Kőrösi Gábor

Algoritmusok és adatszerkezetek a gyakorlatban

Jelen tananyag a Szegedi Tudományegyetemen készült az Európai Unió támogatásával.

Projekt azonosító: EFOP-3.4.3-16-2016-00014

Szegedi Tudományegyetem
Cím: 6720 Szeged, Dugonics tér 13.
www.u-szeged.hu
www.szechenyi2020.hu

Hasító táblák

Összefoglaló

Az eddigiekben számos olyan gyakorlati alkalmazással találkozhatunk, melyeknek a Beszúr, Keres és Töröl műveleteket támogató adatszerkezetre volt szükségük. Ilyen volt például a láncolt lista vagy a keresőfa is. A hasító táblázat hatékony adatszerkezet ennek megvalósítására. Bár egy elem megkeresésének ideje a hasító táblázatban ugyanolyan hosszú lehet, mint egy láncolt lista esetében (O(n)), a gyakorlatban a hasítás nagyon jól működik. Ésszerű feltételek mellett egy elem megkeresésének várható ideje a hasító táblázatban O(1). A tananyag végén látni fogjuk, hogy csak és kizárólag a Beszúr, Keres és Töröl műveletekre van szükségünk, akkor a hasító táblák a leghatékonyabb adatszerkezet (ha egyéb műveletekre is szükség van, akkor viszont már nem).

Lecke fejezetei:

- Mi is az a hasító tábla? – Olvasó (25 perc)
- Ütközésfeloldás láncolással (chaining) – Olvasó (20 perc)
- Gyakorló feladatok – Gyakorlati (45 perc)

Téma típusa: Gyakorlati
Olvasási és gyakorlási idő: 90 perc

Mi is az a hasító tábla? *(25 perc)*

Bonyolult matematikai képletek helyett emlékezzünk vissza két keresési technikára.
Az elemeink tömbben vannak és nincs meghatározva a sorrendjük:

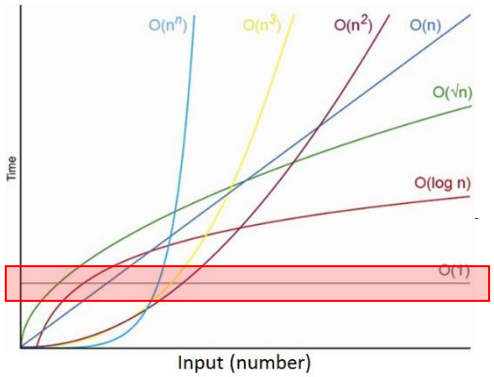
Lineáris keresés O(n):

8	5	12	6	15	9	4	3	7	10
0	1	2	3	4	5	6	7	8	9

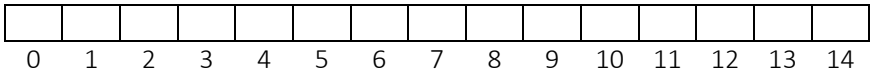
Bináris keresés O(log n):

3	4	5	6	7	8	9	10	12	15
0	1	2	3	4	5	6	7	8	9

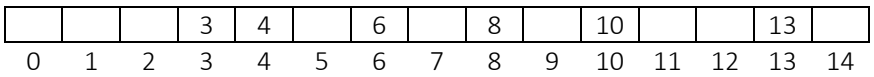
A Lineáris keresés nagyon időigényes, ezért célszerű elkerülni a használatát. Egy jó alternatíva a bináris keresés, ennél azonban időt veszítünk az elemek rendezésével. Célunk, hogy a egyre közelebb kerüljünk az O(1) időkomplexitáshoz, melyre a hasító tábla nyújt megoldást.



A példa kedvéért vegyünk egy táblázatot, melyben a 8, 3, 13, 6, 4, 10-es kulcsokat (értékeket) szeretnénk eltárolni.



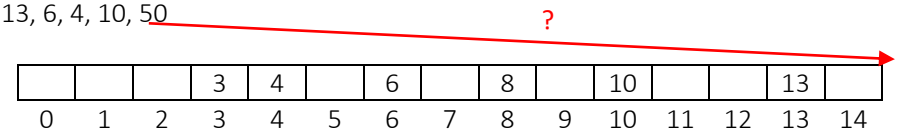
A létező legegyszerűbb és kézenfekvőbb megoldás, ha a kulcsokat az értékükkel megegyező indexen helyezzük el, hiszen így az értékeket eleve jó sorrendbe tároltuk el.



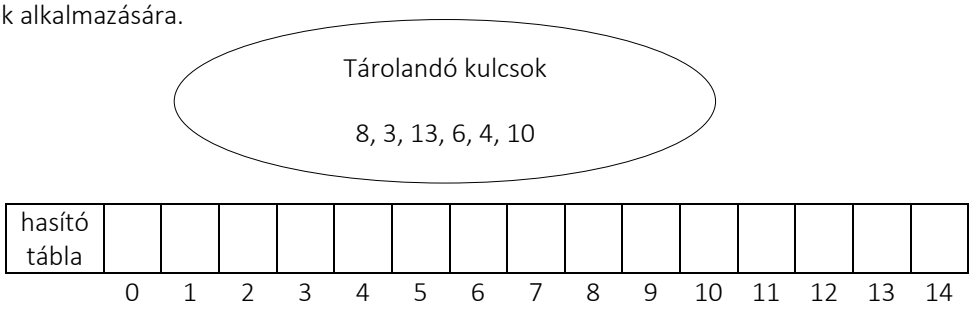
Az így eltárolt adatokat könnyen megtalálhatjuk a tömbünkben, és ha a kulcs benne van a táblában, akkor visszakapjuk annak értékét, egyébként NIL. Az ilyen módon eltárolt elemek elérése (keresése) O(1).

Azonban egy problémával szembesülünk: mi történik akkor, ha egy új elem értéke nagyobb a tömb méreténél?

Kulcsok: 8, 3, 13, 6, 4, 10, 50

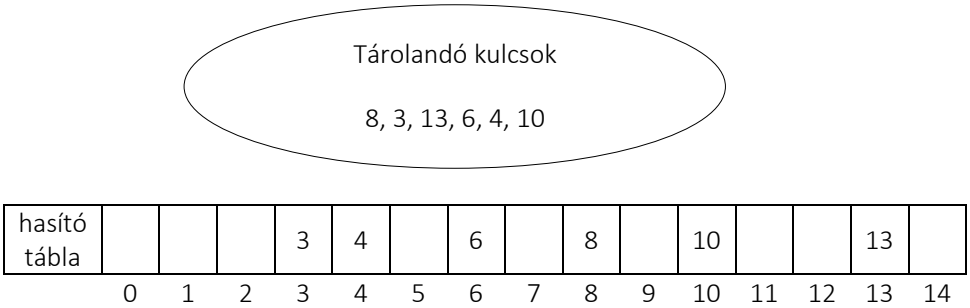


Egy megoldás lenne, ha megnövelnénk a tömb méretét, viszont emiatt sok felesleges helyet foglalnánk le. A probléma megoldására a HASÍTÓ függvények gondolatát vezettük be. Nézzünk egy példát a hasító függvények alkalmazására.



Egy nagyon „primitív megoldásban” a már előzőekben megismert, *ÉRTÉK == KULCS/INDEX* függvény is alkalmazható lenne.

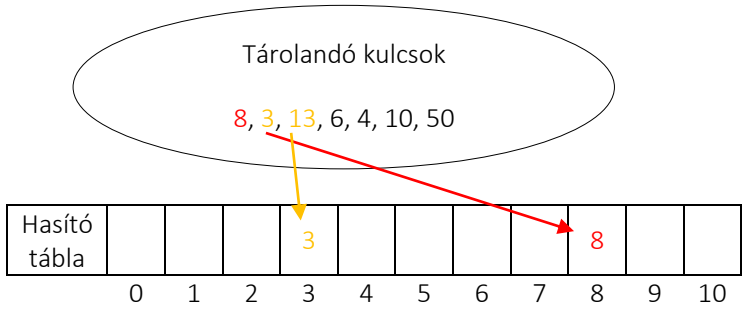
h(x) = x



Viszont, ha egy új 50-es értéket adunk a halmazunkhoz, akkor azzal nem tudunk mit kezdeni. Mielőtt továbbmennénk, le kell tisztáznunk, hogy van ONE-ONE és MANY-ONE típusú függvény. Az imént látott h(x) = x hasító függvény egy ONE-ONE megoldás. Ahhoz, hogy megoldjuk a problémát, át kell írnunk a módosító függvényt (h(x) = x) valami jobb megoldásra.

Módosítsuk úgy a feladat megoldását, hogy a h(x) = x -et cseréljük fel a h(x) = x%10 formulára, ahol a 10 a tárolótömb méretét hivatott ábrázolni. Teszteljük a megoldást. Néhány szám erejéig jól működik az elgondolás. A 8-as és a 3-as érték bekerülhet a tömbbe, ám a 13 modulusa 3, így ütközés áll fenn, mivel nem kerülhet két elem ugyanarra a helyre.

Many-One
h(x) = x%10



Mi ilyenkor a teendő? A hasító táblák alapja a „megfelelően” megválasztott hasító függvény. A rosszul megválasztott függvény kulcsütközéshez (collision) vezet. Gyakori hiba továbbá, a kulcsok csoportosulása (clustering). Ebben az esetben annak a valószínűsége, hogy több kulcs ugyanazt az indexet adja, jóval nagyobb, mintha egy véletlenszám-generátor függvényt használtunk volna.
A leírtak nem azt jelentik, hogy az ütközés miatt ki kell dobnunk az iménti h(x) = x%10 módosító függvényünket, ugyanis az ütközés feloldására létezik megoldás.

Ütközésfeloldásra a következő megoldásokat használjuk:

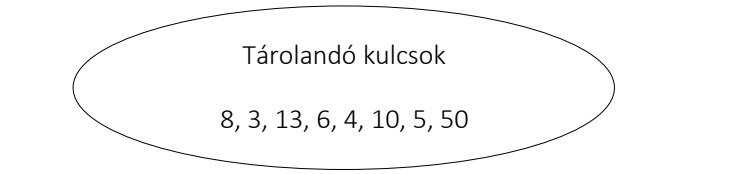
- Zárt hasítás
  - Láncolás (chaining)

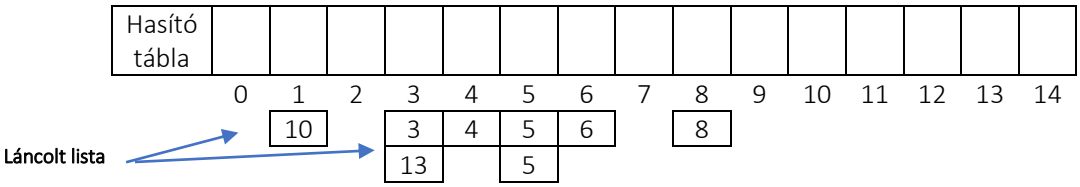
A zárt hasítás esetében a kulcsok létrehozásánál ugyanazt a függvényt használjuk, mint amit a láncolt listáknál ismertünk meg.

Ütközésfeloldás láncolással (chaining) (20 perc)

A zárt hasítás láncolt listával a következőképpen orvosolja az ütközés problémáját:

Many-One
h(x) = x%10





Nézzük ezt meg egy Python példán keresztül:

```
# Node osztaly
class Node:
    # a node objektum inicializalasa
    def __init__(self, data):
        self.data = data  # a lista elem erteke
        self.next = None  # a lista elem kovetkezo erteke
class LinkedList:
    def __init__(self):
        self.head = None

def append(self, uj_adat):
    # 1-2. lepes: Hozzuk letre az uj valtozot. es tegyuk bele az uj adatot
    uj_node = Node(uj_adat)
    # 3. lepes: Ha a lista ures, akkor az uj elemunk legyen az uj head
    if self.head is None:
        self.head = uj_node
        return
```

```
    # 4. lepes: kulonben menjunk el a lista vegere
    utolso= self.head
    while (utolso.next):
        utolso = utolso.next
    # 5. lepes: az utolso elem kovetkezoje az uj elem legyen
    utolso.next =  uj_node

def printHashtable(self):
        # 1. lepes: eltaroljuk a lista fejet
        for i in range(0,10):
            if (self[i]!=None):
                temp = self[i].head
                # 2. lepes: ha a lista feje tartalmazza az erteket, akkor toroljuk ki
                while (temp is not None):
                    print("--"+ str(temp.data),end ="")
                    temp = temp.next
                print(),
        return

def insert(self, keyvalue, uj_adat):
    append(self[keyvalue], uj_adat)

# ures hasito szotar letrehozasa
Hashtable = {0:LinkedList(),
             1:LinkedList(),
             2:LinkedList(),
             3:LinkedList(),
             4:LinkedList(),
             5:LinkedList(),
             6:LinkedList(),
             7:LinkedList(),
             8:LinkedList(),
             9:LinkedList()
            }
arr = [8,3,13,6,4,10,5,50]
size = 10
for i in range(0,8):
    h = arr[i]%size
    insert(Hashtable, h, arr[i])

printHashtable(Hashtable)
```

```
Futasi eredmeny:
--10--50


--3--13
--4
--5
--6

--8
```

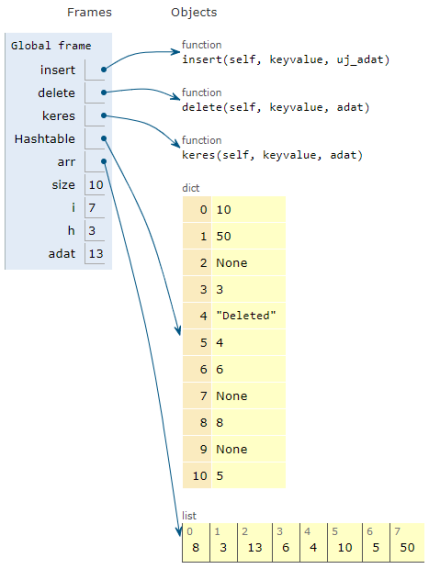
Ha szeretnénk megkeresni egy értéket a hasító táblában (pl.: key = 13), akkor ehhez használnunk kell a h(x) = x%size hasító függvényt. Ebben az esetben a harmadik elem láncában kell keresni az elemet, így a megoldás időkomplexitása nem O(1) , de jobb, mint O(log n).

```
def kesesHash(self,h,key):
    temp = self[h].head
    while (temp is not None):
        if(temp.data==key):
            print(temp.data)
            temp=temp.next
        else: temp=temp.next
    return
```

```
size=10
key=13
h=key%size
kesesHash(Hashtable,h,key)
```

Gyakorló feladatok *(45 perc)*

1. Szemléltessük, hogy a hasítótábla Beszúrás, Törlés, Keres függvényei milyen időkomplexitással dolgoznak a különböző méretű tömbökön.
2. Adott egy nyílt címzéses, m = 11 hosszúságú (üres) hasító táblázat a h(k) = k mod m elsődleges hasító függvénnyel, és a következő beszúrandó kulcsok: 10, 22, 31, 4, 15, 28, 17, 88, 59.
   Szemléltessük a beszúrás eredményét akkor, ha a lineáris kipróbálást, a c1 = 1 és c2 = 3 állandókat használó négyzetes kipróbálást, illetve a h2(k) = 1 + (k mod (m − 1)) másodlagos hasító függvényű dupla hasítási módszert alkalmazzuk.
3. Szemléltessük a mintafeladatainkat a http://pythontutor.com/live.html oldal segítségével.



4. Készítsünk egy programot mely keresőfában és hasítótáblában tárolt adatokat ír ki rendezetten. Vizualizáljuk a megoldáshoz felhasznált lépések számát.
5. Készítsünk egy programot mely keresőfában és hasítótáblában tárolt adathalmazban keres meg egy (vagy több) elemet. Vizualizáljuk a megoldáshoz felhasznált lépések számát.

Ajánlott kitekintő anyag *(99 perc)*

Halmaz és Szótár **(18 perc)** – Videó link magyar nyelven

Hasító táblák **(12 perc)** – Videó link magyar nyelven

Ütközésfeloldás láncolással **(9 perc)** – Videó link magyar nyelven

Hasító függvények **(9 perc)** – Videó link magyar nyelven

Hasító tábla implementációk **(7 perc)** – Videó link magyar nyelven

Bináris keresőfa vs hasító tábla **(8 perc)** – Videó link magyar nyelven

Hasítás a kriptográfiában **(4 perc)** – Videó link magyar nyelven

Hasító táblák **(7 perc)** – YouTube link

Hasító táblák **(5 perc)** – link

Algoritmusok és adatszerkezetek gyakorlat - Hasító táblák **(20 perc)** - Gelle Kitti - link

SZÉCHENYI 2020
MAGYARORSZÁG KORMÁNYA
Európai Unió
Európai Szociális Alap
BEFEKTETÉS A JÖVŐBE