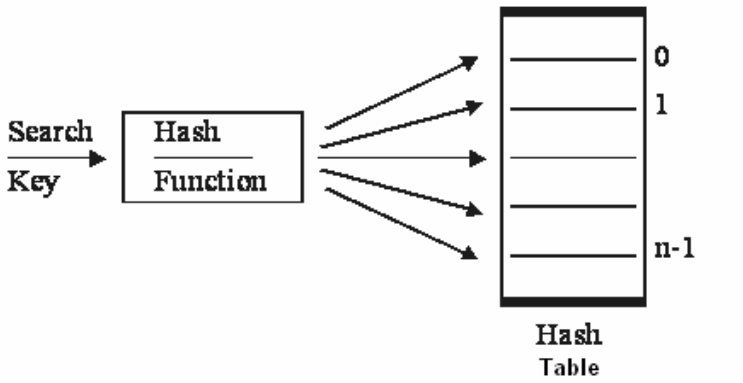
MODULE - 3
HASHING

- In a sequential search, the table that stores the elements is searched successively and the key comparison determines whether an element has been found.
- In a binary search, the table (or data structure), that stores the elements is divided successively into halves and again key comparison determines whether an element has been found.
- In a binary search tree also, the decision to continue the search in a particular direction is accomplished by comparing keys.
- A different approach to searching calculates the position of the key in the table (index) based on the value of the key.
- This means that the search time is reduced from O (n), as in a sequential search, or from O ($\log_2$ n), as in a binary search to O (1).
- This means that the search run time is regardless of the number of elements being searched.
- This technique is referred to as '**hashing**'.
- We need to find a function 'h' that can transform a particular key K, be it a string, number, record or the like into an index in the table used for storing the items of the same type as K.
- The function 'h' is called the **hash function**.
- The memory available to maintain the symbol table is assumed to be sequential.
- The memory is referred to as the **hash table**.
- The hash table is partitioned into buckets h[0],h[1]….
- Each bucket is said to consist of 's' slots, each slot being large enough to hold one record. Usually s=1, and each bucket can hold exactly one record
- An overflow is said to occur when a new key K is hashed into a full hash table.



h (k) = i

- A hash function maps key K into an array index.
- Ideally, you want the hash function to map each key into a unique index.
- The hash function in the ideal situation is called a perfect hash function.
- A perfect hash function maps each search key into a unique location of the hash table.
- A perfect hash function is possible if you know all the search keys that actually occur in the table.
- Usually, you will not know the values of the search keys in advance.

- In practice, hash function can map two or more search keys x and y into the same integer (index).
- That is, the hash function tells you to store two or more items in the same array location
- This occurrence is called a **collision.**
- Collisions occur when the hash function maps more than one item into the same table location.

Hashing can be divided into:

- Static hashing : what has been described till now is static hashing
- Dynamic hashing

Desired Properties of a Hash Function

- Easy and fast to compute
- Minimizes the number of collision / places items evenly throughout the hash table.

Uniform hash function

- is one which does not result in a biased use of the hash table for random inputs
- ie. if x is an identifier chosen at random, then we want the probability that h(x) = i to be 1/b for all table positions.
- Here 'b' is the total number of position (indices) in the hash table.

Several types of uniform **hash functions** are described below:

Mid – Square:

- In the mid-square method, the key is squared and our appropriate number of bits from the middle of the square is taken to obtain the table index.
- Since the middle bits of the square usually depend on all the characters in the key, different keys are expected to result in different hash addresses with high probability, even when some characters are the same.
- If the key is a string, it has to be preprocessed to produce a number.

For example, if the key is 3, 121, then 3121^2 = 9, 740,641

And for a 1000 – cell table h (3121) = 406 which is the middle part of 3121^2

- The number of digits to be used to obtain the index / address depends on the table size.
- If 'r' digits are used, the range of values is 10^r so the size of the hash table is to be chosen as a power of two when this kind of scheme is used.

Division [Modulo Arithmetic]

- A hash function must guarantee that the number it returns is a valid index to one of the table cells.
- The simplest way to accomplish this is to use module arithmetic to create a hash function.

h (k) = K mod Table_size Where K is the key Table_size = size of (table)

For example, if Table_size is 101, h(k) = k mod 101

- Now h (k) maps any integer K into a range 0 through 100.
- For h(k) = K mod table_size, many K's map into table (0), many K's map into
- table[1], and so on.
- That is, collisions occur.
- However, you can distribute the table items evenly over the entire table – thus reducing collisions, by choosing a prime number as table_size.

Folding

- In this method, the key is divided into several parts.
- These parts are combined or folded together and are often transformed in a certain way to create the target index.
- There are two types of folding
    - Shift folding
    - Boundary folding

Shift folding

- The key is divided into several parts and are put underneath one another and then processed. For example, the number 123456789 can be divided into three parts 123, 456, 789 and then these three parts can be added.
- The resulting number is 1,368 and if the index is of three digits, it can be divided module 1000.

Boundary folding

- The key is seen as being written on a piece of paper that is folded on the borders between different parts of the key.
- In this way, every other part will be put in reverse order.
- For example, consider the same three parts 123, 456 and 789
- The first part 123 is taken in the same order, then the piece of paper with the second part is folded underneath it so that 123 is aligned with 654 (which is the reverse of the second part 456)
- When the folding continues, 789 is aligned with the two previous parts.
- The result is 123 + 654 + 789 = 1,566
- This process is simple and fast, especially when bit patterns are used instead of numerical values.
- A bit oriented version of shift folding is obtained by applying the exclusive-or operations

Digit Analysis

- This method is particularly useful in the case of a static file where only a part of the key is used to compute the address / index.
- For the above number : 123456789, this method might use the first four digits (1234), last four digits (6789), the first two combined with the last two (1289), or some other combination
- Each time, only a position of the key is used.

- If this portion is chosen carefully, it can be sufficient for hashing, provided the omitted portion distinguishes the keys only in an insignificant way.

For example – In some university settings, all international students ID number start with 999 Therefore, the first three digits can be safety omitted in a hash function that uses student ID's for computing table indices.

- Similarly, the starting digits of the ISBN code are the same for all books published by the same publisher
- Therefore, they can be safety excluded from the computation of address if a data table contains only books from one publisher.

Collision Resolution

- For almost all hash functions, more than one key can be assigned to the same position.
- There are two factors that can minimize the number of collisions – hash function and table sizes, but they cannot completely eliminate them.
- Two general approaches to collision resolution are common.
- One approach allocates another location within the hash table to the new item (key)
- A second approach changes the structure of the hash table so that each location in the hash table can accommodate more than one item.

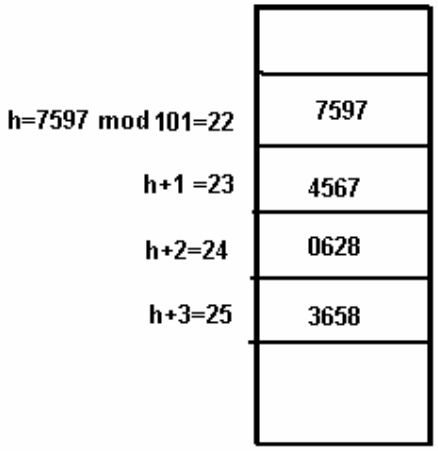
Several collision resolution schemes are described below:

Open Addressing

- During an attempt to insert a new item into a table if the hash function indicates a location in the hash table is already occupied, you probe for some other empty location in which to place the item.
- The sequence of locations that you examine is called the probe sequence.
- Such schemes are said to use open addressing.
- The condition is that once you have inserted items using this scheme, you must be able to find it efficiently.
- i.e. for searching and deleting, you must be able to reproduce the probe sequence that the insert operation used.
- The difference among the various open addressing schemes is the method used to probe for an empty location.
- Three open addressing schemes are described below:

1. Linear Probing

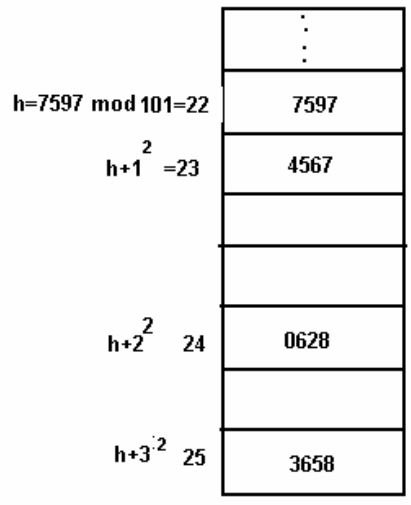
- In this simple scheme to resolve a collision, you search the hash table sequentially, starting from the original hash location.
- i.e., if table[h(search key)] is occupied, your check table [h (search key) +1], [h (search key) +2], and so on until you find an available location.
- The example below illustrates the placement of four items that all hash into the same location table (22) of the hash table, assuming a hash function h(x) = x mod 101



- Linear probing however has a tendency to create clusters in the table.
- That is, the table contains groups of consecutively occupied locations.
- This phenomenon is called **primary clustering**.
- If a cluster is created, it has a tendency to grow, because the empty cells following the clusters have a much greater chance to be filled than other position.
- Thus, one part of the table might be quite dense, even through another part has relatively few items.
- Primary clustering causes long probe searches and therefore decreases the overall efficiency of hashing.

2. Quadratic Probing

- Primary clusters can be eliminated by adjusting the linear probing scheme.
- Instead of probing consecutive locations from the original hash location, you check location table[h (search key) + 1^2], table[h (search key + 2^2], table [h (search key) + 3^2] and so on until you find an available location.



- The figure below illustrates that scheme – called quadratic probing – for the same items that appear in the earlier figure.

- Unfortunately, when two items hash into the same location, quadratic probing uses the same probe sequence for each item.
- This phenomenon called **secondary clustering** delays the resolution of the collision.

3. Double Hashing

- Double hashing drastically reduces clustering.
- The probe sequences that linear and quadratic probing uses are key independent.
- For example, linear probing inspects the table location sequentially no matter what the hash key is.
- In contrast, double hashing defines key dependent probe sequences.
- In this scheme, the probe sequence still searches the table in a linear order, starting at the location h(key) but a second hash function h2 determines the size of the steps taken
- Therefore, a hash address and a step size determine the probe sequence.
- For the second hash function h2, certain guidelines have to be followed:
    - h_2 (key) $\neq 0$ (non zero step size)
    - $h_2 \neq$ h1 (to avoid clustering)

For example let h1 and h2 be the primary and secondary hash function defined as

- h_1 (key) = key mod 11
- h_2 (key) = 7 – (key mod 7)

where a hash table of only 11 items is assumed.

- If key = 58, h_1 hashes the key to table location 3 (58 mod 11) and h2 indicates that the probe sequence should take steps of size 5 (7 - 58 mod 7)
- In other words, the probe sequence will be 3,8,2,7,1,6,0,5,10,4,9.
- If key = 14, h1 hashes the key to table location 3 (14 mod 11) and the probe sequence will be 3,10,6,2,9,5,1,8,4,0
- We can see that each of these probe sequences visits all the table locations.
- This occurs, if the size of the table and the size of the probe step are relatively prime, i.e, their greater common divisor is 1.
- Since the size of the hash table is commonly a prime member, it will be relatively prime to all step sizes.

Restructuring the Hash Table

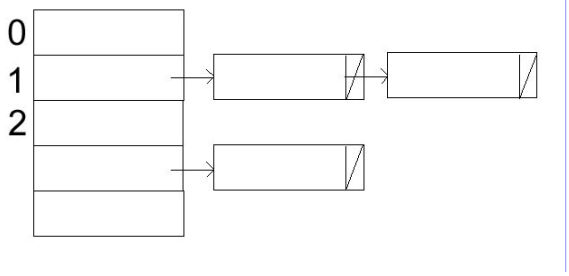
Another way to resolve collision is to change the structure of the hash table, so that it can accommodate more than one item in the same location.

1. Buckets or Bucket Addressing

- If you define the hash table so that each location is itself an array called a bucket, you can store multiple items that hash in to a location.
- A bucket is a block of space large enough to store multiple items.
- The problem with this approach is that if the bucket is already full, then an item hashed to it has to be stored somewhere else.
- If the buckets size is too small, you will only have postponed the problem of collision until (B) items map in to some array location.
- If you attempt to make B large enough so that each array location can accommodate the largest number of items that might map into it, you are likely to waste a good deal of storage.

2. Separate Chaining

- In chaining, each position of the table is associated with a linked list whose nodes store the keys and also references (points) to the next node.
- In this method, the table can never overflow, because the linked lists are extended upon the arrival of new keys as illustrated below:



- Colliding keys are put on the same linked list.
- Each location of the hash table contains a pointer to a linked list.
- For short linked lists, this is a very fast method, but increasing the length of these lists can significantly degrade retrieval performance.
- Performance can be improved by maintaining an order on all these list, so that, for unsuccessful searches an exhaustive search is not required in most cases.

Insert As, A2, A3, B5, C2

0		
1		
2	A2	B2
3	A3	C2
4		
5	A5	B5

Collision resolution with buckets and linear probing method

The Efficiency of Hashing

- The Analysis of the average case efficiency of hashing involves the load factor α, which is the ratio of the current number of items in the table to the maximum size of the hash table.

$$\alpha = \frac{\text{Current number of tables items (N)}}{\text{Size of the Hash table}}$$

- α is a measure of how full the hash table is.
- As the table fills, α increases and the chance of collision increases, so search times increases.
- Thus, hashing efficiency decreases as α increases.
- In order to increase efficiency, you should estimate the largest possible number of items in the table and the select the hash table size so that α is small.
- Hashing efficiency for a particular search also depends on whether the search is successful.

- An unsuccessful search requires more time in general than a successful search.
- For linear probing, the approximate average number of comparison that a search requires is

$$\frac{1}{2}\left[1+\frac{1}{1-\alpha}\right] \text{ for a successful search}$$

$$\frac{1}{2}\left[1+\frac{1}{(1-\alpha)^2}\right] \text{ for an unsuccessful search}$$

- For example- For a table that is $\frac{2}{3}$ full$\left(\alpha=\frac{2}{3}\right)$, an average unsuccessful search will require at most five comparisons (probes) while an average successful search might require at the most two comparisons.
- For Quadratic probing and Double hashing, the average number of comparisons that a search requires is

- Both methods require fewer comparisons than linear probing.
- For example – For a hash table that is 2/3 full (α =2/3)
- An average unsuccessful search might require at most three comparisons while an average successful search might require at most two comparisons.
- As a result, you can use smaller hash table for both quadratic probing and double hashing than you can for linear probing.
- Since all the three methods are open addressing schemes, all three methods suffer when you are unable to predict the number of insertions and deletion that will occur.
- If the hash table is too small, it will fill up and search efficiency will decrease.

Dynamic & Extendible Hashing

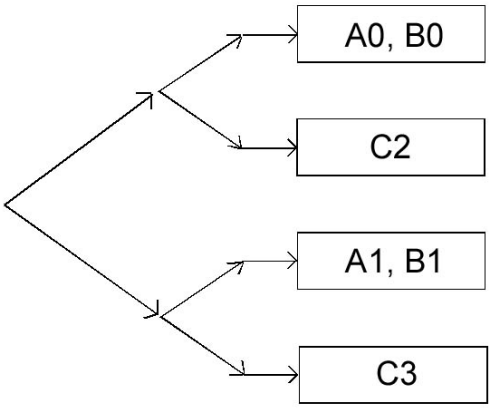
- In traditional hashing schemes, one must statically allocate a portion of memory to hold the hash table.
- This hash table is used to point to the identifiers (keys) or to the pages used to hold the identifiers.
- In either case, if the table is allocated to be as large as possible, then space can be wasted.
- If it allocated to be too small, then when the data exceed the capacity of the hash table, the entire file must be restructured, a time consuming process.
- The purpose of dynamic hashing is to retain the fast retrieval time of conventional hashing while extending the technique so that it can accommodate dynamically increasing and decreasing file size without penalty.
- There are several techniques in the category of directory based hashing
    - Dynamic hashing (Larson, 1979)
    - Extendible hashing (Fagin et al, 1979)
    - Expendable hashing (Knott, 1971)
- All three methods distribute keys among buckets (data storage) in a similar fashion.
- The main difference is the structure of the index (directory)
- In dynamic and expandable hashing a binary tree is used as an index of buckets. On the order hand, in extendible hashing, the directory is maintained as a table.

Dynamic hashing (using Directories)

- Consider an example in which an identifier consists of two characters, and each character is represented by three bits.
- The figure below given a list source of these identifiers

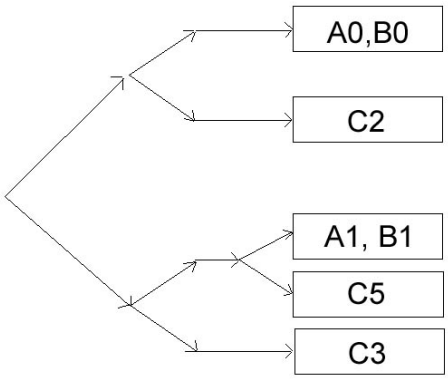
Identifier	Binary Representation
A0	100000
A1	100001
B0	101000
B1	101001
C0	110000
C1	110001
C2	110010
C3	110011
C5	110101

- Consider an example in which an identifier consists of two characters and each character is represented by three bits.
- The figure below gives a list of some of these identifiers.
- Consider placing these identifiers into a binary tree base directory which has four pages.
- Each page can hold at most two identifiers, and each page is indexed by the two bit sequence 00, 01, 10 and 11 respectively.
- We select the bits from least significant to most significant.
- The result of placing A0, B0, C2,A1, B1 and C3 is shown below:

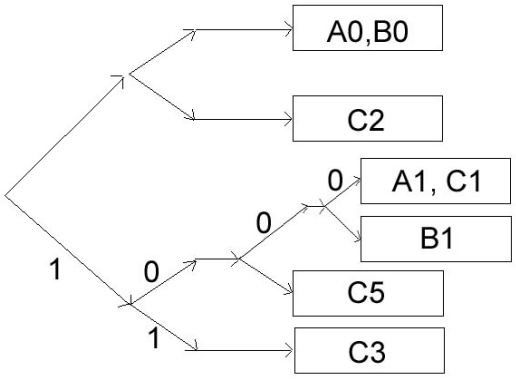


- Branching at the root determined by the least significant bit.
- If this bit is zero, the upper branch is taken; otherwise, the lower branch is taken.
- Branching at the next level is determined by the second last significant bit and so on.
- A0 and B0 are in the first page since their two order bits are 00.
- The second page contains only C2.
- To reach this pages, we first branch on the LSB of C2 (i.e. 0) and then on the next bit (i.e. 1)

- The third page contain A1 and B1 (bit pattern 01)
- We use the term **trie** to denote a binary tree in which we locate an identifier by following its bit sequence.
- The nodes of this trie always branch in two directions corresponding to 0 or 1.
- Only the leaf nodes of the trie contain a pointer to a page.
- Now if we try to insert a new identifier, say C5
- Since the two low order bits of C5 are 1 and 0, must place it in the third page.
- However, this page is full and an over flow occurs.
- A new page is added and the depth of the trie increases by one level.
- This is shown below:



- If we now try to insert the new identifier C1, as overflow of the page containing A1 and B1 occurs.
- A new page is obtained and the three identifiers are divided between the two pages according to their four lower order bits.



Drawbacks

1. Access time for a page depends on the number of bits needed to distinguish the identifiers
2. If the identifiers have a skewed distribution, the tree is also skewed.

Both these factors increase the retrieval time.

Solutions

- To avoid the skewed distribution of identifier, a hash function is used. This function takes the key and produces a random set of binary digits.
- Instead of a trie, if a directory is used, the long search down the trie can be avoided.

Extendible Hashing

- This method incorporate the techniques mentioned above to above the drawbacks of dynamic hashing.
- A directory is a table of page pointers.
- To find the page for an identifier, we use the integer with binary representatives equal to the last 'k' bits of the identifier.
- The page pointed at by this directory entry is searched.
- The following figure shows the directories corresponding to the previous three tries

Bits	Page pointer	Page containing Key values	Bits	Page pointer	Page containing Key values
00	*a* →	A0	000	*a* →	A0, B0
01	*c* →	A1, B1	001	*c* →	A1, B1
10	*b* →	C2	010	*b* →	C2
11	*d* →	C3	011	*e* →	C3
			100	*a* →	
			101	*b* →	C5
			110	*d* →	
			111	*e* →	

- The first directory contains four entries indexed from 0 to 3 (in binary)
- Each entry contains a pointer to a page (shown as arrow in the figure)
- The letter above each pointer is a label.
- The page labels were obtained by labeling the pages of the first trie, top to bottom, beginning with label a.
- To see the correspondence between the directory and the trie, notice that if the bits in the directory index are used to follow a path in the trie (beginning with the LSB), we will reach the page pointed at by the corresponding directory entry.
- Page 'a' of the second directory has two directory entries (000 and 100) pointing to it. Page 'b' has two pointers to it, page 'c' has one pointer etc.
- Using a directory to represent a trie allows the table of identifiers to grow and shrink dynamically.

- In addition, accessing any page requires only two steps.
- In the first step, we use the hash function to find the address of the directory entry, and in the second, we retrieve the page associated with the address.
- Suppose a page identified by ‘i’ bits overflows.
- We allocate a new page and rehash the identifiers in to those two pages.
- The identifiers in the both pages have their low order ‘i’ bits in common.
- We refer to these pages as ‘buddies’.
- When the number of identifiers in two buddy pages is coalesce the pages into one.