

Részproblémára bontó algoritmusok (mohó, oszd-meg-és-uralkodj, dinamikus programozás), rendező algoritmusok, gráfalgoritmusok (szélességi- és mélységi keresés, minimális feszítőfák, legrövidebb utak)

Részproblémára bontó algoritmusok

Oszd meg és uralkodj

- a feladatot több részfeladatra osztjuk
- a részfeladatok hasonlóak az eredeti feladathoz, de kisebbek
- rekurzívan megoldjuk a kisebb részfeladatokat
- a megoldásokat összevonjuk, és az adja a végső megoldást

Felosztás: hogyan osztjuk több részfeladatra

Uralkodás: a részfeladatokat rekurzív módon megoldjuk

Összevonás: a részfeladatok megoldásait összevonjuk

Példa

Felező-csúcskereső algoritmus

Input: egy számsorozat Output: van-e benne csúcs?

Algoritmus: az n méretű sorozat helyett vizsgáljunk egy $(n-1)/2$ méretűt, és ebben keressünk csúcsot, majd ezt folytatjuk rekurzívan

Dinamikus programozás

Pénzváltási feladat

Input: P_1, P_2, \dots, P_n típusú pénzérmék, F forint Output: legkevesebb hány érmével fizethető ki pontosan F forint?

Dinamikus programozás akkor, ha a részproblémák nem függetlenek, hanem vannak közös részeik

Alapgondolat: a már megoldott részproblémák optimális megoldásait megjegyezzük

- így minden részproblémát csak egyszer fogunk megoldani

Pénzváltási feladat megoldása DP-vel: minden összegre F -ig kiszámoljuk, hogy azt minimum hány pénzérmével tudjuk kifizetni

- ötlet: minden érmére megnézzük, hogy a korábbi optimális megoldás a jó, amiben nem volt benne az az érme, vagy az, ha benne van az érme

- futásidő: $O(Fn)$

Iteratív megoldás: bottom-up építkezünk, és minden lehetséges értéket megnézünk Rekurzív megoldás memorizálással: top-down építkezünk, és kulcs-érték párokat nézünk (csak akkor, ha nem kell minden részmegoldás)

Mohó algoritmusok

Részfeladatra bontás Optimalizálás

A mohó algoritmus minden lépésben az aktuálisan optimálisnak tűnő megoldást választja. Nem minden problémára adható mohó algoritmus! De ha igen, akkor az nagyon hatékony

Részproblémára bontáskor a cél: a mohó választás egyetlen részproblémát eredményezzen, aminek az optimális megoldása a probléma optimális megoldása is egyben

Mohó algoritmusok helyessége:

- fogalmazzuk meg a feladatot úgy, hogy minden döntés hatására egy kisebb részprobléma keletkezzen
- bizonyítsuk be, hogy mindig van mohó választási lehetőség, tehát biztonságos
- mohó választással olyan részprobléma keletkezik, amihez hozzávéve a mohó választást, az eredeti probléma optimális megoldását kapjuk (optimális részstruktúrák)

Egy feladat optimális részstruktúrájú, ha a probléma egy opt. megoldása tartalmazza a részfeladatok optimális megoldásait is.

Példák

Hátizsák probléma

- adott egy hátizsák kapacitása, és n tárgy, mindegyik értékkel és súllyal megadva
- mekkora a legnagyobb összérték, amit a hátizsákba tehetünk?

Töredékes hátizsák probléma

- a tárgyak feldarabolhatók
- de minden tárgyból egy darab van

Mohó algoritmus a töredékes hátizsákra:

- számoljuk ki minden tárgyra az érték/súly arányt
- tegyük a hátizsákba a legnagyobb ilyen arányú tárgyat, az egészet ha belefér, vagy törjük, ha nem

2. Elemi adatszerkezetek, bináris keresőfák, hasító táblázatok, gráfok és fák számítógépes reprezentációja

Elemi adatszerkezetek

tömb, láncolt lista, sor, verem, gráf, map, kupac - saját vélemény

Adatszerkezet

- adatok tárolására és szervezésére szolgáló módszer
- lehetővé teszi a hatékony hozzáférést és módosításokat

Leggyakoribb műveletek

- beszúr
- keres
- töröl
- min
- max
- előző
- következő

Megfelelő adatszerkezet kulcs az implementáció futásidejéhez!

Listák

Az adatok lineárisan követik egymást. Egy érték többször is előfordulhat.

Műveletek: érték, értékad, keres, beszúr, töröl

Közvetlen elérés

- minden index közvetlen elérésű, közvetlenül írható/olvasható
- érték: $O(1)$, keres: $O(n)$
- beszúr és töröl esetén változik a méret, át kell másolni az elemeket egy új helyre
- beszúr: $O(n)$, töröl: $O(n)$
- ötlet: ha tele van a tömb, duplázzuk meg a kapacitást
- ha negyedére csökken, felezzük a kapacitást
- így nem kell mindig az egész tömböt másolni

Láncolt lista

minden érték mellé mutatókat tárolunk a következő/megelőző elemre

- egyszeresen láncolt: következő elemre mutat
- kétszeresen láncolt: előző és következőre is mutat
- ciklikus: az utolsó az első elemre mutat
- őrszem: egy nil elem, ami a lista elejére (fej) mutat

Közvetlen elérés vs Láncolt lista

- KÉ: érték konstans, módosító lassú
- LL: érték lassú, módosító gyors, sok memória kell a mutatóknak

Verem és sor

Stack, Queue

Olyan listák, ahol a beszúrás és a törlés csak adott pozíción történhet

- verem: legutoljára beszúrt elemet vehetjük csak ki (LIFO - Last In First Out)

- sor: legkorábban beszúrt elemet vehetjük csak ki (FIFO - First In First Out)

Verem műveletek

- push: verem tetejére rakunk egy elemet
- pop: verem tetejéről levesszük

Sor műveletek:

- enqueue: sor végére rakunk
- dequeue: sor elejéről elveszünk

Prioritási sor és kupac

Prioritási sor

- érkezés sorrendje lényegtelen, mindig a min/max elemet akarjuk kivenni

lehet mondjuk listával megvalósítani, veremmel vagy sorral nem érdemes, mert nem számít a sorrend

Prioritási sor hatékony megvalósítása: kupac (heap)

Kupac

- majdnem teljes bináris fa, minden csúcsa legalább akkora, mint a gyerekei -> max elem a gyökérben

Bináris keresőfák

Keres, beszúr, töröl, min, max, következő, előző Mind legyen $O(\log n)$

Bináris keresőfa

- minden csúcsnak max két gyereke van
- balra vannak a kisebb elemek
- jobbra a nagyobbak
- keresés $O(h)$ idejű (h a fa magassága)
- min/max is $O(h)$: vagy teljesen jobbra, vagy teljesen balra kell lemennünk
- következő/előző szintén $O(h)$ - amíg jobb/bal gyerek, addig megyünk max
- beszúr szintén $O(h)$ - mindig levélként szúrunk be, úgy, hogy kb megkeressük a helyét
- töröl is $O(h)$, levelet simán törölünk, egy gyerekeset úgy, hogy a gyereket linkeljük a szülőhöz, két gyerekeset pedig a következővel helyettesítjük

Hasító táblák

Halmaz és szótár

Halmaz

- egy elem legfeljebb egyszer szerepelhet benne
- keres helyett tartalmaz-e
- beszúr, töröl
- metszet, unió

Szótár

- kulcs érték párok halmaza
- minden kulcs legfeljebb egyszer szerepelhet
- egy érték tetszőleges számban előfordulhat

Asszociatív tömb

- egészek helyett bármilyen típussal indexelhetünk

Map

- kulcs->érték leképezés

Hasítótáblák

Halmazok és szótárak hatékony megvalósítása Keres, beszúr, töröl legyen hatékony

Átlagos esetben $O(1)$

Hasítótábla olyan szótár, amikor egy hash függvény segítségével állapítjuk meg, hogy melyik kulcshoz milyen érték tartozzon

példa: $h(k) = k \bmod m$ ahol m a hasító tábla mérete lehetnek ütközések! cél: az ütközések minimalizálása

Gráfok és fák számítógépes reprezentációja

Szomszédsági mátrix

- minden csúcshoz hozzárendelünk egy számot
- ha a és b között van él, akkor $matrix[a][b] = 1$ és $matrix[b][a] = 1$
- ha nincs, akkor 0

Szomszédsági lista

- minden listaelem egy csúcs, ami szintén egy lista
- minden csúcshoz tartozó listában tároljuk a vele szomszédos csúcsokat

Bal gyerek, jobb testvér

Bal gyerek, jobb gyerek

Binary Search Tree - tömbbel is meg lehet

- Index of parent = $\text{INT}[\text{index of child node}/2]$
- Index of Left Child = $2 * \text{Index of parent}$
- Index of Right Child = $2 * \text{Index of parent} + 1$

3. Hatékony visszavezetés. Nemdeterminizmus. A P és NP osztályok. NP-teljes problémák

Hatékony visszavezetés

Visszavezetésnek nevezzük azt, mikor ha van egy problémánk, amit nem tudjuk, hogy kéne megoldanunk, és egy problémánk, amit tudjuk hogy oldjunk meg, és a nem ismert probléma inputjaiból elkészítjük az ismert probléma egy inputját, és így oldjuk azt meg.

- Az átalakításnak tartania kell a választ
- Mindenre jó outputot kell adnia

Hatékonyan akkor nevezhetjük, ha ez az *inputkonverzió* polinomidejű. Ezt Turing-visszavezetésnek is hívják.

Nemdeterminizmus

Egy algoritmus *nemdeterminisztikus*, ha egy ponton úgymond szétválik a futása, és többféle eredménye is lehet a futás végére.

P és NP osztályok

A *P* osztályban azok a problémák vannak, amelyek determinisztikusan polinomidőben megoldhatók.

Az *NP* osztályban azok a problémák vannak, amelyek nemdeterminisztikusan polinomidőben megoldhatók.

NP teljes problémák

Egy probléma akkor *NP*-teljes, ha *NP*-beli és *NP*-nehéz.

- *NP*-beli, ha nemdeterminisztikusan tudunk tanúkat generálni hozzá, amik igen példányai a problémának
- *NP*-nehéz, ha minden más *NP*-beli problémát hatékonyan vissza tudunk vezetni rá.

Példák

SAT, Hátizsák, Hamilton-út, Hamilton-kör, Euler-kör, ILP, Részletösszeg, Partíció

4. A PSPACE osztály. PSPACE-teljes problémák. Logaritmikus tárigényű visszavezetés. NL-teljes problémák

PSPACE osztály

Savitch-tétel

- Elérhetőség eldönthető $O(\log^2 n)$ tárban

Az $f(n)$ tárban nemdeterminisztikusan eldönthető problémák mind eldönthetők determinisztikusan, $f^2(n)$ tárban is

Tehát: $\text{NSPACE}(f(n))$ részhalmaza $\text{SPACE}(f^2(n))$ -nek és mivel polinom négyzete polinom $\text{PSPACE} = \text{NPSPACE}$

Polinom tárban (det. vagy nemdet.) eldönthető problémák osztálya

PSPACE-teljes problémák

QSAT PSPACE-teljes

QSAT (kvantifikált SAT)

- adott egy ítéletkalkulusbeli logikai formula, változó kvantorokkal az elején (létezik, bármely, létezik, bármely stb)
- magja CNF alakú, kvantormentes
- igaz-e ez a formula?

QSAT mint kétszemélyes játék

- input ugyanaz
- van-e az első játékosnak nyerő stratégiája abban a játékban, ahol:
 - a játékosok sorban értéket adnak a változóknak, első játékos x_1 -nek, második x_2 -nek stb
- ha a formula igaz lesz, az első játékos nyert, ha hamis, akkor a második
- ez ugyanaz tkp, mint a sima QSAT, szóval ez is PSPACE-teljes

hasonlít a minimaxra az éses csúcsoknál lévő játékos minimalizál

Földrajzi játék

- adott egy irányított gráf és egy kijelölt kezdőcsúcs
- az első játékosnak van-e nyerő stratégiája?
 - az első játékos kezd, lerakja a bábút a kezdőcsúcsra
 - felváltva lépnek
 - egy olyan csúcsba kell húzni a bábút, ami egy lépésben elérhető, és ahol még nem voltak
 - aki először nem tud lépni, veszített

Földrajzi játék PSPACE-teljes

Adott két reguláris kifejezés, igaz-e, hogy ugyanazokra a szavakra illeszkednek? Adott két nemdet automata, ekvivalensek-e? Adott, egy SOKOBAN/RUSH HOUR feladvány, megoldható-e?

Logtáras visszavezetés

Polinomidejű visszavezetés túl erős, ha pl P-beli problémákat akarunk egymáshoz viszonyítani, mert egy polinomidejű visszavezetés alatt már akár meg is oldhatnánk egy P-beli problémát

Logtáras visszavezetés

f egy olyan függvény, hogy

- A inputjaiból B inputjait készíti
- választartó módon
- és logaritmikus térben kiszámítható

akkor f egy logtáras visszavezetés A-ról B-re.

5. Véges automata és változatai, a felismert nyelv definíciója. A reguláris nyelvtanok, a véges automaták

és a reguláris kifejezések ekvivalenciája. Reguláris nyelvekre vonatkozó pumpáló lemma, alkalmazása és következményei

Véges automata és változatai, a felismert nyelv definíciója

Lásd pdf

A reguláris nyelvtanok, a véges automaták és a reguláris kifejezések ekvivalenciája

Reguláris nyelvtanok

- N : nemterminális abc
- Σ : terminális abc
- P : szabályok halmaza
- S : eleme N , kezdő nemterminális Egy $G=(N, \Sigma, P, S)$ nyelvtan reguláris (vagy jobblinéáris), ha P -ben minden szabály
- $A \rightarrow xB$ vagy
- $A \rightarrow x$

alakú.

Azért jobblinéáris, mert minden szabály jobb oldalán max. egy nemterminális állhat, és ez mindig a szó végén helyezkedik el. Levezetést csak $A \rightarrow x$ alakú szabállyal fejezhetünk be, ahol x eleme Σ^* . A reguláris nyelvtanok speciális környezetfüggetlen nyelvtanok.

Példa: $S \rightarrow aaS|abS|baS|bbS|\epsilon$, vagyis a páros hosszú szavakat generáló nyelvtan.

Reguláris kifejezések

Vesünk egy abc -t, és hozzáveszünk néhány szimbólumot, ezekből építünk reguláris kifejezéseket.

A szigma feletti reguláris kifejezések halmaza a $(\Sigma \cup \{\emptyset, \epsilon, (,), +, *\})^*$ halmaz legszűkebb olyan U részhalmaza, hogy

- \emptyset, ϵ eleme U -nak
- minden a eleme Σ eleme U -nak
- ha R_1, R_2 eleme U , akkor R_1+R_2, R_1R_2, R_1^* is eleme U -nak

Prioritási sorrend: $*$, konkatenáció, $+$

Jelentések:

- $|R|$, az R által reprezentált nyelv
- $R = \emptyset, |R| = \emptyset$, azaz az üres nyelv

- $R = \epsilon$, $|R| = \{\epsilon\}$, azaz az epszilon szimbólum önmagában, mint nyelv
- $R = a$, $|R| = \{a\}$, azaz az a szimbólum önmagában, mint nyelv
- $R = R_1 + R_2$, $|R| = |R_1| \cup |R_2|$, azaz a két regex által generált nyelv uniója
- $R = R_1 R_2$, $|R| = |R_1| |R_2|$, azaz a két regex által generált nyelv konkatenációja
- $R = R_1^*$, akkor $|R| = |R_1|^*$, azaz a regex által generált nyelv iterációja, az összes szó összekonkatenálva egy másik nyelvbéli szóval az összes lehetséges módon

Ekvivalencia

Tetszőleges L részhalmaza szigmacsillag nyelv esetén a következő három állítás ekvivalens:

- 1. L generálható reguláris nyelvtannal
- 2. L felismerhető automatával
- 3. L reprezentálható reguláris kifejezéssel

3 \rightarrow 1

Ha L reprezentálható reguláris kifejezéssel, akkor generálható reguláris nyelvtannal.

Bizonyítás: L -et reprezentáló R reguláris kifejezés struktúrája szerinti indukcióval.

- $R = \emptyset$: ekkor $L = |R| = \emptyset$, ekkor L generálható a $G = (N, \Sigma, \emptyset, S)$ nyelvtannal.
- $R = a$: a eleme Σ -nak, vagy $a = \epsilon$, ekkor $L = |R| = \{a\}$, ami generálható a $G = (N, \Sigma, \{S \rightarrow a\}, S)$ nyelvtannal
- INDUKCIÓ $R = R_1 + R_2$, ekkor $L = |R| = L_1 \cup L_2$, $L_1 = |R_1|$, $L_2 = |R_2|$
 - ekkor tegyük fel, hogy L_i generálható egy $G_i = (N_i, \Sigma_i, P_i, S_i)$ ($i=1,2$) reguláris nyelvtannal
 - ekkor L generálható egy $G = (N_1 \cup N_2 \cup \{S\}, \Sigma, P_1 \cup P_2 \cup \{S \rightarrow S_1, S \rightarrow S_2\}, S)$ nyelvtannal, ahol S egy új szimbólum
 - aka vesszük az összes nemterminálist, az abc -t, az összes korábbi szabályt
 - továbbá egy új kezdőszimbólumot
 - és az új kezdőszimbólumból elérhető a régi kettő kezdőszimbólum, aka kiválaszthatjuk, melyik nyelvből származó szót akarjuk generálni
 - ahhoz, hogy elérhető legyen a régi két kezdőszimbólum, felvesszünk két új szabályt értelemszerűen a régiéik közé
- INDUKCIÓ $R = R_1 R_2$, ekkor $L = |R| = L_1 L_2$, $L_1 = |R_1|$, $L_2 = |R_2|$
 - ekkor tegyük fel, hogy L_i generálható egy $G_i = (N_i, \Sigma_i, P_i, S_i)$ ($i=1,2$) reguláris nyelvtannal
 - Akkor L generálható a $G = (N_1 \cup N_2, \Sigma, P, S_1)$ nyelvtannal, ahol P :
 - bele vesszük az összes szabályt az első nyelvet generáló nyelvtanból, a befejező szabályok végére odaírjuk a második nyelvtan kezdőszimbólumát
 - a második nyelvtan összes szabályát is bele vesszük
- INDUKCIÓ $R = R_1^*$, ekkor $L = |R| = L_1^*$, ahol $L_1 = |R_1|$
 - ekkor tegyük fel, hogy L_1 generálható egy $G_1 = (N_1, \Sigma, P_1, S_1)$ nyelvtannal
 - ekkor L generálható egy $G = (N_1 \cup \{S\}, \Sigma, P, S)$ nyelvtannal, ahol S egy új szimbólum
 - a szabályokat úgy módosítjuk, hogy:
 - S -ből elérhető az üresszó és az eredeti kezdőszimbólum
 - a nem "befejező" szabályokat felvesszük úgy, ahogy voltak
 - a "befejező" szabályok jobb oldalára odaírjuk az S -t

1 \rightarrow 2

Ha L nyelv reguláris, akkor felismerhető automatával.

Bizonyítás: legyen L egy reguláris nyelv, és $L=L(G)$, ahol G egy reguláris nyelvtan.

Minden $G=(N,\Sigma,P,S)$, reguláris nyelvtanhoz megadható vele ekvivalens $G'=(N',\Sigma,P',S)$ reguláris nyelvtan, úgy hogy P' -ben minden szabály $A \rightarrow B$, $A \rightarrow aB$ vagy $A \rightarrow \epsilon$ alakú, ahol $A,B \in N$ és $a \in \Sigma$.

Bizonyítás

- amelyik szabály már alpból ilyen alakú, azokat felvesszük P' -be
- az $A \rightarrow a_1a_2a_3\dots a_nB$ alakú szabályokat szétdaraboljuk
 - lesz belőle $A \rightarrow a_1A_1$, $A_1 \rightarrow a_2A_2 \rightarrow \dots \rightarrow A_{n-1} \rightarrow a_nB$
- az $A \rightarrow a_1a_2a_3\dots a_n$ szabályokat feladarboljuk
 - lesz belőle $A \rightarrow a_1A_1$, $A_1 \rightarrow a_2A_2 \rightarrow \dots \rightarrow A_n \rightarrow \epsilon$

az új nemterminálisokat felvesszük N -be

Minden olyan $G=(N,\Sigma,P,S)$ reguláris nyelvtanhoz, melynek csak $A \rightarrow B$, $A \rightarrow aB$ vagy $A \rightarrow \epsilon$ alakú szabályai vannak megadható olyan $M=(Q,\Sigma, \delta, q_0, F)$ nemdeterminisztikus ϵ -automata, amelyre $L(M) = L(G)$.

Bizonyítás

- $Q = N$, azaz a nemterminálisokból lesznek az állapotok
- $q_0 = S$, a kezdőszimbólumból lesz a kezdőállapot
- azokból a B nemterminálisokból lesz végállapot, amikből van $B \rightarrow \epsilon$ szabály
- minden $A \rightarrow aB$ kinézetű szabályból pedig legyen egy átmenet A -ból B -be a hatására

2 \rightarrow 3

Minden automatával felismert nyelv reprezentálható reguláris kifejezéssel. (Kleene tétele)

Bizonyítás: legyen $L=L(M)$, ahol M egy determinisztikus automata. Megadunk egy olyan reguláris kifejezést, ami L -et reprezentálja.

Tételezzük fel, hogy $Q=\{1,2,3,\dots,n\}$, és $q_0=1$. Minden 0 kisebbegyenlő k , azaz k darab állapot, és 0 kisebbegyenlő i, j kisebbegyenlő n esetén definiáljuk az $L^{(k)}_{i,j}$ nyelvet a következőképpen:

Nézzük úgy az automatát, mintha az i . állapotból indulnánk, és a j -be akarnánk eljutni, de csak az $\{1,\dots,k\}$ állapotokat érinthetjük. Az $L^{(k)}_{i,j}$ nyelv azokat a szavakat tartalmazza, amelyeket ez a "kisebb" automata felismer.

Ezután vegyük észre azt, hogy az L nyelv tulajdonképpen úgy áll elő, hogy venni kell az összes olyan $L^{(n)}_{1,j}$ nyelvet, ahol j egy végállapot! Tehát vegyük az összes olyan nyelvet, ahol az első állapotból akarunk elindulni, és az utolsóba eljutni, és használhatjuk az összes állapotát M -nek (mind az n darabot). Tehát ezen $L^{(n)}_{1,j}$ nyelvek uniója lesz L .

Ezért elég az $L^{(n)}_{1,j}$ -ket reprezentáló reguláris kifejezéseket megadnunk ($R^{(n)}_{1,j}$).

Ehhez meg kell adnunk az $R^{(k)}_{i,j}$ reguláris kifejezéseket k szerinti indukcióval.

$k=0$ azt jelenti, hogy 0 közbülső állapotból kell eljutnunk az i állapotból a j állapotba. Ez lehet úgy, hogy valamilyen szimbólum hatására átmegyünk, vagy ha $i=j$, akkor hurokkel helyben maradunk, vagy epszilonnal

nem csinálunk semmit.

Az indukciós feltevésünk az, hogy minden i, j -re megadtuk az $R^{(k)}_{i,j}$ -t

$k+1$ -hez ÉSZREVESSÜK (ja ugye tök triviális lmao)

$L^{(k+1)}_{i,j}$ egyenlő azzal, hogy

- vagy amúgyis eljutunk k köztes állapottal is i -ből j -be
- vagy bele vesszük a $k+1$. állapotot is a levesbe, elmegyünk az 1 -estől a $k+1$. állapotba, ott körözünk akármeddig, és utána $k+1$ -ből pedig eljutunk j -be

Ekkor, mivel az $L^{(k)}_{i,j}$ nyelvekhez az indukciós feltevés miatt tudtunk megfelelő regexet adni, ezekre elvégezve az előző azonosságot, megkapjuk az $R^{(k+1)}_{i,j}$ -t is, ezzel pedig meg tudjuk kapni az összes regexet, ami a kezdőállapotból a végállapotokba visz, ezeket uniózva pedig az egész nyelvhez tartozó regexet.

Reguláris nyelvekre vonatkozó pumpáló lemma, alkalmazása és következményei

Pumpáló lemma

Minden L reguláris nyelv esetén megadható egy L -től függő $k > 0$ szám, melyre minden w eleme L esetén, ha $|w|$ nagyobb egyenlő k , akkor van olyan $w = w_1 w_2 w_3$ felbontás, hogy

- $0 < |w_2|$ és $|w_1 w_2|$ kisebb egyenlő k
- minden n nagyobb egyenlő 0 -ra $w_1 w_2^n w_3$ eleme L

Fordítva, ha egy nyelvhez nem adható meg ilyen k szám, akkor a nyelv nem reguláris.

Kb a lényeg, hogy ha egy nyelv reguláris, akkor a k -nál hosszabb szavak felbonthatók három részre, és a középső rész ismétlődhet akármeddig

Alkalmazása

Pl bebizonyíthatjuk vele egy nyelvről, hogy nem reguláris. $a^n b^n \mid n \geq 0$ nyelv nem reguláris

tegyük fel, hogy van ilyen k , amivel felbontható a feltételek miatt w_2 -ben csak a betűk vannak ha ezt pumpáljuk, több a betű lesz benne, mint b - rossz

Következményei

Vannak olyan nyelvek, amelyek nem környezetfüggetlenek, de nem regulárisak, pl $a^n b^n \mid n \geq 0$.

6. A környezetfüggetlen nyelvtan és nyelv definíciója. Derivációk és derivációs fák kapcsolata.

Veremautomaták és környezetfüggetlen nyelvtanok ekvivalenciája. A Bar-Hillel lemma és alkalmazása

A környezetfüggetlen nyelvtan és nyelv definíciója

Egy $G=(N, \Sigma, P, S)$ nyelvtan, környezetfüggetlen, ha minden szabálya $A \rightarrow \alpha$ alakú, ahol α egy terminálisokból és nemterminálisokból álló szó.

Egy nyelv környezetfüggetlen, ha van olyan CF nyelvtan, ami őt generálja.

Derivációk és derivációs fák kapcsolata

Korlátozás nélküli deriváció

- bármely nemterminális helyére helyettesíthetünk

Bal/jobboldali deriváció

- csak a legbal/jobboldalabbi nemterminálisba helyettesíthetünk

Derivációs fák

Mindig csak egy gyökere van

- vagy csak a gyökekből áll
- vagy van egy epsilon gyerek
- vagy kiindul belőle k darab él, amelyek végpontjai további derivációs fák gyökerei

Legyen t egy derivációs fa, gyökere X t magassága $h(t)$ t határa $fr(t)$ - határ kb a levelek balról jobbra olvasva

Derivációs fák kapcsolata a derivációkkal

Tetszőleges X gyökerű derivációs fára és α szóra X -ből akkor vezethető le α , ha van olyan X gyökerű derivációs fa, amelyre $fr(t) = \alpha$

Veremautomaták és környezetfüggetlen nyelvtanok ekvivalenciája

Veremautomata

Veremautomatának nevezzük azt a $P = (Q, \Sigma, \Gamma, \delta, q_0, Z_0, F)$, ahol

- Q : állapotok halmaza
- Σ : input abc
- Γ : verem abc
- q_0 eleme Q : kezdőállapot
- Z_0 : verem kezdőszimbólum
- F : végállapotok halmaza
- δ : átmenetfüggvény

Az átmenet a következőképpen történik: ha az automata a q állapotban van, a szimbólum érkezik és Z van a verem tetején, akkor átmegy a q_i állapotba, a veremben pedig Z helyére α_i kerül. Az átmenetnél az automata kiolvass egy betűt az inputból, leveszi Z -t a verem tetejéről, és tetszőleges hosszú szót odaír a helyére.

Egy szó elfogadása történhet végállapotokkal, vagy üres veremmel is. Ugyanazon automatánál általában nem egyezik meg az üres veremmel és a végállapotokkal felismert nyelv.

Ekvivalencia

Minden környezetfüggetlen nyelvtanhoz meg lehet adni veremautomatát úgy, hogy a veremautomata (üres veremmel vagy végállapottal) ugyanazt a nyelvet ismeri fel, amit a környezetfüggetlen nyelvtan generál.

A bizonyításhoz egy környezetfüggetlen nyelvtanhoz konstruálunk egy egyállapotos nemdeterminisztikus veremautomatát, ami üres veremmel ismeri fel a szavakat. A veremabc legyen a nemterminálisok unió terminálisok. Ezzel a veremautomatával szimuláljuk a nyelvtan levezetéseit. Tudjuk továbbá, hogy az üres veremmel és a végállapotokkal felismert nyelvek halmaza ugyanaz, így ez az állításunk igaz lesz.

Minden veremautomatával felismert nyelv környezetfüggetlen.

Itt pedig veremautomatához adunk meg egy környezetfüggetlen nyelvtant.

Lásd pdf 2.

Bar-Hillel lemma és alkalmazása

Tulajdonképpen pumpáló lemma CF nyelvekre

Ha L egy környezetfüggetlen nyelv, akkor létezik egy nyelvtől függő k szám, amire ha egy L -beli szó hossza nagyobb k -nál, akkor feldarabolható 5 részre, amikre a következők teljesülnek:

- $|w_2w_3w_4| \leq k$
- w_2w_4 nem epszilon
- minden $n \geq 0$ -ra $w_1w_2^nw_3w_4^nw_5$ eleme L -nek

Alkalmazás: az $L = \{a^n b^n c^n \mid n \geq 1\}$ nyelv nem környezetfüggetlen.

Tegyük fel, hogy igen, ekkor léteznie kell olyan k számnak, amire teljesülnek a Bar-Hillel lemmában a feltételek.

Vegyük az $a^k b^k c^k$ szót, aminek hossza $3k \geq k$, tehát jó lesz fixen.

A lemma szerint ennek létezik $w_1w_2w_3w_4w_5$ felbontása, melyre w_2w_4 nem epszilon, és minden $n \geq 0$ -ra $w_1w_2^nw_3w_4^nw_5$ eleme a nyelvnek.

Nézzük ekkor mi lehet w_2 -ben és w_4 -ben! Egyik sem tartalmazhat két betűt, mert ekkor pl ha kétszer vesszük w_2 -t és w_4 -et, akkor a betűk sorrendje nem abc lesz. Tehát biztosan csak egyféle betűt tartalmaznak. Ekkor a $w_1w_2^2w_3w_4^2w_5$ szóbal legalább egy, és legfeljebb két betű száma több, mint a többi betűé, tehát biztos nem eleme ez a szó L -nek.

7. Eliminációs módszerek, mátrixok trianguláris felbontásai. Lineáris egyenletrendszerek megoldása iterációs módszerekkel. Mátrixok sajátértékeinek és sajátvektorainak numerikus meghatározása

Eliminációs módszerek

Gauss-elimináció

- $Ax=b$ alakú lineáris egyenletrendszerek megoldásához tudjuk használni
- az $Ax=b$ egyenletrendszernek pontosan akkor van egy megoldása, ha $\det(A)$ nem 0
- ekkor $x = A^{-1}b$
 - de az inverzet kiszámolni túl lassú lenne

A Gauss-eliminációval az A mátrixot felső háromszögmátrixszá alakítjuk, és ha ez sikerül, akkor abból visszahelyettesítésekkel megkaphatjuk x -et. Műveletigénye $O(n^2/2)$.

A felső háromszögmátrixot ún. eliminációs mátrixok segítségével kapjuk meg. Egy eliminációs mátrix dolga, hogy kinullázza az A mátrix egyik oszlopában a főátló alatti elemeket. Ha az összes ilyen eliminációs mátrixot összeszorozzuk balról egymással, akkor kapjuk az M mátrixot. Ekkor az MA szorzás eredménye lesz a kívánt felső trianguláris mátrix.

LU felbontás

Az LU felbontás lényege, hogy az A mátrixot egy alsó és egy felső háromszögmátrixra bontjuk. A Gauss eliminációhoz nagyon hasonlít, ott az MA szorzás eredménye egy U felső trianguláris mátrix volt. Ha mindkét oldalt megszorozzuk balról M^{-1} -gyel, akkor azt kapjuk, hogy $A = M^{-1}U$. Legyen $M^{-1}=L$, mert M^{-1} egy alsó trianguláris mátrix. Ezzel elvégeztük az A mátrix LU felbontását.

Ekkor az $Ax=b$ egyenletrendszer megoldását a következőképpen kaphatjuk:

- $Ax=b$
- $LUx=b$
- $y = Ux$
- $Ly=b$
- $y = L^{-1}b$
- $L^{-1}b = Ux$

Cholesky felbontás

Ha az A mátrix

- szimmetrikus
- pozitív definit
 - minden sajátértéke pozitív

akkor felbontható a következőképpen: $A=LL^T$, tehát U az most pont L transzponáltja lesz.

QR felbontás

Ha az A mátrix

- négyzetes
- valós
- reguláris ($\det(A)$ nem 0)

akkor létezik QR felbontása.

Q egy úgynevezett ortogonális mátrix (és négyzetes is). Ez azt jelenti, hogy $Q^TQ = QQ^T = I$. R egy felső háromszögmátrix

Bizonyítás: $A^T A$ pozitív definit, így létezik $R^T R$ Cholesky felbontása.

Legyen ekkor Q egyenlő $A^T R^{-1}$ -gyel.

Igazoljuk, hogy Q ortogonális.

$Q^T Q = (A^T R^{-1})^T (A^T R^{-1}) = (R^{-1})^T A^T A R^{-1} = (R^{-1})^T R^T R R^{-1} = I^T I = I$ behelyettesítés
transzponálásos azonosság $A^T A = R^T R$ inverzek kiütik egymást

Tehát Q valóban ortogonális.

Lineáris egyenletrendszerek megoldása iterációs módszerekkel

Jacobi iteráció

Átrendezzük úgy az egyenletrendszert, hogy a baloldalon egy-egy változót kifejezünk

Választunk valami indulóvektort, ami ilyen kezdő megoldás kb A vektor elemeit behelyettesítjük a jobboldalra, és ebből kapunk egy új vektort a baloldalon, ezzel folytatjuk.

Csak akkor konvergál, ha a mátrix *szigorúan diagonálisan domináns*, vagyis az összes főátlóbeli elem abszolút értéke a legnagyobb az adott sorban.

Gauss-Seidel iteráció

Ugyanaz, mint a Jacobi, csak ha már egy változó új értékét kiszámoltuk, akkor a következő sorokban már azt az új értéket használjuk.

Mátrixok sajátértékeinek és sajátvektorainak numerikus meghatározása

Sajátérték, sajátvektor

$$Ax = \lambda x$$

x a sajátvektor, λ a sajátérték

A sajátérték olyan szám, amivel ha megszorozzuk a hozzá tartozó sajátvektort, akkor ugyanazt az eredményt kapjuk, mintha azt a vektort a mátrixszal szoroztuk volna meg.

Meghatározása: $\det(A - \lambda I) = 0$ tehát, a főátló minden eleméből kivonunk λ -t, és ennek a mátrixnak keressük a determinánsát ez egy polinomot fog eredményezni, amiben λ -k a változók, és ennek a polinomnak a gyökei lesznek a sajátértékek

Ezt a polinomot nevezzük a mátrix *karakterisztikus polinomjának*. Valós mátrixnak is lehetnek komplex sajátértékei!

A mátrix sajátértékeinek a halmazát a mátrix *spektrumának* hívjuk.

Hatványmódszer

A hatványmódszer a legnagyobb abszolútértékű sajátérték meghatározására szolgál. Iterációs módszer.

$$y^{(k)} = Ax^{(k)} \quad x^{(k+1)} = y^{(k)} / \|y^{(k)}\|$$

a kiindulási x vektor ne legyen a nullvektor, és nem lehet merőleges a legnagyobb abszolútértékű sajátértékhez tartozó sajátvektorra.

A k betűk a kitevőben a k . iterációt jelentik, nem k . hatványt.

Inverz hatványmódszer

$$A y = x^k, x^{(k+1)} = y / \|y\|$$

Az inverz hatványmódszer azon a felismerésen alapul, hogy ha az A mátrix sajátértéke λ , és a hozzá tartozó sajátvektor x , akkor A^{-1} egy sajátértéke λ^{-1} , és a hozzá tartozó sajátvektor x .

8. Érintő, szelő, és húr módszer, a konjugált gradiens eljárás. Lagrange interpoláció. Numerikus integrálás

Érintő, szelő, húrmódszer, konjugált gradiens eljárás

Mindegyik egyváltozós függvény zérushelyét keresi, iterációs módszerrel.

Érintőmódszer

Más néven Newton-módszer

$f(x)=0$ egyenlet zérushelyét keressük, ez legyen x^*

Ennek egy környezetében, ha $f(x)$ differenciálható, válasszunk ebből a környezetből egy kezdőértéket

Az iteráció, amit használunk:

$$x_{k+1} = x_k - f(x_k) / f'(x_k)$$

Magyarul, a következő megoldást úgy kapjuk, hogy az előző megoldásból kivonjuk a függvény x_k helyen felvett értékének és a függvény deriváltjának az x_k pontban felvett értékének a hányadosát.

Ha az $f(x)$ függvény kétszer folytonosan differenciálható az x^* egy környezetében, akkor van olyan pont, ahonnan indulva a Newton-módszer kvadratikusan konvergens sorozatot ad meg, aka gyorsan konvergál a megoldáshoz.

$$|x^* - x_{k+1}| \leq C |x^* - x_k|^2$$

Szelőmódszer

Legyen megint x^* az $f(x)=0$ egyenlet egyszeres gyöke, és megint ezt keressük iterációval.

A függvény deriváltját nem mindig tudjuk, de a függvényt ki tudjuk értékelni minden helyen. Ekkor f' helyett használhatjuk az $(f(x_k) - f(x_{k-1})) / (x_k - x_{k-1})$ képletet.

Ekkor f' helyére a felső képletet behelyettesítve megkapjuk a szelőmódszer iterációs képletét:

$$x_{k+1} = x_k - f(x_k) * (x_k - x_{k-1}) / (f(x_k) - f(x_{k-1}))$$

Azért szelőmódszer a neve, mert x_{k+1} az az $(x_k, f(x_k))$ és $(x_{k-1}, f(x_{k-1}))$ pontokon átmenő egyenes és az x tengely metszéspontjának koordinátája.

Olyan x_0, x_1 kezdőértékekkel szokás indítani, amelyek közrefogják a gyököt, amit keresünk.

Húrmódszer

A szelőmódszer egy változata.

Feltesszük, hogy a kezdeti x_0, x_1 pontokban az $f(x)$ függvény ellentétes előjelű, és $f(x_{k+1})$ függvényében a megelőző két pontból azt választjuk, amivel ez a tulajdonság fennmarad.

Konjugált gradiens eljárás

Szimmetrikus, pozitív definit mátrixú lineáris egyenletrendszerek megoldására alkalmas. Pontos számolásokkal véges sok lépésben megtalálná a megoldást, de a kerekítési hibák miatt iterációs eljárásnak veszik.

$q(x) = 1/2x^T Ax - x^T b$ kvadratikus függvény minimumpontját keressük, mert ez ugyanaz, mint az eredeti egyenletrendszerünk megoldása, ha létezik.

Úgy keressük a következő közelítő megoldást, hogy van egy keresési irányunk, és egy lépésközünk, és az aktuális pontból lépünk ebbe az irányba ekkora lépésközzel egyet.

A negatív gradiensvektort nevezzük reziduális vektornak (erre csökken a függvényünk). Ez lesz $r = b - Ax$. A keresési irányban ott lesz a célfüggvény minimális ahol az új reziduális vektor merőleges az előző keresési irányra, szóval tudjuk pontosan, hogy hova kell lépnünk az adott irányban.

Tehát a konjugált gradiens módszer:

- meghatározzuk a lépéshosszt
- meghatározzuk az új közelítő megoldást (lépünk egyet az előző megoldásból az adott irányba az új lépéshosszal)
- ebből kiszámoljuk az új reziduális vektort
- és az új keresési irányt
- és kezdjük előlről

A megállási feltételünk lehet az, hogy az utolsó néhány iterált közelítés eltérése és a reziduális vektorok eltérése bizonyos kicsi határ alatt maradtak.

Lagrange interpoláció

Függvényközelítési módszer. Van pár alappontunk, és ezekre szeretnénk egy polinomot illeszteni. Ezek az alappontok legyenek páronként különbözőek.

Minden pontra felírunk egy egyenletet. Ahány alappontunk van, max annyiad fokú lesz a kapott polinomunk. Az egyenlet úgy fog kinézni, hogy ismerjük az x_i értéket, és mindenhova behelyettesítjük őket, és ezeknek az x_i^1, x_i^2 , stb változóknak keressük az együtthatóját. Az egyenlet jobb oldalán pedig az $f(x_i)$ értékek vannak.

Ebből kapunk egy lineáris egyenletrendszert, ahol az együtthatókat keressük. Ennek az egyenletrendszernek a mátrixa egy Vandermonde-mátrix lesz. Ebből következik, hogy pontosan egy polinom létezik, ami az adott

pontokon áthalad.

A Lagrange-interpoláció az interpoláló polinomot a $\sum f(x_i)L_i(x)$ alakban adja meg. $L_i(x)$ -et úgy kapjuk, hogy egy nagy törtet veszünk - a számlálóban összeszorozzuk az összes $x-x_j$ -t, ahol j nem egyenlő i -vel, tehát $x-x_i$ szorzó kimarad belőle. A nevezőben pedig x_i-x_j -ket szorzunk össze, mindenhol, ahol j nem egyenlő i -vel szintén (különben nullával osztanánk).

Numerikus integrálás

Határozott integrálokat akarunk közelíteni, úgynevezett kvadratúra formulákkal.

$Q_n(f)$ -fel jelöljük, $Q_n(f) = \sum_{i=1}^n w_i \cdot f(x_i)$

Általában feltesszük, hogy az összes x_i az $[a,b]$ intervallumban van, ugye ebben az intervallumban keressük a határozott integrálját f -nek. A w_i számokat pedig súlyoknak hívjuk. Homogén és additív leképezés, azaz két függvény összegének a határozott integrálja a két függvény határozott integráljának az összege, és egy függvény számszorosának határozott integrálja a függvény határozott integráljának számszorosa.

A határok szerinti additivitás fontos tulajdonság, tehát pl integrál a -tól b -ig az ugyanaz mint integrál a -tól c -ig plusz integrál c -tól b -ig, ahol $a < c < b$

A kvadratúra-formula hibája a határozott integrál mínusz a kvadratúra formula kifejezéssel definiáljuk. Ha ez nulla, akkor pontos a kvadratúra formula.

Kvadratúra formula pontossági rendje az r természetes szám, ha az pontos az $1, x, x^2, x^3, \dots, x^r$ hatványfüggvényekre, de nem pontos x^{r+1} -re. A rend meghatározása ekvivalens egy egyenletrendszer megoldásával. Ha az alappontokat (tehát x_1, x_2 , stb) ismeretlennek tekintjük, akkor ez egy $r+1$ egyenletből álló egyenletrendszer (mert elmegyünk x^r -ig, plusz az x^0 , azaz 1), amiben $2n$ változó van (n súly és n darab x).

Az n alappontos kvadratúra formula rendje legfeljebb $2n-1$ lehet.

10. Normálformák a predikátumkalkulusban. Egyesítési algoritmus. Következtető módszerek: Alap rezolúció, elsőrendű rezolúció

Normálformák predikátumkalkulusban

Prenex alak

- elimináljuk a nyilakat
- kiigazítjuk a formulát (változókat átnevezzük, ha van változónév-ütközés)
- az összes kvantort kihozzuk a formula elejére, ha páratlan negálás scope-jában volt, akkor fordul, ha páros, nem

Skolem alak

- prenex alak

- a létezik kvantorhoz tartozó változókat lecseréljük új függvényekre, amik az előtte álló bármely-kvantált változóktól függenek

Zárt Skolem alak

- Skolem alak
- a szabad változókat lecseréljük konstansokra, pl minden x helyére c_x -et írunk

Egyesítési algoritmus

Ha F egy formula, akkor $F[x/t]$ azt jelenti, hogy F -ben x összes előfordulását helyettesítjük t -vel.

Ha x_1, x_2, \dots, x_n változók, és t_1, \dots, t_n termek, akkor az $[x_1/t_1], \dots, [x_n/t_n]$ helyettesítés azt jelenti, hogy először x_1 helyére írunk t_1 -et, aztán az eredményben x_2 helyére t_2 -t, stb.

Formulák halmazaira, pl klózokra is értelmezhetjük ezt.

Klóz végzett helyettesítésnél $[x/t]$ azt jelenti, hogy minden klózra elvégezzük az x helyére t helyettesítést, és az eredményeket visszapakoljuk egy halmazba. Ha $C = \{I_1, I_2, \dots, I_n\}$ literálok halmaza, akkor s a c egyesítője, ha $I_1 * s = \dots = I_n * s$. C -re akkor mondjuk, hogy egyesíthető, ha van egyesítője.

Az s helyettesítés általánosabb az s' helyettesítésnél, ha van olyan s'' helyettesítés, hogy $s * s'' = s'$.

Egyesítési algoritmus:

- input: C klóz
- output: C legáltalánosabb helyettesítője, ha egyesíthető, különben azzal tér vissza, hogy nem egyesíthető
- veszünk két literált, és keressük az első eltérést
- ha az egyik helyen egy x változó áll, a másikon egy t term, amiben nincs x , akkor x/t és vissza az előző pontra
- különben return nem egyesíthető

Nem egyesíthető pl

- ha $f(x)$ és c a különbség a két literál azonos pontján
- ha x és $f(x)$ a különbség
- ha $g(x)$ és $f(x)$ a különbség

Alaprezolúció

- input: elsőrendű formulák egy szigma halmaza
- output: kielégíthetetlen véges sok lépésben, vagy kielégíthető véges sokban vagy végtelen ciklus
- szigma elemeit zárt skolem alakra hozzuk, a formula belsejét pedig CNF-re, ez legyen szigma'
- ekkor $E(\text{szigma}')$ a klózok alappéldányainak a halmaza
- $E(\text{szigma}')$ -n futtatjuk az ítétekalkulusbeli rezolúciós algoritmust
- $E(\text{szigma}')$ általában végtelen
- vegyük fel $E(\text{szigma}')$ egy elemét, és rezolváljunk vele, amíg lehet
- ha kijön az üres klóz, akkor jók vagyunk, ha nem, generálunk tovább

Elsőrendű rezolúció

- input: elsőrendű formulák egy szigma halmaza
- output: kielégíthetetlen-e?
- szigma zárt skolemre, mag cnfre, szigma'
- szigma' elemeit közvetlenül felvehetjük a listára
- ha kijön az üres klóz, kielégíthetetlen
- ha nem tudunk több klózt levezetni, kielégíthető

Rezolvensképzés:

- C1 és C2 klózekat akarjuk rezolválni
- átnevezzük a változókat úgy, hogy ne legyen közös változó C1-ben és C2-ben
- kiválasztunk C1-ből és C2-ből is literálokat, az egyikből pozitívokat, a másikkból negatívokat
- ezeket pozitívan belepakoljuk egy C halmazba
- ha C egyesíthető egy s helyettesítéssel, akkor vehetjük a rezolvensét C1-nek és C2-nek
- elmentjük s-t
- vesszük C1-ből és C2-ből a maradék literálokat, és berakjuk egy halmazba
- ezen a halmazon elvégezzük az s helyettesítést, ez lesz a rezolvens

9. Normálformák az ítéletkalkulusban, teljes rendszerek. Következtető módszerek: Hilbert-kalkulus és rezolúció

Normálformák az ítéletkalkulusban

Diszjunktív normálforma

A formula olyan alakja:

- a változók pozitívan vagy negatívan szerepelhetnek benne
- a zárójelekben lévő pozitív vagy negatív változók között éselés van
- a zárójelek között vagyolás van

Nyílmentes formula

A nyilakat elimináljuk a formulából a következő szabályok alkalmazásával:

- $F \rightarrow G \equiv \neg F \vee G$
- $F \leftrightarrow G \equiv (F \rightarrow G) \wedge (G \rightarrow F) \equiv (\neg F \vee G) \wedge (\neg G \vee F)$

NNF

A negációkat bevisszük teljesen a változók elé, hogy semmilyen zárójeles kifejezés előtt ne szerepeljen negáció. Ez a formula már nyílmentes is. Ehhez a De Morgan szabályokat alkalmazzuk:

- $\neg(F \vee G) \equiv \neg F \wedge \neg G$
- $\neg(F \wedge G) \equiv \neg F \vee \neg G$

CNF

A CNF alakban klózik vannak, és a klózik vannak összeésselve egymással. Egy klózikban változóik vannak, negatíván vagy pozitíván, és ezek között vagyolás van. Úgy kapjuk, hogy egy már NNF-ben lévő formulában alkalmazzuk a disztribúciós szabályt:

- $(F \&\& G) \parallel H == (F \parallel H) \&\& (G \parallel H)$
- $(F \&\& G) \parallel (H \&\& I) == (F \parallel H) \&\& (F \parallel I) \&\& (G \parallel H) \&\& (G \parallel I)$

Teljes rendszerek

Logikai műveletek egy rendszerét akkor nevezzük teljesnek, ha egy, már korábban teljesnek ítélt rendszer minden műveletét ki tudjuk fejezni ezen műveletekkel. A $\{-, \&\&, \parallel\}$ rendszer teljes, mert minden formulát CNF alakra tudunk hozni. Ezek alapján teljes még:

- $\{-, \&\&\}$
- - A negáció okés, az éselés okés, a vagyolást ki tudjuk fejezni:
- - $p \parallel q == \neg(\neg p \&\& \neg q)$
- $\{-, \parallel\}$
- - A negáció okés, a vagyolás okés, az éselést ki tudjuk fejezni:
- - $p \&\& q == \neg(\neg p \parallel \neg q)$

A $\{-, \rightarrow\}$ rendszer is teljes, mert tudjuk, hogy a $\{-, \parallel\}$ rendszer teljes, és ki tudjuk fejezni a műveleteit:

- negáció okés, vagyolás:
- - $p \parallel q == (\neg p) \rightarrow q$

A $\{-, \rightarrow, \text{lenyíl}\}$ rendszer is teljes, mert tudjuk, hogy a $\{-, \rightarrow\}$ rendszer teljes, és ki tudjuk fejezni a műveleteit:

- nyíl okés
- $\neg p == p \rightarrow \text{lenyíl}$

Ezt a rendszert nevezzük Hilbert rendszerének.

Rezolúció

A rezolúciónál a formuláink CNF alakban vannak. A rezolúcióval logikai következményeket tudunk bebizonyítani, pl. hogy egy formulahalmaznak logikai következménye egy formula.

Alapból a logikai következmény azt jelenti, hogy azoknak az értékadásoknak a halmaza, amelyek kielégítik a jobboldali formulá(ka)t, részhalmaza a jobboldali formulákat kielégítő értékadások halmazának. Ezzel az a baj, hogy az összes ilyen értékadást megtalálni nagyon hosszadalmas.

A rezolúciós algoritmus inputja klózikoknak egy halmaza, és outputja egy igen vagy egy nem, attól függően, hogy kielégíthető vagy kielégíthetetlen ez a klózhalmaz. A baloldali formulák közé felvesszük először a jobboldali formula negáltját, hiszen ha az így kapott új formulahalmaz kielégíthetetlen (azaz $\text{Mod}(\Sigma)$ üreshalmaz), akkor az eredeti logikai következmény fennáll.

Ezután listát vezetünk a klózikokról. Egy klóz felkerülhet a listára, ha

- eleme a Szigmának
- két, korábban már a listán szereplő klóz rezolvense

Két klóznak akkor vehetjük a rezolvensét, ha a mindkettőben szerepel ugyanaz a változó, de az egyikben negatívan, a másikban pedig pozitívan. Ekkor a rezolvens egy olyan klóz lesz, ahol ez a változó már nem fog szerepelni, hanem csak a két klózban maradt összes többi változó.

Ha a listára valamelyik lépésben rákerül az üresklóz, az azt jelenti, hogy Szigma kielégíthetetlen, vagyis az eredeti logikai következmény fennáll. Ha sehogy sem tudjuk levezetni az üresklózt, az azt jelenti, hogy a Szigma kielégíthető, és az eredeti logikai következmény nem áll fenn.

Hilbert-kalkulus

A Hilbert-kalkulusban Hilbert rendszerét használjuk. Az ilyen alakú formulákra is tudunk következtető rendszert építeni. A továbbiakban a formuláink mind Hilbert rendszeréből származnak.

A következtető rendszerünkben az input szintén egy formulahalmaz, illetve egy formula, amiről be akarjuk látni, hogy logikai következménye a formulahalmazunknak.

Ekkor a formulákról szintén listát vezetünk, ahol a listára felkerülhet egy formula, ha:

- benne van a Szigmában
- axiómapéldány
- modus ponense két, korábban a listán szereplő formulának

Háromféle axiómánk van: $Ax1: (F \rightarrow (G \rightarrow H)) \rightarrow ((F \rightarrow G) \rightarrow (F \rightarrow H))$ $Ax2: F \rightarrow (G \rightarrow F)$ $Ax3: ((F \rightarrow \text{lenyíl}) \rightarrow \text{lenyíl}) \rightarrow F$

Két formulának vehetjük a modus ponensét, ha az egyik formula F , a másik pedig $F \rightarrow G$ alakú. Ekkor a modus ponens pontosan G lesz.

Ezekkel a szabályokkal ha a listára kerül a logikai következmény jobb oldalán szereplő formula, akkor igazoltuk a logikai következményt.

Érdemes még az algoritmus előtt a dedukció művelettel kezdeni. Ha a jobb oldali formula $F \rightarrow G$ alakú, akkor F -et átvehetjük a Szigmába, és ezt mindaddig ismételtethetjük, amíg a jobb oldal ilyen alakú.

11. Keresési feladat: feladatrepresentáció, vak keresés, informált keresés, heurisztikák. Kétszemélyes zéró összegű játékok: minimax, alfa-béta eljárás. Korlátozás kielégítési feladat

Keresési feladat: feladatrepresentáció, vak keresés, informált keresés, heurisztikák

Feladatrepresentáció

Tekintsünk egy diszkrét, statikus, determinisztikus és teljesen megfigyelhető feladatkörnyezetet. Tegyük fel, hogy a világ tökéletesen modellezhető a következőkkel:

- lehetséges állapotok halmaza

- egy kezdőállapot
- lehetséges cselekvések halmaza (állapotátmenet függvény, minden állapothoz hozzárendelünk egy (cselekvés, állapot) rendezett párokból álló halmazt, tehát egy állapotban milyen cselekvések hatására milyen állapotba juthat az ágensünk)
- állapotátmenet költségfüggvénye, minden lehetséges állapot-cselekvés-állapot hármashoz hozzárendelünk egy költséget, azaz egy állapotból egy (másik) állapotba jutásnak mekkora a költsége
- célállapotok halmaza, tehát hova szeretnénk, hogy eljusson az ágensünk

Ez egy súlyozott gráfot definiál, ez a gráf az állapottér

Feltesszük továbbá, hogy az állapotok száma véges, vagy megszámlálható. Úton állapotok cselekvésekkel összekötött sorozatát értjük, ennek van egy összköltsége is.

Vak (informálatlan) keresés

Fakeresés

Adott kezdőállapotból találjunk minimális költségű utat egy célállapotba. Az állapottér nem mindig adott explicit módon, és végtelen is lehet.

Ötlet: keresőfa építése, a kezdőállapotból növekszünk fát a szomszédos állapotok hozzávételével, amíg célállapotot nem találunk. A keresőfa NEM azonos a feladat állapottérével, pl ha van két csúcs között oda-vissza él.

```
fakeresés 1 perem = { újcsúcs(kezdőállapot) } 2 while perem.nemüres() 3 csúcs = perem.elsőkivesz() 4 if csúcs.célállapot() return csúcs 5 perem.beszúr(csúcs.kiterjeszt()) 6 return failure
```

A csúcs.kiterjeszt() létrehozza a csúcsból elérhető összes állapotból a keresőfa csúcsot. A perem egy prioritási sor, ettől függ a bejárési stratégia.

A hatékonyságot növelhetjük, ha úgy szúrunk be csúcsokat a perembe, hogy abban az esetben, ha a peremben található már ugyanazzal az állapottal egy másik csúcs, akkor ha az új csúcs költsége kisebb, lecseréljük a régi csúcsot az újra, különben nem tesszük bele az újat.

Szélességi keresés

Fakeresés, ahol a perem egy FIFO perem.

- Teljes, minden, véges számú állapot érintésével elérhető állapotot véges időben elér
- Általában nem optimális, de pl akkor igen, ha a költség a mélység nem csökkenő függvénye
- időigény = tárigény $O(b^{d+1})$
 - b: szomszédok maximális száma
 - d: a legkisebb mélységű célállapot mélysége

Mélységi keresés

Fakeresés, LIFO perem

- Teljes, ha a keresési fa véges mélységű
- Nem optimális

- Időigény: legrosszabb esetben $O(b^m)$ (nagyon rossz, lehet végtelen), tárigény legrosszabb esetben $O(bm)$ (ez egész bízható)

Iteratíván mélyülő keresés

Mélységi keresések sorozata 1, 2, 3 stb mélységre korlátozva, amíg célállapotot nem találunk.

- Teljesség és optimalitás a szélességivel egyezik meg
- időigény = $O(b^d)$ (akár jobb is lehet, mint a szélességi), tárigény = $O(bd)$ (jobb, mint a mélységi)

Ez a legjobb informálatlan kereső.

Egyszerű költségű keresés

A peremben a rendezés költség alapú, mindig először a legkisebb útköltségű csúcsot terjesztjük ki.

- Teljes és optimális, ha minden él költsége nagyobb mint nulla
- Idő és tárigény nagyban függ a költségfüggvénytől

Gráfkeresés

Ha nem fa az állapottér!

Fakeresés, de a perem mellett még tárolunk egy ún. zárt halmazt is. A zárt halmazba azok a csúcsok kerülnek, amiket már kiterjesztettünk. A perembe helyezés előtt minden csúcsra megnézzük, hogy már a zárt halmazban van-e. Ha igen, nem tesszük a perembe. Másrészt minden peremből kivett csúcsot a zárt halmazba teszünk. Így minden állapothoz a legelső megtalált út lesz tárolva.

Informált keresés, heurisztikák

Itt már tudjuk, hogy "hova megyünk".

Heurisztika: minden állapotból megbecsüli, hogy mekkora az optimális út költsége az adott állapotból egy célállapotba: tehát értelmesebben tudunk következő szomszédot választani. Pl. légvonalbeli távolság a cél és a térképen egy útvonal-tervezési problémához jó heurisztika.

$h(n)$: optimális költség közelítése a legközelebbi célállapotba $g(n)$: tényleges költség a kezdőállapotból a jelenlegi állapotba

Mohó

Fakeresés, peremben a rendezést $h()$ alapján csináljuk, mindig a legkisebb értékű csúcsot vesszük ki.

- Teljes, de csak ha a keresési fa véges mélységű
- Nem optimális
- időigény, tárigény $O(b^m)$

A*

A peremben a rendezést $f()=h()+g()$ alapján végezzük, a legkisebb csúcsot vesszük ki. $f()$ a teljes út költségét becsüli a kezdőállapotból a végállapotba. Ha $h = 0$, és gráfkeresést alkalmazunk, akkor a Dijkstra-t kapjuk.

Egy h heurisztika elfogadható, ha nem ad nagyobb értéket, mint a tényleges optimális érték. Fakeresést feltételezve, ha h elfogadható és a keresési fa véges, akkor A^* optimális.

Egy h heurisztika konzisztens, ha $h(n) \leq$ mint a valódi költség n egyik bármely, plusz a szomszéd heurisztikája. Gráfkeresést feltételezve, ha h konzisztens és az állapottér véges, akkor A^* optimális.

Az A^* optimálisan hatékony, de a tárigénye általában exponenciális, és nagyon nagyban függ h -től. Az időigény szintén nagyon nagyban függ h -től.

Heurisztikák

A relaxált probléma optimális megoldása pl jó heurisztika lehet.

Relaxált probléma: elhagyunk feltételeket az eredeti problémából. Kombinálhatunk több heurisztikát is. Készíthetünk mintaadatbázisokat, ahol részproblémák egzakt költségét tároljuk.

Kétszemélyes zero összegű játékok: minimax, alfa-béta eljárás

Kétszemélyes, lépésváltásos, determinisztikus, zero összegű játék

- lehetséges állapotok halmaza
- egy kezdőállapot
- lehetséges cselekvések halmaza, és egy állapotátmenet függvény
- célállapotok
- hasznosságfüggvény

Két ágens van, felváltva lépnek. Az egyik maximalizálni akarja a hasznosságfüggvényt (MAX játékos), a másik minimalizálni (MIN játékos). Konvenció szerint MAX kezd. Az első célállapot elérésekor a játéknak definíció szerint vége.

Zero összegű játék: A MIN játékos minimalizálja a hasznosságot, ami ugyanaz, mint maximalizálni a negatív hasznosságot. Ez a negamax formalizmus. Itt a két játékos nyereségének az összege a végállapotban mindig nulla, innen a zero összegű elnevezés.

Minimax algoritmus, alfa-béta vágás

Mindkét játékos ismeri a teljes játékgráfot, bármilyen komplex számítást képes elvégezni és nem hibázik (tökéletes racionalitás). A minimax algoritmus alapján lehet megvalósítani a legjobb stratégiát tökéletes racionalitás esetén.

Minimax:

```
maxÉrték(n) 1 if végállapot(n) return hasznosság(n) 2 max = -végtelen 3 for a in n szomszédai 4 max = max(max, minÉrték(a)) 5 return max
```

```
minÉrték(n) 1 if végállapot(n) return hasznosság(n) 2 min = +végtelen 3 for a in n szomszédai 4 min = min(min, maxÉrték(a)) 5 return min
```

Ha n végállapot, visszaadja a hasznosságát. Különböző a max-nál n szomszédaira kiszámolja a maximális értéket, ami vagy az aktuális maximum, vagy nézi, hogy a másik játékos mit lépne. Csak elméleti jelentőségű, a

minimax algoritmus nem skálázódik. Az összes lehetséges állapot kiszámolása rettentő sok idő lenne pl sakknál.

Alfa-béta vágás

Ha tudjuk, hogy pl MAX-nak már van egy olyan stratégiája, ahol biztosan egy 10 értékű hasznosságot el tud érni az adott csúcsban, akkor a csúcs további kiértékelésekor nem kell vizsgálni olyan csúcsokat, ahol MIN ki tud kényszeríteni ≤ 10 hasznosságot, mert ennél már MAX-nak van jobb stratégiája

minÉrték és maxÉrték hívásakor átadjuk az alfa és béta paramétereket is a függvénynek.

Alfa jeletése: MAXnak már felfedeztünk egy olyan stratégiát, amely alfa hasznosságot biztosít, ha ennél kisebbet találunk, azt nem vizsgáljuk. Béta jelentése: MINnek már felfedeztünk egy olyan stratégiát, amely béta hasznosságot biztosít, ha ennél nagyobbakat találunk, azt nem vizsgáljuk

A gyakorlatban a minimax és az alfa-béta vágásos algoritmusokat is csak meghatározott mélységig vizsgáljuk, illetve heurisztikákat is alkalmazhatunk. A csúcsok bejárásának sorrendje is nagyon fontos, mert pl alfa béta vágásnál egy jó rendezés mellett nagyon sok csúcsot vághatunk le.

Korlátozás kielégítési feladat

A feladat az állapottérrel adott keresési problémák és az optimalizálási problémák jellemzőit ötvözi. Az állapotok és célállapotok speciális alakúak.

Lehetséges állapotok halmaza: a feladat állapotai az n db változó lehetséges kombinációi. Célállapotok: a megengedett állapotok, adottak különböző korlátozások, és azok az állapotok a célállapotok, amik minden korlátozást kielégítenek.

Az út a megoldásig lényegtelen, és gyakran célfüggvény is értelmezve van az állapotok felett, ilyenkor egy optimális célállapot megtalálása a cél.

12. Teljes együttes eloszlás tömör reprezentációja, Bayes hálók. Gépi tanulás: felügyelt tanulás problémája, döntési fák, naiv Bayes módszer, modellillesztés, mesterséges neuronhálók, k-legközelebbi szomszéd módszere

Teljes együttes eloszlás tömör reprezentációja, Bayes hálók

Teljes együttes eloszlás

Minden lehetséges eseményre tudjuk annak a valószínűségét. Pl van 3 logikai típusú véletlen változónk, akkor összesen $2^3=8$ -féle eset lehet ezekre. A teljes együttes eloszlásnál mind a 8 esetnek tudjuk a valószínűségét.

Tömör reprezentáció

A kijelentések függetlensége a legfontosabb tulajdonság a teljes együttes eloszlás tömöríthetőségéhez. Van függetlenség, és feltételes függetlenség.

Függetlenség a és b kijelentések függetlenek, ha $P(a \text{ és } b) = P(a) \cdot P(b)$

A függetlenség struktúrát takar. Ha pl n logikai változónk van, amik két független részhalmazra oszthatók m és k mérettel, akkor a 2^n valószínűség tárolása helyett elég $2^m + 2^k$ valószínűséget tárolni, ami sokkal kevesebb lehet.

Extrém esetben, ha pl. az A_1, \dots, A_n diszkrét változók kölcsönösen függetlenek (tetszőleges két részhalmaz független), akkor csak $O(n)$ értéket kell tárolni, mivel ez esetben

$$P(A_1, \dots, A_n) = P(A_1) \dots P(A_n)$$

Feltételes függetlenség a és b kijelentések feltételesen függetlenek c feltevésével, akkor és csak akkor, ha $P(a \text{ és } b \mid c) = P(a \mid c) \cdot P(b \mid c)$. Tipikus eset, ha a és b közös oka c.

Naiv-Bayes szabály Ha A feltevése mellett B_1, \dots, B_n kölcsönösen függetlenek, akkor $P(B_1, \dots, B_n \mid A) = \prod_{i=1}^n P(B_i \mid A)$

Bayes hálók

A feltételes függetlenség hasznos, mert tömöríthetjük a teljes együttes eloszlást.

Gépi tanulás: felügyelt tanulás problémája, döntési fák, naiv Bayes módszer, modellillesztés, mesterséges neuronhálók, k-legközelebbi szomszéd módszere

Felügyelt tanulás problémája

Tapasztalati tények felhasználása arra, hogy egy racionális ágens teljesítményét növeljük.

Felügyelt tanulás:

- van az adatok mögött valami $f: X \rightarrow Y$ függvény, ezt nem ismerjük
- adottak tanuló példák, amik rendezett párok $(x, f(x))$
- egy $h: X \rightarrow Y$, függvényt keresünk, ami illeszkedik a példákra, és közelíti f -et
- egy példában az első elem pl egy email, a második pedig egy valamilyen címke, pl spam
- h konzisztens az adatokra, ha $h(x) = f(x)$ minden x tanuló példára
- a h függvényt mindig valami H hipotézistérben keressük, vagyis valamilyen "alakban"
- a tanulás realizálható, ha van olyan h eleme H , amire h konzisztens
- a gyakorlatban elég, ha h közel van a példákhoz, mert a példák zajt is tartalmazhatnak, amit kifejezetten káros lenne, ha megtanulna az ágens (túltanulás)
- egy olyan h -t keresünk, ami a tanuló példákon kívül is jól általánosít
- nem szabad a tanuló példákat bemagolni
- occam borotvája: mindig a legtömörebb leírást kell venni
- a priori ismeretek fontosak, a nulláról való tanulás kb lehetetlen
- számítási szempontból egyszerű reprezentáció is fontos

Döntési fák

Induktív (felügyelt) tanulás konkrét példája.

Feltesszük, hogy X-ben diszkrét változók egy vektora van, Y-ban pedig szintén valami diszkrét változó egy értéke, pl igen-nem

Tulajdonképpen osztályozás, X elemeit kell Y valamelyik osztályába sorolni.

Előnye, hogy döntései megmagyarázhatók, mert emberileg értelmezhető lépésekben jutottunk el odáig.

Kifejezőereje megegyezik az ítéletkalkulussal.

Döntési fa építése

- adottak pozitív és negatív példák felcímkezve, tipikusan több száz
- vegyük a gyökérbe azt a változót, ami a legjobban szeparálja a pozitív és negatív példákat
- ezt folytassuk rekurzív módon
- ha csak pozitív vagy negatív példa van, akkor levélhez értünk, felcímkezzük ezzel a levelet
- ha üreshalmaz, akkor a szülő szerint többségi szavazattal címkézzük
- ha nincs több változó, de vannak negatív és pozitív példák is, akkor szintén többségi szavazattal címkézhetjük a levelet

A legjobban szeparáló attribútumot az információtartalma, azaz entrópiája segítségével választhatjuk ki.

Naiv Bayes módszer

Statisztikai következtetési módszer, amely adatbázisban található példák alapján ismeretlen példákat osztályoz.

Például emaileket akarunk spam vagy nem spamként osztályozni. Az emailben lévő szavakra meghatározzuk, hogy milyen valószínűséggel fordul elő egy normális üzenetben, vagy egy spam-ban. Ezután meg kell határozni, hogy milyen valószínűséggel kapunk normális üzenetet, és milyennel spam-et.

Ezután, ha pl kíváncsiak vagyunk, hogy egy szókombinációt tartalmazó email spam vagy nem spam, a szókombinációban előforduló szavak valószínűségét össze kell szorozni, majd megszorozni azzal, hogy milyen valószínűséggel kaptunk normális emailt, és milyennel spam-et. Amelyik valószínűségre nagyobb értéket kapunk, abba az osztályba soroljuk a szókombinációt tartalmazó üzenetet.

Modellillesztés

Lineáris regresszió?

Mesterséges neuronhálók

A mesterséges neuron a következőképpen épül fel

- bemeneti értékek, valamilyen súlyokkal megszorozva
- w_0 bias weight, eltolássúly
- először minden bemeneti értéket megszorozza a hozzá tartozó súllyal, ezeket összeadja, majd kivonja belőle az eltolássúlyt
- majd a kapott értéken alkalmazzuk az aktivációs függvényt

Az aktivációs függvény célja, hogy 1-hez közeli értéket adjon, ha jó input érkezik, és 0-hoz közelít, ha rossz.

Példa aktivációs függvények:

- küszöbfüggvény: 0, ha az input \leq mint 0, 1, ha nagyobb (perceptron)
- szigmoid függvény: $g(x) = 1/(1 + e^{-x})$
- Rectifier aktiváció: $g(x) = \max(0, x)$ (ReLU)

Neuronokból hálózatokat szokás építeni. Egy hálózatnak lehet több rétege is. Van egy input, egy output és lehet több rejtett rétege is. Egy rétegen belül a neuronok között nincs kapcsolat, csak a rétegek között (előrecsatolt hálózatok).

k-legközelebbi szomszéd módszere

13. LP alapfeladat, példa, simplex algoritmus, az LP geometriája, generálóelem választási szabályok, kétfázisú simplex módszer, speciális esetek (ciklizáció-degeneráció, nem korlátos feladat, nincs lehetséges megoldás)

LP alapfeladat

@kép

LP alapfeladat: Keressük adott lineáris célfüggvény szélsőértékét, értelmezési tartományának adott lineáris korlátokkal meghatározott részében. Lehetséges megoldás: olyan p vektor, hogy p -t behelyettesítve x -be kielégíti a feladat feltételrendszerét. Lehetséges megoldási tartomány: az összes lehetséges megoldás halmaza. Optimális megoldás: egy olyan lehetséges megoldás, ahol a célfüggvény felveszi a maximumát/minimumát.

Példa: @kép

Szimplex algoritmus

Ahhoz, hogy lecseréljük az egyenlőtlenségeket egyenlőségekre az LP alapfeladatban, adjunk hozzá minden egyenlőtlenség bal oldalához egy mesterséges változót.

Ezután fejezzük ki a mesterséges változókat az egyenlet átrendezésével.

A kapott egyenletrendszert hívjuk *szótár alaknak*.

Természetes változók: az eredeti változók Mesterséges változók: az újonnan felvett nemnegatív változók

Bázisváltozók: a szótár alakban bal oldalt álló változók Nem-bázis változók: a szótár alakban jobb oldalt álló

változók Szótár bázismegoldása: olyan x vektor, amelyben a szótár nem-bázis változóinak értéke nulla, így a

bázisváltozók értéke a jobboldali konstans lesz Lehetséges (fizibilis) bázismegoldás: olyan bázismegoldás, ami egyben lehetséges megoldás is

A simplex algoritmus:

- iteratív optimum keresés
- ismételt áttérés más szótárakra, a következő feltételek betartása mellett:

- minden iteráció szótára ekvivalens az őt megelőzőével
- minden iteráció bázismegoldásán a célfüggvény értéke nagyobb vagy egyenlő, mint az előző iterációban
- minden iteráció bázismegoldása lehetséges megoldás

Mi alapján térjünk át másik szótárra? Hogyan térjünk át, hogy a feltételek teljesüljenek? Honnan tudjuk, ha az aktuális bázismegoldás optimális? Létezik-e minden LP feladatnak optimuma?

Pivot lépés: új szótár megadása egy bázis és nembázis változó szerepének felcserélésével
 Belépőváltozó: az a nembázis változó, ami a következő szótárra áttéréskor bázisváltozónak válik
 Kilépő változó: az a bázisváltozó, ami a köv. szótárra áttéréskor nembázissá válik
 Szótárak ekvivalenciája: két szótár ekvivalens, ha az általuk leírt egyenletrendszer összes lehetséges megoldása és a hozzájuk tartozó célfüggvényértékek rendre megegyeznek

Pivot lépés előtti és utáni szótárak ekvivalensek.

SZIMPLEX:

- ha adott szótárban minden célfüggvény együttható negatív, akkor az aktuális bázismegoldás optimális
- ha nem, válasszuk a nembázis változók közül belépőváltozónak valamely k -at, amelyre a k . célfüggvény együttható pozitív
- ha ennek a változónak minden egyenletben az együtthatója negatív, a feladat nem korlátos, megállunk
- ha nem, akkor válasszuk az l . pozitív együtthatót, amelyre a konstans/együttható abszolút értéke minimális
- hajtsunk végre egy pivot lépést úgy, hogy x_k legyen a belépőváltozó, és az l . feltétel bázisváltozója legyen a kilépő változó

Generálóelem választási szabályok

Klasszikus szimplex pivot szabály:

- a lehetséges belépőváltozók közül válasszuk a legnagyobb c_k értékűt, több ilyen esetén a legkisebb indexűt
- a lehetséges kilépőváltozók közül válasszuk a legkisebb l indexű egyenlet változóját

Bland szabály

- a lehetséges belépőváltozók közül válasszuk a legkisebb indexűt
- a lehetséges változók közül válasszuk a legkisebb indexűt

Legnagyobb növekmény

Lexikografikus szabály

- kiegészítjük epszilonokkal mesterségesen a szótárt
- a lehetséges belépőváltozók közül a legnagyobb c_k értékűt válasszuk, több ilyenél a legkisebb indexűt
- a lehetséges kilépőváltozók közül azt, amelynek l indexű egyenletére az együtthatókból álló vektor lexikografikusan a legkisebb

Véletlen pivot

- 1 valószínűséggel terminál

Az lp geometriája

Ábrázolhatjuk pl a lehetséges megoldások halmazát koordináta rendszerben, két változó esetén.

Minden feltétel egy egyenest határoz meg, ezeket berajzoljuk. Ezzel valamilyen sokszöget kapunk meg, ennek a sokszögnek a csúcsainak a koordinátái lesznek a lehetséges megoldások.

Kétfázisú szimplex módszer

Ha minden konstans nemnegatív az LP feladatban, akkor mehet a szimplex

De mi van, ha vannak negatív konstansok is?

Vegyünk egy segédfeladatot

- bevezetünk egy új, x_0 segédváltozót
- legyen w az új célfüggvény, $w = -x_0$
- térjünk át szótár alakra
- vegyük a legnegatívabb jobboldalú egyenletet, és ebből fejezzük ki x_0 -t
- a többiből a mesterséges változókat
- ezután már egy lehetséges indulószótárat kapunk

A standard feladatnak csak akkor létezik lehetséges megoldása, ha $w=0$ a hozzá felírt segédfeladat optimuma.

Ha a segédfeladatot megoldjuk a szimplexszel, és annak optimuma 0, akkor a megoldás utolsó szótárából könnyen felírhatunk egy olyan szótárat, amely az eredeti feladat szótára, és bázismegoldása lehetséges megoldás is egyben.

A szótár felírásának lépései:

- az $x_0 = 0$ feltételt elhagyjuk
- ha x_0 bázisváltozó, akkor az egyenletének jobb oldalán lévő nem 0 együtthatójú változók egyikével végrehajtunk egy pivot lépést
- elhagyjuk x_0 megmaradt erőforrásait
- a célfüggvény egyenletét lecseréljük az eredeti célfüggvényre, amit átírtunk az aktuális bázisváltozóknak megfelelően

A következő fázisban pedig az átírt szótáron futtatjuk a szimplex algoritmust

Speciális esetek

Ciklizáció

Degenerált iterációs lépés: olyan szimplex iteráció, amelyben nem változik a bázismegoldás

Degenerált bázismegoldás: olyan bázismegoldás, amelyben egy vagy több bázisváltozó értéke is 0

Ciklizáció: ha a szimplex algoritmus valamely iterációja után egy korábbi szótárat visszkapunk, akkor az a ciklizáció

Ha a szimplex algoritmus nem áll meg, akkor ciklizál! A ciklizáció elkerülhető megfelelő pivot szabály alkalmazásával (lexikografikus, Bland szabály) A ciklizáció oka a degeneráció, azaz a bázisváltások 0-vá válása a bázismegoldásban

Nem korlátos

Ha az LP feladat maximalizálandó/minimalizálandó, és a célfüggvénye tetszőlegesen nagy/kicsi értéket felvehet, akkor nem korlátos a feladat.

Nincs lehetséges megoldás

Ha a standard alakú LP feladatot kétfázisú szimplex módszerrel oldjuk meg, az első fázis eldönti, hogy van-e lehetséges megoldás.

Ha a felírt segédfeladatban az optimum értéke kisebb, mint nulla, akkor nincs lehetséges megoldás, ha 0, akkor van.

14. Primál-duál feladatpár, dualitási komplementaritási tételek, egész értékű feladatok és jellemzőik, a branch and bound módszer, a hátizsák feladat

Primál-duál feladatpár

A primál feladat

- maximalizálunk
- c^T a célfüggvény együtthatóinak a vektora
- A az együtthatók mátrixa
- b a konstansok vektora

A duál feladat

- minimalizálunk
- b^T a célfüggvény együtthatóinak a vektora
- A^T az együtthatók mátrixa
- c a konstansok vektora
- \leq -ket \geq -re cseréljük

A duál feladat duálisa az eredeti primál feladat

TFH az LP feladatunk egy korlátozott erőforrások mellett maximális nyereséget célzó gyártási folyamat modellje. A duál feladat megoldásában az optimális megoldás a primál feladat i . erőforrásához tartozó marginális ár/árnyékár, azaz az erőforrás értéke az LP megoldójának szemszögéből. Ha túl sok van egy erőforrásból, az nem érhet túl sokat. Továbbá y_i^* -nál többet nem érdemes fizetni az i . erőforrásért, kevesebbet igen.

Dualitási komplementaritási tételek

Gyenge dualitás

Ha x egy lehetséges megoldása a primál feladatnak, és y egy lehetséges megoldása a duál feladatnak, akkor a duális feladat bármely lehetséges megoldása felső korlátja a primál bármely lehetséges megoldásának (azaz az optimális megoldásnak is) (azaz a duális feladat bármely lehetséges megoldása nagyobb vagy egyenlő a primál bármely lehetséges megoldásánál)

A korlátosság és a megoldhatóság nem függetlenek egymástól

Ha a primál nem korlátos, akkor a duálnak nincs lehetséges megoldása és fordítva. Lehet, hogy egyiknek sincs lehetséges megoldása. Ha mindkettőnek van, akkor mindkettő korlátos. A primál és a duál feladat egyidejű optimalitása ellenőrizhető.

Erős dualitás

Ha x^* egy optimális megoldása a primálnak, és y^* egy optimális megoldása a duál feladatnak, akkor $c^T x^* = b^T y^*$.

Ha valamelyik i . feltétel egyenlet nem éles, azaz nem pontosan egyenlő a két oldal, akkor a kapcsolódó duál változó biztosan 0. Ha egy primál változó pozitív, akkor a kapcsolódó duális feltétel biztosan éles.

Egész értékű feladatok és jellemzőik

Tiszta egészértékű feladat (Integer Programming)

- Minden változónak egésznek kell lennie a megoldásban.

Vegyes egészértékű programozási feladat (Mixed Integer Programming)

- Csak néhány változóra követeljük meg, hogy egész legyen

0-1 IP

- minden változó értéke csak vagy 0 vagy 1 lehet

LP lazítás

Egészértékű programozási feladat LP lazítása az az LP, amelyet úgy kapunk, hogy a változókra tett minden egészértékűségi vagy 0-1 megkötést eltörölünk.

- Bármelyik IP lehetséges megoldáshalmaz része az LP lazítás lehetséges megoldástartományának
- Maximalizálásnál az LP lazítás optimum értéke nagyobb egyenlő, mint az IP optimum értéke
- Ha az LP lazítás lehetséges megoldáshalmazának minden csúcspontja egész, akkor van egész optimális megoldása, ami az IP megoldása is egyben
- Az LP lazítás optimális megoldása bármilyen messze lehet az IP megoldásától.

Branch and bound módszer

1. lépés

Megoldjuk az LP lazítást, ha a megoldás egészértékű, akkor done

2. lépés

Ha van lezáratlan részfeladatunk, akkor azt egy x_i nem egész változó szerint két részfeladatra bontjuk. Ha x_i értéke x_i^* , akkor $x_i \leq \text{floor}(x_i^*)$ és $x_i \geq \text{ceil}(x_i^*)$ feltételeket vesszük hozzá egy egy részfeladatunkhoz

- a részproblémákat egy fába rendezzük
- a gyökér az első részfeladat, az LP lazítás
- a leszármazottai az ágaztatott részproblémák
- a hozzávett feltételeket az éleken adjuk meg
- a csúcsokban jegyezzük az LP-k optimális megoldásait

Lehet, hogy olyan részproblémát kapunk, aminek nincs lehetséges megoldása, ekkor ezt a levelet elhagyjuk. Találhatunk megoldásjelölteket is, ezek alsó korlátok az eredeti IP optimális értékére. Ha találunk korábbi megoldásjelöltnél jobb megoldást, akkor a rosszabbat elvetjük.

Egy csúcs felderített/lezárt, ha

- nincs lehetséges megoldása
- megoldása egészértékű
- felderítettünk már olyan egész megoldást, ami jobb a részfeladat megoldásánál

Egy részfeladatot kizárunk, ha

- nincs lehetséges megoldása
- felderítettünk már olyan egész megoldást, ami jobb a részfeladat megoldásánál

A hátizsák feladat

Egy olyan IP-t, amiben csak egy feltétel van, hátizsák feladatnak nevezünk. Van egy hátizsákunk egy fix kapacitással, és tárgyaink, értékekkel és súlyokkal megadva.

Maximalizálni akarjuk a táskába rakott tárgyak értékét, úgy hogy a benne lévő tárgyak nem haladhatják meg a hátizsák kapacitását persze. 0-1 IP feladat, egy tárgyat viszünk vagy nem

Az LP lazítás könnyen számítható, a relatív hasznosság szerint tesszük a tárgyakat a táskába, vagyis az érték/súly hányadosuk szerint. Branch and bound módszerrel ez is megoldható

Legrosszabb esetben 2^n részfeladatot kell megoldani, NP nehéz a feladat. Egészértékűnél még rosszabb, $2^M n$, ahol M a lehetséges egészek száma egy változóra

15. Processzusok, szálak/fonalak, processzus létrehozása/befejezése, processzusok állapotai, processzus leírása. Ütemezési stratégiák és algoritmusok köteget, interaktív és valós idejű rendszereknél, ütemezési algoritmusok céljai.

Kontextus-csere

Operációs rendszer

A számítógépeknek azt az alapprogramját, mely közvetlenül kezeli a hardvert, és egy egységes környezetet biztosít a számítógépen futtatandó alkalmazásoknak, op.rendszernek nevezzük. Egy moder számítógép a következőkből áll:

- egy vagy több processzor
- memória
- lemezek
- I/O eszközök
- ...

Ezen komponensek kezelése egy szoftver réteget igényel – Ez az op. rendszer Feladatai:

- felhasználó kényelmének, védelmének biztosítása
- egy egységes környezetet biztosít a számítógépen futtatandó alkalmazásoknak
- a rendszer hatékonyságának, teljesítményének maximalizálása = erőforrások kezelése
- a programok végrehajtását vezérli
- biztosítja a felhasználó és a számítógépes rendszer közötti kommunikációt

Felépítése:

- Rendszerhéj (shell)
 - Feladata a parancsértelmezés.
 - Lehet a shell parancssoros (CLI - Command Line Interface - mint, pl. DOS), vagy grafikus felületű
 - Kapcsolattartás a felhasználóval
- Alacsony szintű segédprogramok
 - felhasználói "élményt" fokozó kiegészítő programok (pl szövegszerkesztők, fordítóprogramok), amelyek nem képzik a rendszer elválaszthatatlan részét
- Kernel
 - az operációs rendszer alapja (magja), amely felelős a hardver erőforrásainak kezeléséért)
 - közvetlenül a hardverrel áll kapcsolatban.
 - Ki- és bemeneti eszközök kezelése (billentyűzet, monitor stb.)
 - Programok, folyamatok futásának kezelése
 - Indítás, futási feltételek biztosítása, leállítás
 - Memória-hozzáférés biztosítása
 - Processzor idejének elosztása

Az operációs rendszerek csoportosítása

- Felhasználók száma szerint:
 - egy felhasználós pl.: DOS, Win 9x
 - több felhasználós pl. Linux, Win NT
- Hardver mérete szerint:
 - kisgépes (UNIX)
 - nagygépes (Main Frame, Cray - szuper számítógép)
 - mikrogépes (DOS, WIN 9X, UNIX)
- Processzorkezelés szerint:
 - egy feladatos (DOS)
 - több feladatos (WIN 9X, WIN NT, UNIX)
- Cél szerint:

- általános (DOS, WIN 9X, WIN NT, UNIX)
- speciális (folyamatvezérlő operációs rendszerek)
- Operációs rendszer felépítése szerint:
 - monolitikus
 - A monolitikus operációs rendszer (mint például a UNIX) magja egyetlen programból áll. Ebben a programban az eljárások szabadon hívhatják egymást, a köztük lévő kommunikáció eljárásparamétereken és globális változókon keresztül zajlik.
 -
 - réteges szerkezetű (WIN NT, UNIX)
 - A rétegzett szerkezetű operációs rendszer magja több modulból áll, és a modulok között egy export-import hierarchia figyelhető meg: minden modul kizárólag a hierarchiában alatta lévő modul interfészét használja.
 - kliens/szerver felépítésű - Hálózati operációs rendszer
 - a szerveren fut, és lehetővé teszi a szervernek az adatok, felhasználók, csoportok, alkalmazások, a hálózati biztonság és egyéb hálózati funkciók kezelését.
 - A kliens/szerver hálózati operációs rendszerek lehetővé teszik funkciók és alkalmazások központosítását egy vagy több dedikált szerveren. A szerver a rendszer központja, engedélyezi az erőforrásokhoz való hozzáférést és biztonságos kapcsolatot nyújt
 - vegyes
 - virtuális gépek
 - A virtuális gépeken alapuló operációs rendszerben központi részen helyezkednek el a virtuális gépeket menedzselő (hypervisor) rendszerrutinok. Ez a program lehetővé teszi a hardver erőforrásainak (CPU, diszk, perifériák, memória, ...) több operációs rendszer közötti hatékony elosztását. A hypervisor leggyakrabban a számítógép hardverét "többszörözi meg" úgy, hogy a rajta futó operációs rendszerek azt higgyék, hogy övéké az egész gép (pedig "csak" egy virtuális gépen futnak)
- A felhasználói felület szerint:
 - szöveges (DOS, UNIX)
 - grafikus (WINDOWS)

Op. rendszer generációk

- Első generációs (1945-1955)
 - nincs os. leginkább csak hardver elemekből állt (különbféle kapcsolók, címkiválasztó kulcsok, indító-, megállító-, lépésenkénti végrehajtást kiváltó gombok stb.)
 - programozó=gépkezelő=programok végrehajtásának vezérlője
 - bináris kódolás
- Második generáció (1955-1965)
 - os van
 - egyidejűleg 1 processzus
 - fortran programozás
- Harmadik generációs (1965-1980)
 - os van, szoftverrel megvalósított operációs rendszer
 - integrált áramkörök, multiprogramozás

- egyidejűleg több proc
- CPU időszeletelés (time slicing): egy processzus egy meghatározott max időintervallumon keresztül használhatja a CPU-t folyamatosan (ez a tmax idő). Ha ez letelik az op.rendszer processzus ütemező átadja a CPU-t egy másik processzusnak
- Átmeneti tárolás (spooling): az I/O adatok először gyors háttértárolókra kerülnek, majd a processzus innen kapja/ide írja az adatait
- Negyedik generációs (1980-tól)
 - személyi számítógépek
 - parancssoros, grafikus felület
 - pc, workstation: egyetlen felhasználó, egy időben több feladat (Windows, MacOS)
 - hálózati os: hálózaton keresztül több felhasználó, kapcsolódik, minden felhasználó több feladatot futtathat (Unix, Linux)
 - osztott os: egy feladatot egy időben több számítógépes rendszer végez

SZERINTEM INNENTŐL KELL

Processzusok, szálak/fonalak, processzus létrehozása/befejezése, processzus állapotai, processzus leírása

Processzus

- A végrehajtás alatt lévő program.
- Szekvenciálisan végrehajtódó program
- Egyidejűleg több processzus létezik: A processzor idejét meg kell osztani az egyidejűleg létező processzusok között: időosztás (time sharing)
- Futó processzusok is létrehozhatnak processzusokat: Kooperatív folyamatok, egymással együttműködő, de amúgy független processzusok
- Az erőforrásokat az OS-től kapják (centralizált erőforrás kezelés)
- jogosultságokkal rendelkeznek
- Előtérben és áttérben futó folyamatok Processzus állapotok:
- Futáskész: készen áll a futásra, csak ideiglenesen le lett állítva, hogy egy másik processzus futhasson
- Futó: a proc bitkolja a CPU-t
- Blokkolt: bizonyos külső esemény bekövetkezéséig nem képes futni
- Inicialis
- Terminális
- Felfüggesztett

Processzustáblázat és PCB

A proc nyilvántartására, tulajdonságainak leírására szolgáló memóriaterület. Processusonként egy egy bejegyzés - Processzus vezérlő blokk (PCB) PCB tartalma:

- azonosító: processzus id
- processzus állapota
- CPU állapota: a kontextus cseréhez
- jogosultságok, prioritás
- birtokolt erőforrások

Processzus létrehozása

- Futó processzusok is létrehozhatnak processzusokat: Kooperatív folyamatok, egymással együttműködő, de amúgy független processzusok
- Egyszerű esetekben megoldható, hogy minden processzus elérhető az OS elindulása után
- Általános célú rendszerekben szükség van a processzusok létrehozására és megszüntetésére
- Processzusokat létrehozó események:
 - Rendszer inicializálása
 - Felhasználó által kezdeményezett
 - Köteget feladat kezdeményezése
- Az OS indulásakor sok processzus keletkezik:
 - Felhasználókkal tartják a kapcsolatot: Előtérben futnak
 - Nincsenek felhasználóhoz rendelve:
 - Saját feladatuk van
 - Háttérben futnak
- Lépései:
 1. Memóriaterület foglalása a PCB számára
 2. PCB kitöltése iniciális adatokkal
 3. Programszöveg, adatok, verem számára memóriefoglalás, betöltés
 4. A PCB procok láncra fűzése, futáskész állapot. Ettől kezdve a proc osztozik a CPU-n.

Processzus befejezése

- Szabályos kilépés (exit(0)): önkéntes, végzett a feladatával
- Kilépés hiba miatt
- Kilépés végzetes hiba miatt: önkéntelen, illegális utasítás, nullával osztás
- Egy másik proc megsemmisíti: önkéntelen, másik proc kill() utasítására
- Lépései: 1. Gyermek procok megszüntetése (rekurzívan) 2. PCB procok láncról való levétele, terminális állapot. Ettől kezdve a proc nem osztozik a CPU-n 3. Proc bitrozában lévő erőforrások felszabadítása (pl. fájlok lezárása) 4. A memóriatérképnek (konstansok, változók, dinamikus változók) megfelelő memóriaterület felszabadítása 5. PCB memóriaterületének felszabadítása

Szálak/fonalak (thread)

- Önálló végrehajtási egységként működő program, objektum, szekvenciálisan végrehajtható utasítás-sorozat
- A proc hozza létre (akár többet is egyszerre)
- Osztozik a létrehozó proc erőforrásaiban
- Egy folyamaton belül több tevékenység végezhető párhuzamosan
- Szálak megvalósítása:
 - A felhasználó kezeli a szálakat egy függvénykönyvtár segítségével. Ekkor a kernel (az operációs rendszer alapja (magja), amely felelős a hardver erőforrásainak kezeléséért) nem tud semmit a szálakról
 - A kernel kezeli a szálakat. Szálak létrehozása és megszüntetése kernelhívásokkal történik

Ütemezési stratégiák és algoritmusok köteget, interaktív és valós idejű rendszereknél, ütemezési algoritmusok céljai

Ütemező

- Egy CPU áll rendelkezésre. Processzusok versengenek a CPU-ért
- Az OS dönti el, hogy melyik kapja meg a CPU-t
- Az ütemező (scheduler) hozza meg a döntést □ Ütemezési algoritmus
- Feladata: Egy adott időpontban futáskész procok közül egy kiválasztása, amely a következőkben a CPU-t bitrolni fogja
- Mikor kell ütemezni?: amikor egy processus befejeződik vagy blokkolódik
- Céljai:
 - a CPU legyen jól kihasznált
 - az átfutási idő (proc létrejöttétől megszűnéséig eltelt idő) legyen rövid
 - egységnyi idő alatt minél több proc teljesüljön

Ütemezés kötegelt rendszerekben

A manapság használatos op.rendszerek nem tartoznak a kötegelt rendszerek (: Előre meghatározott sorrend szerint végrehajtandó feladatok együttese.) világába, mégis érdemes röviden megemlíteni ezek ütemezési típusait.

- Sorrendi ütemezés:
 - Futásra kész folyamatok egy várakozó sorban helyezkednek el.
 - A sorban levő első folyamatot hajtja végre a központi egység. Ha befejeződik a folyamat végrehajtása, az ütemező a sorban következő feladatot veszi elő.
 - Új feladatok a sor végére kerülnek
 - Ha az aktuálisan futó folyamat blokkolódik, akkor a sorban következő folyamat jön, míg a blokkolt folyamat, ha újra futásra kész lesz, akkor a sor végére kerül, és majd idővel újra rá kerül a vezérlés.
- Legrövidebb feladat először:
 - az a folyamat kerül először ütemezésre, melyeknek a legkisebb a futási ideje.
 - az alkalmazhatóság szempontjából nem ideális, ha nem tudjuk előre a folyamatok végrehajtási idejét.
- Legrövidebb maradék futásidejű:
 - Ismerni kell a folyamatok futási idejét előre.
 - Amikor új folyamat érkezik, vagy a blokkolás miatt egy következő folyamathoz kerül a vezérlés, akkor nem a teljes folyamat végrehajtási idejét, hanem csak a hátralévő időt vizsgálja az ütemező, és amelyik folyamatnak legkisebb a maradék futási ideje, az kerül ütemezésre
- Háromszintű futásidejű:
 - A feladatok a központi memóriában vannak, közülük egyet hajt végre a központi egység. Előfordulhat, hogy a többi feladat közül ki kell rakni egyet a háttértárba, mivel a működés során elfogyhat a memória.
 - Az a döntést, hogy a futásra jelentkező folyamatok milyen sorrendben kerüljenek be a memóriába, a bebocsátó ütemező hozza meg.

Ütemezés interaktív rendszereknél

- Round Robin
 - Az ütemező beállít egy időintervallumot egy időzítő segítségével és amikor az időzítő lejár megszakítást ad.
 - Megadott időközönként óramegszakítás következik be és ekkor az ütemező a következő folyamatnak adja a processzort.
 - A folyamatokat egy sorban tárolja a rendszer, és amikor lejárt az időszelvény, akkor az a folyamat, amelyiktől az ütemező éppen elveszi a vezérlést, a sor végére kerül
- Prioritásos ütemezés
 - Felmerül az igény, hogy nem feltétlenül egyformán fontos minden egyes folyamat.
 - A folyamatokhoz egy fontossági mérőszámot, prioritást (prioritási osztályt) rendel hozzá
 - A legmagasabb prioritású futáskész processzus kapja meg a CPU-t

Ütemezés valós idejű rendszereknél

Alapvető szerepe van az időnek Ha a feladatainknak nemcsak azt szabjuk meg, hogy hajtódjanak végre valamilyen korrekt ütemezés szerint, hanem az is egy kritérium, hogy egy adott kérést valamilyen időn belül ki kell szolgálni, akkor valós idejű op.rendszerrel beszélünk. A megfelelő határidők betartása úgy valósítható meg, hogy egy programot több folyamatra bontunk, és ezeknek a rövid folyamatoknak az ütemező biztosítja a számukra előírt határidő betartását

- Szigorú valós idejű rendszer
 - a határidő betartása kötelező
- Toleráns valós idejű (soft real-time) rendszer
 - a határidők kis mulasztása még elfogadható, tolerálható.

Kontextus csere

Egy CPU van és több egyidejűleg létező processzus. A CPU váltakozva hajtja végre a processzusokat. A kontextus csere, amikor a CPU átvált P1 processzusról a P2 processzusra. Ilyenkor P1 állapotát el kell menteni a CPU regisztereiből, az erre fenttartott memóriaterületre, majd P2 mentett állapotát vissza kell állítani a CPU regisztereiben.

SZERINTEM INNENTŐL MÁR NEM KELL

Operációs rendszerek feladatai, fajtái, felépítései és felhasználási területei. Párhuzamossággal kapcsolatos fogalmak, problémák és megoldásaik. Folyamatok, szálak fogalma, megvalósításaik és ütemezési módszereik. Memóriakezeléssel, állományrendszerekkel és szolgáltatásaikkal kapcsolatos fogalmak és megvalósítási módszereik

Memóriakezeléssel, állományrendszerekkel és szolgáltatásaikkal kapcsolatos fogalmak és megvalósítási módszereik Memóriakezelés A memória az egyik legfontosabb (és gyakran a legszűkösebb) erőforrás, amivel egy operációs rendszernek gazdálkodnia kell; főleg a többfelhasználós rendszerekben, ahol gyakran olyan sok és nagy folyamat fut, hogy együtt nem férnek be egyszerre a memóriába. A többfeladatos feldolgozás megjelenésével azonban szükségessé vált a memóriának a futó folyamatok közötti valamilyen „igazságos” elosztására.

- Multiprogramozás megvalósítása rögzített memória szeletekkel.

- Osszuk fel a memóriát n szeletre. (Fix szeletek) pl. rendszerindításnál ez megtehető
- a főmemória kihasználása nem jó: minden program, méretétől függetlenül egy egész partíciót elfoglal.
- Megoldás: nem egyenlő méretű partíciók
- Multiprogramozás megvalósítása memória csere használatával.
 - Teljes folyamat mozgatása memória-lemez között
 - Nincs rögzített memória partíció, mindegyik dinamikusan változik, ahogy az op.rendszer odavissza rakosgatja a folyamatokat.
 - Dinamikus, jobb memória kihasználtságú lesz a rendszer, de a sok csere lyukakat hoz létre!
 - Memória tömörítést kell végezni
- Multiprogramozás megvalósítása virtuális memória használatával.
 - Egy program használhat több memóriát mint a rendelkezésre álló fizikai méret.
 - Az operációs rendszer csak a „szükséges részt” tartja a fizikai memóriában
 - MMU - virtuális címek fizikai címekre való leképezése
- Multiprogramozás szegmentálással A megoldást a virtuálmemória kezelés jelentette. Az operációs rendszer úgy szabadít fel memóriát az éppen futó program számára, hogy a memóriában tárolt, de éppen nem használt blokkokat (lapokat) kiírja a külső tárolóra, amikor pedig ismét szükség van rájuk, visszaolvassa őket. Ilyenkor az operációs rendszer ad a központi memóriából egy akkora részt, amelyben a folyamat a legfontosabb részeit el tudja tárolni. A többit kirakja a háttértárra (az ún. lapozófájlba, Unix-ban ezt swap-nek hívják (a procok akkor is futhatnak ha csak részeik vannak a memóriában)). Ez a megoldás azért működik, mert a programok legtöbbször egy eljáráson belül ciklusban dolgoznak, nem csinálnak gyakran nagy ugrásokat a program egyik végéről a másikra. A központi egység fel van szerelve egy úgynevezett memóriakezelő egységgel (MMU), amely figyeli, hogy olyan kódrészre kerül-e a vezérlés, amely nincs benn a központi memóriában (mert például a háttértárra van kirakva). Memóriahasználat szerint a programokat 2 részre oszthatjuk:
 - rezidens (állandóan a memóriában van, gyorsabb, tűzfal, vírusirtó)
 - tranzien (csak meghíváskor töltődik be, helytakarékosabb)

Állományrendszerek (file system) A számítógépek az adatokat különböző fizikai háttértárakon tárolhatják, a számítógép kényelmes használhatósága érdekében az operációs rendszerek egységes logikai szemléletet vezetnek be az adattárolásra és adattárakra Az operációs rendszer támogatást nyújthat a fájl tartalmának kezelésében, a fájl szerkezetének (adatszerkezet) létrehozásában. Állományrendszer: fájlok tárolásának és rendszerezésének a módszere, ideértve a tárolt adatokhoz való hozzáférése és az adatok egyszerű megtalálása is

Párhuzamossággal kapcsolatos fogalmak, problémák és megoldásaik A CPU minden időpillanatban egy programot futtat, az egyik program végrehajtásáról a másik program végrehajtására vált. Ez a párhuzamosság illúzióját kelti a felhasználóban, de valójában nem erről van szó. Nem összekeverendő a többprocesszoros rendszerek valódi hardverpárhuzamosságával.

Valódi párhuzamosság:

- Többprocesszoros rendszerek
- Akár processzorok százai egy számítógépben
- Közös sín, közös órajel, akár közös memória és perifériák, gyors kommunikáció A párhuzamosítás tipikus megoldása az időosztás, amikor minden folyamat kap egy-egy ún. időszeletet, melynek leteltét követően egy másik folyamat kapja meg a vezérlést. Párhuzamosság problémái:
 - a rendszerben futó folyamatok általában nem függetlenek

- Közös erőforrásokat használnak
- függő folyamatok együttes viselkedése új típusú hibákat eredményezhet
- versenyhelyzetek kialakulása: párhuzamos végrehajtás által okozott nemdeterminisztikus hibás eredmény Multiprogramozás: Ha az operációs rendszer egy időben több programot futtat, multiprogramozásról beszélünk, melynek célja az erőforrások jobb kihasználása és a kényelem

16. Processzusok kommunikációja, versenyhelyzetek, kölcsönös kizárás. Konkurens és kooperatív processzusok. Kritikus szekciók és megvalósítási módszereik: kölcsönös kizárás tevékeny várakozással (megszakítások tiltása, változók zárolása, szigorú váltogatás, Peterson megoldása, TSL utasítás). Altatás és ébresztés: termelő-fogyasztó probléma, semaforok, mutex-ek, monitorok, Üzenet, adás, vétel. Írók és olvasók problémája. Sorompók

Processzusok kommunikációja, versenyhelyzetek, kölcsönös kizárás.

Processzusok kommunikációja

- A processzusoknak szükségük van a kommunikációra
 - Adatok átadása az egyik folyamatból a másiknak (Pipelining)
 - Közös erőforrások használata (memória, nyomtató, stb.)

Versenyhelyzet

- Kooperatív processzusok közös tárolóterületen dolgoznak (olvasnak és írnak).
- Processzusok közös adatot olvasnak és a végeredmény attól függ, hogy ki és pontosan mikor fut
- Megoldás: Egyszerre csak egy folyamat lehet kritikus szekcióban. Amíg a folyamat kritikus szekcióban van, azt nem szabad megszakítani. Ebből a megoldásból származhatnak új problémák.

Kölcsönös kizárás

- Az a módszer, ami biztosítja, hogy ha egy folyamat használ valamilyen megosztott, közös adatot, akkor más folyamatok ebben az időben ne tudják azt elérni
- pl.: egy adott időben csak egy processzus számára engedélyezett, hogy a nyomtatónak utasításokat küldjön
- Kölcsönös kizárás miatt előfordulható problémák:
 - holtpon (deadlock): processzusok egymásra befejeződésére várnak, hogy a várt erőforrás felszabaduljon
 - éhezés (starvation): egy processzusnak határozatlan ideig várnia kell egy erőforrás használatára

Kritikus szekció

- A program azon része, amelyben a programunk a közös adatokat használja
- Szabályok:
 - legfeljebb egy proc lehet kritikus szekciójában
 - kritikus szekción kívüli proc nem befolyásolhatja másik proc kritikus szekciójába lépését
 - véges időn belül bármely kritikus szekciúba lépni kívánó proc beléphet

Kritikus szekciók és megvalósítási módszereik: kölcsönös kizárás tevékeny várakozással (megszakítások tiltása, változók zárolása, szigorú váltogatás, Peterson megoldása, TSL utasítás).

Láthattuk, hogy a kritikus szekciójába való belépés nem feltétel nélküli. Hogyan biztosíthatjuk a kölcsönös kizárás teljesülését?

- Hardware-es módszer
 - Megszakítások tiltásával
 - letiltjuk a megszakítást a kritikus szekciójába lépés után, majd újra engedélyezzük, mielőtt elhagyja azt, így nem fordulhat elő óramegszakítás, azaz a CPU nem fog másik processzusra váltani
 - jól használható, de általánosan nem biztos, hogy a legszerencsésebb
 - a legegyszerűbb hiba, hogy elfelejtjük újra engedélyezni a megszakítást a kritikus szekció végén
 - TSL utasítás segítségével
 - A mai rendszerekben a processzornak van egy „TSL reg, lock” (TSL EAX, lock) formájú utasítása (TSL – Test and Set Lock).
 - Ez az utasítás beolvassa a LOCK memóriaszó tartalmát a „reg” regiszterbe, majd egy nem nulla értéket ír a „lock” memóriacímre.
 - A CPU zárolja a memóriasínt, azaz tiltva van a memória elérés a CPU-knak a művelet befejezéséig.
 - A művelet befejezésekor 0 érték kerül a LOCK memóriaterületre
- Software-es módszer

◦ Szigorú váltogatás módszere A folyamat B folyamat

- Peterson-féle megoldás
 - Van két metódus a kritikus szekciójába való belépésre (enter_region) és kilépésre (leave_region). A kritikus szekciójába lépés előtt a processzus meghívja az enter_region eljárást, kilépéskor pedig a leave_region eljárást. Az enter_region eljárás biztosítani fogja, hogy a másik processzus várakozik, ha szükséges.
 -
- Változók zárolása
 - Van egy osztott zárolási változó, aminek a kezdeti értéke 0. Kritikus szekciójába lépés előtt a processzus teszteli ezt a változót. Ha 0 az értéke, akkor 1-re állítja és belép a kritikus szekciójába. Ha az értéke 1, akkor várakozik, amíg nem lesz 0.

Altatás és ébresztés: termelő-fogyasztó probléma, szemaforok, mutex-ek, monitorok, Üzenet, adás, vétel.

Altatás-ébresztés

Ahogy láttuk az előző, tevékeny várakozást használó versenyhelyzet-elkerülő megoldásokban a legfontosabb gond az, hogy processzoridőt pazarolnak. Ahhoz, hogy a drága processzoridőt se pazaroljuk, olyan megoldást lehet javasolni, ami vagy blokkolni tud egy folyamatot (aludni küldi), vagy fel tudja ébreszteni ebből a blokkolt állapotból.

A tevékeny várakozás feloldására az egyik eszköz a sleep-wakeup rendszerhívás páros. A lényege, hogy a sleep rendszerhívás blokkolja a hívót, azaz fel lesz függesztve, amíg egy másik processzus fel nem ébreszti. A wakeup rendszerhívás a paraméterül kap egy processzus azonosítót, amely segítségével felébreszti az adott processzust, tehát nem lesz blokkolva továbbá.

Termelő-fogyasztó probléma

Véges méretű memóriaterületen (tárolón) dolgozik két processzus (osztottnak). A gyártó adatokat helyez el a tárolón, a fogyasztó kiveszi az adatokat a tárolóból és feldolgozza azokat, viszont a memória véges. Ha a tároló tele van és a gyártó elemet akar berakni, akkor elalszik, majd felébreszti a fogyasztót, ha egy elemet kivesz a tárolóból. Fordítva is: ha a tároló üres, és a fogyasztó ki akar venni egy elemet, akkor elalszik, és majd felébreszti a gyártót, ha legyártott egy eleme

Szemafor

- „A vonat megáll egy piros szemafor előtt, és addig várakozik, amíg szabad utat nem kap, mert valamilyen oknál fogva (elaludt a bakter, foglalt a pálya stb.) a továbbhaladás meg van tiltva.”
- A szemafor a számítógép-programozásban használt változó vagy absztrakt adattípus, amit az osztott erőforrásokhoz való hozzáférések szabályozásához használnak a többszálú környezetekben.
- Ha értéke pozitív, akkor nyitott állapotban van, ha nulla, akkor tilosat mutat
- Amekkora értékkel inicializáljuk a szemafort annyi „vonatot” enged át, mielőtt tilosat mutatna.
- pl.: Tekintsünk egy egész számot. Legyen, mondjuk, a kezdőértéke egy. Amikor a kritikus művelethez érek, akkor azt mondom, hogy jelzem az erőforrás-használati igényemet. A jelzés jelentse azt, hogy eggyel csökkentem a szám értékét. Ezt szokás „down” vagy sok más helyen „P” operációnak is nevezni. Ha a csökkentés eredmény nem negatív lesz, akkor szabad az út, és végzi a dolgát a program. Ha ezután érkezik egy másik folyamat, ami ugyanezt az erőforrást szeretné használni, szintén hasonló módon kezdi a dolgot, de neki már a P operáció pirosra állítja a szemafort hiszen az „egész” értéke mínusz egy lesz. Ekkor ez a második folyamat mindaddig vár, amíg a szemafor értékét egy úgynevezett „up” vagy „V” operációval – ami az eggyel való növelést jelenti – fel nem szabadítja az erőforrást ami után a P operációnál várakozó program tovább haladhat. Ezzel tulajdonképpen újra tilos jelzés lesz érvényben a kritikus erőforrásra a kezdőérték egy volt.

Mutex

- Olyan speciális szemafor, amelynek csak két értéke lehet
- Ha csak kölcsönös kizárás biztosítására kell a szemafort létrehozni, és nincs szükség annak számlálási képességére, akkor azt egy kezdőértékkel hozzuk létre. Ezt a kétállapotú (értéke 0 és 1) szemafort sok környezetben speciális névvel, az angol kölcsönös kizárás kifejezésből mutexnek nevezzük.
- Ha egy folyamatnak zárolásra van szüksége, a „mutex_lock” eljárást hívja, míg ha a zárolást meg akarja szüntetni, a „mutex_unlock” utasítást hívja.
- Aki másodszor (vagy harmadszor) hívja a „mutex_lock” eljárást, az blokkolódik, és csak a „mutex_unlock” hatására tudja folytatni a végrehajtást.

Monitor

- Eljárások, változók és adatszerkezetek együttese egy speciális modulba összegyűjtve, hogy használható legyen a kölcsönös kizárás megvalósítására
- Legfontosabb tulajdonsága, hogy egy adott időpillanatban csak egy proc lehet aktív benne
- A processzusok bármikor hívhatják a monitorban lévő eljárásokat, de nem érhetik el a belső adatszerkezeit (mint OOP-nál)
- wait(c): alvó állapotba kerül a végrehajtó proc
- signal(c): a c miatt alvó procot felébreszti

Üzenet, adás, vétel.

- Folyamatok együttműködéshez információ cserére van szükség. Két mód:
 - közös tárterületen keresztül
 - kommunikációs csatornán keresztül (egy vagy kétirányú)
- Folyamat kommunikáció fajták:
 - Közvetlen kommunikáció
 - csak egy csatorna létezik, és más folyamatok nem használhatják
 -
 - Közvetett kommunikáció
 - Közbülső adatszerkezeten (pl. postaládán (mailbox)) keresztül valósul meg.
 -
 - Aszimmetrikus
 - Adó vagy vevő megnevezi, hogy melyik folyamattal akar kommunikálni
 - A másik fél egy kaput (port) használ, ezen keresztül több folyamathoz, is kapcsolódhat.
 - Tipikus eset: a vevőhöz tartozik a kapu, az adóknak kell a vevő folyamatot és annak a kapuját megnevezni. (Pl. szerver, szolgáltató folyamat)
 -
 - Üzenetszórás
 - A közeg több folyamatot köt össze.
 -
- Műveletek:
 - send(cél, &üzenet)
 - receive(forrás, &üzenet)

Írók és olvasók problémája. Sorompók.

Írók és olvasók problémája

Több proc egymással versengve írja és olvassa ugyanazt az adatot. Megengedett az egyidejű olvasás, de ha egy proc írni akar, akkor más procok sem nem írhatnak se nem olvashatnak. (pl, adatbázisok, fájlok, hálózat)

Sorompók:

- Sorompó primitív
 - Könyvtári eljárás
- Fázisokra osztjuk az alkalmazást
- Szabály
 - Egyetlen processzus sem mehet tovább a következő fázisra, amíg az összes processzus nem áll készen

- Sorompó elhelyezése mindegyik fázis végére
 - Amikor egy processzus a sorompóhoz ér, akkor addig blokkolódik ameddig az összes processzus el nem éri a sorompót
- A sorompó az utolsó processzus beérkezése után elengedi a azokat
- Nagy mátrix-okon végzett párhuzamos műveletek

1. Adatbázis-tervezés: A relációs adatmodell fogalma. Az egyed-kapcsolat diagram és leképezése relációs modellre, kulcsok fajtái. Funkcionális függőség, a normalizálás célja, normálformák

1. Adatbázis-tervezés: A relációs adatmodell fogalma. Az egyed-kapcsolat diagram és leképezése relációs modellre, kulcsok fajtái. Funkcionális függőség, a normalizálás célja, normálformák

A relációs adatmodell fogalma

A relációs adatmodell mind az adatokat, mind a köztük lévő kapcsolatokat kétdimenziós táblákban tárolja.

Attribútum:

- névvel, értéktartománnyal megadott tulajdonság
- Z értéktartományát $\text{dom}(Z)$ jelöli
- csak elemi típusú értékekből állhat
- gyakran megadjuk az ábrázolás hosszát is

Relációséma:

- névvel ellátott attribútumhalmaz
- $R(A)$, ahol A az attribútumok halmaza
- névütközés esetén kiírhatjuk a tábla nevét is az attribútum elé

A relációséma nem tárol adatot! Csak szerkezeti leírást jelent.

Az adatok relációkkal adhatók meg. Egy $R(A)$ séma feletti reláció A értéktartományainak direktszorzatának egy részhalmaza (mindegyik attribútum értékei közül választunk egyet, és ezt egy vektorba pakoljuk). Egy ilyen reláció már megjeleníthető adattábla formájában, egy reláció a táblázat egy sorának felel meg.

Az egyed-kapcsolat diagram és leképezése relációs modellre

EK-diagram

Az egyed-kapcsolat modell konkrét adatmodellről függetlenül, szemléletesen adja meg az adatbázis szerkezetét.

Egyed vagy entitás

- a valós világ egy objektuma
- szeretnénk róla információt tárolni az adatbázisban
- egyedtípus: általánosságban jelent egy valós objektumot
- egyedpéldány: egy konkrét objektum
- gyenge egyed: ha az egyedet nem határozza meg egyértelműen attribútumainak semmilyen részhalmaza

Tulajdonság vagy attribútum

- az egyed egy jellemzője
- tulajdonságtípus vs tulajdonságpéldány
- az attribútumok egy olyan legszűkebb részhalmazát, amely egyértelműen meghatározza az egyedet, kulcsnak nevezzük

Kapcsolatok

- egyedek között alakulhatnak ki
- kapcsolattípus: pl felhasználó és üzenet között
- kapcsolatpéldány: pl Kis József és a 69420. üzenet
- kapcsolatoknak is lehet tulajdonsága

Azt a modellt, amelyben az adatbázis a tárolandó adatokat egyedekkel, tulajdonságokkal és kapcsolatokkal írja le, egyed-kapcsolat modellnek nevezzük, a hozzá kapcsolódó diagramot pedig egyed-kapcsolat diagrammnak.

A diagramon

- az egyedeket téglalappal
- a tulajdonságokat ellipszissel
- a kulcsot aláhúzással
- a kapcsolatokat rombusszal

jelöljük.

EK leképezése relációs adatmodellre

Egyedek leképezése

- minden egyedhez egy relációsémát írunk fel, melynek neve az egyed neve, attribútumai pedig az egyed attribútumai, kulcsa pedig az egyed kulcsa
- gyenge egyednél az attribútumokhoz hozzá kell venni a meghatározó kapcsolatokon keresztül csatlakozó egyedek kulcsattribútumait is, külső kulcsként

Összetett attr. leképezése

- összetett attribútumot helyettesítünk az őt alkotó elemi attribútumokkal

Többértékű attribútumok leképezése

- egyik lehetőség:

- eltekintünk attól, hogy többértékű, és egyszerű szöveggént tároljuk
- hátránya, hogy nem kezelhetők külön külön az elemek
- másik lehetőség:
 - minden sorból annyit veszünk fel, ahány értéke van a többértékű attribútumnak
 - hátránya a sok fölösleges sor
 - kulcsok elromlanak
 - kerülendő
- harmadik lehetőség
 - új táblát veszünk fel, ahova kigyűjtjük, hogy melyik sorhoz milyen értékei tartoznak a többértékű attribútumnak
 - akár külön kigyűjthetjük egy táblába az összes lehetséges értékét a többértékű attribútumnak, és egy kapcsolótáblával kötjük össze az egyeddel

Kapcsolatok leképezése

- minden kapcsolathoz felveszünk egy új sémát
- neve a kapcsolat neve, attribútumai a kapcsolódó egyedek kulcsattribútumai és a kapcsolat saját attribútumai
- meg kell határozni ennek a sémának is a kulcsát
- ha ez a kulcs megegyezik valamelyik kapcsolt egyed kulcsával, akkor ez a séma beolvasztható abba az egyedbe, ezt hívjuk konszolidációnak, ez a gyakorlatban egy lépésben is elvégezhető persze
- 1:1 kapcsolat esetén az egyik tetszőlegesen választott egyedbe beolvaszthatjuk a kapcsolat sémáját
- 1:N kapcsolat esetén az N oldali egyedet bővítjük a másik egyed kulcsattribútumaival, és a kapcsolat saját attribútumaival
- N:M kapcsolat esetén új sémát veszünk fel

Specializáló kapcsolatok leképezése

Minden megközelítésnek lehetnek hátrányai, mérlegelnünk kell

Első lehetőség

- főtípus és altípus is külön sémában, és az altípus attribútumai közé felvesszük a főtípus attribútumait is
- minden egyedpéldány csak egy táblában fog szerepelni

Második lehetőség

- minden altípushoz új séma, de abban csak a főtípus kulcsattribútumai jelennek meg
- minden egyedpéldány szerepel a saját altípusának táblájában és a főtípus táblájában is

Harmadik lehetőség

- egy közös tábla az összes lehetséges attribútummal
- minden sorban csak a releváns cellákat töltjük ki

Kulcsok fajtái

Szuperkulcs

- egyértelműen azonosítja a tábla sorait
- R(A) bármely két sora különbözik a szuperkulcson

- mivel a táblában általában nem engedünk meg ismétlődő sorokat, ezért ha az összes attribútumot vesszük, az mindig szuperkulcs

Kulcs

- olyan szuperkulcs, amelynek egyetlen valódi részhalmaza sem szuperkulcs
- ha egyelemű, egyszerű kulcsnak nevezzük
- ha többelemű, összetettnek
- előfordulhat, hogy van több kulcs is, ekkor kiválasztunk egyet
- a kiválasztott kulcsot elsődleges kulcsnak nevezzük

Külső kulcs

- másik, vagy ugyanazon séma elsődleges kulcsára vonatkozik

Mind az elsődleges kulcs és a külső kulcsok is a sémára vonatkozó feltételek, függetlenek az adatoktól

Funkcionális függőség

P és Q attribútumhalmazok, az R(A) sémán P-től funkcionálisan függ Q, ha bármilyen R feletti tábla esetén ha P-n megegyezik két sor, akkor Q-n is meg fog egyezni.

Triviális, ha Q részhalmaza P-nek, és nemtriviális, ha P-nek és Q-nak nincs közös attribútuma.

PI a felhasználónévtől funkcionálisan függ az email sokszor.

Normalizálás célja, normálformák

Tárolhatnánk az összes adatunkat egy nagy táblában is, de ilyenkor gondok merülhetnek fel az adatbázisműveletek során, illetve nagyon redundáns lenne az adattárolás. A normalizálás célja kisebb táblák létrehozása a redundancia elkerülése érdekében.

Normálformák

Dekompozíció segítségével megszüntetjük lépésről lépésre a redundanciát úgy, hogy a sémában lévő függőségekre egyre szigorúbb feltételeket adunk.

Elsődleges, másodlagos attribútum: szerepel a séma valamelyik kulcsában, ha nem akkor másodlagos
 Tranzitív, közvetlen függés: Ha X-től függ Z, és van olyan Y, hogy $X \rightarrow Y$ és $Y \rightarrow Z$, ellenkező esetben közvetlenül függ

1NF:

- Ha az attribútumok értéktartománya csak egyszerű adatokból áll (nincs többszörös vagy összetett attribútum)

2NF:

- Ha minden másodlagos attribútum teljesen függ bármely kulcstól

3NF:

- Minden másodlagos attribútum közvetlenül függ bármely kulcstól, azaz nincs tranzitív függés

BCNF:

- Egy relációséma Boyce-Codd normálformában van, ha bármely nemtriviális $L \rightarrow B$ függés esetén L szuperkulcs.

6ef0bd001a5ea93950836f58b711eb2c3bb3daee

2. Az SQL adatbázisnyelv: Az adatdefiníciós nyelv (DDL) és az adatmanipulációs nyelv (DML). Relációsémák definiálása, megszorítások típusai és létrehozásuk. Adatmanipulációs lehetőségek és lekérdezések

SQL

Structured Query Language

Arra szolgál, hogy adatokat kezeljünk vele

- beszúrás
- törlés
- módosítás
- lekérdezés

A nyelv elemeit két fő részre oszthatjuk.

Az adatdefiníciós nyelv

Ide tartoznak az adatbázisok, sémák, típusok definíciós utasításai, pl:

- CREATE DATABASE
- CREATE TABLE
- ALTER TABLE
- DROP TABLE
- CREATE TRIGGER

Az adat manipulációs nyelv

Ide tartoznak a beszúró, módosító, törlő, lekérdező utasítások.

- INSERT INTO
- UPDATE
- DELETE FROM
- SELECT

Egyes irodalmak különválasztják a lekérdező utasításokat a manipulációs utasításoktól.

Relációsémák definiálása, megszorítások típusai és létrehozásuk

Relációsémákat a CREATE TABLE utasítással hozhatunk létre. A sémák különböznek a tábláktól, és nevével ellentétben a CREATE TABLE utasítás csak a relációsémát hozza létre. A tábla már az adatrekordok halmazát jelenti.

Megszorítások

Oszlopfeltételek:

Csak az adott mezőre vonatkoznak

- PRIMARY KEY, az elsődleges kulcs
- UNIQUE, kulcs, minden érték egyszer fordulhat elő az oszlopban
- NOT NULL, az oszlop értéke nem lehet NULL, azaz kötelező kitölteni
- REFERENCES T(oszlop), a T tábla oszlop oszlopára vonatkozó külső kulcs
- DEFAULT tartalom, az oszlop alapértelmezett értéke tartalom lesz

Táblafeltételek

Ha több oszlopra is vonatkoznak feltételek, azt itt tudjuk megadni.

- PRIMARY KEY(oszloplista), az elsődleges kulcs
- UNIQUE (oszloplista), kulcs, minden érték egyszer fordulhat elő az oszlopban
- FOREIGN KEY (oszloplista) REFERENCES T(oszloplista), a T tábla oszloplista oszloplistájára vonatkozó külső kulcs

Külső kulcs feltételek és szabályok

Az integritás megőrzése szempontjából a külső kulcsokhoz meghatározhatjuk azt is, hogy hogyan viselkedjenek a hivatkozott kulcs törlése vagy módosítása esetén.

ON DELETE

- RESTRICT, ha van a törlendő rekord kulcsára van vonatkozó külső kulcs, megtiltjuk a törlést
- SET NULL, a törlendő rekord kulcsára hivatkozó külső kulcs értékét NULL-ra állítjuk
- NO ACTION, a törlendő rekord kulcsára vonatkozó külső kulcs értéke nem változik
- CASCADE, a törlendő rekord kulcsára hivatkozó külső kulcsú rekordok is törlődnek

ON UPDATE

- RESTRICT, ha van a módosítandó rekord kulcsára van vonatkozó külső kulcs, megtiltjuk a módosítást
- SET NULL, a módosítandó rekord kulcsára hivatkozó külső kulcs értékét NULL-ra állítjuk
- NO ACTION, a módosítandó rekord kulcsára vonatkozó külső kulcs értéke nem változik
- CASCADE, a módosítandó rekord kulcsára hivatkozó külső kulcsú rekordok is az új értékre változnak

Táblákra és attribútumokra vonatkozó megszorítások

Elsődleges feladata, hogy megelőzzük az adatbeviteli hibákat, és elkerüljük a hiányzó adatokat a kötelező mezőkből.

NOT NULL: a cella értékét kötelező kitölteni, nem lehet NULL

CHECK (feltétel): ellenőrző feltétel arra, hogy milyen értékeket vehet fel az adott oszlop

DOMAIN: értéktartomány egy oszlop értékeire vonatkozóan

Adatmanipulációs lehetőségek és lekérdezések

Adatok beszúrása:

Ha csak adott oszlopoknak akarunk értéket adni (pl mert nem kötelező, vagy alapértelmezett érték): INSERT INTO táblanév (oszloplista) VALUES (értéklista);

Ha minden oszlop értékét ki akarjuk tölteni: INSERT INTO táblanév VALUES (értéklista);

Adatok módosítása:

UPDATE táblanév SET oszlop=kifejezés [oszlop2=kifejezés2] [WHERE feltétel];

Módosítjuk egy vagy több oszlop értékét az adott táblában, azokon a sorokon, amelyek eleget tesznek a WHERE záradékban tett feltételnek.

Adatok törlése:

DELETE FROM táblanév [WHERE feltétel];

Töröljük az összes rekordot a táblából, amelyek megfelelnek a WHERE záradékban megadott feltételnek.

Lekérdezések:

SELECT oszloplista FROM tábla;

A megadott oszlopokat kilistázza az adott táblából. oszloplista helyére megadható *, ha az összes oszlopot listázni akarjuk.

Teljes szintaxisa:

SELECT[**DISTINCT**] oszloplista FROM táblalista [WHERE feltétel] [GROUP BY oszloplista] [HAVING csoportfeltétel] [ORDER BY oszloplista [DESC]];

DISTINCT: csak a különböző sorokat írja ki FROM táblalista: a táblalistában megadott táblákbó képez Descartes szorzatot **WHERE**: kiválasztás a feltétel szerint **GROUP BY**: csoportosítás az oszloplistában szereplő oszlopok szerint **HAVING**: a csoportosítás után a csoportokra vonatkozó feltétel **ORDER BY**: az oszloplistában szereplő adatok rendezése abc szerint növekvő vagy csökkenő sorrendben

összesítő függvények

Leggyakrabban a **GROUP BY**-jal együtt szoktuk használni, de enélkül is lehet. Leginkább a **SELECT** utáni oszloplistában, de a **where**-ben és a **having**-ban is használható. Az eredményoszlopokat **AS** kulcsszóval el is nevezhetjük.

MIN(oszlop): az oszlopban lévő minimumot adja vissza **MAX**(oszlop): maxot **AVG**(oszlop): az oszlop átlaga **SUM**(oszlop): az oszlop összege **COUNT**([**DISTINCT**] oszlop): az eredményben szereplő (különböző) rekordok száma

Természetes összekapcsolás

SELECT * FROM T1, T2 WHERE T1.X = T2.X;

X az most egy oszlop, egy kulcs-külső kulcs kapcsolat.

Erre használható még SQL-ben az INNER JOIN kulcsszó is.

```
SELECT * FROM T1, T2 INNER JOIN T2 ON T1.X = T2.X;
```

Használható még a NATURAL JOIN kifejezés is, de ez egy picit máshogy működik. Ennek a használatához a két tábla közös attribútumhalmaza ugyanazokat az oszlopneveket tartalmazza mindkét táblában és a párosított oszlopok típusa is megegyezik. Ebből kifolyólag nem kell megadnunk a kapcsolódó, kulcs és külső kulcs oszlopokat. A közös oszlop csak egy példányban jelenik majd meg.

```
SELECT * FROM T1 NATURAL JOIN T2;
```

Jobboldali, baloldali és teljes külső összekapcsolás

Valamelyik, vagy mindkét tábla összes rekordja szerepelni fog az eredményben.

Baloldali összekapcsolásnál a baloldali tábla minden rekordja megmarad, és ezekhez a rekordokhoz párosítjuk a jobboldali tábla rekordjait. Jobboldalinál pont fordítva. Teljes összekapcsolásnál pedig mindkét tábla összes rekordja megmarad, és mindenhol a hiányzó helyeken NULL értékek lesznek.

Lekérdezések eredményén, amikor ugyanannyi és ugyanolyan típusú oszlopot kérünk le, használhatunk halmazműveleteket is, pl UNION vagy INTERSECT.

3. Simítás/szűrés képtérben (átlagoló szűrők, Gauss simítás és mediánszűrés); élek detektálása (gradiens-operátorokkal és Marr-Hildreth módszerrel)

Simítás/szűrés képtérben

Átlagoló szűrés

Vesszük egy képpontnak egy környezetét, és vesszük ebben a környezetben az összes képpont átlagát. Ezzel az átlag lesz a képpont új értéke. Ezt az átlagolást konvolúcióval is végezhetjük, ahol a konvolúciós maszkunkban minden érték $1/n^2$, ha $n \times n$ -es a maszk.

- ez a fajta zajszűrés rontja az éleket
- minél nagyobb környezetet nézünk, annál erősebb a simító hatás
- haszna: csökkenti a zajt
- kára: gyengíti az éleket, homályossá teszi a képet
- súlyozott átlagolást is lehet csinálni - konvolúció
 - a legnagyobb súly az aktuális pontunknak legyen
 - ahogy távolodunk a ponttól, annál kisebbek legyenek a súlyok

Gauss simítás

- ahogy távolodunk a ponttól, annál kisebbek legyenek a súlyok
- erre nagyon jó a gauss harang

- minden sűrűségfüggvény integrálja 1
 - minél nagyobb a szigma, annál szélesebb, de annál alacsonyabb a harang
 - ezzel szépen lehet jeleket simítani
- binomiális együtthatók jól közelítik a normális eloszlás görbáját
- van 2D gauss is, harang alakú

hogyan lehet gauss függvényt közelíteni diszkrét értékekkel?

- vegyük a binomiális együtthatókat tartalmazó sorvektort, és osszuk el minden elemet 2^n -nel
- ezt szorozzuk össze a transzponáltjával, és így kapjuk a gauss görbe közelítését

Hozzájuthatunk így diszkrét gauss eloszlású $n \times n$ -es konvolúciós maszkokhoz, és az ilyenekkel vett konvolúció a Gauss szűrés. Az élek itt is rombolódnak.

Lehet olyat is, hogy csak akkor simítunk, ha az adott képpont intenzitásának környezeti átlagtól való eltérése meghalad egy T küszöbértéket.

Medián szűrés

medián = sorbarendezzük az értékeket, és a középsőt vesszük $\min \leq \text{med} \leq \max$

medián nem lineáris

medián szűrés: nézzük egy környezetét a pontnak, ezt rendezzük sorba, és a középső érték legyen a képpont új értéke. Só-bors zaj eltüntetésére szépen alkalmas. Tiszta képet kapunk, ha pl. 5×5 -ös környezetben nézve a 25 képpontból max 12 teljesen fekete vagy teljesen fehér képpont van megszünteti az egyedi, és kis kiterjedésű kiugrásokat jobban megőrzi az éleket, mint az átlagolás nagy kiterjedésű zajfoltoknál. Jel-elnyomó a zajt hagyja meg, és a lényeg tűnhet el.

Élek detektálása

él ott van a képen, ahol az intenzitás valamilyen irányban felugrik, vagy lecsökken.

élek nagyon fontosak a látásban, ahol markánsak az élek, azokat jól érzékeljük.

lehet ideális/lépcsős él lejtős él tető vonal zajos

felidézés: tangens függvény tangens: érintő iránytangense/meredeksége első derivált: hol vannak szélsőértékek, monotonitás derivált pozitív, nő, negatív, csökken

él ott van, ahol az intenzitásprofil első deriváltja nagy

Gradiens operátorokkal

többváltozós függvényeket is lehet deriválni, pl. parciálisan egyik változót lerögzítjük, és a másik szerint deriválunk. Gradiens - első parciális deriváltakból alkotott vektor. 2D-ben az érintőre merőleges vektor ennek van két komponense.

gradiens nagysága - magnitúdó valamilyen vektornorma - legyen kettes norma (euklideszi norma) ekkor a gradiens kettes normája a komponensek négyzetösszegének a négyzetgyöke első vektornormánál a

gradienskomponensek abszolútértékének az összegét nézzük

2D-ben a kettes vektornorma az a pitagorasz tételből jön

2D-ben van a gradiensnek iránya is $\arctan(y/x)$

él iránya a gradiensre merőleges

diszkrét gradiens operátorok

roberts, prewitt, sobel, frei-chen

mind a négy módszer konvolúciós maszkpárokat alkalmaz roberts operátor adott két 3x3-as mátrix, ha az egyikkel konvolválunk, akkor az x irányú parciális deriváltat közelítjük, ha a másikkal, akkor az y irányút igazából nem is kell konvolúció x: a képpont értékéből kivonjuk az északkeleti szomszédját y: a képpont értékéből kivonjuk az északnyugati szomszédját pro: könnyen számítható kontra: zajérzékeny

prewitt operátor itt is két 3x3-as maszk van, csak kicsit más, mint az előbb x: baloldali oszlop csupa 1, jobboldali csupa -1, középen 0 y: felső sor -1, alsó sor 1, középen 0

sobel operátor két 3x3 maszk ha négyzet mozaikon mintavételezett a képünk akkor ami két pixel élen osztozkodik (vízszintesen vagy függőlegesen szomszédos) akkor azok közelebb vannak egymáshoz, mintha csak csúcson érintkeznének

frei-chen operátor ugyanaz, mint a sobel, csak 2 helyett gyök(2)

gradiens maszk tervezése x irányban szimmetrikus ne húzzon el se balra, se jobbra asszimmetrikus ne húzzon el se fel, se le legyen az összege az elemeknek 0

8 irányban élt kereső gradiens operátorok compass operátorok

prewitt compass operátor 8 különböző maszkkal dolgozik, a 8 égtáj irányába maszkelemek összege 0

robinson-3 compass operátor 3-féle elem szerepel a maszkokban robinson-5 compass operátor 5-féle elem

kirsch compass operátor 0, -3, 5 értékek szerepelnek benne

Marr-Hildreth módszer

konvolváljuk a képet egy vagy több alkalmas LoG függvénnyel keressünk közös nulla átmeneteket nulla átmenet ott van, ahol adott pont kis környezetében előfordulnak pozitív és negatív értékek is eredménye mindig egy bináris éltérkép lehetnek fantomélek is de ez a gyakorlatban elhanyagolható

LoG a frekvenciatérben konvolúciós tétel szerint $f * \text{LoG}$ gyorsan számítható fourier-trafóval meg pontonkénti szorzással adott szigmára előre kiszámíthatjuk a sombrero fourier trafóját ezt is eltárolhatjuk

4. Alakreprezentáció, határ- és régió-alapú alakleíró jellemzők, Fourier leírás

Alakreprezentáció

Az alak/forma megítélésének fontos szerep jut a látásunkban. Az alak (shape) nem bír egzakt matematikai definícióval

A szegmentálást követően az objektumok kontúrjaiból vagy foltjaiból (attól függően, hogy határ- vagy régió-alapú szegmentálást vetettünk-e be) számos alakleíró jellemzőt vonhatunk ki. Hangsúlyozandó, hogy itt már elszakadhatunk a digitális képektől, némelyik jellemző csak egy szám, mások pedig összetett struktúrák is lehetnek.

Az alakleíró jellemzőket három osztályba soroljuk.

Határ alapú alakleíró jellemzők

- lánckód, alakleíró szám
- kerület, terület, kompaktság, cirkularitás
- közelítés poligonnal
- parametrikus kontúr, határvonal leíró függvény
- meredekségi hisztogram
- görbület, energia
- strukturális leírás

Freeman féle lánckód

- 4 vagy 8 szomszédok felé mutató vektort sorszámozza
- óramutató járásával ellentétes irányban növekszik
- kiválaszt a kontúron egy kezdőpontot
- egymás után írja a kontúrt körbekötő vektorok sorszámait
- a kontúr leírható egy négyes vagy nyolcas számrendszerbeli számmal, ez a lánckód
- pro: gyors, kompakt, eltolás-invariáns
- kontra: nem forgás- és skála-invariáns, zajérzékeny,
- Mivel a lánckód függ a kezdőpont megválasztásától, valamint nem invariáns még a 90 többszöröseivel való forgatásra sem, így bevezették az alakleíró számot, amit a lánckód első deriváltjából kapunk.

Kerület, terület számítása

- A kerület és a terület két gyakran bevetett alakleíró jellemző. Mindkettő származtatható a lánckódból is.
- 8-as lánckód esetén: kerület = $\sqrt{2}$ * (páratlan elemek száma) + páros elemek száma a lánckódban
- 4-es lánckód esetén: kerület = lánckód rendje (hossza)
- poligon területe 8-as lánckód esetén:
 - számontartunk egy y-t, ami kezdetben 0. Ehhez ha a lánckódban lévő következő szám "felfele" mutat hozzáadunk 1-et, ha "lefele", akkor kivonunk 1-et
 - a területváltozást szintén a lánckódban következő szám iránya határozza meg (y alapján), ahogy az alábbi képen is látszik
 - a területet úgy kapjuk, hogy foylton összeadogatjuk a területváltozásokat, és a végén vesszük az abszolútértékét

Kompaktság és cirkularitás

- $\text{kompaktság} = (\text{kerület})^2 / \text{terület}$
- $\text{cirkularitás} = \text{terület} / (\text{kerület})^2$

Parametrikus kontúr

- A parametrikus kontúr két egyváltozós függvénnyel reprezentálja a szegmenst. A kontúron végighaladva követjük az x és az y koordináták változásait.

Régió alapú alakleíró jellemzők

A határ-alapúakhoz hasonlóan, számos régió-alapú alakleíró jellemzőt javasoltak.

- befoglaló téglalap, rektangularitás
- főtengely, melléktengely, átmérő, excentricitás, főtengely szöge
- konvex burok, konvex kiegészítés, konkávitási fa, partícionált határ,
- vetületek, törés-költség
- topológiai leírások, Euler-szám, szomszédsági fa,
- váz,
- momentumok, invariáns momentumok

Befoglaló téglalap, rektangularitás

- álló befoglaló téglalap: az objektum koordinátáinak minimumai és maximumai megadják az álló befoglaló téglalap csúcsait.
- minimális befoglaló téglalap
- rektangularitás: Azt mondja meg, hogy az objektum „bedobozolásakor” mennyi a tárgy és a „levegő” által elfoglalt területek aránya, tehát \rightarrow alakzat területe / minimális befoglaló téglalap

Főtengely, melléktengely, átmérő, excentricitás, főtengely szöge

- főtengely: az alakzaton belül haladó leghosszabb egyenes szakasz
- melléktengely: az alakzaton belüli, a főtengelyre merőleges leghosszabb egyenes szakasz
- átmérő: a határ két legtávolabbi pontját köti össze. A főtengely hossza általában nem egyezik meg az átmérővel (csak a konvexeknél)
- excentricitás: a fő- és melléktengely hosszaránya $\rightarrow d_1/d_2$
- főtengely szöge: a főtengely és az x-tengely által bezárt szög

Konvex burok, konvex kiegészítés, konkávitási fa, partícionált határ

- konvex burok: az alakzatot tartalmazó minimális konvex alakzat
- konvex kiegészítés: a konvex burok és az alakzat különbsége
- konkávitási fa: A fa gyökere a kiindulási alakzat, az első szinten a konvex különbség alakzatai helyezkednek el, melyekre a faépítést rekurzív módon folytatjuk.
- partícionált határ: A konvex burok határát osztja fel részekre.

Vetületek, törés-költség

- vetületek: A bináris képekből képzett nem-negatív egészekből álló (1D) tömbök.
- törés-költség: A vetületek továbbbragozása, kiszűri a zajos képek oszlopaiban lévő „magányos” objektumpontokat.

Topológiai leírások, Euler-szám, szomszédsági fa,

- topológiai leírások
 - bináris kép: kétféle érték lehet benne, az 1-es az alakzatot (komponenst) reprezentálja feketével, míg a 0-s a háttér(lyukakat) fehérrel
 - komponens: maximálisan összefüggő fekete halmaz
 - üreg: a negált kép egy véges komponense
- Euler-féle szám: egyetlen egész szám ---> komponensek száma - üregek száma, rengeteg képre lehet az ugyanaz. Valamit elárul a képről, de önmagában keveset.
- összefüggőségi-fa: A bináris képekhez rendelt irányított gráf
 - minden egyes csúcs megfelel a kép egy (fehér vagy fekete) komponensének,
 - a gráf tartalmazza az (X,Y) éleket, ha az X komponens „körülveszi” a vele szomszédos Y komponens

Váz

A váz egy gyakran alkalmazott régió-alapú alakleíró jellemző, mely leírja az objektumok általános formáját. Alapvetően 3-féleképp határozhatjuk meg:

- a vázat az objektum azon pontjai alkotják, melyekre kettő vagy több legközelebbi határpont található.
- Az objektum határát (minden pontjában) egyidejűleg felgyűjtjük. A váz azokból a pontokból áll, ahol a tűzfrontok találkoznak és kioltják egymást. (Feltételezzük, hogy a tűzfrontok minden irányban egyenletes sebességgel, vagyis izotropikusan terjednek.)
- A vázat az objektumba beírható maximális (nyílt) hipergömbök középpontjai alkotják. Egy beírható hipergömb maximális, ha őt nem tartalmazza egyetlen másik beírható hipergömb sem.

Invariáns az eltolásra, elforgatásra és az uniform skálázásra

Momentumok, invariáns momentumok

Pro: számok, többszintű képekre is értelmezettek, invariánsak a főbb geometriai műveletekre
Bizonyos (centrális) momentumoknak geometriai jelentés is tulajdonítható, illetve fontos jellemzők kifejezhetők a segítségükkel, például súlypont.

Javasoltak viszont 7 ún. invariáns momentumot is (ld. 56. dia), amelyekhez nem társíthatók különösebb jelentések, de a belőlük alkotott rendezett hetesek (vagy akár hármasok, ha nem vesszük mindet figyelembe) jól jellemzik az objektumokat.

Fourier leírás


Ez egy transzformáción alapuló alakleírás


Transzformáljuk (hangsúlyozandó, hogy habár 2D képek szegmenseit jellemezzük, itt szigorúan 1D Fourier transzformációt alkalmazunk) a határ K darab mintavételezett pontjából (mint komplex $s(k)$ számokból) képzett \mathbf{s} vektort. Az eredményül kapott \mathbf{a} vektor (komplex $a(k)$ együtthatók) adják a Fourier leírást (vagyis

tartalmazza a Fourier együtthatókat, a transzformáció bázisfüggvényeinek súlyait). Az alakzat rekonstrukciójához az inverz Fourier-transzformációt kell végrehajtani.

A K darab Fourier együtthatóból visszakaphatnánk torzítatlanul az eredeti mintavételezett pontokat, az alakleíráshoz viszont nem az összes súlyt, hanem csak egy részüket tartjuk meg, mindössze $P < K$ darab együttható alapján térünk vissza a képtérbe

- ekkor a képtérben ismét K darab pontot kapunk vissza, de nem a kiindulás mintavételezettjeit.
- Az együtthatók egy részének eldobásával kapott leírás (a meghagyott együtthatók adják a jellemzést) voltaképpen egy veszteséges tömörítés: kevesebb adattal tudjuk jól-rosszul közelíteni a kiindulást.

Az alábbi kép azt mutatja, hogy a 64 kontúrponthall mintavételezett négyzetre csak sok együttható megtartásával tudunk négyzetfélét rekonstruálni.  alt text

A következő képen viszont tesztobjektum határa közel 3000 ponttal adott, és már 36 együttható is visszaadhat 3000 pontot úgy, hogy azok jól közelítik a kiindulási kontúrt.  alt text

5. Algoritmusok vezérlési szerkezetei és megvalósításuk C programozási nyelven. A szekvenciális, iterációs, elágazásos, és az eljárás vezérlés

Algoritmusok vezérlési szerkezetei és megvalósításuk C nyelven

Algoritmus: bármilyen jól definiált számítási eljárást, amely bemenetként bizonyos értéket vagy értékeket kap és kimenetként bizonyos értéket vagy értékeket állít elő. Vizsgálhatjuk helyesség, idő- és tárigény szempontjából

Algoritmus vezérlése: Az az előírás, amely az algoritmus minden lépésére (részműveletére) kijelöli, hogy a lépés végrehajtása után melyik lépés végrehajtásával folytatódjon (esetleg fejeződjék be) az algoritmus végrehajtása. Az algoritmusnak, mint műveletnek a vezérlés a legfontosabb komponense.

Négy fő vezérlési módot különböztetünk meg:

- Szekvenciális: Véges sok adott művelet rögzített sorrendben egymás után történő végrehajtása. (sorban egymás után)
- Szelekciós: Véges sok rögzített művelet közül adott feltétel alapján valamelyik végrehajtása. (if, else, if else, switch)
- Ismétléses: Adott művelet adott feltétel szerinti ismételt végrehajtása. (for, while, do while)
- Eljárás: Adott művelet alkalmazása adott argumentumokra, ami az argumentumok értékének pontosan meghatározott változását eredményezi. (void func, int func, double func, ...)

A vezérlési módok nyelvek feletti fogalmak.

A imperatív (algoritmikus) programozási nyelvekben ezek a vezérlési szerkezetek (közvetlenül vagy közvetve) megvalósíthatók.

A szekvenciális, iterációs, elágazásos, és az eljárás vezérlés

Szekvenciális vezérlés

Szekvenciális vezérlésről akkor beszélünk, amikor a P probléma megoldását úgy kapjuk, hogy a problémát P_1, \dots, P_n részproblémákra bontjuk, majd az ezekre adott megoldásokat (részalgoritmusokat) sorban egymás után hajtjuk végre.

P_1, \dots, P_n lehetnek elemi műveletek, vagy nem elemi részproblémák megnevezései.

Eljárásvezérlés

Eljárásvezérlésről akkor beszélünk, amikor egy műveletet adott argumentumokra alkalmazunk, aminek hatására az argumentumok értékei pontosan meghatározott módon változnak meg.

Az eljárásvezérlés fajtái:

- Eljárasművelet
- Függvényművelet

C-ben kicsi a különbség a kettő között.

Függvényművelet

- A matematikai függvény fogalmának általánosítása
- Ha egy részprobléma célja egy érték kiszámítása adott értékek függvényében, akkor a megoldást megfogalmazhatjuk függvényművelettel.
- A függvényművelet specifikációja tartalmazza:
 - A művelet elnevezését
 - A paraméterek felsorolását
 - Mindegyik paraméter adattípusát
 - A művelet hatásának leírását
 - A függvényművelet eredménytípusát
- Minden függvényben szerepelnie kell legalább egy return utasításnak
- Ha a függvényben egy ilyen utasítást hajtunk végre, akkor a függvény értékének kiszámítása befejeződik. A hívás helyén a függvény a return által kiszámított értéket veszi fel

Eljárasművelet

- Ha eljárást szeretnénk készíteni C nyelven, akkor egy olyan függvényt kell deklarálni, melynek eredménytípusa void. Ebben az esetben a függvény definíciójában nem kötelező a return utasítás, illetve ha mégis van ilyen, akkor nem adható meg utána kifejezés

Megvalósítás

- csak bemenő módú argumentumok vannak
- pointerekkel lehet kezelni kimenő argumentumokként is

Szelekciós vezérlés

Szelekciós vezérlésről akkor beszélünk, amikor véges sok rögzített művelet közül véges sok feltétel alapján választjuk ki, hogy melyik művelet kerüljön végrehajtásra.

Típusai:

- Egyszerű szelekciós vezérlés
- Többszörös szelekciós vezérlés
- Esetkiválasztásos szelekciós vezérlés
- A fenti három „egyébként” ággal

Egyszerű szelekciós vezérlés

- Egyszerű szelekció esetén egyetlen feltétel és egyetlen művelet van (ami persze lehet összetett).
- A vezérlés bővíthető úgy, hogy a 3. pontban üres művelet helyett egy B műveletet hajtunk végre, ekkor beszélünk egyébként ágról.

Egyszerű szelekciós utasítás megvalósítása C nyelven:

```
if(F) {
    A;
}
```

Többszörös szelekciós vezérlés

- Ha több feltételünk és több műveletünk van, akkor többszörös szelekcióról beszélünk.
- A többszörös szelekció is bővíthető egyébként ággal úgy, hogy egy nemüres B műveletet hajtunk végre a 3. lépésben.
- Legyenek F_i logikai kifejezések, A_i (és B) pedig tetszőleges műveletek. Az F_i feltételekből és A_i (és B) műveletekből képzett többszörös szelekciós vezérlés a következő vezérlési előírást jelenti:
 - Az F_i feltételek sorban történő kiértékelésével adjunk választ a következő kérdésre: Van-e olyan i amelyre teljesül, hogy az F_i feltétel igaz és az összes F_j feltétel hamis?
 - Ha van ilyen i , akkor hajtunk végre az A_i műveletet és fejezzük be az összetett művelet végrehajtását.
 - Egyébként, vagyis ha minden F_i feltétel hamis, akkor (hajtunk végre B-t és) fejezzük be az összetett művelet végrehajtását.

Többszörös szelekciós utasítás megvalósítása C nyelven:

```
if(F1) {
    A1;
} else if (F2) {
    A2;
}...
```

- C nyelvben nincs külön utasítás a többszörös szelekció megvalósítására, ezért az egyszerű szelekció ismételt alkalmazásával kell azt megvalósítani.
- Ez azon az összefüggésen alapszik, hogy a többszörös szelekció levezethető egyszerű szelekciók megfelelő összetételével.

Esetkiválasztós szelekciós vezérlés

Ha a többszörös szelekciós vezérlésben minden F_i feltételünk $K \in H_i$ alakú, akkor esetkiválasztásos szelekcióról beszélünk.

- Legyen K egy adott típusú kifejezés, legyenek H_i ilyen típusú halmazok, A_i (és B) pedig tetszőleges műveletek. A K szelektor kifejezésből, H_i kiválasztó halmazokból és A_i (és B) műveletekből képzett esetkiválasztásos szelekciós vezérlés a következő vezérlési előírást jelenti:
 - Értékeljük ki a K kifejezést és folytassuk a 2.) lépéssel.
 - Adjunk választ a következő kérdésre: Van-e olyan i ($1 \leq i \leq n$), amelyre teljesül, hogy a kiszámolt érték eleme a H_i halmaznak és nem eleme az összes H_j ($1 \leq j < i$) halmaznak?
 - Ha van ilyen i , akkor hajtsuk végre az A_i műveletet és fejezzük be az összetett művelet végrehajtását.
 - Egyébként, vagyis ha K nem eleme egyetlen H_i halmaznak sem, akkor (hajtsuk végre B -t és) fejezzük be az összetett művelet végrehajtását.
- A kiválasztó halmazok megadása az esetkiválasztásos szelekció kritikus pontja.
- Algoritmusok tervezése során bármilyen effektív halmazmegadást használhatunk, azonban a tényleges megvalósításkor csak a választott programozási nyelv eszközeit alkalmazhatjuk.

A switch utasítás: Ha egy kifejezés értéke alapján többféle utasítás közül kell választanunk, a switch utasítást használhatjuk. Megadhatjuk, hogy hol kezdődjön és meddig tartson az utasítás-sorozat végrehajtása. A switch utasítás szintaxisa C-ben:

```
switch(kifejezés) {  
    case konstans1:  
        A;  
        break;  
    case konstans2:  
        B;  
        break;  
    default:  
        D;  
}
```

- A szelektor kifejezés és a konstansok típusának meg kell egyeznie. Egy konstans legfeljebb egy case mögött és a default kulcsszó is legfeljebb egyszer szerepelhet egy switch utasításban.
- A default címke olyan, mintha a szelektor kifejezés lehetséges értékei közül minden olyat felsorolnánk, ami nem szerepel case mögött az adott switch-ben.
- A címkék (beleértve a default-ot is) sorrendje tetszőleges lehet, az nem befolyásolja, hogy a szelektor kifejezés melyik címkét választja.
- A szelektor kifejezés értékétől csak az függ, hogy melyik helyen kezdjük el végrehajtani a switch magját. Ha a végrehajtás elkezdődik, akkor onnantól kezdve az első break (vagy return) utasításig, vagy a switch végéig sorban hajtódnak végre az utasítások. Ebben a fázisban a további case és default címkéknek már nincs jelentősége.
- A H_i halmazok elemszáma tetszőleges lehet, viszont a case-ek után csak egy-egy érték állhat.

Ismétléses vezérlések

Ismétléses vezérlésen olyan vezérlési előírást értünk, amely adott műveletnek adott feltétel szerinti ismételt végrehajtását írja elő.

Az algoritmustervezés során a leginkább megfelelő ismétléses vezérlési formát használjuk, függetlenül attól, hogy a megvalósításra használt programozási nyelvben közvetlenül megvalósítható-e ez a vezérlési mód.

Ismétléses vezérlés képzését ciklusszervezésnek is nevezik, így az ismétlésben szereplő művelet ciklusmagnak hívjuk.

Az ismétlési feltétel szerint ötféle ismétléses vezérlést különböztetünk meg:

- Kezdőfeltételes
- Végfeltételes
- Számlálósos
- Hurok
- Diszkrét

Kezdőfeltételes ismétléses vezérlés

Kezdőfeltételes vezérlésről akkor beszélünk, ha a ciklusmag (ismételt) végrehajtását egy belépési (ismétlési) feltételhez kötjük.

- Legyen F logikai kifejezés, M pedig tetszőleges művelet. Az F ismétlési feltételből és az M műveletből (a ciklusmagból) képzett kezdőfeltételes ismétléses vezérlés a következő vezérlési előírást jelenti:
 - Értékeljük ki az F feltételt és folytassuk a 2.) lépéssel.
 - Ha F értéke hamis, akkor az ismétlés és ezzel együtt az összetett művelet végrehajtása befejeződött.
 - Egyébként, vagyis ha az F értéke igaz, akkor hajtsuk végre az M műveletet, majd folytassuk az 1.) lépéssel.
- A feltétel ellenőrzése a művelet előtt történik
- Ha az F értéke kezdetben hamis, az összetett művelet végrehajtása befejeződik anélkül, hogy az M művelet egyszer is végrehajtásra kerülne
- Ha az F értéke igaz, és az M művelet nincs hatással az F feltételre, akkor F igaz is marad, tehát az összetett művelet végrehajtása nem tud befejeződni. Ilyenkor végtelen ciklus végrehajtását írtuk elő.
- Fontos tehát, hogy az M művelet hatással legyen az F feltételre.

A while utasítás: Ha valamilyen műveletet mindaddig végre kell hajtani, amíg egy feltétel igaz, a while utasítás használható.

```
while(F) {  
    M;  
}
```

Végfeltételes ismétléses vezérlés

A végfeltételes ismétléses vezérlés alapvetően abban különbözik a kezdőfeltételes ismétléses vezérléstől, hogy a ciklusmag legalább egyszer végrehajtódik. Végfeltételes vezérlésről akkor beszélünk, ha a ciklusmag elhagyását egy kilépési feltételhez kötjük.

- Legyen F logikai kifejezés, M pedig tetszőleges művelet. Az F kilépési feltételből és az M műveletből (a ciklusmagból) képzett végfeltételes ismétléses vezérlés a következő vezérlési előírást jelenti:
 - Hajtsuk végre az M műveletet majd folytassuk a 2.) lépéssel.
 - Értékeljük ki az F feltételt és folytassuk a 3.) lépéssel.
 - Ha F értéke igaz, akkor az ismétléses vezérlés és ezzel együtt az összetett művelet végrehajtása befejeződött.
 - Egyébként, vagyis ha az F értéke hamis, akkor folytassuk az 1.) lépéssel.
- Ha az F értéke kezdetben hamis, és az M művelet nincs hatással F-re, akkor végtelen ciklust kapunk. Ha az F értéke kezdetben igaz, M legalább egyszer akkor is végrehajtásra kerül.
- A kezdő és végfeltételes vezérlések kifejezhetőek egymás segítségével.

A do while: utasítás Ha valamilyen műveletet mindaddig végre kell hajtani, amíg egy feltétel igaz, a do while utasítás használható. A művelet végrehajtása szükséges a feltétel kiértékeléséhez. A feltétel ellenőrzése a művelet után történik, így ha a feltétel kezdetben hamis volt, a műveletet akkor is legalább egyszer végrehajtjuk.

```
do {  
    M;  
} while (!F);
```

Számlálásos ismétléses vezérlések

Számlálásos ismétléses vezérlésről akkor beszélünk, ha a ciklusmagot végre kell hajtani sorban minden olyan értékére (növekvő vagy csökkenő sorrendben), amely egy adott intervallumba esik.

Legyen a és b egész érték, i egész típusú változó, M pedig tetszőleges művelet, amelynek nincs hatása a, b és i értékére.

Növekvő számlálásos ismétléses vezérlések:

- Az a és b határértékekből, i ciklusváltozóból és M műveletből (ciklusmagból) képzett növekvő számlálásos ismétléses vezérlés az alábbi vezérlési előírást jelenti:
 - Legyen $i = a$ és folytassuk a 2.) lépéssel.
 - Ha $b < i$ (i nagyobb mint a intervallum végpontja), akkor az ismétlés és ezzel együtt az összetett művelet végrehajtása befejeződött.
 - Egyébként, vagyis ha $i \leq b$, akkor hajtsuk végre az M műveletet, majd folytassuk a 4.) lépéssel.
 - Növeljük i értékét 1-gyel, és folytassuk a 2.) lépéssel.

Csökkenő számlálásos ismétléses vezérlések:

- Az a és b határértékekből, i ciklusváltozóból és M műveletből (ciklusmagból) képzett csökkenő számlálásos ismétléses vezérlés az alábbi vezérlési előírást jelenti:
 - Legyen $i = b$ és folytassuk a 2.) lépéssel.
 - Ha $i < a$, akkor az ismétlés és ezzel együtt az összetett művelet végrehajtása befejeződött.

- Egyébként, vagyis ha $a \leq i$, akkor hajtsuk végre az M műveletet, majd folytassuk a 4.) lépéssel.
- Csökkentsük i értékét 1-gyel, és folytassuk a 2.) lépéssel.

A for utasítás: Ha valamilyen műveletet sorban több értékére is végre kell hajtani, akkor a for utasítás használható.

```
for (i = a; i <= b; i++) {
    M;
}
for (kif1; kif2; kif3) {
    M;
}
```

C-ben a for utasítás általános alakja:

- A kif1 és kif3 többnyire értékadás vagy függvényhívás, kif2 pedig relációs kifejezés.
- Bármelyik kifejezés elhagyható, de a pontosvesszőknek meg kell maradniuk
- kif2 elhagyása esetén a feltételt konstans igaznak tekintjük, ekkor a break vagy return segítségével lehet kiugrani a ciklusból.

Hurok ismétléses vezérlés

Amikor a ciklusmag ismétlését a ciklusmagon belül vezéreljük úgy, hogy a ciklus különböző pontjain adott feltételek teljesülése esetén a ciklus végrehajtását befejezzük, hurok ismétléses vezérlésről beszélünk.

- Legyenek F_i logikai kifejezések, K_i és M_j pedig tetszőleges (akár üres) műveletek $1 \leq i \leq n$ és $0 \leq j \leq n$ értékekre. Az F_i kijárat feltételekből, K_i kijárat műveletekből és az M_i műveletekből képzett hurok ismétléses vezérlés a következő előírást jelenti:
 - Az ismétléses vezérlés következő végrehajtandó egysége az M_0 művelet.
 - Ha a végrehajtandó egység az M_j művelet, akkor ez végrehajtódik. $j = n$ esetén folytassuk az 1.) lépéssel, különben pedig az F_{j+1} feltétel végrehajtásával a 3.) lépésben.
 - Ha a végrehajtandó egység az F_i feltétel ($1 \leq i \leq n$), akkor értékeljük ki. Ha F_i igaz volt, akkor hajtsuk végre a K_i műveletet, és fejezzük be a vezérlést. Különben a végrehajtás az M_i művelettel folytatódik a 2.) lépésben.
- A kezdő- és végfeltételes ismétléses vezérlések speciális esetei a hurok ismétléses vezérlésnek.
- A C nyelvben a ciklusmag folyamatos végrehajtásának megszakítására két utasítás használható:
- break: Megszakítja a ciklust, a program végrehajtása a ciklusmag utáni első utasítással folytatódik. Használható a switch utasításban is, hatására a program végrehajtása a switch utáni első utasítással folytatódik.
- continue: Megszakítja a ciklusmag aktuális lefutását, a vezérlés a ciklus feltételének kiértékelésével (while, do while) illetve az inkrementáló kifejezés kiértékelésével (for) folytatódik.

Diszkrét ismétléses vezérlés:

Diszkrét ismétléses vezérlésről akkor beszélünk, ha a ciklusmagot végre kell hajtani egy halmaz minden elemére tetszőleges sorrendben.

- Legyen x egy T típusú változó, H a T értékkészletének részhalmaza, M pedig tetszőleges művelet, amelynek nincs hatása x és H értékére. A H halmazból, x ciklusváltozóból és M műveletből (ciklusmagból) képzett diszkrét ismétléses vezérlés az alábbi vezérlési előírást jelenti:
 - Ha a H halmaz minden elemére végrehajtottuk az M műveletet, akkor vége a vezérlésnek.
 - Egyébként vegyük a H halmaz egy olyan tetszőleges e elemét, amelyre még nem hajtottuk végre az M műveletet, és folytassuk a 3.) lépéssel.
 - Legyen $x = e$ és hajtsuk végre az M műveletet, majd folytassuk az 1.) lépéssel.
- A H halmaz számossága határozza meg, hogy az M művelet hányszor hajtódik végre. Ha a H az üres halmaz, akkor a diszkrét ismétléses vezérlés az M művelet végrehajtása nélkül befejeződik.
- A diszkrét ismétléses vezérlésnek nincs közvetlen megvalósítása a C nyelvben.

6. Egyszerű adattípusok: egész, valós, logikai és karakter típusok és kifejezések. Az egyszerű típusok reprezentációja, számábrázolási tartományuk, pontosságuk, memória igényük és műveleteik. Az összetett adattípusok és a típusképzések, valamint megvalósításuk C nyelven. A pointer, a tömb, a rekord és az unió típus. Az egyes típusok szerepe, használata

Egyszerű adattípusok: egész, valós, logikai és karakter típusok és kifejezések. Az egyszerű típusok reprezentációja, számábrázolási tartományuk, pontosságuk, memória igényük és műveleteik

Az elemi adattípusok értékeit nem lehet önmagukban értelmes részekre bontani.

Ha a nyelv szintaktikája szerint a program egy adott pontján típusnak kellene következnie de az hiányzik, a fordító a típus helyére automatikusan `int`-et helyettesít.

Egész típusok

A C nyelvben az egész típus az `int`.

Az `int` típus értékkészlete az alábbi kulcsszavakkal módosítható:

- `signed`: A típus előjeles értékeket fog tartalmazni (`int`, `char`).
- `unsigned`: A típus csak előjeltelen, nemnegatív értékeket fog tartalmazni (`int`, `char`).
- `short`: Rövidebb helyen tárolódik, így kisebb lesz az értékkészlet (`int`).
- `long`: Hosszabb helyen tárolódik, így bővebb lesz az értékkészlet (`int`). Duplán is alkalmazható (`long long`).

Az egész típusok az értékkészlet határain belüli minden egész értéket pontosan ábrázolnak.

Az egyes gépeken az egyes típusok mérete más-más lehet, de minden C megvalósításban teljesülnie kell a $\text{sizeof}(\text{short}) \leq \text{sizeof}(\text{int}) \leq \text{sizeof}(\text{long}) \leq \text{sizeof}(\text{long long})$ relációnak.

A C nyelv különféle egész adattípusai az értékalmazukban különböznek egymástól, az értelmezett műveletükben megegyeznek

Az egész adattípusokon általában az 5 matematikai alpműveletet és az értékadás műveletét értelmezzük, de C nyelven ennél jóval többet.

Értékadó művelet jobb oldalán álló kifejezés kiértékelése független attól, hogy a bal oldalon milyen típusú változó van.

A / művelet két egész értékre alkalmazva maradékos osztást jelent!

Tárolás:

n bites tárterületnek 2^n állapota van, vagyis egy n biten tárolt adattípusnak legfeljebb ennyi különböző értéke lehet.

Egész típusoknál a kettes komplementet szokás használni, ha negatív értékek is szerepelhetnek az értékalmazban.

Kettes komplement:

- van egy pozitív számunk, és annak keressük a negatív párját
- a számot kettes számrendszerben felírjuk
- invertáljuk az összes bitet
- majd hozzáadunk a végén egyet
- a kapott szám lesz a szám ellentettje

Értékalmaz mérete:

Ha negatív számok nem szerepelnek az értékalmazban, akkor az értékalmaz a $[0 \dots 2^n - 1]$ zárt intervallum. Ha az értékalmazban negatív számok is szerepelnek, akkor az értékalmaz a $[-2^{(n-1)} \dots 2^{(n-1)} - 1]$ zárt intervallum.

Műveletei:

- bitenkénti
 - negáció
 - és
 - vagy
 - kizáró vagy
 - balra léptetés
 - jobbra léptetés

Karakter típus

A char adattípus a C nyelv eleve definiált elemi adattípusa, értékészlete 256 elemet tartalmaz.

A char adattípus egészként is használható, de alapvetően karakterek (betűk, számjegyek, írásjelek) tárolására való.

- Hogy melyik értékhez melyik karakter tartozik, az az alkalmazott kód táblázattól függ.

- Bizonyos karakterek (általában a rendezés szerint első néhány) vezérlő karakternek számítanak, és nem megjeleníthetők.

Egy C programban karakter értékeket megadhatunk

- karakterkóddal számértékként, vagy
- aposztrófok közé írt karakterrel

A speciális karaktereket, illetve magát az aposztrófot (és végső soron tetszőleges karaktert is) escape-szekvenciákkal lehet megadni. Az escape-szekvenciákat a \ (backslash) karakterrel kell kezdeni.

Konvertáljunk egy tetszőleges számjegy karaktert (ch) a neki megfelelő egész számmá és egy egyjegyű egészet (i) karakterré:

```
i = ch - '0';
ch = i + '0';
```

Valós típusok

A C nyelvben a valós adattípusok a float és double.

A double adattípus az alábbi kulcsszóval módosítható: - long: Implementációfüggő módon 64, 80, 96 vagy 128 bites pontosságot megvalósító adattípus

A valós adattípusok az értékkészlet határain belül sem képesek minden valós értéket pontosan ábrázolni. Viszont az értékkészlet határain belüli minden a valós értéket képesek egy típusfüggő e relatív pontossággal ábrázolni, az a-hoz legközelebbi a típus által pontosan ábrázolható x valós értékkel.

- A C nyelv különféle valós adattípusai az értékalmazukban különböznek egymástól, az értelmezett műveletükben megegyeznek.
- Valós kifejezésben bármely valós vagy egész típusú tényező (akár vegyesen többféle is) szerepelhet.
- Valós konstans típusa double, vagy a száMLEÍrásban megadott típus (f, l suffix).
- Értékadó művelet jobb oldalán álló kifejezés kiértékelése független attól, hogy a bal oldalon milyen típusú változó van.
- A típus pontatlansága miatt az == műveletet nagyon körültekintően kell használni!

Ábrázolása:

Egy valós értéket tároló memóriaterület három részre osztható: az előjelbitet, a törtet és az exponenciális kitevőt kódoló részre.

- Az előjelbit 0 értéke a pozitív, 1 értéke a negatív számokat jelöli
- A számot kettes számrendszerben $1.m \times 2^k$ alakra hozzuk, majd az m számjegyeit eltároljuk a törtnek, a k-nak egy típusfüggő b konstanssal növelt értékét pedig a kitevőnek fenntartott részen.
- Így a tört rész hossza az ábrázolás pontosságát (az értékes számjegyek számát), a kitevő pedig az értéktartomány méretét határozza meg.
- Nagyon kicsi számokat speciálisan $0.m \times 2^{(1-b)}$ alakban tárolhatunk, ekkor a kitevő összes bitje 0.
- Ha a kitevő összes bitje 1, az csupa 0 bitből álló tört esetén a ∞ , minden más esetben NaN értéket jelenti.

- A 32/64 bites float/double az 1 előjelbit mögött 8/11 biten a kitevő $b = 127$ -tel/1023-mal növelt értékét, majd 23/52 biten a törtet tárolja.

Logikai típus

A C nyelvnek csak a C99 szabvány óta része a logikai (`_Bool`) típus (melynek értékkészlete a $\{0, 1\}$ halmaz), de azért logikai értékek persze előtte is keletkeztek.

A műveletek eredményeként keletkező logikai hamis értéket a 0 egész érték reprezentálja, és a 0 egész érték logikai értéként értelmezve hamisat jelent.

A műveletek eredményeként keletkező logikai igaz értéket az 1 egész érték reprezentálja, de bármely 0-tól különböző egész érték logikai értéként értelmezve igazat jelent.

`stdbool.h`-ban definiált a `bool` típus és a `true`, `false` konstansok

Konstansként is definiálhatjuk, pl

```
#define TRUE 1
#define FALSE 0
```

Az összetett adattípusok és a típusképzések, valamint megvalósításuk C nyelven. A pointer, a tömb, a rekord és az unió típus. Az egyes típusok szerepe, használata

Összetett adattípusok, típusképzések

Az összetett adattípusok értékei tovább bonthatóak, további értelmezésük lehetséges.

A C nyelv összetett adattípusai:

- Pointer típus
 - Függvény típus
- Tömb típus
 - Sztringek
- Rekord típus
 - Szorzat-rekord
 - Egyesítési-rekord

Pointer típus

Az eddigi tárgyalásunkban szerepelt változók statikusak abban az értelemben, hogy létezésük annak a blokknak a végrehajtásához kötött, amelyben a változó deklarálva lett. A programozónak a deklaráció helyén túl nincs befolyása a változó létesítésére és megszüntetésére.

Az olyan változókat, amelyek a blokkok aktivizálásától függetlenül létesíthetők és megszüntethetők, dinamikus változóknak nevezzük.

Dinamikus változók megvalósításának általános eszköze a pointer típus.

Egy pointer típusú változó értéke (első megközelítésben) egy meghatározott típusú dinamikus változó.

Pointer típusú változót a * segítségével deklarálhatunk:

```
típus * változónév;
```

Az eddigiek során lényegében azonosítottuk a változóhivatkozást és a hivatkozott változót.

A dinamikus változók megértéséhez viszont világosan különbséget kell tennünk az alábbi három fogalom között:

- Változóhivatkozás
- Hivatkozott változó
- Változó értéke

A változóhivatkozás szintaktikus egység, meghatározott formai szabályok szerint képzett jelsorozat egy adott programnyelven, tehát egy kódrészlet.

A változó a program futása során a program által lefoglalt memóriaterület egy része, amelyen egy adott (elemi vagy összetett) típusú érték tárolódik.

Különböző változóhivatkozások hivatkozhatnak ugyanarra a változóra, illetve ugyanaz a változóhivatkozás a végrehajtás különböző időpontjaiban különböző változókra hivatkozhat.

Egy változóhivatkozáshoz nem biztos, hogy egy adott időben tartozik hivatkozott változó.

Műveletek:

- NULL
 - NULL, nem tartozik hozzá dinamikus változó
- Létesít
 - `x = malloc(sizeof(E))`
- Értékadás
 - `x = y`
- Törlés
 - `free(x)`
- Dereferencia: A pointer által mutatott dinamikus változó elérése, eredménye egy változóhivatkozás.
 - `*x`
- Egyenlő
 - `p == q`
- NemEgyenlő
 - `p != q`

A memóriaműveletekhez szükség van az `stdlib.h` vagy a `memory.h` használatára.

`malloc(S)`, lefoglal egy `S` méretű memóriaterületet `sizeof(E)`, megmondja, hogy egy `E` típusú érték mekkora helyet igényel a memóriában `malloc(sizeof(E))`, létrehoz egy `E` típusú érték tárolására is alkalmas változó `free(p)`, felszabadítja a `p`-hez tartozó memóriaterületet, ezután a `p`-hez nem lesz érvényes változóhivatkozás

Linux alatt logikailag minden programnak saját memória-tartománya van, amin belül az egyes memóriacímeket egy sorszám azonosítja.

Pointer típusú változó 32 bites rendszereken 4 bájt, 64 bites rendszereken 8 bájt hosszban a hozzá tartozó dinamikus változóhoz foglalt memóriamező kezdőcímét (sorszámát) tartalmazza.

A pointer értéke tehát (második megközelítésben) értelmezhető egy tetszőleges memóriacímként is, amely értelmezés egybeesik a pointer megvalósításával.

Ilyen módon viszont értelmezhetjük a címképző műveletet, ami egy változó memóriabeli pozícióját, címét adja vissza.

- Cím
 - `p = &x`

A `void*` egy speciális, úgynevezett típustalan pointer. Az ilyen típusú pointerok „csak” memóriacímek tárolására alkalmasak, a dereferencia művelet alkalmazása rájuk értelmetlen. Viszont minden típusú pointerrel kompatibilisek értékadás és összehasonlítás tekintetében.

Tömb típus

Algoritmusok tervezésekor gyakran előfordul, hogy adatok sorozatával kell dolgozni, vagy mert az input adatok sorozatot alkotnak, vagy mert a feladat megoldásához kell.

Tegyük fel, hogy a sorozat rögzített elemszámú (n) és mindegyik komponensük egy megadott (elemi vagy összetett) típusból (E) való érték.

Ekkor tehát egy olyan összetett adathalmazzal van dolgunk, amelynek egy eleme $A = (a_0, \dots, a_{n-1})$, ahol $a_i \in E, \forall i \in (0, \dots, n-1)$ -re.

Ha az ilyen sorozatokon a következő műveleteket értelmezzük, akkor egy (absztrakt) adattípushoz jutunk, amit Tömb típusnak nevezünk.

Jelöljük ezt a Tömb típust T -vel, a $0, \dots, n-1$ intervallumot pedig I -vel.

Műveletek

- Kiolvas
 - a sorozat i . elemének kiolvasása egy változóba
- Módosít
 - a sorozat i . elemének módosítása egy E típusú értékre
- Értékadás
 - a változó felveszi a tömb értékét

Tömb típusú változót az alábbi módon deklarálhatunk:

```
típus változónév[elemszám];
```

Tömbelem hivatkozásra a `[]` operátort használjuk.

Ez egy olyan tömbökön értelmezett művelet C-ben, ami nagyon magas precedenciával rendelkezik és balasszociatív.

Egy tömbre a tömbindexelés operátort (megfelelő index használatával) alkalmazva a tömb adott elemét változóként kapjuk vissza.

Rekord típus

A tömb típus nagyszámú, de ugyanazon típusú adat tárolására alkalmas.

Problémák megoldása során viszont gyakran előfordul, hogy különböző típusú, de logikailag összetartozó adatelemek együttesével kell dolgozni.

Az ilyen adatok tárolására szolgálnak a rekord típusok, ezek létrehozására pedig a rekord típusképzések.

Ha az egyes típusú adatokat egyszerre kell tudnunk tárolni, szorzat-rekordról beszélünk.

Az új adattípusra a $T = \text{Rekord}(T_1, \dots, T_k)$ jelölést használjuk és szorzat-rekordnak vagy struktúrának nevezzük.

- kiolvas
- módosít
- értékadás

```
typedef struct T {  
    T1 M1;  
    ...  
    Tk Mk;  
} T;
```

A fenti típusképzésben az M_1, \dots, M_k azonosítókat mezőazonosítóknak (tagnak, member-nek) hívjuk és lokálisak a típusképzésre nézve.

Az absztrakt típus műveletei mezőhivatkozások segítségével valósíthatóak meg.

A mezőhivatkozásra a $.$ operátort használjuk. Ez egy olyan rekordokon értelmezett művelet C-ben, ami nagyon magas precedenciával rendelkezik és balasszociatív.

Egy rekordra a mezőkiválasztás operátort (megfelelő mezőnévvel) alkalmazva a rekord mezőjét változóként kapjuk vissza.

Unió típus

Ha az egyes típusú adatokat nem kell egyszerre tárolni, egyesített-rekordról beszélünk

A T halmazon is a szorzat rekordhoz hasonló módon értelmezhetünk kiolvasó és módosító műveletet.

Az új adattípust a T_0 változati típusból és T_1, \dots, T_k egyesítési-tag típusokból képzett egyesített-rekord típusnak nevezzük.


```
T1 M1;  
...  
Tk Mk;  
} T;
```

A union típusú változó számára foglalt memória mérete, amely a `sizeof` függvénnyel lekérdezhető:

```
sizeof(T) = max{sizeof(T1), ..., sizeof(Tk)}
```

Valamennyi változati mező ugyanazon a memóriacímen kezdődik, ami megegyezik a teljes union típusú érték címével (azaz minden mező eltolása, offset-je 0).

7. Objektum orientált paradigma és annak megvalósítása a JAVA és C++ nyelvekben. Az absztrakt adattípus, az osztály. Az egységbe zárás, az információ elrejtés, az öröklődés, az újrafelhasználás és a polimorfizmus. A polimorfizmus feloldásának módszere

Objektum orientált paradigma

Az objektum orientál paradigma az objektumok fogalmán alapuló programozási paradigma. Az objektumok egységbe foglalják az adatokat és a hozzájuk tartozó műveleteket. A program egymással kommunikáló objektumok összességéből áll melyek használják egymás műveleteit és adatait.

Az objektum-orientáltság három alapillére:

- Egységbezárás és adatelrejtés (Encapsulation & information hiding)
- Újrafelhasználás, polimorfizmus és öröklődés (Reusability, polymorphism & inheritance)
- Magasabb fokú absztrakció

Egységbezárás és adatelrejtés

Az egységbe zárás azt fejezi ki, hogy az összetartozó adatok és függvények, eljárások együtt vannak, egy egységbe tartoznak. További fontos fogalom az adatelrejtés, ami azt jelenti, hogy kívülről csak az férhető hozzá közvetlenül, amit az objektum osztálya megenged.

Ha az objektum, illetve osztály elrejt az összes adattagját, és csak bizonyos metódusokon keresztül férhetnek hozzá a kliensek, akkor az egységbe zárás az absztrakciót és információelrejtés erős formáját valósítja meg

Az osztály és objektum

Absztrakt adattípus: Az adattípus leírásának legmagasabb szintje, amelyben az adattípust úgy specifikáljuk, hogy az adatok ábrázolására és a műveletek implementációjára semmilyen előírást nem adunk.

Osztály: Egy absztrakt adattípus. Az adattagokból és a rajta elvégezhető műveleteket zárja egy egységbe. Egészen konkrétan objektumok csoportjának leírása, amelyeknek közös az attribútumaik, operációik és szemantikus viselkedésük van. Ugyanúgy viselkedik, mint minden egyéb primitív típus, tehát pl. változó (objektum) hozható létre belőlük.

- **Létrehozás:** Java-ban és C++-ban is a `class` kulcsszóval tudunk osztályokat definiálni. Az osztályokból tetszőleges mennyiségben létrehozhatunk példányokat, azaz objektumokat.

Objektum: Egy változó, melynek típusa valamely objektumosztály, vagyis az osztály egy példánya amely rendelkezik állapottal, viselkedéssel, identitással. Az objektumok gyakran megfeleltethetők a való élet objektumainak vagy egyedeinek

- **állapot:** Egy az objektum lehetséges létezési lehetőségei közül (a tulajdonságok aktuális értéke, pl: `lámpaBekapcsolva` `true` vagy `false`)
- **viselkedés:** Az objektum viselkedése annak leírása, hogy az objektum hogy reagál más objektumok kéréseire. (metódusok, pl: `lámpa.bekapcsol()`)
- **identitás:** Minden objektum egyedi, még akkor is, ha éppen ugyanabban az állapotban vannak, és ugyanolyan viselkedést képesek megvalósítani.

Információ elrejtése

A láthatóságok segítségével tudjuk szabályozni adattagok, metódusok elérését, ugyanis ezeket az objektumorientált paradigma értelmében korlátozni kell, kívülről csak és kizárólag ellenőrzött módon lehessen ezeket elérni, használni.

Az adattagok, és metódusok láthatóságának vezérléséhez vannak kulcsszavak, amelyekkel megfelelően el tudjuk rejteni őket.

Láthatósági opciók

- `public`: mindenhol látható
- `protected`: csak az osztály `scope`-ján belül, illetve a később az adott osztályból származtatott gyerekosztályokon belül lehet hivatkozni.
- `private`: csak az adott osztályon belül lehet hivatkozni rá

(Java-ban alapértelmezetten `package private` (az adott package-n belül `public`, egyébként `private`) a láthatóság, míg C++-ban `private`)

Törekedni kell a minél nagyobb adatbiztonságra és információ elrejtésre: az adat tagok láthatósága legyen `private`, esetleg indokolt esetben `protected`.

Öröklődés

Osztályok között értelmezett viszony, amely segítségével egy általánosabb típusból (ősosztály) egy sajátosabb típust tudunk létrehozni (utódosztály). Az utódosztály adatokat és műveleteket örököl, kiegészíti ezeket saját adatokkal és műveletekkel, illetve felülírhat bizonyos műveleteket. A kód újrafelhasználásának egyik módja. Megkülönböztetünk egyszeres és többszörös örökítést.

A hasonlóság kifejezése az ősz felé az általánosítás. A különbség a gyerek felé a specializálás.

Java: az extends kulcsszóval tudjuk jelezni, hogy az adott osztály egy másik osztálynak a leszármazottja. Java-ban egyszeres öröklődés van, vagyis egy osztály csak is egy ősz osztályból származhat (viszont több interfészt implementálhat)

- super: segítségével gyerekosztályból hivatkozhatunk szülőosztály adattagjaira és metódusaira.

C++: Az osztály neve után vesszővel elválasztva lehet megadni az ősz osztályokat és velük együtt a láthatóságaikat. Lehetőség van többszörös öröklődésre is

- Az öröklődés során lehetőség van az ősz osztály tagjainak láthatósági opcióján változtatni. Ezt az ősz osztályok felsorolásakor kell definiálni. Az változtatás csak szigorítást (korlátozást) jelenthet. Az alábbi táblázat a gyermek osztálybeli láthatóságot mutatja be az ősz osztálybeli láthatóság és a módosítás függvényében:

Virtuális öröklődés

Többszörös öröklődésnél előfordulhat olyan eset, amikor egy-egy ősz osztály az öröklődési hierarchia különböző pontján ismét megjelenik. Ekkor a gyermek osztályban ennek az ősz osztálynak több példánya jelenhet meg. Erre néhány esetben nincs szükség, például ha az ősz osztály csak egy eljárás-erőforrás, akkor minden esetben elegendő egyetlen előfordulás a gyermek osztályokban.

A virtuális ősz osztályt az öröklődésnél az ősz osztályok felsorolásakor virtual módosítóval kell jelezni.

(Ha nem adom meg a virtual módosító szót, akkor az A osztály többször fog megjelenni a D osztály példányaiban. Hivatkozásnál mindig meg kell mondani, hogy az A melyik példányáról van szó: C::A::m_iN, B::A::m_iN.)

Újrafelhasználás, Polimorfizmus:

Az újrafelhasználhatóság az OOP egyik legfontosabb előnye.

Az a jelenség, hogy egy változó nem csak egyfajta típusú objektumra hivatkozhat a polimorfizmus.

A polimorfizmus lehetővé teszi számunkra, hogy egyetlen műveletet különböző módon hajtsunk végre. Más szavakkal, a polimorfizmus lehetővé teszi egy interfész definiálását és több megvalósítást. Az objektumok felcserélhetőségét biztosítja. Az objektumok őstípusai alapján kezeljük, így a kód nem függ a specifikus típusoktól.

Polimorfizmusra két lehetőség van:

- statikus polimorfizmus (korai hozzárendelés) - a hívott metódus nevének és címének összerendelése szerkesztéskor történik meg. A futtatható programban már fix metóduscímek találhatók. (statikus, private, final metódusok)
- dinamikus polimorfizmus (késői hozzárendelés) - metódus nevének és címének hozzárendelése a hívás előtti sorban történik, futási időben

Virtuális eljárások

Egy virtuális eljárás címének meghatározása indirekt módon, futás közben történik.

Java-ban eleve csak virtuális eljárások vannak (kivéve a final metódusokat, amelyeket nem lehet felüldefiniálni és a private metódusokat, amelyeket nem lehet örökölni)

C++-ban a virtuális függvénytábla tartja nyilván a virtuális eljárások címeit. A VFT táblázat öröklődik, feltöltéséről a konstruktor gondoskodik. A származtatott osztály konstruktora módosítja a virtuális függvénytáblát, kijavítja az őosztályból örökölt metóduscímeket. Amikor a konstruálási folyamat véget ér, a VFT táblázat minden sora értéket kap, mégpedig a ténylegesen létrehozott osztálynak megfelelő metódus címeit. A VFT táblázat sorai ezután már nem változnak meg.

- Virtuális eljárásokat a virtual kulcsszóval tudunk létrehozni. Az újrafelhasználás során nagy valószínűséggel módosításra kerülő eljárásokat a szülő osztályokban célszerű egyből virtuálisra megírni, mert ezzel jelentős munkát lehet megtakarítani a későbbiekben.

Absztrakt osztály, interfész

Java: Absztrakt osztályok

- Az abstract kulcsszóval hozható létre.
- Egy absztrakt osztályból nem hozható létre objektum.
- Tartalmazhat absztrakt metódusokat (absztrakt metódusnak nincs implementációja, azaz törzse), illetve nem absztraktokat
- Gyerek osztályban az abstract metódusokat felül KELL definiálni, ha példányosítható osztályt szeretnénk
- Ha egy osztály rendelkezik legalább egy absztrakt metódussal, akkor osztálynak is absztraktnak kell lennie
- Lehetnek adattagjai

Interfész

- Az interface kulcsszóval lehet létrehozni
- Egy speciális absztrakt osztály
- Nincsenek sem megvalósított metódusok, sem adattagok. Csupán metódus deklarációkat tartalmaz
- Gyerekosztályban az implements kulcsszóval lehet implementálni

C++: Absztrakt osztályok:

A törzs nélküli virtuális eljárásokat pure virtual eljárásoknak nevezzük (pl.: virtual int getArea() = 0;). A pure virtual eljárás egy üres (NULL) bejegyzést foglal el a VFT (Virtual Function Table) táblázatban. Ha egy osztály ilyen eljárást tartalmaz, akkor azt absztrakt osztálynak nevezzük amiatt, mert ebből az osztályból objektum példányokat létrehozni nem lehet. A gyermek osztályokban minden pure virtual eljárást megfelelő törzsszel kell ellátni, ezt a fordító ellenőrzi. Amíg egyetlen pure virtual eljárás is marad, az osztály absztrakt lesz.

8. Objektumok életciklusa, létrehozás, inicializálás, másolás, megszüntetés. Dinamikus, lokális és statikus objektumok létrehozása. A statikus adattagok és metódusok, valamint szerepük a programozásban.

Operáció és operátor overloading a JAVA és C++ nyelvekben. Kivételkezelés

Objektumok létrehozása

Az objektumokat Java-ban és C++-ban is tárolhatjuk statikusan (az adatszegmensben), a veremben (lokálisan) vagy a heapben (dinamikusan).

Java-ban az objektumok mindig a heap-ben keletkeznek, kivéve a primitív típusokat. Az osztályok konstruktora fogja inicializálni az objektumot. A konstruktor neve meg kell egyezzen az osztály nevével. A konstruktornak nincs visszatérési értéke, de paraméterei lehetnek, amelyekkel meg lehet adni, hogy hogyan inicializáljuk az objektumot.

A new operátor:

- Szintaxis: `new Osztály(args)`
- Létrehozás lépései:
 - Lefoglalja a számára szükséges memóriát
 - Meghívja az osztály konstruktorát
 - Visszaadja az objektumra mutató referenciát

Egy osztályhoz készíthetünk több konstruktort, amelyek különböző paraméterlistával rendelkeznek.

C++-ban is hasonlóan működik a konstruktor: a konstruktor inicializálja az objektumot, azaz tölti fel az adattagjait értékekkel, több különböző paraméter listájú konstruktort lehet létrehozni egy osztályhoz, a konstruktor neve meg kell egyezzen az osztály nevével és visszaadott értéke nem lehet.

A paraméter nélküli konstruktor eljárás neve: alapértelmezett (default) konstruktor. Csak ős osztályokban kötelező, akkor ha az osztályból gyermek osztályokat szeretnének létrehozni öröklődéssel. Megvalósítható oly módon is, hogy egy nem default konstruktor minden paraméteréhez default eljárás paramétereket adunk (pl. `Osztaly(int x = 1, int y = 2)`).

Amennyiben egy gyermek osztály konstruálunk, akkor a konstruktor minden esetben meg kell hívja rekurzívan az ős osztály(ok) konstruktorait mielőtt elkezdené a saját eljárás törzsét végrehajtani. Java-ban ez impliciten megtörténik, ha az ősosztálynak van default konstruktora.

C++-ban a heapbeli objektumok létrehozása a new operátorral történik, megszüntetésük pedig a delete operátorral. A létrehozáshoz nem elegendő a memória megfelelő méretben történő lefoglalása, hanem a konstruktor eljárást is meg kell hívni. (Ezért nem lehet objektum példányt létrehozni malloc eljárással.) A new operátorral egyetlen objektum példányt vagy megadott méretű tömböt hozhatunk létre. A new operátor alkalmazásának eredménye mindig egy pointer a new operandusában megadott osztályra.

Szintaxis:

Tömbök foglalásakor a default konstruktor hívódik meg. Megszüntetésüknél az üres [] zárójelpár használata kötelező.

C++ban az objektum megszüntetése előtti takarítást, erőforrás felszabadítást a destruktork végzi. A neve meg kell egyezzen az osztály nevével, ami elé egy ~ (tilde) jelet is kell tenni. Paramétere és visszaadott értéke nem

lehet. A destruktork már nem állíthatja meg az objektum megszüntetését. Amikor a destruktork véget ér, az objektumot a rendszer a memóriából törli. Mindig a gyerek osztály destruktora hívódik meg először, és azt követi rekurzívan az ősz osztályok destruktoraik a meghívása.

Java-ban nincs szükség a heap-ben létrehozott objektumok manuális törlésére. A takarítást automatikusan elvégzi a garbage collector (szemétygyűjtő). Ez biztonságosabb, a programozónak nem kell emlékeznie, hogy fel kell szabadítani az erőforrásokat, viszont sokkal lassabb. A szemétygyűjtést kézzel is el lehet indítani, de ez nem egyenlő a destruktorkkal, kiszámíthatatlan, hogy mikor fog végrehajtódni.

Objektum másolás

Akkor beszélünk klónozásról, ha egy objektum példányt két (vagy több) példányban sokszorosítunk úgy, hogy az egyes példányok adat tagjai azonosak lesznek.

Klónozás lehetséges az „=” segítségével, viszont ilyenkor az objektumok ugyan lemásolódnak, de a referenciájuk ugyanarra a memóriaterületre fog mutatni, azaz, ha pl. az egyik másolt objektum egyik adattagját módosítjuk, az az eredeti objektumra is hatással lesz.

Java: Valódi másolást Java-ban a clone() metódussal tudunk végrehajtani. Az osztálynak, amit szeretnénk klónozzhatóvá tenni implementálnia kell a Cloneable interfészt és meg kell hívnia az ősz clone() metódusát (super.clone()).

C++: C++-ban a valós klónozás megvalósítására szolgál a copy konstruktor. A copy konstruktor paramétereinek száma 1, ennek az egy paraméternek a típusa pedig a tartalmazó osztályra mutató referencia típus.

Dinamikus, lokális és statikus objektumok létrehozása:

C++:

A statikusan létrehozott objektum az adott kód blokk végén megszűnik, amelyekben létre lett hozva.

Lokális objektumokat default paraméter vagy objektumokat tartalmazó kifejezésekben használhatunk. Szokás még objektum konstansnak is nevezni őket.

Objektumokat dinamikusan a new operátor segítségével tudunk létrehozni, amelynek törléséről a programozónak kell gondoskodnia.

Java: Java-ban minden objektum dinamikusan jön létre a heap-ben.

A statikus adattagok és metódusok

A statikus adattagok és metódusok hasonlóan működnek Java-ban és C++-ban. Mindkét nyelven a static módosítóval tudjuk jelezni, hogy az adott member statikus lesz.

Az ilyen adattagok és metódusok csak egy példányban jönnek létre és az osztálynak lesznek a tagjai, amelyet az objektumok közösen használhatnak.

A statikus metódusok nem lehetnek virtuálisak, nem hivatkozhatnak az adott objektumra (this-re).

Az ilyen adattagok, metódusok példányosítás nélkül is használhatóak.

Olyan esetekben lehetnek hasznosak, amikor az adott adattag, metódus független az objektumoktól, és mindenhol megegyezne az implementáció. Például van egy statikus adattagunk, amely a diákok számát tárolja és egy statikus metódus, amely visszaadja ennek a statikus adattagnak az értékét.

Operáció és operátor overloading

Operáció kiterjesztés

Az operáció kiterjesztés mind Java, mind C++ nyelven támogatott és hasonló módon működik. A lényege, hogy azonos nevű függvények többször vannak implementálva melyek paraméterei eltérő számúak és típusúak lehetnek. Ilyenek a változó paraméterű konstruktorok is többek között.

A fordító a kiterjesztett metódusokat a paraméterlistájuk alapján különbözteti meg

```
void sum(int a,int b){ System.out.println(a+b); }  
void sum(int a,int b,int c){ System.out.println(a+b+c); }
```

Operátor kiterjesztés

Java-ban nincs lehetőség az operátorok kiterjesztésére. A C++ programozási nyelv lehetőséget biztosít arra, hogy az osztályokra kiterjesszük a nyelvben definiált bináris és unáris operátorokat. A kiterjesztésre vonatkozóan több megszorítás is van, ennek ellenére ez a szolgáltatás jelentős lépés az absztrakció növelésének irányába.

- A kiterjesztés CSAK osztályok esetén lehetséges (ebben benne van a class, struct és a union), viszont nem működik tömbökre, pointerekre.
- Bizonyos elemi operátorok kiterjesztésére nincs lehetőség, ilyenek a . (member selection), :: (scope resolution), ? : (ternary), * (pointer to member), # és ## a preproceszorból. Kiterjeszthető viszont a (typecast) operátor!
- Az operátorok precedenciája nem változtatható meg.
- Az operátor eljárások öröklődnek (kivéve az „=” operátor).
- Az operátorok egyik operandusa osztály (vagy osztályra mutató referencia típus) kell legyen. Ettől függetlenül lehet a két operandus különböző.

A friend osztályok és eljárások

Az operátor eljárások implementációjakor szükség lehet objektumok private és protected adattagjainak elérésére. Az adattagok elérésére használhatunk segéd eljárásokat, azonban itt egy speciális esetről van szó, ahol számításba jöhet egy "barát" eljárás alkalmazása is (talán szándékaink szerint metódussal szeretnénk volna megvalósítani az operátor eljárást, csak valamiért ez nem volt megengedett).

- A friend eljárást az osztály belsejében kell deklarálni.
- Nemcsak eljárás lehet friend, hanem másik osztály is!
- A friend eljárások nem lesznek az osztály tagjai!
- Abban az osztályban kell őket deklarálni, amelynek a tagjait el kívánják érni.
- A friend eljárásokat a global scope-ban kell megvalósítani.

Kivételkezelés

A kivétel a program futása során előálló rendellenes állapot, amely közbeavatkozás nélkül a program futásának abnormális megszakadását eredményezheti. A kivételes helyzetek kezelésének elmulasztása például a hálózati a kommunikáció, vagy az adatbázis tranzakció félbeszakadásával járhat úgy, hogy mindeközben meghatározatlan állapotba kerül a rendszer. A lényeg, hogy amikor egy hiba megjelenik a programban, azaz egy kivételes esemény történik, a program normális végrehajtása megáll, és átadódik a vezérlés a kivételkezelő mechanizmusnak.

A Java kivételkezelése a C++ kivételkezelésére alapul. A kivételkezelés eszköze a try és a catch utasítás, míg a manuális kivétel kiváltásra szolgál a throw utasítás. A kivételkezelő blokk végén a finally mindenképpen lefut.

A program azon részeit, ahol a kivételek keletkezhetnek, és amiket utána kivétel kezelő részek követnek, a program védett régióinak nevezzük. A kivétel bekövetkezésekor a throwbeli kifejezés típusának megfelelő catch blokk hívódik meg, ezt a veremben visszafelé haladva keresi meg a rendszer. A verem tartalmát az adott pontig kiüríti a rendszer, végrehajtja a catch blokkot, majd a try utáni sorral folytatja a végrehajtást.

Kivétel létrehozása

Beépített kivétel osztályok mellett létrehozhatunk sajátokat is. Java: Ha akarunk, akár saját kivételeket is hozhatunk létre bármely kivétel osztály specializálásával. Ekkor egy osztályt az Exception osztályból kell származtatni és meg kell hívni az őosztály konstruktorát.

C++:

```
class MyException : public std::exception {
    std::string _msg;
public:
    MyException(const std::string& msg) : _msg(msg){}
    virtual const char* what() const noexcept override {
        return _msg.c_str();
    }
};
```

9. Java és C++ programok fordítása és futtatása. Parancssori paraméterek, fordítási opciók, nagyobb projektek fordítása. Absztrakt-, interfész- és generikus osztályok, virtuális eljárások. A virtuális eljárások megvalósítása, szerepe, használata

C++ fordítás, futtatás

Előfordítás

Első lépésben az előfordító(preprocessor) a tényleges fordítóprogram futása előtt szövegesen átalakítja a forráskódot. Az előfordító különböző szöveges változtatásokat hajt végre a forráskódon, előkészíti azt a tényleges fordításra. Feladatai:

- Header fájlok beszúrása.
- A forrásfájlban fizikailag több sorban elhelyezkedő forráskód logikailag egy sorbatörtéző csoportosítása (ha szükséges).
- A kommentek helyettesítése whitespace karakterekkel.
- Az előfordítónak a programozó által megadott feladatok végrehajtása (szimbólumokbehelyettesítése, feltételes fordítás, makrók, stb.) A leggyakoribb műveletei a szöveghelyettesítés (#define), a szöveges állomány beépítése (#include) valamint a program részeinek feltételtől függő megtartása
- Az előfeldolgozó az #include direktíva hatására az utasításban szereplő szöveges fájl tartalmát beszúrja a programunkba, a direktíva helyére.

Fordítás

Fordításkor a forrásfájlokból az első lépésben tárgymodulok (.o) keletkeznek, önmagukban nem futóképesek. Ezt követően szükség van egy szerkesztőre, ami ezeket a modulokat összeszerkeszti. Linux/Unix rendszerek esetén a fordító a gcc. Az alábbi módon tudjuk lefordítani a több forrásfájlból álló projektet: gcc -o prog main.cpp class1.cpp class2.cpp Felsoroljuk azokat a fájlokat (a felsorolás sorrendje lényegtelen), amiket le szeretnénk fordítani. Fontos a main.cpp megadása hiszen ez a program belépési pontja. A -o prog, megadásakor megadhatjuk a program nevét, ekkor prog néven hozza létre az .exe fájlt. Ha nem mondunk semmit, akkor az alapértelmezett exe fájl neve a.out lesz. Célszerű használni a -o kapcsolót. Az exe kiterjesztés csak Windows esetén van, Linux esetén csak futtatási jogú fájlt kapunk. A fordító először mindegyiket lefordítja, melyek a .o kiterjesztésű tárgymodul fájlok lesznek, majd ezek összeszerkesztésre kerülnek

Fordítási lehetőségek

- forrásfájlokból kiindulva: gcc -o prog class1.cpp class2.cpp
 - Ekkor modulonként létrejönnek a tárgymodulok .o kiterjesztéssel.
 - Amennyiben több forrásfájl van, akkor megoldható: gcc -o prog *.cpp -ként is.
- tárgymodul és forrásfájl megadásával: Amely modulok nem változtak meg, azokat felesleges újrafordítani, tehát megadhatjuk tárgymodul és forrásfájl megadásával
 - F: gcc -o prog class1.o class2.cpp
- tárgymodulkönyvtár és forrásfájl felhasználásával:
 - a tárgymodulkönyvtár kiterjesztése .a
 - Tárgymodulkönyvtárat létrehozni (archiver) (-cr : create): ar -cr liba.a a.o
 - F: gcc -o prog b.cpp liba.a
- csak tárgymodulok felhasználásával: ekkor a -c kapcsolóval csak fordítást végzünk, szerkesztést nem.
 - F: gcc -c a.cpp b.cpp: Ekkor a.o és b.o tárgymodulokat kapunk
 - Ezt követően az ld nevű (link editor) szerkesztőprogrammal kell összeszerkeszteni a modulokat.
 - F: ld -o prog a.o b.o

A gcc fordító fontosabb fordítási opciói

Szintaxis: gcc [kapcsolók] forrásfájlok

- -Ob[szint]: A gcc fordítónak a -Ob[szint] kapcsolóval tudjuk megmondani, hogy milyen optimalizálásokat alkalmazzon, a szint maximum 3 lehet (0,1,2), inline eljárások.
- -c: mint compile, lefordítja és összeállítja a forrást, linkelést nem végez.
- -o: lehetőségünk van megadni a futtatható állomány nevét, amennyiben nem adunk meg, az alapértelmezett az a.out lesz.

- -Wall: A figyelmeztetéseket írja ki.
- -g: engedélyezi a hibakeresési információk elhelyezését a programban, ami emiatt sokkal nagyobb lesz, de nyomon lehet követni a futását például a gdb programmal.

C++ parancssori paraméterek

```
int main(int argc, char* argv[])
```

A C++ programok kezdő eljárása minden esetben a main() eljárás. A main függvény első két paramétere az argc, ami egy int és az argv tömb:

- az argc a parancssorban szereplő argumentumok száma,
- az argv a string alakban tárolt argumentumok címeit tároló tömb, az első argumentum címe argv[0], a másodiké argv[1], ..., az utolsó argumentum után egy NULL pointer következik. Az argv[0] a program nevét és útvonalát tartalmazza. A paraméterek valójában az 1 indextől kezdődnek.

Java fordítás, futtatás:

Ahhoz, hogy Java programokat tudjunk futtatni, illetve fejleszteni, szükségünk lesz egy fordító- és/vagy futtatókörnyezetre, valamint egy fordítóprogramra. A kész programunk futtatásához mindösszesen a JRE (Java Runtime Environment) szükséges, ami biztosítja a Java alkalmazások futtatásának minimális követelményeit, mint például a JVM (Java Virtual Machine) Azonban a fejlesztéshez szükségünk lesz a JDK-ra (Java Development Kit) is. Ez tartalmazza a Java alkalmazások futtatásához, valamint azok készítéséhez, fordításához szükséges programozói eszközöket is (tehát a JRE-t nem kell külön letölteni, a JDK tartalmazza). A fordítás folyamata az alábbiak alapján történik:

- Először a .java kiterjesztésű fájlokat a Java-fordító (compiler) egy közbülső nyelvre fordítja
- Java bájtkódot kapunk eredményül (ez a bájtkód hordozható). A java bájtkód a számítógép számára még nem értelmezhető. (kiterjesztése .class)
- Ennek a kódnak az értelmezését és fordítását gépi kódra a JVM (Java Virtual Machine) végzi el futásidőben.

Fordítás: javac filename.java Futtatás: java filename Java fordítási opciók:

- -g: debug információk generálása
- -s <könyvtár>: a generált fájlok könyvtárának megadása
- -sourcepath : a forrásfájlok elérési útvonalát meg lehet adni
- -Werror: figyelmeztetés esetén megáll a fordítás Java parancssori paraméterek public static void main(String[] args) A main függvény paramétere az args string tömb, amely tartalmazza a parancssori paramétereket. Ezen a tömbön valamilyen ciklus segítségével végig iterálhatunk és a parancssori paramétereket tetszés szerint kezelhetjük.

Nagyobb projektek esetén szokás build fájlokat alkalmazni: ant, gradle, makefile, stb.

Virtuális eljárások

Egy virtuális eljárás címének meghatározása indirekt módon, futás közben történik. Java-ban eleve csak virtuális eljárások vannak (kivéve a final metódusokat, amelyeket nem lehet felüldefiniálni és a private metódusokat, amelyeket nem lehet örökölni)

C++-ban a virtuális függvénytábla tartja nyilván a virtuális eljárások címeit. A VFT táblázat öröklődik, feltöltéséről a konstruktor gondoskodik. A származtatott osztály konstruktora módosítja a virtuális függvénytáblát, kijavítja az őosztályból örökölt metóduscímeket. Amikor a konstruálási folyamat véget ér, a VFT táblázat minden sora értéket kap, mégpedig a ténylegesen létrehozott osztálynak megfelelő metódus címeit. A VFT táblázat sorai ezután már nem változnak meg.

- Virtuális eljárásokat a `virtual` kulcsszóval tudunk létrehozni. Az újrafelhasználás során nagy valószínűséggel módosításra kerülő eljárásokat a szülő osztályokban célszerű egyből virtuálisra megírni, mert ezzel jelentős munkát lehet megtakarítani a későbbiekben. Java: Absztrakt osztályok
- Az `abstract` kulcsszóval hozható létre.
- Egy absztrakt osztályból nem hozható létre objektum.
- Tartalmazhat absztrakt metódusokat (absztrakt metódusnak nincs implementációja, azaz törzse), illetve nem absztraktokat
- Gyerek osztályban az `abstract` metódusokat felül KELL definiálni, ha példányosítható osztályt szeretnénk
- Ha egy osztály rendelkezik legalább egy absztrakt metódussal, akkor osztálynak is absztraktnak kell lennie
- Lehetnek adattagjai

Interfész

- Az `interface` kulcsszóval lehet létrehozni
- Egy speciális absztrakt osztály
- Nincsenek sem megvalósított metódusok, sem adattagok. Csupán metódus deklarációkat tartalmaz
- Gyerekosztályban az `implements` kulcsszóval lehet implementálni

C++: Absztrakt osztályok: A törzs nélküli virtuális eljárásokat `pure virtual` eljárásoknak nevezzük (pl.: `virtual int getArea() = 0;`). A `pure virtual` eljárás egy üres (NULL) bejegyzést foglal el a VFT (Virtual Function Table) táblázatban. Ha egy osztály ilyen eljárást tartalmaz, akkor azt absztrakt osztálynak nevezzük amiatt, mert ebből az osztályból objektum példányokat létrehozni nem lehet. A gyermek osztályokban minden `pure virtual` eljárást megfelelő törzsszel kell ellátni, ezt a fordító ellenőrzi. Amíg egyetlen `pure virtual` eljárás is marad, az osztály absztrakt lesz.

Generikus osztályok

Az generikus programozás módszere a kód hatékonyságának növelése érdekében valósul meg. Az általános programozás lehetővé teszi a programozó számára, hogy általános algoritmust írjon, amely minden adattípussal működik. Nincs szükség több, különféle algoritmusok létrehozására, ha az adattípus egész szám, karakterlánc vagy karakter.

Java Lehetőség nyílt az osztályok paraméterezésére más típusokkal. Gyakorlatilag statikus polimorfizmusról van szó, egy típusparamétert adunk meg, mivel az osztály maga úgy lett megírva, hogy a lehető legáltalánosabb legyen, és ne kelljen külön `IntegerList`, `StringList`, `AllatList`, stb. osztályokat megírunk, hanem egy általános osztályt, mint sablont használunk, és a tényleges típust a kacsacsőrök között mondjuk meg. Primitív típussal nem lehet paraméterezni, az fordítási hibát okoz.

A típusparamétereket konvenció szerint egyetlen nagybetűvel szokás elnevezni, hogy egyértelműen megkülönböztethető legyen. Gyakori elnevezések:

- E : Element (tárolók használatánál)

- K : Key
- N : Number
- T : Type
- V : Value

Néha szükség lehet, hogy a típusparaméterre valamilyen megszorítást tegyünk:

- `public class NaturalNumber`
- Wildcard-ok, ismeretlen típusok:
 - `public void process(List<? extends Foo> list)`
 - minden olyan listára, ami vagy a Foo, vagy annak leszármazottaiból áll
 - `public void addNumbers(List<? super Foo> list)`
 - minden olyan listára, ami vagy a Foo, vagy annak őseiből áll

C++ C++-ban generikus osztályokat sablonok (template) segítségével tudunk létrehozni. A függvénysablonok speciális funkciók, amelyek genrikus típusokkal működhetnek. Ez lehetővé teszi számunkra, hogy létrehozzunk egy függvénysablont, amelynek funkcionalitása egynél több típushoz vagy osztályhoz igazítható anélkül, hogy megismételnénk az egyes típusok teljes kódját.

10. A programozási nyelvek csoportosítása (paradigmák), az egyes csoportokba tartozó nyelvek legfontosabb tulajdonságai

11. Szoftverfejlesztési folyamat és elemei; a folyamat különböző modelljei

Alapvető elemek

- Szoftverspecifikáció: a szoftver funkcióit és korlátait meg kell határozni
- Szoftvertesztelés és implementáció: a specifikációnak megfelelően a szoftvert elő kell állítani
- Szoftvervalidáció: a szoftvert ellenőrizni kell, hogy tényleg azt fejlesztettük ki, amit az ügyfél kíván
- Szoftverevolúció: a szoftvert úgy alakítani, hogy megfeleljen a későbbi kívánságoknak

A szoftverfolyamat modelljei

A szoftverfolyamat modellje a szoftverfolyamat absztrakt reprezentációja. Ezek a modellek egy-egy egyedi perspektívából reprezentál egy szoftverfolyamatot, de nem pontos specifikációja annak. Sokkal inkább hasznos absztrakciók, amit a szoftverfejlesztési folyamat különböző megközelítési módjainak megértéséhez használunk.

Vízesés modell

- Specifikáció: rögzítjük a termék követelményeit. Mit tudjon a szoftver, és mit nem.
- Tervezés: szétválasztódnak a szoftver- és hardverkövetelmények. Megtervezzük a rendszer architektúráját.

- Implementáció: a szoftver fejlesztése, egységtesztelése. Az egységtesztelés azt a célt szolgálja, hogy a szoftver minden egyes egysége megfelel-e a specifikációnak.
- Verifikáció: a különálló programegységes és programok integrálása és teljes rendszerként való tesztelése, hogy a rendszer megfelel-e a specifikációnak. A tesztelés után a rendszer átadható az ügyfélnek.
- Karbantartás: a szoftver életciklusának leghosszabb fázisa. A karbantartásba beletartozik olyan hibák javítása is, amelyek nem merültek fel az életciklus korábbi szakaszaiban, illetve a szolgáltatások továbbfejlesztése.

A fázisok eredménye egy vagy több dokumentum, amelyek jóváhagyása megtörtént. A következő fázis nem indulhat, amíg az előző be nem fejeződött.

Probléma: a folyamat korai szakaszaiban állást kell foglalnunk és el kell köteleznünk magunkat, és nehéz az ügyfélhez történő alkalmazkodás. Akkor jó, ha előre ismerjük a követelményeket. Nagyobb rendszerek kisebb folyamatainál használják főleg.

Evolúciós fejlesztés

Az evolúciós fejlesztés lényege, hogy ki kell fejleszteni egy korai implementációt, azt a felhasználókkal véleményeztetni, és finomítani a felhasználói visszajelzések alapján, amíg megfelelő rendszert el nem élünk.

Két különböző típusa ismert:

- Feltáró fejlesztés: a folyamat célja az hogy a megrendelővel együtt feltárjuk a követelményeket, és kialakítsuk a végleges rendszert. A végleges rendszer úgy alakul ki, hogy egyre több, az ügyfél által kért tulajdonságot társítunk a már meglévőkhöz.
- Eldobható prototípus fejlesztése: ekkor az evolúciós fejlesztés célja, hogy minél jobban megértsük az ügyfél követelményeit, és azokra alapozva a legpontosabban fejlesszük le a terméket.

Az evolúciós fejlesztés jobb, mint a vízesés modell, ha a lehető legpontosabban szeretnénk az ügyfél kívánságainak megfelelő szoftvert fejleszteni. Előnye, hogy a specifikáció inkrementálisan fejleszthető.

A vezetőség és a tervezők szempontjából két probléma merülhet fel:

- A folyamat nem látható. A menedzsereknek rendszeresen leszállítható eredményekre van szükségük, hogy mérhessék a fejlődést.
- A rendszerek sokszor szegényesen struktúráltak. A folyamatos változtatások rontják a szoftver struktúráját.

A várhatóan rövid élettartamú kis vagy közepes rendszerek esetén az evolúciós megközelítési mód a legcélravezetőbb.

Iterációs, inkrementális modell

- Folyamat iterációja elkerülhetetlen
- ha a követelmények változnak, akkor a folyamat bizonyos részeit is változtatni kell
- ennél a modellnél minimális a specifikáció, fejlesztésben sok iteráció van, és menet közben alakul ki a végleges specifikáció
- Inkrementalitás: részfunkciókkal már működő rendszert fejlesztünk, amit minden iterációban (inkrementálisan) javítunk

- Nagy körvonalakban specifikáljuk a rendszert
 - „Inkremensek” meghatározása
 - Funkcionalitásokhoz prioritásokat rendelünk
 - Magasabbakat előbb kell biztosítani
- Architektúrát meg kell határozni
- További inkremensek pontos specifikálása menet közben történik
- Egyes inkremensek kifejlesztése történhet akár különböző folyamatokkal is - Vízésés vagy evolúciós, amelyik jobb
- Az elkészült inkremenseket akár szolgálatba is lehet állítani
- Ha határidő csúszás van kilátásban a teljes projekt nem lesz kudarcra ítélve, esetleg csak egyes inkremensek
- Megfelelő méretű inkremensek meghatározása nem triviális feladat
 - Ha túl kicsi: nem működőképes
 - Ha túl nagy: elveszítjük a modell lényegét

Bizonyos esetekben számos alapvető funkcionálisitást kell megvalósítani. Egész addig nincs működő inkremens

eXtreme Programming (XP)

- Szélsőséges inkrementális modell
- Nagyon kis funkcionálisú inkremensek
- Megrendelő intenzív részvétele fontos
- Programozás csoportos tevékenység - többen ülnek a képernyő előtt
- Sok támadója van

RAD (Rapid Application Development)

- Extrém rövid életciklus
- Működő rendszer 60-90 nap alatt
- Vízésés modell „nagysebességű” adaptálása
- Párhuzamos fejlesztés
- Komponens alapú fejlesztés
- Fázisai:
 - Üzleti modellezés
 - Milyen információk áramlanak funkciók között
 - Adatmodellezés
 - Finomítás adatszerkezetekre
 - Adatfolyam processzus
 - Adatmodell megvalósítása
 - Alkalmazás generálás
 - 4GT alkalmazása, automatikus generálás, komponensek
 - Tesztelés
 - Csak komponens tesztelés
- Nagy emberi erőforrásigény
- Fejlesztők és megrendelők intenzív együttműködése szükséges
- Nem minden típusú fejlesztésnél alkalmazható

Spirális modell

- Olyan evolúciós modell, amely kombinálja a prototípus modellt a vízésés modellel
- Inkrementális modellhez hasonló, csak általánosabb megfogalmazásban
- Nincsenek rögzített fázisok
- Más modelleket ölelhet fel
 - Prototípuskészítés pontatlan követelmények esetén
 - Vízésés modell egy későbbi körben
 - Kritikus részek esetén formális módszerek
- A spirál körei a folyamat egy-egy fázisát reprezentálják
- Minden körben a kimenet egy „release” (modell vagy szoftver)
- Körök céljai pl.:
 - Megvalósíthatóság (elvi prototípusok)
 - Követelmények meghatározása (prototípusok)
 - Tervezés (modellek és inkremensek)
 - Stb. (javítás, karbantartás, stb.)
- A körök 3-6 darab szektorokra oszthatók

WINWIN spirális modell

- WINWIN = mindenki nyer
- Megrendelő és fejlesztő is
- Sok tárgyalás kell a két fél között
- WINWIN modell számos tárgyalási szempontot visz bele a spirális modellbe
 - Egyes (al)rendszerek kulcsszereplői, érdekeltek
 - Az érdekeltek nyerő feltételei
 - Tárgyalás, kompromisszumok

12. Projektmenedzsment. Költségbecslés, szoftvermérés

Projektmenedzsment

Összetevői:

- Az emberek menedzselése
- Minőség-ellenőrzés és -biztosítás
- Folyamat továbbfejlesztése
- Konfiguráció kezelés
- Rendszer építés
- Hibamenedzsment

Projekt sikertelenségének okai

- A szükséges ráfordítások alulbecslése
- Technikai nehézségek
- A projekt csapatban nem megfelelő a kommunikáció
- A projekt menedzsment hibái

Az Emberek menedzselése

Szoftverfejlesztő szervezet legnagyobb vagyona az emberek Sok projekt bukásának legfőbb oka a rossz humánmenedzsment Hatékony együttműködés fontos - Csapatszellemet kell kialakítani Fontos a kommunikáció Az emberek kiválasztása különböző tesztekkel történhet:

- Programozási képesség
- Pszichometrikus tesztek

Minőség-ellenőrzés és –biztosítás

Mindenki célja: termék vagy szolgáltatás minőségének magas szinten tartása A termék feleljen meg a specifikációnak Fejlesztőnek is lehetnek (belső) igényei, pl. karbantarthatóság Egyes jellemzőket nem könnyű specifikálni , pl. karbantarthatóság

Folyamat továbbfejlesztése

CMM(I) (Capability Maturity Model (Integration)): a szoftver folyamat mérése

- Cél: a szoftverfejlesztési folyamat hatékonyságának mérése
- Egy szervezet megkaphatja valamely szintű minősítését
- 5 besorolási szint
 - 1. Kezdeti: csak néhány folyamat definiált, a többségük esetleges
 - 2. Reprodukálható: az alapvető projekt menedzsment folyamatok definiáltak. Költség ütemezés, funkcionalitás kezelése
 - 3. Definiált: a menedzsment és a fejlesztés folyamatai is dokumentáltak és szabványosítottak az egész szervezetre.
 - 4. Ellenőrzött: a szoftver folyamat és termék minőségének részletes mérése, ellenőrzése.
 - 5. Optimalizált: a folyamatok folytonos javítása az új technológiák ellenőrzött bevezetésével

A szoftver folyamat javítása - Az alapvető cél a minőség és a hatékonyság növelése

Használjunk metrikákat

Hiba analízis

- Hibák eredetének kategorizálása
- Hibák javítási költségei

Konfiguráció kezelés

A rendszer változásainak kezelése

Változások felügyelt módon történjenek

Fejlesztés, evolúció, karbantartás miatt van rá szükség

Minőségkezelés része

Szoftver változatok

- Verziók (version, revision)

- Kiadások (release)
- Alapvonal (baseline, mainline, trunk)
- Fejlesztési ágak (branch)

Konfigurációs adatbázis - mindent tárol:

- Forráskód, (bináris kód), dokumentumok
- Építési folyamat, szkriptek
- Hiba adatbázis
- Változtatások története
- Verziók

Rendszer építés

Komponensek fordítása és szerkesztése

Komponensek (és fájlok) között építési függőségek vannak

Nagy rendszernél bonyolult a folyamat - Hosszadalmas is, ezért inkrementálisan kell végezni

Automatizálni kell: építési szkriptek: configure, make

Eszközkiválasztás (fordítóprogram), beállítások

Hibamenedzsment

Hibák követése fontos

Fontos, mert sok hiba van/lesz: kategorizálás, prioritások felállítása, követés elengedhetetlen

Milyen jellegű a hiba - (hibabejelentés, új feature, ...)

Hibák követésére hibaadatbázis

- Minden hibának egyedi azonosítója van
- Bejelentő neve
- Kijelölt felelős személy, megfigyelők listája
- Dátum
- Rövid összegzés
- Súlyosság: pl. triviális, kicsi, nagy
- Platform, operációs rendszer
- Termék, komponens, verziószám
- Függőségek más hibákkal
- Fontos a hiba életútjának rögzítése

Szoftverkötség becslése

Projekt tevékenységeinek kapcsolódása a munka-, idő- és pénzköltségekhez Becsléseket lehet és kell adni

Projekt összköltsége:

- Hardver és szoftver költség karbantartással

- Utazási és képzési költség
- Munkaköltség

Ezeket meg kell becsülni:

- Mennyi pénz?
- Mennyi ráfordítás?
- Mennyi idő?

Munkaköltség:

- Legjelentősebb
- Fejlesztők fizetése
- Kisegítő személyzet fizetése
- Bérleti díj, rezsi
- Infrastruktúra használat (pl. hálózat)
- Járulékok, adók

Szoftvermérés

Szoftvermérés: termék vagy folyamat valamely jellemzőjét numerikusan kifejezni (metrika). Ezen értékekből következtetések vonhatók le a minőségre vonatkozóan.

Két csoport:

- Vezérlési metrikák. Folyamattal kapcsolatosak, pl. egy hiba javításához szükséges átlagos idő(folyamat és projekt metrikák)
- Prediktor metrikák. Termékkel kapcsolatosak, pl. LOC, ciklomatikus komplexitás, osztály metódusainak száma (termék metrikák)
 - LOC = Lines Of Code
 - Több technika: Csak nem üres sorok, Csak végrehajtható sorok
 - Félrevezető lehet - Nem összehasonlítható programozási nyelvek (assembly, magas szintű nyelv)
- Mérési folyamat:
 - Alkalmazandó mérések kiválasztása
 - Mérni kívánt komponensek kiválasztása
 - Mérés (metrika számítás)
- Termék metrikák
 - Dinamikus
 - Szorosabb kapcsolat egyes minőségi jellemzőkkel
 - (pl. teljesítmény, hibák száma)
 - Statikus
 - Közvetett kapcsolat
 - Számtalan konkrét metrikát ajánlottak már
 - Kritikus kérdés: hogyan következtetünk a minőségi jellemzőkre a sok számból?
 - Fajták:
 - Méret
 - Komplexitás, csatolás, kohézió
 - Objektumorientáltsággal kapcsolatos metrikák

- Méret alapú metrikák (folyt.)
 - Széleskörűen használják ezeket a metrikákat, de nagyon sok vita van alkalmazásokról
 - Hibák / KLOC
 - Defekt / KLOC
 - Költség / LOC
 - Dokumentációs oldalak / KLOC
 - Hibák / emberhónap
 - LOC / emberhónap
 - Költség / dokumentációs oldal
- Funkció alapú metrikák
 - Felhasználói inputok száma - alkalmazáshoz szükséges adatok
 - Felhasználói outputok számar riportok, képernyők, hibaüzenetek
 - Felhasználói kérdések száma - on-line input és output
 - Fájlok száma- adatok logikai csoportja
- 3D mérték
 - Számítás: $Index = I + O + Q + F + E + T + R$
 - I=input
 - O=output
 - Q=lekérdezés
 - F=fájlok
 - E=külső interfész
 - T=transzformáció
 - R=átmenetek
- Minőség mérése
 - Integritás: külső támadások elleni védelem
 - Fenyegetettség: annak valószínűsége, hogy egy adott típusú támadás bekövetkezik egy adott időszakban
 - Biztonság: annak valószínűsége, hogy egy adott típusú támadást visszaver a rendszer
 - Integritás = $\sum [1 - (\text{fenyegetettség} \times (1 - \text{biztonság}))]$ (Összegzés a különböző támadás típusokra történik)
 - DRE (defect removal efficiency)
 - $DRE = E / (E + D)$, ahol E olyan hibák száma, amelyeket még az átadás előtt felfedezünk, D pedig az átadás után a felhasználó által észlelt hiányosságok száma

15. Neumann-elvű gép egységei. CPU, adatút, utasítás-végrehajtás, utasítás- és processzorszintű párhuzamosság. Korszerű számítógépek tervezési elvei. Példák RISC (UltraSPARC) és CISC (Pentium 4) architektúrákra, jellemzőik

Számítógép architektúra: A hardver egy általános absztrakciója: a hardver struktúráját és viselkedését jelenti más rendszerek egyedi, sajátos tulajdonságaitól eltekintve

Neumann elvű gép

- Neumann-architektúra mára a tárolt programú számítógép fogalmává vált
- Számítógép működését tárolt program vezérli (Turing).
- A vezérlést vezérlés-folyam (control-flow) segítségével lehet leírni
- Az aritmetikai és logikai műveletek (programutasítások) végrehajtását önálló részegység (ALU) végzi
- 2-es (bináris) számrendszer alkalmazása
- Öt funkcionális egység (aritmetikai egység, központi vezérlőegység, memóriák, bemeneti és kimeneti egységek)

Neumann-elvű gép egységei

- központi memória: a program kódját és adatait tárolja számokként
- központi feldolgozóegység (CPU): a központi memóriában tárolt program utasításait beolvassa és végrehajtja
- külső sín: a részegységeket köti össze, adatokat, címeket, vezérlőjeleket továbbít
- belső sín: CPU részegységei közötti kommunikációt hozza létre (vezérlőegység-ALU-regiszterek)
- beviteli/kiviteli eszközök: kapcsolatot teremtet a felhasználóval, adatot tárol a háttértáron, nyomtat, stb.
- működést biztosító járulékos eszközök: például gépház, tápellátás, hűtés...

CPU, adatút, utasítás-végrehajtás, utasítás- és processzorszintű párhuzamosság

CPU

A CPU feladata a központi memóriában tárolt program utasításainak beolvasása és végrehajtása 3 fő egysége:

- vezérlőegység (CU):
 - Utasítások beolvasása a memóriából
 - az ALU és regiszterek vezérlése
- aritmetika-logikai egység (ALU):
 - Egy tipikus Neumann-féle CPU belső szerkezetének részében az ALU végzi az összeadást, a kivonást és más egyszerű műveleteket az inputjain, így adva át az eredményt az output regiszternek, azaz a kimeneten ez fog megjelenni.
 - az utasítások végrehajtásához szükséges aritmetikai és logikai műveleteket végzi el
 - Aritmetikai operátorok: +, -, *, / (alapműveletek)
 - Logikai operátorok: NOT, AND, OR, NAND, NOR, XOR, NXOR (EQ)
- regiszterek:
 - kisméretű, gyors memóriarekeszek, amelyek részeredményeket és vezérlőinformációkat tárolnak
 - A regiszterek a számítógépek központi feldolgozó egységeinek, illetve mikroprocesszorainak gyorsan írható-olvasható, ideiglenes tartalmú, és általában egyszerre csak 1 gépi szó feldolgozására alkalmas tárolóegységei
- adatút

Adatút

- Az adatút az adatok áramlásának útja, alapfeladata, hogy kiválasszon egy vagy két regisztert, az ALU-val műveletet végeztessen el rajtuk (összeadás, kivonás...), az eredményt pedig valamelyik regiszterben tárolja. Egyes gépeken az adatút működését mikroprogram vezérli, másutt a vezérlés közvetlenül a hardver feladata.

- Folyamata:
 - A regiszter készletből feltöltődik az ALU két bemenő regisztere (A és B)
 - Az eredmény az ALU kimenő regiszterébe kerül
 - Az ALU kimenő regiszteréből a kijelölt regiszterbe kerül az eredmény
- Két operandusnak az ALU-n történő átfutásából és az eredmény regiszterbe tárolásából álló folyamatot adatútciklusnak nevezzük.

Utasítás-végrehajtás

A mikroprocesszor 1-1 utasítása úgynevezett gépi ciklusok egymásutániságából áll, vagyis 1 utasítás egy vagy több gépi ciklusból tevődik össze.

A CPU minden utasítást apró lépések sorozataként hajt végre. Ezek a lépések a következők:

1. A soron következő utasítás beolvasása a memóriából az utasításregiszterbe.
2. Az utasításslámláló beállítása a következő utasítás címére.
3. A beolvasott utasítás típusának meghatározása.
4. Ha az utasítás memóriabeli szót használ, a szó helyének megállapítása.
5. Ha szükséges, a szó beolvasása a CPU egy regiszterébe.
6. Az utasítás végrehajtása.
7. Vissza az 1. pontra, a következő utasítás végrehajtásának megkezdése.

Ezt a lépéssorozatot gyakran nevezik betöltő-dekódoló-végrehajtó ciklusnak, és központi szerepet tölt be minden számítógép működésében.

Nagy probléma a számítógépeknél, hogy a memória olvasása lassú, ezért az utasítás és az adatok beolvasása közben a CPU több része kihasználatlan. A gyorsítás egyik módja a lapkák gyorsítása az órajel frekvenciájának növelésével, de ez korlátozott. Emiatt a legtöbb tervező a párhuzamosság kiaknázásában lát lehetőséget.

A párhuzamosság két féleképpen lehet jelen: utasításszintű párhuzamosság vagy processzorszintű párhuzamosság formájában.

Utasításszintű párhuzamosság

Az utasítások végrehajtásának gyorsítása érdekében előre be lehet olvasni az utasításokat, hogy azok rendelkezésre álljanak, amikor szükség van rájuk. Ezeket az utasításokat egy előolvasási puffer (prefetch buffer) elnevezésű regiszterkészlet tárolja. Ilyen módon a soron következő utasítást általában az előolvasási pufferból lehet venni ahelyett, hogy egy egy memóriaolvasás befejeződésére kellene várni.

Csővezeték:

Lényegében az előolvasás az utasítás végrehajtását két részre osztja: beolvasás és tulajdonképpeni végrehajtás. A csővezeték ezt a stratégiát viszi sokkal tovább. Az utasítás végrehajtását kettő helyett több részre osztja, minden részt külön hardverelem kezel, amelyek mind egyszerre működhetnek.

A csővezeték lehetővé teszi, hogy kompromisszumot kössünk késleltetés (mennyi ideig tart egy utasítás végrehajtása) és áteresztőképesség (hány MIPS a processzor sebessége) között.

Párhuzamos csővezeték:

Az előolvasó egység két utasítást olvas be egyszerre, majd ezeket az egyik, illetve a másik csővezetékre teszi. A csővezetéknek saját ALU-juk van, így párhuzamosan tudnak működni, feltéve, hogy a két utasítás nem használja ugyanazt az erőforrást, és egyik sem használja fel a másik eredményét. Ugyanúgy, mint egyetlen csővezeték esetén, a feltételek betartását vagy a fordítóprogramnak kell garantálnia, vagy a konfliktusokat egy kiegészítő hardvernek kell a végrehajtás során felismernie és kiküszöbölnie.

Szuperskaláris architektúra:

Itt egy csővezetékot használnak, de több funkcionális egységgel. Ezek olyan processzorok, amelyek több – gyakran négy vagy hat – utasítás végrehajtását kezdik el egyetlen órajel alatt. Természetesen egy szuperskaláris CPU-nak több funkcionális egységének kell lennie, amelyek kezelik mindezeket az utasításokat. Az utasítások megkezdését sokkal nagyobb ütemben végzik, mint amilyen ütemben azokat végre lehet hajtani, így a terhelés megoszlik a funkcionális egységek között. A szuperskaláris processzor elvében implicit módon benne van az a feltételezés, hogy a megfelelő fázis lényegesen gyorsabban tudja előkészíteni az utasításokat, mint ahogy a rákövetkező fázis képes azokat végrehajtani. Ez a fázis funkcionális egységeinek többsége egy órajelnél jóval több időt igényel feladata elvégzéséhez – a memóriához fordulók vagy a lebegőpontos műveleteket végzők biztosan. Akár több ALU-t is tartalmazhat.

Processzorszintű párhuzamosság

Tömb processzorok:

Egy tömbprocesszor nagyszámú egyforma processzorból áll, ugyanazon műveleteket egyszerre végzik különböző adathalmazokon. A feladatok szabályossága és szerkezete különösen megfelelővé teszi ezeket párhuzamos feldolgozásra. Olyan utasításokat hajthatnak végre, mint amilyen például két vektor elemeinek páronkénti összeadása.

Multiprocesszorok:

Ezekben több teljes CPU van, amelyek egy közös memóriát használnak. Amikor két vagy több CPU rendelkezik azzal a képességgel, hogy szorosan együttműködjenek, akkor azokat szorosan kapcsoltaknak nevezik. A legegyszerűbb, ha egyetlen sín van, amelyhez csatlakoztatjuk a memóriát és az összes processzort. Ha sok gyors processzor próbálja állandóan elérni a memóriát a közös sínen keresztül, az konfliktusokhoz vezet. Az egyik megoldás, hogy minden processzornak biztosítunk valamekkora saját lokális memóriát, amelyet a többiek nem érhetnek el. Így csökken a közös sín forgalma. Jellemzően maximum pár száz CPU-t építenek össze.

Multiszámítógépek:

Nehéz sok processzort és memóriát összekötni. Ezért gyakran sok összekapcsolt számítógépből álló rendszereket építenek, amelyeknek csak saját memóriájuk van. A multiszámítógépek CPU-it lazán kapcsoltaknak nevezik. A multiszámítógép processzorai üzenetek küldésével kommunikálnak egymással. Nagy rendszerekben nem célszerű minden számítógépet minden másikkal összekötni, ezért 2 és 3 dimenziós rácsot, fákat és gyűrűket használnak. Ennek következtében egy gép valamelyik másikkal küldött üzeneteinek gyakran egy vagy több közbeeső gépen vagy csomóponton kell áthaladniuk ahhoz, hogy a kiindulási helyükről elérjenek a céljukhoz. Néhány mikroszekundumos nagyságrendű üzenetküldési idők nagyobb nehézség nélkül elérhetők. 10 000 processzort tartalmazó multiszámítógépeket is építettek már.

Korszerű számítógépek tervezési elvei

- Minden utasítást közvetlenül a hardver hajtson végre.
 - Ezek nem bonthatók fel interpretált mikROUTASÍTÁSOKRA. Az interpretációs szint kiküszöbölésével a legtöbb utasítás gyors lesz.
- Maximalizálni kell az utasítások kiadásának ütemét
 - Megpróbálják egy másodperc alatt a lehető legtöbb utasítás végrehajtását elkezdni, tehát a párhuzamosságra kell törekedni.
- Az utasítások könnyen dekódolhatók legyenek.
 - Az utasítások szabályosak, egyforma hosszúak legyenek, és kevés mezőből álljanak.
- Csak a betöltő és tároló utasítások hivatkozzanak a memóriára
 - A memóriaműveletek sok időt vehetnek igénybe, legjobb más utasításokkal átfedve végrehajtani, ha semmi mást nem tesznek, csak adatokat mozgatnak a regiszterek és a memória között, minden más utasítás csak regisztereket használhat.
- Sok regiszter legyen.
 - Mivel a memóriaműveletek lassúak, sok regiszterre van szükségünk, hogy egy beolvasott szó mindig a regiszterben maradjon, amíg szükség van rá.

RISC Reduced Instruction Set Computer – Csökkentett utasításkészletű számítógép

A mikroprocesszorok létrejöttét követően két irányzat alakult ki. – RISC, CISC. Azt a szempontot tartották szem előtt, hogy a processzor kevés alapvető utasítást tudjon végrehajtani, de azokat rendkívül gyorsan (jellemzően 1 órajelciklus alatt). Ezek a RISC (Reduced Instruction Set Computer - redukált utasításkészletű) processzorok. Itt az összetettebb funkciókat több utasítás kombinációjával lehet megvalósítani. A RISC mikroprocesszorokba számos belső regiszter kerül integrálásra, ezáltal is csökkentve a memóriához való fordulás gyakoriságát és gyorsítva a működésként. Ugyancsak sajátja ezen processzoroknak a - később ismertetett - ún. pipeline architektúra. Ennek lényege az, hogy a műveleteket részműveletekké bontják szét, és e részműveleteket időben párhuzamosítják. A RISC processzorok az utolsó 10 évben - első sorban a nagyobb teljesítményt igénylő rendszereknél (pl. munkaállomások) nyertek teret. Nagyon kevés utasítással rendelkeznek, tipikusan 50 körül. Az adatút egyszeri bejárásával végrehajthatók ezek az utasítások, tehát egy órajel alatt. Nem használ mikroprogram interpretálást, ezért sokkal gyorsabb, mint a CISC.

Példa: IBM 801, UltraSPARC, ARM

CISC Complex Instruction Set Computer – Összetett utasításkészletű számítógép

Azok a processzorok tartoznak ide, amelyek utasításkészlete lehetőleg minden programozói igényt ki próbál elégíteni, vagyis komplex utasításkészletet alkot. Ezeket nevezzük CISC (Complex Instruction Set Computer = komplex utasításkészletű számítógép) processzoroknak. Markáns elemei az Intel processzorok. A CISC törekvésnek az egyik mozgatórugója, hogy megpróbálják a magasabb szintű nyelvek lehetőségeit közelíteni, vagyis, hogy a programozás "munkaigényes" alacsony szintjét, gépközelit így is ellensúlyozzák. Interpretálást használ, ezért sokkal összetettebb utasításai vannak, mint egy RISC gépnek. Több száz ilyen utasítása lehet. Az interpretálás miatt lassabb a végrehajtás.

Példa: x86 architektúrák pl. Intel 80x86 család.

16. Számítógép perifériák: Mágneses és optikai adattárolás alapelvei, működésük (merevlemez, Audio CD, CD-ROM, CD-R, CD-RW, DVD, Bluray). SCSI, RAID. Nyomtatók, egér, billentyűzet. Telekommunikációs berendezések (modem, ADSL, KábelTV-s internet)

Számítógép perifériák

A számítógéphez különböző perifériák kapcsolhatók, melyek segítségével a felhasználók kommunikálni tudnak a gazdagéppel. Ezek egy része beviteli, vagy kiviteli eszköz, - amely az adatok bevitelére, vagy kiírására szolgál. A háttértárolók feladata az adatok és programok hosszabb ideig tartó tárolása. Tartalmuk a számítógép kikapcsolása után is megmarad. A fogalmat általában azokra az eszközökre alkalmazzák, melyek külsőleg csatlakoznak a gazdagéphez, tipikusan egy számítógépes buszon keresztül, mint például az USB.

Csoportosításuk:

- bemeneti perifériák
- kimeneti perifériák

Mágneses adattárolás alapelvei, működése

Egy mágneslemez egy vagy több mágnesezhető bevonattal ellátott alumíniumkorongból áll. Egy indukciós fej lebeg a lemez felszíne felett egy vékony légpárnán. Ha pozitív vagy negatív áram folyik az indukciós tekercsben, a fej alatt a lemez magnetizálódik, és ahogy a korong forog a fej alatt, így bitsorozatot lehet felírni. Amikor a fej egy mágnesezett terület felett halad át, akkor pozitív vagy negatív áram indukálódik benne, így a korábban eltárolt biteket vissza lehet olvasni. Egy teljes körülfordulás alatt felírt bitsorozat a sáv. Minden sáv rögzített méretű, tipikusan 512 bájt méretű szektorokra van osztva, melyeket egy fejléc előz meg, lehetővé téve a fej szinkronizálását írás és olvasás előtt. Az adatok után hibajavító kód helyezkedik el (Hamming vagy Reed-Solomon).

Minden lemeznek vannak mozgatható karjai, melyek a forgástengelytől sugárirányban ki-be tudnak mozogni. Minden sugárirányú pozíción egy-egy sáv írható fel. Tehát a sávok forgástengely középpontú koncentrikus körök.

Egy lemezegység több, egymás felett elhelyezett korongból áll. Minden felülethez tartozik egy fej és egy mozgatókar. A karok rögzítve vannak egymáshoz, így a fejek mindig ugyanarra a sugárirányú pozícióra állnak be. Egy adott sugárirányú pozícióhoz tartozó sávok összességét cilindereknek nevezzük. Általában 6-12 korong található egymás felett.

Egy szektor beolvasásához vagy kiírásához először a fejet a megfelelő sugárirányú pozícióba kell állítani, ezt keresésnek (seek) hívják. A fej kívánt sugárirányú pozícióba való beállása után van egy kis szünet, az ún. forgási késleltetés, amíg a keresett szektor a fej alá fordul. A külső sávok hosszabbak, mint a belsők, a lemezek pedig a fejek pozíciójától függetlenül állandó szögsebességgel forognak, ezért ez problémát vet fel. Megoldásként a cilindereket zónákba osztják, és a külső zónákban több szektort tesznek egy sávba. Minden szektor mérete egyforma. Minden lemezhez tartozik egy lemezvezérlő, egy lapka, amely vezérli a meghajtót.

Optikai adattárolás alapelvei, működése

Az optikai adattárolók megjelenése kör alakú lemez, amelyek felületén helyezkedik el az adattárolásra alkalmas réteg. A lemezek írása és olvasása a nevükből adódóan optikai eljárással történik. Az optikai írás és az olvasás lézersugárral történik a lemez forgatása közben. A lemezen történő adatrögzítéskor a lézersugár apró mélyedéseket hoz létre spirál alakú vonalban, így tárolva a digitális adatot; az adat kiolvasásához ugyanilyen hullámhosszú lézersugár halad végig a mélyedések sorozatán és olvassa vissza a digitális adatot aszerint, hogy a sugár visszatükröződik, vagy szétszóródik a lemez felületéről. Az optikai tárolókat több tulajdonságuk is jelentősen megkülönbözteti a mágneses tárolótól: az optikai tárolás elméletben sokkal nagyobb adatsűrűséget enged meg, mivel a fény sokkal kisebb területre fókuszálható, mint a mágneses adattárolókban az elemi mágnesezhető részecskék mérete. Továbbá, a megfelelő minőségű és megfelelően kezelt optikai lemezek élettartama évszázadokban mérhető, ezenkívül nem érzékenyek az elektromágneses behatásokra sem.

A felületen elhelyezkedő mélyedéseket üregnek (pit), az üregek közötti érintetlen területeket pedig szintnek (land) hívják.

Az tűnik a legegyszerűbbnek, hogy üreget használjunk a 0, szintet az 1 tárolásához, ennél azonban megbízhatóbb, ha az üreg/szint vagy a szint/üreg átmenetet használjuk az 1-hez, az átmenet hiányát pedig a 0-hoz, ezért ez utóbbi módszert alkalmazzák.

Merevlemez (HDD)

- Mágneses adattároló
- Tárolókapacitás: 500 GB – 12 TB
- Írása és olvasási sebesség: függ a forgási sebességtől, ez jellemzően 5400, 7200, 1000 vagy 15000 fordulat/perc, és az adatsűrűségtől (egy adathordozó fizikai felületével arányos tárolókapacitása)

Audio CD

- A jel sűrűsége állandó a spirál mentén
- 74 percnyi anyag fér rá (Beethoven IX. szimfóniája kiadható legyen)
- Állandó kerületi sebesség, ehhez szükséges a változó forgási sebesség (120 cm/mp)
- Nincs hibajavítás, mivel nem gond, ha néhány bit elvész az audio anyagból

CD-ROM

- Univerzális adathordozó, illetve médialemez.
- Csak olvasható (véglegesített) adathordozó.
- Népszerűen használták szoftverek és adatok terjesztésére
- Az ilyen típusú lemezeket kereskedelmi forgalomban hozzák létre, és létrehozásuk után nem menthet rájuk adatokat.
- 650 MB tárolható

CD-R

- Író berendezéssel rögzíthető az adat (1x)
- Újdonság: ◦ Író lézernyaláb ◦ Alumínium helyett arany felület ◦ Üregek és szintek helyett festékréteg alkalmazása: Kezdetben átlátszó a festékréteg (cianin (zöld) vagy ftalocianin (sárgás))
- 700 MB tárolható

CD-RW

- Újraírható optikai lemez
- A CD-RW lemez adatait számos alkalommal törölhetjük és rögzíthetjük.
- Újdonság: ◦ Más adattároló réteg: ▪ Ezüst, indium, antimon és tellúr ötvözet ▪ Kétféle stabil állapot: kristályos és amorf (más fényvisszaverő képesség)
- 3 eltérő energiájú lézer: ▪ Legmagasabb energia: megolvad az ötvözet → amorf ▪ Közepes energia: megolvad → kristályos állapot ▪ Alacsony energia: anyag állapotnak érzékelése, de meg nem változik

DVD

- Nagy kapacitású optikai tároló, amely leginkább mozgókép és jó minőségű hang, valamint adat tárolására használatos
- Méreteit tekintve általában akkora, mint a CD, vagyis 120 mm átmérőjű.
- Létezik egyrétegű/kétretegű illetve egyoldalas/kétoldalas lemez (4,5 GB – 17 GB)
- Nagyobb jelsűrűség, mert ◦ Kisebbek az üregek (0,4 µm (CD: 0,8 µm)) ◦ Szorosabb spirálok ◦ Vörös lézert használtak

Blu-Ray

- A DVD technológia továbbfejlesztése, a Blu-Ray disc
- Kék lézer használata írásra és olvasásra a vörös helyett ◦ Rövidebb hullámhossz, jobban fókuszálható, kisebb mélyedések
- 25 GB (egyoldalas) és 50 GB (kétoldalas) adattárolási képesség

SCSI, RAID

SCSI

Az SCSI-lemezek nem különböznek az IDE-lemezektől abban a tekintetben, hogy ezek is cilinderekre, sávokra és szektorokra vannak osztva, de más az interfészük, és sokkal nagyobb az adatátviteli sebességük. Az 5 MHz-estől a 160 MHz-ig nagyon sok változatot kifejlesztettek.

A SCSI több egy merevlemez-interfésznel. Ez egy sín, amelyre egy SCSI-vezérlő és legfeljebb hét eszköz csatlakoztatható. Ezek között lehet egy vagy több SCSI-merevlemez, CD-ROM, CD-író, szkennel, szalagegység és más SCSI-periféria.

A SCSI-vezérlők és –perifériák kezdeményező és fogadó üzemmódban működhetnek. Általában a kezdeményezőként működő vezérlő adja ki a parancsokat a fogadóként viselkedő lemezegységeknek és egyéb perifériáknak.

A szabvány megengedi, hogy az összes eszköz egyszerre működjön, így nagyban növelhető a hatékonyság több folyamatot futtató környezetben.

RAID

A RAID tárolási technológia, mely segítségével az adatok elosztása vagy replikálása több fizikailag független merevlemezen, egy logikai lemez létrehozásával lehetséges. Minden RAID szint alapjában véve vagy az adatbiztonság növelését vagy az adatátviteli sebesség növelését szolgálja.

Azon túl, hogy a RAID szoftverszempontról egyetlen lemeznek látszik, az adatok szét vannak osztva a meghajtók között, lehetővé téve a párhuzamos működést.

A RAID alapötlete a lemezegységek csíkokra (stripes) bontása. Ezek a csíkok azonban nem azonosak a lemez fizikai sávjával.

RAID-0 (összefűzés vagy csíkozás)

Lemezek egyszerű összefűzését jelenti, viszont semmilyen redundanciát nem ad, így nem biztosít hibatűrést, azaz egyetlen meghajtó meghibásodása az egész tömb hibáját okozza. Mind az írási, mind az olvasási műveletek párhuzamosítva történnek, ideális esetben a sebesség az egyes lemezek sebességének összege lesz, így a módszer a RAID szintek közül a legjobb teljesítményt nyújtja (a többi módszernél a redundancia kezelése lassítja a rendszert)

RAID-1 (tükrözés)

A RAID 1 eljárás alapja az adatok tükrözése (disk mirroring), azaz az információk egyidejű tárolása a tömb minden elemén. Az adatok olvasása párhuzamosan történik a diszkekről, felgyorsítván az olvasás sebességét; az írás normál sebességgel, párhuzamosan történik a meghajtókon. Az eljárás igen jó hibavédelmet biztosít, bármely meghajtó meghibásodása esetén folytatódhat a működés.

RAID-2

Egyes meghajtókat hibajavító tárolására tartanak fenn. A hibajavító kód lényege, hogy az adatbitekből valamilyen matematikai művelet segítségével redundáns biteket képeznek. A használt eljárástól függően a kapott kód akár több bithiba észlelésére, illetve javítására alkalmas. A védelem ára a megnövekedett adatmennyiség. A módszer esetleges lemezhiba esetén képes annak detektálására, illetve kijavítására

RAID-3

A RAID 3 felépítése hasonlít a RAID 2-re, viszont nem a teljes hibajavító kód, hanem csak egy lemeznyi paritásinformáció tárolódik. Egy adott paritáscsík a különböző lemezekben azonos pozícióban elhelyezkedő csíkokból XOR művelet segítségével kapható meg. A rendszerben egy meghajtó kiesése nem okoz problémát, mivel a rajta lévő információ a többi meghajtó (a paritást tároló meghajtót is beleértve) XOR-aként megkapható.

RAID-4

A RAID 4 felépítése a RAID 3-mal megegyezik. Az egyetlen különbség, hogy itt nagyméretű csíkokat definiálnak, így egy rekord egy meghajtón helyezkedik el, lehetővé téve egyszerre több (különböző meghajtókon elhelyezkedő) rekord párhuzamos írását, illetve olvasását (multi-user mode). Problémát okoz viszont, hogy a paritás-meghajtó adott csíkját minden egyes íráskor frissíteni kell (plusz egy olvasás és írás), aminek következtében párhuzamos íráskor a paritásmeghajtó a rendszer szűk keresztmetszetévé válik.

RAID-5

A RAID 5 a paritás információt nem egy kitüntetett meghajtón, hanem „körbeforgó paritás” (rotating parity) használatával, egyenletesen az összes meghajtón elosztva tárolja, kiküszöbölve a paritás-meghajtó jelentette szűk keresztmetszetet. Mind az írási, mind az olvasási műveletek párhuzamosan végezhetőek. Egy meghajtó meghibásodása esetén az adatok sértetlenül visszaolvashatóak, a hibás meghajtó adatait a vezérlő a többi meghajtóról ki tudja számolni.

Nyomtatók, egér, billentyűzet

Nyomtatók

Mátrixnyomtatók

A nyomtatófejben apró tűk vannak (általában 9 vagy 24 db). A papír előtt egy kifeszített festékszalag mozog, amelyre a tűk ráütnek, és létrehoznak a papíron egy pontot. A kép ezekből a pontokból fog állni. A tűket elektromágneses tér mozgatja, és rugóerő húzza vissza eredeti helyükre. Ezzel az eljárással nem csak karakterek, hanem képek, rajzok is nyomtathatóak. A nyomtatott képek felbontása gyenge, a nyomtató lassú viszont olcsók és nagyon megbízhatók.

Tintasugaras nyomtató:

A tintasugaras nyomtatók tintapatronok segítségével tintacseppeket juttatnak a papírlapra. A patronban van egy porlasztó, ez megfelelő méretű tintacseppekre alakítja a tintát, és a papírlapra juttatja azt. A színes tintasugaras nyomtató színes tintapatronokat használ, általában négy alapszín használatával keveri ki a megfelelő árnyalatokat: ciánkék, bíborvörös, sárga és fekete színek használatával. Minden tintasugaras nyomtató porlasztással juttatja a tintacseppeket a papírlapra, de a porlasztás módszere változó. Ez történhet piezoelektromos úton, elektrosztatikusan, vagy gőzbuborékok segítségével.

A gőzbuborékos nyomtató a következő módon működik: A nyomtató cserélhető tintapatronja a papír felett oldalirányban mozog. A nyomtatófejben lévő, tintával töltött kamrácskákhoz szabad szemmel alig látható fúvókák (porlasztók) kapcsolódnak. Azokat a kamrákat, mely a nyomtatandó képrészlet soron következő képpontjához szükségesek, elektromos impulzus melegíti fel, minek következtében a tinta a melegítési helyeken felforr, és a keletkező gőzbuborék egy-egy tintacseppet lő a porlasztókon keresztül a papírlapra. A tintasugaras nyomtatók egy-egy karaktert sokkal több képpontból állítanak össze mint például a mátrixnyomtatók, ezért sokkal szebb képet is adnak annál: megfelelő tintasugaras nyomtatóval igen jó minőségű, színes képek, akár fotók is nyomtathatók.

Lézernyomtató

A nyomtató szíve egy fényérzékeny anyaggal bevont forgó henger. Egy-egy lap nyomtatása előtt elektromosan feltöltődik. Ezt követően egy lézer fénye pásztázza végig a hengert hosszában, amelyet egy nyolcszögletű tükörrel irányítanak a hengerre. A fényt modulálják, hogy világos és sötét pontokat kapjanak. Azok a pontok, ahol fény éri a hengert, elvesztik elektromos töltésüket. Miután egy pontokból álló sor elkészült, a henger elfordul és elkezdődhet a következő sor előállítás. Később az első sor eléri a tonerkazettát, amely elektrosztatikusan érzékeny fekete port tartalmaz. A por hozzátapad a még feltöltött pontokhoz, így láthatóvá válik a sor. Tovább fordulva a bevont henger hozzányomódik a papírhoz, átadva a papírnak a festéket. A papír ezután felmelegített görgők között halad el, ezáltal a festék véglegesen hozzátapad a papírhoz. A lézernyomtatók kiváló minőségű képet készítenek, nagy a rugalmasságuk, sebességük és elfogadható a költség.

Egér

Az egér egy grafikus felületen való mutató mozgatására szolgáló periféria. Az egéren egy, kettő vagy akár több nyomógomb van, illetve egy görgő is lehet rajta. Belsejében található érzékelő felismeri és továbbítja a számítógép felé az egér mozgását egy sima felületen Optikai Az optikai egér a mozgásokat egy optikai szenzor segítségével ismerte fel, mely egy fénykibocsátó diódát használt a megvilágításhoz. Az első optikai

egereket csak egy speciális fémes egérpadon lehetett használni, melyre kék és szürke vonalak hálójá volt felfestve. Miután a számítógépes eszközök egyre olcsóbbak lettek, lehetőség nyílt egy sokkal pontosabb képelemző chip beépítésére is az egérbe, melynek segítségével az egér mozgását már szinte bármilyen felületen érzékelni lehetett, így többé nem volt szükség speciális egérpadra. Ez a fejlesztés megnyitotta a lehetőséget az optikai egerek elterjedése előtt. A modern optikai egerek egy reflexszenzor segítségével sorozatos képeket készítenek az egér alatti területről. A képek közötti eltérést egy képelemző chip dolgozza fel, és az eredményt a két tengelyhez viszonyított elmozdulássá alakítja.

Mechanikus Egy golyó két egymáshoz képest 90 fokban elhelyezett tengelyt forgat, melyek továbbítják a mozgását a fényáteresztő résekkel rendelkező korongoknak. Az optocsatolók infravörös LEDjei átvilágítanak a hozzájuk tartozó korongok résein. Bármely korong elfordulásakor a rajta lévő rések átengedik LED fényét, míg a rések közötti fogak nem. Végeredményben az egér elmozdulása fényimpulzusok sorozatává változik, mégpedig annál több fényimpulzus keletkezik, minél nagyobb az egér által megtett út. A fényérzékeny szenzorok érzékelik a fényimpulzusokat és elektromos jelekké alakítják.

Billentyűzet

A billentyűzet gombjai kábelezés szempontjából egy ún. billentyűzet-mátrixban vannak elhelyezve. Egy meghatározott billentyű lenyomásának vagy felengedésének észlelésekor a belső mikroprocesszor egy, az adott billentyűt egyértelműen azonosító ún. scan-kódot küld a számítógép felé. Ugyanezen billentyű felengedésekor a mikroprocesszor egy másik, felengedési scan-kódot továbbít a billentyűzet-illesztő áramkör felé. Ezáltal részint kiküszöbölhető a több billentyű közel egyidejű lenyomásából adódó jelenség, a karakterek "elvesztése". A megfelelő gomb vagy kombinációk értelmezése és feldolgozása így teljesen a számítógép billentyűzetkezelő rutinjának feladata.

Telekommunikációs berendezések

Modem

A modem egy olyan berendezés, ami egy vivőhullám modulációjával a digitális jelet analóg információvá, illetve a másik oldalon ennek demodulációjával újra digitális információvá alakítja. Az eljárás célja, hogy a digitális adatot analóg módon átvihetővé tegye. A moduláció különféle eljárások csoportja, melyek biztosítják, hogy egy tipikusan szinuszos jel - a vivő - képes legyen információ hordozására. A szinuszos jel három fő paraméterét, az amplitúdóját, a fázisát vagy a frekvenciáját módosíthatja a modulációs eljárás, azért, hogy a vivő információt hordozhasson. Néhány ok, ami miatt szükséges a közvetítő közegen való átküldést megelőző moduláció: A modem egy másik modemmel működik párban, ezek az átviteli közeg két végén vannak. Szigorú értelemben véve a két modem két adatátviteli berendezést köt össze, azonban a másik végberendezés tovább csatlakozhat az internet felé.

ADSL

Az ADSL vagyis az aszimmetrikus digitális előfizetői vonal valójában egy kommunikációs technológia, amely egy csavart rézérpárú telefonkábelén keresztül juttat el adatot A pontból B pontba. A technológia segítségével a hagyományos modemekhez képest gyorsabb digitális adatátvitel érhető el, ezért igazi áttörés volt megjelenése az internetszolgáltatás piacán. Az ADSL jellemzője, hogy a letöltési és a feltöltési sávszélesség aránya nem egyenlő (vagyis a vonal aszimmetrikus), amely az otthoni felhasználóknak kedvezve a letöltés sebességét helyezi előnybe a feltöltéssel szemben. Mind technikai, mind üzleti okai vannak az ADSL gyors elterjedésének. A technikai előnyt az adja, hogy a zajelnyomási lehetőségeket kihasználva lehetővé teszi

nagyobb távolságon is a gyors adatátvitelt a felhasználó lakása és a DSLAM eszköz között (amely a telefonközpontokban helyezkedik el).

KábelTV-s internet

A kábelszolgáltatók minden városban fő telephellyel rendelkeznek, valamint rengeteg, elektronikával zsúfolt dobozzal szerte a működési területükön, amelyeket fejállomásoknak neveznek. A fejállomások nagy sávszélességű kábelekkel vagy üvegkábelekkel kapcsolódnak a fő telephelyhez. Minden fejállomásról egy vagy több kábel indul el, otthonok és irodák százain halad keresztül. Minden előfizető a rajta keresztülhaladó kábelhez csatlakozik. Így a felhasználók osztoznak egy fejállomáshoz vezető kábelben, ezért a kiszolgálás sebessége attól függ, hogy pillanatnyilag hányan használják az adott vezetékét. A kábelek sávszélessége 750 MHz.

13. Számítógép-hálózati architektúrák, szabványosítók (ISO/OSI, Internet, ITU, IEEE)

14. Kiemelt fontosságú kommunikációs protokollok (PPP, Ethernet, IP, TCP, HTTP, RSA)
