

Tartalom

1. TÉTEL.....	4
Adattípusok.....	4
Deklaráció.....	5
Feltételes utasítások.....	5
Adat és információ	5
Entrópia és fajtái	6
Kifejezések infix és postfix alakja	6
Keresési problémák állapottér-reprezentációja	6
Neminformált keresési eljárások	7
2. TÉTEL.....	8
Ciklusszervezési lehetőségek.....	8
Függvénykezelés, paraméter-kiértékelés	8
Hatáskörkezelés.....	9
Számrendszerek, számábrázolás.....	9
Karakter, szöveg és logikai adat ábrázolása.....	10
Heurisztika fogalma.....	10
A* algoritmus.....	10
Kétszemélyes, teljes információjú, determinisztikus játékok.....	11
A stratégia fogalma, minimax-algoritmus.....	11
Minimax-algoritmus.....	11
Alfa-béta vágás	11
3. TÉTEL.....	12
Sztring létrehozása és inicializálása.....	13
CNF: Chomsky-féle normálalak.....	14
CYK: Cocke-Younger-Kasami algoritmus.....	14
Hálózatok	14
Elemi adatkapcsolati protokollok:.....	15
A LAN hálózat elemei.....	15
HTML.....	16
4. TÉTEL.....	17

Funkcionális követelmények.....	17
PR. Tétel: A lineáris keresés tétele	17
PR. Tétel: A logaritmikus keresés tétele	17
PR. Tétel: Az eldöntés tétele.....	18
PR. Tétel: A kiválogatás tétele	18
Buborékos rendezés	19
Relációs adatmodell	19
Turing-gépek.....	20
Church-tézis	21
Megállási probléma	21
Logikai függvények megadása	22
KNF, DNF	22
Logikai hálózatok	22
5. TÉTEL.....	23
Adatszerkezetek	23
Láncolt lista.....	23
Bináris fa műveletei:	24
Operációs rendszerek folyamatai memóriakezelése	24
Állománykezelés	25
Üresszó lemma	26
Automata.....	26
6. TÉTEL.....	27
OOP (objektum-orientált programozás).....	27
Típusok és konverziók	27
Operátorok	28
Utasítások	29
Láthatóság	30
Ábécé, szó, nyelv, nyelvtan fogalma.....	30
Chomsky-féle nyelvtani osztályok és az általuk generált nyelvosztályok tartalmazási hierarchiája.....	31
Logikai áramkörök.....	31

Kombinációs logikai hálózatok.....	32
7. TÉTEL.....	32
Öröklődés, túlterhelés, polimorfizmus.....	32
Kivételkezelés	33
Nézettáblák.....	33
Indexelés.....	34
Az adatbázis-tervezés elmélete.....	34
E/K modell és átfordítása adatmodellé.....	36
Nyomkövetés és hibakeresés, egységtesztelés, naplózás.....	36
Kiírás.....	36
Nyomkövetés	36
Adatok nyomkövetése	37
Töréspontok elhelyezése	37
Lépésenkénti végrehajtás	37
Egységtesztelés.....	37
Naplózás	37
Kollekciók használata.....	37
Relációs adatbázisok kezelése OO programozási nyelvekben	38
8. TÉTEL.....	38
Osztály- és példány inicializálás	38
Konstruktor.....	39
Interfészek	39
Generikus programozás	40
Összetett adatszerkezeteket implementáló osztályok és fontosabb műveleteik.....	40
Rendszerfejlesztési modellek (tervezés,tesztelés)	41
Vízesés modell.....	41
Evolúciós modell.....	42
Feltáró fejlesztés.....	43
Formális modell	43
Újra felhasználás-orientált modell	43
Inkrementális modell:	44

Spirális modell.....	45
UML osztálydiagram.....	46
Verziókezelés	47
Számítógép-architektúrák – mikroelektronika, processzor és memória	47
CPU főbb részei:.....	48
9. TÉTEL.....	48
Adatdefiníciós résznyelv (DDL)	48
Adatlekérdező nyelv (DQL).....	49
Adatmanipulációs nyelv (DML).....	50
Adatvezérlő nyelv (DCL).....	50
Tervezési minták egy OO programozási nyelvben.....	51
MVC.....	53
Topológiák és architektúrák.....	53
OSI modell.....	54
Fizikai átviteli jellemzők és módszerek, közeg hozzáférési módszerek.....	54

1.TÉTEL

Adattípusok

A C – Programozási nyelv **elemi adattípusai** az **int**, **char**, **float** és **double**. Az adattípusok elé tehetünk minősítő jelzőket, amelyek a tárolható adat méretét és értékhatárát változtatja meg.

Egész adattípusok:

Type	Storage size	Value range
char	1 byte	-128 to 127 or 0 to 255
unsigned char	1 byte	0 to 255
signed char	1 byte	-128 to 127
int	2 or 4 bytes	-32,768 to 32,767 or -2,147,483,648 to 2,147,483,647
unsigned int	2 or 4 bytes	0 to 65,535 or 0 to 4,294,967,295
short	2 bytes	-32,768 to 32,767
unsigned short	2 bytes	0 to 65,535
long	8 bytes	-9223372036854775808 to 9223372036854775807
unsigned long	8 bytes	0 to 18446744073709551615

Lebegőpontos adattípusok:

Type	Storage size	Value range	Precision
float	4 byte	1.2E-38 to 3.4E+38	6 decimal places
double	8 byte	2.3E-308 to 1.7E+308	15 decimal places
long double	10 byte	3.4E-4932 to 1.1E+4932	19 decimal places

A **signed** és **unsigned** minősítők csak az egész adattípusok esetén használhatók. Az **unsigned** minősítővel ellátott változók csak 0 vagy pozitív értéket vehetnek fel. Az **int** adattípus mérete megegyezik a használt számítógép alap szóméretével (16 vagy 32 bit). A **short** int adattípus 16 bites, a **long** pedig 64 bites hosszúságú. A **void** kulcsszó adattípus függvények kimeneti értékénél adható meg, ha NINCS értéke. Nem deklarálható változó vele.

Összetett adattípusok közé tartoznak a **tömbök** és **struktúrák**. A tömbök használatával azonos elemi típusú adatokból tárolhatunk többet egy változón belül. A struktúrák használatával többféle adattípusból álló egyedi típust tudunk létrehozni. Minden adattípushoz tartozik egy **pointer** típus (char*, int*) ami a memória címét tartalmazza a változóról.

Deklaráció

A **változókat** használat előtt **deklarálni** kell. A változók a deklaráció során kezdeti értéket is kaphatnak. Ha a nevet egyenlőségjel és egy kifejezés követi, akkor a kifejezés értéke lesz a kezdeti érték. A deklaráció során a rendszer **lefoglalja** a változó számára a típusnak megfelelő **memória** területet. Bármely változó deklarációjában alkalmazható a **const** minősítő, ami azt jelzi, hogy a változó értéke nem fog megváltozni a program futása során.

Feltételes utasítások

Az **if - else** feltételes utasítás használatával elágazásokat készíthetünk a programban. Szintaxis:

```
if ( feltétel kifejezés ) { utasítások } else (if) { utasítások }
```

Az **else ág opcionális**. Amikor a program if utasításba fut, először kiértékeli a feltétel kifejezést, ha annak az értéke igaz, akkor lefut a törzsében levő kifejezés, ha hamis akkor az **else** ág törzsében levő utasítások következnek, vagy annak hiányában folytatódik a program.

else – if

A **switch - case** utasítás egy többfelé elágazó utasítás. A switch utasítás többféle esetet (case) tartalmaz, amelyek állandó értékekkel rendelkeznek. A feltétel függvényében az a törzs fog lefutni, amelyikkel egyezik a kifejezés értéke. Egy switch utasításon belül bármennyi esetet tartalmazhat. Opcionálisan **default** ágot is használhatunk, amely akkor fut le, ha egyik esettel sem egyezett a kifejezés. Az ágak törzsét **break** utasítással zárhatjuk, ami kiugrik az elágazás törzséből.

Adat és információ

Az **adat** egy **elemi ismeret**. Olyan tények, hírek, amelyek alkalmasak az emberek vagy számítógépek által való értelmezésre. Az adat feldolgozása információt eredményezhet.

Az **információ** olyan ismeret, amely a fogadó fél számára korábbi ismeretek alapján értelmezhető, tehát az adat feldolgozásával olyan új ismeretet nyerünk, amellyel eddig nem rendelkezünk.

Entrópia és fajtái

Az **entrópia** egy jelsorozat információtartalmát fejezi ki. Az entrópia értéke 0 vagy nagyobb szám. Az entrópia akkor a legkisebb (0), ha a jelsorozat ugyan azt a jelet sugározza. Értéke akkor lesz maximális, ha valamennyi jel azonos valószínűséggel fordul elő.

FAJTÁI	
Maximális entrópia (H_{\max}):	Ha az egyes események bekövetkezési valószínűsége azonos.
Tényleges entrópia (H'):	
Relatív entrópia (H_{rel}):	Az entrópia és a maximális entrópia hányadosa. (H' / H_{\max})

Kifejezések infix és postfix alakja

A matematikában a műveletek leírására általában **infix** jelölést használunk. Az operátorokat az operandusok közé írjuk és a sorrendiség a megszokott módon történik, például a szorzás hamarabb következik, mint az összeadás. Zárójelezéssel módosítható az operátorok sorrendje.

infix példa: $2 * (2 + 1)$

Ha az operátorokat az operandusok után írjuk, akkor **postfix** formáról beszélünk. Postfix formában nem használunk zárójeleket. A sorrend meghatározza a műveletek sorrendjét.

postfix példa: $2\ 2\ 1\ +\ *$

Keresési problémák állapotter-reprezentációja

Az **állapotter-reprezentációja** egy probléma megoldásához szükséges tér, tulajdonságok és jellemzők modellezése. Az **állapotok halmaza** tartalmazza az összes lehetséges állapotot, amelyek előállhatnak a jellemzők kombinációjából.

A lehetséges állapotok közül meg kell adnunk egy speciális állapotot, mely a probléma világához tartozó jellemzők kezdőértékeit határozza meg. Ezt az állapotot **kezdőállapotnak** nevezzük.

A probléma elvégzéséhez meg kell határoznunk a **célállapotot**, amelyből akár többet is megadhatunk.

Az állapotok változtatásához meg kell adni a lehetséges műveleteket, cselekményeket. Ezeket nevezzük **operátoroknak**. Az operátorokhoz tartoznak költségek és megszorítások. Az **operátor költsége** megadja, hogy milyen költsége van az operátor elvégzésének. Az **operátor alkalmazási előfeltételek** pedig megszabják, hogy milyen állapotban használható az adott operátor.

Példa: Robotporszívó.

Állapottér:	Szobák, szobák tisztasága, porszívó helyzete.
Kezdőállapot:	Első szobában a porszívó, másik két szoba koszos.
Célállapot:	Minden szoba tiszta legyen és a porszívó legyen az első szobában.
Operátorok:	Porszívó mozgása, takarítás.
Költségek:	Energia fogyasztás.
Előfeltételek:	Legyen elég energia takarítás után, hogy visszatérjen az állomásra.

Neminformált keresési eljárások

A **nem informált keresési eljárásokat** olyan problémák esetén használjuk, amelyeknél semmilyen információnk nincs az állapotokról. Működésük során új állapotokat generálnak a megadott operátorok elvégzésével, amelyek egy keresési fát alkotnak.

A kereső algoritmusokat a következő tulajdonságokkal jellemezzük:

- Teljesség:** A rendszer minden olyan esetben megtalálja-e a megoldást, ha az létezik?
- Optimalitás:** Több megoldás létezése esetén rendszer az optimális megoldást találja-e meg?
- Időigény:** Mennyi ideig tart egy megoldás megtalálása?
- Tárigény:** Mekkora tároló területre van szükség a megoldás megtalálásához?

Szélességi keresés (breadth-first search) mindig azt a csomópontot járja be először, amelyik legelőször lett hozzá adva, tehát az algoritmus sort használ. (FIFO)

Tárigénye nagy mértékű $O(b^{d+1})$ (d a mélység, b az elágazási tényező). Az időigénye megegyezik a tárigénnyel. Ez a keresés teljes, ha van megoldás és véges sok lépés után megtalálja. Ha a legsekélyebb célcsomópont valamilyen véges d mélységben fekszik, a szélességi keresés eljut hozzá az összes nála sekélyebben fekvő csomópontot kifejtve. **A szélességi keresés optimális, ha minden cselekvésnek ugyanannyi a költsége).**

Mélységi keresés (depth-first search) mindig a keresési fa legmélyebb csomópontot fejti ki elsőnek. A kereső algoritmus vermet használ a nyitott csomópontok hozzáadásához. (LIFO)

A tárigénye kis mértékű. Csak egyetlen, a gyökércsomóponttól egy levélcsomópontig vezető utat kell tárolnia, kiegészítve az út minden egyes csomópontja melletti kifejtetlen csomópontokkal. Egy kifejtett csomópont el is hagyható a memóriából, feltéve, hogy az összes leszármazottja meg lett vizsgálva. Egy b elágazási tényezőjű és m maximális mélységű állapottér esetén a mélységi keresés **Tárigénye $bm + 1$. Időigénye** legrosszabb esetben **$O(bm)$. Nem optimális. Nem teljes keresés**, mivel, korlátlan mélységű keresés esetén, ha első csomópont nem tartalmazza a megoldást, akkor sosem jön ki belőle. Ha viszont le korlátozzuk a mélységet, akkor nem biztos, hogy megtaláljuk a megoldást.

Az **optimális kereső** azon problémák esetén használható kereső, melyeknél az operátor alkalmazásokhoz valamilyen költség van rendelve. Az o operátor az állapotra való alkalmazásának költségét költség $o(a)$ -val jelöljük, illetve egy megoldás költsége alatt a megoldást alkotó operátoralkalmazások költségeinek összegét értjük. A szélességi és a

mélységi kereső olyan keresők voltak, melyeknél az operátoralkalmazásokhoz nem rendeltünk költséget. Természetesen nem minden probléma ilyen, ezért van szükség az optimális keresőre. Az optimális kereső a nyílt csúcsok közül mindig a legkisebb költségűt terjeszti ki.

2.TÉTEL

Ciklusszervezési lehetőségek

A **ciklus** utasítások ismétlése egy megadott feltétel függvényében. A ciklus törzse egy vagy több utasításból állhat, ha csak egy utasítást írunk, nem kell blokkot használni.

Megkülönböztetünk elől és hátul tesztelő ciklusokat. Elő tesztelő ciklus a **for** és a **while** ciklus. Hátul tesztelő ciklus pedig a **do – while**.

A **for** ciklust akkor használunk, amikor a ciklus törzsében szereplő utasításokat fix ismétlés N darabszor szeretnénk végrehajtani. A for szintaxisa a következő:

for (inicializáló utasítás; feltétel kifejezés; Léptető utasítás)

Amikor a program kódja a ciklushoz ér, az inicializációs utasítás egyszer lefut. Majd a feltétel kifejezés kiértékelődik és ha igaz az értéke, lefut a ciklus törzsében található utasítás. Ezután a léptető utasítás fut le, ez növeli például a feltételben használt változót növeljük.

A **while** ciklus lényegében egy inicializáció és léptető utasítás nélküli for ciklus. A feltétel kiértékelődik majd, ha igaz az állítás lefut a törzsében lévő utasítás. Általában a törzsében helyezünk el valamilyen utasítást, ami módosítja a feltételben használt változót.

A hátul tesztelő **do – while** utasítás annyiban különbözik a while ciklustól, hogy először egyszer mindenképp le fog futni a ciklus törzse, majd utána értékelődik ki a feltétel, amennyiben igazat kap, megismétli a törzsét.

Függvénykezelés, paraméter-kiértékelés

A **függvények** olyan alprogramok, amelyet a programkódban igény szerint bármennyiszer meghívhatunk. A függvények tetszőleges kódot tartalmazhatnak és más függvényeket is meghívhatnak.

A függvény szerkezete:

visszatérésiTípus név(paraméterek) {utasítások}

A függvény rendelkezik egy **visszatérési értékkel**. Ennek a típusát a függvény elején meg kell adni, ha nem akarunk visszaadni semmit se akkor void típust használunk. Majd következik a **függvény neve**, amely nem kezdődhet számmal. Ezután a **paraméterek** listája következik, ezeket formális paramétereknek nevezzük, de elhagyható. Egy függvényt létrehozhatunk paraméterek nélkül is.

A **formális** paraméterek az alprogram lokális paraméterei lesznek. Az alprogram hívásakor a neve mellett **aktuális** paramétereket adunk meg.

A **Paraméter kiértékelés** az a folyamat, amikor egymáshoz rendelődnek a formális és az aktuális paraméterek. Mindig a formális paraméterlista az elsődleges, mert ezekhez rendelődnek az aktuális paraméterek. A legegyszerűbb eset, amikor a formális paraméterlista első elemének értéke az aktuális paraméterlista első értéke lesz, a formális paraméterlista második elemének értéke az aktuális paraméterlista második értéke lesz és így tovább.

Hatáskörkezelés

A **hatáskörök** a programkódban használt változók láthatóságára vonatkozik, megszabja, hogy melyik változót hol használhatjuk a programban. A hatáskörön kívül a változó nem használható, nem lehet rá hivatkozni. A C programozási nyelvben megkülönböztetünk **lokális**, **globális** és **statikus** változókat láthatóság szerint.

Lokális változó például egy függvényben deklarált változó. Csak a függvényen belül érhető el és csak addig létezik amíg a függvény törzsében vagyunk. A függvény lefutásával megsemmisülnek. A **globális** változók olyan változók, amelyek függvényeken kívül lettek deklarálva, általában a main függvény előtt. A **statikus** változók pedig olyan változók, amelyek a program futása során csak egyszer deklarálódnak és megtartják az értéküket különböző függvény hívások között.

A **lokális és globális hatáskörök kettőset dinamikus változóknak nevezzük**, mert a program futása közben változhat az értékük.

Számrendszerek, számábrázolás

A számrendszerek használata a helyiértékes ábrázoláson alapul. Bármely valós számot elő tudunk állítani egy választott alapszám hatványainak segítségével. Az matematikában használt számrendszer a 10-es számrendszer. A számítástechnikában a leggyakrabban használt számrendszer pedig a 2-es és 16-os számrendszer.

A **kettes (bináris) számrendszerben** két számjegy van, a 0 és az 1 és a helyi értékkel tüntetjük fel azt, hogy az adott számjegyet kettőnek hányadik hatványával kell szorozni.

A **tizenhatos (hexadecimális) számrendszer** 16 számjegye van. Mivel csak 10 decimális számjegyük van, ezért 6 betűvel kiegészül következőképpen: a decimális 10, 11, 12, 13, 14, 15 számjegyeknek sorban megfelel az A, B, C, D, E, F jel. Egy hexadecimális számjegy ábrázolható négy bináris számjeggyel, mivel $16=2^4$.

A fix pontos számábrázolás minden számot tizedesvessző (kettedes pont) nélkül kezel, ezért egész számok ábrázolásához használjuk. A számítógépen a fixpontos számokat általában két vagy négy bájtban ábrázoljuk, azaz egy szám hossza 16 vagy 32 bit, de long típus használatával 64 bites számot is tárolhatunk.

Lebegőpontos ábrázolást akkor használjunk, ha túl nagy/kicsi számokkal, illetve, ha pontosan (törtekkel) akarunk számolni. A lebegőpontos szám lényege, hogy az ábrázolásánál a tizedespont „lebeg”, vagyis az ábrázolható értékes számjegyeken belül bárhova kerülhet. (Példa erre az 1,23, 12,3, 123 számok, melyek mindegyike 3 értékes számjegyet tartalmaz.)

Karakter, szöveg és logikai adat ábrázolása

Nem-numerikus **karakterek** ábrázolásához kódtáblát használunk. karakterek kódolva, számként ábrázolhatók általánosan használt kódrendszer az ASCII, amely egy byte-on tárolja a karaktereket. Az angol nyelvben lévő betűknek, számjegyeknek és egyéb írásjeleknek van egy 0 és 127 közé eső kódszáma. A különböző nyelvek speciális karakterei részére a 128 és 255 közötti kódok foglaltak a kódrendszerben. Itt helyezkednek el az ékezetes, a görög és a matematikai jelek. A szoftverek nemzetközivé válása során kiderült, hogy a sokféle karakterkészlet a számítástechnika fejlődésének egyik akadály. Megoldást egy olyan kódolás adhat, amely képes az összes nyelv összes karakterét ábrázolni. Ezért született meg a 16 bites Unicode (UCS - Universal Character Set).

Szöveget karakterek sorozataként tárolunk, tömbben. Egy szöveg végén \0 -val zárunk.

A **logikai** értékek ábrázolásához a logikában használt állások vannak, melyek vagy igazak vagy hamisak. Ennek megfelelően a logikai adatok két értéket vehetnek fel: ha igaz, az értéke 1, ha hamis, az értéke 0. A logikai adatok ábrázolása általában 1 Byte-on történik.

Heurisztika fogalma

A **heurisztika** egy állapot értékét fejezi. A kezdőállapot értéke mindig 0. Minél közelebb járunk a célállapothoz, annál nagyobb a heurisztika értéke. Nincs tökéletes heurisztika. Tipikus példák a játékok. Egy sakkjátékban minden lehetséges lépést megtesznek bizonyos mélységi szinten, és néhány értékelési funkciót alkalmaz a táblára. A heurisztika kizárja azokat a teljes ágakat, amelyek nyilvánvalóan rossz mozdulatokkal kezdődnek.

A* algoritmus

Az **A* kereső algoritmus** mindig a legolcsóbb utat keresi és tárolja. Ha olyan utat talál, ami olcsóbb akkor módosítja az útvonalat. A költség számítása az odáig megtett út költsége + a pont költsége.

Egy keresés **teljessége** abból áll, hogy tetszőleges reprezentációs gráfban véges sok keresőlépés után képe-e előállítani egy megoldást, mely a start állapottól a célba jutásig optimális költségek szerint lép, ezzel optimális megoldást előállítva, amennyiben van erre lehetőség. **Az A* kereső teljes** keresés.

Kétszemélyes, teljes információjú, determinisztikus játékok

Teljes információjú játékokban a játék minden eleméről ismeretünk van. Pl: Sakkban az egész pályát látjuk.

Egy játék **determinisztikus**, ha nincs a véletlennek szerepe a játékban, tehát minden esetben tudjuk, hogy mi fog történni a következő körben, következő lépésben.

A stratégia fogalma, minimax-algoritmus

A **stratégia** nem más, mint egyfajta előírás: egy-egy állapotban melyik operátort alkalmazza a gép. **Nyerő stratégia** olyan stratégia, melynek az előírásai szerint alkalmazva az operátorokat az adott játékos mindenképpen nyer (az ellenfél lépéseitől függetlenül).

Minimax-algoritmus

A **minimax algoritmus** az optimális döntést az aktuális állapotból számítja ki, felhasználva az egyes követő állapotok minimax értékeinek kiszámítására a definiáló egyenletekből közvetlenül származtatott, egyszerű rekurzív formulát. A rekúzió egészen a falevelekig folytatódik, majd a minimax értékeket a fa mentén **visszafelé** terjesztjük ahogy a rekúzió visszalép. Valamilyen heurisztikára lesz szükség, amely megmondja, hogy az adott játékos számára az adott állás mennyire jó, mennyire ígéretes. Minél jobb az állás, annál nagyobb értéket rendelünk hozzá. Döntetlen: 0 körüli érték. Ha az ellenfél számára jobb: negatív érték.

A minimax algoritmus a játékfa teljes mélységi feltárását végzi. Ha a fa maximális mélysége m , és minden csomópontban b legális lépés létezik, akkor a minimax algoritmus **időkomplexitása** $O(b^m)$.

A **tárkomplexitása** $O(bm)$ egy olyan algoritmus számára, amely az összes követőt egyszerre számítja ki, és $O(m)$ egy olyan algoritmus esetében, amely a követőket egyenként generálja.

Alfa-béta vágás

Ha az az **alfa-béta vágást** egy standard minimax fára alkalmazzuk, ugyanazt az eredményt adja vissza, mint a minimax, a döntésre hatással nem lévő ágakat azonban lenyesi. Csökkenti a játékfa méretét, legjobb esetben megfelelhetjük, azáltal, hogy a nem előnyös ágakat ki sem bontjuk.

Az alfa-béta keresés az α és a β értékeit keresés közben frissíti, és a csomópontnál a megmaradó ágakat lenyesi, amint csak biztossá válik, hogy az aktuális csomópont értéke rosszabb lesz, mint az aktuális α és β érték, MAX-ra, illetve MIN-re.

α = az út mentén tetszőleges döntési pontban a MAX számára eddig megtalált legjobb (azaz a legmagasabb értékű) választás értéke.

β = az út mentén tetszőleges döntési pontban a MIN számára eddig megtalált legjobb (azaz a legkisebb értékű) választás értéke.

3. TÉTEL

Tömbök: A tömb egy olyan változó, amely több azonos típusú adatot tartalmaz. A tömb hossza a létrehozáskor dől el, és attól kezdve a tömb egy állandó méretű adatszerkezet. A tömböket egy fiókos szekrényhez lehetne hasonlítani, amelyekben az egyes fiókokban egy-egy adatot helyezünk el. A fiókokra a sorszámukkal (A tömbben elfoglalt helye.) hivatkozunk. **A sorszámozást nullával kezdjük!** Ha valamelyik fiók tartalmára szükségünk van, akkor megadjuk, hogy hányadik fiókról van szó és kiolvassuk a tartalmát. A tömb elemei egyforma típusúak kell legyenek, de ez a típus bármi lehet. Létrehozhatunk egészek, valósak, karakterek, akár logikai típusú változók tömbjét is. C-ben az elemek számozása 0-tól történik, és ez a legtöbb másik programozási nyelvben is így van.

A tömb *indexelése* (indexing), más néven címezése szögletes zárójellel (bracket) történik. A művelet által megkapjuk a tömb egyetlen egy elemét, amely ugyanúgy használható, mint egy önálló változó.

```
tomb[9] = 3;
```

A tömböket gyakran *ciklussal* dolgozzuk fel. Ilyenkor figyelni kell arra, hogy a tömbindexek tartománya *0-tól méret-1-ig* terjed. A lenti egy tipikus tömbös ciklus. Nullától indul az iterátor (ez a tömb legelső eleme), és egyesével növekszik.

```
for (i = 0; i < 10; i += 1)
    tomb[i] = 0;
```

A tömbre mint fix méretű tárolóra kell gondolnunk elsősorban. Egy tömb nem fog magától megnyúlni, sem összemenni: akkora marad, annyi számot tud tárolni, amennyit a létrehozásakor megadtunk. Emiatt a meg nem adott méretű tömböt definiáló programsor (`int tomb[];`) teljesen értelmetlen, és nagyon súlyos hibának számít.

A C nyelv újabb változata elfogadja azt, ha a tömb méretét változóval adjuk meg, mint lent a `scanf()`-es példában. Ezzel azonban óvatosan kell bánni, illet csak ellenőrzött körülmények között szabad csinálni. Mi történik akkor, ha a felhasználó negatív számot ad meg? És akkor, ha egy óriási pozitív számot, amennyi memóriája nincs a gépnek?

Helyes, de vigyázni vele:

```
scanf("%d", &db);
double tomb[db];
```

Mutató: A mutató egy olyan változó, amely memóriacímet tartalmaz. A mutató egy hatékony eszköz, amellyel a memóriát közvetlen elérhetjük.

A mutatókat * operátorral jelöljük C-ben.

A cím előállítás a *címképző* & (address of) operátorral történik.

A pointer által hivatkozott változót az *indirekció* * (indirection) operátorral érjük el: *p.

```
double szamok1[5] = { 4.5, 9.2, 7.1, -6.9, 8 };
double szamok2[5] = { 9.3, 78, -7, 0.01, 4.6 };
double *p; pointer típusú változó
```

```
p = &szamok1[1];
printf("%f\n", *p); 9.2
```

```
p = &szamok2[3];
*p = 5.7; a szamok2[3]-at módosítja
```

```
scanf("%lf", &szamok1[2]);
```

Karakterlánckezelés: Azt az adattípust, amelyben objektumok tárolva vannak, szintén stringnek **is** nevezik. Tetszés szerinti hosszúságú. Általában, meghatározott elválasztó jelek határolnak, amelyek többnyire idézőjelek – általában ' vagy ". Általánosságban, két fajtája létezik a string adattípusnak:

- *fix hosszúságú stringek*, amikor a függetlenül attól, hogy a string elemeinek száma eléri-e a maximumot, mindig ugyanakkora memóriaterületet foglalnak le a tárolásához,
- *változó hosszúságú stringek*, ahol a string hossza nem fix, és csak az aktuális hossza szerinti helyet foglal el memóriában.

A sztringek C-ben nullával ('\0' vagy 0) lezárt karaktertömbök.

h	e	l	l	o	\0
---	---	---	---	---	----

Sztring létrehozása és inicializálása

```
char szoveg1[50] = { 'h', 'e', 'l', 'l', 'o', '\0' };
char szoveg2[50] = "hello";
char szoveg3[50];
```

Ha a "hello" formát írjuk, akkor is hozzáteszi a fordító a lezáró nullát.

Egy adott méretű tömbbe méret-1 hosszú, azaz egy karakterrel rövidebb szöveg fér csak! A lezáró 0-nak is kell hely! Például: Az „alma” szó eltárolásához egy 5 (öt!) elemű karaktertömbre van szükség: `char szoveg[5]="alma"`. Négy nem elég, mert a lezáró nulla akkor már nem férne bele.

Környezetfüggetlen egy nyelvtan, ha minden A nemterminális szó jobboldalán egy nemterminálisokból, illetve terminálisokból álló szó van.

A környezetfüggetlen nyelvtanok két legfontosabb alkalmazása:

- Természetes nyelvek feldolgozása.
- Programozási nyelvek szintaxisának megadása.

CNF: Chomsky-féle normálalak

Minden Chomsky-féle normálalakú nyelvtan lambda-mentes környezetfüggetlen nyelvtan. Egy nyelvtant λ -mentesnek nevezünk, ha a szabályok jobb oldalán egyáltalán nem fordul elő a λ . Minden λ -mentes környezetfüggetlen grammatikához meg tudunk konstruálni egy vele ekvivalens Chomsky-normálformájú környezetfüggetlen grammatikát, tehát **minden környezetfüggetlen nyelv felírható normálalakra**. Egy környezetfüggetlen nyelvtan Chomsky-féle normálalakú, ha minden szabálya a következő alakú, ahol $A, B, C \in V_N$ és $a \in V_T$: $A \rightarrow a$, $A \rightarrow BC$

CYK: Cocke-Younger-Kasami algoritmus

Az algoritmus egy alulról felfele történő elemzést valósít meg. Ahhoz, hogy működjön az kell, hogy a nyelvtan Chomsky normál alakban (CNF) legyen. A felismerő algoritmus egy tetszőleges bemenő szóhoz igyekszik megkonstruálni a megfelelő levezetési fát. Tehát **eldönti, hogy lehet-e generálni a megadott szót**. A CYK algoritmus csak akkor használható, ha a nyelvtan Chomsky-normál alakban van.

Hálózatok

Az adatkapcsolati réteg az OSI modell második rétege. Feladata az adatok megbízható továbbítása az adó és vevő között. Az adategysége a keret (frame).

A fizikai réteg felett elhelyezkedő adatkapcsolati réteg tördeli szét az átküldendő információt **adatkeretekre (frames)**. Feladata, hogy hibamentes adatátviteli vonalat alakítson ki, melyen az adatok eljutnak a hálózati réteghez. Minden keretet ellenőrző összeggel, valamint a keretek elé és mögé helyezett kódokkal lát el. A kialakított kereteket sorrendhelyesen továbbítja, s a vevő által visszaküldött nyugtakereteket feldolgozza. A nyugtázás feldolgozása során összeveti az előzetesen kiszámított összeget a vevő által a fogadást követően kiszámított és visszaküldött összeggel. Ha e kettő nem egyezik meg, a keret küldését sikertelennek minősíti, és megismétli a küldést.

A hálózatok kétpontos vagy adatszóró csatornákat használnak. Az adatszóró csatornákon az ütközés szinte elkerülhetetlen ezért az adatkapcsolati réteget két alrétegre bontották:

- A **MAC-alréteghez (Medium Access Control – közegelérési alréteg)** tartoznak azok a protokollok, amelyek a közeg használatának vezérléséért felelősek.
- A **LLC-alréteg (Logical Link Control - logikai kapcsolatvezérlés)** képes hibajavításra és forgalomszabályozásra, és még arra is képes, hogy teljesen eltakarja a különböző 802-es hálózatokat azzal, hogy egységes formátumot és felületet biztosít a hálózati rétegek számára.

Ha hibás egy keret, vagy eldobásra került, az adónak értesülnie kell róla, különben nem tudja kezelni. Ezért az adótól *nyugtázást* vár.

Elemi adatkapcsolati protokollok:

Szimplex: Az adatátvitel mindig csak egy irányban, az adótól a vevőhöz folyhat, csak egy irányban továbbítható az adatok. Tehát az adó csak adhat, a vevő csak vehet. Nincs meghatározva az adatátviteli sebesség, a feldolgozás. Amilyen sebességgel küldi az adó a kereteket, a vevő ugyanolyan sebességgel képes azt fogadni. Ez azt jelenti, hogy az adó és vevő hálózati rétegé mindig készen áll. Az adatkapcsolati rétegek közötti csatorna hibamentes, kerethiba nem fordul elő.

Fél-duplex: Többször előfordul, hogy a vevő nem képes olyan sebességgel feldolgozni a kapott információt, amilyen sebességgel azt az adó küldte. Ezért valamilyen módon le kell lassítani az adót, hogy a vevő képes legyen a kereteket feldolgozni. Ez egy módon lehetséges, ha a vevő valamilyen nyugtát küld az adónak, hogy megkapta a keretet és feldolgozta, és csak ezután indulhat a következő keret. Tehát az adónak addig várni kell, amíg valamilyen üzenetet nem kap vissza a vevőtől. Ezért nevezik ezt a protokoll "megáll és vár" protokollnak.

Duplex: A gyakorlatban az adatátvitel legtöbbször kétirányú. Egyazon a csatornán küldi el az adó az adatkereteket, és küldi vissza a vevő a nyugtakeretet. A két keret megkülönböztethető egymástól a keret fejrészében elhelyezett jelző alapján, ami a keret vételekor azonosítható. Hogy ne legyen olyan nagy forgalom az átviteli vonalon, a keretek számát lehet csökkenteni. Ennek lehetséges módja, hogy bármelyik irányba tartó adatkeretre ráültetjük az előző másik irányból jövő adatkeret nyugtáját.

A **lokális hálózatok** (LAN) általában egy épületen vagy intézményen belül számítógépek kapcsolata.

A LAN hálózat elemei

Kiszolgáló gépek, kliens gépek, hálózati adapterkártyák, hálózati protokollok, modemek, forgalom irányítók, elosztók.

Elterjedtebb **topológiái**:

Busz (sín): Minden elem egy kábelre van felfűzve, mely a két végén lezáró elemmel van ellátva. Az elrendezés hátránya, hogy vonalszakadás esetén az egész hálózat használhatatlanná válik.

Csillag: Egy központi vezérlő (HUB) kapcsolja össze a két kommunikálni kívánó gépet. Ezen minden jel kötelezően áthalad, mielőtt elér a rendeltetési helyére. A kapcsolat létrejötte után a hálózat úgy működik, mintha közvetlen kapcsolatban lenne a két gép. Az elrendezés előnye, hogy vonalszakadás esetén csak az adott gép válik használhatatlanná, és nem az egész hálózat. A többi gép továbbra is tud kommunikálni egymással.

Gyűrű (token-ring): A hálózat elemei olyan átviteli közeghez kapcsolódnak, melynek eleje és vége ugyan az. vagyis egy kört alkot. Ennek mentén az adatcsomag körbe fut, míg el nem éri a címzettet. Előnye, hogy egyszeres vonalszakadás esetén a hálózat nem válik használhatatlanná és nincs leterhelt központi csomópont. Nagyobb hálózatok esetében kétszeres gyűrűt szoktak alkalmazni a biztonság növelése érdekében.

Az **Internet** egy globális méretű számítógép-hálózat, amelyen a számítógépek az internetprotokoll (IP) segítségével kommunikálnak, amely az OSI modell 3. rétegében a hálózati rétegben helyezkedik el.

Az IP-ben a forrás- és célállomásokat (az úgynevezett hostokat) címekkel (IP-címek) azonosítja, amelyek 32 biten ábrázolt egész számok. Hogy a címek között ne legyenek ütközések, azaz egyediek legyenek, ezt nemzetközi szervezetek koordinálják.

Annak az érdekében, hogy a szervezetek a nekik kiosztott címosztályokat további alhálózatokra bonthassák, vezették be az alhálózatot jelölő maszkot. Ezzel lehetővé válik pl. egy B osztályú cím két vagy több tartományra bontása, így elkerülhető további internetcímek igénylése.

Az alhálózati maszk szintén 32 bitből áll: az IP-cím hálózati részének hosszáig csupa egyeseket tartalmaz, utána nullákkal egészül ki - így egy logikai ÉS művelettel a host mindig megállapíthatja egy címről, hogy az ő hálózatában van-e. Az IP-címekhez hasonlóan az alhálózati maszkot is byte-onként (pontosított decimális formában) szokás megadni - például 255.255.255.0. De gyakran találkozhatunk az egyszerűsített formával - például a 192.168.1.1/24 - ahol az IP-cím után elválasztva az alhálózati maszk 1-es bitjeinek a számát jelezzük.

HTML

Az oldalak leíró nyelve a HTML (Hyper Text Markup Language). A HTML-oldalak csak ASCII karakterekből állnak, nem érzékenyek kis- és nagybetűkre. Egy HTML-dokumentum 2 részből épül fel: fejléc + dokumentumtörzs. A fejléc tartalmazza a dokumentum címét, meta információkat (pl. készítő neve, érvényesség), script-programot, megjegyzéseket. Tag-eknek nevezzük a HTML-oldal elemeit, amelyeket < és > jelek között helyezünk el. Általában a tag-eknek van nyitó és záró része. A zárórész esetén a tag neve előtt / jel van.

A webcím, más néven URL (mely a Uniform Resource Locator [egységes erőforrás-azonosító] rövidítése), az interneten megtalálható bizonyos erőforrások (például szövegek, képek) szabványosított címe. A webcím az internet történetének alapvető újítása. Egyetlen címben összefoglalja a dokumentum megtalálásához szükséges négy alapvető információt:

- a protokollt, amit a célgéppel való kommunikációhoz használunk;
- a szóban forgó gép vagy tartomány nevét;
- a hálózati port számát, amin az igényelt szolgáltatás elérhető a célgépen;
- a fájlhoz vezető elérési utat a célgépen belül

4. TÉTEL

Funkcionális követelmények

Egy rendszer funkcionális követelményei leírják, hogy a rendszernek milyen funkciókkal kell rendelkezni, hogyan kellene működnie (Például aktualizálások, lekérdezések, jelentések, kimenetek, adatok, más rendszerekkel való kapcsolat). Ezek a követelmények a fejlesztett szoftver típusától, a szoftver leendő felhasználóitól függenek. Természetesen itt is megjelenik a követelmények másik megközelítése, miszerint kifejthetők mind felhasználói követelményekként, melyek rendszerint egészen absztrakt leírások, mind pedig rendszerkövetelményként, amelyek a rendszerfunkciókat részletesen írják le. Nagyon fontos, hogy a rendszer funkcionális követelményt leíró specifikációjának tartalmaznia és definiálnia kell a megrendelő által igényelt összes szolgáltatást, az az teljesnek és ellentmondásmentesnek kellene lennie.

PR. Tétel: A lineáris keresés tétele

Rendelkezésre áll egy N elemű sorozat, és egy, a sorozat elemein értelmezett T tulajdonság. Olyan algoritmust kell írni, amely eldönti, hogy van-e T tulajdonságú elem a sorozatban, s ha van, akkor megadja a sorszámát (ennyivel több mint az eldöntés tétele). Az algoritmus:

Eljárás Lin_Keresés

$i:=1$

Ciklus amíg $i \leq N$ **És** $A[i]$ nem T tulajdonságú

$i:=i+1$

Ciklus vége

Van: $i \leq N$

Ha van akkor Ki: i

Eljárás vége // csak egy elemet keres

PR. Tétel: A logaritmikus keresés tétele

Rendelkezésre áll egy N elemű növekvő sorrendbe rendezett (!!!!!) sorozat és egy keresett elem (X). Olyan algoritmust kell írni, amely eldönti, hogy szerepel-e a keresett elem a sorozatban, s ha igen, akkor megadja a sorszámot. Most kihasználjuk, hogy a sorozat rendezett. Ez alapján bármely elemről el tudjuk dönteni, hogy a keresett elem előtte vagy utána van-e, esetleg megtaláltuk. Az eljárás lényegének megértéséhez tudni kell, hogy az E és az F változóknak kiemelt szerepük van: mindig annak a részintervallumnak az alsó és felső végpontjai, amelyben a keresett elem benne van.

Eljárás Log_Keresés

$E:=1$; $F:=N$

Ciklus // K -ban lesz az intervallum közepe

$K:=(E+F) \text{ DIV } 2$ // $A \text{ DIV}$ továbbra is egész osztás

Ha $A[K] < X$ **akkor** $E := K + 1$
Ha $A[K] > X$ **akkor** $F := K - 1$
Amíg $E > F$ **vagy** $A[K] = X$
 // Ha összeért a két részintervallum, vagy megtaláltam, akkor vége
 $Van := E \leq F$
Ha Van **akkor** $Sorsz := K$

Eljárás vége

PR. Tétel: Az eldöntés tétele

Adott egy N elemű sorozat és egy, a sorozat elemein értelmezett T tulajdonság (pl. a kettővel való oszthatóság, vagy a számjegyek összege prímszám, stb.). Az algoritmus eredménye: annak eldöntése, hogy van-e a sorozatban legalább egy T tulajdonsággal rendelkező elem.

Eljárás Eldönt

$i := 1$

Ciklus **amíg** $i \leq N$ **És** $A[i]$ nem T tulajdonságú $i := i + 1$

Ciklus vége

$VAN := i \leq N$ // Van értéke igaz, ha $i \leq N$ igaz, egyébként hamis

Eljárás vége

PR. Tétel: A kiválogatás tétele

Egy N elemű sorozat összes T tulajdonsággal rendelkező elemét kell meghatározni. Gyűjtsük a kiválogatott elemek sorszámait a B vektorban!

Eljárás Kiválogat $j := 0$ // Nem szabad elfelejteni

Ciklus $i := 1$ -től N -ig

Ha $a[i]$ T tulajdonságú **akkor** $j := j + 1$

$B[j] := i$ // a sorszámot tároljuk

Elágazás vége

Ciklus vége

Eljárás vége

Itt is azért tároljuk az indexeket, mert így kevesebb helyet foglal a $B[]$ vektor a memóriából. Ha ki is kell írni az $A[]$ vektor T tulajdonságú elemeit, akkor az a következőképpen történhet:

Ciklus $i := 1$ -től j -ig // azért megy j -ig a ciklus, mert B -nek j db eleme van

Ki: $A[B[i]]$

Ciklus vége

Az $A[B[i]]$ az a vektor $B[i]$ -edik elemét jelenti. Ha pl. $i = 1$ és a $B[i]$ (a B vektor i -edik eleme) egyenlő 5, akkor $A[5]$ -öt, azaz a vektor 5. eleme kerül kiírásra.

Buborékos rendezés

A módszer lényege

Végigmenve az adatsort tartalmazó vektoron minden szomszédos elempárt növekvő sorrendbe rakunk, azaz meghagyjuk helyükön, vagy megcseréljük őket aszerint, hogy jó sorrendben voltak vagy sem. Egy ilyen menet végén a legnagyobb elem a vektor végére kerül. Ugyanezt a páronkénti cserét végrehajtjuk a még rendezetlen $A[1..(n-1)]$ részvektoron. Ezzel a második legnagyobb elem az $A[n-1]$ elem helyére kerül. Addig folytatjuk az egyre rövidebb rendezetlen sor párcseréit, míg minden elem a helyére kerül. Egy menetben a maradék sor legnagyobb eleme, mint egy buborék halad végig az adatsoron. Innen származik a módszer neve.

Eljárás Buborék

$r:=n$;

$csere:=hamis$

Ciklus amíg $r>1$ és **Nem** $csere:=igaz$

Ciklus $i:=1$ -től $r-1$ -ig

Ha $A[i]>A[i+1]$ **akkor** //itt a páronkénti összehasonlítás

$Csere(A[i], A[i+1])$

$csere:=hamis$

Elágazás vége $i:=i+1$

Ciklus vége

$r:=r-1$

Ciklus vége

Eljárás vége

A buborékredezési algoritmust a következőképpen javíthatjuk: A módszer azon az észrevételen alapul, hogy egy menetnek az eredményeként a maximális elem a sor végére kerül. Eszerint a menet közbeni adatcseréket megtakaríthatjuk, ha a maximális elemet és ennek helyét megállapítjuk és csak a menet végén egyetlen cserével tesszük a helyére. Ezt a módszert maximum-kiválasztásos rendezésnek nevezzük.

Relációs adatmodell

A **relációs modellben** az adatokat táblázatok soraiban képezzük le. A táblázat soraiban található elemek alkotnak egymással relációt. A reláció résztvevőit, a reláció típusát a táblázat fejléce adja meg. A táblázatban egy sort rekordnak neveztek el, míg a táblázat egy oszlopa pedig egy mező. **Egyed:** adatállomány minden olyan dolgot egyednek nevezünk, melynek jellegzetes tulajdonságai vannak. Ezen jellegzetes tulajdonságoknak lehetővé kell tenniük az egyedek megkülönböztetését. Az így megkülönböztetett dolgokról, most már egyedekről adatot, adatokat tárolunk. Egyszerűbben, minden egyed, amiről adatot tudunk tárolni.

Attribútum: adatmező (tulajdonság) Az adatállományban a rekordok jellemzéséhez használjuk. Az egyedeket tulajdonságaik alapján jellemezzük, különböztetjük meg ezeket a tulajdonságokat tároljuk el az egyedről. A tulajdonságok lehetnek összetettek is, állhatnak több alrendű tulajdonságból is.

Reláció-kapcsolat: Kapcsolatnak nevezzük az egyedek közötti viszonyt, összefüggést. **Egy-egy kapcsolat:** Az egy-egy kapcsolat során az egyik egyedhalmaz minden egyes eleméhez, pontosan egy másik egyedhalmazbeli elem tartozik. Egyszerűbben: kölcsönösen egyértelmű a két egyedhalmaz közötti megfeleltetés. Jelölése, 1:1 kapcsolat. Pl.: Mindenkinek csak egy személyi száma van, és minden egyes személyi számhoz csak egy ember tartozik.

Egy-több kapcsolat: Az A egyed és B egyed között egy-több kapcsolat áll fent, ha az A egyedhalmaz mindegyik eleméhez a B egyedhalmaz több eleme is tartozhat. Jelöltése: 1: N kapcsolat. Pl.: egy anyanévhez tartozhat több személy neve is.

Több-több kapcsolat: Több-több kapcsolatról beszélünk amennyiben A egyedhalmaz minden eleméhez a B egyedhalmaz több eleme tartozik és fordítva. (B egyedhalmaz minden egyes eleméhez több A egyedhalmazbeli elem tartozik.)

Kulcs: Az egyed azon tulajdonsága vagy tulajdonságai mely már egyértelműen meghatározza, az egyedet. Egyszerűbben azon tulajdonság mely azonosítja az egyedet, egyértelműen meghatározza a reláció egy sorát: például az emberek esetén: a személyi szám.

- Elsődleges kulcs: Elsődleges kulcsnak nevezzük, ha egyértelműen meghatározzák.
- Egyszerű kulcs: ha egyetlen attribútumból áll.
- Összetett kulcs: ha több attribútumból áll.
- Idegen kulcs: egy reláció azon attribútumai, melyek egy másik relációban kulcsot alkotnak. Az egyedhalmaznak lehet olyan attribútuma, mely más egyedhalmaz elsődleges kulcsa. Ezt idegen kulcsnak nevezzük.

A **hivatkozási integritás** szerint az adatbázisban nem lehet olyan idegen kulcs, melynek értéke nem egyezik meg a hivatkozott relációban valamelyik elsődleges kulcs értékével. Ennek a feltételnek az adatbázist érintő minden változtatás után érvényben kell maradnia.

Kényszer (constraint): A lehetséges adatok halmazát leíró, korlátozó szabály. Sokan a tábla elsődleges kulcsát is egyfajta megszorításnak tekintik, hiszen az elsődleges kulcs maga után von egy egyediségi (UNIQUE) kényszerfeltételt. Mi itt az idegen kulcsokról (foreign key) szólnak, amelyek a relációs adatbázis szempontjából rendkívüli fontosságúak.

Turing-gépek

A Turing-gép egy **potenciálisan végtelen szalagmemóriával és egy író-olvasó fejjel ellátott véges automata**. A szalagmemória pozíciókra van osztva, s minden egyes pozíció, mint memória-egység az úgynevezett szalagábécé pontosan egy betűjének tárolására képes. Kezdetben a Turing-gép egy specifikált kezdőállapotában van, s a szalagon egy véges hosszúságú input szó helyezkedik el. Az eddig tárgyalt automatákhoz hasonlóan ez a modell is szekvenciális működésű. Működésének kezdetekor a Turing-gép író-olvasó feje az input szó első betűjén áll. Az input szó előtti és utáni

(végtelen sok) szalagpozíció egy speciális betűvel, a szóközzel (üres betűvel) van feltöltve, ami nem tévesztendő össze az üresszóval. Többek között azért is, hogy az input szó

elkülöníthető lehessen a szalag többi részén tárolt mindkét irányban végtelen számú szóköztől, feltételezzük, hogy az input szó utolsó betűje nem lehet szóköz. Az input szó tehát az író-olvasó fej alatti betűtől (jobbra haladva) tart a szalag utolsó nem üres betűjéig. Speciálisan, üres input szó is elképzelhető. Ez esetben a szalag minden egyes pozíciója szóközzel van feltöltve, és az író-olvasó fej ezek egyikére mutat. (Utolsó szóköztől különböző betű pedig ekkor értelemszerűen nincs.) A Turing-gép diszkrét időskála mentén, elkülönített időpillanatokban hajt végre egy-egy elemi műveletet, mely az író-olvasó fej alatti betű olvasásából, ezen betű felülírásából, a belső állapot változtatásából, s az író-olvasó fej egy pozícióval való balra, vagy jobbra mozgatásából, vagy éppen a fej helybenhagyásából áll. Amennyiben a Turinggép eljut egy végállapotba, megáll.

Működését tekintve a **többszalagos Turing-gép** egy lépésben olvashat/írhat egyszerre több szalagra is. Kezdő konfigurációban az egyik szalagon (input-szalag) van a feldolgozandó adat, a többi szalag pedig üres. Többszalagos gépek esetén szokás egy szalagot az outputnak is fenntartani, ekkor a számítás végén azon a szalagon olvasható az eredmény, illetve sokszor az input szalag csak olvasható.

Minden többszalagos Turing-gép működése szimulálható egyszalagos Turing-géppel, vagyis egyszalagos Turing-gép is el tudja végezni azt a számítást, amit egy többszalagos Turing-gép.

Az **univerzális Turing-gép** egy speciális fajtája a Turing-gépnek. Egy Turing-gépet univerzálisnak nevezünk, ha minden Turing-géphez futásának eredménye az s inputon megegyezik az UTM futásának eredményével a vXs inputon (ahol $X \in V \setminus T$).

Az **univerzális Turing-gép** egy általános, elvont számítógép, **ami minden Turing-gépet képes szimulálni**, vagyis elvileg a programjának megfelelően feldolgozni az input szót. Ez azt jelenti, hogy van olyan gép, ami minden kiszámítható függvényt ki tud számolni.

Az Univerzális Turing-gép létezése azt mutatja, hogy elvileg konstruálható olyan számítási eszköz, amely programozható és mindent ki tud számítani, ami kiszámítható.

Church-tézis

A **Church-tézis** szerint minden formalizálható probléma, ami megoldható algoritmussal, az megoldható Turing-géppel is.

Megállási probléma

A Turing gépek megállási problémája nem megoldható.

Akkor mondjuk, hogy egy Turing-gép valamely input szó hatására **megáll**, ha az input szó eleme a Turing-gép által felismert nyelvnek, azaz az input szóhoz tartozó kezdő konfigurációból kiindulva eljut egy végkonfigurációba. Mondjuk azt, hogy a Turing-gépek megállási problémája megoldható, ha létezik olyan TM' Turing-gép, melynek egy alkalmas kódolási algoritmussal egy tetszőleges TM Turing-gép leírását és a TM gép egy w input szavának kódolt alakját input szóként megadva megáll. Más szóval, egyértelműen megállapítható, hogy a leírt TM Turing-gép a kérdéses w szó, mint input szó hatására el tud-e jutni egy végkonfigurációba, avagy sem. Ha ilyen TM' Turing-gép nem létezik, akkor mondjuk azt, hogy a Turing-gépek megállási problémája megoldhatatlan.

A **Megállási probléma** kérdése az, hogy egy (a kódjával adott) Turing-gép adott bemenettel egyáltalán megáll-e. Algoritmikusan nem lehet eldönteni, hogy egy univerzális Turing gép egy adott bemenettel véges időn belül leáll-e.

Algoritmikusan nem eldönthető problémák

A **Dominóprobléma** a következő: Adott dominó-típusok egy véges F halmaza; eldöntendő, hogy a sík lefedhető-e hézagtalanul szabályosan illeszkedő F -beli típusú dominókkal. Természetesen a fedéshez az egyes típusokból végtelen sok példány használható. A Dominóprobléma **nem oldható meg algoritmussal**.

Logikai függvények megadása

A logikai függvények megadása a következő módokon lehetséges:

Szöveges megadás: Az alapfeltételek (független változók) kombinációit, a logikai kapcsolatot (függvénykapcsolat) és a következtetéseket (függőváltozókat) egyaránt szavakban fogalmazzák meg.

Táblázatos megadás: Olyan értéktáblázatot hoznak létre, amely tartalmazza az alapfeltételek (független változók) minden kombinációjához tartozó következtetések (függő változók) értékeit. Az így létrejövő igazságtáblázatban logikai igazságok rögzítése történik.

Halmazokkal történő leírás: az alapfeltételekhez tartozó következtetések közötti függvénykapcsolatot illeszkedő halmazokkal lehet szemléletessé tenni.

Logikai vázlat: az alapfeltételekhez tartozó következtetések közötti függvénykapcsolatot áramköri szimbólumokkal, logikai kapuk összekapcsolásával valósítják meg.

Algebrai megadás: az alapfeltételekhez tartozó következtetések közötti logikai kapcsolatot, függvénykapcsolatot műveleti szimbólumokkal valósítják meg.

KNF, DNF

A **diszjunktív normálformánál** a literálok (betűk) ÉS-eléséből áll egy-egy kifejezés, és ezek vannak össze VAGY-olva. Tehát akkor IGAZ, ha egy-egy esetben igaz kifejezés bármelyike IGAZ. elemi konjunkciók diszjunkciója.

A **konjunktív normálformánál** a literálok VAGY-olásából áll egy-egy kifejezés, és ezek vannak össze ÉS-elve. Tehát akkor HAMIS, ha az egy-egy esetben hamis kifejezések bármelyike HAMIS. elemi diszjunkciók konjunkciója.

Logikai hálózatok

A tervezés eredménye alapvetően meghatározza, hogy a megvalósításhoz szükséges logikai függvények eredménye a bemeneti változókon kívül függ-e az események bekövetkezési sorrendjétől. Ezért kell a logikai függvényeket megvalósító logikai hálózatokkal foglalkoznunk. A logikai függvények az időfüggésük szerint lehetnek időfüggetlen, és időfüggő logikai függvények. Ennek megfelelően az őket megvalósító logikai hálózatok is két ilyen tulajdonságú csoportra oszthatók:

- A kombinációs hálózatok.
- A sorrendi (szekvenciális) hálózatok.

A **kombinációs hálózatok** időfüggetlen logikai függvényeket valósítanak meg. Olyan logikai hálózatokat, melyeknek kimeneti jelei csak a bemeneti jelek pillanatnyi értékétől függnnek.

- Memória nélküli logikai áramkörök.

A **sorrendi (szekvenciális) hálózatok** időfüggő logikai függvényeket valósítanak meg. Logikai hálózatokat, melyek kimeneti jelei nemcsak a pillanatnyi bemeneti jelkombinációtól függenek, hanem attól is, hogy korábban milyen bemeneti jelkombinációk voltak.

- Memóriával is rendelkező logikai áramkörök.

5. TÉTEL

Adatszerkezetek

A **sor** (queue) egy veremhez hasonló adatszerkezet, annyi különbséggel, hogy FIFO elven működik, azaz először azt az elemet tudjuk kivenni, amelyiket legelsőnek tettünk bele. (**push**, **pop**)

A **verem** (stack) egy LIFO adatszerkezet, amely azt jelenti, hogy mindig a legutóbb betett elemet érjük el. Alapműveletei a:

- push() -Belehelyez egy elemet a verembe
- pop() -törli a verem tetején levő elemet
- top() -Visszaadja a verem tetején levő elemet
- empty() -igaz, ha a verem üres
- size() -Visszatér a verem méretére

Láncolt lista

Láncolt lista(linked list): adatszerkezet, ahol az egyes elemek (node) láncba vannak fűzve azáltal, hogy tárolják a szomszédos elem címét. Tetszőleges - ráadásul akár széles skálán változó - számú elem tárolására, gyűjtésére ad lehetőséget. A láncolt lista nagy **előnye** a tömbbel szemben, hogy eltérő típusú és méretű elemeket is képes magába foglalni, amelyek ráadásul a memóriában nem feltétlenül kell, hogy szekvenciálisan - és a listában szereplő sorrendben - helyezkedjenek el, hanem tetszőleges módon szétszórva lehet tárolni őket. A láncolt lista **hátránya**, hogy - szemben pl. a tömbbel - az elemek véletlenszerűen ill. pusztán sorszámuk alapján közvetlenül nem, csak a lista (részleges) bejárásával érhetők el és címezhetők meg, így az ilyen hozzáférést igénylő algoritmusokban történő felhasználás általában jóval lassabb működést eredményez, mintha pl. tömböt alkalmaznánk az adatok tárolására.

Bináris fa: Hierarchikus adatszerkezet, ahol minden elemre igaz, hogy legfeljebb két rákövetkező eleme (gyerekeleme) lehet, és minden elemnek pontosan egy szülőeleme van, kivéve az úgynevezett gyökér elemet, melynek nincs szülő eleme. A fák tárolására szétszórt ábrázolást szokás alkalmazni. Az adat rész mellett minden adatelem tartalmaz két mutató típusú tagot is, melyek az elem bal, illetve jobb oldali leszármazottjára mutatnak. Ha a leszármazott nem létezik, akkor az adott oldali mutató NULL értéket vesz fel.

Bináris fa műveletei:

- **Létrehozás:** Létrehozunk egy dinamikusan kezelhető adat struktúrát.
- **Bővítés:** Hozzáadunk egy elemet a meglévő struktúrához
- **Bejárás:** az adatszerkezet valamennyi elemének egyszeri elérése (feldolgozása)
- A csomópontokban található adatok (tartalom, bal, jobb) feldolgozásának sorrendje alapján három fő változat különböztethető meg
 - o **Preorder** bejárás: tartalom, bal, jobb
 - o **Inorder** bejárás: bal, tartalom, jobb
 - o **Postorder** bejárás: bal, jobb, tartalom
- Az általános **keresés** (tetszőleges feltételnek megfelelő tartalom keresése) az előzőleg megismert bejárások segítségével valósítható meg.
- **Megadott kulcsú elem** keresésekor már ki tudjuk használni a fa rendezettségét: a fa gyökérelemének kulcsa vagy egyenlő a keresett kulccsal, vagy egyértelműen meghatározza, hogy melyik részében kell a keresést folytatni
- A **beszúrás** során az elem beláncolásán kívül ügyelnünk kell a keresőfa tulajdonság fenntartására is
- Ugyanazok az elemek **többféleképpen** is **elhelyezkedhetnek** egy bináris keresőfában, **beszúrás**kor ez alapján több stratégiánk is lehet
 - o minél kisebb erőforrásigényű legyen a beszúrás ,
 - o minél kiegyensúlyozottabb legyen a fa a beszúrás(ok) után
- A **törlés** során az elem kiláncolásán kívül ügyelnünk kell a keresőfa tulajdonság fenntartására is
- **Törlés** során az alábbi **problémák** merülhetnek fel
 - o Két gyerekkel rendelkező elem mindkét gyereket nem tudjuk az ő szülőjének egy mutatójára rákapcsolni.
 - o Gyökérelem törlése

Operációs rendszerek folyamatai memóriakezelése

Operációs rendszer – koordinálja és vezérli a hardvererőforrások különböző felhasználók különböző alkalmazói programjai által történő használatát.

Folyamatkezelés, ütemezés (scheduling): Process egy végrehajtás alatt álló program. Egy programból úgy lesz folyamat, hogy az op. rendszer betölti a programot a háttértárból a memóriába. És átadja neki a vezérlést. A folyamat megszűnésekor az op. rendszer felszabadítja az általa lefoglalt területet. A modern op. rendszerek több folyamatot képesek kezelni egyidejűleg. Már maga az op. rendszer is több folyamatból áll. Minden folyamatot csak

egy másik, már működő folyamat hozhat létre (szülő). Az op. rendszer képes erőszakkal leállítani a folyamatot. A **folyamat ütemezése** többféleképpen is lehetséges: **prioritásos** (a nagyobb prioritású folyamat mindaddig fut, amíg be nem fejeződik), **időosztásos** (mindegyik futásra váró folyamat, ugyan azonos időszületet kap), vegyes.

Memóriakezelés: Az operációs rendszer szemszögéből a memóriát egy bájtokból álló tömbnek tekinthetjük. Az operációs rendszernek nyilván kell tartani, hogy az operatív memória melyik részét ki mire használja. El kell döntenie, hogy a felszabadult memóriaterületre melyik folyamatot tölti be. Szükség szerint memóriaterületeket foglal le, vagy szabadít fel. A memóriák kiosztása a programozók számára átláthatóan kell történnie. Mivel a memória törlődik, ezért egy olyan másodlagos tárolóra volt szükség, amely adatok tárolására hosszabb időre is alkalmas. A másodlagos tár pl. HDD kezeléséért is az operációs rendszer felel. Tudni kell kezelnie a szabadhelyeket, allokálást, lemezelosztást, ütemezést.

Állománykezelés

A létrehozó által összetartozónak ítélt információk gyűjteménye. Több százezer állomány egyidejűleg, egyedi azonosító (név) különbözteti meg őket. Az állomány elrejt a tárolásának, kezelésének fizikai részleteit:

- Melyik fizikai eszközön található (logikai eszköz névvel határozza meg)
- a perifériás illesztő tulajdonságait,
- az állományhoz tartozó információk elhelyezkedését a lemezen,
- az állományban lévő információk hogyan helyezkednek el a fizikai egységen (szektor, blokk),
- az információ átvitelénél alkalmazott blokkosítást, pufferelement.

Az állománykezelő feladatai: információátvitel (állomány és folyamatok között), műveletek (állományokon és könyvtárokon), osztott állománykezelés, hozzáférés szabályozása (itt a más felhasználók által végezhető műveletek korlátozása (access controll), a tárolt információk védelme illetéktelen olvasók ellen (encryption), információk védelme a sérülések ellen és a mentés.

A fájlokkal történő műveletek: Megnyitás, létrehozás, törlés, visszaállítás, másolás áthelyezés, átnevezés és nyomtatás.

A fájl (állomány) fogalma: Logikailag összefüggő adatok rendezett halmaza, melyek egy közös névvel rendelkeznek. A fájlok általában mágneses adathordozón helyezkednek el. A fájlok tartalma szöveg, numerikus adat, grafika, hang, stb. lehet. A fájlok típusa: a fájlokat az utolsó **kiterjesztésük** alapján különböző típusokba soroljuk:

Programfájlok: EXE, COM, BAT. Azonnal futtatható fájlok (szoftverek).

Adatfájlok: szöveg (TXT, DOC, INI), kép (BMP, PCX, JPG), szoftverfájlok (XLS/ Excel/stb.).

Tárolásuk alapján beszélhetünk tömörített és tömörítés nélküli fájlokról.

Üresszó lemma

Minden 2 típusú (környezetfüggetlen) nyelvtanhoz megadható vele ekvivalens 1 típusú (környezet-függő) nyelvtan.

Minden környezetfüggetlen G grammatikához megadható olyan G' környezetfüggetlen nyelvtan, hogy $L(G)=L(G')$ (azaz az általuk generált nyelv ugyanaz), s ha $\lambda \notin L(G)$, akkor a G' -beli szabályok jobboldalán λ nem fordul elő. Ha viszont $\lambda \in L(G)$, akkor az egyetlen G' -beli szabály, aminek jobboldala az üresszó $S' \rightarrow \lambda$, ahol S' a G' mondatzimbólumát jelöli. Ezesetben, azaz $\lambda \in L(G')$ fennállása esetén viszont S' (azaz a G' mondatzimbóluma) nem fordulhat elő egyetlen G' -beli szabály jobboldalán sem. Ennek megfelelően, tehát G' nemcsak környezetfüggetlen, de egyben a környezetfüggő definíciónak is eleget tesz.

Automata

Egy olyan absztrakt rendszer, mely egy diszkrétnek képzelt időskála időpillanataiban érkezett jelek hatására ezen időpillanatokban válasszal reagál, miközben belső állapotát megadott szabályok szerint változtatja a külső jelek hatására. Az **automata véges**, ha az **állapothalmaz**, a **bemenő jelhalmaz** és a **kimenő jelhalmaz végesek**.

Fajtái: Nemdeterminisztikus és determinisztikus automata | Moore-automata, Mealy-automata?

Amennyiben az átmeneti és a kimeneti nem egyértelműen definiáltak, **nemdeterminisztikus automatáról** van szó. További változata a nemdeterminisztikus automatáknak, ha megengedjük, hogy az automata bemenő jel nélkül is állapotot váltson. Ezeket szokás **üresszóátmenetes** (nemdeterminisztikus) **automatáknak** is nevezni.

A **determinisztikus** automata esetén tehát a szóban forgó függvény értékek mindig pontosan egy meghatározott értéket vesznek fel. Determinisztikus automatáknál **nem fordulhat elő üresszóátmenet**.

Egy nyelv akkor és csak akkor ismerhető fel nemdeterminisztikus automatával, ha felismerhető determinisztikus automatával.

Egy nemdeterminisztikus véges automata **determinisztikus**, ha

$$|I| = 1 \text{ és } |\delta(q, a)| \leq 1, \forall q \in Q, \forall a \in \Sigma.$$

6. TÉTEL

OOP (objektum-orientált programozás)

Az **objektum-orientált programozás** a „dolgokat” („objektumokat”) és köztük fennálló kölcsönhatásokat használja alkalmazások és számítógépes programok tervezéséhez. Az OOP olyan megoldásokat foglal magában, mint a **bezárás** (encapsulation), a **többalakúság** (polymorphism) valamint az **öröklés** (inheritance).

Egy programnyelv objektum-orientáltságát az említett elvek támogatásával lehet elérni.

Bezárás, adatrejtés (encapsulation): Lehetnek olyan jellemzők és metódusok, melyeket elfedünk más objektumok előtt. Az adatok és a metódusok osztályba való összezárását jelenti. Tulajdonképpen az objektum egység bezárja az állapotot (adattagok értékei) a viselkedésmóddal (műveletekkel). Következmény: az objektum állapotát csak a műveletein keresztül módosíthatjuk. Minden objektum egy jól meghatározott interfészt biztosít a külvilág számára, amely megadja, hogy kívülről mi érhető el az objektumból.

Öröklés (inheritance): Segítségével egy általánosabb típusból (ősosztály) egy sajátosabb típust tudunk létrehozni (leszármazott osztály). Az leszármazott osztály adatokat és műveleteket (viselkedésmódot) örököl, kiegészíti ezeket saját adatokkal és műveletekkel, illetve felülírhat bizonyos műveleteket. A kód újra felhasználásának egyik módja. Megkülönböztetünk egyszeres és többszörös öröklést. Javában csak egyszeres öröklődés van, azaz egy osztály legfeljebb egy őszosztályt örökölhet.

Javában az *Object* osztály az osztályhierarchia legfelső eleme, minden más osztály belőle származik.

Polimorfizmus (polymorphism): Lehetővé teszi, hogy az öröklés során bizonyos viselkedési formákat (metódusokat) a származtatott osztályban új tartalommal valósítsunk meg, és az új, lecserélt metódusokat a szülő osztály tagjaiként kezeljük. A gyerekosztály egy példánya kezelhető a szülő egy példányaként is. Egy ős típusú tömbben eltárolhatjuk az ős- és gyerektípusokat vegyesen. Azonban, ha ős típusként tárolunk egy gyerek típusú objektumot, akkor a gyerek típusú objektum saját osztályában definiált metódusait nem látjuk.

Típusok és konverziók

A Java nyelvben az adattípusoknak két csoportja van: **primitív és referencia típusok**. A primitív adattípusok egy egyszerű értéket képesek tárolni: számot, karaktert vagy logikai értéket. A változó neve közvetlenül egy értéket jelent.

A primitív típusok a következők:

Egészek

Típus	Leírás	Méret/formátum
byte	bájt méretű egész	8-bit kettes komplement
short	rövid egész	16-bit kettes komplement
int	egész	32-bit kettes komplement
long	hosszú egész	64-bit kettes komplement

Valós számok

Típus	Leírás	Méret/formátum
float	gyors pontosságú lebegőpontos	32-bit IEEE 754
double	dupla pontosságú lebegőpontos	64-bit IEEE 754

Egyéb típusok

Típus	Leírás	Méret/formátum
char	karakter	16-bit Unicode karakter
boolean	logikai érték true vagy false	

A tömbök, az osztályok és az interfészek referencia-típusúak. A referencia-változó más nyelvek mutató vagy memóriacím fogalmára hasonlít.

Sokszor van szükség a különböző adattípusok közti átváltásokra, ezt **típuskonverciónak** nevezzük. Megkülönböztetünk **implicit** és **explicit** típuskonverziót.

- Az **implicit konverzió** esetén a Java automatikusan átkonvertálja az egyik típust a másikra.
- Az **explicit konverzió** olyankor történik, amikor a kódban megjelöljük (rá kényszerítjük) az adott típusra a változót.

Operátorok

Az **operátorok** egy, kettő vagy három operanduson hajtanak végre egy műveletet. Az egyoperandusú operátorokat unáris operátoroknak hívjuk. Például a ++ operátor az operandusát 1-gyel növeli.

Javában egy háromoperandusú operátor van, a ?: feltételes operátor.

A művelet végrehajtása után a kifejezés értéke rendelkezésre áll. Az érték függ az operátortól és az operandusok típusától is. Aritmetikai operátorok esetén a típus alá van rendelve az operandusoknak: ha két int értéket adunk össze, az érték is int lesz. A Javában a kifejezések kiértékelési sorrendje rögzített, vagyis egy **művelet operandusai mindig balról jobbra értékelődnek** ki (ha egyáltalán kiértékelődnek, lásd rövidzár kiértékelés), még a művelet elvégzése előtt. Javában a következő operátorokkal dolgozhatunk:

- Aritmetikai operátorok
- Relációs operátorok
- Logikai operátorok
- Bitléptető és bitenkénti logikai operátorok
- Értékadó operátorok
- Egyéb operátorok

Az **aritmetikai operátorok** a + (összeadás), - (kivonás), * (szorzás), / (osztás) és % (maradékképzés).

A **relációs operátorok** összehasonlítanak két értéket, és meghatározzák a köztük lévő kapcsolatot. Például: >, >=, <, <=, ==, !=

A relációs operátorokat gyakran használják **logikai operátorokkal** együtt, így összetettebb logikai kifejezéseket hozhatunk létre. (&& (és), || (vagy))

A **léptető operátorok bit műveleteket** végeznek, a kifejezés első operandusának bitjeit jobbra, vagy balra léptetik. (<< (balra léptetés), >> (jobbra léptetés), >>> (jobbra léptetés, balról nullákkal tölti fel))

A **bitenkénti logikai operátorok**:

- op1 & op2 Bitenkénti és ha mindkét operandus szám; feltételes és ha mindkét operandus logikai
- op1 | op2 Bitenkénti vagy, ha mindkét operandus szám; feltételes vagy, ha mindkét operandus logikai
- op1 ^ op2 Bitenkénti kizáró vagy (xor)
- ~op2 Bitenkénti negáció

Az alap **értékadó (=) operátort** használhatjuk arra, hogy egy értéket hozzárendeljünk egy változóhoz. A Java programozási nyelv azt is megengedi a rövidített értékadó operátorok segítségével, hogy aritmetikai, értéknövelési, valamint bitenkénti műveletvégzést összekössük az értékadással.

Ilyenek például: -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, >>>=

Utasítások

Az **utasítás** egy konkrét futási egységet hoz létre. A következő kifejezéstípusok szervezhetők utasításokba, amelyek pontosvesszővel (;) végződnek:

- értékadó kifejezések
- ++ és -- használata
- metódushívások
- objektumot létrehozó kifejezések

A **metódusok** az osztályok tagfüggvényei, amik lehetnek példánymetódusok, vagy osztálymetódusok (static kulcsszóval deklaráljuk). Egy metódust is két részből áll: a metódus deklarációja és a metódus törzse. A metódus deklaráció meghatározza az összes metódus tulajdonságát úgy, mint az elérési szint, visszatérő típus, név és paraméterek. A metódus törzs az a rész, ahol minden művelet helyet foglal. A metóduson belül a return utasítással lehet a visszaadott értéket előállítani. A void-ként deklarált metódusok nem adnak vissza értéket, és nem kell, hogy tartalmazzanak return utasítást. Minden olyan metódus, amely nem void-ként

lett deklarálva, kötelezően tartalmaz return utasítást. A visszaadott érték adattípusa meg kell, hogy egyezzen a metódus deklarált visszatérési értékével.

Egy osztályon belül lehet több azonos nevű metódus, melyek a paraméterezésben és/vagy a visszatérési érték típusában térhetnek el egymástól ezt nevezzük túlterhelés-nek (overloading).

Az **osztály deklaráció** az osztály kódjának az első sora. Minimálisan az osztály deklaráció meghatározza az osztály nevét. Az osztálytörzs az osztály deklarációt követi, és kapcsos zárójelek között áll. Az osztály törzs tartalmazza mindazt a kódot, amely hozzájárul az osztályból létrehozott objektumok életciklusához: konstruktorok, új objektumok inicializálására, változó deklarációk, amelyek megadják az osztály és objektumának állapotát, és eljárásokat az osztály és objektumai viselkedésének meghatározására.

Láthatóság

Egy **elérési szint (láthatóság)** meghatározza, hogy lehetséges-e más osztályok számára használni egy adott tagváltozót, illetve meghívni egy adott metódust. A Java programozási nyelv négy elérési szintet biztosít a tagváltozók és a metódusok számára. Ezek a **private**, **protected**, **public**, és amennyiben nincsen jelezve, a **csomag szintű elérhetőség**.

- Egy osztály mindig elérheti saját tagjait.
- A **csomag** szintű elérhetőség azt jelenti, hogy az eredeti osztállyal azonos csomagban lévő más osztályok elérhetik a többi osztály tagjait.
- A **protected** elérhetőség azt jelenti, hogy az osztály leszármazottjai elérhetik-e a tagokat – függetlenül attól, hogy melyik csomagban vannak.
- A **public** pedig azt jelenti, hogy az összes osztály elérheti-e a tagokat.

Minden osztályban van legalább egy **konstruktor**. A konstruktor **inicializálja az új objektumot**. A neve ugyanaz kell, hogy legyen, mint az osztályé. A konstruktor nem metódus, így **nincs visszatérő típusa**. A konstruktor a **new operátor hatására hívódik meg**, majd visszaadja a létrejött objektumot. Nem kell konstruktorokat írni az osztályainkhoz, ha úgy is el tudja látni a feladatát. A rendszer automatikusan létrehoz egy paraméter nélküli konstruktort, különben nem tudnánk példányt létrehozni.

Ábécé, szó, nyelv, nyelvtan fogalma

Szimbólumok tetszőleges nemüres, véges halmazát ábécének nevezzük, és V-vel jelöljük. A V elemeit az **ábécé** betűinek mondjuk.

V-beli betűkből **felírható véges hosszúságú sorozatok**, az úgynevezett V feletti nem üres **szavak** halmazát. Egy p **szó** hosszát | p | -al jelöljük.

Szokás beszélni az úgynevezett **üresszóról** is, ami egy olyan szót jelent, melynek egyetlen betűje sincs, vagyis az egyetlen betűt sem tartalmazó betűsorozatot. Jelölése a Λ

A **V** ábécé feletti szavak egy **tetszőleges L halmazát** (formális) **nyelvnek** nevezzük, vagyis a **V*** halmaz részhalmazait V feletti formális nyelveknek, vagy röviden V feletti nyelveknek, vagy csak **egyszerűen nyelveknek hívjuk**. Egy **nyelvet üresnek, végesnek** vagy **végtelennek** hívunk, ha az L (mint halmaz) üres, véges, illetve végtelen.

Azt a nyelvet, **amelynek egyetlen szava sincs, üres nyelvnek nevezzük**. Jelölés: \emptyset . Nem tévesztendő össze a $\{\lambda\}$ nyelvvel, amely egyedül az üresszót tartalmazza.

Egy $W=(V, Ax, H)$ generatív rendszerből úgy kapunk **generatív nyelvtant**, ha a V ábécét felbontjuk közös elemet nem tartalmazó nem üres N és T részhalmazokra az Ax pedig egyetlen, egybetűs, N- beli (általában S- el jelölt) elemből áll, s minden H- beli szabály baloldala legalább egy N- beli betűt tartalmaz.

Chomsky-féle nyelvtani osztályok és az általuk generált nyelvosztályok tartalmazási hierarchiája.

Chomsky-féle hierarchia a generatív nyelvtanokat négy csoportra osztja:

- **0-s típusú (általános vagy mondszerkezetű)**, ha semmilyen megkötést nem teszünk a helyettesítési szabályaira.
- **1-es típusú (környezetfüggő)**, ha minden szabálya $\alpha A \gamma \rightarrow \alpha \beta \gamma$ alakú, ahol $A \in N$, $\alpha, \gamma \in (N \cup T)^*$, $\beta \in (N \cup T)^+$. Ezenkívül megengedhető az $S \rightarrow \lambda$ szabály is, ha S nem szerepel egyetlen szabály jobb oldalán sem.
- **2-es típusú (környezetfüggetlen)**, ha minden szabálya $A \rightarrow \beta$ alakú, ahol $A \in N$, $\beta \in (N \cup T)^+$. Ezenkívül megengedhető az $S \rightarrow \lambda$ szabály is, ha S nem szerepel egyetlen szabály jobb oldalán sem.
- **3-as típusú (reguláris)**, ha szabályai $A \rightarrow aB$ vagy $A \rightarrow a$ alakúak, ahol $a \in T$ és $A, B \in N$. Ezenkívül megengedhető az $S \rightarrow \lambda$ szabály is, ha S nem szerepel egyetlen szabály jobb oldalán sem.

Logikai áramkörök

A számítógépek digitális **áramkörökből** épülnek fel. A digitális áramkör két logikai értékkel dolgozik, a 0-val és az 1-gyel. Ezeket a logikai értékeket az elektronikusan működő gépekben jellemzően feszültségértékekkel reprezentálják. Az áramkörök és kapuk működését a bemenetek és kimenetek összes kombinációjának megadásával egyértelműen meghatározhatjuk.

Az öt logikai kapu a **NEM (NOT)**, a **NEM-ÉS (NAND)**, a **NEM-VAGY (NOR)**, az **ÉS (AND)** és a **VAGY (OR)** kapuk. A kapukból épülő áramkörök leírására alkalmas a Boole-algebra.

A NAND és NOR kapuk két tranzisztorból épülnek fel, míg az AND és OR kapuk háromból. Emiatt a gyártáskor a NAND és NOR kapukat preferálják. A NAND és NOR kapuk teljesek, azaz bármely Boole-függvény kiszámítható velük.

Kombinációs logikai hálózatok

A **kombinációs áramkörök** olyan áramkörök, melyeknek többszörös bemeneteik, többszörös kimeneteik vannak, és a pillanatnyi bemenetek határozzák meg az aktuális kimeneteket, tehát nincs memóriaelem az áramkörben.

A **multiplexernek** n vezérlőbemenete van, egy adatkimenete és 2^n adatbemenete. A multiplexer feladata az, hogy a több bemenetére érkezőjelből egyet vezessen a kimenetére. A multiplexer címezése mindig kettes számrendszerben történik és mindig annyi bemenőcsatornája van, amennyi különféle értéket a címbitek felvehetnek.

A multiplexer fordítottja a **demultiplexer**, amely egy egyedi bemenő jelet irányít a 2^n kimenet valamelyikére az n vezérlővonal értékétől függően, a többi kimenet 0. Ha a vezérlővonalak bináris értéke k , a k -adik kimenet a kiválasztott kimenet.

A **dekódoló** működését tekintve egy kapcsoló, amely az n -bites bemenete által kiválasztott 2^n -kimenet közül aktivál egyet. Ez azt jelenti, hogy itt 1 lesz a kimenet, az összes többi kimeneten pedig 0.

Egy számítógép egyik alapvető művelete az összeadás. Ezt **félösszeadók** segítségével végzik el a gépek. A félösszeadó azért fél, mert nem kezeli az előző helyi értékről érkező átvitel (carry) bitet.

Egy **teljes összeadó** két fél összeadóból épül fel. A két fél összeadó együtt számolja ki az összeg- és az átvitelbiteket.

7. TÉTEL

Öröklődés, túlterhelés, polimorfizmus

Az **objektum-orientált programozás** a „dolgozat” („objektumokat”) és köztük fennálló kölcsönhatásokat használja alkalmazások és számítógépes programok tervezéséhez. Az OOP olyan megoldásokat foglal magában, mint a **bezárás** (encapsulation), a **többalakúság** (polymorphism) valamint az **öröklés** (inheritance).

Egy programnyelv objektum-orientáltságát az említett elvek támogatásával lehet elérni.

Bezárás, adatrejtés (encapsulation): Lehetnek olyan jellemzők és metódusok, melyeket elfedünk más objektumok előtt. Az adatok és a metódusok osztályba való összezárását jelenti. Tulajdonképpen az objektum egység bezárja az állapotot (adattagok értékei) a viselkedésmóddal (műveletekkel). Következmény: az objektum állapotát csak a műveletein keresztül módosíthatjuk. Minden objektum egy jól meghatározott interfészt biztosít a külvilág számára, amely megadja, hogy kívülről mi érhető el az objektumból.

Öröklés (inheritance): Segítségével egy általánosabb típusból (ősosztály) egy sajátosabb típust tudunk létrehozni (leszármazott osztály). Az leszármazott osztály adatokat és műveleteket (viselkedésmódot) örököl, kiegészíti ezeket saját adatokkal és műveletekkel, illetve felülírhat bizonyos műveleteket. A kód újra felhasználásának egyik módja. Megkülönböztetünk egyszeres és többszörös örökítést. Javában csak egyszeres öröklődés van, azaz egy osztály legfeljebb egy őosztályt örökölhet.

Javában az *Object* osztály az osztályhierarchia legfelső eleme, minden más osztály belőle származik.

Polimorfizmus (polymorphism): Lehetővé teszi, hogy az öröklés során bizonyos viselkedési formákat (metódusokat) a származtatott osztályban új tartalommal valósítsunk meg, és az új, lecserélt metódusokat a szülő osztály tagjaiként kezeljük. A gyerekosztály egy példánya kezelhető a szülő egy példányaként is. Egy ős típusú tömbben eltárolhatjuk az ős- és gyerektípusokat vegyesen. Azonban, ha ős típusként tárolunk egy gyerek típusú objektumot, akkor a gyerek típusú objektum saját osztályában definiált metódusait nem látjuk.

Kivételkezelés

A **kivételkezelés** egy [programozási](#) mechanizmus, melynek célja a [program](#) futását szándékosan vagy nem szándékolt módon megszakító esemény (hiba) vagy utasítás kezelése. Az eseményt magát **kivétel**nek hívjuk. A hagyományos, szekvenciális és strukturált programozási kereteken túlmutató hibakezelésre, valamint magasabb szintű hibadetekcióra, esetleg korrigálásra használható.

A OO nyelveknél try-catch blokkot használunk kivételkezelésre:

```
try      {blokk}  
catch    {hibakezelő blokk}  
finally  {mindenképpen lefutó blokk}
```

Nézettáblák

A **nézettáblázat** az adatbázisban létező reláción vagy relációkon végrehajtott művelet eredményét tartalmazó olyan új táblázat, amely valóságban nem létezik fizikailag az adatbázisban. Akár több táblázatból is vehetünk oszlopokat a nézettáblába.

A nézettábla csak egy utasítás sorozatot tárol el, amely meghatározza, hogy a teljes adatbázis mely részhalmazát kell megjelenítenie, tehát valójában **nem tárol adatot**, ha az adat az eredeti táblában megváltozik, akkor a nézettábla is változik vele. A nézettáblák alkalmazásával növelhetjük az adatbázis biztonságát és egyszerűsíthatjuk a felhasználók munkáját. Biztonsági szempontból nézve, lehetővé tehetjük azt, hogy minden felhasználó csak annyit láthasson az adatokból, amennyi az ő hatáskörébe tartozik, illetve amennyivel dolgoznia kell.

Nézettáblát a **CREATE VIEW** utasítás segítségével készíthetünk. A nézettáblák törlése a **DROP VIEW** paranccsal történik.

Indexelés

Az indexelés célja a lekérdezések gyorsítása.

A keresés nagyságrenddel gyorsítható, ha a vizsgált tábla rekordjai a WHERE feltételek szempontjából legalább részlegesen rendezettek (indexelve). Az első, egyszerűen csak INDEX-ként emlegetett mechanizmus **B-fa** (B-Tree) típusú indexstruktúrát hoz létre a tábla rekordjai felett, a kulcsmezők alapján. Az index segítségével az adatbázis-kezelő rendszernek nem kell a tábla minden sorát végig néznie, hanem annak csak egy töredék részét. Az index alapjául szolgáló oszlopokat indexkulcsoknak nevezzük.

Egy tábla létrehozása után alaphelyzetben semmilyen indexeléssel sem rendelkezik kivéve, ha a tábla valamelyik mezője UNIQUE vagy PRIMARY KEY típusú. Ekkor az ilyen mezőkhöz INDEX-et hoz létre. Az indexek lehetnek a kulcsérték szerint rendezettek, lehetnek összetettek, ebben az esetben a kulcs összetett, több oszlopból tartalmaz adatokat.

Nincs általánosan legjobb optimalizáció. Az egyik cél a másik rovására javítható (például indexek használatával csökken a keresési idő, nő a tárméret, és nő a módosítási idő). Olyan mezőket érdemes INDEX struktúrával ellátni, melyeknek nagy a szelektivitás, és amelyekre gyakran hajtunk végre lekérdezést.

Az adatbázis-tervezés elmélete

Funkcionális függésről akkor beszélünk, ha egy tábla valamelyik mezőjében lévő érték meghatározza egy másik mező értékét. Például: *Irányítószám - Város*

A funkcionális függés kiterjesztése a **teljes funkcionális függés**. Amikor egy adatbázist "normalizálunk", arra törekszünk, hogy minden táblában teljes funkcionális függések legyenek.

Részleges funkcionális függés a teljes funkcionális függés egyik akadály. Akkor fordulhat elő egy táblában, ha abban van összetett kulcs és nem teljesül a teljes funkcionális függés alábbi feltétele. Részleges funkcionális függés csak akkor fordulhat elő egy táblában, ha abban összetett kulcs van. A normalizálás során a részleges funkcionális függést meg kell szüntetni!

Tranzitív függés esetén minden, nem kulcs mező függ a kulcstól, de van olyan mező, esetleg mezők, amely a kulcson kívül más mezőtől is függnek. A teljes funkcionális függés egyik feltétele, hogy minden nem kulcs mező csak a kulcstól függjön.

Az adatbázisok kialakításakor egyik legfőbb feladatunk, a redundancia-mentes adatszerkezet kialakítása. Ennek egyik módja a **normalizálás**. A normalizálás során egy kezdeti állapotból több fázison keresztül átalakítjuk az adatbázist. Az átalakítás fázisait

normálformáknak nevezzük. Megkülönböztetjük a nulladik, az első, második, harmadik, negyedik, ötödik, és hatodik normálformát. A normálformák jelölésére az 0NF, 1NF, 2NF, 3NF...jelöléseket használjuk.

Egy relációséma **első normálformában** van, ha az attribútum-halmaz minden eleme atomi, vagyis nem többértékű és nem összetett attribútum.

- Az összetett attribútumokat a részattribútumaikkal helyettesítjük.
- A többértékű attribútumokat célszerűen új relációsémába vesszük fel, amelyhez külső kulcsként hozzávesszük az őt tartalmazó relációséma kulcsát.

A **2NF** előfeltétele, hogy adatbázisunk minden táblája legalább 1NF-ben legyen. 2NF-ben vagyunk akkor, ha 1NF-ben vagyunk, és a táblákban **megszüntetjük a részleges funkcionális függéseket**. A 2NF kialakításakor azt a táblát, amiben részleges funkcionális függés van, két új táblára bontjuk. Az egyik táblába az összetett kulcs egyik eleme kerül, a tőle függő összes mezővel együtt. A másik táblába a kulcs másik eleme kerül a tőle függő összes mezővel együtt.

A **3NF** előfeltétele, hogy adatbázisunk minden táblája legalább 2NF-ben legyen. 3NF-ben vagyunk akkor, ha 2NF-ben vagyunk, és a táblákban **megszüntetjük a tranzitív függéseket**. Minden tranzitív függést tartalmazó táblából két táblát csinálunk. Új táblába kerülnek a tranzitív függésben lévő mezők, azzal a tranzitív kulcs mezővel együtt, amelytől a kulcson kívül függnék. Az új táblában a tranzitív kulcs mező lesz az azonosító.

A harmadik normál forma egy szigorúbb definíciója, a **Boyce-Codd-féle normálforma**. Feltétele, hogy a reláció harmadik normál formában van és minden elsődleges attribútum teljes funkcionális függőségben van azokkal a kulcsokkal, melyeknek nem része. El kell tüntetnünk a nem kedvező függéseket a relációból

Nem normalizált adatbázisok esetén **beszúrási, törlési és módosítási anomáliákkal** találkozhatunk, ezeket összesítve **karbantartási anomáliáknak** nevezzük.

Beszúrási anomália, amikor egy adatrekord beszúrása egy másik, hozzá logikailag nem kapcsolódó adatcsoport beszúrást kívánja meg. Felesleges, már letárolt információkat is újra fel kell vinni.

Törlési anomália, amikor egy adat törlésével másik, hozzá logikailag nem kapcsolódó adatcsoportot is elveszítünk, törlési anomáliáról beszélünk.

Módosítási anomália, amikor egy adat módosítása több helyen történő módosítást igényel. Egy sor adatmódosításával a többi sor nem változik, az adatbázis inkonzisztenssé válik.

E/K modell és átfordítása adatmodellé

Az **egyed-kapcsolat** modell egy grafikus leíró eszköz, mely diagram segítségével adja meg az adatbázis szerkezetét (struktúráját). Egyedosztályok, kapcsolatok, típusok, egyéb feltételezések ábrázolása.

Egyed: egy valós világban létező dolog, amelyet tulajdonságokkal írunk le.

Kapcsolat: két egyed közötti reláció. Típusai:

- **1:1 kapcsolat:** minden egyedhez legfeljebb egy másik egyed tartozhat
- **1: N kapcsolat:** minden egyedhez több egyed tartozhat
- **N: N kapcsolat:** több egyedhez több másik fajta egyed tartozhat

Az E-K modell minden egyedéhez felírunk egy relációsémát, amelynek neve az egyed neve, attribútumai az egyed attribútumai, kulcsa az egyed kulcs-attribútumai. A séma feletti adattábla minden egyes sora egy egyedpéldánynak felel meg.

Az átalakítás előtt a tulajdonságokat átnevezhetjük, hogy a relációsémában ne szerepeljen kétszer ugyanaz az attribútum.

Nyomkövetés és hibakeresés, egységtesztelés, naplózás

A hibakeresési eszközök a programozási környezet olyan elemei, amelyek a hiba okának megállapítását, a hiba helyének megkeresését teszik könnyebbé azzal, hogy futás közbeni információt szolgáltatnak a programról.

Kiírás

A legegyszerűbben használható eszköz, amellyel egyszerűen adatkírást hajtunk végre. Kétféle fajtája lehet: az egyikben a programszöveg bizonyos helyeire helyezünk el tesztíráásokat. Ha a futás során arra a pontra érünk, akkor a benne szereplő változókat a program kiírja, majd várakozik a továbbindításra. A másikban a kiírandó változókat rögzítjük, és értékük a futás során mindig látszik/megnézhető a képen. Ez utóbbi gyakran más eszközökkel (pl. töréspont) kombináltan jelenik meg.

Nyomkövetés

A nyomkövetés lényege a végrehajtott utasítások követése a programban. Itt tehát futás során az eredményképernyő mellett a programszöveget is látnunk kell. A programszövegből vagy az éppen végrehajtott utasítást látjuk, vagy a teljes programszövegben mutatja egy mutató az aktuális utasítást, vagy pedig algoritmikus struktúrák végrehajtását figyelhetjük meg. A nyomkövetés általában sokféle információt adhat a program futásáról: a végrehajtott utasítás mellett kiírhatjuk a képernyőre annak hatását (értékadásnál a változóba elhelyezett értéket, elágazás - vagy ciklusfeltétel kiértékelésénél annak igaz vagy hamis értékét).

Adatok nyomkövetése

A nyomkövetés egy speciális változatában nem az utasításokat vizsgáljuk, hanem a változókat. Ebben az esetben akkor kapunk a képernyőn üzenetet, ha a kijelölt változó(ka)t valaki használja, illetve módosítja.

Töréspontok elhelyezése

A töréspontok a program olyan utasításai, amelyeknél a végrehajtásnak meg kell állnia, a felhasználó információt szerezhet a program állapotáról, majd folytatódhat a végrehajtás. Leálláskor a felhasználó dönthet a futtatás abbahagyásáról, illetve folytatásáról. Megnézheti változók értékeit, nyomkövetést be- és kikapcsolhat, töréspontokat megszüntethet, illetve újakat definiálhat stb. Sőt némely környezet azt a – nem veszélytelen – lehetőséget is biztosítja, hogy egy-egy változó értékét módosítsuk, és újabb elindítás nélkül lokalizálhassunk esetleges más hibákat is.

Lépésenkénti végrehajtás

Ez tulajdonképpen olyan eszköz, amely a program minden utasítására egy töréspontot definiál. A program minden utasításának végrehajtása után lehetőség van a töréspontoknál ismertetett beavatkozásokra.

Egységtesztelés

Az egységtesztelés (unit testing) egy olyan ellenőrző programrészlet, amely vizsgálja az adott metódusok visszatéréseinek lehetőségeit. Egy metódusra több tesztet is lehet írni, amivel több végkimenetelt tudunk ellenőrizni. Ezekre a kimenetlekre tudunk számítani, ezzel megbízhatóbb programot írhatunk. Sajnos minden lehetőségre nem lehetséges tesztet írni, de nem is szükséges, elég csak a kritikus metódusok elvárt kimenetelére (elvárt eredmény, továbbá felismert kivételek) tesztelni.

Naplózás

A naplózást (logging) általában fontosabb eseményekhez vagy állapotok leírásához használják fel. A naplózott eseményeket vizsgálva is látni lehet a program működését, esetleges hibáit. A naplózást fel lehet használni hibakeresés (debugging) során, amit röktön a konzol felületen tüntetünk fel, vagy akár fájlba is írhatjuk egy működő program, rendszer során.

Kollekciók használata

A kollekciók meglévő adatszerkezetek tulajdonságait, szabályosságait (listában sorrendiség, halmazban egyediség, stb) felhasználva foglal össze egyedi elemeket, objektumokat.

Javában:

- List (interface) – elemeket tárol adott sorrendben, iterálható módon.
Megvalósító osztályokra példa:
 - ArrayList: Tömbbel megvalósított lista
 - LinkedList: Láncolt lista
- Set (interface) – egyedi elemeket tárol sorrendiség nélkül. a tárolt elemeknek felül kell definiálnia az equals metódust

Megvalósító osztályokra példa

- HashSet: gyors keresést biztosító halmaz
- TreeSet: Rendezett halmaz (fával megvalósított)
- Map(interface) – kulcs-adat objektum párok csoportja, a kulcsra gyors keresést biztosít.

Megvalósító osztályokra példa:

- HashMap: hash táblával implementált
- TreeMap: piros-fekete fával implementált

A kollekciók használatához metódusok is járnak, amikkel elemet vagy elemeket tudunk hozzáadni, törölni, keresni, és ahol lehetséges rendezni.

Relációs adatbázisok kezelése OO programozási nyelvekben

Az objektum orientált nyelvek biztosítanak csatlakozási lehetőséget adatbázis rendszerekhez különböző ún. connectorokkal. Minden adatbázis rendszerhez különböző csatlakozó van, ezeket csak be kell építeni a programunkba. A csatlakozáshoz meg kell adni a DBMS felhasználó paramétereit (username, password) esetleg a DBMS server elérhetőségét/címét.

A csatlakozás után lehet SQL utasításokat írni, és végrehajtani a programon belül. A lekérdezéseket hasonlóan lehet kezelni, mint a fájlokat – soronként/rekordonként olvassuk be és így hozhatunk létre paraméterezett példányokat. Ezekből a példányokból pedig egy kollekciókba foglalhatjuk.

Az objektum orientált programozási nyelvek igyekeznek egységben tartani az összefüggő adatokat, míg a relációs adatbázisok külön táblákba csoportosítanak adatokat, amik közt kulcsokkal tartja fent a kapcsolatot. Erre a különbözőségekre adnak megoldást a különböző ORM (Object-Relational Mapping) rendszerek, amelyek bizonyos kereteken belül, automatikusan oldja meg a konverziókat az adatbázis táblái és az objektum zártsága között. A programozó szempontjából csak objektumokkal való műveleteket kell végrehajtani, nem kell a különböző SQL utasításokkal foglalkozni. Java-ban az egyik ilyen keretrendszer a Hibernate.

8. TÉTEL

Osztály- és példány inicializálás

```
public class Number { int a; int b; }
```

Akármennyi osztály létrehozható. Az osztály létrehozása a **new** paranccsal az osztály 1 db futó példányát, objektumát eredményezi.

```
public static void main(String[] args) { Number number = new Number(); }
```


Az osztály konstruktorán keresztül történik. A példányváltozó az 'a' é a 'b'. Automatikusan 0 értéket kap. Ezen változók a metódusok belépési pontjai (az alábbi kódban `int a` és `int b`). Hatókörük csakis a metóduson belül érvényes.

Konstruktor

- Beállítja az objektum kezdeti állapotát,
- Adatok, kapcsolatok
- Hasonlít a metódushoz, de nincs visszatérési értéke
- Neve megegyezik az osztály nevével
- Csak a new operátorral hívható
- Csak a hozzáférési módosítók használhatók
- Túlterhelhető
- Nem öröklődik

Minden osztályban van legalább egy konstruktor. A konstruktor inicializálja az új objektumot. A neve ugyanaz kell, hogy legyen, mint az osztályé. A konstruktor nem metódus, így nincs visszatérő típusa. A konstruktor a new operátor hatására hívódik meg, majd visszaadja a létrejött objektumot. Nem kell konstruktorokat írni az osztályainkhoz, ha úgy is el tudja látni a feladatát. A rendszer automatikusan létrehoz egy paraméter nélküli konstruktort, különben nem tudnánk példányt létrehozni.

ALAPÉRTTELMEZETT KONSTRUKTOR	
Alkalmazott a = new Alkalmazott(); a.nev = "GIPSZ JAKAB"; a.fizetes = 500000; a.fizetesEmel(6300);	public class Alkalmazott{ String nev; int fizetes; void fizetesEmel(int novekmeny){ fizetes += novekmeny; } }

Ha egy osztály több konstruktort definiál, akkor az egyikből meghívható egy másik konstruktor a this referencia segítségével.

Interfészek

Az interfész olyan viselkedéseket definiál, amelyet egy tetszőleges osztállyal megvalósíthatunk. Egy interfész metódusok halmazát definiálja, de nem valósítja meg azokat. Egy konkrét osztály megvalósítja az interfészt, ha az összes metódusát megvalósítja.

Az interfész implementáció nélküli metódusok névvel ellátott halmaza. Mivel az interfész a megvalósítás nélküli, vagyis absztrakt metódusok listája, alig különbözik az absztrakt osztálytól. A különbségek:

- Az interfész egyetlen metódust sem implementálhat, az absztrakt osztály igen.
- Az osztály megvalósíthat több interfészt, de csak egy ősosztálya lehet.
- Az interfész nem része az osztályhierarchiának. Egymástól „független” osztályok is megvalósíthatják ugyanazt az interfészt.

Az interfész deklarációban két elem kötelező: az interface kulcsszó és az interfész neve. Ez után szerepelhetnek a szülőinterfészek. Az interfész törzs metódus deklarációkat tartalmaz ;'-el lezárva. Minden deklarált metódus alapértelmezetten publikus és absztrakt. Nem használhatók a transient, volatile, synchronized, private és protected módosítók.

Generikus programozás

A generikus programozás egy általános programozási modellt jelent. Maga a technika olyan programkód írását foglalja magába, amely nem függ a program egyes típusaitól. Ez az elv növeli az újra felhasználás mértékét, hiszen típusoktól független tárolókat és algoritmusokat lehet a segítségével írni. Például egy absztrakt adatszerkezetet (mondjuk egy láncolt listát) logikus úgy tervezni és megvalósítani, hogy bármi tárolható lehessen benne. A Java generic-ek, azaz generikus osztályok valamilyen típussal paraméterezett osztályok. **Példa:**

```
public interface List<E>{    void add(E x); Iterator<E> iterator(); //....}
public interface Iterator<E>{ E next();    boolean hasNext();    //....}
```

A fenti példában „E” formális típusparaméter, amely aktuális értéket a kiértékelésnél vesz fel. (Pl.: Integer). Példa a használatukra:

```
List<String> l1 = new ArrayList<String>();
List<Object> l2 = l1;    //hiba
```

```
//Mert akkor lehetne ilyet is csinálni
l2.add(new Object());
l1.get(0); // törés, Object -> String csatolás
```

Összetett adatszerkezeteket implementáló osztályok és fontosabb műveleteik

- TÖMB
- ARRAYLIST - Tömbbel megvalósított lista, gyors elérés, de lassú beszúrás/törlés.
- LIST - Elemeket tárol adott sorrendben ListIterator, oda-vissza be lehet járni a listát és be lehet szűrni elemeket a listába elemek közé is.
- SET - Egyedi elemeket tárol sorrendiség nélkül.
- MAP - Kulcs-adat objektum-párok csoportja, a kulcsra gyors keresést biztosít.

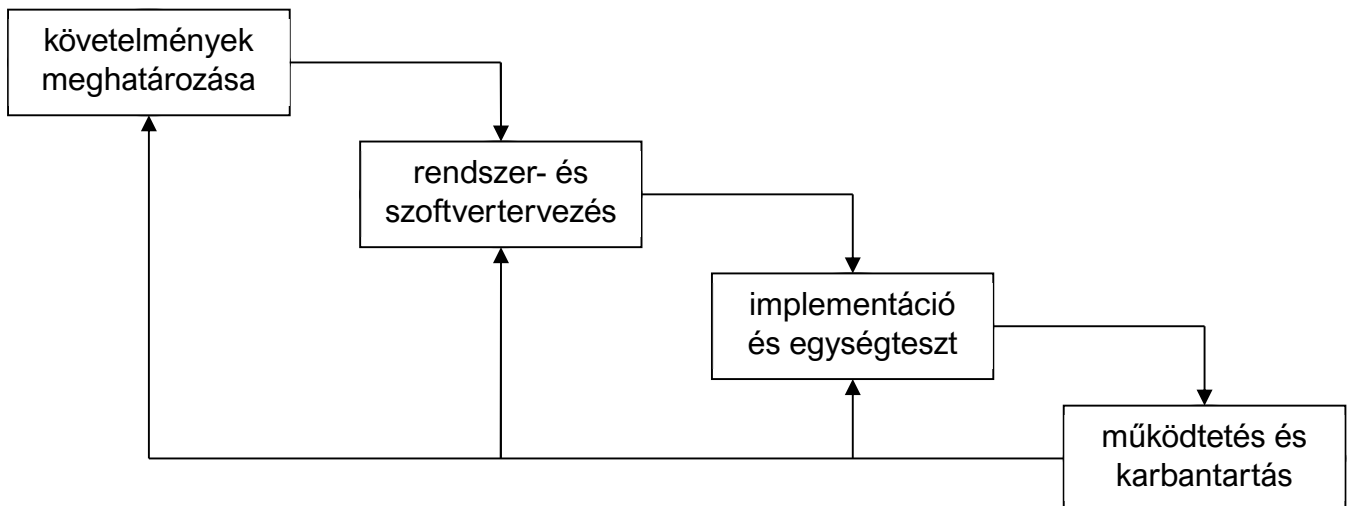
Collection(List,Set) funkcionalitása	
boolean add(Object)	elem hozzáadása
boolean addAll(Collection)	elemek hozzáadása
void Clear()	minden elem törlése
boolean contains(Object)	true, ha tartalmazza az elemet
boolean containsAll(Collection)	true, ha tartalmazza az elemeket
boolean isEmpty()	true, ha üres
Iterator iterator()	készít egy iterátort
boolean remove(Object)	elem törlése
boolean removeAll(Collection)	elemek törlése
boolean retainAll(Collection)	csak akkor tartja meg az elemet ha az szerepel a paraméterben (metszet)
int size()	elemek száma
Object[] toArray()	felépít egy tömböt, amely tartalmazza a kollekció elemeit
Object[] toArray(Object[])	mint az előző csak a paraméterben kapott tömb futás idejű típusának megfelelő típusú tömböt épít fel.
MAP funkcionalitása	
void Clear()	töröl minden elemet
boolean containsKey(Object)	true, ha tartalmazza a kapott kulcsú elemet
boolean containsValue(Object)	true, ha tartalmazza a kapott tartalmú elemet
Set entrySet()	visszaadja halmazként a tárolt adatokat
Object get(Object key)	a kapott kulcsú elemet adja vissza
boolean isEmpty()	true, ha üres
Set keySet()	visszaadja halmazként a tárolt kulcsokat
Object put(Object key, Object value)	beszúr egy új elemet (párosít)
void putAll(Map)	beszúrja a kapott MAP elemeit
Object remove(Object key)	törli a kapott kulcsú elemet
Collection values()	visszaadja kollekcióként a tárolt elemeket.

Rendszerfejlesztési modellek (tervezés,tesztelés)

Vízesés modell, evolúciós modell, formális fejlesztési modell, újra felhasználás-alapú modell, iteratív modellek (spirális fejlesztési modell | inkrementális fejlesztési modell).

Vízesés modell

A vízesésmodell a nevét onnan kapta, hogy a folyamatlemek, az egyes lépések meglehetősen mereven egymásra épülnek, ilyen lépcsőzetesen kapcsolódnak egymáshoz, mint ahogy a víz folyik le a sziklán. Ez a modell előnyeit és hátrányait egyaránt magába foglalja.

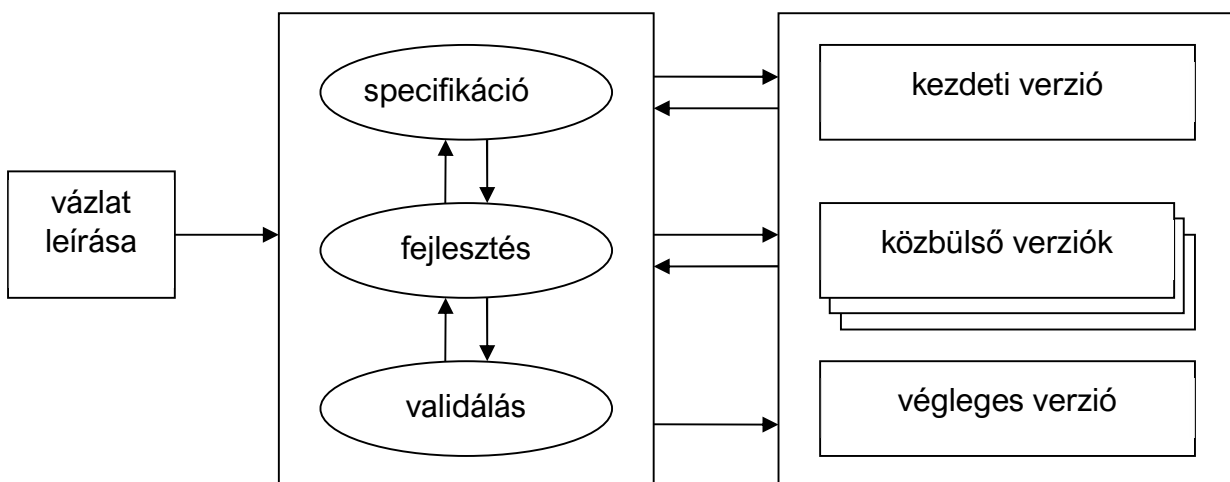


Előnyei: A legegyszerűbb, a legjobban alkalmazható modell. Kicsi és közepes méretű rendszereknél jól strukturált, jól felépített, jó architekturális jellemzőkkel rendelkező, robusztus rendszer fejleszthető.

Hátrányai: Merev, nem flexibilis egymásra épülés. Minden egyes folyamat teljes körűen lezárul, mielőtt a következő folyamat elindulna. A legutolsó lépésben derülnek ki a korai lépések problémái: nem teljesített követelmények, félreértések stb.

Evolúciós modell

Adott egy kezdeti implementáció, majd annak vannak valamilyen változatai és végül megszületik a végleges verzió. Ezeket a verziókat hívja a modell prototípusoknak. Tehát ez a prototípusorientált vagy prototípusalapú modell. A párhuzamosság és a verziókezelés nagyon gyors visszacsatolási lehetőséget jelent, gyors beavatkozást tesz lehetővé. Tehát nem a végén derülnek ki a hiányosságok, hanem menet közben.



Feltáró fejlesztés

A prototípusok segítségével a felhasználó és a fejlesztő közösen finomítja, pontosítja a követelményeket, tehát prototípus sorozaton keresztül tárjuk fel a követelményeket. Tehát a felhasználó kap egy működő verziót, ezek alapján pontosítja a követelményeket, a fejlesztésnél ezt figyelembe vesszük és a következő verzió már jobban teljesíti a követelményeket. Így a verziósorozat egyes verzióiban egyre több felhasználói igény kerül beépítésre, míg a végső verzió már a tényleges követelményeknek megfelelő verzió lesz. Ez lesz a rendszer. Ilyenkor az első prototípusoknak a rendszer azon részeivel kell kezdődnie, amelyek tiszták, világosak, amelyeknek a követelményei egyértelműek, a rendszer ismert részeinek kifejlesztésével kezdődik. A kevésbé ismert, zűrösebb rendszerelemek a későbbi prototípusokba kerülnek bele. Minden prototípus az előzőből következik, az előző prototípusra épül, a végleges változat prototípus sorozat alapján készül el.

Eldobható prototípuskészítés

Adunk a felhasználó kezébe egy verziót, és ennek segítségével a felhasználó kipróbálja a prototípust, így rájöhet, hogy mit akar. Ha arra jön rá, hogy nem ezt akarja, a prototípust eldobjuk, újra fogalmazzuk a követelményrendszert, és az ennek megfelelő prototípust állítjuk elő. Ebben a megközelítésben az első prototípusok a legkevésbé megértett, legkevésbé feltárt rendszerelemekkel kezdődnek azért, hogy a felhasználó kitalálja, mit akar.

Előnyei: Nagyon hamar kap kipróbálható verziók. A felhasználó hamar rájöhet, hogy a követelmények jók vagy nem.

Hátrányai: Túl sok verzió áttekinthetetlenné válhat. Nem robusztusak: általában rossz strukturáltságú, architektúrájú szoftverhez vezet. Gyors fejlesztést, rapid technikákat igényel.

Formális modell

A 70-es években alakult ki a vízesés modell egy változataként.

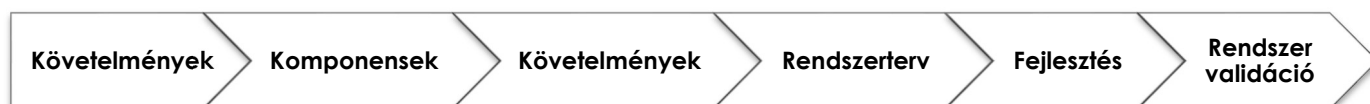


Előnye: Automatizált a folyamat; bármilyen rendszer esetén alkalmazható. Biztonságkritikus rendszerek esetén alkalmazzák.

Hátránya: A rendszerkövetelmények formalizálása kézzel kell, hogy történjen.

Újra felhasználás-orientált modell

Léteznek olyan komponensek, amelyeket újra felhasználhatunk (kód, tervezési szinten)



Ha megvan a követelmények specifikációja, akkor elsőként megnézzük, hogy a világban vannak-e olyan szoftverek, netalán olyan tervelemek, amelyek nekünk jók, felhasználhatók a mi rendszerünkben; megkeressük ezeket az elemeket.

Problémák:

- Ø tudnunk kellene, hogy vannak ilyen komponensek
- Ø tudnunk kellene, hogy egyáltalán milyen komponensek vannak
- Ø meg kellene tudni találni ezeket a komponenseket

ha meg is találtuk, ezek nagy valószínűséggel nem felelnek meg teljes mértékben az igényeinknek.

Megoldásként módosítjuk a követelményeinket annak megfelelően, hogy a kész komponenseket be tudjuk építeni a rendszerünkbe. Aztán ennek megfelelően végrehajtjuk a tervezést és beépítjük a tervbe vagy az implementációba a komponenseket.

Hátrány: kompromisszumok, leromolhat a rendszer struktúrája

Előnyök: idő, pénz megtakarítása; a komponensek kipróbáltak, teszteltek stb., így egyfajta biztonságérzetet nyújt, sok validációs lépést megspórolhatunk.

Inkrementális modell:

Az inkrementális modell előnyei:

- az inkremensek kicsik (úgy kell megtervezni, hogy kicsik legyenek), ezekre önmagukban a legmegfelelőbb modell alkalmazható, például vízesésmodell is (az inkremensek kicsik, jól körülhatárolhatók, adott esetben a követelmények egyértelműen specifikálhatók).
- az inkremensek fejlesztése történhet párhuzamosan
- kezdhethetjük a fejlesztést a legfontosabb funkciókat realizáló inkremensekkel, és mivel az inkremensek kicsik, a felhasználó nagyon hamar kap egy nem eldobható prototípust, amely viszont már nem a követelményfeltáráshoz való, hanem már tudja használni. Tehát bizonyos funkciókat, a legalapvetőbb funkciókat tartalmazó rendszert nagyon hamar kap a felhasználó, a kevésbé lényeges funkciókat majd később beépítjük a rendszerbe. A rendszerválidálás mindig megtörténik, tehát a rendszer ilyen értelemben a beépített inkremensekkel önmagában egy részrendszer, tehát egy használható rendszer.
- a rendszerfejlesztés kockázata kisebb, mint az összes többi modellnél (a fejlesztések igen nagy része, több mint 50 százaléka sikertelen), mivel a rendszer validált, kis elemekkel,

tehát van egy validált működő rendszer még akkor is, ha befúlad a projekt, legfeljebb nem teljes funkcionalitással. Nagyon nagy a valószínűsége, hogy az első pár inkrementnél még nem fogy el a pénz, az idő, az ember, nem változik meg a környezet. Ezért a részsikeres befejezés valószínűbb ennél a fejlesztési modellnél.

- a fontosabb funkciókat implementáljuk először, ennek a következtében azokat a funkciókat a felhasználók rendszeresen tesztelik, a fontosabb funkciókat jóval többször használják, mint a kevésbé fontosakat. Így már a rendszerfejlesztés közben a hibák hamarabb kiderülnek, a mindenkorai részrendszerben valóban a legfontosabb funkciók a legteszteltebbek. Tehát egy robusztusabb rendszer fejlesztéséhez vezet ez a modell.

Spirális modell

- 1. lépés: célok meghatározása
 - 2. lépés: kockázatelemzés, kockázatok felismerése, megszüntetése
 - 3. lépés: fejlesztés és validálás
 - 4. lépés: áttekintés, döntés a folytatásról
-
- ✓ 1. lépés: a követelmények, pontosabban célok meghatározása: az adott spirális fázisban mit fogunk megvalósítani? azonosítjuk a tevékenységeket, meghatározzuk a megköveteléseket, a menedzselési tervet és azt, hogy milyen kockázati tényezők várhatók ebben a lépésben, és ezeket a kockázatokat hogyan fogjuk kezelni, milyen alternatívák lehetnek a kezelésre.
 - ✓ 2. lépés: kockázatelemzés, kockázatok felismerése, kockázatok tényezőinek megszüntetése, ezekre való felkészülés: megtervezzük azokat a lépéseket, amelyek azt célozzák, hogy a kockázatokat el tudjuk kerülni. (Ha például a követelmények meghatározása kritikus pont, tehát úgy gondoljuk, hogy a teljes fejlesztésben nagyon nagy kockázatot jelent, rosszak a követelmények, akkor egy feltáró evolúciós modellt alkalmazunk ebben a lépésben, építenünk kell egy prototípus sorozatot a követelmények minél pontosabb meghatározása érdekében.)
 - ✓ 3. lépés: fejlesztés és validálás: a 2. lépés után fejlesztési modellt választunk, adott esetben minden egyes spirálban más-más fejlesztési modell alkalmazható, adott esetben a kockázatanalízis eredményeként választható a fejlesztési modell. Ezek után hajtjuk végre a szokásos tervezés, implementáció, validálás lépéseket. Az első spirálok esetén nyilvánvalóan a követelményfeltárás, követelménytervezés és a validációs lépések következnek, amíg el nem jutunk odáig, hogy...
 - ✓ 4. lépés: tekintsük át a teljes projektet, az eddigi fejlesztést, az eddigi spirálokat, elemezzük ezeket, és döntsünk a folytatásról olyan értelemben, hogy megállunk-e (a megvalósíthatósági esettanulmány után dönthetünk úgy, hogy befejezzük). Amennyiben a folytatásról döntünk, a projektet tervezzük meg, azt tervezzük meg, hogy hogyan folytatódjon a projekt, és kezdődik előlről a spirál.

Előnyei: Foglalkozik a kockázatokkal. Sokkal szisztematikusabban foglalkozik a projekttel

Hátrányai: Nem a legtriviálisabb modell. Nagy rendszereknél javallott.

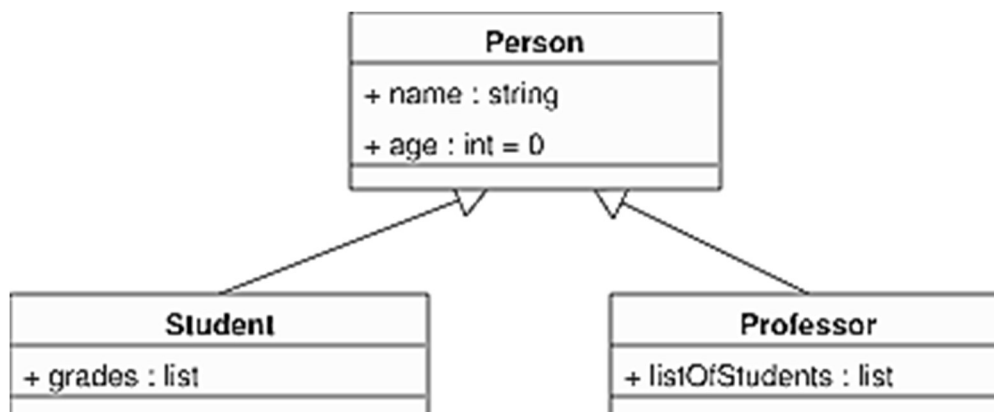
UML osztálydiagram

- Az UML egy grafikus modellező nyelv a szoftver-rendszer különböző nézeteinek modellezésére.
- Segítségével tervezni és dokumentálni tudjuk a szoftvereket.
- Az UML-ben modellek és diagramok adhatók meg, különböző nézetekben.
- Az UML „csak” egy jelölésrendszer, amely független a szoftverfejlesztési módszertől.
- Az UML nem írja elő, hogy az egyes modelleket, illetve diagramokat mily módon és milyen sorrendben állítjuk elő.

UML Modell diagramjai

- Használati eset diagramok.
- Osztály diagramok.
- Objektum diagramok.
- Szekvencia diagramok.
- Együttműködési diagramok.
- Állapot diagramok.
- Aktivitás diagramok.
- Komponens diagramok.
- Telepítési diagramok.

- Osztálydiagram (class diagram): Olyan diagram, amely az osztályokat és a közöttük lévő társítási és öröklési (általánosítási) kapcsolatokat ábrázolja.



Verziókezelés

Verziókezelés: Revision Control, Version Control, Source Control, Source Code Management (SCM)

Verziókezelő rendszerek: *Version Control System (VCS), Distributed Revision Control (DRCS)*

Verziókezelés alatt több verzióval rendelkező adatok kezelését értjük. Leggyakrabban a mérnöki tudományokban és a szoftverfejlesztésben használnak verziókezelő rendszereket fejlesztés alatt álló dokumentumok, tervek, forráskódok és egyéb olyan adatok verzióinak kezelésére, amelyeken több ember dolgozik egyidejűleg. Az egyes változtatásokat verziószámokkal vagy verzióbetűkkel követik nyomon. Minden verziókezelési művelet egy közösen használt szerveren történik

Számítógép-architektúrák – mikroelektronika, processzor és memória

A **félvezetők** gyengén vezetnek, fajlagos vezetőképességük a szigetelőkénel nagyságrendekkel nagyobb, de még mindig olyan kicsi, hogy az anyag gyakorlati szempontból szigetelőnek tekinthető. A leggyakrabban használt félvezető a szilícium (Si).

A **dióda** olyan elektronikai alkatrész, amelyet többségében egyenirányításra, illetve egyszerűbb logikai kapuáramkörökben is alkalmazható.

A **tranzisztor** egy szilárdtest félvezető, amelyet elektronikus áramkörökben használnak erősítési és kapcsolási célokra. A tranzisztor három, egymást felváltva követő különböző vezetési típusú tartományú, egymáson elhelyezkedő rétegből áll. Minden réteg ki van vezetve egy lábra. A két szélső réteget kollektornak (C), és emitternek (E) nevezik, a középső réteget bázisnak (B) hívják.

- **Bipoláris** tranzisztor - két, elektromosan szétválasztott (vagyis polarizált) rétegből áll
- **Unipoláris** tranzisztor - működésében egynemű töltéshordozók vesznek részt.

Tranzisztorok használatával megvalósítható az öt alapvető logikai kapu a NEM (NOT), a NEM-ÉS (NAND), a NEM-VAGY (NOR), az ÉS (AND) és a VAGY (OR) kapuk.

A **CPU** (Central Processing Unit) feladata az operatív tárban (memóriában) elhelyezkedő program feldolgozása és végrehajtása. A CPU sebességét a másodpercenkénti működési ciklusok (órajel periódusok) száma jelenti. Az egy működési ciklus során a CPU által feldolgozható adatok mennyisége a processzor adatbuszának szélességétől (az egyidejűleg átvitt bitek számától) függ. A CPU buszt rendszerbusznak (FSB front side bus) is hívják. Minél szélesebb a processzor adatbusza, annál nagyobb a processzor teljesítőképessége. A jelenlegi processzorok 32 vagy 64 bites adatbusszal rendelkeznek.

CPU főbb részei:

ALU: (Arithmetic and Logical Unit – Aritmetikai és Logikai Egység). A processzornak azon része, mely a számolási, összehasonlítási, logikai műveleteket végzi. Az ALU végrehajtási sebessége növelhető egy co-processor (**FPU**, Floating Point Unit, lebegőpontos műveleteket végző egység) beépítésével, ami egyes feladatokat gyorsabban hajt végre, mint az ALU.

CU: A processzor vezérlő egységének feladata a program utasításai, vagy külső kérések alapján, vezérlő jelek segítségével a gép részeinek irányítása.

Regiszterek: A processzorok ideiglenes adattárolási céljaira szolgálnak. A regiszterek a belső sínrendszeren keresztül tartanak kapcsolatot a processzor más részeivel.

Cache: A modern processzorok fontos része a cache (gyorsító tár). A cache a processzorba vagy a processzor környezetébe integrált memória, ami a viszonylag lassú rendszermemóriaelérést hivatott kiváltani azoknak a programrészeknek és adatoknak előzetes beolvasásával.

A **memória** közvetlen kapcsolatban van az aritmetikai egységgel és a vezérlőegységgel (CU).

A memóriákat fizikai szempontból két csoportra osztjuk:

- **ROM** (csak olvasható) típusú. Adatok írására nincs lehetőség. Csak olvasható.
- **RAM** (írható és olvasható) típusú memóriák. A futó programok ide töltődnek be.

Elsődleges, másodlagos és harmadlagos memória, a processzortól való távolság szerint. Kisebb szinteken rövidebb az elérési idő.

9. TÉTEL

Adatdefiníciós résznyelv (DDL)

Adatdefiníciós utasítások (Data Definition Language - DDL) amelyek objektumok létrehozására, módosítására, törlésére valók. (TABLE, DATABASE, TRIGGER).

- CREATE – egy objektum létrehozására
- ALTER – egy objektum módosítására
- DROP – egy objektum megszüntetésére

Táblák létrehozására a **CREATE TABLE** utasítást, módosítására az **ALTER TABLE** utasítást használjuk.

Egy **tábla létrehozásához** tehát meg kell adnunk a tábla nevét, majd zárójelek között felsorolni az oszlopait. Egy oszlop létrehozásához szükség van az oszlop nevére, az adattípusra és megadhatunk az adott oszlopra vonatkozó megszorításokat.

A megszorítások például:

- **NOT NULL** vagy **NULL** – lehet-e null értékű, vagy sem.
- **PRIMARY KEY** - Az adott oszlop a tábla elsődleges kulcsa lesz, megköveteli az adott oszlopban az értékek egyediségét.
- **UNIQUE** - Az adott oszlopban minden érték egyedi. Hatására index állomány jön létre.
- **DEFAULT** – Mi legyen az alapértelmezett értéke, ha nem kap értéket beszúrás során.

Az ALTER utasítások az objektumdefiníciók módosításra szolgálnak. Hozzáadhatunk, újra definiálhatunk vagy törölhetünk oszlopokat, megszorításokat.

Adatlekérdező nyelv (DQL)

Adatlekérdező nyelv (Data Query Language - DQL) amelyekkel a letárolt adatokat tudjuk visszakeresni.

A lekérdező nyelv egyetlen utasításból áll, ez pedig a **SELECT**, mely számos alparancsot tartalmazhat, és a lekérdező utasítások többszörös mélységben egymásba ágyazhatók. A bemeneti adatokon, a relációs algebra műveletei hajthatóak végre, aminek következményeként egy eredmény táblát kap a felhasználó.

Végrehajtási sorrendjük a következő: FROM, WHERE, GROUP BY, HAVING, SELECT, ORDER BY.

- **FROM:** Meghatározza, hogy mely adatbázis-táblákból szeretnénk összegyűjteni az adatokat.
- **WHERE:** Szűrési feltételeket fogalmaz meg, amelyek szűkítik az eredményhalmazt (a Descartes-szorzathoz képest)
- **GROUP BY:** Egyes sorok összevonását, csoportosítását írja elő az eredménytáblában.
- **HAVING:** A WHERE-hez hasonlóan itt is szűrést fogalmazhatunk meg, azonban itt a csoportosítás utáni eredményhalmazra.
- **ORDER BY:** Az eredményhalmaz rendezését adja meg.

Egy lekérdezésben egyszerre több táblából is lekérhetünk adatokat. A legegyszerűbb módja, ha a FROM után több táblát nevét írjuk. Ilyen esetben a táblákon a Descartes szorzat relációalgebrai művelet hajtodik végre és az eredményt a szorzat reláció adja, ez viszont nagyon ritkán használható.

Az SQL nyelv lehetőséget biztosít az összekapcsolás (JOIN) relációalgebrai műveletek közvetlen megvalósítására is. Ez azt jelenti, hogy speciális utasítások állnak rendelkezésre, amelyek az összekapcsolás különböző fajtáit adják meg.

<táblanév> JOIN <táblanév> ON <feltétel>

Az ON záradékban tetszőleges feltételt adhatunk meg.

Alapértelmezetten amikor JOIN-t írunk, az értelmező **INNER JOIN**-t hajt végre. Ekkor csak azok az adatok jelennek meg, amelyek mindkét táblában szerepelnek.

Az **OUTER JOIN** olyan szelekciós JOIN melyben az illeszkedő pár nélküli rekordok is bekerülnek az eredményhalmazba (üres értékekkel kiegészítve)

Adatmanipulációs nyelv (DML)

Adatmanipulációs nyelv (Data Manipulation Language - DML) amelyek rekordok felvitelére, módosítására és törlésére alkalmazhatók.

Egy **INSERT INTO** utasítás segítségével egy sor adható meg az adott relációhoz. Az attribútum nevek megadása csak akkor kötelező, ha nem minden attribútumhoz rendelünk értéket, vagy az attribútumok értékét nem a definiálás sorrendjében adjuk meg.

Az INSERT utasítás lehetőséget biztosít arra is, hogy a relációt egy másik relációból átvett értékekkel töltsük fel. Ekkor az értékek megadása helyén egy lekérdező utasítás állhat. A lekérdezés eredményei kerülnek be a megadott relációba, egyszerre akár több sor is.

Az **UPDATE** utasítás segítségével egyidőben a relációk több sorát is módosíthatjuk. A SET után adhatjuk meg a módosítandó attribútumot és értékeit. A WHERE után egy feltétel adható meg, az utasítás csak a reláció azon sorain dolgozik, amelyekre a feltétel értéke igaz. A WHERE rész el is maradhat, ekkor a reláció összes sorára vonatkozik az UPDATE parancs.

A relációk sorait a **DELETE** parancs segítségével törölhetjük. *DELETE FROM reláció_név [WHERE feltétel];* A feltételben az UPDATE parancshoz hasonlóan egy zárójelek közé tett lekérdező utasítás is megadható. A WHERE alparancs elmaradása esetén a reláció összes sora törlődik.

Adatvezérlő nyelv (DCL)

Adatelérést Vezérlő Nyelv (Data Control Language - DCL) amelyekkel az adatvédelmi és a tranzakció-kezelő műveletek végrehajthatóak.

Beágyazott allekérdezések

Az SQL nyelv lehetővé teszi a SELECT utasítások egymásba ágyazását, illetve később látni fogjuk, hogy más utasításokba is ágyazhatunk be SELECT utasításokat.

ALL: Ezzel a kulcsszóval felírt feltétel akkor teljesül, ha az alszelekt által visszaadott összes sorra teljesül a kifejezés operátor kifejezés_2 feltétel.

ANY: Ezzel a kulcsszóval feltétel akkor teljesül, ha az alszelekt által visszaadott sorok közül legalább egyre teljesül a kifejezés operátor kifejezés_2 feltétel.

IN: Ezzel a kulcsszóval felírt feltétel akkor teljesül, ha az IN előtt álló kifejezés, vagy zárójelbe tett kifejezések szerepelnek az alszelekt által visszaadott sorok közt.

EXISTS: Ezzel a kulcsszóval felírt feltétel akkor teljesül, ha az utána álló alszelekt legalább 1 sort visszaad.

A **NOT** kulcsszót az **IN** és az **EXISTS** előtt használhatjuk közvetlenül: NOT IN, NOT EXISTS. A többi esetben csak az egész kifejezést tudjuk tagadni a NOT szóval.

Korrelált vagy más néven **kapcsolt allekérdezés** esetén a külső selectnél kezdődik a kiértékelés, átadja a hivatkozott értéket a belső SELECT-nek mely előállítja az értékekhez tartozó eredményt. Ezután a belső SELECT által előállított értékkel folytatódik a külső SELECT kiértékelése. Az alszelekt tehát újra, és újra lefut, minden átadott értéknél. Ha a külső és a belső SELECT azonos táblára vonatkozik akkor az egyik táblát át kell nevezni, hogy a minősített nevek különbözzenek egymástól.

PÉLDA: *select * from cikk where exists (select gyartokod from gyartok where cikk.gyarto = gyartok.gyartokod);*

Tervezési minták egy OO programozási nyelvben

Az informatikában **tervezési mintának** (*Design Patterns*) nevezik a gyakran előforduló programozási feladatokra adható általános, újrafelhasználható megoldásokat. Egy programtervezési minta rendszerint egymással együttműködő objektumok és osztályok leírása.

A tervminták nem nyújtanak kész tervet, amit közvetlenül le lehet kódolni, habár vannak hozzájuk példakódok, amiket azonban meg kell tölteni az adott helyzetre alkalmas kóddal. Céljuk az, hogy leírást vagy sablont nyújtsanak. Segítik formalizálni a megoldást.

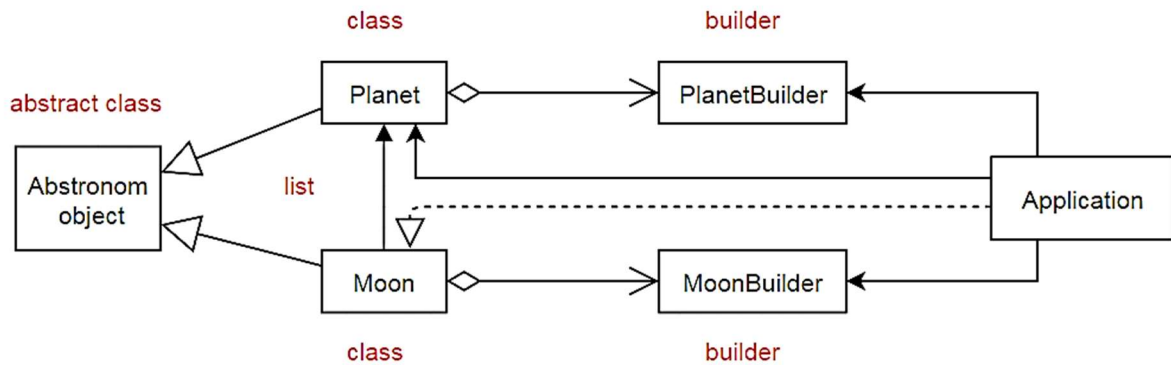
A minták rendszerint osztályok és objektumok közötti kapcsolatokat mutatnak, de nem specifikálják konkrétan a végleges osztályokat vagy objektumokat. A modellek absztrakt osztályai helyett egyes esetekben interfészek is használhatók, habár azokat maga a tervminta nem mutatja. Egyes nyelvek beépítetten tartalmazznak tervmintákat. A tervminták tekinthetők a strukturált programozás egyik szintjének a paradigma és az algoritmus között.

A programtervezési minták c. könyv számos tervezési mintát taglal, amiket a következő kategóriákba sorolunk:

- létrehozási minták
- szerkezeti minták
- viselkedési minták

Példa a létrehozási mintára:

Builder pattern: egy komplex objektum létrehozásában segít. Így átláthatóvá válik a konstruálás menete, ezáltal kevesebb a hibalehetőség.



```

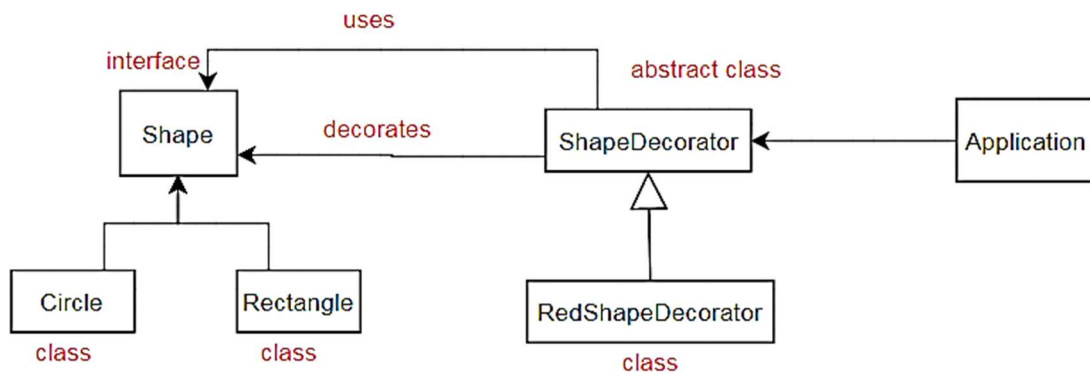
MoonBuilder moonBuilder = new MoonBuilder();
Moon moon = moonBuilder.setName("Moon")
moon.setWeight(new BigDecimal("3"))
moon.setDiameter(new BigDecimal("3474.2"))
moon.buildMoon();

```

A builder osztály minden metódusa önmagával tér vissza, így kapcsolhatóak össze a metódusok. A buildMoon() metódus konstruálja meg a Moon objektumot és azt adja vissza.

Példa szerkezeti mintára:

Decorator Pattern: Lehetővé teszi az absztrakció változtatása nélkül további funkciók, felelősségi körök dinamikus hozzáadását.



```

Shape circle = new Circle();
Shape redCircle = new RedShapeDecorator(circle);
Shape redRectangle = new RedShapeDecorator(new Rectangle());

```

```

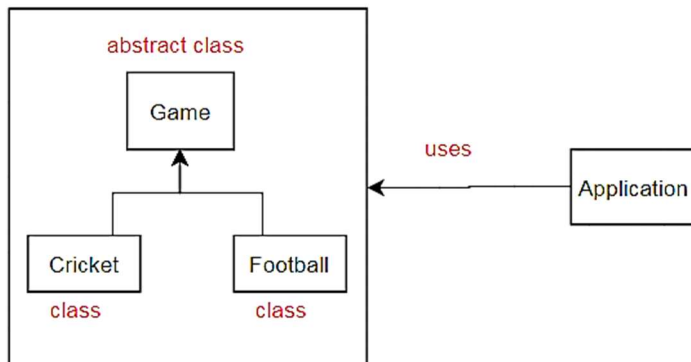
circle.draw();
redCircle.draw();
redRectangle.draw();

```

Példa viselkedési mintára:

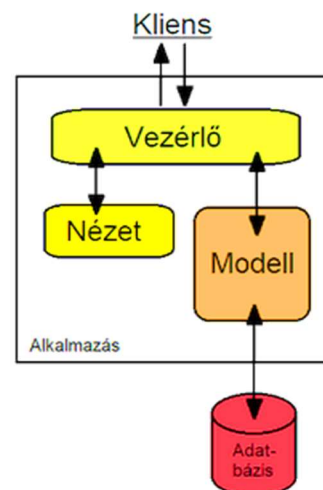
Template Pattern: egy absztrakt osztály egy metódusát fedi fel, amit nem lehet módosítani. Ez a template method felhasználja a többi nem implementált metódust.

```
public abstract class Game {  
    abstract void initialize();  
    abstract void startPlay();  
    abstract void endPlay();  
  
    public final void play() {  
        initialize();  
        startPlay();  
        endPlay();  
    }  
}
```



MVC

Az MVC egy összetett tervezési minta, ami a grafikus felülettel rendelkező alkalmazásoknál használható, főleg WEB-es környezetben alkalmazzák. Gyakori egy alkalmazás több rétegre való felbontása: **megjelenítés** (felhasználói felület), **tartománylogika** és **adatelérés**. Az MVC-ben a megjelenítés tovább bomlik nézetre és vezérlőre. Az MVC sokkal inkább meghatározza egy alkalmazás szerkezetét, mint az egy programtervezési mintára jellemző.



Modell – Az alkalmazás által kezelt információk tartomány-specifikus ábrázolása. A tartománylogika jelentést ad a puszta adatnak (pl. kiszámolja, hogy a mai nap a felhasználó születésnapja-e, vagy az összeget, adókat és szállítási költségeket egy vásárlói kosár elemeihez). Sok alkalmazás használ állandó tároló eljárásokat (mint mondjuk egy adatbázis) adatok tárolásához. Az MVC nem említi külön az adatelérési réteget, mert ezt beleérti a modellbe.

Nézet – Megjeleníti a modellt egy megfelelő alakban, mely alkalmas a felhasználói interakcióra, jellemzően egy felhasználói felületi elem képében. Különböző célokra különböző nézetek létezhetnek ugyanahhoz a modellhez.

Vezérlő – Az eseményeket, jellemzően felhasználói műveleteket dolgozza fel és válaszol rájuk, illetve a modellben történő változásokat is kiválthat.

Topológiák és architektúrák

A hálózati topológia a hálózat struktúráját adja meg. Lehet fizikai és logikai topológiáról beszélni. Fizikai topológián a vezetékek, az átviteli közeg tényleges elhelyezkedését értjük.

Leggyakoribb topológiák: Busz, Csillaghálózat, gyűrű, hierarchikus, háló

Két számítógép közötti közvetlen kapcsolatot, **pont-pont** (point to point) kapcsolatnak nevezzük.

A **gyűrű hálózati topológia** a klienseket egy gyűrűvé szervezi és általában a fizikai közeggel is gyűrűt hoz létre. A gyűrűben egy busz típusú adatátvitel létesül, amely egyirányú. Tehát ha a „B” állomás akar kapcsolatot létesíteni az „A” állomással, akkor az adat végig kell haladjon a gyűrűn. Ha egy periféria meghibásodik, megbéníthatja az egész hálózatot.

Csillaghálózat: A kliensek egy központi berendezéshez csatlakoznak. Ez általában a központi Switch. A Switch a küldő és a fogadó számítógép között felépít egy logikai pont-pont kapcsolatot. A csillag topológia nagy előnye, hogy nem érzékeny egyes kliensek meghibásodására vagy kiesésére.

A **logikai topológia** azt határozza meg, hogy hogyan kommunikálnak egymással az állomások. A két legelterjedtebb logikai topológia a szórás és a vezérjeles topológia.

A **szórásos** topológiaesetében az állomások minden adatot elküldenek minden, a hálózati közeghez csatlakozó állomásnak.

A **vezérjeles** topológiahasználatakor sorban minden állomás megkap egy elektronikus vezérjelet, ezzel megkapja a jogot, hogy adatokat küldjön a hálózatban.

OSI modell

1. **Fizikai réteg** - Átviteli közegek tulajdonságaival, a jelátvitel megvalósításával foglalkozik.
2. **Adatkapcsolati réteg** - Megbízható jelátvitel két közvetlenül összekötött eszköz között.
3. **Hálózati réteg** - Összeköttetés két hálózati csomópont között (IP)
4. **Szállítási réteg** - Megbízható összeköttetés két csomóponton lévő szoftver között. (TCP)
5. **Viszony réteg** - Végfelhasználók közötti logikai kapcsolat felépítése, bontása (Duplex)
6. **Megjelenítési réteg** - Az információ azonos módon értelm. a kapcsolat mindkét oldalán.
7. **Alkalmazási réteg** - Interfész az alkalmazások és a felhasználók között.

Fizikai átviteli jellemzők és módszerek, közeg hozzáférési módszerek.

Átviteli közeg: A hálózat állomásait kommunikációs csatornák kötik össze. Ezeket a csatornákat más néven átviteli közegeknek nevezzük. Az adatátvitelhez többféle fizikai közeg használható.

- **Vezetékes rendszerek:** Az ilyen rendszerekben valóban elektromos, vagy fény impulzusok továbbítására alkalmas kábelek kötik össze a számítógépeket.
- **Csavart érpár:** Egy kábel általában több érpárt tartalmaz. Ha az érpárokat árnyékoló fémburkolat takarja, Shielded Twisted Pair-ról (STP), azaz árnyékolott sodrott érpárról beszélünk, az árnyékolás hiánya esetén a kábelt Unshielded Twisted Pairnak (UTP), árnyékolatlan sodrott érpárnak nevezzük. Az UTP napjaink legelterjedtebb kábele.

Optikai kábel: Az optikai, vagy üvegszál kábelek nem elektromos, hanem fényimpulzusok segítségével továbbítják az üzenetek biteit.

Vezeték nélküli rendszerek: A vezetékes rendszerek kiépítése nem mindig megoldható. Ilyenkor vezeték nélküli technológiák közül lehet választanunk.

- infravörös kommunikáció kisebb távolságra
- rövidhullámú, rádiófrekvenciás átvitel (WiFi, Bluetooth) kisebb távolságra,
- mikrohullámú átvitel, mely működésének feltétele, hogy a két antennának látnia kell egymást
- lézer
- műholdas átvitel.

Az átviteli közeg hozzáférésére számos eljárást használnak. A hozzáférés módja — amint azt a későbbiekben látni fogjuk — függ az hálózat topológiájától is, vagyis attól, hogy milyen módon vannak az állomások összekapcsolva. A közeg elérési módja szerint három fő hozzáférési módszer lehetséges:

- **Véletlen vezérlés:** akkor a közeget elvileg bármelyik állomás használhatja, de a használat előtt meg kell győződnie arról, hogy a közeg más állomás által nem használt.
- **Osztott vezérlés:** ebben az esetben egy időpontban mindig csak egy állomásnak van joga adatátvitelre, és ez a jog halad állomásról-állomásra.
- **Központosított vezérlés:** ilyenkor van egy kitüntetett állomás, amely vezérli a hálózatot, engedélyezi az állomásokat. A többi állomásnak figyelnie kell, hogy mikor kapnak engedélyt a közeg használatára.