Algoritmusok és adatszerkezetek

Feladatgyűjtemény
(ideiglenes változat)

Eötvös Loránd Tudományegyetem
Informatikai Kar
Algoritmusok és Alkalmazásaik Tanszék

Oktatási segédanyag az Eötvös Loránd Tudományegyetem Informatikai Karának programtervező matematikus és programtervező informatikus szakos hallgatói számára az Algoritmusok és adatszerkezetek I és II című tárgyakhoz.

A feladatgyűjtemény az előadás tematikájának megfelelően lépésenként halad az egyes ismertetetett adatszerkezeteken, illetve algoritmusokon, melynek megadja általános leírását, és működési elvét, valamint egy példán keresztül illusztrálja az algoritmus működését. Az algoritmus (absztrakt) megvalósítását az olvasóra bízzuk a feladatokon keresztül.

Szerkesztették:

Dr. Fekete István (fekete@inf.elte.hu, http://people.inf.elte.hu/fekete)

Giachetta Roberto (groberto@inf.elte.hu, http://people.inf.elte.hu/groberto)

Kovács Péter (kpeter@inf.elte.hu, http://people.inf.elte.hu/kpeter)

Nagy Tibor (ntibi@inf.elte.hu, http://people.inf.elte.hu/ntibi)

Frissítve: 2015. február 24.

Tartalom

18. Rendezés lineáris időben

Ebben a fejezetben olyan rendező algoritmusokkal foglalkozunk, amelyek képesek lineáris nagyságrendi időben rendezni az inputsorozatot, azaz ha az inputsorozat hossza n, az algoritmusok $\Theta(n)$ idő alatt sorba rendezik az elemeit, amennyiben bizonyos szükséges feltételek teljesülnek a kulcstartományra. Ezek az algoritmusok úgy határozzák meg az elemek sorrendjét a sorozatban, hogy nem végeznek összehasonlítást az elemek között, hanem valamilyen alternatív módon helyezik el az elemeket, így ezekre nem vonatkoznak az összehasonlító rendezések esetén kimondott tételek.

Ez azonban egyáltalán nem jelenti azt, hogy az összehasonlító rendezések feleslegesek, ugyanis a nem összehasonlító rendezések sok esetben nem alkalmazhatók, illetve kevésbé hatékonyak. Általában elmondható, hogy ezeket a módszereket olyankor lehet érdemes használni, amikor a lehetséges kulcsok száma a rendezendő elemek számához viszonyítva nem túl nagy.

Edényrendezések

Leírás:

Az *edényrendező algoritmusok* elve, hogy a bemenő sorozat elemeit halmazokba sorolják, amelyeket *edények*nek nevezünk, és e halmazok alapján állapítják meg, hogy az elemek melyik helyre kerülnek a rendezés során. Az elemek általában kulcs-érték párok, ahol a rendezésben természetesen a kulcs rész játszik szerepet. Az edények nem a kulcs teljes értékére, csak annak egy részére, komponensére – szám esetén például egy számjegyre – vonatkoznak. Két megközelítést tárgyalunk: előre-, illetve visszafelé haladó rendezést, annak függvényében, hogy a kulcs mezőit milyen sorrendben dolgozzuk fel.

Edényrendezés előre:

Elve, hogy a bemenő sorozat elemeit a kulcs első mezője szerint szétválogatjuk edényekbe, majd az egyes edényekben lévő elemeket a kulcs második mezője szerint további edényekbe tesszük, és így rekurzívan tovább folytatjuk az eljárást, amíg a kulcs végére nem érünk.

A szintek számát általában a kulcs hossza (a komponensek száma) határozza meg, míg az egyes szinteken lévő edények számát az, hogy a megfelelő kulcskomponensnek – például egy szöveg egy karakterének vagy egy szám egy számjegyének – hány lehetséges értéke lehet**.** Például, ha 16-os számrendszerbeli, 0 és FFF közötti számokat szeretnénk rendezni, akkor összesen 3 szintre van szükségünk – mert a legnagyobb szám 3 számjegyből áll –, és szintenként $16, 16 \cdot 16$, illetve $16 \cdot 16 \cdot 16$ edényt kell nyilvántartanunk. Előfordulhat, hogy az edények száma komponensenként változik. Például dátumokat rendezünk, amelyek év, hónap és nap komponensekből állnak. Ekkor először év szerint rendezünk, ami igénybe vehet 2009 edényt. Ezt követik a hónapok, ami 12 edény évenként, végül a hónapokat (legfeljebb) 31 napra bontjuk.

Legyen a kulcsok hossza d, és az elemek száma n, ekkor a műveletigény arányos ezek szorzatával, tehá $\Theta(d \cdot n)$t. Ha a kulcsok hossza konstans, akkor a műveletigény $\Theta(n)$-es. Tehát látható, hogy ez egy gyors algoritmus, csak nagyon memóriaigényes. Például az említett dátumos esetben $2009 + 2009 \cdot 12 + 2009 \cdot 12 \cdot 31$ db edényt kell nyilvántartanunk a memóriában még

akkor is, ha csak 10 dátumot szeretnénk rendezni. Ez elfogadhatatlan, ezért ez az algoritmus a gyakorlatban általában használhatatlan (kivéve bináris számok rendezését, l. később).

Edényrendezés vissza:

Javítsunk a korábbi problémán. Induljunk ki az első helyett az utolsó kulcskomponenstől, és haladjunk visszafelé (például dátumok esetén először válogassunk napok szerint, majd hónapok szerint, végül évek szerint), továbbá egy rendezési fázis után az adatokat vegyük ki az edényekből, és helyezzük őket egy listába, megőrizve azt a sorrendet, amelyet ez az iteráció eredményezett. A következő szétválogatás előtt töröljük az edényeket – vagy használjuk őket a következő fázishoz –, így azok helye a memóriában felszabadul, és a következő iteráció edényeit oda helyezhetjük a memóriában.

Könnyen belátható, hogy ez a módszer is rendezett sorozatot eredményez (a k. iteráció után az elemek az utolsó k mező szerint rendezve lesznek), ellenben mindössze annyi edényre van szükségünk, amennyi a kulcsok legnagyobb komponensében a lehetséges értékek száma, tehát az évszámok esetében elegendő 2009 edény, míg tízes számrendszerbeli számoknál mindössze 10 edény szükséges függetlenül attól, hogy hány jegyűek a számok. Ha a műveletigényt nézzük, pusztán a fázisonkénti listaépítés adódik a korábbihoz, tehát maradt a $\Theta(d \cdot n)$-es lépésszám, de sokkal kevesebb memóriára volt szükség.

Példa:

Rendezzük a következő sorozatot előre haladó, illetve visszafelé haladó edényrendezéssel: **`kjf`**, **`uht`**, **`abs`**, **`oim`**, **`zhm`**, **`ahf`**, **`uhs`**. Tegyük fel, hogy a lehetséges karakterek az angol ábécé betűi, azaz összesen 26 edényre lesz szükségünk minden karakterhez.

Előre haladó rendezéssel:

1. Felveszünk 26 edényt a 26 karakternek. A szétválogatás után az összes üres lesz a következők kivételével: A: **`abs`**, **`ahf`**, K: **`kjf`**, O: **`oim`**, U: **`uht`**, **`uhs`**, Z: **`zhm`**.

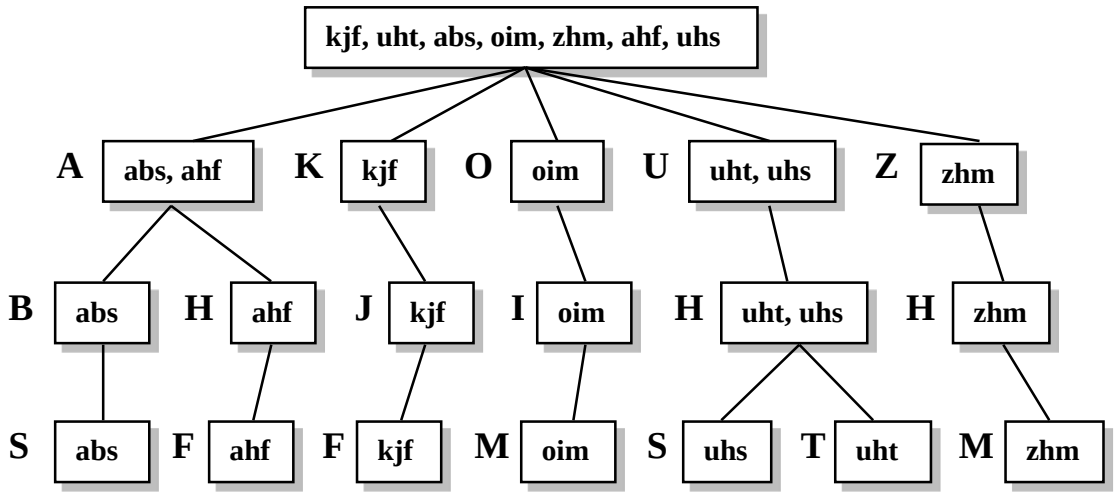
   Látható, hogy két edény maradt, amelyekben több elem is előfordul.

2. Felveszünk minden edényhez további 26 edényt, azaz összesen 676 edényt. Az összes üres lesz a következők kivételével: AB: **`abs`**, AH: **`ahf`**, KJ: **`kjf`**, OI: **`oim`**, UH: **`uht`**, **`uhs`**, ZH: **`zhm`**.

   Még mindig maradt egy edény, amelyben több elem van, ezért folytatjuk az algoritmust.

3. Felveszünk minden edényhez 26 edényt, azaz összesen 17576 edényt. Az összes üres lesz a következők kivételével: ABS: **`abs`**, AHF: **`ahf`**, KJF: **`kjf`**, OIM: **`oim`**, UHS: **`uhs`**, UHT: **`uht`**, ZHM: **`zhm`**.

4. Az utolsó lépésben az eredmények tartalmát összefűzzük, így egy rendezett sorozatot kapunk: **`abs`**, **`ahf`**, **`kjf`**, **`oim`**, **`uhs`**, **`uht`**, **`zhm`**. Ehhez összesen $4 \cdot 7 = 28$ lépésre volt szükségünk, illetve 18278 edényre a memóriában.



Visszafelé haladó rendezéssel:

1. Felveszünk 26 edényt a 26 karakternek. A szétválogatás után az összes üres lesz a következők kivételével: F: **`kjf`**, **`ahf`**, M: **`oim`**, **`zhm`**, S: **`abs`**, **`uhs`**, T: **`uht`**. Ezután az edények tartalmát összefűzzük: **`kjf`**, **`ahf`**, **`oim`**, **`zhm`**, **`abs`**, **`uhs`**, **`uht`**, és az edényeket kiürítjük.
2. A sorozatot a 2. karakter szerint újra szétválogatjuk a 26 edénybe: B: **`abs`**, H: **`ahf`**, **`zhm`**, **`uhs`**, **`uht`**, I: **`oim`**, J: **`kjf`**, és a kapott sorozatot összeolvassuk: **`abs`**, **`ahf`**, **`zhm`**, **`uhs`**, **`uht`**, **`oim`**, **`kjf`**.
3. A sorozatot az 1. karakter szerint újra berakjuk a 26 edénybe: A: **`abs`**, **`ahf`**, K: **`kjf`**, O: **`oim`**, U: **`uhs`**, **`uht`**, Z: **`zhm`**, és a kapott sorozatot összeolvassuk: **`abs`**, **`ahf`**, **`kjf`**, **`oim`**, **`uhs`**, **`uht`**, **`zhm`**. Így egy rendezett sorozatot kaptunk, amihez $2 \cdot 3 \cdot 7 = 42$ műveletet végeztünk, és mindössze 26 edényre volt szükségünk.

18.1. Feladat:

Rendezzük az alábbi sorozatokat előre-, illetve visszafelé haladó edényrendezéssel:

a) 460, 314, 831, 12, 7, 953 (decimális számok);

b) 012, 111, 120, 122, 110, 001, 022 (hármas alapú számrendszerbeli számok);

c) 12, 367, 111, 100, 73, 5 (oktális számok);

d) 43f3, 98aa, 1841, e49a, 12bc, e24c (hexadecimális számok);

e) bsu, nvo, nja, uhs, mpc, nno, upu (angol ábécé betűi);

f) 2003.01.25, 1999.09.30, 2001.09.14, 2001.01.30, 2002.12.30, 2003.01.14.

18.2. Feladat:

Adott egy *n* alapú számrendszerben leírt számok sorozata, amelyekben a számjegyek száma *k*. Összesen *s* adatunk van. Adjuk meg, hogy előre-, illetve visszafelé haladó edényrendezés esetében mekkora műveletigény segítségével rendezhetjük a számokat, és mennyi edényre lesz szükségünk. Van-e olyan *k*, *n*, *s* kombináció, amely mellett az előre haladó edényrendezés gyorsabb a visszafelé haladónál?

18.2.1. Feladat:

Tegyük fel, hogy adottak az alábbi, 1901 és 2000 közötti dátumok: 1963.02.11, 1988,08.30, 1960.11.03, 2000.11.18, 1988.04.15, 1911.07.28. Végezzünk egy lineáris keresést az évszámokon, és csak a talált évszámoknak hozzunk létre edényeket, így rendezzük a sorozatot. Vizsgáljuk meg, mennyi műveletigény-növekedést, illetve memóriacsökkenést von maga után a változtatás ahhoz képest, ha mind a 100 évnek megfelelő edényt létrehozzuk.

RADIX rendezések

Leírás:

Mivel a számítástechnikában mindent binárisan ábrázolunk, a rendezéseket is gyakran közvetlenül a bináris számokon érdemes lefolytatnunk, ami gépi kód szintjén némileg nagyobb hatékonyságot jelent, mintha nem az adatok bináris ábrázolásán dolgoznánk. Az edényrendezések bináris számokon dolgozó változatait *RADIX* rendezéseknek nevezzük. Itt is megkülönböztetjük az *előre haladó* (MSD, most significant digit), illetve *visszafelé haladó* rendezést (LSD, least significant digit).

A RADIX MSD rendezés értelemszerűen minden fázisban két részre bontja a sorozatot, annak függvényében, hogy az adott helyi értéken 0 vagy 1 szerepel. Ennek módja, hogy két mutatóval haladunk a sorozatban, egyikkel az elejéről, másikkal a végéről külön-külön. Amennyiben az elejéről haladva 1-est, és a végéről haladva 0-t találunk, megcseréljük a két számot, majd lépkedünk tovább, amíg a két mutató ugyanarra az értékre nem lép vagy át nem lép egymáson. Ahol ez megtörténik, két részre bontjuk a tömböt – ezek felelnek meg az egyes edényeknek –, majd az eljárást rekurzívan folytatjuk a kialakult résztömbökre a következő számjegytől. Így végül egy rendezett sorozatot kapunk, és a rendezést az eredeti helyen végeztük – csupán egy további helyre volt szükségünk a cseréhez. A módszer alkalmazhatósága tehát azon múlik, hogy bináris számok esetén minden pozíción csak kétféle érték szerepelhet, így az egymásba ágyazott edények megvalósíthatók a tömb rekurzív felosztásával, nem kell külön-külön tárolnunk őket.

A RADIX LSD rendezésnél abból indulunk ki, hogy minden fázisban két edényre van szükségünk, és ezek együttes mérete mindig megegyezik az eredeti inputsorozat méretével. Tehát, ha az eredeti sorozatunk egy tömb, vegyünk fel egy ugyanakkora méretű segédtömböt, és az első fázisban pakoljuk át ide az elemeket, majd a második fázisban vissza az eredeti tömbbe, és így tovább, tehát a páratlan fázisokban a segéd-, a páros fázisokban pedig az eredeti tömböt töltjük fel.

A pakolás történjen a következőképpen:

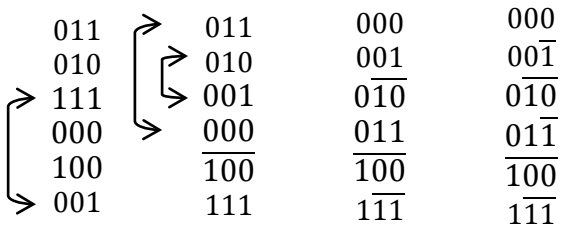
1. Olvassuk be az adatokat egymás után a kezdő tömbből, és amelyiknek az utolsó jegye 0, azt rakjuk a segédtömb elejére egymás után, amelyiknek az utolsó jegye 1, azt pedig rakjuk a tömb végére egymás elé.
2. Ezután a segédtömbben a beolvasás szerinti sorrendben foglalnak helyet a 0-ra végződő számok, majd ezt követik a beolvasás szerinti fordított sorrendben az 1-re végződőek. A tömb két részének határindexét tároljuk el.

3. Ismételjük meg az előbbi lépéseket az utolsótól visszafelé az összes többi számjegyre, azzal a módszerrel, hogy a tömbből elölről olvasunk be, amíg el nem értük az előbb eltárolt indexet, majd ezután a végétől visszafelé olvasunk be a határig.
4. Miután az első számjegyre is lefuttattuk a rendezést, az eredményt helyezzük el a másik tömbben úgy, hogy az utoljára eltárolt indexig előrefelé olvasunk be a tömbből, majd ezt követően a végétől visszafelé olvasunk be a határig.

Végül egy rendezett sorozatot kapunk. Ha a számok hossza d volt, számosságuk n, akkor $\Theta(d \cdot n)$ műveletigénnyel végeztük el a rendezést, MSD esetben $n + 1$, LSD esetben pedig $2n$ memóriaigénnyel. Tehát ezen algoritmusok esetében már nem hatalmasodik el a memóriaigény, mint az általános edényrendezésnél, cserébe d értéke megnőhet, de csak konstansszorosára.

Példa:

Rendezzük Radix MSD rendezéssel a következő értékeket: **011**, **010**, **111**, **000**, **100**, **001**. Nézzük menetenként az eljárás működését a tömb reprezentációjával, minden esetben jelölve a helycseréket.



Látható, hogy az eljárás végén megkaptuk a rendezett sorozatot. Most rendezzük Radix LSD rendezéssel a fenti értékeket. Az eredmény nyilván ugyanaz lesz, csupán az eljárás fog változni. Tekintsük szövegesen a műveleteket.

1. Utolsó számjegy szerint a segédtömb elejébe tesszük a **010**, **000**, **100** értékeket, a tömb végébe a **011**, **111**, **001** értékeket, tehát a tömb: **010**, **000**, **100**, **001**, **111**, **011**, és megjegyezzük, hogy a 3. szám után van a határ.
2. A segédtömböt rendezzük a második számjegy szerint, az eredeti tömb elejébe tesszük a **000**, **100**, **001** értékeket, a végébe a **010**, **011**, **111** értékeket, tehát a tömbünk: **000**, **100**, **001**, **111**, **011**, **010**, és ismét a 3. szám után van a határ.
3. A tömböt rendezzük a segédtömbbe az első számjegy szerint, az elejébe **000**, **001**, **010**, **011**, a végébe **100**, **111** kerül, azaz a tömb: **000**, **001**, **010**, **011**, **111**, **100**, és a 4. szám után van a határ.
4. Minden számjegyen végighaladtunk, olvassuk össze a segédtömböt úgy, hogy most már rendezve legyenek a számok az eredeti tömbben, amely így **000**, **001**, **010**, **011**, **100**, **111** lesz.

18.3. Feladat:

Rendezzük Radix MSD, illetve LSD rendezéssel az alábbi bitsorozatokat:

a) 1001, 1110, 0110, 0000, 1100, 0101;

b) 1100, 0001, 1111, 0111, 1010, 0010;

c) 00010, 11000, 11110, 00101, 01110, 00011;

d) 11001, 10010, 11100, 00101, 00001, 11111.

18.4. Feladat:

Rendezzük az alábbi számokat visszafelé haladó edényrendezéssel, majd a bináris megfelelőiket rendezzük Radix LSD rendezéssel. Vizsgáljuk meg, hogy viszonyul az egyik eset memória, illetve műveletigény szükséglete a másik esethez: 101 (01100101), 230 (11100110), 45 (00101101), 3 (00000011), 89 (0101011001), 192 (110000000).

Leszámláló rendezés

Leírás:

Tekintsük azt a feladatot, ahol az inputsorozat elemeit kulcsok segítségével azonosítjuk, amelyek egy adott k számnál nem nagyobb természetes számok – a későbbieknek bemenő adatoknak csak ezeket a kulcsokat tekintjük –, amelyek között ismétlés is előfordulhat. Rendezzük ezeket a számokat egy olyan eljárással, amely az elemek előfordulási száma alapján határozza meg helyüket.

A leszámláló rendezésben tegyük fel, hogy az adatokat egy A tömb tartalmazza, valamint vegyünk fel egy ezzel azonos méretű B tömböt, ide kerül a rendezett sorozat. Végül szükségünk lesz egy C segédtömbre, amelynek mérete $k + 1$, és 0-tól van indexelve. Az algoritmus három egymás utáni ciklussal végzi el a rendezést:

1. A C tömbbe írjuk bele, hogy az egyes kulcsokból (elemekből) mennyi található az A tömbben. Ehhez először C minden elemét állítsuk be nullára, majd az A tömböt egyszer végigolvasva minden $A[i]$ elemre növeljük meg eggyel $C[A[i]]$ értékét.
2. Számoljuk össze a C tömbben, hogy egy adott értéknél nem nagyobb kulcsokból mennyi található az A tömbben. Ehhez nincs más dolgunk, mint a C tömböt végigolvasva minden eleméhez hozzáadni az előtte lévőt.
3. Végül vegyük egymás után az elemeket az A tömbből, és tekintsük őket indexnek, majd keressük meg az ennek az indexnek megfelelő értéket a C tömbben. Miután ezt is kiolvastuk, tekintsük indexnek, és a B tömbben erre a helyre tegyük be az A-beli elemet. Röviden képlettel: $B[C[A[i]] = A[i]$. Ezt követően a megadott helyen csökkentsük eggyel a C tömbben tárolt értéket.

Az utolsó ciklus végeztével a B tömbben rendezve kapjuk meg az elemeket, amihez $\Theta(k) + \Theta(n) + \Theta(k) + \Theta(n)$ műveletet végeztünk, azaz az algoritmus $\Theta(n + k)$ futásidővel dolgozik.

Könnyen belátható, hogy az algoritmus általánosítható a természetes számokon kívül bármilyen olyan halmazra, amelynek elemei diszkrétek és felállítható egy rendezési reláció rajtuk – tehát a kulcshalmaz kölcsönösen egyértelműen megfeleltethető egy $[0..k]$ halmaznak –, például egész számokra vagy karakterekre. Ehhez mindössze C definícióján kell módosítanunk vagy egy címfüggvényt bevezetnünk, amely a C megfelelő helyére képezi le az A-beli elemeket.

Példa:

Végezzük el az alábbi bemeneten a leszámláló rendezést: **2**, **3**, **0**, **2**, **3**, **1.**

1. Töltsük fel a C tömböt aszerint, hogy mely elemből mennyi található az inputsorozatban. Ha tömböt 0-tól indexeljük, a következőt kapjuk:

C:	1	1	2	2

2. Eztán adjuk hozzá a C tömb értékeihez az őket megelőző értékeket:

C:	1	2	4	6

3. Végül haladjunk végig az A tömb elemein, és rakjuk őket a megfelelő sorrendben a B tömbbe, amelyet a C tömbön keresztül kapunk meg. Közben csökkentsük a C tömb megfelelő értékeit. Első lépésbe $i = 1$ n -re vesszük az $A[1]$ -t, ami 2, a 2-es indexen a C tömbben a 4 szerepel, tehát a B tömb 4. mezőjébe helyezzük a számot, és csökkentjük $C[2]$-t eggyel, és így tovább. A ciklus lefutását az alábbi táblázat szemlélteti:

i	$A[i]$	***C***				***B***					
		0	***1***	***2***	***3***	***1***	***2***	***3***	***4***	***5***	***6***
1	2	1	2	**4**	6				**2**		
2	3	1	2	3	**6**				2		**3**
3	0	**1**	2	3	5	**0**			2		3
4	2	0	2	**3**	5	0		**2**	2		3
5	3	0	2	2	**4**	0		2	2	**3**	3
6	1	0	**1**	2	4	0	**1**	2	2	3	3

4. Látható, hogy az eredményül kapott B tömbben az elemek rendezve találhatók, ehhez összesen $4 + 6 + 4 + 2 \cdot 6 = 26$ lépést kellett megtennünk.

18.5. Feladat:

Végezzük el a leszámláló rendezés algoritmusát az alábbi sorozatokon:

a) 1, 0, 2, 1, 3, 2, 0;

b) 0, 8, 4, 4, 2, 1, 6, 0, 1, 8;

c) -2, 0, 1, 1, -1, 2, 0, -2;

d) a, f, b, e, e, a, b, a, d.

18.6. Feladat:

Adjuk meg a leszámláló rendezés algoritmusát struktogram segítségével arra az esetre, ha

a) a kulcsok 0 és k közötti egész számok;

b) a kulcsok m és $m + k$ közötti egész számok, és a C tömb továbbra is 0 és k között indexelt;

c) a kulcsok $–k$ és $+k$ közötti páros egész számok, ahol k páros szám, és a C tömb továbbra is 0 és k között indexelt.

18.7. Feladat:

Adott a következő inputsorozat: 6, 3, 1, 7, 6, 1, 0, 8. Végezzük el a sorozaton a leszámláló, a kupac-, illetve a versenyrendezés algoritmusát. Számoljuk ki az egyes algoritmusok műveletigényét, memóriaszükségletét, és hasonlítsuk össze a kapott eredményeket.

18.8. Feladat:

Adjunk meg egy olyan algoritmust, amely 0 és k közötti n db egész számot előzetesen feldolgoz, majd $\Theta(1)$ időben eldönti, hogy az n egész számból hány esik az $[a..b]$ intervallumba. Az algoritmus $\Theta(n+k)$ előfeldolgozási időt használhat.

18.9. Feladat:

Rendezni szeretnénk egy 1024 rekordból álló adathalmazt. A kulcsnak két mezője van, testmagasság (32 különböző érték) és életkor (16 különböző érték). A rendezés: testmagasság szerint, azon belül évszám szerint. Hány lépésben, illetve mekkora memóriaigénnyel oldható meg a feladat, ha

a) kupacrendezést használunk, amely egyben kezeli az összetett kulcsot;

b) előbb egyenletesen 32 edénybe rakjuk a kulcs 1. mezőjének hasításával a rekordokat (tegyük fel, hogy mindbe 32 rekord jut), és utána a kulcs 2. mezője alapján minden edényt kupacrendezéssel rendezünk (vegyes módszer);

c) előbb egyenletesen 32 edénybe rakjuk a kulcs 1. mezőjének hasításával a rekordokat (tegyük fel, hogy mindbe 32 rekord jut), és utána a kulcs 2. mezője alapján minden edényt leszámláló rendezéssel rendezzünk.

19. Hasítótáblák

Az adathalmazokban az egyes elemeket, adatrekordokat azonosítani szoktuk úgynevezett kulcsok segítségével, amelyek általában egész számok. Ezek a számok egy nagyobb intervallumban mozoghatnak, de számosságuk ennél általában jóval kisebb. Például van 1000 rekordunk, amelyek kulcsai az 1 és 1 millió közötti tartományban mozognak, és lehetnek ismétlődő kulcsok is. Ezeket a rekordokat szeretnénk leképezni egy kisebb tartományra. Ennek a megvalósítását *hasítás*nak (hash-elésnek) nevezzük.

Leírás:

A hasítás során az inputsorozat kulcsaira alkalmazunk egy leképezést, amelyet *hasítófüggvény*nek szokás nevezni. Ez a függvény mondja meg, hogy az adatokat hova helyezzük el az úgynevezett *hasítótábl*ában. Persze mivel a kulcsok lehetséges értékei jóval meghaladják a hasítótábla méretét, kulcsütközések léphetnek fel, amelyeket valamilyen módon kezelnünk kell. Ennek megfelelően az eljárásnak két változatát különböztetjük meg.

Hasítás láncolással:

Hasítótáblának egy tömböt veszünk, amelynek elemei láncolt listák fejmutatói. Beszúráskor az új adat kulcsára alkalmazzuk a hash-függvényt, amely megadja a tömb egy indexét, és ekkor az így megadott listába fűzzük az adatot. Címütközéskor egy listába több adat is kerülhet, amelyek sorrendjére nem teszünk megszorítást.

A beszúrás műveletigénye – a hasítófüggvény kiszámítása, illetve a listába való befűzés – konstans, hiszen a hasítófüggvény által szolgáltatott helyre csak betesszük az elemet. Keresésnél, illetve törlésnél – mivel az egyes listákban meg kell keresnünk lineárisan az elemet –, a műveletigény $O(k)$, ahol k az adott lista hossza. Ez a legrosszabb esetben megegyezhet az inputsorozat hosszával – ekkor a hasítófüggvényünk minden kulcsot ugyanarra az indexre képezett.

Hasítás nyílt címzéssel:

Hasítótáblának egy tömböt veszünk fel, és ebben szeretnénk tárolni az adatokat, tehát – szemben a láncolással – a tárolható elemek maximális száma rögzített. Az adatokat alapértelmezés szerint a hasítófüggvénynek megfelelő helyre tesszük, kulcsütközés esetén azonban valahogyan keresnünk kell egy még üres mezőt. Ezt többféle módszerrel is megtehetjük, amelyek lényege, hogy egy segédfüggvénnyel egészítjük ki a hash-függvényt, mely az elhelyezési próbálkozások számának (legyen ez i), a tömbméretnek (legyen ez M), illetve az aktuális kulcsnak (legyen ez k) a függvénye. A 3 elterjedt segédfüggvény:

a) lineáris próbálkozás: $H_i(k) = (h(k) - i)\ mod\ M$;

b) négyzetes próbálkozás: $H_i(k) = \left(h(k) - (-1)^i \cdot \left\lfloor \frac{i+1}{2} \right\rfloor^2\right)\ mod\ M$;

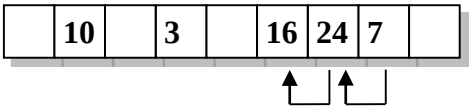
c) második hash-függvény: $H_i(k) = (h(k) - i \cdot h'(k))\ mod\ M$.

Példa:

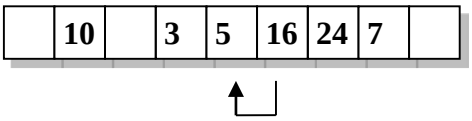
Nyílt címzéssel, lineáris próbálással és a $h(k) = k\ mod\ 9$ hasítófüggvénnyel hasítsuk a következő sorozatot: **7**, **10**, **3**, **24**, **16**, **5**, majd ezt követően keressük meg, és töröljük ki a 16-os

elemet. Alkalmazzuk a hash-függvényt egymást követően a bemenő adatokra, és rakjuk őket a megfelelő indexre:

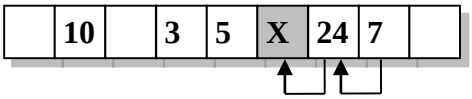
1. $7\ mod\ 9 = 7$, betesszük a 7-es indexű helyre.
2. $10\ mod\ 9 = 1$, betesszük az 1-es indexű helyre.
3. $3\ mod\ 9 = 3$, betesszük a 3-as indexű helyre.
4. $24\ mod\ 9 = 6$, betesszük a 6-os indexű helyre.
5. $16\ mod\ 9 = 7$, a 7-es indexű hely foglalt, lineáris próbálással csökkentsük az indexet. Az első próba a 6-os pozíció, amelyik szintén foglalt, a második próba az 5-ös pozíció, amelyik szabad, ezért ide tesszük be a 16-ot.



6. $5\ mod\ 9 = 5$, az 5-ös indexű hely foglalt, ezért lineáris próbálással a 4-es helyre kerül.



7. Most töröljük ki a 16-os elemet. Először alkalmazva a hasítófüggvényt, a 16-ost a 7-es helyen keressük, ám az nem található ott, ezért lineáris próbálással arrébb lépünk a 6-os, majd az 5-ös indexre, ahol megtaláljuk és kitöröljük a 16-ot.



19.1. Feladat:

Fogalmazzuk meg az alábbi algoritmusokat nyílt címzés és osztómódszer esetén stuktogrammal. A kulcsütközés feloldására használjuk az $Offset(s)$ függvényt, ahol s a próbálkozás száma – tehát ezt nem kell megvalósítani.

a) Elem beszúrása ($Insert$);

b) Elem keresése ($Find$);

c) Elem törlése ($Delete$);

d) Hash-tábla méretének megváltoztatása ($Resize$).

19.2. Feladat:

Tegyük fel, hogy egy *h* véletlen hasító függvényt használunk, amely *n* különböző kulcsot képez le egy *m* hosszúságú táblázatba. Mi a kulcsütközések várható száma? Pontosabban, mi a $\{(x, y)|h(x) = h(y)\}$ halmaz elemeinek a várható számossága?

19.3. Feladat:

Miért nem jók a következő hasítófüggvények?

a) $h(k) = 5 \cdot k\ mod\ 10$;

b) $h(k) = 6 \cdot k\ mod\ 32$.

19.4. Feladat:

Nyílt címzéssel, $h(k) = k \; mod$ 11 hasítófüggvénnyel hasítsuk a következő sorozatot: 18, 28, 36, 17, 62, 48, 50. Rajzoljuk le a keletkezett táblát – amely kezdetben üres volt –, ha a kulcsütközések esetén

a) lineáris próbálást alkalmazunk;

b) négyzetes próbálást alkalmazunk;

c) kettős hasítást alkalmazunk, $h'(k) = k \; mod \; 7 + 1$ függvénnyel.

19.5. Feladat:

Nyílt címzéssel és szorzó módszerrel kapott hasítófüggvénnyel hasítsuk a következő sorozatot: 15, 7, 12, 18, 23, 5. A kulcsütközést lineáris próbálással oldottuk fel. Rajzoljuk le a kapott táblázatot, ha

a) A=1/2 és M=8;

b) A=1/3 és M=16.

19.6. Feladat:

Mutassuk meg miért szükséges a kettős hash-elésnél, hogy $h'(k)$ és M relatív prímek legyenek.

19.7. Feladat:

Legyenek $h(k) = k \; mod \; M$, $M = 2^p - 1$, továbbá a kulcsok 2^p alapú számrendszerben felírt számok. Legyen *x* kulcs egy kétjegyű szám, és *y* az *x* számjegyeinek a felcserélésével kapott kulcs. Mutassuk meg, hogy $h(x) = h(y)$.

19.8. Feladat:

Adott egy nyílt címzésű hasítótábla, amelyben még nem töröltünk elemet. Tegyük fel, hogy a táblában az elemek száma k, a tábla mérete M. Legfeljebb hány elem bekerülése esetén lehetett kulcsütközés, ha:

a) minden második cellája üres, és kulcsütközés esetén lineáris próbálást alkalmaztunk;

b) minden második cellája üres, és kulcsütközés esetén négyzetes próbálást alkalmaztunk;

c) minden harmadik cellája üres $3|M$, és kulcsütközés esetén lineáris próbálást alkalmaztunk;

d) minden negyedik cellája üres, $4|M$, és kulcsütközés esetén négyzetes próbálást alkalmaztunk.

19.9. Feladat:

Adott egy nyílt címzésű hasítótábla, amelynél a kulcsütközések feloldására lineáris próbálást alkalmaztak. A hasítótábla a $H[0..M-1]$ vektorban van ábrázolva. Adjunk eljárást, amely megállapítja, hogy legfeljebb hány kulcsütközés lehetett. Feltételezhetjük, hogy nem volt törlés a táblából.

19.10. Feladat:

A hasítótábla típusának műveletei:

- $Empty$: üres tábla létrehozása;
- $IsEmpty(ht)$: megadja, hogy üres-e a tábla;
- $Insert(ht, q)$: beszúr egy új kulcsot a táblába;
- $Delete(ht, q)$: kitöröl egy adott kulcsot, ha az megtalálható a táblában;
- $Find(ht, q)$: visszaadja egy keresett kulcs helyét a táblában, illetve -1-t, ha nincs benne;
- $IsFull(ht)$: megadja, hogy a tábla tele van-e, természetesen csak nyílt címzésű megvalósítás esetén.

Valósítsuk meg az alábbi a műveleteket stuktogram segítségével. Segítség: ha nyomon követjük, hány elem található a táblában, akkor egyszerűen megkapjuk a választ bizonyos kérdésekre.

a) osztómódszer és láncolt ábrázolás esetén;

b) osztómódszer és nyílt címzésű ábrázolás esetén lineáris próbálással;

c) osztómódszer és nyílt címzésű ábrázolás esetén négyzetes próbálással;

d) szorzómódszer és nyílt címzésű ábrázolás esetén lineáris próbálással.

20. Gráfok alapfogalmai, ábrázolása

A *gráf* az utolsó, de igen lényeges adatszerkezet-forma, amelybe betekintést nyerünk a tárgy során. Mint ismert, a gráfokat legkönnyebben úgy írhatjuk le, mint pontokat, és az őket összekötő éleket. Ebből a tárgyból általában *egyszerű gráf*okkal foglalkozunk, amelyekben nincsenek párhuzamos, illetve hurokélek. A szokásos matematikai jelöléssel egy G gráfot egy $G = (V, E)$ párral adunk meg, ahol V a *csúcsok* halmaza, $E \subseteq V \times V$ pedig az *élek* halmaza. (Megjegyezzük, hogy ez a formalizmus nem egyszerű gráfok esetén valójában pontatlan.) Sok feladatban előfordul, hogy a gráf éleit súlyozzuk. Ilyenkor egy $G = (V, E, \varrho)$ *súlyozott* gráfról beszélünk, ahol $\varrho: E \to \mathbb{R}$ a súlyfüggvény**.** Továbbá megkülönböztetünk *irányított* és *irányítás nélküli gráf*okat. Irányítatlan gráfok esetén egy *u*-ból *v*-be vezető és egy *v*-ből *u*-ba vezető élt nem különböztetünk meg, tehát eltekintünk az élek irányításától. A továbbiakban a gráf csúcsainak számát *n*-nel, az élek számát pedig *e*-vel fogjuk jelölni.

A gráfok sok gyakorlati feladat reprezentálására alkalmazhatók, ennek megfelelően számos tulajdonság szerint lehet osztályozni őket, sok kérdés merül fel velük kapcsolatban, és egy kérdésre több különböző algoritmus is megoldást nyújthat.

A gráfok alapvetően kétféleképpen ábrázolhatók, függetlenül attól, hogy irányítottak-e, illetve súlyozottak-e.

Az egyik megközelítés a *csúcsmátrix* (C), amely leginkább egyszerű gráfok ábrázolására alkalmazható. Ekkor a gráf csúcsai a mátrix indexei függőlegesen, illetve vízszintesen is, és a mátrixban megadhatók az egyik pontból a másikba vezető élek. Súlyozatlan esetben logikai értékekként adhatók meg az élek, tehát $C[u, v] = igaz$, ha van él az u csúcsból a v csúcsba, különben *hamis*. Súlyozott esetben végtelent írunk oda, ahol nincs él, míg a súlyértéket, ha van. Amennyiben hurokéleket nem akarunk ábrázolni, a főátlóba általában 0 értékeket írunk (hiszen egy csúcsból önmagába 0 súlyozottságú él vezet). Amennyiben a gráfunk irányítatlan, akkor a mátrix szimmetrikus lesz, hiszen $C[u, v] = C[v, u]$. Többszörös éleket csúcsmátrix segítségével nehezen tudunk ábrázolni, ezért ilyenkor érdemes inkább az alábbi módszert alkalmazni.

A másik ábrázolási lehetőség az *éllista*. Felveszünk egy tömböt (Adj), amely a csúcsokkal van indexelve, és listák fejmutatóit tartalmazza. Minden ilyen listában helyet kapnak az abból a csúcsból induló élek, így egy listaelem adat része a célcsúcsból, valamint az esetleges súlyozásból fog állni. Tehát $Adj[u]$ megadja az *u* csúcsból induló élek listáját. Amennyiben a gráf irányítatlan, akkor minden élt kétszer veszünk fel a listákra, tehát mindkét irányítását eltároljuk.

20.1. Feladat:

a) Egy fogadás alkalmával a résztvevők kézfogással üdvözlik egymást. Mindenki mindenkivel kezet fog, közben mindenki megjegyzi, hogy hány emberrel fogott kezet. A fogadás végén a főnök összegzi ezeket a számokat. Mutassuk meg, hogy az eredmény páros szám. Igaz ez akkor is, ha nem fog kezet mindenki mindenkivel?

b) Egy kilenc tagú társaságban mindenki pontosan öt másik embernek átad 100Ft-ot. Bizonyítsuk be, hogy az ajándékozás után van két olyan ember, akinek ugyanannyi forinttal változott a pénze.

c) Tíz csapat körmérkőzést játszik. Mindegyik pár pontosan egyszer mérkőzik, és a mérkőzések nem végződhetnek döntetlennel. A győzelemért 1 pont, a vereségért 0 pont jár. Bizonyítsuk be, hogy a csapatok pontszámainak négyzetösszege nem lehet több 285-nél.

d) Egy klubesten 7 lány és 7 fiú vett részt. Az est végén mindenki felírta egy cédulára hány különböző partnerrel táncolt (tegyük fel, hogy csak különböző neműek táncoltak együtt). A cédulákon az alábbi számok szerepeltek: 3, 3, 3, 3, 3, 5, 6, 6, 6, 6, 6, 6, 6, 6. Bizonyítsuk be, hogy valaki tévedett.

e) Bizonyítsuk be, hogy egy 6 tagú társaságban van 3 ember, akik ismerik egymást, vagy van 3 ember, akik közül egyik sem ismeri a másik kettőt.

20.2. Feladat:

a) Bizonyítsuk be, hogy egy irányítatlan gráfban a páratlan fokú csúcsok száma páros.

b) Bizonyítsuk be, hogy egy irányítatlan gráfban a csúcsok fokszámainak összege megegyezik az élek számának kétszeresével, vagyis $\sum_{u \in V} d(u) = 2|E|$.

c) Bizonyítsuk be, hogy egy legalább kétpontú, egyszerű, irányítás nélküli gráfban mindig van két olyan pont, amelyek fokszáma azonos.

d) Bizonyítsuk be, hogy ha egy $G = (V, E)$ irányított vagy irányítatlan gráfban tetszőleges $u, v \in V$ csúcsok összeköthetők úttal, akkor ezek a csúcsok egyszerű úttal is összeköthetők.

e) Tegyük fel, hogy a *G* összefüggő, irányítatlan gráf leghosszabb útjának a hossza *m*. Mutassuk meg, hogy amennyiben van két *m* hosszú út, akkor ezeknek van közös pontjuk.

f) Bizonyítsuk be, hogy ha egy irányítatlan gráf minden csúcsának a fokszáma legalább kettő, akkor van a gráfban kör.

g) Bizonyítsuk be, hogy ha egy 2n pontú egyszerű, irányítatlan gráfban minden pont fokszáma legalább n, akkor a gráf összefüggő.

20.3. Feladat:

Az alábbi ábrán látható karikák egy telken álló fákat jelképeznek. Az *A* jelű fán egy cinke a *B* jelű fán egy rigó ül. Percenként mindkét madár átrepül a tőle északi, déli, keleti vagy nyugati irányban lévő legközelebbi fára (átlós irány tilos). Lehetséges-e, hogy valamikor mindkét madár azonos fán lesz?

A O O O O B
O O O O O O
O O O O O O
O O O O O O
O O O O O O
O O O O O O

20.4. Feladat:

Egy n pontú irányítatlan, teljes gráf éleit (felváltva egy-egy élt) az A és B jelű játékosok kiszíneznek pirosra. Az veszít, aki elsőként hoz létre piros kört. Melyik játékosnak van nyerő stratégiája, ha A kezdi a színezést?

20.5. Feladat:

Egy országban n város van. Bármely két város között vezet egy egyirányú, közvetlen út. Bizonyítsuk be, hogy kijelölhető egy város úgy, hogy minden város elérhető a kijelölt városból legfeljebb egyetlen másik város érintésével.

20.6. Feladat:

Adjuk meg az alábbi algoritmusok leírását struktogram segítségével csúcsmátrixos, valamint éllistás ábrázolás mellett irányított, illetve irányítás nélküli, súlyozott, illetve súlyozatlan gráfon.

- Új (adott költségű) él beszúrása két csúcs közé ($AddEdge(u, v, c)$).
- Két csúcs közötti él törlése ($DeleteEdge(u, v)$).
- Annak megállapítása, hogy egy csúcs szomszédja-e egy másiknak ($Adjacent(u, v)$).
- Egy csúcs szomszéd halmazának megállapítása ($Adjacents(u)$).
- Egy csúcs (kimeneti, illetve bemeneti) fokszámának megállapítása ($OutDegree(u)$ és $InDegree(u)$, illetve irányítatlan esetben $Degree(u)$).

20.7. Feladat:

Adjunk algoritmust, amely meghatározza egy irányított gráf esetén a gráf csúcsainak kimeneti fokát és bemeneti fokát. A program az értékeket a $kifok[1..n]$ és $befok[1..n]$ tömbökbe írja be, amely tömbök a csúcsok sorszámával vannak indexelve.

a) A gráf csúcsmátrixos ábrázolással van megadva, az algoritmus futási ideje legyen $O(n^2)$.

b) A gráf éllistás ábrázolással van megadva, az algoritmus futási ideje legyen $O(n + e)$.

20.8. Feladat:

Adjunk algoritmust, amely egy gráf csúcsmátrixos ábrázolásából előállítja a gráfot éllistás ábrázolásban.

a) A gráf legyen élsúlyozás nélküli.

b) A gráf legyen élsúlyozott.

20.9. Feladat:

Adjunk algoritmust, amely egy gráf éllistás ábrázolásából előállítja a gráfot csúcsmátrixos ábrázolásban.

a) A gráf legyen élsúlyozás nélküli.

b) A gráf legyen élsúlyozott.

20.10. Feladat:

Adjunk $O(n + e)$ idejű algoritmust egy éllistával ábrázolt $G = (V, E)$ irányítatlan és súlyozatlan gráf éllistás ábrázolásban megadott egyszerűsített gráfjának, $G' = (V, E')$-nek az előállítására. $G' = (V, E')$ gráf a $G = (V, E)$ egyszerűsített gráfja, ha E' ugyanazokat az éleket tartalmazza, mint E, azzal a különbséggel, hogy E'-ben egy éllel helyettesítjük az E-beli többszörös éleket, és a hurokéleket elhagyjuk.

20.11. Feladat:

Adjunk algoritmust, amely előállítja egy irányított gráf transzponáltját/megfordítottját. $G = (V, E)$ irányított gráf transzponáltja a $G^T = (V, E^T)$ irányított gráf, ha $E^T = \{(u, v) \in V \times V \mid (v, u) \in E\}$, azaz G^T-t úgy kapjuk G-ből, hogy minden él irányítását megfordítjuk.

a) Csúcsmátrixos ábrázolás esetén, konstans mennyiségű többletmemória felhasználásával.

b) Éllistás ábrázolás esetén, tetszőleges mennyiségű többletmemória felhasználásával.

c) Éllistás ábrázolás esetén, konstans mennyiségű többletmemória felhasználásával.

20.12. Feladat:

Adott $G = (V, E)$ és $G' = (V, E')$ véges, éllistás, súlyozatlan, irányított gráfok. Adjunk algoritmust, amely eldönti, hogy G' **részgráfja**-e G-nek

a) más adatszerkezet használatának elkerülésével.

b) egy n hosszú indikátortömb használatával, amely logikai értékeket vehet fel.

20.13. Feladat:

Adott $G = (V, E)$ és $G' = (V, E')$ véges gráfok. Adjunk algoritmust, amely eldönti, hogy G' feszített **részgráfja**-e G-nek

a) Irányítatlan gráfok esetén.

b) Irányított gráfok esetén.

20.14. Feladat:

Adott G irányított vagy irányítatlan, súlyozatlan gráf. Adjunk algoritmust, amely előállítja G **komplementergráfját** (azt a gráfot, amelyben az élek pont fordítottan állnak az eredetihez képest)

a) egy másik gráfban csúcsmátrixos reprezentáció mellett.

b) egy másik gráfban éllistás reprezentáció mellett.

c) ugyanabban a gráfban éllistás reprezentáció mellett, úgy, hogy felveszünk egy n hosszú indikátortömböt, amely logikai értékeket vehet. A tömböt használjuk arra, hogy eltároljuk, mely éleket kell a komplementerben felvenni.

20.15. Feladat:

Adott egy számpárokat tartalmazó sorozat, ahol a számpárok egy **konvex sokszög** síkbeli koordinátáit tartalmazzák. Írassuk ki az **átlókat** csúcspontjaik koordinátáival.

20.16. Feladat:

Csúcsmátrixos ábrázolás esetén a legtöbb gráfalgoritmus futási ideje $\Theta(n^2)$, azonban van kivétel. Egy irányított gráf csúcsa univerzális nyelő, ha bemeneti foka $n - 1$ és kimeneti foka 0. Mutassuk meg, hogy eldönthető $O(n)$ idő alatt, hogy egy csúcsmátrixszal megadott irányított gráfban van-e **univerzális nyelő** csúcs.

20.17. Feladat:

Turnamentnek hívunk egy olyan irányított gráfot, melyben bármely két pontot pontosan egy él köt össze. Amennyiben teljes körmérkőzést játszik n versenyző, olyan sportágban, ahol nincs döntetlen (pl.: ping-pong), akkor az eredményeket egy n pontú tournamenttel ábrázolhatjuk, ahol i-ből j-be vezet él, ha i-edik versenyző legyőzte a j-edik versenyzőt. Adjunk algoritmust, amely eldönti, hogy van-e **abszolút győztes** (aki mindenki mást legyőzött).

20.18. Feladat:

Adott egy n pontú irányítatlan gráf. Adjunk algoritmust, amely eldönti, hogy van-e a gráfnak olyan pontja, amely minden más ponttal össze van kötve! Nevezzük ezt a pontot röviden **teljes pontnak**.

20.19. Feladat:

Adott $G = (V, E)$ irányított gráf. Állítsuk elő a $G' = (V, E')$ irányított gráfot, ahol $E' = \{(u, w) \in V \times V \mid \exists v \in V: (u, v) \in E \text{ és } (v, w) \in E\}$.

a) A gráf csúcsmátrixos ábrázolással van megadva.

b) A gráf éllistás ábrázolással van megadva.

21. Szélességi bejárás

Leírás:

A legalapvetőbb feladat a gráfokkal kapcsolatban, hogy fel tudjuk térképezni azok szerkezetét, azaz meg tudjuk határozni egy kezdőcsúcsból indulva milyen éleken tudunk eljutni az egyes csúcsokhoz. A feladatnak két megoldása van, a *szélességi* (*breadth-first search*, *BFS*), illetve a *mélységi* (*depth-first search*, *DFS*) *bejárás*. A szélességi bejárás alkalmas egy forrásból induló legrövidebb utak keresésére is, ha minden él költségét azonosnak tekintjük, vagyis ha legkevesebb élből álló utakat keresünk. Ezen kívül mindkét algoritmusnak számos további alkalmazása van, amelyek közül néhányat feladatok formájában tárgyalunk.

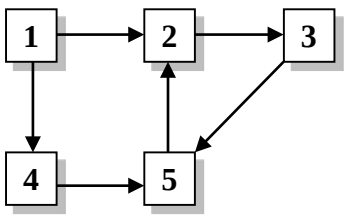
A szélességi bejárás egy kezdő csúcsból feltérképezi annak szomszédait, majd ezeket sorba veszi, és azok szomszédait is feltérképezi, és így tovább, tehát minél hosszabb út vezet egy csúcshoz a kezdőcsúcsból, annál később kerül sorra.

Először minden csúcs távolságát végtelenre, szülő csúcsát pedig *NIL*-re állítjuk, kivéve a kezdőcsúcsnál, amelynek távolsága 0 lesz. Amikor az algoritmus elér egy csúcsot (először a kezdőcsúcsot), akkor berakja azt egy sorba. Majd kiveszi a sor első elemét, és az ő szomszédait is berakja a sorba (*S*), ha még korábban nem kerültek bele, és addig dolgozik, amíg van elem a sorban. Nyilván fel kell jegyeznünk a már elért csúcsokat, hogy ne kerüljenek bele újra a sorba. Az elkülönítést végezhetjük úgy, hogy egy külön halmazba gyűjtjük az elért csúcsokat (*ELÉRT* vagy *Reached*) vagy *színezést* vezetünk be, és *fehérek* lesznek a még el nem ért csúcsok, *szürkék* pedig az elértek. Gyakran bevezetnek egy harmadik színt is (*fekete*) a feldolgozott csúcsok jelölésére. Ezen felül feljegyezzük egy tömbben a kezdőcsúcstól vett távolságot (*d* tömb), illetve a közvetlen szülő csúcsot (*Π* tömb), azaz hogy mely csúcsból jutottunk el az adott csúcshoz.

Az algoritmus $O(n+e)$ lépésben eléri egy véges gráf összes olyan csúcsát, amely a kezdőpontból elérhető, ez teljes indukcióval igazolható. Amennyiben a gráf irányítatlan és összefüggő, illetve irányított és gyökeresen összefüggő – azaz a kezdőpontból, mint gyökérből minden további pontba létezik irányított út –, akkor az algoritmus eléri a gráf összes csúcsát.

Példa:

Futtassuk le az algoritmust az alábbi irányított gráfon az 1-es csúcsból kiindulva. Az algoritmus lépéseit ábrázoljuk egy táblázatban.



Az algoritmus első lépésben eléri az 1-es csúcsot, feljegyzi, hogy az ő távolsága 0, beteszi az *ELÉRT* halmazba, a szülőjét pedig *NIL*-re állítja. Az 1-es csúcs szomszédai a 2-es és a 4-es, ezért azokat beteszi az *S* sorba.

Ezzel a módszerrel az algoritmus feltölti a *d* és *Π* tömböket, és egészen addig halad, amíg az *S* sor ki nem ürül.

Végül az *ELÉRT* halmaz tartalmazni fogja az összes csúcsot, ugyanis mindegyik elérhető a kezdőcsúcsból.

Az algoritmus lefutása:

	S	ELÉRT	*d*					*Π*				
			1	**2**	**3**	**4**	**5**	**1**	**2**	**3**	**4**	**5**
1	1	1	0	∞	∞	∞	∞	*NIL*	*NIL*	*NIL*	*NIL*	*NIL*
2	2, 4	1, 2, 4	0	**1**	∞	**1**	∞	*NIL*	**1**	*NIL*	**1**	*NIL*
3	4, 3	1, 2, 3, 4	0	1	**2**	1	∞	*NIL*	1	**2**	1	*NIL*
4	3, 5	1, 2, 3, 4, 5	0	1	2	1	**2**	*NIL*	1	2	1	**4**
5	5	1, 2, 3, 4, 5	0	1	2	1	2	*NIL*	1	2	1	4
6		1, 2, 3, 4, 5	0	1	2	1	2	*NIL*	1	2	1	4

21.1. Feladat:

Adjuk meg a szélességi bejárás algoritmusát stuktogrammal. Használjunk egy sort az aktuális csúcsok tárolására, és egy halmazt az elért csúcsok tárolására.

a) A gráfot tekintsük ADT szinten, használjuk a $Szomszéd(u)$ függvényt.

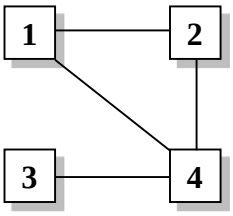
b) A gráfot ábrázoljuk csúcsmátrixszal.

c) A gráfot ábrázoljuk éllistával.

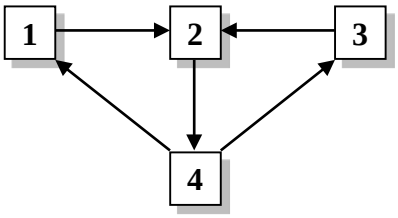
21.2. Feladat:

Szemléltessük a szélességi bejárást az alábbi gráfon az 1-es csúcsból kiindulva. Adjuk meg a sor, az *ELÉRT* halmaz, valamint a *d* és a *Π* tömb tartalmát menetenként. (Egy menet egy csúcs kivétele a sorból és feldolgozása.)

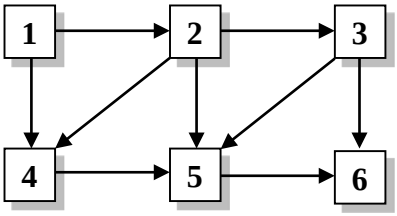
a)



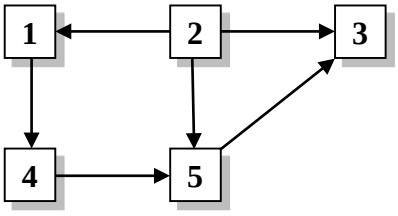
b)



d)



d)



21.3. Feladat:

Milyen szélességi fát határoz meg az alábbi *Π* tömb, melyet egy szélességi bejárás után kaptunk?
Π = [2, 6, 2, 7, 10, 0, 10, 7, 2, 6]

21.4. Feladat:

Elemezzük a szélességi keresés műveletigényét.

21.5. Feladat:

Adjunk algoritmust, amely a szélességi bejárás lefuttatása után kiírja egy $v \in V$ csúcsra az egyik **legrövidebb $s \rightsquigarrow v$ utat**.

21.6. Feladat:

Adott egy *G* irányítatlan gráf. Adjunk algoritmust, amely eldönti, hogy **páros-e a gráf**.

a) *G* összefüggő.

b) *G* nem biztos, hogy összefüggő.

21.7. Feladat:

Adott egy *G* irányítatlan gráf. Adjunk algoritmust, amely eldönti, hogy G **tartalmaz-e kört**. (Irányítatlan esetben körnek legalább 3 pontból álló részgráfot tekintünk.)

a) *G* összefüggő.

b) *G* nem feltétlenül összefüggő.

21.8. Feladat:

Adott egy *G* irányított gráf. Adjunk algoritmust, amely eldönti, hogy *G* **tartalmaz-e** – irányítástól eltekintett – **kört**. (Irányított esetben egy két pontból álló részgráf is tekinthető körnek.)

a) *G* gyökeresen összefüggő egy *s* csúcsból.

b) *G* nem feltétlenül összefüggő.

21.9. Feladat:

Adott egy *G* **irányítatlan gráf**. Adjunk algoritmust, amely *G* **komponenseit kiszínezi**. (Az azonos komponensbe eső csúcsokat azonos, a különböző komponensbe eső csúcsokat különböző színűre.)

21.10. Feladat:

Adott egy *G* **irányított gráf**. Adjunk algoritmust, amely *G* (gyengén) összefüggő **komponenseit kiszínezi**. (Az azonos komponensbe eső csúcsokat azonos, a különböző komponensbe eső csúcsokat különböző színűre.)

a) Csúcsmátrixos ábrázolás estén.

b) Éllistás ábrázolás esetén.

21.11. Feladat:

Adjunk algoritmust, amely megoldja az alábbi feladatot. Adott egy n × n-es **sakktábla**, egy huszárral (lóval), egy induló mezőről jussunk el egy célmezőre a lehető legkevesebb szabályos lépéssel.

21.12. Feladat:

Legyen G egy élsúlyozott gráf, amelynek minden élsúlya természetes szám. A szélességi **bejárás algoritmusát átalakítva** adjunk algoritmust, amely minden csúcsra megadja egy $s \in V$ csúcsból az illető csúcsba vezető legkisebb költségű utat.

22. Minimális költségű utak: Dijkstra

Leírás:

Gyakran előfordul, hogy élsúlyozott gráfok egy adott csúcsából szeretnénk meghatározni a legrövidebb utat egy vagy több másik csúcsba. Ilyen feladat például a legrövidebb vagy leggyorsabb autóút, illetve a legolcsóbb repülőút meghatározása. Az első algoritmusunk erre a problémára a *Dijkstra-algoritmus*, amely negatív költségű éleket nem tartalmazó gráfokra alkalmazható.

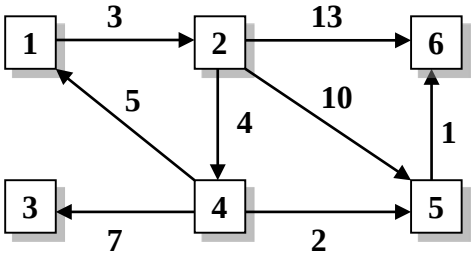
Az algoritmus a szélességi bejáráshoz hasonlóan nyilvántartja a csúcsok megelőzőét (Π), a távolságot a kezdőcsúcstól (d), illetve, hogy befejeztük-e az adott csúcs feldolgozását, azaz *kész* van-e a csúcs. Egy csúcsot akkor minősítünk késznek, ha már megtaláltuk az oda vezető legrövidebb utat. Ennek jelölésére bevezethetünk egy *KÉSZ* (vagy *Finished*, *Ready*, *Processed*) halmazt, illetve színezhetjük a csúcsokat. Itt három színt szoktak használni: *fehérrel* színezzük a nem elért csúcsokat, *szürkével* az elérteket, *feketével* pedig a készeket.

A szélességi bejárással ellentétben egy-egy él vizsgálatakor nem egyet adunk a távolsághoz, hanem az él költségét. Az algoritmus minden iterációban az eddig legkisebb távolsággal elért csúcsot választja ki a nem készek közül, majd megvizsgálja, hogy a szomszéd csúcsokba a kiválasztott csúcson keresztül vezet-e az eddig megtaláltnál rövidebb út. Amennyiben igen, akkor lecseréljük a csúcs megelőzőjét, illetve távolságát. Az algoritmus addig dolgozik, amíg ki nem választott minden elérhető csúcsot, vagyis mindegyik be nem került a készek halmazába.

A távolságokat szokás egyszerű tömbbel, illetve kupaccal is nyilvántartani, utóbbi esetben a gyors minimum-kiválasztás érdekében. Persze ekkor gondoskodnunk kell a kupac helyreállításáról is minden lépésben. Ez befolyásolja az algoritmus műveletigényét, ami tömbös esetben $O(n^2 + e)$, kupacos esetben $O((n + e)\log n)$. A szélességi bejáráshoz hasonlóan az irányítatlan és összefüggő, illetve irányított és a kezdőpontból gyökeresen összefüggő gráfok esetén az összes csúcsot eléri.

Ha a gráfban vannak negatív élköltségek, akkor a Dijkstra-algoritmus nem alkalmazható, nem megfelelő az a mohó kiválasztási módszer, amelyet követ (minden iterációban a minimális távolságú nem kész csúcsot választjuk). Erre az esetre majd a Bellman–Ford-algoritmus fog megoldást nyújtani.

Példa:



Futtassuk le az algoritmust az alábbi irányított gráfon az 1-es csúcsból kiindulva.

Az algoritmus először minden csúcs távolságát beállítja végtelenre, szülőjét pedig *NIL*-re. Első lépésben eléri az 1-es csúcsot, feljegyzi, hogy az ő távolsága 0. Az 1-es csúcs egyetlen szomszédja a 2-es, ezért annak távolsága ezután 3 lesz, szőlője az 1-es, míg az 1-es csúcs bekerül a kész halmazba. Hasonlóan a 2-es csúcsból elérjük a 4, 5, 6 csúcsokat, és a 4-esből a 3-ast.

Amikor a 4-esből is elérjük az 5-öst, az ő korábbi távolsága 13 volt, míg a 4-esé 7 volt, így az új úton 9 távolsággal találtuk meg, és mivel ez rövidebb a korábbinál, felülírjuk a 5-ös csúcs szülejét a 4-esre, távolságát pedig 9-re.

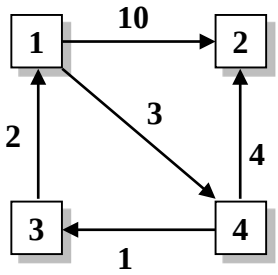
Nézzük a teljes lefutást táblázaton keresztül.

	KÉSZ	***d***						***Π***					
		1	**2**	**3**	**4**	**5**	**6**	**1**	**2**	**3**	**4**	**5**	**6**
1		**0**	∞	∞	∞	∞	∞	*NIL*	*NIL*	*NIL*	*NIL*	*NIL*	*NIL*
2	1	0	**3**	∞	∞	∞	∞	*NIL*	**1**	*NIL*	*NIL*	*NIL*	*NIL*
3	1, 2	0	3	∞	**7**	**13**	**16**	*NIL*	1	*NIL*	**2**	**2**	**2**
4	1, 2, 4	0	3	**14**	7	9	16	*NIL*	1	**4**	2	**4**	2
5	1, 2, 4, 5	0	3	14	7	9	**10**	*NIL*	1	4	2	4	**5**
6	1, 2, 4, 5, 6	0	3	14	7	9	10	*NIL*	1	4	2	4	5
7	1, 2, 4, 5, 6, 3	0	3	14	7	9	10	*NIL*	1	4	2	4	5

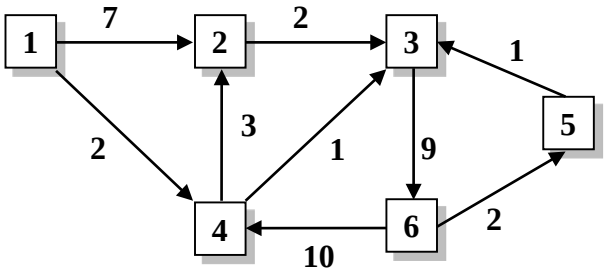
22.1. Feladat:

Szemléltessük a Dijkstra-algoritmus működését az alábbi gráfokon az 1-es csúcsból kiindulva. Adjuk meg menetenként a *KÉSZ* halmazt, valamint a *d* és *Π* tömbök tartalmát. (Egy menet egy csúcs prioritásos sorból való kivétele és feldolgozása.)

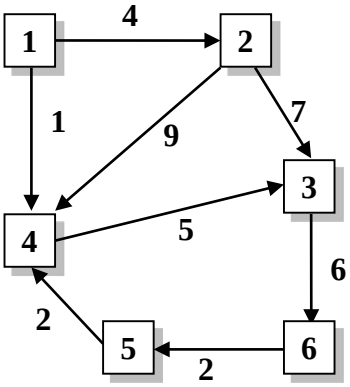
a)



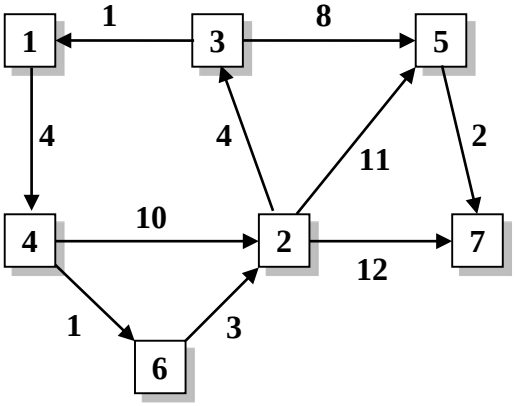
b)



b)



d)



22.2. Feladat:

A Dijkstra-algoritmus lefuttatása után a d tömbök alábbi, menetenkénti sorozatát kaptuk. **Mi lehetett a gráf?** Próbáljuk behúzni azokat az éleket, amelyek a d-beli értékekből következnek, és próbáljuk előállítani a Π tömbök menetenkénti sorozatát.

1	**2**	**3**	**4**	**5**	**6**
0	∞	∞	∞	∞	∞
0	**2**	∞	∞	3	∞
0	2	6	4	**3**	∞
0	2	5	**4**	3	9
0	2	**5**	4	3	9
0	2	5	4	3	**9**

22.3. Feladat:

Adjuk meg a Dijkstra-algoritmust az **ábrázolás szintjén** stuktogram vagy pszeudokód segítségével

a) csúcsmátrixos ábrázolás esetén;

b) éllistás ábrázolás esetén.

22.4. Feladat:

Mivel a Dijkstra-algoritmus nem működik **negatív élsúlyok esetén**, ha a gráfunk tartalmaz negatív élsúlyt, keressük meg a legkisebb élsúlyt, és ennek abszolút értékével növeljük meg az összes él súlyértékét. Ezután a gráf már nem tartalmaz negatív élsúlyt. Ezután lefuttatva a Dijkstra-algoritmust, megkapjuk a legrövidebb utakat, melyeknek valódi költségét az eredeti súlyértékek felhasználásával már könnyen kiszámíthatjuk. Tegyük fel, hogy a gráfunk nem tartalmaz negatív összköltségű kört. Bizonyítsuk vagy cáfoljuk a fenti elgondolást.

22.5. Feladat:

a) Adottak **repülőjáratok** induló és cél állomássokkal, továbbá a repülőjegyárakkal. Jussunk el repülővel A városból B városba a legolcsóbb úton.

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

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

22.6. Feladat:

Egy **kommunikációs hálózatot** egy irányítatlan gráffal modellezünk, ahol $(u, v) \in E$ él, az u-t és a v-t összekötő kommunikációs csatornát szimbolizálja. Az élekhez hozzárendeltük annak valószínűségét, hogy a csatorna működik (nem sérült). Az egyes csatornák működőképessége egymástól független. Adjunk algoritmust két pont közötti **legmegbízhatóbb út** előállítására.

22.7. Feladat:

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

22.8. Feladat:

Egy kitűzött időpontban A városból elindulva **menetrendszerinti járatokkal** minél gyorsabban jussunk el B városba. Adottak a használható közlekedési eszközök menetrendjei (repülők, hajók vonatok stb.). Tekintsünk el egy városon belül az egyes érkezési-indulási helyszínek (reptér-kikötő stb.) közötti eljutási időktől.

22.9. Feladat:

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

22.10. Feladat:

a) Legyen G súlyozott gráf $c: E \to \{0, \dots, W-1\}$ egész értékű súlyfüggvénnyel. Módosítsuk a Dijkstra-algoritmust, hogy a műveletigénye $O(W \cdot n + e)$ legyen.

b) Módosítsuk az előző részben kapott algoritmust $O((n+e)\, log\, W)$ futási idejűre.

23. Minimális költségű utak: Bellman-Ford

Leírás:

A Dijkstrával ellentétben a *Bellman–Ford-algoritmus* már negatív élköltségeket tartalmazó gráfokra is alkalmazható, viszont nyilván továbbra is ki kell zárnunk a negatív összköltségű köröket, hiszen ebben az esetben bizonyos csúcsok között nem létezik legrövidebb út.

Mivel negatív körök nincsenek, ezért minden úthoz található egy nála nem hosszabb egyszerű út, így elég csak ilyeneket keresnünk. Egy n pontú gráfban egy egyszerű út legfeljebb $n-1$ élből állhat, ezért az algoritmus $n-1$ iterációt hajt végre, és mindegyikben minden egyes él mentén megpróbálja javítani a végpontjához feljegyzett d és Π értékeket (a kezdőpontjába talált út folytatásával). Ez a javítás ugyanúgy történik, mint a Dijkstra-algoritmusban, a különbség abban áll, hogy itt nem tudunk olyan mohó kiválasztási módszert követni, így egy csúcsból kivezető éleket nem elég egyszer megvizsgálnunk, minden iterációban bármelyik élre szükség lehet. A kezdeti inicializálás megegyezik a szélességi bejárásnál és a Dijkstra-algoritmusnál látott módszerrel.

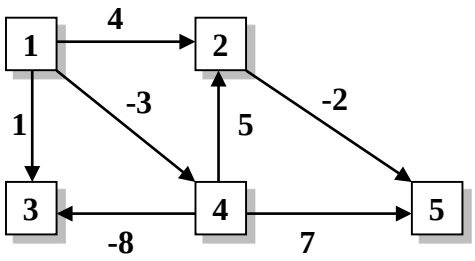
A Bellman–Ford-algoritmus az i-edik menet végére minden csúcsba megtalál egy olyan egyszerű utat, amelynek költsége nem nagyobb, mint bármelyik legfeljebb i élből álló út költsége. Vagyis az algoritmus „előreszaladhat”, az i-edik iterációban i-nél több élből álló utakat is megtalálhat, ugyanis az egyes menetekben felhasználjuk az aktuális menet részeredményeit is. Vagyis az $(n-1)$-edik iteráció végére megtaláljuk minden csúcsba egy legrövidebb (egyszerű) utat.

Az algoritmus műveletigénye $\theta(n \cdot e)$, azaz nagyságrendileg lassabb, mint a Dijkstra-algoritmus.

Példa:

Futtassuk le az algoritmust az alábbi irányított gráfon az 1-es csúcsból kiindulva.

Első lépésben az algoritmus felülírja a 2-es, 3-as és 4-es csúcsok távolságát, hiszen azok a 1-es csúcson keresztül kisebb értékkel elérhetőek, mint a végtelen. Második lépésben az algoritmus már az 1, 2, 3, és 4 csúcsokból elérhető csúcsokoat vizsgálja, és például a 3-as csúcs a 4-esen keresztül -11 nagyságú úton érhető el, míg korábban 1 nagyságúval értük el, ezért felül kell írnunk az értékét.



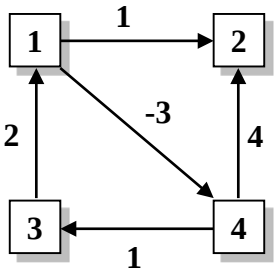
Az algoritmus teljes lefutása:

	d					***Π***				
	1	**2**	**3**	**4**	**5**	**1**	**2**	**3**	**4**	**5**
1	0	∞	∞	∞	∞	*NIL*	*NIL*	*NIL*	*NIL*	*NIL*
2	0	**4**	**1**	**-3**	∞	*NIL*	**1**	**1**	**1**	*NIL*
3	0	**2**	**-11**	-3	**7**	*NIL*	**4**	**4**	1	**4**
4	0	2	-11	-3	**0**	*NIL*	4	4	1	**2**

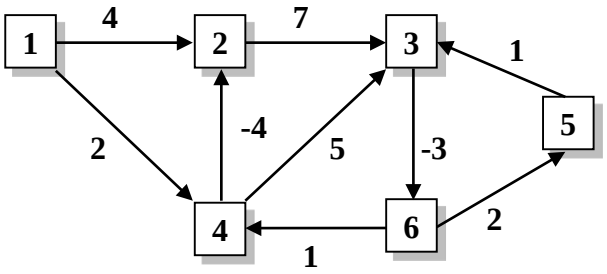
23.1. Feladat:

Szemléltessük a Bellman–Ford-algoritmus működését az alábbi gráfon. Rajzold fel a d és Π tömbök tartalmát iterációnként $(1..n-1)$.

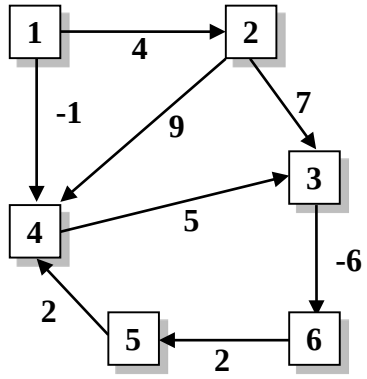
a)



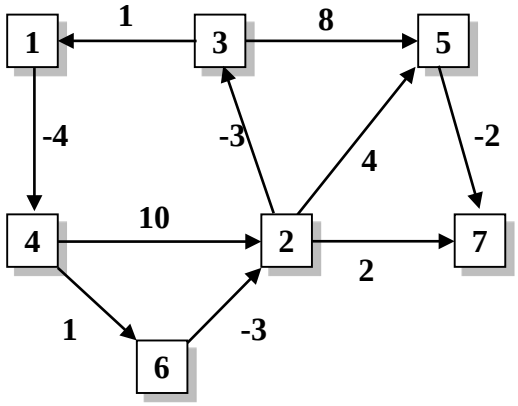
b)



c)



d)



23.2. Feladat:

Készítsük el a Bellman–Ford-algoritmust az **ábrázolás szintjén** stuktogram vagy pszeudokód segítségével

a) csúcsmátrixos ábrázolás esetén;

b) éllistás ábrázolás esetén.

23.3. Feladat:

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

23.4. Feladat:

a) Gyorsítsuk a Bellman–Ford-algoritmust úgy, hogy ha **egy iterációban nem volt változás**, akkor megállunk.

b) Gyorsítsuk a Bellman–Ford-algoritmust úgy, hogy **bizonyos éleket nem vizsgálunk**.

c) Tegyük fel, hogy az egyszerű utak **élszáma legfeljebb** konstans k. Gyorsítsuk a Bellman–Ford-algoritmuson.

23.5. Feladat:

Egészítsük ki a Bellman–Ford-algoritmust úgy, hogy egy logikai változóban visszaadja, hogy a gráfban **van-e negatív összköltségű kör** vagy sem, mivel az algoritmus által kiszámított távolságok csak akkor érvényesek, ha ilyen kör nincs a gráfban.

23.6. Feladat:

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

23.7. Feladat:

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

23.8. Feladat:

Arbitrázs a valutaváltási árfolyamokban rejlő egyenlőtlenségek olyan hasznosítása, amikor egy valuta 1 egységéből elindulva, egy valutaváltási sorozat lefolytatása után ugyanazon valuta 1 egységénél nagyobb értékére teszünk szert. (Pl.: 1 dollárért veszünk 0,7 fontot, majd a fontot átváltjuk frankra 9,5-es szorzóval, majd a frankot ismét dollárra váltjuk 0,16-os szorzóval, ekkor 0,7*9,5*0,16=1,064. Tehát 6,4%-os haszonra tettünk szert.) Adjunk algoritmust, amely meghatározza, hogy létezik-e a valutáknak ilyen sorozata, és ha létezik, adjunk meg egy ilyet.

23.9. Feladat:

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

24. Legrövidebb utak és tranzitív lezárt

Floyd algoritmus

Leírás:

A feladatunk most az, hogy nem csak egy csúcsból, hanem az összes csúcsból kiindulva meghatározzuk a legrövidebb utakat. Erre nyilván a legegyszerűbb megoldást az jelenti, ha minden csúcsból kiindulva lefuttatjuk a Dijkstra vagy a Bellman–Ford-algoritmust, annak függvényében, hogy a gráf tartalmaz-e negatív költségű élt. Amennyiben igen, akkor $O(n^2e)$ műveletigényt kapunk, ami sűrű gráfok esetén $O(n^4)$. A Floyd algoritmus azonban egy nagyságrenddel gyorsabban, $O(n^3)$ időben oldja meg ezt a feladatot.

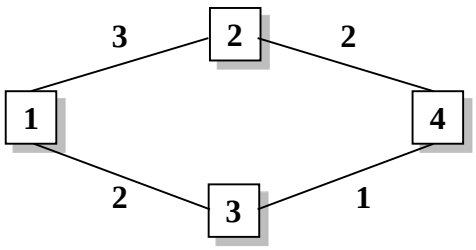
Floyd algoritmusában induljunk ki egy egyszerű, irányított vagy irányítatlan, súlyozott gráfból, amely tartalmazhat negatív élköltségeket is, de negatív összköltségű köröket nem. Az algoritmus a k-adik iterációban előállítja azokat a legrövidebb utakat, amelyeknek közbülső csúcsai k-nál nem nagyobb sorszámúak. (Az egyszerűség kedvéért tegyük fel, hogy $V = \{1, 2, \dots, n\}$.) Ezen utak hosszát tároljuk a $D^{(k)}$ mátrixokban ($k = 0, 1, \dots, n$). Ekkor $D^{(0)}$ megegyezik a gráf csúcsmátrixával, $D^{(k)}$-t pedig az alábbi formulával kaphatjuk meg $D^{(k-1)}$-ből.

$$D_{i,j}^{(k)} = \min\left\{D_{i,j}^{(k-1)},\ D_{i,k}^{(k-1)} + D_{k,j}^{(k-1)}\right\} \quad \forall i, j \in V$$

Ennek megfelelően az algoritmus n menetet végez $k = 1, \dots, n$ értékekre, és minden menetben minden $i, j \in V$ csúcspárra végrehajtunk egy $D[i,j] := min\{D[i,j], D[i,k] + D[k,j]\}$ értékadást. Kezdetben pedig a D mátrix a gráf csúcsmátrixával egyezik meg. Így nyilván egy $O(n^3)$ idejű algoritmust kapunk, amely működik negatív élköltségek esetén is, tehát nagyságrendileg sikerült javítanunk a Bellman–Ford-algoritmus többszöri futtatásának műveletigényén.

Példa:

Szemléltessük a Floyd-algoritmust a következő 4 csúcsbál alló irányítatlan gráfon. Első lépésben vegyük fel D mátrixnak a gráf csúcsmátrixát, azaz legyen



$$D^{(0)} = \begin{bmatrix} 0 & 3 & 2 & \infty \\ 3 & 0 & \infty & 2 \\ 2 & \infty & 0 & 1 \\ \infty & 2 & 1 & 0 \end{bmatrix}.$$

Első iterációs lépésben vegyük közbülső csúcsnak az 1-es csúcsot, vagyis vizsgáljuk az 1-es csúcsból kiinduló éleket, és ekkor a 2 és 3 csúcsok között találunk egy 3 + 2 = 5 hosszú utat, ami jobb, mint a korábbi végtelen, ezért felülírjuk, és az alábbi $D^{(1)}$ mátrixhoz jutunk.

A második iterációban olyan utakat keresünk, amelyeknél a belső csúcs sorszáma legfeljebb 2. Ekkor sikerül találnunk utat az 1-es és 4-es csúcsok között, ahol korábban nem volt út. Ennek hossza 3 + 2 = 5 lesz.

A harmadik iterációban már csak javító utakat vizsgálunk (hiszen a mátrixban már nincs végtelen érték). A 3-as csúcson átmenő, 1-ből 4-be vezető út hossza 3, ami rövidebb a korábbinál, ezért azt felülírjuk.

Végül, a negyedik lépésben megnézzük a 4-es csúcson áthaladó utakat, és találunk a 2-esből 3-asba vezető, 3 hosszú utat (ezt javítjuk), és a 2-esből az 1-esbe vezető 5 hosszú utat (ezt nem javítjuk), így megkapjuk a végeredményt.

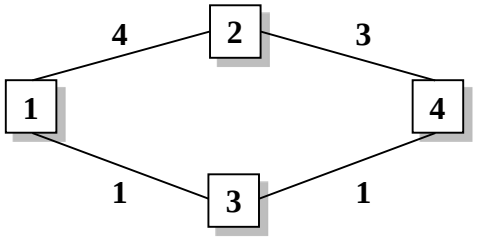
$$D^{(1)} = \begin{bmatrix} 0 & 3 & 2 & \infty \\ 3 & 0 & 5 & 2 \\ 2 & 5 & 0 & 1 \\ \infty & 2 & 1 & 0 \end{bmatrix} \qquad D^{(2)} = \begin{bmatrix} 0 & 3 & 2 & 5 \\ 3 & 0 & 5 & 2 \\ 2 & 5 & 0 & 1 \\ 5 & 2 & 1 & 0 \end{bmatrix}$$

$$D^{(3)} = \begin{bmatrix} 0 & 3 & 2 & 3 \\ 3 & 0 & 5 & 2 \\ 2 & 5 & 0 & 1 \\ 3 & 2 & 1 & 0 \end{bmatrix} \qquad D^{(4)} = \begin{bmatrix} 0 & 3 & 2 & 3 \\ 3 & 0 & 3 & 2 \\ 2 & 3 & 0 & 1 \\ 3 & 2 & 1 & 0 \end{bmatrix}$$

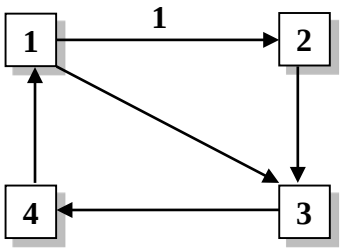
24.1. Feladat:

Alkalmazzuk a Floyd-algoritmust az alábbi gráfokra, és minden lépésben ábrázoljuk a változást a csúcsmátrixon:

a)



b)



Warshall algoritmus

Leírás:

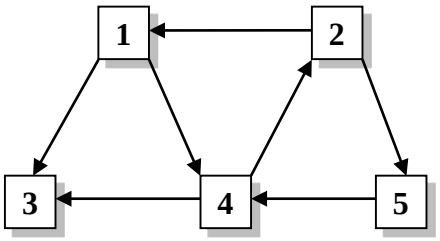
Egy gráf *tranzitív lezártja* az a gráf, amelyben két csúcs között pontosan akkor van él, ha az eredeti gráfban van út a két csúcs között, tehát egy olyan gráfot szeretnénk előállítani, amelynek élei az eredeti gráf útjai, és ehhez meg kell állapítanunk minden csúcspárra, hogy az eredeti gráfban van-e út közöttük. (Nyilván a probléma inkább irányított gráfokra érdekes.)

Ez a feladat nagyon hasonlít a Floyd-algoritmus feladatára, a különbség mindössze annyi, hogy nem az élköltségekre vagyunk kíváncsiak, pusztán az élek létezésére, azaz elég logikai értékeket használnunk. Így kapjuk a *Warshall-algoritmust*, amelynek műveletigénye nyilván megegyezik a Floyd-algoritmuséval, de a gyakorlatban gyorsabb lehet, hiszen logikai értékekkel egy számítógép hatékonyabban tud műveleteket végezni, mint számokkal.

Kezdetben vegyünk egy olyan W logikai mátrixot, amelyben $W[i,j]$ pontosan akkor igaz, ha van él i csúcsból j-be vagy $i = j$. Ezután minden k-adik iterációban minden $i, j \in V$ csúcspárra végrehajtunk egy $W[i,j] := W[i,j] \vee (W[i,k] \wedge W[k,j])$ értékadást.

Példa:

Tekintsük a következő irányított gráfot, és alkalmazzuk rá a Warshall-algoritmust. Induljunk ki egy olyan mátrixból, amely csak azt tartalmazza, hogy két csúcspont között van-e él, tehát legyen:

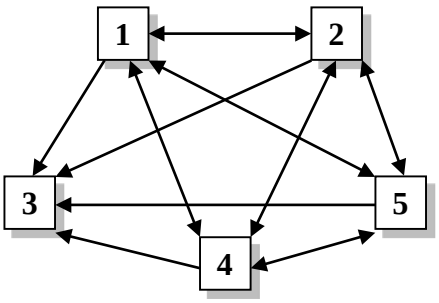


$$W^{(0)} = \begin{bmatrix} i & h & i & i & h \\ i & i & h & h & i \\ h & h & i & h & h \\ h & i & i & i & h \\ h & h & h & i & i \end{bmatrix}$$

Első iterációban nézzük meg, hogy az 1-es csúcson mint közbülső csúcson keresztül vezet-e új út a mátrixban. Ilyen lesz a $2 \to 1 \to 3$, és a $2 \to 1 \to 4$ út, ezeket bevesszük a mátrixba. Második iterációban a 2-es csúcson keresztül haladó utakat vizsgáljuk, így bekerül a $4 \to 2 \to 5$ és a $4 \to 2 \to 1$ út, tehát a 4-es csúcsból már minden csúcs elérhető lesz. A harmadik iterációban a 3-as csúcsból nem vezet ki él, ezért újabb utat nem találunk a mátrixban, ellenben a negyedik lépésben a következő utakat találjuk: $1 \to 4 \to 2$, $1 \to 4 \to 2 \to 5$, $5 \to 4 \to 2$, $5 \to 4 \to 2 \to 1$, $5 \to 4 \to 3$.

$$W^{(1)} = \begin{bmatrix} i & h & i & i & h \\ i & i & i & i & i \\ h & h & i & h & h \\ h & i & i & i & h \\ h & h & h & i & i \end{bmatrix} \quad W^{(2)} = \begin{bmatrix} i & h & i & i & h \\ i & i & i & i & i \\ h & h & i & h & h \\ i & i & i & i & i \\ h & h & h & i & i \end{bmatrix} \quad W^{(4)} = W^{(5)} = \begin{bmatrix} i & i & i & i & i \\ i & i & i & i & i \\ h & h & i & h & h \\ i & i & i & i & i \\ i & i & i & i & i \end{bmatrix}$$

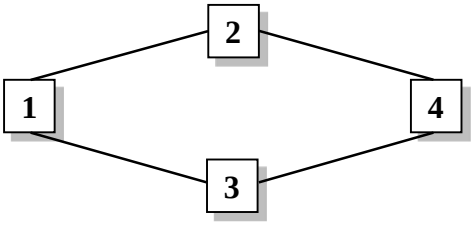
Az utolsó lépésben az 5-ös csúcson átmenő utakat vizsgáljuk, amelyek nem jelentenek további változást. Az eredményül kapott W mátrix alapján pedig elkészíthetjük a gráf tranzitív lezártját (ami már nem lesz síkba rajzolható).



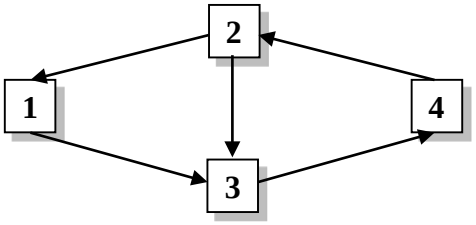
24.2. Feladat:

Alkalmazzuk a Warshall-algoritmust az alábbi gráfokra, és minden lépésben ábrázoljuk a változást a csúcsmátrixon:

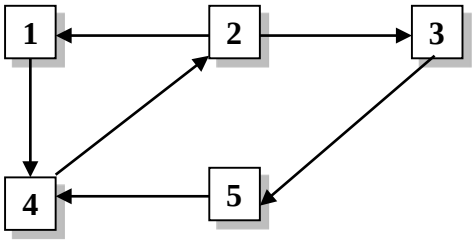
a)



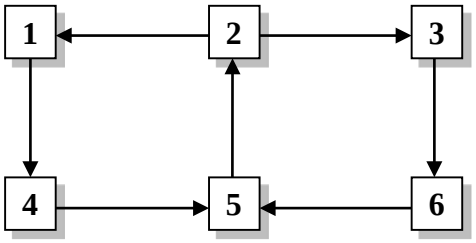
b)



c)



d)



24.3. Feladat:

Adjuk meg a Warshall-algoritmust az **ábrázolás szintjén** struktogram vagy pszeudokód segítségével

a) csúcsmátrixos ábrázolás esetén;

b) éllistás ábrázolás esetén.

24.4. Feladat:

Ha a Warshall-algoritmus mátrixában már minden értéket igazra váltottunk, akkor természetesen a további iterációkat nem kell lefuttatnunk, hiszen úgysem találnának újabb utat. Gyorsítsuk tehát az algoritmust azzal, hogy ha már minden értéket igazra váltott, akkor álljon le. Ezt az ötletet finomíthatjuk úgy, hogy ha egy adott sorban minden érték igazzá vált, akkor a további menetek során azt a sort kihagyjuk.

25. Minimális költségű feszítőfák

Piros-kék és Kruskal algoritmusok

Leírás:

Állítsuk elő egy irányítatlan, összefüggő gráfnak a *minimális költségű feszítőfáját*, vagyis azt a részgráfját, amely fa, minden csúcsát lefedi, és ezek közül a legkisebb az összköltsége (ha vannak azonos élköltségek, akkor ez nem feltétlenül egyértelmű).

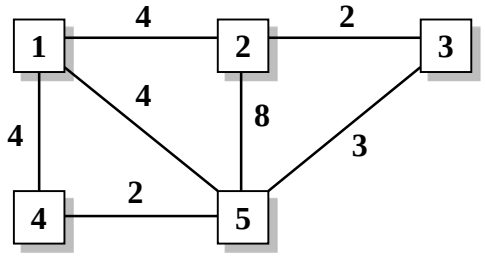
Az absztrakt *piros-kék eljárás*ban a célunk, hogy a feszítőfába kerülő éleket kékre (*kék szabály*), a nem fába kerülő éleket pedig pirosra színezzük (*piros szabály*) amíg van nem színezett él a gráfban. A kék szabálynál válasszunk ki egy olyan $X \subseteq V, X \neq \emptyset, X \neq V$ csúcshalmazt, amelyből még nem vezet ki kék él, és színezzük a legkisebb költségű kivezető élt kékre. A piros szabálynál válasszunk egy kört a gráfban, amely még nem tartalmaz piros élt, és színezzük a legnagyobb költségű élt a körben pirosra. Bizonyítható, hogy az eljárás során a színezés végig ún. *takaros* lesz – azaz nem színezünk olyan élt kékre, amely nem kerülhet be a minimális feszítőfába, és nem színezünk olyan élt pirosra, amely nem hagyható ki belőle –, továbbá nem akad el, amíg minden él színes nem lesz, és a végén egy minimális feszítőfát kapunk.

Az absztrakt piros-kék eljárás egyik megvalósítása a *Kruskal-algoritmus*, amely abból indul ki, hogy a gráf minden pontját egy kék fának veszi (ezeket eltárolhatjuk valamilyen halmaz adatszerkezet segítségével vagy a csúcsok számozásával), majd minden lépésben kiválasztja az egyik minimális költségű, még be nem színezett élt a gráfban. Ha az él két végpontja azonos fában van, akkor az élt pirosra színezi, különben kékre, és összevonja a két csúcshoz tartozó fát egy fába. Az algoritmus akkor ér véget, ha már csak egy kék fa található a gráfban, amely egy minimális költségű feszítőfa lesz.

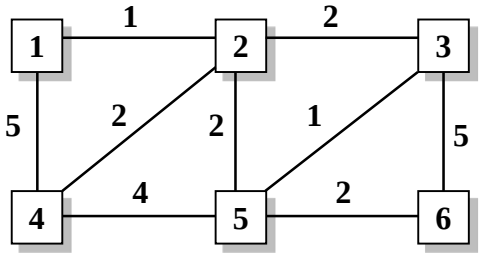
25.1. Feladat:

Határozzuk meg az alábbi gráfok minimális költségű feszítőfáját. Alkalmazzuk a piros, kék szabályokat felváltva, amíg lehet.

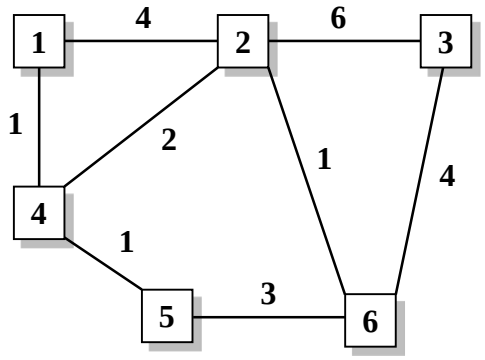
a)



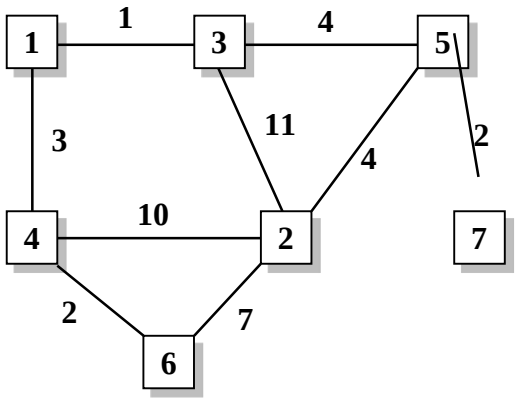
b)



c)



d)



25.2. Feladat:

Megoldható-e a minimális költségű feszítőfa keresése

a) csak **piros szabály** alkalmazásával;

b) csak **kék szabály** alkalmazásával?

25.3. Feladat:

Tegyük fel, hogy a piros-kék eljárással a G gráf egy minimális feszítőfáját már előállítottuk. Hogyan lehet módosítani ezt a fát, ha G-hez hozzáveszünk **egy új csúcsot** és az ehhez kapcsolódó éleket?

25.4. Feladat:

Határozzuk meg a 7.7.1. feladatban megadott gráfok egy minimális feszítőfáját a Kruskal-algoritmus segítségével. Hasonlítsuk össze az eredményt a piros-kék algoritmus eredményével. Ha vannak eltérések, azok milyen okra vezethetők vissza?

25.5. Feladat:

Valósítsuk meg a **Kruskal-algoritmust** csúcsmátrixos vagy éllistás ábrázolás esetén úgy, hogy a csúcsokat számozzuk aszerint, hogy melyik kék fába tartoznak, és a fák összevonásánál valamelyik fa csúcsaihoz tartozó értékeket átírjuk.

a) A minimális költségű színtelen él kiválasztásához használhatunk egy $(u, v) := MinArc(C)$ függvényt, amely visszaadja a megfelelő él u és v végpontajiat.

b) Az élek kiválasztását valósítsuk meg oly módon, hogy először az éleket a költségük szerint növekvő sorrendbe rendezve eltároljuk valamilyen adatszerkezetben, majd ebben a sorrendben dolgozzuk fel őket.

25.6. Feladat:

A Kruskal-algoritmus kritikus része a **fák ábrázolása**. Hogyan oldhatnánk meg ezt minél hatékonyabban?

Prim algoritmusa

25.7. Feladat:

Szemléltessük a Prim-algoritmus működését a 7.7.1. feladatban megadott gráfokon. Írjuk fel a $minQ$ prioritásos sor, valamint a d és Π tömbök tartalmát menetenként (minden $minQ$-ból történő kivitelnél).

25.8. Feladat:

Írjuk fel a Prim-algoritmust

a) csúcsmátrixos ábrázolás esetén, lineáris minimumkeresés mellett;

b) éllistás ábrázolás esetén, kupac felhasználásával.

25.9. Feladat:

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

25.10. Feladat:

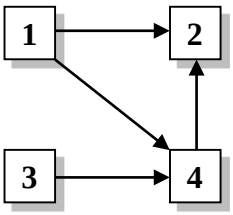
Tegyük fel, hogy a G irányítatlan, súlyozott gráf élsúlyai $[1..W]$ egészek. Hogyan lehetne gyorsítani a Prim-algoritmust?

26. Mélységi bejárás

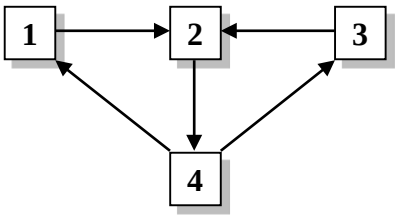
26.1. Feladat:

Szemléltessük a mélységi bejárást az alábbi gráfokon. Minden csúcs mellé rögzítsd a mélységi és befejezési számokat, az éleknek pedig adjuk meg a típusát (*faél*, *előreél*, *visszaél*, *keresztél*).

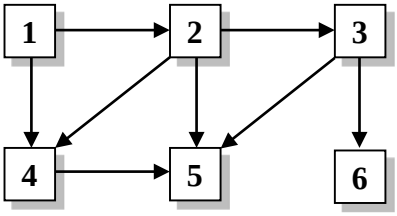
a)



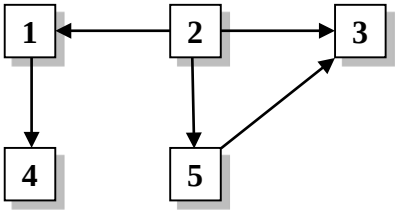
b)



c)



d)



26.2. Feladat:

Írjuk fel a mélységi bejárást a reprezentáció szintjén

a) csúcsmátrixos ábrázolás esetén;

b) éllistás ábrázolás esetén;

c) csúcsmátrixos ábrázolás esetén úgy, hogy az algoritmus kiírja az élek típusát is (ehhez használjuk az absztrakt $Write(u, v, type)$ függvényt).

26.3. Feladat:

a) Elegendő lenne-e a mélységi bejárás során csak **két színt** használni?

b) Irányított gráf esetén hogyan lehet a színezés alapján azonosítani az egyes **éltípusokat**?

c) **Irányítatlan gráfoknál** milyen **éltípusok** lehetnek, és a mélységi bejárás során hogyan azonosíthatjuk azokat?

d) Lehet-e olyan eset a mélységi bejárás során, hogy egy irányított gráf u csúcsa a keresés után egy olyan mélységi fába kerül, amely csak az u csúcsból áll, annak ellenére, hogy G-ben vannak u-ba bevezető és u-ból kivezető élek?

26.4. Feladat:

Adott egy G **irányítatlan gráf**. Adjunk algoritmust, amely G **komponenseit kiszínezi**. (Az azonos komponensbe eső csúcsokat azonos, a különböző komponensbe eső csúcsokat különböző színűre.)

26.5. Feladat:

Adott egy G **irányított gráf**. Adjunk algoritmust, amely G (gyengén) összefüggő **komponenseit kiszínezi**. (Az azonos komponensbe eső csúcsokat azonos, a különböző komponensbe eső csúcsokat különböző színűre.)

26.6. Feladat:

A G gráfon lefuttattuk a mélységi bejárást, de sajnos a gráfot és a Π tömböt kitöröltük, csak az $mszám$ ($beszám$) és a $bszám$ ($kiszám$) tömbjeink maradtak meg. Adjunk algoritmust, amely előállítja a mélységi feszítő erdőt, azaz a Π tömböt.

26.7. Feladat:

Adjunk algoritmust annak eldöntésére, hogy egy gráfban bármely $u, v \in V$ csúcsra legfeljebb egy egyszerű út vezet u-ból v-be

a) irányítatlan gráf esetén;

b) irányított gráf esetén.

26.8. Feladat:

Adjunk hatékony, az erőforrásgráf topologikus rendezésén alapuló eljárást holtpontok megelőzésére.

26.9. Feladat:

Adjunk $O(n + e)$ futási idejű algoritmust egy irányított gráf **komponensgráfjának** a meghatározására. Ügyeljünk arra, hogy legfeljebb egy él kösse össze az algoritmus által előállított komponensgráf csúcsai.

29. Huffman-kód

Leírás:

A *Huffman-kódolás* egy olyan eljárás, amelyben a bemenet szimbólumait a gyakoriságuk alapján kódoljuk különböző hosszúságú bináris sorozatokkal, így próbálva helyet megtakarítani. Például ha egy szöveg nagy része 'a' betűkből áll, akkor nyilván célszerű az 'a' betű kódjának egy nagyon rövid bináris számot venni, például az 1-et, mint egy hosszabb kódot, hiszen így rövidül le a legjobban a szöveg. Fontos azonban, hogy amikor az egyes szimbólumokat nem azonos hosszúságú bináris sorozatokkal kódoljuk, akkor valamilyen más módon kell biztosítanunk, hogy a szimbólumok határai felismerhetők legyenek. Erre egy kézenfekvő lehetőség, hogy ún. prefixmentes kódot (prefixkódot) állítunk elő, vagyis az egyes szimbólumok kódsorozatai között egyik sem valódi prefixe a másiknak.

A Huffman-kódolás 4 lépésből áll. Az első lépésben megszámoljuk, melyik szimbólumból mennyi található az adatsorozatban. A szimbólumokat és előfordulási gyakoriságukat (frekvenciájukat) egy minimum-kiválasztó prioritásos sorban tároljuk. Ezt felhasználva a második lépésben felépítjük az úgynevezett *Huffman-fát*, amelyből a harmadik lépésben minden szimbólumnak ki tudjuk olvasni a kódját, és létrehozzuk a kódtáblát. Az utolsó lépésben a kódtábla alapján szimbólumonként kódoljuk a bemenetet.

A Huffman-fa egy bináris fa, amelynek levélcsúcsai a szimbólumokat tartalmazzák, a gyökértől az egyes levelekbe vezető (egyértelmű) utak élei pedig a megfelelő szimbólum kódját írják le oly módon, hogy egy elemtől a bal gyereke felé haladva 0-t, a jobb gyereke felé lépve pedig 1-t olvasunk a kódhoz. A fát a következő eljárással építjük fel:

1. Kivesszük az első két elemet a prioritásos sorból.
2. Létrehozunk egy új szülőcsúcsot, amelyhez ez a két elemet hozzákapcsoljuk. Az új részfa frekvenciája a két gyereke frekvenciájának összege lesz.
3. Az új részfát hozzáadjuk a prioritásos sorhoz.
4. Ha már csak egy elem maradt a prioritásos sorban, akkor az az előállított Huffman-fa, különben pedig ugorjuk az 1. lépéshez.

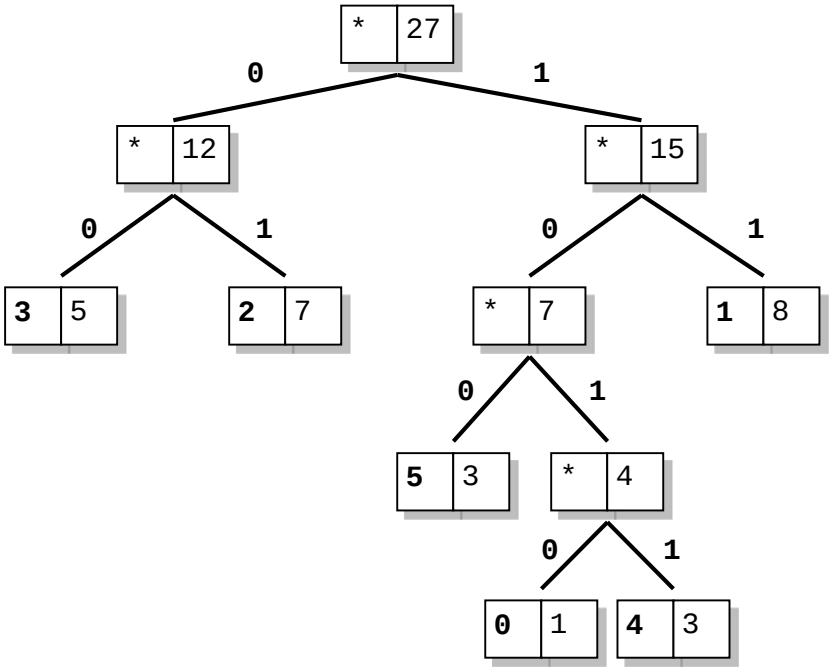
A Huffman-kódolásnak több továbbfejlesztett változata van, pl. *n*-áris fával dolgozás, kódhossz megszorításos, sablon frekvenciás. Az algoritmust például a JPEG képformátumnál, az MP3 hangformátumnál és a ZIP tömörítésnél használják.

Példa:

Vegyük a következő adatsorozatot: **`14231321013213215525143124 21`**. Tegyük fel, hogy eredetileg minden számjegyet 3 biten ábrázoltunk. Igaz, 3 biten már 8 különböző szimbólumot tudunk meghatározni, de 2 biten még csak 4-t, ezért mindenképpen legalább 3 bit kell. Ekkor az adatsor hossza $28 \cdot 3 = 84$ bit.

1. Határozzuk meg az adatsorozatban található értékeket $(0, 1, 2, 3, 4, 5)$ és frekvenciájukat $(1, 8, 7, 5, 3, 3)$. Az ezekből alkotott rendezett sorozat: (0,1), (4,3), (5,3), (3,5), (2,7), (1,8).
2. Építsük fel a Huffman-fát:
   1. A (0,1) és a (4,3) párokat kivesszük a prioritásos sorból, ezek szülője $(*, 4)$ lesz (jelölje a * azt, hogy ez egy belső csúcs), amit beteszünk a sorba: (5,3), $(*, 4)$, (3,5), (2,7), (1,8).

2. Az (5,3) és a (∗,4) párokat kivesszük, ezek szülője (∗,7) lesz, amit beteszünk a prioritásos sorba: (3,5), (2,7), (∗,7), (1,8).
3. A (3,5) és a (2,7) párokat kivesszük, ezek szülője (∗,12) lesz, amit beteszünk a prioritásos sorba: (∗,7), (1,8), (∗,12).
4. Az (∗,7) és (1,8) párokat kivesszük, ezek szülője (∗,15) lesz, amit beteszünk a prioritásos sorba: (∗,12), (∗,15).
5. A (∗,12) és (∗,15) párokat kivesszük, ezek szülője (∗,27) lesz, amit beteszünk a prioritásos sorba:(∗,27).
6. A sor már csak egy elemből áll, ez lesz a menet közben már felépített fa gyökere.



3. A fa alapján leírhatjuk a szimbólumokhoz tartozó kódokat, úgy, hogy a gyökérből végighaladunk az összes ágon, amíg levélhez nem érünk. Így a generált kódtáblánk:

```
0: 1010      1: 11        2: 01
3: 00        4: 1011      5: 100
```

Hasonlítsuk össze a kód eredeti hosszát a generált kód hosszával. A kódolt sorozat hosszát könnyen megkaphatjuk úgy, hogy a szimbólumok előfordulási számát megszorozzuk a kódjuk hosszával, és ezek összegét vesszük, tehát $4 \cdot 1 + 2 \cdot 8 + 2 \cdot 7 + 2 \cdot 5 + 4 \cdot 3 + 3 \cdot 3 = 65$ bit. Eredetileg 84 bit hosszú volt a sorozatunk, így 19 bitet tudtunk megtakarítani, a tömörítés során a sorozatot 22,6 %-kal redukáltuk. Fontos azonban, hogy a kódolt sorozat visszafejthetőségéhez tárolnunk kell a felépített kódtáblát is, így ilyen rövid inputok esetén a kódolt sorozat többnyire hosszabb lesz az eredetinél. Jóval nagyobb inputok esetén azonban a kódtábla mérete már sokkal kisebb lehet, mint a megspórolt bitek száma, így érdemes a módszert alkalmazni.

29.1. Feladat:

Adottak az alábbi sorozatok. Készítsük el a hozzájuk tartozó Huffman-fát, illetve kódtáblát. Számoljuk ki a keletkezett kód hosszát (kódtábla, valamint kód)

a) „abcdbaaaeebcbabcbabdecbabcbabc” (8 bit-es karakterek);

b) „algortimusok adatszerkezetek” (8 bit-es karakterek);

c) 345265326534516235534536651543333;

d) 439623446701233466866328764;

e) (0,1,0), (1,1,1), (1,1,1), (0,0,1), (1,1,1), (1,0,0), (0,0,1), (0,0,0), (1,1,0), (1,0,0), (0,1,1), (0,0,0);

f) (0,0,0), (1,1,0), (1,0,1), (1,1,0), (0,0,1), (1,0,1), (0,0,0), (0,1,1), (1,1,1), (0,0,1), (1,1,1), (1,1,0).

29.2. Feladat:

Valósítsuk meg a Huffman-kódolás egyes részalgoritmusait ADS vagy reprezentációs szinten, alkalmazhatjuk a korábban tanult adatszerkezeteket, illetve azok eljárásait. Számítsuk ki az algoritmusok műveletigényét.

a) Adatsor beolvasása, és az értékek számának meghatározása (ehhez használhatunk például két sort, illetve a sor $Exists(q, e)$ függvényét, amely lekérdezi, hogy a sorban megtalálható-e az elem).

b) Értékek alapján a Huffman-fa felépítése (implementáljuk olyan facsúcsokkal, amelynek elérhető a bal, illetve jobb gyereke).

c) Huffman-fa alapján a kódtábla létrehozása (a kódfát egy listában hozzuk létre).

d) Adatsor kódolása a kódtábla alapján.

29.3. Feladat:

Valósítsuk meg a következő módosításokat a Huffman-kódolás algoritmikus leírásában.

a) Ne készítsünk külön kódtáblát, hanem a kódokat közvetlenül a kódfa segítségével olvassuk ki, lineáris algoritmus segítségével (használjunk szülőre való hivatkozást). Számítsuk ki a módosítások műveletigényét.

b) Ne kétágú, hanem 4 ágú fát hozzunk létre. Vizsgáljuk meg, ez mennyiben befolyásolja a generált fa magasságát, a generált kód hosszát, illetve a keresés idejét.

30. LZW algoritmus

Leírás:

Az *LZW* (Lempel-Ziv-Welsh) *algoritmus* olyan tömörítési eljárás, ahol a kódtáblát a bemenő adatok feldolgozása közben építjük fel. A tábla kezdetben a szimbólumok kódjait tartalmazza – például az angol ábécé betűihez 5 hosszú bitsorozat vagy az ASCII karakterekhez 8 hosszú bitsorozat –, majd a következő algoritmus alapján építi tovább a kódtáblát:

1. Kezdésnek *s* legyen az adatsor első betűje
2. Beolvassa a következő szimbólumot az adatsorozatból, legyen ez *x*.
3. Ha kifogytunk az adatsorból – üres karaktert vagy vége jelet olvastunk be –, akkor írjuk be *s*-t a kódba.
4. Ha az *sx* (egymás után írva) már benne van a kódtáblában, akkor legyen $s = sx$, és menjünk az első lépésre.
5. Ha az *sx* még nincs benne a kódtáblában, akkor írjuk ki s kódját, vegyünk be *sx*-t a kódtáblába a következő kóddal. Legyen $s = x$, és menjünk az első lépéshez.

Az algoritmus előnye, hogy a kódtáblát nem kell mellékelni a kódhoz, hiszen azt a dekódolás során automatikusan vissza tudja generálni az algoritmus, feltéve, hogy a szimbólumok kódjai a dekódoló algoritmusnál megegyeznek a kódoló algoritmuséval. A dekódolás tehát szinte teljes mértékben megegyezik a kódolással, azzal a különbséggel, hogy nem a szimbólumokhoz írja ki a kódot, hanem a kódokhoz a szimbólumot.

Az LZW algoritmus a ma gyakorlatban használt adatkódolási eljárások legtöbbjének alapja, konkrétan az eljárás például a PDF dokumentumformátumnál, illetve a GIF és TIFF képformátumoknál használják.

Példa

Kódoljuk a következő sorozatot az LZW algoritmussal: **TOBEORNOTTOBEORTOBEORNOT**. Tegyük fel, hogy rendelkezésünkre áll az angol ábécé 26 karakterének kódja, amik 1 és 26 közötti számot rendelnek a betűkhöz (a = 1, z = 26). Ha az üzenetet tömörítés nélkül küldenénk, akkor egy szimbólumot 5 biten tudnánk kódolni, a szöveg 24 karakterből áll, tehát a tömörítetlen üzenet hossza 120 bit.

A tömörítési eljárást felírhatjuk egy tábla segítségével is, amelynek oszlopai:

- x: az aktuálisan kiolvasott karakter, jelölje # a szöveg végét,
- sx: a megmaradt karakter és az új karakter összeillesztése,
- K: új elem hozzávétele a kódtáblába,
- S: új elem hozzávétele a kódba,
- s: a művelet végrehajtás után megmaradt karakter.

Hajtsuk végre az algoritmust a szövegen.

x	sx	K	S	s
T				T
O	TO	28	20	O

← az alap kódtáblában 27-ig voltak sorszámozva az elemek, ezért az új elem sorszáma 28 lesz

B	OB	29	15	B
E	BE	30	2	E
O	EO	31	5	O
R	OR	32	15	R
N	RN	33	18	N
O	NO	34	14	O
T	OT	35	15	T
T	TT	36	20	T
O	TO			TO
B	TOB	37	28	B
E	BE			BE
O	BEO	38	30	O
R	OR			OR
T	ORT	39	32	T
O	TO			TO
B	TOB			TOB
E	TOBE	40	37	E
O	EO			EO
R	EOR	41	31	R
N	RN			RN
O	RNO	42	33	O
T	OT			OT
#	OT#	43	35	#

az algoritmus ezen a ponton a 32. kódot generálja a kódtáblába, ami már nem fér el 5 biten, ezért ettől a ponttól 6 biten kell tárolnia a kódot, ezért a korábbi kódokat is lecseréli 6 bitesre

Tehát a generált kódunk a következő: **20 15 2 5 15 18 14 15 20 28 30 32 37 31 33 35**. Ehhez hozzátartozik, hogy az első öt elem még 5 bit hosszan került be a kódba, míg az utána következők 6 biten. Tehát a kódunk binárisan ábrázolva: **10100 01111 00010 00101 01111 010010 001110 001111 010100 011100 011110 100000 100101 011111 100001 100011**.

Nézzük meg, mennyi helyet foglal el a tömörített szöveg. A kódtáblánkban vannak 5 hosszú, illetve 6 hosszú kódok is, és az algoritmus az első 5 helyen írt ki 5 hosszú kódot, míg a további 11 helyen 6 hosszút írt ki, tehát a kód hossza: $5 \cdot 5 + 11 \cdot 6 = 91$ bit lesz, míg az eredeti 120 bit volt, tehát a szöveg hosszát 24,1%-kal csökkentettük a tömörítés során.

30.1. Feladat:

Adottak az alábbi sorozatok. Tömörítsük a sorozatokat az LZW algoritmussal, adjuk meg a sorozatok kódját. Számítsuk ki a tömörítés méretét, majd alakítsuk vissza dekódolással az eredeti sorozatot.

a) „abbbaaacbcbabbbbb” (kezdeti kódtábla: 26 karakter);

b) „babababababababa” (kezdeti kódtábla: 26 karakter);

c) „aaaaaaaaaaaaaaaaaa” (kezdeti kódtábla: 26 karakter);

d) „algoritmusok adatszerkezetek” (kezdeti kódtábla: 256 ASCII karakter);

e) „lenni vagy nem lenni lenni vagy nem lenni” (kezdeti kódtábla: 256 ASCII karakter);

f) 111111111111111111 (kezdeti kódtábla: 2 számjegy);

g) 100110001111001010 (kezdeti kódtábla: 2 számjegy);

h) 345265326534516235534 (kezdeti kódtábla: 10 számjegy).

30.2. Feladat:

a) Készítsük el az LZW-kódoló algoritmus struktogramját.

b) Készítsük el az LZW-dekódoló algoritmus struktogramját.