제8장 Hashing

경북대학교 임경식 교수

8.1 Introduction

Examples of dictionaries are found in many applications, including the spelling checker, the thesaurus, the index for a database, and the symbol tables generated by loaders, assemblers, and compilers. When a dictionary with *n* entries is represented as a binary search tree as in Chapter 5, the dictionary operations *search, insert* and *delete* take *O(n)* time. These dictionary operations may be performed in O(log n) time using a balanced binary search tree (Chapter 10). In this chapter, we examine a technique, called hashing, that enables us to perform the dictionary operations *search, insert* and *delete* in 0(1) expected time. We divide our discussion of hashing into two parts: *static hashing* and *dynamic hashing.*

8.2 Static Hashing – hash table

In *static hashing* the dictionary pairs are stored in a table, *ht*, called the *hash table*. The hash table is partitioned into b buckets, $ht[0], \cdots, ht[b-1]$. Each bucket is capable of holding s dictionary pairs (or pointers to this many pairs). Thus, a bucket is said to consist of s slots, each slot being large enough to hold one dictionary pair. Usually $s = 1$, and each bucket can hold exactly one pair. The address or location of a pair whose key is k is determined by a hash function, h, which maps keys into buckets. Thus, for any key k, $h(k)$ is an integer in the range 0 through $b-1$. $h(k)$ is the hash or home address of k. Under ideal conditions, dictionary pairs are stored in their home buckets.

Definition: The *key density* of a hash table is the ratio n/T, where n is the number of pairs in the table and T is the total number of possible keys. The *loading density* or *loading factor* of a hash table is $\alpha = n/(sb)$. □

Suppose our keys are at most six characters long, where a character may be a decimal digit or an uppercase letter, and that the first character is a letter. Then the number of possible keys is $T = \sum_{0 \le i \le 5} 26 \times 36^i > 1.6 \times 10^9$. Any reasonable application, however, uses only a very small fraction of these. So, the key density, n/T, is usually very small. Consequently, the number of buckets, b, which is usually of the same magnitude as the number of keys, in the hash table is also much less than T. Therefore, the hash function h maps several different keys into the same bucket. Two keys, k_1, and k_2, are said to be *synonyms* with respect to h if $h(k_1) = h(k_2)$.

8.2 Static Hashing – hash table

As indicated earlier, under ideal conditions, dictionary pairs are stored in their home buckets. Since many keys typically have the same home bucket, it is possible that the home bucket for a new dictionary pair is full at the time we wish to insert this pair into the dictionary. When this situation arises, we say that an *overflow* has occurred. A *collision* occurs when the home bucket for the new pair is not empty at the time of insertion. When each bucket has 1 slot (i.e., $s = 1$), collisions and overflows occur simultaneously.

8.2 Static Hashing – hash table

Example 8.1: Consider the hash table *ht* with $b = 26$ buckets and $s = 2$. We have $n = 10$ distinct identifiers, each representing a C library function. This table has a loading factor, α, of $10/52 = 0.19$. The hash function must map each of the possible identifiers onto one of the numbers, 0–25. We can construct a fairly simple hash function by associating the letters, a–z, with the numbers, 0–25, respectively, and then defining the hash function, $f(x)$, as the first character of x. Using this scheme, the library functions **acos**, **define**, **float**, **exp**, **char**, **atan**, **ceil**, **floor**, **clock**, and **ctime** hash into buckets 0, 3, 5, 4, 2, 0, 2, 5, 2, and 2, respectively. Figure 8.1 shows the first 8 identifiers entered into the hash table.

	Slot 0	Slot 1
0	**acos**	**atan**
1		
2	**char**	**ceil**
3	**define**	
4	**exp**	
5	**float**	**floor**
6		
...		
25		

The next identifier, clock, hashes into the bucket *ht* [2]. Since this bucket is full, we have an overflow. Where in the table should we place clock so that we may retrieve it when necessary? We consider various solutions to the overflow problem

8.2 Static Hashing - – hash table

When no overflows occur, the time required to insert, delete or search using hashing depends only on the time required to compute the hash function and the time to search one bucket. Hence, the insert, delete and search times are independent of n, the number of entries in the dictionary. Since the bucket size, s, is usually small (for internal-memory tables s is usually 1) the search within a bucket is carried out using a sequential search.

The hash function of Example 8.1 is not well suited for most practical applications because of the very large number of collisions and resulting overflows that occur. This is so because it is not unusual to find dictionaries in which many of the keys begin with the same letter. Ideally, we would like to choose a hash function that is both easy to compute and results in very few collisions. Since the ratio b/T is usually very small, it is impossible to avoid collisions altogether.

In summary, hashing schemes use a hash function to map keys into hash-table buckets. It is desirable to use a hash function that is both easy to compute and minimizes the number of collisions. Since the size of the key space is usually several orders of magnitude larger than the number of buckets and since the number of slots in a bucket is small, overflows necessarily occur. Hence, a mechanism to handle overflows is needed.

8.2 Static Hashing – hash fts

A hash function maps a key into a bucket in the hash table. As mentioned earlier, the desired properties of such a function are that it be easy to compute and that it minimize the number of collisions. In addition, we would like the hash function to be such that it does not result in a biased use of the hash table for random inputs; that is, if k is a key chosen at random from the key space, then we want the probability that $h(k) = i$ to be $1/b$ for all buckets i. With this stipulation, a random key has an equal chance of hashing into any of the buckets. A hash function satisfying this property is called a *uniform hash function*.

Several kinds of uniform hash functions are in use in practice. Some of these compute the home bucket by performing arithmetic (e.g., multiplication and division) on the key. Since, in many applications, the data type of the key is not one for which arithmetic operations are defined (e.g., string), it is necessary to first convert the key into an integer (say) and then perform arithmetic on the obtained integer. In the following subsections, we describe four popular hash functions as well as ways to convert strings into integers.

8.2 Static Hashing – hash fts

A hash function maps a key into a bucket in the hash table. As mentioned earlier, the desired properties of such a function are that it be easy to compute and that it minimize the number of collisions. In addition, we would like the hash function to be such that it does not result in a biased use of the hash table for random inputs; that is, if k is a key chosen at random from the key space, then we want the probability that $h(k) = i$ to be $1/b$ for all buckets i. With this stipulation, a random key has an equal chance of hashing into any of the buckets. A hash function satisfying this property is called a *uniform hash function*.

Several kinds of uniform hash functions are in use in practice. Some of these compute the home bucket by performing arithmetic (e.g., multiplication and division) on the key. Since, in many applications, the data type of the key is not one for which arithmetic operations are defined (e.g., string), it is necessary to first convert the key into an integer (say) and then perform arithmetic on the obtained integer. In the following subsections, we describe four popular hash functions as well as ways to convert strings into integers.

8.2 Static Hashing – hash fts

8.2.2.1 Division

This hash function, which is the most widely used hash function in practice, assumes the keys are non-negative integers. The home bucket is obtained by using the modulo (%) operator. The key k is divided by some number D, and the remainder is used as the home bucket for k. More formally,

$$h(k) = k \% D$$

This function gives bucket addresses in the range 0 through $D - 1$, so the hash table must have at least $b = D$ buckets. Although for most key spaces, every choice of D makes h a uniform hash function, the number of overflows on real-world dictionaries is criticaly dependent on the choice of D. If D is divisible by two, then odd keys are mapped to odd buckets (as the remainder is odd), and even keys are mapped to even buckets. Since real-world dictionaries tend to have a bias toward either odd or even keys, the use of an even divisor D results in a corresponding bias in the distribution of home buckets. In practice, it has been found that for real-world dictionaries, the distribution of home buckets is biased whenever D has small prime factors such as 2, 3, 5, 7 and so on. However, the degree of bias decreases as the smallest prime factor of D increases. Hence, for best performance over a variety of dictionaries, you should select D so that it is a prime number. With this selction, the smallest prime factor of D is D itself. For most practical dictionaries, a very uniform distribution of keys to buckets is seen even when we choose D such that it has no prime factor smaller than 20.

8.2 Static Hashing – hash fts

When you write hash table functions for general use, the size of the dictionary to be accommodated in the hash table is not known. This makes it impractical to choose D as suggested above. So, we relax the requirement on D even further and require only that D be odd to avoid the bias caused by an even D. In addition, we set b equal to the divisor D. As the size of the dictionary grows, it will be necessary to increase the size of the hash table ht dynamically. To satisfy the relaxed requirement on D, array doubling results in increasing the number of buckets (and hence the divisor D) from b to $2b + 1$.

8.2.2.2 Mid-Square

The mid-square hash function determines the home bucket for a key by squaring the key and then using an appropriate number of bits from the middle of the square to obtain the bucket address; the key is assumed to be an integer. Since the middle bits of the square usually depend on all bits of the key, different keys are expected to result in different hash addresses with high probability, even when some of the digits are the same. The number of bits to be used to obtain the bucket address depends on the table size. If r bits are used, the range of values is 0 through 2^r-1. So the size of hash tables is chosen to be a power of two when the mid-square function is used.

8.2 Static Hashing – hash fts

8.2.2.3 Folding

In this method the key k is partitioned into several parts, all but possibly the last being of the same length. These partitions are then added together to obtain the hash address for k. There are two ways of carrying out this addition. In the first, all but the last partition are shifted to the right so that the least significant digit of each lines up with the corresponding digit of the last partition. The different partitions are now added together to get $h(k)$. This method is known as *shift folding*. In the second method, *folding at the boundaries*, the key is folded at the partition boundaries, and digits falling into the same position are added together to obtain $h(k)$. This is equivalent to reversing every other partition and then adding.

Example 8.2: Suppose that $k = 12320324111220$, and we partition it into parts that are three decimal digits long. The partitions are $P_1 = 123$, $P_2 = 203$, $P_3 = 241$, $P_4 = 112$, and $P_5 = 20$. Using shift folding, we obtain

$$h(k) = \sum_{1=1}^{5} P_i = 123 + 203 + 241 + 112 + 20 = 699$$

When folding at the boundaries is used, we first reverse P_2 and P_4 to obtain 302 and 211, respectively. Next, the five partitions are added to obtain $h(k) = 123 + 302 + 241 + 211 + 20 = 897$. □

8.2 Static Hashing – hash fts

8.2.2.5 Converting Keys to Integers

To use some of the described hash functions, keys need to first be converted to nonnegative integers. Since all hash functions hash several keys into the same home bucket, it is not necessary for us to convert keys into unique nonnegative integers. It is ok for us to convert the strings *data*, *structures*, and *algorithms* into the same integer (say, 199). In this section, we consider only the conversion of strings into non-negative integers. Similar methods may be used to convert other data types into non-negative integers to which the described hash functions may be applied.

Example 8.3: ***[Converting Strings to Integers]*** Since it is not necessary to convert strings into unique nonnegative integers, we can map every string, no matter how long, into an integer. Programs 8.1 and 8.2 show you two ways to do this.

```
unsigned int stringToInt(char *key)
{/* simple additive approach to create a natural number
    that is within the integer range */
   int number = 0;
   while (*key)
      number += *key++;
   return number;
}
```

Program 8.1: Converting a string into a non-negative integer

8.2 Static Hashing – overflow

8.2.3.1 Open Addressing

There are two popular ways to handle overflows: *open addressing* and *chaining*. In this section, we describe four open addressing methods—linear probing, which also is known as linear open addressing, quadratic probing, rehashing and random probing. In linear probing, when inserting a new pair whose key is k, we search the hash table buckets in the order, $ht[h(k)+i]\ \%\ b$, $0 \le i \le b-1$, where h is the hash function and b is the number of buckets. This search terminates when we reach the first unfilled bucket and the new pair is inserted into this bucket. In case no such bucket is found, the hash table is full and it is necessary to increase the table size. In practice, to ensure good performance, table size is increased when the loading density exceeds a prespecified threshold such as 0.75 rather than when the table is full. Notice that when we resize the hash table, we must change the hash function as well. For example, when the division hash function is used, the divisor equals the number of buckets. This change in the hash function potentially changes the home bucket for each key in the hash table. So, all dictionary entries need to be remapped into the new larger table.

8.2 Static Hashing – overflow

Example 8.4: Assume we have a 13-bucket table with one slot per bucket. As our data we use the words **for**, **do**, **while**, **if**, **else**, and **function**. Figure 8.2 shows the hash value for each word using the simplified scheme of Program 8.1 and the division hash function. Inserting the first five words into the table poses no problem since they have different hash addresses. However, the last identifier, **function**, hashes to the same bucket as **if**. Using a circular rotation, the next available bucket is at *ht* [0], which is where we place **function** (Figure 8.3). □

Identifier	Additive Transformation	x	Hash
for	102 + 111 + 114	327	2
do	100 + 111	211	3
while	119 + 104 + 105 + 108 + 101	537	4
if	105 + 102	207	12
else	101 + 108 + 115 + 101	425	9
function	102 + 117 + 110 + 99 + 116 + 105 + 111 + 110	870	12

[0]	**function**
[1]	
[2]	**for**
[3]	**do**
[4]	**while**
[5]	
[6]	
[7]	
[8]	
[9]	**else**
[10]	
[11]	
[12]	**if**

8.2 Static Hashing – overflow

When $s = 1$ and linear probing is used to handle overflows, a hash table search for the pair with key k proceeds as follows:

(1) Compute $h(k)$.

(2) Examine the hash table buckets in the order $ht[h(k)]$, $ht[(h(k) + 1) \% b]$, $\cdots$, $ht[(h(k) + j) \% b]$ until one of the following happens:

   (a) The bucket $ht[(h(k) + j) \% b]$ has a pair whose key is k; in this case, the desired pair has been found.

   (b) $ht[h(k) + j]$ is empty; k is not in the table.

   (c) We return to the starting position $ht[h(k)]$; the table is full and k is not in the table.

8.2 Static Hashing – overflow

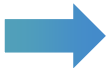
Some improvement in the growth of clusters and hence in the average number of comparisons needed for searching can be obtained by *quadratic probing*. Linear probing was characterized by searching the buckets $(h(k) + i) \% b$, $0 \le i \le b - 1$, where b is the number of buckets in the table. In quadratic probing, a quadratic function of i is used as the increment. In particular, the search is carried out by examining buckets $h(k)$, $(h(k) + i^2) \% b$, and $(h(k) - i^2) \% b$ for $1 \le i \le (b - 1)/2$. When b is a prime number of the form $4j + 3$, for j an integer, the quadratic search described above examines every bucket in the table. Figure 8.5 lists some primes of the form $4j + 3$.

An alternative method to retard the growth of clusters is to use a series of hash functions $h_1, h_2, \cdots, h_m$. This method is known as *rehashing*. Buckets $h_i(k)$, $1 \le i \le m$ are examined in that order. Yet another alternative, random probing, is explored in the exercises.

8.2 Static Hashing – chaining

bucket	x	buckets searched
0	**acos**	1
1	**atoi**	2
2	**char**	1
3	**define**	1
4	**exp**	1
5	**ceil**	4
6	**cos**	5
7	**float**	3
8	**atol**	9
9	**floor**	5
10	**ctime**	9
...		
25		

Hash table with linear probing (26 buckets, one slot per bucket)



[0] → **acos atoi atol**
[1] → *NULL*
[2] → **char ceil cos ctime**
[3] → **define**
[4] → **exp**
[5] → **float floor**
[6] → *NULL*
...
[25] → *NULL*

Hash chains

8.2 Static Hashing – chaining

Linear probing and its variations perform poorly because the search for a key involves comparison with keys that have different hash values. In the hash table of Figure 8.4, for instance, searching for the key **atol** involves comparisons with the buckets *ht* [0] through *ht* [8], even though only the keys in *ht* [0] and *ht* [1] had a collision with with **atol**; the remainder cannot possibly be **atol**. Many of the comparisons can be saved if we maintain lists of keys, one list per bucket, each list containing all the synonyms for that bucket. If this is done, a search involves computing the hash address $h(k)$ and examining only those keys in the list for $h(k)$. Although the list for $h(k)$ may be maintained using any data structure that supports the search, insert and delete operations (e.g., arrays, chains, search trees), chains are most frequently used. We typically use an array *ht* [0:b–1] with *ht* [i] pointing to the first node of the chain for bucket i. Program 8.4 gives the search algorithm for chained hash tables.

overflows. Although this is true when the keys are selected at random from the key space, it is not true in practice. In practice, there is a tendency to make a biased use of keys. Hence, in practice, different hash functions result in different performance. Generally, the division hash function coupled with chaining yields best performance.

The worst-case number of comparisons needed for a successful search remains $O(n)$ regardless of whether we use open addressing or chaining. The worst-case number of comparisons may be reduced to $O(\log n)$ by storing synonyms in a balanced search tree (see Chapter 10) rather than in a chain.

8.2 Static Hashing – chaining

```
element* search(int k)
{/* search the chained hash table ht for k, if a pair with
    this key is found, return a pointer to this pair;
    otherwise, return NULL.
   nodePointer current;
   int homeBucket = h(k);
   /* search the chain ht[homeBucket] */
   for (current = ht[homeBucket]; current;
                                  current = current→link)
      if (current→data.key == k) return &current→data;
   return NULL;
}
```

Program 8.4: Chain search

Thank You !

노력 없이 이룰 수 있는 것 아무것도 없다.