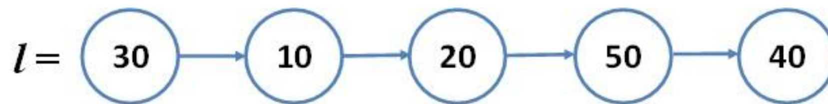
6. LISTÁK

Az előző fejezetekben megismerkedtünk a láncolt ábrázolással. Láttuk a verem és a sor, valamint – előre tekintve – a keresőfa pointeres megvalósításának a lehetőségét és előnyeit. A láncolt ábrázolással egy olyan dinamikus adatszerkezetet hozhatunk létre, amelyben az egyes rekordoktól a rákövetkezőikhez pointerek vezetnek. Lineáris adattípusok esetén erre a pointeres reprezentációra gyakran azt mondjuk, hogy ez a *láncolt ábrázolás* vagy a *listával történő megvalósítás*.

A „lista” kifejezés a szakmai szóhasználatban kettős jelentésű; vonatkozhat a láncolt ábrázolásra, de utalhat egy önálló adattípusra is, ha értelmezzük a műveleteit. Ebben a fejezetben főként az utóbbi értelemben beszélünk a listákról.

6.1. A listák absztrakciós szintjei

A lista egy önálló adattípus, amelyhez hozzá tartoznak saját műveletei. A listát, a többi típushoz hasonlóan lehet az *absztrakt adattípus* (ADT) szintjén is definiálni, ettől azonban most eltekintünk. Az *absztrakt lista adatszerkezet* (ADS) bevezetésének sincs akadálya, de nem okoz hiányt a tárgyalásban, ha ezt a szintet – az egyetlen erre utaló 6.1. ábrával – lényegében átlépjük most, és közvetlenül az ábrázolás szintjén kezdjük a listák tárgyalást.



6.1. ábra. A lista absztrakt adatszerkezet (ADS)

A *reprezentáció* szintjén két alapvető ábrázolási módot alkalmazunk ezúttal is: a *pointeres* és a *tömbös* megvalósítást. Ha megkülönböztetjük a – reprezentáció eszközeként használt – *lista adatszerkezet* és a – saját műveletekkel rendelkező – *lista adattípust*, akkor felvetődik a következő kérdés: beszélhetünk-e a lista típus láncolt ábrázolásáról, illetve lehet-e a listákat tömbösen ábrázolni? A válasz mindkét kérdésre „igen”, azzal a megjegyzéssel, hogy az alkalmazások döntő többségében láncoltan megvalósított listákkal találkozunk. (A listák láncolt ábrázolása annyira természetes, hogy ezt a reprezentációt nem is tünteti fel külön cím, hanem majd csak a listák tömbös ábrázolását, ami viszonylag ritka megoldás.)

Az itt következő tárgyalásban először megismerjük a listák lehetséges *fajtáit*, amelyeket esetünkben három tényező (fejelem, láncolás irányai, ciklikusság) alapján alakítunk ki. Az egyik tipikus listafajtára megadjuk a lista-műveleteket. Látni fogjuk, hogy itt már az egymáshoz is illeszkedő műveletek összehangolt rendszerére van szükség.

A listákat gyakran alkalmazzuk feladatok megoldásában. A listák tartalmával kapcsolatos tevékenységeket összeállíthatjuk a típusműveletekből, illetve megvalósíthatjuk *alacsonyabb szintű listakezeléssel*, amelyben kívülről „látjuk” a pointereket, és segítségükkel magunk kezeljük az listán történő lépkedést, az elemek tartalmának elérését.

Ha a *lista műveleteit* használjuk, akkor a pointereihez közvetlenül nem férhetünk hozzá, csak a műveleteken keresztül tudjuk elérni az elemeket. Az algoritmikus szempontból egyszerűbb feladatok megoldásban járhatunk el így, vagyis a lista-műveletek alkalmazásával. A feladatok megoldásban gyakrabban találkozunk a közvetlen listakezeléssel. Példaként egy lista helyben történő megfordítását látjuk majd. Utalhatunk a beszúró rendezés listás változatára is (lásd: 12. fejezet), amely már összetettebb feladatnak számít.

A listákat lehet tömbösen is ábrázolni, ahol a rákövetkező elemhez nem egy pointer, hanem egy index érték vezet. A megvalósítás tömbös eszközéhez folyamodunk például egy olyan programnyelv esetén, amely nem tartalmazza pointer nyelvi elemét.

6.2. A listák fajtái

A lista adattípust az ábrázolás szintjén ismertetjük. A fejezet nagyobb részében a *láncolt megvalósítást* részletezzük. Először a szerkezeti lehetőségeket vesszük sorra, utána bevezetjük a lista műveleteit.

A lista leggyakoribb megvalósításában a rekordok közötti kapcsolatot pointerek biztosítják. Ahhoz, hogy a lista első elemét is elérjük, egy arra mutató pointerre is szükségünk van. Ezt vagy egy közvetlen mutató biztosítja, vagy a listát kiegészítjük egy fizikai első elemmel, a *fejelemmel*, amelynek pointere mutat a logikailag első listaelemre. A listát azonosító önálló pointer ebben az esetben a fejelemre mutat. Egy lista ebből a szempontból lehet *fejelem nélküli*, vagy *fejelemes*.

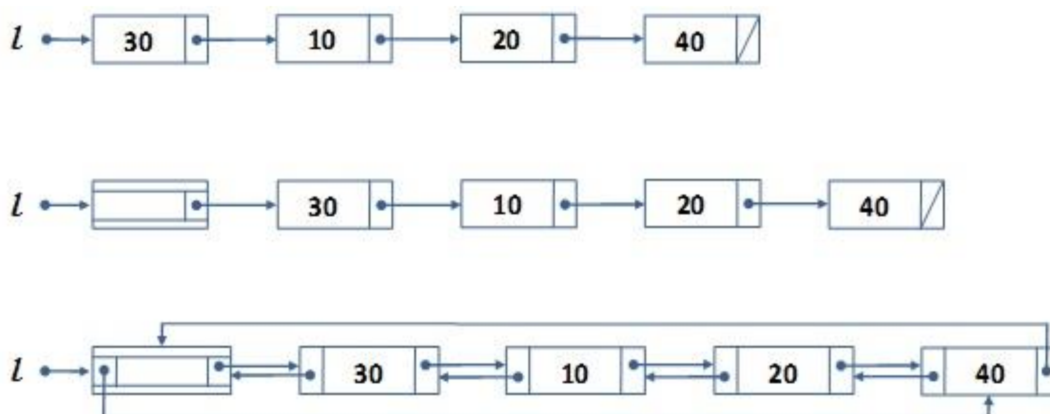
A pointerek láncán végighaladva sorban érhetjük el a lista elemeit, az elsőtől az utolsóig. Ha nem kell a memóriával erősen takarékoskodnunk, akkor kiegészíthetjük a listaelemeket visszafelé irányuló, a megelőző elemre mutató pointerekkel. A lista láncolása ennek megfelelően lehet *egyirányú*, vagy *kétirányú*.

A lista utolsó elemének előre mutató pointere szokás szerint NIL, mivel a rákövetkező elemek sora nem folytatódik tovább. A bejárást még rugalmasabbá teszi az, ha az utolsó elem pointere visszamutat az első elemre. Az ilyen listát nevezzük *ciklikusnak*, míg az eredetit *nem-ciklikusnak* (ha hangsúlyozni szeretnénk a ciklikusság hiányát).

A felsorolt három tényező mindegyike két-két lehetőséget kínál. Ezeket tetszőlegesen össze lehet párosítani, így nyolc lista fajtához jutunk. Ezek közül jobbára a következő hármat használják:

- az *egyszerű listának* nevezett valóban legegyszerűbb szerkezetet, amely nem tartalmaz fejelemet, egyirányú és nem ciklikus;
- a fejelemes egyszerű listát, amely annyiban különbözik az előzőtől, hogy tartalmaz fejelemet;
- a legtöbb lehetőséget támogató listát, amely tartalmaz fejelemet, kétirányú láncolás köti össze elemeit és ciklikus.

A felsorolt listafajtákat a 6.2. ábra szemlélteti.



6.2. ábra. A listák néhány fajtája

A fejelemes, kétirányú, ciklikus lista esetén meg kell fontolni annak a két mutatónak az irányítását, amelyek a ciklikusságot biztosítják. A fejelem visszafelé mutató pointere természetes módon az utolsó listaelemre mutat. Az utolsó elem előre irányuló pointerét irányítsuk a fejelemre. (Irányíthatnánk az első elemre is, de célszerűbb, ha az utolsó elemtől a fejelemhez vezet az út.) Két érv, ami emellett szól: a szimmetria, ami a többi közvetlen „oda-vissza” pointeres kapcsolatban lévő elempár között fennáll; valamint az, hogy így könnyebb ellenőrizni, hogy a lista végére értünk.

6.3. Listák műveletei

Kiválasztunk egy listafajtát, és megadjuk rá a listaműveleteket, pontosabban a műveletek egy lehetséges halmazát, hiszen több megoldás is lehetséges. Általában, a listaműveletek témakörében erős sokféleség uralkodik, ami nem zavaró. Az alábbi tárgyalásban néhol – zárójelben – rámutatunk az alternatív lehetőségekre. A műveleteknek egy ilyen halmaza már átgondolt tervezést kíván. A műveleteknek ugyanis illeszkedniük kell egymáshoz, valamint egységes arculatot is kell mutatniuk.

Legyen az egyszerű fejelemes lista az, amelynek megadjuk a műveleteit. A műveletek bevezetésének az alapja az, hogy értelmezzük a lista *aktuális elemét*, amelyet az *akt* pointer azonosítja. A műveletek legtöbbje az aktuális elemre vonatkozik. A műveletek általában módosítják is azt, hogy melyik elem lesz ezután az aktuális.

Az *akt* pointer lehetséges értékeit a következőképpen határozzuk meg:

- mutathat a lista elemeire;
- nem mutathat azonban a fejelemre, viszont
- „leléphet” az utolsó elemről és ekkor NIL az értéke.

(Az utóbbi két esetben ellenkezőleg is dönthettünk volna.)

A műveletek egy részének a végrehajtása hibához vezet, így szükségessé válik a hibakezelés. Ez megoldható például egy *hibaváltozó* logikai értékének a beállításával. Ekkor bevezethetünk egy olyan műveletet, amely a hibaállapotot kérdezi le, és hiba esetén törli ezt a státuszt.

(A lista adattípus részeként bevezethetők további változók is, pl. az *akt* pointer mögött járó mutató, vagy az utolsó elem pointere. Olyan megoldást is lehet látni, amelyben az *akt* pointer fizikailag az aktuális elem előtti elemre mutat.)

Nem foglalkozunk a nyelvi implementáció kérdésével. Szemléletünk az osztály és az objektum fogalmai felé mutat.

Az alábbi leírásokban az *l* pointer a lista fejelemére mutat, a listaelemek adatrészét *adat*, a pointer mezőt *mut* azonosítja. A teljes lista adatstruktúra részét képező *akt* és *hiba* változókra közvetlenül hivatkozunk, és ezek nem szerepelnek az eljárások paraméterei között; a listát csak *l* azonosítja. Ha egy művelet a lista egy elemének az adatrészével végez műveletet, akkor az eljárás input vagy output paramétere lesz a megfelelő rekordtípusú változó. (Az adatmozgatás nem *return* utasítással és nem pointeres hozzáférés biztosításával történik, hanem a paraméter-átadás segítségével.)

Összességében az alábbi tizenkét műveletet vezetjük be.

Üres (*l*). Üres lista létrehozása; egyúttal az üres lista-konstans neve. A létrehozott fejelem pointere NIL, vagyis maga a lista üres, nem tartalmaz rekordot.

Üres-e (*l*). Annak lekérdezése, hogy a lista üres-e. A logikai függvény az ennek megfelelő logikai értékkel tér vissza.

Hiba-e (*l*). Történt-e hiba az utolsó hiba-lekérdezés óta? A logikai függvény visszaadja a hiba-változó értékét, egyúttal törli a hiba-státuszt. Ebben a szemléletben a műveletek felhasználójának kell rákérdeznie a hibára.

Elsőre (*l*). A lista első eleme lesz az aktuális elem. Üres lista esetén hibajelzést vált ki ez a művelet.

Következőre (*l*). A lista következő eleme lesz az aktuális. Üres lista, vagy nem definiált aktuális esetén: hibajelzés. Ha az utolsó listaelem az aktuális, akkor a művelet hatására „lelép" erről az aktuális elem mutatója és értéke NIL lesz (nem számít hibának).

Utolsó-e (*l*). Annak lekérdezése, hogy a lista utolsó eleme-e az aktuális? Üres lista, vagy nem definiált aktuális elem esetén hibajelzést kapunk.

Vége-e (*l*). Annak lekérdezése, hogy az aktuális pointer „lelépett-e" a listáról? (EOF-jellegű állapot.) Üres listára is teljesül (definíció szerint), hogy a végén vagyunk.

AktÉrték (*l*, *x*). Hozzáférés az aktuális elem tartalmához. A művelet üres lista, vagy nem definiált aktuális elem esetén hibajelzést vált ki.

AktMod (*l*, *d*). Az aktuális elem adat-tartalmának felülírása. Üres lista, vagy nem definiált aktuális elem esetén a művelet hibajelzést vált ki.

BeszúrElsőnek (*l*, *d*). Rekord beszúrása első elemként a listába. Üres vagy nem-üres lista esetén egyaránt működik. A beszúrt első elem lesz az aktuális.

BeszúrUtán (*l*, *d*). Rekord beszúrása az aktuális elem után. Üres lista, vagy nem definiált aktuális elem esetén ez a művelet hibajelzéshez vezet. Sikeres végrehajtás esetén a beszúrt rekord lesz a lista aktuális eleme.

Töröl (*l*, *x*). A lista aktuális elemének törlése. Üres lista, vagy nem definiált aktuális elem esetén a művelet hibajelzést ad. Sikeres végrehajtás esetén a törölt elem utáni elem lesz az aktuális. A törölt rekord adattartalmát visszaadja az eljárás (e nélkül előbb az aktuális elem értékének lekérdezését kellene szükség esetén végrehajtani).

Az egyszerű fejelemes listára specifikált műveleteink egy kivételével konstans időben végrehajthatók, függetlenül a lista méretétől. A kivételt az aktuális elem törlése jelenti, ugyanis a fejelemtől indulva, a pointerek láncán el kell jutni az aktuális elemet megelőző elemhez, mivel annak pointerét módosítani kell: ugyanaz az érték kerül oda, mint ami a törlendő aktuális elem pointer-mezőjében található. Formálisan kifejezve:

$$T_{Töröl}(n) = O(n),$$

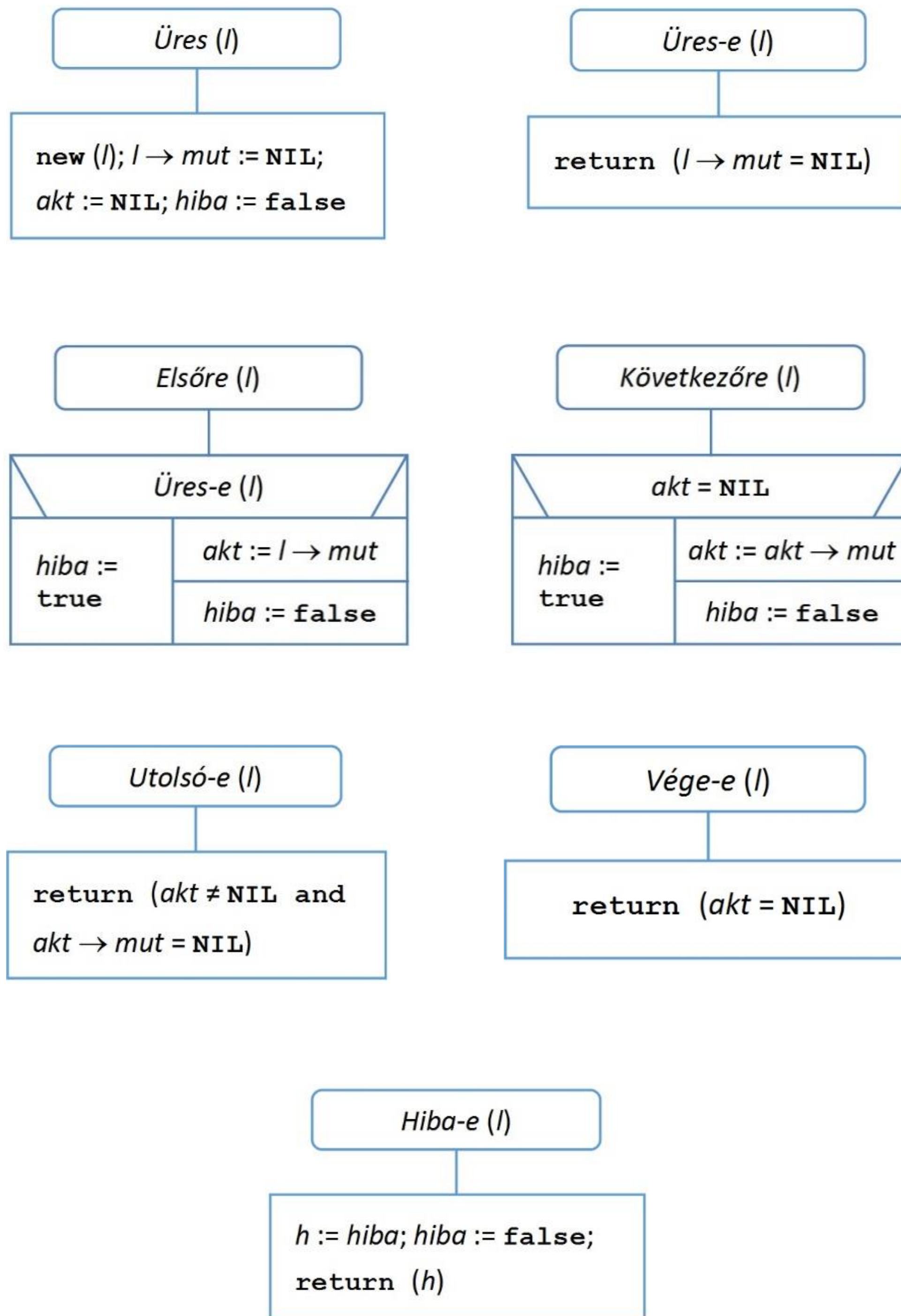
illetve

$$T_{op}(n) = \Theta(1),$$

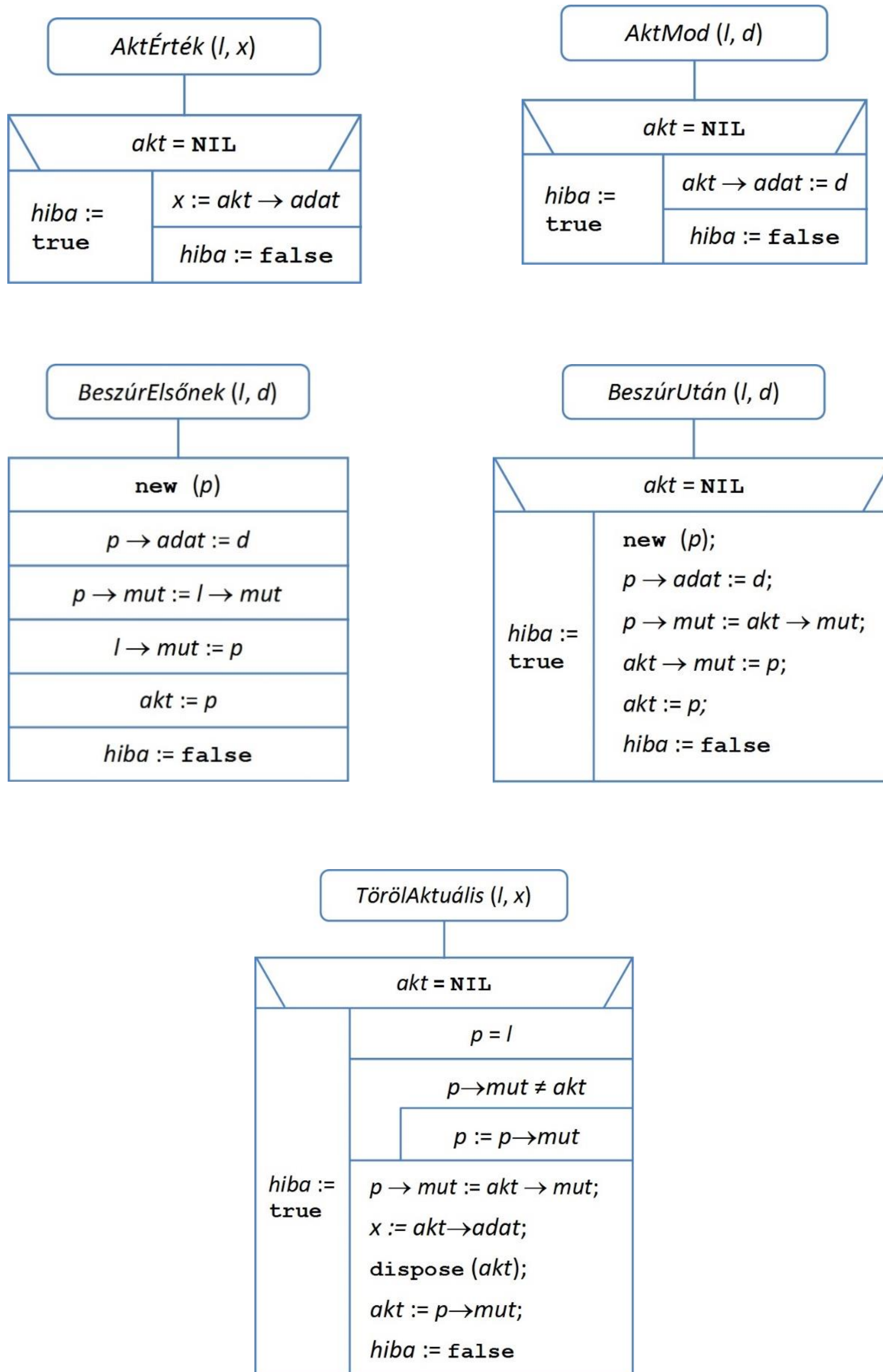
ahol *op* az egyszerű fejelemes lista bármely más (az aktuális elem törlésétől különböző) műveletét jelöli.

A törlés lineáris műveletigénye – és számos feladat – vezetett arra a gyakorlatra, hogy ma már jobbára a legösszetettebb lista fajtát alkalmazzák. Mivel a memóriával józan keretek között lényegében nem kell takarékoskodni, egy visszafelé mutató pointer bevezetése a rekordokba nem okoz észrevehető terhet. Viszont, az aktuális elemet megelőző elem közvetlenül elérhetővé válik.

A műveletek algoritmusai a 6.3.a és a 6.3.b ábrán láthatók.



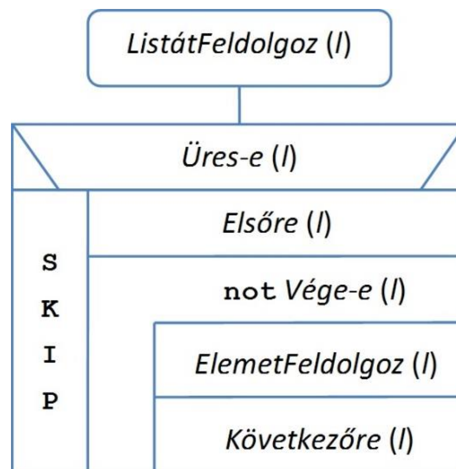
6.3.a ábra. Listaműveletek (1. rész)



6.3.b ábra. Listaműveletek (2. rész)

6.4. A művelek szintjén megoldott feladatok

A tapasztalat azt mutatja, hogy a lista bevezetett műveletei kevésbé rugalmasak, mint maga a pointeres lista-struktúra. A mutatók közvetlen átállításával olyan feladatokat is meg lehet oldani, amelyeket a műveletek szintjén csak körülményesen tudunk kezelni. A lista műveleteiből alapvetően olyan egyszerű eljárások állíthatók össze, mint például az elemek a feldolgozása a listában elfoglalt helyük szerinti sorrendben. A 6.4. ábrán látható eljárás ezt a tevékenységet valósítja meg.



6.4. ábra. Lista feldolgozása (műveletek szintjén)

Az algoritmus hasonlóságot mutat a szekvenciális input fájlok feldolgozásához, tartalmazza az ott gyakran alkalmazott *előre olvasás* technikáját: egy iterációs lépésben először az aktuális elem feldolgozására kerül sor, utána a következő elem válik aktuálissá. Az iteráció így minden lépésében az előző lépésben előkészített elemet dolgozza fel, majd kiválasztja a rákövetkező elemet feldolgozásra a következő iterációs lépés számára.

6.5. A reprezentáció szintjén megoldott feladatok

A listákkal kapcsolatos feladatok között könnyű olyanokat találni, amelyek megoldó algoritmusa jóval könnyebben adható meg saját pointer-kezelés mellett, mint a lista-műveletek alkalmazásával. Egy ilyen példa egy egyszerű lista megfordítása helyben.

Adott az *l* egyszerű (fejelem nélküli, egyirányú) lista. Az a feladat, hogy fordítsuk meg a lista elemeinek a sorrendjét helyben, azaz legfeljebb konstans méretű további memória felhasználásával. A megoldás elve a 6.5. ábrán látható.

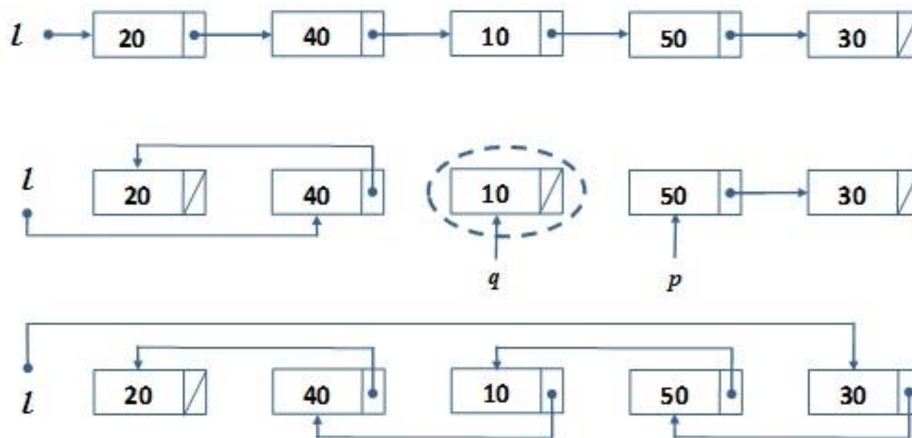
Az ábra első listája a kiinduló állapotot mutatja. Arra gondolunk, hogy alkalmazhatjuk az *elemenkénti feldolgozás* elvét. A listát mindenkor két részre lehet vágni:

1. a már megfordított sorrendű elejére és
2. a többi elem részlistájára.

Kezdetben az első rész üres és a második részt a teljes lista alkotja. Végül az első rész kiterjed a teljes listára, míg a második rész kiürül, vagyis végül minden elemet megfordítottunk.

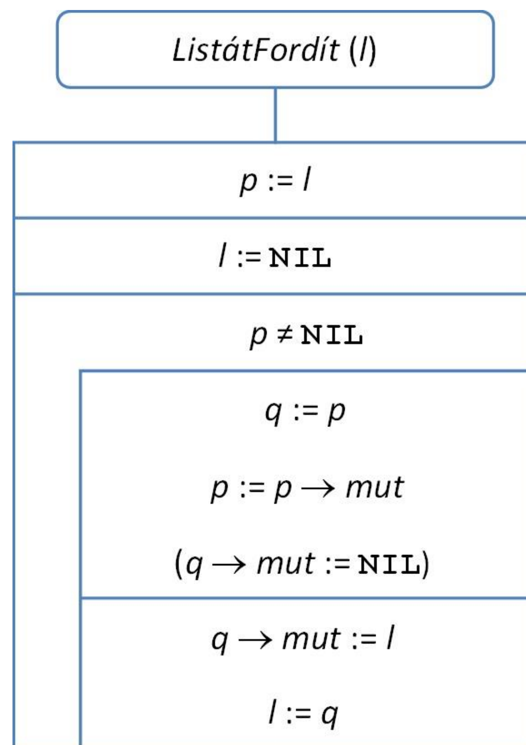
Minden lépésben leválasztunk egy elemet a lista második feléből és bevonjuk a sorrend megfordításába. A második ábra egy pillanatfelvétel a megoldás folyamatának egy közbülső állapotáról. A lista első két elemének a sorrendjét már megfordítottuk. A még fel nem dolgozott részből az ábrán már leválasztottuk az első elemet (a 10-eset), amelyet a *q* pointer

mutat, míg a további elemek kezdetére a p mutat. A leválasztott elem befűzése a már megfordított rész elejére két mutató átállításával megoldható.



6.5. ábra. Egyszerű lista megfordításának fázisai

Az algoritmust, amelyet az ábra és annak elemzése alapján már könnyen el lehet készíteni, a 6.6. ábra illusztrálja.



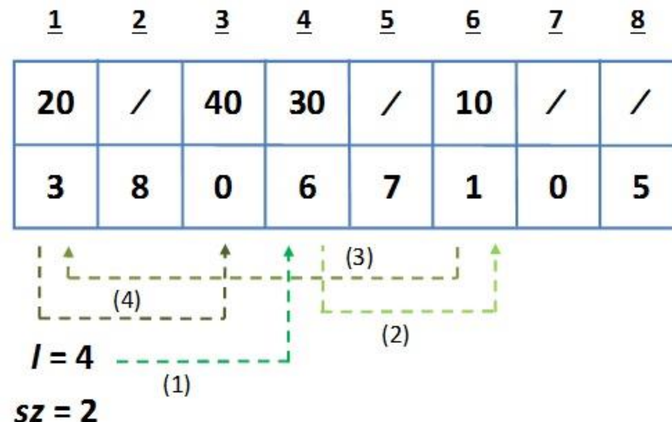
6.6. ábra. Egyszerű lista megfordítása (reprezentáció szintjén)

Érdemes felfigyelni arra, hogy a megfordított sorrendű első részt üres listával inicializáltuk, nem pedig egy-elemű listával. Ez több szempontból is így javasolható: az eljárás működik üres listára is, egyszerűbb és elegánsabb a kód és könnyebben bizonyítható annak helyessége.

6.6. Listák tömbös ábrázolása

A lista típus természetes megvalósítása pointerek alkalmazásával történik. A pointerek valójában memóriacímeket tartalmaznak, amelyeket a számítógépes végrehajtás elfed előlünk. A lista *tömbös ábrázolása* esetén magunk kezeljük az elemek közötti rákövetkezést biztosító indexeket.

A 6.7. ábra egy olyan tömbös lista-ábrázolást mutat, amelyben a tömb szabad helyei is láncolt kapcsolatban állnak.



6.7. ábra. Tömbben elhelyezett lista

A kétdimenziós tömbben elhelyezett lista ugyanaz, amely a 6.1. és 6.2. ábrákon is szerepel. Az $l = 4$ indexérték a lista első elemére mutat, az $sz = 2$ érték pedig a *szabad helyek* listájának kezdő indexe. Ebben az ábrázolásban a két lista terjedelme együttesen teszi ki a tömb helyfoglalását. Mindkét lista a másik „rovására” terjeszkedik. Ha a listába új elemet szúrunk be, akkor az a szabad helyek listájából láncoljuk ki és töltjük ki a megfelelő tartalommal. Ha törlünk egy elemet a listából, akkor azt a szabad helyek közé láncoljuk be.

A szabad helyek listájába célszerű első elemként beszúrni a törléskor felszabaduló elemet és megfordítva: beszúrás esetén legjobb az első szabad elemet kiláncolni, és azt a lista megfelelő helyére átfűzni. Ebben a működésben az adattartalmak a helyükön maradnak a tömbben. Az adatszerkezet dinamizmusát, az egyes tömbelemek „közlekedését” a két lista között az indexek értékeinek megfelelő beállításával valósítjuk meg.