Tartalom

1.TÉTEL

Adattípusok

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

Char = 1, int = 4, short int = 2, long int = 8, float = 4, double = 8, long double = 16

A **signed** és **unsigned** minősítők csak az egész adattípusok esetén használhatók. Az **unsigned** minősítővel ellátott változók csak 0 vagy pozitív értéket vehetnek fel.

Összetett adattípusok közé tartoznak a **tömbök**, **struktúrák**, **pointerek** és az **unió**. 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. Az **unió** formailag megegyezik a struktúrával, de tagjai azonos memóriaterületen helyezkednek el. Mérete mindig a legnagyobb tag mérete lesz. Arra szolgál, hogy ugyanazt a memóriaterületet a program különböző időpontokban különböző célokra használhassa.

Deklaráció

A C nyelvben 3 féle deklaráció van:

- változódeklaráció
- típusdeklaráció
- függvénydeklaráció

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.

Példa:

int i = 0;

Típusdeklaráció nevet ad egy adattípusnak. A deklaráció után a típusnév úgy használható, mint a beépített típusok.

Példa:

typedef unsigned int UINT;

UINT a;

Ezután az ’a’ változó egy unsigned int adattípusú változóként használható.

Függvénydeklarálása

```
visszatérésiTípus név(paraméterek) {utasítások}
```

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 lévő 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** ágat 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 információ olyan új ismeret, amely megszerzője számára szükséges, és korábbi tudása alapján értelmezhető. Az információ olyan tény, amelynek megismerésekor olyan tudásra teszünk szert, ami addig nem volt a birtokunkban. (Úgy is fogalmazhatunk, hogy az információ valamely meglévő bizonytalanságot szüntet meg.) Azokat az információkat, amelyekből valamilyen konkrét tényt tudunk meg adatnak is nevezzük. Az információ értelmezett adat.

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

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 log2 n, ahol n az információk (hírek) darabszáma.

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)

A **postfix** formára hozott kifejezés tulajdonságai:

- *Nem* tartalmaz zárójeleket.
- Az operandusok sorrendje *nem változik* az eredeti (infix) kifejezésbeli sorrendhez képest.
- Az operátorok (műveleti jelek) vannak úgy áthelyezve, hogy az operátor közvetlenül a hozzá tartozó operandusok/operandus *után* jelenjen meg.
  Például egy két operandusú műveleti jel előtt előbb a bal, majd a jobb operandusa szerepel, ami lehet egy egyszerű operandus, vagy egy postfix kifejezés.
- Így a műveleti jelek kiértékelésének sorrendje *egyértelművé* válik.

A sorrend meghatározza a műveletek sorrendjét.

postfix példa: 2 2 1 + *

Keresési problémák állapottér-reprezentációja

Az **állapottér-reprezentáció** az egy keresési probléma modellezésére használható. Egy ilyen problémát 4 dolog definiál:

- A lehetséges állapotok halmaza, köztük a kezdeti állapot
- Akciók, amik az operátorok és az alkalmazhatóságig feltételük, amik megmondják minden állapotra, hogy adott állapotban milyen akciók lehetségesek
- Célállapot(ok)
- Útköltség, ami az egyes lépések költségeiből adódik össze

Egy probléma megoldása olyan akciók sorozata, amely a kezdőállapotból egy célállapotba visz.

Példa: Robotporszívó.

Állapottér:	Szobák, szobák tisztasága, porszívó helyzete.
Kezdőállapot:	Első szobában a porszívó, másik két szoba koszos.
Célállapot:	Minden szoba tiszta legyen.
Operátorok:	Porszívó mozog szobák között, takarítás
Költségek:	1 egység / operátor

Példa: 8-as csúszkajáték

Állapottér:	0..8-ig tartó számok sorozata
Kezdőállapot:	Véletlenszerű sorrend
Célállapot:	0,1,2,3,4,5,6,7,8 sorrend
Operátorok:	0 szám cserélhető a vele szomszédos számokkal
Költség:	1 egység/csere

Neminformált keresési eljárások

A **nem informált keresési eljárás** azt jelenti, hogy ezen stratégiáknak semmilyen információjuk nincs az állapotokról a probléma definíciójában megadott információn kívül. Működésük során mást nem tehetnek, mint a következő állapotok generálása és a célállapot megkülönböztetése a nem célállapottól.

Az algoritmusok hatékonyságát négyféle módon lehet értékelni:

- Teljesség: az algoritmus garantáltan megtalál-e egy megoldást, ha létezik megoldás?
- Optimalitás: a stratégia az optimális megoldást találja-e meg?
- Időigény: mennyi ideig tart egy megoldást megtalálni?
- Tárigény: a keresés elvégzéséhez mennyi memóriára van szükség?

A **szélességi keresés (breadth-first search)** egy egyszerű keresési stratégia, ahol először a gyökércsomópontot fejtjük ki, majd a következő lépésben az összes a gyökércsomópontból generált csomópontot, majd azok követőit stb., tehát az algoritmus sort használ (FIFO). Általánosságban a keresési stratégia minden adott mélységű csomópontot hamarabb fejt ki, mielőtt bármelyik, egy szinttel lejjebbi csomópontot kifejtené.

A szélességi keresés teljes, mert amennyiben egy célcsomópont véges mélységben fekszik, a kereső eljut hozzá.

A szélességi keresés optimális, ha miden cselekvésnek ugyanannyi a költsége.

Tárigénye nagymértékű $O(b^{d+1})$ *(d a mélység, b az elágazási tényező),* tehát el kell tárolni minden legenerált csomópontot.

Az időigénye megegyezik a tárigénnyel

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

A tárigénye kismértékű. 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 lekorlátozzuk 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. Hátultesztelő 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.

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 függvényhívásnál 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 függvényhívások között.

A **lokális és globális hatáskörök kettősét 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. A 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 számjegyünk 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 bájton 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 akadálya. 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 használ 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 költségének becslése. A célállapotban a heurisztika 0, azaz a célállapotból a célállapotba eljutni 0 költséget jelent. A heurisztika segít kereső algoritmusoknak több lehetőség közül a becsült jobbat kiválasztani.

A* algoritmus

A legjobbat-először keresés leginkább ismert változata az **A* keresés**. A csomópontokat úgy értékeli, hogy összekombinálja az aktuális csomópontig megtett út költségét és az adott csomóponttól a célhoz vezető út becsült költségét, tehát a teljes becsült útköltséget minimalizálja.

Az **A* kereső algoritmus** mindig a legolcsóbb utat keresi és tárolja. Ha olyan utat talál, ami olcsóbb, akkor módosítja az útvonalat. Egy keresés **teljessége** abból áll, hogy tetszőleges reprezentációs gráfban véges sok keresőlépés után képes-e előállítani egy megoldást, mely a kezdőállapottól 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

Az egy **függvény**, ami minden játékállásban meghatároz egy lépést, vagy lépések egy véges részhalmazát. **Nyerő *stratégia*** olyan stratégia, melynek az előírásai szerint alkalmazva az operátorotokat az adott játékos mindenképpen nyer (az ellenfél lépéseitől függetlenül). **Nem-vesztő stratégia:** amely az ellenfél akármilyen játékvezetésénél sem vezet veréséghez. Nagyjából azt lehet mondani, hogy egy optimális stratégia olyan kimenetelekhez vezet, amelyek legalább olyan jók, mintha bármilyen más stratégiával egy tévedhetetlen opponens ellen játszanánk.

Minimax-algoritmus

A **minimax algoritmus** az optimális döntést az aktuális állapotból számítja ki, felhasználva az egyes követő állapotok minimax értékeinek kiszámítását. Tehát minimalizálja a maximális veszteséget, azaz azt az ágat választja egy döntési fában, ahol, ha az ellenfél tökéletes választásokat tenne, akkor a legkisebb lenne a vesztesége. A számítás lentről felfelé történik, a levélcsomópontoktól felfelé haladva. 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 megfelezhetjük, 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 MAXszámára eddig megtalált legjobb (azaz a legmagasabb értékű) választás értéke.

β = az út mentén tetszőleges döntési pontban a MINszámára 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 tömb elemei egyforma típusúak kell legyenek, de ez a típus bármi lehet. C-ben az elemek számozása 0-tól történik, és ez a legtöbb másik programozási nyelvben is így van.

A tömb *indexelése* (indexing), más néven címzése 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*-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 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, ilyet 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 előállítása a *címképző* & (address of) operátorral történik.
A pointer által hivatkozott változót az ***p** módon érjük el, a memóriacímet %p p-vel lehet kiiratni.

```
doubleszamok1[5] = { 4.5, 9.2, 7.1, -6.9, 8};
doubleszamok2[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: C-ben a karakterláncok karakter típusú tömbként vannak értelmezve. A karakterlánc végét ’\0’ karakter jelzi.

h	e	l	l	o	\0

Sztring létrehozása és inicializálása

```
charszoveg1[50] = { 'h', 'e', 'l', 'l', 'o', '\0'};
charszoveg2[50] = "hello";
charszoveg3[50];
```

Egy adott méretű tömbbe méret−1 hosszú, azaz egy karakterrel rövidebb szöveg fér csak! A lezáró 0-nak is kell hely! Pédául: 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.

C nyelvben a karakterláncon értelmezhetők különböző parancsok.

Pl. strcpy(s1,s2) – másolja az s2 string tartalmát az s1-be

strcat(s1,s2) – s2 tartalmát az s1 végéhez toldja

strlen(s1) az s1 hosszát adja meg

Dinamikus memóriakezelés

Segítségével mi dönthetjük el, mennyi memóriát foglalunk le, mikor foglaljuk le a memóriát és mikor szabadítjuk fel. Ez hasznos is, de egyben felelősség is a foglalás és a felszabadítás is, amiről ha elfeledkezünk gondok lehetnek.

malloc() függvény

void *malloc(size_t meret)

Lefoglal egy bájtban megadott méretű memóriaterületet (malloc: memory allocation)
visszaad egy pointert ami a lefoglalt területre mutat, vagy NULL pointert ad, ha nem sikerült lefoglalni a területet, a terület pedig inicializálatlan, tehát memóriaszemetet tartalmaz.

free() függvény

Felszabadít egy memóriaterületet, amit a malloc() foglalt
Ugyanazt a pointert kell megadni, amit a malloc() adott a foglalásnál.
Példa:

```
double *tomb;
int n = 10;
tomb = (double*) malloc(n*sizeof(double));
…
free(tomb);
```

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 → a , A → BC

(A,B és C nem terminálisok, a terminális, és A jobb oldalán a áll, A jobb oldalán pedig BC áll.

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**.

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 (Médium 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,

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

Elemi adatkapcsolati protokollok:

Szimplex: Az adatátvitel mindig csak egy irányban, az adótól a vevőhöz folyhat, csak egy irányban továbbítódhatnak 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, forgalomirányítók, 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 (pontozott 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 specifikáció

A funkcionális specifikáció módszerében először megadjuk az adattípus matematikai reprezentációját, amelyre azután az egyes műveletek elő-, utófeltételes specifikálása épül.

Például a sor adatszerkezetnél elmondható, hogy absztrakt szinten úgy tekinthető a sor, mint (elem,időpont) rendezett párok halmaza. Az időpontok jelzik, hogy az egyes elemek mikor kerültek be a sorba. Azt kikötjük, hogy az időpontok nem lehetnek azonosak.

Egy elem kivételénél az az elem kerül kivételre, melynek az időpontja a legkisebb, azaz legelőször került be a sorba.

PR. Tétel: A lineáris keresés tétele

Lineáris vagy szekvenciális keresés néven is ismert, mivel végig megyünk a számokon sorba.

Adott elem szerepel-e a tömbben és hányadik helyen. Az algoritmus:

```
keresett_elem = 30  //keresett elem
i = 0               //ciklusváltozó
ciklus amíg i<n és t[i]!=ker        //tömb bejárása, amíg elem nincs megtalálva
   i = i + 1                        //ciklusváltozó növelése
ciklus vége

Ha i<n akkor                        //ha a ciklusváltozó kisebb mint a tömb hossza
   ki "Van ilyen"                   //van ilyen elem
   ki: "Indexe: ", i                //kiírni az elem helyét
különben
   ki: "A keresett érték nem található"     //ha nincs, kiírjuk h nincs
ha vége
```

PR. Tétel: A logaritmikus keresés tétele

Bináris keresés pl. Rendezett tömbben alkalmazható csak. Megnézzük a középső elemet. Ha az a keresett szám, akkor vége. Ha nem akkor megnézzük, hogy a keresett elem a tömb alsó vagy felső részében van. Amelyik tömbrészben benne van a keresett szám, a fentieknek megfelelően keresem a számot. A ciklus lépésszáma nagyjából log2(n), ahol n a tömb elemeinek darabszáma.

```
                                    //első = 0, utolsó = tömb hossz-1
ciklus amíg  első <= utolsó  és  van = hamis    //amíg nagyobb a felső határ, és nincs meg
   Középső :=  (Első + Utolsó)  Div 2           //középső elem első+utolsó / 2
```

```
   Ha keresett = t[középső] akkor          //ha megvan
     van := igaz                           //flag igaz
     index := középső                      //beállitom az indekszét
   else
      ha Keresett < t[középső] akkor       //ha nincs meg, és a keresett kisebb mint középső
         utolsó := Középső – 1             //utolsó legyen a középső -1
      Ellenben                             //ha nincs meg és keresett nagyobb mint középső
         Első := Középső + 1               //első legyen a középső +1
      Ha vége
ciklus vége
```

PR. Tétel: Az eldöntés tétele

Szeretnénk tudni, hogy egy érték megtalálható-e egy tömbben.

```
i = 0
keresett = 20
ciklus amíg i<n és t[i] != keresett
   i=i+1
ciklus vége

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

PR. Tétel: A kiválogatás tétele

A tömb elemit egy másik tömbbe rakom, feltételhez kötve. Például: Adott a és b tömb. Az a tömb egész számokat tartalmaz. Az a tömbből az 5-nél kisebb számokat átrakom b tömbbe.

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

Buborékos rendezés

Feltételezzük, hogy az n a tömb elemeinek száma.

A sorozat két első elemét összehasonlítjuk, és ha fordított sorrendben vannak felcseréljük. Utána a másodikat és a harmadikat hasonlítom össze.

Ha a nagyobb számokat a végén szeretném látni, akkor az első körben a legnagyobb szám tulajdonképpen a sor végére kerül. Ezért a következő körben azzal már nem foglalkozunk, ezért megyünk mindig a belső ciklusban csak i változó értékéig. A külső ciklus így megmutatja meddig kell mennünk.

```
{
    int n = arr.length;
    for (int i = 0; i < n-1; i++)          //külső ciklus, 0-tól n-1ig
        for (int j = 0; j < n-i-1; j++)    //belső ciklus, 0tól, n-i-1ig
            if (arr[j] > arr[j+1])         //ha az adott elem nagyobb, mint az utána
            {
                // swap arr[j+1] and arr[j] //megcseréljük a két elemet
                int temp = arr[j];          //tempbe berak
                arr[j] = arr[j+1];          //átrak
                arr[j+1] = temp;            //tempből kivesz
            }
}
```

Relációs adatmodell

A reláció gyakorlatilag egy táblázat, annak összes tartalmával együtt. A táblázatban egy sort rekordnak neveznek, míg a táblázat egy oszlopa az attribútum(tulajdonság), egy sor és egy oszlop metszete a mező, amiben az adat található.

Egyed más néven entitás, az, aminak az adatait gyűjteni és tárolni akarjuk az adatbázisban.

Attribútum, tulajdonság, az egyed valamilyen jellemzője, az egyed az attribútumainak összességével jellemezhető.

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 egyed halmazbeli elem tartozik. Egyszerűbben: kölcsönösen egyértelmű a két egyedhalmazközötti 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.)

Elsődleges Kulcs, olyan attribútum, vagy attribútumok halmaza, amellyel egy egyed egyértelműen beazonosítható. (Egyszerű v összetett kulcs), értéke egyedi.

Idegen kulcs olyan azonosító, melynek segítségével egy másik tábla elsődleges kulcsára hivatkozhatunk.

Hivatkozási integritás, megóv minket attól, hogy az elsődleges táblában végzett módosítás veszélyeztesse a táblák kapcsolatát.

A **hivatkozási integritás** megőrzése a következő korlátozásokat jelenti:

1. Új rekord hozzáadása a kapcsolt táblához csak akkor lehetséges, ha létezik hozzá a csatolás alapján megegyező rekord az elsődleges táblában.

2. Az elsődleges tábla elsődleges kulcsát nem módosíthatjuk, ha létezik hozzá kapcsolt rekord a kapcsolt táblában.

3. Az elsődleges táblából nem törölhetünk olyan rekordot, ha létezik hozzá kapcsolt rekord a kapcsolt táblában.

Lehetséges azonban, hogy mégis szükségünk van az elsődleges táblában az elsődleges kulcs módosítására, vagy rekord törlésére. Ilyenkor az Access lehetőséget nyújt a kapcsolt mezők kaszkádolt frissítésére, törlésére, ami azt jelenti, hogy az elsődleges táblában végzett módosításokat a kapcsolt tábla megfelelő rekordjaiban is elvégzi.

Kényszer (constraint): A lehetséges adatok halmazát leíró, korlátozó szabály. Ilyen kényszerek pl a NOT NULL, UNIQUE, PRIMARY KEY (UNIQUE ÉS NOT NULL kombinációja) FOREIGN KEY, CHECK, DEFAULT, CREATE INDEX

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 nevezzük, 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 függnek.

•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 FIFO (First In First Out)** működési elvű, a programozásban használatos adatszerkezet. A besorolt elemek mindig a sor végére kerülnek. Elemet kivenni kizárólag a sor elejéről lehet. Jellemző felhasználási módja a sorszámhúzó automata, vagy a gyorsabb és lassabb rendszerek közötti puffer, például a billentyűzetpuffer vagy a nyomtatási sor.

Jellemző sorműveletek: elem hozzadása, elem eltávolítása, elem megtekintése eltávolítás nélkül.

A **verem** (angolul stack) egy **LIFO** adatszerkezet, amelyben általában véges számú azonos típusú (méretű) adatot lehet tárolni.

Jellemző veremműveletek:

- 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 méretéve

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 dinamiksuan kezelhető adatstruktúrát.
- **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
    - **Preorder** bejárás: tartalom, bal, jobb
    - **Inorder** bejárás: bal, tartalom, jobb
    - **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 részfában 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, **beszúráskor** ez alapján több stratégiánk is lehet
    - minél kisebb erőforrásigényű legyen a beszúrás ,
    - 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
    - Két gyerekkel rendelkező elem mindkét gyerekét nem tudjuk az ő szülőjének egy mutatójára rákapcsolni.
    - 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** többféleképen is lehetséges: **prioritásos** (a nagyobb prioritású folyamat mindaddig fut, amíg be nem fejeződik), **időosztásos** (mindegyik futásra váró folyamat, ugyan azonos időszeletet 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 elrejti a tárolásának, kezelésének fizikai részleteit:

- Melyik fizikai eszközön található (logikai eszköznévvel 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, pufferelést.

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 λ∉L(G), akkor a G '- beli szabályok jobboldalán λ nem fordul elő. Ha viszont λ∈L(G), akkor az egyetlen G '- beli szabály, aminek jobboldala az üresszó S ' →λ, ahol S ' a G ' mondatszimbólumát jelöli. Ezesetben, azaz λ∈L (G ') fennállása esetén viszont S ' (azaz a G ' mondatszimbóluma) 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 az **egységbebezárás** (encapsulation), valamint az **öröklődés** (inheritance), **polimorfizmus**.

Egységbezárás (encapsulation): Az adatok és hozzájuk tartozó eljárások egyetlen egységben való kezelését jelenti, objektumban, vagy osztályban. Az osztály mezői tárolnak információkat, és az osztályt publikus metódusain keresztül lehet elérni más osztályokból. Az osztály mezőit is csak a metódusokon keresztül lehet megváltoztatni.

Öröklődés (inheritance): Öröklődés azt jelenti, hogy egy objektum egy másik objektum tulajdonságait, metódusait megkapja. Ez az újrahasznosíthatóságot segíti. Az öröklött tulajdonságokat vagy metódusokat kiegészítheti újakkal, vagy meg is változtathatja azokat valamilyen módon. Egy osztálynak csak egy ősosztálya lehet Java-ban, de akárhány osztály öröklődhet egy adott osztályból. Az összes osztály az Object osztályból öröklődik.

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 következőek:

Egészek

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

Valós számok

Típus	Leírás	Méret/formátum
float	gyszeres 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 a stringek referencia-típusúak. A referencia-változó más nyelvek mutató vagy memóriacím fogalmára hasonlít.

Sokszor van szükség a különböző adattípusok közti átváltásokra, ezt **típuskonverziónak** nevezzük. Megkülönböztetünk **implicit** és **explicit** típuskonverziót. Ha egy típust egy nagyobb méretű típusra akarunk átváltani, automatikusan megtehető ez, külön kényszerítés nélkül. Ha viszont egy típust egy kisebb méretű típusra szeretnénk átváltani, akkor ezt kóddal kell kényszeríteni.

- 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. Javaban a következő operátorokkal dolgozhatunk:

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

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

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

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

Logikai operátorok, két érték közötti logikát vizsgálják pl:

&& logikai és

|| logikai vagy

! logikai not

Utasítások

Az **utasítás** a programkód egy lépését adják meg, vannak:

- kifejezés utasítások
- ciklus utasítások
- feltételes utasítások
- összetett utasítások (blokk)
- változó deklaráció
- növelés és csökkentés utasítások

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 tartalmaznak 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 a class kulcsszóból és az osztály nevéből áll. 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 **csomagszintűelérhetőség**.

- 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 visszatérési 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: ∅. Nem tévesztendő össze a {λ} nyelvvel, amely egyedül az üresszót tartalmazza.

Generatív nyelvtan, azoknak a szabályoknak a halmaza, amelyekkel minden, a nyelvben lehetséges jelsorozat előállítható, azaz leírja, hogyan lehet előállítani egy átírási eljárással a kitüntetett kezdő szimbólumból a többi jelsorozatot a szabályokat egymás után alkalmazásával.

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 mondatszerkezetű)**, ha semmilyen megkötést nem teszünk a helyettesítési szabályaira.
- **1-es típusú (környezetfüggő)**, a helyettesítési szabályok jobboldala nem lehet rövidebb a baloldalnál. alfa – béta, ahol |alfa| <= |béta|
- **2-es típusú (környezetfüggetlen)**, a szabály bal oldala egyetlen nem terminális A szinbólum, és ez mindig helyettesíthető egy alfa jelsorozattal, függetlenül attól mi a nemterminális környezete, A - alfa
- **3-as típusú (reguláris),** Kétféle szabálytípus engedélyezett, A – a és A – aB, tehát a helyettesítési szabály bal oldala mindig egyetlen nem terminális, jobb oldala pedig egyetlen terminális, vagy egyetlen terminális és egyetlen nem terminális

Logikai áramkörök

A számítógépek digitális **áramkörökből** épülnek fel. A digitális áramkör két logikai értékkel dolgozik, a 0-val és az 1-el. 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.

A **hét logikai kapu** az **AND(és), OR(vagy), XOR(kizáró vagy), NOT(negálás), NAND(negált és), NOR(negált vagy), and XNOR(negált kizáró vagy).**

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 2n adatbemenete. A multiplexer feladata az, hogy a több bemenetére érkező jelből egyet vezessen a kimenetére. A multiplexer címzése 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 2n 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 2n-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 „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 elől. 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. Javaban csak egyszeres öröklődés van, azaz egy osztály legfeljebb egy ősosztályt 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 léteznek 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 egyszerűsíthetjük 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 (indexeltek). 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 szelektivitásra, és amelyekre gyakran hajtunk végre lekérdezést.

Az adatbázis-tervezés elmélete

Funkcionális függésről akkor beszélünk, ha egy táblában egy tulajdonság értéke egy másik tulajdonság értéke egyértelműen meghatározza.

Teljes funkcionális függésről akkor beszélünk, ha a meghatározó oldalon nincsenek felesleges adatok, egyébként **részleges funkcionális függésről** beszélünk.

Tranzitív függés esetén pl 3 attribútum van, A-B-C

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

Egy reláció **első normál formában** van, ha minden attribútuma egyszerű, nem összetett adat.

Második normál forma, ha

- A reláció első normál formában van
- A reláció minden nem elsődleges attribútuma teljes funkcionális függőségben van az összes reláció kulccsal

Harmadik normál forma, ha

- A reláció második normál formában van.
- A reláció nem tartalmaz funkcionális függőséget a nem elsődleges attribútumok között.

BCNF (Boyce-Codd normál forma), ha

- A reláció harmadik normál formában van
- Minden elsődleges attribútum teljes funkcionális függőségben van azokkal a kulcsokkal, melyeknek nem része

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

Beszúrási anomália, amikor egy adatrekord beszúrása egy másik, hozzá logikailag nem kapcsolódó adatcsoport beszúrását 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 adatkiírást hajtunk végre. Kétféle fajtája lehet: az egyikben a programszöveg bizonyos helyeire helyezünk el tesztkiírásokat. Ha a futás során arra a pontra érünk, akkor a benne szereplő változókat a programkiírja, 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 programfutásáról: 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 programolyan utasításai, amelyeknél a végrehajtásnak meg kell állnia, a felhasználó információt szerezhet a programállapotáról, 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 változóértékét módosítsuk, és újabb elindítás nélkül lokalizálhassunk esetleges más hibákat is.

Lépésenkénti végrehajtás

Ez tulajdonképpen olyan eszköz, amely a programminden utasítására egy töréspontot definiál. A programminden 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) teszteleni.

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.

Kollekciókhasználata

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

Javaban:

- 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 ú.n. 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 végrehajtatni 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 különbözőségre 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

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

Így lehet osztályt inicializálni, ebben az osztályban az a ’a’ és ’b’ mező értéke 0-ra állítódik alapértelmezetten. Megadható más érték is, akkor azt az értéket veszi fel.

```
Number num = new Number();
```

Az osztály példányosítása az osztály egy konstruktorának meghívásával történik. Ebben a példában a default konstruktor hívodik meg, annak megfelelő értékeket vesz fel, attól függ melyik konstruktor hívódik meg, hogy hány darab és milyen típusú paramétert kap a konstruktor, ha van olyan konstruktor.

Konstruktor

Konstruktor az egy sepciális feladatú metódus, feladatuk a létrehozott objektum-példány alaphelyzetének beállítása. A konstruktorok nevének meg kell egyeznie az osztály nevével, lehet formális paraméterlistája, és túlterhelhető a konstruktor, azaz más típusú vagy más számú paraméterekkel lehet több konstruktort is létrehozni, és példányosítás során az fog meghívódni, amelyikkel megegyeznek az aktuális paraméterek. Visszatérési típusa nincs, meghívása a new kulcsszóval történik. Példa:

```
class Teglalap {
        private float x,y;

        public Teglalap(){
                x = 10;
                y = 20;
        }
        public Teglalap(int x, int y){
                this.x = x;
                this.y = y;
        }
}
```

Interfészek

Az interfész olyan viselkedéseket definiál, amelyet egy tetszőleges osztályával 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 utánszerepelhetnek a szülőinterfészek. Az interfész törzs metódus deklarációkat tartalmaz ;’-el lezárva.Minden deklarált metódus alapértelmezetten publikus és absztrakt.

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> 11 = new ArrayList<String>(); List<Object> 12 = 11; //hiba

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

Összetett adatszerkezeteket implementáló osztályok és fontosabb műveleteik

- 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 szúrni elemeket a listába elemek közé is.
- SET - Egyedi elemeket tárol sorrendiség nélkül.
- MAP - Kulcs-adat objektum-párok csoportja, a kulcsra gyors keresést biztosít.

ArrayList/List/Set fontosabb műveletei:

- add/addAll() (elem vagy elemek hozzáadása)
- Clear() (minden elem törlése)
- contains/ContainsAll() (igaz, ha tartalmaz elemet, elemeket)
- isEmpty() (igaz, ha üres)
- remove/removeAll (elem, vagy elemek törlése)
- size() (elemek száma)
- iterator (iterátort készit)

Map fontosabb műveletei:

- put/putAll (beszúr elemet elemeket)
- remove (kulcs alapján töröl)
- containsKey/containsValue() (igaz, ha tartalmazza kulcsot / értéket)
- get (visszaadja azt a kulcsú elemet)
- isEmpty (igaz, ha üres)
- clear (töröl minden elemet)

Rendszerfejlesztési modellek

A rendszerfejlesztési modellek három szempontból közelítik a rendszerspecifikációkat:

- **Környezeti modellek**: kívülről közelítenek, azt a környezetet modellezi, amelybe bel kell helyezni a rendszert, azaz hol működik a rendszer
- **Viselkedési modellek**: belülről közelítenek, a rendszer viselkedését helyezik a központba, azaz hogyan működik a rendszer
- **Adatmodellek**: belülről közelítenek, középpontban a rendszer és az általuk feldolgozott adatok struktúrája áll, azaz mivel dolgozik a rendszer

A **környezeti modell** nem mond magáról a rendszerről belülről semmit, de a rendszer és környezet között fennálló viszonyról sem, azon felül, hogy van köztük kapcsolat. Csak a statikus környezet leírására szolgál.

A **viselkedési modellek** állapotátmenet modellek, tipikusan objektum orientált.

Az **adatfolyam modell** az adatok mozgását, áramlását vizsgálja, ezek intuitívak, a felhasználó könnyen megérti. Jelölések: ellipszis -> feldolgozás, függvénynek tekinthető, nyíl -> az adatáramlás irányát jelöli, adat milyensége a nyílra van írva, téglalap -> ezek állományok, adatbázisok, adat megőrzése.

Tervezés

Követelménytervezés során előáll a dokumentum, ami majd a fejlesztés alapjául szolgál. Alfolyamatai:

- megvalósíthatósági tanulány
- követelmények feltárása, elemzése
- követelmények specifikálása, dokumentálása
- követelmények validálása

Megvalósíthatósági tanulmány

A rendszer nagyvonalú leírását tartalmazza, eldöntjük, hogy elindítható egyáltalán a folyamat. Megvalósítható-e az adott időkeret, költségkeret, technológiai keret között, integrálható-e a jelenlegi rendszerbe.

Követelmények feltárása

A leendő felhasználóktól beszerzett információk alapján történik, itt történik meg a követelmények leírása, sorrendbe állítása, ellentmondások feloldása, dokumentálás.

Követelmények validálása

Felülvizsgálata a dokumentumoknak, prototípuskészítés, tesztelhetőség vizsgálata, tesztesetek készítése.

Tesztelés

Tesztelés kideríti, vannak-e hiányosságok a szoftverben a belső és külső követelményekhez képest, a teljesítményproblémákat. A tesztelést követi a hibák behatárolása, a debuggolás, és a regressziós tesztelés, azaz, hogy a hiba javítása nem-e befolyásolt más, eddig működő rendszereket negatívan.

Átvizsgálás során az összes résztermékét lehet vizsgálni, a dokumentációtól kezdve a kódon át a kész szoftverig. Unit teszt – általában a fejlesztők készítik a saját kódjukra, ez white-box tesztelés, lehet a kód megírása után írni, vagy még előtte. Alrendszer integrációs teszt, itt osztályok vagy metódusok együttműdökését vizsgálják, megfelelően működnek-e együtt. Integrációs tesztben azt vizsgálják a rendszer jól működik-e más rendszerekkel, esetleg külső rendszerekkel.

2 féle tesztelés van, funkcionális és nem-funkcionális

Funkcionális teszteléskor azt teszteljük, hogy a program megfelelően működik-e, a funckionális követelményeket teljesítni-e. Nem funkcionális tesztelésbe tartozik a teljesítménytesztelés, pl adott válaszidő alatt teljesül-e valami, vagy UI tesztelés, azaz használható-e a program, egyértelmű-e stb.

Funkcionálison belül van white-box és black-box tesztelés. Black-box tesztelésnél a tesztelő nem ismeri a kódot, csak inputok és outputok alapján tesztel, megadott inputok esetén azt vizsgálja, az elvárt output érkezik-e vagy sem. White-box tesztelésnél a rendszer is ismert, lehet követni a kódot stb.

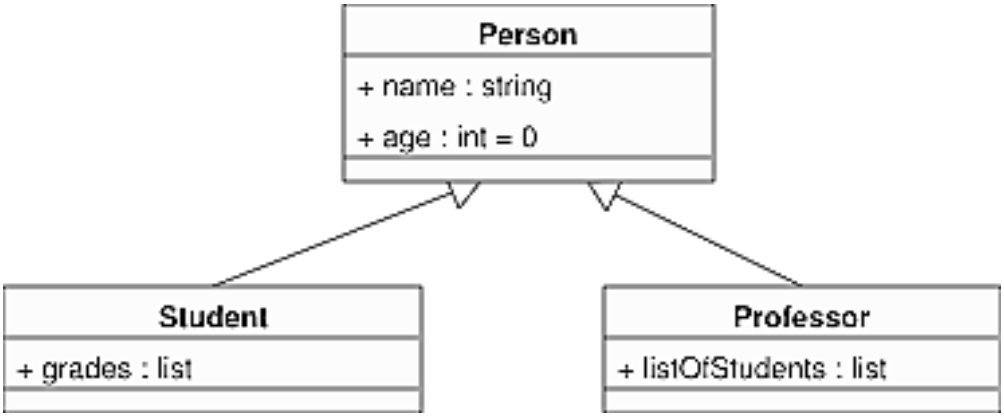
UML osztálydiagram (Unified Modeling Language)

- 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 alatt több verzióval rendelkező adatok kezelését értjük. Általában szoftverek megépítéséhez szükséges forrásfileok, követelmények, tesztesetek tárolására és megosztására használatos. Biztonságos formában, ellenőrzött hozzáféréssel. Fájlok újabb és korábbi változatainak megőrzésével.

Biztonság abban áll, hogy védve kell legyen hardverhibák ellen, rendszeres mentések történjenek.

Ellenőrzött hozzáférés: tartozzon hozzá felhasználó-azonosítás, változások legyenek névhez kötve, csak a jogosult felhasználók férjenek hozzá adott állományhoz.

Alapműveletek a checkout, ahol egy másolatot készítünk a közös fileból, update, ahol a másolatunkat frissitjuk ha azóta történt változás, változások megtekintése, commut, push, visszírás a repo-ba.

Számítógép-architektúrák – mikroelektronika, processzor és memória

A **félvezetők** gyengén vezetnek, fajlagos vezetőképességük a szigetelőkénél 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-processzor (**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 hajtódik 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 **INSERTINTO** 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.

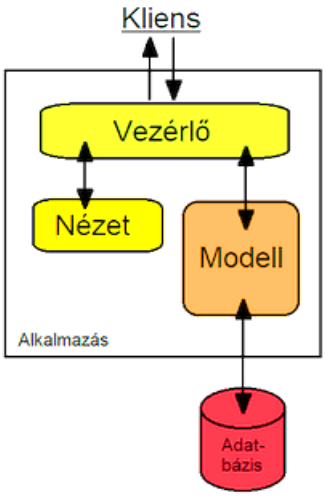
A **NOT** kulcsszót az **IN** és az **EXISTS** előtt használhatjuk közvetlenül: NOT IN, NOT EXISTS. A többi esetben csak az egész kifejezést tudjuk tagadni a NOT szóval.

Korrelált vagy más néven **kapcsoltallekérdezés** 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

A tervezési minták sokszor felmerülő problémák leírásait adják meg úgy, hogy közben megadják a megoldás magját, amit adaptálva a konkrét rendszerhez újra és újra alkalmazhatunk. Objektum orientált programok esetében a tervezési minta leírása megadja azokat az egymással kommunikáló objektumokat, osztályokat, amelyek együttes viselkedése az adott problémára megoldás lehet. A minták persze nem fednek le mindent, és mindenre nem adnak megoldást, csak a gyakorta előforduló problémákra adnak megoldás javaslatot.



Tervezési minta elemei:

- a minta neve
- a probléma leírása
- megoldás, az adott problémára

MVC

A modell

Itt tároljuk az adatokat és az üzleti logikát. Egy tipikus webapplikációban ez a rész foglalkozik például az adatbázisokkal és egyéb objektumokkal amivel az applikáció dolgozik.

A nézet

A nézet a tényleges vizuális leképzése a modellednek. Egy tipikus webalkalmazásban ez lenne például a weblap ami megjeleníti a modellt a felhasználó részére, legyen az egy adatbevitelre használt űrlap, adatkivitelre használt mezők, vagy a kettő keveréke.

Természetesen a nézetnek nem kell egy modell minden részét mutatnia, és egy modellhez tartozhat több nézet is.

A vezérlő

A vezérlő kezeli a modell és a nézet közti kommunikációt. Egy webalkalmazásban a vezérlő metódusai vannak végrehajtva amikor a felhasználó egy weblapot tölt be vagy egy gombra kattint. A vezérlő ezekkel frissíti a modellt, ha szükséges, majd új nézetet ad vissza, ha pedig az szükséges.

További tervezési minták pl: Gyártási minták, pl gyártófüggvény, ami interfészt definiál egy objektum létrehozásához, de az alosztályokra bízza, melyik osztályt példányosítsák.
Szerkezeti minták, pl az Adapter, az adott osztály interfészét alakítja át, hogy kompatibilissé váljon más rendszerekével.
Viselkedési minták: Iterator pl, ami egy adott tároló-objektum elemeinek sorozatos elérését teszi lehetővé.

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 **gyűrűhálózatitopológia** 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 vezér jelet, 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 árnyékolt 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 üvegszálas kábelek nem elektromos, hanem fényimpulzusok segítségével továbbítják az üzenetek bitjeit.

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 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.