

Tömörítés

Alapfogalmak

- **Ábécé:** $\Sigma = \{\sigma_1, \sigma_2, \dots, \sigma_d\}$
- **Ábécé mérete:** d
- **Tömörítendő szöveg hossza:** n
- **Kódszó hossza:** $L = \lceil \log_2 d \rceil$ (kettes alapon vett logaritmus, felfelé kerekítve)
- **Kódtáblázat:** Az ábécé karaktereihez hozzárendelt, fix hosszúságú bináris kódokat tartalmazza
- **Kódfa:**
 - levelei az ábécé karaktereihez tartoznak
 - minden élhez egy bináris címkét rendelünk: bal él $\rightarrow 0$, jobb él $\rightarrow 1$
 - egy levélhez vezető úton az élek címkéit összeolvasva kapjuk meg az adott karakter kódszavát
 - a kódfában nincs olyan kódszó, ami egy másik kódszó előtagja (prefixmentes kód)

Naiv módszer

A naiv egy olyan veszteségmentes tömörítési eljárás, amelyben minden karaktert azonos hosszúságú, fix bitsorozattal kódolunk. A kódhossz a karakterkészlet méretének függvényében a legkisebb olyan egész szám, amelyen minden karakter külön bináris kódot kaphat.

Tömörítési eljárás

1. Meghatározzuk az ábécé elemeit: $\Sigma = \{ \dots \}$
2. Meghatározzuk az ábécé méretét: $d = \dots$
3. Meghatározzuk a tömörítendő szöveg hosszát: $n = \dots$
4. Meghatározzuk kódszó hosszát: $L = \lceil \log_2 d \rceil$
5. minden karakterhez egyedi, L bites kódszót rendelünk
6. A bemeneti szöveget karakterenként helyettesítjük a kódszavával
7. A kimeneti fájl tartalmazza:
 - a. a kódtáblázatot
 - b. a tömörített kódsorozatot

Kitömörítési eljárás

Minden L -bites szakaszhoz a kódtáblázat alapján hozzárendeljük a megfelelő karaktert

Példa 1

- Tömörítendő szöveg: ABRAKADABRA

1. $\Sigma = \{A, B, R, K, D\}$
2. $d = 5$
3. $n = 11$
4. $L = \lceil \log_2 5 \rceil = 3$
- 5.

Karakter	Kód
A	000
B	001
D	010
K	011
R	100

6. 000 001 100 000 011 000 010 000 001 100 000
A B R A K A D A B R A

- Tömörített szöveg mérete: $n * L = 11 \cdot 3 = 33$ bit
- Kódtábla mérete: $d * 8 + d * 3 = 5 * 8 + 5 * 3 = 40 + 15 = 55$
- Teljes tömörített méret: $33 + 55 = 88$ bit (*plusz meta adatok*)
- Eredeti méret: $11 * 8 = 88$ bit (*plusz meta adatok*)

Példa 2

- Tömörítendő szöveg: ABRAKADABRAABRAABRAABRAABRAKADABRA

1. $\Sigma = \{A, B, R, K, D\}$

2. $d = 5$

3. $n = 33$

4. $L = \lceil \log_2 5 \rceil = 3$

5.

Karakter	Kód
A	000
B	001
D	010
K	011
R	100

- Tömörített szöveg mérete: $n * L = 33 \cdot 3 = 99$ bit
- Kódtábla mérete: $d * 8 + d * 3 = 5 * 8 + 5 * 3 = 40 + 15 = 55$
- Teljes tömörített méret: $99 + 55 = 154$ bit (plusz meta adatok)
- Eredeti méret: $33 * 8 = 264$ bit (plusz meta adatok)

Előnye

- Egyszerű
- Az eredeti adat teljesen visszaállítható, semmilyen információ nem vesz el
- minden kódszó fix hosszúságú → a bitsorozatot egyszerűen szakaszokra lehet bontani
- Nem szükséges bonyolult fa vagy keresés (ellenértében pl. a Huffman-kóddal)

Hátránya

- A gyakori karakterek ugyanannyi bitet kapnak, mint a ritkán előfordulók, ezért a tömörítési arány sokkal rosszabb, mint változó hosszúságú kódoknál (pl. Huffman)
- A fájlban a kódtáblát is el kell menteni, ami rövid szövegeknél nagy többletet jelenthet.
- Ha az ábécé túl nagy (d nagy), akkor a szükséges kódszóhossz ($\lceil \log_2 d \rceil$) is megnő, és a tömörítés értelme elvész
- Egy 90%-ban „A”-ból álló szöveg ugyanannyiba kerül, mint egy teljesen vegyes szöveg (ugyanakkora ábécé mellett)

Huffman-kód

A Huffman-kódolás egy olyan veszteségmentes tömörítési eljárás, amelyben a gyakrabban előforduló karakterek rövidebb, a ritkábbak hosszabb kódszót kapnak. A kódfát a karakterek gyakorisága alapján építjük, így a teljes kódolt üzenet hossza optimálisan minimális lesz a prefixmentes kódok között.

Tömörítési eljárás

1. Határozzuk meg a karakterekhez tartozó gyakoriságokat
2. Hozzunk létre minden karakterből egy fát a gyakoriság, mint kulcs segítségével (tehát itt lesz sok-sok 1 csúcsból álló fánk)
3. Tegyük be az összes így kapott fát, egy min-prioritásos sorba (tehát itt a kis fák egy csúcsként fognak viselkedni)
4. Vegyünk ki két csúcsot (fát) a sorból
5. Készítsünk egy szülőt a két fa gyökérülcsűcsának, ahol a kulcs a két fa gyökérülcsűcsében lévő kulcs összege lesz, továbbá a bal élt címkézzük „0”-val, a jobb élt pedig „1”-el
6. Az így 2 fából és egy harmadik csúcsból alkotott fát tegyük vissza a min-prioritásos sorba
7. Ismételjük meg az egészet a 4. lépéstől, ameddig a min-prioritásos sorban legalább 2 elem van
8. Ha szerencsénk van, itt már csak egy fa szerepel a sorban, ezt nevezzük kódfának
9. A kódfában balra vagy jobbra egészen a levél haladva ki tudjuk olvasni az adott karakterhez (ami ugye a levélben van) tartozó kódszót az élek címkéiről
10. Építsük fel a kódtáblázatot a 9. pont alapján
11. A kimeneti fájl tartalmazza:
 - a. a kódtáblázatot
 - b. a tömörített kód sorozatot

Kitömörítési eljárás

A tömörített szöveget elkezdjük olvasni, és ha találunk olyan sorozatot ami a kódtáblában szerepel, akkor az a sorozat a kódtáblában lévő karakterek fog megfelelni

Példa 1

- Tömörítendő szöveg: AZABBRAKADABRAA
- $\Sigma = \{A, Z, B, R, K, D\}$

Karakter	Előfordulás
A	7
B	3
D	1
K	1
R	2
Z	1

1. $< 1,D \ 1,K \ 1,Z \ 2,R \ 3,B \ 7,A >$

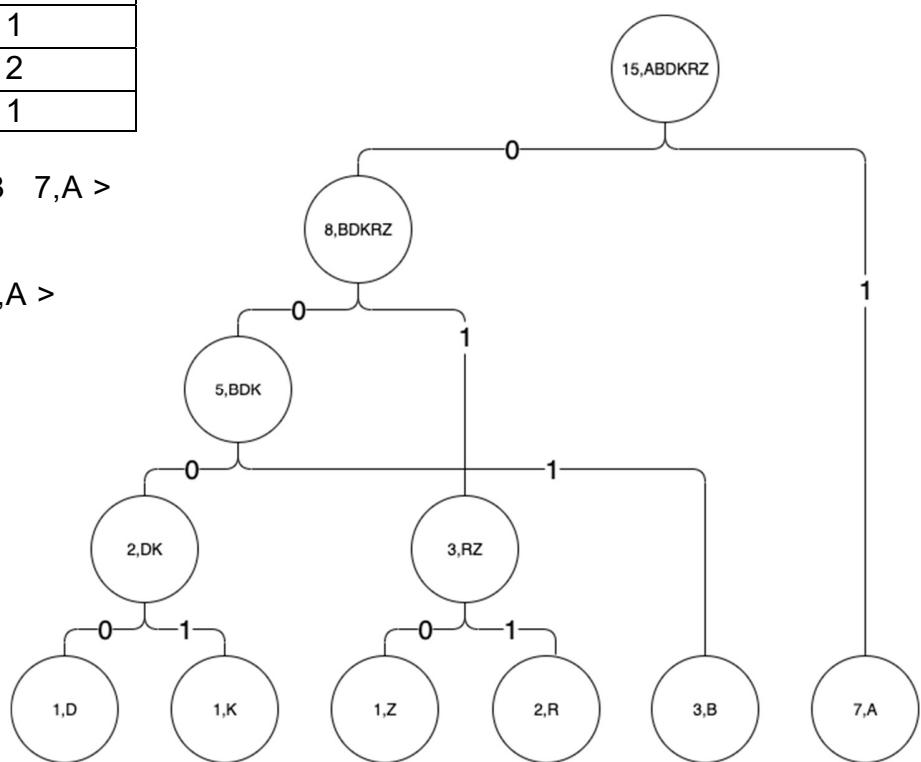
2. $< 1,Z \ 2,DK \ 2,R \ 3,B \ 7,A >$

3. $< 2,DK \ 3,B \ 3RZ \ 7,A >$

4. $< 3RZ \ 5,BDK \ 7,A >$

5. $< 7,A \ 8,BDKRZ >$

6. $< 15,ABDKRZ >$



Karakter	Kód	Előfordulás
A	1	7
B	001	3
D	0000	1
K	0001	1
R	011	2
Z	010	1

• 1 010 1 001 001 011 1 0001 1 0000 1 001 011 1 1
 A Z A B B R A K A D A B R A A

- Tömörített szöveg mérete: $7*1 + 3*3 + 1*4 + 1*4 + 2*3 + 1*3 = 33$ bit
- Kódtábla mérete: $6*8 + 1 + 3 + 4 + 4 + 3 + 3 = 66$ bit
- Teljes tömörített méret: $33 + 66 = 99$ bit (*plusz meta adatok*)
- Eredeti méret: $15 * 8 = 120$ bit (*plusz meta adatok*)

Példa 2

- Tömörítendő szöveg:
AZABBRAKADABRAAAZABBRAKADABRAAAZABBRAKADABRAA
- $\Sigma = \{A, Z, B, R, K, D\}$

Karakter	Előfordulás
A	21
B	9
D	3
K	3
R	6
Z	3

7. $< 3,D \ 3,K \ 3,Z \ 6,R \ 9,B \ 21,A >$

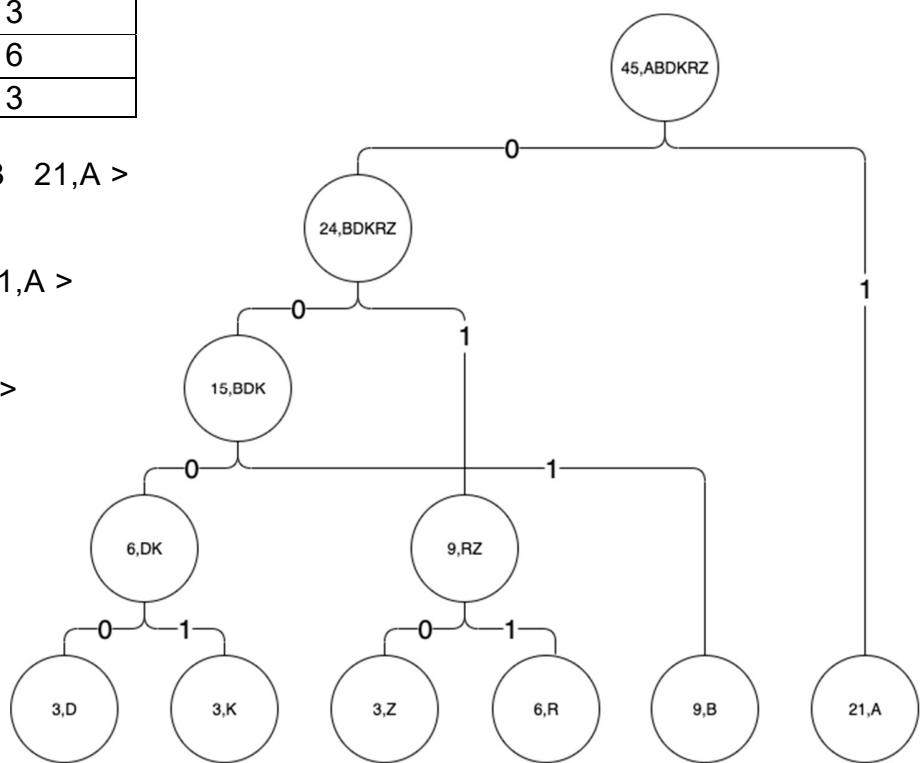
8. $< 3,Z \ 6,DK \ 6,R \ 9,B \ 21,A >$

9. $< 6,DK \ 9,B \ 9,RZ \ 21,A >$

10. $< 9,RZ \ 15,BDK \ 21,A >$

11. $< 21,A \ 24,BDKRZ >$

12. $< 45,ABDKRZ >$



Karakter	Kód	Előfordulás
A	1	21
B	001	9
D	0000	3
K	0001	3
R	011	6
Z	010	3

- Tömörített szöveg mérete: $21*1 + 9*3 + 3*4 + 3*4 + 6*3 + 3*3 = 99$ bit
- Kódtábla mérete: $6*8 + 1 + 3 + 4 + 4 + 3 + 3 = 66$ bit
- Teljes tömörített méret: $99 + 66 = 165$ bit (*plusz meta adatok*)
- Eredeti méret: $45 * 8 = 360$ bit (*plusz meta adatok*)

Előnye

- Mindig a lehető legrövidebb átlagos kódhosszt adja meg a karaktergyakoriságok alapján
- Az eredeti adat teljesen visszaállítható, semmilyen információ nem vesz el
- Minél nagyobb az eltérés a gyakori és ritka karakterek előfordulásában, annál hatékonyabb

Hátránya

- A dekódoláshoz ismerni kell a kódfát vagy a kódtáblát, ami plusz helyet foglal
- A kódszavak változó hosszúságúak, ezért nem lehet egyszerűen szakaszokra vágni a bitsorozatot

LZW (Lempel–Ziv–Welch) tömörítés

Az LZW tömörítés egy veszteségmentes adat-tömörítési algoritmus. Az algoritmus célja, hogy ismétlődő minták és szimbólumok hatékony kódolásával csökkentse az adatmennyiséget.

Az LZW tömörítés során:

- Nincs szükség előzetes statisztikai elemzésre (ellenértben a Huffman-kódolással)
- A kódtáblázat a feldolgozás közben épül fel

Tömörítési eljárás

1. Vegyük fel a kódtáblázatba az összes karaktert
2. Olvassunk be egy karaktert
 - a. Ha nem tudtunk beolvasni (elfogyott a bemenet), írjuk ki az előző „szó” kódját
 - b. Végeztünk
 - c. Egyébként menjünk tovább a 3. lépévre
3. Konkatenáljuk az előző és a beolvasott „karaktereket”
 - a. Ha a konkatenált „szó” benne van a kódtáblázatban
 - i. Akkor az előző „szó” legyen a konkatenált „szó”
 - b. Ha a konkatenált „szó” nincs benne a szótárban
 - i. Vegyük fel a konkatenált „szót” a kódtáblázatba
 - ii. Írjuk ki az előző „szó” kódját
 - iii. Az előző „szó” legyen a beolvasott „szó”
4. Vissza a 2. lépéstre

```
1  inicializál()
2  előző := ""
3  beolvasott := új_karakter
4  if beolvasott = null then
5      kiír(előző)
6      exit()
7  konkatenált := előző + beolvasott
8  ismert := konkatenált ∈ kódtáblázat
9  if ismert then
10     előző := konkatenált
11     jump(line_3)
12 else
13     kódtáblázat.add(konkatenált)
14     kiír(előző)
15     előző := beolvasott
16     jump(line_3)
```

Példa

Tömörítendő szöveg: ABAB ABAA CAAA AAAA A (*space nélkül*)

karakter / szó	kód	lépés	előző	beolvasott	konkatenált	ismert	output
A	1	1	-	A	-	1	-
B	2	2	A	B	AB	0	1
C	3	3	B	A	BA	0	2
AB	4	4	A	B	AB	1	-
BA	5	5	AB	A	ABA	0	4
ABA	6	6	A	B	AB	1	-
ABAA	7	7	AB	A	ABA	1	-
AC	8	8	ABA	A	ABAA	0	6
CA	9	9	A	C	AC	0	1
AA	10	10	C	A	CA	0	3
AAA	11	11	A	A	AA	0	1
AAAA	12	12	A	A	AA	1	-
		13	AA	A	AAA	0	10
		14	A	A	AA	1	-
		15	AA	A	AAA	1	-
		16	AAA	A	AAAA	0	11
		17	A	A	AA	1	-
		18	AA	-	-	-	10

- Tömörített szöveg: 1,2,4,6,1,3,1,10,11,10
- Ellenőrzés (*nem kitömörítés*): A,B,AB,ABA,A,C,A,AA,AAA,AA = ABAB ABAA CAAA AAAA A
- Megjegyzés:
 - Az első lépés kicsit értelemszerűen eltérő
 - A lépés oszlop nem vonatkozik a kódtáblára
 - A lépésszámokat és a beolvasott karaktereket érdemes előre felvenni a táblázatba, mivel bemenettől függően ezek előre meghatározhatók (így könnyebb nem belebonyolódni)
 - Gyakorlásra: https://denvaar.dev/playground/lzw_encode.html
 - A weboldalon már van egy alapértelmezett kódtábla, amiben az ascii karakterekhez a hozzájuk tartozó integert rendeli kódként, tehát pl. az „A” kódja 65, a „B” kódja 66, a „C” kódja 67, ...
 - Az új „szavak” kódja 257-től kezdődik

Kitömörítési eljárás

1. Vegyük fel a kódtáblázatba az összes karaktert
2. Olvassunk be egy kódot
 - a. Ha nem tudtunk beolvasni (elfogyott a bemenet) végeztünk
 - b. Egyébként menjünk tovább
 - c. Ha a beolvasott kód benne van a kódtáblázatban
 - i. Vegyük ki a hozzá tartozó karakterláncot
 - ii. Írjuk ki a karakterláncot
 - iii. Vegyük fel a kódtáblázatba: előző karakterlánc + első karaktere a jelenlegi karakterláncnak
 - d. Ha a beolvasott kód NINCS benne van a kódtáblázatban (*speciális eset*)
 - i. Vegyük az előző karakterlánc + előző karakterlánc első karakterének a konkatenációját
 - ii. Vegyük fel a konkatenált „szót” a szótárba
 - iii. Írjuk ki a konkatenált „szót”
3. Legyen az előző karakterlánc a jelenlegi karakterlánc
4. Vissza a 2. lépéstre

1	inicializ()
2	előző := ""
3	beolvasott := új_kód
4	if beolvasott is null then
5	exit()
6	if beolvasott in kódtáblázat then
7	jelenlegi := kódtáblázat[beolvasott]
8	else
9	jelenlegi := előző + előző[0]
10	kiír(jelenlegi)
11	kódtáblázat.add(previous + jelenlegi[0])
12	előző := jelenlegi

Példa

Kitömörítendő szöveg: 1,2,4,6,1,3,1,10,11,10

karakter / szó	kód	lépés	beolvasott	előző	jelenlegi	új szó	output
A	1	1	1	-	A	-	A
B	2	2	2	A	B	AB	B
C	3	3	4	B	AB	BA	AB
AB	4	4	6	AB	ABA	ABA	ABA
BA	5	5	1	ABA	A	ABAA	A
ABA	6	6	3	A	C	AC	C
ABAA	7	7	1	C	A	CA	A
AC	8	8	10	A	AA	AA	AA
CA	9	9	11	AA	AAA	AAA	AAA
AA	10	10	10	AAA	AA	AAAA	AA
AAA	11						
AAAA	12						

- Output: A B AB ABA A C A AA AAA AA = ABAB ABAA CAAA AAAA A
- Megjegyzés:
 - Az első lépés kicsit értelemszerűen eltérő
 - A lépés oszlop nem vonatkozik a kódtáblára
 - A lépésszámokat és a beolvasott karaktereket érdemes előre felvenni a táblázatba, mivel bemenettől függően ezek előre meghatározhatók (így könnyebb nem belebonyolódni)
 - Gyakorlásra: https://denvaar.dev/playground/lzw_decode.html
 - A weboldalon már van egy alapértelmezett kódtábla, amiben az ascii karakterekhez a hozzájuk tartozó integert rendeli kódként, tehát pl. az „A” kódja 65, a „B” kódja 66, a „C” kódja 67, ...
 - Az új „szavak” kódja 257-től kezdődik

Előnye

- Nem kell előre ismerni a szöveg statisztikáját, menet közben építi a szótárat
- Általánosan jó tömörítési arány, különösen akkor, ha sok ismétlődés vagy minta van a szövegen
- Veszeségmentes tömörítés

Hátránya

- rövid vagy nagyon változatos adatknál a tömörítés akár nagyobb is lehet, mint az eredeti
- ha nincs korlátozva, a szótár túl nagyra duzzadhat; ha korlátozva van, akkor kezelni kell a túlcordulást

AVL fák

Az AVL fa egy speciális bináris keresőfa (Binary Search Tree, BST), amelyet önkiegyensúlyozó bináris keresőfának is nevezünk. Lényege, hogy a beszúrások és törlések után automatikusan biztosítja, hogy a fa magassága a lehető legkisebb maradjon.

Az alapelvek:

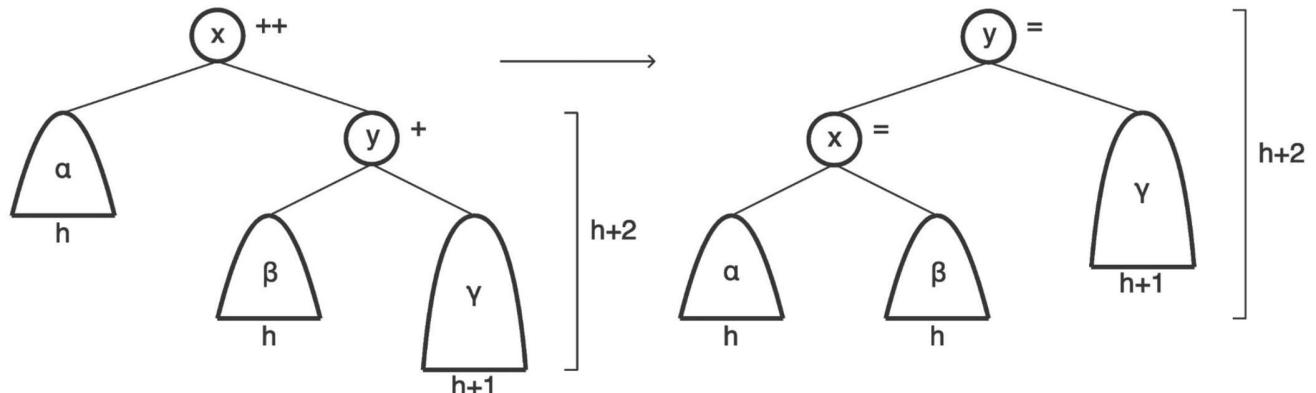
- Ha ez a különbség mindig -1 , 0 vagy $+1$, akkor a fa kiegyensúlyozott
- Ha ennél nagyobb eltérés keletkezne, akkor a fa rotációkkal kiegyenlíti magát

Ennek köszönhetően az AVL fa garantálja, hogy:

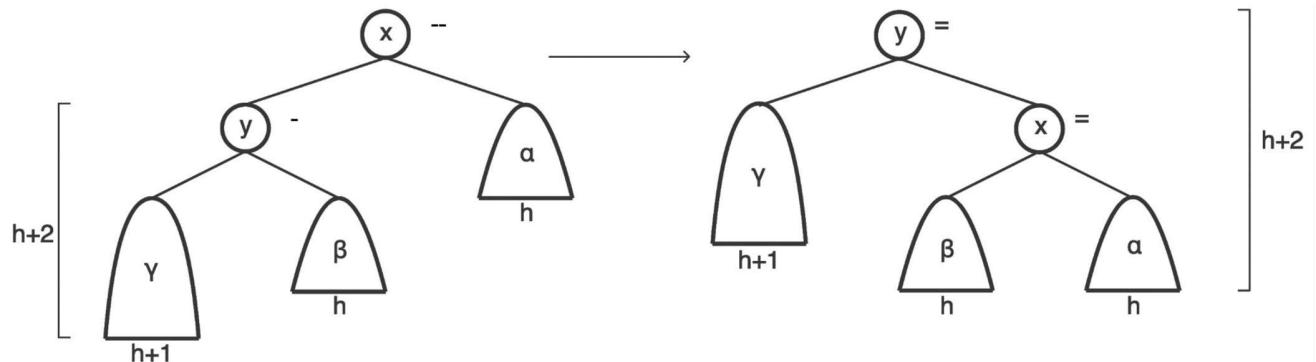
- a keresés, beszúrás és törlés műveletek $O(\log n)$ időben futnak
- a fa magassága minden logaritmikus marad a csomópontok számához képest

Forgatások

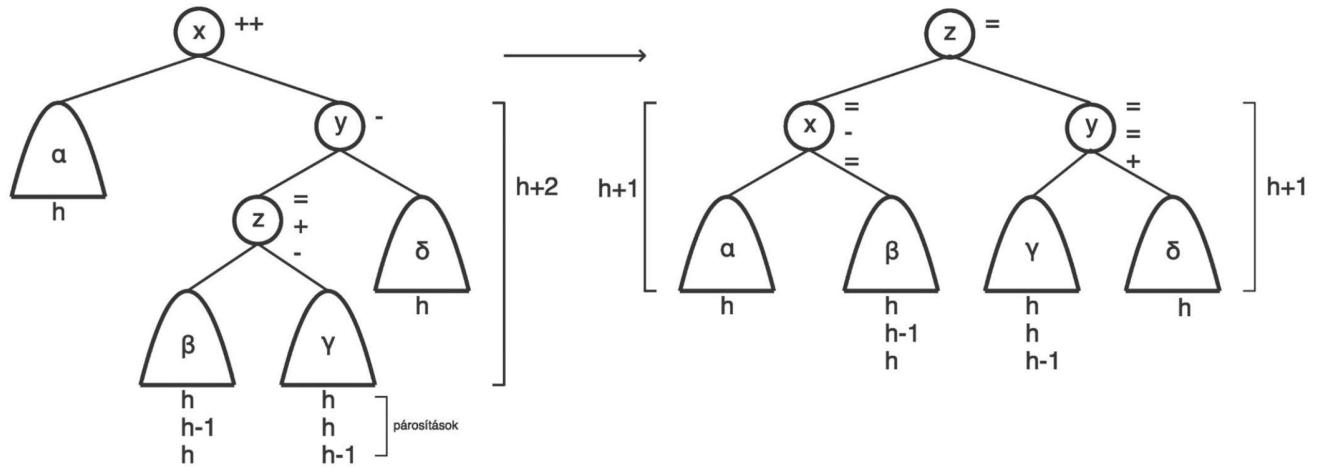
$(++, +)$



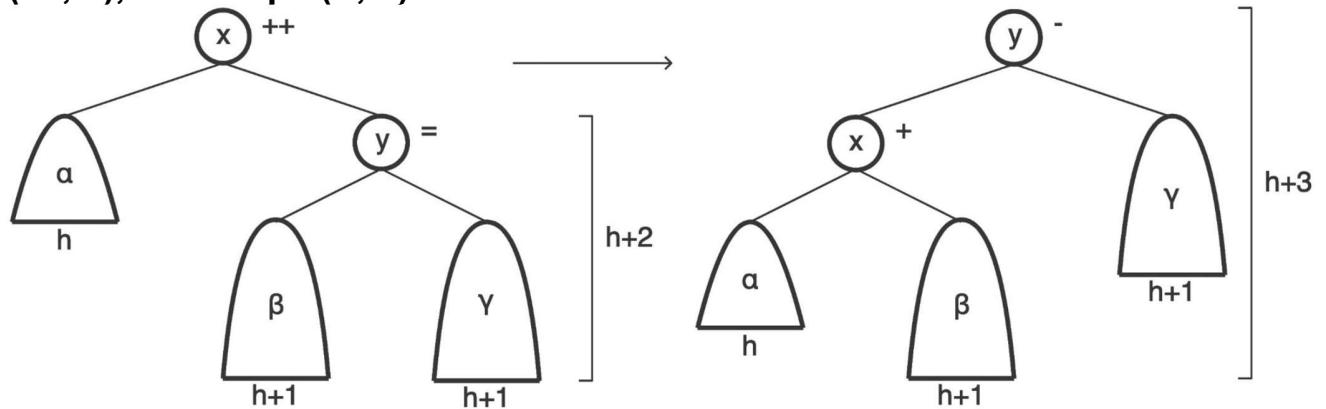
$(--, -)$



(++, -), tükörképe (--, +)



(++, =), tükörképe (--, =)



Beszúrás

1. Megkeressük a kulcs helyét a keresőfa szabály szerint
2. Ha ott üres részfa van, új levélként beszúrjuk
3. Felfelé haladva minden csomópontnál:
 - a. frissítjük az egyensúlyt
 - b. ha az új érték $0 \rightarrow$ megállhatunk (nem nőtt a magasság)
 - c. ha az új érték $\pm 1 \rightarrow$ tovább kell menni (a részfa magasabb lett)
 - d. ha az új érték $\pm 2 \rightarrow$ rotációval kiegyensúlyozzuk, és kész vagyunk

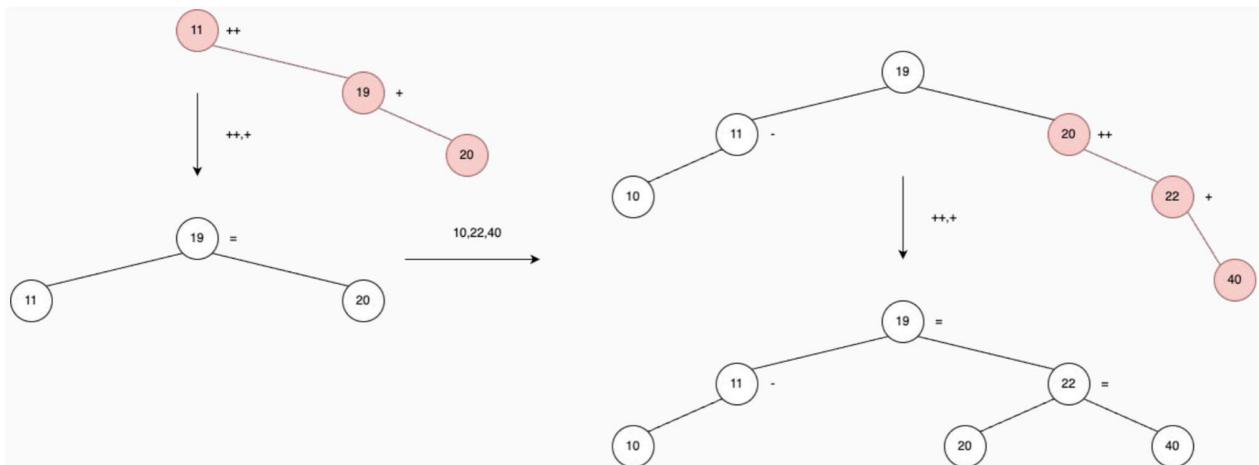
Röviden: beszúrás után alulról felfelé frissítjük a magasságokat és balance faktorokat, és ha kell, egy forgatással helyreállítjuk az egyensúlyt.

Törlés

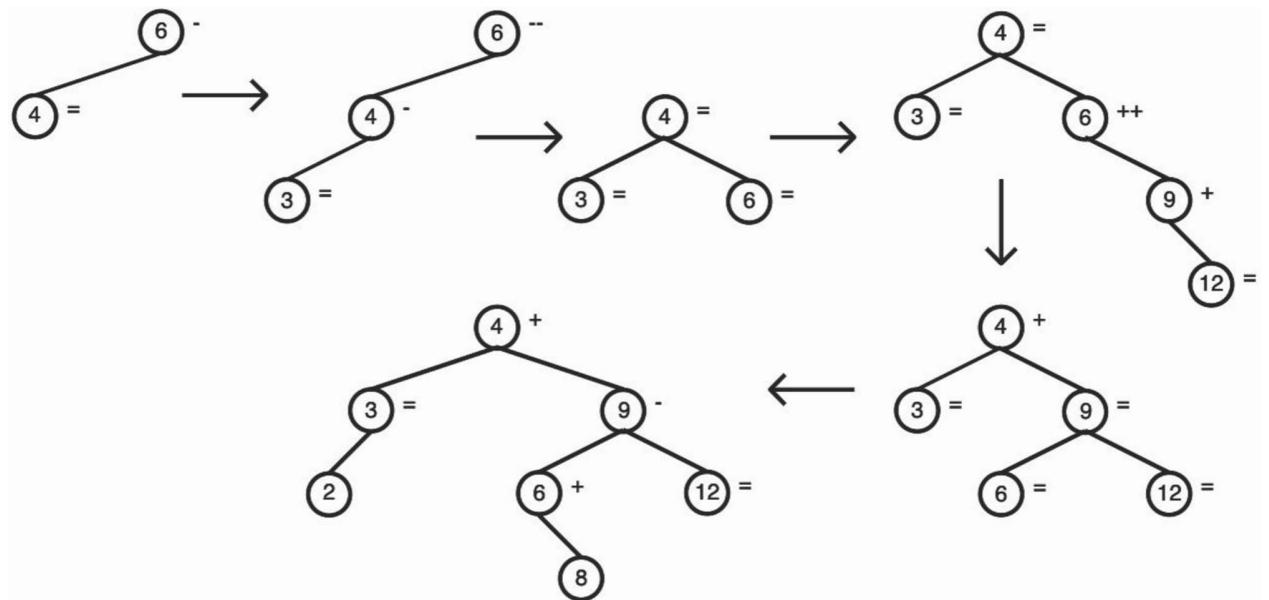
1. Megkeressük a törlendő elemet a keresőfa szabály szerint.
2. Három eset lehet:
 - a. Levél vagy egygyerekes csúcs: egyszerűen eltávolítjuk, a másik részfát (ha van) láncoljuk a helyére
 - b. Kétgyerekes csúcs: megkeressük a jobb részfa minimumát (vagy bal részfa maximumát), azt kiveszük, és az új helyére tesszük (Ez biztosítja, hogy a keresőfa tulajdonság megmaradjon)
3. Az eltávolítás miatt a részfa magassága csökkenhet, ezért felfelé haladva:
 - a. frissítjük az egyensúlyt (újra kell számolni minden node-ra)
 - b. ha az érték $\pm 2 \rightarrow$ rotációval kiegyensúlyozzuk
4. a kiegyensúlyozás után is folytatjuk a szülők felé, mert törlésnél több szinten is szükség lehet rotációra (akár a gyökérig)

Példa

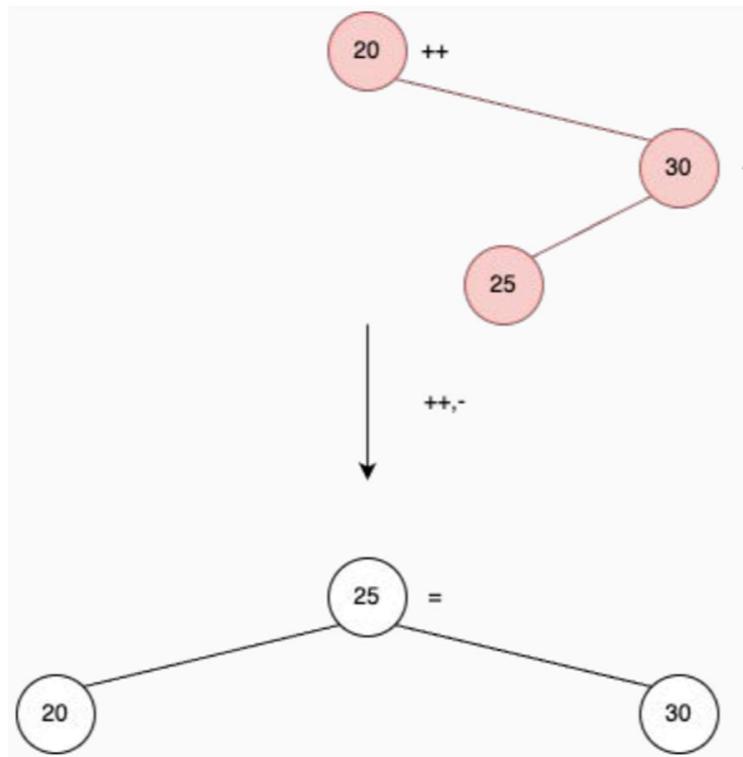
Építsünk fát a következő adatokból: 11, 19, 20, 10, 22, 40



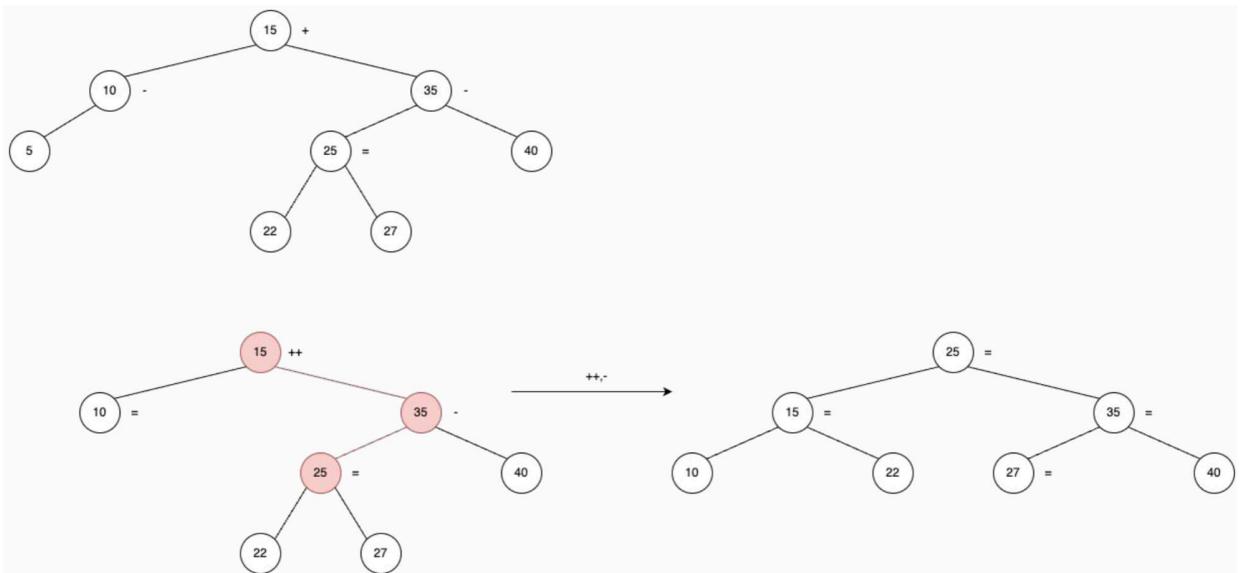
Építsünk fát a következő adatokból: 6, 4, 3, 9, 12, 2, 8



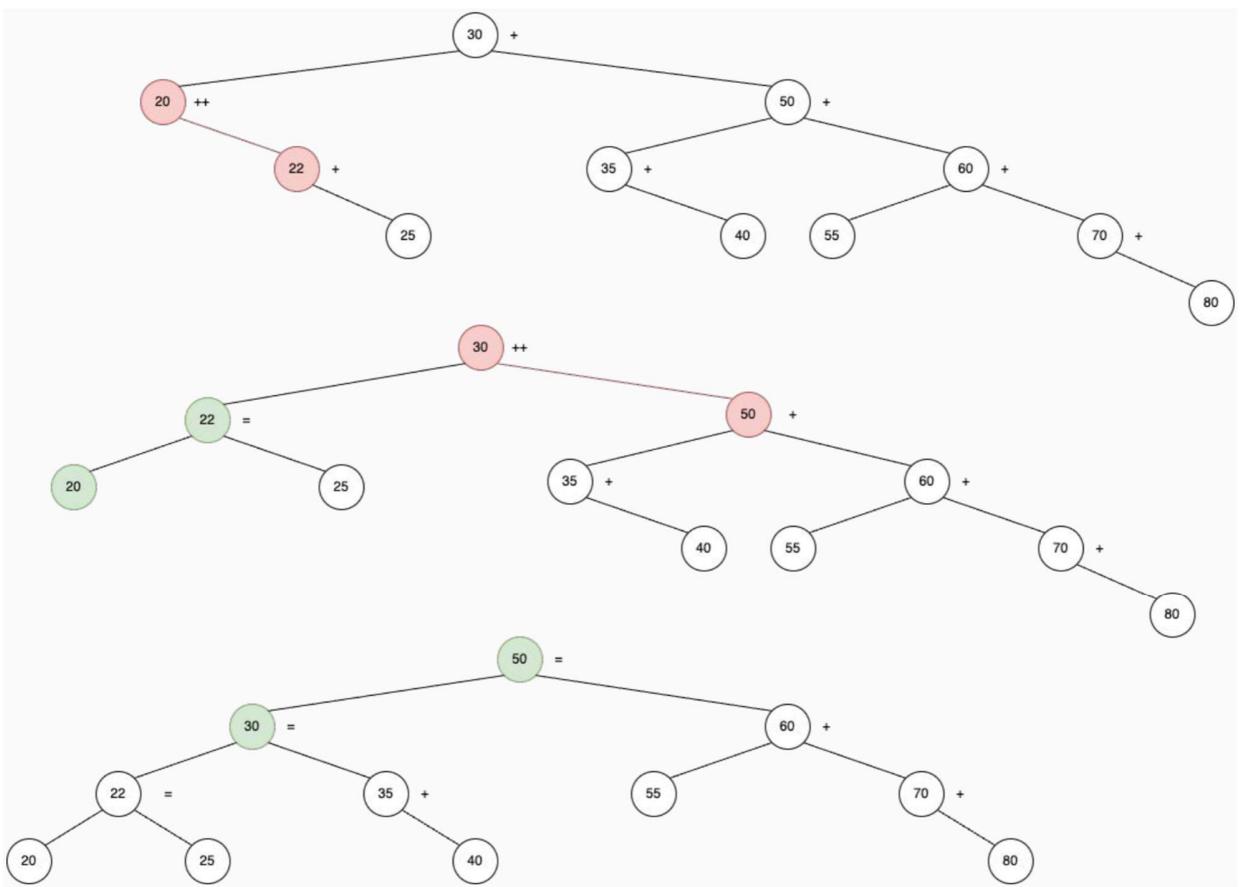
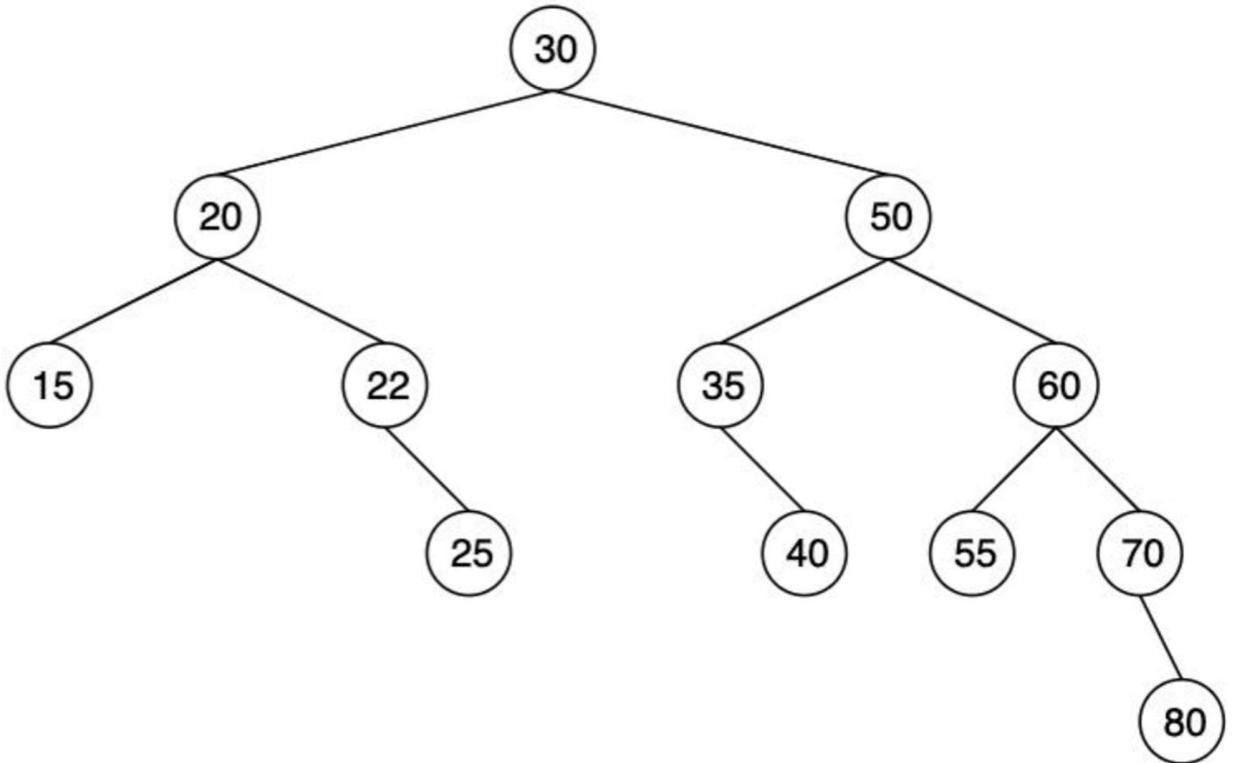
Építsünk fát a következő adatokból: 20, 30, 25



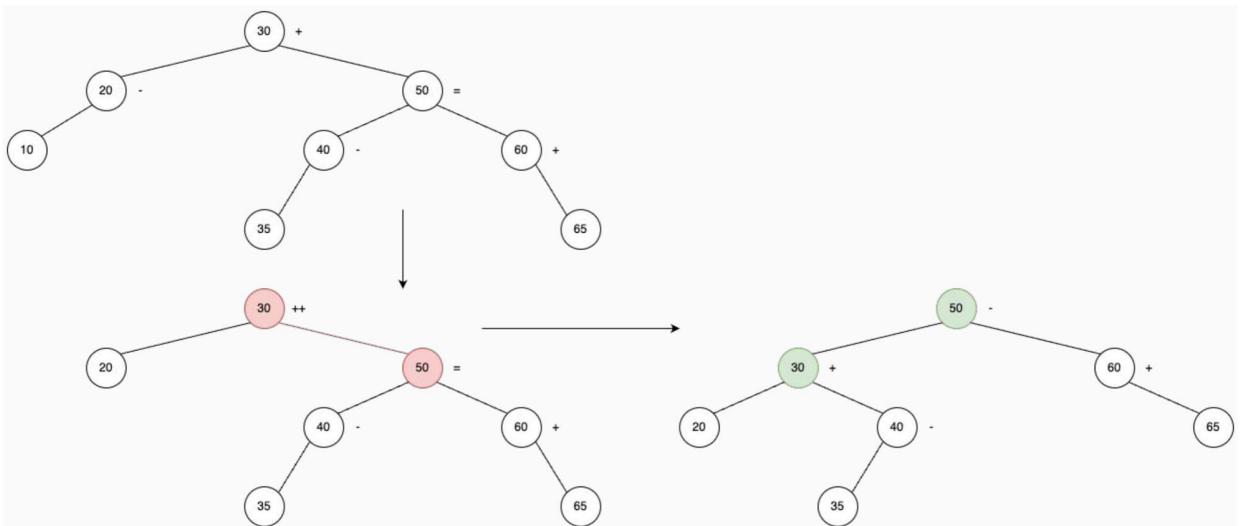
Töröljük az alábbi fából az 5-ös elemet



Töröljük az alábbi fából a 15-ös elemet



Töröljük az alábbi fából a 10-es elemet



Gyakorlásra: <https://www.cs.usfca.edu/~galles/visualization/AVLtree.html> (a jelölés picit más, de a beszúrás, törlés, és forgatások ugyanaz)

B+ fák

A B+ fa egy kiegyensúlyozott, többágú keresőfa (multiway search tree), amelyet elsősorban adatbázisokban és fájlrendszerben használnak hatékony kereséshez, beszúráshoz és törléshez.

Tulajdonságai

- Egy csomópontban több kulcs és több mutató (pointer) lehet
- Gyökér, belső csomópontok, levelek:
 - A belső csomópontok csak kulcsokat és mutatókat tartalmaznak
 - Az adatok (rekordok) mindig a levelekben találhatók
 - A levélszint csomópontjai láncolva vannak, ami lehetővé teszi a hatékony tartománykeresést
- minden levél ugyanazon a szinten helyezkedik el
- Tömbösített diszk-hozzáférésre optimalizált
 - A B+ fa sok kulcsot és gyermeket tartalmaz egy csomópontban
 - Egy csomópont általában pontosan egy diszk-blokkhoz illeszkedik
 - Ezért sok kulcs beolvasása egyetlen diszk-hozzáféréssel lehetséges
 - Az AVL fánál minden csomópont külön blokk lenne, így sokkal több diszk-olvasás kellene

Belső csúcsok tulajdonságai

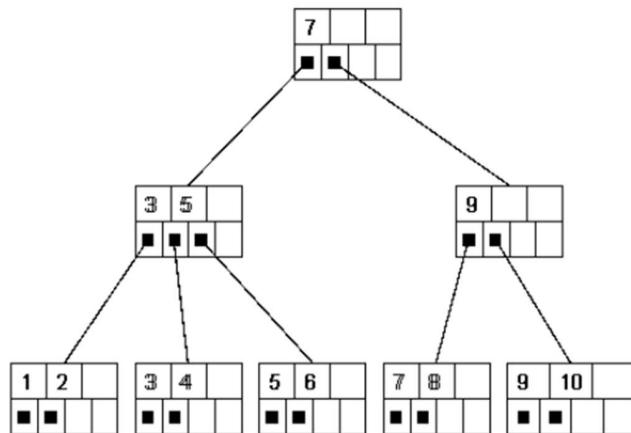
1. minden csúcs legfeljebb d mutatót ($4 \leq d$), és legfeljebb $d-1$ kulcsot tartalmaz. (d : állandó, a B+ fa fokszáma)
2. minden Cs belső csúcsra, ahol k a Cs csúcsban a kulcsok száma: az első gyerekhez tartozó részfában minden kulcs kisebb, mint a Cs első kulcsa; az utolsó gyerekhez tartozó részfában 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 $Cs.kulcs[i-1] \leq r < Cs.kulcs[i]$.
3. A gyökérCsúcsnak legalább két gyereke van (kivéve, ha ez a fa egyetlen csúcsa, következésképpen az egyetlen levele is).
4. minden, a gyökértől különböző belső csúcsnak legalább $\lfloor d/2 \rfloor$ gyereke van.

Levelek tulajdonságai

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

Zárójeles ábrázolása

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



Beszúrás

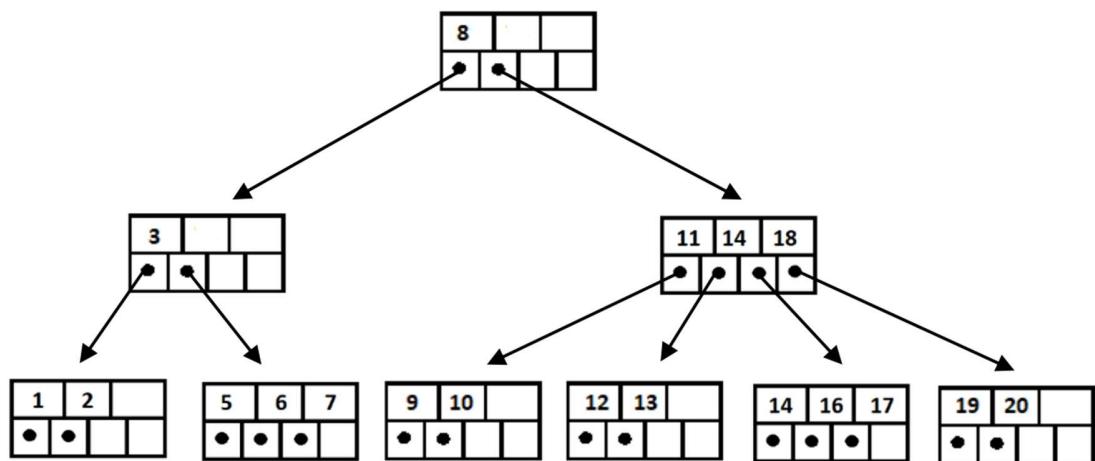
Szabályok a példákon keresztül.

Törlés

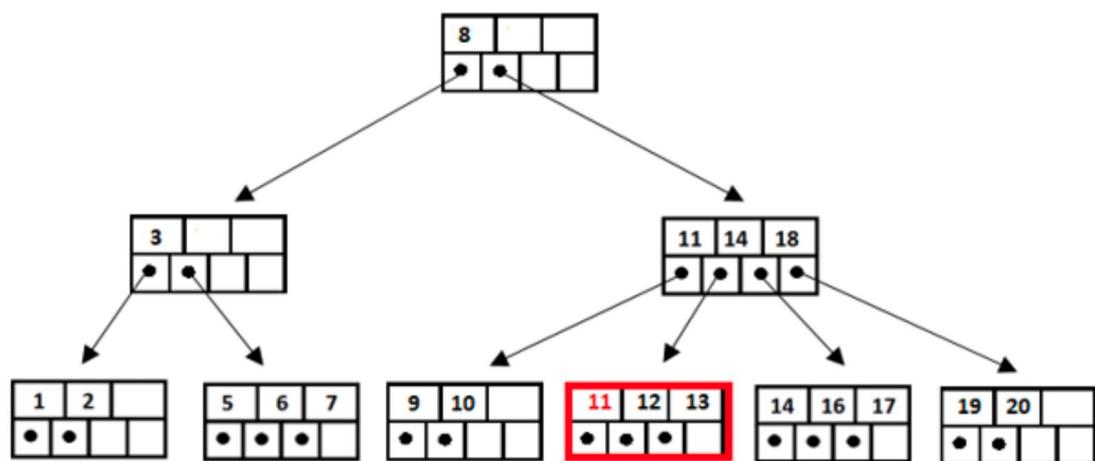
Szabályok a példákon keresztül.

Példa

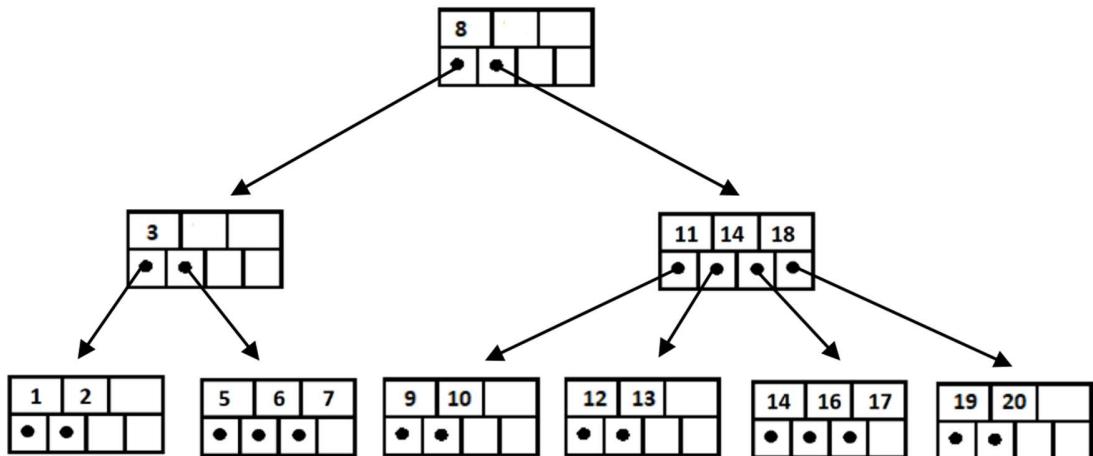
Szűrjuk be a 11-t az alábbi fába!



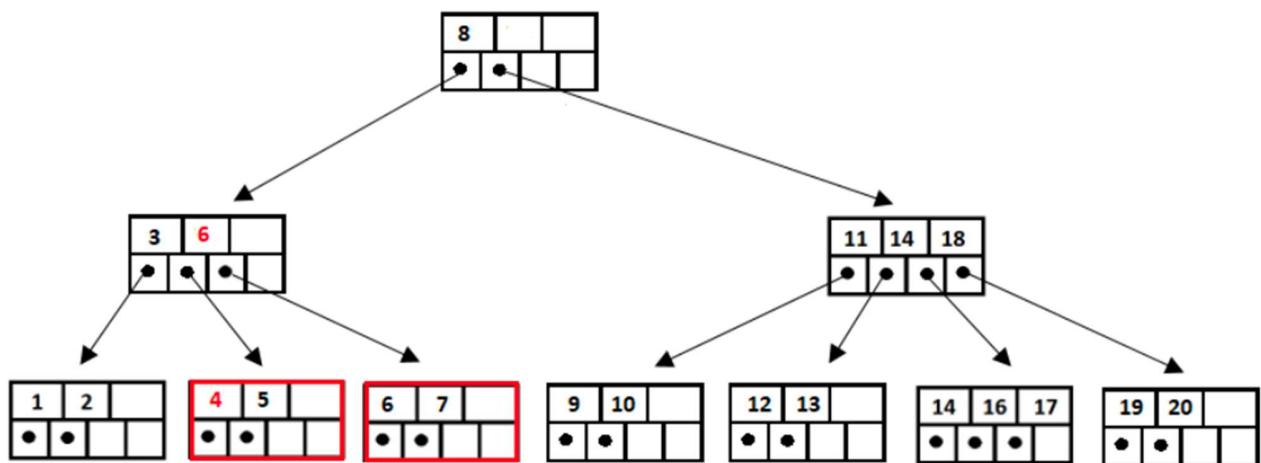
1. Megkeressük a 11 helyét: (12 13 levél)
2. Mivel a levélben nem szerepel a 11-es, és van hely, ezért egyszerűen beszűrjuk



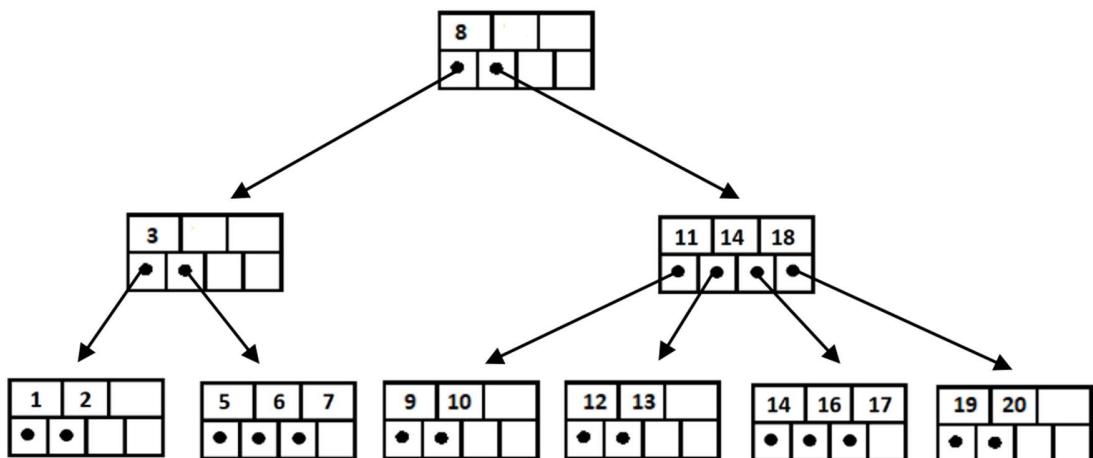
Szúrjuk be a 4-t az alábbi fába!



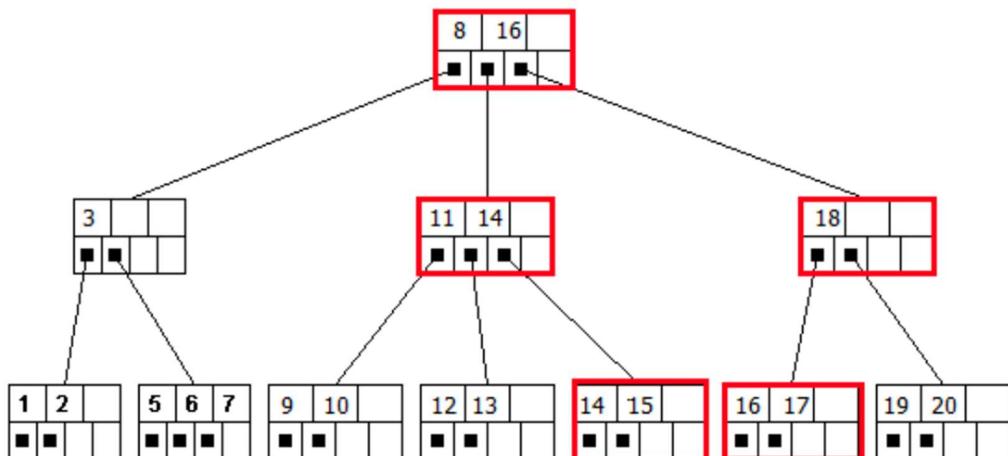
1. Megkeressük a 4-es helyét: (5 6 7) levél
2. Mivel nincs több hely, ezért szétvágjuk és beszúrjuk a 4-est: (4 5) (6 7)
3. Változtatjuk a szülő csúcsot:
 - a. Mivel az nem telített, egy új kulcsot helyezünk el. Az új kulcs minden az új levél első eleme lesz: (3 6)



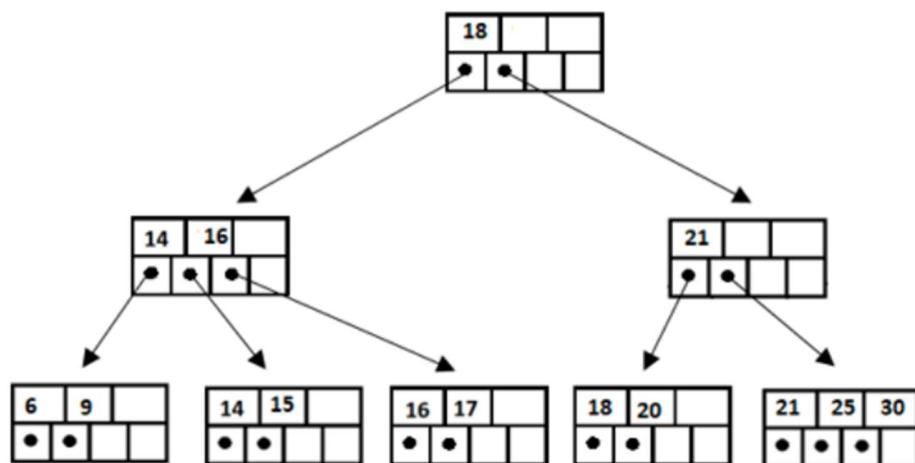
Szűrjuk be a 15-t az alábbi fába!



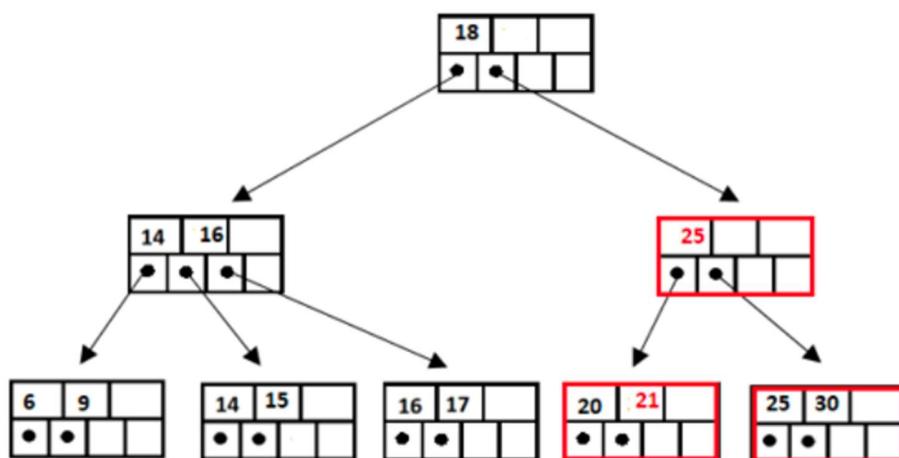
1. Megkeressük a fában a 15 helyét: (14 16 17) levél
2. Mivel ez a levél már telített, szétvágjuk: (14 15) (16 17)
3. Változtatjuk a szülő (11 14 18) csúcsot:
 - a. Mivel az is telített, azt a csúcsot is kettévágjuk.
 - b. Mivel ennek a szülője már nem telített, egy új kulcsot és mutatót helyezünk el benne, ez az adott részfa legkisebb eleme lesz



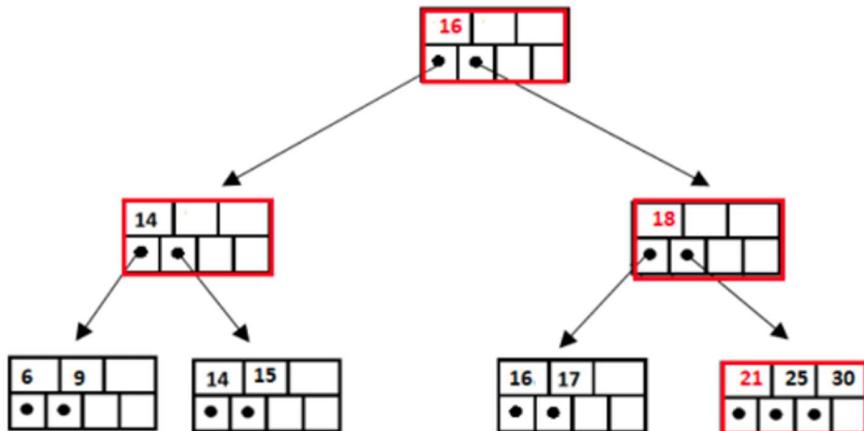
Töröljük az alábbi fából a 18-t, majd a 20-t, majd a 6-t!



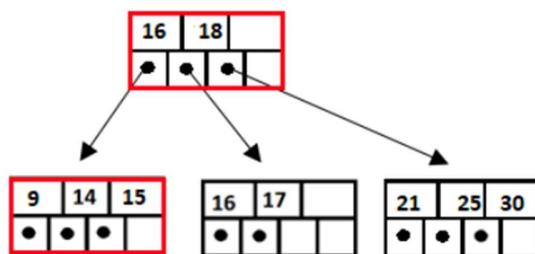
1. Megkeressük a fában a 18 helyét: (18 20) levél
2. Mivel 18 törlésével a levél egyedül marad, a testvérétől kap kulcsot: (20 21) (25 30)
3. Változtatjuk a szülő csúcsot:
 - a. A 21 kulcs törlődik és helyette a 2. gyerekének (25 30) az első eleme lesz: 25



1. Megkeressük a fában a 20 helyét: (20 21) levél
1. Mivel 20 törlésével a levél egyedül marad, 21 átadódik a jobb testvérenek: (21 25 30)
2. Így az említett levél szülőjének (25) is csak egy gyereke lenne, ezért a baltestvérétől fog kapni egy gyermeket: (16 17)
3. A hasítókulcsok átíródnak 25->18 és 18->16
4. *Megjegyzés: Ha a jobb belső csúcs kap egy levelet a bal belső csúcstól, akkor a bal belső csúcs utolsó csúcsa lesz a gyökér első kulcsa, és a gyökér utolsó kulcsa lesz a jobb belső csúcs első kulcsa. (Mondhatni forognak egyet a kulcsok)*



1. Megkeressük a fában a 6 helyét: (6 9) levél
2. Mivel 6 törlésével a levél egyedül marad, 9 átadódik a jobb testvérenek: (9 14 15)
3. Mivel az említett levél szülőjének (14) is csak egy gyereke lesz, a jobbtestvére pedig nem tud átadni gyereket, az említett csúcsot összevonjuk a jobbtestvérével
4. A gyökérnek egy gyermeke marad, így azt törlődik



Megjegyzés:

1. A valóságban kulcsot és levelet is tudunk balról illetve jobbról kölcsönözni, de az egyszerűség kedvéért mi próbálunk balra tömöríteni, így ha tudunk, akkor kulcsot jobbról, ha pedig már nincs kulcs, akkor levelet balról kölcsönözni.
2. Beszúrnál minden igaz, hogy a belső csúcsban a kulcs minden egyenlő lesz annak a levélnek az első kulcsával amire a „nagyobb egyenlő mutatója” mutat, ha a fából törlünk akkor ez már nem feltétlen lesz igaz.
3. Ha a gyökérnek csak 1 gyereke van akkor ő egyszerűen megszűnik létezni.
4. Egy másik jegyzet: <https://people.inf.elte.hu/pgm6rw/Algo2/Trees/bTree/index.html>
5. B+ fának több implementációja is van, amik kisebb nagyobb pontokban eltérhetnek. Amit mi tanulunk az hatékonyságát tekintve optimálisabb, viszont nehezebb is, mint amit az interneten találhattok, próbáljuk meg azokat a szabályokat alkalmazni amit itt tanulunk. (A fenti jegyzet erre megfelel)
6. A tanult szabályokat követve törlés során lehet, hogy olyan ponthoz érünk, ahol a tanult szabályok alapján nem lehet pontosan eldönteni mi a helyes lépés. Ilyen feladat nem lesz zh-n.

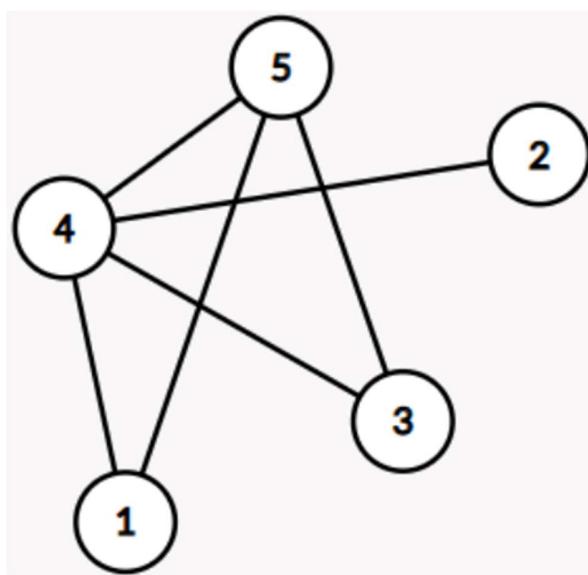
Gráfok

Gráf alatt egy $G = (V, E)$ rendezett párost értünk, ahol
 V a csúcsok vagy pontok (vertices) tetszőleges, véges halmaza,
 $E \subseteq V \times V \setminus \{(u, u) : u \in V\}$ pedig az élek (edges) halmaza.
Ha $V = \{ \}$, akkor a gráf üres

Típusai

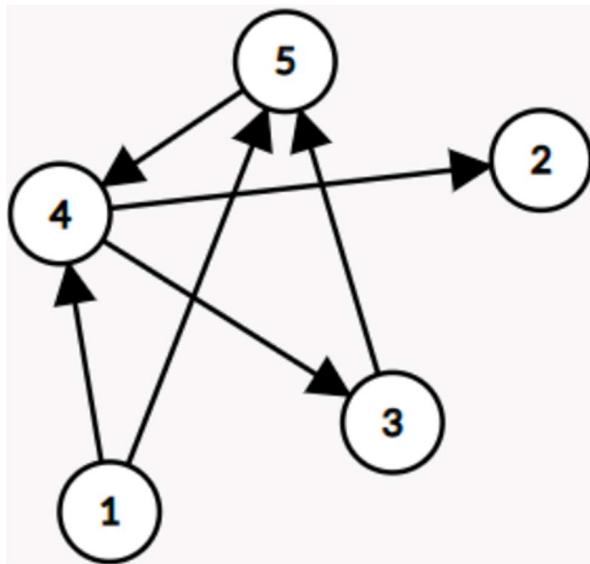
Irányítatlan gráf

- A csúcsok (pontok) között az élek (vonalak) nincs irányuk
- Ha két csúcsot összeköt egy él, akkor az kapcsolat kétirányú
- Példa: barátsági háló – ha A barátja B-nek, akkor B is barátja A-nak



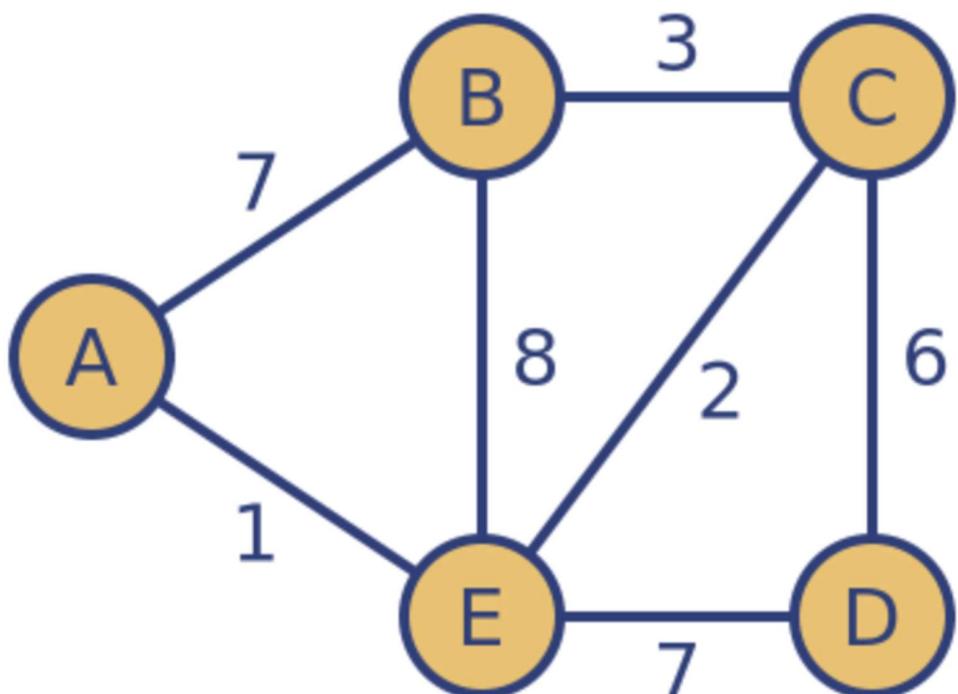
Irányított gráf

- Az éleknek irányuk van, egyik csúcsból a másikba mutatnak
- A kapcsolat egyirányú
- Példa: követési háló a közösségi médiában – ha A követi B-t, attól még B nem feltétlen követi A-t



Súlyozott gráf (irányított vagy irányítatlan is lehet)

- Az élekhez súly (érték, költség, hossz, idő stb.) van rendelve
- A súly jelentése a feladattól függ
- Példa: útvonalhálózat – az él súlya lehet a távolság kilométerben vagy az utazási idő



Definíciók

Út

- A $G = (V, E)$ gráf csúcsainak (V) egy $\langle u_0, u_1, \dots, u_n \rangle$ ($n \in \mathbb{N}$) 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 n , azaz az utat alkotó élek számával egyenlő.

Kör

- A kör olyan út, aminek kezdő és végpontja (csúcsa) azonos, a hossza > 0 , és az élei páronként különbözök.
- Az egyszerű kör olyan kör, aminek csak a kezdő és a végpontja azonos.
- Tetszőleges út akkor tartalmaz kört, ha van olyan részütje, ami kör.

Ritka gráf

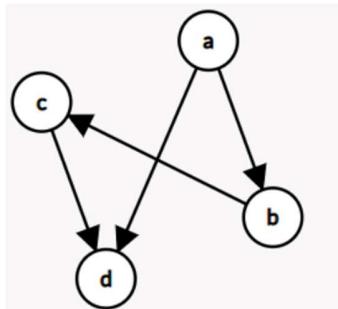
- Egy gráf ritka gráf, ha $|E|$ sokkal kisebb, mint $|V|^2$.

Sűrű gráf

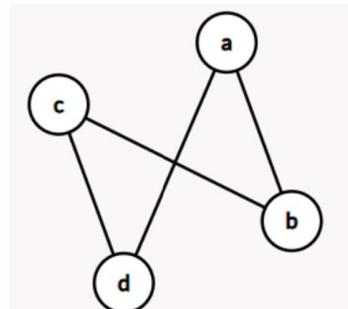
- Egy gráf sűrű gráf, ha $|E|$ megközelíti $|V|^2$ -et.

Ábrázolás

Grafikusan



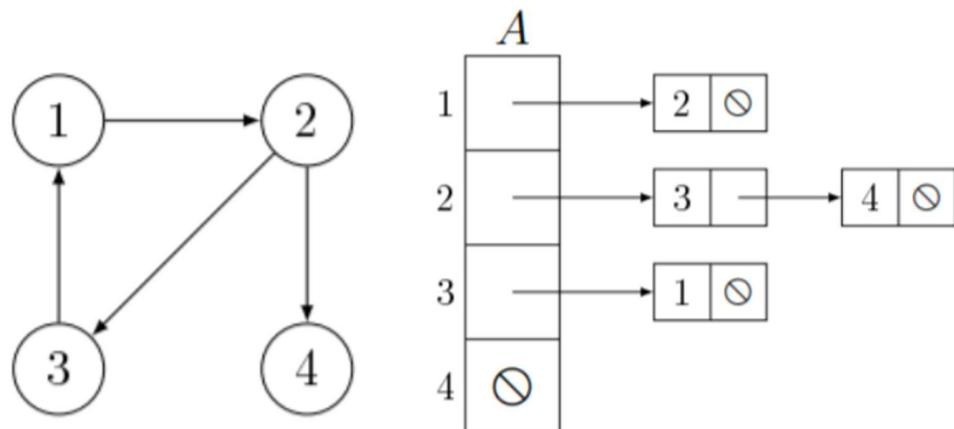
$a \rightarrow b; d$
 $b \rightarrow c$
 $c \rightarrow d$



$a - b; d$
 $b - a; c$
 $c - b; d$
 $d - a; d$

Szomszédsági lista

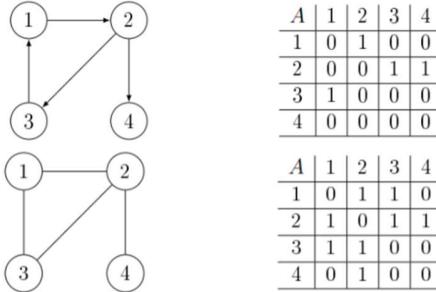
- Az ábrázoláshoz egy tömböt használunk. Ez a tömb $|V|$ darab listából áll, és a tömbben minden csúcshoz egy lista tartozik.
- minden u csúcs esetén a tömbben az ahhoz tartozó szomszédossági lista tartalmazza az összes v ilyen csúcsot, amelyre létezik $(u,v) \in E$ él.
- Azaz a lista elemei u csúcs G-beli szomszédjai. A szomszédossági listákban a csúcsok sorrendje általában tetszőleges.
- Irányítlan és irányított gráfok esetén is alkalmazhatjuk ezt az ábrázolást.
- Az ábrázoláshoz szükséges tárterület $\Theta(V+E)$ minden esetben.
- Ha G irányított gráf, akkor a szomszédsági listák hosszainak összege $|E|$, ugyanis egy (u,v) élt úgy ábrázolunk, hogy v-t felvesszük a megfelelő listába.
- Ha G irányítatlan gráf, akkor a szomszédsági listák hosszainak összege $2|E|$, mivel (u,v) irányítatlan él ábrázolása során u-t betesszük v szomszédsági listájába, és fordítva.



- Az ábrán egy irányított gráf van ábrázolva
- Ha egy ugyanilyen irányítatlan gráfot szeretnénk ábrázolni, akkor
 - Az első listához hozzáadjuk a 3-t
 - A második listához hozzáadjuk az 1-t
 - A harmadik listához hozzáadjuk a 2-t
 - A negyedik listához hozzáadjuk a 2-t

Szomszédsági mátrixos / Csúcsmátrixos

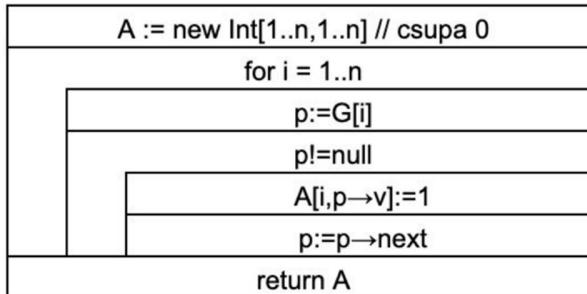
- Feltesszük, hogy a csúcsokat tetszőleges módon megszámozzuk az $1, 2, \dots, |V|$ értékekkel.
- A gráf ábrázolásához használt $A = (a_{ij})$ csúcsmátrix $|V| \times |V|$ méretű, és
 - $a_{ij} = 1$, ha $(i, j) \in E$
 - $a_{ij} = 0$, különben
- Irányítatlan gráf esetén (u, v) és (v, u) ugyanaz az él, 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 csökkenhetjük a szükséges tárhelyet.



Stuktogrammos feladatok

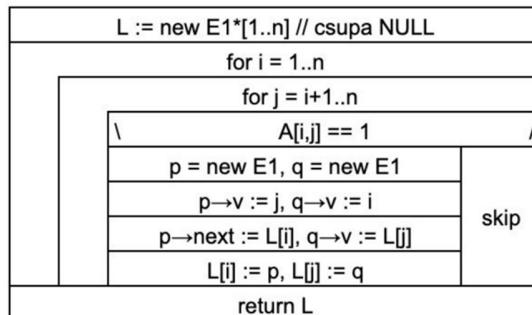
- Adott egy egyszerű (nincs hurok, és párhuzamos él) irányított gráf éllistás ábrázolással, amelynek n db csúcsa van. Transzformáljuk át csúcsmátrixos ábrázolásba!

```
AdjList2AdjMatrix(G:E1*[1..n]):Int[1..n,1..n]
```



- Adott egy egyszerű irányítatlan gráf csúcsmátrixos ábrázolással, amelynek n db csúcsa van. A csúcsmátrixnak csak a felső háromszögmátrixa van kitöltve. Transzformáljuk át éllistás ábrázolásba!

```
AdjMatrix2AdjList(A:Int[1..n,1..n]):E1*[1..n]
```

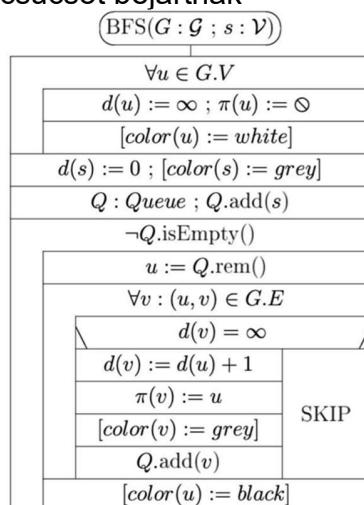


Szélességi gráfbejárás (BFS – Breadth First Search)

A BFS rétegenként (vagyis szélességi sorrendben) járja be a gráfot, azaz először minden olyan csúcsot bejár, amely egy adott távolságra van a kezdőcsúcstól, mielőtt tovább lépne a következő szintre.

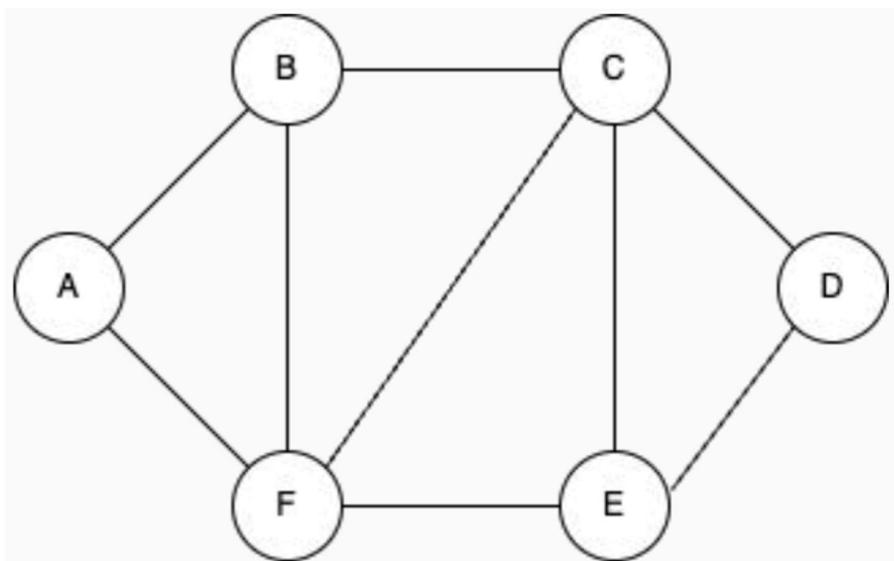
Működési elv

1. Válasszunk egy kezdőcsúcsot
2. Helyezzük a kezdőcsúcsot egy sorba (queue)
3. Amíg a sor nem üres:
 - a. Vegyük ki a sor elején lévő csúcsot
 - b. Látogassuk meg az összes még nem látogatott szomszédját, és tegyük őket a sor végére
 - c. Jelöljük meg a kivett csúcsot bejártnak

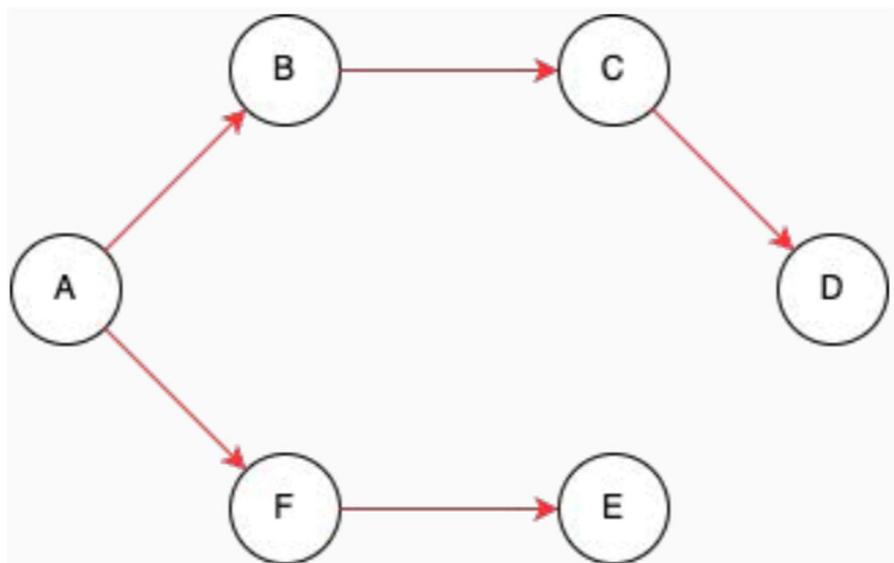


Youtube: [Breadth-first search in 4 minutes](#)

Példa



d változásai						Kiterjesztett csúcs	Q tartalma	Π változásai					
A	B	C	D	E	F			A	B	C	D	E	F
0	inf	inf	inf	inf	inf	-	<A>	/	/	/	/	/	/
1					1	A	<F,B>		A				A
	2					B	<C,F>			B			
		2				F	<E,C>						F
			2			C	<D,E>				C		
				3		E	<D>						
						D	<>						
0	1	2	3	2	1			/	A	B	C	F	A



Stuktogrammos feladatok

1. Adott egy nem súlyozott, éllistás ábrázolással ábrázolt irányítatlan gráf, és két csúcs: s (start) és t (target).

Keresd meg a legrövidebb (él-szám szerinti) utat s és t között!

```
BFS_shortest_path(Graph[1..n], start:V, target:V) -> SingleLinkedList
// Tfh. a node csúcsai objektumok, amiben több adat is szerepelhet
// tehát nem feltétlen egyetlen betűből v. számból állnak
// Nem egy E1* adunk vissza,
// hanem már megcsináltunk egy "E1 típusú lácolt listát",
// és ezt most felhasználjuk
queue := new Queue()
visited := new HashSet()
parent := new HashMap(Graph)
// (a táblázatban ő Π),
// tfh. a konstruktor minden csúcsra beállítja null értékre a szülőt

queue.add(start)
visited.add(start)

while !queue.isEmpty()
    current := queue.pop()

    if current = target // megtaláltuk a targetet, meg kell nézni mi volt az út
        path := SingleLinkedList()
        node := target
        while node is not null
            path.insert_at_front(node)
            node := parent[node] // node-t beállítjuk node-nak a szülőjére
        return path

    L := Graph.get(current)
    // visszatér azoknak a node-oknak a listájával,
    // ahova current-ból el tudunk menni
    // tfh. ez egy copy, és nem referencia (tehát a gráf alapstruktúráját nem változtatjuk)
    while L is not null
        if L -> val is not in visited // L -> val a csúcs, ahova el tudunk menni
            queue.add(L -> val)
            parent[L->val] := current
            visited.add(L -> val)
        L := L -> next

return null
```

2. Adott egy éllistás ábrázolással ábrázolt irányítatlan gráf.

Számold meg, hány összefüggő komponens van benne!

```
Count_Components(Graph[1..n]) -> Int
visited = new HashSet()
count := 0

for i := 1..n
    if Graph[i] is not in visited
        BFS(visited, Graph[i])
        count := count + 1

return count
```

```
BFS(&visited, start:V): // segédfüggvény
queue = new Queue()
queue.add(start)
visited.add(start)

while !queue.isEmpty():
    current := queue.pop()

    L := Graph.get(current)
    while L is not null:
        if L -> val is not in visited:
            queue.add(L -> val)
            visited.add(L -> val)
        L := L -> next
```

Mélységi gráfbejárás (DFS – Depth First Search)

A DFS egy olyan gráfbejáró algoritmus, amely úgy működik, hogy:

- minden lehető legmélyebbre megy,
- és csak akkor lép vissza, ha már nincs további út.

Úgy kell elképzelni, mint amikor egy labirintusban minden kimész egy folyosón a végéig, és ha zsákutca, visszamész az előző elágazásig és megpróbálsz egy másik irányt.

Zsákutcát érhetünk az alatt is, ha a labirintusban egy olyan ponthoz érsz, ahol már voltál. Fizikailag tudnál tovább menni, de felesleges. (ez csak akkor lehetséges ha A-ból nem tudtál továbbmenni B-be, de B-ből tudsz menni A-ba, pl. B-ből egy csúszda vezet A-ba. A csúszdán tudsz lefelé, de nem tudsz felfelé menni, magyarul: ez egy irányított él)

Rekurzívan

$\text{DFS}(G : \mathcal{G})$	$\text{DFSvisit}(G : \mathcal{G} ; u : \mathcal{V} ; \&time : \mathbb{N})$
$\forall u \in G.V$ $\text{color}(u) := \text{white}$ $time := 0$ $\forall r \in G.V$ $\text{color}(r) = \text{white}$ $\pi(r) := \odot$ $\text{DFSvisit}(G, r, time)$	$d(u) := ++time ; \text{color}(u) := \text{grey}$ $\forall v : (u, v) \in G.E$ $\text{color}(v) = \text{white}$ $\pi(v) := u$ $\text{DFSvisit}(G, v, time)$

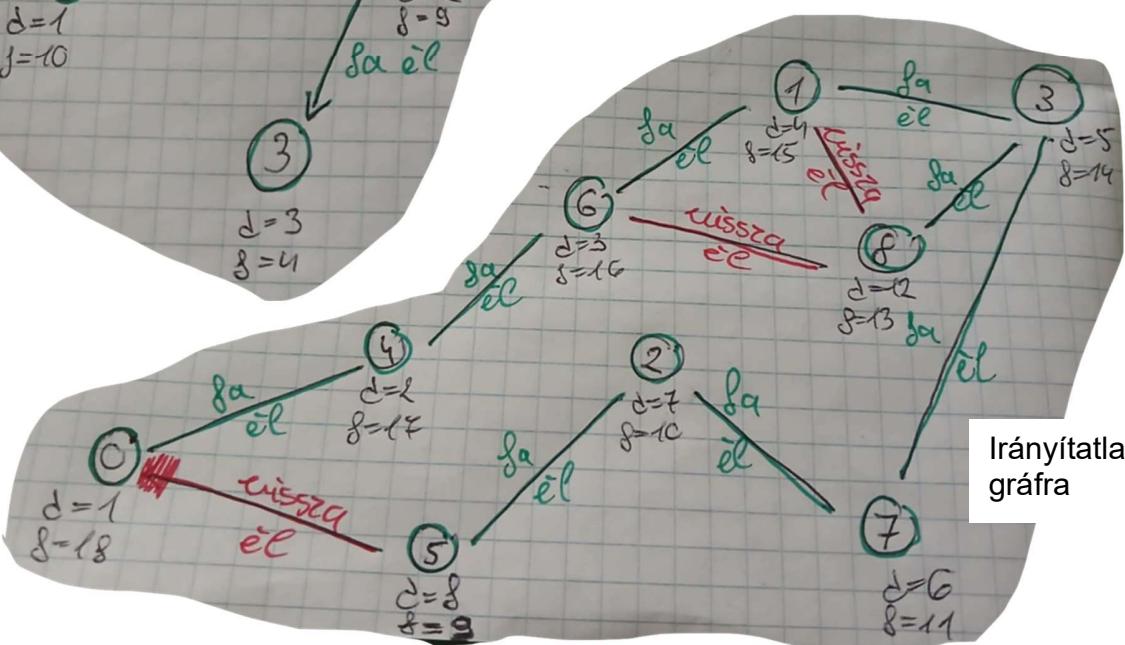
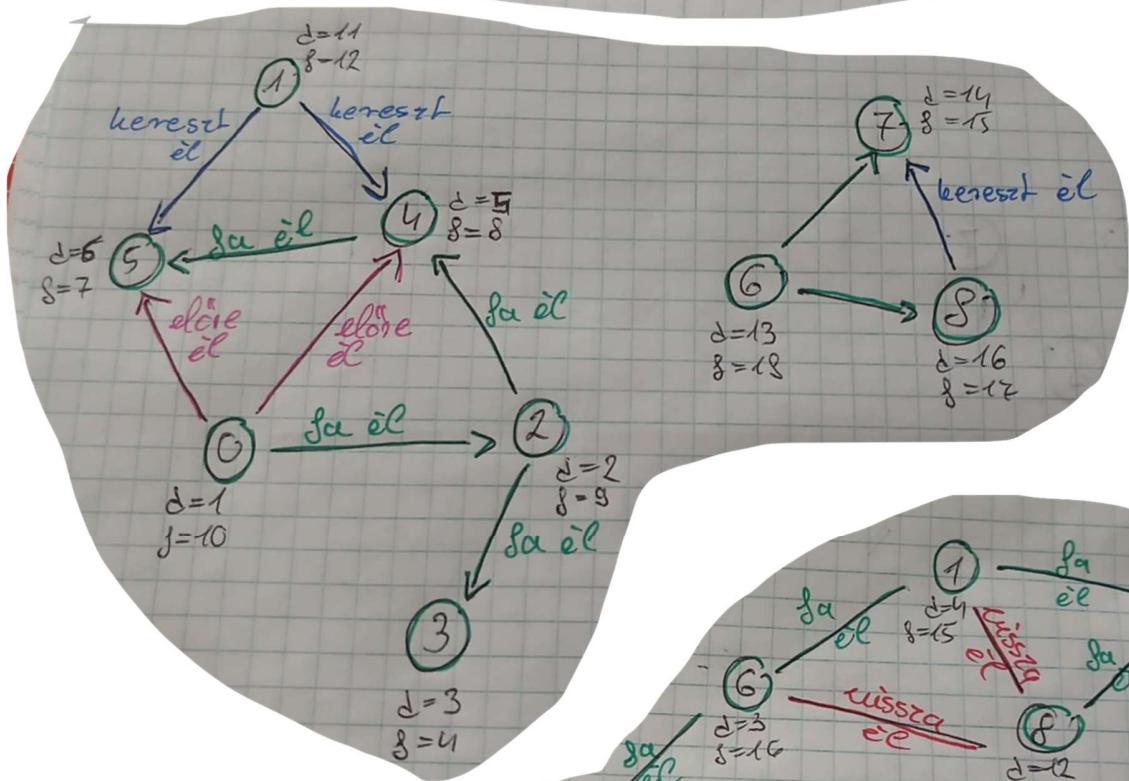
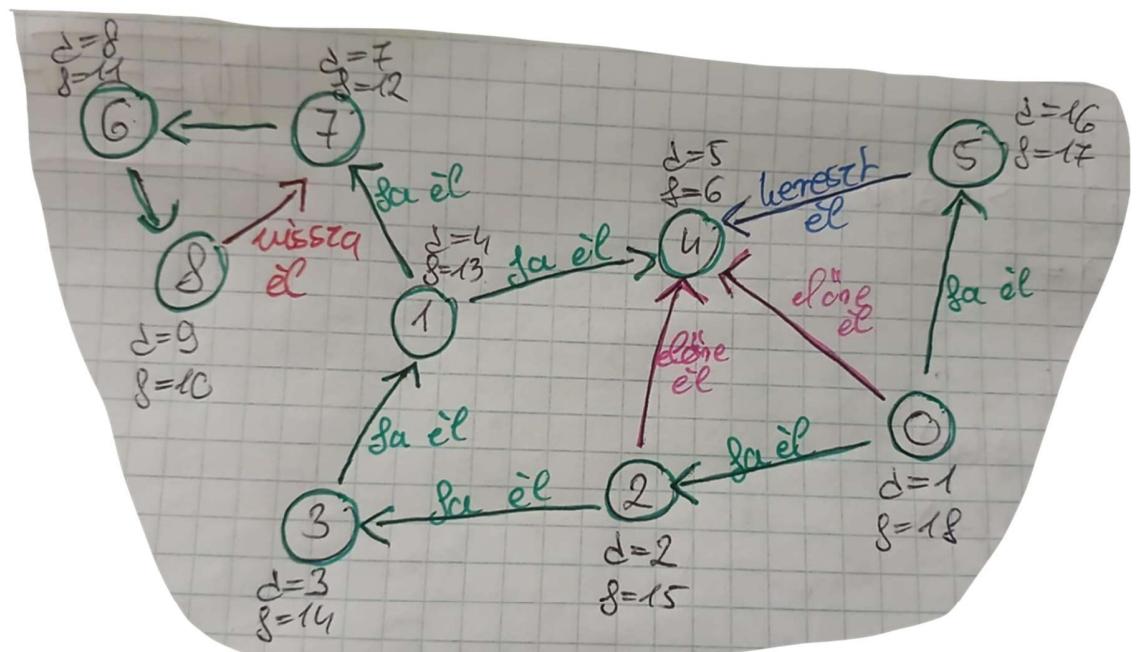
```
dfs_recursive(Graph):
    visited = HashSet()
    for node in Graph.V:
        if node not in visited:
            visit(node, &visited)
```

```
visit(node, visited):
    ...
    visited.add(node)
    ptr = Graph[node]
    while ptr is not null:
        if ptr->key is not in visited:
            visit(ptr->key, &visited)
```

Iteratívan

```
def dfs_iterative(Graph):
    visited = HashSet()
    for node in Graph.V:
        if node not in visited:
            sk = Stack()
            sk.push(node)
            while not sk.isEmpty():
                node = sk.pop()
                ...
                ptr = Graph[node]
                while ptr is not null:
                    if ptr->key is not in visited:
                        visited.add(ptr->key)
                        sk.push(ptr->key)
```

Példák



Irányítatlan gráfra

Élek csoportosítása

1. Fa él

- Olyan él, amelyen keresztül a DFS először fedez fel egy új csúcsot.

2. Vissza él

- Olyan él, amely egy csúcsból egy DFS őshöz (ancestor) mutat vissza.
- $d(u) > d(v)$
- $f(v) = ?$

3. Előre él

- Olyan él, amely egy csúcsból egy már felfedezett utódhoz (descendant) megy, de nem fa él.
- $d(u) < d(v)$

4. Kereszt él

- Olyan él, amely két olyan csúcs között megy, amelyek külön ágakban vannak a DFS során — azaz egyik sem őse a másiknak.
- $d(u) > d(v)$
- $f(v) \neq ?$

Minimális feszítőfák (MST)

Alapfogalmak

Gráf:

Egy gráf $G=(V,E)$:

- V : csúcshalmaz
- E : élek halmaza
- Az élekhez általában súly tartozik ez lehet költség, távolság, idő, ...

Feszítőfa

Egy feszítőfa egy összefüggő, körmentes részgráf, amely:

- tartalmazza az összes csúcsot,
- pontosan $|V|-1$ élt használ,
- nem tartalmaz ciklusokat.

Minimális feszítőfa (Minimal Spanning Tree)

Egy feszítőfa, amelyben az élek összsúlya minimális.

Mire jó?

A minimális feszítőfa olyan, mint egy optimális pókháló: minden pontot összeköt, de a lehető legkevesebb „anyaggal”.

Használják többek közt:

- hálózatépítés (kábelfektetés, infrastruktúra),
- klaszterezés (pl. gépi tanulásban),
- útvonaloptimalizálás,
- redundancia minimalizálás.

Tulajdonságok

Egyértelműség

- Ha minden él súlya különböző, az MST egyetlen.
- Ha vannak azonos súlyok, több különböző MST is létezhet.

Körtétel

- Egy körben a legnehezebb él soha nem része MST-nek.

Vágástétel

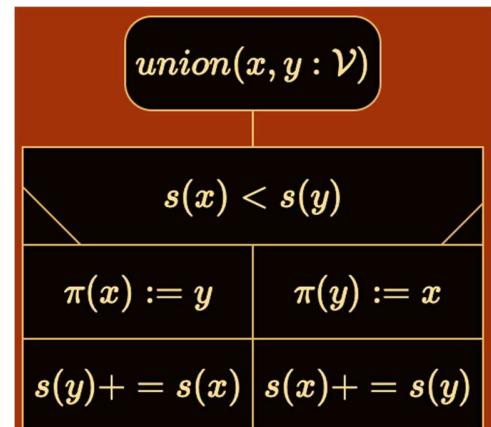
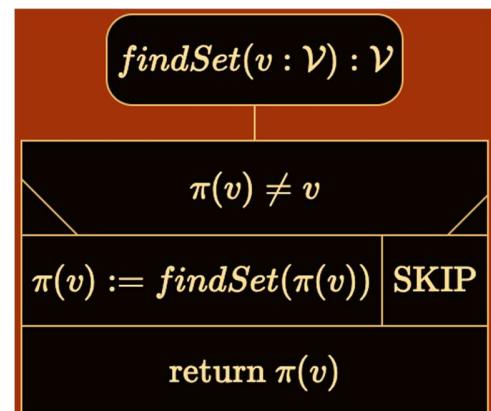
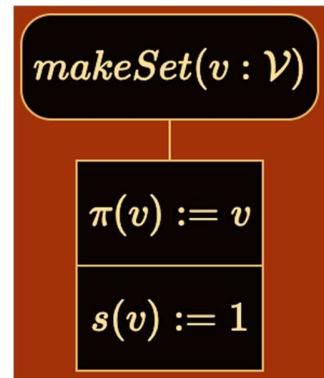
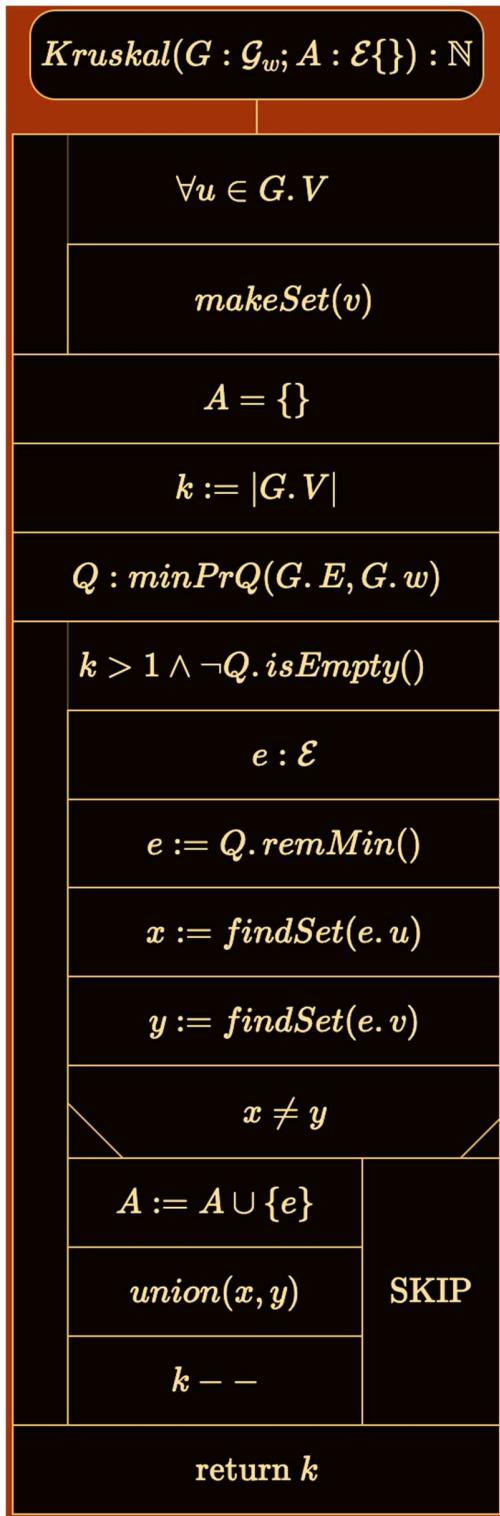
- Egy vágás mentén a legkisebb súlyú él mindig szerepel minden MST-ben.

Kruskal algoritmus

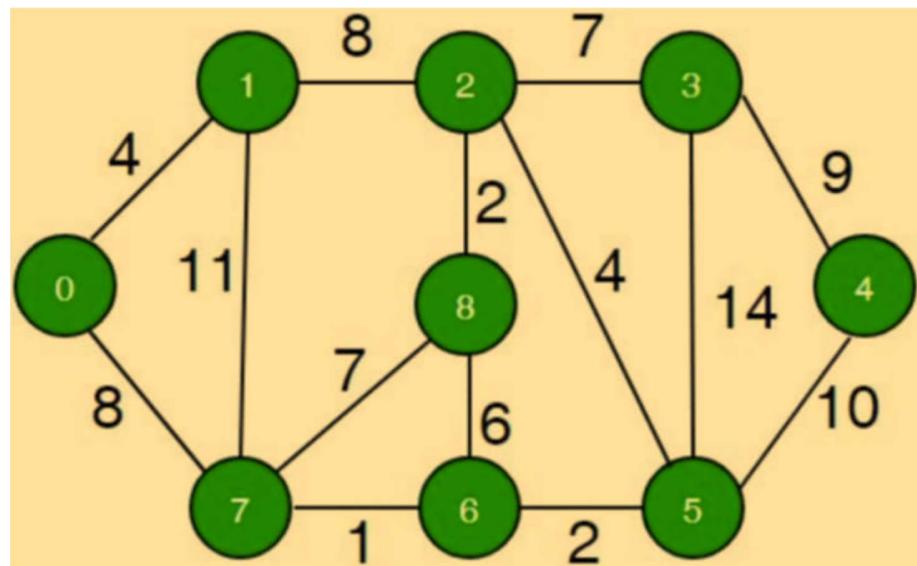
Filozófiája: „Mindet meg akarok tartani - kivéve, ami ciklust okoz.”

Lépések:

- Rendezés: sorold az éleket növekvő súly szerint.
- Haladj végig a listán:
 - ha az él nem alakít ki ciklust, add a fához;
 - ha igen, dob ki.
- Állj meg, ha van $|V|-1$ éled.



Példa



komponensek	következő él	él súlya	sikeress?
0; 1; 2; 3; 4; 5; 6; 7; 8	6 - 7	1	+
0; 1; 2; 3; 4; 5; 67; 8	2 - 8	2	+
0; 1; 28; 3; 4; 5; 67	5 - 6	2	+
0; 1; 28; 3; 4; 567	0 - 1	4	+
01; 28; 3; 4; 567	2 - 5	4	+
01; 25678; 3; 4	6 - 8	6	-
01; 25678; 3; 4	2 - 3	7	+
01; 235678; 4	7 - 8	7	-
01; 235678; 4	0 - 7	8	+
01235678; 4	1 - 2	8	-
01235678; 4	3 - 4	9	+
12345678			

szülő, méret									
0	1	2	3	4	5	6	7	8	
0, 1	1, 1	2, 1	3, 1	4, 1	5, 1	6, 1	7, 1	8, 1	
						6, 2	6		
		2, 2							2
						6	6, 3		
0, 2	0		6				6, 5		
				6			6, 6		
					6			6, 8	
						6		6, 9	

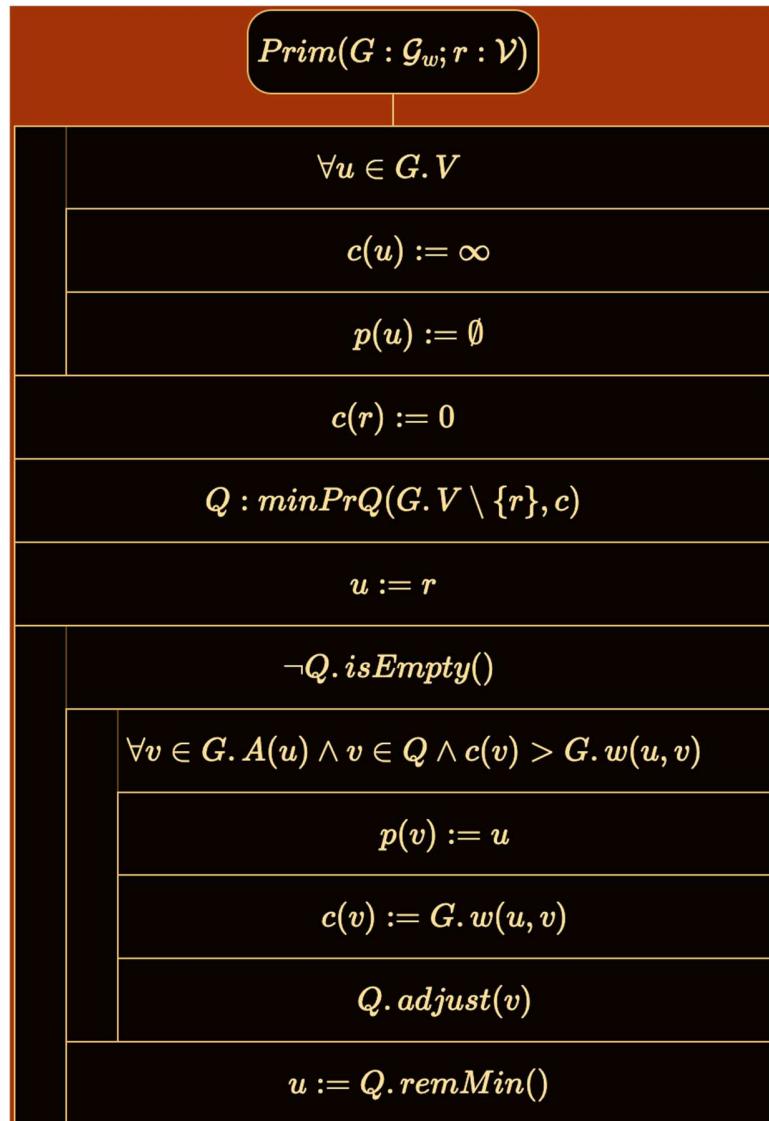
Méret csak ahhoz a szülőhöz van írva, amelyik gyökér az adott komponensben.

Prim algoritmus

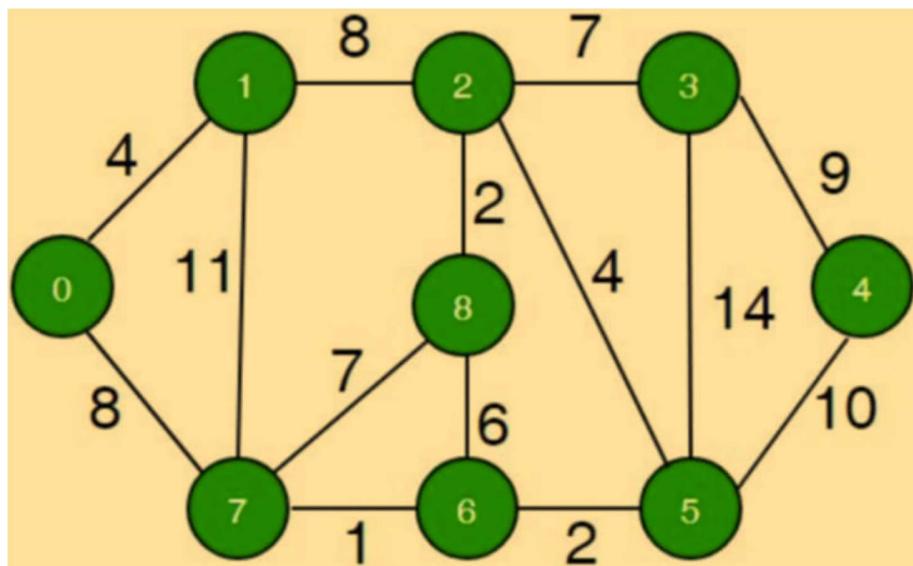
Filozófiája: „Növesztek egy fát. Szépen lassan lépésenként.”

Lépések:

- Válassz tetszőleges kezdőcsúcst.
- Mindig vedd fel azt a legkisebb súlyú élt, amely a jelenlegi fát összeköti egy külső csúccsal.
- Addig ismételd, amíg minden csúcs bekerül.



Példa



c									kiválasztva	p								
0	1	2	3	4	5	6	7	8		0	1	2	3	4	5	6	7	8
0	∞	0	\emptyset															
4							8		0		0							0
8									1			1						
		7		4				2	2				2		2		8	8
					6	7			8						5	5		6
			10		2		1		5									
									6									
									7									
									3								3	
									4									

c: A már kialakított részfából kiindulva melyik a legkisebb súlyú él, amely közvetlenül összeköti v-t a részfával. Nem egy összeg, nem a teljes út költsége, hanem egyetlen minimális él súlya.

Topologikus Rendezés

- A topologikus rendezés irányított gráfokra alkalmazható módszer, amely egy olyan lineáris sorrendet ad a gráf csúcsaira, hogy minden él a sorrendben korábban szereplő csúcsból mutat a későbbi felé.
- A módszer kizárolag irányított, körmentes gráfokon (DAG – Directed Acyclic Graph) értelmezhető.

Mit mond meg?

- A lista elején állók nem függenek a mögöttük lévőktől.
- A lista végén állók függhetnek az előttük lévőktől, de nem tudjuk, hogy kiktől vagy éppen hánytól. (nyilván az elsőtől biztosan)
- Az, hogy két csúcs egymás mellett van, nem jelenti, hogy kapcsolatban állnak.
- Ami biztos: ha u előbb van, mint v , akkor nem lehet él $v \rightarrow u$.

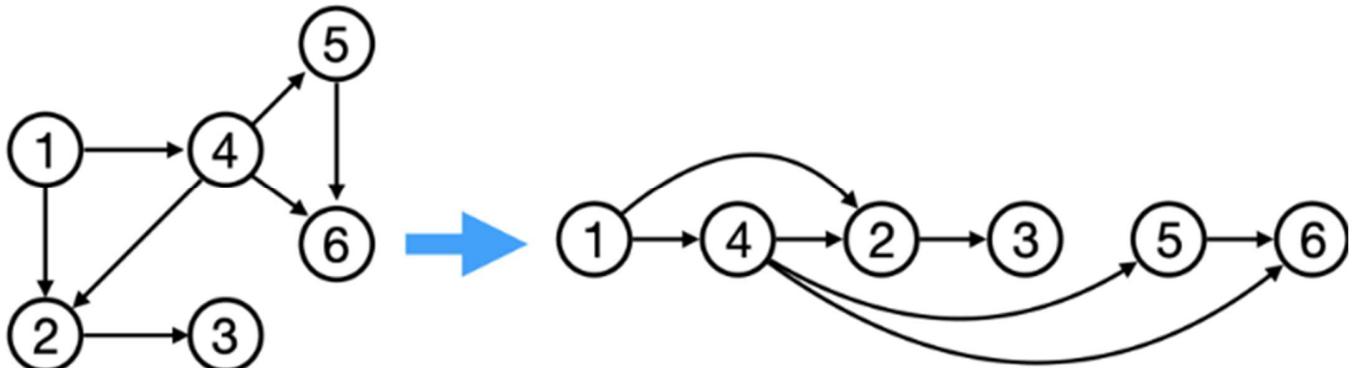
Mire jó?

- Függőségek vizsgálata
- Adatfolyamok elemzése
- Ütemezési problémák

Működése

- Létrehozunk egy üres listát
- Elindítunk egy gráfra egy DFS bejárást, és amikor az adott node-ot tartalmazó rekurziós hívás lekerül a stackről (vagy feketére színezzük), akkor hozzáadjuk ezt a node-ot a lista elejére.

Példa



Dijkstra algoritmus

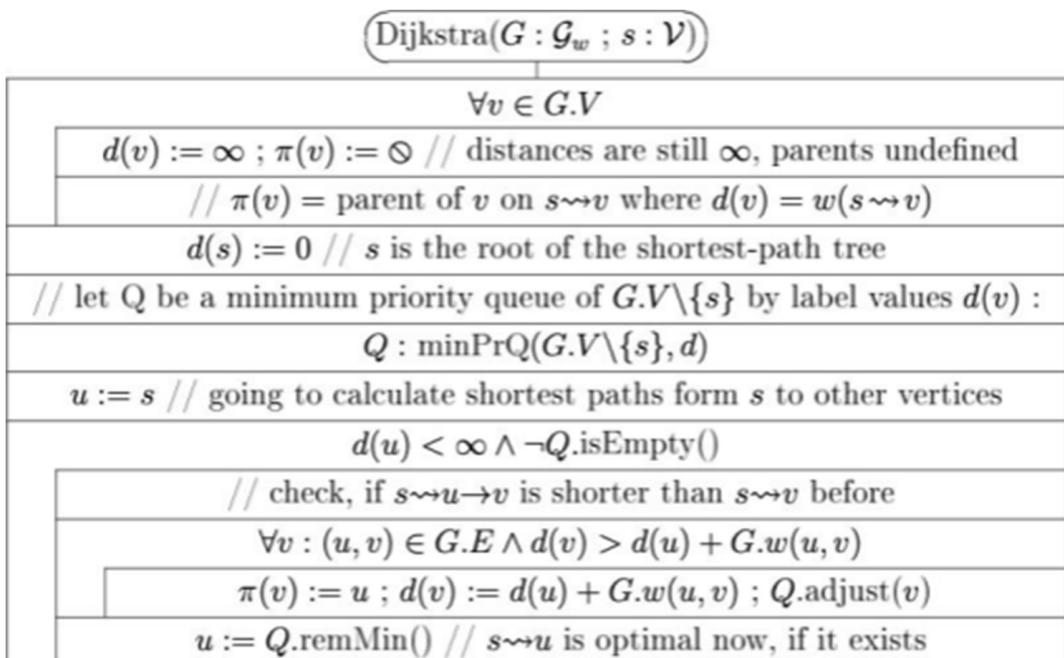
- Az algoritmus feladata egy forráspontból kiindulva megtalálni a legolcsóbb utak hosszát egy nem-negatív élhosszokkal rendelkező, irányított vagy irányítatlan gráf minden más csúcsához.

Működése

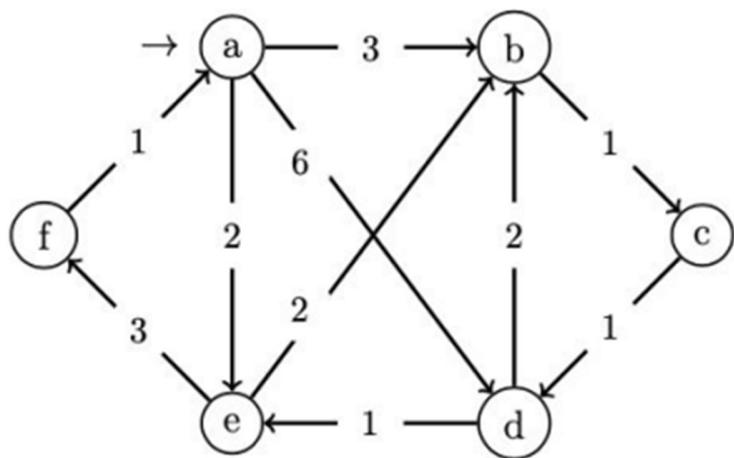
1. Inicializálás
 - minden csúcs: $d[v] = \infty$
 - minden csúcs: $\pi[v] = \emptyset$
 - Start: $d[s] = 0$
 - Üres prioritási sor
 - A sorba belerakjuk az összes csúcsot, a priority a d lesz
2. Iteráció
 - Amíg van feldolgozatlan csúcs:
 - Kivesszük a sor első elemét
 - Megnézzük van-e olyan csúcs, ahova ebből a csúcsból olcsóbban el tudnánk jutni, mint jelenleg

$$d[v] > d[u] + w(u, v) \Rightarrow d[v] := d[u] + w(u, v)$$
3. Leállás
 - A sor kiürül, vagy a kivett csúcs esetében $d=\infty$, magyarul fizikailag nem tudunk oda elmenni.

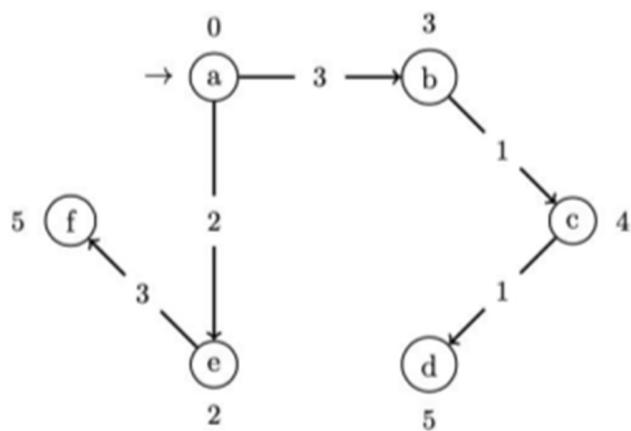
Zárójel: Az algoritmus úgy is működik, hogy az elején csak s kerül a sorba, így a végtelen távolságú csúcsok alapból be sem fognak kerülni. Egyszerűbb, gyorsabb, kevesebb memóriát használ.



Példa



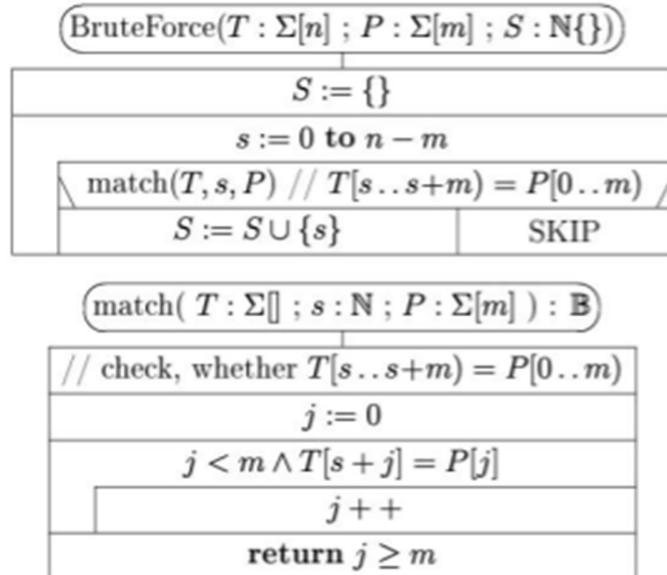
d értékek Q-ban						kiterjesztett csúcs:d	π címkék változásai					
a	b	c	d	e	f		a	b	c	d	e	f
0	∞	∞	∞	∞	∞	—	∅	∅	∅	∅	∅	∅
3	∞	6	2	∞		a : 0	a	a	a			
3	∞	6		5		e : 2						e
	4	6		5		b : 3				b		
		5		5		c : 4				c		
			5			d : 5						
0	3	4	5	2	5	eredmény	∅	a	b	c	a	e



Mintaillesztés

Brute Force

Pontosan azt csinálja, amit a neve sugall: minden lehetséges pozíció kipróbálja a mintát, míg meg nem unja.



$i =$	0	1	2	3	4	5	6	7	8	9	10
$T[i] =$	A	B	A	B	B	A	B	A	B	A	B
	<u>B</u>	A	B	A							
		<u>B</u>	<u>A</u>	<u>B</u>	<u>A</u>						
			<u>B</u>	A	B	A					
				<u>B</u>	<u>A</u>	B	A				
$s=4$					<u>B</u>	<u>A</u>	<u>B</u>	<u>A</u>			
						<u>B</u>	A	B	A		
$s=6$							<u>B</u>	<u>A</u>	<u>B</u>	<u>A</u>	
								<u>B</u>	A	B	A

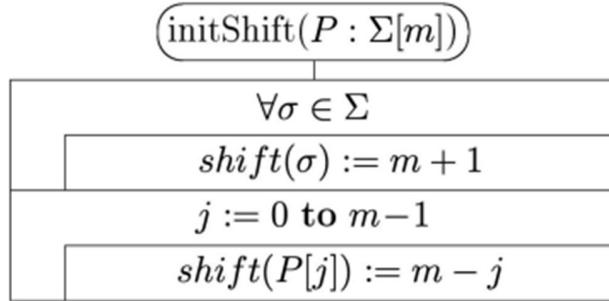
$$S = \{4, 6\}$$

Megjegyzés:

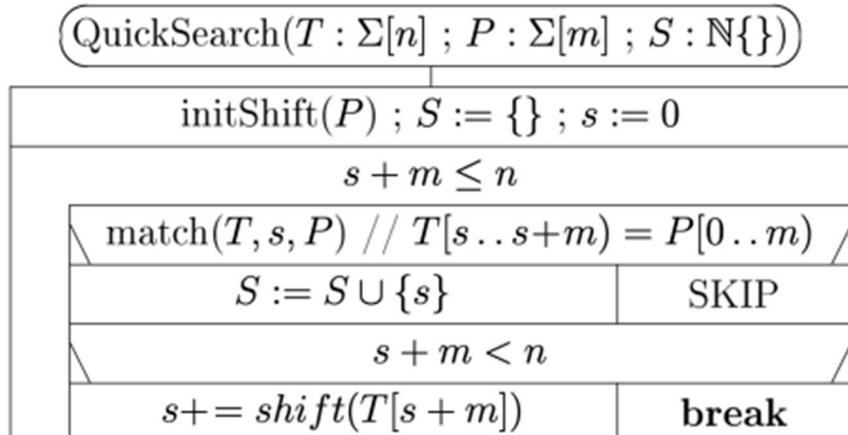
A táblázatban miután át van húzva egy betű, utána az algoritmus nyilván kilép a match() függvényből, és egyből növeli s értékét.

Quick Search

Ahelyett, hogy minden pozícióra végigellenőrizné a mintát, a QS kihasználja az információt arról, mi jön a minta után a szövegben. A shift táblát a mintára épít, és a vizsgált pozíció utáni karakter alapján dönti el, mennyit ugorhat előre.



σ	A	B	C	D
initial $shift(\sigma)$	5	5	5	5
C			4	
A	3			
D				2
A	1			
final $shift(\sigma)$	1	5	4	2



$i =$	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21	22		
$T[i] =$	<u>A</u>	<u>D</u>	<u>A</u>	<u>B</u>	<u>A</u>	<u>B</u>	<u>C</u>	<u>A</u>	<u>D</u>	<u>A</u>	<u>B</u>	<u>C</u>	<u>A</u>	<u>B</u>	<u>A</u>	<u>D</u>	<u>A</u>	<u>C</u>	<u>A</u>	<u>D</u>	<u>A</u>	<u>D</u>	<u>A</u>		
	\emptyset	<u>A</u>	<u>D</u>	<u>A</u>																					
	\emptyset	<u>A</u>	<u>D</u>	<u>A</u>																					
$s = 6$								<u>C</u>	<u>A</u>	<u>D</u>	<u>A</u>														
															<u>C</u>	<u>A</u>	<u>D</u>	<u>A</u>							
																\emptyset	<u>A</u>	<u>D</u>	<u>A</u>						
$s = 17$																		<u>C</u>	<u>A</u>	<u>D</u>	<u>A</u>				
																	\emptyset	<u>A</u>	<u>D</u>	<u>A</u>					

KMP (Knuth–Morris–Pratt)

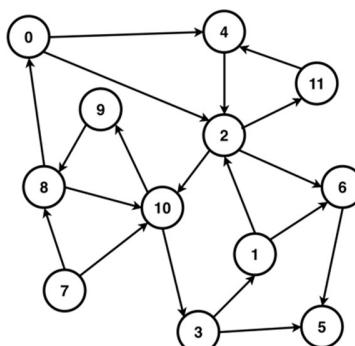
<https://people.inf.elte.hu/pgm6rw/algo/Algo2/PatternSearching/knuthMorrisPratt/index.html>

Szorgalmi feladatok

1. Tömörítsd a Naiv módszerrel az alábbi szöveget. (1 pont)
 - HUMBABUMBLAKUMPALUMPABUUUUU
Továbbá add meg a
 - Kódtáblát
 - Kódtábla méretét
 - Tömörített szöveg méretét
 - A szöveg eredeti méretét
2. Tömörítsd a Huffman módszerrel az alábbi szöveget. (1 pont)
 - HUMBABUMBLAKUMPALUMPABUUUUU
Továbbá add meg a
 - Kódfát
 - Kódtáblát
 - Kódtábla méretét
 - Tömörített szöveg méretét
 - A szöveg eredeti méretét
3. Miért működik úgy az LZW speciális esete ahogy? (1 pont)
4. Tömörítsd az LZW módszerrel az „XXYXXXXYXXYZYYZZZZXZY” szöveget. (1 pont)
5. Építs AVL fát a következő elemekből: 20, 10, 80, 25, 50, 40, 70, 60, 90, 30. Töröld a következő elemeket az előbbi fából: 10, 25, 20, 30, 70. (2 pont)
6. Építs B+ fát a következő elemekből: 17, 5, 10, 24, 12, 20, 32, 27, 9, 7, 4, 14, 16, 13, 6, 3. Töröld a következő elemeket az előbbi fából: 10, 32, 27, 7, 17, 20, 9, 24, 16. (2 pont)
7. BFS-el járd be az alábbi gráfot! (add meg a táblázatot) (1 pont)

0	1	0	1	1	0	1
1	0	1	0	0	1	1
0	1	0	1	1	0	1
1	0	1	0	1	1	0
1	0	1	1	0	1	0
0	1	0	1	1	0	1
1	1	1	0	0	1	0

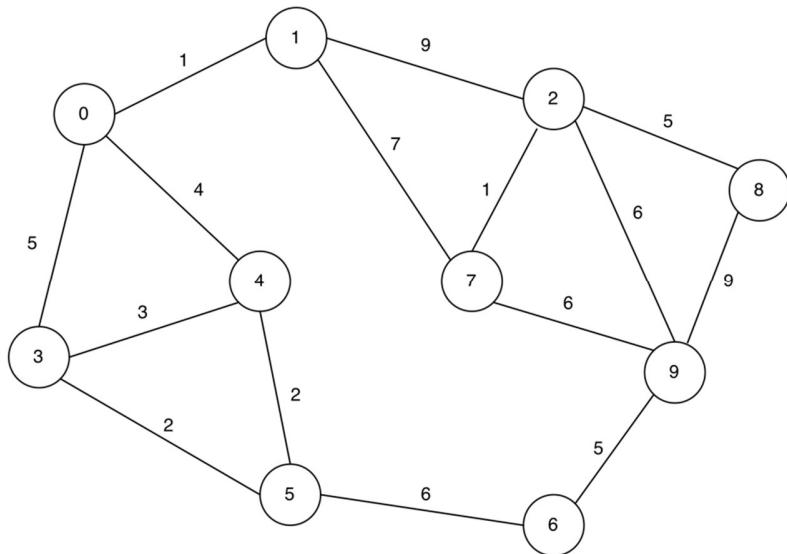
8. DFS-el járd be az alábbi gráfot! (add meg “d” és “f” értékeit, illetve osztályozd az éleket) (1 pont)



9. Építs MST-t az alábbi gráfból az alábbi algoritmus segítségével:

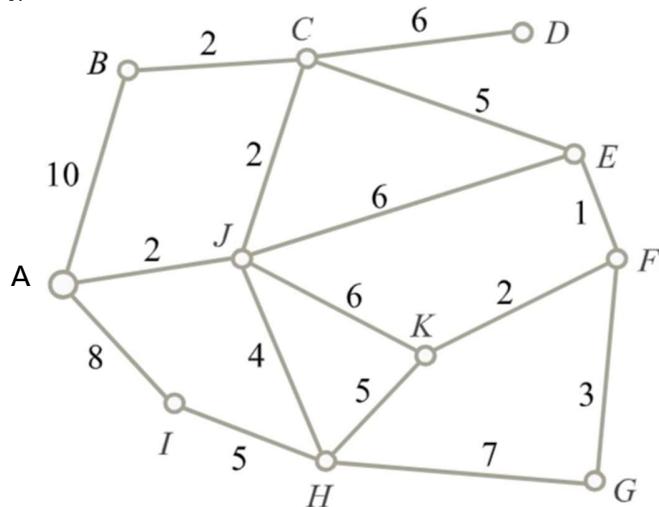
- a. Kruskal (1 pont)
- b. Prim (1 pont)

Add mega táblázatot(okat) és a végeredményt.



10. A 8. feladatban DFS által bezárt gráfon add meg a DFS által generált topologikus sorrendet.
Majd adj meg egy különböző, de helyes sorrendet (Nem kell a bezárást). (1 + 1 pont)

11. Dijkstra (A-ból indulj)



12. Mintaillesztés

Szöveg: aaaaaaaaaabcdeabcfgeabcfdeabccdfababcdeeeee

Minta: abcde

- a) Brute Force (1 pont)
- b) Quick Search (1 pont)
- c) KMP (1 pont)