2. AZ ADATTÍPUS ABSZTRAKCIÓS SZINTJEI

Ebben a tárgyban nem fejlesztünk, és nem veszünk át más forrásból egy típuselméletet. Az alábbiakban inkább csak egy szemléletmódról lesz szó. Arra a kérdésre próbálunk válaszolni, hogy az adattípusok, adatszerkezetek milyen absztrakciós szinten jelennek meg az elméleti vizsgálatok és a gyakorlati problémamegoldás során. Ezen absztrakciós szintek, felfogásunk szerint, a következők:

1. Absztrakt adattípus (ADT)
   1.1. Algebrai specifikáció
   1.2. Funkcionális specifikáció
2. Absztrakt adatszerkezet (ADS)
3. Reprezentáció (ábrázolás)
   3.1. Aritmetikai (tömbös) ábrázolás
   3.2. Láncolt (pointeres) ábrázolás
4. Implementáció (megvalósítás)
5. Fizikai szint

Egy elméletileg igényes programozó, aki specifikál, tervez és fejleszt, továbbá – mondjuk hiba esetén – a nyomkövetés (debugging) eredményét (trace) is nézegeti és az adatszerkezet memóriabeli bitképét tanulmányozza, felfogásunk szerint a fenti öt szinten találkozik az adattípusokkal.

Megjegyzések:

1.) A 4. és 5. pont nem része a tematikának, ezért ezek nincsenek kidolgozva a jegyzetben. Az implementációs szinten mindig valamilyen nyelvre vagy fejlesztő eszközre gondolunk, amelyen az egyes adatszerkezeteket „beprogramozzuk”. A fizikai szint az adatszerkezet bitképét jelenti, ami a számítógép memóriájában megvalósul (pl. lebegőpontos valós szám).

2.) A *típus* és az *adatszerkezet* kifejezéseket általában szinonimaként használjuk, noha a típus elvileg többet fejez ki (ADS szinten a szerkezet, az invariánsok és a műveletek hármasát jelenti). Ezzel szemben az adatszerkezet – régebbi eredetű – kifejezés is jelentheti mindezt, de eredetileg csak magát a struktúrát jelenti, műveletek nélkül.

3. Az itt szereplő példák (pl. verem, sor, kupac, rendező fa) precíz definíciójára később visszatérünk, most csak intuitív módon használjuk őket.

2.1. Absztrakt adattípus (ADT)

Ez a szint az adattípus leírásának legmagasabb absztrakciós szintje. A tankönyvek nagyobb részében nem szerepel ez a leírási mód. Az adattípust úgy specifikáljuk, hogy szerkezetére, reprezentálására, implementálására semmilyen megfontolást, előírást, utalást nem teszünk. A leírásban kizárólag matematikai fogalmak használhatók.

Az ADT szintű leírás lehet formális (mint a következő két példában), de lehet informális is: pl. természetes magyar nyelven is elmondhatjuk, hogy mit várunk el egy veremtől. A lényeg nem a formalizáltság mértéke, hanem az, hogy „nem látjuk” az adattípus belső struktúráját.

Mire jó az ADT szintű leírás?

– Hasonlóan a „Bevezetés a programozáshoz” c. tárgy specifikációs (A, B, Q, R) modelljéhez és sok algoritmikus példához, vannak olyan problémák is, melyekben éppen az adattípus reprezentálása a kérdés, pl. az elsőbbség (prioritásos) sor hatékony megvalósításának problémájánál. (Míg a legtöbben tudják, hogy milyen a verem, vagy a sor adatszerkezet, addig kevesen ismerik az elsőbbségi sort.) Ilyen esetekben célszerű, hogy úgy tudjuk leírni az adattípussal kapcsolatos elvárásainkat, hogy nem teszünk megszorítást a szerkezetre nézve.

– A leírás közvetíti az „enkapszuláció” gondolatát. Ha valaki egy típust implementál, akkor az ezt tartalmazó modult (hívható, végrehajtható program-egységet) úgy írja meg, hogy magához az adatszerkezethez a felhasználó közvetlenül ne férhessen hozzá, csak a műveleteken keresztül érhesse el azt. A másik oldalról pedig (ugyanebben a szellemben) a program felhasználója is elfogadja, hogy egy adatszerkezetet eleve csak annak műveleteivel manipulálja és „nem nyúl bele” abba.

Alapvetően két leírási mód terjedt el:

1. algebrai specifikáció (axiómák megadásával),
2. funkcionális specifikáció (elő- és utófeltétellel).

2.1.1. Algebrai (axiomatikus) specifikáció

Példa: Az E alaptípus feletti V verem absztrakt adattípus (ADT) axiomatikus jellemzése három lépésben. A verem „lényege” az, hogy a legutoljára betett elemet lehet kivenni belőle.

1. Műveletek (mint leképezések)

Empty:	$\rightarrow V$	Üres verem konstans; az üres verem „létrehozása”
IsEmpty:	$V \rightarrow L$	A verem üres voltának lekérdezése
Push:	$V \times E \rightarrow V$	Elem betétele a verembe
Pop:	$V \rightarrow V \times E$	Elem kivétele a veremből
Top:	$V \rightarrow E$	A felső elem lekérdezése

Megjegyzés: Definiálhatnánk még az *IsFull* lekérdezést is, de (hasonlóan a témával foglalkozó könyvek többségével) ezt nem tesszük meg.

2. Megszorítások a műveletekre (értelmezési tartományok)

Az üres veremből nem lehet elemet kivenni, és a felső elemét sem lehet lekérdezni.

$D_{Pop} = V \setminus \{Empty\}$

$D_{Top} = V \setminus \{Empty\}$

3. Axiómák

1. $IsEmpty(Empty)$ vagy $v = Empty \rightarrow IsEmpty(v)$
2. $IsEmpty(v) \rightarrow v = Empty$

3. $\neg IsEmpty(Push(v,e))$
4. $Pop(Push(v,e)) = (v,e)$
5. $Push(Pop(v)) = v$
6. $Top(Push(v,e)) = e$

Az axiómákat úgy írjuk fel, hogy sorra vesszük az öt művelet (ill. egy konstans és négy művelet) összes lehetséges párosítását – mindkét sorrendben –, és ha van értelme az adott egymásra hatásnak, akkor azt kifejezzük egy logikai állítással.

Érezzük, hogy axiómáink nem tartalmaznak *ellentmondást*, tehát a rendszer *helyes*. Nem foglalkozunk azzal, hogy az axiómarendszer *redundáns-e*; az ember általában szeret egy kevés redundanciát használni, főleg ha kalkulus szabályairól van szó. Legfontosabb a *teljesség* megléte, de azt is csak „érzésre" teljesítjük.

2.1.2. Funkcionális (elő-, utófeltételes) specifikáció

Példa: Az E alaptípus feletti S sor absztrakt adattípus leírása (itt csak egy részlete) elő- és utófeltételekkel. A verem és a sor sajátos „duálist" alkotnak, mert a sorból a legrégebben betett elemet lehet kivenni.

Ebben a leírásban típust matematikailag reprezentáljuk, de ez semmilyen módon nem utal a számítástechnikai reprezentációra! A matematikai reprezentációra azért van szükség, mert itt nem a műveletek kölcsönhatásainak eredményét írjuk le, hanem minden művelet hatását önmagában jellemezzük, és azt valahogyan ki kell fejezni az adattípuson végbement változással.

Egy szellemes ötlettel az $s \in S$ sor absztrakt szinten elempárok halmazának tekinthető, ahol az elempár első komponense a sorba betett E-beli érték, a második komponense pedig a behelyezés időpontja:

$$s = \{(e_1,t_1),(e_2,t_2),...,(e_n,t_n)\}, \quad \text{ahol } n \geq 0, \text{ és } \forall i,j \in \{1,...,n\}: i \neq j \Rightarrow t_i \neq t_j,$$

vagyis minden elem különböző időpontban lépett be a sorba.

Írjuk le pl. a sorból való kivétel műveletének szemantikáját. Az *Out*: $S \rightarrow S \times E$ művelettel a sor legrégebben betett elemét vesszük ki. A sor nem lehet a kivétel előtt üres, ezért $D_{Out} = S \setminus \{Empty\}$. Az alábbi specifikáció követi a „Bevezetés a programozáshoz" c. tárgyban tanultakat.

$$A = \underset{s}{S} \times \underset{e}{E}$$

$$B = \underset{s'}{S}$$

$$Q = (s = s' \wedge s \neq \varnothing)$$

$$R = (s = s' \setminus \{(e,t)\} \wedge (e,t) \in s' \wedge \forall i \left((e_i,t_i) \in s'\right): t \leq t_i)$$

R-ben azt fogalmaztuk meg, hogy a sorból, mint halmazból, mindig azt a párt vesszük ki, amelyet legelőször tettünk be. (A leírásban azért használjuk a $\leq$ relációt, mert ahogy az (e_i,t_i) pár végig fut az s halmazon, magát a kiválasztott (e,t) párt is felveszi értékül. Mivel minden időpont különböző, csak ebben az egy esetben áll fenn egyenlőség, tehát az (e,t) pár egyértelműen meghatározott.

Még egyszer hangsúlyozzuk, hogy senki sem implementálja a sort így, rendezett párok halmazaként, noha ez elvben lehetséges lenne.

2.1.3. Algoritmusok leírása (probléma-megoldás) ADT-szinten

Amikor problémát oldunk meg, egy adattípus felhasználásáról – legalábbis ADT-szinten – úgy gondolkodhatunk, hogy csak a műveletei által tudunk hozzáférni, sőt arról sincs tudomásunk, hogy ezek a műveletek hogyan működnek. A típus „fekete doboz”, ugyanis ezen a szinten semmilyen megkötést nem teszünk a szerkezetéről.

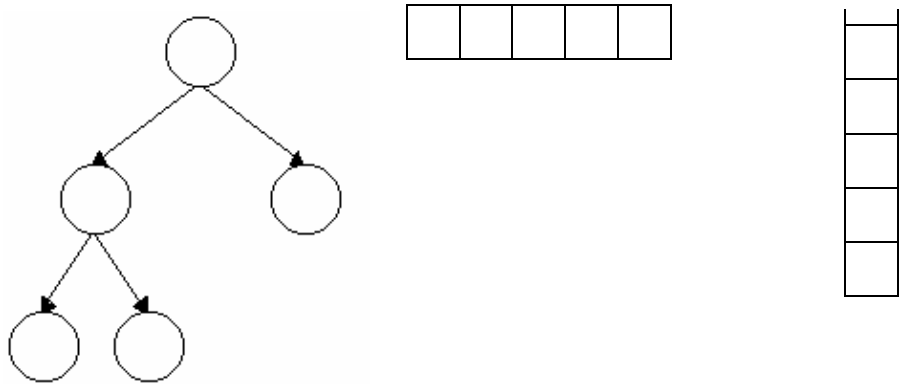
Számos példát látunk majd az ADT szintű algoritmus leírásra. Pl. a *helyes zárójelezés feldolgozásának* algoritmusa így használja a vermet (lásd: ott). Szembeötlő a különbség, ha egy listán annak (definiált és megírt) műveleteivel dolgozunk, szemben azzal, amikor a lista mutatóit kezelve valósítjuk meg ugyanazt az eredményt.

Érdekes példát láthatunk a műveletek szintjén való gondolkodásra a prioritásos sornál. Ott szerepel a *prioritásos sorral való rendezés* algoritmusa (lásd: 6. előadás). Az algoritmus két ciklus szekvenciája: először egyesével betesszük a rendezendő elemeket a sorba, majd egymás után kivesszük és kiírjuk őket. A prioritásos sor – belső szerkezetétől független – működése garantálja azt, hogy az elemeket nagyság szerint csökkenő sorrendbe rendezve kapjuk meg.

2.2. Absztrakt adatszerkezet (ADS)

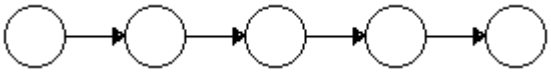
Az ADS szinten egy fontos döntést hozunk meg az ADT szintű specifikáció alapján. Megmondjuk azt, hogy alapvetően milyen struktúrája van a szóban forgó adattípusnak. Közelebbről ez azt jelenti, hogy megadjuk az adatelem közötti legfontosabb rákövetkezési kapcsolatot, és ezt egy irányított gráf formájában le is rajzoljuk. Az ADS szintet egy szerkezeti gráf és az ADT szinten bevezetett műveletek alkotják együttesen.

Ez a szint illeszkedik legjobban az ember *kognitív pszichológiai* tulajdonságaihoz. Egy veremről nem az axiómák formájában őrzünk képet emlékezetünkben, hanem egy (feltehetően függőleges) tömbszerű tároló jelenik meg előttünk. Hacsak mást nem mondunk, akkor ezen a szinten beszélünk és gondolkodunk az adatszerkezetről. Ez a magyarázat, a megértés, a felidézés és az algoritmizálás, általában a kreatív problémamegoldás leggyakoribb kognitív szintje. Az egyes adatszerkezetek említésekor megjelenik előttünk egy ilyen szintű ábra.



2.1. ábra. A bináris fa, a sor és a verem szerkezeti rajza

Az utóbbi két adatszerkezet is nyilván irányított gráf:

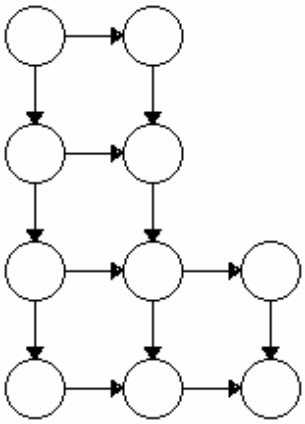


2.2. ábra. A sor és a verem lineáris adatszerkezet

Az ADS szinten az adattípus legfontosabb szerkezeti összefüggéseit adjuk meg egy irányított gráffal. A gráf csúcspontjai adatelemeket azonosítanak, az irányított élek pedig a közöttük fennálló rákövetkezési relációt ábrázolják. A fenti ábrák mindegyike nyilvánvalóan meghatároz egy-egy irányított gráfot.

Az adatszerkezeteket osztályozhatjuk az általuk meghatározott gráf alakja szerint:

- lineáris,
- kétirányú vagy szimmetrikus lineáris,
- fa (struktúrájú),
- ortogonális (pl. mátrix; lásd még: 2.3. ábra), ill.
- többszörösen összefüggő = általános gráf alakú adatszerkezet.



2.3. ábra. Egy ortogonális adatszerkezet

Példa: A *kupac* (*heap*) absztrakt adatszerkezet (ADS) leírása: olyan majdnem teljes, balra tömörített bináris fa, amelyben minden nem-levél csúcsra igaz, hogy a benne lévő érték nagyobb (vagy egyenlő), mint a gyerek csúcs(ok)ban lévő érték(ek).

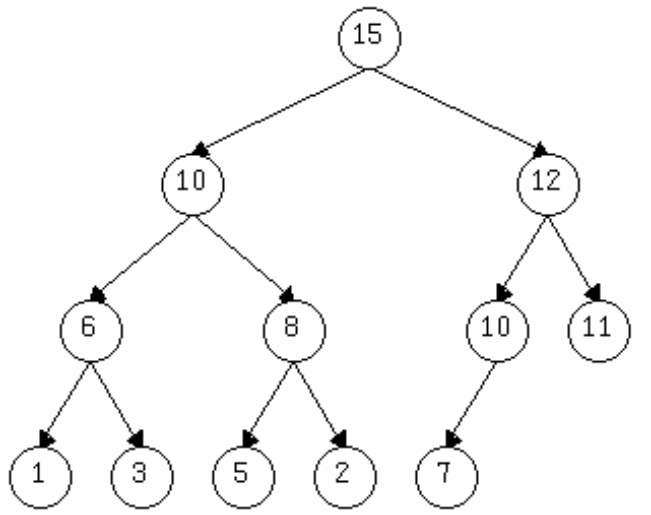
A meghatározás az adatszerkezethez tartozó gráf (bináris fa) alakját megszorítja két invariánssal (majdnem teljes és balra tömörített), továbbá megad egy tartalmi invariánst is. A 2.4. ábra egy heap-et szemléltet.

Fontos megjegyzések az ADS-ről:

- Az ADS-gráfban az adatszerkezet csupán a *legalapvetőbb* szerkezeti tulajdonságait szokás ábrázolni. Pl. egy bináris fában nem használunk kettős (oda-vissza) nyilakat, hogy esetlegesen egy csúcsból a szülőjéhez is eljuthassunk. Hasonlóan nem kötjük össze éllel a két testvércsúcsot, noha egyes esetekben erre a kapcsolatra is szükségünk lehet.

- Ha az adatelemek között olyan kapcsolatra lenne szükség egy probléma megoldásánál, amelyet az ADS-gráfban ábrázolt rákövetkezési relációkkal csak bonyolultan lehetne felírni, vagy éppenségges sehogyan sem, akkor is nyugodtan bevezethetjük a megfelelő függvényeket (pl. bináris fa esetén a *szülője*(*c*), vagy *testvére*(*c*) függvényeket). Általában, bármilyen

függvényt vagy tulajdonságot szabadon bevezethetünk anélkül, hogy a megvalósításán gondolkodnánk.



2.4. ábra. Egy kupac (heap)

- A fenti „szabad gondolkodásmódra” példa a kupacrendezés (heap sort) azon változata, amelyben a heap-fa éleinek színe van: az élek eredetileg fehérek és az eljárás minden iterációban beszínez egyet pirosra. Ugyanebben a változatban hivatkozunk a fa „alsó-jobb” elemére (más néven „utolsó” elemére) is, anélkül, hogy a megvalósítás nehézségeivel foglalkoznánk.
- Az ADS-szintű típushoz is hozzátartoznak a műveletek, még akkor is, ha esetenként ezt elfelejtik említeni az ismertetések. Sőt, az egyes műveletek hatását szemléletesen magyarázhatjuk az ADS-gráfon kifejtett hatásukkal.

Az iménti, műveletekről tett megjegyzéshez kapcsolódva, az ADS szintű algoritmus-leírás kérdését érintjük röviden. Nincs olyan formai jegy, amely alapján egyértelműen el tudnánk dönteni, hogy egy absztrakt algoritmus-leírás ADT vagy ADS szintűnek mondható-e. Azt kellene ehhez megválaszolni, hogy az algoritmus megadásában kifejeződik-e annak ismerete, hogy milyen a szereplő adattípus(ok) struktúrája. Ha pl. egy algoritmus bináris fát használ és (függvények formájában) hivatkozik egy csúcs bal vagy jobb gyerekére, szülőjére, esetleg testvérére, akkor gyanítható, hogy a szerző „látja maga előtt” a bináris fa szerkezetét. Ám ezek a függvények bevezethetők és definiálhatók ADT szinten, a strukturális szemlélet nélkül is. Ez az eldöntetlenség egyáltalán nem zavaró.

Példa: Egy bináris fa preorder bejárása. (Előadáson)

2.3. Reprezentációs szint

Ezen a szinten arról hozunk döntést, hogy az ADS-szinten megjelent gráf rákövetkezési relációit milyen módon ábrázoljuk. Az ábrázolás még mindig absztrakt, noha már a számítógépes megvalósítást modellezi.

Két tiszta reprezentálási mód van, illetve ezeket lehet vegyesen is alkalmazni:

1. aritmetikai (tömbös) ábrázolás, ill.
2. láncolt (pointeres) ábrázolás.

2.3.1. Aritmetikai ábrázolás

A memóriát úgy képzeljük el, mint egy hosszú, egydimenziós tömböt, vagyis vektort, amelynek elemei byte-ok. Ezen alapul az aritmetikai ábrázolás első változata.

a) Címfüggvényes aritmetikai ábrázolás: az absztrakt memória-vektor adott kezdőcímétől kezdve elhelyezzük az adott adatszerkezet elemeit. Egy adatelem több byte-ot is elfoglalhat. Meg kell adni a címfüggvényt, amelynek segítségével minden adatelem helye (címe) meghatározható.

Példa: Az $A[1..m,1..n]$ kétdimenziós tömb (mátrix) cím(A) kezdőcímtől kezdett sorfolytonos elhelyezésekor a címfüggvény a következő lesz (ahol egy elem l byte-ot foglal el):

$$cím\ (A[i,j]) = \text{cím}(A) + [n(i\text{-}1) + (j\text{-}1)]\cdot l \qquad (1\le i\le m, 1\le j\le m)$$

b) Indexfüggvényes aritmetikai ábrázolás: ez az absztraktabb ábrázolás. Az adatszerkezet elemeit egy olyan tömbben helyezzük el, amelynek nem byte-ok az elemei, hanem az adatszerkezet alaptípusának példányai. Az elhelyezést az első elemtől kezdjük, és ekkor egy indexfüggvényt kell megadnunk.

Példa: A fenti $A[1..m,1..n]$ kétdimenziós tömb (mátrix) sorfolytonos elhelyezésének indexfüggvénye a következő lesz:

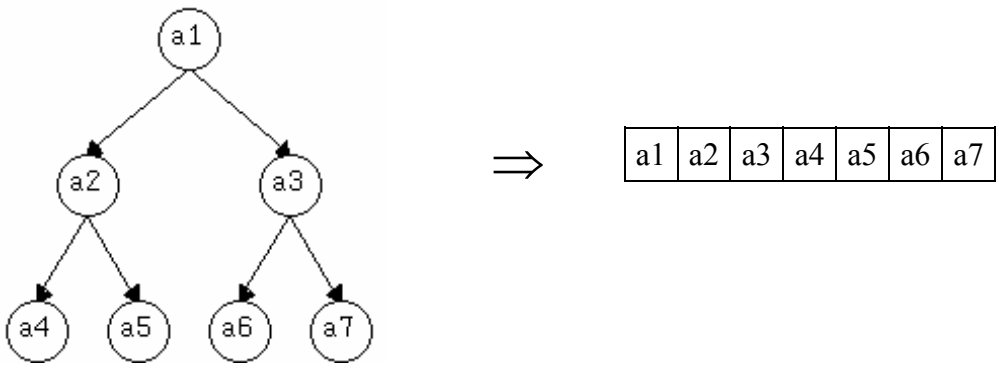
$$ind\ (A[i,j]) = n(i\text{-}1) + j \qquad (1\le i\le m, 1\le j\le m)$$

Példa: Helyezzünk el egy teljes bináris fát szintfolytonosan egy ugyanolyan alaptípusú tömbben, mint amilyen a fa elemeinek típusa. Ekkor egy c belső csúcs bal gyereke kétszer akkora indexszű cellába kerül, mint a c csúcs:

$$ind(bal(c)) = 2\cdot ind(c)$$

$$ind(jobb(c)) = 2\cdot ind(c) + 1$$

Az összefüggés helyességét gyakorlaton bizonyítani fogjuk.



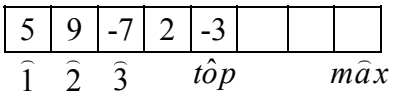
2.5. ábra. Teljes bináris fa szintfolytonos elhelyezése tömbben

A fenti két példában szereplő cím-, ill. indexfüggvény között fennáll még egy különbség: míg az első explicit módon adja meg az elemek címét, addig a második rekurzívan, a rákövetkezési relációt követve a szülő csúcs indexéből számítja ki a gyermek csúcs indexét.

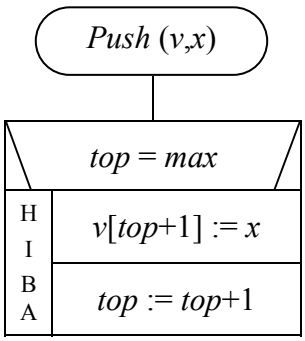
Mihelyst egy adatszerkezetet reprezentáltunk, meg kell adnunk a műveletek algoritmusait az adott (aritmetikai ill. láncolt) ábrázolási módban.

Példa: Teljes bináris fa inorder bejárása tömbös reprezentáció esetén. (Előadáson)

Példa: A verem *Push* művelete tömbös ábrázolás esetén (lásd: 2.6. ábra).



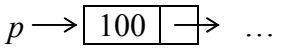
A vermet ekkor egy *v*[1..*max*] tömb tárolja/ábrázolja.



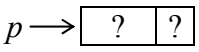
2.6. ábra. A verem Push művelete

2.3.2. Láncolt ábrázolás

Ehhez az ábrázoláshoz bevezetjük az absztrakt mutató (pointer) fogalmát. Ha *p* egy mutató típusú változó, akkor *p*ˆ jelöli az általa mutatott adatelemet. Ennek az adatelemnek az esetleges komponenseit (pl. adat és mutató) *p*ˆ.*adat*, ill. *p*ˆ.*mut* jelöli. A sehová sem mutató pointer értéke NIL.



Megegyezünk abban, hogy az absztrakt mutató típusos, azaz mindig valamilyen meghatározott típusú adatszerkezetre mutat. A láncolt ábrázolás algoritmusaiban tipikus helyzet, hogy egy új adatelemet kell létrehozni, helyet foglalni neki a memóriában (ezzel mintegy kiszakítva a szabad memóriából). Ezt a *new*(*p*) absztrakt utasítással tehetjük meg. Ennek hatására létrejön egy adott típusú, definiálatlan tartalmú adatelem, amelyre a *p* változó mutat.



Egy ilyen adatelem felszabadítása a *dispose*(*p*) absztrakt utasítással történik. Hatására az adatelem visszakerül a szabad helyek közé és *p* tartalma meghatározatlan lesz (általában még mindig az eldobott adatra mutat). Megállapodunk abban is, hogy a szabad helyek listájára az *SZH* pointer mutat, és annak lekérdezése, hogy van-e még szabad hely a memóriában, az

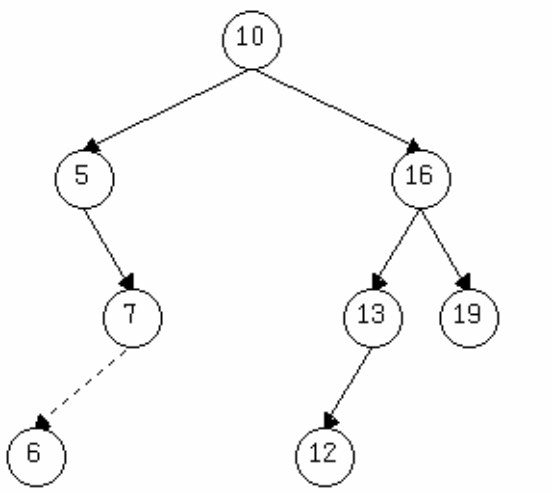
SZH = NIL kérdéssel történik. (Absztrakt algoritmusainkban a szabad helyeket nem kell kezelni, feltételezünk egy olyan mechanizmust, mint amit egy operációs rendszer biztosít a gyakorlatban.)

Példa: A verem *Pop* művelete láncolt ábrázolás esetén. (Előadáson)

Példa: Bináris fa postorder bejárása láncolt ábrázolás mellett. (Előadáson)

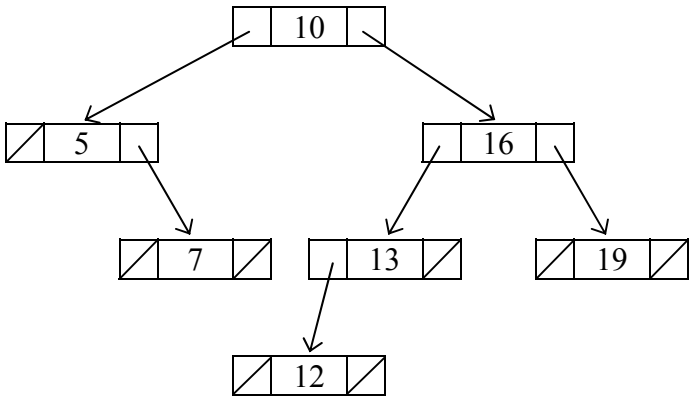
Példa: Egy *keresőfa* láncolt ábrázolását mutatjuk be. A keresőfa olyan bináris fás adatszerkezet, amelyben minden csúcsban az érték nagyobb, vagy egyenlő, mint a bal gyerek csúcsában levő érték, és kisebb, mint a jobb gyerek csúcsában lévő érték (természetesen csak a megfelelő gyerek csúcsok létezése esetén).

A keresőfa egyik művelete az, amely egy új értéket helyez el a fában a megfelelő helyre, pl. a 6-ot a következő ábrán látható helyre.



2.7. ábra. Egy adott keresőfába beszúrjuk a 6-ot

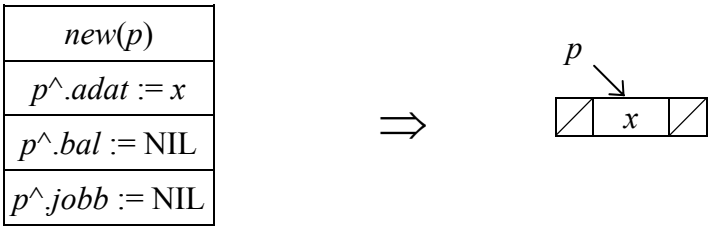
A 2.8. ábrán az látható, hogy a fenti ADS szinten lerajzolt fát hogyan reprezentáljuk mutatók alkalmazásával.



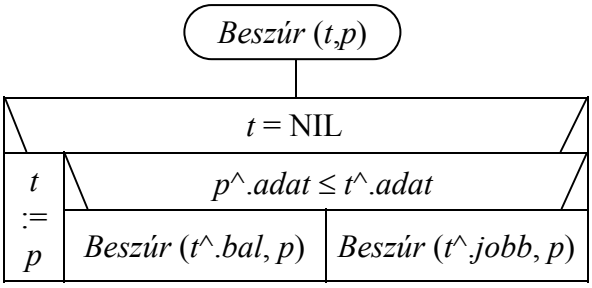
2.8. ábra. A keresőfa láncolt ábrázolása.

A NIL mutatókat a „/” ferde vonal jelöli az ábrán.

Ha keresőfába való beszúrást szeretnénk megírni a láncolt ábrázolás szintjén, akkor először az x (a fenti példában $x = 6$) értékkel létrehozunk egy fába beszúrható csúcsot, amelyre a p pointer mutat.

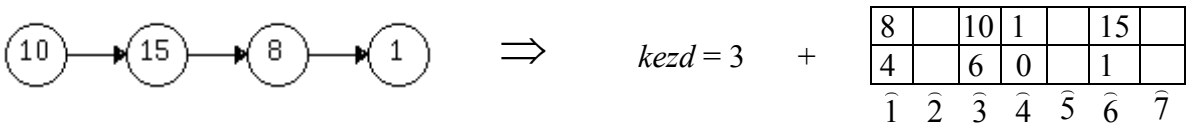


A t pointer által mutatott fába való beszúrás másik paramétere a p mutató, amely a beszúrásra alkalmas, összetett adatelemre mutat. A beszúrás rekurzív algoritmusa a következő:



Megjegyzés: A fás algoritmusokban az ADS szint és a láncolt ábrázolás között hasonlóság áll fenn, ami azonban inkább csak vizuális jellegű, a rajzok hasonlósága miatt. Más jelöléseket használunk a két szinten: az ADS szintű $t = \Omega$ helyett t = NIL, a $bal(t)$ helyett $t\hat{}.bal$ jelenik meg a láncolt ábrázolás szintjén.

Az itt bemutatott ábrázolást úgy is nevezhetjük, hogy *dinamikusan láncolt*, vagy valódi pointeres ábrázolás. Ezzel szemben *statikus láncolt* ábrázolásnak nevezhetjük azt a kuriózumnak tűnő reprezentációt, amelyben a láncolt ábrázolású adatszerkezetet egy kétdimenziós tömbben helyezzük el úgy, hogy a mutatókat indexek jelenítik meg. Pl.:



2.9. ábra. Lista ábrázolása statikus láncolással

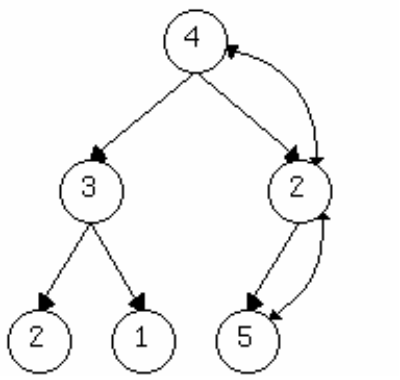
Mint a példán látható, az adatszerkezet ekkor egy rekord, mely egy *kezd* nevű, kezdő indexet jelölő változóból és a kétdimenziós tömbből áll. A tömb első sorában az adott alaptípusú értékek helyezkednek el, míg az értékek alatt, a tömb második sorában a következő érték indexe található. Az utolsó értéknek nincs rákövetkezője, ezért alá egy nemlétező (a példában 0) index kerül. Az így megadott rákövetkezések egyértelműen meghatározottak. (A tömb szabad elemeit felfűzhetjük egy másik, ún. szabad listába.)

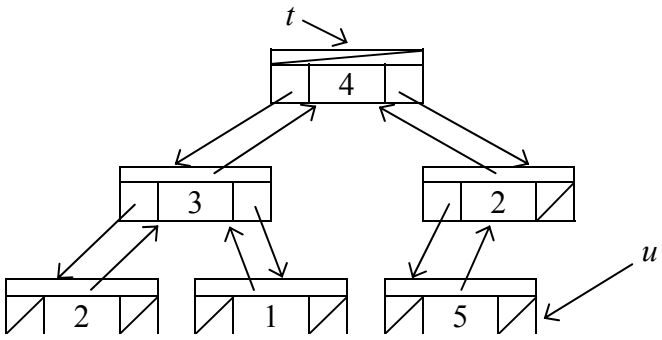
2.3.3. Mit ábrázolunk a reprezentáció szintjén?

Azt mondtuk, hogy ezen a szinten az ADS-gráf éleit, vagyis az adatelemek közötti rákövetkezési relációkat ábrázoljuk. Ezt támasztja alá pl. a keresőfa reprezentálása is (lásd: 2.8. ábra).

Ez az ábrázolás azonban gyakran nem olyan egyszerű, mint a fenti példa mutatja. Ha az ADS-szinten definiált függvények, tulajdonságok, műveletek megvalósítása, illetve a megoldandó feladat algoritmusa szükségessé teszi, akkor más, *új kapcsolatokat* is ábrázolhatunk az adatelemek között.

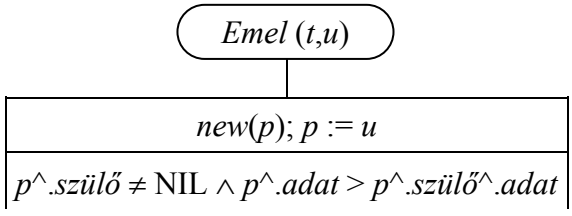
Példa: Tegyük fel, hogy egy heap fában az alsó szint jobb szélső eleme, mint újonnan beszúrt elem, olyan értéket tartalmaz, amely elrontja a heap-tulajdonságot, pl. a mellékelt ábrán az 5-ös elem valóban ilyen.





2.10. ábra. „gyerek-szülő” kapcsolattal kibővített bináris fa-ábrázolás

Ezt az elemet „fel kell vinni” cserékkel a gyökér felé vezető úton a helyére; itt most egészen a gyökérig. Ha ezt az eljárást pointeres ábrázolásban szeretnénk megírni, akkor célszerű a szülő csúcsra mutató pointert is felvenni az adatszerkezet részeként. A fa gyökerére (tulajdonképpen a fára) mutasson a *t* pointer, míg az esetleg „rossz” utolsó elemre pedig az *u* mutató. Ekkor A heap tulajdonságot helyreállító eljárás:



Csere (*p^.adat*, *p^.szülő^.adat*)
p := *p^.szülő*

Ez a példa is mutatja, hogy a hatékony működés érdekében alkotó módon kell a reprezentációt kitalálnunk, megterveznünk! Bizonyos feladatokhoz például az a jó ábrázolás, ha a testvér csúcsokat láncoljuk össze.

Megjegyzés: VIGYÁZAT!!! A fenti példa mesterkélt annyiban, hogy soha, senki nem reprezentál heap-et láncolt bináris fával! A heap-et kizárólag tömbös ábrázolással szokás megvalósítani. A fák aritmetikai ábrázolása (lásd: 2.5. ábra) azért szerencsés, mert a megfelelő indexfüggvény segítségével könnyen elérhetjük egy csúcs szülőjét, és a heap alsó jobb elemének megtalálása triviális, míg a láncolt ábrázolásban ez, vagyis az *u* változó karbantartása nehézkes lenne.

Ez a megjegyzés jól megvilágítja az ADS szint jelentőségét. Pl. a heap-ről ADS szinten gondolkodunk, ide értve a heap-en operáló algoritmusok tervezését is, de a heap-et és algoritmusait a tömbös ábrázolással valósítjuk meg. Egyik sem lenne jó fordítva! Nem szívesen követnénk az elemek szabálytalan ugrálását egy tömbben, ahol nincs vizuális kapcsolat szülő és gyerek között. Másrészt nem lenne jó egy pointerekkel teletűzdelt struktúrában bonyolult algoritmusokat programozni, ahol bármikor újabb pointer felvételére kerülhet sor.