Unit I : Dictionaries : Sets, Dictionaries, Hash Tables, Open Hashing, Closed Hashing(Rehashing Methods),Hashing Functions(DivisionMethod,MultiplicationMethod,UniversalHashing),Analysisof ClosedHashingResult(UnsuccessfulSearch,Insertion,SuccessfulSearch,Deletion), HashTableRestructuring,SkipLists,AnalysisofSkipLists.

Data :-

Data is the basic entity of fact that is used in calculation or manipulation process.

Data Structure :—

The way of organizing of the data & performing the opesations is called as DS.

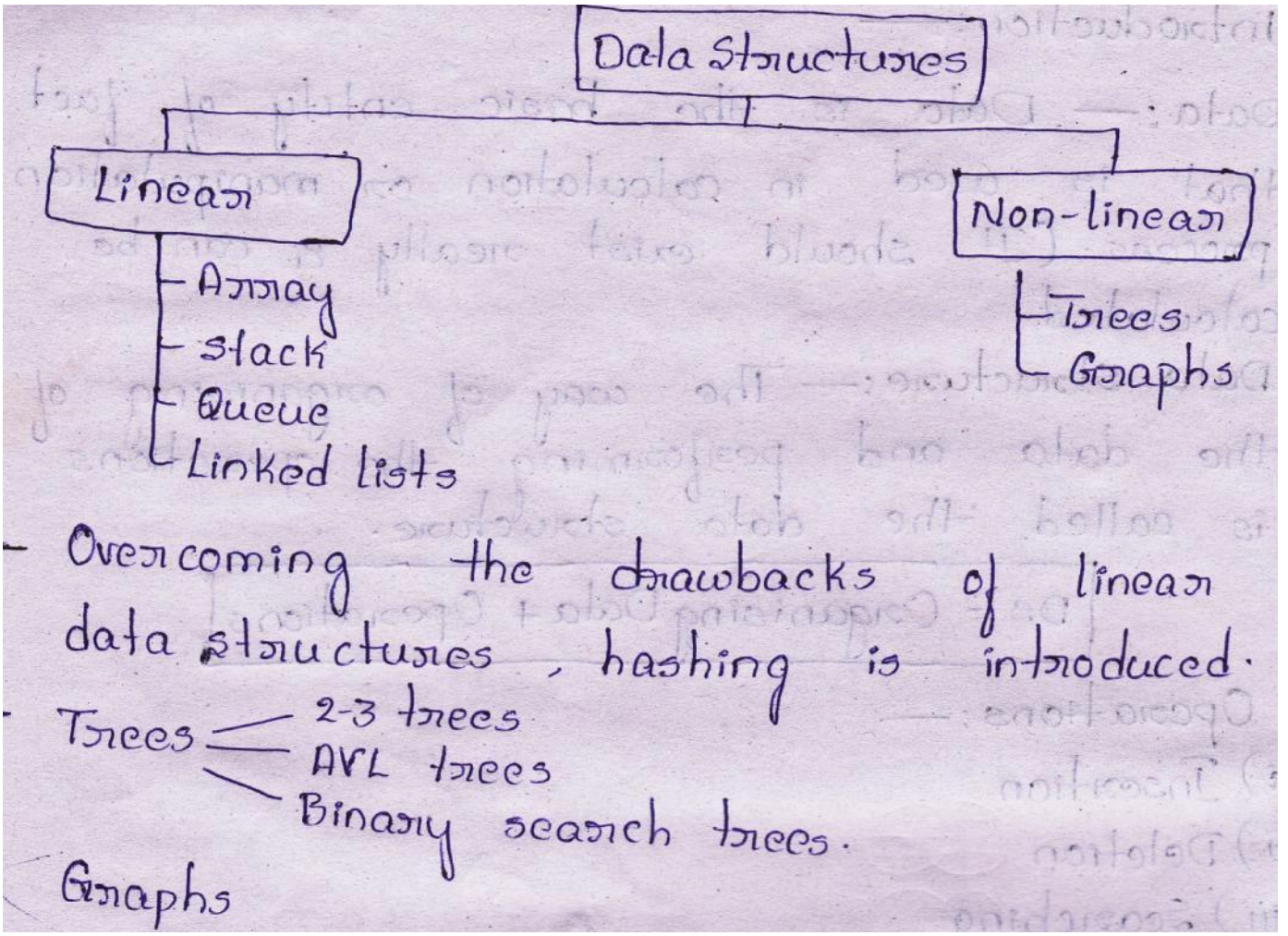
DS = Organized data + Opesations

Operations :— (1) Insestion
(2) Deletion
(3) Searching
(4) Traversing

The organization must be convenient for users.

DS are implemented in the real time in the following situations :—

1. Car park
2. File storage
3. Machinery
4. Shortest path
5. Sorting
- Networking (Lane connections).
- Evaluation of expression.



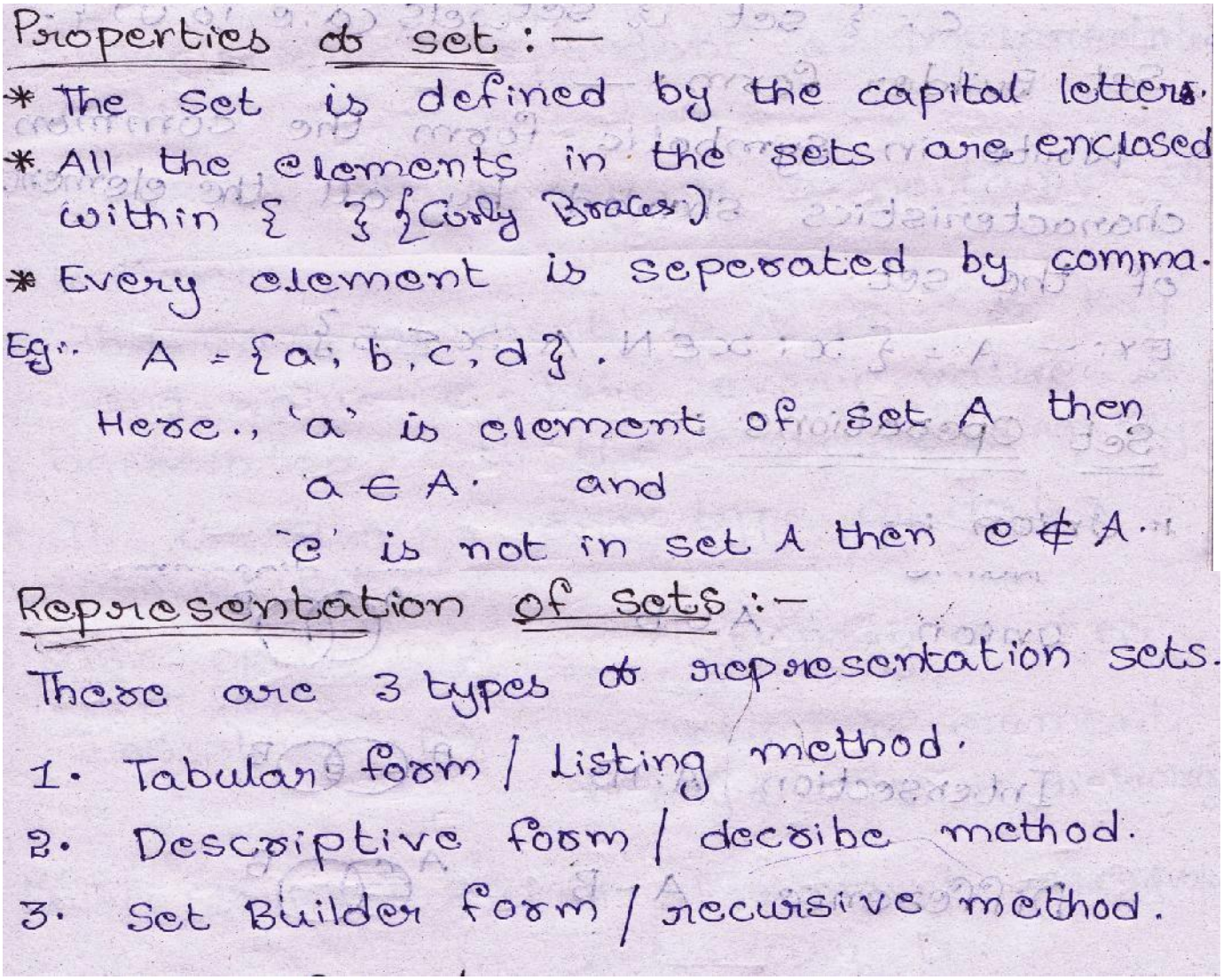
- Overcoming the drawbacks of linear data structures, hashing is introduced.
- Trees — 2-3 trees
  AVL trees
  Binary search trees.
- Graphs

Unit-I
Dictionaries

SET:-A set is a collection of welldefinedelements.The members ofasetare alldifferent.
A set is a group of "objects"

- People in a class: { Alice, Bob, Chris }
- Classes offered by a department: { CS 101, CS 202, … }
- Colors of a rainbow: { red, orange, yellow, green, blue, purple }
- States of matter { solid, liquid, gas, plasma }
- States in the US: { Alabama, Alaska, Virginia, … }
- Sets can contain non-related elements: { 3, a, red, Virginia }

- Although a set can contain (almost) anything, we will most often use sets of numbers
  - All positive numbers less than or equal to 5: {1, 2, 3, 4, 5}
  - A few selected real numbers: { 2.1, π, 0, -6.32, e }



(I) Tabular Form:
Listingalltheelementsofasetandseparatedby commasandenclosedwithincurlybrackets {}.

EX: $A=\{1,2,3,4,5\}$, $B=\{2,4,6,\ldots,50\}$, $C=\{1,3,5,7,9,\ldots\}$

(II) DescriptiveForm:
Stateinwordstheelementsofaset.Thatis,thepropertyofelementsinthesetdefendastheset

EX:

$A =$ Set ofthe first five natural numbers.

$B =$ Set of positive even integers less or equal to fifty

$C =$ Set ofpositiveodd numbers.

(III)Set BuilderForm:

Writingin symbolic form the commoncharacteristic sharedbyall theelements ofthe sets.Ex:

$A = \{x : x \in \mathbb{N} \wedge x \leq 5\}$, $B = \{x : x \in E \wedge 0 < x \leq 50\}$, $C = \{x : x \in O \wedge x > 0\}$

Descriptive form/Describe method/Statement form:

In this, well-defined description of the elements of the set is given and the same are enclosed in curly brackets.

For example:

(i) The set of odd numbers less than 7 is written as: **{odd numbers less than 7}.**

(ii) A set of football players with ages between 22 years to 30 years.

(iii) A set of numbers greater than 30 and smaller than 55.

2. Tabular form/ Listing method/ Roster form or tabular form:

In this, elements of the set are listed within the pair of brackets { } and are separated by commas.

For example:

(i) Let N denote the set of first five natural numbers.

Therefore, N = {1, 2, 3, 4, 5} → Roster Form

(ii) The set of all vowels of the English alphabet.

Therefore, V = {a, e, i, o, u} → Roster Form

(iii) The set of all odd numbers less than 9.

Therefore, X = {1, 3, 5, 7} → Roster Form

3. Set builder form

In this, a rule, or the formula or the statement is written within the pair of brackets so that the set is well defined. In the set builder form, all the elements of the set, must possess a single property to become the member of that set.

In this form of representation of a set, the element of the set is described by using a symbol ‘x’ or any other variable

followed by a colon The symbol ':' or '|' is used to denote such that and then we write the property possessed by the elements of the set and enclose the whole description in braces. In this, the colon stands for 'such that' and braces stand for 'set of all'.

Let P is a set of counting numbers greater than 12;

the set P in set-builder form is written as :

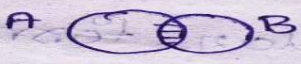
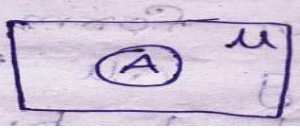
P = {x : x is a counting number and greater than 12}

or

P = {x | x is a counting number and greater than 12}

This will be read as, 'P is the set of elements x such that x is a counting number and is greater than 12'.

Set Operations :—

Name		venn diagram
union	$A \cup B$	
Intersection	$A \cap B$	
Difference	$A - B$	
Complement	$\bar{A}$ $= U - A$	

Definition A dictionary is an ordered or unordered list of key-element pairs, where keys are used to locate elements in the list.

Example: consider a data structure that stores bank accounts; it can be viewed as a dictionary, where account numbers serve as keys for identification of account objects.

Dictionaries:—

A dictionary is a dynamic set ADT (Abstract data type).

* ADT is an object with a generic description independent of implementation details.
* A dictionary is a container of elements. $(a+b)^2 = a^2+b^2+2ab$
* The each element is a pair of key and value, where every value is associated with the corresponding key.
* It is also one type of data structure.

Basic Operations:—

Insert (x, D) → insertion of element x (key & value) in dictionary D

Delete (x, D) → deletion of element x (key & value) in dictionary D with the help of key corresponding to x.

Search (x, D) → searching prescribed value x in the dictionary D with a key of an element x.

Member (x, D) → It returns true if $x \in D$ else return false.

Size (D) → It returns the count of total no-of elements in D.

MAX (D) → It returns the maximum element in the dictionary D.

Consider an empty unordered dictionary and the following set of operations:

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,C), (8,D), (2,E)}	
findItem(7)	{(5,A), (7,B), (2,C), (8,D), (2,E)}	B
findItem(4)	{(5,A), (7,B), (2,C), (8,D), (2,E)}	NO_SUCH_KEY
findItem(2)	{(5,A), (7,B), (2,C), (8,D), (2,E)}	C
findAllItems(2)	{(5,A), (7,B), (2,C), (8,D), (2,E)}	C, E
size()	{(5,A), (7,B), (2,C), (8,D), (2,E)}	5
removeItem(5)	{(7,B), (2,C), (8,D), (2,E)}	A
removeAllItems(2)	{(7,B), (8,D)}	C, E
findItem(4)	{(7,B), (8,D)}	NO_SUCH_KEY

MIN (D) ⟶ It returns the minimum element in the dictionary D.

Implementation of dictionary:—

1) Fixed length Array:— It is an array
2) Linked list
   * Sorted list
   * Skip list
3) Hashing
4) Trees
   * Binary search tree (BST)
   * Balance BST < AVL tree, Red black tree
   * Splay trees
   * multiway search tree
   * Tries.

6/9/2014

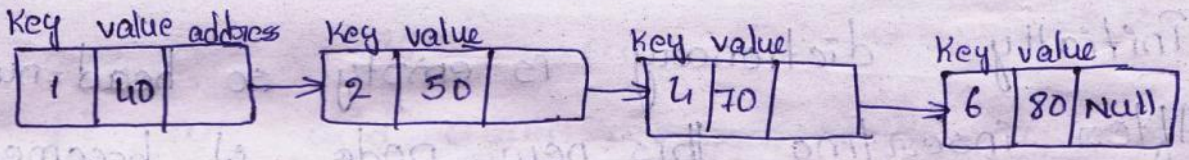
- Representation of dictionary by using LinkedList:

The dictionary can be represented as a linear list. The linear list is a collection of pairs. (key and value).

There are 2 methods in representation of a dictionary in linked list.

i) Sorted Array.

ii) Sorted chain.



- The contents of dictionary are always in sorted form.

(Pseudocode)

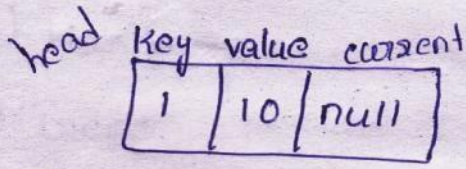
- Structure of LL for representing dictionary:-

```
class DLL
{
 struct node
 {
  int key;
  int value;
  struct node * next;
 }head;
 void insert();
 void delete();
 vord print();
```

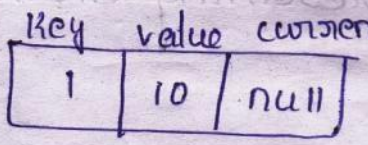
int void length();
}

i) Insertion:—

- Consider initially the dictionary is empty. That means. i.e., head is null.
- We will create a new node with some key & value.



- Initially dictionary is empty, so head=null.
- After inserting this new node, it becomes head node.
- This node will be current and previous.



- This node will be 'curr' and 'prev' as well.

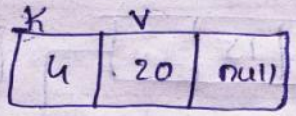
curr ⟶ will always point to current visiting node.

prev ⟶ will always point to previous to current node.

- Above figure shows there is one node in the list. Curr & loc will node 'curr'

node as prev node.

- If we want to insert more records like. K=4, value=20, we will create such a new node.



new node

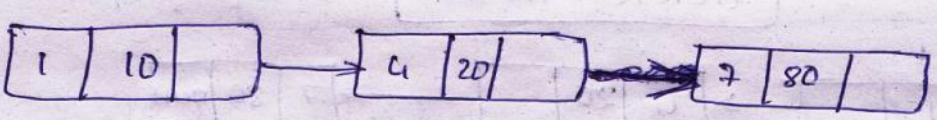
if new → key > curr → key

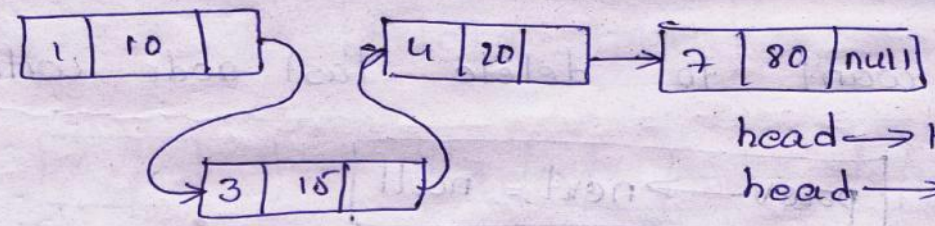
4 > 1

curr → next = new

prev = curr

- If we add again a new node <7, 80> then



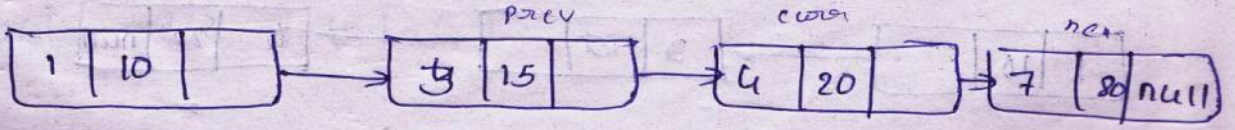


head → key < new → key

head → curr = new

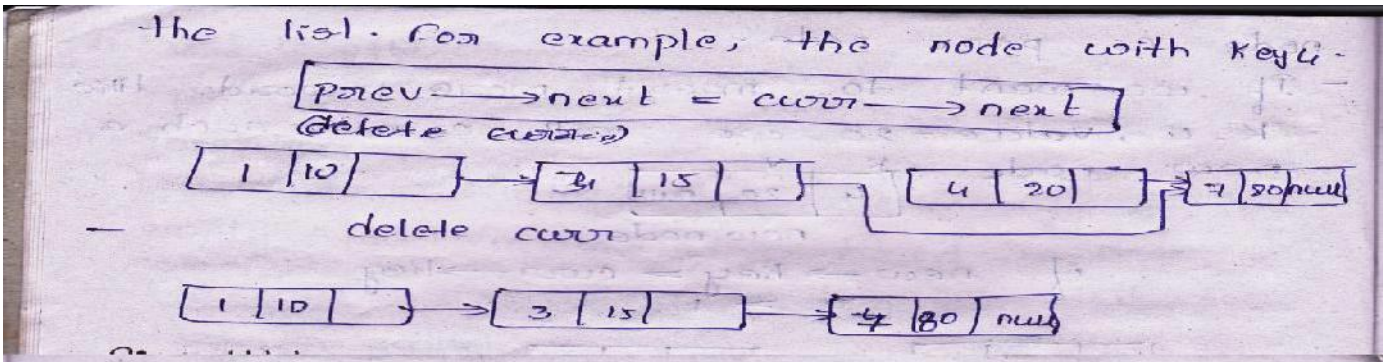
new → next = curr

2) Deletion :—



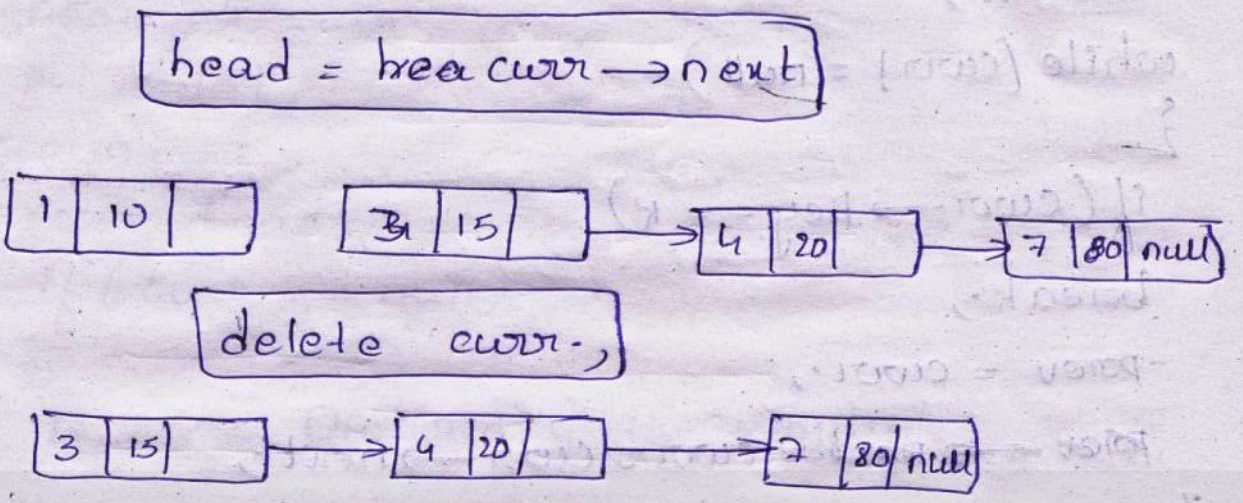
case 1:—

If we want to delete middle node of

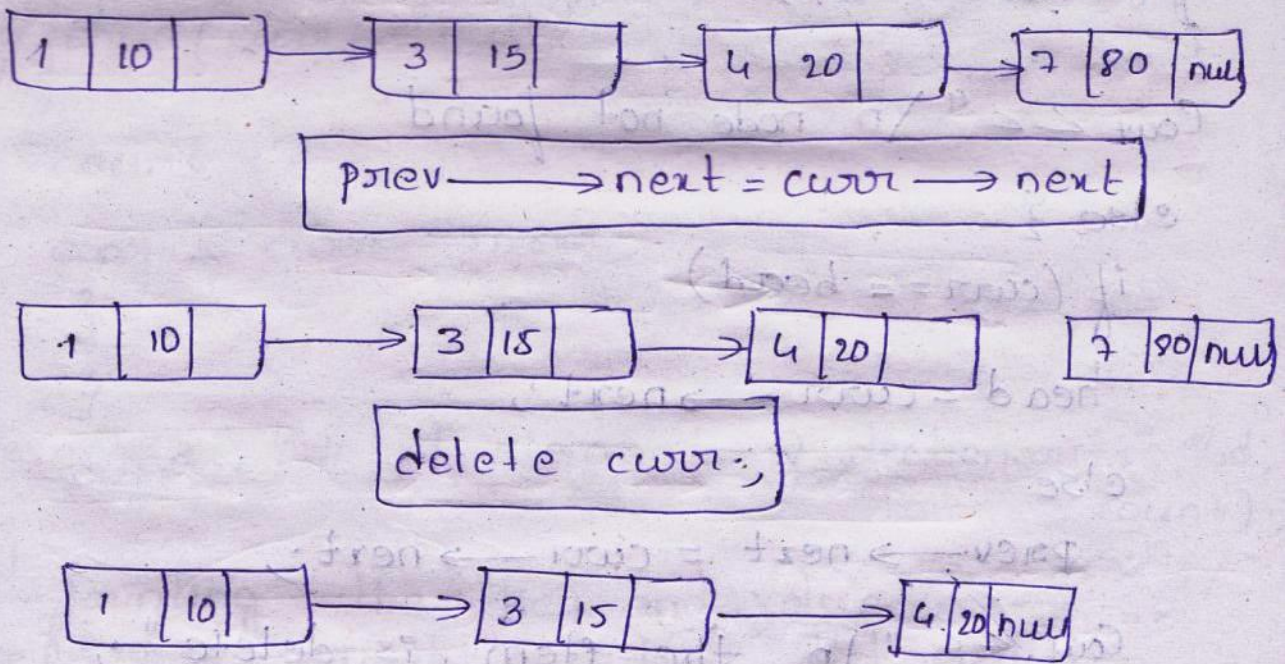


case (ii):—

If we want to delete head node with key 1



case(iii):—



The delete operation

```
void sll:: del()
{
  node *curr, *prev ;
  int   k;
  curr=head;
  clrscr();
  cout<<"\nEnter the key value that you want to delete: ";
  cin>>k;
  while(curr!=NULL)
  {
    if(curr->key==k)//traverse till required node to delete
       break;                        //is found
    prev=curr;
       curr=curr->next;
    }
    if(curr==NULL)
        cout<<"\nNode not found";
        else
        {
          if(curr==head)  //first node
            head=curr->next;
          else
            prev->next=curr->next;   //intermediate or end node
          delete curr;
          cout<<"\nThe item is deleted\n";
       }
    getch();
  }
```

The length operation

```
int sll ::length()
{
       node *curr ;
       int   count;

       count = 0;
       curr = head;
       if ( curr == NULL )
       {
            cout<<"The list is empty\n";
            getch();
            return 0;
       }
       while ( curr != NULL )
       {
            count++;
            curr = curr -> next;
       }
       getch();
       return count;
}
```

```
void sll ::print()
{
        node *temp ;

        temp = head;
        if ( temp == NULL )
        {
             cout<<"\nThe list is empty\n";
             getch(); clrscr();
             return;
        }
        while ( temp != NULL )
        {
             cout<<" <"<<temp->key<<","<<temp->value<<"> ";
             temp = temp -> next;
        }
        getch();
}
```

→ Searching operation;——

```
void dll::find()
{
 node * curr , * prev;
 int k;
 curr = head;
 Cout << "Enter key, that we want to search";
 Cin >> k;
 while(curr != null)
 {
  if (curr -> key == k) {
 Cout << " key is found";
 break;
 prev = curr;
 curr = curr -> next;
 }
 }
 if (curr == null)
 {
 Cout << "dictionary is empty";
 }
 }
```

Selecting an implementation

	Insertion	Removal	Retrieval	Traversal
Unsorted array-based	O(1)	O(*n*)	O(*n*)	O(*n*)
Unsorted link-based	O(1)	O(*n*)	O(*n*)	O(*n*)
Sorted array-based	O(*n*)	O(*n*)	O(log *n*)	O(*n*)
Sorted link-based	O(*n*)	O(*n*)	O(*n*)	O(*n*)
Binary search tree	O(log *n*)	O(log *n*)	O(log *n*)	O(*n*)

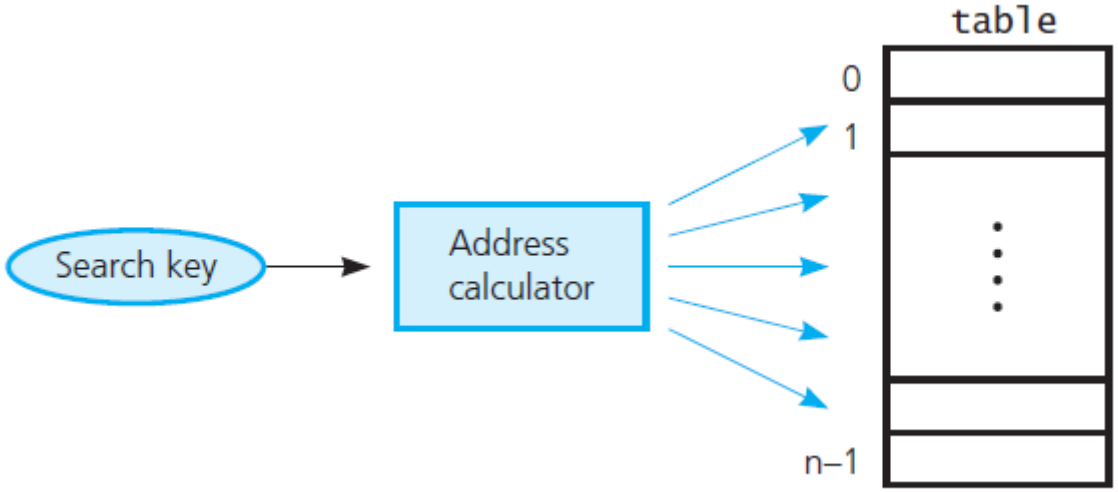
- **Hashing:—**

Hash Table:—

- Hash table (also hash map) is a data-structure used to store and retrieve data very quickly.
- Insertion of data in the hash table is based on the key value. Every entry in the hash table is associated with some key.

Eg:— Storing the voter recode in the hash table, voter id will work as key.

- **Hashing:—** Hashing is the process of mapping large amount of data item to a small table with the help of hash function. means we can place the dictionary entries <key, value> in the hash table using hash function.



Hash Table is a data structure in which keys are mapped to array positions by a hash function.
A Hash Table is a data structure for storing key/value pairs
This table can be searched for an item in O(1) time using a hash function to form an address from the key.
Hash Function: ***Hash function is any well-defined procedure or mathematical function which converts a large, possibly variable-sized amount of data into a small datum, usually a single integer that may serve as an index into an array***

· ***Hash function is a function which maps key values to array indices. (OR)***
· ***Hash Function is a function which, when applied to the key, produces an integer which can be used as an address in a hash table.***
· ***We will use h(k) for representing the hashing function***
Hash Values: The values returned by a hash function are called hash values or hash codes or hash sums or simply hashes

Hashing is the process of mapping large amount of data item to a smaller table with the help of a hashing function.

- Hash table is an extremely effective and practical way of implementing dictionaries.
- It takes O(1) time for search, insert, and delete operations in the **average case. And O(n) time in the worst case.**

- Hash function:—

The fixed process to convert a key to a hash key is known as Hash Function, which is used to put the data in the hash table and same hash function is used for retrieve the data from the hash table.

→ Hash function is used to implement the hash table.

→ The integer (returned written) by the hash function is called hash - key.

→ One common method (hash function) for determining the hash key is 'Division-method' or 'Hashing'.

Syntax:—

Hash Key = key % size of the table

Eg:—

voter id	Name	age
57	aaa	25

- consider that we want to place some voter records in the hash table.
- The voter record is placed with the help of key.

- Here voter id is the key.

Note:— If the voter_id is longer number (Eg:- seven digit number) then consider last 3 digits of the voter_id as key.

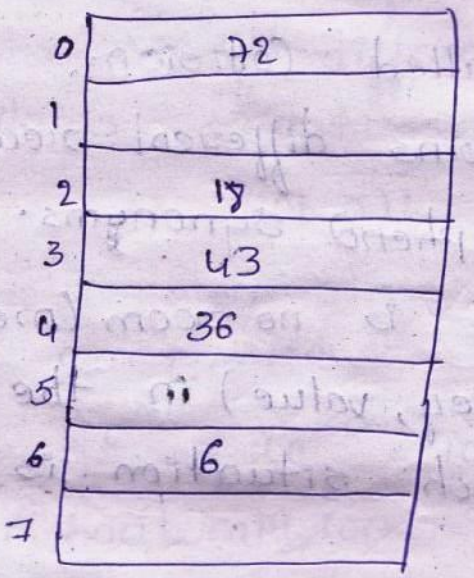
- If the table size is 100, then the hash function will be

$$H(key) = key \% 100$$

For example:—

Assume a table with 8 strokes put the values in the hash table {36, 18, 72, 43, 6}. size of the table = 8.

Hash table



$H(36) = 36 \% 8 = 4$

$H(18) = 18 \% 8 = 2$

$H(72) = 72 \% 8 = 0$

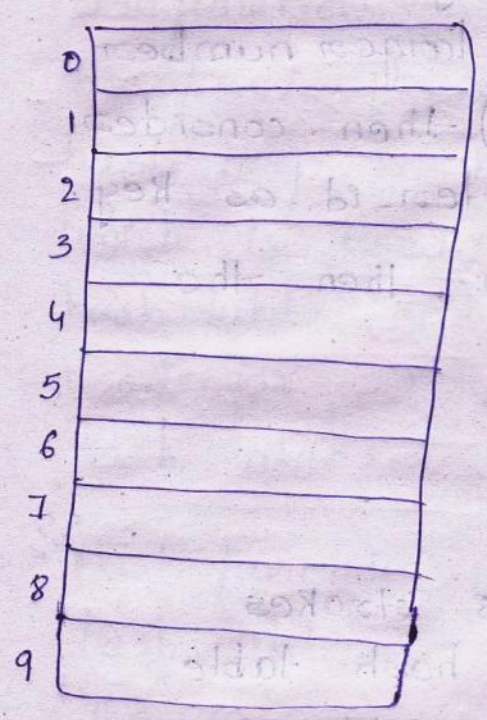
$H(43) = 43 \% 8 = 3$

$H(6) = 6 \% 8 = 6$

- Collision:—

{131, 3, 4, 21, 61, 24, 7, 97, 89}

$ts = 10$ (table size)



H(131) = 131%10 = 1
H(3) = 3%10 = 3
H(4) = 4%10 = 4
H(21) = 21%10 = 1
H(61) = 61%10 = 1
H(24) = 24%10 = 4
H(7) = 7%10 = 7
H(97) = 97%10 = 7
H(89) = 89%10 = 9

collision

Def:— If the hash function returns same hash key for more than one record; then this situation is called Collision.

- If the hash key returns different records is called ~~phenomns~~ phend synonyms.
- Similarly, when there is no room (memory location) for new pair (key, value) in the hash table then such situation is called overflow.
- Few times when we handle collision, it may be lead to overflow ~~collision~~ condition.
- Collisions and overflow shows the poor hash function.

Collisions: If x1 and x2 are two different keys, but the hash values of x1 and x2 are equal (i.e., h(x1) = h(x2))

then it is called as a collision.

Ex: Assume a hash function = h(k) = k mod 10

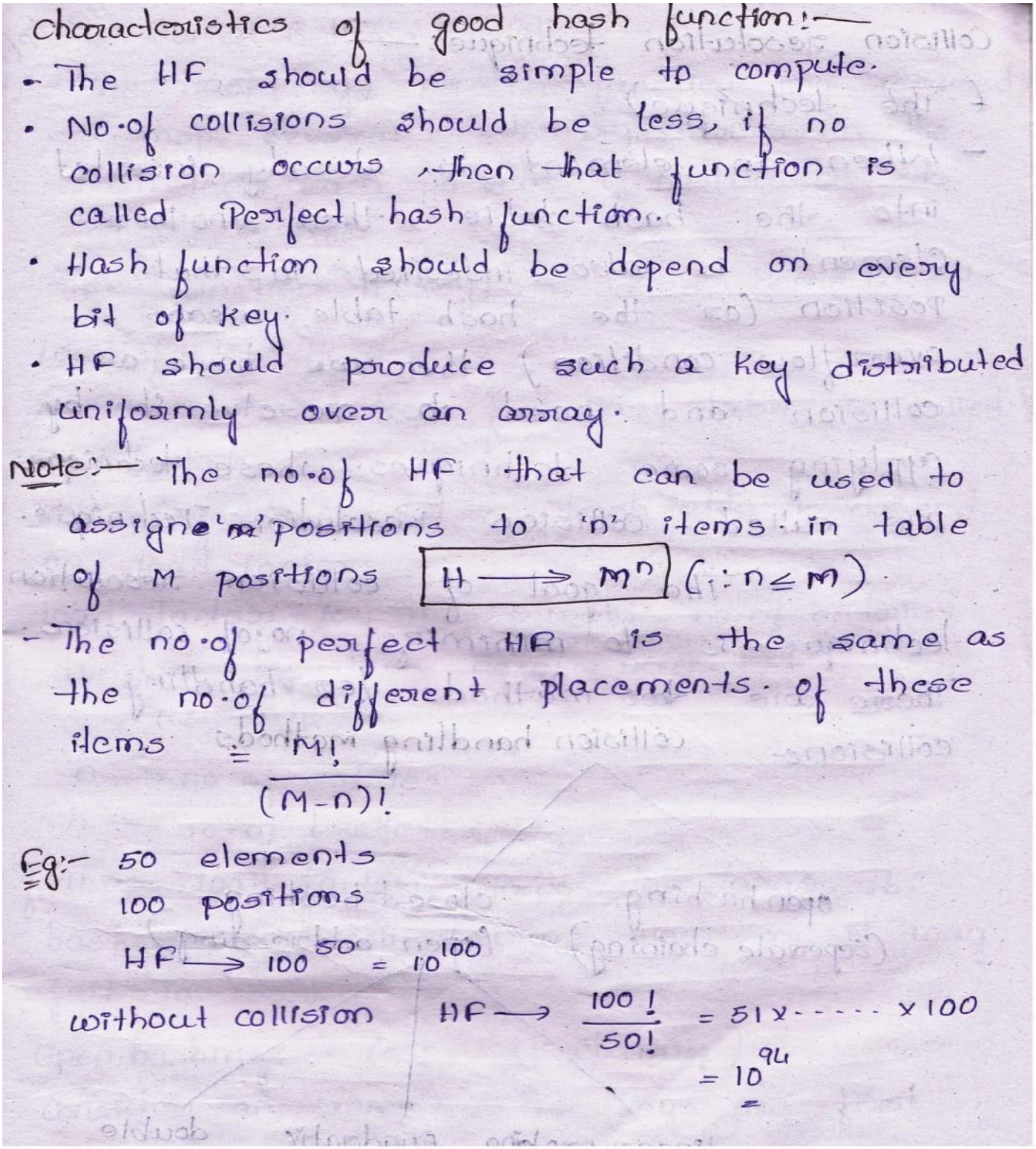
h(19)=19 mod 10=9

h(39)=39 mod 10=9

here h(19)=h(39) this is called collision.

Collision resolution is the most important issue in hash table implementations. To resolve the collisions two techniques are there.

1. Open Hashing 2. Closed Hashing



Perfect Hash Function is a function which, when applied to all the members of the set of items to be stored in a hash table, produces a unique set of integers within some suitable range. Such function produces no collisions.

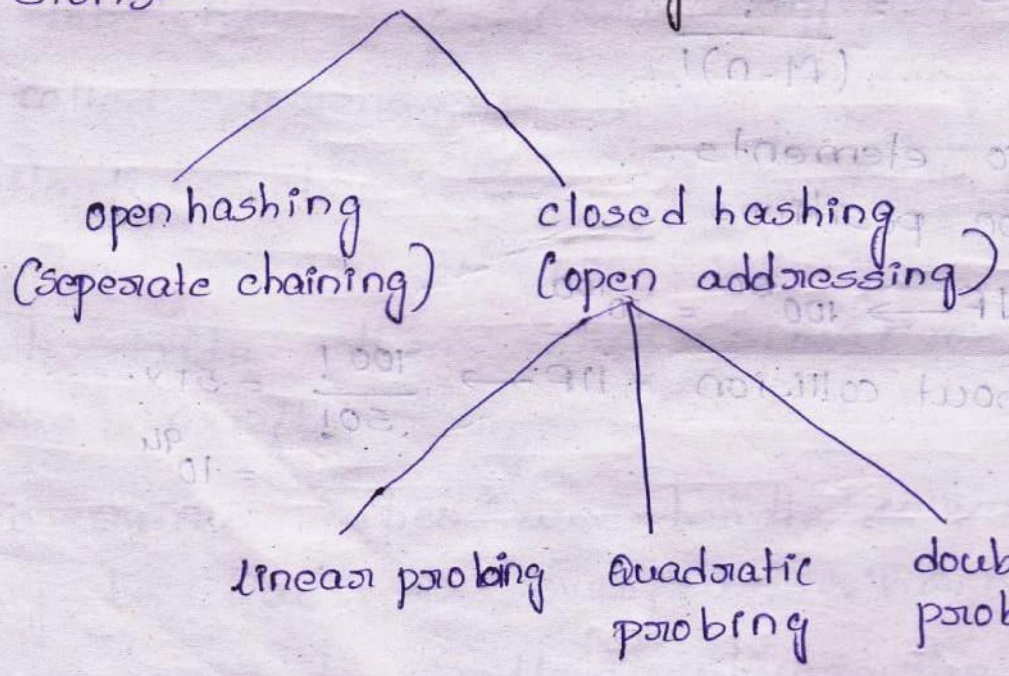
Good Hash Function: minimizes collisions by spreading the elements uniformly throughout the array.

Collision resolution technique:—

(The technique)

- When an element is already inserted into the hash table, then another element is also inserted in that position (or the hash table means overflow condition) then we have a collision and need to resolve it by applying some techniques. These techniques are called collision Resolution Technique.

The goal of collision resolution technique is to minimize no.of collisions. There are two methods for handling collisions.



11/9/2014

Bucket & Home Bucket:—

The Hash(key) or hashfunction key is used to map several dictionary entries in the hash table. Each position of hash table is called Bucket.

The function H(Key) is called Home Bucket.

Load factor:—

A critical statistic for hash table is called Load factor.

The performance of a hash table is depends on load factor.

Load factor is the simply no.of entries divided by the no.of buckets.

i.e; n/k

$n \rightarrow$ no.of entries

$k \rightarrow$ no.of buckets.

If the load factor grows too large, the hash table will become slow or it may fail to work.

Open hashing:— (Seperate chaining)

Consider an example, the keys are first ten perfect squares.

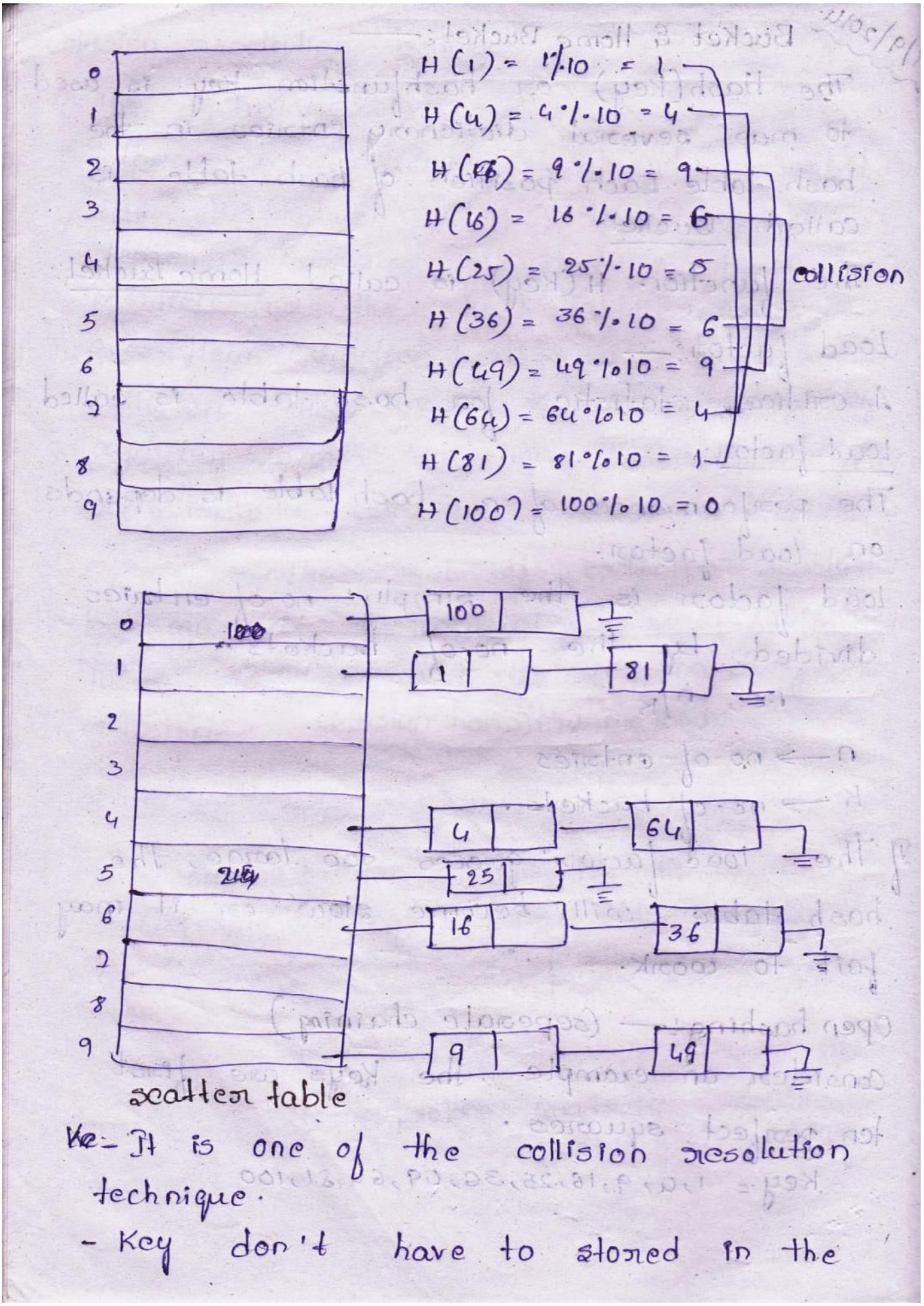
Key = 1, 4, 9, 16, 25, 36, 49, 64, 81, 100

(1) Open Hashing (OR) Separate Chaining:

- In this case hash table is implemented as an array of linked lists.
- Every element of the table is a pointer to a list. The list (chain) will contain all the elements with the same index produced by the hash function.
- In this technique the array does not hold elements but it holds the addresses of lists that were attached to every slot.
- Each position in the array contains a Collection of values of unlimited size (we use a linked implementation of some sort, with dynamically allocated storage).

Here we will chain all collisions in lists attached to the appropriate slot. This allows an unlimited number of collisions to be handled and doesn't require a priori knowledge of how many elements are contained in the collection.

The tradeoff is the same as with linked lists versus array implementations of collections: linked list overhead in space and, to a lesser extent, in time.



Ke:- It is one of the collision resolution technique.

- Key don't have to stored in the

table itself.
- In chaining, each position of table is associated with linked list or chain of structure.
- This method is called seperate chaining & a table of reference (is called a scatter table).
- In this method the table can never overflow because the Linked lists extended only upto the arrival of new keys.

Drawback:—
- For short linked list, this is very fast. but increasing the length of the list can significantly degrad retrival performance.
- The load factor of the open hashing / seperate chaining is too large, so the performance is degraded.

closed hashing / open addressing:—
- It is one of the collision resolution technique.
- It overcomes the dracoback of seperate chaining.
- In this solution, collisions are resolved by tracing the collision element in the alternative bucket (cell), until an empty bucket is found.
- Suppose cells $h_0(x), h_1(x), \ldots$ are tried to succession.

Advantages

1) The hash table size is unlimited. In this case you don't need to expand the table and recreate a hash function.

2) Collision handling is simple: just insert colliding records into a list.

Disadvantages

1) As the list of collided elements (collision chains) become long, search them for a desired element begin to take longer and longer.

2) Using pointers slows the algorithm: time required is to allocate new nodes.

Algorithm for Separate chaining hashing: search

Open hashing is implemented with the following data structures

```
struct node
{
int k;
struct node *next;
};
struct node *r[10];
typedef struct node list;
```

```
list*search( key, r )
 {
list *p;
 p = r[ hashfunction(key) ];
 while ( p!=NULL && key!=p->k )
  p = p->next;
 return( p );
}
```

Algorithm for Separate chaining hashing: insertion

```
void insert( key, r )
   {
       int i;
     i = hashfunction( key );/*evaluates h(k)*/
     if (empty(r[i])) /*** insert in main array ***/
               r[i].k = key;
          else /*** insert in new node ***/
              r[i].next = NewNode( key, r[i].next );
     }
```

where $h_i(x) = (hash(x) + f(i)) \text{ Mod T-size}$

f → collision resolution strategy.

- In this hashing load factor should be below $\lambda = 0.5$.

Types:— There are 3 types of closed hashing.

i) Linear probing.
ii) Quadratic probing
iii) Double probing

Probing is investigation.

i) Linear Probing:—

$h_i(x) = (hash(x) + f(i)) \text{ Mod T-size}$

- In linear probing $f(i) = i$.
- In linear probing, the position in which a key can be stored is found by sequentially searching all positions starting from the position calculated by hash function until empty cell is found.
- If the end of the table is reached and no empty cell has been found then the searching continue from the beginning of the table.

1. **Linear Probing:**
   - In case of linear probing, we are looking for an empty spot by incrementing the offset by 1 every time.
   - We explore a sequence of location until an empty one is found as follows:

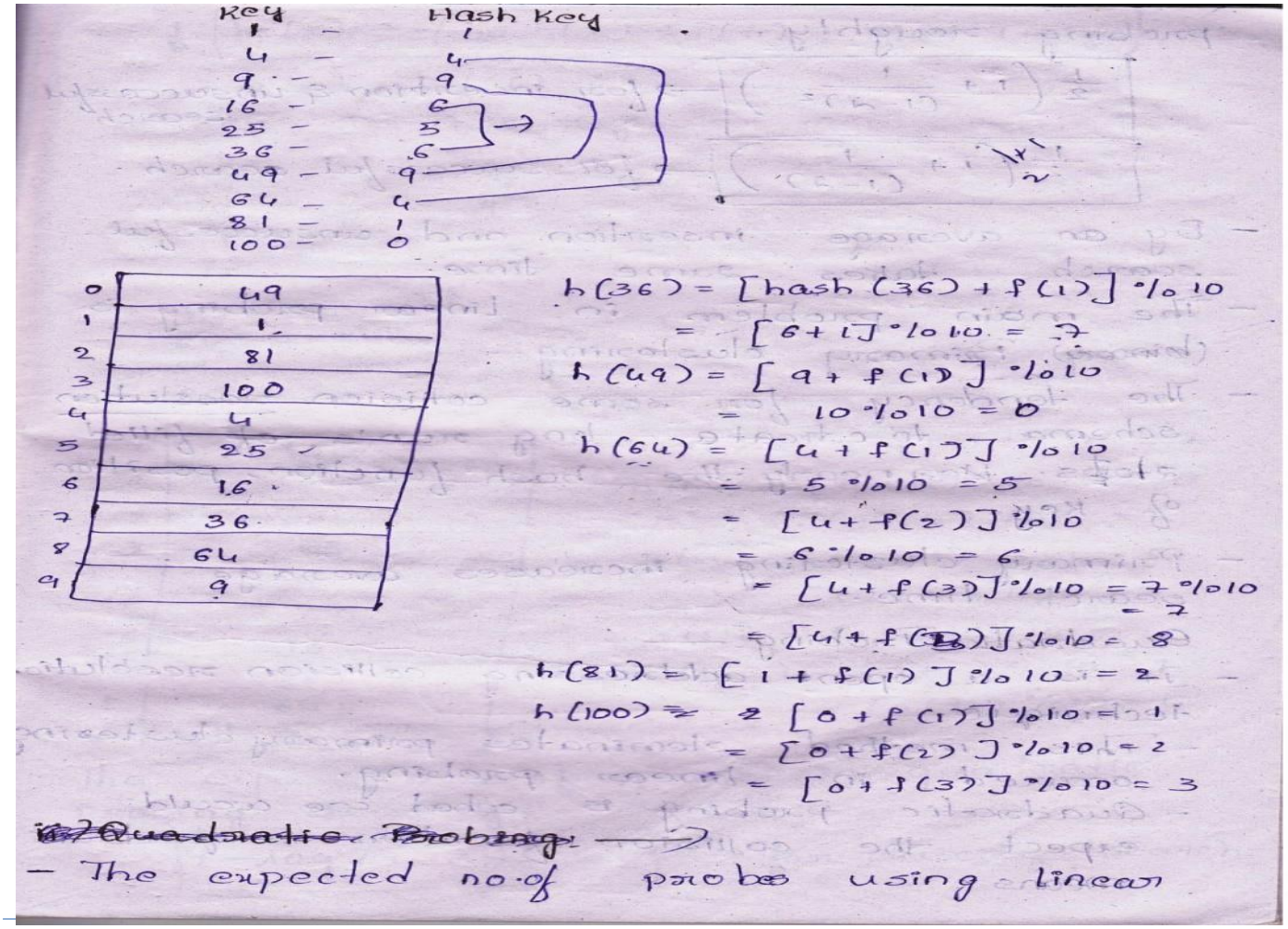
h(x, i) = (h(x) + i) mod m

where **m** is the hash table size
and
i = 0, 1, 2, . . . , m-1

It is just h(x), h(x)+1, h(x)+2, ..., wrapping around at the end of the array. The idea is that if we can't put an element in a particular place, we just keep walking up through the array until we find an empty slot.

Advantages to this approach :

- All the elements (or pointer to the elements) are placed in contiguous storage. This will speed up the sequential searches when collisions do occur.
- It Avoids pointers;



Disadvantages to this approach

Linear probing suffers from primary clustering problem

- Clustering: Element tend to cluster around elements that produce collisions. As the array fills, there will be gaps of unused locations.
  Suffers from ***primary clustering***:
  - Long runs of occupied sequences build up.
  - Long runs tend to get longer, since an empty slot preceded by *i* full slots gets filled next with probability $(i+1)/m$.
  - © Hence, average search and insertion times increase
- As the number of collisions increases, the distance from the array index computed by the hash function and the actual location of the element increases, increasing search time.
- The hash table has a fixed size. At some point all the elements in the array will be filled. The only alternative at that point is to expand the table, which also means modify the hash function to accommodate the increased address space.

Algorithm for Linear probing hashing: insertion

```
void insert( key, r[])
     { int n;
       int i, last;
     i = hashfunction( key ) ;/*computes h(x)*/
     last = (i+m-1) % m;
     while ( i!=last && !empty(r[i]) && !deleted(r[i]) && r[i]!=key )
           i = (i+1) % m;
     if (empty(r[i]) || deleted(r[i]))
      r[i] = key; /*** insert here ***/
     else Error     /*** table full, or key already in table ***/;
     }
```

probing roughly.

$\frac{1}{2}\left(1+\frac{1}{(1-\alpha)^2}\right)$ → for insertion & unsuccessful search

$\frac{1}{2}\left(1+\frac{1}{(1-\alpha)}\right)$ → for successful search.

- By an average insertion and unsuccesful search takes same time.
- The main problem in linear probing is (binary) Primary clustering.
- The tendency for some collision resolution schema to create long run's of filled slots. Near the hash function, position of key.
- Primary clustering increases average search time.

Quadratic Probing:—

- It is a open addressing collision resolution technique.
  - This method eleminates primary clustering occured in linear probing.
  - Quadratic probing is what we would expect the collision function is quadratic means -

2. **Quadratic Probing**: is a different way of rehashing. In the case of quadratic probing we are still looking for an empty location. However, instead of incrementing offset by 1 every time, as in linear probing, we will increment the offset by 1, 4,9, 16, ... We explore a sequence of location until an empty one is found as follows:

$h(x, i) = (h(x) + i^2) \bmod m$ where **m** is the hash table size **and** **i** = 0, 1, 2, . . . , m-1

- Disadvantage: Can suffer from secondary clustering:
  If two keys have the same initial probe position, then their probe sequences are the same.

```
void insert( key, r[])
     { int n;
       int i, last;
     i = hashfunction( key ) ;/*computes h(x)*/
     last = (i+m-1) % m;
     while ( i!=last && !empty(r[i]) && !deleted(r[i]) && r[i]!=key )
           i = (i*i+1) % m;
     if (empty(r[i]) || deleted(r[i]))
      r[i] = key; /*** insert here ***/
     else Error     /*** table full, or key already in table ***/;
     }
```

$h_i(x) = (hash(x) + f(i)) \text{ Mod Tsize}$

→ here $f(i) = i^2$

Key	hash_key
1	1
4	4
9	9
16	6
25	5
36	6
49	9
64	4
81	1
100	0

0	49
1	1
2	81
3	
4	4
5	25
6	16
7	36
8	64
9	9

$h(36) = (6+1) \% 10 = 7$

$h(49) = (9+1) \% 10 = 0$

$= (9+4) \% 10 = 3$

$h(64) = (4+1) \% 10 = 5$

$= (4+4) \% 10 = 8$

$h(81) = (1+1) \% 10 = 2$

$h(100) = (0+1) \% 10 = 1$

$= (0+4) \% 10 = 4$

- The expected no. of probes in quadratic probing roughly.

$$1 - \log(1-\lambda) - \lambda/2$$ → for successful search.

$\frac{1}{1-\lambda} - \lambda - \log(1-\lambda)$ ⟶ for unsuccessful search.

- ~~(With)~~ The table values is prime:, then the $\lambda \leq 0.5$ (for 100, hash)
- If the $\lambda > 0.5$ then the element did not have location in hash table.

Double Hashing:—

- It is one of the open addressing collision resolution technique.

$$h_i(x) = (hash(x) + f(i)) \text{ Mod T-size}$$

where $f(i) = i \cdot hash_2(x)$

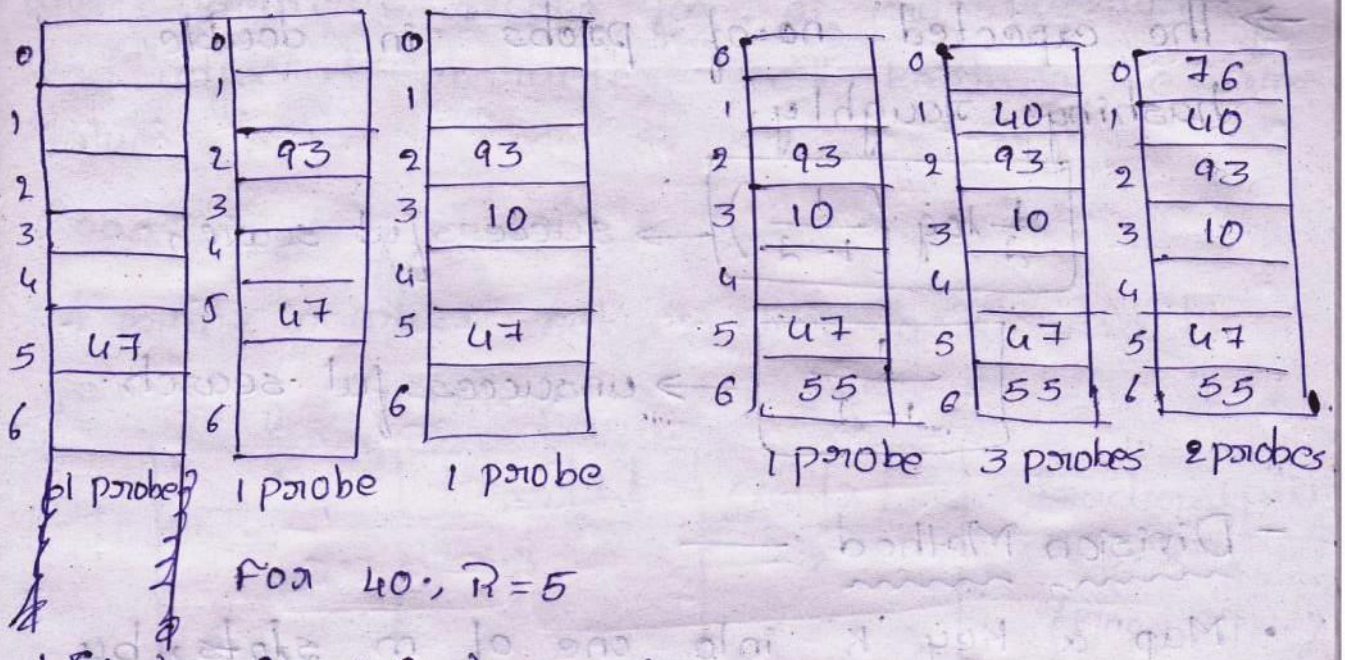
$f(i) = i \cdot hash_2(x)$ formula says that we apply a second hashing to x & prob at distance $hash_2(x)$, $2hash_2(x)$, $3hash_2(x)$, ...

- $hash_2(x)$ is a function.

$$hash_2(x) = R - (x \bmod R)$$

R ⟶ Prime number smaller than table size.

Eg:- 47, 93, 10, 55, 40, 76 , T-size :- 7



For 40:, R = 5

$h(40) = (hash(40) + f(1)) \% 7$

$f(1) = 1 \cdot hash_2(40)$

$hash_2(40) = R - (40 \% R)$

$= 5 - (40 \% 5) = 5 - 0 = 5$

$f(1) = 1 \cdot 5 = 5$

$h(40) = (5+5) \% 7 = 10 \% 7 = 3$

$f(2) = 2 \cdot 5 = 10$

$h(40) = (5+10) \% 7 = 15 \% 7 = 1$

For 76:, R = 5

$hash_2(76) = 5 - (76 \% 5) = 5 - 1 = 4$

$f(1) = 1 \cdot 4 = 4$

$h(76) = (6+4) \% 7 = 10 \% 7 = 3$

$f(2) = 2 \cdot 4 = 8$

$h(76) = (6+8) \% 7 = 14 \% 7 = 0$

Algorithm for Double hashing: insertion

```
void insert( key, r )
     {
       int i, inc, last;
     i = hash1(key) ;
     inc = hash2(key);
     last = (i+(m-1)*inc) % m;
     while ( i!=last && !empty(r[i]) && !deleted(r[i]) && r[i]!=key )
          i = (i+inc) % m;
     if ( empty(r[i]) || deleted(r[i]) )
               {
               /*** insert here ***/
               r[i] = key;
               n++;
               }
     else Error     /*** table full, or key already in table ***/;
     }
```

//Where hash1 (key) is the first hash function,hash2(key) is the second hash function .

→ The expected no. of probs in double hashing roughly.

$\frac{1}{\alpha} \log\left(\frac{1}{1-\alpha}\right)$ → successful search

$\frac{1}{1-\alpha}$ → unsuccessful search.

- Division Method :—

- Map a key 'K' into one of m slots, by taking the remainder of K % m.

i.e; $h(K) = K \% m$

Eg:- T-size $m = 12$

Key $K = 100$

$h(100) = 100 \% 12 = 4$

Poor choice of M:—

- A power of 2^P; since if $m = 2^P$ then h(K) is just the keyp lowest order bits of K.
- A power of 10, since if $m = 10^P$ then the hash function doesn't depend on all the decimal digits of 'K'.
- $m = 2^P - 1$:- If K is a character string interrupted in radix 2^P, two strings that

are identical except for a transposition of two adjacent characters will hash to the same value?

Good choice of M:—

- A prime number not 2 close to an exact power of 2. (M=p)

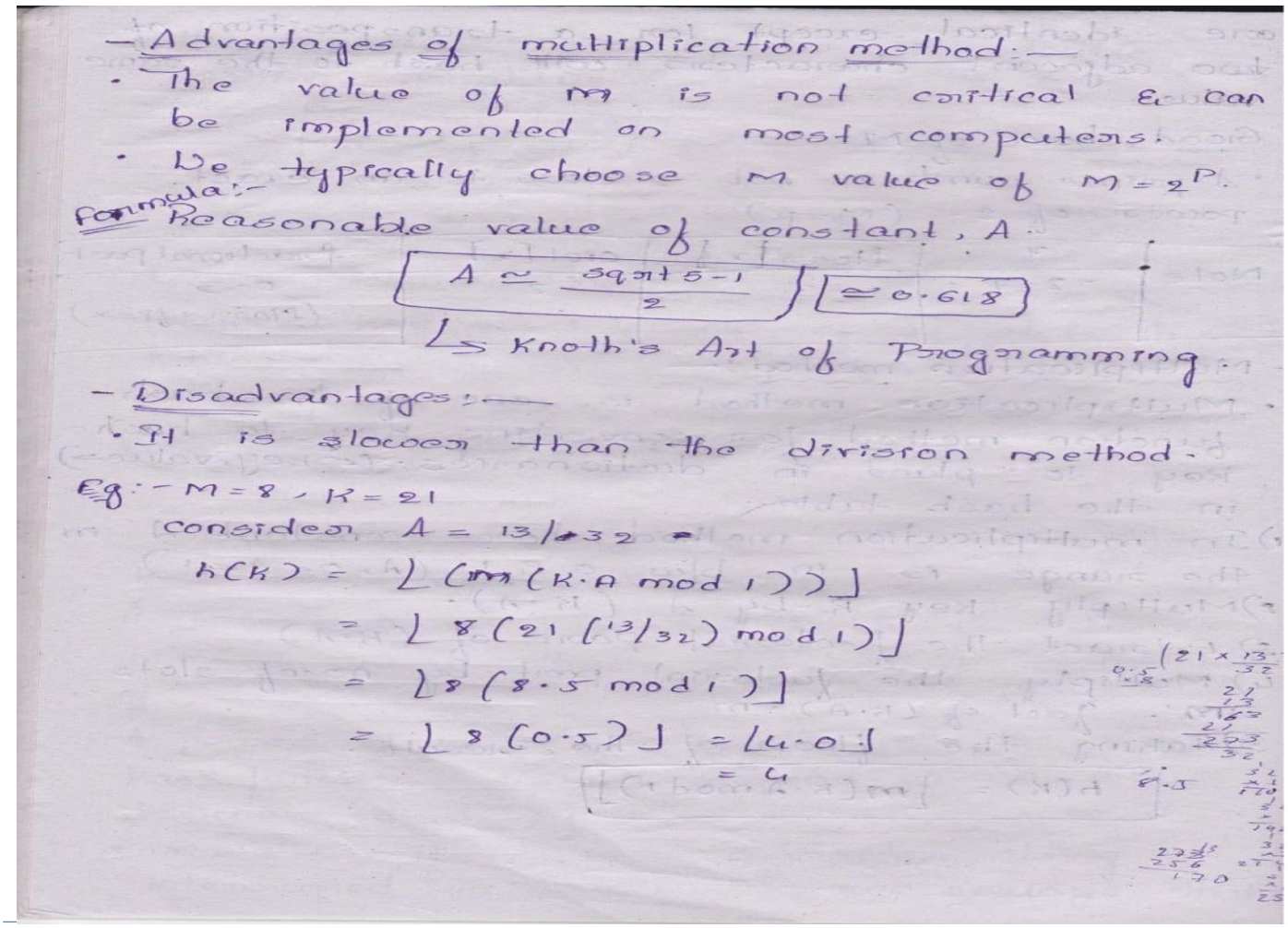
Note	x	floor [x]	ciel [x]	fractional part
	-2.7	-3	-2	0.3 (floor + fp = x)

- Multiplication method:—

- Multiplication method is one of the hash function method for converting key to hash-key to place in dictionaries. (<key, value>) in the hash table.

1) In multiplication method, choose constant 'A' in the range is in b/w 0 & 1. (A, 0<A<1)
2) Multiply key K by A (K·A).
3) Extract the factorial part of (K·A)
4) Multiply the factorial part by no. of slots 'm'. fact of (K·A)·m.
5) Taking the floor of the result.

$$h(K) = \lfloor m(K \cdot A \bmod 1) \rfloor$$



Division Method

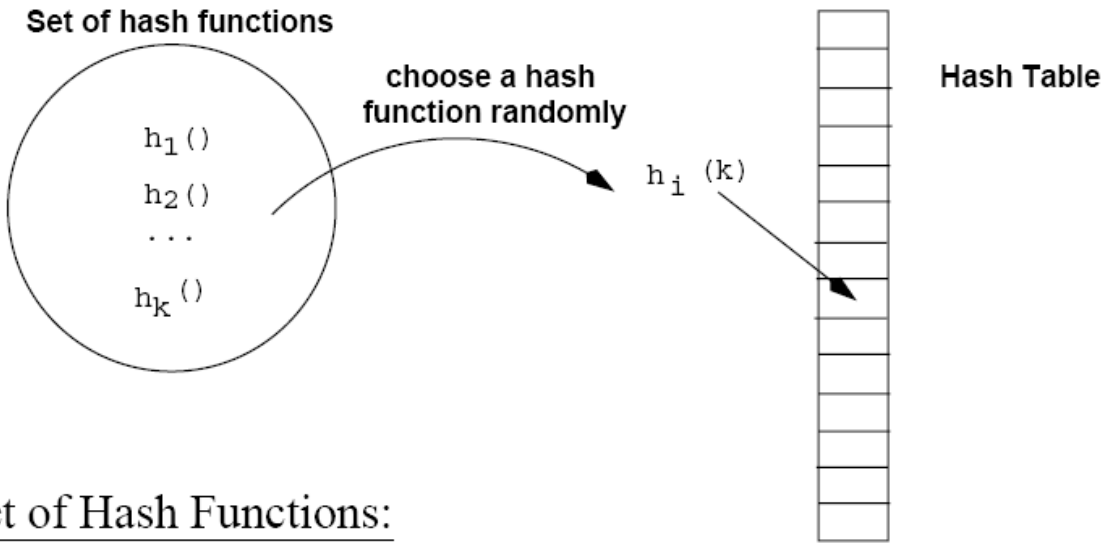
- **Idea:**
  - Map a key k into one of the m slots by taking the remainder of k divided by m
    h(k) = k mod m
- **Advantage**:
  - fast, requires only one operation
- **Disadvantage**:
  - Certain values of m are bad, e.g.,
    - power of 2,
    - non-prime numbers

Multiplicative method

Idea:

- Multiply key k by a constant A, where $0 < A < 1$
- Extract the fractional part of kA
- Multiply the fractional part by m
- Take the floor of the result
  h(k) = $= \lfloor m\ (k\ A \bmod 1) \rfloor$
- **Disadvantage:** Slower than division method
- **Advantage:** Value of m is not critical, e.g., typically 2^p

Universal Hashing



rsal Set of Hash Functions:

→ Universal hash method:—

- This involves choosing a HF randomly in a way that is independent of keys that are actually going to be stored.
- We select the HF at random from a carefully designed class of functions.
- Map key from universe U into m-buckets.
- Let ϕ be the finite collection of HF's that map a given universe U of keys in the range (0, 1, ... (m-1)).
- ϕ is called Universal if for each pair of distinct keys $x, y \in U$, the no. of HF's $h \in \phi$ for which $h(x) = h(y) = \frac{|\phi|}{m}$ (collision).
- With a function randomly choosen by the chance of collision b/w x & $y = 1/m$.

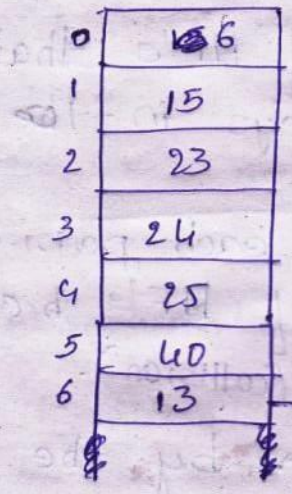
→ Hash table re-structuring or Re-hashing:—

When a hash table has a large no. of entries ((i.e., $n \geq 2m$) in open hashtable, ($n \geq 0.9m$) in closed hash table); the average time for operations will start taking too long time & insertion may be failed. In such cases one idea is to simply create a new hash table with more no. of ~~buckets~~ buckets (say twice or any appropriate large number). In this situation the currently existing elements will

have to be inserted into new hash table. This is called Re-hashing of all these key-values.

By this the effect will be less than for operations & insertion of new element.

Eg:- 13, 15, 24 & 6, 23, 25, 40 (table-size = 7)



$13 \% 7 = 6$
$15 \% 7 = 1$
$24 \% 7 = 3$
$6 \% 7 = 6$
— collision.

$23 \% 7 = 2$, $25 \% 7 = 4$, $40 \% 7 = 5$

$n \geq 0.9M$ $(n = 4, M = 7)$

$4 \geq 0.9(7)$

$7 \geq 6.3$.

$13 \% 16 = 13$
$15 \% 16 = 15$
$24 \% 16 = 8$
$6 \% 16 = 6$
$23 \% 16 = 7$
$25 \% 16 = 9$
$40 \% 16 = 8$

Runtime of hashing

- the load factor λ is the fraction of the table that is full
- λ = 0 (empty) λ = 0.5 (half full) λ = 1 (full table)

Linear probing:

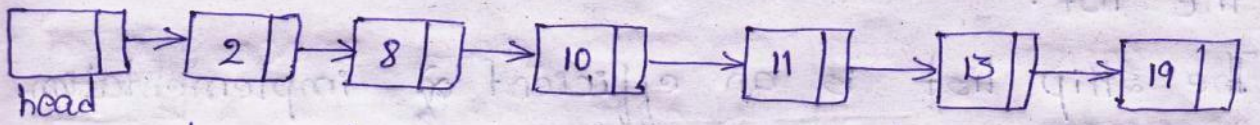
- If hash function is fair and λ < 0.5 - 0.6, then hashtable
- operations are all O(1)

Double hashing:

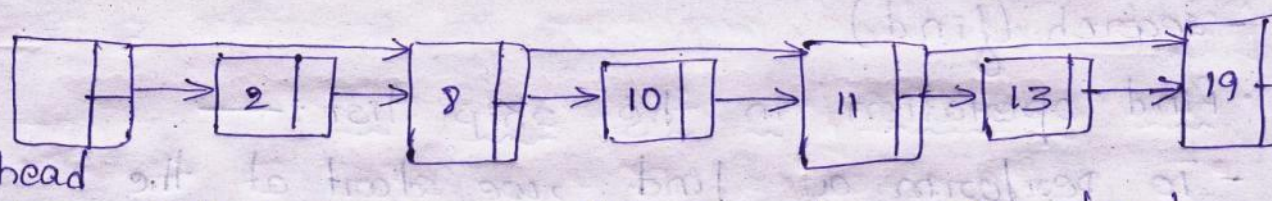
- If hash function is fair and λ < 0.9 - 0.95, then hashtable
- operations are all O(1)

→ Skip list:—

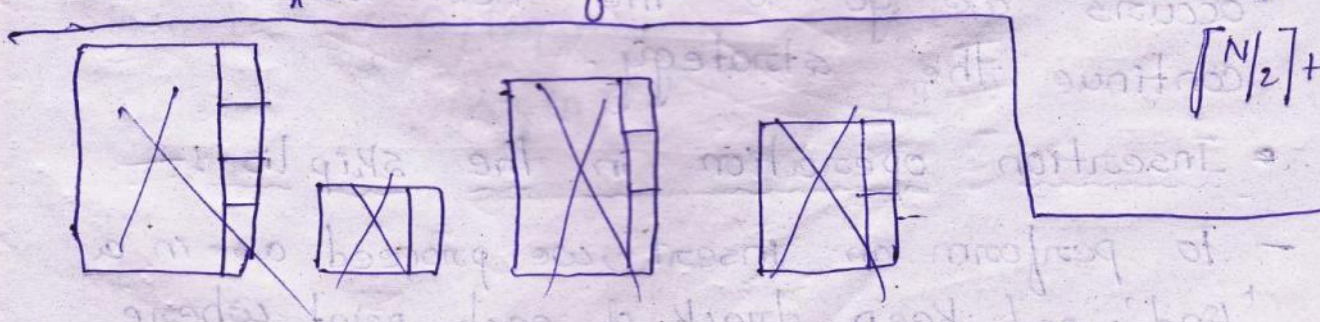
- A skip list is a data structure that allows fast search within an ordered sequence of elements.
- Fast search is made possible by maintaining a linked heirarchy of sub sequence. Each skipping over fewer elements.
- The simplest possible data structure to suppor searching is linked list

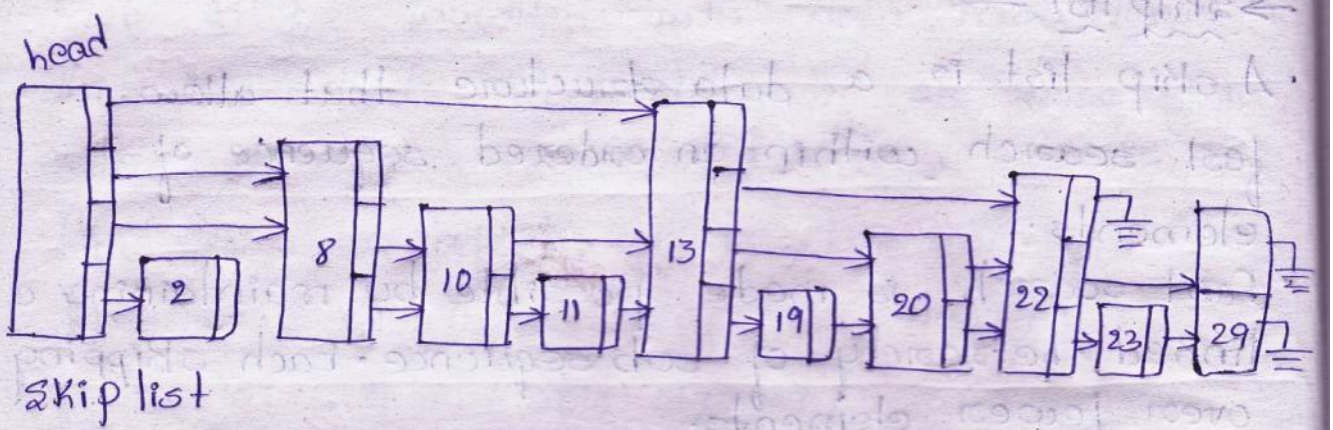


- The time to perform a search is proportion to (α) the no. of nodes that have to be examined, which atmost n.



- Linked list with pointer to two cells ahead.
- Above diagram shows a linked list in which every other node has an additional pointer to the node two ahead of it in the list.



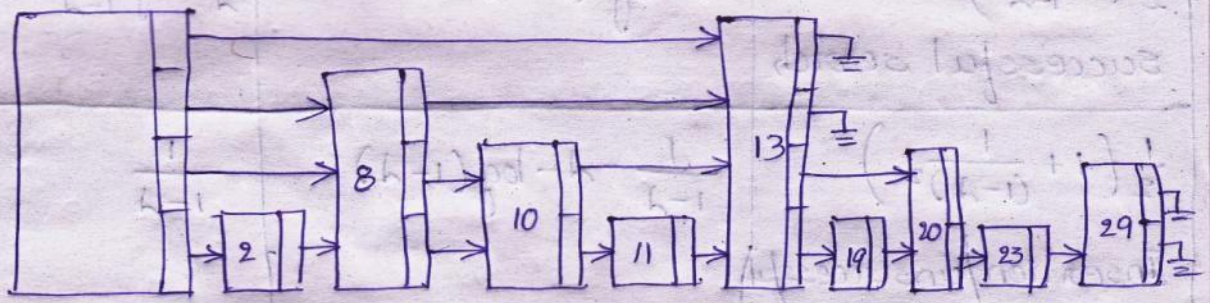


Skip list

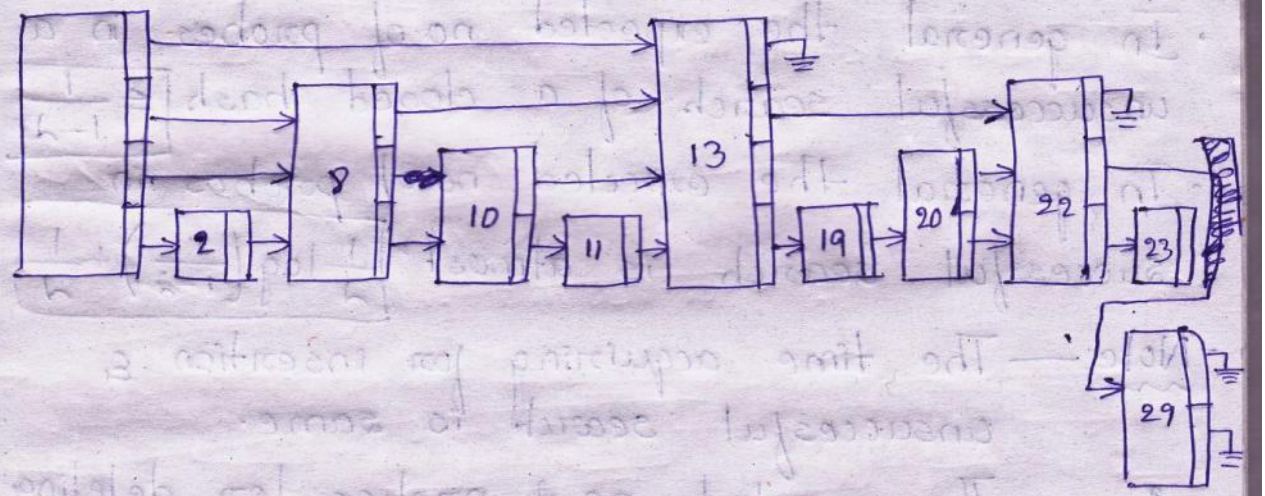
- Skip list has 2 special nodes, one is head node which is the starting node of the list. next one is tail node which is the last node of the list.
- The skip list is an efficient implementation of dictionaries.
- The main operations of data structures is
  - insertion
  - search (find)
- Find operation in the skip list:—
  - To perform a find, we start at the highest pointer at the head we traverse along this level until we find the next node larger than one we are looking, when this occurs we go to the next lower level & continue the strategy.
- Insertion operation in the skip list:—
  - To perform an insert, we proceed as in a 'find'. and keep track of each point where

the switch to a lower level.

- The new node whose level is determined randomly is then inserted into the list.



insert 22



- The time complexity for search, insert, delete is $O[\log(n)]$.

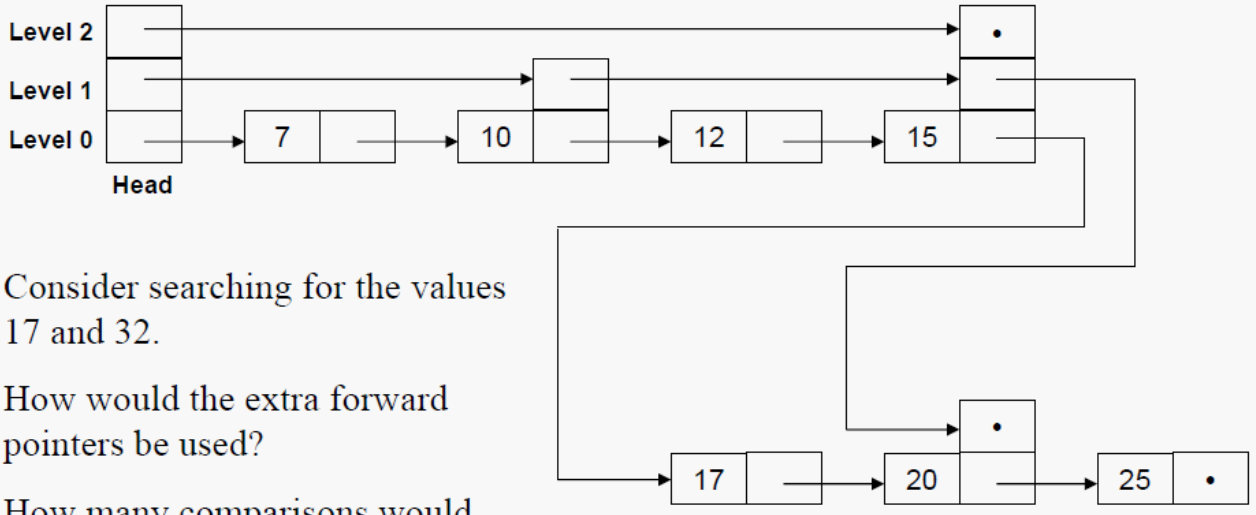
	Avg	coonst
• Search	$O[\log(n)]$	$O(n)$
insert	$O[\log(n)]$	$O(n)$
Delete	$O[\log(n)]$	$O(n)$

The Skip List Concept

Linear linked structures are relatively simple to implement, and well-understood.

We can improve search costs by adding some additional pointers to selected nodes to allow "skipping" over nodes that can safely be ignored.

Consider:



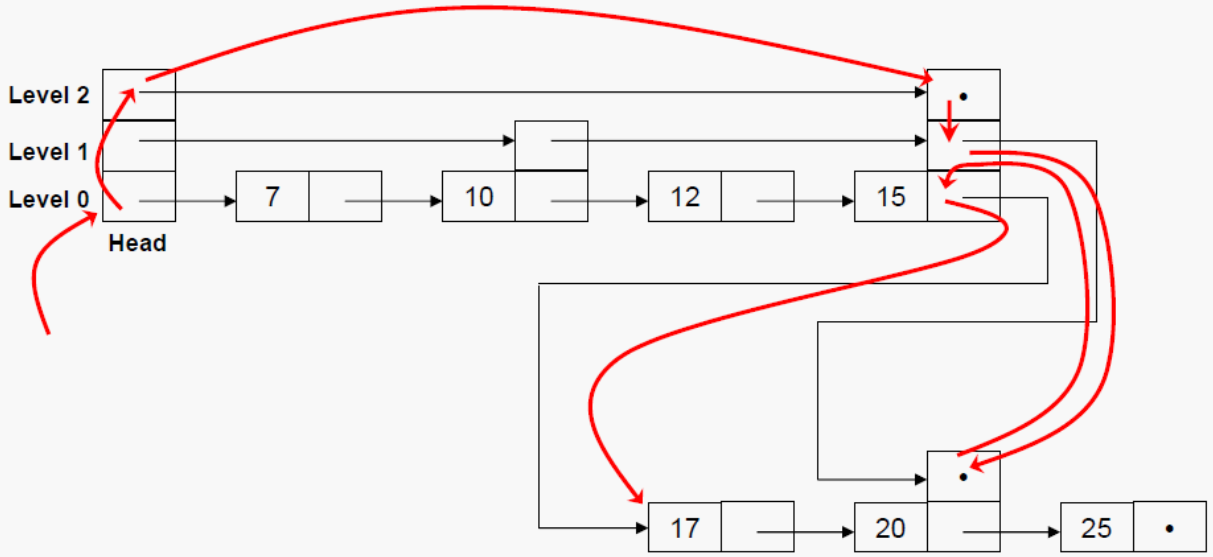
Consider searching for the values 17 and 32.

How would the extra forward pointers be used?

How many comparisons would be required?

Search Example

Consider searching for the value 17:



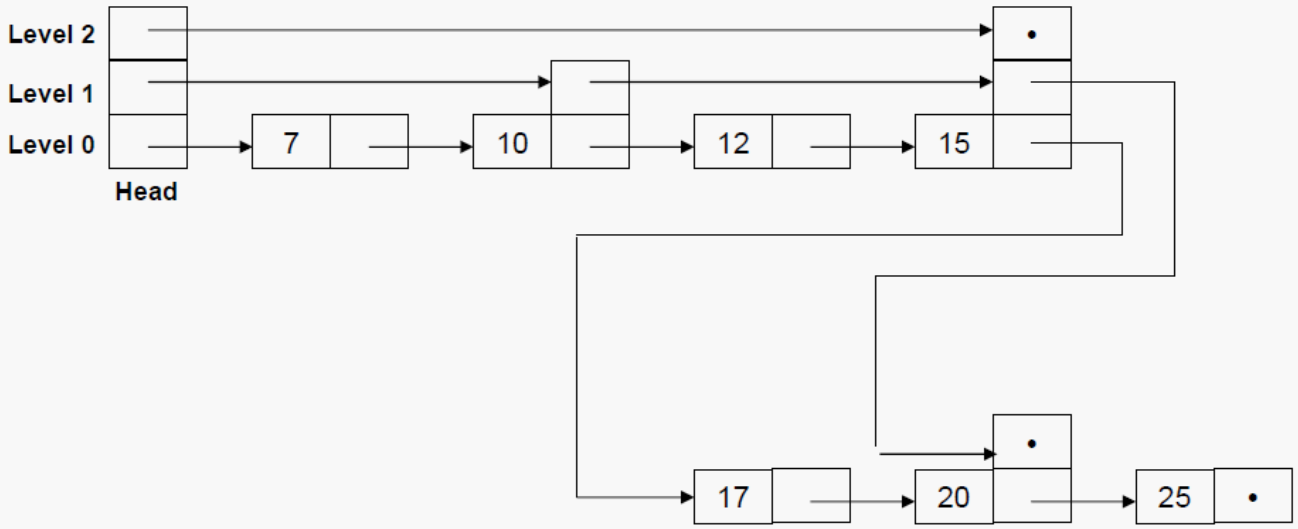
By using the extra pointers, we can jump over the first half of the list, and then determine that the value does not lie within the fourth quarter of the list. A careful count of operations doesn't show any advantage for this tiny list, but…

Terminology

We can view the list as consisting of a hierarchy of parallel, intersecting sub-lists or levels.

Level 1 is a subset of level 0, level 2 a subset of level 1, and so forth.

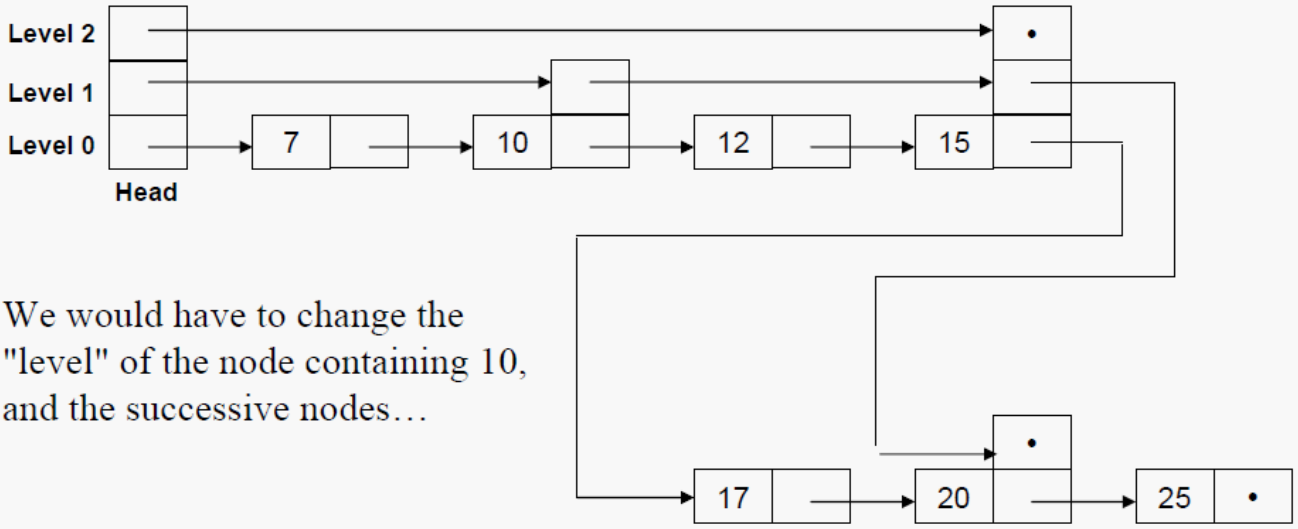
By convention, we say each node belongs to level K-1 if it contains K forward pointers.



Modifying the List

Unfortunately, inserting new nodes into this "ideal" skip list structure is very expensive.

Consider inserting the value 8:



We would have to change the "level" of the node containing 10, and the successive nodes…

… must the structure be so regular in order to provide improved performance?

In the "ideal" skip list:

- 1/2 the nodes are in level 0, 1/4 in level 1, 1/8 in level 2, and so forth.
- nodes in level 0 point to the next node; nodes in level 1 to the next and second-next node; nodes in level 2 to the next, second-next and fourth-next nodes; and so forth
- the level of a node is determined entirely by its position in the list

We can reduce the cost of insertion by:

- selecting a random level for the new node, so that the proportion of level 0, level 1, etc., nodes is roughly preserved
- level 0 nodes point to the next node;

  level 1 nodes also point to the next node that belongs to level 1 or higher;

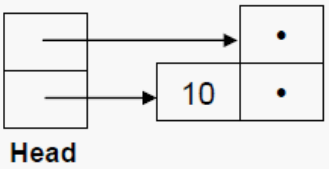
  level 2 nodes also point to the next node that belongs to level 2 or higher;

  and so forth

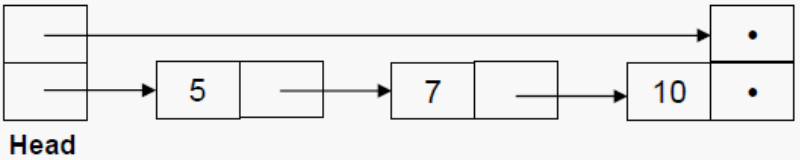
Insertion Example

Empty list configured to provide 1 level:

Insert the value 10; assume that level 1 is selected:

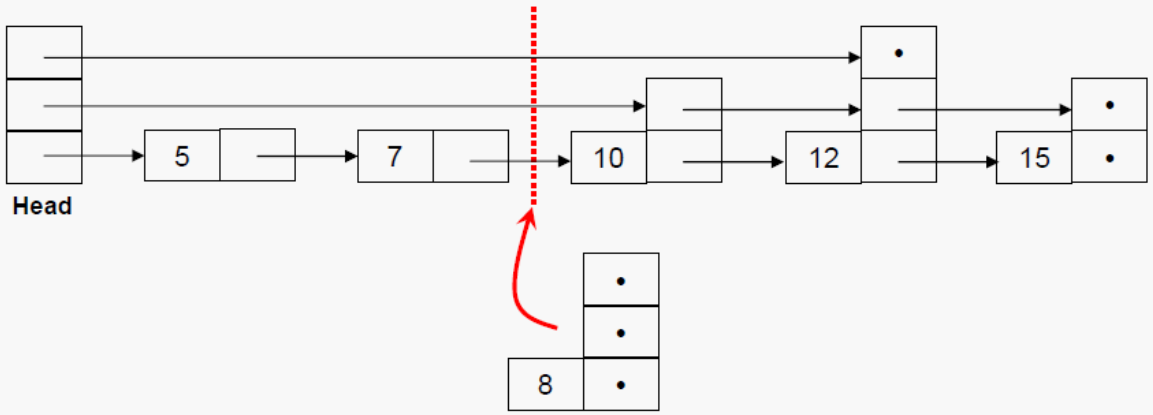


Insert the value 5 at level 0 and then insert the value 7 at level 0:



Insertion Details

Consider inserting the value 8 into the skip list below, and assume that the new node is assigned to level 2:



The first step is to search for the largest value already in the list that is less than or equal to the new value. The new node will become the level-0 successor of that existing node.

To complete the insertion, we must modify pointers to and from the new node.

But which pointers need to be modified?

Precisely the pointers that break the dashed line and are at a level containing the new node.

Insertion Algorithm

Given a data value (that may or may not occur in the list):

```
Node* Update[MaximumLevel]        ; for remembering "pass" nodes
Node* x := &HeadNode;

for i := MaximumLevel downto 0 do

   ; go as far as possible on current level
   while x->forward[i]->keyField < searchKey do
      x := x->forward[i]
   endwhile

   ; search level will now change
   Update[i] := x                 ; remember node for updating later
   ; drop to previous level (via for loop)
endfor

x := x->forward[0]                ; predecessor is in next Node
. . .
```

Insertion Algorithm

The first part of the algorithm has found the predecessor node, now it's time for the physical insertion:

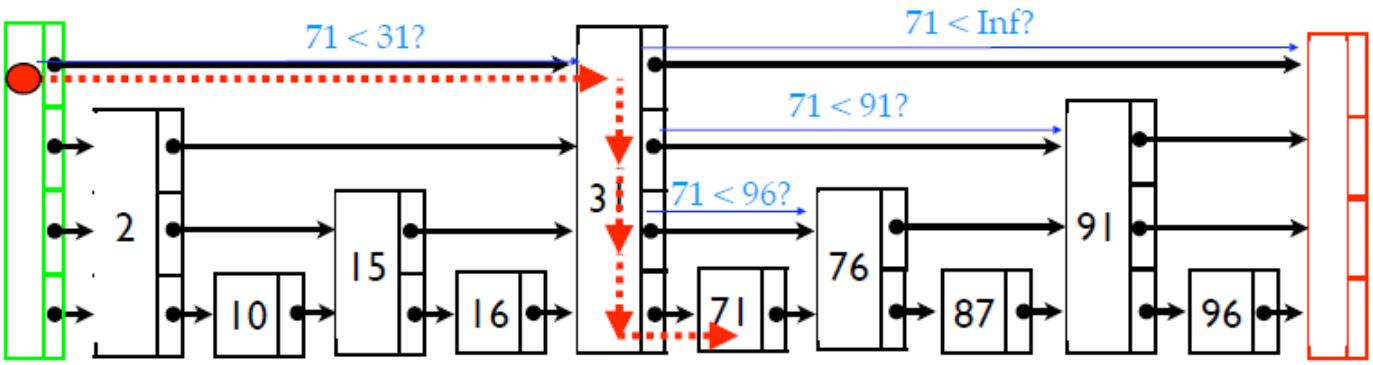
```
. . .
if x->keyField = searchKey then
   ; take action for case of duplicate key value
else
   Lvl := randomLevel()      ; select random level for new node
   if Lvl > MaximumLevel then  ; adjust if list needs new level
      for i := MaximumLevel + 1 to Lvl do
         Update[i] := &HeadNode
      endfor
      MaximumLevel := Lvl
   endif
   ; create new node
   x := makeNode(Lvl, searchKey, Data)
   ; patch it into the list, updating "pass" pointers
   for i := 0 to MaximumLevel do
      x->forward[i] := Update[i]
      Update[i]->forward[i] := x
   endfor
endif
```

Perfect Skip Lists, continued

Comparison

Change current location

Find 71



When search for k:

If k = key, done!

If k < next key, go down a level

If k ≥ next key, go right

Search Algorithm

Given a data value (that may or may not occur in the list):

```
Node* x := &HeadNode;
for i := MaximumLevel downto 0 do
   ; go as far as possible on current level
   while x->forward[i]->keyField < searchKey do
      x := x->forward[i]
   endwhile
   ; drop to previous level (via for loop)
endfor
x := x->forward[0]     ; step to next Node
if x->keyField = searchKey then
   return x->Data
else
   return failure
endif
```

Skip List Analysis

- Expected number of levels = O(log n)
  - E[# nodes at level 1] = n/2
  - E[# nodes at level 2] = n/4
  - ...
  - E[# nodes at level log n] = 1

Rehashing

- Hash Table may get full
  - No more insertions possible
- Hash table may get *too* full
  - Insertions, deletions, search take longer time
- Solution: Rehash
  - Build another table that is twice as big and has a new hash function
  - Move all elements from smaller table to bigger table
- Cost of Rehashing = *O(N)*
  - But happens only when table is close to full
  - Close to full = table is X percent full, where X is a tunable parameter

Draw the result of hashing 19, 50, 89, 39 using quadratic hashing with mod 5. REHASH if necessary

0	50
1	
2	
3	89
4	19

0	
1	
2	
3	
4	
5	
6	
7	
8	
9	
10	

RE-Hashing

If you come across an element that won't fit using the type of Hashing you are doing, you must re-hash!
Make a second table that is 2x the size of the original and then some.

You after multiplying by 2, keep incrementing by 1 until you reach a prime number. Indices from 0 to prime-1.

You now will be modding the new numbers by the prime you arrived at.
Your order for adding the numbers relies on you starting with the numbers you inserted in the first table!
If you have a collision try the cell right below the collision. If that isn't free try the one right above the collision!

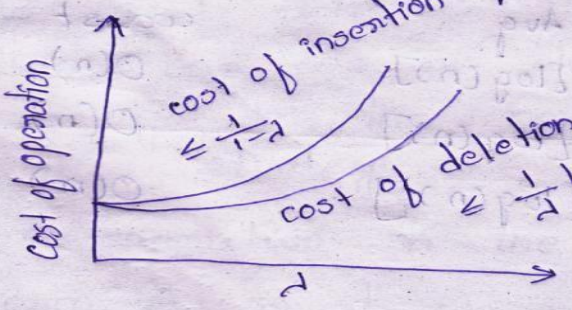
- Analysis of closed hashing:— $\lambda = n/k$

Linear probing	Quadratic probing	Double hashing
$\frac{1}{2}\left(1+\frac{1}{1-\lambda}\right)$ successful search	$1-\log(1-\lambda)-\lambda/2$	$\frac{1}{\lambda}\log\frac{1}{1-\lambda}$
$\frac{1}{2}\left(1+\frac{1}{(1-\lambda)^2}\right)$ Insertion/unsuccessful search	$\frac{1}{1-\lambda}-\lambda-\log(1-\lambda)$	$\frac{1}{1-\lambda}$

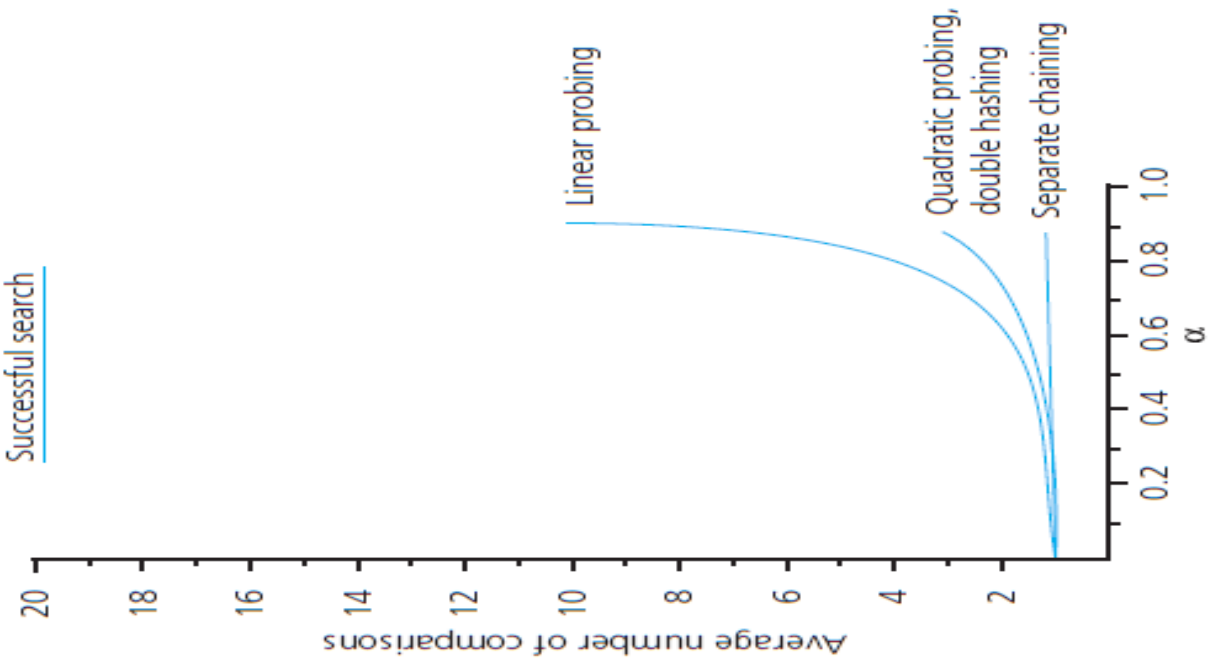
- In general the expected no. of probes in a unsuccessful search of a closed hash $\leq \frac{1}{1-\lambda}$
- In general the expected no. of probes in successful search is atmost $\frac{1}{\lambda}\log\left(\frac{1}{1-\lambda}\right)+\frac{1}{\lambda}$

Note:— The time acquiring for insertion & unsuccesful search is same.

- The expected no. of probes for deleting an element in closed hashing is $\leq \frac{1}{\lambda}\log\left(\frac{1}{1-\lambda}\right)+\frac{1}{\lambda}$



Efficiency of Hashing


Successful search
Linear probing
Quadratic probing,
double hashing
Separate chaining
Average number of comparisons
20
18
16
14
12
10
8
6
4
2
0.2
0.4
0.6
0.8
1.0
α