1. TÉTEL

1. **A C programozási nyelv I.**: Adattípusok, deklarációik, feltételes utasítások.
   Általános ismeretek: adat és információ, entrópia fajtái, kifejezések infix és postfix alakja.
   Mesterséges intelligencia: Keresési problémák állapottér-reprezentációja, példák.
   Neminformált keresési eljárások (mélységi, szélességi, optimális).

Adattípusok

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

A **signed** és **unsigned** minősítők csak az egész adattípusok esetén használhatók.

Az **unsigned** minősítővel ellátott változók csak 0 vagy pozitív értéket vehetnek fel.

Az **signed** minősítővel ellátott változók esetén eggyel több negatív érték ábrázolható, mint pozitív.

Egész adattípusok

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

Lebegőpontos adattípusok

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

Összetett adattípusok

Összetett adattípusok közé tartoznak a **tömbök** és **struktúrák**. A tömbök használatával **azonos elemi típusú adatokból** tárolhatunk többet egy változón belül. A struktúrák használatával **többféle adattípusból álló egyedi típust** tudunk létrehozni.

Minden adattípushoz tartozik egy **pointer** típus (char*, int*) ami a **memória címét tartalmazza a változóról**.

Deklaráció

A változókat használat előtt deklarálni kell. A változók 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 memóriaterületet.

A C nyelvben 3 féle deklaráció van	
változódeklaráció	int i = 0;
típusdeklaráció	typedef unsigned int UINT; UINT a;
függvénydeklaráció	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 az 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.

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

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

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 maximális, ha valamennyi jel azonos valószínűséggel fordul elő.

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

Kifejezések infix és postfix alakja

A **matematikában** a műveletek leírására általában **infix** jelölést használunk. Az operátorokat az operandusok közé írjuk és a sorrendiség a megszokott módon történik. Zárójelezéssel módosítható az operátorok sorrendje.

infix példa: 2 * (2 + 1)

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

postfix példa: 2 2 1 + *

Keresési problémák állapottér-reprezentációja

Az **állapottér-reprezentáció** egy probléma megoldásához szükséges tér, tulajdonságok és jellemzők modellezése.

Az **állapotok halmaza** tartalmazza az összes lehetséges állapotot, amelyek előállhatnak a jellemzők kombinációjából. A lehetséges állapotok közül meg kell adnunk egy **speciális** állapotot, mely a jellemzők kezdőértékeit határozza meg. Ezt az állapotot kezdőállapotnak nevezzük.

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

Az állapotok változtatásához meg kell adni a lehetséges műveleteket, cselekményeket, ezeket **operátoroknak** nevezzük. Az operátokhoz tartoznak **költségek** és **megszorítások**.

Az **operátor költsége** megadja, hogy milyen költsége van az operátor elvégzésének.

Az **operátor alkalmazási előfeltételei** pedig megszabják, hogy milyen állapotban használható az adott operátor.

Robotporszívó példa:

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

Neminformált keresési eljárások

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

A kereső algoritmusokat a következő tulajdonságokkal jellemezzük:

Teljesség: A rendszer minden olyan esetben megtalálja-e a megoldást, ha az létezik?

Optimalitás: Az optimális megoldást találja-e meg?

Időigény: Mennyi ideig tart egy megoldást megtalálni?

Tárigény: Mekkora memóriára van szükség a megoldás megtalálásához?

- 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, tehát az algoritmus sort használ. (FIFO)

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**, mert el kell tárolni minden legenerált csomópontot.

Az időigénye **megegyezik** a tárigénnyel.

- A **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**, mivel 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.

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, melyeknél az operátor alkalmazásokhoz költség van rendelve. Egy megoldás költsége alatt a megoldást alkotó operátoralkalmazások költségeinek összegét értjük.

Az **optimális kereső** a nyílt csúcsok közül mindig a legkisebb költségűt terjeszti ki.

2. TÉTEL

2. **A C programozási nyelv II.**: Ciklusszervezési lehetőségek, függvénykezelés, paraméterkiértékelés, hatáskörkezelés (statikus, dinamikus).
Általános ismeretek: számrendszerek, számábrázolás (fix és lebegőpontos), karakter, szöveg és logikai adat ábrázolása.
Mesterséges intelligencia: A heurisztika fogalma, példák. A* algoritmus. Az A* algoritmus teljessége. Kétszemélyes, teljes információjú, determinisztikus játékok: a stratégia fogalma, minimax-algoritmus, alfa-béta vágás.

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 hátultesztelő ciklusokat. **Előltesztelő** ciklus a **for** és a **while** ciklus.

Hátultesztelő ciklus a **do - while**.

A **for** ciklust akkor használjuk, ha a ciklus törzsében szereplő utasításokat fixen tudjuk hányszor szeretnénk végrehajtani.

A for ciklus szintaxisa:

```
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, lefutnak a ciklus törzsében található utasítások. Ezután a léptető utasítás fut le, ez növeli/csökkenti a feltételben használt változó.

A **while** ciklus lényegében egy inicializációs és léptető utasítás nélküli for ciklus. A feltétel kiértékelődik majd, ha igaz az állítás lefutnak a törzsében lévő utasítások. 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 lefut a ciklus törzse, majd utána értékelődik ki a feltétel, amennyiben igazat kap, megismétli a törzsét.

Függvénykezelés, paraméterkiértékelés

A **függvények** olyan alprogramok, amelyeket 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 akkor **void** típust használunk. Következik a **függvény neve**, amely nem kezdődhet számmal. Ezután a **paraméterek** listája következik, ezeket **formális** paramétereknek nevezzük. Egy függvényt létrehozhatunk paraméterek nélkül is.

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

A **paraméterkiértékelés** 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.
Globális változó
Olyan változók, amelyek függvényeken kívül lettek deklarálva, általában a main függvény előtt. Bárhol elérhetők a programban.
Statikus változó
Olyan változók, amelyek a program futása során csak egyszer deklarálódnak és megtartják az értéküket különböző függvény hívások 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. Az matematikában használt számrendszer a 10-es számrendszer. A számítástechnikában a leggyakrabban használt számrendszer pedig a 2-es és 16-os számrendszer.

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

A **tizenhatos (hexadecimális) számrendszer**nek 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. 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.

Karakter, szöveg és logikai adat ábrázolása

A **nem-numerikus** karakterek ábrázolásához **kódtáblát** használunk. A karakterek kódolva, számként ábrázolhatók. Általánosan használt **kódrendszer** az **ASCII**, amely 1 Byte-on tárolja a karaktereket.

Az **angol nyelvben lévő betűknek, számjegyeknek és egyéb írásjelek** kódszáma 0 és 127 közé esik.

A **különböző nyelvek speciális karaktereinek** (ékezetes, görög és matematikai jelek) kódszáma 128 és 255 közé esik.

Az összes nyelv összes karakterét ábrázolni tudja a 16 bites Unicode kódolás.

Szöveget karakterek sorozataként tárolunk, tömbben. Egy szöveg végét \0-val zárjuk.

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

A heurisztika egy **állapot értékét** fejezi. A kezdőállapot értéke mindig 0. Minél közelebb járunk a célállapothoz, annál nagyobb a heurisztika értéke. Nincs tökéletes heurisztika.

A heurisztika segít kereső algoritmusoknak több lehetőség közül a becsült jobbat kiválasztani.

A* algortimus

A **legjobbat-először keresés** változata az A* keresés.

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

Egy **keresés teljessége** abból áll, hogy tetszőleges véges sok keresőlépés után képes-e előállítani egy megoldást, mely a kezdőállapottól a célba jutásig optimális költségek szerint lép.

Az **A* kereső teljes** keresés.

Kétszemélyes, teljes információjú, determinisztikus játékok

Teljes információjú játékokban a játék **minden eleméről** ismeretünk van. Például a 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.

Stratégia fogalma

A **stratégia** egyfajta előírás, amely egy állapotban meghatározza melyik operátort alkalmazza a gép.

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 vereséghez.

Minimax-algoritmus

A **minimax-algoritmus** az optimális döntést az aktuális állapotból számítja ki, felhasználva az egyes követő állapotok minimax értékeinek kiszámítását. 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 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 biztossá válik, hogy az aktuális csomópont értéke rosszabb lesz, mint az aktuális α és β érték, MAX-ra, illetve MIN-re.

α = az út mentén tetszőleges döntési pontban a MAX számára eddig megtalált legjobb választás értéke.

β = az út mentén tetszőleges döntési pontban a MIN számára eddig megtalált legjobb választás értéke.

3. TÉTEL

3. **A C programozási nyelv III.**: Tömbök, mutatók, dinamikus memóriakeresés. Karakterlánc-kezelés.
Formális nyelvek és automaták: környezetfüggetlen nyelvtanok, CNF, CYK algoritmus.
Hálózatok: adatkapcsolati protokollok, rétegek. Lokális hálózatok. Az internet alapjai, HTML.

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 (indexelése) 0-tól kezdődik, és ez a legtöbb programozási nyelvben is így van.

A **tömb indexelése** (indexing), más néven címzése szögletes zárójellel (bracket) történik.

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

A tömböket gyakran **ciklussal** dolgozzuk fel. Ilyenkor figyelni kell arra, hogy a **tömbindexek tartománya 0-tól méret-1-ig** terjed.

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

A tömb méretét változóval is megadhatjuk a scanf() használatával. Ezzel azonban vigyázni kell, ilyet csak ellenőrzött körülmények között szabad csinálni, mivel a felhasználó negatív számot is megadhat, vagy egy olyan óriási pozitív számot ad meg, amihez nincs elég memóriája a gépnek.

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

A meg nem adott méretű tömb értelmetlen és súlyos hibának számít.

Mutatók

A mutató (pointer) 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 mutató által hivatkozott változót ***p** módon érjük el, a memóriacímet **%p**-vel lehet kiíratni.

```
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 sztringek karakter típusú tömbként vannak értelmezve.
A sztring végét '\0' karakter jelzi.

Sztring létrehozása és inicializálása

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

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

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

Például: Az "alma" szó eltárolásához egy 5 elemű karaktertömbre van szükség. Négy nem elég, mert a lezáró nulla akkor már nem férne bele.

```
char szoveg[5] = "alma"
```

C nyelvben a karakterláncon értelmezhetők különböző parancsok:

- strcpy(s1, s2) – másolja az s2 sztring tartalmát az s1-be
- strcat(s1, s2) – s2 tartalmát az s1 végéhez toldja
- strcmp(s1, s2) – összehasonlítja s1 és s2 sztringeket karakterenként, ha igaz visszatér 0-val.
- 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.

- *malloc() függvény:* Lefoglal egy bájtban megadott méretű memóriaterületet visszaad egy pointert, ami a lefoglalt területre mutat, vagy NULL pointert ad, ha nem sikerült lefoglalni a területet, tehát memóriaszemetet tartalmaz.

  void *malloc(int size);

- *free() függvény*: Felszabadít egy memóriaterületet, amit a malloc() foglalt.

Példa:

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

Környezetfüggetlen nyelvtanok

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

Két legfontosabb alkalmazása:

- Természetes nyelvek feldolgozása.
- Programozási nyelvek szintaxisának megadása.

V_N: nemterminális → nagybetű
V_T: terminális → kisbetű

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 létre tudunk hozni 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 ∈ VN és a ∈ VT: 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 a nyelvtan Chomsky normál alakban (CNF) kell legyen. Az algoritmus egy tetszőleges bemenő szóhoz igyekszik megalkotni a megfelelő levezetési fát. Tehát **eldönti, hogy lehet-e generálni a megadott szót.**

Hálózat

Az adatkapcsolati réteg az OSI modell második rétege. Feladata az adatok megbízható továbbítása az adó és vevő között.

Az adatkapcsolati réteg tördeli szét az átküldendő információt **adatkeretekre** (**frames**). Feladata, hogy hibamentes adatátviteli vonalat alakítson ki, melyen az adatok eljutnak a hálózati réteghez. A kialakított kereteket sorrendhelyesen továbbítja, és 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.

Az adatkapcsolati réteg két alrétege:

- MAC-alréteg – Medium Access Control – közegelérési alréteg
- LLC-réteg – Logical Link Control – logikai kapcsolatvezérlés

Elemi adatkapcsolati protokollok

Szimplex: Az adatátvitel mindig csak egy irányban, az adótól a vevőhöz folyhat, csak egy irányban továbbíthatók az adatok. 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.

Fél-duplex: A vevő nem képes olyan sebességgel feldolgozni a kapott információt, amilyen sebességgel azt az adó küldte. Ezért valamilyen módon le kell lassítani az adót. A vevő 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. Ezt nevezik a "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. 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 LAN hálózat elemei

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

Kiszolgáló (szerver) gépek, kliens gépek, hálózati adapterkártyák, hálózati protokollok, modemek, forgalom irányítók (router), elosztók (switch).

Elterjedtebb topológiái:

- **Busz** (sín): Minden elem egy kábelre van felfűzve, mely a két végén lezáró elemmel van ellátva. Hátránya, hogy vonalszakadás esetén az egész hálózat használhatatlanná válik.
- **Csillag**: Egy központi vezérlő (HUB) kapcsolja össze a két kommunikálni kívánó gépet. 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 eleje és vége ugyan az, vagyis egy kört alkot. 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.

Az internet alapjai

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.

Az alhálózati maszk szintén 32 bitből áll. Az IP-címekhez hasonlóan az alhálózati maszkot is byte-onként 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 (Uniform Resource Locator - egységes erőforrás-azonosító) az interneten megtalálható bizonyos erőforrások szabványosított címe.

4. TÉTEL

4. **Adatszerkezetek és algoritmusok:** Funkcionális specifikáció. Programozási tételek: keresés, rendezés, döntés, kiválogatás.
Adatbázis-rendszerek: A relációs adatmodell. Egyed, attribútum, reláció és kapcsolat. Kulcs, idegen kulcs, hivatkozási integritás. Kényszerfeltételek az adatbázis elemein. Triggerek.
Számításelmélet: Turing-gépek, Church-tézis, megállási probléma, algoritmikusan eldönthetetlen problémák. Logikai függvények megadása, KNF, DNF, logikai hálózatok. Tár- és időbonyolultság.

Funkcionális specifikáció

Megelőzi az adatszerkezet meghatározása az algoritmus meghatározását.

A programtervező specifikálhatja a megírandó program megengedett futási idejét, az igénybe vehető tárrész méretét.

Az ilyen és ehhez hasonló specifikációk arra késztetik a programozót, hogy implementálás közben olyan hatékony algoritmikai megoldásokat keressen, amelyek eleget tesznek az igényeknek.

Programozási tételek: Lineáris keresés

Eldönti, hogy van-e adott tulajdonságú elem a sorozatban, és ha van, akkor megadja a sorszámát. (Ennyivel több, mint az eldöntés tétele.)

```
keresett = 30          //keresett elem
i = 0                  //ciklusváltozó
ciklus amíg i<n és t[i]!=keresett  //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 hogy nincs
ha vége
```

Programozási tételek: Logaritmikus keresés

Bináris keresés másnéven. 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 van a keresett szám, annak 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
```

Programozási tételek: Döntés

Eldönti, hogy van-e adott tulajdonságú elem a sorozatban.

```
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
```

Programozási tételek: Kiválogatás

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

A tömb elemeit egy másik tömbbe helyezzük, feltételhez kötve.

Például: Adott *A* és *B* tömb. Az *A* tömb egész számokat tartalmaz. Az A tömbből az 5-nél kisebb számokat kigyűjtjük B tömbbe.

Buborékos rendezés

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

```
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 **relációs adatmodell**ben az adatokat kétdimenziós táblákban (relációkban) tároljuk. A táblázat soraiban található elemek alkotnak egymással relációt.

A **rekord** a tábla egy **sor**a. A **mező** a tábla egy **oszlop**a.

Egyed (entitás), amelynek az adatait gyűjteni és tárolni akarjuk az adatbázisban.

Attribútum (tulajdonság), az egyed valamely jellemzője.

Reláció-kapcsolat: Az egyedek közötti viszonyt, összefüggést jelenti.

Egy-egy kapcsolat: 1:1 kapcsolat esetén minden egyes egyedhez pontosan egy másik egyed tartozik.

Például mindenkinek csak egy személyi száma van, és minden egyes személyi számhoz csak egy ember tartozik.

Egy-több kapcsolat: 1:N kapcsolat ("egy a sokhoz" kapcsolat) esetén az egyik egyedhez több másik egyedet tudunk társítani, de a másik csoport minden egyes példányához pontosan egyet társítunk.

Például minden oktatónak több hallgatója lehet, de egy hallgató csak egy oktatónál készítheti el a szakdolgozatát.

Több-több kapcsolat: N:M kapcsolat ("sok a sokhoz" kapcsolat) esetén egy egyed példány több másikkal áll relációban, és ez fordítva is igaz.

Például: Egy hallgató több kurzusra jelentkezik és egy kurzust többen is felvehetnek.

Kulcs: Az egyed azon tulajdonsága mely már egyértelműen meghatározza az egyedet.

Például az emberek esetén a személyi szám.

Elsődleges kulcs: Azok a mezők, melyek egyértelműen azonosítják a tábla rekordjait, értéke egyedi.

- Egyszerű kulcs: ha egy attribútumból áll.
- Összetett kulcs: ha több attribútumból áll.

Idegen kulcs: Az idegen kulcs egy olyan azonosító, amellyel egy másik tábla elsődleges kulcsára hivatkozhatunk.

A **hivatkozási integritás**: Az adatbázisban nem lehet olyan idegen kulcs, mely nem egyezik meg a hivatkozott tábla valamelyik elsődleges kulcsával. Ennek az adatbázist érintő minden változtatás után is érvényben kell maradnia.

Kényszer (constraint): A lehetséges adatok halmazát leíró, korlátozó szabály.

NOT NULL	*Nem lehet NULL*
UNIQUE	*Egyedi*
PRIMARY KEY	*Elsődleges kulcs*
FOREIGN KEY	*Idegen kulcs*
CHECK	*Feltétel*
DEFAULT	*Alapértelmezett érték*

Trigger

A **trigger** olyan tevékenységet definiál, amely **automatikusan végbemegy**, ha egy tábla vagy nézet módosul, vagy ha egyéb felhasználói vagy rendszeresemények következnek be. A trigger adatbázis-objektum.

Turing-gépek

A **Turing-gép** egy **potenciálisan végtelen szalagmemóriával és egy író-olvasó fejjel rendelkező véges automata.** A szalagmemória pozíciókra van osztva, és minden egyes pozíció, mint memória-egység az úgynevezett szalagábécé pontosan egy betűjének tárolására képes.

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.

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. Az **univerzális Turing-gép** egy általános, elvont számítógép, ami minden Turing-gépet képes szimulálni. Ez azt jelenti, hogy van olyan gép, ami minden kiszámítható függvényt ki tud számolni.

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.

Input szó hatására el tud-e jutni egy végkonfigurációba, avagy sem. Ha ilyen Turing-gép nem létezik, akkor mondjuk azt, hogy a Turing-gépek megállási problémája megoldhatatlan.

A **megállási probléma** kérdése az, hogy egy Turing-gép adott bemenettel egyáltalán megáll-e.

Logikai függvények megadása

A logikai függvények megadása a következő módokon lehetséges:

Szöveges megadás: Az alapfeltételek kombinációit, a logikai kapcsolatot és a következtetéseket egyaránt szavakban fogalmazzák meg.

Táblázatos megadás: Olyan értéktáblázatot hoznak létre, amely tartalmazza az alapfeltételek minden kombinációjához tartozó következtetések értékeit.

Halmazokkal történő leírás: Az alapfeltételekhez tartozó következtetések közötti függvénykapcsolatot 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.

A logikai függvények az időfüggésük szerint lehetnek **időfüggetlen**, és **időfüggő** logikai függvények.

- A **kombinációs hálózatok** időfüggetlen logikai függvényeket valósítanak meg. Memória nélküli logikai áramkörök.
- A **sorrendi (szekvenciális)** hálózatok időfüggő logikai függvényeket valósítanak meg. Memóriával is rendelkező logikai áramkörök.

5. TÉTEL

5. **Adatszerkezetek és algoritmusok:** sor, verem, hiányos mátrix, láncolt lista, bináris fák műveleteinek algoritmusai.
Operációs rendszerek: Folyamatkezelés és -ütemezés. Memóriakezelés. Állománykezelés.
Formális nyelvek és automaták: Az üresszó-lemma. Véges automata fogalma, fajtái, véges automaták determinisztikussá tétele.

Adatszerkezetek: sor, ver

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

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

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

Alapműveletei:

- 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 méretével.

Láncolt lista

Láncolt lista (linked list): Adatszerkezet, ahol az egyes elemek (node) láncba vannak fűzve azáltal, hogy tárolják a szomszédos elem címét. Tetszőleges számú elem tárolására, gyűjtésére ad lehetőséget.

Előnye a tömbbel szemben, hogy eltérő típusú és méretű elemeket is képes magába foglalni.

Hátránya, hogy az elemek véletlenszerűen, illetve sorszámuk alapján közvetlenül nem, csak a lista bejárásával érhetők el és címezhetők meg. Lassabb működést eredményez, mintha 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.

Minden adatelem tartalmaz két mutató típusút is, melyek az elem bal illetve jobb oldali leszármazottjára mutatnak.

Ha a leszármazott nem létezik, akkor az adott oldali mutató NULL értéket vesz fel.

Bináris fa műveletei

Létrehozás: Létrehozunk egy dinamikusan kezelhető adatstruktúrát.

Bővítés: Hozzáadunk egy elemet a meglévő struktúrához.

Bejárás: Az adatszerkezet valamennyi elemének egyszeri elérése (feldolgozása).

- Preorder bejárás: tartalom, bal, jobb
- Inorder bejárás: bal, tartalom, jobb
- Postorder bejárás: bal, jobb, tartalom

Operációs rendszer

Az **operációs rendszer** programok gyűjteménye, amelyek elősegítik a számítógép hardverének könnyű, sokoldalú és biztonságos használatát.

Folyamatkezelés, **ütemezés**: 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.

Memóriakezelés: Az op. rendszer szemszögéből a memóriát egy bájtokból álló tömbnek tekinthetjük. Az operációs rendszernek nyilván kell tartani, hogy az operatív memória melyik részét ki használja és mire. El kell döntenie, hogy a felszabadult memóriaterületre melyik folyamatot tölti be.

Állománykezelés

Az informatikában adatállománynak, állománynak vagy fájlnak nevezzük a **logikailag összefüggő adatok halmazát,** melyek egy közös névvel rendelkeznek. Tárolásuk **bármilyen adathordozón** történhet.

A **fájlok tartalma** lehet szöveg, numerikus adat, grafika, hang stb.

Kétféle fájlt különböztetünk meg:

- Programfájlok: exe, bat, com
- Adatfájlok: txt, docx, xlsx, pdf, ppt, html, jpg, png, zip, rar stb.

Tárolásuk alapján beszélhetünk **tömörített** és **tömörítés nélküli** fájlokról.

Az **állománykezelő feladatai:**

- információátvitel
- műveletek az állományokon és a könyvtárakon
- osztott állománykezelés
- hozzáférés szabályozása (más felhasználók által végezhető műveletek korlátozása)
- tárolt információk védelme illetéktelen olvasók ellen
- információk védelme a sérülések ellen
- mentés

A **fájlokkal történő műveletek**: megnyitás, létrehozás, törlés, visszaállítás, másolás, áthelyezés, átnevezés, nyomtatás.

Üresszó lemma

Minden környezetfüggetlen (2-es típusú) nyelvtanhoz megadható vele ekvivalens környezetfüggő (1-es típusú) nyelvtan.

Minden környezetfüggetlen G grammatikához megadható olyan G' környezetfüggetlen nyelvtan, hogy L(G)=L(G') (azaz az általuk generált nyelv ugyanaz), és ha $\lambda \notin L(G)$, akkor a G'-beli szabályok jobboldalán λ nem fordul elő. Ha viszont $\lambda \in L(G)$, akkor az egyetlen G'-beli szabály, aminek jobboldala az üresszó $S' \rightarrow \lambda$, ahol S' a G' mondatszimbólumát jelöli. $\lambda \in L(G')$ fennállása esetén S' 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.

Amennyiben az átmeneti és a kimeneti függvények nem egyértelműen definiáltak, **nemdeterminisztikus** automatáról van szó.

Üresszóátmenetes (nemdeterminisztikus) **automata:** Ha a nemdeterminisztikus automatáknak megengedjük, hogy az automata bemenő jel nélkül is állapotot váltson.

Determinisztikus automata esetén a szóban forgó függvényértékek mindig pontosan egy meghatározott értéket vesznek fel. Determinisztikus automatáknál nem fordulhat elő üresszóátmenet.

6. TÉTEL

6. **Objektum-orientált programozás:** OOP, típusok és konverzióik, operátorok, utasítások. Metódusok, osztálykészítés, láthatóság, konstruktor.
Formális nyelvek és automaták: Ábécé, szó, nyelv, nyelvtan fogalma. Chomsky-féle nyelvtani osztályok és az általuk generált nyelvosztályok tartalmazási hierarchiája.
Számítógép-architektúrák: logikai áramkörök, kombinációs logikai hálózatok (fél és teljes összeadó, multiplexer, demultiplexer, dekóder).

Objektum-orientált programozás (OOP)

Az **objektum** egységbe foglalja az adatokat és a hozzájuk tartozó műveleteket.

Az **objektum-orientált programozás (OOP)** az objektumokat és a köztük fennálló kölcsönhatásokat használja alkalmazások és számítógépes programok tervezéséhez. Az OOP olyan megoldásokat foglal magában, mint az **egységbezárás**, **öröklődés, polimorfizmus**.

Egységbezárás: Az adatok és hozzájuk tartozó eljárások egyetlen egységben való kezelését jelenti, objektumban vagy osztályban.

Öröklődés: Azt jelenti, hogy egy objektum egy másik objektum tulajdonságait, metódusait megkapja. Az örökölt 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.

Polimorfizmus: 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.

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 **primitív típusok** a következőek:

1. táblázat: Primitív adattípusok [15]

Típus	Leírás	Méret/Formátum
byte	bájt méretű egész szám	8 bit
short	rövid egész szám	16 bit
int	egész szám	32 bit
long	hosszú egész szám	64 bit
float	egyszeres pontosságú lebegőpontos valós szám	32 bit
double	dupla pontosságú lebegőpontos valós szám	64 bit
char	Unicode karakter	16 bit
boolean	logikai érték	true vagy false

Referencia adattípusok közé tartoznak a sztringek, a tömbök, az osztályok, és az interfészek.

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

Az **implicit konverzió** esetén a Java automatikusan átkonvertálja az egyik típust a másikra.

Az **explicit konverzió** olyankor történik, amikor a kódban megjelöljük (rákényszerítjük) az adott típusra a változót. (Ha egy típust egy kisebb méretű típusra szeretnénk átváltani.)

Operátorok

Az **operátorok** egy, kettő vagy három operanduson hajtanak végre egy műveletet.

A Javában egy művelet operandusai **mindig balról jobbra** értékelődnek ki.

Javában a következő operátorokkal dolgozhatunk:

- **Aritmetikai** operátorok: Alapvető matematikai műveletek végzésére használjuk őket.
  + (összeadás), - (kivonás), * (szorzás), / (osztás), % (maradékképzés)
- **Relációs** operátorok: Összehasonlít két értéket és meghatározza a köztük lévő kapcsolatot.
  > (nagyobb), >= (nagyobb vagy egyenlő), < (kisebb), <= (kisebb vagy egyenlő),
  == (egyenlő), != (nem egyenlő)
- **Értékadó** operátorok: Az alap értékadó (=) operátort használhatjuk arra, hogy egy értéket hozzárendeljünk egy változóhoz.
  -=, *=, /=, %=, &=, |=, ^=, <<=, >>=
- **Logikai** operátorok: Két érték közötti logikát vizsgálja.
  && (logikai és), || (logikai vagy), ! (logikai nem)
- **Inkrementáló** (növelő) operátorok: Létezik két speciális operátor, mely egy változó értékének 1-gyel való növelésére és csökkentésére szolgál.
  ++ (növelés), -- (csökkentés)
- **Bitenkénti logikai** operátorok:
  & (bitenkénti és), | (bitenkénti vagy), ^ (bitenkénti kizáró vagy), ~ (bitenkénti tagadás)
- **Egyéb** operátorok
  ? : (feltételes operátor), [ ] (tömbképző operátor),new (új objektum létrehozása)

Utasítások

Az utasítás a programkód egy lépését adják meg, vannak:

- kifejezés utasítások
- ciklus utasítások
- feltételes utasítások
- összetett utasítások (blokk)
- változó deklaráció
- növelő és csökkentő utasítások
- metódushívások
- objektumot létrehozó kifejezések

Metódusok

A **metódusok** az **osztályok tagfüggvényei**, amik lehetnek példánymetódusok, vagy osztálymetódusok (static kulcsszóval deklaráljuk). Egy metódust két részből áll: a **metódus deklarációja** és a **metódus törzse**.

A **metódusdeklaráció** meghatározza a metódus összes tulajdonságát.

A **metódustörzs** 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.

Túlterhelés (overloading): Egy osztályon belül lehet több azonos nevű metódus, melyek a paraméterezésben és/vagy a visszatérési érték típusában térnek el egymástól.

Osztálykészítés

Az osztály definíciója 2 fő alkotóelemből áll: Az **osztálydeklaráció**ból, és az **osztálytörzs**ből.

public class Iskolabal {}

Az **osztálytörzs** az osztálydeklarációt követi, és kapcsos zárójelek között áll. Az osztály törzs tartalmazza mindazt a kódot, amely hozzájárul az osztályból létrehozott objektumok életciklusához: tagváltozók, konstruktorok, metódus deklarációk

Láthatóság

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

- A **private** elérhetőség esetén csak az eredeti osztály érheti el a tagokat.
- A **csomag** szintű elérhetőség azt jelenti, hogy az eredeti osztállyal azonos csomagban lévő osztályok elérhetik a többi osztály tagjait.
- A **protected** elérhetőség azt jelenti, hogy az osztály leszármazottjai elérhetik a tagokat, függetlenül attól, hogy melyik csomagban vannak.
- A **public** elérhetőség azt jelenti, hogy az összes osztály elérheti a tagokat.

Konstruktor

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.

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 halmaza. Egy **p szó hosszát |p|** -al jelöljük.

Üresszóról olyan szót jelent, melynek egyetlen betűje sincs. 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.

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

Logikai áramkörök

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

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 **multiplexer**nek n vezérlőbemenete van, egy adatkimenete és 2n adatbemenete. A multiplexer feladata az, hogy a több bemenetére érkező jelből egyet vezessen a kimenetére.

A multiplexer fordítottja a **demultiplexer**, amely egy egyedi bemenő jelet irányít a 2n kimenet valamelyikére az n vezérlővonal értékétől függően, a többi kimenet 0.

A **dekódoló** működését tekintve egy kapcsoló, amely az n-bites bemenete által kiválasztott 2n-kimenet közül aktivál egyet.

Egy számítógép egyik alapvető művelete az összeadás. Ezt **félösszeadók** segítségével végzik el a gépek.Egy **teljes összeadó** két fél összeadóból épül fel.

7. TÉTEL

7. **Objektum-orientált programozás:** Öröklődés, túlterhelés, polimorfizmus. Kivételkezelés. **Adatbázis-rendszerek:** Nézettáblák relációs adatbáziskezelőkben. Indexelés a táblákon – mikor használjuk? Az adatbázis-tervezés elmélete: funkcionális függőségek és normalizáció – Boyce-Codd normálforma (BCNF). Anomáliák nem normalizált adatbázissémák esetén. Az E/K modell és átfordítása relációs adatmodellé. **Programozási technológiák:** Nyomkövetés és hibakeresés, egységtesztelés, naplózás. Kollekciók használata, relációs adatbázisok kezelése OO programozási nyelvekben.

Öröklődés, túlterhelés, polimorfizmus

Az **objektum** egységbe foglalja az adatokat és a hozzájuk tartozó műveleteket.

Az **objektum-orientált programozás (OPP)** az objektumokat és a köztük fennálló kölcsönhatásokat használja alkalmazások és számítógépes programok tervezéséhez. Az OOP olyan megoldásokat foglal magában, mint az **egységbezárás**, **öröklődés, polimorfizmus**.

Egységbezárás: Az adatok és hozzájuk tartozó eljárások egyetlen egységben való kezelését jelenti, objektumban vagy osztályban.

Öröklődés: Azt jelenti, hogy egy objektum egy másik objektum tulajdonságait, metódusait megkapja. 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.

Polimorfizmus: 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.

Túlterhelés (overloading): Egy osztályon belül lehet több azonos nevű metódus, melyek a paraméterezésben és/vagy a visszatérési érték típusában térnek el egymástól.

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 **kivételnek** hívjuk.

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

Nézettáblák

A **nézettábla** az adatbázisban lévő reláción vagy relációkon végrehajtott művelet eredményét tartalmazó új tábla, amely valóságban nem léteznek fizikailag az adatbázisban.

A nézettábla csak **egy utasítássorozatot tárol**, amely meghatározza, hogy a teljes adatbázis mely részhalmazát kell megjelenítenie, tehát valójában **nem tárol adatot**.

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

Indexelés

Az indexelés célja a **lekérdezések gyorsítása**.

Az index segítségével az adatbázis-kezelő rendszernek nem kell a tábla minden sorát végig néznie, hanem csak egy töredékét.

A keresés gyorsítható, ha a vizsgált tábla rekordjai a WHERE feltételek szempontjából legalább részlegesen rendezettek (indexeltek).

Az INDEX-ként emlegetett mechanizmus **B-fa** (B-Tree) típusú indexstruktúrát hoz létre a tábla rekordjai felett, a kulcsmezők alapján.

Egy tábla létrehozása után alaphelyzetben semmilyen indexeléssel sem rendelkezik kivéve, ha a tábla valamelyik mezője UNIQUE vagy PRIMARY KEY típusú.

Indexek használatával csökken a keresési idő, nő a tárméret, és nő a módosítási idő.

Az adatbázis-tervezés elmélete

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

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

Tranzitív funkcionális függés: A → C, ha, A → B és B → C. (C tranzitíven függ A-tól, ha B funkcionálisan függ A-tól és C funkcionálisan függ B-től.)

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 normálformában** van, ha minden attribútuma egyszerű, nem összetett adat.

Második normálforma, ha

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

Harmadik normálforma, ha

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

BCNF (Boyce-Codd normálforma), ha

- A reláció harmadik normálformában 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 (entitás), amelynek az adatait gyűjteni és tárolni akarjuk az adatbázisban.

Reláció-kapcsolat: Az egyedek közötti viszonyt, összefüggést jelenti.

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

Hibakeresés (debugging)

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. A másikban a kiírandó változókat rögzítjük, és értékük a futás során mindig megnézhető.

Nyomkövetés

A **nyomkövetés** lényege a végrehajtott utasítások követése a programban. Futás során az eredmény mellett a programszöveget is látnunk kell. A nyomkövetés általában sokféle információt adhat a program futásáról: a végrehajtott utasítás mellett kiírhatjuk a képernyőre annak hatását.

Adatok nyomkövetése

A változókat vizsgáljuk. Ebben az esetben akkor kapunk a képernyőn üzenetet, ha a kijelölt változó(ka)t valaki használja, illetve módosítja.

Töréspontok elhelyezése

A töréspontok a program olyan utasításai, amelyeknél a végrehajtásnak meg kell állnia, a felhasználó információt szerezhet a program állapotáról. A felhasználó dönthet a futtatás abbahagyásáról, illetve folytatásáról. Megnézheti változók értékeit, nyomkövetést be- és kikapcsolhat, töréspontokat megszüntethet, illetve újakat definiálhat.

Lépésenkénti végrehajtás

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

Egységtesztelés

Az **egységtesztelés** (unit testing) egy olyan ellenőrző programrészlet, amely vizsgálja az adott metódusok visszatéréseinek lehetőségeit. Egy metódusra több tesztet is lehet írni.

Naplózás (logging)

A **naplózást** általában fontosabb események vagy állapotok leírásához használják fel. Látni lehet a program működését, esetleges hibáit. Fel lehet használni hibakeresés során, amit rögtön a konzol felületen tüntetünk fel, vagy akár fájlba is írhatjuk egy működő program, rendszer során.

Kollekciók használata

A **kollekció** meglévő adatszerkezetek tulajdonságait felhasználva foglal össze egyedi elemeket, objektumokat.

A **List** (interface) elemeket tárol adott sorrendben, iterálható módon.

Megvalósító osztályok:

- ArrayList: tömbbel megvalósított lista
- LinkedList: láncolt lista

A **Set** (interface) egyedi elemeket tárol sorrendiség nélkül, a tárolt elemeknek felül kell definiálnia az equals metódust.

Megvalósító osztályok:

- HashSet: gyors keresést biztosító halmaz
- TreeSet: rendezett halmaz (fával megvalósított)

A **Map** (interface) kulcs-adat objektum párok csoportja, a kulcsra gyors keresést biztosít.

Megvalósító osztályok:

- 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 programozási nyelvek biztosítanak csatlakozási lehetőséget adatbázis rendszerekhez különböző úgynevezett 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.

Az objektum-orientált programozási nyelvek igyekeznek egységben tartani az összefüggő adatokat, míg a relációs adatbázisok külön táblákba csoportosítanak adatokat.

Erre adnak megoldást a különböző ORM (Object-Relational Mapping) rendszerek, amelyek automatikusan megoldják a konverziókat.

8. TÉTEL

8. **Objektum-orientált programozás:** Osztály- és példány inicializálás, konstruktor. Interfészek. Generikus programozás, összetett adatszerkezeteket implementáló osztályok és fontosabb műveleteik.
A rendszerfejlesztés technológiája: rendszerfejlesztési modellek, tervezés, tesztelés, UML osztálydiagram. Verziókezelés.
Számítógép-architektúrák: A mikroelektronika alapjai (félvezetők, dióda, tranzisztorok, fajtái és az általuk megvalósítható kapuk). A CPU és felépítése. Integrált áramkörök. Memóriák fajtái, csoportosításuk.

Osztály- és példány inicializálás

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

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

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.

Interfész

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 deklarációban két elem kötelező: az interface kulcsszó és az interfész neve. Ez után szerepelhetnek a szülőinterfészek.

Generikus programozás

A generikus programozás egy általános programozási modellt jelent.

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.

Összetett adatszerkezeteket implementáló osztályok és fontosabb műveleteik

ARRAYLIST: Tömbbel megvalósított lista, gyors elérés, de lassú beszúrás/törlés.

LIST: A **List** (interface) elemeket tárol adott sorrendben, iterálható módon.

SET: A **Set** (interface) egyedi elemeket tárol sorrendiség nélkül, a tárolt elemeknek felül kell definiálnia az equals metódust.

MAP: A **Map** (interface) 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 vagy elemeket)
- isEmpty() (igaz, ha üres)
- remove()/removeAll() (elem, vagy elemek törlése)
- size() (elemek száma)
- iterator (iterátort készít)

Map fontosabb műveletei:

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

Rendszerfejlesztési modellek

Vízesés modell

A vízesésmodell a nevét onnan kapta, hogy az egyes lépések lépcsőzetesen kapcsolódnak egymáshoz, mint ahogy a víz folyik le a sziklán.

Előnye, hogy kicsi és közepes méretű rendszereknél jól strukturált, jól felépített, robusztus rendszer fejleszthető.

Hátránya, hogy minden egyes folyamat teljeskörűen lezárul, mielőtt a következő folyamat elindulna.

Evolúciós modell

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

Spirális fejlesztési modell

1. lépés: célok meghatározása

2. lépés: kockázatelemzés, kockázatok felismerése, megszüntetése

3. lépés: fejlesztés és validálás

4. lépés: áttekintés, döntés a folytatásról

Tervezés

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

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

Megvalósíthatósági tanulmány:

A rendszer nagyvonalú leírását tartalmazza.

Követelmények feltárása:

A leendő felhasználóktól beszerzett információk alapján 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 tesztelést követi a hibák behatárolása, a debuggolás, és a regressziós tesztelés, azaz, hogy a hiba javítása nem-e befolyásolt más, eddig működő rendszereket negatívan.

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: Osztályok vagy metódusok együttműködését vizsgálja.

Integrációs teszt: Azt vizsgálja, hogy 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 funkcionális követelményeket teljesíti-e.

Nem funkcionális tesztelésbe tartozik a teljesítménytesztelés, például adott válaszidő alatt teljesül-e valami, vagy UI tesztelés, azaz használható-e a program, egyértelmű-e.

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.

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.

Osztálydiagram (Class diagram): Az osztályokat és a közöttük lévő kapcsolatokat ábrázolja.

Használati eset diagram (Use case diagram): Modellezi a felhasználó(k) által kiváltható eseményeket.

Verziókezelés

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

Olyan adatok verzióinak kezelésére, amelyeken több ember dolgozik egyidejűleg.

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

Mikroelektronika

A **félvezetők** gyengén vezetnek, vezetőképességük a szigetelőkénél nagyságrendekkel nagyobb, de még mindig olyan kicsi, hogy szigetelőnek tekinthető.

A leggyakrabban használt félvezető a szilícium (Si).

A **dióda** olyan elektronikai alkatrész, amelyet egyenirányításra, illetve egyszerűbb logikai kapuáramkörökben alkalmazunk.

A **tranzisztor** egy szilárdtest félvezető, amelyet elektronikus áramkörökben használnak erősítési és kapcsolási célokra, **bipoláris** tranzisztor és **unipoláris** tranzisztor

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.

CPU

A **CPU** (Central Processing Unit) feladata a memóriában elhelyezkedő program feldolgozása és végrehajtása. A CPU sebességét az órajel periódusok száma jelenti.

A CPU által feldolgozható adatok mennyisége a processzor adatbuszának szélességétől függ. A jelenlegi processzorok 32 vagy 64 bites adatbusszal rendelkeznek.

CPU főbb részei

ALU: Aritmetikai és Logikai Egység. A számolási, összehasonlítási, logikai műveleteket végzi.

CU: A processzor **vezérlő egységének** feladata a program utasításai alapján a gép részeinek irányítása.

Regiszterek: A processzorok ideiglenes adattárolási céljaira szolgálnak.

Cache: **Gyorsító tár**, melynek célja az információ-hozzáférés gyorsítása.

Memória

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

A memóriákat fizikai szempontból két csoportra osztjuk:

- ROM: Adatok írására nincs lehetőség. Csak olvasható.
1. RAM: A futó programok ide töltődnek be. Írható és olvasható.

9. TÉTEL

9. **SQL:** Adatdeklarációs résznyelv (DDL), a CREATE TABLE és ALTER TABLE utasítás lehetőségei. Adatlekérdező nyelv (SELECT): rendezés, szűrés, csoportosítás, többtáblás lekérdezések, az INNER JOIN és OUTER JOIN különbsége. Adatmódosító (DML) résznyelv: INSERT, UPDATE, DELETE. Beágyazott allekérdezések lehetőségei: IN, EXISTS, ALL, ANY. Kapcsolt allekérdezés.
Programozási technológiák: Tervezési minták egy OO programozási nyelvben. MVC, mint modell-nézet-vezérlő minta és néhány másik tervezési minta.
Hálózatok: Topológiák és architektúrák. Az OSI modell. Fizikai átviteli jellemzők és módszerek, közeg hozzáférési módszerek.

Adatdefiníciós résznyelv (DDL)

Adatdefiníciós résznyelv utasításai objektumok létrehozására, módosítására, törlésére valók.

- 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** 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**.

Megszorítások például: NOT NULL, UNIQUE, PRIMARY KEY, DEFAULT.

Adatlekérdező nyelv (DQL)

Adatlekérdező nyelvvel a letárolt adatokat tudjuk visszakeresni. A lekérdező nyelv egyetlen utasításból áll, ez pedig a **SELECT,** mely számos alparancsot tartalmazhat.

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

Az SQL nyelv lehetőséget biztosít az összekapcsolásra is (JOIN).

<táblanév> JOIN <táblanév> ON <feltétel>

Alapértelmezetten 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.

Adatmanipulációs nyelv (DML)

Adatmanipulációs nyelv rekordok felvitelére, módosítására és törlésére alkalmazható.

Egy **INSERT INTO** utasítás segítségével egy sor adható meg az adott relációhoz.

Az **UPDATE** utasítás segítségével a relációk több sorát is módosíthatjuk. A SET után adjuk 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 WHERE alparancs elmaradása esetén a reláció összes sora törlődik.

Adatvezérlő nyelv (DCL)

Adatvezérlő nyelv az adatvédelmi és a tranzakció-kezelő műveletek hajtja végre.

Az SQL nyelv lehetővé teszi a SELECT utasítások egymásba ágyazását.

ALL: Akkor teljesül, ha az alszelekt által visszaadott összes sorra teljesül a feltétel.

ANY: Akkor teljesül, ha az alszelekt által visszaadott sorok közül legalább egyre teljesül a feltétel.

IN: Akkor teljesül, ha az IN előtt álló kifejezés szerepel az alszelekt által visszaadott sorok közt.

EXISTS: Akkor teljesül, ha az utána álló alszelekt legalább 1 sort visszaad.

Kapcsolt allekérdezés

Kapcsolt allekérdezés esetén a külső SELECT-nél kezdődik a kiértékelés, átadja a hivatkozott értéket a belső SELECT-nek, ezután a belső SELECT által előállított értékkel folytatódik a külső SELECT kiértékelése.

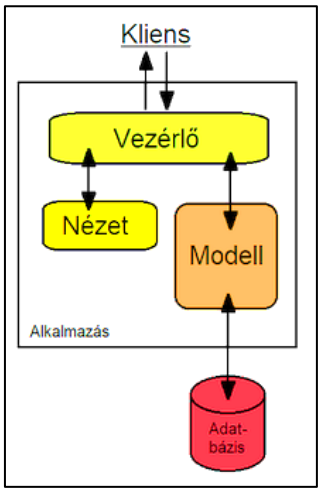
Tervezési minták egy OO programozási nyelvben

A tervezési minták sokszor felmerülő problémákra adnak megoldásokat. Objektum orientált programok esetében a tervezési minta leírása megadja azokat az egymással kommunikáló objektumokat, osztályokat, amelyek együttes viselkedése az adott problémára megoldás lehet.

Tervezési minta elemei:

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

MVC



A ***Model*** réteg felelős az adatszerkezetek definiálásáért, az adatok tárolásáért, kezeléséért és módosításáért, valamint az adatbázisért. Itt tároljuk az üzleti logikát is.

A ***View*** réteg nem mást jelent, mint a felhasználói felületet, melynek feladata, hogy megjelenítse a *Model* réteg tartalmát.

A ***Controller*** réteg feladata, hogy kapcsolatot teremtsen a *Model* és a *View* réteg között. A **Controller** nincs mindig külön rétegként megvalósítva, előfordul néha, hogy összeolvad a *View* réteggel.

További tervezési minták

- Gyártási minták (Pl.: gyártófüggvény)
- Szerkezeti minták (Pl.: Adapter)
- Viselkedési minták (Pl.: Iterátor)

Topológiák és architektúrák

A hálózati topológia a hálózat struktúráját adja meg.

- **Busz** (sín): Minden elem egy kábelre van felfűzve, mely a két végén lezáró elemmel van ellátva. Hátránya, hogy vonalszakadás esetén az egész hálózat használhatatlanná válik.
- **Csillag**: Egy központi vezérlő (HUB) kapcsolja össze a két kommunikálni kívánó gépet. 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 eleje és vége ugyan az, vagyis egy kört alkot. 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.

OSI modell

- Fizikai réteg: Átviteli közegek tulajdonságaival, a jelátvitel megvalósításával foglalkozik.
- Adatkapcsolati réteg: Megbízható jelátvitel két közvetlenül összekötött eszköz között.
- Hálózati réteg: Összeköttetés két hálózati csomópont között. (IP)
- Szállítási réteg: Megbízható összeköttetés két csomóponton lévő szoftver között.
- Viszony réteg: Végfelhasználók közötti logikai kapcsolat felépítése, bontása.
- Megjelenítési réteg: Az információ azonos módon történő értelmezése a kapcsolat mindkét oldalán.
- 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 átviteli közegeknek nevezzük.

Vezetékes rendszerek: Elektromos, vagy fény impulzusok továbbítására alkalmas kábelek kötik össze a számítógépeket.

Csavart érpár: Egy kábel általában több érpárt tartalmaz.

- STP (Shielded Twisted Pair): Árnyékolt sodrott érpár.
- UTP (Unshielded Twisted Pair): Árnyékolatlan sodrott érpár. **Napjaink legelterjedtebb kábele.**

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

Vezeték nélküli technológiák:

- infravörös kommunikáció
- rövidhullámú, rádiófrekvenciás átvitel (WiFi, Bluetooth)
- mikrohullámú átvitel
- lézer
- műholdas átvitel

Közeg hozzáférési módszerek

Függ az hálózat topológiájától, vagyis attól, hogy milyen módon vannak az állomások összekapcsolva.

3 fő hozzáférési módszer lehetséges:

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