UNIT V
SEARCHING, SORTING, HASHING

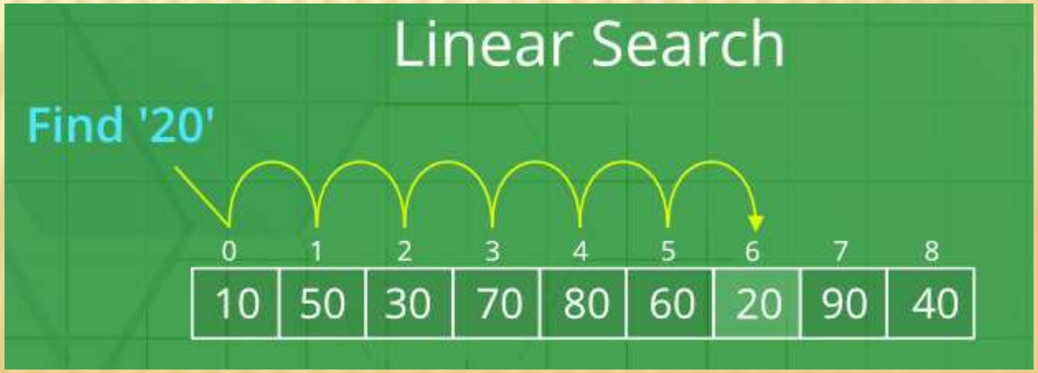
Searching- Linear Search - Binary Search. Sorting - Bubble sort - Selection sort - Insertion sort - Shell sort – Radix sort. Hashing- Hash Functions – Separate Chaining – Open Addressing – Rehashing – Extendible Hashing.

SEARCHING

- Searching 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 search is found or not. There are two types of searching as follows.
- ✓ Linear Search
- ✓ Binary Search

LINEAR SEARCH.

- In Linear search, searching an element of value in a given array by traversing the array from 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 returns -1.
- Linear search is applied on unsorted or unordered list, when there are fewer elements in a list.



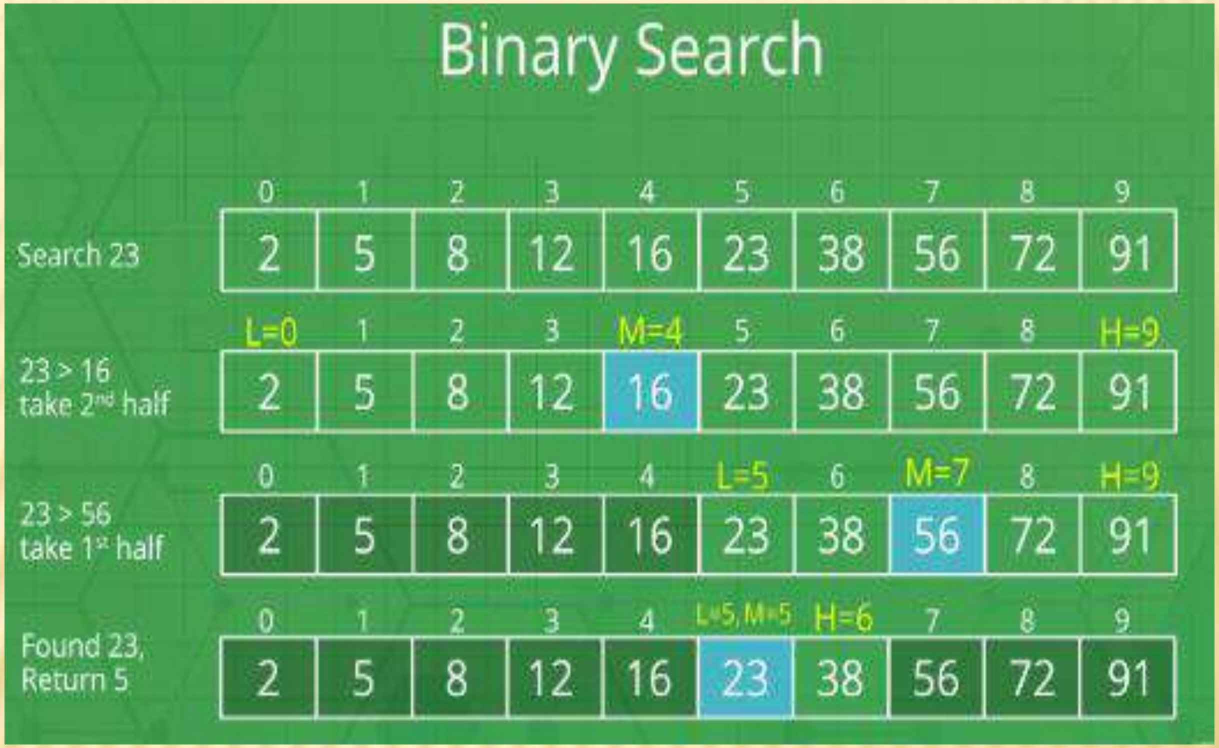
LINEAR SEARCH

```
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
```

BINARY SEARCH

- ✓ 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.

BINARY SEARCH



BINARY SEARCH

Step 1 – Start searching data from middle of the list.

Step 2 – If it is a match, return the index of the item, and exit.

Step 3 – If it is not a match, probe position.

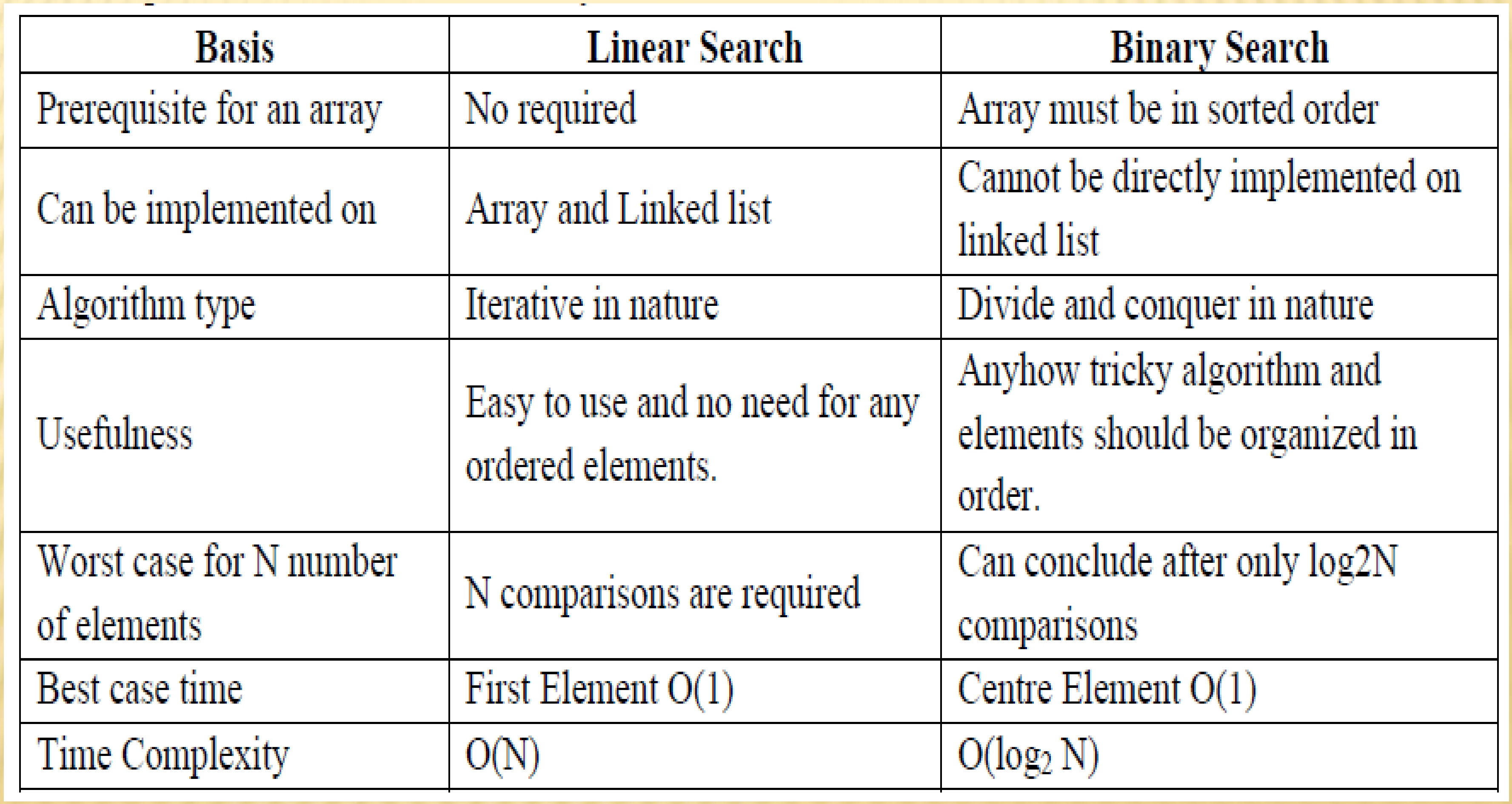
Step 4 – Divide the list using probing formula and find the new middle.

Step 5 – If data is greater than middle, search in higher sub-list.

Step 6 – If data is smaller than middle, search in lower sub-list.

Step 7 – Repeat until match.

LINEAR SEARCH VS BINARY SEARCH.



Basis	Linear Search	Binary Search
Prerequisite for an array	No required	Array must be in sorted order
Can be implemented on	Array and Linked list	Cannot be directly implemented on linked list
Algorithm type	Iterative in nature	Divide and conquer in nature
Usefulness	Easy to use and no need for any ordered elements.	Anyhow tricky algorithm and elements should be organized in order.
Worst case for N number of elements	N comparisons are required	Can conclude after only log2N comparisons
Best case time	First Element O(1)	Centre Element O(1)
Time Complexity	O(N)	$O(\log_2 N)$

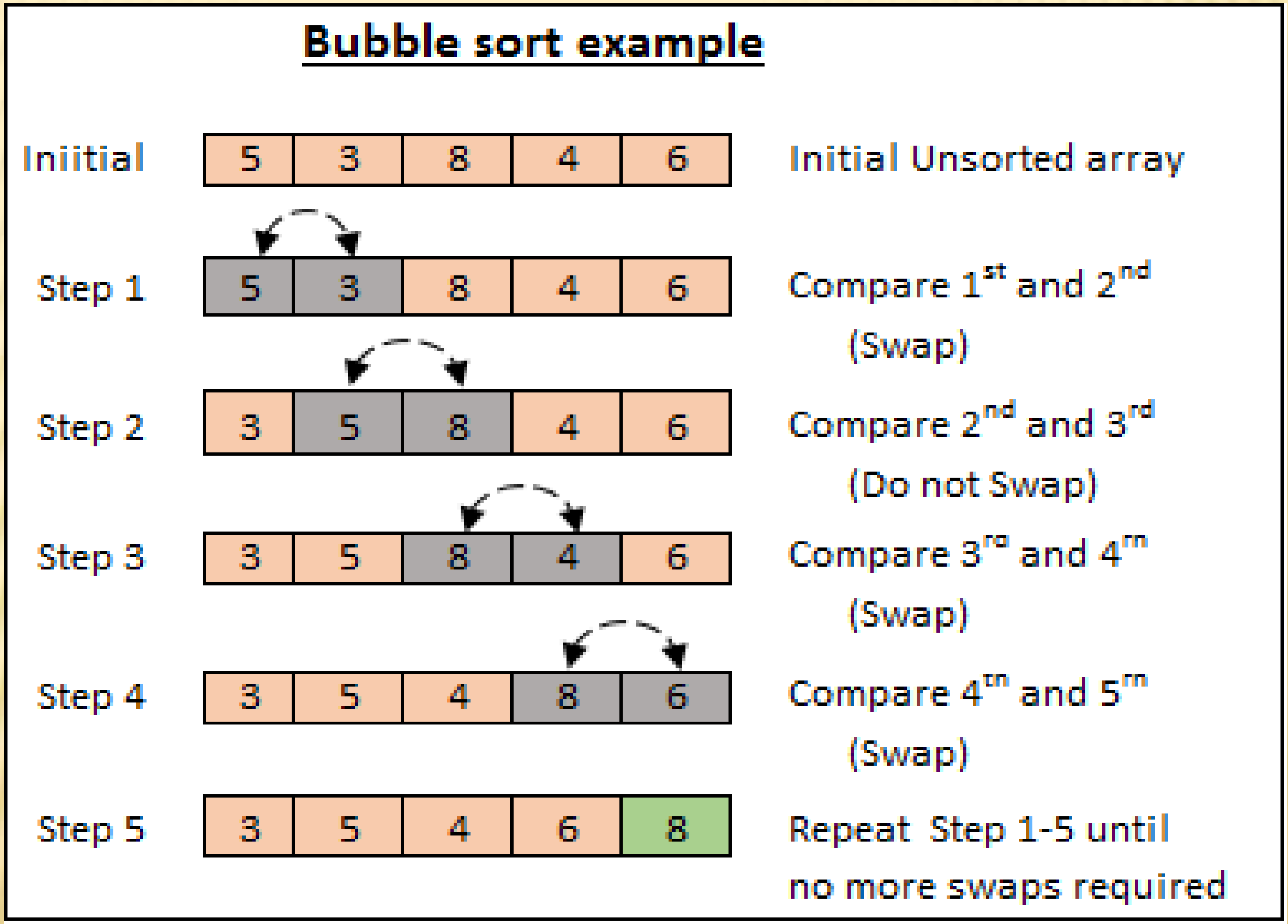
BUBBLE SORT

- Bubble sort is a sorting algorithm that works by repeatedly stepping through lists that need to be sorted, comparing each pair of adjacent items and swapping them if they are in the wrong order.
- ✓ This passing procedure is repeated until no swaps are required, indicating that the list is sorted.
- ✓ Bubble sort gets its name because smaller elements bubble toward the top of the list.
- ✓ Bubble sort is also referred to as Sinking Sort or Comparison Sort.

BUBBLE SORT

```
Algorithm Bubble_Sort(A[0,1,....n-1])
Input:An array of elements A[0,1,....n-1]
Output: The sorted array A[0,1,....n-1]
for i=1 to n-2 do
{
for j=0 to n-2-I do
{
if (A[j]>A[j+1]) then
{
temp=A[j]
A[j]= A[j+1]
A[j+1]=temp
}
}
```

BUBBLE SORT



Bubble sort example

Iniitial	5	3	8	4	6	Initial Unsorted array
Step 1	5	3	8	4	6	Compare 1st and 2nd (Swap)
Step 2	3	5	8	4	6	Compare 2nd and 3rd (Do not Swap)
Step 3	3	5	8	4	6	Compare 3rd and 4th (Swap)
Step 4	3	5	4	8	6	Compare 4th and 5th (Swap)
Step 5	3	5	4	6	8	Repeat Step 1-5 until no more swaps required

SELECTION SORT

- ✓ The selection sort algorithm sorts an array by repeatedly finding the minimum element (considering ascending order) from unsorted part and putting it at the beginning.
- ✓ The algorithm maintains two subarrays in a given array.
- ➢ The subarray which is already sorted.
- ➢ Remaining subarray which is unsorted.
- ✓ In every iteration of selection sort, the minimum element (considering ascending order) from the unsorted subarray is picked and moved to the sorted subarray.

SELECTION SORT

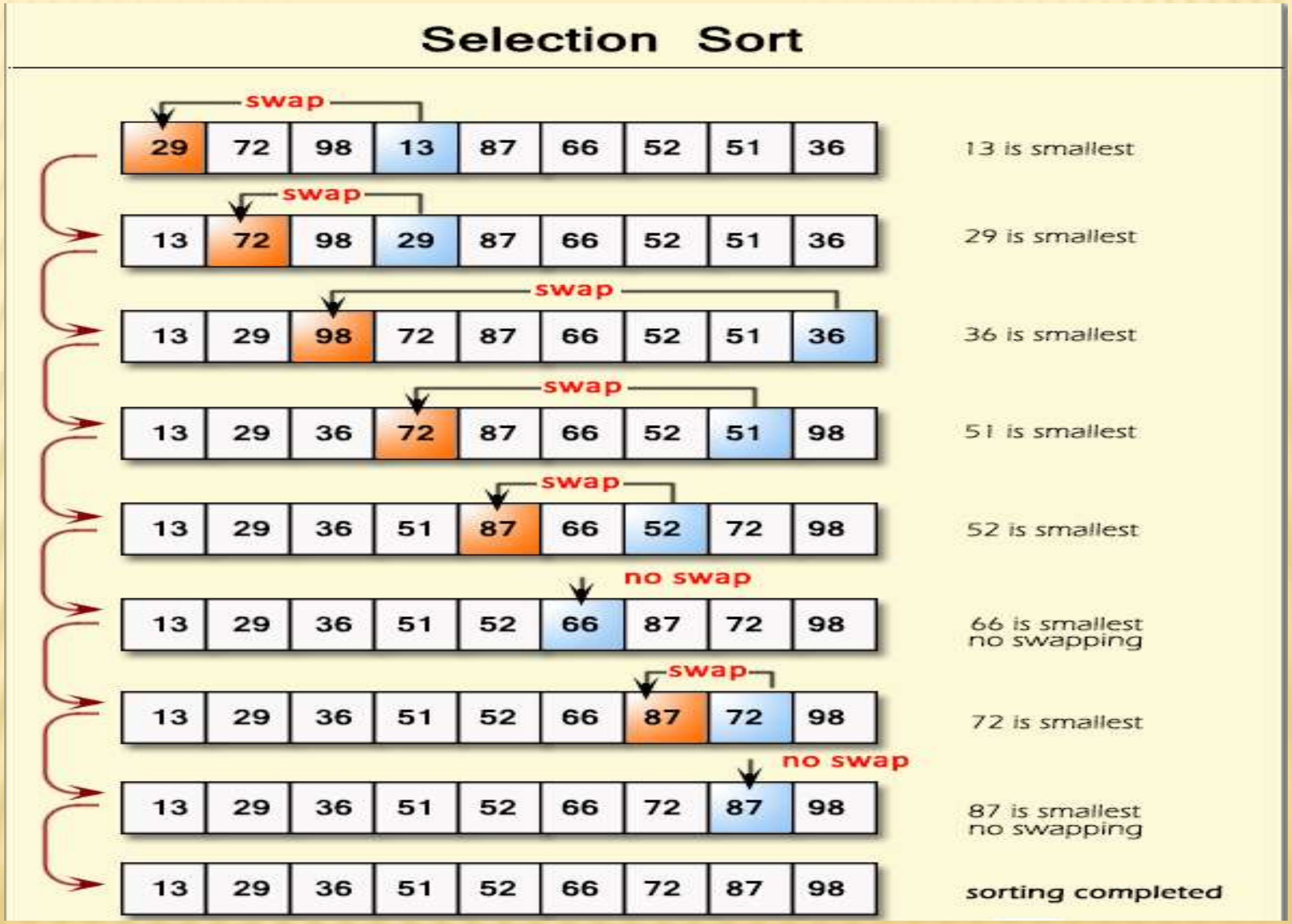
Algorithm Selection_Sort(A[0,1,....n-1])

Input:An array of elements A[0,1,....n-1] that is to be sorted.

Output: The sorted array A[0,1,....n-1]

```
for i = 1 to n – 1
{
min = i
for j = i+1 to n
{
if list[j] < list[min] then
{
min = j;
}
}
if indexMin != i then
{
swap (list[min], list[i]);
} }
```

SELECTION SORT



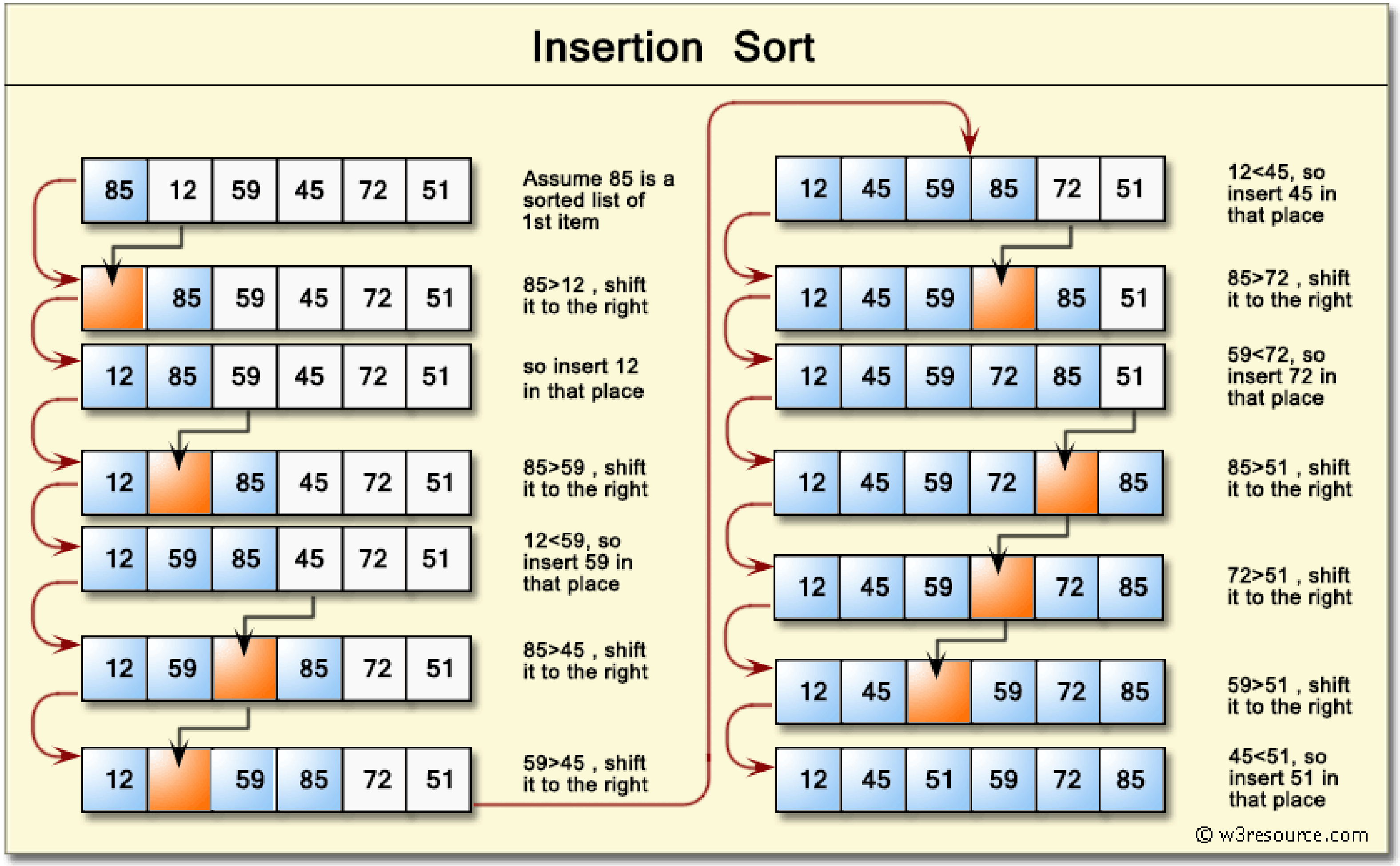
INSERTION SORT

- ✓ Insertion sort is a sorting algorithm in which the elements are transferred one at a time to the right position.
- ✓ In other words, an insertion sort helps in building the final sorted list, one item at a time, with the movement of higher-ranked elements.
- ✓ An insertion sort has the benefits of simplicity and low overhead.

INSERTION SORT

```
Algorithm Insertion_Sort(A[0,1,....n-1])
Input: An array of elements A[0,1,....n-1] that is to be sorted.
Output: The sorted array A[0,1,....n-1]
for i=1 to n-1 do
{
temp=A[i]
j=i-1
while(j>=0) AND (A[j]>temp) do
{
A[j+1]= A[j]
j=j-1
}
A[j+1])=temp
} }
```

INSERTION SORT



Insertion Sort
85 12 59 45 72 51
Assume 85 is a sorted list of 1st item
85 59 45 72 51
85>12 , shift it to the right
12 85 59 45 72 51
so insert 12 in that place
12 85 45 72 51
85>59 , shift it to the right
12 59 85 45 72 51
12<59, so insert 59 in that place
12 59 85 72 51
85>45 , shift it to the right
12 59 85 72 51
59>45 , shift it to the right
12 45 59 85 72 51
12<45, so insert 45 in that place
12 45 59 85 51
85>72 , shift it to the right
12 45 59 72 85 51
59<72, so insert 72 in that place
12 45 59 72 85
85>51 , shift it to the right
12 45 59 72 85
72>51 , shift it to the right
12 45 59 72 85
59>51 , shift it to the right
12 45 51 59 72 85
45<51, so insert 51 in that place
© w3resource.com

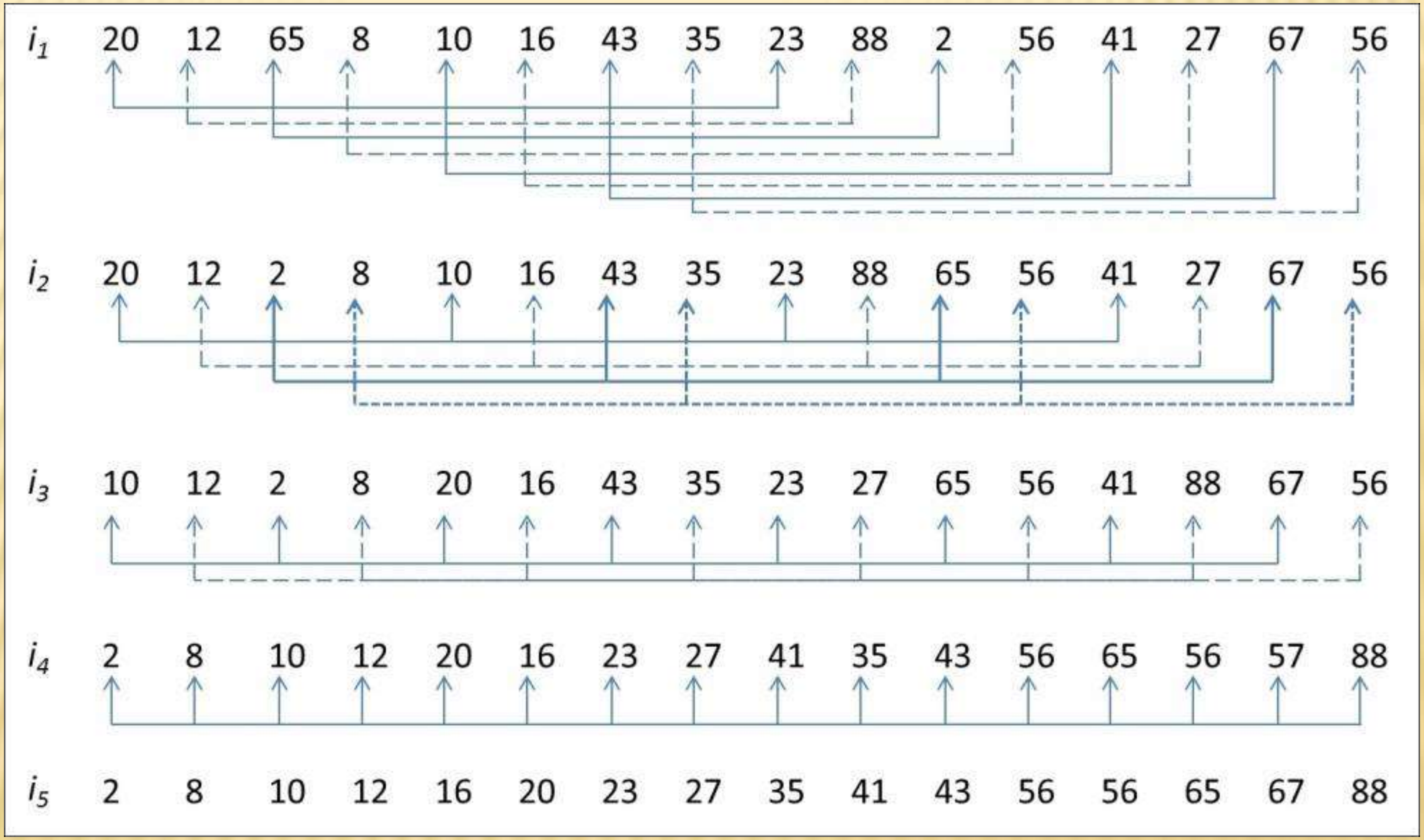
SHELL SORT

- ✓ Shell sort improves bubble sort and insertion sort by moving out of order elements more than one at at time.
- ✓ It works arranging the data sequence in a two dimensional array and then sorting the columns of the array using insertion sort.
- ✓ In shell sort 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 while 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 as “Diminishing Increment Sort”, because the value of K decreases continuously.

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;
}
}
```

SHELL SORT

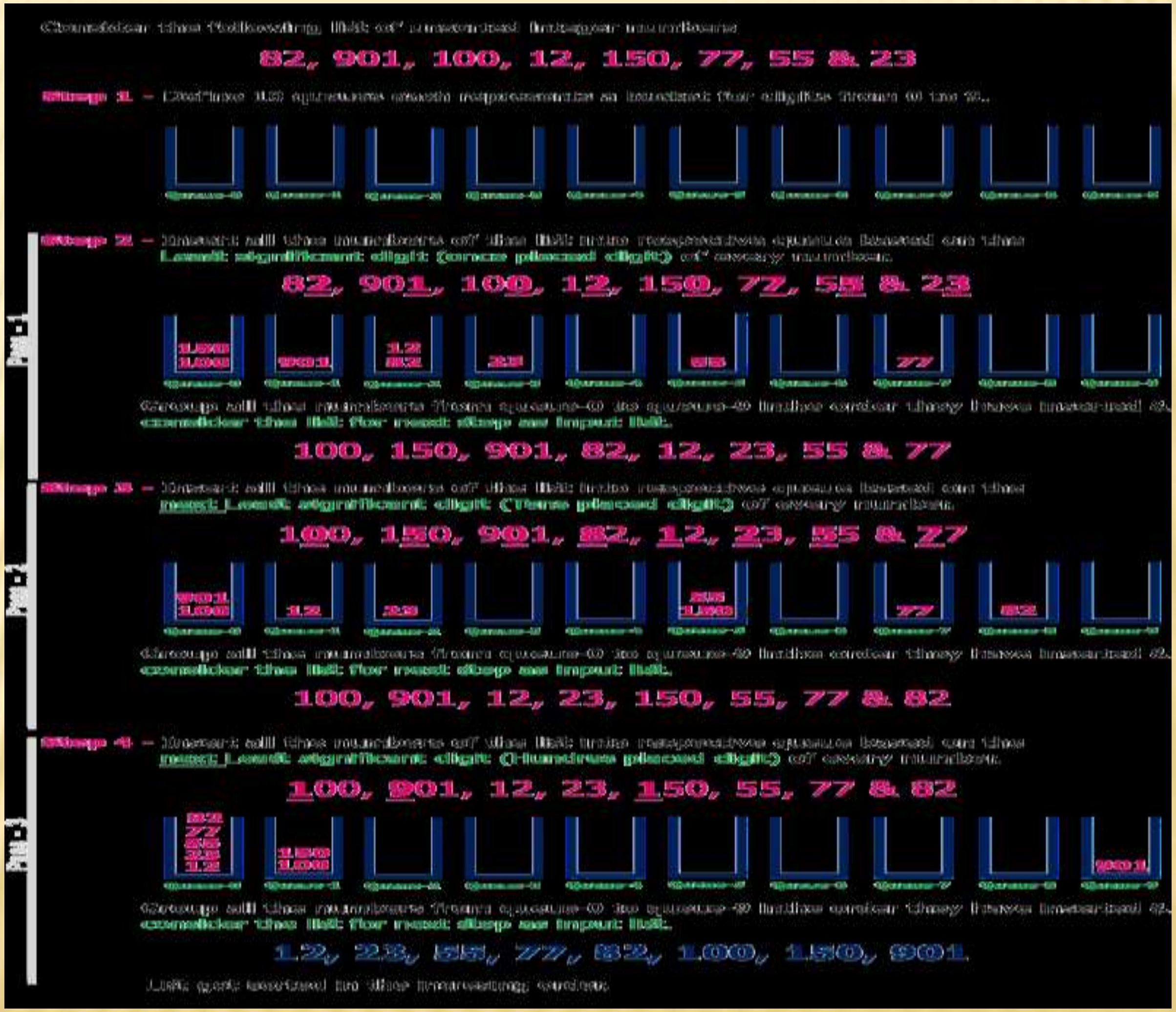


i_1 20 12 65 8 10 16 43 35 23 88 2 56 41 27 67 56
i_2 20 12 2 8 10 16 43 35 23 88 65 56 41 27 67 56
i_3 10 12 2 8 20 16 43 35 23 27 65 56 41 88 67 56
i_4 2 8 10 12 20 16 23 27 41 35 43 56 65 56 57 88
i_5 2 8 10 12 16 20 23 27 35 41 43 56 56 65 67 88

RADIX SORT

- ✓ Radix sort is one of the sorting algorithms used to sort a list of integer numbers in order.
- ✓ In radix sort algorithm, a list of integer numbers will be sorted based on the digits of individual numbers.
- ✓ Sorting is performed from least significant digit to the most significant digit.
- ✓ Radix sort algorithm requires the number of passes which are equal to the number of digits present in the largest number among the list of numbers.
- ✓ For example, if the largest number is a 3 digit number then that list is sorted with 3 passes.

RADIX SORT



Consider the following list of unsorted integer numbers

82, 901, 100, 12, 150, 77, 55 & 23

Step 1 – Define 10 queues each represents a bucket for digits from 0 to 9.

Step 2 – Insert all the numbers of the list into respective queue based on the Least significant digit (once placed digit) of every number.

82, 901, 100, 12, 150, 77, 55 & 23

Group all the numbers from queue-0 to queue-9 inthe order they have inserted & consider the list for next step as input list.

100, 150, 901, 82, 12, 23, 55 & 77

Step 3 – Insert all the numbers of the list into respective queue based on the next Least significant digit (Tens placed digit) of every number.

100, 150, 901, 82, 12, 23, 55 & 77

Group all the numbers from queue-0 to queue-9 inthe order they have inserted & consider the list for next step as input list.

100, 901, 12, 23, 150, 55, 77 & 82

Step 4 – Insert all the numbers of the list into respective queue based on the next Least significant digit (Hundreds placed digit) of every number.

100, 901, 12, 23, 150, 55, 77 & 82

Group all the numbers from queue-0 to queue-9 inthe order they have inserted & consider the list for next step as input list.

12, 23, 55, 77, 82, 100, 150, 901

List got sorted in the increasing order.

HASHING AND HASH TABLE.

- Hash Table: It is a data structure used for storing and retrieving data quickly. Every entry in the hash table is made using Hash function.
- Hash function is used to place data in the hash table. It is also used to retrieve data from hash table.
- Hash function is Hash (Key) =Keyvalue % TableSize

HASH FUNCTION

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

PROPERTIES OF HASH FUNCTION

- ✓ The hash function should be simple to compute.
- ✓ Number of collision should be less while placing the record in the hash table. Ideally no collision should occur. Such a function is called Perfect Hash Function.
- ✓ Hash function should produce such keys which will get distributed uniformly over an array.
- ✓ The hash function should depend on every bit of the key. Thus the hash function that simply extracts the portion of a key is not suitable.

COLLISION AND COLLISION RESOLUTION.

- A Collision occurs when two or more elements are hashed (mapped) to same value that is 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.

COLLISION RESOLUTION TECHNIQUES

- ➢ Separate Chaining
- ➢ Open Addressing (Closed Hashing)
  - ✓ Linear Probing
  - ✓ Quadratic Probing
  - ✓ Double Hashing
- ➢ Multiple Hashing

SEPARATE CHAINING

➢ Separate Chaining is an open addressing.

➢ A pointer field is added to each 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

Advantage:

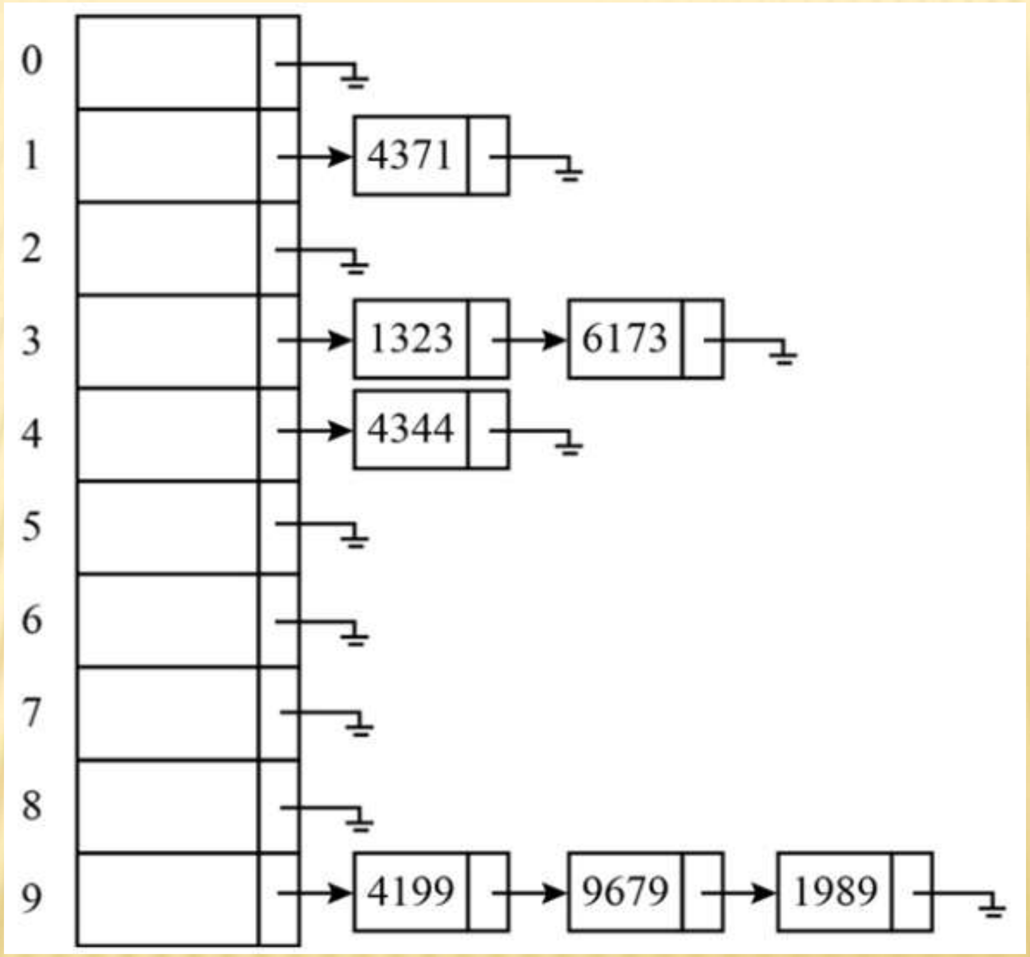
➢ More number of elements can be inserted.

➢ Disadvantages;

➢ It requires pointers, which occupies more memory space.

➢ It takes more effort to perform a search. are only extended upon the arrival of new keys.

SEPARATE CHAINING



OPEN ADDRESSING

Open addressing also called as Closed Hashing. It is an alternative to resolve the collision.

In this hashing if collision occurs, alternative cells are tried until an empty cell is found.

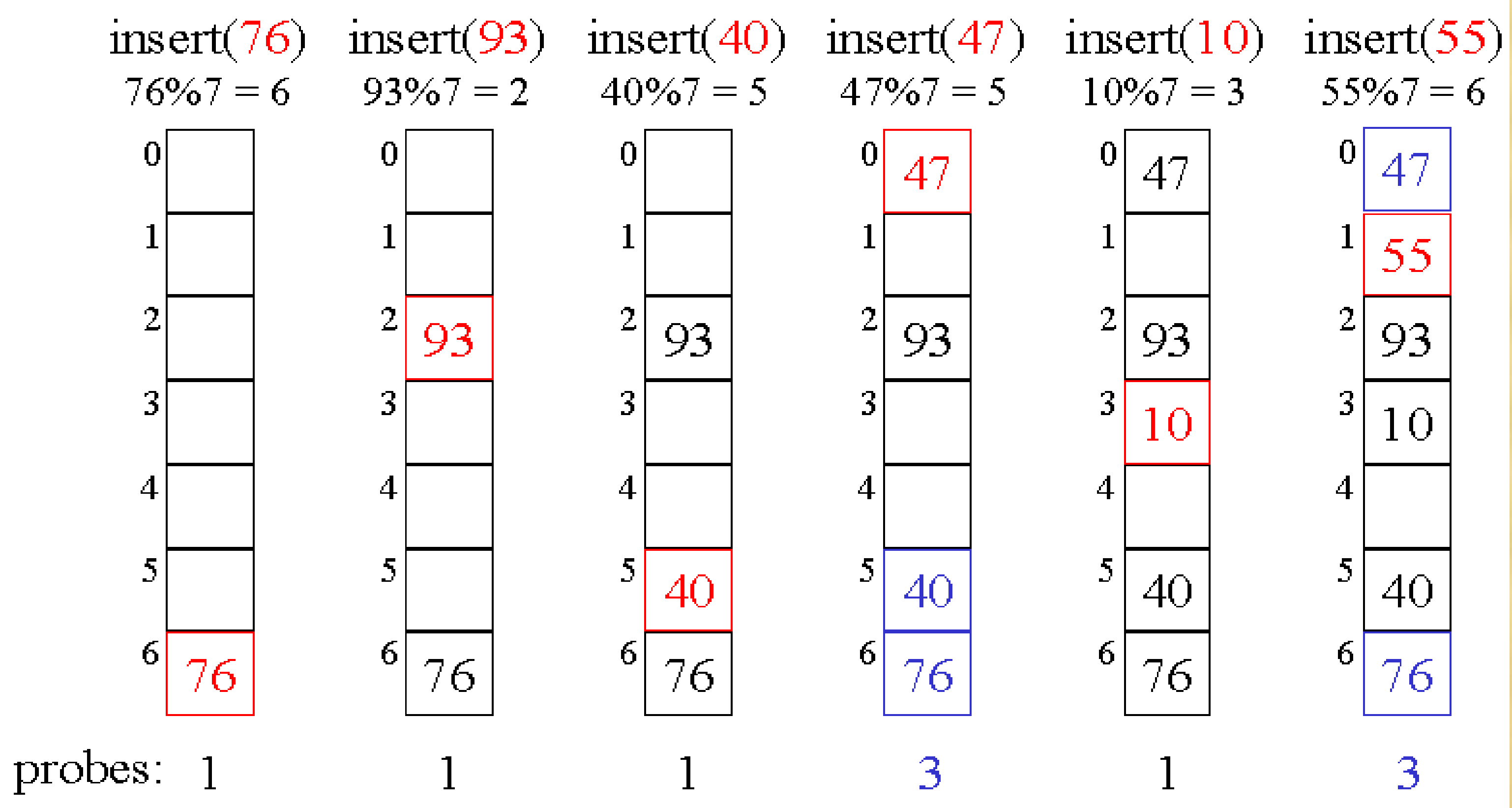
linear probing works:

➢ In linear probing, the position in which a key can be stored is found by sequentially searching all position starting from the position calculated by the hash function until an empty cell is found.

➢ If the end of the table is reached and no empty cell has been found, then the search is continued from the beginning of the table.

➢ It has a tendency to create clusters in the table.

Linear Probing Example



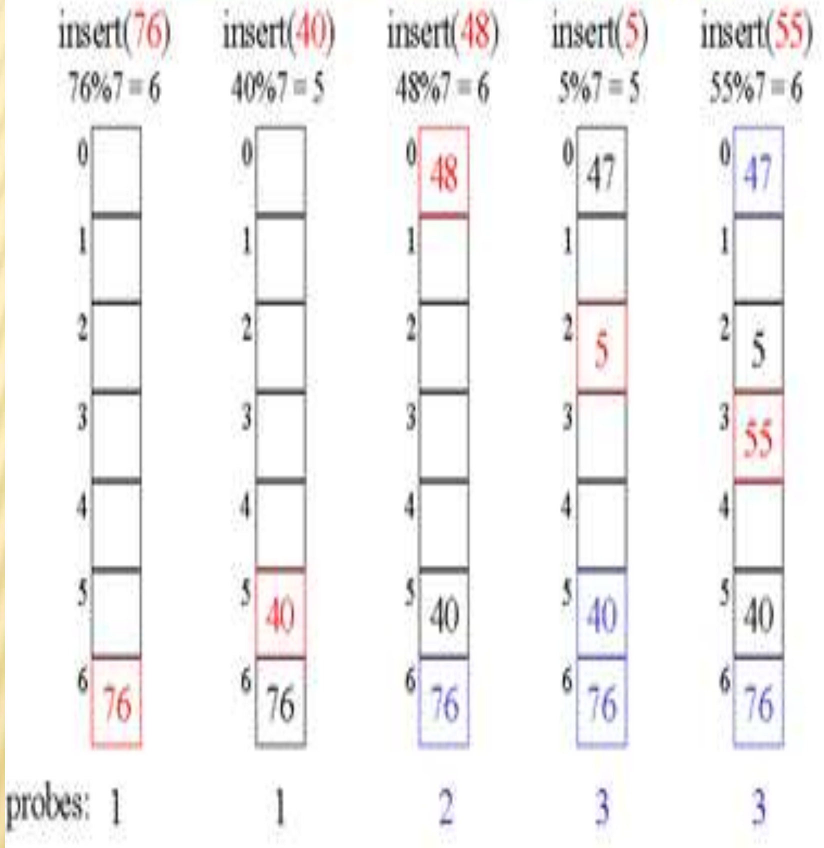
QUADRATIC PROBING

Quadratic probing is that if the element is inserted into a space that is filled, then 1^2=1 element away then 2^2=4 element away, then 3^2=9 elements away then 4^2=16 elements away and so on is checked

Probe Sequence is For example Tablesize=16

h(k) mod Tablesize	First element to Index2.
(h(k)+1) mod Tablesize	Second element to 3.[(2+1)%16]
(h(k)+4) mod Tablesize	Third element to 6.[(2+4)%16]
(h(k)+9) mod Tablesize	Fourth element to 11[(2+9)%16].

Quadratic Probing Example

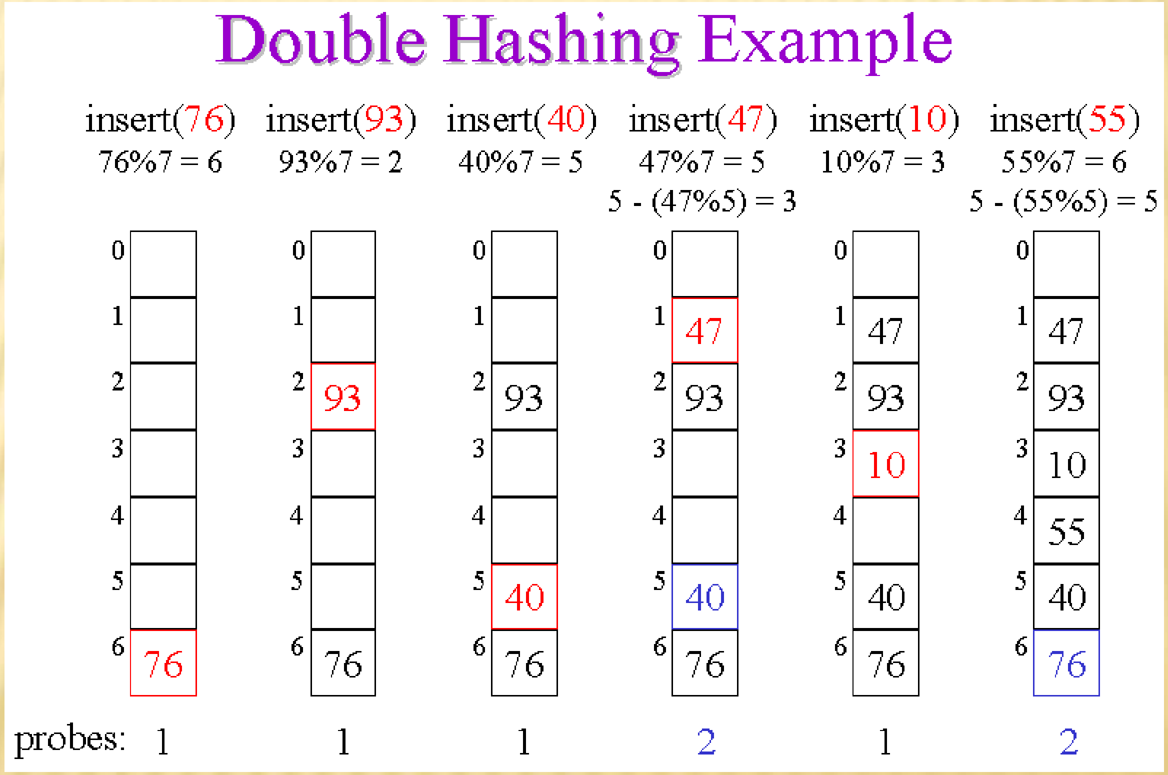


- findEntry using quadratic probing:

```
bool findEntry(const Key & k, Entry *& entry) {
  int probePoint = hash1(k), numProbes = 0;
  do {
    entry = &table[probePoint];
    numProbes++;
    probePoint = (probePoint + 2*numProbes - 1) % size;
  } while (!entry->isEmpty() && entry->key != key);
  return !entry->isEmpty();
}
```

DOUBLE HASHING

One choice to choose a prime R< size and hash2(x)=R-(X mod R)



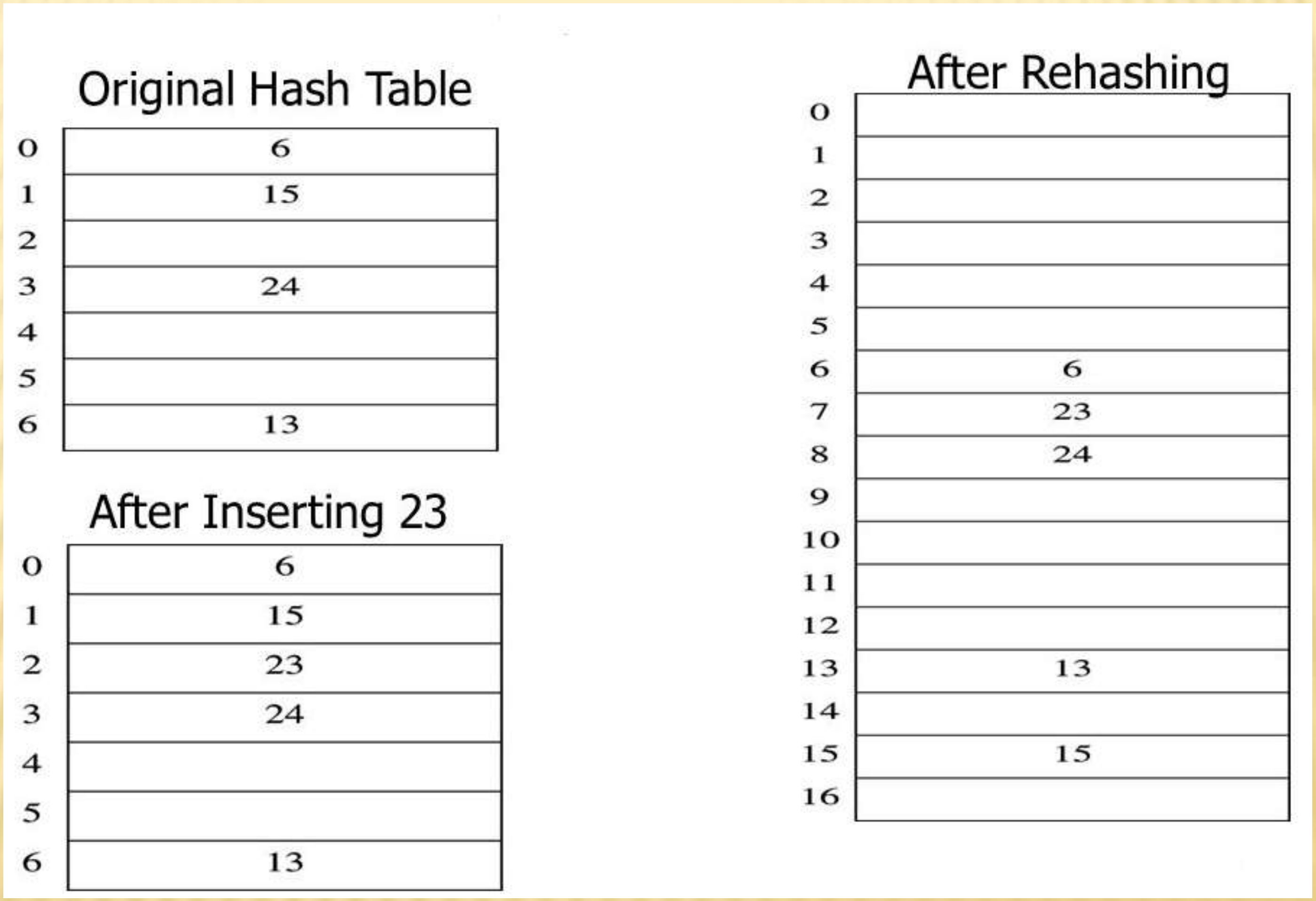
REHASHING

➢ If the table gets too full, then the rehashing method builds a new table that is about twice as big and scans down the entire original hash table, computing the new hash value for each element and inserting it in the new table.

➢ For example Old Tablesize is 7 and the size of the new table is 17, as this is the first prime number that as twice as large as the old table size.

Rehashing can be implemented in several ways such as

- Rehash as soon as the table is full.
- Rehash only when an insertion fails.
- Rehash when the table reaches a certain load factor.



Original Hash Table

0	6
1	15
2	
3	24
4	
5	
6	13

After Inserting 23

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

After Rehashing

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

EXTENDIBLE HASHING

A hash table in which the hash function is the last few bits of the key and the table refer to buckets.

Table entries with the same final bits may use the same bucket. If a bucket overflows, it splits, and if only one entry referred to it, the table doubles in size.

If a bucket is emptied by deletion, entries using it are changed to refer to an adjoining bucket, and the table may be halved.