözőnek számítanak.

- Minden nyitó címkének meg kell legyen a záró párja is (kivéve a rövidített formátumban leírt üres elemeknél). Rövidített üres elem pl. a </Past>.
- Egymásba ágyazás esetén *a címkék nem lapolódhatnak át.*
- Az attribútumok értékének mindig idézőjelek között kell megjelennie.
- Minden XML-dokumentumnak kell tartalmaznia *gyökérelemet* (angolul root element).

Az XML szintaxisa: Az első sor a dokumentum formátumát adja meg: verziószámot, és a karakterkódolást.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<note>
<to>Tove</to>
<from>Jani</from>
<heading>Reminder</heading>
<body>Don't forget me this weekend!</body>
</note>
```

- A `<note>` és `</note>` tag-ek határolják az egész dokumentumot kitöltő gyökér-elemet.
- A közbülső négy sor a bejegyzés adatait tartalmazza.

17.2.4 CSS

A CSS (angolul Cascading Style Sheets) a számítástechnika egyik ***stíluslesíró nyelve***, amely a HTML- vagy XHTML-dokumentumok megjelenését írja le. A CSS-t a arra használják, hogy a lapok színét, hátterét, betűtípusait, elrendezéseit stb. beállítsák. A stíluslapokat külön ***.css***-kiterjesztésű állományban szokás elhelyezni. Így könnyedén lehet ugyanazt a megjelenítést adni a honlap összes oldalához, mindössze *egyetlen CSS-állomány* szerkesztésével. A CSS-technika alapvető célja, hogy szétválassza a dokumentum ***tartalmi*** részét a ***megjelenítési*** stílus-elemektől. A stíluslapok alkalmazásával hatéko-

nyabbá, gyorsabbá és rugalmasabbá tehetjük a webszerkesztést, elfelejthetjük a korlátozott formázási lehetőségeket, segítségükkel átláthatóbbá tehetjük forráskódjainkat.

17.2.5 XSL

Az XSL (Extensible Stylesheet Language – kiterjeszthető stílusleíró nyelv) támogató-technológia, ami leírja, hogy hogyan kell formázni az XML-dokumentumban található adatot. Hasonló tevékenységet old meg, mint a CSS-es, de XML-dokumentumokon. Egy sor az XML dokumentum tetején azt mondja meg, hogy a transform.xsl egy XSLT-stíluslap, ami az XML-ről HTML-formátumra való átalakításra vonatkozóan hordoz információt:

```
<?xml-stylesheet type="text/xsl" href="transform.xsl"?>
```

17.3. A kliensoldali script-nyelvek használata

A legnépszerűbb kliensoldali script nyelv a ***JavaScript***. *Dinamikus viselkedéssel* ruházhatjuk fel segítségével a HTML/XHTML-ben elkészített weblapunk elemeit. Ellentétben a HTML-lel (amely leíró-nyelv) és a CSS-sel (amely stíluslap-nyelv), a JavaScript *programozási nyelv*. A HTML-oldalakba ágyazott kliensoldali JavaScript-utasítások válaszolhatnak a felhasználói eseményekre (egérkattintás, űrlap adatbevitel stb.). Például, írhatunk olyan JavaScript-függvényt, amely ellenőrzi, hogy a felhasználó által beírt adatok *érvényesek*-e (telefonszám, irányítószám stb.). A HTML-oldalba beágyazott JavaScript minden *hálózati adatátvitel nélkül* meg tudja vizsgálni a beírt adatot, és ha a felhasználó érvénytelen adatot írt be, arra párbeszédablak megjelenítésével figyelmeztetheti.

A kliensoldali scriptek olyan programrészletek, amelyek forrásnyelvi alakban a HTML dokumentum keretében töltődnek le a felhasználó számítógépre. A kliens oldali scripteket a böngészőprogram hajtja végre.

A végrehajtás bekövetkezhet:

- egérmozgatás a HTML dokumentum objektumára
- egérkattintás, vagy
- a HTML dokumentum letöltése miatt.

A Javascript szolgáltatásai:

- Egyszerű használat
- Módosíthatja a HTML oldalak tartalmát, kinézetét
- Eseményekre tud reagálni
- Bevitt adat helyességének ellenőrzése
- Megvizsgálhatjuk a böngésző típusát, így ennek függvényében más-más böngésző-specifikus tartalmat tölthetünk be
- Sütiket (cookie) hozhatunk létre a kliens gépen való információtárolás érdekében.

Annak érdekében, hogy azoknál a klienseknél, akiknél nincs telepítve a megfelelő Java-környezet, a JavaScript-et úgy kell beépíteni a weboldalakba, hogy anélkül is helyesen működjenek. Azaz *diszkrét* JavaScript-megoldást kell alkalmazni.

A *diszkrét* JavaScript azt mondja, hogy a HTML-kódunkban ne használjunk JavaScriptet, válasszuk le, s ***tegyük külön fájlba*** scriptjeinket, s építsük fel úgy az oldalt, hogy azok nélkül is teljes funkcionalitással működjön. Ismerősnek tűnhet az ötlet: a mai CSS-technikák pontosan ezt mondják a stíluslapok esetén is: válasszuk szét a megjelenést és a tartalmat.

A JavaScript esetén a tartalom, s a használhatósági javítások szétválasztásáról van szó. A HTML azt mondja meg, mi ez a szöveg, a CSS azt, hogy hogyan nézzen ki, a JavaScript pedig azt, hogy hogyan viselkedjen (az oldal).

18. Az információs rendszer fogalma és összetevői. Adat, információ, tevékenység, esemény, felhasználó, szabvány. Az információs rendszer szintjei és nézetei. (Egy példán bemutatva)

18.1. Az információs rendszer fogalma és összetevői. Adat, információ, tevékenység, esemény, felhasználó, szabvány

18.1.1 Az információs rendszer fogalma:

Szervezett együttese az adatoknak (információknak), velük kapcsolatos eseményeknek, rajtuk végzett tevékenységeknek, ezekkel kapcsolatos erőforrásoknak, felhasználóiknak, és ezeket szabályozó szabványoknak és eljárásoknak.

- eljárásokat biztosít információk rögzítésére, feldolgozására és elérhetővé tételére
- valamilyen szervezethez vagy annak egy részéhez kapcsolódik
- a szervezet céljainak elérését segíti

Adat: értelmezhető, de nem értelmezett ismeret. (Tények, fogalmak, utasítások egyezményesen ábrázolt alakja, amely alkalmas arra, hogy az emberek vagy automatikus eszközök továbbítsák, értelmezzék vagy feldolgozzák.)

Információ: az adatokból elemzéssel, rendszerezéssel kinyert új ismeret, az ember által értelmezett (és valamilyen tudásbázisba beillesztett) adat. (Kinyert ismeret.)

Tevékenység: adatkezelés, előállítás, illetve ezeket vezérlő műveletek.

Esemény: információs tevékenységet kiváltó és azt lezáró momentum.

Felhasználó: az információs rendszerrel kapcsolatban lévő ember (csoport). Ismeretátadásban két fél érdekelt: az ismeret küldője és fogadója. Az emberek eltérő szereppel kapcsolódnak az információs rendszerhez, egy felhasználó több szerepet is betölthet, lehet

- adatszolgáltató,
- adatfelhasználó,
- alkalmazás-felhasználó,
- végső felhasználó.

Az információs rendszer fejlesztői is felhasználók; a vezető sajátos felhasználói szerepet tölt be.

Szabvány: az információs rendszerben vagy annak környezetében levő tényezőkre vonatkozó megegyezés. Hármas szerepet töltenek be: eligazítás, korlátozás és tájékoztatás.

18.2. Az információs rendszer szintjei és nézetei (Egy példán bemutatva).

18.2.1 Információs rendszer szintjei

Fogalmi szint: a valóságnak a kompromisszumoktól mentes képe.

Logikai szint: adott környezet korlátjainak megfelelően átalakított, kompromisszumokat tartalmazó fogalmi kép.

Fizikai szint: adott környezet konkrét fizikai adottságaira alkalmazott, tehát ilyen módon felhasznált és átalakított logikai kép. Ismeretek ábrázolása, tárolókon való elhelyezkedése. (Környezet által konkretizált verzió, konkrét végrehajtás és implementáció)

18.2.2 Információs rendszer nézetei (vetületei)

Információ- vagy adat-vetület: az alapvető ismeretek lényege és struktúrája (adatok, adatszerkezetek), viszonylag stabil, objektív az információs rendszer többi tényezőjétől viszonylag független.

Feldolgozás-vetület: az eseményt és a tevékenységet foglalja magába (lekérdezések, kimutatások, jelentések), viszonylag instabil, szubjektív, a rendszer többi részétől függ.

Környezet-vetület: meghatározza az információs rendszert, az eszközök objektív képességeit, az erőforrásokat, a felhasználók szubjektív igényeit, és a szabványok feltételrendszerét.

18.2.3 Példák

	Fog	Log	Fiz	Adat	Feldolg	Körny
A vezetők a keresleti trendeket színes, grafikus képernyőn akarják látni	X				X	
A munkavállaló, a pótlék és a pótlékra jogosultság különböző egyedtípusok		X		X		
A rendelésszámot 4bájtos fixpontos bináris számként tárolom			X	X		
Meghatározom a rendszer folyamatait, azok bemenetét és kimenetét		X				X
A függvény milyen algoritmussal állítja elő a bemenetből a kimenetet		X			X	
Nyilvántartásba veszem a felhasználók különféle igényeit	X					X
A közölt eljárásokat C nyelven programozom			X		X	

53. ábra – Példák a szintekre és a vetületekre

19. Az informatikai biztonság fogalma. A biztonsági rendszer tervezése, a tervezés szakaszai. Az egyes tervezési szakaszok fő feladatai. A kockázatelemzés célja és lépései. Az informatikai rendszerek elleni támadások típusai. Kriptográfiai módszerek és eszközök, azok gyakorlati alkalmazásai

19.1. Az informatikai biztonság fogalma. A biztonsági rendszer tervezése, a tervezés szakaszai. Az egyes tervezési szakaszok fő feladatai.

Az **informatikai biztonság** a védelmi rendszer olyan, a védő számára kielégítő mértékű állapota, amely az informatikai rendszerben kezelt adatok ***bizalmassága, sértetlensége és rendelkezésre állása*** szempontjából ***zárt, teljes körű, folytonos és a kockázatokkal arányos***.

Bizalmasság, annak biztosítása, hogy az információ csak az arra felhatalmazottak számára legyen elérhető.

Sértetlenség: Az adat tulajdonsága, amely arra vonatkozik, hogy az adat tartalma és tulajdonságai az elvárttal megegyeznek, ideértve a bizonyosságot abban, hogy az elvárt forrásból származik (hitelesség) és a származás ellenőrizhetőségét, bizonyosságát (letagadhatatlanság) is, illetve az elektronikus információs rendszer elemeinek azon tulajdonságát, amely arra vonatkozik, hogy az elektronikus információs rendszer eleme rendeltetésének megfelelően használható.

Rendelkezésre állás, annak biztosítása, hogy a felhatalmazott felhasználók mindig hozzáférjenek az információkhoz és a kapcsolódó értékekhez, amikor szükséges.

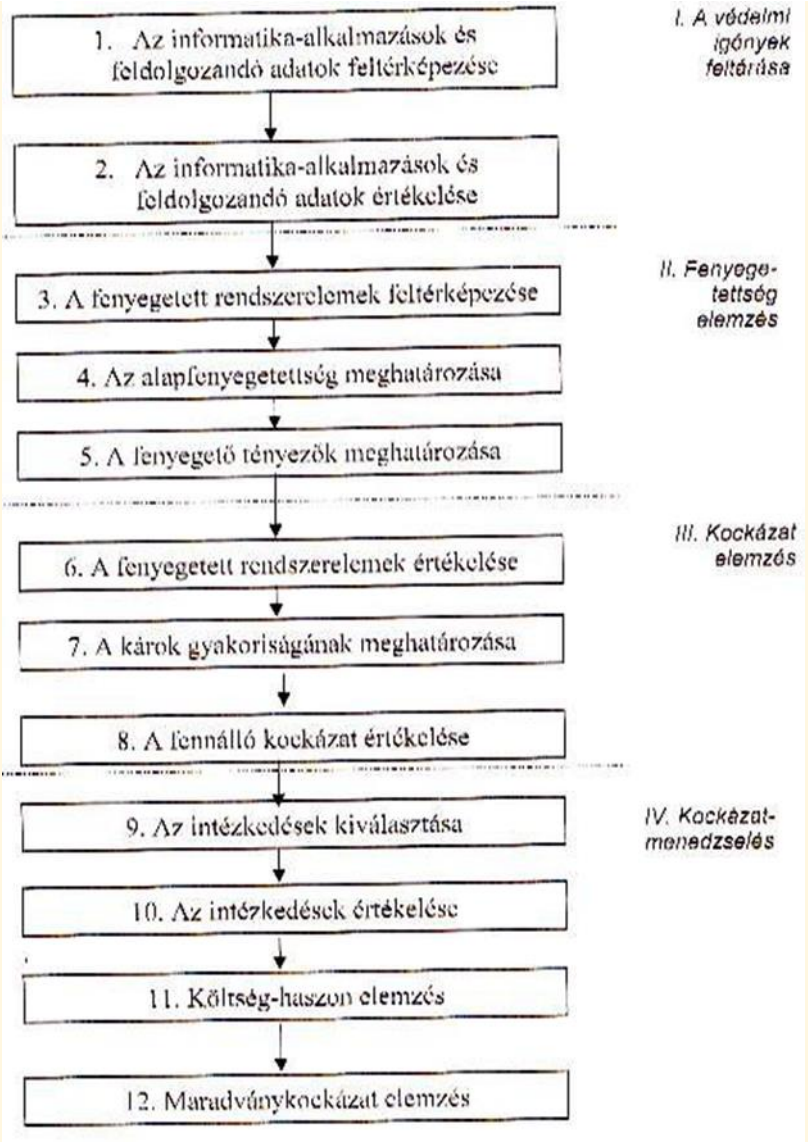
19.1.1 A biztonsági rendszer tervezése, a tervezés szakaszai

A védelmet ***teljes körűen és zártan*** kell kialakítani. A ráfordítás mértékét az elviselhető kockázat mértéke szabja meg, amelyet a kárérték és a bekövetkezési gyakoriság osztályok alapján felállított ***kockázati mátrixban*** kijelölt ***elviselhetőségi határ*** szabja meg. Ezt a határt minden szervezet informatikai biztonsági vizsgálatánál egyedileg kell meghatározni. *Szakaszai*:

- **Védelmi igények feltárása**: kiválasztjuk azokat az informatikai *alkalmazásokat*, amelyek a szervezet szempontjából a legfontosabbak, és később már csak ezekkel foglalkozunk. Ennek a szakasznak az a célja, hogy reális és teljes képet kapjunk a védendő rendszer felépítéséről, tartalmáról – gondoljunk csak bele, hogy hogyan védjünk meg egy rendszert, ha pontosan nem is tudjuk, mit kell védeni. Feltérképezés után valamennyi informatikai alkalmazást és feldolgozandó adatot sorba kell ál-

lítani és kiválasztani azokat, amelyek a legfontosabbak és védelmet igényelnek (akár *értékskála* készítése és hozzárendelése az elemekhez).

- **Fenyegetettségelemzés**: megkeressük az informatikai rendszer ***gyengepontjait*** és azokat a fenyegető tényezőket, amelyek az informatikai rendszerre veszélyt jelenthetnek. Az ITB (Informatikai Tárcaközi Bizottság) az informatikai rendszert nyolc tényezőre bontja, amelyek a teljes rendszert és annak környezetét lefedik:
  - *környezeti infrastruktúra* (épület, helyiségek, víz, világítás, telefont, védelmi berendezések stb.) fenyegetés: földrengés, árvíz, sztrájk, merénylet, jogosulatlan személyek, közműellátás stb.
  - *hardverfenyegetés*: műszaki hibák, környezeti hatások, szoftver által keltett probléma, személyek stb.
  - *szoftverfenyegetés*: szoftverhiba, vírus, kezelési hiba, karbantartási hiba stb.
  - *adathordozó-fenyegetés*: ellenőrizetlen folyamatok, gyári hibás termékek, privát felhasználás stb.
  - *dokumentációs fenyegetés*: hiányzó adminisztráció, ellenőrizetlen sokszorosítás stb.
  - *adat-fenyegetés*: hardver-szoftverhiba, személyek stb.
  - *kommunikáció-fenyegetés*: jogosulatlan bejutás, hálózati hibák, lehallgatás stb.
- **Kockázatelemzés**: az vizsgáljuk, hogy az informatikai rendszerre ***milyen káros hatása lehet*** a fenyegető tényezőknek. Meghatározzuk a lehetséges kár ***gyakoriságát*** és a ***kárértékét***.



54. ábra – Kockázatelemzés lépései

19.2. A kockázatelemzés célja és lépései

- **Fenyegetett rendszerelemek értékelése**, azaz a különböző rendszerelemekhez fontosságtól, prioritástól függően értéket

rendelni. Például, ha egy lemezes tároló különböző adatokat tárol, akkor a legnagyobb prioritású adat értékét adjuk neki.

- **Károk gyakoriságának meghatározása**, azaz egy skálán megbecsüljük, hogy milyen gyakran következhetnek be fenyegetések a rendszer ellen. Akár szakértő véleményét is kikérhetjük.
- **Fennálló kockázat értékelése**: egy ***mátrix*** segítségével értékeljük a kockázatot. Egyik tengely a gyakoriság, másik a kárérték lesz. A tábla alapján meg kell állapítani, hogy mely értékpárok jelentenek elviselhető, és melyek elviselhetetlen kockázatot. A kockázat nagysága a két értékből együttesen adódik. Az eredményt dokumentálni kell és a vezetőkkel, felelősökkel meg kell osztani.
- **Kockázat-menedzselés**: kiválasztjuk a fenyegető tényezők elleni intézkedéseket és azok hatását értékeljük. Megnézzük, hogy az egyes intézkedések milyen költségekkel járnak és milyen hasznot hoznak. Intézkedések általánosan: védett elhelyezés, tűzvédelem, vízvédelem, sugárzásvédelem, naplózás, védelmi eszközök, javítás, karbantartás, mentések, ellenőrzések, előírások stb.

19.3. Az informatikai rendszerek elleni támadások típusai

A támadás az informatikai rendszer valamennyi elemén keresztül történhet. Ezek a következők:

- **A környezeti infrastruktúra**: a számítóközpont épületének területe, maga az épület, az épületben lévő helyiségek, átviteli vezetékek, áramellátás, klíma, víz, világítás, telefon, és különböző rendeltetésű berendezések (belépés-ellenőrző, tűzvédelem, betörésvédelem). ***Gyengepontok***:
    - Nem védett átviteli vezetékek, kábelek, informatikai berendezések

- Illetéktelen személyek felügyelet nélküli jelenléte, vagyis a belépési biztonság hanyag kezelése.
- A védelmi berendezések működési módjának vagy gyengeségeinek jogosulatlanok általi +ismerése. Fenyegető tényezők:
- „Vis major": robbanás, repülőgép lezuhanása, sztrájk, háborús helyzet.
- Személyek által kifejtett erőszak: robbantásos merénylet, fegyveres behatolás, gyújtogatás, savazás, vandalizmus, betörés.
- Jogosulatlanok ellenőrizetlen belépése épületekbe helyiségekbe, szervezeten kívüli személyek által végzett, nem felügyelt munkálatok.
- Közműellátás: (áram, víz, telefon) és védelmi berendezések zavara vagy kiesése.
- Hardver: ide tartoznak a számítástechnikai eszközök a hálózati csatoló eszközök a hálózat építő eszközök, speciális biztonsági berendezések. Gyengepontok:
- Eltulajdonítás: a készülékek csekély mérete, súlya miatt a lopás könnyen lehetséges.
- Külső behatások miatti meghibásodás: hőhatás, vízzel való elárasztás, érzékenység az elektromágneses sugárzásra, mechanikai behatásokra.
- Tartozékok utánpótlásának szervezetlensége: pótalkatrészek, printer festékek. Fenyegető tényezők:
- Tervezési és kivitelezési hiányosságok.
- Személyekkel összefüggő fenyegetés: készülékek károsítása vagy roncsolása, ellopása, jogosulatlan szerelés és alkatrészcsere.

❖ **Az adathordozók**: ebbe a csoportba tartoznak a raktározott állapotú szoftverek, a biztonsági másolatokat, munkakópiákat,

archív adatokat jegyzőkönyvi adatokat tartalmazó adathordozók, valamint az újonnan beszerzett és még használatba nem vett, ill. a felszabadított adathordozók melyek tartalmára a továbbiakban nincs szükség. ***Gyengepontok***:

- Fizikai instabilitás: érzékenység a behatásokra.
- Kikapcsolható írásvédelem.
- Könnyen szállíthatóak, a szállítás nehezen ellenőrizhető. Fenyegető tényezők:
- Újrafelhasználásra vagy megsemmisítésre történő kiadás előzetes törlésük, felülírásuk nélkül.
- Ellenőrizetlen másolás ill. hozzájutás az adathordozókhoz.
- A szervezet tulajdonát képező adathordozók privát célú használata és privát adathordozók szolgálati használata.
- A szoftver: ebbe a kategóriába csak a használatban lévő szoftverek tartoznak, a raktározott állapotú szoftvereket az adathordozóknál tárgyaltuk. Gyengepontok:
- Specifikációs hiba, a progik átvételének és ellenőrzésének hiánya.
- Bonyolult felhasználói felület, felhasználó hitelesítés hiánya.
- Az események hiányzó jegyzőkönyvezése (pl. belépés, CPU használat, fájlok +változtatása).
- A rendszer védelmi eszközeinek könnyű kiismerhetősége.
- A hozzáférési jogok helytelen odaítélése.
- Más felhasználók ismeretlen programjainak használata (vírusfertőzés). Fenyegető tényezők:
- Jogosulatlan bejutás az informatikai rendszerbe a kezelői helyről vagy a hálózatról.
- Visszaélés a kezelési funkciókkal.
- Szoftver ellenőrizetlen bevitele, vírusveszély.

- Karbantartási hiba / visszaélés a karbantartási funkciókkal, helytelen karbantartási funkciók (távoli karbantartás a hálózaton)

❖ **Kommunikáció**: ezen elemcsoport tárgya fennállhat valamennyi adat továbbítási ideje alatt, amelyeket valamely szolgáltatás realizálása érdekében hálózaton továbbítanak. A hálózatok lehetnek az üzemi területen belül (pl. LAN-ok) vagy azon kívül (pl. közüzemi hálózatok), ill. e kettő kombinációja. A kommunikációhoz szükséges hardvert mindaddig, amíg az informatikai rendszer üzemeltetőjének felelősségi körén belül van, a hardver elemcsoportban, a vezetékeket pedig amennyiben az üzem területén belül vannak, a környezetiinfrastruktúra-elemcsoportban szerepeltetjük. ***Gyengepontok***:

- A hálózati szoftver és hardver hibái, azok manipulálhatósága.
- Üzenetek lehallgatása, +hamisítása, az adó és a fogadó hiányzó azonosítása.
- A jelszavak vagy titkos kulcsok nyílt szövegben való továbbítása.
- Függés az átvitel sorrendjétől.
- Lehetőség az üzenet elküldésének, kézhezvételének hiányzó bizonyítása. Fenyegető tényezők:
- Jogosulatlanok bejutása a hálózatba nem ellenőrizhető csatlakozások révén.
- Hálózati hardverek/szoftverek manipulálása, átviteli hibák.
- Nem várt forgalmazási csúcsok. Célzott terhelési támadások.
- A vezetékek kompromittáló sugárzásának kihasználása.
- A kapcsolat felépítésének lehallgatása – a lehallgatásnál a támadó illetéktelenül rákapcsolódik az átviteli vonalra,

az ott folyó adat- és kulcs-forgalmat figyeli. Az üzenetek gyűjtésével, különleges helyzetek észlelésével támadhat.
  - A kommunikációs kapcsolatok kikutatása (forgalmazás elemzés), a kommunikációs partnerek névtelenségének veszélyeztetése.
  - A kapcsolat felépítése meghamisított azonossággal, megszemélyesítéssel. A megszemélyesítés esetén a támadó beépül a kommunikációs összeköttetésbe, az üzeneteket elnyeli, és az ellen- állomások helyett válaszol mindkét irányba. Különös veszélyforrás lehet, ha a támadó az egymással kommunikáló állomásokat sorozatos ismétlésre kényszeríti, esetleg ugyanazon üzenet kétkülönböző rejtjelezett variációját szerzi meg, vagy valamelyik állomásról ismert választ kényszerít ki, amellyel megszerzi annak rejtjelezett változatát.

- **Személyek**: e csoportban csak olyan személyek szerepelnek, akikre közvetlenül vagy közvetve szükség van az informatikai rendszer használatához, ezáltal hozzáférhetnek a másik hét csoport elemeihez. A személyeket két nézőpontból kell figyelembe venni: egyrészt az üzemeltetéshez szükség van rájuk, ebből következően ők maguk is védelemigényes rendszerelemek. Másfelől viszont ők bírnak a belépés és a bejutás lehetőségével, ebből következően a fenyegetések jelentős része rajtuk keresztül realizálódik. ***Gyengepontok***:
  - A munkából való kiesés, hiányos kiképzés, a veszélyforrások ismeretének hiánya, a fenyegetettségi helyzet lebecsülése.
  - Kényelmesség, eltérő reakciók.
  - Hiányzó vagy hiányos ellenőrzés. Fenyegető tényezők:
  - Szándéktalan hibás viselkedés: stresszhelyzet, fáradság, hiányos ismeretek, hibás szabályozás, az előírások ismeretének hiánya miatt, az információk gyanútlan kiadása.

- Szándékos hibás viselkedés: az előírások megsértése, fenyegetés, zsarolás, megvesztegetés, haszonszerzési célból, bosszú, frusztráció miatt.

19.4. Támadástípusok

- Hozzáférés megszerzése.
- Jogosultság kiterjesztése.
- Szolgáltatásbénító támadások (DoS)
- Elosztott szolgáltatásbénítás (DdoS)
- Hamis megszemélyesítés, jogosultság szerzése.
- Sebezhetőségek kihasználása.
- Hálózati eszközök támadása.

19.5. A leggyakrabban előforduló támadások ismertetése:

- **Szolgáltatásmegtagadás-típusú támadások** (Dean of Service, DoS): a szolgáltatásmegtagadás típusú támadások lényege, hogy egy rosszindulatú személy olyasmit tesz a hálózattal vagy a kiszolgálóval, ami zavarja a rendszer működését, lehetetlenné teszi a munkavégzést. (Pl. elárasztjuk a gépet pingekkel, ekkor nem marad ideje más, hasznos tevékenységekre.) Egy ilyen támadásból nem sok haszna lehet egy kalóznak, betörni nem tud a rendszerbe, csak épp működésképtelenné teheti azt.
- **Trójai falovak**: Olyan kártékony program, amelyet alkalmazásnak, játéknak, szolgáltatásnak, vagy más egyéb tevékenység mögé rejtenek, álcáznak. Futtatásakor fejti ki károkozó hatását. Ugyanolyan jogosultságokkal rendelkezik, mint az őt futtató felhasználó.
- **Lehallgatott átvitel**: az azonosítási folyamat nyitva áll a lehallgatásra, amit megtehet bárki, pl. 1 hálózatfigyelő progi használatával.

19.6. Kriptográfiai módszerek és eszközök, azok gyakorlati alkalmazásai.

A **kriptológia** az adatok, üzenetek ***rejtjelezésével*** (kódolás, sifrírozás) és megoldásával (rejtjelfejtés, dekódolás, desifrírozás) foglalkozó tudományága, a matematikai tudományok egyik részterülete. A kriptológia egyik fő területe a kriptográfia, magyarul a rejtjelezés, amelynek alapvető feladata matematikai módszereket alkalmazó algoritmusokkal és azok használatának pontos leírását tartalmazó – szigorúan betartandó – kriptográfiai protokollok segítségével biztosítani az üzenetek, illetve tárolt információk bizalmasságát, védettségét, hitelességét. A kriptológia másik tudományága a kriptoanalízis (kriptográfiai bevizsgálás), amely a rejtjeles üzenet birtokában, de az eljárás teljes ismerete nélküli megfejtéssel (feltörésére) irányuló eljárásokkal foglalkozik. A kriptoanalízis főként matematikai módszereket használ. A rejtjelezett kommunikáció folyamatában a küld és a fogadó üzenetváltása történik meg. A küldő a nyílt szövegből rejtjelezés segítségével rejtjelezett szöveget állít elő, majd elküldi a vevőnek, aki azt visszafejtve megkapja az eredeti nyílt szöveget. A rejtjelezési folyamat – kódolás – során a rejtjelezett szöveg előállításához az algoritmuson kívül általában szükséges egy kulcs is, amelynek ismerete elengedhetetlen a rejtjelezésnél és a visszafejtésnél is.

19.6.1 Kriptográfiai módszerek

- **Behelyettesítés**: a nyílt szöveg minden karakteréhez valamilyen algoritmus szerint hozzárendelünk egy vagy több karaktert
- **Keverés**: a nyílt szöveg karakterei változatlanok maradnak, de sorrendjük megváltozik

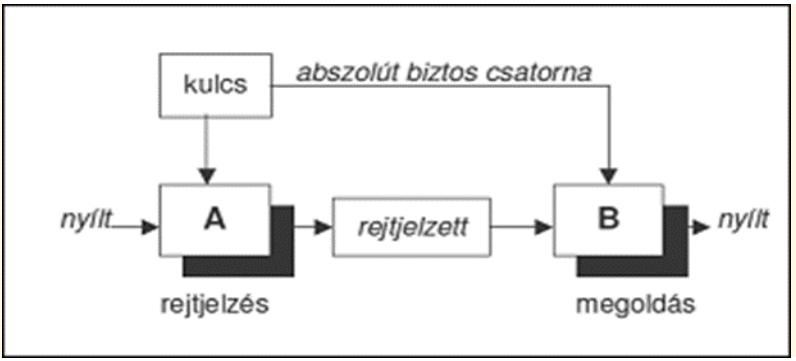
Akár kriptográfiai titkosításról, akár hitelesítésről (pl. aláírásról) van szó, az információt először ***kódolnunk***, majd később, amikor fel szeretnénk használni, ***dekódolnunk*** kell (azaz vissza kell fejtenünk).

Mind a kódolás, mind annak visszafejtése (dekódolás) általában valamilyen kriptográfiai kulcs segítésével történik. E megoldások biztonsága arra épül, hogy a dekódoláshoz, illetve a hitelesítéshez szükséges kriptográfiai kulcsok kizárólag a jogosult felek birtokában vannak.

A kulcs egy szám, bitsorozat, amely meghatározott hosszúságú lehet. E hosszúságot nevezzük kulcsméretnek, amelye alapvetően meghatározza a kódolás biztonságát. Amikor a kulcsot létrehozzuk (ezt nevezzük kulcsgenerálásnak, az adott hosszúságú bitsorozatok közül – valamilyen véletlent is használó módszerrel – kiválasztunk egyet.

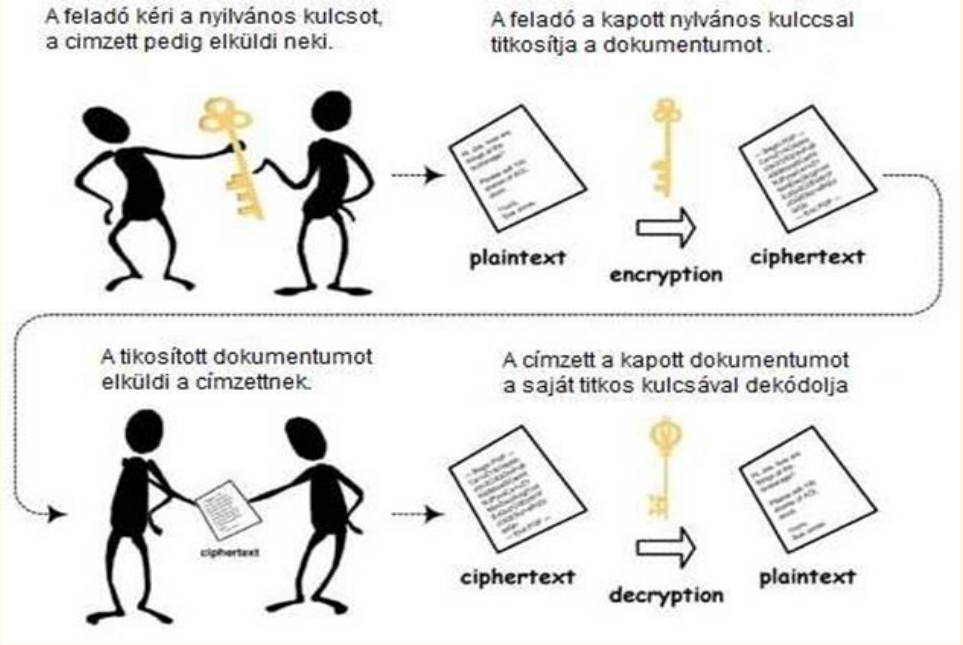
A támadó, aki nem ismeri a dekódoláshoz vagy hitelesítéshez használt kulcsot, megpróbálhatja kitalálni a kulcsot, ezt nevezzük a kódolás megtörésének. Ennek egyik módja, hogy egy számítógép segítségével az összes lehetséges (adott hosszúságú) kulcsot kipróbálja. Ha a kulcs elég hosszú, akkor ez nagyon nehéz feladat is lehet.

- **Szimmetrikus kulcsú kriptográfia**: A szimmetrikus kulcsú titkosítás során a dokumentumot ugyanazzal a kulccsal kódoljuk, amivel majd dekódolni is lehet; a kódolásra és dekódolásra használt kulcsot ekkor titkokban kell tartanunk, ezért titkos kulcsnak is nevezzük. E módszer hátránya, hogy a titkos kulcsot biztonságos módon kell eljuttatni a fogadó fél számára, hogy illetéktelen fél ne ismerhesse meg. Ez bizonyos esetekben nagyon nehéz problémát jelenthet. A szimmetrikus kulcsú titkosítás előnye, hogy az ilyen megoldások gyorsak és rövid kulcsokkal (pl. 128 bit vagy 256 bit) dolgoznak, és elég sok szimmetrikus kulcsú algoritmust ismerünk.



55. ábra – Szimmetrikus kulcsú kriptográfia

- **Szimmetrikus kulcsú titkosító-algoritmusok** például a következők: DES (már nem használatos), 3DES, AES stb.
- **Aszimmetrikus kulcsú (nyilvános kulcsú) kriptográfia**: Az aszimmetrikus kulcsú (más néven nyilvános kulcsú) titkosításnál a kódolás és a dekódolás nem ugyanazzal a kulccsal történik.



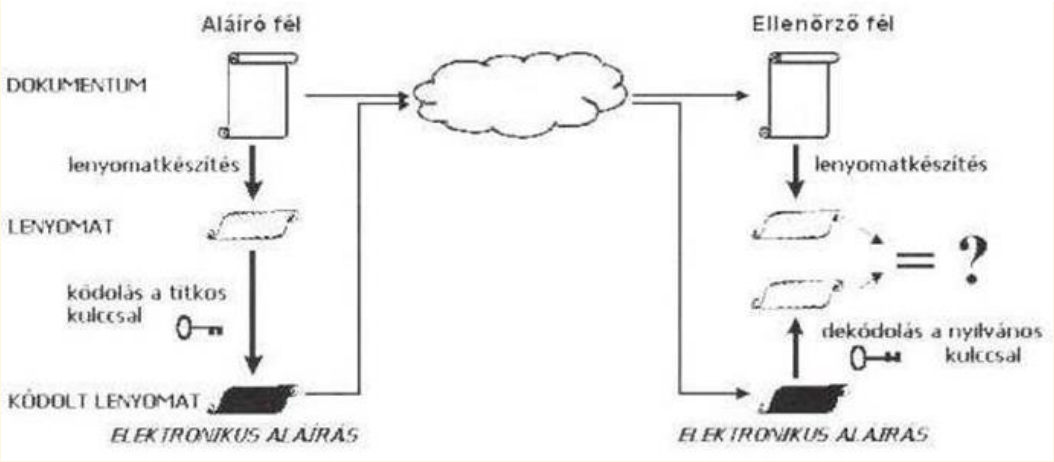
56. ábra – Aszimmetrikus kulcsú (nyilvános kulcsú) kriptográfia

Minden félnek van egy nyilvános kulcsa és egy magánkulcsa. Az egyik kulccsal kódolt üzenetet csak a hozzá tartozó másik kulccsal lehet dekódolni. A magánkulcs soha nem kerül ki birtokosa tulajdonából, de bárki hozzáférhet mások nyilvános kulcsához. A nyilvános kulcsot nem kell titokban tartani, azt bárki megismerheti. Ha titkosított üzenetet szeretnénk küldeni valakinek, meg kell szereznünk az ő nyilvános kulcsát, és azzal kell kódolnunk a neki szóló üzeneteket. Az így kódolt üzeneteket a címzett a saját magánkulcsával fejtheti vissza. A nyilvános kulcsú kriptográfia más módon is használható: ha a saját magánkulcsunkkal kódolunk egy dokumentumot, az így kapott adatról – a nyilvános kulcsunk alapján – bárki megállapíthatja, hogy azt mi hoztuk létre. E műveletet aláírásnak nevezzük.

- **Elektronikus aláírás**: Digitális aláírásnak olyan elektronikus karakter-sorozatot neveznek, amelyet igen nagy valószínűséggel ***csak az aláírótól származhat***. A digitális aláírás tartalmazza az üzenet egyirányú képét (lenyomatát), s egyéb adatokat, például keltezést (dátumot, pontos időpontot), sorszámot, a küldött üzenetből képzett ellenőrző számot. Az aláírás jellemző a létrehozójára és az üzenetre egyaránt. Az elektronikus aláírást bárki ellenőrizni tudja, aki a megfelelő infrastruktúrához hozzáfér. A digitális aláírás két részből áll:
  - A személyhez kötött aláírást generáló részből, s az
  - ellenőrzést bárki számára lehetővé tevő részből.

  A digitális aláírás elkészítéséhez először kiegészítjük a dokumentumot a megfelelő azonosítókkal, majd ennek a kiegészített dokumentumnak egy alkalmas sűrítményét készítjük el. Ez lesz a digitális aláírás. Az alkalmas sűrítmények elkészítésére szolgálnak az úgy nevezett Hash-eljárások. A Hash-algoritmus egy olyan transzformáció, amely egy tetszőleges hosszú szöveg fix hosszúságú digitális sűrítményét

készíti el, amely kizárólag az adott szövegre jellemző. Tulajdonságai: hamisíthatatlan, nem használható fel újra, letagadhatatlan, megváltoztathatatlan.



57. ábra – Elektronikus aláírás

- **Kulcskezelés**, PKI, CA: A nyilvános kulcsú rendszerben fontos tudni, hogy a nyilvános kulcs tulajdonosa valóban az a személy, akinek a levelet szánjuk. A digitális aláírást bárki létrehozhatja, ezért valakinek tanúsítani kell, hogy valóban az az aláíró, akinek vallja magát. Ennek valódiságát egyrészt az alkalmazott digitális aláírások biztosítják, másrészt különféle, úgy nevezett biztonsági modellek segítségével. A legbiztosabb megoldás a direkt biztonsági modell, amelyben – mint a neve is mutatja – a vevő személyesen adja át nyilvános kulcsát az adónak. Ez a valóságban – a fizikailag nagy távolságok miatt – a legtöbbször kivihetetlen, ezért széles körben a hierarchikus biztonsági modell alapján kiépített Hitelesítésszolgáltatón, vagy közismert nevén a ***Certificate Authority***-n (CA) alapuló rendszer terjedt el a gyakorlatban. A résztvevők által megbízhatónak tekintett harmadik fél egy digitális közjegyző szerepét játssza. Olyan szakoso-

dott szervezet vagy cég, amely tanúsítványokat adhat ki kliensek és szerverek számára. A CA igazolja, hogy egy adott azonosítóval rendelkező felhasználó az, akinek vallja magát. A nemzetközi feltételeket, szabványokat kielégítő infrastruktúrát magyarul is az angol ***Public Key Infrastructure*** (Nyilvános Kulcsú Infrastruktúra) kifejezésből származó *PKI* rövidítés jelöli.

20. Modellező nyelvek és eszközök szerepe az alkalmazások tervezésében és dokumentálásában. UML diagramok: használati eset diagram, objektumdiagram, kommunikációs diagram, állapot diagram, osztálydiagram és osztályleírás, komponens diagram

20.1. Modellező nyelvek és eszközök szerepe az alkalmazások tervezésében és dokumentálásában

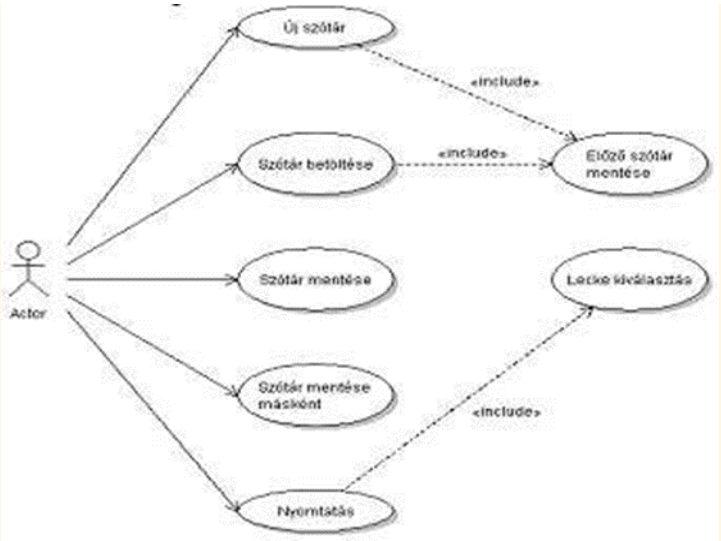
Előzmények: A szoftverek bonyolultságának, komplexitásának növekedésével és az objektumorientált nyelvek megjelenésével vált szükségessé egy mindenki által használható tervezési módszer kialakítása, elemzési módszerek, egységes nyelvek, jelölésrendszerek kidolgozása. A '90-es évek elejéig több javaslat is született, ezek többsége azonban nem felelt meg az igényeknek. A nagy áttörést 1997-ben az **UML** modellezőnyelv, illetve 1998-ban a **RUP** hozta. Az UML egy ***szabványos, egységesített modellezőnyelv***, amelynek segítségével a tervezés, a specifikáció, a dokumentálás mind grafikus formában, beszédes ábrák, diagramok, táblázatok segítségével végezhető. Olyan módszer, amely képes kezelni ezt a sokrétűséget, viszont kellően egyszerű ahhoz, hogy széles körben elterjedjen. A tervezési folyamatban az UML-t a kiindulási fázisban használjuk, azzal a céllal, hogy minél biztonságosabban, áttekinthetőbben és megbízhatóbban készítsük el a teljes szoftver tervét, az objektumorientált fejlesztési elvhez kapcsolódóan. Az UML nem más, mint egy tervezési nyelv, ami egy szoftver rendszer minél megalapozottabb kidolgozásának előkészítésének folyamatát szolgálja. Az igy készült tervezési diagramok alapján válik lehetővé a forráskód megírása és a futtatható szoftver alkészítése, valamint segítségével a későbbiekben is könnyebben módosíthatjuk a programot.

20.2. (Legfőbb) UML diagramok

- használatieset-diagram,
- objektumdiagram,
- kommunikációs diagram,
- állapot-diagram,
- osztálydiagram és osztályleírás,
- komponensdiagram.

20.2.1 Használatieset-diagram (Use-Case)

A HE-diagram a rendszer viselkedésének egy *kiragadott részét* írja le külső *aktorok szemszögéből*. A HE diagram elemei: ***aktorok***, ***használati esetek***, valamint ***az ezek közötti kapcsolatok***. Egy aktor üzeneteken keresztül kommunikál a rendszerrel. Az aktor üzenetet küld a rendszernek, a rendszer pedig a használati eset végrehajtása közben üzeneteket küldhet vissza az aktor számára.



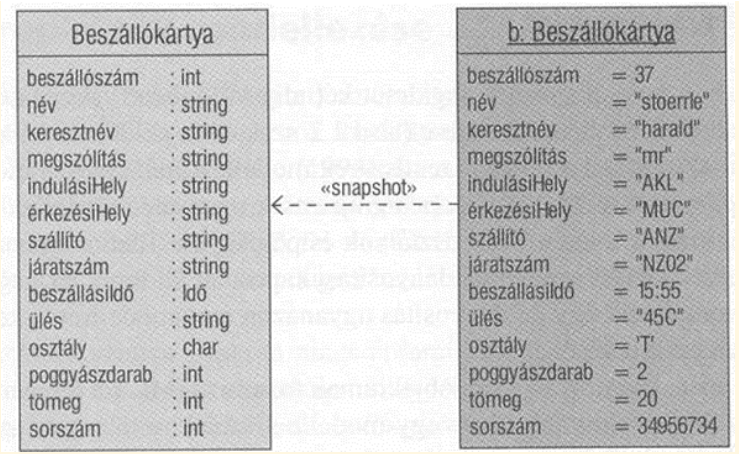
58. ábra – Használatieset-diagram

Az aktor üzeneteinek és a rendszer válaszainak megadásával a rendszer határait húzzuk meg. A használati eset a szoftver használatának

egy értelmes egysége, az aktor kommunikációja, párbeszédje a szoftverrel. A használati eset a rendszer viselkedését írja le *a rendszeren kívülről*. ***Felhasználói célnak*** nevezzük a felhasználónak *azt a célját, melyet a szoftver használatával szeretne elérni*. A felhasználói célok *használati esetekre* bontandók: a cél elérése, illetve a feladat megoldása érdekében a felhasználó *konkrét használati eseteket* hajt végre. A használati esetek a szoftverbe előre beépített lehetőségek, melyeket a felhasználó (aktor) indítványozhat. Egy forgatókönyv (eseményfolyam) a használati eset egy konkrét végrehajtása (példánya). Egy használati esetnek elvileg számtalan forgatókönyve lehetséges. A használatieset-modell a teljes rendszer viselkedésének a leírása. A HE modell alapján a felhasználó érti, hogyan kell használni a szoftvert. A fejlesztés HE-centrikus; a használati esetek a teljes fejlesztés során központi szerepet játszanak. A ***HE-modell*** elemei: *HE-diagramok és leírások*.

20.2.2 Objektumdiagramok

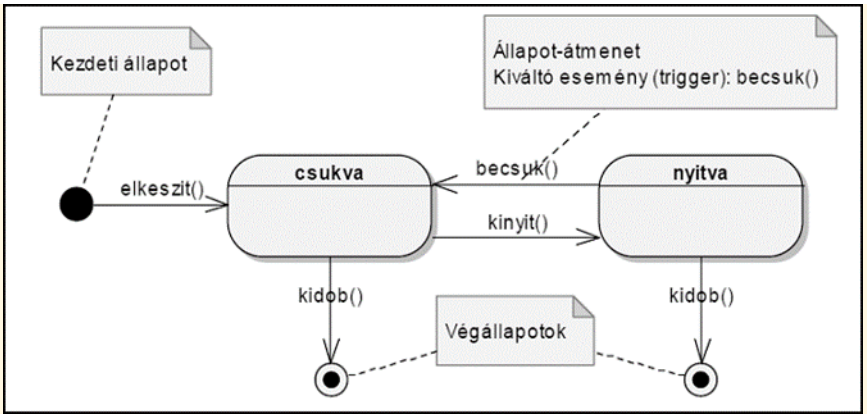
A modellezett rendszer ***egy adott időpillanat-beli állapotát*** mutatják az objektumdiagramok. Az objektumdiagram pillanatfelvétel a rendszer állapotáról. Osztályok *példányait és kapcsolatait* jeleníti meg. Az objektumdiagram konkrétabb az osztálydiagramnál, mert objektumok példányainak a kapcsolatát írja le objektumosztályok kapcsolata helyett.



59. ábra – Objektumdiagram

20.2.3 Állapotdiagram

Az állapot-átmeneti diagram (state transition diagram) egyetlen osztály (annak egy előfordulásának) dinamikus viselkedését, a külvilággal való kapcsolatát ábrázolja.



60. ábra – Állapotdiagram

Az állapot-átmeneti diagram egy gráf, melynek csomópontjai állapotok, élei pedig átmenetek. Megadja, hogy az objektum mely események hatására milyen állapotból milyen állapotba kerül. Egy adott állapotban levő objektum ugyanarra az eseményre mindig ugyanúgy

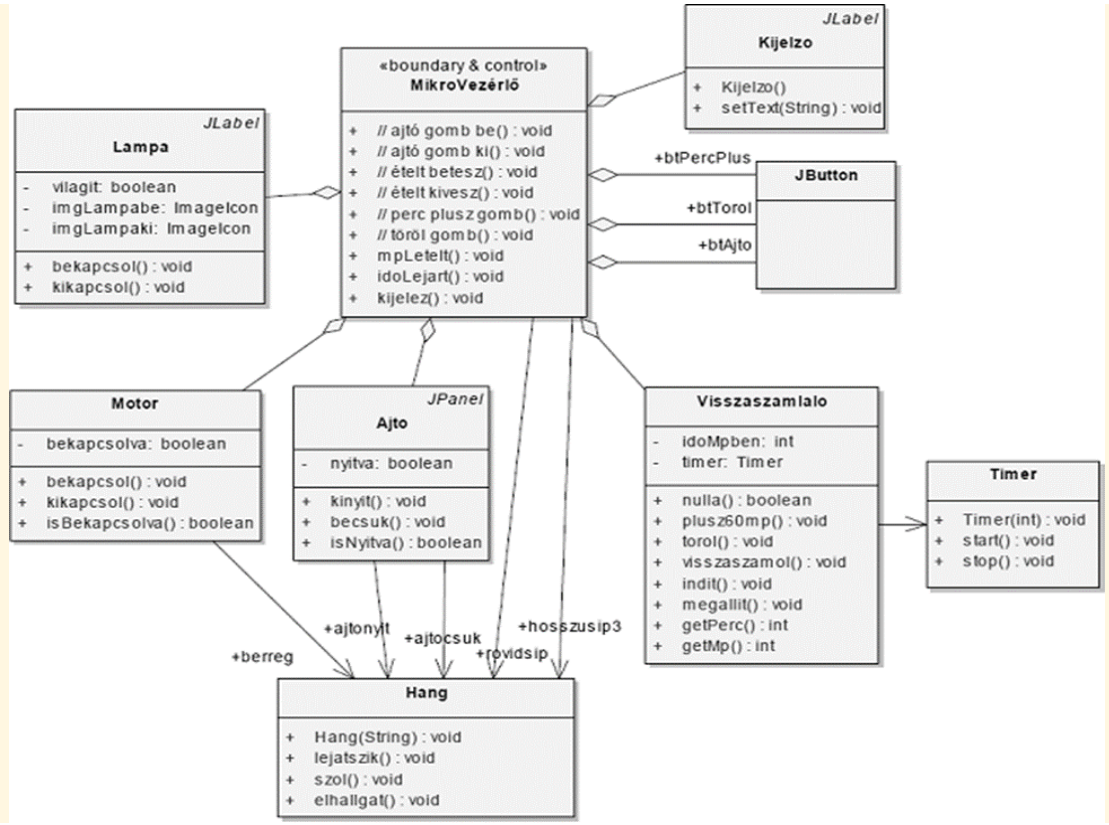
reagál (ugyanazt az akció sorozatot hajtja végre). Az állapot az objektum életének egy szakaszát írja le. Az állapot jele az UML-ben egy lekerekített téglalap, melynek részei (bármelyik rész elhagyható): Állapot-átmenetnek nevezzük azt a folyamatot, melyben az objektum egy adott állapotából egy egyértelműen megkülönböztethető másik állapotba kerül. Az átmenet lehetséges tulajdonságai:

- **Kiváltó esemény** (*trigger* vagy event): Az átmenet a kiváltó esemény hatására következik be. Az esemény egyaránt jöhet kívülről vagy belülről.
- **Őrfeltétel** (*guard*): Az őrfeltétel egy logikai kifejezés, amely hivatkozhat az objektum adataira. Az átmenet csak akkor következik be, *ha az őrfeltétel igaz*.
- **Akció** (action): Átmenetkor végrehajtódik.

Az Ajtó osztályból hozzunk létre egy példányt. Életét egy állapot-átmeneti diagrammal szemléltetjük. Születéskor –, amikor elkészítik – csukva van. Aztán egész életében nyitják-csukják. Az ajtót ki lehet dobni akár nyitott, akár csukott állapotban.

20.2.4 Osztálydiagram (class diagram)

Olyan diagram, amely az osztályokat és a közöttük lévő társítási és öröklési kapcsolatokat ábrázolja. Az objektumdiagram az osztálydiagram elő fordulása, példánya. Az osztálydiagram rögzíti az objektumok közötti kapcsolatok szabályait. Két osztály közötti társítási kapcsolat főbb jellemzői: ismeretségi vagy tartalmazási kapcsolat (ha tartalmazási kapcsolat, akkor erős vagy gyenge); ***multiplicitás*** (egy-egy, egy-sok vagy sok-sok jellegű; kötelező vagy opcionális); szerepnév; megszorítás.

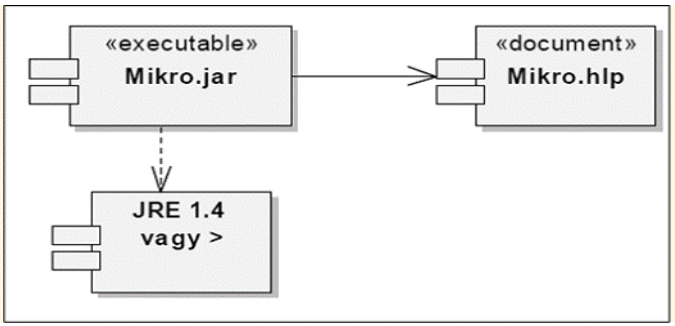


61. ábra – Osztálydiagram

20.2.5 Komponensdiagram

A komponensdiagram a rendszert alkotó fizikai komponenseket (szoftverelemeket) és az azok közti kapcsolatokat ábrázolja. A komponensdiagramon megadható a logikai nézet osztályainak forráskomponensekhez való hozza rendelése, valamint a forráskódok hozza rendelése futtatható komponensekhez. A komponensdiagram az implementációs nézet jellemző diagramja. A komponens (component) egy fizikailag bonthatatlan szoftver egység. A komponens lehet egy állomány, például forráskód, szerkesztendő vagy futtatható szoftver elem: lefordított tárgykód, bájtkód, futtatható program vagy dinamikus

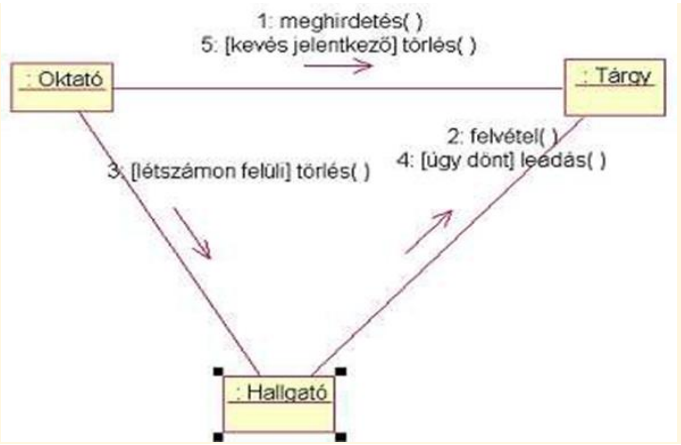
könyvtár (DLL). Mint más diagramelemek, a komponensek is csomagokba csoportosíthatók.



62. ábra – Komponensdiagram

20.2.6 Együttműködési diagram – Kommunikációs diagram

Az együttműködési diagram (collaboration diagram) ***az üzeneteket küldő és fogadó objektumok kapcsolatát*** és a közöttük lezajló ***üzenetváltás*** strukturális szerkezetét ábrázolja.



63. ábra – Együttműködési (kommunikációs) diagram

21. Szoftverfejlesztési módszer és módszertan. A vízesés módszer összehasonlítása az inkrementális és iterációs módszerekkel

A szoftverfejlesztések célja, hogy ***eladható szoftvertermékeket*** állítsanak elő. A szoftvertermékeknek e tekintetben két nagyobb csoportja létezik:

- **Általános szoftvertermékek**. Ezek olyan általános szoftverek, amelyeket egy fejlesztő szervezet készít, és amelyeket a szoftverpiacon árulnak. Bárki megvásárolhatja és a licensznek megfelelően használhatja. Az általános termékek esetében a szoftverspecifikációt az a fejlesztő cég dolgozza ki és menedzseli, amelyik a terméket fejleszti.
- **Egyedi szoftvertermékek**. Az ilyen szoftverek egyéni megrendelők megbízásai alapján készülnek. A szoftver szállítója speciálisan a megrendelő igényei alapján fejleszti a szoftvert megegyezett áron, szerződés alapján. Az egyedi szoftvereket nem árulják a szoftverpiacon. Ebben az esetben a megrendelő cég határozza meg a szoftverkövetelményeket és a legtöbbször ő dolgozza ki a szoftverspecifikációt.

21.1. Szoftverfolyamat

A szoftverfolyamat, vagy más néven **a szoftverfejlesztés életciklusa** meghatározott fejlesztési tevékenységek és kapcsolódó eredmények, termékek együttese, amelynek a végeredménye a *kész* szoftvertermék.

- **Szoftverspecifikáció**: A szoftver funkcionális és nem-funkcionális működését meghatározó követelményrendszer kidolgozása.
- **Szoftverfejlesztés, implementáció**: A szoftver elkészítése a specifikáció szerint.

- **Szoftver minőségbiztosítása (V&V)**: A szoftver verifikációja és validációja, avagy minőségbiztosítási (*quality control*) folyamatok összessége a
  - **Verifikáció (igazolási eljárás):** Ellenőrzés, mely arra irányul, hogy a szoftvert a megadott funkcionális és nem funkcionális követelményeknek, különféle feltételeknek megfelelően valósították-e meg.
  - **Validáció (szakértényesítés)**: Annak megvizsgálása, hogy a szoftver jelen állapotában beilleszthető-e, megfelel-e a kívánt elvárásoknak (azaz jól specifikáltuk-e eredetileg a követelményeket).
- **Szoftverevolúció**: A szoftver továbbfejlesztése a megrendelő változó követelményeinek megfelelően.

21.2. A szoftverfejlesztési módszerek általános módszertani modelljei

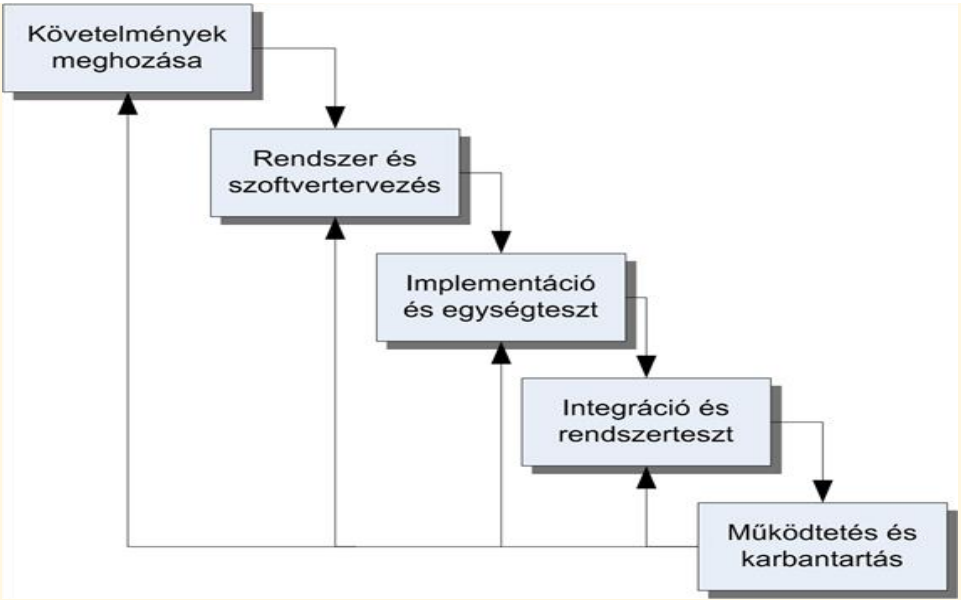
A legtöbb szoftverfolyamat-, szoftverfejlesztési módszer az alábbi **három általános módszertani modell** valamelyikére épül.

21.2.1 A vízesés-modell

Ez a folyamatmodell a fejlesztés alapvető tevékenységeit a folyamat különálló, és szigorúan egymást követő fázisaiként reprezentálja, mint például szoftverspecifikáció, szoftvertervezés, implementáció, tesztelés stb. Miután egy fázis befejeződött, a fejlesztés a következő fázisban folytatódik tovább. A modell alapvető szakaszai alapvető fejlesztési tevékenységekre képezhetők le. Ezek:

- **Követelmények elemzése és meghozása**: a rendszer szolgáltatásai, megszorításai és célja a rendszer felhasználóival történő konzultáció alapján alakul ki. Ezeket később részletesen kifejtik, és ezek szolgáltatják a rendszer specifikációt.

- **Rendszer- és szoftvertervezés**: a rendszer tervezési folyamatában választódnak szét a hardver- és szoftverkövetelmények. Itt kell kialakítani a rendszer átfogó architektúráját. A szoftver tervezése az alapvető szoftverrendszer-absztrakciók, illetve a közöttük levő kapcsolatok azonosítását és leírását is magában foglalja.
- **Implementáció és egységteszt**: ebben a szakaszban a szoftverterv programok, illetve programegységek halmazaként realizálódik. Az egységteszt azt ellenőrzi, hogy minden egység megfelel-e a specifikációjának.
- **Integráció és rendszerteszt**: megtörténik a különálló programegységek, illetve programok integrálása és teljes rendszerként való tesztelése, hogy a rendszer megfelel-e a követelményeknek. A tesztelés után a szoftverrendszer átadható az ügyfélnek.



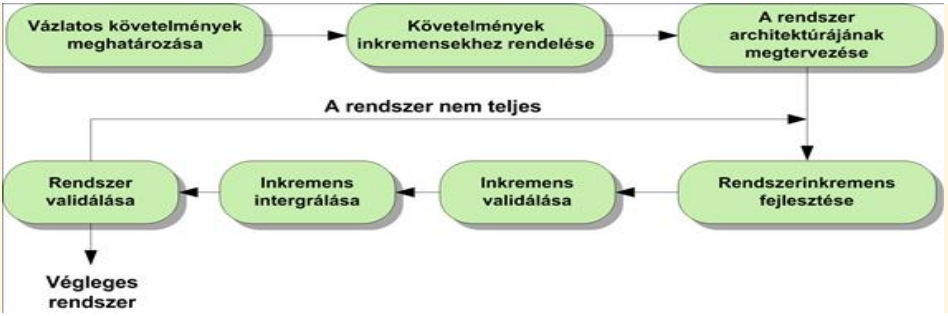
64. ábra – Vízesés-modell

- **Működtetés és karbantartás**: általában ez a szoftver életciklusának leghosszabb fázisa. Megtörtént a rendszertelepítés és megtörtént a rendszer gyakorlati használatbavétele. A karbantartásba beletartozik az olyan hibák javítása, amelyekre nem derült fény az életciklus korábbi szakaszaiban, a rendszeregységek implementációjának továbbfejlesztése, valamint a rendszer szolgáltatásainak továbbfejlesztése a felmerülő új követelményeknek megfelelően.

21.2.2 Iteratív fejlesztés

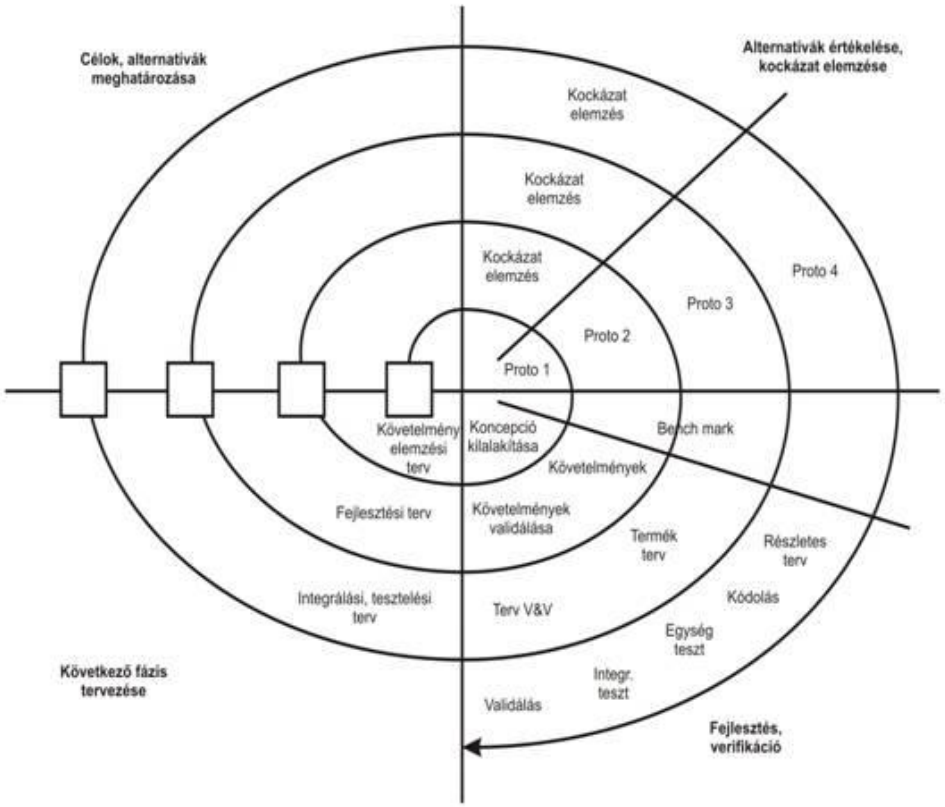
A fejlesztés egy kezdetleges specifikációból indul ki és egy ennek megfelelő kezdeti rendszer kerül gyorsan kifejlesztésre. Ez lesz a kiindulórendszer a megrendelő kívánságainak eleget tevő rendszer kifejlesztéséhez. A rendszer specifikációjának további finomításával egymást követik a *specifikáció*, a *fejlesztés* és a *verifikálás-validálás* újabb ciklusai mint ***iterációk***, amelynek a végén kialakul a kívánt funkciókkal rendelkező rendszer. A folyamatiteráció támogatására több modell is kidolgozásra került. A két legismertebbet említve:

- **Inkrementális fejlesztés**: a *szoftverspecifikáció*, a *tervezés*, az *implementálás* kis ***inkrementációs lépésekre*** van felosztva. Egy ***köztes*** *megközelítés* a *vízesésmodell* és az *evolúciós fejlesztési modellek* (egy kezdeti implementációt a felhasználókkal véleményeztetnek, majd sok-sok verzión keresztül addig finomítják, amíg nem lesz megfelelő a rendszer) között.



65. ábra – Inkrementális fejlesztési modell

- **Spirális fejlesztés** (Boehm): A (***teljes***) fejlesztési folyamat egy ***belülről kifelé tartó spirálvonalat*** követ. Jellemzői:
  - A spirálmodell ***iterációkból*** *áll*, melyek *folyamatosan ismétlődnek* a projekt során.
  - Valamennyi iteráció *ugyanazon* lépésekből áll.
  - Lehetővé teszi a ***kockázatok*** *korai felismerését*.
  - A megrendelőt minden fázisba aktívan bevonja.
  - A modell elég *komplex*, megértése nem egyszerű.
  - Jelentős *kockázatkezelési szakértelem* szükséges.
  - A nagyszámú köztes iteráció miatt sok, végül ***feleslegessé*** *váló dokumentáció* születhet.
  - A spirál minden egyes ciklusát ***négy fő szektorra*** oszthatjuk fel:
    - **Célok kijelölése**: az adott projektfázis által kitűzött célok meghatározása. Azonosítani kell a folyamat megszorításait, a terméket, fel kell vázolni a kapcsolódó menedzselési tervet. Fel kell ismerni a projekt kockázati tényezőit, és azoktól függően alternatív stratégiákat kell tervezni, ha lehetséges.
    - **Kockázat becslése**: minden egyes felismert kockázati tényező esetén részletes elemzésre kerül sor. Lépéseket kell tenni a kockázat csökkentése érdekében.
    - **Fejlesztés és validálás**: a kockázat kiértékelése után egy fejlesztési modellt kell választani a problémának megfelelően. Pl. evolúciós, vízesés stb. modellek.
    - **Tervezés**: A folyamat azon fázisa, amikor dönteni kell arról, hogy *folytatódjon-e* egy következő ciklussal vagy sem. Ha a folytatás mellett döntünk, akkor fel kell vázolni a projekt következő fázisát.



66. ábra – Spiráll-modell

21.3. Komponensalapú szoftvertervezés

Ezekben a fejlesztésekben feltételezik, hogy a rendszer részei, komponensei már *korábban kifejlesztésre kerültek* vagy a szoftverpiacon megvásárolhatóak. Ebben az esetben a fejlesztési folyamat azon tevékenységeire helyeznek nagyobb hangsúlyt, amely ***ezen részek integrálásával*** foglalkozik. Ez a fejlesztési forma manapság az egyik legelterjedtebb, köszönhetően annak, hogy *a továbbfejlesztés költséghatékonyabb*, mint egy új rendszer fejlesztése.

21.4. Szoftverfejlesztési módszertanok felsorolása

- Agilis egységes folyamat (AUP)
- Építőjellegű design módszertan (CDM)
- Dinamikus rendszerfejlesztés (DSDM)
- Extrém programozás (XP)
- Iteratív és fokozatos fejlesztés
- Kanban
- Lean szoftverfejlesztés
- Nyílt egységes fejlesztés (OpenUP)
- Páros programozás
- Gyors alkalmazásfejlesztés (RAD)
- Egységesített racionális fejlesztés (RUP)
- Scrum
- SSADM (Structured Systems Analysis and Design Method)
- Egységes folyamat (UP)

Ábrajegyzék