Marton László – Fehérvári Arnold

ALGORITMUSOK ÉS ADATSTRUKTÚRÁK

Z

Győr, 2001

Szerzők:
Dr. Marton László PhD
programtervező matematikus
főiskolai tanár
Fehérvári Arnold
informatikus-mérnök

Lektor:
Dr. Imreh Balázs CSc
egyetemi docens
Dr. Horváth Gyula
egyetemi adjunktus

Készült a SZIF–UNIVERSITAS Kft. sokszorosítójában.
Felelős vezető dr. Jámbor Attila.

TARTALOMJEGYZÉK

ELŐSZÓ

A *számítógép* az ember fontos eszköze. Alaptulajdonságait, a gyors és pontos műveletvégzést, a nagy információtárolási kapacitást, a tárolt *program* alapján történő automatikus működést az élet minden területén felhasználjuk.

Az *átlagos alkalmazó* számára a számítógép egy nem túl bonyolult munkaeszköz, hiszen a feladatot egy készen kapott, könnyen kezelhető, szolgáltatásait a felhasználót segítő–irányító módon felajánló és megfelelő felhasználói útmutatóval ellátott programmal oldja meg. Számára alapfokú számítástechnikai, számítógép-kezelési ismeretek is elegendőek. Magasabb fokú ismeretekre inkább a megoldandó feladat témakörében (pl. egy könyvelőprogram alkalmazójának a számvitelben) van szüksége.

A *számítógépes szoftverfejlesztő* szakembernek, akinek gyakran feladata az, hogy egyedi, új feladatokat megoldó programokat tervezzen és készítsen, vagy „konfekcionált" programokat „méretre igazítson", tehát az ember-gép kommunikáció legmélyebb, legnehezebben elsajátítható és gyakorolható, de leginkább intellektuális szintjén dolgozik, egészen más jellegű felkészültségre, tudásanyagra van szüksége. Egy probléma számítógépre vitelében a legfontosabb részfolyamat egy megfelelő *számítástechnikai modell* keresése, megalkotása.

A *számítástechnikai modell* a számítógépben tárolható adatszerkezetek, *adatstruktúrák* és az ezeket kezelő, átalakító, ezekből információkat lekérdező *algoritmusok* összessége, rendszere. A modellalkotás egy absztrakciós folyamat, amelynek során egyrészt elhagyjuk a feladat szempontjából lényegtelen jellemzőket, másrészt pontosítjuk és formalizáljuk a lényegeseket, állandóan szem előtt tartva a feladatot majd megoldó eszköz (jelen esetben egy számítógépes hardver–szoftver környezet) tulajdonságait és képességeit.

Vannak egyszerűen modellezhető feladatok. Ha például iskolai osztályok név vagy életkor szerint rendezett névsorait kell elkészítenünk, könnyen adódik a modell. Korlátozható elemszámú és pontosan meghatározott típusú (korlátozható hosszú szöveg és megadott intervallumba eső szám) elemekkel rendelkező adatsorokat kell tárolnunk és célszerű cserélgetésekkel a megfelelő sorrendbe állítanunk. Tehát egy lehetséges modellt képez két darab, azonos elemszámú egydimenziós tömb és egy tömbrendező algoritmus.

Nézzünk egy bonyolultabb példát is. Egy olyan számítógépes információs rendszer megalkotása a célunk, amely választ tud adni arra, hogy egy város egy pontjából milyen útvonalon tudunk eljutni személygépkocsival a legrövidebb idő alatt egy másik pontba. Az egyszerűsítési és pontosítási lépések hosszú so

rával jutunk el oda, hogy a megfelelő adatstruktúra legfontosabb része egy matematikai *hálózat* (gráf + él és pontjellemző adatok) és a modell legfontosabb algoritmusai a hálózaton dolgozó útvonalkereső eljárások.

Egy bonyolultabb probléma megoldása közben tapasztalhatjuk azt is, hogy a modell két komponense nem független, az adatstruktúra megválasztása befolyásolja az algoritmus megválasztását és ez fordítva is igaz. Ugyanaz az algoritmus, amely hatékony egyfajta adatstruktúrával, nagyon rossz hatásfokú lehet egy másikkal. Ezért a gyakorlati tervezés általában egy iteratív, javító, módosító lépéseket is tartalmazó folyamat.

A fentebbi két példa egyben kijelöli könyvünk anyagának határait is. Az adatstruktúrák bonyolultságát vesszük anyagunk rendezési elvének, ennek vezérfonala mentén haladunk. Elindulunk a legegyszerűbb adatstruktúrától, a tömbtől és tárgyalva a halmazokat, kollekciókat, majd az alapvető láncolt adatszerkezeteket, a listákat és a fákat, megérkezünk a gráfokhoz és a hálózatokhoz, illetve az ezekkel megoldható – megoldandó feladatokhoz.

Ez az ismeretanyag képezi a szoftverfejlesztés alapjait. De meggyőződésünk, hogy ez egyben az *„informatika matematikája”* is, abban az értelemben, hogy oktatása, bármely informatikai szakterületre vonatkozóan, hasonló szerepet tölt be, mint a matematika oktatása a műszaki–természettudományi diszciplínákban. Tanulása előkészíti, alkalmasabbá teszi, (szabadjon itt egy szakkifejezést használni) „formázza” a leendő informatikus szakembert, úgy a már ismert, oktatott, mint a majd a későbbi munkája során jelentkező új informatikai tudásanyag megismerésére és elsajátítására.

Könyvünket mind a középfokú mind a felsőfokú informatikusképzésben való használatra szánjuk, az előbbinek inkább a befejező, míg az utóbbinak az induló szakaszára gondolva. Használata általános középfokú szintű matematikai és alapszintű informatikai ismereteket feltételez. Nem tűzhettük ki célul a témakör teljességre törekvő, monográfia jellegű feldolgozását, ilyen, régebbi és újabb kiadású magyar nyelvű munkák is vannak [9, 2, 8, 1], ezeket a területen való továbbhaladásra ajánljuk. Korábbi, hasonló célú és jellegű oktatási anyagaink vannak [6, 3] szemléletéhez is igazodva az egyszerűbb modellekre, a könnyen érthető megfogalmazásokra törekszünk.

Az alapozó szintű képzés egyik fő célja az, hogy a tanulóban kialakuljon egyfajta készség arra, hogy a problémát tanulmányozva felismerje annak típusát, és nagy valószínűséggel a helyes úton indulva keresse a megoldást. Ezt elősegítendő témakörönként bőséges feladatanyagot adunk. A tárgyalt modellek és kitűzött feladatok egy részét a teljes megvalósításig, a működő számítógépes programig elvisszük. Természetesen itt nem tárgyalhatjuk a megfelelő program

fejlesztő környezetet is, erre vonatkozóan a szakirodalomra utalunk és hangsúlyozzuk a minél több gyakorlás, a működő program szintjéig végrehajtott feladatmegoldás fontosságát.

A könyv több mint száz részletesen megoldott mintafeladatot, több mint 200 (többségében összetett) kitűzött feladatot, közel száz adatszerkezeti táblát és struktúradiagramot, valamint egyéb magyarázó ábrát tartalmaz.

Győr, 2001. október Szerzők

1. BEVEZETÉS

1.1. Alapfogalmak

1.1.1. Adat, adatstruktúra

A számítógép egy információ-feldolgozó eszköz. Egy *feladat adatai* mindazok az információk, amelyekből kiindulva, amelyekkel műveleteket végezve, amelyeket átalakítva a feladat megoldásáig eljutunk. (Hangsúlyozzuk, hogy egy adat mindig egy feladat vonatkozásában létezik, ebben a viszonyban értelmezhető és jellemezhető.) A számítástechnikai modellekben megjelenő adatokat az alábbiak szerint csoportosíthatjuk és jellemezhetjük.

Az adat lehet *konstans* vagy *változó*. A konstans adat értéke nem változhat a megoldás folyamatában, míg a változóé igen.

Az adat *azonosítója* az adat neve, amellyel rá, vagyis az *értékére* hivatkozhatunk. A változó adatnak kötelezően van azonosítója, ezt szoktuk röviden *változónak* nevezni. A konstansok egy részénél programozás-technikai okokból szintén szoktunk külön azonosítót alkalmazni, más esetekben nem, ilyenkor egyszerűen feltüntetjük az értéket a megfelelő helyen.

Az adat lehet *egyszerű* (más néven: *elemi adat*) vagy *összetett* (más néven: *adatcsoport*). Az összetett adat egyszerű adatokból áll elő valamilyen szabály szerinti csoportképzéssel.

Az adat *típusa* egy komplex, és a számítástechnikai környezethez erősen kötődő jellemző, amely magában foglalja és meghatározza a következőket:

- *Összetettség:* egyszerű vagy összetett, összetetteknél a csoportképzés módja.
- *Műveleti jellemzők:* milyen műveletek végezhetők az adattal, mi a műveletek értelmezése.
- *Tárolási/értelmezési jellemzők:* Az adat mekkora memóriaterületen, és ezen belül milyen kódolási szabályok szerint tárolódik a számítógépben. Ez egyben az ilyen típusú változó által felvehető értékeket (az értékkészletet) is meghatározza.

Az adat *jellegén* azt értjük. hogy a feladat megoldásában *input* (kiinduló adat), *output* (eredmény adat) vagy *munkaterület* szerepet tölt be. Ugyanaz az adat több jelleggel is bírhat.

Az adat *funkciója* a feladat megoldásában betöltött szerepe, általános értelemben véve. Ez egy szöveges leírás, a más (formalizálható) módon meg nem adható adatkapcsolatok megadására is szolgál. (Például, ha egy adatsor tárolását

ra egy tömböt alkalmazunk, akkor az adatok összetartozását már maga az alkalmazott típus jelzi, de azt, hogy melyik változó jelenti az adatsor elemszámát, külön, szövegesen meg kell mondanunk).

A feladat *adatstruktúráját* a feladat adatai alkotják a fentebb felsorolt jellemzőikkel és a köztük lévő kapcsolatokkal együtt.

Ezzel kapcsolatban röviden ki kell térnünk az adatstruktúrák és az összetett adattípusok viszonyára, külön is hangsúlyozva, hogy a kettő nem ugyanaz. Az összetett típusok az adatstruktúrák leírásának jól használható eszközei, de nem maguk az adatstruktúrák. Ez a különbség ott is megmutatkozik, hogy a típus erősen programozási környezetfüggő fogalom (például a Pascal nyelvben léteznek bizonyos halmaz adattípusok, míg a C nyelvben nem, de ugyanazt az adatkapcsolati formát természetesen a C-ben is meg tudjuk valósítani, csak más eszközökkel).

1.1.2. Algoritmus

Az *algoritmus,* teljes általánosságában, egy bonyolult számítástudományi fogalom, itt egy egyszerűsített formájával dolgozunk. Az algoritmus *műveletekből* és *vezérlő szerkezetekből* épül fel.

A *művelet* általános értelemben véve egy olyan átalakítás (transzformáció), amely az adatok aktuális értékeit felhasználva előállítja az adatok új értékeit.

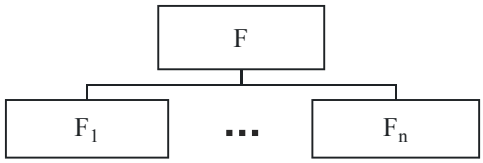
Vannak egyszerűbb műveletek (pl. egy változó értékét megnöveljük eggyel) és bonyolultabb műveletek (pl. egy másodfokú egyenlet egy valós gyökének kiszámítása, a megfelelő képlettel). A bonyolultabb műveletek egyszerűbbekből épülnek fel. Az egyszerűség és bonyolultság itt nagyon viszonylagos fogalmak. Függnek a számítástechnikai környezettől, például van olyan programnyelv, amelyben két adatsor elemenkénti összeadása alapművelet, egy utasítással leírható, míg más nyelvekben ehhez egy összetett műveletsort kell leírni. Még inkább függenek a modellezés részletességétől, finomságától is, például egy nagyobb feladat megoldásában egy adatsor rendezése lehet egy relatíve nagyon kicsi rész, amelyet nem részletezünk, egy műveletnek tekintünk, és ennek megfelelően is jelzünk az algoritmus leírásában, de egy ugyanilyen rendezés egy másik feladatban lehet ennek teljesen kirészletezendő lényegi része, tulajdonképpen maga az algoritmus.

A *vezérlő szerkezetek* a feladat műveletekre bontását, és ezek végrehajtási sorrendjét írják le, beleértve egy-egy művelet adott esetben való elhagyását, vagy éppen többszöri végrehajtását is.

Kizárólag az alábbi, ún. strukturált vezérlőszerkezeteket alkalmazzuk:

Szekvencia

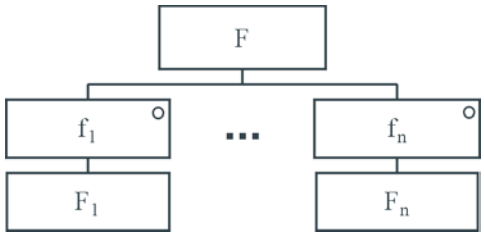
Az `F` művelet felbontása olyan F_1, ..., F_n műveletekre, amelyeknek a felbontás sorrendjében való egymás utáni végrehajtása magának az `F`-nek a végrehajtását jelenti. Struktúradiagram jelölésben lásd 1. ábra.



1. ábra. Szekvencia

Szelekció

Az `F` művelet felbontása olyan F_1, ..., F_n műveletekre, amelyek közül egy kiválasztása és végrehajtása magának az `F` műveletnek a végrehajtását jelenti. A kiválasztást egy f_1, ..., f_n feltételrendszer szabályozza, ahol az f_i feltétel teljesülése az F_i kiválasztásának a feltétele. A feltételrendszer független feltételekből áll, tehát minden konkrét esetben legfeljebb egy teljesül. Ha egy sem teljesül, akkor a szelekció (ebben az esetben) az üres tevékenységet képviseli. Struktúradiagram jelölésben 2. ábra.



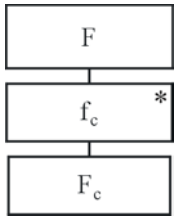
2. ábra. Szelekció

Iteráció

Az `F` művelet végrehajtása egy F_c műveletnek az egymás utáni ismételt végrehajtásával. Az ismételtetést egy f_c feltétel szabályozza. Az iteráció másik elnevezése a *ciklus* (körfolyamat), ilyen értelemben szokás az f_c feltételt ciklusfeltételnek, az F_c-t pedig ciklusmagnak nevezni. A feltétel jellegétől és az ellenőrzés módjától függően többfajta ciklus is értelmezhető. Ezekből itt (tekintettel a programnyelvre) hármat veszünk.

Elöltesztelős ciklus

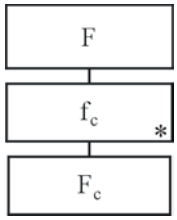
A feltétel vizsgálata megelőzi a ciklusmag végrehajtását, ami csak akkor következik be, ha a feltétel igaz (tehát a kérdés az, hogy kell-e még ismételni). Ha az `fc` hamis, az `F` művelet készen van. Ilyen ciklusnál az ismétlések száma lehet `0` is, ez esetben az iteráció az üres tevékenységet képviseli. Struktúradiagram jelölésben 3. ábra.



3. ábra. Elöltesztelős ciklus

Hátultesztelős ciklus

A feltétel vizsgálata követi a ciklusmag végrehajtását, ami csak akkor következik be, ha a feltétel hamis (tehát a kérdés az, hogy be lehet-e már fejezni). Ha az `fc` igaz, az `F` művelet készen van. Ilyen ciklusnál az ismétlések száma legalább 1. Struktúradiagram jelölésben 4. ábra.



4. ábra. Hátultesztelős ciklus

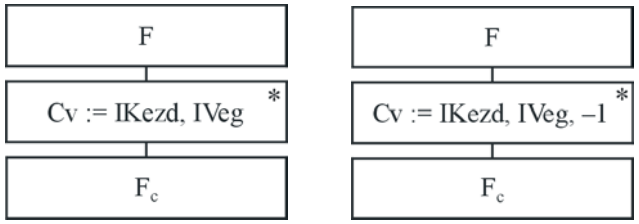
Növekményes ciklus

Egy *intervallum minden elemén, egyesével* végiglépkedve, minden értékre egyszer végrehajtja a ciklusmagot. A haladás iránya lehet:

- a legkisebb értéktől a legnagyobbig *növekvően*
- a legnagyobb értéktől a legkisebbig *csökkenően*

a közben felvett értékeket egy változó, a *ciklusváltozó* tárolja.

Jelölje `Cv` a ciklusváltozót, `IKezd` az intervallum legkisebb, `IVeg` pedig a legnagyobb értékét. Struktúradiagram jelölésben a kétféle növekményes ciklus az 5. ábrán látható.



5. ábra. Növekményes ciklus

Ha az algoritmus megtervezésénél a teljes feladatból, mint egy komplex műveletből indulunk ki, és a felbontás során a fenti vezérlőszerkezeteket alkalmazzuk, egyértelműen meghatározzuk a műveletsor (egyetlen) kezdőpontját, (egyetlen) végpontját és a műveletek végrehajtási sorrendjét.

1.1.3. Számítástechnikai modell

Egy feladat *számítástechnikai modelljén* a feladat megoldásának adatstruktúráját és algoritmusát értjük.

A modelleket több szempont szerint is osztályozhatjuk, csoportosíthatjuk. Didaktikai szempontból célszerűbbnek tartjuk az adatstruktúrák kiemelését, az ezek szerinti csoportosítást. Az adatstruktúrákat vizsgálva a legtöbb feladatnál találhatunk általánosan alkalmazott, szabványos építőelem jellegű összetevőket, amelyek maguk is tekinthetők adatstruktúráknak. Ezek – éppen jellegüknél fogva – külön névvel is rendelkeznek, mint például tömb, halmaz, lista. Hívhatjuk őket akár *nevezetes* adatstruktúra elemeknek, vagy nevezetes adatstruktúráknak is.

A könyvünkben a tárgyalt modellek viszonylag egyszerűek, olyan értelemben is, hogy könnyen megtalálható az adatstruktúrában a domináns, meghatározó nevezetes építőelem. Az anyagot e szerint tárgyaljuk, tehát például a Halmazok című fejezetben olyan modellek szerepelnek, amelyeknél az adatstruktúra domináns része valamilyen halmaz adatstruktúra.

A modellek helyességének (a kitűzött feladatnak való megfelelésének) vizsgálata természetesen meghaladja könyvünk kereteit. Az egyszerűbb feladatoknál a helyesség nyilvánvalóan következik, a bonyolultabbakkal kapcsolatosan a szakirodalomra utalunk. Hasonló okokból nem törekedhetünk arra sem, hogy

minden problémánál a legjobb (elméletileg legelegánsabb, vagy valamilyen szempontból optimális) megoldást adjuk meg.

1.2. A modellezés módszerei

A modellalkotáshoz meg kell választanunk az alkalmazandó fogalom és eszközrendszert (programtervezési–programozási paradigmát). A bevezető jellegű tananyagokban általunk hatékonyabbnak tartott alulról építkező, induktív oktatás a hagyományos, *procedurális* szemléletmódot indokolja, a jelenleg már szintén széles körben alkalmazott objektumorientációval szemben.

Az ismeretek közvetítő, hordozó közegeként a *Pascal* nyelvet és az általánosan elérhető gépi hátteret figyelembe véve a *Turbo Pascal* valamint a *Delphi* programfejlesztő környezetet választjuk. Bár léteznek „tisztább", a témakör oktatására elvileg alkalmasabb nyelvek is, a *Pascal* választása indokolható. Ez a nyelv, amellett, hogy célkitűzésünknek jól megfelelő eszköztárral rendelkezik, egyben gyakorlati nyelv is, jelentős irodalma van (ezen nemcsak a tankönyveket, hanem az ezen a nyelven írt programszövegeket is értve), nagy programrendszerek készítésére is alkalmas, így a programozás gyakorlata is tanítható vele. A kétféle reprezentáns (Turbo Pascal és Object Pascal) eltérései példáink szintjén nem lényegesek, a teljesen kidolgozott megoldások ismertetésénél ki fogunk térni ezekre, ahol ez szükséges.

Az algoritmusokat lehetőleg mindig *teljesen paraméterezett szubrutinok* (eljárások, függvények) formájában valósítjuk meg. A teljes paraméterezésen azt értjük, hogy a szubrutin kizárólagosan a paramétereken keresztül cserél információt programbeli környezetével, tehát kerüljük a (szubrutin szempontjából) külső változók szubrutinbeli használatát. Ha ettől az elvtől valahol eltérünk, ezt (külön indoklással) mindig jelezzük.

Ennek megfelelően az adatstruktúrát a szubrutin paraméterei és lokális változói alkotják. Ezeket az *azonosító,* a *típus,* a modellben betöltött *funkció* és a szubrutinban való felhasználási *jelleg* (input paraméter, output paraméter, munkaváltozó) megadásával specifikáljuk. Feltételezzük, hogy az algoritmus indulásakor az input paraméterek helyes tartalommal rendelkeznek, ennek ellenőrzése nem része az algoritmusnak.

Az algoritmusokban kizárólag a fentebb tárgyalt strukturált vezérlőszerkezeteket alkalmazzuk. Az algoritmusokban, mint egy részfeladat megoldását, alkalmazhatjuk egy másik feladat megoldását is, szubrutinként beépítve. Néhány esetben, az algoritmus tömör leírása céljából alkalmazzuk a *rekurzív* szubrutinhívást is.

1.3. A modellek leírása

A modellek leírására alapvetően három eszközt alkalmazunk:

- *Adatszerkezeti táblázat:* Ebben felsoroljuk a feladat változóit, és mindegyikre megadjuk a változó funkcióját, azonosítóját, Pascal nyelvi típusát, és jellegét (lásd pl. 1. adatstruktúra). A tömörebb leírás kedvéért a típusokhoz a táblázatban csak típusneveket alkalmazunk, az ehhez esetlegesen szükséges konstans és típusdeklarációkat előrebocsátjuk.
- *Struktúradiagram:* Az algoritmus szerkezetét adja meg, a bonyolultnak az egyszerűbb összetevőkre való bontásával kapjuk. A teljes feladatból, mint egy egységből indulunk ki, és a felbontás során csak a fentebb ismertetett strukturált vezérlőszerkezeteket alkalmazzuk (lásd pl. 1. struktúradiagram). A felbontás mélységét, részletességét általánosan nem kötjük meg, az érthetőséget és a pontosságot tartjuk szem előtt.
- *Programnyelvi szöveg:* A Pascal szubrutin, az esetlegesen szükséges előzetes konstans és típusdeklarációkkal. Ha nem adtunk adatszerkezeti táblázatot, akkor kommentárokban tüntetjük fel a fontosabb változók funkcióját és jellegét. (A jelleget rövidítve az „i", „o", „m" betűkkel jelöljük, alapértelmezése: munkaváltozó: „m"). A szubrutinban csak a fentebb ismertetett strukturált vezérlőszerkezeteket egyértelműen leképező programelemeket alkalmazunk.

A három eszközt vegyesen alkalmazzuk, a konkrét feladat típusa, nagysága, bonyolultsága szerint választva.

A tömörebb leírás céljából modelljeink egymásra is utalhatnak, tehát előfordul, hogy egy feladat megoldásának részeként egy másik feladat megoldását is felhasználjuk. Ugyanezen célból alkalmazunk bizonyos általános rendeltetésű szoftver erőforrásokat (konstansok, típusok, eljárások, függvények).

A feltételezett fejlesztőrendszerektől való minél kisebb függőség céljából, ha csak lehetett mindig olyan megoldásokat készítettünk, amely mind a két rendszerben fordíthatók. Tehát, kerültük az olyan nyelvi eszközök (pl. standard szubrutinok) használatát, amelyek az Object Pascalban már vannak, de a Turbo Pascalban még nincsenek.

Ennek ellenére bizonyos eltérő belső tárolási és ellenőrzési módok miatt egyes megoldásokat programnyelvi szinten külön kellett választani, de ezek csak részletek, részfunkció megvalósítási különbségek, az algoritmusok szerkezete egységes.

A programnyelvi anyagokat Pascal programkönyvtárakba, tehát unitokba csoportosítva adjuk meg. Az `u` (univerzális) kezdőbetűs nevű unitok (pl. `uTombR`) mind a két rendszerben fordíthatók. A `d` (Delphi) kezdőbetűs nevek (pl. `dMKollU`) csak Delphiben, míg a `t` kezdőbetűs nevek (pl. `tMKollU`) csak Turbo Pascalban fordítható unitokat jelölnek. A programokban elhelyezett kommentárok a `t` kezdőbetűs unitokban DOS szövegként, a többiekben a Windows jelkészletével íródtak.

A törzsszövegben és a feladatkitűzéseknél `unitnév.szubrutinnév` formában hivatkozunk a szubrutinokra, pl. `uSzov.JobbTolt`. A unitok jegyzékét a függelék tartalmazza.

1.4. Egyéb terminológia és jelölések

Rendezett adatsoroknál elvben rendezési irányon belül is különbséget kell tenni a csak különböző értékeket tartalmazó, valamint az azonos értékeket is megengedő rendezettség között. Ilyen értelemben:

- *Növekvő* a rendezettség, ha a felsorolás sorrendjében a következő érték *nagyobb,* mint a megelőző.
- *Nem csökkenő* a rendezettség, ha a felsorolás sorrendjében a következő érték *nagyobb vagy egyenlő,* mint a megelőző.
- *Csökkenő* a rendezettség, ha a felsorolás sorrendjében a következő érték *kisebb,* mint a megelőző.
- *Nem növekvő* a rendezettség, ha a felsorolás sorrendjében a következő érték *kisebb vagy egyenlő,* mint a megelőző.

Az egyszerűség kedvéért, általában rendezési irányonként csak a rövidebb (növekvő, csökkenő) jelzőt használjuk mind a két esetre. Ha a további különbségtétel lényeges a feladat szempontjából, ezt külön jelezzük.

A *szövegek* feldolgozásával kapcsolatosan:

- *Szövegsoron,* vagy egy szöveg egy során egy stringet (tetszőleges jelsorozatot) értünk.
- *Szónak* nevezünk egy szövegsor egy részét, ha nem tartalmaz szóközt és ennek a tulajdonságnak a megtartásával nem bővíthető, vagyis vagy maga a teljes sor (amiben nincs szóköz), vagy ha egy sor része, akkor
  - két szóköz között van, vagy
  - a sor elejétől az első szóközig tart, vagy
  - az utolsó szóköztől a sor végéig tart.

- *Szövegen* általában a szövegsorok egy sorozatát (pl. stringek tömbjét).
- *Bekezdésen* értünk olyan szöveget, amely nem tartalmaz üres sort (üres stringet).

Sorsoláson, sorsolással való előállításon azt értjük, hogy az adatot a fejlesztőrendszerben lévő véletlenszám-generátor segítségével állítjuk elő.

Az azonos modellhez (mintafeladathoz) tartozó adatszerkezeti táblát és struktúradiagramot – az egyértelműség kedvéért – közös azonosító számmal látjuk el.

2. ELEMI ADATOK

2.1. Általános jellemzés

Ebben a fejezetben olyan modellekkel foglalkozunk, amelyeknél az adatstruktúra nem tartalmaz összetett típusú adatot, tehát a feladat elemi adatok egy célszerűen választott rendszerével, mint adatstruktúrával, megoldható. Mint a példákból is látni fogjuk, az ilyen feladatokhoz viszonylag kis számú adat szükséges. Nagy számú adat esetén általában igaz az, hogy ésszerű algoritmus csak adatcsoportokra készíthető.

2.2. Mintapéldák

2.2.1. mintafeladat: Határozzuk meg két pozitív egész szám legkisebb közös többszörösét.

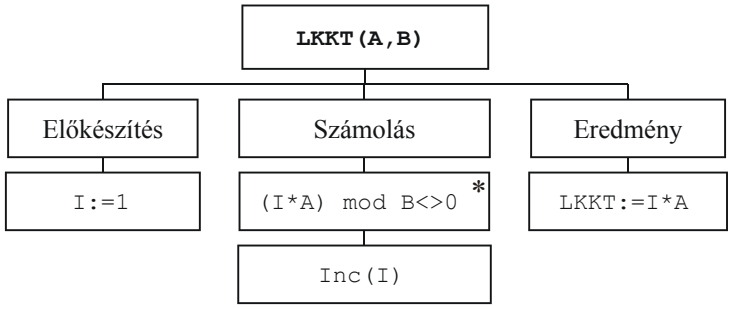
Útmutató ♦ Képezzük az egyik szám többszöröseit (növekvő sorrendben), és mindegyikre megnézzük, hogy osztható-e vele a másik szám. Az első ilyen lesz a megoldás. Megoldás biztosan van, hiszen a két szám szorzata mindkét számmal osztható. A típusokat ezt figyelembe véve választjuk meg.

Adatszerkezet (1)

```
const
  MaxSzam=255;
type
  Szam=1..MaxSzam;
```

Azonosító	Funkció	Típus	Jelleg
A	egyik szám	Szam	input
B	másik szám	Szam	input
I	szorzó	Szam	munka
LKKT	eredmény	Word	output

Struktúradiagram (1)



Szubrutin: `uElemi.Lkkt`.

Megjegyzések

- Az algoritmus gyorsabb lesz (kevesebb szorzás kell), ha biztosítjuk (pl. előzetes cserével) az `A >= B` viszonyt.
- A `Szam` típus bővítő jellegű módosítása esetén a eredmény típusa is értelemszerűen módosítandó.

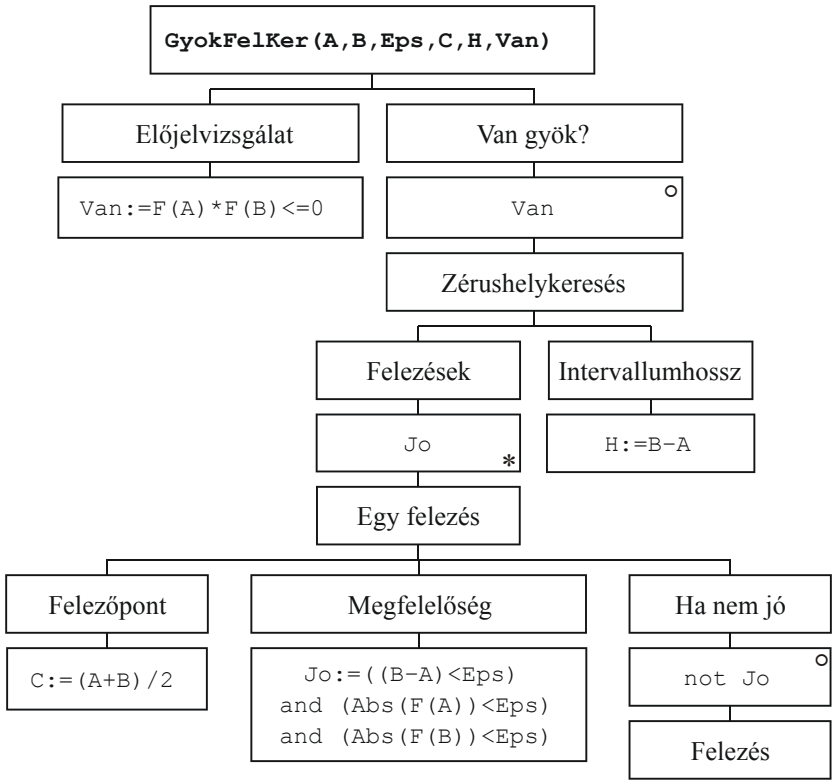
2.2.2. mintafeladat: Határozzuk meg egy `f(x)` folytonos valós függvény zérushelyét egy adott `[a,b]` intervallumban `eps` pontossággal, az intervallumfelezés módszerével! Egy intervallum megfelel a kívánt pontosságnak, ha hossza, valamint mindkét végpontjában a függvényérték kisebb mint `eps`.

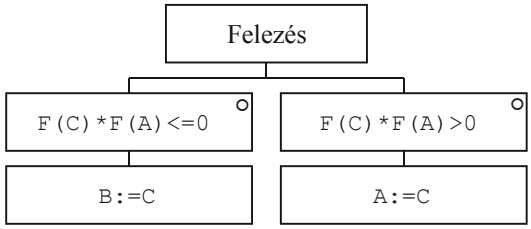
Útmutató ♦ A módszer csak akkor alkalmazható, ha a kiinduló intervallum végpontjaiban a függvényérték ellenkező előjelű. (Ha nem így van, azt az algoritmus jelzi.) Az ilyen intervallumban a folytonosság miatt biztosan páratlan számú (tehát legalább egy) zérushely van. Ha az intervallum nem felel meg a végfeltételnek, akkor felezzük, egyik fele szükségszerűen örökli azt a tulajdonságot, végpontjaiban a függvényérték ellenkező előjelű. Az ismételt felezések és a függvény folytonossága garantálja a megoldást. Eredményként adjuk a keresett intervallum hosszát és középpontját. A középpont tekinthető a közelítő zérushelynek. A függvényérték egy **`function`** `F(X: Real): Real;` típusú függvénnyel adott.

Adatszerkezet (2)

Azonosító	Funkció	Típus	Jelleg
`A`	kiinduló és aktuális intervallum kezdőpontja	`Real`	input, munka
`B`	kiinduló és aktuális intervallum végpontja	`Real`	input, munka
`Eps`	pontosság	`Real`	input
`C`	aktuális és eredmény intervallum középpontja	`Real`	output, munka
`H`	eredmény intervallum hossza	`Real`	output
`Van`	van-e eredmény	`Boolean`	output
`Jo`	megfelel-e az intervallum a pontosságnak	`Boolean`	munka

Struktúradiagram (2)

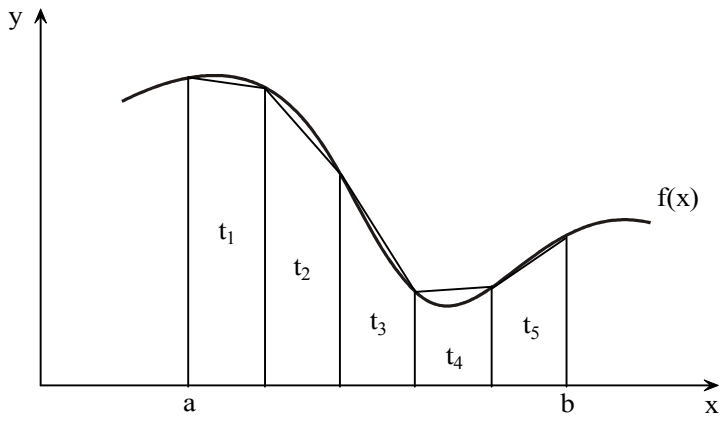




Szubrutin: `uElemi.GyokFelKer`.

Megjegyzés ♦ Mint minden numerikus közelítő eljárásnál, itt is figyelembe kell venni az adatok, valamint a függvény megadásánál a gépi számábrázolási és műveletvégzési pontosság határait, továbbá az elméleti matematikai hátteret. Gyakorlati alkalmazásnál célszerű ezek miatt még egy lépésszám-korlátot is beépíteni a modellbe.

2.2.3. mintafeladat: Határozzuk meg egy `f(x)` *folytonos valós függvény határozott integrálját egy adott* `[a,b]` *intervallumban* `eps` *pontossággal, a „trapéz" módszerrel! Egy közelítő összeg megfelel a kívánt pontosságnak, ha az előző közelítő összegtől való eltérése kisebb, mint* `eps`.



6. ábra. Trapézmódszer

Útmutató ♦ Az alapintervallumot részintervallumokra osztva, az osztópontokban számított függvényértékekkel – szemléletesen szólva – trapézokat képezhetünk, amelyek területösszege közelíti a keresett értéket (6. ábra). Az eltérés a felosztás továbbosztásával csökken. Feltételezzük, hogy a függvényérték

egy **function** `F(X: Real): Real;` típusú függvénnyel adott. Az algoritmusba beépítünk egy lépésszám-korlátot, és visszajelezzük az elért pontosságot (az utolsó és az azt megelőző közelítő összeg eltérése) és lépésszámot is.

Szubrutin: `uElemi.TrapezInt`.

Megjegyzés ♦ Mint minden numerikus közelítő eljárásnál, itt is figyelembe kell venni az adatok, valamint a függvény megadásánál a gépi számábrázolási és műveletvégzési pontosság határait, továbbá az elméleti matematikai hátteret.

2.3. Feladatok

1 ♦ Adott egy hónap és egy nap (hónapon belüli) sorszáma. Meghatározandó a nap sorszáma az éven belül! Feltételezhetjük, hogy nem szökőévről van szó.

2 ♦ Adott kettő, egy napon belüli időpont. Határozzuk meg a köztük eltelt időt! A kiindulás és az eredmény is óra, perc, másodperc alakú legyen.

3 ♦ Adott egy kifizetendő összeg, pozitív egész szám. A felhasználható címletek: 5 Ft, 2 Ft, 1 Ft. Az összeget pontosan (visszaadás nélkül) és a minimális számú címlettel kell kifizetni. Határozzuk meg a címletenkénti darabszámot!

4 ♦ Adott egy pozitív egész szám. Állítsuk elő a kettes számrendszerű alakját! A bináris jegyeket egy stringbe gyűjtsük össze! (`uElemi.BinAlak`)

5 ♦ Állapítsuk meg, hogy egy bájt adott sorszámú bitje 1-es értékű-e!

6 ♦ Állapítsuk meg egy bájt 1-es értékű bitjeinek számát!

7 ♦ Adott `N` pozitív egész szám. Állítsuk elő a Fibonacci sorozat `N`-edik elemét! A képzés módszere: az első két elem értéke 1, ezután minden következő az előző kettő összege.

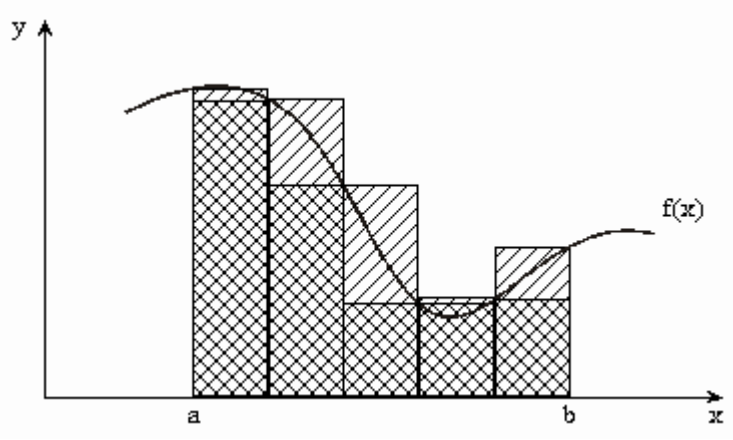
8 ♦ Adott egy `1 < A < 1000` egész szám. Állítsuk elő a prímtényezős felbontását! A prímtényezőket egy stringbe gyűjtsük össze, a „*” elválasztójelet alkalmazva!

9 ♦ Adott `X > 0` valós számhoz határozzuk meg azt a legkisebb `N` egész számot, amelyre igaz, hogy

$$1 + 1/2 + 1/3 + \ldots + 1/N >= X$$

(A sor monoton növő és nem korlátos, tehát van megoldás.)

10 ♦ Határozzuk meg egy f(x) folytonos valós függvény határozott integrálját egy adott [a, b] intervallumban, az intervallum beosztásának fokozatos finomításával, eps pontossággal, a „téglány” módszerrel! Feltételezhetjük, hogy a függvény az intervallumon monoton, így részintervallumonként egyértelműen számítható egy, a valódi integrálértéknél nagyobb és egy, a valódi integrálértéknél kisebb területű téglalap (7. ábra). A kívánt pontosságot akkor érjük el, ha a két területösszeg (nagyobbak összege, kisebbek összege) abszolút értékben vett különbsége kisebb, mint eps.



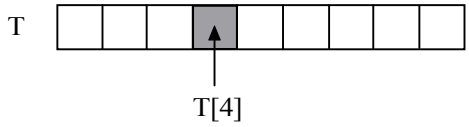
7. ábra. Téglánymódszer

3. TÖMBÖK ÉS STRINGEK

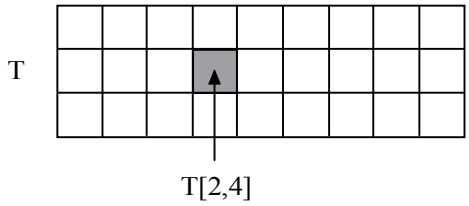
3.1. Általános jellemzés

A tömb olyan adatcsoport, amelynek *elemei azonos típusúak* és ún. dimenziók szerint vannak elrendezve. A dimenziószám azt jelenti, hogy hány kijelölő érték (index) kell ahhoz, hogy az adatcsoportból egy elemet kiválasszunk. Az index sorszámjellegű adat, de nem csak szám lehet.

Ilyen értelemben az *egydimenziós* tömb egy adatsor (egy index) (8. ábra), a *kétdimenziós* egy téglalapalakú adattáblázat (egy sor és egy oszlopindex) (9. ábra). Ha a tömbről a dimenziószám külön említése nélkül beszélünk, akkor általában az egydimenziós tömbről van szó. A kétdimenziós tömb szokásos elnevezése még a *mátrix* vagy a *táblázat* is.



8. ábra. Egydimenziós tömb

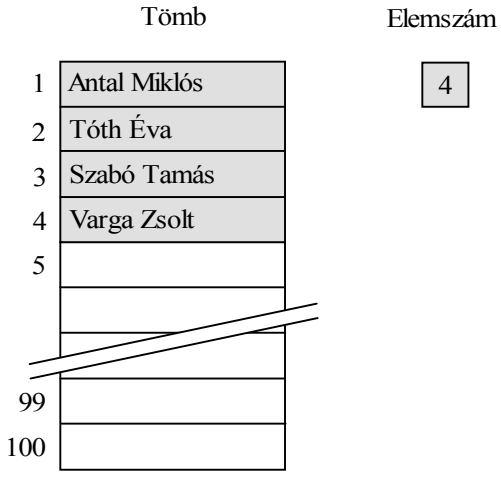


9. ábra. Kétdimenziós tömb

A tömbelemek típusa bármilyen lehet, akár összetett is. Ebből következően, ugyanazt az adatcsoportot többféleképpen is tipizálhatjuk. Például egy téglalapalakú adattáblázatot felfoghatunk egy kétdimenziós tömbként, de úgy is, mint egy olyan egydimenziós tömböt, amelynek elemei maguk is egydimenziós tömbök (a sorok vagy az oszlopok az elemek).

Az ebben a fejezetben tárgyalt tömb *statikus,* tehát a deklarációjában konstans értékekkel rögzítenünk kell a dimenziókat és az egyes dimenziók szerinti alsó és felső indexhatárokat, ezzel meghatározva az egyes dimenziók szerinti maximális elemszámokat is. Az így rögzített maximális területet a programban

változó, az aktuális adatoktól függő mértékben használjuk ki. (Például egy neveket ábécé sorrendbe rendező program írásához tudnunk kell, hogy a program alkalmazási területén maximum hány névből álló névsorok fordulhatnak elő, legyen ez pl. 100. Ha a neveket egyszerű sorszámmal indexeljük, akkor egy külön, 1..100 típusú változóban tartjuk nyilván az aktuálisan rendezendő nevek darabszámát.) A létező elemek szokásos elhelyezése balra zárt, vagyis a feltöltést növekvő indexsorrendben végezzük (10. ábra).



10. ábra. 100 elemű statikus tömb

Következésképpen a tömb, mint adatstruktúra-elem több, legalább 2 változóból épül fel. Ezek között egy természetesen maga a tömbváltozó, a többiek (dimenziónként egy) az *aktuális elemszámot* adják a megfelelő dimenzió szerint. Az aktuális elemszám sohasem lépheti túl a neki megfelelő dimenzió szerinti deklarált maximális értéket.

A tömb, mint jelölési, azonosítási módszer a matematikai–számítástechnikai modelleken kívül is általánosan használatos. Gondoljunk például egy utcanévre (tömbnév) és ezen belül egy házszámra (index), vagy egy helységek közötti távolságtáblázatra (mátrix).

A tömb adatcsoport jellemző kezelési módja az elemenkénti feldolgozás, tehát a műveletekben a tömbelemek az operandusok.

A string és a tömb egy fejezetben való tárgyalását az indokolja, hogy a string – amellett, hogy elemi adatként is kezelhető – ugyanakkor mint speciális tárolási módú egydimenziós tömb is értelmezhető (lásd részletesebben alább).

3.2. Egydimenziós tömbök és stringek

3.2.1. Alapfeladatok

Ahhoz, hogy feladatainkat és ezek megoldásait pontosan megfogalmazhassuk, formálisan is meg kell adnunk legalább egy tömbtípust. A tömbkezelő algoritmusok szerkezete alapjában véve független attól, hogy mik a tömb elemei, erre a célra az általános egész számot, az Integer típust választjuk. Szükség van még index és darabszám típusokra. A maximális elemszámot is rögzítenünk kell valamilyen konkrét értékben, legyen ez 100. Ennek megfelelően:

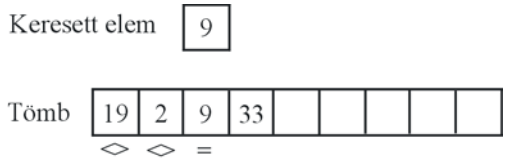
```
const
  EMaxDb=100;  {maximális elemszám}
type
  TTElem=Integer;  {elemtípus}
  TEIndex=1..EMaxDb;  {indextípus}
  TEIndex0=0..EMaxDb;
  TEIndex1=1..EmaxDb+1;
  TElemDb=0..EMaxDb;  {darabszámtípus}
  TSor=array[TEIndex] of TTElem;  {tömbtípus}
```

A `TEIndex0` és `TEIndex1` típusok programtechnikai okokból szükségesek, ugyanis egyes tömbkezelő algoritmusokban létrejöhetnek nem valódi, a tömbből kimutató index jellegű értékek is.

Minden adatstruktúránál alapvető fontosságúak az elemi lekérdezés és karbantartás feladatai, nézzük ezeket.

3.2.1.1. mintafeladat: Keressünk meg egy adott értéket egy tömbben!

Útmutató ♦ Egyenként meg kell vizsgálnunk a tömb elemeit. A vizsgálat akkor érhet véget, ha megtaláltuk az értéket, vagy már nincs több vizsgálandó elem. A vizsgálatot a leggyakoribb és legegyszerűbb módon, növekvő indexsorrendben végezzük (11. ábra).

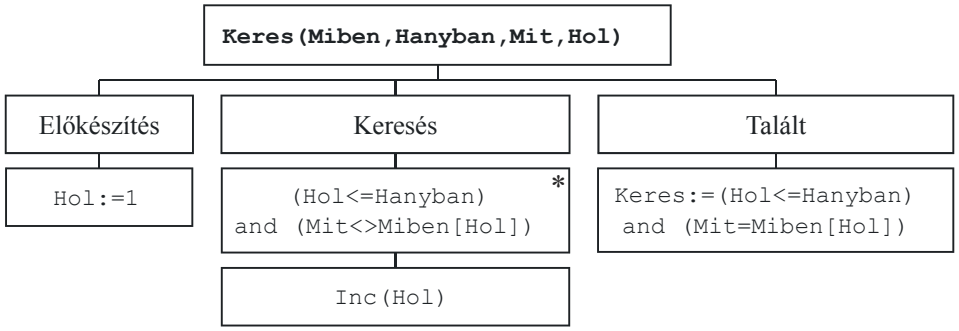


11. ábra. Érték keresése egy tömbben

Adatszerkezet (3)

Azonosító	Funkció	Típus	Jelleg
`Miben`	ebben keresünk	`TSor`	input
`Hanyban`	`Miben` aktuális elemszáma	`TElemDb`	input
`Mit`	a keresett érték	`TTElem`	input
`Hol`	a `Mit` helye, ha létezik	`TEIndex1`	output
`Keres`	a `Mit` létezése	`Boolean`	output

Struktúradiagram (3)

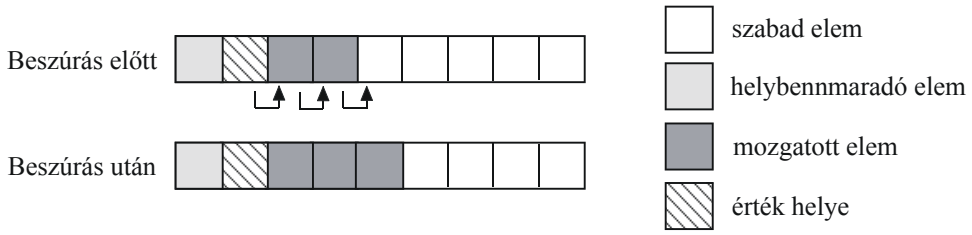


Szubrutin: `uTomb.Keres`.

Megjegyzés ♦ Több `Mit` érték esetén a `Hol` az első indexét adja. Sikertelen keresésnél a `Hol` az utolsó elem utánra mutat.

3.2.1.2. mintafeladat: Szúrjunk be egy adott értéket egy tömb adott helyére!

Útmutató ♦ A balra igazított tárolásnak megfelelően a tömb a nagyobb indexek felé bővül. Az adott helytől kezdődően jobbra léptetjük az elemeket (12. ábra).

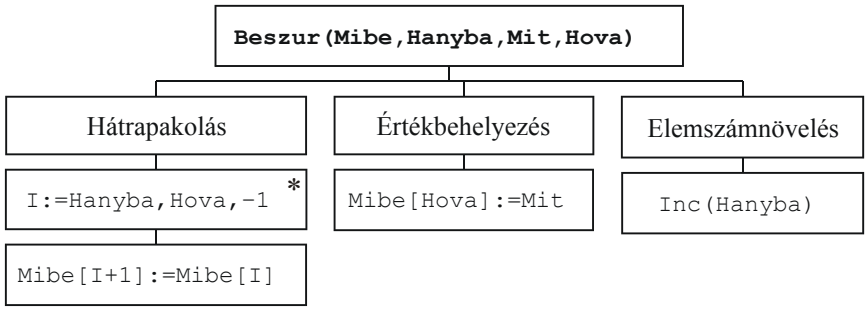


12. ábra. Érték beszúrása egy tömb adott helyére

Adatszerkezet (4)

Azonosító	Funkció	Típus	Jelleg
`Mibe`	ebben szúrunk be	`TSor`	input, output
`Hanyba`	`Mibe` aktuális elemszáma	`TElemDb`	input, output
`Mit`	a beszúrandó érték	`TTElem`	input
`Hova`	a `Mit` helye	`TEIndex`	output
`I`	index	`TEIndex`	munka

Struktúradiagram (4)

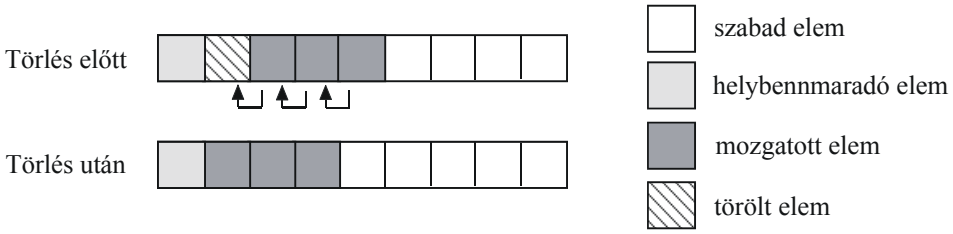


Szubrutin: `uTomb.BeSzur`.

Megjegyzés ♦ Feltételeztük, hogy az új elemszám sem haladja meg a maximumot.

3.2.1.3. mintafeladat: Töröljük a tömb egy elemét!

Útmutató ♦ Az adott helytől kezdődően balra léptetjük az elemeket (13. ábra).

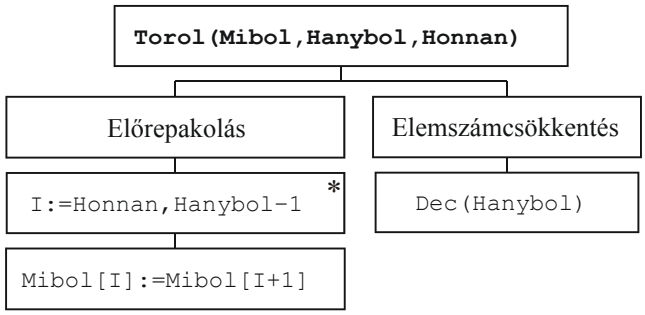


13. ábra. Tömb egy elemének törlése

Adatszerkezet (5)

Azonosító	Funkció	Típus	Jelleg
`Mibol`	ebből törlünk	`TSor`	input, output
`Hanybol`	`Mibol` aktuális elemszáma	`TElemDb`	input, output
`Honnan`	a törlendő elem helye	`TEIndex`	input
`I`	index	`TEIndex`	munka

Struktúradiagram (5)



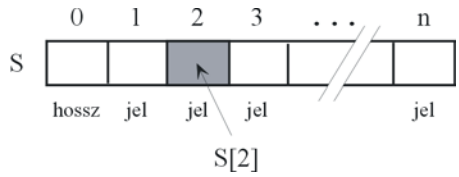
Szubrutin: `uTomb.Torol`.

Megjegyzés ♦ Az új elemszám nulla is lehet.

3.2.2. Stringek

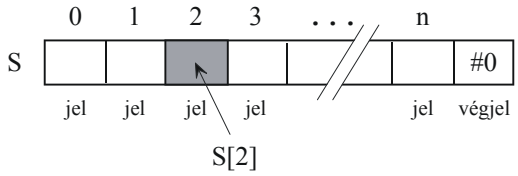
A stringek tulajdonképpen speciális tárolású egydimenziós tömbök. A tömbelemek jelek, az indexek 1-el kezdődő sorszámok. Kétféle alapvető tárolási mód szokásos:

- Pascal stílusú string: Az első elem előtt (a 0 indexű helyen, a hosszbájton) az aktuális elemszám van tárolva. (14. ábra).



14. ábra. Pascal stílusú string

- C stílusú string: Az utolsó valódi elem után egy speciális tartalmú elem (végjel) áll. (15. ábra).



15. ábra. C stílusú string

Mint ebből is látható a string annyiban speciális, hogy magából az adatból megállapítható az aktuális elemszám, tehát ehhez nem kell külön adat. Ez teszi lehetővé, hogy a string elemi adatként is kezelhető, bizonyos műveletekben. Ha a stringet jelenként kell feldolgoznunk, akkor szükség lehet az aktuális elemszám kezelésére is. Erre a programnyelvek tartalmaznak megfelelő standard eszközöket. (Elvben lehetséges lenne a hossz közvetlen kezelése, a hosszbájt átírásával, vagy a végjel áthelyezésével, de mivel a tárolási mód egy nyelven belül is verziófüggő lehet, ezeket a módszereket kerüljük.)

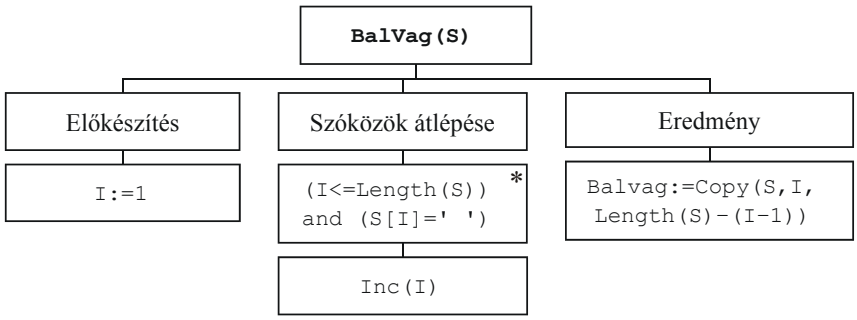
3.2.2.1. mintafeladat: Adott egy S string. Képezzük azt a stringet, amely az eredetiből úgy keletkezik, hogy a kezdő szóközöket töröljük.

Útmutató ♦ Megkeressük az első nem szóköz elemet. A string ezzel kezdődő része lesz az eredmény.

Adatszerkezet (6)

Azonosító	Funkció	Típus	Jelleg
`S`	kiinduló adat	`String`	input
`I`	index	`Byte`	munka
`BalVag`	eredmény	`String`	output

Struktúradiagram (6)



Szubrutin: `uString.BalVag`.

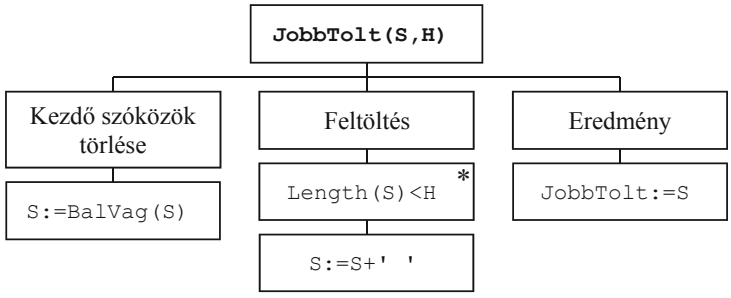
3.2.2.2. mintafeladat: Adott egy S string. Képezzük azt a stringet, amely az eredetiből úgy keletkezik, hogy adott H hosszban, a kezdő szóközök törlésével balra igazítjuk, és jobbról szóközökkel feltöltjük!

Útmutató ♦ Felhasználjuk a `BalVag` szubrutint.

Adatszerkezet (7)

Azonosító	Funkció	Típus	Jelleg
`S`	kiinduló adat	`String`	input
`JobbTolt`	eredmény	`String`	output

Struktúradiagram (7)



Szubrutin: `uString.JobbTolt`.

Megjegyzés ♦ A szubrutin a H értékétől függetlenül megtartja a nem szóköz jeleket.

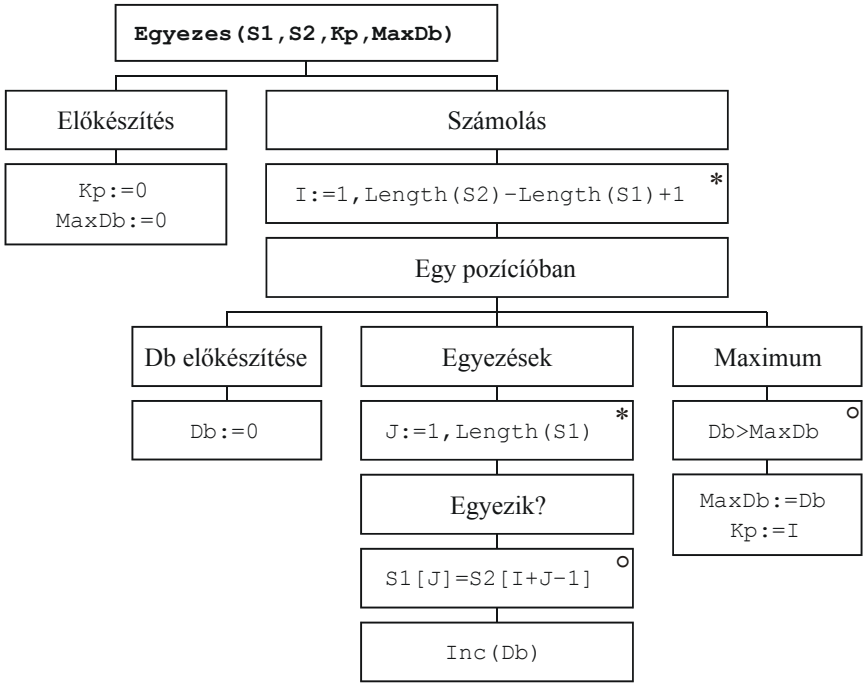
3.2.2.3. mintafeladat: Adott egy `S1` és egy `S2` string, az `S1` nem lehet hosszabb, mint az `S2`. Keressük, hogy az `S2` mely részével egyezik legtöbb helyen az `S1`. Az eredményt a legtöbb helyen egyező rész kezdőpozíciója és az egyező helyek száma jelenti. A vizsgálatot balról jobbra haladva végezzük el!

Útmutató ♦ Jelenként összehasonlítjuk az `S1`-et az `S2` megfelelő szeleteivel.

Adatszerkezet (8)

Azonosító	Funkció	Típus	Jelleg
`S1`	amit keresünk	`String`	input
`S2`	amiben keresünk	`String`	input
`Kp`	maximális egyezésszámú rész kezdőpozíciója	`Byte`	output
`MaxDb`	egyezések száma a maximális részben	`Byte`	output
`I, J`	index	`Byte`	munka
`Db`	számláló	`Byte`	munka

Struktúradiagram (8)



Szubrutin: `uString.Egyezes`.

3.2.2.4. mintafeladat: A bináris alakból kiindulva, határozzuk meg egy pozitív egész szám hexadecimális alakját! Mind a kiindulás, mind az eredmény egy stringben tárolt.

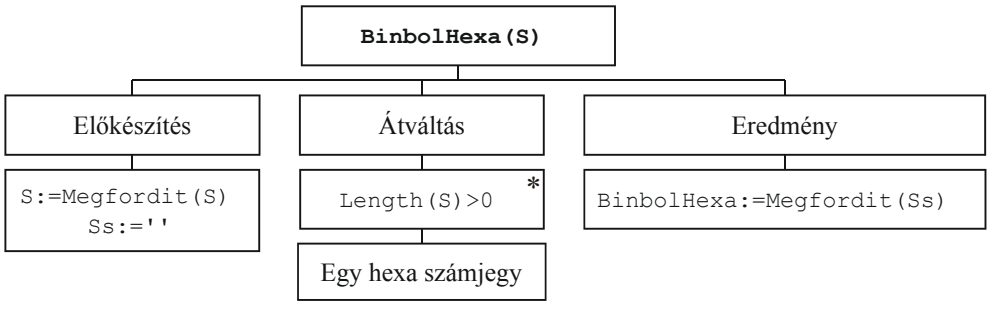
Útmutató ♦ Mivel a szám nagyságára nem adtunk korlátozást, korrekt megoldást csak arra tudunk alapozni, hogy a bináris alakot (a végéről indulva) 4 bitenként összefogva, és a megfelelő hexadecimális jeggyel helyettesítve, a hexadecimális alakot kapjuk. A hexadecimális jegyeket egy konstans stringben adjuk meg, ebből egyszerű indexezéssel emeljük ki a megfelelő jegyet. Hasonlóan kezeljük a számításhoz szükséges hatványokat. Az egyszerűbb kezelés céljából a kiinduló stringet megfordítjuk. A megfordításhoz az `uString.Megfordit` szubrutint használjuk.

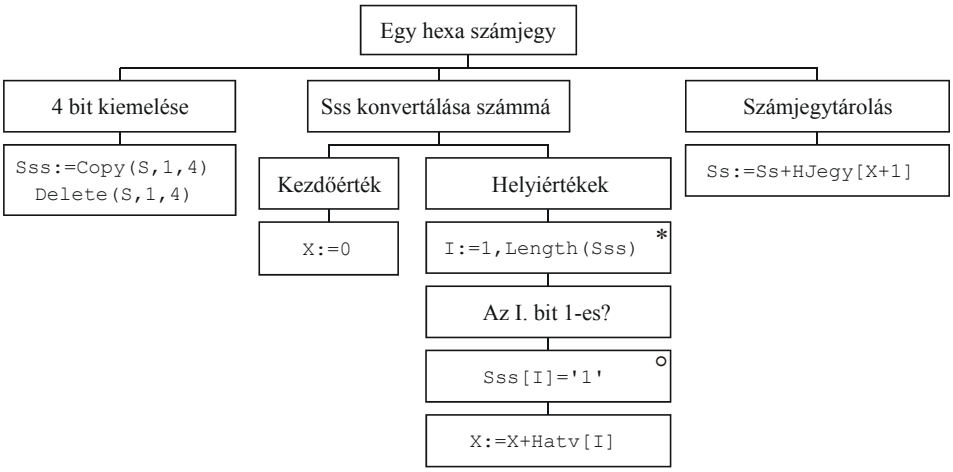
Adatszerkezet (9)

```
const
  HJegy: String='0123456789ABCDEF';
  Hatv: array[1..4] of Byte=(1, 2, 4, 8);
```

Azonosító	Funkció	Típus	Jelleg
`S`	bináris alak	`String`	input
`BinbolHexa`	hexadecimális alak	`String`	output
`I`	index, ciklusváltozó	`Byte`	munka
`X`	hexadecimális jegy értéke	`Byte`	munka
`Ss`	hexadecimális jegyek	`String`	munka
`Sss`	bináris jegyek	`String`	munka

Struktúradiagram (9)



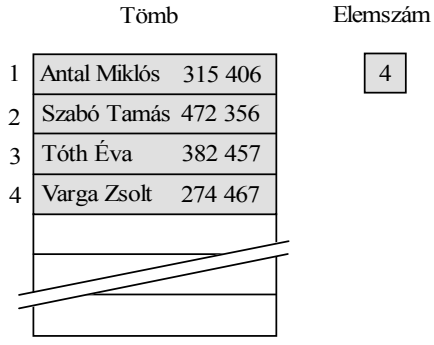


Szubrutin: `uString.BinbolHexa`.

3.2.3. Rendezett tömbök

3.2.3.1. Rendezettség és keresés

Az adatstruktúrák leggyakoribb lekérdező jellegű felhasználása a keresés, amikor is azt szeretnénk megtudni, hogy egy érték előfordul-e az adatok között, és ha igen, akkor hol van. A keresés csak akkor lehet hatékony (viszonylag gyors), ha az értékek nagysága és az adatszerkezetben elfoglalt helye között olyan szabályszerű összefüggés van, amit a keresésnél fel tudunk használni. Egy köznapi példa: egy telefonkönyvben azért találjuk meg viszonylag gyorsan a keresett nevet, mert a könyv tételei (egy tételhez tartozik a név, telefonszám stb.) a nevek ábécé sorrendjében vannak. Ha nincs ilyen szabály, akkor a struktúra minden elemét meg kell vizsgálni (lásd pl. 3.2.1.1. mintafeladat).



16. ábra. Növekvő rendezettség

Az egydimenziós tömböknél a legegyszerűbb ilyen szabály a rendezettség, amely azt jelenti, hogy az elemek vagy (összetett elemeknél, pl. telefonkönyv) az elemekhez rendelt értékek nagysága az index értékével nem csökken (ezt nevezzük *növekvő* rendezettségnek, 16. ábra), vagy nem nő (ezt nevezzük *csökkenő* rendezettségnek).

Már akkor is gyorsítottuk a keresést, ha ezt a tulajdonságot csak annyiban használjuk ki, hogy nem keresünk tovább, ha a keresett értéknél egy nagyobbat találunk (lineáris keresés).

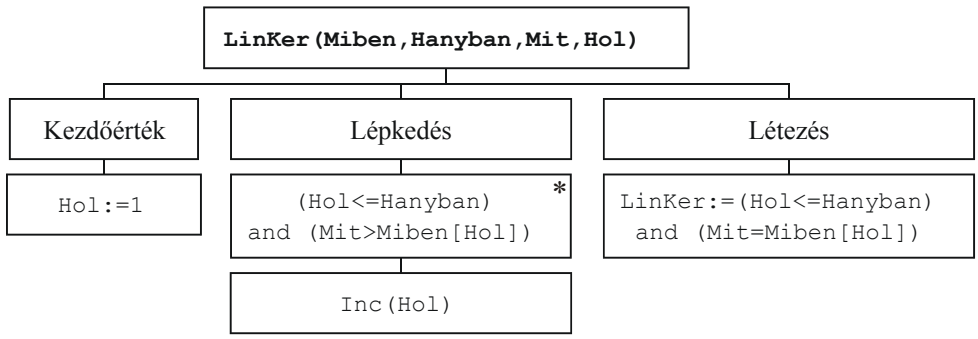
3.2.3.1.1. mintafeladat: Keressünk meg egy adott értéket egy növekvően rendezett tömbben (lineáris keresés)!

Útmutató ♦ A tömb elejétől indulva, egyenként meg kell vizsgálnunk a tömb elemeit. A vizsgálat akkor érhet véget, ha megtaláltuk az értéket, vagy nagyobb értéket találtunk.

Adatszerkezet (10)

Azonosító	Funkció	Típus	Jelleg
`Miben`	ebben keresünk	`TSor`	input
`Hanyban`	`Miben` aktuális elemszáma	`TElemDb`	input
`Mit`	a keresett érték	`TTElem`	input
`Hol`	a `Mit` helye	`TEIndex1`	output
`LinKer`	a `Mit` létezése	`Boolean`	output

Struktúradiagram (10)



Szubrutin: `uTombR.LinKer`.

Megjegyzés ♦ Több `Mit` érték esetén a `Hol` az első indexét adja. Sikertelen keresésnél is értékes információt tartalmaz a `Hol`, hiszen a `Mit` helyét jelzi (pl. beszúráshoz).

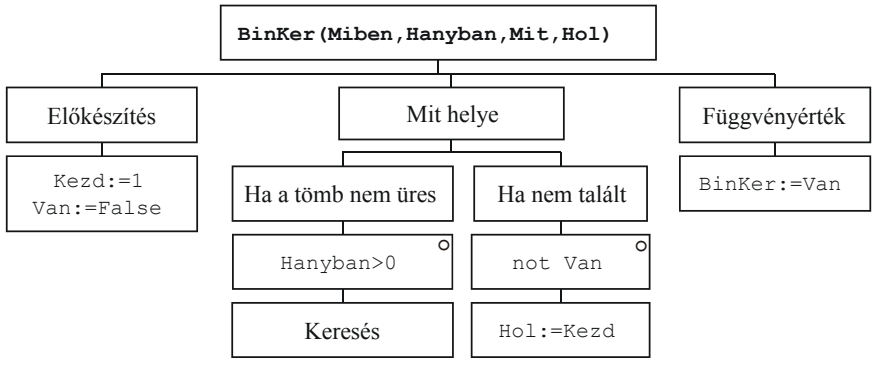
Ennél azonban sokkal többet is kihasználhatunk. A tömb bármely indexe két részre osztja a tömböt: tőle balra nem lehetnek nagyobb értékek, jobbra nem lehetnek kisebb értékek. Tehát egy hely és a keresett érték ismeretében egy vizsgálattal ki tudjuk zárni a tömb egyik részét. Az algoritmus akkor a leggyorsabb, ha a két rész azonos vagy közel azonos (max. eltérés: 1) számú elemet tartalmaz. Így jutunk a bináris kereséshez.

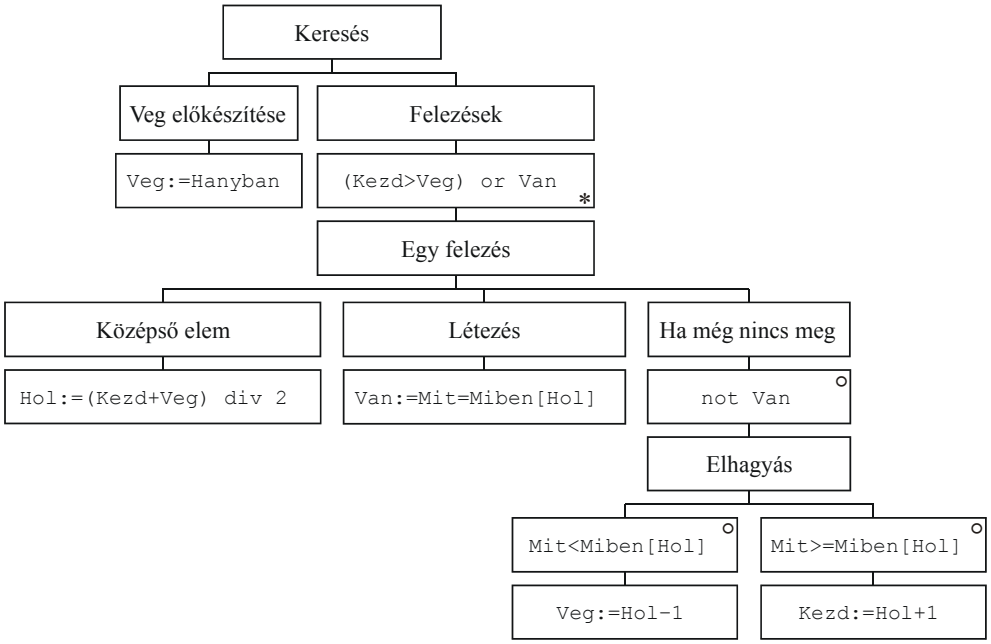
3.2.3.1.2. mintafeladat: Keressünk meg egy adott értéket egy növekvően rendezett tömbben (bináris keresés)!

Adatszerkezet (11)

Azonosító	Funkció	Típus	Jelleg
`Miben`	ebben keresünk	`TSor`	input
`Hanyban`	`Miben` aktuális elemszáma	`TElemDb`	input
`Mit`	a keresett érték	`TTElem`	input
`Hol`	a `Mit` helye	`TEIndex1`	output
`BinKer`	a `Mit` létezése	`Boolean`	output
`Kezd`	aktuális kezdőindex	`TEIndex1`	munka
`Veg`	aktuális végindex	`TEIndex0`	munka
`Van`	a `Mit` létezése	`Boolean`	munka

Struktúradiagram (11)





Szubrutin: `uTombR.BinKer`.

Megjegyzés ♦ Több `Mit` érték esetén a `Hol` valamelyik (nem feltétlenül az első) indexét adja. Sikertelen keresésnél a `Hol` a `Mit` helyét jelzi (pl. beszúráshoz).

A kereső szubrutinokat hatékonyan felhasználhatjuk a karbantartó jellegű feladatokhoz, lásd pl. `uTombR.BeSzurR_B`, `uTombR.TorolR_B`.

Jelölje az elemek számát `N`. A kétféle keresést összehasonlítandó, bizonyítás nélkül megemlítjük, hogy a lineáris keresésnél átlagosan `N/2`, a bináris keresésnél viszont legfeljebb $\mathtt{Log}_2\mathtt{(N)}$ a megvizsgálandó elemek száma. A két nagyságrend aránya nagy elemszámnál jelentős, `N=8000` esetén pl. `300`-szoros. Következésképpen minden olyan esetben, amikor ezt az adatstruktúra lehetővé teszi, a bináris keresést kell alkalmazni.

3.2.3.2. Rendezés

Minden tömbrendező (a tömböt helyben, más adatstruktúra felhasználása nélkül rendező) algoritmus úgy dolgozik, hogy az elemek vizsgálatával és felcserélésével fokozatosan csökkenti a kívánt elhelyezkedéstől való eltérést, közelíti az elemeket a rendezettség szerinti helyükhöz. Az ún. egyszerű (vagy elemi) eljárá

soknál a tömb két részre oszlik. A tömb elején, (vagy a végén) egy már végleges állapot van (rendezett rész), a tömb másik része rendezetlen, és a rendezetlen részből minden lépésben legalább 1 elem átkerül a rendezett részbe. (17. ábra) Ilyen például a kiválasztásos rendezés.

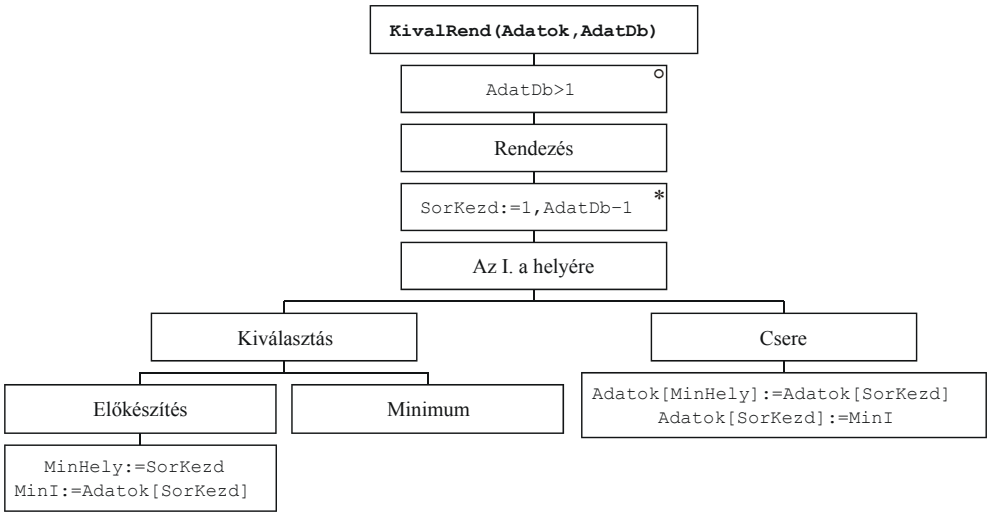
3.2.3.2.1. mintafeladat: Rendezzünk egy tömböt növekvően (minimum-kiválasztás módszere)!

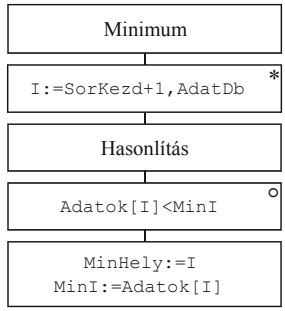
Útmutató ♦ Minden lépésben a rendezetlen rész minimális értékű elemét felcseréljük a rész első elemével (vagyis a rendezett részt növeljük, a rendezetlent csökkentjük 1 elemmel).

Adatszerkezet (12)

Azonosító	Funkció	Típus	Jelleg
`Adatok`	a rendezendő/rendezett tömb	`TSor`	input, output
`AdatDb`	a tömb aktuális elemszáma	`TElemDb`	input
`SorKezd`	a rendezetlen rész kezdete	`TEIndex`	munka
`MinHely`	a minimum helye a rendezetlen részben	`TEIndex`	munka
`I`	index	`TEIndex`	munka
`Mini`	minimális érték a rendezetlen részben	`TTElem`	munka

Struktúradiagram (12)





Szubrutin: `uTombR.KivalRend`.

A hatékonyságot elemezve megállapíthatjuk, hogy a vizsgálatok száma `N * (N - 1) / 2`, a cserék száma `N`, tehát a műveletszám az elemszámmal négyzetesen nő. Bár a vizsgálatok és cserék aránya eltérő, hasonló hatékonyságúak a szomszédos elemek felcserélésével (lásd `uTombR.CserelRend`) és a rendezett részbe való besorolással (lásd `uTombR.BeszurRend`) dolgozó eljárások is.

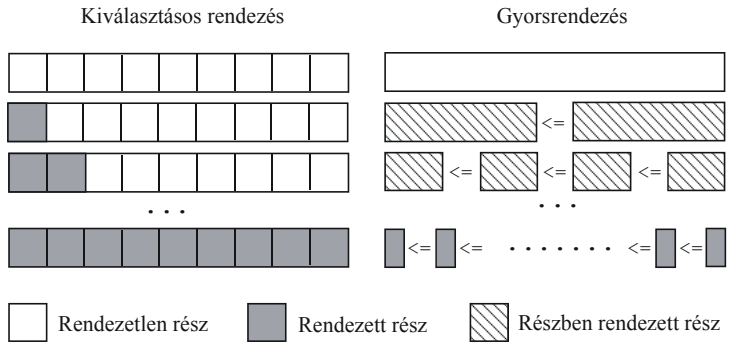
A vizsgálatok és cserék más jellegű szervezésével jobb hatékonyság is elérhető. Tételezzük fel, hogy van egy olyan módszerünk, eljárásunk, amely bármely tömböt két részre tud osztani, az alábbiak teljesülése mellett:

- Az egyik rész bármely eleme kisebb vagy egyenlő, mint a másik rész bármely eleme.
- Minden elemről egy vizsgálattal eldönthető, hogy melyik részbe kerül.
- A két rész elemszáma maximum 1-gyel térhet el egymástól (felezés).

Ezt az eljárást szükség szerinti számban alkalmazva rendezzük a tömböt (17. ábra). Jelölje az elemek számát `N`, akkor az eljárás „meneteinek” (az ábra megosztást tartalmazó sorainak) száma $\text{Log}_2(N)$ hiszen a felezésekkel így jutunk el az egyes elemekig. Minden menetben elemenként egy, tehát összesen $N * \text{Log}_2(N)$ vizsgálatot végzünk, ezzel arányos számú elemi lépésben lesz a tömb rendezett.

A ténylegesen létező tömbrendező eljárások, így a későbbiekben, a verem adatstruktúrákkal kapcsolatosan ismertetendő *gyorsrendezés* is ezt igyekeznek minél jobban megközelíteni.

Megjegyezzük, hogy a fentebb tárgyalt egyszerű tömbrendező algoritmusoknál a felezés nem teljesül, sőt a megosztás a lehető legkedvezőtlenebb arányú, hiszen az egyik részbe csak egy elem kerül. Az is belátható szemléletesen, hogy a felezéses kettéosztásnál az egyes elemek átlagosan közelebb kerülnek a végleges helyükhöz, mint az egyszerű módszereknél (az elem már nem fog átlépni a másik részbe).



17. ábra. Kiválasztásos rendezés és gyorsrendezés

A tömbrendezések hatékonyságával kapcsolatban érdemes még megjegyezni, hogy az előbbiek ellenére, a konkrét gépi megvalósításnál, nem túl nagy elemszámoknál mindig a kiválasztásos módszer bizonyul a leggyorsabbnak. Ennek a magyarázata az, hogy a csere művelet jóval nagyobb időigényű, mint a vizsgálat, és a kiválasztásos módszer igényli a minimális számú (pontosan `N - 1`) cserét.

Végül a nem helybenrendező eljárásokra nézzünk egy példát.

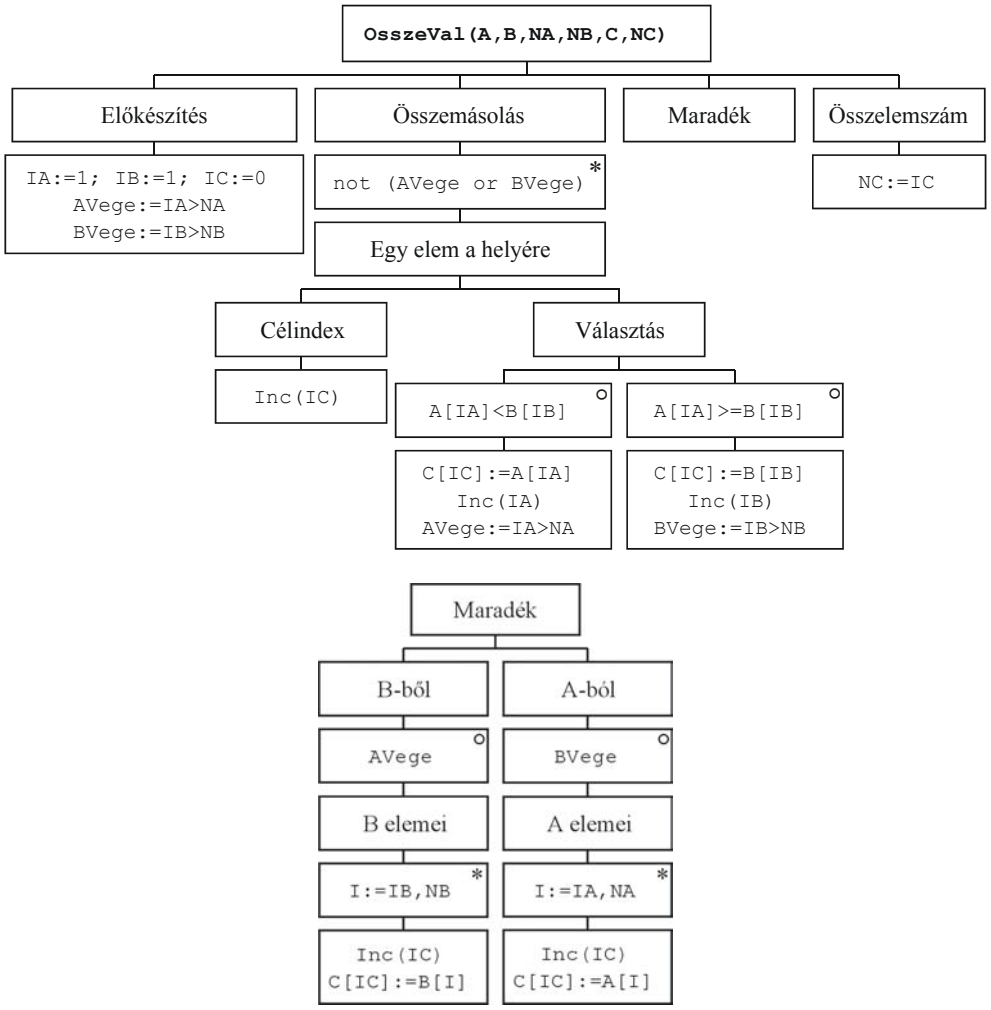
3.2.3.2.2. mintafeladat: Válogassunk össze két növekvően rendezett tömböt!

Útmutató ♦ A két tömböt párhuzamosan, az elejétől kezdve elemenként összehasonlítva vizsgáljuk. Minden lépésben átveszünk egy értéket, a kisebb (vagy egyenlő) értéket az eredménytömbbe. Ha az egyik tömb előbb elfogy, a másik maradékát hozzámásoljuk az eredményhez.

Adatszerkezet (13)

Azonosító	Funkció	Típus	Jelleg
`A`	kiinduló tömb	`TSor`	input
`B`	kiinduló tömb	`TSor`	input
`NA`	az `A` tömb aktuális elemszáma	`TElemDb`	input
`NB`	a `B` tömb aktuális elemszáma	`TElemDb`	input
`C`	eredménytömb	`TSor`	output
`NC`	a `C` tömb aktuális elemszáma	`TElemDb`	output
`IA`	az `A` tömb indexe	`TEIndex`	munka
`IB`	a `B` tömb indexe	`TEIndex`	munka
`IC`	a `C` tömb indexe	`TEIndex0`	munka
`AVege`	az `A` tömb végének elérését jelzi	`Boolean`	munka
`BVege`	a `B` tömb végének elérését jelzi	`Boolean`	munka

Struktúradiagram (13)



Szubrutin: `uTombR.OsszeVal`.

Megjegyzés ♦ Feltételezzük, hogy az eredménytömb aktuális elemszáma nem haladja meg a maximumot.

A rendezett tömböket felhasználhatjuk különféle statisztikák elkészítésére is. Erre nézzünk egy példát.

3.2.3.2.3. mintafeladat: Gyűjtsünk egy érték szerint növekvően rendezett gyakorisági statisztikát!

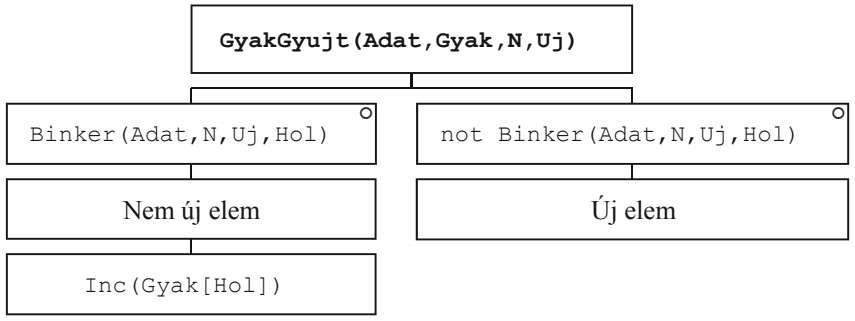
Útmutató ♦ A feladat pontosítva az, hogy egy értéket soroljunk be a statisztikába. A statisztika két, azonos elemszámú tömbből áll. Az egyikben a különböző értékek vannak, növekvő sorrendben, a másikban párhuzamosan, az értékek előfordulási darabszáma. Feltételezzük, hogy legfeljebb `EMaxDb = 100` különböző érték lehet és egy érték legfeljebb `MaxErtDb = 1000` példányban fordulhat elő. A felveendő érték újdonságának, illetve a statisztikábani helyének meghatározásához a bináris keresést használjuk fel.

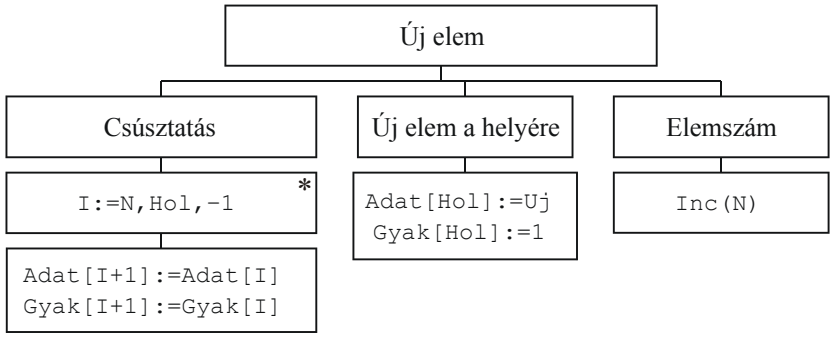
Adatszerkezet (14)

```
const
  MaxErtDb=1000;
type
  TErtDb=0..MaxErtDb;
  TStat=array[TEIndex] of TErtDb;
```

Azonosító	Funkció	Típus	Jelleg
`Adat`	értékek	`TSor`	input, output
`Gyak`	előfordulási darabszámok	`TStat`	input, output
`N`	az `Adat` és `Gyak` közös aktuális elemszáma	`TElemDb`	input, output
`Uj`	a felveendő érték	`TTElem`	input
`I`	index	`TEIndex0`	munka
`Hol`	keresési eredmény	`TEIndex1`	munka

Struktúradiagram (14)





Szubrutin: `uTbStat.GyakGyujt`.

3.2.3.3. Indextáblás rendezettség

Az alkalmazások nagy részében az adatok *fizikai* rendezettsége gyakorlatilag nem biztosítható, rendkívül rossz hatékonysággal járna. Ennek oka lehet például az adatok nagy száma (egy fizikai beszúrás nagyon sok áthelyezést igényelne), vagy a többféle lekérdezési szempont (fizikailag rendezni csak egy szempont szerint lehet).

A vázolt probléma az informatika egyik központi problémája, a megoldások közös jellemzője az, hogy magukat az alapadatokat egy rögzített illetve relatíve nagyon keveset változó sorrendben tároljuk, a rendezettségi, sorrendi információkat pedig kiegészítő adatokkal adjuk meg. Ezt hívjuk *logikai* rendezésnek, illetve rendezettségnek. A kiegészítő adatok akár nagyon bonyolult szerkezetűek (pl. gráfok) is lehetnek. Az alábbiakban a módszert a lehető legegyszerűbb formájával, az egydimenziós tömbök *indextáblás* rendezettségével szemléltetjük.

A modell adatstruktúrája rendezési szempontonként/irányonként egy tömbbel (indextábla) bővül. Ez a tömb írja le a logikai sorrendet az adott szempont/irány szerint. A tömb azonos elemszámú az alapadatok tömbjével. Jelölje az alaptömböt `A` az indextábla tömbjét (röviden: mutatótömböt) `M`, ez a *logikai* sorrendet azon szabály alapján mutatja, hogy a logikai sorrendben az `i`. helyen álló elemre a mutató `i`. eleme mutat, vagyis a *logikailag* i. elem helye az alaptömbben: `M[i]`, értéke pedig: `A[M[i]]`. A mutatótömb indextípusa és elemtípusa ugyanaz, azonos a fizikai index típusával.

Példának vegyük egy olyan tömböt, amely elemenként egy nevet és egy számot tartalmaz.

	Alaptömb		Mutatók		
Fizikai index	**Név**	**Szám**	**Növekvő név szerint**	**Növekvő szám szerint**	**Csökkenő szám szerint**
1	ABA	441233	1	6	1
2	BELA	433232	2	4	2
3	JANI	321423	5	8	7
4	KFT	311234	7	5	3
5	BELGA	319898	8	3	5
6	DIPL	311010	6	7	8
7	CANDIDE	424545	3	2	4
8	CILI	312101	4	1	6

Az indextáblás algoritmusok készítésének alapszabályai:

- *Az adatokra mindig közvetve, a mutatón keresztül hivatkozunk.*
- *Az adatok helyett indexeik mozognak.*

Ebben az adatstruktúrában a beszúrás pl. úgy oldható meg, hogy az *új adat* mindig a fizikai sor végére kerül, indexét pedig beszúrjuk a mutatóba. A törlés, minimális adatmozgatással elvégezhető úgy, hogy a törlendő elem helyére a fizikailag utolsót tesszük, a mutatót pedig ennek megfelelően módosítjuk.

Mint látható az adatmozgatás átkerült a kiegészítő adatok szintjére. Ahhoz, hogy a mozgások számát ezen a szinten is minimalizáljuk, már bonyolultabb adatstruktúrákra (pl. fákra) lenne szükség.

A „normál" algoritmusok indextáblás átírásaira nézzünk mintapéldaként egy rendezést, egy keresést és egy beszúrást. A példákhoz be kell vezetnünk egy új típust a mutatótömb számára.

```
type
  TEIndexSor=array[TEIndex] of TEIndex;
```

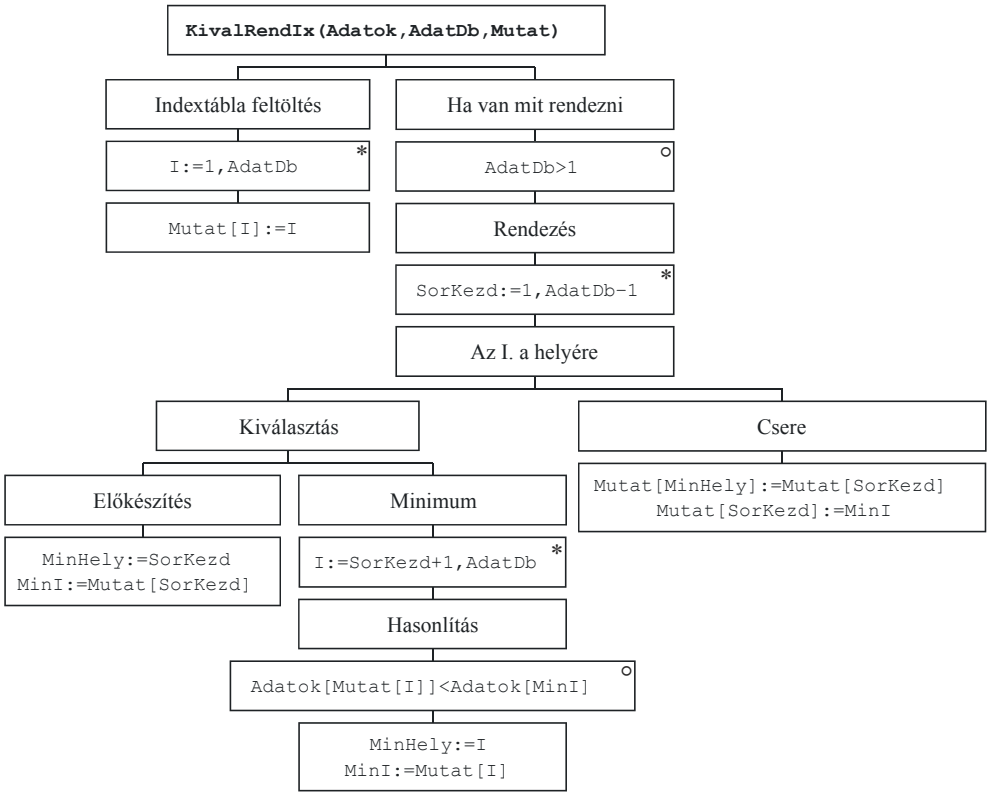
3.2.3.3.1. mintafeladat: Rendezzünk egy tömböt növekvően, a kiválasztás módszerével, indextáblával!

Útmutató ♦ Az alapadatok tömbje nem változik, input jellegű adat lesz. Az eredmény maga a mutatótömb. Az algoritmuson belül követjük a fentebbi átírási alapszabályokat.

Adatszerkezet (15)

Azonosító	Funkció	Típus	Jelleg
`Adatok`	a rendezendő tömb	`TSor`	input
`AdatDb`	a tömbök aktuális elemszáma	`TElemDb`	input
`Mutat`	a mutatótömb	`TEIndexSor`	output
`SorKezd`	a rendezetlen rész kezdete	`TEIndex`	munka
`MinHely`	a minimum helye a rendezetlen részben	`TEIndex`	munka
`MinI`	minimális érték a rendezetlen részben	`TTEIndex`	munka
`I`	index	`TEIndex0`	munka

Struktúradiagram (15)



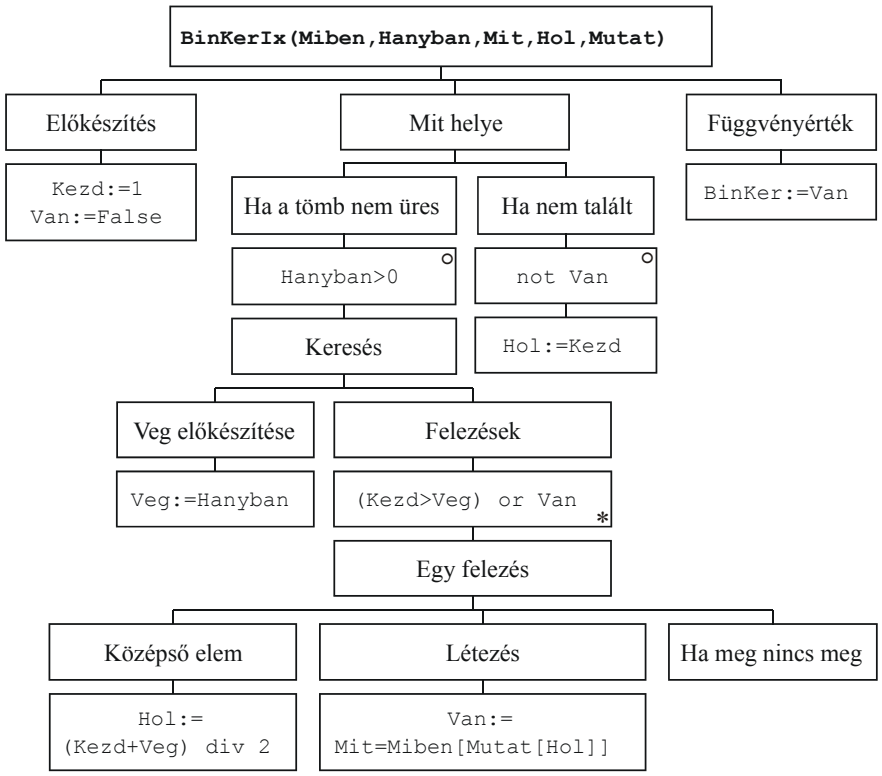
Szubrutin: `uTombRI.KivalRendIx`.

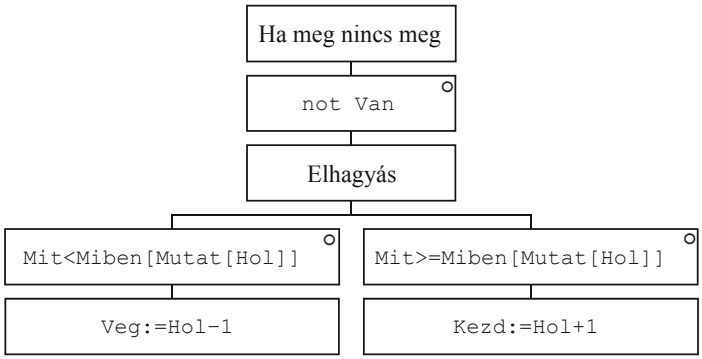
3.2.3.3.2. mintafeladat: Keressünk meg egy adott értéket egy indextáblával növekvően rendezett tömbben, bináris kereséssel!

Adatszerkezet (16)

Azonosító	Funkció	Típus	Jelleg
`Miben`	ebben keresünk	`TSor`	input
`Hanyban`	`Miben` aktuális elemszáma	`TElemDb`	input
`Mit`	a keresett érték	`TTElem`	input
`Hol`	a `Mit` indexének helye a `Mutat`-ban	`TEIndex1`	output
`Mutat`	a mutatótömb	`TEIndexSor`	output
`BinKerIx`	a `Mit` létezése	`Boolean`	output
`Kezd`	aktuális kezdőindex	`TEIndex1`	munka
`Veg`	aktuális végindex	`TEIndex0`	munka
`Van`	a `Mit` létezése	`Boolean`	munka

Struktúradiagram (16)

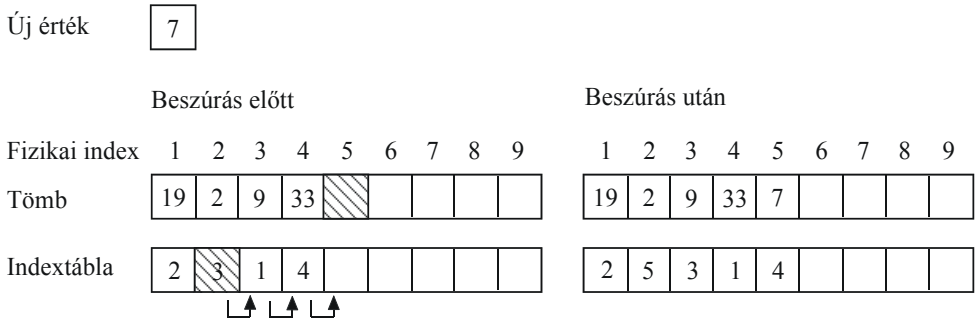




Szubrutin: `uTombRI.BinKerIx`.

3.2.3.3.3. mintafeladat: Szúrjunk be egy adott értéket egy indextáblával növekvően rendezett tömbbe!

Útmutató ♦ A keresést a `BinKerIx` segítségével végezzük. Balra igazított tárolásnak megfelelően a tömb a nagyobb indexek felé bővül. Az adott helytől kezdődően jobbra léptetjük az elemeket (18. ábra).

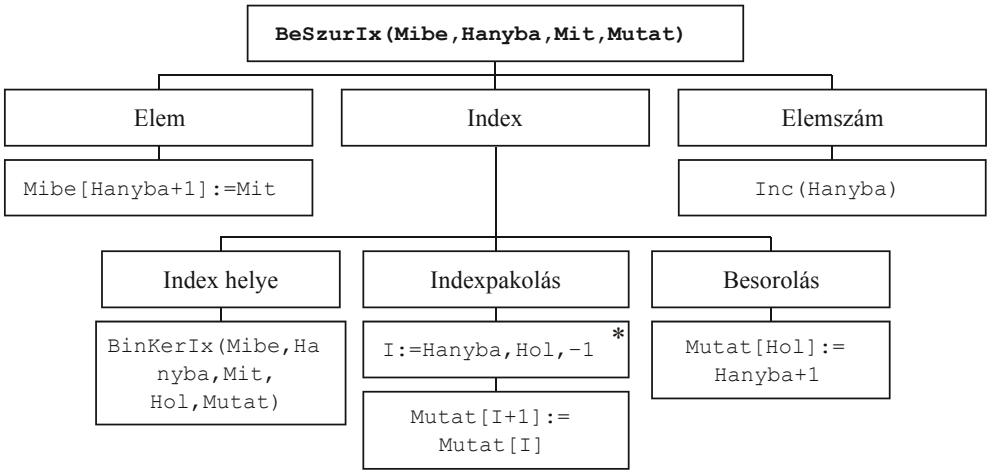


18. ábra. Elem beszúrása egy indextáblával rendezett tömbbe

Adatszerkezet (17)

Azonosító	Funkció	Típus	Jelleg
`Mibe`	ebbe szúrunk be	`TSor`	input
`Hanyba`	a `Mibe` és a `Mutat` aktuális elemszáma	`TElemDb`	input, output
`Mit`	a beszúrandó érték	`TTElem`	input
`Mutat`	a mutatótömb	`TEIndexSor`	input, output

Struktúradiagram (17)



Szubrutin: `uTombRI.BeszurIx`.

3.2.4. Feladatok

A kitűzött feladatok rövidebb leírhatósága céljából vezessünk be néhány elnevezést:

- *Számsoron* egydimenziós tömböt értünk
- Ha *maximális* darabszámot vagy elemszámot nem mondunk, akkor az legyen `100`.
- Ha *elemtípust* nem mondunk, az legyen valós (`Real`), *egész* elemtípuson értsük az `Integer` típust.
- Egyéb elnevezésekre nézve lásd Bevezetés.

1 ♦ Adott egy `S` string. Képezzük azt a stringet, amely az eredetiből úgy keletkezik, hogy:

- A kezdő szóközöket töröljük (`uString.BalVag`)!
- Adott `H` hosszban balra igazítjuk és jobbról szóközökkel feltöltjük (`uString.JobbTolt`)!
- A befejező szóközöket töröljük (`uString.JobbVag`)!
- Adott `H` hosszban jobbra igazítjuk és balról az adott `Tolto` karakterrel feltöltjük (`uString.BalTolt`)!
- Az első nem szóköz jelig balról nullákkal feltöltjük. (`uString.ElolNull`)!

- Minden kisbetű jelét nagybetűre (upcase) cseréljük (`uString.UpSzov`)!
- Minden nagybetű jelét kisbetűre cseréljük (`uString.LoSzov`)!

Megjegyzés az adott megoldásokhoz: a `JobbTolt` és `BalTolt` értékes (nemszóköz) jelet nem vágnak le.

2 ♦ Adottak bizonyos jelek és egy string. Készítsük el az adott jeleknek a stringbeni előfordulási statisztikáját (darabszám jelenként)!

3 ♦ Előírt `K` sorhosszra vonatkoztatva balra igazítottnak nevezünk egy sort, ha a hossza nem nagyobb, mint `K`, valamint a szavak közt egy szóköz van és a sor elején nem áll szóköz. Képezzük egy sor (string) balra igazított formáját, `K` adott előírt sorhosszra! A `K` akkor érvényes, ha vele az igazítás végrehajtható.

4 ♦ Előírt `K` sorhosszra vonatkoztatva jobbra igazítottnak nevezünk egy sort, ha a hossza `K`, valamint a szavak közt egy szóköz van és a sor végén nem áll szóköz. Képezzük egy sor (string) jobbra igazított formáját, `K` adott előírt sorhosszra! A `K` akkor érvényes, ha vele az igazítás végrehajtható.

5 ♦ Előírt sorhosszra vonatkoztatva középre igazítottnak nevezünk egy sort, ha a hossza `K`, valamint szavak közt egy szóköz van és a sor elején és végén a szóközök száma (legfeljebb 1 eltéréssel) azonos. Képezzük egy sor (string) középre igazított formáját, `K` adott előírt sorhosszra! A `K` akkor érvényes, ha vele az igazítás végrehajtható.

6 ♦ Kódoljunk egy stringet úgy, hogy a szomszédos jeleit felcseréljük!

7 ♦ Adott egy `S1` és egy `S2` string. Töröljük ki az `S1`-ből az `S2` első előfordulását (`uString.KiVag`).

8 ♦ Adott `H` hosszhoz és `C` jelhez állítsunk elő egy `H` hosszú és csupa `C` jelből álló stringet (`uString.JelSor`)!

9 ♦ Készítsük el egy stringről a benne előforduló számjegyek statisztikáját (darabszám jelenként)!

10 ♦ Egy stringből gyűjtsük ki a benne szereplő számjegyeket, valamint ezek összegét!

11 ♦ Adott egy string, amelyben zárójelpárok is vannak, helyes szerkezetben (pl. egy helyesen zárójelezett formula). Határozzuk meg a balról első legbelső zárójelpár kezdő- és végzárójelének pozícióját!

12 ♦ Adott egy string, amelyben zárójelpárok is vannak, helyes szerkezetben (pl. egy helyesen zárójelezett formula), de feltételezhetjük, hogy egy zárójelpáron belül már nincs újabb zárójel. Készítsünk egy olyan stringet, amely az eredetiből a zárójeles részek elhagyásával áll elő!

13 ♦ Adott egy string, amelyben zárójelpárok is vannak, helyes szerkezetben (pl. egy helyesen zárójelezett formula). Készítsünk egy olyan stringet, amely az eredetiből a zárójeles részek elhagyásával áll elő!

14 ♦ Adott egy egész számokat tartalmazó tömb. Módosítsuk a tartalmát az alábbiak szerint:

- Egy érték beszúrása adott helyre (`uTomb.Beszur`)!
- Adott indexű elem törlése (`uTomb.Torol`)!
- Egy érték összes előfordulásának törlése (`uTomb.ErtekTorl`)!
- Bármely érték közvetlen egymás utáni ismétlődéseinek törlése (`uTomb.DuplaTorl`)!

15 ♦ Adottak egész számok, és egy eltérési értékhatár. Töröljük a számsorból a sor átlagától az értéknél többel eltérő elemeket!

16 ♦ Adottak valós számok. Számítsunk átlagot egy minimális és egy maximális értéknek az átlagszámításból való elhagyásával!

17 ♦ Kódoljunk egy egész számokból álló számsort úgy, hogy

- az első érték marad változatlan;
- a többi értéket helyettesítsük a megelőző eredeti értéktől való (előjeles) eltérésével!

A dekódolás ennek értelemszerű fordítottja.

Készítsünk szubrutint, amely (paramétertől függően) kódol vagy dekódol!

18 ♦ Az `(x1, y1)`, …, `(xn, yn)` értékpárok egy síkbeli pontrendszer elemeinek koordinátái. A koordináták valós típusú értékek. A pontok száma min. 2. Határozzuk meg:

- A két egymáshoz legközelebb eső pontot!
- A két egymáshoz legtávolabb eső pontot!
- A pontrendszer súlypontját!

19 ♦ Adott stringek rendezett tömbje és egy további `S` string. Meghatározandó, hogy hány elem van a tömbben, amely az adott `S` szövegrésszel kezdődik, és hányadik az első és utolsó ilyen elem! (Ha egyszer sem fordul elő ilyen szó, akkor mindhárom eredményadat legyen 0.)

20 ♦ Adott egy szöveg (stringek tömbje) és egy további `S` string. Meghatározandó egy olyan elem a tömbben, amely az adott `S` szövegrésszel a leghosszabb kezdőrészben megegyezik, valamint a megegyező kezdőrész hossza!

21 ♦ Adott egy névsor, amelyben minden név pontosan 2 részből áll (családnév, keresztnév), a két részt szóközök választják el. Készítendő egy eljárás, amely a név két részét a névsorban végig felcseréli és a két részt pontosan 1 szóközzel választja el, valamint a részek kezdőbetűit nagybetűre, a többi betűt kisbetűre cseréli.

22 ♦ Válogassunk össze két, azonos értelemben (azonos irányban) rendezett tömböt (`uTombR.OsszeVal`)!

23 ♦ Adott egy egész számokat tartalmazó, növekvő tömb. Keressünk egy értéket és a helyét a tömbben. Ha az érték nem létezik, a hely információ akkor is legyen értelmezett és azt az indexet jelentse, ahova az érték bekerülne a sorba.

- Keressünk lineáris (soros) kereséssel (`uTombR.LinKer`)!
- Keressünk bináris (felezéses) kereséssel (`uTombR.BinKer`)!

24 ♦ Egy stringben egy olyan egyszerűsített kifejezés van amelyben:

- számjeggyel kezdődik és végződik;
- a számok csak számjegyek;
- zárójelek nincsenek;
- csak +, − és * műveleti jelek vannak;
- számjegy után csak művelet jöhet;
- művelet után számjegynek kell jönnie.

Ellenőrizzük a kifejezést ilyen szempontból, és ha helyes számítsuk ki az értékét!

25 ♦ Előírt sorhosszra vonatkoztatva kiegyenlítettnek nevezünk egy sort, ha a hossza `K`, valamint ha szóval kezdődik és végződik, és a szavak közti szóközök száma (legfeljebb 1 eltéréssel) azonos. Képezzük egy sor (string) kiegyenlített formáját, `K` adott előírt sorhosszra! A `K` akkor érvényes, ha vele az igazítás végrehajtható.

26 ♦ Kódoljunk egy stringet úgy, hogy adott hosszú részeit részenként megfordítjuk! A részeket a string elejétől kezdjük képezni. Ha a végén egy nem teljes hosszú maradék rész lenne, ez maradjon változatlan.

27 ♦ Adott egy `S1` és egy `S2` string. Töröljük ki az `S1`-ből az `S2` eredeti előfordulásait (vagyis, ha törléssel keletkezett egy újabb `S2` rész, azt már ne töröljük)!

28 ♦ Adott egy `S1` és egy `S2` string. Töröljük ki az `S2`-ből az `S1` előfordulásait (ha törléssel keletkezett egy újabb `S1` rész, azt is töröljük)!

29 ♦ Töröljük egy stringből a jelismétlődéseket!

30 ♦ Adott egy stringben egy lebegőpontos decimális alakú szám (pl. `123.4567E+03`). Írjuk át fixpontos alakra (pl. `123456.7`)! Feltételezhetjük, hogy a kiinduló adatban tizedespont és kitevőrész mindig van, a kitevő `E` betűvel kezdődik, mindig előjeles és kétjegyű valamint azt is, hogy az eredmény is elfér egy stringben. Ha az eredmény ponttal végződne, hagyjuk el a pontot. Ha az eredmény ponttal kezdődne, szúrjunk be elé egy nullát.

31 ♦ Adott egy `Alap` és egy `Keres` string. A `Keres` minden különböző jeléhez megállapítandó, hogy hányszor fordul elő az `Alap` stringben!

32 ♦ Adott egy `Szotar` string, amely különböző szavakat tartalmaz. Tömörítsünk egy olyan stringet, amely nem tartalmaz $ jelet, a következő módszerrel: minden szavát, amely a szótárban is előfordul x jelkombinációval helyettesítjük, ahol x a szó előfordulási sorszáma a `Szotar`-ban (hányadik szó a szótárban).

Oldjuk meg az alábbi feladatokat (feladatonként külön szubrutin):

- String tömörítése.
- Tömörített alak kipakolása.

33 ♦ Ellenőrizzük egy stringként adott formula zárójelezésének helyességét! Feltételezhetjük, hogy csak egyfajta zárójelpár fordul elő benne, a „(", „)" jelpár. Javasolt módszer: A formula zárójelezése helyes, ha a kezdő és végzárójelek száma egyenlő, és a formulában balról jobbra haladva a kezdő zárójelek száma sohasem kisebb a végzárójelek számánál.

34 ♦ Egy tervezett tárgyaláson, résztvevő személyenként egy kezdő- és végidőponttal (mindkettő egész óra a napon belül) adott a szabadidő. A tárgyaláson mindenkinek jelen kell lenni. Határozzuk meg a tárgyalás maximális időtartamát (a kezdő- és végórával)!

35 ♦ Adott egy számsor, amely maximum 100 elemet tartalmaz. Az elemek pozitív valós számok. Bontsuk a sort monoton nem csökkenő részekre! Az egyes részeket elhatárolandó, szúrjunk be nulla elemeket a sorba.

36 ♦ Adottak valós számok. Töröljük a számsorból a sor minimális értékének összes előfordulását!

37 ♦ Válogassunk össze három, azonos értelemben (azonos irányban) rendezett tömböt!

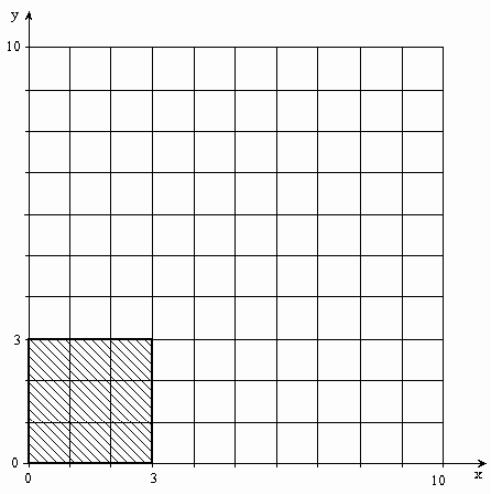
38 ♦ Stringben adott egy nemnegatív egész bináris érték. Állítsuk elő a hexadecimális alakját (`uString.BinbolHexa`)!

39 ♦ Stringben adott egy nemnegatív egész hexadecimális érték. Állítsuk elő a bináris alakját!

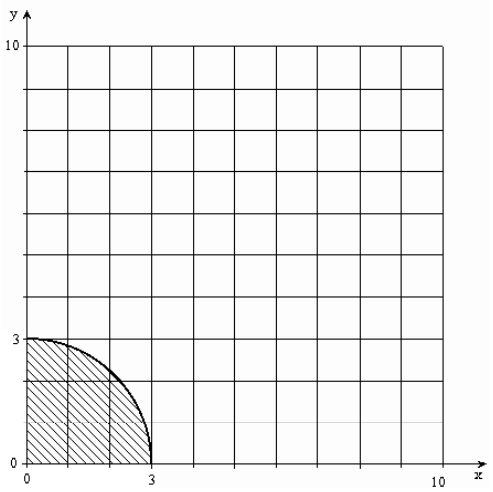
40 ♦ Az `(x1, y1)`, …, `(xn, yn)` értékpárok egy síkbeli pontrendszer elemeinek koordinátái. A koordináták valós típusú értékek. A pontok száma min. 2. Határozzuk meg:

- A pontrendszer egy olyan elemét, amelynek a többi ponttól vett átlagos távolsága minimális!
- A pontoknak az origótól vett távolsági sorrendjét!
- A pontoknak a pontrendszer súlypontjától vett távolsági sorrendjét!

41 ♦ Adott az `N` egész szám, mint a koordinátasíkon a 19. ábra szerint elhelyezkedő négyzetrács oldalhossza. Az `N` értéke az [1, 10] intervallumba eshet, a rajz az `N = 3` esetet mutatja. Input adatként adott még a sorsolással generálandó pontok (koordináta párok) száma, ez maximum 1000 lehet. A pontokat úgy kell generálni, hogy biztosan a négyzetrács területére essenek. A négyzetrács minden négyzetére (az ábrán 9 ilyen van) meghatározandó az, hogy hány darab pont esett rá, és ez a pontok számának hány százaléka. Meghatározandó még a generált pontrendszer súlypontja, valamint ennek távolsága a négyzetrács súlypontjától.



19. ábra. 3 oldalhosszú négyzetrács



20. ábra. 3 egységsugarú negyedkör

42 ♦ Adott az `N` egész szám, mint a koordinátasíkon a 20. ábra szerint elhelyezkedő negyedkör sugara. Az `N` értéke az [1, 10] intervallumba eshet, a rajz az `N` = 3 esetet mutatja. Input adatként adott még a sorsolással generálandó pontok (koordináta párok) száma, ez maximum 1000 lehet. A pontokat úgy kell generálni, hogy biztosan a negyedkör területére essenek. A negyedkör minden körgyűrűjére (a példán 3 ilyen van, a belső körcikket is ennek tekintjük) meghatározandó az, hogy hány darab pont esett rá, és ez a pontok számának hány százaléka.

43 ♦ Adottak a számegyenesen lévő zárt intervallumok, amelyek közt átfedőek is lehetnek, és mind beleesnek a `[0, 1000]` intervallumba. Input adat még a sorsolással előállítandó pontok (valós számok) darabszáma. Ez maximum 1000 lehet. A pontokat úgy kell generálni, hogy biztosan beleessenek a `[0, 1000]` intervallumba. Meghatározandó az egyes intervallumokra vonatkozóan, az intervallum hossza szerint rendezetten:

- a generált pontokból hány esett az intervallumba;
- a generált pontok hány százaléka esett az intervallumba;
- a beeső pontok számának és az intervallum hosszának az aránya.

44 ♦ Adott egy egész számokat tartalmazó növekvő irányban rendezett tömb. Módosítsuk a tartalmát az alábbiak szerint:

- Egy érték beszúrása (`uTombR.BeszurR_L`, `uTombR.BeszurR_B`)!
- Egy érték törlése (`uTombR.TorolR_L`, `uTombR.TorolR_B`)!

45 ♦ Adott egy egész számokat tartalmazó tömb. Rendezzük a tömböt növekvő irányban:

- A szomszédos elemek cseréjével (buborékmódszer) (`uTombR.CserelRend`)!
- A minimális elem kiválasztásával (`uTombR.KivalRend`)!
- A következő elemnek a már rendezett (rész)tömbbe való beszúrásával (`uTombR.BeszurRend`)!

46 ♦ Adott stringek tömbje és egy további `S` string. A tömb minden eleméből töröljük az `S` összes eredeti előfordulását! Az esetlegesen üressé (üres stringgé) vált tömbelemeket töröljük a tömbből is!

47 ♦ Adottak stringek. Soroljuk be őket 3 osztályba, az alábbi szempontok szerint! Egy string

- növekvő, ha jelei növekvő sorrendben vannak;
- csökkenő, ha jelei csökkenő sorrendben vannak;
- konstans ha jelei azonosak;
- az egyéb osztályba tartozik minden más esetben.

Az egyes osztályok tartalmát külön-külön, növekvő módon rendezetten adjuk meg.

48 ♦ Adott egy szöveg (stringek tömbje) és egy további `S` string. Meghatározandó egy olyan elem a tömbben, amely az adott `S` szövegrésszel a legtöbb pozícióban egyezik.

49 ♦ Adottak stringek. Egy string csak betűket és számjegyeket tartalmazhat. Egy string súlyát itt úgy definiáljuk, mint a benne található különböző jelek számát (ha egy jel a stringben többször is előfordulna, a súlyban akkor is csak egyszer veendő figyelembe). Határozzuk meg a stringeknek a súly szerint növekvő sorrendjét!

- Fizikai rendezéssel.
- Indextáblával.

50 ♦ Adottak stringek. Egy string súlyát itt úgy definiáljuk, mint a benne található számjegyek össszegét. Határozzuk meg a stringeknek a súly szerint növekvő sorrendjét!

- Fizikai rendezéssel.
- Indextáblával.

51 ♦ Adott egy `Alap` és egy `Keres` string. Az `Alap`-ban keresünk, a `Keres` tartalmazza a keresendő szavakat:

- Egy szó szigorú értelemben véve bent van az `Alap`-ban, ha ez tartalmazza a keresett szót, mint szót.
- Egy szó tágabb értelemben véve bent van az `Alap`-ban, ha ez tartalmazza a keresett szót, mint szövegrészt.
- Az `Alap` teljesen megfelel a `Keres`-nek, ha a `Keres` minden szava bent van a sorban.
- Az `Alap` részben megfelel a `Keres`-nek, ha a `Keres` legalább egy szava bent van a sorban.
- Az `Alap` nem felel meg a `Keres`-nek, ha sem a részbeni, sem a teljes megfelelés nem áll fenn.

Határozzuk meg `Alap` és a `Keres` viszonyát ebben az értelemben!

52 ♦ Készítsünk szubrutint egész számok érték szerint növekvően rendezett gyakorisági statisztikájának gyűjtésére (`uTbStat.GyakGyujt`)!

53 ♦ Egy stringben egy olyan egyszerűsített kifejezés van amelyre igaz:

- csak előjel nélküli és max. 8 jegyű egész számok és műveleti jelek vannak benne;
- a műveleti jel csak + és – lehet;
- szám után csak művelet, vagy a kifejezés vége jöhet;
- művelet után számnak kell jönnie;
- az első jel számjegy.

Ellenőrizzük a kifejezést ilyen szempontból, és ha helyes, számítsuk ki az értékét!

54 ♦ Adott egy `Kulcs` string és egy `T` string. A `Kulcs` alapján kódoljuk vagy dekódoljuk a `T` stringet! A kódolás módszere:

- a `T` minden egyes jelére
  - meghatározzuk azt a sorszámot, ahol a jel a `Kulcs` szövegben először előfordul;
  - a jelet kicseréljük a sorszámmal egyező kódú jellel.
- a nem kódolható jelek változatlanul maradnak

A dekódolás ennek értelemszerű fordítottja.

Készítsünk szubrutint, amely (paramétertől függően) kódol vagy dekódol!

55 ♦ Adott egy `Kulcs` string és egy `T` string, amelyek mindegyike csak számjegyeket tartalmaz. A `Kulcs` alapján kódoljuk vagy dekódoljuk a `T` stringet. A `Kulcs` rövidebb mint a `T`.

A kódolás módszere: a `T` elejéről kezdve és periódikusan ismételve, a `T` jeleihez „hozzáadjuk" a `Kulcs` jeleit. Két jel összegén itt a két összeadandó jel kódjainak összegéhez mint kódhoz tartozó jelet értjük. A dekódolás ennek értelemszerű fordítottja.

Készítsünk szubrutint, amely (paramétertől függően) kódol vagy dekódol!

56 ♦ Kódoljunk egy stringet úgy, hogy a string elejéről indulva, a növekvő részstringeket (ahol a jelek növekvő sorozatot képeznek) megfordítjuk! A dekódolás ennek értelemszerű fordítottja. Készítsünk szubrutint, amely (paramétertől függően) kódol vagy dekódol!

57 ♦ Adott egy string. Állapítsuk meg, hogy a string tartalma egy periodikusan ismétlődő jelsorozat-e vagy sem! Ha igen, határozzuk meg a minimális és a maximális hosszú periódust!

58 ♦ Adott egy `A` és egy `B` string. Töröljük az `A`-ból azon jelek ismétlődéseit, amelyek a `B`-ben is előfordulnak! (Más megfogalmazással: a feldolgozott `A` nem tartalmazhatja a `B`-vel megadott jelek többszöri egymás utáni előfordulásait).

59 ♦ Tömörítsünk egy olyan stringet, amely nem tartalmaz $ jelet, a következő módszerrel: ha egymás után több mint 3 azonos jel van, akkor ezeket egy $xn jelhármassal helyettesítjük, ahol:

- az x jel az ismételt jel;
- az n jel az x előfordulási számának, mint kódnak megfelelő jel (pl. ha az előfordulási szám 32, akkor a szóköz)!

Oldjuk meg az alábbi feladatokat (feladatonként külön szubrutin):

- String tömörítése.
- Tömörített alak kipakolása.

60 ♦ Előjel nélküli, tízes számrendszerű egész számot (max. 255 jegyűek) stringben tárolunk. Egy jegy a string egy jele, a jegyek sorrendje a csökkenő helyérték szerinti sorrend. Oldjuk meg az alábbi feladatokat (feladatonként külön szubrutin):

- Számítsuk ki két szám összegét, ha ez lehetséges (ha az eredmény is tárolható így)!
- Számítsuk ki két szám különbségét, ha ez lehetséges (ha a kivonandó nem nagyobb, mint a kisebbítendő)!
- Határozzuk meg két szám viszonyát (kisebb, nagyobb, egyenlő)!
- Számítsuk ki két számra az egész osztás hányadosát és maradékát (ismételt kivonással)!
- Számítsuk ki két szám szorzatát (ismételt összeadással), ha ez lehetséges (ha az eredmény is tárolható így)!

61 ♦ Adott egy string. Oldjuk meg az alábbi feladatokat (feladatonként külön szubrutin):
- Számoljuk ki az átlagos szóhosszt!
- Minden szókezdő kisbetűt cseréljünk ki nagyra. Minden, a szó belsejében lévő nagybetűt cseréljünk ki kicsire!
- Fordítsuk meg a szavak sorrendjét!
- Ha egy szó közvetlenül egymás után többször is előfordul, csak egy előfordulását hagyjuk meg!
- Készítsünk egy szójegyzéket, vagyis gyűjtsük ki a különböző szavakat, előfordulási gyakoriságukkal együtt!

62 ♦ Adott az `N`, mint a dolgozók száma, valamint dolgozónként egy kor (egész év) és egy fizetés (egész forint) adat. Adott még az `1 <= KH <= 10` egész szám, amely azt fejezi ki, hogy a minimális dolgozói életkorral indulva hány egymást követő évjárat vonandó össze egy korcsoportba. (A minimális és maximális dolgozói életkor az aktuális adatokból számítandó!). Az utolsó korcsoport lehet kisebb is (pl. ha `Min = 20`, `Max = 62` és `KH = 10`, akkor az első korcsoport a 20..29 éveseké, az utolsó az 59..62 éveseké). Határozzuk meg korcsoportonként és összesen a minimális, maximális és átlagfizetéseket!

63 ♦ Adott egy string. A string tartalma olyan, hogy benne csak számjegyek és szóközök vannak, de egymás után több szóköz is előfordulhat. Számon értünk egy olyan számjegysort, amelyben nincs szóköz, és szóközök vagy a string kezdete vagy vége határolják. Gyűjtsük ki a stringben lévő, maximum 8 jegyű számokat! Ha valamely számnak 8-nál több jegye lenne, bontsuk max. 8 jegyű részekre!

64 ♦ Adottak kifizetendő összegek, pozitív egész számok. Adott a címletek darabszáma, max. 20 és maguk a címletek (pozitív egész számok). Minden összeget pontosan (visszaadás nélkül) és a minimális számú címlettel kell kifizetni. Határozzuk meg a címletjegyzéket (címletenkénti darabszám)!

65 ♦ Válogassunk össze két, azonos értelemben rendezett tömböt úgy, hogy az eredmény csak különböző elemeket tartalmazzon! (A többször előforduló értékeket csak egyszer vegyük figyelembe.)

66 ♦ Adott `N <= 1000` elemszámra és adott értékhatárokra sorsolással állítsunk elő `N` darab, a határok közé eső egész számot és számoljuk ki az alábbi statisztikai jellemzőket:
- minimum, maximum, átlag, szórás,
- medián (a rendezett számsorban a középső indexnél álló érték).

Az eredményt csak a jellemzők képezik (magára a számsorra, mint eredményre nincs szükség).

67 ♦ Adottak egy évre vonatkozóan string típusú `HH.NN.OO` (hónap, nap, óra) formátumú időpontok, és hozzájuk tartozó egész értékű hőmérséklet adatok. Az adatok száma maximum 3000.

Határozzuk meg naponta a napi és havonta a havi átlaghőmérséklet adatokat! Jelezzük vissza azt is, hogyha valamely napra/hónapra nincs adat!

68 ♦ Adottak a számegyenesen lévő zárt intervallumok. Egy intervallumot a kezdő és végpontjával adunk meg. A kezdő és végpontok valós típusú számok, de beleesnek a `[0, 1000]` intervallumba. Az intervallumok közt átfedőek is lehetnek. Input adat még a sorsolással előállítandó pontok (valós számok) darabszáma. Ez maximum 1000 lehet. A pontokat úgy kell generálni, hogy biztosan beleessenek a `[0, 1000]` intervallumba. Meghatározandó az egyes intervallumokra vonatkozóan:

- a generált pontokból hány esett az intervallumba;
- a generált pontok hány százaléka esett az intervallumba;
- a beeső pontok számának és az intervallum hosszának az aránya.

69 ♦ Személyenként adottak életkor (egész év) és testsúly (egész kg) adatok. Meghatározandók életkoronként és összesen az átlagsúlyok (egészre kerekítve)! A kiinduló évadatok intervalluma `0..100`, a súlyadatoké `0..250`.

70 ♦ Adottak nemnegatív egész számok, maximum négyjegyűek. Állítsuk elő érték szerint növekvő sorrendben a számok kettes, nyolcas és tizenhatos számrendszerbeli alakját!

71 ♦ Adottak a $[-9999, 9999]$ intervallumba eső számok. Állítsuk elő érték szerint növekvő sorrendben a számok kettes és tizenhatos számrendszerbeli alakját. A bináris és hexadecimális alakok (számrendszerenként) azonos hosszúak legyenek, az előjel feltüntetésével és a leghosszabb számnak megfelelően előnullázva.

72 ♦ Az `(x1, y1), ..., (xn, yn)` értékpárok egy síkbeli pontrendszer elemeinek koordinátái. A koordináták `Byte` típusú értékek. A pontok száma min. 2, és páros szám. Határozzuk meg a pontrendszer olyan két pontját, amelyek által meghatározott egyenes két azonos elemszámú részre osztja a pontrendszert (az egyenesre ráeső pontok bármelyik részbe számíthatók)!

73 ♦ Az `(x1, y1), ..., (xn, yn)` értékpárok egy síkbeli pontrendszer elemeinek koordinátái. A koordináták valós típusú értékek. Alakítsuk át a pontrendszert a következő módon:

- csak különböző pontok szerepeljenek;
- minden pontnak a szimmetrikus párja is szerepeljen.

Egy pont szimmetrikus párján a koordináták felcserélésével kapott pontot értjük pl. `(3, 4)` és `(4, 3)`. Ha a két koordináta azonos, pl. `(3, 3)`, akkor nincs ilyen pár.

74 ♦ Polinomokat a fokszámmal (pozitív egész szám, max. 20) és csökkenő kitevő szerinti sorrendben adott együtthatókkal adunk meg. Oldjuk meg az alábbi feladatokat (feladatonként külön szubrutin):

- Határozzuk meg egy polinom deriváltját!
- Állítsuk elő egy polinom határozatlan integrálját!

- Állítsuk elő két polinom összegét és különbségét!
- Állítsuk elő egy legalább másodfokú polinomot egy adott pontban érintő egyenest (mint elsőfokú polinomot)!
- Számítsuk ki egy polinom határozatlan integrálját egy adott intervallumon!

75 ♦ Egy időszakban (max. 100 nap) minden napon naponta többször (minimum kétszer és maximum ötször) megmérik a levegő hőmérsékletét. Egy mérési adat két részből áll:

- a nap sorszáma az időszakon belül,
- a hőmérséklet (valós szám).

A kapott adatsorból számítsuk ki:

- Az átlaghőmérsékletet naponta!
- Az átlaghőmérsékletet a teljes időszakra vonatkoztatva!
- A mért minimum és maximum hőmérsékletet a teljes időszakra vonatkoztatva (a napjával együtt)!

76 ♦ Adott egy osztály (max. 50 fő), minden tanulóhoz két adat tartozik:

- név,
- érdemjegy (1..5).

Rendezzük át az adatokat csökkenő (pontosabban: nem növekvő) érdemjegy szerint úgy, hogy az azonos érdemjegyűek ezen belül ábécé sorrendben legyenek!

77 ♦ Adott egy egész számokat tartalmazó növekvő irányban, indextáblával rendezett tömb. Módosítsuk a tartalmát az alábbiak szerint:

- Egy érték beszúrása (`uTombRI.BeszurIx`)!
- Egy érték törlése!

78 ♦ Adott egy egész számokat tartalmazó tömb. Rendezzük a tömböt indextáblával növekvő irányban:

- A szomszédos elemek cseréjével (buborékmódszer) (`uTombRI.CserelRendIx`)!
- A minimális elem kiválasztásával (`uTombRI.KivalRendIx`)!
- A következő elemnek a már rendezett (rész)tömbbe való beszúrásával!

79 ♦ Adott egy egész számokat tartalmazó, indextáblával növekvő irányban rendezett tömb. Keressünk egy értéket és a helyét a tömbben. Ha az érték nem létezik, a hely információ akkor is legyen értelmezett és azt az indexet jelentse, ahova az érték bekerülne a sorba.

- Keressünk lineáris (soros) kereséssel!
- Keressünk bináris (felezéses) kereséssel (`uTombRI.BinKerIx`)!

80 ♦ Adott egy tömb, amely elemenként egy DOS fájlazonosítót (név, kiterjesztés) tartalmaz. Állítsuk elő a név és a kiterjesztés szerinti rendezettséget leíró indextáblákat!

81 ♦ Egy tanulócsoportban ismerjük a neveket és a tanulmányi átlageredményeket. Állítsuk elő a név és a tanulmányi átlag szerinti rendezettséget leíró indextáblákat!

82 ♦ Előírt `K` sorhosszra vonatkoztatva balra, jobbra vagy középre igazítottnak nevezünk egy bekezdést, ha a sorok egyenként is ilyenek (lásd 3., 4., 5. feladatok), és maximálisan kitöltöttek, vagyis nincs olyan sor, amelynek első szavát az igazítási szabály megsértése nélkül át lehetne vinni a megelőző sorba. Oldjuk meg az alábbi feladatokat (feladatonként külön szubrutin):

- Állítsuk elő egy adott bekezdés balra igazított formáját, adott előírt sorhossz mellett, ha ez lehetséges!
- Állítsuk elő egy adott bekezdés jobbra igazított formáját, adott előírt sorhossz mellett, ha ez lehetséges!
- Állítsuk elő egy adott bekezdés középre igazított formáját, adott előírt sorhossz mellett, ha ez lehetséges!

83 ♦ Előírt `K` sorhosszra vonatkoztatva kiegyenlítettnek nevezünk egy bekezdést, ha a sorok egyenként is ilyenek (lásd 25. feladat), kivéve az utolsó sort, amely balra igazított (lásd 3. feladat) és maximálisan kitöltöttek, vagyis nincs olyan sor, amelynek első szavát az igazítási szabály megsértése nélkül át lehetne vinni a megelőző sorba.

Állítsuk elő egy adott bekezdés kiegyenlített formáját, adott előírt sorhossz mellett, ha ez lehetséges!

84 ♦ Szövegillesztés: két stringet összehasonlítva keressük meg a maximális hosszú, egymáshoz legjobban illeszkedő (legtöbb jelben megegyező) két szakaszt!

85 ♦ Adott egy `S1` és egy `S2` string, az `S1` nem lehet hosszabb, mint az `S2`. Keressük, hogy az `S2` mely részével egyezik legtöbb helyen az `S1`! Az eredményt a legtöbb helyen egyező rész kezdőpozíciója és az egyező helyek száma jelenti. A vizsgálatot balról jobbra haladva végezzük el!

86 ♦ Azt mondjuk, hogy az `X` szó `Y`-ra javítható, ha az alábbi esetek valamelyike fennáll:

- a két szó ugyanolyan hosszú és csak egy pozícióban különbözik;
- az `Y` egy jellel hosszabb, mint az `X`, de van az `Y`-nak olyan jele, amelyet belőle elhagyva megkapjuk az `X`-et;
- az `X` egy jellel hosszabb, mint az `Y`, de van az `X`-nek olyan jele, amelyet belőle elhagyva megkapjuk az `Y`-t.

Adott a helyes szavak egy sorozata (szótár) és egy keresett `X` szó. Állapítsuk meg, hogy az `X` bent van-e a szótárban és ha nincs, akkor melyek a szótár azon szavai, amelyekre `X` javítható!

A szótár is input adat, legfeljebb 100 szót tartalmazhat. Az eredmény az `X` helyessége és a javító szavak.

87 ♦ Adott egy Pascal halmazváltozó, illetve egy string, amely egy Pascal halmazkonstanst tartalmaz. Készítsük el a kettő közti konvertáló szubrutinokat:

- A változó értékét a stringbe, úgy, hogy a string a minimális hosszú legyen!
- A stringben megadott értéket a változóba!

Oldjuk meg a feladatot `Byte` és `Char` elemtípusú halmazokra!

88 ♦ Adott egy szöveg, valamint egy `X` és egy `Y` szó. A szöveg minden sorában végrehajtandó az `X` eredeti előfordulásainak lecserélése `Y`-ra, egy további paramétertől függően kétféle módon:

- Az `X`-nek csak olyan előfordulásait cseréljük, ahol az `X` szóként fordul elő!
- Az `X`-nek minden előfordulását lecseréljük, tehát azokat is, ahol az `X` egy más szó részeként fordul elő!

A cseréket soronként hajtsuk végre! Feltételezzük, hogy a módosított sor is elfér az eredeti helyén (a tömbelemben).

89 ♦ Adott egy `Szotar` string, amely különböző szavakat tartalmaz és egy szöveg (szövegsorok tömbje). Tömörítsük a szöveget soronként a 32. feladat módszerével!

Oldjuk meg az alábbi feladatokat (feladatonként külön szubrutin):

- Szöveg tömörítése.
- Tömörített alak kipakolása.

90 ♦ Ismert a következő módszer prímszámok meghatározására (Eratosztenész rostája):

a) írjuk fel a természetes számokat 1-től N-ig;
b) húzzuk ki az 1-et;
c) jelöljük meg a következő még nem jelölt és ki nem húzott számot és húzzuk ki az összes többszöröseit;
d) ismételjük a *c)* lépést, amíg van megjelölhető szám!

A megjelölt számok az [1, N] intervallum prímszámai.

Adott `1 <= A < B <= N <= 10000` egész számokra állapítsuk meg ezzel a módszerrel az [A, B] intervallum prímszámait!

91 ♦ Adott az, hogy az év első napja milyen napra esett, valamint az év hivatalos munkaszüneti napjai (hó, nap) nap formában, ezek száma max. 20 lehet. Ezeken kívül a szombatokat és vasárnapokat eleve munkaszüneti napnak vesszük. Adott dátumra (hó, nap) meghatározandó, hogy milyen napra esik, valamint ha munkanap, akkor az év hányadik munkanapja! Feltételezhetjük, hogy nem szökőévről van szó.

92 ♦ Adottak a számegyenesen lévő zárt intervallumok. Egy intervallumot a kezdő- és végpontjával adunk meg. A kezdő- és végpontok egész számok. Oldjuk meg az alábbi feladatokat (feladatonként külön szubrutin):

- Ellenőrizendő, hogy az intervallumrendszer tagjai páronként idegenek-e (nincs két olyan intervallum, amelynek lenne közös pontja). Ha ez nem áll fenn, meghatározandó egy olyan intervallum, amely a legtöbb más intervallummal van átfedésben.
- Minimalizáljuk az intervallumrendszer tagjainak számát az átfedések összevonásával!
- Határozzuk meg az intervallumrendszer által lefedett egész számok darabszámát! (Lefedett szám, amelyik legalább egy intervallumba beleesik).
- Határozzuk meg az intervallumrendszer metszetét, mint intervallumot!

93 ♦ Az `(x1, y1)`, ..., `(xn, yn)` valós értékpárok egy síkbeli, `N` csúcspontú konvex sokszög csúcspontjainak az óramutató járásával ellenkező irányú körüljárási sorrend szerinti koordinátái. Számítsuk ki a sokszög kerületét és területét (a terület kiszámításához egy belső pontból, pl. a súlypontból, bontsuk háromszögekre a sokszöget)!

94 ♦ Az `(x1, y1)`, ..., `(xn, yn)` valós értékpárok egy síkbeli, `N` csúcspontú konvex sokszög csúcspontjainak az óramutató járásával ellenkező irányú körüljárási sorrend szerinti koordinátái. Adott még egy további `(x, y)` pont. Határozzuk meg, hogy ez a pont a sokszög belső pontja-e! Módszer: ha a pont nem belső, akkor van olyan csúcspont, amellyel összekötő egyenes a sokszög valamely oldalát az oldal belső (nem csúcs) pontjában (is) metszi.

95 ♦ Az `(x1, y1)`, ..., `(xn, yn)` valós értékpárok egy síkbeli, `N` csúcspontú sokszög csúcspontjainak az óramutató járásával ellenkező irányú körüljárási sorrend szerinti koordinátái. Határozzuk meg, hogy a sokszög konvex-e! Módszer: ha a sokszög nem konvex, akkor van olyan átló, amelynek az egyenese a sokszög valamely oldalát az oldal belső (nem csúcs) pontjában (is) metszi.

96 ♦ Az `(x1, y1)`, ..., `(xn, yn)` értékpárok egy síkbeli pontrendszer elemeinek koordinátái. A koordináták `Byte` típusú értékek. A pontok száma előírt, min. 2.

Állítsunk elő sorsolással egy olyan pontrendszert, amely nem tartalmaz ismétlődéseket (egy pont legfeljebb egy példányban szerepeljen), és rendezzük át a pontokat az alábbi módszer szerint:

- egy sorsolással kiválasztott pont legyen az első;
- az elsőhöz legközelebbi legyen a második;
- a másodikhoz legközelebb eső legyen a harmadik;
- és így tovább!

97 ♦ Adottak valós számok. Határozzuk meg a nagyság szerinti középső elem indexét és értékét! (Ha páros számú eleme van, akkor a kisebbik indexű középső elemet vegyük.)

98 ♦ Adottak egész számok. Meghatározandó az adatok `Min` minimuma, `Max` maximuma, statisztikai átlaga és szórása, valamint az, hogy a `[Min, Max]` intervallu

mot 4 egyenlő részre osztva, az adatok hány százaléka esik az egyes részintervallumokba!

99 ♦ Adottak stringek. Egy string bármilyen jeleket tartalmazhat. Egy string súlyát itt úgy definiáljuk, mint a leghosszabb olyan szeletének (összefüggő részének) hosszát, amelyen belül a jelek monoton növekvő sorrendben vannak. Határozzuk meg a stringek súly szerinti növekvő sorrendjét:

- Fizikai rendezéssel!
- Indextáblával!

100 ♦ Adottak számpárok. Egy számpár két db kétjegyű egész számból áll. A párok egy láncán értjük ezek egy olyan sorrendjét, amelyben minden szomszédos számpárra igaz az, hogy a megelőző második eleme azonos a következő első elemével. Állapítsuk meg, hogy az első párral indulva képezhető-e egyértelműen lánc az adatokból! Ha igen, állítsuk elő a láncot is!

101 ♦ Tömörítsük a szöveget (szövegsorok tömbje), amely nem tartalmaz $ jelet, soronként a következő módszerrel:

- Kigyűjtjük a különböző szavakat a szövegben való előfordulási gyakoriságukkal együtt. Egy szó helyfoglalásán a szó hosszának és előfordulási gyakoriságának szorzatát értjük.
- Megállapítjuk a szavak csökkenő helyfoglalási sorrendjét, és ebben a sorrendben hozzájuk rendelünk x helyettesítő jelkombinációkat, mint kódokat, ahol x a szó sorszáma a csökkenő a sorrend szerint.

Oldjuk meg az alábbi feladatokat (feladatonként külön szubrutin):

- Szöveg tömörítése.
- Tömörített alak kipakolása.

102 ♦ Adottak a `0 < K <= N <= 9` pozitív egész számok. Állítsuk elő az `N` elem `K`-ad osztályú kombinációit, vagyis az `N` elem összes különböző, `K` elemű részhalmazait úgy, hogy meghatározzuk az összes olyan `N` jegyű bináris számot, amely pontosan K darab `1`-es jegyet tartalmaz. Egy ilyen szám megfelel egy részhalmaznak, olyan értelemben, hogy az összes elem közül az 1-es értékű jegyek jelölik ki a részhalmaz elemeit.

A számokat tömbben vagy stringben állítsuk elő (a jegyek az elemek) úgy, hogy kiindulva a legkisebb megfelelő értékből (`K` darab egyes előtt `N - K` nulla van), a bináris összeadás szabályait alkalmazva elszámlálunk egyesével a legnagyobb megfelelő értékig (`K` darab egyes után `N - K` nulla van), és közben gyűjtjük a megfelelő értékeket.

103 ♦ Adottak stringek és a `K` mint kezdőszelet hossz. Meghatározandó a stringek egy olyan csoportja, amelyen belül az elemek egymással `K` hosszú kezdőszeletben megegyeznek és a csoport elemszáma maximális.

3.3. Mátrixok

3.3.1. Alapfeladatok

A többdimenziós tömbök kezelése lényegében az egydimenzióban megismert módszerek, eljárások értelemszerű kiterjesztésével, alkalmazásával történhet. Erre adunk néhány mintapéldát a kétdimenziós tömbök, vagyis a mátrixok esetére. A modellek adatstruktúráinak közös konstansai és típusai:

```
const
  SorMaxDb=50;
  OszlMaxDb=50;  {maximális sor és oszlopszám}
type
  TSInd=1..SorMaxDb;
  TSInd0=0..SorMaxDb;  {sorindexek}
  TOInd=1..OszlMaxDb;
  TOInd0=0..OszlMaxDb;  {oszlopindexek}
  TSorDb=0..SorMaxDb;  {sorszám}
  TOszlDb=0..OszlMaxDb;  {oszlopszám}
  TMElem=Integer;  {mátrixelem}
  TMElemOssz=Longint;  {mátrixelemek összege}

  TMatrix=array[TSInd, TOInd] of TMElem;  {mátrix}
  TMatSor=array[TSInd] of TMElem;  {mátrixsor}
  TMatOszl=array[TOInd] of TMElem;  {mátrixoszlop}
  TOsszSor=array[TOInd] of TMElemOssz;  {összegsor}
  TOsszOszl=array[TSInd] of TMElemOssz;  {összegoszlop}
```

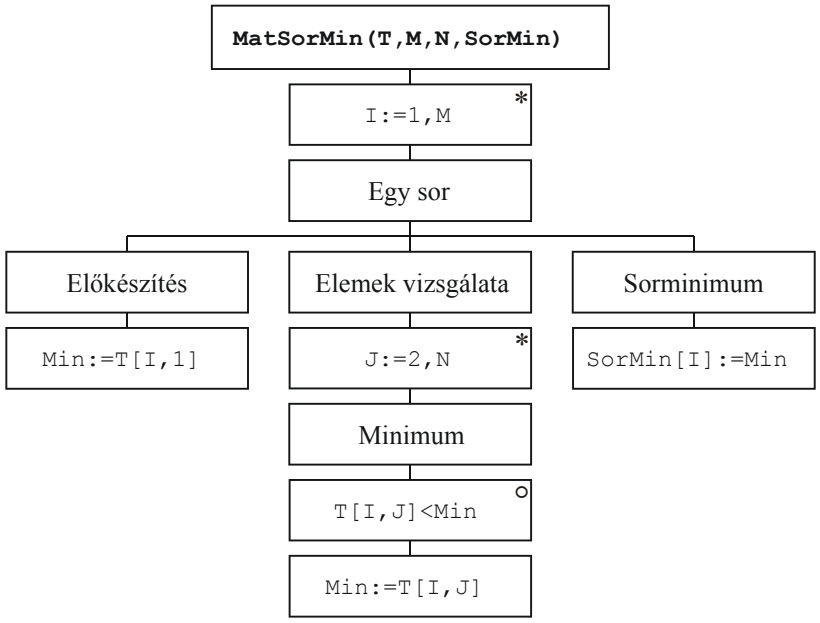
3.3.1.1. mintafeladat: Határozzuk meg egy mátrix sorminimumait!

Útmutató ♦ A soronként kiválasztott minimumértékek egy mátrixoszlop típusú tömböt alkotnak.

Adatszerkezet (18)

Azonosító	Funkció	Típus	Jelleg
T	mátrix	TMatrix	input
M	sorok aktuális száma	TSInd	input
N	oszlopok aktuális száma	TOInd	input
SorMin	a sorminimumok	TMatOszl	output
I	sorindex	TSInd	munka
J	oszlopindex	TOInd	munka
Min	egy sor minimuma	TMElem	munka

Struktúradiagram (18)



Szubrutin: `uMatrix.MatSorMin`.

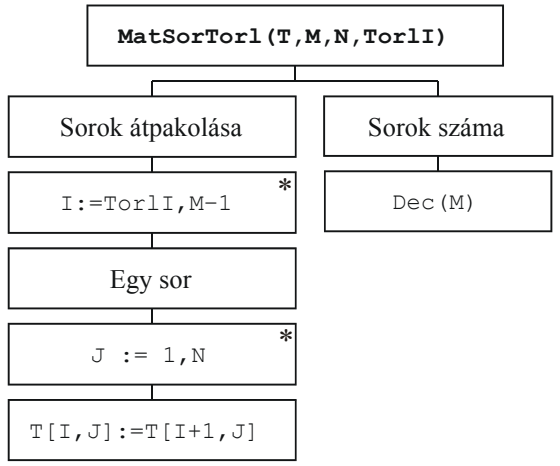
3.3.1.2. mintafeladat: Töröljük egy mátrix egy adott sorszámú sorát!

Útmutató ♦ Az egydimenziós technikát alkalmazzuk a sorokra.

Adatszerkezet (19)

Azonosító	Funkció	Típus	Jelleg
`T`	mátrix	`TMatrix`	input, output
`M`	sorok aktuális száma	`TSInd`	input, output
`N`	oszlopok aktuális száma	`TOInd`	input
`TorlI`	a törlendő sor indexe	`TSInd`	input
`I`	sorindex	`TSInd0`	munka
`J`	oszlopindex	`TOInd`	munka

Struktúradiagram (19)



Szubrutin: `uMatrix.MatSorTorl`.

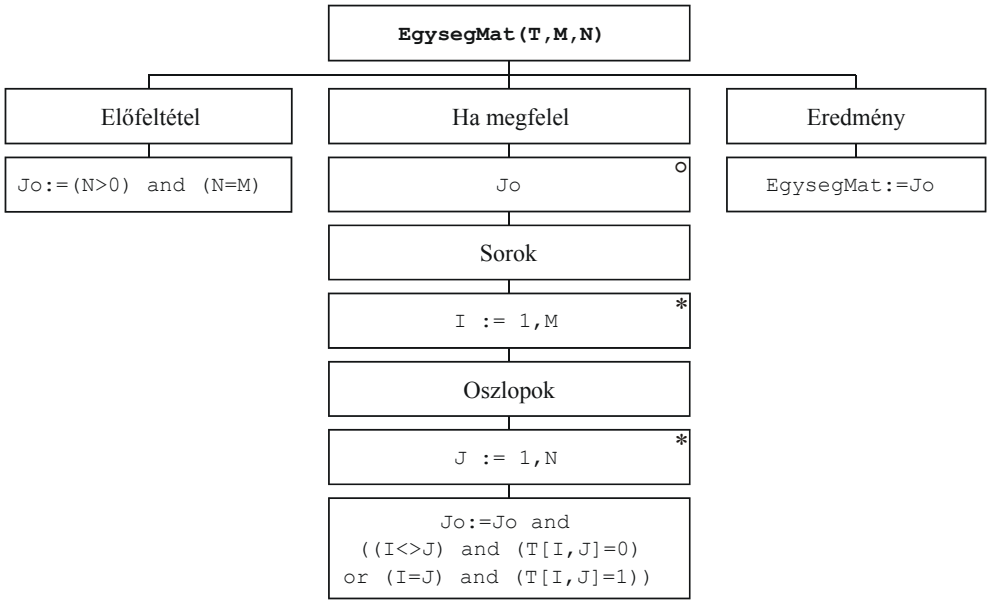
3.3.1.3. mintafeladat: Állapítsuk meg, hogy egy mátrix egységmátrix-e!

Útmutató ♦ Az egységmátrix négyzetes, a főátlójában csak `1`, a többi helyén csak `0` értékű elemek vannak.

Adatszerkezet (20)

Azonosító	Funkció	Típus	Jelleg
`T`	mátrix	`TMatrix`	input
`M`	sorok aktuális száma	`TSInd`	input
`N`	oszlopok aktuális száma	`TOInd`	input
`EgysegMat`	egységmátrix-e	`Boolean`	output
`I`	sorindex	`TSInd`	munka
`J`	oszlopindex	`TOInd`	munka
`Jo`	egységmátrix-e	`Boolean`	munka

Struktúradiagram (20)



Szubrutin: `uMatrix.EgysegMat`.

3.3.2. Feladatok

A kitűzött feladatok rövidebb leírhatósága céljából: ha a feladatból nem határozható meg a mátrix valamely jellemzője, akkor pótoljuk ezt az előző mintafeladatokhoz használt deklarációk szerint.

1 ♦ Ellenőrizzük, egy adott, valós elemtípusú mátrix lehet-e távolságmátrix! Ennek feltételei:

- a főátlóban csupa 0 van;
- a mátrix szimmetrikus és elemei nemnegatívak.

2 ♦ Szimmetrizáljunk egy valós számokból álló négyzetes mátrixot, úgy, hogy a főátlóra szimmetrikusan elhelyezkedő elemek átlagát írjuk mindkét elem helyébe!

3 ♦ Adott egy egész számokból álló mátrix. Oldjuk meg az alábbi feladatokat (feladatonként külön szubrutin):

- Mátrix minimumhelye (`uMatrix.MatMin`).
- Sorminimumok tömbje (`uMatrix.MatSorMin`).
- Sor-, oszlop- és teljes összegek (`uMatrix.MatSorMin`).
- Adott indexű sor törlése (`uMatrix.MatSorTorl`).
- Mátrix transzponálása (`uMatrix.MatTranszp`).
- Két mátrix algebrai szorzása (`uMatrix.MatSzor`).
- Az egységmátrix tulajdonság ellenőrzése (`uMatrix.EgysegMat`).

4 ♦ Egy adott mátrixra és sorindexre határozzuk meg a mátrix ezen sor szerinti növekvő rendezettségét (vagyis azt, hogy milyen sorrendbe kellene szedni az oszlopokat, hogy a megjelölt sor elemei sorindex szerint növekvően legyenek), indextáblával!

5 ♦ Keressük egy adott mátrix első olyan sorát, amely megegyezik egy adott egydimenziós tömbbel!

6 ♦ Adott egy valós számokból álló mátrix. Oldjuk meg az alábbi feladatokat (feladatonként külön szubrutin):

- Töröljük a nulla összegű oszlopokat!
- Töröljük a csak nullát tartalmazó sorokat!
- Szúrjuk be első sorként az eredeti oszlopösszegeket tartalmazó sort!
- Szúrjuk be első oszlopként az eredeti sorösszegeket tartalmazó oszlopot!
- Töröljük a legkisebb összegű oszlopot!
- Emeljük ki minden sorban a minimális értéket a sor elejére!
- Minden sort osszunk le a sor maximális abszolútértékű elemével, ha ez lehetséges!

7 ♦ Egy négyzetes mátrixot „bűvös négyzetnek” nevezünk, ha a sorok és oszlopok valamint az átlók összege ugyanaz a szám. Ellenőrizzük, hogy egy adott mátrix bűvös négyzet-e!

8 ♦ Az `A` és a `B` egész elemű mátrixok. Ellenőrizzük, hogy az `A*B` algebrai mátrixszorzat eredménye egységmátrix-e!

9 ♦ Adott egy mátrix, amely elemenként egy ékezet nélküli nagybetűt tartalmaz. Rendezzük át a sorait úgy, hogy a soronkénti összeolvasással kapott stringek ábécé sorrendben legyenek!

10 ♦ Adott egy mátrix, amely elemenként egy valós értéket tartalmaz. Rendezzük át a sorait növekvő sorösszeg szerint!

11 ♦ Adott `N` és `M` mint egy stringtáblázat sorainak ill. oszlopainak száma. Az `N` és `M` értéke min. 2, max. 10 lehet.

Sorsolással töltsük fel a táblázatot szám- és szövegoszlopokkal! Szám táblázatelemen azt értjük, hogy az elem egy csak számjegyet tartalmazó és nem 0-val kezdődő string. Szöveg táblázatelemen azt értjük, hogy az elem egy csak kisbetűket tartalmazó string. Azt, hogy melyik oszlopba kerüljenek csak számok, illetve csak szövegek, kisorsolandó. A táblázatelemek hossza min. 1 és max. 5 lehet, de ezen belül szintén kisorsolandó.

A feltöltött táblázatot formázzuk az alábbi módon:

- A számoszlopok elemei feltöltendők balról, a maximális hosszú oszlopelem hosszára.
- A szövegoszlopok elemei feltöltendők jobbról, a maximális hosszú oszlopelem hosszára.

A töltőkarakter mindkét esetben a „*” jel.

12 ♦ Adott egy mátrix, amely számokat tartalmaz. Töröljünk a mátrixból minden olyan sort, amely valamely másik sortól csak egy szorzófaktorban különbözik, vagyis a sor egy másikból egy konstanssal való szorzással előállítható! A sorok ellenőrzésének sorrendje bármilyen lehet.

13 ♦ Egy mátrixban az ötös lottó húzási eredményei vannak valahány (max. 10) hétre, egy sor egy heti húzás, tehát a mátrixnak 5 oszlopa van, az elemek 1..90 értékű számok.

Egy fogadó hetente azonos számú (max. 20), és azonos tippekkel játszik. Ezeket egy az előzőhöz hasonló szerkezetű mátrixban adjuk meg (5 oszlop, egy sor egy tipp). Határozzuk meg, hetente és találatszámonként (1..5) hogy a fogadónak hány találata van!

4. HALMAZOK

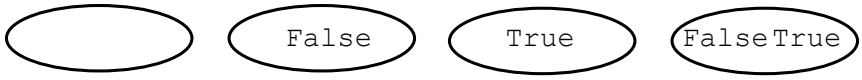
4.1. Általános jellemzés

Az általános halmazfogalom a matematikában alapfogalom. Ha mégis definiálni akarnánk, mint bármilyen típusú elemek bármilyen számú illetve véges vagy végtelen számosságú összességét írnánk le.

A számítástechnikai modellekben használatos halmazfogalom ennél természetesen jóval szűkebb, egyrészt eleve csak véges lehet, másrészt

- *egy halmazban csak azonos típusú elemek lehetnek,*
- *az elemtípus nem lehet akármilyen,*
- *az elemszám korlátozott.*

Az adat és az adatcsoport viszonya a halmaz esetén a lehető legegyszerűbb*: egy érték vagy eleme a halmaznak, vagy nem.* Egyéb jellemzők, mint pl. a tömbnél sorszám, sorrend, rendezettség, itt nincsenek. A halmazadat (konstans vagy változó) egy értéke az elemtípus értékkészletének egy részhalmaza. Például a logikai elemtípusú halmazváltozó értékkészlete négyelemű (21. ábra), az a halmazváltozó, amelynek elemtípusa az `1..5` intervallumtípus, összesen `32` különböző értéket vehet fel (22. ábra). Mint a példákból is látjuk az üres halmaz minden halmaztípus értékkészletében bent van.



`set of Boolean`

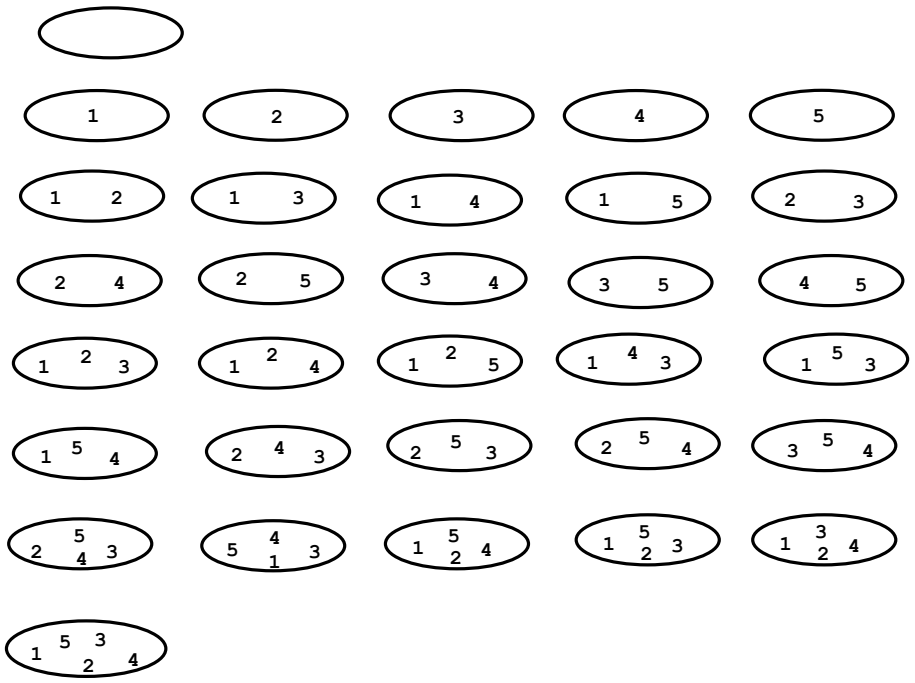
21. ábra. Logikai elemtípusú halmazváltozó értékkészlete

Halmazt akkor használunk, ha a feladat megoldásánál elsősorban olyan kérdésekre kell válaszolnunk, hogy

- egy érték bent van-e az adatcsoportban;
- két adatcsoport viszonya milyen (pl. vannak-e közös elemek, melyek a közös elemek),

és kevésbé érdekes, pl. az értékek darabszáma, sorrendje. Az egyszerűbb feladatok közül tipikusan ilyenek, pl. a sorsolási (véletlenszerű kiválasztási) és bizonyos adatellenőrzési feladatok, de matematikai-operációkutatási problémák

jelentős részének is a véges halmazok segítségével van megfogalmazva az elvi megoldó algoritmusa.



22. ábra. Az 1..5 elemtípusú halmazváltozó értékkészlete

A gyakorlatban is elterjedt programnyelvek között a Pascal szinte egyedül áll abban, hogy *standard* típusként is ismeri a *halmaz* adatstruktúrát. Ez amellett, hogy még teljesebbé, univerzálisabbá teszi a nyelvet általában is, oktatási és publikációs szempontból kiváltképpen szerencsés. A Pascal halmaztípusainál az elemtípus olyan megszámlálható típus lehet, amelynek értékkészlete max. `256` elemet tartalmaz, és számok esetén csak a `0..255` intervallumban van, tehát lehet:

- `Boolean`, `Byte`, `Char`;
- max. `256` elemű felsorolt típus;
- az előzőek intervallumtípusa.

A halmaztípus *alaptípusán* az elemtípusnak megfelelő standard (`Boolean`, `Byte`, `Char`), vagy felsorolt alaptípust értjük.

A Pascal halmaz tárolása „bittérkép”-szerűen valósul meg. A tárterületen minden lehetséges elemnek egy bit felel meg. Ennek értéke jelzi, hogy az elem aktuálisan bent van (=1) vagy nincs bent (=0) a halmazban. Így egy halmazváltozó max. 32 bájtot (256 bitet) foglal le. A ténylegesen lefoglalt tárterület nagysága a konkrét elemtípustól függ, az intervallumtípusoknál ugyanis az alaptípus biztosan 0 értékű (az intervallumba nem tartozó) bitjeinek egy része nem tárolódik. A pontos szabály:

Az alaptípusú halmaznak megfelelő tárterületen elhelyezzük a halmazt, és az általa lefedett területet mindkét oldalon bájthatárra egészítjük ki. Az így lefoglalt bájtok száma a változó helyigénye.

Például legyen egy halmaz elemtípusa a 10..19 intervallum. Ekkor az alaptípusa a Byte. Az elemtípus elhelyezkedése az alaptípusban a 23. ábrán látható.

0....7	8..9..10..							......18..19									
0	1							2								.	31
	0	1	2	3	4	5	6	7	0	1	2	3	4	5	6	7	.
			*	*	*	*	*	*	*	*	*	*					
	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	-	
	0	0	1	0	1	0	0	0	0	0	1	0	0	0	0	0	

23. ábra. A 10..19 elemtípus elhelyezkedése a Byte típusban

Az ábrán a nagyobb betűs sorszámok bájt, a kisebb betűsek pedig bit sorszámok. A „*” jelöli az elemtípus intervallumát, a „-” a kiegészített területet. Eszerint egy ilyen típusú halmaz helyigénye 2 bájt. Az ábra legalsó sorában egy konkrét érték, a [10, 12, 18] halmaz tárolása látható.

A halmaz bizonyos esetekben *tömörebb, takarékosabb* tárolást tesz lehetővé mint más típusok. A fenti példa ezt jól illusztrálja, hiszen, ha ilyen típusú értékeket tömbben akarunk tárolni, ahhoz legalább egy 10 elemű tömb, tehát 10 bájtnyi terület szükséges.

A Pascal halmazokkal a szokásos *halmazműveletek* (egyesítés, különbség, metszet) mellett, két halmaz összehasonlítása (a valódi tartalmazás nélkül) és egy *érték tartalmazásának* vizsgálata végezhető.

4.2. Nagy halmazok

A standard Pascal halmazok elemszáma erősen korlátozott. Ez azonban nem jelenti azt, hogy – akár modellezési gyakorlatként is – ne lehetne ilyen tulajdonságokkal rendelkező, de kevésbé korlátozott adatstruktúrát létrehozni. Ha ezt

az adatstruktúrát úgy hozzuk létre, hogy – a standard adattípusok módjára – általánosan használható lesz, akkor *implementáltuk* az adattípust a nyelvben.

A standard halmazok tárolási módját követve, a bittérképet valamilyen alkalmas módon, pl. tömbben tárolva és kezelve, *minden olyan elemtípushoz, amely indexként használható,* konstruálhatunk egy halmaztípust. Példaképpen válasszuk a `-8192..8191` intervallumtípust. Itt egy halmaz elemeinek maximális száma `16384` lesz.

Az implementációban csak a rendszerben standard módon meglévő eszközöket használunk fel. A nyelv sajátosságainak megfelelően, az implementációt megtestesítő szoftver erőforrásokat egy unit formájában adjuk meg. Az alapvető műveletek mellett még a halmaz aktuális elemszáma is lekérdezhető. Az implementációs unitunk az `uNHalm` unit. Ehhez fűzünk az alábbiakban magyarázatot. A teljesebb megértéshez javasoljuk a programszöveg tanulmányozását is, különös tekintettel az érték és a neki megfelelő bit egymáshoz rendelésére és a bájtokon végzett logikai műveletekre.

Publikus (kívülről is elérhető) programelemek

```
const
  {elemtípus min., max.}
  MinNHElem=-8192; MaxNHElem=8191;
  {egy halmaz max. elemszáma= 16384}
  MaxNHDb=MaxNHElem-MinNHElem+1;
  {a bittérkép tömb elemszáma=2048}
  NHTombEdb=MaxNHDb div 8;
type
  TNHElem=MinNHElem..MaxNHElem;  {elemtípus}
  TNHDb=0..MaxNHDb;  {elemdarabszám típus}
  TNHTombInd=1..NHTombEdb;  {a bittérkép tömb indextípusa}
  {a bittérkép tömb típusa}
  TNHTomb=array[TNHTombInd] of Byte;

  {a halmaz típusa}
  TNHalmaz=record
    EDb: TNHDb;  {aktuális elemszám}
    Adat: TNHTomb;  {bittérkép tömb}
  end;

{az A üres lesz}
procedure NHUres(var A: TNHalmaz);
{az A ürese-e}
function NHUrese(const A: TNHalmaz): Boolean;
```

```
{az E eleme-e A-nak}
function NHbane(E: TNHElem; const A: TNHalmaz): Boolean;
{az A elemszáma}
function NHEDb(const A: TNHalmaz): TNHDb;
{az E eleme lesz A-nak}
procedure NHba(E: TNHElem; var A: TNHalmaz);
{az E törlődik A-ból}
procedure NHbol(E: TNHElem; var A: TNHalmaz);
{a C az A és B egyesítése lesz}
procedure NHOssze(const A, B: TNHalmaz; var C: TNHalmaz);
{a C az A és B metszete lesz}
procedure NHMetsz(const A, B: TNHalmaz; var C: TNHalmaz);
{az A a B része-e}
function NHResz(const A, B: TNHalmaz): Boolean;
{az A és a B egyenlő-e}
function NHAzon(const A, B: TNHalmaz): Boolean;
```

Privát (kívülről nem elérhető) programelemek

```
const
  MaxBit=8;  {bitek száma egy bájtban}
type
  ByteInd=TNHTombInd;  {bájt index a bittérkép tömbben}
  BitInd=1..MaxBit;  {bit index a bájtban}
const
  Egyseg: array[BitInd] of Byte=(1, 2, 4, 8, 16, 32, 64, 128);
  {pontosan 1 darab 1-es bitet tartalmazó bájtok a logikai
   műveletekhez}

{az E értéknek megfelelő bit helye a bittérképen}
procedure ErtekHely(E: TNHElem; var ByteI: ByteInd;
  var BitI: BitInd);
{a B bájt I-edik bitjének értéke}
function BitErt(B: Byte; I: BitInd): Byte;
{a B bájt I-edik bitje 1-e}
function Bit1e(B: Byte; I: BitInd): Boolean;
```

Könnyen belátható, hogy ezzel a módszerrel a programfejlesztő környezetben használható legnagyobb méretű bájttömb elemszámának nyolcszorosáig mehetünk el (ez a szám Turbo Pascalban `8*65520`, Object Pascalban `8*4294967296`).

Az implementáció programozása némileg egyszerűbb lenne, ha bittérkép helyett bájttérképet használnánk, tehát egy logikai tömböt vennénk fel, amelynek egy eleme (egy bájt) lenne hozzárendelve egy értékhez. Ezzel persze a halmaz kapacitása nyolcadára csökken, de modellezési gyakorlatként javasoljuk egy ilyen elkészítését.

Megjegyezzük még, hogy az alkalmazott módszer (akár a bittérkép, akár a bájttérkép esetén) úgy is megfogalmazható, hogy egyfajta *inverz* tárolást valósítunk meg, az érték helyett az érték *sorszámát* használjuk, vagy másképpen kifejezve: *az értékkel indexelünk*. Ez a módszer tömören leírható és gyorsan, hatékonyan végrehajtható algoritmusokhoz vezet általában. Természetesen látnunk kell azt is, hogy ennél a módszernél az aktuálisan nemlétező elemeket is tároljuk (invertáltan), tehát mindig a teljes értéktartomány számára foglaljuk le a helyet. Az alkalmazás minimális feltétele, hogy az értékkel lehessen indexelni és legyen elegendő tárkapacitás.

4.3. Mintapéldák

4.3.1. Alapfeladatok

Egy halmaz elemeit csak úgy tudjuk egyenként feldolgozni, ha az elemtípus által meghatározott tartomány minden elemét megvizsgáljuk, a halmazba tartozás szempontjából. Ha ismerjük a halmazban lévő minimális vagy maximális értéket, akkor a vizsgálati tartomány szűkíthető. Az alábbiakban `Byte` elemtípusú standard halmazokat veszünk, de az algoritmusok szerkezete más esetekben is ugyanez.

Közös adatszerkezeti definíciók:

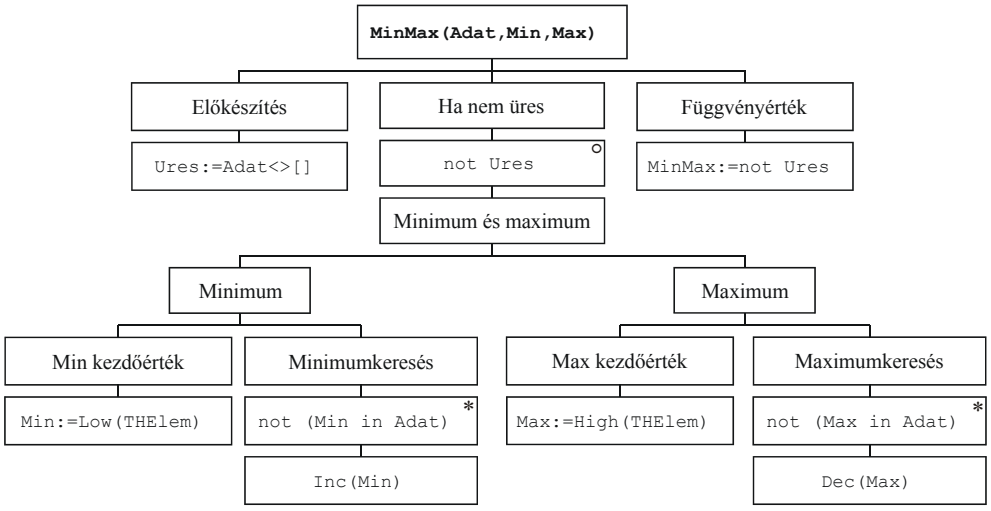
```
const
  HElemMaxDb=256;
type
  THElem=Byte;
  THalmaz=set of THElem;
  THElemDb=0..HElemMaxDb;
```

4.3.1.1. mintafeladat: Határozzuk meg egy halmaz minimális és maximális értékű elemét!

Adatszerkezet (21)

Azonosító	Funkció	Típus	Jelleg
`Adat`	ebben keresünk	`THalmaz`	input
`Min`	minimális érték	`THElem`	output
`Max`	maximális érték	`THElem`	output
`MinMax`	az eredmény létezése (nem üres-e a halmaz)	`Boolean`	output

Struktúradiagram (21)



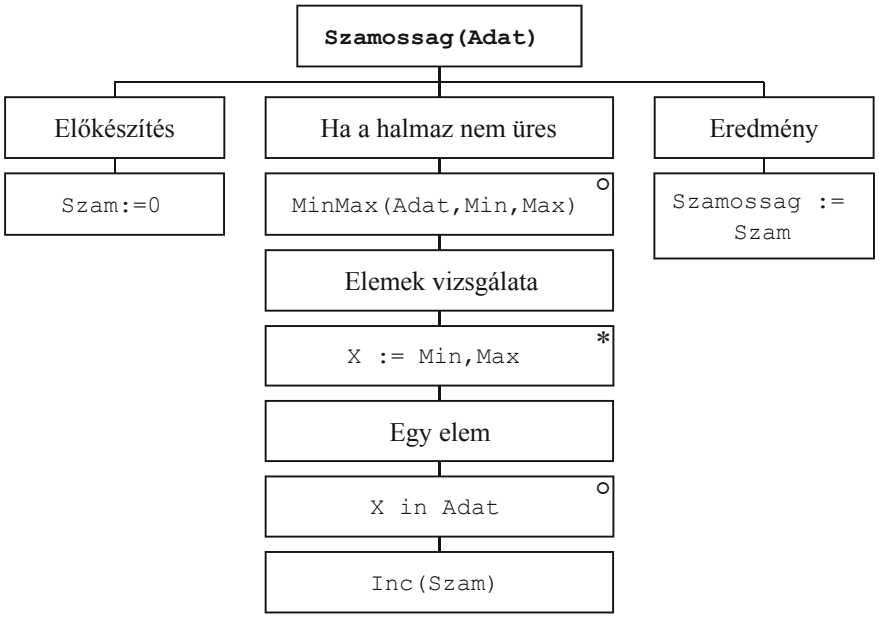
Szubrutin: `uHalm.MinMax`.

4.3.1.2. mintafeladat: Határozzuk meg egy halmaz elemszámát!

Adatszerkezet (22)

Azonosító	Funkció	Típus	Jelleg
`Adat`	adott halmaz	`THalmaz`	input
`Szamossag`	az `Adat` elemszáma	`THElemDb`	output
`Min`	minimális érték	`THElem`	munka
`Max`	maximális érték	`THElem`	munka
`X`	ciklusváltozó	`THElem`	munka
`Szam`	az `Adat` elemszáma	`THElemDb`	munka

Struktúradiagram (22)



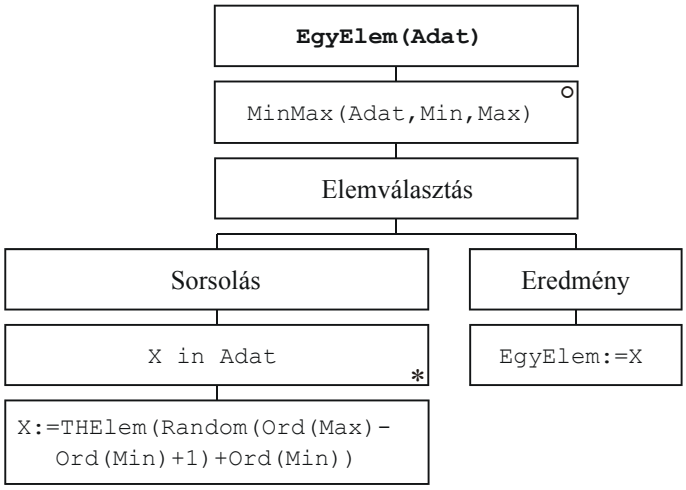
Szubrutin: `uHalm.Szamossag`.

4.3.1.3. mintafeladat: Válasszuk ki sorsolással egy halmaz egy elemét!

Adatszerkezet (23)

Azonosító	Funkció	Típus	Jelleg
`Adat`	ebben keresünk	`THalmaz`	input
`EgyElem`	eredmény	`THElem`	output
`Min`	minimális érték	`THElem`	munka
`Max`	maximális érték	`THElem`	munka
`X`	sorsolt érték	`THElem`	munka

Struktúradiagram (23)



Szubrutin: `uHalm.EgyElem`.

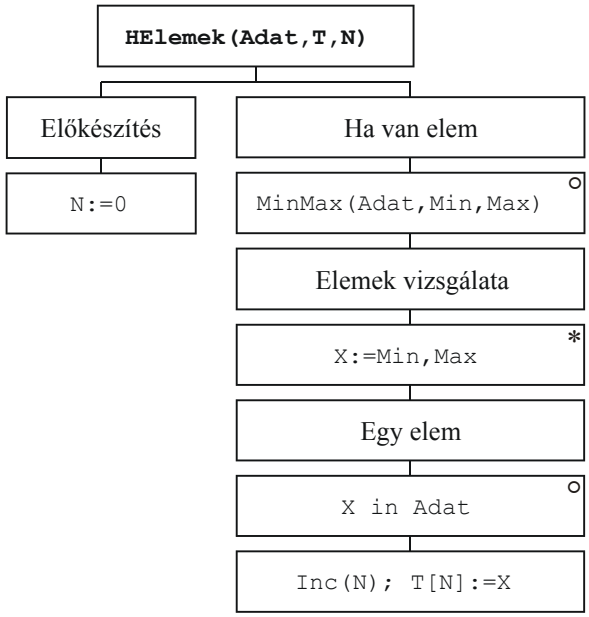
4.3.1.4. mintafeladat: Gyűjtsük ki egy halmaz elemeit egy tömbbe!

```
type
  THTomb=array[THElem] of THElem;
```

Adatszerkezet (24)

Azonosító	Funkció	Típus	Jelleg
`Adat`	ebből gyűjtünk	`Thalmaz`	input
`T`	eredménytömb	`THTomb`	output
`N`	`T` elemszáma	`THElemDb`	output
`Min`	minimális érték	`THElem`	munka
`Max`	maximális érték	`THElem`	munka
`X`	ciklusváltozó	`THElem`	munka

Struktúradiagram (24)



Szubrutin: `uHalm.HElemek`.

Megjegyzések:

- Vegyük észre, hogy a tömb növekvően rendezetten keletkezik.
- Ha ismerjük a halmaz elemszámának egy pontosabb felső korlátját, akkor a tömb ennek megfelelően kisebb méretűre is deklarálható.

4.3.2. Sorsolás

A számítógépes sorsolásoknál, véletlenszerű kiválasztásoknál gyakori követelmény, hogy az adatok ne ismétlődjenek. Ezt, ha az adatok *sorszámjellegűek,* vagy *sorszámozhatók,* legegyszerűbben halmazok segítségével tudjuk megoldani. Nézzünk erre mintapéldákat, az előző pontbeli típusdeklarációk megtartásával.

4.3.2.1. mintafeladat: Állítsunk elő sorsolással egy előírt elemszámú és előírt értékhatárok közé eső elemeket tartalmazó halmazt!

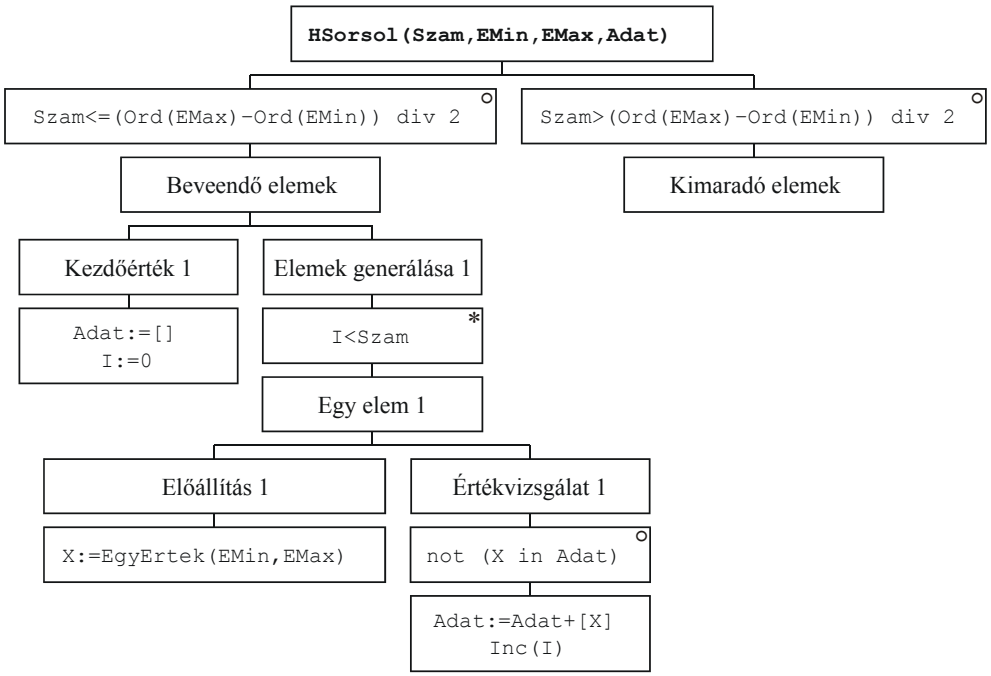
Útmutató ♦ Az algoritmust gyorsítjuk azzal, ha minél kevesebb elemet sorsolunk, ugyanis csak az első elemnél bizonyos az, hogy egy sorsolással megkap

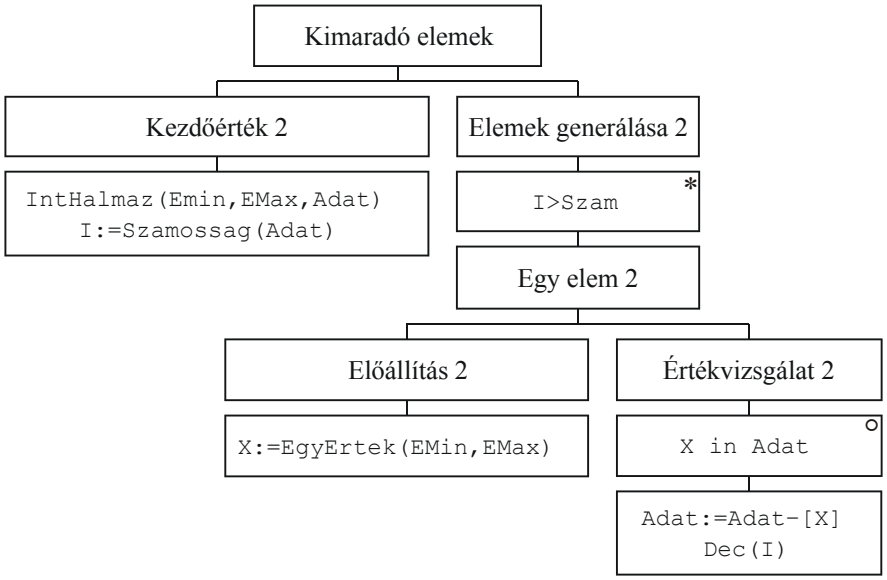
juk. Minél több a már kisorsolt elemek száma, annál kisebb a valószínűsége annak, hogy a következő sorsolás egy új értéket ad, más szóval egy új érték kiválasztásához egyre több sorsolás kell. Ezért, ha az előírt darabszám nagyobb mint a lehetséges értékek (az előírt értékhatárok közé eső értékek) számának fele, akkor az eredményből kimaradó értékeket sorsoljuk, hiszen ezek vannak kevesebben. A megoldásban felhasználjuk az `uHalm.EgyErtek`, `uHalm.Szamossag` és `uHalm.Inthalmaz` szubrutinokat.

Adatszerkezet (25)

Azonosító	Funkció	Típus	Jelleg
`Szam`	előírt elemszám	`THElemDb`	input
`EMin`	előírt minimum	`THElem`	input
`EMax`	előírt maximum	`THElem`	input
`Adat`	eredmény	`THalmaz`	output
`I`	számláló	`THElemDb`	munka
`X`	sorsolt érték	`THElem`	munka

Struktúradiagram (25)





Szubrutin: `uHalm.HSorsol`.

Megjegyzés ♦ Ha az előírt elemszám nagyobb lenne a lehetségesnél, akkor az eredmény az előírt értékhatárok közé eső összes elemet tartalmazó halmaz lesz.

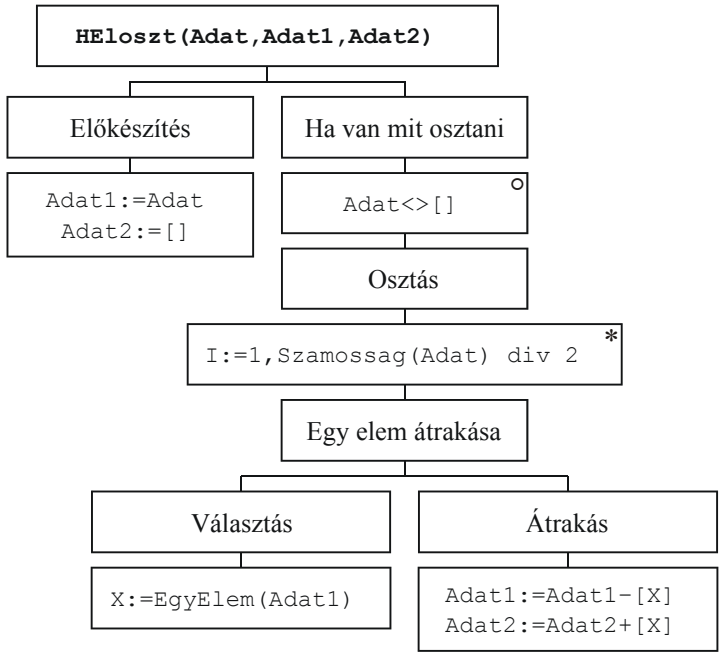
4.3.2.2. mintafeladat: Egy halmazt osszunk sorsolással két részre úgy, hogy a két fél elemszáma közt max. 1 lehet a különbség!

Útmutató ♦ Az elemek felét sorsolással kiválasztva átrakjuk egy másik halmazba. A megoldásban felhasználjuk az `uHalm.Szamossag` és `uHalm.EgyElem` szubrutinokat.

Adatszerkezet (26)

Azonosító	Funkció	Típus	Jelleg
`Adat`	kiindulási halmaz	`THalmaz`	input
`Adat1`	egyik eredmény	`THalmaz`	output
`Adat2`	másik eredmény	`THalmaz`	output
`I`	számláló	`THElemDb`	munka
`X`	sorsolt érték	`THElem`	munka

Struktúradiagram (26)



Szubrutin: `uHalm.HEloszt`.

4.3.2.3. mintafeladat: Állítsuk elő sorsolással az 5-ös lottó (5 a 90-ből) egy húzási eredményét, növekvő számsorrendben!

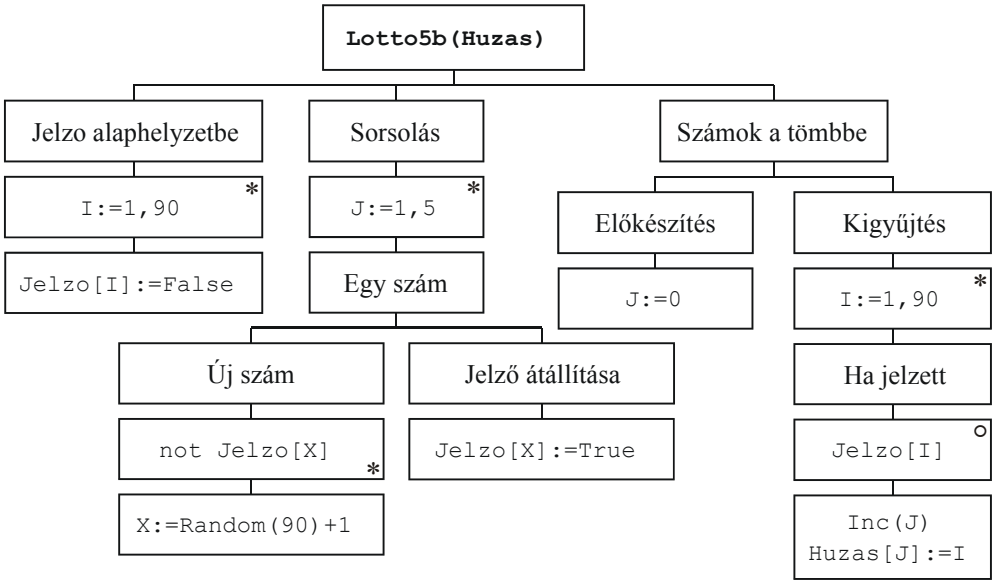
Útmutató ♦ A feladatot megoldhatjuk közvetve, az `uHalm.HSorsol` és az az `uHalm.HElemek` szubrutinok megfelelően paraméterezett hívásaival is. Ezt a megoldást adjuk meg az `uHalm.Lotto5a` szubrutinban. Egy másik, közvetlen megoldás, hogy nem használjuk a Pascal halmazt, hanem direkt erre a célra definiálunk egy halmaz adatstruktúrát, egy 90 elemű logikai tömbbel (bájt térkép). Ezt részletezzük az alábbiakban.

Adatszerkezet (27)

```
type
  TL5Szam=1..90; TL5Ind=1..5; TL5Db=0..5;
  TL5Huzas=array[TL5Ind] of TL5Szam;
  TL5Jelzo=array[TL5Szam] of Boolean;
```

Azonosító	Funkció	Típus	Jelleg
Huzas	eredmény	TL5Huzas	output
Jelzo	bájt térkép	TL5Jelzo	munka
I	ciklusváltozó	TL5Szam	munka
J	ciklusváltozó	TL5Db	munka
X	sorsolt érték	TL5Szam	munka

Struktúradiagram (27)



Szubrutin: `uHalmAlk.Lotto5b`.

4.3.3. Ellenőrzött input

4.3.3.1. Alapfogalmak

Egy rendszer működésénél alapvető követelmény, hogy a „kívülről érkező", tehát még nem ellenőrzött (például az adatbevitellel létrejövő, ekkor keletkező) adatok maximálisan ellenőrizve legyenek a rendszerben való tárolás és/vagy felhasználás előtt. Tipikus adatbeviteli forma a billentyűzetről érkező jelsorozat, ebben a fejezetben ezzel foglalkozunk.

Az *ellenőrzött inputon* itt azt az adatbevitel-programozási technikát értjük, amellyel az adatbeviteli formai és/vagy tartalmi hibákat minél hamarabb, lehetőleg már a hiba keletkezésének pillanatában, optimális esetben már a hibát

okozó jel beérkezésekor észreveszi és lekezeli a program (utólagos hibaüzenetek helyett) ezzel megakadályozva a hibás adat keletkezését.

Az adatbevitelnél elsődlegesen keletkező adat formailag mindig egy string, az ellenőrzési technikák jelentős részben a *stringek* és *jelhalmazok* kezelésére épülnek, itt is elsősorban ilyeneket tárgyalunk. Az általánosan alkalmazott módszerek:

- *Maszkolt,* vagyis mintához igazított bevitel. A string minden pozíciójához hozzá van rendelve az abban a pozícióban elfogadható jelek halmaza. A jelhalmazok rövid jelölésére egyezményes jeleket használunk, ezekből áll össze a maszkot (mintát) megadó string. Például, ha a nagybetűk halmazának jele az `A` betű, a számjegyeké a `9` számjegy, a `0` által jelzett halmaz pedig az előző kettő egyesítése, akkor `AA0999` a jelenlegi magyar gépkocsirendszám adat maszkja. Ez a módszer nem elég általános, alapformájában csak a fix hosszú adatokra alkalmazható, további feltételeket (pl. azt hogy a rendszám számrésze nem lehet csupa nulla) csak kiegészítő ellenőrzésekkel lehet érvényesíteni.
- Jelenkénti előzetes *halmazkonstrukció* módszere. Alapelve az, hogy az adatbevitel minden pillanatában meg tudjuk határozni, a string aktuális állapotának a függvényében, a következő jelként elfogadható jelek halmazát. Ez a módszer elvben teljesen általános, például alkalmas változó hosszú adatok, elválasztójelek, sorrendi viszonyok kezelésére is. Könnyen belátható, hogy speciális esetként magában foglalja a maszkos módszert is. Konkrét megvalósítására több példát fogunk adni. Természetesen itt előfordulhat, hogy bizonyos feltételek kezelése egyszerűbb utólagos kiegészítő ellenőrzéssel, mint azonnali érvényesítéssel.
- Teljes, utólagos *stringellenőrzés* módszere. Alapelve az, hogy az érkező jelet előzetes ellenőrzés nélkül beleilleszti a stringbe, majd a létrejött teljes stringet vizsgálja. Ha ez már nem felel meg a feltételeknek, akkor visszaállítja az előző állapotot (vagyis törli a jelet a stringből). Ez a módszer is teljesen általános. Konkrét megvalósításaiban alkalmazhatunk halmazokat is, de nem kötődik olyan mértékben ehhez az adatstruktúrához, mint az előző kettő.

A billentyűzet a számítógép egy periféria egysége, külső eszköze, ezért a hozzáférés, kezelés módja erősen operációsrendszer függő. Ez a függőség érvényesül a programfejlesztő környezetekben is. Egy hagyományos, a szekvenciális programvégrehajtás alapelvére épülő (pl. MS–DOS alapú) fejlesztőrendszerben (Turbo Pascal) a billentyűzetről érkező jelsorozatot teljes mértékben „kezünk-

ben tarthatjuk", egyenként elemezhetjük, akár normál adatjelek (pl. betűk), akár vezérlő jellegű jelek (pl. ESC, ENTER, BACKSPACE) ezek. A fejlesztő rendszerben a billentyűzetkezeléshez adott eszközöket (`Crt` unit) felhasználva, maradéktalanul meg tudjuk valósítani az előzetes jelellenőrzést a halmazkonstrukcióval.

Más a helyzet egy alapvetően az eseménykezelés alapelvére épülő (pl. Windows) alapú fejlesztőrendszerben (Delphi). Itt a program, némi egyszerűsítéssel, a bekövetkezett eseményekre adott válaszokat leíró algoritmusokból (szubrutinokból) áll. Az operációs rendszer eleve lekezel bizonyos eseményeket, így az input folyamat egyes részletei (pl. egyes vezérlőjelek kezelése) rejtve maradnak a programozó elől. Az operációs rendszer bizonyos szolgáltatásai (pl. vágólap) is megnehezítenék az előzetes jelellenőrzést. A fejlesztő rendszer kész kereteket ad, amelyekbe meghatározott helyre és formában be kell illeszteni a saját készítésű szubrutinokat. Az ilyen rendszerekhez jobban illeszkedik az utólagos stringellenőrzés módszere.

Az alábbiakban a halmazkonstrukciós és a stringellenőrzéses módszerre adunk mintákat. A maszkolt adatbevitel az előbbi speciális esete (a halmaz csak az aktuális pozíciótól függ), ezzel külön nem foglalkozunk.

4.3.3.2. Halmazkonstrukció

Előrebocsátjuk hogy az itteni mintapéldák és az ide kapcsolódó kitűzött feladatok a fentebb vázolt rendszerfüggőség miatt csak Turbo Pascal rendszerben értelmezhetők.

Jelhalmazokat adunk meg és építünk fel a hibák kiszűrésére, kizárására. Az adatot mindig egy string változóban gyűjtjük össze. Adatbevivő szubrutinjaink mind *felvivő,* mind *módosító* funkcióban működnek, ha az adattároló string induláskor üres, akkor új adat összeállítása történik, ha nem üres, akkor feltételezzük, hogy tartalma egy módosítandó, de *nem hibás* adat.

Az egyszerűség kedvéért csak 3 *vezérlőjelet* értelmezünk:

- Adatbevitel közben csak visszatörléssel (BACKSPACE billentyű) javíthatunk
- Az adatbevitel végét ENTER billentyűvel jelezzük (van új, helyes adat).
- Az adatbevitelt bármikor *megszakíthatjuk* az ESC billentyűvel, az adat tartalma üres lesz, (nincs új, helyes adat).

Egyéb jellegzetességek:

- az adatbevitel *helye* a képernyőn *paraméterezhető,*
- az adat *helyét,* a lehetséges maximális adathosszban, *jelezzük* (inverz csíkkal)
- formailag kizárt jel nem vihető be, leütését *hibajelzés* (hang) kíséri

Az adattároló stringet *tömbként* kezeljük, elemeinek, jeleinek kezelésével állítjuk be az adat aktuális értékét. A billentyűzetről a standard `Crt.ReadKey` függvénnyel olvasunk be. Az adatbevitelnél és a képernyőre írásnál használt típusokat és konstansokat a `tDef`, a segédszubrutinokat (más hasonló jellegű és célú szubrutinokkal együtt) a `tKom` unit tartalmazza. Ezek:

- `InverzIr`, `NormalIr`: A `Vilagos` és `Sotet` színbeállítás segítségével jelöljük ki az adat helyét a képernyőn. Ez az adatbevitel alatt egy inverz csík. Természetesen más színkombináció is választható.
- A `BillBe` egy általános jelbekérő, mind a normál mind a funkcióbillentyűk (két jelet adó billentyűk) kezelésére alkalmas.
- A `JelBe` a `BillBe` egy szűkítése, alkalmazása az egy jelet adó billentyűkre.
- Az `Ir` a pozícionálást és képernyőre írást vonja össze. Ha egy helykoordináta nem megadott (= `0`) akkor *kurzor* aktuális koordinátája a mérvadó (alapértelmezés).
- A `KiIras` az adatbekérő szubrutinok általános kiíró eljárása, a *hellyel,* a megjelenítés *módjával* (normál, inverz) valamint az adat logikai *hosszával* paraméterezhető. Ez utóbbi azért kell, hogy a kurzor ne az inverz csík végén, hanem az utolsó bevitt jel után jelenjen meg.

4.3.3.2.1. mintafeladat: Olvassuk be ellenőrzött inputtal az alábbiak szerint specifikált általános szövegadatot:

- *Az adatban kétféle jel lehet, adat- és elválasztójel. Ezek paraméterként adottak.*
- *Előírt az adat minimális és maximális hossza.*
- *Előírt az adatban lévő elválasztójelek minimális és maximális száma.*
- *Elválasztójel nem állhat sem az adat első, sem utolsó jeleként, sem elválasztójel után.*

Útmutató ♦ A kijelzés kedvéért, a stringet mindig a maximális hosszban, szóközökkel jobbról feltöltve tartjuk. A mindenkori tényleges adathosszt egy külön változóban adminisztráljuk. A módosítási lehetőség miatt a string induló tartalmát külön meg kell vizsgálni, beállítandó a specifikációból következően a halmazkonstrukciónál lényeges jellemzők (adathossz, elválasztójel darabszám, utolsó jel) aktuális értékét. A halmazkonstrukcióban érvényesítjük a specifikációs előírásokat. Az ezután beolvasott jel az előzetes konstrukció és szűrés következtében csak olyan jel lehet, amely itt helyes. Ennek beolvasása után feldolgozzuk, vagyis vagy beillesztjük az adatba (normál jel), vagy végrehajtjuk a

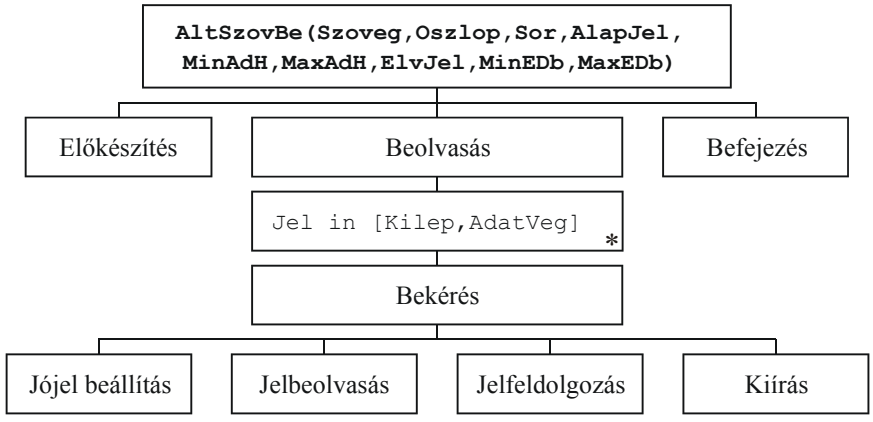
megfelelő akciót (visszatörlés, kilépés). A kilépési jelet (ENTER vagy ESC) a ciklus végfeltétele dolgozza fel.

Adatszerkezet (28)

```
type
  TJelek=set of Char;
```

Azonosító	Funkció	Típus	Jelleg
Szoveg	az adat	String	input, output
Oszl	képernyőpozíció oszlop	Byte	input
Sor	képernyőpozíció sor	Byte	input
AlapJel	alapjelek	TJelek	input
MinAdh	minimális adathossz	Byte	input
MaxAdh	maximális adathossz	Byte	input
ElvJel	elválasztó jelek	TJelek	input
MinEDb	elválasztó jelek minimális száma	Byte	input
MaxEDb	elválasztó jelek maximális száma	Byte	input
AltSzovBe	az adat érvényessége	Boolean	output
Jel	aktuális (utolsó) jel	Char	munka
JoJel	aktuális jelhalmaz	TJelek	munka
Hossz	aktuális valódi adathossz	Byte	munka
ElvDb	elválasztó jelek aktuális száma	Byte	munka
I	ciklusváltozó	Byte	munka
VanAdat	az adat érvényessége	Boolean	munka

Struktúradiagram (28)



Szubrutin: `tInpA.AltSzovBe`.

4.3.3.2.2. mintafeladat: Olvassuk be ellenőrzött inputtal az alábbiak szerint specifikált gépkocsi rendszámot:

- *A rendszám pontosan 6 jegyű, az első két jel nagybetű, a harmadik lehet nagybetű vagy számjegy, a többi jel számjegy.*
- *A számrész nem lehet csupa 0.*

Útmutató ♦ Az előző mintapéldában megismert általános módszert alkalmazzuk a konkrét specifikációnak megfelelő, az előzőnél egyszerűbb halmazkonstrukcióval. A szükséges halmazkonstansokat a `tDef` unitban találjuk meg.

Adatszerkezet (29)

```
type
  TJelek=set of Char;
const
  AdatHossz=6;
  RBetuk=NagyBetuk+KisBetuk;
  AdatJel=RBetuk+SzamJegyek;
```

Azonosító	Funkció	Típus	Jelleg
`Rendszam`	az adat	`String`	input, output
`Oszl`	képernyőpozíció oszlop	`Byte`	input
`Sor`	képernyőpozíció sor	`Byte`	input
`RendSzamBe`	az adat érvényessége	`Boolean`	output
`Jel`	aktuális (utolsó) jel	`Char`	munka
`JoJel`	aktuális jelhalmaz	`TJelek`	munka
`Hossz`	aktuális valódi adathossz	`Byte`	munka
`i`	ciklusváltozó	`Byte`	munka
`VanAdat`	az adat érvényessége	`Boolean`	munka

Szubrutin: `tInpB.RendSzamBe`.

4.3.3.2.3. mintafeladat: Olvassuk be ellenőrzött inputtal egy adott határok közé eső egész számot!

Útmutató ♦ Mivel az értéktartomány a halmazkonstrukció közben csak nehezen lenne ellenőrizhető, célszerű ezt külön, második szinten elvégezni. Az

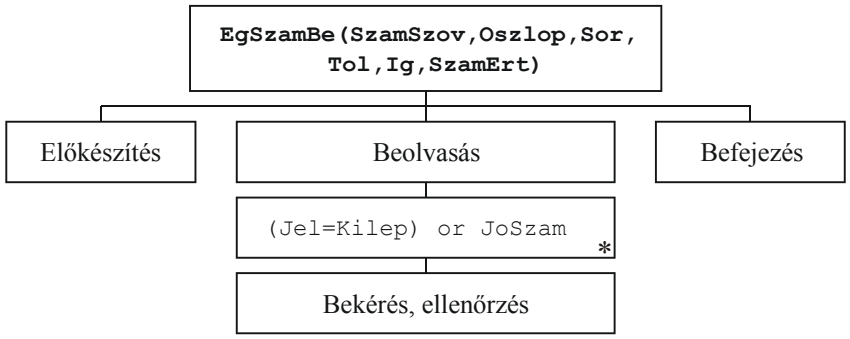
eredményt a string mellett egy számváltozóban is megadjuk. Az adat (string) hosszát az értéktartományból számoljuk ki.

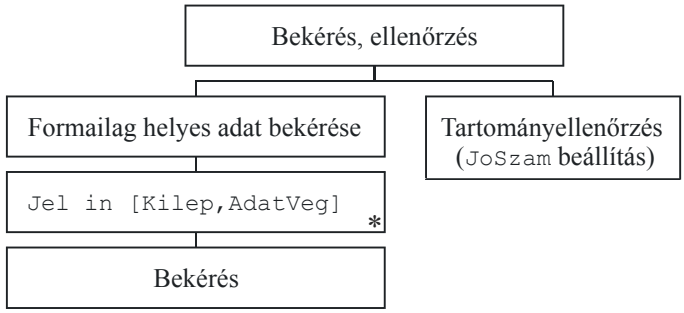
Adatszerkezet (30)

```
type
  TJelek=set of Char;
  TEgSzamHossz=0..11; {Longint, előjellel}
```

Azonosító	Funkció	Típus	Jelleg
SzamSzov	az adat stringben	String	input, output
Oszl	képernyőpozíció oszlop	Byte	input
Sor	képernyőpozíció sor	Byte	input
Tol	minimális érték	Longint	input
Ig	maximális érték	Longint	input
SzamErt	az adat számtípusban	Longint	output
EgSzamBe	az adat érvényessége	Boolean	output
Jel	aktuális (utolsó) jel	Char	munka
Hossz	aktuális valódi adathossz	TegSzamHossz	munka
JoJel	aktuális jelhalmaz	TJelek	munka
JoSzam	értéktartomány helyessége	Boolean	munka
W	munkaváltozó a hosszhoz	String	munka
MaxH	munkaváltozó a hosszhoz	TegSzamHossz	munka
I	munkaváltozó a konverzióhoz	Integer	munka
X	munkaváltozó a konverzióhoz	Longint	munka
VanAdat	az adat érvényessége	Boolean	munka

Struktúradiagram (30)





Szubrutin: `tInpA.EgSzamBe`.

A `tInpA` unit szubrutinjai arra is alkalmasak, hogy megfelelően paraméterezett meghívásukkal, tehát *közvetett halmazkonstrukciós* módon oldjuk meg az ellenőrzött inputot. Erre példák a `tInpB.SzemNevBe` és `tInpB.TestSulyBe` szubrutinok.

4.3.3.3. Stringellenőrzés

Ha egy, az eseménykezelés alapelvére épülő modern (pl. Windows) alapú fejlesztőrendszerben saját ellenőrzött inputot szeretnénk megvalósítani, akkor alapvetően két dolgot kell programoznunk és beillesztenünk a megfelelő keretrendszerbe:

- A választ arra ez eseményre, hogy az input adatbeviteli mező (string) megváltozott. A változás pontos okát nem is kell feltétlenül ismerni, ez többféle lehet (pl. adatjel bevitel, jeltörlés, beszúrás vágólapról). A feladatunk az, hogy minősítsük a változás eredményeként (feltételesen) létrejött stringet (közbenső ellenőrzés). Ha a válasz az, hogy a string nem megfelelő, akkor a keretrendszer visszaállítja a string eredeti állapotát.
- A választ arra ez eseményre, ha az input adatbeviteli mezőt elhagyjuk, vagyis az adat bevitelét befejezzük. A feladatunk az, hogy minősítsük az aktuális stringet (végellenőrzés). A nem megfelelőség esete aztán már többféleképpen is lekezelhető (pl. hibaüzenet).

Amint látható mindkét eseménykezelő egy olyan szubrutin, amelynek egyik (ez lehet az egyetlen is) input paramétere az ellenőrizendő string és egyetlen, logikai típusú eredményt (megfelelőség) ad. Az alábbiakban ilyenekre adunk mintapéldákat. Feltételezzük, hogy a keretrendszer biztosítja azt, hogy a végellenőrzés csak olyan stringgel hívódik meg, amelyiket a közbenső ellenőrzés elfo-

gadott. A megoldások, mint stringellenőrző szubrutinok alaprendszertől függetlenek.

4.3.3.3.1. mintafeladat: Készítsük el a közbenső és a végellenőrzést az alábbiak szerint specifikált általános szövegadathoz:

- *Az adatban kétféle jel lehet, adatjel és elválasztójel. Ezek paraméterként adottak.*
- *Előírt az adat minimális és maximális hossza.*
- *Előírt az adatban lévő elválasztójelek minimális és maximális száma.*
- *Elválasztójel nem állhat sem az adat első sem utolsó jeleként, sem elválasztójel után.*

Útmutató ♦ A *közbenső* ellenőrzésnél vizsgálható a maximális hossz és maximális darabszám, valamint az alapjelek és elválasztójelek egymásra következése. Az utóbbit itt is úgy oldjuk meg, hogy a string minden jeléhez előállítjuk az ott megfelelő jelhalmazt. A *végellenőrzésnél* vizsgálni kell még a minimális hosszat és minimális darabszámot, valamint az utolsó jelet.

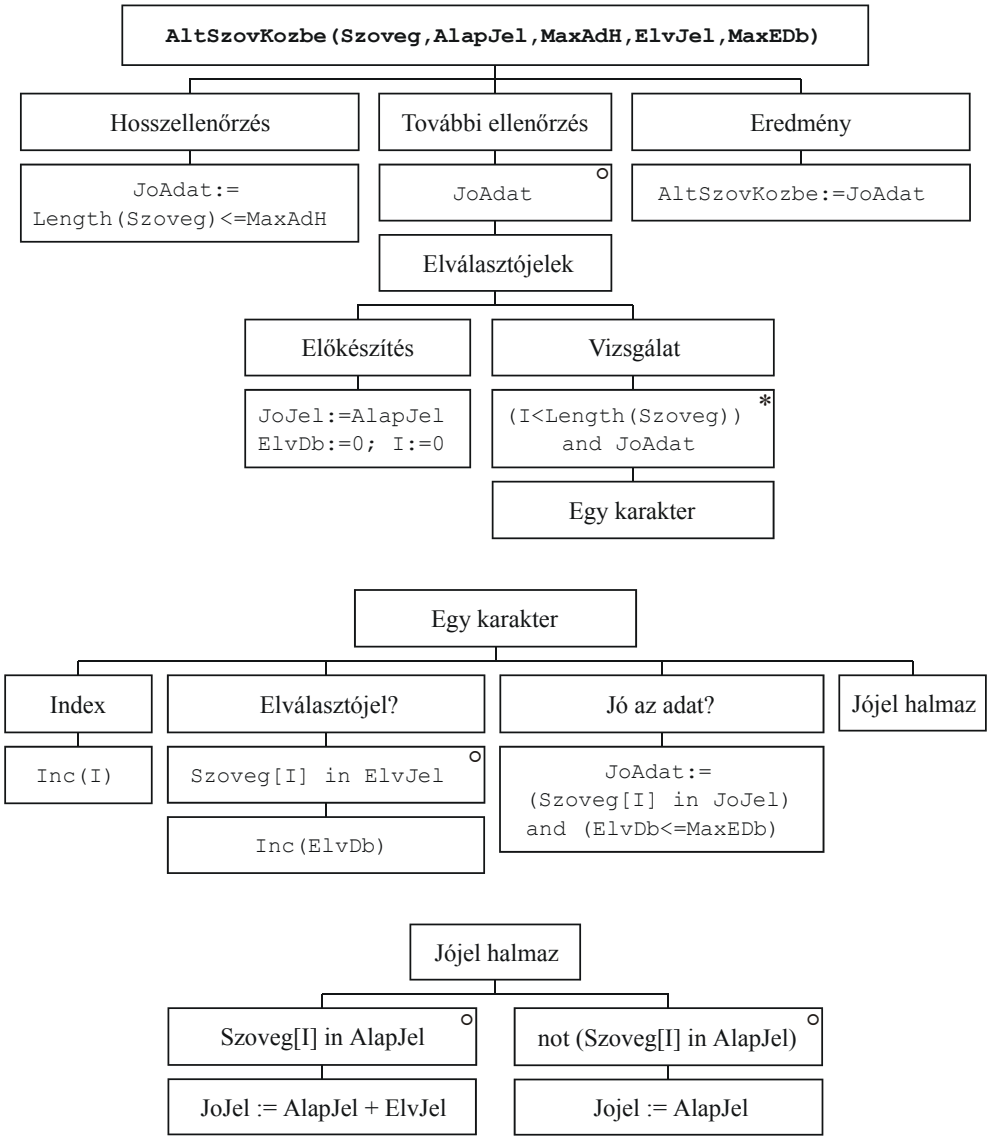
Közbenső ellenőrzés:

Adatszerkezet (31)

```
type
  TJelek=set of Char;
```

Azonosító	Funkció	Típus	Jelleg
Szoveg	az adat	String	input
AlapJel	alapjelek	TJelek	input
MaxAdH	maximális adathossz	Byte	input
ElvJel	elválasztó jelek	TJelek	input
MaxEDb	elválasztó jelek maximális száma	Byte	input
AltSzovKozBe	az adat érvényessége	Boolean	output
JoJel	aktuális jelhalmaz	TJelek	munka
ElvDb	elválasztó jelek aktuális száma	Byte	munka
I	ciklusváltozó	Byte	munka
JoAdat	az adat érvényessége	Boolean	munka

Struktúradiagram (31)



Szubrutin: `uInp.AltSzovKozbe`.

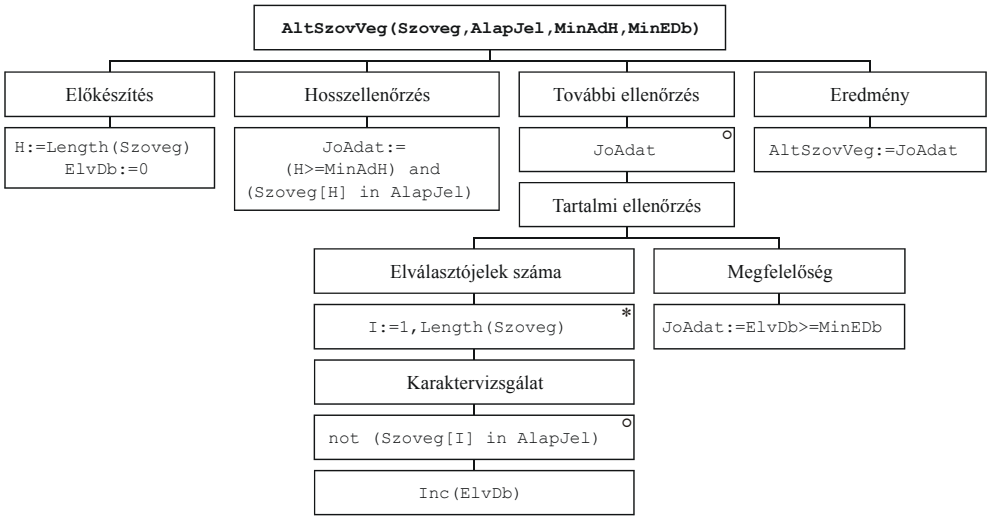
Végellenőrzés:

Adatszerkezet (32)

```
type
  TJelek=set of Char;
```

Azonosító	Funkció	Típus	Jelleg
Szoveg	az adat	String	input
AlapJel	alapjelek	TJelek	input
MinAdH	minimális adathossz	Byte	input
MinEDb	elválasztó jelek minimális száma	Byte	input
MaxEDb	elválasztó jelek maximális száma	Byte	input
AltSzovVeg	az adat érvényessége	Boolean	output
H	adathossz	Byte	munka
ElvDb	elválasztó jelek aktuális száma	Byte	munka
I	ciklusváltozó	Byte	munka
JoAdat	az adat érvényessége	Boolean	munka

Struktúradiagram (32)



Szubrutin: `uInp.AltSzovVeg`.

4.3.3.2.2. mintafeladat: Készítsük el a közbenső és a végellenőrzést az alábbiak szerint specifikált gépkocsirendszám adathoz:

- *A rendszám pontosan 6 jegyű, az első két jel nagybetű, a harmadik lehet nagybetű vagy számjegy, a többi jel számjegy.*
- *A számrész nem lehet csupa 0.*

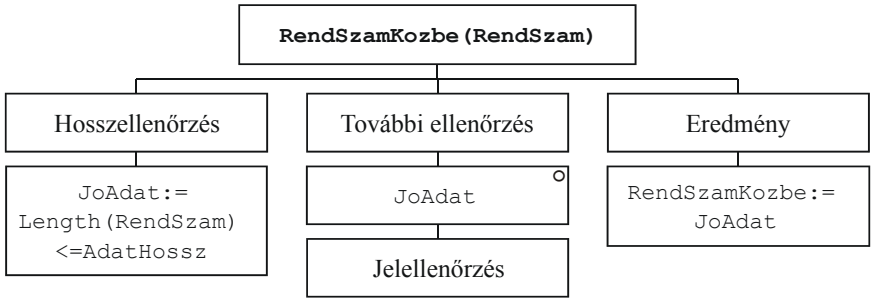
Útmutató ♦ A közbenső esetben az előző mintapéldában megismert módszert alkalmazzuk a konkrét specifikációnak megfelelően. A szükséges halmazkonstansokat az `uDef` unitban találjuk meg. A végellenőrzés egy logikai kifejezés, itt nem részletezzük (lásd `uInp.RendSzamVeg`).

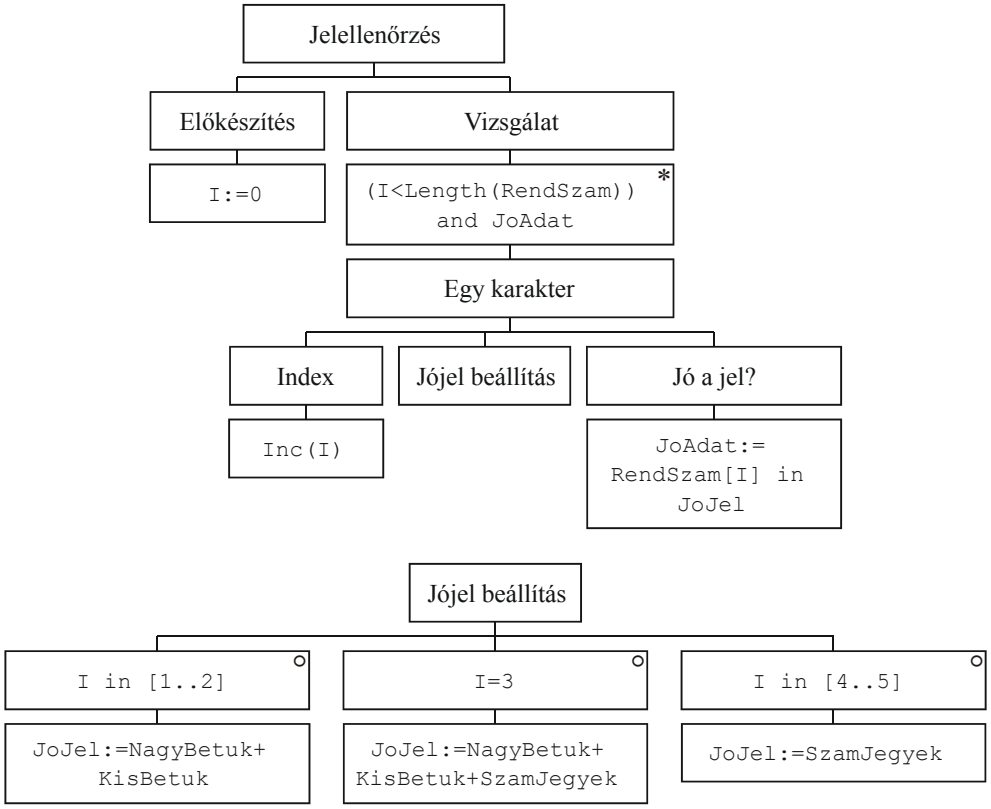
Adatszerkezet (33)

```
type
  TJelek=set of Char;
const
  AdatHossz=6;
```

Azonosító	Funkció	Típus	Jelleg
`Rendszam`	az adat	`String`	input
`RendszamKozbe`	az adat érvényessége	`Boolean`	output
`JoJel`	aktuális jelhalmaz	`TJelek`	munka
`I`	ciklusváltozó	`Byte`	munka
`JoAdat`	az adat érvényessége	`Boolean`	munka

Struktúradiagram (33)





Szubrutin: `uInp.RendSzamKozbe`.

4.3.4. Rendezés és statisztika

Az általános értelemben vett halmazstruktúra lehetőséget ad arra, hogy nagyon hatékony, gyors kereső, rendező és elemi statisztikai algoritmusokat készítsünk. Mint tudjuk, a keresés ebben a struktúrában egyetlen művelet (eleme-e), és látni fogjuk, hogy a rendezés műveletszáma csak lineárisan függ az elemek számától. (Mint korábban láttuk, a legjobb általános rendező algoritmusok is rosszabbak ennél egy logaritmikus szorzótényezővel). Természetesen a halmazmódszer nem általános (nem lehet vele pl. stringeket rendezni), az alkalmazhatóság szükséges feltétele az, hogy az értékek sorszámjellegűek legyenek, vagyis *az értékkel* lehessen *indexelni.*

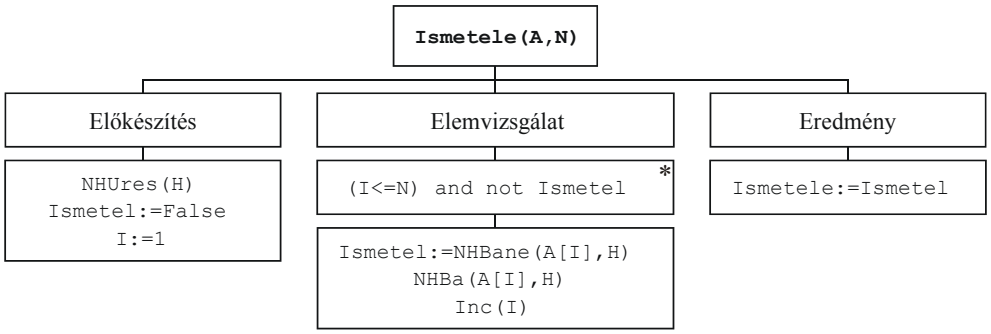
4.3.4.1. mintafeladat: Egy adatsor évszámokat tartalmaz. Állapítsuk meg, hogy van-e benne ismétlődés!

Útmutató ♦ Feltételezzük, hogy az évszámok belefférnek az `uNHalm` unit halmazelem típusába. Az itteni halmaztípust és az `uTomb`-beli tömbtípust használjuk.

Adatszerkezet (34)

Azonosító	Funkció	Típus	Jelleg
`A`	a vizsgálandó sor	`TSor`	input
`N`	az `A` elemszáma	`TElemDb`	input
`Ismetele`	ismétlődés jelző	`Boolean`	output
`Ismetel`	ismétlődés jelző	`Boolean`	munka
`H`	halmaz a vizsgálathoz	`TNHalmaz`	munka
`I`	index	`TEIndex1`	munka

Struktúradiagram (34)



Szubrutin: `uHalmAlk.Ismetele`.

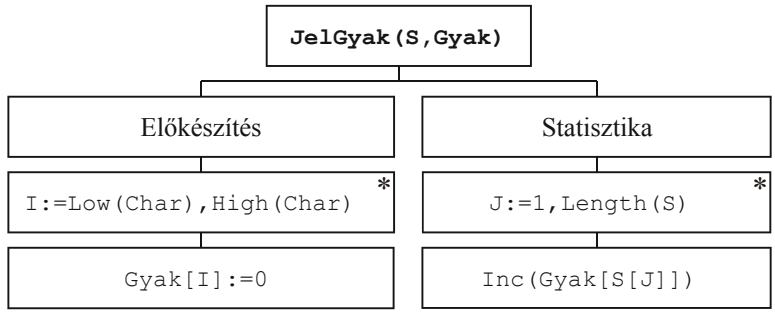
4.3.4.2. mintafeladat: Állítsuk elő egy stringben lévő jelek érték szerint növekvően rendezett gyakorisági statisztikáját!

Adatszerkezet (35)

```
type
  TJelGyak=array[Char] of Byte;
```

Azonosító	Funkció	Típus	Jelleg
S	a vizsgálandó string	String	input
Gyak	a gyakoriságok	TJelGyak	output
J	index a stringhez	Byte	munka
I	index a Gyak tömbhöz	Char	munka

Struktúradiagram (35)



Szubrutin: uHalmAlk.JelGyak.

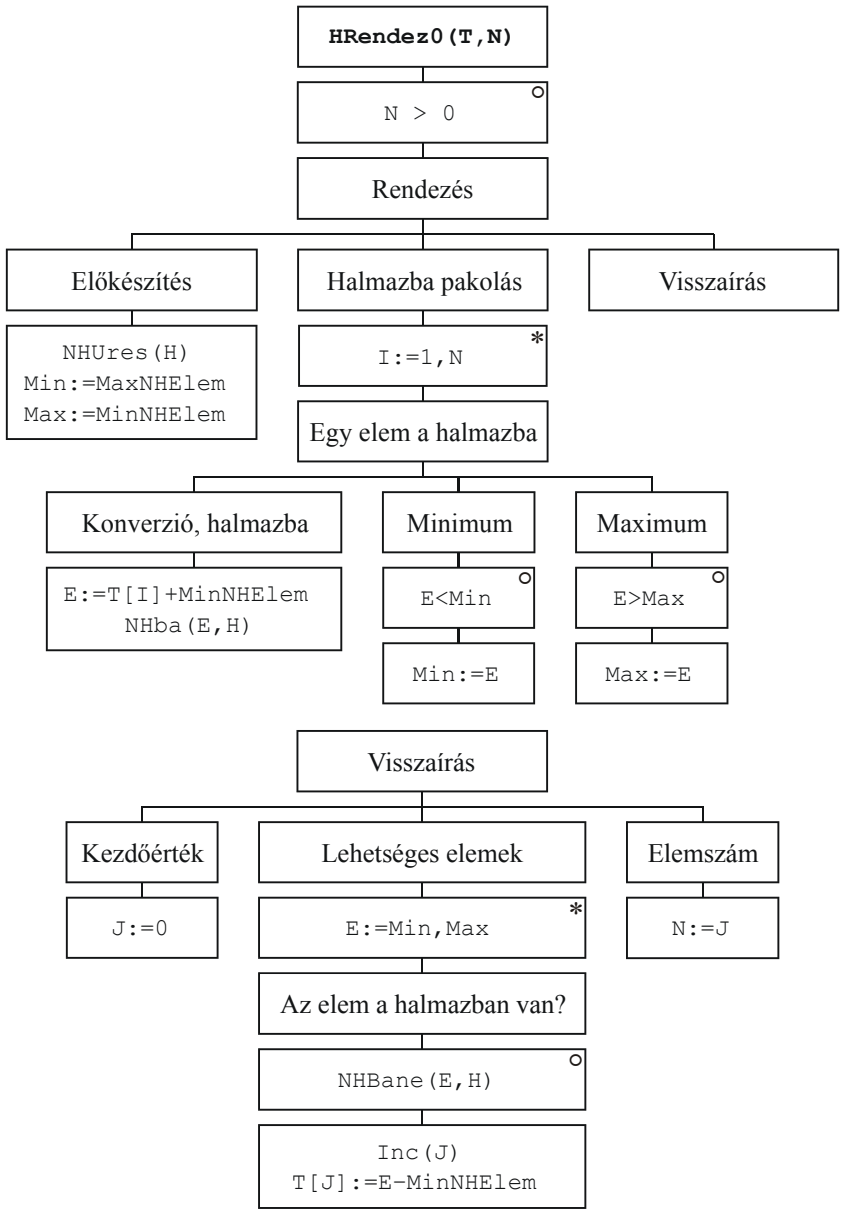
4.3.4.3. mintafeladat: Egy adatsor maximum 4 jegyű nem negatív egész számokat tartalmaz. Rendezzük az elemeket növekvő sorrendbe az ismétlődések törlésével!

Útmutató ♦ Az eredeti értékek ugyan nem férnek bele az uNHalm unit halmazelem típusába, de ez egyszerű transzformációval (eltolás) elérhető. Az itteni halmaztípust és az uTomb-beli tömbtípust használjuk. A végeredményben természetesen vissza kell állítanunk az eredeti értékeket. Az esetleges törlések miatt az adatsor elemszáma is változhat.

Adatszerkezet (36)

Azonosító	Funkció	Típus	Jelleg
T	a rendezendő tömb	TSor	input, output
N	a T elemszáma	TElemDb	input, output
H	halmaz a rendezéshez	TNHalmaz	munka
I	index	TEIndex	munka
I	számláló	TElemDb	munka
E	halmazelem	TNHElem	munka
Min	minimális halmazelem	TNHElem	munka
Max	maximális halmazelem	TNHElem	munka

Struktúradiagram (36)



Szubrutin: `uHalmAlk.HRendez0`.

Megjegyzés ♦ A megoldás minden olyan tömbre alkalmazható, ahol az elemek között a maximális eltérés kisebb mint 16384.

4.3.4.4. mintafeladat: Egy adatsor maximum 4 jegyű nem negatív egész számokat tartalmaz. Rendezzük az elemeket növekvő sorrendbe!

Útmutató ♦ Az előző megoldás annyiban általánosítandó, hogy a halmazba való átírásnál gyűjteni kell az értékek gyakoriságát, és a visszaírást ennek figyelembevételével kell elvégezni. Az adatsor elemszáma nem változik.

Szubrutin: `uHalmAlk.HRendez1`.

4.4. Feladatok

1 ♦ Készítsen egy közvetlen halmazkonstrukciós szubrutint adott korlátok közé eső és adott hosszúságú valós (törtrészt is tartalmazó) szám ellenőrzött inputtal való beolvasására (`tInpA.ValSzambe`)!

2 ♦ Egy intézményben a szobákat egy `ESXX` alakú adattal azonosítják, amelyben:

- `E` (épületszárny): lehet `A`, `B`, `C`, `D`, `I`;
- `S` (szint): lehet: az `A` és `B` szárnyban `1..6`, a `C` és `D` szárnyban `1..7`, az `I` szárnyban `1..5`;
- `XX` (sorszám): lehet `1..12` (előnullázott).

- Készítsen közbenső és végellenőrző szubrutint egy adott azonosító helyességének ellenőrzésére!
- Készítsen egy közvetlen halmazkonstrukciós szubrutint azonosító ellenőrzött inputtal való beolvasására!

3 ♦ Egy bástya egy lépését a sakktáblán egy `XX-YY` formátumú string alakjában adjuk meg, ahol `XX` a régi, az `YY` az új hely, pl.: `A1-A6`. Mint tudjuk, egy bástya mindig csak az aktuális helyének a során vagy az oszlopán mozoghat.

- Készítsen közbenső és végellenőrző szubrutint egy adott lépés helyességének ellenőrzésére!
- Készítsen egy közvetlen halmazkonstrukciós szubrutint lépés ellenőrzött inputtal való beolvasására!

4 ♦ Egy futó egy lépését a sakktáblán egy `XX-YY` formátumú string alakjában adjuk meg, ahol `XX` a régi, az `YY` az új hely, pl.: `A1-B2`. Mint tudjuk, egy futó mindig csak átlós irányban mozoghat.

- Készítsen közbenső és végellenőrző szubrutint egy adott lépés helyességének ellenőrzésére!
- Készítsen egy közvetlen halmazkonstrukciós szubrutint lépés ellenőrzött inputtal való beolvasására!

5 ♦ Közút azonosítóján (pl. M1, M0, 8265, E75) értünk egy olyan stringet, amely:

- min. 1, és max. 6 jelből áll;
- számjegyeket tartalmaz, de az első jele lehet nagybetű is;
- nem kezdődhet nullával.

- Készítsen közbenső és végellenőrző szubrutint egy adott azonosító helyességének ellenőrzésére!
- Készítsen egy közvetlen halmazkonstrukciós szubrutint azonosító ellenőrzött inputtal való beolvasására!

6 ♦ Egy Pascal halmazkonstanst stringben adunk meg.

- Készítsen közbenső és végellenőrző szubrutint egy adat helyességének ellenőrzésére!
- Készítsen egy közvetlen halmazkonstrukciós szubrutint adat ellenőrzött inputtal való beolvasására!

Oldjuk meg a feladatot `Byte` és `Char` elemtípusú halmazokra!

7 ♦ Személy nevén értsünk egy min. 3 max. 30 jel hosszú betűkből és szóközökből álló stringet, amely 2 vagy 3 részből áll, a részeket egy szóköz választja el, és máshol nem lehet szóköz. Az egyes részek kezdőbetűje nagy betű, a többi pozícióban kisbetűk vannak.

- Készítsen közbenső és végellenőrző szubrutint egy adott név helyességének ellenőrzésére!
- Készítsen egy közvetlen halmazkonstrukciós szubrutint név ellenőrzött inputtal való beolvasására!

8 ♦ Dátumokat string típusban `ÉÉÉÉ.HH.NN` (év, hónap, nap) formátumban (hónap és nap előnullázva) adunk meg. Feltételezhetjük, hogy ezek a 1990.01.01–1999.12.31 időszakba esnek.

- Készítsen közbenső és végellenőrző szubrutint egy adott dátum helyességének ellenőrzésére!
- Készítsen egy közvetlen halmazkonstrukciós szubrutint dátum ellenőrzött inputtal való beolvasására!
- Készítsen egy közvetett halmazkonstrukciós szubrutint dátum ellenőrzött inputtal való beolvasására!

9 ♦ Időpontokat string típusban `OO:PP:SS` (óra, perc, másodperc) formátumban (mindegyik előnullázva) adunk meg.

- Készítsen közbenső és végellenőrző szubrutint egy adott időpont helyességének ellenőrzésére!
- Készítsen egy közvetlen halmazkonstrukciós szubrutint időpont ellenőrzött inputtal való beolvasására!

- Készítsen egy közvetett halmazkonstrukciós szubrutint időpont ellenőrzött inputtal való beolvasására!

10 ♦ Egy adat egy legalább 1, legfeljebb 4 jegyű, tizenhatos számrendszerű, előjel nélküli egész, amely nem tartalmazhat vezető nullákat.

- Készítsen közbenső és végellenőrző szubrutint az adat helyességének ellenőrzésére!
- Készítsen egy közvetlen halmazkonstrukciós szubrutint az adat ellenőrzött inputtal való beolvasására!
- Készítsen egy közvetett halmazkonstrukciós szubrutint dátum ellenőrzött inputtal való beolvasására!

11 ♦ Helységkódon értsünk egy mmsssj alakú stringet, ahol:

- mm megyekód 01..20 (előnullázva);
- sss sorszám 000...999 (előnullázva);
- j jellegkód 1..5.

- Készítsen közbenső és végellenőrző szubrutint egy adott helységkód helyességének ellenőrzésére!
- Készítsen egy közvetlen halmazkonstrukciós szubrutint helységkód ellenőrzött inputtal való beolvasására!
- Készítsen egy közvetett halmazkonstrukciós szubrutint helységkód ellenőrzött inputtal való beolvasására!

12 ♦ Fájlazonosítón értsünk n.k alakú stringet, ahol az n és k nagybetű és számjegy jeleket tartalmazhat, valamint:

- n névrész min. 1 és max. 8 jel hosszú;
- k kiterjesztés rész min. 1 és max. 3 jel hosszú.

A kiterjesztés a ponttal együtt el is maradhat (ha kiterjesztés nincs, pont sem lehet).

- Készítsen közbenső és végellenőrző szubrutint egy adott fájlazonosító helyességének ellenőrzésére!
- Készítsen egy közvetlen halmazkonstrukciós szubrutint fájlazonosító ellenőrzött inputtal való beolvasására!
- Készítsen egy közvetett halmazkonstrukciós szubrutint fájlazonosító ellenőrzött inputtal való beolvasására!

13 ♦ Egy speciális adaton egy olyan szót értünk, amely legalább 2 és legfeljebb 20 hosszú:

- nagybetűvel kezdődik;
- a többi jel csak kisbetű lehet;
- két egymás utáni jele nem lehet azonos.

- Készítsen közbenső és végellenőrző szubrutint egy adott adat helyességének ellenőrzésére!
- Készítsen egy közvetlen halmazkonstrukciós szubrutint adat ellenőrzött inputtal való beolvasására!
- Készítsen egy szubrutint adat sorsolással való előállítására!

14 ♦ Egy speciális adaton egy olyan szót értünk, amelyben:

- minden betű pontosan egyszer szerepel;
- a kis és nagybetűk váltakozva szerepelnek (két kisbetű vagy két nagybetű nem állhat egymás mellett).

- Készítsen közbenső és végellenőrző szubrutint egy adott adat helyességének ellenőrzésére!
- Készítsen egy közvetlen halmazkonstrukciós szubrutint adat ellenőrzött inputtal való beolvasására!
- Készítsen egy szubrutint adat sorsolással való előállítására!

15 ♦ Helységnéven értsünk egy min. 2 max. 20 jel hosszú nagybetűkből álló stringet, az alábbi megkötések mellett:

- két azonos magánhangzó nem állhat egymás mellett;
- a név nem állhat csupa magánhangzóból;
- a név nem állhat csupa mássalhangzóból.

- Készítsen közbenső és végellenőrző szubrutint egy adott helységnév helyességének ellenőrzésére!
- Készítsen egy közvetlen halmazkonstrukciós szubrutint helységnév ellenőrzött inputtal való beolvasására!
- Készítsen egy szubrutint helységnév sorsolással való előállítására!

16 ♦ Egy egyszerűsített kifejezésen értünk egy olyan stringet, amely:

- számjeggyel kezdődik és végződik;
- a számok csak számjegyek;
- zárójelek nincsenek;
- csak +, − és * műveleti jelek vannak;
- számjegy után csak művelet jöhet;
- művelet után számjegynek kell jönnie.

- Készítsen közbenső és végellenőrző szubrutint egy adott kifejezés helyességének ellenőrzésére!
- Készítsen egy közvetlen halmazkonstrukciós szubrutint kifejezés ellenőrzött inputtal való beolvasására!
- Készítsen egy szubrutint kifejezés sorsolással való előállítására!

17 ♦ Egy speciális adat egy olyan string, amelyben:

- csak számjegy és nagybetű (`'A'..'Z'`) fordulhat elő;
- külön-külön tekintve a számjegyek és a betűk is egymás között nemcsökkenő sorrendben vannak.

- Készítsen közbenső és végellenőrző szubrutint egy adat helyességének ellenőrzésére!
- Készítsen egy közvetlen halmazkonstrukciós szubrutint adat ellenőrzött inputtal való beolvasására!
- Készítsen egy szubrutint adat sorsolással való előállítására!

18 ♦ Készítsen egy közvetett halmazkonstrukciós szubrutint 5 db egymástól különböző, 1 és 90 közé eső egész szám ellenőrzött inputtal való beolvasására!

19 ♦ Készítsen egy közvetett halmazkonstrukciós szubrutint maximum 20 db, egyenként maximum 5 betűs, csupa ékezet nélküli nagybetűkből álló szó ellenőrzött inputtal való beolvasására! A szavakat csak ábécé sorrendben lehessen beadni. Az adatbevitelnek akkor legyen vége, ha megvan a 20 db szó vagy a `'ZZZZZ'` szót adják be (végjel, nem része az eredménynek).

20 ♦ Készítsen egy közvetett halmazkonstrukciós szubrutint maximum 20 db, egyenként maximum 5 jegyű, pozitív egész szám ellenőrzött inputtal való beolvasására! A számokat csak növekvő sorrendben lehessen beadni. Az adatbevitelnek akkor legyen vége, ha megvan a 20 db szám vagy a `99999` számot adják be (végjel, nem része az eredménynek).

21 ♦ Egy könyv alapvető adatain értünk egy olyan rekordot, amelynek mezői:

a) szerző(k) neve (max. 3 szerző lehet)
b) szerkesztő (2 vagy 3 szerző esetén lehet)
c) cím
d) kiadó neve
e) megjelenés éve
f) ISBN szám (max. 10 jegyű) szám
g) oldalak száma
h) eladási ár

A rekord érvényességéhez az *a), b), c), d), e)* adatok helyes kitöltése szükséges, a többi adat opcionális.

- Készítsen egy végellenőrző szubrutint egy adott rekord helyességének ellenőrzésére!
- Készítsen egy közvetett halmazkonstrukciós szubrutint a rekord ellenőrzött inputtal való beolvasására!

22 ♦ Egy város körül az utakon forgalomszámlálási állomások (kordonpontok) vannak. Egy számlálási időszak egy napon belüli egybefüggő időszak, amely legkorább-

ban reggel 5 órakor kezdődhet és legkésőbben este 22 órakor fejeződhet be. Feltesszük, hogy egy kordonponton egy órán belül 10000 gépjármű haladhat át. A kordonpontok maximális száma 50, azonosításukra sorszámot használunk. A számlálásról kordonpontonként egy adatrekord készül a következő tartalommal:

- a kordonpont azonosítója;
- a számláló személy neve;
- a számlálás kezdete (óra);
- a számlálás vége (óra);
- behaladó járművek száma;
- kihaladó járművek száma;
- átlaghőmérséklet (Celsius fok egész);
- időjárás (derült, borult, esős, havas) a kezdőbetűvel kódolva.

A rekord csak akkor érvényes ha az első 4 adat meg van adva. Ha az 5. vagy 6. adatot nem adják meg vegyünk helyette 0 értéket. Az utolsó két adat megadása nem kötelező.

- Készítsen egy végellenőrző szubrutint egy adott rekord helyességének ellenőrzésére!
- Készítsen egy közvetett halmazkonstrukciós szubrutint a rekord ellenőrzött inputtal való beolvasására!

23 ♦ Egy család alapvető adatain értünk egy olyan rekordot, amelynek mezői:

- az apa neve, születési éve;
- az anya neve, születési éve;
- a gyerekek száma;
- gyerekenként név és születési év.

A rekord érvényességét a mindennapi életnek megfelelő általános formai és tartalmi szabályoknak megfelelően definiáljuk. (Pl. nyilván hiányozhat egy családban az egyik szülő, de nyilvánvaló az is, hogy a gyerek később születik, mint a szülő.)

- Készítsen egy végellenőrző szubrutint egy adott rekord helyességének ellenőrzésére!
- Készítsen egy közvetett halmazkonstrukciós szubrutint a rekord ellenőrzött inputtal való beolvasására!

24 ♦ Egy kisváros közlekedési hálózata max. 52 csomópontot tartalmazhat. A csomópontokat az angol abc kis- és nagybetűivel kódoljuk. Az utcákat pozitív 3 jegyű számokkal kódoljuk. Egy utcát úgy adunk meg, hogy megadjuk az utcakódot és az utca valamelyik végéről indulva, a végighaladási sorrendben minden érintett csomópontra a

- csomópontot;
- a kezdőponttól a csomópontig mért távolságot (pozitív egész).

Az utcamegadás természetesen nem tartalmazhat pontismétlést, valamint csomópontok (a távolság szerint) nem eshetnek egybe.

Az utca adatait egy rekordban tároljuk. A csomópontok sorrendje távolság szerinti a rekordban. A rekord érvényességét is jelezzük vissza. A rekord érvényességéhez az utcakód és legalább 2 csomópont kell.

- Készítsen egy végellenőrző szubrutint egy adott rekord helyességének ellenőrzésére!
- Készítsen egy közvetett halmazkonstrukciós szubrutint a rekord ellenőrzött inputtal való beolvasására!

25 ♦ Egy lakás adatain értünk egy olyan rekordot, amelynek mezői:

- alapterület, szobaszám, havi bérleti díj (kötelező adatok);
- telefonszám (6 jegyű egész, nem kezdődhet nullával) (hiányozhat is).

- Készítsen egy végellenőrző szubrutint egy adott rekord helyességének ellenőrzésére!
- Készítsen egy közvetett halmazkonstrukciós szubrutint a rekord ellenőrzött inputtal való beolvasására!

26 ♦ Egy halmaz `Byte` típusú elemeket tartalmaz. Oldjuk meg az alábbi feladatokat (feladatonként külön szubrutin):

- Keressük meg a halmaz minimális és maximális elemét (`uHalm.MinMax`)!
- Határozzuk meg a halmaz elemszámát (`uHalm.Szamossag`)!
- Válasszuk ki egy nem üres halmaz egy elemét sorsolással (`uHalm.EgyElem`)!
- Állítsuk elő az alaptípus minden, az adott intervallumba eső értékét tartalmazó halmazt (`uHalm.IntHalmaz`)!
- Állítsuk elő az alaptípus minden értékét tartalmazó halmazt (`uHalm.TeljHalmaz`)!
- Állítsuk elő sorsolással egy előírt elemszámú és előírt értékhatárok közé eső elemeket tartalmazó halmazt (`uHalm.HSorsol`)!
- Egy halmazt osszunk sorsolással a lehető legegyenletesebben két részre (a két fél elemszáma közt max. 1 lehet a különbség) (`uHalm.HEloszt`)!

27 ♦ Síkbeli pontokat derékszögű koordinátáikkal adunk meg. A koordináták egy- és kétjegyű nem negatív egész számok lehetnek A pontokból ponthalmazokat képezünk. Készítsük el a ponthalmazok implementációját, logikai mátrixokra alapozva: az (i, j) pont eleme a halmaznak, ha a mátrix i. sor j. oszlopában igaz érték van!

28 ♦ Készítsük el egy egész számokat tartalmazó nagy halmaz implementációját, egy logikai tömbre alapozva! A tömb egy eleme feleljen meg egy halmazelemnek.

29 ♦ Generáljunk sorsolással egy szót! A szó hossza is generált de legalább 3 és legfeljebb 15 legyen, a szó nagybetűvel kezdődjön kisbetűkkel folytatódjon és minden betű különböző legyen. (A nagybetű–kisbetű eltéréstől itt eltekintünk).

30 ♦ Egy szöveget (stringet) jelenként úgy kódolunk, hogy minden jelét egy másik jellel helyettesítjük. Az eredeti jel–kódjel megfeleltetéseket egy kódtáblával adjuk meg. Természetesen a kódtáblával szemben alapkövetelmény az, hogy a string egyértelműen kódolható és dekódolható (visszakódolható) is legyen. Kódoljunk egy stringet, úgy, hogy a kódtáblát is ez az eljárás állítja elő sorsolással!

31 ♦ Egy játékos adott számú de legalább 1 és legfeljebb 10 db szelvénnyel játszik egy héten az 5-ös lottójátékban. Egy szelvényen egy tipp (5 szám) szerepel. Generáljunk egyheti tippelést a következő feltételek mellett:

- legyen két olyan szám, amely minden szelvényen szerepel;
- ne legyen két azonos tipp!

32 ♦ Adott a kísérletek száma, maximum 10000 lehet. Egy kísérlet annak megállapítása, hogy az ötös lottó (5 szám a 90-ből) egy húzásánál hány találatot érünk el. Ehhez mind a kihúzott, mind a tippelt 5 számot sorsolással kell előállítani. Adott még egy kísérlet költsége (a szelvény ára), valamint a 2, 3, 4 és 5 találat nyereménye. Ez az öt adat pozitív, a felsorolás sorrendjében növekvő számérték lehet. Meghatározandó hogy összesen hány 0, 1, 2, 3, 4 és 5 találatot értünk el, valamint a pénzbeli egyenleg (költség–nyereség)!

33 ♦ Egy kártyajátékot négyen játszanak egy 42 lapos kártyával. A játék kezdetén mindenki 9 lapot kap, 6 lap megmarad (talon). A kártyalapokat egyszerűen sorszámokkal azonosítjuk. Állítsunk elő sorsolással egy leosztást!

34 ♦ Egy osztályban max. 50 gyerek van. Adottak a nevek, mint azonosítók. Az osztályteremben kétszemélyes padok vannak. Generáljunk sorsolással egy ülésrendet!

35 ♦ Egy évfolyamon max. 400 hallgató van. Adottak a nevek, mint azonosítók. Generáljunk sorsolással egyenlő létszámú csoportokat! A csoportlétszám adott, maximum 40 lehet. Ha a hallgatók száma nem osztható a kívánt csoportlétszámmal, akkor egy csoportban legyen kevesebb hallgató. Az eredményt csoportonként rendezetten adjuk meg!

36 ♦ Adott két halmaz, mindegyik karakter típusú értékeket tartalmaz. Gyűjtsük ki egy tömbbe a két halmaz közös elemeit!

37 ♦ Adott két tömb, amelyek mindegyike 0 és 99 közti egész értékeket tartalmaz. Gyűjtsük ki egy halmazba azokat az értékeket, amelyek mind a két tömbben előfordulnak!

38 ♦ Egy tömb a `10000..20000` intervallumba eső egész értékeket tartalmaz. Rendezzük a tömböt növekvő irányban:

- Az ismétlődések törlésével (`uHalmAlk.HRendez0`)!
- Az ismétlődések megtartásával (`uHalmAlk.Hrendez1`)!

39 ♦ Egy tömbben XX. századi évszámok vannak, készítsük el a gyakorisági statisztikát!

40 ♦ Sorsolással állítsunk elő egy számsort, amely `M` elemet tartalmaz, és az egyes elemek az `1..N` intervallumba esnek! Az `M` értéke max. 100000, az `N` értéke max. 10 lehet. Gyűjtsük ki egy mátrixba az elemek egymásra következési gyakoriságát! (A mátrix `[I, J]`. eleme megadja, hogy a generálás során hányszor követte az `I` értéket a `J` érték.)

41 ♦ Adottak egy betűből és egy számból álló adatpárok, maximum 100 darab. Egy adatpárban a betű az `'a'..'z'` intervallumba, a szám az 1..5 intervallumba esik. Meghatározandó az adatpárok gyakorisági táblázata (sor: betű, oszlop: szám, elem: az illető adatpár hányszor fordult elő az adatok között). A gyakorisági táblázat a betűk növekvő sorrendje szerint állítandó elő.

5. DINAMIKUS TÖMBÖK

5.1. Általános jellemzés

Az eddigi fejezetekben tárgyalt tömbfogalom *statikus,* tehát a deklarációban rögzítenünk kell a dimenziókat és az egyes dimenziók szerinti alsó és felső indexhatárokat. Ezek az értékek, tehát a tömb által elfoglalt memóriaterület is, csak konstansok lehetnek, a program futása közben nem változhatnak. Következésképpen a statikus tömböt mindig a maximális méretre kell deklarálnunk, hiszen futás közben már nem nyújthatjuk meg. Az aktuális kihasználtságot (elemszámot) egy külön változóban kell nyilvántartanunk.

A *dinamikus tömb* olyan tömb, amely ezt a korlátot nem tartalmazza. A tömb által lefoglalt terület, tehát a maximális elemszám, a program futása közben beállítható, változtatható. Ez például lehetőséget ad a külön aktuális elemszám változó elhagyására is, a maximumnak a mindenkori kihasznált területre való beállításával.

Az Object Pascal programnyelv a dinamikus tömböt standard adattípusként tartalmazza (lásd dynamic arrays), tehát elegendő a megfelelően deklarálni, és ezután a közönséges tömbök módjára lehet használni, kivéve természetesen a maximális elemszám beállítását és lekérdezését, ami a típushoz rendelt standard eszközökkel végezhető.

A Turbo Pascal programnyelvben nincs dinamikus tömb adattípus, de példaképpen az alábbiakban megadunk egy implementációt. Egy ilyen bonyolultabb implementációs feladat megoldása egyrészt, mint modellezési feladat is érdekes, másrészt még inkább világossá teszi számunkra azt (ami valójában a fejlesztő környezetben implementált bonyolultabb standard típusokra is érvényes), hogy egy típus jellemzőit teljes pontossággal csak az implementációjából ismerhetjük meg.

5.2. Implementáció

Az implementációban csak a rendszerben standard módon meglévő eszközöket használunk fel. A nyelv sajátosságainak megfelelően, az implementációt megtestesítő szoftver erőforrásokat egy unit formájában adjuk meg. Az implementálandó dinamikus tömb egydimenziós lesz, egész (`Integer`) típusú elemekkel.

A fejlesztő környezet sajátosságaiból adódnak az alábbi következmények, részben kötöttségek:

- A tömb elemeit (vagyis az adatstruktúra szűkebb értelemben vett tömb részét) a dinamikus adatterületen, tehát a *Heap* szegmensben, dinamikus változók formájában kell elhelyeznünk.
- A lefoglalt terület változását csak az adatok fizikai áthelyezésével tudjuk biztosítani.
- A tömbelemek számának felső korlátját az egy változó által maximálisan elfoglalható terület határozza meg.
- A tömbelemek mellett az aktuálisan lefoglalt terület méretét is kezelnünk kell az implementáción belül, a különféle adatokat egy `record` típusú változóban foghatjuk össze.

A kötöttségek mellett választási lehetőségeink is vannak. A legfontosabb (előre nem determinált) döntési pont a modellalkotásban az, hogy hogyan kezeljük az aktuális elemszám (tehát a lefoglalt terület) *csökkenésének* esetét. Erre két megoldási mód is kínálkozik:

- Ha az elemszám csökken, csökkentjük a területet is. A megoldási mód jellemzői:
  - A területfoglalás mindenkor csak a minimálisan szükséges (előny).
  - Nem csak bővítésnél, de szűkítésnél is fizikailag át kell helyeznünk az összes tömbelemet (hátrány).
- Az elemszám csökkenésekor meghagyjuk az eredetileg lefoglalt területet. A megoldási mód jellemzői:
  - A területfoglalás mindig az addigi tényleges felhasználás maximuma (hátrány).
  - Csak bővítésnél kell fizikailag áthelyeznünk a tömbelemeket (előny).

Az, hogy melyik megoldás a jobb általánosan nem mondható meg, a konkrét alkalmazástól (a tömbméret csökkenésének gyakoriságától) függ. Az implementációs példában a második módszert választjuk.

Az implementációs unitunk az `uDinTomb` unit. Ehhez fűzünk az alábbiakban magyarázatot. A teljesebb megértéshez javasoljuk a programszöveg tanulmányozását is.

```
const
  DEDbMax=32760;  {maximális elemszám (max. 65520 byte
    lehet a helyfoglalás)}
type
  TDElem=Integer;  {elemtípus}
  TDEDb=0..DEDbMax;  {elemszámtípus}
  TDEIndex=1..DEDbMax;  {indextípusok}
  TDEIndex0=0..DEDbMax;
  TDEIndex1=1..DEDbMax+1;
  TDTomb=array[TDEIndex] of TDElem;  {tömb alaptípus}
  PDTomb=^TDTomb;  {tömb mutatótípus}

  {a dinamikus tömb típusa}
  TDinTomb=record
    Elemek: PDTomb;  {tömbmutató}
    ADb: TDEDb;  {aktuális elemszám}
    FoglDb: TDEDb;  {aktuálisan használható elemszám}
  end;
```

A `TDTomb` típus a lehető legnagyobb ilyen statikus tömb típusa. Ez a megfelelő, de ennél mindig kisebb vagy maximum egyenlő helyfoglalású dinamikus változó `GetMem` utasítással való létrehozásához kell. A rekordon belül az `Elemek` ennek a dinamikus változónak a mutatója. Mivel a területkezelésnél a második módszert választottuk kezelnünk kell az aktuálisan lefoglalt területhez tartozó `FoglDb` elemszámot, ami lehet nagyobb is mint az aktuálisan értelmezett `ADb` elemszám.

A rekord belseje (`Elemek`, `ADb`, `FoglDb`) az implementációhoz tartozik, a típus használójának ezekről tudni sem kell, az adatstruktúrát úgy építheti be a programjába, hogy:

- Hivatkozik az `uDinTomb` unitra (`uses`).
- Deklarál `TDinTomb` típusú változó(ka)t.
- Meghívja a típushoz adott műveleti szubrutinokat.

A műveleti szubrutinok:

```
{inicializálás}
procedure DTIndit(var A: TDinTomb);
```

Kötelező meghívni az `A` további használata előtt. Beállítja az implementáció belső változóinak kezdőértékeit. Az aktuális elemszám 0.

```
{aktuális elemszám beállítás}
function DTHossz(var A: TDinTomb; EDb: TDEIndex): Boolean;
```

Beállítja az `EDb` elemszámot, lefoglalja a szükséges területet. Ezután értelmezettek az `1..EDb` indexű elemek. Ha a függvényérték hamis, a beállítás nem hajtható végre (szabad memória híján).

```
{aktuális elemszám lekérdezés}
function DTEDb(const A: TDinTomb): TDEDb;

{(létező) I. elemnek értékadás}
procedure DTBe(var A: TDinTomb; I: TDEIndex; Ertek: TDElem);
```

Az `A[I]:=Ertek` közönséges értékadás megfelelője. Ha az `I` index nem értelmezett, programhibát válthat ki.

```
{(létező) I. elem értéke}
function DTErt(const A: TDinTomb; I: TDEIndex): TDElem;
```

Az `A[I]` közönséges hivatkozás megfelelője. Ha az `I` index nem értelmezett, programhibát válthat ki.

```
{zárás}
procedure DTZar(var A: TDinTomb);
```

Ajánlott meghívni az `A` használatának befejezése után. Felszabadítja a lefoglalt területet.

A jelen megvalósításban (`uDinTomb`) nem akadályozható meg, hogy a típus felhasználója „illegálisan” közvetlenül kezelje az implementációs változókat (`Elemek`, `ADb`, `FoglDb`), hiszen magát a `TDinTomb` deklarációt a kívülről is elérhető (public) részbe kell tennünk. Csak megjegyezzük, hogy objektumorientált realizációban lehetőség van az implementációs változók tényleges elrejtésére is (private rész).

5.3. Mintapéldák

A statikus és dinamikus tömböket kezelő algoritmusok alapjában véve csak az aktuális elemszám kezelésében térnek el. Ezért külön feladatsort nem közlünk, az alábbiakban csak néhány mintapéldát adunk a fenti implementációhoz, struktúradiagramok nélkül.

5.3.1. mintafeladat: Keressünk meg egy adott értéket egy tömbben!

Adatszerkezet (37)

Azonosító	Funkció	Típus	Jelleg
`Miben`	ebben keresünk	`TDinTomb`	input
`Mit`	a keresett érték	`TDElem`	input
`Hol`	a `Mit` helye, ha létezik	`TDEIndex1`	output
`Dkeres`	a `Mit` létezése	`Boolean`	output

Szubrutin: `uTombD.DKeres`.

Megjegyzés ♦ Több `Mit` érték esetén a `Hol` az első indexét adja. Sikertelen keresésnél a `Hol` az utolsó elem utánra mutat.

5.3.2. mintafeladat: Szúrjunk be egy adott értéket egy tömb adott helyére!

Útmutató ♦ A balra igazított tárolásnak megfelelően a tömb a nagyobb indexek felé bővül. Az adott helytől kezdődően jobbra léptetjük az elemeket.

Adatszerkezet (38)

Azonosító	Funkció	Típus	Jelleg
`Mibe`	ebben szúrunk be	`TDinTomb`	input, output
`Mit`	a beszúrandó érték	`TDElem`	input
`Hova`	a `Mit` helye	`TDEIndex`	output
`DBeszur`	a művelet végrehajthatósága	`Boolean`	output
`I`	index	`TDEIndex`	output
`Jo`	a művelet végrehajthatósága	`Boolean`	munka

Szubrutin: `uTombD.DBeSzur`.

Megjegyzés ♦ A végrehajtás lehetetlenségét memóriahiány okozhatja (`uDinTomb.DTHossz`).

5.3.3. mintafeladat: Töröljük a tömb egy elemét!

Útmutató ♦ Az adott helytől kezdődően balra léptetjük az elemeket.

Adatszerkezet (39)

Azonosító	Funkció	Típus	Jelleg
`Mibol`	ebből törlünk	`TDinTomb`	input, output
`Honnan`	a törlendő elem helye	`TDEIndex`	input
`I`	index	`TDEIndex`	munka

Szubrutin: `uTombD.DTorol`.

Megjegyzés ♦ Az elemtörlés nem csökkenti a lefoglalt helyet (`uDinTomb.DTHossz`).

5.3.4. mintafeladat: Rendezzünk egy tömböt növekvően (minimum-kiválasztás módszere)!

Útmutató ♦ Minden lépésben a rendezetlen rész minimális értékű elemét felcseréljük a rész első elemével (vagyis a rendezett részt növeljük, a rendezetlent csökkentjük 1 elemmel).

Adatszerkezet (40)

Azonosító	Funkció	Típus	Jelleg
`Adatok`	a rendezendő/rendezett tömb	`TDinTomb`	input, output
`SorKezd`	a rendezetlen rész kezdete	`TDEIndex`	munka
`MinHely`	a minimum helye a rendezetlen részben	`TDEIndex`	munka
`I`	index	`TDEIndex`	munka
`Mini`	minimális érték a rendezetlen részben	`TDElem`	munka

Szubrutin: `uTombD.DKivalRend`.

5.3.5. mintafeladat: Sorsolással állítsunk elő egy adott határok közé eső elemszámú és adott határok közé eső értékű, egész számokból álló számsort. Határozzuk meg az alábbi statisztikai jellemzőket:

- *elemszám, minimum, maximum,*
- *medián (a rendezett számsorban a középső indexnél álló érték).*

Az eredményt csak a jellemzők képezik (magára a számsorra, mint eredményre nincs szükség).

Útmutató ♦ A dinamikus tömb munkaváltozó lesz, amit sorsolással feltöltünk, majd rendezünk. A rendezett alakból leolvashatók az eredmények.

Adatszerkezet (41)

Azonosító	Funkció	Típus	Jelleg
`NTol`	az elemszám alsó határa	`TDEIndex`	input
`NIg`	az elemszám felső határa	`TDEIndex`	input
`ETol`	az értékek alsó határa	`TDElem`	input
`EIg`	az értékek felső határa	`TDElem`	input
`N`	a sorsolt elemszám	`TDEdb`	output
`Min`	az értékek minimuma	`TDElem`	output
`Max`	az értékek maximuma	`TDElem`	output
`Med`	az értékek mediánja	`TDElem`	output
`A`	a sorsolt értékek tárolója	`TDinTomb`	munka
`I`	index	`TDEIndex`	munka

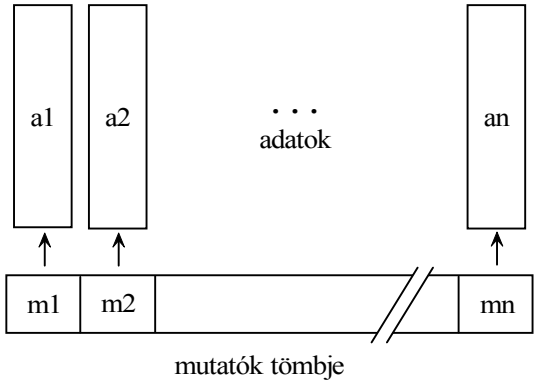
Szubrutin: `uTombD.DStatJell`.

Megjegyzés ♦ Az `N = 0` eredmény azt jelzi, hogy a feladat nem volt végrehajtható. Ezt memóriahiány okozhatja (`uDinTomb.DTHossz`).

6. KOLLEKCIÓK

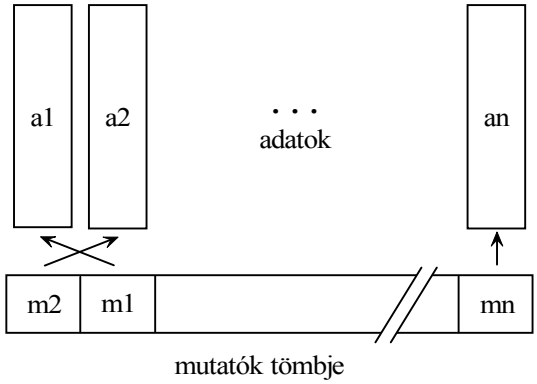
6.1. Általános jellemzés

A *kollekció* a tömb általánosítása. Szerkezetileg meghatározó része egy tömb, amely mutatókat tartalmaz, nevezzük ezt *mutatótömbnek*. A mutatók által hivatkozott adatok a kollekció *tételei* (angol terminológiában: item), ezek alkotják a kollekció adatrészét. A mutatótömb és a tételek együttese a kollekció. (24. ábra)



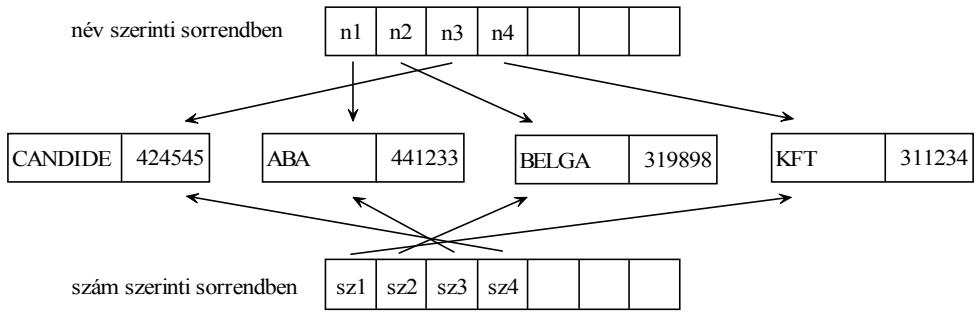
24. ábra. Kollekció

A kollekció megtartja a *tömbszerűséget* vagyis az ilyen adatcsoportoknak (tömb, dinamikus tömb) azt a nagyon előnyös tulajdonságát, hogy a csoport elemei közvetlen hivatkozással, *indexezéssel* elérhetők.



25. ábra. Mutatók felcserélése

A kollekció adatstruktúra „beépítve” adja *logikai* és fizikai sorrend elválasztásának, a minimális adatmozgatással való *sorrendcserének* azt a lehetőségét, amit a tömböknél az indextábla segítségével tudtunk megvalósítani. Két tétel sorrendjének felcseréléséhez nyilván elegendő csak a megfelelő mutatókat felcserélni. (25. ábra) Ugyanarra a tételsorra a mutatók különböző sorrendjével is hivatkozhatunk. A 26. ábra tehát két kollekciót mutat, amelyek adatrésze azonos, az egyik mutatótömb a név szerinti, a másik a szám szerinti sorrendet mutatja.



26. ábra. Azonos adatrészű kollekciók

A kollekció az előnyök megtartása mellett ki is bővíti a tömb lehetőségeit. Minden fejlesztőrendszerben korlátozott az egy változóval, következésképpen az egy tömbként kezelhető adatterület nagysága. (Ez a dinamikus tömbre is ugyanúgy érvényes). A kollekció ezt a korlátot lényegesen, nagyságrendekkel megemeli, hiszen itt az eredeti korlát *külön-külön* és egyenként vonatkozik a mutatótömbre és az egyes tételekre. (Turbo Pascalban az egy tömbben kezelhető bájtok maximális száma `65520`, míg a kollekcióra ez a korlát `16380 * 65520`, lévén, hogy egy cím, tehát a mutatótömb egy eleme `4` bájtot foglal el. Az Object Pascal `32` bites alapú verzióiban már maga az alapkorlát is igen nagy – a mai személyi számítógépek memóriakapacitásához viszonyítva – így a kollekció ezen előnye Delphiben csak a maiaknál lényegesen nagyobb kapacitású számítógépeken mutatkozna meg.)

A kollekció megvalósításához mind a Turbo Pascal, mind az Object Pascal tartalmazza a megfelelő eszközöket. A mutatótömb elemei típusos mutatók, a tételek pedig *dinamikus változók* lesznek. A mutatótömb lehet akár statikus, akár dinamikus tömb. Mivel egy tömbben csak azonos típusú elemek lehetnek, a mutatók azonos típusú mutatók, következésképpen a tételek is mind csak azonos típusú változók lehetnek. (Ez az alapeset, ami objektumorientált fejlesztésnél – a szélesebb körű típuskompatibilitás eredményeképpen – túlléphető, egy

kollekcióban eltérő típusú objektumok is tárolhatók. Itt csak az alapesettel foglalkozunk.)

A kollekció adatstruktúra Pascal realizációja már önmagában egy *dinamikus adatszerkezet,* nemcsak tartalmilag, hanem szerkezetileg és helyfoglalás szempontjából is változhat a program futása folyamán. Hiszen a tételek dinamikus változók, az aktuálisan nemlétező tételek esetén maguk a tételek nem foglalnak helyet (24. ábra). Ha a mutatótömb statikus, akkor ugyan a nemlétező tételek mutatói is foglalják a helyet, de ez általában elhanyagolható maguknak a tételeknek a helyfoglalásához képest. Emiatt a tárgazdálkodás szempontjából a kollekciók esetén kevésbé lényeges, hogy a mutatótömb statikus vagy dinamikus tömb.

A Turbo Pascal fejlesztőrendszerben a kollekció nem standard adattípus a mutatótömb deklarációjának segítségével hozható létre. A Delphi fejlesztőrendszer több olyan objektumtípust és komponenstípust is tartalmaz (pl. `TList`), amely szerkezetileg kollekció.

6.2. Alapfeladatok

A kollekciók alapvető karbantartási és lekérdezési műveleti algoritmusai szerkezetileg megegyeznek a tömbökével, hiszen a kollekciót szerkezetileg a mutatótömb határozza meg. Ami eltér, az egyrészt az elemekre való hivatkozás módja, másrészt új teendőként jelentkezik a tételek, mint *dinamikus* változók kezelése.

Mint minden dinamikus adatszerkezetnél, itt is, az adatszerkezet *bővítésének* (beszúrás) lépései (43. struktúradiagram):

- Az új elem létrehozása:
  - A szükséges hely meglétének (a bővítés lehetőségének) ellenőrzése.
  - A dinamikus változó létrehozása, ha van hely.
- Ha van új elem:
  - A dinamikus változó feltöltése adatokkal.
  - Az új elem beillesztése a struktúrába.

A programnyelvi megvalósításban a bővítési műveletnek az új elemet létrehozó része, a helyellenőrzés alapvetően eltérő módja miatt, fejlesztőrendszertől (sőt ezen belül még verziótól) is függ. Ezért ezt külön szubrutinként programozzuk. A szubrutin a dinamikus változó mutatóját adja vissza, ennek (univerzálisan használható) `nil` értéke jelzi azt, hogy az új elem nem jöhet létre (helyhiány miatt). A törlésnél ilyen probléma nem lép fel, ez egységesen programozható.

A mondottakat egy stringekből, mint tételekből álló, maximum `10000` tételt tartalmazó kollekció példáján szemléltetjük. Az alapdeklarációk (lásd `uSKollD` unit):

```
const
  MaxSTetDb=10000;  {maximális tételszám}
type
  {indextípusok}
  TSTetI=1..MaxSTetDb; TSTetI0=0..MaxSTetDb;
  TSTetI1=1..MaxSTetDb+1;
  TSTetDb=0..MaxSTetDb; {darabszám típus}
  TSTet=String;  {tétel adattípus}
  PSTet=^TSTet;  {tétel mutatótípus}
  TSKoll=array[TSTetI] of PSTet;  {mutatótömb típus}
```

Ha már van egy `SKoll` nevű mutatótömb változónk, akkor:

- Hivatkozás az `I`-edik tételre: `SKoll[I]^`.
- Hivatkozás az `I`-edik tétel (string) `J`-edik jelére: `SKoll[I]^[J]`.

Az új elemet létrehozó szubrutinok: `tSKollU.UjSTetel` és `dSKollU.UjSTetel`. (Minden olyan programmodulban, amely új elemet is hoz létre, a fejlesztő rendszertől függően kell cserélni a **`uses`** `tSKollU` és **`uses`** `dSKollU` hivatkozást.)

Példaképpen nézzük a legalapvetőbb feladatokat, a lekérdezést, a beszúrást és a törlést:

6.2.1. mintafeladat: Keressünk meg egy adott értéket a kollekcióban!

Adatszerkezet (42)

Azonosító	Funkció	Típus	Jelleg
`Miben`	ebben keresünk	`TSKoll`	input
`Hanyban`	a `Miben` aktuális elemszáma	`TSTetDb`	input
`Mit`	a keresett érték	`TSTet`	input
`Hol`	a `Mit` helye, ha létezik	`TSTetI1`	output
`Skeres`	a `Mit` létezése	`Boolean`	output

Szubrutin: `uSKoll.SKeres`.

Megjegyzés ♦ Több `Mit` érték esetén a `Hol` az első indexét adja. Sikertelen keresésnél a `Hol` az utolsó elem utánra mutat.

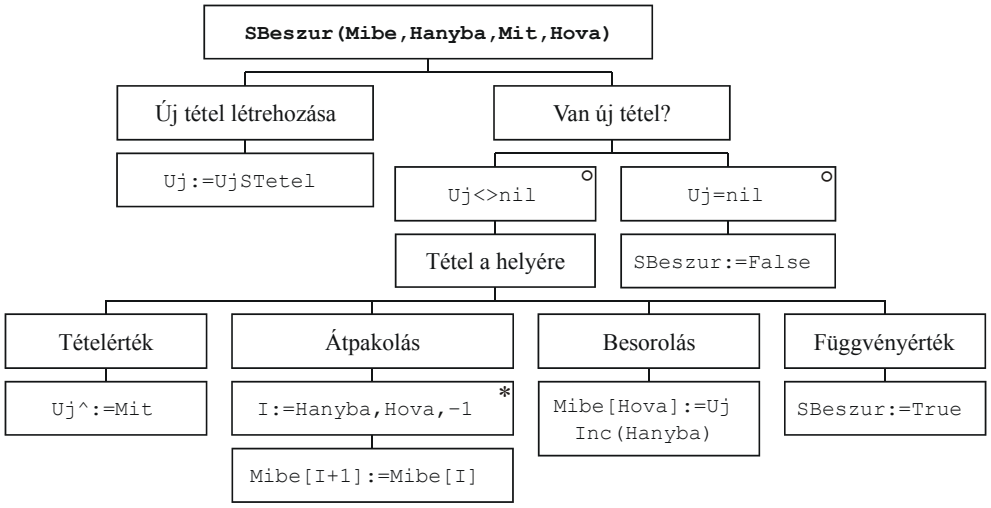
6.2.2. mintafeladat: Szúrjunk be egy adott értéket egy kollekció adott helyére!

Útmutató ♦ A balra igazított tárolásnak megfelelően a kollekció a nagyobb indexek felé bővül. Az adott helytől kezdődően jobbra léptetjük a mutatókat. Azt tételezzük fel, hogy a mutatótömb bővíthető.

Adatszerkezet (43)

Azonosító	Funkció	Típus	Jelleg
`Mibe`	ebben szúrunk be	`TSKoll`	input, output
`Hanyba`	a `Mibe` aktuális elemszáma	`TSTetDb`	input, output
`Mit`	a beszúrandó érték	`TSTet`	input
`Hova`	a `Mit` helye	`TSTetI`	input
`SBeszur`	a művelet végrehajthatósága	`Boolean`	output
`I`	index	`TSTetI`	munka
`Uj`	az új tétel mutatója	`PSTet`	munka

Struktúradiagram (43)



Szubrutin: `uSKoll.SBeSzur`.

Megjegyzés ♦ A végrehajtás lehetetlenségét memóriahiány okozhatja (`UjSTetel`).

A törlésnek (ezzel is általánosítva a tömbfogalmat) a kollekcióknál már legalább két formája, lehetősége van:

- *Törlés:* A tételt töröljük a kollekcióból, de a megfelelő dinamikus változó megmarad. (A programban még szükség van rá, pl. egy másik kollekcióban is szerepel). Ez a mutatótömb egy elemének törlése.
- *Felszámolás:* A tételt töröljük a kollekcióból, és a megfelelő dinamikus változót is megszüntetjük.

6.2.3. mintafeladat: Töröljük a kollekció egy tételét!

Adatszerkezet (44)

Azonosító	Funkció	Típus	Jelleg
`Mibol`	ebből törlünk	`TSKoll`	input, output
`Hanybol`	a `Mibol` aktuális elemszáma	`TSTetDb`	input, output
`Honnan`	a törlendő elem helye	`TSTetI`	input
`I`	index	`TSTetI`	munka

Szubrutin: `uSKoll.STorol`.

6.2.4. mintafeladat: Számoljuk fel a kollekció egy tételét!

Útmutató ♦ Az egyszerű törléstől csak a felszabadításban különbözik.

Szubrutin: `uSKoll.SFelszamol`.

6.3. Rendezett kollekciók

Rendezett kollekcióról akkor beszélhetünk, ha magukra a tételekre értelmezett az összehasonlítás (pl. stringek, számok), vagy tételekhez valamilyen eljárással rendelünk ilyen értékeket. Ezeket az értékeket nevezzük a tétel értékének. A kollekció akkor rendezett, ha a mutatók fizikai sorrendjében a tételértékek nagysága nem csökken (ezt nevezzük *növekvő* rendezettségnek, 26. ábra) vagy nem nő (ezt nevezzük *csökkenő* rendezettségnek).

A kollekciós rendezési és keresési algoritmusok készítésének alapszabályai:

- *A tételekre a mutatón keresztül hivatkozunk.*
- *A tételek helyett a mutatók mozognak.*

Ebben az adatstruktúrában a rendezett beszúrás úgy oldható meg, hogy az *új tételt* mint dinamikus változót létrehozzuk, mutatóját pedig új elemként beszúrjuk a mutatótömbbe, a tétel értékének megfelelő helyre. Két tétel sorrendjének megváltoztatásához elegendő mutatóiknak a tömbben való felcserélése (25. ábra). Mintapéldaként veszünk egy egyszerű rendezést és a bináris keresést. Az algoritmusok szerkezetileg megegyeznek a megfelelő tömbkezelő algoritmusokkal, ezért struktúradiagramokat nem közlünk.

A példák a fentebb definiált stringkollekcióra vonatkoznak (lásd még `uSKollD unit`).

6.3.1. mintafeladat: Rendezzünk egy kollekciót növekvően, a kiválasztás módszerével!

Adatszerkezet (45)

Azonosító	Funkció	Típus	Jelleg
`Adatok`	a rendezendő kollekció mutatótömbje	`TSKoll`	input, output
`AdatDb`	a kollekció aktuális elemszáma	`TSTetDb`	input
`SorKezd`	a rendezetlen rész kezdete	`TSTetI`	munka
`MinHely`	a minimum helye a rendezetlen részben	`TSTetI`	munka
`Mini`	minimális érték a rendezetlen részben	`TSTet`	munka
`I`	index	`TSTetI`	munka
`Cs`	segédváltozó a mutatók cseréjéhez	`PSTet`	munka

Szubrutin: `uSKoll.SKivalRend`.

6.3.2. mintafeladat: Keressünk meg egy adott értéket egy növekvően rendezett kollekcióban, bináris kereséssel!

Adatszerkezet (46)

Azonosító	Funkció	Típus	Jelleg
`Miben`	ebben keresünk	`TSKoll`	input
`Hanyban`	`Miben` aktuális elemszáma	`TSTetDb`	input
`Mit`	a keresett érték	`TSTet`	input
`Hol`	a `Mit` helye	`TSTetI1`	output

Azonosító	Funkció	Típus	Jelleg
`SBinKer`	a `Mit` létezése	`Boolean`	output
`Kezd`	aktuális kezdőindex	`TSTetI1`	munka
`Veg`	aktuális végindex	`TSTetI0`	munka
`Van`	a `Mit` létezése	`Boolean`	munka

Szubrutin: `uSKoll.SBinKer`.

6.4. Mintapéldák

A kollekció alkalmas arra is, hogy a háttértárolókon tárolt lineáris jellegű adatstruktúrák (a Pascal nyelvben ilyenek a szövegfájlok, a típusos és típus nélküli fájlok) tartalmát az operatív memóriában tárolja. Ha a tárkapacitás megengedi célszerű az a munkamenet, amelynek során:

- először a fájl tartalmát egy kollekcióba betöltjük,
- majd a szükséges feldolgozásokat a kollekción elvégezzük,
- végül a fájlt (ha szükséges) a kollekcióból újraírjuk,

mivel a háttértáron való feldolgozás nagyságrendekkel lassúbb, mint az operatív tárbeli.

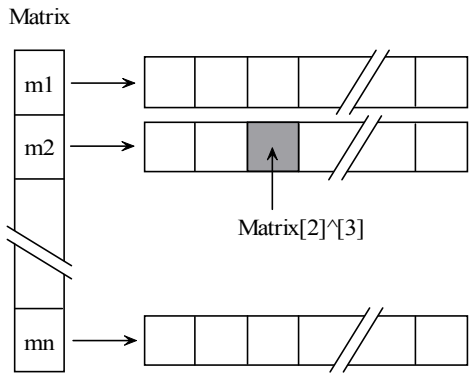
6.4.1. mintafeladat: Egy szövegfájl maximum 255 jel hosszú sorokat tartalmaz. A sorok maximális száma 10000. Rendezzük a fájl sorait növekvően!

Útmutató ♦ A fájból létrehozunk egy megfelelő stringkollekciót (lásd `uSKollD`). A kollekciót rendezzük, majd belőle újraírjuk a fájlt. A beolvasást, rendezést és a kiírást külön szubrutinban adjuk meg. A beolvashatóság feltétele az elegendő memória. A nem használt mutatók definiálatlanok maradnak. A visszaírás a mi megoldásunkban fel is számolja a kollekciót, általában ez nem szükségszerű. A rendezést az előző pontban tárgyaltuk. A feladat teljes megoldását a három szubrutin megfelelő keretben történő meghívása jelenti. Feltételezzük, hogy a keretprogram a fájl létezését is ellenőrzi.

Szubrutinok: `uSKoll.SKollBe`, `uSKoll.SKivalRend`, `uSKoll.SKollKi`.

Egy más jellegű, szintén nem túl bonyolult alkalmazása a kollekciónak a *nagy mátrixok* kezelése. A mátrix „nagyságán” itt azt értjük, hogy az adott fejlesztőrendszerben (az egy változóval kezelhető maximális adatterületre vonatkozó korlát túllépése miatt) nem deklarálható olyan tömb, amelyben tárolni tudnánk

az adatokat. A kollekció segítségével mégis elérhetjük a mátrix-szerű kezelést, tehát az elemek egy sor és oszlopindexszel való egyszerű hozzáférhetőségét. A kollekció létrehozásának feltétele az hogy maga a mutatótömb, valamint egy sor (vagy oszlop) már tárolható legyen egy változóban. A kollekció tételei a sorok (vagy oszlopok) lesznek (27. ábra).



27. ábra. Mátrixkollekció

Példaként deklaráljunk egy max. `1000*1000` méretű, egész értékeket tartalmazó mátrixot, mint a sorok kollekcióját (`uMKollD`).

```
const
  MaxMSorDb=1000;  {maximális sorszám (tételszám)}
  MaxMOszlDb=1000;  {maximális elemszám soronként}
type
  {indextípusok}
  TMSorI=1..MaxMSorDb;  TMOszlI=1..MaxMOszlDb;
  {darabszám típusok}
  TMSorDb=0..MaxMSorDb;  TMOszlDb=0..MaxMOszlDb;
  TMSor=array[TMOszlI] of Longint;  {sor (tétel) adattípus}
  PMSor=^TMSor;  {sor (tétel) mutatótípus}
  TMKoll=array[TMSorI] of PMSor;  {mutatótömb típus}
```

6.4.2. mintafeladat: Egy táblázat nemnegatív egész értékeket tartalmaz. A sorok és oszlopok száma azonos, maximum 1000. Oldjuk meg az alábbi feladatokat (feladatonként egy szubrutinban):

- *Hozzuk létre a táblázatot adott méretre!*
- *Töltsük fel a táblázatot adott méretre és adott értékhatáron belül sorsolt értékekkel!*

- *Bővítsük a táblázatot egy újabb sorral, amely az oszlopösszegeket tartalmazza, valamint egy újabb oszloppal, amely a sorösszegeket tartalmazza!*
- *Töröljük a táblázat egy adott indexű sorát!*

Útmutató ♦ A fenti típusokat alkalmazzuk. A létrehozás és hivatkozás formájától eltekintve egyszerű mátrixkezelő algoritmusokról van szó, ezért struktúradiagramokat nem közlünk. Az új tételek létrehozására, a stringkollekcióhoz hasonlóan, két szubrutint alkalmazunk: `dMKollU.UjMTetel`, `tMKollU.UjMTetel`.

Hozzuk létre a táblázatot adott méretre!

Adatszerkezet (47)

Azonosító	Funkció	Típus	Jelleg
`MKoll`	az új kollekció mutatótömbje	`TMKoll`	output
`N`	a létrehozandó tételek száma	`TMSorDb`	input
`UjMKoll`	az új kollekció létrejötte	`Boolean`	output
`I`	tételszámláló	`TMSorDb`	munka
`VanHely`	van-e hely az új tételhez	`Boolean`	munka
`UjSor`	az új tétel mutatója	`PMSor`	munka

Szubrutin: `uMKoll.UjMKoll`.

Megjegyzés ♦ A kollekció létrejöttének feltétele csak az, hogy a tételek mint dinamikus változók számára legyen hely.

Töltsük fel a táblázatot adott méretre és adott értékhatáron belül sorsolt értékekkel!

Adatszerkezet (48)

Azonosító	Funkció	Típus	Jelleg
`MKoll`	a kollekció mutatótömbje	`TMKoll`	input
`N`	a feltöltés mérete	`TMSorDb`	input
`K`	az értékek felső korlátja	`Longint`	input
`I`	sorindex	`TMSorI`	munka
`J`	oszlopindex	`TMOszlI`	munka

Szubrutin: `uMKoll.ToltMKoll`.

Megjegyzés ♦ A szubrutinnak csak input paraméterei vannak. A változás a kollekció tételein belül megy végbe, ezek pedig nem jelennek meg a paraméterlistán. Maguk a mutatók nem változnak.

Bővítsük a táblázatot egy újabb sorral, amely az oszlopösszegeket tartalmazza, valamint egy újabb oszloppal, amely a sorösszegeket tartalmazza!

Adatszerkezet (49)

Azonosító	Funkció	Típus	Jelleg
`MKoll`	a kollekció mutatótömbje	`TMKoll`	input, output
`N`	a táblázat mérete	`TMSorDb`	input, output
`BovMKoll`	a bővítés létrejötte	`Boolean`	output
`UjSor`	az új tétel mutatója	`PMSor`	munka
`I`	sorindex	`TMSorI`	munka
`J`	oszlopindex	`TMOszlI`	munka
`X`	teljes összeg	`Longint`	munka

Szubrutin: `uMKoll.BovMKoll`.

Megjegyzés ♦ A bővítés létrejöttének feltétele csak az, hogy az új tétel mint dinamikus változó számára legyen hely. Azt ellenőrzés nélkül feltételezzük, hogy a kiinduló méret kisebb, mint a maximális, tehát a táblázat legalább egy sorral és oszloppal még növelhető.

Töröljük a táblázat egy adott indexű sorát!

Adatszerkezet (50)

Azonosító	Funkció	Típus	Jelleg
`MKoll`	a kollekció mutatótömbje	`TMKoll`	input, output
`N`	a táblázat mérete	`TMSorDb`	input, output
`TI`	a törlendő sor indexe	`TMSorI`	input
`I`	sorindex	`TMSorI`	munka

Szubrutin: `uMKoll.TorlSorMKoll`.

Megjegyzés ♦ Először meg kell szüntetni a tételt, mint dinamikus változót, majd ki kell törölni a mutatóját a mutatótömbből.

6.5. Feladatok

1 ♦ Nagy mátrixon értünk itt egy olyan mátrixot, amely megfelel a következő feltételeknek:

- elemtípusa `Byte`;
- sorainak száma max. `10`;
- oszlopainak száma max. `20000`.

Oldjuk meg az alábbi feladatokat (feladatonként külön szubrutin):

- Mátrix létrehozása és sorsolással való feltöltése!
- Mátrix kimentése szövegfájlba!
- Mátrix visszatöltése szövegfájlból!

2 ♦ A feladat egy szövegfájl sorainak szétválogatása. Egy sor egy max. 80 jel hosszú string. A szövegfájl maximum 1000 sort tartalmazhat. A feladat pontos definiálásához vezessük be a következő elnevezéseket:

- Egy szövegrésznek nevezzük egy string tetszőleges összefüggő részét.
- Egy szó tágabb értelemben véve bent van a szövegfájl egy sorában, ha a sor tartalmazza a keresett szót, mint szövegrészt.
- Egy szó szigorú értelemben véve bent van a szövegfájl egy sorában, ha a sor tartalmazza a keresett szót, mint szót.

Adott keresendő szó és a keresési mód (tágabb, szigorú) alapján képezzünk egy új fájlt a kiinduló fájl egyes soraiból! Az új fájlba belekerülnek a keresendő szót a megadott módon tartalmazó sorok. Az új fájlok formátumban és rendezettségében megegyezik az eredetivel. Az eredeti fájl változatlan marad.

3 ♦ Bizonyos szövegfájlokra a következő megkötések érvényesek:

- a sorok száma maximum 9000;
- minden sor azonos hosszú;
- a maximális sorhossz 10 jel;
- a sorokban lévő jelek csak számjegyek lehetnek.

Oldjuk meg az alábbi feladatokat (feladatonként külön szubrutin):

- Előállítás és sorsolással való feltöltés, adott sorszám és sorhossz mellett.
- Adott számértékű sorok megkeresése (létezik-e, és ha igen, hány darab).
- Adott sorszámú sorok törlése (a törlendő rész első és utolsó sorának számát adjuk meg).
- A minimális és maximális számértékű sor kikeresése.

4 ♦ Adottak pontok és köztük lévő távolságok. A távolságok egész méterben mért értékek. Egy távolság min. 0, max. 1000 méter. A pontokat egyszerűen egy sorszámmal azonosítjuk, a pontok maximális darabszáma 1000.

Távolságtáblázaton egy olyan mátrixot értünk, amely rendelkezik az alábbi tulajdonságokkal:

- négyzetes és szimmetrikus;
- az i-edik sor j-edik oszlopa az i-edik és j-edik pont távolságát tartalmazza. A főátlóban minden elem 0 (önmagától vett távolság).

Oldjuk meg az alábbi feladatokat (feladatonként külön szubrutin):

- A táblázat létrehozása és feltöltése sorsolt adatokkal.
- A két, egymástól maximális távolságra eső pont kiválasztása.
- A két, egymástól minimális távolságra eső pont kiválasztása.
- A centrális pont kiválasztása. Ezen azt a pontot értjük, amelynek a többitől vett átlagos távolsága minimális az ilyen átlagok között.

5 ♦ Egy közút leíró adata egy olyan string, amely (az alábbi sorrendben) három részből áll:

- útazonosító (csak számjegyeket és betűket tartalmazhat, min. 1, max. 6 jelből áll);
- útkategória (a 100..999 intervallumba eső szám);
- egyéb adatok (szöveg, max. 60 jel).

Az adatrészek elválasztására egy-egy szóközt használunk. Maguk az adatrészek nem tartalmaznak szóközt. A közutak száma maximum 5000. Az adatok egy szövegfájlban vannak, soronként egy úttal. Nevezzük ezt alapfájlnak. Feltehetjük, hogy a fájl tartalma ellenőrzött, az adatok formailag helyesek és a fájl rendezett (a sorok mint stringek növekvő sorrendben vannak).

Adott még egy másik szövegfájl, nevezzük ezt szelekciós fájlnak, ebben csak útazonosítók és útkategóriák vannak, soronként egy darab, de a felismerhetőség kedvéért az azonosítót tartalmazó sorok egy * jellel kezdődnek.

Oldjuk meg az alábbi feladatot (esetleg több szubrutinnal): állítsunk elő egy új alapfájlt az eredeti és a szelekciós fájl alapján! Az új fájlba azok az utak kerüljenek, amelyek azonosítója, vagy kategóriája szerepel a szelekciós fájlban.

6 ♦ Nagy táblázaton egy olyan mátrixot értünk, amelyre igazak a következő állítások:

- elemei egész típusúak;
- a sorok és oszlopok maximális száma `1000`.

A táblázatot megőrzési célból fájlban is tárolni kell. Ehhez a fájlformátum célszerűen megválasztandó. A táblázat egy blokkján a táblázat egy téglalap alakú részét értjük, amelyet a kezdő és vég sorindexszel valamint a kezdő és vég oszlopindexszel adunk meg.

Oldjuk meg az alábbi feladatokat (feladatonként külön szubrutin):

- Betöltés a fájlból.
- Kimentés a fájlba.
- Egy oszlop törlése.

- Egy sor törlése.
- Egy oszlopösszeg meghatározása.
- Egy sorösszeg meghatározása.
- Egy blokk-összeg meghatározása.
- Ellenőrzés: a mátrix egységmátrix-e.
- A maximális összegű sor kiválasztása.
- Két oszlop felcserélése.
- Két sor felcserélése.
- Feltöltés sorsolt értékekkel.

7 ♦ Nagy mátrixon egy olyan mátrixot értünk, amelyre igazak a következő állítások:

- elemei valós típusúak;
- a sorok és oszlopok maximális száma `10000`.

A táblázat egy négyzetes blokkján a táblázat egy téglalap alakú részét értjük, amelyet a kezdő és vég sorindexszel, valamint a kezdő és vég oszlopindexszel adunk meg.

Oldjuk meg az alábbi feladatokat (feladatonként külön szubrutin):

- Állítsuk elő (ha lehetséges) két ilyen mátrix algebrai szorzatát!
- Négyzetes blokk szimmetrizálása: a blokkban a blokk főátlójára szimmetrikusan elhelyezkedő elemeket számtani átlagukkal helyettesítjük.
- Négyzetes blokk szimmetricitás ellenőrzése: a blokkban a blokk főátlójára szimmetrikusan elhelyezkedő elemek értéke azonos-e?

8 ♦ Egy szövegfájl maximum 255 jel hosszú sorokat tartalmaz. A sorok maximális száma 10000.

Oldjuk meg az alábbi feladatokat (feladatonként külön szubrutin):

- Betöltés kollekcióba.
- Kimentés a fájlba.
- Egy sor beszúrása adott sorszámú sor után.
- Adott sorszámú sor törlése.
- Adott szövegrész keresése a teljes szövegben (eredmény: első előfordulás sor és azon belüli pozíciószáma).
- Adott sorszámú sor adott pozíciónál való megtörése (két sor lesz).
- Törlés adott sorszámú sor adott pozíciójától adott sorszámú sor adott pozíciójáig.

9 ♦ Nagy táblázaton egy olyan adattáblázatot értünk amelyre igazak a következő állítások:

- elemei min. 1, max. 5 hosszúságú stringek, amelyek csak nagybetűket (`'A'..'Z'`) és számjegyeket tartalmazhatnak;

- a sorok és oszlopok száma azonos (négyzetes táblázat), ezt az értéket a táblázat méretének nevezzük és ez az érték maximum 200 lehet;
- a táblázat egy blokkján a táblázat egy téglalap alakú részét értjük, amelyet a kezdő és vég sorindexszel, valamint a kezdő és vég oszlopindexszel adunk meg.

Egy blokk formázásán az alábbi átalakítást értjük: oszloponként az oszlop minden eleme feltöltendő balról, a maximális hosszú oszlopelem hosszára.

- Ha egy elem csak számjegyeket tartalmaz, akkor a feltöltés a „0" jellel történjen (előnullázás).
- Ha egy elem nem csak számjegyeket tartalmaz, akkor a feltöltés a „*" jellel történjen

Oldjuk meg az alábbi feladatokat (feladatonként külön szubrutin):

- Létrehozás és feltöltés sorsolt értékekkel, a hossz is elemenként sorsolt.
- Kijelölt blokk formázása.

10 ♦ A Turbo Pascal programozási nyelvben egy string maximum 255 jelet tartalmazhat. Lépjük túl ezt a korlátot úgy, hogy stringeket kollekcióban tárolunk, tételenként 255 jelet, kivéve az utolsó tételt, amely kevesebbet is tartalmazhat. A „nagy string" a tételek növekvő index szerinti összeolvasásával kapott jelsor lesz. Készítsünk el a normál stringekre standard módon adott stringkezelő eljárások és függvények megfelelőit (feladatonként külön szubrutin):

- Length.
- Pos.
- Delete.
- Insert.
- Copy.
- Két string összehasonlítása.

11 ♦ Időponton értünk egy év, hó, nap és óra (0..23) adatot, bejegyzésen értünk egy stringet. Egy időponthoz egy bejegyzés tartozhat. Egy fájlban egy rekordban tárolunk egy időpontot és a hozzátartozó bejegyzést. A fájl időpont szerint növekvően rendezett. A teljes adatrendszert „napló"-nak nevezzük. A napló egyszerre maximum `1000` időpontot tartalmazhat. A napló kezeléséhez oldjuk meg az alábbi feladatokat (feladatonként külön szubrutin):

- A napló betöltése egy kollekcióba.
- A fájl újraírása a kollekcióból.
- Egy időpont és a hozzá tartozó bejegyzés keresése.
- Új bejegyzés felvitele (időponttal).
- Bejegyzés törlése (időpont marad).
- Időpont törlése.

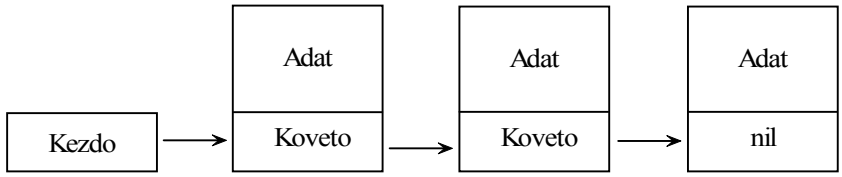
- Bejegyzés módosítása.
- Adott időintervallum törlése.
- Adott hosszú szabad (bejegyzésmentes) időszak keresése adott időintervallumban.
- Maximális hosszú szabad (bejegyzésmentes) időszak keresése adott időintervallumban.
- Adott szót tartalmazó bejegyzés keresése adott időintervallumban.

7. LÁNCOLT LISTÁK

7.1. Általános jellemzés

Az eddigiekben tárgyalt tömbszerű adatszerkezetek (statikus tömb, dinamikus tömb és kollekció) egydimenziós változatai egyben ún. *lineáris szerkezetek* is, ami azt jelenti, hogy van egyértelműen meghatározott *első* és *utolsó* elem, valamint (az első elemet kivéve) minden elemhez van egyértelműen meghatározott *követő* elem, valamint (az utolsó elemet kivéve) minden elemhez van egyértelműen meghatározott *előző* elem.

A *láncolt listák* is ilyen, lineáris adatszerkezetek, méghozzá olyanok, amelyeknél a sorrendi kapcsolatot (előző, követő) a lista elemeiben elhelyezett *mutatók* hozzák létre. A legegyszerűbb láncolt lista az egy irányban láncolt (vagy röviden egyirányú) lista (28. ábra). Egyszerűsége ellenére magában hordja a láncolt listák meghatározó jellegzetességeit és alkalmas ezek bemutatására ill. megértésére.



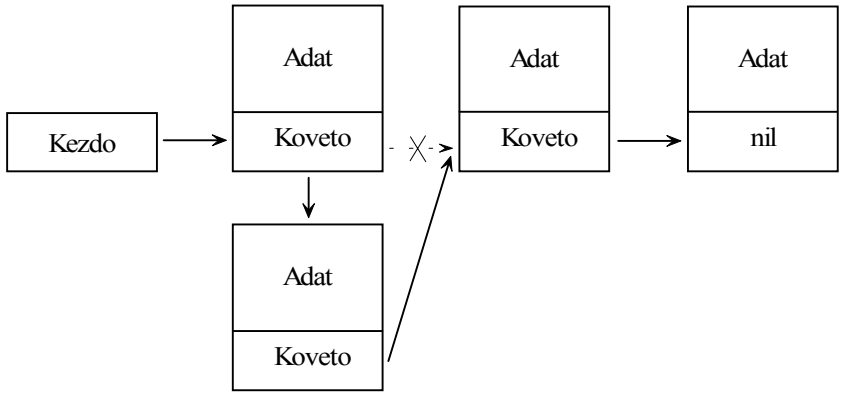
28. ábra. Egy irányban láncolt lista

Egyik általunk feltételezett programnyelv sem tartalmaz a láncolt listáknak megfelelő standard adattípust, sőt a Delphi fejlesztőrendszer eddigi verziói sem adnak ilyen objektum- vagy komponenstípust. Természetesen mindegyik nyelv biztosítja a megfelelő eszközöket a láncolt listák megvalósításához. A listaelemek rekordok lesznek (mivel az ésszerűen elképzelhető esetekben legalább kétfajta adat van egy elemben, az egyik a mutató, a másik az adatsor egy eleme). A mutatók azonos típusú, típusos mutatók, tehát az elemek azonos rekordtípusú *dinamikus* változók. (Ez az alapeset, ami objektumorientált fejlesztésnél – a szélesebb körű típuskompatibilitás eredményeképpen – túlléphető, egy listában eltérő típusú objektumok is tárolhatók. Itt csak az alapesettel foglalkozunk.) A lista végének jelzésére záróértékként a `nil` mutatóértéket alkalmazzuk. Az egyirányú lista elemeinek elérhetőségéhez szükséges ismernünk az első elem mutatóját (az ábrán a `Kezdo` adat). A megfelelő deklarációs séma:

```
type
  TATip=adattípus;
  PL1RekTipus=^TL1RekTipus;
  TL1RekTipus=record
    Adat: TATip;
    Koveto: PL1RekTipus;
  end;
var
  Kezdo: PL1RekTipus;
```

Az üres (elemekkel aktuálisan nem rendelkező) listát a `Kezdo = nil` érték reprezentálja. A `nil` mutatóértéket (amely minden mutatótípussal kompatíbilis) használjuk általánosan az aktuálisan nemlétező elemek jelzésére.

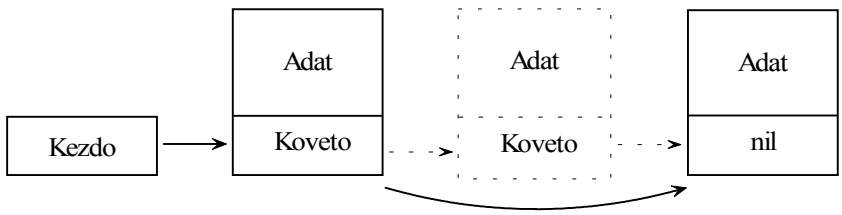
A tömbszerű lineáris adatszerkezetekben a fizikai tárolási sorrend adja az egymásra következést, ezzel lehetővé téve az indexezést és az ezzel járó előnyöket (lekérdezés közvetlen hivatkozással, bináris keresés lehetősége stb.). A listaelemben lévő mutató ilyen lehetőségeket nem ad, a *láncolt listáknál nincs indexe* az elemnek, a hozzáférés alapvetően *soros,* vagyis, ha egy elemet el akarunk érni, ahhoz végig kell járnunk a megelőző elemeket is. (Ebből következően, tehát pl. egy rendezett listánál sem tudjuk a leghatékonyabb módszert, a bináris keresést alkalmazni.)



29. ábra. Beszúrás egy irányban láncolt listába

A láncolt listák előnyös tulajdonságai leginkább a *módosító–karbantartó* jellegű feladatoknál mutatkoznak meg. Míg a tömböknél egy elem beszúrása vagy a törlése szükségszerűen az elem helyétől függő mennyiségű adatmozgatással jár (gondoljuk meg pl. hogy az első tömbelem törléséhez az összes többit „át kell pakolni"), addig a láncolt listáknál a beszúrás és a törlés az *elem helyétől függetlenül,*

bármely elemre nézve is csak *néhány műveletet* igényel, hiszen egy-két mutató átírásával megvalósítható (29. és 30. ábra). A láncolt lista a helykihasználás szempontjából is jobb mint a tömb vagy a kollekció, hiszen a *listában soha sincs nem használt mutató* (a statikus tömböknél ill. a statikus mutatótömbű kollekcióknál a legtöbbször van ilyen, ha a tömb dinamikus, akkor viszont áthelyezések válhatnak szükségessé az optimális helykihasználáshoz).



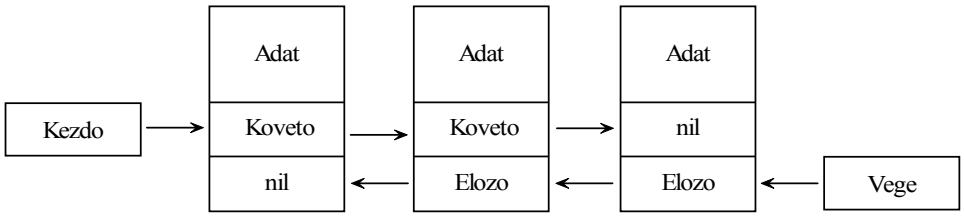
30. ábra. Elem törlése egy irányban láncolt listából

Levonhatjuk a következtetést, hogy a láncolt listákat akkor előnyös alkalmazni, ha a modellben nincs szükség az elemek közvetlen elérésére, vagy a módosító–karbantartó jellegű részek dominálnak a lekérdező jellegűek felett. Tipikusan ilyenek például a *sorban állást – kiszolgálást,* a várakozó sorokat leképező számítástechnikai modellek.

A lineáris jelleg megtartása mellett a láncolt lista jellegű adatszerkezet több alapváltozata is széleskörűen alkalmazott a számítástechnikai modellekben. Ezek közül mutatunk néhányat az alábbiakban.

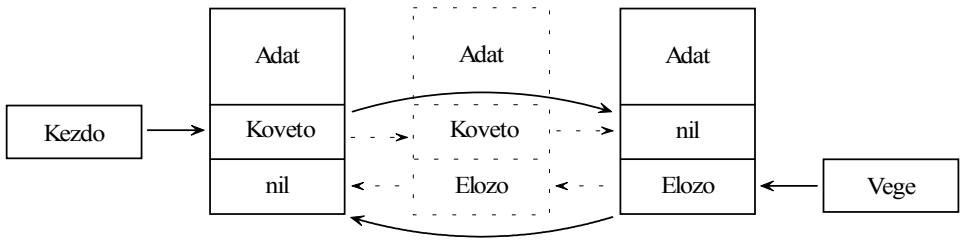
Az egyirányú listában nem lehet *visszafelé* lépkedni. Ha a feladatban ez is szükséges, vagy hasznos, akkor még egy mutatómezőt veszünk be a rekordba, ami mindig az előzőre mutat (a legelsőnél értéke `nil`), így kapjuk a *két irányban láncolt listát* (31. ábra), amelynek általános deklarációs sémája:

```
type
  TATip=adattípus;
  PL2RekTipus=^TL2RekTipus;
  TL2RekTipus=record
    Adat: TATip;
    Koveto: PL2RekTipus;
    Elozo: PL2RekTipus;
  end;
var
  Kezdo, Vege: PL2RekTipus;
```



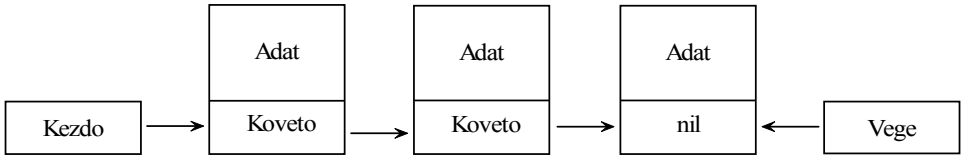
31. ábra. Két irányban láncolt lista

A `Vege` az utolsó elemre mutat, hogy az olvasás bármelyik végről indulhasson. Természetesen az elemenkénti két mutató megnöveli a karbantartási adminisztrációt is, például a törlésnél lásd 32. ábra.



32. ábra. Elem törlése két irányban láncolt listából

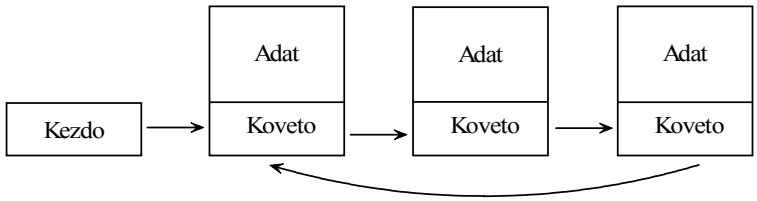
Bizonyos modellekben (pl. egy olyan várakozó sorban, amibe csak a végén lehet beállni) az egyirányú listánál is hasznos az utolsó elem külön nyilvántartása (33. ábra).



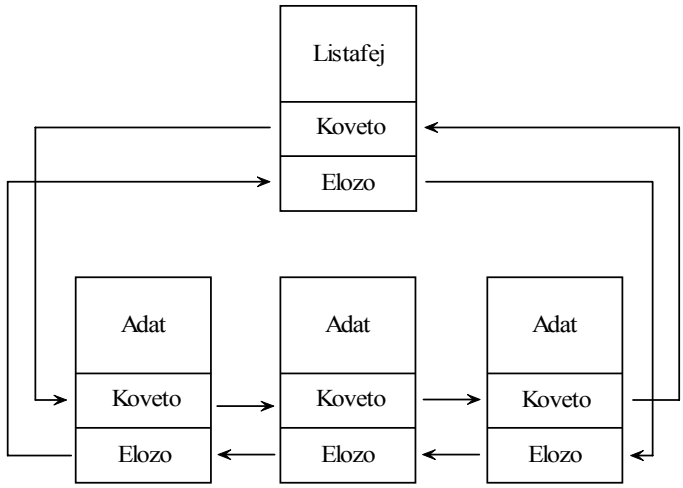
33. ábra. Utolsó elem nyilvántartása egyirányú listánál

Ciklikus listának nevezzük azt a szerkezetet, amelynél a záróérték helyett a kezdőpontra mutató érték van. Egyirányú ciklikus listát láthatunk a 34. ábrán.

Ha a lista első elemét valamilyen okból megkülönböztetjük a többitől (például azért, hogy a teljes listára jellemző adatokat tároljunk benne), akkor *fejelt* listáról és a lista fejéről vagy fejrekordjáról beszélünk. Például egy kétirányú lista fejrekordjába (tetszőleges adatrész mellett) betehetjük a `Kezdo` és a `Vege` mutatókat, ezt az esetet szemlélteti a 35. ábra.



34. ábra. Egyirányú ciklikus lista



35. ábra. Kétirányú fejelt lista

7.2. Alapfeladatok

A láncolt listák is dinamikus változókból felépülő *dinamikus adatszerkezetek,* tehát ennek megfelelően célszerű a módosítási műveleteket tagolni.

A *bővítés* (beszúrás) elvi sémája a láncolt listákra specializálva:

- Az új listaelem (rekord) létrehozása:
  - A szükséges hely meglétének (a bővítés lehetőségének) *ellenőrzése.*
  - A listaelem *létrehozása,* ha van hely.
- Ha van új elem:
  - A listaelem *feltöltése.* Ez általában az adatrész végleges állapotának és a mutató(k) valamilyen célszerű kezdőállapotának előállítását jelenti.
  - Az új elem *beillesztése* a struktúrába. Ennek során egyrészt az új elem mutatórészét, másrészt a láncbani környezetének (előző és követő ele-

mek) mutatóit kell helyesen beállítani. A lista fajtájától, aktuális állapotától és az új elem helyétől függően szükség lehet még a lista kezdő és végmutatóinak, valamint a fejrekordnak a módosítására is. Szemléletes a beillesztést a szerint tagolni, hogy az új elem hova kerül: a lista *elejére, végére,* vagy két, már meglévő elem *közé.*

A *szűkítés* (törlés) elvi sémája a láncolt listákra specializálva (feltételezzük, hogy a törlendő elem létezik):

- A törlendő elem *kikapcsolása* a struktúrából. Ennek során a törlendő elem láncbani környezetének (előző és követő elemek) mutatóit kell helyesen beállítani. A lista fajtájától, aktuális állapotától és az elem helyétől függően szükség lehet még a lista kezdő- és végmutatóinak, valamint a fejrekordnak a módosítására is. Szemléletes a kikapcsolást a szerint tagolni, hogy az elem honnan törlődik: a lista *elejéről, végéről,* vagy két már meglévő elem *közül.*
- A törlendő listaelem (rekord) megszüntetése.

A módosító algoritmusok készítésénél:

- *Minden* esetet vegyünk figyelembe és minden érintett *mutatót* állítsunk be.
- Bonyolultabb esetekben a gépi teszt előtt célszerű egy *rajzos* ellenőrzést végezni, tehát rajzoljuk fel sematikusan a struktúrát, és a rajzon hajtsuk végre a mutatók változásait.

A mintafeladatok adatstruktúráihoz az alábbi típusdeklarációkat használjuk (`uListaD` unit):

```
type
  TAdat=String;
 {egyirányban láncolt lista}
  PLElem1=^TLElem1;
  TLElem1=record
    Adat: TAdat;
    Koveto: PLElem1;
  end;
  {két irányban láncolt lista}
  PLElem2=^TLElem2;
  TLElem2=record
    Adat: TAdat;
    Elozo, Koveto: PLElem2;
  end;
```

A programnyelvi megvalósításban a bővítési műveletnél, a fejlesztőrendszerek eltéréséből adódó problémát ugyanúgy oldjuk meg, mint a kollekcióknál. A megfelelő szubrutinok: `tListaU.UjLElem1`, `tListaU.UjLElem2`, `dListaU.UjLElem1` és `dListaU.UjLElem2`.

A listakezelő algoritmusoknak magát a listát minimálisan a kezdőmutatóval adjuk át. Ha van végmutató és ez az algoritmusban szükséges vagy hasznos, akkor természetesen ezt is át kell adni. Ugyanez vonatkozik a fejrekordra is.

Példaképpen nézzünk egy egyszerűbb és egy bonyolultabb listaszerkezetet, és a hozzá tartozó néhány alapfeladatot:

Az első listaszerkezetünk a fenti elemdeklarációnak megfelelő egyirányban láncolt lista. A kezdő- és végmutatót külön változókban tároljuk.

7.2.1. mintafeladat: Inicializáljuk a listát!

Útmutató ♦ A kezdő és végmutató is vegye fel a `nil` értéket.

Szubrutin: `uLista.LInit1`.

7.2.2. mintafeladat: Keressünk meg egy adott értéket egy egyirányban láncolt listában!

Útmutató ♦ A mutató szerint haladva elemenként vizsgáljuk a listát. A keresés eredménye a keresett érték első előfordulásának mutatója, ha ilyen van. Ha a listán nem fordul elő a keresett érték, az eredmény `nil`.

Adatszerkezet (51)

Azonosító	Funkció	Típus	Jelleg
`Kezdo`	a lista kezdete	`PLElem1`	input
`Mit`	a keresett érték	`TAdat`	input
`LKeres1`	a `Mit` mutatója vagy `nil`	`PLElem1`	output
`Akt`	aktuális mutató a listán lépkedésnél	`PLElem1`	munka

Szubrutin `uLista.LKeres1`.

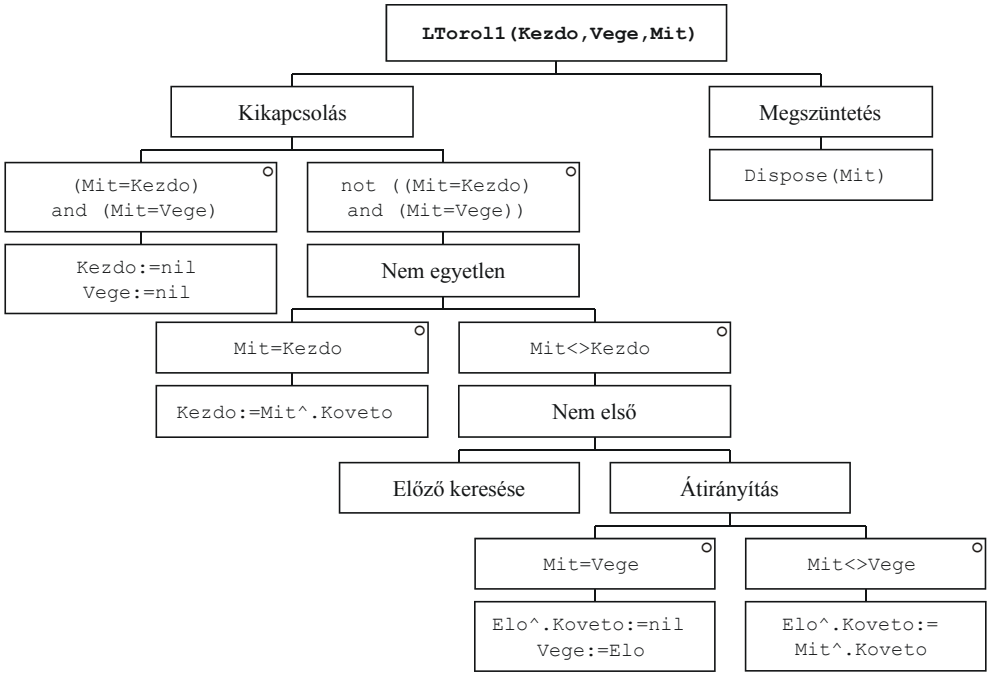
7.2.3. mintafeladat: Töröljünk egy adott mutatójú elemet a listából!

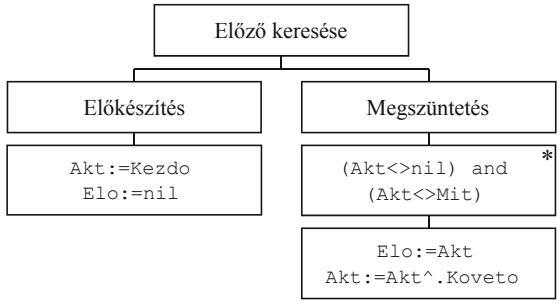
Útmutató ♦ Az általános elvi séma szerint haladunk, ha a törlendő elem nem az első, meg kell keresni a láncban a törlendőt megelőző elemet is. Feltételezzük, hogy a törlendő mutató létezik a listában.

Adatszerkezet (52)

Azonosító	Funkció	Típus	Jelleg
`Kezdo`	a lista kezdete	`PLElem1`	input, output
`Vege`	a lista vége	`PLElem1`	input, output
`Mit`	a törlendő elem mutatója	`PLElem1`	input
`Akt`	aktuális mutató a listán lépkedésnél	`PLElem1`	munka
`Elo`	az aktuálist megelőző mutató a listán lépkedésnél	`PLElem1`	munka

Struktúradiagram (52)





Szubrutin: `uLista.LTorol1`.

7.2.4. mintafeladat: A listához adjunk egy új adatot, ez a lista végére kerüljön!

Útmutató ♦ A feladat nagyon egyszerű, a beillesztés csak a vég- és kezdőmutatókat érinti. Ha a lista nem bővíthető, az eredmény `nil`.

Adatszerkezet (53)

Azonosító	Funkció	Típus	Jelleg
`Kezdo`	a lista kezdete	`PLElem1`	input, output
`Vege`	a lista vége	`PLElem1`	input, output
`Mit`	az új adat	`TAdat`	input
`LujVegere1`	az új elem címe vagy `nil`	`PLElem1`	output
`Uj`	az új elem címe vagy `nil`	`PLElem1`	munka

Szubrutin: `uLista.LUjVegere1`.

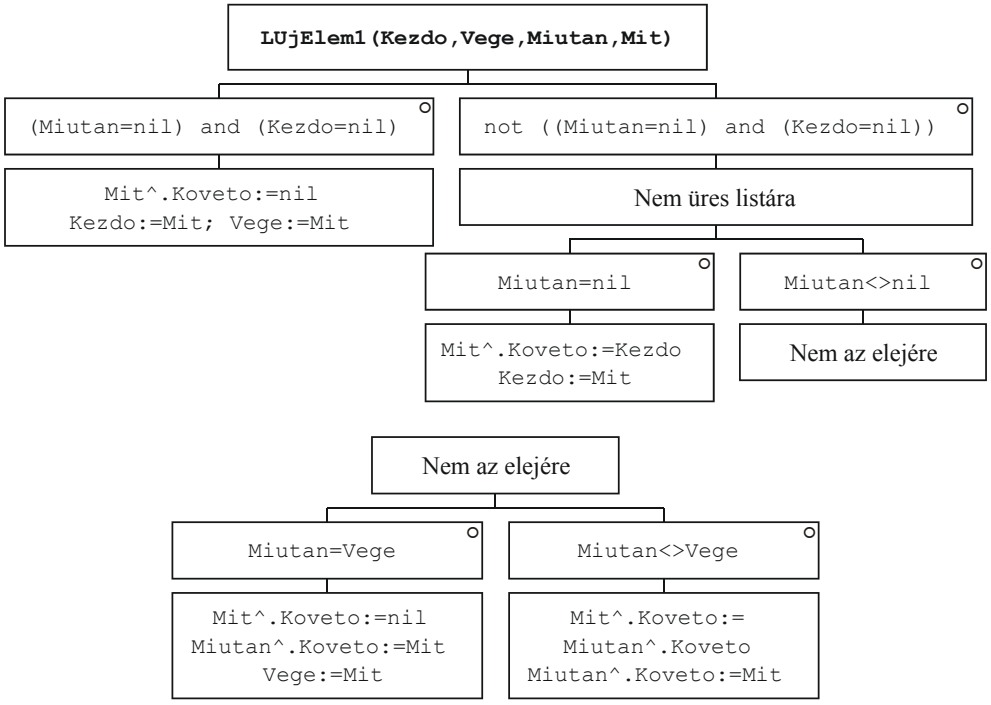
7.2.5. mintafeladat: Szúrjunk be egy új adatot a listára, adott mutatójú elem utánra!

Útmutató ♦ A megoldásban követjük az általános elvi sémát, de itt ez a beillesztéssel kezdődik, mert az új elemet készen kapjuk. Ha az adott mutató `nil`, ezt a lista elejére való beszúrásként értelmezzük. Feltételezzük, hogy az adott megelőző mutató létezik a listában.

Adatszerkezet (54)

Azonosító	Funkció	Típus	Jelleg
`Fej`	a lista feje	`PLElem2`	input, output
`Miutan`	az új elemet megelőző mutató a listán vagy `nil`	`PLElem2`	input
`Mit`	az új elem címe	`PLElem2`	input

Struktúradiagram (54)



Szubrutin: `uLista.LUjElem1`.

7.2.6. mintafeladat: Számoljuk fel a listát!

Útmutató ♦ Addig töröljük az első elemet, amíg van elem a listán.

Szubrutin: `uLista.LFelszFej2`.

A *második listaszerkezetünk* a fenti elemdeklarációnak megfelelő, két irányban láncolt lista. A lista első elemét fejrekordként, tehát magát a listát *fejelt* listaként értelmezzük, úgy, hogy az `Adat` a lista aktuális elemszámát (természetesen `String` típusban), az `Elozo` a lista első (valódi) elemének címét, a `Koveto` pedig a lista utolsó elemének címét tartalmazza. A lényeg kiemelése céljából a fejben lévő számlálóhoz külön szubrutinban adjuk meg a technikai segédeszközöket: `uLista.StrPluszEgy`, `uLista.StrMinuszEgy`.

7.2.7. mintafeladat: Inicializáljuk a listát!

Útmutató ♦ A fejrekord mutatói vegyék fel a `nil` értéket, az adatrésze a 0 értéket.

Szubrutin: `uLista.LInitFej2`.

7.2.8. mintafeladat: Szúrjunk be egy új adatot a listára, adott mutatójú elem után!

Útmutató ♦ A megoldásban követjük az általános elvi sémát, de itt ez a beillesztéssel kezdődik, mert az új elemet készen kapjuk. Ha az adott mutató `nil`, ezt a lista elejére való beszúrásként értelmezzük. Feltételezzük, hogy az adott megelőző mutató létezik a listában.

Adatszerkezet (55)

Azonosító	Funkció	Típus	Jelleg
`Fej`	a lista feje	`PLElem2`	input, output
`Miutan`	az új elemet megelőző mutató a listán vagy `nil`	`PLElem2`	input
`Mit`	az új elem címe	`PLElem2`	input

Szubrutin: `uLista.LUjElemFej2`.

Megjegyzés ♦ A fejrekord értelmezéséből következően `Fej^.Elozo` = a lista kezdete, `Fej^.Koveto` = a lista vége.

7.2.9. mintafeladat: Töröljünk egy adott mutatójú elemet a listából!

Útmutató ♦ Az általános elvi séma szerint haladunk, az elemekben minden információ megvan. Feltételezzük, hogy a törlendő mutató létezik a listában.

Adatszerkezet (56)

Azonosító	Funkció	Típus	Jelleg
`Fej`	a lista feje	`PLElem2`	input, output
`Mit`	a törlendő elem címe	`PLElem2`	input

Szubrutin: `uLista.LTorolFej2`.

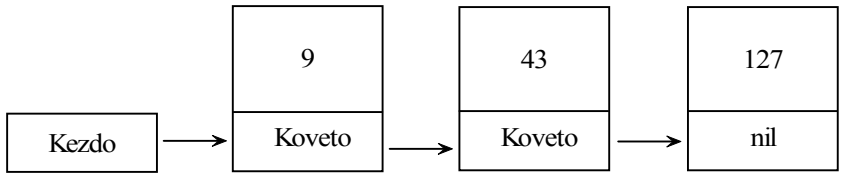
7.2.10. mintafeladat: Számoljuk fel a listát, a fejrekord megtartása mellett!

Útmutató ♦ Addig töröljük az első elemet, amíg van elem a listán.

Szubrutin: `uLista.LFelszFej2`

7.3. Rendezett láncolt listák

Rendezett listáról akkor beszélhetünk, ha a listaelemek adatrészeire értelmezett az összehasonlítás (pl. stringek, számok), vagy az adatrészekhez valamilyen eljárással rendelünk ilyen értékeket. Ezeket az értékeket nevezzük az elem értékének. A lista akkor rendezett, ha a kezdőelemmel indulva és a követő mutatók szerinti sorrendben haladva az elemértékek nagysága nem csökken (ezt nevezzük *növekvő* rendezettségnek, 36. ábra), vagy nem nő (ezt nevezzük *csökkenő* rendezettségnek).



36. ábra. Rendezett láncolt lista

Az elemek soros elérhetőségének megfelelően a rendezett listákra jellemzően a *lineáris (soros) keresési* és a *beszúrásos rendezési* algoritmusokat alkalmazzuk.

Az adatstruktúra nagyon lényeges jellemzője még, hogy megenged-e ismétlődést, vagyis több, azonos értékű elemet. Ezt mindig a konkrét alkalmazás ismeretében kell eldönteni a tervezőnek.

Mintapéldáink az előző pontban már megismert két listaszerkezet *növekvően* rendezett változatára vonatkoznak az *ismétlődések* megengedésével. Az elem értékeként a tárolt string értékét vesszük. A megoldások igyekeznek minél job-

ban felhasználni a rendezettséget nem feltételező alapalgoritmusokat. Ez sematikusan azt jelenti, hogy az érték alapján meghatározunk mutatót (környezetet) és ezzel visszavezetjük a problémát az alapszintre.

A következő három mintafeladat az egyirányban láncolt listára vonatkozik. A végmutatót is adminisztráljuk.

7.3.1. mintafeladat: Keressünk meg egy adott értéket a listában!

Útmutató ♦ A rendezetlen esettől csak annyi az eltérés, hogy a keresés eredménye már az első, a keresett értéknél nem kisebb értékű elemet elérve kiderül. Jól hasznosítható mellékeredményként előállítjuk még az utolsó, a keresettnél még kisebb értékű elem mutatóját is. Ha ilyen nincs, ez az eredmény is `nil` lesz. Ha ilyen van, ez a keresett értékű (aktuálisan létező, vagy nemlétező) elemet a listában megelőző elem mutatója lesz.

Adatszerkezet (57)

Azonosító	Funkció	Típus	Jelleg
`Kezdo`	a lista kezdete	`PLElem1`	input
`Mit`	a keresett érték	`TAdat`	input
`Elozo`	a keresett értéket megelőző mutató vagy `nil`	`PLElem1`	output
`RLKeres1`	a `Mit` mutatója vagy `nil`	`PLElem1`	output
`Akt`	aktuális mutató a listán lépkedésnél	`PLElem1`	munka

Szubrutin: `uListaR.RLKeres1`.

Megjegyzés ♦ Ismétlődés esetén az első elem az eredmény.

7.3.2. mintafeladat: Szúrjunk be egy új adatot a listába!

Útmutató ♦ Az új elem létrehozása és feltöltése után a beszúrás helyét az előző szubrutinnal keressük meg, magához a beszúráshoz az `uLista.LUjElem1` szubrutint hívjuk meg. A művelet eredménye az új elem mutatója vagy `nil` lesz. Az utóbbi akkor, ha helyhiány miatt nem hozható létre az új elem.

Adatszerkezet (58)

Azonosító	Funkció	Típus	Jelleg
`Kezdo`	a lista kezdete	`PLElem1`	input, output
`Vege`	a lista vége	`PLElem1`	input, output
`UjAdat`	az új adat	`TAdat`	input
`RListara1`	az új elem mutatója vagy `nil`	`PLElem1`	output
`Elo`	a keresett értéket megelőző mutató vagy `nil`	`PLElem1`	munka
`Uj`	az új elem mutatója vagy `nil`	`PLElem1`	munka

Szubrutin: `uLista.RListara1`.

7.3.3. mintafeladat: Töröljünk egy adatot a listáról!

Útmutató ♦ A törlendő elemet és környezetét a fenti kereső szubrutinnal határozzuk meg, magához a törléshez az `uLista.LTorol1` szubrutint hívjuk meg. A művelet eredményével jelezzük azt, hogy volt-e ilyen adat a listán.

Adatszerkezet (59)

Azonosító	Funkció	Típus	Jelleg
`Kezdo`	a lista kezdete	`PLElem1`	input, output
`Vege`	a lista vége	`PLElem1`	input, output
`TorlAdat`	a törlendő adat	`TAdat`	input
`RListarol1`	volt-e törlés	`Boolean`	output
`Elozo`	a keresett értéket megelőző mutató vagy `nil`	`PLElem1`	munka
`Torl`	a törlendő elem mutatója vagy `nil`	`PLElem1`	munka

Szubrutin: `uLista.RListara1`.

A következő három mintafeladat a két irányban láncolt fejelt listára vonatkozik.

7.3.4. mintafeladat: Keressünk meg egy adott értéket a listában!

Útmutató ♦ Az egyirányú esettől csak annyi az eltérés, hogy az előző mutatót külön nem kell kezelnünk, ezt az eredmény tartalmazza.

Adatszerkezet (60)

Azonosító	Funkció	Típus	Jelleg
`Fej`	a lista feje	`PLElem2`	input
`Mit`	a keresett érték	`TAdat`	input
`RLKeresFej2`	a `Mit` mutatója vagy `nil`	`PLElem2`	output
`Akt`	aktuális mutató a listán lépkedésnél	`PLElem2`	munka

Szubrutin: `uListaR.RLKeresFej2`.

7.3.5. mintafeladat: Szúrjunk be egy új adatot a listába!

Útmutató ♦ Az új elem létrehozása és feltöltése után a beszúrás helyét a lista elején és végén közvetlenül, belső esetben az előző szubrutinnal keressük meg, magához a beszúráshoz az `uLista.LUjElemFej2` szubrutint hívjuk meg. A művelet eredménye az új elem mutatója vagy `nil` lesz. Az utóbbi akkor, ha helyhiány miatt nem hozható létre az új elem.

Adatszerkezet (61)

Azonosító	Funkció	Típus	Jelleg
`Fej`	a lista feje	`PLElem2`	input, output
`UjAdat`	az új adat	`TAdat`	input
`RListaraFej2`	az új elem mutatója vagy `nil`	`PLElem2`	output
`Elo`	a keresett értéket megelőző mutató vagy `nil`	`PLElem2`	munka
`Uj`	az új elem mutatója vagy `nil`	`PLElem2`	munka

Szubrutin: `uLista.RListaraFej2`.

7.3.6. mintafeladat: Töröljünk egy adatot a listáról!

Útmutató ♦ Az egyirányú eset értelemszerű változata.

Adatszerkezet (62)

Azonosító	Funkció	Típus	Jelleg
`Fej`	a lista feje	`PLElem2`	input, output
`TorlAdat`	a törlendő adat	`TAdat`	input
`RListarolFej2`	volt-e törlés	`boolean`	output
`Torl`	a törlendő elem mutatója v. `nil`	`PLElem2`	munka

Szubrutin: `uLista.RListarolFej2`.

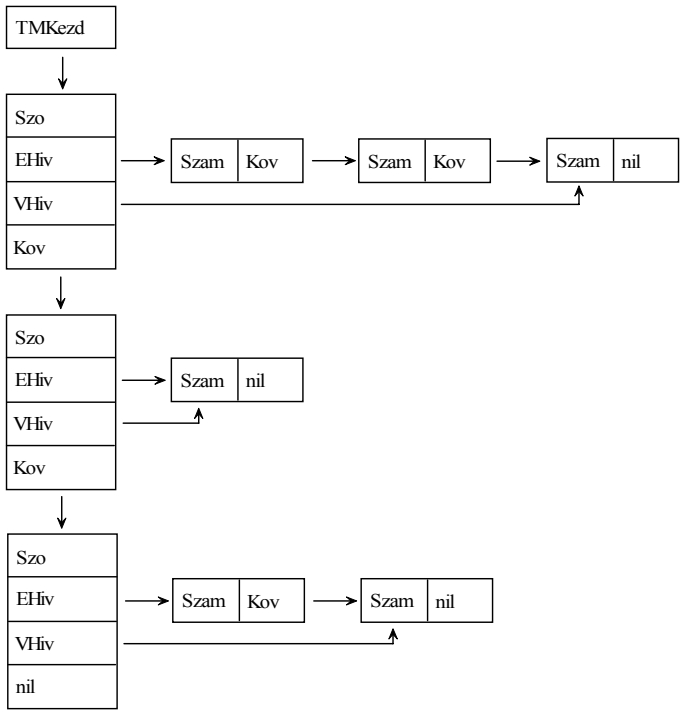
7.4. Összetett listák

Egy láncolt lista bármely eleme tartalmazhatja egy másik lista kezdőmutatóját, vagy fejét, így tetszőleges bonyolultságú összetételeket hozhatunk létre. Példaképpen egy *tárgymutató* adatstruktúráját nézzük meg.

Általánosan ismertek a szakkönyvekben lévő *tárgymutatók,* amelyekben a szöveg egyes kiemelt szavai vannak ábécé sorrendben, mindegyik mellett feltüntetve (növekvő sorrendben) azokat az oldalszámokat, ahol a szó előfordul. Ehhez (pl. egy tárgymutató belső előállításához és tárolásához) tervezzünk egy adatstruktúrát.

A szavakat egy egyirányú, növekvően rendezett (tehát ábécé sorrendű) listára fűzzük, és minden szóhoz hozzákapcsoljuk a hivatkozások – szintén növekvő sorrendű – egyirányú listáját (37. ábra). A könnyebb kezelhetőség céljából a hivatkozási lista eleje mellett a végére is rámutatunk a szórekordban. A tárgymutató általános deklarációs sémája (feltételezve, hogy a hivatkozások értéke nem haladja meg a `65535`-öt):

```
type
  TTargySzo=String;  {tárgyszó}
  THivSzam=Word;  {hivatkozás}
type
  PHivRek=^THivRek;
  {hivatkozási rekord}
  THivRek=record
    Szam: THivSzam;  {hivatkozás}
    Kov: PHivRek;
  end;
  PSzoRek=^TSzoRek;
  {szórekord}
  TSzoRek=record
    Szo: TTargySzo;  {tárgyszó}
    EHiv, VHiv: PHivRek;  {a kapcsolódó hivatkozási lista
                           kezdő- és végmutatója}
    Kov: PSzoRek;
  end;
```



37. ábra. Tárgymutató felépítése

7.4.1. mintafeladat: Vegyünk fel egy új hivatkozást a tárgymutatóba!

Útmutató ♦ Az új hivatkozást egy tárgyszó és egy hivatkozási szám adja meg. A megoldásnál az alábbi feltételezésekkel dolgozunk:

- Az új dinamikus változókhoz (maximum egy szórekord és egy hivatkozási rekord) szükséges memória rendelkezésre állásának ellenőrzése a szubrutint hívó feladata.
- A hivatkozások növekvő sorrendben érkeznek, így az egyediséget és a növekvő sorrendet ezeknél nem kell ellenőrizni. (A feltételezés eléggé természetes, gondoljunk arra, pl. ha egy könyvet dolgozunk fel, akkor az oldalszámok növekvő sorrendje szerint haladunk.)

A jobb áttekinthetőség kedvéért azt az esetet, amikor a tárgymutató egy új eleme keletkezik (eddig még elő nem fordult szó az első hivatkozásával) egy külön szubrutinban emeljük ki (`uListaO.UjSzoRek`). Ha a kapott szó már bent van a tárgymutatóban, akkor csak új hivatkozásról van szó, ezt egyszerűen a már meglévő hivatkozási lista végére tesszük.

Adatszerkezet – UjSzorek (63)

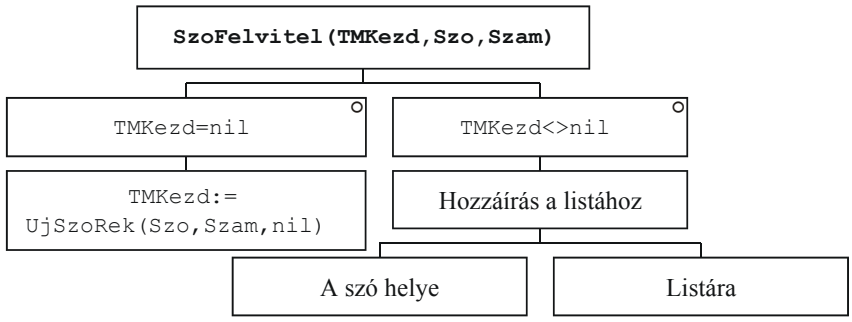
Azonosító	Funkció	Típus	Jelleg
`Szo`	a felveendő új szó	`TTargySzo`	input
`Szam`	a felveendő új szó hivatkozása	`THivSzam`	input
`KovSzo`	a szólistában az új szót követő cím	`PSzoRek`	input
`UjSzoRek`	az új szórekord címe	`PSzoRek`	output
`UjSzoRek`	az új szórekord címe	`PSzoRek`	munka
`UjHiv`	az új hivatkozási rekord címe	`PHivRek`	munka

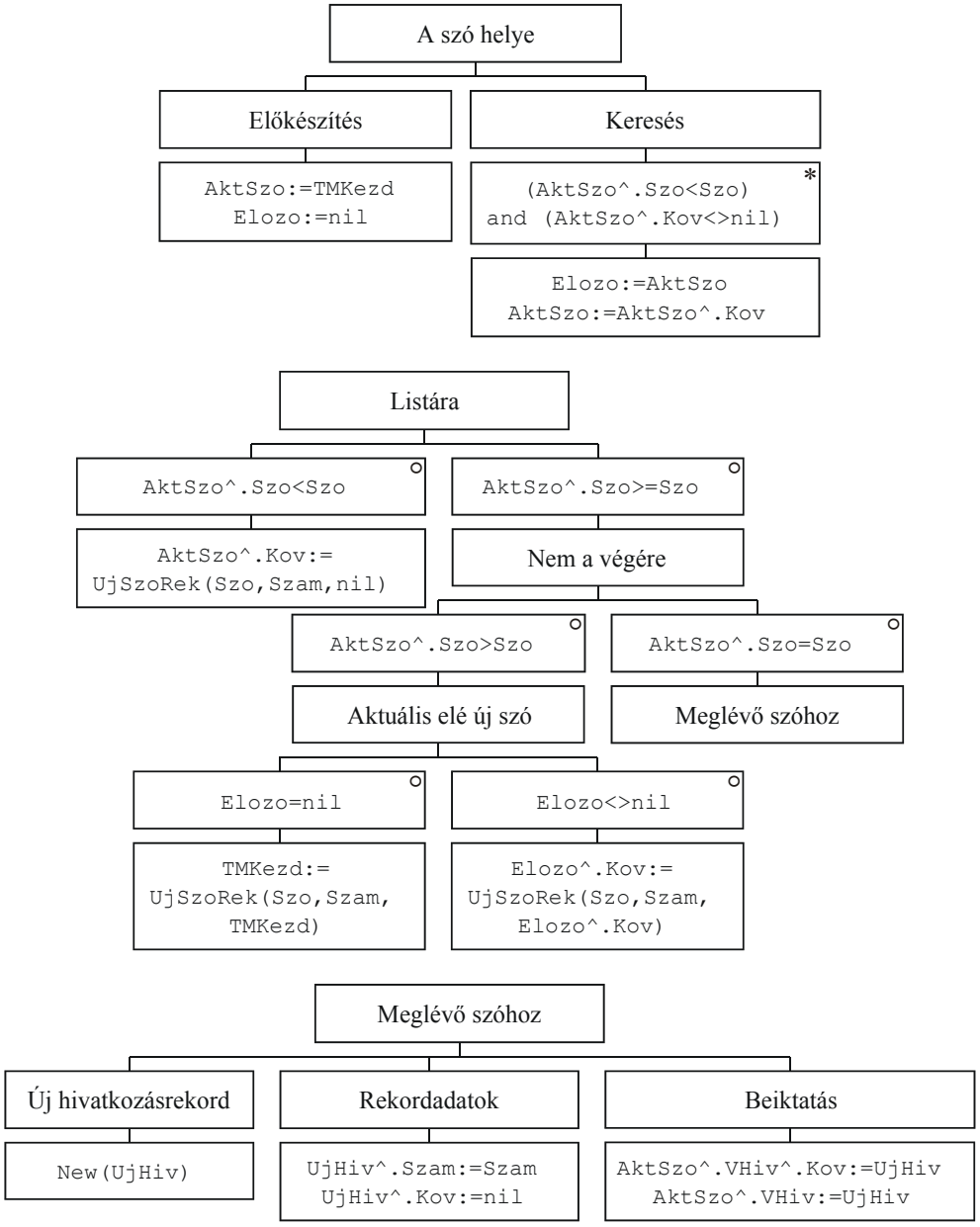
Szubrutin: `uListaO.UjSzoRek`.

Adatszerkezet – Szofelvitel (64)

Azonosító	Funkció	Típus	Jelleg
`TMKezd`	az összetett lista kezdőpontja	`TTargySzo`	input, output
`Szo`	a felveendő hivatkozás szava	`TTargySzo`	input
`Szam`	a felveendő hivatkozás száma	`THivSzam`	input
`AktSzo`	munkaváltozó a szólistában való keresésénél	`PSzoRek`	munka
`Elozo`	munkaváltozó a szólistában való keresésénél	`PSzoRek`	munka
`UjHiv`	az új hivatkozási rekord címe	`PHivRek`	munka

Struktúradiagram (64)





Szubrutin: `uListaO.SzoFelvitel`.

7.5. Listák és fájlok

A lista, még inkább, mint a kollekció, alkalmas arra is, hogy a háttértárolókon tárolt lineáris jellegű adatstruktúrák (Pascal fájlok) tartalmát az operatív memóriában tárolja. Ha a tárkapacitás megengedi, a fájlok (mint már a kollekciónál részleteztük) célszerűen feldolgozhatók a *teljes betöltés, belső feldolgozás, újraírás* munkamenettel.

7.5.1. mintafeladat: Egy szövegfájl maximum 255 jel hosszú sorokat tartalmaz. Rendezzük a fájl sorait növekvően!

Útmutató ♦ A fájlból létrehozunk egy egyirányú rendezett listát, majd belőle újraírjuk a fájlt. A beolvasást és a kiírást külön szubrutinban adjuk meg. A beolvashatóság feltétele az elegendő memória. A visszaírás a mi megoldásunkban fel is számolja a listát ez nem mindig szükséges. A feladat teljes megoldását a szubrutinok megfelelő keretben történő meghívása jelenti. Feltételezzük, hogy a keretprogram a fájl létezését is ellenőrzi.

Szubrutinok: `uListaR.SListaBe`, `uListaR.SListaKi`.

7.5.2. mintafeladat: Egyheti lottó adaton értünk 5 db. különböző, az 1..90 intervallumba eső egész számot. Egyheti adatot az év és az éven belül a hét sorszámával azonosítunk (pl.: 1993. év 2. hete). Az adatok egy típusos fájlban vannak, szigorúan növekvő időrendben, egy hét – egy rekord, de lehetnek hiányzó hetek. A legkorábbi adat az 1960-as év 1. hetére vonatkozik. A legutolsó, még kezelendő évszám 2099. Oldjuk meg a következő feladatokat:

- *Egyheti adat lekérdezése.*
- *Egyheti adat felvitele (még nem létező adatra).*
- *Egyheti adat módosítása (már létező adatra).*
- *A számok (1..90) relatív előfordulási gyakorisága, adott időszakra vonatkozóan.*

Útmutató ♦ A fájlból létrehozunk egy egyirányú rendezett listát, ezen elvégezzük a műveleteket, majd belőle újraírjuk a fájlt (bonyolultabb listát a kívánt funkciók nem indokolnak). A lista egy eleme a fájl egy rekordját tartalmazza adatrészként. Az időadat (év és hét) egyedi, tehát amellett, hogy rendezési szempont, azonosítóként is használható. Mivel a listaelemhez rendelt érték (a rendezési szempont) itt nem jelenik meg önálló adatként az elemben, hanem két adatból kell számolni, célszerű egy külön függvényt készíteni két időadat

összehasonlítására: `uListaF.HetViszony`. A beolvasást és a kiírást, valamint az egyes funkciókat külön szubrutinban adjuk meg. A feladat teljes megoldását a szubrutinok megfelelő keretben történő meghívása jelenti. Feltételezzük, hogy a keretprogram a fájl létezését is ellenőrzi. Az új elem létrehozhatóságát az eddigiekben már megismert módon ellenőrizzük: `dListaU.UjLotElem`, `tListaU.UjLotElem`. Az elmondottaknak megfelelő alapdeklarációk (`uListaLD`):

```
const
  LotSzDb=5;  MinEv=1960;  MaxEv=2099;  MaxHet=53;
  MaxSzam=90;
type
  TLotSzam=1..MaxSzam;
  TLotInd=1..LotSzDb;
  TEv=MinEv..MaxEv;
  THet=1..MaxHet;
  THetAdat=array[TLotInd] of TLotSzam;  {egy heti adat}
  THetAzon=record  {egy időadat, egy hét azonosítója}
    Ev: TEv;
    Het: THet;
  end;
  TLotRekord=record  {egy hét a fájlban}
    Azon: THetAzon;
    Adat: THetAdat;
  end;
  PLotElem=^TLotElem;
  TLotElem=record  {egy hét a listán}
    Adatok: TLotRekord;
    Koveto: PLotElem;
  end;

  THetViszony=(HetK, HetE, HetN);  {két időadat viszonya}
  TRelSzamGyak=array[TLotInd] of Real;  {gyakorisági tábla}
  TLotFajl=file of TLotRekord;
```

A megoldást szubrutinonként részletezzük.

7.5.2.1. mintafeladat: Hozzuk létre a listát fájlból!

Útmutató ♦ A fájl rendezett, ennek megfelelően az új elem mindig a lista végére kerül. A beolvashatóság feltétele az elegendő memória.

Szubrutin: `uListaF.LotListaBe`.

7.5.2.2. mintafeladat: Egyheti adat lekérdezése.

Útmutató ♦ A rendezett listán való keresés módszerét alkalmazzuk. A keresés eredménye a kérdezett hét adatait tartalmazó listaelem címe vagy `nil`. Jól hasznosítható mellékeredmény a megelőző elem mutatója.

Adatszerkezet (65)

Azonosító	Funkció	Típus	Jelleg
`Kezdo`	a lista kezdete	`PLotElem`	input
`Mit`	a keresett érték	`THetAzon`	input
`Elozo`	a keresett értéket megelőző mutató vagy `nil`	`PLotElem`	output
`LotKeres`	a `Mit` mutatója vagy `nil`	`PLotElem`	output
`Akt`	aktuális mutató a listán lépkedésnél	`PLotElem`	munka

Szubrutin: `uListaF.LotKeres`.

7.5.2.3. mintafeladat: Egyheti adat felvitele/módosítása.

Útmutató ♦ A két feladatot közös szubrutinnal oldjuk meg. Ha az adott hét már rajta van a listán, akkor a heti adatot lecseréljük (módosítás), ha nincs, akkor, ha van hely, új elemet képezünk, és beszúrjuk az időadatának megfelelő helyre.

Adatszerkezet (66)

Azonosító	Funkció	Típus	Jelleg
`Kezdo`	a lista kezdete	`PLotElem`	input, output
`Vege`	a lista vége	`PLotElem`	input, output
`UjAdat`	a hét és a heti adat	`TLotRekord`	input
`LotRa`	az `UjAdat` mutatója vagy `nil`	`PLotElem`	output
`Elo`	a keresett értéket megelőző mutató vagy `nil`	`PLotElem`	munka
`Akt`	aktuális mutató a listán lépkedésnél	`PLotElem`	munka

Szubrutin: `uListaF.LotRa`.

7.5.2.4. mintafeladat: A számok relatív előfordulási gyakorisága, adott időszakra vonatkozóan.

Útmutató ♦ A listán először is átlépjük a kezdő hetet megelőző heteket. Ezután az időszakba eső hetekre számonként meghatározzuk az előfordulások darabszámát, és számoljuk a hetek darabszámát. Végül számonkénti osztással megkapjuk az eredményt.

Adatszerkezet (67)

Azonosító	Funkció	Típus	Jelleg
`Kezdo`	a lista kezdete	`PLotElem`	input
`KezdHet`	az időszak kezdete	`THetAzon`	input
`VegHet`	az időszak vége	`THetAzon`	input
`Gyak`	a relatív gyakoriságok	`TRelSzamGyak`	output
`HetDb`	az időszakba eső hetek száma	`Word`	munka
`Akt`	aktuális mutató a listán lépkedésnél	`PLotElem`	munka
`J`	ciklusváltozó	`TLotInd`	munka
`I`	ciklusváltozó	`TLotSzam`	munka
`Szam`	egy lottószám	`TLotSzam`	munka

Szubrutin: `uListaF.LotGyak`.

7.5.2.5. mintafeladat: Hozzuk létre a fájlt listából!

Útmutató ♦ A listán végighaladva kiírjuk a listaelemek adatrészét a fájlba. Mivel a lista rendezett, a fájl is az lesz.

Szubrutin: `uListaF.LotListaKi`.

7.6. Feladatok

1 ♦ Adott egy egyirányban láncolt lista, amelynek adatrésze egy string. A végmutatót is kezeljük. Oldjuk meg az alábbi feladatokat (feladatonként külön szubrutin):

- A lista inicializálása (`uLista.LInit1`).
- Adott érték keresése (`uLista.LKeres1`).
- Adott mutatójú elem törlése (`uLista.LTorol1`).
- Elem törlése a lista elejéről.
- Elem törlése a lista végéről.

- Új adat beszúrása a lista végére (`uLista.LUjVegere1`).
- Új adat beszúrása a lista elejére (`uLista.LUjElore1`).
- Új elem beszúrása adott mutatójú elem utánra (`uLista.LUjElem1`).
- A teljes lista felszámolása (`uLista.LFelsz1`).

2 ♦ Adott egy két irányban láncolt fejelt lista, amelynek adatrésze egy string. A fejrekord felépítése:

- Az adatrészben az aktuális elemszámot tároljuk string típusban.
- A fejrekord „előző” mutatója a lista kezdetére mutat.
- A fejrekord „követő” mutatója a lista végére mutat.

Oldjuk meg az alábbi feladatokat (feladatonként külön szubrutin):

- A lista inicializálása (`uLista.LInitFej2`).
- Adott érték keresése.
- Adott mutatójú elem törlése (`uLista.LTorolFej2`).
- Elem törlése a lista elejéről.
- Elem törlése a lista végéről.
- Új adat beszúrása a lista végére.
- Új adat beszúrása a lista elejére.
- Új elem beszúrása adott mutatójú elem utánra (`uLista.LUjElemFej2`).
- A teljes lista felszámolása, a fejrekord megmarad (`uLista.LFelszFej2`).

3 ♦ Adott egy egyirányban növekvően rendezett láncolt lista, amelynek adatrésze egy string. Az ismétlődéseket megengedjük. A végmutatót is kezeljük. Oldjuk meg az alábbi feladatokat (feladatonként külön szubrutin):

- Adott érték keresése (`uListaR.RLKeres1`).
- Új adat beszúrása (`uListaR.RListara1`).
- Adat törlése (`uListaR.RListarol1`).

4 ♦ Oldjuk meg a 1., 2., 3. feladatokat az ismétlődések tiltása mellett!

5 ♦ Adott egy `X` szó és egy egyirányban láncolt, növekvően rendezett lista, amelynek adatrésze egy string. A végmutatót is kezeljük. Oldjuk meg az alábbi feladatokat (feladatonként külön szubrutin):

- Állítsunk elő egy ugyanilyen listát, az eredeti azon elemeiből, amelyekben az `X` mint szó előfordul, az eredeti lista változatlanul hagyása mellett!
- Állítsunk elő egy ugyanilyen listát, az eredeti azon elemeiből, amelyekben az `X` mint szó előfordul, úgy, hogy ezeket az elemeket az eredeti listából elhagyjuk!
- A listában előforduló azonos adatokból csak egyet hagyjunk meg (duplázások törlése)!

6 ♦ Oldjuk meg a 5. feladatokat a mintapéldáknál specifikált kétirányú fejelt listákra!

7 Állítsunk elő két olyan növekvően rendezett, egyirányban láncolt listából, amelyek adatrésze egy string egy harmadik ilyen listát, az alábbi módokon:

- Válogassuk össze az elemeket a rendezettség megtartásával!
- Válogassuk össze az elemeket a rendezettség megtartásával és a duplázások törlésével!

A megoldásokban a végmutatókat is kezeljük.

8 ♦ Oldjuk meg a 7. feladatokat a mintapéldáknál specifikált kétirányú fejelt listákra!

9 ♦ Adott egy két irányban láncolt, növekvően rendezett fejelt lista, amelynek adatrésze egy string. A fejrekord felépítése:

- Az adatrészben az aktuális elemszámot tároljuk string típusban.
- A fejrekord „előző" mutatója a lista kezdetére mutat.
- A fejrekord „követő" mutatója a lista végére mutat.

Oldjuk meg az alábbi feladatokat (feladatonként külön szubrutin):

- Adott érték keresése (`uListaR.RLKeresFej2`).
- Új adat beszúrása (`uListaR.RListaraFej2`).
- Adat törlése (`uListaR.RListarolFej2`)

10 ♦ Oldjuk meg a 9. feladatokat az ismétlődések tiltása mellett!

11 ♦ Tetszőlegesen nagy számokkal dolgozó egész aritmetikát valósítunk meg tízes számrendszerben úgy, hogy a számok előjelét és jegyeit egy láncolt listán tároljuk, a lista elemenként 1 jegyet tartalmaz.

Oldjuk meg az alábbi feladatokat (feladatonként külön szubrutin):

- Adott számú jegyet tartalmazó szám előállítása sorsolással.
- Két szám közül a nagyobb kiválasztása.
- Szám osztása 100-zal, két eredmény legyen, az egész hányados és a maradék.
- Szám kimentése szövegfájlba.
- Szám beolvasása szövegfájlból.
- Két szám összehasonlítása.
- Szám szorzása 10-zel.
- Számhoz 1 hozzáadása.
- Két szám összeadása és kivonása.

12 ♦ Oldjuk meg a 11. feladatokat úgy hogy a lista elemenként 8 jegyet tartalmazzon (kivéve az utolsó elemet, amely kevesebbet is tartalmazhat)!

13 ♦ Egy születési adat egy olyan string, amely három részből áll:

- vezetéknév;
- keresztnév;
- évszám.

Az adatrészek elválasztására egy-egy szóközt használunk. A vezetéknév és keresztnév nem tartalmaz szóközt. A string teljes hossza nem haladhatja meg a 30 jelet. Az adatok egy szövegfájlban vannak tárolva, soronként egy születési adat. A fájl nem rendezett.

- Készítsünk egy növekvő évszám szerint rendezett szövegfájlt!
- Készítsünk egy a nevek ábécé szerinti sorrendjében rendezett szövegfájlt!

14 ♦ Levélcímen értünk egy olyan stringet, amely három részből áll:

- irányítószám, amely egy pontosan 4 jegyű (nem előnullázott) pozitív egész;
- helységnév;
- többi adat.

Az adatrészek elválasztására egy-egy szóközt használunk. A helységnév nem tartalmaz szóközt. A string teljes hossza nem haladja meg a 90 jelet. Az adatok egy szövegfájlban vannak tárolva, soronként egy születési adat. A fájl nem rendezett.

- Készítsünk egy növekvő irányítószám szerint rendezett szövegfájlt!
- Készítsünk egy a helységnevek ábécé szerinti sorrendjében rendezett szövegfájlt!

15 ♦ Egy adatsoron értjük olyan egész számok véges sorozatát, amelyeket (adott korlátok között) sorsolással. Egy adatsor előállításához tehát meg kell adni az elemszámot valamint a sor elemeinek alsó és felső korlátját. Az elemek számát statikusan nem korlátozzuk. Az adatsort a memóriában láncolt listában tároljuk, úgy hogy egy listaelem 100 darab sorelemet tartalmazzon (kivéve az utolsó elemet, amely kevesebbet is tartalmazhat). Gyakorisági táblán értjük az adatsorban előforduló különböző értékek darabszámát értékenként. Oldjuk meg az alábbi feladatokat (feladatonként külön szubrutin):

- Sor előállítása.
- Minimum, maximum képzése.
- Átlag és szórás számítása.
- Gyakorisági tábla számítása.

16 ♦ Ismert a következő módszer prímszámok meghatározására (Eratosztenész rostája):

a) írjuk fel a természetes számokat `1`-től `N`-ig
b) húzzuk ki az 1-et
c) jelöljük meg a következő még nem jelölt és ki nem húzott számot és húzzuk ki az összes többszöröseit
d) ismételjük a *c)* lépést amíg van megjelölhető szám

A megjelölt számok az `[1, N]` intervallum prímszámai.

A számítások lebonyolítására láncolt listákat használva, és listaelemenként 1 számot tárolva oldjuk meg az alábbi feladatokat (feladatonként külön szubrutin):

- Adott `N`-re a prímszámok meghatározása.
- Adott értékre megválaszolandó, hogy prímszám-e.
- Adott intervallumra az intervallumba eső prímszámok meghatározása.

17 ♦ Oldjuk meg a 16. feladatokat úgy hogy a lista elemenként 8 számot tartalmazzon (kivéve az utolsó elemet, amely kevesebbet is tartalmazhat)!

18 ♦ Egy időjárási adatcsoport egy napra vonatkozik, az év és az éven belül a nap sorszámával azonosítjuk (pl.: 1993. év 125. napjának adatai). A legkorábbi adat az 1900-as év 1. napjára vonatkozik, ennél korábbi adat nem lehetséges. Maximum 200 évre tervezünk. Az adatcsoport elemei:

- hőmérséklet reggel, délben, este;
- csapadék mennyisége (milliméter).

Az adatok egy az időadat szerint növekvően rendezett típusos fájlban állnak rendelkezésre, a funkciók egy belső listán valósítandók meg.

Megvalósítandó funkciók (szubrutinonként):

- Egy nap lekérdezése.
- Egy új időjárási adatcsoport felvitele.
- Egy időjárási adatcsoport módosítása (tévesen felvitt adat módosítása).
- Átlagértékek adott időszakra.

19 ♦ A 7.4. pontban specifikált tárgymutató adatstruktúrához oldjuk meg az alábbi feladatokat (feladatonként külön szubrutin):

- Új hivatkozás felvétele (`uListaO.Szofelvitel`).
- Új hivatkozás felvétele, de a hivatkozások érkezésének tetszőleges sorrendje mellett.
- Hivatkozás törlése. (Hivatkozás nélküli szó nem lehet a struktúrában.)
- Szó módosítása (egyediség, sorrend!).
- Hivatkozás módosítása (egyediség, sorrend!).
- Az adatstruktúra kimentése szövegfájlba (soronként egy szó és a hivatkozásai).
- Az adatstruktúra létrehozása és betöltése szövegfájlból (soronként egy szó és a hivatkozásai).

20 ♦ Egy áruházban több (maximum 10) pénztár van. Pénztáranként egy várakozó sor van. Egy vásárló maximum 100 darab árucikket vásárol, egy darab ára minimum 1 Ft, maximum 10000 Ft. A rendszer szimulációjához oldjuk meg az alábbi feladatokat (feladatonként külön szubrutinban):

- Inicializáljuk a rendszert (üres sorok és kasszák)!
- Új vásárló érkezik, sorsolt darabszámmal és fizetendő összeggel és beáll a legrövidebb sor végére.

- Új vásárló érkezik, sorsolt darabszámmal és fizetendő összeggel és beáll annak a sornak a végére, amelyikben a várakozóknál lévő árucikkek összes darabszáma minimális.
- Készen van egy adott sor első vásárlója, távozik a sorból.
- Pénztáranként az addigi bevétel meghatározása.
- Zárjuk a rendszert (a várakozók lekezelése)!

21 ♦ Egy áruházban több (maximum 10) pénztár van. Pénztáranként egy várakozó sor van. Egy vásárló maximum 100 darab árucikket vásárol, egy darab ára minimum 1 Ft, maximum 10000 Ft. A pénztáraknál a fizetés nem az érkezési sorrend szerint, hanem elsőbbségi (prioritásos) alapon történik. Elsőbbsége van annak a vásárlónak, aki kevesebb darabszámú árucikkel érkezik. A rendszer szimulációjához oldjuk meg az alábbi feladatokat (feladatonként külön szubrutinban):

- Inicializáljuk a rendszert (üres sorok és kasszák)!
- Új vásárló érkezik, sorsolt darabszámmal és fizetendő összeggel és beáll a prioritása szerint neki legkedvezőbb sorba, a prioritása szerinti helyre.
- Készen van egy adott sor első vásárlója, távozik a sorból.
- Pénztáranként az addigi bevétel meghatározása.
- Zárjuk a rendszert (a várakozók lekezelése)!

22 ♦ Oldjuk meg a 21. feladatokat úgy hogy a vásárló prioritását az egy árucikkre jutó fizetendő összeg határozza meg. Elsőbbsége van annak a vásárlónak, akinél ez a szám nagyobb.

23 ♦ Egy benzinkúton 5 fajta terméket forgalmaznak: B91, B95, B98, K95 és G. Mindegyik mértékegysége: egész liter. A benzinkúton 5 kiszolgálóhely van minden termék csak 1 kiszolgálóhelyen kapható. Ezek előtt állnak sorban a vásárlók. Egy vásárló minimum 1 és maximum 200 liter mennyiségben vásárol. A rendszer szimulációjához oldjuk meg az alábbi feladatokat (feladatonként külön szubrutinban):

- Inicializáljuk a rendszert (üres sorok és nulla kiszolgált mennyiségek)!
- Új vásárló érkezik, sorsolt árufajtával és vásárlási igénnyel, és beáll a neki megfelelő sor végére.
- Készen van egy sorsolt sor első vásárlója, távozik a sorból.
- Kiszolgálóhelyenként az addigi eladott mennyiség meghatározása.
- Zárjuk a rendszert (a várakozók lekezelése)!

24 ♦ Értelmező szótáron egy olyan adatrendszert értünk, amelyben egyrészt az alapszavak vannak, másrészt minden alapszóhoz adott egy értelmező szöveg, amely egy vagy több sorból állhat. A szótár egy szövegfájlban tárolt, alapszavanként:

- az első sor az alapszó, „*” jellel kezdve;
- ezután soronként az értelmező sorok (ezek nem kezdődhetnek „*” jellel).

Az értelmező sorok összeolvasva adják az értelmező szöveget, ezért sorrendjüket nem szabad megváltoztatni. A fájlban az alapszavak ábécé sorrendben követik egymást. A rendszer kezeléséhez oldjuk meg az alábbi feladatokat (feladatonként külön szubrutinban):

- A fájl betöltése egy összetett listába.
- A fájl újraírása a listából.
- Egy alapszó keresése.
- Egy új alapszó és értelmezése felvétele.
- Alapszó törlése.
- Egy alapszó adott sorszámú értelmező sorának törlése.
- Egy alapszó adott sorszámú értelmező sorának átírása.
- Egy alapszó értelmezésének bővítése új sorral.

25 ♦ Alapanyagokból keverékeket állítunk elő. Minden keveréknek van egy egyedi neve, amely max. 25 jelből áll, és betűvel kezdődik. Minden alapanyagnak van egy egyedi azonosító száma, amely egy ötjegyű pozitív egész szám. Egy keverék tetszőleges számú alapanyagból állhat, egy keveréket meghatároznak a benne szereplő alapanyagok mennyiségei alapanyagonként. Az adatrendszert egy szövegfájlban tároljuk, keverékenként:

- az első sor a név;
- ezután soronként egy alapanyag a mennyiségével.

A fájlban a keverékek ábécé sorrendben követik egymást. Egy keveréken belül az alapanyagok azonosítói is növekvően rendezettek A rendszer kezeléséhez oldjuk meg az alábbi feladatokat (feladatonként külön szubrutinban):

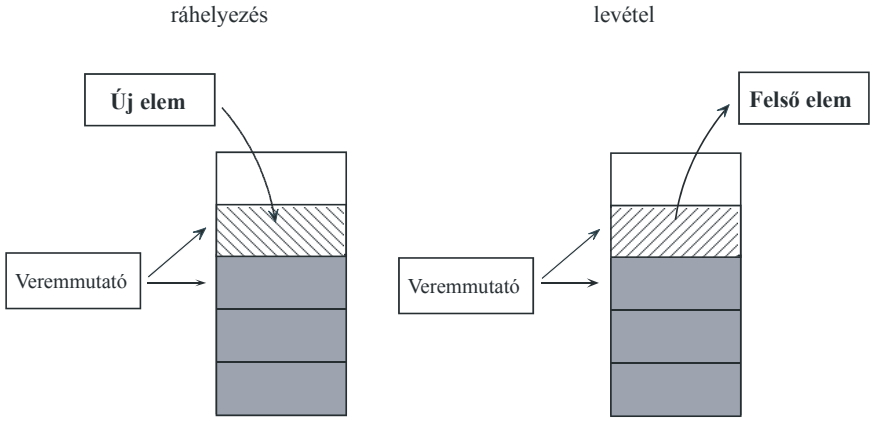
- A fájl betöltése egy összetett listába.
- A fájl újraírása a listából.
- Egy keverék keresése.
- Egy új keverék felvétele.
- Keverék törlése.
- Egy keverékhez egy új alapanyag felvétele.
- Egy keverékből egy alapanyag törlése.
- Egy keverékben egy alapanyag mennyiségének megváltoztatása.

8. VERMEK

8.1. Általános jellemzés

A *verem* adatstruktúra egy olyan összetett adatstruktúra, amelyet két művelet jellemez (38. ábra):

- Ráhelyezés, szakszóval *push* művelet: egy elemet teszünk a verembe, ez lesz a verem legfelső eleme.
- Levétel, szakszóval *pop* művelet: a verem legfelső elemét eltávolítjuk a veremből.



38. ábra. Veremműveletek

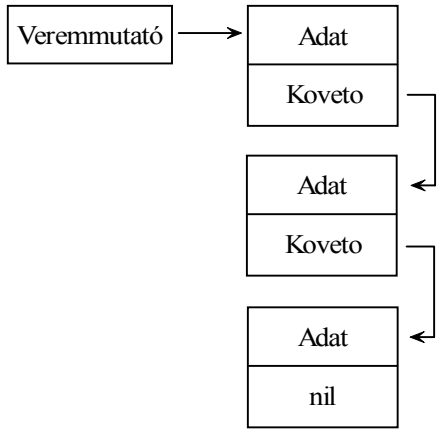
Verem adatstruktúra, mint standard adattípus nincs a Pascal nyelvben. Más nyelvi eszközökkel való implementációjánál figyelembe kell vennünk, hogy több levétel is jöhet egymás után, tehát egy levétel után is ismernünk kell a legfelső elemet (ha a verem nem üres). Következésképpen legegyszerűbb a vermet valamilyen egyszerű lineáris adatszerkezettel például egydimenziós tömbbel vagy egy irányban láncolt listával megvalósítani.

A tömbnél a verem tetején a tömb legnagyobb indexű (utolsó) eleme van, az aktuális elemszám mint veremmutató (a verem tetejére mutató index) funkcionál (38. ábra):

- Ráhelyezésnél a tömb aktuális elemszáma eggyel nő, az új elem az utolsó (legfelső) lesz.
- Levételnél az utolsó elemet vesszük le, az aktuális elemszám eggyel csökken.

A listánál a verem tetején a lista kezdőeleme van, a kezdőmutató tölti be a veremmutató szerepét 39. ábra:

- Ráhelyezésnél, az új elem a lista elejére kerül.
- Levételnél az első elemet vesszük le.



39. ábra. Listás verem

A verem a rekurzió, a rekurzív módon definiált algoritmusok megvalósításának jellemző eszköze. A rekurzív módon definiált algoritmusra egyik legegyszerűbb példa a Fibonacci féle számsorozat (`1`, `1`, `2`, `3`, `5`, `8`, `13`, ...) `N`-edik tagjának kiszámítása:

- Ha `N` < `3`, akkor az érték legyen `1`.
- Ha `N` >= `3`, akkor az értéket úgy kapjuk, hogy kiszámítjuk az `N` – `1`-edik és `N` – `2`-edik tagot és összeadjuk őket.

Ha a rekurzív algoritmus megvalósítását mint szubrutint képzeljük el, akkor ez egy olyan szubrutin, amely valamely pontján közvetlenül vagy közvetve (egy hívási lánc végén) meghívja önmagát. Hogy egy ilyen eljárás ne kerüljön végtelen ciklusba a következő két tulajdonsággal kell rendelkeznie:

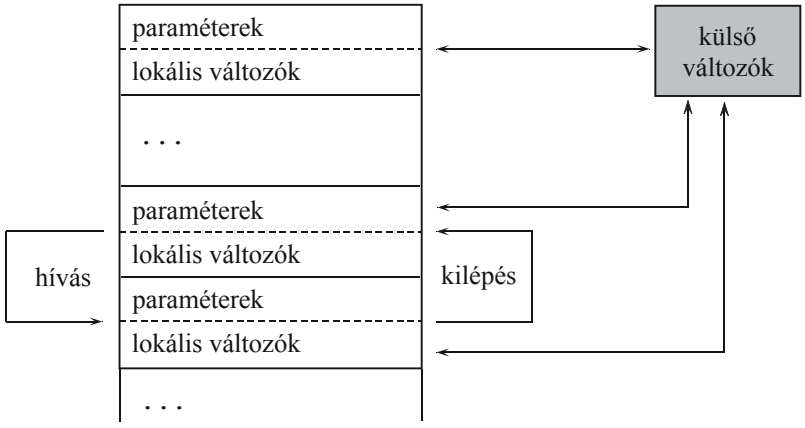
- Léteznie kell egy ún. báziskritériumnak, amelynek teljesülésekor már nem hívja meg önmagát (ez a példában az `N` < `3`).
- Egyre közelebb kell kerülni a báziskritérium teljesüléséhez (a példában: az `N` csökken).

Azokban a programozási nyelvekben, amelyekben a rekurzív hívás formálisan megengedett, a fordítóprogram általában verem adatstruktúra alkalmazásával

oldja fel a rekurziót, fordítja le a programunkat. A Pascal belső blokkokra, tehát a szubrutinokra vonatkozó szabályok következtében a szubrutin neve ismert, tehát *hivatkozható* azonosító *magában* a *szubrutinban*. Tehát lehetséges az, hogy a szubrutin a saját utasításrészében meghívja önmagát.

A rekurzív szubrutin működési módja az, hogy a kapott paraméterektől függően vagy befejezi működését vagy új paramétereket állít elő és ezekkel meghívja *(újraindítja)* önmagát. Az előbbi eset, a működés befejezése jelentheti a feladat megoldásának végét, vagy egy olyan pontra való visszatérést, ahonnan a megoldás során a szubrutint önmagából meghívtuk, azaz egy alacsonyabb (a báziskritériumhoz közelebbi) szintre való visszatérést, ahonnan a megoldás folytatódik.

A gépi *megvalósítás* szintjén ez azt jelenti, hogy a rekurzív hívásnál a szubrutin paraméterei és lokális változói által meghatározott adatcsoport egy *újabb példányban* megjelenik a *Stack* (verem) szegmensben, úgy hogy az eredeti (a hívó) példány is megmarad, és ha a hívottból nem tovább (újabb rekurzív hívással), hanem visszalépünk, akkor a hívó példány szituációjába jutunk vissza. A hívó és hívott a paraméterek mellett még a szubrutinban ismert nem saját (pl. globális) változókon keresztül is cserélhet információt (40. ábra). A külső változók alkalmazásának lehet célja a jobb áttekinthetőség is, de a fő indítéka ezek alkalmazásának az, hogy minél kevesebb, csak a feltétlenül szükséges változókat vonjunk be ismételten a verembe kerülő adatcsoportba.



40. ábra. Információcsere a hívó és a hívott szubrutin között

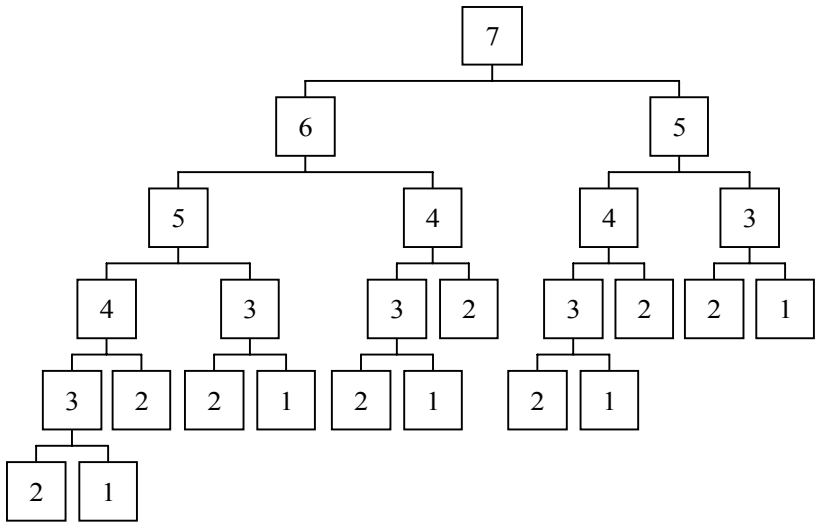
A hívási sorozat mélységét a *Stack* mérete korlátozza. (A rosszul programozott „végtelen” rekurzió futás közben pillanatok alatt teleírja a *Stack* szegmenst és hibaüzenettel megszakítja a program futását.)

Bár bizonyított az a tény, hogy minden algoritmus, amely rekurzív hívást tartalmaz, megvalósítható e nélkül is, vagyis a rekurzív szubrutinok *helyettesíthetők* a feladat megoldása szempontjából egyenértékű, rekurzív hívás nélküli szubrutinokkal is (pl. a verem programbeli implementálásával), sok esetben egyszerűbben*, áttekinthetőbben, a lényeget jobban kifejező, a tartalom és a forma egységét biztosító módon* leírható a feladat megoldása rekurzív hívással.

A rekurzivitás nagyon hatékony eszköz a *tömör, kifejező* programok írásához, de ha a gépi megvalósításra is gondolunk, akkor csak kellő körültekintéssel, meggondolással alkalmazható. Ennek szemléltetésére a fentebb már említett Fibonacci sorozatot hozzuk.

A fentebbi definíció közvetlenül átírható egy rekurzív Pascal függvénnyé:

```
function F(N: Longint): Longint;
begin
  if N<3 then F:=1 else F:=F(N-1)+F(N-2);
end;
```



41. ábra. F(7) kiszámításához szükséges hívások

Ennek a függvénynek a működését szemléltetjük a 41. ábrán, amely az `F(7)` kiszámításához szükséges hívásokat ábrázolja. Látjuk, hogy a szükséges hívások száma, tehát ezzel a számítás és tárigény az `N`-nel ugyanúgy nő, mint maga az `F(N)` érték, tehát exponenciálisan. Viszont egy egyszerű egyciklusos algoritmussal a sorozat tagjai jóval kisebb tár és számításigénnyel előállíthatók:

```
function F(N: Longint): Longint;
var
  I1, I2, E, I: Longint;
begin
 if N<3 then
    F:=1
 else begin
  I1:=1; I2:=1;
  for I:=3 to N do begin
     E:=I2+I1; I1:=I2; I2:=E
  end;
  F:=E;
  end;
end;
```

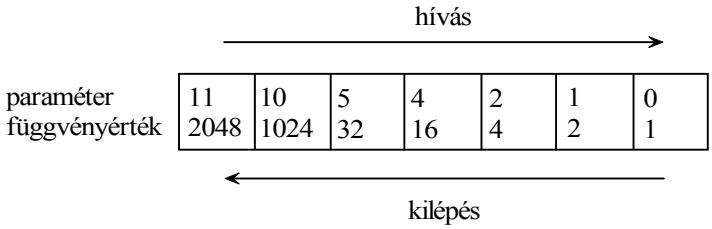
Ennek az algoritmusnak a tárigénye konstans, a számításigénye pedig az `n`-nel csak lineárisan nő. A példa, bár ellenpélda a rekurzió alkalmazására, jól szemlélteti a rekurzív szubrutinok működését.

8.2. Mintapéldák

8.2.1. mintafeladat: Készítsünk függvényt egész kitevős hatványozásra!

Útmutató ♦ A `Hatvany` függvény azon az egyszerű felismerésen alapszik, hogy ha egész kitevős hatványt számítunk, akkor (szorzásokat megtakarítandó) a már kiszámolt alacsonyabb hatványokat felhasználjuk a további számításoknál. (Pl. az ötödik hatványt úgy kapjuk, hogy a negyediket megszorozzuk az alappal, a negyediket a második négyzetre emelése adja.)

A függvény is így számol. Az „előremenő” szakaszban a rekurzív hívásokkal lebontva továbbadja paraméterként a kitevőket, a „visszatérő” szakaszban számolja és adja vissza a hatványokat (42. ábra).



42. ábra. A `2` a `11`-edik hatványon számolása

Adatszerkezet (68)

Azonosító	Funkció	Típus	Jelleg
`N`	a hatvány alapja	`Word`	input
`K`	a hatvány kitevője	`Byte`	input
`Hatvany`	a hatvány értéke	`Longint`	output

Szubrutin: `uVerem.Hatvany`.

8.2.2. mintafeladat: Rendezzünk egy tömböt növekvően a gyorsrendezés módszerével!

Útmutató ♦ Az elvi optimális megosztási eljárást a rendezett tömböknél már tárgyaltuk. A *gyorsrendezés* megosztási módszere a következő:

- Válasszunk egy „középértéket”, legyen ez a fizikailag (index szerint) középső elem értéke.
- A tömbben balról jobbra haladva keressük meg az első olyan elemet, amelyik nem kisebb, mint a középérték.
- A tömbben jobbról balra haladva keressük meg az első olyan elemet, amelyik nem nagyobb mint a középérték.
- A két elemet cseréljük fel.
- A kereséseket és a cseréket a cserélt elemektől az eredeti irányban továbbhaladva mindaddig ismételjük, amíg a két oldal nem találkozik.
- A találkozási pont két részre osztja a tömböt, a baloldalon a középértéknél nem nagyobb, a jobboldalon a középértéknél nem kisebb értékű elemek vannak csak.

Az optimális megosztás első két feltétele teljesül, a felezés itt sem garantált, de a módszerre vonatkozó elméleti és gyakorlati statisztikai vizsgálatok szerint [9] ez az egyik legjobb tömbrendező eljárás.

Néhány lépést szemléltetünk a 43. ábrán, ahol egy `12` elemű karaktertömböt rendezünk. A táblázaton a `KE` az aktuális középértéket, a `*` az aktuális cserepartnereket, a + a felosztás „nagyobb”, a − pedig a „kisebb” részét jelöli.

A fentebb vázolt *megosztásos* rendező algoritmus *rekurzív hívásos* megfogalmazása kézenfekvő:

- Ha a tömb elemszáma nagyobb, mint 1, osszuk két részre.
- A két részre külön-külön hajtsuk végre a megosztást.

1	2	3	4	5	6	7	8	9	10	11	12	KE			1	2	3	4
a	g	c	a	e	d	g	b	a	e	d	b	***d***		**BHI**	1			
	*										*			**JHI**	12			

1	2	3	4	5	6	7	8	9	10	11	12	KE			1	2	3	4
a	b	c	a	e	d	g	b	a	e	d	g	***d***		**BHI**	1			
				*						*				**JHI**	12			

1	2	3	4	5	6	7	8	9	10	11	12	KE			1	2	3	4
a	b	c	a	d	d	g	b	a	e	e	g	***d***		**BHI**	1			
					*			*						**JHI**	12			

1	2	3	4	5	6	7	8	9	10	11	12	KE			1	2	3	4
a	b	c	a	d	a	g	b	d	e	e	g	***d***		**BHI**	1			
						*	*							**JHI**	12			

1	2	3	4	5	6	7	8	9	10	11	12	KE			1	2	3	4
a	b	c	a	d	a	b	g	d	e	e	g	***d***		**BHI**	1	8		
-	-	-	-	-	-	-	+	+	+	+	+			**JHI**	7	12		

1	2	3	4	5	6	7	8	9	10	11	12	KE			1	2	3	4
a	b	c	a	d	a	b	e	d	e	g	g	***e***		**BHI**	1	8		
							-	-	+	+	+			**JHI**	7	12		

1	2	3	4	5	6	7	8	9	10	11	12	KE			1	2	3	4
a	b	c	a	d	a	b	e	d	e	g	g	***e***		**BHI**	1	10	8	
														JHI	7	12	9	

1	2	3	4	5	6	7	8	9	10	11	12	KE			1	2	3	4
a	b	c	a	d	a	b	d	e	e	g	g	***g***		**BHI**	1	10		
														JHI	7	12		

1	2	3	4	5	6	7	8	9	10	11	12	KE			1	2	3	4
a	a	c	b	d	a	b	d	e	e	g	g	***a***		**BHI**	1			
-	-	+	+	+	+	+								**JHI**	7			

1	2	3	4	5	6	7	8	9	10	11	12	KE			1	2	3	4
a	a	c	b	d	a	b	d	e	e	g	g	***a***		**BHI**	3	1		
														JHI	7	2		

43. ábra. 12 elemű karaktertömb rendezése

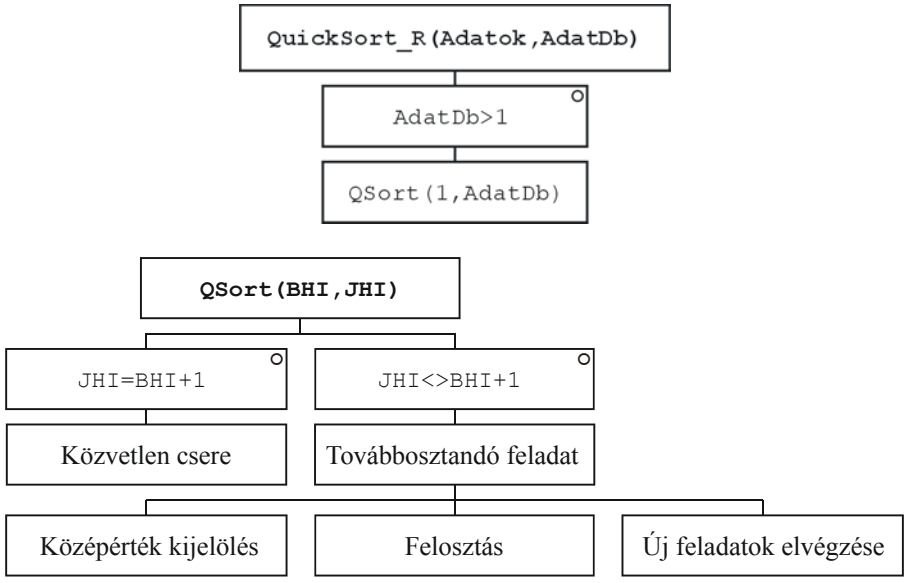
A szemléltető táblázatok jobboldalán találjuk az aktuálisan még hátralévő (nem teljesen rendezett) részeket, a bal és jobboldali határoló indexekkel megadva (`BHI`, `JHI`).

8.2.2.1 mintafeladat: Rendezzünk egy tömböt növekvően a gyorsrendezés módszerével, rekurzív hívással!

Adatszerkezet (69)

Azonosító	Funkció	Típus	Jelleg
`Adatok`	a rendezendő/rendezett tömb	`TSor`	input, output
`AdatDb`	a tömb aktuális elemszáma	`TElemDb`	input
`BHI`	aktuális megosztandó rész kezdete	`TEIndex`	munka
`JHI`	aktuális megosztandó rész vége	`TEIndex`	munka
`KE`	középérték	`TTElem`	munka
`BI`	megosztásnál balról haladó index	`TEIndex`	munka
`JI`	megosztásnál jobbról haladó index	`TEIndex`	munka

Struktúradiagram (69)



Szubrutin: `uVerem.QuickSort_R`.

Megjegyzések

- A `QuickSort_R` külső kereteljárás adja a rendezendő tömböt, a belső rekurzív eljárás paraméterei az eredeti, kiinduló tömb aktuális megosztandó részét jelölik ki. Ha az aktuális rész csak két elemből áll, akkor már nem osztunk tovább, hanem ha kell, közvetlenül cserélünk.
- Tárgazdálkodási szempontból hasznos a rekurzív hívások olyan rendezése, hogy a két rész közül mindig a rövidebb felé megyünk tovább, a hosszabb lesz félretéve. Bebizonyítható, hogy ebben az esetben a hívási mélység, vagyis az egy időpontban még meg nem osztott részek száma nem haladhatja meg a $\log_2 N$ értéket.

8.2.2.2 mintafeladat: Rendezzünk egy tömböt növekvően a gyorsrendezés módszerével, saját veremkezeléssel!

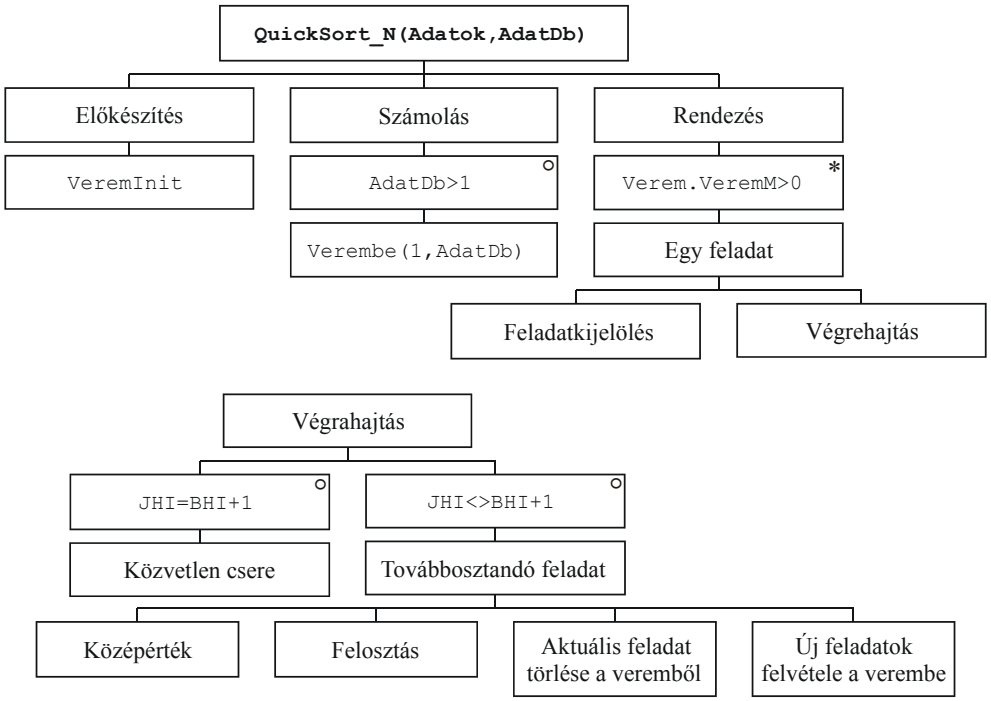
Útmutató ♦ Mivel a feladatot, vagyis a rendezendő tömbrészt egyértelműen meghatározza a kezdő és a vég (bal és jobb határoló) index, ezeket kell a verembe tenni. A vermet egy tömbbel valósítjuk meg, a tömb egy eleme egy indexpár. Ennek deklarációjánál kihasználtuk a fentebb említett maximum $\log_2 N$-es hívási mélységet, ami itt az egyszerre a veremben lévő feladatok maximális számát jelenti. A veremkezelés alapműveleteit belső eljárásokban adjuk meg: `VeremInit`, `VeremBe`, `VeremBol`.

Adatszerkezet (70)

```
const
  VeremMaxHossz=7;  {log2(EMaxDb)}
type
  TVeremInd=1..VeremMaxHossz;
  TVeremMutato=0..VeremMaxHossz;
  TVeremTomb=array[TVeremInd] of record
    BI, JI: TEIndex;
  end;
  TVerem=record
    VeremT: TVeremTomb;
    VeremM: TVeremMutato;
  end;
```

Azonosító	Funkció	Típus	Jelleg
Adatok	a rendezendő/rendezett tömb	TSor	input, output
AdatDb	a tömb aktuális elemszáma	TElemDb	input
Verem	a sajátkezelésű verem	TVerem	munka
BHI	aktuális megosztandó rész kezdete	TEIndex	munka
JHI	aktuális megosztandó rész vége	TEIndex	munka
KE	középérték	TTElem	munka
BI	megosztásnál balról haladó index	TEIndex	munka
JI	megosztásnál jobbról haladó index	TEIndex	munka

Struktúradiagram (70)



Szubrutin: `uVerem.QuickSort_N`.

Megjegyzés ♦ A verem működését konkrét példán szemléltető táblázatok jobboldalán láthatjuk.

8.2.3. mintafeladat: Állítsuk elő jelek összes különböző sorrendjét (permutációját) a következő algoritmussal:

- *Ha a jelek száma 1, akkor az egyetlen permutáció maga a jel!*
- *Ha a jelek száma N>1, akkor az összes permutáció előáll úgy, hogy az első N-1 jel összes permutációjába, minden lehetséges helyre beszúrjuk az N-edik elemet!*

Útmutató ♦ Egy szó (string) jeleit permutáljuk, ezek összes lehetséges sorrendjét állítjuk elő. Az új jel bevonásánál az előző lépés permutációi egy egyirányban láncolt listán vannak, az új permutációkat ennek elemeiből állítjuk elő és tesszük egy másik listára.

Adatszerkezet (71)

```
type
  TPerm=String; TPermDb=0..MaxLongint;
  TPermH=Byte; {hossz} TPermLista=PLElem1;
```

Azonosító	Funkció	Típus	Jelleg
`Szo`	a permutálandó jelek	`TPerm`	input
`Szavak`	a permutációk listája	`TPermLista`	output
`SzoDb`	a permutációk darabszáma	`TPermDb`	output
`Permutal`	a permutációk létrejötte	`Boolean`	output
`SzavakKezd`	`Szavak` kezdete	`PLElem1`	munka
`SzavakVeg`	`Szavak` vége	`PLElem1`	munka
`MunkaKezd`	munkalista kezdete	`PLElem1`	munka
`MunkaVeg`	munkalista vége	`PLElem1`	munka

Szubrutin: `uVerem.Permutal`.

Megjegyzés ♦ A rekurzív eljárás a belső eljárás (`Permutal_R`), a külső csak a keretet adja. A megosztás célja itt a jobb áttekinthetőség.

8.3. Feladatok

1 ♦ Számítsuk ki az `N` faktoriális (`N!` = `N` elem összes lehetséges sorrendjének darabszáma) értékét a következő definíció alapján:

- Ha `N = 0` akkor `N! := 1`.
- Ha `N > 0` akkor `N! :=N * (N - 1)!`.

Adjunk meg a megoldást:

- Rekurzív hívással.
- Saját veremmel.

2 ♦ Adott egy egész számokat tartalmazó tömb. Rendezzük a tömböt növekvő irányban, a kiválasztásos módszer rekurziós változatával, a következő definíció alapján:

- Válasszuk ki a minimális elemet és cseréljük ki az elsővel!
- Ha a maradék tömb legalább két elemű, alkalmazzuk rá az előző eljárást!

Adjunk meg a megoldást:

- Rekurzív hívással.
- Saját veremmel.

3 ♦ Állítsuk elő jelek összes különböző sorrendjét (permutációját) a következő algoritmussal:

- Ha a jelek száma `1`, akkor az egyetlen permutáció maga a jel.
- Ha a jelek száma `N > 1`, akkor az összes permutáció előáll úgy, hogy az első `N – 1` jel összes permutációjába minden lehetséges helyre beszúrjuk az `N`-edik elemet.

Adjunk meg a megoldást:

- Rekurzív hívással (`uVerem.Permutal`).
- Saját veremmel.

4 ♦ Adott egy egész számokat tartalmazó tömb. Rendezzük a tömböt növekvő irányban:

- A gyorsrendezés módszerével, rekurzív hívással (`uVerem.QuickSort_R`).
- A gyorsrendezés módszerével, saját veremmel (`uVerem.QuickSort_N`).

5 ♦ Hanoi tornyai: Legyen adott 3 függőleges rúd, `A`, `B` és `C`. Legyen az `A` rúdon – alulról felfelé csökkenő méretben `N` darab korong. Az `A`-ról a `C`-re kell átpakolni a korongokat a B használatával az alábbi szabályok betartása mellett:

- Egy lépésben csak egy, a rúdon legfelül lévő korong helyezhető át másik rúdra.
- Nagyobb korong nem tehető kisebbre.

Adjunk meg rekurziós megoldást (az átpakoló algoritmust):

- Rekurzív hívással.
- Saját veremmel.

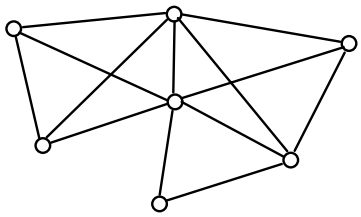
9. GRÁFOK ÉS FÁK

9.1. Általános jellemzés

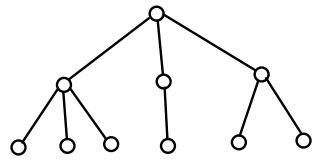
Az eddigiekben lineáris szerkezetű (tömb, lista) vagy ilyenekből összetett adatszerkezetekkel foglalkoztunk (összetett lista). Ezekben az elemek viszonya egymáshoz meglehetősen egyszerű, alapvetően sorrendiségről, egymásra következésről van szó. Ebben a fejezetben a *gráfokkal* foglalkozunk, amelyek már sokkal bonyolultabb és összetettebb viszonyok és kapcsolatrendszerek modellezésére is alkalmasak. A *gráf* egyben matematikai fogalom is, több matematikai tudományág tárgya vagy eszköze, de itt mi az alábbiakban nem matematikai jellegű definíciókat adunk, inkább a számítástechnikai modellekhez, a *programtervezéshez* elegendő pontosságú, intuitív fogalmak kialakítására törekszünk.

Gráfon bizonyos elemek és a köztük fennálló *közvetlen* kapcsolatok halmazát értjük. Két elem között elvben bármennyi közvetlen kapcsolat lehet (beleértve a közvetlen kapcsolat hiányát is). Nevezzük az elemeket *pontoknak,* a kapcsolatokat *éleknek*. Grafikus ábrázolásban a *pontokat* egyszerű geometriai alakzatokkal (kör, négyzet…), az *éleket* a pontokat összekötő szakaszokkal szoktuk jelölni (44. ábra). Nézzünk két példát:

- A pontok emberek, két pont között akkor van él, ha a két ember ismeri egymást. Lehet a megfelelő gráf akár a 44. ábra szerinti is. Viszont, ha egy olyan szervezetről van szó, ahol mindenki csak a saját közvetlen főnökét és saját közvetlen beosztottjait ismerheti, akkor a megfelelő gráf inkább a 45. ábra szerinti lesz (természetesen a helyes értelmezéshez meg kell jelölnünk még a fő főnököt, akinek már nincs főnöke. Az ilyen speciális struktúrájú gráfokat fa gráfoknak nevezzük, a megjelölendő pont a fa gyökérpontja, lásd alább.

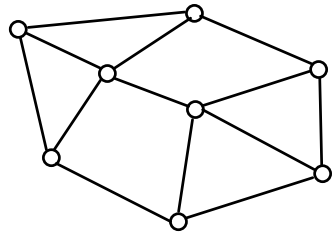


44. ábra. Általános gráf



45. ábra. Fa gráf

- Egy város utcahálózatának kereszteződései legyenek a pontok, az összekötő szakaszok az élek. Ilyen gráf lehet pl. a 46. ábra.

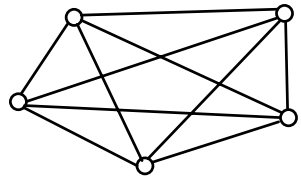


46. ábra. Városi utcahálózat gráf

Ezek után vezessünk be még néhány fogalmat, elnevezést:

Egyszerű gráf az olyan gráf, amelyben bármely két pont között legfeljebb egy él van és nincs hurokél (hurokél: egy pont önmagával való kapcsolata pl. egy olyan útszakasz, amelynek kezdő és végpontja azonos). Mi itt csak egyszerű gráfokkal foglalkozunk, példáink ilyenek, algoritmusaink is ezekre vonatkoznak.

Teljes gráf az olyan gráf, amelyben bármely két pont között van él. Például egy baráti társaság, ahol mindenki ismeri egymást (47. ábra).

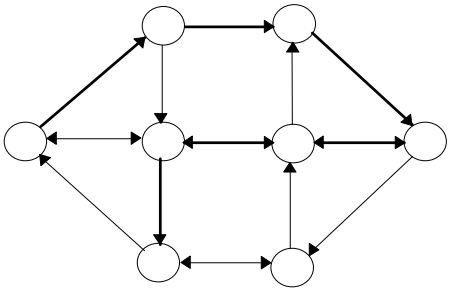


47. ábra. Teljes gráf

Irányított gráf az olyan gráf, amelyben az élekhez irányítást is rendelünk, kifejezve ezzel a nem feltétlenül szimmetrikus kapcsolatokat. Az él irányítása nyilvánvalóan éppen háromféle lehet, egy `a` és egy `b` pont viszonylatában:

- az `a`-ból a `b`-be mutató (csak az `a–b` kapcsolat áll fenn)
- a `b`-ből az `a`-ba mutató (csak a `b–a` kapcsolat áll fenn)
- mindkét irányban mutató (mind az `a–b`**,** mind a `b–a` kapcsolat fennáll)

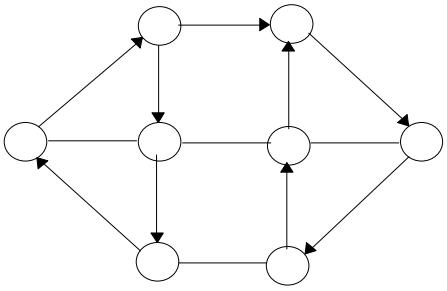
Például: a pontok közlekedési csomópontok, az `a`-ból `b`-be akkor mutat él, ha az ez irányú közlekedés lehetséges (az egyirányú útszakaszok nem szimmetrikus kapcsolatok). 48. ábra.



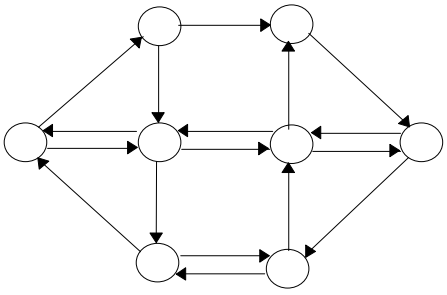
48. ábra. Irányított gráf

Egyszerű irányított gráf az olyan gráf, amelyben bármely két pont között, egy irányban legfeljebb egy él van és nincs hurokél.

Sok esetben egyszerűbb rajzban a szimmetrikus (mindkét irányú) kapcsolatokat az irányítás nélkül jelölni 49. ábra, máskor viszont kifejezőbb (gondoljunk például egy utcaszakasz két oldalára) a *kétirányú* él elnevezés helyett *két darab egyirányú* élről beszélni és a rajzi ábrázolásmódban is ezt követni (50. ábra).



49. ábra. Szimmetrikus kapcsolat
irányítás nélküli jelölése



50. ábra. Két darab egyirányú él

Út a gráf egy olyan pont illetve élsorozata, amelynek a felsorolás (bejárás) sorrendjében szomszédos pontjai közt van él. (48. ábrán a vastag vonal). A *körös út* olyan út, amelyben van legalább egy olyan pont, amely ismétlődik. *Körmentes út az, amelyben ilyen pont nincs.*

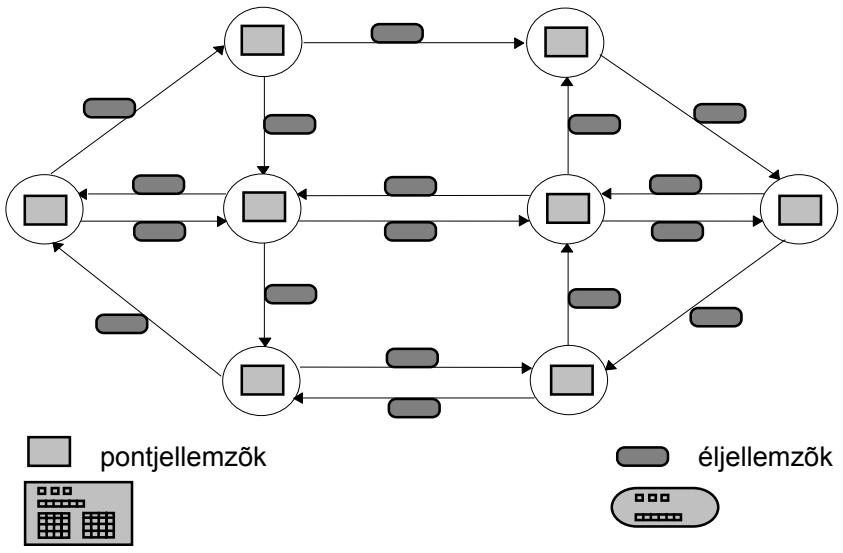
Összefüggő a gráf, ha bármely két pont között van út.

Fa struktúrájú gráf, röviden fa *az olyan gráf, amelyben körút csak élismétléssel hozható létre* (45. ábra).

Egy gráf egy *részgráfján* értjük a pontok és a köztük lévő élek egy részhalmazát.

A modellezendő problémák jelentős részében nem csak a pontok közti kapcsolatok megléte vagy hiánya fontos, hanem az is, hogy a kapcsolathoz egy *mennyiség,* mérőszám is tartozzon, például egy közlekedési hálózat egy éle milyen hosszú, vagy (egy átlagos sebességet feltételezve) mennyi idő alatt jutunk el az élen haladva az egyik pontból a másikba. Ha a modellünket ezzel bővítjük, vagyis az élekhez mérő, súlyozó számokat, más szóval *éljellemzőt* rendelünk, akkor már *hálózatról* beszélünk. Irányított gráf esetén ezek a számok irányonként is különbözhetnek, tehát egy kétirányú él két irányához tartozhat két különböző érték. Szokásos, hogy ha ez az egyértelműséget nem zavarja, a *hálózat* helyett is a *gráf* elnevezést használni.

Egy gráfot *szimmetrikusnak* nevezünk, ha irányítatlan, vagy minden él kétirányú.



51. ábra. Alaphálózati modell

Egy hálózatot egy éljellemzőre nézve *szimmetrikusnak* nevezünk, ha a gráf szimmetrikus és irányított esetben az éljellemző mind a két irányban ugyanolyan értékű.

Az alkalmazásokban nem csak az élekhez, hanem a pontokhoz is tartozhatnak ilyen számok, például egy közlekedési csomópont esetén a térképi koordináták vagy a forgalomirányítás módját (pl. lámpás, egyenrangú) leíró adatok.

A *hálózat* tehát a kapcsolatokat leíró *gráf* valamint a pontokhoz és az élekhez tartozó adatok, a *pontjellemzők* és az *éljellemzők* együttese (51. ábra). Ennek tárolási és feldolgozási kérdéseivel foglalkozunk.

9.2. Implementáció

9.2.1. Általános szempontok

A gráf, illetve a hálózat mint standard adattípus nem szerepel az univerzális jellegű programnyelvekben, így a Pascal nyelvben sem, viszont már elég bonyolult adatstruktúra ahhoz, hogy több értelmes és bizonyos (természetesen eltérő) szempontból optimális implementációja legyen.

A számítástechnikai modellekben elsődlegesen megoldandó részfeladat az *azonosítás,* tehát a hálózati pontokhoz és élekhez egyértelmű, és programmal kezelhető elnevezéseket, azonosítókat kell rendelnünk. Elsődleges a pontok azonosítása, az éleké ezekre már könnyen visszavezethető.

A pontok „természetes" vagyis a modellezendő feladatbeli azonosítója általában egy hosszú „beszélő" jellegű kód, mint pl. a személyi szám vagy közlekedési hálózatnál a keresztező két út neve. Ezt azokban a számítástechnikai modellekben, ahol a pontot indexként célszerű használni, egyszerűsítjük, egy sorszámmá, vagy sorszám jellegű adattá transzformáljuk. (Persze az *eredeti azonosító–sorszám* megfeleltetést külön tárolnunk kell, hogy az eredményadatokat az eredeti azonosítási rendszerben is közölni tudjuk.) Mi itt a példák egy részénél egyszerűen megbetűzzük a pontokat, valamelyik ponttól kezdve, *ábécé* sorrend szerint haladva és csak az angol ABC kisbetűit (`'a'..'z'`) használva. Nyilván így csak maximum 26 pontos hálózatokat vehetünk fel, de célunknak ez is megfelel, az elvek és módszerek szemléltetéséhez ez is elegendő. Nagyobb hálózatoknál, vagy ha az algoritmusok leírásához ez célszerűbb, a betűzés helyett a természetes számokkal való sorszámozást alkalmazhatjuk.

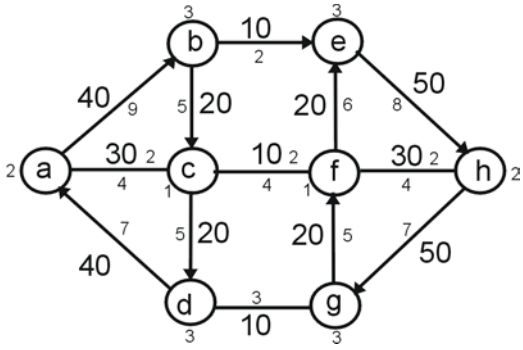
Az éleket az általuk összekapcsolt két ponttal azonosítjuk, tehát az `a` és `b` pont közti élet `ab` jelöli, illetve ha a gráf irányított akkor `ab` jelöli az `a`-ból a `b`-be mutató, `ba` pedig a fordított irányú élet (vagyis az él másik irányítását).

Az azonosítási kérdést megoldva már lehetséges, hogy kapcsolatot keressünk a hálózat és a programnyelvben (Pascal) leírható adatszerkezetek között.

Egy adatszerkezet megválasztásánál, minősítésénél számítástechnikai szempontból általában három fő tényező veendő tekintetbe:

- A *tárkihasználás* és az operatív tárban való *tárolhatóság*.
- A *karbantarthatóság* vagyis a változások átvezetésének műveletigénye.
- A *lekérdezhetőség* vagyis az információ kinyerés műveletigénye.

Az első tényező a számításigényes feladatoknál – és a hálózatkezelő algoritmusok ilyenek – kiemelt fontosságú, hiszen a háttértár használata nagyságrendekkel lassítja a számításokat. A második tényező fontossága attól függ milyen gyakoriak a változások. A harmadik tényezőt illetően, jellemző lekérdezési mód a pont és éljellemzők kötetlen, véletlenszerű sorrendű elérése.



52. ábra. Úthálózat modell

Három, alapelveiben gyökeresen eltérő implementációs lehetőséget mutatunk be és hasonlítunk össze a következőkben. A szemléltető kis példahálózat a 52. ábrán látható. Ezt egy úthálózat modelljének feltételezve, a gráfhoz három pontjellemzőt veszünk fel: `PT` a csomópont típusa (egyjegyű szám, az ábrán a pontokhoz írt kis számok), `PX` és `PY` a csomópont koordinátái (az ábrán nincs feltüntetve). Az éljellemzők száma kettő: egyik az útszakasz hossza (`ElH`, az ábrán az élek mellett a nagyobb számokkal feltüntetve), a másik az útszakasz közlekedési minősítését kifejező kategóriakód (`ElK`, az ábrán az élek mellett, a haladási irány szerinti jobb oldalon a kisebb számokkal feltüntetve). A példában feltételezzük, hogy a kétirányú éleknél a hossz mindkét irányban ugyanaz, a kategória irányonként eltérhet (pl. az egyik oldalon lehet parkolni, a másikon nem).

9.2.2. Mátrixreprezentáció

Közvetlenül adódik egy egyszerű, az alkalmazott legbonyolultabb tömbről *mátrixos* tárolási formának nevezett adatszerkezet. Ebben a pontjellemzőket a pontsorszámmal (nevezzük ezt *pontindexnek*) indexelve egydimenziós tömbökben, az éljellemzőket (irányítás szerint bontva) a kétdimenziós tömbökben, mátrixokban tároljuk. A példahálózat mátrixosan tárolt alakja a 1. táblázaton látható.

		PT	PX	PY
a	1	2	100	100
b	2	3	150	150
c	3	1	150	100
d	4	3	150	50
e	5	3	200	150
f	6	1	200	100
g	7	3	200	50
h	8	2	250	100

ElH	1	2	3	4	5	6	7	8
1	–	40	30	–	–	–	–	–
2	–	–	20	–	10	–	–	–
3	30	–	–	20	–	10	–	–
4	40	–	–	–	–	–	10	–
5	–	–	–	–	–	–	–	50
6	–	–	10	–	20	–	–	30
7	–	–	–	10	–	20	–	–
8	–	–	–	–	–	30	50	–

	ElK	1	2	3	4	5	6	7	8
a	1	–	9	4	–	–	–	–	–
b	2	–	–	5	–	2	–	–	–
c	3	2	–	–	5	–	4	–	–
d	4	7	–	–	–	–	–	3	–
e	5	–	–	–	–	–	–	–	8
f	6	–	–	2	–	6	–	–	4
g	7	–	–	–	3	–	5	–	–
h	8	–	–	–	–	–	2	7	–

1. táblázat. A hálózat mátrixosan tárolt alakja

A mátrix *tárkihasználása* erősen függ a gráftól. Mint könnyen beláthatjuk, legjobb a tárkihasználás a teljes gráfok esetén, hiszen ezeknél csak a főátló tartalma a redundáns információ. Az ún. ritka gráfoknál, amelyeknél az elvben lehetséges kapcsolatoknak csak nagyon kis része valósul meg, a mátrix ebből a szempontból nagyon gazdaságtalan. Például egy `1000` pontos hálózat `999000` irányított élet tartalmazhatna, de ha ez pl. egy közlekedési hálózat, akkor az élek száma nagy valószínűséggel `5000` alatt van, tehát a kihasználtság az `1` százalékot sem éri el. Az operatív tárban való *tárolhatóság* csak nem nagy hálózatoknál kivitelezhető. Példának véve `3` darab, egyenként `4` bájton tárolható egész típusú éljellemzőt, ezek mátrixai `1000` pontos hálózatnál `3 * 4` millió bájtot igényelnek, ez egyszerűbb PC-n is teljesíthető, de `10000` pont esetén a `3 * 400` millió bájtos területhez már lényegesen komolyabb kiépítettség kell.

A mátrix *karbantarthatósága* kedvező. Élet felvenni, törölni, él vagy pontjellemzőt módosítani nagyon egyszerű, hiszen csak át kell írni egy tömbelemet. Az új pont felvétele vagy törlése már műveletigényesebb, hiszen ez sorok és oszlopok mozgatását igényli.

A mátrix *lekérdezhetősége* optimális, a lehető leggyorsabb és legegyszerűbb módon jutunk el az azonosítótól a jellemzőig. Pontjellemzőnél ez egy, éljellemzőnél két indexelést jelent.

Sajnos, a gyakorlati méretű feladatok többségénél előnyei ellenére sem választhatjuk a mátrixtárolást a hatalmas tárigény miatt. Keresni kell a tárolási redundancia csökkentését, esetleg a másik két szempont rovására is.

9.2.3. Éltárolás

A következő, csak egydimenziós tömböket alkalmazó módszer lényege az *éltárolás,* vagyis az az elv, hogy csak a *ténylegesen* meglévő kapcsolatok jellemzőit tároljuk, az élek bizonyos rendezettsége szerint. Az adatszerkezetet a példahálózatra a 2. táblázat szemlélteti. Mint ebből látható, az éljellemzőket tömören, egy a kezdőpont (`Kp`) szerint rendezett sorban tároljuk, az `M` élmutatóérték mondja meg, hogy a sornak mely szakasza tartozik egy-egy kezdőponthoz, és ez a `Vp` pontindex értékkel együtt határozza meg az élet.

Kp	**a**	**b**	**c**	**d**	**e**	**f**	**g**	**h**
	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
PT	2	3	1	3	3	1	3	2
PX	100	150	150	150	200	200	200	250
PY	100	150	100	50	150	100	50	100

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**
M	1	3	5	8	10	11	14	16

	1	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Vp	2	3	3	5	1	4	6	1	7
ElH	40	30	20	10	30	20	10	40	10
ElK	9	4	5	2	2	5	4	7	3

	10	**11**	**12**	**13**	**14**	**15**	**16**	**17**
Vp	8	3	5	8	4	6	6	7
ElH	50	10	20	30	10	20	30	50
ElK	8	2	6	4	3	5	2	7

2. táblázat. A példahálózat éltárolásos adatszerkezete

Az éltárolás *tárkihasználása* nagyon jó, a *memóriaigény* az élek számával csak lineárisan nő, több ezer, sőt több tízezer élet tartalmazó hálózat is tárolható a ma már elérhető, átlagosnak tekinthető személyi számítógépes memóriaméretben (32–64 Mbájt).

A számítástechnikai modellezésnél általánosan érvényesül az, hogy tárat csak idő, időt pedig tár árán nyerhetünk. Várható tehát, hogy az éltárolás *karbantarthatósága* és *lekérdezhetősége* a mátrixénál rosszabb lesz. Akár új élet veszünk fel, akár meglévőt törölünk (vagy pontot törlünk), a tömör tárolás miatt a tömbelemek egy részét meg kell mozgatni, át kell pakolni. Az éljellemzők eléréséhez a kezdőpontsorszámmal az `M` mutatót indexelve a neki megfelelő részszakaszt ugyan egy lépésben elérjük, de innen kiindulva a már keresni kell a végpontot. A keresés meggyorsítása céljából célszerű a `Vp` soron belül az azonos kezdőponthoz tartozó végpontindexeket rendezetten tartani (mint a példában is). Minél ritkább a hálózat, annál gyorsabb átlagosan a lekérdezés.

9.2.4. Dinamikus adatszerkezet

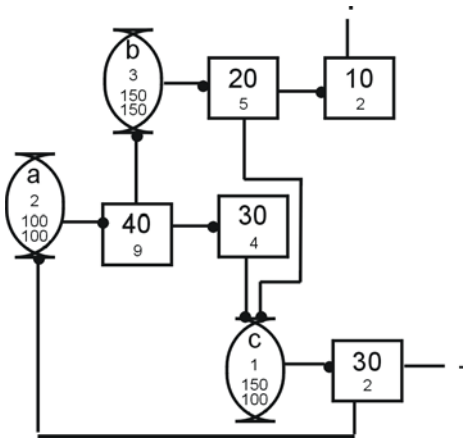
Ha egy gráfra, mint rajzra nézünk, természetesnek tűnik a kapcsolatokat mutatók formájában leképező dinamikus adatszerkezetek használata. Ez az általános gráfoknál azonban nem olyan egyszerű, mint elsőre látszik. Ha csak egyféle elemet (rekordot) akarunk alkalmazni, akkor a tárolás eléggé redundáns lesz. Válasszuk például a gráf pontját alapelemnek a dinamikus szerkezetben. Rögtön adódik a probléma, hogy hány élnek (pontosan: hány mutatónak és éljellemzőnek) legyen helye a pontrekordban. Az elvi maximumnak helyet fenntartva a mátrixtárolással egyenértékűen rossz helykihasználást kapunk. (Speciális gráfoknál, pl. a későbbiekben tárgyalandó bináris fáknál persze lehet egészen más is a helyzet.)

Célszerűbb kétféle, egy pont és egy élrekordból építkezni, a jellemzőket a rekordokban tárolva, a kapcsolatokat mutatókkal megvalósítva (az 53. ábrán csak a hálózat egy része szerepel, de a módszer ebből is látható).

A megfelelő deklarációs séma:

```
type
  TGrafPontAzon=pontazonosító adattípus;
  TGrafPontJell=pontjellemző adattípus;
  TGrafElJell=éljellemző adattípus;

  PGrafPont=^TGrafPont;
  PGrafEl=^TGrafEl;
```

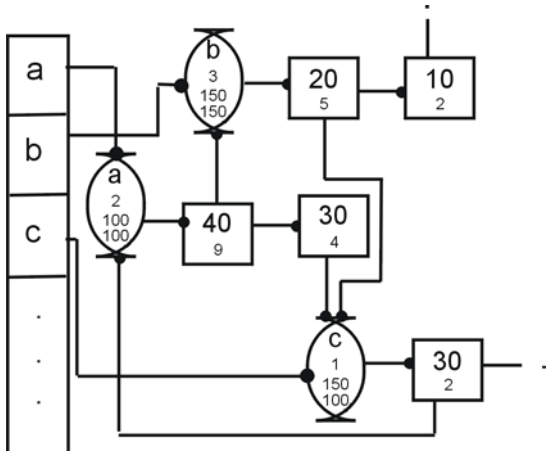
```
TGrafPont=record
  Azon: TGrafPontAzon;
  PontJell: TGrafPontJell;
  KezdEl: PGrafEl;
end;

TGrafEl=record
  ElJell: TGrafElJell;
  KovEl: PGrafEl;
  ElVegPont: PGrafPont;
end;
```



53. ábra. A gráf dinamikus adatszerkezettel

Ennél a tárigény kedvező, hiszen az adatokon kívül csak a mutatóknak kell tárhely. A karbantarthatóság is jó, hiszen ebben élet, pontot felvenni vagy törölni egyszerűen lehet, az adatokat nem kell mozgatni, a funkciók adatok és mutatók felvételével, törlésével megoldhatók. A lekérdezhetőség már kevésbé jó, hiszen listákon való lépkedéssel jutunk el az adatokhoz, ami mindig lassúbb, mint a tömbökben való keresés.

Természetesen a hatékonyabb karbantarthatóság és lekérdezhetőség céljából – újabb adatszerkezeti elemek felvételével – további tárolási módokat is alkalmazhatunk. Erre csak egy példát hozunk: a dinamikus szerkezetet egy olyan tömbbel (vagy listával) bővítjük, amely minden ponthoz a pontra mutató mutatót tartalmaz, az élkeresést lerövidíthetjük, hiszen a kezdőpontot gyorsan megtaláljuk (54. ábra).



54. ábra. Pontra mutatóval bővített dinamikus adatszerkezet

9.3. Alapfeladatok

Az elemi karbantartási és lekérdezési feladatok szemléltetésére egy kisméretű gráfot deklarálunk kétféleképpen, egyrészt a mátrix, másrészt az éltárolás módszerével (`uGrafD`). A pontok azonosítására az angol ábécé kisbetűit használjuk. Pontjellemzőként egy típuskódot és két koordinátát veszünk fel, éljellemzőként egy kategóriakódot és egy élhossz adatot választunk. A gráf irányított.

```
const
  GMinPontAz='a';
  GMaxPontAz='z';
  GMaxPontDb=Ord(GMaxPontAz)-Ord(GMinPontAz)+1;
  GMaxElDb=GMaxPontDb*(GmaxPontDb-1);
  GMaxPontTip=6;
  GMaxElKat=9;
  GMaxElHossz=999;

type
  TGPontAz=GMinPontAz..GMaxPontAz;  {pontazonosító}
  TGKoord=Integer;  {koordináta pontjellemző}
  TGPontTip=1..GMaxPontTip;  {ponttípus pontjellemző}
  TGElKat=1..GMaxElKat;  {kategória éljellemző}
  TGElHossz=-1..GMaxElHossz;  {hossz éljellemző}
  {a -1 a nemlétező él hossza, a mátrixtárolásnál szükséges}
```

```
TGPont=record  {pontadat: azonosító, típus, koordináták}
 Azon: TGPontAz;
 Tip: TGPontTip;
 X, Y: TGKoord;
end;

TGPontDb=0..GMaxPontDb;  {pontok száma}
TGPontI=1..GMaxPontDb;  {pontindex}
TGPontI01=0..GMaxPontDb+1;  {bővített pontindex}
TGPontI0=0..GMaxPontDb;  {bővített pontindex}
TGPontok=array[TGPontI] of TGPont;  {pontok adatai}

{*gráf mátrixtárolással}
{hossz-mátrix}
TGElHosszMtx=array[TGPontI, TGPontI] of TGElHossz;
{kategória-mátrix}
TGElKatMtx=array[TGPontI, TGPontI] of TGElKat;

TGrafMtx=record
  PontDb: TGPontDb;  {pontok száma}
  Pontok: TGPontok;  {pontadatok, azonosító szerint
    növekvően rendezve}
  ElHossz: TGElHosszMtx;  {hossz-mátrix, -1 hossz esetén
    nincs él}
  ElKat: TGElKatMtx;  {kategória-mátrix}
end;
```

A *mátrixtároláshoz,* a pontosság és egyértelműség kedvéért két kiegészítő megjegyzés:

- A `Pontok`-ban a pontadatok pontazonosító szerint növekvően rendezetten vannak tárolva.
- A nem létező élet a `Elhossz` mátrixban a `-1` érték jelzi. Ilyen esetben az `ElKat`-beli megfelelő érték nem használatos, a típusba beleférő bármilyen kategóriaérték lehet.

```
type
  {*gráf éltárolással}
  TGEl=record  {éladat: végpontindex, hossz, kategória}
    VpInd: TGPontI;
    ElHossz: TGElHossz;
    ElKat: TGElKat;
  end;
```

```
TGElDb=0..GMaxElDb;  {élek száma}
TGElI=1..GMaxElDb;  {élindex}
TGElI01=0..GMaxElDb+1;  {bővített élindex}
TGElek=array[TGElI] of TGEl;  {élek adatai}

{élmutatók tömbje}
TGPontI1=1..GMaxPontDb+1;  {index}
TGElI1=1..GMaxElDb+1;  {elem}
TGElMut=array[TGPontI1] of TGElI1;  {tömb}

TGrafElT=record
  PontDb: TGPontDb;  {pontok száma}
  Pontok: TGPontok;  {pontadatok, azonosító szerint
    növekvően rendezve}
  ElMut: TGElMut;  {élmutatók}
  Eldb: TGElDb;  {élek száma}
  Elek: TGElek;  {éladatok}
end;
```

Az *éltároláshoz,* a pontosság és egyértelműség kedvéért néhány kiegészítő megjegyzés:

- A `Pontok`-ban a pontadatok pontazonosító szerint növekvően rendezetten vannak tárolva.
- Az `Elek`-ben az `I` indexű kezdőponthoz tartozó élek adatai az `ElMut[I]..ElMut[I+1]-1` intervallumba tartozó indexekkel érhetők el. Hogy ez a szabály a legnagyobb indexű pontra is alkalmazható legyen, és így az algoritmusokban ne kelljen külön kezelni az utolsó pontot, az `ElMut` tömböt kiegészítjük egy `ElMut[PontDb+1]=ElDb+1` értékű elemmel, lásd `TGPontI1` típus.
- Az `Elek`-ben az egy kezdőponthoz tartozó élek végpont-azonosító szerint növekvően rendezettek.
- Ha valamely pontból mint kezdőpontból nincs kiinduló él, akkor az `ElMut`-ban a ponthoz tartozó mutató értéke azonos a sorrendben következő ponthoz tartozó mutatóértékkel.
- Ha a legnagyobb azonosítójú (utolsó) pontból, mint kezdőpontból nincs kiinduló él, akkor az `ElMut`-ban a hozzá tartozó mutató a legutolsó létező él utánra mutat (`ElDb+1` értékű lesz, lásd `TGElI1` típus).

A hálózati adatokat szövegfájlban archiváljuk (kimentés, visszatöltés), a következő szerkezetben:

- első sor: a *pontok darabszáma,* a sor elejéhez igazítva, ezután:
- pontonként egy *pontadat sor,* az azonosító szerinti növekvő sorrendben, ezután:
- az *élek darabszáma,* a sor elejéhez igazítva, ezután:
- élenként egy *éladat sor,* kezdőpont-azonosító és azon belül végpont-azonosító szerinti növekvő sorrendben.

A *pontadat* sorok formátuma és tartalma, a mezők sorrendjében:

Sorszám	Tartalom	Hossz	Formátum	Igazítás
1	azonosító	1	karakter	
2	típus	3	egész szám	jobbra
3	x koordináta	13	egész szám	jobbra
4	y koordináta	13	egész szám	jobbra

Az *éladat* sorok formátuma és tartalma a mezők sorrendjében:

Sorszám	Tartalom	Hossz	Formátum	Igazítás
1	kezdőpont azonosító	1	karakter	
2	üres	1	karakter	
3	végpont azonosító	1	karakter	
4	élhossz	6	egész szám	jobbra
5	kategória	3	egész szám	jobbra

9.3.1. mintafeladat: Keressünk meg egy adott pontazonosító indexét!

Útmutató ♦ A megoldáshoz csak a pontadatok szükségesek, ezek viszont mind a két reprezentációban azonosan tároltak. A rendezettség lehetővé teszi a bináris keresés alkalmazását. Feltételezzük, hogy az adott azonosító formálisan helyes (kisbetű).

Adatszerkezet (72)

Azonosító	Funkció	Típus	Jelleg
`PontDb`	a pontok száma	`TGPontDb`	input
`Pontok`	pontadatok	`TGPontok`	input
`PontAzon`	a keresett azonosító	`TGPontAz`	input
`Van`	a keresett azonosító létezése	`Boolean`	output
`PontIndKeres`	a keresett azonosító indexe vagy a helye	`TGPontI`	output
`E, V, Hol`	a keresés munkaváltozói	`TGPontDb`	munka

Szubrutin: `uGrafKez.PontIndKeres`.

Megjegyzés ♦ Ha nincs az adott azonosítójú pont a gráfban, a szubrutin eredménye a pont helye (indexe) a `Pontok` tömbben.

9.3.2. mintafeladat: Vegyünk fel egy új pontot a gráfba (éltárolás)!

Útmutató ♦ Az új pontot él nélkül vesszük fel (az élfelvétel egy másik feladat lesz). A tömböket megfelelően átalakítjuk. Ha a pont már létezik a gráfban, akkor a funkció csak a pontadatok átírása lesz.

Adatszerkezet (73)

Azonosító	Funkció	Típus	Jelleg
`Graf`	a hálózat	`TGrafElT`	input, output
`Pont`	a felveendő pont adatai	`TGPont`	input
`PontInd`	a pont indexe	`TGPontI`	munka
`Van`	a pont létezése	`Boolean`	munka
`I`	pontindex ciklusváltozó	`TGPontI01`	munka
`L`	élindex ciklusváltozó	`TGElDb`	munka

Szubrutin: `uGrafKez.PontFelVeszElt`.

9.3.3. mintafeladat: Töröljünk egy pontot a gráfból (mátrix)!

Útmutató ♦ A ponttal együtt törölnünk kell az összes csatlakozó élet is, ez a mátrixokban egy sor és egy oszlop törlését jelenti.

Adatszerkezet (74)

Azonosító	Funkció	Típus	Jelleg
`Graf`	a hálózat	`TGrafMtx`	input, output
`PontInd`	a pont indexe	`TGPontI`	input
`I, J`	pontindex ciklusváltozók	`TGPontI1`	munka

Szubrutin: `uGrafKez.PontTorolMtx`.

9.3.4. mintafeladat: Keressünk meg egy, a kezdő- és végpontindexével meghatározott élet (éltárolás)!

Útmutató ♦ A kezdőpontonkénti rendezettség lehetővé teszi a bináris keresés alkalmazását.

Adatszerkezet (75)

Azonosító	Funkció	Típus	Jelleg
`Graf`	a gráf	`TGrafElt`	input
`KpInd`	a kezdőpont indexe	`TGPontI`	input
`VpInd`	a végpont indexe	`TGPontI`	input
`Van`	a keresett él létezése	`Boolean`	output
`ElKeresElt`	a keresett él indexe vagy a helye	`TGElI`	output
`E, V, Hol`	a keresés munkaváltozói	`TGElDb`	munka

Szubrutin: `uGrafKez.ElKeresElt`.

Megjegyzés ♦ Ha nincs a specifikált él a gráfban, a szubrutin eredménye az él helye (indexe) az `Elek` tömbben.

9.3.5. mintafeladat: Vegyünk fel egy új élet a gráfba (éltárolás)!

Útmutató ♦ Az új élet a kezdőpont indexével és a többi éladattal (köztük a végpont indexével) adjuk meg. A tömböket megfelelően átalakítjuk. Ha az él már létezik a gráfban, akkor a funkció csak az éladatok átírása lesz.

Adatszerkezet (76)

Azonosító	Funkció	Típus	Jelleg
`Graf`	a hálózat	`TGrafElT`	input, output
`KpInd`	a kezdőpont indexe	`TGPontI`	input
`El`	a felveendő él adatai	`TGEl`	input
`ElInd`	az él indexe	`TGElI`	munka
`Van`	az él létezése	`Boolean`	munka
`I`	pontindex ciklusváltozó	`TGPontI1`	munka
`L`	élindex ciklusváltozó	`TGElDb`	munka

Szubrutin: `uGrafKez.ElFelVeszElt`.

9.3.6. mintafeladat: Mentsük ki a hálózati adatokat szövegfájlba (éltárolás)!

Útmutató ♦ A belső adatstruktúra tömbjein végighaladva előállítjuk a fentebb leírt formátumú szövegfájlt. A fájl előállíthatóságát a hívó ellenőrzi.

Adatszerkezet (77)

Azonosító	Funkció	Típus	Jelleg
`Graf`	a hálózat	`TGrafElT`	input
`F`	a szövegfájl	`text`	output
`i`	pontszámláló	`TGPontDb`	munka
`l`	élszámláló	`TGElDb`	munka

Szubrutin: `uGrafKez.MentesElt`.

9.3.7. mintafeladat: Töltsük be a hálózati adatokat szövegfájlból (mátrixtárolás)!

Útmutató ♦ Feltételezzük, hogy a fájl létezik, valamint tartalmilag és formailag is hibátlan (adatellenőrzést nem végzünk). A beolvasás előtt az éljellemzők mátrixait ellátjuk megfelelő kezdőértékekkel.

Adatszerkezet (78)

Azonosító	Funkció	Típus	Jelleg
`Graf`	a hálózat	`TGrafMtx`	output
`F`	a szövegfájl	`Text`	input
`I, J`	pontszámlálók	`TGPontI01`	munka
`Kezdo,Veg`	pontazonosítók	`TGPontAz`	munka
`L, Db`	élszámlálók	`TGElDb`	munka
`Van`	pont létezése	`Boolean`	munka
`H`	élhossz	`TGElHossz`	munka
`Kat`	élkategória	`TgElKat`	munka
`C`	elválasztó karakter	`Char`	munka

Szubrutin: `uGrafKez.BetoltesMtx`.

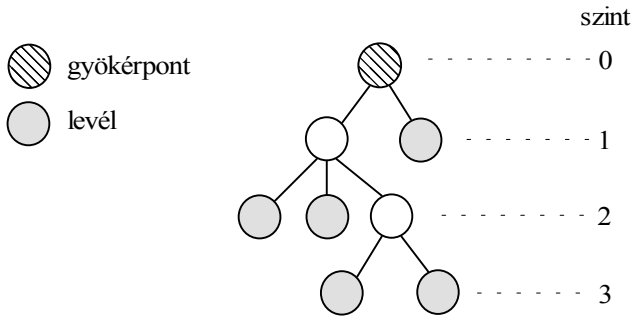
Megjegyzés ♦ A kezdeti feltételezések miatt a `Van` változó csak szintaktikailag szükséges.

9.4. Fák

9.4.1. Általános jellemzés

Mint már korábban is definiáltuk, *fa struktúrájú gráf,* röviden *fa* az olyan gráf, amelyben körút csak élismétléssel hozható létre.

Gyökeres fa az olyan fa, amelyben egy pontot kitüntetünk. Ezt a fa gyökérpontjának (root) nevezzük. A gyökérponthoz nem értelmezünk megelőző pontot. Minden más pontnak egyértelműen definiálható a *megelőzője* (más terminológiával szülő, vagy ős). Egy pont megelőzője a fában, a hozzá a gyökérpontból vezető (a fa struktúra következtében egyértelműen meghatározott) körmentes útban az őt megelőző pont. Egy pont *követője* (más terminológiával rákövetkező, gyermek vagy leszármazott) az a pont, amelynek ő a megelőzője. A fa struktúrából következően egy pontnak több leszármazottja is lehet, de szülője csak egy, a gyökérpontnak pedig nincs szülője. (55. ábra). Az olyan pontot, amelyeknek nincs követője, *levélnek* is hívjuk. Megjegyezzük, hogy a fák rajzos ábrázolásánál a matematikai és informatikai szakirodalomban a „fordított" állás a szokásos, tehát a gyökérpontot rajzoljuk legfelülre, a fa lefele ágazik el.



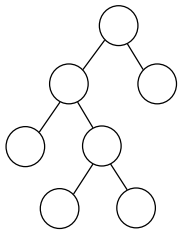
55. ábra. Gyökeres fa

A gyökeres fánál az irányítás fogalmát is megfelelően kell értelmezni. Két pont között két egyirányú él eleve ellentmondana a fa definíciójának. A gyökeres fa definíciójából következően pedig, ha valamilyen okból egyáltalán szükség lenne az irányított élre, akkor ezt minden élre ugyanúgy kell értelmezni, tehát vagy minden él a szülőtől a gyermek felé, vagy minden él a gyermektől a szülő felé van irányítva.

Gyökeres fa esetén értelmezhetjük a *szintek* és a *magasság* fogalmát is. A nulladik szint a gyökérpont szintje, első szinten vannak a gyökérpont leszármazottai, a második szinten vannak az első szintű pontok leszármazottai és így tovább. A fa magassága a legnagyobb szintszám értéke. Könnyen látható, hogy egy gyökeres fa bármely pontja, ha őt gyökérpontnak tekintjük, szintén meghatároz egy gyökeres fát. Ezt az eredeti fa *részfájának* is nevezhetjük. A következőkben, ha fáról beszélünk, ezen mindig gyökeres fát értünk, ha nem ilyenről van szó, azt külön jelezzük.

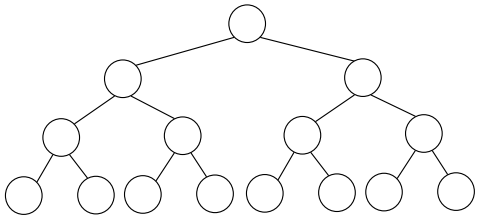
Rendezett fa az olyan gyökeres fa, amelyben a pontok leszármazottai között valamilyen sorrendet értelmezünk, tehát beszélünk első, második… leszármazottról, vagy ha két leszármazott van, akkor pl. baloldali és jobboldali leszármazottról. Külön felhívjuk a figyelmet, hogy ez a fogalom *nem tévesztendő össze* a pontokhoz vagy élekhez rendelt értékek viszonyával! (De szükséges a fás adatrendezések és keresések pontos értelmezéséhez.)

Bináris fa az olyan rendezett fa, amelyben egy pontnak legfeljebb kettő leszármazottja van. Az egyik leszármazottat *bal* leszármazottnak, az általa meghatározott részfát *bal részfának* nevezzük. A másik leszármazottat *jobb* leszármazottnak, az általa meghatározott részfát *jobb részfának* nevezzük. A definíció értelmében egy-egy konkrét pontnál akár az egyik, akár a másik, akár mind a kettő hiányozhat (56. ábra).

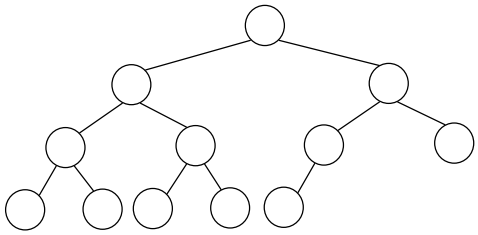


56. ábra. Bináris fa

Teljes bináris fa az olyan bináris fa, amelyben az utolsó szinten lévő pontokat kivéve, minden pontnak megvan mind a két leszármazottja, és az utolsó szinten csak levelek vannak (57. ábra). *Majdnem teljes bináris fa* az olyan bináris fa, amelyben az utolsó és az utolsó előtti szinten lévő pontokat kivéve, minden pontnak megvan mind a két leszármazottja, az utolsó szinten csak levelek vannak, az utolsó előtti szinten is lehetnek levelek, ha ilyenek vannak, akkor azok (a fát mint rendezett fát tekintve) mind jobbra esnek a szint nem levél jellegű pontjaitól (58. ábra).



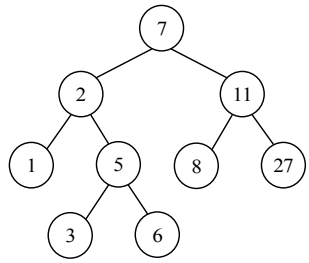
57. ábra. Teljes bináris fa



58. ábra. Majdnem teljes bináris fa

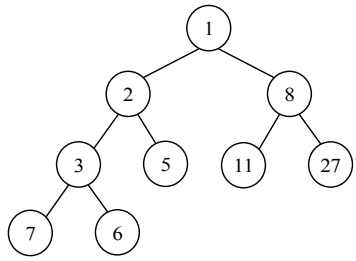
A pontokhoz és/vagy az élekhez jellemzőket, tehát értékeket, adatokat rendelve a fa adatstruktúra egy széleskörűen alkalmazott és nagy hatékonyságú eszköz a rendezési és keresési feladatok megoldásánál. Az ilyen alkalmazásokat legegyszerűbben a bináris fákon tudjuk szemléltetni. Vezessünk be két ilyen adatstruktúrát. A ponthoz rendelt, összehasonlítható értékeket röviden a pont értékének nevezzük.

Bináris keresőfának nevezünk egy olyan bináris rendezett fát, amelyben minden pontra igaz, hogy a ponthoz, mint gyökérponthoz tartozó bal részfa pontértékei *nem nagyobbak,* a *jobb részfa* pontértékei *nem kisebbek* a pont értékénél (59. ábra).



59. ábra. Bináris keresőfa

Bináris kupacnak vagy röviden *kupacnak* nevezünk egy olyan, majdnem teljes bináris fát, amelyben minden pontra igaz, hogy a ponthoz, mint gyökérponthoz tartozó részfa pontértékei *nem kisebbek* a pont értékénél (60. ábra). Megjegyezzük, hogy ez a definíció a növekvő rendezési iránynak felel meg, így a fa gyökérpontjának értéke minimális a fában. Ugyanígy definiálhatjuk a másik rendezési iránynak megfelelő kupacot, ha azt kötjük ki, hogy a részfa pontértékei *nem nagyobbak* a pont értékénél. Ez esetben a fa gyökérpontjának értéke maximális a fában. (A bináris keresőfánál ilyen kettősség nincs, ugyanaz a fa mind a két rendezési irányhoz jó, ugyan úgy, mint pl. a rendezett tömb vagy lista.)



60. ábra. Bináris kupac

9.4.2. Implementáció

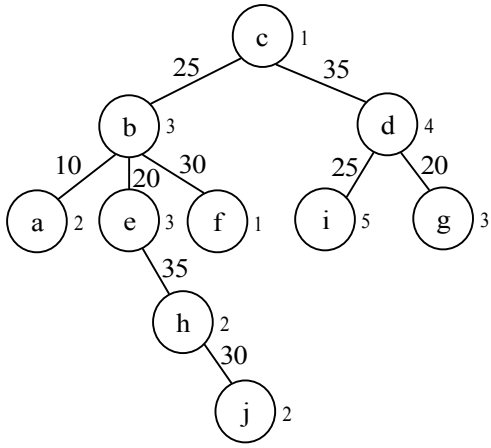
A pont és éljellemzőkkel felszerelt fák is hálózatok, tehát elvben alkalmazhatnánk a fentebb tárgyalt általános módszereket, viszont a fa tulajdonság kihasználása lényegesen tömörebb és az algoritmusokkal könnyebben kezelhető implementációkat is lehetővé tesz.

Minden gyökeres fát ábrázolhatunk *címketömbbel*. Egy pont címkéjén az őt a fában megelőző pont azonosítóját értjük. A gyökérpontra ez nem értelmezett, de minden más pontnak van egyértelmű címkéje. Mivel a címke egyértelműen a pontokhoz rendeli az éleket is, mind a kapcsolatokat, mind az él, mind a pont-

jellemzőket megadhatjuk ugyanebben a rendszerben (3. táblázat, 61. ábra). A táblázatban a `P` a pontokat, a `C` a címkéket jelöli, a `PT` egy pontjellemző (az ábrán a pont mellett kisebb számmal), az `ElH` egy éljellemző (az ábrán az él mellett nagyobb számmal).

P	**a**	**b**	**c**	**d**	**e**	**f**	**g**	**h**	**i**	**j**
C	**b**	**c**	**-**	**c**	**b**	**b**	**d**	**e**	**d**	**h**
PT	2	3	1	4	3	1	3	2	5	2
ElH	10	25	10	35	20	30	20	35	25	30

3. táblázat. Címketömb



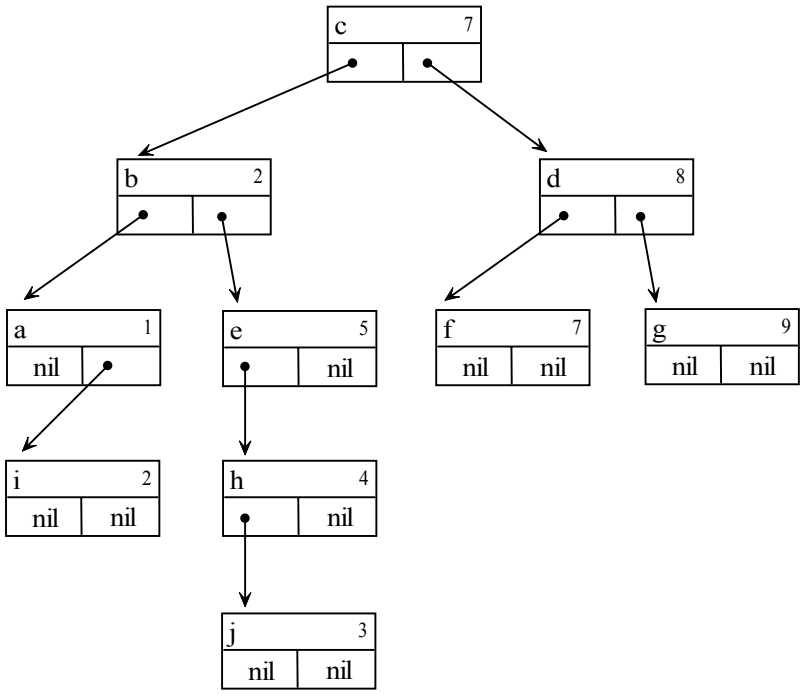
61. ábra. A címketömbbel megadott fa

A bináris fáknál jól alkalmazhatók a dinamikus adatszerkezetek is. Pontonként egy rekordot veszünk fel, ebben tároljuk a pontjellemzőket, valamint az élenként az éljellemzőt és a megfelelő követőre mutató mutatót. A nem létező követőket a mutató `nil` értéke jelzi. (62. ábra) Bár ez a dinamikus adatszerkezet egyértelműen leírja a fát, esetleg bővíthető lenne még a szülőre mutató mutatóval is.

A bináris fa deklarációs sémája (csak pontjellemzővel):

```
type
  TFaPontAzon=pontazonosító adattípus;
  TFaPontJell=pontjellemző adattípus;
```

```
PFaPont=^TFaPont;
TFaPont=record
  Azon: TFaPontAzon;
  PontJell: TFaPontJell;
  BalAg: PFaPont;
  JobbAg: PFaPont;
end;
```



62. ábra. Dinamikus adatszerkezettel megadott fa

Az előző példa a `PT` pontjellemzőre nézve egyben bináris keresőfa is. A keresőfában egy érték megkeresésének módja a fa definíciójából közvetlenül adódik:

- Elindulunk a gyökérponttól.
- Minden érintett pontra elvégezzük az alábbi vizsgálatot, illetve elágazást:
  - Ha a pontjellemző egyenlő a keresett értékkel, készen vagyunk
  - Ha a pontjellemző kisebb a keresett értéknél, balra lépünk, a következő pont a bal leszármazott lesz.
  - Ha a pontjellemző nagyobb a keresett értéknél, jobbra lépünk, a következő pont a jobb leszármazott lesz.

Az eljárás nagyon hasonlít a tömbben való bináris kereséshez, hiszen itt is minden lépésben kizárjuk az adathalmaz egyik részét (de nem biztosan a felét, mint a bináris keresésnél). Minden lépésben egy szinttel lejjebb kerülünk, tehát a keresés annál gyorsabb, minél kisebb a fa magassága. Szemlélet alapján is adódik, hogy a magasság akkor lesz minimális, ha a fa *kiegyensúlyozott,* vagyis minden pontjára, mint gyökérpontra igaz, hogy a bal részfa és a jobb részfa pontjainak száma közti eltérés a minimális, vagyis legfeljebb `1`. Ez esetben a keresés ugyanolyan gyors, mint a bináris keresés. Ha a fa ehhez képest nagyon „torz", akkor a keresés akár közel lineárissá is fajulhat. A kiegyensúlyozottság pontos fogalmával és az ehhez kapcsolódó további érdekes és fontos algoritmusokkal (pl. fa kiegyensúlyozása) kapcsolatban a szakirodalomra utalunk [8, 1].

Sok feladat csak úgy oldható meg, hogy a fa pontjait (az összes pontot, vagy valamilyen feltétel teljesüléséig minden pontot) valamilyen rendszer szerint egyenként meg kell vizsgálni. Ezt másképpen úgy mondjuk, hogy a fát be kell járni. A fabejárási algoritmusok tipikusan *rekurzív* algoritmusok. Sokféle *fabejárás* lehetséges, néhány nevezetes alapmódszer:

- *Inorder* bejárás: a gyökérpontot a két részfa között érintjük. A megfelelő sematikus rekurzív eljárás, arra az esetre, ha balról jobbra haladunk, vagyis a bal részfát vesszük előre:

```
procedure Inorder(X: PFaPont);
{X az aktuális részfa gyökérpontja}
begin
  if X<>nil then begin
    Inorder(X bal leszármazottja);
    X vizsgálata
    Inorder(X jobb leszármazottja);
  end;
end;
```

- *Preorder* bejárás: a gyökérpontot a részfák előtt érintjük.
- *Posztorder* bejárás: a gyökérpontot a részfák után érintjük.

Megjegyezzük, hogy a feladattól függően akár az azonosító, akár a pontjellemző elhagyható. Például ha egy bináris keresőfát csak adatok rendezésére és keresésére akarunk használni, akkor az azonosító felesleges.

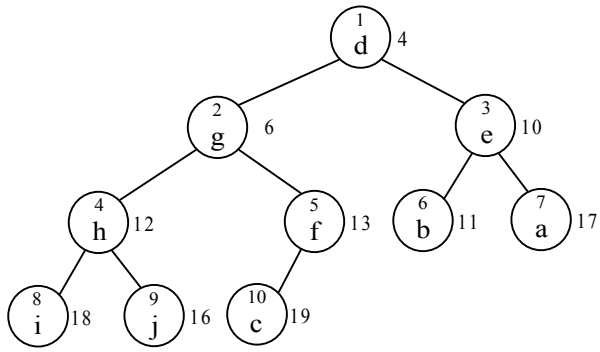
A *kupac* adatstruktúra, mint speciális bináris fa, szabályosságánál fogva lehetővé tesz egy másfajta, bizonyos feladatoknál nagyon jó hatékonyságú töm-

bös implementációt is. Ennél *csak pontjellemzőket* kezelünk és a pontadatokat az egydimenziós tömbben a következő módon tároljuk (4. táblázat, 71. ábra):

- A gyökérpont indexe `1` lesz.
- Ha egy pont indexe `I`, akkor a bal gyermeke a `2 * I`, a jobb gyermeke a `2 * I + 1` index alatt tárolódik.

Index	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**	**10**
Pontazonosító	**d**	**g**	**e**	**h**	**f**	**b**	**a**	**i**	**j**	**c**
Pontjellemző	4	6	10	12	13	11	17	18	16	19

4. táblázat. Kupactömb



63. ábra. Kupac

Közvetlenül látható, hogy a tömb első eleme mindig minimális a pontértékek között. Mind a beszúrás, mind a törlés viszonylag gyorsan végrehajtható. (Pontosabban: ha a pontok száma `N`, akkor a fa szintjeinek száma a `Log2(N) + 1` egész része, és mind a két alapművelet ezzel arányos számú elemi lépéssel végrehajtható.)

Ez az adatstruktúra kifejezetten előnyös olyan feladatoknál, amelyeknél a kezelendő adathalmaz sűrűn változik, viszont a *lekérdezés csak a minimális* (vagy a maximális) értékre vonatkozik. Ha összevetjük egy rendezett tömbbel vagy listával, láthatjuk, hogy a minimális (maximális) érték ezeknél is mindig közvetlenül rendelkezésre áll, de a beszúrás és a törlés műveletigénye lényegesen nagyobb (magával az elemszámmal, nem pedig ennek logaritmusával arányos). Viszont azt is észrevehetjük, hogy egy tetszőleges érték keresésénél már elveszik a kupac előnye a rendezett tömbbel szemben, hiszen bináris keresésre nem alkalmas, a belső elemeket a kupacban csak sorosan lehet keresni.

A fák kezelését két feladatcsoporttal szemléltetjük. Az egyik a dinamikus adatszerkezettel megvalósított *bináris fára, a másik a tömbben tárolt kupacra vonatkozik*.

9.4.3. Alapfeladatok

9.4.3.1. Bináris fa

A bináris fa adatszerkezete (`uGrafD`):

```
type
  TBinFaAzon=String;
  TBinFaAdat=String;
  PBinFaPont=^TBinFaPont;
  TBinFaPont=record  {pont}
    Azon: TBinFaAzon;  {pontazonosító}
    Adat: TBinFaAdat;  {pontjellemző}
    BalAg, JobbAg: PBinFaPont;  {az elágazások mutatói}
  end;
```

Tehát a példában mind a pontazonosító, mind a pontjellemző tetszőleges string lehet. Az azonosítót egyedinek tekintjük a fában, viszont nem zárjuk ki, hogy ugyanaz az `Adat` érték több pontban is előforduljon. Feltételezzük, hogy az `Adat`-ra nézve a fa egyben bináris keresőfa is.

A programnyelvi megvalósításban a bővítési műveletnél, a fejlesztőrendszerek eltéréséből adódó problémát a korábbiakhoz hasonló módon oldjuk meg. A megfelelő szubrutinok: `tGrafU.UjBinFaPont`, `dGrafU.UjBinFaPont`. A fát, mint paramétert a gyökérpontjával adjuk át, az üres fát ennek `nil` értéke jelenti (lásd `uGBinFa.BinFaInit`).

9.4.3.1.1. mintafeladat: Keressünk meg egy adott pontjellemző értékkel rendelkező pontot!

Útmutató ♦ A keresést a fentebb vázolt balra/jobbra lépegetéssel valósítjuk meg. Az eredmény (a lépegetés sorrendjében) első ilyen jellemzőjű pont mutatója, vagy `nil`, ha nincs ilyen pont. Jól hasznosítható mellékeredményként előállítjuk még a keresett pont szülőjének mutatóját, és azt az információt, hogy a szülő melyik ágán van a keresett pont.

Adatszerkezet (79)

Azonosító	Funkció	Típus	Jelleg
`Gyoker`	a fa gyökérpontja	`PBinFaPont`	input
`Mit`	a keresett érték	`TBinFaAdat`	input
`Szulo`	a keresett pont elődje vagy `nil`	`PBinFaPont`	output
`SzuloBalAg`	a keresett a szülő bal ágán van-e	`boolean`	output
`BinFanAdatKeres`	a keresett pont vagy `nil`	`PBinFaPont`	output
`Akt`	aktuális mutató a fán lépkedésnél	`PBinFaPont`	munka

Szubrutin: `uGBinFa.BinFanAdatKeres`.

9.4.3.1.2. mintafeladat: Keressünk meg egy adott azonosítójú pontot!

Útmutató ♦ Az azonosító szerint a fa nem feltétlenül keresőfa, ezért az előző feladat módszere nem alkalmazható, be kell járnunk a fát. A bejárásra egy pre-order rekurzív algoritmust adunk.

Adatszerkezet (80)

Azonosító	Funkció	Típus	Jelleg
`Gyoker`	a fa gyökérpontja	`PbinFaPont`	input
`Mit`	a keresett azonosító	`TbinFaAzon`	input
`BinFanAzonKeres`	a keresett pont vagy `nil`	`PbinFaPont`	output
`Akt`	segédmutató	`PBinFaPont`	munka

Szubrutin: `uGBinFa.BinFanAzonKeres`.

9.4.3.1.3. mintafeladat: Bővítsük a fát egy új ponttal.

Útmutató ♦ A funkció csak akkor hajtható végre, ha az adott azonosító új a fában, ezt előzetesen ellenőrizzük. Az új pont jellemzője szerint balra/jobbra lépkedve keressük meg az első szabad ágat (`nil` mutatót) ahova a pont bekötheő, így a struktúra továbbra is bináris keresőfa marad.

Adatszerkezet (81)

Azonosító	Funkció	Típus	Jelleg
`Gyoker`	a fa gyökérpontja	`PBinFaPont`	input, output
`A`	az új pont azonosítója	`TBinFaAzon`	input
`Mit`	az új pont jellemzője	`TBinFaAdat`	input
`BinFara`	az új pont mutatója vagy `nil`	`PBinFaPont`	output
`UjPont`	az új pont mutatója vagy `nil`	`PBinFaPont`	munka
`Akt,Szulo`	segédmutató a helykereséshez	`PBinFaPont`	munka
`UjAzon`	az azonosító új-e	`Boolean`	munka

Szubrutin: `uGBinFa.BinFaRa`.

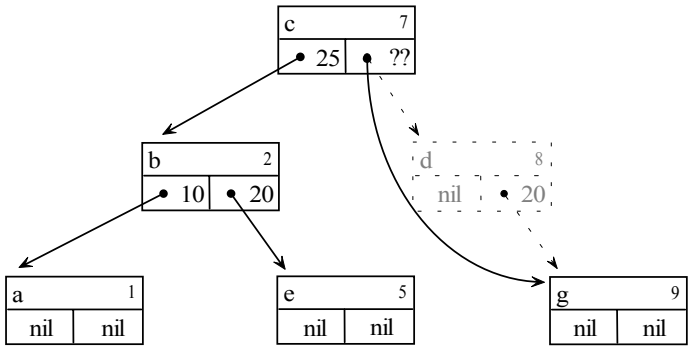
Megjegyzés ♦ A funkció eredménytelenségét (eredmény = `nil`) az adott azonosító ismételt előfordulása, vagy a pont létrehozásához szükséges hely hiánya okozhatja.

9.4.3.1.4. mintafeladat: Töröljünk egy adott jellemzőjű pontot a fáról!

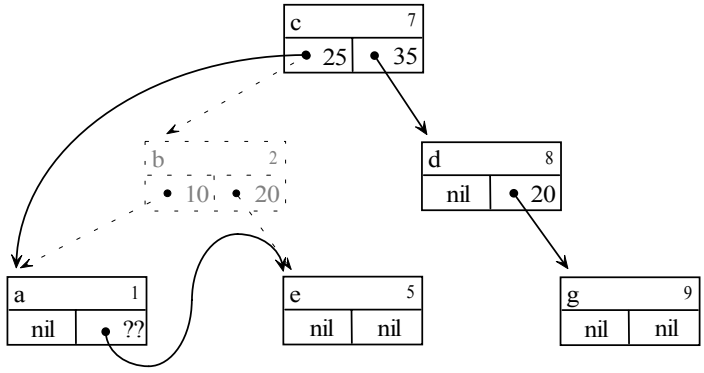
Útmutató ♦ Az első feladat egy ilyen jellemzőjű pont keresése, ezt a `BinFanAdatKeres` végzi, megadva a szülőt és az ágat is. Ha van ilyen pont, akkor a pont helyzete szerint tagoljuk a struktúrából való kikapcsolást. A struktúrának meg kell őriznie a bináris keresőfa tulajdonságot:

- A legegyszerűbb eset, ha a pont levél, ekkor egyszerűen lekapcsoljuk a szülő megfelelő ágáról.
- Ha a törlendő pontnak csak egy leszármazottja van, akkor a leszármazottat kapcsoljuk be a törlendő helyére (64. ábra).
- Ha a törlendő pontnak két leszármazottja van, akkor kikapcsolásával két ág válik szabaddá. Először ezt a két ágat egyesítjük, úgy, hogy a jobb ágat bekötjük (szükségszerűen szintén jobb ágként) a bal ág „jobbszélén” lefelé haladva talált első szabad helyre, majd az így létrejött részfát kötjük be a törlendő pont helyére (65. ábra).

Végül természetesen megszüntetjük a törlendő pontot.



64. ábra. Törlés egy leszármazott esetén



65. ábra. Törlés két leszármazott esetén

Adatszerkezet (82)

Azonosító	Funkció	Típus	Jelleg
`Gyoker`	a fa gyökérpontja	`PBinFaPont`	input, output
`Mit`	a törlendő adat	`TBinFaAdat`	input
`BinFaRol`	történt-e törlés	`PBinFaPont`	output
`Akt`	a törlendő pont	`PBinFaPont`	munka
`Szulo, Munka`	segédmutatók a keresésekhez	`PBinFaPont`	munka
`Volt`	történt-e törlés	`Boolean`	munka
`SzuloBalAg`	a törlendő pont elhelyezkedése	`Boolean`	munka

Szubrutin: `uGBinFa.BinFaRol`.

Megjegyzés ♦ A törlés más algoritmussal is elvégezhető, minden olyan algoritmus helyes, amely megtartja a struktúra bináris keresőfa jellegét. Természetesen, további szempontokat figyelembe véve rangsorolhatjuk is ezeket az algoritmusokat.

9.4.3.1.5. mintafeladat: Írjuk ki a pontjellemzőket növekvő rendezettségben egy szövegfájlba!

Útmutató ♦ A bináris keresőfa definíciójából következően ezt a sorrendet egy balról jobbra haladó inorder bejárással kapjuk.

Adatszerkezet (83)

Azonosító	Funkció	Típus	Jelleg
`Gyoker`	a fa gyökérpontja	`PBinFaPont`	input
`Lista`	a szövegfájl	`Text`	output

Szubrutin: `uGBinFa.BinFaListaNo`.

9.4.3.1.6. mintafeladat: Írjuk ki a pontjellemzőket csökkenő rendezettségben egy szövegfájlba!

Útmutató ♦ A bináris keresőfa definíciójából következően ezt a sorrendet egy jobbról balra haladó inorder bejárással kapjuk.

Adatszerkezet (84)

Azonosító	Funkció	Típus	Jelleg
`Gyoker`	a fa gyökérpontja	`PBinFaPont`	input
`Lista`	a szövegfájl	`Text`	output

Szubrutin: `uGBinFa.BinFaListaCsokken`.

9.4.3.1.7. mintafeladat: Töröljük a fa minden pontját!

Útmutató ♦ Töröljük a gyökérpontot (`BinFaRol`) mindaddig, amíg van pont a fán.

Szubrutin: `uGBinFa.BinFaTorol`.

Megjegyzés ♦ Megjegyezzük, hogy ez a megoldás nem optimális, olyan értelemben, hogy futás közben sok felesleges adminisztrációt végez (lásd 9.6. pont 5. feladat hatodik részfeladata).

9.4.3.2. Bináris kupac

Az itt bemutatandó kupackezelő mintamegoldásainkat fel fogjuk használni a későbbiekben a teljes hálózaton dolgozó összetett feladatoknál is. Ezekhez alkalmazkodva a kupacban pontazonosítóként a pontnak a hálózatbeli sorszámát (pontindexét) használjuk. A kupacban kezelt pontjellemző a pontnak egy rögzített ponttól számított *távolsága* lesz. A távolságot itt a hálózatban értelmezzük, tehát két pont távolsága a hálózatban valamely, – a két pont közti – gráfbeli útnak a hossza. A megfelelő deklarációk (`uGrafD`):

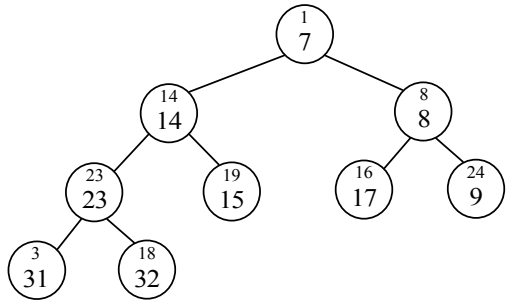
```
const
  GMaxUtHossz=GMaxElDb*GMaxElHossz;  {maximális úthossz
    körmentes utakra}
  GVegtelen=GmaxUtHossz+1;  {technikailag szükséges adat}
type
  TGUtHossz=0..GVegtelen;  {úthossz}
  TGPontITmb=array[TGPontI] of TGPontI;  {pontindextömb}
  TGTavTmb=array[TGPontI] of TGUtHossz;  {távolságtömb}
```

Index	1	2	3	4	5	6	7	8	9
Pontindex	1	14	8	23	19	16	24	3	18

Pontindex	**1**	2	**3**	4	5	6	7	**8**	9	10	11	12	13
Távolság	7	13	31	21	17	29	1	8	18	16	42	25	32

Pontindex	**14**	15	**16**	17	**18**	**19**	20	21	22	**23**	**24**	25
Távolság	14	12	17	11	32	15	4	10	8	23	9	3

Kupac aktuális elemszáma	9
Gráf pontszáma	25



66. ábra. A gráf egyes pontjait tartalmazó kupac

A kupacban a gráf bizonyos pontjai lesznek, úgy, hogy a távolságadatukra, mint pontjellemzőre nézve teljesüljön a kupactulajdonság.

Ennek megfelelően egy kupac adatstruktúrát alkot (66. ábra):

- Egy `TGPontITmb` típusú tömb, ebben vannak a gráf bizonyos pontjai, mint pontindexek, egy pont legfeljebb 1 példányban.
- Egy `TGPontDb` típusú adat, a kupac (vagyis az előző tömb) aktuális elemszáma.
- Egy `TGTavTmb` típusú tömb. Ez a tömb a gráf összes pontjához tartalmaz egy, a pontindex által kijelölt távolságadatot, de a kupachoz csak azokat használjuk fel, amelyek indexei bent vannak a kupacban.

9.4.3.2.1. mintafeladat: Keressünk meg egy pont kupacbeli indexét!

Útmutató ♦ A keresést csak sorosan lehet végezni. Az eredmény a pontnak a kupactömbbeli indexe, vagy 0, ha a pont nincs bent a kupacban.

Adatszerkezet (85)

Azonosító	Funkció	Típus	Jelleg
`Y`	a keresendő pont	`TGPontI`	input
`N`	a kupac aktuális elemszáma	`TGPontDb`	input
`A`	a kupactömb	`TGPontITmb`	input
`KupacIndex`	a keresett pont kupacbeli indexe, vagy 0	`TGPontI0`	output
`I`	a kupactömb indexe	`TGPontI1`	munka

Szubrutin: `uGKupac.KupacIndex`.

9.4.3.2.2. mintafeladat: Vegyünk fel egy új pontot a kupacba!

Útmutató ♦ Feltételezzük (nem ellenőrizzük), hogy a pont még nincs a kupacban. Az új pont a bináris fa új leveleként, vagyis a tömb utolsó elemeként jelenik meg. Ezután a fában addig visszük felfelé (cserével) az új pontot, amíg (a hozzátartozó érték szerint) nem teljesül a kupactulajdonság. Vegyük észre, hogy a kupacban a gyermekponttól a szülőpont felé haladva az index feleződik.

Adatszerkezet (86)

Azonosító	Funkció	Típus	Jelleg
`Y`	a felveendő pont	`TGPontI`	input
`N`	a kupac aktuális elemszáma	`TGPontDb`	input, output
`A`	a kupactömb	`TGPontITmb`	input, output
`T`	a pontjellemző tömb	`TGtavTmb`	input
`I`	a kupactömb indexe	`TGPontI`	munka
`J`	a kupactömb indexe	`TGPontI0`	munka
`Vege`	a kupactulajdonság teljesülése	`Boolean`	munka

Szubrutin: `uGKupac.KupacBa`.

9.4.3.2.3. mintafeladat: Vegyük ki a minimális értékű pontot a kupacból!

Útmutató ♦ Ez a kupac gyökérpontjának eltávolítását jelenti. Ezt úgy hajtjuk végre, hogy az utolsó elemmel (utolsó levéllel) felülírjuk a gyökérpontot, majd helyreállítjuk a kupacot. A helyreállításhoz az új gyökérpontot (mivel ennek értéke csak nagyobb vagy egyenlő lehet, mint gyerekeinek értéke) cserékkel „lefele” visszük, „belesüllyesztjük” a kupacba. A szülőpontot, ha szükséges, mindig a kisebb értékű gyerekével cseréljük fel, így áll helyre a kupactulajdonság.

Adatszerkezet (87)

Azonosító	Funkció	Típus	Jelleg
`N`	a kupac aktuális elemszáma	`TGPontDb`	input, output
`A`	a kupactömb	`TGPontITmb`	input, output
`T`	a pontjellemző tömb	`TGtavTmb`	input
`I1`	aktuális szülő index (cseréhez)	`TGPontI`	munka
`I2b`	aktuális bal gyerek index	`TGPontI`	munka
`I2j`	aktuális jobb gyerek index	`TGPontI`	munka
`I2`	aktuális gyerek index (cseréhez)	`TGPontI`	munka
`I`	segédváltozó cseréhez	`TGPontI`	munka

Szubrutin: `uGKupac.KupacBol`.

9.4.3.2.4. mintafeladat: Állítsuk helyre a kupacot egy pont értékének csökkenése után (átsorolás)!

Útmutató ♦ A megoldás nagyon hasonló az új pont felvételéhez. A fában addig visszük felfelé (cserével) a csökkent értékű pontot, amíg nem teljesül a kupactulajdonság.

Adatszerkezet (88)

Azonosító	Funkció	Típus	Jelleg
`Y`	a csökkent értékű pont	`TGPontI`	input
`N`	a kupac aktuális elemszáma	`TGPontDb`	input
`A`	a kupactömb	`TGPontITmb`	input, output
`T`	a pontjellemző tömb	`TGtavTmb`	input
`I`	a kupactömb indexe	`TGPontI`	munka
`J`	a kupactömb indexe	`TGPontI0`	munka
`X`	munkaváltozó a cseréhez	`TGPontI`	munka
`Vege`	a kupactulajdonság teljesülése	`Boolean`	munka

Szubrutin: `uGKupac.KupacFel`.

9.4.3.2.5. mintafeladat: Rendezzük pontok egy tömbjét növekvő távolság szerint!

Útmutató ♦ A pontokat felvesszük a kupacba (`KupacBa`), majd sorra kiveszszük a minimális értékű elemet (`KupacBol`).

Adatszerkezet (89)

Azonosító	Funkció	Típus	Jelleg
`V`	a rendezendő tömb	`TGPontITmb`	input, output
`m`	a rendezendő tömb elemszáma	`TGPontDb`	input
`T`	a pontjellemző tömb	`TGtavTmb`	input
`A`	a kupactömb	`TGPontITmb`	munka
`N`	a kupac aktuális elemszáma	`TGPontDb`	munka
`I`	a rendezendő tömb indexe	`TGPontI0`	munka

Szubrutin: `uGKupac.KupacRend`.

Megjegyzés ♦ A feladat ugyanezzel a módszerrel helyben (csak 1 tömb alkalmazásával) is megoldható. Itt csak a könnyebb érthetőség kedvéért alkalmaztunk egy külön munkatömböt a kupac számára.

9.5. Összetett feladatok

9.5.1. Útkeresés

9.5.1.1. Bevezetés

A gyakorlati alkalmazásokban az egyik legfontosabb, legtöbbször használt hálózati algoritmus a bizonyos szempontokból legkedvezőbb *útvonalak* keresése. A „legkedvezőbb útvonal" fogalmat természetesen pontosítani kell ahhoz, hogy algoritmusokat adhassunk a meghatározására. E célból vezessünk be néhány eddig még pontosan nem definiált új fogalmat.

Mint a bevezetőben már definiáltuk, a természetes szemlélettel megegyezően út a gráf egy olyan pont illetve élsorozata, amelynek a felsorolás (bejárás) sorrendjében szomszédos pontjai közt van a megelőző pontból a következő pontba mutató él. A *körös út* olyan út, amelyben van legalább egy olyan pont, amely ismétlődik. *Körmentes út* az, amelyben ilyen pont nincs. Nevezzük el az út első pontját *kiindulópontnak,* utolsó pontját pedig *célpontnak*. Egy kiinduló pont–célpont párt egy *viszonylatnak* nevezünk. Egy gráfban általában több út is van egy adott viszonylatban (minél nagyobb a gráf, általában annál több a lehetséges utak száma).

Ha már hálózatról beszélünk, vagyis az élekhez értelmeztünk éljellemző értéket, akkor az utakat is mérhetjük. Az él egyik (vagy esetleg egyetlen) mérőszámát az egyszerűség kedvéért általában *élhossznak* szoktuk nevezni, így az utat *élei hosszának összegével* mérjük, és ezt *úthossznak* nevezzük. (Az alkalmazásokban ez a „hossz" természetesen sok minden lehet, pl. idő, költség, munkaigény stb.). Így már beszélhetünk két út hossz szerinti összehasonlításáról, *rövidebb, hosszabb* utakról. Közlekedési analógiával élve, itt számunkra a *„kedvezőbb"* a kisebb hosszúságút, a rövidebbet jelenti, a legkedvezőbb a *minimális* hosszúságút. (Megjegyezzük, hogy ilyen, minimális hosszú útból egy adott viszonylatban több is lehet.)

Bár elvben egy élhez rendelt szám negatív vagy nulla is lehet, de ettől az esettől itt tekintsünk el, legyenek az éleink, így az utjaink is mind pozitív hosszúak. Ebből rögtön következik az, hogy amikor minimális hosszúságú utat keresünk, akkor biztosan körmentes utat kapunk, hiszen egy kör levágása csökkenti a hosszt.

A minimális hosszúságú út (rövidebben kifejezve a *minimális út*) keresése több szempontból is érdekes problémakör. Jól demonstrálja például azt, hogy a keresési, optimalizálási problémák „naiv" megközelítése sokszor nem vezet célhoz, sőt gyakorlati méretekben egyszerűen használhatatlan. Itt egy ilyen jellegű módszer lenne az, hogy: „egy adott viszonylatban vizsgáljuk meg az összes körmentes utat, számítsuk ki mindegyiknek a hosszát és így megkapjuk a keresett minimális hosszút". Ez esetleg menne igen kis hálózatokra, de gondoljuk meg, hogy már egy 15 pontos hálózatnál is több milliárd esetet kellene vizsgálni. (A kezdő és végpontot rögzítve, a többi pontok az összes lehetséges részhalmazát és az egyes részhalmazbeli pontok összes sorrendjét kellene képezni, mindegyikről eldönteni, hogy út-e vagy sem, és ha út, akkor mennyi a hossza). Nagyobb hálózatoknál ez még az elképzelhető leggyorsabb számítógépekkel is lehetetlen lenne.

A feladat egy másik jellegzetessége, hogy jól mutatja az *adatstruktúra* és az *algoritmus* erős *összefüggését.* Mint látni fogjuk egészen más jellegű algoritmus vezet célhoz a hálózat mátrixos tárolása esetén, mint az éltárolási módszernél. Az a számítási, keresési mód, ami az egyiknél nagyon hatékony, a másiknál gyakorlatilag kivitelezhetetlen. Példaképpen nézzünk két, a témakörben klasszikusnak számító és jól ismert konkrét eljárást.

9.5.1.2. Mátrix módszer

Az alább ismertetendő eljárást a szakirodalom – első publikálójáról – Warshall féle eljárásnak nevezi. Akkor alkalmazzuk, ha minden viszonylatban (vagy legalábbis a viszonylatok nagy részében) meg akarjuk határozni a minimális utat, és van elegendő operatív tárterületünk az algoritmus által igényelt két darab *pontszám*pontszám* méretű mátrix tárolására.

Az algoritmus könnyebb leírásához vezessünk be egy újabb fogalmat: Az, hogy egy `x-y` *viszonylat* minimális útját egy `w` pont *bevonásával* keressük, azt jelenti, hogy az utat egy `x-w` kezdő és egy `w-y` befejező részútból próbáljuk összerakni (vagyis az `x`-ből az `y`-ba a `w`-n keresztül megyünk).

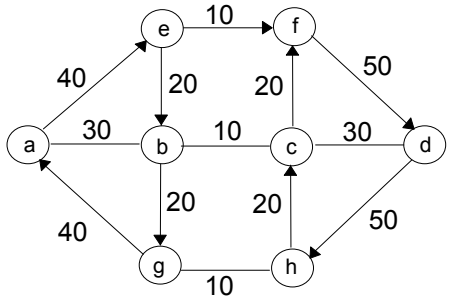
Az algoritmus egy `TT` távolságmátrixot és egy `CC` címkemátrixot használ. Mindkettő *pontszám*pontszám* méretű és soronként és oszloponként is a pontokkal (vagy a pontindexekkel) van indexelve (úgy mint a hálózat mátrixos tárolásánál). A sorindex egy viszonylat kezdőpontjának, az oszlopindex egy viszonylat végpontjának felel meg.

A `TT` elemei a viszonylatok *aktuális távolságát* vagyis az aktuális minimális útjának *hosszát* tartalmazzák, tehát a `TT[x, y]` az `x-y` viszonylat ilyen távolsá-

ga. A CC elemei *pontok* (vagy pontindexek) és a minimális út összerakásához, tehát a megfelelő pontsorozat előállításához szükséges adatokat tartalmazzák abban a formában, hogy a CC[x, y] az a pont, amely az x–y viszonylat aktuális minimális útján az x kezdőpont után jön (merre induljunk a kezdőpontból a végpont felé).

Az algoritmus a TT és a CC egy kezdőállapotából kiindulva, a mátrixokat *lépésenként javítva,* több lépés megtétele után jut el a végeredményhez. Mint látni fogjuk, pontosan annyi lépés kell, ahány pont van a hálózatban. Ezek után az algoritmust a következőképpen definiálhatjuk:

- *Kezdőállapot*
  TT: ha a kezdő és végpont között van közvetlen összeköttetés (él), akkor ennek hossza a távolság, egyébként a távolság végtelen nagy. (Mint látható, ez tulajdonképpen a hálózat mátrixos tárolásának megfelelő mátrix.)
  CC: a viszonylat címke eleme a végpont. Minden oszlop az oszlopindexet tartalmazza, annak megfelelően, hogy a TT kezdőállapota tulajdonképpen egy élből álló utakat jelent
- *Javító lépések*
  A hálózat minden w pontjára (egyszer) és ezen belül minden x–y viszonylatra (egyszer) végrehajtandó: kíséreljük meg a w-t *bevonni* a viszonylatba. Ez azt jeleni, hogy ha TT[x, y] > TT[x, w] + TT[w, y] (vagyis a w bevonásával rövidítünk), akkor legyen az új távolság TT[x, y] = TT[x, w] + TT[w, y] és az új címke CC[x, y] = CC[x, w]. (A címke azt jelenti, hogy az y-hoz vezető minimális úton ugyanúgy kell indulni, mint a w-hez vezető minimális úton).
- *Végállapot*
  A TT végállapota megadja a *tényleges minimális út* hosszát minden viszonylathoz. Ennél rövidebb út nem található. (Ez persze azt nem zárja ki, hogy esetleg több út is legyen ezzel a minimális hosszal.) Ha van olyan pont a hálózatban, amelyből vagy amelybe egyáltalán nincs út, tehát a pont valamelyik vagy mindkét értelemben *elszigetelt* (izolált), akkor a megfelelő mátrixelemek végtelen értékűek maradnak. Ha magukra az *útvonalakra* is szükségünk van, ezeket viszonylatonként kiolvashatjuk a CC-ből, ennek definíciója szerint.



67. ábra. Példahálózat

Az algoritmushoz a 67. ábrán adunk példaadatokat. Az ábrán az irányítás nélküli szakaszok kétirányú, és mindkét irányban azonos hosszú éleket jelölnek. Az 5. táblázat a kezdőállapot, a 6. táblázat az első javítás (`w = a`), a 7. táblázat a második javítás (`w = b`) utáni állapot, a 8. táblázat a végállapot. Például határozzuk meg a példahálózatban az `f–g` viszonylatra az utat, a végállapotból: `CC[f, g] = d`, `CC[d, g] = c`, `CC[c, g] = b`, `CC[b, g] = g`, tehát az út: `f–d–c–b–g`.

TT	a	b	c	d	e	f	g	h		CC	a	b	c	d	e	f	g	h
a	0	30	~	~	40	~	~	~		**a**	a	b	c	d	e	f	g	h
b	30	0	10	~	~	~	20	~		**b**	a	b	c	d	e	f	g	h
c	~	10	0	30	~	20	~	~		**c**	a	b	c	d	e	f	g	h
d	~	~	30	0	~	~	~	50		**d**	a	b	c	d	e	f	g	h
e	~	20	~	~	0	10	~	~		**e**	a	b	c	d	e	f	g	h
f	~	~	~	50	~	0	~	~		**f**	a	b	c	d	e	f	g	h
g	40	~	~	~	~	~	0	10		**g**	a	b	c	d	e	f	g	h
h	~	~	20	~	~	~	10	0		**h**	a	b	c	d	e	f	g	h

5. táblázat. Kezdőállapot

TT	a	b	c	d	e	f	g	h		CC	a	b	c	d	e	f	g	h
a	0	30	~	~	40	~	~	~		**a**	a	b	c	d	e	f	g	h
b	30	0	10	~	70	~	20	~		**b**	a	b	c	d	a	f	g	h
c	~	10	0	30	~	20	~	~		**c**	a	b	c	d	e	f	g	h
d	~	~	30	0	~	~	~	50		**d**	a	b	c	d	e	f	g	h
e	~	20	~	~	0	10	~	~		**e**	a	b	c	d	e	f	g	h
f	~	~	~	50	~	0	~	~		**f**	a	b	c	d	e	f	g	h
g	40	70	~	~	80	~	0	10		**g**	a	a	c	d	a	f	g	h
h	~	~	20	~	~	~	10	0		**h**	a	b	c	d	e	f	g	h

6. táblázat. Első javítás

TT	a	b	c	d	e	f	g	h		CC	a	b	c	d	e	f	g	h
a	0	30	40	~	40	~	50	~		**a**	a	b	b	d	e	f	b	h
b	30	0	10	~	70	~	20	~		**b**	a	b	c	d	a	f	g	h
c	40	10	0	30	80	20	30	~		**c**	b	b	c	d	b	f	b	h
d	~	~	30	0	~	~	~	50		**d**	a	b	c	d	e	f	g	h
e	50	20	30	~	0	10	40	~		**e**	b	b	b	d	e	f	b	h
f	~	~	~	50	~	0	~	~		**f**	a	b	c	d	e	f	g	h
g	40	70	80	~	80	~	0	10		**g**	a	a	a	d	a	f	g	h
h	~	~	20	~	~	~	10	0		**h**	a	b	c	d	e	f	g	h

7. táblázat. Második javítás

TT	a	b	c	d	e	f	g	h		CC	a	b	c	d	e	f	g	h
a	0	30	40	70	40	50	50	60		**a**	a	b	b	b	e	e	b	b
b	30	0	10	40	70	30	20	30		**b**	a	b	c	c	a	c	g	g
c	40	10	0	30	80	20	30	40		**c**	b	b	c	d	b	f	b	b
d	70	40	30	0	110	50	60	50		**d**	c	c	c	d	c	c	c	h
e	50	20	30	60	0	10	40	50		**e**	b	b	b	b	e	f	b	b
f	120	90	80	50	160	0	110	100		**f**	d	d	d	d	d	f	d	d
g	40	40	30	60	80	50	0	10		**g**	a	h	h	h	a	h	g	h
h	50	30	20	50	90	40	10	0		**h**	g	c	c	c	g	c	g	h

8. táblázat. Végállapot

9.5.1.2.1 mintafeladat: Határozzuk meg minden viszonylatban a minimális utat a Warshall módszerrel!

Útmutató ♦ A módszer alapvetően támaszkodik a mátrixtárolásra. A mátrixokban pontindexeket használunk (`uGrafD.TGrafMtx`). A javító lépéseket *három egymásba ágyazott* ciklus hajtja végre, a külső a bevonandó pont szerinti, ezen belül van a viszonylat kezdőpontja szerinti, és a legbelső a viszonylat végpontja szerinti ciklus. Az elvi algoritmus sorrendet nem ír elő, de – mint legegyszerűbbet – a *növekvő* sorrendet alkalmazzuk mind a három ciklusban. A végtelen helyébe egy megfelelően nagy számot veszünk (az összes élhossz összegénél nagyobb szám már megfelelő a körmentes utakhoz {`uGrafD.GVegtelen`}) és ezt „végtelenként” kezeljük.

Adatszerkezet (90)

Az `uGrafD` definícióit kiegészítjük a távolság és címkemátrix adattípussal:

```
type
  TGTavMtx=array[TGPontI, TGPontI] of TGUtHossz;
  TGCimkeMtx=array[TGPontI, TGPontI] of TGPontI;
```

Azonosító	Funkció	Típus	Jelleg
`Graf`	a hálózat	`TGrafMtx`	input
`TT`	a távolságmátrix	`TGTavMtx`	output
`CC`	a címkemátrix	`TGCimkeMtx`	output
`X, Y, W`	pontindexek	`TGPontI`	munka

Szubrutin: `uGrafUt.OsszMinutMtx`.

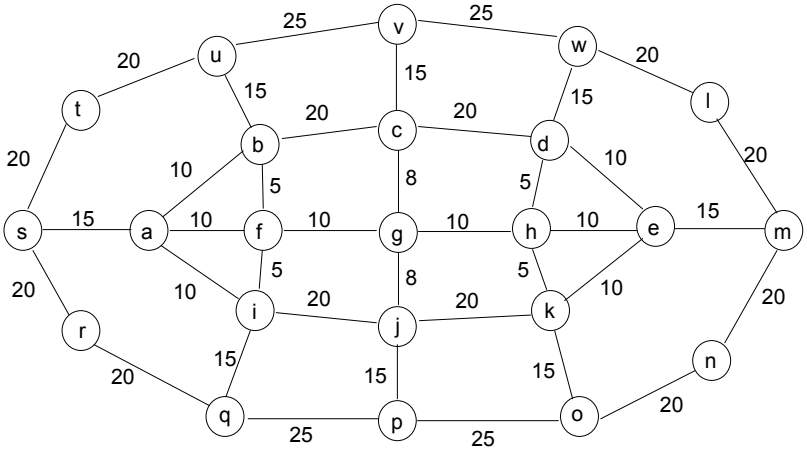
Megjegyzés ♦ Az algoritmus egyszerű és *számításigénye* a pontszám köbével jellemezhető (lásd a három ciklus). Az algoritmus erősen kihasználja, hogy a `TT` és `CC` mátrixok (elemeik két index segítségével közvetlenül, egy lépésben elérhetők) így ezeket ténylegesen operatív tárbeli mátrixként kell tárolni, mert egyéb megoldásoknál (pl. lemezen tárolás) az eljárás egyszerűségét és hatékonyságát veszti.

9.5.1.3. Fa építés

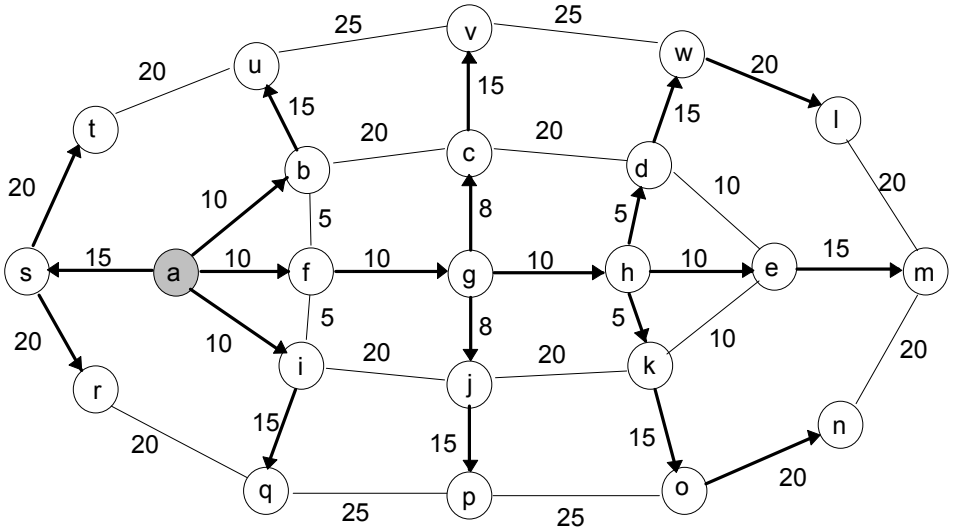
Ha a hálózatunk a számítástechnikai kapacitásunkhoz képest nagy, vagy csak a viszonylatok kisebb részében akarunk minimális utat keresni, akkor alkalmazhatjuk az ún. *faépítő* eljárásokat. Ezek nem az összes viszonylatra, hanem csak az *azonos kezdőpontú* viszonylatokra – tehát egy kezdőpontból az összes többi pontba mint végpontba – határozzák meg a minimális utakat. Az ilyen utak egy gyökeres *fa* struktúrájú részgráfot alkotnak a hálózatban a kezdőponttal mint gyökérponttal, ezt a fát konstruálja meg, építi fel több lépésben az eljárás, innen származik a módszer elnevezése. Az ilyen fát röviden *minimális fának* fogjuk nevezni.

Az eljárás során keletkező fákat *címketömbbel* reprezentáljuk. Az egyszerűbb kezelhetőség kedvéért a gyökérpontnak is címkét adunk, mégpedig saját maga lesz a címke is. A címketömb mellé még felveszünk egy ugyanúgy indexelt *távolságtömböt* is, amely az indexhez megadja a gyökérponttól a pontba (természetesen a fán belül) vezető út hosszát.

A faépítés módszerét az 68. ábra hálózatával fogjuk szemléltetni. Ez a hálózat csupa kétirányú, és a két irányban azonos hosszú éleket tartalmaz. Az 69. ábra az `a` gyökérpontú minimális fát mutatja be a hálózatban (a fához tartozó élek vastagítottak), a 9. táblázaton láthatjuk a megfelelő címke- és távolságtömböt, amelyeket `C` és `T` jelöl.



68. ábra. Hálózat a faépítéshez



69. ábra. Az a gyökérpontú minimális fa

	a	b	c	d	e	f	g	h	i	j	k	l
C	a	a	g	h	h	a	f	g	a	g	h	w
T	0	10	28	35	40	10	20	30	10	28	35	70

	m	n	o	p	q	r	s	t	u	v	w	l
C	e	o	k	j	i	s	a	s	b	c	d	w
T	55	70	50	43	25	35	15	35	25	43	50	70

9. táblázat. Címke- és távolságtömb

Többféle *faépítő* alapeljárás, alapmódszer ismert. Itt egy olyat mutatunk be, amelynek a működése könnyen követhető és a módszeren alapuló konkrét algoritmusok a gyakorlatban is jól alkalmazhatók. Az alapeljárást a szakirodalom első leírójáról *Dijkstra*-féle eljárásnak nevezi. Legegyszerűbben a hálózat pontjaiból képzett halmazok segítségével írható le:

Vezessünk be három jelölést:

- a `K` a *kész pontok* halmaza, elemei már végleges (nem rövidíthető) távolsággal és végleges (nem áthelyezhető) címkével rendelkeznek;
- az `A` az *aktív pontok* halmaza, elemei már rendelkeznek távolsággal és címkével de ezek még változhatnak;
- az `x` kezdő és `y` végpontú él hossza `H(x, y)`.

(Természetesen a faépítés folyamán van még egy harmadik halmaz is, azon pontokból, amelyeknek még egyáltalán nincs címkéje, de ezt nem szükséges külön jegyezni.)

Az algoritmus a `T` és a `C` egy kezdőállapotából kiindulva, a tömböket – tehát a fát – *lépésenként építve és korrigálva,* több lépés megtétele után jut el a végeredményhez. Mint látni fogjuk, pontosan eggyel kevesebb lépés kell, mint ahány pont van a hálózatban. Ezek után az algoritmust a következőképpen definiálhatjuk:

- *Kezdőállapot*
  Legyen az `a` a kezdőpont. Rendeljük a kezdőponthoz a `0`, a többihez a végtelen távolságértéket, a `K` legyen üres, az `A` tartalmazza csak a kezdőpontot, tehát `C[a] = a`, `T[a] = 0`, `K = []`, `A = [a]`. Ez tehát az a fa, amely egy pontból, a gyökérpontból áll.
- *Javító lépések*
  a) Válasszuk ki az `A` minimális távolságú elemét, jelölje ezt `x`, ezt töröljük az `A`-ból és vegyük hozzá a `K`-hoz (ez már kész, nem változik), tehát `K = K + [x]`, `A = A - [x]`.
  b) Az `x`-ből *kiinduló minden él* végpontjára végrehajtandó: Jelölje a végpontot `y`. Megvizsgáljuk, hogy `y` útja `x`-en keresztül rövidíthető-e. Ha igen, a pont címkéjét `x`-re, távolságát a rövidebbre állítjuk, és (ha még nem volt benn) hozzávesszük az `A` halmazhoz. Tehát ha `y` még nem volt a fában, akkor `x` előddel bekerül, ha már bent volt, akkor elődjét lecseréljük `x`-re. Képletekben: ha `T[y] > T[x] + H[x, y]` akkor `T[y] = T[x] + H[x, y]` és `C[y] = x`, `A = A + [y]`.

c) Ha van még aktív elem, vagyis ha az A nem üres, akkor folytatjuk a *b)* ponttól.

- *Végállapot*
  Ha már nincs aktív pont, akkor az eljárás véget ért, a minimális fa készen van, a C és T meghatározzák a végeredményt. Az egyes végpontokhoz tartozó utak a C-ből (a végpontból visszafelé haladva) egyszerűen összerakhatók.

Az algoritmus első négy lépését a példahálózaton bemutatjuk. A 10. táblázatokon követhetjük a halmazok és a tömbök változását, az 70. ábra a negyedik lépés után kialakult állapotot mutatja be. Az itteni fának a kész pontok által meghatározott része már végleges, míg a többi pont előd je, így a fa szerkezete még módosulhat. Összevetve a végállapottal láthatjuk, hogy az eljárás még hátralévő részében módosul pl. a c és j pont előd je.

K	a																					
A	b	f	i	s																		
1	**a**	**b**	**c**	**d**	**e**	**f**	**g**	**h**	**i**	**j**	**k**	**l**	**m**	**n**	**o**	**p**	**q**	**r**	**s**	**t**	**u**	**v**	**w**
C	a	a				a			a										a				
T	0	10	~	~	~	10	~	~	10	~	~	~	~	~	~	~	~	~	15	~	~	~	~

K	a	b																				
A	f	i	s	u	c																	
2	**a**	**b**	**c**	**d**	**e**	**f**	**g**	**h**	**i**	**j**	**k**	**l**	**m**	**n**	**o**	**p**	**q**	**r**	**s**	**t**	**u**	**v**	**w**
C	a	a	b			a			a										a		b		
T	0	10	30	~	~	10	~	~	10	~	~	~	~	~	~	~	~	~	15	~	25	~	~

K	a	b	f																			
A	i	s	g	u	c																	
3	**a**	**b**	**c**	**d**	**e**	**f**	**g**	**h**	**i**	**j**	**k**	**l**	**m**	**n**	**o**	**p**	**q**	**r**	**s**	**t**	**u**	**v**	**w**
C	a	a	b			a	f		a										a		b		
T	0	10	30	~	~	10	20	~	10	~	~	~	~	~	~	~	~	~	15	~	25	~	~

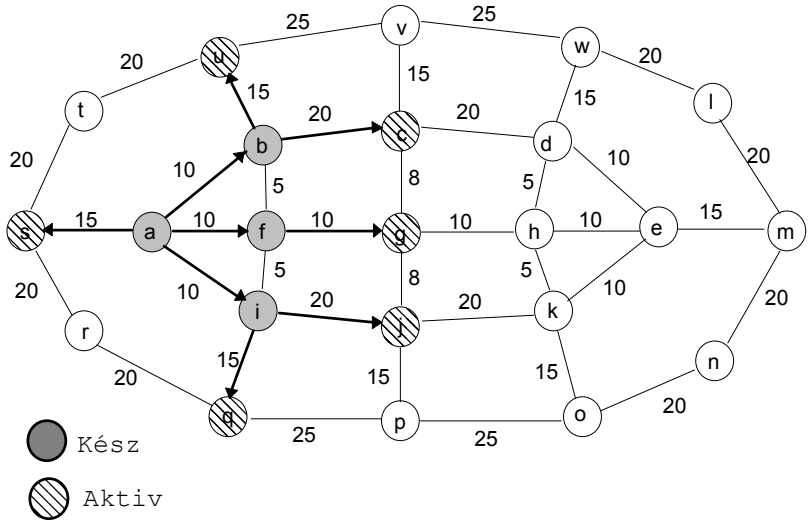
K	a	b	f	i																		
A	s	g	q	u	c	j																
4	**a**	**b**	**c**	**d**	**e**	**f**	**g**	**h**	**i**	**j**	**k**	**l**	**m**	**n**	**o**	**p**	**q**	**r**	**s**	**t**	**u**	**v**	**w**
C	a	a	b			a	f		a	i							i		a		b		
T	0	10	30	~	~	10	20	~	10	30	~	~	~	~	~	~	25	~	15	~	25	~	~

10. táblázat. A tömbök változása az első négy lépésben

Megjegyezzük, hogy a K halmazt csak a jobb szemléltetés kedvéért használtuk, az algoritmusból el is hagyható.

Az eljárást *tárigény* szempontjából elemezve láthatjuk, hogy végrehajtásához csak 3 darab, egyenként *pontszám* elemű tömb szükséges (feltéve, hogy mint a példában, az aktív pontok halmazát is egy tömb tárolja). Ez lehetővé teszi az

alkalmazását nagy (több ezer vagy tízezer pontos) hálózatoknál is. Az is könnyen látható, hogy a faépítés jól illeszkedik a hálózatnak az *éltárolás* formájú tárolási módjához is, hiszen az egy kezdőpontból kimutató élek (amiket az eljárás *b)* pontjában kell sorra vennünk) egymás mellett vannak ennél a tárolási formánál.



70. ábra. A negyedik lépés után kialakult állapot

Az eljárás *számításigénye* a pontszám négyzetével arányos, mivel minden lépésben (az *a)* pontban) átkerül egy pont a kész halmazba (külső ciklus) és ezen belül a *b)* pontban egy maximum pontszámszor futó ciklus megy. Tehát ha az összes utat (minden minimális fát) ki kell számolnunk, akkor sem rosszabb elvi nagyságrendben, mint a mátrixos módszer. Ténylegesen persze itt több a számolás a több adminisztrációs számítás miatt.

A tényleges számításigényt erősen befolyásolja az, hogy hogyan kezeljük az `A` halmazt. A fenti példában ezt egy – a távolság szerint növekvően rendezett – tömbben tartjuk, az új aktív pontot eszerint besoroljuk, ennek következtében a kész halmazba mindig az első aktív pont kerül át. Ez egy egyszerű, könnyen követhető módszer, de nem a legjobb. Az `A` speciális tárolásával – például egy kupac adatstruktúrában – és kezelésével lényegesen csökkenthető a *besorolás–kiválasztás* számításigénye és ezzel a faépítés ideje is.

9.5.1.3.1 mintafeladat: Határozzuk meg minden viszonylatban a minimális utat a Dijkstra módszerrel, az aktív pontok halmazát egy, a távolság szerint rendezett tömbben tárolva!

Útmutató ♦ A módszer az éltárolásos reprezentációhoz igazodik (`uGrafD.TGrafElt`). A pontokat pontindexekkel azonosítjuk. Az aktív pontokat a gyökérponttól vett aktuális távolság szerint növekvően rendezett tömbben tároljuk. A mindenkori távolságértékekhez (minden pont, nemcsak az aktív pontok esetén) a ponttal indexelve közvetlenül hozzáférünk. Azt, hogy egy pont aktuálisan bent van-e az aktív pontok között, célszerű (és egyszerű) egy logikai jelzőtömbbel külön nyilvántartani.

Adatszerkezet (91)

Az `uGrafD` definícióit kiegészítjük a címketömb és az aktivitásjelző adattípusal:

```
type
  TGCimkeTmb=TGPontITmb;
  TGAktivJel=array[TGPontI] of Boolean;  {pontaktivitás
    jelző}
```

Azonosító	Funkció	Típus	Jelleg
`Graf`	a hálózat	`TGrafElt`	input
`T`	a távolságtömb	`TGTavTmb`	output
`C`	a címketömb	`TGCimkeMtx`	output
`KezdoPont`	az utak kezdőpontja	`TGPontI`	input
`ATmb`	aktív pontok tömbje	`TGPontITmb`	munka
`AktJel`	pont aktivitásjelző	`TGAktivJel`	munka
`ADb`	aktív pontok száma	`TGPontDb`	munka
`J`	tömbindex	`TGPontDb`	munka
`X, Y, I`	pontindexek	`TGPontI`	munka
`L`	élindex	`TGElDb`	munka

Szubrutin: `uGrafUt.MinfaTmbElt`.

9.5.1.3.1 mintafeladat: Határozzuk meg minden viszonylatban a minimális utat a Dijkstra módszerrel, az aktív pontok halmazát egy, a távolság szerinti kupacban tárolva!

Útmutató ♦ Az előző feladattól csak annyi a formai különbség, hogy az aktív pontokat a gyökérponttól vett aktuális távolság szerinti kupacban tároljuk. A kupac kezelésére a kupac mintafeladatoknál megismert szubrutinokat használjuk (`uGKupac`).

Adatszerkezet (92)

Itt is az címketömb és az aktivitásjelző adattípust használjuk:

```
type
  TGCimkeTmb=TGPontITmb;
  TGAktivJel=array[TGPontI] of Boolean;  {pontaktivitás
    jelző}
```

Azonosító	Funkció	Típus	Jelleg
`Graf`	a hálózat	`TGrafElt`	input
`T`	a távolságtömb	`TGTavTmb`	output
`C`	a címketömb	`TGCimkeMtx`	output
`KezdoPont`	az utak kezdőpontja	`TGPontI`	input
`AKup`	aktív pontok kupaca	`TGPontITmb`	munka
`AktJel`	pont aktivitásjelző	`TGAktivJel`	munka
`ADb`	aktív pontok száma	`TGPontDb`	munka
`J`	tömbindex	`TGPontDb`	munka
`X, Y, I`	pontindexek	`TGPontI`	munka
`L`	élindex	`TGElDb`	munka

Szubrutin: `uGrafUt.MinfaKupElt`.

Megjegyzés ♦ A kupac adatstruktúra kifejezetten előnyös a faépítésnél, hiszen az aktivitás halmaz sűrűn változik, viszont a *lekérdezés csak a minimális* értékre vonatkozik. Ha összevetjük az előző megoldással, láthatjuk, hogy a minimális távolságú aktív ott is mindig közvetlenül rendelkezésre áll, de a beszúrás és a törlés műveletigénye lényegesen nagyobb, hiszen magával az aktivitás halmaz elemszámával, nem pedig ennek logaritmusával arányos.

9.5.2. Összefüggőség

9.5.2.1. Bevezetés

Önmagában is érdekes és az alkalmazásokban is gyakran előforduló probléma a gráfok ill. a hálózatok *összefüggőségének* vizsgálata. A gráfelméletben többféle „összefüggőség" fogalom is definiált, itt mi csak egyfélével dolgozunk, ahogy fentebb is definiáltuk: *Összefüggő gráf* az olyan gráf, amelyben bármely két pont között van legalább egy út.

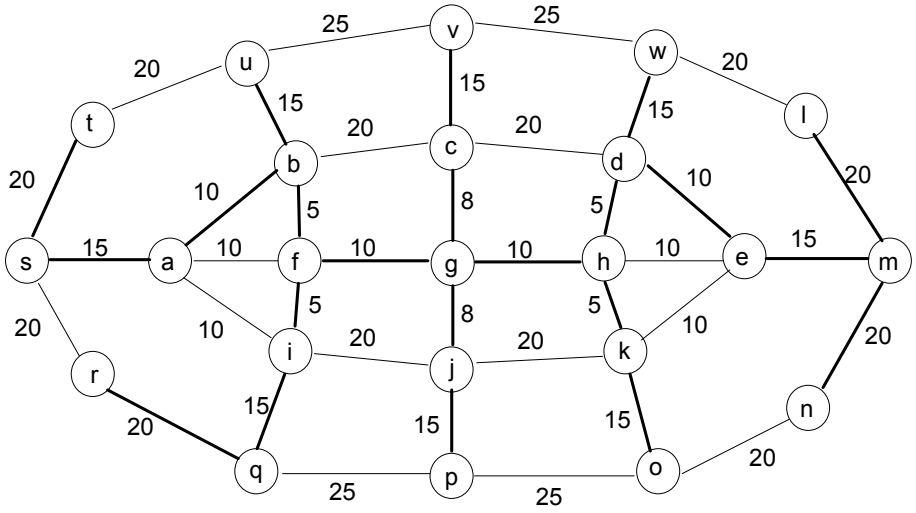
A probléma egyféle megoldása az lenne, ha egyszerűen visszavezetnénk a minimális utak keresésére. A dolog természetesen nem ilyen egyszerű ennek a témakörnek is megvannak a saját speciális feladatai és megoldásai, az alábbiakban ebből hozunk mintafeladatokat.

9.5.2.2. Minimális feszítőfa

Egy összefüggő gráf egy *feszítőfáján* értjük a gráf egy olyan – fa struktúrájú – részgráfját, amely minden pontot tartalmaz. Ha már hálózatról beszélünk, vagyis az éleknek van hossza, akkor egy ilyen fát is minősíthetünk, mérhetünk az *élei hosszának összegével.* Így már beszélhetünk két ilyen fa hossz szerinti összehasonlításáról és kereshetjük a *minimális* hosszúságút. (Megjegyezzük, hogy ilyenből egy hálózaton belül több is lehet.) A minimális hosszúságú feszítőfát a továbbiakban röviden *minimális feszítőfának* nevezzük. A definícióból közvetlenül következik, hogy csak a minimális fa éleit használva is bejárhatjuk a gráfot, eljuthatunk bármely pontból bármely pontba.

A keresés bonyolultsága lényegesen megnő akkor, ha a gráf irányított. Ezért itt csak az egyszerűbb esettel foglalkozunk: a hálózat *irányítatlan,* más szóval minden él két irányú és a két irányban azonos hosszú. Az 71. ábra hálózata ilyen, a vastagított élek egy minimális feszítőfát adnak. A feszítőfáknál nincs jelentősége a gyökérpontnak, ezeket nem szükséges gyökeres faként kezelni.

Az ismertetendő algoritmus (*Kruskal*-féle algoritmus) a gráf pontjaiból képez először egyelemű, majd minden lépésben bővülő halmazokat. Ezek a halmazok önmagukban összefüggő részhálózatoknak felelnek meg. Két ilyen halmazt akkor tudunk egyesíteni, ha van a két részgráfot összekötő él. Az élek vizsgálatát hossz szerinti növekvő sorrendben végezzük. Ha sikerül az összes pontot egy halmazba összeszedni, akkor a gráf összefüggő, és az egyesítéseknél „összekötő" élek egy minimális feszítőfát alkotnak.



71. ábra. Minimális feszítőfa

Az algoritmus pontosabb leírásához vezessünk be néhány jelölést.

- `xy` jelölje az `x` és `y` pontokat összekötő élet.
- `EH` jelölje az élek egy részhalmazát. Ez kezdetben az összes élt tartalmazza. Ebből választjuk ki sorra a vizsgálandó éleket.
- `FH` jelölje az élek egy részhalmazát. Ez kezdetben üres. Ebbe gyűjtjük a fát alkotó éleket.
- `PHH` jelöljön egy olyan halmazt, amelynek elemei is halmazok, mégpedig ponthalmazok. A `PHH` elemei önmagukban összefüggő részhálózatoknak felelnek meg.

Ezek után az algoritmust a következőképpen definiálhatjuk:

- *Kezdőállapot*
  `EH`: minden élt tartalmaz. `FH`: üres. `PHH`: minden egyelemű ponthalmazt tartalmaz.
- *Javító lépések*
  Mindaddig, amíg a `PHH`-ban *legalább kettő halmaz* van és emellett az `EH`-ban is van *legalább egy él,* ismételjük az alábbi műveletsort: Válasszunk ki és töröljünk is az `EH`-ból egy minimális hosszú élet, jelölje ezt `xy`. Ha az `x` és az `y` a `PHH`-ban lévő két különböző halmazban van, akkor egyrészt az `xy` élet vegyük fel az `FH`-ba, másrészt a két halmazt egyesítsük, az egyesített halmazt vegyük fel a `PHH`-ba, az eredetieket pedig töröljük belőle.

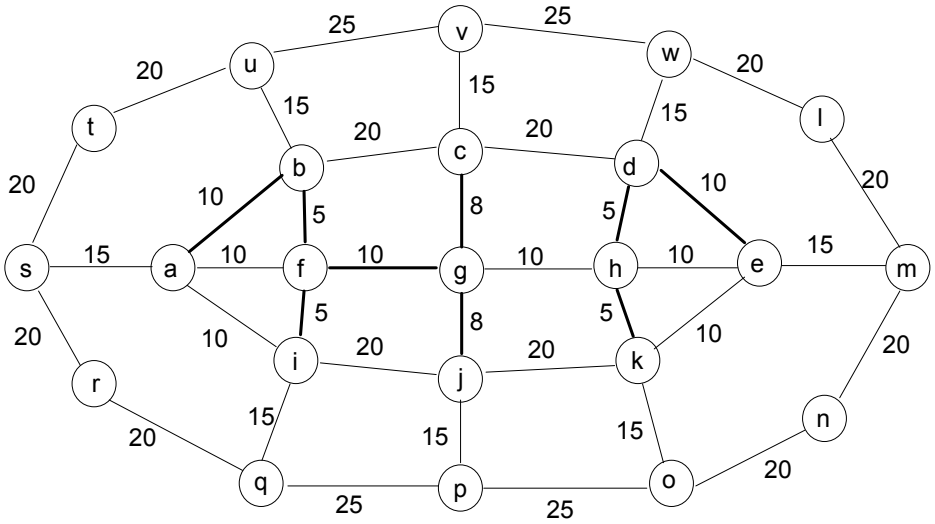
- *Végállapot*
  Ha a javító lépéseket azért nem tudjuk folytatni, mert a PHH-ban *csak egy halmaz* van, akkor készen vagyunk, az FH adja a megoldást. Ha viszont a befejezés oka az, hogy ugyan még *több halmaz* van, de *nincs több él* az EH-ban, akkor a hálózat *nem összefüggő,* nincs feszítőfa.

FH																						
PHH	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w
FH	bf																					
PHH	a	b, f		c	d	e	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w
FH	bf	dh																				
PHH	a	b, f		c	d,h		e	g	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w
FH	bf	dh	fi																			
PHH	a	b, f, i			d,h		e	g	c	j	k	l	m	n	o	p	q	r	s	t	u	v	w

FH	bf	dh	fi	hk	cg	gj	ab	de														
PHH	a, b, f, i				d, h, k, e			g, c, j				l	m	n	o	p	q	r	s	t	u	v	w
FH	bf	dh	fi	hk	cg	gj	ab	de	fg													
PHH	a, b, f, i, g, c, j							d, h, k, e				l	m	n	o	p	q	r	s	t	u	v	w

FH	bf	dh	fi	hk	cg	gj	ab	de	fg	gh	as	bu	cv	dw	em	ko	jp	iq	lm	mn	qr	st	
PHH	a, b, f, i, g, c, j, d, h, k, e, s, u, v, w, m, o, p, q, l, n, r, t																					

11. táblázat. Kezdőállapot, javítólépések és a végállapot



72. ábra. A közbenső lépés utáni fa

Az algoritmust a 11. táblázaton követhetjük a példahálózaton. Megadjuk a kezdőállapotot, az első három javító lépést, egy közbeeső javító lépést és a végálla-

potot. A közbeeső lépés utáni fát a 72. ábra, a végeredményt a 71. ábra is mutatja. A példában az EH halmazt közvetlenül nem jegyezzük. A példához megjegyezzük még, hogy a minimális hosszú él kiválasztásánál – ha erre több lehetőség is volt – az *abc* sorrend szerinti elsőt választottuk mindig ki. (Ez természetesen nem előírás, máshogy választva esetleg egy másik minimális feszítőfát kaphatunk.)

9.5.2.1.1 mintafeladat: Határozzunk meg egy minimális feszítőfát a Kruskal módszerrel (mátrixtárolás)!

Útmutató ♦ Az elvi algoritmus programnyelvi megvalósításánál a halmazok kezelését kell jól megoldanunk.

- A PHH halmaz olyan, közös elem nélküli ponthalmazokból áll, amelyek egyesítése magába foglalja a gráf összes pontját. A PHH elemeit (a halmazokat) sorszámmal azonosítjuk. Az algoritmus bármely lépésében, a gráf valamennyi pontjára igaz az, hogy benne van PHH-nak egy és csakis egy elemében, így minden pontra csak az őt tartalmazó halmaz sorszámát kell feljegyeznünk. Ezzel magát a PHH-t is egyértelműen leírtuk.
- Az algoritmus végrehajtása folyamán a PHH úgy változik, hogy elemeit egyesítjük és töröljük. Ahhoz, hogy a PHH-beli halmazok megtartsák 1-től induló folyamatos sorszámozásukat, a nagyobb sorszámú összevonandó halmaz elemeit áttesszük a kisebb sorszámú halmazba, míg ezen „kiürülő” (nagyobb sorszámú) halmazt úgy töröljük, hogy a legutolsó halmaz elemeit átrakjuk ebbe a halmazba. Ezek a „pontmozgatások” egyetlen ciklussal megtehetők.
- Az EH élhalmaz kezelése egyszerűbben megoldható, egy a mátrixtároláshoz igazodó logikai tömbbel.
- Mivel az eredményt, a minimális feszítőfát ugyanúgy a mátrixtárolásban adjuk meg mint az alaphálózatot, az FH élhalmazt ennek élhosszmátrixaként kezeljük.

Adatszerkezet (93)

Az uGrafD definícióit kiegészítjük a halmazok kezeléséhez szükséges típusokkal:

```
type
  {egy él léte az élhalmazokban}
  TGElVanMtx=array[TGPontI, TGPontI] of Boolean;
```

```
{a PHH ilyen sorszámú halmazában van az adott pont)}
TGPontHely=TGPontI;  {mivel a PHH maximális elemszáma is
  GMaxPontDb}
TGPontHelyek=array[TGPontI] of TGPontI;
```

Azonosító	Funkció	Típus	Jelleg
Graf	a hálózat	TGrafMtx	input
MFFa	a minimális feszítőfa	TGrafMtx	output
MinFFaMtx	a minimális feszítőfa létezése	Boolean	output
PHH	a ponthalmazok halmaza	TGPontHelyek	munka
HDb	PHH elemszáma	TGPontI	munka
A, B	PHH elemindexek	TGPontHely	munka
EH	élhalmaz	TGElVanMtx	munka
EDb	EH elemszáma	TGElDb	munka
X, Y, I, J	pontindexek	TGPontI	munka
Min	munkaváltozó a minimumkereséshez	TGUtHossz	munka

Szubrutin: uGrafFa.MinFFaMtx.

9.5.2.3. Komponensek

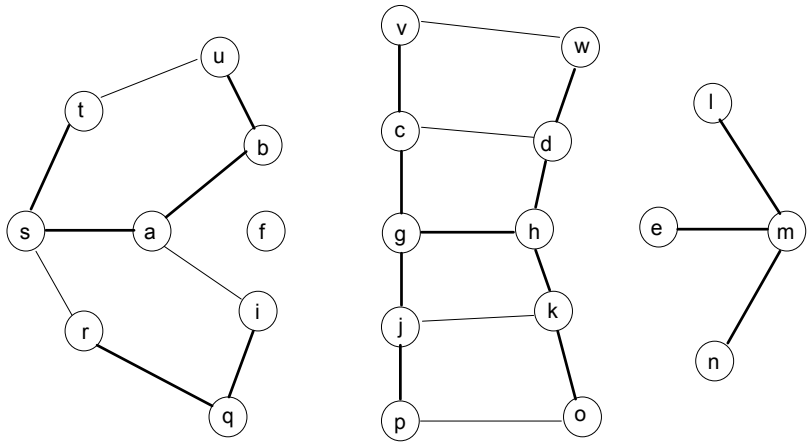
A témakör egy másik alapfeladata a következőképpen fogalmazható meg: *Bontsuk fel a gráfot minimális számú, önmagában összefüggő részgráfra, más néven komponensre!*

A feladat ugyan látszólag más, mint az előbb, de valójában a fenti feladat egy speciális esetéről van szó, amelyet az előbbi algoritmus egy egyszerűsített változatával oldhatunk meg. Mivel itt nincs szó hosszakról (tehát a hálózat csak mint gráf játszik szerepet) az élek vizsgálati sorrendje közömbös. Fára mint eredményre nincs szükségünk, tehát az éleket nem kell gyűjteni. Az algoritmus ezek után:

- *Kezdőállapot*
  EH: minden élt tartalmaz. PHH: minden egyelemű ponthalmazt tartalmaz.
- *Javító lépések*
  Mindaddig, amíg a PHH-ban *legalább kettő halmaz* van és emellett az EH-ban is van *legalább egy él,* ismételjük az alábbi műveletsort: Válasszunk ki és töröljünk is az EH-ból egy élt, jelölje ezt xy. Ha az x és az y a PHH-ban lévő két különböző halmazban van, akkor a két halmazt egyesítsük, az egyesített halmazt vegyük fel a PHH-ba, az eredetieket pedig töröljük belőle.

- *Végállapot*
  A PHH-ban lévő halmazok adják a komponenseket. (Ha csak egy ilyen van, akkor a hálózat összefüggő.)

Az algoritmus szemléltetésére töröltünk néhány élt a fentebbi példahálózatból, így kaptuk a 73. ábra nem összefüggő gráfját. A PHH változását a 12. táblázaton követhetjük. Az ábrán soronként az egy-egy kezdőpontból kiinduló összes él vizsgálata utáni állapotot adjuk meg.



73. ábra. Nem összefüggő gráf

PHH	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o	p	q	r	s	t	u	v	w
PHH	a,b,i,s				e	f	g	h	c	j	k	l	m	n	o	p	q	r	d	t	u	v	w
PHH	a,b,i,s,u					f	g	h	c	j	k	l	m	n	o	p	q	r	d	t	e	v	w
PHH	a,b,i,s,u					f	g	h	c	j	k	l	m	n	o	p	q	r	d	t	e	v	w
PHH	a,b,i,s,u					f	h	c,d,g,v				l	m	n	o	p	q	r	j	t	e	k	w
PHH	a,b,i,s,u					f	c,d,g,v,h,w						m	n	o	p	q	r	j	t	e	k	l
PHH	a,b,i,s,u					f	c,d,g,v,h,w					k	n	o	p	q	r	j	t	e,m		l
PHH	a,b,i,s,u					f	c,d,g,v,h,w,j					n	o	p	q	r	k	t	e,m		l	
PHH	a,b,i,s,u					f	c,d,g,v,h,w,j,k					o	p	q	r	n	t	e,m		l		
PHH	a,b,i,s,u,q					f	c,d,g,v,h,w,j,k					o	p	r	n	t	e,m		l			
PHH	a,b,i,s,u,q					f	c,d,g,v,h,w,j,k,p					o	r	n	t	e,m		l				
PHH	a,b,i,s,u,q					f	c,d,g,v,h,w,j,k,p,o					r	n	t	e,m		l					
PHH	a,b,i,s,u,q					f	c,d,g,v,h,w,j,k,p,o					r	n	t	e,m,l							
PHH	a,b,i,s,u,q					f	c,d,g,v,h,w,j,k,p,o					r	t	n,e,m,l								
PHH	a,b,i,s,u,q,r					f	c,d,g,v,h,w,j,k,p,o					t	n,e,m,l									
PHH	a,b,i,s,u,q,r,t					f	c,d,g,v,h,w,j,k,p,o					n,e,m,l										

12. táblázat. A PHH változása

9.5.2.3.1 mintafeladat: Határozzunk meg egy gráf komponenseit a Kruskal módszerrel (mátrixtárolás)!

Útmutató ♦ A minimális feszítőfa algoritmust értelemszerűen egyszerűsítjük.

Adatszerkezet (94)

Azonosító	Funkció	Típus	Jelleg
`Graf`	a hálózat	`TGrafMtx`	input
`PHH`	a ponthalmazok halmaza	`TGPontHelyek`	output
`HDb`	`PHH` elemszáma	`TGPontI`	output
`A, B`	`PHH` elemindexek	`TGPontHely`	munka
`EH`	élhalmaz	`TGElVanMtx`	munka
`Edb`	`EH` elemszáma	`TGElDb`	munka
`X, Y, I, J`	pontindexek	`TGPontI`	munka
`VanEl`	munkaváltozó az élkereséshez	`Boolean`	munka

Szubrutin: `uGrafFa.KompBontMtx`.

9.6. Feladatok

1 ♦ Mind a három reprezentációban oldjuk meg az alábbi feladatokat hálózatra (feladatonként és reprezentációként külön szubrutin):

- A hálózat betöltése szövegfájlból (`uGrafKez.BetoltesMtx`).
- A hálózat kimentése szövegfájlba (`uGrafKez.MentesElt`).
- Pontadat keresése azonosító alapján.
- Éladat keresése kezdő és végpont indexek alapján (`uGrafKez.ElKeresElt`).
- Pont felvétele/módosítása (`uGrafKez.PontFelveszElt`).
- Él felvétele/módosítása (`uGrafKez.ElFelveszElt`).
- Pont törlése (`uGrafKez.PontTorolMtx`).
- Él törlése.

2 ♦ Konvertáljuk a hálózati adatokat az egyes reprezentációk között:

- Mátrixtárolásból éltárolásba és fordítva!
- Mátrixtárolásból dinamikus adatszerkezetekbe és fordítva!
- Éltárolásból dinamikus adatszerkezetekbe és fordítva!

3 ♦ Mind a három reprezentációban oldjuk meg az alábbi feladatokat hálózatra (feladatonként és reprezentációként külön szubrutin):

- Ellenőrizzük, hogy egy hálózat egy éljellemzőre nézve szimmetrikus-e!
- Szimmetrizáljuk a hálózatot egy éljellemzőre nézve!

4 ♦ Egy gyökeres fát címketömbbel tárolunk. Oldjuk meg az alábbi feladatokat (feladatonként külön szubrutin):

- Állítsunk elő sorsolással egy fát!
- Állapítsuk meg egy pontról, hogy levélpont-e!
- Vegyünk fel egy új levélpontot a fába, adott pont gyermekeként!
- Töröljünk egy levélpontot a fából!
- Töröljünk egy (a gyökérponttól eltérő) pontot a fából, leszármazottait elődjéhez kötve!
- Határozzuk meg egy pontból a gyökérpontba vezető körmentes utat!
- Töröljünk egy részfát a fából!
- Szintezzük be a fát: minden ponthoz vegyük fel jellemzőként a pont szintszámát!
- Szúrjunk be egy fát egy másik fába, úgy, hogy a beszúrandó fa gyökérpontja, egy adott pont gyermeke legyen!
- Állítsunk elő sorsolással egy adott magasságú fát!
- Keressük meg a maximális számú leszármazottal rendelkező pontot!

5 ♦ Oldjuk meg az alábbi feladatokat dinamikus adatszerkezettel tárolt bináris fára (feladatonként külön szubrutin):

- Állítsunk elő sorsolással egy fát!
- Keressünk egy adott azonosítójú pontot (`uGBinFa.BinFanAzonKeres`)!
- Töröljünk egy adott azonosítójú pontot!
- Keressük meg a fa „legbaloldalibb” szabad ágát (a fán csak akkor lépünk jobbra, ha balra nem lehet)!
- Keressük meg a fa „legbaloldalibb” levelét!
- Töröljük a fa összes pontját bejárással, úgy hogy mindig csak levelet törlünk!
- Szintezzük be a fát: minden ponthoz vegyük fel jellemzőként a pont szintszámát!
- Keressünk egy minél kisebb szintszámú szabad ágat!
- Számoljuk meg a fa pontjait!

6 ♦ Oldjuk meg az alábbi feladatokat dinamikus adatszerkezettel tárolt bináris kereső fára (feladatonként külön szubrutin):

- Keressünk egy adott jellemzőjű pontot (`uGBinFa.BinFanAdatKeres`)!
- Bővítsük a fát egy adott azonosítójú és jellemzőjű ponttal. (`uGBinFa.BinFara`)!

- Állítsunk elő sorsolással egy fát!
- Töröljünk egy adott jellemzőjű pontot (`uGBinFa.BinFaRol`)!
- Keressük meg a fa minimális jellemzőjű pontját!
- Keressük meg a fa maximális jellemzőjű pontját!
- Listázzuk a jellemzőket növekvően rendezetten (`uGBinFa.BinFaListaNo`)!
- Listázzuk a jellemzőket csökkenően rendezetten (`uGBinFa.BinFaListaCsokken`)!
- Töröljük a fa minden pontját a gyökérpont ismételt törlésével (`uGBinFa.BinFaTorol`)!

7 ♦ Oldjuk meg a kupacrendezési mintafeladatot egy tömbbel!

8 ♦ Vegyünk fel egy olyan egyszerű kupac adatstruktúrát, amelyben a pontjellemzők stringek, a pontokat külön nem azonosítjuk (pontazonosítót nem kezelünk). A kupacban egyszerre max. 100 db. string lehet. Oldjuk meg az alábbi feladatokat (feladatonként külön szubrutin):

- Vegyünk fel egy stringet a kupacba!
- Vegyük ki a minimális elemet a kupacból!
- Alakítsunk át egy stringtömböt kupaccá!
- Rendezzünk egy stringtömböt kupaccal!

9 ♦ Adott a hálózatnak a Warshall algoritmussal számított címke és távolságmátrixa. Határozzuk meg adott két pont között a minimális távolságot és a minimális utat!

10 ♦ Egy gráfban két pont között a legkedvezőbb útvonalat a minimális számú pontot tartalmazó útvonalként definiáljuk. Határozzuk meg a minimális utakat:

- Minden viszonylatra a Warshall algoritmussal!
- Adott kezdőpontú viszonylatokra a Dijkstra algoritmussal!

11 ♦ Hasonlítsuk össze konkrét hálózatokon a programba épített futási idő méréssel a mintapéldaként adott három algoritmust.

12 ♦ Adott egy hálózat az éltárolásos reprezentációban.

- Keressünk meg egy minimális feszítőfát a Kruskal algoritmussal!
- Határozzuk meg a komponenseket a Kruskal algoritmussal!

10. A MEGOLDÁSOK TESZTELÉSE

10.1. Bevezetés

Az elkészített modelleket „természetes" környezetükben, a számítógépen is próbálni, tesztelni kell. A gondos megtervezés és kivitelezés után a tesztkörnyezetben való kipróbálás sikere is megerősíthet bennünket abban a tudatban, hogy modellünk tartalmilag is helyes, vagyis ténylegesen a kitűzött feladatot oldottuk meg.

A tesztelendő megoldást a Pascal programfejlesztő környezetek sajátosságainak megfelelően, unitokba szervezett deklarációk (konstansok, típusok, változók és szubrutinok) alkotják, tehát a tesztkörnyezet egyik elemét a tesztelendő modellt tartalmazó egy vagy több unit képezi.

Ahhoz, hogy a modell működjön, konkrét input adatokat kell előállítani, az eredmények ellenőrzéséhez pedig meg kell jeleníteni az output adatokat. Ehhez egy megfelelő, végrehajtható modul, egy főprogram szükséges. A teszteléshez szükség lehet még kiegészítő jellegű szoftver erőforrásokra (pl. input adatok előállítása sorsolással), ezeket egy külön unitba csoportosítjuk. Egy tesztprogrammal természetesen több modellt is tesztelhetünk.

Összefoglalva és pontosítva, tesztprogramjaink a következő elemekből állnak:

- A *tesztelendő* modell(eke)t tartalmazó unit(ok). Ezekben a modellt konstansok, típusok és teljesen paraméterezett szubrutinok írják le.
- A *kiegészítő* szoftver erőforrások unitja. Konstansokat, típusokat és teljesen paraméterezett szubrutinokat tartalmazhat.
- A *főprogram,* feladata a felhasználói (tesztelői) kommunikáció lebonyolítása, részletesebben:
  - A teszteléshez szükséges input adatok előállítása/bekérése.
  - A tesztelendő modellek működtetése.
  - Az eredményadatok megjelenítése.

  A főprogram feladatait menüvezérléssel, ezen belül az előzőkben említett unitok egyes szubrutinjainak meghívásával látja el. A főprogram tartalmazza:

  - A modellek működtetéséhez szükséges adatterületeket (globális változókat, ezekből állnak össze a szubrutinhívások aktuális paraméterei).

– A felhasználói kommunikációhoz közvetlenül szükséges szoftvereszközöket.

A kétféle fejlesztői környezet – tesztelendő modelljeink vonatkozásában – a felhasználói kommunikáció, tehát a főprogram feladatkör megvalósításában tér el leginkább.

A mintaprogramokat könyvtárakba (mappákba) összefoglalva adjuk meg.

10.2. Turbo Pascal környezet

A főprogram neve `TFoprog`. Ez a `tKom` unitban megvalósított felhasználói környezet alkalmazásával készül. Deklarációs részében megadjuk a menüvezérléshez szükséges konstansokat és változókat, a modell szubrutinjainak, valamint a kiegészítő funkcióknak a meghívásához szükséges változókat. Utasításrésze a menüciklusból, valamint az ezt megelőző előkészítő és ezt követő befejező utasításokból áll. Az egyes menüpontok végrehajthatósága függhet az aktuális állapottól. Az állapotinformációkat logikai változók aktuális értékeivel adjuk meg.

A *kiegészítő* szoftver erőforrásokat a `tKieg` nevű unitba tesszük.

10.2.1. mintafeladat: Teszteljük magát a `tKom` környezetet, valamint a `tInpA` unitban lévő ellenőrzött input szubrutinokat!

Útmutató ♦ A teszthez egy egyszerű, de a tesztelendő erőforrások minden lényeges részét „megmozgató” feladatot definiálunk. A feladat:

Menüválasztással bekérjük egy árucikk *nevét* (szöveg), *egységárát* (valós szám) valamint *darabszámát* (egész szám), és kiírjuk eredményként a két számadat szorzatát. Eredmény csak akkor számítható, ha már mind a három adat megvan. Az egységár és darabszám bekérési/módosítási sorrendje tetszőleges, de mindkettő csak a név megadása után jöhet. Az árucikk minden adatát kiajánljuk a következőnél módosításra. A végén kiírjuk a teljes összeget.

A megoldásban a `tKieg` a konkrét adatspecifikációkat, valamint az ezekhez az általános bekérő szubrutinokból „levezetett” ellenőrzött szubrutinokat tartalmazza.

Megoldás: `TKomInp` könyvtár.

10.2.2. mintafeladat: Teszteljük a tömbrendező és kereső szubrutinokat!

Útmutató ♦ A tesztelendő szubrutinokat az `uTomb` és `uTombR` tartalmazza. A konkrét tömböket sorsolással, a keresendő értékeket bekéréssel állítjuk elő. A rendezést a kiválasztásos eljárással, a keresést rendezetlen állapotban a soros, rendezett állapotban a bináris eljárással végezzük. Tehát a közvetlenül tesztelt szubrutinok az `uTomb.Keres`, `uTombR.KivalRend` és `uTombR.BinKer`. A tesztelő keretprogramban a konkrét kereső és rendező szubrutinok könnyen cserélhetők. A megoldásban a `tKieg` a konkrét tömbök előállítását és kiírását tartalmazza.

Megoldás: `TRendKer` könyvtár.

10.2.3. mintafeladat: Teszteljük a bináris keresőfát kezelő eszközöket!

Útmutató ♦ A közvetlenül tesztelendő szubrutinokat az `uGBinFa` unit tartalmazza. Mivel az ebben felhasznált egyik szoftver erőforrás, az új dinamikus változó előállítása fejlesztési környezet függő (`tGrafU` vagy `dGrafU`), a teszteléshez a `TFoprog` és a `tKieg` mellé készítenünk kell egy `tGBinFa` változatot is. (A `GBinFa` változatok csak abban különböznek, hogy a `uses` listán a `tGrafU` vagy a `dGrafU` szerepel). A fát előállíthatjuk sorsolással, vagy indulhatunk az üres fából, majd bekért adatokkal tesztelhetjük a legfontosabb karbantartó funkciókat (beszúrás, törlés, keresés). A megoldásban a `tKieg` a sorsolási, bekérési és kiírási eszközöket tartalmazza.

Megoldás: `TBinFa` könyvtár.

10.3. Delphi környezet

A fejlesztőrendszer komponensek formájában készen tartalmazza a felhasználói kommunikáció eszközeit. Ezek közül csak a legszükségesebbeket alkalmazzuk. A tesztelő környezetek az objektumorientáció és a Delphi rendszerek mélyebb ismerete nélkül, az itt közölt „recept” és a mintapéldák alapján is előállíthatók.

A főprogram funkciót két modul, a `DPFoprog` projekt modul és az alkalmazás fő ablakát (formját) megadó `DFFoprog` unit modul írja le. A projekt modul, valamint a főablak modul üres alakja automatikusan létrejön. A felhasználói adatbekérés és eredménykiírás megkönnyítésére egy további modult készítettünk, ez a `dInpOut` unit.

Az általános teendők:

- Indítsunk egy új projektet, vagy induljunk ki a fejlesztő rendszer indításakor automatikusan létrejött új projektből (`Project1`, `Unit1`). A kívánt modulneveket (`DPFoprog`, `DFFoprog`) a fejlesztőrendszerbeli megfelelő mentő (`Save As…`) funkciókkal állíthatjuk be. Az automatikusan `Form1` névvel megjelenő főablak formát nevezzük át (`Name` tulajdonság) `Forma`-ra.
- A tesztelendő és kiegészítő (`dInpOut`, `dKieg`) unitokat hozzá kell venni a projekthez (`Add to Project…`).
- A tesztelendő modell adatszerkezetének főbb, több menüpontban is használt változóit a `Forma` *privát* szekciójában helyezzük el. Ugyanide tegyük a közvetlenül a `Forma`-hoz kötődő, a felhasználói kommunikációhoz szükséges szubrutinokat is, ha vannak ilyenek.
- A `Forma`-hoz készítsünk `OnCreate` eseménykezelő metódust (`TForma.FormCreate`), ide tegyük az előkészítő jellegű tevékenységeket.
- Tegyünk a formára egy főmenü (`TMainMenu`) komponenst, nevezzük el `MFoMenu`-nek, a menüszerkesztővel adjuk hozzá, a funkcióra utaló névvel (`Name`) és felirattal (`Caption`) a menüpontokat. Utolsó menüpontként egy kilépés (`MKilep`, Kilépés) menüpontot tegyünk.
- A menüpontokhoz készítsünk `OnClick` eseménykezelő metódusokat (pl. `TForma.MKilepClick`).
- A kilépés menüpont eseménykezelője a befejező tevékenységeket és utolsó utasításként a formát (és ezzel a projektet is) lezáró `Close;` utasítást tartalmazza.
- A többi menüpont eseménykezelője a megfelelő funkció végrehajtásához szükséges szubrutinhívásokat és az ezekhez szükséges lokális deklarációkat tartalmazza.
- Az egyes menüpontoknak a konkrét állapottól való végrehajthatóságát a menüpont `Enabled` (engedélyezett) tulajdonságának beállításával szabályozzuk.
- Tegyünk a formára `Memo` névvel egy memo (`TMemo`) komponenst, erre fogjuk kiírni az eredményeket. A komponenst terjesszük ki a forma teljes szabad területére (`Align = alClient`), állítsuk be minden irányban görgethetőre (`ScrollBars = ssBoth`). Ha az eredmények kiírásánál szükséges a függőleges igazítottság (adatoszlopok, mátrixok), akkor tiltsuk le az

automatikus sortördelést (`WordWrap = False`) és állítsunk be egyenközű betűtípust (pl. `Font.Name = Courier New`).

A felhasználói (billentyűzet–képernyő) ellenőrzött inputhoz a nyelv `InputQuery` függvényét használjuk. A felhasználás módjára a `dInpOut.AltSzovBe` és a `dInpOut.EgSzamBe` szubrutinok adnak példát.

Az eredményeket a `Memo` jeleníti meg. Erre legegyszerűbb soronként egy stringet írni. Ehhez segédeszköz a `dInpOut.MemoSorIr` szubrutin. Az eredmények kiírását ehhez alkalmazkodva szervezzük meg (lásd mintapéldák).

10.3.1. mintafeladat: Teszteljük a tömbrendező és kereső szubrutinokat!

Útmutató ♦ A fejlesztőkörnyezethez értelemszerűen igazítva ugyanaz, mint az előző pont megfelelő feladatánál. Megjegyezzük, hogy itt a memo görgethetősége elhagyhatóvá teszi a Turbo Pascal megoldásban a jobb követhetőség kedvéért alkalmazott automatikus tömbkiírást.

Megoldás: `DRendKer` könyvtár.

10.3.2. mintafeladat: Teszteljük a bináris keresőfát kezelő eszközöket!

Útmutató ♦ A fejlesztő környezethez értelemszerűen igazítva ugyanaz, mint az előző pont megfelelő feladatánál.

Megoldás: `DBinFa` könyvtár.

IRODALOMJEGYZÉK

1. A.V. Aho – J.E.Hopcroft – J.D.Ullman: *Számítógép algoritmusok tervezése és analízise.* Műszaki Könyvkiadó.
2. Knuth: *A számítógép-programozás művészete I–III.* Műszaki Könyvkiadó, Budapest, 1987. ISBN 963 16 0078 5 (összefoglaló)
3. Marton – Pukler – Pusztai: *Bevezetés a programozásba.* NOVADAT, Győr, 1998. ISBN 963 04 3432 6.
4. Marton – Pusztai: *Gráfok és hálózatok kezelése számítógéppel I–VI.* Új Alaplap 1997/4–9.
5. Marton: *Bevezetés a Pascal nyelvű programozásba.* NOVADAT, Győr, 1998, ISBN 963 04 3432 6.
6. Marton: *Programozási példatár.* Tankönyvkiadó, Budapest, 1982.
7. S. Lipschutz: *Adatszerkezetek.* Panem – McGraw-Hill, ISBN 963 7628 67 3.
8. T.H. Cormen – C.E. Leiserson – R.L. Rivest: *Algoritmusok.* Műszaki Könyvkiadó, Budapest. ISBN 963 16 1389 5
9. Wirth: *Algoritmusok + Adatstruktúrák = Programok.* Műszaki Könyvkiadó, Budapest, ISBN 963 10 3858 0.

FÜGGELÉK

Unitok

Tesztprogramok

Unitok

dGrafU unit

```
unit dGrafU;

interface

uses uGrafD;

function UjBinFaPont: PBinFaPont;

implementation

function UjBinFaPont: PBinFaPont;
begin
  try
    New(Result);
  except Result:=nil; end;
end;
end.
```

dInpOut unit

```
unit dInpOut;

interface

uses Dialogs, StdCtrls, uDef;

{ Általános szöveg }
function AltSzovBe(
  const FejS:    String;  {i fejléc a bekéréshez}
  var   Szoveg:  String;  {i/o adat szöveg}
        AlapJel: TJelek;  {i alapjelek}
        MinAdH:  Byte;    {i min adathossz}
        MaxAdH:  Byte;    {i max adathossz}
        ElvJel:  TJelek;  {i elválasztó jelek}
        MinEDb:  Byte;    {i min elválasztójel db}
        MaxEDb:  Byte):   {i max elválasztójel db}
                 Boolean  {o érvényesség};

{ Longint }
function EgSzamBe(
  const FejS:    String;   {i fejléc a bekéréshez}
  var   SzamErt: Longint;  {i/o eredményszám}
        Tol, Ig: Longint): {i határok}
                 Boolean;  {o van-e új adat}

procedure MemoSorIr(AMemo: TMemo; S: String);

implementation
```

```
function AltSzovBe(const FejS: String; var Szoveg: String; AlapJel: TJelek;
  MinAdH: Byte; MaxAdH: Byte; ElvJel: TJelek; MinEDb: Byte;
  MaxEDb: Byte): Boolean;
var
  VanAdat, JoAdat: Boolean;  S: String;
  Jel: Char;  JoJel: TJelek;  ElvDb, I, H: Byte;
begin
  JoAdat:=False;  S:=Szoveg;
  repeat
    VanAdat:=InputQuery(FejS, '', S);
    if VanAdat then begin
      H:=Length(S);  JoAdat:=(H<=MaxAdH) and (H>=MinAdH);
      if JoAdat then begin
        JoJel:=AlapJel;  ElvDb:=0;
        for I:=1 to H do if JoAdat then begin
          Jel:=S[I];  JoAdat:=JoAdat and (Jel in JoJel);
          if Jel in AlapJel then JoJel:=AlapJel+ElvJel
          else begin JoJel:=AlapJel; Inc(ElvDb); end;
        end;
        JoAdat:=JoAdat and (ElvDb<=MaxEDb) and (ElvDb>=MinEDb)
          and (S[H] in AlapJel);
        if JoAdat then Szoveg:=S;
      end;
    end;
  until JoAdat or not VanAdat;
  Result:=JoAdat;
end;

function EgSzamBe(const FejS: String; var SzamErt: Longint;
  Tol, Ig: Longint): Boolean;
const
  JoJel=Szamjegyek+['+', '-'];
var
  VanAdat, JoAdat: Boolean;  S: String;
  I: Integer;  R: Longint;
begin
  Str(SzamErt, S);
  repeat
    VanAdat:=InputQuery(FejS, '', S);  JoAdat:=False;
    if VanAdat then begin
      JoAdat:=True;
      for I:=1 to Length(S) do JoAdat:=JoAdat and (S[I] in JoJel);
      if JoAdat then begin
        Val(S, R, I);  JoAdat:=(I=0) and (R>=Tol) and (R<=Ig);
        if JoAdat then Szamert:=Trunc(R);
      end;
    end;
  until JoAdat or not VanAdat;
  Result:=JoAdat;
end;

procedure MemoSorIr(AMemo: TMemo; S: String);
begin
  AMemo.Lines.Add(S);
end;
end.
```

dListaU unit

```
unit dListaU;

interface

uses uListaD, uListaLD;

function UjLElem1: PLElem1;
function UjLElem2: PLElem2;
function UjLotElem: PLotElem;

implementation

function UjLElem1: PLElem1;
begin
  try
    New(Result);
  except Result:=nil; end;
end;

function UjLElem2: PLElem2;
begin
  try
    New(Result);
  except Result:=nil; end;
end;

function UjLotElem: PLotElem;
begin
  try
    New(Result);
  except Result:=nil; end;
end;
end.
```

dMKollU unit

```
unit dMKollU;

interface

uses uMKollD;

function UjMTetel: PMSor;

implementation

function UjMTetel: PMSor;
begin
  try
    New(Result);
  except Result:=nil; end;
end;
end.
```

dSKollU unit

```
unit dSKollU;

interface

uses uSKollD;

function UjSTetel: PSTet;

implementation

function UjSTetel: PSTet;
begin
  try
    New(Result);
  except Result:=nil; end;
end;
end.
```

tDef unit

```
unit tDef;

interface

const
  SzamJegyek=['0'..'9'];  NagyBetuk=['A'..'Z'];  KisBetuk=['a'..'z'];
  ENBetuk=['É','Ö','Ü'];  EKBetuk=['á','é','ú','ó','í','ü','ö'];
  Betuk=NagyBetuk+KisBetuk+ENBetuk+EKBetuk;
  AzonJelek=NagyBetuk+KisBetuk+SzamJegyek+['_'];
  KozJelek=[' ', ',', '.', '/', '-'];
  KiLep=#27;  {Esc}  Valaszt=#13;  {Enter}
  {a megfelelő billentyűk második jelei}
  Fel=#72;  Le=#80;  Jobb=#77;  Bal=#75;  {kurzor billentyűk}
  Eleje=#71;  {Home}  Vege=#79;  {End}
  LapKov=#73;  {PgDown}  LapElo=#81;  {PgUp}
  Iranyok=[Fel, Le, Jobb, Bal, Eleje, Vege, LapKov, LapElo];
  Lepesek=Iranyok+[KiLep, Valaszt];
  Torol=#8; {visszatörlés, BackSpace}  AdatVeg=Valaszt; {adatvég, Enter}
  Ures=' ';  {helykitöltő, jelző}  Elojel='-';  TPont='.';
  InpUzen='Adatvég: Enter  Javítás: BackSpace  Kilépés: Esc';
  Vilagos=7;  Sotet=0;  MaxMenuSor=9;
  MenuInfo='Keres: '#24' '#25' Home End  Választ: Enter  Kilép: Esc';

type
  SzamJegy='0'..'9';  NagyBetu='A'..'Z';  KisBetu='a'..'z';
  TJelek=set of Char;  Irany=Char;  Lepes=Char;
  IrasMod=(Normal, Inverz, Marad);
  TEgSzamHossz=0..11;  {Longint, előjellel}
  MenuInd=0..MaxMenuSor;
  MenuSorok=array[MenuInd] of String;  {menüsorok, a 0. elem cím}

implementation
end.
```

tGrafU unit

```
unit tGrafU;

interface

uses uGrafD;

function UjBinFaPont: PBinFaPont;

implementation

function UjBinFaPont: PBinFaPont;
var
  U: PBinFaPont;
begin
  if MaxAvail>Sizeof(TBinFaPont) then New(U)
  else U:=nil;
  UjBinFaPont:=U;
end;
end.
```

tInpA unit

```
unit tInpA;

interface

uses Crt, uString, tDef, tKom;

{ Általános szöveg }
function AltSzovBe(
  var Szoveg:          String;   {i/o eredmény szöveg}
      Oszl, Sor:       Byte;     {i képernyőpozíció}
      AlapJel:         TJelek;   {i alapjelek}
      MinAdH, MaxAdH: Byte;     {i min, max adathossz}
      ElvJel:          TJelek;   {i elválasztó jelek}
      MinEDb, MaxEDb: Byte):    {i min, max elválasztó jel db}
                       Boolean;  {o van-e új adat}

{ Longint }
function EgSzamBe(
  var SzamSzov:  String;     {i/o eredmény szöveg}
      Oszl, Sor: Byte;       {i képernyőpozíció}
      Tol, Ig:   Longint;    {i határok}
  var SzamErt:   Longint):   {o eredmény szám}
                 Boolean;    {o van-e új adat}

{ Real }
function ValSzamBe(
  var SzamSzov:  String;     {i/o eredmény szöveg}
      Oszl, Sor: Byte;       {i k,pernyőpozíció}
      Tol, Ig:   Real;       {i határok}
      MaxH:      Byte;       {i mezőhossz}
  var Szamert:   Real):      {o eredmény szám}
                 Boolean;    {o van-e új adat}
```

```
implementation

function AltSzovBe(var Szoveg: String; Oszl, Sor: Byte; AlapJel: TJelek;
  MinAdH, MaxAdH: Byte; ElvJel: TJelek; MinEDb, MaxEDb: Byte): Boolean;
var
  Jel: Char;  {aktuális jel}  JoJel: TJelek;  {aktuális jelhalmaz}
  Hossz: Byte;  {aktuális hossz}  ElvDb: Byte;  {aktuális elválasztójel db}
  I: Byte;  VanAdat: Boolean;
begin
  {előkészítés}
  {adat}
  Hossz:=Length(Szoveg);  Szoveg:=JobbTolt(Szoveg, MaxAdH);
  ElvDb:=0;
  if Hossz>0 then begin
    for I:=1 to Hossz do if Szoveg[I] in ElvJel then Inc(ElvDb);
    Jel:=Szoveg[Hossz];
  end;
  {képernyő}
  if Oszl=0 then Oszl:=WhereX;  if Sor=0 then Sor:=WhereY;
  KiIras(Szoveg, Oszl, Sor, Hossz, Inverz);
  {beolvasás}
  repeat
    {jojel halmaz konstrukció}
    {kilépés}
    JoJel:=[KiLep];
    {alapjelek}
    if Hossz<MaxAdh then JoJel:=JoJel+AlapJel;
    {elválasztó jelek}
    if (Hossz>0) and (Hossz<MaxAdH-1) and not (Jel in ElvJel)
      and (ElvDb<MaxEDb) then JoJel:=JoJel+ElvJel;
    {visszatörlés}
    if Hossz>0 then JoJel:=JoJel+[Torol];
    {adatvég}
    if (Hossz>=MinAdH) and not (Jel in ElvJel) and (ElvDb>=MinEDb) then
      JoJel:=JoJel+[AdatVeg];
    {jel beolvasás}
    Jel:=JelBe(JoJel);
    {jel feldolgozás}
    {alapjelek}
    if Jel in AlapJel then begin
      Hossz:=Hossz+1;  Szoveg[Hossz]:=Jel;
    end else  {elválasztó jelek}
    if Jel in ElvJel then begin
      Hossz:=Hossz+1;  ElvDb:=ElvDb+1;  Szoveg[Hossz]:=Jel;
    end else {visszatörlés}
    if Jel=Torol then begin
      if Szoveg[Hossz] in ElvJel then ElvDb:=ElvDb-1;
      Szoveg[Hossz]:=Ures;  Hossz:=Hossz-1;
      if Hossz>0 then Jel:=Szoveg[Hossz];
    end;
    KiIras(Szoveg, Oszl, Sor, Hossz, Marad);
  until Jel in [KiLep, AdatVeg];
  {befejezés}
  KiIras(JobbTolt('', MaxAdH), Oszl, Sor, MaxAdH, Normal);
  VanAdat:=Jel<>KiLep;
  if VanAdat then Szoveg[0]:=Chr(Hossz) else Szoveg:='';
```

```
  KiIras(Szoveg, Oszl, Sor, Hossz, Normal);
  AltSzovBe:=VanAdat;
end;

function EgSzamBe(var SzamSzov: String; Oszl, Sor: Byte; Tol, Ig: Longint;
  var SzamErt: Longint): Boolean;
var
  Jel: Char;  {aktuális jel}  Hossz: TEgSzamHossz;  {aktuális hossz}
  JoJel: TJelek;  {aktuális jelhalmaz}
  JoSzam: Boolean;  {tartomány helyesség}
  W: String;  MaxH: TEgSzamHossz;  I: Integer;  X: Longint;
  VanAdat: Boolean;
begin
  {előkészítés}
  {adat}
  Str(Tol, W);  MaxH:=Length(W);  Str(Ig, W);
  if MaxH<Length(W) then MaxH:=Length(W);
  Hossz:=Length(SzamSzov);  SzamSzov:=JobbTolt(SzamSzov, MaxH);
  {képernyő}
  if Oszl=0 then Oszl:=WhereX;  if Sor=0 then Sor:=WhereY;
  KiIras(SzamSzov, Oszl, Sor, Hossz, Inverz);
  {beolvasás}
  repeat {második szint: értéktartomány}
    repeat {első szint: csak szám}
      {jójel beállítás}
      {kilépés}
      JoJel:=[KiLep];
      {alapjelek}
      if Hossz<MaxH then JoJel:=JoJel+SzamJegyek;
      if (Hossz=0) and (Tol<0) then JoJel:=JoJel+[Elojel];
      {visszatörlés}
      if Hossz>0 then JoJel:=JoJel+[Torol];
      {adatvég}
      if (Hossz>0) and (SzamSzov[Hossz] in SzamJegyek) then
        JoJel:=JoJel+[AdatVeg];
      {jel beolvasás}
      Jel:=JelBe(JoJel);
      {jel feldolgozás}
      {alapjelek}
      if (Jel in SzamJegyek) or (Jel=Elojel) then begin
        Hossz:=Hossz+1;  SzamSzov[Hossz]:=Jel;
      end else {visszatörlés}
      if Jel=Torol then begin
        SzamSzov[Hossz]:=Ures;  Hossz:=Hossz-1;
      end;
      KiIras(SzamSzov, Oszl, Sor, Hossz, Marad);
    until Jel in [KiLep, AdatVeg];
    {tartomány ellenőrzés}
    if Jel<>KiLep then begin
      Val(Copy(SzamSzov, 1, Hossz), X, I);
      JoSzam:=(I=0) and (X>=Tol) and (X<=Ig);
    end;
  until (Jel=KiLep) or JoSzam;
  {befejezés}
  VanAdat:=Jel<>KiLep;
  KiIras(JobbTolt('', MaxH), Oszl, Sor, MaxH, Normal);
```

```
  if VanAdat then begin
    SamSzov[0]:=Chr(Hossz);  SzamErt:=X;
  end else SzamSzov:='';
  KiIras(SzamSzov, Oszl, Sor, Hossz, Normal);
  EgSzamBe:=VanAdat;
end;

function ValSzamBe(var SzamSzov: String; Oszl, Sor: Byte; Tol, Ig: Real;
  MaxH: Byte; var Szamert: Real): Boolean;
var
  Jel: Char;  {aktuális jel}  JoJel: TJelek;  {aktuális jelhalmaz}
  Hossz: Byte;  {aktuális hossz}  W: String;  {szöveg munkaterület}
  JoSzam: Boolean;  {tartalmi helyesség}
  VoltPont: Boolean;  {volt-e tizedespont}
  I: Integer;  X: Real;  VanAdat: Boolean;
begin
  {előkészítés}
  Str(Trunc(Tol), W);  if Length(W)>MaxH then MaxH:=Length(W);
  Str(Trunc(Ig), W);  if Length(W)>MaxH then MaxH:=Length(W);
  Hossz:=Length(SzamSzov);  SzamSzov:=JobbTolt(SzamSzov, MaxH);
  if Hossz>0 then Jel:=SzamSzov[Hossz];
  VoltPont:=Pos(TPont, SzamSzov)>0;
  {képernyő}
  if Oszl=0 then Oszl:=WhereX;  if Sor=0 then Sor:=WhereY;
  KiIras(SzamSzov, Oszl, Sor, Hossz, Inverz);
  {beolvasás}
  repeat {második szint: értéktartomány}
    repeat {első szint: csak szám}
      {JoJel halmazkonstrukció}
      {kilépés}
      JoJel:=[Kilep];
      {alapjelek}
      if (Hossz=0) and (Tol<0) then JoJel:=JoJel+[Elojel];
      if (Hossz>0) and (Hossz<MaxH) and not VoltPont then
        JoJel:=JoJel+[TPont];
      if Hossz<MaxH then JoJel:=JoJel+SzamJegyek;
      {visszatörlés}
      if Hossz>0 then JoJel:=JoJel+[Torol];
      {adatvég}
      if Jel in SzamJegyek+[TPont] then JoJel:=JoJel+[AdatVeg];
      {Jel beolvasás}
      Jel:=JelBe(JoJel);
      {Jel feldolgozás}
      {alapjelek}
      if Jel in (SzamJegyek+[TPont, Elojel]) then begin
        Inc(Hossz);  SzamSzov[Hossz]:=Jel;
        VoltPont:=VoltPont or (Jel=TPont);
      end else
      {visszatörlés}
      if Jel=Torol then begin
        VoltPont:=VoltPont and (SzamSzov[Hossz]<>TPont);
        SzamSzov[Hossz]:=Ures;  Dec(Hossz);
        if Hossz>0 then Jel:=SzamSzov[Hossz];
      end;
      KiIras(SzamSzov, Oszl, Sor, Hossz, Marad);
    until (Jel in [Kilep, AdatVeg]);
```

```
    if Jel<>Kilep then Val(Copy(SzamSzov, 1, Hossz), X, I);
    JoSzam:=(I=0) and (X>=Tol) and (X<=Ig);
  until (Jel=Kilep) or JoSzam;
  {befejezés}
  VanAdat:=Jel<>Kilep;
  KiIras(JobbTolt('', MaxH), Oszl, Sor, MaxH, Normal);
  if VanAdat then begin
    SzamSzov[0]:=Chr(Hossz);  SzamErt:=X;
  end else SzamSzov:='';
  KiIras(SzamSzov, Oszl, Sor, Hossz, Normal);
  ValSzamBe:=VanAdat;
end;
end.
```

tInpB unit

```
unit tInpB;

interface

uses Crt, uString, tDef, tKom, tInpA;

function SzemNevBe(var Nev: String; Oszl, Sor: Byte): Boolean;

type
  TestSuly=Byte;

function TestSulyBe(var SulyStr: String; Oszl, Sor: Byte;
  var Suly: TestSuly): Boolean;
function RendSzamBe(var RendSzam: String; Oszl, Sor: Byte): Boolean;

implementation

function SzemNevBe(var Nev: String; Oszl, Sor: Byte): Boolean;
begin
  SzemNevBe:=AltSzovBe(Nev, Oszl, Sor, Betuk, 3, 25, [' '], 1, 2);
end;

function TestSulyBe(var SulyStr: String; Oszl, Sor: Byte;
  var Suly: TestSuly): Boolean;
var
  L: Longint;  VanAdat: Boolean;
begin
  VanAdat:=EgSzamBe(SulyStr, Oszl, Sor, 0, High(TestSuly), L);
  if VanAdat then Suly:=L;
  TestSulyBe:=VanAdat;
end;

function RendSzamBe(var RendSzam: String; Oszl, Sor: Byte): Boolean;
const
  AdatHossz=6;
  RBetuk=NagyBetuk+KisBetuk;  AdatJel=RBetuk+SzamJegyek;
var
  Jel: Char;  {aktuális jel}  JoJel: TJelek;  {aktuális jelhalmaz}
  Hossz: Byte;  {aktuális hossz}
  I: Byte;  VanAdat: Boolean;
```

```
begin
  {előkészítés}
  {adat}
  Hossz:=Length(RendSzam);  RendSzam:=JobbTolt(RendSzam, AdatHossz);
  {képernyő}
  if Oszl=0 then Oszl:=WhereX;  if Sor=0 then Sor:=WhereY;
  KiIras(RendSzam, Oszl, Sor, Hossz, Inverz);
  {beolvasás}
  repeat
    {jojel beállítás}
    {kilépés}
    JoJel:=[KiLep];
    {normál jelek}
    case Hossz of
      0..1: JoJel:=JoJel+RBetuk;
         2: JoJel:=JoJel+AdatJel;
      3..5: JoJel:=JoJel+SzamJegyek;
    end;
    if (Hossz=5) and (RendSzam[4]+RendSzam[5]='00') then
      JoJel:=JoJel-['0'];
    {visszatörlés}
    if Hossz>0 then JoJel:=JoJel+[Torol];
    {adatvég}
    if Hossz=AdatHossz then JoJel:=JoJel+[AdatVeg];
    {jel beolvasás}
    Jel:=JelBe(JoJel);
    {jel feldolgozás}
    {normál jelek}
    if Jel in AdatJel then begin
      Inc(Hossz);  RendSzam[Hossz]:=UpCase(Jel)
    end else {visszatörlés}
    if Jel=Torol then begin RendSzam[Hossz]:=Ures;  Dec(Hossz); end;
    KiIras(RendSzam, Oszl, Sor, Hossz, Marad);
  until (Jel in [KiLep, AdatVeg]);
  {befejezés}
  VanAdat:=Jel<>KiLep;
  KiIras(JobbTolt('', AdatHossz), Oszl, Sor, AdatHossz, Normal);
  if VanAdat then RendSzam[0]:=Chr(Hossz) else RendSzam:='';
  KiIras(RendSzam, Oszl, Sor, Hossz, Normal);
  RendSzamBe:=VanAdat;
end;
end.
```

tKoll unit

```
unit tKoll;

interface

const
  MaxSorI=10000;

type
  TSorI=1..MaxSorI;  TSorDb=0..MaxSorI;  PString=^String;
  TSzovMut=array[TSorI] of PString;  {stringkollekció mutatótömb}
```

```
function StrKollBe(
      Fajl:    String;    {i rendezendő szövegfájl}
  var StrKoll: TSzovMut;  {o stringkollekció mutatótömb}
  var SorDb:   TSorDb):   {o stringkollekció tételek száma}
               Boolean;   {o beolvasás sikeres-e}

procedure StrKollRend(
  var StrKoll: TSzovMut;  {i/o stringkollekció mutatótömb}
      SorDb: TSorDb);     {i stringkollekció tételek száma}

procedure StrKollKi(
        Fajl:    String;    {i eredmény szövegfájl}
  const StrKoll: TSzovMut;  {i stringkollekció mutatótömb}
        SorDb:   TSorDb);   {i stringkollekció tételek száma}

implementation

function StrKollBe(Fajl: String; var StrKoll: TSzovMut;
  var SorDb: TSorDb): Boolean;
var
  F: Text;  VanHely: Boolean;
begin
  Assign(F, Fajl);  Reset(F);
  SorDb:=0;  VanHely:=MaxAvail>SizeOf(String);
  while VanHely and (not Eof(F)) do begin
    Inc(SorDb);  New(StrKoll[SorDb]);
    Readln(F, StrKoll[SorDb]^);  VanHely:=MaxAvail>SizeOf(String);
  end;
  Close(F);
  StrKollBe:=Eof(f);
end;

procedure StrKollRend(var StrKoll:TSzovMut;SorDb:TSorDb);
var
  I, J, K: TSorI;  MinSor: PString;
begin
  for I:=1 to SorDb-1 do begin
    MinSor:=StrKoll[i];  K:=I;
    for J:=I+1 to SorDb do if StrKoll[J]^<MinSor^ then begin
      MinSor:=StrKoll[J];  K:=J;
    end;
    StrKoll[K]:=StrKoll[I];  StrKoll[I]:=MinSor;
  end;
end;

procedure StrKollKi(Fajl: String; const StrKoll: TSzovMut; SorDb: TSorDb);
var
  F: Text;  I: TSorDb;
begin
  Assign(F, Fajl);  Rewrite(F);
  for I:=1 to SorDb do begin
    Writeln(F, StrKoll[SorDb]^);  Dispose(StrKoll[SorDb]);
  end;
  Close(F);
end;
end.
```

tKom unit

```
unit tKom;

interface

uses Crt, uString, tDef;

{-- Billentyűzetkezelés --}

{ Normál billentyű jelének bekérése visszaírás nélkül }
function JelBe(JoJelek: TJelek): Char;
{ Normál billentyű jelének bekérése visszaírás nélkül a JoJelek elemeinek
  mindkét állása (UpCase, LowCase) elfogadott, az eredmény az UpCase érték
  lesz }
function JelBeUp(JoJelek: TJelek): Char;
{ Normál billentyű jelének bekérése visszaírással, ha a jel kiírható }
function JelBeIr(JoJelek: TJelek): Char;
{ Funkcióbillentyű bekérése, eredmény a második jel }
function F_JelBe(JoJelek: TJelek): Char;
{ Irány vagy kilépés ill. választás bekérése }
function LepesBe: Lepes;

{-- Képernyőre írás --}

{ Általános megjegyzés: ha a sor és/vagy az oszlop paraméter 0, akkor a
  pozíció az aktuális sor és/vagy oszlop lesz }

procedure InverzIr;  { Vilagos háttérrel Sotet írásra állít }
procedure NormalIr;  { Sotet háttérrel Vilagos írásra állít }
procedure Ir(Mit: String; Oszl, Sor: Byte);  { Adott helyre ír }
procedure KozepIr(Mit: String; Sor: Byte);   { Adott sorba középre ír }
{ Ír és cursort visszaállít Oszl+Hossz-ra}
procedure KiIras(S: String; Oszl, Sor, Hossz: Byte; Irmod: IrasMod);

{-- Üzenet és válaszkezelés --}

{ Üzenetet ír a képernyő utolsó sorába, az ezelőtti sorba egy választóvonal
  kerül }
procedure Uzen(Mit: String);
{ Üzenetet ír és Enter után továbblép }
procedure Tovabb(Uzenet: String);
{ Az üzenetre a ValJel valamely jelét várja válaszként, ha az nem üres,
  ha üres, akkor bármely jelet elfogad }
function JelVal(MitUzen: String; ValJel: TJelek): Char;
{ Az üzenet eldüntendő kérdés, 'i' vagy 'I' (=igen) vagy
                               'n' vagy 'N' (=nem) a válasz }
function IgenVal(MitUzen: String): Boolean;

{-- Képernyőkezelés --}

{ Az (O1, S1) (O2, S2) átlós ablakot törli }
procedure AblTorl(O1, S1, O2, S2: Byte);
{ Az S1..S2 sorokat törli }
procedure SorTorl(S1, S2: Byte);
```

```
{ A munkaterület bal felső sarkára áll }
procedure MunkaTerKezd;
{ A fejléc és az üzenetsorok közti sorokat törli, és a munkaterület bal
  felső sarkára áll }
procedure MunkaTerTorl;
{ Munkaterületet állítja be ablaknak }
procedure MunkaAblak;
{ A teljes képernyőt állítja be ablaknak }
procedure TeljAblak;
{ Max. 3 címes fejlécet ír a képernyő tetejére, keretezve az 1..5 sorokba,
  a címeket szimmetrikusan helyezi el csak a kezdő nemüres címeket írja ki }
procedure Fejlec(Cim1, Cim2, Cim3: String);
{ (O1, S1) (O2, S2) átlós téglalapot rajzol }
procedure Keret(O1, S1, O2, S2: Byte);

{ Egy teljes képernyős menü, az első MenuDb menüsor kerül kiajánlásra
  induláskor az első sorra áll, a Menu[0] a címbe kerül
  a visszaadott érték=0, ha nem volt választás egyébként a választott
  sor száma
  a kilépés után a menü törlődik, ha nem volt választás }
function FoMenu(Menu: MenuSorok; MenuDb: MenuInd): MenuInd;

{ A képernyőn az (O1, S1) (O2, S2) ablakban elhelyezkedő menü
  induláskor a Kezd sorra áll a többi paraméter jelentése a FoMenu-vel
  megegyező
  a kilépés után a menü csak akkor törlődik, ha nem volt választás }
function AblMenu(Menu: MenuSorok; MenuDb: MenuInd; var Kezd: MenuInd;
  O1, S1, O2, S2: Byte): MenuInd;

implementation

const
  FejlecSorKezd=1;  FejlecSorVeg=3;
  MunkaSorKezd=FejlecSorVeg+1;
  MunkaSorVeg=23;  UzenetSor=25;

{ Hangjelz,s }
procedure HibaJel;
begin
  Write(#7);
end;

function BillBe(Normal: TJelek; Dupla: TJelek; var Duplae: Boolean): Char;
var
  Jel, Jel1: Char;  VoltJel: Boolean;
begin
  repeat
    Jel:=ReadKey;  Duplae:=Jel=#0;
    if Duplae then Jel1:=ReadKey;
    VoltJel:=(Jel in Normal) or (Duplae and (Jel1 in Dupla));
    if not VoltJel then HibaJel;
  until VoltJel;
  if Duplae then BillBe:=Jel1 else BillBe:=Jel;
end;
```

```
function JelBe(JoJelek: TJelek): Char;
var
  Duplae: Boolean;
begin
  JelBe:=BillBe(JoJelek, [], Duplae);
end;

function JelBeUp(JoJelek: TJelek): Char;
var
  C: Char;
begin
  for C:='a' to 'z' do
    if C in JoJelek then JoJelek:=JoJelek+[UpCase(C)];
  for C:='A' to 'Z' do
    if C in JoJelek then JoJelek:=JoJelek+[LoJel(C)];
  JelBeUp:=UpCase(JelBe(JoJelek));
end;

function JelBeir(JoJelek: TJelek): Char;
var
  C: Char;
begin
  C:=JelBe(JoJelek);
  if C>=' ' then Write(C);
  JelbeIr:=C;
end;

function F_JelBe(JoJelek: TJelek): Char;
var
  Duplae: Boolean;
begin
  F_JelBe:=BillBe([], JoJelek, Duplae);
end;

function LepesBe;
var
  Duplae: Boolean;
begin
  LepesBe:=BillBe([Valaszt, KiLep], Iranyok, Duplae);
end;

procedure InverzIr;
begin
  TextColor(Sotet);  TextBackground(Vilagos);
end;

procedure NormalIr;
begin
  TextColor(Vilagos);  TextBackground(Sotet);
end;

procedure Ir(Mit: String; Oszl, Sor: Byte);
begin
  if Oszl=0 then Oszl:=Wherex;  if Sor=0 then Sor:=Wherey;
  GotoXY(Oszl, Sor);  Write(Mit);
end;
```

```
procedure KozepIr(Mit: String; Sor: Byte);
var
  H: Byte;
begin
  H:=Lo(WindMax)-Lo(WindMin);
  if H-2<Length(Mit) then Mit:=Copy(Mit, 1, H-2);
  Ir(Mit, (H-Length(Mit)+1) div 2+1, Sor);
end;

procedure KiIras(S: String; Oszl, Sor, Hossz: Byte; Irmod: IrasMod);
begin
  case Irmod of
    Normal: NormalIr;
    Inverz: InverzIr;
  end;
  Ir(S, Oszl, Sor);  GotoXY(Oszl+Hossz, Sor);
end;

{ Üzenetsor vonala }
procedure Vonal;
var
  I: Byte;
begin
  GotoXY(1, UzenetSor-1);
  for I:=1 to 80 do Write(#196);
end;

procedure Uzen(Mit: String);
begin
  Vonal; GotoXY(1,UzenetSor); ClrEol;
  Ir(Mit,2,UzenetSor);
end;

procedure Tovabb(Uzenet: String);
var
  C: Char;
begin
  C:=JelVal(Uzenet+' (Tovább: Enter)', [Valaszt]);
end;

function JelVal(MitUzen: String; ValJel: TJelek): Char;
begin
  Uzen(MitUzen);
  if ValJel=[] then JelVal:=ReadKey else JelVal:=JelBeUp(ValJel);
end;

function IgenVal(MitUzen: String): Boolean;
begin
  IgenVal:=JelVal(MitUzen, ['I', 'N'])='I';
end;

procedure AblTorl(O1, S1, O2, S2: Byte);
begin
  Window(O1, S1, O2, S2);  ClrScr;
  TeljAblak;
end;
```

```
procedure SorTorl(S1, S2: Byte);
begin
  AblTorl(1, S1, 80, S2)
end;

procedure MunkaTerKezd;
begin
  GotoXY(1, MunkaSorKezd);
end;

procedure MunkaTerTorl;
begin
  SorTorl(MunkaSorKezd, MunkaSorVeg);
  GotoXY(1, MunkaSorKezd);
end;

procedure MunkaAblak;
begin
  Window(1, MunkaSorKezd, 80, MunkaSorVeg);
end;

procedure TeljAblak;
begin
  Window(1, 1, 80, 25);
end;

procedure Fejlec(Cim1, Cim2, Cim3: String);
var
  Cim: String;
begin
  SorTorl(FejlecSorKezd, FejlecSorVeg);
  Keret(1, FejlecSorKezd, 80, FejlecSorVeg);
  if Cim1='' then Cim:='' else
  if Cim2='' then Cim:=Cim1 else
  if Cim3='' then Cim:=Cim1+'-'+Cim2
  else Cim:=Cim1+'-'+Cim2+'-'+Cim3;
  KozepIr(Cim, FejlecSorKezd+1);
end;

procedure Keret(O1, S1, O2, S2: Byte);
var
  I: Byte;
begin
  if O1>O2 then begin {csere}
    I:=O1;  O1:=O2;  O2:=I;
  end;
  if S1>S2 then begin {csere}
    I:=S1;  S1:=S2;  S2:=I;
  end;
  Ir(#218+JelSor(#196, O2-O1-1)+#191, O1, S1);
  GotoXY(O1, S1+1);
  for I:=S1+1 to S2-1 do Ir(#179, O1, I);
  Ir(#192+JelSor(#196, O2-O1-1)+#217, O1, S2);
  GotoXY(O2, S1+1);
  for I:=S1+1 to S2-1 do Ir(#179, O2, I);
end;
```

```
{ A két menüfüggvény közös része }
function MenuValasz(Menu: MenuSorok; MenuDb: MenuInd; var Kezd: MenuInd;
  O1, S1, O2, S2: Byte; KellKeret: Boolean): MenuInd;
var
  SorTav: 1..2;  I: MenuInd;  Valasz: Lepes;

  procedure MenuSorIr(I: MenuInd);
  begin
    KozepIr(Menu[I], 2+SorTav*I);
  end;

begin
  TeljAblak;  Uzen(MenuInfo);  AblTorl(O1, S1, O2, S2);
  if KellKeret then Keret(O1, S1, O2, S2);
  Window(O1, S1, O2, S2);
  if MenuDb<=(S2-S1-2) div 2 then SorTav:=2 else SorTav:=1;
  InverzIr;  KozepIr(Menu[0], 1);  NormalIr;
  for I:=1 to MenuDb do MenuSorIr(I);
  I:=Kezd;  InverzIr;  MenuSorIr(Kezd);
  repeat
    Valasz:=LepesBe;
    NormalIr;  MenuSorIr(I);
    case Valasz of
      Eleje: I:=1;
      Vege: I:=MenuDb;
      Fel: if I>1 then Dec(I);
      Le: if I<MenuDb then Inc(I);
    end;
    InverzIr;  MenuSorIr(I);
  until Valasz in [Valaszt, KiLep];
  NormalIr;
  if Valasz=Valaszt then MenuValasz:=I
  else begin
    MenuValasz:=0;  ClrScr;
  end;
  Kezd:=I;  TeljAblak;
end;

function FoMenu(Menu: MenuSorok; MenuDb: MenuInd): MenuInd;
var
  Kezd: MenuInd;
begin
  MunkaterTorl;  Kezd:=1;
  FoMenu:=MenuValasz(Menu, MenuDb, Kezd, 1, MunkaSorKezd+2, 80,
    MunkaSorVeg, False);
  MunkaterTorl;
end;

function AblMenu(Menu: MenuSorok; MenuDb: MenuInd; var Kezd: MenuInd;
  O1, S1, O2, S2: Byte): MenuInd;
var
  J: Byte;
begin
  if O1>O2 then begin {csere}
    J:=O1;  O1:=O2;  O2:=J;
  end;
```

```
  if S1>S2 then begin {csere}
    J:=S1;  S1:=S2;  S2:=J;
  end;
  AblMenu:=MenuValasz(Menu, MenuDb, Kezd, O1, S1, O2, S2, True);
end;
end.
```

tListaU unit

```
unit tListaU;

interface

uses uListaD, uListaLD;

function UjLElem1: PLElem1;
function UjLElem2: PLElem2;
function UjLotElem: PLotElem;

implementation

function UjLElem1: PLElem1;
var
  U: PLElem1;
begin
  if MaxAvail>SizeOf(TLElem1) then New(U) else U:=nil;
  UjLElem1:=U;
end;

function UjLElem2: PLElem2;
var
  U: PLElem2;
begin
  if MaxAvail>SizeOf(TLElem2) then New(U) else U:=nil;
  UjLElem2:=U;
end;

function UjLotElem: PLotElem;
var
  U: PLotElem;
begin
  if MaxAvail>SizeOf(TLotElem) then New(U) else U:=nil;
  UjLotElem:=U;
end;
end.
```

tMKollU unit

```
unit tMKollU;

interface

uses uMKollD;

function UjMTetel: PMSor;
```

```
implementation

function UjMTetel: PMSor;
var
  U: PMSor;
begin
  if MaxAvail>SizeOf(TMSor) then New(U) else U:=nil;
  UjMTetel:=U;
end;
end.
```

tSKollU unit

```
unit tSKollU;

interface

uses uSKollD;

function UjSTetel: PSTet;

implementation

function UjSTetel: PSTet;
var
  U: PSTet;
begin
  if MaxAvail>SizeOf(STet) then New(U) else U:=nil;
  UjSTetel:=U;
end;
end.
```

uDef unit

```
unit uDef;

interface

const
  SzamJegyek=['0'..'9'];  NagyBetuk=['A'..'Z'];
  KisBetuk =['a'..'z'];   ENBetuk=['É', 'Ö', 'Ü', 'Ű', 'Ő'];
  EKBetuk=['á', 'é', 'ú', 'ó', 'í', 'ü', 'ö', 'ű', 'ő'];
  Betuk=NagyBetuk+KisBetuk+ENBetuk+EKBetuk;
  AzonJelek=NagyBetuk+KisBetuk+SzamJegyek+['_'];
  KozJelek=[' ', ',', '.', '/', '-'];
  Elojel='-';  TPont='.';

type
  SzamJegy='0'..'9';  NagyBetu='A'..'Z';
  KisBetu='a'..'z';   TJelek=set of Char;
  TEgSzamHossz=0..11; {Longint előjellel}

implementation
end.
```

uDinTomb unit

```
unit uDinTomb;

interface

const
  DEDbMax=32760;  {maximális elemszám (max. 65520 byte lehet a
    helyfoglalás)}

type
  TDElem=Integer;  {elemtípus}
  TDEDb=0..DEDbMax;  {elemszámtípus}
  TDEIndex=1..DEDbMax;  {indextípusok}
  TDEIndex0=0..DEDbMax;
  TDEIndex1=1..DEDbMax+1;
  TDTomb=array[TDEIndex] of TDElem;  {tömb alaptípus}
  PDTomb=^TDTomb;  {tömb mutatótípus}

  {a dinamikus tömb típusa}
  TDinTomb=record
    Elemek: PDTomb; {tömbmutató}
    ADb: TDEDb;  {aktuális elemszám}
    FoglDb: TDEDb;  {aktuálisan használható elemszám}
  end;

{ Inicializálás }
procedure DTIndit(var A: TDinTomb);
{ Aktuális elemszám beállitás }
function DTHossz(var A: TDinTomb; EDb: TDEIndex): Boolean;
{ Aktuális elemszám lekérdezés }
function DTEDb(const A: TDinTomb): TDEDb;
{ (Létező) I. elemnek értékadás }
procedure DTBe(var A: TDinTomb; I: TDEIndex; Ertek: TDElem);
{ (Létező) I. elem értéke }
function DTErt(const A: TDinTomb; I: TDEIndex): TDElem;
{ Zárás }
procedure DTZar(var A: TDinTomb);

implementation

procedure DTIndit(var A:TDinTomb);
begin
  with A do begin
    Elemek:=nil;  ADb:=0;  FoglDb:=0;
  end;
end;

function MemKell(Db:TDEDb):Longint;
begin
  MemKell:=Longint(Db)*SizeOf(Integer);
end;
```

```
function DTHossz(var A: TDinTomb; EDb: TDEIndex): Boolean;
var
  Jo: Boolean;  P: PDTomb;  I: TDEIndex;
begin
  if EDb<=A.FoglDb then begin  {nem kell fizikai bővítés}
    A.ADb:=EDb;  DTHossz:=True;
  end
  else with A do begin  {fizikai bővítés, áthelyezés}
    Jo:=MaxAvail>MemKell(Edb);
    if Jo then begin
      GetMem(P, MemKell(Edb));  {új hely foglalása}
      if Elemek<>nil then begin
       for I:=1 to ADb do P^[I]:=Elemek^[I];  {régi elemek átrakása}
       FreeMem(Elemek, MemKell(FoglDb));  {régi hely felszabadítása}
      end;
      {új állapot beállítása}
      Elemek:=P;  FoglDb:=EDb;  ADb:=EDb;
    end;
    DTHossz:=Jo;
  end;
end;

procedure DTBe(var A: TDinTomb; I: TDEIndex; Ertek: TDElem);
begin
  A.Elemek^[I]:=Ertek;
end;

function DTErt(const A:TDinTomb; I: TDEIndex):TDElem;
begin
  DTErt:=A.Elemek^[I];
end;

function DTEDb(const A: TDinTomb): TDEDb;
begin
  DTEdb:=A.ADb;
end;

procedure DTZar(var A:TDinTomb);
begin
  with A do begin
    if Elemek<>nil then FreeMem(Elemek, MemKell(FoglDb));
    Elemek:=nil;  FoglDb:=0;  ADb:=0;
  end;
end;
end.
```

uElemi unit

```
unit uElemi;

interface

const
  MaxSzam=255;
```

```
type
  Szam=1..MaxSzam;

function LKKT(A, B: Szam): Word;
procedure GyokFelKer(A, B, Eps: Real; var C, H: Real; var Van: Boolean);

procedure TrapezInt(
      A, B:   Real;     {i intervallum kezdő és végpont}
      Eps:    Real;     {i adott pontosság }
  var VEps:   Real;     {o m elért pontosság }
      MaxLep: Word;     {i adott maximális lépésszám}
  var VLep:   Word;     {o, m elért lépésszám}
  var Pontos: Boolean;  {o elért pontosság}
  var T: Real           {o, m utolsó összeg});

{ Példafüggvény a GyokFelKer és TrapezInt szubrutinokhoz }
function F(X: Real): Real;

implementation

function LKKT(A, B: Szam): Word;
var
  I: Szam;
begin
  I:=1;
  while (I*A) mod B<>0 do Inc(I);
  LKKT:=I*A;
end;

procedure GyokFelKer(A, B, Eps: Real; var C, H: Real; var Van: Boolean);
var
  Jo: Boolean;
begin
  Van:=F(A)*F(B)<=0;
  if Van then begin
    repeat
      C:=(A+B)/2;
      Jo:=((B-A)<Eps) and (Abs(F(A))<Eps) and
        (Abs(F(B))<Eps);  {megfelelőség}
      if not Jo then begin  {felezés}
        if F(C)*F(A)<=0 then B:=C  {baloldal jó}
        else A:=C  {jobboldal jó};
      end;
    until Jo;
    H:=B-A;
  end;
end;

procedure TrapezInt(A, B: Real; Eps: Real; var VEps: Real; MaxLep: Word;
  var VLep: Word; var Pontos: Boolean; var T: Real);
var
  I: Word  {ciklusváltozó az összegzéshez};
  K: Word  {részintervallumok száma};
  H: Real  {részintervallumok hossza};
  S: Real  {területösszeg};
  Fa, Fb: Real  {függvényérték a-ban és b-ben};
```

```
begin
  {előkészítés}
  Fa:=F(A);  Fb:=F(B);  K:=1;  VLep:=0;
  T:=(B-A)*(Fa+Fb)/2;
  {iteráció}
  repeat
    {előkészítés}
    S:=T;  H:=(B-A)/(2*K);
    {területösszeg}
    T:=(Fa+Fb)/2;
    for I:=1 to 2*K-1 do T:=T+F(A+I*H);
    T:=T*H;
    {továbbosztás}
    K:=2*K;  VLep:=VLep+1;  VEps:=Abs(T-S);
  until (VEps<Eps) or (VLep>MaxLep);
 {befejezés}
  Pontos:=VEps<Eps;
end;

function F(X: Real): Real;
begin
  F:=X*X*X-2*X*X-1;
end;
end.
```

uGBinFa unit

```
unit uGBinFa;

interface

uses uGrafD, dGrafU, tGrafU;

procedure BinFaInit(var Gyoker: PBinFaPont);  {o a fa gyökérpontja}

function BinFanAdatKeres(
      Gyoker:     PBinFaPont;  {i a fa gyökérpontja}
      Mit:        TBinFaAdat;  {i a keresendő adat}
  var Szulo:      PBinFaPont;  {o a keresett elődjének címe}
  var SzuloBalAg: Boolean):    {o a keresett az előd bal ágán van-e}
                  PBinFaPont;  {o a keresett címe, nil ha nincs}

function BinFanAzonKeres(
  Gyoker: PBinFaPont;   {i a fa gyökérpontja}
  Mit:    TBinFaAzon):  {i a keresendő azonosító}
          PBinFaPont;   {o a keresett címe, nil ha nincs}


function BinFaRa(
  var Gyoker: PBinFaPont;   {i/o a fa gyökérpontja}
      A:      TBinFaAzon;   {i a beszúrandó adat azonosítója}
      Mit:    TBinFaAdat):  {i a beszúrandó adat}
              PBinFaPont;   {o új pont}
```

```
function BinFaRol(
  var Gyoker: PBinFaPont;   {i/o a fa gyökérpontja}
      Mit:    TBinFaAdat):  {i a törlendő adat}
              Boolean;      {o volt-e a törlendö adat}

procedure BinFaListaNo(
  var Lista:  Text;         {o lista szövegfájl}
      Gyoker: PBinFaPont);  {i a fa gyökérpontja}

procedure BinFaListaCsokken(
  var Lista:  Text;         {o lista szövegfájl}
      Gyoker: PBinFaPont);  {i a fa gyökérpontja}

procedure BinFaTorol(var Gyoker: PBinFaPont);  {i/o a fa gyökérpontja}

implementation

procedure BinFaInit(var Gyoker: PBinFaPont);
begin
  Gyoker:=nil;
end;

function BinFanAdatKeres(Gyoker: PBinFaPont; Mit: TBinFaAdat;
  var Szulo: PBinFaPont; var SzuloBalAg: Boolean): PBinFaPont;
var
  Akt: PBinFaPont;
begin
  {keresés}
  Akt:=Gyoker;  Szulo:=nil;
  SzuloBalAg:=False;
  while (Akt<>nil) and (Akt^.Adat<>Mit) do begin
    Szulo:=Akt;
    if Mit<Akt^.Adat then begin
      Akt:=Akt^.BalAg;  SzuloBalAg:=True
    end else begin
      Akt:=Akt^.JobbAg;  SzuloBalAg:=False
    end;
  end;
  {eredmény}
  BinFanAdatKeres:=Akt;
end;

function BinFanAzonKeres(Gyoker: PBinFaPont; Mit: TBinFaAzon): PBinFaPont;
var
  Akt: PBinFaPont;
begin
  Akt:=nil;
  if (Gyoker<>nil) and (Akt=nil) then begin
    if Gyoker^.Azon=Mit then Akt:=Gyoker else Akt:=nil;
    if Akt=nil then Akt:=BinFanAzonKeres(Gyoker^.BalAg, Mit);
    if Akt=nil then Akt:=BinFanAzonKeres(Gyoker^.JobbAg, Mit);
  end;
  BinFanAzonKeres:=Akt;
end;
```

```
function BinFaRa(var Gyoker: PBinFaPont; A :TBinFaAzon;
  Mit: TBinFaAdat): PBinFaPont;
var
  UjPont, Akt, Szulo: PBinFaPont;  UjAzon: Boolean;
begin
  UjAzon:=BinFanAzonKeres(Gyoker, A)=nil;
  if not UjAzon then
    UjPont:=nil
  else begin
    UjPont:=UjBinFaPont;
    if UjPont<>nil then begin
      {feltöltés}
      with UjPont^ do begin
        Azon:=A;  Adat:=Mit;  BalAg:=nil;  JobbAg:=nil
      end;
      {beillesztés}
      if Gyoker=nil then {üres fa}
        Gyoker:=UjPont
      else begin
        {helykeresés}
        Akt:=Gyoker;  Szulo:=nil;
        while Akt<>nil do begin
          Szulo:=Akt;
          if UjPont^.Adat<Akt^.Adat then
            Akt:=Akt^.BalAg
          else Akt:=Akt^.JobbAg;
        end;
        {beillesztés}
        if UjPont^.Adat<Szulo^.Adat then
          Szulo^.BalAg:=UjPont
        else Szulo^.JobbAg:=UjPont;
      end;
    end;
  end;
  BinFaRa:=UjPont;
end;

function BinFaRol(var Gyoker: PBinFaPont; Mit: TBinFaAdat): Boolean;
var
  Akt, Szulo, Munka: PBinFaPont;
  Volt: Boolean;
  SzuloBalAg: Boolean;  {az elöd bal ágán van-e a törlendő}
begin
  Akt:=BinFanAdatKeres(Gyoker, Mit, Szulo, SzuloBalAg);
  Volt:=Akt<>nil;
  if Volt then begin
    {talált, törlés}
    if (Akt^.BalAg=nil) and (Akt^.JobbAg=nil) then begin
      {levéltörlés}
      if Szulo=nil then  {egyben gyökér is}
        Gyoker:=nil else
      if SzuloBalAg then
        Szulo^.BalAg:=nil
      else Szulo^.JobbAg:=nil;
    end else {nem levél}
```

```
    if Akt^.BalAg=nil then begin  {egy létező gyermek, JobbAg}
      if Szulo=nil then  {egyben gyökér is}
        Gyoker:=Akt^.JobbAg else
      if SzuloBalAg then
        Szulo^.BalAg:=Akt^.JobbAg
      else Szulo^.JobbAg:=Akt^.JobbAg;
    end else
    if Akt^.JobbAg=nil then begin  {egy létező gyermek, BalAg}
      if Szulo=nil then  {egyben gyökér is}
        Gyoker:=Akt^.BalAg else
      if SzuloBalAg then
        Szulo^.BalAg:=Akt^.BalAg
      else Szulo^.JobbAg:=Akt^.BalAg;
    end else begin {két létező gyermek}
      {jobb ág beillesztése a balba}
      Munka:=Akt^.BalAg;
      while Munka^.JobbAg<>nil do Munka:=Munka^.JobbAg;
      Munka^.JobbAg:=Akt^.JobbAg;
      {bal ág bekötés}
      if Szulo=nil then {egyben gyökér is}
        Gyoker:=Akt^.BalAg else
      if SzuloBalAg then
        Szulo^.BalAg:=Akt^.BalAg
      else Szulo^.JobbAg:=Akt^.BalAg;
    end;
   {helyfelszabadítás}
   Dispose(Akt);
  end;
  BinFaRol:=Volt;
end;

procedure BinFaListaNo(var Lista: Text; Gyoker: PBinFaPont);
begin
  if Gyoker<>nil then begin
    BinFaListaNo(Lista, Gyoker^.BalAg);
    Writeln(Lista, Gyoker^.Azon, '-', Gyoker^.Adat);
    BinFaListaNo(Lista, Gyoker^.JobbAg);
  end;
end;

procedure BinFaListaCsokken(var Lista: Text; Gyoker: PBinFaPont);
begin
  if Gyoker<>nil then begin
    BinFaListaCsokken(Lista, Gyoker^.JobbAg);
    Writeln(Lista, Gyoker^.Azon, '-', Gyoker^.Adat);
    BinFaListaCsokken(Lista, Gyoker^.BalAg);
  end;
end;

procedure BinFaTorol(var Gyoker: PBinFaPont);
begin
  while Gyoker<>nil do BinFaRol(Gyoker, Gyoker^.Adat);
end;
end.
```

uGKupac unit

```
unit uGKupac;

interface

uses uGrafD;

{ Kupacindex }
function KupacIndex(
        Y: TGPontI;     {i a keresendő pont}
        N: TGPontDb;    {i a kupac elemszáma}
  const A: TGPontITmb): {i a kupac pontjai}
           TGPontI0;    {o a kupacbeli index, vagy 0}

{ Beszúrás }
procedure KupacBa(
        Y: TGPontI;     {i a felveendő pont}
  var   N: TGPontDb;    {i/o a kupac elemszáma}
  var   A: TGPontITmb;  {i/o a kupac pontjai}
  const T: TGTavTmb);   {i a pontok távolságai}

{ Minimális elem törlése }
procedure KupacBol(
  var   N: TGPontDb;    {i/o a kupac elemszáma}
  var   A: TGPontITmb;  {i/o a kupac pontjai}
  const T: TGTavTmb);   {i a pontok távolságai}

{ Átsorolás, csökkent távolságértékkel }
procedure KupacFel(
        Y: TGPontI;     {i   az átsorolandó pont}
        N: TGPontDb;    {i a kupac elemszáma}
  var   A: TGPontITmb;  {i/o a kupac pontjai}
  const T: TGTavTmb);   {i a pontok távolságai}

{ Rendezés kupaccal }
procedure KupacRend(
  var   V: TGPontITmb;  {i/o a rendezendő ponttömb}
        M: TGPontDb;    {i a rendezendő tömb elemszáma}
  const T: TGTavTmb);   {i a pontok távolságai}

implementation

function KupacIndex(Y: TGPontI; N: TGPontDb; const A: TGPontITmb): TGPontI0;
var
  I: TGPontI1;
begin
  I:=1;
  while (I<=N) and (A[I]<>Y) do Inc(I);
  if (I<=N) and (A[I]=Y) then KupacIndex:=I else KupacIndex:=0;
end;

procedure KupacBa(Y: TGPontI; var N: TGPontDb; var A: TGPontITmb;
  const T: TGTavTmb);
var
  I: TGPontI;  J: TGPontI0;  Vege: Boolean;
```

```
begin
  { Y a kupac végére }
  Inc(N);  A[N]:=Y;
  {helyére léptetés}
  I:=N;  J:=I div 2;  Vege:=False;
  while (J>=1) and not Vege do
    if T[Y]<T[A[J]] then begin
      A[I]:=A[J];  I:=J;  J:=I div 2;
    end else Vege:=True;
  A[I]:=y;
end;

procedure KupacBol(var N: TGPontDb; var A: TGPontITmb; const T: TGTavTmb);
var
  I1, I2, I2b, I2j, Cs: TGPontI;
begin
  A[1]:=A[N];  I1:=1;  I2b:=2*I1;
  I2j:=I2b+1;  {első elágazás}
  while (I2b<N) and (T[A[I1]]>T[A[I2b]]) or (I2j<N)
    and (T[A[I1]]>T[A[I2j]]) do
  begin
    if (I2b<N) and (I2j<N) {két utód} then begin
      if T[A[I2j]]<T[A[I2b]] then I2:=I2j else I2:=I2b;
    end else {egy utód} if (I2b<N) then
      I2:=I2b
    else I2:=I2j;
    {csere}
    Cs:=A[I1];  A[I1]:=A[I2];  A[I2]:=Cs;
    {új elágazás}
    I1:=I2;  I2b:=2*I1;  I2j:=I2b+1;
  end;
  Dec(N);
end;

procedure KupacFel(Y: TGPontI; N: TGPontDb; var A: TGPontITmb;
  const T: TGTavTmb);
var
  I, X: TGPontI;  J: TGPontI0;  Vege: Boolean;
begin
  I:=KupacIndex(Y, N, A);
  X:=A[I];  J:=I div 2;  Vege:=False;
  while (J>=1) and not Vege do  {helyére léptetés}
    if T[X]<T[A[J]] then begin
      A[I]:=A[J];  I:=J;  J:=I div 2;
    end else Vege:=True;
  A[I]:=X;
end;

procedure KupacRend(var V: TGPontITmb; M: TGPontDb; const T: TGTavTmb);
var
  A: TGPontITmb;  {a kupac}
  N: TGPontDb;  I: TGPontI0;
begin
  N:=0;
  for I:=1 to M do KupacBa(V[i], N, A, T);
  I:=0;
```

```
  while N>0 do begin
    Inc(I);  V[I]:=A[1];
    KupacBol(N, A, T)
  end;
end;
end.
```

uGrafD unit

```
unit uGrafD;

interface

{-- általános gráfok --}

{gráf alapdeklarációk}
const
  GMinPontAz='a';  GMaxPontAz='z';
  GMaxPontDb=Ord(GMaxPontAz)-Ord(GMinPontAz)+1;
  GMaxElDb=GMaxPontDb*(GMaxPontDb-1);
  GMaxPontTip=6;
  GMaxElKat=9;
  GMaxElHossz=999;

type
  TGPontAz=GMinPontAz..GMaxPontAz;  {pontazonosító}
  TGKoord=Integer;                  {koordináta pontjellemző}
  TGPontTip=1..GMaxPontTip;         {ponttípus pontjellemző}
  TGElKat=1..GMaxElKat;             {kategória éljellemző}
  TGElHossz=-1..GMaxElHossz;        {hossz éljellemző, a -1 a nemlétező él
                                       hossza}

  TGPont=record  {pontadat: azonosító, típus, koordináta}
    Azon: TGPontAz;
    Tip: TGPontTip;
    X, Y: TGKoord;
  end;

  TGPontDb=0..GMaxPontDb;             {pontok száma}
  TGPontI =1..GMaxPontDb;             {pontindex}
  TGPontI0=0..GMaxPontDb;             {bővített pontindex}
  TGPontI01= 0..GMaxPontDb+1;         {bővített pontindex}
  TGPontok=array[TGPontI] of TGPont;  {pontok adatai}

  {gráf mátrixtárolással}
  TGElHosszMtx=array[TGPontI, TGPontI] of TGElHossz;  {hossz-mátrix}
  TGElKatMtx=array[TGPontI, TGPontI] of TGElKat;      {kategória-mátrix}

  TGrafMtx=record
    PontDb: TGPontDb;       {pontok száma}
    Pontok: TGPontok;       {azonosító szerint növekvően rendezettek}
    ElHossz: TGElHosszMtx;  {hossz-mátrix, -1 hossz esetén nincs él}
    ElKat: TGElKatMtx;      {kategória-mátrix}
  end;
```

```
  {gráf éltárolással}
  TGEl=record  {éladat: végpontindex, hossz, kategória}
    VpInd: TGPontI;
    ElHossz: TGElHossz;
    ElKat: TGElKat;
  end;

  TGElDb=0..GMaxElDb;           { élek száma}
  TGElI=1..GMaxElDb;            { élindex}
  TGElI01=0..GMaxElDb+1;      { bővitett élindex}
  TGElek=array[TGElI] of TGEl;  {élek adatai}

  {élmutatók tömbje}
  TGPontI1=1..GMaxPontDb+1;         {index}
  TGElI1=1..GMaxElDb+1;             {elem}
  TGElMut=array[TGPontI1] of TGElI1;  {tömb}

  TGrafElT=record
    PontDb: TGPontDb;  {pontok száma}
    Pontok: TGPontok;  {azonosító szerint növekvően rendezettek}
    ElMut: TGElMut;    {az élmutatók}
    ElDb: TGElDb;      {élek száma}
    Elek: TGElek;      {egy pont élei végpont szerint rendezettek}
  end;

{-- bináris fa mint dinamikus adatszerkezet --}

type
  TBinFaAzon=String;
  TBinFaAdat=String;
  PBinFaPont=^TBinFaPont;
  TBinFaPont=record           {pont}
    Azon: TBinFaAzon;           {pontazonosító}
    Adat: TBinFaAdat;           {pontjellemző}
    BalAg, JobbAg: PBinFaPont;  {az elágazások mutatói}
  end;

{-- kupachoz és utak kereséséhez --}

const
  GMaxUtHossz=GMaxElDb*GMaxElHossz; {maximális úthossz körmentes utakra}
  GVegtelen=GMaxUtHossz+1;          {technikailag szükséges adat}

type
  TGUtHossz=0..GVegtelen;                {úthossz}
  TGPontITmb=array[TGPontI] of TGPontI;  {pontindextömb}
  TGTavTmb=array[TGPontI] of TGUtHossz;  {távolságtömb}

implementation
end.
```

uGrafFa unit

```
unit uGrafFa;

interface

uses uGrafD;

type
  {egy él léte az élhalmazokban}
  TGElVanMtx=array[TGPontI, TGPontI] of Boolean;
  {egy pont helye a PHH-ban (PHH ilyen sorszámú halmazában van az adott
   pont)}
  TGPontHely=TGPontI;  {mivel a PHH maximális elemszáma is GMaxPontDb}
  TGPontHelyek=array[TGPontI] of TGPontI;

{ A minimális feszítőfa meghatározása }
function MinFFaMtx(const Graf: TGrafMtx; var MFFa: TGrafMtx): Boolean;
{ A gráf komponensekre bontása }
procedure KompBontMtx(const Graf: TGrafMtx; var PHH: TGPontHelyek;
  var HDb: TGPontDb);

implementation

function MinFFaMtx(const Graf: TGrafMtx; var MFFa: TGrafMtx): Boolean;
var
  HDb, X, Y, I, J: TGPontI;  EDb: TGElDb;  EH: TGElVanMtx;
  PHH: TGPontHelyek;  A, B: TGPontHely;  Min: TGUtHossz;
begin
  with Graf do begin
    { Kezdőállapot }
    { EH-be minden élt }
    EDb:=0;
    for I:=1 to PontDb do
      for J:=1 to PontDb do EH[I, J]:=ElHossz[I, J]>=0;
    { FH legyen üres }
    MFFa:=Graf;
    for I:=1 to PontDb do
      for J:=1 to PontDb do MFFa.ElHossz[I, J]:=-1;
    { PHH-ba minden pontot mint egyelemű részhalmazokat }
    for I:=1 to PontDb do PHH[I]:=I;
    HDb:=PontDb;
    { Javító lépések }
    while (HDB>1) and (EDb>0) do begin
      {minimális hosszú EH -bani él kiválasztása és törlése EH-ból}
      Min:=GVegtelen;
      for I:=1 to PontDb do for j:=1 to PontDb do
        if EH[I, J] and (ElHossz[I, J]<Min) then begin
          X:=I;  Y:=J;
          Min:=ElHossz[I, J];
        end;
      EH[X, Y]:=False;  Dec(EDb);
      if PHH[X]<>PHH[Y] then begin
        { különböző halmazba esnek, egyesítünk }
        { a kisebb sorszámú halmazba (A) tesszük valamennyit, }
        { a nagyobb sorszámú halmazt (B) pedig töröljük az utolsó }
```

```
        { ponthalmaz elemeinek ebbe a halmazba való átsorolásával }
        if PHH[X]<PHH[Y] then begin
          A:=PHH[X];  B:=PHH[Y]
        end else begin
          A:=PHH[Y];  B:=PHH[X]
        end;
        for I:=1 to PontDb do
          if PHH[I]=B then
            PHH[I]:=A
          else if PHH[I]=HDb then PHH[I]:=B;
        Dec(HDb);
        { XY élt FH-be }
        MFFa.ElHossz[X, Y]:=Graf.ElHossz[X, Y];
      end;
    end;
    MinFFaMtx:=HDb=1;
  end;
end;

procedure KompBontMtx(const Graf: TGrafMtx; var PHH: TGPontHelyek;
  var HDb: TGPontDb);
var
  X, Y, I, J: TGPontI;  EDb: TGElDb;  EH: TGElVanMtx;
  A, B: TGPontHely;  VanEl: Boolean;
begin
  with Graf do begin
    EDb:=0;
    for I:=1 to PontDb do
      for J:=1 to PontDb do EH[I, J]:=ElHossz[I, J]>=0;
    for I:=1 to PontDb do PHH[I]:=I;
    HDb:=PontDb;
    while (HDB>1) and (EDb>0) do begin
      {egy EH-bani él kiválasztása és törlése EH-ból}
      VanEl:=False;
      for I:=1 to PontDb do for J:=1 to PontDb do
        if not VanEl and EH[I, J] then begin
          X:=I;  Y:=J;
          VanEl:=True;
        end;
      EH[X, Y]:=False;  Dec(EDb);
      if PHH[X]<>PHH[Y] then begin
        if PHH[X]<PHH[Y] then begin
          A:=PHH[X];  B:=PHH[Y]
        end else begin
          A:=PHH[Y];  B:=PHH[X]
        end;
        for I:=1 to PontDb do
          if PHH[I]=B then
            PHH[I]:=A
          else if PHH[I]=HDb then PHH[I]:=B;
        Dec(HDb);
      end;
    end;
  end;
end;
end.
```

uGrafKez unit

```
unit uGrafKez;

interface

uses uGrafD;

{ Pont (vagy a helyének) keresése azonosító alapján }
function PontIndKeres(PontDb: TGPontDb; const Pontok: TGPontok;
  PontAzon: TGPontAz; var Van: Boolean): TGPontI;
{Pont felvétele a gráfba (éltárolás)}
procedure PontFelveszElT(var Graf: TGrafElT; Pont: TGPont);
{ Pont törlése a gráfból az éleivel együtt (mátrix) }
procedure PontTorolMtx(var Graf: TGrafMtx; PontInd: TGPontI);
{ Él (vagy a helyének) keresése pontindexek alapján (éltárolás) }
function ElKeresElT(const Graf: TGrafElT; KpInd, VpInd: TGPontI;
  var Van: Boolean): TGElI;
{ Él felvétele (éltárolás) }
procedure ElFelveszElt(var Graf: TGrafElT; KpInd: TGPontI; const El: TGEl);
{ Mentés szövegfájlba }
procedure MentesElt(const Graf: TGrafElT; var F: Text);
{ Betöltés szövegfájlból }
procedure BetoltesMtx(var Graf: TGrafMtx; var F: Text);

implementation

function PontIndKeres(PontDb: TGPontDb; const Pontok: TGPontok;
  PontAzon: TGPontAz; var Van: Boolean): TGPontI;
var
  E, V, Hol: TGPontDb;
begin
  E:=1;  V:=PontDb;  Van:=False;
  while (E<=V) and not Van do begin
    Hol:=(E+V) div 2;
    Van:=Pontok[Hol].Azon=PontAzon;
    if not Van then
      if Pontok[Hol].Azon<PontAzon then
        E:=Hol+1
      else V:=Hol-1;
  end;
  if not Van then Hol:=E;
  PontIndKeres:=Hol;
end;

procedure PontFelveszElT(var Graf:TGrafElT; Pont:TGPont);
var
  PontInd: TGPontI;  I: TGPontI01;  Van: Boolean;  L: TGElDb;
begin
  with Graf do begin
    { a pont helyének megkeresése }
    PontInd:=PontIndKeres(Pontdb, Pontok, Pont.Azon, Van);
    if Van then
      Pontok[PontInd]:=Pont
    else begin
```

```
      { a nagyobb vagy egyenlő pontindexek léptetése}
      for L:=1 to ElDb do
        with Elek[L] do if VpInd>=PontInd then Inc(VpInd);
      {élmutató}
      for I:=PontDb+1 downto PontInd do ElMut[I+1]:=ElMut[I];
      { ezután ElMut[PontInd]=ElMut[PontInd+1] lesz, ami megfelel annak,
        hogy az új pontnak még nincs éle }
      { pontfelvétel }
      for I:=PontDb downto PontInd do  Pontok[I+1]:=Pontok[I];
      Pontok[PontInd]:=Pont;
      Inc(PontDb);
    end;
  end;
end;

procedure PontTorolMtx(var Graf:TGrafMtx; PontInd:TGPontI);
var
  I, J: TGPontI1;
begin
  with Graf do begin
    for I:=PontInd+1 to PontDb do {sorok}
      for J:=1 to PontDb do begin
        ElHossz[I-1, J]:=ElHossz[I, J];
        ElKat[I-1, J]:=ElKat[I, J];
      end;
    for J:=PontInd+1 to PontDb do {oszlopok}
      for i:=1 to PontDb do begin
        ElHossz[I, J-1]:=ElHossz[I, J];
        ElKat[I, J-1]:=ElKat[I, J];
      end;
    { pont }
    for I:=PontInd+1 to PontDb do Pontok[I-1]:=Pontok[I];
    Dec(PontDb);
  end;
end;

function ElKeresElT(const Graf: TGrafElT; KpInd, VpInd: TGPontI;
  var Van: Boolean): TGElI;
var
  E, V, Hol: TGElDb;
begin
  with Graf do begin
    E:=ElMut[KpInd];  V:=ElMut[KpInd+1]-1;
    Van:=False;
    while (E<=V) and not Van do begin
      Hol:=(E+V) div 2;
      Van:=Elek[Hol].VpInd=VpInd;
      if not Van then
        if Elek[Hol].VpInd<VpInd then E:=Hol+1 else V:=Hol-1;
    end;
    if not Van then Hol:=E;
  end;
  ElKeresElt:=Hol;
end;
```

```
procedure ElFelveszElt(var Graf: TGrafElT; KpInd: TGPontI; const El: TGEl);
var
  ElInd: TGElI;  L: TGElDb;  I: TGPontI1;  Van: Boolean;
begin
  with Graf do begin
    ElInd:=ElKeresElt(Graf, KpInd, El.VpInd, Van);  {az él helye}
    if Van then
      Elek[Elind]:=El
    else begin
      for I:=KpInd+1 to PontDb+1 do Inc(ElMut[I]);  {élmutatók növelése}
      for L:=ElDb downto ElInd do Elek[L+1]:=Elek[L];  {helykészítés}
      Elek[ElInd]:=El;  {felvétel}
      Inc(ElDb);
    end;
  end;
end;

procedure MentesElt(const Graf: TGrafElT; var F: Text);
var
  I: TGPontDb;  L: TGElDb;
begin
  Rewrite(F);
  with Graf do begin
    { pontok }
    WriteLn(F, PontDb);
    for I:=1 to PontDb do
      with Pontok[I] do WriteLn(F, Azon, Tip:3, X:13, Y:13);
    { élek }
    WriteLn(F, ElDb);
    for I:=1 to PontDb do
      for L:=ElMut[I] to ElMut[I+1]-1 do with Elek[L] do
        WriteLn(F, Pontok[I].Azon, ' ', Pontok[VpInd].Azon, ElHossz:6,
          ElKat:3);
  end;
  Close(F);
end;

procedure BetoltesMtx(var Graf: TGrafMtx; var F: Text);
var
  I, J: TGPontI01;  Van: Boolean;  Kezdo, Veg: TGPontAz;
  H: TGElHossz;  Kat: TgElKat;  L, Db: TGElDb;
  C: Char;  {az azonosítók közötti szóköz átlépéséhez}
begin
  Reset(F);
  with Graf do begin
    { init }
    for I:=1 to GMaxPontDb do
      for J:=1 to GMaxPontDb do begin
        ElHossz[I, J]:=-1;  ElKat[I, J]:=1;
      end;
    { pontok }
    ReadLn(F, PontDb);
    for I:=1 to PontDb do
      with Pontok[I] do ReadLn(F, Azon, Tip, X, Y);
    { élek }
    ReadLn(F, Db);
```

```
    for L:=1 to Db do begin
      ReadLn(F, Kezdo, C, Veg, H, Kat);
      I:=PontIndKeres(Pontdb, Pontok, Kezdo, Van);
      J:=PontIndKeres(Pontdb, Pontok, Veg, Van);
      ElHossz[I, J]:=H;
      ElKat[I, J]:=Kat;
    end;
    Close(F);
  end;
end;
end.
```

uGrafUt unit

```
unit uGrafUt;

interface

uses uGrafD, uGKupac;

{mátrix módszer}
type
  TGTavMtx=array[TGPontI, TGPontI] of TGUtHossz;  {távolságmátrix}
  TGCimkeMtx=array[TGPontI, TGPontI] of TGPontI;  {címkemátrix}

procedure OsszMinutMtx(const Graf: TGrafMtx; var TT: TGTavMtx;
  var CC: TGCimkeMtx);

{faépítés}
type
  TGCimkeTmb=TGPontITmb;
  TGAktivJel=array[TGPontI] of Boolean;  {pontaktivitás jelző}

procedure MinfaTmbElt(const Graf: TGrafElt; var T: TGTavTmb;
  var C: TGCimkeTmb; KezdoPont: TGPontI);
procedure MinfaKupElt(const Graf: TGrafElt; var T: TGTavTmb;
  var C: TGCimkeTmb; KezdoPont: TGPontI);

implementation

procedure OsszMinutMtx(const Graf: TGrafMtx; var TT: TGTavMtx;
  var CC: TGCimkeMtx);
var
  X, Y, W: TGPontI;
begin
  with Graf do begin
    { kezdőállapot }
    for X:=1 to PontDb do for Y:=1 to PontDb do begin
      if ElHossz[X, Y]>=0 then
        TT[X, Y]:=ElHossz[X, Y]
      else TT[X, Y]:=GVegtelen;
      CC[X, Y]:=Y;
    end;
    { javító lépések }
    for W:=1 to PontDb do for X:=1 to PontDb do
      for Y:=1 to PontDb do
```

```
        if (TT[X, W]<GVegtelen) and (TT[W, Y]<GVegtelen) and
          (TT[X, W]+TT[W, Y]<TT[X, Y]) then
        begin
          TT[X, Y]:=TT[X, W]+TT[W, Y];
          CC[X, Y]:=CC[X, W];
        end;
  end;
end;

procedure MinfaTmbElt(const Graf: TGrafElt; var T: TGTavTmb;
  var C: TGCimkeTmb; KezdoPont: TGPontI);
var
  ATmb: TGPontITmb;  { az aktív pontok indexei, tömb}
  AktJel: TGAktivJel;  ADb, J: TGPontDb;
  X, Y, I: TGPontI;  L: TGElDb;
begin
  with Graf do begin
    { kezdőállapot }
    for I:=1 to PontDb do begin
      T[I]:=GVegtelen;  AktJel[I]:=False;
    end;
    T[KezdoPont]:=0;  C[KezdoPont]:=KezdoPont;
    ADb:=1;
    ATmb[ADb]:=KezdoPont;  AktJel[KezdoPont]:=True;
    { javító lépések }
    while ADb>0 do begin  { van még aktív pont }
      X:=ATmb[1];  { az első aktív pont a minimális távolságú }
      { X törlése az aktív pontok közül előreléptetéssel }
      for I:=2 to ADb do ATmb[I-1]:=ATmb[I];
      Dec(ADb);  AktJel[X]:=False;
      { 'rövidítés' x-en keresztül }
      for L:=ElMut[X] to ElMut[X+1]-1 do with Elek[L] do begin
        Y:=VpInd;
        if T[X]+ElHossz< T[Y] then begin
          T[Y]:=T[X]+ElHossz;  C[Y]:=X;
          if not AktJel[Y] then begin
            { Y még nem aktív, besoroljuk az aktív pontok közé távolság
              szerint }
            J:=ADb;
            while (J>=1) and (T[Y]<T[ATmb[J]]) do begin
              ATmb[J+1]:=ATmb[J];  Dec(J);
            end;
            ATmb[J+1]:=Y;  Inc(ADb);  AktJel[Y]:=True;
          end else begin
            { Az aktív pontok között lévő Y távolsága csökkent, ezért }
            { helyére kell tennünk a távolság szerint rendezett aktív }
            { pontok ATmb tömbjében.}
            J:=ADb;
            while (J>=1) and (ATmb[J]<>Y) do Dec(J); { ráállunk y-ra }
            if J>1 then begin
              Dec(J);
              while (J>=1) and (T[ATmb[J]]>T[Y]) do begin
                ATmb[J+1]:=ATmb[J];  Dec(J);
              end;
              ATmb[J+1]:=Y;
            end;
```

```
          end;
        end;
      end;
    end;
  end;
end;

procedure MinfaKupElt(const Graf: TGrafElt; var T: TGTavTmb;
  var C: TGCimkeTmb; KezdoPont: TGPontI);
var
  AKup: TGPontITmb;  { az aktív pontok indexei, kupac}
  AktJel: TGAktivJel;  ADb: TGPontDb;  X, Y, I: TGPontI;  L: TGElDb;
begin
  with Graf do begin
    { kezdőállapot }
    for I:=1 to PontDb do begin
      T[I]:=GVegtelen;  AktJel[I]:=False;
    end;
    T[KezdoPont]:=0;  C[KezdoPont]:=KezdoPont;
    ADb:=1;
    AKup[ADb]:=KezdoPont;  AktJel[KezdoPont]:=True;
    { javító lépések }
    while ADb>0 do begin
      X:=AKup[1];
      AktJel[X]:=False;  { az első pont a minimális távolságú }
      KupacBol(ADb, AKup, T);
      { 'rövidítés' X-en keresztül }
      for L:=ElMut[X] to ElMut[X+1]-1 do with Elek[L] do begin
        Y:=VpInd;
        if T[X]+ElHossz<T[Y] then begin
          T[Y]:=T[X]+ElHossz;  C[Y]:=X;
          if not AktJel[Y] then begin
           { Y aktív lesz és felvesszük a kupacba }
            KupacBa(Y, ADb, AKup, T);  AktJel[Y]:=True;
          end else
           { az aktív pontok között lévő Y távolsága csökkent, ezért }
           { helyére kell tennünk a távolság szerint kupacot alkotó }
           { aktív pontok AKup tömbjében }
           KupacFel(Y, ADb, AKup, T);
        end;
      end;
    end;
  end;
end;
end.
```

uHalm unit

```
unit uHalm;

interface

const
  HElemMaxDb=256;
```

```
type
  THElem=Byte;
  THalmaz=set of THElem;
  THElemDb=0..HElemMaxDb;

{ Keressük meg egy THalmaz minimális és maximális elemét }
function MinMax(
      Adat: THalmaz;  {i ebben keresünk}
  var Min: THElem;    {o minimális THElem}
  var Max: THElem):   {o maximális THElem}
           Boolean;   {a fügvényérték jelzi, hogy van-e Min és Max}

{ Határozzuk meg egy THalmaz elemszámát }
function Szamossag(
  Adat: THalmaz):  {i a halmaz}
        THElemDb;  {o az Adat elemszáma}

{ Válasszuk ki egy nemüres THalmaz egy elemét véletlenszerűen
  Csak nemüres halmazzal hívható, egyébként definiálatlan az eredmény! }
function EgyElem(Adat: THalmaz): THElem;

{ Állítsuk elő az alaptípus egy, két előírt határ közé eső értékét
  véletlenszerűen }
function EgyErtek(
  EMin, EMax: THElem): {i a  minimum ill. maximum}
              THElem;  {o eredmény}

{ Állítsuk elő az alaptípus minden, az adott intervallumba eső elemét
  tartalmazó halmazt }
procedure IntHalmaz(
      EMin, EMax: THElem;    {i a  minimum ill. maximum}
  var Adat:       THalmaz);  {o eredmény}

{ Állítsuk elő az alaptípus minden elemét tartalmazó halmazt }
procedure TeljHalmaz(var Adat: THalmaz);  {o eredmény}

type
  THTomb=array[THElem] of THElem;

procedure HElemek(
  const Adat: THalmaz;    {i a halmaz}
  var   T:    THTomb;     {o a tömb}
  var   N:    THElemdb);  {o a tömb elemszáma}

{ Állítsunk elő egy előírt elemszámú és előírt értékhatárok közé eső
  elemeket tartalmazó halmazt véletlenszerű elemekkel
  ha a az előírt elemszám nagyobb lenne a lehetségesnél akkor az eredmény
  az előírt értékhatárok közé eső elemeket tartalmazó halmaz lesz }
procedure HSorsol(
      Szam:       THElemdb;  {i az előírt elemszám}
      EMin, EMax: THElem;    {i az előírt minimum ill. maximum}
  var Adat:       THalmaz);  {o az eredmény}
```

```
{ Egy halmazt osszunk véletlenszerűen két részre úgy, hogy
 a két fél elemszáma közt max. 1 lehet a különbség }
procedure HEloszt(
      Adat:  THalmaz;   {i ezt osztjuk}
  var Adat1: THalmaz;   {o egyik fél}
  var Adat2: THalmaz);  {o másik fél}

{ 5-ös lottó sorsolás, közvetett megoldás }
procedure Lotto5a(
  var Huzas: THTomb;     {o az első 5 elem az eredmény}
  var N:     THElemDb);  {o =5}

implementation

function MinMax(Adat: THalmaz; var Min: THElem; var Max: THElem): Boolean;
var
  Ures: Boolean;
begin
  Ures:=Adat=[];
  if not Ures then begin
    Min:=Low(THElem);  while not (Min in Adat) do Inc(Min);
    Max:=High(THElem);  while not (Max in Adat) do Dec(Max);
  end;
  MinMax:=not Ures;
end;

function Szamossag(Adat: THalmaz): THElemDb;
var
  Szam: THElemDb;  Min, Max, X: THElem;
begin
  Szam:=0;
  if MinMax(Adat, Min, Max) then
    for X:=Min to Max do if X in Adat then Inc(Szam);
  Szamossag:=Szam;
end;

function EgyElem(Adat:THalmaz):THElem;
var
  X, Min, Max: THElem;
begin
  if MinMax(Adat,Min,Max) then begin
    repeat
      X:=THElem(Random(Ord(Max)-Ord(Min)+1)+Ord(Min));
    until X in Adat;
    EgyElem:=X;
  end;
end;

function EgyErtek(EMin, EMax: THElem): THElem;
begin
  EgyErtek:=THElem(Random(Ord(EMax)-Ord(EMin)+1)+Ord(EMin));
end;

procedure IntHalmaz(EMin, EMax: THElem; var Adat: THalmaz);
var
  I: THElem;
```

```
begin
  Adat:=[];
  for I:=EMin to EMax do Adat:=Adat+[I];
end;

procedure TeljHalmaz(var Adat: THalmaz);
begin
  IntHalmaz(Low(THElem), High(THElem), Adat);
end;

procedure HElemek(const Adat: THalmaz; var T: THTomb; var N: THElemdb);
var
  Min, Max, X: THElem;
begin
  N:=0;
  if MinMax(Adat, Min, Max) then
    for X:=Min to Max do
      if X in Adat then begin
        Inc(N);
        T[N]:=X;
      end;
end;

procedure HSorsol(Szam: THElemdb; EMin, EMax: THElem; var Adat: THalmaz);
var
  I: THElemDb;  X: THElem;
begin
  if Szam<=(Ord(EMax)-Ord(EMin)) div 2 then begin
    {ez esetben a beveendő elemeket generáljuk}
    Adat:=[];  I:=0;
    while I<Szam do begin
      X:=EgyErtek(EMin, EMax);
      if not (X in Adat) then begin
        Adat:=Adat+[X];  Inc(I);
      end;
    end;
  end else begin
    {ez esetben a kimaradó elemeket generáljuk}
    IntHalmaz(EMin, Emax, Adat);
    I:=Szamossag(Adat);
    while I>Szam do begin
      X:=EgyErtek(EMin, EMax);
      if X in Adat then begin
        Adat:=Adat-[X];  Dec(I);
      end;
    end;
  end;
end;

procedure HEloszt(Adat: THalmaz; var Adat1: THalmaz; var Adat2: THalmaz);
var
  X: THElem;  I: THElemDb;
begin
  Adat1:=Adat;  Adat2:=[];
  if Adat<>[] then
    for I:=1 to Szamossag(Adat) div 2 do begin
```

```
      X:=EgyElem(Adat1);
      Adat1:=Adat1-[X];  Adat2:=Adat2+[X];
    end;
end;

procedure Lotto5a(var Huzas: THTomb; var N: THElemDb);
var
  H: THalmaz;
begin
  HSorsol(5, 1, 90, H);  HElemek(H, Huzas, N);
end;
end.
```

uHalmAlk unit

```
unit uHalmAlk;

interface

uses uTomb, uNHalm;

{ Egy tömb olyan egész (Integer) számokat tartalmaz, amelyek között
  a maximális eltérés kisebb mint 16384. Rendezzük az elemeket növekvő
  sorrendbe az ismétlődések törlésével }
procedure HRendez0(
  var T: TSor;      {i/o ezt rendezzük}
  var N: TElemDb);  {i/o a T elemszáma}

{ Rendezzük az elemeket növekvő sorrendbe az ismétlődések megőrzésével }
procedure HRendez1(
  var T: TSor;     {i/o ezt rendezzük}
      N: TElemDb); {i a T elemszáma}

{ Állítsuk elő sorsolással az 5-ös lottó (5 a 90-ből) egy húzási eredményét,
  növekvő számsorrendben }
type
  TL5Szam=1..90;
  TL5Ind=1..5;
  TL5Db=0..5;
  TL5Huzas=array[TL5Ind] of TL5Szam;

procedure Lotto5b(var Huzas: TL5Huzas);  {o a húzási eredmény}

{ Egy adatsor évszámokat tartalmaz. Állapítsuk meg, hogy van-e benne
  ismétlődés }
function Ismetele(const A: TSor; N: TElemDb): Boolean;

{ Állítsuk elő egy stringben lévő jelek érték szerint növekvően rendezett
  gyakorisági statisztikáját}
type
  TJelGyak=array[Char] of Byte;

procedure JelGyak(S: String; var Gyak: TJelGyak);

implementation
```

```
procedure HRendez0(var T: TSor; var N: TElemDb);
var
  H: TNHalmaz;  I: TEIndex;  J: TElemDb;  E, Min, Max: TNHElem;
begin
  if N>0 then begin
    NHUres(H);
    Min:=MaxNHElem;  Max:=MinNHElem;
    {az elemeket transzformáltan felvesszük a tömbből halmazba,
     közben minimumot és maximumot számítunk}
    for I:=1 to N do begin
      E:=T[I]+MinNHElem;  NHba(E, H);
      if E<Min then Min:=E;
      if E>Max then Max:=E;
    end;
    J:=0;
    {az elemeket a halmazból transzformáltan visszaírjuk a tömbbe}
    for E:=Min to Max do
      if NHBane(E, H) then begin
        Inc(J);  T[J]:=E-MinNHElem;
      end;
      N:=J;
  end;
end;

procedure HRendez1(var T: TSor; N: TElemDb);
var
  H: TNHalmaz;  I: TEIndex;  J, K: TElemDb;  E, Min, Max: TNHElem;
  W: array[TNHElem] of TElemDb; {gyakoriságok}
begin
  if N>0 then begin
    for E:=MinNHElem to MaxNHElem do W[E]:=0;
    NHUres(H);
    Min:=MaxNHElem;  Max:=MinNHElem;
    {az elemeket transzformáltan felvesszük a tömbből halmazba,
     közben minimumot és maximumot számítunk, valamint gyakoriságot
     gyűjtünk}
    for I:=1 to N do begin
      E:=T[I]+MinNHElem;  NHba(E, H);  Inc(W[E]);
      if E<Min then Min:=E;
      if E>Max then Max:=E;
    end;
    {az elemeket a halmazból transzformáltan visszaírjuk a tömbbe,
     a gyakoriság számolásával}
    J:=0;
    for E:=Min to Max do if NHBane(E, H) then
      for K:=1 to W[E] do begin
        Inc(J);
        T[J]:=E-MinNHElem;
      end;
  end;
end;
```

```
procedure Lotto5b(var Huzas: TL5Huzas);
type
  TL5Jelzo=array[TL5Szam] of Boolean;
var
  Jelzo: TL5Jelzo;  I, X: TL5Szam;  J: TL5Db;
begin
  for I:=1 to 90 do Jelzo[I]:=False;
  for J:=1 to 5 do begin
    repeat
      X:=Random(90)+1;
    until not Jelzo[X];
    Jelzo[X]:=True;
  end;
  J:=0;
  for I:=1 to 90 do if Jelzo[I] then begin
    Inc(J);  Huzas[J]:=I;
  end;
end;

function Ismetele(const A: TSor; N: TElemDb): Boolean;
var
  H: TNHalmaz;  I: TEIndex1;  Ismetel: Boolean;
begin
  NHUres(H);  Ismetel:=False;  I:=1;
  while (I<=N) and (not Ismetel) do begin
    Ismetel:=NHBane(A[I], H);  NHBa(A[I], H);  Inc(I);
  end;
  Ismetele:=Ismetel;
end;

procedure JelGyak(S: String; var Gyak: TJelGyak);
var
  I: Char;  J: Byte;
begin
  for I:=Low(Char) to High(Char) do Gyak[I]:=0;
  for J:=1 to Length(S) do Inc(Gyak[S[J]]);
end;
end.
```

uInp unit

```
unit uInp;

interface

uses uDef;

{ Általános szöveg }
function AltSzovKozbe(
  Szoveg:  String;    {i adat szöveg}
  AlapJel: TJelek;    {i alapjelek}
  MaxAdH:  Byte;      {i max adathossz}
  ElvJel:  TJelek;    {i elválasztó jelek}
  MaxEDb:  Byte):     {i max elválasztó jel db}
           Boolean;   {o érvényesség}
```

```
function AltSzovVeg(
  Szoveg:  String;   {i adat szöveg}
  AlapJel: TJelek;   {i alapjelek}
  MinAdH:  Byte;     {i min adathossz}
  MinEDb:  Byte):    {i min elválasztó jel db}
           Boolean;  {o érvényesség}

{ Gépkocsirendszám }
function RendSzamKozbe(RendSzam: String): Boolean;
function RendSzamVeg(RendSzam: String): Boolean;

implementation

function AltSzovKozbe(Szoveg: String; AlapJel: TJelek; MaxAdH: Byte;
  ElvJel: TJelek; MaxEDb: Byte): Boolean;
var
  JoJel: TJelek;  {aktuális jelhalmaz}
  ElvDb: Byte;  {aktuális elv.jel db}
  I: Byte;  JoAdat: Boolean;
begin
  JoAdat:=Length(Szoveg)<=MaxAdH;
  if JoAdat then begin
    JoJel:=AlapJel;  ElvDb:=0;  I:=0;
    while (I<Length(Szoveg)) and JoAdat do begin
      Inc(I);  Jel :=;
      if Szoveg[I] in ElvJel then Inc(ElvDb);
      JoAdat:= (Szoveg[I] in JoJel) and (ElvDb<=MaxEdb);
      if Szoveg[I] in AlapJel then
        JoJel:=AlapJel+ElvJel
      else JoJel:=AlapJel;
    end;
  end;
  AltSzovKozbe:=JoAdat;
end;

function AltSzovVeg(Szoveg: String; AlapJel: TJelek; MinAdH: Byte;
  MinEDb: Byte): Boolean;
var
  I, H, ElvDb: Byte;  JoAdat: Boolean;
begin
  H:=Length(Szoveg);  ElvDb:=0;
  JoAdat:=(H>=MinAdH) and (Szoveg[H] in AlapJel);
  if JoAdat then begin
    for I:=1 to Length(Szoveg) do
      if not (Szoveg[I] in AlapJel) then Inc(ElvDb);
    JoAdat:=ElvDb>=MinEdb;
  end;
  AltSzovVeg:=JoAdat;
end;

function RendSzamKozbe(RendSzam: String): Boolean;
const
  AdatHossz=6;
var
  JoJel: TJelek;  {aktuális jelhalmaz}
  I: Byte;  JoAdat: Boolean;
```

```
begin
  JoAdat:=Length(RendSzam)<=AdatHossz;
  if JoAdat then begin
    I:=0;
    while (I<Length(RendSzam)) and JoAdat do begin
      Inc(I);
      case I of
        1..2: JoJel:=NagyBetuk+KisBetuk;
           3: JoJel:=NagyBetuk+KisBetuk+SzamJegyek;
        4..5: JoJel:=SzamJegyek;
      end;
      JoAdat:=RendSzam[I] in JoJel;
    end;
  end;
  RendSzamKozbe:=JoAdat;
end;

function RendSzamVeg(RendSzam: String): Boolean;
const
  AdatHossz=6;
begin
  RendSzamVeg:=(Length(RendSzam)=AdatHossz) and
    (Copy(RendSzam, 4, 3)<>'000') and (Copy(RendSzam, 3, 4)<>'0000');
end;
end.
```

uLista unit

```
unit uLista;

interface

uses uListaD, dListaU;

{-- egyirányú lista --}

{ Lista inicializálása }
procedure LInit1(var Kezdo, Vege: PLElem1);
{ Keressünk egy adatot a listában }
function LKeres1(Kezdo: PLElem1; Mit: TAdat): PLElem1;
{ Új elem adott mutató utánra }
procedure LUjElem1(var Kezdo, Vege: PLElem1; Miutan, Mit: PLElem1);
{ Töröljünk egy adott mutatójú elemet a listából}
procedure LTorol1(var Kezdo, Vege: PLElem1; Mit: PLElem1);
{ Új adat a lista végére }
function LUjVegere1(var Kezdo, Vege: PLElem1; Mit: TAdat): PLElem1;
{ Új adat a lista elejére }
function LUjElore1(var Kezdo, Vege: PLElem1; Mit: TAdat): PLElem1;
{ Lista felszámolása }
procedure LFelsz1(var Kezdo: PLElem1);

{-- kétirányú fejelt lista --}

{ Lista inicializálása }
procedure LInitFej2(var Fej: PLElem2);
```

```
{ Új elem adott mutató utánra }
procedure LUjElemFej2(var Fej: PLElem2; Miutan, Mit: PLElem2);
{ Töröljünk egy adott mutatójú elemet a listából }
procedure LTorolFej2(var Fej: PLElem2; Mit: PLElem2);
{ Lista felszámolása }
procedure LFelszFej2(var Fej: PLElem2);

implementation

procedure LInit1(var Kezdo, Vege: PLElem1);
begin
  Kezdo:=nil;  Vege:=nil;
end;

function LKeres1(Kezdo: PLElem1; Mit: TAdat): PLElem1;
var
  Akt: PLElem1;
begin
  Akt:=Kezdo;
  while (Akt<>nil) and (Akt^.Adat<>Mit) do Akt:=Akt^.Koveto;
  if (Akt=nil) or (Akt^.Adat<>Mit) then
    LKeres1:=nil
  else LKeres1:=Akt;
end;

procedure LUjElem1(var Kezdo, Vege: PLElem1; Miutan, Mit: PLElem1);
begin
  {beillesztés}
  if (Miutan=nil) and (Kezdo=nil) {üres listára} then begin
    Mit^.Koveto:=nil;  Kezdo:=Mit;  Vege:=Mit;
  end else
  if Miutan=nil {nem üres lista elejére} then begin
    Mit^.Koveto:=Kezdo;  Kezdo:=Mit;
  end else {nem az elejére, a Miutan valódi cím}
  if Miutan=Vege then begin {lista végére}
    Mit^.Koveto:=nil;  Miutan^.Koveto:=Mit;  Vege:=Mit;
  end else begin {az új elem belső elem lesz}
    Mit^.Koveto:=Miutan^.Koveto;  Miutan^.Koveto:=Mit;
  end;
end;

procedure LTorol1(var Kezdo, Vege: PLElem1; Mit: PLElem1);
var
  Akt, Elo: PLElem1;
begin
  {kikapcsolás}
  if (Mit=Kezdo) and (Mit=Vege) {egyetlen} then begin
    Kezdo:=nil;  Vege:=nil;
  end else
  if Mit=Kezdo {első, de nem egyetlen} then
    Kezdo:=Mit^.Koveto
  else begin {nem első, keresni kell a megelőzőt, ilyen van}
    Akt:=Kezdo;  Elo:=nil;
    while (Akt<>nil) and (Akt<>Mit) do begin
      Elo:=Akt;  Akt:=Akt^.Koveto;
    end;
```

```
    if Mit=Vege {utolsó} then begin
      Elo^.Koveto:=nil;  Vege:=Elo
    end else {belső elem} Elo^.Koveto:=Mit^.Koveto;
  end;
  {megszüntetés}
  Dispose(Mit);
end;

procedure LFelsz1(var Kezdo: PLElem1);
var
  Vege: PLElem1;
begin
  while Kezdo<>nil do LTorol1(Kezdo, Vege, Kezdo);
end;

function LUjVegere1(var Kezdo, Vege: PLElem1; Mit: TAdat): PLElem1;
var
  Uj: PLElem1;
begin
  {létrehozás}
  Uj:=UjLElem1;
  if Uj<>nil then begin
    {feltöltés}
    with Uj^ do begin
      Adat:=Mit;  Koveto:=nil
    end;
    {beillesztés}
    if Kezdo=nil {üres volt} then Kezdo:=Uj else Vege^.Koveto:=Uj;
    Vege:=Uj;
  end;
  LUjVegere1:=Uj;
end;

function LUjElore1(var Kezdo, Vege: PLElem1; Mit: TAdat): PLElem1;
var
  Uj:PLElem1;
begin
  {létrehozás}
  Uj:=UjLElem1;
  if Uj<>nil then begin
    {feltöltés}
    Uj^.Adat:=Mit;  Uj^.Koveto:=Kezdo;
    {beillesztés}
    if Kezdo=nil {üres volt} then Vege:=Uj;
    Kezdo:=Uj;
  end;
  LUjElore1:=Uj;
end;

procedure LInitFej2(var Fej: PLElem2);
begin
  with Fej^ do begin
    Elozo:=nil;  Koveto:=nil;  Adat:='0';
  end;
end;
```

```
function StrPluszEgy(S: String): String;
var
  X, I: Integer;
begin
  Val(S, X, I);
  if I=0 then Str(X+1, S);
end;

function StrMinuszEgy(S: String): String;
var
  X, I: Integer;
begin
  Val(S, X, I);
  if I=0 then Str(X-1, S);
end;

procedure LUjElemFej2(var Fej:PLElem2;Miutan,Mit:PLElem2);
begin
  {Fej^.Elozo=a lista kezdete, Fej^.Koveto=a lista vége}
  {beillesztés}
  if (Miutan=nil) and (Fej^.Adat='0') {üres listára} then begin
    Mit^.Koveto:=nil;  Mit^.Elozo:=nil;
    Fej^.Elozo:=Mit;  Fej^.Koveto:=Mit;
  end else
  if Miutan=nil {nem üres lista elejére} then begin
    Mit^.Koveto:=Fej^.Elozo;  Mit^.Elozo:=nil;
    Fej^.Elozo:=Mit;
  end else {nem az elejére, a Miutan valódi cím}
  if Miutan=Fej^.Koveto then begin {lista végére}
    Mit^.Koveto:=nil;  Mit^.Elozo:=Miutan;
    Miutan^.Koveto:=Mit;  Fej^.Koveto:=Mit;
  end else begin {az új elem belső elem lesz}
   Miutan^.Koveto^.Elozo:=Mit;  Mit^.Koveto:=Miutan^.Koveto;
   Mit^.Elozo:=Miutan;  Miutan^.Koveto:=Mit;
  end;
  with Fej^ do Adat:=StrPluszEgy(Adat);
end;

procedure LTorolFej2(var Fej: PLElem2; Mit: PLElem2);
begin
  if (Mit=Fej^.Elozo) and (Mit=Fej^.Koveto) {egyetlen} then begin
    Fej^.Elozo:=nil;  Fej^.Koveto:=nil;
  end else
  if Mit=Fej^.Elozo {első, de nem egyetlen} then begin
    Mit^.Koveto^.Elozo:=nil;  Fej^.Elozo:=Mit^.Koveto;
  end else
  if Mit=Fej^.Koveto {utolsó, de nem egyetlen} then begin
    Mit^.Elozo^.Koveto:=nil;  Fej^.Koveto:=Mit^.Elozo;
  end else begin {belső elem}
    Mit^.Elozo^.Koveto:=Mit^.Koveto;  Mit^.Koveto^.Elozo:=Mit^.Elozo;
  end;
  with Fej^ do Adat:=StrMinuszEgy(Adat);
  Dispose(Mit);
end;
```

```
procedure LFelszFej2(var Fej: PLElem2);
begin
  while Fej^.Elozo<>nil do LTorolFej2(Fej, Fej^.Elozo);
end;
end.
```

uListaD unit

```
unit uListaD;

interface

type
  TAdat=String;

  {egyirányban láncolt lista}
  PLElem1=^TLElem1;
  TLElem1=record
    Adat: TAdat;
    Koveto: PLElem1;
  end;

  {két irányban láncolt lista}
  PLElem2=^TLElem2;
  TLElem2=record
   Adat: TAdat;
   Elozo, Koveto: PLElem2;
  end;

implementation
end.
```

uListaF unit

```
unit uListaF;

interface

uses uListaLD, dListaU;

function HetViszony(A, B: THetAzon): THetViszony; {A ~ B}
{ Hét keresése }
function LotKeres(Kezdo: PLotElem; Mit: THetAzon; var Elozo: PLotElem):
PLotElem;
{ Hét felvitele/módosítása }
function LotRa(var Kezdo, Vege: PLotElem; UjAdat: TLotRekord): PLotElem;
{ Lista előállítása fájlból }
function LotListaBe(var Kezdo, Vege: PLotElem; var F: TLotFajl): Boolean;
{ Fájl előállítása listából }
procedure LotListaKi(var Kezdo: PLotElem; var F: TLotFajl);
{ Gyakoriság számítása listából }
procedure LotGyak(Kezdo: PLotElem; KezdHet, VegHet: THetAzon;
  var Gyak: TRelSzamGyak);

implementation
```

```
function HetViszony(A,B:THetAzon):THetViszony;
begin
  if A.Ev<B.Ev then HetViszony:=HetK else
  if A.Ev>B.Ev then HetViszony:=HetN else
  if A.Het<B.Het then HetViszony:=HetK else
  if A.Het>B.het then HetViszony:=HetN else HetViszony:=HetE;
end;

function LotKeres(Kezdo: PLotElem; Mit: THetAzon; var Elozo: PLotElem):
PLotElem;
var
  Akt: PLotElem;
begin
  Akt:=Kezdo;  Elozo:=nil;
  while (Akt<>nil) and (HetViszony(Mit, Akt^.Adatok.Azon)=HeTN) do begin
    Elozo:=Akt;  Akt:=Akt^.Koveto;
  end;
  if (Akt=nil) or (HetViszony(Akt^.Adatok.Azon, Mit)<>HetE) then
    LotKeres:=nil
  else LotKeres:=Akt;
end;

function LotRa(var Kezdo, Vege: PLotElem; UjAdat: TLotRekord): PLotElem;
var
  Uj, Elo: PLotElem;
begin
  Uj:=LotKeres(Kezdo, Uj^.Adatok.Azon, Elo);
  if Uj<>nil {már létező hét} then
    Uj^.Adatok:=UjAdat
  else begin {új hét}
    Uj:=UjLotElem;
    if Uj<>nil then begin
      Uj^.Adatok:=UjAdat;
      {beillesztés}
      if (Elo=nil) and (Kezdo=nil) then begin
        Uj^.Koveto:=nil;  Kezdo:=Uj;  Vege:=Uj;
      end else
      if Elo=nil then begin
        Uj^.Koveto:=Kezdo;  Kezdo:=Uj;
      end else
      if Elo=Vege then begin
        Uj^.Koveto:=nil;  Elo^.Koveto:=Uj;  Vege:=Uj;
      end else begin
        Uj^.Koveto:=Elo^.Koveto;  Elo^.Koveto:=Uj;
      end;
    end;
  end;
  LotRa:=Uj;
end;

function LotListaBe(var Kezdo, Vege: PLotElem; var F: TLotFajl): Boolean;
var
  UjAdat: TLotRekord;  UjElem: PLotElem;  VanHely: Boolean;
begin
  Reset(F);  Kezdo:=nil;  Vege:=nil;  VanHely:=True;
  while (not Eof(F)) and VanHely do begin
```

```
    Read(F, UjAdat);
    UjElem:=UjLotElem;  VanHely:=UjElem<>nil;
    if VanHely then begin {végére}
      UjElem^.Adatok:=UjAdat;  UjElem^.Koveto:=nil;
      if Kezdo=nil then Kezdo:=UjElem else Vege^.Koveto:=UjElem;
      Vege:=UjElem;
    end;
  end;
  LotListaBe:=Eof(F) and VanHely;
  Close(F);
end;

procedure LotGyak(Kezdo: PLotElem; KezdHet, VegHet: THetAzon;
  var Gyak: TRelSzamGyak);
var
  I, Szam: TLotSzam;  J: TLotInd;  Akt: PLotElem;  HetDb: Word;
begin
  Akt:=Kezdo;
  while (Akt<>nil) and (HetViszony(Akt^.Adatok.Azon, KezdHet)=HetK) do
    Akt:=Akt^.Koveto;
  HetDb:=0;
  for I:=1 to MaxSzam do Gyak[I]:=0;
  while (Akt<>nil) and (HetViszony(Akt^.Adatok.Azon, VegHet)<>HetN) do begin
    for J:=1 to LotSzDb do begin
      Inc(HetDb);
      Szam:=Akt^.Adatok.Adat[J];  Gyak[Szam]:=Gyak[Szam]+1;
    end;
    Akt:=Akt^.Koveto;
  end;
  if HetDb>0 then
    for I:=1 to MaxSzam do Gyak[I]:=Gyak[I]/HetDb;
end;

procedure LotListaKi(var Kezdo: PLotElem; var F: TLotFajl);
var
  Akt: PLotElem;
begin
  Rewrite(F);
  Akt:=Kezdo;
  while Akt<>nil do begin
    Write(F, Akt^.Adatok);  Akt:=Akt^.Koveto;
  end;
  Close(F);
end;
end.
```

uListaLD unit

```
unit uListaLD;

interface

const
  LotSzDb=5;  MinEv=1960;  MaxEv=2099;  MaxHet=53;  MaxSzam=90;
```

```
type
  TLotSzam=1..MaxSzam;  TLotInd=1..LotSzDb;
  TEv=MinEv..MaxEv;     THet=1..MaxHet;
  THetAdat=array[TLotInd] of TLotSzam; {egy heti adat}
  THetAzon=record       {egy időadat, egy rekord azonosítója}
    Ev: TEv;
    Het: THet;
  end;
  TLotRekord=record     {egy hét a fájlban}
   Azon: THetAzon;
   Adat: THetAdat;
  end;
  PLotElem=^TLotElem;
  TLotElem=record       {egy hét a listán}
   Adatok: TLotRekord;
   Koveto: PLotElem;
  end;

  THetViszony=(HetK, HetE, HetN);  {két időadat viszonya}
  TRelSzamGyak=array[TLotInd] of Real;  {relatív gyakoriságok}
  TLotFajl=file of TLotRekord;

implementation
end.
```

uListaO unit

```
unit uListaO;

interface

type
  TTargySzo=String;     {tárgyszó}
  THivSzam=Word;        {hivatkozás}
  PHivRek=^THivRek;
  THivRek=record        {hivatkozási rekord}
    Szam: THivSzam;       {hivatkozás}
    Kov: PHivRek;
  end;
  PSzoRek=^TSzoRek;
  TSzoRek=record        {szórekord}
    Szo: TTargySzo;       {tárgyszó}
    EHiv, VHiv: PHivRek; {a kapcsolódó hivatkozási lista kezdő- és
                          végmutatója}
    Kov: PSzoRek;
  end;

procedure SzoFelvitel(
  var TMKezd: PSzoRek;    {i/o a tárgymutató kezdőcíme}
      Szo:    TTargySzo;  {i a felveendő hivatkozás szava}
      Szam:   THivSzam);  {i a felveendő hivatkozás száma}
{a helybiztosítás a hívó feladata}

implementation
```

```
function UjSzoRek(
  Szo:    TTargySzo; {i a felveendő új szó}
  Szam:   THivSzam;  {i a felveendő új szó hivatkozása}
  KovSzo: PSzoRek):  {i a szólistában a követő cím}
          PSzoRek;   {o az új szórekord címe}
var
  UjSzo: PSzoRek;  UjHiv: PHivRek;
begin
  New(UjSzo);
  UjSzo^.Szo:=Szo;  UjSzo^.Kov:=KovSzo;
  New(UjHiv);
  UjSzo^.EHiv:=UjHiv;  UjSzo^.VHiv:=UjHiv;
  UjHiv^.Szam:=Szam;  UjHiv^.Kov:=nil;
  UjSzoRek:=UjSzo;
end;

procedure SzoFelvitel(var TMKezd: PSzoRek; Szo: TTargySzo; Szam: THivSzam);
var
  AktSzo, Elozo: PSzoRek;  UjHiv: PHivRek;
begin
  if TMKezd=nil then
    TMKezd:=UjSzoRek(Szo, Szam, nil)
  else begin
    {keresés}
    AktSzo:=TMKezd;  Elozo:=nil;
    while (AktSzo^.Szo<Szo) and (AktSzo^.Kov<>nil) do begin
      Elozo:=AktSzo;  AktSzo:=AktSzo^.Kov;
    end;
    if AktSzo^.Szo<Szo then {végére új szó}
      AktSzo^.Kov:=UjSzoRek(Szo, Szam, nil)
    else
    if AktSzo^.Szo>Szo then begin {aktuális elé új szó}
      if Elozo=nil then {szólista elejére}
        TMKezd:=UjSzoRek(Szo, Szam, TMKezd)
      else {lista belsejébe} Elozo^.Kov:=UjSzoRek(Szo, Szam, Elozo^.Kov);
    end else begin {már meglévő szó új hivatkozása}
      New(UjHiv);
      UjHiv^.Szam:=Szam;  UjHiv^.Kov:=nil;
      AktSzo^.VHiv^.Kov:=UjHiv;  AktSzo^.VHiv:=UjHiv;
    end;
  end;
end;
end.
```

uListaR unit

```
unit uListaR;

interface

uses uListaD, dListaU, uLista;

{-- egyirányú lista --}

{ Adat keresése }
function RLKeres1(Kezdo: PLElem1; Mit: TAdat; var Elozo: PLElem1): PLElem1;
```

```
{ Új adat }
function RListara1(var Kezdo, Vege: PLElem1; UjAdat: TAdat): PLElem1;
{ Adat törlése }
function RListarol1(var Kezdo, Vege: PLElem1; TorlAdat: TAdat): Boolean;

{-- kétirányú fejelt lista --}

{ Adat keresése }
function RLKeresFej2(Fej: PLElem2; Mit: TAdat): PLElem2;
{ Új adat }
function RListaraFej2(var Fej: PLElem2; UjAdat: TAdat): PLElem2;
{ Adat törlése }
function RListarolFej2(var Fej: PLElem2; TorlAdat: TAdat): Boolean;

{ Lista létrehozása szövegfájlból }
function SListaBe(
  var SLista: PLElem1;  {o a létrehozandó lista kezdete}
  var Vege:   PLElem1;  {o a létrehozandó lista vége}
  var F:      Text):    {i az input szövegfájl}
              Boolean;  {o a lista létrejötte}

{ Szövegfájl létrehozása listából }
procedure SListaKi(
  var SLista: PLElem1;  {i a lista kezdete}
  var F:      Text);    {o az eredmény szövegfájl}

implementation

function RLKeres1(Kezdo: PLElem1; Mit: TAdat; var Elozo: PLElem1): PLElem1;
var
  Akt: PLElem1;
begin
  Akt:=Kezdo;  Elozo:=nil;
  while (Akt<>nil) and (Mit>Akt^.Adat) do begin
    Elozo:=Akt;  Akt:=Akt^.Koveto;
  end;
  if (Akt=nil) or (Akt^.Adat<>Mit) then
    RLKeres1:=nil
  else RLKeres1:=Akt;
end;

function RListara1(var Kezdo, Vege: PLElem1; UjAdat: TAdat): PLElem1;
var
  Uj, Elo: PLElem1;
begin
  Uj:=UjLElem1;
  if Uj<>nil then begin
    Uj^.Adat:=UjAdat;
    {helykeresés, beillesztés}
    RLKeres1(Kezdo, UjAdat, Elo);
    LUjElem1(Kezdo, Vege, Elo, Uj);
  end;
  RListara1:=Uj;
end;
```

```
function RListarol1(var Kezdo, Vege: PLElem1; TorlAdat: TAdat): Boolean;
var
  Torl, Elozo: PLElem1;
begin
  Torl:=RLKeres1(Kezdo, TorlAdat, Elozo);
  RListarol1:=Torl<>nil;
  if Torl<>nil then LTorol1(Kezdo, Vege, Torl);
end;

function RLKeresFej2(Fej: PLElem2; Mit: TAdat): PLElem2;
var
  Akt: PLElem2;
begin
  Akt:=Fej^.Elozo;
  while (Akt<>nil) and (Mit>Akt^.Adat) do Akt:=Akt^.Koveto;
  if (Akt=nil) or (Akt^.Adat<>Mit) then
    RLKeresFej2:=nil
  else RLKeresFej2:=Akt;
end;

function RListaraFej2(var Fej: PLElem2; UjAdat: TAdat): PLElem2;
var
  Uj, Elo: PLElem2;
begin
  Uj:=UjLElem2;
  if Uj<>nil then begin
    Uj^.Adat:=UjAdat;
    {helykeresés, beillesztés}
    if (Fej^.Adat='0') or (UjAdat<=Fej^.Elozo^.Adat) then
      Elo:=nil
    else if UjAdat>Fej^.Koveto^.Adat then
      Elo:=Fej^.Koveto
    else Elo:=RLKeresFej2(Fej, UjAdat)^.Elozo;
    LUjElemFej2(Fej, Elo, Uj);
  end;
  RListaraFej2:=Uj;
end;

function RListarolFej2(var Fej: PLElem2; TorlAdat: TAdat): Boolean;
var
  Torl: PLElem2;
begin
  Torl:=RLKeresFej2(Fej, TorlAdat);
  RListarolFej2:=Torl<>nil;
  if Torl<>nil then LTorolFej2(Fej, Torl);
end;

function SListaBe(var SLista: PLElem1; var Vege: PLElem1;
  var F: Text): Boolean;
var
  UjAdat: TAdat;  VanHely: Boolean;
begin
  Reset(F);
  LInit1(SLista, Vege);
  VanHely:=True;
  while (not Eof(F)) and VanHely do begin
```

```
    Readln(F, UjAdat);
    VanHely:=RListara1(SLista, Vege, UjAdat)<>nil;
  end;
  Close(F);
  SListaBe:=Eof(F) and VanHely;
end;

procedure SListaKi(var SLista: PLElem1; var F: Text);
var
  Akt: PLElem1;
begin
  Rewrite(F);  Akt:=SLista;
  while Akt<>nil do begin
    Writeln(F, Akt^.Adat);  Akt:=Akt^.Koveto;
  end;
  Close(F);
  LFelsz1(SLista);
end;
end.
```

uMatrix unit

```
unit uMatrix;

interface

const
  SorMaxDb=50;  OszlMaxDb=50;  {maximális sor és oszlopszám}

type
  TSInd=1..SorMaxDb;   TSInd0=0..SorMaxDb;    {sorindexek}
  TOInd=1..OszlMaxDb;  TOInd0=0..OszlMaxDb;   {oszlopindexek}
  TSorDb=0..SorMaxDb;  TOszlDb=0..OszlMaxDb;  {sor és oszlopszám}
  TMElem=Integer;      {mátrixelem}
  TMElemOssz=Longint;  {mátrixelemek összege}

  TMatrix=array[TSInd, TOInd] of TMElem;  {mátrix}
  TMatSor=array[TSInd] of TMElem;         {mátrixsor}
  TMatOszl=array[TOInd] of TMElem;        {mátrixoszlop}
  TOsszSor=array[TOInd] of TMElemOssz;    {összegsor}
  TOsszOszl=array[TSInd] of TMElemOssz;   {összegoszlop}

{ Mátrix minimumhelye }
procedure MatMin(
  const T:    TMatrix;  {i a mátrix}
        M:    TSInd;    {i sorok aktuális száma}
        N:    TOInd;    {i oszlopok aktuális száma}
  var   MinS: TSInd;    {o a minimum sorindexe}
  var   MinO: TOInd);   {o a minimum oszlopindexe}

{ Mátrix sorminimumok tömbje }
procedure MatSorMin(
  const T:      TMatrix;    {i a mátrix}
        M:      TSInd;      {i sorok aktuális száma}
        N:      TOInd;      {i oszlopok aktuális száma}
  var   SorMin: TMatOszl);  {o a sorminimumok tömbje}
```

```
{ Mátrixösszegek }
procedure MatSumm(
  const T:     TMatrix;      {i a mátrix}
        M:     TSInd;        {i sorok aktuális száma}
        N:     TOInd;        {i oszlopok aktuális száma}
  var   SO:    TOsszSor;     {o a sorösszegek tömbje}
  var   OO:    TOsszOszl;    {o az oszlopösszegek tömbje}
  var   TeljO: TMElemOssz);  {o a teljes összeg}

{ Mátrixsor törlése }
procedure MatSorTorl(
  var T:     TMatrix;  {i/o a mátrix}
  var M:     TSorDb;   {i sorok aktuális száma}
      N:     TOInd;    {i oszlopok aktuális száma}
      TorlI: TSInd);   {i a törlendő sor indexe}

{ Mátrix transzponálása }
procedure MatTranszp(
  var T: TMatrix;  {i/o a mátrix}
  var M: TSInd;    {i sorok aktuális száma}
  var N: TOInd);   {i oszlopok aktuális száma}

{ Mátrixok algebrai szorzása }
procedure MatSzor(
  const A:   TMatrix;   {i baloldali szorzó}
        MA:  TSInd;     {i az A sorainak száma}
        NA:  TOInd;     {i az A oszlopainak száma}
  const B:   TMatrix;   {i jobboldali szorzó}
        MB:  TSInd;     {i a B sorainak száma}
        NB:  TOInd;     {i a B oszlopainak száma}
  var   C:   TMatrix;   {o az eredmény}
  var   MC:  TSInd;     {o a C sorainak száma}
  var   NC:  TOInd;     {o a C oszlopainak száma}
  var   Ert: Boolean);  {o értelmezett-e a művelet}

{ Mátrix egységmátrix-e }
function EgysegMat(
  const T: TMatrix;   {i mátrix}
        M: TSorDb;    {i sorok aktuális száma}
        N: TOszlDb):  {i oszlopok aktuális száma}
           Boolean;   {o eredmény}

implementation

procedure MatMin(const T: TMatrix; M: TSInd; N: TOInd; var MinS: TSInd;
  var MinO: TOInd);
var
  I: TSInd;  J: TOInd;  Min: TMElem;
begin
  Min:=T[1, 1];  MinS:=1;  MinO:=1;
  for I:=1 to M do for J:=1 to N do
    if T[I, J]<Min then begin
      Min:=T[I, J];  MinS:=I;  MinO:=J;
    end;
end;
```

```
procedure MatSorMin(const T: TMatrix; M: TSInd; N: TOInd; var SorMin:
TMatOszl);
var
  I: TSInd;  J: TOInd;  Min: TMElem;
begin
  for I:=1 to M do begin
    Min:=T[I, 1];
    for J:=2 to N do if T[I, J]<Min then Min:=T[I, J];
    SorMin[I]:=Min;
  end;
end;

procedure MatSumm(const T: TMatrix; M: TSInd; N: TOInd; var SO: TOsszSor;
  var OO: TOsszOszl; var TeljO: TMElemOssz);
var
  I: TSInd;  J: TOInd;
begin
  TeljO:=0;
  for I:=1 to M do OO[I]:=0;
  for J:=1 to N do SO[J]:=0;
  for I:=1 to M do for J:=1 to N do begin
    OO[J]:=OO[J]+T[I, J];
    SO[I]:=SO[I]+T[I, J];
    TeljO:=TeljO+T[I, J];
  end;
end;

procedure MatSorTorl(var T: TMatrix; var M: TSorDb; N: TOInd; TorlI: TSInd);
var
  I: TSInd0;  J: TOInd;
begin
  for I:=TorlI to M-1 do for J:=1 to N do T[I, J]:=T[I+1, J];
  Dec(M);
end;

procedure MatTranszp(var T: TMatrix; var M: TSInd; var N: TOInd);
var
  I: TSInd;  J: TOInd;  Cse: TMElem;  W: TMatrix;  Csi: TSInd;
begin
  if M=N then {szimmetrikus eset}
    for I:=1 to M-1 do
      for J:=I+1 to N do begin
        Cse:=T[I, J];  T[I, J]:=T[J, I];  T[J, I]:=Cse;
      end
  else begin {nem szimmetrikus eset}
    for I:=1 to M do for J:=1 to N do W[J, I]:=T[I, J];
    T:=W;  Csi:=N;  N:=M;  M:=Csi;
  end;
end;

procedure MatSzor(const A: TMatrix; MA: TSInd; NA: TOInd; const B: TMatrix;
  MB: TSInd; NB: TOInd; var C: TMatrix; var MC: TSInd; var NC: TOInd;
  var Ert: Boolean);
var
  I: TSInd;  J: TOInd;  K: TOInd;
```

```
begin
  Ert:=NA=MB;
  if Ert then begin
    for I:=1 to MA do
      for J:=1 to NB do begin
        C[I, J]:=0;
        for K:=1 to NA do C[I, J]:=C[I, J]+A[I, K]*B[K, J];
    end;
    MC:=MA;  NC:=NB;
  end;
end;

function EgysegMat(const T: TMatrix; M: TSorDb; N: TOszlDb): Boolean;
var
  I: TSInd;  J: TOInd;  Jo: Boolean;
begin
   Jo:=(N>0) and (N=M);
   if Jo then for I:=1 to M do for J:=1 to N do
     Jo:=Jo and ((I<>J) and (T[I, J]=0) or (I=J) and (T[I, J]=1));
   EgysegMat:=Jo;
end;
end.
```

uMKoll unit

```
unit uMKoll;

interface

uses uMKollD, dMKollU;

{ Táblázat létrehozása }
function UjMKoll(
  var MKoll: TMKoll;    {o új kollekció mutatótömb}
      N:     TMSorDb):  {i mátrix mérete}
             Boolean;   {o a kollekció létrejötte}

{ Táblázat feltöltése }
procedure ToltMKoll(
  const MKoll: TMKoll;    {i kollekció mutatótömb}
        N:     TMSorDb;   {i mátrix mérete}
        K:     Longint);  {i az értékek felső korlátja}

{ Táblázat bővítése összegsorral és -oszloppal }
function BovMKoll(
  var MKoll: TMKoll;    {i/o új kollekció mutatótömb}
  var N:     TMSorDb):  {i/o mátrix mérete}
             Boolean;   {o a bővítés létrejötte}

{ Sor törlése a táblázatból }
procedure TorlSorMKoll(
  var MKoll: TMKoll;   {i/o kollekció mutatótömb}
  var N:     TMSorDb;  {i/o mátrix mérete}
      TI:    TMSorI);  {i a törlendő sor indexe}

implementation
```

```
function UjMKoll(var MKoll: TMKoll; N: TMSorDb): Boolean;
var
  I: TMSorDb;  VanHely: Boolean;  UjSor: PMSor;
begin
  VanHely:=True;  I:=0;
  while (I<N) and VanHely do begin
    UjSor:=UjMTetel;
    VanHely:=UjSor<>nil;
    if VanHely then begin
      Inc(I);  MKoll[I]:=UjSor;
    end;
  end;
  UjMKoll:=N=I;
end;

procedure ToltMKoll(const MKoll: TMKoll; N: TMSorDb; K: Longint);
var
  I: TMSorI;  J: TMOszlI;
begin
   for I:=1 to N do for J:=1 to N do MKoll[I]^[J]:=Random(K);
end;

function BovMKoll(var MKoll: TMKoll; var N: TMSorDb): Boolean;
var
  I: TMSorI;  J: TMOszlI;  UjSor: PMSor;  X: Longint;
begin
  UjSor:=UjMTetel;
  if UjSor<>nil then begin
    MKoll[N+1]:=UjSor;
    for J:=1 to N+1 do MKoll[N+1]^[J]:=0;
    for I:=1 to N+1 do MKoll[I]^[N+1]:=0;
    X:=0;
    for I:=1 to N do
      for J:=1 to N do begin
        X:=X+MKoll[I]^[J];
        MKoll[N+1]^[J]:=MKoll[N+1]^[J]+MKoll[I]^[J];
        MKoll[I]^[N+1]:=MKoll[I]^[N+1]+MKoll[I]^[J];
      end;
    MKoll[N+1]^[N+1]:=X;
    Inc(N);  BovMKoll:=True;
  end else BovMKoll:=False;
end;

procedure TorlSorMKoll(var MKoll: TMKoll; var N: TMSorDb; TI: TMSorI);
var
  I: TMSorI;
begin
  {tétel megszűntetése}
  Dispose(MKoll[TI]);
  {mutató törlése}
  for I:=TI to N-1 do MKoll[I]:=MKoll[I+1];
  Dec(N);
end;
end.
```

uMKollD unit

```
unit uMKollD;

interface

const
  MaxMSorDb=10000;  {maximális sorszám (tételszám)}
  MaxMOszlDb=10000;  {maximális elemszám soronként}

type
  TMSorI=1..MaxMSorDb;  TMOszlI=1..MaxMOszlDb;  {indextípusok}
  TMSorDb=0.. MaxMSorDb;  TMOszlDb=0.. MaxMOszlDb;  {darabszám típusok}
  TMSor=array[TMOszlI] of Longint;  {sor (tétel) adattípus}
  PMSor=^TMSor;  { sor (tétel) mutatótípus}
  TMKoll=array[TMSorI] of PMSor; {mutatótömb típus}

implementation
end.
```

uNHalm unit

```
unit uNHalm;

interface

const
  MinNHElem=-8192; MaxNHElem=8191;  {elemtípus min., max.}
  MaxNHDb=MaxNHElem-MinNHElem+1;  {egy halmaz max. elemszáma=16384}
  NHTombEDb=MaxNHDb div 8;            {a bittérkép tömb elemszáma=2048}

type
  TNHElem=MinNHElem..MaxNHElem;       {elemtípus}
  TNHDb=0..MaxNHDb;                   {elemdarabszám típus}
  TNHTombInd=1..NHTombEdb;            {a bittérkép tömb indextípusa}
  TNHTomb=array[TNHTombInd] of Byte;  {a bittérkép tömb típusa}

  TNHalmaz=record    {a halmaz típusa}
    EDb: TNHDb;        {aktuális elemszám}
    Adat: TNHTomb;     {bittérkép tömb}
  end;

{ Az A üres lesz }
procedure NHUres(var A: TNHalmaz);
{ Az A ürese-e }
function NHUrese(const A: TNHalmaz): Boolean;
{ Az E eleme-e A-nak }
function NHbane(E: TNHElem; const A: TNHalmaz): Boolean;
{ Az A elemszáma }
function NHEDb(const A: TNHalmaz): TNHDb;
{ Az E eleme lesz A-nak }
procedure NHba(E: TNHElem; var A: TNHalmaz);
{ Az E törlödik A-bol }
procedure NHbol(E: TNHElem; var A: TNHalmaz);
```

```
{ A C az A és B egyesítése lesz }
procedure NHOssze(const A, B: TNHalmaz; var C: TNHalmaz);
{ A C az A és B metszete lesz }
procedure NHMetsz(const A, B:TNHalmaz; var C: TNHalmaz);
{ Az A a B része-e }
function NHResz(const A, B:TNHalmaz): Boolean;
{ Az A és a B egyenlő-e }
function NHAzon(const A, B:TNHalmaz): Boolean;

implementation

{-- Belső, implementációs konstansok, típusok és szubrutinok --}

const
  MaxBit=8;

type
  ByteInd=TNHTombInd;
  BitInd=1..MaxBit;

const
  Egyseg: array[BitInd] of Byte=(1, 2, 4, 8, 16, 32, 64, 128);

procedure ErtekHely(E: TNHElem; var ByteI: ByteInd; var BitI: BitInd);
var
  BitHely: Longint;
begin
  BitHely:=E-MinNHElem;
  ByteI:=BitHely div MaxBit+1;
  BitI:=BitHely mod MaxBit+1;
end;

function BitErt(B: Byte; I: BitInd): Byte;
begin
  BitErt:=B and Egyseg[I];
end;

function Bit1e(B: Byte; I: BitInd): Boolean;
begin
  Bit1e:=(B and Egyseg[I])>0;
end;

{-- A műveletek megvalósítása --}

procedure NHUres(var A:TNHalmaz);
var
  I: ByteInd;
begin
  for I:=1 to NHTombEdb do A.Adat[I]:=0;
  A.Edb:=0;
end;

function NHUrese(const A: TNHalmaz): Boolean;
begin
  NHUrese:=A.Edb=0;
end;
```

```
function NHbane(E: TNHElem; const A: TNHalmaz): Boolean;
var
  ByteI: ByteInd;  BitI: BitInd;
begin
  ErtekHely(E, ByteI, BitI);
  with A do NHbane:=Bit1e(Adat[ByteI], BitI);
end;

function NHEDb(const A: TNHalmaz): TNHDb;
begin
  NHEDb:=A.EDb;
end;

procedure NHba(E: TNHElem; var A: TNHalmaz);
var
  ByteI: ByteInd;  BitI: BitInd;
begin
  ErtekHely(E, ByteI, BitI);
  with A do
    if not Bit1e(Adat[ByteI], BitI) then begin
      Adat[ByteI]:=Adat[ByteI] or Egyseg[BitI];  Inc(EDb);
    end;
end;

procedure NHbol(E: TNHElem; var A: TNHalmaz);
var
  ByteI: ByteInd;  BitI: BitInd;
begin
  ErtekHely(E, ByteI, BitI);
  with A do
    if Bit1e(Adat[ByteI],BitI) then begin
      Adat[ByteI]:=Adat[ByteI] xor Egyseg[BitI];  Dec(EDb);
    end;
end;

procedure EDbBeall(var A: TNHalmaz);
var
  ByteI: ByteInd;  BitI: BitInd;
begin
  A.EDb:=0;
  for ByteI:=1 to NHTombEDb do for BitI:=1 to MaxBit do
    if Bit1e(A.Adat[ByteI], BitI) then Inc(A.EDb);
end;

procedure NHOssze(const A, B: TNHalmaz; var C: TNHalmaz);
var
  I: ByteInd;
begin
  for I:=1 to NHTombEDb do C.Adat[I]:=A.Adat[I] or B.Adat[I];
  EDbBeall(C);
end;

procedure NHMetsz(const A, B: TNHalmaz; var C: TNHalmaz);
var
  I: ByteInd;
begin
```

```
  for I:=1 to NHTombEDb do C.Adat[I]:=A.Adat[I] and B.Adat[I];
  EDbBeall(C);
end;

function NHResz(const A,B:TNHalmaz):boolean;
var
  ByteI: ByteInd;  BitI: BitInd;  Jo: Boolean;
begin
  Jo:=True;
  for ByteI:=1 to NHTombEDb do for BitI:=1 to MaxBit do
    Jo:=Jo and (BitErt(A.Adat[ByteI], BitI)<=BitErt(B.Adat[ByteI],
      BitI));
  NHResz:=Jo;
end;

function NHAzon(const A, B: TNHalmaz): Boolean;
var
  ByteI: ByteInd;  BitI: BitInd;  Jo: Boolean;
begin
  Jo:=True;
  for ByteI:=1 to NHTombEDb do for BitI:=1 to MaxBit do
    Jo:=Jo and (BitErt(A.Adat[ByteI], BitI)=BitErt(B.Adat[ByteI],
      BitI));
  NHAzon:=Jo;
end;
end.
```

uSKoll unit

```
unit uSKoll;

interface

uses uSKollD, dSKollU;

{ Egy érték keresése }
function SKeres(
  const Miben:   TSKoll;    {i ebben keresünk}
        Hanyban: TSTetDb;   {i a Miben aktuális elemszáma}
        Mit:     TSTet;     {i a keresett érték}
  var   Hol:     TSTetI1):  {o a Mit helye}
                 Boolean;   {a Mit létezése a Miben}

{ Egy érték beszúrása adott helyre }
function SBeszur(
  var Mibe: TSKoll;    {i/o ebbe szúrunk be}
  var Hanyba:TSTetDb;  {i/o a Mibe aktuális elemszáma}
      Mit   :TSTet;    {i a beszúrandó érték}
      Hova  :TSTetI):  {i a beszúrás helye}
            Boolean;   {o a művelet végrehajthatósága}

{ Egy elem törlése }
procedure STorol(
  var Mibol:   TSKoll;   {i/o ebből törlünk }
  var Hanybol: TSTetDb;  {i/o a Mibol aktuális elemszáma}
      Honnan:  TSTetI);  {i a törlendő elem indexe}
```

```
{ Egy elem felszámolása }
procedure SFelszamol(
  var Mibol:   TSKoll;   {i/o ebből törlünk }
  var Hanybol: TSTetDb;  {i/o a Mibol aktuális elemszáma}
      Honnan:  TSTetI);  {i a törlendő elem indexe}

{ Rendezés mimimumkiválasztással }
procedure SKivalRend(
  var Adatok: TSKoll;   {i/o a rendezendő kollekció mutatótömbje}
      AdatDb: TSTetDb); {i a kollekció aktuális elemszáma}

{ Bináris keresés }
function SBinKer(
  const Miben:   TSKoll;    {i nemcsökkenően rendezett, ebben keresünk}
        Hanyban: TSTetDb;   {i a Miben aktuális elemszáma}
        Mit:     TSTet;     {i a keresett érték}
  var   Hol:     TSTetI1):  {o a Mit helye}
                 Boolean;   {o a Mit létezése a Miben}

{ Kollekció létrehozása szövegfájlból }
function SKollBe(
  var SKoll: TSKoll;   {o a létrehozandó kollekció mutatótömbje}
  var N:     TSTetDb;  {o a létrehozandó kollekció elemszáma}
  var F:     Text):    {i az input szövegfájl}
             Boolean;  {o a kollekció létrejötte}

{ Szövegfájl létrehozása kollekcióból }
procedure SKollKi(
  const SKoll: TSKoll;   {i a kollekció mutatótömbje}
        N:     TSTetDb;  {i a kollekció elemszáma}
  var   F:     Text);    {o az eredmény szövegfájl}

implementation

function SKeres(const Miben: TSKoll; Hanyban: TSTetDb; Mit: TSTet;
  var Hol: TSTetI1): Boolean;
begin
  Hol:=1;
  while (Hol<=Hanyban) and (Mit<>Miben[Hol]^) do Hol:=Hol+1;
  SKeres:=(Hol<=Hanyban) and (Mit=Miben[Hol]^);
end;

function SBeszur(var Mibe: TSKoll; var Hanyba: TSTetDb; Mit: TSTet;
  Hova: TSTetI): Boolean;
var
  I: TSTetI;  Uj: PStet;
begin
  Uj:=UjSTetel;
  if Uj<>nil then begin
    Uj^:=Mit;
    for I:=Hanyba downto Hova do Mibe[I+1]:=Mibe[I];
    Mibe[Hova]:=Uj;  Inc(Hanyba);  SBeszur:=True;
  end else SBeszur:=False;
end;
```

```
procedure STorol(var Mibol: TSKoll; var Hanybol: TSTetDb; Honnan: TSTetI);
var
  I: TSTetI;
begin
  for I:=Honnan to Hanybol-1 do Mibol[I]:=Mibol[I+1];
  Dec(Hanybol);
end;

procedure SFelszamol(var Mibol: TSKoll; var Hanybol: TSTetDb;
  Honnan: TSTetI);
begin
  Dispose(Mibol[Honnan]);  STorol(Mibol, Hanybol, Honnan);
end;

procedure SKivalRend(var Adatok: TSKoll; AdatDb: TSTetDb);
var
  SorKezd, MinHely, I: TSTetI;  MinI: TSTet;  Cs: PSTet;
begin
  if AdatDb>1 then
    for SorKezd:=1 to AdatDb-1 do begin  {minimumhely meghatározása}
      MinHely:=SorKezd;  MinI:=Adatok[SorKezd]^;
      for I:=SorKezd+1 to AdatDb do
        if Adatok[I]^<Mini then begin
          MinHely:=I;  MinI:=Adatok[I]^;
        end;
      {csere}
      Cs:=Adatok[MinHely];
      Adatok[MinHely]:=Adatok[SorKezd];
      Adatok[SorKezd]:=Cs;
    end;
end;

function SBinKer(const Miben: TSKoll; Hanyban: TSTetDb; Mit: TSTet;
  var Hol: TSTetI1): Boolean;
var
  Kezd: TSTetI1;  Veg: TSTetI0;  Van: Boolean;
begin
  Kezd:=1;  Van:=False;
  if Hanyban>0 then begin
    Veg:=Hanyban;
    repeat
      Hol:=(Kezd+Veg) div 2;
      Van:=Mit=Miben[Hol]^;
      if not Van then
        if Mit<Miben[Hol]^ then Veg:=Hol-1 else Kezd:=Hol+1;
    until (Kezd>Veg) or Van;
  end;
  if not Van then Hol:=Kezd;
  SBinKer:=Van;
end;

function SKollBe(var SKoll: TSKoll; var N: TSTetDb; var F: Text): Boolean;
var
  Uj: PSTet;  VanHely: Boolean;
begin
  N:=0;  Reset(F);  VanHely:=True;
```

```
  while (not Eof(F)) and VanHely do begin
    Uj:=UjSTetel;  VanHely:=Uj<>nil;
    if VanHely then begin
      Readln(F, Uj^);  Inc(N);  SKoll[N]:=Uj;
    end;
  end;
  Close(F);
  SKollBe:=Eof(F);
end;

procedure SKollKi(const SKoll: TSKoll; N: TSTetDb; var F: Text);
var
  I: TSTetI0;
begin
  Rewrite(F);
  for I:=1 to N do begin
     Writeln(F, SKoll[I]^);  Dispose(SKoll[I]);
  end;
  Close(F);
end;
end.
```

uSKollD unit

```
unit uSKollD;

interface

const
  MaxSTetDb=10000;  {maximális tételszám}

type
  {indextípusok}
  TSTetI=1..MaxSTetDb;  TSTetI0=0..MaxSTetDb;  TSTetI1=1..MaxSTetDb+1;
  TSTetDb=0..MaxSTetDb;  {darabszám típus}
  TSTet=String;  {tétel adattípus}
  PSTet=^TSTet;  {tétel mutatótípus}
  TSKoll=array[TSTetI] of PSTet;  {mutatótömb típus}

implementation
end.
```

uString unit

```
unit uString;

interface

{ H hosszú, csupa C jelből álló string }
function JelSor(C: Char; H: Byte): String;
{ H db szóközből álló string }
function Szokoz(H: Byte): String;
{ Az eredmény az S értékéből a kezdő szóközök törlésével keletkezik }
function BalVag(S: String): String;
```

```
{ Az eredmény az S értékének H hosszban balra igazításával és
  jobbról szóközökkel való feltöltésével keletkezik }
function JobbTolt(S: String; H: Byte): String;
{ Az eredmény az S értékéből a befejező szóközök törlésével keletkezik }
function JobbVag(S: String): String;
{ Az eredmény az S értékének H hosszban jobbra igazításával és
  balról a Tolto karakterrel való feltöltésével keletkezik }
function BalTolt(S: String; H: Byte; Tolto: Char): String;
{ Az eredmény az S értékének egy H hosszú, az első nem szóköz jelig
  balról nullákkal feltöltött stringgé alakításával keletkezik }
function ElolNull(S: String; H: Byte): String;

{ Általános megjegyzés: a JobbTolt és BalTolt függvények értékes (nemszóköz)
  jelet nem vágnak le }

{ A Mibol értékéből a Mit értéke első előfordulását eltávolítja }
procedure KiVag(var Mibol: String; Mit: String);
{ Az S minden jelére végrehajtjuk a nagybetűvé alakítást (UpCase) }
procedure UpSzov(var S: String);
{ A C kisbetűs alakja }
function LoJel(C: Char): Char;
{ Az S minden jelére végrehajtjuk a kisbetűvé alakítást }
procedure LoSzov(var S: String);
{ Az eredmény az S értékéből a jelek fordított sorrendbe állításával
  keletkezik }
function MegFordit(S: String): String;

procedure Egyezes(const S1, S2: String; var Kp, MaxDb: Byte);
function BinbolHexa(S: String): String;

implementation

function JelSor(C: Char; H: Byte): String;
var
  I: Byte;  S: String;
begin
  S:='';
  for I:=1 to H do S:=S+C;
  JelSor:=S;
end;

function Szokoz(H: Byte): String;
begin
  Szokoz:=JelSor(' ', H);
end;

function BalVag(S:string):string;
var
  I: Byte;
begin
  I:=1;
  while (I<=Length(S)) and  (S[I]=' ') do Inc(I);
  BalVag:=Copy(S, I, Length(S)-(I-1));
end;
```

```
function JobbTolt(S: String; H: Byte): String;
begin
  S:=BalVag(S);
  while Length(S)<H do S:=S+' ';
  JobbTolt:=S;
end;

function JobbVag(S:string):string;
var
  I: Byte;
begin
  I:=Length(S);
  while (S[I]=' ') and (I>0) do I:=I-1;
  JobbVag:=Copy(S, 1, I);
end;

function BalTolt(S: String; H: Byte; Tolto: Char): String;
var
  I: Byte;
begin
  S:=JobbVag(BalVag(S));
  for I:=Length(S)+1 to H do S:=Tolto+S;
  BalTolt:=S;
end;

function ElolNull(S: String; H: Byte): String;
begin
  ElolNull:=BalTolt(S, H, '0');
end;

procedure KiVag(var Mibol:string; Mit:string);
var
  I: Byte;
begin
  I:=Pos(Mit, Mibol);
  if I>0 then Delete(Mibol, I, Length(Mit));
end;

procedure UpSzov(var S:string);
var
  I: Byte;
begin
  for I:=1 to Length(S) do S[I]:=UpCase(S[I]);
end;

function LoJel(C: Char): Char;
begin
  if C in ['A'..'Z'] then Inc(C, Ord('a')-Ord('A'));
  LoJel:=C;
end;

procedure LoSzov(var S:string);
var
  I: Byte;
begin
  for I:=1 to Length(S) do S[I]:=LoJel(S[I]);
end;
```

```
procedure Egyezes(const S1, S2: String; var Kp, MaxDb: Byte);
var
  I, J, Db: Byte;
begin
  Kp:=0;  MaxDb:=0;
  for I:=1 to Length(S2)-Length(S1)+1 do begin
    Db:=0;
    for J:=1 to Length(S1) do if S1[J]=S2[I+J-1] then Inc(Db);
    if Db>MaxDb then begin
      MaxDb:=Db;  Kp:=I
    end;
  end;
end;

function MegFordit(S: String): String;
var
  I: Byte;  Ss: String;
begin
  Ss:='';
  for I:=Length(S) downto 1 do Ss:=Ss+S[I];
  Megfordit:=Ss;
end;

function BinbolHexa(S:string):string;
const
  HJegy: String='0123456789ABCDEF';
  Hatv: array[1..4] of Byte=(1, 2, 4, 8);
var
  I, X: Byte;  Ss, Sss: String;
begin
  S:=Megfordit(S);  Ss:='';
  while Length(S)>0 do begin
   Sss:=Copy(S, 1, 4);  Delete(S, 1, 4);  X:=0;
   for I:=1 to Length(Sss) do if Sss[I]='1' then X:=X+Hatv[I];
   Ss:=Ss+HJegy[X+1];
  end;
  BinbolHexa:=Megfordit(Ss);
end;
end.
```

uTbStat unit

```
unit uTbStat;

interface

uses uTomb, uTombR;

const
  MaxErtDb=1000;

type
  ErtDb=0..MaxErtDb;
  TStat=array[TEIndex] of ErtDb;
```

```
{ Érték szerint növekvően rendezett gyakorisági statisztika gyűjtése}
procedure GyakGyujt(
  var Adat: TSor;     {i/o rendezett adatsor}
  var Gyak: TStat;    {i/o előfordulási darabszámok}
  var N:    TElemDb;  {i/o adat és gyak aktuális elemszáma}
      Uj:   TTElem);  {i új, felveendő érték}

implementation

procedure GyakGyujt(var Adat: TSor; var Gyak: TStat; var N: TElemDb;
  Uj: TTElem);
var
  I: TEIndex0;  Hol: TEIndex1;
begin
  if BinKer(Adat, N, Uj, Hol) then
    Inc(Gyak[Hol])
  else begin
    for I:=N downto Hol do begin
      Adat[I+1]:=Adat[I];  Gyak[I+1]:=Gyak[I];
    end;
    Adat[Hol]:=Uj;  Gyak[Hol]:=1;  Inc(N);
  end;
end;
end.
```

uTomb unit

```
unit uTomb;

interface

{-- közös deklarációk egydimenziós tömbök kezeléséhez --}

const
  EMaxDb=100; {maximális elemszám}

type
  TTElem=Integer;  {elemtípus}
  TEIndex=1..EMaxDb;  {indextípus}
  TEIndex0=0..EMaxDb;  TEIndex1=1..EMaxDb+1;
  TElemDb=0..EMaxDb;  {darabszámtípus}
  TSor=array[TEIndex] of TTElem; {tömbtípus}

{ Egy érték keresése }
function Keres(
  const Miben:   TSor;       {i ebben keresünk}
        Hanyban: TElemDb;    {i a Miben aktuális elemszáma}
        Mit:     TTElem;     {i a keresett érték}
  var   Hol:     TEIndex1):  {o a Mit helye}
                 Boolean;    {a Mit létezése a Miben}

{ Egy érték beszúrása adott helyre }
procedure BeSzur(
  var Mibe:   TSor;      {i/o ebbe szúrunk be}
  var Hanyba: TElemDb;   {i/o a Mibe aktuális elemszáma}
      Mit:    TTElem;    {i a beszúrandó érték}
      Hova:   TEIndex);  {i a beszúrás helye}
```

```
{ Egy elem törlése }
procedure Torol(
  var Mibol:   TSor;      {i/o ebből törlünk}
  var Hanybol: TElemDb;   {i/o a Mibol aktuális elemszáma}
      Honnan:  TEIndex);  {i a törlendő TTElem indexe}

{ Egy érték összes előfordulásának törlése }
procedure ErtekTorl(
  var Mibol:   TSor;     {i/o ebből törlünk}
  var Hanybol: TElemDb;  {i/o a Mibol aktuális elemszáma}
      Mit:     TTElem);  {i a törlendő érték}

{ Bármely érték közvetlen egymásutáni ismétlődéseinek törlése }
procedure DuplaTorl(
  var Mibol:   TSor;      {i/o ebből törlünk }
  var Hanybol: TElemDb);  {i/o a Mibol aktuális elemszáma}

implementation

function Keres(const Miben: TSor; Hanyban: TElemDb; Mit: TTElem;
  var Hol: TEIndex1): Boolean;
begin
  Hol:=1;
  while (Hol<=Hanyban) and (Mit<>Miben[Hol]) do Inc(Hol);
  Keres:=(Hol<=Hanyban) and (Mit=Miben[Hol]);
end;

procedure BeSzur(var Mibe: TSor; var Hanyba: TElemDb; Mit: TTElem;
  Hova: TEIndex);
var
  I: TEIndex;
begin
  for I:=Hanyba downto Hova do Mibe[I+1]:=Mibe[I];
  Mibe[Hova]:=Mit;
  Inc(Hanyba);
end;

procedure Torol(var Mibol: TSor; var Hanybol: TElemDb; Honnan: TEIndex);
var
  I: TEIndex;
begin
  for I:=Honnan to Hanybol-1 do Mibol[I]:=Mibol[I+1];
  Dec(Hanybol);
end;

procedure ErtekTorl(var Mibol: TSor; var Hanybol: TElemDb; Mit: TTElem);
var
  I, J: TEIndex0;
begin
  J:=0;
  for I:=1 to Hanybol do
    if Mibol[I]<>Mit then begin
      Inc(J);  Mibol[J]:=Mibol[I];
    end;
  Hanybol:=J;
end;
```

```
procedure DuplaTorl(var Mibol: TSor; var Hanybol: TElemDb);
var
  I: TEIndex1;  J: TEIndex0;
begin
  I:=1;  J:=0;
  while I<=Hanybol do begin
    Inc(J);
    Mibol[J]:=Mibol[I];  {egy példány átmásolása}
    {az ismétlődések átlépése}
    while (I<=Hanybol) and (Mibol[I]=Mibol[J]) do Inc(I);
  end;
  Hanybol:=J; {új elemszám}
end;
end.
```

uTombD unit

```
unit uTombD;

interface

uses uDinTomb;

{ Egy érték keresése }
function DKeres(
  const Miben: TDinTomb;     {i ebben keresünk}
        Mit:   TDElem;       {i a keresett érték}
  var   Hol:   TDEIndex1):   {o a Mit helye}
               Boolean;      {o a Mit létezése a Miben}

{ Egy érték beszúrása adott helyre }
function DBeSzur(
  var Mibe: TDinTomb;    {i/o ebbe szúrunk be}
      Mit:  TDElem;      {i a beszúrandó érték}
      Hova: TDEIndex):   {i a beszúrás helye}
            Boolean;     {o a művelet végrehajthatósága}

{ Egy elem törlése }
procedure DTorol(
  var Mibol:  TDinTomb;    {i/o ebből törlünk}
      Honnan: TDEIndex);   {i a törlendő TDElem indexe}

{ Rendezés mimimumkiválasztással }
procedure DKivalRend(var Adatok: TDinTomb);  {i/o a rendezendő tömb}

{ Statisztikai jellemzők }
procedure DStatJell(
      NTol, NIg:     TDEIndex;  {i elemszám értékhatárok}
      ETol, EIg:     TDElem;    {i elem értékhatárok}
  var N:             TDEDb;     {o sorsolt elemszám}
  var Min, Max, Med: TDElem);   {o minimum, maximum, medián}

implementation
```

```
function DKeres(const Miben: TDinTomb; Mit: TDElem; var Hol: TDEIndex1):
  Boolean;
begin
  Hol:=1;
  while (Hol<=DTEdb(Miben)) and (Mit<>DTErt(Miben, Hol)) do Inc(Hol);
  DKeres:=(Hol<=DTEdb(Miben)) and (Mit<>DTErt(Miben, Hol));
end;

function DBeSzur(var Mibe: TDinTomb; Mit: TDElem; Hova: TDEIndex): Boolean;
var
  I: TDEIndex;  Jo: Boolean;
begin
  Jo:=DTHossz(Mibe, DTEdb(Mibe)+1);
  if Jo then begin
    for I:=DTEdb(Mibe) downto Hova do DTBe(Mibe, I+1, DTErt(Mibe, I));
    DTBe(Mibe, Hova, Mit);
  end;
  DBeszur:=Jo;
end;

procedure DTorol(var Mibol: TDinTomb; Honnan: TDEIndex);
var
  I: TDEIndex;
begin
  for I:=Honnan to DTEdb(Mibol)-1 do
    DTBe(Mibol, I, DTErt(Mibol, I+1));
  DTHossz(Mibol, DTEdb(Mibol)-1);
end;

procedure DKivalRend(var Adatok: TDinTomb);
var
  SorKezd, MinHely, I: TDEIndex;  MinI: TDElem;
begin
  if DTEdb(Adatok)>1 then
    for SorKezd:=1 to DTEdb(Adatok)-1 do begin  {minimum meghatározása}
      MinHely:=SorKezd;  MinI:=DTErt(Adatok, SorKezd);
      for I:=SorKezd+1 to DTEdb(Adatok) do
        if DTErt(Adatok, I)<Mini then begin
          MinHely:=I;  MinI:=DTErt(Adatok, I);
        end;
      {csere}
      DTBe(Adatok, MinHely, DTErt(Adatok, SorKezd));
      DTBe(Adatok, SorKezd, Mini);
    end;
end;

procedure DStatJell(NTol, NIg: TDEIndex; ETol, EIg: TDElem; var N: TDEDb;
  var Min, Max, Med: TDElem);
var
  A: TDinTomb;  I: TDEIndex;
begin
  N:=Random(NTol-NIg+1)+NIg;
  DTIndit(A);
  if DTHossz(A, N) then begin
    for I:=1 to N do DTBe(A, I, Random(ETol-EIg+1)+EIg);
    DKivalRend(A);
    Min:=DTErt(A, 1);  Max:=DTErt(A, N);
```

```
    Med:=DTErt(A, (1+N) div 2);
  end else N:=0;
  DTZar(A);
end;
end.
```

uTombR unit

```
unit uTombR;

interface

uses uTomb;

{ Lineáris keresés }
function LinKer(
  const Miben:   TSor;       {i nemcsökkenően rendezett, ebben keresünk}
        Hanyban: TElemDb;    {i a Miben aktuális elemszáma}
        Mit:     TTElem;     {i a keresett érték}
  var   Hol:     TEIndex1):  {o a Mit helye}
                 Boolean;    {a Mit létezése a Miben}

{ Új érték beszúrása nemcsökkenő sorba }
procedure BeSzurR_L(
  var Mibe:   TSor;     {i/o ezt bővítjük}
  var Hanyba: TElemDb;  {i/o a Mibe aktuális elemszáma}
      Mit:    TTElem);  {i az új érték}

{ Egy érték első előfordulásának törlése nemcsökkenő sorból}
procedure TorolR_L(
  var Mibol:   TSor;     {i/o ebből törlünk}
  var Hanybol: TElemDb;  {i/o a Mibol aktuális elemszáma}
      Mit:     TTElem);  {i a törlendő érték}

{ Bináris keresés }
function BinKer(
  const Miben:   TSor;       {i nemcsökkenően rendezett, ebben keresünk}
        Hanyban: TElemDb;    {i a Miben aktuális elemszáma}
        Mit:     TTElem;     {i a keresett érték}
  var   Hol:     TEIndex1):  {o a Mit helye}
                 Boolean;    {o a Mit létezése a Miben}

{ Egy érték egy előfordulásának törlése nemcsökkenő sorból}
procedure TorolR_B(
  var Mibol:   TSor;     {i/o ebből törlünk}
  var Hanybol: TElemDb;  {i/o a Mibol aktuális elemszáma}
      Mit:     TTElem);  {i a törlendő érték}

{ Új érték beszúrása nemcsökkenő sorba }
procedure BeSzurR_B(
  var Mibe:   TSor;     {i/o ezt bővítjük}
  var Hanyba: TElemDb;  {i/o a Mibe aktuális elemszáma}
      Mit:    TTElem);  {i az új érték}
```

```
{ Rendezés a szomszédos elemek cseréjével }
procedure CserelRend(
  var Adatok: TSor;      {i/o a rendezendő tömb}
      AdatDb: TElemDb);  {i a tömb aktuális elemszáma}

{ Rendezés mimimumkiválasztással }
procedure KivalRend(
  var Adatok: TSor;      {i/o a rendezendő tömb}
      AdatDb: TElemDb);  {i a tömb aktuális elemszáma}

{ Rendezés beszúrással }
procedure BeSzurRend(
  var Adatok: TSor;      {i/o a rendezendő tömb}
      AdatDb: TElemDb);  {i a tömb aktuális elemszáma}

procedure OsszeVal(
  const A, B:   TSor;      {i kiinduló tömbök}
        NA, NB: TElemDb;   {i kiinduló tömbök elemszámai}
  var   C:      TSor;      {o eredmény tömb}
  var   NC:     TElemDb);  {o eredmény tömb aktuális elemszáma}

implementation

function LinKer(const Miben: TSor; Hanyban: TElemDb; Mit: TTElem;
  var Hol: TEIndex1): Boolean;
begin
  Hol:=1;
  while (Hol<=Hanyban) and (Mit>Miben[Hol]) do Inc(Hol);
  LinKer:=(Hol<=Hanyban) and (Mit=Miben[Hol]);
end;

procedure BeSzurR_L(var Mibe: TSor; var Hanyba: TElemDb; Mit: TTElem);
var
  I: TEIndex0;
begin
  I:=Hanyba;
  while (I>=1) and (Mit<Mibe[I]) do begin
   Mibe[I+1]:=Mibe[I];  I:=I-1;
  end;
  Mibe[I+1]:=Mit;  Hanyba:=Hanyba+1;
end;

procedure TorolR_L(var Mibol: TSor; var Hanybol: TElemDb; Mit: TTElem);
var
  Hol: TEIndex1;
begin
  if LinKer(Mibol, Hanybol, Mit, Hol) then Torol(Mibol, Hanybol, Hol);
end;

function BinKer(const Miben: TSor; Hanyban: TElemDb; Mit: TTElem;
  var Hol: TEIndex1): Boolean;
var
  Kezd: TEIndex1;  Veg: TEIndex0;  Van: Boolean;
begin
  Kezd:=1;  Van:=False;
  if Hanyban>0 then begin
```

```
    Veg:=Hanyban;
    repeat
      Hol:=(Kezd+Veg) div 2;  Van:=Mit=Miben[Hol];
      if not Van then
        if Mit<Miben[Hol] then Veg:=Hol-1 else Kezd:=Hol+1;
    until (Kezd>Veg) or Van;
  end;
  if not Van then Hol:=Kezd;
  BinKer:=Van;
end;

procedure TorolR_B(var Mibol: TSor; var Hanybol: TElemDb; Mit: TTElem);
var
  Hol: TEIndex1;
begin
  if BinKer(Mibol, Hanybol, Mit, Hol) then Torol(Mibol, Hanybol, Hol);
end;

procedure BeSzurR_B(var Mibe: TSor; var Hanyba: TElemDb; Mit: TTElem);
var
  Hol: TEIndex1;
begin
  BinKer(Mibe, Hanyba, Mit, Hol);  Beszur(Mibe, Hanyba, Mit, Hol);
end;

{-- egyszerű rendező eljárások --}

procedure CserelRend(var Adatok: TSor; AdatDb: TElemDb);
var
  SorVeg: TEIndex0;  I: TEIndex;  UjMenet: Boolean;  Csere: TTElem;
begin
  {előkészítés}
  SorVeg:=AdatDb;  UjMenet:=True;
  {cseremenetek}
  while UjMenet and (SorVeg>1) do begin
    UjMenet:=False;
    for I:=2 to SorVeg do if Adatok[I-1]>Adatok[I] then begin  {csere}
      Csere:=Adatok[I-1];
      Adatok[I-1]:=Adatok[I];  Adatok[I]:=Csere;
      UjMenet:=True
    end;
    Dec(SorVeg);  {új sorvég}
  end;
end;

procedure KivalRend(var Adatok: TSor; AdatDb: TElemDb);
var
  SorKezd, MinHely, I: TEIndex;  MinI: TTElem;
begin
  if AdatDb>1 then
    for SorKezd:=1 to AdatDb-1 do begin  {minimum meghatározása}
      MinHely:=SorKezd;  MinI:=Adatok[SorKezd];
      for I:=SorKezd+1 to AdatDb do if Adatok[I]<MinI then begin
        MinHely:=I;  MinI:=Adatok[I]
      end;
```

```
      {csere}
      Adatok[MinHely]:=Adatok[SorKezd];  Adatok[SorKezd]:=MinI;
    end;
end;

procedure BeSzurRend(var Adatok: TSor; AdatDb: TElemDb);
var
  I: TElemDb;
begin
  {I+1. elemet besoroljuk az előtte lévő I elemű részsorba}
  I:=1;
  while I<AdatDB do BeSzurR_B(Adatok, I, Adatok[I+1]);
end;

procedure OsszeVal(const A, B: TSor; NA, NB: TElemDb; var C: TSor;
  var NC: TElemDb);
var
  IA, IB, I: TEIndex;  IC: TEIndex0;  AVege, BVege: Boolean;
begin
  IA:=1;  IB:=1;  IC:=0;  AVege:=IA>NA;  BVege:=IB>NB;
  while not (AVege or BVege) do begin
    Inc(IC);
    if A[IA]<B[IB] then begin
      C[IC]:=A[IA];  Inc(IA);  AVege:=IA>NA;
    end else begin
      C[IC]:=B[IB];  Inc(IB);  BVege:=IB>NB;
    end;
  end;
  if AVege then for I:=IB to NB do begin
    Inc(IC);  C[IC]:=B[I]
  end;
  if BVege then for I:=IA to NA do begin
    Inc(IC);  C[IC]:=A[I]
  end;
  NC:=IC;
end;
end.
```

uTombRI unit

```
unit uTombRI;

interface

uses uTomb;

type
  TEIndexSor=array[TEIndex] of TEIndex;

{ Rendezés a szomszédos elemek cseréjével }
procedure CserelRendIx(
  const Adatok: TSor;         {i az adatok tömbje}
        AdatDb: TElemDb;      {i a tömb aktuális elemszáma}
  var   Mutat:  TEIndexSor);  {o a rendezettséget leíró indextábla}
```

```
{ Rendezés mimimumkiválasztással }
procedure KivalRendIx(
  const Adatok: TSor;          {i az adatok tömbje}
        AdatDb: TElemDb;       {i a tömb aktuális elemszáma}
  var   Mutat:  TEIndexSor);   {o a rendezését leíró indextábla}

{ Bináris keresés indextáblával }
function BinKerIx(
  const Miben:   TSor;         {i az adatok tömbje}
        Hanyban: TElemDb;      {i a Miben aktuális elemszáma}
        Mit:     TTElem;       {i a keresett érték}
  var   Hol:     TEIndex1;     {o a Mit indexének helye a Mutat-ban}
  const Mutat:   TEIndexSor):  {i a Miben rendezését leíró indextábla}
                 Boolean;      {o a Mit létezése a Miben}

procedure BeSzurIx(
  var Mibe:   TSor;          {i/o a tömb, amibe beszúrunk}
  var Hanyba: TElemDb;       {i/o a Mibe aktuális elemszáma}
      Mit:    TTElem;        {i az új érték}
  var Mutat:  TEIndexSor);   {i/o a Mibe rendezését leíró indextábla}

implementation

procedure CserelRendIx(const Adatok: TSor; AdatDb: TElemDb;
  var Mutat: TEIndexSor);
var
  SorVeg: TEIndex0;  I: TEIndex;  UjMenet: Boolean;  Csere: TEIndex;
begin
  {előkészítés}
  SorVeg:=AdatDb;  UjMenet:=True;
  for I:=1 to AdatDb do Mutat[I]:=I;
  {cseremenetek}
  while UjMenet and (SorVeg>1) do begin
    UjMenet:=False;
    for I:=2 to SorVeg do
      if Adatok[Mutat[I-1]]>Adatok[Mutat[I]] then begin  {csere}
        Csere:=Mutat[I-1];  Mutat[I-1]:=Mutat[I];
        Mutat[I]:=Csere;  UjMenet:=True
      end;
    Dec(SorVeg);  {új sorvég}
  end;
end;

procedure KivalRendIx(const Adatok: TSor; AdatDb: TElemDb;
  var Mutat: TEIndexSor);
var
  SorKezd, MinHely, MinI: TEIndex;  I: TEIndex0;
begin
  for I:=1 to AdatDb do Mutat[I]:=I;
  if AdatDb>1 then
    for SorKezd:=1 to AdatDb-1 do begin
      MinHely:=SorKezd;  MinI:=Mutat[SorKezd];
      for I:=SorKezd+1 to AdatDb do
        if Adatok[Mutat[I]]<Adatok[MinI] then begin
          MinHely:=I;  MinI:=Mutat[I];
        end;
```

```
      {csere}
      Mutat[MinHely]:=Mutat[SorKezd];  Mutat[SorKezd]:=MinI;
    end;
end;

{-- bináris keresés és beszúrás indextáblával --}

function BinKerIx(const Miben: TSor; Hanyban: TElemDb; Mit: TTElem;
  var Hol: TEIndex1; const Mutat: TEIndexSor): Boolean;
var
  Kezd: TEIndex1;  Veg: TEIndex0;  Van: Boolean;
begin
  Kezd:=1;  Van:=False;
  if Hanyban>0 then begin
    Veg:=Hanyban;
    repeat
      Hol:=(Kezd+Veg) div 2;  Van:=Mit=Miben[Mutat[Hol]];
      if not Van then
        if Mit<Miben[Mutat[Hol]] then Veg:=Hol-1 else Kezd:=Hol+1;
    until (Kezd>Veg) or Van;
  end;
  if not Van then Hol:=Kezd;
  BinKerIx:=Van;
end;

procedure BeSzurIx(var Mibe: TSor; var Hanyba: TElemDb; Mit: TTElem;
  var Mutat: TEIndexSor);
var
  Hol: TEIndex1;  I: TEIndex0;
begin
 {az új TTElem a tömb végére megy, indexét a helyére soroljuk}
  Mibe[Hanyba+1]:=Mit;
  BinKerIx(Mibe, Hanyba, Mit, Hol, Mutat);
  for I:=Hanyba downto Hol do Mutat[I+1]:=Mutat[I];
  Mutat[Hol]:=Hanyba+1;  Inc(Hanyba);
end;
end.
```

uVerem unit

```
unit uVerem;

interface

uses uTomb, uListaD, uLista;

{ Hatvány }
function Hatvany(N: Word; K: Byte): Longint;  {az N K-adik hatványa}

{ Gyorsrendezés, rekurzív hívás }
procedure QuickSort_R(
  var Adatok: TSor;      {i/o a rendezendő tömb}
      AdatDb: TElemDb);  {i a tömb aktuális elemszáma}
```

```
{ Gyorsrendezés, saját veremkezelés }
procedure QuickSort_N(
  var Adatok: TSor;      {i/o a rendezendő tömb}
      AdatDb: TElemDb);  {i a tömb aktuális elemszáma}

{ Permutáció }
type
  TPerm=String;
  TPermDb=0..MaxLongint;
  TPermH=Byte; {hossz}
  TPermLista=PLElem1;

function Permutal(
      Szo:    TPerm;       {i a permutálandó szó}
  var Szavak: TPermLista;  {o a permutációk listája}
  var SzoDb:  TPermDb):    {o a permutációk száma}
              Boolean;     {o a lista létrejött-e (hely)}

implementation

function Hatvany(N: Word; K: Byte): Longint;
begin
  if K=0 then Hatvany:=1
  else if Odd(K) then Hatvany:=N*Hatvany(N, K-1)
  else Hatvany:=Sqr(Hatvany(N, K div 2));
end;

function Permutal(Szo: TPerm; var Szavak: TPermLista;
  var SzoDb: TPermDb): Boolean;
var
  SzavakKezd, SzavakVeg, MunkaKezd, MunkaVeg: PLElem1;

  function Permutal_R(N: TPermH): Boolean;
  var
    PDb: TPermDb;  J: TPermH;  P: TPerm;  KovJel: Char;
    Akt: PLElem1;  Jo: Boolean;
  begin
    if N=0 then
      Jo:=False
    else if N=1 then begin
      SzoDb:=1;
      Szavak:=LUjElore1(SzavakKezd, SzavakVeg, Szo[1]);
      Jo:=Szavak<>nil;
    end else begin
      Jo:=Permutal_R(N-1);
      if Jo then begin
        MunkaKezd:=SzavakKezd;  MunkaVeg:=SzavakVeg;
        LInit1(SzavakKezd, SzavakVeg);
        PDb:=0;  KovJel:=Szo[N];  Akt:=MunkaKezd;
        while (Akt<>nil) and Jo do begin
          for J:=1 to N do begin
            P:=Akt^.Adat;  Insert(KovJel, P, J);  Inc(PDb);
            Szavak:=LUjElore1(SzavakKezd, SzavakVeg, P);
            Jo:=Szavak<>nil;
          end;
          Akt:=Akt^.Koveto;
```

```
        end;
      end;
      SzoDb:=PDb;  LFelsz1(MunkaKezd);
    end;
    Permutal_R:=Jo;
  end;

begin
  LInit1(SzavakKezd, SzavakVeg);
  Permutal:=Permutal_R(Length(Szo));
end;

procedure QuickSort_R(var Adatok: TSor; AdatDb: TElemDb);

  procedure QSort(BHI, JHI: TEIndex);
  var
    KE: TTElem;  BI, JI: TEIndex;

    procedure Csere(I, J: TEIndex);
    var
      Cs: TTElem;
    begin
      Cs:=Adatok[I];  Adatok[I]:=Adatok[J];  Adatok[J]:=Cs;
    end;

  begin
    if JHI=BHI+1 then begin
      {triviális feladat, közvetlen csere}
      if Adatok[BHI]>Adatok[JHI] then Csere(BHI, JHI);
    end else begin  {továbbosztandó feladat}
      {középérték kijelölés}
      KE:=Adatok[(BHI+JHI) div 2];
      {felosztás}
      BI:=BHI;  JI:=JHI;
      while BI<=JI do begin  {párkeresés}
        while Adatok[BI]<KE do BI:=BI+1;  {balról}
        while Adatok[JI]>KE do JI:=JI-1;  {jobbról}
        if BI<=JI then begin
          Csere(BI, JI);  JI:=JI-1;  BI:=BI+1;
        end;
      end;
      {új feladatok, elsőként a kisebb rész}
      if (JI-BHI)<(JHI-BI) then begin
        if BHI<JI then QSort(BHI, JI);
        if BI<JHI then QSort(BI, JHI)
      end else begin
        if BI<JHI then QSort(BI, JHI);
        if BHI<JI then QSort(BHI, JI)
      end;
    end;
  end;

begin
  if AdatDb>1 then QSort(1, AdatDb);
end;
```

```
procedure QuickSort_N(var Adatok: TSor; AdatDb: TElemDb);
const
  VeremMaxHossz=7;  {log2(EMaxDb)}
type
  TVeremInd=1..VeremMaxHossz;  TVeremMutato=0..VeremMaxHossz;
  TVeremTomb=array[TVeremInd] of record
    BI, JI: TEIndex
  end;
  TVerem=record
    VeremT: TVeremTomb;
    VeremM: TVeremMutato;
  end;
var
  KE: TTElem;  BHI, JHI, BI, JI: TEIndex;  Verem: TVerem;

  procedure VeremInit;
  begin
    Verem.VeremM:=0;
  end;

  procedure VeremBe(BI, JI: TEIndex);
  begin
    with Verem do begin
      Inc(VeremM);
      VeremT[VeremM].BI:=BI;  VeremT[VeremM].JI:=JI;
    end;
  end;

  procedure VeremBol;
  begin
    with Verem do Dec(VeremM);
  end;

  procedure Csere(I, J: TEIndex);
  var
    Cs: TTElem;
  begin
    Cs:=Adatok[I];  Adatok[I]:=Adatok[J];  Adatok[J]:=Cs;
  end;

begin
  {előkészítés}
  VeremInit;
  if AdatDb>1 then VeremBe(1, AdatDb);
  while Verem.VeremM>0 do begin
    {feladatkijelölés a verem tetejéről}
    with Verem do begin
      BHI:=VeremT[VeremM].BI;  JHI:=VeremT[VeremM].JI;
    end;
    if JHI=BHI+1 then begin
      {triviális feladat, közvetlen csere}
      if Adatok[BHI]>Adatok[JHI] then Csere(BHI, JHI);
      VeremBol;  {törlés a veremből}
    end else begin  {továbbosztandó feladat, középérték kijelölés}
      KE:=Adatok[(BHI+JHI) div 2];
```

```
      {felosztás}
      BI:=BHI;  JI:=JHI;
      while BI<=JI do begin  {párkeresés}
        while Adatok[BI]<KE do BI:=BI+1;  {balról}
        while Adatok[JI]>KE do JI:=JI-1;  {jobbról}
        if BI<=JI then begin
          Csere(BI, JI);  JI:=JI-1;  BI:=BI+1
        end;
      end;
      VeremBol;  {törlés a veremből}
      {felvétel a verembe, a kisebb rész a tetejére}
      if (JI-BHI)<(JHI-BI) then begin
        if JHI>BI then VeremBe(BI, JHI);
        if JI>BHI then VeremBe(BHI, JI);
      end else begin
        if JI>BHI then VeremBe(BHI, JI);
        if JHI>BI then VeremBe(BI, JHI);
      end
    end;
  end;
end;
end.
```

Tesztprogramok

DBinfa

DFFoprog.dfm

```
object Forma: TForma
  Left=380
  Top=132
  Width=544
  Height=375
  Caption='Forma'
  Color=clBtnFace
  Font.Charset=DEFAULT_CHARSET
  Font.Color=clWindowText
  Font.Height=-11
  Font.Name='MS Sans Serif'
  Font.Style=[]
  Menu=MFoMenu
  OldCreateOrder=False
  OnCreate=FormCreate
  PixelsPerInch=96
  TextHeight=13
  object Memo: TMemo
    Left=0
    Top=0
    Width=536
    Height=329
    Align=alClient
    Font.Charset=EASTEUROPE_CHARSET
    Font.Color=clWindowText
    Font.Height=-13
    Font.Name='Courier New'
    Font.Style=[fsBold]
    ParentFont=False
    ScrollBars=ssBoth
    TabOrder=0
    WordWrap=False
  end
  object MFoMenu: TMainMenu
    Left=96
    Top=56
    object MEloallit: TMenuItem
      Caption='Előállítás'
      OnClick=MEloallitClick
    end
    object MKiir: TMenuItem
      Caption='Kiírás'
      OnClick=MKiirClick
    end
    object MFelvesz: TMenuItem
      Caption='Felvétel'
      OnClick=MFelveszClick
    end
```

```
    object MTorol: TMenuItem
      Caption='Törlés'
      OnClick=MTorolClick
    end
    object MKeres: TMenuItem
      Caption='Keresés'
      OnClick=MKeresClick
    end
    object MKilep: TMenuItem
      Caption='Kilépés'
      OnClick=MKilepClick
    end
  end
end
```

DFFoprog

```
unit DFFoprog;

interface

uses Windows, Messages,SysUtils,Classes,Graphics, Controls, Forms, Dialogs,
  Menus, StdCtrls, uGrafD, dGBinFa, dKieg, dInpOut;

const
  FejSzov='Bináris keresőfa';  SzovFajl='FAAdat.txt';

type
  TForma=class(TForm)
    MFoMenu: TMainMenu;
    MEloallit: TMenuItem;
    MKiir: TMenuItem;
    MFelvesz: TMenuItem;
    MTorol: TMenuItem;
    MKeres: TMenuItem;
    MKilep: TMenuItem;
    Memo: TMemo;
    procedure FormCreate(Sender: TObject);
    procedure MEloallitClick(Sender: TObject);
    procedure MKiirClick(Sender: TObject);
    procedure MFelveszClick(Sender: TObject);
    procedure MTorolClick(Sender: TObject);
    procedure MKilepClick(Sender: TObject);
    procedure MKeresClick(Sender: TObject);
  private
    Fa: PBinFaPont;  {fa}
    N: Byte;
    {segédváltozók}
    Lista: Text;
    Az: TBinfaAzon;
    Ad: TBinfaAdat;
  public
  end;

var
  Forma: TForma;
```

```
implementation

{$R *.DFM}

procedure TForma.FormCreate(Sender: TObject);
begin
  Caption:=FejSzov;
  Randomize;
  BinFaInit(Fa);
  AssignFile(Lista, Szovfajl);
end;

procedure TForma.MEloallitClick(Sender: TObject);
var
  X: Integer;
begin
  MemoSorIr(Memo,'Előállítás');  X:=1;
  if EgSzamBe('Elemszám:', X, 1, MaxSorsDb) then begin
    BinFaTorol(Fa);  Sorsol(Fa, X, N);
  end;
end;

procedure TForma.MKiirClick(Sender: TObject);
var
  S: String;
begin
  MemoSorIr(Memo, 'Kiírás');
  Rewrite(Lista);
  BinFaListaNo(Lista, Fa);
  CloseFile(Lista);  Reset(Lista);
  while not Eof(Lista) do begin
    Readln(Lista, S);  MemoSorIr(Memo, '*'+S);
  end;
  CloseFile(Lista);
end;

procedure TForma.MFelveszClick(Sender: TObject);
var
  S: String;
begin
  MemoSorIr(Memo, 'Felvétel');
  if Azonbe('Új', Az) and Adatbe('Új', Ad) then begin
    S:=Az+'-'+Ad;
    if BinFaRa(Fa, Az, Ad)<>nil then MemoSorIr(Memo, S+'Felvéve')
    else MemoSorIr(Memo, Az+': Ismételt azonosító vagy helyhiány');
  end;
end;

procedure TForma.MTorolClick(Sender: TObject);
begin
  MemoSorIr(Memo, 'Törlés');
  if Adatbe('Törlendő', Ad) then begin
    if BinFarol(Fa, Ad) then MemoSorIr(Memo, Ad+': adat törölve')
    else MemoSorIr(Memo, Ad+': nem volt ilyen adat');
  end;
end;
```

```
procedure TForma.MKeresClick(Sender: TObject);
var
  P, PSz: PBinFaPont;  Bal: Boolean;
begin
  MemoSorIr(Memo, 'Keresés');
  if Adatbe('Keresett', Ad) then begin
    P:=BinFanAdatKeres(Fa, Ad, PSz, Bal);
    if P<>nil then
      MemoSorIr(Memo, Ad+': megvan, azonosítója: '+P^.Azon)
    else MemoSorIr(Memo, Ad+': nincs ilyen adat');
  end;
end;

procedure TForma.MKilepClick(Sender: TObject);
begin
  BinFatorol(Fa);  Close;
end;
end.
```

dKieg

```
unit dKieg;

interface

uses uDef, dInpOut, dGrafU, uString, uGrafD, dGBinfa;

const
  MaxSorsDb=100;  MaxAzHossz=5;  MaxAdHossz=5;

procedure Sorsol(var Fa: PBinFaPont; Db: Byte; var N: Byte);
function AzonBe(FSzov: String; var Az: TBinfaAzon): Boolean;
function AdatBe(FSzov: String; var Ad: TBinfaAdat): Boolean;

implementation

function SorsKisBetu:char;
begin
  SorsKisbetu:=Chr(Random(Ord('z')-Ord('a'))+Ord('a'));
end;

procedure Sorsol(var Fa: PBinFaPont; Db: Byte; var N: Byte);
var
  Az: TBinfaAzon;  Ad: TBinfaAdat;  Jo: Boolean;
begin
  N:=1;
  repeat
    Str(N, Az);
    Az:=ElolNull(Az, 3);  Ad:=SorsKisbetu+SorsKisbetu+SorsKisbetu;
    Jo:=BinFaRa(Fa, Az, Ad)<>nil;
    Inc(N);
  until (not Jo) or (N=Db+1);
  Dec(N);
end;
```

```
function AzonBe(FSzov: String; var Az: TBinfaAzon): Boolean;
begin
  Az:='';
  AzonBe:=AltSzovbe(FSzov+' Azonosító', Az, SzamJegyek, 1, MaxAzHossz,
    [], 0, 0)
end;

function AdatBe(FSzov: String; var Ad: TBinfaAdat): Boolean;
begin
  Ad:='';
  AdatBe:=AltSzovbe(FSzov+' Adat', Ad, KisBetuk, 1, MaxAdHossz,
    [], 0, 0)
end;
end.
```

DRendKer

DFFoprog.dfm

```
object Forma: TForma
  Left=431
  Top=172
  Width=544
  Height=375
  Caption='Forma'
  Color=clBtnFace
  Font.Charset=DEFAULT_CHARSET
  Font.Color=clWindowText
  Font.Height=-11
  Font.Name='MS Sans Serif'
  Font.Style=[]
  Menu=MFoMenu
  OldCreateOrder=False
  OnCreate=FormCreate
  PixelsPerInch=96
  TextHeight=13
  object Memo: TMemo
    Left=0
    Top=0
    Width=536
    Height=329
    Align=alClient
    Font.Charset=EASTEUROPE_CHARSET
    Font.Color=clWindowText
    Font.Height=-13
    Font.Name='Courier New'
    Font.Style=[fsBold]
    ParentFont=False
    ScrollBars=ssBoth
    TabOrder=0
    WordWrap=False
  end
  object MFoMenu: TMainMenu
    Left=96
    Top=56
```

```
    object MEloallit: TMenuItem
      Caption='Előállítás'
      OnClick=MEloallitClick
    end
    object MKiir: TMenuItem
      Caption='Kiírás'
      OnClick=MKiirClick
    end
    object MRendez: TMenuItem
      Caption='Rendezés'
      OnClick=MRendezClick
    end
    object MKeres: TMenuItem
      Caption='Keresés'
      OnClick=MKeresClick
    end
    object MKilep: TMenuItem
      Caption='Kilépés'
      OnClick=MKilepClick
    end
  end
end
```

DFFoprog

```
unit DFFoprog;

interface

uses Windows, Messages,SysUtils,Classes,Graphics, Controls, Forms, Dialogs,
  Menus, uTomb, uTombR, StdCtrls, dInpOut, dKieg, uString;

const
  FejSzov='Tömbrendezés és keresés';

type
  TForma=class(TForm)
    MFoMenu: TMainMenu;
    MEloallit: TMenuItem;
    MKiir: TMenuItem;
    MRendez: TMenuItem;
    MKeres: TMenuItem;
    MKilep: TMenuItem;
    Memo: TMemo;
    procedure MKilepClick(Sender: TObject);
    procedure FormCreate(Sender: TObject);
    procedure MEloallitClick(Sender: TObject);
    procedure MKiirClick(Sender: TObject);
    procedure MRendezClick(Sender: TObject);
    procedure MKeresClick(Sender: TObject);
  private
    {tömb}
    Sor: TSor;
    N: TElemDb;
    {segédváltozó}
    Rendezett: Boolean;
```

```
    procedure FejlecIr;
  public
  end;

var
  Forma: TForma;

implementation

{$R *.DFM}

procedure TForma.FormCreate(Sender: TObject);
begin
  Caption:=FejSzov;
  Rendezett:=False;
  Randomize;
  MKiir.Enabled:=False;  MRendez.Enabled:=False;
  MKeres.Enabled:=False;
end;

procedure TForma.MEloallitClick(Sender: TObject);
var
  X: Integer;
  Tol, Ig: Longint;
begin
  X:=1;  Tol:=0;  Ig:=0;
  if EgSzambe('Elemszám', X, 1, EMaxDb)
    and Egszambe('Tól', Tol, -Maxint, Maxint)
    and Egszambe('Ig', Ig, Tol, Maxint) then
  begin
    N:=X;
    Sorsol(N, Sor, Tol, Ig);
    Rendezett:=False;
    FejlecIr;
    MKiir.Enabled:=True;  MRendez.Enabled:=True;
    MKeres.Enabled:=True;
  end;
end;

procedure TForma.MKiirClick(Sender: TObject);
var
  SIndex, SAdat: String;  I: TEIndex;
begin
  SIndex:='';  SAdat:='';
  for I:=1 to N do begin
    SIndex:=SIndex+BalTolt(IntToStr(I), 8, '-');
    SAdat:=SAdat+BalTolt(IntToStr(Sor[I]), 8, ' ');
    if (I mod 10=0) or (I=N) then begin
      MemoSorIr(Memo, SIndex);  SIndex:='';
      MemoSorIr(Memo, SAdat);  SAdat:='';
    end;
  end;
  MemoSorIr(Memo, '**********');
end;
```

```
procedure TForma.MRendezClick(Sender: TObject);
begin
  KivalRend(Sor, N);
  Rendezett:=True;  MRendez.Enabled:=False;
  FejlecIr;
end;

procedure TForma.MKeresClick(Sender: TObject);
var
  X: Integer;  S: String;  Hol: TEIndex1;
begin
  X:=0;
  if EgSzambe('Keresett', X, -Maxint, Maxint) then begin
    S:=IntToStr(X);
    if Rendezett then begin
      if BinKer(Sor, N, X, Hol) then
        S:=S+' megvan, indexe: '+IntToStr(Hol)
      else S:=S+' nincs meg, helye: '+IntToStr(Hol);
    end else begin
      if Keres(Sor, N, X, Hol) then
        S:=S+' megvan, indexe: '+IntToStr(Hol)
      else S:=S+' nincs meg';
    end;
    MemoSorIr(Memo, S);
  end;
end;

procedure TForma.FejlecIr;
begin
  Caption:=FejSzov+' Elemszám: '+IntToStr(N)+RendSzov[Rendezett];
end;

procedure TForma.MKilepClick(Sender: TObject);
begin
  Close;
end;
end.
```

dKieg

```
unit dKieg;

interface

uses dInpOut, uTomb;

const
  RendSzov: array[Boolean] of String=(' Nem rendezett', ' Rendezett');

procedure Sorsol(Db: TElemDb; var Sor: TSor; Tol, Ig: TTElem);

implementation
```

```
procedure Sorsol(Db: TElemDb; var Sor: TSor; Tol, Ig: TTElem);
var
  I: TEIndex;
begin
  for I:=1 to Db do Sor[I]:=Tol+Random(Ig-Tol+1);
end;
end.
```

TBinfa

TFoProg

```
program TFoProg;

uses Crt, tDef, tKom, tInpA, uGrafD, tGBinfa, tKieg;

const
  MenuDb=5;
  Menu: MenuSorok=('Funkciók', 'Előállítás', 'Kiírás', 'Felvétel',
    'Törlés', 'Keresés', '', '', '', '');
  FejSzov='Bináris keresőfa';
  SzovFajl='FAAdat.txt';

var
  M: MenuInd;
  Fa:PBinFaPont;  {fa}
  {segédváltozók}
  Lista: Text;
  N: Byte;
  Bal: Boolean;
  X: Longint;
  Az: TBinfaAzon;
  Ad: TBinfaAdat;
  S: String;
  Oszl: Byte;
  P, PSz: PBinFaPont;

begin
  ClrScr;
  Fejlec(FejSzov, '', '');
  Assign(Lista, SzovFajl);
  BinFaInit(Fa);
  Randomize;
  repeat
    M:=FoMenu(Menu, MenuDb);
    if M>0 then begin
      Fejlec(FejSzov, Menu[M], '');
      MunkaTerTorl;
      FaKiir(Lista, Fa);
    end;
    case M of
      1: begin
        Write('Elemszám: ');
        S:='';
        if EgSzamBe(S, 0, 0, 1, MaxSorsDb, X) then begin
          BinFaTorol(Fa);
```

```
          Sorsol(Fa, X, N);
        end;
      end;
      2: Tovabb('');
      3: begin
        Write('Új azonosító, új adat: ');
        Oszl:=Wherex;
        if Azonbe(Az, 0, 0) and Adatbe(Ad, Oszl+MaxAzHossz+2, 0) then
        begin
          if BinFaRa(Fa, Az, Ad)<>nil then begin
            FaKiir(Lista, Fa);
            Tovabb('Felvéve');
          end else Tovabb('Ismételt azonosító vagy helyhiány');
        end;
      end;
      4: begin
        Write('Törlendő adat: ');
        if AdatBe(Ad,0,0) then begin
          if BinFarol(Fa, Ad) then begin
            FaKiir(Lista, Fa);
            Tovabb('Törölve');
          end else Tovabb('Nem volt ilyen adat');
        end;
      end;
      5: begin
        Write('Keresett adat: ');
        if AdatBe(Ad, 0, 0) then begin
         P:=BinFanAdatKeres(Fa, Ad, PSz, Bal);
         if P<>nil then
           Tovabb('Megvan, azonosítója: '+P^.Azon)
         else Tovabb('Nincs ilyen adat');
        end;
      end;
    end;
  until M=0;
  BinFaTorol(Fa);
end.
```

tKieg

```
unit tKieg;

interface

uses tDef, tInpA, tGrafU, uString, uGrafD, tGBinfa;

const
  MaxSorsDb=100;  MaxAzHossz=5;  MaxAdHossz=5;

procedure Sorsol(var Fa: PBinFaPont; Db: Byte; var N: Byte);
function AzonBe(var Az: TBinfaAzon; Oszl, Sor: Byte): Boolean;
function AdatBe(var Ad: TBinfaAdat; Oszl, Sor: Byte): Boolean;
procedure FaKiir(var Lista: Text; Fa: PBinFaPont);

implementation
```

```
function SorsKisBetu: Char;
begin
  SorsKisbetu:=Chr(Random(Ord('z')-Ord('a'))+Ord('a'));
end;

procedure Sorsol(var Fa: PBinFaPont; Db: Byte; var N: Byte);
var
  Az: TBinfaAzon;  Ad: TBinfaAdat;  Jo: Boolean;
begin
  N:=1;
  repeat
    Str(N, Az);
    Az:=ElolNull(Az, 3);  Ad:=SorsKisbetu+SorsKisbetu+SorsKisbetu;
    Jo:=BinFaRa(Fa, Az, Ad)<>nil;  Inc(N);
  until (not Jo) or (N=Db+1);
  Dec(N);
end;

function AzonBe(var Az:TBinfaAzon;Oszl,Sor:byte):boolean;
begin
  Az:='';
  AzonBe:=AltSzovbe(Az, Oszl, Sor, SzamJegyek, 1, MaxAzHossz, [], 0, 0);
end;

function AdatBe(var Ad: TBinfaAdat; Oszl, Sor: Byte): Boolean;
begin
  Ad:='';
  AdatBe:=AltSzovbe(Ad, Oszl, Sor, KisBetuk, 1, MaxAdHossz, [], 0, 0);
end;

procedure FaKiir(var Lista:text;Fa:PBinFaPont);
var
  S: String;
begin
  Rewrite(Lista);  BinFaListaNo(Lista, Fa);  Close(Lista);
  Reset(Lista);  WriteLn;
  while not Eof(Lista) do begin
    ReadLn(Lista, S);  Write('*', S);
  end;
  WriteLn;  Close(Lista);
end;
end.
```

TKomInp

TFoProg

```
program TFoProg;

uses Crt, tDef, tKom, tKieg, uString;

var
  Cikk: TCikk;  VanNev, VanEar, VanDb, VanEredm: Boolean;
  TeljO, Ert: Real;  S: String;  M, Kezd: MenuInd;
```

```
const
  {adatok megnevezésének és értékének helye a képernyőn}
  MegnO=2;  AdatO=15;  NevC=10;  EArC=NevC+2;  DbC=EArC+2;
  ErtC=DbC+2;
  {menü helye a képernyőn}
  MeKO=60;  MeVO=MeKo+15;  MeKS=NevC-3;  MeVS=MeKs+10;
  MenuDb=4;
  {menüsorok}
  Menu: MenuSorok=('Egy árucikk', 'Név', 'Egységár', 'Darabszám',
    'Érték', '', '', '', '', '');

begin
  ClrScr;  Fejlec('Leltár', '', '');
  with Cikk do begin
    Nev:='';  Db:=DbMin;  EAr:=EArMin;
  end;
  TeljO:=0;
  repeat
    VanNev:=False;  VanDb:=False;  VanEredm:=False;  Kezd:=1;
    repeat
      M:=AblMenu(Menu, MenuDb, Kezd, MeKO, MeKS, MeVO, MeVS);
      case M of
        1: begin
          Ir(Menu[1]+':', MegnO, NevC);
          VanNev:=NevBe(Cikk.Nev, AdatO, NevC);
        end;
        2: if VanNev then begin
          Ir(Menu[2]+':', MegnO, EArC);
          VanEAr:=EArBe(Cikk.EAr, AdatO, EarC);
        end else begin
          Tovabb('Nincs név!');  Kezd:=1;
        end;
        3: if VanNev then begin
          Ir(Menu[3]+':', MegnO, DbC);
          VanDb:=DbBe(Cikk.Db, AdatO, DbC);
        end else begin
          Tovabb('Nincs név!');  Kezd:=1;
        end;
        4: begin
          VanEredm:=VanNev and VanEAr and VanDb;
          if VanEredm then begin
            with Cikk do Ert:=EAr*Db;
            TeljO:=TeljO+Ert;
            Str(Ert:2*EArMaxHossz:2, s);
            Ir(Menu[4]+':', MegnO, ErtC);  Ir(BalVag(S), AdatO, ErtC);
            Tovabb('');
          end else Tovabb('Hiányos adatok!')
        end;
      end;
    until VanEredm or (M=0);
    MunkaTerTorl;
  until not IgenVal('Folytatás (i/n)? ');
  Str(TelJo:2*EArMaxHossz:2, S);
  MunkaTerKezd;  WriteLn;  Ir('Teljes összeg: '+BalVag(S), 2, 0);
  Tovabb('Kész!');
end.
```

tKieg

```
unit tKieg;

interface

uses tDef, tInpA, uString;

const
  NevHosszMax=30;  NevElvJelMin=0;   NevElvJelMax=5;  EArMin=0.1;
  EArMax=5000;  EArMaxHossz=7;  DbMin=1;  DbMax=1000;

type
  TNev=String[NevHosszMax];
  TDb=DbMin..DbMax;
  TCikk=record
    Nev: TNev;
    Db: TDb;
    EAr: Real;
  end;

function NevBe(var Nev: TNev; Oszl, Sor: Byte): Boolean;
function EArBe(var EAr: Real; Oszl, Sor: Byte): Boolean;
function DbBe(var Db: TDb; Oszl, Sor: Byte): Boolean;

implementation

function NevBe(var Nev: TNev; Oszl, Sor: Byte): Boolean;
var
  S: String;  Jo: Boolean;
begin
  S:=Nev;
  Jo:=AltSzovBe(S, Oszl, Sor, Betuk, 1, NevHosszMax, KozJelek,
    NevElvJelMin, NevElvJelMax);
  if Jo then Nev:=S;
  NevBe:=Jo;
end;

function EArBe(var EAr: Real; Oszl, Sor: Byte): Boolean;
var
  S: String;
begin
  if EAr=0 then S:='' else Str(EAr:EArMaxHossz:2, S);
  S:=BalVag(S);
  EArBe:=ValSzamBe(S, Oszl, Sor, EArMin, EArMax, EArMaxHossz, EAr);
end;

function DbBe(var Db: TDb; Oszl, Sor: Byte): Boolean;
var
  S: String;  Jo: Boolean;  X: Longint;
begin
  Str(Db, S);  Jo:=EgSzamBe(S, Oszl, Sor, DbMin, DbMax, X);
  if Jo then Db:=X;
  DbBe:=Jo;
end;
end.
```

TRendKer

TFoProg

```
program TFoProg;

uses Crt, tDef, tKom, tInpA, uTomb, uTombR, tKieg;

{menü}
const
  MenuDb=4;
  Menu: MenuSorok=('Funkciók', 'Előállítás', 'Kiírás', 'Rendezés',
    'Keresés', '', '', '', '', '');
  FejSzov='Tömbrendezés és keresés';
  RendSzov: array[Boolean] of String=(' Nem rendezett', ' Rendezett');

var
  M: MenuInd;

{tömb}
var
  Sor: TSor;  N: TElemDb;
  {segédváltozók}
  VanSor, Rendezett: Boolean;
  S, SN: String;  X: Longint;  Tol, Ig: TTElem;  Hol: TEIndex1;  KSor: Byte;

begin
  ClrScr;
  Fejlec(FejSzov, '', '');
  VanSor:=False;
  Rendezett:=False;
  Randomize;
  repeat
    M:=FoMenu(Menu, MenuDb);
    case M of
      1: begin
        Fejlec(FejSzov, 'Elemszámbekérés', '');
        MunkaTerTorl;
        Write('Elemszám: ');
        KSor:=WhereY;  S:='';
        if EgSzamBe(S, 0, 0, 1, EMaxDb, X) then begin
          Ir('Tól: ', 1, KSor+2);
          S:='';  Str(N, SN);  N:=X;
          if Egszambe(S, 0, 0, -MaxInt, MaxInt, X) then begin
            Ir('Ig: ', 1, KSor+4);
            S:='';  Tol:=X;
            if Egszambe(S, 0, 0, Tol, MaxInt, X) then begin
              Ig:=X;
              Sorsol(N, Sor, Tol, Ig);  Str(N, S);
              VanSor:=True;  Rendezett:=False;
              Fejlec(FejSzov, 'Elemsz m: '+S, RendSzov[Rendezett]);
            end;
          end;
        end;
      end;
```

```
      2: if VanSor then begin
        Fejlec(FejSzov, 'Elemszám: '+S, RendSzov[Rendezett]);
        MunkaTerTorl;  SorKiir(N, Sor);
        Tovabb('');
      end;
      3: if VanSor and not Rendezett then begin
        KivalRend(Sor, N);  Rendezett:=True;
        Fejlec(FejSzov, 'Elemszám: '+S, RendSzov[Rendezett]);
      end;
      4: if VanSor then begin
        MunkaTerTorl;  Write('Keresett: ');
        S:='';
        if Egszambe(S, 0, 0, -MaxInt, MaxInt, X) then begin
          if Rendezett then begin
            if BinKer(Sor, N, X, Hol) then WriteLn('Megvan, indexe: ', Hol)
            else WriteLn('Nincs meg, helye: ', Hol);
          end else
            if Keres(Sor, N, X, Hol) then WriteLn('Megvan, indexe: ', Hol)
            else WriteLn('Nincs meg');
          WriteLn;  SorKiir(N, Sor);
          Tovabb('');
        end;
      end;
    end;
  until M=0;
end.
```

tKieg

```
unit tKieg;

interface

uses uTomb;

procedure Sorsol(Db: TElemDb; var Sor: TSor; Tol, Ig: TTElem);
procedure SorKiir(Db: TElemDb; const Sor: TSor);

implementation

procedure Sorsol(Db: TElemDb; var Sor: TSor; Tol, Ig: TTElem);
var
  I: TEIndex;
begin
  for I:=1 to Db do Sor[I]:=Tol+Random(Ig-Tol+1);
end;

procedure SorKiir(Db: TElemDb; const Sor: TSor);
var
  I: TEIndex;
begin
  for I:=1 to Db do Write(I:3, ':', Sor[I]:6);
end;
end.
```