SNS COLLEGE OF ENGINEERING
Kurumbapalayam (Po), Coimbatore – 641 107
AN AUTONOMOUS INSTITUTION
Accredited by NBA–AICTE and Accredited by NAAC–UGC with 'A' Grade
Approved by AICTE, New Delhi & Affiliated to Anna University, Chennai

DEPARTMENT OF COMPUTER SCIENCE AND DESIGN

COURSE NAME : 19CS307- DATA STRUCTURES

II YEAR / III SEMESTER

UNIT-V

SORTING AND SEARCHING

INTRODUCTION TO SEARCHING ALGORITHMS

Searching is an operation or a technique that helps finds the place of a given element or value in the list. Any search is said to be successful or unsuccessful depending upon whether the element that is being searched is found or not. Some of the standard searching technique that is being followed in data structure is listed below:

1. Linear Search
2. Binary Search

LINEAR SEARCH

Linear search is a very basic and simple search algorithm. In Linear search, we search an element or value in a given array by traversing the array from the starting, till the desired element or value is found.

It compares the element to be searched with all the elements present in the array and when the element is **matched** successfully, it returns the index of the element in the array, else it return -1.

Linear Search is applied on unsorted or unordered lists, when there are fewer elements in a list.

For Example,

Linear Search

10	**14**	**19**	**26**	**27**	**31**	**33**	**35**	**42**	**44**

=
33

Algorithm

Linear Search (Array A, Value x)

Step 1: Set i to 1

Step 2: if i > n then go to step 7

Step 3: if A[i] = x then go to step 6

Step 4: Set i to i + 1

Step 5: Go to Step 2

Step 6: Print Element x Found at index i and go to step 8

Step 7: Print element not found

Step 8: Exit

Pseudocode

```
procedure linear_search (list, value)
  for each item in the list
    if match item == value
      return the item‟s location
    end if
  end for
end procedure
```

Features of Linear Search Algorithm

1. It is used for unsorted and unordered small list of elements.
2. It has a time complexity of **O(n)**, which means the time is linearly dependent on the number of elements, which is not bad, but not that good too.
3. It has a very simple implementation.

BINARY SEARCH

Binary Search is used with sorted array or list. In binary search, we follow the following steps:

1. We start by comparing the element to be searched with the element in the middle of the list/array.
2. If we get a match, we return the index of the middle element.
3. If we do not get a match, we check whether the element to be searched is less or greater than in value than the middle element.
4. If the element/number to be searched is greater in value than the middle number, then we pick the elements on the right side of the middle element(as the list/array is

ence on the right, we will have all the numbers greater than the middle number), and start again from the step 1.

5. If the element/number to be searched is lesser in value than the middle number, then we pick the elements on the left side of the middle element, and start again from the step 1.

Binary Search is useful when there are large number of elements in an array and they are sorted, so a necessary condition for Binary search to work is that the list/array should be sorted.

Features of Binary Search

1. It is great to search through large sorted arrays.
2. It has a time complexity of **O(log n)** which is a very good time complexity. It has a simple implementation.

Binary search is a fast search algorithm with run-time complexity of Ï(log n). This search algorithm works on the principle of divide and conquers. For this algorithm to work properly, the data collection should be in the sorted form.

Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched for in the sub-array to the right of the middle item. This process continues on the sub-array as well until the size of the sub array reduces to zero.

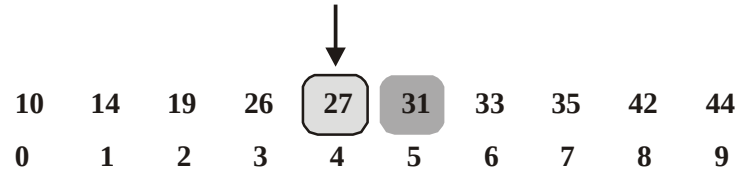
How Binary Search Works?

For a binary search to work, it is mandatory for the target array to be sorted. We shall learn the process of binary search with a pictorial example. The following is our sorted array and let us assume that we need to search the location of value 31 using binary search.

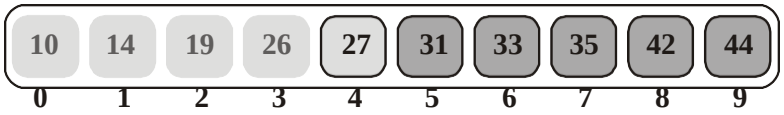
First, we shall determine half of the array by using this formula -

mid = low + (high - low) / 2

Here it is, 0 + (9 - 0) / 2 = 4 (integer value of 4.5). So, 4 is the mid of the array.



Now we compare the value stored at location 4, with the value being searched, i.e. 31. We find that the value at location 4 is 27, which is not a match. As the value is greater than 27 and we have a sorted array, so we also know that the target value must be in the upper portion of the array.

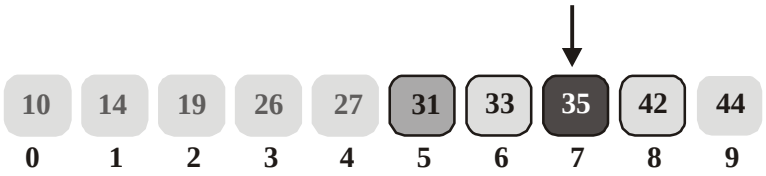


We change our low to mid + 1 and find the new mid value again.

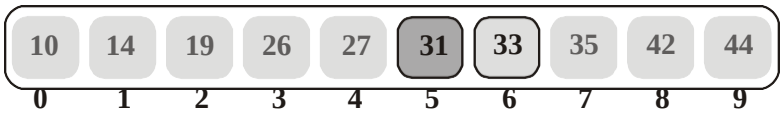
low = mid + 1

mid = low + (high - low) / 2

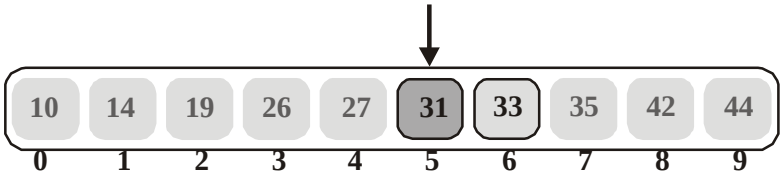
Our new mid is 7 now. We compare the value stored at location 7 with our target value 31.



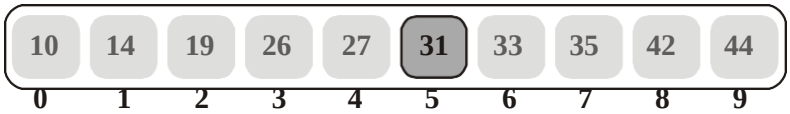
The value stored at location 7 is not a match, rather it is more than what we are looking for. So, the value must be in the lower part from this location.



Hence, we calculate the mid again. This time it is 5.



We compare the value stored at location 5 with our target value. We find that it is a match.



We conclude that the target value 31 is stored at location 5.

Binary search halves the searchable items and thus reduces the count of comparisons to be made to very less numbers.

Pseudocode

The pseudocode of binary search algorithms should look like this “

Procedure binary_search

```
A ! sorted array
n ! size of array
x ! value to be searched
Set lowerBound =  1
Set upperBound = n
while x not found
```

```
        if upperBound < lowerBound
          EXIT: x does not exists.
              set midPoint = lowerBound + ( upperBound - lowerBound ) / 2
         if A[midPoint] < x
          set lowerBound = midPoint + 1
         if A[midPoint] > x
          set upperBound = midPoint - 1
        if A[midPoint] = x
          EXIT: x found at location midPoint
      end while
    end procedure
```

SORTING

Preliminaries

A sorting algorithm is an algorithm that puts elements of a list in a certain order. The most used orders are numerical order and lexicographical order. Efficient sorting is important to optimizing the use of other algorithms that require sorted lists to work correctly and for producing human - readable input.

Sorting algorithms are often classified by :

* Computational complexity (worst, average and best case) in terms of the size of the list (N).

  For typical sorting algorithms good behaviour is O(NlogN) and worst case behaviour is $O(N^2)$ and the average case behaviour is O(N).

* Memory Utilization
* Stability - Maintaining relative order of records with equal keys.
* No. of comparisions.
* Methods applied like Insertion, exchange, selection, merging etc.

Sorting is a process of linear ordering of list of objects.

Sorting techniques are categorized into

$\Rightarrow$ Internal Sorting

$\Rightarrow$ External Sorting

Internal Sorting takes place in the main memory of a computer.

eg : - Bubble sort, Insertion sort, Shell sort, Quick sort, Heap sort, etc.

External Sorting, takes place in the secondary memory of a computer, Since the number of objects to be sorted is too large to fit in main memory.

eg : - Merge Sort, Multiway Merge, Polyphase merge.

THE BUBBLE SORT

The **bubble sort** makes multiple passes through a list. It compares adjacent items and exchanges those that are out of order. Each pass through the list places the next largest value in its proper place. In essence, each item "bubbles" up to the location where it belongs.

Fig. 5.1 shows the first pass of a bubble sort. The shaded items are being compared to see if they are out of order. If there are *n* items in the list, then there are n - 1n - 1 pairs of items that need to be compared on the first pass. It is important to note that once the largest value in the list is part of a pair, it will continually be moved along until the pass is complete.

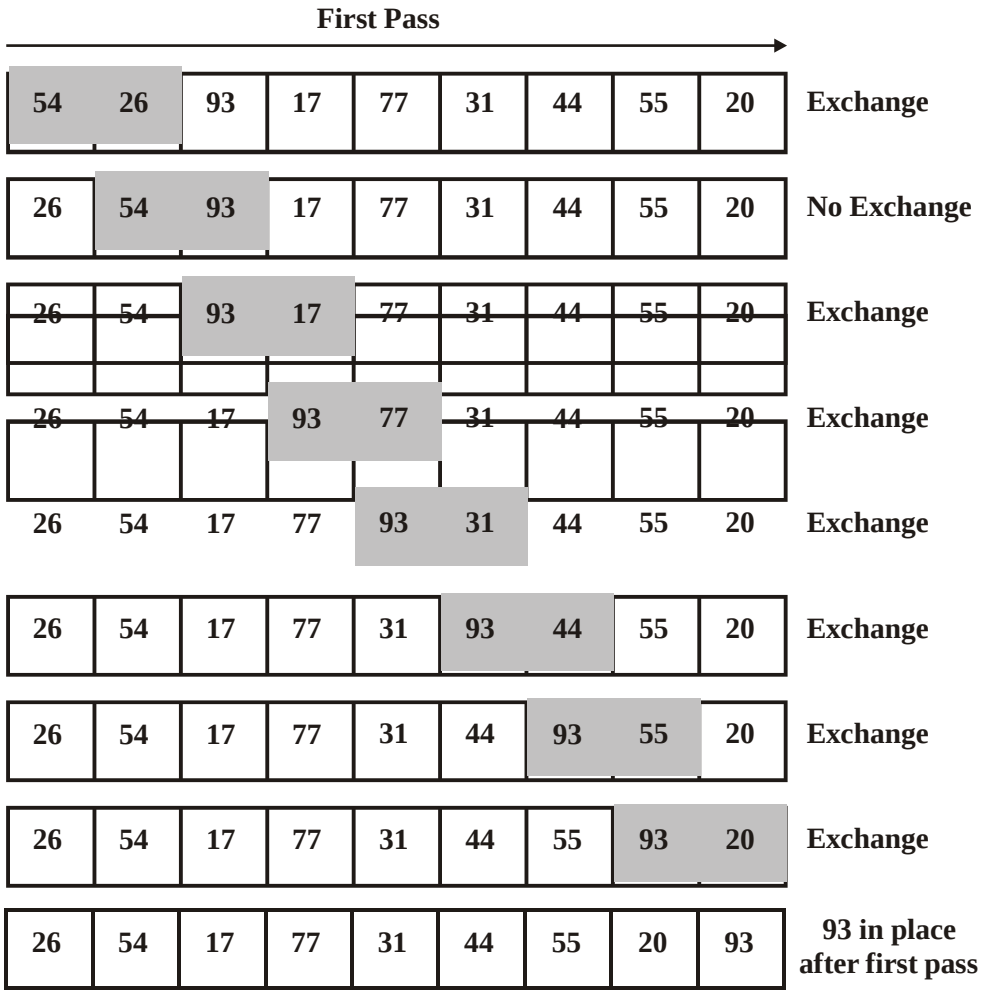


Fig. 5.1Bubble Sort

At the start of the second pass, the largest value is now in place. There are n - 1n - 1 items left to sort, meaning that there will be n - 2 n - 2 pairs. Since each pass places the next largest value in place, the total number of passes necessary will be n - 1 n - 1. After completing the n - 1 n - 1 passes, the smallest item must be in the correct position with no further processing required. The exchange operation, sometimes called a "swap".

Program for bubble sort:

```
def bubbleSort(alist):
    for passnum in range(len(alist)-1,0,-1):
        for i in range(passnum):
            if alist[i]>alist[i+1]:
                temp = alist[i]
                alist[i] = alist[i+1]
                alist[i+1] = temp
alist = [54,26,93,17,77,31,44,55,20]
bubbleSort(alist)
print(alist)
```

Output:

[17, 20, 26, 31, 44, 54, 55, 77, 93]

Analysis:

To analyze the bubble sort, we should note that regardless of how the items are arranged in the initial list, n"1n"1 passes will be made to sort a list of size *n*. Table -1 shows the number of comparisons for each pass. The total number of comparisons is the sum of the first n - 1n - 1 integers. In the best case, if the list is already ordered, no exchanges will be made. However, in the worst case, every comparison will cause an exchange. On average, we exchange half of the time.

Pass	Comparisons
1	n - 1n - 1
2	n - 2n - 2
3	n - 3n - 3
...	...
n - 1n - 1	11

Disadvantages:

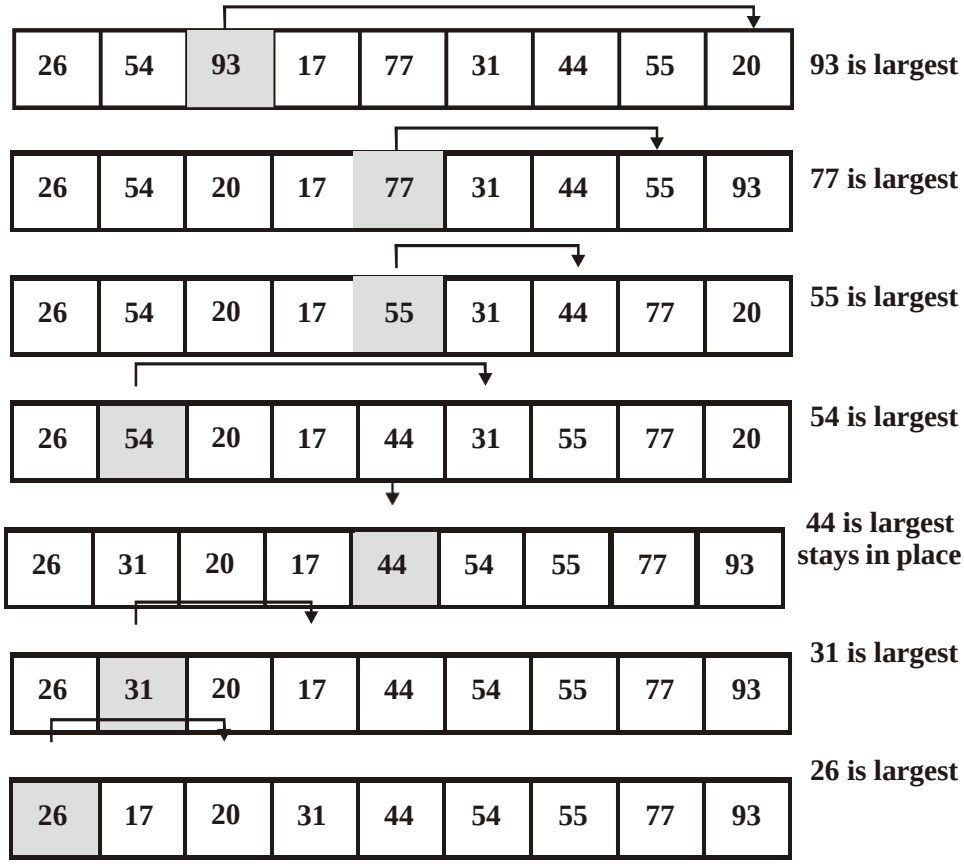
A bubble sort is often considered the most inefficient sorting method since it must exchange items before the final location is known. These "wasted" exchange operations are very costly. However, because the bubble sort makes passes through the entire unsorted portion of the list, it has the capability to do something most sorting algorithms cannot. In particular, if during a pass there are no exchanges, then we know that the list must be sorted. A bubble sort can be modified to stop early if it finds that the list has become sorted. This means that for lists that require just a few passes, a bubble sort may have an advantage in that it will recognize the sorted list and stop

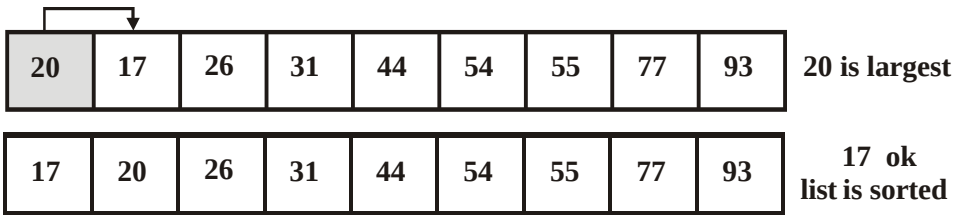
Bubble sort: https://youtu.be/p6I7LIUqQnU

5.6. THE SELECTION SORT

The **selection sort** improves on the bubble sort by making only one exchange for every pass through the list. In order to do this, a selection sort looks for the largest value as it makes a pass and, after completing the pass, places it in the proper location. As with a bubble sort, after the first pass, the largest item is in the correct place. After the second pass, the next largest is in place. This process continues and requires n"1n"1passes to sort *n* items, since the final item must be in place after the (n"1)(n"1) last pass.

Figure shows the entire sorting process. On each pass, the largest remaining item is selected and then placed in its proper location. The first pass places 93, the second pass places 77, the third places 55, and so on.





Program for Selection Sort:

```
def selectionSort(alist):
  for fillslot in range(len(alist)-1,0,-1):
      positionOfMax=0
      for location in range(1,fillslot+1):
          if alist[location]>alist[positionOfMax]:
              positionOfMax = location
      temp = alist[fillslot]
      alist[fillslot] = alist[positionOfMax]
      alist[positionOfMax] = temp
alist = [54,26,93,17,77,31,44,55,20]
selectionSort(alist)
print(alist)
```

Output:

[17, 20, 26, 31, 44, 54, 55, 77, 93]

Selection sort: https://youtu.be/xWBP4lzkoyM

INSERTION SORT

Insertion sorts works by taking elements from the list one by one and inserting them in their current position into a new sorted list. Insertion sort consists of N - 1 passes, where N is the number of elements to be sorted. The i^{th} pass of insertion sort will insert the i^{th} element A[i] into its rightful place among A[1], A[2] --- A[i - 1]. After doing this insertion the records occupying A[1] A[i] are in sorted order.

Insertion Sort Procedure

```
void Insertion_Sort (int a[ ], int n)
{
  int i, j, temp ;
  for (i = 0; i < n ; i++)
  {
        temp = a[i] ;
        for (j = i ; j>0 && a[j-1] > temp ; j--)
```

```
            {
                a[j] = a[ j - 1] ;
            }
            a[j] = temp ;
        }
    }
```

Example

Consider an unsorted array as follows,

20 10 60 40 30 15

Passes of Insertion Sort

ORIGINAL	20	10	60	40	30	15	POSITIONS MOVED
After i = 1	10	20	60	40	30	15	1
After i = 2	10	20	60	40	30	15	0
After i = 3	10	20	40	60	30	15	1
After i = 4	10	20	30	40	60	15	2
After i = 5	10	15	20	30	40	60	4
Sorted Array	10	15	20	30	40	60	

Analysis Of Insertion Sort

WORST CASE ANALYSIS - $O(N^2)$

BEST CASE ANALYSIS - $O(N)$

AVERAGE CASE ANALYSIS - $O(N^2)$

Limitations Of Insertion Sort :

* It is relatively efficient for small lists and mostly - sorted lists.
* It is expensive because of shifting all following elements by one.

Insertion Sort:https://youtu.be/gSdLGSM--dw

SHELL SORT

Shell sort was invented by Donald Shell. It improves upon bubble sort and insertion sort by moving out of order elements more than one position at a time. It works by arranging the data sequence in a two - dimensional array and then sorting the columns of the array using insertion sort.

In shell short the whole array is first fragmented into K segments, where K is preferably a prime number. After the first pass the whole array is partially sorted. In the next pass, the value of K is reduced which increases the size of each segment and reduces the number of segments. The

next value of K is chosen so that it is relatively prime to its previous value. The process is repeated

until K = 1, at which the array is sorted. The insertion sort is applied to each segment, so each successive segment is partially sorted. The shell sort is also called the Diminishing Increment Sort, because the value of K decreases continuously.

Shell Sort Routine

```
void shellsort (int A[ ], int N)
{
    int i, j, k, temp;
    for (k = N/2; k > 0 ; k = k/2)
        for (i = k; i < N ; i++)
        {
            temp = A[i];
            for (j = i; j >= k & & A [ j - k] > temp ; j = j - k)
            {
                A[j] = A[j - k];
            }
                A[j] = temp;
        }
}
```

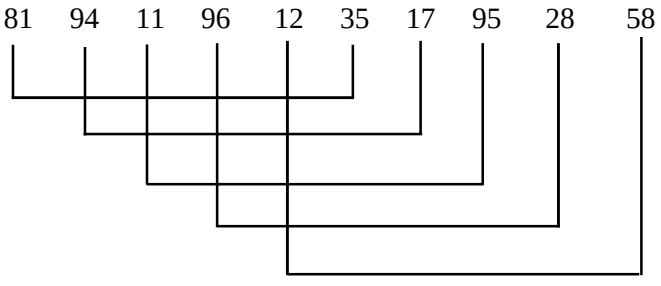
Example

Consider an unsorted array as follows.

81 94 11 96 12 35 17 95 28 58

Here N = 10, the first pass as K = 5 (10/2)

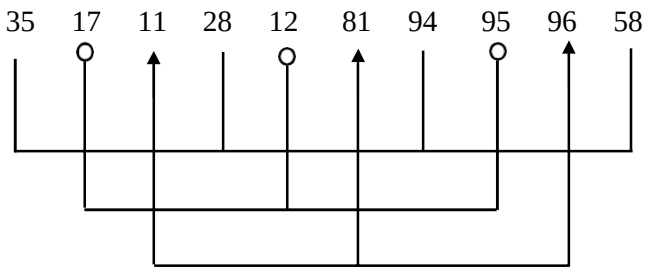
81 94 11 96 12 35 17 95 28 58



After first pass

35 17 11 28 12 81 94 95 96 58

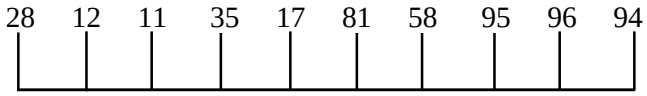
In second Pass, K is reduced to 3



After second pass,

28 12 11 35 17 81 58 95 96 94

In third pass, K is reduced to 1



The final sorted array is

11 12 17 28 35 58 81 94 95 96

Analysis Of Shell Sort :

WORST CASE ANALYSIS - $O(N^2)$
BEST CASE ANALYSIS- O(N log N)
AVERAGE CASE ANALYSIS - $O(N^{1.5})$

Advantages Of Shell Sort :

* It is one of the fastest algorithms for sorting small number of elements.
* It requires relatively small amounts of memory.

Shell Sort : https://youtu.be/SHcPqUe2GZM

Merge Sort

The Merge Sort algorithm is a sorting algorithm that is based on the Divide and Conquer paradigm. In this algorithm, the array is initially divided into two equal halves and then they are combined in a sorted manner.

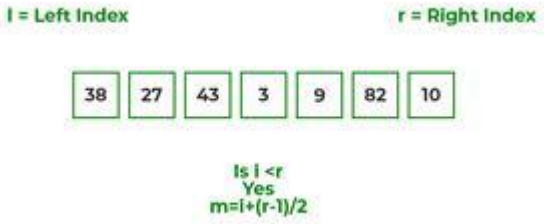
Merge Sort Working Process:

Think of it as a recursive algorithm continuously splits the array in half until it cannot be further divided. This means that if the array becomes empty or has only one element left, the dividing will stop, i.e. it is the base case to stop the recursion. If the array has multiple elements, split the array into halves and recursively invoke the merge sort on each of the halves. Finally, when both halves are sorted, the merge operation is applied. Merge operation is the process of taking two smaller sorted arrays and combining them to eventually make a larger one.

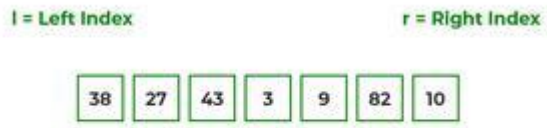
Illustration:

To know the functioning of merge sort, lets consider an array arr[] = {38, 27, 43, 3, 9, 82, 10}

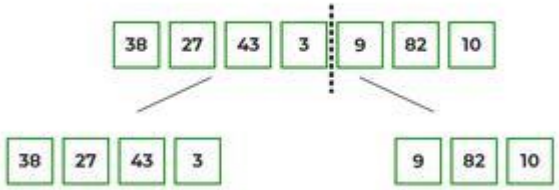
- At first, check if the left index of array is less than the right index, if yes then calculate its mid point



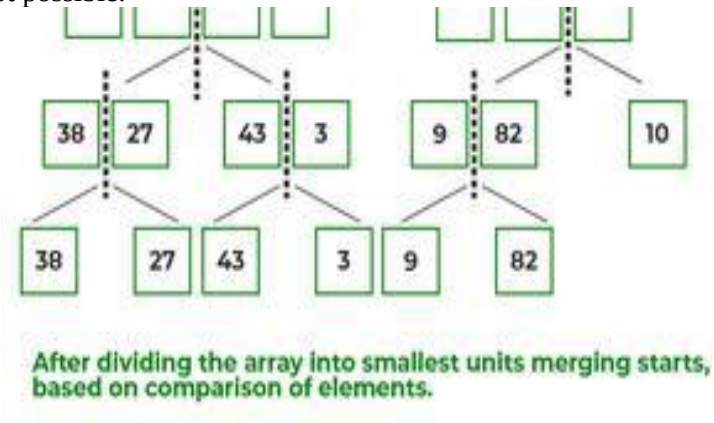
- Now, as we already know that merge sort first divides the whole array iteratively into equal halves, unless the atomic values are achieved.
- Here, we see that an array of 7 items is divided into two arrays of size 4 and 3 respectively.



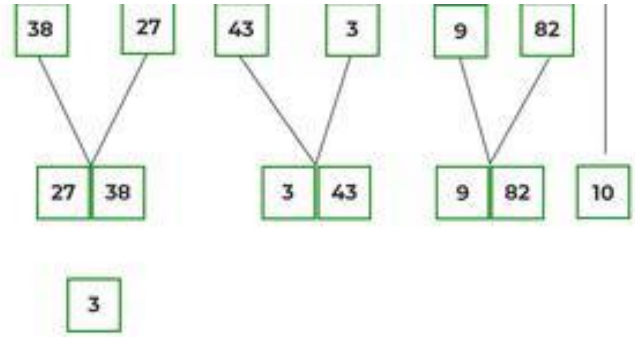
- Now, again find that is left index is less than the right index for both arrays, if found yes, then again calculate mid points for both the arrays.



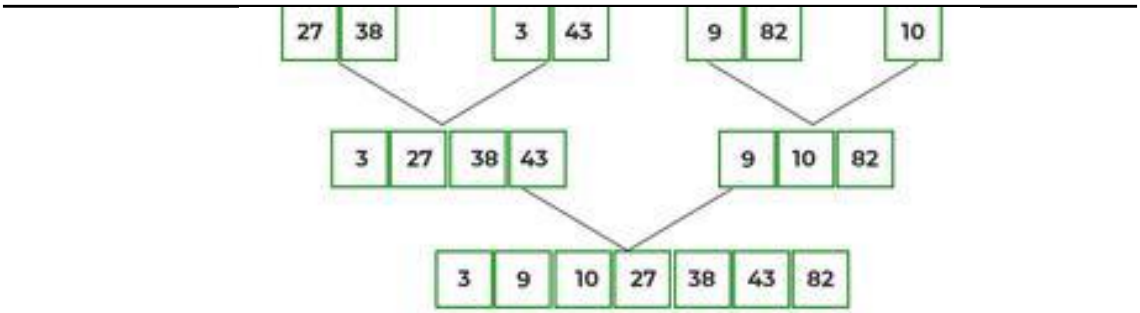
- Now, further divide these two arrays into further halves, until the atomic units of the array is reached and further division is not possible.



- After dividing the array into smallest units, start merging the elements again based on comparison of size of elements
- Firstly, compare the element for each list and then combine them into another list in a sorted manner.

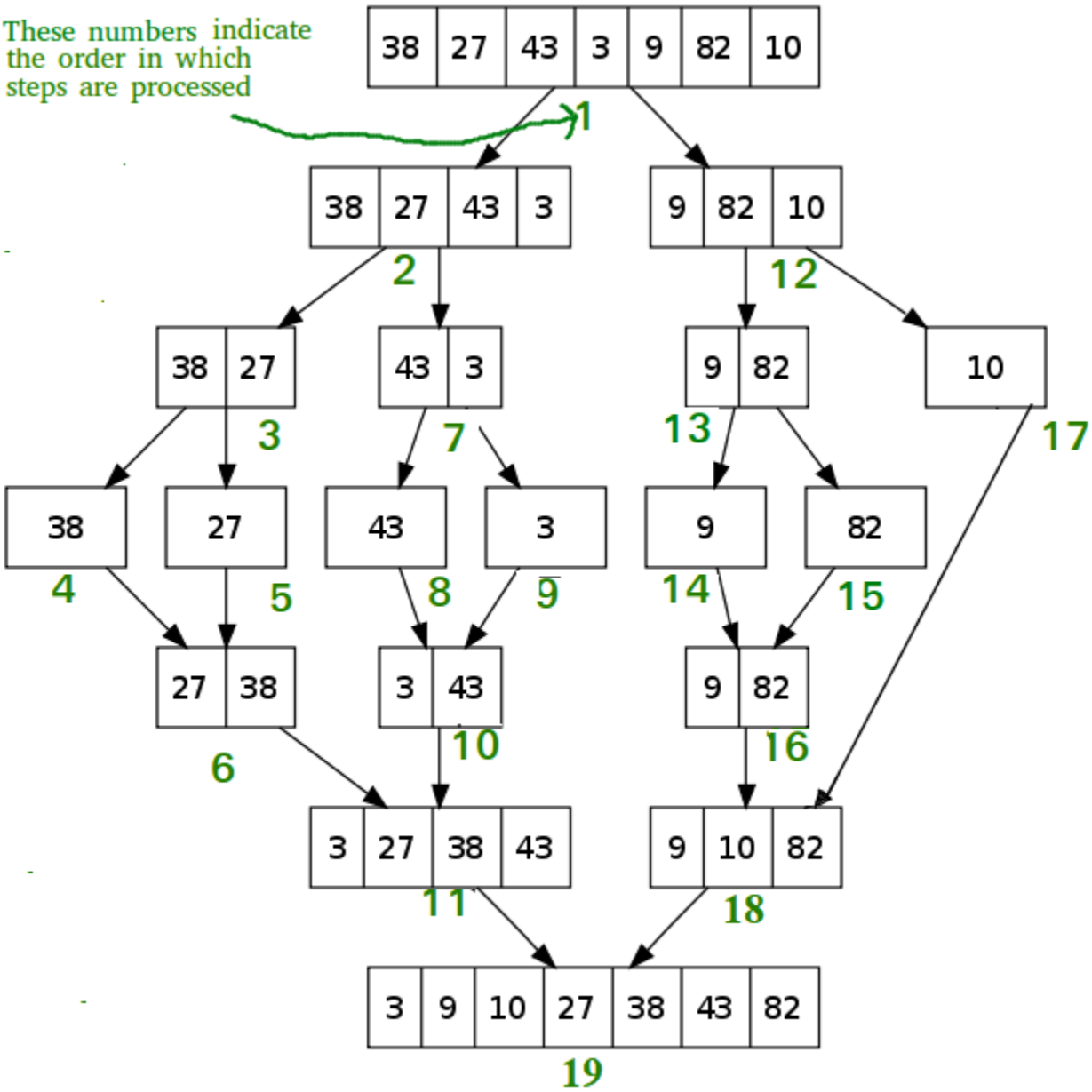


- After the final merging, the list looks like this:



The following diagram shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}.

If we take a closer look at the diagram, we can see that the array is recursively divided into two halves till the size becomes 1. Once the size becomes 1, the merge processes come into action and start merging arrays back till the complete array is merged.



Recursive steps of merge sort

Algorithm:

step 1: start

step 2: declare array and left, right, mid variable

step 3: perform merge function.

```
if left > right
    return
mid= (left+right)/2
mergesort(array, left, mid)
mergesort(array, mid+1, right)
merge(array, left, mid, right)
```

step 4: Stop

Follow the steps below to solve the problem:

MergeSort(arr[], l, r)
If r > l

- Find the middle point to divide the array into two halves:
    - middle m = l + (r – l)/2
- Call mergeSort for first half:
    - Call mergeSort(arr, l, m)
- Call mergeSort for second half:
    - Call mergeSort(arr, m + 1, r)
- Merge the two halves sorted in steps 2 and 3:
    - Call merge(arr, l, m, r)

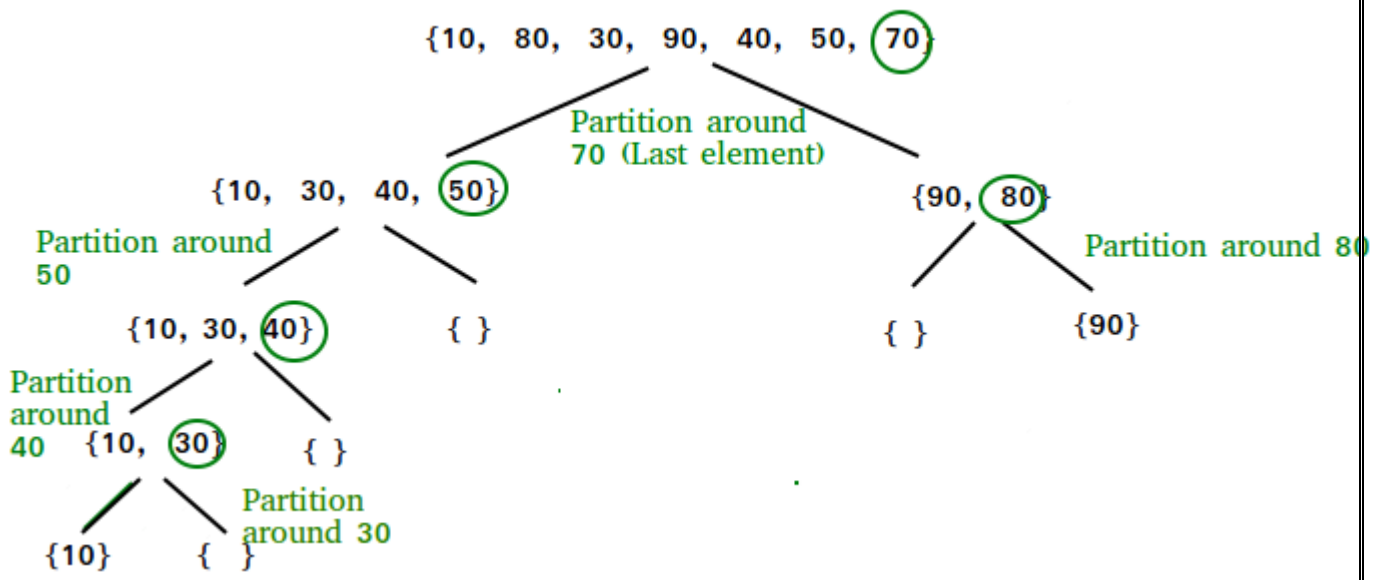
Merge sort: **https://youtu.be/JSceec-wEyw**

Quick sort

Like Merge Sort, QuickSort is a Divide and Conquer algorithm. It picks an element as a pivot and partitions the given array around the picked pivot. There are many different versions of quickSort that pick pivot in different ways.

- Always pick the first element as a pivot.
- Always pick the last element as a pivot (implemented below)
- Pick a random element as a pivot.
- Pick median as the pivot.

The key process in quickSort is a partition(). The target of partitions is, given an array and an element x of an array as the pivot, put x at its correct position in a sorted array and put all smaller elements (smaller than x) before x, and put all greater elements (greater than x) after x. All this should be done in linear time.



Partition Algorithm:

There can be many ways to do partition, following pseudo-code adopts the method given in the CLRS book. The logic is simple, we start from the leftmost element and keep track of the index of smaller (or equal to) elements as i. While traversing, if we find a smaller element, we swap the current element with arr[i]. Otherwise, we ignore the current element.

Pseudo Code for recursive QuickSort function:

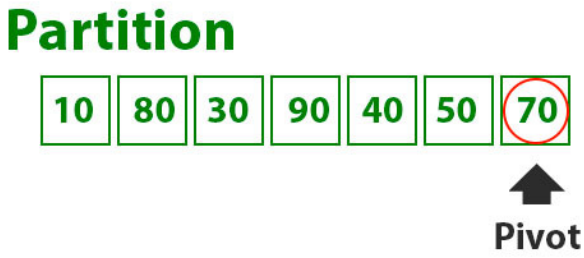
```
/* low  –> Starting index,  high  –> Ending index */
quickSort(arr[], low, high) {
   if (low < high) {
      /* pi is partitioning index, arr[pi] is now at right place */
      pi = partition(arr, low, high);
      quickSort(arr, low, pi – 1);  // Before pi
      quickSort(arr, pi + 1, high); // After pi
   }
}
```

Pseudo code for partition()

```
/* This function takes last element as pivot, places the pivot element at its correct position in sorted array, and
places all smaller (smaller than pivot) to left of pivot and all greater elements to right of pivot */
partition (arr[], low, high)
{
   // pivot (Element to be placed at right position)
   pivot = arr[high];
   i = (low – 1)  // Index of smaller element and indicates the
   // right position of pivot found so far
   for (j = low; j <= high- 1; j++){
      // If current element is smaller than the pivot
      if (arr[j] < pivot){
         i++;    // increment index of smaller element
         swap arr[i] and arr[j]
```

```
        }
    }
    swap arr[i + 1] and arr[high])
    return (i + 1)
}
```

Illustration of partition() :
Consider: arr[] = {10, 80, 30, 90, 40, 50, 70}
Indexes: 0 1 2 3 4 5 6
low = 0, high = 6, pivot = arr[h] = 70
Initialize index of smaller element, i = -1

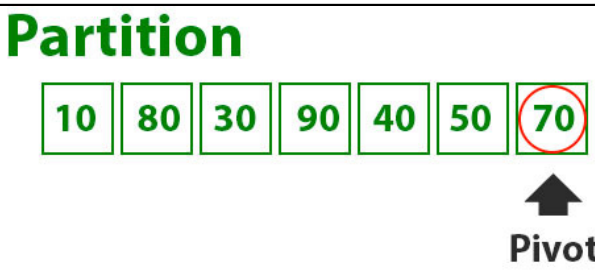


Counter variables
I: Index of smaller element
J: Loop variable

We start the loop with initial values

Test Condition	Actions	Value of variables
arr [J] <= pivot		I = -1 J = 0

Traverse elements from j = low to high-1
j = 0: Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 0
arr[] = {10, 80, 30, 90, 40, 50, 70} // No change as i and j are same
j = 1: Since arr[j] > pivot, do nothing

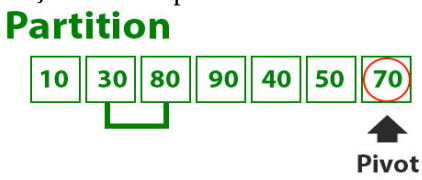


Counter variables
I: Index of smaller element
J: Loop variable

Pass 2

Test Condition	Actions	Value of variables
arr [J] <= pivot 80 < 70 false	No Action	I = 0 J = 1

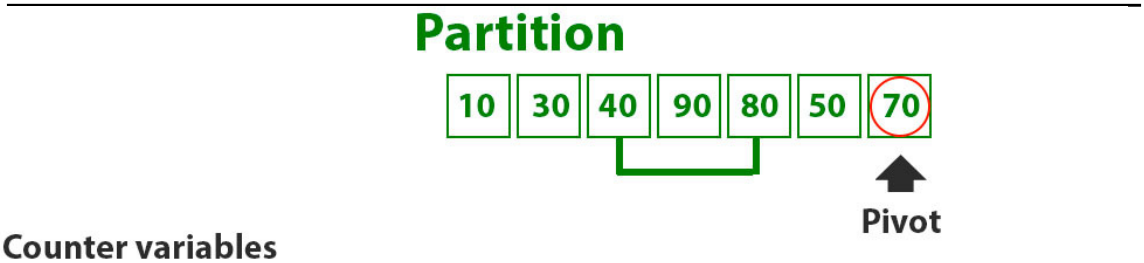
j = 2 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 1
arr[] = {10, 30, 80, 90, 40, 50, 70} // We swap 80 and 30



Counter variables
I: Index of smaller element
J: Loop variable

Test Condition	Actions	Value of variables
arr [J] <= pivot 30 < 70 true	i++ Swap(arr[i],arr[j])	I = 1 J = 2

j = 3 : Since arr[j] > pivot, do nothing // No change in i and arr[]
j = 4 : Since arr[j] <= pivot, do i++ and swap(arr[i], arr[j])
i = 2
arr[] = {10, 30, 40, 90, 80, 50, 70} // 80 and 40 Swapped

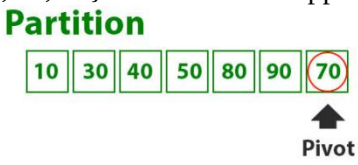


Counter variables
I: Index of smaller element
J: Loop variable

Pass 5

Test Condition	Actions	Value of variables
arr [J] <= pivot 40 < 70 true	i++ Swap(arr[i],arr[j])	I = 2 J = 4

j = 5 : Since arr[j] <= pivot, do i++ and swap arr[i] with arr[j]
i = 3
arr[] = {10, 30, 40, 50, 80, 90, 70} // 90 and 50 Swapped

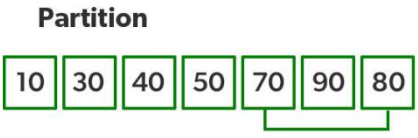


Counter variables
I: Index of smaller element
J: Loop variable

Before Pass 7, J becomes 6
so we come out of the loop

Test Condition	Actions	Value of variables
arr [J] <= pivot		I = 3 J = 6

We come out of loop because j is now equal to high-1.
Finally we place pivot at correct position by swapping arr[i+1] and arr[high] (or pivot)
arr[] = {10, 30, 40, 50, 70, 90, 80} // 80 and 70 Swapped

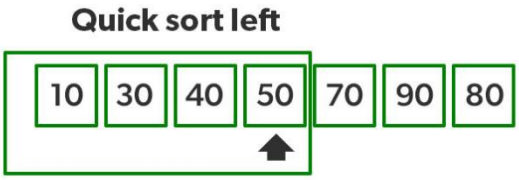


Counter Variable
I : Index of smaller element
J : Loop variable

We know swap arr[i+1] and pivot

I = 3

Now 70 is at its correct place. All elements smaller than 70 are before it and all elements greater than 70 are after it.
Since quick sort is a recursive function, we call the partition function again at left and right partitions

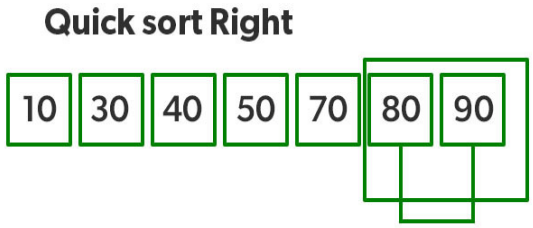


Since quick sort is a recursion function,
wecall the Partition function again

First 50 is the pivot.

As it is already at its correct position
we call the quicksort function again on the left part.

Again call function at right part and swap 80 and 90



80 is the Pivot

80 and 90 are swapped to bring pivot
to correct position

HASHING

Hash Table

The hash table data structure is an array of some fixed size, containing the keys. A key is a value associated with each record.

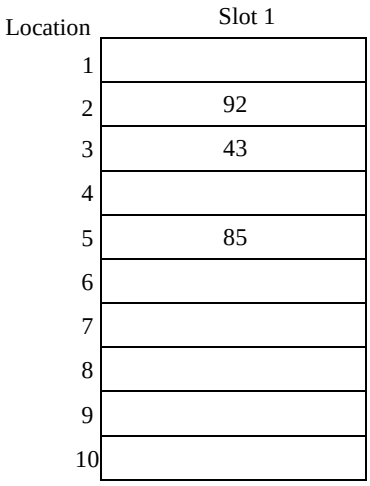


Fig. 5.10 Hash Table

HASHING FUNCTION

A hashing function is a key - to - address transformation, which acts upon a given key to compute the relative position of the key in an array.

A simple Hash function

HASH (KEYVALUE) = KEYVALUE MOD TABLESIZE

Example : - Hash (92)

Hash (92) = 92 mod 10 = 2

The keyvalue „92" is placed in the relative location „2".

Routine For Simple Hash Function

```
Hash (Char *key, int Table Size)
{
        int Hashvalue = 0;
        while (* key ! = „\0")
                Hashval + = * key ++;
        return Hashval % Tablesize;
}
```

Some of the Methods of Hashing Function

1. Module Division
2. Mid - Square Method
3. Folding Method
4. PSEUDO Random Method
5. Digit or Character Extraction Method
6. Radix Transformation.

Collisions

A Collision occurs when two or more elements are hashed (mapped) to same value (i.e) When two key values hash to the same position.

Collision Resolution

When two items hash to the same slot, there is a systematic method for placing the second item in the hash table. This process is called collision resolution.

Some of the Collision Resolution Techniques

1. Seperate Chaining 2. Open Addressing 3. Multiple Hashing

SEPERATE CHAINING

Separate chaining is an open hashing technique. A pointer field is added to each record location. When an overflow occurs this pointer is set to point to overflow blocks making a linked list.

In this method, the table can never overflow, since the linked list are only extended upon the arrival of new keys.

Insert : 10, 11, 81, 10, 7, 34, 94, 17

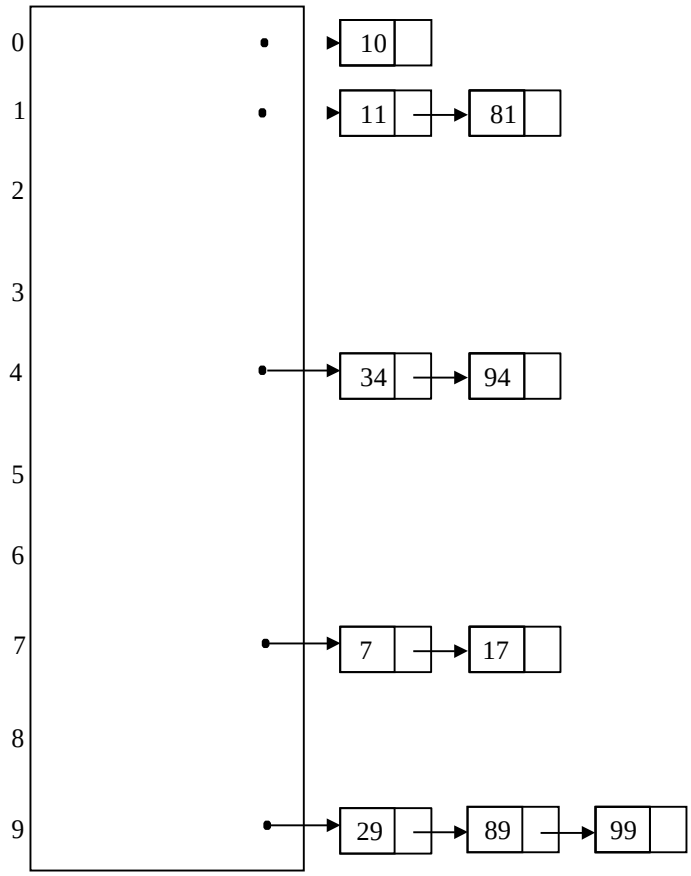


Fig. 5.12

Insertion

To perform the insertion of an element, traverse down the appropriate list to check whether the element is already in place.

If the element is new one, the inserted it is either at the front of the list or at the end of the list.

If it is a duplicate element, an extra field is kept and placed.

INSERT 10 :

Hash (k) = k% Tablesize

Hash (10) = 10 % 10

Hash (10) = 0

INSERT 11 :

Hash (11) = 11 % 10

Hash (11) = 1

INSERT 81 :

Hash (81) = 81% 10

Hash (81) = 1

The element 81 collides to the same hash value 1. To place the value 81 at this position perform the following.

Traverse the list to check whether it is already present.

Since it is not already present, insert at end of the list. Similarly the rest of the elements are inserted.

Routine To Perform Insertion

```
void Insert (int key, Hashtable H)
{
    Position Pos, Newcell;
    List L;
/* Traverse the list to check whether the key is already present */
    Pos = FIND (Key, H);
    If (Pos = = NULL) /* Key is not found */
    {
        Newcell = malloc (size of (struct ListNode));
        If (Newcell ! = NULL)
```

```
            {
                L = H → Thelists [Hash (key, H → Tablesize)];
                Newcell → Next = L → Next;
                Newcell → Element = key;
                /* Insert the key at the front of the list */
                 L → Next = Newcell;
            }
        }
    }
```

Find Routine

```
Position Find (int 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;
}
```

Advantage

More number of elements can be inserted as it uses array of linked lists.

Disadvantage of Separate Chaining

* It requires pointers, which occupies more memory space.
* It takes more effort to perform a search, since it takes time to evaluate the hash function and also to traverse the list.

Hashing: https://www.upgrad.com/blog/hashing-in-data-structure/

Rehashing

Link : https://www.scaler.com/topics/data-structures/load-factor-and-rehashing/