11 HASHING

OBJECTIVES

After completing this chapter, the reader will be able to understand the following:

- Use of hashing techniques that support very fast retrieval via a key
- Factors that affect the performance of hashing
- Collision resolution strategies

One of the most frequent and prolonged tasks in computer science is searching for a particular data record from a large amount of data. The expectation is to retrieve data within average constant time. Searching is the process of finding the location of the target among the list of objects using a key. *Key* is a field or combination of more than one field within the data record. It is used to uniquely identify the record and also to manage its access and usage.

We have discussed search techniques in Chapter 9. In both sequential and binary searches as well as in Fibonacci search, we need to perform many operations to locate the target data. The operations include computing the search index, comparing the target with the record at that index, and modifying the index again if not found. In an ideal situation, we expect the target to be searched in one or fewer attempts. One way to achieve this is that we should know (or should be able to obtain) the address of the record where it is stored. *Hashing* is a method of directly computing the address of the record with the help of a key by using a suitable mathematical function called the *hash function.* A *hash table* is an array-based structure used to store <key, information> pairs. In this chapter, we will learn about hashing, hash functions, and other related aspects.

11.1 INTRODUCTION

For many applications, we want to retrieve the target in one access or in constant average time. Hashing is finding an address where the data is to be stored or to locate using a key with the help of an arithmetic function. One of the applications this finds use in is language translators, such as assemblers and compilers. The compiler keeps all the variables used in a program in a symbol table, where the key is an arbitrary character string that corresponds to the identifiers in the language. The operations performed on a symbol table are those of dictionary operations. A hash table is an effective data structure for

implementing it. There are many such applications. Let us consider an array implementation for better understanding. The concept can be easily extended to other structures such as files.

Consider an example of an institute that has many departments in it. There is a central library and a departmental library for each department. Suppose we want to make a table of books for the departmental library using their unique identification number, say Accession No (Acc_No) as a key. A set of departmental library books is a subset of central library books, and the set of central library books is large enough. As the data is large enough, the range of Acc_No is 0000001 to 9999999; with 10^7 (may be minus one as we may omit 0000000) possible values. Let us assume that the departmental library has 20,000 books. Let us use an array for storing the book records and call it as Array_Book[]. As these books are from the central library, their Acc_No population is greater than the size of the storage area.

One way to access the book using one attempt is to store a book with Acc_No at $(\text{Acc\_No})^{\text{th}}$ location of Array_Book[] and for that we need an array of size 1,000,000. Instead of taking an array of size 1,000,000, we can use array of size just 20,000 and use the function $f(x)$ to map the numbers in the domain [0, …, 9,999,999] to the range [0, …, 19,999]. Figure 11.1 represents such mapping.

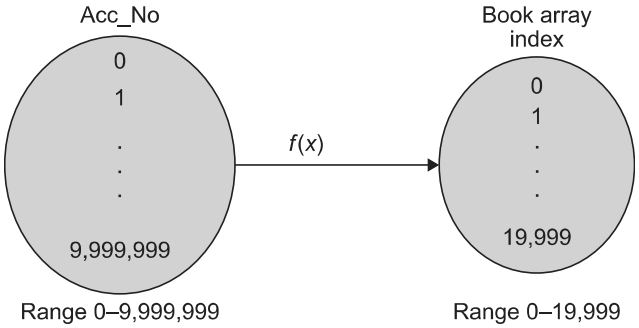


Fig. 11.1 Hash function

The function $f(x)$ will take Acc_No and return the indices where the book record is to be stored in the array and is called the *hash function*. Now each departmental book's address, which is an index in the table named Array_book, is calculated while storing as well as retrieving it. This concept of hashing is shown in Fig. 11.2.

Hash functions transform a key into an address. Hashing is a technique used for storing and retrieving information associated with it that makes use of the individual characters or digits in the key itself.

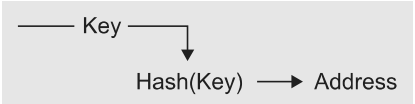


Fig. 11.2 Hashing concept

The resulting address is used as the basis for storing and retrieving records and this address is called the *home address* of the record. For an array to store a record in a hash table, the hash function is applied to the key of the record being stored, returning an index within the range of the hash table. The item is then stored in the table at that index position. To retrieve an item from a hash table, the same scheme that was used to store the record is followed.

Hashing is similar to indexing as it involves associating a key with a relative record address. However, it differs from indexing in the following two important ways:

1. With hashing, the address generated appears to be random—there is no obvious connection between the key and the location of the corresponding record, even though the key is used to determine the location of the record. For this reason, hashing is sometimes referred to as *randomizing*.
2. With hashing, two different keys may be transformed to the same address, so two records may be sent to the same place in a file. When this occurs, it is called a *collision* and some means must be found to deal with it. The two or more records that result in the same home address are known as *synonyms*.

11.2 KEY TERMS AND ISSUES

A problem arises, however, when the hash function returns the same value when applied to two different keys. To handle the situation, where two records need to be hashed to the same address we can implement a table structure, so as to have a room for two or more members at the same index positions. However, what happens if a third key hashes to the same index value? Before discussing such issues let us see some terms associated with hashing and the hash table:

Hash table Hash table is an array [0 to Max – 1] of size Max.

Hash function Hash function is one that maps a key in the range [0 to Max – 1], the result of which is used as an index (or address) in the hash table for storing and retrieving records. One more way to define a hash function is as the function that transforms a key into an address. The address generated by a hashing function is called the *home address*. All home addresses refer to a particular area of the memory called the *prime area.*

Bucket A bucket is an index position in a hash table that can store more than one record. Tables 11.1 and 11.2 show a bucket of size 1 and size 2, respectively. When the same index is mapped

Table 11.1 Table with bucket size 1

Index	Bucket of size 1
0	Alka
1	Bindu
2	
3	Deven
4	Ekta
5	
6	Govind
⋮	⋮
13	Monika
⋮	⋮
18	Sharmila
⋮	⋮
25	Zinat

with two keys, both the records are stored in the same bucket. The assumption is that the buckets are equal in size.

Consider the following example. Suppose we want to store 5 records with the key of each record as the person's name. The key can be hashed by taking the address from the ASCII representations of the first characters of the name. The table is of size 26, i.e., one bucket for each alphabet with size 2 (Table 11.2a) or size 3 (Table 11.2b).

Table 11.2(a) Table with bucket size 2

Index	Bucket of size 2	
0	Alka	Abhay
1	Bindu	Babali
2		
3	Deepa	Deven
4	Ekta	Esha
5		
6	Govind	Gopal
⋮	⋮	⋮
13	Monika	Meera
⋮	⋮	⋮
18	Sharmila	Sindhu
⋮	⋮	⋮
25	Zinat	Ziya

Table 11.2(b) Table with bucket size 3

Index	Bucket of size 3		
0	Alka	Abhay	Asmita
1	Bindu	Babali	Bhanu
2			
3	Deepa	Deven	Deepak
4	Ekta	Esha	Eshwar
5			
6	Govind	Gopal	Gautam
⋮	⋮	⋮	⋮
13	Monika	Meera	Manisha
⋮	⋮	⋮	⋮
18	Sharmila	Sindhu	Shilpi
⋮	⋮	⋮	⋮
25	Zinat	Ziya	Zeba

Probe Each action of address calculation and check for success is called as a *probe.*

Collision The result of two keys hashing into the same address is called collision.

Synonym Keys that hash to the same address are called synonyms.

Overflow The result of many keys hashing to a single address and lack of room in the bucket is known as an overflow. Collision and overflow are synonymous when the bucket is of size 1.

Open or external hashing When we allow records to be stored in potentially unlimited space, it is called as *open* or *external hashing*.

Closed or internal hashing When we use fixed space for storage eventually limiting the number of records to be stored, it is called as *closed* or *internal hashing*.

Hash function Hash function is an arithmetic function that transforms a key into an address which is used for storing and retrieving a record.

Perfect hash function The hash function that transforms different keys into different addresses is called a *perfect hash function*. The worth of a hash function depends on how well it avoids collision.

Load density The maximum storage capacity, that is, the maximum number of records that can be accommodated, is called as *loading density*.

Full table A full table is one in which all locations are occupied. Owing to the characteristics of hash functions, there are always empty locations, rather a hash function should not allow the table to get filled in more than 75%.

Load factor Load factor is the number of records stored in a table divided by the maximum capacity of the table, expressed in terms of percentage.

Rehashing Rehashing is with respect to closed hashing. When we try to store the record with Key1 at the bucket position Hash(Key1) and find that it already holds a record, it is collision situation. To handle collision, we use a strategy to choose a sequence of alternative locations Hash1(Key1), Hash2(Key1), and so on within the bucket table so as to place the record with Key1. This is known as *rehashing*.

Issues in hashing In case of collision, there are two main issues to be considered:

1. We need a good hashing function that minimizes the number of collisions.
2. We want an efficient collision resolution strategy so as to store or locate synonyms.

Let us learn about these two issues and techniques to resolve them in Sections 11.3 and 11.4, respectively.

11.3 HASH FUNCTIONS

To store a record in a hash table, a hash function is applied to the key of the record being stored, returning an index within the range of the hash table. The record is stored at that index position, if it is empty. With direct addressing, a record with key K is stored in slot K. With hashing, this record is stored at the location Hash(K), where Hash(K) is the function. The hash function Hash(K) is used to compute the slot for the key K. Let us discuss some issues regarding the design of good hash functions and also study the schemes for their creation.

11.3.1 Good Hash Function

The average performance of hashing depends on how the hash function distributes the set of keys among the slots. An assumption is that any given record is equally likely to hash into any of the slots, independent of whether any other record has been already hashed to it or not. This assumption is known as *simple uniform hashing*. A good hash function is one which satisfies this assumption.

If the probability that a key 'Key' occurs in our collection is P(Key), and for M slots in our hash table, a uniform hashing function, Hash(Key), should ensure that for $0 \leq \text{Key} \leq M - 1$, Σ P(Key) = 1, are all equiprobable with probability $1/M$. The hash function should ensure that they are hashed to different locations.

Sometimes, this is easy to ensure. For example, if the keys are randomly distributed in [0 ... r], with 0 to $M - 1$ locations then, Hash(Key) = floor(($M \times$ Key)/r) will provide uniform hashing.

Features of a Good Hashing Function

1. Addresses generated from the key are uniformly and randomly distributed.
2. Small variations in the value of the key will cause large variations in the record addresses to distribute records (with similar keys) evenly.
3. The hashing function must minimize the occurrence of collision.

There are many methods of implementing hash functions, let us discuss a few of them.

11.3.2 Division Method

One of the required features of the hash function is that the resultant index must be within the table index range. One simple choice for a hash function is to use the modulus division indicated as MOD (the operator % in C/C++). The function MOD returns the remainder when the first parameter is divided by the second parameter. The result is negative only if the first parameter is negative and the parameters must be integers. The function returns an integer. If any parameter is NULL, the result is NULL.

$$\text{Hash(Key)} = \text{Key } \% \ M$$

Key is divided by some number M, and the remainder is used as the hash address. This function gives the bucket addresses in the range of 0 through $(M - 1)$, so the hash table should at least be of size M. The choice of M is critical. While using this method, we usually avoid certain values of M. Binary keys of length in powers of two are usually avoided. A good choice of M is that it should be a prime number greater than 20.

11.3.3 Multiplication Method

Another hash function that has been widely used in many applications is the multiplication method. The multiplication method works as follows:

1. Multiply the key 'Key' by a constant A in the range $0 < A < 1$ and extract the fractional part of Key $\times A$.
2. Then multiply this value by M and take the floor of the result.

$$\text{Hash(Key)} = \lfloor M \times ((\text{Key} \times A) \text{ MOD } 1) \rfloor,$$

where Key $\times A$ MOD 1 is the fractional part of Key $\times A$,
that is, Key $\times A - \lfloor$Key $\times A \rfloor$ and one of the commonly used values of A = (sqrt(5) – 1/2 = 0.6180339887).

An advantage of the multiplication method is that the value of M is not critical. We typically choose it $M = 2^p$ for some integer p, since we can then easily implement the function in any programming language as:

1. Choose $M = 2^p$.
2. Multiply the ω bits of Key by floor $(A \times 2\omega)$ to obtain a 2ω bit product.
3. Extract the p most significant bits of the lower half of this product as address.

Note that we have used the function *floor*; *floor* and *ceil* are the commonly used math functions available in the library of almost all programming languages. These functions map a real number to the largest preceding or the smallest following integer, respectively. More precisely, floor$(x) = \lfloor x \rfloor$ is the largest integer not greater than x and ceil$(x) = \lceil x \rceil$ is the smallest integer not less than x.

11.3.4 Extraction Method

When a portion of the key is used for address calculation, the technique is called as the *extraction method*. In digit extraction, a few digits are selected, extracted from the key and are used as the address. For example, if the book accession number is of six digits and we require an address of 3 digits, then we can select the odd number digits—first, third, and fifth—which can be used as the address for the hash table.

For example, Table 11.3 shows the keys with their respective hashed addresses using digit extraction.

Table 11.3 Keys and addresses using digit extraction

Key	Hashed address
345678	357
234137	243
952671	927

Another way is to extract the first two and the last one or two digits. For example, for key 345678, the address is 3478 if the first two and the last two digits are extracted or 348 if the first two and the last digit are extracted.

If the portion of the key is carefully selected, it can be sufficient for hashing, provided the remaining portion distinguishes the keys in an insufficient way.

11.3.5 Mid-square Hashing

Mid-square hashing suggests to take the square of the key and extract the middle digits of the squared key as the address. The difficulty is when the key is large. As the entire key participates in the address calculation, if the key is large, then it is very difficult to store its square as it should not exceed the storage limit. So mid-square is used when the key size is less than or equal to 4 digits. For example, Table 11.4 shows the keys with their hashed addresses. If the key is a string, it has to be preprocessed to produce a number.

Table 11.4 Keys and addresses using mid-square

Key	Square	Hashed address
2341	5480281	802
1671	2792241	922

The difficulty of storing the squares of larger numbers can be overcome if we use fewer digits of the key (instead of the whole key) for squaring. If the key is large, we can select a portion of the key and square it. For example, Table 11.5 gives the keys and the squares of the first three digits with their hashed addresses.

Table 11.5 Keys and addresses using squares of fewer digits

Key	Square	Hashed address
234137	234 × 234 = 54756	475
567187	567 × 567 = 321489	148

11.3.6 Folding Technique

In this technique, the key is subdivided into subparts that are combined or folded and then combined to form the address. For a key with digits, we can subdivide the digits into three parts, add them up, and use the result as an address. Here the size of the subparts of the key is the same as that of the address.
There are two types of folding methods:

1. *Fold shift*—Key value is divided into several parts of the size of the address. Left, right, and middle parts are added.
2. *Fold boundary*—Key value is divided into parts of the size of the address. Left and right parts are folded on the fixed boundary between them and the centre part.

For example, if the key is 987654321, it is understood as
Left 987 Centre 654 Right 321

For fold shift, the sum is 987 + 654 + 321 = 1962. Now discard digit 1 and the address is 962. For fold boundary, sum of the reverse of the parts is 789 + 456 + 123 = 1368. Discard digit 1 and the address is 368.

11.3.7 Rotation

When the keys are serial, they vary only in the last digit and this leads to the creation of synonyms. Rotating the key would minimize this problem. This method is used along with other methods. Here, the key is rotated right by one digit and then folding technique is used to avoid synonyms. For example, let the key be 120605, when it is rotated we get 512060. Then the address is calculated using any other hash function.

11.3.8 Universal Hashing

Sometimes wrong operations are performed deliberately, such as choosing N keys all of which hash to the same slot, yielding an average retrieval time of $O(n)$. Any fixed hash function is helpless to this sort of worst-case behaviour. The only effective way to improve the situation is to choose the hash function randomly in a way that is independent of the keys that are actually going to be stored. This approach is called *universal hashing* and yields good performance on the average, no matter what keys are chosen.

The main idea behind universal hashing is to select the hash function at random at runtime from a carefully designed set of functions. Because of randomization, the algorithm can behave differently on each execution; even for the same input. This approach guarantees good average case performance, no matter what keys are provided as input.

11.4 COLLISION RESOLUTION STRATEGIES

No hash function is perfect. If Hash(Key1) = Hash(Key2), then Key1 and Key2 are synonyms and if bucket size is 1, we say that collision has occurred. As a consequence, we have to store the record Key2 at some other location. A search is made for a bucket in which a record is stored containing Key2, using one of the several collision resolution strategies. The collision resolution strategies are as follows:

1. Open addressing
   (a) Linear probing
   (b) Quadratic probing
   (c) Double hashing
   (d) Key offset
2. Separate chaining (or linked list)
3. Bucket hashing (defers collision but does not prevent it)

The most important factors to be taken care of to avoid collision are the table size and choice of the hash function. As we know, no hash function is perfect and we have a limitation on the table size too. Let us learn a few techniques to resolve this collision.

11.4.1 Open Addressing

In open addressing, when collision occurs, it is resolved by finding an available empty location other than the home address. If Hash(Key) is not empty, the positions are probed in the following sequence until an empty location is found. When we reach the end of table, the search is wrapped around to start and the search continues till the current collision location.

$$N(\text{Hash}(\text{Key}) + C(1)), N(\text{Hash}(\text{Key}) + C(2)), \ldots, N(\text{Hash}(\text{Key}) + C(i)), \ldots \quad (11.1)$$

Here N is the normalizing function, Hash(Key) is the hashing function, and C(i) is the collision resolution (or probing) function with the i^{th} probe. The normalizing function is required when the resulting index is out of range. A commonly used normalization function is MOD.

Closed hash tables use open addressing. In open addressing, all records are stored in the hash table itself also said to be resolving in the prime area which contains all home addresses. In case of chaining, the collisions are resolved by storing them at a separate area known as the *overflow area.*

In open addressing, when collision occurs, the table is searched for empty locations to store synonyms. Each table entry either contains a record or is empty. While searching

for a record, we systematically examine table slots until the desired record is found or it is clear that the record is not in the table.

While open addressing, to store the record, we successively examine, or probe, the hash table until we find an empty slot. Three techniques are commonly used to compute the probe sequences required for open addressing—linear probing, quadratic probing, and rehashing.

Linear Probing

A hash table in which a collision is resolved by placing the item in the next empty place following the occupied place is called *linear probing*. This strategy looks for the next free location until it is found. The function that we can use for probing linearly from the next location is as follows:

$$(\text{Hash}(x) + \text{C}(i)) \text{ MOD Max} \tag{11.2}$$

As C(i) = i for linear probing in Eq. (11.1), the function becomes

$$(\text{Hash}(x) + i) \text{ MOD Max}$$

Initially i = 1, if the location is not empty then it becomes 2, 3, 4, …, and so on till an empty location is found. We simply add one to the current address when collision occurs or till we find an empty location within the hash table limits. Alternatively, we can also add 2, subtract 2, or add 4, etc. Here Max is the table size or the nearest prime number greater than the table size. The use of MOD wraps the linear probing to the table start, if it reaches the end.

Let Max be 100, consider Table 11.6.

Let Key1 be 1044, now it hashes to location 44 and let us save it at that location. Now let Key2 be 3544 that also maps to address 44 and collision occurs as the table location 44 is already occupied. Here 1044 and 3544 are synonyms. Now the locations HashTable[45], HashTable[46], and so on are to be examined until a free location is found. The location 45 is found empty and the key 3544 is stored there.

Linear probing is easy to implement and the synonyms are stored nearer to the home address resulting in faster searches. When many synonyms are clustered around the home address, it is known as primary clustering. High degree of clustering increases the number of probes for locating data, increasing the average search time. Although linear probing is easy to implement, it tends to form clusters of synonyms, resulting in secondary clustering. The secondary clustering occurs when data is widely distributed in the hash table and have formed clusters throughout the table.

Table 11.6 Keys and Address

Index	Key
0	
1	
2	⋮
⋮	
44	1044
⋮	3544
98	
99	⋮

Linear probing can be done using the following:

1. *With replacement*—If the slot is already occupied by the key there are two possibilities, that is, either it is the home address (collision) or the location is occupied by some key. If the key's actual address is different, then the new key having the address at that slot is placed at that position and the key with the other address is placed in the next empty position.
   For example, in hash table of size 100, suppose Key1 = 127 is stored at address 25 and a new Key2 = 1325 is to be stored. Address for Key2 (1325 MOD 100) is 25. Now as the location 25 is occupied by Key1, the with replacement strategy places Key2 at location 25 and searches for an empty location for Key1 = 127.
2. *Without replacement*—When some data is to be stored in the hash table, if the slot is already occupied by the key, then another empty location is searched for a new record. There are two possibilities when the location is occupied—it is either its home address or not. In both the cases, the without replacement strategy searches for empty positions for the key that is to be stored.

Example 11.1 provides a better insight into linear probing.

EXAMPLE 11.1 Store the following data into a hash table of size 10 and bucket size 1. Use linear probing for collision resolution.

12, 01, 04, 03, 07, 08, 10, 02, 05, 14

Assume buckets from 0 to 9 and bucket size = 1 using hashing function key % 10.

Solution Let us use both techniques with and without replacement, as follows:

Linear probing with replacement For linear probing with replacement, when collision occurs, if the location is occupied by a record whose home address is not that location, it is replaced and the current record is stored there. Table 11.7 demonstrates all the operations.

Table 11.7 MOD as hash function and linear probing with replacement

Bucket	Initially empty	Insert 12	Insert 01	Insert 04	Insert 03	Insert 07	Insert 08	Insert 10	Insert 02	Insert 05	Insert 14
0								10	10	10	10
1			01	01	01	01	01	01	01	01	01
2		12	12	12	12	12	12	12	12	12	12
3					03	03	03	03	03	03	03
4				04	04	04	04	04	04	04	04
5									02	05	05
6										02	02
7						07	07	07	07	07	07
8							08	08	08	08	08
9											14

Here when key 02 is to be stored, it is hashed to address 2. However, that location is already occupied by 12. As 2 is the home address of 12, it resides there itself, and we linearly probe for the next empty location for key 02 to be stored. The location 5 is found empty and 02 is stored there.

When key 05 is to be stored, it maps to location 5 and is filled with key 02. Location 5 is not the home address of 02 and hence it is replaced. Key 05 is stored at location 5 and we again probe for the next empty location for 02 and store it at location 6.

Linear probing without replacement For linear probing without replacement when collision occurs, if the location is occupied, the next empty location is linearly probed for synonyms. Table 11.8 shows linear probing without replacement.

Table 11.8 MOD as hash function and linear probing without replacement

Bucket	Initially empty	Insert 12	Insert 01	Insert 04	Insert 03	Insert 07	Insert 08	Insert 10	Insert 02	Insert 05	Insert 14
0								10	10	10	10
1			01	01	01	01	01	01	01	01	01
2		12	12	12	12	12	12	12	12	12	12
3					03	03	03	03	03	03	03
4				04	04	04	04	04	04	04	04
5									02	02	02
6										05	05
7						07	07	07	07	07	07
8							08	08	08	08	08
9											14

Program Code 11.1 defines a function for inserting a record using linear probing without replacement.

PROGRAM CODE 11.1

```
//hash function to get position
int hash(int key)
{
   return(key % MAX);
}

//function for inserting a record using linear probe
int linear_prob(int Hashtable[], int key)
{
   int pos, i;
   pos = Hash(Key);
   if(Hashtable[pos] == 0)       // empty slot
```

```
    {
       Hashtable[pos] = key;
       return pos;
    }
    else      // slot is not empty
    {
       for(i = pos + 1; i % MAX != pos; i++)
       {
          if(Hashtable[i] == 0)
          {
             Hashtable[i] = key;
             return i;
          }
       }
    }
    // Table overflow
    return -1;
}
```

Quadratic Probing

In quadratic probing, we add the offset as the square of the collision probe number. In quadratic probing, the empty location is searched by using the following formula:

$$(\text{Hash(Key)} + i^2) \text{ MOD Max where } i \text{ lies between 1 and (Max} - 1)/2 \qquad (11.3)$$

Here if Max is a prime number of the form ($4 \times \text{integer} + 3$), quadratic probing covers all the buckets in the table.

Quadratic probing works much better than linear probing, but to make full use of the hash table, there are constraints on the values of i and Max so that the address lies within the table boundaries. In addition, if two keys have the same initial probe position, then their sequences are the same. Similar to linear probing, the initial probe determines the entire sequence and hence maximum distinct probe sequences are used. As the offset added is not 1, quadratic probing slows down the growth of primary clusters.

Program Code 11.2 depicts this logic.

PROGRAM CODE 11.2

```
//hash function to get position
int hash(int key)
{
   return(key % MAX);
}
```

```
//function for inserting record using linear probe
int quadratic_prob(int Hashtable[], int key)
{
   int pos, i;
   pos = hash(key);
   for(i = 0; i % MAX != pos; i++)
   {
      pos = (pos + i * i ) % MAX;
      if(Hashtable[pos] key == 0) // empty slot
      {
         Hashtable[pos] = key;
         return pos;
      }
   }      // Table overflow
   return -1;
}
```

Let us see Examples 11.2 and 11.3, which use linear probing and quadratic probing, respectively.

EXAMPLE 11.2 Suppose Max = 8 and keys *A*, *B*, *C*, *D* have hash values Hash(*A*) = 3, Hash(*B*) = 0, Hash(*C*) = 4, and Hash(*D*) = 3. Use linear probing for collision resolution.

Solution Linear probing is the simplest strategy where Hash(Key) = Hash((Key + *i*) MOD Max).

Suppose we wish to insert *D* and find that bucket 3 has been filled already, then we would try buckets 4, 5, 6, 7, 0, 1, and 2 in sequence. We find bucket 5 empty and we store *D*.

0	*B*
1	
2	
3	*A*
4	*C*
5	*D*
6	
7	

EXAMPLE 11.3 Consider the keys 22, 17, 32, 16, 5, and 24. Let Max = 7. Let us use quadratic probing to handle synonyms.

Solution Let the hash functions be (Key MOD Max); for quadratic probing (Hash(Key) ± i^2) MOD Max.

After storing 22, 17, 32, 5, and 7, the table looks as shown in the left column of Table 11.9.

Table 11.9 Keys and quadratic probing

Index	Key		Index	Key
0			0	24
1	22		1	22
2	16	insert 24 →	2	16
3	17		3	17
4	32		4	32
5	5		5	5
6			6	

We can see that while inserting 24, the address we get is

$$\text{Hash}(24) = 24 \text{ MOD } 7$$
$$= 3$$

It is also noted that the location 3 is already occupied.
We may now go for the quadratic function as

$$[\text{Hash}(24) - (1)^2 \text{ MOD } 7]$$
$$= (24 \text{ MOD } 7) + 1 \text{ MOD } 7$$
$$= (3 + 1) \text{ MOD } 7 = 4 \text{ which is not occupied.}$$

$$\text{Hence, Hash}(24) + (2)^2 \text{ MOD } 7$$
$$= (3 + 4) \text{ MOD } 7 = 0$$

which is empty, so store 24 there.

Double Hashing

Double hashing uses two hash functions, one for accessing the home address of a Key and the other for resolving the conflict. The sequence for probing is generated as follows:

$$(\text{Hash1(Key)}, (\text{Hash1(Key)} + i \times \text{Hash2(Key)}), \ldots. \; i = 1, 2, 3, 4, \ldots$$

and the resultant address is modulo Max. Example 11.4 illustrates the double hashing concept.

EXAMPLE 11.4 Let the hash function be Key % 10, Max = 10, and the keys be 12, 01, 18, 56, 79, 49. Perform double hashing.

Solution Table 11.10 demonstrates all insertions and collision handling using double hashing.

Table 11.10 Double hashing

	Initially empty	Insert 12	Insert 01	Insert 18	Insert 56	Insert 79	Insert 49
0							
1			01	01	01	01	01
2		12	12	12	12	12	12
3							
4							
5							49
6					56	56	56
7							
8				18	18	18	18
9						79	79

While inserting 49, the hashed location 9 is found occupied by key 79, so let us use Hash2(Key) = R – (Key MOD R), where R is a small prime number, even smaller than the table size. Let us use $R = 7$.

To insert 49, using Hash1(Key) = 49 % 10, we get 9 which is already occupied, so we use Hash2 as follows:

$$\text{Hash2}(49) = 7 - (49 \% 7) = 7 - 0 = 7$$

Hence by double hashing,

$$\begin{aligned}\text{Hash}(49) &= [\text{Hash1}(49) + \text{Hash2}(49)] \% 10 \\ &= (9 + 7) \% 10 \\ &= 6 \text{ and location 6 is not empty, so let us recompute again.}\end{aligned}$$

$$\begin{aligned}\text{Hash}(49) &= [\text{Hash1}(49) + 2 \times \text{Hash2}(49)] \% 10 \\ &= 9 + 2 \times 7 \\ &= 25 \% 10 \\ &= 5 \text{ and is empty, so store key 49 there.}\end{aligned}$$

Example 11.5 illustrates the various types of open addressing.

EXAMPLE 11.5 Given the input {4371, 1323, 6173, 4199, 4344, 9699, 1889} and hash function as Key % 10, show the results for the following:

1. Open addressing using linear probing
2. Open addressing using quadratic probing
3. Open addressing using double hashing h2(x) = 7 – (x MOD 7)

Solution The results are as follows:

1. Open addressing using linear probing

These keys are inserted using linear probing as shown in Table 11.11

Table 11.11 Inserting keys using linear probing

	Initially empty	Insert 4371	Insert 1323	Insert 6173	Insert 4199	Insert 4344	Insert 9699	Insert 1889
0							9699	9699
1		4371	4371	4371	4371	4371	4371	4371
2								1889
3			1323	1323	1323	1323	1323	1323
4				6173	6173	6173	6173	6173
5						4344	4344	4344
6								
7								
8								
9					4199	4199	4199	4199

Using linear probing, while inserting 9699 and 1889, as the hashed locations are not empty, the keys are stored at the next empty locations probed in circular at positions 0 and 2, respectively.

2. Open addressing using quadratic probing

Let us insert these keys using quadratic probing now as shown in Table 11.12.

Table 11.12 Inserting keys using quadratic probing

	Initially empty	Insert 4371	Insert 1323	Insert 6173	Insert 4199	Insert 4344	Insert 9699	Insert 1889
0							9699	9699
1		4371	4371	4371	4371	4371	4371	4371
2								
3			1323	1323	1323	1323	1323	1323
4				6173	6173	6173	6173	6173
5						4344	4344	4344
6								
7								
8								1889
9					4199	4199	4199	4199

For 6173, the hashed address 6173 % 10 gives 3 and it is not empty, hence using quadratic probing we get the address as follows: Hash(6173) = (6173 + 1^2) % 10 = 4 and as it is

empty, the key 6173 is stored there. Now while inserting 4344, the location 4 is not empty and hence quadratic probing generates the address as Hash(4344 + 1^2) % 10 = 5 and as is empty 4344 is stored. For key 9699, the address is Hash(9699 + 1^2) % 10 = 0 and is empty so store it there. While inserting 1889, the address Hash(1889 + 1^2) % 10 = 0 is not empty so probe again. The address Hash(1889 + 2^2) % 10 = 3 is not empty so probe again. The address Hash(1889 + 3^2) % 10 = 8 is empty so store 1889 at location 8.

3. Open addressing using double hash function

Table 11.13 shows the status of the hash table after inserting each key using open addressing using double hashing

Table 11.13 Open addressing using double hash

	Initially empty	Insert 4371	Insert 1323	Insert 6173	Insert 4199	Insert 4344	Insert 9699	Insert 1889
0								1889
1		4371	4371	4371	4371	4371	4371	4371
2							9699	9699
3			1323	1323	1323	1323	1323	1323
4				6173	6173	6173	6173	6173
5								
6								
7						4344	4344	4344
8								
9					4199	4199	4199	4199

While inserting 6173, the address is Hash1(6173) = 6173 % 10 = 3 and 3 is not empty. Let us use double hashing. Hence the address is as follows:

$$\begin{aligned} \text{Hash}(6173) &= [\text{Hash1}(6173) + \text{Hash2}(6173)] \% 10 \\ &= 3 + (R - 6173 \% R) \text{ (let } R \text{ be 7)} \\ &= 3 + (7 - 6) = 4 \end{aligned}$$

Since 4 is empty, we store 6173 at location 4.

Now let us store 4344. The address 4344 % 10 = 4 and as location 4 is not empty, we use double hashing and we get Hash(4344) = 7. Now for 9699 double hashing generates address 2 and as it is empty, we store it there. For key 1889, double hashing generates address 0 and as it is empty, we store 1889 at location 0.

Rehashing

If the table gets full, insertion using open addressing with quadratic probing might fail or it might take too much time. The solution for this problem is to build another table that

is about twice as big and scan down the entire original hash table, compute the new hash value for each record, and insert them in a new table.

For example, if initially, the table is of size 7 and the hash function is key % 7 then, this would be as shown in Table 11.14.

As the table is more than 70% full, a new table is created (Table 11.15) and the values are inserted in the new table. The size of the new table is 17, that is next prime of double of 7 that is 14. Rehashing is very expensive, as its running time is O(N).

Table 11.14 Table of size 7

	Insert 7, 15, 13, 74, 73
0	7
1	15
2	
3	73
4	74
5	
6	13

Table 11.15 New table of size 17 when Table 11.14 is 70% full

0	
1	
2	
3	
4	
5	73
6	74
7	7
8	
9	
10	
11	
12	
13	13
14	
15	15
16	

11.4.2 Chaining

We have discussed three techniques that are used to compute probe sequences (to relocate synonyms) namely, linear probing, quadratic probing, and rehashing. Of course, we can store the linked lists inside the hash table, in the unused hash table slots. The technique used to handle synonyms is chaining; it chains together all the records that hash to the same address. Instead of relocating synonyms, a linked list of synonyms is created whose head is the home address of synonyms. In Chapter 6, we have discussed implementing a linked list within an array. However, we need to handle pointers to form a chain of synonyms. The extra memory is needed for storing pointers.

In Fig. 11.3, a hash table with Max = 10, both keys 322 and 262 probe to address 2. A chain, a linked list, stores all items at a particular home address (home address is an address within the hash table itself).

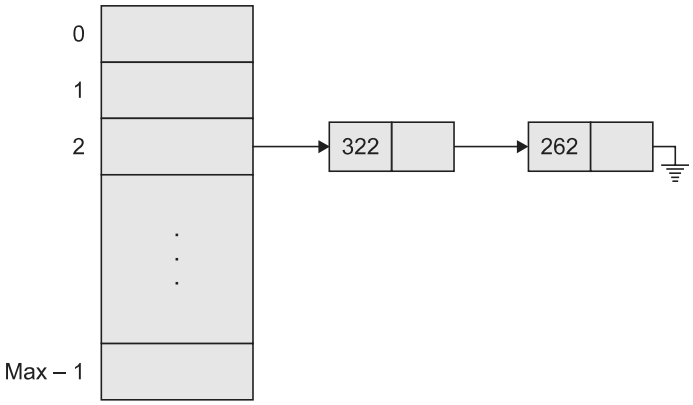


Fig. 11.3 An example of chaining

Let us compare rehashing and chaining (Table 11.16).

Table 11.16 Comparison of chaining and rehashing

Chaining	Rehashing
Unlimited number of synonyms can be handled.	A limited but good number of synonyms are taken care of.
Additional cost to be paid is an overhead of multiple linked lists.	The table size is doubled but no additional fields of links are to be maintained.
Sequential search through the chain takes more time.	Searching is faster when compared to chaining.

Program Code 11.3 illustrates chaining.

PROGRAM CODE 11.3

```
#define MAX 10
class node
{
   public:
      int key;
      struct node *next;
};

Node *hashtable[max];
```

```
void int()
{
   int i;
   for(i = 0; i < n; i++)
   {
      Hashtable[i] = null;
   }
}

int hash(int key)
{
   return(key % 10);
}

void insert(int k)
{
   int pos;
   Node *Curr, *Temp;
   Curr = new node;
   Curr->key = k;
   Curr->next = null;
   pos = hash(Curr->key);
   if(Hashtable[pos] == null)
      Hashtable[pos] = Curr;
   else
   {
      // goto last node and attach
      Temp = Hashtable[pos];
      while(Temp->next != null)
         Temp = Temp->next;
         // attach
         Temp->next = Curr;
   }
}

void display()
{
   Node *Curr;
   for(i = 0; i < 10; i++)
   {
      Curr = Hashtable[i];
```

```
        while(curr != null)
        {
            cout << curr->key << "\t";
            Curr = Curr->next;
        }
    }
}

void search(int x)
{
    Node *Curr;
    pos = hash(x);
    Curr = Hashtable[pos];
    while(curr != null && Curr->key != x)
    {
        cout << curr->key << "\t";
        Curr = Curr->next;
    }
    if(Curr == null)
        cout << "\n Not Found";
    else
        cout << "\n  Key Found";
}
```

11.5 HASH TABLE OVERFLOW

Even if a hashing algorithm (function) is very good, it is likely that collisions will occur. The identifiers that have hashed into the same bucket, as discussed earlier, are called synonyms.

An *overflow* is said to occur when a new identifier is mapped or hashed into a full bucket. When the bucket size is one, a collision and an overflow occur simultaneously. Therefore, any hashing program must incorporate some method for dealing with records that cannot fit into their home addresses. There are a number of techniques for handling overflow of records.

11.5.1 Open Addressing for Overflow Handling

We shall study two ways to handle overflows—open addressing and chaining. In open addressing, we assume that the hash table is an array. When a new identifier is hashed into a full bucket, we need to find another bucket for this identifier. The simplest solution is to find the closest unfilled bucket through linear probing or linear open addressing.

When linear open addressing is used to handle overflows, a hash table search for an identifier I proceeds as follows:

1. Compute Hash(I)
2. Examine identifiers position
   Table[Hash(I)], Table[Hash(I) + 1], …, Table[Hash[I] + i], in order until:
   (a) If Table[Hash(I) + j] = I then
   In this case I is found.
   (b) If Table[Hash(I) + j] is NULL, then I is not in the table.
   (c) If we return to the start position Hash(I), then the table is full and I is not in the table.

One of the problems with linear open addressing is that it tends to create clusters of identifiers. Moreover, these clusters tend to merge as more identifiers are entered, leading to big clusters. An alternative method to retard the growth of clusters is to use a series of hash functions $h_1, h_2, \ldots, h_m$. This method is called as rehashing. Buckets $h_i(x)$, $1 \le i \le m$ are examined in that order.

11.5.2 Overflow Handling by Chaining

Linear probing and its variations are inefficient as the search for an identifier involves comparison with identifiers that have different hash values. Consider the following hash table shown in Fig. 11.4.

0	1	2	3	4	5	6	7	8	9	10	11		25
A	A_2	A_1	D	A_3	A_4	G_A	G	Z_A	E		L	…	Z

Fig. 11.4 Chaining

In the above hash table of 25 buckets, one slot per bucket, searching for the identifier Z_A involves comparisons with the buckets Table[0] to Table[7], even though none of the identifiers in these buckets had a collision with Table[25] and so cannot possibly be Z_A. Many of the comparisons can be saved if we maintain lists of identifiers, one list per bucket, each list containing all the synonyms for that bucket. If this is done, a search involves computing the hash address Hash(I) and examining only those identifiers in the list for Hash(I). Since the sizes of these lists are not known in advance, the best way to maintain them is as linked chains. In each slot, additional space is required for a link. Each chain has a head node. The head node, however, usually is much smaller than the other nodes, since it has to retain only a link. As the list is accessed at random, the head nodes should be sequential. We assume that they are numbered 0 to $n - 1$, if hash function Hash() has range 0 to $n - 1$.

For hash table in Fig. 11.4 can be represented as hash table in Fig. 11.5 using the hash chains.

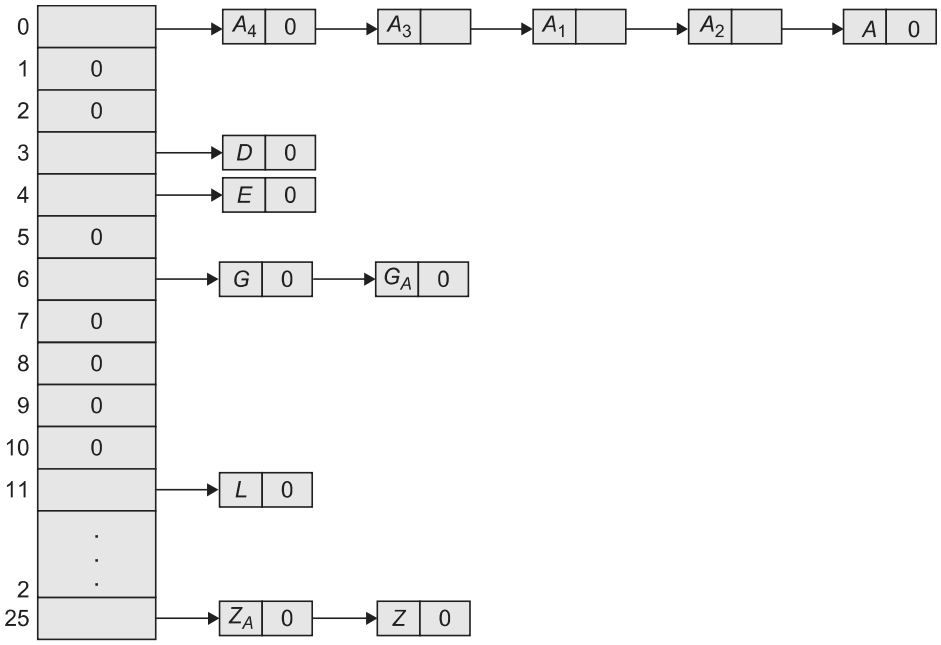


Fig. 11.5 Hash chains

To insert a new identifier, *I*, into a chain, we must first verify that it is not currently in chain. Then, if not present, *I* is inserted at any position in the chain.

11.6 EXTENDIBLE HASHING

If linear probing or separate chaining is used for collision handling, then in case of collision, several blocks are required to be examined to search a key and when table is full, then expensive rehash should be used. For fast searching and less disk access, extendible hashing is used. It is a type of hash system, which treats a hash as a bit string, and uses a trie for bucket lookup.

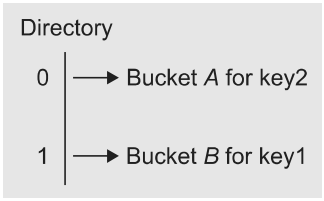
For example, assume that the hash function Hash(Key) returns a binary number.

The first *i* bits of each string will be used as indices to figure out where they will go in the hash table. Additionally, *i* is the smallest number such that the first *i* bits of all keys are different.

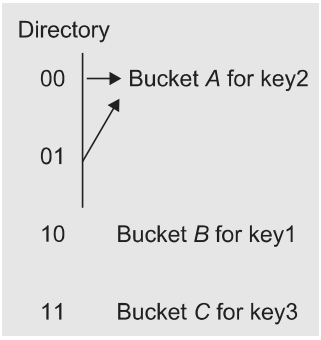
The keys to be used are as follows:

1. h(key1) = 100101
2. h(key2) = 011110
3. h(key3) = 110110

Let us assume that for this particular example, the bucket size is 1. The first two keys to be inserted, key1 and key2, can be distinguished by the most significant bit, and would be inserted into the table as follows:



When key3 is hashed to the table, it would not be enough to distinguish all three keys by one bit (because key3 and key1 have 1 as their leftmost bit). Also, because the bucket size is one, the table would overflow. Because comparing the first two most significant bits would give each key a unique location, the directory size is doubled as follows:



And so now key1 and key3 have unique locations being distinguished by the first two leftmost bits. Since key2 is in the top half of the table, both 00 and 01 point to it because there is no other key that begins with a 0 to compare.

The root of the tree contains four pointers determined by the leading two bits of data. Each leaf has upto 4 records. *D* will be represented by the number of bits used by the root, which is known as a directory.

11.7 DICTIONARY

A set is an unordered collection of distinct elements. Each element has a field called *key* that is usually unique. The requirement of uniqueness is sometimes circumvented and is known as a *multiset* or a *bag*. Multiset is a set whose members are not necessarily distinct. The most common operations performed on a set or multiset are searching, inserting, and deleting elements from a group. A dictionary is a data structure for efficiently implementing these operations. The simplest way to implement a dictionary is through the use of arrays. Arrays are efficient for searching an element, whereas insertion and deletion cannot be easily performed. The proficient implementation has to balance the efficiency of searching with the other two operations. Other sophisticated ways to implement a dictionary is using hashing and balanced search trees.

A typical dictionary includes the following operations:

1. Empty—checks whether the dictionary is empty or not
2. Size—determines the dictionary size
3. Insert—inserts a pair into the dictionary
4. Search—searches the pair with a specified key
5. Delete—deletes the pair with a specified key

11.8 SKIP LIST

A balanced tree is one of the most popular data structures used for searching. One of the variants of balanced trees is the skip list. The skip list is a probabilistic data structure that has become the method of choice for many search-based applications instead of balanced trees.

A skip list stores the sorted data in the form of a linked list. These items are stored as a hierarchy of linked lists where each list links increasingly sparse subsequences of the items. These supplementary lists result in an item search that is as efficient as that of balanced binary search trees. Since each link of the sparser lists skips over many items of the full list in one step, the list is called *skip list*. These forward links are added on the basis of the probability of the element search. Hence, insert, search, and delete operations are performed in logarithmic expected time. The links may also be added in a non-probabilistic way. Skip list algorithms have the same asymptotic expected time bounds as balanced trees and are simpler, faster, and use less space. Figure 11.6 shows the diagrammatic representation of a skip list.

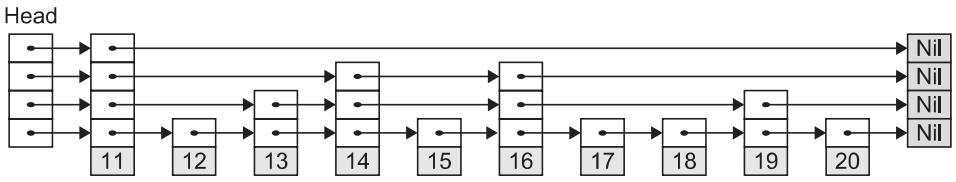


Fig. 11.6 Diagrammatic representation of a skip list

11.9 COMPARISON OF HASHING AND SKIP LISTS

The following is a list of similarities and differences between hashing and skip lists:

- The hash table is a simple array of items; hashing algorithms calculate an index from the data item's key and use this index to place the data into the array. A hash table is an alternative method for representing a dictionary. It is a popular data structure which is simple and easy to implement.
- The skip list is a linked list augmented with layers of pointers for quickly jumping over a large numbers of elements and then descending to the next layer. This process continues down to the bottom layer, which is the actual list. Skip lists are interesting data structures which are powerful and flexible.
- Skip lists are one way of implementing a dictionary abstract data type, which stores a set of items and allows us to add, remove, and search for items. Though hash tables are more popular, skip lists improve the performance of insert and delete operations.
- The expected performance of search and delete operations on skip lists is $O(\log n)$; however, the worst-case performance is $\Theta(n)$. The hash table is used in many applications. In ideal situations, the hash table search, insert, or delete takes $\Theta(1)$.

- There are many issues associated with hash tables such as the choice of the hash function, overflow handling, and the size (i.e., number of buckets) of the hash table.

RECAPITULATION

- Many applications need a dynamic set of operations that supports only insert, member search, and delete. A keyed table is an effective data structure for implementing them.
- Hashing is an excellent technique for implementing keyed tables. A hash table is an array-based structure used to store <key, information> pairs.
- Hash tables are used to implement insertions and searches in constant average time. To store an item in a hash table, a hash function is applied to the key of the item being stored, returning an index within the range of the hash table.
- Hashing is a technique that is used for storing and retrieving information associated with and that makes use of the individual characters or digits in the key itself.
- A problem arises, however, when the hash function returns the same value when applied to two different keys called collision. However, there are various collision resolution techniques to overcome these problems.
- Dictionary and skip lists are types of data structures used for storing data in the form of an array and linked list, respectively. However, skip list is more efficient and thus the preferred option for performing search operations on a given data set as it is simpler, faster, and uses less space when compared to other techniques.

KEY TERMS

Bucket An index position in hash table that stores a fixed number of buckets.

Collision The result of two keys hashing into the same bucket (index positions).

Dictionary A dictionary is a type of data structure that can efficiently implement operations such as searching, inserting, and deleting elements on a set or multiset from a group.

Hash function To store an item in a hash table, a hash function is applied to the key of the item being stored, returning an index within the range of the hash table.

Hashing Hashing is a technique that is used for storing and retrieving information associated with and that makes use of the individual characters or digits in the key itself. Hashing is an excellent technique for implementing keyed tables.

Hash table A hash table is an array-based structure used to store <key, information> pairs. In other words, we can say that the hash table is a table for storing key and related information.

Overflow When more than one key has the same index and if there is no space in bucket, we say that overflow has occurred.

Skip list A skip list is one of the variants of balanced trees, which is used most efficiently for searching operations.

Synonym Keys that hash to the same bucket are called synonyms.