JEPPIAAR INSTITUTE OF TECHNOLOGY

“Self Belief | Self Discipline | Self Respect”

DEPARTMENT OF

COMPUTER SCIENCE AND ENGINEERING

LECTURE NOTES

CS8391 / DATA STRUCTURES

(2017 Regulation)

Year/Semester: II / 03

Prepared by

Dr. K. Tamilarasi,

Professor / Dept. of CSE.

SYLLABUS

CS8391 DATA STRUCTURES L T P C 3 0 0 3

UNIT I LINEAR DATA STRUCTURES – LIST 9

Abstract Data Types (ADTs) – List ADT – array-based implementation – linked list implementation ––singly linked lists- circularly linked lists- doubly-linked lists – applications of lists –Polynomial Manipulation – All operations (Insertion, Deletion, Merge, Traversal).

UNIT II LINEAR DATA STRUCTURES – STACKS, QUEUES 9

Stack ADT – Operations - Applications - Evaluating arithmetic expressions- Conversion of Infix to postfix expression - Queue ADT – Operations - Circular Queue – Priority Queue - deQueue – applications of queues.

UNIT III NON LINEAR DATA STRUCTURES – TREES 9

Tree ADT – tree traversals - Binary Tree ADT – expression trees – applications of trees – binary search tree ADT –Threaded Binary Trees- AVL Trees – B-Tree - B+ Tree - Heap – Applications of heap.

UNIT IV NON LINEAR DATA STRUCTURES - GRAPHS 9

Definition – Representation of Graph – Types of graph - Breadth-first traversal - Depth-first traversal – Topological Sort – Bi-connectivity – Cut vertex – Euler circuits – Applications of graphs.

UNIT V SEARCHING, SORTING AND HASHING TECHNIQUES 9

Searching- Linear Search - Binary Search. Sorting - Bubble sort - Selection sort - Insertion sort - Shell sort – Radix sort. Hashing- Hash Functions – Separate Chaining – Open Addressing – Rehashing – Extendible Hashing.

TOTAL: 45 PERIODS

TEXT BOOKS:

1. Mark Allen Weiss, “Data Structures and Algorithm Analysis in C”, 2nd Edition, Pearson Education,1997.
2. ReemaThareja, “Data Structures Using C”, Second Edition , Oxford University Press, 2011

REFERENCES:

1. Thomas H. Cormen, Charles E. Leiserson, Ronald L.Rivest, Clifford Stein, “Introduction to Algorithms", Second Edition, Mcgraw Hill, 2002.
2. Aho, Hopcroft and Ullman, “Data Structures and Algorithms”, Pearson Education,1983.
3. Stephen G. Kochan, “Programming in C”, 3rd edition, Pearson Education.

UNIT - V

SEARCHING, SORTING & HASHING TECHNIQUES

Searching - Linear Search, Binary Search. Sorting - Bubble sort, Selection sort, Insertion sort, Shell sort, Radix sort. Hashing - Hash functions, Separate Chaining, Open Addressing, Rehashing, Extendible Hashing.

* ——— * ——— * ——— * ——— * ——— *

Searching:-

Searching means to find whether a particular value is present in an array or not. If the value is present in the array, then searching is said to be successful and the searching process gives the location of that value in the array.

There are (2) popular methods for searching the array elements:
- Linear Search
- Binary Search.

1) Linear Search:-

Linear search, also called as sequential search, is a very simple method used for searching an array for a particular value.

* It works by comparing the value to be searched with every element of the array one by one in a sequence until a match is found.

* Linear search is mostly used in searching an unordered list of elements (array in which data elements are not sorted).

* For example, if an array A[] is declared and initialized as,

int A[] = {10, 8, 2, 7, 3, 4, 9, 1, 6, 5};

and the value to be searched is VAL = 7., then searching means to find whether the value '7' is present in the array or not.

* If it is present, then it returns the position of its occurrence. Here POS = 3. (index starts from 0).

Algorithm:

```
LINEAR_SEARCH (A, N, VAL)
Step 1: [INITIALIZE] SET POS = -1
Step 2: [INITIALIZE] SET I = 1
Step 3:     Repeat Step 4 while I <= N
Step 4:         IF A[I] = VAL
                    SET POS = 1
                    PRINT POS
                    GO TO STEP 6
                [END IF]
                SET I = I+1
            [END OF LOOP]
Step 5: IF POS = -1
            PRINT "VALUE IS NOT PRESENT"
        [END IF]
Step 6: EXIT
```

Program Code:

```
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
```

```
# define size 20
int main (int argc, char * argv[])
{
    int arr[size], num, i, n, found = 0, pos = -1;
    printf("\n Enter the number of elements in the array:");
    scanf("%d", &n);
    printf("\n Enter the elements: ");
    for (i=0; i<n; i++)
        scanf("%d", &arr[i]);
    printf("\n Enter the number that has to be searched:");
    scanf("%d", &num);
    for (i=0; i<n; i++)
    {   if (arr[i] == num)
        {
            found = 1;
            pos = i;
            printf("\n %d is found in the array at
                        position = %d", num, i+1);
            break;
        }
    }
    if (found == 0)
        printf("\n %d is not present in the array", num);
    return 0;
}
```

<u>Time Complexity</u>:-

* Linear Search executes in <u>O(n)</u> time, where n is the no. of elements in the array. This is for the worst case.
* In best case, the linear search executes in <u>O(1)</u>, where the element present at the first position.

2) Binary Search:-

Binary search is a searching alg that works efficiently with a sorted list.

* The mechanism of binary search can be better understood by an analogy of a telephone directory.
* When we are searching for a particular name in a directory, we first open the directory from the middle and then decide whether to look for the name in the first part of the directory or in the second part of the directory.
* Again, we open some page in the middle and the whole process is repeated until we finally find the right name.

Now, let us consider how this mechanism is applied to search for a value in a sorted array. Consider an array A[ ] that is declared and initialized as,

int A[ ] = { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10 };

and the value to be searched is VAL = 9.

* The alg will proceed in the following manner.

0	1	2	3	4	5	6	7	8	9	10
0	1	2	3	4	5	6	7	8	9	10

BEG = 0, END = 10, MID = (0+10)/2 = 5

* Now VAL = 9. and A[MID] = A[5] = 5.

* A[5] is less than VAL, therefore, we now search for the value in the second half of the array. So we change the values of BEG and MID.

Now BEG = MID+1 = 6, END = 10, MID = (6+10)/2 = 8

VAL = 9 and A[MID] = A[8] = 8.

A[8] is less than VAL, therefore, we now search for the value in the second half of the segment. So again we change the values of BEG and MID.

Now BEG = MID+1 = 9, END = 10, MID = (9+10)/2 = 9

Now VAL = 9 and A[MID] = A[9] = 9.

In this alg; we see that BEG and END are the beginning and ending positions of the segment that we are looking to search for the element.

* MID is calculated as (BEG + END)/2.
* Initially BEG = lower_bound and END = upper_bound.
* The alg will terminate when A[MID] = VAL.
* When the alg ends, we will set POS = MID.
* POS is the position at which the value is present in the array.

<u>Algorithm:</u>

BINARY_SEARCH(A, lower_bound, upper_bound, VAL)

Step 1: [INITIALIZE] SET BEG = lower_bound

END = upper_bound, POS = -1

Step 2: Repeat Steps 3 and 4 while BEG <= END

Step 3: SET MID = (BEG + END)/2

Step 4: IF A[MID] = VAL

SET POS = MID

PRINT POS

Go to Step 6

```
        ELSE IF A[MID] > VAL
            SET END = MID -1
        ELSE
            SET BEG = MID+1
        [END OF IF]
    [END OF LOOP]
Step 5: IF POS = -1
            PRINT "VALUE IS NOT PRESENT"
        [END OF IF]
Step 6: EXIT.
```

Complexity:

* The complexity of binary search alg can be expressed as f(n), where n is the no. of elements in an array.
* The complexity of the alg is calculated as the no. of comparisons are made.
* In binary search, the size of the segment of the search is reduced by half.
* Therefore, the total no. of comparisons will be

$$2^{f(n)} > n \quad \text{or} \quad f(n) = \log_2(n).$$

* The time complexity of this alg is $O(\log_2 n)$ in worst case.

Pgm Code:

```
/* Search an element in an array using binary search */
#include <stdio.h>
#include <stdlib.h>
#include <conio.h>
#define size 10
```

```
int smallest (int arr[], int k, int n);
void selection_sort ( int arr[ ] , int n);
int main()
{
    int arr [size ], num, i, n, beg, end, mid, found=0;
    printf(" \n Entee the no. of elements in the array:");
    scanf("%d", & n);
    printf(" \n Entee the elements:");
    for(i=0; i<n; i++)
        scanf ("%d", & arr[i]);
    selection_sort (arr, n);
    printf("\n The sorted array is :\n");
    for(i=0; i<n; i++)
        printf ("%d\t, arr[i]);
    printf(" \n\n Enter the no that has to be searched:");
    scanf ("%d",& num);
    beg=0, end=n-1;
    while (beg <= end)
    {
        mid= (beg+end)/2;
        if (arr[ mid ]==num)
        {   printf("\n %d is present in the array at
                        position %d", num, mid+1);

            found=1;
            break;
        }
        else if (arr[mid]>num)
        end=mid-1;
        else
        beg= mid+1;
    }
```

```
    if (beg > end && found ==0)
        printf("\n %d does not found in the array", num);
  return 0;
}
int smallest (int arr[], int k, int n)
{
    int pos =k, small= arr[k], i;
    for( i=k+1; i<n; i++)
    {
        if (arr[i] < small)
        {   small = arr[i];
            pos =i;
        }
    }
    return pos;
}
void selection_sort( int arr[], int n)
{
    int k, pos, temp;
    for(k=0; k<n; k++)
    {
        pos = smallest (arr, k, n);
        temp = arr[k];
        arr[k]= arr[pos];
        arr[pos] = temp;
    }
}
```

SORTING :-

Sorting means arranging the elements of an array so that they are placed in some relevant order which may be either ascending or descending.

A sorting alg is defined as an alg that puts the elements of a list in a certain order, which can be either numerical order, lexicographical order, or any user-defined order.

* Efficient sorting algs are widely used to optimize the use of other algs like search and merge algs which require sorted lists to work correctly.

2 types of sorting
- Internal sorting
- External sorting.

* Internal sorting which deals with sorting the data stored in the computer's memory.
* External sorting which deals with sorting the data stored in files. It is applied when there is voluminous data that cannot be stored in the memory.

1) Bubble Sort :-

Bubble sort is a very simple method that sorts the array elements by repeatedly moving the largest element to the highest index position of the array segment.

* In bubble sorting, consecutive adjacent pairs of elements in the array are compared with each other.
* If the element at the lower index is greater than the element in the higher index, the two elements are interchanged so that the element is placed before the bigger one.
* This process will continue till the list of unsorted elements exhausts.

Alg :-

BUBBLE_SORT (A, N)

Step 1: Repeat Step 2 for i = 0 to N-1

Step 2: Repeat for j = 0 to N-1

Step 3: If A[j] > A[j+1]

swap A[j] and A[j+1]

[END OF INNER LOOP]

[END OF OUTER LOOP]

Step 4: EXIT

In this alg, the outer loop is the total no. of passes which is N-1.

* In this alg, the inner loop will be executed for every pass.

* However, the frequency of the inner loop will decrease with every pass because after every pass, one element will be in its correct position.

* Therefore, for every pass, the inner loop will be executed N-I times, where N is the no. of elements and I is the counter of the pass.

Complexity :-

The complexity of any sorting alg depends upon the no. of comparisons.

* In bubble sort, we have N-1 passes in total.

* In the first pass, N-1 comparisons are made to place the highest element in its correct position.

* Then, in Pass 2, there are N-2 comparisons and the second highest element is placed in its position.

* Therefore to compute the complexity of bubble sort,

$$f(n) = (n-1) + (n-2) + (n-3) + \dots + 3 + 2 + 1$$

$$f(n) = n(n-1)/2 = n^2/2 - n/2$$

$$\therefore f(n) \in O(n^2).$$

* The time complexity of this alg is $O(n^2)$.

Example:

Sort 95, 65, 51, 2, 89 using bubble sort.

Pass 1:

95 65 51 2 89 → Compare 95 and 65

65 95 51 2 89 → Compare 95 and 51

65 51 95 2 89 → Compare 95 and 2

65 51 2 95 89 → Compare 95 and 89

65 51 2 89 95 → Now 95 has reached its position.

Pass 2:

65 51 2 89 95 → Compare 65 and 51

51 65 2 89 95 → Compare 65 and 2

51 2 65 89 95 → Compare 65 and 89

51 2 65 89 95 → Compare 89 and 95
Now 89 has reached its position.

Pass 3:

51 2 65 89 95 → Compare 51 and 2.

2 51 65 89 95 → Compare 51 and 65

2 51 65 89 95 → Now 65 has reached its position.

Pass 4:

2 51 65 89 95 → Compare 2 and 51.

Sorted list:

2 51 65 89 95 → Now 51 has reached its position.

Routine:

```
void   bubble_sort (int *arr, int n)
{
  int i, j, temp, flag=0;
  for (i=0; i<n; i++)
  { for (j=0; j<n-i-1; j++){
     if (arr[j] > arr[j+1])
      {
        flag = 1;
        temp = arr[j+1];
        arr[j+1] = arr[j];
        arr[j] = temp;
      }
    }
    if (flag == 0) // array is sorted
       return;
  }
}
```

2) Selection Sort :-

Selection sort is a simple and slow sorting alg that repeatedly selects the lowest or highest element from the unsorted section of the list and moves it to the end of the sorted section.

* It has quadratic running time complexity of $O(n^2)$, thereby, making it inefficient to be used on large lists.

* Selection sort is generally used for sorting files with very large objects (records) and small keys.

Consider an array ARR with N elements.

* First find the smallest value in the array and place it in the first position.

* Then, find the 2nd smallest value in the array and place it in the second position.

* Repeat this procedure until the entire array is sorted.

* In Pass 1, find the position POS of the smallest value in the array and then swap ARR[POS] and ARR[0]. Thus, ARR[0] is sorted.
* In Pass 2, find the position POS of the smallest value in sub-array of N-1 elements. Swap ARR[POS] with ARR[1]. Now ARR[0] and ARR[1] is sorted.
* In Pass N-1, find the position POS of the smaller of the elements ARR[N-2] and ARR[N-1]. Swap ARR[POS] and ARR[N-2] so that ARR[0], ARR[1], ..., ARR[N-1] is sorted.

Example:

Sort the array of elements using selection sort.
39, 9, 81, 45, 90, 27, 72, 18.

PASS	POS	ARR[0]	ARR[1]	ARR[2]	ARR[3]	ARR[4]	ARR[5]	ARR[6]	ARR[7]
1	1	9	39	81	45	90	27	72	18
2	7	9	18	81	45	90	27	72	39
3	5	9	18	27	45	90	81	72	39
4	7	9	18	27	39	90	81	72	45
5	7	9	18	27	39	45	81	72	90
6	6	9	18	27	39	45	72	81	90
7	6	9	18	27	39	45	72	81	90

Routine:

```
void selection_sort (int arr[], int n)
{
   int i, j, temp;
   for (i=0; i<n-1; i++)
   { for (j=i+1; j<n; j++)
     { if (arr[i] > arr[j])
       {
          temp = arr[i];
          arr[i] = arr[j];
          arr[j] = temp;
       }
     }
   }
}
```

Complexity:-

In Pass 1, selecting the element with smallest value calls for scanning all n elements, thus, n-1 comparisons are required in the first pass.

* Then, the smallest value is swapped with the element in the first position.

* In Pass 2, selecting the second smallest value requires scanning the remaining n-1 elements and so on, Therefore

$$f(n) = (n-1)+(n-2)+ \ldots + 3+2+1.$$

$$= n(n-1)/2$$

$$f(n) \in O(n^2).$$

* Thus, the time complexity of this sorting is $O(n^2)$.

2) Insertion Sort :-

Insertion sort is a very simple sorting alg in which the sorted array (or list) is built one element at a time.

* It repeatedly takes the next element from the unsorted section of the list and inserts it into the sorted section at the correct position.

* While playing cards, the new card picked will be inserted into the sorted cards in the hand. This is the principle idea behind the insertion sort.

* Insertion sort is less efficient as compared to other more advanced algs such as quick sort, heap sort & merge sort.

Insertion sort works as follows:

1). The array of values to be sorted is divided into two sets. One that stores sorted values and another that contains unsorted values.

2). The sorting alg will proceed until there are elements in the unsorted set.

3). Initially all the elements are in unsorted section. Take an element from the unsorted section.

4). Insert the element into the sorted section at the correct position based on the comparable property.

5) Repeat step 3 and 4 until no more elements left in the unsorted section.

↳ Assuming there are n elements in the array, the insertion sort must index through n-1 entry.

↳ For each entry, n-1 entries are examined and shifted.

- The insertion sort is an in-place sort i.e, no extra memory is required.
- The insertion sort is also a <u>stable sort</u>. A stable sort retain the original ordering of keys when identical keys are present in the input data.

<u>Alg:-</u>

INSERTION_SORT(ARR, N)

Step 1: Repeat Steps 2 to 5 for k = 1 to N-1

Step 2: Set temp = arr[k]

Step 3: set j = k-1

Step 4: Repeat while temp <= arr[j]

set arr[j+1] = arr[j]

set j = j-1

[END OF INNER LOOP]

Step 5: Set arr[j+1] = temp

[END OF LOOP]

Step 6: EXIT.

<u>Example</u>: Sort using insertion sort: 34, 8, 64, 51, 32, 21.

							Positions Moved
Original	34	8	64	51	32	21	
After pass 1	8	34	64	51	32	21	1
After pass 2	8	34	64	51	32	21	0
After pass 3	8	34	51	64	32	21	1
After pass 4	8	32	34	51	64	21	3
After pass 5	8	21	32	34	51	64	4

Routine:

```
void  insertion_sort (int arr[], int n)
{
  int i,j, temp;
  for (i=1; i<n; i++)
   {
      temp = arr[i];
      j=j-1;
      while ((temp < arr[j]) && (j>=0))
      {
         arr[j+1] = arr[j];
         j--;
      }
      arr[j+1] = temp;
   }
}
```

Complexity:-

Best case: For insert sort, the best case occurs when the array is already sorted. In this case, the running time of the alg has a linear running time $O(n)$. This is because, during each iteration, the first element from the unsorted set is compared only with the last element of the sorted set of the array.

Worst case:

The worst case of the insertion sort occurs when the array is sorted in the reverse order. Here, the first element of the unsorted set has to be compared with almost every element in the sorted set. Furthermore, every iteration of the inner loop will have to shift the elements of the sorted set of the array before inserting the next element. ∴, insertion sort has a quadratic running time, ie, $O(n^2)$.

Avg case:

The insertion sort will have to take at least (k-1)/2 comparisons. Thus, the average case also has a quadratic running time.

4) Shell Sort :-

Shell sort, invented by Donald Shell, is a sorting alg that is a generalization of insertion sort. This is also referred to as diminishing increment sort.

* Shell sort tries to take a list of items to be sorted and make it "nearly sorted". So that a final sorting by insertion can complete the work.
* It has the potential to yield worst case running time better than $O(n^2)$.

The given set of array is divided into number of shells. Each time the element in the shell being compared with elements in other shells and so on.

↳ The initial array is first fragmented into k sections, where k is preferably a prime number.

↳ After the Pass I, the whole array is partially sorted.

↳ The value of k is reduced which increases the no. of segments and reduce the size of the segments in the next pass.

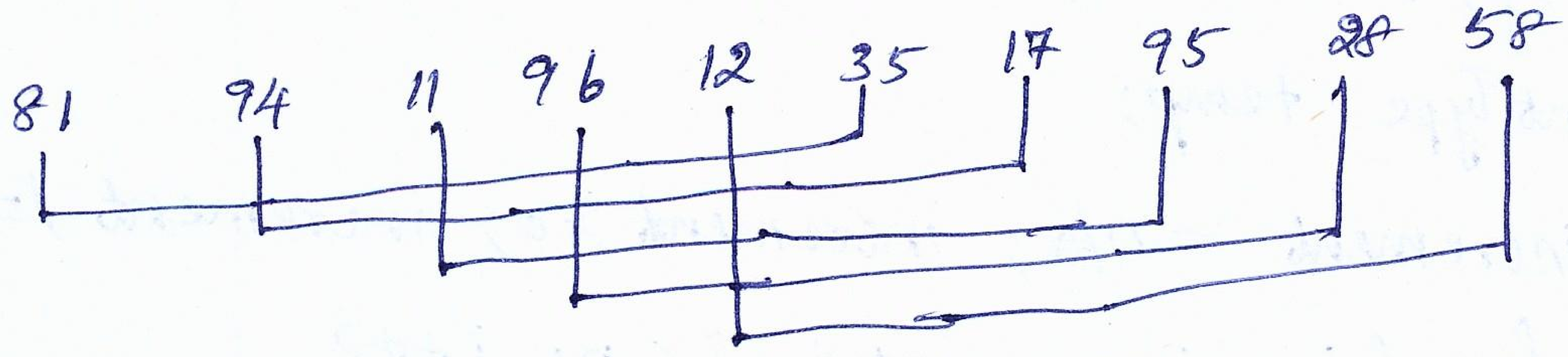
↳ The process is repeated until k=1 at which the array is sorted.

↳ It is called diminishing increment sort because the value of k decrease continuously.

Example:

81, 94, 11, 96, 12, 35, 17, 95, 28, 58.

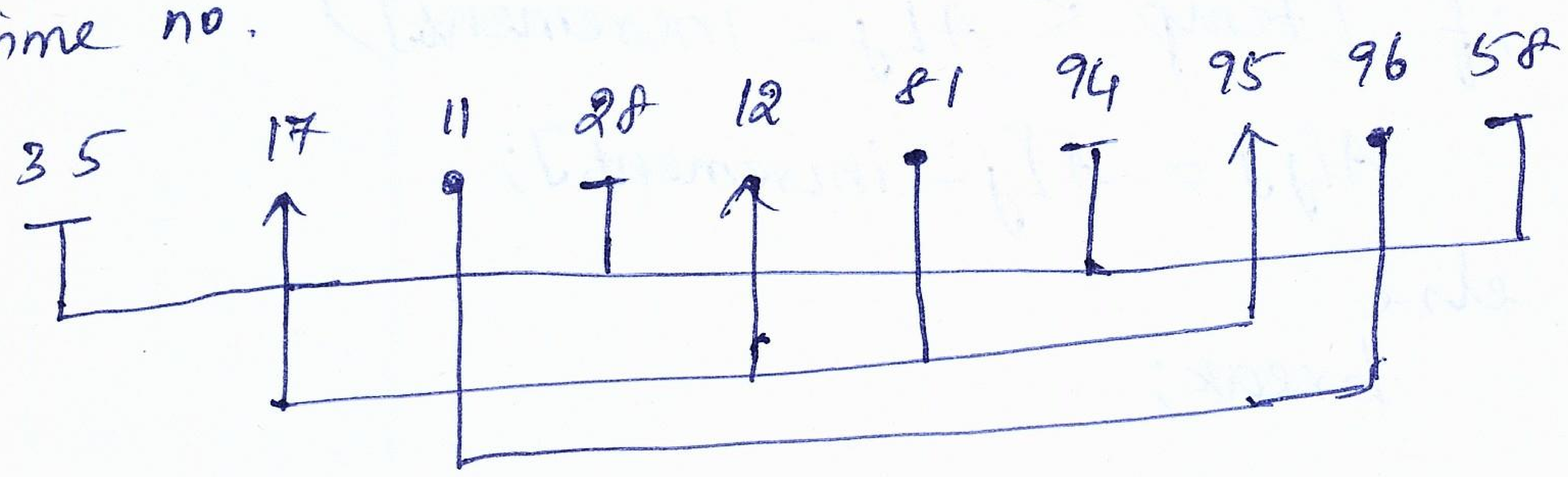
Here $n = 10$, the first pass $k = 10/2 = 5$.



After First Pass

35, 17, 11, 28, 12, 81, 94, 95, 96, 58

In the second pass the k value is reduced to 3, i.e., 3 is a prime no.



35, 28, 94, 58 ⇒ 28, 35, 58, 94
↓ ↓ ↓ ↓
1 4 8 10

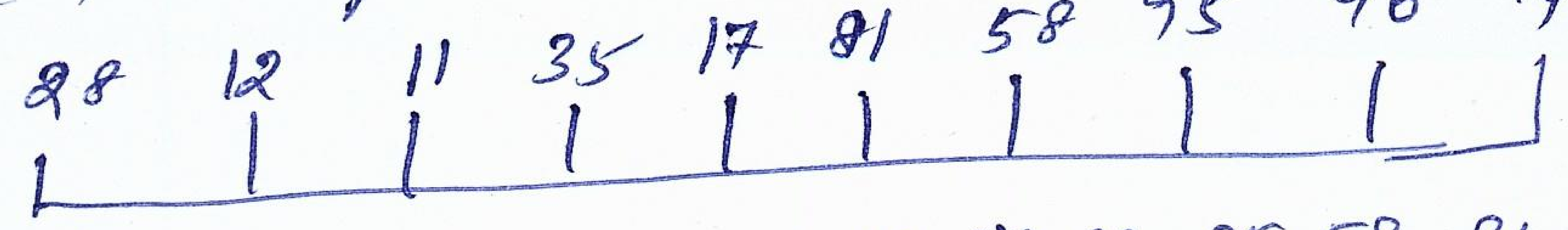
17, 12, 95 ⇒ 12, 17, 95
↓ ↓ ↓
2 5 9

11, 81, 96 ⇒ 11, 81, 96
↓ ↓ ↓
3 6 11

After the second pass,

28, 12, 11, 35, 17, 81, 58, 95, 96, 94

In the third pass k is reduced to 1



The sorted array contains 11, 12, 17, 28, 35, 58, 81, 94, 95, 96.

Routine:

```
void   Shellsort (ElementType A[], int n)
{
   int i, j, increment;
   ElementType temp;
   for (increment = n/2; increment > 0; increment /= 2)
       for (i = increment; i < n; i++)
       {
         temp = A[i];
         for (j = i; j >= increment; j -= increment)
            if (temp < A[j - increment])
                A[j] = A[j - increment];
            else
                break;
         A[j] = temp;
       }
}
```

Complexity:-

Best case:

If appropriate sequence of increments is used, the running time is $O(n \log n)$.

Worst case:

If the increment sequence is not chosen properly, the running time is $O(n^2)$.

5) Radix Sort:-

Radix sort is a linear sorting alg for integers and uses the concept of sorting names in alphabetical order.

* It manages to sort the values without actually performing any comparisons on the input data.

* When we have a list of sorted names, the radix is 26 (or 26 buckets) because there are 26 letters in the English alphabet. This sorting is also known as bucket sort.

Observe that words are first sorted according to the first letter of the name. i.e, the first class stores the names that begin with A, the second class contain the names with B, and so on.

During the 2nd pass, names are grouped according to the 2nd letter. After the 2nd pass, names are sorted on the first two letters. This process is continued till the nth pass, where n is the length of the name with maximum no. of letters.

When radix sort is used on integers, sorting is done on each of the digits in the number.

* The sorting procedure proceeds by sorting the least significant to the most significant digit.

* While sorting the numbers, we have ten buckets, each for one digit (0,1,2,... 9) and the no. of passes will depend on the length of the number having maximum no. of digits.

Als:-

RadixSort (ARR, N)

Step 1: Find the largest number in ARR as LARGE

Step 2: [INITIALIZE] Set NOP = Number of digits in LARGE

Step 3: Set PASS=0

Step 4: Repeat Step 5 while PASS <= NOP-1

Step 5: Set I=0 and INITIALIZE buckets

Step 6: Repeat Steps 7 to 9 while I < N-1

Step 7: Set DIGIT = digit at PASSth place in A[I]

Step 8: Add A[I] to the bucket numbered DIGIT

Step 9: INCREMENT bucket count for bucket numbered DIGIT

[END OF LOOP]

Step 10: Collect the numbers in the bucket

[END OF LOOP]

Step 11: END

Example:

345, 654, 924, 123, 567, 472, 555, 808, 911

In the first pass, the numbers are sorted according to the digit at ones place.

Number	0	1	2	3	4	5	6	7	8	9
345						345				
654					654					
924					924					
123				123						
567								567		
472			472							
555						555				
808									808	
911		911								

After this pass, the numbers are collected bucket by bucket. The new list thus formed is used as an input for the next pass. In the second pass, the numbers are sorted according to the digit at the tens place.

Number	0	1	2	3	4	5	6	7	8	9
911		911								
472								472		
123			123							
654						654				
924			924							
345					345					
555						555				
567							567			
808	808									

In the third pass, the numbers are sorted according to the digit at the hundreds place.

Number	0	1	2	3	4	5	6	7	8	9
808									808	
911										911
123		123								
924										924
345				345						
654							654			
555						555				
567						567				
472					472					

The numbers are collected bucket by bucket. After the third pass, the list can be given as,

123, 345, 472, 535, 567, 654, 808, 911, 924.

<u>Routine:</u>

```
void radixsort (int arr[], int n, int max)
{   /* max is the maximum element in the array */

  int mul =1;
  while (max)
  {
    countsort (arr, n, mul);
    mul*=10;
    max /=10;
  }
}
void countsort (int arr[], int n, int place)
{
  int i, freq [range] ={0};
  /* range for integers is 10 as digits range from 0-9
  int output[n];
  for (i=0; i<n; i++)
     freq [(arr[i]/place) % range]++;
  for (i=1; i<range; i++)
         freq[i] += freq[i-1];
  for (i=n-1; i>=0; i--)
  {
    output [freq [(arr [i] /place) % range]-1] = arr[i];
    freq [(arr [i]/ place) % range]--;
  } ~~for (i=0; i<n;~~
```

```
    for (i=0; i<n; i++)
        arr[i] = output[i];
}
```

Complexity:-

The running time of the radix sort depends on the no. of digit k, which is equal to the no. of passes and the no. of element n to be sorted in the list. It is in $O(k.n)$.

Hashing:-

Linear search has a running time proportional to $O(n)$, while binary search takes time proportional to $O(\log n)$, where n is the no. of elements in the array.

* If we want to perform the search operation in $O(1)$, i.e, in constant time, we have two solutions.

① Let us take an example to explain, in a small company of 100 employees, each employee is assigned an Emp-ID in the range 0 - 99.

* To store the records in an array, each employee's Emp-ID acts as an index into the array where the employee's records will be stored as in Fig.

key	Array of Employee's Records
key 0 → [0]	Employee record with Emp-ID 0
key 1 → [1]	" " " " 1
:	:
key 98 → [98]	" " " " 98
key 99 → [99]	" " " " 99

Fig. Record of employees

In this case, we can directly access the record of any employee, once we know his Emp-ID. But practically, this implementation is hardly infeasible as the Emp-ID goes high values.

② In the second solution, the elements are not stored according to the value of the key.

* So, in this case, we need to convert a two-digit array index into an another form.

* In this case, we will use the term hash table for an array and the function that will carry out the transformation will be called a hash function.

Hash Tables;-

Hash table is a data structure in which keys are mapped to array positions by a hash function.

* Here, we will use a hash function that extract the last two digit of the key. (in previous example).

* Therefore, we map the keys to array locations/indices.

* A value stored in a hash table can be searched in O(1) time by using a hash function which generates an address from the key

Fig shows a direct correspondence b/n the keys and the indices of the array.

* This is equivalent to our first example, where there are 100 keys for 100 employees.

However, when the set K of keys that are actually used is smaller than the universe of keys (U), a hash table consumes less storage space.

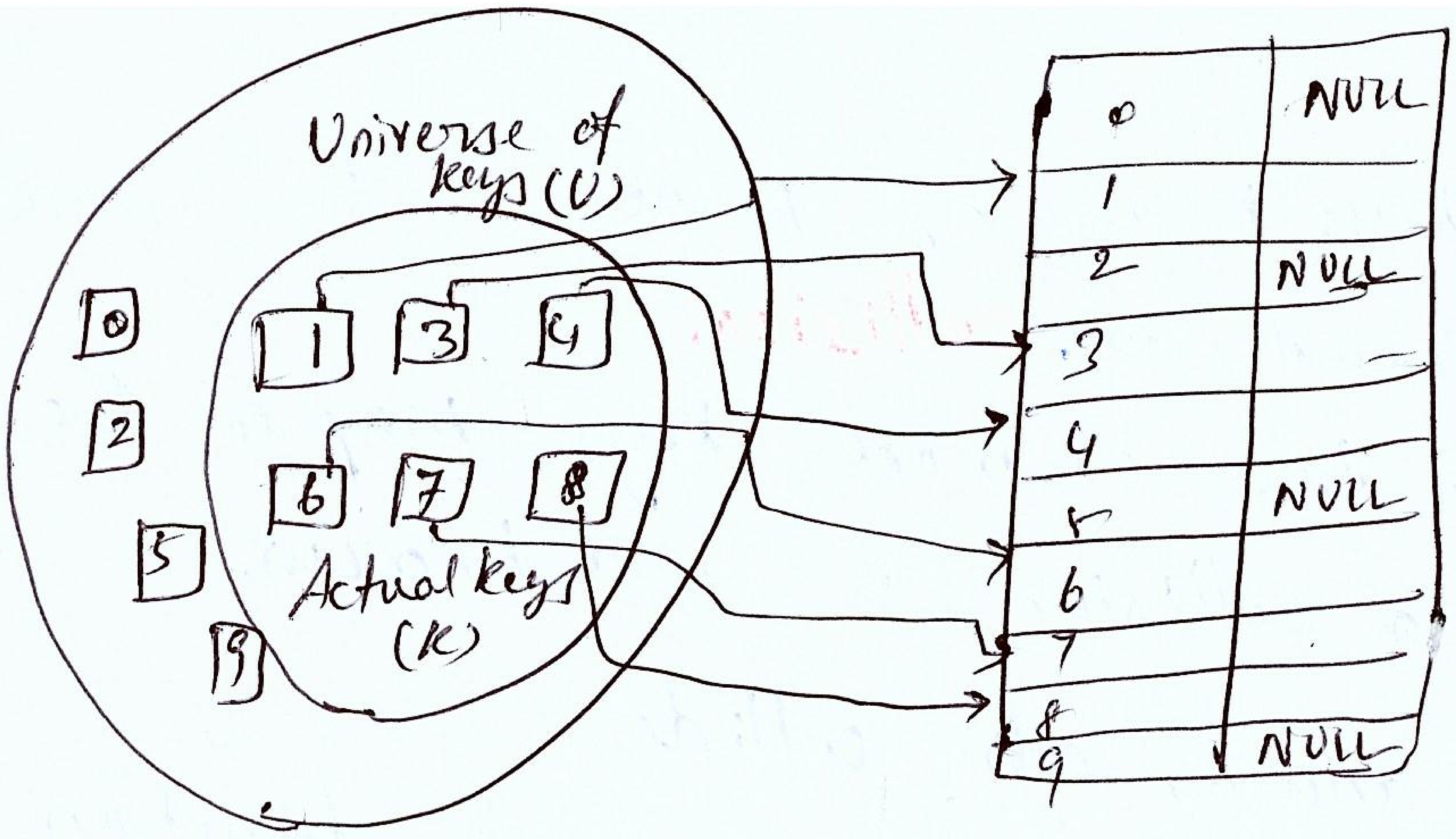


Fig. Direct relationship b/n key and index in the array

* The storage requirement for a hash table is O(k), where k is the no. of keys used.

In a hash table, an element with key k is stored at index h(k) and not k.

* It means a hash function h is used to calculate the index at which the element with key k will be stored.

* This process of mapping the keys to appropriate locations in a hash table is hashing.

The below Fig. shows a hash table in which each key from the set k is mapped to locations generated by using a hash function.

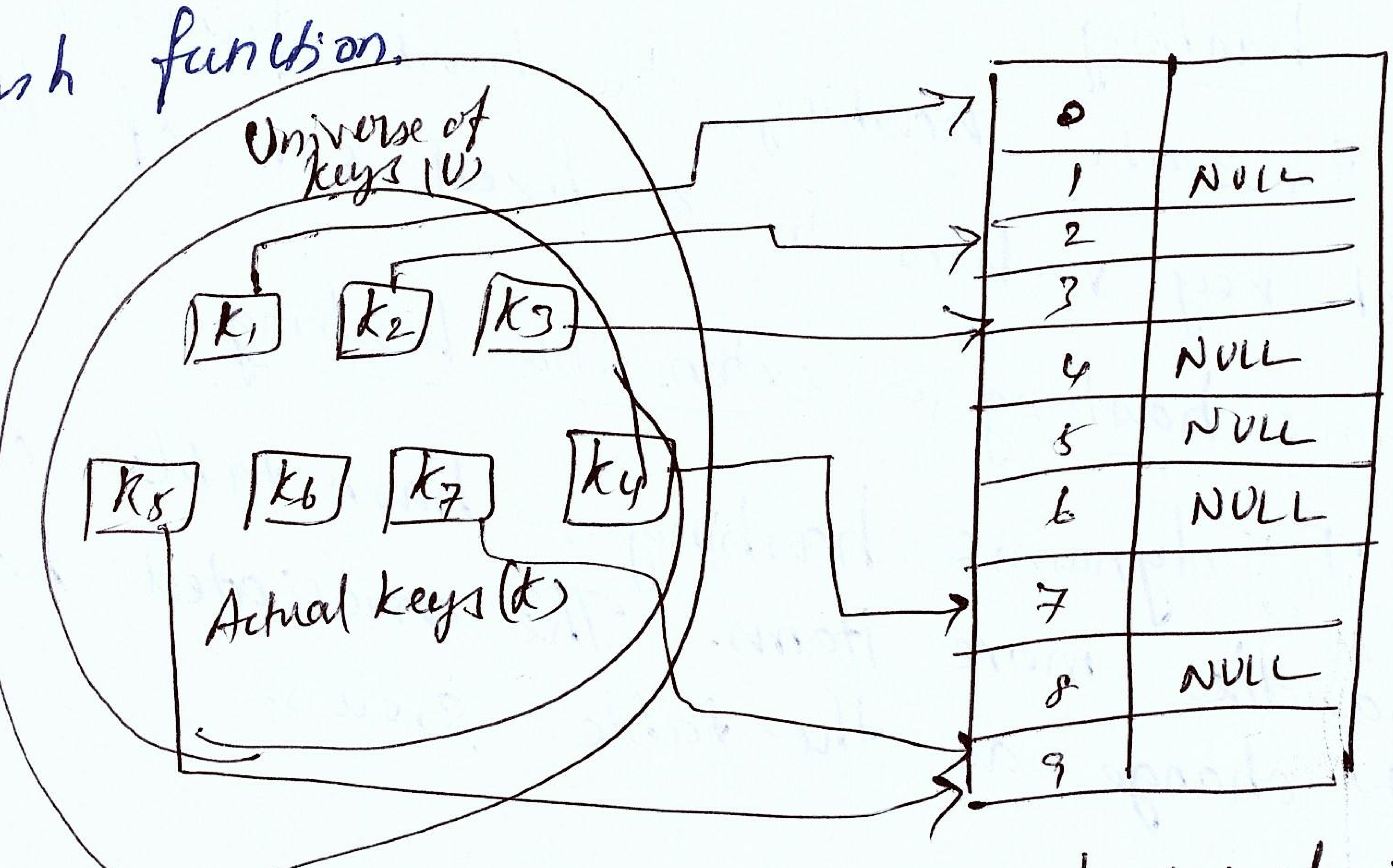


Fig. Relationship between keys and hash table index

* Note that keys k_2 and k_6 point to the same memory location. This is known as collision.

* That is, when two or more keys map to the same memory location, a collision is said to occur.

* Similarly k_5 and k_7 also collide.

* The main goal of using a hash function is to reduce the range of array indices that have to be handled and reduce the amount of storage space required.

There are 3 operations in hashing:

1). Insert: To insert a record, we compute the hash value and place the record in the index value returned.

2). Look Up: For lookup operation, compute the hash value as above and search each record in the location for the specific record.

3). Delete: To delete simply look up and remove.

There are 2 types of hashing.

1) Static hashing:

In static hashing, the hash function maps the search-key values to a fixed set of locations.

2) Dynamic hashing (or) extensible hashing:

In dynamic hashing, a hash table can grow to handle more items. The associated hash function must change as the table grows.

Hash Function;-

A hash function is a mathematical formula, which when applied to a key, produces an integer which can be used as an index for the key in the hash table.

* The values returned by a hash function are called hash values, hash codes, hash sums or hashes.

To achieve a good hashing mechanism, it is important to have a good hash function with the following basic requirements:

1) Easy to compute: It should be easy to compute and must not become an alg in itself.
2) Uniform distribution: It should provide a uniform distribution across the hash table and should not result in clustering.
3) Less collisions: Collisions occur when pairs of elements are mapped to the same hash value. These should be avoided.

Different Hash Functions;-

In real-world applications, we have alphanumeric keys rather than simple numeric keys. In this case, the ASCII value of the character can be used to transform it into its equivalent numeric key.

1) Division Method:

It is the most simple method of hashing an integer x. This method divides 'x' by M and then uses the remainder obtained.

$$\boxed{h(x) = x \mod M}$$

Ex: $M = 97$,

$h(1234) = 1234 \% 97 = 70$

$h(5642) = 5642 \% 97 = 16$

Suppose M is an even number then $h(x)$ is even if x is even and $h(x)$ is odd if x is odd.

* If all possible keys are equi-probable, then this is not a problem.
* But if even keys are more likely than odd keys, then the division method will not spread the hashed values uniformly.

Generally, it is best to choose M to be a prime number.

Drawback:

Using this method, consecutive keys map to consecutive hash values. This may lead to degradation in performance.

2) **Multiplication Method:-**

Step 1 : Choose a constant A such that $0 < A < 1$

Step 2 : Multiply the key k by A.

Step 3 : Extract the fractional part of kA.

Step 4 : Multiply the result of Step 3 by the size of the hash table (m).

Here, the has function is

$$h(k) = \lfloor m(kA \bmod 1) \rfloor$$

where $(kA \bmod 1)$ gives the fractional part of kA and m is the total no. of indices in the hash table.

The greatest adv. of this method is that it works practically with any value of A. Knuth has suggested that the best choice of A is $(\text{sqrt}5-1)/2 = 0.6180339887$.

Ex: $m = 1000$, $12345 \rightarrow$ key

$h(12345) = \lfloor 1000 \ (12345 \times 0.0618033 \bmod 1) \rfloor$

$= \lfloor 1000 \ (0.617385) \rfloor = \lfloor 617.385 \rfloor = 617.$

3) Mid-Square Method:-

The mid-square method is a good hash function.

Step 1: Square the value of the key, ie, find k^2.

Step 2: Extract the middle r digits of the result of k^2.

In mid-square method, the same r digits must be chosen from all the keys.

$$h(k) = s$$

where s is obtained by selecting r digits from k^2.

Ex:

1) Here, key = 5642, The hash table has 100 memory locations.

$$k^2 = 5642^2 = 318\underline{32}164$$

$$h(5642) = 21$$

Note that the hash table has 100 memory locations (0 to 99). This means that only two digits are needed to map the keys to a location in the hash table, so r=2.

2) Here, key = 456. The hash table has 100 memory locations.

$$k^2 = 456^2 = 20\underline{79}36.$$

$$h(456) = 79$$

4) Folding Method:-

Step 1: Divide the key value into a number of parts. ie, divide k into $k_1, k_2, \ldots k_n$ parts, where each part has the same no. of digits except the last part which may have lesser digits than the other parts.

Step 2: Add the individual parts. The hash value is ~~pro~~ duced by ignoring the last carry, if any.

Ex: Given a hash table of 100 locations keys 5678, 321, 34567.

key	5678	321	34567
Parts	56, 78	32, 1	34, 56, 7
Sum	134	33	97
Hash value	34 (ignore the last carry)	33	97

5) Truncation Method:-

It is the simplest and easiest method for computing hash values. This method truncates a part of a given keys, depending on the size of the hash table.

Ex:

Keys 987456, 125978, 963294 size of hash table = 1000

* Right or left most 3 digits are truncated and used as hash table addressed.
* The hash table addresses for the given keys are 456, 978, 294.
* There is a chance of collision by using method.

Hash Function:-

General format:

Hash (Key value) = key value mod TableSize

Ex:

Hash (38) = 38 mod 5 = 3., TableSize = 5.

The key value 38 is placed in location 3.

Routine for simple hash function:

```
Hash (char *key, int TableSize)
{
   int HashValue =0;
   while (*key != '\0')
        HashValue += *key ++;
   return HashValue % TableSize;
}
```

Ex:

key = "Hill"

ASCII values of this key is 72, 73, 76, 76.

Hill = 72+73+76+76 = 297

Hash(Hill) = 297 mod 5 = 2.

It occupies 2nd location in the hash table.

Collisions:-

Collisions occur when the hash function maps two different keys to the same location in the hash table. Obviously, two records cannot be stored in the same location. ∴, a method used to solve the pbm of collision, called collision resolution technique is applied.

1) Separate Chaning
2) Open Addressing
3) Rehashing.

1) Separate Chaining:-

Separate chaining is a collision resolution technique that maintains a linked list at every hash index for collided elements.

Routine:

```
void selection_sort (int arr[], int n)
{
  int i, j, temp;
  for (i=0; i<n-1; i++)
  {  for (j=i+1; j<n; j++)
     {  if (arr[i] > arr[j])
        {
           temp = arr[i];
           arr[i] = arr[j];
           arr[j] = temp;
        }
     }
  }
}
```

Complexity:-

In Pass 1, selecting the element with smallest value calls for scanning all n elements, thus, n-1 comparisons are required in the first pass.

* Then, the smallest value is swapped with the element in the first position.

* In Pass 2, selecting the second smallest value requires scanning the remaining n-1 elements and so on, Therefore

$$f(n) = (n-1) + (n-2) + \dots + 3 + 2 + 1.$$

$$= n(n-1)/2$$

$$f(n) \in O(n^2).$$

* Thus, the time complexity of this sorting is $O(n^2)$.

Adv:-

* Dynamic memory allocation is done
* Easy to locate the elements
* Collided elements can be searched at the same index

Disadv:-

* Longer linked lists could negatively impact on the performance.
* More memory is needed.

Coding:-

Type Declaration for separate chaining hash table.

```
#ifndef _HashSep_H

struct ListNode;
typedef struct ListNode *Position;
struct HashTbl;
typedef struct HashTbl *HashTable;

HashTable InitializeTable(int TableSize);
void DestroyTable(HashTable H);
Position Find(ElementType Key, HashTable H);
void Insert(ElementType Key, HashTable H);
ElementType Retrieve(Position P);
#endif /*_HashSep_H */

/* Place in the implementation file */
struct ListNode
{
    ElementType Element;
    Position Next;
};
typedef Position List;
```

```
struct HashTbl
{
  int TableSize;
  List * TheLists;
};
```

Initialization routine

```
HashTable Initialization (int TableSize)
{
  HashTable H;
  int i;
  if (TableSize < MinTableSize)
  {
    Error ("Table size too small");
    return NULL;
  }
  /* Allocate table */
  H = malloc (sizeof (struct HashTbl));
  if (H == NULL)
     FatalError ("Out of space!");
  /* Sets the table size to a prime number */
  H->TableSize = NextPrime (TableSize);
  /* Allocate array of lists */
  H->TheLists = malloc (sizeof (List) * H->TableSize);
  if (H->TheLists == NULL)
     FatalError ("Out of space");
  /* Allocate list headers */
  for (i = 0; i < H->TableSize; i++)
  {
```

```
H -> TheList[i] = malloc(sizeof(struct ListNode));
if (H -> TheLists[i] == NULL)
    FatalError("Out of Space");
else
    H -> TheLists[i] -> Next = NULL;
}
return H;
}
```

Find routine:

```
Position Find(ElementType key, HashTable H)
{
  Position P;
  List L;
  L = H -> TheLists[Hash(key, H -> TableSize)];
  P = L -> Next;
  while (P! = NULL && P -> Element ! = key)
      P = P -> Next;
  return P;
}
```

Insert routine;

```
void Insert(ElementType key, HashTable H)
{
  Position Pos, NewCell;
  List L;
  Pos = Find(Key, H);
  if (Pos == NULL)        /* key is not found */
  { NewCell = malloc(Size of(struct ListNode));
    if (NewCell == NULL)
      FatalError("Out of Space!!!");
```

```
else
{
    L = H -> TheList [ Hash (Key, H -> TableSize)];
    NewCell -> Next = L -> Next;
    NewCell -> Element = Key ;
    L -> Next = NewCell;
  }
 }
}
```

2) Open Addressing :-

In an open addressing hashing system, if a collision occurs, alternative cells are tried until an empty cell is found.

* More formally, cells $h_0(x)$, $h_1(x)$, $h_2(x)$, ... are tried in succession, where

$$\boxed{h_i(x) = (Hash(x) + F(i)) \bmod TableSize, \text{ with } F(0) = 0.}$$

* The function F, is the collision resolution strategy.

* There are 3 common probing strategies:

1). Linear Probing
2). Quadratic Probing
3). Double hashing.

* Probing is the process of look up and storage of the keys in hash table using open addressing.

1) Linear Probing:-

* In this method, searches for the empty position in a linear fashion (sequential manner).

* In linear probing, F is a linear function of i, typically.

$$\boxed{F(i) = i}$$

* In this method, the position of a key can be found sequentially searching all positions starting from the position calculated by the hash function until an empty cell is found.

* Suppose, if it reaches end of the table, no empty cells has been found, then the search is continued from the beginning of the table.

Example: Perform linear probing for the key values:
{ 89, 18, 49, 58, 69 }

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1					58	58
2						69
3						
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Fig. Open Addressing hash table with linear probing, after insertion of all keys.

$H(89) = 89 \% 10 = 9$. It will be mapped in the 9th position of the hash table.

$H(18) = 18 \% 10 = 8$. It will be mapped in the 8th position of the hash table

$H(49) = 49 \% 10 = 9$. It will be mapped in the 9th position of the hash table, since it is already mapped, it will search for the next free place which is the 0th position in the hash table.

$H(58) = 58 \% 10 = 8$. 8, 9 and 0th positions are already mapped, so it will search for the next free place 1st position in the hash table.

$H(69) = 69 \% 10 = 9$. 8, 9, 0 & 1st position are already mapped. So it will search for the next free place i.e, 2nd position in the hash table.

Adv:-

* This method is simple and fast
* It does not require pointers.
* No extra memory is needed.

Disadv:-

* It does not offer uniform hashing
* Primary clustering problem, i.e, if the hash table becomes half full and if a collision occurs, it is difficult to find an empty space in hash table and hence the insertion process takes longer time.
* Searching the collided elements takes more time.

2) Quadratic Probing :-

* It eliminates the clustering problem. It is quadratic, i.e.,

$$\boxed{F(i) = i^2}$$

* It is similar to linear probing.

Example: {89, 18, 49, 58, 69}

$h_0(89) = 89 \% 10 = 9$. It will be mapped in 9th position of the hash table.

	Empty Table	After 89	After 18	After 49	After 58	After 69
0				49	49	49
1						
2					58	58
3						69
4						
5						
6						
7						
8			18	18	18	18
9		89	89	89	89	89

Fig. Open Addressing hash table with quadratic probing, after the insertion of all keys.

$h_0(18) = 18 \% 10 = 8$. It will be mapped in 8th position of the hash table.

$h_0(49) = 49 \% 10 = 9$. Since it is already mapped in that position, Collision occurred. So find h_1

$h_1(49) = (h_0 + 1^2)\ \%\ 10 = (9+1)\ \%\ 10 = 0$. So it will be mapped in 0th position in hash table.

$h_0(58) = 58\ \%\ 10 = 8$. Since it is already mapped in that position, collision occurs. So find h_1.

$h_1(58) = (h_0 + 1^2)\ \%\ 10 = (8+1)\ \%\ 10 = 9$. It is already occupied. So find h_2.

$h_2(58) = (h_0 + 2^2) = (8+4)\ \%\ 10 = 2$. So it will be mapped in 2nd position.

$h_0(69) = 69\ \%\ 10 = 9$. Since it is already mapped, so find h_1.

$h_1(69) = (h_0 + 1^2)\ \%\ 10 = (9+1)\ \%\ 10 = 0$. Since it is already mapped in that position, so find h_2.

$h_2(69) = (h_0 + 2^2)\ \%\ 10 = (9+4)\ \%\ 10 = 3$. So it will be mapped in 3rd position in hash table.

Adv:-

* It is better than linear probing because it eliminates primary clustering.
* Easy to implement.

Disadv:-

* It may result in secondary clustering: if $h(k_1) = h(k_2)$, the probing sequences for k_1 and k_2 are exactly the same. This sequence of location is called a secondary cluster. This is harmful than primary clustering because secondary clustering do not combine to form large clusters.
* It will not search all locations in the hash table to find an empty slot, due to this insertion takes a longer time when compared to linear probing.

3) Double Hashing :-

For double hashing,

$$\boxed{F(i) = i \cdot hash_2(X).}$$

* This eliminates the secondary clustering problem.

* This formula says that we apply a second hash function to X and probe at a distance $hash_2(X)$, $2hash_2(X)$, ...

Example: [89, 18, 49, 58, 69]

	Empty Table	After 89	After 18	After 49	After 58	After 69
0						69
1						
2						
3					58	58
4						
5						
6				49	49	49
7						
8			18	18	18	18
9		89	89	89	89	89

Fig. Open addressing hash table with double hashing, after each insertion

Here, the function, $\boxed{hash_2(X) = R - (X \bmod R)}$, with R a prime number smaller than TableSize.

* If we choose R = 7, then Fig. shows the result of inserting the keys.

Keys = {89, 18, 49, 58, 69}

~~$h_0(89) = hash_2(89) = 7 - (89 \bmod 10)$~~

$h_0(89) = 89 \% 10 = 9$. So 89 is mapped in 9th position.

$h_0(18) = 18 \% 10 = 8$. So 18 is mapped in 8th position.

$h_0(49) = 49 \% 10 = 9$. Since it is already mapped, it will search for the next space, using the formula.

$hash_2(49) = 7 - (49 \bmod 7) = 7 - 0 = 7$.

$\therefore$ It is mapped at 6th position in the hash table

$h_0(58) = 58 \% 10 = 8$. Since it is already mapped, it will search for the next space using the formula.

$hash_2(58) = 7 - (58 \bmod 7) = 7 - 2 = 5$.

$\therefore$ It is mapped at 3rd position in the hash table.

$h_0(69) = 69 \% 10 = 9$. Since it is already mapped, it will search for the next space using the formula.

$hash_2(69) = 7 - (69 \bmod 7) = 7 - 6 = 1$

$\therefore$ It is mapped at 0th location in the hashtable.

Suppose the key is 60

$h_0 = 60 \% 10 = 0$. Since it is already mapped, it will search for the next space using the formula.

$hash_2(60) = 7 - (60 \bmod 7) = 7 - 4 = 3$.

Since this space is already mapped, we have to do $2\, hash_2(x)$

$2\, hash_2(60) = 2 * 3 = 6$.

$\therefore$ it is mapped at 5th location in the hash table.

Adv:-

* This avoids secondary clustering
* It is correctly implemented.
* Expected no. of probes

Disadv:-

* Time consuming process because it uses 2nd hash function
* Performance of double hashing will degrade rapidly, if the hash table is full.
* Deletions are difficult.

Program Code:

Type Declaration for open addressing hash tables

```
#ifndef _HashQuad_H

typedef unsigned int Index;
typedef Index Position;

struct HashTbl;
typedef struct HashTbl *HashTable;

HashTable InitializeTable (int TableSize);
void DestroyTable (HashTable H);
Position Find (ElementType key, HashTable H);
void Insert (ElementType key, HashTable H);
ElementType Retrieve (Position P, HashTable H);
HashTable Rehash (HashTable H);
#endif   /* _HashQuad_H */

/* Place in the implementation file */
enum KindOfEntry { Legitimate, Empty, Deleted };
```

```
struct HashEntry
{
  ElementType Element;
  enum KindOfEntry Info;
};
typedef struct HashEntry Cell;

struct HashTbl
{
  int TableSize;
  Cell *TheCells;
};
```

Initialization of hash table

```
HashTable InitializeTable(int TableSize)
{
   HashTable H;
   int i;
   if( TableSize < MinTableSize)
   {
      Error("Table size too small");
      return NULL;
   }
   /* Allocate table */
   H = malloc(sizeof(struct HashTbl));
   if(H == NULL)
       FatalError("Out of space!!!");

   H -> TableSize = NextPrime(TableSize);
   /* Allocate array of cells */
   H -> TheCells = malloc(sizeof(Cell) * H -> TableSize);
   if(H -> TheCells == NULL)
       FatalError("Out of Space!!!");
```

```
    for (i=0; i<H->TableSize; i++)
        H->TheCells [i]. Info = Empty;

    return H;
}
```

<u>Find routine</u>:-

```
Position Find(ElementType Key, HashTable H)
{
  Position CurrentPos;
  int CollisionNum;
  CollisionNum=0;
  CurrentPos = Hash (Key, H->TableSize);
  while( H->TheCells [ CurrentPos]. Info != Empty &&
         H->TheCells [ CurrentPos]. Element != Key)
  {
     CurrentPos += 2 * ++CollisionNum - 1;
     if (CurrentPos >= H->TableSize)
          CurrentPos -= H->TableSize;
  }
  return CurrentPos;
}
```

<u>Insert routine</u>:

```
Void Insert (ElementType Key, HashTable H)
{
  Position Pos;
  Pos = Find ( Key, H);
  if (H->TheCells [Pos]. Info != Legitimate)
  {
     H->TheCells [Pos]. Info = Legitimate;
     H->TheCells [Pos]. Element = Key;
  }
}
```

3) Rehashing:-

If the table gets too full, the running time for the operations will start taking too long and insertions might fail for open addressing hashing with quadratic/double hashing.

* This can happen if there are too many removals intermixed with insertions.
* A solution is to build an another hash table that is about twice as big.

As an example, suppose the elements 13, 15, 24, 6 are inserted into an open addressing hash table of size 7.

* The hash function is $h(x) = x \bmod 7$.
* Suppose linear probing is used to resolve collisions.
* The resulting table appears as

0	6
1	15
2	
3	24
4	
5	
6	13

$h(13) = 13 \% 7 = 6$

$h(15) = 15 \% 7 = 1$

$h(24) = 24 \% 7 = 3$

$h(6) = 6 \% 7 = 6$. Collision occurs.

So search next position is $\underline{0}$.

If 23 is inserted into the table, the resulting table is below will be over 70 percent full.

0	6
1	15
2	23
3	24
4	
5	
6	13

$h(23) = 23 \% 7 = 2$

* Because the table is so full, a new table is created.
* The size of this table is 17, because this is the first prime that is twice as large as the old table size.
* The new hash function is $h(X) = X \bmod 17$.
* The old table is scanned, and the elements 6, 15, 23, 24, 13 are inserted into the new table, appears below.
* This entire operation is called <u>rehashing</u>.
* The running time is $O(N)$, since there are N elements to rehash and the table size is 2N.

0	
1	
2	
3	
4	
5	
6	6
7	23
8	24
9	
10	
11	
12	
13	13
14	
15	15
16	

$h(6) = 6 \bmod 17 = 6$
$h(23) = 23 \bmod 17 = 7$
$h(24) = 24 \bmod 17 = 8$
$h(13) = 13 \% 17 = 13$
$h(15) = 15 \% 17 = 15$.

Rehashing can be implemented in several ways with quadratic probing.

1). Rehash as soon as the table is half full.
2) Rehash only when an insertion fails.
3). Middle-of-the-road strategy is to rehash when the table reaches a certain load factor.

Adv:-
* It is simple to implement
* It is used in applications where memory is not a constraint.

Disadv:-
* Consumes more memory.
* The hash values must be calculated for the rehashing, every time.

Extendible Hashing :- (Dynamic Hashing)

If the amount of data is too large, then we cannot able to fit in main memory.

* We assume that at any point we have N records to store; the value of N changes over time.

* There are at most M records fit in one disk block. Let us take M=4.

The major pbm in open addressing and separate chaining is that collisions could cause several blocks to be examined during a Find, even for a well-distributed hash table.

* Furthermore, when the table gets too full, an expensive rehashing step must be performed, which requires O(N) disk accesses.

* To overcome this, extendible hashing is used to allow a Find to be performed in two disk accesses. Insertions also requires few disk accesses.

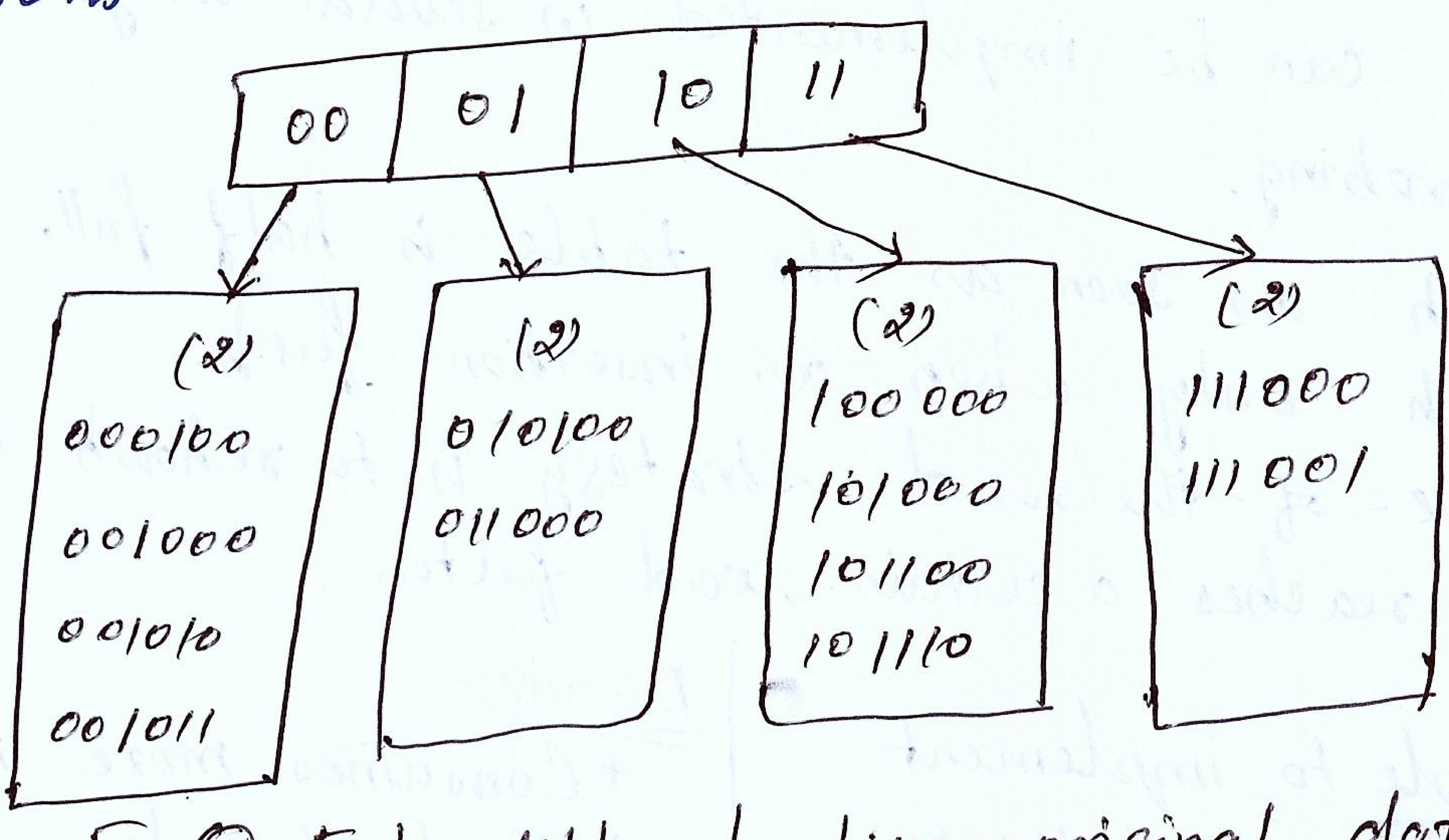


Fig.① Extendible hashing: original data

Let us take our data consists of several six-bit integers.

* Fig ① shows an extendible hashing scheme for these data.

* The root of the tree contains 4 pointers (00, 01, 10, 11) determined by the leading two bits of the data.

* Each leaf has up to $\underline{M=4}$ elements.
Table/bucket size

* The elements in the leaf are the binary numbers which is the hash key value. e.g. $h(288) = 288 \bmod 4 = 32.$
$= 100000.$

* It happens that in each leaf, the first two bits are identical and this is indicated by the number in parenthesis.

Let 'D' will represent the no. of bits used by the root, which is called as the directory.

* The no. of entries in the directory is 2^D.

* Let 'd_L' is the no. of leading bits that all the elements of some leaf L have in common.

* d_L will depend on the particular leaf, and $\boxed{d_L \leq D}$

Suppose we want to insert the key 100100, this would go into the third leaf, but there is no space to insert it.

* We split this leaf into two leaves, which are now determined by the first three bits.

* This requires increasing the directory size to 3. i.e, $D=3$.

* These changes are reflected in Fig ②.

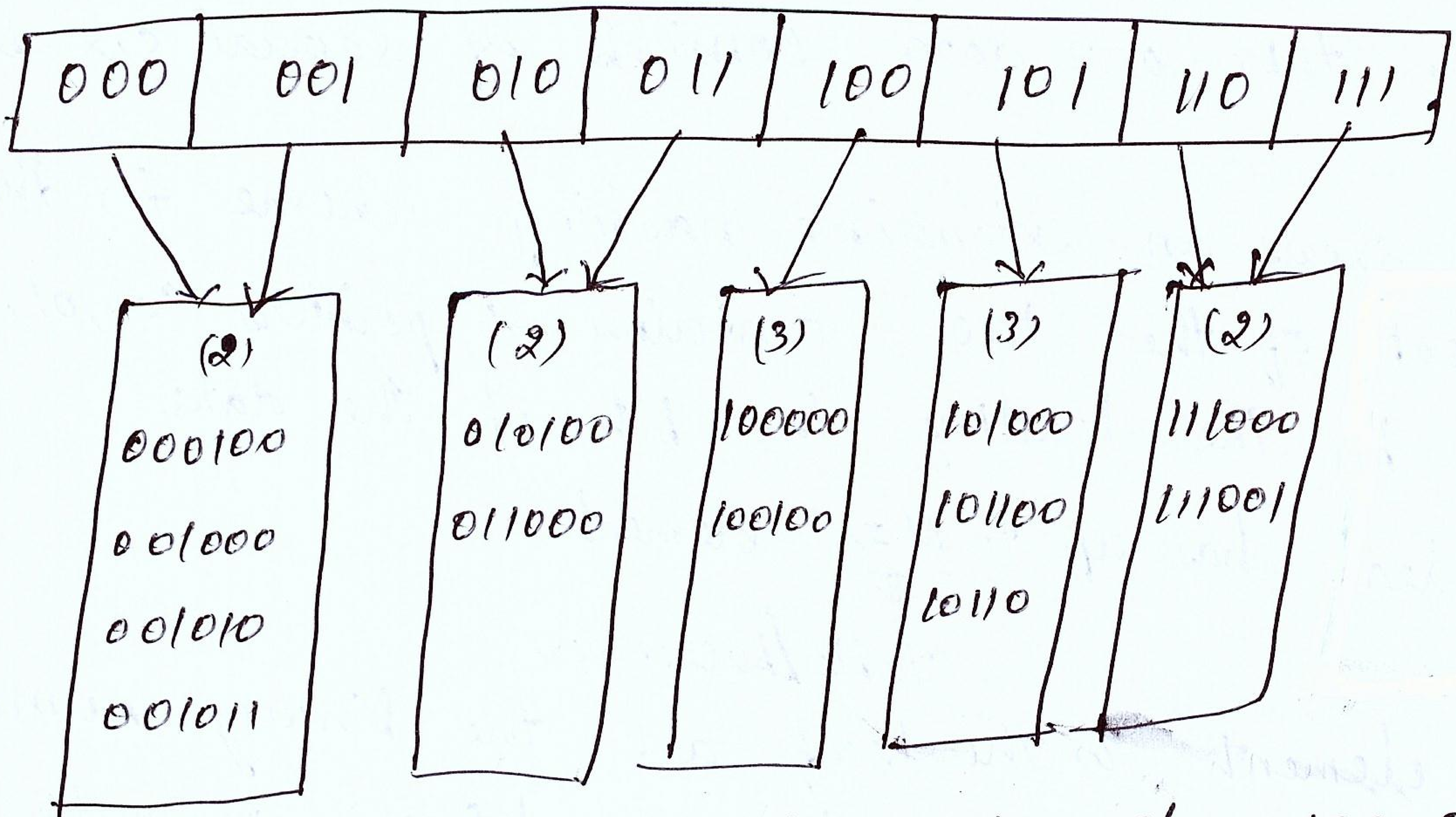


Fig ② Extendible hashing: after insertion of 100100 and directory split

If the key 000000 is now inserted, then the first leaf is split, generating two leaves with $d_L = 3$. Since $D = 3$, the only change required in the directory is the updating of 000 and 001 pointers. This is shown in Fig ③.

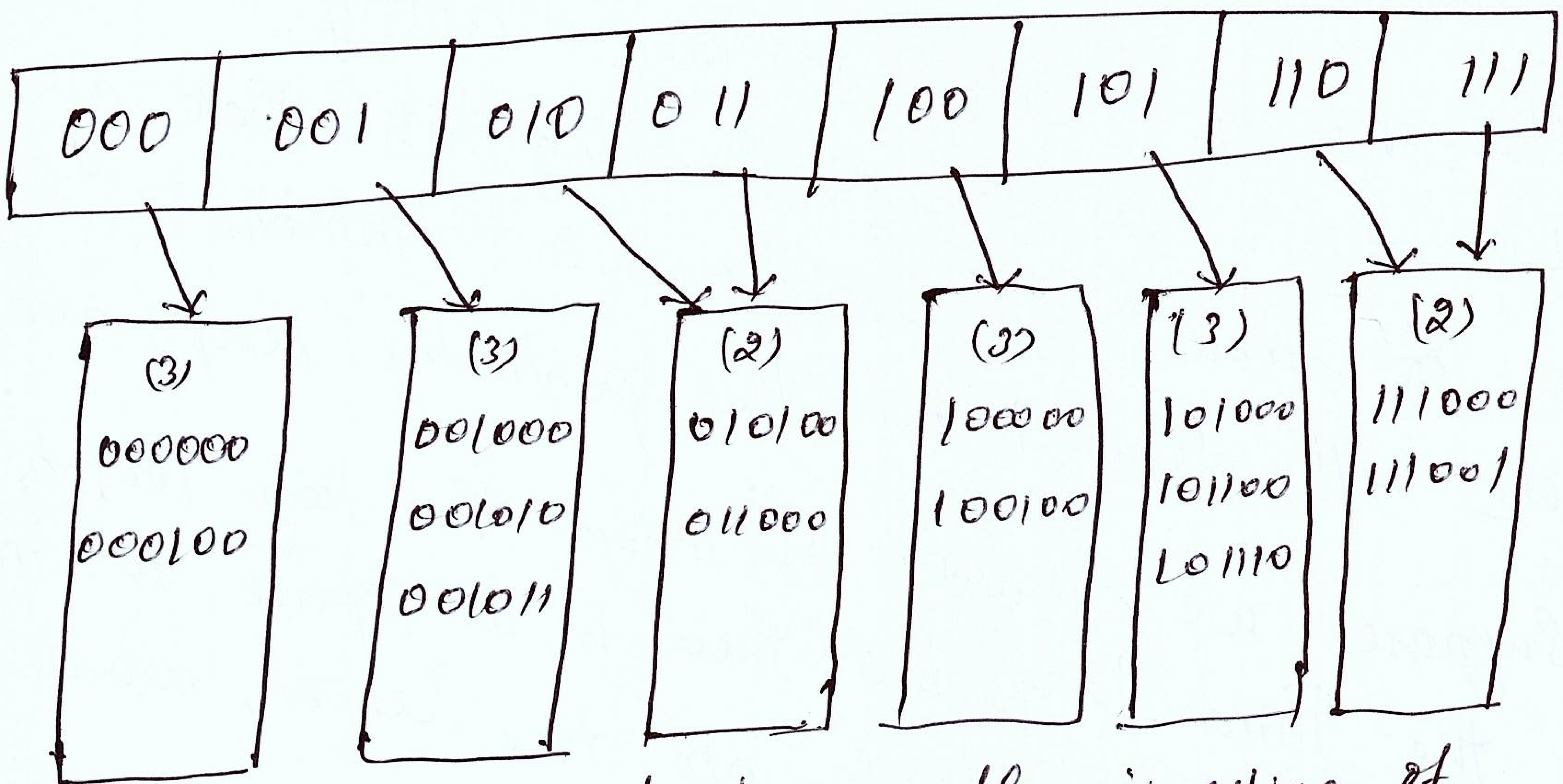


Fig. ③ Extendible hashing: after insertion of 000000 and leaf split

* This very simple strategy provides quick access times for Insert and Find operations on large databases.

Adv:-

* Improves the access time when large volumes of data have to be stored and accessed.
* Only the main directory has to be stored in the main memory. The leafs can be placed in secondary storage.

Disadvs:-

* Directory splits and re-hashing of values are tiresome.
* More no. of empty slots in certain leaf wastes space.