1. Adatok, adattípusok, adatműveletek és adatstruktúrák. Szám adattípus. Számrendszerek, konverziók. A logikai, halmaz, karakter, sztring absztrakt adattípusok és realizációjuk. A tömb (vektor, mátrix), rekord, egyéb absztrakt adattípusok.

Adat: minden olyan információ, amelynek segítségével leírunk egy jelenséget, tanulmányunk tárgyát, vagy annak egy részét (általános megfogalmazásban: valamely előre rögzített halmaznak az eleme).
Adattípusok: Az absztrakt adattípus egy leírás, amely absztrakt adatok halmazát és a rajtuk végezhető műveleteket adja meg, nem törődve azok konkrét (gépi) realizálásával. (pl. természetes, egész számok)
- megadásuk: **T = (A, M)**, ahol T az absztrakt adattípus, A az adatok halmaza és M a műveletek halmaza
Adatműveletek:
- halmazok Descartes-szorzata: A és B halmazok Descartes szorzatán azt a C = A × B halmazt értjük, amelyre C = {(a, b), a ∈ A, b ∈ B}.
- halmaz hatványa: ha $A^1 = A$, ha $A^2 = A \times A$, ha $n \in \mathbb{N}$ és $n > 2$, akkor $A^n = A^{n-1} \times A$.
- n-áris művelet A halmazon: a következő leképezés (függvény): $f : A^n \rightarrow A$.
Adatstruktúrák: Adatstruktúrának nevezzük az absztrakt adattípus konkrét megjelenési formáját.
Szám adattípus: természetes szám: T = ($\mathbb{N}$, M), ahol $\mathbb{N}$ = {0, 1, 2, 3, ...} az adatok (természetes számok) halmaza és M = {+, ·}

Számrendszerek: a számrendszer meghatározza, hogyan ábrázolható egy adott szám
- informatikában használt számrendszerek:
    - bináris számrendszer: T = {0, 1}
    - decimális számrendszer: T = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9}
    - hexadecimális: T = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F}

Konverzió számrendszerek közt: Maradékos osztások ismételt alkalmazásával: minden lépés után a következő lépésben a kapott hányadost osztjuk tovább, egészen addig, míg a hányados 0 lesz, ekkor az osztási maradékok összeolvasva adják az új számrendszerben a számot.
Tétel a számjegyek számáról: Pozitív x egész szám számjegyeinek száma b alapú számrendszerben eggyel több, mint a szám b alapú logaritmusának az alsó egészrésze, azaz ha a szám számjegyei c_n, c_{n-1}, ..., c_1, c_0, akkor a jegyek száma $\mathbf{n + 1 = \lfloor \log_b x \rfloor + 1}$.

Logikai absztrakt adattípus: Egy olyan halmazt ad meg, amelynek két eleme van, az igaz és a hamis.
- jelölése: L = {igaz, hamis}
- megadása: T = (L, M), ahol L a logikai adatok halmaza és M a rajtuk végezhető műveletek halmaza
- műveletei: unáris és bináris műveletek:

	Hamis	Identikus	Negáció, tagadás NEM, NOT	Igaz
x	h	x	$\bar{x}$	i
h	h	h	i	i
i	h	i	h	i

x	y	Diszjunkció (VAGY, OR) $x \vee y$	Konjunkció (ÉS, AND) $x \wedge y$	Antivalencia (KIZÁRÓ VAGY, XOR) $x \oplus y$	Ekvivalencia $x \leftrightarrow y$	Implikáció $x \rightarrow y$	Peirce nyíl (NEM VAGY, NOR) $x \downarrow y$	Scheffer vonás (NEM ÉS, NAND) $x \mid y$
h	h	h	h	h	i	i	i	i
h	i	i	h	i	h	i	h	i
i	h	i	h	i	h	h	h	i
i	i	i	i	h	i	i	h	h

- elemi konjunkció: változók vagy tagadottjainak a konjunkciója, melyben a változók legfeljebb egyszer fordulnak elő.
- diszjunktív normálforma (DNF): elemi konjunkciók diszjunkciója
- realizációja: a logikai adat (változó) realizálása bitekkel történhet a hamis = 0, igaz = 1 módon

Halmaz absztrakt adattípus: adott tulajdonságú elemek, objektumok összessége, együttese, ahol minden elemről, objektumról egyértelműen el kell tudni dönteni, hogy a kijelölt halmaznak eleme-e, vagy sem.
- jelölése: latin nagy betűkkel
- megadása: A = {x | P(x) igaz}, azaz A halmaz olyan x elemekből áll, amelyekre a P állítás igaz, ahol P-ben valamely tulajdonság van megfogalmazva, VAGY az elemek egyszerű felsorolásával: A = $\{a_0, a_1, ..., a_n\}$
- műveletei:
    - unáris: komplemens B = $\bar{A}$
    - bináris: unió A ∪ B, metszet A ∩ B, különbség A \ B, szimmetria A ∆ B
- realizációja: Venn-diagram

Karakter absztrakt adattípus: szöveges információ kezelését, megjelenítését teszi lehetővé, ahol a szövegeinket elemi egységekből építjük fel.
- karakter: a tágabb értelemben szövegesen lejegyzett adat legkisebb, elemi egysége, egy tovább már nem bontható szimbólum.
- jelölése: X karakterek halmaza
- megadása: T = (X , M), ahol az M az X -en végezhető műveletek halmaza
- műveletei: ▹, ◃ (rendezettség esetén "hátrébb" illetve "előrébb" áll-e az adott karakter az ábécében)
- realizációja: ASCII kódtáblázat, Unicode, UTF kódolási formák

Sztring absztrakt adattípus:
- jelölése: rögzített X karakterek halmaza esetén S sztring absztrakt adattípus
- megadása: $S = X^*$, ahol a halmaz elemeit sztringnek nevezzük (ε neve üres sztring)
- attribútuma: Hossz['S']: egy szám, amely megadja, hogy a sztring hány karakterből áll
- műveletei: bináris: konkatenáció: a benne szereplő sztringek egymáshoz fűzése, ahol az első végéhez hézagmentesen hozzáillesztjük a másodikat, így formálva az eredménysztringet (jele: •)
- realizációja: X = ASCII szerint

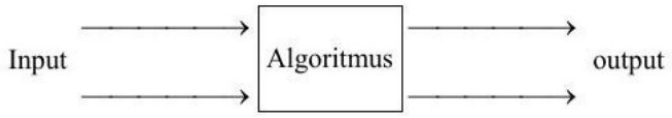
Sorozat absztrakt adattípus:
- jelölése: az elemekre az index megadásával (1,2,3, ...) hivatkozunk (mátrixnál két index)
- megadása: $T = (A^n , M)$, ahol A absztrakt adattípus és $n \in \mathbb{N}$ rögzített szám
- attribútuma: Hossz[x]: az n szám
- műveletei:
    - Köv(.): minden elemnek (az utolsót kivéve) van közvetlen rákövetkezője
    - Elő(.): minden elemnek (az elsőt kivéve) van közvetlen megelőzője
- realizációja: tömb
    - vektor: tömb elemeinek megkülönböztetésére elegendő egyetlen indexet használni
    - mátrix (összetett): olyan vektor, amelynek az elemei is vektorok

Rekord absztrakt adattípus: olyan összetett adattípus, amelyben rögzített sorrendben általában különböző típusú adatelemek követik egymást
- jelölése: az egyes adatelemek a rekord mezői, a mező nevének leírásával és mögötte szögletes zárójelben a rekord nevének megadásával hivatkozunk a rekord mezőire
- megadása: $T = (A_1 \times A_2 \times ... \times A_k , M)$, ahol $A_1, A_2, \ldots, A_k$ absztrakt adattípusok és $k \in \mathbb{N}$ rögzített szám
- műveletei: műveletek halmaza sokféle lehet az egyes mezőkkel végezhető műveletek sokfélesége miatt
- realizációja: az adatelemek memóriában történő egymást követő elhelyezésével

Mutató absztrakt adattípus: A mutató absztrakt adattípus valamely adatra mutat a memóriában azáltal, hogy az adat memóriabeli címét (byte-sorszám) tárolja. Adattal együtt van értelme használni. Ha nem mutat adatra, akkor NIL mutatónak nevezzük. A mutatóhoz hozzátartozik az a típus is, amilyen típusú adatra mutat.

2. Az algoritmus. Iteratív és rekurzív algoritmus. A számítógépes memória. Adat és program. Verem és procedúra. Az algoritmus lejegyzése. A folyamatábra és a pszeudokód. Elemi algoritmusok.

Algoritmus:
1. Heurisztikus: Meghatározott számítási eljárás, a számítási probléma megoldási eszköze.
2. Számítási folyamat: Az algoritmus valamely előre meghatározott adathalmaz valamely tetszőleges kiinduló eleméből kezdve az ezen elem által meghatározott eredmény elérésére törekszik.



Iteratív algoritmus: hasonló, vagy azonos műveletek sorozatát ismétlik
Rekurzív algoritmus: a probléma mérete redukálható kisebb méretekre és a kisebb méretű feladat megoldása után visszatérhetünk a nagyobb méretűnek a megoldásához

Számítógépes memória: Azonos méretű tárolórekeszek összessége, ahol minden rekesznek van egy sorszáma (címe), amivel azonosítani tudjuk. A számítógépes memória byte-okból épül fel, melyek a memóriában lineárisan helyezkednek el, azaz egymást követik.
- byte: adott sorrendben elhelyezkedő, egymást követő, egy egységbe összefogott bitnyolcas
- típusai: ROM (Read Only Memory): csak olvasható, RAM (Random Access Memory): írható és olvasható

Adat: minden olyan információ, amelynek segítségével leírunk egy jelenséget, tanulmányunk tárgyát, vagy annak egy részét (általános megfogalmazásban: valamely előre rögzített halmaznak az eleme).
Program: a számításaink, adatokon végzett tevékenységeink elvégzéséhez szükséges gépi utasítások, parancsok rögzített sorozata
- szegmens (=terület): a program fő részei: a kódterület, az adatterület és a veremterület
- felépítése: főprogram, eljárások/procedúrák
Procedúra (eljárás): a logikailag önálló, zárt programegység, amely egy specializált részfeladatot old meg
- a többi eljárással és a főprogrammal a softvare interface-en keresztül tartja a kapcsolatot
- eljáráshívás: egy aktivizáló utasítás hatására kezd el működni, addig, amíg egy befejező utasításhoz nem ér, ekkor a hívás helyét követő helyre visszaadja a vezérlést (ez a mechanizmus a verem által valósul meg)
Verem: a memóriának a programfutás idejére kiválasztott része
- átmenetileg tárolunk fontos információkat, vagy átmeneti időre segédtárolóként használjuk
- ha egy procedúra aktivizál egy másik procedúrát, akkor a verembe az új aktivizálási információk is bekerülnek, a verem mélyül, a veremmélység szintszáma eggyel nő
- visszatéréskor a veremből az aktivizálási információk törlődnek
Algoritmus lejegyzése:
- szövegesen
- pszeudokóddal: szöveges leírás az algoritmusban megfogalmazott gondolatokat tükrözve, bármely programozási nyelv szintaktikai szabályai nélkül
- folyamatábrával: grafikus szimbólumok felhasználásával szemlélteti az algoritmust

Pszeudokód:
- általános konvenciók: blokkszerkezet, értékadás jele: ←, megjegyzés // után, lokális változók használata, tömbelemekre való hivatkozás indexeléssel, objektumok használata, mutatók használata tömbök és objektumok megadására, input paraméterek érték szerinti átadása, visszatérési érték “return” utasításban
- utasítások:

Utasítás kihagyásos elágazás
IF feltétel
THEN blokk

Kétirányú elágazás
IF feltétel
THEN blokk
ELSE blokk

Többirányú elágazás
CASE kifejezés
feltétel1: blokk
...
feltétel n: blokk

CASE kifejezés
feltétel1: blokk
...
feltétel n:blokk
ELSE blokk

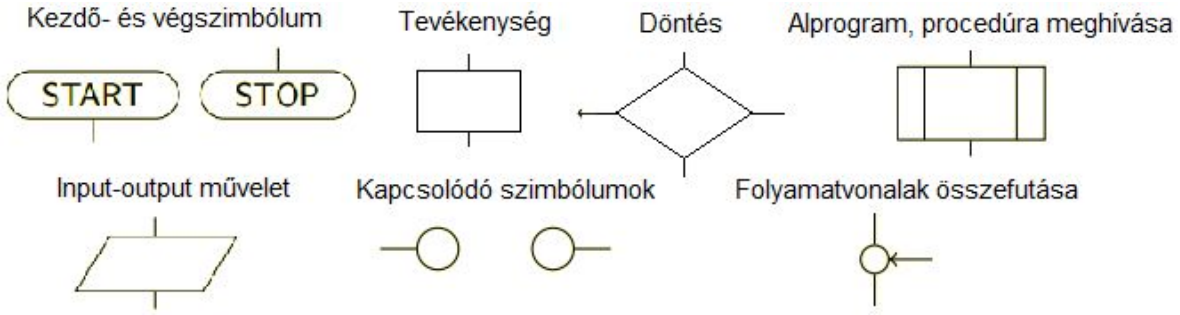
- ciklusok:

Előrehaladó leszámláló, elöltesztelő ciklus	Visszafeléhaladó leszámláló, elöltesztelő ciklus	Elöltesztelő iteratív ciklus	Hátultesztelő iteratív ciklus
FOR ciklusváltozó←kezdőérték **TO** végérték **DO** blokk	**FOR** cv←ké **DOWNTO** vé **DO** blokk	**WHILE** feltétel **DO** blokk	**REPEAT** blokk **UNTIL** feltétel

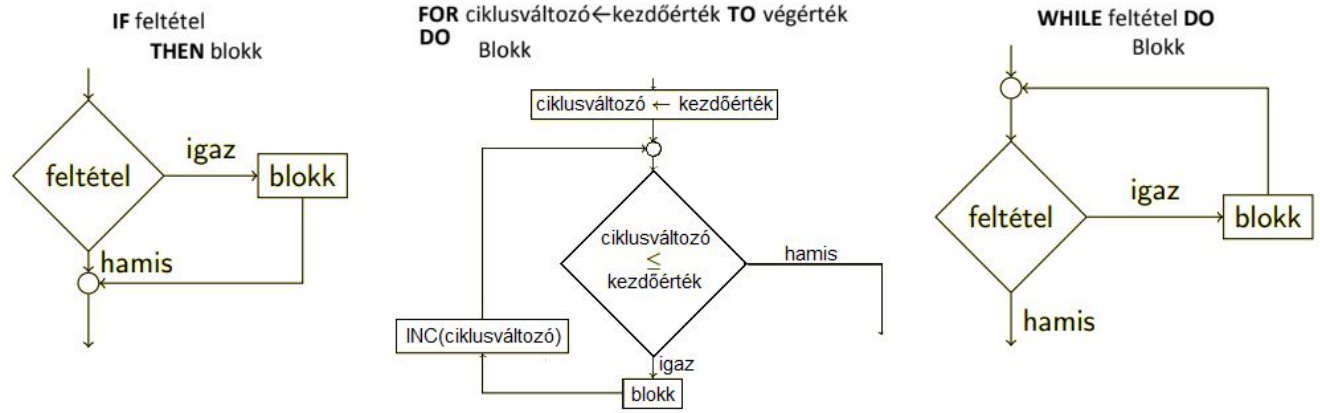
- egyéb utasítás: INC(változó): szám típusú változó értékének megnövelése eggyel, DEC(változó): szám típusú változó értékének csökkentése eggyel, RETURN(paraméterlista): kilépés a procedúrából

Folyamatábra: vízszintes vagy függőleges folyamatvonalak révén egymáshoz kapcsolható szimbólumokat használunk, melyek összefutását kis körökkel jelöljük

- szimbólumok:



- utasítások:



Elemi algoritmusok:

1. **Sorozatszámítás**: sorozathoz rendel egy értéket, amelyet a sorozaton végrehajtott bináris művelet ad

```
1. Sorozatszamitas (X,s)
2. // Input: X sorozat x_i ϵ A
3. // Output: s ϵ A (kapott egyetlen érték)
4. // f:AxA→A (bináris – kétváltozós művelet)
5. // Null az f nulleleme, Null ϵ A, f(Null, a) = a, ∀aϵA
6. s ← Null                                  // Inicializálás
7. FOR i←1 TO Hossz[X] DO
8.         s←f(s, x_i)
9. RETURN (s)
```

2. **Másolás**: a bemeneti sorozat minden elemének elkészítjük egy transzformáltját, ez a sorozat az eredmény, ahol a transzformáció: z = f (x)

```
1. Másolás (X, Y)
2. // Input: X sorozat, x_i ϵ A
3. // Output: Y sorozat, y_i ϵ B
4. // f : A → B
5. FOR i ← 1 TO Hossz[X] DO
6.         y_i ← f (x_i)
7. RETURN(Y)
```

3. **Eldöntés**: a bemeneti sorozathoz egy logikai értéket kell rendelni

Van-e adott T tulajdonságú elem a sorozatban?

```
1. Eldöntés1 (X,Van)
2. // Input: X – sorozat, x_i ϵ A
3. // Output: Van ϵ L = {hamis, igaz}
4. // T: A → L
5. i←1
6. WHILE i ≤ Hossz[X] ÉS NOT T(x_i) DO
7.         INC(i)
8. Van ← (i ≤ Hossz[X])
9. RETURN (Van)
```

Minden eleme a sorozatnak T tulajdonságú?

```
1. Eldöntés2 (X,Mind)
2. // Input: X sorozat, x_i ϵ A
3. // Output: Mind ϵ L = {igaz, hamis}
4. // T: A→L
5. i←1
6. WHILE i ≤ Hossz[X] AND T(x_i) DO
7.         INC(i)
8. Mind ←(i > Hossz[X])
9. RETURN(Mind)
```

4. **Kiválasztás**: meg kell adni a sorozat egy adott T tulajdonságú elemét a sorszáma révén (feltesszük, hogy ilyen elem van a sorozatban)

```
1. Kiválasztás(X, Sorszam)
2. // Input: X – sorozat, x_i ϵ A
3. // Output: Sorszam ϵ Z – T tulajdonságú elem indexe.
4. // T: A→L
5. i ←1
6. WHILE NOT T(x_i) DO
7.         INC(i)
8. Sorszam ← i
9. RETURN (sorszam)
```

5. **Megszámolás**: T tulajdonságú elemek számát kell meghatároznunk

```
1. Megszámolás(X, db)
2. // Input X – sorozat, x_i ϵ A
3. // Output db ϵ Z – T tulajdonságú elemek darabszáma
4. // T: A →L
5. db ← 0
6. FOR i ←1 TO Hossz[X] DO
7.         IF T(x_i)
8.                 THEN INC(db)
9. RETURN(db)
```

6. **Maximum kiválasztás**: egy teljesen rendezett halmazból származó sorozat maximális elemének indexét kell meghatározni

```
1. Maximum_kiválasztás(X, Sorszam)
2. // Input: X – Sorozat, x_i ϵ A, A teljesen rendezett halmaz
3. // Output: Sorszam ϵ Z – a T tulajdonságú elem indexe
4. Sorszam ← 1
5. FOR i ← 2 TO Hossz[X] DO
6.         IF x_Sorszam < x_i
7.                 THEN Sorszam ← i
8. RETURN (Sorszam)
```

7. **Kiválogatás**: a bemenő sorozat adott T tulajdonságú elemei sorszámának kigyűjtése

```
1. Kiválogatás (X, Y)
2. // Input: X – sorozat, x_i ϵ A
3. // Output: Y – sorozat, y_i ϵ A
4. // T: A → L
5. Hossz[Y] ← 0
6. FOR i ← 1 TO Hossz[X] DO
7.         If T(x_i)
8.                 THEN INC(Hossz[Y])
9.                         y_Hossz[Y] ← i
10. RETURN (Y)
```

8. **Lineáris keresés**: eldöntés és kiválasztás
Adott T tulajdonságú elemet kell megtalálni az input sorozatban ha benne van, egyébként “nincs”

```
1. Keresés(X, Van, Sorszam)
2. // Input: X – sorozat, x_i ϵ A
3. // Output: Van ϵ L
4. // Sorszam ϵ Z – a T tulajdonságú elem indexe
5. // T: A →L
6. i←1
7. WHILE ( i ≤ hossz[X] ÉS NOT T(x_i) ) DO
8.         INC(i)
9. Van ←(i ≤ Hossz[X])
10. IF Van
11.         THEN Sorszam ← i
12.                 RETURN (Van, Sorszam)
13.         ELSE    RETURN (Van)
```

9. **Szétválogatás**: T tulajdonságú és nem T tulajdonságú elemek kiválogatása

```
1. Szétválogatás (X, X, Z)
2. // Input: X – sorozat, x_i ϵ A
3. // Output:        Y – sorozat, y_i ϵ A, T(y_i) igaz
4. //                Z – sorozat, z_i ϵ A, T(z_i) hamis
5. // T: A →L
6. Hossz[Y] ← 0
7. Hossz[Z] ← 0
8. FOR i ← 1 TO Hossz[X] DO
9.         IF T(x_i)
10.                THEN INC (Hossz[Y])
11.                        y_Hossz[Y] ← x_i
12.                ELSE INC (Hossz[Z])
13.                        z_Hossz[Z] ← x_i
14. RETURN(Y, Z)
```

10. **Egyesítés**: két ismétlés nélküli bemeneti sorozat azon elemeinek kiválogatása, amelyet legalább az egyik tartalmaz úgy, hogy a kimenet is ismétlés nélküli

```
1. Egyesítés (X, Y, Z)
2. // Input:        X – sorozat, x_i ϵ A,  különböző
3. //               Y – sorozat, y_i ϵ A, különböző
4. // Output:      Z – sorozatr, z_i ϵ A, különböző
5. Hossz[Z] ← Hossz[X]
6. FOR i ← 1 TO Hossz[X] DO
7.         z_i ←x_i
8. FOR j ← 1 TO Hossz[Y] DO
9.         i ← 1
10.        WHILE  i ≤ Hossz[X] ÉS x_i ≠ y_j DO
11.                INC(i)
12.        IF i > Hossz[X]
13.                THEN    INC(Hossz [Z])
14.                        z_Hossz[Z] ← yi
15. RETURN(Z)
```

11. **Metszet**: két ismétlés nélküli bemeneti sorozat közös elemeinek kiválogatása úgy, hogy a kimenet is ismétlés nélküli legyen

```
1. Metszet (X, Y, Z)
2. // Input:        X – sorozat, x_i ϵ A, különböző
3. //               Y – sorozat, y_i ϵ A, különböző
4. // Output:      Z – sorozat, z_i ϵ A, különböző
5. Hossz[Z] ← 0
6. FOR i ← 1 TO Hossz[X] DO
7.         j ← 1
8.         WHILE  j ≤ Hossz[Y] ÉS x_i ≠ y_j DO
9.                 INC(j)
10.        IF j ≤  Hossz[Y]
11.                THEN    INC(Hossz[Z])
12.                        z_Hossz[Z] ← x_i
13. RETURN (Z)
```

12. **Összefésülés**: két szigorúan monoton növekvő sorozatból egy kimenő szigorúan monoton növekvő sorozatot készíteni úgy, hogy az azonos elemekből csak egy szerepeljen a kimeneti sorozatban

```
1. Összefésülés (X, Y, Z)
2. // Input:        X – sorozat, x_i ϵ A különböző, x_i < x_i+1
3. //              Y – sorozat, y_i ϵ A, különböző y_i < y_i+1
4. // Output:     Z – sorozat, z_i ϵ A, különböző z_i < z_i+1
5. Hossz[Z] ← 0
6. i ← 1
7. j ← 1
8. WHILE i ≤ Hossz[X] ÉS j ≤ Hossz[Y] DO
9.         INC(Hossz[Z])
10.        CASE
11.                x_i < y_j :  z_Hossz[Z] ← x_i
12.                             INC(i)
13.                x_i = y_j :  z_Hossz[Z] ← x_i
14.                             INC(i)
15.                             INC(j)
16.                x_i > y_j :  z_Hossz[Z] ← y_j
17.                             INC(j)
18. WHILE i ≤ Hossz[X] DO
19.        INC(Hossz[Z])
20.        z_Hossz[Z] ← x_i
21.        INC(i)
22. WHILE j ≤ Hossz[Y] DO
23.        INC(Hossz[Z])
24.        z_Hossz[Z] ← y_j
25.        INC(j)
26. RETURN(Z)
```

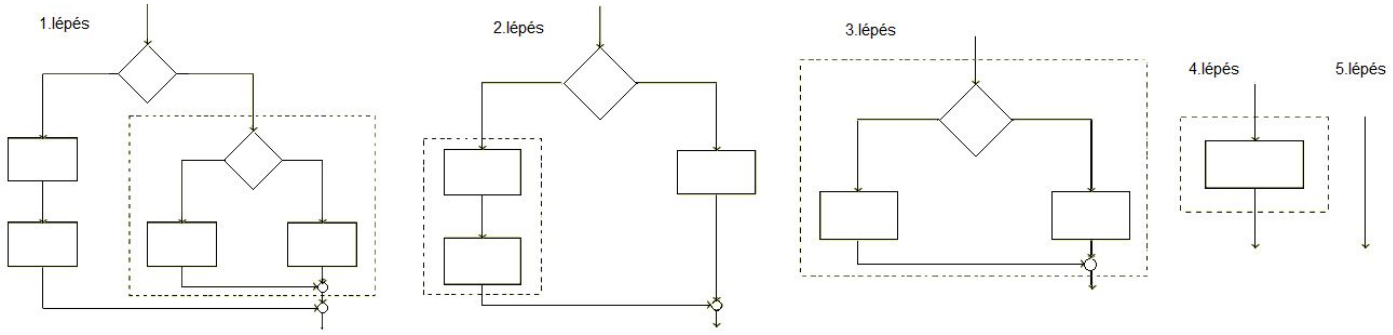
3. Strukturált programozás. Programgráf, valódi program, vezérlőgráf lebontása, strukturált program és annak formulája. Strukturált programgráf kialakítása, struktogram. Ciklikus bonyolultság és egyéb bonyolultsági tételek.

Strukturált programozás: olyan programozási elvek összessége, amelyek segítenek megteremteni, hogy a program szövegszerkezete tükrözze a program végrehajtása során követett vezérlési folyamatot
- Edsger W. Dijkstra (1968)
- a műveletekre összpontosítunk, az adatok szerepe, szerkezete másodlagos
- a programokat három alapelemből építi fel: szekvencia, ciklus, elágazás

Programgráf: A programgráf, vagy vezérlési gráf olyan összefüggő irányított gráf, amely vonalakból, szekvenciákból, elágazásokból és gyűjtő szimbólumokból épül fel. A vonalak a gráf éleit adják, a többiek pedig a gráf csomópontjait. A predikátumok az adatmezőt osztják fel két diszjunkt részre.

Valódi program: Egy programot valódi programnak nevezünk, ha:
1. programgráfja véges számú nem zérus bemenő éllel és kimenő éllel rendelkezik
2. programgráfjának csomópontjai predikátumok, függvények és gyűjtők
3. programgráfjának minden csomópontján keresztül vezet legalább egy útvonal, amely egy bemenő éllel és egy kimenő éllel rendelkezik

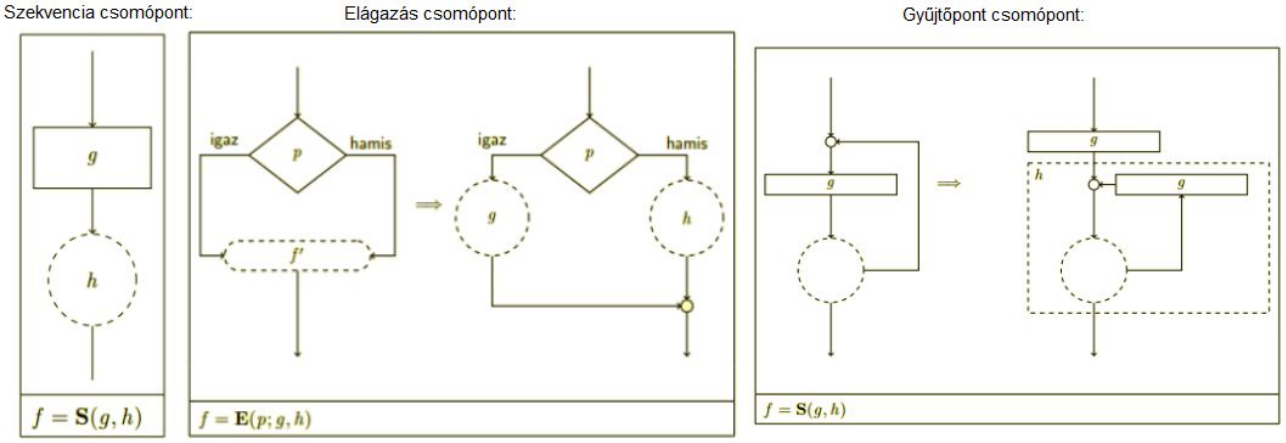
Vezérlőgráf lebontása: A vezérlőgráf lebontásának nevezzük azt az eljárást, melynek során a strukturált alapszerkezetek valamelyikét egy függvény csomóponttal helyettesítjük, és ezt mindaddig folytatjuk, amíg ez lehetséges. Az egyetlen csomópontból álló gráfot egyetlen éllel helyettesítjük (üres program).



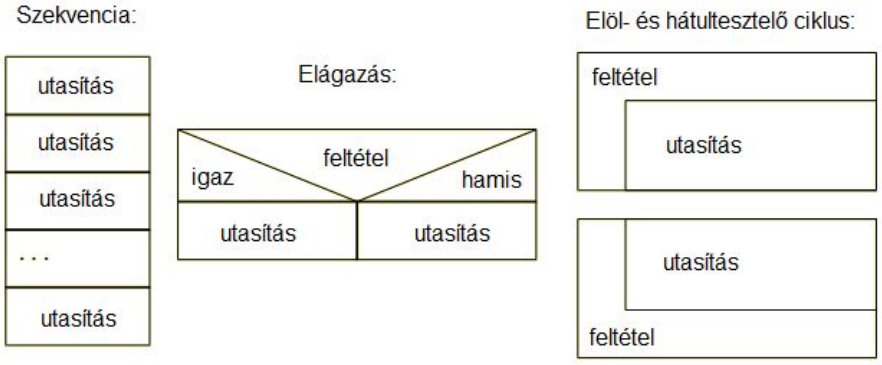
Strukturált program: Strukturált programnak nevezzük azt a programot, amelynek vezérlőgráfja lebontható az önmagában álló irányított élre.

Strukturált program formulája: Egy f program akkor és csak akkor strukturált program, ha formulája felírható az S(g) vagy a C(p; g) vagy az E(p; g, h) alakban, ahol p predikátum, g és h pedig strukturált programok.

Strukturált programgráf kialakítása:
- a program tekinthető függvényként → felírható y = f (x) formulával → megadható az S (szekvencia), C (csomópont), és E vagy I (elágazás) programgráf formulákkal → ezután lépésenként redukáljuk a programgráfot, amíg strukturált nem lesz



Struktogram: folyamatábra szabadosságának korlátozott verziója a folyamatvonalak kiiktatásával
- Isaak Nassi és Ben Shneidermann (1972)
- csak kvázi-strukturált szimbólumokat tartalmaz → kvázi-strukturált programok szerkezete ábrázolható
- maga a program egyetlen partícionált (felosztott) téglalap
- utasításai:



Programgráf ciklikus bonyolultsága: A P programgráf ciklikus bonyolultságának nevezzük az m(P) = e − c mérőszámot, ahol e a programgráf éleinek a száma, c pedig a programgráf csúcsainak a száma.

Programgráf ciklikus bonyolultságának tétele: A P programgráf ciklikus bonyolultsága kiszámítható a következő módon is: m(P) = p + 1, ahol p a programgráf predikátumainak a száma.

Nem strukturált program ciklikus bonyolultságának tétele: A nem struktúrált P programgráf ciklikus bonyolultsága legalább három, azaz m(P) ≥ 3, ha P nem struktúrált.

Programgráf lényeges bonyolultsága: Legyen k a P programgráf azon részgráfjainak a száma, amelyek az E, C, I formulák valamelyikével felírhatók. Ekkor P lényeges bonyolultsága az M(P) = m(P) − k szám.

Strukturált program lényeges bonyolultságának tétele: A strukturált P programgráf lényeges bonyolultsága 1, azaz M (P) = 1, ha P strukturált.

4. Számelméleti algoritmusok. Legnagyobb közös osztó, euklideszi és kibővített euklideszi algoritmus, lineáris kongruencia egyenletek. Multiplikatív inverz, moduláris hatványozás, Fermat prímteszt. RSA.

Oszthatóság: Azt mondjuk, hogy a d egész szám osztja az a egész számot, ha az osztásnak zérus a maradéka, azaz, ha létezik olyan k egész szám, hogy **a = k · d**. Jelölésben: d|a. Ekkor a d számot az a osztójának nevezzük, az a szám pedig a d többszöröse.
Prímszám: Az az 1-nél nagyobb egész szám, amelynek csak az 1 és saját maga az osztója.
A maradékos osztás tétele: Ha a egész szám, n pedig pozitív egész szám, akkor egyértelműen létezik olyan q és r egész szám, hogy **a = q · n + r**, ahol 0 ≤ r < n. Ekkor a q szám neve hányados, az r neve maradék. A hányados: **q = ⌊a/n⌋**, a maradék: **r = a − q · n = a mod n**.
Közös osztó: Azt mondjuk, hogy a d egész szám az a és b egészek közös osztója, ha d mindkét számot osztja (azaz d|a és d|b).
Tulajdonságai:
- |d| ≤ |a|, vagy a = 0.
- Ha d|a és a|d, akkor d = ±a.
- A közös osztó osztója az a és b szám minden lineáris kombinációjának is, azaz ∀s ∈ L(a, b)-re d|s.

Lineáris kombináció: Az s egész számot az a és b egészek (egész) lineáris kombinációjának nevezzük, ha létezik olyan x és y egész szám, hogy s = x · a + y · b. Az x és y számokat a lineáris kombináció együtthatóinak nevezzük. Az a és b számok összes lineáris kombinációjának halmazát L(a, b)-vel jelöljük.

Legnagyobb közös osztó: Az alább megadott d^* egész számot az a és b egész számok legnagyobb közös osztójának nevezzük. Jele: lnko(a, b).

$$d^* = \text{lnko}(a,b) = \begin{cases} 0 & \text{ha } a = 0 \text{ és } b = 0 \\ \max\limits_{\substack{d|a \\ d|b}} d & \text{egyébként.} \end{cases}$$

Tulajdonságai:
- 1 ≤ d^* ≤ min{|a|, |b|}
- lnko(a, b) = lnko(b, a) = lnko(−a, b) = lnko(|a|, |b|)
- lnko(a, 0) = |a|
- lnko(a, k · a) = |a|, k ∈ ℤ
- Ha d közös osztó és d^* ≠ 0, akkor d ≤ d^*
- A legnagyobb közös osztó minden lineáris kombinációnak osztója, azaz ∀s ∈ L(a, b)-re d^*|s

Relatív prímek: Az a és b egész számokat relatív prímeknek nevezzük, ha lnko(a, b) = 1.

Euklideszi algoritmus:

```
1 Euklidesz (a, b, d*)
2 // Input paraméter : a ∈ Z, a ≥ 0
3 //                   b ∈ Z, b ≥ 0
4 // Output paraméter: d* ∈ Z, d* ≥ 0
5 WHILE b ≠ 0 DO
6       r ← a mod b
7       a ← b
8       b ← r
9       d* ← a
10 RETURN (d*)
```

Kibővített Euklideszi algoritmus:

1	Kibővített_Euklidesz (a, b, d*, x*, y*)
2	// Input paraméterek : a,b∈Z, a,b≥0
3	// Output paraméterek: d*, x*, y*∈Z, d*≥0
4	$x_0 \leftarrow 1, x_1 \leftarrow 0, y_0 \leftarrow 0, y_1 \leftarrow 1, s \leftarrow 1$
5	**WHILE** $b \neq 0$
6	$r \leftarrow a \bmod b, q \leftarrow a \operatorname{div} b$
7	$a \leftarrow b, b \leftarrow r$
8	$x \leftarrow x_1, y \leftarrow y_1$
9	$x_1 \leftarrow q \cdot x_1 + x_0; y_1 \leftarrow q \cdot y_1 + y_0$
10	$x_0 \leftarrow x, y_0 \leftarrow y$
11	$s \leftarrow -s$
12	$x \leftarrow s \cdot x_0, y \leftarrow -y_0$
13	$(d^*, x^*, y^*) \leftarrow (a, x, y)$
14	**RETURN** (d^*, x^*, y^*)

Kongruencia: Az a és b egész számokat kongruensnek mondjuk az n modulus szerint, ha az n szerinti osztás utáni maradékaik megegyeznek, vagy ami ugyanaz: ha n|(a − b). Jelölése: a ≡ b mod n.
A kongruenciákon végezhető műveletek tétele: Legyen a ≡ b mod n és c ≡ d mod n. Ekkor:
- a ± c ≡ b ± d mod n
- a · c ≡ b · d mod n
- a/k ≡ b/k mod n ha k|a , k|b és lnko (k, n) = 1
- a ≡ b mod m, ha m|n

A lineáris kongruencia egyenlet: Az **a · x ≡ b mod n** egyenletet, ahol a, b ∈ $\mathbb{Z}$, n ∈ $\mathbb{Z}^+$ és x ∈ $\mathbb{Z}$ az ismeretlen, lineáris kongruencia egyenletnek nevezzük.

Multiplikatív inverz: Legyen a lineáris kongruencia egyenlet **ax ≡ 1 mod n**, ahol a ∈ $\mathbb{Z}$, n ∈ $\mathbb{Z}^+$, lnko(a, n) = 1 alakú (azaz a és n legyenek relatív prímek). Az egyenlet egyetlen alapmegoldását az a szám n szerinti multiplikatív inverzének nevezzük. Jelölése: $x = a^{-1} \bmod n$.

Fermat-tétel: Ha p prím, akkor $a^{p-1} \equiv 1 \bmod p$, ahol a = 1, 2, ... , p−1.

Moduláris hatványozás:

```
1  Moduláris_hatványozó (a, b, n, c)
2  // Input paraméterek: a, b, n ∈ Z, a, b, n > 0
3  // Output paraméter: c ∈ Z, c ≥ 0
4  p ← 0
5  c ← 1
6  FOR i ← k DOWNTO 0 DO
7        p ← 2p
8        c ← c^2 mod n
9        IF b_i = 1
10             THEN p ← p + 1
11                  c ← (c · a) mod n
12 RETURN (c)
```

Fermat prímteszt:

```
1  Fermat_teszt (n, p)
2  // Input paraméter: n ∈ Z, n > 1
3  // Output paraméter: p logikai érték
4  //                  igaz – lehet prím
5  //                  hamis – nem prím
6  Moduláris_hatványozó (2, n − 1, n, c)
7  p ← (c = 1)
8  RETURN (p)
```

RSA: nyilvános kulcsú titkosítás algoritmusa
- Rivest – Shamir - Adleman
- a szöveg titkosítása a C = P(M) = M^e mod n alapján történik
- dekódolása az M = S(C) = C^d mod n alapján

```
1  RSA_kulcsok_meghatározása (p, q, e, P, S)
2  // Input paraméterek: p, q, e
3  // Output paraméterek: P, S
4  IF p vagy q nem prím vagy e<3 vagy e páros
5       THEN RETURN („Nincs kulcs”)
6  n ← p · q
7  f ← (p − 1) · (q − 1)
8  IF lnko (e, f) ≠ 1
9       THEN RETURN („Nincs kulcs”)
10 d ← e^-1 mod f
11 RETURN (P = (e, n), S = (d, n))
```

5. Rendezések: Beszúró rendezés. Az oszd meg és uralkodj elv. Összefésülő rendezés. Gyors rendezés. Buborék rendezés. Shell rendezés. Minimum kiválasztásos rendezés. Négyzetes rendezés. Lineáris idejű rendezések: leszámláló rendezés, számjegyes rendezés. Időelemzéseik. Az összehasonlító rendezések időtétele.

Beszúró rendezés: A sorozat második elemétől kezdve egyenként a kulcsokat a sorozat eleje felé haladva a megfelelő helyre mozgatjuk összehasonlítások révén.
Időigénye: $T(n) = \Theta(n^2)$

```
1  BESZÚRÓ_RENDEZÉS ( A )
2  // Input paraméter: A - a rendezendő tömb
3  // Output paraméter: A - a rendezett tömb
4  //
5  FOR j ← 2 TO hossz[A] DO
6      kulcs ← A_j
7      // Beszúrás az A_{1...j-1} rendezett sorozatba
8      i ← j − 1
9      WHILE i > 0 és A_i > kulcs DO
10         A_{i+1} ← A_i
11         DEC(i)
12     A_{i+1} ← kulcs
13 RETURN (A)
```

Oszd meg és uralkodj elv: Egy algoritmus tervezési stratégia, ahol a problémát olyan kisebb méretű, azonos részproblémákra osztjuk fel, amelyek rekurzívan megoldhatók, ezután egyesítjük a megoldásokat.

Összefésülő rendezés: oszd meg és uralkodj elven működik:
- felosztás: a tömböt két n/2 elemű részre osztjuk
- uralkodás: rekurzív összefésüléses módon mindkettőt rendezzük
- egyesítés: a két részsorozatot összefésüljük

Időigénye: $T(n) = \Theta(n \cdot \log n)$

```
1  ÖSSZEFÉSÜLŐ_RENDEZÉS ( A, p, r )
2  // Input paraméter: A - a tömb, melynek egy részét rendezeni kell
3  //                  p - a rendezendő rész kezdőindexe
4  //                  r - a rendezendő rész végindexe
5  // Output paraméter: A - a rendezett résszel rendelkező tömb
6  //
7  IF  p < r
8      THEN q ← ⌊(p+r)/2⌋
9           ÖSSZEFÉSÜLŐ_RENDEZÉS ( A, p, q )
10          ÖSSZEFÉSÜLŐ_RENDEZÉS ( A, q + 1, r )
11          ÖSSZEFÉSÜL ( A, p, q, r )
12 RETURN (A)
```

Gyors rendezés: oszd meg és uralkodj elven működik:
- felosztás: az $A_{p..r}$ tömböt két nemüres $A_{p..q}$ és $A_{q+1..r}$ részre osztjuk úgy, hogy $A_{p..q}$ minden eleme ≤ legyen, mint $A_{q+1..r}$ bármely eleme
- uralkodás: az $A_{p..q}$ és $A_{q+1..r}$ résztömböket rekurzív gyorsrendezéssel rendezzük
- egyesítés: nincs rá szükség, mivel a tömb már rendezett

Időigénye: legjobb: $T(n) = \Theta(n^2)$, átlagos: $T(n) = \Theta(n \cdot \log n)$

```
1  GYORSRENDEZÉS(A,p,r )
2  // Input paraméter: A – a tömb, melynek egy részét rendezeni kell
3  //                  p - a rendezendő rész kezdőindexe
4  //                  r - a rendezendő rész végindexe
5  // Output paraméter: A - a rendezett résszel rendelkező tömb
6  //
7  IF   p < r
8       THEN FELOSZT ( A, p, r, A_p, q ) //Lásd 4.1.5.1 algoritmus
9            GYORSRENDEZÉS ( A, p, q )
10           GYORSRENDEZÉS ( A, q+1, r )
11 RETURN (A)
```

Buborék rendezés: Az egymás mellett álló elemeket hasonlítjuk össze, és szükség esetén sorrendjüket felcseréljük. Ezt mindaddig folytatjuk, míg szükség van cserére.
Időigénye: $T(n) = \Theta(n^2)$

```
1  BUBORÉKRENDEZÉS( A )
2  // Input paraméter: A - a rendezendő tömb
3  // Output paraméter: A - a rendezett tömb
4  //
5  j ← 2
6  REPEAT  nemvoltcsere ← igaz
7          FOR   i ← hossz[A] DOWNTO j  DO
8                IF   A_i < A_{i-1}
9                     THEN csere A_i ↔ A_{i-1}
10                         nemvoltcsere ← hamis
11         INC(j)
12 UNTIL   Nemvoltcsere
13 RETURN ( A )
```

Shell rendezés: Egymástól távol álló elemeket hasonlít és cserél fel, miközben a távolságot (itt növekménynek nevezik) fokozatosan csökkenti, míg az 1 nem lesz. Minden növekmény esetén beszúrásos rendezést végez az adott növekménynek megfelelő távolságra álló elemekre.
Időigénye: $T(n) = \Theta(n^{1,5})$

```
1  SHELL_RENDEZÉS( A )
2  // Input paraméter: A - a rendezendő tömb
3  // Output paraméter: A - a rendezett tömb
4  //
5  FOR   s ← 1 TO t DO
6        m ← h_s
8        FOR j ← m + 1 TO hossz[A] DO
9              i  ← j - m
10             k ← kulcs[A_j]
               r ← A_j
15             WHILE  i > 0 és k < A_j DO
16                    A_{i+m} ← A_i
17                    i ← i – m
18             A_{i+m} ← r
19 RETURN ( A )
```

Minimum kiválasztásos rendezés: Hossz[A] − 1-szer végigmegyünk a tömbön, minden alkalommal eggyel magasabb indexű elemtől indulunk. Megkeressük a minimális elemet, és azt az aktuális kezdőelemmel felcseréljük.
Időigénye: T(n) = Θ(n^2)

```
1  MINIMUM_KIVÁLASZTÁSOS_RENDEZÉS( A )
2  // Input paraméter: A - a rendezendő tömb
3  // Output paraméter: A - a rendezett tömb
4  //
5  FOR     i ← 1 TO hossz[A]-1 DO
6          // minimumkeresés
7          k ← i
8          x ← A_i
9          FOR     j ← i + 1 TO hossz[A] DO
10                 IF     A_j < x
11                        THEN  k ← j
12                              x ← A_j
13         // az i. elem és a minimum felcserélése
14         A_k ← A_i
15         A_i ← x
16 RETURN ( A )
```

Négyzetes rendezés: Felosztjuk az n elemű A tömböt √n számú alcsoportra. Mindegyikben √n elemet helyezünk el, majd mindegyikből kiemeljük a legkisebbet. A kiemeltekből egy főcsoportot képzünk. Kiválasztjuk a főcsoport legkisebb elemét és azt az eredménytömbbe írjuk, a főcsoportból pedig eltávolítjuk. Helyére abból az alcsoportból, ahonnan ő származott újabb legkisebbet emelünk be a főcsoportba. Az eljárást folytatjuk, míg az elemek el nem fogynak a főcsoportból.
Időigénye: T(n) = Θ(n · √n) = Θ($n^{1,5}$)

Lineáris idejű rendezők: nem használnak összehasonlítást
Leszámláló rendezés: A minden elemére meghatározza a nála kisebb elemek számát, ezáltal tudja az elemet a kimeneti tömb megfelelő helyére tenni.
Stabil eljárás: az azonos értékűek sorrendje megegyezik az eredetivel
Időigénye: T(n) = Θ(n)

```
1  LESZÁMLÁLÓ_RENDEZÉS ( A, k, B )
2  // Input paraméter: A - a rendezendő tömb
3  //                  k – kulcs felső korlát, pozitív egész
4  // Output paraméter: B - a rendezett tömb
5  //
6  FOR i ← 1 TO k DO
7      C_i ← 0
8  FOR j ← 1 TO hossz[A] DO
9      INC(C_Aj)
10     // C_i azt mutatja, hogy hány i értékű számunk van
11 FOR i ← 2 TO k DO
12     C_i ← C_i + C_i-1
13     // C_i most azt mutatja, hogy hány i-től nem nagyobb
       számunk van
14 FOR j ← hossz[A] DOWNTO 1 DO
15     B_C_Aj ← A_j
16     DEC(C_Aj)
17 RETURN (B)
```

Számjegyes rendezés: Azonos hosszúságú szavak, stringek rendezésére használhatjuk. Legyen d a szó hossza, k az egy karakteren, mezőben előforduló lehetséges jegyek, jelek száma, n pedig az adatok száma
Időigénye: T(n) = Θ(d · (n + k))

1	SZÁMJEGYES_RENDEZÉS (A)
2	// Input paraméter: A - a rendezendő tömb
4	// Output paraméter: A - a rendezett tömb
4	//
5	**FOR** $i \leftarrow d$ **DOWNTO** 1 **DO**
6	Stabil módszerrel rendezzük az A tömböt az i. számjegyre
7	**RETURN** (A)

Alsó korlát összehasonlító rendezésre: bármely n elemet rendező döntési fa magassága T(n) = Ω(n · log n)

6. Gráf algoritmusok. Szélességi keresés. Mélységi keresés. Topologikus rendezés. Optimum feladatok fákon (Minimális feszítőfák, a Kruskal és Prim algoritmus). Legrövidebb utak meghatározása (Dijkstra algoritmus, Bellman-Ford algoritmus. Floyd-Warshall algoritmus).

Gráf: Legyen V egy véges halmaz, E pedig V-beli rendezetlen elempárok véges rendszere. Ekkor a G = (V, E) párt gráfnak nevezzük. Továbbá n = |V | (csúcsok száma) és e = |E| (élek elemszáma).

Szélességi keresés: G éleit vizsgálja és rátalál minden s-ből elérhető csúcsra, majd kiszámítja az elérhető csúcsok legrövidebb (legkevesebb élből álló) távolságát s-től. Ezután létrehoz egy s gyökerű szélességi fát, amelyben az s-ből elérhető csúcsok vannak. A csúcsoknak színt tulajdonít (fehér, szürke, fekete). Kezdetben minden csúcs fehér, kivéve s-et, amely szürke. Szürke lesz egy csúcs, ha elértük, és fekete, ha megvizsgáltuk az összes belőle kiinduló élt. A szélességi fa kezdetben az s csúcsból áll (ez a gyökér). Ha egy fehér v csúcshoz értünk az u csúcsból, akkor azt felvesszük a fába (u, v) éllel, ahol u lesz a v szülője.

	SZÉLESSÉGI_KERESŐ(G, s) $O(V + E)$
1	**FOR** ∀ ∈V[G]\[s] csúcsra **DO**
2	$szin[u] \leftarrow$FEHÉR
3	$táv[u] \leftarrow \infty$
4	$\pi[u] \leftarrow$ NIL
5	$Szin[s] \leftarrow$ SZÜRKE
6	$táv[s] \leftarrow 0$
7	$\pi[s] \leftarrow$ NIL
8	$Q \leftarrow \{s\}$
9	**WHILE** $Q \neq \emptyset$ **DO**
10	$u \leftarrow fej[Q]$
11	**FOR** $\forall v \in szomszéd[u]$-ra **DO**
12	**IF** $szin[v] =$ FEHÉR
13	**THEN** $szin[v] \leftarrow$SZÜRKE
14	$táv[v] \leftarrow táv[u]+1$
15	$\pi[v] \leftarrow u$
16	SORBA(Q,v)
17	SORBÓL(Q)
18	$szin[u] \leftarrow$ FEKETE

Mélységi keresés: G éleit vizsgálja, mindig az utoljára elért, új kivezető élekkel rendelkező v csúcsból kivezető, még nem vizsgált éleket deríti fel. Az összes ilyen él megvizsgálása után visszalép és azon csúcs éleit vizsgálja, amelyből v-t elértük. Létrehoz egy mélységi erdőt, amely az előd részgráf fáiból áll. A csúcsoknak színt tulajdonít. Kezdetben minden csúcs fehér. Szürke lesz, amikor elértük, és fekete, amikor elhagytuk. Minden csúcshoz két időpontot rendel, az elérési d[v] és az elhagyási időpontot f[v].

	MÉLYSÉGI_KERESŐ(G) $\Theta(V+E)$
1	**FOR** $\forall u \in V[G]$ csúcsra **DO**
2	$szin[u] \leftarrow$ FEHÉR
3	$\pi[u] \leftarrow$ NIL
4	idő $\leftarrow 0$
5	**FOR** $\forall u \in V[G]$ csúcsra **DO**
6	**IF** $szin[u] =$FEHÉR
7	**THEN** MK_BEJÁR(u)

	MK_BEJÁR(u) $\Theta(V+E)$
1	$szin[u] \leftarrow$ SZÜRKE
2	$d[u] \leftarrow$ idő $\leftarrow$ idő$+1$
3	**FOR** $\forall v \in szomszéd[u]$ csúcsra **DO**
4	**IF** $szin[v] =$ FEHÉR
5	**THEN** $\pi[v] \leftarrow u$
6	MK_BEJÁR(v)
7	$szin[u] \leftarrow$ FEKETE
8	$f[u] \leftarrow$ idő $\leftarrow$ idő$+1$

Topologikus rendezés fákon: Egy irányított G = (V, E) gráf topologikus rendezése a csúcsainak sorba rendezése úgy, hogy ha G-ben szerepel az (u, v) él, akkor u előzze meg v-t a sorban. Az algoritmus egy irányított, körmentes gráf topologikus rendezését állítja elő.

	TOPOLOGIKUS_RENDEZÉS(G) $\Theta(V+E)$
1	MÉLYSÉGI_KERESŐ(G) hívása, minden csúcsra meghatározzuk az $f[u]$ elhagyási időt.
2	Az egyes csúcsok elhagyásakor szúrjuk be azokat egy láncolt lista elejére
3	**RETURN**(a csúcsok listája)

Minimális feszítőfa: Minimális feszítőfáról beszélünk, ha a fa súlya minimális az összes T feszítőfára nézve. Biztonságos él: Legyen A egy minimális feszítőfa egy része. A-ra nézve biztonságos egy él, ha A-hoz hozzávéve A továbbra is valamely minimális feszítőfa része marad.

	Minimális_Feszítő_Fa(G, w)
1	$A \leftarrow \emptyset$
2	**WHILE** A nem feszítőfa **DO**
3	keresünk egy biztonságos (u, v) élt az A-ra nézve
	$A \leftarrow A \cup \{(u, v)\}$
4	RETURN (A)

Kruskal algoritmus:

	MFF_KRUSKAL(G, w) O(E log E)
1	$A \leftarrow \emptyset$
2	**FOR** $\forall v \in V[G]$-re **DO**
3	HALMAZT_KÉSZÍT(v)
4	Rendezzük E éleit a súly szerint növekvő sorrendben
5	**FOR** $\forall (u, v) \in E$ élre az élek súly szerint növekvő sorrendjében **DO**
6	**IF** HALMAZT_KERES$(u) \neq$ HALMAZT_KERES(v)
7	**THEN** $A \leftarrow A \cup \{(u, v)\}$
8	EGYESÍT(u, v)
9	RETURN(A)

Prim algoritmus:

	MFF_PRIM(G, w) O(E log V)
1	$Q \leftarrow V[G]$
2	**FOR** $\forall v \in Q$-ra **DO**
3	$kulcs[v] \leftarrow \infty$
4	$kulcs[r] \leftarrow 0$
5	$\pi[r] \leftarrow$ NIL
6	**WHILE** $Q \neq \emptyset$ **DO**
7	$u \leftarrow$ KIVESZ_MIN(Q)
8	**FOR** $\forall v \in szomszéd[u]$-ra **DO**
9	**IF** $v \in Q$ és $w(u, v) \leq kulcs[v]$
10	**THEN** $\pi[v] \leftarrow u$
11	$kulcs[v] \leftarrow w(u, v)$
12	RETURN(π)

Legrövidebb utak meghatározása: A v csúcs legrövidebb út távolsága s-től legyen δ(s, v), amelyet az s-ből v-be vezető egyes utak élszámainak minimumaként definiálunk, ha van ilyen út, és ∞, ha nem vezet út s-ből v-be. Ekkor egy δ(s, v) hosszúságú s-ből v-be vezető utat s és v közötti legrövidebb útnak nevezzük.

Dijkstra algoritmus: Dijkstra algoritmusa az adott kezdőcsúcsból induló legrövidebb utak problémáját egy élsúlyozott, irányított G = (V, E) gráfban abban az esetben oldja meg, ha egyik élnek sem negatív a súlya.

	Dijkstra(G, s) O(V^2)
1	EGY_FORRÁS_KEZDŐÉRTÉK(G, s)
2	$S \leftarrow \emptyset$
3	$Q \leftarrow V[G]$
4	**WHILE** $Q \neq \emptyset$ **DO**
5	$u \leftarrow$ KIVESZ_MIN(Q)
6	$S \leftarrow S \cup \{u\}$
7	**FOR** $\forall v \in szomszéd[u]$-ra **DO**
8	KÖZELÍT(u, v, w)

EGY_FORRÁS_KEZDŐÉRTÉK(G, s) $\Theta(V + E)$
FOR $\forall u$ csúcsra azok topologikus sorrendjében **DO**
FOR $\forall v \in szomszéd[u]$-ra **DO**
KÖZELIT(u, v, w)

Bellman-Ford algoritmus: Az algoritmus az adott kezdőcsúcsból induló legrövidebb utak problémáját abban az általánosabb esetben oldja meg, amikor az élek között negatív súlyúakat is találhatunk.

	BELLMAN–FORD(G, w, s) O(VE)
1	EGY_FORRÁS_KEZDŐÉRTÉK(G, s)
2	**FOR** $i \leftarrow 1$ **TO** $\mid V[G] \mid -1$ **DO**
3	**FOR** $\forall(u, v) \in E[G]$-re **DO**
4	KÖZELÍT(u, v, w)
5	**FOR** $\forall(u, v) \in E[G]$-re **DO**
6	**IF** $d[v] > d[u] + w(u, v)$
7	**THEN** RETURN(HAMIS)
8	RETURN (IGAZ)

Floyd-Warshall algoritmus: A Floyd-Warshall algoritmus a p út és az olyan legrövidebb utak kapcsolatát alkalmazza, melynek belső csúcsait az {1, 2, ... , k−1} részhalmazból választjuk.

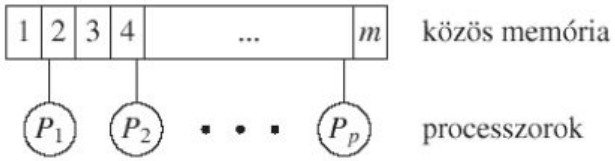
	Floyd-Warshall(W)
1	$n \leftarrow$ sorok_száma[W] $\Theta(n^3)$
2	$D^{(0)} \leftarrow W$
3	**FOR** $k \leftarrow 1$ **TO** n **DO**
5	**FOR** $i \leftarrow 1$ **TO** n **DO**
6	**FOR** $j \leftarrow 1$ **TO** n **DO**
7	$d_{ij}^{(k)} \leftarrow \min\{d_{ij}^{(k-1)}, d_{ik}^{(k-1)} + d_{kj}^{(k-1)}\}$
8	**RETURN** $D^{(n)}$

7. Párhuzamos algoritmusok alapfogalmai, hatékonysági mértékek. Párhuzamos gépek: prefixszámítás, determinisztikus tömbrangsorolás, összefésülés, kiválasztás, rendezés.

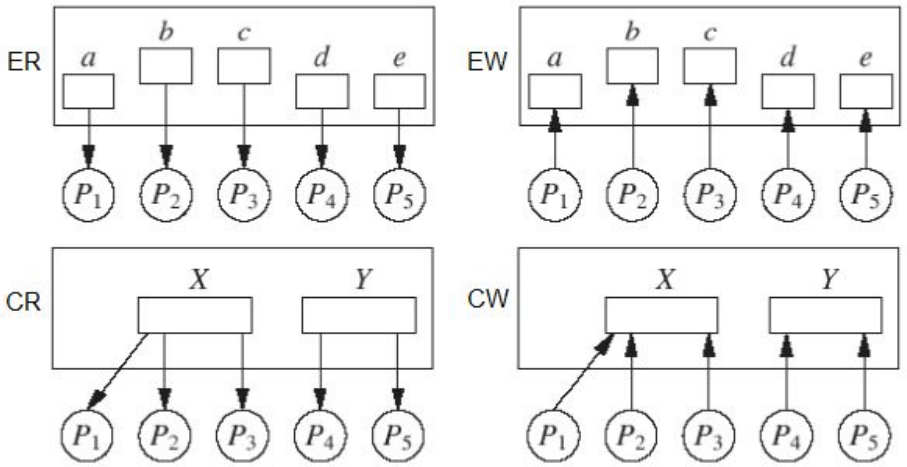
Párhuzamos algoritmusok alapfogalmai: feltesszük, hogy az f és g függvények a pozitív egészek halmazán vannak értelmezve, az f(n) és g(n) függvényértékek pedig nemnegatív valós számok.
O: aszimptotikus felső korlát, Ω: aszimptotikus alsó korlát, Θ: a vizsgált függvény aszimptotikus viselkedése
1. O(g(n)) (nagy ordó): ha $n \geq n_0$, akkor $\mathbf{f(n) \leq cg(n)}$, ahol f(n) függvények halmazához $\forall$ c pozitív valós és n_0 pozitív egész állandók
2. Ω(g(n)) (nagy omega): ha $n \geq n_0$, akkor $\mathbf{f(n) \geq cg(n)}$.
3. Θ(g(n)) (nagy teta): ha $n \geq n_0$, akkor $\mathbf{c_1\, g(n) \leq f(n) \leq c_2\, g(n)}$, ahol $\forall$ pozitív valós c_1, c_2 állandók
4. o(g(n)) (kis ordó): ha $n \geq n_0$, akkor $\mathbf{f(n) \leq cg(n)}$, ahol $\exists$ pozitív valós c állandóhoz megadható egy pozitív egész n_0
5. ω(g(n)) (kis omega): ha $n \geq n_0$, akkor $\mathbf{f(n) \geq cg(n)}$, ahol $\exists$ pozitív valós c_1, c_2 állandóhoz megadható egy pozitív egész n_0
6. Ha f(n) = o(g(n)), akkor $\lim(n\to\infty)\ f(n)/g(n) = 0$, és ha f(n) = ω(g(n)), akkor $\lim(n\to\infty)\ f(n)/g(n) = \infty$.
7. O˜(g(n)) (logaritmikus nagy ordó): ha $n \geq n_0$, akkor $\mathbf{f(n) \leq cg(n)(\lg n)^k}$, ahol $\forall$ c pozitív, valamint n_0 és k pozitív egész állandók
8. Oˆ(g(n)) (polinomiális nagy ordó): ha $n \geq n_0$, akkor $\mathbf{f(n) \leq cg(n)n^k}$.

Hatékonysági mértékek: Legyen P párhuzamos és A soros algoritmus.
1. Ha $\mathbf{pW(n,P,\ p) = O\hat{}(W(n,\ A))}$, akkor P A-ra nézve polinomiálisan munkahatékony.
2. Ha egy párhuzamos algoritmus nem polinomiálisan munkahatékony, de a megfelelő feladatot mindig helyesen megoldja, akkor a munkahatékonysága exponenciális.
3. Ha $\mathbf{pW(n,P,\ p) = O\tilde{}(W(n,\ A))}$, akkor P A-ra nézve logaritmikusan munkahatékony.
4. Ha $\mathbf{pW(n,P,\ p) = O(W(n,\ A))}$, akkor P A-ra nézve munkaoptimális.

Párhuzamos gépek:
- RAM (Random Access Memory): soros számításokhoz
- PRAM (Parallel Random Access Memory): párhuzamos számításokhoz



Típusai:
- EREW (Exclusive Read – Exclusive Write): kizárólagos olvasás – kizárólagos írás
- ERCW (Exclusive Read – Concurrent Write): kizárólagos olvasás – egyidejű írás
- CREW (Concurrent Read – Exclusive Write): egyidejű olvasás – kizárólagos írás
- CRCW (Concurrent Read – Concurrent Write): egyidejű olvasás – egyidejű írás



Prefixszámítás: Legyen Σ egy alaphalmaz (zárt), melyen definiáltuk a $\oplus$ bináris asszociatív operátort. Legyenek az $X = \langle x_1, x_2, \ldots, x_p \rangle$ sorozat elemei a Σ alaphalmaz elemei. Ekkor a prefixszámítás bemenő adatai az X sorozat elemei, a prefixszámítási feladat pedig az $x_1, x_1 \oplus x_2, \ldots, x_1 \oplus x_2 \oplus x_3 \oplus \cdots \oplus x_p$ elemek meghatározása. Ezeket a meghatározandó elemeket prefixeknek hívjuk.

CREW-PREFIX(p, X, Y) párhuzamos rekurzív eljárás
Számítási modell: p CREW PRAM $\Theta(\log p)$

Input: p (a bemenet hossza) és $X[1 : p] = \langle x_1, x_2, \ldots, x_p \rangle$ (a bemenő adatok)
Output: $Y[1 : p] = \langle y_1, y_2, \ldots, y_p \rangle$ (a kimenő sorozat, azaz a prefix)
1. IF p = 1 THEN
Δ Az i az elem indexe az X inputból.
2. $y_i \leftarrow x_i$
3. IF p > 1 THEN
4. CALL CREW-PREFIX(p/2, X[1..p/2], Y [1..p/2])
5. CALL CREW-PREFIX(p/2, X[p/2 + 1..p], Y [p/2 + 1..p])
6. P_i IN PARALLEL FOR $i \leftarrow (p/2 + 1)$ TO p DO
7. $y_i \leftarrow y_{p/2} \oplus y_i$
8. RETURN Y

EREW-PREFIX(p, X, Y) párhuzamos eljárás
Számítási modell: p EREW PRAM $\Theta(\log p)$

Input: p (a bemenet hossza) és $X[1 : p] = \langle x_1, x_2, \ldots, x_p \rangle$ (a bemenő adatok)
Output: $Y[1 : p] = \langle y_1, y_2, \ldots, y_p \rangle$ (a kimenő sorozat, azaz a prefix)
1. $Y[1] \leftarrow X[1]$
2. P_i IN PARALLEL FOR $i \leftarrow 2$ TO p DO
3. $Y[i] \leftarrow X[i - 1] \oplus X[i]$
4. $k \leftarrow 2$
5. WHILE $k < p$
6. P_i IN PARALLEL FOR $i \leftarrow k + 1$ TO p DO
7. $Y[i] \leftarrow Y[i - k] \oplus Y[i]$
8. $k \leftarrow k + k$
9. RETURN Y

OPTIMÁLIS-PREFIX(p, X, Y) párhuzamos eljárás
Számítási modell: $\lceil p/\log p \rceil$ CREW PRAM $\Theta(\log p)$

Input: p (a bemenet hossza) és $X[1 : p] = \langle x_1, x_2, \ldots, x_p \rangle$ (a bemenő adatok)
Output: $Y[1 : p] = \langle y_1, y_2, \ldots, y_p \rangle$ (a kimenő sorozat, azaz a prefix)
1. P_i IN PARALLEL $i \leftarrow 1$ TO $\lceil p/\log p \rceil$ DO
2. $z_{(i-1)\log p+1} \leftarrow x_{(i-1)\log p+1}$
3. FOR $j \leftarrow 2$ TO log p DO
4. $z_{(i-1)\log p+j} \leftarrow z_{(i-1)\log p+j-1} \oplus x_{(i-1)\log p+j}$
Δ Legyen $Z^* = \{z_{\log p}, z_{2\log p}, \ldots, z_p\}$, $W^* = \{w_{\log p}, w_{2\log p}, \ldots, w_p\}$.
5. P_i IN PARALLEL $i \leftarrow 1$ TO $\lceil p/\log p \rceil$ DO
6. CALL CREW-PREFIX($\lceil p/\log p \rceil$, Z^*, W^*)
7. P_i IN PARALLEL $i \leftarrow 1$ TO $\lceil p/\log p \rceil$ DO
8. IF i = 1 THEN
9. FOR $j \leftarrow 1$ TO log p DO
10. $y_j \leftarrow z_j$
11. ELSE
12. FOR $j \leftarrow 1$ TO log p DO
13. $y_{(i-1)\log p+j} \leftarrow w_{(i-1)\log p} \oplus z_{(i-1)\log p+j}$
14. RETURN Y

Tömbrangsorolási feladat: A tömbrangsorolási feladat bemenő adata egy p elemű tömbben ábrázolt lista: minden elem tartalmazza jobb oldali szomszédjának az indexét. A feladat az elemek rangjának (jobb oldali szomszédai számának) meghatározása.

Determinisztikus tömbrangsorolás: Először mindegyik elem a jobb oldali szomszédjának indexét tartalmazza, és ennek megfelelően a rangja 1. Ezután módosítjuk a csúcsokat úgy, hogy mindegyik a jobb oldali szomszédjának a jobb oldali szomszédjára mutasson.

DET-RANGSOROL(szomsz[1 : p], rang[1 : p])	párhuzamos eljárás
Számítási modell: p EREW PRAM	Θ(log p)

```
Input: szomsz[1 : p] (az indexek)
Output: rang[1 : p] (a rangok)
1. P_i IN PARALLEL FOR i ← 1 TO p DO
2.       IF szomsz[i] = 0 THEN
3.               rang[i] ← 0
4.       ELSE
5.               rang[i] ← 1
6. FOR j ← 1 TO ⌈log p⌉ DO
7.       P_i IN PARALLEL FOR i ← 1 TO p DO
8.               IF szomsz[i] ≠ 0 THEN
9.                       rang[i] ← rang[i] + rang[szomsz[i]]
10.                      szomsz[i] ← szomsz[szomsz[i]]
11. RETURN rang
```

Összefésülés: Adott 2 csökkenőleg (vagy növekvőleg) rendezett sorozat, melyek együtt p elemet tartalmaznak. A feladat ennek a sorozatnak egy csökkenő (vagy növekvő) sorozattá való rendezése.

PÁROS-PÁRATLAN-ÖSSZEFÉSÜL(X1, X2, Y)	párhuzamos rekurzív eljárás
Számítási modell: 2m EREW PRAM	O(log m)

Input: X_1 és X_2 (a bemenő sorozatok)
Output: L (az összefésült sorozat)
1. IF m = 1 THEN
2. Fésüljük össze a sorozatokat egyetlen összehasonlítással.
3. IF m > 1 THEN
4. $X_1^{pn} = \langle k_1, k_3, \dots, k_{m-1}\rangle$, $X_1^{ps} = \langle k_2, k_4, \dots, k_m\rangle$, $X_2^{pn} = \langle k_{m+1}, k_{m+3}, \dots, k_{2m-1}\rangle$, $X_2^{ps} = \langle k_{m+2}, k_{m+4}, \dots, k_{2m}\rangle$.
Δ Legyen $L_1 = \langle l_1, l_2, \dots, l_m\rangle$ és $L_2 = \langle l_{m+1}, l_{m+2}, \dots, l_{2m}\rangle$.
5. CALL PÁROS-PÁRATLAN-ÖSSZEFÉSÜL(X_1^{pn}, X_2^{pn}, L_1)
6. CALL PÁROS-PÁRATLAN-ÖSSZEFÉSÜL(X_1^{ps}, X_2^{p2}, L_2)
7. Fésüljük össze az L_1, L_2 sorozatokat, azaz legyen $L = \langle l_1, l_{m+1}, l_2, l_{m+2}, \dots, l_m, l_{2m}\rangle$.
8. Hasonlítsuk össze az (l_{m+i}, l_{i+1}) $(i = 1, 2, \dots, m-1)$ párokat és szükség esetén cseréljük fel őket.
9. RETURN L

Kiválasztás: Adott n ≥ 2 kulcs és egy i (1 ≤ i ≤ n) egész szám. A feladat az i-edik legkisebb kulcs kiválasztása.

NÉGYZETES-KIVÁLASZT(K, y)	párhuzamos eljárás
Számítási modell: n^2 CRCW PRAM	O(1)

Input: $K = \langle k_1, k_2, \dots, k_n\rangle$ (n különböző elem)
Output: y (a maximum)
1. IF n = 1 THEN
2. $y \leftarrow k1$
3. ELSE
4. P_{ij} IN PARALLEL FOR $i \leftarrow 1$ TO n, $j \leftarrow 1$ TO n DO
5. $x_{ij} \leftarrow (k_i < k_j)$ Δ x_{ij} egy logikai változó.
6. Az n 2 processzort n csoportba ($G_1, \dots, G_n$) osztjuk úgy, hogy a G_i csoportba a $P_{i1}, \dots, P_{in}$ processzorok kerüljenek.
7. G_i IN PARALLEL FOR $i \leftarrow 1$ TO n DO
8. $g_i \leftarrow (x_{i1} \vee \dots \vee x_{in})$
9. IF g_i = FALSE THEN
10. $y \leftarrow k_i$
11. RETURN y

GYÖKÖS-KIVÁLASZT(K, y)	párhuzamos rekurzív eljárás
Számítási modell: p CRCW PRAM	O(log log p)

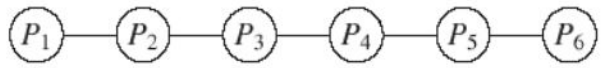
Input: $K = \langle k_1, k_2, \dots, k_p\rangle$ (p különböző elem)
Output: y (a maximum)
1. IF p = 2 THEN
Δ Az i, j indexek az inputból.
2. $y \leftarrow k_i$ (ha $k_i > k_j$) OR $y \leftarrow k_j$ (ha $k_i < k_j$)
3. ELSE
4. Osszuk fel a K bemenetet $\sqrt{p}$ = a részre úgy, hogy a K_i rész tartalmazza a $k_{(i-1)a+1}, k_{(i-1)a+2}, \dots, k_{ia}$ elemeket. Hasonlóan készítsünk csoportokat a processzorokból úgy, hogy a Q_i csoport $(1 \le i \le a)$ a $P_{(i-1)a+1}, P_{(i-1)a+2}, \dots, P_{ia}$ processzorokat tartalmazza.
4. Q_i IN PARALLEL FOR $i \leftarrow 1$ TO a DO
5. CALL GYÖKÖS-KIVÁLASZT(K_i, M_i)
6. CALL NÉGYZETES-KIVÁLASZT($M_1, \dots, M_a$, y)
7. RETURN y

Rendezés: Adott n ≥ 2 kulcs. A feladat ezek csökkenő vagy növekvő sorrendbe való rendezése.

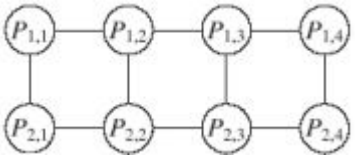
PÁROS-PÁRATLAN-RENDEZ(X, Y)	párhuzamos rekurzív eljárás
Számítási modell: n EREW PRAM	$O(\log^2 n)$
Input: X (a bemenet)	
Output: Y (a kimenet)	
1. IF n = 1 THEN	
2. $Y \leftarrow X$	
3. ELSE	
4. Osszuk fel a bemenetet két alsorozatra: $X_1' = \langle k_1, k_2, \dots, k_{n/2} \rangle$ és $X_2' = \langle k_{n/2+1}, k_{n/2+2}, \dots, k_n \rangle$.	
5. CALL PÁROS-PÁRATLAN-RENDEZ(X_1', X_1)	
6. CALL PÁROS-PÁRATLAN-RENDEZ(X_2', X_2)	
7. CALL PÁROS-PÁRATLAN-ÖSSZEFÉSÜL(X_1, X_2, Y).	
8. RETURN Y	

8. Rácsok: csomagirányítás, üzenetszórás, prefixszámítás. Hiperkocka, pillangó hálózat, hálózatok beágyazása, csomagirányítás. Szinkronizált hálózat: vezetőválasztási algoritmusok.

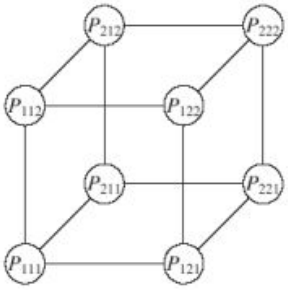
Rács: A k dimenziós (k ≥ 1) rács egy olyan $m_1 \times m_2 \times \dots \times m_k$ ($m_1, m_2, \dots, m_k \geq 2$) méretű háló, amelynek minden metszéspontjában van egy processzor. Az élek a kommunikációs vonalak, melyek kétirányúak.
Topológiák:
- lánc: A legegyszerűbb rács a k = 1 értékhez tartozó rács. Egy lánc processzorai $P_1, P_2, \dots, P_p$. P_1 és P_p kivételével mindegyik processzor össze van kötve a nála eggyel nagyobb, illetve eggyel kisebb indexűvel.



- téglalap/négyzet: Ha k = 2, akkor téglalap alakú rácsot kapunk. Ha most $m_1 = m_2 = \sqrt{p} = a$, akkor a × a méretű négyzetet kapunk.



- tégla/kocka: Ha k = 3, akkor tégla alakú rácsot kapunk. Az $m_1 = m_2 = m_3 = \sqrt[3]{p}$ speciális esetben kockáról és a kocka méretére az n × n × n jelölést alkalmazzuk.



Csomagirányítás: processzorok közötti kommunikáció egyetlen lépése egy rögzített szerkezetű hálózatban
- feladat: A hálózatban minden processzornak van egy adatcsomagja, amit egy másik processzornak akar elküldeni. A feladat a csomagok eljuttatása a céljukhoz a lehető leggyorsabban úgy, hogy egy lépésben egy kommunikációs csatornán egy irányban egyszerre csak egy csomag utazhat.
- probléma: egy adott lépésben kettő vagy több csomag érkezik egy processzorhoz, és mindegyik ugyanazon a csatornán szeretne továbbhaladni, ekkor csak egy csomag utazhat a következő lépésben, a többiek pedig a processzornál egy várakozási sorba kerülnek a későbbi továbbküldés miatt

- megoldás: valamilyen elsőbbségi szabály alapján döntjük el, hogy melyik csomagot küldjük el
    - FDF (Farthest Destination First): legtávolabbra utazó csomag először
    - FOF (Farthest Origin First): legtávolabbról jött csomag először
    - FIFO (First In First Out): előbb érkezett csomag először
    - RAN (RANDOM): véletlenül választunk

Üzenetszórás: a hálózat megadott processzorától üzenetet kell eljuttatni megadott célprocesszorokhoz
- láncon: Legyen L egy p processzoros lánc és legyen M egy üzenet, amelyet a P_1 processzornak kell elküldenie a többi processzorhoz. Megoldás: P_1 elküldi az üzenetet P_2-nek és így tovább. Ekkor tehát az üzenet p−1 lépésben (lánc átmérője) eljut a legtávolabbi processzorhoz, P_p-hez.
- négyzeten: Egy a × a méretű négyzetrácsban az üzenetküldést két fázisban valósíthatjuk meg. Az első fázisban az üzenetet küldő $P_{i,j}$ processzor eljuttatja üzenetét az i-edik sor minden processzorához. Ezután a második fázisban az i-edik sor minden processzora eljuttatja az üzenetet a vele azonos oszlopban lévő processzorokhoz. Ez összesen legfeljebb 2(a−1) lépést vesz igénybe.

Prefixszámítás:

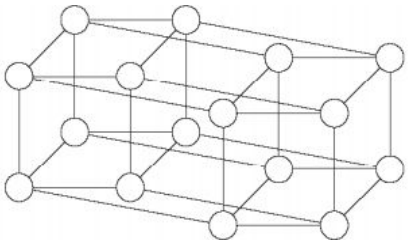
LÁNC-PREFIX(L, X, Y) párhuzamos eljárás
Számítási modell: p lánc $\Theta(p)$

Input: $X = \langle x_1, x_2, \dots, x_p \rangle$ (az összeadandó elemek)
Output: $Y = \langle y_1, y_2, \dots, y_p \rangle$ (a prefixek)
1. P_i IN PARALLEL FOR i ← 1 TO p DO
2. IF i = 1 THEN
3. P_1 az első lépésben elküldi x_1-et P_2-nek.
4. IF i = p THEN
5. P_p a p-edik lépésben kap egy elemet (y_{p-1}-et) P_{p-1}-től, kiszámítja és tárolja $y_{p-1} \oplus x_p$-t.
6. IF i ≠ 1 ∧ i ≠ p THEN
7. P_i az i-edik lépésben kap egy elemet (y_{i-1}-t) P_{i-1}-től, kiszámítja és tárolja $y_i = y_{i-1} \oplus x_i$ -t, majd y_i -t továbbküldi P_{i+1}-nek.
8. RETURN Y

NÉGYZETEN-PREFIX(X, Y) párhuzamos eljárás
Számítási modell: a × a négyzet $O(a) = 3a + 2$

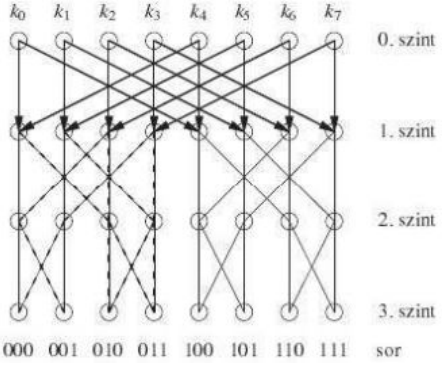
Input: $X = \langle x_1, x_2, \dots, x_p \rangle$ (az összeadandó elemek)
Output: $Y = \langle y_1, y_2, \dots, y_p \rangle$ (a prefixek)
Δ Jelöljük a $P_{i,1}, P_{i,2}, \dots, P_{i,a}$ processzorokat az R_i lánccal.
1. $P_{i,1}$ IN PARALLEL FOR i ← 1 TO a DO
2. CALL LÁNC-PREFIX(R_i , X[i], Y [i])
3. $P_{j,a}$ IN PARALLEL FOR j ← 1 TO a − 1 DO
4. Elküldi a kiszámolt prefixet $P_{j+1,a}$-nak.
5. $P_{i,j}$ IN PARALLEL FOR i ← 2 TO a, j ← 1 TO a − 1 DO
6. Kiszámolja és tárolja $y_{i-1,a} \oplus y_{i,j}$ -t.
7. RETURN Y

Hiperkocka ($\mathcal{H}_d$): Egy d dimenziós hiperkocka 2^d processzorból áll, amelyek d hosszúságú bináris sorozatokkal címezhetők.



- minden P_i processzor pontosan azokkal a processzorokkal van összekötve, amelyek címe pontosan egy bitben tér el P_i címétől
- mivel $\mathcal{H}_d$ minden processzora pontosan d másikkal van összekötve, így $\mathcal{H}_d$ d-reguláris és a fokszáma d
- két processzor között vezető út: Legyenek $i_1, i_2, \dots, i_k$ azon bitpozíciók (növekvő sorrendben) amelyeken P_u és P_v eltérnek. Ekkor létezik a következő útvonal: $u = w_0, w_1, w_2, \dots, w_k = v$, ahol $w_j = w_{j-1}(i_j)$ $(1 \le j \le k)$. Ebből következik, hogy egy d dimenziós hiperkocka átmérője pontosan d.

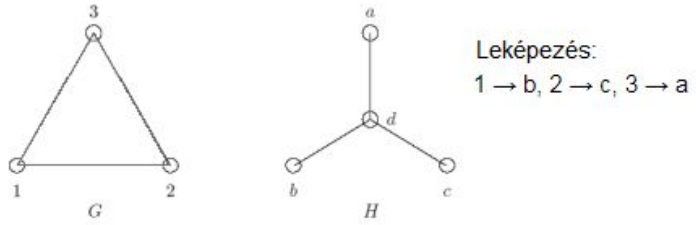
Pillangóhálózat ($\mathcal{B}_d$): Egy d dimenziós pillangóhálózat $p = (d + 1)2^d$ processzorból és $d2^{d+1}$ élből áll.



- $\mathcal{B}_d$ minden processzora egy ⟨r, l⟩ párral jellemezhető, ahol $0 \le r \le 2d - 1$ és $0 \le l \le d$. Ekkor t a processzor sorindexe, l pedig a szintje.
- Egy $P_u = \langle r, l \rangle$ processzor $(0 \le l < d)$ $\mathcal{B}_d$-ben két, az (l + 1)-edik szinten levő processzorral van összekötve, a $P_v = \langle r, l+1 \rangle$ és $P_w = \langle r(l+1), l+1 \rangle$ processzorokkal (ahol a r(l+1) az r bináris szám balról jobbra olvasott l+1-dik bitjének átbillentését jelenti). Az (u, v) kapcsolatot közvetlen kapcsolatnak, (u, w)-t pedig kereszt kapcsolatnak nevezzük.

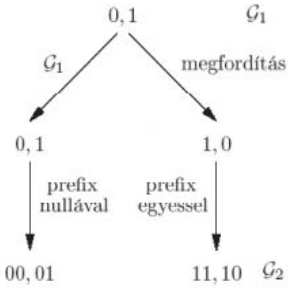
- mohó út: Ha P_u egy 0-adik szintű processzor és P_v egy d-edik szintű, akkor létezik egy d hosszúságú út P_u és P_v között. Legyen $P_u = \langle r, 0\rangle$ és Pv = $\langle r', d\rangle$. Ekkor az út $\langle r, 0\rangle$, $\langle r_1, 1\rangle$, $\langle r_2, 2\rangle$, ... ,$\langle r', d\rangle$, ahol r_i -nek az első i bitje megegyezik r'-vel, a többi pedig r-rel ($1 \le i \le d-1$). Tehát $\mathcal{B}_d$-ben bármely két processzor távolsága legfeljebb 2d.

Hálózatok beágyazása: Egy hálózatnak egy másikra történő leképezését beágyazásnak nevezzük. Formálisan, a $G(V_1, E_1)$ hálózat beágyazása a $H(V_2, E_2)$ hálózatba, egy leképezés V_1-ről V_2-re.



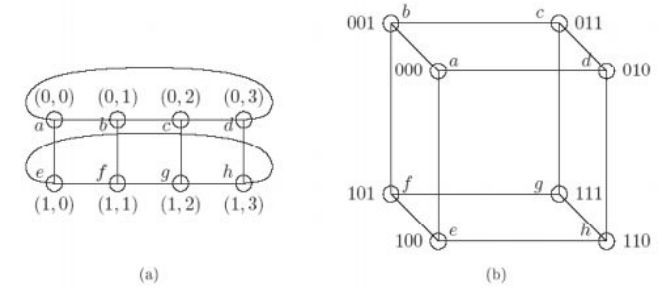
- a hálózat egy éle torlódásának nevezzük az adott élt használó olyan utak számát, amelyekre G valamely élét leképeztük
- egy beágyazás felfúvódása a |V2|/|V1| hányados, késleltetése a leghosszabb út hossza, amelyre G-nek egy éle leképződött és torlódása a H élei torlódásának maximuma

2^d processzoros gyűrű beágyazása $\mathcal{H}_d$-be: $\mathcal{H}_d$ processzorait d bites bináris számokkal címkézzük. Ha a gyűrű processzorait 0-tól (2d−1)-ig indexeljük a gyűrű mentén, akkor a 0-s processzor a $\mathcal{H}_d$ 000...00 jelű processzorára fog leképződni, a többi processzor megfelelőjét a Gray-kód segítségével határozhatjuk meg.
- k-ad rendű Gray-kód (G_k): a k bites bináris számok egy adott permutációját definiálja, ahol a szomszédos elemek pontosan egy bitben térnek el egymástól (G_k i-edik elemét g(i, k)-val jelöljük ($0 \le i \le 2^k-1$))

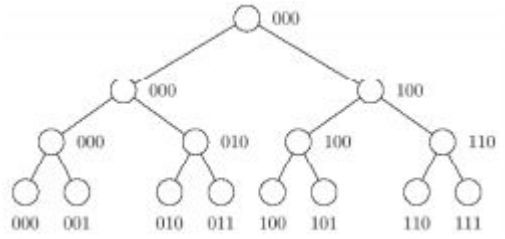


G_d a $\mathcal{H}_d$ processzorainak egy olyan permutációja, hogy a sorban egymást követő processzorok össze vannak kötve éllel a hiperkockában, azaz a gyűrű i-edik processzorát a hiperkocka g(i, d) processzorára képezzük le ($0 \le i \le 2^d-1$). Ebből következik, hogy a felfúvódás, a késleltetés és a torlódás mindegyike 1.

Tórusz beágyazása $\mathcal{H}_d$-be: Legyen M egy $2^r \times 2^c$ méretű tórusz. Ha egy $\mathcal{H}_d$-ban a d bitből valamely q-t rögzítjük ($1 \le q \le d-1$), akkor a processzorok egy $\mathcal{H}_{d-q}$ alkockát alkotnak $\mathcal{H}_d$-ben. Ha egy (r+c)-bites bináris számnak rögzítjük az r legnagyobb helyi értékű bitjét (MSB) és a maradék c bitet tetszőlegesen variáljuk, a kapott 2^c szám egy alkockát határoz meg $\mathcal{H}_{r+c}$-ben. Ebbe az alkockába beágyazható egy 2^c processzorból álló gyűrű. Az r MSB minden lehetséges megválasztásához megvan a megfelelő $\mathcal{H}_c$, így M i-edik sorát azon $\mathcal{H}_c$-re képezzük, ahol az r MSB értékét pontosan g(i, r)-re állítjuk. Így a tórusz $P_{i,j}$ processzora a $\mathcal{H}_{r+c}$ $P_{g(i,r),g(j,c)}$ processzorára képződik. Belátható, hogy minden sor szomszédos processzorai a $\mathcal{H}_d$ szomszédos processzoraira képződnek és ugyanez igaz az oszlopokra is. Ebből következik, hogy mind a felfúvódás a késleltetés és a torlódás 1.



p levelű fa beágyazása $\mathcal{H}_d$-be: Egy d szintes bináris fában $p = 2^d-1$ processzor van: $P_1, P_2, \ldots, P_p$. Ekkor a P_1 processzort gyökérnek, a $P_{(p+1)/2}, P_{(p+1)/2+1}, \ldots, P_p$ processzorokat levélnek, a többi processzort belső processzornak nevezzük. Ha P_i nem levél, akkor össze van kötve a gyerekeinek nevezett P_{2i} és P_{2i+1} processzorokkal. Ha P_j nem a gyökér, akkor össze van kötve a szülőjének nevezett $P_{\lfloor j/2 \rfloor}$ processzorral. Mivel a p levelű fának $2p-1$ processzora van, így a leképezés nem lehet kölcsönösen egyértelmű. Ha a leveleket 0, 1, ... , p−1 jelöli, akkor képezzük az i-edik levelet $\mathcal{H}_d$ i-edik processzorára, a belső processzorokat pedig képezzük az adott processzor legbaloldalibb leszármazottjára.



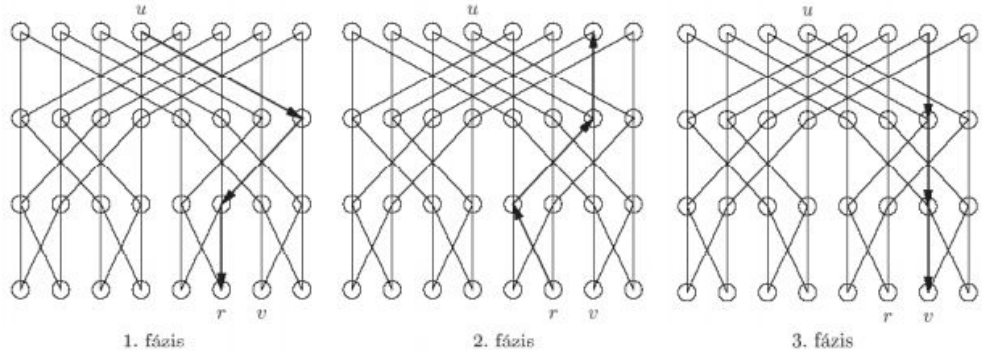
Csomagirányítás: Minden processzor legfeljebb egy csomagot küld és minden processzor legfeljebb egy csomag címzettje. Juttassuk el a csomagokat a feladótól a címzettekhez.

Mohó algoritmus $\mathcal{B}_d$-n: minden csomagot a mohó úton küldünk címzettjéhez, ahol minden csomag pontosan d hosszúságú utat tesz meg
- maximális késleltetés:

$$D = \sum_{l=1}^{d} \min(2^{l-1}, 2^{d-l}).$$

- lépésszáma és maximális sorhosszúsága: $O(2^{d/2})$

Véletlenített algoritmus:a csomagot először egy véletlenszerűen választott közbülső állomásra küldjük, majd onnan továbbítjuk eredeti céljához, így a csomagok nagy valószínűséggel nem találkoznak egyetlen másik csomaggal sem útjuk során
- megoldás: háromfázisú algoritmus
1. fázis: A csomagot egy véletlenszerűen választott közbülső címre irányítjuk a d-edik szinten.
2. fázis: A csomagot az eredeti céljának megfelelő sorba irányítjuk, ám a nulladik szinten.
3. fázis: A csomagok elérik tényleges céljukat a d-edik szinten.
- lépésszáma: $\overline{O}(d)$



Szinkronizált hálózat: H = (V, E), ahol $V = \{P_1, P_2, \ldots, P_p\}$ a processzorok halmaza, ahol két processzor között egy- vagy kétirányú adatátvitel lehetséges, vagy nincs kapcsolat

Vezetőválasztási algoritmusok: Tegyük fel, hogy kezdetben minden processzor azonos állapotban van. A cél olyan állapot elérése, amelyben pontosan egy processzor a vezető a többi processzor pedig a nem-vezető állapotban van.

- LeLann algoritmusa: megengedi, hogy a vezető az úgynevezett kezdő processzorok közül kerüljön ki

LELANN(K) párhuzamos eljárás	11. WHILE a ≠ i DO
Számítási modell: egyirányú gyűrű Θ(p)	12. $J_i \leftarrow J_i \cup \{a\}$
Input: K (a kezdő processzorok indexeinek halmaza)	13. $KÜLD_i(a)$
Output: i (a vezető processzor indexe)	14. $FOGAD_i(a)$
1. P_i IN PARALLEL FOR i ← 1 TO n DO	15. IF i = min(J_i) THEN
2. IF i ∈ K THEN	16. áll[i] ← vez
3. áll[i] ← jelölt	17. l ← i
4. $J_i \leftarrow \{i\}$	18. ELSE
5. ELSE	19. áll[i] ← nem_vez
6. áll[i] ← nem_jelölt	20. ELSE
7. P_i IN PARALLEL FOR i ← 1 TO n DO	21. $FOGAD_i(a)$
Δ a egy lokális változó az indexek összehasonlításához.	22. WHILE a ≠ i DO
8. IF áll[i] = jelölt THEN	23. $KÜLD_i(a)$
9. $KÜLD_i(i)$	24. $FOGAD_i(a)$
10. $FOGAD_i(a)$	25. RETURN(l)

- Chang-Roberts algoritmusa: lényegében ugyanaz, mint a LeLann, annyi különbséggel, hogy a kezdő processzorok csak a saját azonosítójuknál kisebb azonosítókat küldik tovább
    - tehát a **11.sorban: WHILE a < i DO**
    - így az algoritmus átlagos üzenetszáma O(p log p)
- Hirschberg és Sinclair algoritmusa: a legnagyobb azonosítóval rendelkező folyamatot választja vezetőnek, azonban az azonosítók nem körbejárják a gyűrűt, hanem bizonyos (egyre nagyobb) lépés megtétele után visszafordulnak
- időszelet algoritmus: Szakaszokban működik, ahol minden szakasz p lépésből áll. A j-edik szakaszban csak j azonosítót lehet üzenetként elküldeni. Ha a P_i processzor azonosítója a_i, akkor ez a processzor az 1., 2., ... , a_{i-1}. szakaszban nem küld üzenetet. Ha a P_i processzor az első a_{i-1} szakaszban nem kap üzenetet, akkor az a_i-edik szakasz első lépésében elküldi a szomszédjának a saját azonosítóját, és ez az azonosító körbemegy az egyirányú gyűrűn.
- max-terjed algoritmus: alapötlete, hogy a processzorok minden menetben elküldik szomszédaiknak az addig hozzájuk eljutott legnagyobb azonosítót

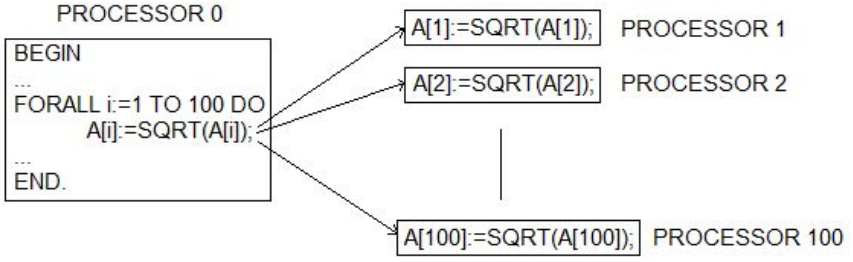
9. A párhuzamos algoritmusok elkészítésének és implementálásának technikái és problémái. A Multi-Pascal nyelv lehetőségeinek bemutatása néhány példán keresztül. A PVM bemutatása a "hello" és a "forkjoin" mintaprogramok segítségével.

Processz: Végrehajtható utasítások sorozata, amely informálisan felfogható úgy, mint egy szubrutin vagy eljárás, amely végrehajtható bármely processzoron vagy röviden csak számítási aktivitás.

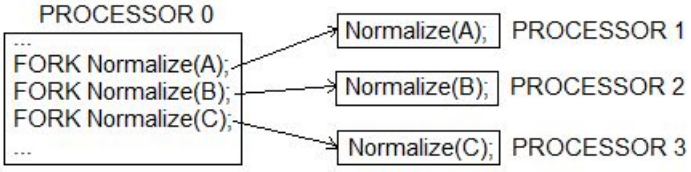
Párhuzamos programozás technikái:
- adat párhuzamosítás: hasonló vagy azonos eljárások alkalmazása/futtatása párhuzamosan nagy mennyiségű, de különböző adatokon
- adat felosztás: az adat párhuzamosítás egy olyan fajtája, melynél az adatterület szomszédos részekre van felosztva és mindegyik részen párhuzamosan működik egy-egy processzor
- relaxált algoritmus: minden egyes processz önálló módon számol, nincs szinkronizáció vagy kommunikáció a processzek között
- szinkronizált iteráció: minden egyes processzor ugyanazt az iterációt/számítást végzi el az adatok egy részén, azonban az iteráció végén szinkronizációra van szükség
- lemásolt dolgozók: hasonló számítási feladatok központi tárolóban való kezelése
- pipeline számítás: a processzek gyűrű vagy kétdimenziós hálóba vannak rendezve

Párhuzamos programozás problémái:
- memória küzdelem: a processzor várakozik egy memória cellára, mivel azt egy másik processzor használja
- kimerítő szekvenciális kód: minden párhuzamos algoritmusban lehet tisztán szekvenciális kód (pl. inicializáció)
- processz létrehozási ideje: minden valós rendszerben a processz(ek) létrehozása időt igényel
- kommunikációs késedelem: a multicomputers architektúrában fordul elő, mivel a processzoroknak kommunikálni kell egymással a hálózaton keresztül
- szinkronizációs késedelem: egy adott ponton a processznek várakoznia kell a többire
- terhelési kiegyensúlyozatlanság: néhány párhuzamos programnál az egyes generált részfeladatok jelentős különbséget mutathatnak

Multi-Pascal: párhuzamos processzek létrehozása és iterációja
Utasításai:
- FORALL: ennek segítségével hozhatjuk létre a párhuzamos processzeinket
Mivel a program maga is egy processz, így őt hívjuk első processznek, míg a többi a kreált processz (szülő-gyermek kapcsolat). A ciklusmagjában szereplő utasítást (processzt) sokszorozza meg és azok szimultán (párhuzamos végrehajtását) okozza. Szintaxisa:
FORALL <induló változó>:=<kezdeti érték> TO <végérték> [GROUPING <méret>] DO <utasítás> ;



- FORK: egyedi párhuzamos processz létrehozására szolgál
Szintaxisa: *FORK <utasítás> ;*
Az utasítás egy gyermek processzé válik, amely egy másik processzoron kerül végrehajtásra. A szülő processz ezután a gyermek processzre való várakozás nélkül folytatódik (aszinkron). Ha a szülő processz előbb fejeződne be, mint a létrehozott gyermek processzek, akkor felfüggesztésre kerül.



- JOIN: Lényegében a FORK ellentéte. Hatására a szülő processz várakozik a gyermek processz befejezésére. Ha a gyermek még nem terminált, akkor várakozik. Ha a gyermek terminált közben, akkor nincs hatása. Szintaxisa: *JOIN ;*
- CHANNEL: csatorna változó, rendelkezik egy “üres” tulajdonsággal
Induláskor a CHANNEL változó üres. Ha a P2 processz üres CHANNEL változót akar olvasni, akkor a P2 processzt addig felfüggesztjük, amíg P1 nem írja ki a változó értékét.
Szintaxisa: *<channel név>: CHANNEL OF <komponens típus>*
- SPINLOCK: adatmegosztási eszköz
Egy processznek exkluzív hozzáférést biztosít az osztott adathoz, miközben a többi processzt várakozásra készteti. Lényegében egy kaput hoz létre, amelyen a processzek egyesével haladhatnak át az adathoz. Két állapota van: zárva (locked), nyitva (unlocked). Ezeket a Lock és an Unlock utasításokkal lehet elérni. Ha a spinlock zárva van, akkor a processz vár, amíg más processz meg nem nyitja, tehát lehetővé teszi, hogy egy processz más processzeket várakoztasson.
- BARRIER: egy olyan pont a programban, ahol a párhuzamos processzek várnak egymásra
A Multi-Pascalban nincs direkt barrier utasítás, hanem channel vagy spinlock változókkal hozható létre.

PVM (Parallel Virtual Machine): elosztott rendszerek készítésére alkalmas
- elosztott rendszer: a több számítógépen futó olyan alkalmazások, ahol az alkalmazás felhasználójának nem kell tudnia az alkalmazás kezeléséhez arról, hogy az alkalmazás több számítógépen fut

“hello” program:

```c
#include "pvm3.h"
#include <stdio.h>

int main() {
    int cc, tid, msgtag;
    char  buf[100];
    printf("i'm t%x\n", pvm_mytid());

    cc = pvm_spawn( "hello_other", (char**)0, 0, "", 1, &tid );
    if (cc== 1) {
    msgtag = 1;
    pvm_recv(tid, msgtag);
    pvm_upkstr(buf);
    printf("from t%x: %s\n", tid, buf);
    } else printf("can't start hello_other\n");

    pvm_exit();
    return 0;
}
```

“hello_other” program:

```c
#include "pvm3.h"
#include <string.h>
#include <unistd.h>

int main() {
    int ptid, msgtag;
    char buf[100];
    ptid = pvm_parent();

    strcpy(buf, "hello, world from ");
    gethostname(buf + strlen(buf), 64);
    msgtag = 1;
    pvm_initsend(PvmDataDefault);
    pvm_pkstr(buf);
    pvm_send(ptid, msgtag);
    pvm_exit();
    return 0;
}
```

A *hello* program elindítása után kiírja a saját tid-jét (a **pvm_mytid** segítségével kapja meg), majd elindítja a hello_other programot a **pvm_spawn** segítségével. Sikeres indítás után jut hozzá egy blokkolt üzenethez a **pvm_recv**-vel, melyet a **pvm_upkstr** csomagol ki, végül a PVM rendszerből a **pvm_exit** távolítja el.
A *hello_other* program elindításával lekérdezésre kerül a szülő tid értéke (**pvm_parent()**), majd a **pvm_initsend** (buffer allokálás), a **pvm_pkstr** (becsomagolás) és a **pvm_send** (küldés) segítségével elküldésre kerül a *hello* programnak a gépnév (hostname) és a „hello, world from” szövegrész.

“forkjoin” program:
A program indulásakor megtörténik a tid lekérdezése (**pvm_mytid**), amely minden más parancsot megelőz. Ez az azonosító pozitív egész szám, ha nem, akkor valamilyen probléma lépett fel pl. a futtatás során. Ezt a program ellenőrzi és adott esetben a **pvm_perror** (argumentuma az argv[0], amely a program neve) által terminálódik. Ezen utóbbi parancs kiírja, hogy mi volt a probléma a legutolsó PVM függvény hívásakor.
Ha a tid lekérdezése rendben megtörtént, akkor jön a szülő tid-jének lekérdezése. Mivel induláskor ez a szülő, így ekkor egy hiba kódot kapunk vissza egész érték helyett: PvmNoParent. Ennek segítségével tudjuk szétválasztani a szülő-gyermek programrészeket. A program indításakor meg lehet adni, hogy mennyi processz induljon el. Ez az érték az **ntask** paraméteren keresztül jelenik meg a programban. Alapértelmezett értéke 3, ha viszont 0-tól kisebb, vagy a megadottól nagyobb értéket adunk meg a programnak, akkor a program meghívja a **pvm_exit**-et, azaz kilép a rendszerből. A létrehozott gyermek processzek mindegyike üzenetet küld a szülőnek, melyből a szülő pl. a küldő processz tid-jét kiíratja a képernyőre.

10. A cmake és az R nyelv használatának bemutatása. Automatikus vektorizálás.

Cmake: eszközrendszer Makefile előállítására
- forráskódjainkhoz CMakeLists.txt nevű állományokat hozunk létre, melyekben megadjuk, hogy mely forráskódokból szeretnénk alkalmazást vagy programkönyvtárat létrehozni, megadhatjuk a fordítóprogram és linker kapcsolóit, a forráskód csomagfüggőségeit, stb.

Használata: A CMakeLists.txt állományt tartalmazó könyvtárban a parancssorból kiadjuk a **cmake .** utasítást, melynek határása a cmake alkalmazás előállítja a fordításhoz/linkeléshez szükséges GNU Makefile-t. Miután előállt a Makefile, a **make** utasítás kiadásával fordíthatjuk le a teljes forráskódot.
Fontosabb makrók:
- *SET(<változó neve> <értéke (sztring)>)*: ezzel adhatunk értéket a változóknak
- *PROJECT(<név> <nyelv>)*: projekt nevét és nyelvét adhatjuk meg
- *SUBDIRS(<könyvtárak> ...)*: feldolgozásra váró konfigurációs állományokat tartalmazó könyvtárak felsorolása
- *AUX_SOURCE_DIRECTORY(<könyvtár> <változó neve>)*: lefordítandó forrásfájlok összegyűjtése
- *INCLUDE_DIRECTORIES(<könyvtárak> ...)*: azon könyvtárak felsorolása, amelyek -I kapcsolóval kerülnek be a fordítási parancsba
- *LINK_DIRECTORIES(<könyvtárak> …)*: linkelendő könyvtárak felsorolása (-L kapcsoló)
- *ADD_EXECUTABLE(<futtatható alkalmazás neve> <forrásfájlok> …)*: futtatható fájl specifikálása
- *TARGET_LINK_LIBRARIES(<könyvtárak> …)*: azon könyvtárak megadása, amelyeket a létrehozandó alkalmazáshoz kell linkelni

Beépített változók:
- *CMAKE_C_FLAGS*: az értékül adott sztring megjelenik a fordítási parancsban
- *CMAKE_BUILD_TYPE*: buildelés tipusának megadása (release/debug)
- *PROJECT_SOURCE_DIR*: projekt gyökérkönyvtárának elérési útja
- *LIBRARY_OUTPUT_PATH*: az a könyvtár, ahová az elkészült programkönyvtárak kerülnek
- *EXECUTABLE_OUTPUT_PATH*: az a könyvtár, ahová az elkészült futtatható alkalmazások kerülnek
- *PROJECT_NAME*: a projekt neve

R nyelv: statisztikai számításokra és grafikus ábrázolásra alkalmas programozási nyelv
- *?parancs*: help menü a <parancs>-hoz
- beépített függvények: *sum(), prod(), min(), max(), mean(), median(), var(), sd(), sqrt(), exp(), log(), abs(), sin(), cos(), tan()*
- beépített adatkészletek: *mtcars, iris, USArrests, longley, AirPassengers*, stb.

Adatstruktúrák:
- vektor: *vektor_neve <- c(értékek)*
- sorozat: *seq(from= , to= , by= / length.out=)*
- mátrix: *matrix(értékek, nrow= , ncol= , /byrow=TRUE/)*
- adattábla: *adattábla_neve <- data.frame(értékek)*
- idősor: *ts(data= ,start= , frequency=)*

Függvények:
- saját függvény írása:

```
függvény_neve <- function (paraméterek) {
        utasítások
        return (kifejezés)
}
```

- *elágazás:*

```
if (feltételek) {
        utasítások
} else {
        utasítások
}
```

- ciklus:

```
for (ciklusváltozó in intervallum) {
        utasítások
}
```

Grafika: mindegyikbe beleilleszthető a típus: *type=*
- vektorpárok kirajzoltatása: *plot(x, y)*
- függvényábrázolás: *curve(függvény, from= , to=)*
- hisztogram: *hist(x, freq=T/F)*
- dobozábra: *boxplot(x)*
- kördiagram: *pie(x, clockwise=T/F)*
- sávdiagram: *barplot(x, horiz=T/F)*
- szövegezés az ábrákon: *mtext("szöveg"), text(x, y, “szöveg”)*
- pont, vonal, poligon ábrázolása: *points(x, y), lines(x, y), abline(x, y), rect(u, v, x, y)*

Automatikus vektorizálás: az automatikus párhuzamosítási módszerek azon csoportjára utal, amelyben egy létező forráskódot feltérképezve a fordítóprogram vektorizálható konstrukciókat azonosít, azaz olyan kódrészleteket, amelyekre hatékonyan alkalmazhatóak a processzor által támogatott vektorműveletek
Kapcsolók: *SET(CMAKE_C_FLAGS ${CMAKE_C_FLAGS} "-O_")*
-O0: nincs optimalizálás (alapértelmezett)
-O1, -O2, -O3: van optimalizálás, mértéke állítható

11. Az OpenMP API eszközei. Fordítás, pragmák, klózok, könyvtári függvények, környezeti változók. A Pthreads API eszközei. Fordítás, hibakezelés, szálak létrehozása és leállítása, szinkronizálás.

OpenMP (Open MultiProcessing): osztott memória modell alapú párhuzamos programozást támogató eszközöket specifikáló szabvány

Fordítás: *-fopenmp* kapcsolóval (enélkül szekvenciálisan fut le a program, mert a pragmákat figyelmen kívül hagyja), illetve az *omp.h* header fájlt kell include-olni a fájlokban

Pragmák: ezekkel adhatjuk meg a párhuzamosítási pontokat és a párhuzamosítás módját a forráskódban
#pragma omp <pragma -nev> [kloz[,kloz]...] általános alakban
- PARALLEL pragma: az őt követő utasítás vagy blokk több szálon, párhuzamosan kerül végrehajtásra
#pragma omp parallel [kloz[,kloz]...] <utasitas vagy blokk>
- FOR pragma: az őt követő for ciklust a tartománya szerint felosztja a szálak között
#pragma omp for [kloz[,kloz]...] (csak parallel pragmával együtt használható)
- SINGLE pragma: olyan blokkot specifikálhatunk a párhuzamos szekción belül, amelyet csak egyetlen szál fog végrehajtani
#pragma omp single [kloz[,kloz]...] <utasitas vagy blokk>
- SECTIONS pragma: az őt követő blokkban több különböző section pragmát és blokkot adhatunk meg, amelyeket az egyes szálak párhuzamosan fognak végrehajtani, ha az egész konstrukció egy parallel pragmát követő blokkban helyezkedik el
#pragma omp sections [kloz[,kloz]...] {
    [#pragma omp section] <blokk>
    ...
}
- TASK pragma: egy parallel régión belül explicit módon adhatunk meg feladatokat (taszkokat), amelyeket a parallel blokkot futtató szálak hajtanak majd végre a parallel többi utasításával párhuzamosan
#pragma omp task [kloz[,kloz]...] <utasitas vagy blokk>

Összevont pragmák:
- parallel + for pragma:
#pragma omp parallel for [kloz[,kloz]...] <utasitas vagy blokk>
- parallel + sections pragma:
#pragma omp parallel sections [kloz[,kloz]...] { [#pragma omp section] <blokk> … }

Klóz nélküli pragmák:
- *#pragma omp taskyield*: a task végrehajtása felfüggeszthető egy másik task végrehajtása érdekében
- *#pragma omp master*: az utasítást vagy blokkot a szálak közül a master szál fogja végrehajtani
- *#pragma omp critical*: egyszerre csak egy szál hajthatja végre ezt a kritikus szekciót
- *#pragma omp barrier*: ezen a ponton az egyes szálaknak be kell várniuk egymást
- *#pragma omp taskwait*: a task felfüggeszti futását egészen addig, amíg az általa generált összes task végrehajtása be nem fejeződött
- *#pragma omp flush*: a processzor a gyorsítótárbeli vagy regiszterbeli értékkel fog dolgozni
- *#pragma omp ordered*: az utasítást vagy blokkot a ciklusváltozó sorrendjében hajtják végre a szálak
- *#pragma omp atomic*: változók olvasásának és írásának értéke rögtön megjelenik a memóriában
- *#pragma omp atomic capture*: egyszerre csak egy szál hajthatja végre és a végrehajtást követően minden szál ugyanazon memóriaképet látja az érintett változóról

Klózok:
- *if(skalar_kifejezes)*: a párhuzamosítás feltételét adhatjuk meg igaz kifejezés esetén (parallel), ha hamis, akkor a taskot generáló programrész felfüggeszti a futását (task)
- *num_threads(egesz_kifejezes)*: a szálak számát adja meg (parallel)
- *private(valtozo_lista)*: a felsorolt változókból annyi példány jön létre, ahány szálon a blokk vagy utasítás elindul, kezdőértékük nem kerül inicializálásra (parallel, for, single, sections, task)
- *firstprivate(valtozo_lista)*: a felsorolt változók private tulajdonságúak lesznek és értékük az eredeti változó értékével kerül inicializálásra (parallel, single, sections, task)
- *shared(valtozo_lista)*: a felsorolt változókat megosztva használják a szálak (parallel, task)
- *copyin(valtozo_lista)*: a threadprivate változókat inicializálja a változók eredeti értékével (parallel)
- *reduction(operator:valtozo_lista)*: a megadott redukciós műveletet hajtja végre a szálak lefutása után (parallel, sections)
- *lastprivate(valtozo_lista)*: a felsorolt változókat private-ként kezelik a szálak és a ciklus végén az eredeti változók megkapják az utolsó iterációban felvett értékeket (for)
- *schedule(kind[,chunk_size])*: a ciklusváltozó lépéseinek ütemezését szabályozhatjuk vele (for)
- *collapse(n)*: a pragmához tartozó for ciklusok számát adja meg (for)
- *ordered*: olyan blokkot specifikálhatunk a ciklusban, amelyet párhuzamos végrehajtás esetén az egyes szálak a ciklusváltozó értékeinek sorrendjében hajtanak végre (for)
- *nowait*: egy parallel pragmában több for pragmát is elhelyezhetünk, melyek várakozás nélkül hajtódnak végre (for, single)
- *copyprivate(valtozo_lista)*: a szálak a listán szereplő változókból saját példányokkal rendelkeznek és azok a single blokk lefutása után a blokkot futtató szál változóinak értékeit kapják kezdőértékként (single)
- *final*(skalar_kifejezes): a taskok szekvenciálisan beágyazódnak a taskot generáló kódba (task)
- *untied*: a task nincsen szálhoz kötve, így ha felfüggesztésre kerül, bármely másik szál folytathatja (task)
- *mergeable*: olyan task, amelynek adatkörnyezete megegyezik a generáló task adatkörnyezetével (task)

Könyvtári függvények: célja, hogy a párhuzamos végrehajtás bizonyos paramétereit dinamikusan, a program futása közben tudjuk vezérelni
- párhuzamos futtatás paramétereit lekérdező/beállító függvények
- hozzáférést korlátozó függvények: kölcsönös kizárást megvalósító függvényeket tartalmaz
- időmérést támogató függvények: a párhuzamos végrehajtás monitorozásának egy fontos eszköze

Környezeti változók: a munkamenetek szintjén állíthatjuk be a párhuzamos futtatás egyes paramétereit, programunk újrafordítása és dinamikus paraméterek alkalmazás szintű megjelenése nélkül
- *OMP_SCHEDULE[type, chunk]*:
    - type: static, dynamic, guided vagy auto
    - chunk: a korábban leírtakkal azonos esetekben és jelentéssel jelenhet meg
- *OMP_NUM_THREADS*: klóz nélküli parallel régiókban használt szálak számát állíthatjuk be értékével
- *OMP_NESTED*: egymásba ágyazott parallel régiók párhuzamos végrehajtását kapcsolhatjuk be vele
- *OMP_MAX_ACTIVE_LEVELS*: egymásba ágyazott parallel régiók számát adhatjuk meg
- *OMP_THREAD_LIMIT*: a párhuzamosítás során egyszerre létező szálak maximális számát állíthatjuk be

Pthreads (POSIX Threads): osztott memória modell alapú, szálak létrehozására és menedzselésére használható szabvány, amellyel szálakra épülő párhuzamos programozást valósíthatunk meg

Fordítás: *-pthread* kapcsolóval, illetve az *pthread.h* header fájlt kell include-olni a fájlokban

Hibakezelés: a függvények konvencionálisan, hibakód visszatérési értékekkel jelzik a futásidőben történő hibákat, a hibakódokhoz tartozó nevesített konstansokat és azok rövid magyarázatát az errno.h header fájlban találjuk:
- EAGAIN: “Try again.”
- EINVAL: “Invalid argument.”
- EPERM: “Operation not permitted.”
- ESRCH: “No such process.”
- EDEADLK: “Resource deadlock would occur.”
- ENOMEM: “Out of memory.”
- EBUSY: “Device or resource busy.”
- ETIMEDOUT: “Connection timed out.”

Szálak létrehozása és leállítása: a létrehozandó szálak kódját speciális szignatúrájú függvények formájában kell megírnunk
void szal(void*);*
- szálak létrehozása:
int pthread_create (pthread_t thread, const pthread_attr_t* attr, void*(*start_routine)(void*), void* arg);*
    - thread: kimeneti paraméter, a létrejött szál azonosítója ezen a címen kerül tárolásra
    - attr: a létrehozandó szál tulajdonságai (0 esetén alapértelmezett)
    - start_routine: a függvény neve, amit a szál futtatni fog
    - arg: a létrejövő szál által futtatott függvény ezen mutatót kapja paraméterként
- szálak leállítása:
void pthread_exit (void status);*
    - status: a függvény hívásakor megadott status paramétert kapja meg visszatérési értékként
(szálakat terminálhatunk return-nel is, viszont ekkor nem kapjuk vissza a visszatérési értéket)

Szinkronizálás:
- kapcsolás:
*int pthread_join (pthread_t thread, void** status);*
    - thread: azon szál azonosítója, amelyhez a függvényt hívó szálat kapcsolni szeretnénk
    - status: ezt a paramétert kapja meg terminálláskor
- leválasztás:
int pthread_detach (pthread_t thread);
    - thread: a leválasztandó szál azonosítója
- mutex változók: kölcsönös kizárást valósít meg, a szálak kritikus szekcióba történő belépésének szabályozására használjuk

12. Az OpenCL API eszközei, az OpenCL modell. Fordítás, az OpenCL API függvényei. Hibakezelés.

OpenCL: lehetővé teszi a heterogén környezetben történő platformfüggetlen párhuzamos programozást
- nem csak grafikus kártyák processzorainak programozását támogatja, hanem a legújabb többmagos processzorok is programozhatóak az OpenCL eszközeivel

OpenCL modell:
1. Platform modell:
- az OpenCL platform egy gazdagépből és a hozzá kapcsolódó OpenCL-t támogató hardvereszközökből áll
- egy OpenCL eszköz egy vagy több számítási egységet tartalmaz, amelyeket további feldolgozó egységekre bonthatunk, amelyek saját utasításszámláló regiszterrel rendelkeznek
- egy OpenCL program két szinten fut:
    - a gazdagépen szekvenciálisan futva az adatokat és az OpenCL eszköz(ök) használatát készíti elő
    - az OpenCL eszköz a gazdagéptől kapott, végrehajtható kódot futtatja párhuzamosan a számítási és feldolgozó egységeken

2. Végrehajtási modell: ez írja le, hogy milyen absztrakt adatszerkezeteken keresztül érhető el az OpenCL eszköz és azon milyen stratégia szerint végezhetünk számításokat
- indextartományok és részproblémák: a megoldandó feladatot párhuzamosan végrehajtható egységekre bontjuk 1, 2 vagy 3 dimenziós indextartományok segítségével
- kernel, munkaelem, munkacsoport:
    - kernel: a legkisebb, párhuzamosan végrehajtható kódok, amelyeket a probléma részfeladatainak megoldására használhatunk
    - munkaelem: a kernel egy részfeladatra történő végrehajtása
    - munkacsoport: a munkaelemek nagyobb egységekbe való szervezése
- globális, munkacsoport és lokális indexek: minden munkaelem 3 különböző azonosítóval rendelkezik
    - globális index: a probléma dekompozíciója során létrehozott indextartomány egy eleme
    - munkacsoport index: a munkaelem munkacsoportjának egyedi azonosítója
    - lokális index: a munkaelem munkacsoporton belüli azonosítója
- környezet objektum: a párhuzamos végrehajtáshoz használandó OpenCL eszközök azonosítóit tartalmazza, mivel kernelek végrehajtása mindig valamilyen környezetben történik
- parancssor objektum: a gazdaprogramban létrehozott, feladatok tárolására szánt sor adatszerkezet
    - kernel végrehajtási feladatok: adott eszközön, egy adott indextartomány elemeire
    - memóriakezelő feladatok: memóriaterületek lefoglalása, összerendelése a gazdagép és az eszköz között, ezek olvasása/írása, adatmozgatás
    - szinkronizációs feladatok: a kernelek végrehajtási sorrendjének vezérlése

3. Memória modell: az OpenCL eszközök memória modellje egy többszörösen összetett hibrid modell, amely négy, különböző elérési tulajdonságokkal bíró memóriaterülettel rendelkezik
    - globális memória: a gazdaprogram által elérhető, olvasható és írható memóriaterület
    - konstans memória: a globális memória egy elkülönített része, amely mindig konstans marad
    - lokális memória: egy munkacsoporthoz tartozó munkaelemek megosztott memóriája
    - privát memória: egy munkaelem saját, privát memóriája
- gyorsítótárak: munkacsoport, azaz számoló egység, illetve munkaelem, azaz feldolgozó egység szinten

4. Programozási modell: implementáció során végrehajtandó lépések:
- fel kell térképezni a rendelkezésre álló hardverkörnyezetet, azonosítani kell az OpenCL platformokat és a hozzájuk tartozó eszközöket
- létre kell hozni a párhuzamos végrehajtás során használandó környezet és parancssor objektumokat
- le kell generálni az OpenCL eszközön futó kernel-kódot és le kell fordítani az eszköz(ök)re, továbbá le kell foglalni a bemenő és kimenő adatok tárolására szolgáló puffereket és fel kell tölteni tartalommal
- definiálni kell egy megfelelő indextartományt, majd minden számítási egység megkap egy végrehajtandó munkacsoportot, ahol a munkaelemeket párhuzamosan hajtják végre, egészen addig, amíg minden munkaelem végrehajtásra nem került

- ha szükséges, a végrehajtás befejeztével le kell töltenünk az OpenCL eszköz memóriájából ez eredményeket tartalmazó puffer tartalmát a gazdagép memóriájába

Fordítás: az OpenCL header fájljainak segítségével, illetve szükség van a specifikált függvények implementációit tartalmazó programkönyvtárra, amit az OpenCL eszköz gyártójának feladata biztosítani

OpenCL API függvényei:
Platform réteg: célja egy megfelelő OpenCL környezet létrehozása
- platform azonosítóinak és tulajdonságainak lekérdezése: *clGetPlatformIDs* és *clGetPlatformInfo*
- egy adott platformhoz tartozó eszközök azonosítóinak és tulajdonságainak lekérdezése: *clGetDeviceIDs* és *clGetDeviceInfo*
- környezet és adott típusú környezet létrehozása: *clCreateContext, clCreateContextFromType*
- létrehozott környezet fő tulajdonságainak lekérdezése: *clGetContextInfo*
- egy környezet objektum által lefoglalt erőforrásainak felszabadítása: *clReleaseContext*
Futtató réteg: célja egy adott környezet objektumhoz kapcsolódóan az OpenCL eszközön történő végrehajtások szervezése
- parancssor objektum létrehozása, tulajdonságainak lekérdezése és általa használt erőforrás felszabadítása: *clCreateCommandQueue, clGetCommandQueueInfo, clReleaseCommandQueue*
- puffer objektum létrehozása és az általa használt erőforrás felszabadítása: *clCreateBuffer, clReleaseMemObject*
- puffer írása és olvasása: *clEnqueueReadBuffer* és *clEnqueueWriteBuffer*
- gazdagépen futó program futásának blokkolása: *clWaitForEvents, clFlush, clFinish*
- program objektum létrehozása a forráskódból vagy bináris kódból: *clCreateProgramWithSource, clCreateProgramWithBinary*
- program tulajdonságainak lekérdezése és az általa használt erőforrás felszabadítása: *clGetProgramInfo, clReleaseProgram*
- forráskód egy lépésben történő fordítása és linkelése: *clBuildProgram*
- kernel objektum létrehozása és általa használt erőforrás felszabadítása: *clCreateKernel, clReleaseKernel*
- kernel objektum paramétereinek megadása: *clSetKernelArg*
- munkacsoportok kialakítása: *clEnqueueNDRangeKernel, clEnqueueTask*

Hibakezelés: a függvények konvencionálisan, hibakód visszatérési értékekkel jelzik a futásidőben történő hibákat, a hibakódokhoz tartozó nevesített konstansokat és azok rövid magyarázatát az error.h header fájlban találjuk

13. A Turing gép fogalma, működése. A RAM-gép. Boole-függvények és logikai hálózatok.

Turing gép: azt nevezzük algoritmikusan kiszámíthatónak, ami Turing-gépen kiszámítható
- a számítások legrégibb, legismertebb és matematikai szempontból „legtisztább” modellje (A. Turing 1936)
- memóriáját nem lehet közvetlenül címezni, egy távoli memória-rekesz kiolvasásához minden korábbi rekeszt is el kell olvasni
Egy vezérlőegységből és k db két irányban végtelen szalagból áll ($k \geq 1$), amelyek végtelen sok mezőre vannak osztva. Minden szalagnak van kezdőmezője és minden mezőre lehet írni egy adott Σ ábécéből (vagy * = üres mező). Minden szalaghoz tartozik egy író-olvasófej, ami minden lépésben egy mezőn áll. Ennek a lépéseit a vezérlőegység irányítja: α adja meg az új állapotot, β a szalagokra írt jeleket, γ azt, hogy mennyit lép a fej. A vezérlőegység lehetséges állapotai egy véges Γ halmazt alkotnak (pl. START, STOP)

Működése: Kezdetben a vezérlőegység START állapotban van, és a fejek a szalagok kezdőmezőjén állnak. Minden lépésben minden fej leolvassa a szalagjának az adott mezőjén álló jelet. Ezután a vezérlőegység a leolvasott jelektől és a saját állapotától függően 3 dolgot csinál:
- átmegy egy új állapotba
- minden fejnek utasítást ad, hogy azon a mezőn, melyen áll, a jelet írja felül
- minden fejnek utasítást ad, hogy lépjen jobbra vagy balra egyet, vagy maradjon helyben.
A gép megáll, ha a vezérlőegység a STOP állapotba jut.

RAM-gép: a valódi számítógépek egy leegyszerűsített modellje
- memóriarekeszeit közvetlenül el lehet érni (beleírni vagy belőle kiolvasni), de ehhez előbb azt meg kell címezni és a címet valamelyik másik rekeszben eltárolni
- a memóriarekeszek és a címek száma nem korlátos → a címet tartalmazó rekeszben akármilyen nagy természetes számot előfordulhat
- indirekt címzés: a rekesznek a tartalma maga is változhat a program futása során
A RAM egy programtárból és egy memóriából áll. A memória végtelen sok memóriarekeszből áll, melyek egész számokkal vannak címezve és x[i] egész számot tartalmaznak. A programtár ugyancsak végtelen sok egész számokkal címzett rekeszből, sorból áll. A RAM bemenete egy természetes számokból álló véges sorozat, melyek hosszát az x[0] memóriarekeszbe, elemeit pedig rendre az x[1], x[2], ... memóriarekeszekbe írjuk be. Akkor áll meg, ha olyan programsorhoz ér, melyben nincsen utasítás. A kimeneten az x[i] rekeszek tartalmát értjük.

Boole-függvény: Boole-függvénynek nevezünk egy $f : \{0, 1\}^n \to \{0, 1\}$ leképezést.

x	Hamis	Identikus	Negáció, tagadás NEM, NOT	Igaz
			$\overline{x}$	
x	h	x	$\overline{x}$	i
h	h	h	i	i
i	h	i	h	i

egyváltozós Boole-függvények

x	y	Diszjunkció (VAGY, OR) $x \vee y$	Konjunkció (ÉS, AND) $x \wedge y$	Antivalencia (KIZÁRÓ VAGY, XOR) $x \oplus y$	Ekvivalencia $x \leftrightarrow y$	Implikáció $x \to y$	Peirce nyíl (NEM VAGY, NOR) $x \downarrow y$	Scheffer vonás (NEM ÉS, NAND) $x \mid y$
h	h	h	h	h	i	i	i	i
h	i	i	h	i	h	i	h	i
i	h	i	h	i	h	h	h	i
i	i	i	i	h	i	i	h	h

kétváltozós Boole-függvények

Boole-polinom: diszjunkció, konjunkció és negáció felhasználásával megadható azonosság
- disztributivitás:

$$x \wedge (y \vee z) = (x \wedge y) \vee (x \wedge z),$$
$$x \vee (y \wedge z) = (x \vee y) \wedge (x \vee z),$$
$$x \wedge (y \oplus z) = (x \wedge y) \oplus (x \wedge z).$$

- De Morgan:

$$\overline{x \wedge y} = \overline{x} \vee \overline{y},$$
$$\overline{x \vee y} = \overline{x} \wedge \overline{y}.$$

Literál: egyetlen változóból álló Boole-polinom
Elemi konjunkció: olyan Boole-polinom, mely $\wedge$ művelettel összekapcsolt literálokból áll
Diszjunktív normálforma: olyan Boole-polinom, mely $\vee$ művelettel összekapcsolt elemi konjunkciókból áll

Logikai hálózat: Legyen G egy olyan irányított gráf, mely nem tartalmaz irányított kört.
- bemeneti csúcsok: a gráf azon csúcsai, amelyekbe nem fut be él
- kimeneti csúcsok: a gráf azon csúcsai, amelyekből nem fut ki él
A gráf minden olyan v csúcsához, mely nem bemeneti csúcs, adjunk meg egy $F_v : \{0, 1\}^d \to \{0, 1\}$ Boole-függvényt. A logikai hálózat egy ilyen függvényekkel ellátott irányított gráf.

14. Algoritmikus eldönthetőség. Church-tézis. Rekurzív és rekurzívan felsorolható nyelvek, rekurzív illetve parciálisan rekurzív függvények. Nevezetes nyelvek (Az R, Re, coR, coRE nyelvosztályok és ezek kapcsolata) és bonyolultságuk. Algoritmikusan eldönthetetlen problémák. Polinomiális idejű algoritmusok.

Algoritmikus eldönthetőség:
- egy probléma eldöntésére: halmazelmélet axiómarendszerével
- probléma-sereg eldöntésére: nem létezik olyan algoritmus

Church-tézis: Minden „számítás” az általa megadott rendszerben formalizálható.

Rekurzív nyelvek (R): Legyen $\mathcal{L} \subseteq \Sigma^*_0$ egy nyelv. Az $\mathcal{L}$ nyelvet rekurzívnak hívjuk, ha a karakterisztikus függvénye kiszámítható:

$$f(x) = \begin{cases} 1, & \text{ha } x \in \mathcal{L}; \\ 0, & \text{ha } x \in \Sigma_0^* - \mathcal{L}. \end{cases}$$

Rekurzívan felsorolható nyelvek (RE): Az $\mathcal{L}$ nyelvet rekurzívan felsorolhatónak nevezzük, ha vagy $\mathcal{L} = \varnothing$, vagy van olyan kiszámítható $f : \Sigma^*_0 \to \Sigma^*_0$ függvény, melynek értékkészlete $\mathcal{L}$.

Rekurzív (R) és rekurzívan felsorolható (RE) nyelvek és komplementerük (coR és coRE) kapcsolata:
- minden véges nyelv rekurzív
- **R = coR** : ha az $\mathcal{L}$ nyelv rekurzív, akkor a komplementere: $\Sigma^*_0 - \mathcal{L}$ is az
- egy rekurzívan felsorolható nyelv komplementere nem szükségképpen rekurzívan felsorolható
- **R ⊆ RE** : minden rekurzív nyelv rekurzívan felsorolható
- egy $\mathcal{L}$ nyelv akkor és csak akkor rekurzív, ha mind az $\mathcal{L}$ nyelv, mind a $\Sigma^*_0 - \mathcal{L}$ nyelv rekurzívan felsorolható.

Rekurzív függvény: Legyen Σ egy véges ábécé, mely tartalmazza a „∗” szimbólumot. Egy $f : \Sigma^*_0 \to \Sigma^*_0$ függvényt rekurzívnak nevezünk, ha van olyan T Turing-gép, mely $\forall\ x \in \Sigma^*_0$ bemenettel véges idő után megáll, és az utolsó szalagjára az f(x) szó lesz írva.
Parciálisan rekurzív függvény: Legyen I egy nem üres véges halmaz. Az $f : I^* \to I^*$ parciális függvény parciálisan rekurzív függvény, ha létezik olyan M Turing-gép, hogy $f = f_M$.

Nyelvek bonyolultsága: mivel megszámlálhatóan végtelen sok rekurzív és rekurzívan felsorolható nyelv létezik, így a nyelvek bonyolultsága sem ugyanakkora mértékű
- időigény: Egy T Turing-gép időigénye az a $time_T(n)$ függvény, mely a gép lépésszámának maximumát adja meg n hosszúságú bemenet esetén.
- tárigény: A $space_T(n)$ tárigény-függvényt úgy definiáljuk, mint a gép szalagjain azon különböző mezők maximális számát az n hosszúságú bemenetek esetén, melyekre a gép ír.
- idő- és tárbonyolultság: Egy $\mathcal{L} \subseteq \Sigma^*_0$ nyelv időbonyolultsága legfeljebb f(n), ha a nyelv egy legfeljebb f(n) időigényű Turing-géppel eldönthető. A legfeljebb f(n) időbonyolultságú nyelvek osztályát **DTIME(f(n))**-nel jelöljük. Hasonlóan definiáljuk egy nyelv tárbonyolultságát, és a **DSPACE(f(n))** nyelvosztályokat.

Algoritmikusan eldönthetetlen problémák:
- Dominó-probléma: Tekintsünk egy dominókészletet melyben minden dominó négyzetalakú, és minden oldalára egy természetes szám van írva. Csak véges sok különböző fajta dominónk van, de mindegyikből végtelen sok példány áll rendelkezésre. Ki van tüntetve továbbá egy kezdődominó.
Ki lehet-e rakni ezekkel az adott négyzetekkel a síkot úgy, hogy a kezdődominó kell, hogy szerepeljen, és az egymáshoz illeszkedő oldalakon mindig ugyanaz a szám legyen?
- Diophantoszi egyenlet: Adott egy egész együtthatós n változós $p(x_1, \ldots, x_n)$ polinom, *döntsük el, hogy van-e a p = 0 egyenletnek egész számokból álló megoldása?*

- Csoportok szóproblémája: Adott az $a_1, \ldots, a_n$ szimbólumok által generált szabad csoportban n + 1 szó: $\alpha_1, \ldots, \alpha_n$ és β. *Benne van-e β az $\alpha_1, \ldots, \alpha_n$ által generált részcsoportban?*
- Poliéderek összehúzhatósága: *Összehúzható-e egy adott poliéder (folytonosan, mindig önmagán belül maradva) egy ponttá?*
- Post szóproblémája: A bemenet egy szótár. *Van-e olyan mondat, ami két nyelven ugyanazt jelenti? Azaz van-e az indexeknek olyan $i_1, i_2, \ldots, i_K$ sorozata, hogy $u_{i1} u_{i2} \ldots u_{iK} = v_{i1} v_{i2} \ldots v_{iK}$?*

P bonyolultsági osztály: a polinom időben megoldható konkrét problémák halmaza
Polinomiális idejű algoritmusok:
- kombinatorikai algoritmusok: összefüggőség-teszt, legrövidebb út keresése, maximális folyam keresése, „magyar módszer", Edmonds párosítás algoritmusa, stb
- aritmetikai algoritmusok: egész számok összeadása, kivonása, szorzása, maradékos osztása, két szám nagyság szerinti összehasonlítása, euklideszi algoritmus két természetes szám legnagyobb közös osztójának megkeresésére
- lineáris algebrai algoritmusok: vektorok összeadása, skaláris szorzása, mátrixok szorzása, invertálása, determinánsok kiszámítása

15. Nemdeterminisztikus Turing gépek, Az NP és a coNP nyelvosztály. Példák NP-beli nyelvekre. A tanú-tétel. Nemdeterminisztikus algoritmusok bonyolultsága. NP-teljesség, Cook-tétel. Néhány NP-teljes probléma, Cook-Levin tétel.

Nemdeterminisztikus Turing gép: Egy olyan $T = \langle k, \Sigma, \Gamma, \alpha, \beta, \gamma \rangle$ rendezett 6-os, ahol $k \geq 1$ egy természetes szám, Σ és Γ véges halmazok, $* \in \Sigma$, START, STOP $\in \Gamma$, (eddig ugyanúgy, mint a determinisztikus Turing gépnél), és $\alpha \subseteq (\Gamma \times \Sigma^k) \times \Gamma$, $\beta \subseteq (\Gamma \times \Sigma^k) \times \Sigma^k$, $\gamma \subseteq (\Gamma \times \Sigma^k) \times \{-1, 0, 1\}^k$ tetszőleges relációk.
- ugyanazzal a bemenettel tehát sok különböző legális számolása lehet
- működése: minden lépésben a vezérlőegység új állapotba megy át, a fejek új jeleket írnak a szalagokra, és legfeljebb egyet lépnek jobbra vagy balra
- a nemdeterminisztikus Turing-géppel felismerhető nyelvek pontosan a rekurzívan felsorolható nyelvek

Tanú-tétel: Legyen $\mathcal{L} \in \Sigma^*$ egy nyelv. Ekkor a következő állítások ekvivalensek:
1. $\mathcal{L} \in$ NP.
2. Létezik egy $\mathcal{L}' \subseteq \Sigma$ P-beli nyelv és egy p(x) polinom, amelyekre $\forall w \in \mathcal{L}$, akkor és csak akkor, ha $\exists v \in \Sigma^*$ úgy, hogy $l(v) \leq p(l(w))$ és $w \neq v \in \mathcal{L}'$.

Nemdeterminisztikus algoritmusok bonyolultsága:
- Legyen f jól számolható függvény és $g(n) \geq n$. Ekkor minden $\mathcal{L} \in$ NTIME(f(n)) nyelvnek van O(f(n)) hosszúságú, lineáris idejű tanúja. Ha egy $\mathcal{L}$ nyelvnek van f(n) hosszúságú, g(n) idejű tanúja, akkor $\mathcal{L} \in$ NTIME(g(n+1+f(n))).
- Legyen f jól számolható függvény. Ekkor $\text{NTIME}(f(n)) \subseteq \text{DSPACE}(f(n))$, $\text{NSPACE}(f(n)) \subseteq \cup_{c>0}\text{DTIME}(2^{cf(n)})$ és $\text{NTIME}(f(n)) \subseteq \cup_{c>0}\text{DTIME}(2^{cf(n)})$ teljesül.
Savitch tétele: Ha f(n) jól számolható függvény, és $f(n) \geq \log n$, akkor minden $\mathcal{L} \in \text{NSPACE}(f(n))$ nyelvhez van olyan $c > 0$, hogy $\mathcal{L} \in \text{DSPACE}(cf(n)^2)$.

NP bonyolultsági osztály: olyan nyelvosztály, mely polinomiális algoritmussal bizonyítható és az alábbi tulajdonságokkal rendelkezik tetszőleges $\mathcal{L} \subseteq \Sigma^*_0$ nyelv esetén:
- $\mathcal{L}$ felismerhető nemdeterminisztikus Turing-gépen polinomiális időben
- $\mathcal{L}$-nek van polinomiális hosszúságú és idejű tanúja
coNP nyelvosztály: azon nyelvek halmaza, melyekre fennáll, hogy $\Sigma^*_0 - \mathcal{L} \in$ NP

Példák NP-beli nyelvekre:
- gráf-tulajdonságok: összefüggőség és nem összefüggőség, síkbarajzolhatóság és nem rajzohatóság, teljes párosítás létezése és nem létezése, Hamilton kör létezése, három színnel való színezhetőség
- számelmélet/algebra: összetettség, prímség, korlátos osztó létezése, polinom reducibilitása a racionális test felett, megoldás létezése és nem létezése, egész megoldás létezése

Polinomiális visszavezethetőség: az $\mathcal{L}_1 \subseteq \Sigma^*_1$ nyelv polinomiálisan visszavezethető az $\mathcal{L}_2 \subseteq \Sigma^*_2$ nyelvre, ha van olyan polinomiális időben kiszámítható $f : \Sigma^*_1 \to \Sigma^*_2$ függvény, hogy minden $x \in \Sigma^*_1$ szóra $x \in \mathcal{L}_1$ akkor és csak akkor, ha $f(x) \in \mathcal{L}_2$.

NP teljesség: Egy NP-beli $\mathcal{L}$ nyelvet NP-teljesnek nevezünk, ha minden NP-beli nyelv polinomiálisan visszavezethető $\mathcal{L}$-re.
Következmény: Ha egyetlen NP-teljes nyelvről meg tudnánk mutatni, hogy P-ben van, akkor következne, hogy P=NP.

Jelölje SAT a kielégíthető konjunktív normálformák által alkotott nyelvet.
Cook-tétel (Cook-Levin tétel): A SAT nyelv NP-teljes.

Példák NP-teljes problémákra:
- Lefogási feladat: Adott egy véges S halmaz részhalmazainak egy $\{A_1, \ldots, A_m\}$ rendszere, és egy k természetes szám. *Van-e olyan legfeljebb k elemű részhalmaza S-nek, mely minden A_i-t metsz?*
- Partíció feladat: *Kiválasztható-e olyan $\{A_{i1}, \ldots, A_{ik}\}$ részrendszer, mely az alaphalmaz egy partícióját adja?*
- Független ponthalmaz feladat: Adott egy G gráf és egy k természetes szám. *Van-e G-ben k független pont?*
- Diophantoszi egyenlőtlenségrendszer: Adott egy $Ax \leq b$ egész együtthatós lineáris egyenlőtlenségrendszer, *el akarjuk dönteni, hogy van-e megoldása egész számokban?*
- Részletösszeg probléma: Adottak az $a_1, \ldots, a_k$ és b természetes számok. *Van-e az $\{a_1, \ldots, a_m\}$ halmaznak olyan részhalmaza, melynek az összege b?*
- kombinatorikai problémák: gráfok 3 színnel való színezhetősége, Hamilton kör létezése, pontok diszjunkt háromszögekkel való lefedhetősége

16. A Java programozási nyelv története, alapvető jellemzői. Java platformok, a JDK. Osztálydefiníció. Hivatkozás az osztály elemeire. Példányosítás. Hozzáférési kategóriák és használatuk. A this pszeudó változó. Metódusnév túlterhelés. Konstruktor.

Java programozási nyelv története:
- 1990: James Gosling fejlesztette ki
alapvető szempontjai:
    - objektumorientáltság
    - függetlenség az operációs rendszertől, amelyen fut (többé-kevésbé)
    - olyan kódokat és könyvtárakat tartalmazzon, amelyek elősegítik a hálózati programozást
    - távoli gépeken is képes legyen biztonságosan futni
- 1996: kijött a Java 1.0 a Sun Microsystems által, ahol a Java nyelv összeforrt a Java fordítójával és virtuális gépével és az ezekhez kapcsolódó fejlesztői programcsomaggal (JDK)
- 2007: GNU General Public License (GPL) szabvány alatt nyílt forráskódúvá vált
- 2011: a Sun Microsystems céget felvásárolta az Oracle → Java 1.7
- jelenlegi legfrissebb verzió: Java SE 12

Alapvető jellemzői:
- általános célú, objektumorientált, magasszintű programozási nyelv
- szerepét tekintve 3 célt szolgál: programozási nyelvként, köztes rétegként és platformként is funkcionál
- robosztus és biztonságos
- hordozható, platform-független: a fordítóprogram a forráskódot egy Java bájtkódra fordítja le, majd a amely ezután egy virtuális gépen fut
- interpretált, elosztott, többszálú és dinamikus

Java platformok: Java Standard Edition (**Java SE**)
- általános célú hordozható alkalmazások készítéséhez, telepítéséhez, futtatásához használható
- két része van: programozási interfész (Java API) és a Java virtuális gép (Java Virtual Machine: JVM)
- általános célú csomagok: java.lang, java.io, java.math, java.net, java.text, java.util
Egyéb platformok:
- Java Platform Enterprise Edition (Java EE): szervereken futó programok számára tartalmazza a Java SE könyvtárakat
- Java Platform Micro Edition (Java ME): kevés- és korlátozott erőforrással rendelkező eszközök számára biztosít hitelesített Java API gyűjteményt

Java Development Kit (JDK):
- Java fejlesztőknek készült termék az Oracle fejlesztésében, amely ingyenesen letölthető
- alap Java specifikációkat valósít meg különböző platformokra
- programozási eszközöket és a futtatókörnyezetet (Java Runtime Environment: JRE) tartalmazza
- JDK 1.0, “Oak”: első verziója a Java virtuális gépnek és az osztálykönyvtáraknak
- JDK 1.2, “Java 2”: nyelvi változások
- JDK 12.0: jelenleg elérhető legfrissebb verzió

Osztálydefiníció: Java program osztályok halmaza
[módosítók] class Osztálynév [extends ősosztály] [implements interface1[, interface2, …]] {
    adattagok és metódusok definíciója
}
- egy osztálydefiníció egy teljes fordítási egység, de nem teljes program
- az osztály és a definícióját tartalmazó file neve meg kell egyezzen
- egy osztály adattagjaira, metódusaira az osztályon belül minősítetlenül lehet hivatkozni

Hivatkozás az osztály elemeire:
- osztályon belül névvel
- minden más esetben: objektum.név
- metódusra hivatkozásnál paramétert is kell adni, a hivatkozás metódushívás

Példányosítás: objektum létrehozásának folyamata
[módosítók] Osztálynév objektumnév = new Osztálynév([paraméterek]);
- zárójelben a konstruktornak szánt paraméterek találhatók
- egy osztályból tetszőleges számú objektum példányosítható

Hozzáférési kategóriák: a [módosítók] szintaktikai elem típusai
- nincs: félnyilvános, csak az azonos csomagban lévő elemek érhetik el
- public: nyilvános, bármely csomagban lévő bármely osztályból elérhető
- private: privát, más osztályból nem, de az adott osztály összes példánya számára elérhető
- protected: leszármazottban, az azonos csomagban lévő osztályok vagy leszármazottak érhetik el
Használatuk:
- osztály csak nyilvános vagy félnyilvános lehet

“This” pszeudó változó: metódusokban az aktuális példányra való hivatkozás
- ezen keresztül éri el az adattagot a tagfüggvény
- explicite is használható

Metódusnév túlterhelés: ha egy osztályhoz több azonos nevű metódus is tartozik, akkor a függvény hívásakor a fordító a paraméterlista alapján megnézi, melyik metódust kell meghívni, ha nem talál illeszkedőt vagy több egyformán illeszkedőt talál, hibaüzenetet küld

Konstruktor: objektum inicializálására szolgál
[módosító] Osztálynév([paraméterek]) {
törzs
}
- neve egyezik az osztály nevével
- nincs semmilyen visszatérési értéke
- csak hozzáférési módosítók
- üres return utasítás, de nem kötelező
- nem öröklődik

17. Szemétgyűjtő mechanizmus. A finalize metódus. Csomagok és fordítási egységek. Osztályváltozó, osztály metódus. final minősítésű adattag. Öröklődés. Statikus és dinamikus típus. A protected hozzáférési kategória. Konstruktorok és az öröklődés. final minősítésű osztály.

Szemétgyűjtő mechanizmus: számon tartja, hogy egy objektumra hány referencia hivatkozik
- ha már nincs érvényes referencia, akkor egy külön szálban futó szemétgyűjtő felszabadítja a felesleges objektumhoz tartozó memória területet (pl ha megszűnik maga a változó, megváltozik a változó értéke, a változónak a null értéket adjuk, stb)

Finalize metódus: a szemétgyűjtő mechanizmus által meghívott metódus, melyet felüldefiniálva minden mellékhatást megszüntethetünk amit az objektum okozott
protected void finalize() throws Throwable

Csomagok: az osztályok csomagokban helyezkednek el
- lehetőséget nyújtanak egyfajta strukturálásra
- külön névtérrel rendelkeznek, így elkerülhetőek az egyező típusnevekből eredő problémák
- a hozzáférési kategóriák egyik eszköze
- a csomagszerkezet hierarchikus szerkezetet (fastruktúrát) alkot, ahol az alcsomagok egyenrangúak
- adott csomagban elhelyezkedő típus (osztály vagy interfész) hivatkozása: *csomag.típus*
- egy csomag tartalmát a fordítási egységek adják meg
Fordítási egységek:
- itt találhatók a csomaghoz tartozó kódok, azaz típusok definíciói
- package utasítással lehet deklarálni, hogy az adott fordítási egység melyik csomagnak a része, egyébként névtelen csomaghoz fog tartozni

Osztályváltozó: olyan változó, amely nem egyes példányokhoz, hanem az osztályhoz kapcsolódik
- egy adott osztályváltozóból egy létezik, az osztály minden egyes példánya ezen az egyen osztozik
- deklarációjában a static módosítót kell használni
- hivatkozás: *osztály.osztályváltozó*

Osztálymetódus: olyan metódus, amely nem egyes példányokhoz, hanem az osztályhoz kötődik
- akkor is végrehajthatók, ha az osztálynak nincsenek példányai
- csak az osztályváltozókhoz férhet hozzá
- deklarációjában a static módosítót kell használni
- hivatkozás: osztálynévvel is lehetséges
- osztálymetódus a main metódus is

Final minősítésű adattag:
final típus azonosító = inicializáló kifejezés
- konstans definiálás, nem változtatható meg az értéke később
- minden példányosítás során végrehajtódik
- static final minősítés: csak inicializáláskor hajtódik végre
Final minősítésű osztály: nem terjeszthető ki, azaz nem lehet szülőosztály

Öröklődés: egy osztály deklarálható más létező osztály leszármazottjaként
public class Osztálynév extends Szülőosztály {
….
}
- ilyenkor az osztály örökli a szülője adatait, metódusait, de lehet újakat is definiálni vagy örökölt metódusokat módosítani

Statikus és dinamikus típus:
- statikus típus: amit a deklarációban megadtunk
- dinamikus típus: általa éppen hivatkozott objektum tényleges típusa

Protected hozzáférési kategória: leszármazottban elérhető
- leszármazottra vonatkozó speciális minősítő, félnyilvános kategória kiterjesztése
- az azonos csomagban deklarált osztályok elérhetik, ezenkívül csak az adott osztály leszármazottai számára elérhető.

Konstruktorok és az öröklődés:
- konstruktorok nem öröklődnek
- ugyanazon osztályának másik konstruktorának meghívása: *(this(paraméterek))*
- ősosztály konstruktorának meghívása: *(super(paraméterek))*
- ha egy konstruktor nem hív meg explicite más konstruktort, akkor egy implicit super() hívással kezdődik a konstruktor végrehajtása

18. A metódus felüldefinálás - alapszabályok. A metódus felüldefinálás, mint a polimorfizmus implementációja. A final minősítésű metódus. Absztrakt osztályok és metódusok. Az interface és az instanceof operátor. Kivételkezelés.

Metódus felüldefiniálás: egy leszármazott osztály az ősosztálytól örökölt metódust felüldefiniálhatja
- explicite kérni kell a *virtual* vagy *override* módosítókkal
Alapszabályok:
- visszatérési típusának, nevének, és szignatúrájának meg kell egyeznie az eredeti metódusával
- csak olyan ellenőrzött kivételeket válthat ki, amilyeneket az eredeti is kiválthat, azaz definiálnia kell minden olyan kivételosztályt vagy annak ősét, amit az eredeti definiál
- hozzáférési kategóriája nem lehet szűkebb az eredeti metódusénál

Polimorfizmus implementációként: ilyen metódus hívásánál el kell dönteni, hogy az örökölt vagy a saját változat hívódjon meg
- a döntés alapja a hivatkozás dinamikus típusa, viszont ez fordítási időben nem ismert
- késői kötés: futásidőben dől el, hogy melyik metódus hívódik meg: a szülő vagy a gyermek metódusa

Final minősítésű metódus: nem definiálható felül

Absztrakt osztályok és metódusok:
- absztrakt osztály: ha van legalább egy absztrakt metódusa
    - nem lehet példányosítani
    - arra szolgál, hogy ősosztálya legyen további osztályoknak
- absztrakt metódus: ha nincs törzse, azt majd csak a felüldefiniálás során kap

protected abstract int akarmi();

Interface operátor:
- interfész: azon nyilvános elemeinek összessége, ami a használatához szükséges
- olyan típus, amelyben csak absztrakt metódusok és konstansok szerepelnek
- nem példányosítható, hiszen nem tartalmaz definíciókat és végrehajtható kódot

```
módosító interface Név [extends ősinterfacek] {
    konstansok, absztrakt metódusok
}
```

Instanceof operátor: ezzel lehet megállapítani, hogy egy változó dinamikus típusa leszármazottja-e egy másik típusnak

Kivételkezelés: hibák kezelésének mechanizmusa
- folyamata: Amikor egy metódus futása során valamilyen hiba lép fel, akkor egy kivételobjektum jön létre a program aktuális állapotával és a kivétel fajtájával. A kivétel kiváltását (*throw kivételobjektum;*) követően a kivétel kezelését egy megfelelő típusú kivételkezelő blokk végzi, melynek megtalálását a kivétel elkapásának nevezzük. Ezután a kivételkezelő blokk utáni utasításon folytatódik a végrehajtás.

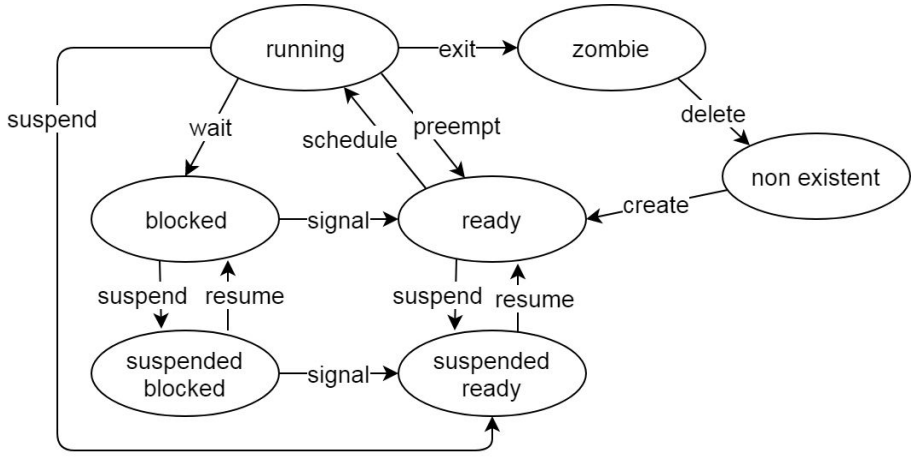
```
try {
    utasítások, amelyekben kivétel megdobódhat
} catch (kivétel) {
    utasítások
}
```

19. Az operációs rendszerbeli folyamat (processz) fogalom. Folyamat kontextus. A fonál fogalom. A processz állapotok, állapotváltások, processz futási módok. Taszk és fonál állapotok, állapotátmenetek.

Processz: egy végrehajtási példánya egy párhuzamosságot nem tartalmazó végrehajtható programnak
Processz kontextus: adatstruktúrákba rendezve minden olyan információ, ami a folyamat futásához szükséges, tehát minden olyan információ, ami a rendszer számára szükséges, hogy a CPU-t a folyamatok között kapcsolja, a folyamatok szekvencialitásának illúzióját biztosítva
Tartalma: a program kódszegmensei és adatszekciói, a processz veremtárai, regiszter állapotok és a processz menedzselési információi
Adatstruktúrája: processz tábla: a kernel által kezelt, adott méretű táblázat, melynek bejegyzésein keresztül minden információt elérhetünk a processzről (a processz menedzselési információit azonosító információkat tartalmazza)

Fonál (lightweight process: LWP): processzek párhuzamosítása esetén
- a CPU használat alapegysége, egy szekvenciálisan futó instrukciósorozat
- van dinamikus kontextusa (regiszterkészlete, verme)
- osztozik más fonalakkal egy taszk statikus kontextusán (címkészlet: adat, kód; stb.)
- valódi vagy pszeudó párhuzamosságban futhatnak
- egyetlen fonalat tartalmazó taszk: klasszikus processz
- egy taszkban a végrehajtandó kódot a fonalak tartalmazzák

Processz állapotok, állapotváltások: az ellipszisek az állapotok, a nyilak az állapotváltások

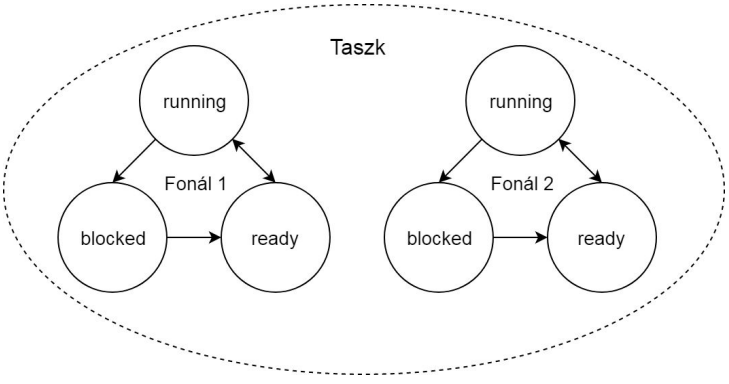


Állapotok: több processz versenyzik a CPU-ért, eközben különböző állapotokban vannak
- running (futó): a processzé a CPU
- ready (futásra kész): a processz számára minden erőforrás rendelkezésre áll, kivéve a CPU
- blocked (blokkolt): a processz valamilyen egyéb erőforrást igényel
- zombie: a processz befejeződött, de még nem törlődött
- non existent (nem létező): a processz nem létezik
- suspended blocked/ready (felfüggesztett állapotok): átmenetileg ki vannak vonva a CPU-ért való versengésből

Állapotváltások:
- wait (vár): maga a processz okozza azzal, hogy bejelenti igényét valamilyen erőforrásra, amely nem áll rendelkezésére, ezért blokkolódik
- signal: egy szignál érkezése okozza, amely jelzi, hogy megszűnt a blokkolódást kiváltó ok
- schedule (ütemezés): a processz megkapja a CPU-t az ütemezőtől
- preempt (elvétel): az ütemező elveszi a processztől a CPU-t, annak ellenére, hogy még használná

Processz futási módok:
Mivel egy processznek van saját, dedikált processzora, e processzor futási módja nevezhető a processz futási módjának. A processzek a saját kódjukat felhasználói módban hajtják végre, a rendszerhívások kiszolgáló kódok, az események kezelőinek kódjai pedig kernel módban futnak.

Taszk és fonál állapotok, állapotátmenetek:



- a taszknak nincs állapota, csak a fonalaknak van
- a fonalak ütemezését az OS scheduler végzi
- a kernel nem tud a fonalakról
- vannak taszk és fonál állapotok is
- running: mikor egy fonál blokkolódik, a taszk kiválaszt másikat és annak adja a CPU-t
- ready: legalább egy fonál futásra-kész
- blocked: nincs kész fonál

20. A memória menedzselés feladatai. Memória, mint erőforrás. Címleképzés fajták. Lapozós virtuális memória menedzselés működése. Allokálás, nyilvántartás, címleképzés. Laphiba fogalma, kezelése, kilapozási stratégiák. Előnyök, hátrányok.

Memória: egy erőforrás amire a processznek szüksége van
- egy véges mennyiségű több példányos erőforrás, ahol egy cellát egyszerre egy időben csak egy processz használhat
- akár el is vehető a processztől és gond nélkül vissza is adható
- Memory Management (MM): erőforrás kezelés egy külön operációs rendszer komponens segítségével

A memória menedzselés folyamatai:
1. Memória foglalások kezelése:
- lefoglalni a processz számára szükséges memória területeket (statikus, dinamikus)
- nyilvántartani a processz számára lefoglalt és a szabad területeket
- biztosítani lehetőséget az extrém használatra is
2. Memória területek védelme: egyik processz se férhessen hozzá egy másik processzhez rendelt memória területhez
3. Osztott memória használat biztosítása: processzek használhassanak igény esetén közös memória területeket (kódmegosztás, adatmegosztás)

Címleképzés (címkötődés): a logikai (virtuális) cím átalakítása valós fizikai címmé
1. Valós címzésű rendszerek: a program készítéskor meghatározott címekre van szüksége a processznek, amíg az nem szabad, nem tud futni, tehát kötődik az adott címtartományhoz, nem helyezhető át
2. Relatív címzésű rendszerek: a program készítésekor relatív címek keletkeznek, melyek egy bázis címhez képesti eltolt címek
- a processzek áthelyezhetőek egy másik szabad, egybefüggő területre a memóriában
- minden címhivatkozás leképzésre szorul: **dinamikus címleképzés**
    - fix méretű leképzett memóriablokk: lapozós rendszerek
    - változó méretű leképzett memóriablokk: szegmentálós rendszerek
3. Virtuális címzésű rendszerek: a programok saját virtuális címtartománnyal rendelkeznek, melyek darabokban vannak elhelyezve bármilyen tároló eszközön (nem csak a memóriában), és a nyilvántartásban csak a darabok kezdetét tároljuk, ezáltal akár a fizikai memória címtartományától nagyobb területet is használhatnak
- virtuális címeket használnak
- minden címhivatkozás leképzésre szorul: **virtuális címleképzés**
    - lapozásos
    - szegmentálásos
    - vegyes

Lapozásos virtuális memória menedzselés: virtuális címzésű rendszereknél
Működése:
- allokálás: a virtuális memóriát azonos méretű blokkokra (lapokra) osztjuk → a fizikai memória is ilyen méretű lapkereteket tartalmaz és a lapokat bármelyik szabad lapkeretbe elhelyezhetjük (vagy akár a háttértárolón kialakított területen is)
- címleképzés: az OS minden processzhez létrehoz egy laptáblát, amely tartalmazza a processz lapjainak tényleges elhelyezkedését
    - a laptábla kezdőcíme a processz kontextusából elérhető
    - a laptáblának annyi bejegyzése van, ahány lapra feloszthető a processz memória területe
- nyilvántartás: minden lapnak külön bejegyzése van, amely tartalmazza, hogy a lap a memóriában van-e vagy sem (valid/invalid), ha igen, a helyét a memóriában (lapkeret cím: f) és a háttértáron, illetve egyéb adatokat

Laphiba: ha a hivatkozott lap nincs betöltve lapkeretbe
Kezelése:
1. Laphiba esemény generálódik és a processz blokkolódik.
2. Az esemény lekezelője (pager) betölti a kívánt lapot. Ha van a processznek üres lapkerete akkor oda, ha nincs akkor helyet csinál (kilapozás).
3. A processz visszakapja a futási jogot és ha futásra kerül, akkor a laphibát okozó utasítás ismételt végrehajtásával folytatja.

Kilapozási stratégiák: a cél azt a lapot kilapozni, amelyikre a legkésőbb lesz szükség
- lokális kilapozás: csak a processz saját lapkeret készletéből lapozhat ki
- globális kilapozás: más processzek lapkeret készletéből is kilapozhat
1. FIFO (First In First Out): a legrégebben belapozottat lapozza ki először (belapozási sor: láncolt lista)
- a régóta bennlévő, de gyakran használt lapok kilapozódnak
2. Második esélyes FIFO: belapozási sor (körkörös láncolt lista) + minden laphoz egy hivatkozás bit (ha hivatkoznak egy lapra 1-re áll)
- kilapozáskor, ha a sor elején levő lapnak (legrégebbi lapnak) 0 a hivatkozás bitje akkor kilapozódik, ha 1, akkor 0 lesz és a sor végére rakja
3. LRU (Least Recently Used): a legrégebben nem használtat lapozza ki (használat ideje alapján sor), ha egy lapot használnak (hivatkoznak rá) sor végére kerül
- viszonylag költséges algoritmus
4. NRU (Not Recently Used): nem túl gyakran használtat lapozza ki
a) 8 biten nyilvántartott: adott referencia byte léptetése jobbra intervallumonként, hivatkozott lapnál 1, nem hivatkozottnál 0 lép be → kisebb a bájt a kilapozásra esélyes lapoknál
b) módosítás bites: minden laphoz egy R és egy M bit, kilapozási sorrend: (R M): (0 0), (1 0), (0 1), (1 1)
- M bit jelzi, hogy módosították-e a lap tartalmát
- R bit hivatkozáskor 1-re, bizonyos időnként 0-ra, azaz jelzi, hogy mostanában hivatkoztak-e a lapra
5. LFU (Least Frequently Used): a legritkábban használtat lapozza ki
- nyilván kell tartani a hivatkozások számát → költséges
- a korábban gyakran használt, de már nem használt is sokáig benn marad
Előnye:
- nincs külső/belső elaprózódás
- könnyű elhelyezés, bármelyik lap bármelyik lapkeretbe
- nagyon nagy memóriaigényű processzek is kezelhetők
- nincs szükség se overlayre, se swappingre
Hátránya:
- címképzés ideje megnő
- nyilvántartás mérete megnő

21. Fájlrendszer megvalósítási feladatok. Jegyzékszerkezetek. Szabad blokk menedzselési lehetőségek. Fájl attribútum rögzítési lehetőségek (ezen belül fájl testet képező blokkok rögzítési lehetőségei).

Fájlrendszer megvalósítási feladatok:
1. Adott fájlhoz blokkok hozzárendelése
2. Az eszközön a szabad blokkok menedzselése (keresni szabad blokkokat, elengedni blokkokat)
3. A fájl egyéb attribútumainak tárolása, jegyzékstruktúra kialakítása

Szabad blokk menedzselési lehetőségek:
- bittérképes (bitmap): kötött helyen a bit-térkép, egy-az-egyes megfeleltetés egy bit és a blokkok között (ha egy bit bebillentett, azt jelenti, hogy a hozzá tartozó blokk foglalt)
- láncolt listás: kötött helyről indulva egy blokk mutatókat tartalmaz szabad blokkokra és egy mutatója további olyan blokkra mutat, ami szintén szabad blokkok mutatóit tartalmazza
    - nehezebben módosítható, mint a bittérképes, de gyorsabb a keresés

Jegyzékszerkezetek: információt adnak a fájlok csoportosítására
- a jegyzék is fájl: kötött vagy változó hosszú lehet, tartalmazhat: bejegyzések neveit, i-node számot vagy egyéb adatokat (kezdő címet, hosszat, attribútumokat, stb.)
- fájl megtalálása történhet abszolút útvonallal vagy relatív útvonallal megadva
- gyorsítás: gyökér- és munkajegyzék tartalma legyen mindig betöltve a memóriába

Fájl attribútum rögzítési lehetőségek: blokk allokációs módszerek
- folytonos allokáció: a fájl blokkjai egymás után, folyamatosan, csak a kezdő blokk cím és hossz tárolására van szükség
    - egyszerű a nyilvántartás
    - nagy a töredezettség
- láncolt listás allokáció: elhelyezés több darabban, nem feltétlenül folytonosan, a jegyzékben a név mellett az első blokk címe, adott blokkban mutató a következő blokkra
    - nincs töredezettség
    - lassú: hosszú fájloknál végig kell menni az egész listán
- láncolt listás allokáció index táblával: ugyanúgy, mint a láncolt listásnál, csak a mutató egy **indextábl**ába kerül: benne a partíció minden blokkjához tartozó bejegyzés, amely tartalmazza a neki megfelelő blokk folytatásának helyét
    - nincs töredezettség
    - index tábla mérete sok helyet foglalhat, hosszú fájloknál végig kell menni az egész listán
    - sérülékenység: dupla index tábla
- i-node módszer: elhelyezés blokkokra osztva a partíció tetszőleges blokkjaiba, minden fájlhoz egy táblázat arról, hogy mely blokkja, mely blokkban van elhelyezve (többszintű táblázat)
    - nincs töredezettség
    - gyorsabb és kevésbé sérülékeny, mint a láncolt listás

22. Relációs adatmodell, relációs struktúra és integritási feltételek. Relációs adatmodell műveleti része, relációs algebra.

Relációs adatmodell (Codd 1970): A reláció egy sora egy egyedet reprezentál, az egyes oszlopokban az adott egyed tulajdonságai szerepelnek. A reláció helyett a tábla vagy táblázat, a sor helyett a rekord, az oszlop helyett pedig a tulajdonság elnevezés használatos. Egy elemi adatot mezőnek nevezünk.
Adatbázis adatmodellek komponensei: relációs adatmodell is adatbázis adatmodell
- strukturális rész: az adatbázis felépítése (adatok, köztük lévő kapcsolatok)
- integritási rész: az adatokra és a műveletekre vonatkozó szabályok
- műveleti rész: milyen műveletek végezhetők és hogyan

Relációs struktúra:
- domain (mezőtípus): értelmezési tartomány, mely megadja az elemhez tartozó értékkészletet, és meghatározza a végrehajtható műveletek körét (pl. char, number, date)
- mező: az adatbázis struktúra legkisebb, atomi egysége, melyből a rekordok felépülnek.

megadásánál meg kell adni a típust (domain) és az integritási feltételeket
- rekord: az adatbázis struktúra azon eleme, amely a logikailag összetartozó, egységként kezelhető elemi adatértékek együttesét jelöli, a mezősorrend rögzített, köthetők hozzá integritási feltételek
- kulcs: a keresés és azonosítás szempontjából meghatározó mező vagy mezőcsoport, amely egy rekord azonosítására szolgál, azaz értéke nem ismétlődik és egyetlen egy rekordban sem lehet üres az értéke
fontosabb típusai: elsődleges kulcs, jelölt kulcs, idegen kulcs, szuper kulcs, index kulcs
- index: az állomány rekordjainak kulcsértékét és a rekord pozíciót tároló szerkezet, melyben a bejegyzések kulcsérték szerinti sorrendben helyezkednek el, gyors keresést lehetővé téve
- reláció: egymáshoz hasonló egyedek bizonyos tulajdonságait leíró táblázat, azonos szerkezetű rekordok halmaza

Integritási feltételek: az adatok érvényességét, megbízhatóságát, helyességét biztosítják
- adatintegritás: az adatok érvényességét, jóságát jelenti

Mező szintű integritás: egy mezőre vonatkozó érvényes érték előfordulások körét lehet megadni
- logikai kifejezés, amely minden lehetséges értékre igaz vagy hamis értéket ad vissza: *Check*
- a mezőben tárolt érték nem lehet üres, kötelező megadni: *Not Null*
- megadható sablon, mely az adat külalakjára vonatkozik

Rekord szintű integritás: egy teljes rekord elfogadhatóságát kell eldönteni
- az egy rekordon belül egymáshoz kapcsolódó mezők értékeinek vizsgálata: *Check*

Reláció szintű integritás: a teljes relációt, vagyis az összes rekord előfordulást át kell vizsgálni
- az adott mezőben ugyanaz az érték nem fordulhat elő többször a relációban (egyediség): *Unique*
- elsődleges kulcs mező: *Primary key*

Adatbázis szintű integritás: a feltétel több relációban, szétszórtan elhelyezkedő mezőkre vonatkozik, az ellenőrzéshez több reláció adatait is át kell olvasni
- idegen kulcs mező (csak egy másik táblában szereplő értékeket vehet fel): *Foreign key*

Hivatkozás integritási szabály: Minden kapcsoló kulcs mező értéke vagy üres, vagy egy létező, hivatkozott táblabeli elsődleges kulcsértékre mutat.

Relációs adatmodell műveleti része: a relációkon alapul, azaz a műveletek relációkon értelmezettek
- adatkezelő utasítások (adatok felvitele, módosítása, törlése)
- adatok lekérdezése: relációs algebrával

Relációs algebra: adatlekérdező műveletcsoport
- a relációkat halmaznak tekintjük, melynek elemei a relációk sorai
- a relációs adatbázisban a lekérdezések matematikai alapját képezi
- halmazorientált, algebrai eszközökkel dolgozik, de csak néhány egyszerű műveletet alkalmaz

Műveletei:
1. egyoperandusú:
- szelekció: a megadott feltételnek eleget tevő rekordok kerülnek át az eredmény relációba, jele: $\boldsymbol{\sigma}_{\text{felt}}(r)$
- projekció: csak a kijelölt mezők jelennek meg az eredmény relációban, jele: $\boldsymbol{\pi}_{\text{mlista}}(r)$
- kiterjesztés: a reláció kibővítése származtatott mezőkkel, jele: $\boldsymbol{\varepsilon}_{\text{mezőlista}}(r)$
- aggregáció: a relációból összesítő rekordot állít elő, jele: $\boldsymbol{\Gamma}^{\text{aggregációs-lista}}(r)$
- csoportképzés: a reláció rekordjait csoportokba rendezi és minden csoportra egy összesítő rekordot állít elő, jele: $\boldsymbol{\Gamma}^{\text{aggregációs-lista}}_{\text{csoport-képző}}(r)$
- átnevezés: reláció nevének vagy oszlop nevének a megváltoztatása, jele: ρ

2. kétoperandusú (halmazműveletek):
- join: táblák összekapcsolására alkalmas műveletek, ha két relációból akarunk egyszerre adatokat lekérdezni
    - cross join: descartes szorzat, jele: r1 x r2
    - theta join: feltételes illesztés, jele: $r1 \rhd\lhd_{\text{felt}} r2$
    - equijoin: egyenlőség alapú illesztés, jele: $r1 \rhd\lhd_{=\text{mező}} r2$
    - natural join: természetes illesztés, jele: $r1 \rhd\lhd r2$ vagy r1 * r2

- semijoin: félig összekapcsolás, jele: r1 ⋉ r2 (baloldali) vagy r1 ⋊ r2 (jobboldali)
- antijoin: pár nélküli rekordok, jele: r1 ▷ r2
- outer join: külső illesztés, jele: r1 $\rhd\lhd^{+}_{\text{feltétel}}$ r2

- metszet: azonos sémájú relációk rekordhalmazának metszete, jele: r1 ∩ r2
- unió: azonos sémájú relációk rekordhalmazának egyesítése, jele: r1 ∪ r2
- különbség: azonos sémájú relációk rekordhalmazának különbsége, jele: r1 \ r2
- osztás: Descartes szorzat inverze, jele: r1 ÷ r2

23. Az SQL szabvány relációs kezelő nyelv bemutatása, a DDL, DML és a SELECT utasítások használata. Az SQL92 szabvány további elemei.

SQL: a relációs adatbázis-kezelők szabványosított adatmanipulációs és lekérdező nyelve
- parancsnyelv jellegű, megfogalmazhatjuk, mit akarunk csinálni, de nem algoritmikus, változó nincs, csak tábla- és oszlopnevekre lehet hivatkozni
- logikai műveletek: AND, OR, NOT
- mintaillesztéses, halmazorientált

Területei:
- DDL (Data Definition Language): adatstruktúra definiáló utasítások, adatbázisok és táblák létrehozása, módosítása és törlése
- DML (Data Manipulation Language): adatokon műveletet végző utasítások, adatok rögzítése, módosítása, törlése
- DQL (Data Query Language): adat lekérdező utasítás
- DCL (Data Control Language): adatvezérlő, felügyelő utasítások, tranzakciók kezelése, jogosultságok menedzselése

DDL utasítások: adatstruktúra definiáló utasítások
- CREATE: objektum létrehozás

CREATE objektumtípus objektumnév paraméterek;
- DROP: objektum megszüntetés

DROP objektumtípus objektumnév;
- ALTER: objektum módosítás (új objektum felvétele, objektum törlése, objektum módosítása)

ALTER TABLE táblanév [ADD (újelem, ..., újelem)] [MODIFY (módosítás, ..., módosítás)] [DROP (oszlop, ..., oszlop)];
- TRUNCATE: tábla tartalmának törlése

TRUNCATE TABLE táblanév;

DML utasítások: adatokon műveletet végző utasítások
- INSERT: rekord felvitel

INSERT INTO táblanév [(oszloplista)] VALUES (értéklista);
- DELETE: rekord törlés

DELETE FROM táblanév [WHERE feltétel];
- UPDATE: rekord módosítás

UPDATE táblanév SET mező = kifejezés, ..., mező = kifejezés [WHERE feltétel];

DQL utasítások: adat lekérdező utasítás: SELECT
- egy vagy több adattáblából egy eredménytáblát állít elő, amely a képernyőn listázásra kerül, vagy más módon használható fel

SELECT [DISTINCT] oszloplista FROM táblanévlista [WHERE feltétel];

- ORDER BY: rendezés lekérdezésben
- GROUP BY: csoportképzés
- HAVING: csoportok szűrése

DCL utasítások: adatvezérlés, jogosultságok
- GRANT: jogosultság adása (tovább is adható)
GRANT művelet ON objektum TO felhasználó | PUBLIC [WITH GRANT OPTION];
- REVOKE: jogosultság visszavonása
REVOKE művelet ON objektum FROM felhasználó | PUBLIC;

Az SQL92 szabvány további elemei: JOIN: táblák összekapcsolása
- CROSS JOIN: Descartes szorzat
SELECT mezőlista FROM táblanév1, táblanév2;
- INNER JOIN: egyenlőség alapú illesztés
SELECT mezőlista FROM tábla1 INNER JOIN tábla2 ON join_feltétel WHERE feltétel;
- NATURAL JOIN: természetes illesztés
SELECT mezőlista FROM tábla1 NATURAL JOIN tábla2;
- OUTER JOIN: külső illesztés
SELECT …FROM tábla1 [LEFT | RIGHT | FULL] OUTER JOIN tábla2 ON join_feltétel [WHERE feltétel] …;
- SELF JOIN: önillesztés
SELECT …FROM tábla t1, tábla t2 [WHERE feltétel] …;

24. Adatkezelés és adatbáziskezelés alapfogalmai, fileszervezési módszerek, B-fa index; adatbázis architektúra; Adatmodellek, SDM modellek áttekintése, ER modell, konverzió és normalizálás.

Adatbázis: egy integrált adatrendszer, mely több különböző egyed előfordulásainak adatait adatmodell szerinti struktúrában, perzisztens (tartós, állandósult) módon tárolja a kapcsolat leíró elemek mellett a meta adatokkal együtt, melyek a hatékonyság, integritásőrzés, az adatvédelem biztosítását szolgálják
Adatbázis-kezelő rendszer: olyan programrendszer, amelynek feladata az adatbázishoz történő szabályozott hozzáférés biztosítása és az adatbázis belső karbantartási műveleteinek végrehajtása
Adatbázis-kezelő rendszer előnyei:
- információ kinyerés, lekérdezések
- program-adat függetlenség
- minimális redundancia (redundancia: fölösleges adatismétlés)
- adatok integritása: az adatok érvényessége, jósága
- adatok megosztott elérése
- gyorsabb alkalmazás-fejlesztés
- többféle hozzáférési mód
- fokozott biztonság: belépéskor minden felhasználónak azonosítania kell magát

Fileszervezési módszerek: rekordok elérése fájlokban
- fizikai szekvenciális: fizikai sorrend azonos a logikaival
- logikai szekvenciális: nincs logikai kapcsolat a rekord azonosítója és az elhelyezés fizikai címe közt
    - teljes listaszerkezet
    - indextábla
- indexszekvenciális: fizikailag folytonos, sorrend szerinti tárolás
    - B-fa
    - bitmap

B-fa (bináris fa) index: csomópontok halmaza, rekord, a rekordon belül két pointer a gyerekre mutat
- csomóponton belül rendezett
- gyorsítja a kulcs szerinti keresést
- egy táblához több index-fa is létrehozható

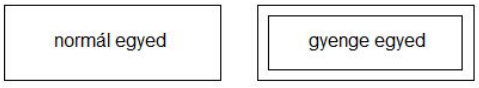
CREATE INDEX indexnév ON tábla(mezőkifejezés);
DROP INDEX indexnév;
- keresés: a gyökértől indulunk ki és úgy haladunk lefelé a fában úgy, hogy a keresett kulcsot összehasonlítjuk az érintett kulcsokban tárolt kulcsokkal

Adatbázis architektúra: absztrakciós szintek:
- külső szint: amit egy felhasználó lát az adatbázisból
- fogalmi szint: az adatbázis modellje, struktúrája, meghatározza, hogy az adatokat hogyan kell értelmezni
- fizikai szint: az adatok elhelyezkedése a háttértárakon

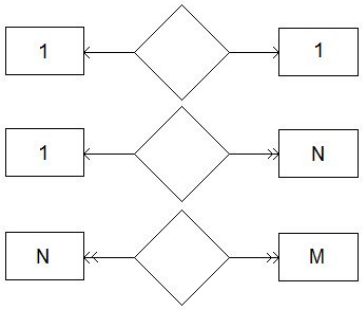
Adatmodellek: az adatok és az azok közötti összefüggések leírására szolgál
Az adatbázis kezelő rendszer legalapvetőbb tulajdonságait rögzíti, meghatározza, hogy az adatbázisban az adatokat milyen szerkezetben tároljuk és milyen mechanizmusokon keresztül lehessen hozzájuk férni.
    - szemantikai adatmodellek (SDM): ER, EER, IFO, UML
    - adatbázis adatmodellek: hierarchikus, hálós, relációs, objektum-orientált
- Strukturális komponens: az adatbázis felépítése (adatok, köztük lévő kapcsolatok)
- Integritási komponens: az adatokra és a műveletekre vonatkozó szabályok
- Műveleti komponens: milyen műveletek végezhetők és hogyan

Szemantikai adatmodellek (SDM): az adatbázis felépítésének (adatok, köztük lévő kapcsolatok) leírására szolgálnak (strukturális komponens)

ER modell: egyed-kapcsolat modell
Egyed: egy a külvilág többi részétől egyértelműen megkülönböztetett, önálló léttel bíró dolog, amiről az információkat tárolni kívánjuk
- normál egyed
- gyenge egyed

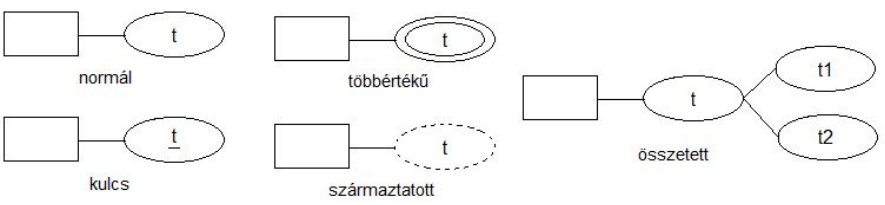


Kapcsolat: az egyedek között fennálló ideiglenes vagy tartós asszociáció, ahol csak az elsődleges kapcsolatokat adjuk meg



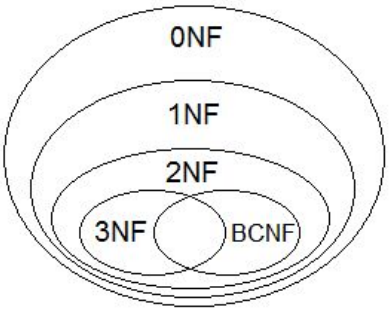
- 1:1: egy egyed-előforduláshoz maximum egy egyed társul a kapcsolatban, mindkét viszonylatban
- 1:N (egy-több): egy egyed-előforduláshoz több egyed társulhat, de a másik irányban csak egy kapcsolódó egyed-előfordulás létezhet
- N:M (több-több): mindkét irányban több kapcsolódó előfordulás létezhet

Tulajdonság: az egyedeket és a kapcsolatokat jellemző mennyiség, a letárolandó információelemeket tartalmazza
- normál: egyértékű
- kulcs: azonosító szerepű
- többértékű: több értéke is lehet
- származtatott: értéke kiszámítható
- összetett: több tagból áll



Konverzió: ER modell konverziója relációs modellre
- Egyed → Reláció
    - normál egyed → reláció kulcs mezővel
    - gyenge egyed → reláció kulcs mező nélkül
- Tulajdonság → Mező
    - elemi → mező
    - kulcs → kulcs mező
    - összetett → több mezőre bontjuk szét
    - többértékű → külön relációba kerül
    - származtatott → csak a képletet tároljuk
- Kapcsolatok → Kapcsolatok
    - 1:1 → egyedi kapcsoló kulcs
    - 1:N → kapcsoló kulcs
    - N:M → kapcsoló tábla

Normalizálás: az adatbázis belső szerkezetének ellenőrzése, lépésenkénti átalakítása oly módon, hogy az adatbázis minden egyes lépésben egy-egy újabb kritériumnak, egymásra épülő normálformának feleljen meg, célja a redundancia- és anomáliamentes adatbázisok kialakítása (3NF vagy BCNF)



- 1NF (első normálforma): ha minden attribútum egyértékű, és a relációban van kulcs
- 2NF (második normálforma): ha 1NF teljesül, és minden attribútum a teljes kulcstól függ
- 3NF (harmadik normálforma): ha 2NF teljesül, és a nem kulcs attribútumok nem függnek tranzitíven a kulcstól
- BCNF (Boyce-Codd normálforma): ha minden függőség csak jelölt kulcsból indul ki

25. Számítógép-hálózatokhoz kötődő alapfogalmak és az ISO-OSI hivatkozási modell: Topológia és méret szerinti osztályozásuk. Kapcsolástechnika szerinti osztályozásuk (vonalkapcsolás, üzenetkapcsolás, csomagkapcsolás és virtuális vonalkapcsolás). Az ISO-OSI hivatkozási modell szerkezete, rétegei és azok főbb funkciói.

Számítógéphálózat: autonóm számítógépek összekapcsolt hálózata
- nem alárendelt kapcsolat (pl. számítógép-periféria)
- tetszőleges kommunikációs alrendszer kötheti össze őket (elektronikus információcsere)

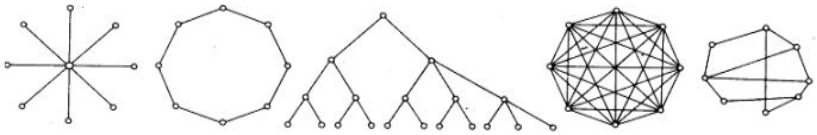
Elosztott rendszer: (a hálózattal szemben) egyetlen virtuális rendszer, melynek elemei együttműködnek egy feladat megvalósítása érdekében
- az egyes elemek konkrét helye, funkciói el vannak rejtve
- implementálható számítógép-hálózatra is

Hálózat célja:
- erőforrás összevonás/megosztás: minden erőforrás a fizikai helyétől függetlenül bárki számára elérhető legyen
- megbízhatóság növelés: több azonos funkciójú erőforrás, redundancia, adatbiztonság
- gazdaságosság növelés: egy drága szupergép helyett több, kisebb, olcsóbb (GRID computing, felhő számítás)
- új (speciális) szolgáltatások: a kommunikáció (pl. e-mail, chat)

Topológia szerinti osztályozásuk:

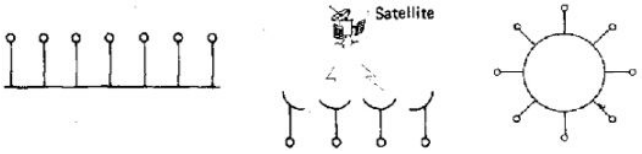
1. Pont-pont közötti kapcsolatokból felépülő: egy csatornán mindig két csomópont kommunikál, az (üzenet) csomagokat a csomópontok tárolják és továbbítják a kívánt irányba (store and forward)

- topológiák: csillag, gyűrű, fa, teljes, szabálytalan



2. Üzenetszórásos csatornára épülő hálózat (broadcast channel): egyetlen csatornán az összes csomópont osztozik, egy csomópont által feladott (üzenet) csomagot az összes többi veszi, de a csomagbeli címzésből tudják, kinek szól (a többi eldobja)

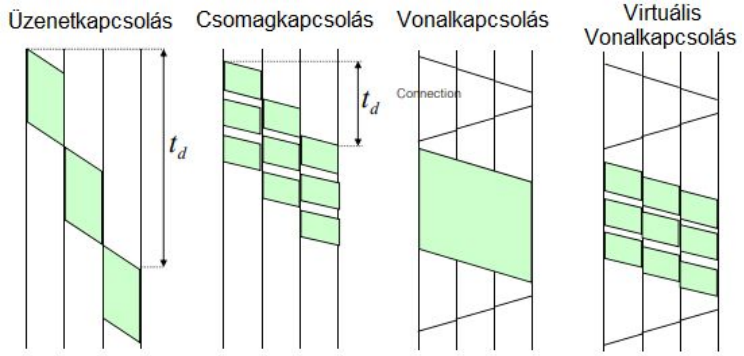
- címmezőkben a feladó és a címzett címe
- csatornamegosztás (melyik állomás adhat):
    - statikus
    - dinamikus (csak azok versenyeznek, akik adni akarnak):
        - centralizált (központosított)
        - decentralizált (elosztott)
- topológiák: sín, műholdas vagy rádiós, gyűrű



Méret szerinti osztályozásuk:

- lokális számítógép-hálózat (LAN): ≈ 0-1 km, szoba-épületcsoport, kis távolság, nagy sebesség
- városi számítógép-hálózat (MAN): <10 km, közepes táv, közepes sebesség
- nagytávolságú-hálózat (WAN): kontinensekre, nagytáv, közepes vagy kis sebesség
- összekapcsolt nagytávolságú hálózat: bolygóra kiterjedő

Kapcsolástechnika szerinti osztályozásuk:



Üzenetkapcsolás:

- teljes üzenet feladása megtörténik
- csomópontok tárolják majd továbbítják az üzenetet (store and forward)
- nincs korlát az üzenet méretére (tetszőleges késleltetés, prioritási lehetőség, tetszőleges tárolókapacitás igény a csomópontokon)
- torlódás jól kontrollálható, jól kihasználja a közeget

Csomagkapcsolás:

- felülről korlátos méretű csomagok az üzenetek feldarabolásával
- a csomópontok közötti kapcsolaton (linken) dinamikusan osztoznak a csomagok
- korlátos tárolókapacitás igény a csomópontokon
- kisebb késleltetés lehetséges (interaktív kommunikációra is alkalmas)
- nagyobb az átbocsátóképesség
- lehetőség van átlapolt működésre

Vonalkapcsolás:
- kapcsolat felépítés a végpontok között (időigényes)
- kommunikáció a dedikált vonalon
- burst (impulzusszerű) forgalom esetén nem kedvező (kihasználatlanság)
- pl.: nyilvános telefonhálózat

Virtuális vonalkapcsolás:
- csomagkapcsolás, de logikai útvonal a végpontok között (a csomagok ugyanazt az útvonalat használják, számít a sorrend)
- hasonlít a vonalkapcsoláshoz de nem dedikált vonal
- a logikai útvonalhoz kapcsolat felépítés kell

ISO-OSI hivatkozási modell: 7 rétegű struktúra
- rétegek: jól definiált szolgáltatásokat nyújtanak a felettük lévő rétegeknek, elrejtik a szolgáltatások megvalósításának részleteit

ISO: Nemzetközi Szabványügyi Szervezet (International Standards Organization)
OSI: Nyílt rendszerek összekapcsolása hivatkozási modellje (Open System Interconnection)

ISO-OSI hivatkozási modell szerkezete:
1. fizikai réteg: bitfolyam - bitek kommunikációs csatornán való áthaladásáért felelős
- a csatlakozók és a közeg fizikai kialakítása
- kapcsolat felépítése/bontása
- egyes bitek reprezentációja
- adatátviteli irányok meghatározása (szimplex, duplex, félduplex)

2. adatkapcsolati réteg: keretek - a hálózati réteg számára hibamentes átvitel
- keretképzés és behatárolás
- hibák ellenőrzése, javítása
- adatfolyam vezérlés, forgalomszabályozás (gyors adók lassú vevőket ne árasszák el)
- szükség esetén csatornamegosztás

3. hálózati réteg: csomagok - kommunikációs hálózat működését vezérli
- a csomagok forrás és célállomás közötti útvonalának meghatározása (statikus v dinamikus)
- torlódásvezérlés

4. szállítási réteg: datagram,szegmens - feladata a viszonyréteg üzeneteinek továbbítása
- üzenetek feldarabolása, összeállítása, hibakezelés, elrejti a konkrét hálózatot a felettes rétegek elől

5. viszonyréteg: üzenetek - különböző gépek között felhasználóviszonyok létesítése
- párbeszédek szervezése (1 vagy 2 irányú kapcsolatok)
- szinkronizáció
- kölcsönhatás menedzselés (két oldal ugyanazzal a művelettel ne próbálkozzon egyszerre)

6. megjelenítési réteg: üzenetek - az átvivendő üzenetek szemantikájával és szintaktikájával foglalkozik
- kód konverzió, titkosítás, tömörítés

7. alkalmazási réteg: üzenetek - széles körben igényelt protokollokat tartalmaz
- fájl és nyomtatószolgáltatások
- kommunikációs szolgáltatások
- directory szolgáltatások
- alkalmazás szolgáltatások

26. A számítógép-hálózatok ISO-OSI hivatkozási modell adatkapcsolati rétegének közeghozzáférési alrétege: A főbb csatorna-megosztási osztályok (statikus - dinamikus, dinamikus versengő - determinisztikus) bemutatása és azok összevetése. Az IEEE 802.3 és az Ethernet (CSMA/CD) keretformátum, MAC címek, támogatott közegek és sebességek bemutatása, működés duplex csatorna esetén. Az IEEE 802.11 WLAN általános felépítése, az Access Point (AP) főbb funkciói. Az alkalmazott közeghozzáférési módszer (CSMA/CA), Distributed Coordination Function (DCF) és Point Coordination Function (PCF) üzemmódok.

ISO-OSI hivatkozási modell adatkapcsolati réteg:
1) Logikai kapcsolatvezérlés alréteg (LLC):
- pont-pont kapcsolat esetén (LLC): keretképzés/behatárolás, hibavédelem, adatfolyam vezérlés, kapcsolat vezérlés.
2) Közeghozzáférés vezérlési alréteg (MAC):
- üzenetszórásos csatorna esetén (LLC+MAC):
    - egyetlen üzenetszórásos csatorna megosztása több egymással versenyző állomás között
    - pont-pont szerű szolgálat biztosítás LLC alréteg számára, csatornakiosztás.

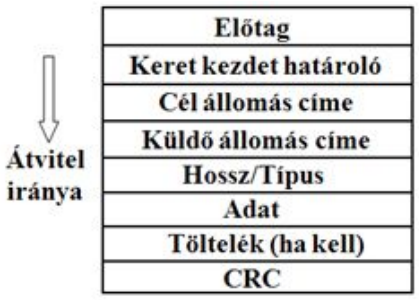
A főbb csatorna-megosztási osztályok:
Statikus: fix felhasználószám, a csatornából mindenki egyformán részesül akkor is, ha arra valamely időpontban ténylegesen nincs is szüksége
- FDM: frekvenciaosztásos nyalábolás: n db állomás esetén a sávszélesség n részre oszlik, mindenki egyenlő részt kap, mindenkinek külön frekvencia, nincs interferencia (nem zavarják egymást)
- TDM: időosztásos nyalábolás: n db állomás esetén a ciklusidő n egyenlő időrés oszlik, mindenkihez 1 időrés

Dinamikus: azon felhasználók között oszlik meg a csatorna, akiknek arra éppen szükségük van
- vezérlés szempontjából lehet:
    - centralizált: egy kijelölt (esetleg speciális) állomás rendelkezik a csatorna kiosztásáról
    - elosztott: elosztott algoritmus dönt (nincs kijelölt állomás)

Dinamikus versengő (ütközéses): kis forgalom esetén minimális késleltetés, de nagy forgalom esetén bedugulás, jóval az ideális alatti korlátos csatornakapacitás
- egyszerű ALOHA: azonos hosszúságú keretek, az állomások bármikor adhatnak és nyugtát várnak, ütközés esetén (ha nincs nyugta) véletlenszerű ideig vár, majd újraad
- réselt ALOHA: idő keretidőnyi résekre osztva, adást csak az időrés elején lehet kezdeni
- CSMA: csatornafigyelés adás előtt, hogy foglalt-e a csatorna (ha igen, nem ad az állomás)
- 1-perzisztens, p-perzisztens, nemperzisztens
- CSMA/CA: csatornafigyelő + ütközéselkerülő (adás előtt belehallgat a csatornába, csak akkor ad, ha szabad, egyébként vár, majd újra belehallgat)
- CSMA/CD: csatornafigyelő + ütközésérzékelő (adás közben érzékel, ütközéskor az ütközött állomások abbahagyják a keret adását)

Determinisztikus (ütközésmentes): a bedugulás elkerülésére
- alap bittérképes módszer (BBMM): bejelentkezési időszak: ha adni akar, 1-re állítja a bitjét
- üzenetszórás felismerés változó prioritással (BRAP): ha adni akar, a bejelentkezési időszak felfüggesztődik, az állomás adhat, ha vége folytatódik a bejelentkezés
- vezérjeles sín: az állomások egy üzenetszórásos csatornához kapcsolódnak, az az állomás adhat, akinél a token van
- vezérjeles gyűrű: ha egy állomás tokent kap: felvágja a gyűrűt és ad, ha nincs több adnivalója vagy lejárt a tokentartási idő: továbbadja a tokent

Az IEEE 802.3 és az Ethernet (CSMA/CD) keretformátum: az IEEE 802.3 szabvány az Ethernet adoptálása a fizikai és az adatkapcsolati réteg megvalósításaira, amely CSMA/CD-t használ az ütközések feloldására



7 bájt: 7 x '10101010' (Szinkronizáció)
1 bájt: '10101011'
6 bájt: 1-3 bájt a gyártó azonosítója, 4-6 bájt a sorszám
6 bájt: 1-3 bájt a gyártó azonosítója, 4-6 bájt a sorszám
2 bájt: hossz/típus jelzése
0 - 1500 bájt adat
0 - 46 bájt: A kerethossz nem lehet kisebb, mint 64 bájt
4 bájt: ellenőrző összeg

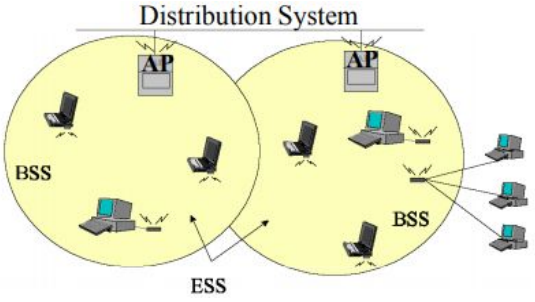
MAC cím: a hálózati eszközök egyedi azonosítója (egy hexadecimális számsorozat), amelyet az adatkapcsolati réteg MAC alrétege használ a hálózat előre meghatározott portjainak azonosítására
- 46.bit: lokális (ha 1) és globális (ha 1) címek megkülönböztetése
Megadása: 12 darab hexadecimális számjegy formájában
- az első hat számjegy kiosztását az IEEE felügyeli, ezek a gyártót vagy az eladót azonosítják
- a fennmaradó hat hexadecimális számjegyet a gyártó adminisztrálja (sorozatszám)
Speciális MAC cím: broadcast address (csupa 1-es): valamennyi állomásnak szóló cím

Támogatott közegek és sebességek bemutatása:
802.3 szabvány (10Mbps):
- sodrott érpár UTP/STP, pont-pont (10BaseT): max 100m
- vékony koaxiális kábel (10Base2): max 185m
- vastag szélessávú koaxiális kábel (10Base5): max 500m
- optikai kábel, üvegszál, pont-pont (10BaseF): multimódusú: 2km, monomódusú: 3-10km
802.3u: Fast Ethernet (100Mbps):
- 100BaseTX: fél-duplex: 100m, full-duplex: 200m
- 100BaseFX: max 200m
- 100VG-AnyLAN rendszer
802.3z: GigaEthernet (1000 Mbps):
- koaxiális, optikai és UTP kábelen is
802.3ae: Ethernet (10Gbps):
- optikai kábel, üvegszál (10GBASE): duplex multimódusú: 240-300m, monomódusú: 10-40km

Működés duplex csatorna esetén: a jelek a kapcsolatban lévő pontok között mindkét irányban haladhatnak
- fél-duplex: egyidőben csak egy irány használható
- full-duplex: egyszerre használható mindkét irány
802.3x szabvány: "pause-frame"
- azok az eszközök, akik meg akarják állítani az adatfolyamot pause-frame-et küldenek
- a pause-frame „slot time"-okban számolva tartalmazza azt az időt, amíg az adónak fel kellene függesztenie az adását (ez az időtartam további pause-frame-k küldésével módosítható)

Az IEEE 802.11 WLAN általános felépítése: az IEEE 802.11 egy vezetéknélküli adatátviteli protokoll a fizikai és az adatkapcsolati réteg közt
Közegek:
- infravörös (IR): közvetlen rálátás szükséges, 1-2Mbps
- frekvenciaugrásos szórt spektrum (FHSS): 2,4GHz, 1-2Mbps
- közvetlen szekvencia szórásos spektrum (DSSS): 5GHz, 1-2Mbps
Szabványok:
- 802.11a: 5GHz-es frekvenciatartomány, 54Mbps, pont-pont kapcsolat, max 120m
- 802.11b: 2,4GHz-es frekvenciatartomány, 11Mbps, pont-multipont kapcsolat, max 140m
- 802.11g: 2.4GHz-es frekvenciatartomány, 54Mbps, pont-multipont kapcsolat, max 140m

Felépítése: cellákra épülő struktúra



- BSS (Basic Service Set): egy cella
- AP (Access Point): az egyes cellákat irányító bázisállomások
- DS (Distribution System): a cellákat összekötő hálózat
- ESS (Extended Service Set): a teljes hálózat (több összekötött cella együtt)

roaming: cellák közötti mozgás

Access Point (AP) főbb funkciói: vezeték-nélküli hozzáférési pont
- csatlakozási pontként funkcionál a vezeték-nélküli kommunikációban, fizikailag összefogja a hálózati kapcsolatokat
- bridge-ként funkcionál a vezeték-nélküli hálózat és a kábelezett hálózat között

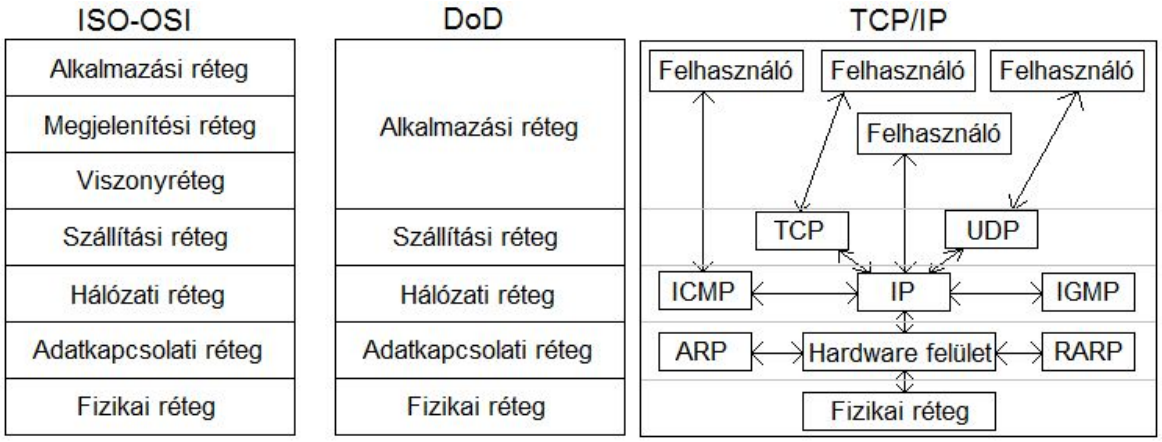
Az alkalmazott közeghozzáférési módszer (CSMA/CA):
Mielőtt adni kezd, belehallgat a csatornába:
- ha üres egy Interframe Spacing (IFS) időn át: adni kezd,
- ha foglalt: megvárja míg felszabadul, véletlenszerű ideig vár (backoff), majd újra próbálkozik
Ütközés esetén úgy veszi, mintha foglalt lenne a csatorna.
DCF (Distributed Coordination Function): speciális CSMA/CA
- belehallgatás közben Distributed Interframe Space (DIFS) időn át nézi, hogy üres-e
- ha egy állomás egy neki címzett hibátlan üzenetet vesz, akkor egy Short Interframe Space (SIFS) eltelte után pozitív nyugtát küld a feladónak
- ütközés: pozitív nyugta hiánya, ekkor új backoff időt sorsol

Point Coordination Function (PCF) üzemmód: centralizált ütközésmentes közeghozzáférés vezérlés
- működését a Point Coordinator (PC) irányítja
- az állomások kérhetik, hogy kerüljenek rá a PC Polling List-jére
- versengéses és versengés nélküli időszakok váltakoznak a DCF-ért
- rendszeres időközönként a PC hamarabb foglalja le a csatornát, mint a többi állomás, ezután a Polling List-en lévő állomások adhatnak

27. A TCP/IP protokoll szövet és az Internet: Az Internet hivatkozási modell (DoD) és az ISO-OSI hivatkozási modell összevetése. A TCP/IP protokoll szövet főbb részei (ARP, RARP, IP, ICMP, TCP, UDP) és azok funkcióik. Az Internet címzés és címosztályok: IPv4 címosztályok, maszk, subnet, supernet, osztály nélküli címzés (CIDR Classless Inter-Domain Routing), és a változó alhálózat méretek (VLSM Variable Length Subnet Mask); címek kiosztása, lokális címek és a címfordítás (NAT). Az IPv6 címek, IPv6 cím típusok (unicast, multicast, anycast; link local, unique local, aggregatable global unicast address).

A TCP/IP protokoll hierarchiák:
- világméretű hálózat alakítható ki belőle (Internet: külső hálózat - WAN, Intranet: belső hálózat - LAN)
- általános hálózati kommunikációs szolgálatok készletét biztosítja
TCP/IP protokoll állapotok: required: szükséges, recommended: ajánlott, elective: választható, limited use: részlegesen használható, not recommended: nem ajánlott

Az Internet hivatkozási modell (DoD) és az ISO-OSI hivatkozási modell összevetése:



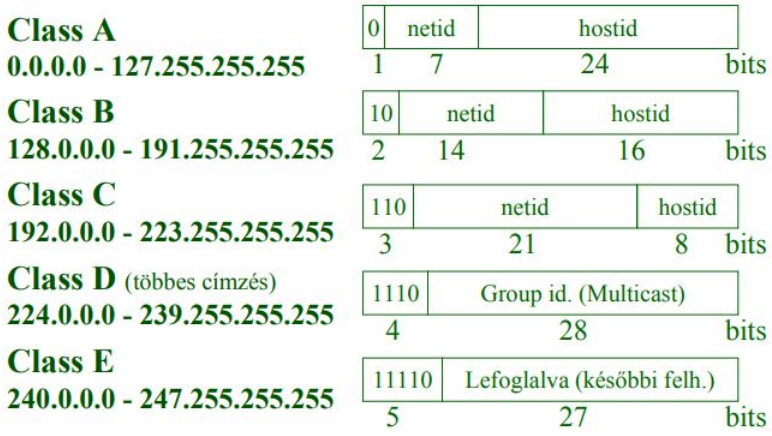
A TCP/IP protokoll hierarchia főbb részei: DoD modell alapján

- szállítási réteg: host-to-host
  - TCP: megbízható adattovábbítás, összeköttetés (sorrendhelyes), kétirányú
  - UDP: összeköttetés mentes datagram szolgálat, nem megbízható
- hálózati réteg: internet
  - ICMP: a hálózati réteggel kapcsolatos üzenetek
  - IP: összeköttetés-mentes datagram szolgálat változó méretű csomagokra
  - IGMP: többes címzéssel kapcsolatos üzenetek
- adatkapcsolati réteg: internet hozzáférés
  - ARP: host vagy router IP címének leképzése MAC címmé
  - RARP: saját cím lekérdezése a saját MAC cím alapján

Az Internet címzés és címosztályok:

32 bites, 4 byte, decimális alak

IPv4 címosztályok: hálózat cím + hoszt cím (netid+hostid)



Maszk: a maszk mutatja az értékes bitek számát, innen tudjuk, hogy melyik tartományba tartozik

- VLSM: ha valamely cél cím több hálózatra (irányra) is illeszkedik, a router arra továbbítja a csomagot, amerre a leghosszabb a prefix
- CIDR: a kiterjesztett prefixet (subnet maszkot) is továbbítania kell, mert a címkiosztás figyelmen kívül hagyja az osztályokat, ezért a kiterjesztett prefixet is el kell küldeni, hogy egyértelmű legyen a cím

Subnet (alhálózat): kiterjesztett prefix, amivel a hálózat bitjeit jelöljük

Supernet: címfeldolgozásnál több hagyományos osztály összefogása, rövidebb subnet maszk esetén

Osztály nélküli címzés (CIDR): több bitszomszédos hálózat összefogása

A szomszédos A,B,C hálózat összevont útvonalválasztási bejegyzése, tehát a ki nem osztott IP címeket változó méretű blokkokban osztja ki osztályokra való tekintet nélkül.

Változó alhálózat méretek (VLSM): különböző alhálózatok létrehozása hatékonyabb címfelhasználással

Címek kiosztása: hálózati címosztályok szerint az IP cím egyértelműen két részre bontható:
- első bitek megmondják hol a határ
- broadcast cím egyértelműen számítható
Lokális címek: magánhálózaton tetszőleges IP cím kiosztást készíthetünk, de csatlakozáskor gondot okozhat, ezt oldja meg a bejegyzett címtartományok használata
Címfordítás (NAT): a belső és külső IP címek összerendelése
- egyetlen külső cím esetén: kicseréli a belső forrás címet a külső címre, majd megnézi, hogy az eredeti forrás port szabad-e:
    - ha szabad, azt választja
    - ha foglalt, akkor a szabadok közül választ
    - ha nincs szabad port, akkor eldobja a csomagot

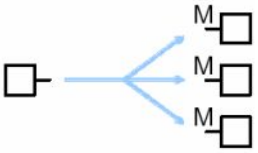
ezután bejegyzi egy táblázatba a fordítást a visszafelé jövő, illetve további csomagok érdekében
- több külső cím: ha nincs szabad port, akkor veszi a következő külső címet és azon keres szabad portot

Az IPv6 címek: hexadecimális alakban **:** -tal (kettősponttal) elválasztva
- 128 bitesek (16 byte)
- összesen $2^{128} = 3.4 * 10^{38}$ cím
- könnyű subnetet kialakítani, nincs szükség NAT-olásra
- IPv6 címek interfész címek: egy hosztnak több interfésze lehet
- nincs broadcast cím, csak unicast, multicast és anycast

IPv6 címosztályok:
Unicast: egyedi címzés (one-to-one communication) 

egyedi hálózati interfészt azonosítanak, ahol a unicast címre küldött csomagokat az internetprotokoll a megfelelő egyedi interfésznek továbbítja
- link local address: csak egy linken (prefix: FE80::/64)
    - nem használhatók globális kommunikációra
    - routereken nem szabad kiengedni
    - mindig automatikusan konfigurálódnak, még akkor is, ha semmilyen más unicast cím sem létezik
- unique local address: egyediek (prefix: FEC0::/48)
    - nem automatikusan konfigurálódnak, hanem vagy állapotmentes (stateless), vagy állapot alapú (stateful) cím konfigurációval kell megadni azokat
- aggregatable global unicast address: globális kommunikáció esetén, szolgáltató szerinti és egyéb csatlakozás alapú aggregáció

Multicast: többes címzés (one-to-many communication)
több hoszt használja, ezek egy multicast csoport tagjai, a multicast címre küldött csomagot a csoport minden interfésze megkapja

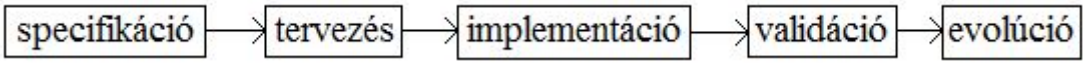


Anycast: legközelebbi címzés (one-to-nearest communication)
- csoport interfészhez vannak hozzárendelve, egy anycast címre küldött csomagot a csoport legközelebbi interfésze kapja meg
- majdnem minden unicast cím használható anycast címként is, mivel ugyanolyan formájúak

28. Szoftvertechnológia és a szoftverfolyamat fogalma. A szoftverfolyamat fázisok részletes bemutatása: Szoftverspecifikáció, tervezés, implementáció, validáció és evolúció. Az evolúciós modell ismertetése.

Szoftvertechnológia: olyan technológiai és vezetési alapelvek összessége, amelyek lehetővé teszik a programok termékszerű gyártását és karbantartását a költség- és határidő korlátok betartásával
- tudományos ismeretek gyakorlati alkalmazása

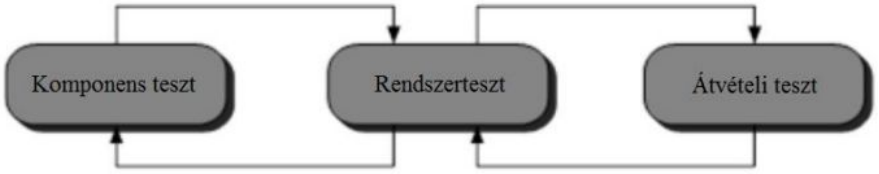
Szoftverfolyamat: tevékenységek és eredmények sora, amelyek egy szoftvertermék előállításához vezetnek
- komplex, nagyban függ az emberi tevékenységektől → nincs ideális megoldás, szervezetenként eltérő
- cél: úgy kialakítani, hogy kiaknázzák a szervezeten belül az emberek képességeit és a fejlesztő rendszer jellegzetességeit
- fázisai:



Szoftverspecifikáció: a szoftver funkcióinak és annak megszorításainak definiálása (követelményervezés)
- cél: meghatározni, hogy a rendszernek milyen szolgáltatásokat kell biztosítania
- eredmény: követelménydokumentum
- folyamata:
1. Megvalósíthatósági tanulmány: az ügyfél igényeinek felmérése, hogy kielégíthetőek-e az adott szoftver- és hardvertechnológia mellett, és ha igen, mennyire költséghatékonyak
2. Követelmények feltárása és elemzése: különböző prototípusok és rendszermodellek elkészítése megbeszélések, megfigyelések és tevékenységelemzések alapján
3. Követelmény specifikáció: az elemzési tevékenységek során összegyűjtött információk egységes dokumentummá alakítása
4. Követelmény-validáció: a követelmények vizsgálata, ahol azt figyeljük, mennyire konzisztensek és valószerűek a követelmények, illetve tartalmaznak-e bármilyen hibát

Tervezés és implementáció: a szoftver logikai szerkezetére vonatkozó döntések meghozatala:
- meg kell határozni az implementálandó szoftver struktúráját, az adatok szervezését és áramlását a rendszerben, a komponensek közötti interfészeket és a használt algoritmusok leírását
- folyamata:
1. Architekturális tervezés: az alrendszerek és a köztük található kapcsolatok dokumentálása
2. Absztrakt specifikáció: az alrendszerek megszorításainak és szolgáltatásainak megadása
3. Interfész tervezése: az alrendszerek közötti interfészek megtervezése és dokumentálása
4. Komponens tervezése: a szolgáltatások komponensekre bontása és interfészeik megtervezése
5. Adatszerkezet tervezése: a rendszer implementációban használt adatszerkezetek megtervezése
6. Algoritmus tervezése: a szolgáltatások biztosításához szükséges algoritmusok megtervezése
- implementáció: a specifikációnak megfelelő szoftver előállítása
    - prototípusok készítése
    - iteratív fejlesztési modellek

Validáció: a kifejlesztett rendszer tesztelése, hogy megfelel-e a szoftverspecifikációnak és az ügyfél követelményeinek egy háromlépéses tesztelési folyamattal:

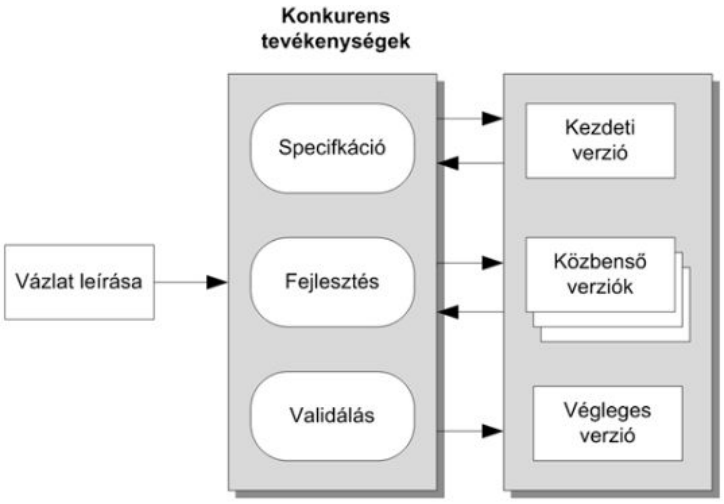


1. Komponens vagy egységteszt: a komponensek funkcionalitásának tesztelése különböző tesztesetekkel a rendszerkomponensektől függetlenül
2. Rendszer- és integrációs teszt:
- integrációs teszt: az alrendszerek és interfészeik közötti hibák felderítésével foglalkozik
- rendszerteszt: annak tesztelése, hogy a teljes rendszer eleget tesz-e a funkcionális és nem-funkcionális követelményeknek
3. Átvételi vagy funkcionális tesztelés: a valós adatokkal való tesztelés a végfelhasználók által

Evolúció: a szoftver karbantartása a lefejlesztés után, hogy megfeleljen a megrendelő általi változtatásoknak
- a szoftver bővítése: új funkciók bevezetése
- adaptálás: a szoftver új (hardver/szoftver) környezetbe történő bevezetése
- hibajavítás: a kiadás után felmerült hibák javítása

Szoftverfejlesztési modellek: a szoftverfolyamat absztrakt reprezentációi
- vízesésmodell: a folyamat alapvető tevékenységei különálló fázisok
- evolúciós modell: iteratív fejlesztés, ahol a szoftver specifikációja, tervezése, implementálása és validálása összefésülődik
- komponens alapú modell: nagy mennyiségű újrafelhasználható komponensek létezésén alapszik

Az evolúciós modell: iteratív fejlesztési modell



1. Feltáró fejlesztés: a követelmények feltárása a megrendelővel egyeztetve történik, a rendszer kialakítása a követelmények érthetőségének és fontosságának sorrendjében történik
- célja egy működőképes rendszer átadása a végfelhasználóknak

2. Eldobható prototípus készítés: a feltáró fejlesztéssel ellentétesen, a kevésbé érthető részek fejlesztése az elsődleges
- célja, hogy a lehető legjobban megértsük az ügyfél követelményeit

29. Szoftverkövetelmény fogalma és osztályozásai. Követelmények általános problémái. Strukturált természetes nyelv. OO felbontás jellemzői. Vezérlési stílusok.

Szoftverkövetelmény: a rendszer szolgáltatásainak és megszorításainak leírásai
- követelmények tervezése: a szolgáltatások és megszorítások kitalálásának, elemzésének, dokumentálásának és ellenőrzésének folyamata
1. Felhasználói követelmények: absztrakt követelmények diagramokkal kiegészített természetes nyelvű kijelentéseinek leírása
2. Rendszerkövetelmények: a rendszer funkcióinak, szolgáltatásainak és működési megszorításainak részletes, pontos leírása

Osztályozásai:
1. Funkcionális követelmények: leírja, hogy hogyan kellene működnie a rendszernek
- a funkciók és szolgáltatások teljes és ellentmondásmentes ismertetése
2. Nemfunkcionális követelmények:
- a rendszertulajdonságok specifikálása és a funkciókra és szolgáltatásokra tett megszorítások
- termékre vonatkozó követelmények: a termék viselkedését határozzák meg (tárfoglalás, megbízhatóság, hordozhatóság, stb.)
- szervezeti követelmények: a megrendelő és a fejlesztő szervezetének belső szabályzataiból eredőek (szabványok, megvalósítás körülményei)
- külső követelmények: együttműködési, törvényi, etikai követelmények
3. Szakterületi követelmények: a rendszer alkalmazási szakterületéről származó jellegzetességeket és megszorításokat tartalmaznak

Követelmények általános problémái: természetes nyelven íródnak
- egyértelműség hiánya: pontatlan, terjengős leírások
- követelmények keveredése: a funkcionális, nemfunkcionális és szakterületi követelmények keveredése
- követelmények ötvöződése: több különböző követelmény egybefonódása

Strukturált természetes nyelv: a rendszerkövetelmények leírásának nyelve
- a természetes nyelv egyik leszűkítése, amely a kifejezőképesség és az érthetőség mellett egységességet is nyújt

OO felbontás: moduláris felbontási stratégia, amely objektumosztályokkal és azok attribútumaival, műveleteivel foglalkozik
- a rendszert lazán kapcsolódó, jól definiált interfészekkel rendelkező objektumok halmazára bontjuk
- az objektumok a többi objektum által használt szolgáltatásokkal kommunikálnak

Vezérlési modellek: alrendszerek közötti vezérlési folyamatokkal foglalkoznak
Vezérlési stílusok:
1. Központosított vezérlés: egyetlen alrendszer (rendszervezérlő) látja el a vezérlés teljes felelősségét
- hívás-visszatérés modell: fa-hierarchia, szekvenciális végrehajtás
- eseményciklus-modell: rendszerkezelő komponens irányítja a vezérlést, párhuzamos végrehajtás
2. Eseményalapú vezérlés: külsőleg létrejött események irányítják a vezérlést
- eseményszóró modell: az esemény minden alrendszerhez eljut és eldönthetik, hogy reagálnak-e rá
- megszakítás-vezérelt modell: valós idejű, ahol egy megszakítás-kezelő észleli a külső megszakítást

30. UML diagramok bemutatása: Use case és osztálydiagram. A szekvencia diagram. Állapotdiagram és aktivitás diagram. Komponens diagram.

UML: egységesített, szabványos, grafikus modellező nyelv
- gazdag jelölésrendszerrel (ábrák, diagramok, táblázatok) vizuálisan megjeleníthető fejlesztési eredmények dokumentálására alkalmas
- objektum-orientált eszközök fejlesztésének támogatása
- rendszer specifikáció: UML diagram + forráskód

UML diagramok: rendszerspecifikáció egyik megjelenési formája
- struktúramodellezés: az objektumok közötti összeköttetések topológiáját írják le
- viselkedésmodellezés: az objektumok közötti üzenetváltásokat és hatásukat írják le

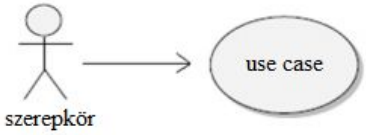
Use case diagram: viselkedés diagram, a rendszer használatát adja meg aktorok felhasználásával
- célja: a rendszer funkcionális működését, viselkedését és a környezetének kapcsolatait írja le a felhasználó szemszögéből nézve
- aktor: a rendszerrel kapcsolatban érdekeltek szerepköre (jelölése: pálcikaember)

- több felhasználó - egy aktor
- egy felhasználó - több aktor

- use case (használati eset): a felhasználó és a rendszer közötti interakciót írja le (jelölése: ellipszis)

- pontos leírás (forgatókönyv) is szükséges

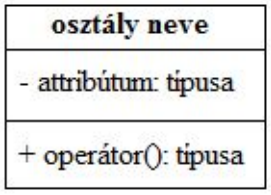
- kapcsolat aktor és a use case között (jelölése: nyíl)



- a nyíl iránya az aktor felől érkezik
- UML ábrákban asszociációnak nevezik

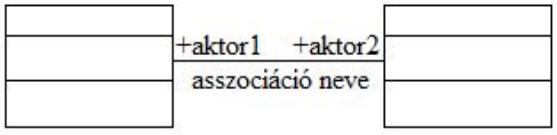
- kapcsolat két use case között
    - viszony szerint lehet include (tartalmazás), extend (kiterjesztés) vagy általános
- kapcsolat két aktor között: öröklődési viszony

Osztálydiagram: strukturális diagram, a rendszer osztályait és a köztük lévő társítási kapcsolatokat írja le
- szintek:
    - analízis, elemzés: a cél az alkalmazási szakterület fogalmainak megértése
    - tervezés: bővebb reprezentáció
    - megvalósítás, implementáció: teljes reprezentáció
- osztály: alapegység (jelölése: függőlegesen három részre osztott téglalap)



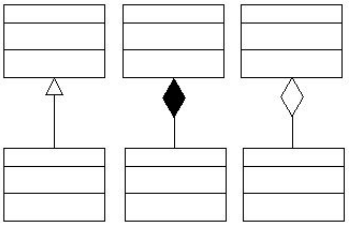
- osztály leve: kötelező megadni
- attribútum: egy osztály tulajdonságainak felsorolása
    - jelölése: *láthatóság név : típus = alapérték*
- operátor: egy osztály példányain végezhető műveletek felsorolása
    - jelölése: *láthatóság név(param) : típus{komment}*

- láthatóság: + public, # protected, ~ csomagszintű, - private
- asszociáció: általános kapcsolat (jelölése: vonal)

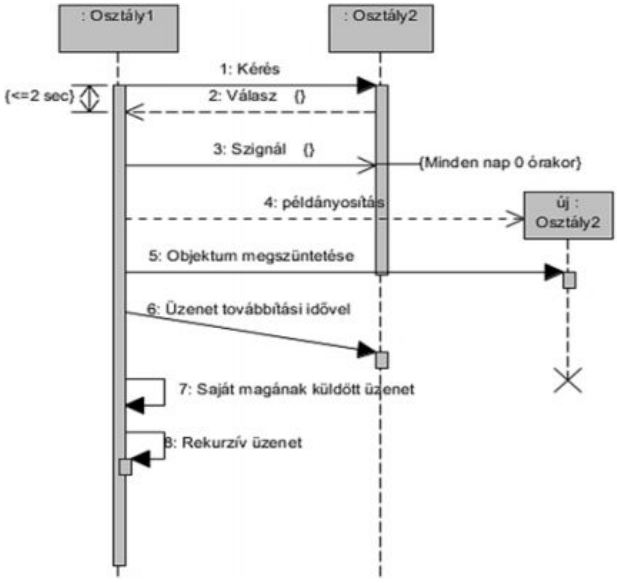


- két osztály közötti asszociációhoz tartozhat több szerep is: minden szerepnek saját vonal
- több osztály között is fennállhat asszociáció

- öröklődés (jelölése: zárt nyílvégű nyíl)
- kompozíció: tartalmazás, ahol a tartalmazó nem jöhet létre az őse előtt (jelölése: tömött rombusz)
- aggregáció: tartalmazás, ahol a tartalmazó létrejöhet az őse előtt (jelölése: üres rombusz)

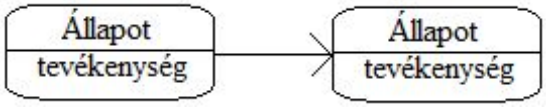


Szekvencia diagram: viselkedés diagram, az aktorok és objektumok közötti kapcsolatot írja le
- feladata: az objektumok egymás közötti üzenetváltásainak ábrázolása egy időtengely (életvonal) mentén
- felépítése: életvonal felülről lefelé halad, nyíl mutatja az irányt, lefelé lejtő nyíl időigényesebb üzenet
- elemei:
    - objektumok
    - üzenetek
    - megjegyzések

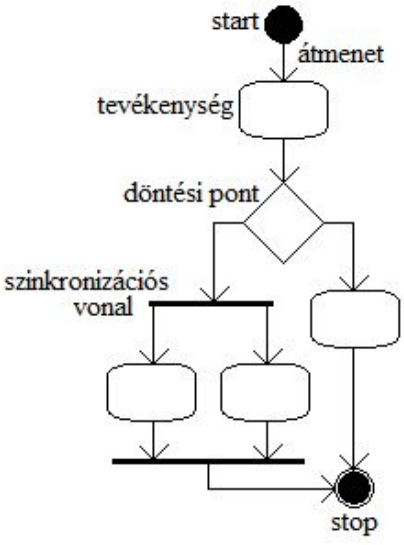


Üzenetfajták:
- szinkron üzenet: kérés, a küldő elküldi az üzenetet, majd vár a válaszra, a fogadó aktiválódik
- válaszüzenet: szinkron üzenetre való válasz, a küldő deaktiválódik, a fogadó aktiválódik
- aszinkron üzenet (szignál): a küldő elküldi az üzenetet, de nem vár válaszra, folytatja a munkáját
- objektum létrehozása: a nyíl a létrejött objektumra mutat
- objektum megszüntetése: az objektum megszűnésekor az életvonala lezárul (X)
- üzenet továbbítási idővel: lefelé lejtő nyíl jelöli az időigényesebb üzenetet
- saját magának küldött üzenet
- rekurzív üzenet: saját magának küldött üzenet beágyazott aktivitási szakasszal

<u>Állapotdiagram</u>: viselkedés diagram, egy adott osztály vagy alrendszer állapotváltozásait írja le
- objektumok által felvehető állapotokat és az állapotok közötti átmeneteket szemlélteti
- állapot: van időtartama, mindkét irányban nyilak vezetnek (speciális: kezdőállapot, végállapot)
    - tevékenységek: *entry/* (belépés), *exit/* (kilépés), *do/* (normál), *on esemény/* (eseményhez kapcsolódó tevékenység)
- az átmenetek atomi egységek, nem szakíthatóak meg (jelölésük: nyitott végű nyíl)



<u>Aktivitás diagram</u>: viselkedés diagram, a rendszer valamely folyamatát írja le
- időben lezajló folyamatokat és változásokat ábrázolhatunk
- folyamatábrán alapszik



- tevékenység: valamilyen végrehajtandó műveletsorozat (jelölése: lekerekített sarkú téglalap)

- átmenet: egymás után végrehajtandó műveletek közti átmenet (jelölése: nyíl)

- döntési pont: elágazás létrehozása (jelölése: rombusz)

- kezdő- és végállapot

- szinkronizációs vonal: párhuzamosítást valósit meg (szétválasztás és összeolvasztás)

<u>Komponens diagram</u>: strukturális diagram, a szoftver fizikai felépítését adja meg
- a rendszert alkotó fizikai komponensek és a köztük lévő kapcsolat ábrázolása
- komponensek egymáshoz rendelése, csoportosítása