



University of Dublin
Trinity College



Hashing Techniques

Owen.Conlan@cs.tcd.ie

How does Hashing work?

We apply some mathematical function to the key to generate a number in the range of record numbers

It is a function, so a given key always maps to the same address

- For example, we might take the ASCII representation of the first two characters in the name, multiply the two numbers together, and use the three rightmost digits of the name as the address

Name	ASCII Code	Product	Address
BALL	66 65	4290	290
LOWELL	76 79	6004	004
SPARK	80 65	5200	200

Collisions

Suppose there is a key in the sample file with the name OLIVIER

- Since OLIVIER starts with the same two letters as the name LOWELL, they produce the same address
- The records collide in position 200

How do we avoid such collisions?

- At first one might try to find a hash function that avoids collisions altogether – a perfect hash function
- However it is almost impossible to achieve unless the keys are allocated deliberately to avoid collisions
 - *Hanson 1982 showed that for a system with 4000 records and 5000 addresses, only one out of $10^{120,000}$ hash algorithms could avoid collisions altogether*

What do we do about Collisions?

Find a better hashing algorithm

- Collisions occur when two or more records compete for the same address. Therefore we should try to find a hashing algorithm that distributes records fairly evenly among the available addresses

Use a bigger table

- The more free slots in the table, the less likely there will be a collision. But if you are doing lots of accesses, using a bigger table will reduce the likelihood that two accesses will reference the same part of the disk

Need a system to deal with collisions

Simple Hashing Algorithm

Our goal in choosing any hashing algorithm is to spread out the records as uniformly as possible over the range of addresses available

Our previous algorithm did not do this very well – it only uses two letters of the key and doesn't do much with them

Let's look at a better algorithm. It has three steps:

1. Represent the key in numerical form
2. Fold and add
3. Divide by the size of the address space and use the remainder as an address

Step 1: Represent the key in numerical form

We take the ASCII code of each character and use it to form a number. For example:

LOWELL = 76 79 87 69 76 76 32 32 32 32 32 32 32
 L O W E L L | blanks |

In this algorithm we use the entire key rather than the first two letters. By using more parts of the key, we increase the likelihood that differences in the keys will cause differences in addresses produced

Step 2: Fold and add

Folding and adding means chopping the number into pieces and adding them together. In our algorithm, we are chopping the key into pairs of ASCII numbers:

76 79 | 87 69 | 76 76 | 32 32 |

Once our key (LOWELL) is in numeric form, we can do arithmetic on it

(e.g.) $7679 + 8769 + 7676 + \dots = 33,820$

Step 3: Divide by the size of the address space

The purpose of this step is to cut down the size of the number produced in the fold and add operation, so that it falls within the range of record numbers in the file

This can be done by dividing that number by the address size of the file and taking the remainder which will be the address of the record

Let **s** represent the sum produced (i.e 33,820)

Let **n** represent the number of addresses

Let **a** represent the address we are trying to produce

$$a = s \bmod n$$

For example, if n is 100, then $a = 33,820 \bmod 100 = 20$

A Simple Hashing Algorithm

A prime number is usually used as the divisor, since primes tend to distribute remainders much more uniformly than non-primes

- This means that the size of the address space is a prime number
- Say we choose 101 as our divisor. We get

$$a = 33820 \bmod 101 = 86$$

- So LOWELL is assigned to record number 86 in the file

Pseudo Code

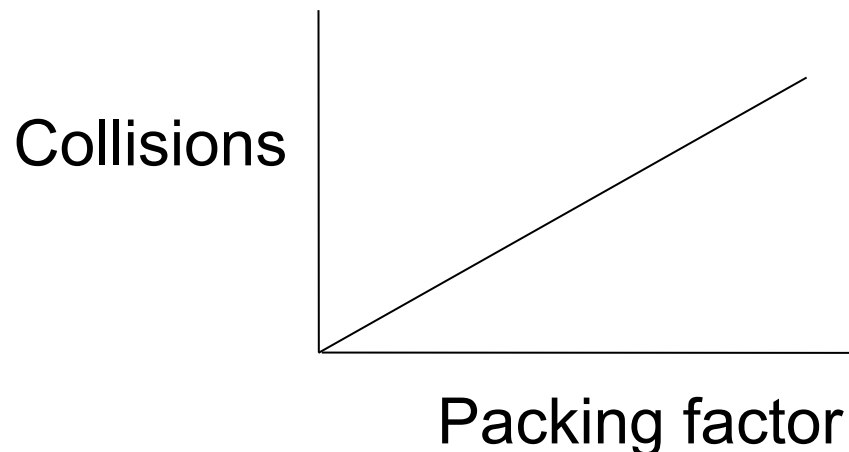
The function takes two inputs, **key**, which is an array of ASCII codes and **table_size** which is the size of the address space

```
function hash(key, table_size)  
    sum = 0; i = 0;  
    while ( i < 12 )  
        sum = sum + key[i] * 100 + key[i + 1];  
        i = i + 2;  
    end while  
    return sum mod table_size  
end function
```

Increasing the File Size

The more free slots in the hash table, the less likely there will be a collision

- The **packing factor** is the ratio of the number of records stored to the capacity of the file.. Number of records/number of spaces
 - E.g. 75 records and 100 addresses => packing factor of 75%
- As the packing factor increases, the likelihood of a collision increases



Assuming that keys are hashed randomly, with one key per slot in the table, this should be linear.

In practice, biases in the data and collisions resolution schemes mean that it is not

Problems with increasing file size

A problem with increasing the file size a lot, is that it may consist mostly of empty space

Another (smaller) problem is locality. The bigger the file, the less likely that subsequent searches will access the same part of the disk

A common strategy is to start with a small hash table and increase the size when necessary if it starts to get full

- Create a new larger hash table
- Remove records from old table and insert them into the new one
- Delete the old table

Dealing with Collisions

When a collision occurs, we need to find somewhere to put one of the records

There are several schemes

- Buckets
- Open Addressing
 - *Linear Probing*
 - *Double Hashing*
 - *Dynamic Schemes*
- Chaining

Buckets

A common strategy is to have space for more than one record at each location in the hash table

Each location contains a bucket (or block or page) of records

Each bucket contains a fixed number of records, known as the **blocking factor**

The size of each bucket is set to the size of the block of data that is read in on each disk read

Reading in a whole bucket takes one disk access

Buckets

Once the bucket has been read in, we then search for the record within the bucket, using some standard searching strategy (perhaps even hashing if the bucket is large)

Searching within the bucket is cheap, because it is within main memory

By placing several colliding records in the same bucket, we can make sure that when collisions occur, as many colliding records as possible can be found with a single disk access

Problems with Buckets

Ultimately, buckets don't solve the problem since their size is limited

Eventually, several collisions in the same location will cause a bucket to become full and overflow

- Some other scheme is needed to deal with overflowing buckets

But, buckets are combined with other schemes to ensure that when we get collisions, we won't have unnecessary disk accesses, due to not reading in as many colliding records as possible with a single disk access

Linear Probing

When there is a collision open addressing schemes find another place in the hash table where the record can be placed

The simplest open addressing scheme is linear probing

- If a collision occurs when inserting data, place the record in the next empty slot
- Do a sequential search of the table, until an slot is found
- If we reach the end of the table, continue the search at the start. The table "wraps around".

Linear Probing

We want to place records with the following keys
27, 18, 29, 28, 39, 13, 16
into a file with eleven slots using the hash function
 $\text{Hash}(\text{key}) = \text{key} \bmod 11$

29 collides with 18 (at 7), 28 hashes to location 6
so put in next free space (8) without collision

39 hashes to 6,
and collides with
28. Next free
location is found at
9.

0	
1	
2	
3	
4	
5	27
6	
7	18
8	
9	
10	

0	
1	
2	
3	
4	
5	27
6	
7	18
8	29
9	
10	

0	
1	
2	
3	
4	
5	27
6	28
7	18
8	29
9	
10	

0	
1	
2	
3	
4	
5	27
6	28
7	18
8	29
9	39
10	

Linear Probing

0	
1	
2	13
3	
4	
5	27
6	28
7	18
8	29
9	39
10	

Key 13 hashes to location 2 and causes no collision

0	
1	
2	13
3	
4	
5	27
6	28
7	18
8	29
9	39
10	16

Key 16 hashes to location 5 and collides with record 27.

We try record 6, but it collides with key 28.

It collides with keys, 18, 29 and 39, before we finally find an empty space in slot 10 of the table.

To retrieve record 16, we need six probes!

We get a lot of collisions in the same area of the table

Linear Probing

Primary clustering occurs when a lot of keys hash to the same, or near address

Secondary clustering happens when our collision resolution scheme tends to try to put several records that have collided in the same address

- Some primary clustering is difficult to avoid, since the keys are rarely random
- But linear probing makes the situation much worse, by putting records that have collided all into the same small part of the file
- There is a "domino effect" where one collision can lead to clustering, which leads to further collisions, further clustering and so forth

Deleting Records

We can't just delete a record that has been involved in a collision

- We delete record 18 from location 7
- We are now searching for record 29, which also hashes to location 7.
- But location 7 is empty. How do we know that 29 is there? Will we always have to search the whole hash table, just in case it is in there somewhere beyond some long sequence of blanks and records?

0	
1	
2	13
3	
4	
5	27
6	28
7	18
8	29
9	39
10	16

Solution

The usual solution to this problem is to place a "deleted" marker in the empty record to say that there was something here before

When searching, we skip over "deleted" slots as if they were occupied

If insertions and deletions are common, eventually almost all empty slots will contain a "deleted" marker

This problem can be solved by periodically rewriting the table, or by moving records to fill "deleted" slots at the time of deletion

Double Hashing

The main problem with linear probing is the large number of collisions that result from secondary clustering

Double hashing schemes reduce secondary clustering, by placing a record which collides with another in some distant part of the file.

The location to place the record is a function of the key, not of the location we tried to place it the first time (as with linear probing)

Hopefully, differences in the key will make a second collision very unlikely

Double Hashing

Double hashing is usually implemented by having a second hash function (H2) that computes an offset in the case of a collision

A simple H2 function might be

$$(key/table_size) \bmod table_size$$

Following our previous example, the first hash function will be based on the remainder from dividing the key by table size, and the second will compute an offset based on the result of integer division

We want to place records with the following keys
 27, 18, 29, 28, 39, 13, 16 as before

Double Hashing

0	
1	
2	
3	
4	
5	27
6	
7	18
8	
9	
10	

0	
1	
2	
3	
4	
5	27
6	
7	18
8	
9	29
10	

0	
1	
2	
3	
4	
5	27
6	28
7	18
8	
9	29
10	

0	
1	39
2	
3	
4	
5	27
6	28
7	18
8	
9	29
10	

29 collides with 18 (at 7),
 so we compute an offset.
 $H_2(29) = (29 / 11) \bmod 11$
 $= 2 \bmod 11 = 2$.
 We place 29 at location
 $7 + 2 = 9$.

Hashing

28 hashes to
 location 6
 without
 collision

39 hashes to 6, colliding with 28
 $H_2(39) = (39 / 11) \bmod 11 =$
 $3 \bmod 11 = 3$. So we try to plac
 39 in location $(6 + 3) \bmod 11 =$
 This collides with 29, so we
 try location $(9 + 3) \bmod 11 =$
 $12 \bmod 11 = 1$. It works.

25

Double Hashing

0	
1	39
2	13
3	
4	
5	27
6	28
7	18
8	
9	29
10	

0	
1	39
2	13
3	
4	
5	27
6	28
7	18
8	16
9	29
10	

Key 16 hashes to location 5 and collides with record 27.
 $H_2(16) = (16 / 11) \bmod 11 = 1 \bmod 11 = 1$.

We therefore try location $5 + 1 = 6$.

This collides with record 28, so we try again with $6 + 1 = 7$.
This collides with record 18, so we try $7 + 1 = 8$. It works

Key 13 hashes to location 2 and causes no collision

To retrieve record 16, we now need 4 searches rather than 6 with linear probing.

Need to be careful in choice of H_2 function

Double Hashing

Double hashing breaks up secondary clusters

The variable offset means that we get a lot more gaps between records, which speeds up unsuccessful searches (which end when an empty slot is found).

Double hashing also reduces locality. If a collision occurs the second record will probably be placed faraway, so it's unlikely to be on the same disk page

Using buckets solves this problem. Each slot in the file is a bucket, holding a page-full of records. We only apply H2 when the bucket is full.



University of Dublin
Trinity College



Hashing Techniques

Dynamic Schemes

Rationale for Dynamic Schemes

Static Hashing Schemes

- fixed number of buckets allocated
- this is a problem if the number of records in the file grows or shrinks

Dynamic Hashing schemes are those which allow the dynamic growth and shrinking of the number of file records. Example schemes include

- Primary/Secondary Probing (Brent's method)
- Extendible Hashing

Intention is to allow self adjustment as files grow/shrink

Rationale for Dynamic Schemes

In the example, retrieving record 39 would take 3 probes.

- This is because when we tried to place record 39, it collided with two other records (record 28 at position 6, and 29 at 9). We finally placed it on slot 1.
- Subsequent searches will have to go through the same process. This is bad.

But what if we could make slot 6 free? Then we could always find record 39 in one probe.

- Try moving record 28 from position 6 to the next position on its probe chain.

Rationale for Dynamic Schemes

Place record 28 in location 8.

Place record 39 in location 6.

- Record 39 can now be found in a single access, and record 28 can be found in two accesses.
- Average of 1.5 accesses for the two. Previous average was $(1 + 4) / 2 = 2.5$
- Assumes that all records are accessed equally often
- Assumes that access is more common than insertion

Brent's method

[CACM 16(2) Feb 1973]

Attempts to move records that fall on the primary probe chain of the record to be inserted.

There may be several records on the primary probe chain.

Let j be the number of extra accesses needed to access a record R assuming that we choose to move R .

Let i be the number of locations that we must visit to access the record we are inserting assuming we move R .

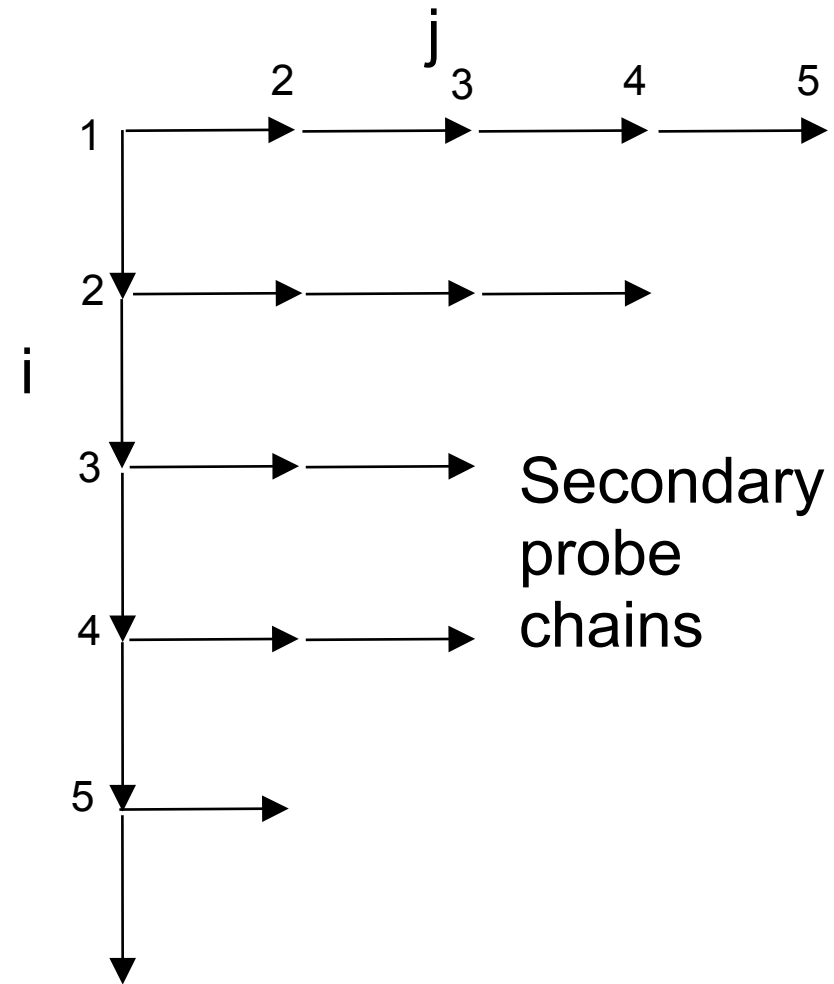
We want to choose R such that $i + j$ is minimised

Brent's Method

Finding i and j such that $(i+j)$ is minimised can be viewed as a search of a two dimensional space.

The vertical axis shows the points in the primary probe chain. Each point on the axis is a location that the key we are inserting hashes to. All but the last point already contain records

Each horizontal line is the chain of locations that each existing record that we are thinking about moving hashes to.



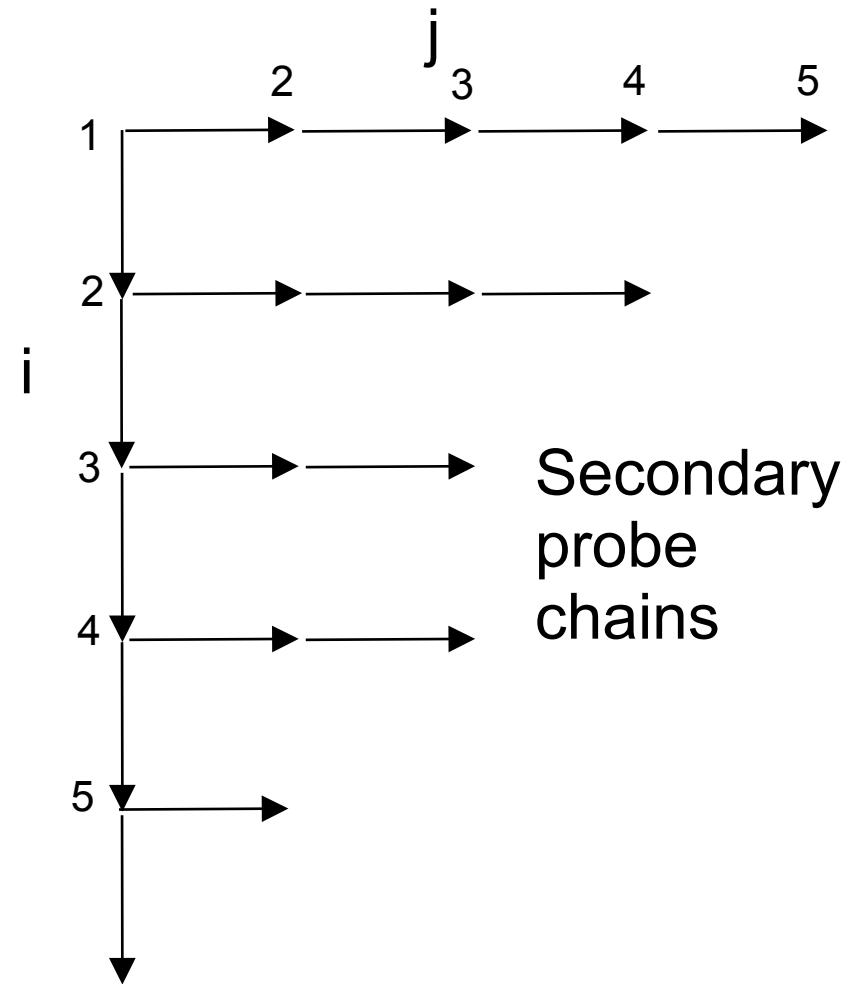
Brent's method

When inserting a record and there are several collisions, we search for a record to move as follows.

- First, we try to move the first record it collides with (at point $(1,1)$) to the next place on that record's probe chain, i.e. to $(1,2)$.
- If $(1,2)$ is already occupied, we try the next point on that record's chain, i.e. $(1,3)$. If this is occupied, we stop trying to move this record for the moment
- Instead, we try to move the next record on the **primary** probe chain, that is the one at $(2,1)$. We try to move this record to $(2,2)$. If point $(2,2)$ is already occupied we go back and try to move the record at $(1,1)$ again, and this time to point $(1,4)$
- If $(1,4)$ is occupied, we try to move the record at point $(3,1)$ to point $(3,2)$. If this is occupied we try $(2,1)$ to $(2,3)$.
- And so forth

Brent's method

We want to minimise $i + j$, so we always try to move an existing record on the primary probe chain to a point that minimises the combined number of accesses.



Example

We want to place records with the following keys
27, 18, 29, 28, 39, 13, 16 as before

0	
1	
2	
3	
4	
5	27
6	28
7	18
8	
9	29
10	

0	
1	39?
2	
3	
4	
5	27
6	28?
7	18
8	
9	29?
10	

39 hashes to 6, colliding with 28.

$$H_2(39) = (39 / 11) \bmod 11 = 3 \bmod 11 = 3.$$

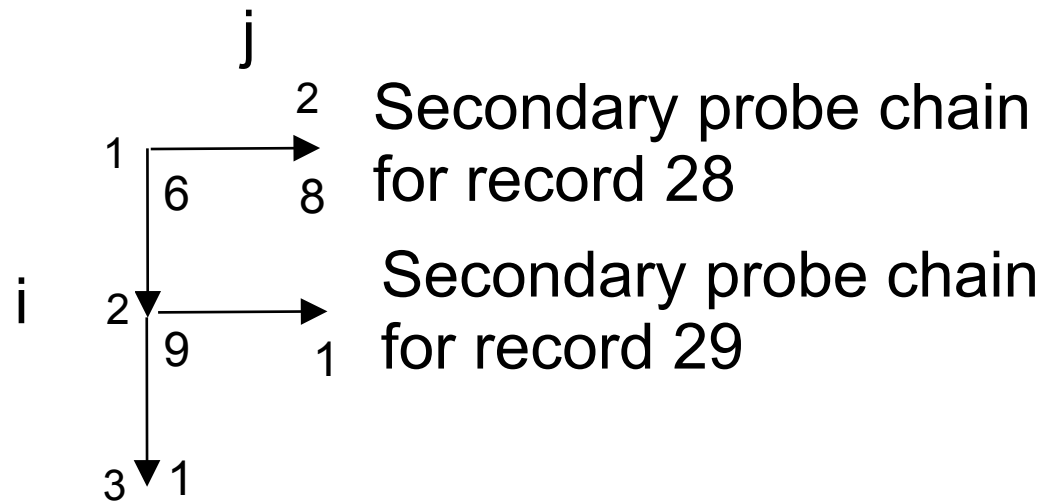
So we try to place 39 in location $(6 + 3) \bmod 11 = 9$.

This collides with 29, so we try location $(9 + 3) \bmod 11 = 12 \bmod 11 = 1$.

Place 1 is free so we could place 39 there.

28 hashes to location 6 without collision

But would it be worthwhile to move record 28 or record 29? Could we reduce the average number of accesses?



If we were to place record 39 at point 1, we would be at point (1,3) in the two dimensional space with $i + j = 1 + 3 = 4$.

Primary probe chain for record 39

We first look at moving the record at point (1, 1), that is record 28, which is in location 6 of the table.

We try to move it to point (1,2), which is the second place that record 28 can be placed in the table. $H_2(28) = 28 / 11 = 2$. $6 + 2 = 8$. Location 8 is free so record 28 can be moved. Point (1, 2) gives a total $i + j = 1 + 2 = 3$. $3 < 4$, so the move is desirable.

Brent's method

We want to place records with the following keys
27, 18, 29, 28, 39, 13, 16 as before

0	
1	
2	
3	
4	
5	27
6	39
7	18
8	28
9	29
10	

0	
1	
2	13
3	
4	
5	27
6	39
7	18
8	28
9	29
10	

We move record
28 from location
6 to location 8
and insert 39 at
location 6

Record 13 hashes
to location
 $13 \bmod 11 = 2$,
and is placed
without problem

Brent's method

0	
1	
2	13
3	
4	
5	27
6	39
7	18
8	28
9	29
10	

Record 16 hashes to location $16 \bmod 11 = 5$, which is already occupied by record 27.

The offset, $H_2(16) = 16/11 = 1$. Locations 5, 6, 7, 8, and 9 are already occupied, and the first empty location is 10.

The primary probe chain for record 16 is 5 (27), 6 (39), 7 (18), 8 (28), 9 (29), 10 (empty).

$s = \text{length of chain} = 6$

Brent's method

We set $i = 2, j = 2$. That is, we try to move the record at the second location of the primary probe chain (i.e. Record 39 at location 6) to the second location that record's secondary search chain. $H_2(39) = 39 / 11 = 3$. $(6 + 3) \bmod 11 = 9 \bmod 11 = 9$.

Location 9 is occupied, so that won't work.

We set $i = 1, j = 4$. That is we try to move record 27 to the fourth position of its secondary probe chain.

$H_2(27) = 2$. $(5 + 2 + 2 + 2) \bmod 11 = 11 \bmod 11 = 0$.

Location 0 is empty, so it works.

0	27
1	
2	13
3	
4	
5	16
6	39
7	18
8	28
9	29
10	

Pseudocode

```
- Home = H1(key); done = false;
  if ( is_empty(home) )
      hash_table[home] = key;
      done = true;
  Else offset = H2 (key); s = 2;
      primary_probe_chain[1] = home;
      while ( ! is_empty (hash_table[location]) )
          location = (location + offset) % table_size;
          primary_probe_chain[s] = location;
          if (location == home )
              error("Hash table full");
          endif
          s = s + 1;
      endwhile
      primary_probe_chain[s] = location;
  endif
```

Pseudocode

```
i = i; j = 2;
while ( (( i + j ) < s) && !done )
    can_move = can record hash_table[ primary_probe_chain[i] ] be
                moved to position j on its secondary probe chain?
    if ( can_move )
        move record hash_table[primary_probe_chain[i]];
        hash_table[primary_probe_chain[i]] = key;
        done = true;
    else
        vary i and j to minimise i + j;
    endif
endwhile

if ( ! done )
    hash_table[primary_probe_chain[k]] = key;
endif
```

Extendible Hashing

Extendible hashing uses the binary representation of the hash value (K) in order to access a directory which is an array of size 2^d where d is called the global depth

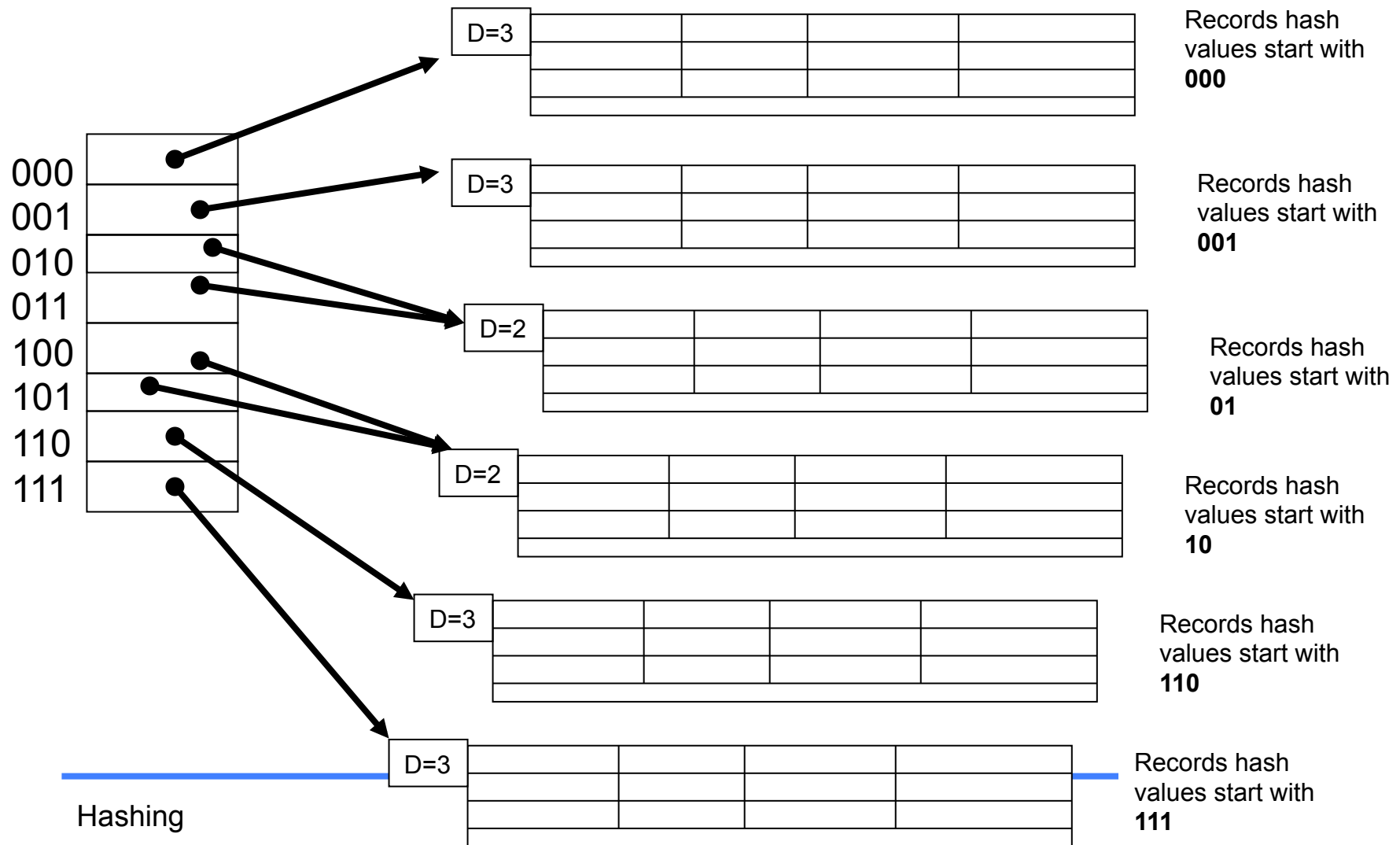
- The directories can be stored on disk, and they expand or shrink dynamically. Directory entries point to the disk blocks that contain the stored records
- An insertion in a disk block that is full causes the block to split into two blocks and the records are redistributed among the two blocks. The directory is updated appropriately
- Extendible hashing does not require an overflow area.

Extendible Hashing

How it works

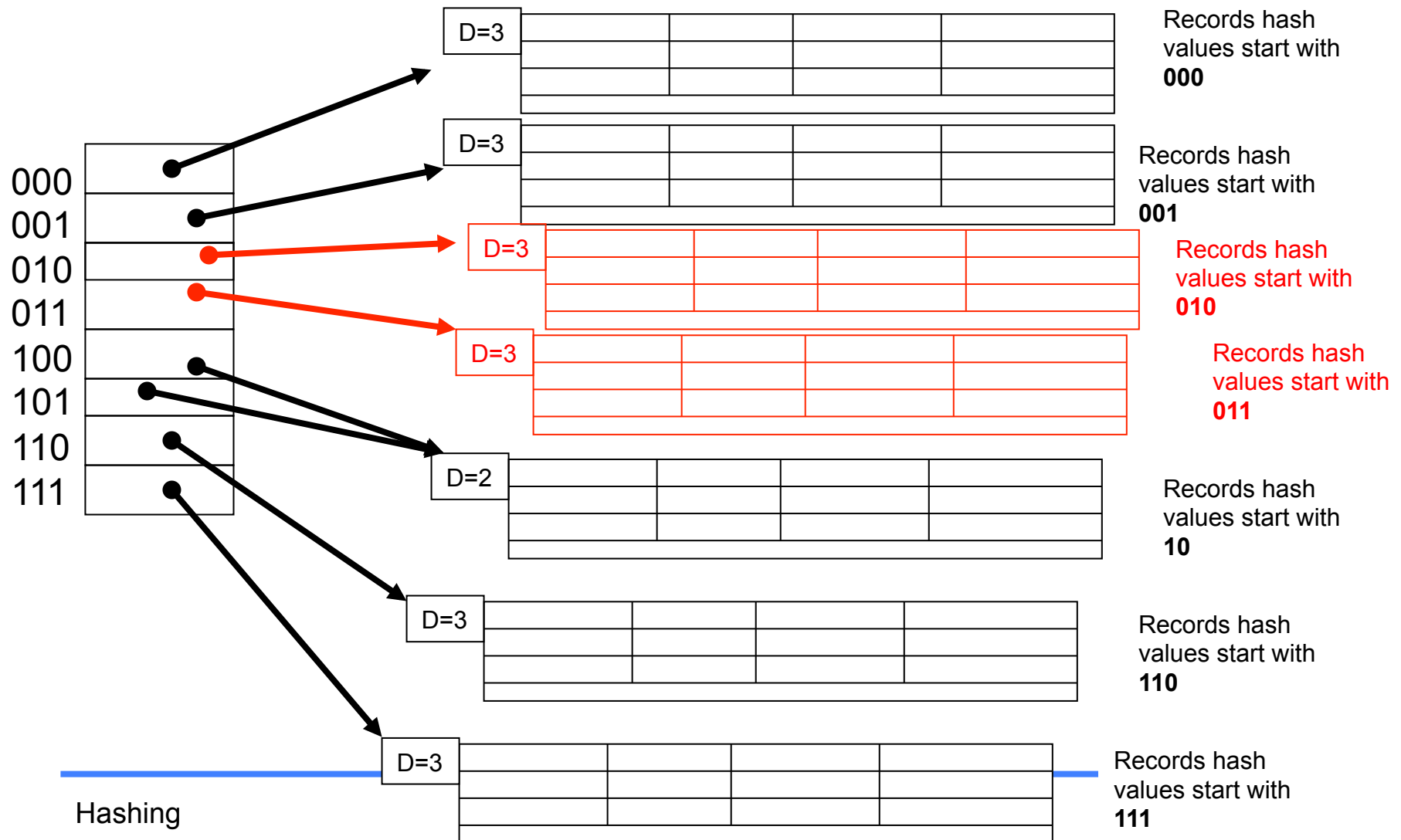
- The integer value corresponding to the first (high order) d bits of a hash value is used as an index to the array to determine the directory entry, and the address in that entry determines the bucket in which the corresponding record is stored
- There does not have to be a distinct bucket for each of the directory entries. Several directory locations with the same first d bits for their hash value may contain the same bucket address if all the records that hash to these locations fit in a single bucket
- **Local depth** stored with each bucket specifies the number of bits on which the bucket contents are based

Extendible Hashing



Extendible Hashing at work

Suppose new inserted records causes overflow in the buckets whose hash value starts with 010. Then split occurs and records redistributed to new buckets according to what the record hash values starts with. Local depth of buckets is increased by one



Extendible Hashing at work

Suppose the bucket to be split has local depth already equal to global depth? E.g. bucket related to 111

Increase directory size

Change Global Depth = 4, add entries for 1110 and 1111

Split overflowing bucket as before, now with local depth = 4

Advantages to Extendible Hashing

Performance of file does not degrade as file grows

No extra space is allocated for growth as buckets can be allocated dynamically as needed

Space overhead for directory is negligible

Overhead

- Splitting of buckets only causes minor reorg of 2 buckets involved
- Although Directory doubling/halving is more expensive

Max size of directory is 2^k where k is the number of bits in the hash value

Minor disadvantage is the fact that the directory needs to be searched before access can be achieved

Review

Hashing schemes provide very fast access

- Very rapid insertion and deletion of records
- Possible to find a record with (on average) a constant number of disk accesses. The number of accesses to find a record is not related to the number of records in the file.

Three ways to reduce collisions

- use a better hash function
- use a bigger table
- use a better collision resolution scheme

Review

Many collision resolution schemes

We have seen several based on open addressing

- Linear probing
- Double Hashing
- Dynamic Schemes
 - *Brent's method*
 - *Extendible Hashing*

There are lots of other schemes that use other approaches other than open addressing

Review

Hashing is (usually) the fastest way to find a record given its key

The speed of hashing comes at a price

- Access on more than one key is impossible unless you set up indexes which have their own hash tables
- Very expensive to find all the keys in a given range
- Variable length records are difficult to use with hashing