Module – 5

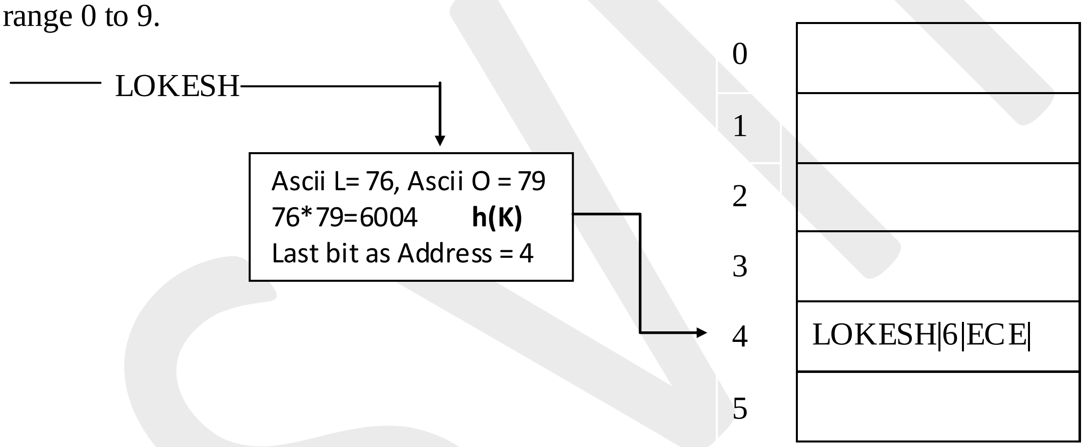
HASHING

Introduction:

Sequential access efficiency is O (N). B trees/ B+ trees efficiency is O($\log_K$ N). Both are dependent on 'N' where N is the number of records in the file i.e file size. So there was a need for a technique whose efficiency is independent of the size of the file. This lead to Hashing which provides an efficiency of O(1).

A Hash function is like a black box that produces an address every time a key is dropped. The process of producing an address when a key is dropped to hash function is called Hashing. Formally hash function is given by h(K) that transforms a key 'K' into an address. The resulting address is used to store and retrieve the record.

Example: Consider a key LOKESH which is dropped in to a hash function which does the following computation to transform the key to its address. Assume the address space is 10. i.e in the range 0 to 9.



In indexing we associate a key with relative record number (RRN). Hashing is also similar to indexing in this way. i.e associating a key to a address (relative record number). But hashing differs from indexing in two major ways. They are:

1. With hashing the address generated appear to be random. In indexing association of key with RRN is sequential whereas in hashing this association is random.
2. In hashing two different keys may be associated (Hashed) to same address. This is called collision. Whereas in indexing there is no collision. i.e there is no chance of two different keys being associated to same RRN.

The ideal solution is to find or devise a transformation algorithm that avoids collision altogether. Such an algorithm is called a perfect hashing algorithm. It is near to impossible to find a perfect hashing algorithm. So a more practical solution is to reduce the number of collision to an acceptable level. Some of the ways to reduce the collusion to acceptable level are:

1. Spread out the records.
2. Use extra memory.
3. Put more than one record at a single address.

Spread out the records: Collision occur when two or more records compete for the same address. If we could find hashing algorithm that distributes the records fairly randomly among the available addresses, then we would not have large number of records clustering around certain address.

Use extra memory: It is easier to find a hash algorithm that avoids collisions if we have only a few records to distribute among many addresses than if we have about the same number of records as addresses. Example chances of collisions are less when 10 records needs to be distributed among 100 address space. Whereas chances of collision is more when 10 records needs to be distributed among 10 address space. The obvious disadvantage is wastage of space.

Put more than one record at a single address: Create a file in such a way that it can store several records at every address. Example, if each record is 80 bytes long and we create a file with 512 byte physical records, we can store up to six records at each address. This is called as bucket technique.

A Simple Hashing Algorithm

This algorithm has three steps:

1. Represent the key in numerical form.
2. Fold and add.
3. Divide by a prime number and use the reminder as the address.

Step 1: Put more than one record at a single address.

If the key is string of characters take ASCII code of each character and use it to form a number. Otherwise no need to do this step.

Example if the key is LOWELL we will have to take ASCII code for each charaters to form a number. If the key is 550 then no need to do this step. We will consider LOWELL as the key.

LOWELL – 76 79 87 69 76 76 32 32 32 32 32 32

32 is ASCII code of blank space. We are padding the key with blank space to make it 12 characters long in size.

Step 2: Fold and add.

In this step Folding and adding means chopping off pieces of the number and add them together. Here we chop off pieces with two ASCII numbers each as shown below:

7679 | 8769 | 7676 | 3232 | 3232 | 3232 ------------- Folding is done

On some microcomputers, for example, integer values that exceed 32767 causes overflow errors or become negative. So addition of the above integer values results in 33820 causing an overflow error. Consequently we need to make sure that each successive sum is less than 32767. This is done by first identifying the largest single value we will ever add in summation and making sure after each addition the intermediate result differs from 32767 by that amount.

Assuming that keys contain only blanks and upper case alphabets, so the largest addend is 9090 corresponding to ZZ. Suppose we choose 19937 as our largest allowable intermediate result. This differs from 32767 by much more than 9090, so no new addition will cause overflow.

Ensuring of intermediate result not exceeding 19937 is done by using mod operator, which returns the reminder when one integer is divided by another.

7679 + 8769 = 16448 => 16448 mod 19937 => 16448
16448 + 7676 = 24124 => 24124 mod 19937 => 4187
4187 + 3232 = 7419 => 7419 mod 19937 => 7419
7419 + 3232 = 10651 => 10651 mod 19937 => 10651
10651 + 3232 = 13883 => 13883 mod 19937 => 13883

The number 13883 is the result of the fold and add operation.

Step 3: Divide by the Size of the Address Space.

In this step the number produced by the previous step is cut down so that it falls in the within the range of addresses of records in the file. This is done by dividing the number by address size of the file. The remainder will be home address of record. It is given as shown below

$$a = s \bmod n$$

If we decide to have 100 addresses i.e 0-99 than a= 13883 mod 100, which is equal to 83. So hash address of LOWELL will be 83.

Hashing Functions & Record Distribution:

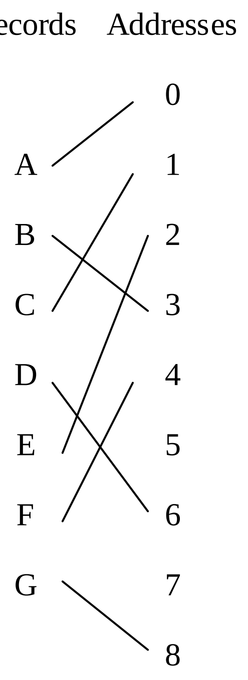
The figure below shows three different distributions of seven records among nine addresses.

If the hash function distributes seven records in ten addresses without any collision then it is called as **uniform distribution** as shown in figure '**a**'.

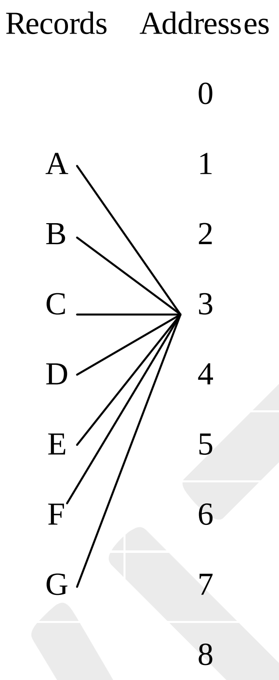
If all the seven records are hashed to the same home address then it is called as **worst distribution** as shown in the figure '**b'**

Figure '**c**' illustrates a distribution in which the records are somewhat spread out, but with a few collisions. This is called as **Acceptable distribution**.

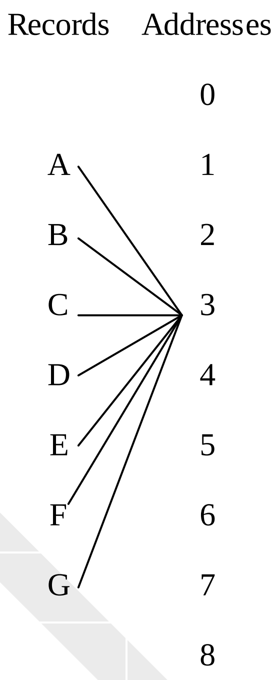
Uniform Distribution



Worst Distribut ion



Acceptable Distribution



Some Other Hashing Methods

Some methods those are potentially better than random. The simple hashing algorithm explained in section 7.2 which has three steps are designed to take advantage of natural ordering among the keys. The next two methods can be tried when for some reasons, the better than random methods do not work.

- **Square the key and take the mid (Mid Square method):** This method involves treating the key as single large number, squaring the number and extracting whatever number of digits are required from the middle of the result.
  For example: consider the key 453, its square is $(453)^2$ = 205209. Extracting the middle two digits yields a number 52 which is between 0 – 99.
- **Radix Transformation:** This method involves converting the key from one base system to another base system, then dividing the result with maximum address and taking the reminder**.**
  For example: If the hash address range is 0 – 99 and key is $(453)_{10.}$. Converting this number to base 11 system results in $(382)_{11}$. Then 382 mod 99 = 85. So 85 is the hash address.

7.3.2 Predicting the Distribution of Records.

7.3.3 Predicting Collision for a Full File.

7.4 How Much Extra Memory should be Used.

Refer class notes & problems solved in the class

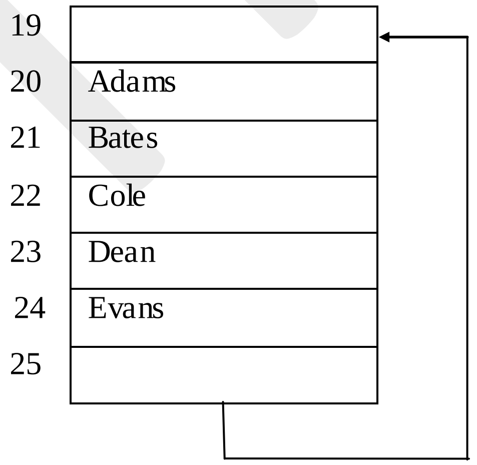
Collision Resolution by Progressive Overflow

- **Progressive overflow** is a collision resolution technique which places overflow records at the first empty address after the home address; if the file is full the search comes back to where it began. Only then it is clear that the file is full and no empty address is available.
- With progressive overflow, a sequential search is performed beginning at the home address and if end of the address is reached, then wrap around is done and the search continuous from the beginning address in the file.
- The search is continued until the desired key or a blank record is found.
- If the file is full the search comes back to where it began. Only then it is clear that the record is not in the file.
- Progressive overflow is also referred to as *linear probing.*

Consider the table given below:

Key	**Home address**
Adams	20
Bates	21
Cole	21
Dean	22
Evans	20

Table 1.



Key	Home address	**Actual Address**	**Search Length**
Adams	20	**20**	**1**
Bates	21	**21**	**1**
Cole	21	**22**	**2**
Dean	22	**23**	**2**
Evans	20	**24**	**5**

Average search length is $\frac{1+1+2+2+5}{5} = 2.2$

Making Deletions

Now suppose record Dean is deleted which makes the address 23 empty. When the key Evans is searched it starts from home address of the i.e 20 and when it reaches address 23, which is empty the search stops and displays that key not present. But actually the key is present in the file. So deletion of records leads to this type of problems. Using the concept of Tombstone technique this limitation can be overcome. The figure below depicts the above mentioned scenario.

19	
20	Adams
21	Bates
22	Cole
23	######
24	Evans
25	

In tombstone technique whenever a record is deleted that address space is marked by a special character. So whenever a special character is encountered the search continuous to next address without stopping at that point only. Further these marked addresses can be reused or reclaimed to store the new record.

Implications of Tombstone for Insertions.

Now suppose a new record with key Evans needs to be stored. When the key Evans is dropped to hash function it produces an address of 20. Since address 20 is already occupied by Adams according to progressive overflow technique the search for empty address slot starts and when it reaches address 23 the New Evans is inserted. But the problem is already there is a record for Evans i.e duplicate records are inserted which is certainly not acceptable. So to avoid we have to search the entire address space if already a key with same name is present or not. Only then the new record can be inserted. This in turn slows down the insertion operation.

Storing more than One Record per Address: Buckets

- **Bucket:** An area of a hash table with a single hash address which has room for more than one record.
- When using buckets, an entire bucket is read or written as a unit. (Records are not read individually).
- The use of buckets will reduce the average number of probes required to find a record.

Address	Counter		Records	
19	0			
20	2	Adams	Evans	
21	2	Bates	Cole	

22	1	Dean		
23	0			

- There should be a counter to keep track of how many records are already stored in any given address. The figure above represents storage of records with bucket size of 3.

Other Collision Resolution Techniques: Double Hashing

- Double hashing is similar to progressive overflow.
- The second hash value is used as a stepping distance for the probing.
- The second hash value should never be one. (Add one.)
- The second hash value should be relatively prime to the size of the table. (This happens if the table size is prime.)
- If there is collision at the hash address produced by $h_1(k)$ = Say **'X',** then the key is dropped to second hash function to produce address **'C'**.
- The new record is stored at the address **'X+C'.**
- A collision resolution scheme which applies a second hash function to keys which collide, to determine a probing distance 'C'.
- The use of double hashing will reduce the average number of probes required to find a record.

Chained Progressive Overflow

- Chained progressive overflow forms a linked list, or chain, of synonyms.
- Each home address contains a number indicating the location of the next record with the same home address.
- The next record in turn contains a pointer to the other record with the same home address.
- This is shown in the figure below: In the figure below Admans contain the pointer to Cole which is synonym. Then Bates contain pointer to Dean which are again synonym. (Consider the below given Table)

Key	**Home Address**
Adams	20
Bates	21
Cole	20
Dean	21
Evans	24
Flint	20

The figure below represents the chained progressive overflow technique.

Home address	Actual Address	Records	Address of next Synonym	Search Length
	19	.	.	.
20	20	Adams	**22**	**1**
21	21	Bates	**23**	**1**
20	22	Cole	**25**	**2**
21	23	Dean	**-1**	**2**
24	24	Evans	**-1**	**1**
20	25	Flint	**-1**	**3**
	26		.	.
			.	.

- Now suppose if the Dean's home address was 22 instead of 21. By the time Dean is loaded, address 22 is already occupied by Cole, so Dean ends up at address 23. Does this mean that Cole's pointer should point to 23 (Dean's actual address) or to 25 (the address of Cole's synonym Flint)? If the pointer is 25, the linked list joining Adams, Cole, Flint is kept intact, but Dean is lost. If the pointer is 23 Flint is lost.

The problem here is that a certain address (22) that should be occupied by a home record (Dean is occupied by a different record. Solution to this type of problem are:

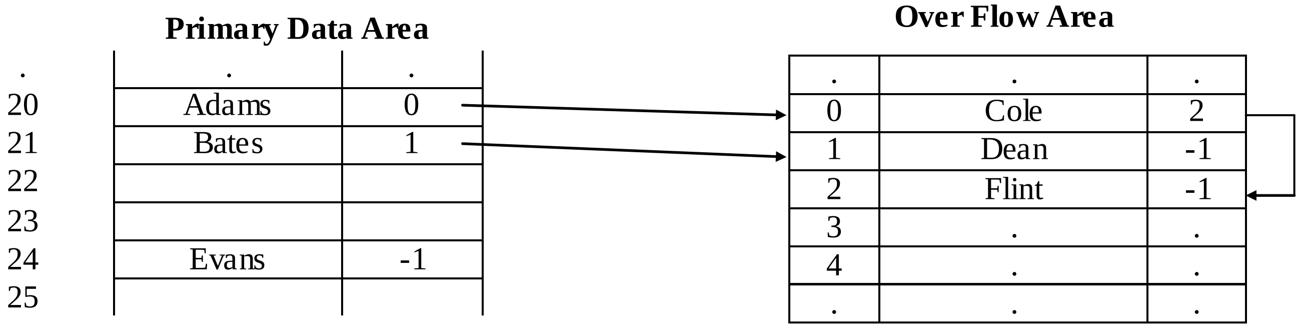
1. Two pass loading.
2. Chaining with a separate overflow area.

Two Pass Loading

- As the name implies, involves loading a hash file in two passes. On the first pass, only home records are loaded.
- All records that are not home records are kept in separate file.
- On the second pass, each overflow record is loaded and stored in one of the free addresses according to whatever collision resolution technique is being used.

Chaining with a separate overflow area.

- One way to keep overflow records from occupying home addresses where they should not be is to move them all to a separate overflow area.
- The set of home addresses is called prime data area, and the set of overflow addresses is called the overflow area.
- With respect to the previous example the records for Cole, Dean and Flint are stored in a separate overflow area as shown in the figure below.
- Whenever a new record is added if its home address is empty, it is stored in primary storage area. Otherwise it is moved to overflow area, where it is added to a linked list that starts at home address.



Primary Data Area

.	.	.
20	Adams	0
21	Bates	1
22		
23		
24	Evans	-1
25		

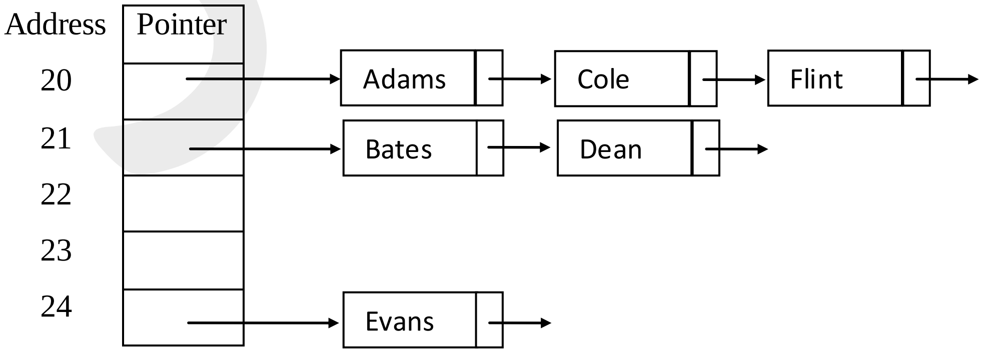
Over Flow Area

.	.	.
0	Cole	2
1	Dean	-1
2	Flint	-1
3	.	.
4	.	.
.	.	.

Scatter Tables

- If all records are moved into a separate "overflow" area, with only links being left in the hash table, the result is a *scatter table*.
- In scatter table records are not stored at the hash address.
- Only pointer to the record is stored at hash address.
- Scatter tables are similar to index but they are searched by hashing rather than some other method which is used in case of indexing.
- Main advantage of scatter tables is they support use of variable length records.

Key	**Home Address**
Adams	20
Bates	21
Cole	20
Dean	21
Evans	24
Flint	20



Hash File

Extendible Hashing

12.2 How Extendible Hashing Works

Tries

The key idea behind extendible hashing is to combine conventional hashing with another retrieval approach called the trie. Tries are also sometimes referred to as *radix searching* because the branching factor of the search tree is equal to the number of alternative symbols (the radix of the alphabet) that can occur in each position of the key.

Suppose we want to build a trie that stores the keys *able, abrahms, Adams, anderson, andrews,* and *Baird.* A schematic form of the trie is shown in Fig. 12.1. As you can see, the searching proceeds letter by letter through the key. Because there are twenty-six symbols in the alphabet, the potential branching factor at every node of the search is twenty-Six.

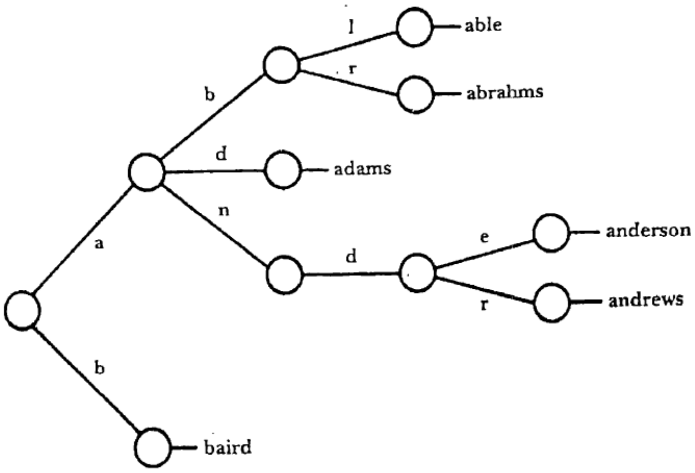


Figure 12.1

Notice that in searching a trie we sometimes use only a portion of the key. We use more of the key as we need more information to complete the search. This use-more-as-we-need-more capability is fundamental to the structure of extendible hashing.

12.2.2 Turning the Trie into a Directory

We use tries with a radix of 2 in our approach to extendible hashing: search decisions are made on a bit-by-bit basis. We will not work in terms of individual keys but in terms of *buckets* containing keys.

Suppose we have bucket A containing keys that, when hashed, have hash addresses that begin with the bits *01.* Bucket B contains keys with hash addresses beginning with *10* and bucket C contains keys with addresses that start with 11. Figure 12.3 shows a trie that allows us to retrieve these

buckets.

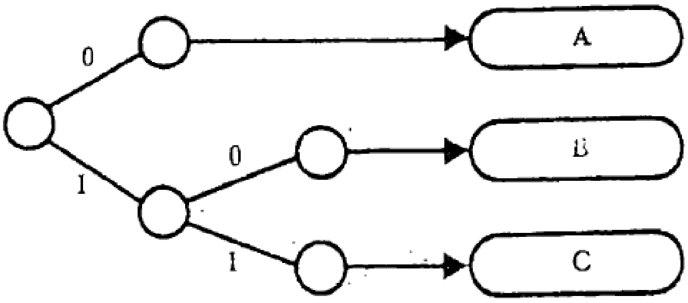


Figure 12.3

How should we represent the trie?

Rather than representing the trie as a tree, we flatten it into an array of contiguous records, forming a directory of hash addresses and pointers to the corresponding buckets.

Step 1: The first Step in turning a tree into an array involves extending it so it is a complete binary tree with all of its leaves at the Same level as shown in Fig. 12.4 (a). Even though the initial 01s enough to select bucket A, the new form of the tree also uses the second address bit so both alternatives lead to the same bucket.

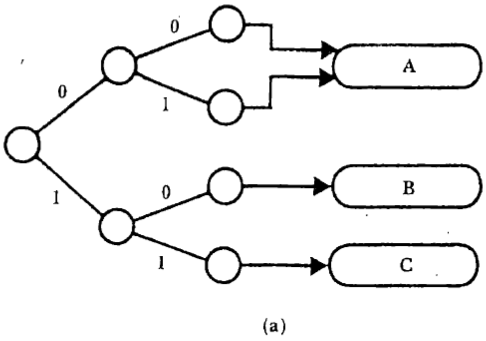


Figure 12.4 (a)

Step 2: Once we have extended the tree this way, we can collapse it into the d.irectory structure shown in Fig. 12.4(b). Now we have a Structure that provides the kind of direct access associated with hashing: given an address beginning with the bits *10,* the 1O2th directory entry gives us a pointer to the associated bucket.

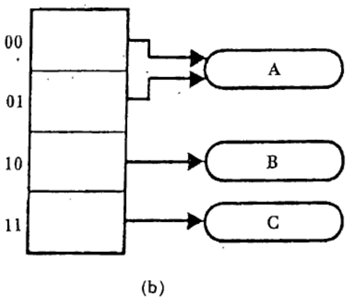


Figure 12.4(b)

12.2.3 Splitting to Handle Overflow

The goal in an *extendible* hashing system is to find a way to increase the address space in response to overflow.

Suppose we insert records that cause bucket A in fig. 12.4(b) to overflow. In this case the solution is Simple: since addresses beginning with *00* and *01* are mixed together in bucket A, we, can Split bucket A by putting all the *01* addresses in a new bucket D, while keeping only the *00* addresses in A.

We must use the full 2 bits to divide the addresses between two buckets. We do not need to extend the address $pace; we simply make full use of the address information that we already have. Figure 12.5 shows the directory and buckets after the split.

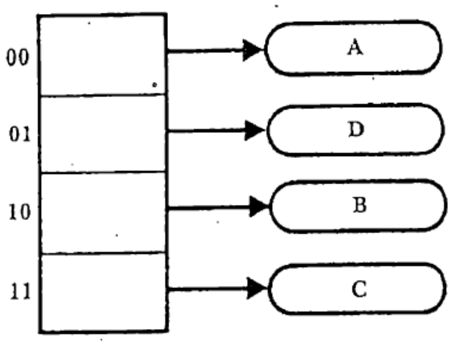


Figure 12.5

Let's consider a more complex case, starting once again with the directory and buckets in Fig. l2.4 (b), suppose that bucket B overflows. How do we split bucket B and where do we attach the new bucket after the split? Unlike our previous example, we do not have additional, unused bits of address space that we can press into duty as we split the bucket we now need to use 3 bits of the hash address in order to divide up the records that hash to bucket B. The trie illustrated in Fig. 12.6 (a) makes the distinctions required to complete the split. Figure 12.6 (b) shows what this trie looks like once it is extended into a completely full binary tree with all leaved at the same level, and Fig. 12.6 (c) shows the collapsed, directory form of the trie.

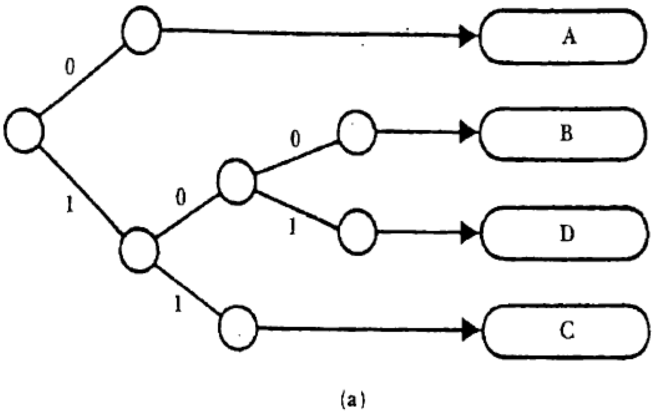
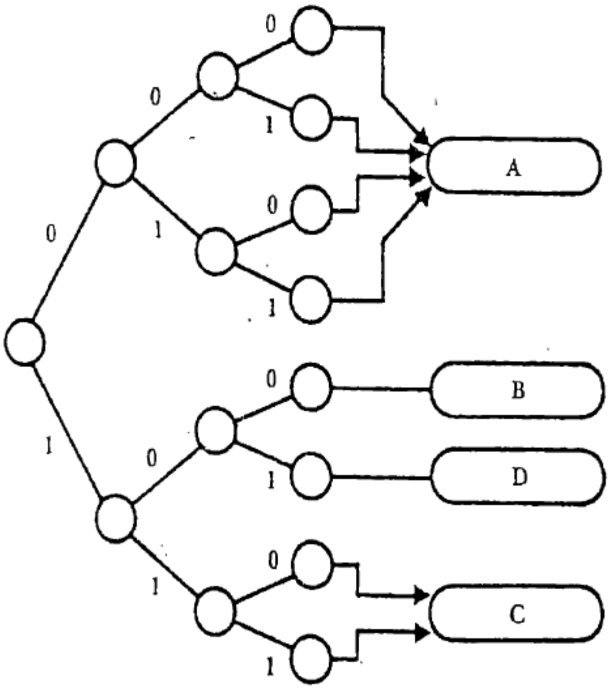
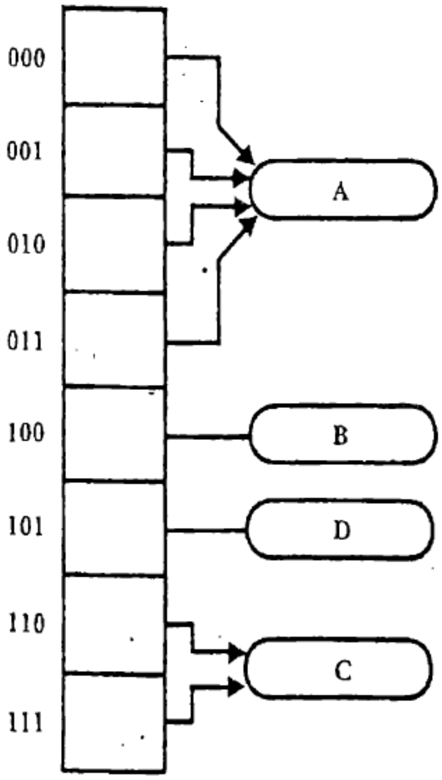


Figure 12.6 (a)



(b)

Figure 12.6 (b)



(c)

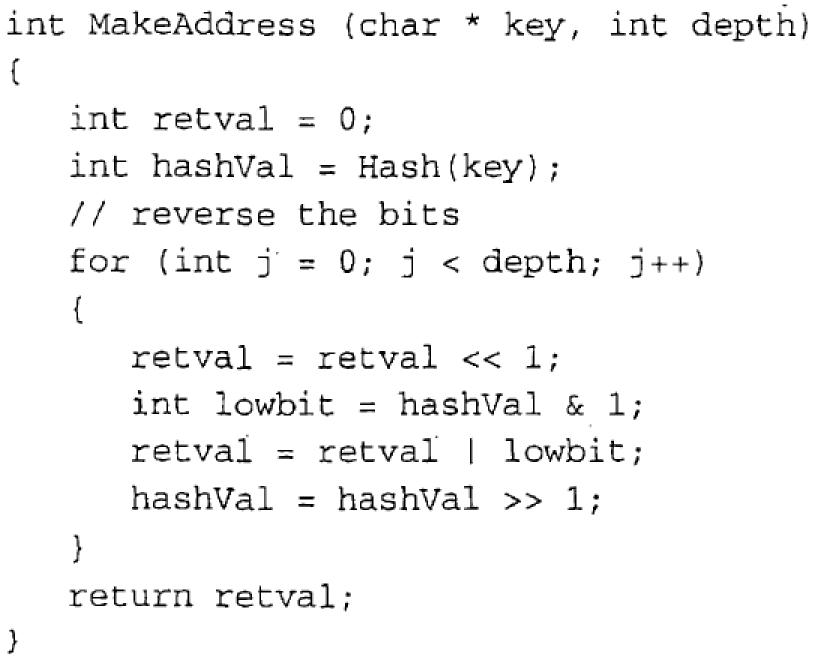
Figure 12.6 (c)

Implementation

Creating the address

```
int Hash (char* key)
{
        int sum = 0;
        int len = strlen(key);
        if (len % 2 == 1) len ++; // make len even
        // for an odd length, use the trailing '\0' as part of key
        for (int j = 0; j < Jen; j+=2)
                sum = (sum + 100 * key[j] + key[j+I]) % 19937;
        return sum;
}
```

The function Hash is a simple variation on the fold-and-add hashing algorithm. Because extendible hashing uses more bits of the hashed address as they are needed to distinguish between buckets, we need a function MakeAddress that extracts a portion of the full hashed address. We also use MakeAddress to reverse the order of the bits in the hashed address, making the lowest-order bit of the hash address the highest-order bit of the value used in extendible hashing. By reversing the bit order, working from right to left, we take advantage of the greater variability of low-order bit values.



```
int MakeAddress (char * key, int depth)
{
   int retval = 0;
   int hashVal = Hash(key);
   // reverse the bits
   for (int j = 0; j < depth; j++)
   {
      retval = retval << 1;
      int lowbit = hashVal & 1;
      retval = retval | lowbit;
      hashVal = hashVal >> 1;
   }
   return retval;
}
```

Deletion

Overview of the Deletion Process

If extendible hashing is to be a truly *dynamic* system like B-trees or AVL trees, it must be able to *shrink* files gracefully as well as grow them.

When do we combine buckets? This question, in turn, leads us to ask which buckets can be combined? In extendible hashing we use a similar concept: buckets that are *buddy* buckets. Look again at the trie in Fig. 12.6 (b). Which buckets could be combined? Trying to combine anything with bucket A would mean collapsing everything else in the trie first. Similarly, there is no single bucket that could be combined with bucket C. But buckets B and D are in the same configuration as buckets that have just split.` They are ready to be combined: they are buddy buckets.

After combining buckets, we examine the directory to see if we can make changes there. Looking at the directory form of the trie in Fig. 12.6 (c), we see that once we combine buckets B and D, directory cells 100 and *101* both point to the same bucket. In fact, each of the buckets has at least a pair of directory cells pointing to it. In other words, none of the buckets requires the depth of address information that is currently available in the directory. That means that we can shrink the directory and reduce the address space to half its size.

Reducing the size of the address space restores the directory and bucket structure to the arrangement shown in Fig. 12.4, before the additions and splits that produced the structure in Fig.12.6 (c). Reduction consists of collapsing each adjacent pair of directory cells into a single cell. This is easy, because both cells in each pair point to the same bucket. Note that this is nothing more than a reversal of the directory splitting procedure that we use when we need to add new directory cells.

A Procedure for Finding Buddy Buckets

The method works by checking to see whether it is possible for there to be a buddy bucket. Clearly, if the directory depth is 0, meaning that there is only a single bucket, there cannot be a buddy. The next test compares the number of bits used by the bucket with the number of bits used in the directory address space. A pair of buddy buckets is a set of buckets that are immediate descendants of the same node in the trie. It is only when a bucket is at the outer edge of the trie that it can have a single parent and a single buddy.

Once we determine that there is a buddy bucket, we need to find its address. First we fin. d the address used to find the bucket we have at hand;

```
int Bucket::FindBuddy ()
{// find the bucket that is paired with this
      if (Dir.Depth == O) return -I; // no buddy, empty directory
```

```
   // unless bucket depth == directory depth, there is no single
   // bucket to pair with
   if (Depth < Dir.Depth) return -1;
   int sharedaddress = MakeAddress(Keys[0], Depth);
      // address of any key
   return sharedaddress ^ I; // exclusive or with low bit
}
```

Since we know that the buddy bucket is the other bucket that was formed from a split, we know that the buddy has the same address in all regards except for the last bit. So, to get the buddy address, we flip the last bit with an exclusive or.

Collapsing the Directory

The other important support function used to implement deletion is the function that handles collapsing the directory. Collapsing directory begins by making sure that we are not at the lower limit of directory size. The test.to see if the directory can be collapsed consists of examining each pair of directory cells to see if they point to different buckets. As soon as we find Such a pair, we know that- we *cannot* collapse the directory. If we get all the way through the directory without encountering such a pair, then we can collapse the directory.

The collapsing operation consists of` allocating space for a new array of bucket addresses that is half the size of the original and then copying the bucket references shared by each cell pair to a single cell in the new directory.

Alternative Approaches

Dynamic Hashing

Functionally, dynamic hashing and extendible hashing are very similar. Both use a directory to track the addresses of the buckets, and both-extend the directory through the use of tries.

The key difference between the approaches is that dynamic hashing, like conventional, static hashing, starts with a hash function that covers an address space of a fixed Size.

As buckets within that fixed address space overflow, they split, forming the leaves of a trie that grows down from the original address node.

Eventually, after enough additions and splitting, the buckets are addressed through a forest of tries that have been seeded out of the original static address space.

Let's look at an example. Figure 12.23 (a) Shows an initial address space of four and four buckets descending from the four addresses in the directory.

In Fig. 12.23 (b) we have split the bucket at address 4. We address the two buckets resulting from the split as 40 and 41. We change the shape of the directory node at address 4 from a square to a circle because it has changed from an external node

In .Fig. 12.23 (c) we split the bucket addressed by node 2, creating the new external nodes *20* and *21*. We also split the bucket addressed by *41*, extending the trie downward to include *410* and *41 1.*

Finding a key in a dynamic hashing scheme can involve the use of two hash functions rather than just one. First, there is the hash function that covers the original address space. If you find that the directory node is an external node and therefore points to a bucket, the search is complete. However, if the directory node is an internal node, then you need additional address information to guide you through the ls and 0s that form the trie.

The primary difference between the two approaches is that dynamic hashing allows for slower, more gradual growth of the directory, whereas extendible hashing extends the directory by doubling it.

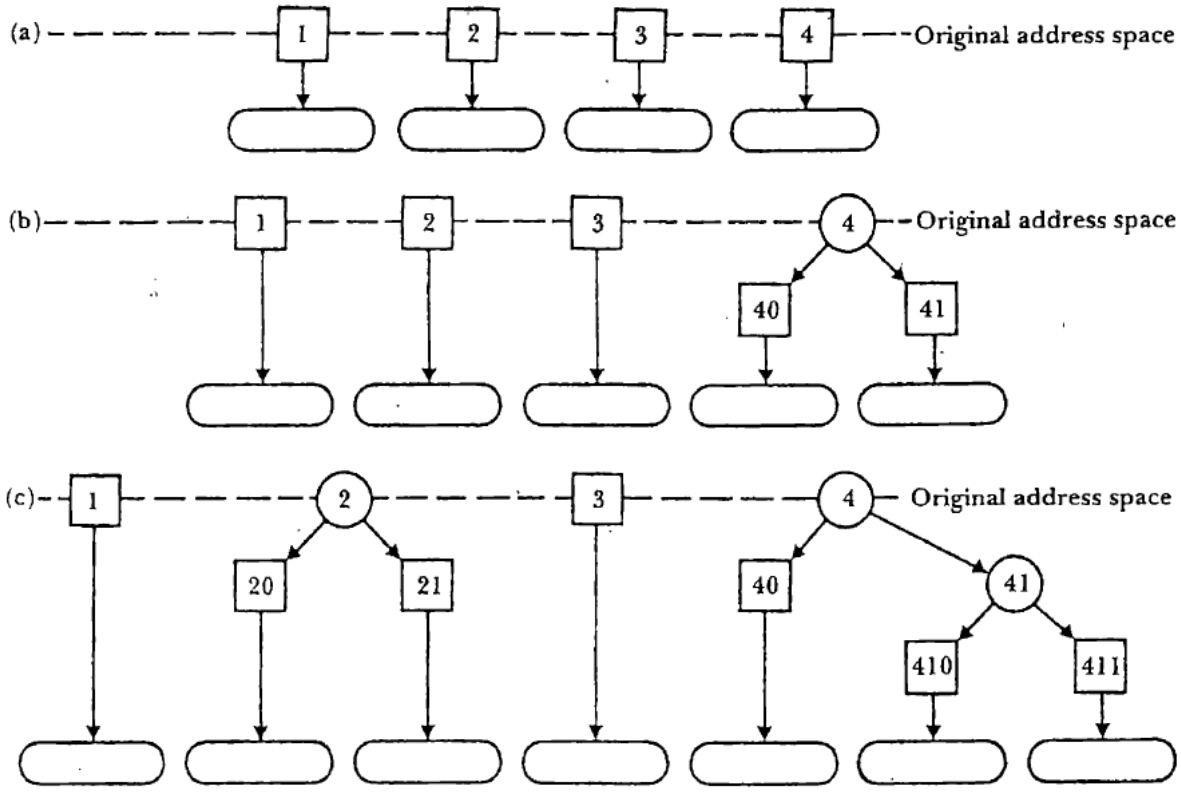


Figure 12.23

Linear Hashing

Linear hashing, introduced by Litwin in 1980, does away with the directory. Linear hashing, like extendible hashing, uses more bits of hashed value as the address space grows. Note that the address space consists of four *buckets* rather than four directory nodes that can point to buckets.

As we add records, bucket b overflows. The overflow forces a split. However, as Fig. 12.24(b) Shows, it is not bucket b that splits, but bucket a. The reason for this is that we are extending the address space *linearly,* and bucket a is the next bucket that must split to create the next linear extension, which we call bucket A. A 3-bit hash function, h3(k), is applied to buckets a‘and A to divide the records between them. Since bucket b was not the bucket that we split, the overflowing record is placed into an overflow bucket w.

We add more records, and bucket d overflows. Bucket b is the next one to split and extend the address space, so we use the h3(k) address function to divide the records from bucket b and its overflow bucket between b and the new bucket B. The record overflowing bucket d is placed In an overflow bucket x. The resulting arrangement is illustrated in Fig. 12.24(c).

Figure 12.24 (d) shows what happens when, as we add more records, bucket d overflows beyond the capacity of the overflow bucket w. Bucket c is the next in the extension sequence, so we use the h3(k)address function to divide the records between c and C.

Finally, assume that bucket B overflows. The overflow record is placed in the overflow bucket z. The overflow also triggers the extension to bucket D, dividing the contents of d, x, and y between buckets d and D. At this point all of the buckets use the h3(k) address function, and we have finished the expansion cycle. The pointer for the next bucket to be split returns, to bucket a to get ready for a new cycle that will use an h4(k) address function to reach new buckets.

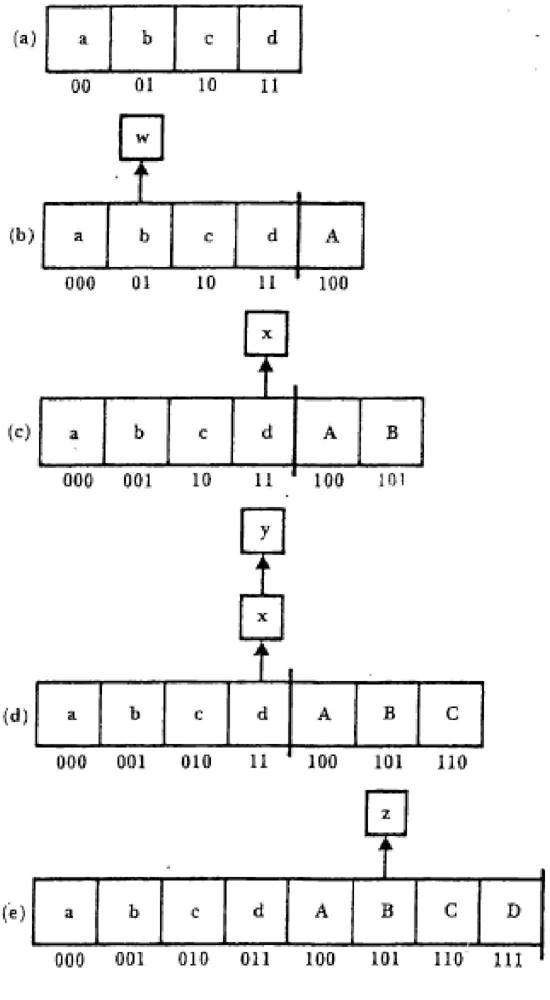


Figure 12.24