Algoritmusok és adatszerkezetek II.

Gyakorlati jegyzet

Bene Fruzsina

Tartalomjegyzék

1. fejezet

Adattömörítés

Informatikában a **kódoláselmélet** adatok különböző reprezentációjával és azok közötti átalakításokkal foglalkozik. Ennek egyik ága, a **forráskódolás** az adott alak hosszát vizsgálja; vagyis azt a kérdést, hogy az adott mennyiségű információt mekkora mennyiségű adattal lehet tárolni. Legtöbb esetben a cél a rövidebb reprezentáció, tehát beszélhetünk **információ-** vagy **adattömörítésről**.

A kódolás során fontos kérdés, hogy az adat teljes egészében visszaállítható-e. Tömörítés esetében ennek megfelelően használhatunk **veszteségmentes** vagy **veszteséggel járó** eljárásokat (pl. JPEG, MPEG, MP3, ...). Mi csak az előbbivel foglalkozunk.

A kódoláselméletnél meg kell adnunk az **információ alapegységét**, azaz azt, mennyi információtartalma van az atomi tárolási egységnek. Mivel a jelenlegi számítógépek bináris elven működnek, ez $r = 2$ és így a kódszavaink a $T = '0', '1'$ ábécé feletti szavak lesznek.

Kódnak nevezzük a T feletti véges szavak (kódszavak) egy tetszőleges nem üres halmazát.

Egy kód szemléletesebb ábrázolásához elkészíthetjük annak **kódfáját**. Ebben a fában a fa csúcsai szavak (nem feltétlenül kódszavak), az éleit pedig a kódszavak lehetséges karaktereivel címkézzük. A fa gyökerében az üres szó szerepel és egy szóhoz tartozó csúcs leszármazottai azok a szavak, amelyeket úgy kapunk, hogy a szó után írjuk az élen szereplő karaktert. A kódhoz tartozó kódfa az a legkevesebb csúcsot tartalmazó ilyen tulajdonságú fa, ami tartalmazza az összes kódszót.

A kódfa szemléltetés mellett más szempontból is hasznos lehet. Egyrészt a fa tulajdonságaiból következtethetünk a kód tulajdonságaira, másrészt a kódfa segítségével egy bitsorozat hatékonyan dekódolható: A gyökérből indulva a bitek szekvenciájának megfelelően járjuk be a fát, az élek mentén kódszót keresve és

találat esetén ismételve a bejárást megkapjuk a dekódolt adatot. (Ha a dekódolás lehetséges és egyértelmű.)

A kódolást **betűnkénti kódolás**nak nevezzük, ha az **eredeti** Σ **ábécé** feletti adatot betűnként egy $\Sigma \longrightarrow C \subset T*$ kölcsönösen egyértelmű (bijektív) leképezéssel készítjük el. Például az ASCII kódolás is ilyen, hiszen a megfelelő táblázat alapján betűnként történik a kódolt adat kiszámolása.

A kódolandó állományokat karaktersorozatnak tekintjük, és a tömörítés célja az adatok kisebb helyen történő tárolásának biztosítása.

1.1. Naiv módszer

A tömörítendő szöveget karakterenként, fix hosszúságú bitsorozatokkal kódoljuk. A kódot **egyenletes kód**nak nevezzük, ha a kód szavainak hossza egyenlő. A **naiv módszer** egyenletes kódot használó betűnkénti kódolás.

$$\Sigma = <\sigma_1, \sigma_2, ..., \sigma_d>$$

A Σ ábécé feletti kódolt adat akkor lesz a legkisebb, ha a kódszavak közös hossza a legkisebb. Mivel $\mid T \mid = r$ és $\mid \Sigma \mid = d$ ez azt jelenti, hogy az egyes karakterek legkevesebb $\lceil \log_r d \rceil$ hosszal kódolhatóak naiv módszer segítségével.

Ez alapján egy-egy karakter $\lceil \log_2 d \rceil$ bittel kódolható, ugyanis $\lceil \log_2 d \rceil$ biten $2^{\lceil \log_2 d \rceil}$ különböző bináris kód ábrázolható, és

$$2^{\lceil \log_2 d \rceil} \geq d > 2^{\lceil \log_2 d \rceil - 1},$$

azaz $\lceil \log_2 d \rceil$ biten ábrázolható d-féle különböző kód, de eggyel kevesebb biten már nem.

(A későbbiekben $\log_2 d = \log d$, az alapértelmezett alap a 2-es lesz, ugyanis $r = 2$ a $T = '0', '1'$ ábécé miatt.)

$In : \Sigma <>$ a tömörítendő szöveg. $n = \mid In \mid$ jelöléssel $n * \lceil \log d \rceil$ bittel kódolható. (n a tömörítendő szöveg hossza, d az ábécé betűinek számossága.)

Példa

ABRAKADABRA szöveg

$d = 5$ és $n = 11 \longrightarrow$ a tömörített kód hossza $11 * \lceil \log 5 \rceil = 11 * 3 = 33$ bit (A 3 bites kódok közül tetszőleges 5 kiosztható az 5 betűnek.) A tömörített fájl tartalmazza

a kódtáblázatot is.

Az ABRAKADABRA szöveg kódtáblázata lehet a következő:

karakter	kód
A	000
B	001
D	010
K	011
R	100

A fenti kódtáblázattal a tömörített kód a következő lesz:
00000110000001100001000000011000000.

Ez a tömörített fájlba foglalt kódtáblázat alapján könnyedén 3 bites szakaszokra bontható és kitömöríthető. Gyakorlatban, ha a tömörítés nem igazán fontos szempont, egyszerűsége miatt sok helyen alkalmazzák, például a 8 bit hosszúságú kódszavakat használó ASCII kód is ilyen. Csak hosszabb szövegeket érdemes így tömöríteni a kódtáblázat mérete miatt.

1.2. Huffman-kód

A Huffman-kód nagyon hatékony módszer az adatállományok tömörítésére. A megtakarítás 20%-tól 90%-ig terjedhet, a tömörítendő adatállomány sajátosságai alapján. A mohó algoritmus egy táblázatot használ az egyes karakterek előfordulási gyakoriságára, hogy meghatározza, hogyan lehet a karaktereket optimálisan ábrázolni bináris jelsorozattal.

Betűnkénti kódolás esetén akkor kapunk rövidebb kódolt adatot, ha a gyakori betűkhöz rövid kódszót, a ritkákhoz pedig hosszabbakat rendelünk. A Huffman-kódolás egy **betűnkénti optimális kódolás**, az ilyen kódolások között szinte a legjobb tömörítés érhető el vele adott adat esetén. Ezt úgy érjük el, hogy a kódhoz tartozó kódfát alulról felfelé építjük az eredeti szöveg karaktereinek gyakorisága alapján.

Bináris esetben a lépések a következők:

1. Olvassuk végig a szöveget és határozzuk meg az egyes karakterekhez tartozó gyakoriságokat.

2. Hozzunk létre minden karakterhez egy csúcsot és helyezzük el azokat egy (min) prioritásos sorban a gyakoriság, mint kulcs segítségével.

3. Következő lépésként:
   (a) Vegyünk ki két csúcsot a prioritásos sorból és hozzunk létre számukra egy szülő csúcsot.
   (b) A szülő-gyerek éleket címkézzük nullával és eggyel a gyakoriságnak megfelelő sorrendben.
   (c) Helyezzük el a szülő csúcsot a prioritásos sorba gyerekei gyakoriságának összegét használva kulcsként.
4. Ismételjük meg az előző pontot, ha több mint egy csúcs szerepel a sorban.
5. Olvassuk ki a karakterekhez tartozó kódszavakat a kódfából.
6. Olvassuk végig újra a bemenetet és kódoljuk azt karakterenként.

A kitömörítést is karakterenként végezzük.

1. Mindegyik karakter kinyeréséhez a kódfa gyökerétől indulunk.
2. Majd a tömörített kód sorban olvasott bitjei szerint 0 esetén balra, 1 esetén jobbra lépünk lefelé a fában, mígnem levélcsúcshoz érünk.
3. Ekkor kiírjuk a levelet címkéző karaktert, majd a Huffman-kódban a következő bittől és újra a kódfa gyökerétől folytatjuk, amíg a tömörített kódon végig nem érünk.

A Huffmann-kódolás tulajdonságai:

- A tömörítendő fájlt, illetve szöveget kétszer olvassa végig.
- A kódfa szigorúan bináris fa.
- A tömörített fájl a kódfát is tartalmazza.

A Huffmann-kód mindig **egyértelműen dekódolható** a kódfa segítségével, mivel prefix-kód. **Prefix-kód** esetén a kódszavak halmaza prefix mentes, a kódszavakra igaz, hogy egyik sem kezdőszelete (valódi prefixe) bármelyik másiknak. A kódolás minden bináris karakterkódra egyszerű: csak egymás után kell írni az egyes karakterek bináris kódját.

Általában a Huffman-kódolás nem egyértelmű. Egyrészt, ha több azonos gyakoriság van, akkor bármelyiket választva Huffman-kódolást kapunk; másrészt a 0

és 1 szerepe felcserélhető.

Mivel a kódolás miden adathoz különböző, a dekódoló oldalon is ismertnek kell lennie. Ez gyakorlatban azt jelenti, hogy a kódfát vagy kódtáblát is csatolnunk kell a kódolt adathoz (ront a tömörítési arányon), vagy a Huffman-kódot általánosított adathoz készítjük el. Emiatt a gyakorlatban Huffmann-kódolással is csak hosszabb szövegeket érdemes tömöríteni.

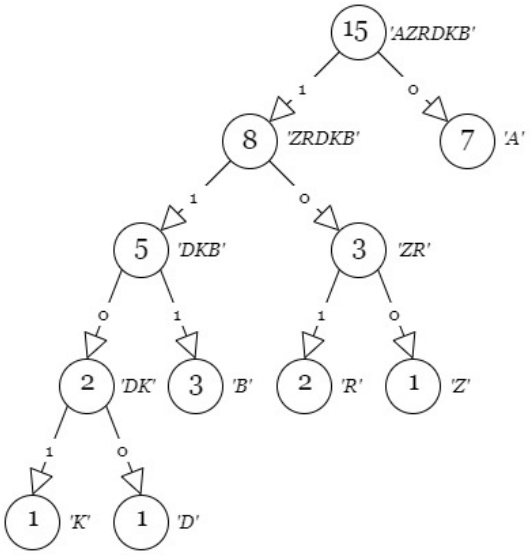
Példa

AZABBRAKADABRAA szöveg

karakter	gyakoriság	kód
A	7	0
B	3	111
D	1	1100
K	1	1101
R	2	101
Z	1	100

Prioritásos minimumsorba hozzáadjuk a címkéket, hogy ennek segítségével felrajzolhassuk a kódfát.

< 1 1 1 2 3 7 >
D K Z R B A
< 1 2 2 3 7 >
Z DK R B A
< 2 3 3 7 >
DK B ZR A
< 3 5 7 >
ZR DKB A
< 7 8 >
A ZRDKB
< 15 >
AZRDKB



Az AZABBRAKADABRAA kódolt alakja: 0100011111110101101011000111101000.

1.3. Lempel – Ziv – Welch (LZW) módszer

A betűnkénti kódolás hatékonysága korlátozott, könnyen találhatunk olyan adatot, amit sokkal tömörebb formában lehet reprezentálni, ha a kódolás nem karakterenként történik. Ezt használják ki a **szótárkódok** úgy, hogy egy kódszó nem csak egy karakter képe lehet, hanem egy szóé is.

Az LZW kódolás egy kezdeti kódtáblát bővít lépésről-lépésre úgy, hogy egyre hosszabb már „látott" szavakhoz rendel új kódszót. Az algoritmus a szöveg bizonyos szavaiból szótárat épít. Ez szavak egy halmaza, amit S-sel jelölünk. A szótárról három dolgot tételezünk fel:

- az egybetűs szavak mind szerepelnek benne;
- ha egy szó benne van a szótárban, akkor annak minden kezdődarabja is benne van;
- a tárolt szavaknak fix hosszúságú kódjuk van; $x \in S$ szó kódját $c(x)$ jelöli.

A gyakorlatban a kódok hosszát 12-15 bitnek érdemes választani. A tömörítendő szöveget S-beli szavak egymásutánjára bontjuk. A kódolás eredménye, a tömörített

szöveg az így kapott szavak kódjainak a sorozata. Az eredeti szöveg olvasásakor egyidőben épül az S szótár és alakul ki a felbontás. A tömörítés abból adódik, hogy sokszor helyettesítünk hosszú szavakat a rövid kódjaikkal.

A szótár egyik szokásos tárolási módja a **szófa adatszerkezet**. Az $x \in S$ szó $c(x)$ kódja úgy is tekinthető, mint egy hivatkozás az x-nek az S-beli előfordulására. A szöveg összenyomása úgy történik, hogy amikor az olvasás során egy $x \in S$ szót találunk, aminek a következő Y betűvel való folytatása már nincs S-ben, akkor $c(x)$-et kiírjuk a kódolt szövegbe. Az xY szót felvesszük az S szótárba. A szó $c(xY)$ kódja a legkisebb még eddig az S-ben nem szereplő kódérték lesz. Ezután az Y betűvel kezdődően folytatjuk a bemeneti szöveg olvasását. Az algoritmus kicsit pontosabb megfogalmazásához legyen z egy szó típusú változó, K egy betű típusú változó. A z változó értéke kezdetben a tömöríteni kívánt állomány első betűje. Az eljárás futása során mindig teljesül, hogy $z \in S$.

Az algoritmus általános lépése a következő:

1. Olvassuk a bemenő állomány következő betűjét K-ba.
2. Ha az előző olvasási kísérlet sikertelen volt (vége a bemenetnek), akkor írjuk ki $c(z)$-t, és álljunk meg.
3. Ha a zK szó is S-ben van, akkor $z \leftarrow zK$, és menjünk vissza (1)-re.
4. Különben (ha $zK \notin S$) írjuk ki $c(z)$-t, tegyük a zK szót S-be. Legyen $z \leftarrow K$, majd menjünk vissza (1)-re.

Példa
Legyen a tömörítendő szöveg: ABABABAACAACCBBAAAAAAAAA

A kezdeti kódtábla (a szöveg karakterei alapján):

karakter	kód
A	1
B	2
C	3

A kódolás során a bemenetet pontosan egyszer olvassuk végig úgy, hogy mindig a már ismert (kóddal rendelkező) leghosszabb szót keressük. Ha megtaláltuk, akkor:

- kiírjuk a talált szó kódját a kimenetre, és
- bővítjük a kódszavak halmazát az αK szó képével, ahol az α a talált szó és K a következő karakter.

Kezdetben a leghosszabb „ismert” szó az A, ennek kódja 1 és az új kóddal rendelkező szó az AB a 4 kóddal. A kódolás folyamán ezt az lépést ismételjük, amíg a szöveg végére nem érünk.

kód	aktuális szó	következő karakter	új kód
1	A	B	4
2	B	A	5
4	AB	A	6
6	ABA	A	7
1	A	C	8
3	C	A	9
1	A	A	10
8	AC	C	11
3	C	B	12
2	B	B	13
5	BA	A	14
10	AA	A	15
15	AAA	A	16
15	AAA	-	-

A kódolt üzenet tehát: 1 2 4 6 1 3 1 8 3 2 5 10 15 15

A **dekódoláshoz** ugyanazt a kezdeti, csak a karakterek kódját tartalmazó kódtáblát használjuk és szemléltethetjük ugyanazzal a táblázattal. Az első és utolsó oszlopot teljes egészében ki tudjuk tölteni, majd soronként haladunk. Jelen esetben az első két sorhoz tartozó aktuális szó nyilván ismert.

Mivel a második sor szavának első karaktere éppen az **első** sorban szereplő **következő karakter**, így az ismert. Ebből ismerjük mit kódolt az első sor új kódja, így folytathatjuk a kitöltést.

kód	aktuális szó	következő karakter	új kód
1	A	**B**	4
2	B	A	5
4	AB	A	6
6	ABA	A	7
...	...	...	...

A harmadik sorig mindig ismert volt a kódszóhoz tartozó szó mielőtt használni szerettük volna. Azonban következőnek azt a kódszót szeretnénk használni, aminek visszaállításához szükség lenne annak inverz képére. Nyilván amíg nem ismert

a kódhoz tartozó szó addig nem is használhatjuk. Szerencsére csak az első karakterére van szükség, ami ismert.

A dekódolás ez alapján már egyszerűen befejezhető, a táblázat meg fog egyezni a kódolásnál már bemutatottal és a dekódolt szöveg kiolvasható a második oszlopból.

Egy hosszú szöveg esetén az ismertetett eljárás annyi új kódszót is bevezethet, hogy az azok közötti keresés a teljes szöveg végigolvasásához lenne hasonló. Ezzel a módszer elveszítené hatékonyságát, ezért gyakorlatban korlátozzuk a kódszavak halmazát.

Ez történhet például

- a kódszavak számának korlátozásával;
- a kódszavakhoz tartozó szavak hosszának korlátozásával;
- azzal, hogy a bemenet csak egy kezdőszeletén építjük a szótárat, utána csak kódolunk.

Mivel a Huffman-kódolás csak a betűnkénti kódolások között optimális az LZW eljárás könnyen eredményezhet rövidebb kódolt alakot, annak ellenére is, hogy az itt használt kódszavakat még binárisan kódolni kell.

Az LZW eljárás egyszerűnek nevezhető (összehasonlítva például a Huffman kódolással), és mivel csak egyszer kell olvasni a bemenetet, hatékony is (amennyiben a kódszavak tárolása hatékony).

1.4. Feladatok

1.4.1. Naiv módszer, Huffman-kód

1. Keressünk olyan szöveget, ahol a Huffman-kódolás pontosan annyi bittel történik mint a naiv módszerrel!

2. Írjunk struktogramot, amely a bemenetként kapott kódolt szöveg és hozzá tartozó Huffman-kód kódfája alapján helyreállítja az eredeti szöveget!

3. Mi az optimális Huffman-kódja az alábbi ábécének, ha a karakterek gyakoriságát az első nyolc Fibonacci-szám adja: $a : 1, b : 1, c : 2, d : 3, e : 5, f : 8, g : 13, h : 21$? Tudjuk ezt általánosítani arra az esetre, amikor a gyakoriságok megegyeznek az első n Fibonacci-számmal?

4. Általánosítsuk a Huffman algoritmust ternáris kódszavakra (vagyis olyan kódolásra, amelyben a 0, 1, 2 jeleket használjuk), és bizonyítsuk be, hogy optimális ternáris kódolást eredményez!

5. Írjunk Huffman-kódolást megvalósító programot egy választott nyelven. Bemenetként egy szöveget kapjon, kimenet a kódolt szöveg és a kódfa legyen!

1.4.2. LZW

1. Szemléltessük a $ABCABCABCAAABBCCAABABA$ kódolását.

2. Dekódoljuk az $1, 3, 4, 5, 6$ üzenetet, ha a kezdeti tábla $A \to 1$ és $B \to 2$.

3. Adjunk meg olyan 20 hosszú három karaktert tartalmazó (legalább egyszer mindegyiket) szöveget, ami esetén a kódolt (LZW) szöveg pontosan 13 hosszú.

4. Keressünk olyan szöveget, aminek LZW kódolása rövidebb eredményt ad a Huffman-kódolással összehasonlítva.

5. Írjunk programot, ami az LZW kódolást és dekódolást valósítja meg!

2. fejezet

AVL fák

A bináris keresőfáknál láttuk, hogy a keresés műveletigénye lényegesen függ a fa szintjeinek számától, vagyis a fa magasságától. Egy fa magassága akkor mondható jónak, ha legfeljebb $c \cdot \log n$, ahol n a csúcsok száma és c egy kis pozitív állandó. A legterebélyesebb fáknál c érték 1 körül van. Az olyan fákat, ahol c 1-nél nem sokkal nagyobb, kiegyensúlyozott fáknak nevezzük.

Az AVL fák **magasság szerint kiegyensúlyozott** keresőfák.

1. Definíció. *A t fa kiegyensúlyozott bináris fa (KBF) $\Leftrightarrow$ t minden $(*p)$ csúcsára:*

$$\mid h(p \to right) - h(p \to left) \mid \leq 1$$

1. Tétel. *Tetszőleges nem üres n csúcsú AVL fa h magasságára igaz, hogy*

$$\lfloor \log n \rfloor \leq h \leq 1,45 \log n$$

Az AVL-fára, mint speciális alakú keresőfára, változatlanul érvényesek a keresőfákra bevezetett műveletek. Minden művelet (beszúrás és törlés) után ellenőrizzük, és ha kell, helyreállítjuk a fa kiegyensúlyozottságát. Az AVL fát láncoltan reprezentáljuk és a csúcsban tároljuk az egyensúlyát (balance), ahol

$$p \to b := h(p \to right) - h(p \to left)$$

és $p \to b \in \{-1, 0, +1\}$.

Jelölések

- A **csúcs jelzője** (indikátora) az '=', ha a csúcs két részfájának magassága egyenlő.
- A csúcs jelzője a '-', ha a csúcs baloldali részfájának magassága eggyel nagyobb, mint a jobboldali részfáé.
- A csúcs jelzője a '+', ha a csúcs jobboldali részfájának magassága eggyel nagyobb, mint a baloldali részfáé.

A **levelek jelzője** mindig az '='. (Ezért a leveleknél nem jelezzük az egyensúlyt.) Ha egy csúcs jelzője beszúrás vagy törlés miatt '++', vagy '- -' lesz (ez jelzi, hogy elromlott az AVL-tulajdonság), javítani kell.

2.1. AVL fa forgatások

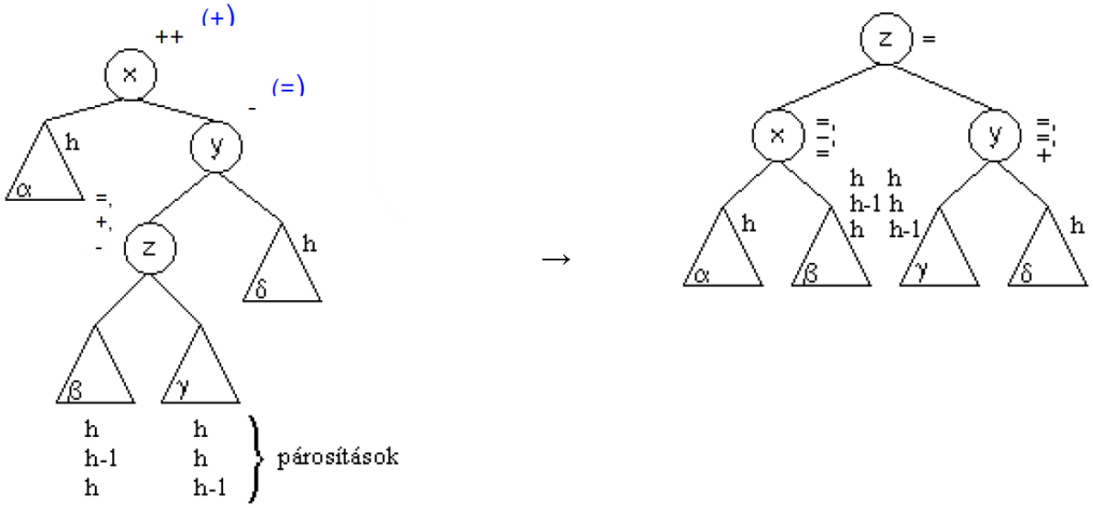
Csúcsok beszúrása esetén kétféle eset van, ahogyan elromolhat az AVL tulajdonság. Az egyik esetben a legjobboldalibb (vagy a legbaloldalibb) részfánál romlik el, a másik esetben pedig az ellenkező irányban.

(++, +) forgatás (tükörképe a (- -, -) forgatás)

A bal oldali ábrán látható AVL fánál elromlott az AVL tulajdonság, forgatással helyre kell hozni. A forgatás után α, β és γ részfák csúcsai a keresőfa tulajdonság szerinti relációk szerint helyezkednek el a fában az új y gyökércsúcs kulcsa alapján.

$\alpha < x < \beta < y < \gamma$ (a reláció az α, β, γ és δ részfák minden csúcsára igaz)

(++, -) forgatás (tükörképe a (- -, +) forgatás)



A bal oldali ábrán látható AVL fánál elromlott az AVL tulajdonság, forgatással helyre kell hozni. A forgatás után α, β és γ részfák csúcsai a keresőfa tulajdonság szerinti relációk szerint helyezkednek el a fában az új y gyökércsúcs kulcsa alapján.

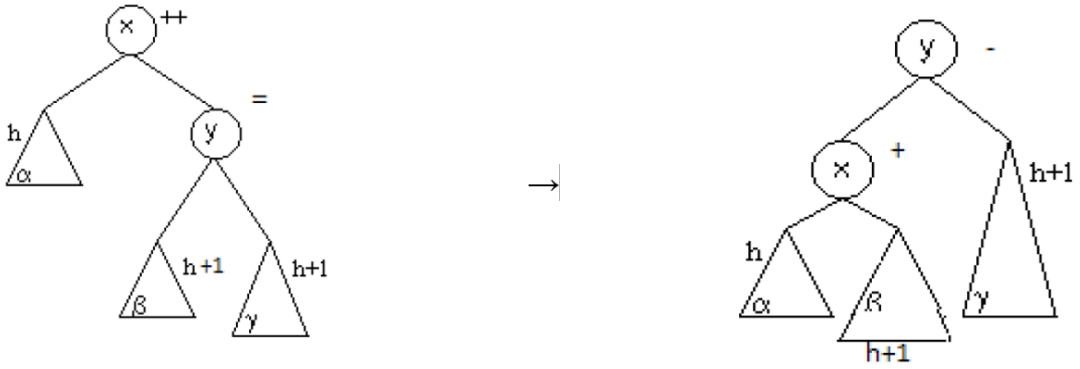
$\alpha < x < \beta < z < \gamma < y < \delta$ (a reláció az α, β, γ és δ részfák minden csúcsára igaz)

A (+ +, -) eset háromféleképpen állhat elő:

- A 'z' az új elem, részfák nincsenek, δ részfa sincs (h , h)
- Az új elem a γ részfába került (h-1 , h)
- Az új elem a β részfába került (h , h-1)

Csúcs törlésekor létrejöhet egy harmadik féle eset is, amikor az egyik részfa két szinttel eltér a másiktól (gyökér ++ vagy - - jelzőt kap), viszont az elromlott oldalon levő gyerek részfái egyenlő magasak lesznek.

(++, =) forgatás (tükörképe a (- -, =) forgatás)



A bal oldali ábrán látható AVL fánál elromlott az AVL tulajdonság, forgatással helyre kell hozni. A forgatás után α, β és γ részfák csúcsai a keresőfa tulajdonság szerinti relációk szerint helyezkednek el a fában az új y gyökércsúcs kulcsa alapján.

$\alpha < x < \beta < y < \gamma$ (a reláció az α, β, γ és δ részfák minden csúcsára igaz)

Ez a forgatási séma nem csökkenti az aktuális részfa magasságát, így ezután nem kell tovább ellenőrizni a címkéket.

2.2. AVL fába beszúrás

A beszúrás menete:

1. Megkeressük a kulcs helyét a fában.
2. Ha a kulcs benne van a fában, akkor KÉSZ vagyunk.
3. Ha a kulcs helyén egy üres részfa található, beszúrunk az üres fa helyére egy új a kulcsot tartalmazó levélcsúcsot, azzal, hogy ez a részfa eggyel magasabb lett.
4. Egyet felfelé lépünk a keresőfában. Mivel az a részfa, amiből felfelé építünk, eggyel magasabb lett, az aktuális csúcs egyensúlyát megfelelően módosítjuk.

(Ha a jobb részfa lett magasabb, hozzáadunk az egyensúlyhoz egyet, ha a bal, levonunk belőle egyet.)

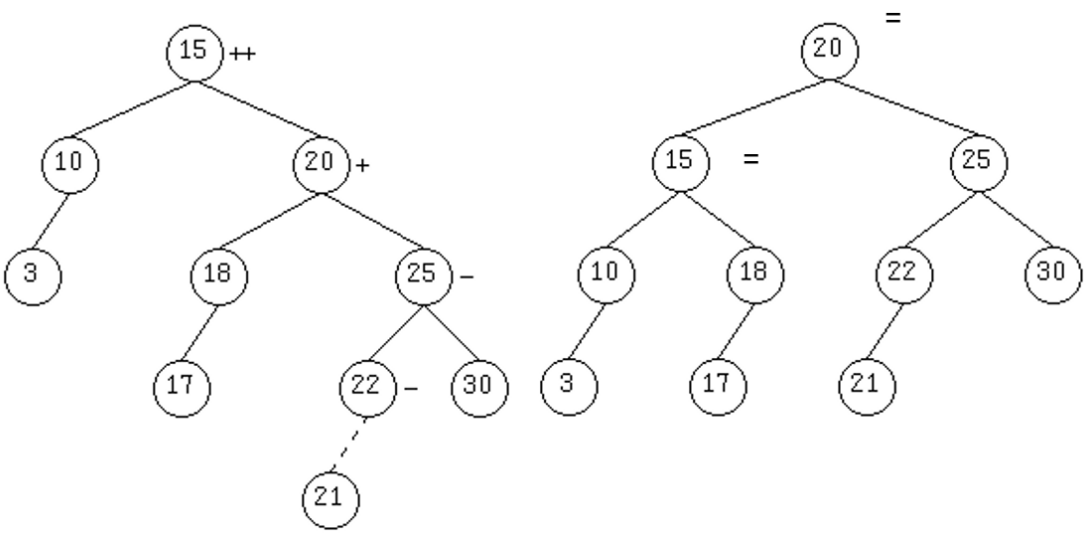
5. Ha az aktuális csúcs egyensúlya 0 lett, akkor az aktuális csúcshoz tartozó részfa alacsonyabb ága hozzánőtt a magasabbhoz, tehát az aktuális részfa most ugyanolyan magas, mint a beszúrás előtt volt, és így egyetlen más csúcs egyensúlyát sem kell módosítani, KÉSZ vagyunk.

6. Ha az aktuális csúcs új egyensúlya 1 vagy -1, akkor előtte 0 volt, ezért az aktuális részfa magasabb lett eggyel. Ekkor a 4. ponttól folytatjuk.

7. Ha az aktuális csúcs új egyensúlya 2 vagy -2, akkor a hozzá tartozó részfát ki kell egyensúlyozni. A kiegyensúlyozás után az aktuális részfa visszanyeri a beszúrás előtti magasságát, ezért már egyetlen más csúcs egyensúlyát sem kell módosítani, KÉSZ vagyunk.

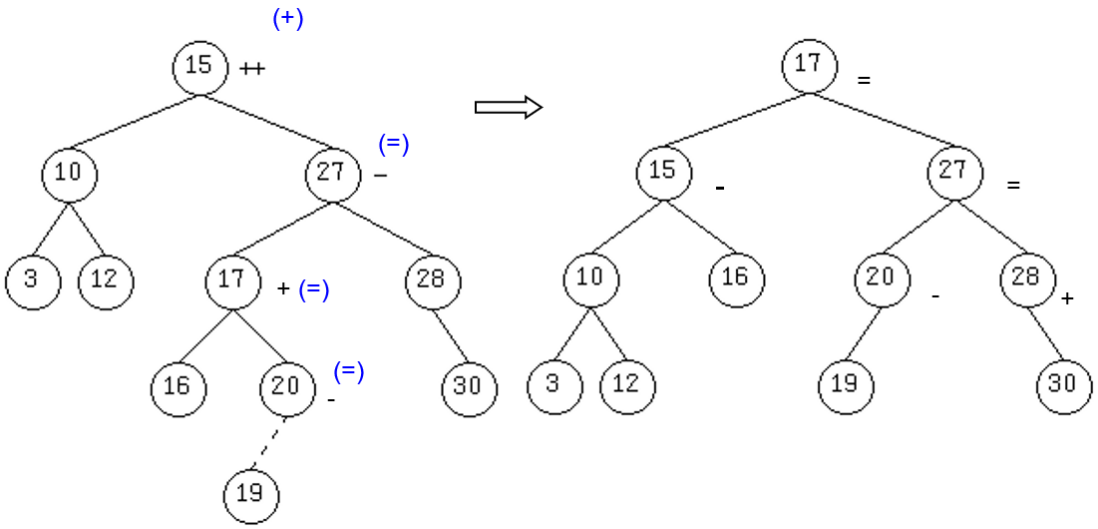
2.2.1. Példák beszúrásra

21 beszúrása



A beszúrt csúcstól haladjunk a gyökér felé. Addig megyünk, amíg '=', '++' vagy '–' jelzőjű csúcshoz, vagy a gyökérhez érünk. Ebben az esetben a gyökérben levő 15-ös kulcsú csúcsig lépkedünk, ennek jelzője '++'. A jobb gyerekének a jelzője '+', így **(++, +) forgatást** végzünk el. Ezzel helyreállítottuk a fát.

19 beszúrása



Most is a beszúrt csúcstól haladunk felfele. A gyökércsúcs '++' jelzőjű, a jobb gyereke '-', így **(++, -) forgatást** kell végrehajtanunk a fa kiegyensúlyozásához.

2.2.2. A beszúrás műveletigénye

A műveletigény az alábbiakból adódik össze:

- Beszúrás helyének megkeresése: $\log n$-nel arányos
- AVL tulajdonság ellenőrzése: $\log n$-nel arányos
- Jelzők beállítása: konstans időben

$$\longrightarrow \text{A teljes műveletigény: } \log n + \log n + konstans \approx \log n$$

2.3. AVL fából törlés

A megismert forgatások csökkentik a részfa magasságát, így törlésnél nem biztos, hogy egy forgatás után meg lehet állni a kiegyensúlyozással. Akár a gyökérig terjedhet a törlés hatása. Az általános törlő eljárás segéd eljárása lesz a minimális elem kivétele (törlése) a fából. Az eljárás a kiemelt minimum elem címét adja vissza.

Három eset van:

- Levelet törlünk. (Ez az eset a programban összevonható a következővel.)
- Egy gyerekes csúcsot törlünk.
- Két gyerekes csúcsot törlünk.

Levél törlése esetén a szülő megfelelő oldali részfája üres lesz és az egyensúlya eggyel nő vagy csökken.

Egy gyerekes csúcs törlése esetén a törlendő csúcs gyerekét beláncoljuk a szülő azon oldalára, ahonnan töröljük az elemet, és módosítjuk a szülő címkéjét, mert a megfelelő oldal mélysége eggyel csökkent. Az előbbi két eset fordul elő a minimum elem kivételénél.

Úgy is mondhatjuk, hogy „ha a törlendő csúcsnak egyik részfája üres, akkor a másik részfát tesszük a törlendő csúcs helyére", függetlenül attól, hogy ez a másik részfa üres-e vagy sem. Ilyen értelemben a fenti három esetből az első kettő összevonható.

Kétgyerekes elem törlésekor előbb kiemeljük a jobb oldali részfájának minimumát. A jobb oldali részfa szükség szerinti kiegyensúlyozása a minimumának kiemelésekor történik.

Ezután a törlendő elem helyére beláncoljuk a kiemelt minimumot, azaz a törlendő elem szülője erre a csúcsra fog mutatni, és ez a csúcs veszi át a törlendő elem két részfáját. Ezt követően beállítjuk aktuális részfa gyökerének egyensúlyát, és szükség esetén kiegyensúlyozzuk.

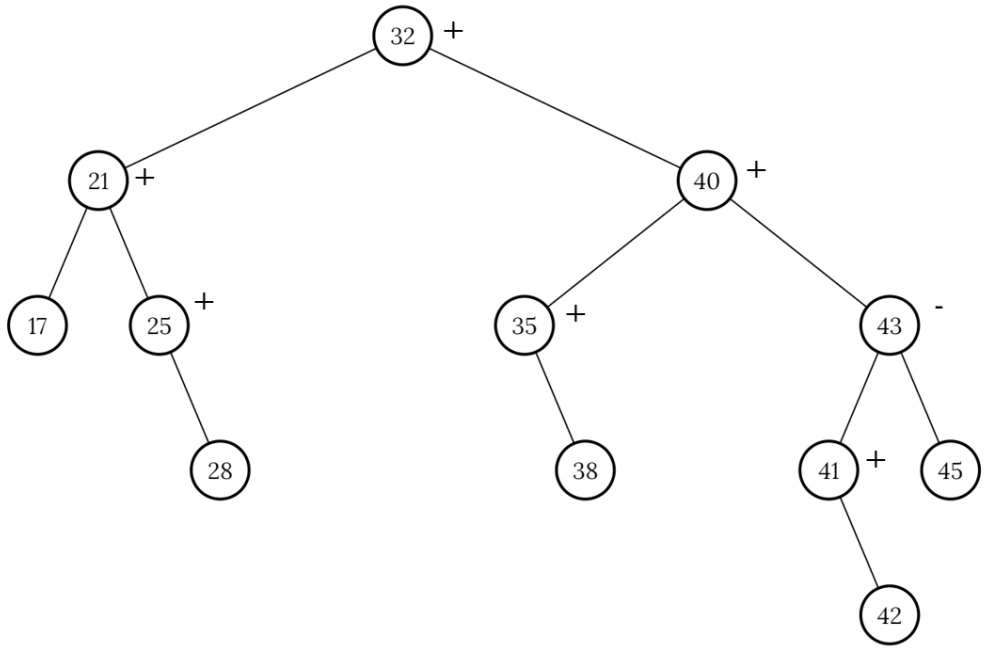
A jobb oldali részfa minimuma kisebb minden jobboldali elemnél és nagyobb a bal oldali részfa összes eleménél, így az új helyre való beillesztése nem rontja a keresőfa tulajdonságot. (Természetesen a bal oldali részfa maximuma is megfelelő lenne a törlendő elem helyére.)

Minimális elem kivétele a fából

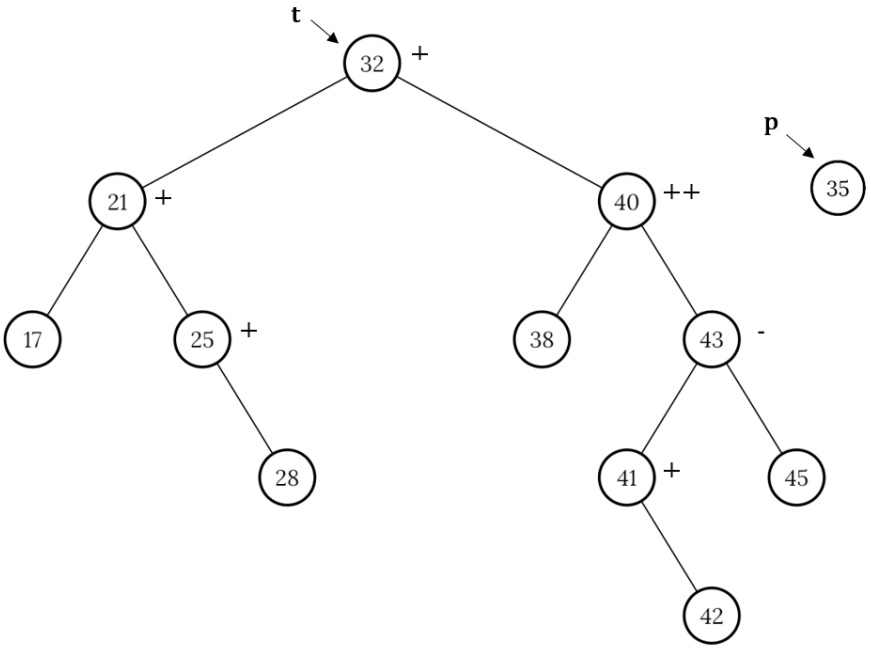
1. Induljunk el a gyökértől és haladjuk balra addig, amíg lehet.
2. Ha az aktuális csúcsnak nincs bal részfája, akkor ő a minimum. A címét visszaadjuk az eljárás output paraméterébe.
3. A minimum szülőjének bal részfája lesz a minimum csúcs jobb részfája.
4. Állítsuk át a szülő címkéjét, azaz növeljük eggyel, mert rövidült a bal részfa.
5. Ha kell, forgassunk.
6. Ha a részfa nem rövidült, azaz nem '=' lett a részfa gyökerének címkéje, akkor kész vagyunk.
7. Ha rövidült, a részfa, akkor ismételjük a szülő címkéjének javítását.

2.3.1. Példa törlésre

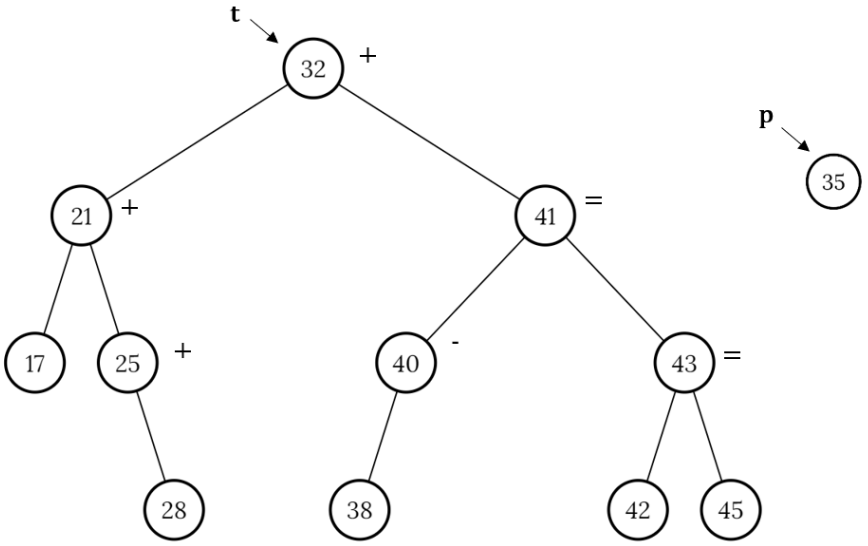
32 törlése



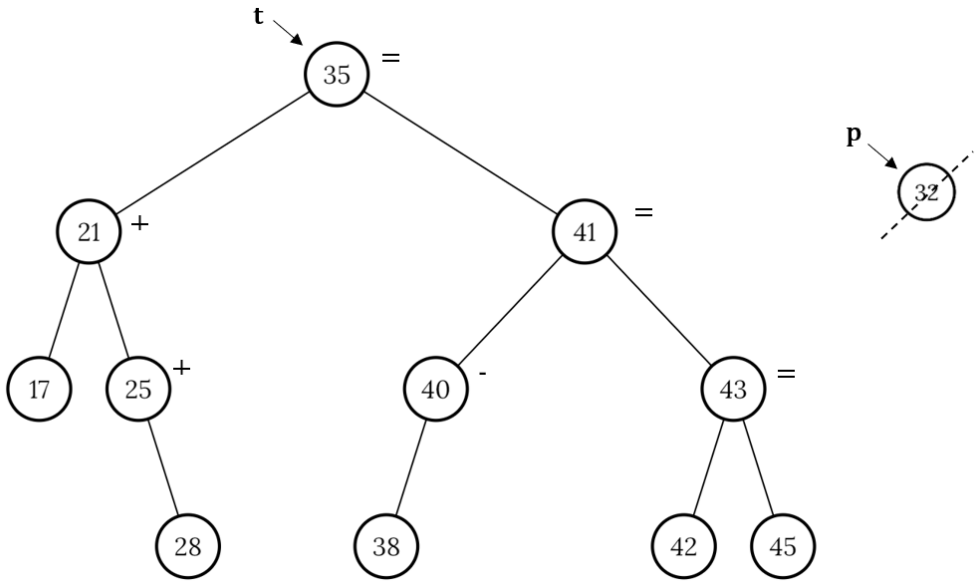
Mivel a fa gyökerét szeretnénk törölni, első lépésben meg kell keresnünk a jobboldali részfa minimumát. Ez a 35-ös kulcsú csúcs lesz.



Ezután a minimum elemet ki kell emelnünk a fából p pointer segítségével, és a törölni kívánt csúcsra ráállítjuk t pointert. Emellett be kell állítanunk a csúcsok címkéit.



Az AVL tulajdonság elromlik a minimális elem kivétele után, $(++, -)$ forgatással kell javítanunk.



Ezután a p pointerrel mutatott csúcsot beláncoljuk a t által mutatott gyökércsúcs helyére. Töröljük a régi gyökeret, és aktualizáljuk az új gyökér címkéjét. (Ha az új gyökér egyensúlya '++' vagy '–' lett volna, akkor további kiegyensúlyozásra lett volna szükség.)

2.4. Feladatok

1. Építsünk AVL fát a következő kulcsok egymás után történő beszúrásával: 12 20 22 8 5 10 25! Minden forgatás előtt adjuk meg a fa csúcsainak egyensúlyát és a forgatás típusát.

   (a) Töröljük a fából a 12-es csúcsot, majd mutassuk meg a helyreállítás lépéseit!

2. Építsünk AVL fát a következő kulcsok egymás után történő beszúrásával: 32 25 23 40 36 34 14!

   (a) Töröljük a fából a 32-es csúcsot, majd mutassuk meg a helyreállítás lépéseit!

3. Bizonyítsuk be, hogy minden n csúcsú AVL fa magassága $O(\log n)$! (Ha az AVL fa magassága h, akkor legalább F_h csúcsot tartalmaz, ahol F_h a h-adik Fibonacci szám.)

4. Adjuk meg a legkevesebb csúcsú, legmagasabb AVL fát!

5. Készítsük el az AVL fákat reprezentáló osztályt a szükséges műveletekkel (beszúrás, törlés, forgatások)!

3. fejezet

B+ fák

Az adathalmazokon a keresés és a módosítás műveletek hatékonyabban hajthatók végre, ha az értékeket rendezve tároljuk. Nem célszerű szekvenciálisan vagy láncolt listákban tárolni a rekordokat, mert így a műveletek nem lesznek túl hatékonyak.

Hatékonyabban valósíthatjuk meg a műveleteket, ha az adatokat keresőfába rendezve képzeljük el. Kézenfekvő megoldás lehetne az AVL fák vagy a piros-fekete fák alkalmazása, most azonban az adatokat egy véletlen elérésű háttértáron, pl. egy mágneslemezen kívánjuk elhelyezni. A mágneslemezek pedig úgy működnek, hogy egyszerre az adatok egy egész blokkját, tipikusan 512 byte vagy négy kilobyte mennyiségű adatot mozgatunk. Egy bináris keresőfa egy csúcsa ennek csak egy töredékét használná, ezért olyan struktúrát keresünk, ami jobban kihasználja a mágneslemez blokkjait.

Innen adódik a **B+ fák** ötlete, amiben minden csúcs legfeljebb d mutatót ($4 \leq d$), és legfeljebb $d-1$ kulcsot tartalmaz, ahol d a fára jellemző állandó, a B+ fa **fokszáma**. A belső csúcsokban mindegyik referencia két kulcs "között" van, azaz egy olyan részfa gyökerére mutat, amiben minden érték a két kulcs között található (mindegyik csúcshoz hozzáképzelve balról egy "mínusz végtelen", jobbról egy "plusz végtelen" értékű kulcsot).

Az adatok a levélszinten vannak. A belső kulcsok csak **hasító kulcsok**. Egy adott kulcsú adat keresése során ezek alapján tudhatjuk, melyik ágon keressünk tovább. A levélszinten minden kulcshoz tartozik egy mutató, ami a megfelelő adatrekordra hivatkozik. A gyökércsúcstól minden levél azonos távolságra kell legyen.

A d-ed fokú B+ fák **belső csúcsainak** tulajdonságai:

- Minden csúcs legfeljebb d mutatót ($4 \leq d$), és legfeljebb $d-1$ kulcsot tartalmaz. (d : állandó, a B+ fa fokszáma)

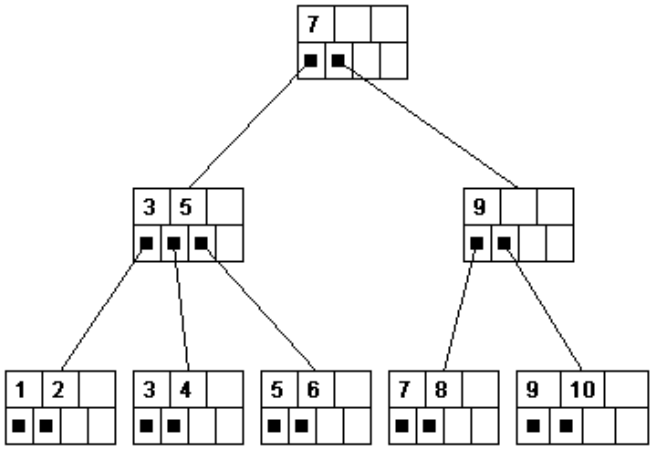
- Minden Cs belső csúcsra, ahol k a Cs csúcsban a kulcsok száma: az első gyerekhez tartozó részfában minden kulcs kisebb, mint a Cs első kulcsa; az utolsó gyerekhez tartozó részfában minden kulcs nagyobb-egyenlő, mint a Cs utolsó kulcsa; és az i-edik gyerekhez tartozó részfában $(2 \leq i \leq k)$ lévő tetszőleges r kulcsra $Cs.kulcs[i-1] \leq r < Cs.kulcs[i]$.
- A gyökércsúcsnak legalább két gyereke van (kivéve, ha ez a fa egyetlen csúcsa, következésképpen az egyetlen levele is).
- Minden, a gyökértől különböző belső csúcsnak legalább $\lfloor d/2 \rfloor$ gyereke van.

A d-ed fokú B+ fák **levélcsúcsainak** tulajdonságai:

- Minden levélben legfeljebb $d-1$ kulcs, és ugyanennyi, a megfelelő (azaz ilyen kulcsú) adatrekordra hivatkozó mutató található
- A gyökértől mindegyik levél ugyanolyan távol található.
- Minden levél legalább $\lfloor d/2 \rfloor$ kulcsot tartalmaz (kivéve, ha a fának egyetlen csúcsa van).
- A B+ fa által reprezentált adathalmaz minden kulcsa megjelenik valamelyik levélben, balról jobbra szigorúan monoton növekvő sorrendben.

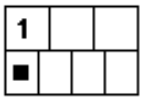
B+ fa zárójeles reprezentációja

[(1 2) 3 (3 4) 5 (5 6)] 7 [(7 8) 9 (9 10)]

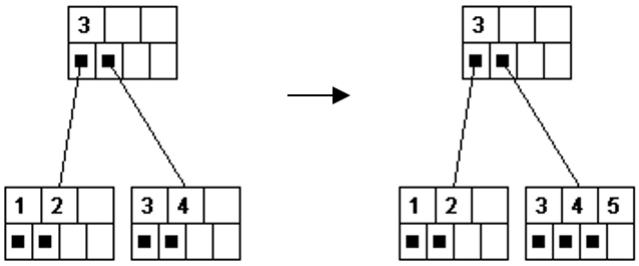


3.1. Beszúrás B+ fába

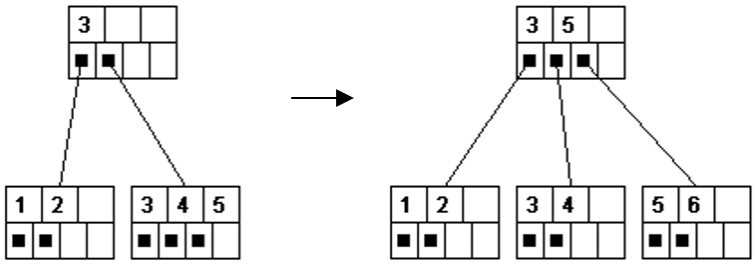
Ha a fa üres, hozzunk létre egy új levélcsúcsot, ami egyben a gyökércsúcs is, és a beszúrandó kulcs/mutató pár a tartalma! Különben keressük meg a kulcsnak megfelelő levelet! Ha a levélben már szerepel a kulcs, a beszúrás sikertelen. Különben menjünk az 1. pontra!



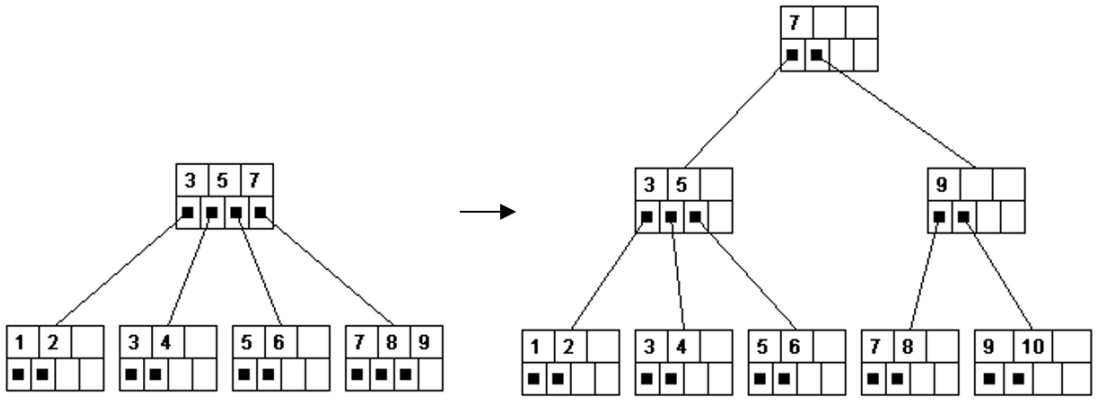
1. Ha a csúcsban van üres hely, szúrjuk be a megfelelő kulcs/mutató párt kulcs szerint rendezetten ebbe a csúcsba!



2. Ha a csúcs már tele van, vágjuk szét két csúccsá, és osszuk el a d darab kulcsot egyenlően a két csúcs között! Ha a csúcs egy levél, vegyük a második csúcs legkisebb értékének másolatát, és ismételjük meg ezt a beszúró algoritmust, hogy beszúrjuk azt a szülő csúcsba!



Ha a csúcs nem levél, vegyük ki a középső értéket a kulcsok elosztása során, és ismételjük meg ezt a beszúró algoritmust, hogy beszúrjuk ezt a középső értéket a szülő csúcsba! (Ha kell, a szülő csúcsot előbb létrehozzuk. Ekkor a B+ fa magassága nő.)

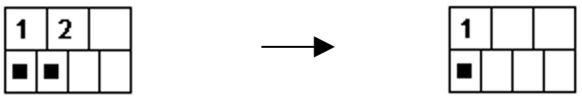


3.2. Törlés B+ fából

Keressük meg a törlendő kulcsot tartalmazó levelet! Ha ilyen nincs, a törlés meghiúsul. Különben a törlő algoritmus futása vagy az A esettel fejeződik be; vagy a B esettel folytatódik, ami után a C eset (nullaszor, egyszer, vagy többször) ismétlődhet, és még a D eset is sorra kerülhet végül.

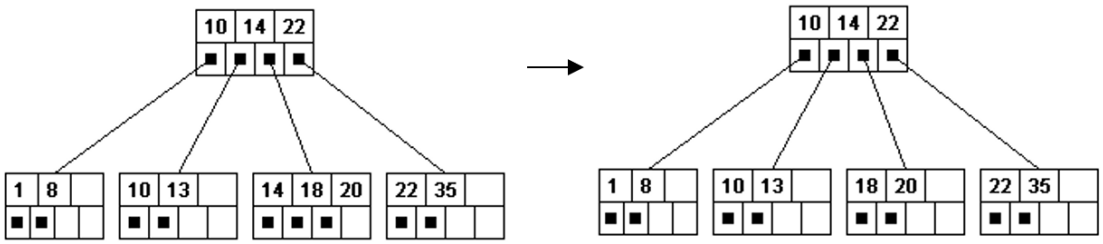
A, A keresés során megtalált levélcsúcs egyben a gyökércsúcs is:

1. Töröljük a megfelelő kulcsot és a hozzá tartozó mutatót a csúcsból!
2. Ha a csúcs tartalmaz még kulcsot, kész vagyunk.
3. Különben töröljük a fa egyetlen csúcsát, és üres fát kapunk.



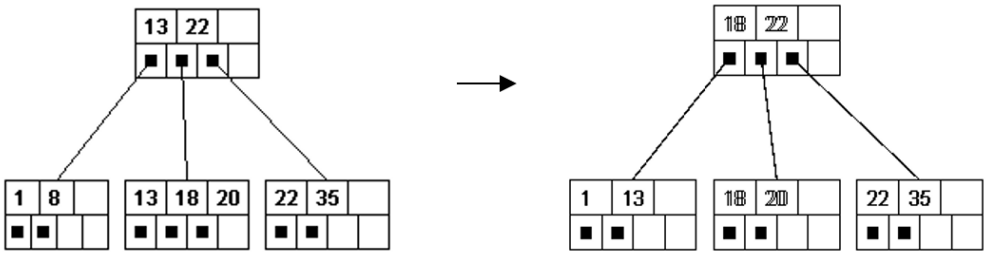
B, A keresés során megtalált levélcsúcs nem a gyökércsúcs:

1. Töröljük a megfelelő kulcsot és a hozzá tartozó mutatót a levélcsúcsból!
2. Ha a levélcsúcs még tartalmaz elég kulcsot és mutatót, hogy teljesítse az invariánsokat, kész vagyunk.



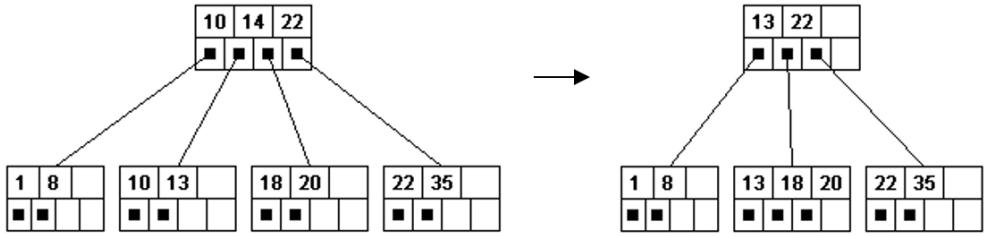
3.1. ábra. *A bal oldali B+ fából töröljük a 14-es kulcsot. Mivel ebben a levélben két másik kulcs is szerepel, az invariáns továbbra is teljesül, nincs más dolgunk. A hasító kulcsokat sem kell ebben az esetben változtatni.*

3. Ha a levélcsúcsban már túl kevés kulcs van ahhoz, hogy teljesítse az invariánsokat, de a következő, vagy a megelőző testvérének több van, mint amennyi szükséges, osszuk el a kulcsokat egyenlően közte és a megfelelő testvére között! Írjuk át a két testvér közös szülőjében a két testvérhez tartozó hasító kulcsot a két testvér közül a második minimumára!



3.2. ábra. *A 8-as kulcs törlésekor, mivel a jobb testvérének három kulcsa van, elosztjuk a megmaradt kulcsokat a két testvér között. A szülő csúcsban frissítjük a hasító kulcsot.*

4. Ha a levélcsúcsban már túl kevés kulcs van ahhoz, hogy teljesítse az invariánst, és a következő, valamint a megelőző testvére is a minimumon van, hogy teljesítse az invariánst, akkor egyesítsük egy vele szomszédos testvérével! Ennek során a két testvér közül a (balról jobbra sorrend szerinti) másodikból a kulcsokat és a hozzájuk tartozó mutatókat sorban átmásoljuk az elsőbe, annak eredeti kulcsai és mutatói után, majd a második testvért töröljük. Ezután meg kell ismételnünk a törlő algoritmust a szülőre, hogy eltávolítsuk a szülőből a hasító kulcsot, ami eddig elválasztotta a most egyesített levélcsúcsokat, a most törölt második testvérre hivatkozó mutatóval együtt.

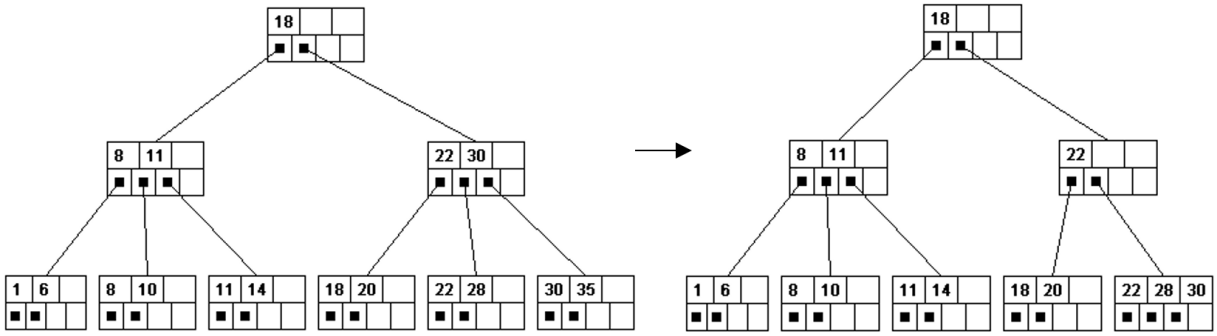


3.3. ábra. *A 10-es kulcs törlésekor az őt tartalmazó levélben csak a 13-as kulcs marad, így a következő testvérével egyesítenünk kell a csúcsokat. Ennek hatására a szülőből is törölnünk kell a 18, 20 kulcsokat tartalmazó levélcsúcshoz tartozó 14-es hasító kulcsot, és frissítsük a második levélhez tartozó hasító kulcsot.*

C, Belső — a gyökértől különböző — csúcsból való törlés:

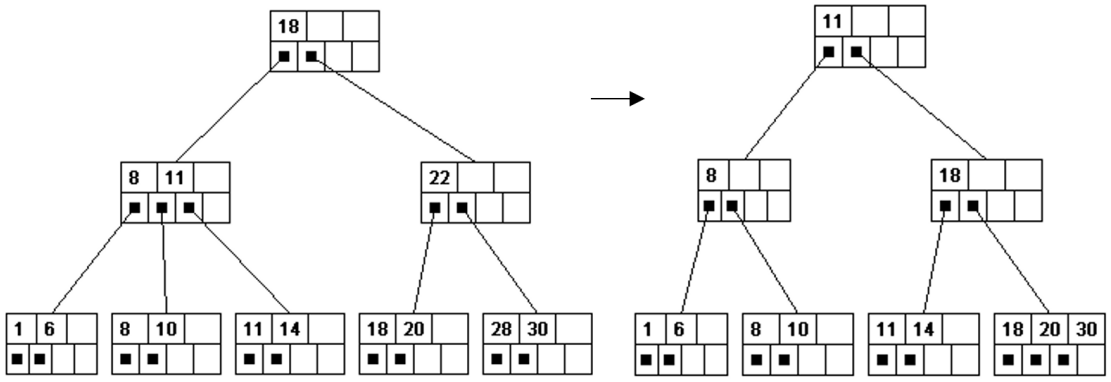
1. Töröljük a belső csúcs éppen most egyesített két gyereke közti hasító kulcsot és az egyesítés során törölt gyerekére hivatkozó mutatót a belső csúcsból!

2. Ha a belső csúcsnak van még floor(d/2) gyereke, (hogy teljesítse az invariánsokat) kész vagyunk.

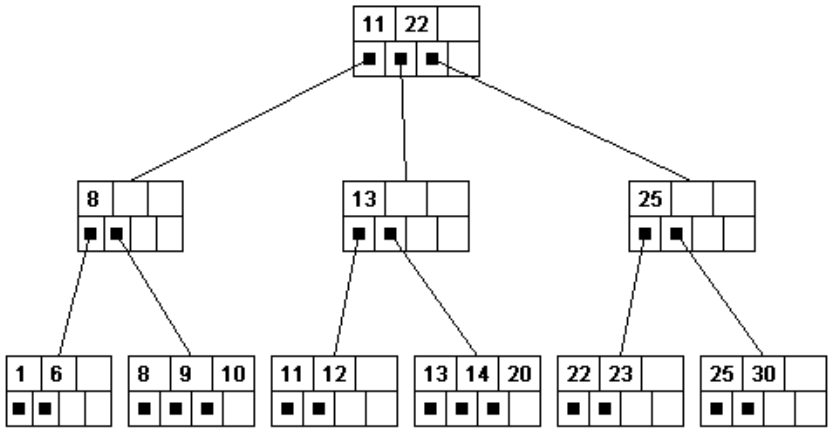


3. Ha a belső csúcsnak már túl kevés gyereke van ahhoz, hogy teljesítse az invariánsokat, de a következő, vagy a megelőző testvérének több van, mint amennyi szükséges, osszuk el a gyerekeket és a köztük levő hasító kulcsokat egyenlően közte és a megfelelő testvére között, a hasító kulcsok közé a testvérek közti (a közös szülőjükben lévő) hasító kulcsot is beleértve! A gyerekek és a hasító kulcsok újraelosztása során, a középső hasító kulcs a testvérek közös szülőjében a két testvérhez tartozó régi hasító kulcs helyére kerül úgy, hogy megfelelően reprezentálja a köztük megváltozott vágási pontot! (Ha a két testvérben a gyerekek száma páratlan, akkor az újraelosztás

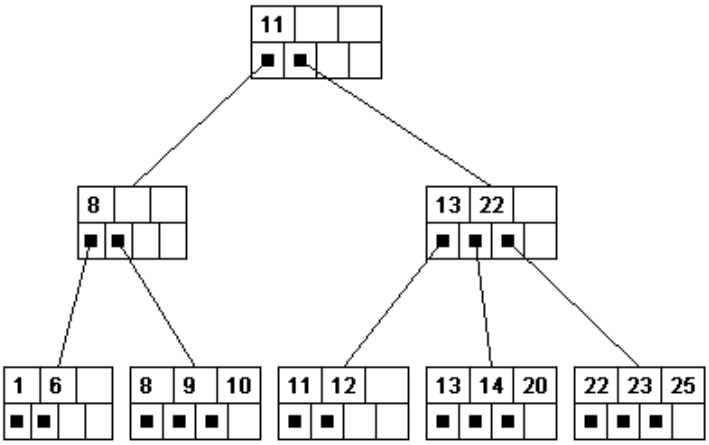
után is annak a testvérnek legyen több gyereke, akinek előtte is több volt!)



4. Ha a belső csúcsnak már túl kevés gyereke van ahhoz, hogy teljesítse az invariánst, és a következő, valamint a megelőző testvére is a minimumon van, hogy teljesítse az invariánst, akkor egyesítsük egy vele szomszédos testvérével! Az egyesített csúcsot a két testvér közül a (balról jobbra sorrend szerinti) elsőből hozzuk létre. Gyerekei és hasító kulcsai először a saját gyerekei és hasító kulcsai az eredeti sorrendben, amiket a két testvér közti (a közös szülőjükben lévő) hasító kulcs követ, és végül a második testvér gyerekei és hasító kulcsai jönnek, szintén az eredeti sorrendben. Ezután töröljük a második testvért. A két testvér egyesítése után meg kell ismételnünk a törlő algoritmust a közös szülőjükre, hogy eltávolítsuk a szülőből a hasító kulcsot (ami eddig elválasztotta a most egyesített testvéreket), a most törölt második testvérre hivatkozó mutatóval együtt.



3.4. ábra. *A 30-as kulcs törlésekor az őt tartalmazó levélben csak a 25-ös kulcs marad, a testvére is minimumon van, emiatt össze kell vonni ezeket a levélcsúcsokat.*



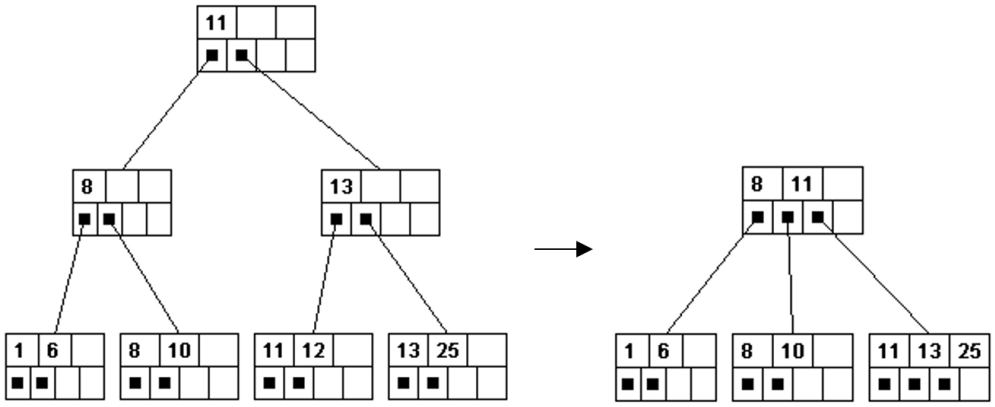
3.5. ábra. *Az összevonás után a 25-ös hasító kulcsot tartalmazó belső csúcsnak csak egy gyereke maradna, nem teljesíti az invariánst, a 13-as hasító kulcsot tartalmazó szomszédja is minimumon van, ezért egyesíteni kell őket.*

D, A gyökércsúcsból való törlés, ha az nem levélcsúcs:

1. Töröljük a gyökércsúcs éppen most egyesített két gyereke közti hasító kulcsot és az egyesítés során törölt gyerekére hivatkozó mutatót a gyökércsúcsból!

2. Ha a gyökércsúcsnak van még két gyereke, kész vagyunk.

3. Ha a gyökércsúcsnak csak egy gyereke maradt, akkor töröljük a gyökércsúcsot, és a megmaradt egyetlen gyereke legyen az új gyökércsúcs! (Ekkor a B+ fa magassága csökken.)



3.3. Feladatok

1. Hozzunk létre B+ fát a következő kulcsok egymás utáni beszúrásával:

   3, 8, 10, 22, 38, 2, 4, 13, 25, 24, 12, 11, 6, 18, 5, 30, 34, 42, 44

   (a) Töröljük egymás után a következő kulcsokat a fából: 4, 30, 12, 13!

   (b) Ezután szúrjuk be a következő kulcsokat a fába: 14, 7!

   (c) Adjuk meg az így kapott fa zárójelezett alakját is!

2. Adott a következő negyedfokú B+ fa (d=4, azaz 4 pointer található minden csúcsban).

   [(1 3) 6 (7 8 9) 11 (11 15) 20 (20 22)] 30 [(32 33 34) 38 (38 39) 40 (41 42)]

   (a) Rajzoljuk le, hogyan néz ki a fa, majd

   (b) szúrjuk be a fába a 35 és 10 kulcsokat!

   (c) Az eredeti fából töröljük a 38-as kulcsot.

3. A t pointer egy d fokszámú, h magasságú B+ fa gyökerére mutat. Írjuk meg a *printBplusTree*(t, d, h) eljárást, ami kiírja a B+ fa zárójelezett alakját csak kerek zárójeleket használva! Az algoritmus műveletigénye: $MT(n, d) \in O(n * d)$ legyen! (n a t fa csúcsainak száma)

4. Írjuk meg a B+ fák és műveleteik reprezentálására szolgáló osztályt! Az osztály segítségével bármilyen fokszámú B+ fát létre lehessen hozni!

4. fejezet

Gráfábrázolások

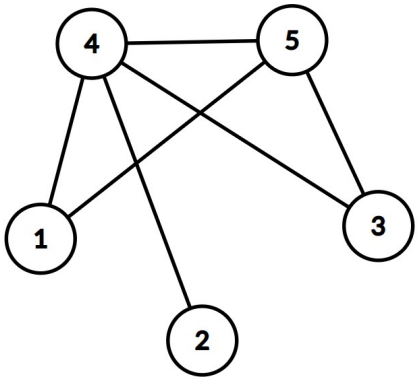
4.1. Gráfokkal kapcsolatos definíciók

2. Definíció. *Gráf alatt egy $G = (V, E)$ rendezett párost értünk, ahol V a csúcsok vagy pontok (vertices) tetszőleges, véges halmaza, $E \subseteq V \times V$ $\{(u, u) : u \in V\}$ pedig az élek (edges) halmaza. Az élek E halmazának elemei bizonyos V-beli párok. Ha $V = \{\}$, akkor üres gráfról, ha $V \neq \{\}$, akkor nemüres gráfról beszélünk.*

Párhuzamos és hurokélek nem szerepelhetnek a gráfokban, amikkel foglalkozni fogunk.

Irányítatlan gráf

3. Definíció. *A $G = (V, E)$ gráf irányítatlan, ha tetszőleges $(u, v) \in E$ élre $(u, v) = (v, u)$. Azaz, ha (u, v) él létezik, akkor (v, u) él is létezik, és mindkettőt ábrázolni kell.*

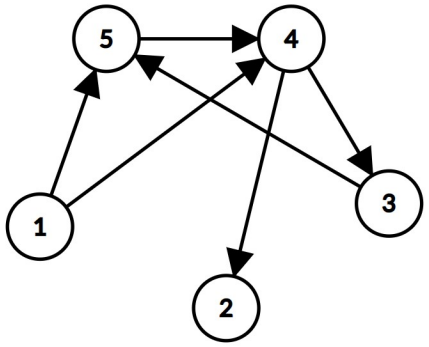


Az irányítatlan gráfokat szimmetrikus kapcsolatok leírására használhatjuk.

Az irányítatlan gráfok elképzelhetők speciális irányított gráfokként, ha mindkét élét oda és vissza mutató irányított éllel helyettesítjük. Ezzel az átírással irányított gráfokra megfogalmazott feladatok és algoritmusok értelmezhetők/használhatók irányítatlan gráfokra is.

Irányított gráf

4. Definíció. *A $G = (V, E)$ gráf irányított, ha tetszőleges $(u, v), (v, u) \in E$ élpárra $(u, v) \neq (v, u)$. Ilyenkor azt mondjuk, hogy az (u, v) él fordítottja a (v, u) él, és viszont.*



Az élek E halmazában levő csúcspárok rendezettek. Az irányított gráfokat nem szimmetrikus kapcsolatok leírására használhatjuk.

Az irányított gráfok is tekinthetők irányítatlannak úgy, hogy elfeledkezünk az élek irányításról. Ekkor nem teszünk különbséget az (u, v) és (v, u) él között.

Út

5. Definíció. *A $G = (V, E)$ gráf csúcsainak (V) egy $<u_0, u_1, ..., u_i>$ $(n \in \mathbb{N})$ sorozata a gráf egy útja, ha tetszőleges $i \in 1..n$-re $(u_{i-1}, u_i) \in E$. Ezek az (u_{i-1}, u_i) élek az út élei. Az út hossza ilyenkor n, azaz az utat alkotó élek számával egyenlő.*

Tetszőleges $<u_0, u_1, ..., u_i>$ út rész-útja $0 \leq i \leq j \leq n$ esetén az $<u_i, u_{i+1}, ...u_j>$ út.

Kör

6. Definíció. *A kör olyan út, aminek kezdő és végpontja (csúcsa) azonos, a hossza > 0, és az élei páronként különbözőek. Az egyszerű kör olyan kör, aminek csak a kezdő és a végpontja azonos. Tetszőleges út akkor tartalmaz kört, ha van olyan részútja, ami kör.*

Ritka gráf

7. Definíció. *Egy gráf ritka gráf, ha* $\mid E \mid$ *sokkal kisebb, mint* $\mid V \mid^2$.

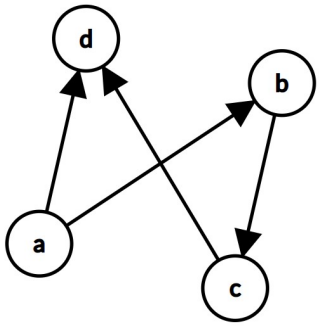
Sűrű gráf

8. Definíció. *Egy gráf sűrű gráf, ha* $\mid E \mid$ *megközelíti* $\mid V \mid^2$*-et.*

Két módszert szokás használni $G = (V, E)$ gráf ábrázolására a **szöveges** vagy **rajzzal** történő ábrázolás mellett. A gráfot megadhatjuk **szomszédossági csúcsmátrix**szal és **szomszédossági éllistá**val. Mindkét módszer alkalmas irányított és irányítatlan gráfok ábrázolására is. Leggyakrabban az éllistás ábrázolást használják, mert ezzel a ritka gráfok tömören ábrázolhatók. Sok gráfalgoritmus dolgozik szomszédossági éllistával megadott gráfokkal. A csúcsmátrixos ábrázolás sűrű gráfok esetén előnyösebb, vagy akkor, ha gyorsan kell eldönteni, hogy két csúcsot összeköt-e él.

4.2. Grafikus ábrázolás, szöveges ábrázolás

A **grafikus ábrázolás** a gráfok legismertebb ábrázolási módja. A csúcsokat kis körök jelölik, az éleket irányított gráfoknál a körök közti nyilak, irányítatlan esetben a köröket összekötő vonalak reprezentálják. A csúcsok sorszámát (illetve az azt reprezentáló betűt) általában a körökbe írjuk.

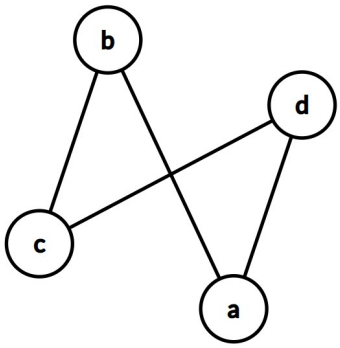


a → b; d
b → c
c → d

4.1. ábra. Irányított gráf grafikus és szöveges ábrázolása

A **szöveges ábrázolás** irányítatlan gráfoknál a szomszédokat írja le. Az $u - v_1; ...; vk$ azt jelöli, hogy a gráfban az u csúcs és a $v_1; ...; vk$ csúcsok között szerepelnek élek, és ezek $(u, v_1), ..., (u, vk)$.

Irányított gráfok esetén az $u \rightarrow v_1; ...; vk$ jelölés azt jelenti, hogy u csúcsból irányított élek indulnak ki $v_1; ...; vk$ csúcsokba. Az u csúcs közvetlen rákövetkezői, vagy gyerekei $v_1; ...; vk$ csúcsok.



a - b; d
b - c
c - d

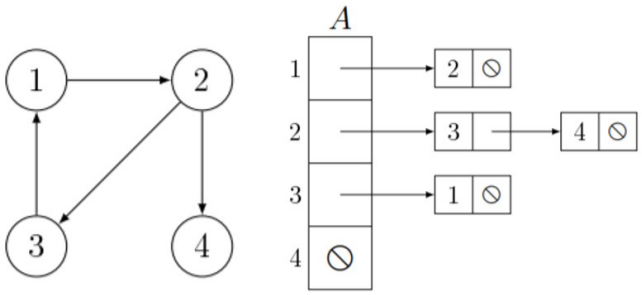
4.2. ábra. Irányítatlan gráf grafikus és szöveges ábrázolása

4.3. Szomszédossági listás ábrázolás

Az ábrázoláshoz egy tömböt használunk. Ez a tömb $\mid V \mid$ darab listából áll, és a tömbben minden csúcshoz egy lista tartozik.

Minden u csúcs esetén a tömbben az ahhoz tartozó szomszédossági lista tartalmazza az összes v olyan csúcsot, amelyre létezik $(u, v) \in E$ él. Azaz a lista elemei u csúcs G-beli szomszédjai. A szomszédossági listákban a csúcsok sorrendje általában tetszőleges. Irányítatlan és irányított gráfok esetén is alkalmazhatjuk ezt az ábrázolást. Az ábrázoláshoz szükséges tárterület $\Theta(V+E)$ mindkét esetben.

Ha G irányított gráf, akkor a szomszédsági listák hosszainak összege $\mid E \mid$, ugyanis egy (u, v) élt úgy ábrázolunk, hogy v-t felvesszük a megfelelő listába. Ha G irányítatlan gráf, akkor a szomszédsági listák hosszainak összege $2 \mid E \mid$, mivel (u, v) irányítatlan él ábrázolása során u-t betesszük v szomszédsági listájába, és fordítva.

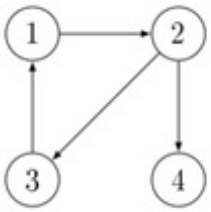


4.4. Szomszédossági mátrixos (csúcsmátrixos) ábrázolás

Feltesszük, hogy a csúcsokat tetszőleges módon megszámozzuk az $1, 2, \ldots, \mid V \mid$ értékekkel. A gráf ábrázolásához használt $A = (a_{ij})$ csúcsmátrix $\mid V \mid \times \mid V \mid$ méretű, és

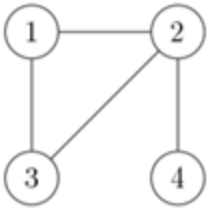
$$a_{ij} = \begin{cases} 1, & \text{ha } (i,j) \in E \\ 0, & \text{különben.} \end{cases} \tag{4.1}$$

Irányítatlan gráf esetén (u, v) és (v, u) ugyanaz az él, így a gráfhoz tartozó A csúcsmátrix megegyezik önmaga transzponáltjával. Így a mátrix szimmetrikus a főátlójára. Ebben az esetben tárolhatjuk a csúcsmátrixból csak a főátlóban és efölött szereplő elemeket, ezzel majdnem a felére csökkenthetjük a szükséges tárhelyet.



A	1	2	3	4
1	0	1	0	0
2	0	0	1	1
3	1	0	0	0
4	0	0	0	0

4.3. ábra. Irányított gráf grafikus és csúcsmátrixos ábrázolása



A	1	2	3	4
1	0	1	1	0
2	1	0	1	1
3	1	1	0	0
4	0	1	0	0

4.4. ábra. Irányítatlan gráf grafikus és csúcsmátrixos ábrázolása

4.5. Feladatok

4.5.1. G^2 gráf előállítása csúcsmátrixra, szomszédossági listára

1. Legyen $G = (V, E)$ egy irányított gráf. G^2 gráfnak nevezzük azt a $G^2 = (V, E^2)$ gráfot, melynek csúcsai megegyeznek az G gráf csúcsaival, élei pedig a következők: $(u, v) \in E^2 \Leftrightarrow (u, w)$ és $(w, v) \in E$. Azaz (u, v) éle a G^2 gráfnak pontosan akkor, ha létezik u-ból v-be kettő hosszú út az eredeti gráfban. Ha hurokél keletkezne, azt ne ábrázoljuk a négyzet gráfban.
   Készítsünk algoritmust, mely előállítja a G^2 gráfot

   (a) ha G csúcsmátrixszal van ábrázolva az A mátrixban, G^2 keletkezzen A^2 mátrixban. Műveletigény: $O(n^3)$.

   (b) ha G szomszédossági listával van ábrázolva A pointer tömbbel. G^2 keletkezzen A^2 tömbben. Műveletigény: $O(n * m)$

4.5.2. Abszolút nyelő csúcs keresése csúcsmátrixos ábrázolású irányított gráfban

2. Legyen $G = (V, E)$ egy irányított gráf. Csúcsmátrixszal ábrázolva az A mátrixban. Határozzuk meg van-e abszolút nyelő csúcsa a gráfnak, ha van, adjuk is meg a csúcsot. A gráf u csúcsát abszolút nyelőnek nevezzük, ha az u csúcs befoka $n - 1$, kifoka pedig 0. Azaz minden más csúcsból létezik u-ba mutató él, viszont u-ból nem indul ki egyetlen él sem. A feladat tehát a következő: egy olyan i indexű sort kell találni a mátrixban, hogy a sor csak nullát tartalmaz, de ugyanakkor az i-edik oszlop a főátlót kivéve csupa 1-est tartalmaz. Könnyű találni $O(n^2)$ műveletigényű algoritmust: keresünk egy csupa nulla sort (ez $O(n^2)$, ha találtunk, ellenőrizzük a megfelelő oszlopot, hogy a főátlót kivéve csupa egyes-e, ez $O(n)$, összességében $O(n^2)$. Oldjuk meg $O(n)$ műveletigénnyel!

5. fejezet

Szélességi gráfkeresés

A **szélességi keresés** (*BFS - Breadth-first Search*) az egyik legegyszerűbb gráfbejáró algoritmus, és ezen alapul sok fontos gráfalgoritmus. Adott $G = (V, E)$ **irányított vagy irányítatlan gráf** és egy kitüntetett s **kezdő csúcs** esetén a szélességi keresés módszeresen megvizsgálja G gráf éleit, és így megtalálja az összes s-ből elérhető csúcsot. Emellett kiszámítja az elérhető csúcsok távolságát s-től (legkevesebb él).

A csúcsok az algoritmus során **címkéket** kapnak:

- d - hány élből áll az aktuális legrövidebb út az adott csúcsba a kezdő csúcsból,
- π - melyik csúcsból jutunk el közvetlenül az adott csúcsba, melyik csúcs a szülője.

Ha egy adott v csúcs nem érhető el a start csúcsból, akkor $d(v) = \infty$ és $\pi(v) = \varnothing$. Emellett a start csúcsnak sincsen szülője, tehát $\pi(s) = \varnothing$.

Létrehoz egy s gyökerű **szélességi fát**, amely tartalmazza az összes, a kezdőcsúcsból elérhető csúcsot. Bármely s-ből elérhető v csúcsra, A szélességi fában s-ből v-be vezető út a legrövidebb s-ből v-be vezető útnak felel meg G-ben (bármely s-ből elérhető v csúcsra). **Legrövidebb út**nak most a legkevesebb élből álló utat nevezzük.

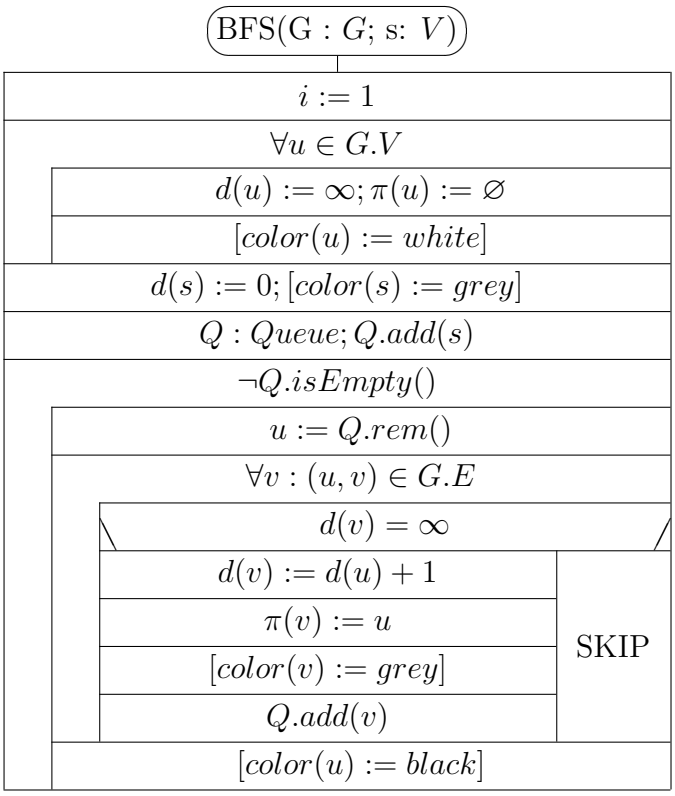
Az algoritmus a **csúcsok színezésével** tartja számon a bejárás pillanatnyi állapotát.

- Kezdetben minden csúcs **fehér**,
- majd az elért csúcsokat **szürkére**,

- aztán **feketére** színezzük.

Egy csúcs akkor válik **elértté**, amikor először rátalálunk a keresés során, ezután a színe nem lehet fehér. A szürke és fekete csúcsokat is megkülönbözteti az algoritmus, hogy a keresés jellege szélességi maradjon. A szürke csúcsokat már megtalálta az algoritmus, de még nem dolgozta fel, a fekete csúcsokat pedig már feldolgozta, többször nem kell érinteni őket.

Ha $(u, v) \in E$, és u fekete, akkor v fekete vagy szürke lehet. Tehát egy fekete csúcs összes szomszédja elért csúcs. A szürke csúcsoknak lehetnek fehér szomszédjaik. Ezek alkotják az elért és a még felfedezetlen csúcsok közötti határt.



Az algoritmus paraméterül a bejárandó G gráfot és a kijelölt s start csúcsot kapja meg.

Az első ciklus az inicializálásért felel. Minden csúcs d címkéjét ∞-re és π címkéjét $\varnothing$-ra állítja. Ez azt jelenti, hogy még egyik csúcsot sem értük el.

Az s kezdő csúcs d címkéjét 0-ra kell beállítani, ugyanis tudjuk, hogy az s-ből s-be vezető optimális út hossza 0.

Q sor segítségével tartjuk számon azokat a csúcsokat, amiket már elértünk, de a gyerekeiket még nem dolgozta fel az algoritmus (szürke csúcsok).

A második ciklus addig fut, amíg a sor ki nem ürül, vagyis amíg van elért, de nem feldolgozott csúcsa a gráfnak.

Az algoritmus eljut az összes olyan csúcsba, amely s-től k távolságra van, mielőtt egy $k+1$ távolságra levő csúcsot elérne.

Az s csúcstól k távolságra levő csúcsok a gráf k-adik szintjén vannak. Így a szélességi keresés a gráfot szintenként járja be, a nulladik szinttől kezdve egészen az utolsó szintig. Egy adott szintet teljesen feldolgoz, mielőtt a következő szint feldolgozását elkezdené, és egy szint feldolgozásakor éppen a következő szinten levő csúcsokat találja meg. Így lesz a gráfkeresés jellege *szélességi*, és az elnevezése is ebből adódik.

5.1. Szélességi fa

A szélességi keresés létrehoz egy **szélességi fát**, amely kezdetben csak a gyökeret tartalmazza, ami s kezdő csúcs.

- Ha egy fehér v csúcsot elérünk egy már elért u csúcshoz tartozó szomszédsági lista vizsgálata során, akkor a fát kiegészítjük a v csúccsal és az (u, v) éllel.
  - Azt mondjuk, hogy a szélességi fában u a v csúcs elődje vagy szülője $(\pi(v) = u)$.
- Egy csúcsot legfeljebb egyszer érhetünk el, így legfeljebb egy szülője lehet. Az ős és leszármazott relációkat, a szokásos módon, az s gyökérhez viszonyítva definiáljuk: ha u az s-ből v-be vezető úton helyezkedik el a fában, akkor u a v őse és v az u leszármazottja. Az **előd részgráf** ebben az esetben egy fa.

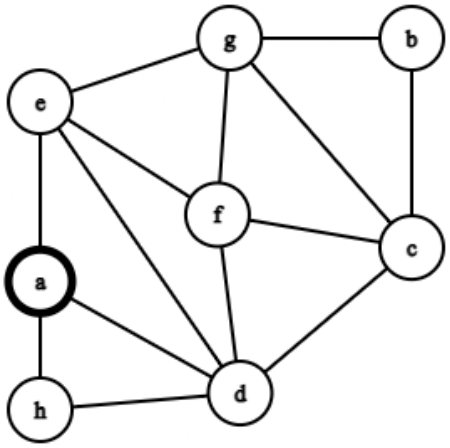
9. Definíció. *Előd részgráf:*
$G_\pi = (V_\pi, E_\pi)$, *ahol*

$$V_\pi = \{v \in V \,:\, \pi(v) \neq \varnothing\}$$
$$E_\pi = \{(\pi(v), v) \in E \,:\, v \in V_\pi \setminus \{s\}\}.$$

A szélességi fa másik neve **legrövidebb utak fája**, mivel minden, a kezdő csúcsból elérhető csúcsra a szélességi keresés alapján meghatározott legrövidebb utakat tartalmazza (fordított irányban).

5.2. A szélességi keresés példa

Szemléltessük a következő gráfra a szélességi bejárás működését. Legyen a start csúcs: $s = a$.



Konvenció: Nemdeterminisztikus esetekben a **kisebb indexű csúcsot** érjük el hamarabb (az kerül be előbb a sorba).

Kiterjesztett csúcs	Csúcsok d értékei								Sor tartalma	Csúcsok π értékei							
	a	b	c	d	e	f	g	h		a	b	c	d	e	f	g	h
	0	∞	∞	∞	∞	∞	∞	∞	$< \delta >$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$
a, d:0				1	1			1	<a,c,e,h>				a	a			a
d, d:1			2			2			<c,e,h,f>			d			d		
e, d:1							2		<e,h,f,b,g>							e	
h, d:1									<h,f,b,g>								
c, d:2		3							<f,b,g>		c						
f, d:2									<b,g>								
g, d:2									<g>								
b, d:3									<>								
	0	3	2	1	1	2	2	1		$\emptyset$	c	d	a	a	d	e	a

A táblázat utolsó sorából ki tudjuk olvasni a végső d és π értékeket a gráf minden csúcsára.

5.3. Feladatok

1. Adott egy sakktábla, rajta egy huszárral. Határozzuk meg a legkevesebb szükséges lépést, amit a huszárnak meg kell tenni ahhoz, hogy a megadott kezdőpontból eljusson a megadott végpontba. A megoldást szélességi keresés és sor felhasználásával valósítsuk meg!

2. Adott egy M × N mátrix, amiben 1 és 0 értékek találhatók. A mátrix egy táblát reprezentál, az 1 értékek a biztonságos mezőket, a 0 értékek az aknákat jelzik. A tábla első oszlopából kell eljutnunk az utolsó oszlopába a lehető legrövidebb úton, az aknák elkerülésével. Az aknákkal szomszédos nyolc mező is tudja aktiválni az aknákat. Egy mezőről a következő mezőre az oldalai mentén léphetünk tovább egy mezővel (átlósan nem). Határozzuk meg az első oszlop valamely mezőjétől az utolsó oszlop valamely mezőjére vezető legrövidebb utat, úgy, hogy közben nem aktiválunk egy aknát sem!

3. Adott egy bináris mátrix, amiben a 0-kkal jelzett mezők a vizet, az 1-es értékekkel jelzett mezők a szárazföldet jelzik. Az egymással szomszédos 1-es (átlósan is) mezők szigeteket formálnak. Számoljuk meg, hogy hány sziget található a területen!

4. Határozzuk meg a minimálisan szükséges dobókocka dobások számát, amivel meg lehet nyerni a Kígyók és létrák társasjátékot. A játéktábla 1-től 100-ig számozott mezőkből áll. Megtalálhatók rajta létrák, amelyek aljáról előre lehet lépni a létra tetején található mezőre és kígyók, amelyek fejétől vissza lehet csúszni a kígyó farkánál található mezőre. A megoldáshoz használjunk szélességi keresést!

6. fejezet

Mélységi gráfkeresés

A **mélységi keresés** (*DFS - Depth-first Search*) célja keresés a gráfban a lehető legmélyebben. Leginkább általános fák **preorder bejárásához** hasonlít. Csak **egyszerű irányított gráfokra** értelmezzük.

A keresés menete:

- A keresés során az utoljára elért, új kivezető élekkel rendelkező v csúcsból kivezető, még nem vizsgált éleket derítjük fel.
- Ha a v-hez tartozó összes élt megvizsgáltuk, akkor a keresés visszalép, és megvizsgálja annak a csúcsnak a kivezető éleit, amelyből v-t elértük.
  - Ezt addig folytatja, amíg el nem éri az összes csúcsot, amely elérhető az eredeti kezdő csúcsból.
- Ha marad érintetlen csúcs, akkor ezek közül valamelyiket kiválasztjuk, mint új kezdőcsúcsot, és az eljárást ebből kiindulva megismételjük.
  - Ezt egészen addig folytatjuk, amíg az összes csúcsot el nem érjük.

Az **elől részgráf** több fából is állhat, mivel a keresést többször hajthatjuk végre különböző kezdőcsúcsokból kiindulva.

10. Definíció. *Elől részgráf mélységi keresés esetén:*
$G_\pi = (V, E_\pi)$, *ahol*

$$E_\pi = \{(\pi(v), v) \,:\, v \in V \text{ és } \pi(v) \neq \varnothing\}.$$

Az előd részgráf egy **mélységi erdő**, amely több **mélységi fát** tartalmaz. E_π éleit **fa élek**nek nevezzük.

A csúcsok állapotait ebben az esetben is színekkel különböztetjük meg. Kezdetben minden csúcs **fehér**, amikor *elérünk egy csúcsot*, akkor **szürkére** színezzük azt, és ha *elhagytuk*, akkor **fekete** színű lesz (akkor, amikor a szomszédsági listájának minden elemét megvizsgáltuk). Ez biztosítja, hogy minden csúcs pontosan egy mélységi fában legyen benne, így ezek a fák diszjunktak legyenek.

A mélységi keresés minden csúcshoz **időpontot** rendel. Minden v csúcshoz két időpont tartozik:

- $d(v)$ **kezdési időpont** (discovery time), ezt akkor rögzítjük, amikor v-t elérjük (és szürkére színezzük);
- $f(v)$ **befejezési időpont** (finishing time), ezt akkor jegyezzük fel, amikor befejezzük v szomszédsági listájának vizsgálatát (és v-t befeketítjük).

A keresés eredményét befolyásolja, hogy milyen sorrendben vizsgáljuk meg az aktuális csúcsot követő csúcsokat, de alapvetően ekvivalens eredményeket kapunk.

Az algoritmus költsége $\Theta(V + E)$.

6.1. Élek osztályzása

Mélységi keresés segítségével a gráf éleit osztályokba sorolhatjuk.

Négy éltípust különböztethetünk meg G_π mélységi erdő segítségével, amelyet a $G_\pi = (V, E_\pi)$ gráf mélységi keresése során kaptunk.

- **Fa élek** (*tree edge*) a G_π mélységi erdő élei.
  - Az (u, v) él fa él, ha v-t az (u, v) él vizsgálata során értük el először.

  A fa élek mentén járjuk be a gráfot.
- **Visszamutató él** (*back edge*) (u, v) él, ha v őse u-nak egy mélységi fában.
  - A hurokéleket, amelyek előfordulhatnak irányított gráfokban, visszamutató éleknek tekintjük.
- **Előremutató él** (*forward edge*) az (u, v) él, ha v leszármazottja u-nak egy mélységi fában, és (u, v) nem éle a mélységi fának.

- **Kereszt él** (*cross edge*) az összes többi él.
  - Ezek ugyanazon mélységi fa csúcsait köthetik össze, ha azok közül egyik sem őse a másiknak, illetve végpontjaik különböző mélységi fákhoz tartozó csúcsok lehetnek.

Egy **irányított gráf** akkor és csak akkor **körmentes**, ha a mélységi keresés során nem találunk visszamutató éleket.

6.2. DFS irányítatlan gráfokra

Az élek a bejárás során irányítást kapnak. A bejárás során csak fa él és visszamutató él lesz a gráfban.

A visszamutató éleken keresztül megtaláljuk az irányítatlan gráfban is az irányítatlan köröket.

Az algoritmus hasznos, ha **kört keresünk** a gráfban, és ha a **komponenseket** szeretnénk meghatározni. Annyi komponensből áll a gráf, ahány mélységi fa keletkezik a bejárás során.

Az (u, v) él feldolgozásakor:

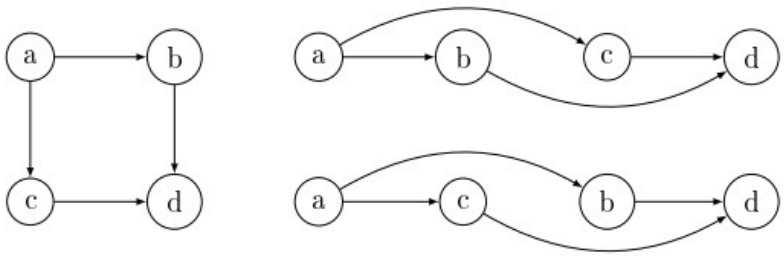
- v fehér $\longrightarrow$ (u, v) fa él
- v szürke $\longrightarrow$ (u, v) visszamutató él
- v fekete $(\pi(u) = v) \longrightarrow$ (u, v) fa él

6.3. Dag gráfok

A G irányított gráf akkor DAG (*Directed Acyclic Graph* = körmentes irányított gráf), ha nem tartalmaz irányított kört, vagyis a mélységi keresés során nem találunk benne visszaélt.

6.4. Topologikus rendezés

Irányított gráf **topologikus rendezése** alatt a gráf csúcsainak olyan sorba rendezését értjük, amelyben minden él egy-egy később jövő csúcsba (szemléletesen: balról jobbra) mutat.



6.4.1. DAG-ra a topologikus rendezés befokokkal

1. Határozzuk meg a csúcsok befokait (hány él vezet beléjük).
2. Kiválasztunk egy csúcsot, aminek a befoka 0, majd rakjuk be a rendezésbe.
   - Ilyen csúcsot mindig tudunk találni, ha a gráf DAG
3. Töröljük a kiválasztott csúcsot, az éleivel együtt. Csökkentsük ennek alapján a többi csúcs befokát.
4. Térjünk vissza a 2. lépésre, és folytassuk ugyanígy, amíg el nem fogynak a csúcsok.

Az elején határozzuk meg a befokokat (pl. egy segédtömbbe) és csak a 0 befokú csúcsokat gyűjtsük egy sorba (*Queue*). Járjuk végig ezt a sort. Az aktuálisan feldolgozott csúcsra vegyük az összes kimenő élét, a cél-csúcsok tárolt befokait csökkentsük, de se csúcsot, se élt ne töröljünk sehonnan. Ha ilyenkor egy befok eléri a 0-t, a hozzá tartozó csúcsot rakjuk a sorba. És így tovább, míg el nem fogy.

Az algoritmus fontos része a **körfigyelés**. Ez annyit jelent, hogy amikor végeztünk az algoritmussal, akkor nézzük meg, hogy bejártunk-e minden csúcsot, azaz üres lett-e a sor. Ha marad csúcs, akkor azok vagy részei egy körnek, vagy egy őse része egy körnek.

6.4.2. DAG-ra a topologikus rendezés DFS segítségével

Futtassuk le a mélységi bejárást a gráfon, közben a tanult módon figyeljük a visszaéleket, hogy eldönthessük, DAG-e (létezik-e topologikus sorrendje).

A következő egyszerű algoritmus topologikusan rendez irányított, körmentes gráfokat:

Algoritmus TopologikusRendezés(G)

1: Mélységi bejárás hívása G gráfra
2: minden csúcsra meghatározzuk $f(v)$ befejezési időt
3: a csúcsok elhagyásakor szúrjuk be azokat egy láncolt lista elejére
4: **return** a csúcsok láncolt listája

Tehát egy lehetséges topologikus rendezést kapunk, ha a DFS-sel kapott befejezési időpontok szerinti csökkenő sorrendben felsoroljuk a csúcsokat.

A topologikus rendezés elvégezhető $\Theta(V+E)$ időben, hiszen a mélységi bejárás ideje $\Theta(V+E)$, és a $\mid V \mid$ csúcs mindegyike $O(1)$ idő alatt beszúrható a láncolt lista elejére.

6.5. Erősen összefüggő komponensek

A mélységi bejárás egy másik klasszikus alkalmazása egy irányított gráf erősen összefüggő komponenseinek meghatározása.

Gyakran találkozhatunk olyan problémákkal, amiknek a megoldásához az adott gráfot részekre kell bontanunk. A részekre bontás alapja sok esetben a gráfok összefüggősége. Ez annyit jelent, hogy a gráfot olyan részekre bontjuk, amelyek a csúcsok közötti közlekedés szempontjából egybe tartoznak.

Egy **irányítatlan gráf** összefüggő, ha bármely két csúcsa összeköthető úttal. Irányítatlan gráfok összefüggő komponensei között nem találunk utat, de egy komponensen belül bármely csúcsból vezet út mindegyik másikba.

Irányított gráfok esetén kétféle összefüggőségről beszélhetünk. A gráf összefüggő, ha az élek irányításától eltekintve a gráf bármely két csúcsa között vezet út. Erős összefüggőség esetén bármely csúcs elérhető bármelyik másikból, a gráf irányított élein haladva.

Egy gráf erősen összefüggő komponenseit meghatározhatjuk két mélységi keresés felhasználásával.

$G = (V, E)$ gráfot erősen összefüggő komponenseire bontó algoritmus két mélységi keresést futtat le, G gráfra és a transzponáltjára, G^T-re.

11. Definíció. *Gráf transzponáltja:*
$G^T = (V, E^T)$, *ahol*

$$E^T = \{(u, v) \, : \, (v, u) \in E\}.$$

Azaz E^T a G gráf éleit tartalmazza az irányításuk felcserélésével. G szomszédossági listás ábrázolása esetén $O(V + E)$ idő alatt meghatározhatjuk G^T-t.

G és G^T összefüggő komponensei megegyeznek, ugyanis u és v csúcsok akkor és csak akkor érhetőek el egymásból G-ben, ha G^T-ben is elérhetőek egymásból.

A következő algoritmus lineáris időben ($\Theta(V + E)$) meghatározza egy $G = (V, E)$ irányított gráf erősen összefüggő komponenseit.

Algoritmus ErősenÖsszefüggőKomponensek(G)

1: Mélységi bejárás hívása G gráfra
2: minden csúcsra meghatározzuk $f(v)$ befejezési időt
3: G^T meghatározása
4: Mélységi bejárás hívása G^T gráfra, a csúcsokat $f(v)$ szerint csökkenő sorrendben vizsgáljuk
5: **return** az előző lépésben kapott mélységi erdő egyes fáinak csúcsai, mint erősen összefüggő komponensek

---

6.6. Feladatok

1. Adott egy M × N tábla, benne betűkkel, és egy lista szavakkal. Találjuk meg a lista szavai közül az összes lehetséges szót, amit a táblában szereplő szomszédos karakterekből képezhetünk! Egy mezőről az összes vele szomszédos mezőre (mind a nyolc irányban) tovább lehet lépni, de egy szóhoz nem használhatjuk fel ugyanazt a mezőt kétszer. Bemenetként a karaktermátrixot és a szavak listáját kapjuk meg, kimenetként adjuk meg a mátrixból kiolvasható szavakat.

2. Adott egy M × N mátrix, amiben karakterek találhatók. Keressük meg a leghosszabb utat egy előre megadott karakterből kiindulva, úgy, hogy a sorozatban egymást követő karakterek ábécé sorrendben növekvőek legyenek. Egy lépésben karakterből a vele szomszédos nyolc mező egyikére lehet továbblépni. Bemenetként a karaktermátrixot és a kezdő karaktert kapjuk, kimenetként a leghosszabb út hosszát adjuk meg.

3. Adott egy M × N bináris mátrix, ahol az 1-es értékek vizet, a 0-s értékek szárazföldet reprezentálnak. A szárazföld mezők összefüggő szigeteket alkotnak. A területen eső után a víz elönti azokat a szigeteket, amiket minden oldalról víz vesz körül. Adjuk meg, hogyan fog kinézni egy adott terület eső után!

4. Adott szavaknak egy nagy halmaza duplikátumokkal és egy k pozitív egész. Keressük meg a halmazban az első k szót, ami a legtöbbször fordul elő!

   - Például:
     - **Bemenet:**
       $szavak$ = ['betű', 'betűr', 'beterel', 'betűző', 'betűz', 'betűk', 'betűzött', 'beletűz', 'betűz', 'betűk', 'betűrődik', 'betűcsere', 'betűrím', 'betűsor', 'betű', 'betűr', 'betükröződik', 'betűz', 'betetőz', 'beton', 'betűz', 'betűk', 'beterít']
       $k = 3$
     - **Kimenet:**
       'betűz' 4-szer fordul elő
       'betűk' 3-szor fordul elő
       'betű' 2-szer fordul elő

7. fejezet

Minimális feszítőfák

Egy irányítatlan $G = (V, E)$ gráf minden (u, v) éle rendelkezik egy $w(u, v)$ **súllyal**, ami az u és v élek összekötésének költségét adja meg. A cél megtalálni azt a $T \subset E$ részhalmazt, aminek segítségével az összes csúcs összekapcsolható, és amelynek a

$$w(T) = \sum_{(u,v)\in T} w(u, v).$$

teljes súlya a lehetős legkisebb.

A körmentes és minden csúcspontot összekapcsoló T egy fa, amelyik „kifeszíti" G gráfot. A T fa meghatározásának problémáját **minimális feszítőfa problémá** -nak hívjuk.

A minimális feszítőfa meghatározására szolgáló alább vázolt algoritmusok **mohó algoritmusok** közé tartoznak. Általában egy algoritmus minden lépésben több lehetőség közül választ. A mohó stratégia ezek közül azt részesíti előnyben, amelyik az **adott pillanatban a legjobb**nak látszik. Az optimális megoldás megtalálását ez a stratégia általában nem garantálja. A minimális feszítőfa problémánál azonban bebizonyítható, hogy bizonyos mohó stratégiák minimális súlyú feszítőfához vezetnek.

7.0.1. Általános minimális feszítőfa algoritmus

Ez az algoritmus fokozatosan, újabb és újabb élek hozzáadásával állítja elő a feszítőfát. Egy minimális feszítőfát szeretnénk találni egy összefüggő, irányítatlan, $w : E \to R$ súlyfüggvénnyel élsúlyozott $G = (V, E)$ gráfban.

Az algoritmus az éleknek azt az A-val jelölt halmazát kezeli, amire a következő invariáns állítás teljesül:

Az iterációk előtt A valamelyik minimális feszítőfának a részhalmaza.

Az algoritmus minden lépésben azt az (u, v) élt határozza meg, amelyiket az A-hoz téve, továbbra is fennáll az előbbi állítás, miszerint $A \cup \{(u, v)\}$ is egy részhalmaza az egyik minimális feszítőfának. Egy ilyen élt A-ra nézve **biztonságos élnek** hívunk, mivel A-hoz történő hozzávétele biztosan nem rontja el az A-ra vonatkozó invariáns állítást.

Algoritmus MinimálisFeszítőfa(G)

1: Létrehozzuk A-t üres halmazként
2: Amíg A nem feszítőfa, addig keresünk A-ra nézve egy biztonságos (u, v) élt, és hozzáadjuk A-hoz
3: Végül A egy minimális feszítőfa lesz

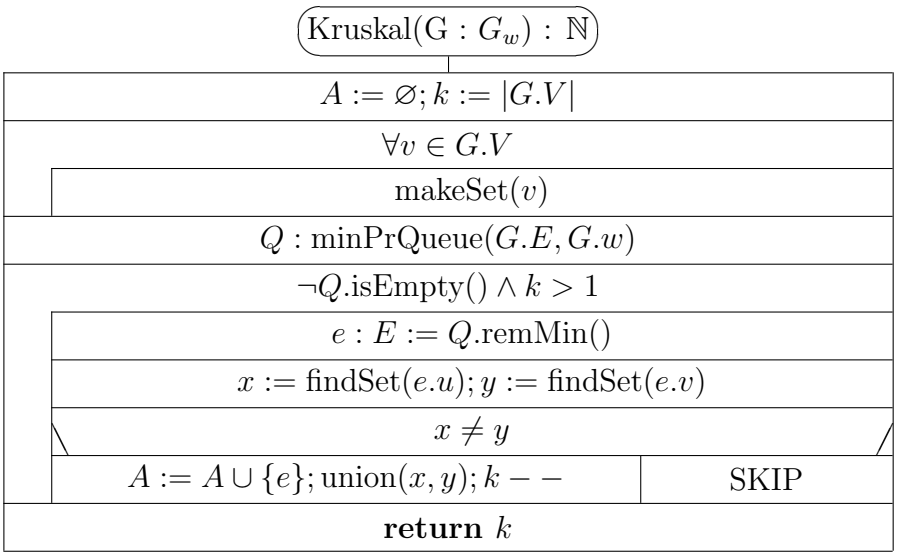
7.1. Kruskal algoritmus

Ebben az esetben A halmaz egy erdő. Az A-hoz hozzáadott biztonságos él mindig az erdő két különböző komponensét összekötő legkisebb súlyú él.

Az algoritmus minden lépésben megkeresi a $G_A = (V, E)$ erdő két tetszőleges komponensét összekötő élek közül a legkisebb súlyú (u, v) élt, és ezt veszi hozzá az egyre bővülő erdőhöz.

Legyen az erdő két fája C_1 és C_2, amit az (u, v) él összeköt. Mivel az (u, v) él a legkisebb súlyú olyan él is, amelyik C_1-et valamelyik másik komponenssel is összeköti, így (u, v) biztosan biztonságos él C_1-re nézve.

Az algoritmus mivel **mohó algoritmus**, így minden lépésben a lehetséges legkisebb súlyú élt adja hozzá az erdőhöz.



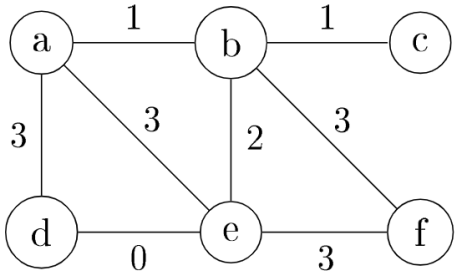
Az algoritmus lépései

1. A halmaz üres kezdőértéket kap, és létrejön a gráf csúcsait külön-külön tartalmazó $|V|$ darab fa.
2. Egy ciklussal megyünk, amíg van feldolgozatlan él. Kivesszük a következő legkisebb súlyú élt.
3. Minden így kapott (u, v) élre meg kell vizsgálni, hogy a végpontjaik ugyanahhoz a fához tartoznak-e.
   (a) Ha igen, akkor nem kell az éllel foglalkozni (az erdőben kör alakulna ki).
   (b) Ha nem, akkor (u, v) élt hozzá kell venni A-hoz, és a két fa csúcsait összevonjuk. (Ebben az esetben a két csúcs különböző fákhoz tartozott.)

Az algoritmus teljes költsége: $O(E \cdot logE)$

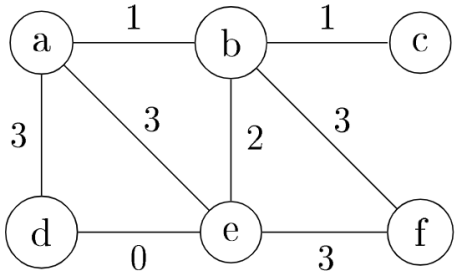
7.1.1. Kruskal példa

Szemléltessük Kruskal algoritmusát az alábbi összefüggő, élsúlyozott, irányítatlan gráfon!



Komponensek	Él	Él súlya	Biztonságos?
a, b, c, d, e, f	(d,e)	0	igen
a, b, c, de, f	(a,b)	1	igen
ab, c, de, f	(b,c)	1	igen
abc, de, f	(b,e)	2	igen
abcde, f	(a,d)	3	nem
abcde, f	(a,e)	3	nem
abcde, f	(b,f)	3	igen
abcdef	-	-	-

Az így kapott minimális feszítőfa:



7.2. Prim algoritmus

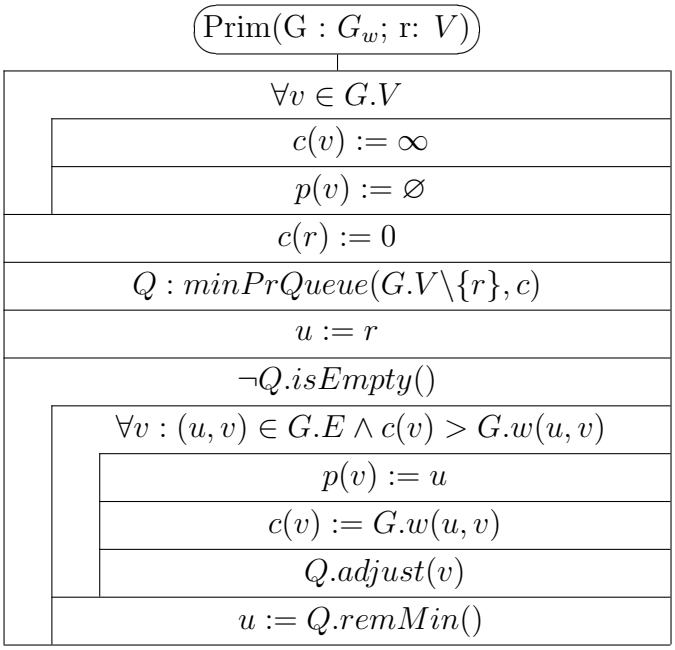
Az A halmaz egyetlen fa. Az A-hoz hozzáadott **biztonságos él** mindig a legkisebb súlyú olyan él, amelyik egy A-beli és egy A-n kívüli csúcsot köt össze. Mivel **mohó algoritmus**, így minden lépésben olyan éllel bővíti a fát, amely a lehető legkisebb mértékben növeli a fa teljes súlyát.

Az A fa építése egy tetszőlegesen kiválasztott gyökérpontból indul, és addig növekszik, amíg a V összes csúcsa bele nem kerül. Minden lépésben azt a **könnyű élt** vesszük hozzá az A fához, amelyik egy A-beli csúcsot köt össze a $G_A = (V, E)$ egy izolált csúcsával. Ez **biztonságos élt** ad A-ra nézve. Az algoritmus befejezésekor az A-beli élek minimális feszítőfát alkotnak.

Az algoritmus hatékony megvalósításának kulcsa az A-hoz hozzáadandó él ügyes kiválasztása. Tároljuk el az A fához hozzáadott éleket egy **minimum prioritásos sorban**, nevezzük ezt $minQ$-nak. Azt mondhatjuk, hogy a mindenkori **vágás** a $minQ$ és a V $minQ$ elemei között történik (legalábbis az első kör után, amikor már nem üres az egyik halmaz), azaz a már lezárt csúcshalmaz egyre terjeszkedik. Mindig a frissen a $minQ$-ból kikerült elemet a többire rávezető él lesz a könnyű, azaz a hozzáveendő.

A **vágás** a gráf csúcsainak két nemüres részhalmazra való bontása, osztályozása. Minden csúcs pontosan az egyikbe kerül.

Az algoritmus minden u csúcsra meghatározza a költségét ($c(u)$ vagy $d(u)$) és a szülőjét ($p(u)$).



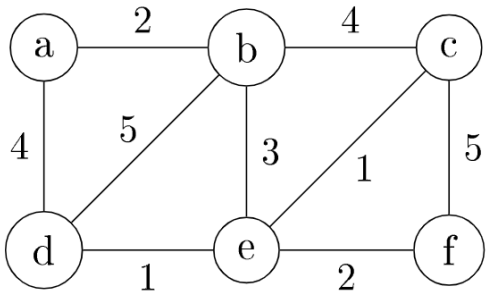
Az első ciklusban inicializáljuk minden csúcs szülőjét ismeretlenre és költségét végtelenre. Csak a kezdőcsúcs szülője marad NULL, a költsége semelyik csúcsnak sem lesz a végére végtelen.

A következő részben kiválasztjuk a tetszőleges kezdőcsúcsot (ez ingyen a fa része), létrehozunk egy minimum prioritásos sort és inicializáljuk azt, azaz bepakoljuk a kezdőcsúcsot 0-s és az összes többit ∞ prioritási értékkel.

A második ciklus során a kezdetben n elemű *minQ*-ból mindig az aktuálisan legolcsóbb csúcsot kivéve csúcsról csúcsra feldolgozzuk a gráfot. Végig nézzük az adott csúcs kimenő éleit, és ha úgy tűnik, ebből a csúcsból olcsóbb úton tudunk eljutni egy még nem véglegesített/nem látogatott másik csúcsba, akkor azt frissítjük. Ilyenkor az új prioritások mentén a sort is újra kellhet rendezni.

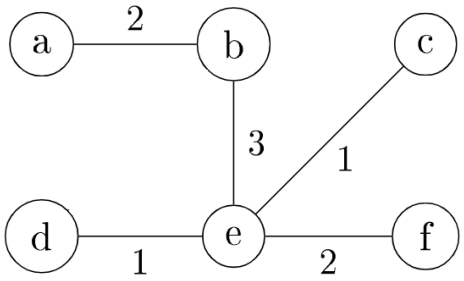
7.2.1. Prim példa

Futtassuk le a Prim algoritmust az alábbi gráfon az **a** csúcsból indítva!



c értékek Q-ban						mst-be:	p címkék					
a	b	c	d	e	f		a	b	c	d	e	f
0	∞	∞	∞	∞	∞		$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$	$\varnothing$
	2	∞	4	∞	∞	a		a		a		
		4	4	3	∞	b			b		b	
		1	1		2	e			e	e		e
			1		2	c						
					2	d						
						f						
0	2	1	1	3	2	eredmény	$\varnothing$	a	e	e	b	e

Az algoritmus lejátszása után kapott c és p értékeket a táblázat utolsó sorából olvashatjuk ki. A kapott p értékekből fel tudjuk rajzolni a minimális feszítőfát:



7.3. Feladatok

1. Kruskal algoritmusa ugyanazon G gráf esetén eltérő feszítőfákat eredményezhet attól függően, hogy az azonos súlyú éleket hogyan rendeztük sorba. Mutassuk meg, hogy minden T minimális feszítőfához megadható olyan rendezés, hogy az algoritmus éppen a T-t állítja elő.

2. Tegyük fel, hogy a G = (V, E) gráfot egy szomszédsági mátrixban ábrázoljuk. Adjunk megvalósítást Prim algoritmusára

   (a) $O(V^2)$ futási idővel,

   (b) és $O(m * \log n)$ futási idővel!

   (A műveletigények eltérését a felhasznált prioritásos sor különböző megvalósításai adják.)

3. Tegyük fel, hogy egy gráf súlyai 1 és $|V|$ közé eső egész számok. Milyen gyorssá tehető a Kruskal-algoritmus? Mit mondhatunk arról az esetről, amikor a súlyok 1 és W közé esnek, ahol W állandó?

4. Tegyük fel, hogy egy gráf súlyai 1 és $|V|$ közé eső egész számok. Milyen gyorssá tehető a Prim-algoritmus? Mit mondhatunk arról az esetről, amikor a súlyok 1 és W közé esnek, ahol W állandó?

5. Tegyük fel, hogy a G gráf minimális feszítőfáját már előállítottuk. Milyen gyorsan lehet módosítani ezt a fát, ha G-hez egy új csúcsot és ahhoz kapcsolódó éleket veszünk hozzá?

6. Tegyük fel, hogy egy gráf élsúlyai egyenletesen helyezkednek el a [0, 1) félig nyílt intervallumon. Kruskal vagy Prim algoritmusa a gyorsabb ilyenkor?

8. fejezet

Legrövidebb utak egy forrásból

8.1. Feladatok

8.1.1. Dijkstra algoritmus

1. Írjunk struktogramot, ami a Dijkstra algoritmus megvalósítását írja le

   (a) csúcsmátrixos ábrázolás estén,

   (b) szomszédossági listás ábrázolás esetén.

2. Adott egy olajfúró torony és néhány olajfinomító. Milyen csőhálózatot építsünk ki a torony és a finomítók között, hogy az olaj szállítása minimális összköltségű legyen? Tegyük fel, hogy a csővezetékeknél és az olajfúró toronynál nem ütközünk kapacitási korlátokba, és ismerjük az egyes pontok között létesíthető vezetékeken egy egységnyi olaj szállításának költségét.

   (a) Általánosítsuk az előző feladatot k olajfúró torony és m olajfinomító esetére.

3. Legyen G súlyozott gráf $c : E \to \{0, 1\}$ súlyfüggvénnyel. Valósítsuk meg a Dijkstra-algoritmust a szélességi bejárás módosításával, amelyben egy kétvégű sort használunk (azaz a sor mindkét végére helyezhetünk be új elemet, és vehetünk ki elemet).

4. Adott egy irányított hálózat, amelynek minden éléhez hozzárendelünk egy nemnegatív kapacitásértéket. Adjunk algoritmust két pont közötti legszélesebb (legnagyobb kapacitású) út megkeresésére. Egy út kapacitása alatt az utat alkotó élek kapacitásának minimumát (a legszűkebb keresztmetszetet) értjük.

8.1.2. Bellman-Ford algoritmus

5. Működik-e irányítatlan gráfokon a Bellman–Ford-algoritmus, és ha igen milyen feltételek mellett?

6. Adott egy súlyozott, irányított gráf. Amennyiben tartalmaz negatív összköltségű kört, írassuk ki egy ilyen kör mentén lévő csúcsokat.

7. Adottak valuták és valutaárfolyamok, azaz egy olyan irányított gráf, amelyben a csúcsok az egyes valuták, és az élek súlyai az árfolyamok. Az u-ból v-be menő él súlya megadja, hogy egységnyi u valutáért mennyi v valuta kapható. Váltsuk át u valutát v-re úgy, hogy a lehető legtöbb v-t kapjunk.

8. Adott egy élsúlyozott, irányított vagy irányítás nélküli véges gráf. Továbbá adott egy $s \in V$ forrás (kezdőcsúcs) és $v \in V$ célcsúcs. Adjunk algoritmust, amely meghatározza s-ből v-be vezető leghosszabb utat és annak hosszát. Milyen esetekben tudjuk megoldani a feladatot?

9. fejezet

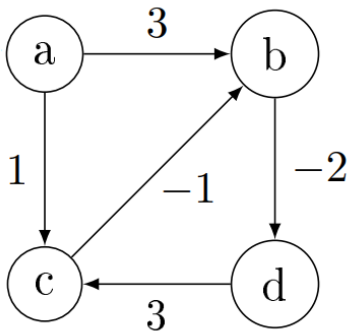
Legrövidebb utak minden csúcspárra

9.1. Floyd-Warshall algoritmus

9.2. Feladatok

1. Hogyan ismerhető fel a Floyd-Warshall algoritmus eredményéből, ha a gráf negatív összesúlyú kört tartalmazott?

2. Alkalmazzuk a Floyd-Warshall algoritmust az alábbi gráfra, és minden lépésben ábrázoljuk a változást csúcsmátrixon:



3. Írjunk programot, ami megvalósítja a Floyd-Warshall algoritmust!

10. fejezet

Gráf tranzitív lezártja

10.1. Warshall algoritmus

10.2. Feladatok

1. Adjunk algoritmust, amely egy irányított $G = (V, E)$ gráf tranzitív lezártját $O(V * E)$ időben megadja.

2. Tételezzük fel, hogy a tranzitív lezárt irányított körmentes gráfon $f(V, E)$ időben számítható, ahol az $f(V, E)$ mindkét változójában monoton növekvő függvény. Mutassuk meg, hogy tetszőleges $G = (V, E)$ irányított gráf $G^*(V, E^*)$ tranzitív lezártja megtalálható $f(V, E) + O(V + E^*)$ időben.

3. Írjunk struktogramot, ami a Warshall algoritmus megvalósítását írja le

   (a) csúcsmátrixos ábrázolás estén,

   (b) szomszédossági listás ábrázolás esetén.

4. Írjunk hatékony programot, ami egy bemenetként kapott gráf tranzitív lezártját adja eredményül!

11. fejezet

Mintaillesztés

A mintaillesztési probléma a következőképpen fogalmazható meg: tegyük fel, hogy a **szöveget** egy n hosszú $T[1..n]$ tömb tartalmazza, a **mintát** pedig egy m hosszú $P[1..m]$ tömbben tároljuk, és $1 \leq m \leq n$. Mindkét tömb elemei a Σ véges ábécé jelei.

P minta előfordul s eltolással a T szövegben akkor, ha P minta a T szöveg $(s+1)$-edik pozíciójára illeszkedik, vagyis, ha

$$0 \leq s \leq n - m \text{ és } T[s+1..s+m] = P[1..m].$$

Ekkor s-t **érvényes eltolásnak** nevezzük.

A mintaillesztési probléma megoldásakor egy adott P minta összes érvényes eltolását kell megtalálnunk a T szövegben.

Mindegyik algoritmus első lépésben valamilyen módon feldolgozza a mintát, és ezután találja meg az érvényes eltolásokat. Az első lépést **előfeldolgozásnak**, a másodikat **illesztésnek** nevezzük. Ezek idejének összegeként fogjuk megkapni a teljes futási időt.

Az érvényes eltolások halmaza S.

11.1. Egyszerű mintaillesztő algoritmus

Az egyszerű mintaillesztő algoritmus (*Brute force*) alapötlete az, hogy a mintát toljuk végig a szövegen, és balról jobbra ellenőrizzük, hogy a minta karakterei megegyeznek-e a szöveg lefedett karaktereivel. Ha nem egyeznek meg a karakterek, akkor eggyel arrébb toljuk a mintát, és ott végezzük el újra az ellenőrzést.

Egy ciklusban vizsgáljuk a $T[s+1..s+m] = P[1..m]$ feltétel teljesülését az összes, $(n-m+1)$ darab lehetséges s értékre.

A futási idő $O((n-m+1)*m)$, a legrosszabb futási idő pedig $\Theta((n-m+1)*m)$. A futási idő megegyezik az illesztési idővel, mert nincsen előfeldolgozás. Ez a módszer nem biztosít hatékony megoldást a problémára, ugyanis új s értékekkor figyelmen kívül hagyja a szövegre vonatkozó ismereteket, amiket a korábbi lehetséges s értékek kipróbálásakor szerzett.

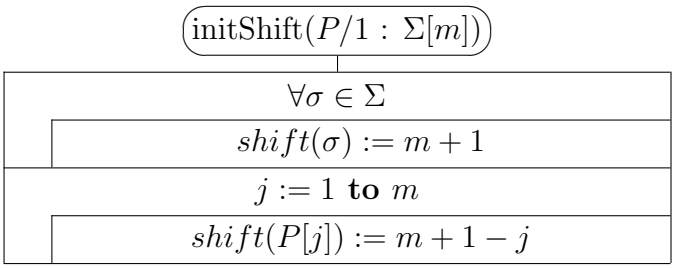
11.2. Quick-Search

Az alapötlet az, hogy ha elromlik az illeszkedés, akkor nézzük a szövegben a minta utáni karaktert, és úgy toljuk el a mintát, hogy illeszkedjen a szöveg ezen karakteréhez. Ha a mintában nem szerepel ez a karakter, akkor átugorjuk a mintával. Az új vizsgálatot mindig a minta elejéről nézzük.

A mintával való „ugrás" végrehajtásához bevezetjük a **shift** függvényt.

11.2.1. Shift függvény

A shift függvény az ABC minden betűjére megadja az "ugrás" nagyságát, amelyet akkor tehetünk, ha az illeszkedés elromlása esetén az illető betű lenne a szöveg minta utáni első karaktere.



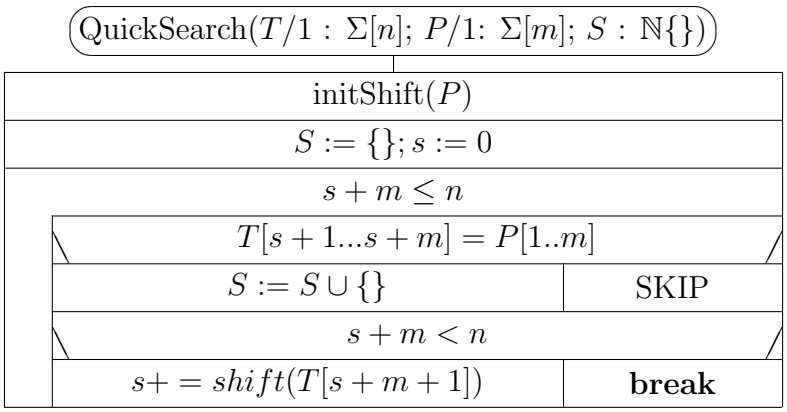
- Az alapértelmezett eltolás minden betűre a minta hossza + 1.
- Általános shift érték az adott betűhöz a minta hossza + 1 – j index.

Az **initShift** műveletigénye: $\Theta(d) + \Theta(m) \in \Theta(m)$

(d : az ábécé elemszáma, konstans).

Legjobb eset: $\ddot{\text{mO}}(n,m) \in \Theta(n/(m+1))$
→ (A minta első karakterénél már elromlik az illeszkedés, továbbá a minta utáni karakter sem fordul elő a mintában, így azt „átugorjuk".)

Legrosszabb eset: $\text{MÖ}(n) \in \Theta(n \cdot m)$
$\rightarrow$ ($m << n$ esetén; a minta végén romlik el az illeszkedés, így kicsi az „ugrás”.)



A szövegben „ugrálunk”, ezért az olyan adatszerkezeteknél ahol nem megengedett az indexelés, szükség van buffer használatára.

11.3. Knuth-Morris-Pratt (KMP) algoritmus

Ez az algoritmus egy lineáris idejű mintaillesztő algoritmus.

Ennél az algoritmusnál nem szükséges minden esetben a minta elejéről kezdeni az illeszkedést. Előfeldolgozással el tudjuk dönteni, hogy honnan kezdjük az illeszkedés vizsgálatát.

Ha a mintával akkorát ugrunk, hogy a minta kezdőszelete (**prefixe**) egy valódi végszeletnél (**szuffix**) kezdődjön, azaz a prefix a szuffixel kerüljön fedésbe, a prefixet már nem kell újra vizsgálni.

11.3.1. Előfeldolgozás

Az előfeldolgozás során definiálunk egy **next függvényt**, amely megadja a minta egyes kezdőrészleteire a leghosszabb egymással egyező prefix-szuffix párok hosszát. Ezt felhasználva tudjuk megadni a mintával való „ugrás” mértékét.

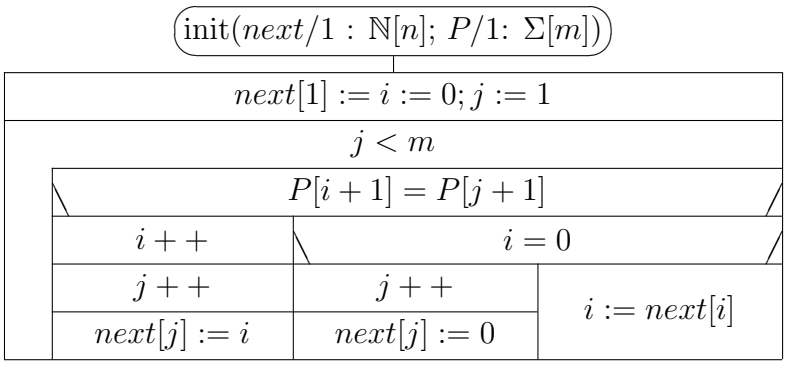
Megadja, hogy hogyan illeszkedik a minta **önmaga eltoltjaira**, így elkerülhetjük, hogy kizárható eltolási értékeket vizsgáljunk az illesztés során.

- A $next(j)$ a leghosszabb olyan P-beli prefix hossza, amely valódi szuffixe P_j-nek.

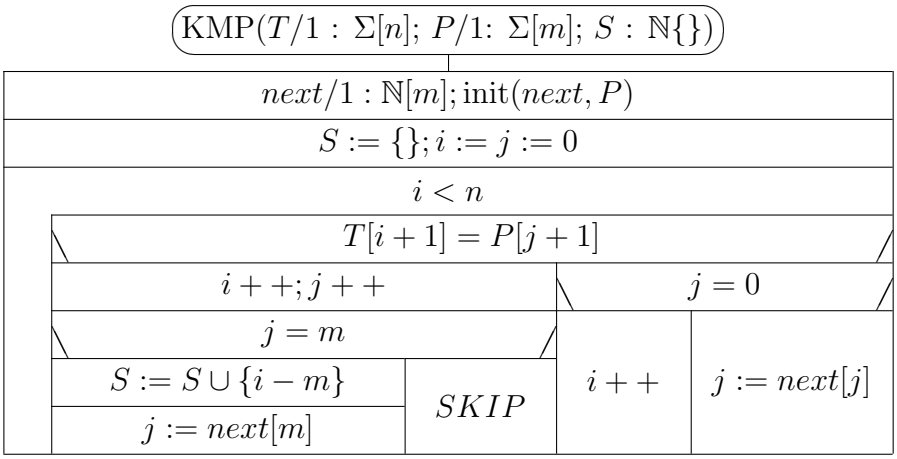
- A *next* függvényt $\Theta(m)$ idő alatt számítjuk ki.

A next függvény alapvető tulajdonságai:

1. $next(j) \in 0..(j-1)(j \in 1..m)$
2. $next(j+1) \leq next(j)+1(j \in 1..m-1)$
3. $P_{h+1} \sqsupset T_{j+1} \Leftrightarrow P_h \sqsupset T_j \wedge P[h+1] = T[j+1]$
4. $0 \leq h < j \leq m$ és $P_j \sqsupset T_i$ esetén $P_h \sqsupset Ti \Leftrightarrow P_h \sqsupset P_j$
5. $max_{l+1}H(j) = next(max_l H(j))(j \in 1..m, l \in 1..|H(j)|-1)$



Az algoritmus előnye az, hogy a szövegben nem kell visszaugrani. Ennek jelentősége például szekvenciális sorozat/fájl formában adott szövegnél van, mivel ekkor buffer használata nélkül is tudjuk alkalmazni a KMP algoritmust.



Az **init** műveletigénye $\Theta(m)$. (Ahol m a minta hossza.)

Tegyük fel, hogy $m \leq n$, ekkor a **KMP** műveletigénye legjobb és legrosszabb esetben is $\Theta(n)$.

($T \in \Omega(n)$, mivel i növekedni egyesével tud, és n-ig nő $\Leftarrow$ biztos van n lefutás $T \in O(n) 0 \leq j \leq i \leq n$, mivel a fő ciklus maximum $2n$-szer fut le.)

11.4. Feladatok

11.4.1. Quick-Search algoritmus

1. Számítsuk ki a shift[0..1] értékeket a $P = 000$ mintára. Futtassuk le a Quick-Search algoritmust a P mintára és a $T = 000010001010001$ szövegre.

2. Számítsuk ki a shif t['A'..'F'] értékeket a $P =$ ABABACD mintára. Futtassuk le a Quick-Search algoritmust a P mintára és a $T =$ BABAEABABAFDBABABACD szövegre.

3. Írjunk programot, ami megvalósítja a Quick-Search algoritmust! A program bemenetként kapja a T szöveget és P mintát, kimeneteként az érvényes eltolások halmazát kapjuk meg.

11.4.2. KMP algoritmus

4. Számítsuk ki a next[1..19] értékeket a következő mintára: *ababbabbabbababbabb*.

5. Számítsuk ki a next[1..6] értékeket a $P =$ babbab mintára. Futtassuk le a Knuth-Morris-Pratt algoritmust a P mintára és a $T =$ *ababbabbababbababbabb* szövegre.

6. Számítsuk ki a next[1..4] értékeket a $P = 0001$ mintára. Futtassuk le a Knuth-Morris-Pratt algoritmust a P mintára és a $T = 000010001010001$ szövegre.

7. Írjunk programot, ami megvalósítja a KMP algoritmust! A program bemenetként kapja a T szöveget és P mintát, kimeneteként az érvényes eltolások halmazát kapjuk meg.

Irodalomjegyzék

[1] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein - *Új algoritmusok*

[2] Ásványi Tibor - *Algoritmusok és Adatszerkezetek II. előadásjegyzet*

[3] Ivanyos Gábor, Rónyai Lajos, Szabó Réka - *Algoritmusok*

[4] Nagy Ádám: Algoritmusok és adatszerkezetek II. gyakorlati segédlet - *Tömörítés*

[5] CSci 340 - *B+-trees*

[6] Szita B. - *Minimális költségű feszítőfák*

[7] *https://www.techiedelight.com/*

[8] *https://people.inf.elte.hu/groberto/elte_alg2/gyakorlati_anyagok/elte_alg_feladatgyujtemeny.pdf*