I. ADATSZERKEZETEK JELLEMZÉSE ALGEBRAI AXIÓMÁKKAL

1 Elméleti alapok

Ebben a fejezetben elsősorban az adatszerkezetekkel, azok formális és informális leírásával, valamint jellemzésével fogunk megismerkedni.

Elsőként azt fontos tisztázni, mit is értünk adatszerkezeten. Az adatszerkezet – mint ahogy azt a neve is mutatja – egy olyan objektum, amely képes más adatok tárolására és feldolgozására. Ez utóbbi (vagyis az adatok feldolgozása, módosítása) az ún. műveletekkel, avagy operációkkal történik. Mégpedig úgy, hogy ugyanazon az adatsoron több műveltet is végezhetünk egymás után. Így formálisan akár azt is mondhatjuk, hogy az **absztrakt adatszerkezet adatok és a rajtuk értelmezett különböző műveletek összessége**.

Az adatszerkezetet azonban elsősorban **műveletei**, és nem pedig adatai **jellemzik**. Ezért az adatszerkezetek megjelenése mintegy előfutára volt a moduláris programozásnak: ha az absztrakt adatszerkezetnél nem a műveletek implementációja, adatai, hanem a működése a fontos, akkor ez lehetőséget ad arra, hogy egy-egy adatszerkezet belső adatait, implementációs részét elrejtsék a program többi része elől.

Az adatszerkezet pontos, egzakt leírása ezért nem egyszerű feladat: egzakt le kell leírni a műveletek eredményét, viselkedését, egymásra gyakorolt hatását. A moduláris programozás jegyében arról azonban nem kell – és nem is szabad – beszélni, hogy egy adott adatszerkezetet hogyan kell megvalósítani, implementálni.

1.1 Adatszerkezetek informális definíciója

Az adatszerkezeteket kezdetben elsősorban informálisan, szöveggel definiálták.

Például: a stack (verem) egy olyan adatszerkezet, amelyen három művelet értelmezett, a PUSH, a POP és a TOP. A PUSH a verem tetejére rak egy elemet, a POP leveszi a verem tetején álló elemet, míg a TOP megmutatja a verem tetején álló elemet.

A sima szövegnél egy pontosabb informális leírást kapunk, ha minden műveletet működését specifikáljuk. Ekkor azonban bizonyos esetekben viszont elkerülhetetlen az implementálásra vonatkozó feltétel megadása.

Az ilyen informális leírások nyilvánvaló előnye, hogy egyszerűek. De ugyanakkor az idők során nem bizonyultak elég precíznek, mert midig előfordult egy-egy olyan eset, amikor valamilyen műveletet nem adtak meg elég pontosan, és ebből később felesleges bonyodalmak adódtak. Ezért az akkori számítástechnikusok a nagyon is formális és precíz matematikához, azon belül is az algebrához fordultak.

1.2 Adatszerkezetek algebrája

Az adatszerkezetekkel kapcsolatos kutatások később igazolták, hogy az adatszerkezetek jól jellemezhetők az algebra módszereivel. Nevezetesen algebrai úton tudunk megalkotni egy olyan szabályrendszert, amely szinte tökéletesen jellemzi az adott adatszerkezetet.

Ebben az algebrában az adatszerkezet, mint egy algebrai struktúra jelenik meg, és a struktúra műveletei éppen az adatszerkezet műveletei lesznek.

Formálisan a következőképpen adunk meg egy absztrakt adatszerkezetet:

ADATSZERKEZET (elem: [undefined->elem])

A fenti definíciót követheti még néhány kulcsszó. Például az **import** kulcsszó szó után olyan típusokat sorolhatunk fel, amelyeket az adatszerkezet egy-egy művelete használ. Míg az **inherits** kulcsszó után azokat az adatszerkezeteket sorolhatjuk fel, amelytől a jelenlegi örökölt valamilyen tulajdonságot.

A fenti adatszerkezetet azért nevezzük absztraktnak, mert nem foglalkozunk külön azzal, hogy mit is értünk *elem* alatt; illetve azzal, hogy éppen milyen elemek tartózkodnak benne. Valamint nem leszünk tekintettel semmilyen fizikai és/vagy implementációs korlátozásra (tárkorlát, méret korlátja, idő, stb.) sem. Így egy olyan általános leírást kapunk, amely az igazi, a kertben ásott veremtől kezdve a dupla pontosságú BCD számokat tároló veremig minden vermet közösen jellemez.

1.2.1 Műveletek az algebrában

A **műveleteket** az algebrában **formálisan függvénynek tekintjük**, és ezért a következő módon adjuk meg őket:

értékkészlet: **művelet(***értelmezési tartomány***)** illetve

művelet: *értelmezési tartomány* $\rightarrow$ *értékkészlet*

A fenti leírás nyilvánvalóan nem írja le a műveletek viselkedését; ezt majd más módon kell megtenni: vagy szövegesen, vagy pedig a majd később ismertetett axiómarendszerrel.

A fenti leírással kapcsolatosan meg kell még mondanunk, mit is értünk értelmezési tartomány és értékkészlet alatt. Ezek matematikailag természetesen halmazok, mégpedig a típusok halmazainak Descartes-szorzata. Magyarul: egy művelet értelmezési tartományában ill. értékkészletében többféle típus is előfordulhat. A továbbiakban az egyszerűség kedvéért azonban csak olyan műveletekkel foglalkozunk, amelyek értékkészlete egy adott típus. Például a stack PUSH műveletének értelmezési tartománya egy stack és egy elem, értékkészlete pedig egy stack.

Fontos viszont, hogy **minden olyan műveletnek**, **amely változtat is az adatokon, kell lennie legalább egy adatszerkezet-típusú visszatérési értékének is.** Ez az érték adja majd vissza a megváltoztatott adatszerkezetet.

1.2.2 Műveletek csoportosítása

Mint ahogyan általában mi is, az adatszerkezet-algebra is különbséget tesz az adatszerkezet műveletei között, nevezetesen három csoportba sorolja azokat:

- Egy művelet **konstruktor**, ha **bővíti adatszerkezet**, azaz **adatainak számosságát növeli**. Mindezek mellett a konstruktorokkal szemben fontos követelmény, hogy egy adatszerkezet összes konstruktorával ,és csak azokkal előállítható legyen a szerkezet valamennyi formája.

  Az undefined értelmezési tartományú művelet az adatszerkezet speciális konstruktora. Ez a művelet hozza létre az új, általában üresnek tekintett szerkezetet. A neve ezért általában CRT(), illetve NEW().

- Az adatszerkezetet megváltoztató minden egyéb művelet neve **modifier**. A leírás alapján nyilván van adatszerkezet típusú paramétere és visszatérési értéke is. A modifier tehát **általában nem bővíti az adatszerkezetet**, hanem csak módosítja azt; illetve, ha mégis bővíti, akkor **a bővített adatszerkezet előállítható más módon is a konstruktorok segítségével.** Azaz: minden olyan nem-konstruktor művelet modifier, amelynek értékkészletében adatszerkezet típus van.

  A konstruktor-modifier felosztás nyilvánvalóan nem egyértelmű: egyes műveletek egy felosztásban lehetnek konstruktorok, egy másikban modifierek is.

- Az olyan műveletek, amelyek nem változtatnak az adatokon úgynevezett **behaviour** műveletek. A behaviour-ök semmilyen módon nem változtathatják az adatszerkezetet (nincs is ilyen visszatérési értékük), hanem **arról valamilyen információt szolgáltatnak**.

Egy műveletről a következőképpen dönthetjük el hovatartozását:

- ha a visszatérési értékei között nincs adatszerkezet-típus, akkor **behaviour**.
- ha van adatszerkezet-típusú visszatérési értéke és bővíti is a halmazt, akkor **konstruktor**. Később persze meg kell vizsgálni, hogy valamennyi konstruktorra szükség van-e.
- egyébként **modifier**.

Példaként tekintsük most a fenti stack példánk néhány műveletét, név és leírás nélkül:

stack: α**(***undefined***)**
boolean: β**(***satck***)**
numeric: γ**(***stack, numeric***)**
stack: δ**(***stack***)**

stack: **ε(***stack, numeric***)**

A β művelettel nem lehet konstruktor (*boolean*-t ad vissza), ezért behaviour. A γ művelet szintén biztosan behaviour, mert numeric-et ad vissza. Az α művelet értelemszerűen konstruktor. A δ illetve az ε műveletek vagy konstruktorok vagy modifierek. Pontosan csak akkor tudjuk eldönteni, ha ismerjük a működésüket.

Természetesen mindenkiben felmerülhet kérdés, hogy pontosan mi is az értelme a csoportosításnak. A válasz: a különböző műveletek segítségével megalkothatjuk az adatszerkezet egy egyszerű szabályrendszerét: az axiómarendszert.

1.3 Algebrai axiómák

Mint azt már az előzőekben is említettük, egy adatszerkezet működésének, viselkedésének pontos leírásához szükség van még egy formális segédeszközre, az adatszerkezet axiómáira.

Az axióma fogalom nem ismeretlen számunkra: egy nem bizonyítható, alapvető igazságot, szabályt fogalmaz meg. Valami hasonlatos dolgot fogunk mi is csinálni: előállítjuk az adatszerkezet egy axiómarendszerét, hogy az informális leírás helyett formálisan le tudjuk írni a viselkedését.

Már csak az a kérdés, hogy hogyan is írjuk fel ezeket az axiómákat. Egy bizonyos matematikai tanulmányainkból: csak az axióma bal oldalára tudunk formális sablont ráhúzni, a jobb oldalról, csak annyit tudunk, hogy ott valamilyen más kifejezés kell hogy álljon.

Most – minden további elméleti bizonyítást mellőzve – megadjuk az axiómák alakját:

modifier(konstruktor)= ...

behaviour(konstruktor)= ...

Azaz: **Egy adatszerkezet axiómáit úgy állítjuk elő, hogy a nem-konstruktor műveletek egy-egy adatszerkezet típusú paraméterét egy-egy konstruktorral helyettesítjük, és a kifejezést kiértékeljük.**

Magyarul: az axiómákban azt írjuk le, hogy mi is történik két művelet egymás után alkalmazásával.

Az összes axiómát természetesen akkor kapjuk meg, ha minden modifier és minden behaviour minden adatszerkezet típusú paramétere helyére beírtunk minden konstruktort.

A fenti módszerrel előálló axiómák számát a következő egyszerűen belátható kombinatorikai kifejezés adja meg:

Ha minden művelet legfeljebb egy adatszerkezeten dolgozik, akkor az axiómák száma = nem-konstruktorok száma * konstruktorok száma. Illetve egyéb esetben a következő kifejezés érvényes:

$$\#\text{axióma} = \sum_{\substack{\text{minden}\\ \text{nem}\\ \text{kostruktorra}}} \text{kostruktorok száma}^{\text{paraméterek száma}}$$

Most már csak egy dologgal maradtunk adósak: meg kell mondanunk, hogy az axióma jobb oldalán milyen kifejezéseket fogadunk el.

1.3.1 Axiómák jobb oldalának leírása

Az axióma jobb oldalán általában egy összetett kifejezés szerepel, erre nincs megkötés. Az axióma szempontjából általában az a legkedvezőbb, ha az újabb összetett kifejezés vagy egyszerűbb az előzőnél, vagy pedig valamilyen invarianciát mutat ki. (Például X esetben az adatszerkezet invariáns a műveletek sorrendjére.)

A jobb oldalon ezenkívül a szám ill. logikai típusú értékeken használhatók a matematikai műveletek (összeadás, logikai és, stb.)

Végül, de nem utolsó sorban használhatók az

if (feltétel**) then** kifejezés1

else kifejezés2

típusú elágazások. Egyetlen feltétel, hogy a feltétel egy logikai igen-nem típusú kifejezés legyen; a kifejezés természetesen tartalmazhat további elágazásokat is. Az összehasonlítás műveletek (==,<,>) használhatók a feltétel

előállításakor. **Kivéve: tilos az adatszerkezet==adatszerkezet típusú összehasonlítás minden formája**, **ha** ez a művelet konkrétan, explicit **nem definiált** az adatszerkezeten. (Általában ez a helyzet.)

A matematikai műveletek jelölése tetszőleges, ám nem ismert jelölésrendszer esetén illik utalni a művelet jelentésére. (Például: a logikai negálást ~-vel jelöltem, stb.)

2 Az axiómák felírásának alapvető lépései

1. **Konstruktorok megkeresése:** konstruktor általában minden olyan művelet, amely *bővíti* a jellemzett adatszerkezetet, de – persze – ki is elégíti a konstruktor definícióját.
2. **Egyéb műveletek összetétele a konstruktorokkal**: az egyéb művelet megfelelő paramétere(i) helyére beírunk egy-egy konstruktort, majd a művelet leírásából megpróbáljuk „kitalálni" a végeredményt, és ezt a matematika formális eszközeivel leírni. Az összes axiómát akkor kapjuk, ha *minden* egyes *nem-konstruktor művelet minden paramétere helyére* beírtunk *minden* konstruktort. Ne feledkezzünk meg az esetleges dupla ill. tripla konstruktoros variációkról sem!
3. **Fontos szempontok, elvárások az axiómákkal szemben:**
   - az axiómának rövidnek, tömörnek kell lennie
   - tartalmát tekintve „triviálisnak"
   - az axióma NEM kiértékelés – a túl bonyolult kifejezések mindig gyanúsak

3 Mintafeladatok

3.1 Várakozási sor

Algebrai axiómák segítségével specifikálja az alábbi műveletekkel jellemzett várakozási sort (queue-t)
*queue: **NEW()** - új, üres várakozási sor*
*queue: **DROP(queue,item)** - queue-ba item-et elhelyezi, mint legutolsó (last) elem*
*queue: **REMOVE(queue)** - queue-ból kiveszi a tetjén álló (first) elemet,*
*boolean: **ISIN(queue,item)** - igaz, ha a queue-ban item már bent van*
*int: **ELEMENTS(queue)** – a queue elemeinek száma*
*int: **NUM(queue,item)** - hányszor fordul elő item a queue-ban*
*item: **MINIMAL(queue)** – queue-ban a legkisebb elem*
*queue: **MINDEL(queue)** – a legkisebb elem törlése a queue-ből*
– – – – – – – – – – – – – – – – – – –

Először tisztázzuk a műveletek szerepét!

Konstruktorok:	NEW(), DROP(). Ezekkel ugyanis bármilyen más várakozási sor előállítható.
Modifierek:	REMOVE(), MINDEL()
Behaviour:	ISIN(), NUM(), ELEMENTS(), MINIMAL(), mert ezek visszatérési értéke nem queue típusú.

6*2=12 axiómát várunk, mert minden művelet egy queue-t vesz paraméternek.

Axiómák:

- **REMOVE(NEW())== NEW()** *(üres sornál nincs még üresebb)*
- **ISIN(NEW(),i) == false** *(az új sor üres)*
- **ELEMENTS(NEW())==0** *(az új sorban nincsenek elemek)*
- **NUM(NEW(),i)==0** *(új sorban egyetlen elem sem fordul elő)*
- **MINIMAL(NEW())==undefined** *(üres sorban nincs minimális elem)*
- **MINDEL(NEW())==NEW()** *(üres sorból nincs mit törölni)*
- **REMOVE(DROP(s,i))== if ELEMENTS(s)> 0 then DROP(REMOVE(s),i) else NEW()**
  (mindegy, hogy először teszünk-e be elemet, és aztán veszünk ki, ha a sor nem üres. Sor ürességét nem lehet s==NEW() feltétellel tesztelni, hiszen nincs egyenlőség értelmezve)
- **ISIN(DROP(s,i_1),i_2) == (i_2==i_1) | ISIN(s,i_2)**
  (a betett elem biztosan a sorban van)

A | művelet logikai vagy-ot jelöl. A logikai kifejezést másképpen leírva: if (i2==i1) then true else ISIN(s,i2).

- **ELEMENTS(DROP(s,i))==ELEMENTS(s)+1**

  (betétellel a sor hossza eggyel, és csak eggyel bővül)

- **NUM(DROP(s,i_1), i_2)== if (i_2==i_1) then NUM(s,i_1)+1 else NUM(s,i_2)**

  (betételnél a betett elem eggyel többször fordul elő, míg a többit elem számát nem érinti)

- **MINIMAL(DROP(s,i))== if ELEMENTS(s)>0 then if i<MINIMAL(s) then i**
  else MINIMAL(s)
  else i

  (a minimális elem vagy már bent volt, vagy most tettük bele)
  A második elágazást leírhatjuk a min{} matematikai operátor alkalmazásával is.

- **MINDEL(DROP(s,i))== if ELEMENTS(s)>0 then**
  if i<MINIMAL(s) then s else DROP(MINDEL(s),i)
  else NEW()

  (ha most tesszük bele a minimális elemet, akkor gyakorlatilag nem csináltunk semmit, ha már benne volt, akkor a sorrend nem számít)

3.2 Konténer (1996. január 4.)

Algebrai axiómák segítségével specifikálja az alábbi műveletekkel jellemzett konténert!

NEW()	*új (üres) konténert hoz létre;*
ADD(k,x)	*az **x** elemet a **k** konténerbe rakja;*
SUB(k,x)	*egy **x** elemet a **k** konténerből eltávolít;*
DEL(k,x)	*valamennyi **x** elemet eltávolítja a **k** konténerből;*
MBR(k,x)	*igaz, ha van a **k** konténerben legalább egy **x** elem.*

– – – – – – – – – – – – – – – – – – –

Célszerű először a műveletek értelmezési tartományát és értékkészletét tisztázni, mivel ezt a feladat nem adta meg. Az értelmezési tartomány – értékkészlet vizsgálatával gyorsan megtalálhatók a konstruktorok.

MŰVELET	ÉRTELMEZÉSI T.	⇒	ÉRTÉKKÉSZLET	BESOROLÁS
NEW()	*undefined*	⇒	*konténer*	konstruktor
ADD()	*elem × konténer*	⇒	*konténer*	konstruktor *(és minden konténert előállít)*
SUB()	*elem × konténer*	⇒	*konténer*	modifier
DEL()	*elem × konténer*	⇒	*konténer*	modifier
MBR()	*elem × konténer*	⇒	*boolean*	behaviour

3*2=6 axiómát várunk.

Axiómák:

- **MBR(NEW(),x) == FALSE** *(minden új konténer üres)*
- **SUB(NEW(),x) == NEW()** *(üresnél nincs üresebb konténer)*
- **DEL(NEW(),x) == NEW()** *(üresből nincs mit kitörölni)*
- **MBR(ADD(k,x),y) == if (x==y) then TRUE else MBR(k,y)**

  (ha egy elemet beleteszünk a konténerbe akkor az benne lesz)

- **SUB(ADD(k,x),y) == if (x==y) then k else ADD(SUB(k,y),x)**

  (ha egy elemet beleteszünk, majd utána kivesszük, akkor ugyanazt a konténert kapjuk vissza; ha a két elem különböző, akkor az adatszerkezet érzéketlen a sorrendre)

- **DEL(ADD(k,x),y) == if (x==y) then DEL(k,x) else ADD(DEL(k,y),x)**

  (Ha egy elem összes előfordulását töröljük akkor mindegy, hogy előtte még egyszer beletesszük-e. Ha a két elem nem egyforma, akkor mindegy, hogy előbb törlünk, aztán teszünk-e be. Ezt a legegyszerűbben úgy láthatjuk be, hogy a DEL() művelet tulajdonképpen SUB() műveletek egymásutánja, és így minden SUB(ADD())-ra alkalmazható az előző axióma, nevezetesen: DEL(ADD(k,x),y)≡ SUB(SUB(…SUB(ADD(k,x),y),y)…,y) ≡ SUB(SUB(…ADD(SUB(k,y)x),y)… ,y)) ≡ ... és így legvégül éppen az ADD() művelet áll majd elöl, abban pedig az összes SUB(), ami viszont éppen egy DEL().)

3.3 String (1997. január 7.)

Algebrai axiómák segítségével specifikálja az alábbi műveletekkel jellemzett stringet!

CRT()	*új (üres) stringet hoz létre.*
SUBS(s,i,x)	*az **s** string **i**-ik pozícióján álló karaktert **x** karakterre változtatja. Ha a string rövidebb, mint **i**, akkor a művelet hatástalan.*
APPEND(s,x)	*az **s** string végére rakja az **x** karaktert.*
ISIN(s1,s2)	*igaz, ha az **s2** string az **s1** string végén áll.*
LGTH(s)	*az **s** string karaktereinek számát adja.*

– – – – – – – – – – – – – – – – – – –

Kezdjük ismét az értelmezési tartomány – értékkészlet vizsgálatával, a konstruktorok triviálisak.

MŰVELET	ÉRTELMEZÉSI T.	⇒	ÉRTÉKKÉSZLET	BESOROLÁS
CRT()	*undefined*	⇒	*string*	konstruktor
SUBS()	*string × int × elem*	⇒	*string*	modifier *(nem bővíti a stringet)*
APPEND()	*string × elem*	⇒	*string*	konstruktor
ISIN()	*string × string*	⇒	*boolean*	behaviour
LGTH()	*string*	⇒	*integer*	behaviour

Ha 3*2=6 axiómát várunk, akkor nem olvastuk el az elméletet. Az axiómák száma ugyanis $2*2+2^2=8$, mivel az ISIN() két konstruktort vesz paraméternek!

Feltesszük, hogy a SUBS() műveletnél i>0.

Axiómák:

- **LGTH(CRT()) == 0** *(minden új string üres)*
- **LGTH(APPEND(s,x)) == LGTH(s)+1** *(a bővített string egy karakterrel hosszabb)*
- **SUBS(CRT(),i,x)== CRT()** *(üresben nem lehet mit cserélni)*
- **SUBS(APPEND(s,x1),i,x2)== if (LGHT(s)+1<i) then APPEND(s,x1)**
  else if (i=LGHT(s)+1) then APPEND(s,x2)
  else APPEND(SUBS(s,i,x2),x1)

  (Ha i nagyobb, mint a bővített string hossza, akkor a SUBS() hatástalan; ha i pont a bővített string végére mutat, akkor tulajdonképpen x2-vel bővítettük a stringet; ha pedig beljebb, akkor a string érzéketlen a csere-végére illesztés műveletek cseréjére.)
- ISIN(CRT(),CRT())==true

  ISIN(APPEND(s,x), CRT()) ==true

  A fenti két axiómából azonban egyet csinálhatunk, mondván a két konstruktor már minden stringet előállít:

  ISIN(s,CRT())== true *(azaz üres string minden string végén áll)*
- **ISIN(CRT(),APPEND(s,x)) == false** *(rövidebben hosszabb nincs)*
- **ISIN(APPEND(s1,x1), APPEND(s2,x2)) == if (x1==x2) then ISIN(s1, s2)**
  else false

  (Ha két tetszőleges string végére tetszőleges karaktert biggyesztünk, csak akkor állhat az egyik string végén a másik, ha a hozzájuk adott betű azonos (szükséges feltétel) és az egyik eredeti string megtalálható a másik végén. Rekurzív definíció.)

3.4 Kétvégű lista (1995. december 21. – módosított változat)

Algebrai axiómák segítségével specifikálja az alábbi műveletekkel jellemzett mindkét végén (A,B) elérhetô listát.

CRT()	*új (üres) listát hoz létre*
PUTA (l,n)	*az **n** elemet az **l** lista A végére rakja*
PUTB (l,n)	*az **n** elemet az **l** lista B végére rakja*
GETA (l)	*az **l** lista A végérôl leveszi a végén álló elemet*
GETB (l)	*az **l** lista B végérôl leveszi a végén álló elemet*
ATA (l)	*visszaadja az **l** lista A végén álló elemet*
ATB (l)	*visszaadja az **l** lista B végén álló elemet*

LGTH (l) *megadja az **l**-beli elemek számát*

Bár a feladat az előbbi feladatok után triviálisnak tűnik – mint látni fogjuk – nem egészen az. Amikor felírjuk az értelmezési tartományt és az értékkészletet, nyomban kiderül a turpisság.

MŰVELET	ÉRTELMEZÉSI T.	⇒	ÉRTÉKKÉSZLET	BESOROLÁS
CRT()	*undefined*	⇒	*lista*	konstruktor
PUTA()	*lista* × *elem*	⇒	*lista*	konstruktor
PUTB()	*lista* × *elem*	⇒	*lista*	konstruktor
GETA()	*lista*	⇒	*lista*	modifier
GETB()	*lista*	⇒	*lista*	modifier
ATA()	*lista*	⇒	*elem*	behaviour
ATB()	*lista*	⇒	*elem*	behaviour
LGTH()	*lista*	⇒	*integer*	behaviour

A „turpisság”: a PUTA() és a PUTB() műveletek közül legfeljebb az egyik fér bele a konstruktor általunk tett definíciójába. Hiszen üres listából kiindulva folyamatosan csak az A végéről írva bármilyen lista előállítható. Maradjon tehát PUTA() konstruktor, PUTB() innentől kezdve modifier. (De ez csak konvenció!) Így összesen 6*2=12 axiómát várunk.

Ezek után először írjuk fel az egyszerűbb axiómákat:

- **LGTH(CRT()) == 0** *(üres listában nincs elem)*
- **LGTH(PUTA(l,n)) == LGTH (l) + 1** *(a lista betétellel nő)*
- **GETA(CRT()) == CRT()** *(üresből egyetlen végén sem lehet kivenni elemet)*
- **GETB(CRT()) == CRT()**
- **ATA(CRT()) == undefined** *(az üresnek egyik végén sem áll semmi)*
- **ATB(CRT()) == undefined**
- **PUTB(CRT(), n) == PUTA(CRT(), n)** *(üres listának nincs két vége)*
- **GETA(PUTA(l,n)) == l** *(betétel és kivétel ugyanarról a végről nem csinál semmit)*
- **ATA(PUTA(l,n)) == n** *(A betétel után a végén tényleg az az elem áll)*
- **PUTB(PUTA(l, n1), n2) == PUTA(PUTB(l, n2), n1)**

  (a lista két vége ekvivalens és ezekre az adatszerkezet invariáns)
- **GETB(PUTA(l,n)) == if (LGTH(l)==0) then CRT(), else PUTA(GETB(l),n)**

  (a lista két vége nem üres lista estén érzéketlen a műveleti sorrendre)

  ATB(PUTA(l,n)) == if (LGTH(l)==0) then n, else ATB(l)

  (a lista egyik végén álló adatot nem változtatja a másik végen végzett beszúrás)

Azok kedvéért, akik kétkedve fogadják azt, hogy PUTB-t modifier-ré alacsonyítottuk le, megmutatjuk, hogy minden PUTB-re felírható „axióma” már eldönthető lenne. Például tekintsük a GETB(PUTB(l,x)) kifejezést. Erről azonnal látszik, hogy l.

Írjuk fel ugyanis l-et a bal végéről folyamatosan írva a megfelelő elemeket, ez triviálisan lehetséges: GETB(PUTB(l,x)) == GETB(PUTB(PUTA(PUTA(….PUTA(CRT(),c_0)…, c_{n-2}), c_{n-1}), c_n), x)), Erre a kifejezésre alkalmazzuk a PUTB(PUTA()) axiómát: GETB(PUTB(l,x)) == GETB(PUTA(PUTB(PUTA(….PUTA(CRT(),c_0)…, c_{n-2}), c_{n-1}),x), c_n)) . Ezt ismételjük meg még kellően sokszor, a végén nyilván azt kapjuk, hogy GETB(PUTB(l,x)) == GETB(PUTA(PUTA(PUTA(….PUTA(PUTB(CRT(),x),c_0)…, c_{n-2}), c_{n-1}), c_n)), azaz PUTB „bekerült” a CRT() elé. Erre alkalmazzuk a megfelelő PUTB(CRT())==PUTA(CRT()) axiómát: GETB(PUTB(l,x)) == GETB(PUTA(PUTA(PUTA(….PUTA(PUTA(CRT(),x),c_0)…, c_{n-2}), c_{n-1}), c_n)). Végül alkalmazzuk egymás után rendre a GETB(PUTA())==PUTA(GETB()) axiómát. A kapott kifejezés nem más, mint hogyPUTA(PUTA(PUTA(….PUTA(GETB(PUTA(CRT(),x)),c_0)…, c_{n-2}), c_{n-1}), c_n). Az aláhúzott rész az egyik axióma miatt éppen CRT() /felhasználjuk, hogy LGTH(CRT())==0 ⇔ l==CRT() /. Tehát a végén a GETB(PUTB(l,x)) == PUTA(PUTB(PUTA(…. PUTA(CRT(),c_0)…, c_{n-2}), c_{n-1}), c_n) kifejezés marad, ez viszont éppen l, azaz a fenti kifejezést levezettük az axiómákból.

3.5 Palindrómák (1998. április 21. , 1999. december 21.)

Algebrai axiómák segítségével specifikálja az alábbi műveletekkel jellemzett stringet!

CRT()	*új, üres stringet hoz létre.*
LGTH(s)	*az **s** string karaktereinek számát adja.*
TAIL(s)	*az **s** string elsõ karakterének levágása után maradó stringet adja.*
APPEND(s,x)	*az **s** string végére rakja az **x** karaktert.*
PALIN(s)	*igaz, ha az **s** palindróma.*
HEAD(s)	*az **s** string elsõ karakterét mutatja meg.*

Egy string palindróma, ha az elejétõl olvasva ugyanaz, mint visszafelé. Pl.: "görög".

A konstruktorok, modifierek és behaviour-ök triviálisak, így csak felsoroljuk őket.

Konstruktorok:	CRT(), APPEND()
Modifierek:	TAIL()
Behaviour:	LGTH(), PALIN(), HEAD()

4*2=8 axiómát várunk, ebből hét triviális.

- **TAIL(CRT()) == CRT()** *(Az üres stringnél nincs üresebb.)*
- **PALIN(CRT()) == true** *(Az üres string palindróma. Esetlegesen – utólag indokolva – elfogadható a false is, de ekkor egyes axiómákat másképp kell felírni.)*
- **HEAD(CRT()) == undefined** *(Üres stringben nincs elem.)*
- **LGTH(CRT()) == 0** *(Üres stringben nincs egy karakter sem.)*
- **LGTH(APPEND(S,X))=LGTH(S)+1** *(A stringhez hozzádott karakter növeli annak hosszát.)*
- **TAIL(APPEND(S,X))=if (LGTH(s)==0) then CRT() else APPEND(TAIL(S),X)**
  *(A nem üres string vége önálló stringként viselkedik. **Emlékeztetőül**: nem írunk olyat, hogy* if(S==CRT()) !*)*
- **HEAD(APPEND(S,X))=if (LGTH(s)==0) then X else HEAD(S)**
  (A hozzáfűzés a string elejét nem befolyásolja.)
- **PALIN(APPEND(S,X))=if (LGTH(s)==0) then TRUE else ((HEAD(S)=X) & PALIN(TAIL(S)))**
  (Egy string csak úgy egészülhet ki palindrómává, ha éppen olyan karaktert teszünk a végére, mint az első karaktere, valamint a középső része már palindróma. Az egy karakteres string mindig palindróma.)

4 Gyakorlófeladatok

4.1 Számpárok (1994. január 19.)

Algebrai axiómák segítségével specifikálja az alábbi műveletekkel jellemzett listát, amelyek elemei olyan (x,y) párok, ahol x egy kulcs, y egy 0-nál nagyobb egész !

NEW( )	*új (üres) listát hoz létre.*
ADD(L,(x,y))	*az L listához kapcsolja az x,y párt, ha x nem szerepel a listán. Ha a listán már van x, akkor a hozzá tartozó y-t az új y-nal helyettesiti, ha az nagyobb a listán szereplőnél.*
IN(L,x)	*megadja, hogy x kulcsu elem szerepel-e az L listában. Ha igen, akkor a válasz true, ha nem, akkor false.*
VALUE(L,x)	*megadja az L listán az x kulcshoz tartozó y-t. Ha a listán a megadott x nem szerepel, akkor az eredmény 0.*
SUM(L)	*képzi a listán szereplő elemek y értékeinek összegét.*

4.2 Korlátos FIFO (1994. január 19.)

Algebrai axiómák segítségével specifikálja az alábbi műveletekkel jellemzett korlátos FIFO-t!

CRT(x)	*új, **x** kapacitású **f** FIFO-t hoz létre.*
SIZE(f)	*egész, **f** kapacitása.*
FREE(f)	*egész, **f** szabad helyeinek száma.*
PUT(f, item)	***f**-be **item**-et tesz, ha **f**-ben van legalább egy szabad hely. Ha nincs, akkor hatástalan.*
HEAD(f)	*megadja **f** legöregebb elemét. Ha **f** üres, akkor NIL-t ad vissza.*
TAIL(f)	***f**-ből eltávolítja annak legöregebb elemét. Ha **f** üres, akkor hatástalan.*

4.3 String II. (1998. január 7.)

Algebrai axiómák segítségével specifikálja az alábbi műveletekkel jellemzett stringet!

CRT( )	*új (üres) stringet hoz létre*
APPEND(s , x)	*az s string végére rakja az x karaktert*
ISEND(s1,s2)	*igaz, ha az s2 string az s1 végén áll*
SUBS(s1,s2)	*igaz, ha az s2 string az s1 stringben rész-stringként megtalálható*
LGTH(s)	*megadja az s string hosszát*

4.4 Konténer (1994. január 26.)

Algebrai axiómák segítségével specifikálja az alábbi műveletekkel jellemzett konténert

CRT()	*új, üres konténert hoz létre.*
PUT(k,item)	***k**-ba **item**-et tesz.*
HEAD(k)	*megmutatja a **k**-ban található legkisebb **item**-et.*
TAIL(k)	***k**-ból eltávolítja annak legkisebb item-jét.*
EMPTY(k)	*igaz, ha a k konténer üres*

4.5 String III. (1996. január 25.)

Algebrai axiómák segítségével specifikálja az alábbi műveletekkel jellemzett stringet! A string karakterei elõrõl 1-gyel kezdõdõen számozottak.

CRT()	*új (üres) stringet hoz létre.*
APPEND(s, x)	*az **s** string végére rakja az **x** karaktert.*
DLT(s,i)	*az **s** string **i**-ik karakterét törli és a stringet tömöríti. Ha **i** nagyobb, mint a string hossza, akkor a műveletet hatástalan.*
IN(s, i)	*eredményül adja az **s** string **i**-ik karakterét. Ha **i** nagyobb, mint a string hossza, akkor az eredmény értelmetlen (nem definiált).*
LGTH(s)	*az **s** string karaktereinek számát adja.*

5 Gyakorlófeladatok megoldásai

5.1 Számpárok

- **IN(NEW(),x) == false** *(új listában semmilyen kulcs sincs)*
- **VALUE(NEW(), x) == 0** *(új listában elemek sincsenek)*
- **SUM(NEW()) == 0** *(üres listában az összeg is nulla)*
- **IN(ADD(L,(x1,y)), x2)== (x1==x2) OR IN(L,x2)**

  (az, amit most teszünk be, biztosan bent van, ha pedig mást keresünk mindegy a sorrend)

- **VALUE(ADD(L,(x1,y)),x2)= if (IN(L,x2)) then**
  if (x1 == x2) and (y > VALUE(L,x1)) then y
  else VALUE(L,x2)
  else if (x1 == x2) then y else 0

  (már biztosan szereplő betétele esetén az érték nőhet, ha pedig nem arra a kulcsra kérdezünk rá, akkor mindegy, hogy mit csinálunk…)

- **SUM(ADD(L,(x,y))) == if (IN(L,x)) then**
  if (y>VALUE(L,x)) then SUM(L)-VALUE(L,x)+y
  else SUM(L)
  else SUM(L)+y

(Ha már meglévő kulcshoz rakunk elemet, akkor az összeg nőhet, ha nagyobb elemet rakunk a kulcs mellé, ha kisebbet, akkor az összeg változatlan. Végül, ha új kulcsot veszünk fel, az összeg csak nőhet.)

- ugyanez az axióma egy kicsit tömörebben is leírható: **SUM(ADD(L,(x,y))) == SUM(L)+y-min{y, VALUE(L,x)}**
  (Ez a forma azonban átláthatatlan és használ más axiómát is. Ilyet lehetőleg csak akkor írjunk le, ha annak helyességében biztosak vagyunk és át is látjuk azt.)

5.2 Korlátos FIFO

- **SIZE(CRT(x)) == x** *(x kapacitással csinált FIFO kapacitása tényleg x)*
- **SIZE(PUT(f, item)) == SIZE (f)** *(a FIFO kapacitása nem függ az elemek számától)*
- **FREE(CRT(x)) == x** *(üresben minden hely szabad)*
- **HEAD(CRT(x)) == NIL** *(üres elején nincs semmi)*
- **TAIL(CRT(x)) == CRT(x)** *(üresnél nincs üresebb)*
- **FREE(PUT(f, item)) == if (FREE (f) == 0) then 0**
  else FREE (f) – 1
  (ha befér akkor csökken a hely, ha nem, nem)
- **HEAD(PUT(f, item)) == if (SIZE(f) == FREE(f)) then item**
  else HEAD(f)
  (FIFO-ban csak akkor lesz a legújabb a legöregebb elem, ha előtte az üres volt, különben marad a legöregebb elem)
- **TAIL(PUT(f, item)) == if (SIZE(f) == FREE(f)) then CRT(SIZE(f))**
  else if (FREE(f) == 0) then TAIL(f)
  else PUT(TAIL(f), item)

(üresből nincs mit törölni; ha a FIFO-ban már nincs hely, akkor a PUT() hatástalan; különben a legöregebb és a legújabb elemre vonatkozó utasítások felcserélhetők)

5.3 String II.

- **LGTH(CRT()) == 0** *(üres string üres)*
- **LGTH(APPEND(s,x)) == LGTH(s) + 1** *(a stringet az append művelet pontosan egy karakterrel bővíti)*
- ISEND(CRT(), CRT()) == true
  ISEND(APPEND(s_1,x), CRT()) == true , de a fenti kettő megint összevonható:
  ISEND(s , CRT( )) == true *(üres string minden string végén áll)*

- **ISEND(CRT(), APPEND(s,x)) == false**
- **ISEND(APPEND(s_1, x_1), APPEND(s_2,x_2)) == if (x_1==x_2) then ISEND(s_1,s_2) else false**
- SUBS(CRT(), CRT()) == true, illetve SUBS(APPEND(s_1,x), CRT()) == true összevonhatók:
- **SUBS(s , CRT( )) == true** *(üres string minden stringben található)*
- **SUBS(CRT(), APPEND(s,x)) == false** *(üres stringben viszont nincs semmi)*
- **SUBS(APPEND(s_1, x_1), APPEND(s_2,x_2)) == if (x_1==x_2) then**
  ISEND(s_1,s_2) | SUBS(s_1,APPEND(s_2,x_2))
  else SUBS(s_1,APPEND(s_2,x_2))
  (A részstring viszony úgy állhat elő, hogy vagy már benne van a bővítetlen stringben, vagy a két string végére ugyanazt a karaktert tettük és a második már az első végén állt. A | logikai vagy műveletet jelöl.)

5.4 Konténer

- **HEAD(CRT( )) == undefined** *(üres konténernek nincs legkisebb eleme)*
- **TAIL(CRT()) == CRT()** *(üresnél nincs üresebb)*
- **EMPTY(CRT()) == true** *(ami üres az üres)*
- **EMPTY(PUT(k,i)) == false** *(amibe tettünk már valamit, biztosan nem üres)*
- **HEAD(PUT(k, i)) == if (EMPTY(k)) then i else min {HEAD(k); i}**
  (üres konténerbe biztosan most tesszük bele a legkisebb elemet, míg ez nem üres esetén nem biztos, hogy az eddigi minimálisnál kisebb elemet teszünk be)
- **TAIL(PUT(k, i)) == if (EMPTY(k)) then CRT()**

else if (i<HEAD(k)) then k
else PUT(TAIL(k), i)

(üres konténer a két művelet egymás utáni alkalmazásával is üres marad; a nem üresbe, ha betesszük és egyben ki is vesszük a legkisebb elemet, nem csináltunk semmit)

5.5 String III.

- **LGTH (CRT()) == 0** *(üres string 0 hosszú)*
- **IN(CRT(),i) == undefined** *(üres stringben sehol sincs elem)*
- **DLT(CRT(), i) == CRT()** *(üres stringnél nincs üresebb)*
- **LGTH(APPEND(s,x)) == LGTH(s) + 1** *(a stringet csak és kizárólag az APPEND művelet bővíti, pontosan egy karakterrel)*
- **IN(APPEND(s,x), i) == if (i>LGTH(s)+1) then undefined**
  else if (i==LGTH(s)+1) then x
  else IN(s,i)

  (ha a beszúrás előtt akarunk egy karaktert lekérdezni, akkor e művelet szempontjából a beszúrás elhanyagolható; ha éppen a beszúrt karakterre kérdezünk rá, akkor azt tudjuk; az eredő string hosszánál messzebb pedig nincsen semmi)
- **DLT(APPEND(s,x), i) == if (i>LGTH(s)+1) then APPEND(s,x)**
  else if (i==LGTH(s)+1) then s
  else APPEND(DLT(s,i), x)

  (ha túl messze akarunk törölni, akkor a művelet hatástalan; ha éppen a beszúrt karaktert töröljük, akkor nem csináltunk semmit; ha pedig a beszúrás előtt törlünk, akkor mindegy a sorrend)