10. Algoritmusok fogalma, specifikációja és leírása. Algoritmusok hatékonysága. Adatszerkezetek és alkalmazásaik (szekvenciális és asszociatív, listaszerkezet, gyűjtemények, FIFO, LIFO, bináris kupac, hasító táblázat, keresőfák). Faműveletek (keresés, beszúrás, törlés, kiegyensúlyozás). Rendezések (beszúró, összefésülő, gyors, kupac, leszámoló, radix). Gráfok, és gráfalgoritmusok: keresések (szélességi és mélységi), bejárások (pre-, in- és postorder), feszítőfák (Prim és Kruskal algoritmusok), legrövidebb utak (Dijkstra és Bellmann-Ford algoritmusok).

# I. Algoritmus

Az **algoritmus** egy lépésről lépésre történő **utasítássorozat**, amely egy meghatározott bemenettel kezdődik és egy meghatározott kimnetet ad vissza eredményül. Az algoritmus **feladata**, hogy végesszámú lépésben megtalálja a megoldást, eljutni a célig.

Elkészítése:

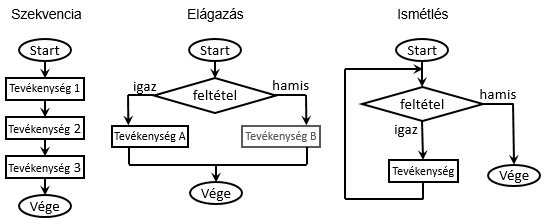
1. A feladat pontos megfogalmazása.
2. A feladat megtervezése (feltételek).
3. Algoritmus összeállítása.
4. Algoritmus ellenőrzése, ha szükséges módosítása.
5. Ha szükséges, az algoritmus dokumentálása.

## I. 1. Specifikáció és leírása

Algoritmus: egy **függvény**, amely

* Bemenete: értelmezési tartomány (adatszerkezet)
* Kimenete: az értékkészlet (adatszerkezet)
* Specifikáció: a függvény definíciója, amely megadja, hogy milyen bemenethez milyen kimenetet kell előállítania

Az algoritmusaink **leírásához** többféle mód áll rendelkezésünkre: ***pszeudokód*** (egy jól áttekinthető beszédes leíró szerkezet), ***folyamatábra*** (különféle alakzatokkal való lépés ábrázolás), ***stuktogram*** (egy strukturált, egységbezárt leíró eszköz). Példa megtekinthető a faktoriális algoritmusra: **VI. 1. Melléklet: algoritmus ábrázolás**



2. ábra folyamatábra

## I. 2. Hatékonyság

Ahhoz, hogy megtudjuk, hogy egy algoritmus mennyire bonyolult, megkell ismernünk annak lépésszámát és a futásidejét. Ha megtudjuk határozni az algoritmusunk lépésszámát, akkor a kapott adatokból eldönthető, hogy egy-egy algoritmus mennyire bonyult. A bonyolultságból következik, hogy egy algoritmus mennyire hatékony, optimális.

**Lépésszám:** a futásidő mellett tudjuk azt is nézni, hogy adott bemenő adatmennyiségre az algoritmus hány lépésben fut le.

Az algoritmus hatékonyságát általában két szempontból értékelik:

* **Időbonyolultság**: mennyi időbe telik az algoritmusnak a bemenet feldolgozása. Ez függ a bemenet méretétől, és általában a legrosszabb eseti vagy átlagos eseti teljesítmény alapján értékelik. Például, ha egy algoritmus futási ideje O(n) egy *n* elemű bemenetre, akkor az lineárisan növekszik (A konstans szorzókat és kisebb rendű tagokat figyelmen kívül hagyjuk).
* **Térbonyolultság, memória**: mennyi memóriát igényel az algoritmus a végrehajtás során. Fontos szempont különösen korlátozott erőforrású rendszerek esetén. Az összehasonlítás ugyanúgy zajlik, mint a futási időnél. Például az O(n) az algoritmus memóriahasználata lineárisan növekszik a bemeneti méret növekedésével.

Az algoritmusok **hatékonyságának növelése érdekében** különböző megközelítéseket alkalmaznak, mint például az adatszerkezetek **optimalizálása**, algoritmus **tervezési minták** ( rekurzió vagy iteráció) .

Miért is fontos nekünk, hogy tudjuk egy algoritmus lépésszámát?

* Az ***átlagos lépésszám*** akkor fontos, ha azt az algoritmust gyakran kell lefuttatni. Példa: Az adatbázisban, egy webáruházban termékek keresése, az algoritmus átlagos lépésszámának ismerete segít megérteni, mennyi időbe telik átlagosan egy lekérdezés feldolgozása
* ***maximális******lépésszám*** fontos, ha egy adott időn belül szeretnénk az eredményt kapni, valamiféle határidőre (például időjárás jelentés). Másik példa: Egy RTS (real-time strategy) játékban az AI-nak gyorsan kell döntenie, hogy hogyan reagáljon a játékos cselekvéseire még ha az nem optimláis megoldás.
* ***minimum******lépésszám*** mi az abszolút legalacsonyabb erőforrás-igény, amire szükség van az eredmény eléréséhez. Példa: Egy egyszerű rendezési algoritmus, mint a buborékrendezés, esetében a minimum lépésszám megmutatja, hogy ha az adatsor már rendezett, akkor az algoritmusnak csak egyszer kell végigmennie az elemeken, hogy ellenőrizze, tényleg rendezettek-e.

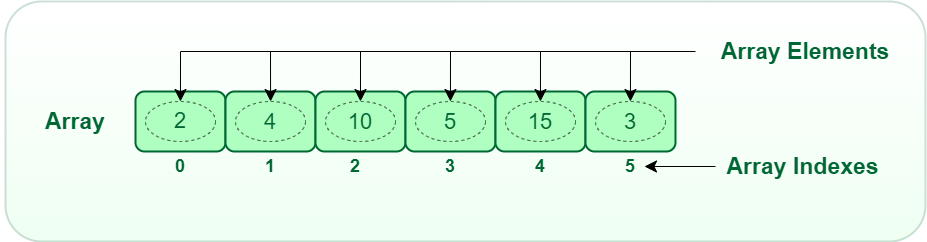
# II. Adatszerkezetek

Az algoritmusok és az adatszerkezetek a számítógépes program építő kövei. Az adatszerkezetek, adatstruktúrák a mi általunk megfogalmazott terv megvalósítására nyújtanak segítséget.

## II. 1. Szekvenciális Adatszerkezetek

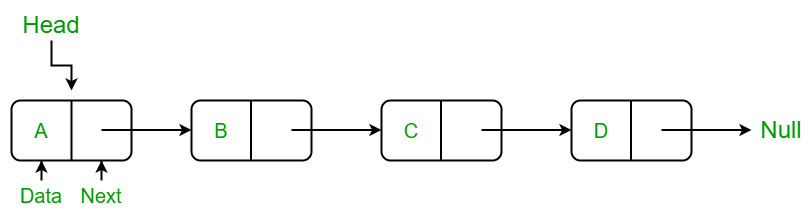
A szekvenciális adatszerkezetek olyan adatstruktúrák, amelyekben az elemek egy meghatározott sorrendben vannak elrendezve, és tipikusan egymás után következnek.

A **Tömbök**: **Szekvenciális** tárolást biztosítanak, ahol minden elemnek van egy meghatározott pozíciója (indexe). Elemekhez való **hozzáférés index alapján** történik, de a **méretük általában fix**.



3. ábra tömb

A **Listák**: **Dinamikus** adatszerkezetek (nem fix a méretük), nincs közvetlen hozzáférés az elemekhez, mutatókat használnuk. Minden értéknek megvan a saját memória címe. A lista első elemének helyét explicit módon meg kell határozni (HEAD). Hasonlóképpen, az utolsó elemnek tudnia kell, hogy nincs következő elem (azaz NULL).



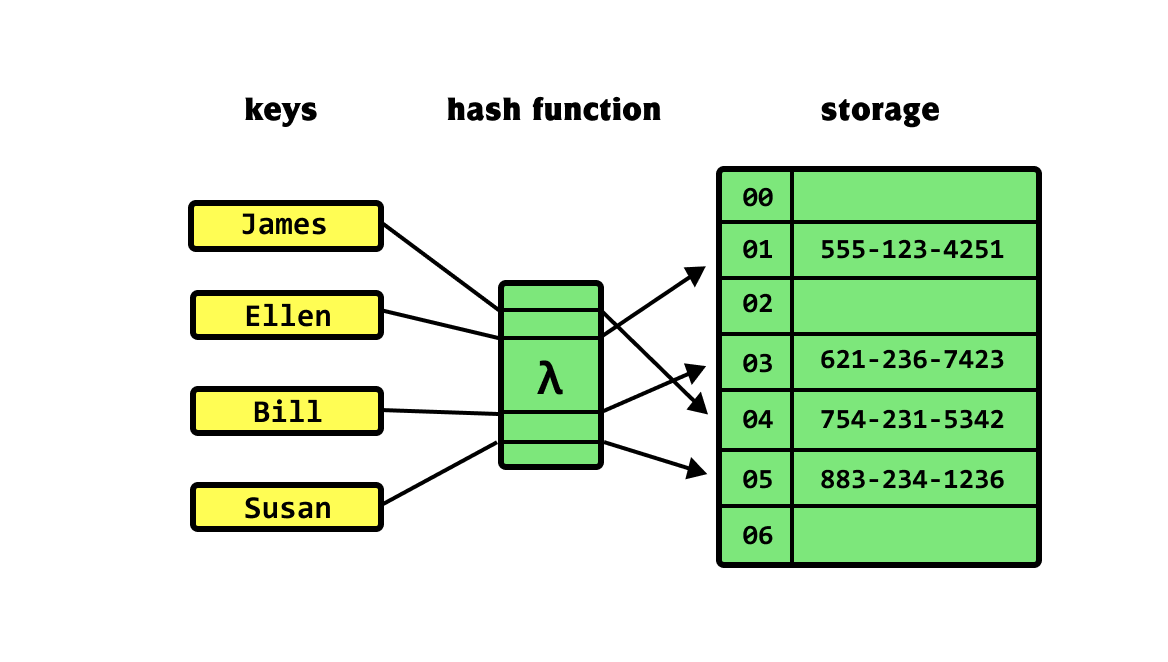
4. ábra lista, láncolt lista

Itt a koncepció nem az, hogy az elemeket folyamatosan tároljuk, mint a tömb esetében, hanem **hivatkozásokat** használunk, úgynevezett mutatókat *(pointerek)*. A **mutatók az értékek memória címeire mutatnak**.

## II. 2. Asszociatív Adatszerkezetek

Az asszociatív adatszerkezetek, más néven társító adatszerkezetek, olyan adattárolási módszerek, ahol az adatokat kulcs-érték párok formájában tárolják.

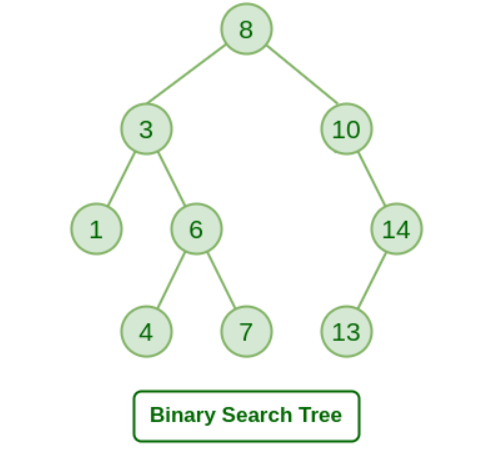
A **Hasító táblázat** (Hash táblák): **kulcs-érték párok** tárolására használják, ahol a kulcs egyedi azonosítója az adatnak. Gyors hozzáférési időt biztosítanak az elemekhez a kulcsuk alapján. A hash táblák alapelve, hogy egy **hash függvény** segítségével átalakítják a kulcsokat olyan indexekké, amelyek meghatározzák, hogy a tárolt elemek mely pozíciókban helyezkednek el a belső tömbben vagy listában.



5. ábra hash tábla

Ez a függvény ideális esetben elosztja egyenletesen az elemeket, hogy minimalizálja az ütközések számát. Mivel a hash függvény véges számú indexet generál, előfordulhat, hogy két különböző kulcs ugyanarra az indexre képződik le. Ezt az esetet ütközésnek nevezik, és különböző módszerek léteznek az ilyen helyzetek kezelésére, mint például a láncolás (a kulcs-érték párok láncolt listába szervezése egy indexen belül) vagy az újraképzés (másodlagos hash függvény alkalmazása).

A **Keresőfák** (pl. bináris keresőfa): **kulcs-érték párok rendezett tárolására használják**. Támogatják az adatok gyors keresését, beszúrását és törlését, megőrizve közben az adatok rendezettségét. Példál bináris keresőfa bemenet: {8, 3, 10, 1, 6, 4, 7, 14, 13}:



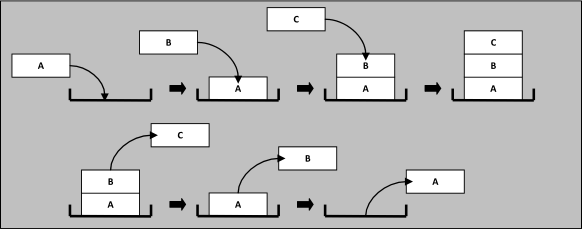
6. ábra keresőfa: bináris

Egy bináris keresőfában minden csomópontnak legfeljebb két gyermeke van, egy **bal** és egy **jobb** gyermek. A **bal** gyermek tartalmazza azokat az értékeket, amelyek **kisebbek** a szülő csomópontnál, a **jobb** gyermek pedig azokat, amelyek **nagyobbak**.

## II. 3. Gyűjtemények

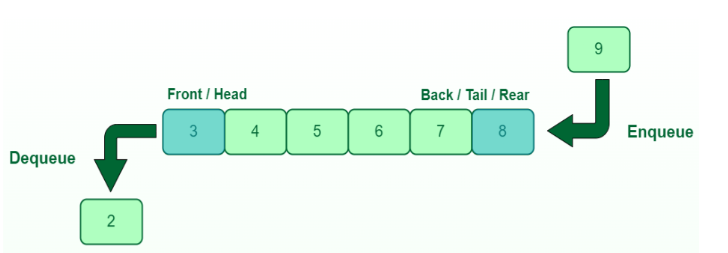
A gyűjtemények általános kategóriát jelentenek, amely magába foglal mindenféle adatszerkezetet, amely adatok csoportját tárolja. A gyűjtemények lehetnek szekvenciálisak vagy asszociatívak, attól függően, hogy hogyan szerveződnek az adatok

**Verem** (Stack, LIFO - Last In, First Out): egy olyan gyűjtemény, ahol az **utoljára hozzáadott elemet veszik ki először**. Ideális visszavonási műveletekhez, rekurzióhoz, és a hívási verem kezeléséhez. Egy olyan rendezést biztosít, amely az elemek a gyűjteményben (veremben) töltött idején alapul. Utasítások ***push*** (elhelyez), ***pop*** (kivesz), ***top*** (utoljára hozzáadott elem lekérdezése).



7. ábra LIFO elv, verem

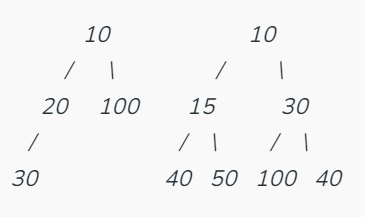
**Sor** (Queue, FIFO - First In, First Out): az **elsőként hozzáadott elemet veszik ki először.** Alkalmazásai közé tartozik a munkafolyamatok kezelése, üzenetvárólisták és szélességi bejárású keresés. Elemet hozzáadni a sorhoz az ***enqueue*** utasítással, míg törölni elemet a ***dequeue*** utasítással lehet



8. ábra FIFO elv, sor

**Speciális Adatszerkezetek**

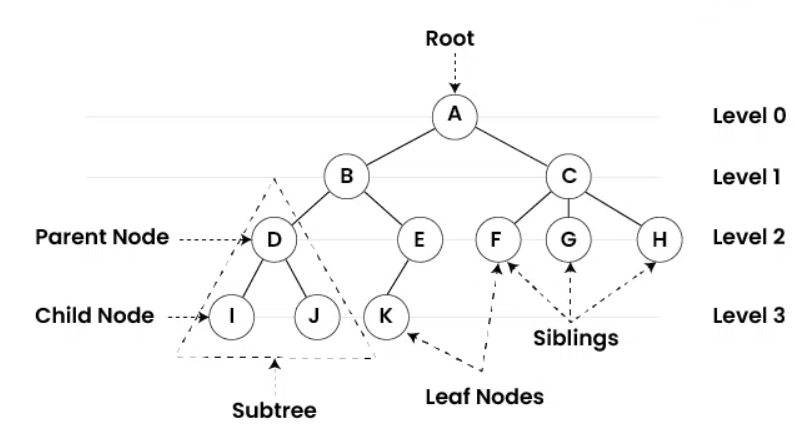
**Bináris kupac** (Binary Heap): teljes bináris fa[[1]](#footnote-1), amit gyakran használnak prioritási sorok implementálására. Támogatja a legkisebb vagy legnagyobb elem gyors elérését, ami hasznos számos algoritmusnál. (Például a Dijkstra algoritmus). Egy minimum bináris kupacban a gyökérben lévő kulcsnak a kupacban lévő összes kulcs közül a legkisebbnek kell lennie (ugyanez igaz a maximumra is).



9. ábra Bináris kupac

# III. Faműveletek

**Fák:** Hierarchikus rendszert képeznek az adatok tárolására és kezelésére. A fák segítenek az adatok szervezésében és hatékony lekérdezésében, így széles körben használatosak különböző alkalmazásokban, mint például adatbázis-kezelő rendszerek, fájlrendszerek, és sok más algoritmusban. Felépítése:



10. ábra Fa felépítése

**Keresés:** binárisfában a keresés egy adott elem megtalálását jelenti. A root-tól kezdjük és minden csomópontnál összehasonlítjuk a keresett értéket a csomópont értékével. Ha kisebb, akkor bal, ha nagyobb akkor jobb ágon folytatjuk. Addig folytatódik, amíg nem találjuk meg az értéket,vagy nem érünk levélhez. Példa: **VI. 2. Melléklet: Faművelet keresés**

**Beszúrás:** kereséshez hasonlóan járunk el. Addig folytatjuk, amíg nem érünk egy olyan pontot, ahol a beszúrandó elemet az adott helyen lévő üres (NULL) referencia helyére szúrhatjuk be. Példa: **VI. 3. Melléklet: Faművelet beszúrás**

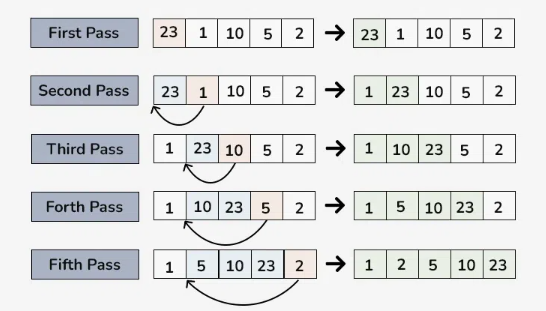
**Törlés:** három pont szerint kell eljárni. Egyik, ha a törlendő node-nak nincsenek gyermekei, akkor egyszerűen eltávolítjuk. Ha van egy gyermeke, akkor hasonló eljárás mint az előbb, csak a törlendő node jobb vagy bal, legkisebb vagy legnagyobb elemét mozgatjuk és távolítjuk el eredeti helyéről. Példa: **VI. 4. Melléklet: Faművelet törlés és kiegyensúlyozás**

# IV. Rendezések

A rendezés – vagy szortírozás – a számítástechnika egyik legalapvetőbb algoritmusa, és gyakran használt művelet a programozásban, illetve az adatbázisokban is. A rendezés az egy olyan folyamat, amikor egy gyűjtemény elemeit **valamilyen sorrendbe** helyezzük **bizonyos feltétel alapján**.

IV. 1. Beszúró rendezés

A beszúró rendezés (Insertion sort): minden iteráció során a rendezetlen rész első elemét beszúrjuk a már rendezett részlistánkba. Az elemek mozgatásával érjük el, hogy a megfelelő helyre kerüljön. Példa: {23, 1, 10, 5, 2}



**11**. ábra beszúró rendezés

**Első lépés**: Az aktuális elem 23. Az első elemet a tömbben rendezettnek tekintjük. Az 0. indexig rendezett rész: {23}

**Második lépés**: Összehasonlítjuk az 1-et a 23-mal (az aktuális elemet a rendezett résszel). Mivel az 1 kisebb, helyezzük az 1-et a 23 elé. Az 1. indexig rendezett rész: {1, 23}

**Harmadik lépés**: Összehasonlítjuk a 10-et az 1-gyel és 23-mal (az aktuális elemet a rendezett résszel). Mivel a 10 nagyobb mint 1, de kisebb mint 23, helyezzük a 10-et az 1 és 23 közé. A 2. indexig rendezett rész: {1, 10, 23}

**Negyedik lépés**: Összehasonlítjuk az 5-öt az 1-gyel, 10-zel és 23-mal (az aktuális elemet a rendezett résszel). Mivel az 5 nagyobb mint 1, de kisebb mint 10, helyezzük az 5-öt az 1 és 10 közé. A 3. indexig rendezett rész: {1, 5, 10, 23}

**Ötödik lépés**: Összehasonlítjuk a 2-t az 1-gyel, 5-tel, 10-zel és 23-mal (az aktuális elemet a rendezett résszel). Mivel a 2 kisebb mint az összes elem a rendezett részben, helyezzük a 2-t az elejére. A 4. indexig rendezett rész: {2, 1, 5, 10, 23}

Beszúrásos rendezés **idő** és **térbonyolultság**

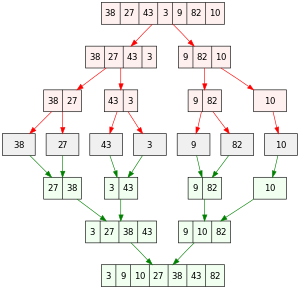
* Legjobb eset: O(n), ha a lista már rendezett, ahol a lista elemeinek száma n.
* Átlagos eset: O(n2), ha a lista véletlenszerűen rendezett.
* Legrosszabb eset: O(n2), ha a lista fordított sorrendben van.

Segédhely: O(1), a beszúrásos rendezés csak O(1) további helyet igényel, ezáltal ez egy helytakarékos rendezési algoritmus

IV. 2. Összefésülő rendezés

Az összefésülő rendezés (Merge sort): egy **rekurzív[[2]](#footnote-2)** rendezési módszer. Oszd meg és uralkodj[[3]](#footnote-3) elven működik. Addig bontjuk a bemenetet több kisebb részre, amíg lehetséges, egy elemű rész már rendezett. Majd az egye elemű, atomi tagokat visszafele építjük.

Nézzük meg az ábrán láthatópélda működését:



12. ábra összefésülő rendezés

**Felosztás:**

A {38, 27, 43, 3, 9, 82, 10} listát felosztjuk {38, 27, 43, 3} és {9, 82, 10} részlistákra. A {38, 27, 43, 3} listát tovább osztjuk {38, 27} és {43, 3} részlistákra. Ezután a {38, 27} lisát is tovább bontjuk {38} és {27} részlistákra, majd elvégezzük ugyanezt a {43, 3} listán is, ami {43} és {3} részlista lesz.

A {9, 82, 10} listát tovább osztjuk {9, 82} és {10} részlistákra. Végül a {9, 82} tovább bontjuk {9} és {82} rész listára.

Összefésülés: Hasonlítsuk össze a részlistákban lévő elemeket a szomszédos elemekkel (balról jobbra haladva).

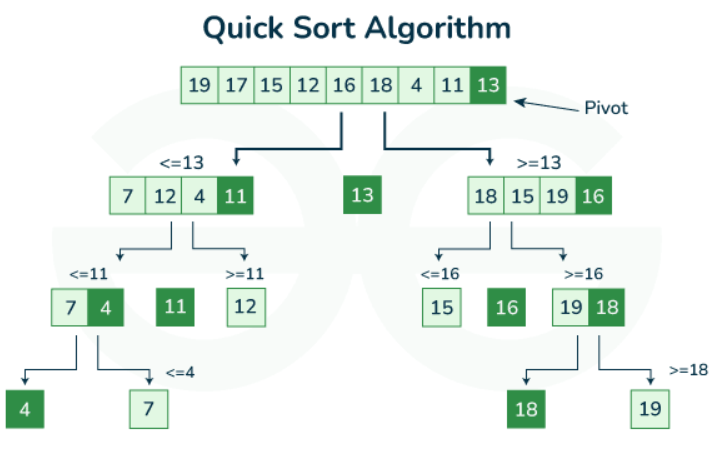
Összefésülő rendezés **idő** és **térbonyolultság**

* Legjobb eset: O(n log n), amikor a lista már rendezett vagy majdnem rendezett.
* Átlagos eset: O(n log n), ha a lista véletlenszerűen rendezett.
* Legrosszabb eset: O(n log n), ha a lista fordított sorrendben van.

Segédhely: O(n), további hely szükséges az összefésülés során használt ideiglenes tömbhöz.

## IV. 3. Gyorsrendezés

A gyorsrendezés (Quick sort): (tetszés szerint) a pivot elem kiválasztása után a tömböt **kettévágjuk**. Az egyik felében a pivotnál **kisebb**, a másikban a pivotnál **nagyobb** elemek vannak. Ezt rekurzívan ismételjük a résztömbökön. Hasonló az összefésüléshez.



13. ábra gyorsrendezés

Gyorsrendezés **idő** és **térbonyolultság**

* Legjobb eset: O(n log n), amikor a pivot elemek jól vannak kiválasztva, lehetővé téve a lista egyenletes felosztását.
* Átlagos eset: O(n log n), amikor a pivot elemek véletlenszerű kiválasztása miatt a lista átlagosan egyenletesen oszlik meg.
* Legrosszabb eset: O(n2), amikor a pivot elemek rosszul vannak kiválasztva, például a lista már rendezett vagy fordított sorrendben van.

Segédhely: - általában - O(log n) a rekurziós hívások miatt szükséges veremhely.

## IV. 4. Kupacrendezés

A kupacrendezés (Heap sort) a tömb elemeiből először minimum vagy maximum kupacot építünk. A **csúcsban** található elemet **eltávolítjuk** és a **tömb** **végére** helyezzük. Addig ismételjük ezt az eljárást, amíg a kupac ki nem ürül. Példa: **VI. 5. Melléklet: Kupacrendezés**

Kupacrendezés **idő** és **térbonyolultsága**:

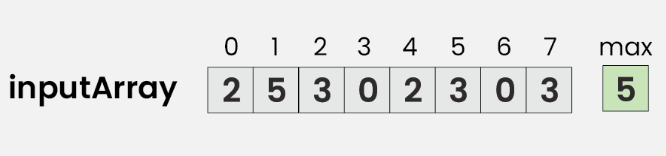
* Legjobb eset: O(n log n), mivel a kupacrendezés minden esetben azonos módon osztja fel a listát.
* Átlagos eset: O(n log n), mivel a kupac felépítése és a legnagyobb elemek kiválasztása ismétlődő logaritmikus lépéseken keresztül történik, függetlenül a lista elemeinek kezdeti sorrendjétől.
* Legrosszabb eset: O(n log n), a kupacrendezés teljesítménye nem függ az elemek kezdeti sorrendjétől, mivel a kupac felépítése és az elemek kiválasztása mindig logaritmikus időt vesz igénybe.

Segédhely: O(1), mivel a kupacrendezés helyben történik, és nem igényel jelentős extra tárhelyet a rekurziós hívásokhoz, ellentétben a gyorsrendezéssel.

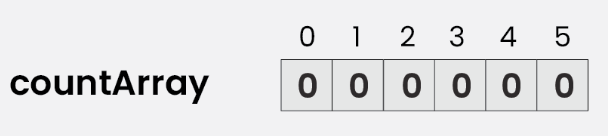
## IV. 5. Leszámoló rendezés

A leszámoló rendezés (Counting sort): A leszámoló rendezés alapötlete az, hogy megszámolja minden egyes különböző elem előfordulási gyakoriságát a bemeneti tömbben, és ezt az információt használja fel az elemek helyes, rendezett pozícióba helyezéséhez. Példa: {2, 5, 3, 0, 2, 3, 0, 3}

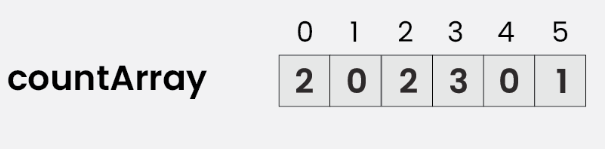
**Első lépés**: Keressük meg a tömb maximumát. A **maximum** érték 5.



**Második lépés**: Hozzunk létre egy **új tömböt** úgy, hogy a hossza a **maximum** + 1 legyen. A tömb minden eleme 0. Itt fogjuk tárolni az előfordulások számát.



**Harmadik lépés**: Az új tömbben (countArray) tároljuk, hogy az eredeti tömbben lévő elemek hányszor szerepeltek. Például a 0 érték az eredeti tömben (inputArray) kétszer szerepel, tehát a countArray tömb 0. indexére egy kettest írunk.

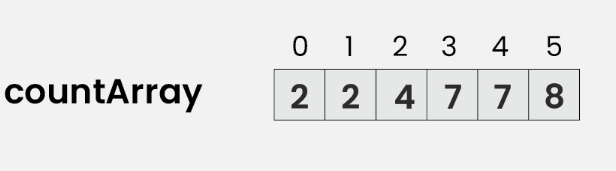


**Negyedik lépés**: Tároljuk a countArray elemeinek kumulatív összegét vagy prefix összegét úgy, hogy a **countArray[i]** = **countArray[i – 1] + countArray[i]** képlettel számoljuk. Példa:

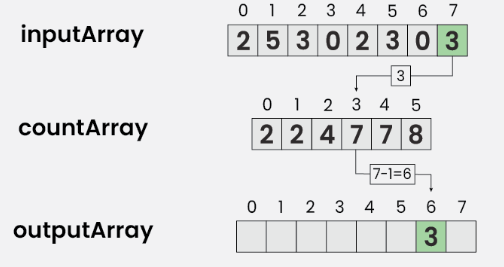
1. index értéke: 0. indexen lévő + aktuális index => 0 + 2 = 2

2. index értéke: 1. indexen lévő + aktuális index => 2 + 2 = 4

így tovább...



**Ötödik lépés**: Haladjunk végig a bemeneti tömb végétől kezdve, mert a bemeneti tömb végéről előre haladva megőrizzük az azonos elemek sorrendjét, ami végül stabilá teszi ezt a rendezési algoritmust.



Teljes levezetés részletesebb leírással: https://www.geeksforgeeks.org/counting-sort/

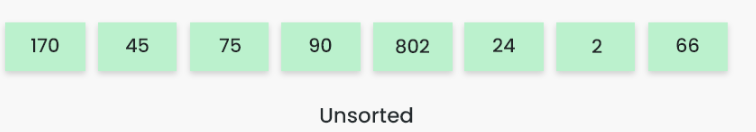
Leszámoló rendezés **idő**- és **térbonyolultsága**:

* Legjobb eset: O(n + k), mivel a számláló rendezés az elemeket a bemeneti tömbből közvetlenül a kimeneti tömbbe helyezi a számolt indexek alapján.
* Átlagos eset: O(n + k), ahol 'n' az elemek száma, és 'k' a különböző értékek tartománya. Ez azért van, mert minden elemet egyszer számolunk meg és helyezünk el.
* Legrosszabb eset: O(n + k), mivel a számláló rendezés nem függ az elemek kezdeti sorrendjétől, és minden elemet pontosan egyszer kell megvizsgálni és elhelyezni.

Segédhely: O(k), mivel szükség van egy segédtömbre a különböző értékek számának tárolására. Ez különösen akkor igényel több helyet, ha a bemeneti értékek tartománya nagy.

## IV. 6. Radix rendezés

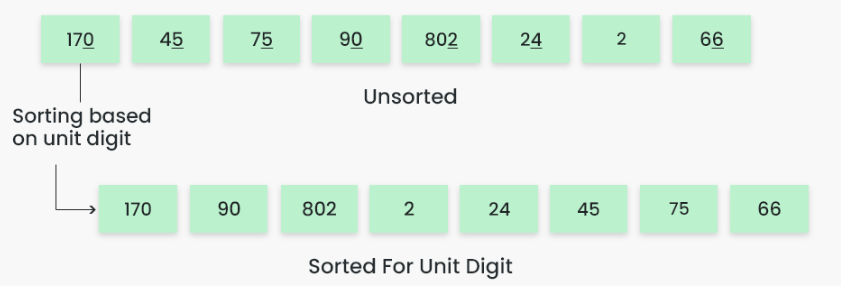
A radix rendezés (Radix sort): a számokat **helyiértékek szerint rendezzük**. A rendezés az utolsó számjegyek/karakterek szerint történik. Ha ez megtörténik, akkor a következő szerint (hátulról a második.) zajlik a rendezés, amíg el nem érjük a legelsőt. Példa: {170, 45, 75, 90, 802, 24, 2, 66}



**Első lépésben** keressük meg a tömb legnagyobb elemét, ami 802. Ez az elem számjegyből áll, így háromszor fogunk iterálni, minden jelentős helyiértékhez egyszer.

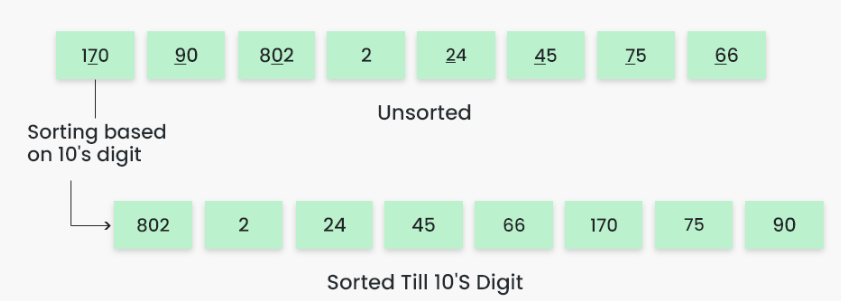
**Második lépésben** rendezzük az elemeket az egyes helyiértéken lévő számjegyek alapján (X=0). Használjunk egy stabil rendezési technikát, például számláló rendezést.

Rendezés az egyes helyiérték alapján: Végezzünk számláló rendezést a tömbön az **egyes** helyiértéken lévő számjegyek alapján. Az egyes helyiérték alapján rendezett tömb: {170, 90, 802, 2, 24, 45, 75, 66}.



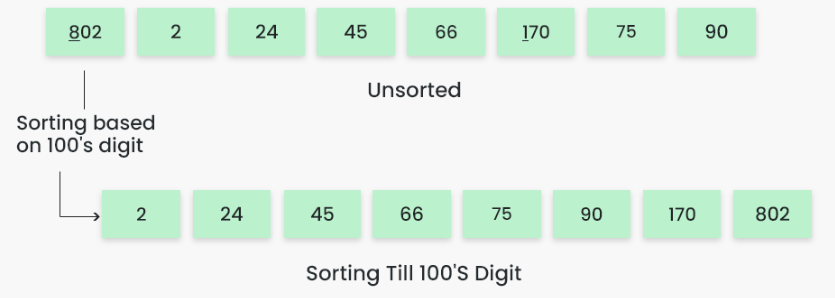
**Harmadik lépésben** rendezzük az elemeket a **tizes** helyiértéken lévő számjegyek alapján (tehát hátulról a második).

Rendezés a tizes helyiérték alapján: Ismét végezzünk számláló rendezést a tömbön a tizes helyiértéken lévő számjegyek alapján. A tizes helyiérték alapján rendezett tömb: {802, 2, 24, 45, 66, 170, 75, 90}.



**Negyedik lépésben** rendezzük az elemeket a **százas** helyiértéken lévő számjegyek alapján (mondhatni az utolsó).

Rendezés a százas helyiérték alapján: Végezzünk számláló rendezést a tömbön a százas helyiértéken lévő számjegyek alapján. A százas helyiérték alapján rendezett tömb: {2, 24, 45, 66, 75, 90, 170, 802}.



A bemeneti tömbünk így már rendezett. Vegyük észre, hogy a rendezést annyiszor kell elvégezni, ahány helyiértéket vizsgálunk (pl. 3 helyérték esetén 3 iteráció)!

Radix rendezés **idő**- és **térbonyolultsága**

* Legjobb eset: O(nk), ahol n az elemek száma, k pedig a számjegyek maximális száma a rendezendő számokban.
* Átlagos eset: O(nk), mivel a Radix Sort független az elemek kezdeti sorrendjétől, és a k számjegyek számától függ.
* Legrosszabb eset: O(nk), a Radix Sort időbonyolultsága megegyezik a legjobb és átlagos esettel, mivel minden elemet számjegyenként meg kell vizsgálni.

Segédhely: O(n + b), ahol *b* a számrendszer alapja (például 10 a decimális számok esetében).

# V. Gráfok, és gráfalgoritmusok

## V. 1. Gráfok

A gráf csomópontok, csúcsok és rajtuk értelmezett összeköttetések (élek) véges halmaza.

* Ha gráf irányítatlan, nem teszünk különbséget „A-ból B-be”, illetve „B-ből A-ba” menő élek között.
* Ha a gráf irányított a két iránynak irányított élek felelnek meg. A 🡪 B

## V. 2. Bejárások

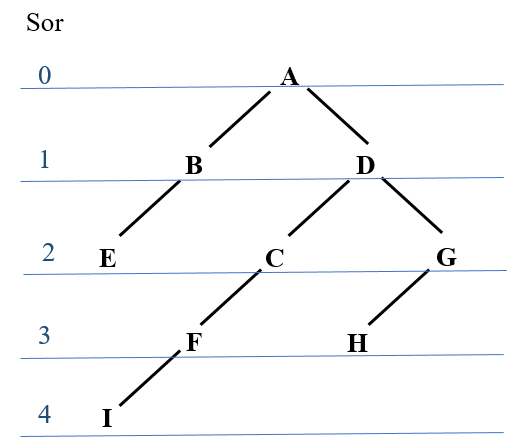
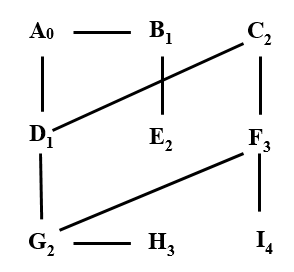
Veszünk egy csúcsot, és az éleken át haladva felderítjük a további csúcsokat. Amennyiben egybefüggő a gráf és nincsen benne szakadás, akkor minden csúcsot feltudunk kutatni.

Módszerei: Szélességi és mélységi. A gráf feltérképezésére szolgáló eljárások.

**Szélességi bejárás:** Van egy kezdeti (A) csúcs. A kezdeti csúcsnak felkutatom a szomszédjait, akik közvetlenül csatlakozik hozzá az éleken át haladva. Majd a szomszédoknak is felkutatom a közvetlen szomszédjait. Minden körben egy szinttel távolabb kerülünk a kiindult csúcstól.

Eljárás: Meghatározza a **legrövidebb** utat, hány lépésből áll. Mindig sorban megy. Mindig balról töltjük fel. Ha csak egy gyerek van, akkor mindig baloldalra kerül.

Példa:

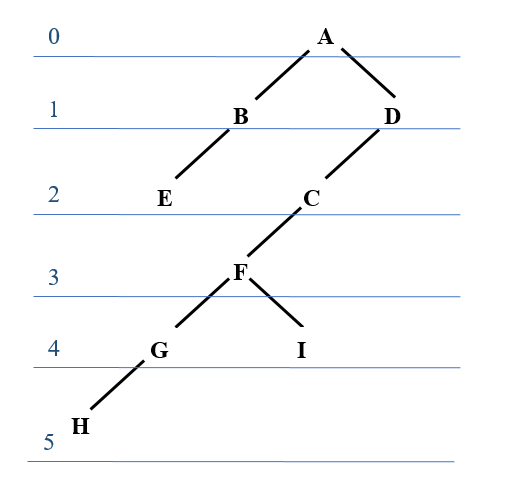
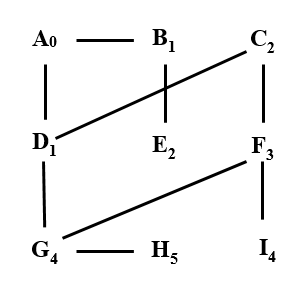


Van N darab csúcsom. A csúcsok között fut él. NxN-es **szomszédsági mátrixban** tudom tárolni a csúcsok között futó éleket. A 0 érték jelölje, hogy nem fut él, míg az 1 érték jelölje, hogy fut él a két csúcs között.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **A** | **B** | **C** | **D** | **E** | **F** | **G** | **H** | **I** |
| **A** | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| **B** | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| **C** | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| **D** | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| **E** | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **F** | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| **G** | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| **H** | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| **I** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

**Mélységi bejárás**: Egy szomszédot derítünk fel. Első szomszéd után felderítjük annak is az első szomszédját. Nem derítjük fel az összesét egyszerre. A lehető legmélyebbre próbálunk menni. Ha elértem egy olyan pontot, amelynek már nincs szomszédja, akkor visszalépek és megyek a következő szomszédra.

Eljárás: Meghatározza az induló ponttól lévő **legmesszebb** lévőt. Az első ágon rögtön végig lemegyek. Elakadás esetén egyet visszalépek, és ott folytatom, ahol még nem jártam. Az algoritmus véget ér, ha visszaérünk a gyökérbe és nincs más csúcs.



A legmesszebb lévő elem a H az A pontból.

|  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
|  | **A** | **B** | **C** | **D** | **E** | **F** | **G** | **H** | **I** |
| **A** | 0 | 1 | 0 | 1 | 0 | 0 | 0 | 0 | 0 |
| **B** | 1 | 0 | 0 | 0 | 1 | 0 | 0 | 0 | 0 |
| **C** | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 0 | 0 |
| **D** | 1 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 0 |
| **E** | 0 | 1 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| **F** | 0 | 0 | 1 | 0 | 0 | 0 | 1 | 0 | 1 |
| **G** | 0 | 0 | 0 | 1 | 0 | 1 | 0 | 1 | 0 |
| **H** | 0 | 0 | 0 | 0 | 0 | 0 | 1 | 0 | 0 |
| **I** | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 1 |

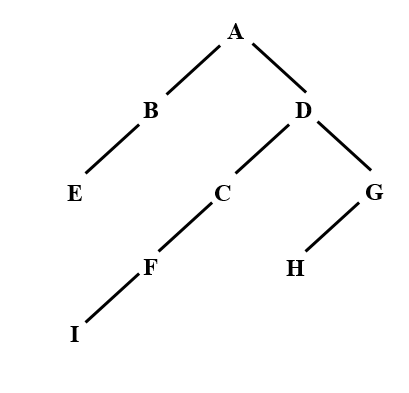
## V. 3. Keresések

P – szülő L – balgyermek R – Jobbgyermek

* Pre-order (**P**LR : szülő, bal, jobb
* In-order (L**P**R): bal, szülő, jobb
* Post-order (LR**P**): bal, jobb, szülő

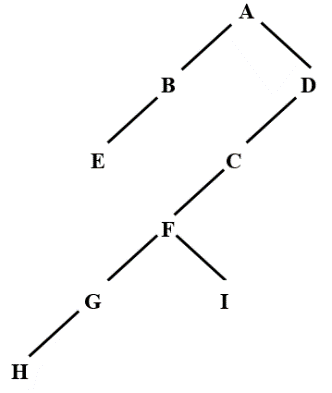
Nézzük meg az előző példán a Pre-In-Post bejárásokat.

Első lépésben készítsük el a szélességi és mélységi kereséseket, majd azokon alkalmazzuk a bejárásokat.

Szélesség

Keresések:

* Pre: ABEDCFIGH
* In: EBAIFCDHG
* Post: EBIFCHGDA

Mélységi

Keresések:

Pre: ABEDCFGHI

In: EBAHGFICD

Post: EBHGIFCDA

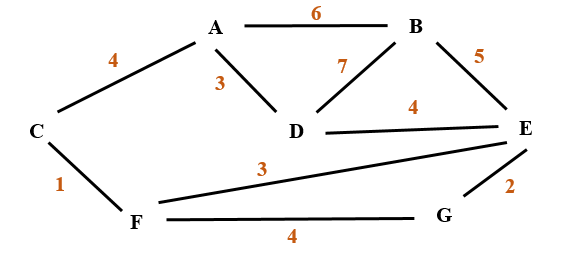
## V. 4. Feszítőfák

Az eredeti gráf **részgráfja** a **feszítőfa**. Egy **súlyozott gráf**. A feszítőfa tartalmazza az eredeti gráf minden csúcsát. Az összefüggő gráfok esetén összefüggő fáról (**körmentes**) beszélünk. Az eredeti élekből N-1 élet tartalmaz. Egy gráfnak több feszítőfája is lehet.

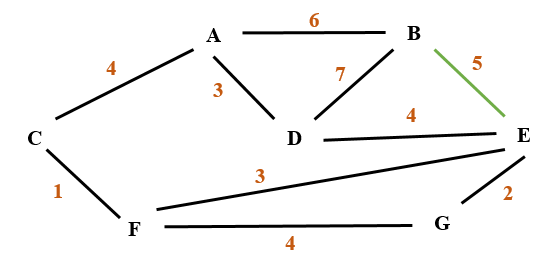
A Prim és Kruskal algoritmusokat alkalmazzák a minimális költségű feszítőfa keresésére.

## V. 5. Prim algoritmus

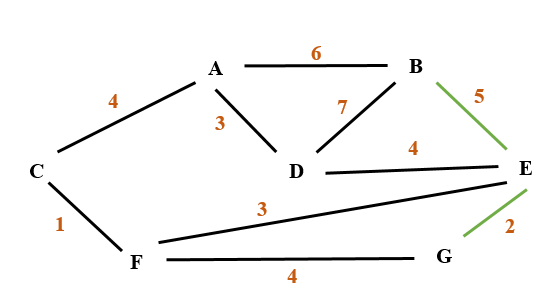
Legyen a **megadott kezdő csúcs B** az alábbi gráfon:



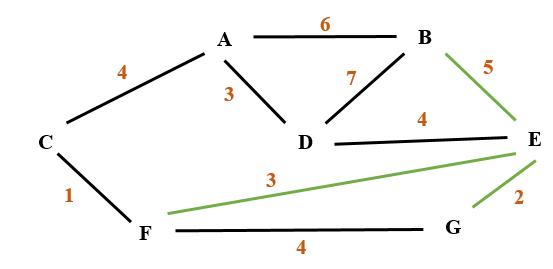
* Mindig a legkisebb költséges választjuk.
* Ha egyenlő a költség, akkor tetszőlegesen választunk.
* Végig **fának kell maradnia a még az algoritmus közben is!**
* Kör nem lehet benne! A feszítőfa körmentes.
* N csúcs esetén N-1 élnek kell lennie.
* B-ből a legkisebb költség az E út.



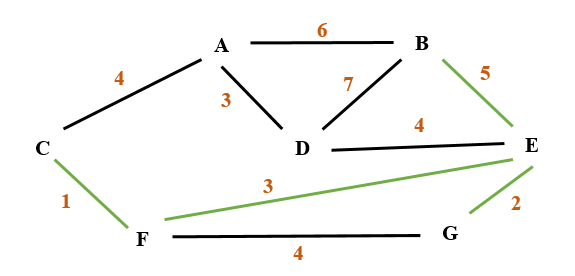
Az E csúcsból a G-be a legkevesebb költség.



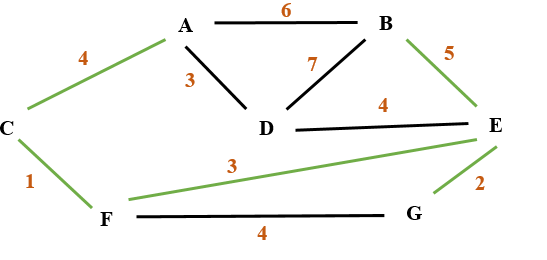
Az E csúcsból F olcsóbb, mint a G csúcsból F út.



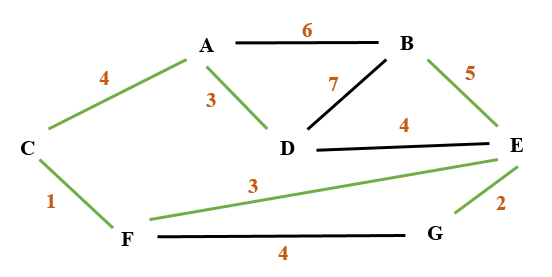
Az F csúcsból C csúcs.



A C csúcsból A csúcs.



Az A csúcsból D csúcs.



Ezzel megkaptuk a minimális költségű feszítőfát. Ha bármely más útvonalon folytatnánk, kört kapnánk!

BE – 5, EG – 2, EF – 3, FC – 1, CA – 4, AD – 3

Összegezzük a költséget: 5 + 2 + 3 + 1 + 4 + 3 = 18

7 csúcsunk van és 7-1 azaz 6 élt választottunk.

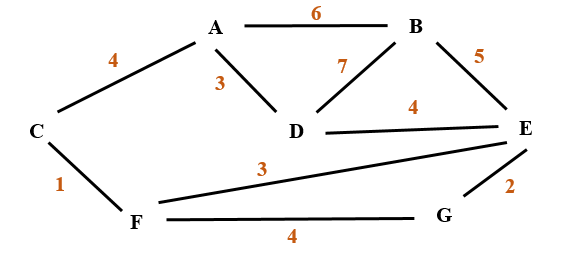
Ugyanúgy felírható egy szomszédsági mátrix, azzal a különbséggel, hogy a csúcsok között az éleken lévő költség szerepel, mint érték

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
|  | **A** | **B** | **C** | **D** | **E** | **F** | **G** |
| **A** | 0 | 6 | 4 | 3 | 0 | 0 | 0 |
| **B** | 6 | 0 | 0 | 7 | 5 | 0 | 0 |
| **C** | 4 | 0 | 0 | 0 | 0 | 1 | 0 |
| **D** | 3 | 7 | 0 | 0 | 4 | 0 | 0 |
| **E** | 0 | 5 | 0 | 4 | 0 | 3 | 2 |
| **F** | 0 | 0 | 1 | 0 | 3 | 0 | 4 |
| **G** | 0 | 0 | 0 | 0 | 2 | 4 | 0 |

## V. 6. Kruskal algoritmus

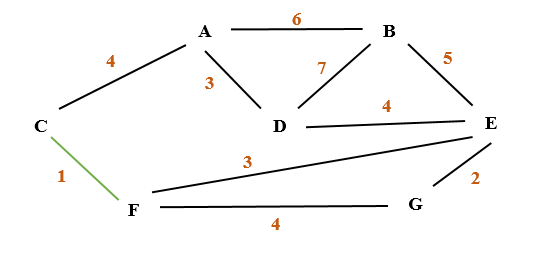
Az algoritmus nagyon hasonló a Prim-hez, a lényegi különbség, hogy a legkevesebb súlyú él keresése során **nem kötelező fának maradnia**! A többi tulajdonság ugyanúgy vonatkozik rá is.

Az előző példán nézzük meg a Kruskal-t

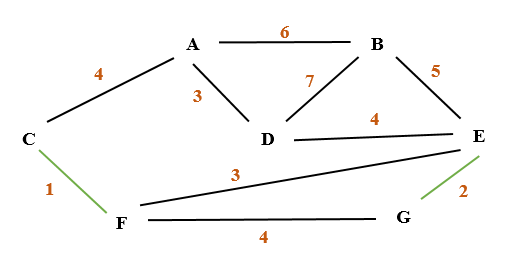


Jelenleg nincs megadva kezdőcsúcs, válasszuk ki az F csúcsot.

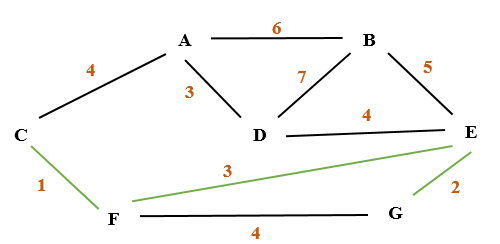
Az CF élt válasszuk.



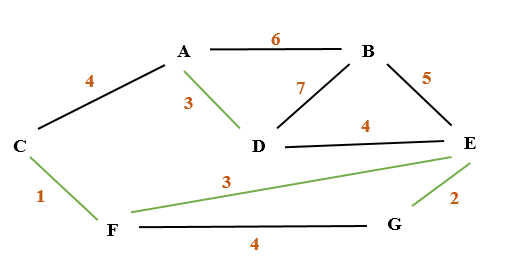
A következő legolcsóbb él csúcsok között az EG. Ahogy fentebb olvasható volt, nem kötelező fának maradnia az algoritmus során.



Következő az EF ( lehetne AD is).

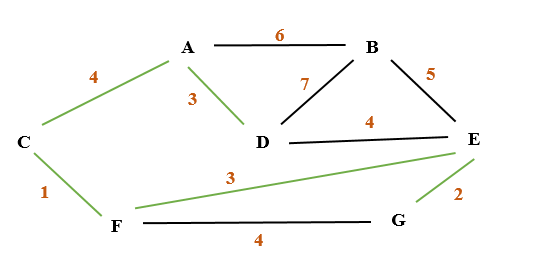


Következő az AD.

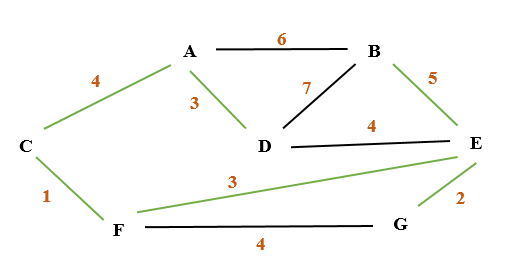


A DE és az FG él között futó élek nem opciók, mivel akkor nem körmentes lenne a fa!

A választás az AC.



Az utolsó él pedig a BE csúcs között van.



Ellenőrizzük, hogy 7 élünk van és N-1 darab él kell, hogy színezve legyen.

CF – 1, EG – 2, EF – 3, AD – 3, CA – 4, BE – 5

Összköltség: 1 + 2 + 3 + 3 + 4 + 5 = 18

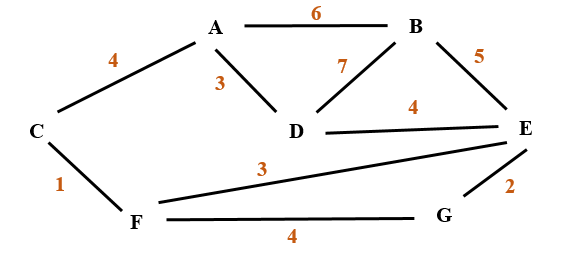
A Kruskal egy monoton növekvő algoritmus. A szomszédsági mátrixot hasonlóképpen fel lehet írni, mint a Prim algoritmusnál.

## V. 7. Legrövidebb utak

**Dijkstra algoritmus**

Egy [mohó algoritmus](https://hu.wikipedia.org/wiki/Moh%C3%B3_algoritmus), amivel irányított vagy irányítás nélküli [gráfokban](https://hu.wikipedia.org/wiki/Gr%C3%A1f) lehet megkeresni a **legrövidebb utakat** egy adott csúcspontból kiindulva. A súlyok nem negatívak. Az algoritmus inputja egy súlyozott G [gráf](https://hu.wikipedia.org/wiki/Gr%C3%A1f).

Nézzünk példát az előző gráf alapján.



A cél, hogy G-ből eljussunk A csúcsba a legrövidebb, minimális költségű úton.

Készítsünk egy táblázatot

|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Lépések** | **A** | **B** | **C** | **D** | **E** | **F** | **G** | **Hova jutottunk** |
| 0 | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | 0 | G |
| 1 | ∞ | ∞ | ∞ | ∞ | (2, G) | (4, G) |  | E |
| 2 | ∞ | (7, E) | ∞ | (6, E) |  | (5, E) |  | F |
| 3 | ∞ | (7, E) | (6, F) | (6, E) |  |  |  | C |
| 4 | (10, E) | (7, E) |  | (6, E) |  |  |  | D |
| 5 | (9, D) | (7, E) |  |  |  |  |  | B |
| 6 | (9, D) |  |  |  |  |  |  | A |

Jelmagyarázat:

**∞** - az adott csúcsból nem tudunk eljutni, az út elérhetetlen. Változhat az érték. Sorra változni fog, amikor egyik csúcsból a másikba haladunk.

**0.** lépésben az G csúcsból indulunk, minden más csúcsra végtelen írunk. A „Hova” ezáltal G lesz.

**1.** lépésben felírjuk az G csúcs szomszédjait, tehát hova tudunk eljutni. Jelölés (költség, honnan megyünk). Mindig a legolcsóbb utat választjuk, tehát az az E-be a legolcsóbb eljutni.

**2.** lépésben itt már E csúcsban vagyunk, tehát ahova eltudunk jutni azt hozzá írjuk a táblázathoz. FONTOS, hogy az előző út költségét, hozzáadjuk a táblázat költségeihez. A többit lemásoljuk, ha nem találunk jobb útvonalat a E-ből. Az E csúcsból eljuthatunk az F-be és az a legolcsóbb is.

**3.** lépésben F csúcsból C-be a legolcsóbb.

**4.** lépésben azt látjuk, hogy bár az A csúcsba eltudunk jutni ahova a feladat szól, de lehet, hogy van olcsóbb útvonal ha tovább keresgélünk az E csúcsnál. Válasszuk a D csúcsot következőként.

**5.** lépés itt a költségek visszaállnak az E csúcsba jutott költségekre. Ha egy csúcsot megközelíthetünk olcsóbban, akkor annak előző értékét felülírjuk például a (10, A) drágább, mint a (9, A). Olcsóbb útvonalat leírhatunk, de drágábbat nem!

**6.** lépés itt szemmel látható, hogy a D-ból jutunk el a legolcsóbban az A csúcsba, de folytassuk az algoritmus kedvéért a „mindig az olcsóbbat választjuk” résszel. Menjünk B-be E csúcsból.

**7.** lépésben látható, hogy a legolcsóbb út G csúcsból A csúcsba 9.

A lépéseket feltudjuk írni, hogy mely csúcsokon mentünk át. A táblázatot alulról olvassuk:

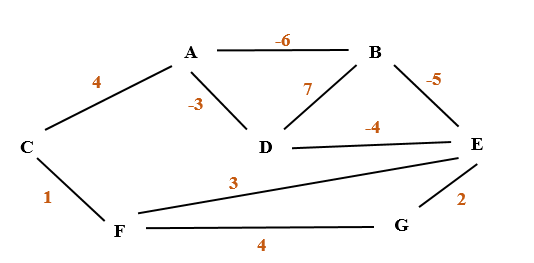
G 🡨 E 🡨 D 🡨 A

**Bellman-Ford algoritmus**

A Bellman-Ford-algoritmus az adott kezdőcsúcsból induló legrövidebb utak problémáját abban az általánosabb esetben oldja meg, amikor az élek között negatív súlyúakat is találhatunk.

A Bellman-Ford-algoritmus egy logikai értéket ad vissza annak jelölésére, hogy van vagy nincs a kezdőcsúcsból elérhető negatív kör. Ha van ilyen kör, az algoritmus jelzi, hogy nem létezik megoldás. Ha nincs ilyen kör, akkor az algoritmus előállítja a legrövidebb utakat és azok súlyait.

Használjuk az előző feladatot példaként, de a súlyok egy része negatív.



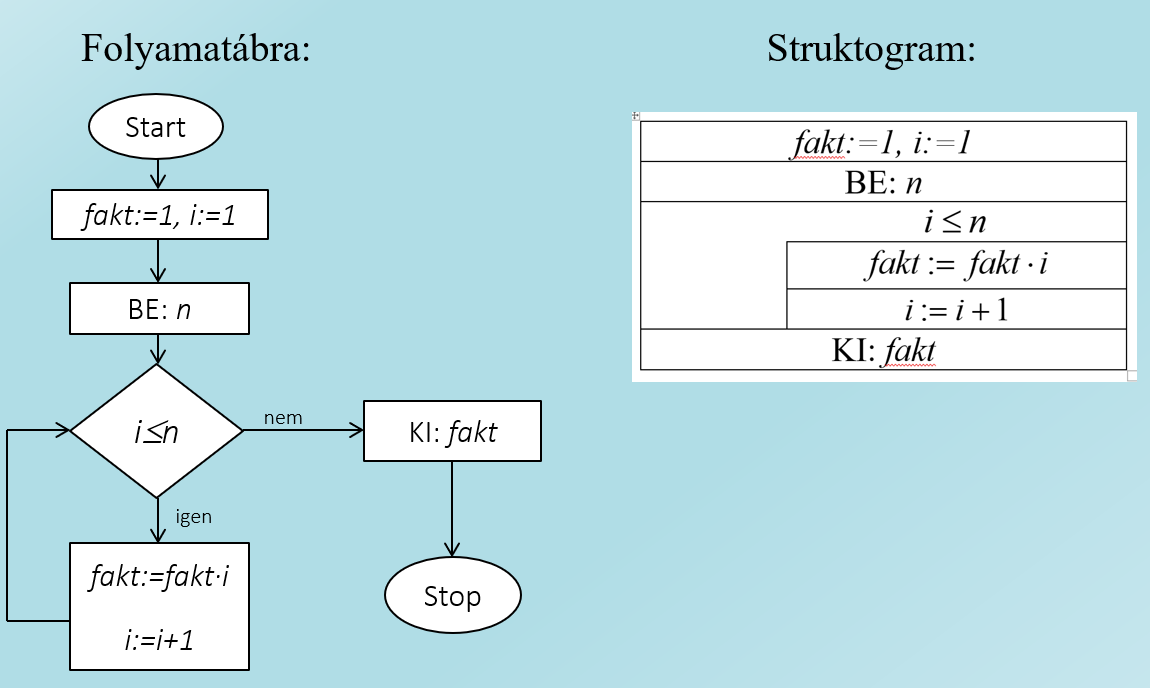
Az eljárás nagyon hasonló a Dijkstra algoritmuséhoz.

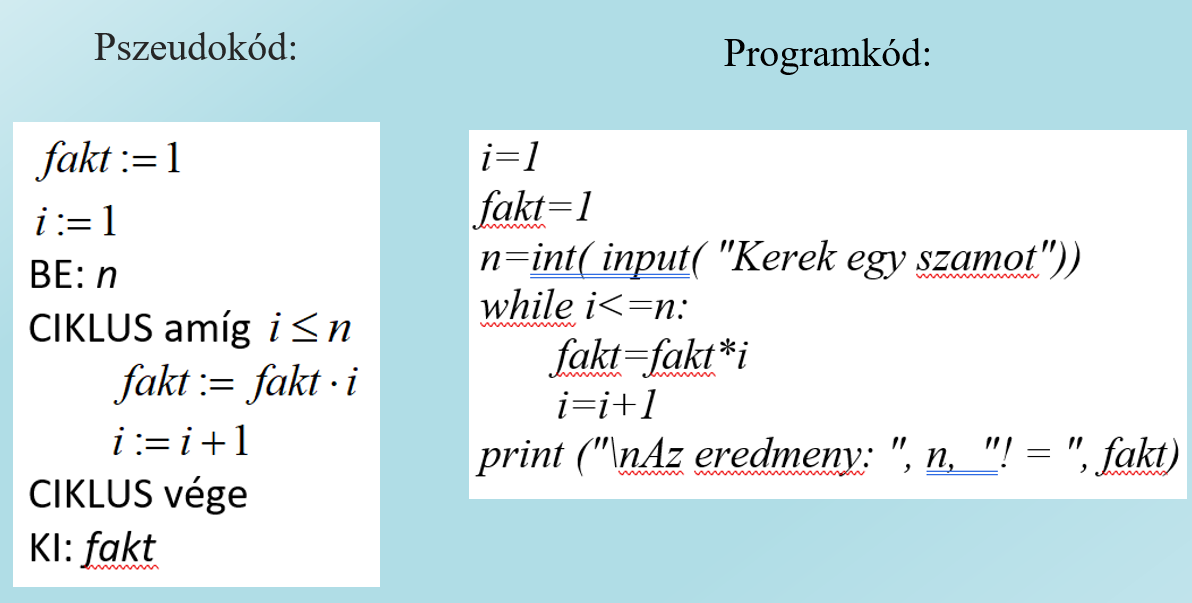
|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| **Lépések** | **A** | **B** | **C** | **D** | **E** | **F** | **G** | **Hova jutottunk** |
| **0** | ∞ | ∞ | ∞ | ∞ | ∞ | ∞ | 0 | G |
| **1** | ∞ | ∞ | ∞ | ∞ | (2, G) | (4, G) |  | E |
| **2** | ∞ | (-3, E) | ∞ | (-2, E) |  | (4, G) |  | D |
| **3** | (-5, D) | (-3, E) | ∞ |  |  | (4, G) |  | A |
| **4** |  | (-11, A) | (-1, A) |  |  | (4, G) |  | B |

Az eljárás hasonló a Dijkstra algoritmushoz, különbség az, hogy itt vannak negatív súlyú utak. Ezek „jutalom” utak is lehetnek. A fenti táblázat folytatható, bár nem érdemes. Ha tovább folytatnánk, látnánk, hogy egy LOOP-ba kerültünk, ahol a negatív súlyokat folyamatosan tudjuk növelni, ezáltal az úti költséget csökkenteni. A negatív kör elérhető a kezdőcsúcsból így nincs megoldás!

# VI. Melléklet

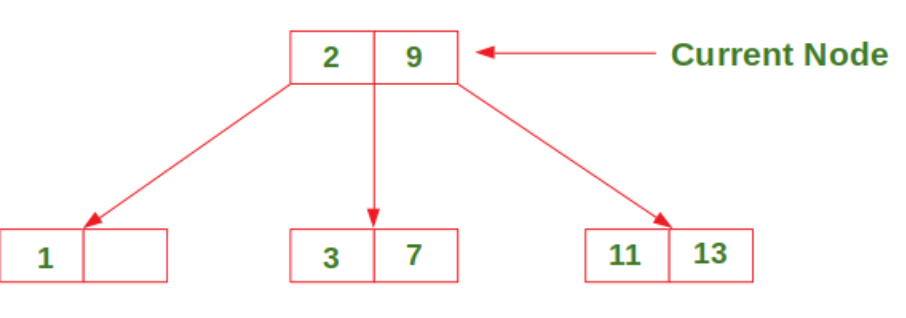
## VI. 1. Melléklet: algoritmus ábrázolás



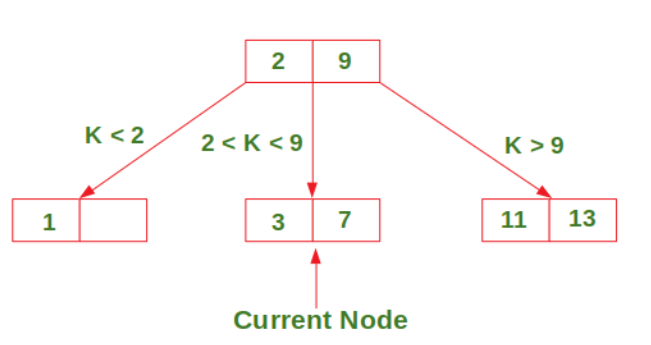


## VI. 2. Melléklet: Faművelet keresés

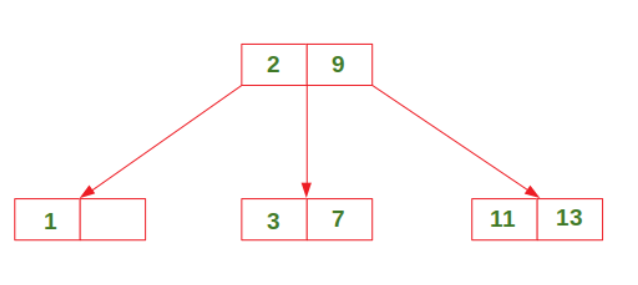
Feladat: Keressük meg az 5-ös számot, ha megtaláljuk igaz értékkel térjünk vissza, ha nem akkor hamis!



Tegyük fel, hogy K jelöli a jelenleg vizsgált csomópontot:



Megvizsgáljuk, hogy a keresett elem az 5, hol helyezkedhet el. A választás a 2-nél nagyobb, de kilencnél kisebb csomópontban folytatjuk a vizsgálatot.



Nem találtuk meg az 5-ös számú elemet, így térjen vissza HAMIS értékkel!

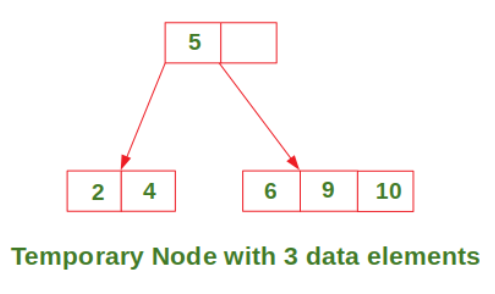
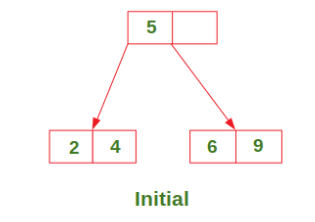
## VI. 3. Melléklet: Faművelet beszúrás

Az alábbiakban három lehetséges esetet tárgyalunk, amelyek beszúráskor fordulhatnak elő:

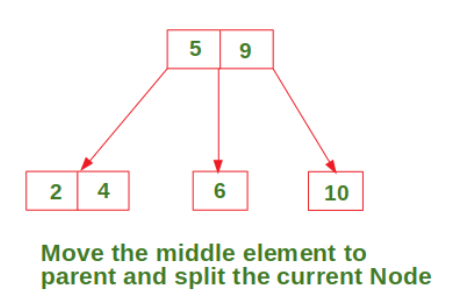
**Első eset**: Beszúrás egy olyan csomópontba, amely csak egy adatelemet tartalmaz. Példa a 4-es elemet szúrjuk be:



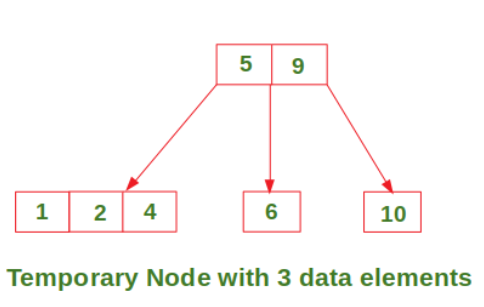
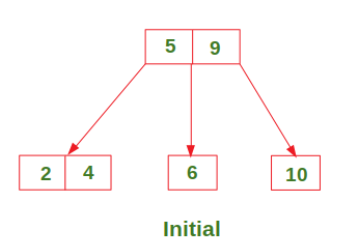
**Második eset**: Beszúrás egy olyan csomópontba, amely két adatelemet tartalmaz, és amelynek szülője csak egy adatelemet tartalmaz. Példa: szúrjuk be a 10-es elemet:



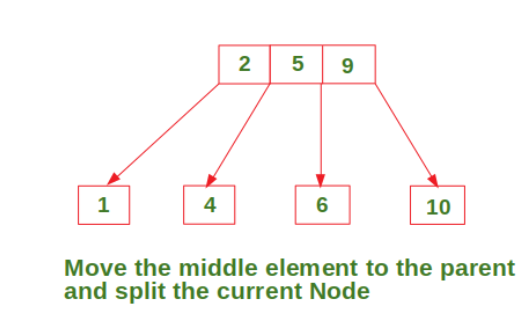
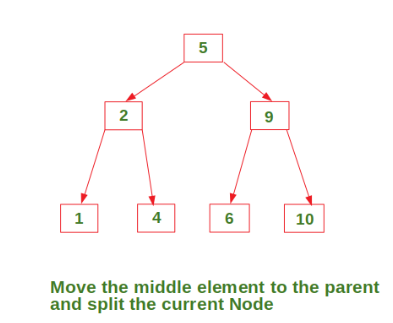
A csomópont beszúrás után három elemet tartalmaz. Szétszedjük a a hármas csomópontot úgy, hogy a középső elem (kilences) felkerül a a szülőbe. A 9,10 elemeket felbontjuk:



**Harmadik eset**: Beszúrás egy olyan csomópontba, amely két adatelemet tartalmaz, és amelynek szülője szintén két adatelemet tartalmaz. Példa: szúrjuk be az 1-es elemet:



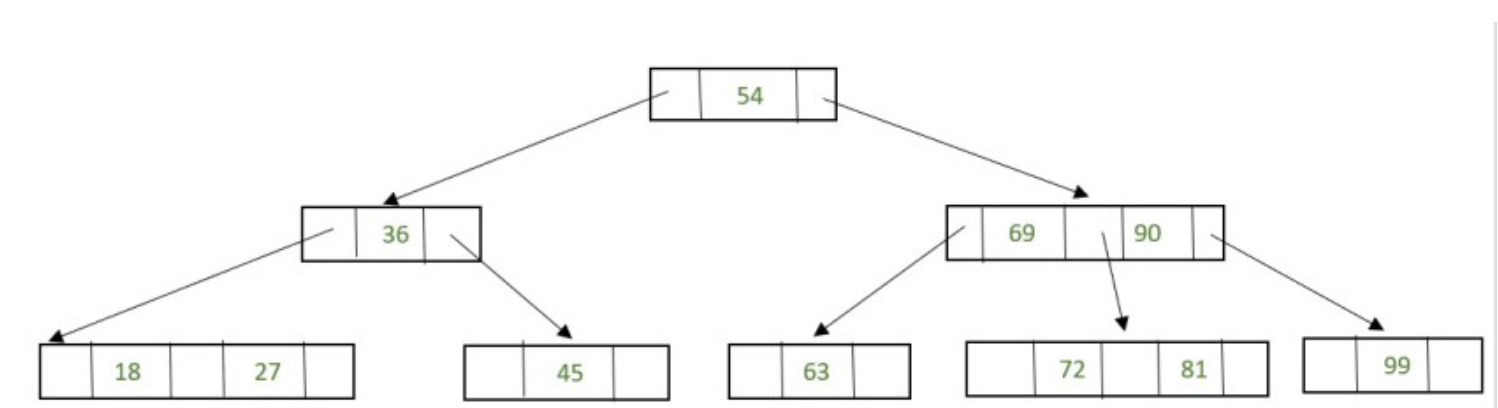
Megkeressük és beszúrjuk a megfelelő helyre. Itt mint a második esetnél, ugyanúgy 3 elemű csomónk van. Szedjük szét:

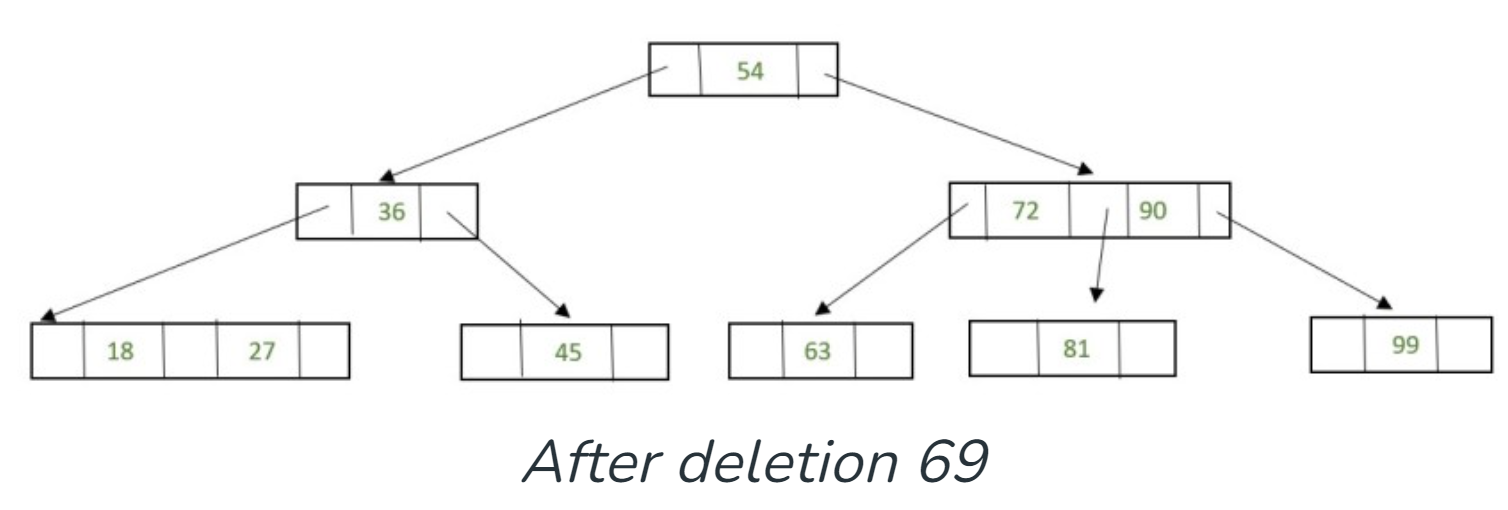
A baloldali ábrán az elemeket egy-egy leaf-re bontjuk. Következő lépésként a szülőt is felbontjuk úgy, hogy a középső elem lesz a szülő, a többi eleme meg a gyermek.

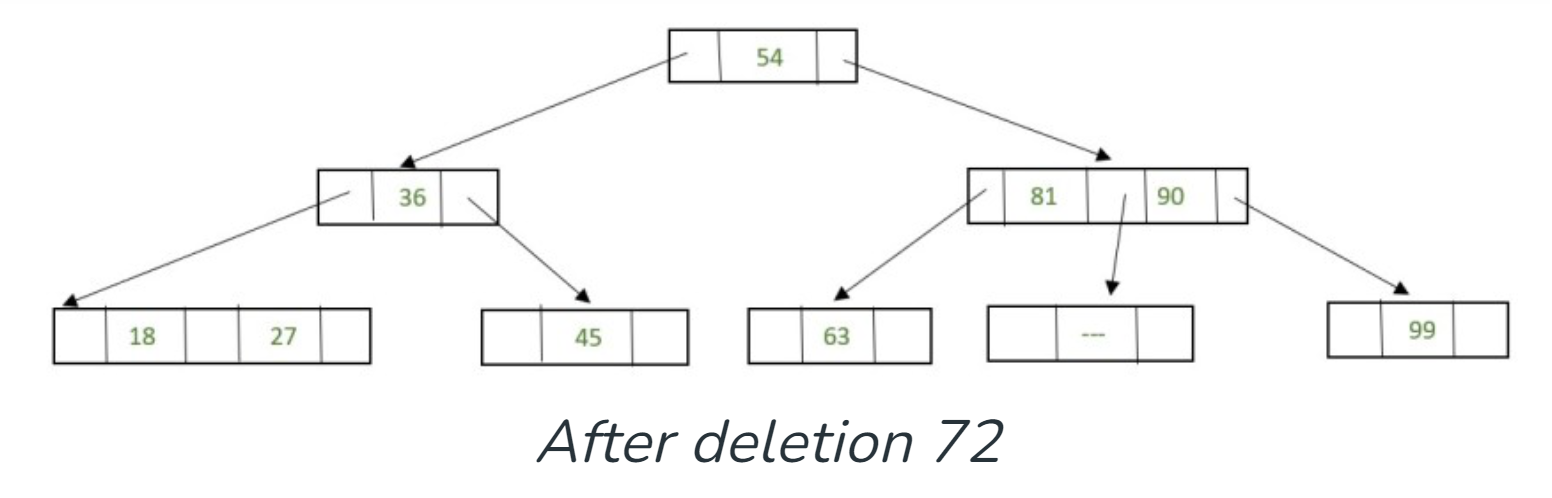
## VI. 4. Melléklet: Faművelet törlés és kiegyensúlyozás

Ha egy csomópontban kevesebb, mint egy adatérték marad, akkor két csomópontot kell összevonni. Ha egy csomópont üressé válik egy érték törlése után, akkor azt egy másik csomóponttal egyesítjük.



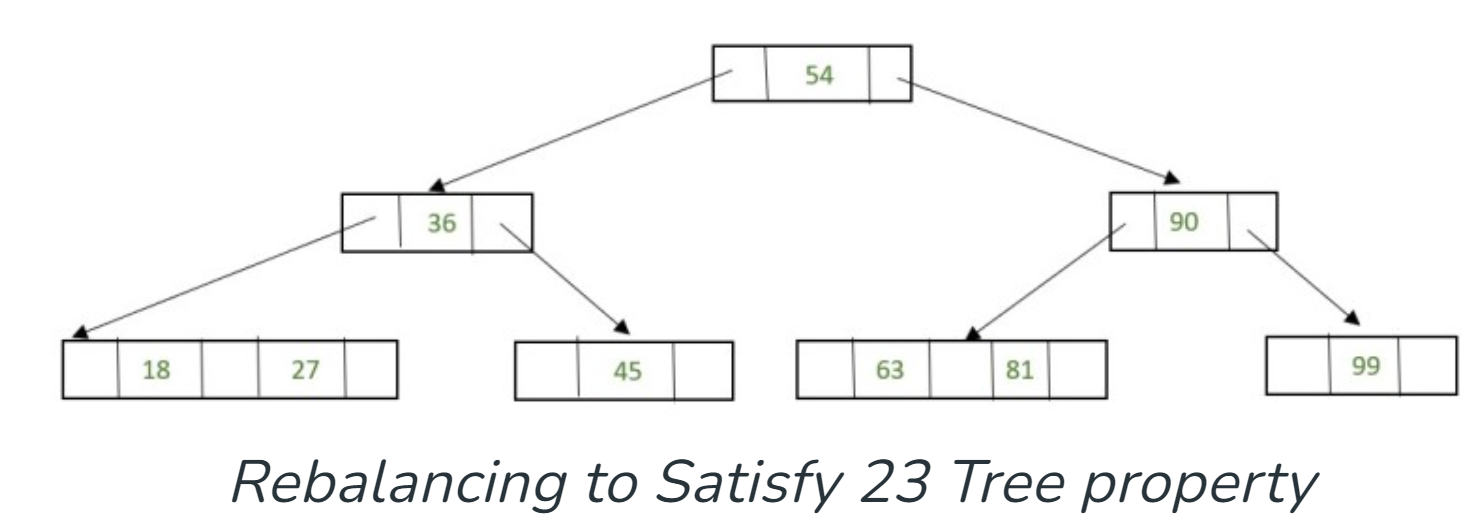
Töröljük a következő értékeket: 69,72, 99

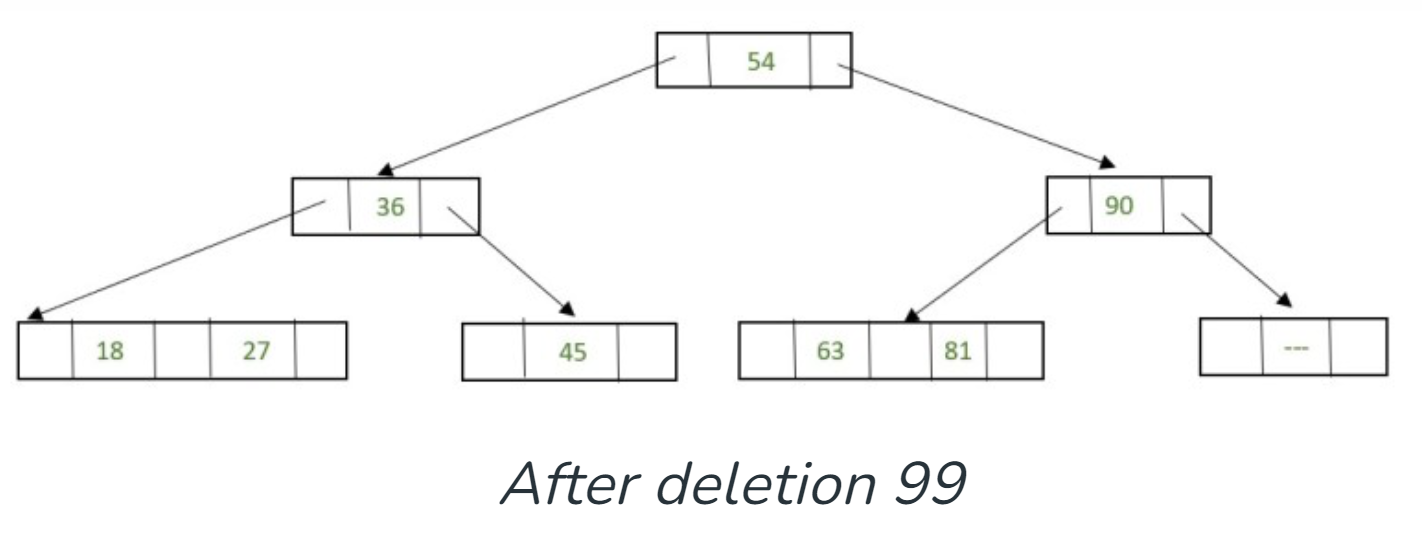
A **69-es érték törléséhez** cseréljük ki az in-order[[4]](#footnote-4) utódjával, azaz a 72-vel. A 69 így a levél csomópontba kerül. Távolítsuk el a 69-es értéket a levél csomópontból.

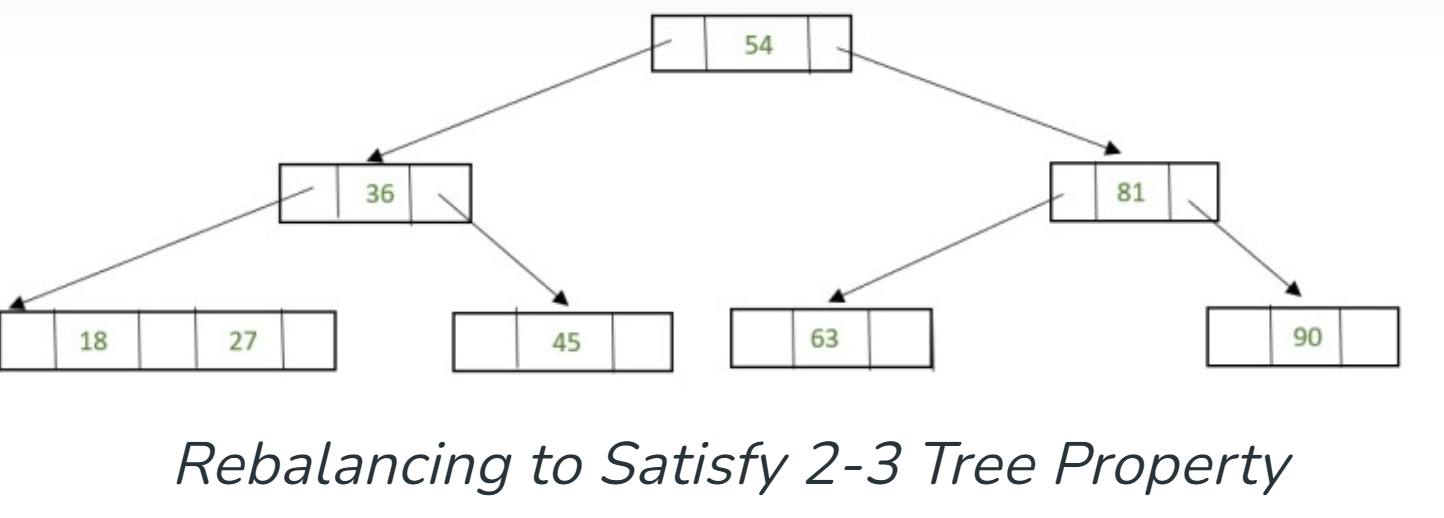
A **72-es érték törléséhez**, mivel a 72 egy belső csomópont, a törléshez cseréljük ki a 72-t az in-order utódjával, az 81-gyel, így a 72 levél csomóponttá válik. Távolítsük el a 72-es értéket a levél csomópontból.

Most van egy levél csomópont, amelyben kevesebb, mint egy adatérték van, ezzel megsértve a 2-3 fa tulajdonságait. Ezért a csomópontot össze kell vonni.

A csomópont összevonásához húzzuk le a szülő csomópont legkisebb adatértékét, és egyesítssük azt a bal oldali szomszédos csomóponttal.

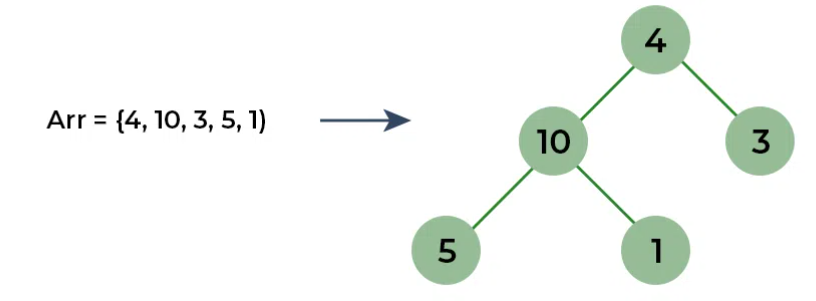


A **99-es érték törléséhez**, mivel a 99 egy levél csomópontban található, az adatérték könnyen eltávolítható.

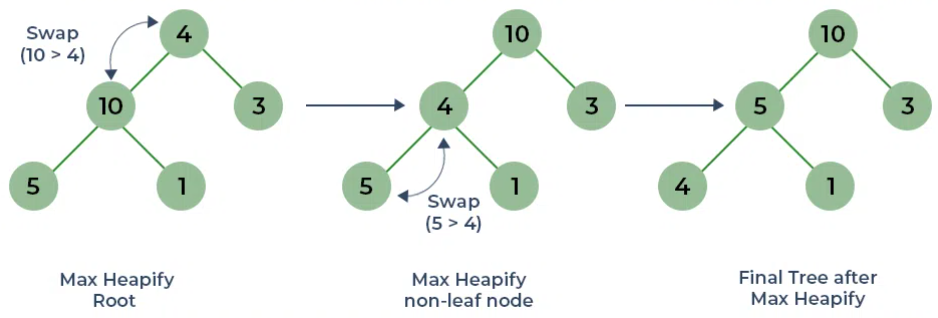
Most van egy levél csomópont, amelyben kevesebb, mint egy adatérték található. Így a csomópontot össze kell vonni. A csomópont összevonásához húzzuk le a szülő csomópont legkisebb adatértékét, és egyesítssük azt a bal oldali szomszédos csomóponttal.

## VI. 5. Melléklet: Kupacrendezés

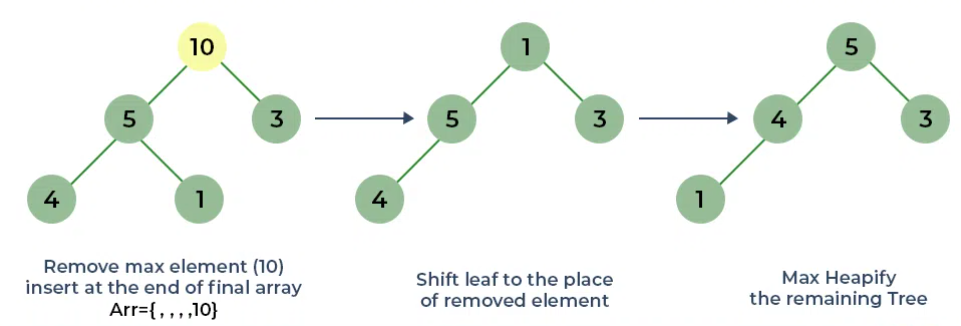
Építsünk a következő tömbből egy bináris kupacot (faműveleteknél látottak alapján): {4, 10, 3, 5, 1}.

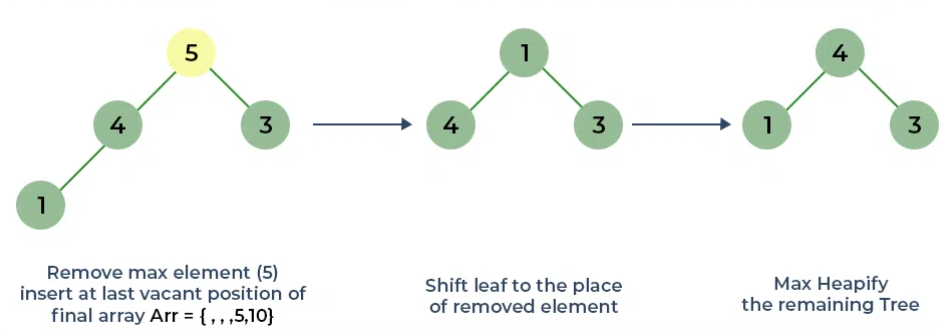


Az eljárás, hogy megkeressük a maximum csúcsot és felcseréljük az értékeket.



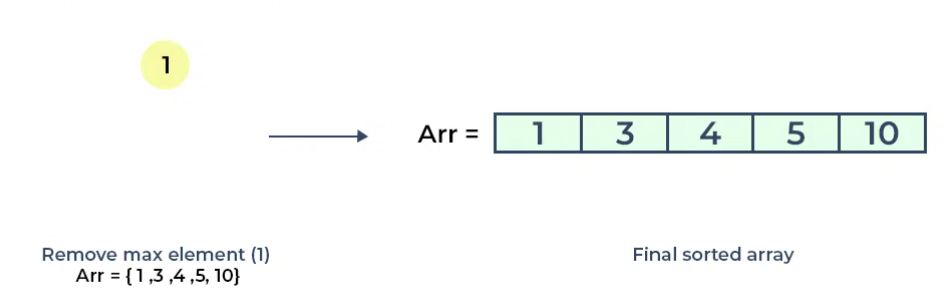
Miután elvégeztük a cseréket, utána a kupacból elhelyezzük az értékeket egy tömbben. A rootban lesz mindig a maximális érték a lenagyobb. Ezt az értékét a tömb utolsó indexére helyezzük. Az érték csomópontját kitöröljük és a legkisebb kerül a helyére (jelen esetben a 10 helyére kerül az 1-es). Ezt követően újra végig csináljuk a maximum csúcs cserét.







Amikor már csak 1 elem van, nem szükséges már a maximum csúcs keresése, elég ha az utolsó elemet eltávolítjuk és hozzáadjuk a tömb index eleméhez.



1. Speciális esete a bináris fának, amely egy olyan fa, ahol minden csomópontnak legfeljebb két gyermekcsomópontja lehet. [↑](#footnote-ref-1)
2. Rekurzíó: a függvény önmagát hívja meg a probléma megoldása érdekében. [↑](#footnote-ref-2)
3. Divide and conquer: atomi elemekre bontjuk a bemenetet, majd összefésüljük őket. [↑](#footnote-ref-3)
4. Az in-order bejárás során a fa elemeit növekvő sorrendben látogatjuk meg, először a bal alsó részfát, aztán a csomópontot magát, végül a jobb alsó részfát. [↑](#footnote-ref-4)