4. Keresés, rendezés egyszerű struktúrában (tömb)

4.1. Keresés

4.1.1. Lineáris keresés

A tömb adatstruktúrában a keresés műveletét részint megtárgyaltuk a 3.1. fejezetben. Két esetet különböztettünk meg, a rendezetlen és a rendezett tömb esetét. Rendezetlen tömbben egy adott *k* kulcsú elem megkereséséhez nem áll rendelkezésre semmilyen információ azon kívül, hogy az elemek lineárisan követik egymást. Hiába érhetők el az elemek tetszőleges sorrendben, minden elemet meg kell vizsgálni, hogy a kulcs megegyezik-e a keresett *k* kulccsal, ugyanis azt sem tételeztük föl, hogy például a kulcsok számok. A kulcsok természete lehet olyan is, hogy mondjuk a rendezésükről szó nem lehet. (Nevezzünk meg ilyen kulcsokat!) Ez pedig azt jelenti, hogy a lineáris keresésnél jobb növekedési rendű időbonyolultsággal rendelkező algoritmus nem adható. A keresés rendezetlen tömbben lineáris idejű, azaz a keresési algoritmus időbonyolultsága $T(n)=\Theta(n)$. Ez egy aszimptotikus $T(n)\approx c\cdot n$ összefüggést jelent (pontosítva: $\lim_{n\to\infty}\frac{T(n)}{n}=c$), melyben c egy pozitív konstans. Nem mindegy azonban ennek a konstansnak a konkrét értéke. Azt megtehetjük, hogy a lineáris keresési algoritmust némiképpen módosítva ezt a konstanst lejjebb szorítjuk. Tekintsük például a 3.1.1. KERESÉS_TÖMBBEN algoritmust. Legyen az algoritmus i számmal számozott sorának a végrehajtási ideje c_i. Tételezzük fel a számolási idő szempontjából a legrosszabb esetet, hogy a keresett elem nincs a tömbben. Ekkor a keresés ideje:

$$\begin{aligned}T(n)&=c_1+c_2+c_3+c_4+c_5+c_6+c_7+c_8+n\cdot(c_9+c_{10})+c_{11}+c_{12}=\\&=c_7+c_8+n\cdot(c_9+c_{10})+c_{11}+c_{12}\end{aligned}\quad. \tag{1}$$

Nem túl sokat torzítunk a valóságon, ha feltételezzük, hogy az értékadás és egy relációvizsgálat valamint logikai művelet (ÉS) körülbelül azonosan $\bar{c}$ ideig tart. Akkor $c_7=c_8=c_{10}=c_{11}=c_{12}=\bar{c}$, $c_9=3\bar{c}$ és így

$$T(n)=\bar{c}+\bar{c}+n\cdot(3\bar{c}+\bar{c})+\bar{c}+\bar{c}=4\bar{c}n+4\bar{c}\ . \tag{2}$$

A $T(n)=\Theta(n)$-nek megfelelő aszimptotikus kifejezésben szereplő c konstans értéke $4\bar{c}$ -nek vehető. Módosítsuk most úgy a 3.1.1. algoritmust, hogy a keresés kezdetén a keresett *k* kulcsot a tömb végéhez hozzáfüggesztjük. Feltesszük, hogy erre van elegendő hely. Ebben az esetben az elem biztosan benne lesz a tömbben. A keresésből a tömbelem indexének végvizsgálata kihagyható. A keresés mindig sikeres lesz, csak ha a visszakapott index nagyobb, mint az eredeti tömb utolsó elemének indexe, akkor valójában az elem nincs a tömbben. Íme a megváltoztatott algoritmus pszeudokódja:

	4.1.1.1. algoritmus **Módosított keresés tömbben** // $T(n)=\Theta(n)$
1	KERESÉS_TÖMBBEN (*A*,*k*, *x*)
2	// Input paraméter: *A* - a tömb
3	// *k* – a keresett kulcs
4	// Output paraméter: *x* - a k kulcsú elem pointere (indexe), ha van ilyen elem vagy NIL, ha nincs
5	// Lineárisan keresi a *k* kulcsot.
6	//
7	*x* ← ***fej****[A]*
8	INC(***vége****[A]*)
9	***kulcs****[A_{vége[A]}]* ← *k*
10	**WHILE** ***kulcs****[A_x]* ≠ *k* **DO**
	INC(*x*)
	DEC(***vége****[A]*)
	IF *x*> ***vége****[A]*
14	**THEN** *x* ← **NIL**
15	**RETURN** (*x*)

Most a legrosszabb eset ideje

$$\begin{aligned}T(n) &= c_1+c_2+c_3+c_4+c_5+c_6+c_7+c_8+c_9+n\cdot(c_{10}+c_{11})+c_{12}+c_{13}+c_{14}= \\ &= c_7+c_8+c_9+n\cdot(c_{10}+c_{11})+c_{12}+c_{13}+c_{14}\end{aligned}. \quad (3)$$

Itt azt feltételezhetjük, hogy $c_7=c_8=c_9=c_{10}=c_{11}=c_{12}=c_{13}=c_{14}=\bar{c}$, amiből

$$T(n)=\bar{c}+\bar{c}+\bar{c}+n\cdot(\bar{c}+\bar{c})+\bar{c}+\bar{c}+\bar{c}=2\bar{c}n+6\bar{c}\,. \quad (4)$$

Itt az aszimptotikus kifejezés konstansa $2\bar{c}$, tehát ezzel a kis trükkel ha a futási idő jellegét (linearitását) nem is, de a konkrét idejét közel felére sikerült csökkenteni a 3.1.1. algoritmus idejéhez képest.

4.1.2. Logaritmikus keresés

Rendezett tömb esetén láttuk a 3.1. fejezetben, hogy a lineáris időnél jobbat is el tudunk érni a bináris kereséssel (3.1.4. algoritmus), amely logaritmikus időt ad. A bináris keresés mellett vele konkuráló érdekes algoritmus a *Fibonacci keresés* algoritmusa. Feltételezzük, hogy a kulcsokat (és az adatrekordokat) tartalmazó tömb neve *A*, mérete *n*, és a tömbelemek indexelése egytől indul. A rekordok a kulcsok növekvő sorrendje szerint követik egymást, a kulcsok pedig mind különbözőek. Alább megadjuk szövegesen a Fibonacci keresés algoritmusát. A felírást azon feltétel mellett tesszük meg, hogy $n+1$ legyen egyenlő az F_{k+1} Fibonacci számmal. (Az algoritmus módosítható tetszőleges pozitív egész *n* esetére is.)

A Fibonacci keresés algoritmusa:

1. Kezdeti beállítások: $i \leftarrow F_k$, $p \leftarrow F_{k-1}$, $q \leftarrow F_{k-2}$
2. Összehasonlítás: Ha $k<$***kulcs***$[A_i]$, akkor a 3. pont következik
   Ha $k>$***kulcs***$[A_i]$, akkor a 4. pont következik
   Ha $k=$***kulcs***$[A_i]$, akkor sikeres befejezés.
3. Az *i* csökkentése: Ha $q=0$, akkor sikertelen befejezés.
   Ha $q\neq 0$, akkor $i \leftarrow i-q$, $\begin{pmatrix} p \\ q \end{pmatrix} \leftarrow \begin{pmatrix} q \\ p-q \end{pmatrix}$ és a 2.pont következik.
4. Az *i* növelése: Ha $p=1$, akkor sikertelen befejezés.
   Ha $p\neq 1$, akkor $i \leftarrow i+q$, $p \leftarrow p-q$, $q \leftarrow q-p$ és a 2. pont következik.

Az eddigi keresési algoritmusok csak a rendezettség tényét használták ki, lényegtelen volt a kulcsok milyensége. Ha föltételezzük, hogy a kulcsok számok, akkor használhatjuk az úgynevezett *interpolációs keresést*. A módszer hallgatólagosan feltételezi, hogy a kulcsok növekedésükben körülbelül egyenletes eloszlásúak (majdnem számtani sorozatot alkotnak). Az átlagos keresési idő: $T(n)=\Theta(\log\log n)$. Az elv azon alapszik, hogy a feltételezéseink mellett a keresett k kulcs a sorban az értékének megfelelő arányosság szerinti távolságra van a keresési intervallum balvégétől. azaz ha a balvég indexe b, a jobbvégé j, a megfelelő kulcsok k_b és k_j, akkor a következő vizsgálandó elem indexe $b+\frac{(j-b)\cdot(k-k_b)}{k_j-k_b}$. Ha a keresett kulcs megegyezik ezen elem kulcsával, akkor az algoritmus sikeresen befejeződik. Ha a k kulcs értéke kisebb, akkor az intervallum jobbvégét, ha a k kulcs nagyobb, akkor a balvégét cseréljük le erre a közbülső elemre és az új intervallummal folytatjuk a keresést.

4.1.3. Hasító táblák

A hasító táblák algoritmusai tömböt használnak a kulcsok (rekordok) tárolására, de nem az eddig megszokott értelemben, vagyis a tömböt általában nem töltik fel teljesen és a rekordok nem feltétlenül hézagmentesen helyezkednek el a tömbben. Az algoritmusok a keresésre, módosításra, beszúrásra és a törlésre vannak kihegyezve, tehát ezek a műveletek végezhetők el a struktúrán hatékonyan. Például a legkisebb kulcs megkeresése a struktúrában már nem olyan hatékony, mint a fent nevezettek. Az alapvető problémát az okozza, és ez az oka ezen adatstruktúra bevezetésének, hogy a kulcsok elméletileg lehetséges U halmaza - az úgynevezett *kulcsuniverzum* - számottevően bővebb, mint a konkrétan szóbajöhető kulcsok halmaza, amelyet ráadásul még csak nem is ismerünk pontosan. Egy példával világítjuk ezt meg. Legyen adott egy cég, amelyről ismert, hogy legfeljebb 5000 alkalmazottja van. Minden alkalmazottról bizonyos adatokat nyilván kell tartani a különböző adminisztrációs feladatok elvégzéséhez. Ezen adatok egyike a TAJ-szám (Társadalombiztosítási Azonosító Jel), amely kilencjegyű, előjel nélküli egész szám. Ezt az adatot néztük ki magunknak kulcs céljára, mivel a TAJ-szám egyértelműen azonosítja a személyt. Ha csak ennyit tudunk a TAJ számról – és most nem is akarunk annak mélyebb ismereteiben elmélyedni, - akkor ez 10^9 lehetséges kulcsot jelent. Ennyi eleme van a kulcsuniverzumnak. Ebből az írdatlan mennyiségű kulcsból nekünk viszont csak körülbelül 5000 kell. Azaz a kulcsuniverzumnak csak egy viszonylag szűk részhalmaza, (a teljes halmaz körülbelül 0,0005%-a). Azt viszont nem tudjuk, hogy melyik részhalmaz. A kulcsokat majd a munkatársak hozzák magukkal. Ráadásul a személyi mozgás, fluktuáció révén ezek a kulcsok változhatnak is. Teljességgel nyílvánvaló, hogy értelmetlen lenne az

adatbázisunkban egy milliárd rekord számára helyet biztosítani. Elég, ha egy kis ráhagyással mondjuk körülbelül 6000 rekordnak foglalunk le helyet (20% körüli ráhagyás). Ezen a helyen kell az 5000 rekordot úgy elhelyezni, hogy a rekordok keresése, módosítása, beszúrása, törlése hatékony legyen. Azt a táblázatot (tömböt), ahol a rekordokat, vagy a rekordokra mutató mutatókat (pointereket) elhelyezzük, *hasító táblázat*nak, hasító táblának nevezzük az angol *hash table* elnevezés után. A hasító tábla elemeinek indexelése nulláról indul. A tábla elemeit *rés*nek is szokás nevezni. Külön érdemes kihangsúlyozni a módosítás műveletét, amely tulajdonképpen két részből áll, egy keresésből, majd a megtalált rekord módosításából. Ha ez a módosítás a rekord kulcsmezejét érinti, akkor a rekordot a táblából először törölni kell, majd a módosítás elvégzése után újra be kell szúrni az új kulcsnak megfelelően.

*Közvetlen címzésű táblázat*ról beszélünk, ha a kulcsuniverzum az $U=\{0,1,...,M-1\}$ számok halmaza, ahol az M egy mérsékelt nagyságú szám. A tárolási célra használandó tábla (tömb) mérete legyen m, amit most válasszunk $M=m$-nek. Ekkor a kulcs egyúttal az index szerepét játszhatja, azaz a kulcsuniverzum minden kulcsa egyidejűleg tárolható. Ha valamely kulcsot nem tároljuk, akkor a helye, a rés üres lesz. Az üres rést az jelenti, hogy a rés tartalma NIL. (Pointeres változat.) A keresés, beszúrás, törlés algoritmusai ekkor rém egyszerűek, pszeudokódjaik következnek alább. Mindegyik művelet időigénye konstans, $T(n)=\Theta(1)$. A tömb neve T, utalásképpen a táblázatra.

	4.1.3.1. algoritmus **Közvetlen címzésű keresés hasító táblában**
	// $T(n)=\Theta(1)$
1	KÖZVETLEN_CÍMZÉSŰ_KERESÉS (*T, k, x*)
2	// Input paraméterek: *T* – a tömb zérus kezdőindexszel
3	// ……………………*k* – a keresett kulcs
4	// Output paraméterek: *x* – a keresett elem indexe, NIL, ha nincs
5	//
6	$x \leftarrow T_k$
7	**RETURN** (*x*)

	4.1.3.2. algoritmus **Közvetlen címzésű beszúrás hasító táblába**
	// $T(n)=\Theta(1)$
1	KÖZVETLEN_CÍMZÉSŰ_BESZÚRÁS (*T, x*)
2	// Input paraméterek: *T* – a tömb zérus kezdőindexszel
3	// *x* – mutató a beszúrandó elemre
4	//
5	$T_{kulcs[x]} \leftarrow x$
6	**RETURN**

	4.1.3.2. algoritmus **Közvetlen címzésű törlés hasító táblába**
	// $T(n)=\Theta(1)$
1	KÖZVETLEN_CÍMZÉSŰ_TÖRLÉS (*T, x*)
2	// Input paraméterek: *T* – a tömb zérus kezdőindexszel
2	// *x* – mutató a törlendő elemre

3	//
4	$T_{kulcs[x]}$ ←**NIL**
5	**RETURN**

Az ismertetett eset nagyon szerencsés és nagyon ritka. Általában M értéke lényegesen nagyobb, mint a ténylegesen tárolható kulcsok m száma. A memóriaigény leszorítható Θ(m)-re úgy, hog az átlagos időigény $\Theta(1)$ maradjon a *láncolt hasító tábla* alkalmazásával. Ebben a táblában minden elem egy listafej mutatója, amely kezdetben az üres táblázat esetén mindenütt **NIL**. Most nem tételezzük fel, hogy az U kulcsuniverzum a 0,1,...,m-1 számok halmaza lenne, de feltételezzük, hogy ismerünk egy úgynevezett *hasító függvényt*, amely az U kulcsuniverzum elemeit képezi bele ebbe a 0,1,...,m-1 számhalmazba, az indexek halmazába: h: $U \rightarrow \{0,1,\ldots,m-1\}$. Ez a függvény egyáltalán nem lesz injektív, azaz nem fog feltétlenül különböző kulcsokhoz különböző számértéket rendelni, hiszen az U elemszáma sokkal több, mint a 0,1,...,m-1 indexhalmazé. (Ezt a viszonyt az $M >> m$ jelöléssel szoktuk jelezni.) A célunk a hasító függvénnyel az, hogy a k kulcsú rekord a tábla $h(k)$ indexű réséből indított láncolt listába kerüljön. Ezzel a stratégiával oldjuk fel az úgynevezett *ütközési problémát*, ami akkor lép fel, ha két különböző kulcs ugyanarra az indexre (résre) képeződik le. (Az ütközésnek nem kicsi az esélye. Ha egy tízemeletes ház földszintjén négyen belépnek a liftbe és mindenki a többitől függetlenül választ magának egy emeletet a tíz közül, akkor 10·10·10·10=10000-féleképpen választhatnak. Ebből a 10000-ből csak 10·9·8·7=5040 olyan van, amikor mindenki a többitől eltérő emeletet választott. Ha továbbá minden ilyen választást azonos esélyűnek tekintünk, akkor annak esélye, hogy legalább két ember ugyanazt az emeletet választotta eszerint $\frac{1000-5040}{10000}=0,496$. Tehát majdnem 50% eséllyel lesznek olyanok, akik ugyanarra az emeletre mennek. A híres von Mises féle születésnap probléma esetén elegendő legalább 23 embernek összejönni, hogy legalább 50% eséllyel legyen köztük legalább kettő olyan, akik azonos napon ünneplik a születésnapjukat.) Egy elemnek a listában történő elhelyezése történhet a lista elejére történő beszúrással, vagy készíthetünk rendezett listát is, ha a kulcsok rendezhetők. Az egyes műveletek pszeudokódjai alább következnek. Az egyes műveletek idejeivel kapcsolatban bevezetünk egy fogalmat, az úgynevezett *telítettségi arányt*, vagy *telítettségi együthatót*.

Definíció: A telítettségi arány

Az $\alpha=\frac{n}{m}$ számot a hasító tábla telítettségi arányának nevezzük, ahol m a tábla réseinek a száma, n pedig a táblába beszúrt kulcsok száma.

A telítettségi arány láncolt hasító tábla esetén nemnegatív szám, amely lehet 1-nél nagyobb is. Szokásos elnevezése még a *kitöltési arány* is.

4.1.3.4. algoritmus **Láncolt hasító keresés**	
//	$T(n)=\Theta(1+\alpha)$
1	LÁNCOLT_HASÍTÓ_KERESÉS (T, k, x)
2	// Input paraméterek: T – a tömb zérus kezdőindexszel
3	// k a keresett kulcs
4	// Output paraméterek: x - a k kulcsú rekord mutatója, **NIL** ha a rekord nincs a

	Struktúrában
5	A k kulcsú elem keresése a $T_{h(k)}$ listában, melynek mutatója x lesz.
6	**RETURN** (x)

	4.1.3.5. algoritmus **Láncolt hasító beszúrás**
	// $T(n)=\Theta(1+\alpha)$
1	LÁNCOLT_HASÍTÓ__BESZÚRÁS (T, x)
2	// Input paraméterek: T – a tömb zérus kezdőindexszel
3	// x – mutató a beszúrandó elemre
4	
5	Beszúrás a $T_{h(\mathbf{kulcs}[x])}$ lista elejére
6	**RETURN**

	4.1.3.6. algoritmus **Láncolt hasító törlés**
	// $T(n)=\Theta(1+\alpha)$
1	LÁNCOLT_HASÍTÓ_TÖRLÉS (T, x)
2	// Input paraméterek: T – a tömb zérus kezdőindexszel
3	// x – mutató a törlendő elemre
4	//
5	x törlése a $T_{h(\mathbf{kulcs}[x])}$ listából
6	**RETURN**

Vezessünk most be két jelölést. A megvizsgált kulcsok átlagos számát jelölje C_n a sikeres keresés esetén és $C_n^{'}$ a sikertelen keresések esetén.

Tétel: <u>A láncolt hasító tábla időigénye</u>
Ha α a kitöltési arány, akkor a láncolt hasító táblában

$$C_n = \Theta(1+\alpha) \text{ és } C_n^{'} = \Theta(1+\alpha). \qquad (1)$$

.

A láncolt hasító tábla mérete nem korlátozza a struktúrában elhelyezett rekordok számát. Természetesen ha a rekordok száma igen nagy, akkor az egyes résekhez tartozó listák mérete is igen nagy lehet. Nem ritkán azonban ismeretes egy felső korlát a rekordok számára és azok (vagy a kulcsaik, vagy a mutató a rekordra) elhelyezhetők magában a táblázatban. Minden táblabeli elem (rés) legalább két mezőből fog állni az alábbi tárgyalásmódban, egy kulcsmezőből és egy mutatóból, amely a következő elemre mutat. Minden réshez tartozik egy foglaltsági bit, amely szerint a rés lehet szabad, vagy lehet foglalt. Közöljük két algoritmus pszeudokódját. Az első a megadott kulcsú elemet keresi a táblában. Ha megtalálta, akkor visszaadja az elem indexét, ha nem találta meg, akkor **NIL**-t ad vissza. A második a megadott kulcsú elemet beszúrja a táblába, ha az elem nincs a táblában és van még ott üres hely. Ha az elem benne lenne a táblában, akkor az algoritmus visszatér. Az algoritmus jellegzetessége, hogy a különböző résekhez tartozó listák egymásba nőnek. Az üres helyek adminisztrálása céljából bevezetünk egy r változót, amely mindig azt fogja mutatni, hogy az r és a magasabb indexű helyeken a táblaelemek már foglaltak. Az *r* a tábla attributuma lesz. Üres táblára *r*=*m*, minden rés szabad és a ***köv*** mutatók mindegyike **NIL**.

4.1.3.7. algoritmus **Összenövő listás hasító keresés**	
//	$T(n)=\Theta(1+\alpha)$
1	ÖSSZENÖVŐ_LISTÁS_HASÍTÓ_KERESÉS (*T, k, x*)
2	// Input paraméterek: *T* – a tömb zérus kezdőindexszel
3	// *k* - a keresett kulcs
4	// Output paraméterek: *x* - a k kulcsú rekord mutatója, **NIL** ha a rekord nincs a Struktúrában
4	//
5	$i \leftarrow h(k)$
6	**IF** T_i foglalt
7	**THEN** **REPEAT**
8	**IF** *k*= ***kulcs****[T_i]*
9	**THEN** $x \leftarrow i$
10	**RETURN** (*x*)
11	**IF** ***kulcs****[T_i]* ≠ **NIL**
12	**THEN** *i* ← ***köv****[T_i]*
13	**UNTIL** ***köv****[T_i]*=**NIL**
14	*x* ← **NIL**
15	**RETURN** (x)

4.1.3.8. algoritmus **Összenövő listás hasító beszúrás**	
//	$T(n)=\Theta(1+\alpha)$
1	ÖSSZENÖVŐ_LISTÁS_HASÍTÓ_BESZÚRÁS(*T, k, hibajelzés*)
2	// Input paraméterek: *T* – a tömb zérus kezdőindexszel
3	// *k* - a beszúrandó kulcs
4	// Output paraméterek: *hibajelzés* – a művelet eredményességét jelzi
4	//
5	$i \leftarrow h(k)$
6	**IF** T_i szabad
7	**THEN** ***kulcs****[T_i]* ← *k*
8	***köv****[T_i]* ← **NIL**
9	*hibajelzés* ←„Sikeres beszúrás”
10	**RETURN** (*hibajelzés*)
11	**ELSE** **REPEAT**
12	**IF** *k*= ***kulcs****[T_i]*
13	**THEN** *hibajelzés*←„Sikeres beszúrás”
14	**RETURN** (*hibajelzés*)
15	**IF** ***kulcs****[T_i]* ≠ **NIL**
16	**THEN** i ← ***köv****[T_i]*
17	**UNTIL** ***köv****[T_i]*=**NIL**
18	// Nincs a táblában, be kell szúrni
19	**IF** $R \le 0$

20	**THEN** *hibajelzés* ←„Betelt a tábla”
21	**RETURN** (*hibajelzés*)
22	**REPEAT**
23	DEC (*r*)
24	**IF** T_r szabad
25	**THEN** ***kulcs***$[T_r] \leftarrow k$
26	***köv***$[T_r] \leftarrow$ **NIL**
27	***köv***$[T_i] \leftarrow r$
28	*hibajelzés* ← „Sikeres beszúrás”
29	**RETURN** (*hibajelzés*)
30	**UNTIL** $r \le 0$
31	*hibajelzés* ← „Betelt a tábla”
32	**RETURN** (*hibajelzés*)

A törlés műveletét itt nem tárgyaljuk, hanem külön diszkusszió tárgyává tesszük a feladatok között. A keresés és beszúrás műveletének átlagos idejére érvényesek az alábbi közelítő formulák:

$$C_n \approx 1+\frac{1}{8\alpha}\left(e^{2\alpha}-1-2\alpha\right)+\frac{1}{4}\alpha\,. \qquad (2)$$

$$C_n' \approx 1+\frac{1}{4}\left(e^{2\alpha}-1-2\alpha\right). \qquad (3)$$

Eddig nem szóltunk a hasító függvényről közelebbit. Egy jó hasító függvény kielégíti az <u>*egyszerű egyenletességi feltételt*</u>, ami azt jelenti, hogy minden kulcs egyforma eséllyel képződik le az *m* rés bármelyikére, amely az ütközések elleni harcban fontos. Ezen felül lényeges, hogy a függvény értéke nagyon gyorsan számítható legyen. Az igazán nem komoly probléma, hogy a kulcsok sokfélék lehetnek, hiszen általában könnyen konvertálhatók számértékké. Például ha a kulcs szöveges, akkor tekinthetjük a szöveg egyes betűinek az ASCII kódját és minden ilyen számértéket egy magasabb alapú számrendszer számjegyeinek vesszük. Ha a szöveg csak (latin) nagybetűket tartalmaz, akkor minden betűhöz hozzárendelhetjük az ábécében elfoglalt helyének az eggyel csökkentett sorszámát. A – 0, B – 1, C – 2, D – 3, E – 4, ..., Z – 25. Ekkor a „ZABA” szöveghez hozzárendelhető szám 26-os számrendszerben $25\cdot 26^3 + 0\cdot 26^2 + 1\cdot 26 + 0 = 439426$. Két nagy módszer osztályt szokás kiemelni a hasító függvények kiválasztásakor, az osztó módszerű és a szorzó módszerű függvényeket.

Az osztó módszer esetében a $h(k) = k \bmod m$ formulával dolgozunk. Nem árt azonban némi óvatosság az *m* kiválasztásánál. Ha *m* páros, akkor *h(k)* paritása is olyan lesz, mint a *k* kulcsé, ami nem szerencsés. Ha *m* a 2-nek hatványa, akkor *h(k)* a k kulcs utolsó bitjeit adja. Általában prímszámot célszerű választani. Knuth javaslata alapján kerülendő az az *m*, amely osztja az $r^k \pm a$ számot, ahol *k* és *a* kicsi számok, *r* pedig a karakterkészlet elemeinek a száma. Például ha r=256 és a kulcsok az ASCII táblázatbeli karakterek lehetnek és az $m=2^{16}+1=65537$ Fermat-féle prímszámot választjuk, akkor mondjuk háromkarakteres kulcsok esetén a $C_1C_2C_3$ kulcsot tekinthetjük egy 256-os számrendszerbeli háromjegyű számnak is. Ha most itt *m*-mel osztunk, ($m=256^2+1$), akkor az osztás eredménye $(C_2C_3\text{-}C_1)_{256}$ lesz (ellenőrizzük!), ami azt jelenti, hogy az eredmény úgy adódik, hogy a szám első jegyét levonjuk a hátsó két jegy által alkotott számból. Ezáltal *egymáshoz közeli kulcsok egymáshoz közelre* képződnek le.

A szorzásos módszer esetében a $h(k) = \lfloor m \cdot (kA \bmod 1) \rfloor$ formulát használjuk, ahol A egy alkalmas módon megválasztott konstans, 0<A<1. Igen jó tulajdonságokkal rendelkezik az $A = \Phi^{-1} = \frac{\sqrt{5}-1}{2} \approx 0{,}618033988...$ számérték, amelyet a Fibonacci számokkal kapcsolatban már megismerhettünk.

Nyílt címzések

A nyílt címzésű hasító táblákban nincsenek táblázaton kívül tárolt elemek, listák. A táblaelemeket (a rekordokat) a 0,1,...,*m*-1 indexekkel indexeljük. Az ütközések feloldására azt a módszert használjuk, hogy beszúráskor amennyiben egy rés foglalt, akkor valamilyen szisztéma szerint tovább lépünk a többi résre, míg üreset nem találunk és oda történik a beszúrás. Keresésnél szintén ha a számított résben nem a keresett kulcs van, akkor a beszúrási szisztéma szerint keressük tovább. Formálisan ezt azáltal érjük el, hogy a hasító függvényünket, amely eddig csak a kulcstól függött, most kétváltozósra terjesztjük ki, a második változója a próbálkozásunk sorszáma lesz. Ez a szám a 0,1,2,...,*m*-1 számok valamelyike lehet. Azaz a függvényünk: $h: U \times (0,1,\ldots,m-1) \to (0,1,\ldots,m-1)$ és egy rögzített k kulcs esetén a $h(k,0), h(k,1), \ldots, h(k,m-1)$ egy úgynevezett *kipróbálási sorozat*ot produkál. Ezek az indexek a 0,1,...,*m*-1 indexhalmaznak egy permutációját kell, hogy adják. Ezzel biztosítjuk, hogy ha van még hely a táblában, akkor a beszúrást minden esetben meg lehessen csinálni. Tekintsük ezután a keresés, beszúrás és törlés pszeudokódjait a nyílt címzésű hasítótábla esetén

	4.1.3.4. algoritmus **Nyílt címzésű hasító keresés**
	// $T(n) = \Theta(1+\alpha)$
1	NYÍLT_CÍMZÉSŰ_HASÍTÓ_KERESÉS (*T, k, x*)
2	// Input paraméterek: *T* – a tömb zérus kezdőindexszel
3	// *k* - a keresendő kulcs
4	// Output paraméterek: *x* – a k kulcsú rekord mutatója, **NIL**, ha nincs
5	//
6	$i \leftarrow 0$
7	**REPEAT**
8	$j \leftarrow h(k,i)$
9	**IF** ***kulcs***$[T_j] = k$ és ***foglaltság***$[T_j]$=Foglalt
10	**THEN** $x \leftarrow j$
11	**RETURN** (*x*)
12	**INC** (*i*)
13	**UNTIL** T_j=**NIL** vagy $i = m$
14	*x*←**NIL**
15	**RETURN** (*x*)

	4.1.3.5. algoritmus
	Nyílt címzésű hasító beszúrás
	// $T(n) = \Theta(1+\alpha)$
1	NYÍLT_CÍMZÉSŰ _HASÍTÓ__BESZÚRÁS (*T, k, hibajelzés*)
2	//

3	// Input paraméterek: T – a tömb zérus kezdőindexszel
4	// k - a beszúrandó kulcs
5	// Output paraméterek: *hibajelzés* – a művelet eredményességét jelzi
6	//
7	$i \leftarrow 0$
8	**REPEAT**
9	$j \leftarrow h(k,i)$
10	**IF** ***foglaltság***$[T_j]$=szabad vagy ***foglaltság***$[T_j]$=törölt
11	**THEN** ***kulcs***$[T_j] \leftarrow k$
12	*hibajelzés*←„ Sikeres beszúrás”
13	**RETURN** (*hibajelzés*)
14	**ELSE** **INC** (*i*)
15	**UNTIL** $i = m$
16	*Hibajelzés*←„tábla betelt”
17	**RETURN** (*hibajelzés*)

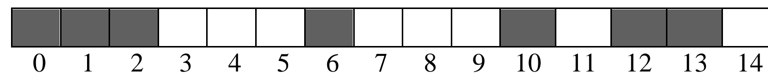
	4.1.3.6. algoritmus	
	Nyílt címzésű hasító törlés	
	// $T(n) = \Theta(1+\alpha)$	
1	NYÍLT_CÍMZÉSŰ_HASÍTÓ_TÖRLÉS (*T, k*)	
2	// Input paraméterek: T – a tömb zérus kezdőindexszel	
3	// k - a törlendő kulcs	
4	//	
5	NYÍLT_CÍMZÉSŰ_HASÍTÓ_KERESÉS (*T, k, x*)	
6	**IF**	$x \neq$ **NIL**
7		**THEN** ***foglaltság***$[T_j] \leftarrow$törölt
8	**RETURN**	

Törlésnél nem megfelelő a **NIL** beírás, helyette a rés foglaltságát TÖRÖLT-re kell állítani, mivel a **NIL** a későbbi kereséseket megzavarhatja. A kipróbálási sorozat végét jelzi ott, ahol annak valójában még nincs vége. Ennek az a következménye, hogy sok beszúrás és törlés után már szinte minden rés vagy foglalt, vagy törölt lesz, ami a keresés sebességét lerontja. Ilyenkor a teljes táblát a benne lévő kulcsokkal újra hasítjuk. A másik lehetőség, hogy a láncolt listás megoldást választjuk, ahol a törlések nem okoznak gondot. Most három gyakran alkalmazott módszer típust említünk meg a nyílt címzésű hasításra.

Lineáris kipróbálás

Ez a módszer a $h(k,i) = ((h_0(k)) + i) \bmod m$ hasító függvényt használja, ahol az i=0,1,2, ..., *m*-1 lehet. A formula alapján látható, hogy egy h_0 alap hasító függvényből indul ki és a kipróbálási sorozat a 0,1,2, ..., *m*-1 számokkal módosítja a kipróbálási indexet, amely nem függ a *k* kulcstól, tehát minden kulcsra azonos. (A kipróbálási sorozat csak a résen keresztül függ a kulcstól, az azonos résre képeződő kulcsok esetén azonos.) A hatás pedig az, hogy ha a vizsgált rés nem megfelelő, akkor tovább lépünk a magasabb indexek felé. A továbblépés ciklikus, azaz a tábla végére érve a másik végén folytatjuk a kipróbálást. A módszernek van egy kellemetlen mellékhatása, az *elsődleges klaszterezés*. *Klaszternek* nevezzük a kipróbálási sorozat mentén az egymást követő kitöltött rések összességét. Ezen halmaz elemeinek a száma a klaszter mérete.

Nem szerencsés, ha a klaszerek mérete nagy, mert ez lelassítja a hasító tábla műveleteit. Egy példa:



A hasító tábla szürkített elemei a foglaltak. A maximális klaszterméret 3, de van két egyelemű és egy kételemű klaszter is. Ha egy új kulcs a 0 indexű helyre kerülne, akkor a negyedik megvizsgált rés lenne csak megfelelő a tárolásra. Tegyük fel azonban, hogy egy új elem a 14-es résbe kerül, majd a következő a 12-esbe. Ez utóbbinak már egy 6 elemű klaszteren kell végigmennie, hogy megfelelő helyet találjon magának. A klaszterek összeolvadnak!

Négyzetes kipróbálás

A hasító függvény a négyzetes kipróbálás esetén $h(k,i) = ((h_0(k)) + c_1 \cdot i + c_2 \cdot i^2) \bmod m$ alakú. Az i értékei itt is a 0,1,2, ..., m-1. A c értékeket úgy kell megválasztani, hogy a kipróbálási sorozat kiadja az összes rést. A kipróbálási sorozat itt is csak a réstől függ, holott az 1,2, ...,m-1 számnak (m-1)! permutációja van. A klaszterezés jelensége itt is fellép, de nem olyan súlyos, mint a lineáris esetben. Itt <u>*másodlagos klaszterezésről*</u> beszélünk, mivel a klaszter elemei nem egymás mellett, hanem a kipróbálási sorozat mentén helyezkednek el. Egy lehetőség például egy négyzetes kipróbálásra, ha a $c_1 \cdot i + c_2 \cdot i^2$ korrekció úgy alakul, hogy az i=0,1,2,... értékekre a zérus, majd egy, azután három stb. értéket vesz fel, szabályát tekintve minden i értéknél az addigi i értékek összege lesz. (Határozzuk meg a megfelelő c konstansok értékét!) Természetesen ez az eset nem minden táblaméret (m) esetén megfelelő. Ha azonban a táblaméret 2-nek hatványa, akkor valóban kiadja az összes rést. (Bizonyítsuk be!)

Dupla hasítás

A dupla hasítás a $h(k,i) = (h_0(k) + i \cdot h_1(k)) \bmod m$ hasító függvényt használja. Láthatóan minden réshez általában m különböző sorozatot ad, azaz a kipróbálási sorozatok száma m^2, ami arra ad reményt, hogy a módszer az előzőekhez képest jobb tulajdonságokkal bír. Mindenesetre a h_1 megválasztásakor arra kell ügyelni, hogy az mindig m-hez relatív prímet szolgáltasson. Ellenkező esetben a kipróbálási sorozat csak a tábla elemeinek a d-edrészét adná, ahol d az m és a szolgáltatott h_1 érték legnagyobb közös osztója. (Lássuk be!) Ha m 2-hatványa és h_1 páratlan értékeket ad, az megfelel a feltételeknek. Ugyancsak megfelelő, ha m prímszám és h_1 mindig kisebb, mint m, de pozitív. Legyen m prím és legyen

$h_0(k) = k \bmod m$

$h_1(k) = 1 + (k \bmod (m-1))$

Válasszuk az m=701-et. A kulcsok legyenek négyjegyű számok. Az 1000-es kulcsra $h_0(1000) = 1000 \bmod 701 = 299$, $h_1(1000) = 1000 \bmod 700 = 301$ és $h(k,i)$=(299+i 301) mod 701. i=0,1,2,…,700. A kipróbálási sorozat: 299, 600, 200, 501, 101, …. A 9999-es kulcsra $h_0(9999)$=9999 mod 701=185, $h_1(9999)$=1+(9999 mod 700)=199 és $h(k,i)$=(185+i 199) mod 701. i=0,1,2,…,699. A kipróbálási sorozat: 185, 384, 583, 81, 280, ….

Tétel: <u>A nyílt címzésű hasító tábla időigénye</u>

Ha α a kitöltési arány, akkor a nyílt címzésű hasító táblában

$$C_n \le \frac{1}{1-\alpha} \text{ és } C_n^{'} \approx \frac{1}{\alpha} \ln \frac{1}{1-\alpha}. \quad (4)$$

4.1.4. Minimum és maximum keresése

Legyen most n kulcs elhelyezve egy tömbben, a kulcsok legyenek egy rendezett halmaz elemei (bármelyik kettő legyen összehasonlítható). Feladatunk a legkisebb és a legnagyobb kulcs megkeresése. A legkisebb kulcs megkeresése lineáris idejű és n-1 összehasonlítást igényel.

	4.1.4.1. algoritmus **Minimumkeresés tömbben**
	// $T(n) = \Theta(n)$
1	MINIMUMKERESÉS(*A, min*)
2	// Input paraméter: *A* – a tömb
3	// Output paraméter: *min* - a minimum értéke
4	//
5	$min \leftarrow A_1$
6	**FOR** $i \leftarrow 2$ **TO** ***hossz[A]*** *DO*
7	**IF** $min > A_i$
8	**THEN** $min \leftarrow A_i$
9	**RETURN** (*min*)

A legnagyobb elemet már n-2 összehasonlítással is megtaláljuk ezt követően. Összesen ez 2n-3 összehasonlítást jelent. A két kulcs meghatározási ideje lineáris és az aszimptotika konstansa 2. Ezen a konstanson tudunk némiképpen javítani az alábbi módszer alkalmazásával.

Legyen az elemek száma páros. Először az első két elemet hasonlítjuk össze (ez 1 összehasonlítás), a kisebbet minimumként, a nagyobbat a maximumként tároljuk. Ezután már csak elempárokkal dolgozunk ($\frac{n-2}{2}$ van). Összehasonlítjuk az elempár elemeit egymással (mindegyik 1 összehasonlítás), majd a kisebbet a minimummal, a nagyobbat a maximummal (további 2 összehasonlítás). Ha az addigi minimumot, vagy maximumot változtatni kell, akkor megtesszük. Összesen az összehasonlítások száma: $1+3\cdot\frac{n-2}{2}=1+\frac{3}{2}n-3\cdot\frac{2}{2}$ ami $\frac{3}{2}n-2$ és ez kevesebb, mint 2n-3

Ha páratlan számú elem van, akkor $\frac{n-3}{2}$ további elempár van az elsőt követően és marad még egy egyedüli elem a legvégén. Ezt az utolsó elemet mind az addigi minimummal, mind a maximummal össze kell hasonlítani legrosszabb esetben. Az összehasonlítások száma ennek megfelelően: $1+3\cdot\frac{n-3}{2}+2=\frac{3}{2}n-3\cdot\frac{3}{2}+3=\frac{3}{2}n-\frac{3}{2}<2n-3$

Az aszimptotika konstansa 2-ről 3/2-re csökkent.

4.1.5. Kiválasztás lineáris idő alatt

Definíció: A kiválasztási probléma
Legyen adott egy A halmaz (n különböző szám), és egy i index $1 \leq i \leq n$. Meghatározandó az A halmaz azon x eleme, melyre nézve pontosan i-1 darab tőle kisebb elem van az A halmazban.

Speciális esetben ha i=1, akkor a minimumkeresési problémát kapjuk. Mivel a minimumkeresési probléma n-1 összehasonlítással megoldható és ennyi kell is, ezért a probléma lineáris idő alatt megoldható. Ha növekvő sorrendbe rendezéssel próbáljuk megoldani a problémát, akkor mint később látni fogjuk $O(n \cdot \log n)$ lépésben a probléma mindig megoldható. Nem szükséges azonban a rendezéshez folyamodni, mert a probléma rendezés nélkül is megoldható, ráadásul lineáris időben, amiről alább lesz szó.

Definíció: A medián
Mediánnak nevezzük az adatsor azon elemét, amely a rendezett sorban a középső helyet foglalja el. Ha páratlan számú elem van az adatsorban, akkor n=2k-1 és így a medián indexe a rendezés után k. Ha páros számú elem van az adatsorban, akkor n=2k, és ekkor két középső elem van a k és a k+1 indexű a rendezés után. (Alsó medián, felső medián.)

Ha nem említjük, akkor az alsó mediánról beszélünk.

1. Példa: A 123, 234, 345, 444, 566, 777, 890 rendezett adatsorban a medián a 444, míg a 123, 234, 345, 444, 566, 777, 890, 975 sorban két medián van, a 444 (alsó medián) és az 566 (felső medián).

A lineáris idejű kiválasztási algoritmusnak szüksége lesz agy segédalgoritmusra, amely egy előre megadott számnak megfelelően az adatsort két részre osztja úgy, hogy az első részbe kerülnek azok az adatok, amelyek nem nagyobbak, a második részbe a nem kisebbek kerülnek.

	4.1.5.1. algoritmus **Résztömb felosztása előre adott érték körül**
	// $T(n) = \Theta(n)$
1	FELOSZT(A,p,r,x, q)
2	// Input paraméter: A – a tömb
3	// p - a felosztandó rész kezdőindexe
4	// r - a felosztandó rész végindexe
5	// x - az előre megadott érték, amely a felosztást szabályozza
6	// Output paraméter: A – a megváltozott tömb
7	// q – a felosztás határa $A_{p...q}$; $A_{q+1...r}$
8	//
9	$i \leftarrow p-1$
10	$j \leftarrow r+1$
11	**WHILE** IGAZ **DO**
12	**REPEAT** $j \leftarrow j-1$
13	**UNTIL** $A_j \leq x$
14	**REPEAT** $i \leftarrow i+1$

15	**UNTIL** $A_i \geq x$
16	**IF** $i < j$
17	**THEN** Csere $A_i \leftrightarrow A_j$
18	**ELSE** $q \leftarrow j$
19	**RETURN (** A, q **)**

Az előre adott x értéket az $A_{p \ldots r}$ résztömb elemei közül jelöljük ki. Elemezzük az algoritmus viselkedését a kijelölt elem függvényében. Lehet-e olyan elemet megadni, hogy az algoritmus végtelen ciklusba kerüljön?

2. Példa: Az algoritmus munkáját az alábbi tömbön szemléltetjük. Itt most p=1, r=15, x=5.

	9	7	2	1	8	3	5	2	9	5	3	1	2	3	6		
i																j	kezdőállapot
	i													j			Csere
	3	7	2	1	8	3	5	2	9	5	3	1	2	9	6		
		i											j				Csere
	3	2	2	1	8	3	5	2	9	5	3	1	7	9	6		
					i							j					Csere
	3	2	2	1	1	3	5	2	9	5	3	8	7	9	6		
							i				j						Csere
	3	2	2	1	1	3	3	2	9	5	5	8	7	9	6		
									i	j							Csere
	3	2	2	1	1	3	3	2	5	9	5	8	7	9	6		
									j	i							eljárás vége

Az eljárás befejeztével a tömb 1,…,9 indexű elemei nem nagyobbak, mint 5, a 10,…,15 indexű elemek pedig nem kisebbek, mint 5.

4.1.5.2. algoritmus **Kiválasztás lineáris időben**	
	// $T(n) = \Theta(n)$
1	KIVÁLASZT (A, i, x)
2	// Input paraméter: A – a tömb
3	// i - a növekvő sorrendbe rendezés esetén a keresett elem indexe
4	// Output paraméter: x – a keresett elem
5	//
6	Ha $n = 1$, akkor x maga az A_1 elem. **RETURN** (x)
7	Ha $n \neq 1$, akkor osszuk fel a tömböt $\lceil n/5 \rceil$ darab 5-elemű csoportra. (Esetleg a legutolsó csoportban lesz 5-nél kevesebb elem.)
8	Az összes $\lceil n/5 \rceil$ csoportban megkeressük a mediánt.
9	A KIVÁLASZT algoritmus rekurzív alkalmazásával megkeressük az $\lceil n/5 \rceil$ darab medián mediánját (medmed)
10	A FELOSZT algoritmus segítségével a **mediánok mediánja (medmed) körül** felosztjuk a bemeneti tömböt két részre. Legyen k elem az alsó és $n–k$ a felső részben.
11	A KIVÁLASZT algoritmus *rekurziójával* keressük az i-dik elemet a felosztás alsó részében, ha $i \leq k$, vagy pedig az $i–k$ -adikat a felső részben egyébként.
12	*RETURN* (x)

Tétel: A KIVÁLASZT algoritmus időigénye

A KIVÁLASZT algoritmus lineáris idejű.

Bizonyítás

$\left\lceil \frac{n}{5} \right\rceil$ csoport alakult ki. Mindegyikben meghatároztuk a mediánt. Ezen mediánok mediánját is meghatározzuk. Az adatok között a mediánok mediánjánál nagyobb elemek számát meg tudjuk becsülni az alábbi meggondolással. Mivel a medián középen lévő elem, így az a mediánok mediánja, amely $\left\lceil \frac{n}{5} \right\rceil$ medián közül kerül ki. Ezen mediánok fele biztosan nagyobb, mint a mediánok mediánja, azaz legalább $\left\lceil \frac{n}{5} \right\rceil \cdot \frac{1}{2} - 1$ ilyen elem van (saját magát nem számítjuk bele). Minden ilyen medián csoportjában akkor legalább három elem nagyobb a medánok mediánjánál, kivéve az esetleg 5-nél kevesebb elemű utolsó csoportot, amit szintén elhagyunk. Ezek alapján legalább $3 \cdot \left(\left\lceil \frac{n}{5} \right\rceil \cdot \frac{1}{2} - 2 \right) \geq \frac{3}{10} n - 6$ elem biztosan nagyobb, mint a mediánok mediánja. (Ugyanennyi adódik a kisebb elemek számára is.)

Az 11-es sorban a KIVÁLASZT algoritmus a fentiek szerint a felosztás másik részében legfeljebb $n - \left(\frac{3}{10} n - 6 \right) = \frac{7}{10} n + 6$ elemmel dolgozhat.

A KIVÁLASZT algoritmus egyes lépéseinek az időigénye:

Sor	Időigény
7.	O(n)
8.	O(n)
9.	T(⌈ n/5 ⌉)
10.	O(n)
11.	T(7n/10+6)

Az időigényeket összegezve érvényes: a $T(n) \leq T\left(\left\lceil \frac{1}{5} n \right\rceil \right) + T\left(\frac{7}{10} n + 6 \right) + O(n)$ összefüggés. Legyen itt $O(n)$ konstansa *a*. Feltételezzük, hogy a megoldás $T(n) \leq c \cdot n$ egy bizonyos n küszöbtől kezdve, és behelyettesítéssel ezt fogjuk bizonyítani.

$$T(n) \leq c \cdot \left\lceil \frac{1}{5} n \right\rceil + c \cdot \left(\frac{7}{10} n + 6 \right) + a \cdot n \leq c \cdot \left(\frac{1}{5} n + 1 \right) + c \cdot \left(\frac{7}{10} n + 6 \right) + a \cdot n =$$

$$= c \cdot \left(\frac{9}{10} n + 7 \right) + a \cdot n = c \cdot n - \left(c \cdot \frac{1}{10} n - 7c - a \cdot n \right)$$

Válasszuk *n*-et úgy, hogy a zárójel nem negatív legyen.Akkor $c \geq \frac{10a \cdot n}{n - 70}$.

Ha ezen felül $n \geq 140$, akkor a $c \geq 20a$ választás megfelelő a kiinduló feltételezésünk teljesüléséhez. ■

A sztring adatszerkezet, sztringkereső algoritmusok

A sztring olyan szekvenciális lista, amelynek az elemei egy ábécé szimbólumai. Ezeket a szimbólumokat karaktereknek nevezzük.

Sztringgel végezhető műveletek

- Létrehozás: explicit módon felsoroljuk a sztring összes karakterét.
- Bővítés: bárhol bővíthető. Bővítéskor két részsztringet képzünk, majd konkatenáljuk azokat a beszúrandó sztringgel.
- Törlés: megvalósítható a fizikai törlés, melynek során két részsztringet képzünk (melyekben már nem szerepel a törlendő részsztring), majd konkatenáljuk azokat.
- Csere: cserélhetünk egy karaktert, de részsztring is cserélhető másik részsztringre. Két részsztringet képzünk, majd konkatenáljuk azokat az új értéket képviselő sztringgel (törlés+bővítés).
- Rendezés: nem értelmezett.
- Keresés: értelmezhető, kereshetünk egy karaktert vagy egy részsztringet.
- Elérés: soros vagy közvetlen.
- Bejárás: értelmezhető.

A listaelemek tartalmazhatnak egy karaktert vagy egy részsztringet. Utóbbi esetben a részsztringek eltérőhosszúságúak lehetnek, és valamelyik folytonos reprezentációval ábrázoljuk őket.

Sztingkereső algoritmusok

Egy sztringben keresünk egy másik sztringet. Azt a sztringet, amelyikbenkeresünk, alapsztringnek, azt a sztringet pedig, amitkeresünk, mintasztringnek nevezzük. A pszeudokódokban az alapsztringet Aval, a mintasztringet Pvel fogjuk jelölni.
Néhány sztringkereső algoritmus:

- mezítlábas (brute force) algoritmus
- Knuth–Morris–Pratt algoritmus
- Boyer–Moore algoritmus
- Rabin–Karp algoritmus
- ShiftAnd (Dömölki Bálint-féle) és ShiftOr algoritmus

1: BRUTEFORCE(A, P)

```
2: n←hossz(A)
3: m ← hossz(P)
4: i ←0
5: j ←0
6: while i< n and j< m do
7:      if A[i+ 1] = P[j+ 1]
8:            then
9:                  i ← i+ 1
10:                 j← j+ 1
11:           else
12:                 i←i-j+1
13:                 j ←0
14: if j= m
15:     then return(i-m+1)
16:     else return 0
```

Prefix, szuffix

Legyen Σ egy ábécé és $x = x_1 \dots x_k$ ($k \in \mathbb{N}$) egy k hosszúságú sztring Σ felett! Az x-nek az u részsztring egy prefixe, ha

$$u = x_1 \dots x_b, \text{ahol } 0 \leq b \leq k,$$

azaz ha x u-val kezdődik. Az x-nek az u részsztring egy szuffixe, ha

$$u = x_{k-b+1} \dots x_k, \text{ahol } 0 \leq b \leq k,$$

azaz ha x u-val végződik.

Valódi prefix, valódi szuffix

Az x egy u prefixét vagy x egy u szuffixét valódi prefixnek vagy szuffixnek nevezzük, ha $u \neq x$, azaz ha $b < k$. Ha $b = 0$, akkor $u = \varepsilon$ (üres sztring).

Border

Legyen Σ egy ábécé és $x = x_1 \ldots x_k$ ($k \in \mathbb{N}$) egy k hosszúságú sztring Σ felett! Az x-nek az r részsztring egy **bordere**, ha

$$r = x_1 \ldots x_b \text{ és } r = x_{k-b+1} \ldots x_k, \text{ahol } 0 \leq b < k.$$

Az x bordere egy olyan részsztring, amely valódi prefixe és valódi szuffixe is x-nek. Ekkor a részsztring b hosszát a border **hosszának** nevezzük. Ha $b = 0$, akkor $r = \varepsilon$ (üres sztring).

Példa

Legyen $x = abacab$. Az x valódi prefixei:

$$\varepsilon, a, ab, aba, abac, abaca.$$

Az x valódi szuffixei:

$$\varepsilon, b, ab, cab, acab, bacab.$$

Az x borderei:

$$\varepsilon, ab.$$

Az ε border hossza 0, az ab border hossza 2.

Megjegyzés

Az ε üres sztring minden $x \in \Sigma^+$ sztringnek bordere. Az ε üres sztringnek nincs bordere.

Példa

1	2	3	4	5	6	7	8	9	...
a	b	c	a	b	c	a	b	d	
a	b	c	a	b	d				
			a	b	c	a	b	d	

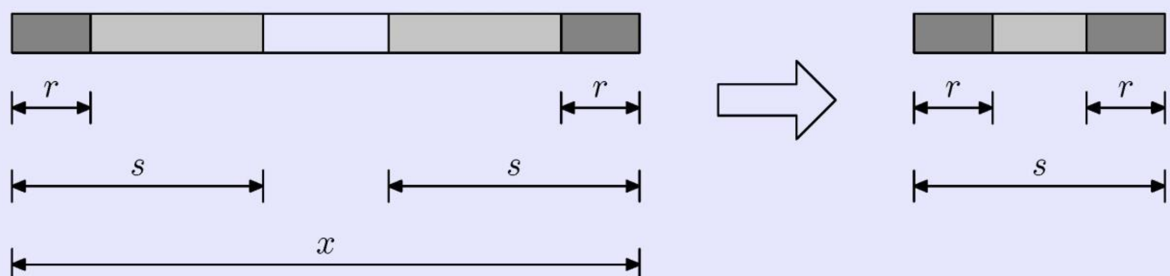
Az 1,...,5 pozíciókon lévő karakterek megegyeznek. A 6. pozíción a *c* és *d* karakterek eltérnek. A minta 3 pozícióval tovább léptethető, és az összehasonlítások a 6. pozíciótól folytathatók.
A léptetés mértékét a *p* egyező prefixének a legszélesebb bordere határozza meg. Ebben a példában az egyező prefix *abcab*, a hossza $j = 5$. Az ő legszélesebb bordere *ab*, amely $b = 2$ hosszúságú. A léptetés mértéke $j - b = 5 - 2 = 3$.

Az előfeldolgozási szakaszban a minta minden egyes prefixéhez meg kell határozni a legszélesebb border hosszát. Később a keresési szakaszban a léptetés mértéke az egyező prefixeknek megfelelően számítható.

Tétel

Legyen *r* és *s* egy *x* sztring bordere, ahol $|r| < |s|$. Ekkor *r* egy bordere *s*-nek.

Bizonyítás

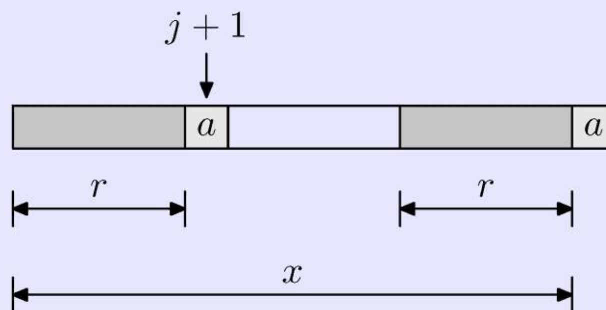


Megjegyzés

Ha s a legszélesebb bordere x-nek, akkor x következő legszélesebb r borderét megkapjuk s legszélesebb bordereként, és így tovább...

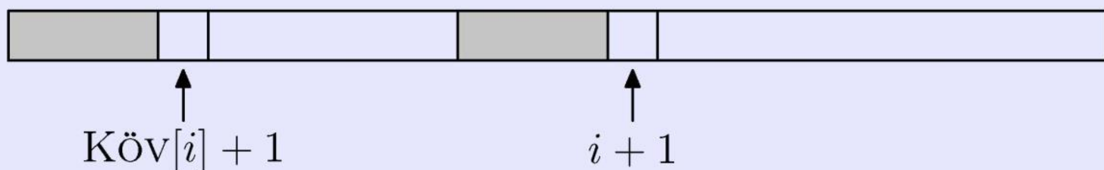
Definíció

Legyen x egy sztring és $a \in \Sigma$ egy karakter. Az x egy r bordere bővíthető a-val, ha ra egy bordere xa-nak.



A Köv tömb

Az előfeldolgozási szakaszban egy $m + 1$ elemű Köv tömböt számítunk ki. A tömb Köv[i] eleme a mintasztring i hosszúságú prefixéhez tartozó legszélesebb border hossza ($i = 0, \ldots, m$). Mivel az $i = 0$ hosszúságú ε sztringnek nincsen bordere, ezért Köv[0] $= -1$.



Feltéve, hogy a Köv[0], ..., Köv[i] értékeket már ismerjük, a Köv[$i + 1$] értékét kiszámíthatjuk, ha ellenőrizzük, hogy a $p_1 \ldots p_i$ prefix egy bordere bővíthető-e a p_{i+1} karakterrel. Ez abban az esetben tehető meg, ha $p_{\text{Köv}[i]+1} = p_{i+1}$. A bordereket a Köv[$i$], Köv[Köv[$i$]], ... értékek csökkenő sorrendjében kell megvizsgálni.

Előfeldolgozó algoritmus

```
1: function KÖVFELTÖLT(P)
2: m← hossz(P)
3: i←0
4: j← -1
5: KÖV[0] ← -1
6: while i< m do
7:      if j=-1 or P[i+1]=P[j+1]
8:             then
9:                   i←i+1
10:                  j←j+1
11:                  KÖV[i] ←j
11:            else
12:                  j ← KÖV[j]
13: return KÖV
```

Példa

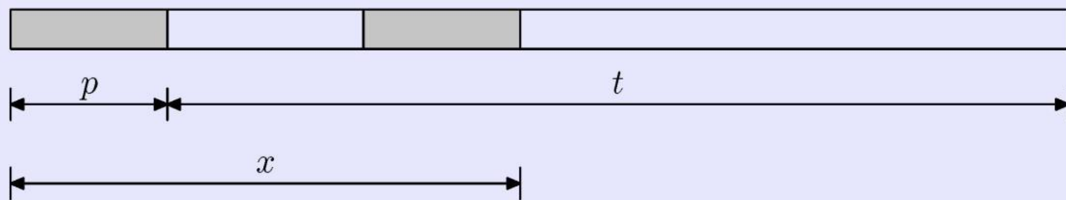
A $p = ababaa$ minta esetén a borderek szélességei a következő értékeket veszik fel a KÖV tömbben:

$i:$	0	1	2	3	4	5	6
$p[i]:$		a	b	a	b	a	a
$\text{KÖV}[i]:$	-1	0	0	1	2	3	1

Láthatjuk például, hogy KÖV$[5] = 3$, mivel az 5 hosszúságú *ababa* prefixnek van egy 3 hosszúságú bordere.

Megjegyzés

Elméletileg semmi akadálya annak, hogy az előző előfeldolgozó algoritmust a *pt* sztringre alkalmazzuk a *p* helyett. Ha a bordereket csak a *p* minta *m* szélességéig számoljuk ki, akkor a *pt* valamely *x* prefixének egy *m* szélességű bordere megfelel a minta egy előfordulásának *t*-ben (feltéve, hogy a border nem önátfedő).



Knuth_Morris-Pratt algoritmus

1: **function** KMP-KERESÉS(A, P)
2: n←hossz(A)
3: m←hossz(P)
4: KÖV←KÖVFELTÖLT(P)
5: i←0
6: j← 0
7: **while** i< n **and** j< m **do**
8: **if** j=-1 **or** A[i+1]=P[j+1]
9: **then**
10: i←i+1
11: j←j+1
12: **else**
13: j←KÖV[j]
14: **if** j=m
15: **then return** (i- m+1)
16: **else return** (0)