mood-book

Dictionaries

→ Key-value pair elements.

→ Key used to search/locate the elements in the list.

→ It is a one-way connection.

→ No two pairs are allowed to have same key. The key value must be unique.

→ But 2 different keys can refers to the same value.

→ Dictionary can either be ordered (or) unordered.

Ordered:

In ordered dictionary, all data elements are placed either in ascending or in descending order of the keys. Here it allows sequential access of (key, value) pairs.

Unordered:

No particular order of data elements is maintained. Here it allows random access of data elements i.e., (key, value) pairs.

Major operations on dictionary :

* int size() - returns the no. of elements in dictionary.

* bool isEmpty() - checks if the dictionary is empty.

* Value get(Key) - returns value for a given key.

* bool put(Key, Value) - Inserts given key-value pair to dictionary. If already the key is present then it associates with new value. Returns true if successful else false.

* bool remove(Key) - removes key-value pair based on the given key.

Representation :

It can be of several ways, such as:

* Linked List
* Skip List
* Hashing
* Tree (Balanced trees)

- The efficient way of ordered dictionary is (Trees) - average O(log N) complexity
- For unordered is Hashtable - average O(1) Complexity.

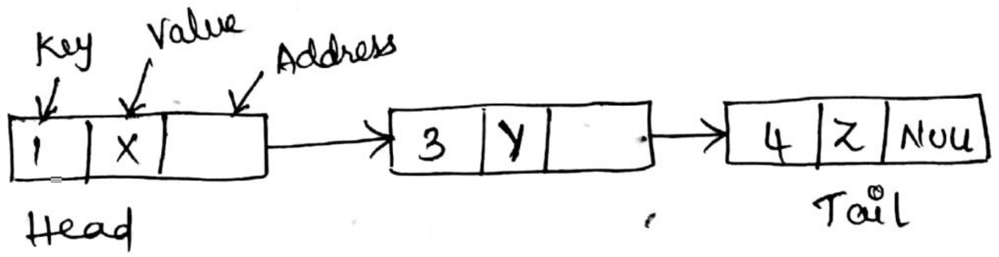
Linear List representation:

In this, every node is a dictionary (key, value) pair stored sequentially but the keys are not stored in the sorted order.

Each element in the list must have some data type. The list can also have elements of different data types. The operations of the list do not depends on the elements data type.

List can be used for list of integers, list of characters, even list of lists.

Eg:



The major operations on linear list representation are,

* Insertion
* Deletion
* Search.

Here, deletion and search depends on the key value.

The worst complexity of a search operation is O(n). Here, random access is not possible. Since, ordering of keys is not maintained here, binary search cannot be applied.

<u>Skip List representation:</u>

It was invented by W. pugh in 1989.

In Skip list, the ordering of keys are maintained. It allows random access of (key, value) pair elements, As the order of key is always in ascending order , binary search can be applied that has the time complexity of O(log n) for insertion, deletion and search.

Skip list representation is simple, efficient and dynamic.

The idea of Skip list is to create multiple layers so that we can skip some nodes while doing some operations.

Properties of perfect skip list:

1. All elements should be in ascending order
2. The number of levels should be (log N)
3. Each higher level contains exactly half the elements present in the level below it.

The higher level lists let you skip over many elements.

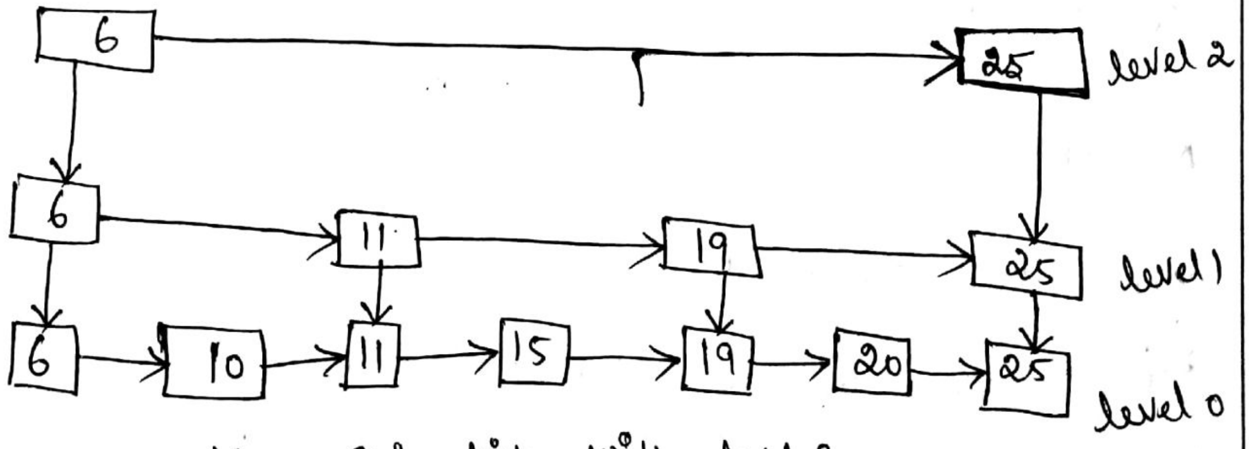


fig: Skip list with level 3

Operations:

* Insert
* delete
* Search

Insertion:

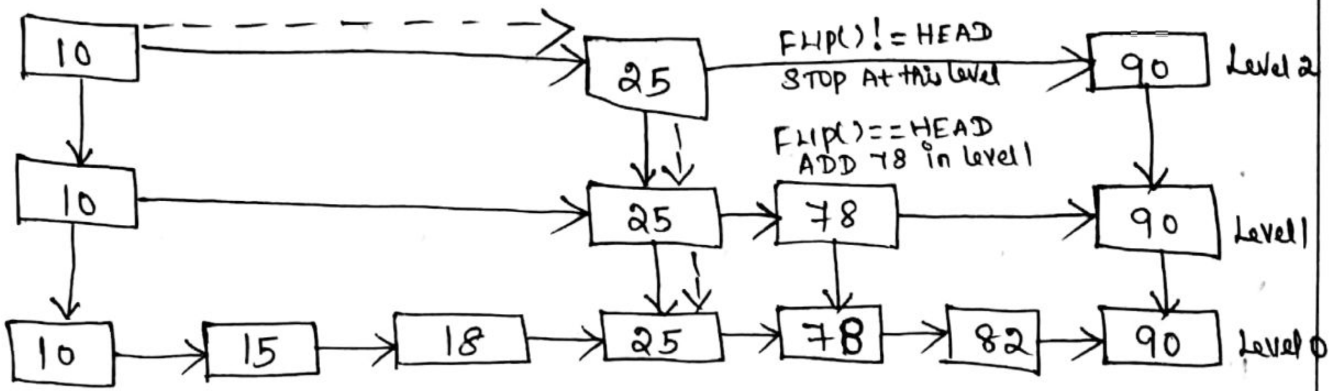
During insertion the key is searched. If it is found then the value is updated with new value, else a node is created and the value is inserted. This results in the rearrangement of entire skip list structure.

For efficient implementation of insert operation, we can flip a coin to decide whether to promote a node to the higher level or not. As the flipped coin has 1/2 chances of resulting in a tail and 1/2 chances of resulting in a head.

Algorithm:

```
Step 1: call search(k)
Step 2: Insert a node with k in level 0.
Step 3: Let i=1
        while FLIP() == "head"
                Insert node into level i as well.
        i++.
```

Insert 78,



A key = 78 is to be inserted.

It searches for the key value from the higher level and move downwards to the lower levels upto level 0 until it finds the desired position to insert the value. When the value is inserted at level 0, then a coin flips to give head/tail. If it is head, then move to the immediate upper level to insert the new node. After insertion, again the coin flips. Now, it is tail and hence the process stops there.

Deletion:

As like insertion, during deletion the entire skip list structure is rearranged.

The deletion operation in the Skip list is a straight forward approach. It Starts by Searching the Key value of the list and delete the corresponding Key-Value pair. The link of the following node which was with the deleted node should be assigned to the node that was left to the deleted node.

Algorithm:

Step 1: Search the element k to be deleted by calling Search (k).

Step 2: If found, delete the element k from all levels.

Search Starts from higher level. If the Search key is found then delete the node from that level to level 0.

Searching:

The Starting point for Searching an element in the Skip list is the Smallest element in the top most layer.

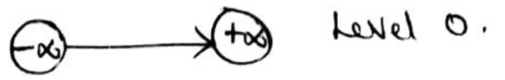
Algorithm: k ← element to be Searched, key ← top left node.

```
If key>k or End of list, k is not found.
If key==k, Key is matched, k is found.
if k< next key, go down a level.
if k>= next key, go right.
```

Skiplist Insertion:

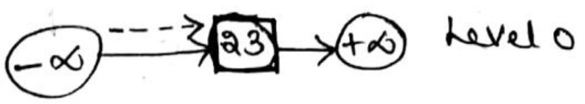
{23, 31, 7, 9, 11, 14, 27} with Level-3.

Step1:

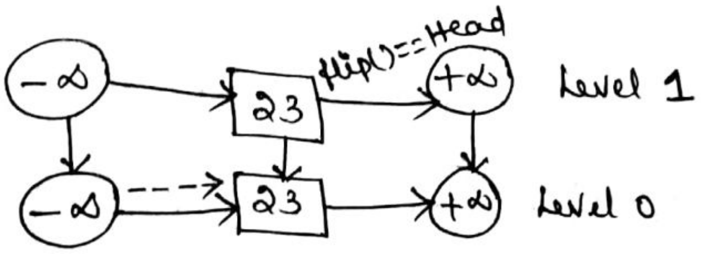


Insert 23,

23 > -∞ and 23 < +∞, so,



if flip()==HEAD then, insert 23 at level 1.

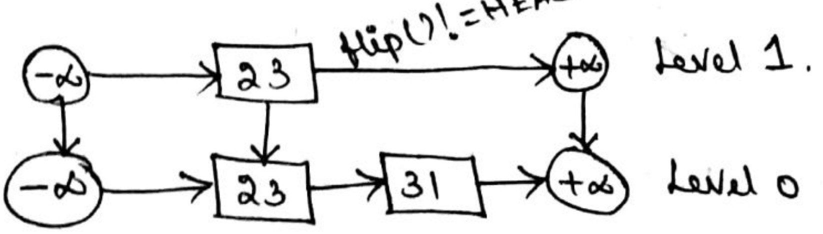


Again flip a coin, if flip() != HEAD then, Stop at this level.

Step 2:

Insert 31,

31 > -∞, 31 > 23, 31 < -∞. Hence,



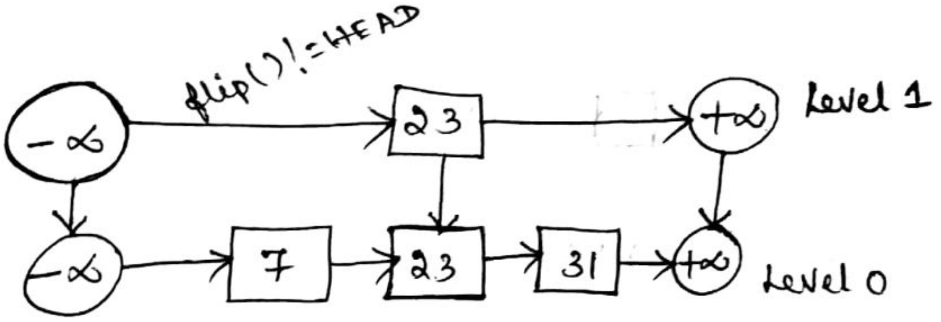
Flip a coin now. If flip()==Head then insert the element at Level 1, otherwise Stop at Level 0.

Now, flip()!=HEAD. So, Stop the process.

Step 3:

Insert 7.

$7 > -\infty$ and $7 < 23$. Hence,



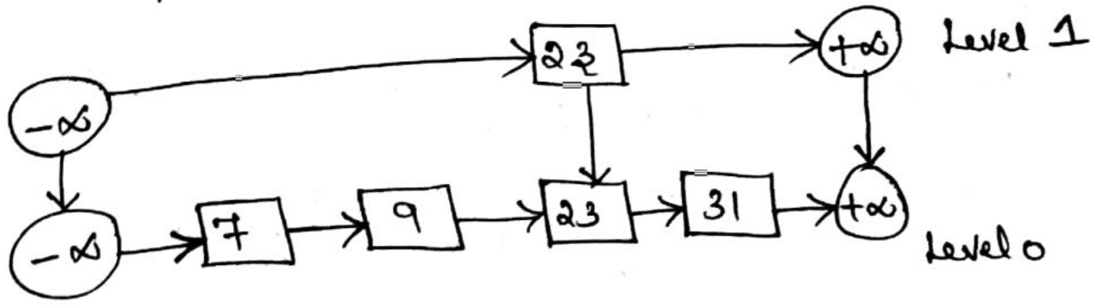
Flip a coin, and See if flip()==Head.

flip()! =HEAD. So, Stop the Process.

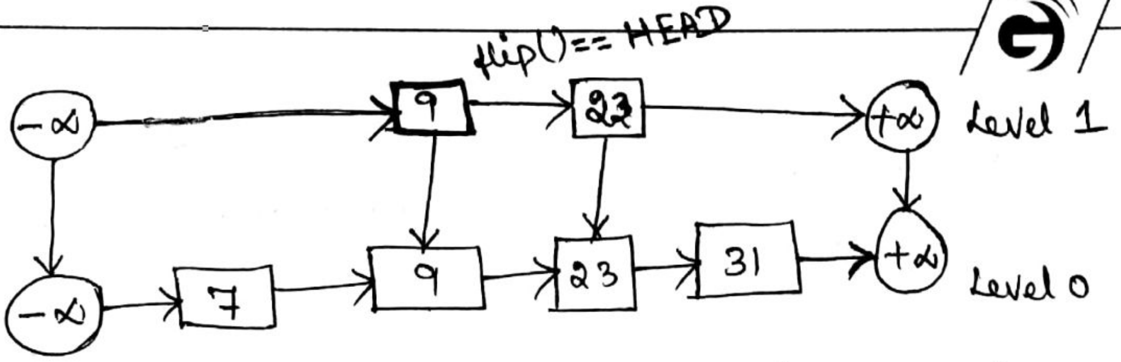
Step 4:

Insert 9.

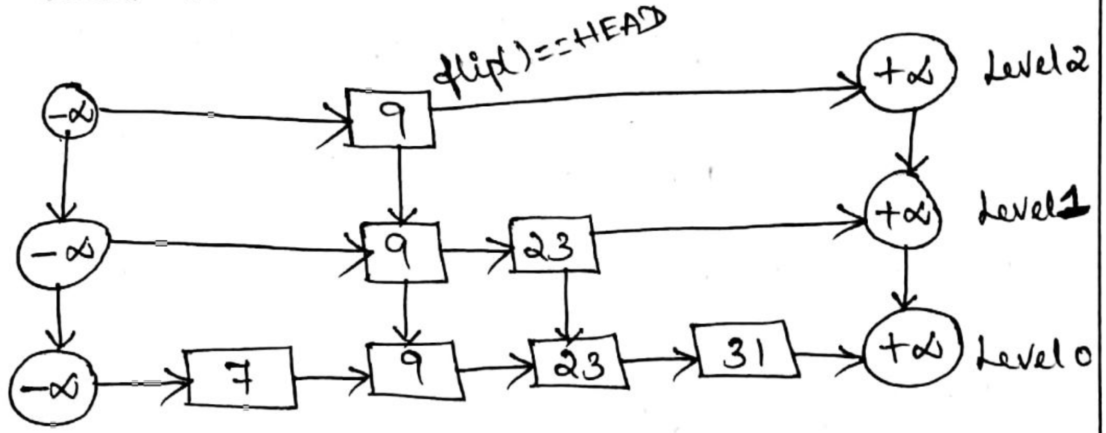
$9 > -\infty$, $9 > 7$, $9 < 23$, Hence,



Now, Flip a coin. if flip==Head, insert at level 1.

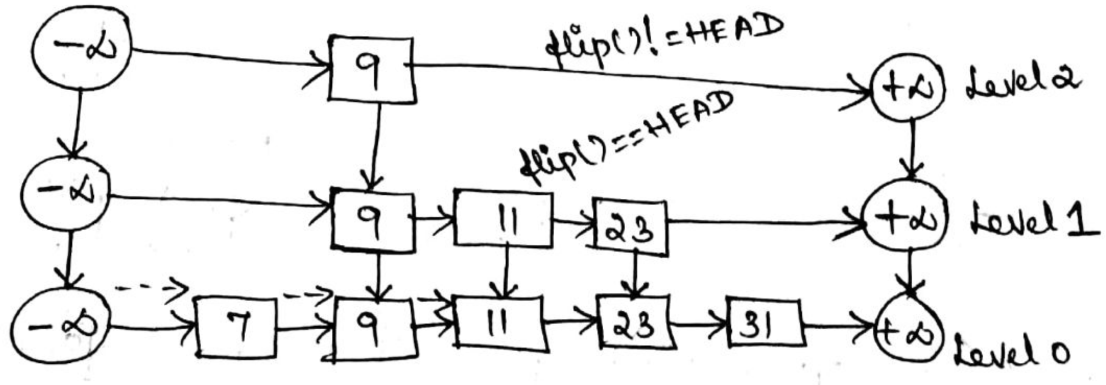


Again flip a coin. If flip==HEAD, insert 9 at level 2.



Step 5:

Insert 11. $11 > -\infty$, $11 > 7$, $11 > 9$ and $11 < 23$.

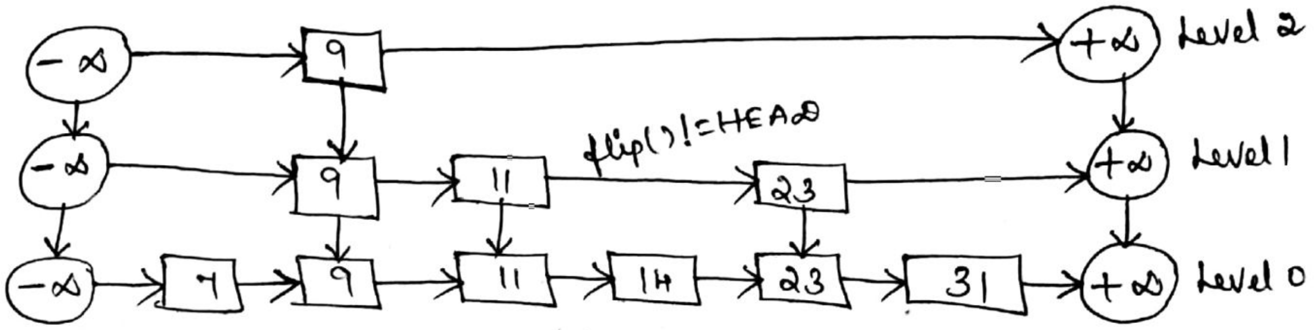


flip a coin. Now, flip()== HEAD, So, 11 is inserted at Level 1. Again flip a coin, flip()!=HEAD. So, Stop at level 1.

Step 6 :

Insert 14.

14 > -∞, 14 > 7, 14 > 9, 14 > 11 and 14 < 23. So,



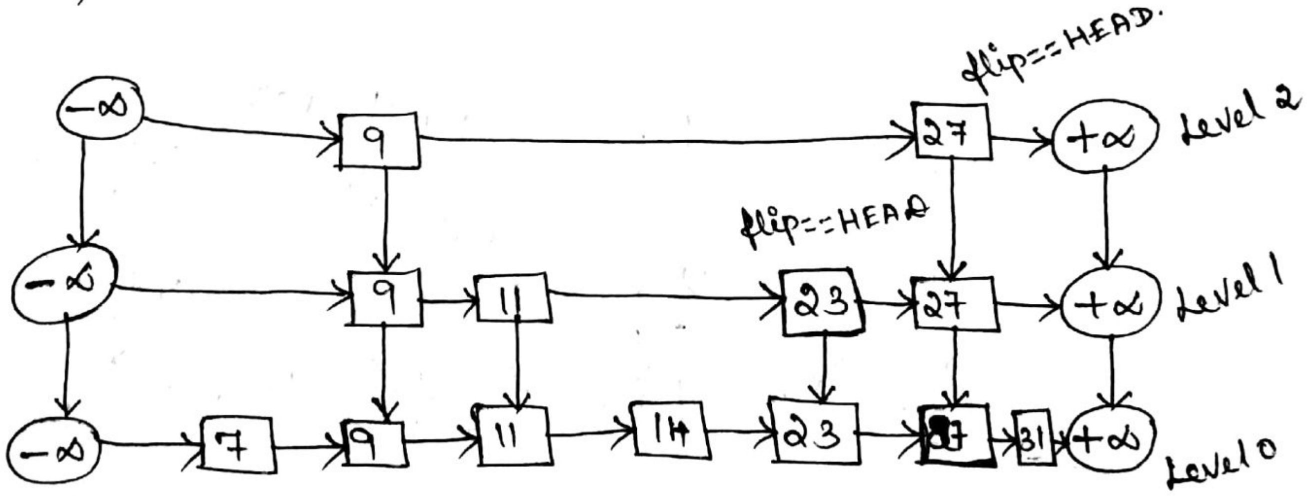
Now, flip a coin, flip()!=Head. So, Stop the process at level 0.

Step 7 :

Insert 27.

27 > -∞, 27 > 7, 27 > 9, 27 > 11, 27 > 14, 27 > 23, 27 < 31.

Hence,



Now, flip a coin, flip()==HEAD. Hence, insert 27 at level 1. Again flip a coin, flip()==HEAD. So, insert 27 again at Level 2. Then, flip()!=HEAD. Hence, Stop the process at Level 2.

Hashing:

* In linear search we compare the key element with each element of the list.

* In binary search the key element is compared with the middle element of the list. therefore the search techniques described so far to find the required key data are based on.

① Performing several tests by comparing keys with the elements of the given list.

② The order in which the elements are inserted affects the no. of keys to compare.

* Another searching technique that avoids no. of comparisions and goes directly where the required data is present is called "Hashing".

⇒ Hashing is a technique to convert a range of keys values into a range of indexes of an array.

Eg: Convert 100 student list into array[100] i.e., 0 to 99 indexes.

if 10,000 student list is required then if we take array[10,000] then it is waste of space incase of 100 students. So, we use hashing to avoid such waste spaces and fit all of them in array[100].

Hash Table:

* Hash table is an efficient method of storing data into the positions of a hash table. A bucket array is used for a hash table. i.e., array of size n.

* Each and every cell in the array is treated as a bucket and it is used to hold the pair key-values. A bucket can store only one element, when two keys are mapped to the same

bucket then collision occurs.

* Collisions are avoided by using hash function. Hash tables are commonly used for symbol table by a language translator in compiler design; associative arrays, sets and caches.

* Hash table is a data structure that can be provide constant time O(1) lookup on an average and speed up information (key) searching.

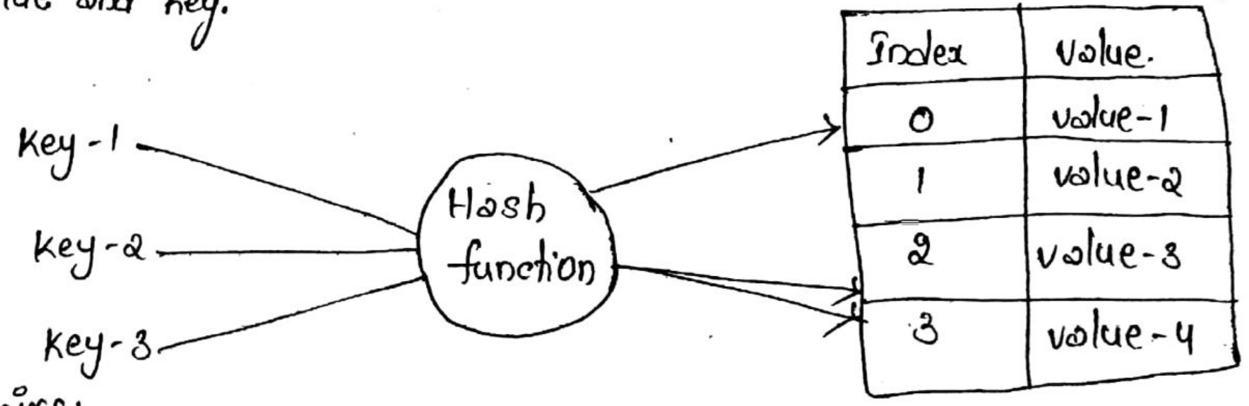
* Using the hash function all the operations on the hash table such as searching for a key-value, inserting a key-value (or) deleting a key-value are performed.

Hash Function:

* The function that maps key space into address space is called a "Hash function". Hashing is a key to address transformation in which the keys map to address in a list. The array in which the address space is defined is called "Hash Table" (buckets)

* So the hash function generates the address that covers the entire set of indices in the hash table, where the record is located.

Eg: Hash table of size "20" and following items are to be stored. Items are in (key, value) format i.e., pair. A bucket contains pair of value and key.



Pairs:

~~(1,20) (2,~~

Pairs:

(1,20) (2,70) (42,80) (4,25) (12,44) (14,32) (17,11) (13,78) (37,98)

S:NO	Key	Hash	Array Index
1	1	1 % 20 = 1	1
2	2	2 % 20 = 2	2
3	42	42 % 20 = 2	2
4	4	4 % 20 = 4	4
5	12	12 % 20 = 12	12
6	14	14 % 20 = 14	14
7	17	17 % 20 = 17	17
8	13	13 % 20 = 13	13
9	37	37 % 20 = 17	17

* Here both (2,70) (42,80) pairs are hashed to the same array index '2'. So, here collision occurs.

* The hash function that never produces a collision is called a "perfect hash function". However, it is very difficult to form a perfect hash function and it cannot always be found.

* The hash function that minimizes collisions by spreading the elements uniformly throughout the array is called "a Good hash function."

* A Good hash function should have the following characteristics.

① It may be ~~found~~ formed by using some mathematical transformation.

② It should be very easy and Quick to compare.

③ It should produce a relatively random and unique distribution of values with in the hash table in that it produces as few collisions as possible.

④ It must minimize collisions.

* Some of the methods that are used for creating a hash function are as follows:

① The division method
② Mid-square method
③ Truncation method
④ Folding method
⑤ Extraction method.

Division Method:

* Map a key 'k' into one of hash table size (m) slots by taking the remainder of 'k' divided by m (i.e., %), and it is defined as:

Hash (key) = key % Hash table size

$h(k) = k \bmod m$.

Eg: If table size $m = 12$, key = $k = 100$. then

$h(100) = 100 \bmod 12$

$= \underline{4}$

If hash table index starts from '1'. then

$h(100) = 100 \bmod (12+1)$

$= 4+1$

$= 5$

* Values of m such as $m=2^p$ should be avoided.
* Good values for m are primes and are not too close to exact powers of 2. If a set of keys does not contain integers, they must be converted into integers before applying any of the hash functions.

② Mid-Square method:

* In this method, the square of the key is calculated, and then the hash key is obtained by selecting an appropriate number of digits from the middle of the square.
* We can say that the hash key is obtained by deleting digits from both ends of the square of the key.
* The number of digits choosen for the hash key depends on the number of digits allowed in the ~~index~~ index and the same number of digits from the same positions of the square of the key must be used for all the keys.

Eg:

Key	Square of key	Hash key
→ 1724	29<u>72</u>176	72
→ 2457	60<u>36</u>849	36
→ 3241	10<u>50</u>4081	50
→ 4112	16<u>90</u>8544	90
→ 2134	45<u>53</u>956	53

* Here, we take 3rd and 4th positions from left of the square of key value.

③ Truncation method:

* Ignore some digits of the key and use the rest

as the array index. When the keys are alphabets, then their numerical equivalents can be considered. The problem with this equivalents can be considered. The problem with this approach is that there may not always be an even distribution throughout the table.

Eg: If student number are the key, 928324312 then from the 9- digit number select just 4rth and 8th position digits as the index i.e., 31 as the index.

④ Folding Method:

* Partition the key into several pieces i.e., two (or) three (or) more parts. Each of the individual parts is combined using any of the basic arithmetic operations such as addition (or) multiplication. The resultant number is truncated.

Eg: Consider a number 123456.

Partition the number into three parts ⇒ 12/34/56.

Add the three parts ⇒ 12+34+56 = 112.

①12
$\hookrightarrow$ Avoid first digit.

Truncating the number ⇒ 12.

123456 stores in 12 index.

⑤ Extraction method:

* In this method computing the address uses only a part of the key.

Eg: student number → 928324312.

→ The extraction method may use first four digits. 9283 (or)

→ The extraction method may use last four digits.

4312
(or)

→ The combination of the first two and last two digits.

9212 (or)

any other combination to store the number in particular index.

Collisions:

* When a hash function maps two different keys to same location then collision occurs. Thus the corresponding record cannot be stored in the same location.

eg: Array size = 4. Key value ⇒ 9, 13, 17. Using simple modules operation for hashing.

i.e., $h(k) = k \bmod 4$. then,

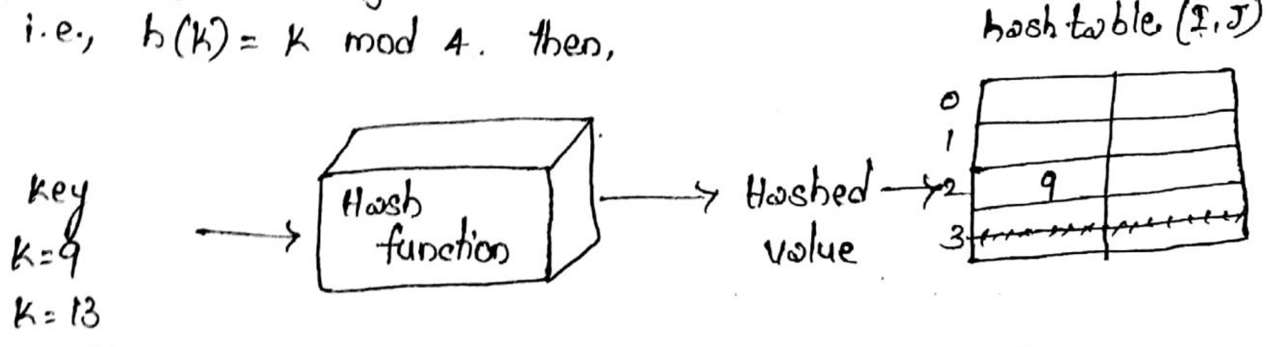


fig: Collision.

Collision Resolution Technique:

① Separate chaining — An array of linked list implementation.

② Open addressing (or) closed hashing — uses an array based implementation.

① Separate chaining:-

* The hash table is implemented as an array of linked lists. To insert an item into the table, it is appended to the corresponding linked lists. The linked list to which it is to be appended is determined by hashing the inserting item.

* This technique is known as chaining (or) separate chaining (or) open hashing because it has separate links.

* Separate chaining is depicted as shown below.

→ Inserting an item 'e' that hashes at index 'i' is simply inserted into the linked list at position 'i'. Synonyms (same type) are chained in the same linked list.

→ Retrieval of an item 'e' with hash address 'i' is simply retriev-ing it from position 'i' of the linked list.

→ Deletion of an item 'e' with hash address 'i' is simply retrieving it from position 'i' of the linked list and deleting (or) disconnec-ting it from the linked list.

Insert: $A_5, A_2, A_7, B_5, A_9, B_2, B_9, C_2$.

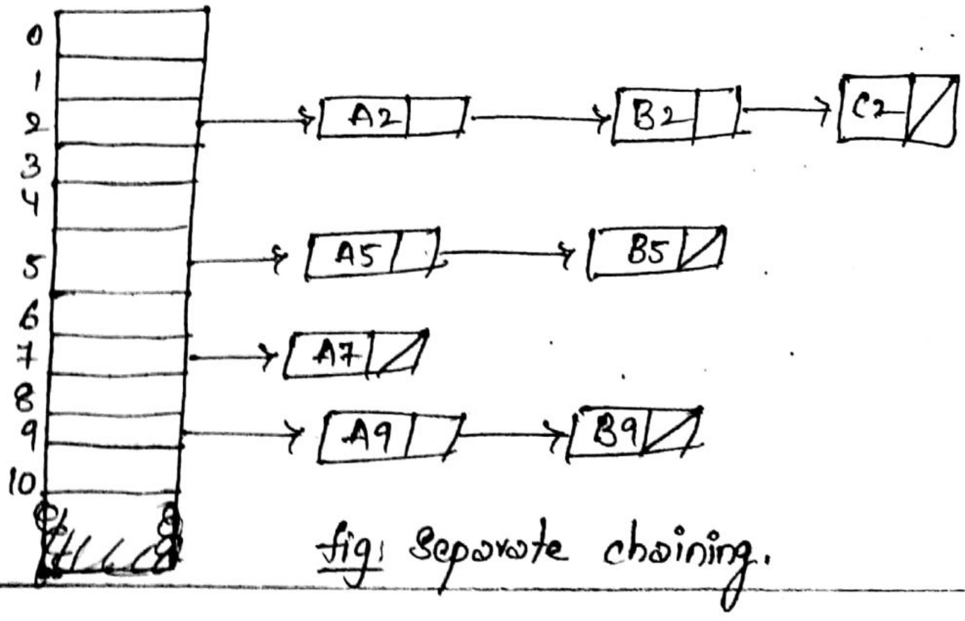


fig: Separate chaining.

Eg:

* Load the keys 23, 13, 21, 14, 7, 8 and 15, in the same order, in a hash table of size 7 using separate chaining with the hash function h(key) = key % 7.

h(23) = 23 % 7 = 2.

h(13) = 13 % 7 = 6

h(21) = 21 % 7 = 0

h(14) = 14 % 7 = 0 (collision)

h(7) = 7 % 7 = 0 (collision)

h(8) = 8 % 7 = 1

h(15) = 15 % 7 = 1 (collision)

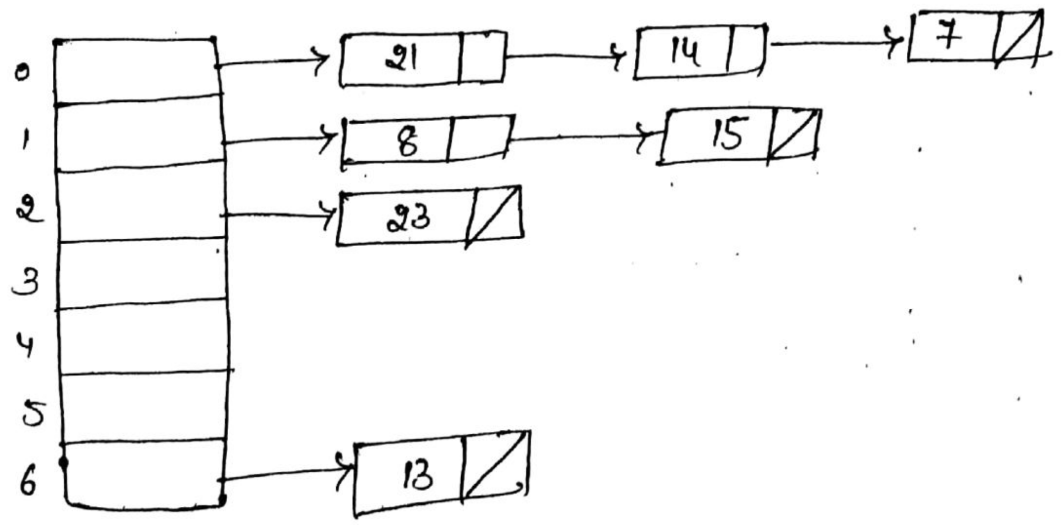


fig: Example of separate chaining.

Operations on separate chained hash table:

* the frequently used search, insert and delete operations on separate chained hash table are discussed here.

Searching: Hash function of the element that is to be searched should be computed. Access the bucket with corresponding hash value and proceed to search the chain of needs sequentially. If the element is found, then search is successful, else it is an unsuccessful search.

Insertion:

* Inserting an element 'e' into the hash table also requires computing hash function on the element to determine the bucket. After finding the corresponding bucket it is similar to inserting an element into a singly linked list.

* If the element in each chain are maintained in either ascending (or) descending order, then it will be less expensive to perform all the operations.

Deletion:

* Deleting an element 'e' from the hash table also requires computing hash function on the element to determine the bucket. After corresponding bucket it is similar to deleting an element from a singly linked list.

Performance Analysis:

* The length of the chain of nodes corresponding to a bucket decides the complexity of separate chained hash tables.

* Its best case complexity of search operation is O(1). The worst case complexity is O(n). This occurs when all the 'n' elements are mapped to the same bucket and the searched element is the last element in the chain of 'n' nodes.

Advantages:

* Collision resolution is simple and efficient.
* The hash table can hold more elements without the large performance deterioration of open addressing (the load factor can be 1 (or) greater).
* The performance of chaining declines must more slowly than open addressing.
* Deletion is easy - no special flag values are necessary.
* Table size need not be a prime number.
* The keys of the objects to be hashed need not be unique.

Disadvantages of separate chaining:-

* It requires the implementation of a separate data structure for chains, and a code to manage it.
* The main cost of chaining is the extra space required for the linked list.
* For some languages, creating new nodes (for linked lists) is expensive and slow down the system.

Open Addressing:

* In open addressing all the record entries are stored in the bucket array. The location of the hash table are termed as buckets. Each bucket is portioned into slots as shown in the below diagram. The buckets are examined whenever a new entry has to be inserted, starting with hashed-to slot using some probe (small memory (or) testing) sequence until an empty slot is

found.

* To search for an ~~empty slot is found.~~ entry all the buckets are scanned in the same sequence until the entry is found or an empty slot is found which indicates that there is no such entry in the table.

* The location (or) address of an item is not determined by the hash value with open addressing. Each location in an array will be in the EMPTY, DELETED (or) OCCUPIED state. If the state of a location is OCCUPIED then it contains key and data otherwise it doesnot have any value. Initially all the locations in the array are in the EMPTY state.

* When an item at a particular location is deleted then its state will be deleted, not EMPTY.

* The item may be deleted but it is not removed until some other element is inserted in that place. The need of these three states is explained by the following algorithm, which inserts an item into a table.

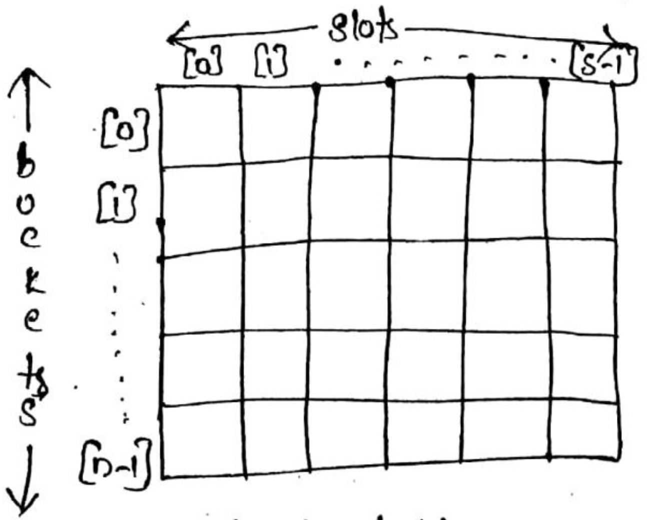


fig: Hash table.

Algorithm:

1. Find the location at which item is to be stored.

   L = H(Key (Item))

2. If location L is not OCCUPIED then insert item.
3. If location L is OCCUPIED find another location using some probe sequence.

   Set location L to ITEM and repeat steps 2 & 3.

4. End.

The following are the wellknown probe sequences:

* Linear probing: in which the interval between probes will be fixed usually ~~toby~~ '1'.

* Quadratic probing: in which interval between probes is increased by adding the successive outputs of a quadratic polynomial to the starting value given by the hash function.

* Double hashing: in which the interval between probes is obtained by using another hash function

Linear Probing:

* Linear probing is a technique for resolving hash collision of values of hash functions. by searching the hash table sequentially for a free location.

* This is done by making use of two values where one is the starting value and another is the interval between successive

values.

* The second value which is the same for all keys repeatedly adds to the starting value until free location is found or entire table is traversed.

* The linear probing function will be.

$h(x) = (h(x) + i) \bmod n$.

where $h(x)$ is the starting value, 'i' is the value added repeatedly to the starting value and, 'n' is the size of the hash table.

* When the 'i' value is '1', the linear probing gives good memory catching through good locality of reference but also results in clustering.

Consider the following example.

Let the table size be 10, keys are 2-digits integers and

$h(x) = x \bmod 10$.

→ Initially all the locations are empty.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]

→ while inserting 16, 18 & 9, there will be no collisions.

$h(x) = x \bmod 10$

$h(16) = 16 \bmod 10 = 6$,

so, insert 16 at the 6th location.

$h(18) = 18 \bmod 10 = 8$, so, insert 18 at the 8th location.

$h(9) = 9 \bmod 10 = 9$, so insert 9 at the 9th location.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
						16		18	9

* Now insert 36, h(36) = 36 mod 10 = 6,

→ collision occurs because the state of location 6 is OCCUPIED using linear probing, the next free location will be (6+1) mod 10 = 7; where the value 36 is to be stored. The state of location 7 is EMPTY, so 36 can be inserted into 7th location.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
						16	36	18	9

* Now insert 76, h(76) = 76 mod 10 = 6,

→ Collision occurs, next location, i.e., ((6+1) mod 10 = 7 which is on [illegible] OCCUPIED state. Then try next location as (7+1) mod 10 = 8, OCCUPIED state. Finally try the location '0' as (9+1) mod 10 = 0, which is in EMPTY state, so insert 76 at the 0th location.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
76						16	36	18	9

* The main disadvantage of linear probing is primary clustering, in the above example when further insertion of 86, 96 and 56 is made, it causes more number of collisions.

* To eliminate clustering each different key should probe the table in a different order.

Search:

* To search an item in the hash table that uses linear probing, start at h(k) and probe consecutive locations until one of the following occurs.

→ Item is found
→ An Empty cell is found (or)
→ Entire table has been scanned.

eg:
* Perform the operations given below, in the given order, on an initially empty hash table of size 10 using linear probing with i=1 and the hash function: h(x) = x mod 10;
insert (18), insert (26), insert (35), insert (8), find (15), find (48), delete (35), delete (40), ~~delete~~(find (8), insert (64), insert (47), find (35)

* the required probe sequences are given by.

$h(Key) = (h(Key)+i) \bmod 10$ where $i=1$

Operation	probe Sequence	Result.
→ Insert (18)	$h(18) = 18 \bmod 10 = 8$	SUCCESS
→ insert (26)	$h(26) = 26 \bmod 10 = 6$	SUCCESS
→ insert (35)	$h(35) = 35 \bmod 10 = 5$	SUCCESS
→ insert (8)	$h(8) = 8 \bmod 10 = 8$	COLLISION
	$h(8) = (8+1) \bmod 10 = 9$	SUCCESS
→ find (15)	$h(15) = 15 \bmod 10 = 5$	COLLISION
	$h(15) = (15+1) \bmod 10 = 6$	COLLISION
	$h(15) = (16+1) \bmod 10 = 7$	FAIL because location '7' does not contain 15, i.e., EMPTY status.
→ find (48)	$h(48) = 48 \bmod 10 = 8$	collision
	$h(48) = (48+1) \bmod 10 = 9$	collision
	$h(48) = (49+1) \bmod 10 = 0$	FAIL, '0' doesnot contain

Operation	Probe Sequence	Result
→ Delete (35)	h(35) = 35 mod 10 = 5	SUCCESS, because location 5 contains 35 and the status is OCCUPIED, changed to DELETE but key 35 is not removed.
→ Delete (40)	h(40) = 40 mod 10 = 4	FAIL, no such key exists in location 4 & status is EMPTY.
→ Find (18)	h(18) = 18 mod 10 = 8	Search success, location 8 contains key 18, SUCCESS
→ insert (64)	h(64) = 64 mod 10 = 4	
→ insert (47)	h(47) = 47 mod 10 = 7	SUCCESS
→ find (35)	h(35) = 35 mod 10 = 5	Fail because location 5 contains 35 but its status is DELETED.

* The following table shows index values and their status.

Index	status	value
0	E	
1	E	
2	E	
3	E	
4	O	64
5	D	35
6	E	26
7	O	47
8	O	18
9	E	8
10	E	
11	E	
12	E	
[illegible]		

Quadratic Probing:

* Quadratic probing is another method for resolving collisions in hash tables. It operates by taking the original hash value and adding successive values of a quadratic polynomial to the starting value.

* Quadratic probing avoids the clustering problem that occurs with linear probing and may also result in probing the same set of alternate ~~cells~~ cells.

* In this method when a collision occurs at location 'l', probes bucket $l+1$, $l+4$, $l+9$ etc., where as in linear probing probes bucket at locations $l+1$, $l+2$, $l+3$, ... etc.

* In general this method probes buckets at location $(l+i^2)$ mod n.

where $i = 1, 2, \ldots$

l = location of home buckets

n = number of buckets.

* Quadratic probing may also results in probing the same set of alternative cells and this is known as secondary clustering, which occurs when hash table size is not print.

Double hashing (or) Rehashing:-

* Double hashing is a technique used in hash table to resolve hash collisions. It uses one hash value as a starting value and continual evolution of an interval until the desired value is found, or an empty location is found or the entire table is searched.

* The interval is decided using another independent hash function. so, it is named as double hashing. Here the interval depends

on the data so that even values mapping the same location will have different sequences that minimize the repeated collisions and the effect of clustering.

* For a given independent hash functions h_1 and h_2, the ith location in the bucket sequence for value 'x', in a hash table of size 'n' is given as.

$$h(x,i) = h_1(x) + i * h_2(x) \bmod n.$$

* Consider the same example as in linear probing.. Suppose, let hash function h_2 be defined as,

$$h_2(x) = 1 + (x \bmod 6)$$

* Adding '1' to h_2 ensures that the increment is not equal to 0. The values 16, 18, 9 are inserted.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
						16		18	9

* Now insert 36, h(36) = 36 mod 10 = 6 : collision occurs because the state of location 6 is OCCUPIED. So find the next free location using double hashing. The next location is determined by $h_2(x)$ function.

i.e., $h_2(36) = 1 + (36 \bmod 6) = 1$. Adding this to the current location leads to probe location 7, which is in an EMPTY state, so 36 can be inserted into 7th location.

[0]	[1]	[2]	[3]	[4]	[5]	[6]	[7]	[8]	[9]
		26				16	36	18	9

Extendible hashing:

* Extendible hashing makes hashing dynamic. With this technique, access to data from the buckets is done indirectly through an index which is adjusted dynamically in order to reflect the changes in a table.

* The main feature of extendible is to organize the index which handles overflow with an expandable table.

* When ever a hash function is applied for a key, the value returned by its gives the position in the index, and these values are known as pseudo keys. Using extendible hashing there is no need to reoorganize the table when insertion and deletion take place.

* A single hash function 'h' is used but depending on the size of the index a portion of address of h(K) is used. The address of h(K) is represented as a string of bits and only the 'i' left-most bits are used. Here 'i' is called the depth of the directory.

* Consider an example where the hash function 'h' returns patterns of five bits. Suppose a pattern is a string 01101 & the depth is two, then the left-most bits 01 are used to represent the location in the directory which contains the pointer to a bucket where the required key can be 's' found (or) the one where the required key can be inserted.

* In the following figure the values of hash functions 'h' as shown in buckets, and these values represent the keys that are actually stored in the buckets.

Extendible Hashing:

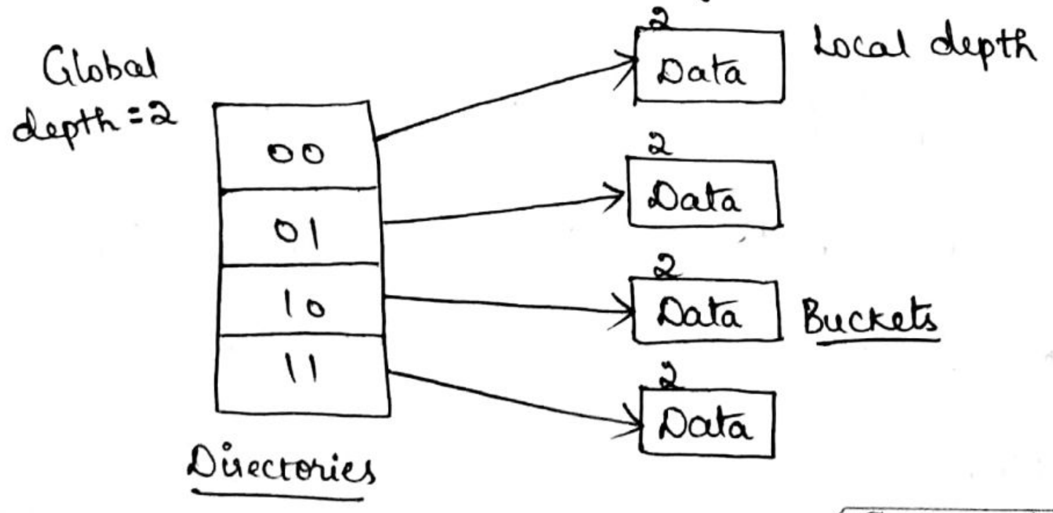
A dynamic hashing methods wherein directories and buckets are used to hash data. It is a flexible method in which hash function also experiences dynamic changes.

Features of Extendible Hashing:

Directories → Stores addresses of the buckets in pointers. An id is given/assigned to each directory which may change each time when directory Expansion takes place.

Buckets → The buckets are used to hash the actual data. It may contains more than one pointer to it if local depth < global depth.

Basic Structure of extendible Hashing:



No. of directories = 2 ^ Global depth.

Global depth → It is associated with directories. They denote number of bits used by hash function to categorize keys.

Global depth = No. of bits in directory id.

Local depth → Local depth is associated with buckets. It is used to decide the action that to be performed in case an overflow occurs.

local depth <= Global depth.

Bucket Splitting → when the number of elements in a bucket exceeds a particular size, then bucket is split into two parts.

Directory Expansion → It takes place when a bucket overflows. It is performed when the local depth of the overflowing bucket is equal to the global depth.

Basic Working :-

Data (Converted into Binary)

↓

Hash Function (returns directory location)

↓

Directory (Points to a Bucket)

↓ Data traverses to the bucket.

Bucket (Data is Stored | Hashed)

Procedure :-

* Analyze dara elements that may exist in various forms such as Integer, String, float., etc..

* Convert the data element into binary form. If the data element is string, then consider ASCII equivalent integer of the Starting character and Convert it into binary form.

* Check the Global depth of the directory.

* Identify the directory.

* Then, navigate to the bucket pointed by the directory with ~~[illegible]~~ directory-id.

* Insert the element and check if the overflow occurs. If the overflow is not encountered, then the element is successfully hashed.

* Otherwise, Tackle over flow condition during data insertion.

Case 1: If local depth = global depth, then directory expansion as well as bucket split needs to be performed. Then increment the global depth and local depth by '1', and assign appropriate pointers. Directory expansion will double the number of directories.

case 2: If local depth < global depth, then only bucket split takes place. Increment only local depth value by 1 and assign appropriate pointers.

* Finally, the elements present in the overflowing bucket are rehashed with respect to the new global depth of the directory.

Example :

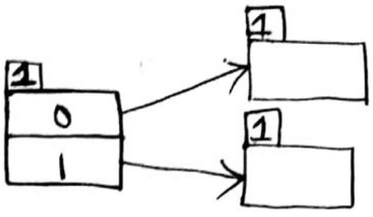
16, 4, 6, 22, 24, 10, 31, 7, 9, 20, 26.

Bucket Size : 3 (Assume)

* Calculate the binary forms,

16 – 10000	24 – 11000	9 – 01001
4 – 00100	10 – 01010	20 – 10100
6 – 00110	31 – 11111	26 – 01101
22 – 10110	7 – 00111	

Initially, the global depth and local depth is always 1.

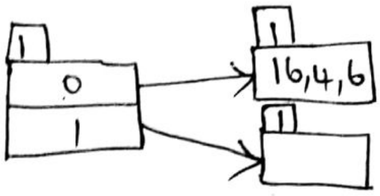


Insert – 16, 4, 6

16 – 1000$\underline{0}$

4 – 0010$\underline{0}$

6 – 0011$\underline{0}$



Insert – 22

22 – 1011$\underline{0}$

The bucket is already full. Hence, overflow occurs. Here, Local depth = Global depth. So, both the bucket splits and directory expansion takes place. Rehashing of numbers also takes place after split.

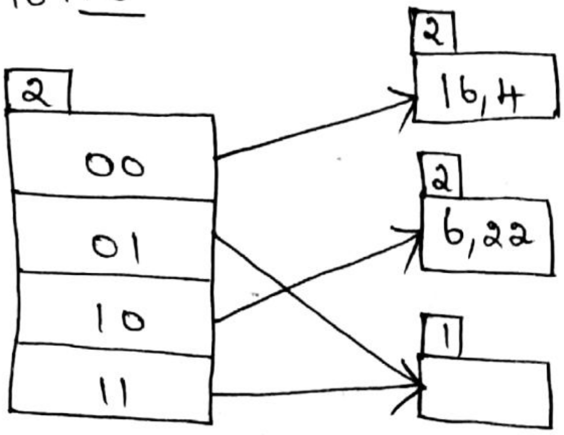
So,

16- 100$\underline{00}$

4- 001$\underline{00}$

6- 001$\underline{10}$

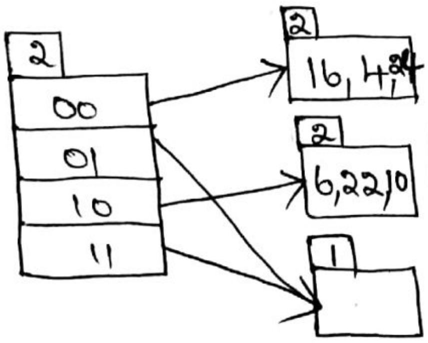
22- 101$\underline{10}$



Insert 24 and 10:

24-110$\underline{00}$

10-010$\underline{10}$

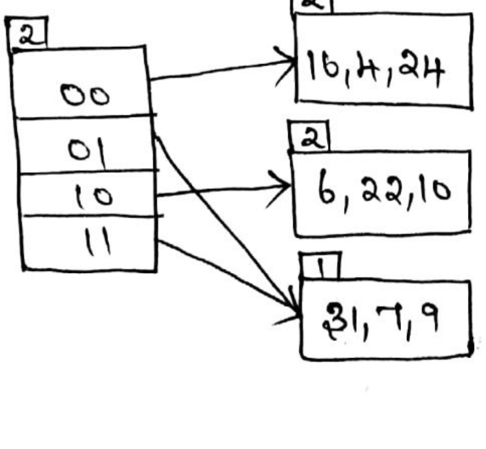


Insert 31, 7, 9:

31-111$\underline{11}$

7-001$\underline{11}$

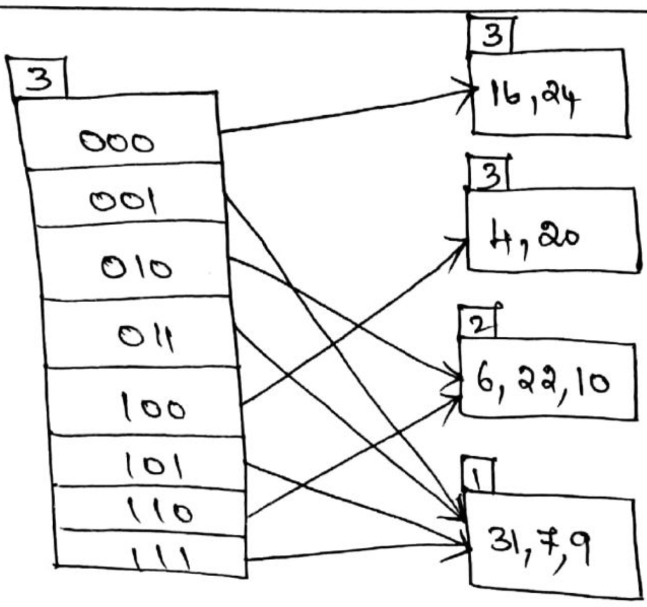
9-010$\underline{01}$



Insert 20:

20 - 101$\underline{00}$ → Cause overflow.

Local depth = Global depth.



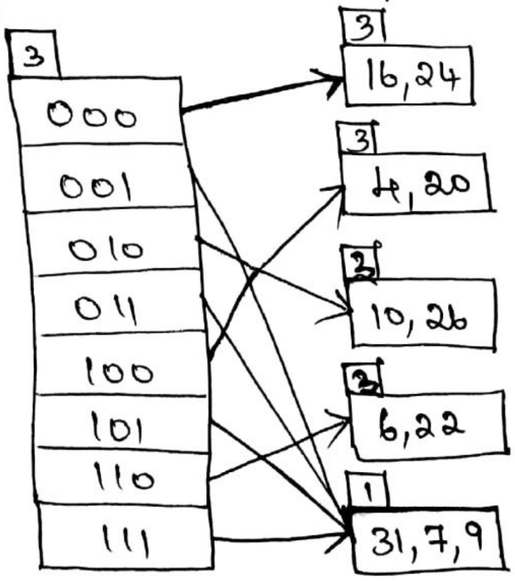
16	10000
4	00100
6	00110
22	10110
24	11000
10	01010
31	11111
7	00111
9	01001
20	10100

Insert 26:

26 - 11010 overflow occurs.

local depth < Global depth.

So, only the bucket split will happen.



6	00110
22	10110
10	01010

Hashing is completed.