## Chapter 3.3 Hash Table

*Definition:*

*Hash Table can be used as Index and Main Memory Data Structure. In such structure, there has Hash Function h, and it uses Data Key as Parameter and calculate the Integer which belongs to (0, B - 1), among which B is the number of bucket.*

Bucket Array, is an array with the Sequence from 0 to B - 1, among which there includes B Bucket Array, among which includes B linked list heads, each one corresponding to one bucket in the Array. If Query Key of Record equals to K, then we can link the record to bucket index h(K) to store Query Key.

### Chapter 3.3.1 Auxiliary Hash Table

### *Introduction:*

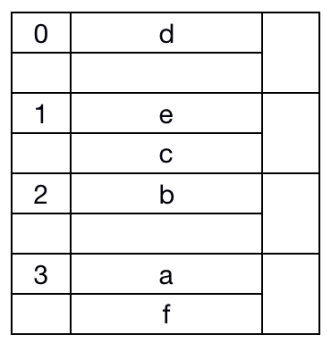
### Since Auxiliary Hash Table is too big to stay in Main Memory, therefore we put it into Auxiliary Storage, and there have tiny and important difference between Auxiliary Hash Table and Main Memory Hash Table.

*First of all, Bucket Array consists of the Storage Block but not Pointers that point to Linked List Head. Through Hash Table h, we can hash all records in this Hash Table. If there has too much records in the Bucket, then we can add extra Linked List into Bucket to save much more records.*

Assume that, given any i, then the Location of first Storage Block in Bucket i can be reached.

### *Example:*

Hash Table is just as shown below. In order to make it easy to manage, we need to assume that each Storage Block can store two pieces of records, and B = 4, and the return value in Hash Table h would between 0 and 3. We has listed some Hash Table records. In the Table below, then Key Value is Alphabet Number range from a to f. Assume that h(d) = 0, h(c) = h(e) = 1, h(b) = 2 and h(a) = h(f) = 3. The record range distribution is as below.



Attention that, the right side of each Storage Block has one tiny square, this tiny square represents additional information which is saved in the Storage Block.

### Chapter 3.3.2 Insertion into Hash Table

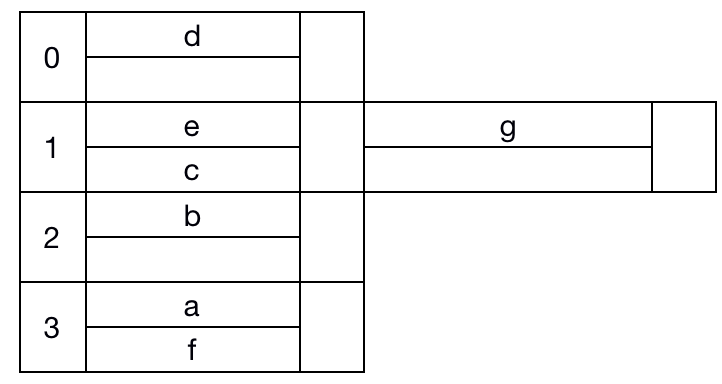
### *Principle:*

When one record with Query Key K needs to be inserted into Hash Table, then we need to calculate h(K).

* *If there has extra space for Bucket Number h(K), then this record would be stored into such bucket.*
* *Otherwise if there has no such space for Bucket with Number h(K), then we need to add one storage block of Bucket into List and store such block into the additional storage block of Bucket.*

### *Example:*

Assume that we add one Key Value which equals to g and h(g) = 1. We need to store such record into No.1 Bucket. However, there already have two Blocks, we need to add one extra new Block and linked the first Block. The Hash Table is just as image below.



### Chapter 3.3.3 Deletion from Hash Table

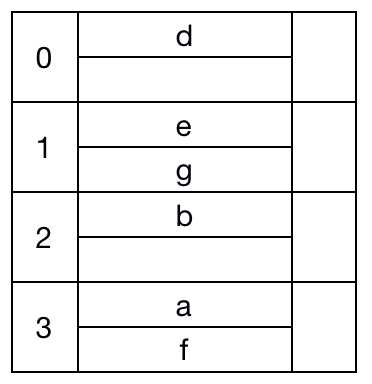
### *Principle:*

The method to delete the record with Query Key K is just the same as the method to insert.

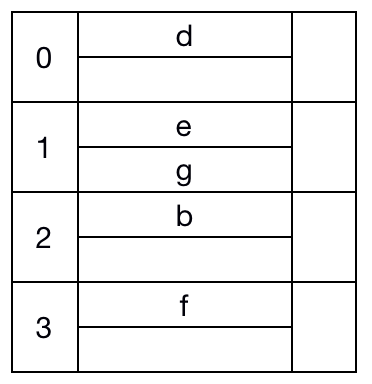
*We need to find the h(K) bucket and search the Query Key with Data Value K, and to find the record to be deleted. If the record can be moved among Blocks, after delete it, we can merge Storage Block in the same linked chain.*

### *Example:*

We need to delete the Data Record with Key Value c from Hash Table. Known from h(c) = 1, we can find the first Bucket and the record with Key Value c. After delete the record c from Hash Table, we need to remove the record with Key Value g from the Second Linked List to the first List and delete the second Storage Block.



We also delete the record with Key Value a. For this Key - Value, we find the third Storage Block and delete this record, and move the left block forward to make it much more compact.



### Chapter 3.3.4 Efficient of Hash Index

### *Principle:*

### The optimal situation is that there would exist enough buckets and most of which consist of single block. If that, then the general Query would only need one Disk I/O and File Insertion and Deletion would only need two times Disk I/O. Such result would be much better than Sparse Index, Dense Index or B - Tree.

However, as the number of File is growing, then the situation that there would have a lot of Blocks linked after the Linked List. Under this kind of situation, we need to Query in the Long Linked List, and each block only needs one time Disk I/O. So we need try to decrease the block number of each Bucket.

*So far, we need to know that the Hash Table is called Static Hash Table, since the number of Bucket B would never change.* And, there also have some other kinds of *Dynamic Hash Table, they allow the number of Bucket B to change, and B equals to the total number of record number divide by the contained record numbers, which means that, each bucket would contain only one Storage Block.* We will discuss two methods:

1. *Extensible Hash Table:*

*The first method would double the number of B when it is considered to small.*

1. *Linear Hash Table*

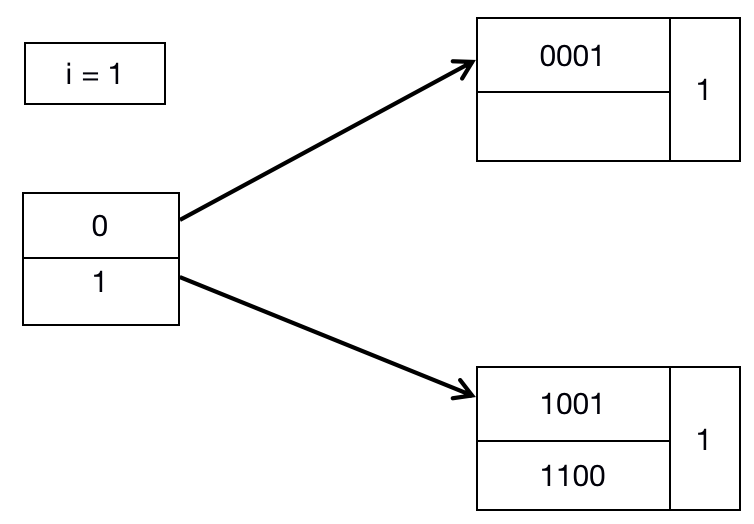
*The second method would add 1 to B when the Statistical Number of File indicates.*

### Chapter 3.3.5 Extensible Hash Table

### *Definition:*

The first kind of Dynamic Hash Method is called Extensible Hash Table. It can only add below items to the Simple Static Hash Table:

* *Introduce the Intermediate Level for the Bucket, which means to use the Pointer Array which points to the Bucket to represent Bucket, but not the Data Block Array itself to represent Bucket.*
* *The Pointer Array can increase, it’s length is always the value of 2 power, because each time the length of Array increases, then the number of Bucket would be doubled.*
* *However, not each Bucket has it’s Data Block; If the records in some Buckets can be put into one Bucket, then these buckets can share one Block.*
* *Hash Function h would calculate one K bytes Binary Sequence for each Key, if the value of K is big enough, such as 32. But, the number of Bucket would always be several bytes start from the first byte of Binary Sequence or from the last byte of Binary Sequence, the number of these bytes is less than K, we can assign it the number of i. Which is to say that when i equals to the used bytes, then the Bucket Array would have 2^i items.*



### *Example:*

The table above has shown one small extensible Hash Table. For simpler, we assume that K = 4, which means by using Hash Function, we can generate 4 bytes Binary Sequence. When we need to use one of byte of the Sequence, then just like i = 1, there will have two items which corresponds to 0 and 1.

We can check from Hash Table that the first Block save the Query Record starts with 0; while the Second Block save the Query Record starts with 1. For convenience, here the data stored in the Hash Table is the Binary Sequence Number. Therefore, the first block saves one record with Data Value equals to 0001 and the second block saves two records with Data Value equals to 1001 and 1100.

Here, we need to pay attention that, the number after the Data Value in the Bucket which are all as 1. This number is used to signify that how much bytes are used to decide the membership of the record in the Bucket.

In the example above, there has only one byte of the Data Value Sequence to decide where to locate among all Bucket Blocks. However, as the size of Bucket increases, then the different bytes in different Bucket Block may need to be considered, which is to say that, the size of Bucket Block is decided by the biggest binary byte, but some Bucket Block may need less bytes.

### Chapter 3.3.6 Insertion into Extensible Hash Table

### *Introduction:*

In order to insert Record with Key Value K into Hash Table, we need to calculate h(K), get the first i bytes from the Binary Sequence, and find the field with Sequence Number equals to i in the Bucket Array. Attention that, since i can be used as a part of Data Structure to save, so we can make sure the value of i.

### *Rule:*

At first, we find the Storage Block B according to the pointer of such item. If there still has extra place for Storage Block B, we would need to store the new record, and then the insertion would finish.

If there have no other space in B, then there would have two different possibilities for different i, the number j stands for the byte number in Hash Table can be used to make sure the membership of Storage Block B. *(The value of j can be found in the last storage block for each Storage Block.)*

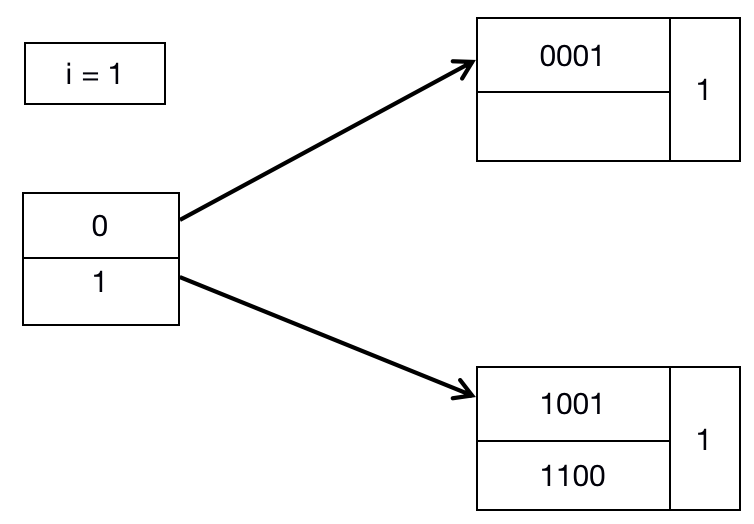
1. *If j < i, then do not need to any steps for Bucket Block, we need to:*
   1. *Divide Block B into two separate Storage Block.*
   2. *According to the value of (j + 1)th in h(K), then we need to divide the record into two separate Storage Blocks, if the (j + 1)th byte equals to 0, then it would be kept into the Storage Block B, otherwise, if the (j + 1)th byte equals to 1, then it would be saved into the new Storage Block.*
   3. *Also, save the (j + 1)th Data Value into last Block, the (j + 1)th digit number which is used to make sure the membership of Storage Block.*
   4. *Adjust the Pointer of the Whole Bucket Block, make it point to the Old Storage Block B or newly created Block which is decided by the (j + 1) byte.*

*Attention that, divide Storage Block B may can not help solve the solution, since the possible Block B may be too full and need to use the bigger j value to repeat the process.*

1. *If j = i, then we need to add 1 to value i. We double the length of Bucket Array, therefore, there would have 2 ^ (i + 1) items in the Array. Assume that the number w is used to record the ith bytes of Binary Sequence. In the new Bucket Block, the sequence number w0 and w1 can be used to point to the original Storage Bucket Block, which is to say that these two items share the same Storage Block, and the Storage Block has not changed. At last, we need to divide Storage Block B as situation 1. Now, since j > j, so it satisfies the condition 1.*

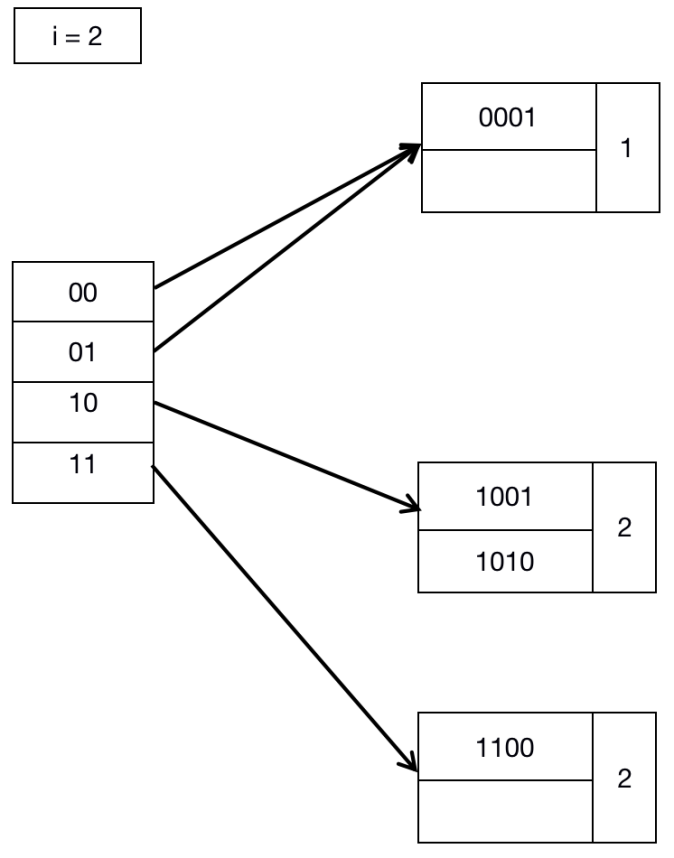
### *Example:*

If now, we need to insert the Key Value 1010 into the current Hash Table.

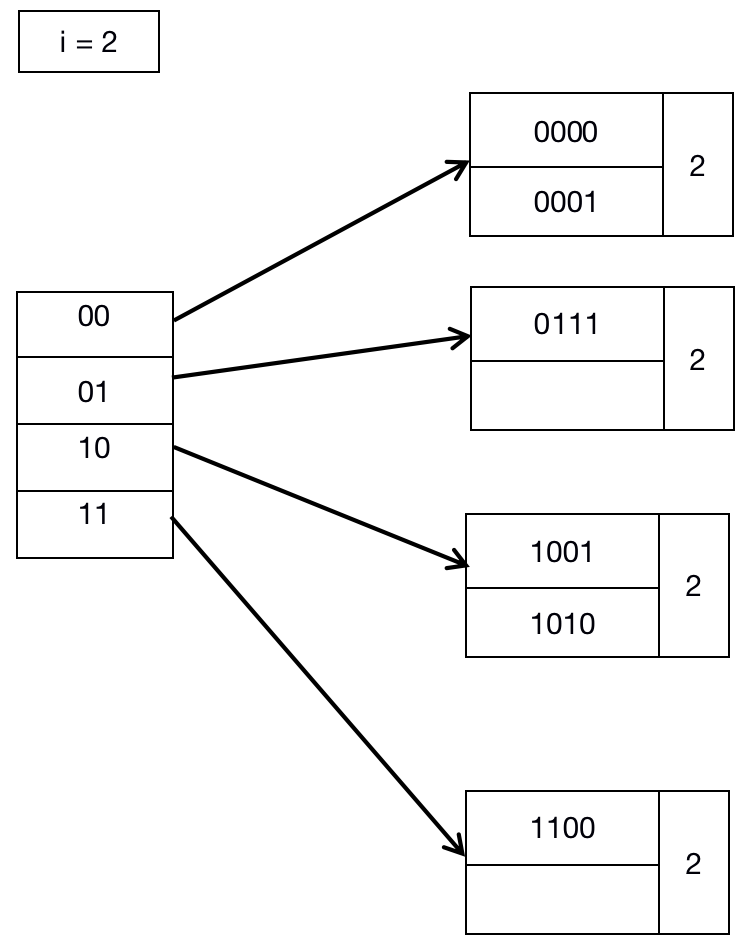


* We find that the first byte of Key Value equals to 1, then it belongs to the Second Block. Then this block has been filled with records, therefore we need to split it.
* We find that this Block is filled with records, therefore we need to double the Storage Bucket Array, in the table above, we need to set up the value of i as 2.

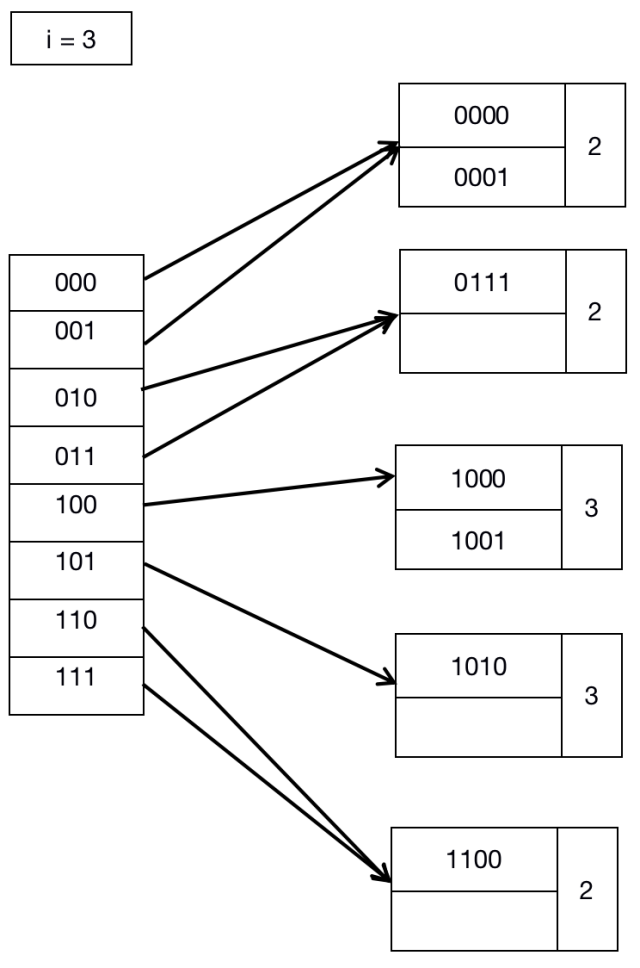
( Attention that Data Sequence starts with 0 points to the Storage Block with Key Hash Value starts with 0, and however the last square of the Storage Block is still the value of 1, this makes sure that the membership of this Storage Block is only decided by the first byte. But the record with the byte sequence starts with 1 still needs to divide, so we need to divide part of the records into two separate Storage Blocks which start with 10 and 11. Among these two Storage Block, the number value in the square equals to 2 which means that the membership can be decided by the first two bytes of Byte Sequence. Luckily, the division is successfully. Since there has at least one record in Two Blocks, so we do not need to division recursively. )



Now, we need to insert two records with Key Value of 0000 and 0111. These two records belongs to the first Storage Block. Also we check value in the last Square equals to 1, so we need to insert into the first Storage Block. We need to divide the Data Records into two parts and let 0000 and 0001 stay into Storage Block but let 0111 stored into the new Storage Block and the 01 item points to the new Storage Block. This time we are lucky that all records have not stored into one Block, so we do not need to divide recursively.



Assume that we need to insert the record with Key Value 1000 and go to the third Storage Block and find that it has already filled with two records. Since we have already known that it is decided by 2 front bytes of the Binary Sequence Number. So we need to divide the Storage Buckets again and set i equals to 3. Attention that, 10 can be divided into 100 and 101. So Hash Table has become as below.



### Chapter 3.3.7 Linear Hash Table

### *Background:*

### There have some important advantages for Extensible Hash Table. However, the most important point is that when query one record, we always need to query one Data Block. We also need to query one item of the Bucket Array, but if the Bucket Array is small enough, then it can be put into Main Memory, then we do not need any Disk I/O to visit Disk I/O. *There also have some disadvantages:*

1. *When the number of Bucket Array need to double, then there will have lots of work to do. These work may stop the access to Data File, or make some insertion cost much more time.*
2. *When the number of Bucket Array double, then it may can not stay in Main Memory, so that some extra data in the Main Memory may be moved out of Main Memory. The result is that, the great system may suddenly increase its Disk I/O of each Operation.*
3. *If the record of each Block is small, then it may possible that one division of some Block may ahead of schedule when compared with the Logic.*

### *Definition - Linear Hash:*

The increase of Linear Hash is slow. We can see the main point in the Linear Hash:

* *The selection of number of n would make ratio of the average records number of Storage Block with the sum of record as a fixed ratio, for example, such as 80%.*
* *Since the Storage Block may not always can be divided, so any overflow Storage Block is permitted, although the Average Overflow Block of each Bucket would far less than 1.*
* *[ log2n ] can be used as the Binary Sequence of the Bucket Array, among which n is the current Bucket Number. The bytes are always got from the first right place of Binary Sequence.*
* *Assume that, i bytes of Hash Function Value is used as encode of Bucket Number, and the record with Key Value K can be inserted into the Bucket numbered from a1a2a3...ai; which means that a1a2a3...ai is the i bytes from the right place of h(K). Then, we need to assign the value of a1a2a3...ai as the value of m. If m < n, then the Bucket with the number m would exists and the records can be stored into the Bucket. If n <= m < 2 ^ i, then the m Bucket doesn’t exist, so we need to store the record into the Bucket m - 2 ^ ( i - 1 ), which means we need to store the record into the Bucket with Property to set a1 from 1 to 0.*

### *Example:*

### Chapter 3.3.8 Insertion into Linear Hash Table