Searching and Sorting

sushmakadge@somaiya.edu

Searching

- Searching is a very common operation in most computer applications.
- If we browse the Internet there is virtually no page where we will not find a search button!
- The Google search -search facility helps Internet users.
- Windows operating systems also have search facility to find files and folders.

Searching

- Searching refers to finding the position of a value in a collection of values
- Two popular methods for searching the array elements:

  Linear search

  Binary search

Linear Search

- Linear search, also called as *sequential search*
- Very simple method
- Linear search is mostly used to search an unordered list of elements (array in which data elements are not sorted).
- For example, if an array A[] is declared and initialized as,
- int A[] = {10, 8, 2, 7, 3, 4, 9, 1, 6, 5};
- Value to be searched is VAL = 7,
- Returns the position of its occurrence i.e. POS = 3

Algorithm for linear search

```
LINEAR_SEARCH(A, N, VAL)

Step 1: [INITIALIZE] SET POS = -1
Step 2: [INITIALIZE] SET I = 1
Step 3:     Repeat Step 4 while I<=N
Step 4:             IF A[I] = VAL
                        SET POS = I
                        PRINT POS
                        Go to Step 6
                    [END OF IF]
                     SET I = I + 1
            [END OF LOOP]
Step 5: IF POS = -1
        PRINT "VALUE IS NOT PRESENT
        IN THE ARRAY"
        [END OF IF]
Step 6: EXIT
```

Linear Search

- In Steps 1 and 2 of the algorithm, initialize the value of POS and I.
- In Step 3, a while loop is executed that would be executed till I is less than N (total number of elements in the array).
- In Step 4, a check is made to see if a match is found between the current array element and VAL.
- If a match is found, then the position of the array element is printed, else the value of I is incremented to match the next element with VAL. However, if all the array elements have been compared with VAL and no match is found, then it means that VAL present in the array.

Binary Search

- Binary search is a searching algorithm that works efficiently with a sorted list.
- Binary search can be better understood by an analogy of a telephone directory.
- Another analogy. How do we find words in a dictionary?
- The same mechanism is applied in the binary search.
- Divide and conquer

Binary Search

- A[] = {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10}; value to be searched is VAL = 9. The algorithm will proceed in the following manner.
- BEG = 0, END = 10, MID = (0 + 10)/2 = 5
- Now, VAL = 9 and A[MID] = A[5] = 5,   A[5] < VAL
- Search for the value in the 2nd half of the array. Change the values of BEG and MID.
- Now, BEG = MID + 1 = 6, END = 10, MID = (6 + 10)/2 =16/2 = 8
- VAL = 9 and A[MID] = A[8] = 8
- A[8] <VAL, therefore, we now search for the value in the second half of the segment.
- So, again we change the values of BEG and MID.
- Now, BEG = MID + 1 = 9, END = 10, MID = (9 + 10)/2 = 9
- Now, VAL = 9 and A[MID] = 9.

Binary Search

- MID is calculated as (BEG + END)/2.
- Initially, BEG = lower_bound and END = upper_bound.
- The algorithm will terminate when A[MID] = VAL.
- When the algorithm ends, we will set POS = MID.
- POS is the position at which the value is present in the array.
- However, if VAL is not equal to A[MID], then the values of BEG, END, and MID will be changed depending on whether VAL is smaller or greater than A[MID].

Binary Search

- If VAL < A[MID], then VAL will be present in the left segment of the array. So, the value of END will be changed as END = MID – 1.
- If VAL > A[MID], then VAL will be present in the right segment of the array. So, the value of BEG will be changed as BEG = MID + 1.

Binary Search

```
BINARY_SEARCH(A, lower_bound, upper_bound, VAL)
Step 1: [INITIALIZE] SET BEG = lower_bound
        END = upper_bound, POS = - 1
Step 2: Repeat Steps 3 and 4 while BEG <= END
Step 3:          SET MID = (BEG + END)/2
Step 4:          IF A[MID] = VAL
                       SET POS = MID
                       PRINT POS
                       Go to Step 6
                 ELSE IF A[MID] > VAL
                       SET END = MID - 1
                 ELSE
                       SET BEG = MID + 1
                 [END OF IF]
        [END OF LOOP]
Step 5: IF POS = -1
              PRINT "VALUE IS NOT PRESENT IN THE ARRAY"
        [END OF IF]
Step 6: EXIT
```

Sorting

- Sorting means arranging the elements of an array so that they are placed in some relevant order which may be either ascending or descending.
- If A is an array, then the elements of A are arranged in a sorted order (ascending order) in such a way that A[0] < A[1] < A[2] < ...... < A[N].
- For example, if we have an array that is declared and initialized as
- int A[] = {21, 34, 11, 9, 1, 0, 22};
- Then the sorted array (ascending order) can be given as:
- A[] = {0, 1, 9, 11, 21, 22, 34};

Sorting

- A sorting algorithm is defined as an algorithm that puts the elements of a list in a certain order, which can be either numerical order, lexicographical order, or any user-defined order.
- Efficient sorting algorithms are widely used to optimize the use of other algorithms like search and merge algorithms which require sorted lists to work correctly.

Sorting on Multiple Keys

- In real-world applications, to sort arrays of records → multiple keys.
- Ex, big organization
- Telephone directories
- Library
- Customers' address

Sorting

- Data records can be sorted based on a property. Such a component or property is called a **sort key**.

- A sort key can be defined using two or more sort keys.

  -The first key is called the **primary sort key**,

  -The second is known as the **secondary sort key**, etc.

Sorting

Name	Department	Salary	Phone Number
Janak	Telecommunications	1000000	9812345678
Raj	Computer Science	890000	9910023456
Aditya	Electronics	900000	7838987654
Huma	Telecommunications	1100000	9654123456
Divya	Computer Science	750000	9350123455

Name	Department	Salary	Phone Number
Divya	Computer Science	750000	9350123455
Raj	Computer Science	890000	9910023456
Aditya	Electronics	900000	7838987654
Huma	Telecommunications	1100000	9654123456
Janak	Telecommunications	1000000	9812345678

Practical Considerations for Internal Sorting

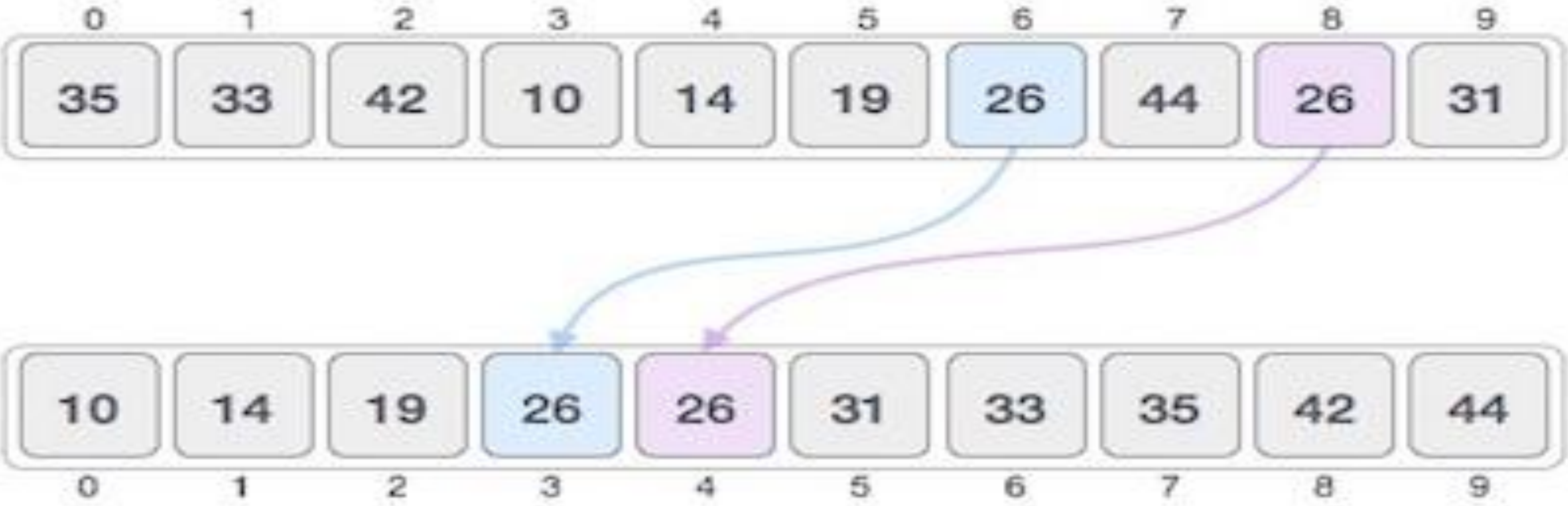
- Records can be sorted either in ascending or descending order based on a field often called as the sort key.
- The list of records can be either stored in a contiguous and randomly accessible data structure (array) or may be stored in a dispersed and only sequentially accessible data structure like a linked list.
- The logic to sort the records will be same and only the implementation details will differ.

Practical Considerations for Internal Sorting

- When analysing the performance of different sorting algorithms, the practical considerations would be the following:
- Number of sort key comparisons that will be performed
- Number of times the records in the list will be moved
- Best case performance
- Worst case performance
- Average case performance
- Stability of the sorting algorithm where stability means that equivalent elements or records retain their relative positions even after sorting is done

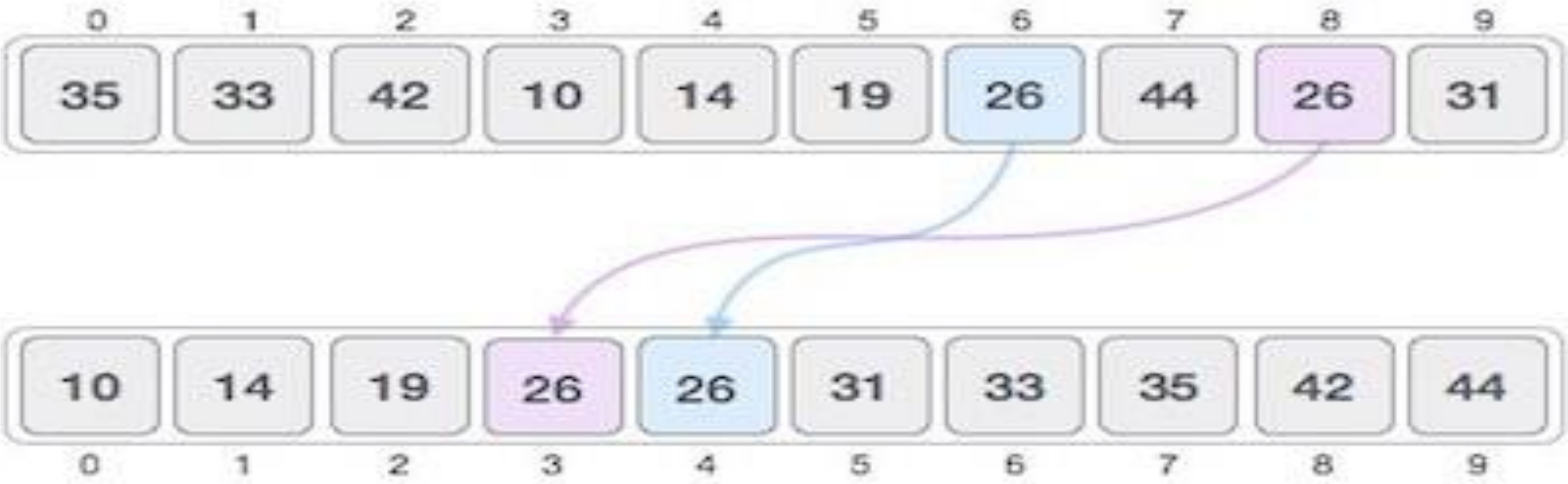
Stable and Not Stable Sorting

- If a sorting algorithm, after sorting the contents, does not change the sequence of similar content in which they appear, it is called **stable sorting**.



Stable and Not Stable Sorting

- If a sorting algorithm, after sorting the contents, changes the sequence of similar content in which they appear, it is called **unstable sorting**.



Stable and Not Stable Sorting

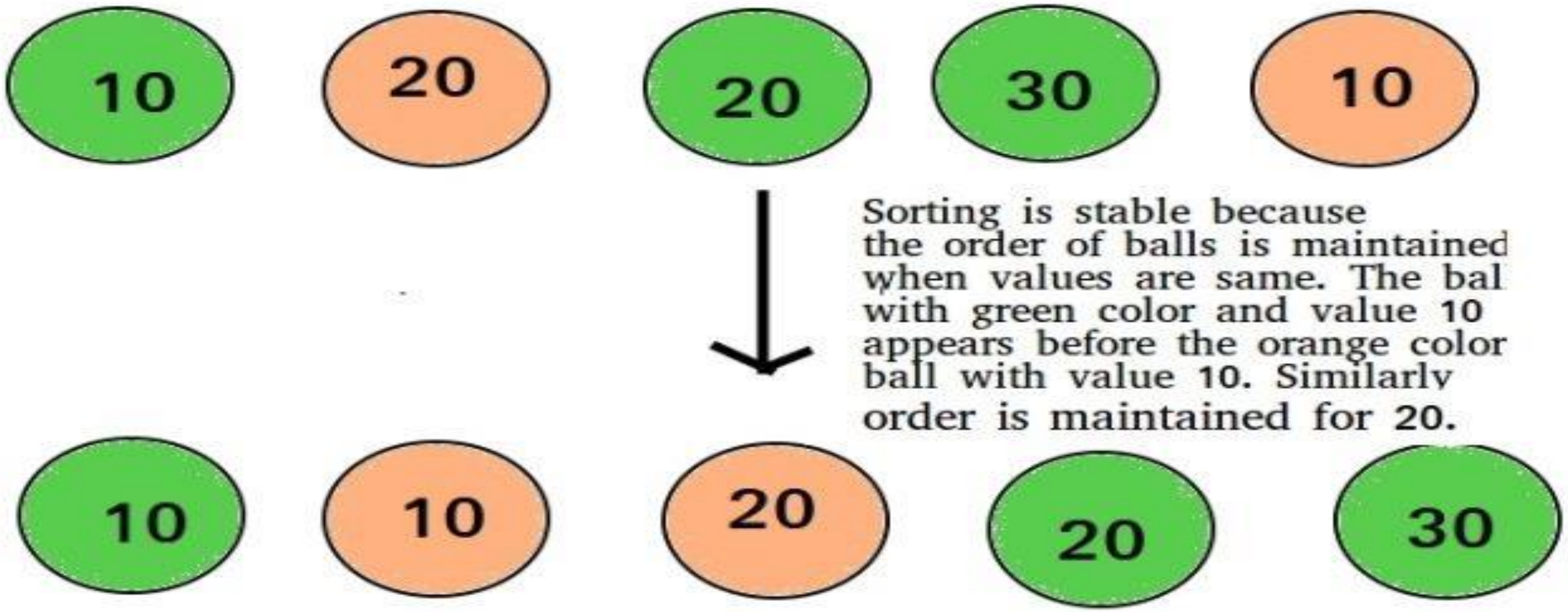
- Stability of an algorithm matters when we wish to maintain the sequence of original elements, like in a tuple for example.

Stable and Not Stable Sorting

- Stability is mainly important when we have key value pairs with duplicate keys possible (like people names as keys and their details as values). And we wish to sort these objects by keys.

- **What is it?**
  A sorting algorithm is said to be **stable if two objects with equal keys appear in the same order in sorted output as they appear in the input array to be sorted.**

Stable and Not Stable Sorting



BUBBLE SORT

- Bubble sort is a very simple method that sorts the array elements by repeatedly moving the largest element to the highest index position of the array segment (in case of arranging elements in ascending order).
- In *bubble sorting*, consecutive adjacent pairs of elements in the array are compared with each other. If the element at the lower index is greater than the element at the higher index, the two elements are interchanged so that the element is placed before the bigger one. This process will continue till the list of unsorted elements exhausts.
- This procedure of sorting is called bubble sorting because elements 'bubble' to the top of the list. Note that at the end of the first pass, the largest element in the list will be placed at its proper position (i.e., at the end of the list).

Technique BUBBLE SORT

- The basic methodology of the working of bubble sort is given as follows:
- (a) In Pass 1, A[0] and A[1] are compared, then A[1] is compared with A[2], A[2] is compared with A[3], and so on. Finally, A[N–2] is compared with A[N–1]. Pass 1 involves n–1 comparisons and places the biggest element at the highest index of the array.
- (b) In Pass 2, A[0] and A[1] are compared, then A[1] is compared with A[2], A[2] is compared with A[3], and so on. Finally, A[N–3] is compared with A[N–2]. Pass 2 involves n–2 comparisons and places the second biggest element at the second highest index of the array.
- (c) In Pass 3, A[0] and A[1] are compared, then A[1] is compared with A[2], A[2] is compared with A[3], and so on. Finally, A[N–4] is compared with A[N–3]. Pass 3 involves n–3 comparisons and places the third biggest element at the third highest index of the array.
- (d) In Pass n–1, A[0] and A[1] are compared so that A[0]<A[1]. After this step, all the elements of the array are arranged in ascending order.

Technique BUBBLE SORT

- To discuss bubble sort in detail, let us consider an array A[] that has the following elements: A[] = {30, 52, 29, 87, 63, 27, 19, 54}
- **Pass 1:**
- 30, **29, 52**, 87, 63, 27, 19, 54
- 30, 29, 52, **63, 87**, 27, 19, 54
- 30, 29, 52, 63, **27, 87**, 19, 54
- 30, 29, 52, 63, 27, **19**, **87**, 54
- 30, 29, 52, 63, 27, 19, **54, 87**

Compare 30 and 52; No swap
Compare 52 and 29 ; Swap
Compare 52 and 87; No swap
Compare 87 and 63 swap
Compare 87 and 27 swap
Compare 87 and 19 swap
Compare 87 and 54 swap

- Observe that after the end of the first pass, the largest element is placed at the highest index of the array. All the other elements are still unsorted.

Technique BUBBLE SORT

- To discuss bubble sort in detail, let us consider an array A[] that has the following elements: A[] = {30, 52, 29, 87, 63, 27, 19, 54}
- Pass 1: 30, 29, 52, 63, 27, 19, **54, 87**
- **Pass 2:**
- **29, 30**, 52, 63, 27, 19, 54, 87
- 29, 30, 52, **27, 63**, 19, 54, 87
- 29, 30, 52, 27, **19, 63**, 54, 87
- 29, 30, 52, 27, 19, **54, 63**, 87

Compare 30 and 29.
Compare 30 and 52
Compare 52 and 63
Compare 63 and 27.
Compare 63 and 19
Compare 63 and 54

- Observe that after the end of the second pass, the second largest element is placed at the second highest index of the array. All the other elements are still unsorted.

Technique BUBBLE SORT

- To discuss bubble sort in detail, let us consider an array A[] that has the following elements: A[] = {30, 52, 29, 87, 63, 27, 19, 54}
- Pass 1: 30, 29, 52, 63, 27, 19, **54, 87**
- Pass 2: 29, 30, 52, 27, 19, **54, 63**, 87
- Pass 3: 29, 30, 27**, 19, 52**, 54, 63, 87
- Pass 4: 29, 27, **19, 30**, 52, 54, 63, 87
- Pass 5: 27, **19, 29**, 30, 52, 54, 63, 87
- Pass 6: **19, 27,** 29, 30, 52, 54, 63, 87

Algorithm for bubble sort

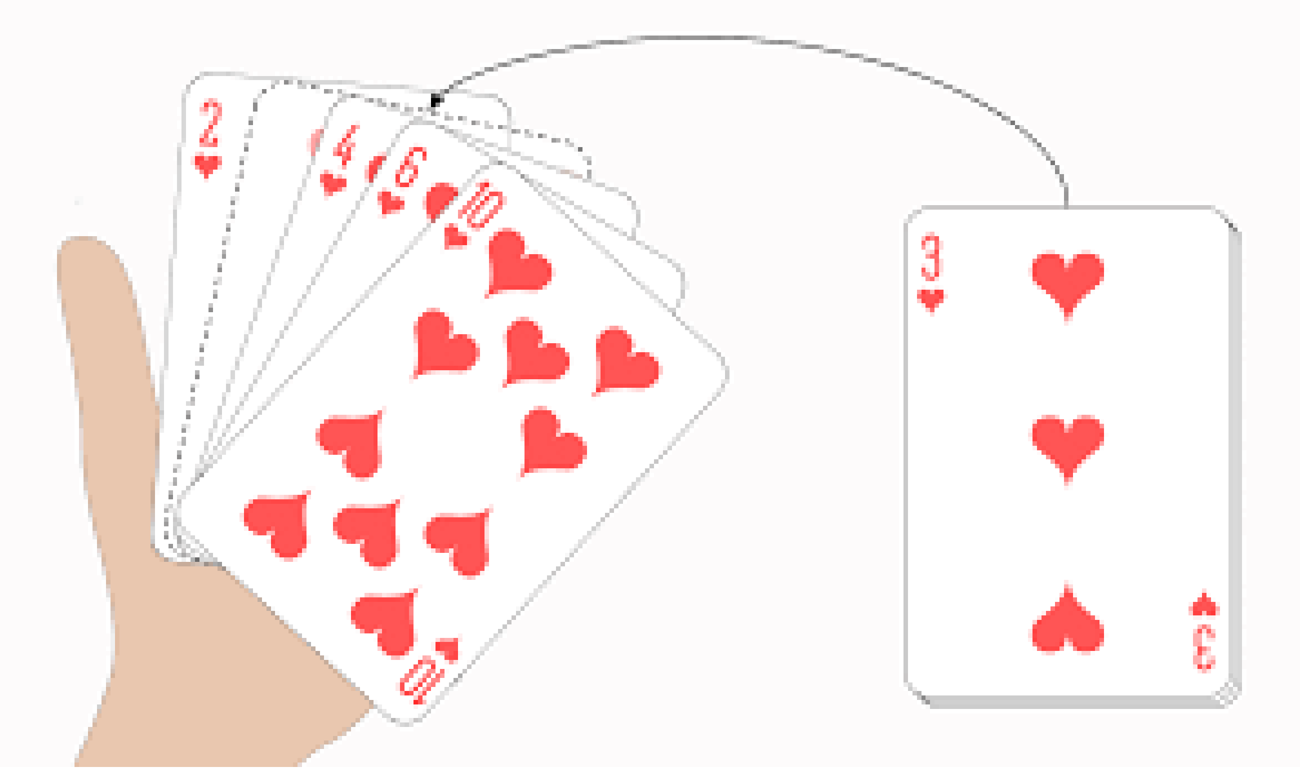
```
BUBBLE_SORT(A, N)

Step 1: Repeat Step 2 For 1 = 0 to N-1
Step 2:     Repeat For J = 0 to N - I
Step 3:                     IF A[J] > A[J + 1]
                            SWAP A[J] and A[J+1]
            [END OF INNER LOOP]
        [END OF OUTER LOOP]
Step 4: EXIT
```

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

INSERTION SORT

- Insertion sort is a very simple sorting algorithm in which the sorted array (or list) is built one element at a time.
- use it for ordering a deck of cards while playing bridge.



INSERTION SORT

- The main idea behind insertion sort is that it inserts each item into its proper place in the final list.
- To save memory, most implementations of the insertion sort algorithm work by moving the current data element past the already sorted values and repeatedly interchanging it with the preceding value until it is in its correct place.

Technique INSERTION SORT

- Insertion sort works as follows:
- The array of values to be sorted is divided into two sets. One that stores sorted values and another that contains unsorted values.
- The sorting algorithm will proceed until there are elements in the unsorted set.
- Suppose there are n elements in the array. Initially, the element with index 0 (assuming LB = 0) is in the sorted set. Rest of the elements are in the unsorted set.
- The first element of the unsorted partition has array index 1 (if LB = 0).
- During each iteration of the algorithm, the first element in the unsorted set is picked up and inserted into the correct position in the sorted set.

INSERTION SORT

- Consider an array of integers given below. We will sort the values in the array using insertion sort.

39	9	45	63	18	81	108	54	72	36

39	9	45	63	18	81	108	54	72	36

A[0] is the only element in sorted list

9	39	45	63	18	81	108	54	72	36

(Pass 1)

9	39	45	63	18	81	108	54	72	36

(Pass 2)

9	39	45	63	18	81	108	54	72	36

(Pass 3)

9	18	39	45	63	81	108	54	72	36

(Pass 4)

9	18	39	45	63	81	108	54	72	36

(Pass 5)

9	18	39	45	63	81	108	54	72	36

(Pass 6)

9	18	39	45	54	63	81	108	72	36

(Pass 7)

9	18	39	45	54	63	72	81	108	36

(Pass 8)

9	18	36	39	45	54	63	72	81	108

(Pass 9)

Algorithm for insertion sort

- Step 1 – If it is the first element, it is already sorted. return 1;
- Step 2 – Iterate from arr[1] to arr[n-1] over the array.
- Pick the current element and store it separately in a **key.**
- Step 3 – Compare the key with its predecessors i.e. all elements in the sorted sub-list
- Step 4 – Shift all the elements in the sorted sub-list that are greater than the key towards the right by one position to make space for the key
- Step 5 – Insert the key
- Step 6 – Repeat until list is sorted

Counting sort

- Sorting is based on keys between a specific range.
- It works by counting the number of objects having distinct key values
- Followed by computation of position of each object in the output sequence.

Counting sort

- Initialize count array of the size of input range
- Update the count array to store the count of each unique key.
- Further update the count array with cumulative additions of previous counts
- Shift the count array to right by one position; no circular shift
- Initialize sort array of the size of input sequence.

Counting sort example

- i/p : 2 3 1 2 4 5 2 1 5 4
- N= 10, range: 1:5

Initialize count array of the size of input range

count array	0	1	2	3	4	5
	0	0	0	0	0	0

Update the count array to store the count of each unique key.

count array	0	1	2	3	4	5
	0	2	3	1	2	2

Further update the count array with cumulative additions of previous counts

count array	0	1	2	3	4	5
	0	2	5	6	8	10

Shift the count array to right by one position; no circular shift

count array	0	1	2	3	4	5
	0	0	2	5	6	8

Initialize sort array of the size of input sequence

Sort Array	0	1	2	3	4	5	6	7	8	9
i/p	2	3	1	2	4	5	2	1	5	4
count array	0	1	2	3	4	5				
	0	2	3	1	2	2				
count array	0	1	2	3	4	5				
	0	0	2	5	6	8				
Output Sorted Array	0	1	2	3	4	5	6	7	8	9
	1	1	2	2	2	3	4	4	5	5

Counting Sort

- Counting sort is a sorting algorithm that sorts the elements of an array by counting the number of occurrences of each unique element in the array.

Counting Sort

Database needed-

- Original Array/Input Array
- Count Array/Auxilary Array
  - Take a count array to store the count of each unique object.
  - Store the count of each element at their respective index in count array

- For example: if the count of element 3 is 2 then, 2 is stored in the 3rd position of *count* array.
- If element "5" is not present in the array, then 0 is stored in 5th position.

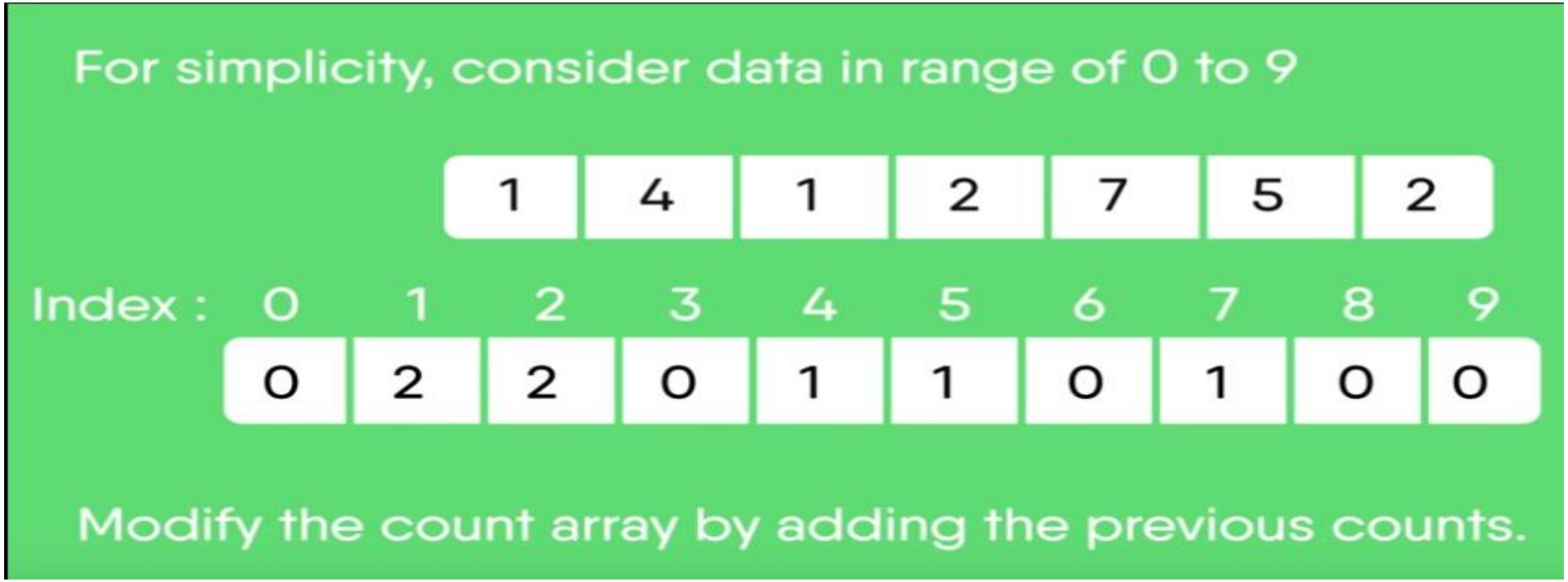
Counting Sort

- Store cumulative sum of the elements of the count array. It helps in placing the elements into the correct index of the sorted array.
- The modified count array indicates the position of each object in the output sequence.
- After placing each element at its correct position, decrease its count by one.

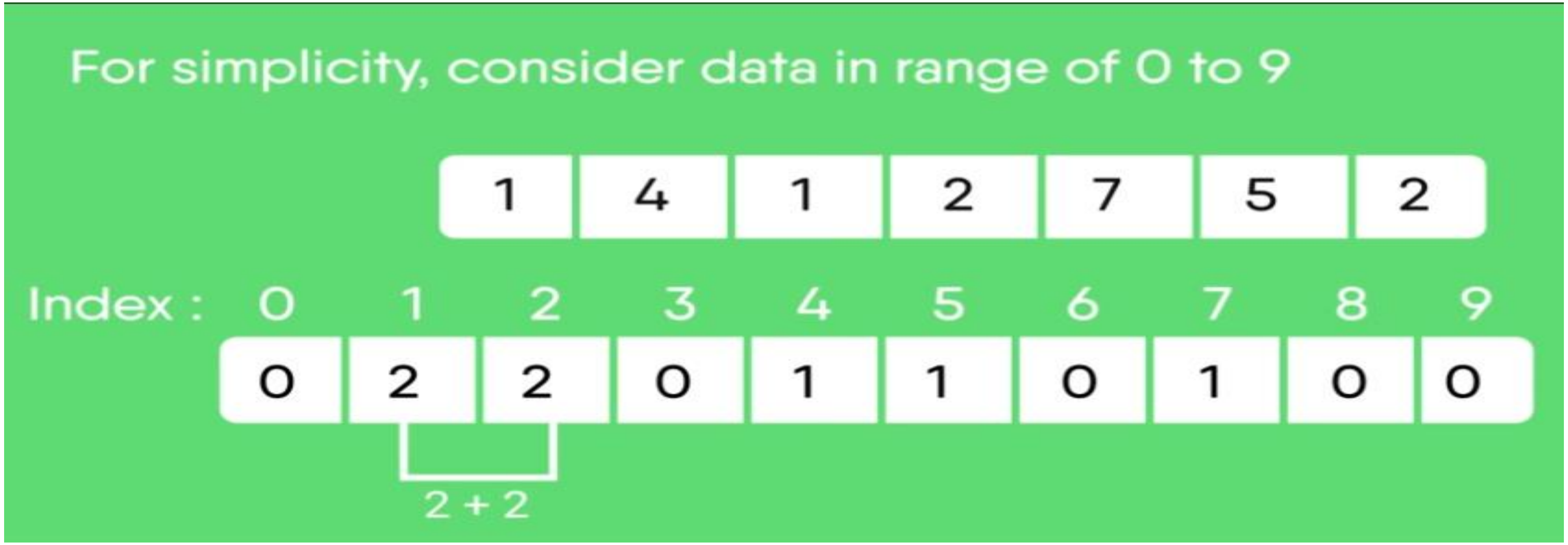
Counting Sort

Let us take an **auxiliary array/count array** from 0 to 9 for simplicity

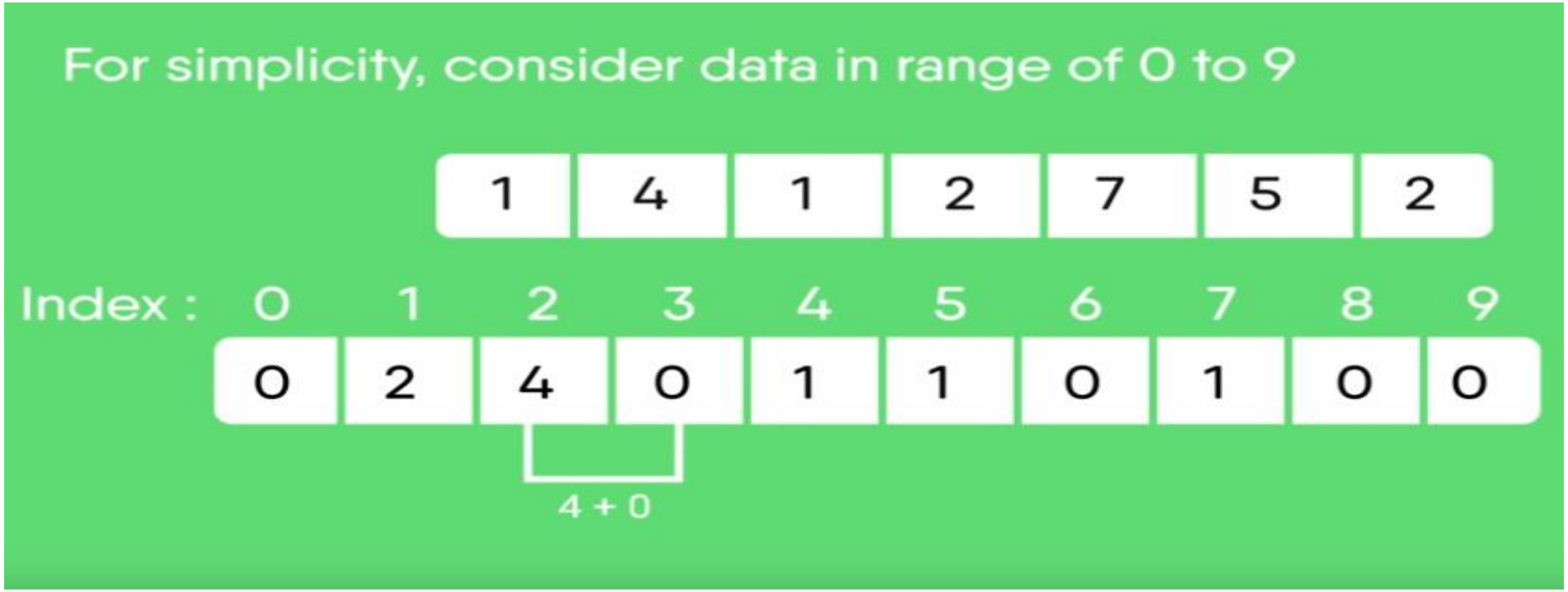
Counting Sort



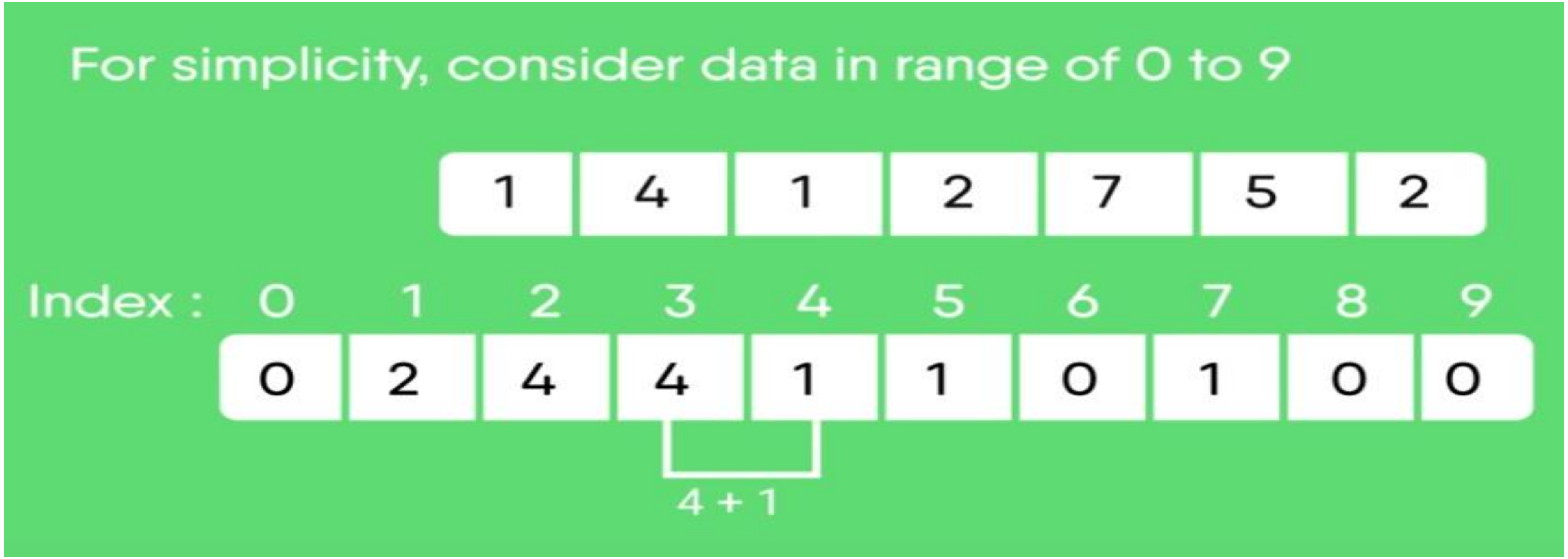
Counting Sort



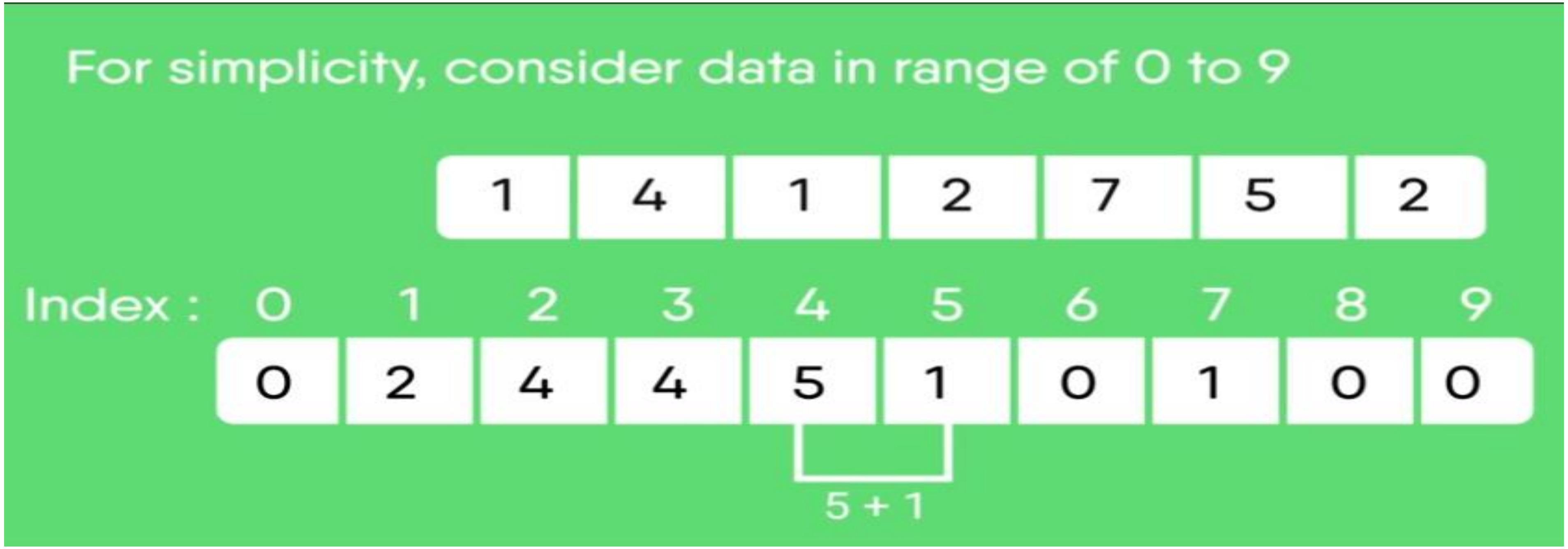
Counting Sort



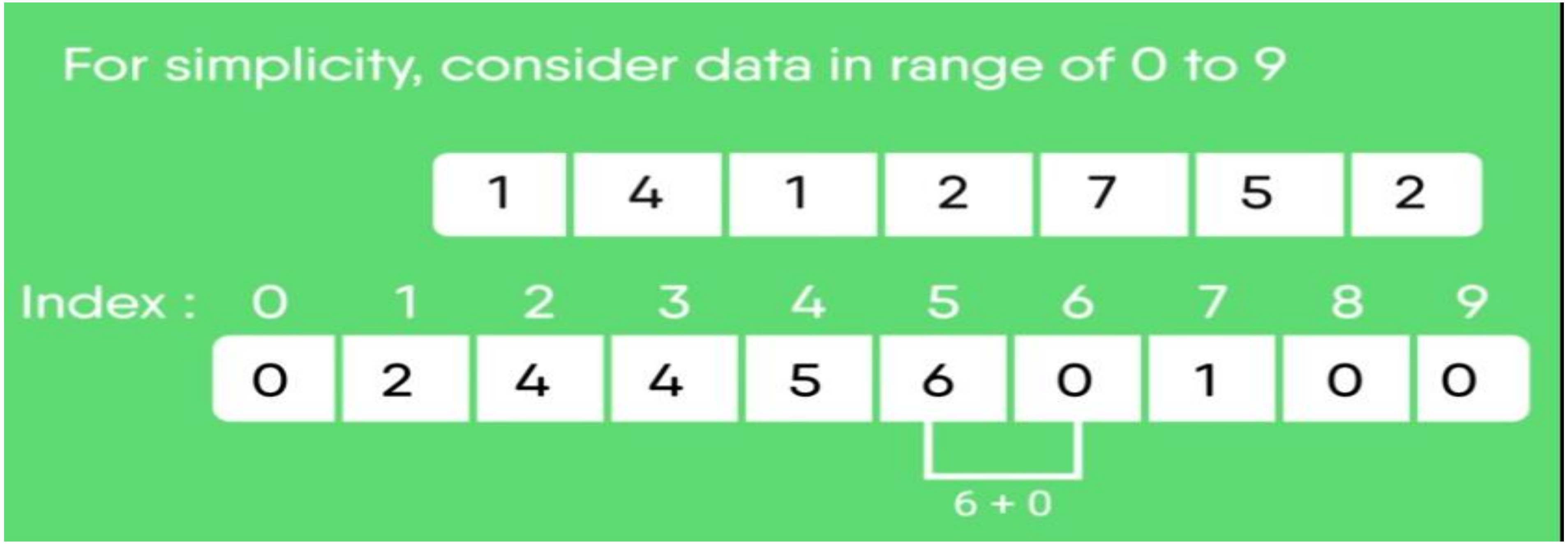
Counting Sort



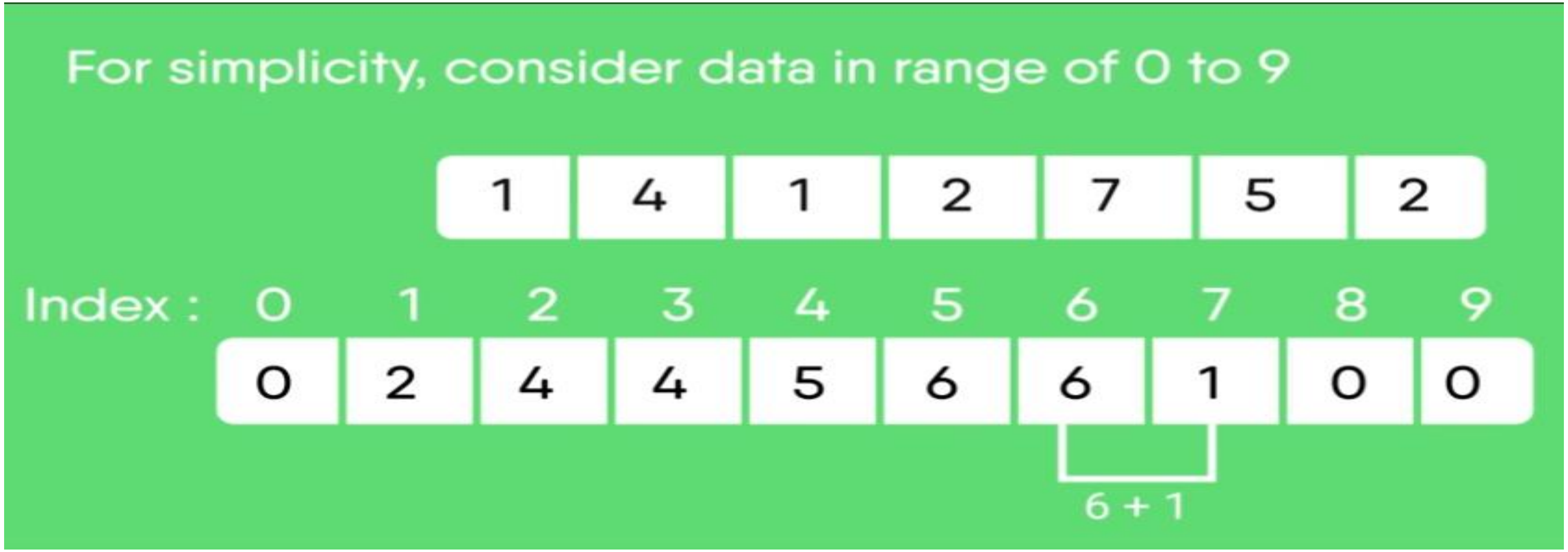
Counting Sort



Counting Sort



Counting Sort

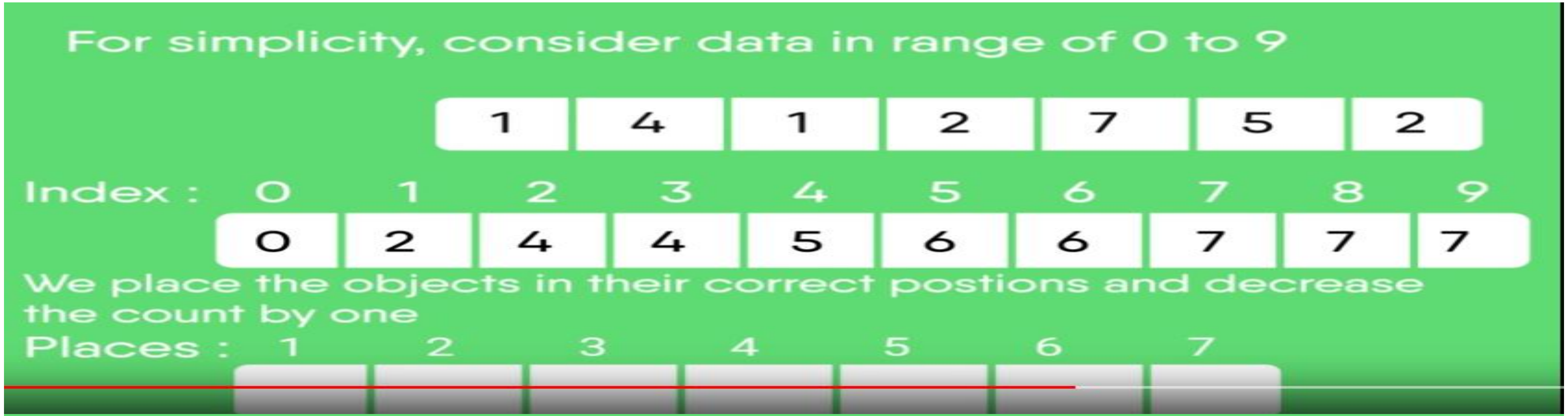


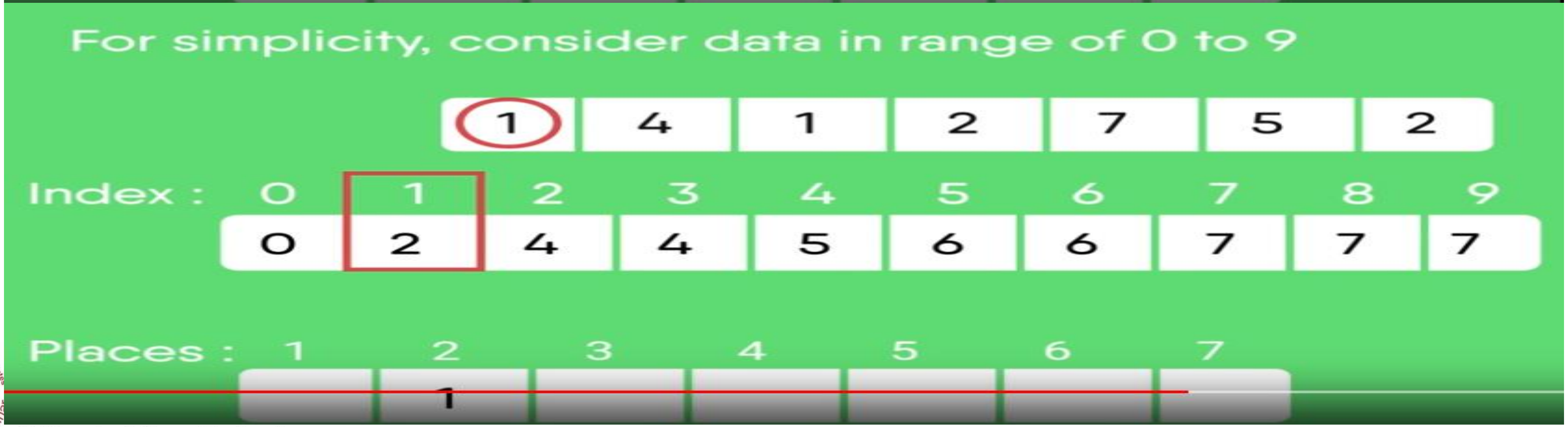
Counting Sort

Counting Sort

Counting Sort

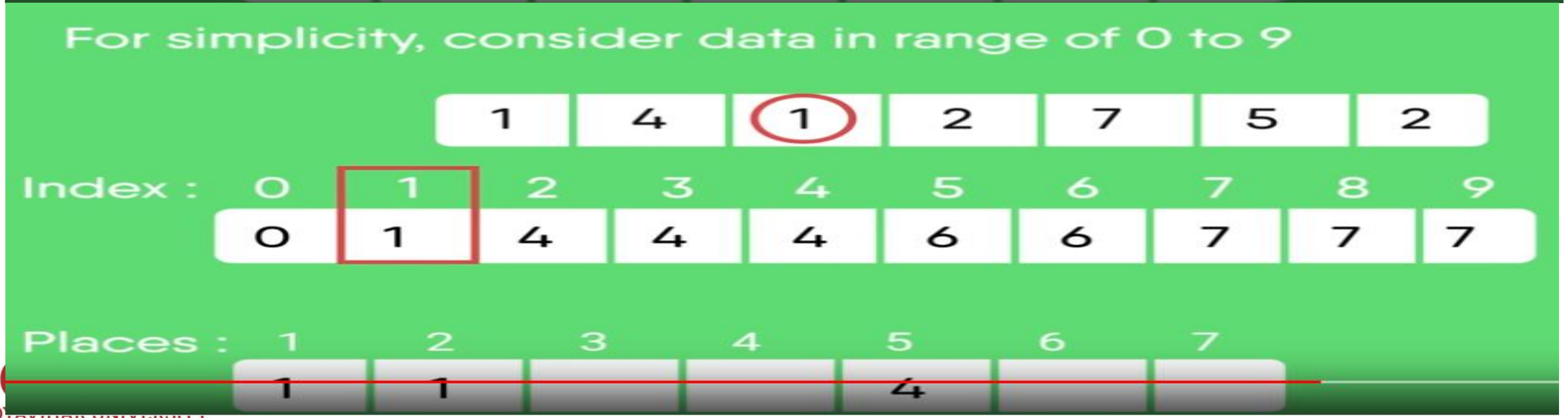
Counting Sort



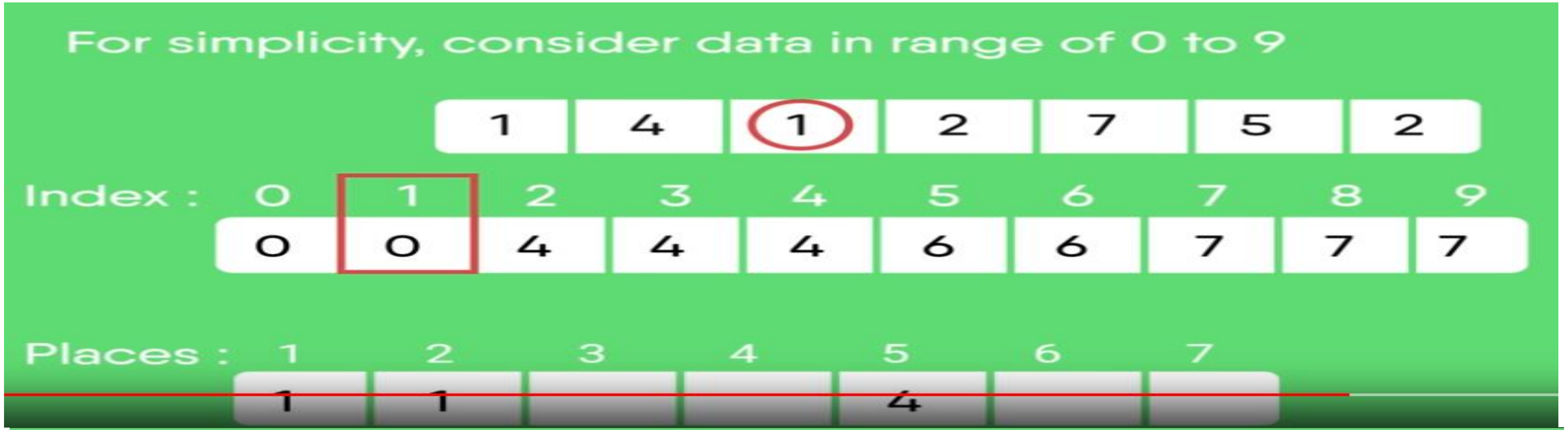


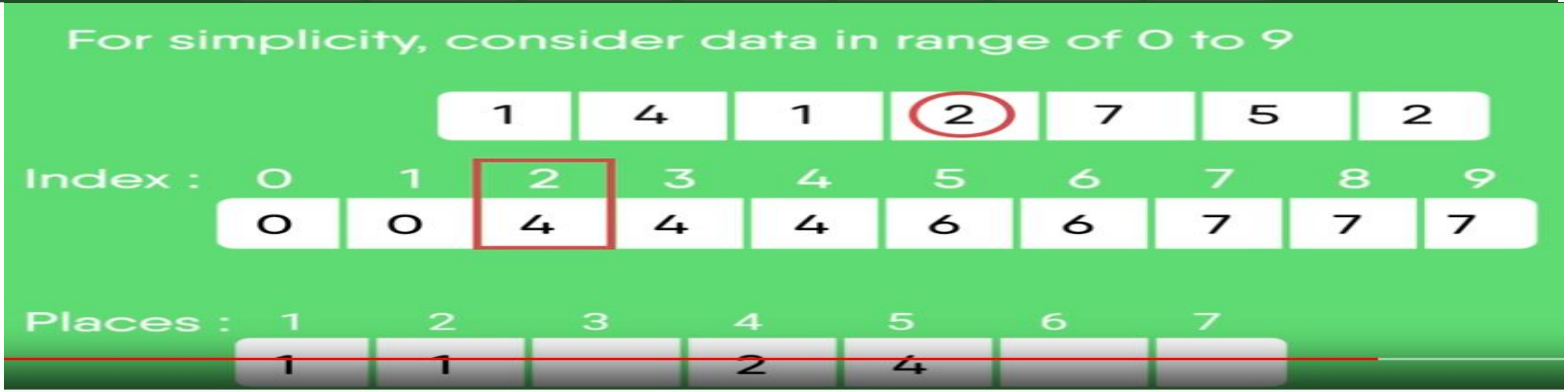
Counting Sort

Counting Sort

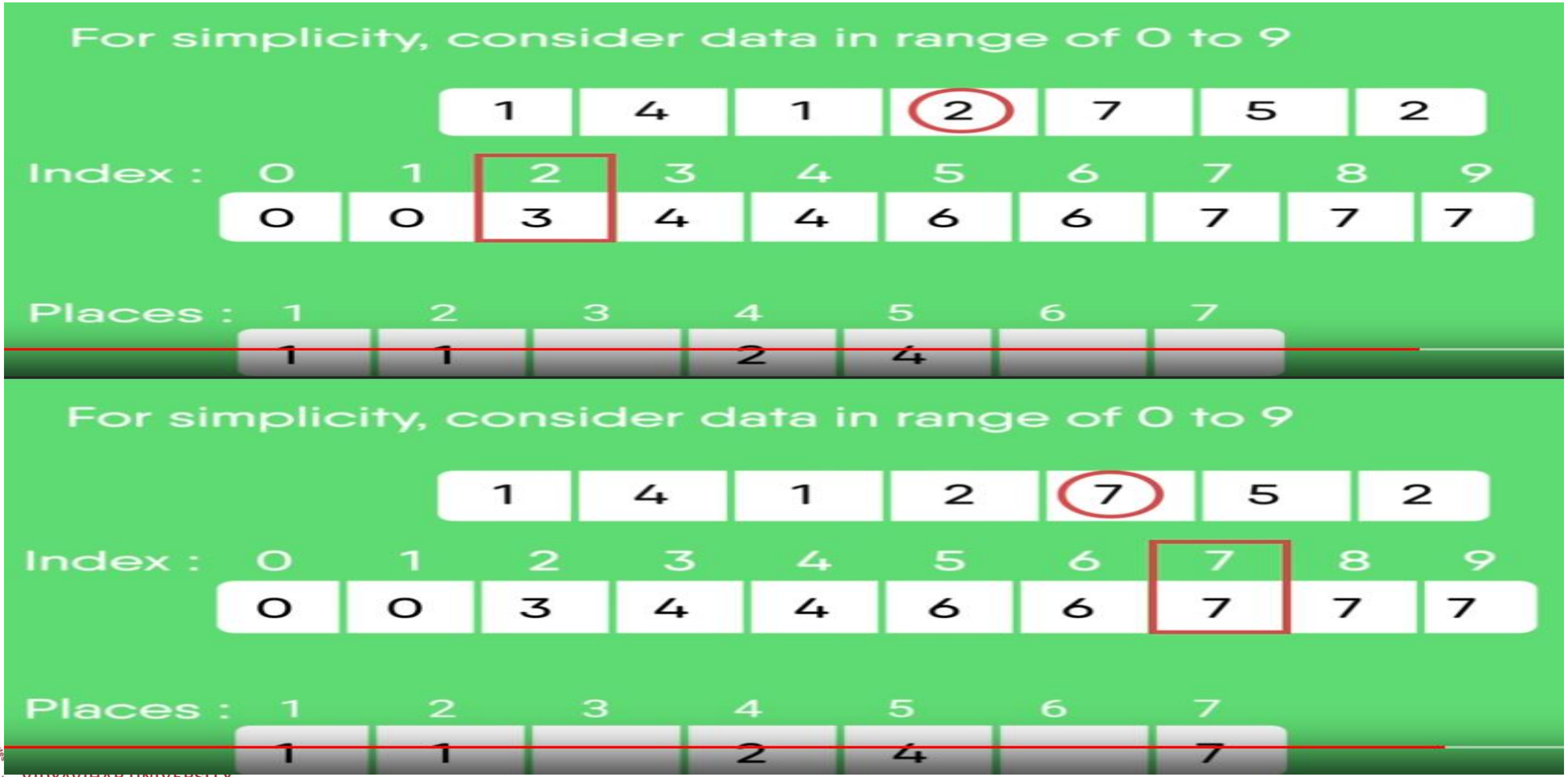


Counting Sort

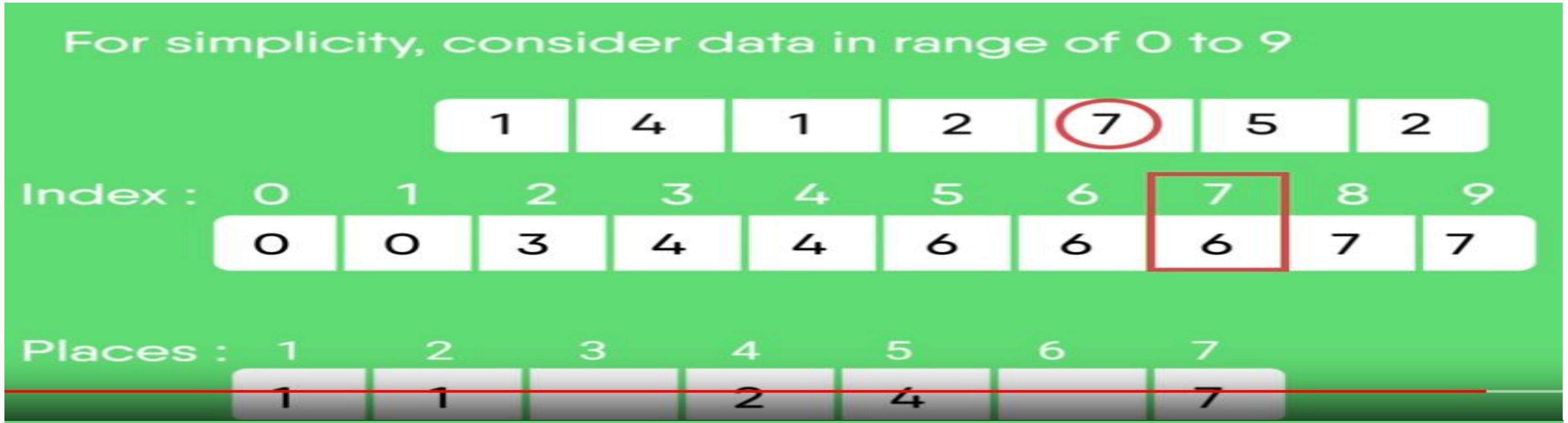


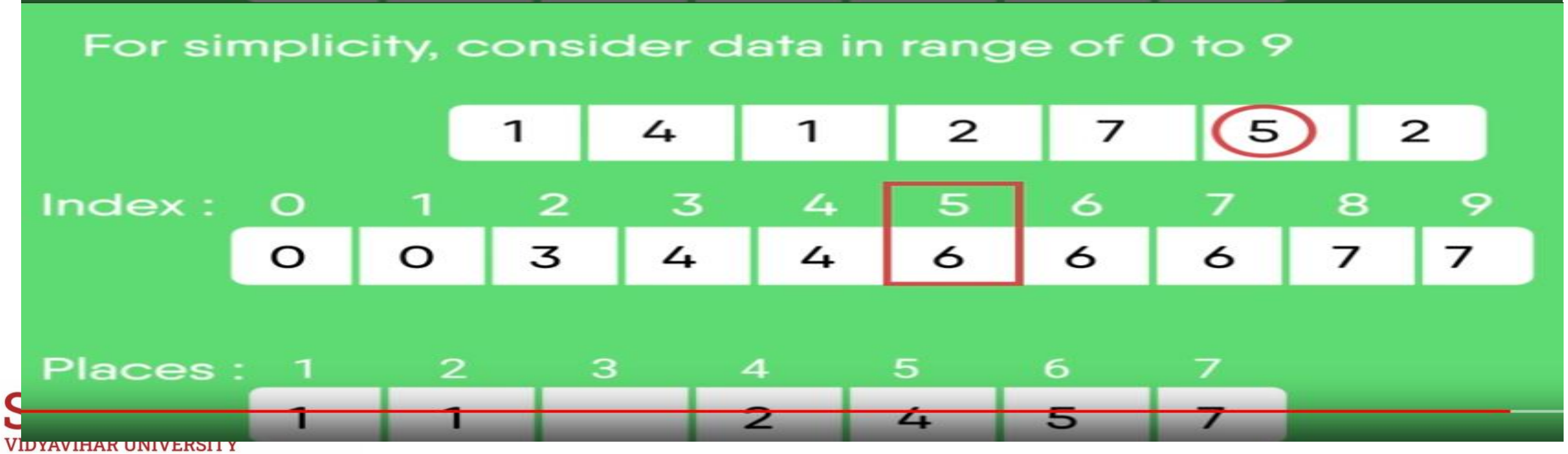


Counting Sort

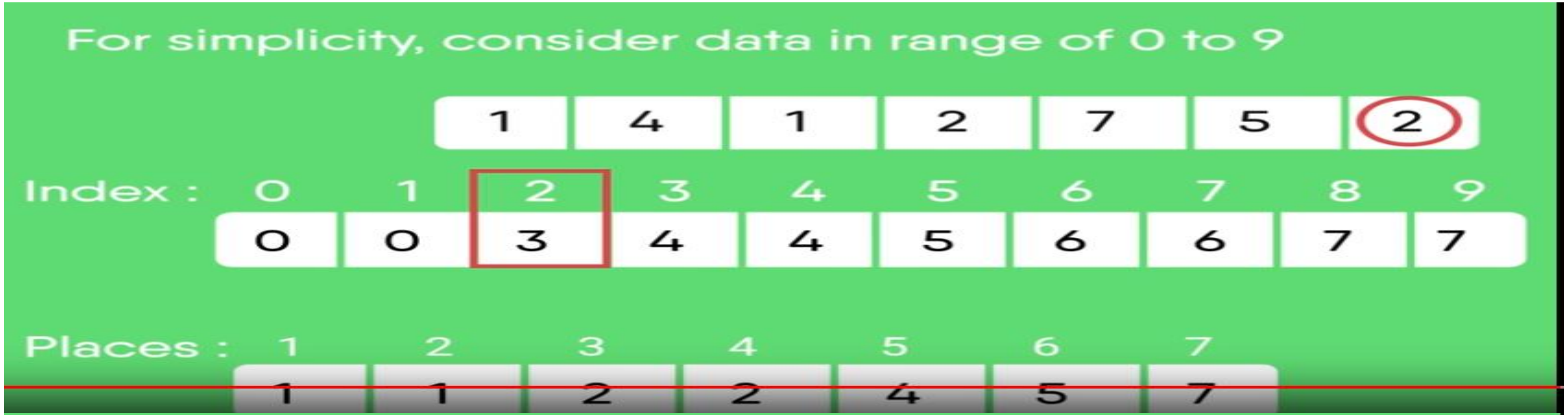


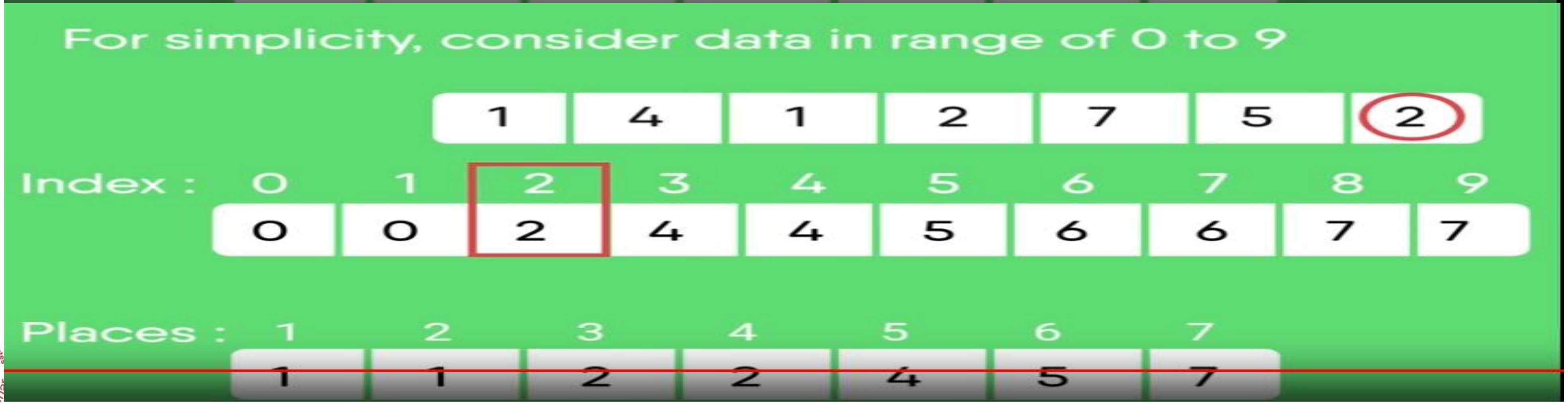
Counting Sort



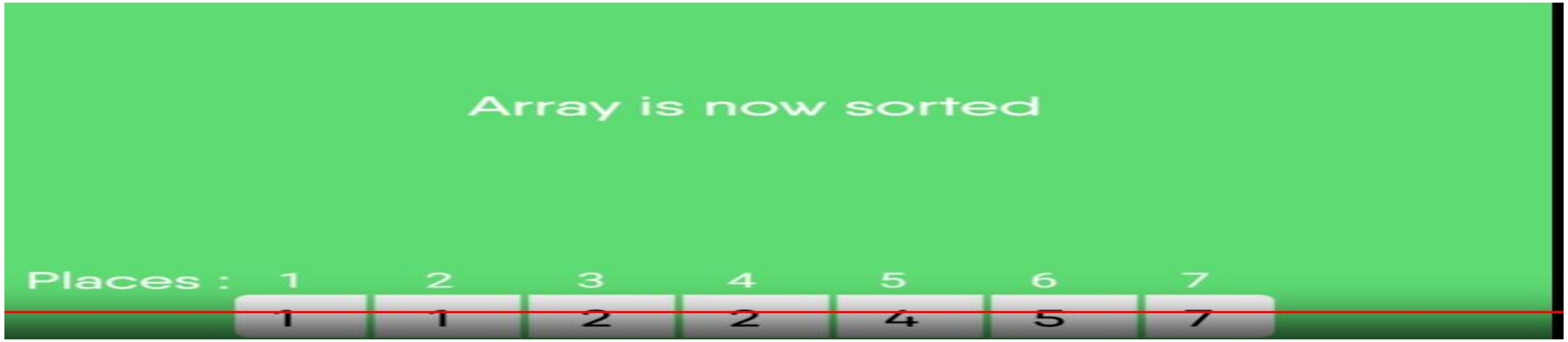


Counting Sort





Counting Sort



Example 2

8	**4**	**2**	**2**	**8**	**3**	**3**	**1**

Sort the array of 8 elements using Counting Sort

Example 2

8	**4**	**2**	**2**	**8**	**3**	**3**	**1**

Let us take an auxiliary array/count array from 0 to 9 for simplicity

Example 2

8	4	2	2	8	3	3	1

Sort the array of 8 elements using Counting Sort

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

Example 2

8	**4**	**2**	**2**	**8**	**3**	**3**	**1**

Sort the array of 8 elements using Counting Sort

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

Cumulative Count array –

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

Example 2

8	4	2	2	8	3	3	1

Cumulative Count array –

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

- Place the objects in the correct place and decrease the count by 1
- Check for Object 8, Count=8,
- Place at 8th Position and decrement count by 1

1	2	3	4	5	6	7	8
							8

- Cumulative Count Array-

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

Example 2

8	4	2	2	8	3	3	1

Cumulative Count array –

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

- Place the objects in the correct place and decrease the count by 1
- Check for Object 4, Count=6,
- Place at 6th Position and decrement count by 1

1	2	3	4	5	6	7	8
					4		8

- Cumulative Count Array-

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

Example 2

8	4	2	2	8	3	3	1

Cumulative Count array –

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

- Place the objects in the correct place and decrease the count by 1
- Check for Object 2, Count=3,
- Place object at 3th Position and decrement count by 1

1	2	3	4	5	6	7	8
		2			4		8

- Cumulative Count Array-

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

Example 2

8	4	2	2	8	3	3	1

Cumulative Count array –

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

- Place the objects in the correct place and decrease the count by 1
- Check for Object 2, Count=2,
- Place object at 2nd Position and decrement count by 1

1	2	3	4	5	6	7	8
	2	2			4		8

- Cumulative Count Array-

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

Example 2

8	4	2	2	8	3	3	1

Cumulative Count array –

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

- Place the objects in the correct place and decrease the count by 1
- Check for Object 8, Count=7,
- Place object at 7th Position and decrement count by 1

1	2	3	4	5	6	7	8
	2	2			4	8	8

- Cumulative Count Array-

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

Example 2

8	4	2	2	8	3	3	1

Cumulative Count array –

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

- Place the objects in the correct place and decrease the count by 1
- Check for Object 3, Count=5,
- Place object at 5th Position and decrement count by 1

1	2	3	4	5	6	7	8
	2	2		3	4	8	8

- Cumulative Count Array-

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

Example 2

8	4	2	2	8	3	3	1

Cumulative Count array –

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

- Place the objects in the correct place and decrease the count by 1
- Check for Object 3, Count=4,
- Place object at 5th Position and decrement count by 1

1	2	3	4	5	6	7	8
	2	2	3	3	4	8	8

- Cumulative Count Array-

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

Example 2

8	4	2	2	8	3	3	1

Cumulative Count array –

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

- Place the objects in the correct place and decrease the count by 1
- Check for Object 1, Count=1,
- Place object at 1st Position and decrement count by 1

1	2	3	4	5	6	7	8
1	2	2	3	3	4	8	8

- Cumulative Count Array-

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

Example 2

Input Array-

8	4	2	2	8	3	3	1

Sorted array-

Position	1	2	3	4	5	6	7	8
Value	**1**	**2**	**2**	**3**	**3**	**4**	**8**	**8**

Hashing

- we discussed two search algorithms: *linear search* and *binary search*. Linear search has a running time proportional to O(n), while binary search takes time proportional to O(log n),where n is the number of elements in the array.
- Binary search and binary search trees are efficient algorithms to search for an element.
- But what if we want to perform the search operation in time proportional to O(1)? In other words, is there a way to search an array in constant time, irrespective of its size?

Hashing

- we can directly access the record of any employee, once we know his Emp_ID, because the array index is the same as the Emp_ID number.

Key	Array of Employees' Records
Key 0 → [0]	Employee record with Emp_ID 0
Key 1 → [1]	Employee record with Emp_ID 1
Key 2 → [2]	Employee record with Emp_ID 2
....	....
....	....
Key 98 → [98]	Employee record with Emp_ID 98
Key 99 → [99]	Employee record with Emp_ID 99

Hashing

Let us assume that the same company uses a five-digit Emp_ID as the primary key. Key values will range from 00000 to 99999. If we want to use the same technique as above, we need an array of size 100,000, of which only 100 elements will be used. Waste so much storage space.

Key	Array of Employees' Records
Key 00000 → [0]	Employee record with Emp_ID 00000
....	....
Key n → [n]	Employee record with Emp_ID n
....	....
Key 99998 → [99998]	Employee record with Emp_ID 99998
Key 99999 → [99999]	Employee record with Emp_ID 99999

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engi

Hashing

- 100 employees in the company.
- Good option is to use just the last two digits of the key to identify each employee.
- For ex, the employee with Emp_ID 79439 will be stored in the element of the array with index 39. Similarly, the employee with Emp_ID 12345 will have his record stored in the array at the 45th location.
- In this case, we need a way to convert a five-digit key number to a two-digit array index. We need a function which will do the transformation.
- In this case, we will use the term *hash table* for an array and the function that will carry out the transformation will be called a *hash function.*

HASH TABLEs

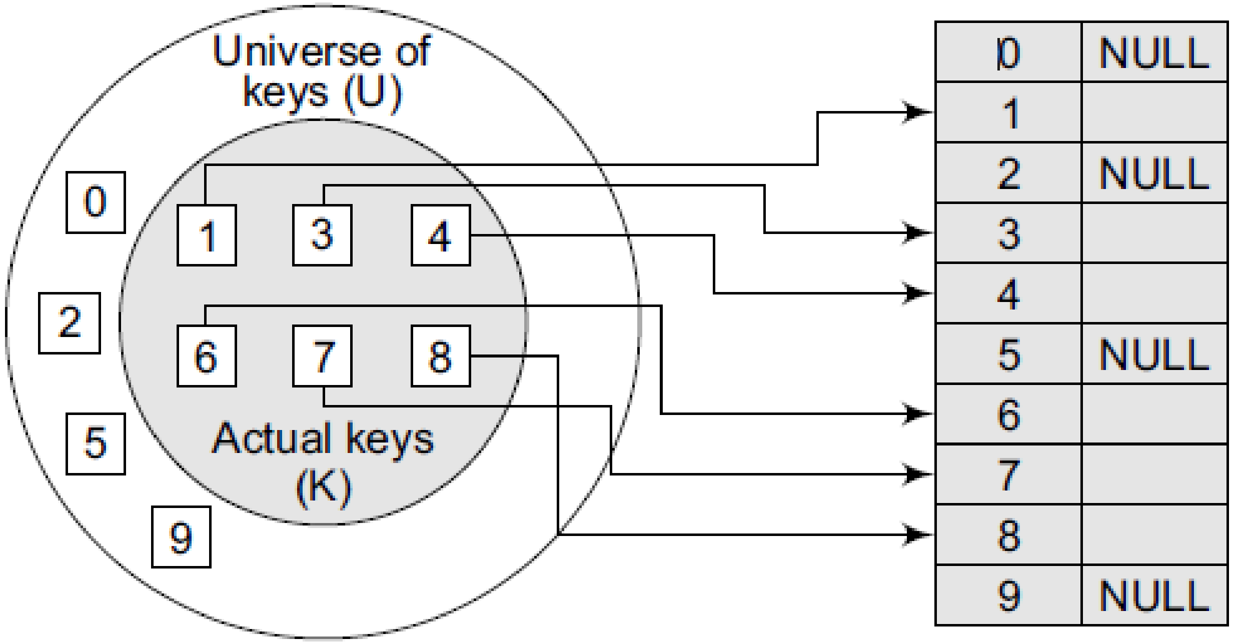
- Hash table is a data structure in which keys are mapped to array positions by a hash function.
- In the example discussed , we will use a hash function that extracts the last two digits of the key.
- Therefore, we map the keys to array locations or array indices. A value stored in a hash table can be searched in O(1) time by using a hash function
- In a hash table, an element with key k is stored at index h(k) and not k.

HASH Function

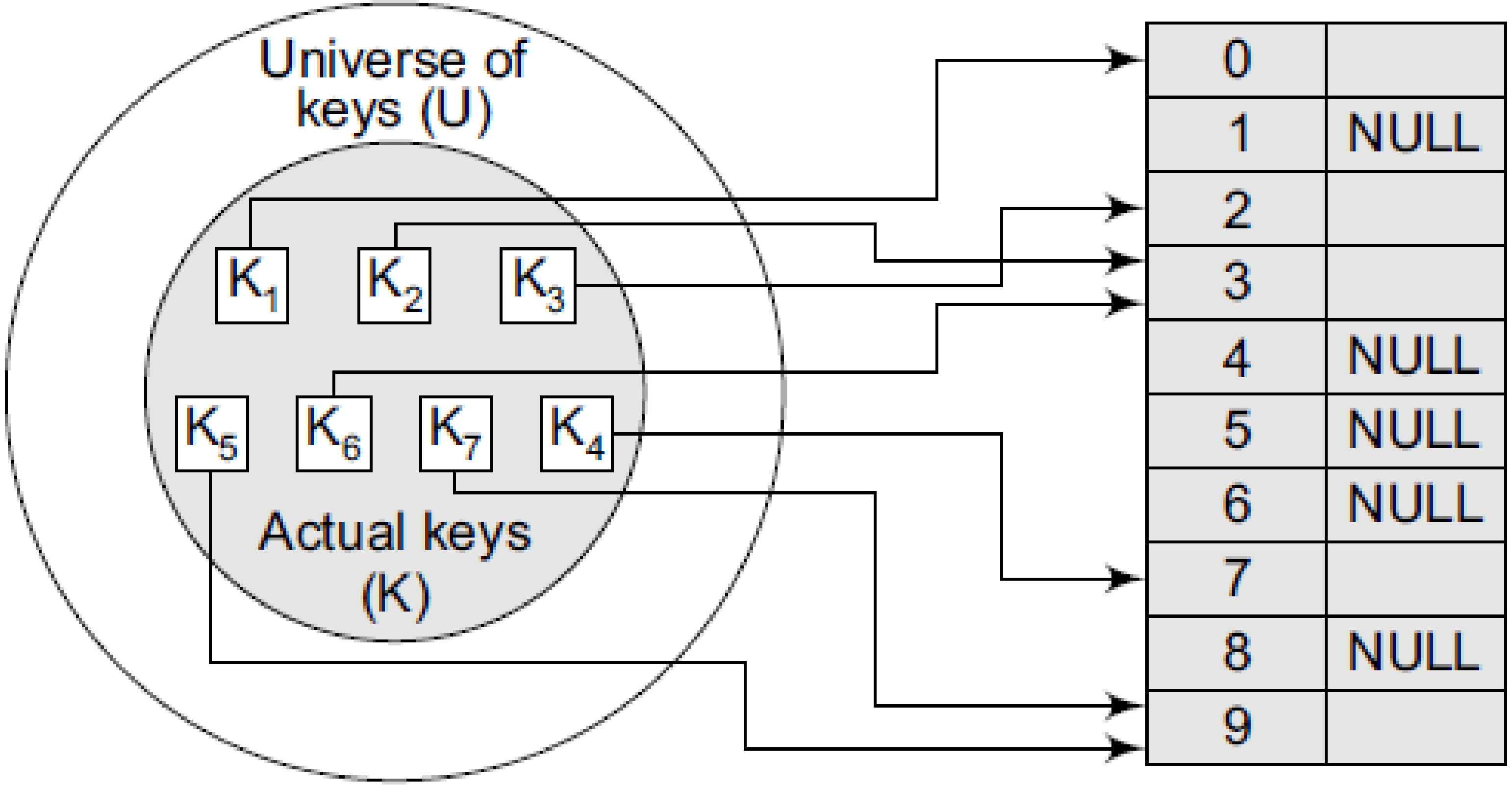
- hash function which generates an address from the key (by producing the index of the array where the value is stored).
- hash function h is used to calculate the index at which the element with key k will be stored. This process of mapping the keys to appropriate locations (or indices) in a hash table is called *hashing*.
- The main goal of using a hash function is to reduce the range of array indices that have to be handled.

Direct relationship between key and index in the array

- Figure shows a direct correspondence between the keys and the indices of the array. This concept is useful when the total universe of keys is small and when most of the keys are actually used from the whole set of keys. This is equivalent to our first example, where there are 100 keys for 100 employees.



Relationship between keys and hash table index



Hashing

- Hash is an important Data Structure which is designed to use a **special function called the Hash function** which is used to map a given value with a particular key for faster access of elements.
- **The efficiency of mapping depends of the efficiency of the hash function used.**
- It is a technique whereby items are placed into a structure based on a **key to-address transformation.**

HASH Function

- A hash function is a mathematical formula which, when applied to a key, produces an integer which can be used as an index for the key in the hash table.
- The main aim of a hash function is that elements should be relatively, randomly, and uniformly distributed.
- It produces a unique set of integers within some suitable range in order to reduce the number of collisions.
- In practice, there is no hash function that eliminates collisions completely.
- A good hash function can only minimize the number of collisions by spreading the elements uniformly throughout the array.

Properties of a Good Hash Function

- ***Efficiently computable.***
- ***Uniformity :*** Should uniformly distribute the keys (Each table position equally likely for each key)
- ***Should generate unique addresses or addresses with minimum collision***
- ***Low cost***
- ***Determinism:*** the same hash value must be generated for a given input value

Hashing

For storing record

Key

↓

Generate array index

↓

Store the record on that array index

Hashing

For accessing record

Key

↓

Generate array index

↓

Get the record from that array index

Hashing

Hash Table :-

- A hash table is a data structure that uses a **random access data structure, such as an array,** and a **mapping function, called a hash function,** to allow average constant time **O(1) searches**.

Hash Function :-

- A hash function is a mapping between a set of input values and a set of integers, known as hash values.
- Denoted by H.

H(K)->A

Hashing

1) **Choosing a hash function which ensures minimum collision**
2) **Resolving collision**

Truncation Method

- Easiest method
- **A part of the key as address**
- **Can be rightmost or leftmost digit**

Eg-

82394561, 87139465, 83567271, 85943228

Suppose table size is 100 then take **the 2 rightmost digits for getting the addresses.**

Address will be 61, 65, 71 and 28

Mid Square Method

- The mid-square method is a good hash function which works in two steps:
- *Step 1*: Square the value of the key. That is, find k^2.
- *Step 2*: Extract the middle r digits of the result obtained in Step 1.
- In the mid-square method, the same r digits must be chosen from all the keys.
- Therefore, the hash function can be given as:
- $h(k) = s$
- where s is obtained by selecting r digits from k^2.

Mid Square Method

- Calculate the hash value for keys 1234 and 5642 using the mid-square method. The hash table has 100 memory locations.
- ***Solution***
- The hash table has 100 memory locations whose indices vary from 0 to 99.
- Only two digits are needed to map the key to a location in the hash table, so r = 2.
- When k = 1234, k^2 = 1522756, h (1234) = 27
- When k = 5642, k^2 = 31832164, h (5642) = 21
- Observe that the 3rd and 4th digits starting from the right are chosen.

Mid Square Method

Eg- **1337 , 1273, 1391, 1026**

Square=1787569, 1620529, 1934881, 1052676

Lets take 3rd, 4th digit from each number as address

Let the table size be 100

Address=75, 05,48, 26

Folding Method

- The folding method works in the following two steps:
- *Step 1*: Divide the key value into a number of parts. That is, divide k into parts k1, k2, ..., kn, where each part has the same number of digits except the last part which may have lesser digits than the other parts.
- *Step 2*: Add the individual parts. That is, obtain the sum of k1 + k2 + ... + kn. The hash value is produced by ignoring the last carry, if any.

Folding Method

- Given a hash table of 100 locations, calculate the hash value using folding method for keys 5678, 321, and 34567.
- ***Solution***
- Since there are 100 memory locations to address, we will break the key into parts where each part (except the last) will contain two digits. The hash values can be obtained as shown below:

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

SOMAIYA VIDYAVIHAR UNIVERSITY
K J Somaiya College of Engineering

Somaiya TRUST

Folding Method

- **Break the key into pieces, add them and get the hash address**
- **Truncate the higher digits of the number**

Eg-Lets take some 8 bit address

82394561, 87139465, 83567271, 85943228

Chop them in pieces 3,2 and 3 digits and them

Address will be->

82394561 = 823+94+561 =1478

87139465= 871+39+465 =1375

83567271= 835+67+271 = 1173

85943228 =859+43+228=1130

Address will be->

H(82394561) = 478

H(87139465) = 375

H(83567271) = 173

H(85943228) = 130

Modular Method

- **Perform Modulus operation, Remainder is address of hash table**
- Ensure address will be in range of hash table
- **Take table size as a prime number**
- Let us take some keys: 82394561, 87139465, 83567271, 85943228
- Table size=97

Address=

82394561%97=45

87139465%%97=0

83567271%97=25

859432285%97=64

Hashing

- Collisions occur when the hash function maps two different keys to the same location. Obviously, two records cannot be stored in the same location.
- Therefore, a method used to solve the problem of collision, also called *collision resolution technique*, is applied.
- The two most popular methods of resolving collisions are:
- 1. Open addressing
- 2. Chaining

Open Addressing/ Closed Hashing

- Once a collision takes place, open addressing or closed hashing computes new positions using a probe sequence and the next record is stored in that position. In this technique, all the values are stored in the hash table.
- Hash table contains two types of values: *sentinel values* (e.g., –1) & *data values*.
- The presence of a sentinel value indicates that the location contains no data value at present but can be used to hold a value.
- When a key is mapped to a particular memory location, then the value it holds is checked. If it contains a sentinel value, then the location is free and the data value can be stored in it. However, if the location already has some data value stored in it, then other slots are examined systematically in the forward direction to find a free slot. If even a single free location is not found, then we have an OVERFLOW condition.
- The process of examining memory locations in the hash table is called *probing*..

Open Addressing/ Closed Hashing

Open addressing technique can be implemented

- linear probing,
- quadratic probing,
- double hashing, and
- rehashing.

Linear probing

- The simplest approach to resolve a collision is linear probing.
- In this technique, if a value is already stored at a location generated by h(k), then the following hash function is used to resolve the collision:
- $h(k, i) = [h'(k) + i] \bmod m$
- Where m is the size of the hash table, $h'(k) = (k \bmod m)$,
- i is the probe number that varies from 0 to m–1.

Linear probing

- First the location generated by [h'(k) mod m] is probed for the first time i=0.
- If the location is free, the value is stored in it, else the second probe generates the address of the location given by [h'(k) + 1]mod m.
- Similarly, if the location is occupied, then subsequent probes generate the address as
- [h'(k) + 2]mod m,
- [h'(k) + 3]mod m,
- [h'(k) + 4]mod m,
- [h'(k)+ 5]mod m, and so on, until a free location is found.

Linear probing

- Consider a hash table of size 10. Using linear probing, insert the keys 72, 27,36, 24, 63, 81, 92, and 101 into the table. m = 10
- Solution
- Initially, the hash table can be given as:

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

Linear probing

Step 1 Key = 72

$$h(72, 0) = (72 \bmod 10 + 0) \bmod 10$$
$$= (2) \bmod 10$$
$$= 2$$

Since T[2] is vacant, insert key 72 at this location.

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	–1	–1	–1	–1	–1	–1

Linear probing

Step 2 Key = 27

h(27, 0) = (27 mod 10 + 0) mod 10

= (7) mod 10

= 7

Since T[7] is vacant, insert key 27 at this location.

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	–1	–1	–1	27	–1	–1

Linear probing

Step 3 Key = 36

$$h(36, 0) = (36 \bmod 10 + 0) \bmod 10$$
$$= (6) \bmod 10$$
$$= 6$$

Since T[6] is vacant, insert key 36 at this location.

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	–1	–1	36	27	–1	–1

Linear probing

Step 4 Key = 24

$h(24, 0) = (24 \bmod 10 + 0) \bmod 10$

$= (4) \bmod 10$

$= 4$

Since T[4] is vacant, insert key 24 at this location.

0	1	2	3	4	5	6	7	8	9
–1	–1	72	–1	24	–1	36	27	–1	–1

Linear probing

Step 5 Key = 63

$$h(63, 0) = (63 \bmod 10 + 0) \bmod 10$$
$$= (3) \bmod 10$$
$$= 3$$

Since T[3] is vacant, insert key 63 at this location.

0	1	2	3	4	5	6	7	8	9
–1	–1	72	63	24	–1	36	27	–1	–1

Linear probing

Step 6 Key = 81

$$h(81, 0) = (81 \bmod 10 + 0) \bmod 10$$
$$= (1) \bmod 10$$
$$= 1$$

Since T[1] is vacant, insert key 81 at this location.

0	1	2	3	4	5	6	7	8	9
0	81	72	63	24	–1	36	27	–1	–1

Linear probing

Step 7

$$
\begin{aligned}
\text{Key} &= 92 \\
h(92, 0) &= (92 \bmod 10 + 0) \bmod 10 \\
&= (2) \bmod 10 \\
&= 2
\end{aligned}
$$

Now `T[2]` is occupied, so we cannot store the key 92 in `T[2]`. Therefore, try again for the next location. Thus probe, `i = 1`, this time.

$$
\begin{aligned}
\text{Key} &= 92 \\
h(92, 1) &= (92 \bmod 10 + 1) \bmod 10 \\
&= (2 + 1) \bmod 10 \\
&= 3
\end{aligned}
$$

Linear probing

$$h(92, 2) = (92 \bmod 10 + 2) \bmod 10$$
$$= (2 + 2) \bmod 10$$
$$= 4$$

I = 2

Key = 92

$$h(92, 3) = (92 \bmod 10 + 3) \bmod 10$$
$$= (2 + 3) \bmod 10$$
$$= 5$$

I = 3

Linear probing

- **Step 8** Key = 101
- h(101, 0) = (101 mod 10 + 0) mod 10

  = (1) mod 10

  = 1
- we cannot store the key 101 in T[1].
- The procedure will be repeated until the hash function generates the address of location 8 which is vacant and can be used to store the value in it.

Quadratic probing

- In this technique, if a value is already stored at a location generated by h(k), then the following hash function is used to resolve the collision:
- $h(k, i) = [h'(k) + c_1\, i + c_2\, i^2\ ] \bmod m$
- Where m is the size of the hash table, h'(k) = (k mod m),
- i is the probe number that varies from 0 to m–1.
- c_1 and c_2 are constants such that c1 and c2 ≠ 0

Quadratic probing

- Consider a hash table of size 10. Using quadratic probing, insert the keys 72, 27, 36, 24, 63, 81, and 101 into the table. Take c1 = 1 and c2 = 3
- Solution : i = 0 and m =10
- $h(k, i) = [h'(k) + c_1 i + c_2 i^2] \bmod m$
- $h(k, i) = [h'(k)\ \ ] \bmod m$
- $h(k, i) = [k \bmod m] \bmod m$

Quadratic probing

- Show all steps.....

- The hash table now becomes

0	1	2	3	4	5	6	7	8	9
–1	81	72	63	24	–1	36	27	–1	–1

Quadratic probing

- Key = 101
- Since T[1] is already occupied, the key 101 cannot be stored in T[1]. Therefore, try again for next location. Thus probe, i = 1, this time
- h(101,1) = [101 mod 10 + 1 * 1 + 3 * 1] mod 10
- =5
- Since T[5] is vacant, insert the key 101 in T[5]. The hash table now becomes:

0	1	2	3	4	5	6	7	8	9
–1	81	72	63	24	101	36	27	–1	–1

Quadratic probing

Elements=29,18,43,10,46,54

Table size=11

H(29)=29%11=7

H(18)=18%11=7, collision h+1*1=8

H(43)=43%11=10

H(10)=10%11=10, collision h+ 1*1=11%size=11%11=0

0	
1	
2	
3	
4	
5	
6	
7	29
8	18
9	
10	

0	**10**
1	
2	
3	
4	
5	
6	
7	29
8	18
9	
10	**43**

Quadratic probing

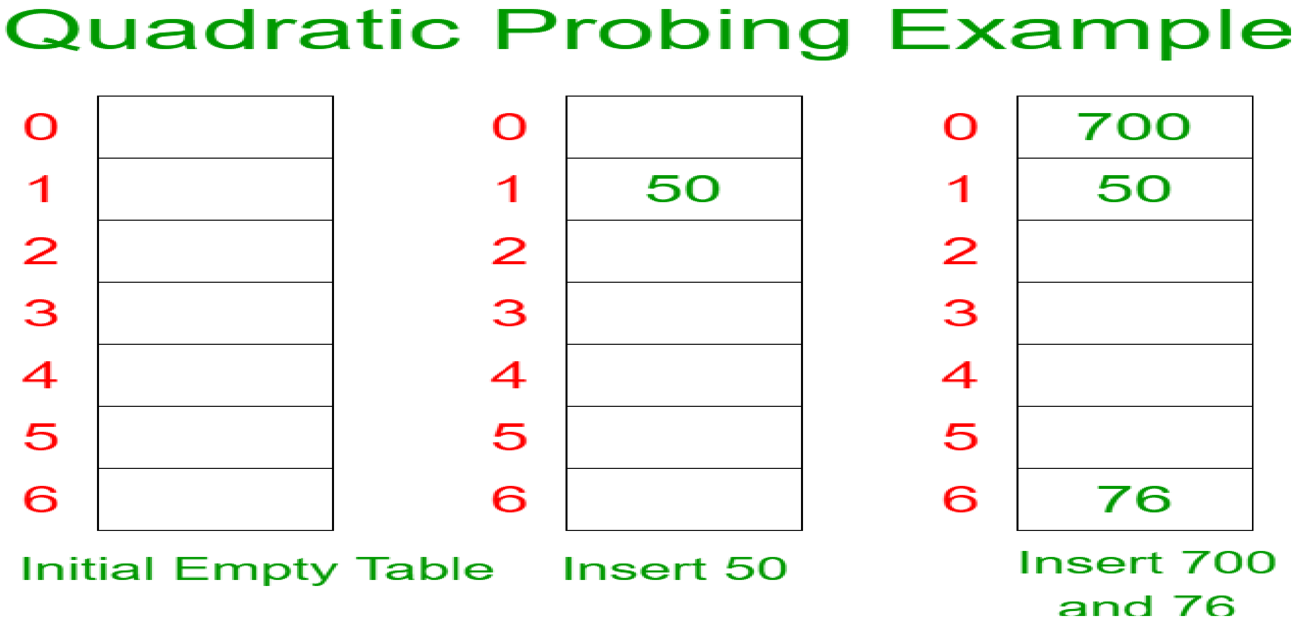
H(46)=46%11=2

H(54)=54%11=10, collision h + 2*2=h+4=14%11=3

0	10
1	
2	46
3	54
4	
5	
6	
7	29
8	18
9	
10	43

Quadratic probing

- Let us consider a simple hash function as “**key mod 7**” and sequence of keys as **50, 700, 76, 85, 92, 73, 101**.
- H(50)=50%7=1
- H(700)=700%7=0
- H(76)=76%7=6



Quadratic probing

- H(50)=50%7=1 H(85)=85%7=1, Collision (h+1)=2
- H(700)=700%7=0 H(92)=92%7=1 , Collision again (h+4)=5
- H(76)=76%7=6

0	700
1	50
2	85
3	
4	
5	
6	76

Insert 85:

Collision occurs.

Insert at 1 + 1*1 position

0	700
1	50
2	85
3	
4	
5	92
6	76

Insert 92:

Collision occurs at 1.

Collision occurs at 1 + 1*1 position

Insert at 1 + 2*2 position.

Quadratic probing

- H(73)=73%7=3
- H(101)=101%7=3, collision (h+1)=4

0	700
1	50
2	85
3	73
4	101
5	92
6	76

Insert 73 and 101

Double Hashing

- In double hashing, we use two hash functions rather than a single function. The hash function in the case of double hashing can be given as:
- $h(k, i) = [h_1 (k) + ih_2 (k)] \bmod m$
- $h_1 (k) = k \bmod m$, $h_2 (k) = k \bmod m'$, choose $m' = m–1$ or $m–2$

Double Hashing

- Consider a hash table of size = 10. Using double hashing, insert the keys 72, 27, 36, 24, 63, 81, 92, and 101 into the table.
- Show all steps……
- Take h1 = (k mod 10) and h2 = (k mod 8)

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

0	1	2	3	4	5	6	7	8	9
–1	81	72	63	24	–1	36	27	–1	–1

Double Hashing

- Key = 92
- h(92, 0) = [92 mod 10 + (0 * 92 mod 8)] mod 10
- = [2 + (0 * 4)] mod 10
- = 2 mod 10 = 2
- Now T[2] is occupied, so we cannot store the key 92 in T[2]. Therefore, try again for the next location. Thus probe, i = 1, this time. Key = 92
- h(92, 1) = [92 mod 10 + (1 * 92 mod 8)] mod 10
- = 6

Double Hashing

- Now T[6] is occupied, so we cannot store the key 92 in T[6]. Therefore, try again for the next location.
- Thus probe, i = 2, this time. Key = 92
- h(92, 2) = [92 mod 10 + (2 * 92 mod 8)] mod 10
- = [2 + (2 * 4)] mod 10
- = [2 + 8] mod 10
- = 10 mod 10
- = 0
- Since T[0] is vacant, insert the key 92 in T[0].

Double Hashing

- Now T[1] is occupied, so we cannot store the key 101 in T[1]. Therefore, try again for the next location.
- Thus probe, i = 1, this time.
- Now T[6] is occupied, so we cannot store the key 101 in T[6]. Therefore, try again for the next location.
- probe i = 2. Repeat the entire process until a vacant location is found.

Rehashing

- .

Rehashing

- Chances of Insertion Failure when table is full
- **Soln=>**
- **Create a new hash table with double size of previous hash table**
- Use the new hash function and Insert all the elements of the previous hash table in the new table

ReHashing

- Scan the elements of the previous hash table one by one
- Calculate the hash key with new hash function
- Insert them in new hash table

Hashing

Rehashing

Elements=7,18,43,10,36,25

Table size=11

H(7)=7%11=7

H(18)=18%7=7, collision

H(43)=43%11=10

H(36)=36%11=3

H(10)=10%11=10, collision

H(25)=25%11=3

0	**10**
1	
2	
3	**36**
4	**25**
5	
6	
7	**7**
8	**18**
9	
10	**43**

Hashing

Rehashing

Elements=7,18,43,10,36,25

Table size=11

Now insert 46

H(46)=46%11=2

0	10
1	
2	46
3	36
4	25
5	
6	
7	7
8	18
9	
10	43

Hashing

Rehashing

- Now to perform Rehashing
- Create a new table
- **For Size of New Hash table, We choose the nearest bigger prime number to double the size of the original hash table**

0	10
1	
2	46
3	36
4	25
5	
6	
7	7
8	18
9	
10	43

Hashing

Rehashing

- Original Size=11
- Double Size=22
- Nearest Bigger prime no=23
- New Hash function:-
- **H(K)=K mod 23**

0	10
1	
2	46
3	36
4	25
5	
6	
7	7
8	18
9	
10	43

Hashing

Rehashing

- New Hash Table-
- **H(K)=K mod 23**

H(7)=7%23=7

H(18)=18%23=18

H(43)=43%23=20

H(36)=36%23=13

H(25)=25%23=2

Now insert 46

H(46)=46%23=23%SIZE=23%23=0

0	46
1	
2	25
3	
4	
5	
6	
7	7
8	
9	
10	10
11	
12	
13	36
14	
15	
16	
17	
18	18
19	
20	43
21	
22	

Hashing

ReHashing

- Bit more expensive technique but it happens very few times
- Decision of rehashing can be taken on different conditions like
    - **Table is occupied more than half,**
    - **Insertion of new element failure or on any given case**

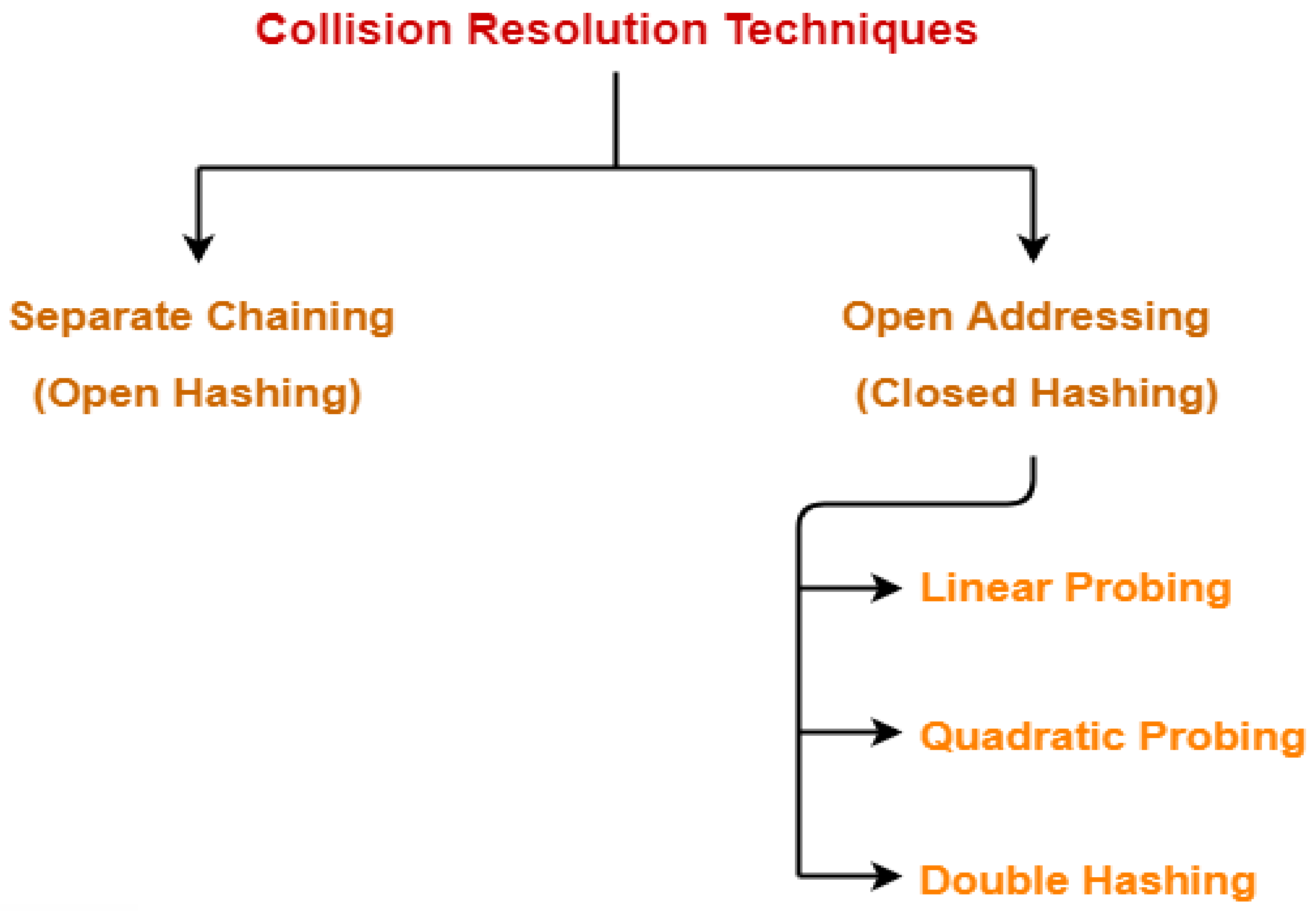
Hashing

Collision Resolution (Open Hashing)

Separate Chaining

- **Maintain chain of elements which have same hash address**
- **Hash table is an array of pointers which point to the linked list**
- **Maintain Linked list in sorted order and the elements which have same hash address will be in this linked list**

Hashing


Collision Resolution Techniques
Separate Chaining
(Open Hashing)
Open Addressing
(Closed Hashing)
Linear Probing
Quadratic Probing
Double Hashing

Hashing

Collision Resolution (Open Hashing)

Separate Chaining

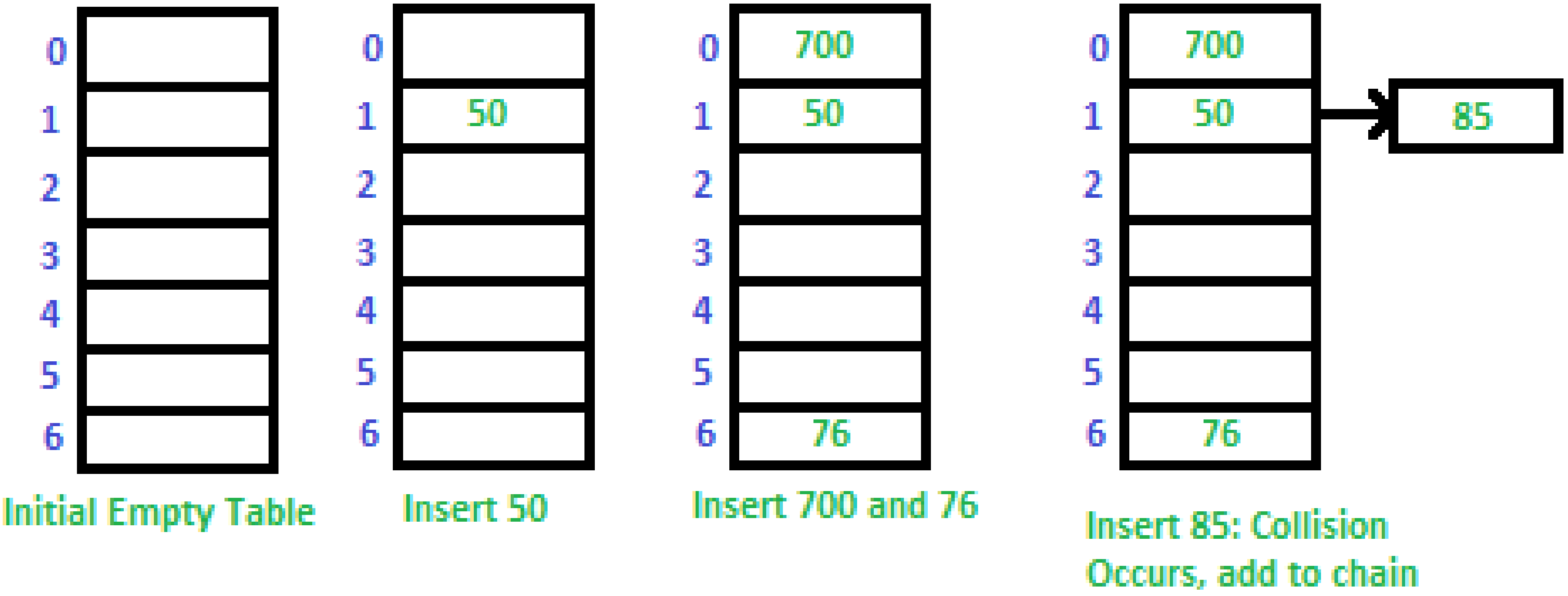
- For inserting one element, we have to get the hash value through the hash function
- Hash value will be mapped in the hash table position then that element will be inserted in the Linked list

- Searching is also same

Example

Separate Chaining

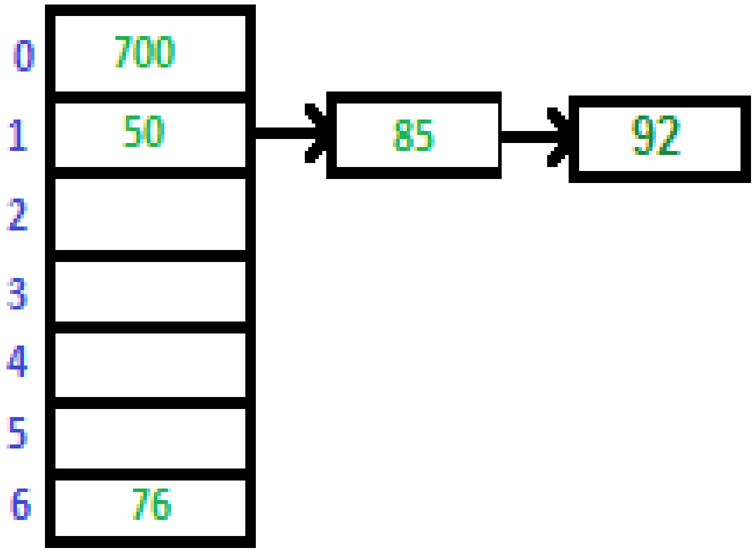
Let us consider a simple hash function as "**key mod 7**" and sequence of keys as 50, 700, 76, 85, 92, 73, 101.



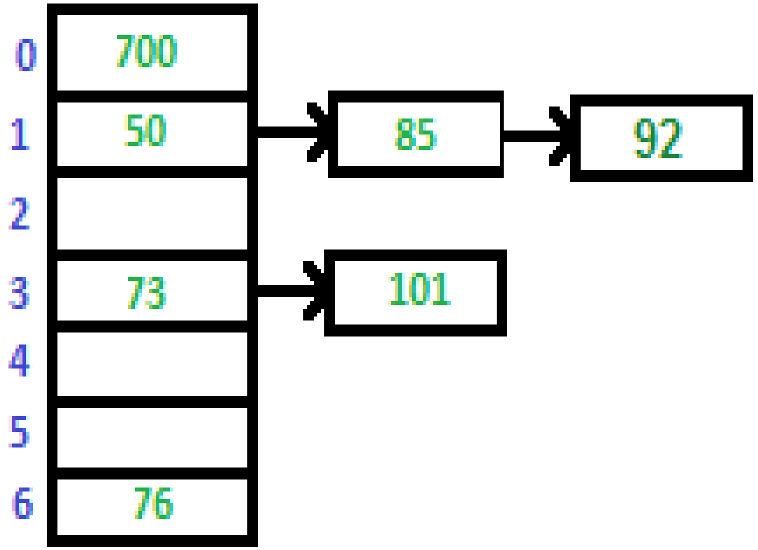
Example

Separate Chaining

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



Inser 92 Collision
Occurs, add to chain



Insert 73 and 101

Hashing

Advantage of Separate Chaining

1) Simple to implement.

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

Hashing

Disadvantages of Separate Chaining

1) Wastage of Space (Some Parts of hash table are never used)
2) If the chain becomes long, then search time can become O(n) in the worst case.
3) Uses extra space for links.

S.NO.	SEPARATE CHAINING	OPEN ADDRESSING
1.	Chaining is Simpler to implement.	Open Addressing requires more computation.
2.	In chaining, Hash table never fills up, we can always add more elements to chain.	In open addressing, table may become full.
3.	Chaining is Less sensitive to the hash function or load factors.	Open addressing requires extra care for to avoid clustering and load factor.
4.	Chaining is mostly used when it is unknown how many and how frequently keys may be inserted or deleted.	Open addressing is used when the frequency and number of keys is known.
5.	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 the same table.
6.	Wastage of Space (Some Parts of hash table in chaining are never used).	In Open addressing, a slot can be used even if an input doesn't map to it.
7.	Chaining uses extra space for links.	No links in Open addressing

Thank you