UNIT V

SORTING, SEARCHING AND HASH TECHNIQUES

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

Sorting:

A **sorting algorithm** is an algorithm that puts elements of a list in a certain order.

Types of Sorting:

- Internal Sorting
- External Sorting

Internal Sorting:

Internal Sorting takes place in the main memory of the computer**.** It is applicable when the number of elements in the list is small.
E.g. Bubble Sort, Inserting Sort, Shell Sort, Quick Sort.

External Sorting:

External Sorting takes place in the Secondary memory of the computer**.** It is applicable when the number of elements in the list is large.
E.g. Merge Sort, Multiway Merge Sort.

Types of sorting algorithms:

- ✓ **Insertion sort**
- ✓ **Selection sort**
- ✓ **Shell sort**
- ✓ **Bubble sort**
- ✓ **Quick sort**
- ✓ **Merge sort**

- ✓ **Radix sort**

Insertion Sort Algorithm

Procedure:

- ✓ Step 1: The second element of an array is compared with the elements that appear before it (only first element in this case). If the second element is smaller than first element, second element is inserted in the position of first element. After first step, first two elements of an array will be sorted.
- ✓ Step 2: The third element of an array is compared with the elements that appears before it (first and second element). If third element is smaller than first element, it is inserted in the position of first element. If third element is larger than first element but, smaller than second element, it is inserted in the position of second element. If third element is larger than both the elements, it is kept in the position as it is. After second step, first three elements of an array will be sorted.
- ✓ Step 3: Similarly, the fourth element of an array is compared with the elements that appear before it (first, second and third element) and the same procedure is applied and that element is inserted in the proper position. After third step, first four elements of an array will be sorted. If there are *n* elements to be sorted. Then, this procedure is repeated *n-1* times to get sorted list of array.

How insertion sort Algorithm works?

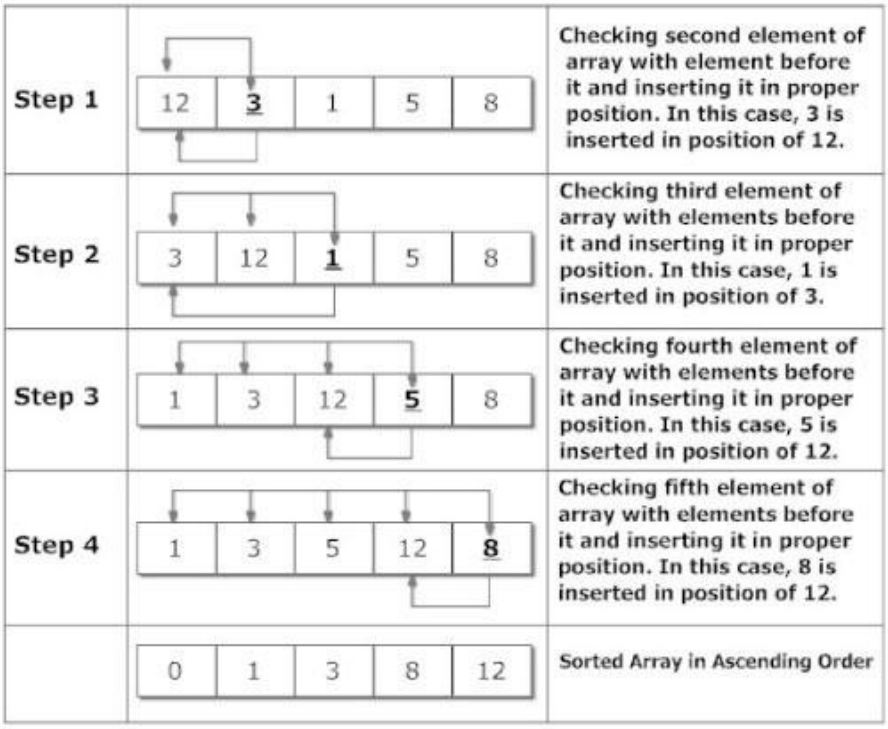


Figure: Sorting Array in Ascending Order Using Insertion Sort Algorithm

Algorithm for Insertion Sort:

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

Limitation of Insertion Sort:-

- Applicable only for small list.
- It is expensive since it involves many data movements.

Selection Sort Algorithm

Selection sort algorithm starts by comparing first two elements of an array and swapping if necessary, i.e., if you want to sort the elements of array in ascending order and if the first element is greater than second then, you need to swap the elements but, if the first element is smaller than second, leave the elements as it is. Then, again first element and third element are compared and swapped if necessary. This process goes on until first and last element of an array is compared. This completes the first step of selection sort.

If there are n elements to be sorted then, the process mentioned above should be repeated n-1 times to get required result. But, for better performance, in second step, comparison starts from second element because after first step, the required number is automatically placed at the first (i.e., In case of sorting in ascending order, smallest element will be at first and in case of sorting in descending order, largest element will be at first.). Similarly, in third step, comparison starts from third element and so on.

A figure is worth 1000 words. This figure below clearly explains the working of selection sort algorithm.

Algorithm for Selection Sort:

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

- For “n” elements, (n-1) passes are required.
- At the end of the i^{th} iteration, the i^{th} smallest element will be placed in its correct position.

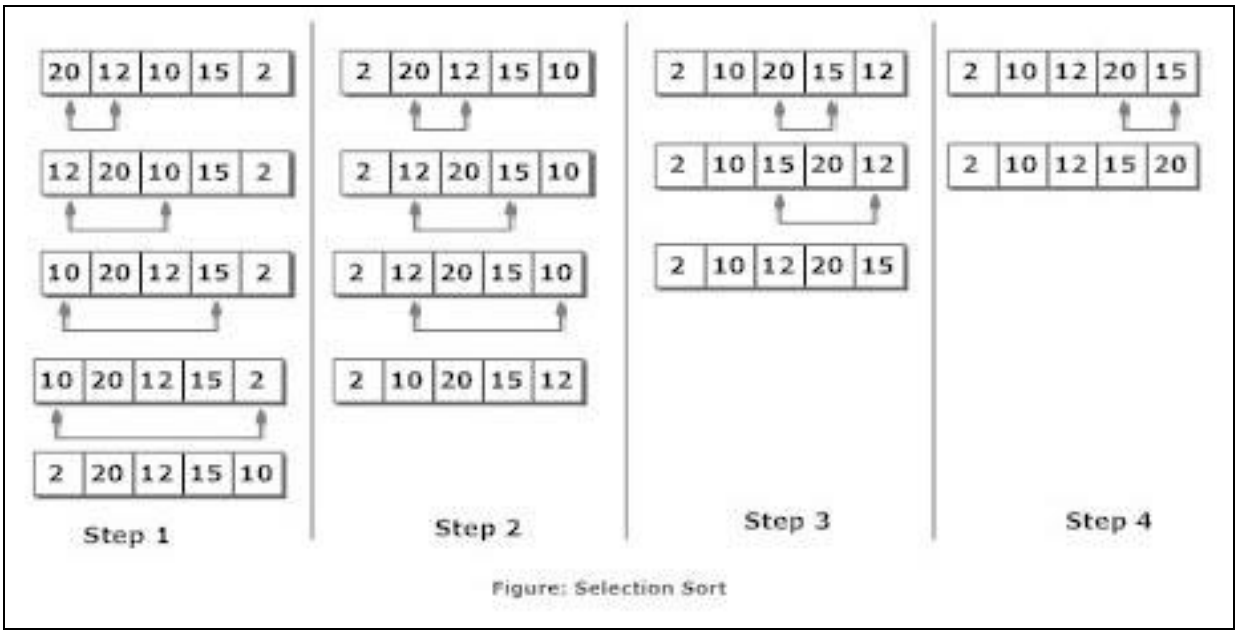
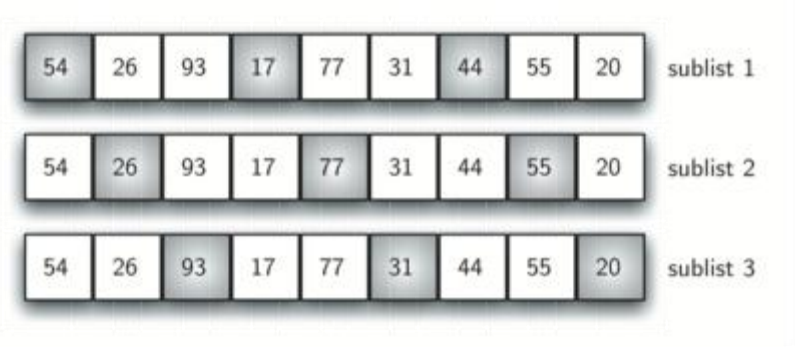


Figure: Selection Sort

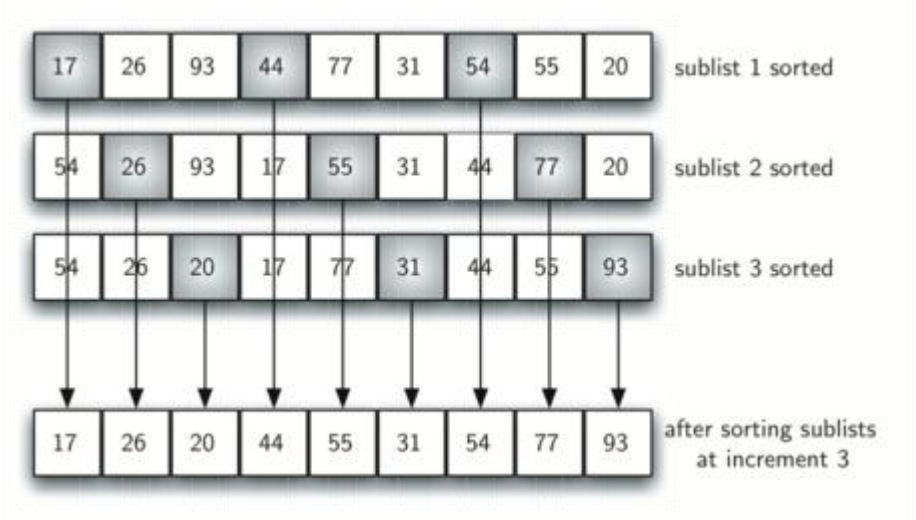
Shell Sort

Shell sort works by comparing elements that are distant rather than adjacent elements in an array or list where adjacent elements are compared. Shell sort uses an increment sequence. The increment size is reduced after each pass until the increment size is 1. With an increment size of 1, the sort is a basic insertion sort, but by this time the data is guaranteed to be almost sorted, which is insertion sort's "best case". The distance between comparisons decreases as the sorting algorithm runs until the last phase in which adjacent elements are compared hence, it is also known as diminishing increment sort.

Consider a list has nine items. If we use an increment of three, there are three sublist, each of which can be sorted by an insertion sort. After completing these sorts, we get the list shown below. Although this list is not completely sorted, something very interesting has happened. By sorting the sublist, we have moved the items closer to where they actually belong.

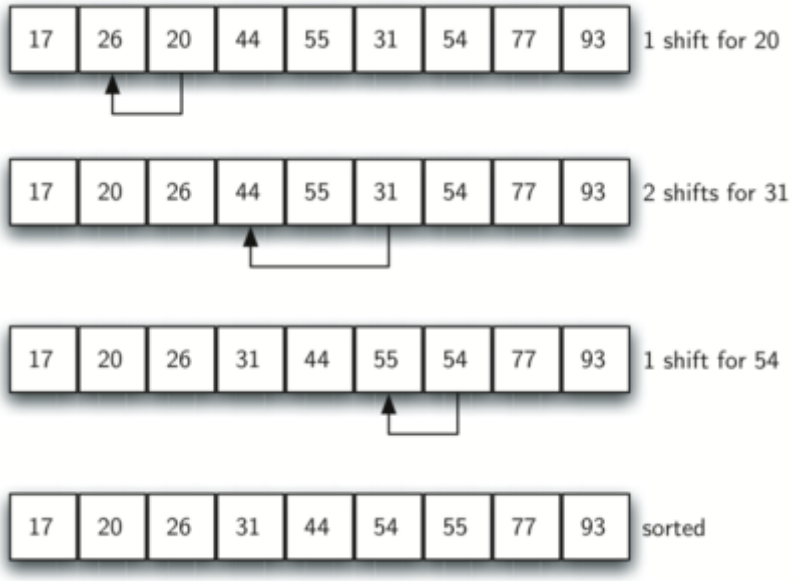


A Shell Sort with Increments of Three

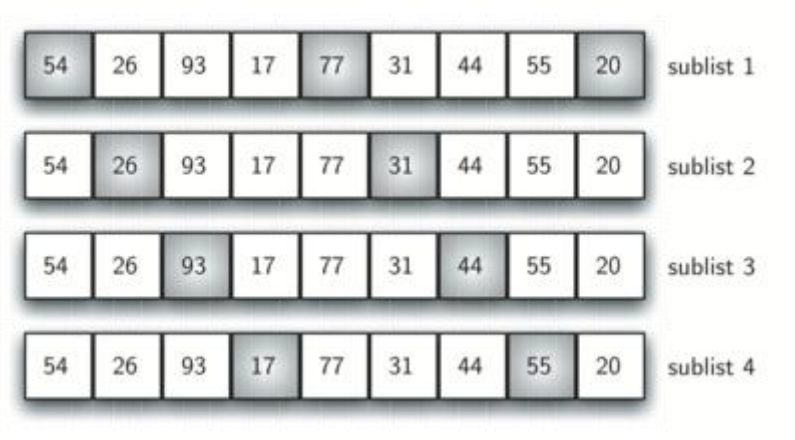


A Shell Sort after Sorting Each Sublist

Shows a final insertion sort using an increment of one; in other words, a standard insertion sort. Note that by performing the earlier sublist sorts, we have now reduced the total number of shifting operations necessary to put the list in its final order. For this case, we need only four more shifts to complete the process.



ShellSort: A Final Insertion Sort with Increment of 1



Initial Sublist for a Shell Sort

We said earlier that the way in which the increments are chosen is the unique feature of the shell sort. The function shown in ActiveCode 1 uses a different set of increments. In this case, we begin with n2 sublist. On the next pass, n4 sublist is sorted. Eventually, a single list is sorted with the basic insertion sort. Above figure shows the first sublist for our example using this increment.

Algorithm for Shell Sort:

```
void Shell_sort(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;
                        }
}
```

Bubble Sort Algorithm

Bubble sort algorithm starts by comparing the first two elements of an array and swapping if necessary, i.e., if you want to sort the elements of array in ascending order and if the first element is greater than second then, you need to swap the elements but, if the first element is smaller than second, you mustn't swap the element. Then, again second and third elements are compared and swapped if it is necessary and this process go on until last and second last element is compared and swapped. This completes the first step of bubble sort.

If there are *n* elements to be sorted then, the process mentioned above should be repeated *n-1* times to get required result. But, for better performance, in second step, last and second last elements are not compared because; the proper element is automatically placed at last after first step. Similarly, in third step, last and second last and second last and third last elements are not compared and so on.

A figure is worth a thousand words so; acknowledge this figure for better understanding of bubble sort.

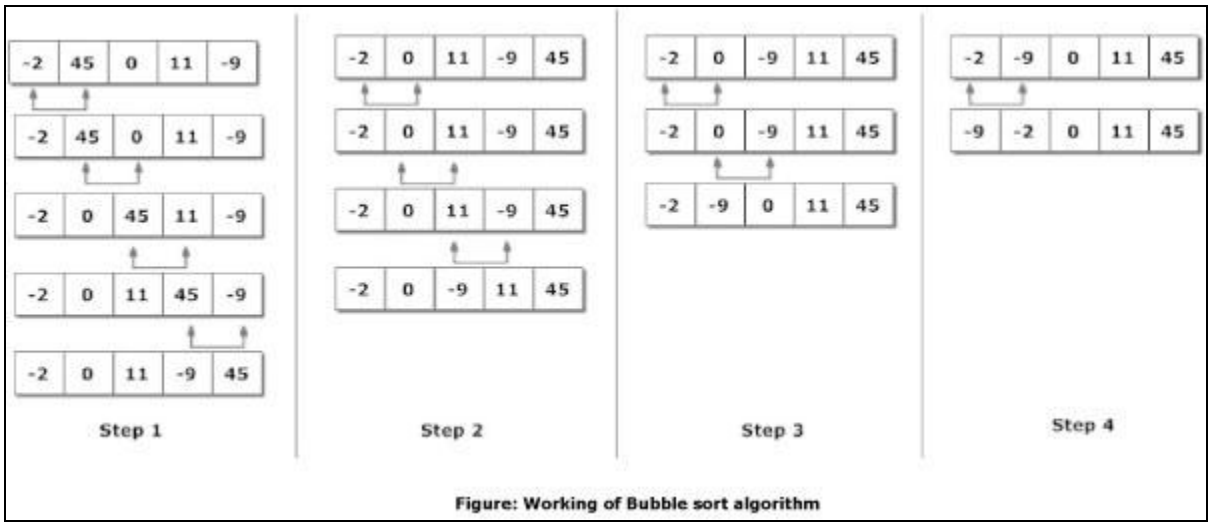


Figure: Working of Bubble sort algorithm

Here, there are 5 elements to be sorted. So, there are 4 steps.

Algorithm for Bubble Sort:

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

- For “n” elements, (n-1) passes are required.

- At the end of i^{th} pass, the i^{th} largest element will be placed in its correct position.

Quick Sort Algorithm

1. Quicksort is a divide and conquer algorithm in the style of merge sort.
2. The basic idea is to find a “pivot” item in the array and compare all other items with pivot element.
3. Shift items such that all of the items before the pivot are less than the pivot value and all the items after the pivot are greater than the pivot value.
4. After that, recursively perform the same operation on the items before and after the pivot. There are many different algorithms to achieve a quicksort and this post explores just one of them.
5. There are two basic operations in the algorithm, swapping items in place and partitioning a section of the array.

The basic steps to partition an array are:

1. Find a “pivot” item in the array. This item is the basis for comparison for a single round.
2. Start a pointer (the left pointer) at the first item in the array.
3. Start a pointer (the right pointer) at the last item in the array.
4. While the value at the left pointer in the array is less than the pivot value, move the left pointer to the right (add 1). Continue until the value at the left pointer is greater than or equal to the pivot value.
5. While the value at the right pointer in the array is greater than the pivot value, move the right pointer to the left (subtract 1). Continue until the value at the right pointer is less than or equal to the pivot value.
6. If the left pointer is less than or equal to the right pointer, then swap the values at these locations in the array.
7. Move the left pointer to the right by one and the right pointer to the left by one.
8. If the left pointer and right pointer don’t meet, go to step 1.

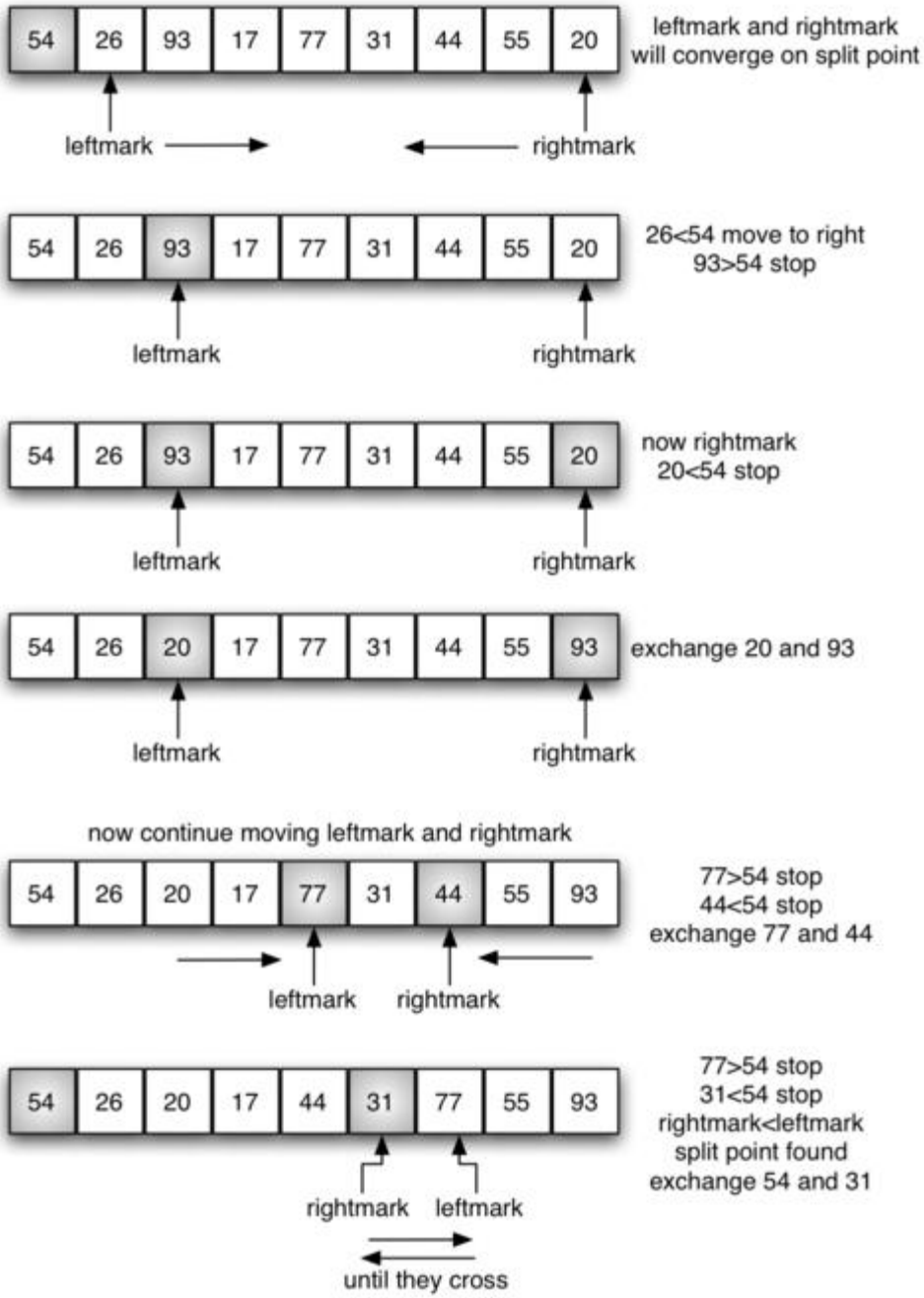
- The **quick sort** uses divide and conquer to gain the same advantages as the merge sort, while not using additional storage.
- A quick sort first selects a value, which is called the **pivot value**. Although there are many different ways to choose the pivot value, we will simply use the first item in the list.
- The role of the pivot value is to assist with splitting the list. The actual position where the pivot value belongs in the final sorted list, commonly called the **split point**, will be used to divide the list for subsequent calls to the quick sort.

Below figure shows that 54 will serve as our first pivot value. Since we have looked at this example a few times already, we know that 54 will eventually end up in the position currently holding 31. The **partition** process will happen next. It will find the split point and at the same time move other items to the appropriate side of the list, either less than or greater than the pivot value.

The First Pivot Value for a Quick Sort

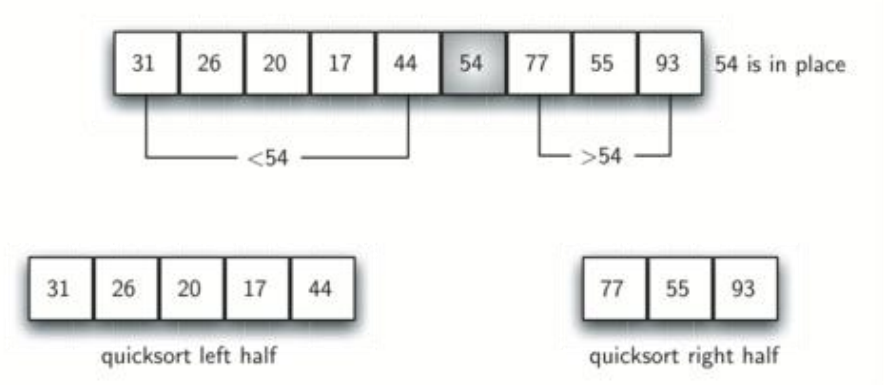
Partitioning begins by locating two position markers—let's call them leftmark and rightmark—at the beginning and end of the remaining items in the list (positions 1 and 8 in). The goal of the partition process is to move items that are on the wrong side with respect to the pivot value while also converging on the split point. Below figure shows this process as we locate the position of 54.

We begin by incrementing leftmark until we locate a value that is greater than the pivot value. We then decrement rightmark until we find a value that is less than the pivot value. At this point we have discovered two items that are out of place with respect to the eventual split point. For our example, this occurs at 93 and 20. Now we can exchange these two items and then repeat the process again.



Finding the Split Point for 54

At the point where rightmark becomes less than leftmark, we stop. The position of rightmark is now the split point. The pivot value can be exchanged with the contents of the split point and the pivot value is now in place. In addition, all the items to the left of the split point are less than the pivot value, and all the items to the right of the split point are greater than the pivot value. The list can now be divided at the split point and the quick sort can be invoked recursively on the two halves.



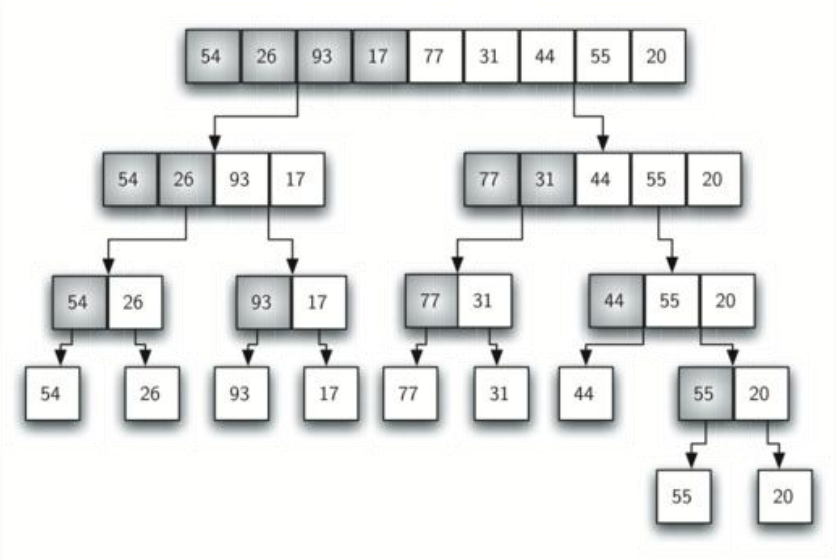
Completing the Partition Process to Find the Split Point for 54

Algorithm for Quick Sort:

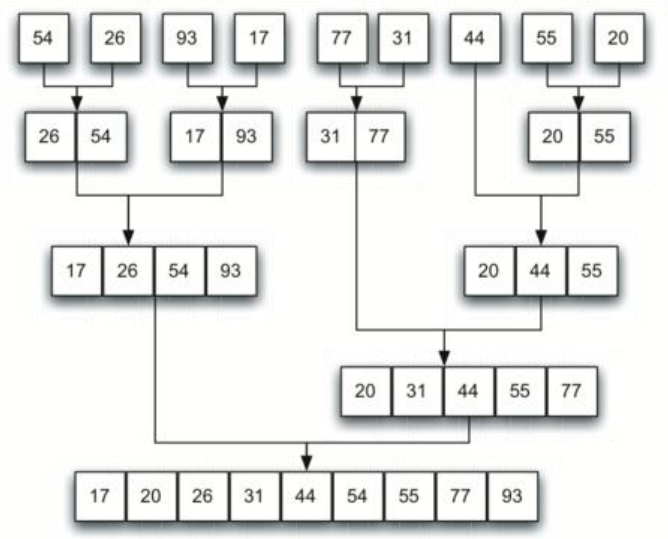
```
void Quicksort(int a[ ], int left, int right)
{
    int i, j, P, temp;
    if( left < right )
    {
            P = left;
            I = left + 1;
            J = right;
            while( i < j )
            {
                    while( a[i] <= a[P] )
                            I = I + 1;
                    while( a[j] > a[P] )
                            j = j - 1;
                    if( i < j )
                    {
                            temp = a[i];
                            a[i] = a[j];
                            a[j] = temp;
                    }
            }
            temp = a[P];
            a[P] = a[j];
            a[j] = temp;
            quicksort( A, left, j-1 );
            quicksort( A, j+1, right );
    } }
```

Merge Sort Algorithm

Merge sort is a recursive algorithm that continually splits a list in half. If the list has more than one item, we split the list and recursively invoke a merge sort on both halves. Once the two halves are sorted, the fundamental operation, called a **merge**, is performed. Merging is the process of taking two smaller sorted lists and combining them together into a single, sorted, new list. Below figure shows our familiar example list as it is being split by mergeSort. Below figure shows the simple lists, now sorted, as they are merged back together.



Splitting the List in a Merge Sort



List, as they are Merged together

Algorithm for Merge Sort:

```
void Merge_sort(int a[ ], int temp[ ], int n)
{
        msort( a, temp, 0, n-1 )
}
```

```
void msort( int a[ ], int temp[ ], int left, int right)
{
     int center;
     if( left < right )
     {
               center = ( left + right ) / 2;
               msort( a, left, center );
               msort( a, temp, center+1, right);
               merge(a, temp, n, left, center, right);
     }
}
```

```
void merge( int a[ ], int temp[ ], int n, int left, int center, int right )
{
        int i = 0, j, left_end = center, center = center+1;

        while( ( left <= left_end ) && ( center <= right ) )
        {
                if( a[left] <= a[center] )
                {
                        temp[i] = a[left];
                        i++;
                        left++;
                }
                else
                {
                        temp[i] = a[center];
                        i++;
                        center++;
                }
        }

        while( left <= left_end )
        {
                temp[i] = a[left];
                left++;
                i++;
        }

        while( center <= right )
        {
                temp[i] = a[center];
                center++;
                i++;
        }

        for( i=0; i<n; i++ )
        {
                print temp[i];
        }
}
```

Radix Sort

Steps1: Consider 10 buckets (1 for each digit 0 to 9)

Step2: Consider the LSB (Least Significant Bit) of each number (numbers in the one's Place…. E.g., in 43 LSB = 3)

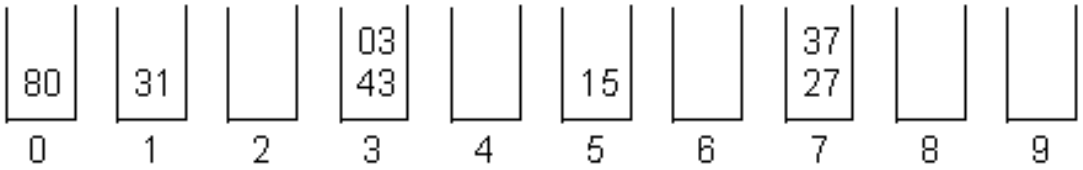
Step3: place the elements in their respective buckets according to the LSB of each number

Step4: write the numbers from the bucket (0 to 9) bottom to top.

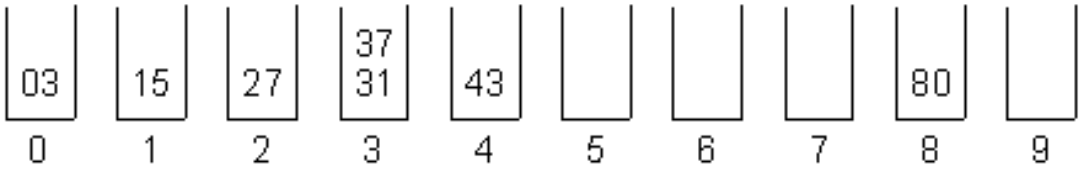
Step5: repeat the same process with the digits in the $10^{'s}$ place (e.g. In 43 MSB =4)

Step6: repeat the same step till all the digits of the given number are consider.

43 27 31 15 37 80 03



80 31 43 03 15 27 37



03 15 27 31 37 43 80

Algorithm for Radix Sort:

```
void Radix_sort( int a[ ], int n)
{
    int bucket[10][5], buck[10], b[10];
    int i, j, k, l, num, div, large, passes;
    div = 1;
    num = 0;
    large = a[0];
    for( i = 0; i < n; i++)
    {
        if( a[i] > large )
        {
            large = a[i];
        }
        while( large > 0 )
        {
            num++;
            large = large / 10;
        }
        for( passes = 0; passes < num; passes++ )
        {
            for( k = 0; k < 10; k++ )
            {
                buck[k] = 0;
            }
            for( i=0; i < n; i++ )
            {
                l = ( ( a[i] / div ) % 10 );
                bucket[l][buck[l]++] = a [i];
            }
            i = 0;
            for( k = 0; k < 10; k++)
            {
                for( j = 0; j < buck[k] ; j++ )
                {
                    a[i++] = bucket[k][j];
                }
            }
            div * = 10;
        }
    }
}
```

SEARCHING

Searching:

Searching is an algorithm, to check whether a particular element is present in the list.

Types of searching:-

- **Linear search**
- **Binary Search**

Linear Search

Linear search or sequential search is a method for finding a particular value in a list, that consists of checking every one of its elements, one at a time and in sequence, until the desired one is found.

Linear search is the simplest search algorithm. For a list with n items, the best case is when the value is equal to the first element of the list, in which case only one comparison is needed. The worst case is when the value is not in the list (or occurs only once at the end of the list), in which case n comparisons are needed.

The worst case performance scenario for a linear search is that it has to loop through the entire collection, either because the item is the last one, or because the item is not found.

Algorithm for Linear Search:

```
void Linear_search(int a[ ], int n)
{
        int search, count = 0;
        for( i = 0; i < n; i++ )
        {
                if( a[i] = = search )
                {
                        count++;
                }
        }
        if( count = = 0 )
                print  “Element not Present” ;
        else
                print “Element is Present in list";
}
```

Binary Search

It is possible to take greater advantage of the ordered list .Instead of searching the list in sequence, a binary search will start by examining the middle item. If that item is the one we are searching for, we are done. If it is not the correct item, we can use the ordered nature of the list to eliminate half of the remaining items. If the item we are searching for is greater than the middle item, we know that the entire lower half of the list as well as the middle item can be eliminated from further consideration. The item, if it is in the list, must be in the upper half.

We can then repeat the process with the upper half. Start at the middle item and compare it against what we are looking for. Again, we either find it or split the list in half, therefore eliminating another large part of our possible search space.

Working principle:

Algorithm is quite simple. It can be done either recursively or iteratively:

1. Get the middle element;
2. If the middle element equals to the searched value, the algorithm stops;
3. Otherwise, two cases are possible:
    - Searched value is less, than the middle element. In this case, go to the step 1 for the part of the array, before middle element.
    - Searched value is greater, than the middle element. In this case, go to the step 1 for the part of the array, after middle element.

Now we should define when iterations should stop. First case is when searched element is found. Second one is when subarray has no elements. In this case, we can conclude that search value is not present in the array.

Example 1.

Find 6 in {-1, 5, 6, 18, 19, 25, 46, 78, 102, 114}.

Step 1 (middle element is 19 > 6): -1 5 6 18 19 25 46 78 102 114

Step 2 (middle element is 5 < 6): -1 5 6 18 19 25 46 78 102 114

Step 3 (middle element is 6 == 6): -1 5 6 18 19 25 46 78 102 114

Algorithm for Binary Search:

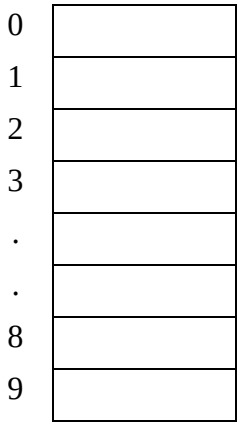
```
void Binary_search ( int a[ ], int n, int search )
{
        int first,  last,  mid;
        first = 0;
        last = n-1;
        mid = ( first + last ) / 2;
        while( first <= last )
        {
                if( Search > a[mid] )
                        first = mid + 1;
                else if( Search = = a[mid] )
                {
                        print “Element is present in the list";
                        break;
                }
                else
                        last = mid - 1;
                  mid = ( first + last ) / 2;
        }
        if( first > last )
                print “Element Not Found”;
}
```

HASHING

Hashing is a technique that is used to overcome the drawback of Linear Search (Comparison) & Binary Search (Sorted order).

- ✓ Technique that is used to store and retrieve data from data structure.
- ✓ It reduces the number of comparison.
- ✓ It involves two concepts-
  - ➢ Hash Function
  - ➢ Hash Table

Hash table



Simple Hash table with table size = 10

- A **hash table** is a data structure that is used to store and retrieve data very quickly.
- Insertion of the data in the hash table is based on the key value obtained from the hash function.
- Using same hash key, the data can be retrieved from the hash table by few or more Hash key comparison.

E.g. For storing student details, the student Id will work as a key.

Hash function:

- It is a function, which is used to put the data in the hash table.
- Using the same hash function we can retrieve data from the hash table.
- Hash function is used to implement hash table.
- **The integer value returned by the hash function is called hash key.**

Bucket:

- ✓ The hash function (H (key) = key % table size) is used to map several data's in the Hash table.
- ✓ Each position of the Hash Table is called Bucket.

Types of Hash Functions

1. Division Method
2. Mid Square Method
3. Multiplicative Hash Function
4. Digit Folding

1. Division Method:

- ✓ It depends on remainder of division.
- ✓ Divisor is Table Size.
- ✓ Formula is **(H (key) = key % table size)**

E.g. consider the following data or record or key (36, 18, 72, 43, 6) table size = 8

Assume a table with 8 slots:

Hash key = key % table size

4 = 36 % 8

2 = 18 % 8

0 = 72 % 8

3 = 43 % 8

6 = 6 % 8

[0]	72
[1]	
[2]	18
[3]	43
[4]	36
[5]	
[6]	6
[7]	

2. Mid Square Method:

We first square the item, and then extract some portion of the resulting digits. For example, if the item were 44, we would first compute 44^2=1,936. Extract the middle two digit 93 from the answer. Store the key 44 in the index 93.

0	
1	
2	
.	
93	44
.	
99	

3, Multiplicative Hash Function:

Key is multiplied by some constant value.

Hash function is given by,

H(key)=Floor (P * (key * A))

P = Integer constant [e.g. P=50]

A = Constant real number [A=0.61803398987]

E.g. Key 107

H(107)=Floor(50*(107*0.61803398987))

=Floor(3306.481845)

H(107)=3306

Consider table size is 5000

0	
1	
2	
.	
3306	107
.	
4999	

4, Digit Folding Method:

The folding method for constructing hash functions begins by dividing the item into equal-size pieces (the last piece may not be of equal size). These pieces are then added together to give the resulting hash key value. For example, if our item was the phone number 436-555-4601, we would take the digits and divide them into groups of 2 (43, 65, 55, 46, 01). After the addition, 43+65+55+46+01, we get 210. If we assume our hash table has 11 slots, then we need to perform the extra step of dividing by 11 and keeping the remainder. In this case 210 % 11 is 1, so the phone number 436-555-4601 hashes to slot 1.

0	
1	436-555-4601
2	
3	
.	
8	
9	
10	

Collision:

If two keys hashes to the same index, the corresponding records cannot be stored in the same location. So, if it's already occupied, we must find another location to store the new record.

Characteristics of Good Hashing Function:

- Simple to compute.
- Number of Collision should be less while placing record in Hash Table.
- **Hash function with no collision ➔ Perfect hash function.**
- Hash Function should produce keys which are distributed uniformly in hash table.

Collision Resolution Strategies / Techniques (CRT):

If collision occurs, it should be handled or overcome by applying some technique. Such technique is called CRT.

There are a number of collision resolution techniques, but the most popular are:

- **Separate chaining (**Open Hashing**)**
- **Open addressing**. (Closed Hashing)
  - **Linear Probing**
  - **Quadratic Probing**
  - **Double Hashing**

Separate chaining (Open Hashing**)**

- Open hashing technique.
- Implemented using singly linked list concept.
- Pointer (ptr) field is added to each record.
- When collision occurs, a separate chaining is maintained for colliding data.
- Element inserted in front of the list.

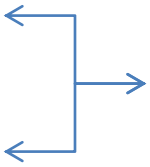
  H (key) =key % table size

Two operations are there:-

- Insert
- Find

Structure Definition for Node

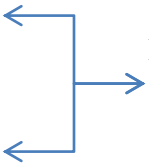
```
typedef Struct node *Position;
Struct node
{
        int data;
        Position  next;
};
```



defines the nodes

Structure Definition for Hash Table

```
typedef  Position  List;
struct Hashtbl
{
        int Tablesize;
        List * theLists;
};
```



Defines the hash table which contains array of linked list

Initialization for Hash Table for Separate Chaining

```
Hashtable  initialize(int Tablesize)
{
        HashTable  H;
        int i;
        H = malloc (sizeof(struct HashTbl));                    ➔Allocates table
        H → Tablesize = NextPrime(Tablesize);
        H→the Lists=malloc(sizeof(List) * H→Tablesize);     ➔ Allocates array of list
        for( i = 0; i < H → Tablesize; i++ )
        {
        H → TheLists[i] = malloc(Sizeof(Struct node));     ➔ Allocates list headers
        H → TheLists[i] → next = NULL;
        }
        return H;
}
```

Insert Routine for Separate Chaining

```
void insert (int Key, Hashtable H)
{
  Position P, newnode;                    *[Inserts element in the Front of the list always]*
  List L;
  P = find ( key, H );
  if(P = = NULL)
   {
        newnode = malloc(sizeof(Struct node));
        L = H → TheLists[Hash(key,Tablesize)];
        newnode → nex t= L → next;
        newnode → data = key;
       L → next = newnode;
    }
}

Position find( int key, Hashtable H)
{
  Position P, List L;
  L = H →TheLists[Hash(key,Tablesize)];
  P = L → next;
  while(P != NULL && P → data != key)
        P = P → next;
  return P;
}
```

If two keys map to same value, the elements are chained together.

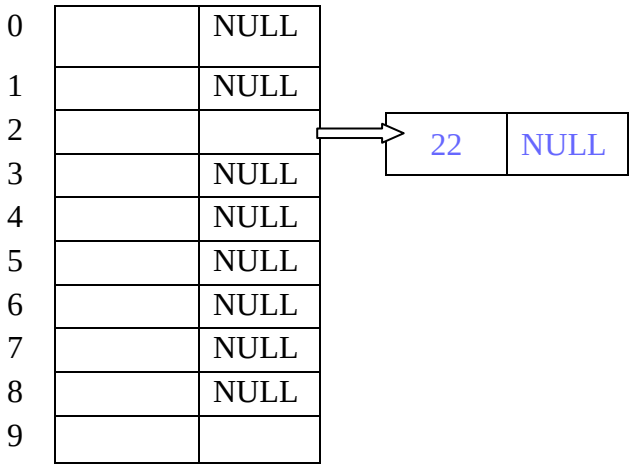
Initial configuration of the hash table with separate chaining. Here we use SLL(Singly Linked List) concept to chain the elements.

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

Insert the following four keys 22 84 35 62 into hash table of size 10 using separate chaining.

The hash function is
H(key) = key % 10

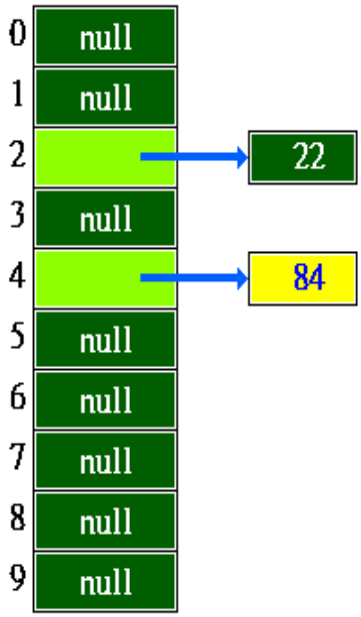
1. H(22) = 22 % 10
   = 2



Insert the following four keys 22 84 35 62 into hash table of size 10 using separate chaining

The hash function is key % 10

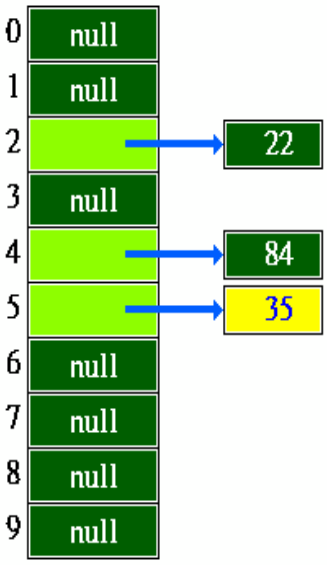
84 % 10 = 4



Insert the following four keys 22 84 35 62 into hash table of size 10 using separate chaining

The hash function is key % 10

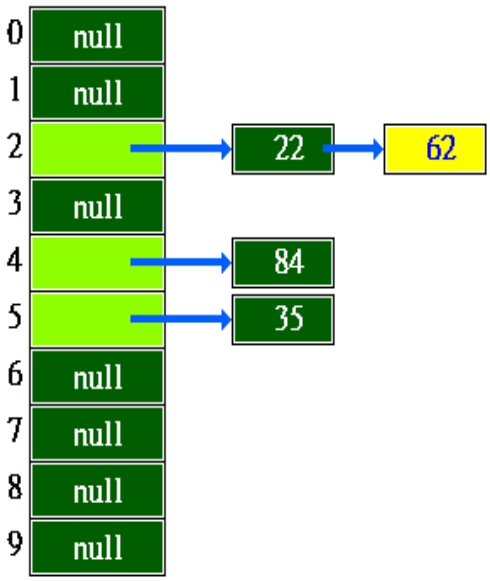
35 % 10 = 5



Insert the following four keys 22 84 35 62 into hash table of size 10 using separate chaining

The hash function is key % 10

62 % 10 = 2



Advantage

1, more number of elements can be inserted using array of Link List

Disadvantage

1, it requires more pointers, which occupies more memory space.

2, Search takes time. Since it takes time to evaluate Hash Function and also to traverse the List

Open Addressing

- Closed Hashing
- Collision resolution technique
- When collision occurs, alternative cells are tried until empty cells are found.
- Types:-
    - Linear Probing

    - Quadratic Probing
    - Double Hashing
- Hash function
    - H(key) = key % table size.
- Insert Operation
    - To insert a key; Use the hash function to identify the list to which the element should be inserted.
    - Then traverse the list to check whether the element is already present.
    - If exists, increment the count.
    - Else the new element is placed at the front of the list.

Linear Probing:

Easiest method to handle collision.

Apply the hash function H (key) = key % table size

How to Probing:

- first probe – given a key k, hash to H(key)
- second probe – if H(key)+f(1) is occupied, try H(key)+f(2)
- And so forth.

Probing Properties:

- we force f(0)=0
- The i^{th} probe is to (H (key) +f (i)) %table size.
- If I reach size-1, the probe has failed.
- Depending on f (), the probe may fail sooner.
- Long sequences of probe are costly.

Probe Sequence is:

- H (key) % table size
- H (key)+1 % Table size
- H (Key)+2 % Table size
- ……

1. **H(Key)=Key mod Tablesize**

    This is the common formula that you should apply for any hashing

    If collocation occurs use Formula 2

2. **H(Key)=(H(key)+i) Tablesize**

Where i=1, 2, 3, …… etc

Example: - 89 18 49 58 69; Tablesize=10

1. H(89) =89%10
   =9
2. H(18) =18%10
   =8
3. H(49) =49%10

   =9 ((coloids with 89.So try for next free cell using formula 2))

   i=1 h1(49) = (H(49)+1)%10
   = (9+1)%10
   =10%10
   =0

4. H(58) =58%10
   =8 ((colloids with 18))

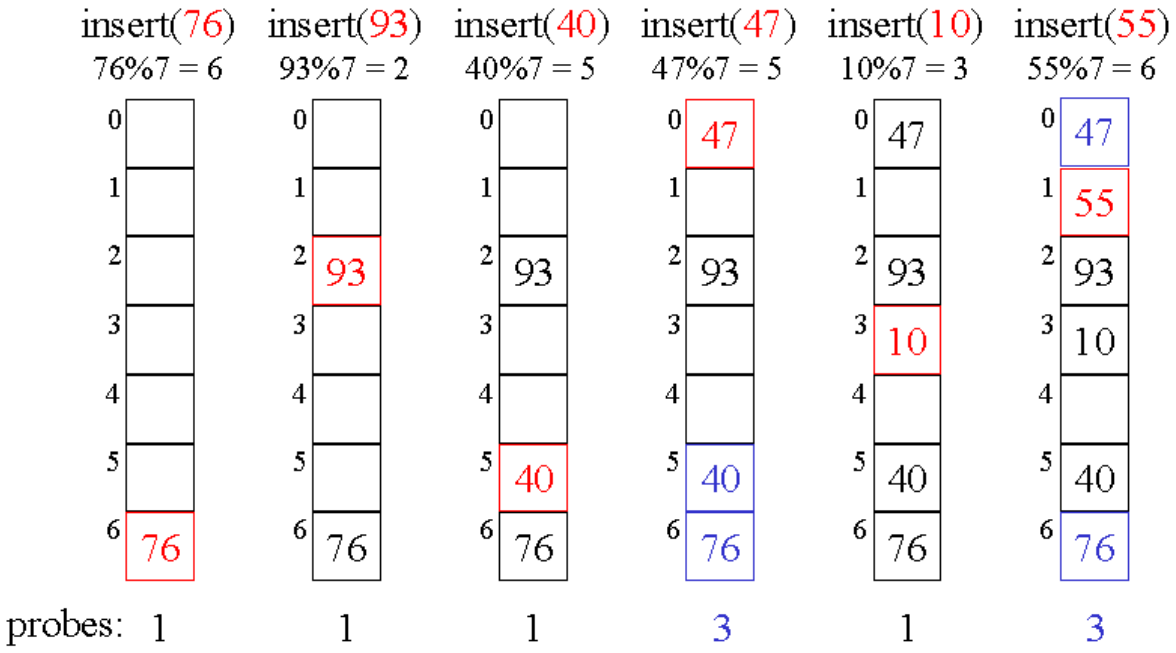
   i=1 h1(58) = (H(58) +1)%10
   = (8+1) %10
   =9%10
   =9 =>Again collision

   i=2 h2(58) =(H(58)+2)%10
   =(8+2)%10
   =10%10
   =0 =>Again collision

	EMPTY	89	18	49	58	69
0				49	49	49
1					58	58
2						69

3						
4						
5						
6						
7						
8			18	18	18	
9		89	89	89	89	

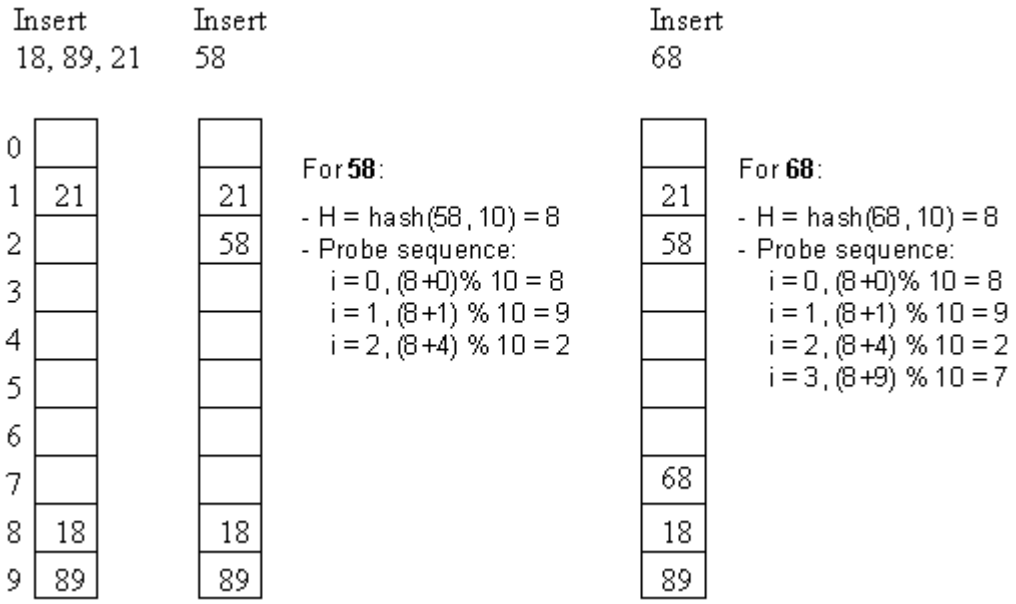
Linear Probing Example



Hashing with Quadratic Probe

To resolve the primary clustering problem, quadratic probing can be used. With quadratic probing, rather than always moving one spot, move i2 spots from the point of collision, where i is the number of attempts to resolve the collision.

- Another collision resolution method which distributes items more evenly.
- From the original index H, if the slot is filled, try cells H+12, H+22, H+32,.., H + i2 with wrap-around.



Limitation: at most half of the table can be used as alternative locations to resolve collisions.

This means that once the table is more than half full, it's difficult to find an empty spot. This new problem is known as secondary clustering because elements that hash to the same hash key will always probe the same alternative cells.

Hashing with Double Hashing

Double hashing uses the idea of applying a second hash function to the key when a collision occurs. The result of the second hash function will be the number of positions forms the point of collision to insert.

There are a couple of requirements for the second function:

It must never evaluate to 0 must make sure that all cells can be probed

A popular second hash function is:

Hash2 (key) = R - (key % R) where R is a prime number that is smaller than the size of the table.

Table Size = 10 elements
$Hash_1(key) = key \% 10$
$Hash_2(key) = 7 - (k \% 7)$

Insert keys : 89, 18, 49, 58, 69

Hash(89) = 89 % 10 = 9

Hash(18) = 18 % 10 = 8

Hash(49) = 49 % 10 = 9 a collision !
= 7 – (49 % 7)
= 7 positions from [9]

Hash(58) = 58 % 10 = 8
= 7 – (58 % 7)
= 5 positions from [8]

Hash(69) = 69 % 10 = 9
= 7 – (69 % 7)
= 1 position from [9]

[0]	49
[1]	
[2]	
[3]	69
[4]	
[5]	
[6]	
[7]	58
[8]	18
[9]	89

Hashing with Rehashing

Once the hash table gets too full, the running time for operations will start to take too long and may fail. To solve this problem, a table at least twice the size of the original will be built and the elements will be transferred to the new table.

Advantage:

- A programmer doesn't worry about table system.
- Simple to implement
- Can be used in other data structure as well

The new size of the hash table:

- should also be prime
- will be used to calculate the new insertion spot (hence the name rehashing)
- This is a very expensive operation! O(N) since there are N elements to rehash and the table size is roughly 2N. This is ok though since it doesn't happen that often.

The question becomes when should the rehashing be applied?

Some possible answers:

- once the table becomes half full
- once an insertion fails
- once a specific load factor has been reached, where load factor is the ratio of the number of elements in the hash table to the table size

Extendible Hashing

- Extendible Hashing is a mechanism for altering the size of the hash table to accommodate new entries when buckets overflow.
- Common strategy in internal hashing is to double the hash table and rehash each entry. However, this technique is slow, because writing all pages to disk is too expensive.

- Therefore, instead of doubling the whole hash table, we use a directory of pointers to buckets, and double the number of buckets by doubling the directory, splitting just the bucket that overflows.
- Since the directory is much smaller than the file, doubling it is much cheaper. Only one page of keys and pointers is split.

000 100
010 100
100 000
111 000
001 000
011 000
101 000
111 001
001 010
101 100
101 110