UNIT-1

Dictionaries : sets, Dictionaries, hash tables, open hashing, closed hashing (Rehashing methods), Hashing functions (Division, multiplication, universal hashing), skip lists, Analysis of skip lists.

Sets :-

A set is a collection of well defined elements.

well defined means it must be absolutely clear that which object belongs to the set & which does not.

Ex: The collection of positive numbers less than 10 is a set because given any numbers we can always find out whether that number belongs to the collection or not. But the collection of good students in your class is not a set as in the case no definite rule is supplied by the help of which you can determine whether a particular student of your class is good or not.

2. A group of singers with ages blw 18 years & 25 years is a set because the range of ages of the singer is given & so it can easily be decided that which singer is to be included & which is to be excluded

3. A collection of red flowers is a set

4. Collection of past presidents of the united states union is a set.

5. A group of young dancers is not a set as the range of the ages of young dancers is not given & so it can't be decided that which dancer is to be considered young

properties of set :-

→ Set is defined by capital letters.
→ All the elements in the sets are enclosed within { } (curly braces)
→ Every Element is seperated by comma

Eg: $A = \{a, b, c, d\}$

here $a \in A$

$c \notin A$

Representation of sets :-

1) Statement form (Descriptive form / Describe method)

In this form the well defined descriptions of a member of a set are written and enclosed in curly braciels.

for example the set of Even numbers less than 15

In state form it can be written as

{ even numbers less than 15 }

Roster form :- (Tabular form / listing method)

In this form all the elements of a set are listed.

For Example set of natural numbers less than 5

It can be written as $N = \{1, 2, 3, 4\}$

Set Builder form :-

In this representation of a set, the element of the set is described by using a symbol x or any one variable followed by colon (:) or '|' is used to denote such that then we write the property possessed by the elements of the set & enclose the whole description in braces

Ex: $A = \{2, 4, 6, 8\}$

$A = \{x : x = 2n, n \in N \text{ & } 1 \leq n \leq 4\}$

read as A is the set of elements x such that $x = 2n$ & $1 \leq n \leq 4$ }

Types of sets :-

Empty set : A set which does not contain any element. It is denoted by $\{\}$ or ϕ

Singleton set :-

A set which contains a single element

Ex: There is only one apple in basket

Finite set :-

A set which consists of a definite no. of elements

Ex: A set of natural numbers upto 5

$A = \{1, 2, 3, 4, 5\}$

Infinite set :-

A set which is not finite

Ex: A set of all natural numbers.

$A = \{1, 2, 3, 4, \ldots\}$

Equal sets --

Two sets A & B can be equal only if each element of set A is also the element of the set B. Also if two sets are subsets of each other they are said to be equal

$A = B$

$A \subset B$ & $B \subset A \Leftrightarrow A = B$

Equivalent sets :-

two sets are equivalent if the no. of elements in both the sets is equal and it is not necessary that they have same elements or they are subset of each other

$P = \{1, 3, 9, 5, -7\}$ $Q = \{5, -7, 3, 1, 9\}$

Here P & Q are Equal sets.

$P = \{1, -7, 10\}$ $Q = \{A, B, C\}$

P & Q are Equivalent sets.

Disjoint sets :-

Two sets A & B are said to be disjoint if the set does not contain any common elements

Ex $A = \{1, 2, 3, 4\}$ $B = \{5, 6, 7\}$

Subsets :-

A set A is said to be subset of B if every element of A is also an element of B, denoted as $A \subseteq B$, even null set is considered to be subset of onomer set.

Ex: $A = \{1, 2, 3\}$

then $\{1, 2\} \subseteq A$

similarly omer subsets of A are $\{1\}, \{2\}, \{3\}, \{1, 2\}, \{2, 3\}, \{1, 3\}, \{1, 2, 3\}, \{\}$

If A is not a subset of B then it is denoted as $A \not\subseteq B$

proper Subset :-

If $A \subseteq B$ & $A \neq B$ then A is called the proper subset of B & can be written as $A \subset B$

Ex if $A = \{2, 5, 7\}$ is a subset of $B = \{2, 5, 7\}$ then it is not a proper subset of B.

But $A = \{2, 5\}$ is a subset of $B = \{2, 5, 7\}$ and is a proper subset also.

Superset :-

Set A is said to be superset of B if all the elements of set B are elements of set A it is represented as $A \supset B$

Ex $A = \{1,2,3,4\}$ & set $B = \{1,3,4\}$ then A is superset of B.

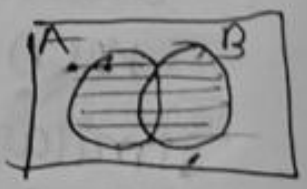
operations on sets :-

union of sets :-

If set A & B are two sets then $A \cup B$ is set contains all elements of set A and set B

$A = \{1,2,3\}$ $B = \{4,5,6\}$

$A \cup B = \{1,2,3,4,5,6\}$



Intersection of sets :-

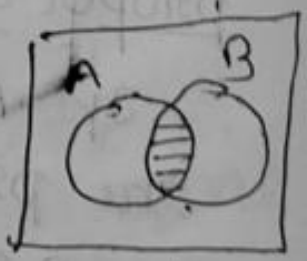
If A & B all two sets then $A \cap B$ is the set that contains only common elements b/w A & set B

1) $A = \{1,2,3\}$ $B = \{3,4,5\}$

$A \cap B = \{3\}$

2) $A = \{1,2,3\}$ $B = \{4,5,6\}$

$A \cap B = \{\}$ or $\emptyset$



compliment of sets :-

compliment of any set P, is in set of all elements in universal set that are not in set P. It is denoted by P'

$P' = U - P$

the final key-value pair. so if we insert duplicate keys then our original data is lost.

3. The key attribute is case sensitive hence key is not the same as KEY

4. The key attribute is immutable hence we cannot have an array or list as keys we can only have numbers, strings etc as the key.

operations in dictionary :-

Add or Insert :-

In this operation a new pair of keys and values is added in the dictionary or associative array object.

Replace or reassign :-

In the replace or reassign operation the already existing value that is associated with a key is changed or modified. In other words a new value is mapped to an already existing key.

Delete or remove :-

In the delete or remove operation the already present element is unmapped from the dictionary or associative array object.

the final key-value pair. so if we insert duplicate keys then our original data is lost.

3. The key attribute is case sensitive hence key is not the same as KEY

4. The key attribute is immutable hence we cannot have an array or list as keys we can only have numbers, strings, etc as the key.

operations in dictionary :-

Add or Insert :-

In this operation a new pair of keys and values is added in the dictionary or associative array object.

Replace or reassign :-

In the replace or reassign operation the already existing value that is associated with a key is changed or modified. In other words a new value is mapped to an already existing key.

Delete or remove :-

In the delete or remove operation the already present element is unmapped from the dictionary or associative array object.

Find or lookup:

In the find operation the value associated with a key is searched by passing the key as a search argument.

operation	dictionary	output
insertItem(5,A)	{(5,A)}	
insertItem(7,B)	{(5,A),(7,B)}	
insertItem(2,C)	{(5,A),(7,B),(2,C)}	
insertItem(8,D)	{(5,A),(7,B),(2,C),(8,D)}	
insertItem(2,E)	{(5,A),(7,B),(2,E),(8,D)}	
findItem(7)		B
findItem(4)		No such key
removeItem(5)	{(7,B),(2,E),(8,D)}	A
replaceItem(7,M)	{(7,M),(2,E),(8,D)}	

Hash tables:-

hash table is one of the most important data structure that uses a special function known as hash function that maps a given value with a key to access the elements faster.

A hash table is DS that stores some information & the information has basically two main components i.e key & value. The hash table efficiency of mapping depends upon the efficiency of the hash function used for mapping

Drawback of hash function :-

A hash function assigns each value with a unique key. Sometimes hash table uses an imperfect hash function that causes a collision because the hash function generates the same key for two different values.

Hashing :-

hashing is one of the searching techniques that uses a constant time. The time complexity in hashing is O(1) while known linear search worst time complexity is O(n) & binary search worst case time complexity is O(log n). In both searching techniques the searching depends upon the no. of element but we want the technique that takes a constant time. So hashing technique came that provides a constant time.

The main idea behind hashing is to create the (key/value) pairs. If the key is given, then the algorithm computes the index at which the value would be stored. It can be written as

Index = hash(key)

There are three ways of calculating hash function

1) Division method
2) Folding method
3) Mid square method.
4) Multiplication method

hashing functions

Division method :-

In this method the hash function can be defined as

$$h(K) = K \% m$$

where m is size of hash table

Generally M is chosen to prime number.

Example

K = 88

M = 11

$h(88) = 88 \% 11$

$= 0$

K = 78

M = 11

$h(78) = 78 \% 11$

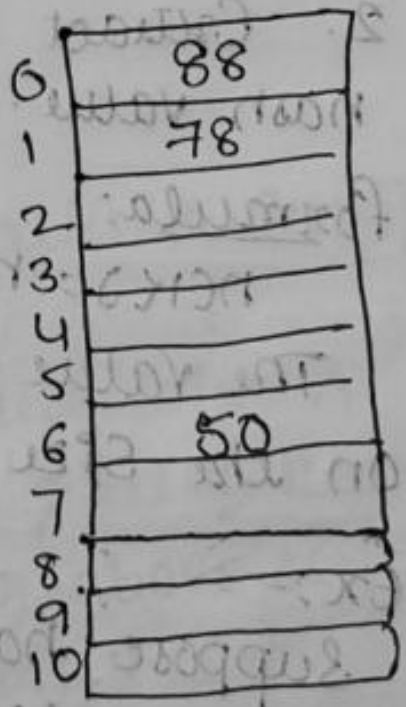
$= 1$

K = 50

M = 11

$h(50) = 50 \% 11$

$= 6$



pros:-

→ This method is quite good for any value of M

→ The division method is very fast since it requires only a single division operation

cons:-

Sometimes extra care should be taken to choose the value of M

Mid Square Method:-

It is very good hashing method. It involves two steps to compute hash value

1. Square the value of the key K. ie K^2
2. Extract the middle r digits as the hash value.

formula:

$h(K) = h(K \times K)$

The value of r can be decided based on the size of the table.

Ex:-

Suppose hash table, has 100 memory locations so r = 2 because two digits are required to map the key to the memory location.

$K = 60$

$K \times K = 60 \times 60$

$= 3600$

$h(60) = 60$

Digit Folding method :-

This method involves two steps

1. Divide the key-value k in to a no. of parts i.e k1, k2, k3, ... kn where each part has the same no. of digits except for the last part may can have lesser digits than other parts
2. Add the individual parts. The hash value is obtained by ignoring the last carry if any.

Ex: 12345

k1 = 12, k2 = 34, k3 = 5

s = k1 + k2 + k3

= 12 + 34 + 5 = 51

h(k) = 51

Multiplication method :-

1. Choose a constant value A such that $0 < A < 1$
2. Multiply the key value with A
3. Extract the fractional part of kA
4. Multiply the result of above step by the size of the hash table i.e M
5. The resulting hash value is obtained by taking the floor of result obtained in step 4

Formula: $h(k) = floor(M(kA \bmod 1))$

M = size

k = key

A = constant = 0.618033

Donald Knuth suggested

k = 23

M = 10

$h(23) = floor(10 * (23 \times 0.618033)) \% 10$

$= floor(142.143) \% 10$

$= 142 \% 10$

$= 2$

$h(23) = floor(10(23 \times 0.618033 \bmod 1))$

$floor(10(14.2147 \% 1))$

$floor(10(0.2147))$

$floor(10 \times 0.2147)$

$floor(2.147) = 2$

UNIVERSAL HASHING → last page of UNIT-I

(open Hashing :- (closed Addressing))

collision Handling techniques

If two pieces of data share the same value in a hash table then it is known as collision in hashing. As the key of hash function these collisions are handled using different techniques known as collision handling techniques. These techniques are also known as collision

Resolution techniques:

The techniques that are used to handle or resolve the collisions are basically classified in to two types they are:

1) open hasing (or) seperate chaining (or) closed Addressing chaining.
2) closed hashing (or) open Addressing

Open Hashing:-

The first collision resolution or handling technique, "open hashing", is popularly known as seperate chaining. This is a technique which is used to implement an array as a linked list known as a chain. It is one of the most used techniques by progeammers to handle collisions. Basically a linked list data structure is used to impliment the seperate chaining technique.

when a number of elements are hashed in to the index of single slot Then they are inserted in to a singly linked list. This singly linked list is the linked list which we refer to as a chain in the open hashing technique.

Advantages of open hashing :

→ This method is simple to implement & understand

→ It can be implemented when we do not know how frequently keys will be inserted or deleted.

→ open hashing is less sensitive to the load factors or hash functions

Disadvantages :

→ The cache performance of seperate chaining method is poor as the keys are stored using a singly linked list.

→ A lot of storage space is wasted as some parts of the hash table are never used

→ In the worst case the search time can become O(n) this happens only if the chain becomes too lengthy.

→ Extra storage is used for storing links of the chain.

Functions used in open hashing :-

1) get(k, keys) :- If the key is present in hash table, get(k key) gives the value corresponding to the key (hash table)

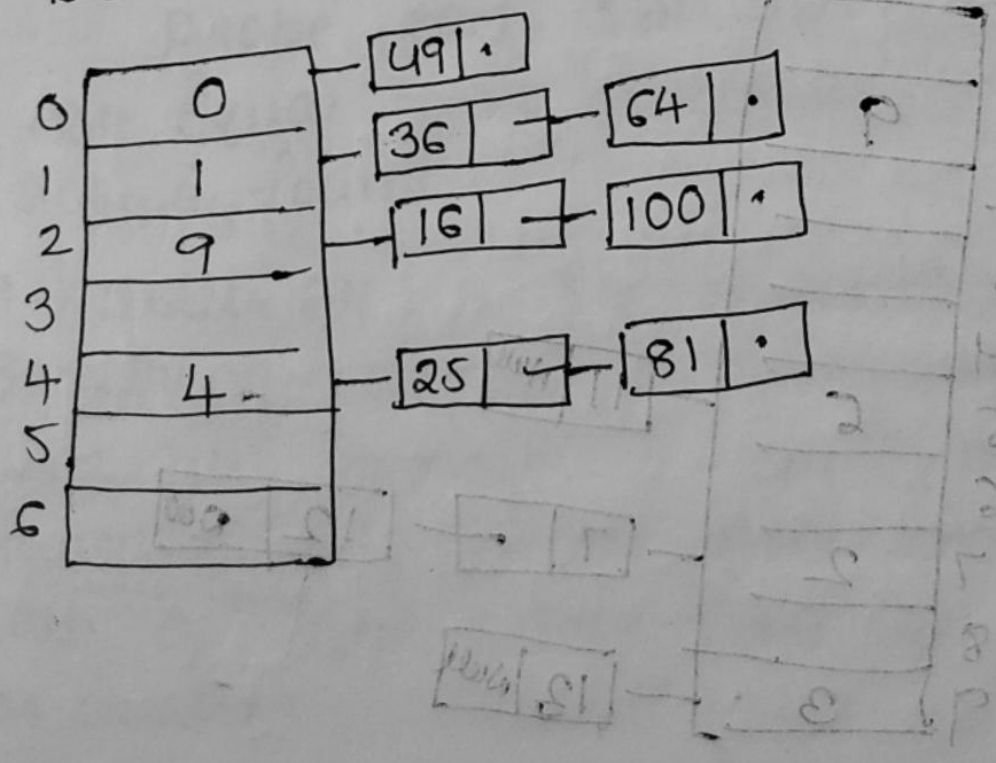
2. getSize() : It will return the size of hash taby.

3. add() :-
changes an existing valid key-value pair in hash table if it already exists before adding a new on.

4. Remove() :-
The remove function will remove both the key & value pair

5. isEmpty() :-
it returns the value "True", if the size is zero.

Example :-
Consider the keys 0, 1, 4, 9, 16, 25, 36, 49, 64, 81, 100 let the hash function be $h(x) = x \% 7$

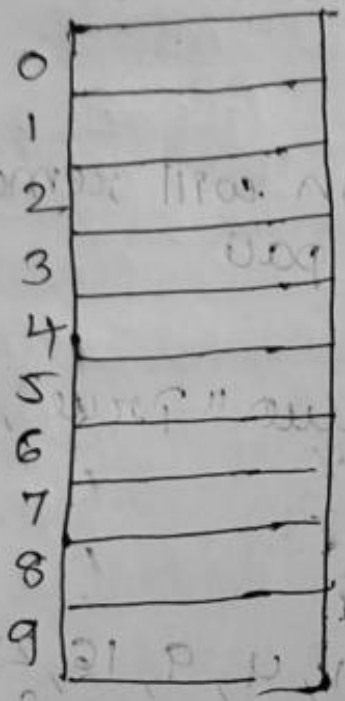


Example: 2.

A. 3, 2, 9, 6, 11, 13, 7, 12

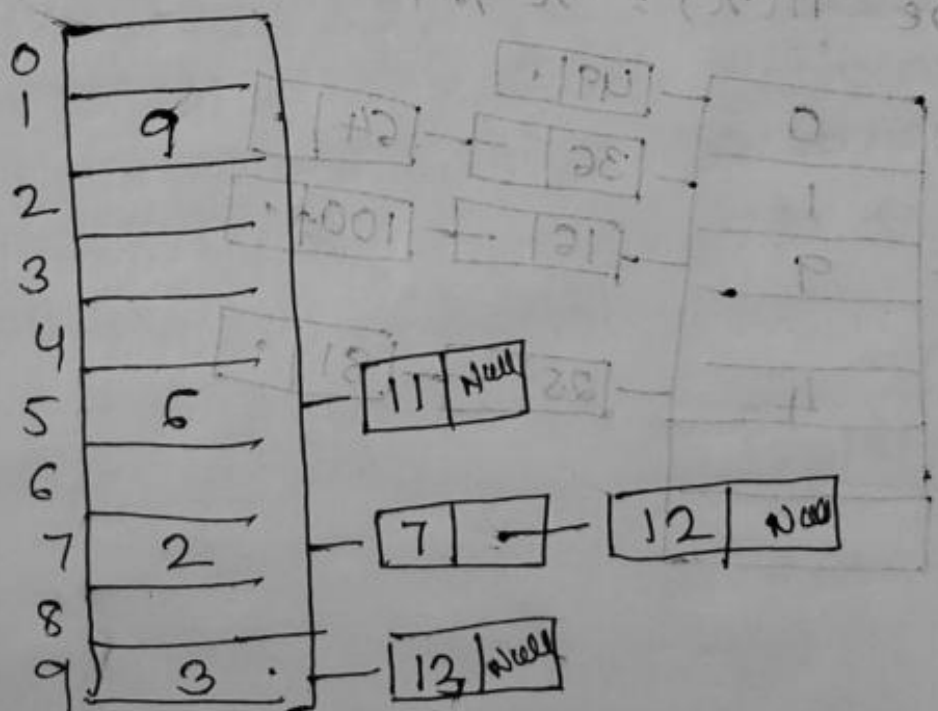
$h(k) = 2k+3$. $m = 10$

use division method



key	location
3	$[2(3)+3] \% 10 = 9$
2	$[(2\times2)+3] \% 10 = 7$
9	$[(2\times9)+3] \% 10 = 1$
6	$[(2\times6)+3] \% 10 = 5$
11	$[(2\times11)+3] \% 10 = 5$
13	$[(2\times13)+3] \% 10 = 9$
7	$[(2\times7)+3] \% 10 = 7$
12	$[(2\times12)+3] \% 10 = 7$

Result:



closed hashing or open Addressing :-

The second most collision resolution technique, closed hashing is a way of dealing with collisions, similar to seperate chaining process. In open Addressing, the hash table alone stores all of its elements. the size of the table should always be greater than or equal to the total no. of keys at all times.

Functions used in closed hashing :-

1. Insert (K) :-
   keep probing. place the key "K" in the first Empty slot you find

2. Search (K) :-
   probe each slot until the key is not equal to K or until an Empty slot is found.

3. Delete (K) :- Its interesting to delete something. the Search may fail when we just remove a key & then perform Search operation. the slots that are a part of deleted key slots are considered as deleted.

Several techniques to perform implementation of closed hashing :-

1) linear probing :- probing - Searching for empty pl

In linear probing, the hash table undergoes clear & neat examination, starting from the hash's initial a beginning point of the slot that is obtained after the calculation is already occupied then we should look for different one. the function that is responsible for performing rehashing is "key = rehash (n+1) % table-size

Difficulties faced with linear probing :-

→ primary clustering :-

It is the one of the major issues that are caused with the linear probing technique. Many elements that are consecutive to each other generaly form clusters or a group of scattering which in turn makes the hash table more difficult to find an Empty slot or search for an element.

→ Secondary clustering :-

Secondary clustering is not as severe as primary clustering & the elements or records that must be placed within the same location are only allowed to share a collision chain which is also known as probe sequence, if they begin at the same location.

→ clustering is the only problem in linear probing. If clustering can be reduced within this mechanism, then the can be considered one of the best collision resolution technique

2. ~~Quadratic probing~~ :-

The function used for rehashing is as follows:

$$\text{rehash(key)} = (n+1) \% \text{table-size}$$

Ex: let us consider a simple hash function as "key mod 7" & sequence of keys as 50, 700, 76, 85, 92, 73, 101

Sol:

key 50

h(x) = x % 7

50 % 7

= 1

50 is inserted at Index 1

key 700

h(x) = x % 7

700 % 7

= 0

700 is inserted at Index 0

key 76

$h(x) = x \% 7$

$= 76 \% 7$

$= 6$

76 is inserted at index 6

key = 85

$h(x) = x \% 7$

$= 85 \% 7$

$= 1$

collision occurs.

0	700
1	50
2	85
3	92
4	73
5	101
6	76

$rehash(x) = (hash(x) + 1) \% 7$

$= (1 + 1) \% 7$

$= 2$

key = 92

$h(x) = x \% 7$

$= 92 \% 7$

$= 1$

collision occurs

$rehash(x) = (h(x) + 1) \% 7$

$= (1 + 1) \% 7 = 2$

collision

$rehash(x) = (h(x) + 2) \% 7$

$= (1 + 2) \% 7$

$= 3$

92 is inserted at index 3

key = 73

h(x) = x % 7

= 73 % 7

= 3

collision

rehash(x) = (h(x) + 1) % 7

= (3+1) % 7 = 4.

Insert 73 at index 4

key = 101

h(x) = x % 7

= 101 % 7

= 3 (collision)

rehash(x) = (h(x)+1) % 7

= 4 % 7 ⇒ 4 (collision)

rehash(x) = (h(x) + 2) % 7

= (3+2) % 7

= 5

Insert 101 at index 5

2. Quadratic probing

Quadratic probing is an open-addressing scheme where we look for the i^2'th slot in the ith iteration if the given hash value x collides in the hash table.

Let hash(x) be the slot index computed using hash function

- If the slot hash(x) % S is full, then we try (hash(x) + 1*1) % S
- If (hash(x) + 1*1) % S is also full, then we try (hash(x) + 2*2) % S
- If (hash(x) + 2*2) % S is also full Then we try (hash(x) + 3*3) % S
- This process is repeated for all the values of i until an empty slot is found.

Ex :-

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

→ K = 50

h(x) = K % 7

= 50 % 7

= 1

→ K = 700

h(x) = 700 % 7

Index = 0

→ K = 76

h(x) = 76 % 7

= 6

→ K = 85

h(x) = 85 % 7

= 1 (Collision)

0	700
1	50
2	85
3	73
4	101
5	92
6	76

12/17
84

(hash(x) + 1*1) %. 7

= (1 + 1) %. 7

= 2

→ K = 92

h(x) = 92 %. 7

= 1 (collision)

(hash(x) + 1*1) %. 7

(1 + 1) %. 7

= 2 (collision)

(hash(x) + 2*2) %. 7

(1 + 4) %. 7

= 5

92 is inserted at index 5

→ K = 73

h(x) = 73 %. 7

= 3

73 is inserted at index 3

→ K = 101

h(x) = 101 %. 7

= 3 (collision)

(hash(x) + 1*1) %. 7

(3 + 1) %. 7

= 4

101 is inserted at index 4

3. Double hashing :-

Double hashing is works on a similar idea to linear & quadratic probing. use a big table & hash into it. whenever a collision occurs, choose another spot in table to put the value. The difference here is that instead of choosing next opening, a second hash function is used to determine the location of the next spot. for example given hash function H1 & H2 & key

→ check location hash1(key). If it is Empty put record in it

→ If it is not Empty calculate hash2(key)

→ check if (hash1(key) + hash2(key))%H is open, if it is, put it in

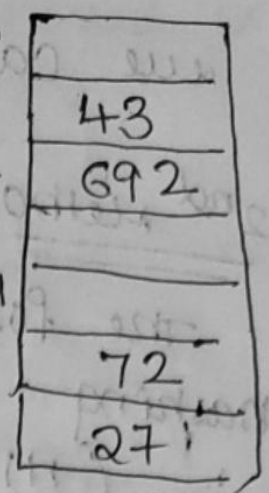
→ repeat with hash1(key) + 2 hash2(key), hash1(key) + 3 hash2(key) & so on until an opening is found.

Example :-

Insert the keys 27, 43, 692, 72 into hash table of size 7 where first hash function is h1(k) = k mod 7 & second hash-function is h2(k) = 1+(k mod 5)

step 1 : Insert 27

27 % 7 = 6, location 6 is Empty so insert 27 into 6 slot



step 2 : Insert 43

43 % 7 = 1, location 1 is Empty so insert 43 in to 1 slot.

step 3 : Insert 692

692 % 7 = 6, location 6 is already being occupied this is a collision so we need to resolve this collision using double hashing

hnew = [h1(692) + i * (h2(692)] % 7

= [6 + 1 * (1 + 692 % 5)] % 7

= 9 % 7

= 2

Now 2 is Empty slot

so we can insert 692 in to 2nd slot

step 4 : Insert 72

72 % 7 = 2, but location 2 is already being occupied this is a collision

hnew = [h1(72) + i * (h2(72)] % 7

= [2 + 1 * (1 + 72 % 5)] % 7

= 5 % 7

= 5

we can insert 72 in to 5th slot

2nd Method :-

The following function denotes double hashing

(H1(K) + i * H2(K)) % table size

where H1(K) & H2(K) are 2 hash functions & table size is size of hash table

A typical first hash function is given by:

H1(K) = K % table size.

A typical second hash function is given by:

H2(K) = CP - (K % CP)

where CP is a no. that must be smaller than the size of the hash table & should be coprime with size of table.

Ex:-

Let us consider keys. 20, 34, 45, 70, 56

Let size of table be 11 so the first hash function will be given by:

H1(K) = K % 11

For the second hash function, CP should be chosen in such a way HCF(CP, 11) = 1

So let CP = 8

second hash function ep should be chosen $H2(k) = 8 - (k\%8)$

key = 20

$h1(20) = 20\%11 = 9$ (No collision)

20 is inserted at 9

→ key = 34

$h1(34) = 34\%11 = 1$

34 is inserted at index 1

→ key = 45

$h1(45) = 45\%11 = 1$ (collision)

$h2(45) = 8 - (45\%8) = 3$

$(1 + 1*3)\%11 = 4$

45 is inserted at index 4

→ key = 70

$h1(70) = 70\%11 = 4$

$H2(70) = 8 - (70\%8) = 2$

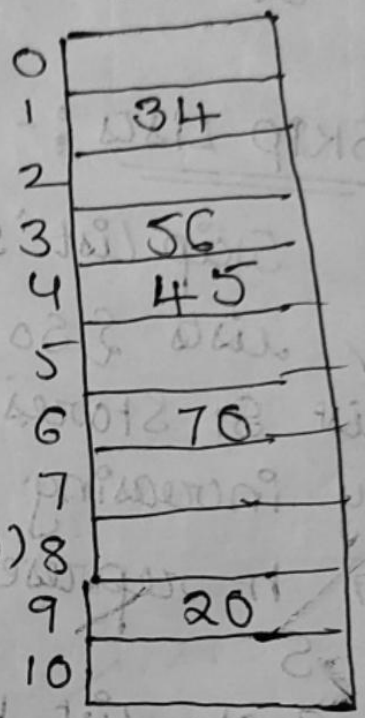
$(4 + 1*2)\%11 = 6$

70 is inserted at index 6

→ key = 56

$H1(56) = 56\%11 = 1$ (collision)

$H2(56) = 8 - (56\%8) = 8$



(1+1*8)%11 = 9 (collision)

(1+2*8)%11 = 6 (collision)

(1+3*8)%11 = 3

56 is inserted at index 3.

SKIP LISTS :-

Skip list 'S' consists of a series of lists $\{S_0, S_1, S_2 \cdots S_n\}$ such that each list S_i stores a subset of the items sorted by increasing key

→ h represents the height of the skip list S.

→ Each list has two sentinel nodes $+\infty$ & $-\infty$. $-\infty$ is smaller than any possible key in the list & $+\infty$ is greater than any possible key in the list. Hence the node with $-\infty$ key is on left most position & node with $+\infty$ key is always the right most node in the list.

→ For visualization, it is customary to have the S_0 list on the bottom & lists $S_1, S_2, \cdots S_h$ above it.

→ Each node in a list has to be "sitting on another node with same key below it meaning that if L_{i+1} has a node

with key K then L_i, L_{i+1} ... all valid lists below it will have same key in them
→ skip list also needs to maintain the "head" pointer i.e the reference to the first member (sentinel node -ve) in the topmost list.
→ on average the skip list will have O(n) space complexity.

Ideal Skip list :-

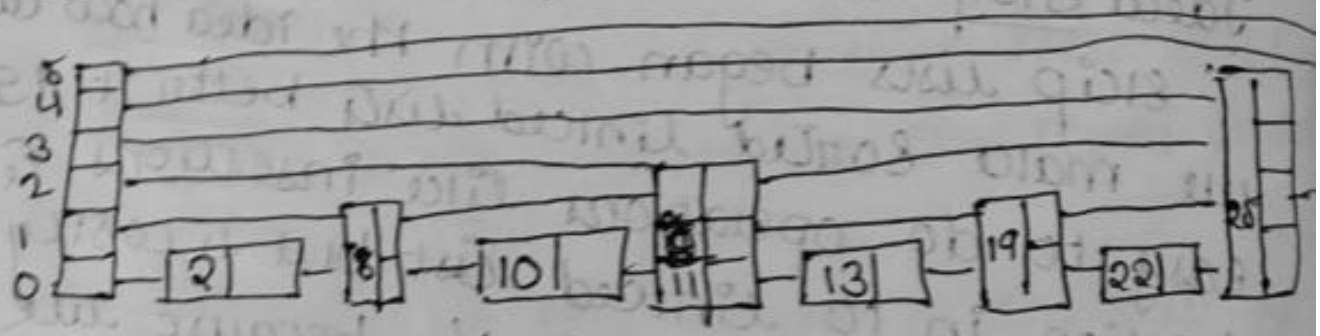
Skip lists began with the idea how can we make sorted linked lists better? It is easy to do operations like insertions & deletion in to linked list but is costly to locate items efficiently because we have to walk through the list one item at a time. If we could over multiple items at a time however then we could perform searches efficiently.

Intuively, a skip list is a data structure that encodes a collection of sorted linked lists, where linked skip over 2, then 4, then 8, & so on, elements with each link.

To make this more concrete, imagine a linked list, sorted by key value. There are two nodes at either end of the list - called head & tail! take every other entry of this linked list & extend it up to a new linked list & extend it.

with 1/2 as many entries. Now take every other entry of this linked list & extend it up to another linked with 1/4 as many entries as orginal list & continue. This until no elements remain. The head & tail nodes are always stored clearly we can repeat this process ⌈log n⌉ times.

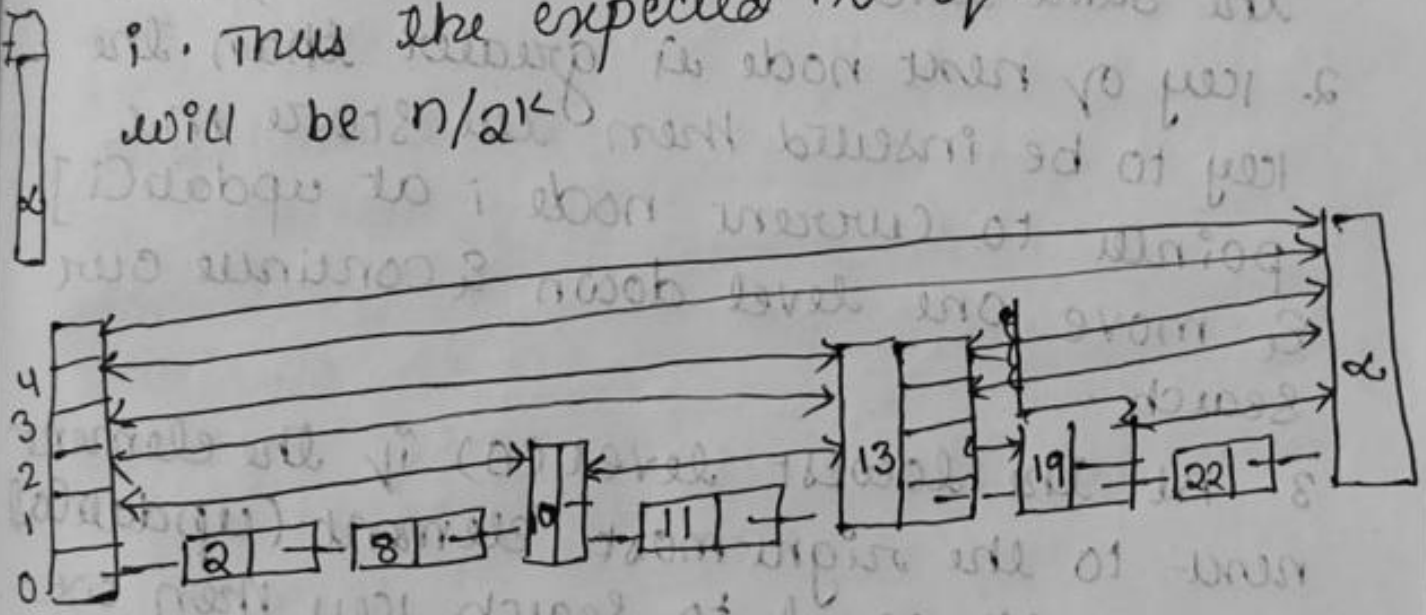
The result is something that we call the ideal skip list



Randomized Skip list :-

Unfortunately like a perfectly balanced binary tree, the ideal skip list is too rare to be able to use for a dynamic data structure. As soon as a single node was added to the middle of the list all the heights following it would need to be modified. but we can relax this requirement to achieve an efficient data structure. In the ideal skip list every other node from level i is extended up to level i+1. Instead how about we do it randomly

Suppose that we have built a skip list up to some level i, & we want to extend this to level i+1. Imagine a node at level i tossing a coin. If the coin comes up heads (with probability 1/2) this node promotes itself to level i+1, & otherwise it stops here. By the nature of randomization the expected no. of nodes at level i+1 will be half the no. of nodes at level i. Thus the expected no. of nodes at level K will be $n/2^K$



operations on skip list :-

1. Insertion operation :-
   To insert any element in a list.
2. Seach operation :-
   To seach any element in a list.
3. Deletion operation :-
   To delete any element from a list.

Searching Operation

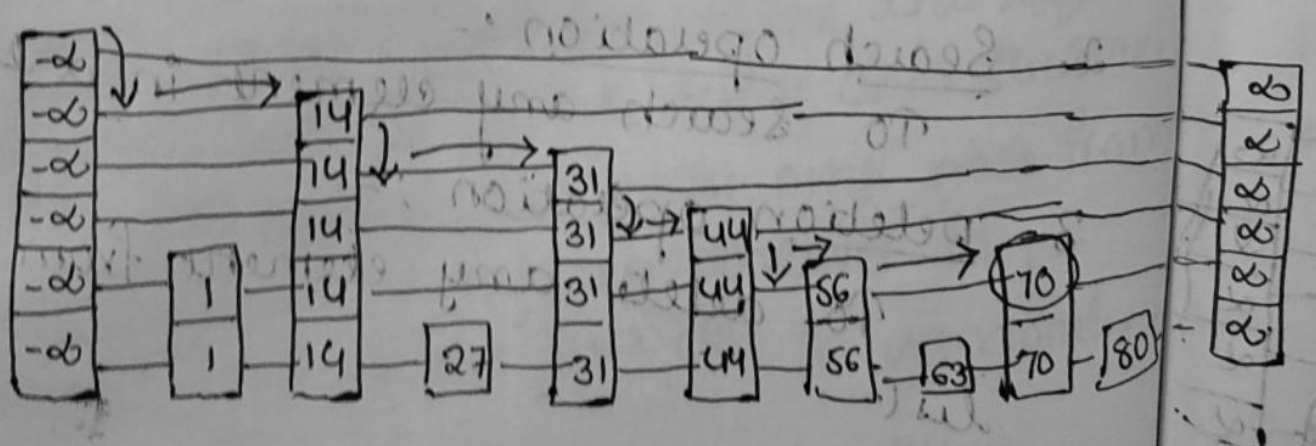
Searching an element is very similar to approach for searching a spot for inserting an element in skip list

Algorithm

1. If key of next node is less than search key then we keep on moving forward on the same level
2. key of next node is greater than the key to be inserted then we store the pointer to current node i at update[i] & move one level down & continue our search.
3. At the lowest level (0) if the element next to the right-most element (update[0]) has key equal to search key then we have found key otherwise failure.

Example :-

finding 70



we will simply start with the left most node in the list on top. Go down if the next value in the same list is greater than the one we are looking for, go right otherwise.

Inserting in Skip list :-

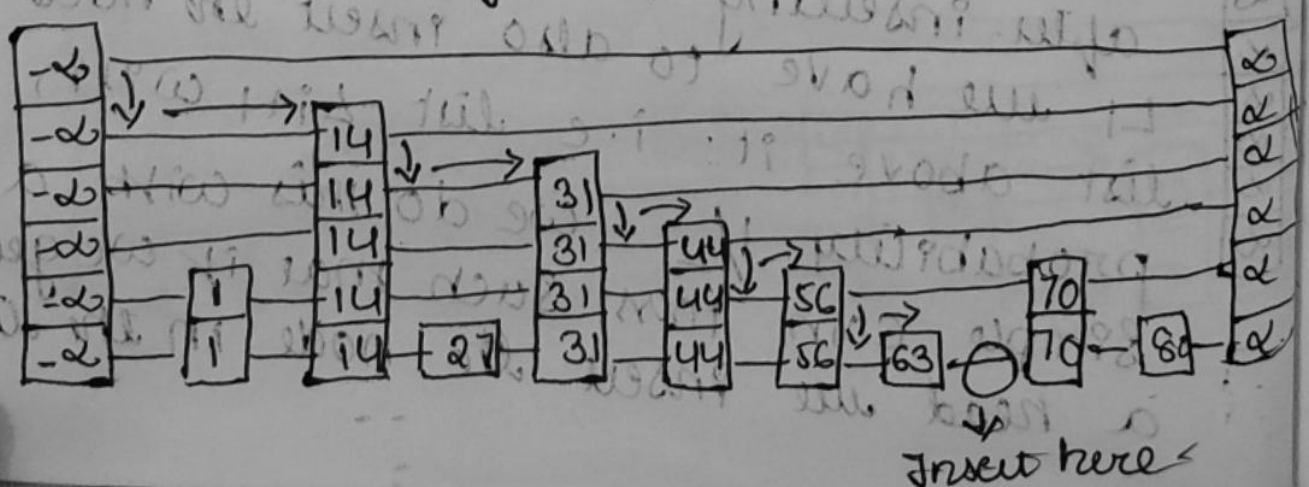
To insert a new node we would first find the location where this new key should be inserted in the list at the very bottom. this can be simply done be reusing the search logic, we start traversing to the right from the list on top & we go one step down if the next item is bigger than the key that we want to insert, else go right. once we reach at a position in bottom list where we can't go more to the right we insert the new value on the right side of that position.

Now comes the probabilistic part, after inserting the new node in the list Li we have to also insert the node in list above it. i.e list L+1 with probability 1/2. we do this with a simple coin toss, such that if we get a head we insert the node in the list

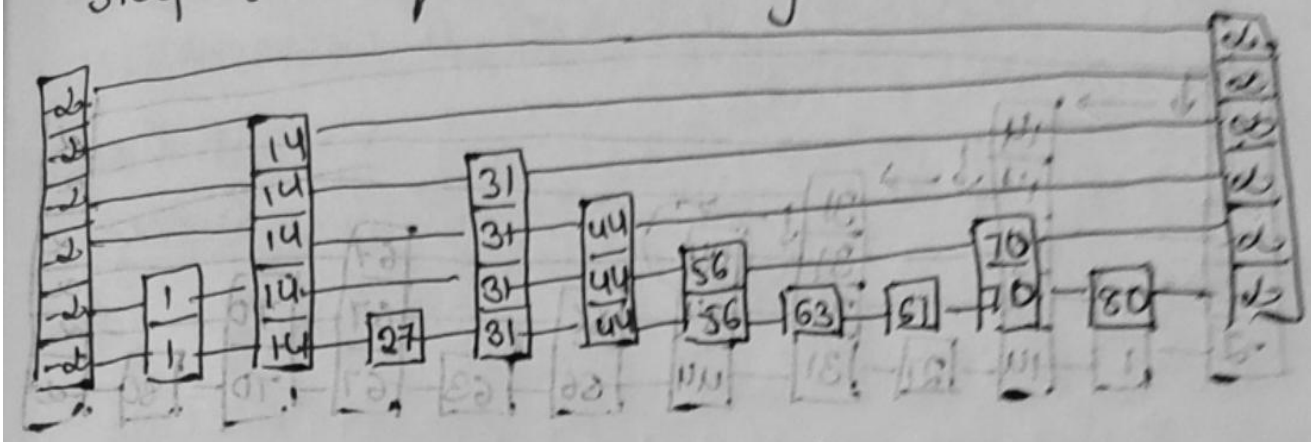
Liti & tos the coin again, we stop when we get a tail we may repeat the coin tos still we keep getting heads, even if we have to insert a new layer (list) at the top.

Ex we also need to re-hook the links for the newly inserted node; the new node needs to have its below reference pointing to the node below and the node below would have above reference pointer to the new node, we simply traverse toward left from the node below it & return above pointer from a node for which above pointer is not null. The right reference of the left neighbor is used to update the right reference of the new node. Also the left reference for the new nodes right neighbor is also updated to the new node. this re-hooking operation is actually pretty easy to implement with linked lists.

Ex: Finding 67



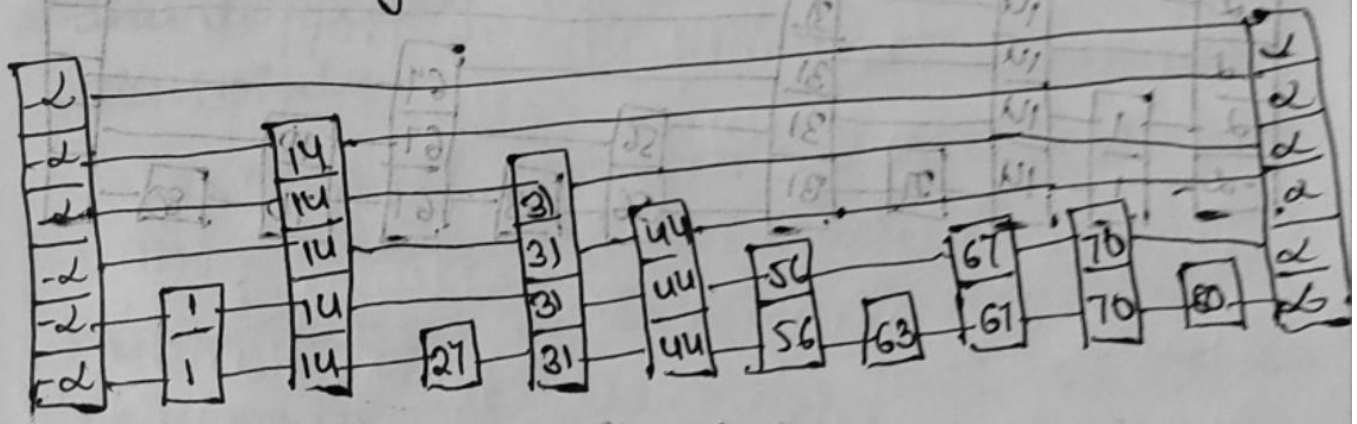
Skip list after inserting 67:



Skip list After re-hooking

Suppose we got head while tossing the coin.

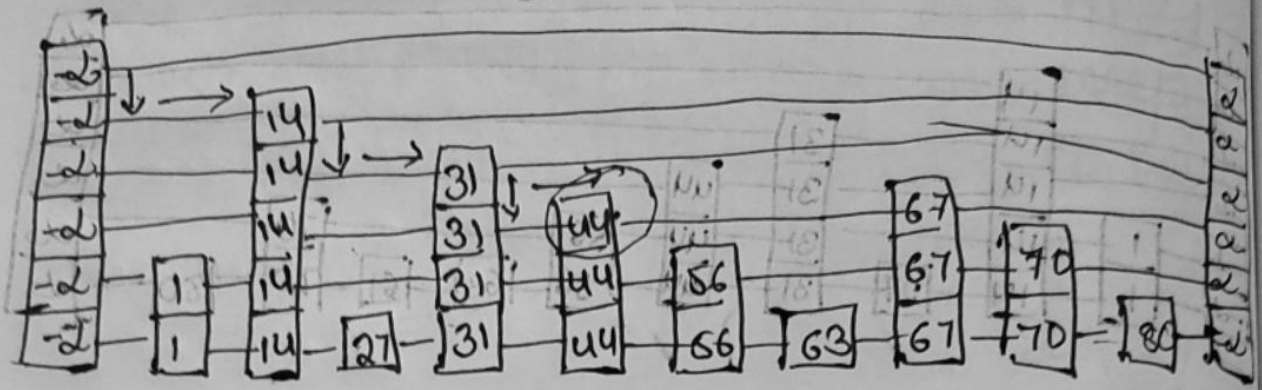
Again we got head while tossing, & we got tail for the third time.



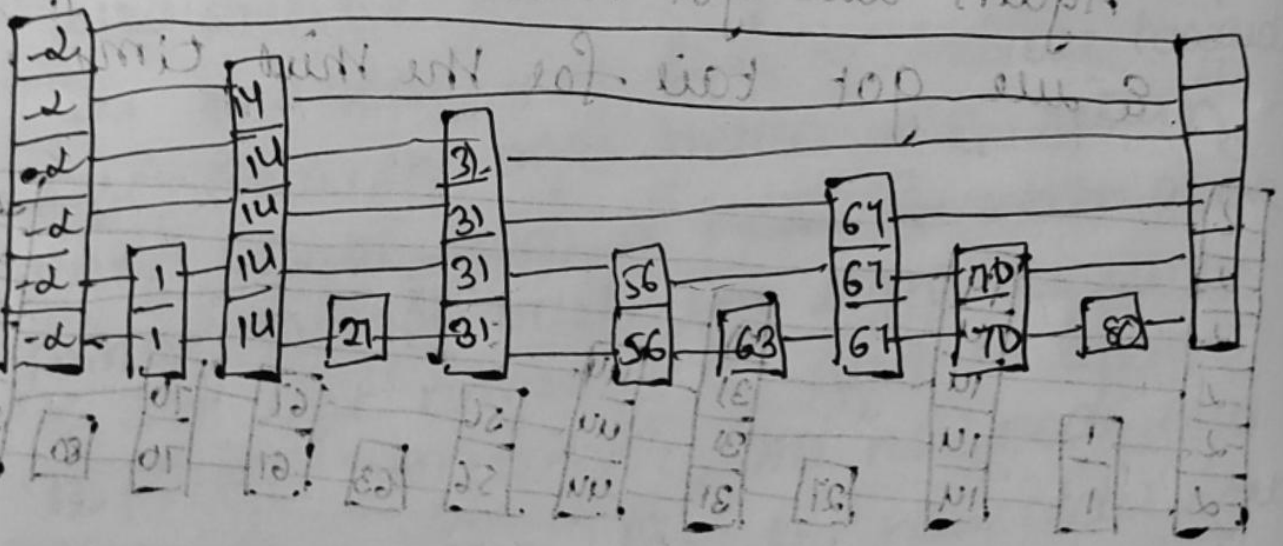
Deletion in the Skip list :-

We first perform search operation to find the location of the node. If we find the node. We simply delete it all levels before deleting a node. we simply ensure that no one node is pointing to it.

Ex Delete 44



After finding release all references to this node & move to node below & release that node as well



Analysis of Skip list

→ Search complexity is O(log n) in Average case & O(n) in worst case

→ Delete complexity is O(log n) in Average case & O(n) in worst case

→ Insert complexity is O(log n) in Avg. case & O(n) in worst case

→ Space complexity is O(n log n) in worst case

→ The height of a skiplist containing n elements is atmost $\log n + 2$

⇒

Universal hashing :-

if we have a fixed hash function such that all keys uses this single hash function, then we may be unfortunate that the keys selected may be such that they all hash to the same slot in hash table. This would be result in an average search time of $O(n)$. We want to avoid this happening or at least limit its possibility.

universal hashing is a way to avoid this situation. In universal hashing the hash function is selected randomly independent of the keys that are to be stored.

In universal hashing at the beginning of the execution we choose a hash function randomly from a carefully designed family of functions this guarentees that no single input would result in worst case situation since the hash functions are random the behaviour would be different for

each execution which would yield average case performance on each input.

Let 'H' be a finite set of hash functions that map a given universe 'U' of keys into the range $\{0, 1, 2, \ldots m-1\}$. This set of functions is said to be universal if for each pair of distinct keys $k, l \in U$ the no. of hash function $h \in H$ for which $h(k) = h(l)$ is at most $|H|/m$

This means that with a hash function randomly chosen from the set 'H' the possibility of collision b/w two distinct keys k & l is atmost the chance of $1/m$ of a collision occuring if $h(k)$ & $h(l)$ were chosen randomly & independently from the set $\{0, 1, 2 \ldots m-1\}$