Rekurzió

Horváth Gyula
horvath@inf.elte.hu

1. Feladat: Sorbaállítások száma

Hány féleképpen lehet sorbaállítani az osztály tanulóit?
Bemenet: a tanulók n száma.
Kimenet: ahány félekeppen az n tanuló sorbaállítható.

Megoldás

Jelölje $P(n)$ a megoldás értékét n tanuló esetén. A Tanulókat az $1,\dots,n$ számokkal azonosítjuk.
$P(1) = 1$
Visszavezés kisebb méretű, ugyanilyen probléma megoldására.
Tekitsük azokat a sorbaállításokat, amelyek esetén az n-edik tanuló a sorban az első helyen áll, és jelöljük ezek számát $S(1,n)$-el. Általában, jelölje $S(i,n)$ azon sorbaállítások számát, ahol az n sorszámú tanuló a sorban az i-edik helyen áll. Tehát

$$P(n) = S(1,n) + S(2,n) + \cdots + S(n,n)$$

Nyilvánvaló, hogy $S(1,n)$ megegyezik $n-1$ tanuló összes lehetséges sorbaállításának számával, tehát $S(1,n) = P(n-1)$. Általában, azon sorbaállítások száma, ahol az n-edik tanuló a i-edik helyen áll, $P(n-1)$. Tehát, ha $n > 1$, akkor

$$P(n) = n * P(n-1)$$

$$P(n) = n * (n-1) * \cdots * 2 * 1$$

```
1    P=1;
2    for (i=2; i<=n; i++)
3      P=P∧i;
```

2. Feladat: Zsebpénz

n Euro zsebpénzt kaptunk. Minden nap veszünk pontosan egy dolgot a következők közül (zárójelben az ár szerepel) perec (1 Eu), fagylalt (2 Eu), csoki (2 Eu).
Számítsuk ki, hogy hányféleképpen költhetjük el a zsebpénzünket!

Megoldás

Jelölje $K(n)$ az n Eu lehetséges elköltéseinek a számát. A következő összefüggések állnak fenn:

- $K(1) = 1$ (csak egy perecet vehetünk)
- $K(2) = 3$ (vagy két perecet, vagy egy csokit, vagy egy fagyit vehetünk)
- $K(n) = K(n-1) + 2K(n-2)$ ha $n \geq 3$, (első alkalommal perecet, csokit vagy fagylaltot vehetünk).

A következő algoritmus adja meg a $K(n)$ függvényt:

```
1  long long K(int n){
2     if (n==1)
3        return 1;
4     else if (n==2)
5        return 3;
```

```
6       else
7          return K(n−1)+2∧K(n−2);
8   }
9   int main(){
10      long long h=K(22);
11      printf("%lld␣\n", h);
12  }
```

3. A rekurzió gyökerei:Peano axiómák

1. $0 \in \mathbb{N}$ (A 0 természetes szám)
2. $S(x) \neq 0$ (A 0 nem rákövetkezője egyetlen természetes számnak sem)
3. $S(x) = S(y) \Rightarrow x = y$
4. Ha $M \subseteq \mathbb{N}$ és $0 \in M$ és $\forall x(x \in M \Rightarrow S(x) \in M)$ akkor $M = \mathbb{N}$ (Indukció axióma)
5. $x + 0 = x$
6. $x + S(y) = S(x + y)$

Az $S(0) = 1$ jelölést használva, $x + 1 = x + S(0) = S(x + 0) = S(x)$
Az 5. és 5. axiómák egy rekurzív algoritmust adnak az 1-es számrendszerbeli összeadásra.
Az $a + b$ összeg kiszámításának (rekurzív) algoritmusa:
Vegyünk a darab kavicsot a bal kezünkbe, b darab kavicsot a jobb kezünkbe.
Ha a jobb kezünk üres, akkor az eredmény a bal kezünkben van (5. axióma).
Egyébként, ($b = S(\overline{b}) = \overline{b} + 1$ valamely $\overline{b}$-ra)

1 tegyünk félre 1 kavicsot a jobb kezünkből

2 adjuk össze (ezen algoritmussal) a két kezünkben lévő kavicsokat

3 tegyük a bal kezünkbe az 1 félretett kavicsot

A szorzás rekurzív megadása

$$x \cdot 0 = 0$$

$$x \cdot S(y) = x + x \cdot y$$

3.1. Eldöntendő, kinek van több birkája

Két szomszédos gazda vitatkozik, hogy kinek van több birkája. Adjunk algoritmust a vita eldöntésére!

A < (lineáris) rendezési reláció rekurzív megadása

$$0 < S(x)$$

$$\neg(x < 0)$$

$$S(x) < S(x) \Leftrightarrow x < y$$

4. Feladat: Partíciószám

Definíció. Az n természetes szám egy partíciója olyan $\pi = \langle a_1, \cdots, a_k \rangle$ sorozat, amelyre:

- $a_1 \geq a_2 \geq \cdots \geq a_k > 0$
- $\sum_{i=1}^{k} a_i = n$

(a_i a π *partíció része*.) Jelölje $P(n)$ n összes partíciójának számát.

Probléma: Partíciószám

Bemenet: n

Kimenet: n partícióinak száma, $P(n)$

Megoldás

Jelölje $P2(n,k)$ n azon partícióinak számát, amelyben minden rész $\leq k$.

Összefüggések:

1. $P2(1,k) = 1, P2(n,1) = 1$
2. $P2(n,n) = 1 + P2(n,n-1)$
3. $P2(n,k) = P2(n,n)$ ha $n < k$
4. $P2(n,k) = P2(n,k-1) + P2(n-k,k)$ ha $k < n$

A megoldás: $P(n) = P2(n,n)$

```
1  long long P2(int n, int k){
2     if (n==1 || k==1)
3        return 1;
4     else if (k>=n)
5        return 1+P2(n,n-1);
6     else
7        return P2(n,k-1) + P2(n-k, k);
8  }
9  long long P(int n){
10    return P2(n, n);
11 }
12 int main(){
13    long long h=P(22);
14    printf("%lld \n", h);
15 }
```

Rekurzív algoritmus helyességének bizonyítása

I. **Terminálás bizonyítása**

Bebizonyítandó, hogy minden eljáráshívás végrehajtása véges lépésben befejeződik. (Az eljárás terminál.)

Terminálás bizonyítása megállási feltétellel.

Megállási feltétel

Legyen $P(x_1, \ldots, x_n)$ n-paraméteres rekurzív eljárás.

A $M(x_1, \ldots, x_n)$ kifejezés megállási feltétele a P rekurzív eljárásnak, ha

1. $M(a_1, \ldots, a_n) \geq 0$ minden megengedett $a_1, \ldots, a_n$ aktuális paraméterre.
2. Ha $M(a_1, \ldots, a_n) = 0$ akkor nincs rekurzív hívás $P(a_1, \ldots, a_n)$ végrehajtásakor.
3. Ha van rekurzív hívás $P(a_1, \ldots, a_n)$ végrehajtásakor valamely $b_1, \ldots, b_n$ paraméterekre, akkor $M(b_1, \ldots, b_n) < M(a_1, \ldots, a_n)$

Állítás: $M(n,k) = (n-1) \times (k-1)$ megállási feltétel P2-re.

II. **Helyesség bizonyítása**

1. *Alaplépés.* Annak bizonyítása, hogy az eljárás helyes eredményt számít ki, ha az aktuális paraméterek esetén nincs rekurzív hívás.
2. *Rekurzív lépés.* Feltéve, hogy minden rekurzív hívás helyes eredményt ad, annak bizonyítása, hogy a rekurzív hívások által adott értékekből az eljárás helyes eredményt számít ki.

5. Feladat: Hanoi tornyai

A hanoi torony probléma: Három pálca egyikén n korong van a többi üres. A korongok nagyság szerinti sorrendben helyezkednek el, alul van a legnagyobb. Át akarjuk helyezni a korongokat egy másik pálcára a következő szabályok alapján. Egyszerre csak egy korong mozgatható. A korong vagy üres pálcára vagy egy nála nagyobb korongra helyezhető. Oldjuk meg a feladatot egy rekurzív algoritmussal! Határozzuk meg a korongmozgatások számát!

Megoldás

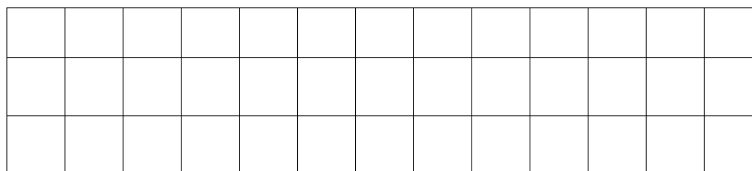
Legyen a hanoi eljárás egy olyan algoritmus, amelynek három argumentuma van, az első argumentum azt adja meg, hogy hány korongot helyezünk át, a második megadja, hogy melyik toronyról a harmadik, hogy melyik toronyra. Ekkor az eljárás az $(n,1,2)$ argumentummal megoldja a feladatot. Amennyiben $i-1$ korongot már át tudunk helyezni, i korongot a következőképpen helyezhetünk át. Elsőként $i-1$ korongot áthelyezünk az oszlopról egy másik oszlopra. Utána az i-edik korongot rárakjuk a kimaradó üres oszlopra. Végül ezen korong tetejére felrakjuk az $i-1$ korongot. Ezt a rekurziót írja le a következő eljárás (a megengedett lépést a mozgat függvény írja le, az argumentumai, hogy honnan hova)

```
1  void mozgat(int rol, int ra){
2     printf("%d -> %d\n ", rol, ra);
3  }
4  void hanoi(int n, int rol, int ra){
5     if (n==1)
6        mozgat(rol, ra);
7     else{
8        hanoi(n-1, rol, 6-(rol+ra));
9        mozgat(rol, ra);
10       hanoi(n-1, 6-(rol+ra), ra);
11    }
12 }
13 int main(){
14    hanoi(5, 1, 2);
15 }
```

Lépések száma: Az i-edik korong átrakásához kétszer kell $i-1$ korongot áthelyezni és egy további mozgatás szükséges. Tehát $T(i)$-vel jelölve az i korong átrakásához szükséges mozgatások számát a $T(i) = 2T(i-1)+1$ rekurzív összefüggés áll fenn. $T(1) = 1$, így $T(2) = 3$, $T(3) = 7$. Azt sejthetjük, hogy $T(i) = 2^i - 1$, amely egyenlőség teljes indukcióval egyszerűen igazolható.

6. Feladat: Járdakövezés

Számítsuk ki, hogy hányféleképpen lehet egy $3 \times n$ egység méretű járdát kikövezni 1×2 méretű lapokkal!
Megoldás

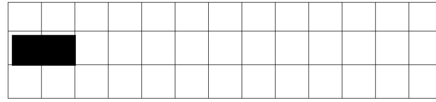


1. ábra.

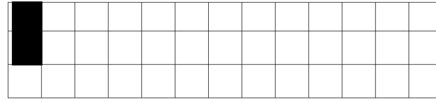
Jelölje $A(n)$ a megoldás értékét $3 \times n$ egység méretű járda esetén.
Az első oszlop középső négyzete háromféleképpen fedhető le.
Az egyes esetek csak az alábbi módon folytathatók:
Jelölje $B(n)$ azt, hogy hányféleképpen fedhető le egy $3 \times n$ egység méretű járda, amelynek a bal alsó sarka már le van fedve.
Szimmetria miatt a jobb felső sarok lefedettsége esetén is $B(n)$ féle lefedés van.

$$A(n) = \begin{cases} 0 & \text{ha} \quad n = 1 \\ 3 & \text{ha} \quad n = 2 \\ A(n-2) + 2B(n-1) & \text{ha} \quad n > 2 \end{cases} \tag{1}$$

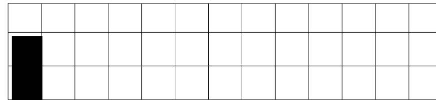
$$B(n) = \begin{cases} 1 & \text{ha} \quad n = 1 \\ 0 & \text{ha} \quad n = 2 \\ A(n-1) + B(n-2) & \text{ha} \quad n > 2 \end{cases} \tag{2}$$



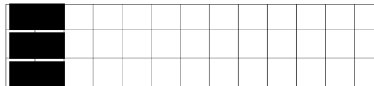
2. ábra. 1. eset



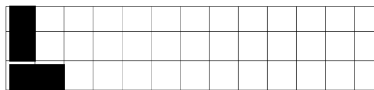
3. ábra. 2. eset



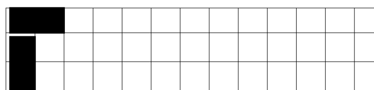
4. ábra. 3. eset



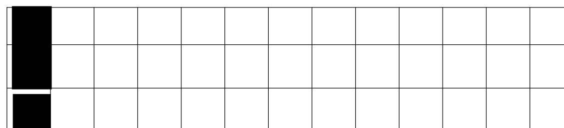
5. ábra. Az 1. eset csak így folytatható



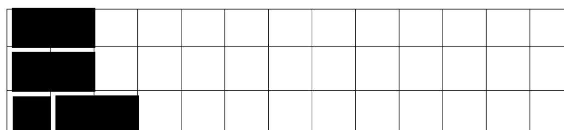
6. ábra. A 2. eset csak így folytatható



7. ábra. A 3. eset csak így folytatható



8. ábra. Az 1. eset csak így folytatható

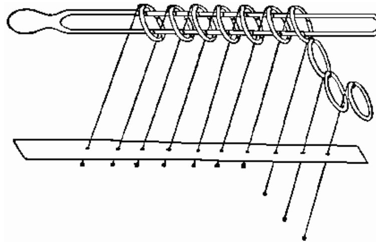


9. ábra. Az 2. eset csak így folytatható

```
program jarda;
long long B(int n);
long long A(int n){
   if (n==1)
      return 0;
   else if (n==2)
      return 3;
   else
      return A(n−2)+2∧B(n−1);
}//A

long long B(int n){
   if (n==1)
      return 1;
   else if (n==2)
      return 0;
   else
      return A(n−1)+B(n−2);
}//B
int main(){
  printf("%lld␣\n", A(32));
  return 0;
}
```

7. Feladat: Ördöglakat kinyitása



10. ábra. Ördöglakat

Az ördöglakat fém gyűrűkből összeállított szerkezet. Minden gyűrűnek van szára, amelyet körbefog a sorrendben következő gyűrű. Zárt állapotban a szárakat körbefogja egy fémből készült hurok. Az a cél, hogy a lakatot kinyissuk, azaz a hurkot eltávolítsuk.
A gyűrűket balról-jobbra 1-től n-ig sorszámozzuk. Minden lépésben egy gyűrű vehető le, vagy tehető fel az alábbi két szabály betartásával.
1. Az első gyűrű bármikor levehető, illetve felrakható.
2. Minden $i > 1$ sorszámú gyűrű akkor és csak akkor vehető le, illetve tehető fel, ha az $i-1$-edik gyűrű fent van, és minden $i-1$-nél kisebb sorszámú gyűrű lent van. A lakat akkor van kinyitva, ha minden gyűrű lent van.
Írjon olyan rekurzív eljárást, amely megadja lépések olyan sorozatát, amely kinyitja a lakatot!

Megoldás

Részproblémákra bontás:

MindLe(m) Leveszi az első m gyűrűt (tetszőleges sorrendben), a többit változatlanul hagyja.

Le(i) Leveszi az i. gyűrűt, minden $j > i$ sorszámút változatlanul hagy. (A $j < i$ sorszámúak helyzete akármilyen lehet a művelet után.)

Fel(i) Felteszi az i. gyűrűt, minden $j > i$ sorszámút változatlanul hagy. (A $j < i$ sorszámúak helyzete akármilyen lehet a művelet után.)

```
1  #define maxGyuru 10
2  bool Lakat[maxGyuru];
3  void Le(int i);
4  void Fel(int i);
5
6  void MindLe(int m){
7      for (int i=m; i>=1; i--)
8          if (Lakat[i]) Le(i);
9  }
10 void Le(int i){
11     if (i>1 && !Lakat[i-1])
12         Fel(i-1);
13     if (i>2)
14         MindLe(i-2);
15     Lakat[i]=false;
16     printf("%d.␣Le␣␣\n",i);
17 }
18 void Fel(int i){
19     if (i>1 && !Lakat[i-1])
20         Fel(i-1);
21     if (i>2)
22         MindLe(i-2);
23     Lakat[i]=true;
24     printf("%d.␣Fel␣␣\n",i);
25 }
26 int main(){
27     for (int i=1; i<=maxGyuru; i++)
28         Lakat[i]=true;
29     MindLe(5);
30 }
```

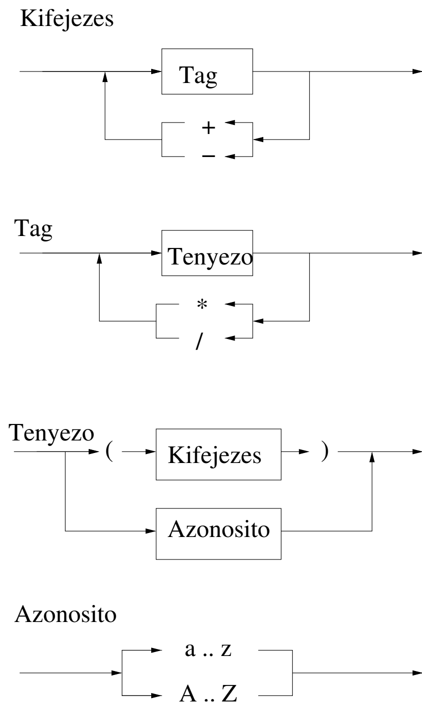
```
1  1. Le
2  3. Le
3  1. Fel
4  2. Le
5  1. Le
6  5. Le
7  1. Fel
8  2. Fel
9  1. Le
10 3. Fel
11 1. Fel
12 2. Le
13 1. Le
14 4. Le
15 1. Fel
16 2. Fel
17 1. Le
18 3. Le
19 1. Fel
20 2. Le
21 1. Le
```

8. Feladat: Postfix konverzió

Aritmetikai kifejezés szokásos írásmódja, hogy a műveleti jel az argumentumok között áll. Ezt az írásmódot infix jelölésnek is nevezzük. Mivel a multiplikatív műveletek (szorzás $*$ és osztás $/$) prioritása magasabb, mint az additív $(+,-)$ műveleteké, ezért

zárójelezni kell.
Pl. $(a+b)*(c-d)+a$.

Lukasiewic lengyel logikatudós vette először észre, hogy ha a műveleti jeleket az argumentumok után írjuk, akkor nincs szükség zárójelekre. Ezért ezt az írásmódot *fordított lengyel jelölésnek*, vagy *postfix* alaknak hívjuk.
Jelölje $\phi(K)$ a K kifejezés postfix alakját.
Pl. $\phi((a+b)*(c-d)+a) = ab+cd-*a+$
Probléma: Postfix konverzió
Bemenet: K aritmetikai kifejezés infix jelölésben.
Kimenet: K postfix alakja.
A szabályos aritmetikai kifejezések megadhatók a következő (rekurzív) szintaxis diagramokkal (Az egyszerűség kedvéért az elemi kifejezések csak egybetűs azonosítok lehetnek).
Tehát minden kifejezés vagy egy *tag*, vagy tagok additív műveleti jelekkel elválasztott sorozata. Minden kifejezés egyértelműen



11. ábra. Kifejezés szintaxis diagramjai

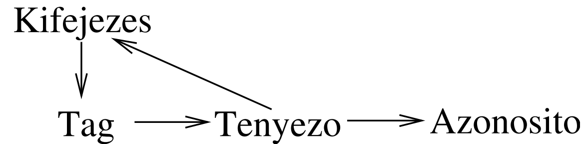
felbontható
$K = t_1 \oplus_1 t_2 \ldots \oplus_m t_{m+1}$ alakban, ahol $\oplus_i \in \{+,-\}$ és t_i tag. Ekkor (a balról-jobbra szabály szerint)
$\phi(K) = \phi(t_1)\phi(t_2) \oplus_1 \ldots \phi(t_{m+1}) \oplus_m$.

Minden tag vagy egy *tényező*, vagy tényezők multiplikatív műveleti jellel elválasztott sorozata. Minden tag egyértelműen felbontható
$T = t_1 \otimes_1 t_2 \ldots \otimes_m t_{m+1}$ alakban, ahol $\otimes_i \in \{*,/\}$ és t_i tényező. Ekkor (a balról-jobbra szabály szerint)
$\phi(T) = \phi(t_1)\phi(t_2) \otimes_1 \ldots \phi(t_{m+1}) \otimes_m$.

Minden tényező vagy zárójelbe tett kifejezés; (K) és $\phi((K)) = \phi(K)$, vagy azonosító; a és $\phi(a) = a$.
Tehát a konverzió algoritmusát megalkothatjuk úgy, hogy minden szintaktikus egységhez (Kifejezés, Tag, Tényező, Azonosító) egy eljárást adunk, amely balról jobbra haladva az aktuális karaktertől elolvassa és konvertálja a neki megfelelő (legszűkebb) karaktersorozatot.

Az eljárásrendszer hívási gráfja.

```
1  #include <string>
2  #include <iostream>
3  using namespace std;
4
5  bool Jo = true;
6  string S;
```



12. ábra. Az eljárások hívási gráfja

```
 7  int i = 0;
 8  char Jel;
 9  string PostForm;
10
11  void KovJel();
12  void Kifejezes();
13  void Tag();
14  void Tenyezo();
15  string Postfix(string K);
16
17  void KovJel(){
18    Jel = S[i++];
19  }
20  void Kifejezes(){
21  //Global:Jel, Jo, PostForm
22    char M;
23    Tag();
24    while (Jo && Jel == '+' || Jel == '−'){
25      M = Jel;
26      KovJel();
27      Tag();
28      PostForm += M;
29    }
30  }
31
32  void Tag(){
33  //Global:Jel, Jo, PostForm
34    char M;
35    Tenyezo();
36    while (Jo && Jel == '∧' || Jel == '/'){
37      M = Jel;
38      KovJel();
39      Tenyezo();
40      PostForm += M;
41    }
42  }
43  void Tenyezo(){
44  //Global:Jel, Jo, PostForm
45    if ('a' <= Jel && Jel <= 'z') {
46      PostForm += Jel;
47      KovJel();
48    }else if (Jel == '('){
49      KovJel();
50      Kifejezes();
51      if (Jo && Jel == ')')
52        KovJel();
53      else
```

```
54          Jo = false;
55      }else
56         Jo = false;
57  }

58  string Postfix(string K){
59  //Global: S, PostForm
60     S = K + '.';
61    KovJel();
62    PostForm = "";
63    Kifejezes();
64    return PostForm;
65  }
66
67  int main()
68  {
69    cout << Postfix("a+b^(a-b)/(x+y)") << endl;
70    return 0;
71  }
```

9. Fák, fák ábrázolása

Az adatkezelés szintjei:

1. Probléma szintje.
2. Modell szintje.
3. Absztrakt adattípus szintje.
4. Absztrakt adatszerkezet szintje.
5. Adatszerkezet szintje.
6. Gépi szint.

Absztrakt adattípus: $A = (E, M)$

1. E: értékhalmaz,
2. M: műveletek halmaza.

"Absztrakt" jelző jelentése:

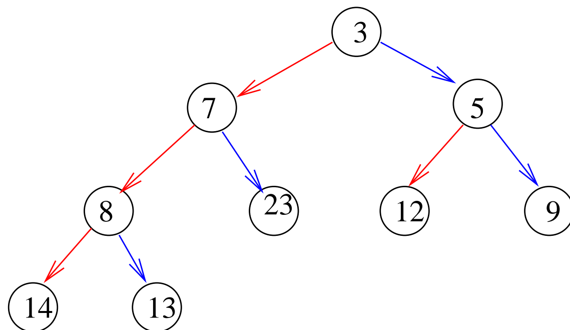
i. Nem ismert az adatokat tároló adatszerkezet.

ii. Nem ismertek a műveleteket megvalósító algoritmusok, a műveletek specifikációjukkal definiáltak.

Absztrakt adatszerkezetek

Olyan $\mathtt{A} = (M, R, Adat)$ rendezett hármas, ahol
1. M az absztrakt memóriahelyek, *cellák* halmaza.
2. $R = \{r_1, \ldots, r_k\}$ a cellák közötti *szerkezeti kapcsolatok*, $r_i : M \to (M \cup \{\perp\})^*$
3. $Adat : M \to E$ parciális függvény, a cellák adattartalma.
$x \in M$, $r \in R$ és $r(x) = \langle y_1, \ldots, y_i, \ldots, y_k \rangle$ esetén az x cella r kapcsolat szerinti szomszédai $\{y_1, \ldots, y_k\}$, y_i pedig az x cella i-edik szomszédja.
Ha $y_i = \perp$, akkor azt mondjuk, hogy x-nek hiányzik az r szerinti i-edik szomszédja.
Példa absztrakt adatszerkezetre: (Minimumos)Kupac. $\mathtt{A} = (M, R, Adat)$
1. M az absztrakt memóriahelyek, *cellák* halmaza.
2. $R = \{bal, jobb\}$ a zerkezeti kapcsolatok.

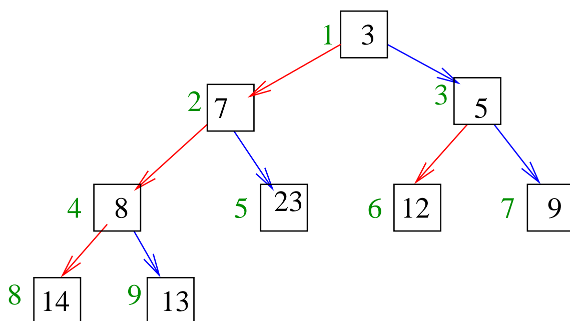
- A $\{bal, jobb\}$ kapcsolatok bináris fát határoznak meg.
- $(\forall x \in M)(Adat(x) \leq Adat(bal(x)) \wedge Adat(x) \leq Adat(jobb(x)))$



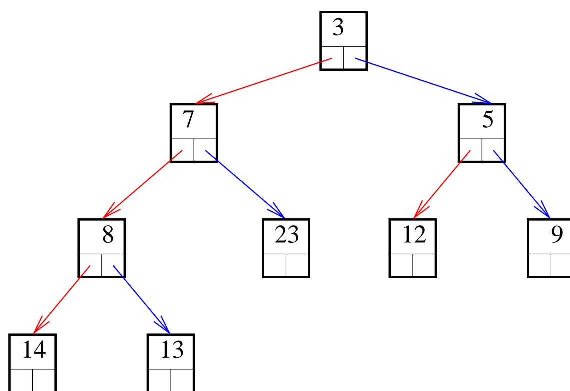
13. ábra. Kupac absztrakt adatszerkezet szemléltetése

Adatszerkezetek

Egy $\mathtt{A} = (M, R, Adat)$ absztrakt adatszerkezet megvalósítása:
1. Konkrét memória allokálás az M-beli absztrakt memória cellák számára.
2. Az R szerkezeti kapcsolatok ábrázolása.
3. Alapműveletek algoritmusainak megadása.
Pl. a kupac megvalósítása konkrét adatszerkezettel:
1. A memóriahelyek tömbelemek. 2. A $\{bal, jobb\}$ kapcsolatokat számítási eljárással adjuk meg:
$bal(x) = 2 * x$, $jobb(x) = 2 * x + 1$.
Teljesül a kupac tulajdonság:
$(\forall x)(Adat(x) \leq Adat(bal(x)) \wedge Adat(x) \leq Adat(jobb(x)))$



14. ábra. Kupac ábrázolása tömbbel. A szerkezeti kapcsolatot nem tároljuk; számítjuk.



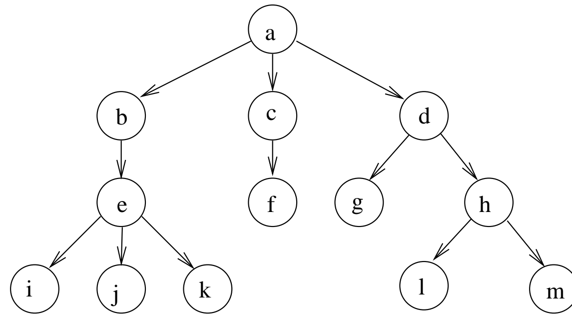
15. ábra. Kupac ábrázolás dinamikus memóriacellákkal; a szerkezeti kapcsolatot táruljuk.

Fák algebrai definíciója

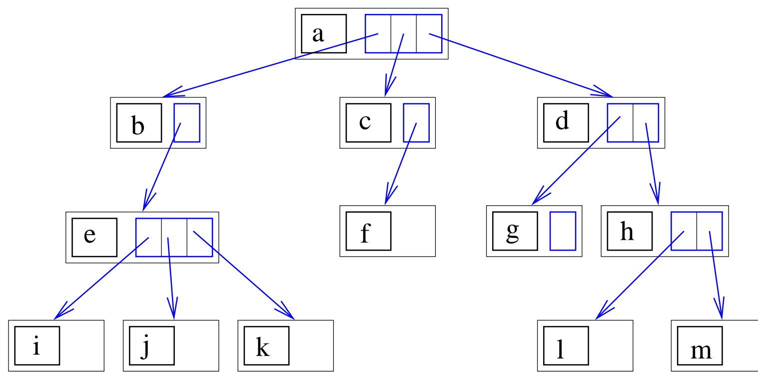
Az A adathalmaz feletti fák halmaza, $Fa(A)$ az a legszűkebb halmaz, amelyre teljesül:

- $a \in A$ akkor $a \in Fa(A)$
- ha $a \in A$ és $f_i \in Fa(A), i = 1, \ldots, k$ akkor $a(f_1, \ldots, f_k) \in Fa(A)$

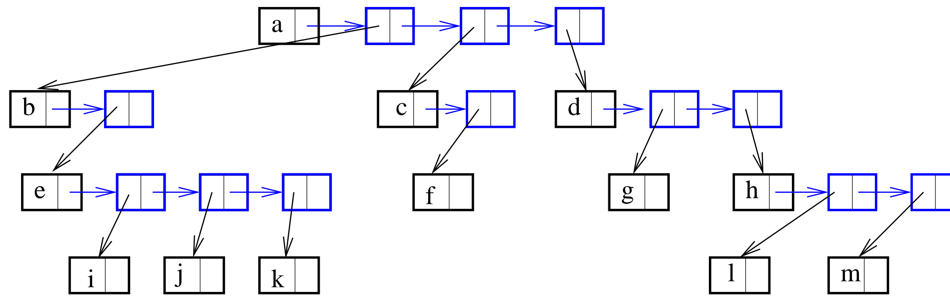
Fák ábrázolása



16. ábra. Példa fa, amelynek algebrai megadása: $a(b(e(i, j, k)), c(f), d(g, h(l, m)))$.



17. ábra. A példa fa ábrázolása kapcsolati tömbbel.



18. ábra. A példa fa ábrázolása kapcsolati lánccal.

9.1. Kapcsolati tömb ábrázolás

Objektumos

```
template <class E>
class FaPontT{
  public:
  E adat;
  int nfiuk;
  FaPontT<E> ΛΛfiuk;
};
```

Pointeres

```
typedef struct FaPontT{
          E adat;
          int nfiuk;
          FaPontT ΛΛfiuk;
//        FaPontT Λapa;
}FaPontT;
```

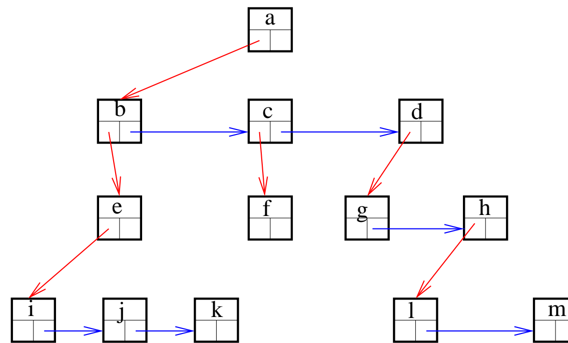
9.2. Kapcsolati lánc ábrázolás

Objektumos

```
template <class E>
class FaPontL{
  public:
  E adat;
  Lanc<FaPontL Λ> Λfiuk;
};
```

Pointeres

```
struct Flanc;
typedef struct FaPontL{
          char adat;
          Flanc Λfiuk;
}FaPontL;
typedef struct Flanc{
          FaPontL Λfiu;
          Flanc Λcsat;
}Flanc;
```



19. ábra. A példa fa ábrázolása elsőfiú-testvér kapcsolattal.

9.3. Fa elsőfiú-testvér ábrázolása

Objektumos

```
template <class E>
class FaPont{
  public:
  E adat;
  FaPont<E> Λelsofiu;
  FaPont<E> Λtestver;
//  FaPont<E> Λapa;
};
```

Pointeres

```
typedef struct FaPont{
         E adat;
         FaPont Λefiu, Λtestver;
//       FaPont Λapa;
}FaPont;
```

9.4. Bináris fa

Bináris fa, objektumos

```
template <class E>
class BinFaPont{
  public:
  E adat;
  BinFaPont<E> Λbal;
  BinFaPont<E> Λjobb;
//  BinFaPont<E> Λapa;
};
```

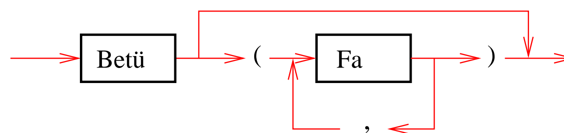
Bináris fa, pointeres

```
typedef struct BinFaPont{
         E adat;
         BinFaPont Λbal, Λjobb;
//       BinFaPont Λapa;
}BinFaPont;
```

10. Feladat: Fa adatszerkezet előállítás

Adjunk olyan algoritmust, amely egy fa algebrai ábrázolásához előállítja a fa elsőfiu-testvér ábrázolását!
Bemenet:
A fa algebrai leírását tartalmazó s string.
Kimenet:
A fa gyökerére mutató pointer.
B. részfeladat
Adjunk olyan algoritmust, amely előállítja egy elsőfiu-testvér ábrázolású fa algebrai leírását!
Bemenet:
A fa gyökerére mutató pointer.
Kimenet:
A fa algebrai leírását tartalmazó s string.

Megoldás



20. ábra. Algebrai fa szintaxis-diagramja

Készítsünk olyan `FaEpit()` rekurzív eljárást,amelyre globális a leírást tartalmazó s string és az aktuális pozíciót kijelölő i változó.
`FaEpit()` specifikációja:
Bemenet: Az aktuális i pozíciótól egy szabályos fa-leíráskezdődik.
Kimenet: Az eljárás elolvassa az aktuális pozíciótólkezdődő fa-leírást, és előállítja a fa elsőfiú-testvér ábrázolását, a fa gyökerére mutató pointert ad vissza.

```
#include <iostream>
#include <string>
#define maxN 100001
```

```
4  #define maxM 10001
5
6  using namespace std;
7  typedef struct FaPont{
8          char adat;
9          FaPont *efiu, *testver;
10 }FaPont;

11 string F;
12 int i;
13 FaPont* FaEpit(){
14     FaPont* p = new FaPont;
15     FaPont* elso=NULL, *utolso, *fiu;
16     p->adat=F[i++];
17     p->efiu=NULL; p->testver=NULL;
18     if (F[i]=='('){
19         i++;
20         while (F[i]!=')'){
21             fiu = FaEpit();
22             if (F[i]==',') i++;
23             if (elso==NULL){
24                 elso=fiu; utolso=fiu;
25             }else{
26                 utolso->testver=fiu;
27                 utolso=fiu;
28             }
29         }//while
30         i++; //")" átlépése
31     }
32     p->efiu=elso;
33     return p;
34 }
```

```
1  void Preorder(FaPont* p){
2      if (p==NULL) return;
3      cout<<p->adat;
4      FaPont* f=p->efiu;
5      if (f!=NULL){
6          cout<<'(';
7          while (f!=NULL){
8              Preorder(f);
9              f=f->testver;
10             if (f!=NULL) cout<<',';
11         }
12         cout<<')';
13     }
14 }
```

Bináris fák bejárásai: preorder, inorder, postorder

```
1  void Inorder(BinFaPont* p){
2      if (p->bal!=NULL) Inorder(p->bal);
3  //művelet elvégzése a pontban lévő adaton
4      cout<<p->adat<<',';
5      if (p->jobb!=NULL) Inorder(p->jobb);
6  }
```

```
void Preorder(BinFaPont∧ p){
//művelet elvégzése a pontban lévő adaton
    cout<<p->adat<<',';
    if (p->bal!=NULL) Preorder(p->bal);
    if (p->jobb!=NULL) Preorder(p->jobb);
}
```

```
void Postorder(BinFaPont∧ p){
    if (p->bal!=NULL) Postorder(p->bal);
    if (p->jobb!=NULL) Postorder(p->jobb);
//művelet elvégzése a pontban lévő adaton
    cout<<p->adat<<',';
}
```

10.1. feladat

Írjunk olyan algoritmust, amely kiszámítja egy fa magasságát!

Megoldás

A fa magasságának definíciója algebrai jelölést használva:

$$h(f) = \begin{cases} 0, & \text{ha } f = \perp \\ 1, & \text{ha } f = a \\ 1 + \max(h(f_1), \ldots, h(f_k)), & \text{ha } f = a(f_1, \ldots, f_k) \end{cases}$$

```
int magassag(FaPont∧ f){
    if (f==NULL) return 0;
    int m=0; int fium;
    f=f->efiu;
    while (f!=NULL){
        fium=magassag(f);
        if (fium>m) m=fium;
        f=f->testver;
    }
    return m+1;
}
```

10.2. feladat

Írjunk olyan algoritmust, amely kiszámítja, hogy hány pontja van a fának adott k szinten!

Megoldás

Jelölje $S(f,k)$ az f fa k-adik szintjén lévő pontjainak számát. Ekkor

$$S(f,k) = \begin{cases} 0, & \text{ha } f = \perp \\ 1, & \text{ha } k = 1 \\ S(f_1, k-1) + \cdots + S(f_m, k-1), & \text{ha } f = a(f_1, \ldots, f_m) \end{cases}$$

```
int szinten(FaPont∧ f, int k){
    if (f==NULL || k==0) return 0;
    if (k==1) return 1;
    int m=0;
    f=f->efiu;
    while (f!=NULL){
        m+=szinten(f, k-1);
        f=f->testver;
    }
    return m;
}
```

11. Feladat: Kaminon

Egy vállalat az ország különböző városaiban levő üzemeiben alkatrészeket termel. A heti termelést a hét végén kamionokkal szállítja a központi raktárába. A kamionforgalom korlátozása miatt minden városból pontosan egy másik városba (egy irányban) mehetnek a kamionok közvetlenül. Ezért a vállalat úgy tervezi a szállításokat, hogy minden olyan városból, amelybe más városból nem lehet eljutni, egy-egy kamiont indít, a többi városból viszont egyet sem. A korlátozások miatt így minden kamion útja a központi raktárig egyértelműen meghatározott.

Minden kamion, amely útja során áthalad egy városon, az ott termelt alkatrészekből bármennyit felvehet, feltéve, hogy nincs tele. Ismerve a városokban termelt alkatrészek számát, ki kell számítani azt a legkisebb kamion kapacitást, amellyel a szállítás megoldható, ha minden kamion azonos kapacitású.

Bemenet

A `kamion.be` szöveges állomány első sorában a városok N $(1 < N \leq 10000)$ száma van. A központi raktár az 1. városban van, és onnan nem kell szállítani. Az állomány következő $N-1$ sorának mindegyike két egész számot tartalmaz, egy szóközzel elválasztva. Az állomány I-edik sorában az első szám azt a várost adja meg, ahova az I-edik városból mehet kamion. A második szám pedig az I-edik városban termelt alkatrészek száma. (Az 1. városból kivezető út nincs megadva.)

Kimenet

A `kamion.ki` szöveges állományba első sorába azt a legkisebb kamion kapacitást (egész szám) kell kiírni, amekkora kapacitású kamionokkal az összes alkatrész elszállítható. **Megoldás**

A feltételekböl következik, hogy az úthálózat alkotta gráf fa, aminek gyökere az 1. varos, ahova szállítani kell. Definiáljuk minden v városra az alábbi értékeket:

- $K(v)$: a v városon áthaladó kamionok száma,
- $E(v)$: a v város gyökerű fában lévő városokban termelt mennyiségek összege,
- $M(v)$: a minimális kamion kapacitas, amivel a v gyükerű fában lévő városokból a v-be történ szállítás elvégezhető.

A feladat megoldasa nyilvanvaloan $M(1)$.

Ha v-be nem vezet út (v levél a fában), akkor $E(v) = Db(v)$, a v-ben termelt mennyiség, továbbá $K(v) = 1$, $M(v) = Db(v)$. Ha v-be a $v1,\ldots,vt$ városokból megy út közvetlenül, akkor

- $K(v) = K(v1) + \ldots + K(vt)$,
- $E(v) = Db(v) + E(v1) + \ldots + E(vt)$,
- $M(v) = Max\{M(v1),...,M(vt)\}$, ha $Max\{M(v1),...,M(vt)\} * K(v) <= E(v)$, és $E(v)/K(v)$ felső egészrésze egyébként.

A fenti osszefuggesek alapjan E, K es M rekurzióval számítható. Ezt csinálja a Szamol eljárás.

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #define maxN 100001
4
5      typedef struct Lanc{
6          int fiu;
7          Lanc *csat;
8      }Lanc;
9      Lanc* Fa[maxN];
10     int Db[maxN];
11     int M[maxN];   int E[maxN];   int K[maxN];

12 void Szamol(int v){
13 //K(v), E(v), M(v) értékét számolja rekurzívan
14     Lanc* p; int f;
15     if (Fa[v]==NULL){ // v levél
16         K[v]=1;
```

```
17          E[v]=Db[v];
18          M[v]=1;
19      }else {
20          K[v]=0; E[v]=Db[v]; M[v]=0;
21          p=Fa[v];
22          while (p!=NULL){
23              f=p->fiu;
24              Szamol(f);
25              K[v]+=K[f];
26              E[v]+=E[f];
27              if (M[f] > M[v]) M[v]=M[f];
28              p=p->csat;
29          }
30          if (M[v]ΛK[v] < E[v]){
31              M[v]=div(E[v]+K[v]-1, K[v]).quot;
32          }
33      }
34  }

35  int main() {
36      LancΛ p;
37      int n, apa;
38      FILEΛ bef=fopen("kamion.be","r");
39      FILEΛ kif=fopen("kamion.ki","w");
40      fscanf(bef, "%d",&n);
41      for (int i=0; i<=n; i++)
42          Fa[i]=NULL;
43      for (int i=1; i<=n; i++){
44          fscanf(bef, "%d",&apa);
45          p = new Lanc;
46          p->fiu=i;
47          p->csat=Fa[apa]; Fa[apa]=p;
48      }
49      for (int i=1; i<=n; i++)
50          fscanf(bef, "%d",&Db[i]);
51      Db[1]=0;
52      fclose(bef);
53      Szamol(1);
54      fprintf(kif, "%d\n",M[1]);
55      fclose(kif);
56      return 0;
57  }
```