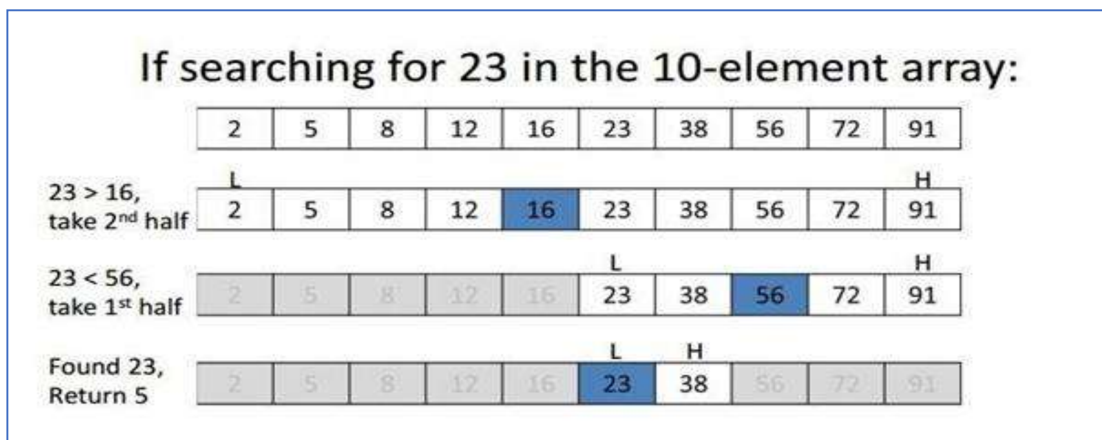
BINARY SEARCH

Search a sorted array by repeatedly dividing the search interval in half. Begin with an interval covering the whole array. If the value of the search key is less than the item in the middle of the interval, narrow the interval to the lower half. Otherwise narrow it to the upper half. Repeatedly check until the value is found or the interval is empty.

Example:



www.binils.com

- We basically ignore half of the elements just after one comparison.

Compare x with the middle element.

- If x matches with middle element, we return the mid index.
- Else If x is greater than the mid element, then x can only lie in right half subarray after the mid element. So we recur for right half.
- Else (x is smaller) recur for the left half.

Program to implement recursive Binary Search

```
#include <stdio.h>

// A recursive binary search function. It returns
```

Download Binils Android App in Playstore Download Photoplex App

```
// location of x in given array arr[l..r] is present,

// otherwise -1

int binarySearch (int arr[], int l, int r, int x)

{

if (r >= l)

{

int mid = l + (r - l)/2;

// If the element is present at the middle

// itself

if (arr[mid] ==

x) return mid;

// If element is smaller than mid, then

// it can only be present in left subarray if (arr[mid]

> x) return binarySearch (arr, l, mid-1, x);

// Else the element can only be present

// in right subarray returnbinarySearch(arr, mid+1, r, x);

}

// We reach here when element is not

// present in array return -1;

}
```

Binary search is a fast search algorithm with run-time complexity of O(log n).

www.binils.com

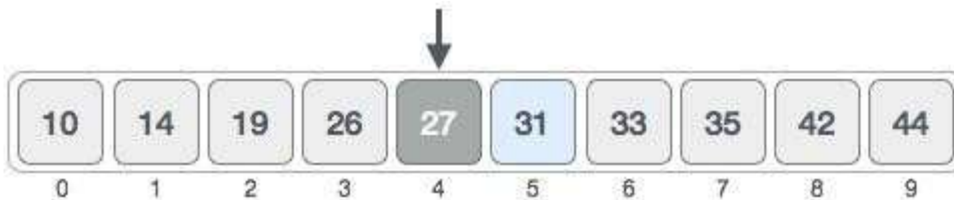
How Binary Search Works?

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.

www.binils.com

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

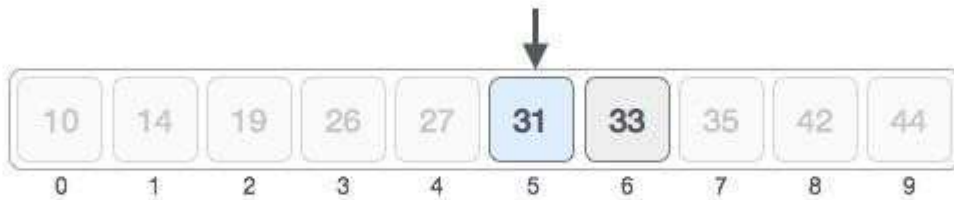
again. low = mid + 1

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.



www.binils.com

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.

HASHING

Hashing is a technique used for performing insertion, deletions, and finds in constant average time. The Hash table data structure is an array of some fixed size, containing the keys. A key is a value associated with each record.

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) = KEY VALUE MOD TABLE SIZE

Example: Hash (92)

Hash (92) = 92 mod 10 =2

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

www.binils.com

ROUTINE FOR SIMPLE HASH FUNCTION:

```
Hash (const char *key, intTableSize )

{

        int HashVal = 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

Collision occurs when a hash value of a record being inserted hashes to an address (i.e. Relative position) that already contain a different record. (ie) When two key values hash to the same position.

Collision Resolution

The process of finding another position for the collide record. Some of the collision Resolution Techniques

www.binils.com

1. Separate chaining
2. Open Addressing
3. Double Hashing

Separate 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.

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 turns to be a new one, it is inserted either at the front of the list or at the end of the list. If it is duplicate element, an extra field is kept a d placed.

Advantages:

1) Simple to implement.

2) Hash table never fills up, we can always add more elements to chain.

3) Less sensitive to the hash function or load factors.

4) It is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.

www.binils.com

Disadvantages:

1) Cache performance of chaining is not good as keys are stored using linked list. Open addressing provides better cache performance as everything is stored in same table.

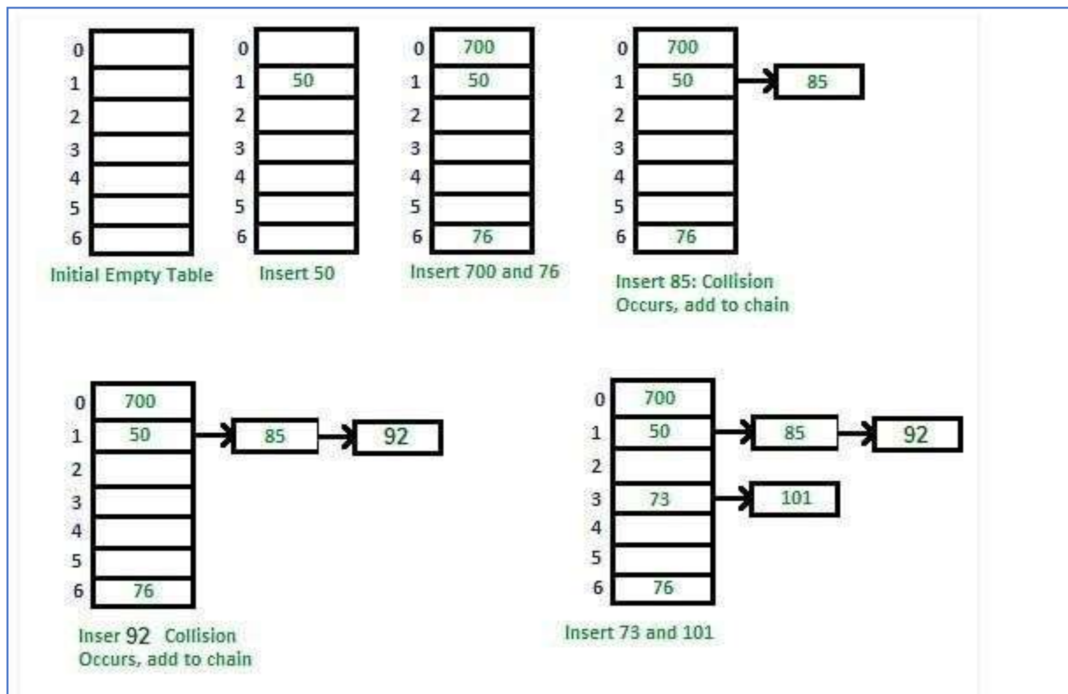
2) Wastage of Space (Some Parts of hash table are never used)

3) If the chain becomes long, then search time can become O(n) in worst case.

4) Uses extra space for links

Example:

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



OPEN ADDRESSING:

www.binils.com

Like separate chaining, open addressing is a method for handling collisions. In Open Addressing, all elements are stored in the hash table itself. So at any point, size of the table must be greater than or equal to the total number of keys (Note that we can increase table size by copying old data if needed).

Insert(k): Keep probing until an empty slot is found. Once an empty slot is found, insert k.

Search(k): Keep probing until slot‟s key doesn‟t become equal to k or an empty slot is

reached.

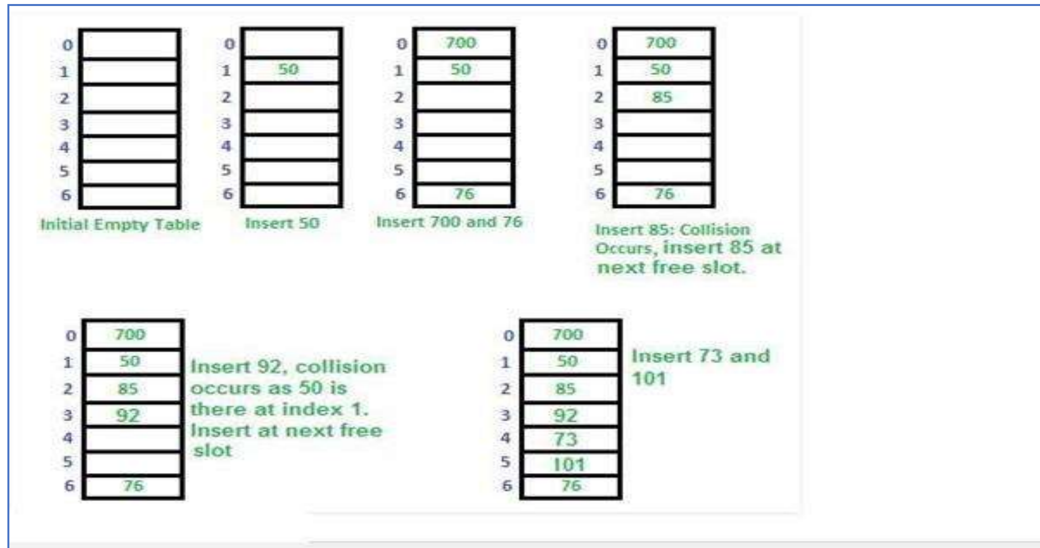
Delete(k): Delete operation is interesting. If we simply delete a key, then search may fail. So slots of deleted keys are marked specially as "deleted".

Insert can insert an item in a deleted slot, but the search doesn‟t stop at a deleted slot.

LINEAR PROBING:

In linear probing, we linearly probe for next slot. For example, typical gap between two probes is 1 as taken in below example also.

let hash(x) be the slot index computed using hash function and S be the table size If slot hash(x) % S is full, then we try (hash(x) + 1) % S
If (hash(x) + 1) % S is also full, then we try (hash(x) + 2) % S If (hash(x) + 2) % S is also full, then we try (hash(x) + 3) % S Let us consider a simple hash function as “key mod 7” and sequence of keys as 50, 700, 76, 85, 92, 73, 101.



www.binils.com

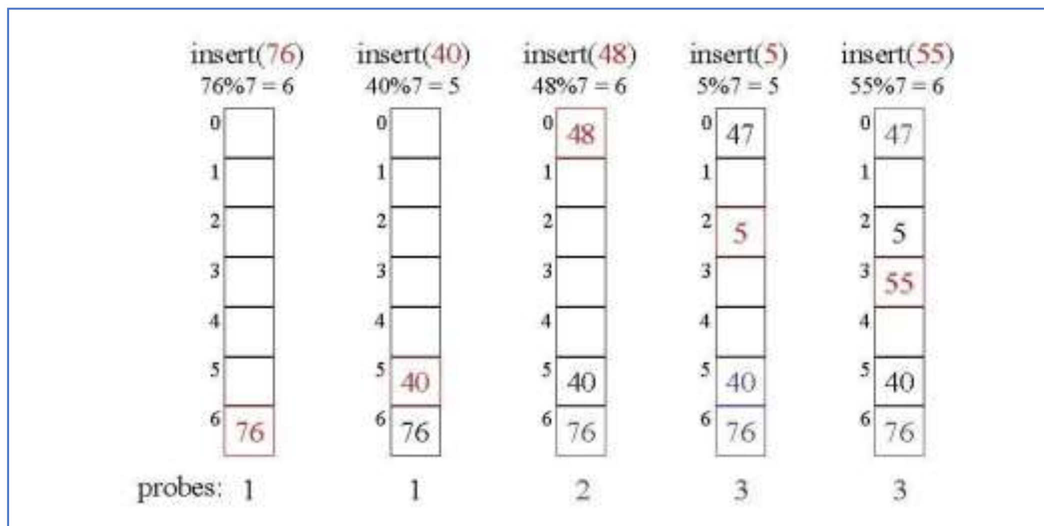
QUADRATIC PROBING

Quadratic probing is similar to linear probing and the only difference is the interval between successive probes or entry slots. Here, when the slot at a hashed index for an entry record is already occupied, you must start traversing until you find an unoccupied slot. The interval between slots is computed by adding the successive value of an arbitrary polynomial in the original hashed index.

Let us assume that the hashed index for an entry is index and at index there is an occupied slot. The probe sequence will be as follows:

index = index % hashTableSize
index = (index + 1) %
hashTableSize index = (index +
4) % hashTableSize index =
(index + 9) % hashTableSize

Quadratic Probing Example



www.binils.com

DOUBLE HASHING

Double hashing is similar to linear probing and the only difference is the interval between successive probes. Here, the interval between probes is computed by using two hash functions.

Let us say that the hashed index for an entry record is an index that is computed by one hashing function and the slot at that index is already occupied. You must start traversing in a specific probing sequence to look for an unoccupied slot.

The Double Hashing is:

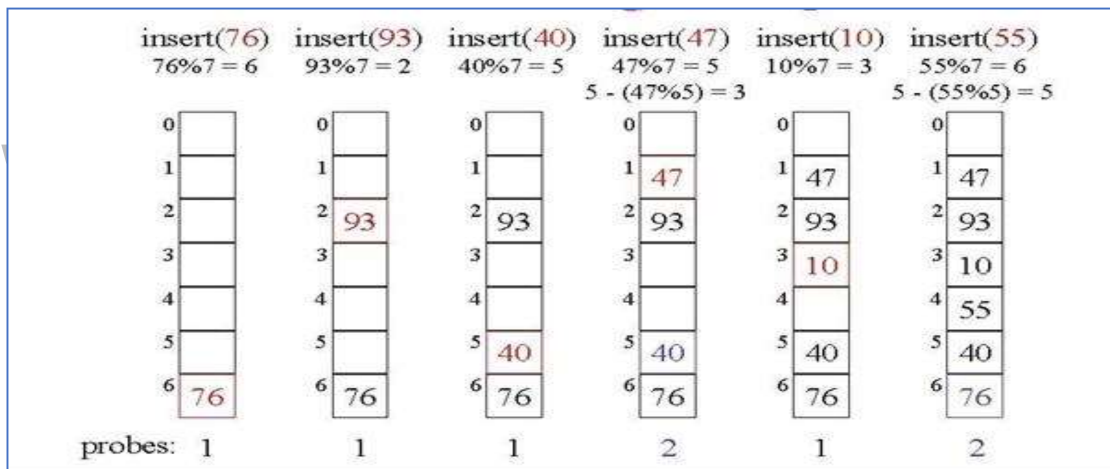
f(i) = i . hash2 (x)

Where hash2(X) =R – (X mod R) To choose a prime

R < size The probing sequence will be

– $h_1(k) \bmod size$
– $(h_1(k) + 1 \cdot h_2(x)) \bmod size$
– $(h_1(k) + 2 \cdot h_2(x)) \bmod size$
–

Example:



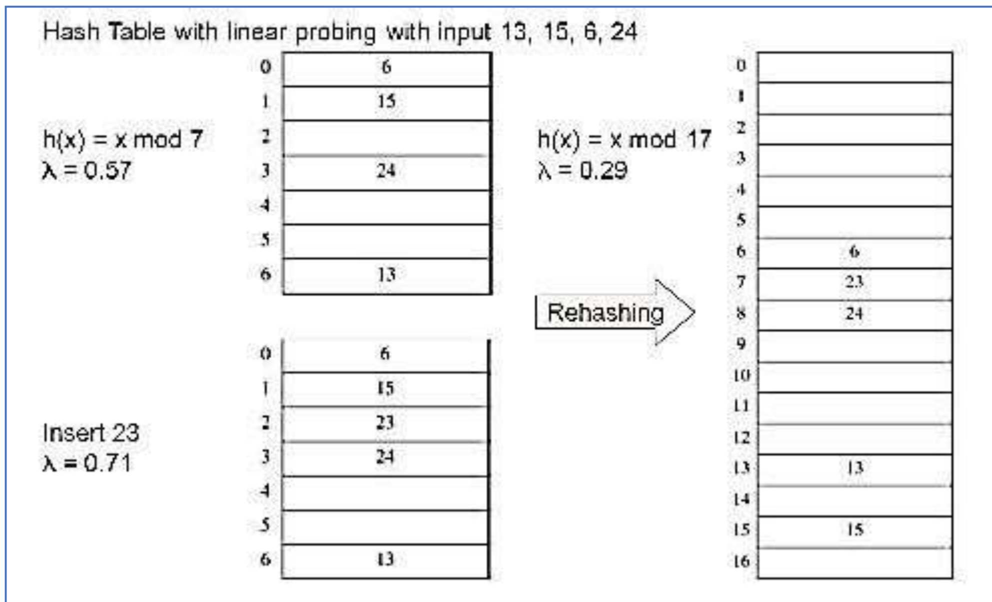
REHASHING

If the table is close to full, the search time grows and may become equal to the table size. When the load factor exceeds a certain value (e.g. greater than 0.5) we do rehashing: Build a second table twice as large as the original

and rehash there all the keys of the original table.

Rehashing is expensive operation, with running time O(N)

However, once done, the new hash table will have good performance.



EXTENDIBLE HASHING

www.binils.com

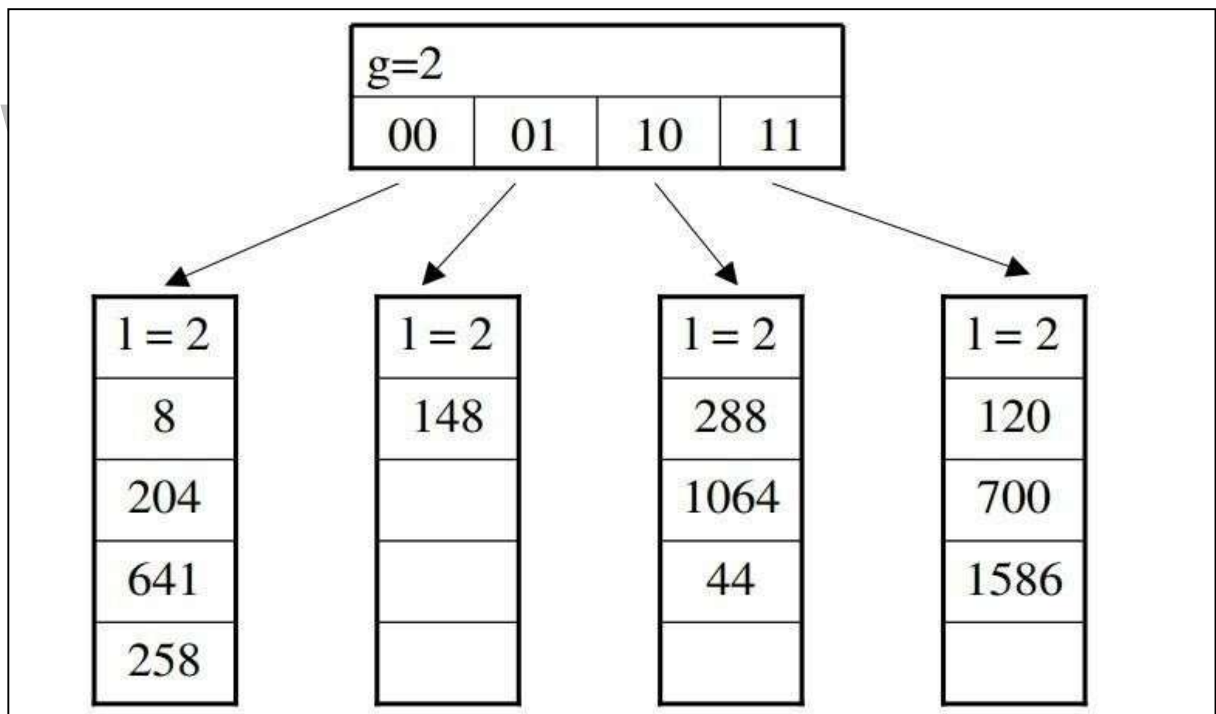
- Used when the amount of data is too large to fit in main memory and external storage is used.
- N records in total to store, M records in one disk block
- The problem: in ordinary hashing several disk blocks may be examined to find an element - a time consuming process.
- Extendible hashing: no more than two blocks are examined.
- Idea: Keys are grouped according to the first m bits in their code.
- Each group is stored in one disk block.
- If the block becomes full and no more records can be inserted, each group is split into two, and m+1 bits are considered to determine the location of a record.

Extendible Hashing Example

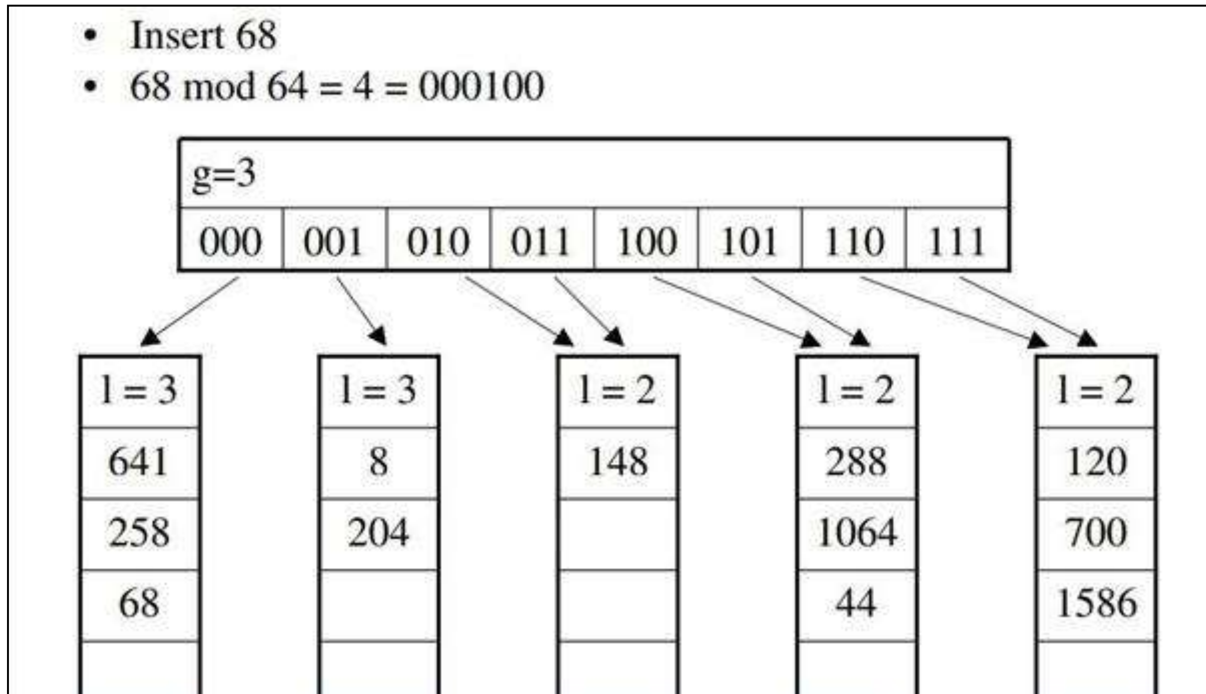
• Suppose that g=2 and bucket size = 4.

- Suppose that we have records with these keys and hash function h(key) = key mod 64:

key	h(key) = key mod 64	bit pattern
288	32	100000
8	8	001000
1064	40	101000
120	56	111000
148	20	010100
204	12	001100
641	1	000001
700	60	111100
258	2	000010
1586	50	110010
44	44	101010

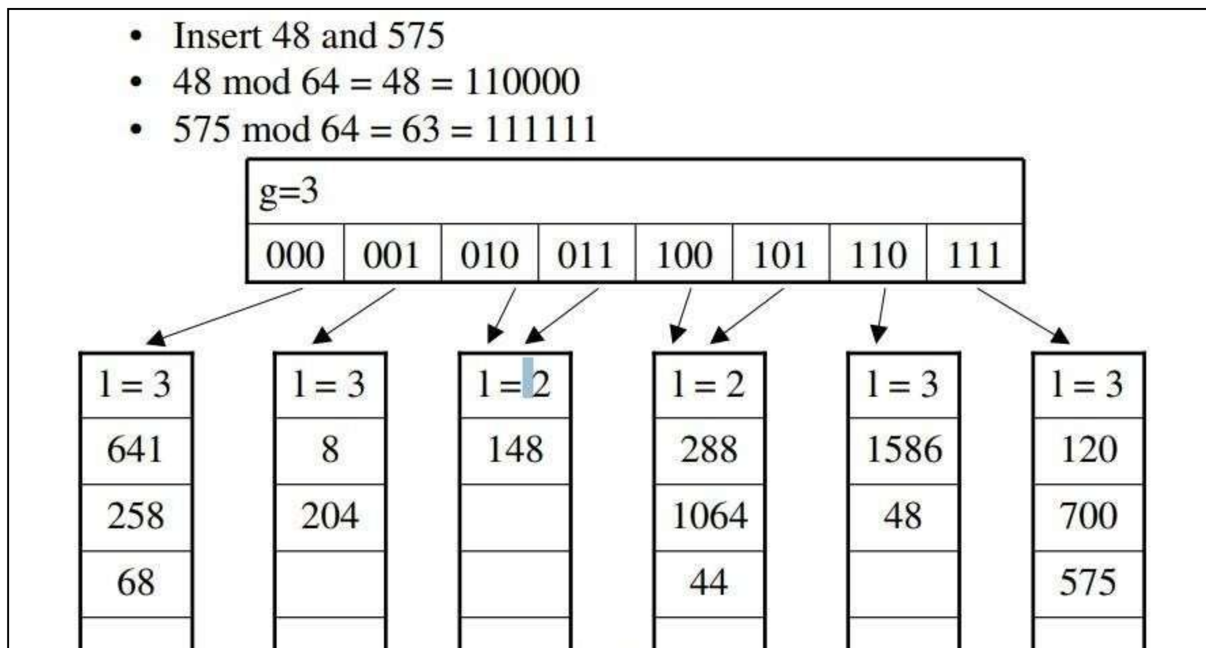


Bucket and directory split



Bucket split – no directory split

www.binils.com

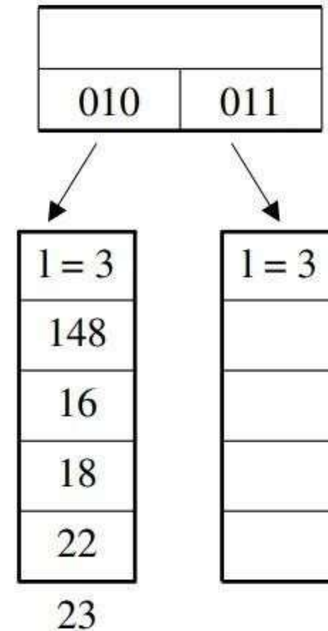


Multiple splits

- Insert 16, 18, 22, 23
- 16 mod 64 = 16 = 010000
- 18 mod 64 = 18 = 010010
- 22 mod 64 = 22 = 010110
- 23 mod 64 = 23 = 010111

Setting l=3 gives this intermediate (partial) picture…

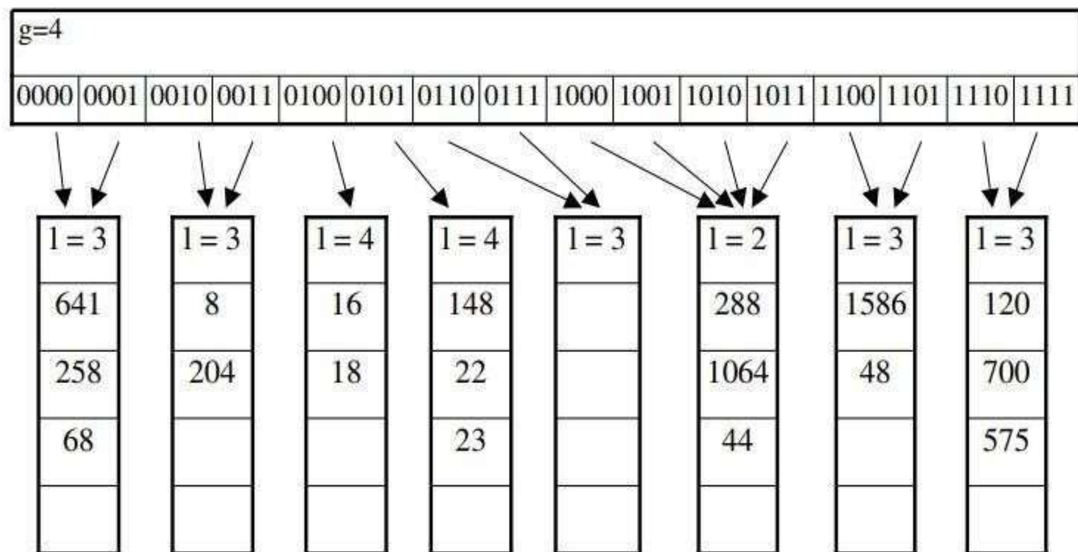
Continue to next page…



www.binils.com

Multiple splits, continued

- Setting l=4 (and thus g=4) gives this final result…



INSERTION SORT:

- Insertion sort is a simple sorting algorithm that works the way we sort playing cards in our hands.
- Insertion sorts works by taking element from the list one by one and inserting them in their current position into a new sorted list.
- Insertion sort is a sorting algorithm that places an unsorted element at its suitable place in each iteration.
- This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted.

  For example, the lower part of an array is maintained to be sorted. An element which is to be 'insert'ed in this sorted sub-list, has to find its appropriate place and then it has to be inserted there. Hence the name, insertion sort.
- The array is searched sequentially and unsorted items are moved and inserted into the sorted sub-list (in the same array). This algorithm is not suitable for large data sets as its average and worst case complexity are of O(n2), where n is the number of items.

Example:

www.binils.com

Consider an unsorted array as

follows, 20 10 60 40 30 15

ORIGINAL	20	10	60	40	30	10	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	

Example-2

How Insertion Sort Works?

We take an unsorted array for our example.

Insertion sort compares the first two elements.

It finds that both 14 and 33 are already in ascending order. For now, 14 is in sorted sub-list.

Insertion sort moves ahead and compares 33 with 27.

And finds that 33 is not in the correct position.

It swaps 33 with 27. It also checks with all the elements of sorted sub-list. Here we see that the sorted sub-list has only one element 14, and 27 is greater than 14. Hence, the sorted sub-list remains sorted after swapping.

www.binils.com

By now we have 14 and 27 in the sorted sub-list. Next, it compares 33 with 10.

These values are not in a sorted order.

So we swap them.

However, swapping makes 27 and 10 unsorted.

Hence, we swap them too.

Again we find 14 and 10 in an unsorted order.

We swap them again. By the end of third iteration, we have a sorted sub-list of 4 items.

www.binils.com

Algorithm

Now we have a bigger picture of how this sorting technique works, so we can derive simple steps by which we can achieve insertion sort.

Step 1 − If it is the first element, it is already sorted.

return 1; Step 2 − Pick next element

Step 3 − Compare with all elements in the sorted sub-list

Step 4 − Shift all the elements in the sorted sub-list that is greater than the value to be sorted

Download Binils Android App in Playstore Download Photoplex App

Step 5 − Insert the value

Step 6 − Repeat until list is sorted

INSERTION SORT ROUTINE

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

www.binils.com

Searching-Linear Search

Searching in data structure refers to the process of finding a desired element in set of items. The desired element is called "target". The set of items to be searched in, can be any data-structure like − list, array, linked-list, tree or graph.

LINEAR SEARCH

Linear search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.

A simple approach is to do linear search, i.e

- Start from the leftmost element of arr[] and one by one compare x with each element of arr[] If x matches with an element, return the index.
- If x doesn"t match with any of elements, return -1.

Example:

www.binils.com

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

Program:

```
#include <stdio.h>

int main()

{

int array[100], search, c, n;

printf("Enter the number of elements in array\n");

scanf("%d", &n);

printf("Enter %d integer(s)\n", n);

for (c = 0; c < n; c++)

scanf("%d", &array[c]);

printf("Enter a number to

search\n"); for (c = 0; c < n; c++)


{

if (array[c] == search) /* If required element is found */

{

printf("%d is present at location %d.\n", search, c+1);
```

www.binils.com

```
break;
}
}
if (c == n)
printf("%d isn't present in the array.\n", search);
return 0;
}
```

www.binils.com

SELECTION SORT

In selection sort, the first element in the list is selected and it is compared repeatedly with remaining all the elements in the list. If any element is smaller than the selected element (for Ascending order), then both are swapped. Then we select the element at second position in the list and it is compared with remaining all elements in the list. If any element is smaller than the selected element, then both are swapped. This procedure is repeated till the entire list is sorted.

The selection sort algorithm is performed using following steps...

Step 1: Select the first element of the list (i.e., Element at first position in

the list). Step 2: Compare the selected element with all other elements in

the list.

Step 3: For every comparison, if any element is smaller than selected element (for Ascending order), then these two are swapped.

Step 4: Repeat the same procedure with next position in the list till the entire list is sorted.

www.binils.com

Example:

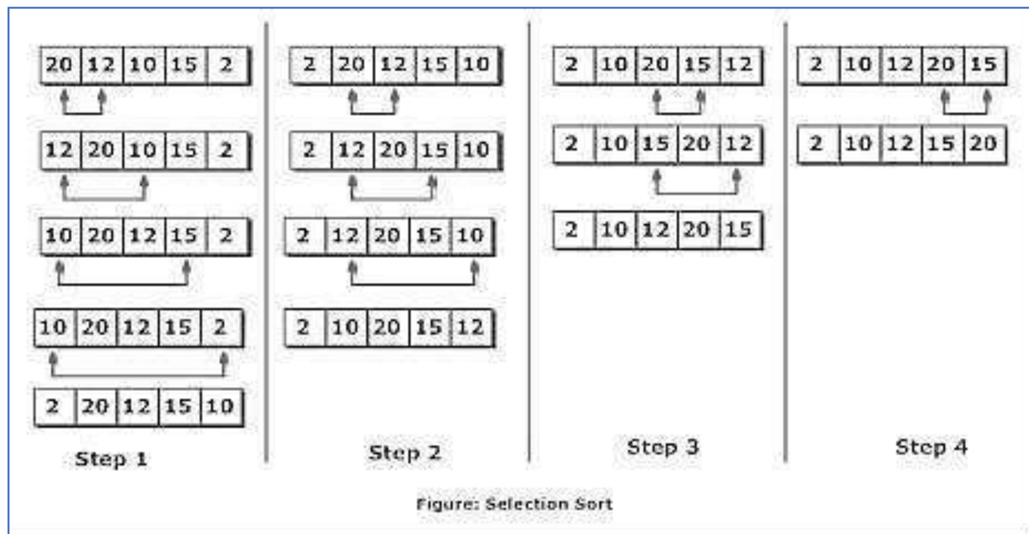


Figure: Selection Sort

Selection sort Routine

```
for(i=0; i<size; i++)
{
        for(j=i+1; j<size; j++)
```

```
            {
                if(list[i] > list[j])
                {
                temp=list[i]; list[i]=list[j]; list[j]=temp;
                }
            }
}
```

Example

www.binils.com

For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.

So we replace 14 with 10. After one iteration 10, which happens to be the minimum value in the list, appears in the first position of the sorted list.

For the second position, where 33 is residing, we start scanning the rest of the list in a linear manner.

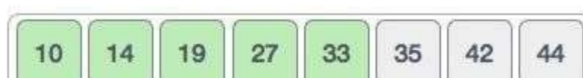
We find that 14 is the second lowest value in the list and it should appear at the second place. We swap these values.

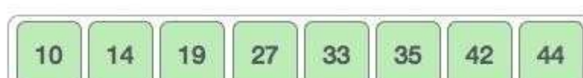
After two iterations, two least values are positioned at the beginning in a sorted manner.

The same process is applied to the rest of the items in the array.

Following is a pictorial depiction of the entire sorting process –

www.binils.com





SHELL SORT

The shell sort, sometimes called the "diminishing increment sort," improves on the insertion sort by breaking the original list into a number of smaller sub lists, each of which is sorted using an insertion sort. The unique way that these sub lists are chosen is the key to the shell sort. Instead of breaking the list into sub lists of contiguous items, the shell sort uses an increment i, sometimes called the gap, to create a sub list by choosing all items that are i items apart.

Example:

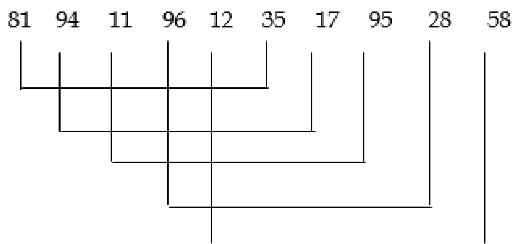
Consider an unsorted array as

follows. 81 94 11 96 12 35 17 95

28 58

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

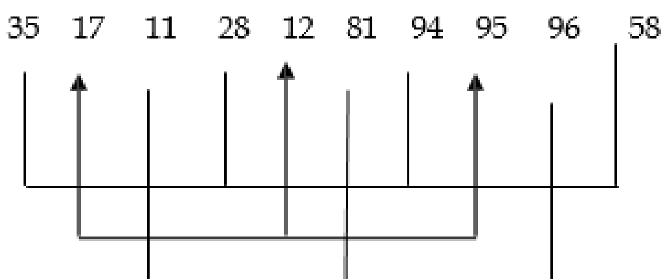
www.binils.com



After the first pass

35 17 11 28 12 81 94 95 96 58

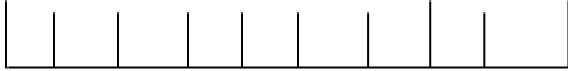
In second pass, K is reduced to 3



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 98

Program:

```
void shellsort(int arr[], int num)

{

        int i, j, k, tmp;

        for (i = num / 2; i > 0; i = i / 2)

        {

        for (j = i; j <num; j++)

        {

        for(k = j - i; k >= 0; k = k - i)

        {

        if (arr[k+i] >=

        arr[k]) break;

        else

        { tmp = arr[k]; arr[k] = arr[k+i]; arr[k+i]
```

www.binils.com

```
        = tmp;
        }
        }
        }
}
```

This algorithm is quite efficient for medium-sized data sets as its average and worst-case complexity of this algorithm depends on the gap sequence the best known is O(n), where n is the number of items. And the worst case space complexity is O(n).

Algorithm

Following is the algorithm for shell

sort. Step 1 − Initialize the

value of h

www.binils.com

Step 2 − Divide the list into smaller sub-list of equal

interval h Step 3 − Sort these sub-lists using insertion

sort

Step 3 − Repeat until complete list is sorted

RADIX SORT

The idea of Radix Sort is to do digit by digit sort starting from least significant digit to most significant digit. Radix sort uses counting sort as a subroutine to sort.

The Radix Sort Algorithm

1) Do following for each digit i where i varies from least significant digit to the most significant digit.

a) Sort input array using counting sort (or any stable sort) according to the i‟th digit.

Example:

Original, unsorted list:

170, 45, 75, 90, 802, 24, 2, 66

Sorting by least significant digit (1s place) gives: [*Notice that we keep 802 before 2, because 802 occurred before 2 in the original list, and similarly for pairs 170 & 90 and 45 & 75.]

170, 90, 802, 2, 24, 45, 75, 66

Sorting by next digit (10s place) gives: [*Notice that 802 again comes before 2 as 802 comes before 2 in the previous list.]

802, 2, 24, 45, 66, 170, 75, 90

Sorting by most significant digit (100s place) gives:

2, 24, 45, 66, 75, 90, 170, 802

www.binils.com

RADIX SORT ROUTINE

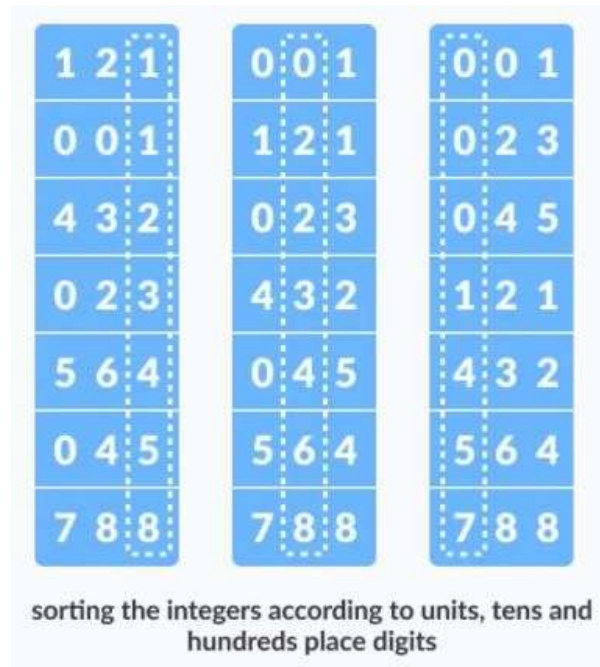
```
void countsort (int arr[],int n,int place)
    {
    int i,freq[range]={0};     //range for integers is 10 as digits range from 0-9 int
    output[n]; for(i=0;i<n;i++)
    freq[(arr[i]/place)%range]+
    +; for(i=1;i<range;i++)
    freq[i]+=freq[i-1];
    for(i=n-1;i>=0;i--)
    {
```

```
output[freq[(arr[i]/place)%range]-
1]=arr[i]; freq[(arr[i]/place)%range]--;
}
for(i=0;i<n;i++
)
arr[i]=output[i];
}
void radixsort(llarr[],intn,intmaxx)       //maxx is the maximum element in the array
{
int mul=1;
while(maxx
)
{
countsort(arr,n,mul);
mul*=10; maxx/=10;
}
}
```

www.binils.com

Example 2

Let the initial array be [121, 432, 564, 23, 1, 45, 788]. It is sorted according to radix sort as shown in the figure below.



sorting the integers according to units, tens and hundreds place digits

www.binils.com

Sorting

Sorting refers to arranging data in a particular format. Sorting algorithm specifies the way to arrange data in a particular order. Most common orders are in numerical or lexicographical order.

BUBBLE SORT

- Bubble sort is a simple sorting algorithm. This sorting algorithm is comparison-based algorithm in which each pair of adjacent elements is compared and the elements are swapped if they are not in order. This algorithm is not suitable for large data sets as its average and worst case complexity are of O(n2) where n is the number of items.

Example:

First Pass:

(5 1 4 2 8) –> (1 5 4 2 8), Here, algorithm compares the first two elements, and swaps since

5 > 1.

www.binils.com

(1 5 4 2 8) –> (1 4 5 2 8), Swap since 5 > 4 (1 4 5 2 8) –> (1 4 2 5 8), Swap since 5 > 2

(1 4 2 5 8) –> (1 4 2 5 8), Now, since these elements are already in order (8 > 5), algorithm does not swap them.

Second Pass:

(1 4 2 5 8) –> (1 4 2 5 8)

(1 4 2 5 8) –> (1 2 4 5 8), Swap since 4 > 2 (1 2 4 5 8) –> (1 2 4 5 8)

(1 2 4 5 8) –> (1 2 4 5 8)

Now, the array is already sorted, but our algorithm does not know if it is completed. The algorithm needs one whole pass without any swap to know it is sorted.

Third Pass:

(1 2 4 5 8) –> (1 2 4 5 8) (1 2 4 5 8) –> (1 2 4 5 8) (1 2 4 5 8) –> (1 2 4 5 8) (1 2 4 5 8) –> (1 2 4 5 8)

C Program to Sort data in ascending order using

bubble sort #include <stdio.h>

```
int main()

{

int data[100],i,n,step,temp;

printf("Enter the number of elements to be sorted:

"); scanf("%d",&n);

for(i=0;i<n;++i)

{

printf("%d. Enter element:

",i+1); scanf("%d",&data[i]);

}

for(step=0;step<n-

1;++step) for(i=0;i<n-

step-1;++i)

{

if(data[i]>data[i+1]) /* To sort in descending order, change > to < in this line. */

{

temp=data[i]; data[i]=data[i+1];

data[i+1]=te mp;

}

}
```

www.binils.com

```
printf("In ascending order:

"); for(i=0;i<n;++i)

printf("%d

",data[i]); return 0;

}
```

Example-2

Bubble sort starts with very first two elements, comparing them to check which one is greater.

In this case, value 33 is greater than 14, so it is already in sorted locations. Next, we compare 33 with 27.

We find that 27 is smaller than 33 and these two values must be swapped.

The new array should look like this –

Next we compare 33 and 35. We find that both are in already sorted positions.

Then we move to the next two values, 35 and 10.

We know then that 10 is smaller 35. Hence they are not sorted.

We swap these values. We find that we have reached the end of the array. After one iteration, the array should look like this –

To be precise, we are now showing how an array should look like after each iteration. After the second iteration, it should look like this –

Notice that after each iteration, at least one value moves at the end.

And when there's no swap required, bubble sorts learns that an array is completely sorted.

www.binils.com