

# 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 kódában 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:  $n * L = 11 * 3 = 33$  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: ABRAKADABRAABRAKADABRAABRAKADABRA

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

## Előnye

- Egyszerű
- Az eredeti adat teljesen visszaállítható, semmilyen információ nem vesz el
- Minden kódszó fix hosszúságú  $\rightarrow$  a bitsorozatot egyszerűen szakaszokra lehet bontani
- Nem szükséges bonyolult fa vagy keresés (ellentétben 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 gyökércsúcsának, ahol a kulcs a két fa gyökércsúcsában 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 levélig 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 kódsorozatot

## Kitömörítési eljárás

A tömörített szöveget elkezdjük olvasni, és ha találunk olyan sorozatot ami a kódtáblában szerepel, akkor az a sorozat a kódtáblában lévő karakternek 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 >

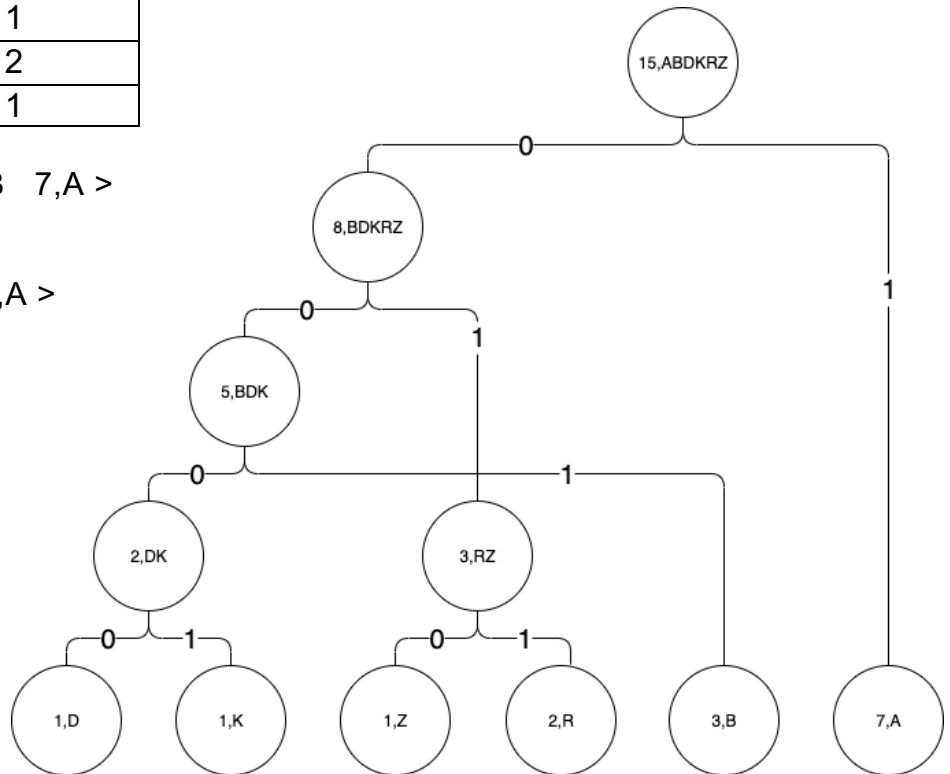
2. < 1,Z 2,DK 2,R 3,B 7,A >

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 >

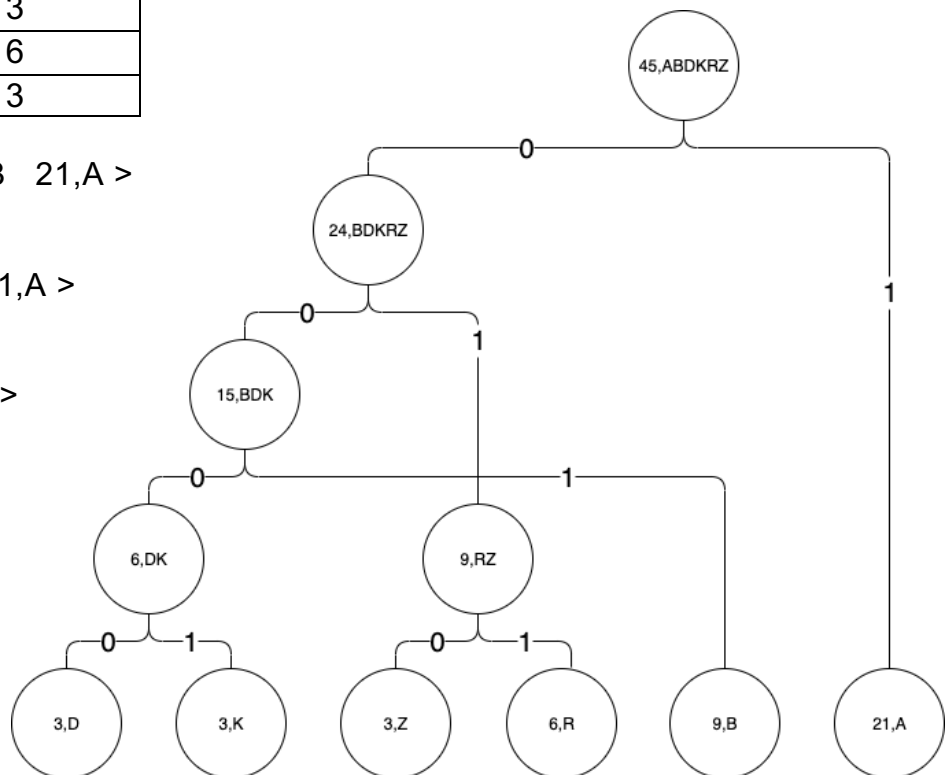
8. < 3,Z 6,DK 6,R 9,B 21,A >

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:  $21 \cdot 1 + 9 \cdot 3 + 3 \cdot 4 + 3 \cdot 4 + 6 \cdot 3 + 3 \cdot 3 = 99$  bit
- Kódtábla mérete:  $6 \cdot 8 + 1 + 3 + 4 + 4 + 3 + 3 = 66$  bit
- Teljes tömörített méret:  $99 + 66 = 165$  bit (plusz meta adatok)
- Eredeti méret:  $45 \cdot 8 = 360$  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 (ellentétben 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. lépésre
3. Konkaténáljuk az előző és a beolvasott „karaktereket”
  - a. Ha a konkaténált „szó” benne van a kódtáblázatban
    - i. Akkor az előző „szó” legyen a konkaténált „szó”
  - b. Ha a konkaténált „szó” nincs benne a szótárban
    - i. Vegyük fel a konkaténált „szót” a kódtáblázatba
    - ii. Írjuk ki az előző „szó” kódját
    - iii. Az előző „szó” legyen a beolvasott „szó”
4. Vissza a 2. lépésre

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



## 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](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 tudunk 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. lépésre

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

## 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](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 szövegben
- Veszteségmentes tömörítés

## Hátránya

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

# 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 alapelv:

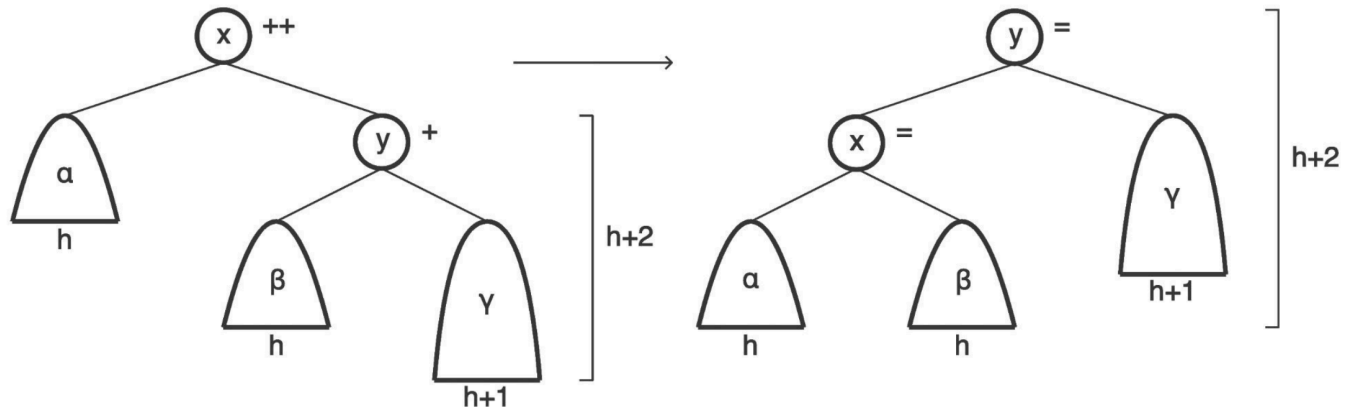
- Ha ez a különbség mindig  $-1$ ,  $0$  vagy  $+1$ , akkor a fa kiegyensúlyozott
- Ha ennél nagyobb eltérés keletkezne, akkor a fa rotációkkal kiegyenlíti magát

Ennek köszönhetően az AVL fa garantálja, hogy:

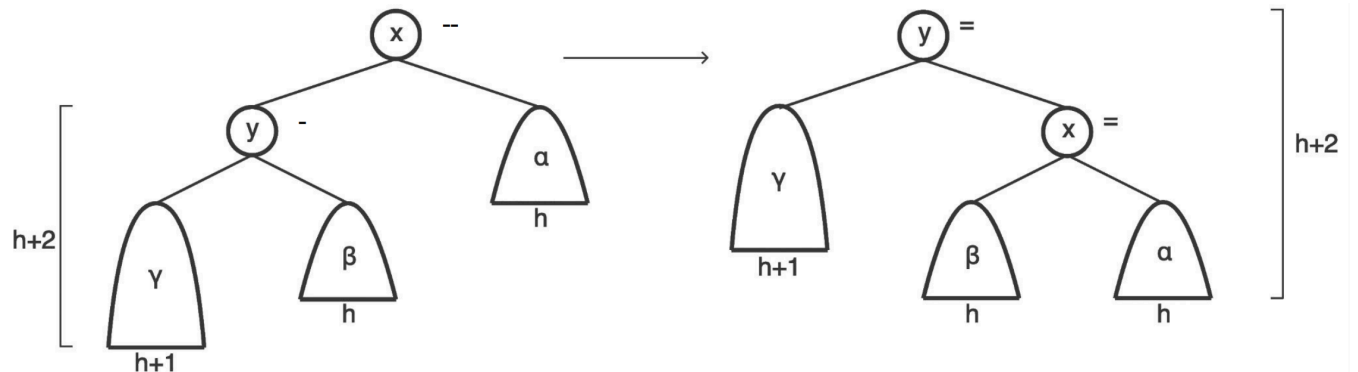
- a keresés, beszúrás és törlés műveletek  $O(\log n)$  időben futnak
- a fa magassága mindig logaritmus marad a csomópontok számához képest

## Forgatások

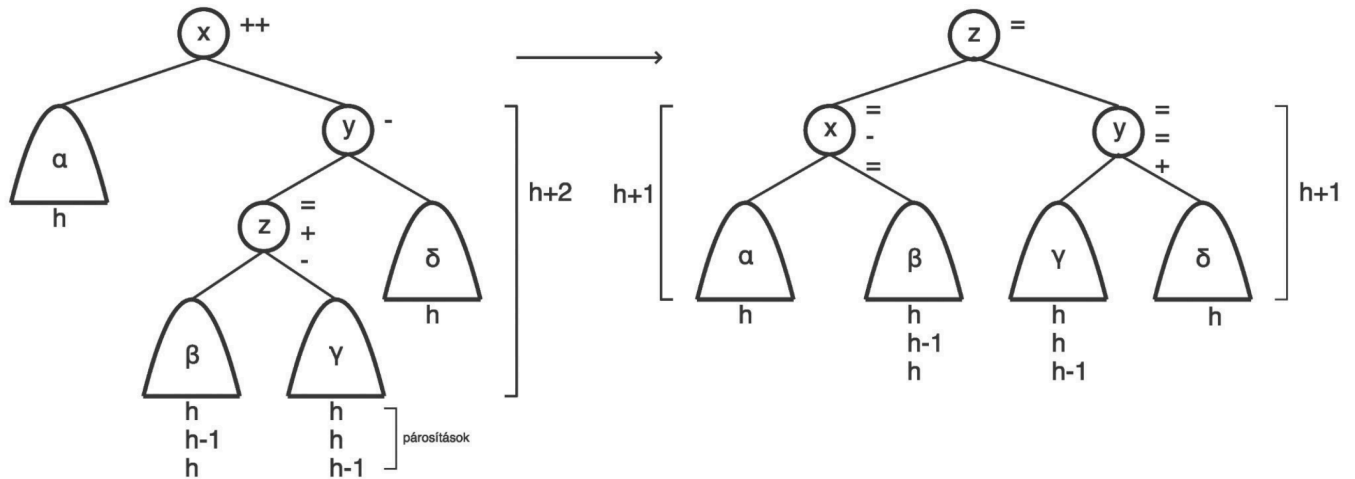
(++, +)



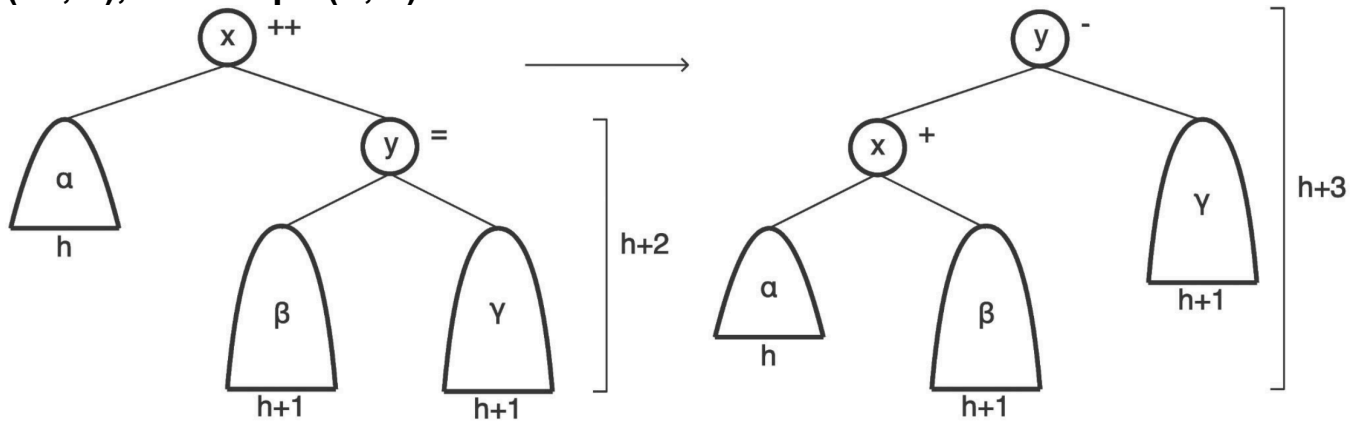
(--, -)



**(++, -), 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

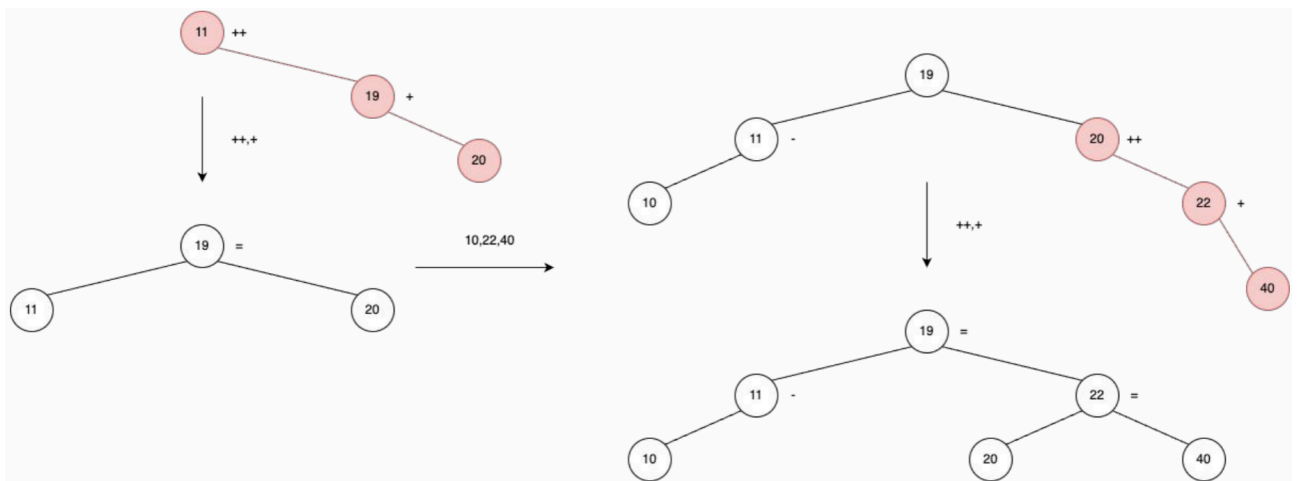
Röviden: beszúrás után alulról felfelé frissítjük a magasságokat és balance faktorokat, és ha kell, egy forgatással helyreállítjuk az egyensúlyt.

## Törlés

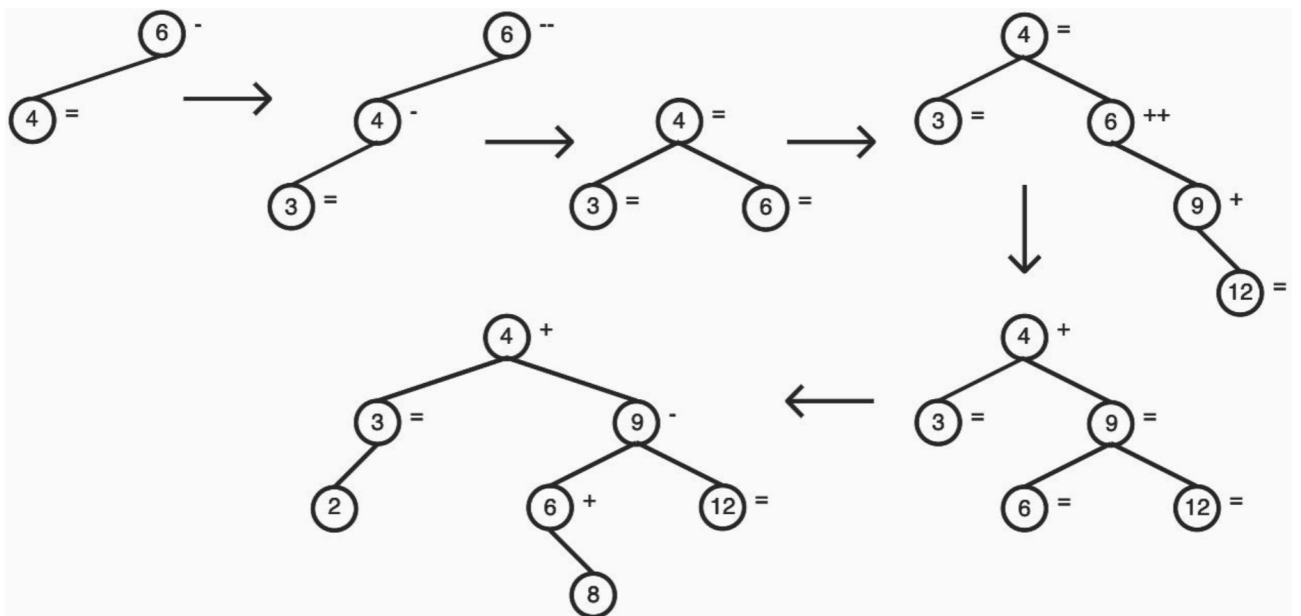
1. Megkeressük a törlendő elemet a keresőfa szabály szerint.
2. Három eset lehet:
  - a. Levél vagy egygyerekes csúcs: egyszerűen eltávolítjuk, a másik részfat (ha van) láncoljuk a helyére
  - b. Kétgyerekes csúcs: megkeressük a jobb részfa minimumát (vagy bal részfa maximumát), azt kivesszük, és az 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

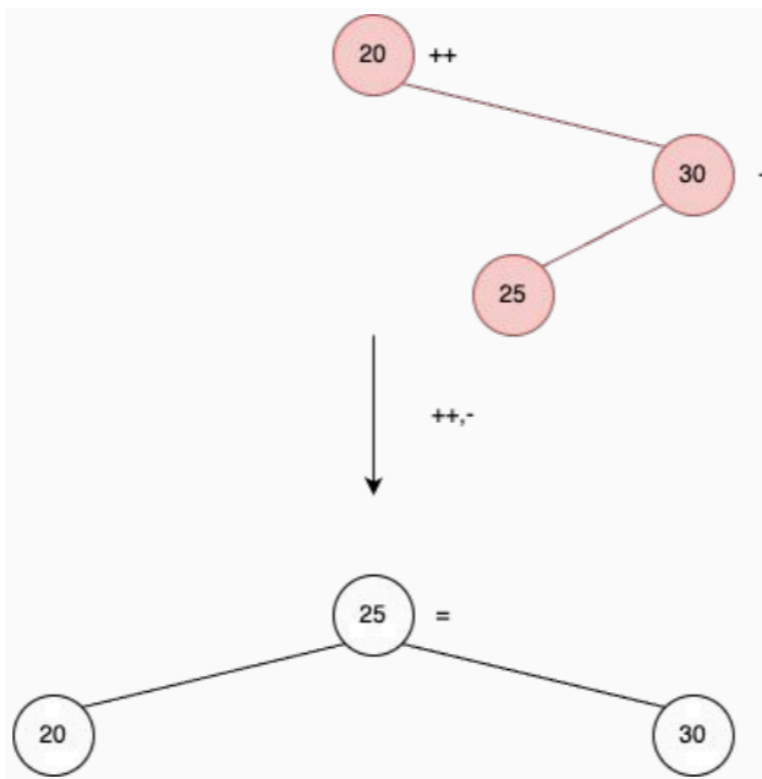
Építsünk fát a következő adatokból: 11, 19, 20, 10, 22, 40



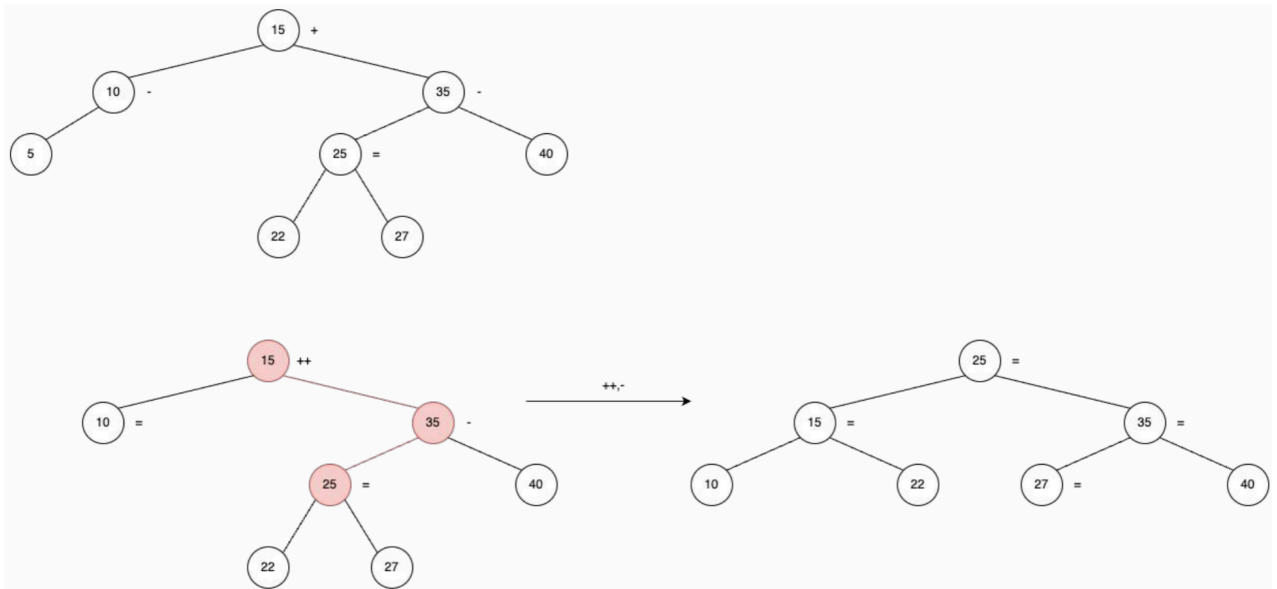
Építsünk fát a következő adatokból: 6, 4, 3, 9, 12, 2, 8



Építsünk fát a következő adatokból: 20, 30, 25

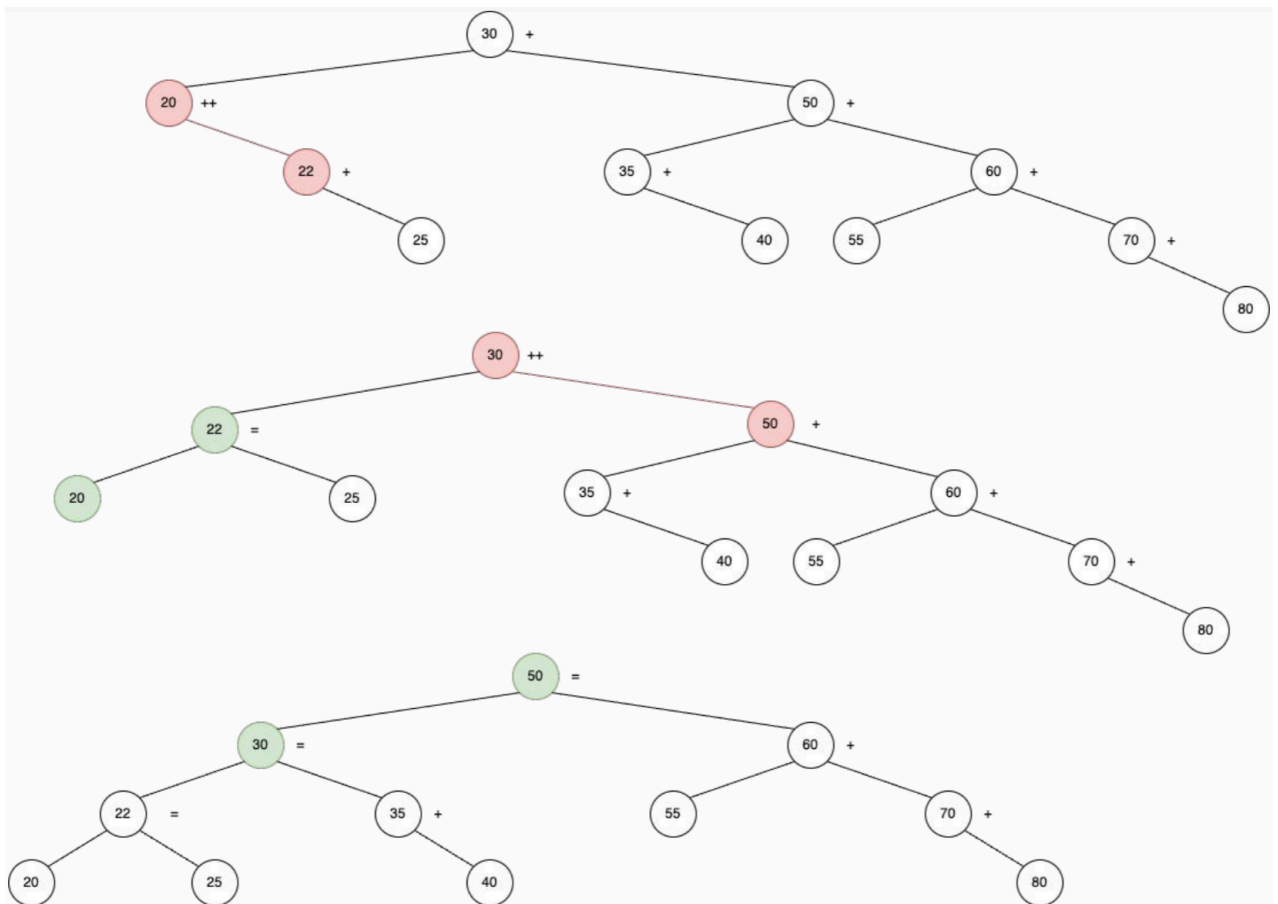
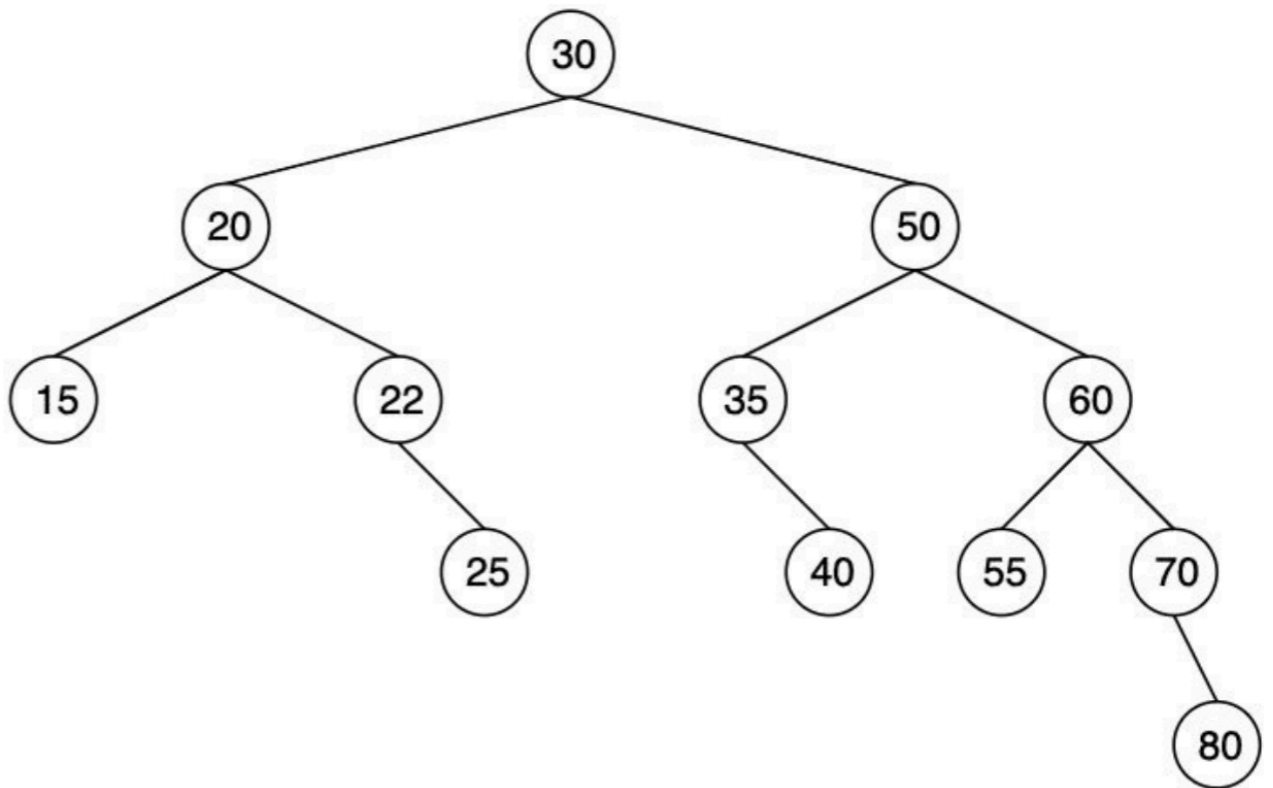


Töröljük az alábbi fából az 5-ös elemet

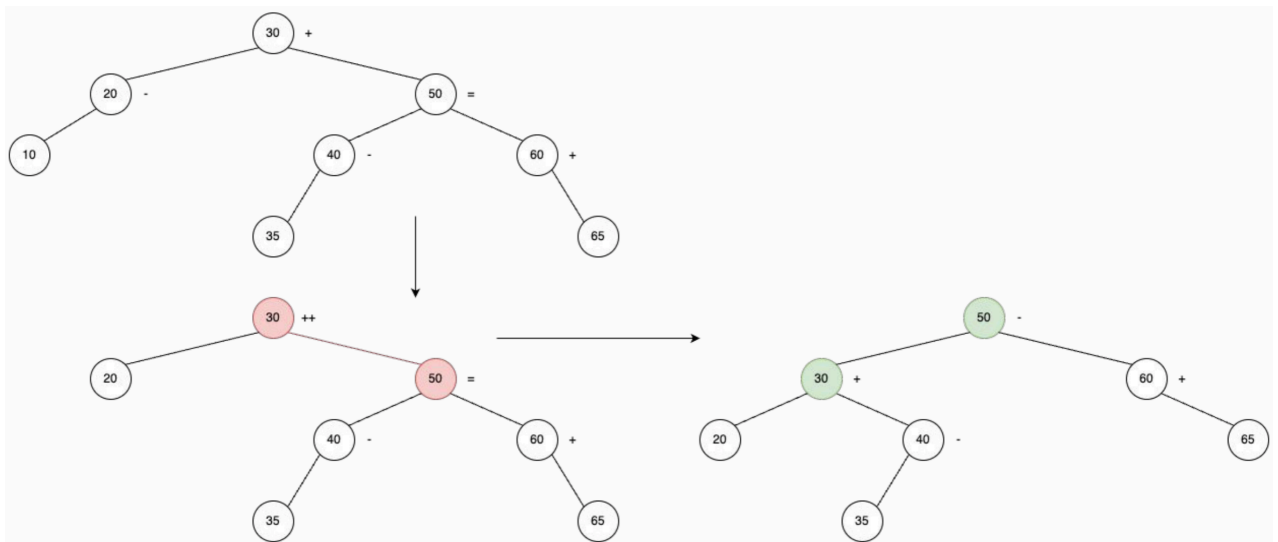




Töröljük az alábbi fából a 15-ös elemet



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 fájlrendszerekben 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 található
  - 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 gyerekekhez tartozó részében minden kulcs kisebb, mint a Cs első kulcsa; az utolsó gyerekekhez tartozó részében minden kulcs nagyobb-egyenlő, mint a Cs utolsó kulcsa; és az  $i$ -edik gyerekekhez tartozó részében ( $2 \leq i \leq k$ ) 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

## **Beszúrás**

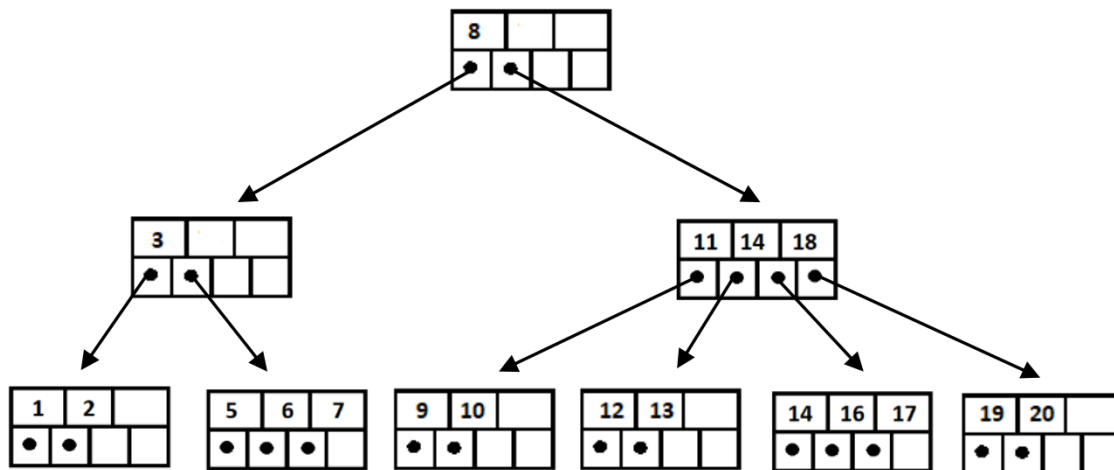
Szabályok a példákon keresztül.

## **Törlés**

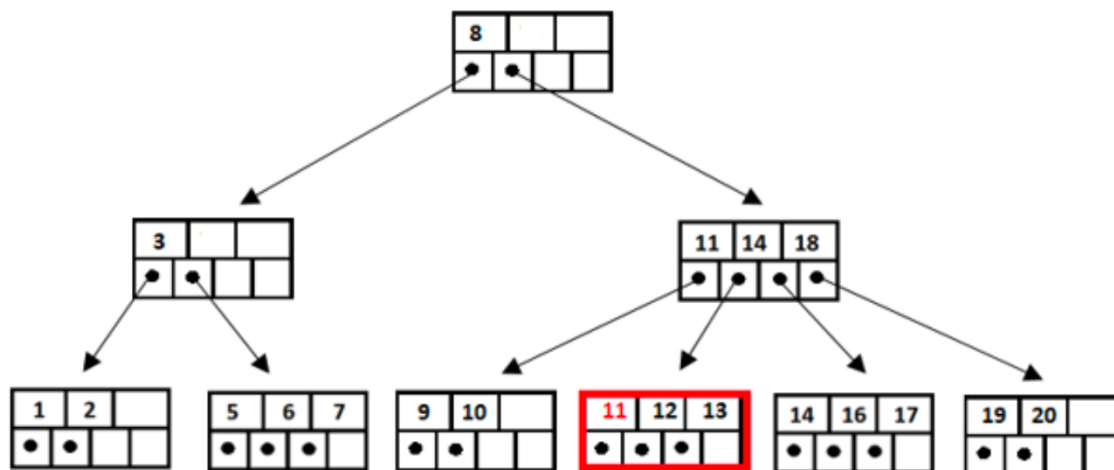
Szabályok a példákon keresztül.

## Példa

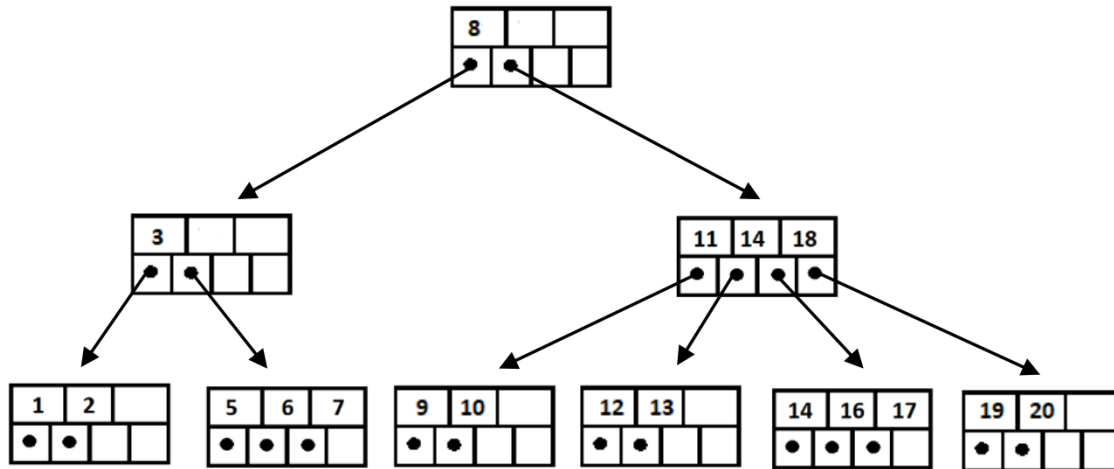
Szúrjuk be a 11-t az alábbi fába!



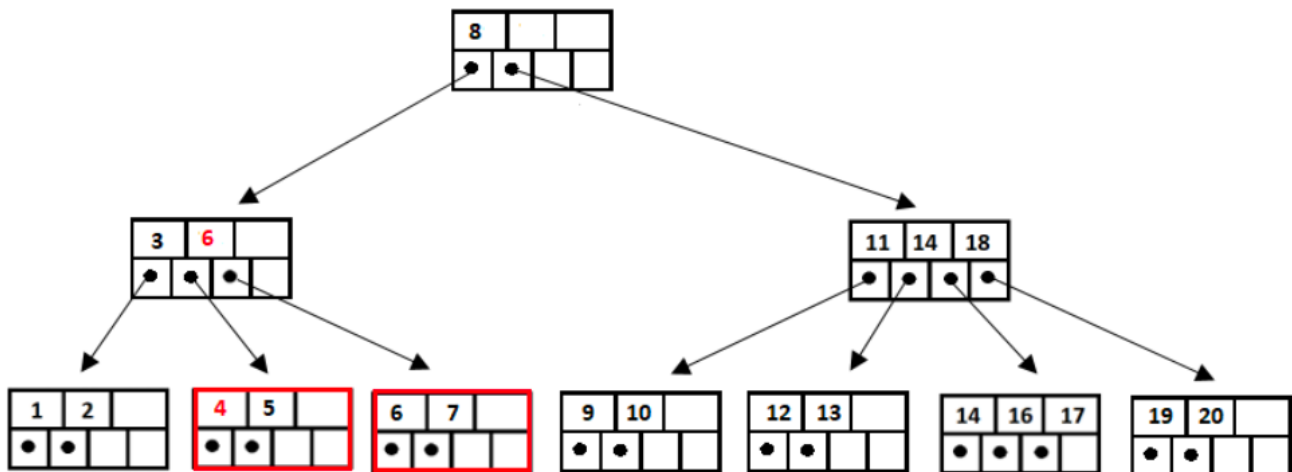
1. Megkeressük a 11 helyét: (12 13 levél)
2. Mivel a levélben nem szerepel a 11-es, és van hely, ezért egyszerűen beszúrjuk



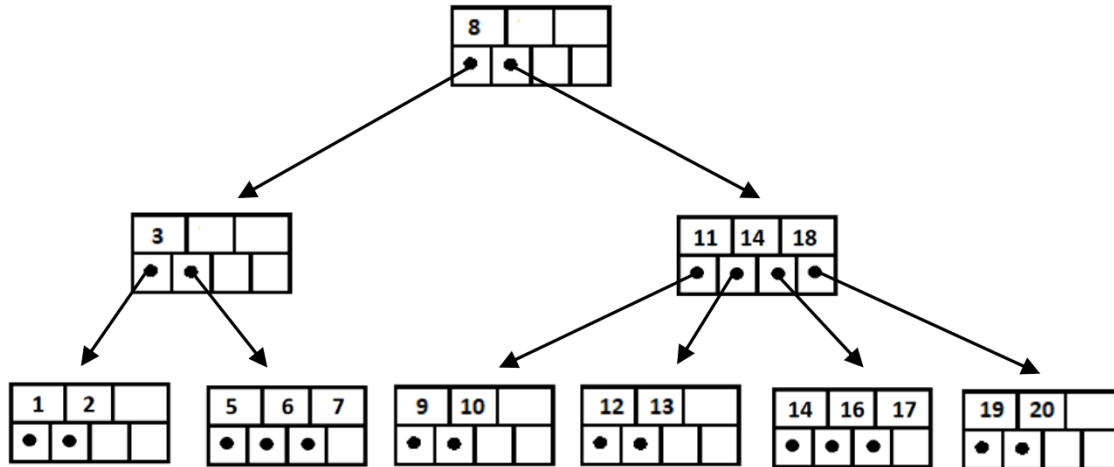
Szűrjük be a 4-t az alábbi fába!



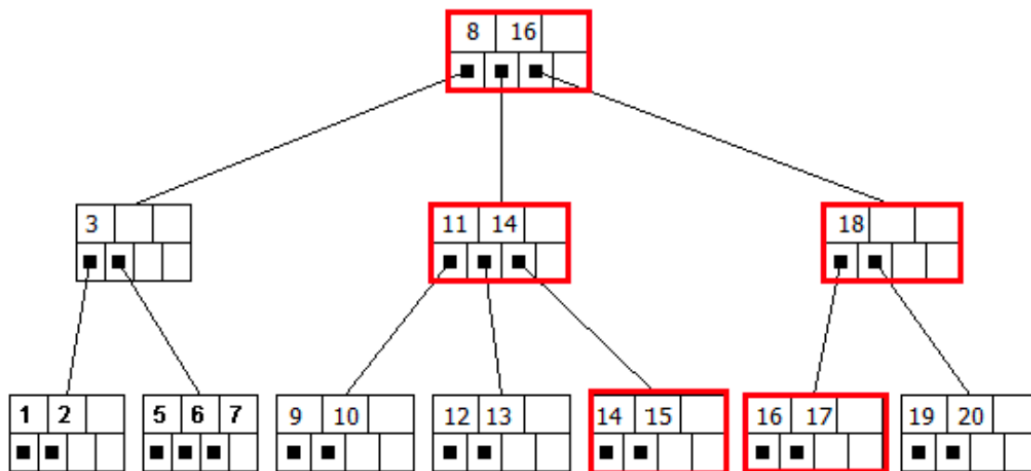
1. Megkeressük a 4-es helyét: (5 6 7) levél
2. Mivel nincs több hely, ezért szétvágjuk és beszúrjuk a 4-et: (4 5) (6 7)
3. Változtatjuk a szülő csúcsot:
  - a. Mivel az nem telített, egy új kulcsot helyezünk el. Az új kulcs mindig az új levél első eleme lesz: (3 6)



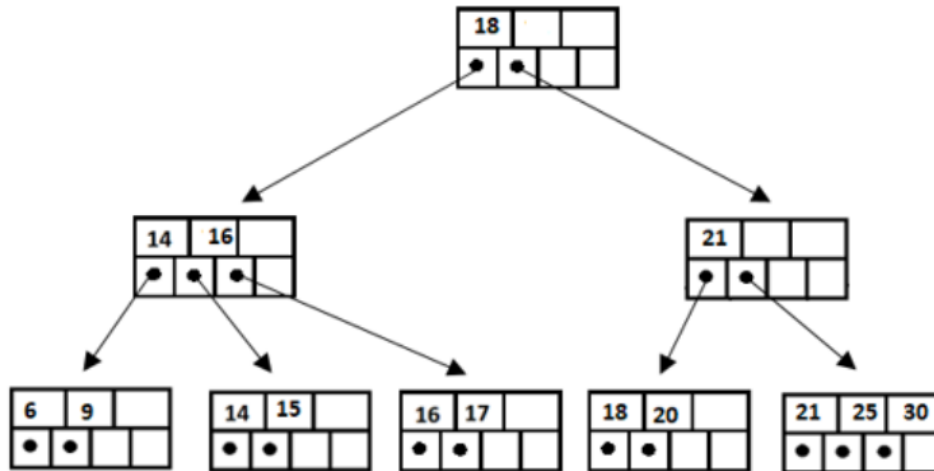
Szúrjuk be a 15-t az alábbi fába!



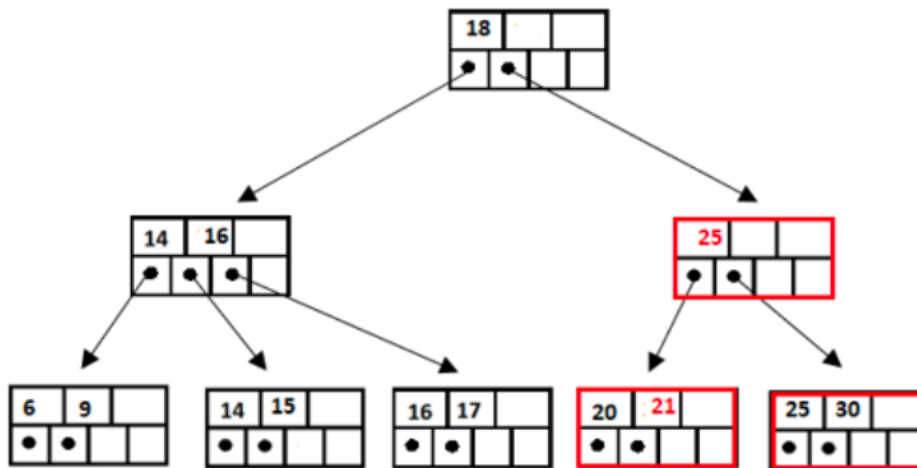
1. Megkeressük a fában a 15 helyét: (14 16 17) levél
2. Mivel ez a levél már telített, szétvágjuk: (14 15) (16 17)
3. Változtatjuk a szülő (11 14 18) csúcsot:
  - a. Mivel az is telített, azt a csúcsot is kettévágjuk.
  - b. Mivel ennek a szülője már nem telített, egy új kulcsot és mutatót helyezünk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



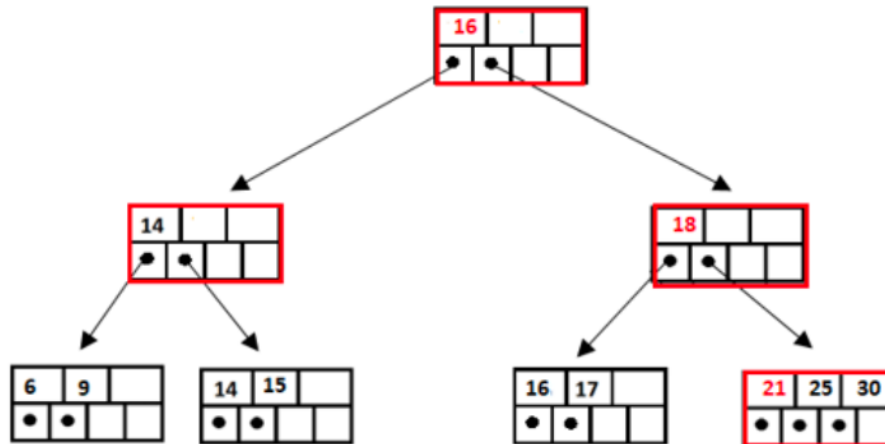
Töröljük az alábbi fából a 18-t, majd a 20-t, majd a 6-t!



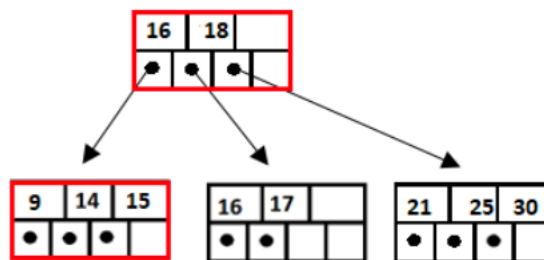
1. Megkeressük a fában a 18 helyét: (18 20) levél
2. Mivel 18 törlésével a levél egyedül marad, a testvérétől kap kulcsot: (20 21) ( 25 30)
3. Változtatjuk a szülő csúcsot:
  - a. A 21 kulcs törlődik és helyette a 2. gyerekének (25 30) az első eleme lesz: 25



1. Megkeressük a fában a 20 helyét: (20 21) levél
1. Mivel 20 törlésével a levél egyedül marad, 21 átadódik a jobb testvérének: (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 testvérének: (9 14 15)
3. Mivel az említett levél szülőjének (14) is csak egy gyereke lesz, a jobbtestvére pedig nem tud átadni gyereket, az említett csúcst összevonjuk a jobbtestvérével
4. A gyökérnek egy gyereke 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. Beszúrásnál mindig igaz, hogy a belső csúcsban a kulcs mindig egyenlő lesz annak a levélnek az első kulcsával amire a „nagyobb egyenlő mutatója” mutat, ha a fából törölünk 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/algo/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.



# Szorgalmi feladatok

1. Tömörítsd a Naiv módszerrel az alábbi szöveget. ( 1 pont )
  - HUMBABUMBLAKUMPALUMPABUUUUUTovábbá add meg a
  - Kódtáblát
  - Kódtábla méretét
  - Tömörített szöveg méretét
  - A szöveg eredeti méretét
2. Tömörítsd a Huffman módszerrel az alábbi szöveget. ( 1 pont )
  - HUMBABUMBLAKUMPALUMPABUUUUUTovábbá 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 „XXYXXYXXYXZYZZZZZXZY” 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 )