Algoritmusok és adatszerkezetek II. Útmutatások a tanuláshoz, Tematika (2016)

Kedves Hallgatók!

A vizsgára való készülésben elsősorban az előadásokon és a gyakorlatokon készített jegyzeteikre támaszkodhatnak. További ajánlott források:

Hivatkozások

[1] Ásványi Tibor, Algoritmusok és adatszerkezetek II.
Útmutatások a tanuláshoz, Tematika (2016)
http://aszt.inf.elte.hu/∼asvanyi/ad/ad2programok.pdf

[2] Cormen, T.H., Leiserson, C.E., Rivest, R.L., Stein, C.,
magyarul: Új Algoritmusok, *Scolar Kiadó*, Budapest, 2003.
ISBN 963 9193 90 9
https://app.box.com/s/7tub2koyp9sx88hrbc68
angolul: Introduction to Algorithms (Third Edititon),
The MIT Press, 2009.

[3] Fekete István, Algoritmusok jegyzet
http://people.inf.elte.hu/fekete/algoritmusok_jegyzet/

[4] Rónyai Lajos – Ivanyos Gábor – Szabó Réka, Algoritmusok,
TypoTEX Kiadó, 1999. ISBN 963 9132 16 0

[5] Weiss, Mark Allen, Data Structures and Algorithm Analysis,
Addison-Wesley, 1995, 1997, 2007, 2012, 2013.

A vizsgákon az elméleti kérdések egy-egy tétel bizonyos részleteire vonatkoznak. Lesznek még megoldandó feladatok, amik részben a tanult algoritmusok működésének szemléltetését, bemutatását, részben a szerzett ismeretek kreatív felhasználását kérik számon. Egy algoritmus, program, művelet bemutatásának mindig része a műveletigény elemzése.

Az előadások elsősorban a CLRS könyv [2] (ld. alább) angol eredetijének harmadik kiadását követik. (Az érintett fejezetekben a magyar és az angol változat között leginkább csak néhány jelölésbeli különbséget találtunk.)

Ez a jegyzet egyenlőre jó esetben is csak az előadások vázlatát és a legfontosabb struktogramokat tartalmazza. Az egyes témák részletes kidolgozása a hivatkozott szakoirodalomban, elsősorban [2]-ben található.

Az egyes struktogramokat általában nem dolgozzuk ki az értékadó utasítások szintjéig. Az olyan implementációs részleteket, mint a listák és egyéb adatszerkezetek, adattípusok műveleteinek pontos kódja, a dinamikusan allokált objektumok deallokálása stb. az Olvasóra hagyjuk, hiszen ezekkel az előző félévben foglalkoztunk. Használni fogunk olyan absztrakt fogalmakat, mint a véges halmazok, sorozatok. A struktogramokban a „for" ciklusok (illetve a „for each" ciklusok) mintájára alkalmazni fogunk a ciklusfeltételek helyén pl. „$\forall v \in V$" alakú kifejezéseket, ami azt jelenti, hogy a ciklusmagot a V halmaz minden v elemére végre kell hajtani.

A fentiek szerint az egyszerűbb programrészletek helyén gyakran szerepelnek majd magyar nyelvű utasítások, amiknek részletes átgondolását, esetleges kidolgozását, a korábban tanuktak alapján, szintén az Olvasóra bízzuk. Az ilyen, formailag általában felszólító mondatok végéről a felkiáltójelet elhagyjuk.

Tematika

Minden tételhez: Hivatkozások: például a „[2] 8.2, 8.3, 8.4” jelentése: a [2] sorszámú szakirodalom adott (al)fejezetei.

1. Rendezés lineáris időben ([2] 8.2). Edényrendezés (Bucket-Sort, [2] 8.4, [3] 21). A stabil rendezés fogalma ([2] 8.2). Leszámláló rendezés (Counting-Sort, [2] 8.2). Radix rendezés (Radix-Sort) tömbökre ([2] 8.3) és láncolt listákra ([3] 21; a láncolt eset struktogramja a gyakorlaton). Gyakorlat: Rendezés bináris számok tömbjén, „előre” és „vissza” ([3] 21).

2. Hasító táblák ([2] 11). Direkt címzés (direct-address tables). Hasító táblák (hash tables). A hasító függvény fogalma (hash functions). Kulcsütközések (collisions).

Kulcsütközések feloldása láncolással (collision resolution by chaining); keresés, beszúrás, törlés (search and update operations); kitöltöttségi arány (load factor); egyszerű egyenletes hasítás (simple uniform hashing); átlagos keresési idő (average-case time of search: a tételek bizonyítás nélkül).

Jó hash függvények (good hash functions), egy egyszerű hash függvény (kulcsok a $[0, 1)$ intervallumon), az osztó módszer (the division method), a szorzó módszer (the multiplication method).

Nyílt címzés (open addressing); próba sorozat (probe sequence); keresés, beszúrás, törlés (search and update operations); üres és törölt rések (empty and deleted slots); a lineáris próba, elsődleges csomósodás (linear probing, primary clustering); négyzetes próba, másodlagos csomósodás (quadratic probing, secondary clustering), a 11-3 problémával együtt; kettős hash-elés (double hashing); az egyenletes hasítás (uniform hashing) fogalma; a keresés próba sorozata várható hosszának felső becslései egyenletes hasítást feltételezve.

3. Elemi gráf algoritmusok ([2] 22). Gráf ábrázolások (representations of graphs).

A szélességi gráfkeresés (breadth-first search: BFS). A szélességi gráfkeresés futási ideje (the run-time analysis of BFS). A legrövidebb utak (shortest paths). A szélességi feszítőfa (breadth-first tree). HF: A szélességi gráfkeresés megvalósítása a klasszikus gráf ábrázolások esetén; hatékonyság.

A mélységi gráfkeresés (depth-first search: DFS). Mélységi feszítő erdő (depth-first forest). A gráf csúcsainak szín és időpont címkéi (colors and timestamps of vertexes). Az élek osztályozása (classification of edges, its connections with the colors and timestamps of the vertexes). A mélységi gráfkeresés futási ideje (the run-time analysis of DFS). Topologikus rendezés (topological sort). Erősen összefüggő komponensek (strongly connected components). HF: A mélységi gráfkeresés és a topologikus rendezés megvalósítása a klasszikus gráf ábrázolások esetén; hatékonyság.

4. Minimális feszítőfák (Minimum Spanning Trees: MSTs). Egy általános algoritmus (A general algorithm). Egy tétel a biztonságos élekről és a minimális feszítőfákról (A theorem on safe edges and MSTs). Prim és Kruskal algoritmusai (The algorithms of Kruskal and Prim). A futási idők elemzése (Their run-time analysis). HF: A Prim algoritmus implementációja a két fő gráfábrázolás és a szükséges prioritásos sor különböző megvalósításai esetén (The implementations of the algorithm of Prim with respect to the main graph representations and representations of the priority queue).

5. Legrövidebb utak egy forrásból (Single-Source Shortest Paths). A legrövidebb utak fája (Shortest-paths tree). Negatív körök (Negative cycles). Közelítés (Relaxation).

A szélességi vagy sor-alapú (Queue-based) Bellman-Ford algoritmus. A menet (pass) fogalma. Futási idő elemzése. Helyessége. A legrövidebb út kinyomtatása.

Legrövidebb utak egy forrásból, körmentes irányított gráfokra. (DAG shortest paths.) Futási idő elemzése. Helyessége.

Dikstra algoritmusa. Helyessége. Fontosabb implementációi a két fő gráfábrázolás és a szükséges prioritásos sor különböző megvalósításai esetén. A futási idők elemzése.

6. Legrövidebb utak minden csúcspárra (All-Pairs Shortest Paths). A megoldás ábrázolása a (D, Π) mátrix-párral. HF: Adott csúcspárra a legrövidebb út kinyomtatása.

A Floyd-Warshall algoritmus és a $(D^{(k)}, \Pi^{(k)})$ mátrix párok. A futási idő elemzése. Összehasonlítás a Dijkstra algoritmus, illetve (HF:) a sor-alapú Bellman-Ford algoritmus $|G.V|$-szeri végrehajtásával.

Irányított gráf tranzitív lezártja (Transitive closure of a directed graph) és a $T^{(k)}$ mátrixok. Az algoritmus és futási ideje. HF: összehasonlítás a szélességi keresés $|G.V|$-szeri végrehajtásával.

7. Mintaillesztés (String Matching). Egy egyszerű mintaillesztő algoritmus (The naive string-matching algorithm). A futási idő elemzése.

A Rabin-Karp algoritmus. Inicializálása. A futási idő elemzése.

A Knuth-Morris-Pratt algoritmus. Inicializálása. A futási idő elemzése.

A Quick Search algoritmus. Inicializálása. A futási idő elemzése.

8. Adattömörítés (Data Compression) [4]. Karakterenkénti tömörítés állandó kódhosszal.

Prefix kód. A Huffman kód. A Huffman fa. A Huffman kód visszafejtése. A Huffman kód optimalitása. Ez a legjobb megoldás?

A Lempel–Ziv–Welch (LZW) módszer. Kódolás szótárépítéssel. Dekódolás a szótár rekonstruálásával. A módszer előnyei.

1. Rendezés lineáris időben ([2] 8)

Alapvetően nem kulcsösszehasonlítással rendezünk ([2] 8.2), így ezekre az algoritmusokra nem vonatkozik az öszehasonlító rendezések alaptétele ([2] 8.1 tétel).

1.1. Edényrendezés (Bucket-Sort, [2] 8.4, [3] 21)

Feltesszük, hogy a rendezendő elemek kulcsai a $[0, 1)$ valós intervallum elemei. (Ha a kulcs egész vagy valós szám típusú, vagy azzá konvertálható, továbbá tudunk a kulcsok számértékeire alsó és felső határt mondani, akkor a kulcsok számértékei nyilván normálhatók a $[0, 1)$ valós intervallumra.)

Az alábbi algoritmus akkor lesz hatékony, ha az input kulcsai a $[0, 1)$ valós intervallumon egyenletesen oszlanak el.

bucket_sort($A[1..n]$)

Hozzuk létre a $B[0..n-1]$ edényeket
$j := 0..n-1$
Legyen $B[j]$ üres lista
$i := 1..n$
Szúrjuk be az $A[i]$ elemet a $B[\lfloor n * (A[i].kulcs) \rfloor]$ listába
$j := 0..n-1$
Rendezzük monoton növekvően a $B[j]$ listát
Fűzzük össze sorban a $B[0]$, $B[1]$, $\ldots$, $B[n-1]$ listákat az L listába
Másoljuk vissza az L lista tartalmát sorban az $A[1..n]$ tömbbe

$mT(n) \in \Theta(n)$. A fenti egyenletes eloszlást feltételezve $AT(n) \in \Theta(n)$. $MT(n)$ attól függ, hogy a $B[j]$ listákat milyen módszerrel rendezzük. Pl. egyszerű beszúró rendezést használva $MT(n) \in \Theta(n^2)$, (vegyes) összefésülő rendezéssel viszont $MT(n) \in \Theta(n \lg n)$.

A fenti séma természetesen könnyen átírható láncolt listák rendezésére. Ebben az esetben megtakaríthatjuk a listaelemek allokálását és deallokálását is, ami, ha aszimptotikusan nem is, a gyakorlatban mégis gyorsabb programot eredményez. Legyen például adott az L egyszerű láncolt lista:

bucket_sort(&L)

bucket_sort(&L)	
n := L hossza	
Hozzuk létre a $B[0..n-1]$ edényeket	
$j := 0..n-1$	
	Legyen $B[j]$ üres lista
$L \neq \oslash$	
	Fűzzük ki L első elemét
	Szúrjuk be ezt az elemet a kulcsa (k) szerint a $B[\lfloor nk \rfloor]$ listába
$j := 0..n-1$	
	Rendezzük monoton növekvően a $B[j]$ listát
Fűzzük össze sorban a $B[0]$, $B[1]$, ..., $B[n-1]$ listákat az L listába	

HF: Részletezzük az elemi utasítások szintjéig ez utóbbi kódot!

1.2. Leszámláló rendezés (Counting-Sort, [2] 8.2)

A stabil rendezés fogalma ([2] 8.2).

Feladat: Adottak $A[1..n]:T$ tömb, $k \in O(n)$ pozitív egész, $\varphi : T \to 0..k$ kulcsfüggvény. Rendezzük az $A[1..n]:T$ tömböt lineáris időben stabil rendezéssel úgy, hogy az eredmény a $B[1..n]:T$ tömbben keletkezzék!

counting_sort($A[1..n], k, \varphi, B[1..n]$)	
Hozzuk létre a $C[0..k]:\mathbb{N}$ tömböt	
$j := 0..k$	
	$C[j] := 0$
$i := 1..n$	
	$C[\varphi(A[i])] + +$
$j := 1..k$	
	$C[j] += C[j-1]$
$i := n..1 \quad (-1)$	
	$j := \varphi(A[i])$
	$B[C[j]] := A[i]$
	$C[j] - -$

A műveletigény $\Theta(n + k)$. Feltételezve, hogy $k \in O(n)$, $\Theta(n + k) = \Theta(n)$, azaz $T(n) \in \Theta(n)$.

1.3. Radix rendezés (Radix-Sort) tömbökre ([2] 8.3)

A rendezendő tömb kulcsai $k+1$ alapú számrendszerben felírt, d-jegyű nemnegatív egész számok. A jobbról első számjegy helyiértéke a legkisebb, míg a d-ediké a legmagasabb.[1]

radix_sort($A[1..n], d$)

$i := 1..d$

Rendezzük az A tömböt az elemek kulcsainak jobbról i-edik számjegye szerint stabil rendezéssel

Ha a stabil rendezés a leszámláló rendezés, akkor a műveletigény $\Theta(d(n+k))$, mivel a teljes rendezés d leszámláló rendezésből áll. Feltételezve, hogy d konstans és $k \in O(n)$, $\Theta(d(n + k)) = \Theta(n)$, azaz $T(n) \in \Theta(n)$. Ha pl. a kulcsok négy bájtos nemnegatív egészek, választhatjuk számjegyeknek a számok bájtjait, így $d = 4$ és $k = 255$, mindkettő n-től független konstans, tehát a feltételek teljesülnek, és a rendezés lineáris időben lefut. Az i-edik számjegy, azaz bájt kinyerése egyszerű és hatékony. Ha *key* a kulcs, akkor pl. C++-ban

$$(key >> (8 * (i - 1)))\&255$$

az i-edik számjegye[2]. Ld. még [2] 8.3-ban, hogy n rendezendő adat, b bites természetes szám kulcsok és r bites „számjegyek" esetén, a radix rendezésben, b és n függvényében, hogyan érdemes r-et megválasztani!

HF: Részletezzük a *radix-sort* fenti, absztrakt programját úgy, hogy a stabil rendezéshez leszámláló rendezést használunk, és a „számjegyek" a teljes b bites kulcs r bites szakaszai! Vegyük figyelembe, hogy ettől a paraméterezés is változik, és hogy érdemes felváltva hol az eredeti $A[1..n]$ tömbből a $B[1..n]$ segédtömbbe, hol a $B[1..n]$-ből az $A[1..n]$-be végezni a leszámláló rendezést!

[1] Általánosítva, a kulcs felbontható d kulcs direkt szozatára, ahol a jobbról első leglevésbé szignifikáns, míg a d-edik a leglényegesebb. Pl. ha a kulcs dátum, akkor először a napok, majd a hónapok és végül az évek szerint alkalmazunk stabil rendezést. Ez azért jó, mert – a stabilitás miatt – amikor a hónapok szerint rendezünk, az azonos hónapba eső elemek a napok szerint rendezettek maradnak, és amikor az évek szerint rendezünk, az azonos évbe eső elemek hónapok és ezen belül napok szerint szintén sorban maradnak.

[2] ami tovább egyszerűsödik, ha a programban i helyett az eltolás mértékét tartjuk nyilván (ez persze egyenlő $8 * (i - 1)$-gyel)

1.4. Radix rendezés láncolt listákra ([3] 21)

Példa: (legyen $k = 1$ (bináris számok) és $d = 3$ (három „számjegy”):

```
Az eredeti lista (szimbolikus jelöléssel):
L = <101,001,100,111,110,010,000,011>

Első menet:
A[0] = <100,110,010,000>
A[1] = <101,001,111,011>
L = <100,110,010,000,101,001,111,011>

Második menet:
A[0] = <100,000,101,001>
A[1] = <110,010,111,011>
L = <100,000,101,001,110,010,111,011>

Harmadik menet:
A[0] = <000,001,010,011>
A[1] = <100,101,110,111>
L = <000,001,010,011,100,101,110,111>
```

A láncolt esetet a struktogrammal együtt a gyakorlatokon tárgyaljuk.

1.5. Gyakorlat: Rendezés bináris számok tömbjén, „előre” és „vissza” ([3] 21)

2. Hasító táblák ([2] 11)

A mindennapi programozási gyakorlatban sokszor van szükségünk ún. szótárakra, amelyek műveletei: (1) adat beszúrása a szótárba, (2) kulcs alapján a szótárban a hozzá tartozó adat megkeresése, (3) a szótárból adott kulcsú, vagy egy korábbi keresés által lokalizált adat törlése.

Az AVL fák, B+ fák és egyéb kiegyensúlyozott keresőfák mellett a szótárakat gyakran hasító táblákkal valósítják meg, feltéve, hogy a műveleteknek nem a maximális, hanem az átlagos futási idejét szeretnék minimalizálni. Hasító táblát használva ugyanis a fenti műveletekre elérhető az ideális, $\Theta(1)$ átlagos futási idő azon az áron, hogy a maximális műveletigény általában $\Theta(n)$.

Jelölések:
m : a hasító tábla mérete
$T[0..m-1]$: a hasító tábla
$T[0], T[1], \ldots, T[m-1]$: a hasító tábla rései (slot-jai)
$\oslash$: üres rés a hasító táblában (NIL)
$\otimes$: törölt rés a hasító táblában (nyílt címzésnél)
n : a hasító táblában tárolt adatok száma
$\alpha = n/m$: a hasító tábla kitöltöttségi aránya
U : a kulcsok univerzuma; $k, k', k_i \in U$
$h : U \to 0..m-1$: hasító függvény
$h : U \times 0..m-1 \to 0..m-1$: hasító próba (nyílt címzésnél)
$\langle h(k,0), h(k,1), \ldots, h(k,m-1) \rangle$: próba sorozat (nyílt címzésnél)

Feltesszük, hogy $h(k)$ illetve $h(k,i)$ $\Theta(1)$ időben számolható.

2.1. Direkt címzés (direct-address tables)

Feltesszük, hogy $U = 0..m-1$, ahol $m \geq n$, de m nem túl nagy. A $T[0..m-1]$ hasító tábla rései ponterek, p a beszúrandó/törlendő rekordra mutat. A hasító táblát $\oslash$ pointerekkel inicializáljuk.

keres$(T, k) = T[k]$
beszúr(T, p) : $T[p \to kulcs] := p$
töröl(T, p) : $T[p \to kulcs] := \oslash$

Mindhárom művelet futási ideje $\Theta(1)$.

2.2. Hasító táblák (hash tables)

Hasító függvény (hash function): Ha $|U| >> n$, a direkt címzés nem alkalmazható, vagy nem gazdaságos, ezért $h : U \to 0..m-1$ hasító függvényt alkalmazunk, ahol tipikusan $|U| >> m$. A k kulcsú adatot a $T[0..m-1]$ hasító tábla $T[h(k)]$ résében tároljuk (próbáljuk tárolni).

Kulcsütközések (collisions): Ha két adat k_1, k_2 kulcsára $h(k_1) = h(k_2)$, kulcsütközésről beszélünk. Mivel $|U| >> m$, a kulcsütközést kezelni kell.

Az egyszerűség kedvéért – a Cormen könyvet [2] követve – feltesszük, hogy a beszúrandó adat kulcsa még nem szerepel a táblázatban (vagy ha úgy tetszik, nem foglalkozunk a duplikált kulcsokkal).

2.3. Kulcsütközések feloldása láncolással (collision resolution by chaining)

Keresés, beszúrás, törlés (search and update operations); kitöltöttségi arány (load factor); egyszerű egyenletes hasítás (simple uniform hashing); átlagos keresési idő (average-case time of search: a tételek bizonyítás nélkül).

2.4. Jó hash függvények (good hash functions)

Egy egyszerű hash függvény (kulcsok a $[0,1)$ intervallumon), az osztó módszer (the division method), a szorzó módszer (the multiplication method).

2.5. Nyílt címzés (open addressing)

Feltesszük, hogy az adatrekordok közvetlenül a résekben vannak. Az üres rés egy speciális rekordot tartalmaz, amit $\oslash$-lel jelölünk. Szükségünk lesz még egy másik speciális rekordra, amit az ún. *törölt* rések tartalmaznak, és amit $\otimes$-lel jelölünk. Az üres és a törölt réseket együtt *szabad* réseknek nevezzük. A többi rés *foglalt*. Feltesszük, hogy a szabad rések kulcs mezői nem elemei U-nak. Egyetlen hasító függvény helyett m darab hasító függvényünk van:

$$h(\cdot, i) : U \to 0..m-1 \qquad (i \in 0..m-1)$$

A beszúrásnál például először a $h(k, 0)$ réssel próbálkozunk. Ha ez foglalt, folytatjuk a $h(k, 1)$-gyel stb., míg szabad rést találunk, és ebbe tesszük az adatot, vagy kimerítjük az összes lehetséges próbát, de nem találunk szabad rést, és így sikertelen lesz a beszúrás. A $\langle h(k,0), h(k,1), \ldots, h(k,m-1)\rangle$ sorozatot ezért *próba sorozatnak* nevezzük. A próba sorozattal szemben megköveteljük, hogy a $\langle 0, 1, \ldots, m-1\rangle$ egy permutációja legyen, azaz, hogy az egész hasító táblát lefedje (és így ne hivatkozzon kétszer vagy többször ugyanarra a résre).

A keresésnél is a fenti próba sorozatot követjük, de itt átlépjük a törölt réseket, és csak akkor állunk meg, ha megtaláltuk a keresett kulcsú elemet (sikeres keresés), üres rést találunk vagy kimerítjük a próba sorozatot (sikertelen keresés). Ha a keresés a $h(k, i-1)$ próbánál áll meg, akkor (és csak akkor) a keresés hossza i.

A törlés egy sikeres keresést követően a megtalált i rés *törölt*-re állításából áll ($T[i] := \otimes$). Itt $T[i] := \oslash$ helytelen lenne, mert ha például feltesszük, hogy a k kulcsú adatot kulcsütközés miatt a $h(k, 1)$ helyre tettük, majd töröltük a $h(k, 0)$ helyen levő adatot, akkor egy ezt követő keresés nem találná meg a k kulcsú adatot.

Ha elég sokáig használunk egy nyílt címzésű hasító táblát, így elszaporodnak a törölt rések, és elfogynak az üres rések, holott a tábla esetleg közel sincs tele. Ez azt jelenti, hogy a sikertelen keresések az egész táblát végig fogják nézni. Ez ellen a tábla időnkénti frissítésével védekezhetünk, ami azt jelenti, hogy kimásoljuk egy temporális területre az adatokat, üresre inicializáljuk a táblát, majd a kimentett adatokat egyesével újra beszúrjuk.

Egy próba sorozat ideális esetben a $\langle 0, 1, \ldots, m-1\rangle$ sorozatnak mind az $m!$ permutációját azonos valószínűséggel állítja elő. Ilyenkor *egyenletes hasításról* beszélünk.

Amennyiben a táblában nincsenek törölt rések, egyenletes hasítást és a hasító tábla $0 < \alpha < 1$ kitöltöttségét feltételezve egy sikertelen keresés illetve egy beszúrás várható hossza legfeljebb

$$\frac{1}{1-\alpha}$$

míg egy sikeres keresés várható hossza legfeljebb

$$\frac{1}{\alpha}\ln\frac{1}{1-\alpha}$$

Ez azt jelenti, hogy egyenletes hasítást feltételezve pl. 50%-os kitöltöttség mellett egy sikertelen keresés illetve egy beszúrás várható hossza legfeljebb 2, míg egy sikeres keresésé kisebb, mint 1,387; 90%-os kitöltöttség mellett pedig egy sikertelen keresés illetve egy beszúrás várható hossza legfeljebb 10, míg egy sikeres keresésé kisebb, mint 2,559 [2].

2.5.1. Lineáris próba

$$h(k, i) = (h'(k) + i) \mod m \qquad (i \in 0..m-1)$$

ahol $h' : U \to 0..m-1$ hasító függvény. Könnyű implementálni, de összesen csak m különböző próba sorozat van, az egyenletes hasításhoz szükséges $m!$ próba sorozathoz képest, hiszen ha két kulcsra $h(k_1, 0) = h(k_2, 0)$, akkor az egész próba sorozatuk megegyezik. Ráadásul a különböző próba sorozatok összekapcsolódásával foglalt rések hosszú, összefüggő sorozatai alakulhatnak ki, megnövelve a várható keresési időt. Ezt a jelenséget *elsődleges csomósodásnak* nevezzük. Minél hoszabb egy ilyen „csomó”, annál valószínűbb, hogy a következő beszúráskor a hossza tovább fog növekedni. Ha pl. egy szabad rés előtt (ciklikusan értve) i foglalt rés van, akkor $(i+1)/m$ a valószínűsége, hogy a következő beszúrásnál ez is foglalttá válik. Ez az egyszerű módszer csak akkor használható, ha a kulcsütközés valószínűsége elenyészően kicsi.

2.5.2. Négyzetes próba

$$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \mod m \qquad (i \in 0..m-1)$$

ahol $h' : U \to 0..m-1$ hasító függvény, $c_1, c_2 > 0$. A különböző próba sorozatok nem kapcsolódnak össze, de itt is csak m különböző próba sorozat van, az egyenletes hasításhoz szükséges $m!$ próba sorozathoz képest, hiszen ha két kulcsra $h(k_1, 0) = h(k_2, 0)$, akkor az egész próba sorozatuk itt is megegyezik. Ezt a jelenséget *másodlagos csomósodásnak* nevezzük.

Annak érdekében, hogy a próba sorozat az egész táblát lefedje, a c_1, c_2 konstansokat körültekintően kell megválasztani. Ha például a tábla m mérete kettő hatvány, akkor $c_1 = c_2 = 1/2$ jó választás. Ráadásul ilyenkor

$$h(k,i) = \left(h'(k) + \frac{i+i^2}{2}\right) \mod m \qquad (i \in 0..m-1)$$

Ezért

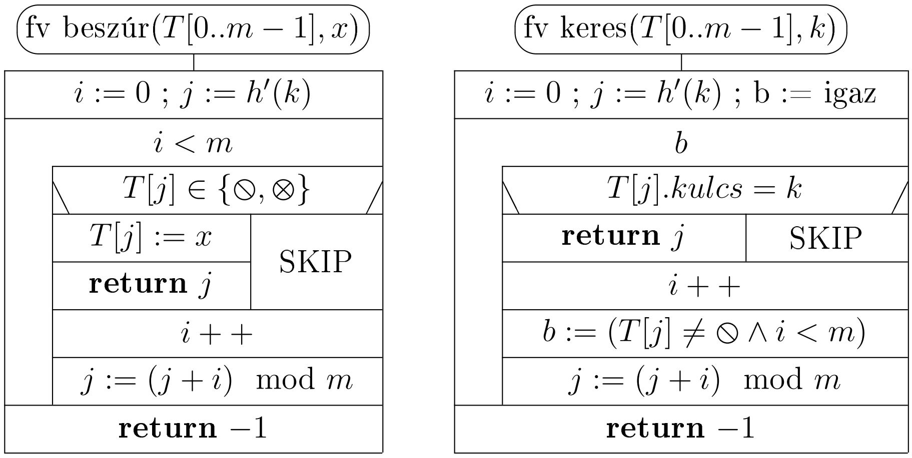
$$(h(k,i+1) - h(k,i)) \mod m = \left(\frac{(i+1)+(i+1)^2}{2} - \frac{i+i^2}{2}\right) \mod m =$$

$$(i+1) \mod m$$

azaz

$$h(k,i+1) = (h(k,i) + i + 1) \mod m$$

Innét a beszúrás és a keresés programjai (x a beszúrandó adat, k a keresett kulcs; a sikertelen műveletet a „-1” visszadásával jelezzük):



HF: Alakítsuk át a beszúrás struktogramját úgy, hogy egyenlő kulcsú adatok felvitelét ne engedje!

2.5.3. Kettős hasítás

$$h(k,i) = (h_1(k) + ih_2(k)) \mod m \qquad (i \in 0..m-1)$$

ahol $h_1 : U \to 0..m-1$ és $h_2 : U \to 1..m-1$ hasító függvények. A próba sorozat pontosan akkor fedi le az egész hasító táblát, ha $h_2(k)$ és m relatív prímek. Ezt a legegyszerűbb úgy biztosítani, ha a m kettő hatvány és $h_2(k)$ minden lehetséges kulcsra páratlan szám, vagy m prímszám. Például ha m prímszám és m' kicsit kisebb (mondjuk $m' = m-1$ vagy $m' = m-2$) akkor

$$h_1(k) = k \mod m$$

$$h_2(k) = 1 + (k \mod m')$$

egy lehetséges választás.

A kettős hasításnál minden különböző $(h_1(k), h_2(k))$ pároshoz különböző próbasorozat tartozik. Ezért itt $\Theta(m^2)$ különböző próbasorozat lehetséges. A kettős hasítás, bár próbasorozatainak száma messze van az ideális $m!$ számú próbasorozattól, úgy tűnik, hogy jól közelíti annak működését.

3. Elemi gráf algoritmusok ([2] 22)

3.1. Gráf ábrázolások

3.2. A szélességi gráfkeresés

A legrövidebb utak. A szélességi gráfkeresés (BFS) algoritmusa és futási ideje. A szélességi feszítőfa. HF: A szélességi gráfkeresés megvalósítása a klasszikus gráf ábrázolások esetén; hatékonyság.

3.3. A mélységi gráfkeresés

A mélységi gráfkeresés (DFS). Mélységi feszítő erdő. A gráf csúcsainak szín és időpont címkéi. Az élek osztályozása. A mélységi gráfkeresés futási ideje.

3.3.1. A topologikus rendezés

HF: A mélységi gráfkeresés és a topologikus rendezés megvalósítása a klasszikus gráf ábrázolások esetén; hatékonyság.

3.3.2. Erősen összefüggő komponensek

4. Minimális feszítőfák ([2] 23)

A minimális feszítőfa (MST) fogalma.

4.1. Egy általános algoritmus

Vágás, vágást keresztező él, élhalmazt elkerülő vágás, könnyű él. Egy tétel a biztonságos élekről és a minimális feszítőfákról.

4.2. Prim algoritmusa

Prim algoritmusa, mint az általános algoritmus megvalósítása. A futási idő elemzése.

HF: A Prim algoritmus implementációja a két fő gráfábrázolás és a szükséges prioritásos sor különböző megvalósításai esetén.

4.3. Kruskal algoritmusa

Kruskal algoritmusa, mint az általános algoritmus megvalósítása. A futási idő elemzése.