**DATA STRUCTURES - UNIT II - Notes**

##### HASHING:

Definition**:** Hashing is a technique that is designed to use a special function called the Hash function which is used to map a given value with a particular key for faster access of elements.

The efficiency of mapping depends of the efficiency of the hash function used.

###### Contents under Hashing:

* Hash Table
* Hash Function
* Collision Resolution Techniques

In general, The process of placing *hash values* into the *hash table*

by using *hash functions* is known as *hashing*.

###### Some examples of how hashing is used in our lives include:

* In universities, each student is assigned a unique roll number that can be used to retrieve information about them.
* In libraries, each book is assigned a unique number that can be used to determine information about the book, such as its exact position in the library or the users it has been issued to etc.

In both these examples the students and books were hashed to a unique number.

###### Hashing is implemented in two steps:

1. An element is converted into an integer by using a hash function. This integer can be used as an index to store the original element, which falls into the hash table.
2. The element is stored in the hash table where it can be quickly retrieved using hashed key.

###### Why Hashing?

* It is used to facilitate the next level searching method when compared with the linear or binary search.
* Hashing allows to update and retrieve any data entry in a

constant time O(1).

* Constant time O(1) means the operation does not depend on the size of the data.
* Hashing is used with a database to enable items to be

retrieved more quickly.

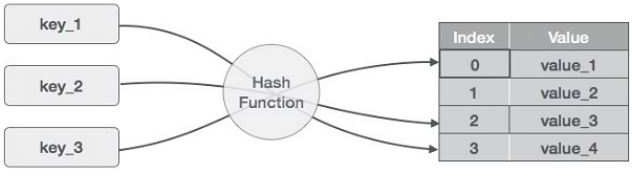
NOTE: Hashing is used to index and retrieve items in a database because it is faster to find the item using the shorter hashed key than to find it using the original value.

##### Hash Table:

Definition: It is a Data structure where the data elements are stored(inserted), searched, deleted based on the keys generated for each element, which is obtained from a hashing function. In a hashing system the keys are stored in an array which is called the Hash Table. A perfectly implemented hash table would always promise an average insert/delete/retrieval time of O(1).

##### Hash Function:

Definition: A function that converts a given number to a small practical integer value. The integer value is used as an index in hash table. In simple terms, a hash function maps a big number to a small integer that can be used as index in hash table.



###### A good hash function should have following properties:

* Efficiently computable.
* Should uniformly distribute the keys. (Each table position equally likely for each key) I.e., The probability of any two keys hashing to the same slot is 1/N.
* Less collisions: Collisions occur when pairs of elements are

mapped to the same hash value. These should be avoided.

NOTE: Irrespective of how good a hash function is, collisions are bound to occur. Therefore, to maintain the performance of a hash table, it is important to manage collisions through various collision resolution techniques.

Perfect Hash Function: If a hash function that maps each key to a distinct address/index, it is known as perfect hash function. To design perfect hash function, usually all the keys must be known beforehand.

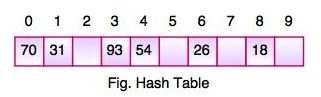
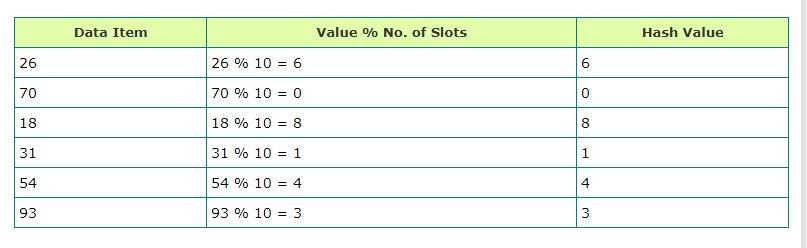
##### There are many Hash Function approaches as follows:

###### Division Method:

This method takes an element and divides it by the table size and returns the remainder as its hash value.

**General Formula: Hash Key = Key Value % Size Of Table h (K) = K mod M**

**Example: For an Array of size 10.**

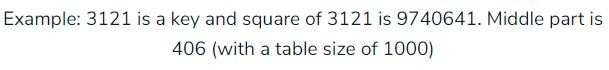


* After computing the hash values, we can insert each item into the hash table at the designated position as shown in the above figure.
* Constant amount of time O(1) is required to compute the hash value and index of the hash table at that location.

###### Midsquare Method:

The hash value is derived by getting the middle digits from value of (element)2.

**General Formula: h(k) = K2 and get middle digits(based on size of table)**



###### Digital Folding Method:

Divide the element into a number of parts(digits), where each part contains same number of digits except the last part may have lesser digits than the other parts. The hash value is generated by adding individual parts, ignoring the last carry.

NOTE: You divide the key in parts whose size matches the size of required address.

**Example:**

Element:123456789 and size of required address is 3 digits.

123+456+789 = 1368

To reduce the size to 3, either 1 or 8 is removed and accordingly the key would be 368 or 136 respectively.

###### Multiplication Method:

* Choose a constant 'a' such that 0<a<1
* Multiply the key with 'a'
* Extract the fractional part of 'ka'
* Multiply the result of step 3 by size of Hash Table.

**General Formula: h(k) = [size\*(key\*a mod 1)] where, a = 0.6180339887**

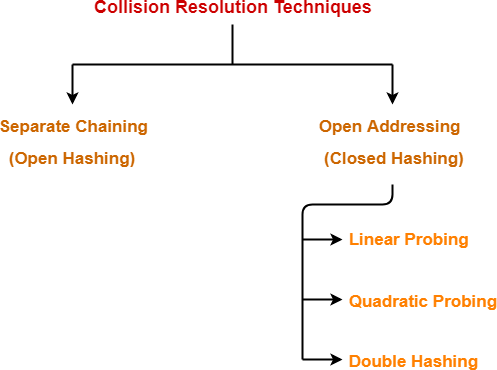
##### Collision Resolution Techniques:

Collision: No matter what the hash function, there is a possibility that two different keys could resolve to the same hash value. This situation is known as Collision. This is the problem with Hashing, because practically it is not possible to avoid collisions unless you have perfect hash function knowing the elements beforehand.

###### Handling The Collisions:

The following techniques can be used to handle the collisions:

* Open Addressing (Array based implementation)
* Separate Chaining (Linked list based implementation)



###### Open Addressing: (Closed Hashing)

1. In open addressing, instead of in linked lists, all entry records are stored in the array itself. When a new entry has to be inserted, the hash index of the hashed value is computed and then the array is examined (starting with the hashed index). If the slot at the hashed index is unoccupied, then the entry record is inserted in slot at the hashed index else it proceeds in some probe sequence until it finds an unoccupied slot.
2. When searching for an entry, the array is scanned in the same sequence until either the target element is found or an unused slot is found. This indicates that there is no such key in the table. The name "open addressing" refers to the fact that the location or address of the item is not determined by its hash value.

##### Operations in Open Addressing:

###### Insertion operation:

* + Hash function is used to compute the hash value for a key to be inserted.
  + Hash value is then used as an index to store the key in the hash table.

In case of Collision:

* + Probing is performed until an empty bucket is found.
  + Once an empty bucket is found, the key is inserted.
  + Probing is performed in accordance with the technique used for open addressing.
* Search operation: (To search any particular key)
  + Its hash value is obtained using the hash function used.
  + Using the hash value, that bucket of the hash table is checked.
  + If the required key is found, the key is returned.
  + Otherwise, the subsequent buckets are checked until the required key or an empty bucket is found.
  + The empty bucket indicates that the key is not present in the hash table.

###### Delete operation:

* + The key is first searched and then deleted.
  + After deleting the key, that particular bucket is marked as "deleted".

##### Methods Of Open Addressing:

###### Linear Probing:

When collision occurs, we linearly probe for the next bucket.

Linear probing is when the interval between successive probes is fixed (usually to 1). Let’s assume that the hashed index for a particular entry is index. The probing sequence for linear probing will be:

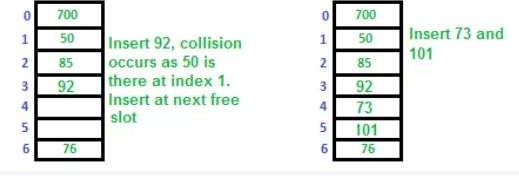
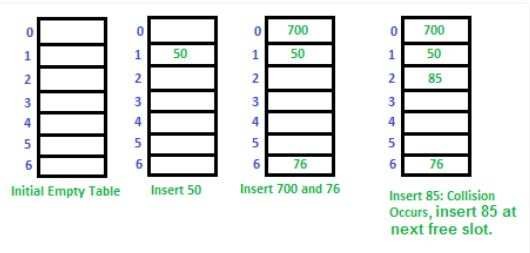
index = index % hashTableSize index = (index + 1) % hashTableSize index = (index + 2) % hashTableSize index = (index + 3) % hashTableSize

and so on...

Example: Let us consider a simple hash function as "key mod 7" and the sequence of keys as 50, 700, 76, 85, 92,

73, 101.

The probing is done as follows:



###### NOTE:

Clustering: The main problem with linear probing is clustering, many consecutive elements form groups and it starts taking time to find a free slot or to search an element.

###### Quadratic Probing:

Quadratic probing is similar to linear probing and the only difference is the interval between successive probes or entry slots. Here, when the slot at a hashed index for an entry record is already occupied, you must start traversing until you find an unoccupied slot. The interval between slots is computed by adding the successive value of an arbitrary polynomial in the original hashed index.

For example, Instead of moving one cell each time, move 'i2' cells from the point of collision.

###### Trace:

* let hash(x) be the slot index computed using hash function.
* If slot [hash(x) % Size] is full, then we try [(hash(x) + 1\*1) % Size]
* If [(hash(x) + 1\*1) % Size] is also full, then we try [(hash(x) + 2\*2)

% Size]

* If [(hash(x) + 2\*2) % Size] is also full, then we try [(hash(x) + 3\*3)

% Size]

###### NOTE:

It can also be of the form of a polynomial. h(k,i)=[(hl(k)+c1\*i+c2\*i2)]

where, hl(k) = k mod m c1 and c2 are constants.

Only when collision occurs, the constants come into the formula.

###### Double Hashing:

When collision occurs, we introduce a secondary hash function, the index generation now depends upon two hash functions.

Double hashing : F(i) = i \* Hash2(X) h(k, i) = (h1(k) + i \* h2(k)) mod TableSize h(k, 0)= (h1(k) + 0 \* h2(k)) mod TableSize h(k, 1)= (h1(k) + 1 \* h2(k)) mod TableSize h(k, 2)= (h1(k) + 2 \* h2(k)) mod TableSize

generally, h1(k)=k mod m(TableSize) h2(k)= k mod ml(Nearest Prime < TableSize)

###### Rehashing:

As the name suggests, rehashing means hashing again. Basically, when the load factor increases to more than its pre- defined value (default value of load factor is 0.75), the complexity increases. So to overcome this, the size of the array is increased (doubled) and all the values are hashed again and stored in the new double sized array to maintain a low load factor and low complexity.

* + **Load Factor**: It is defined as,

In open addressing, the value of load factor always lie between 0 and 1.

It is because,

* + - In open addressing, all the keys are stored inside the hash table.
    - So, size of the table is always greater or at least equal to the number of keys stored in the table.

**NOTE**: This Load Factor needs to be kept low(<0.75), so that number of entries at one index is less and so is the complexity almost constant, i.e., O(1).

###### Why Rehashing?

Rehashing is done because whenever key value pairs are inserted into the map, the load factor increases, which implies that the time complexity also increases as explained above. This might not give the required time complexity of O(1).

Hence, rehash must be done, increasing the size of the Array so as to reduce the load factor and the time complexity.

General Rule of Rehashing:

Double the Size of the array(to decrease load factor) and take 'm' to be the nearest prime number(less than size of array).

**Separate Chaining**: (Open Hashing)

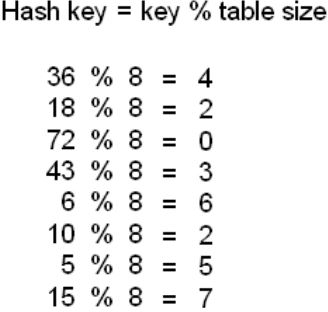
Separate chaining is one of the most commonly used collision resolution techniques. It is usually implemented using linked lists. In separate chaining, each element of the hash table is a linked list. To store an element in the hash table you must insert it into a specific linked list. If there is any collision (i.e. two different elements have same hash value) then store both the elements in the same linked list thus creating a chain.

Let us understand this with an example:

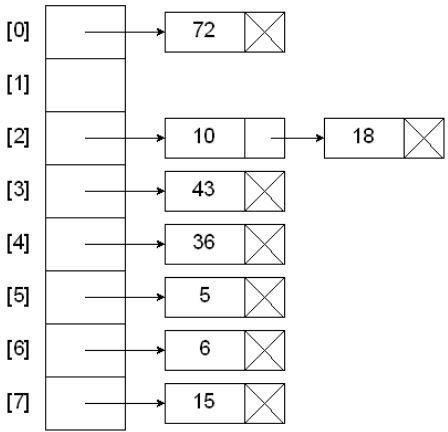
Assume the table has 8 slots (m=8). Using the chaining, insert, the following elements into the hash table. 36, 18, 72, 43, 6,

10, 5 and 15 are inserted in the order.

Assume, h(k)=k mod m



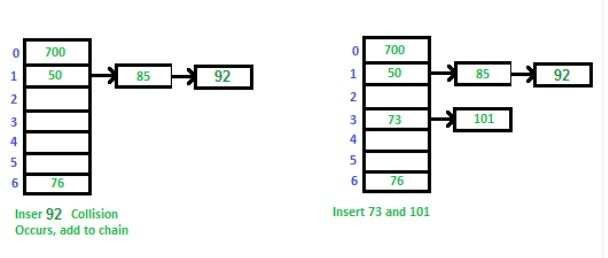
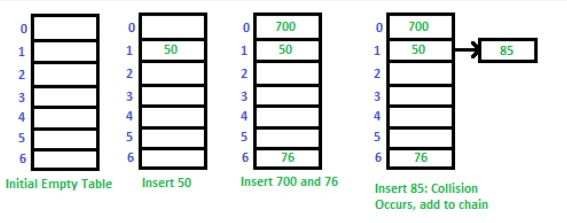
Now representation in a linked list,



###### Let's take another example:

Let us consider a simple hash function as “key mod 7” and sequence of keys as 50, 700, 76, 85, 92, 73, 101.

Representation of linked list:



###### Advantages:

1. Simple to implement.
2. Hash table never fills up, we can always add more elements to the chain.
3. Less sensitive to the hash function or load factors.
4. It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

###### Disadvantages:

1. Wastage of Space (Some Parts of hash table are never used)
2. If the chain becomes long, then search time can become O(n) in the worst case.
3. Uses extra space for links.

###### Analysis of separate chaining:

* + If m is too large, then too many empty chains.
  + If m is too small, then chains are too long.

Differences between Open Addressing and Separate Chaining:



**Extendible Hashing:**

It is an approach that tries to make hashing dynamic, i.e., to allow insertions and deletions to occur without resulting in poor performance.

The dynamic aspects are handled by two mechanisms:

* Insertions and bucket splitting.
* Deletions and bucket combination.

Extendible hashing solves bucket overflow by splitting the bucket into two and if necessary increasing the directory size(depth).

When the directory size increases it doubles its size a certain number of times.

###### Search Algorithm:

* Calculate the hash address of the key

(note that no table size is specified, so we don't take "mod")

* Check how many bits are used in the directory, i.e., we call it "depth". Call this as 'i' bits.
* Take the least significant 'i' bits of hash address. This gives an index of the directory.
* Using this index, go to the directory and find the bucket address where the record might be.

###### Insertion Algorithm:

* When inserting a new record, a search is performed to locate the position for the record.
* If the bucket that should contain the record is less than full, then the record can be inserted into the bucket.
* We convert the element into its binary form and we choose the number of bits from the right side depending on significant bits specified in the directory.
* *The number of significant bits required to identify the bucket is the same as the number of significant bits in the directory.*
* According to the significant bits in the bucket and index, we place the element.

###### Structure:

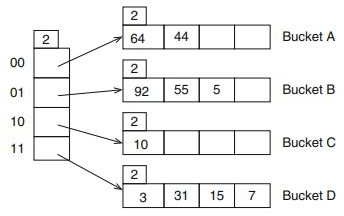
Extendible hashing uses a directory to access its buckets. This directory is usually small enough to be kept in main memory and has the form of an array with 2d entries, each entry storing a

bucket address (pointer to a bucket). The variable 'd' is called the global depth of the directory. To decide where a key k is stored, extendible hashing uses the last 'd' bits of some adopted

hash function h(k) to choose the directory entry. Multiple directory entries may point to the same bucket. Every bucket has a local depth 'leqd'. The difference between local depth and global depth

affects overflow handling.

An example of extendible hashing is shown in Fig. 1. Here there are four directory entries and four buckets. The global depth and all the four local depths are 2. For simplicity assume the adopted hash function is h(k) = k. For instance, to search for record 15, one refers to directory entry 15% 4 = 3 (or 11 in binary format), which points to bucket D.



NOTE: A directory doubling occurs, when we choose to use one more bit from the hash value. (this does not mean that the number of buckets is doubled as buckets will share directory

entries)

###### Advantages:

1. Performance does not degrade as file size increases.
2. Stores the minimum number of buckets.
3. Number of buckets grows/shrinks dynamically.

###### Disadvantages:

1. The directory must be searched.
2. The directory must be stored.

\*Examples are attached from the next page.

## Extendible Hashing

#### Suppose that g=2 and bucket size = 3.

* Suppose that we have records with these keys and hash function h(key) = key mod 64:

|  |  |  |
| --- | --- | --- |
| key | h(key) = key mod 64 | bit pattern |
| 1111 | 23 | 010111 |
| 3333 | 5 | 000101 |
| 1235 | 19 | 010011 |
| 2378 | 10 | 001010 |
| 1212 | 60 | 111100 |
| 1456 | 48 | 110000 |
| 2134 | 22 | 010110 |
| 2345 | 41 | 101001 |
| 1111 | 23 | 010111 |
| 8231 | 39 | 100111 |
| 2222 | 46 | 101110 |
| 9999 | 15 | 001111 |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1111 | 3333 | 1235 | 2378 | 1212 | 1456 | 2134 | 2345 | 1111 | 8231 | 2222 | 9999 |

|  |  |  |  |
| --- | --- | --- | --- |
| l=2 |  | | |
|  | |  |  |

###### local depth is 2 and bucket size is 3

|  |  |  |  |
| --- | --- | --- | --- |
| l=2 |  | | |
|  | |  |  |

global depth is 2

|  |  |
| --- | --- |
| g=2 |  |
| 00 | |
| 01 | |
| 10 | |
| 11 | |

|  |  |  |  |
| --- | --- | --- | --- |
| l=2 |  | | |
|  | |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| l=2 |  | | |
|  | |  |  |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1111 | 3333 | 1235 | 2378 | 1212 | 1456 | 2134 | 2345 | 1111 | 8231 | 2222 | 9999 |

1111

11

1101**11**

|  |  |
| --- | --- |
| g=2 |  |
| 00 | |
| 01 | |
| 10 | |
| 11 | |

|  |  |  |  |
| --- | --- | --- | --- |
| l=2 |  | | |
|  | |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| l=2 |  | | |
|  | |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| l=2 |  | | |
|  | |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| l=2 |  | | |
| **1111** | |  |  |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1111 | 3333 | 1235 | 2378 | 1212 | 1456 | 2134 | 2345 | 1111 | 8231 | 2222 | 9999 |

3333

01

0001**01**

|  |  |
| --- | --- |
| g=2 |  |
| 00 | |
| 01 | |
| 10 | |
| 11 | |

|  |  |  |  |
| --- | --- | --- | --- |
| l=2 |  | | |
| **3333** | |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| l=2 |  | | |
|  | |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| l=2 |  | | |
|  | |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| l=2 |  | | |
| 1111 | |  |  |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1111 | 3333 | 1235 | 2378 | 1212 | 1456 | 2134 | 2345 | 1111 | 8231 | 2222 | 9999 |

1235

11

0100**11**

|  |  |
| --- | --- |
| g=2 |  |
| 00 | |
| 01 | |
| 10 | |
| 11 | |

|  |  |  |  |
| --- | --- | --- | --- |
| l=2 |  | | |
| 3333 | |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| l=2 |  | | |
|  | |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| l=2 |  | | |
|  | |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| l=2 |  | | |
| 1111 | | **1235** |  |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1111 | 3333 | 1235 | 2378 | 1212 | 1456 | 2134 | 2345 | 1111 | 8231 | 2222 | 9999 |

|  |  |
| --- | --- |
| g=2 |  |
| 00 | |
| 01 | |
| 10 | |
| 11 | |

|  |  |  |  |
| --- | --- | --- | --- |
| l=2 |  | | |
| 3333 | | 2345 |  |

|  |  |  |  |
| --- | --- | --- | --- |
| l=2 |  | | |
| 2378 | | 2134 |  |

|  |  |  |  |
| --- | --- | --- | --- |
| l=2 |  | | |
| 1212 | | 1456 |  |

|  |  |  |  |
| --- | --- | --- | --- |
| l=2 |  | | |
| 1111 | | 1235 | 1111 |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1111 | 3333 | 1235 | 2378 | 1212 | 1456 | 2134 | 2345 | 1111 | 8231 | 2222 | 9999 |

8231

11

1001**11**

g = l, directory structure is doubled.

|  |  |  |  |
| --- | --- | --- | --- |
| l=2 |  | | |
| 1212 | | 1456 |  |

|  |  |
| --- | --- |
| g=2 |  |
| 00 | |
| 01 | |
| 10 | |
| 11 | |

|  |  |  |  |
| --- | --- | --- | --- |
| l=2 |  | | |
| 3333 | | 2345 |  |

|  |  |  |  |
| --- | --- | --- | --- |
| l=2 |  | | |
| 2378 Bu | | 2134  cket overfl | ow occurs. As |

|  |  |  |  |
| --- | --- | --- | --- |
| l=2 |  | | |
| 1111 | | 1235 | 1111 |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1111 | 3333 | 1235 | 2378 | 1212 | 1456 | 2134 | 2345 | 1111 | 8231 | 2222 | 9999 |

|  |  |
| --- | --- |
| g=3 |  |
| 00 | |
| 01 | |
| 10 | |
| 11 | |
| 00 | |
| 01 | |
| 10 | |
| 11 | |

|  |  |  |  |
| --- | --- | --- | --- |
| l=2 |  | | |
| 1212 | | 1456 |  |

|  |  |  |  |
| --- | --- | --- | --- |
| l=2 |  | | |
| 3333 | | 2345 |  |

|  |  |  |  |
| --- | --- | --- | --- |
| l=2 |  | | |
| 2378 | | 2134 |  |

|  |  |  |  |
| --- | --- | --- | --- |
| l=2 |  | | |
| 1111 | | 1235 | 1111 |

|  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1111 | 3333 | 1235 | 2378 | 1212 | 1456 | 2134 | 2345 | 1111 | 8231 | 2222 | 9999 |

l=2

|  |  |  |  |
| --- | --- | --- | --- |
| l=2 |  | | |
| 1212 | | 1456 |  |

|  |  |
| --- | --- |
| g=3 |  |
| **0**00 | |
| **0**01 | |
| **0**10 | |
| **0**11 | |
| **1**00 | |
| **1**01 | |
| **1**10 | |
| **1**11 | |

|  |  |  |  |
| --- | --- | --- | --- |
| l=2 |  | | |
| 3333 | | 2345 |  |

|  |  |  |  |
| --- | --- | --- | --- |
| l=2 |  | | |
| 2378 | | 2134 |  |

|  |  |  |  |
| --- | --- | --- | --- |
| l=3 |  | | |
| 1111 | | 1235 | 1111 |

|  |  |  |  |
| --- | --- | --- | --- |
| l=3 |  | | |
|  | |  |  |

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1111 | 3333 | 1235 | 2378 | 1212 | 1456 | 2134 | 2345 | 1111 | 8231 | | 2222 | 9999 |
|  | | | | | | | | | |  | | |

l=2

8231

111

1001**11**

|  |  |  |  |
| --- | --- | --- | --- |
| l=2 |  | | |
| 1212 | | 1456 |  |

|  |  |
| --- | --- |
| g=3 |  |
| 000 | |
| 001 | |
| 010 | |
| **0**11 | |
| 100 | |
| 101 | |
| 110 | |
| 111 | |

|  |  |  |  |
| --- | --- | --- | --- |
| l=2 |  | | |
| 3333 | | 2345 |  |

|  |  |  |  |
| --- | --- | --- | --- |
| l=2 |  | | |
| 2378 | | 2134 |  |

|  |  |  |  |
| --- | --- | --- | --- |
| l=3 |  | | |
| 1235 | |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| l=3 |  | | |
| 1111 | | 1111 | **8231** |

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1111 | 3333 | 1235 | 2378 | 1212 | 1456 | 2134 | 2345 | 1111 | 8231 | 2222 | | 9999 |
|  | | | | | | | | | | |  | |

l=2

2222

110

1011**10**

|  |  |  |  |
| --- | --- | --- | --- |
| l=2 |  | | |
| 1212 | | 1456 |  |

|  |  |
| --- | --- |
| g=3 |  |
| 000 | |
| 001 | |
| 010 | |
| **0**11 | |
| 100 | |
| 101 | |
| 110 | |
| 111 | |

|  |  |  |  |
| --- | --- | --- | --- |
| l=2 |  | | |
| 3333 | | 2345 |  |

|  |  |  |  |
| --- | --- | --- | --- |
| l=2 |  | | |
| 2378 | | 2134 | **2222** |

|  |  |  |  |
| --- | --- | --- | --- |
| l=3 |  | | |
| 1235 | |  |  |

|  |  |  |  |
| --- | --- | --- | --- |
| l=3 |  | | |
| 1111 | | 1111 | 8211 |

|  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| 1111 | 3333 | 1235 | 2378 | 1212 | 1456 | 2134 | 2345 | 1111 | 8231 | 2222 | 9999 | |
|  | | | | | | | | | | | |  |

l=2

9999

111

g = l, ubled

again and same process will repeat.

0011**11**

|  |  |  |  |
| --- | --- | --- | --- |
| l=2 |  | | |
| 1212 | | 1456 |  |

|  |  |
| --- | --- |
| g=3 |  |
| 000 | |
| 001 | |
| 010 | |
| **0**11 | |
| 100 | |
| 101 | |
| 110 | |
| 111 | |

|  |  |  |  |
| --- | --- | --- | --- |
| l=2 |  | | |
| 3333 | | 2345 |  |

|  |  |  |  |
| --- | --- | --- | --- |
| l=2 |  | | |
| 2378 | | 2134 |  |

|  |  |  |  |
| --- | --- | --- | --- |
| l=3 |  | | |
| 1B2u3c5ket director | | overflow occ y structure | urs again. As should be do |

|  |  |  |  |
| --- | --- | --- | --- |
| l=3 |  | | |
| 1111 | | 1111 | 8211 |

Extendible Hashing Example - 2

* + Suppose that g=2 and bucket size = 4.
  + Suppose that we have records with these keys and hash function h(key) = key mod 64:

|  |  |  |
| --- | --- | --- |
| **key** | **h(key) = key mod 64** | **bit pattern** |
| 288 | 32 | 100000 |
| 8 | 8 | 001000 |
| 1064 | 40 | 101000 |
| 120 | 56 | 111000 |
| 148 | 20 | 010100 |
| 204 | 12 | 001100 |
| 641 | 1 | 000001 |
| 700 | 60 | 111100 |
| 258 | 2 | 000010 |
| 1586 | 50 | 110010 |
| 44 | 44 | 101010 |

COSC 2P03 Week 12 1

Extendible Hashing Example – directory and bucket structure

|  |  |  |  |
| --- | --- | --- | --- |
| g=2 | | | |
| 00 | 01 | 10 | 11 |

|  |
| --- |
| l = 2 |
| 8 |
| 204 |
| 641 |
| 258 |

|  |
| --- |
| l = 2 |
| 148 |
|  |
|  |
|  |

|  |
| --- |
| l = 2 |
| 288 |
| 1064 |
| 44 |
|  |

|  |
| --- |
| l = 2 |
| 120 |
| 700 |
| 1586 |
|  |

COSC 2P03 Week 12 2

# Bucket and directory split

### Insert 68

* + 68 mod 64 = 4 = 000100

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| g=3 | | | | | | | |
| 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |

COSC 12

|  |
| --- |
| l = 3 |
| 641 |
| 258 |
| 68 |
|  |

|  |
| --- |
| l = 3 |
| 8 |
| 204 |
|  |
|  |

|  |
| --- |
| l = 2 |
| 148 |
|  |
|  |
| 2P03 Week |

|  |
| --- |
| l = 2 |
| 288 |
| 1064 |
| 44 |
|  |

|  |
| --- |
| l = 2 |
| 120 |
| 700 |
| 1586 |
| 3 |

# Bucket split – no directory split

### Insert 48 and 575

* + 48 mod 64 = 48 = 110000
  + 575 mod 64 = 63 = 111111

|  |
| --- |
| l = 3 |
| 641 |
| 258 |
| 68 |
|  |

|  |
| --- |
| l = 3 |
| 8 |
| 204 |
|  |
|  |

|  |
| --- |
| l = 2 |
| 148 |
|  |
|  |
| COSC |

|  |
| --- |
| l = 2 |
| 288 |
| 1064 |
| 44 |
| eek 12 |

|  |
| --- |
| l = 3 |
| 1586 |
| 48 |
|  |
|  |

|  |
| --- |
| l = 3 |
| 120 |
| 700 |
| 575 |
| 4 |

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| g=3 | | | | | | | |
| 000 | 001 | 010 | 011 | 100 | 101 | 110 | 111 |

2P03 W

# Multiple splits

### Insert 16, 18, 22, 23

|  |  |
| --- | --- |
|  | |
| 010 | 011 |

* + 16 mod 64 = 16 = 010000
  + 18 mod 64 = 18 = 010010
  + 22 mod 64 = 22 = 010110
  + 23 mod 64 = 23 = 010111

|  |
| --- |
| l = 3 |
| 148 |
| 16 |
| 18 |
| 22 |

|  |
| --- |
| l = 3 |
|  |
|  |
|  |
|  |

Setting l=3 gives this intermediate (partial) picture…

Continue to next page…

23

COSC 2P03 Week 12 5

# Multiple splits, continued

### Setting l=4 (and thus g=4) gives this final result…

|  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- | --- |
| g=4 | | | | | | | | | | | | | | | |
| 0000 | 0001 | 0010 | 0011 | 0100 | 0101 | 0110 | 0111 | 1000 | 1001 | 1010 | 1011 | 1100 | 1101 | 1110 | 1111 |

|  |
| --- |
| l = 3 |
| 641 |
| 258 |
| 68 |
|  |

|  |
| --- |
| l = 3 |
| 8 |
| 204 |
|  |
|  |

|  |
| --- |
| l = 4 |
| 16 |
| 18 |
|  |
|  |

|  |
| --- |
| l = 4 |
| 148 |
| 22 |
| 23 |
|  |

|  |
| --- |
| l = 3 |
|  |
|  |
|  |
|  |

|  |
| --- |
| l = 2 |
| 288 |
| 1064 |
| 44 |
|  |

|  |
| --- |
| l = 3 |
| 1586 |
| 48 |
|  |
|  |

|  |
| --- |
| l = 3 |
| 120 |
| 700 |
| 575 |
|  |

COSC 2P03 Week 12 6