Hash Tablosu

Barış Şimşek
simsek ~ enderunix . org
http://www.enderunix.org/simsek/

Pek çok uygulama sözlük tarzı EKLEME, ÇIKARTMA ve ARAMA gibi işlemleri gerçekleştirecek veri yapılarına ihtiyaç duyarlar. Örneğin bir derleyici, program içerisindeki tanımlayıcıları (identifier) bir tabloda tutarak yönetir.

Hash tablosu, veriye bir *anahtar (key)* yardımı ile erişilen basit bir dizi üzerine bina edilmiştir. Anahtar kullanılarak bir indeks üretilir ve bu indeks ile dizideki istenen veriye ulaşılır. Anahtar tekildir yani bir başka kayıtta aynı anahtar olamaz. Ancak veri aynı olabilir. Bir sınıftaki öğrenciler buna çok iyi bir örnektir. Her öğrencinin sadece kendine ait bir numarası vardır. Ancak aynı isme sahip iki öğrenci olması muhtemeldir. Burada öğrenci numarası anahtardır; öğrenci ismi ise, bu anahtara ait veridir.

h(k)	T[h(k)]
0	Buse
1	Ahmet
2	Ali
3	
4	Orhan
5	Fatma
6	
7	Ayşe
8	
9	
10	
11	Ali

Şekil 1. Hash tablosu

Ahmet'in numarası 639, Ayşe'ninki ise 766 olsun. T, m = 11 eleman alabilecek bir hash tablosu olsun. Basitce aşağıdaki gibi bir hash fonksiyonu kullanalım:
h(k) = k mod 11

Ahmet için hash kodu = h(639) = 639 mod 11 = 1
Ayşe için hash kodu = h(766) = 766 mod 11 = 7

Bu durumda T[1] = "Ahmet" ve T[7] = "Ayşe" olacaktır.

Hash tablosu, en hızlı aramayı sunan veri yapılarından biridir. O(1) ile ifade edilebilecek sabit bir zaman diliminde arama yapılır. En kötü durumda (worst case) -ki aşağıda bahsedilen çakışma (collision) durumudur- bu değer O(n) olabilmektedir. Eğer hash fonksiyonu, iki farklı anahtar için aynı hash kodunu üretmiyorsa hash tablosu **doğrudan adreslidir** denir.

Hash tablosunun pratikte kullanılabileceği bir uygulama vekil (proxy) sunucularıdır. Daha önce ziyaret edilen sayfalar diske saklanır ve sonraki erişim istekleri yerel diskten karşılanır. Diskteki web içeriğine ulaşmak için vekil sunucu istenilen adresi anahtar(key) olarak kullanarak bir hash değeri üretir. Ve bu hash değeri ile istenen sayfaya ulaşılır. Doğru veriye ulaşmak için, içerik daha önceden aynı fonksiyonun ürettiği indekse saklanmış olmalıdır.

Hash Fonksiyonu

Hash fonksiyonu, bir anahtar (key) kullanarak tablo üzerinden bir pozisyon (index) üretir. Verilerin tablodaki yerlerini hash fonksiyonu belirler. Hash fonksiyonu girdi olarak anahtarı (key) kullanır ve çıktı olarak *hash kod (hash coding)* veya *hash değeri* üretir.

h(k): Hash fonksiyonu.
k: Key, hash edilecek anahtar kelime.
v: Value, tabloda saklanan veri
x: h(k) ile üretilen hash kodu.
T: Hash tablosu.
M: Hash tablosundaki slot sayısı (T bir dizi olmak üzere, dizi boyu)

hash(k) = x ($0 \leq h(k) < m$)

Bu formülde x'e, k'nın hash kodu denilir.

Veri tabloya T[x] = v şeklinde yerleştirilebilir.

Pek çok hash fonksiyonu anahtar kümesinin N = {0, 1, 2, ...} doğal sayılardan oluştuğunu varsayar. Eğer anahtar kelimeler doğal sayı değilse, onları doğal sayı gibi yönetecek bir mekanizma bulunmalı. Karakterlerin ASCII değerleri kullanılabilir. String'lerden hash değeri üretilirken stringin belirli bir kısmını kullanmak çakışmaları artıracaktır. Örneğin yukarıda bahsedilen vekil sunucu uygulamasında anahtar olarak kullanılan adresler (url) http://www. şeklinde başlayıp, .php, .html, .jsp şeklinde bitmektedirler. Bu durumda farkı oluşturacak olan kısım ortadaki birkaç karakterdir. Eğer bunlar göz önünde bulundurulmaz ise çakışmalar artacağından belirli adresler için bağlı listeler uzayıp gidecektir. İşte bu nedenle karakter dizisinin her bir karakteri hash değerinin hesaplanmasında kullanılmalıdır.

8 karakterli kullanıcı adlarından hash değeri üretilecekse aşağıdaki fonksiyon iyi bir seçim değildir. Çünkü bütün karakterleri kullanmamaktadır:

```
int hash( char *key, unsigned int TBL_SIZE )
{
     return ( ( key[0] + 13*key[1] + 29*key[2] ) % TBL_SIZE );
}
```

Aşağıda D. J. Bernstein tarafından CDB veritabanında kullanılan hash fonksiyon verilmiştir. Bütün karakterleri gözönünde bulundurduğu için iyi sonuçlar vermektedir.

```
#define CDB_HASHSTART 5381
#define TBL_SIZE 10000

unsigned int cdb_hashadd(unsigned int h,unsigned char c)
{
  h += (h << 5);
  return h ^ c;
}

unsigned int cdb_hash(char *buf,unsigned int len)
{
  unsigned int h;

  h = CDB_HASHSTART;
  while (len) {
    h = cdb_hashadd(h,*buf++);
    --len;
  }
  return h % TBL_SIZE;
}
```

Eğer k, N = {0, 1, 2, ...} doğal sayı kümesinden bir anahtar ise, bundan bir hash değeri üretmek için birkaç yöntem vardır.

Bölme Yöntemi

Bölme yönteminde k anahtar kelimesini m adet slottan birisine eşlerken k'nın m'ye bölümünden kalan değer kullanılır.

hash(k) = k mod m

Bu fonksiyon yalnızca bir bölme işlemi gerektirdiğinden oldukça hızlıdır.

Bölme yöntemi kullanılıyorsa m, 2'nin kuvvetlerinden biri olmamalıdır. Eğer m = 2 ^ p seçilirse, h(k) değeri k'nın en düşük p bitinden oluşur. Daha az çakışma için k'nın bütün bitlerini kullanmak gerekir. Eğer anahtar 10'lu sistemde ise yine aynı sebeple 10'un kuvvetlerini kullanmaktan da kaçınılmalıdır. Elemanları 10'un katları olan U: {10, 100, 230, 420, 450, 890} kümesi için eğer m = 10 seçilirse bütün anahtarlar için h(k) = 0 olacaktır. Aynı şekilde 3'ün katları da çakışma olasılığını artıracaktır. Çünkü:

(10^n mod 3) = (4^n mod 3) = 1

M için en iyi değer 2'nin kuvvetlerine yakın olmayan asal sayılardır. Örneğin n=1500 eleman için m=701 (2^9 ve 2^10 arasında, bunlara yakın olmayan bir değer) kullanılabilir. Bu durumda hash fonksiyonu:

h(k) = k mod 701

Çarpma Yöntemi

Bu yöntem iki aşamalıdır. Önce k, 0 < A < 1 olmak üzere sabit A katsayısı ile çarpılır. Sonucun kesirli kısmı ayrılır ve m ile çarpılarak sonuç elde edilir.

Bu yöntemin en önemli avantajı, slot sayısı m'nin değerinin kritik olmamasıdır.

Çarpma yönteminde A sabitinin ne olduğu önemli olmamakla birlikte bazı değerler diğerlerine göre daha iyi sonuç vermektedir. David E. Knuth aşağıdaki değerin daha iyi sonuçlar ürettiğini gösteren çalışmalar yapmıştır:

A ≈ ((Kare Kök 5) –1) / 2 = 0,6180339887498948482045868343656 4...

Örneğin m = 2000, k = 23456

h(k) = h(23456) = 2000 * Kesirli_Kısım(0.618033988 * 23456)
h(23456) = 2000 * 0,067977499789696409173668731 3 = 136

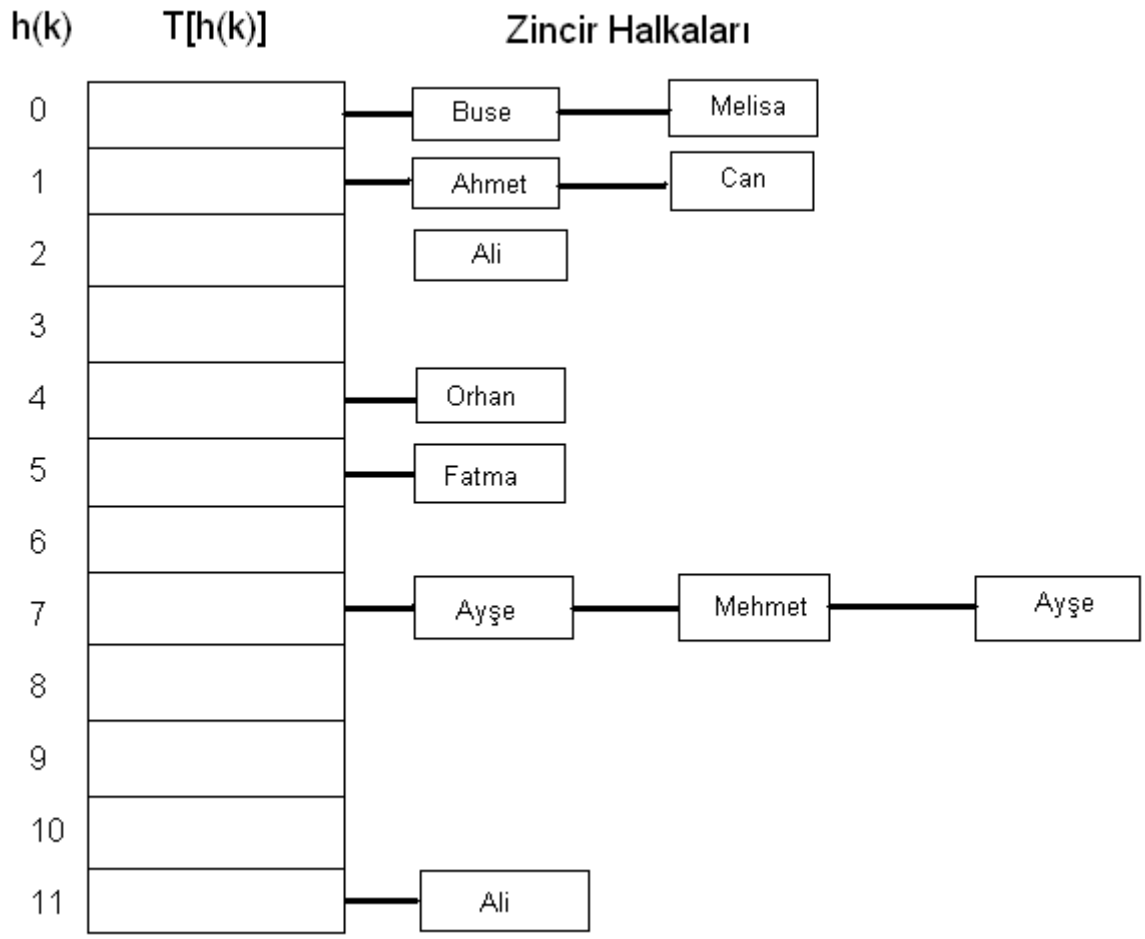
Hash fonksiyonunun farklı anahtarlar için aynı değeri (indeks) üretmesine **çakışma (collision)** denir ve iyi bir hash fonksiyonundan en az çakışmaya neden olması beklenir. İdealde çakışma olmaması hedeflense de teoride bu oldukça zordur. Bu nedenle çakışmanın olacağı varsayılarak alternatif veri yapıları kullanılır.

Çakışma olmasını engellemek için iyi bir hash fonksiyonu seçilmesi yeterli değildir. Hash fonksiyonunun üreteceği tüm hash değerleri için dizide bir indeks numarası ayrılması gerekir. Örneğin 8 haneli kullanıcı adları kullanılarak bir veri tabanındaki hesap bilgilerine ulaşılmak istensin. Kullanıcı adındaki her bir hanenin 26 latin harften biri olabileceği varsayılırsa 26^8 yani yaklaşık 209 milyar farklı kullanıcı adı olabilecektir. Bu kadar büyük bir tablo ayrılırsa ve doğrudan adresleme yapılırsa çakışma ihtimali ortadan kalkacaktır. Ancak sistemde 1000 civarında kullanıcı varsa, bu kadar yer ayırmak büyük bir bellek israfına neden olacaktır. Bu nedenle programcı her durumda çakışma ile uğraşmak zorundadır.

Zincir Hash Tabloları

Çakışmaya karşı en çok kullanılan tekniklerden birisi *zincir (chained) hash* tablolarıdır. Zincir hash tabloları, bağlı listelerin bir dizisi şeklindedir. Dizinin her bir elemanı **buket (bucket)** veya **zincir** olarak adlandırılır ve çakışan verileri bağlı liste olarak tutar. Veri eklemek için önce anahtar ve hash fonksiyon kullanılarak bir indeks üretilir. Bu indeks ilgili buketin seçilmesini sağlar. Bundan sonra bağlı listeye veri eklenir. Veri arandığı zaman önce anahtar ile indeks üretilir ve buket bulunur. Sonra bağlı liste üzerinde yürünerek istenen veri için bakılır.

Çakışan veriler dinamik olarak büyüyebilen bağlı listeye yerleştirilir. Ancak dizi doldukça çakışmalar artacak ve bağlı listeler uzayacaktır. Bu ise arama zamanını uzatacaktır.



Zincir hash tablosu C ile aşağıdaki gibi tanımlanabilir:

```
#define TBL_SIZE 1577

typedef struct HItem HItem;
struct HItem {
      int key;
      char *data;
};

HItem *HTable[TBL_SIZE];
```

Hash fonksiyonu belirli buketlerde yığılmaya neden olabilir. Bu durumda, yığılma olan buketler üzerinde arama yapmak zorlaşacaktır. İdeal olarak buketlerin eşit uzunlukta olması istenir. Yani hash fonksiyonu her bir buket için eşit sayıda çakışma üretir. Buna **eşit dağıtılmış (uniform) hash** denir. Pratikte bu çok zor olduğundan buketlerin yaklaşık eşit uzunlukta olması da kabul edilir.

Dağıtık hash için önemli karakteristiklerden birisi, hash tablosunun *yük (load)* faktörüdür.

Yük faktörü $\alpha = n / m$

n: Kümedeki eleman sayısı
m: Tablodaki slot sayısı.

Yük faktörü, dağıtık hash için her bir zincirde oluşması gereken çakışma miktarını gösterir. Eğer tablo 2000 elemanlı ise ve tabloda 700 pozisyon var ise yük faktörü 2000 / 700 yani yaklaşık 3'tür. Dağıtık hash için her bir bukette 2 veya 3 çakışmış veri bulunmalıdır.

Eşit dağılım (uniform hash) için, tablo boyutu olarak asal sayı tercih edilmelidir. Böylece tablo boyuna bağlı yığılmaların önüne geçilmiş olunur.

En kötü (worst-case) bütün elemanlar aynı zincirde (yani hepsi çakışmış) yer alır ve n elemanlı bir bağlı liste oluşturur. Bu durumda arama işlemi O(n) zaman gerektirir. Bağlı listelerin sıralı olmadığı unutulmamalıdır.

Hash değeri h(k) 'nın üretilmesinin O(1) zaman gerektirdiği varsayılabilir.

Açık Adresleme

Açık adreslemede, bütün elemanlar tabloda kendi başlarına yer alır. Yani tablodaki her slotta bir eleman vardır veya boştur (NULL). Bir eleman aranırken istenen eleman bulunana kadar belirlenen yöntem ile sınanır veya boş bir slota denk gelindiğinde çıkar. Çünkü boş slottan sonra aramaya gerek yoktur. Çünkü tabloda böyle bir eleman yoktur. Eğer olacak olsaydı bu boş slota yerleşmiş olması lazımdı. Zincir hash tablolarında olduğu gibi, çakışan elemanlar bir zincirin ucuna eklenmezler. Kendilerine müstakil bir yer buluncaya kadar devam edilir.

Açık adreslemede yük faktörü tablo dolduğunda 1; diğer durumlarda ise 1'den küçüktür. Çünkü açık adreslemede her zaman $n \leq m$ 'dir.

Bir elemanı tabloya yerleştirmek için *sınama (probe)* yapılır. Boş bir slot bulununcaya kadar bu araştırma devam eder. Boş slotun bulunması 0, 1, ..., m –1 e yapılan sıralı bir işlem değildir. Hangi slotun sınanacağına anahtara bakılarak karar verilir. Hash fonksiyonu parametre açık adresleme yapabilmek için anahtar kelimeyi ve 0'dan başlayıp artan *sınama sıra numarasını (probe sequence number)* parametre alacak şekilde düzenlenir.

$h(k, i) = x$

k: Anahtar
i: Kaçıncı sınama olduğu. i: $0 \rightarrow$ m-1

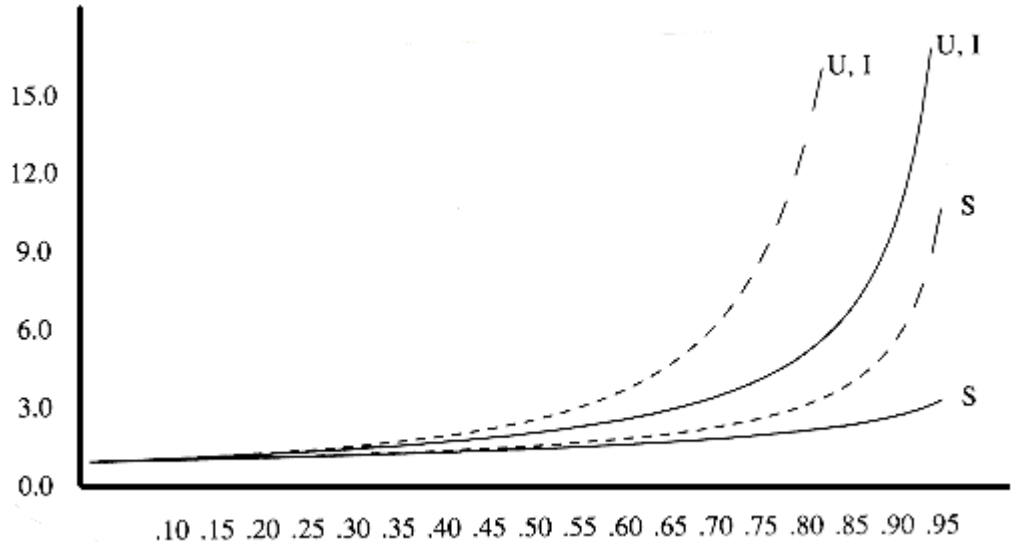
Beklenen sınama sayısı = $1 / (1 - \alpha)$

Örneğin yarı dolu bir tablo için

$\alpha = n / m = 0.5$

Beklenen sınama sayısı = $1 / (1 – 0.5) = 2$ (Uniform hash olduğu varsayılmaktadır.)

α büyüdükçe (1'e yaklaştıkça) tablonun doluluğu artacağından beklenen sınama sayısının da artması kaçınılmaz olmaktadır. İşlemci zamanın önemli olduğu süreçlerde tablo boyunu (m) büyütmek ilave boşluklar açacağından faydalı olacaktır. Aşağıda eşit dağılmış bir hash için yük faktörüne bağlı olarak beklenen sınama sayısının nasıl arttığı gözükmektedir.



Zincir hash tablosunda dağınık hash iyi bir hash fonksiyonu seçmeye bağlı idi. Açık adreslemede ise çakışma durumunda bir sonraki aşamada hangi pozisyonun sınacağını belirleyen algoritma da önemlidir. Sınama politikasını belirleyen değişik yöntemler geliştirilmiştir.

Doğrusal (Linear) Sınama

h', U → {0, 1, ..., m-1} kümesinde tanımlı sıradan bir yardımcı hash fonksiyonu olmak üzere doğrusal sınama yapan hash fonksiyonu şu şekilde tanımlanır:

$h(k,i) = (h'(k) + i) \bmod m$ (i : 0,1...m-1)

İlk önce T[h'(k)], sonra T[h'(k) + 1] sınanacaktır. Eğer hala boş slot bulunamamış ise en son olarak i = m-1 için sınama yapılacaktır.

Yardımcı hash fonksiyon olarak yukarıda bahsedilen bölme veya çarpma yöntemli hash fonksiyonlarından birisi seçilebilir.

Örnek: h'(k) bölme yöntemini kullanıyor.

h'(k) = k mod m
m = 701

20000 öğrencinin olduğu bir okulda k = 1725 numaralı öğrencinin bilgileri veritabanına yerleştirilmek istensin.

h’(1725) = 1725 mod 701 = 323 (1)

h(k, 0) = (323 + 0) mod 701 = 323 (2)

h(k, 1) = (323 + 1) mod 701 = 324 (3)

Yardımcı hash fonksiyonu 323 değerini vermiştir(1). Bunun neticesi olarak T[323] ‘ün sınanmasına karar verilir(2). Eğer burası da dolu ise T[324] sınanır. Bu şekilde doğrusal olarak devam edilir.

Doğrusal sınama, basit olması ve bütün slotları sınaması nedeni ile tercih edilebilir. Ancak sıralı bakılması nedeniyle zamanla dolu slotlar birleşerek büyük zincirler oluşturmakta ve boş slot bulma olasılığı azalmaktadır. Bu problem **birincil kümelenme (primary clustering)** olarak adlandırılır.

Dörtlü Sınama

Dörtlü sınama aşağıdaki biçimde bir hash fonksiyonu kullanır:

$h(k,i) = (h'(k) + c_1 i + c_2 i^2) \bmod m$

h’, doğrusal sınamada olduğu gibi yardımcı hash fonksiyonudur. $c_1 \neq 0$ ve $c_2 \neq 0$ olmak üzere yardımcı sabitlerdir. Bu şartlar altında ilk önce T[h’(k)] sınanır.

Dörtlü sınama, doğrusal sınamaya göre daha iyi sonuçlar verse de eğer h(k1, 0) = h(k2, 0) (yani iki farklı anahtarın ilk sınama pozisyonu aynı) ise h(k1, 1) = h(k2, 1). Bir sonraki sınama pozisyonları da aynı olacaktır. Doğrusal sınamadaki *birincil kümelenmenin* daha hafifi olan bu problem **ikincil kümelenme (secondary clustering)** olarak adlandırılır.

İkili Hash

İki hash fonksiyonu ile daha çok karakteristiğin birlikteliğini kullanan *ikili hash*, açık adresleme için en iyi çözümlerden biridir. Aşağıdaki biçimdedir:

$h(k, i) = (h_1(k) + i h_2(k)) \bmod m$

Burada h1 ve h2 yardımcı hash fonksiyonlardır. İlk sınanacak pozisyon T[h1(k)] ‘dır.

İkili hash kullanılırken m, 2’nin kuvvetlerinden biri ve h2 daima tek sayı üretecek şekilde olmalıdır. Veya m asal sayı seçilir, ancak h2 daima m’den küçük pozitif tamsayı verir. Örneğin m bir asal sayı seçilmiş ise $m' < m$ olmak (m’e yakın ir değer, genelde m – 2) üzere hash fonksiyonları şu şekilde seçilebilir.

$h_1(k) = k \bmod m$
$h_2(k) = k \bmod m'$

İkili hash'de ilk sınama pozisyonu iki hash fonksiyonu tarafından belirlendiği için kümelenme sorunu yaşanmaz. Bu nedenle sınama başlangıç pozisyonu önemli değildir.

İkili hash, doğrusal sınama ve dörtlü sınamaya göre oldukça iyi sonuçlar vermektedir. Bu nedenle dağınık (uniform) hash için tercih edilebilir.

Referanslar:

David E. Knuth, The Art of Computer Programming, Vol 3: Sorting and Searching, second printing, Addison-Wesley, 1975.

Algorithms in C, Robert Sedgewick, Addison-Wesley, ISBN: 0-201-51425-7

Code Reading: The Open Source Perspective, Diomidis Spinellis, Addison Wesley, ISBN: 0-201-79940-5

The Practice of Programming, Brian W. Kernighan, Rob Pike, ISBN: 0-201-61586-X

Barış Şimşek
simsek ~ enderunix . org
Ocak 2006, Beylerbeyi / İstanbul
http://www.enderunix.org/simsek/