E- Lecture
Data Structures and Algorithms

By
H R Choudhary (Asstt. Professor)
Department of CSE
Engineering College Ajmer

Topics to be covered
Searching and Sorting

What do you mean by Searching ?

Linear Search

Binary Search

What is Sorting ?

Different Sorting algorithms.

- Bubble Sort
- Insertion Sort
- Selection Sort
- Merge Sort
- Quick Sort

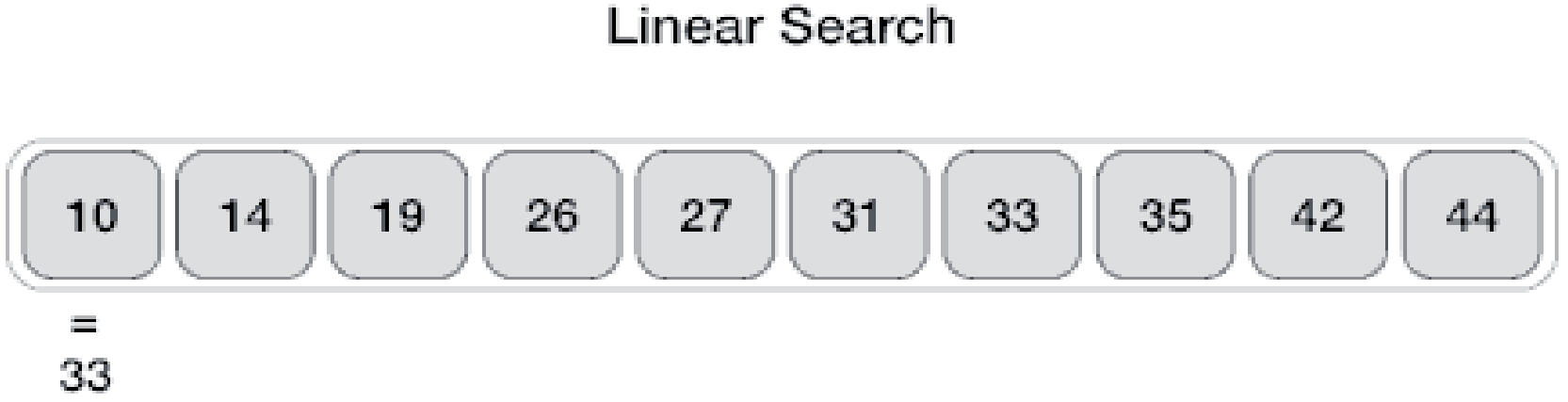
What is searching?

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 searched is found or not. Some of the standard searching technique that is being followed in the data structure is listed below:

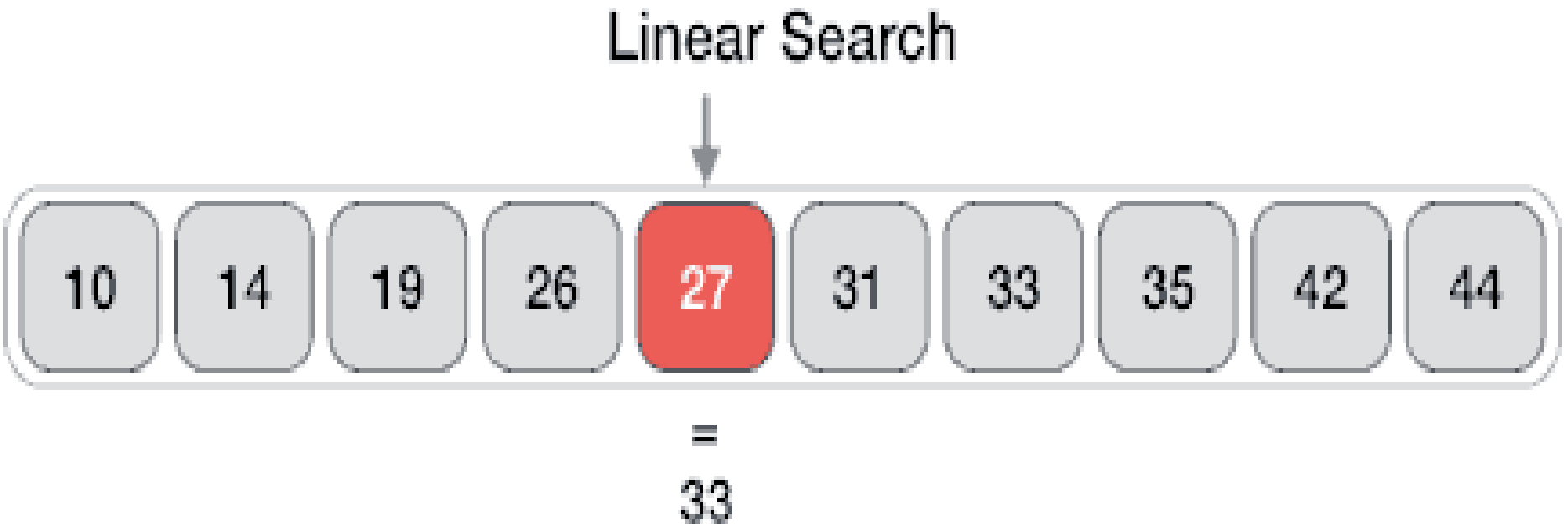
- Linear Search or Sequential Search
- Binary Search

Linear Search

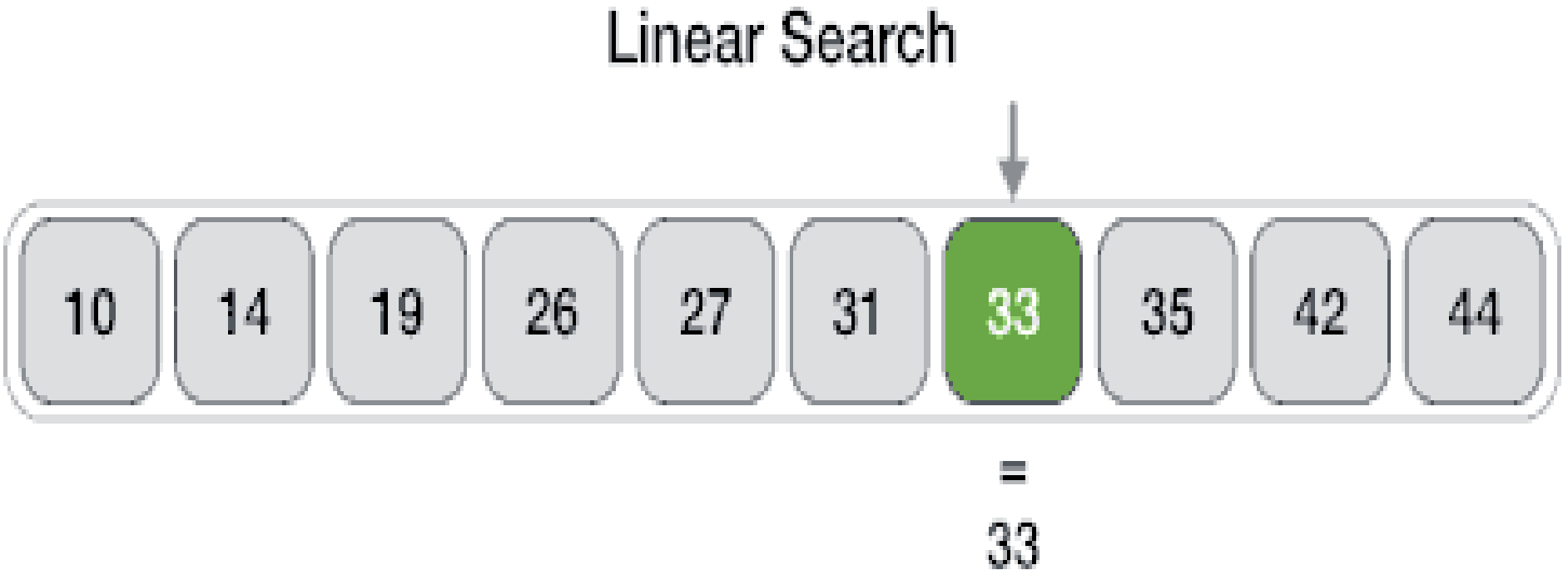
Linear search is a very simple search algorithm. In this type of search, a sequential search is made over all items one by one. Every item is checked and if a match is found then that particular item is returned, otherwise the search continues till the end of the data collection.



Linear Search...



Linaer Search..



Linear Search..

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

Linear Search..

Pseudocode:

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

end procedure
```

Linear Search...

Time Complexity Analysis-

Best case-

In the best possible case,

- The element being searched may be found at the first position.
- In this case, the search terminates in success with just one comparison.
- Thus in best case, linear search algorithm takes O(1) operations.

Linear Search..

Worst Case-

In the worst possible case,

- The element being searched may be present at the last position or not present in the array at all.
- In the former case, the search terminates in success with n comparisons.
- In the later case, the search terminates in failure with n comparisons.
- Thus in worst case, linear search algorithm takes O(n) operations.

Thus, we have-

Time Complexity of Linear Search Algorithm is **O(n).**

Binary Search

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

Binary Search

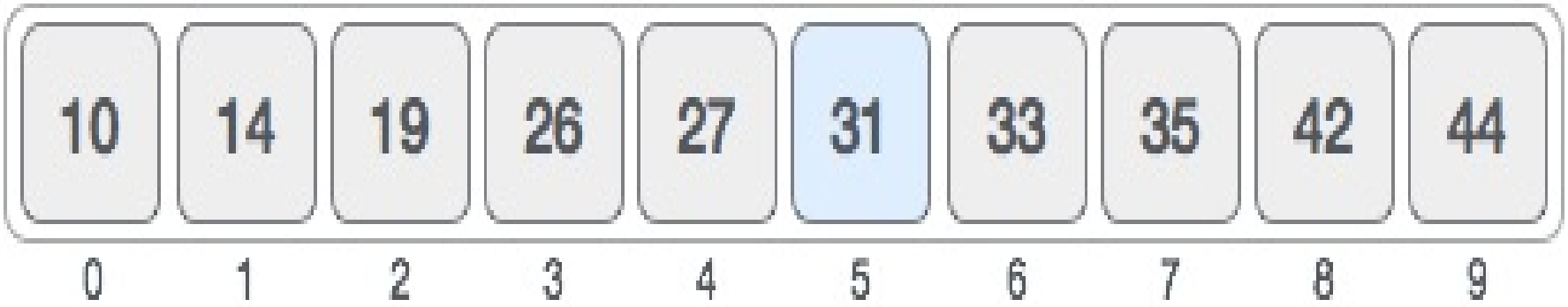
Binary search looks for a particular item by comparing the middle most item of the collection. If a match occurs, then the index of item is returned. If the middle item is greater than the item, then the item is searched in the sub-array to the left of the middle item. Otherwise, the item is searched 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 subarray reduces to zero.

Binary Search...

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.

Binary Search...



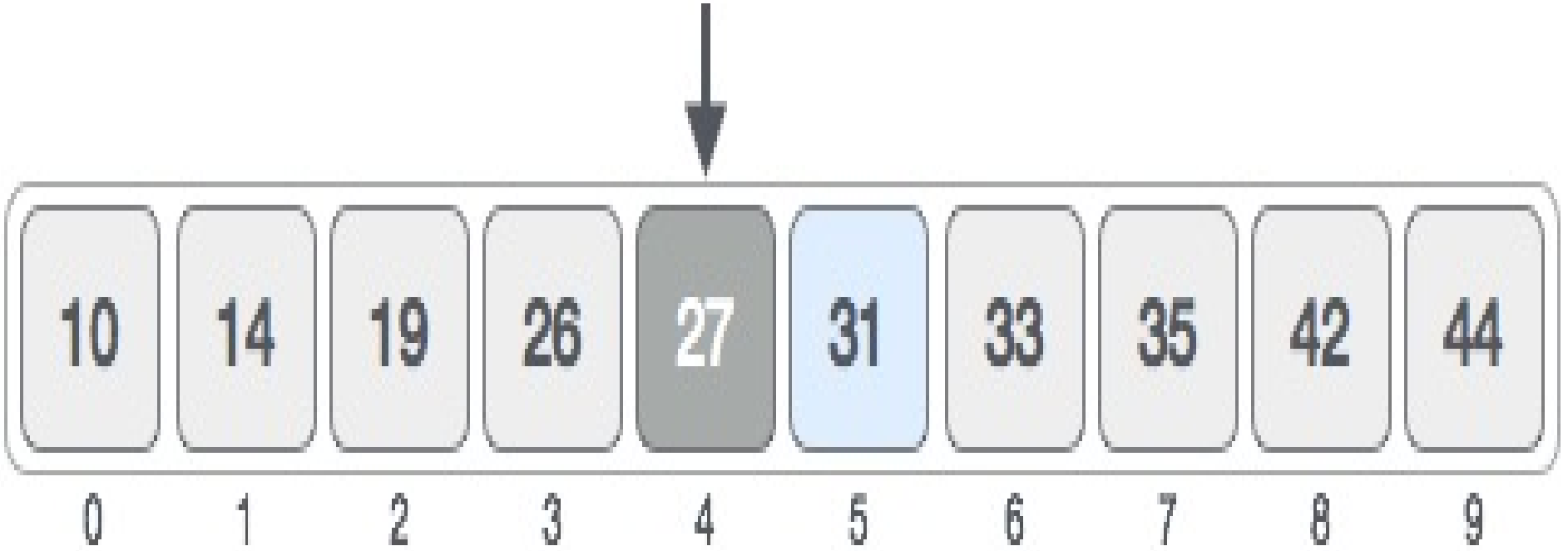
Binary Search...

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.

Binary Search...



Binary Search...

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.

Binary Search...

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

low = mid + 1

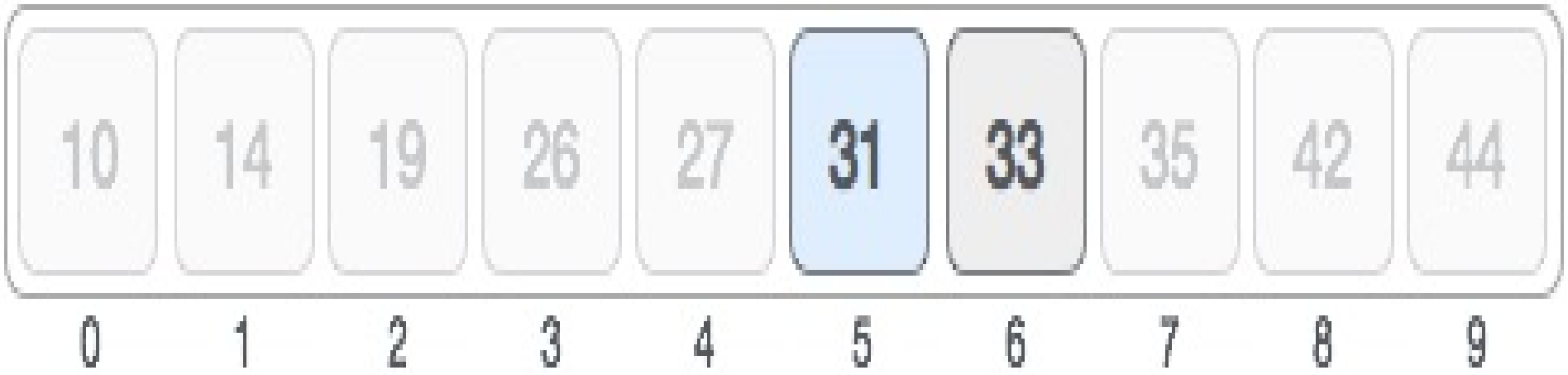
mid = low + (high - low) / 2

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

Binary Search...

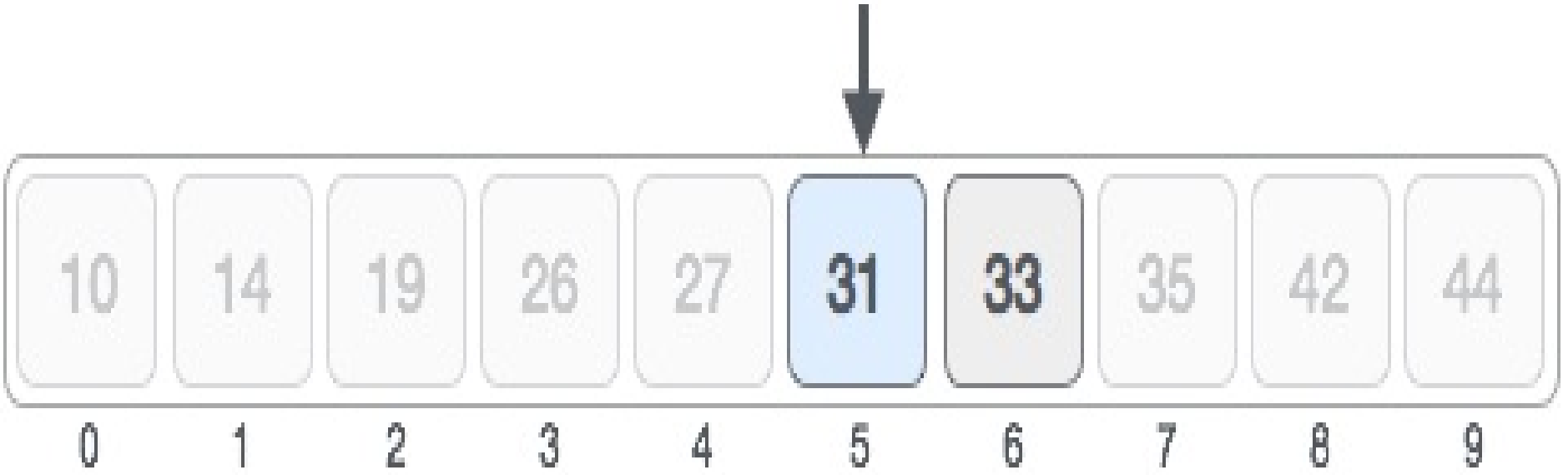
Binary Search...

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.



Binary Search...

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



Binary Search...

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...

Algorithm for binary_search:

A ← sorted array

n ← size of array

x ← value to be searched

Set lowerBound = 1

Set upperBound = n

while x not found

if upperBound < lowerBound

EXIT: x does not exists.

Binary Search...

```
set midPoint = lowerBound + ( upperBound - lowerBound ) / 2


    if A[midPoint] < x
       set lowerBound = midPoint + 1
        if A[midPoint] > x
       set upperBound = midPoint - 1
         if A[midPoint] = x
       EXIT: x found at location midPoint
  end while
  end procedure
```

Thank You

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.

The importance of sorting lies in the fact that data searching can be optimized to a very high level, if data is stored in a sorted manner. Sorting is also used to represent data in more readable formats. Following are some of the examples of sorting in real-life scenarios −

Telephone Directory − The telephone directory stores the telephone numbers of people sorted by their names, so that the names can be searched easily.

Dictionary − The dictionary stores words in an alphabetical order so that searching of any word becomes easy.

Sorting...

Different Types of Sorting:

On the basis of Comparison and Non comparison:

A comparison sort is a type of sorting algorithm that only reads the list elements through a single abstract comparison operation (often a "less than or equal to" operator or a three-way comparison) that determines which of two elements should occur first in the final sorted list

Some of the most well-known comparison sorts include: Quicksort, Heapsort,Merge sort, Introsort, Insertion sort, Selection sort

Bubble sort.

Sorting...

Non-comparison based sorting algorithms:

No comparison sorting includes Counting sort which sorts using key-value, Radix sort, which examines individual bits of keys, and Bucket Sort which examines bits of keys. ... On the other hand, non-comparison based sorting algorithms don't use comparison but rely on integer arithmetic on keys.

Sorting...

In-place Sorting and Not-in-place Sorting:

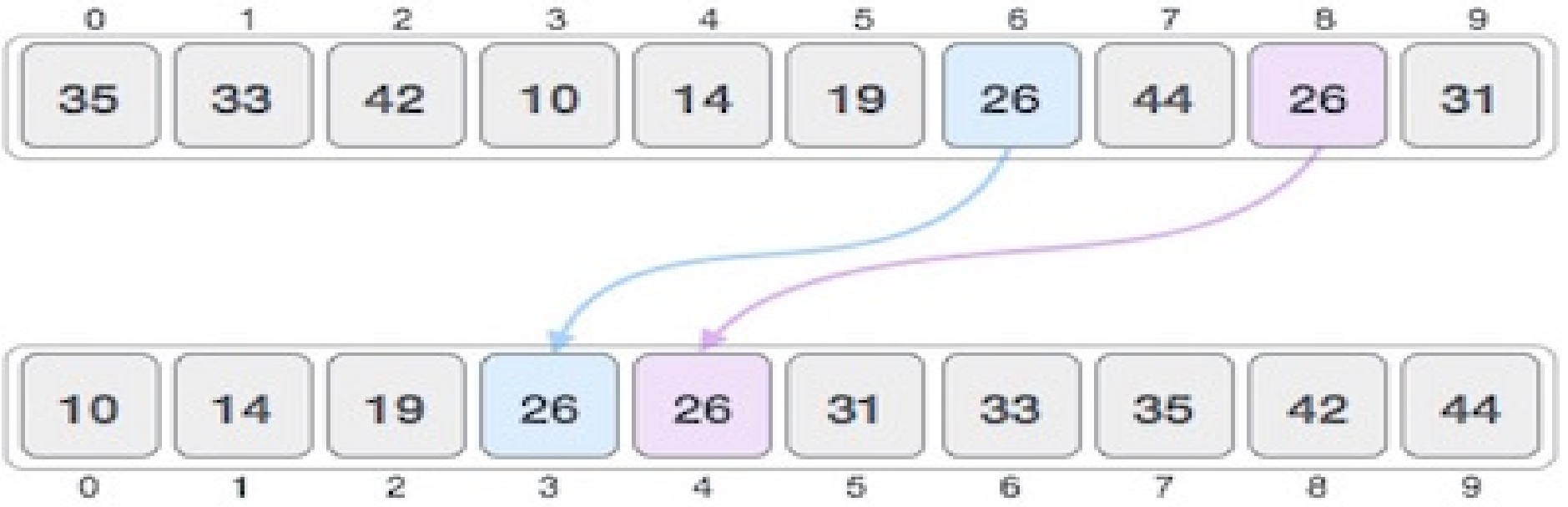
Sorting algorithms may require some extra space for comparison and temporary storage of few data elements. These algorithms do not require any extra space and sorting is said to happen in-place, or for example, within the array itself. This is called in-place sorting. Bubble sort is an example of in-place sorting.

However, in some sorting algorithms, the program requires space which is more than or equal to the elements being sorted. Sorting which uses equal or more space is called not-in-place sorting. Merge-sort is an example of not-in-place sorting.

Sorting...

Stable and Not Stable Sorting:

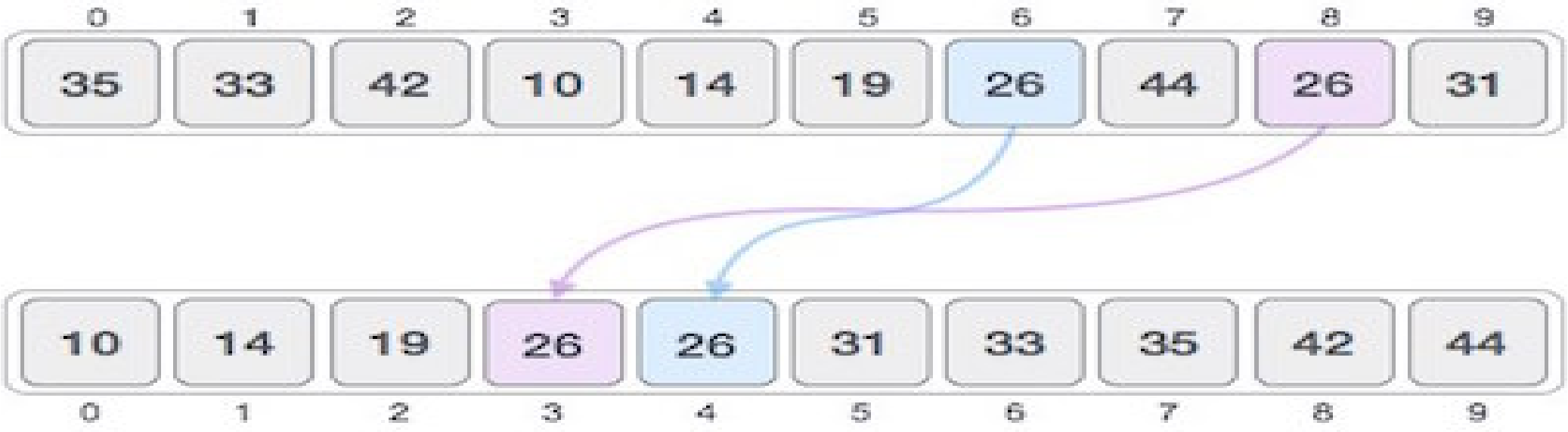
If a sorting algorithm, after sorting the contents, does not change the sequence of similar content in which they appear, it is called stable sorting.



Sorting...

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

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



Sorting...

Adaptive and Non-Adaptive Sorting Algorithm:

A sorting algorithm is said to be adaptive, if it takes advantage of already 'sorted' elements in the list that is to be sorted. That is, while sorting if the source list has some element already sorted, adaptive algorithms will take this into account and will try not to re-order them.

A non-adaptive algorithm is one which does not take into account the elements which are already sorted. They try to force every single element to be re-ordered to confirm their sortedness.

Sorting...

Internal and External Sorting:

In internal sorting all the data to sort is stored in memory at all times while sorting is in progress. In external sorting data is stored outside memory (like on disk) and only loaded into memory in small chunks. External sorting is usually applied in cases when data can't fit into memory entirely.

Insertion sort, quick sort, heap sort, radix sort can be used for internal sorting.

Sorting...

Sorting Terms:

Some terms are generally coined while discussing sorting techniques, here is a brief introduction to them −

Increasing Order:

A sequence of values is said to be in increasing order, if the successive element is greater than the previous one. For example, 1, 3, 4, 6, 8, 9 are in increasing order, as every next element is greater than the previous element.

Decreasing Order:

A sequence of values is said to be in decreasing order, if the successive element is less than the current one. For example, 9, 8, 6, 4, 3, 1 are in decreasing order, as every next element is less than the previous element.

Sorting...

Non-Increasing Order:

A sequence of values is said to be in non-increasing order, if the successive element is less than or equal to its previous element in the sequence. This order occurs when the sequence contains duplicate values. For example, 9, 8, 6, 3, 3, 1 are in non-increasing order, as every next element is less than or equal to (in case of 3) but not greater than any previous element.

Non-Decreasing Order:

A sequence of values is said to be in non-decreasing order, if the successive element is greater than or equal to its previous element in the sequence. This order occurs when the sequence contains duplicate values. For example, 1, 3, 3, 6, 8, 9 are in non-decreasing order, as every next element is greater than or equal to (in case of 3) but not less than the previous one.

Thank You

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.

Bubble Sort Working:

We take an unsorted array for our example. Bubble sort takes O(n2) time so we're keeping it short and precise.

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 −

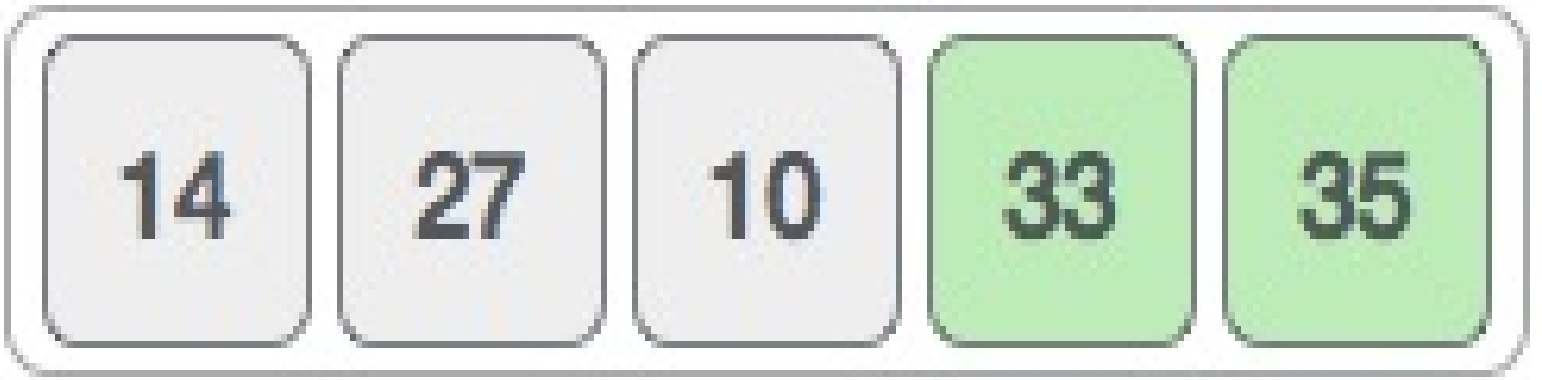
Next we compare 33 and 35. We find that both are in already sorted positions.

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.

Algorithm:

We assume list is an array of n elements. We further assume that swap function swaps the values of the given array elements.

```
begin BubbleSort(list)
    for all elements of list
      if list[i] > list[i+1]
         swap(list[i], list[i+1])
      end if
   end for
    return list
   end BubbleSort
```

We observe in algorithm that Bubble Sort compares each pair of array element unless the whole array is completely sorted in an ascending order. This may cause a few complexity issues like what if the array needs no more swapping as all the elements are already ascending.

To ease-out the issue, we use one flag variable swapped which will help us see if any swap has happened or not. If no swap has occurred, i.e. the array requires no more processing to be sorted, it will come out of the loop.

```
procedure bubbleSort( list : array of items )

   loop = list.count;
      for i = 0 to loop-1 do:
      swapped = false
        for j = 0 to loop-1 do:

         /* compare the adjacent elements */
         if list[j] > list[j+1] then
            /* swap them */
            swap( list[j], list[j+1] )
            swapped = true
         end if
         end for
```

```
/*if no number was swapped that means
      array is sorted now, break the loop.*/

      if(not swapped) then
         break
      end if

   end for

end procedure return list
```

Thank You

Insertion Sort

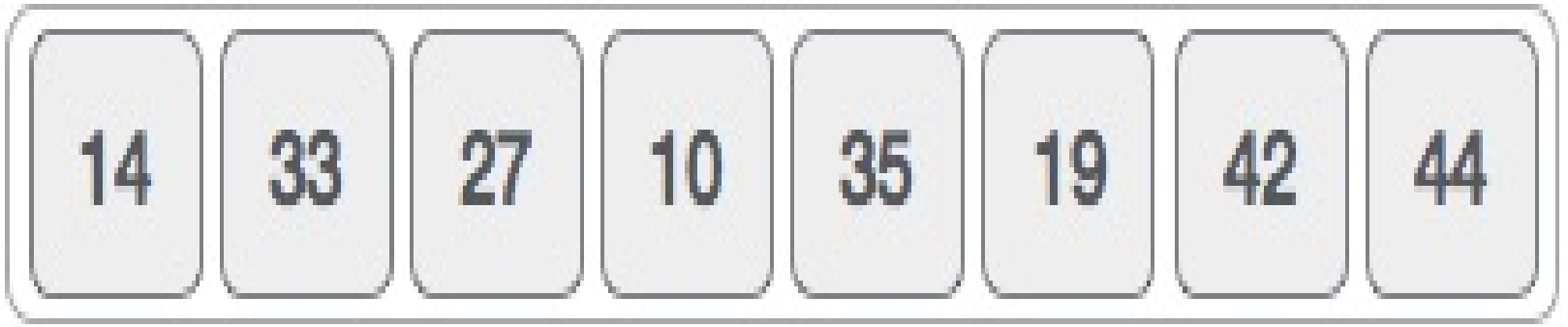
This is an in-place comparison-based sorting algorithm. Here, a sub-list is maintained which is always sorted. For example, the lower part of an array is maintained to be sorted. An element which is to be 'insert'ed in this sorted sub-list, has to find its appropriate place and then it has to be inserted 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.

Insertion Sort...

Insertion Sort Working:

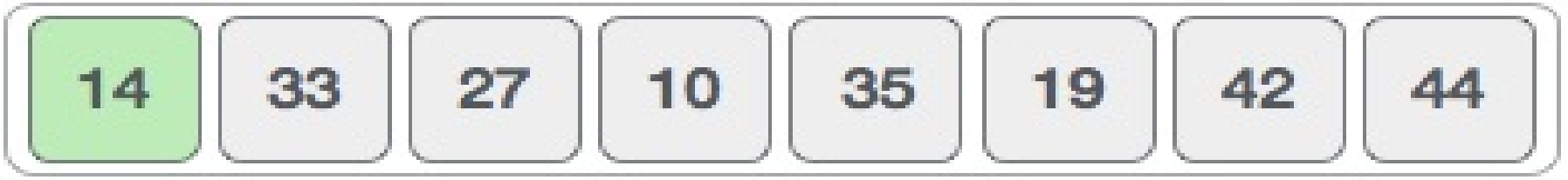
We take an unsorted array for our example.



Insertion Sort...

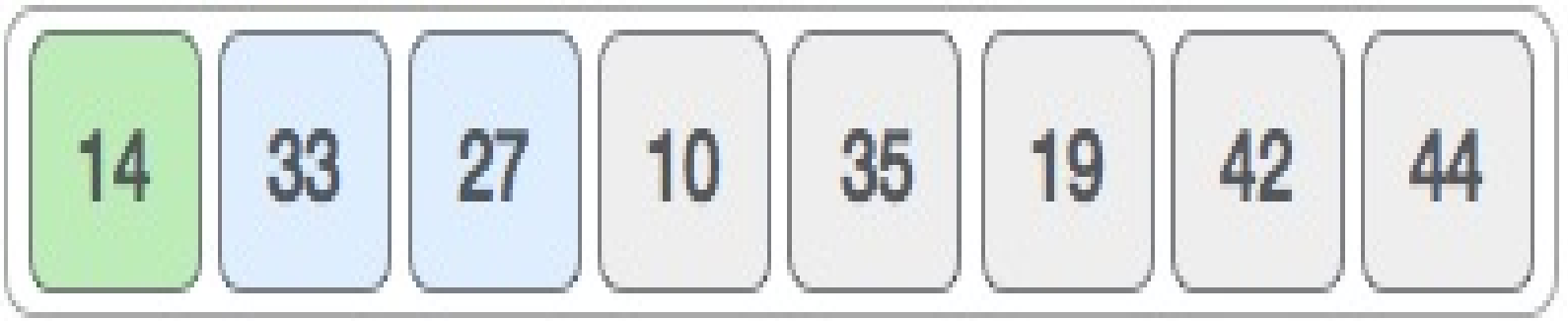
Insertion sort compares the first two elements.

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



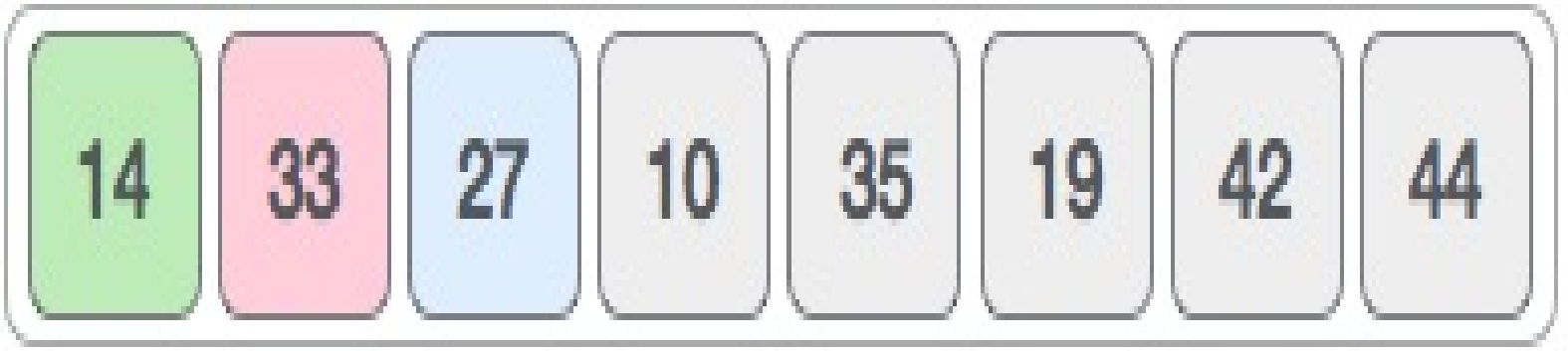
Insertion Sort...

Insertion sort moves ahead and compares 33 with 27.



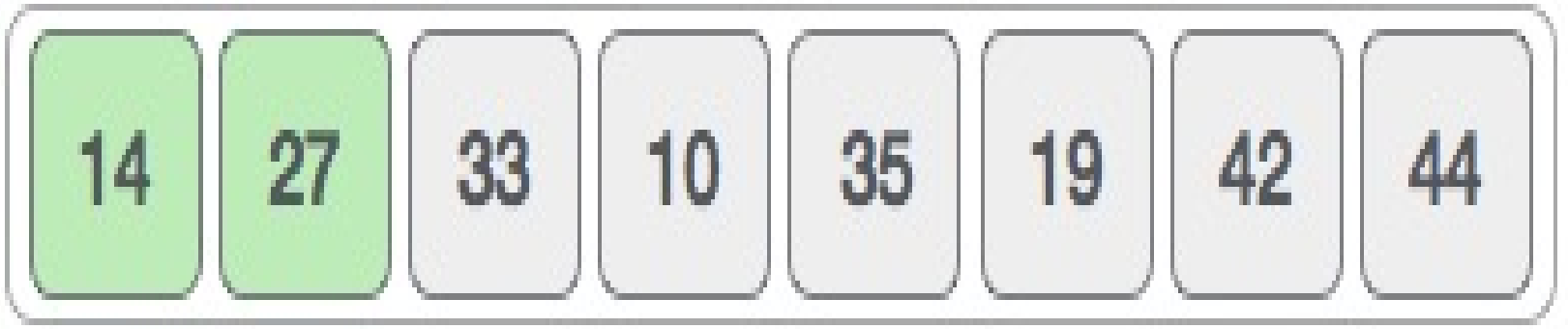
Insertion Sort...

And finds that 33 is not in the correct position.



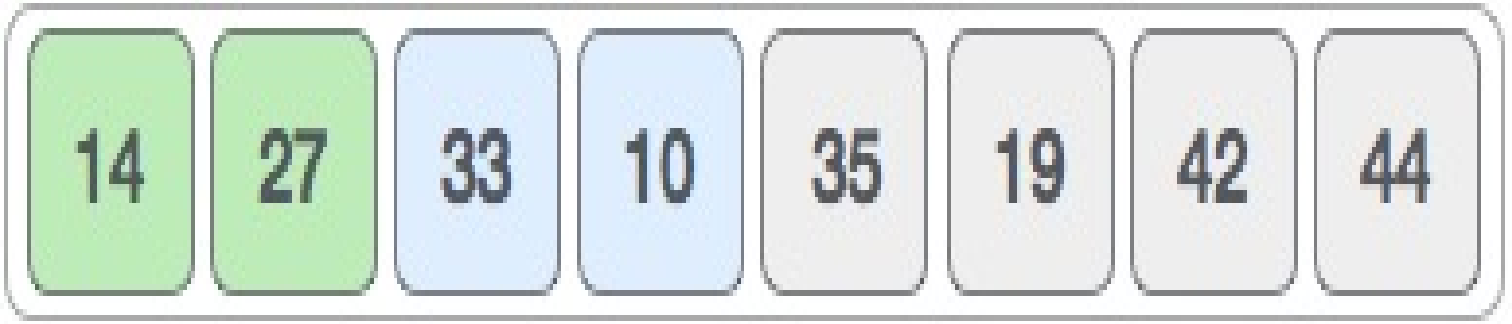
Insertion Sort...

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.



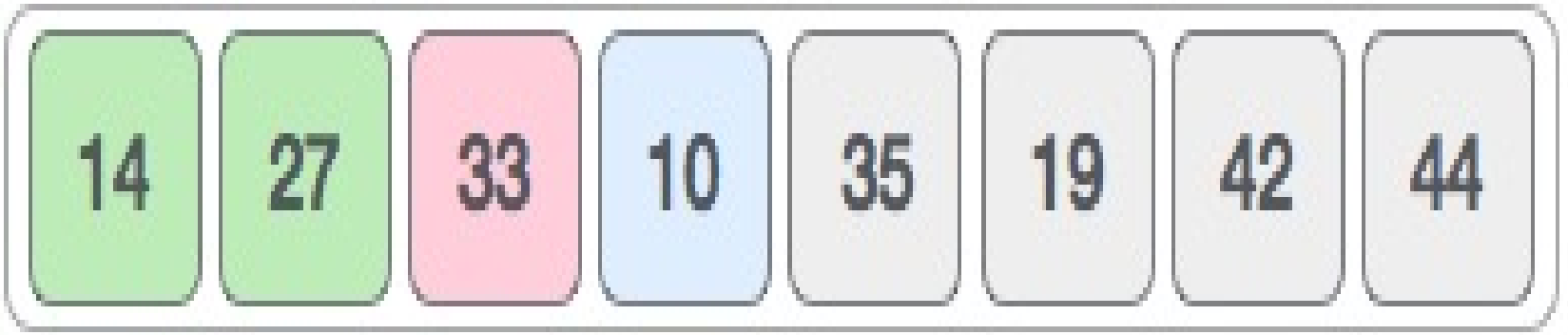
Insertion Sort...

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



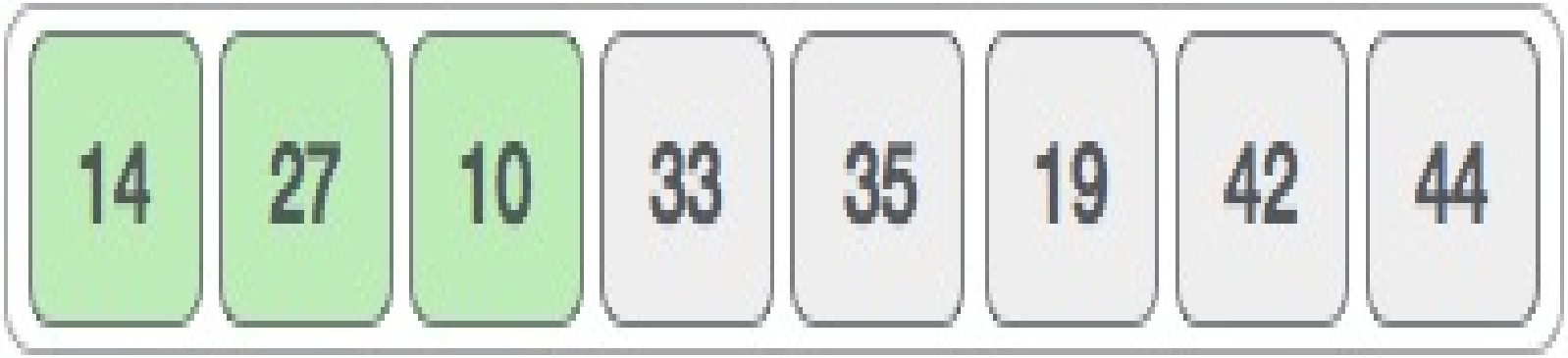
Insertion Sort...

These values are not in a sorted order.



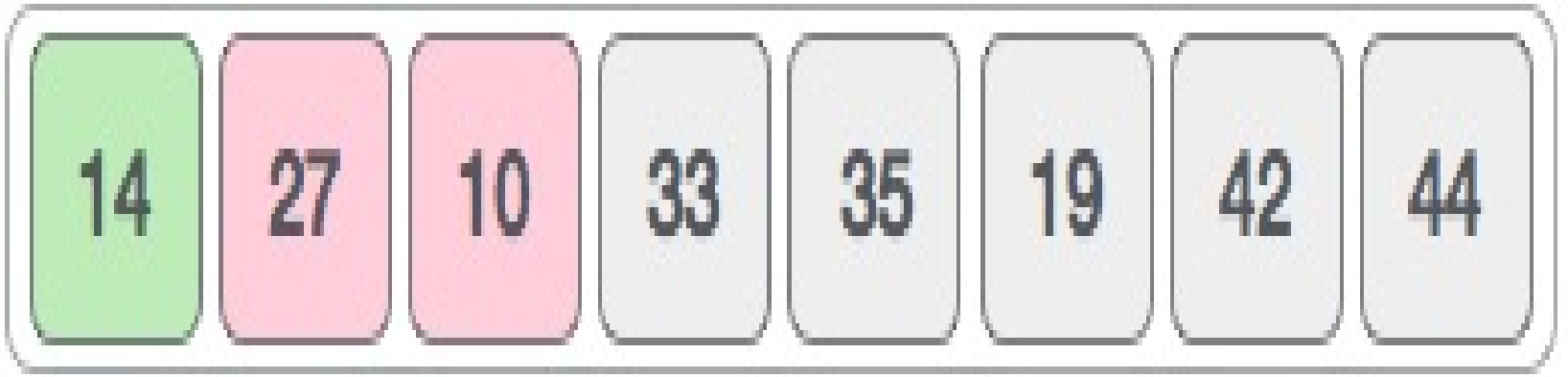
Insertion Sort...

So we swap them.



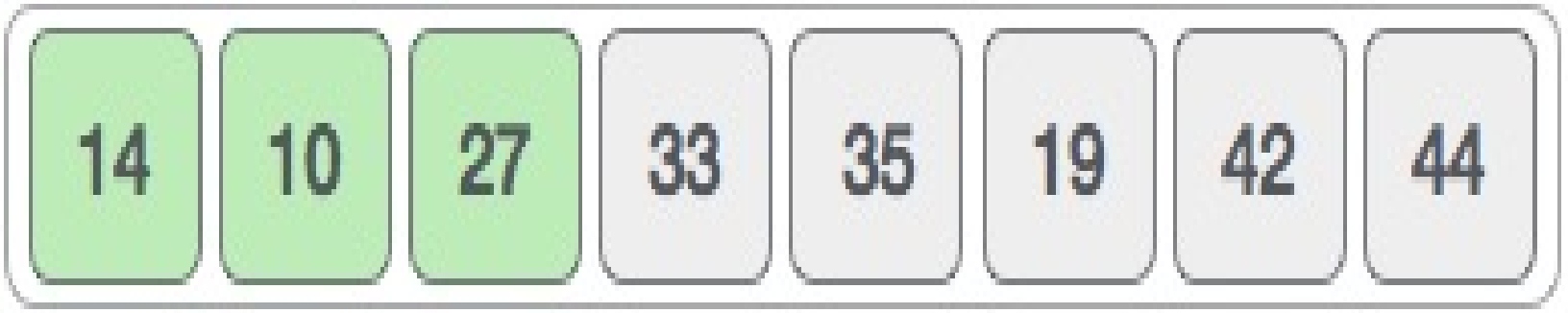
Insertion Sort...

However, swapping makes 27 and 10 unsorted.



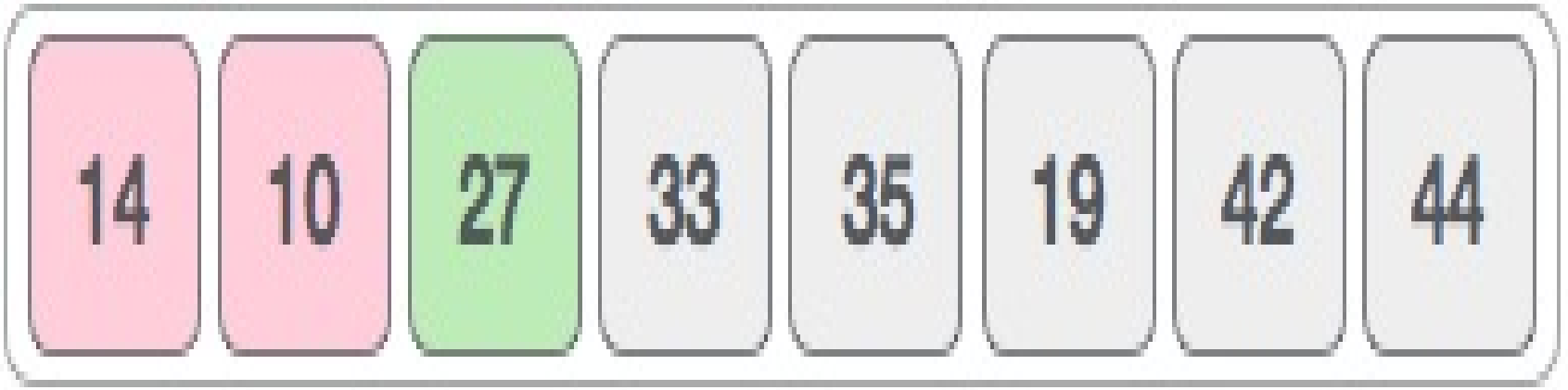
Insertion Sort...

Hence, we swap them too.



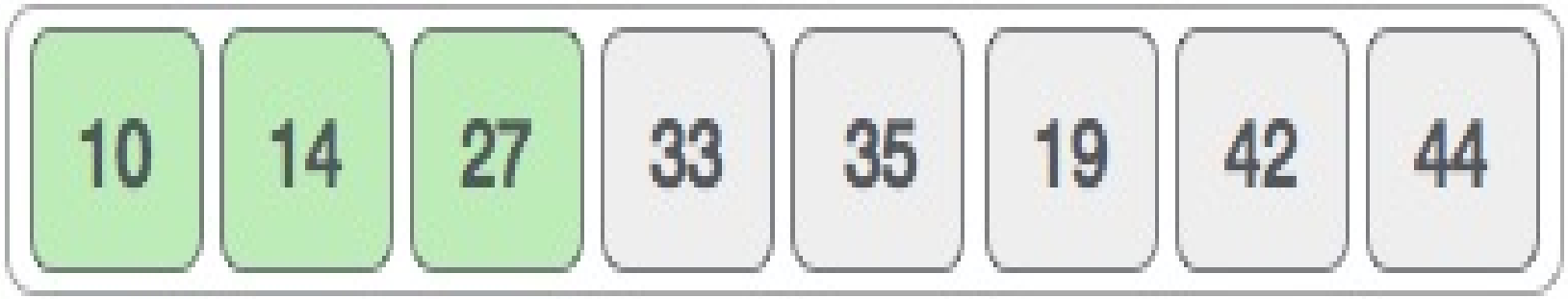
Insertion Sort...

Again we find 14 and 10 in an unsorted order.



Insertion Sort...

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



This process goes on until all the unsorted values are covered in a sorted sub-list.

Insertion Sort...

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.

Step 5 − Insert the value

Step 6 − Repeat until list is sorted

Insertion Sort...

```
procedure insertionSort( A : array of items )
   int Position
   int valueToInsert

   for i = 1 to length(A) inclusive do:

      /* select value to be inserted */
      valueToInsert = A[i]
      Position = i
```

Insertion Sort...

```
/*locate  position for the element to be inserted */

    while Position > 0 and A[Position-1] > valueToInsert do:
       A[Position] = A[Position-1]
       Position = Position -1
    end while

    /* insert the number at hole position */
    A[Position] = valueToInsert

  end for

end procedure
```

Thank You

Selection Sort

Selection sort is a simple sorting algorithm. This sorting algorithm is an in-place comparison-based algorithm in which the list is divided into two parts, the sorted part at the left end and the unsorted part at the right end. Initially, the sorted part is empty and the unsorted part is the entire list.

The smallest element is selected from the unsorted array and swapped with the leftmost element, and that element becomes a part of the sorted array. This process continues moving unsorted array boundary by one element to the right.

This algorithm is not suitable for large data sets as its average and worst case complexities are of O(n2), where n is the number of items.

Selection Sort...

Selection Sort Working:

Consider the following depicted array as an example.

Selection Sort...

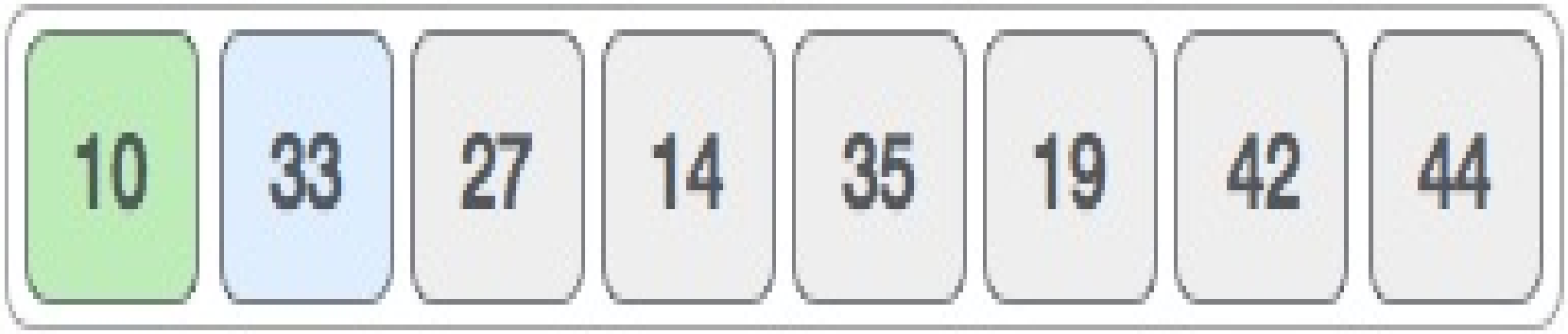
For the first position in the sorted list, the whole list is scanned sequentially. The first position where 14 is stored presently, we search the whole list and find that 10 is the lowest value.

Selection Sort...

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.

Selection Sort...

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



Selection Sort...

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

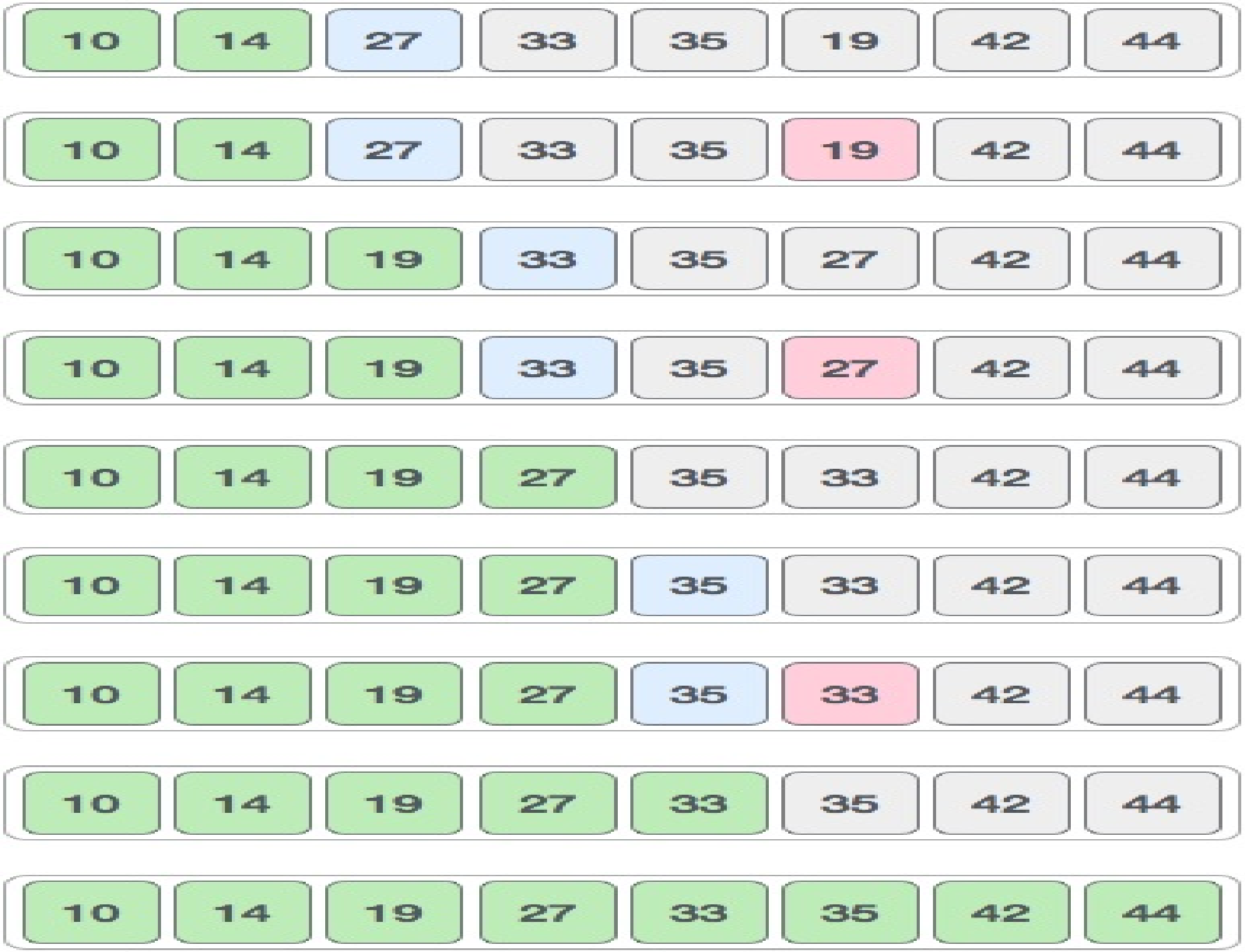
Selection Sort...

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

Selection Sort...

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

Following is a pictorial depiction of the entire sorting process −


10 14 27 33 35 19 42 44
10 14 27 33 35 19 42 44
10 14 19 33 35 27 42 44
10 14 19 33 35 27 42 44
10 14 19 27 35 33 42 44
10 14 19 27 35 33 42 44
10 14 19 27 35 33 42 44
10 14 19 27 33 35 42 44
10 14 19 27 33 35 42 44

Selection Sort...

Algorithm:

Step 1 − Set MIN to location 0

Step 2 − Search the minimum element in the list

Step 3 − Swap with value at location MIN

Step 4 − Increment MIN to point to next element

Step 5 − Repeat until list is sorted

```
procedure selection sort
   list  : array of items
   n     : size of list


   for i = 1 to n - 1
   /* set current element as minimum*/
      min = i
  /* check the element to be minimum */
         for j = i+1 to n
        if list[j] < list[min] then
           min = j;
        end if
      end for
```

Selection Sort...

```
/* swap the minimum element with the current element*/
      if indexMin != i  then
         swap list[min] and list[i]
      end if
   end for

end procedure
```

Thank You

Merge sort

Merge sort is a sorting technique based on divide and conquer technique. With worst-case time complexity being O(n log n), it is one of the most respected algorithms.

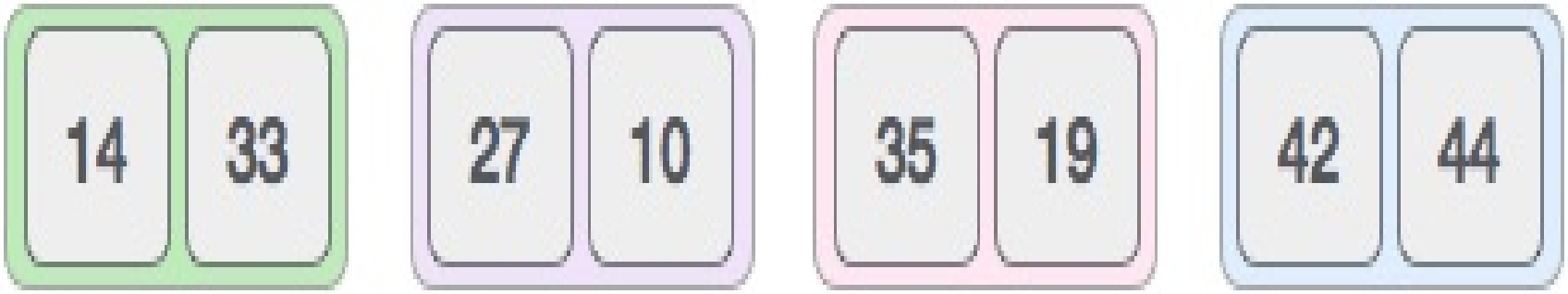
Merge sort first divides the array into equal halves and then combines them in a sorted manner.

How Merge Sort Works?

To understand merge sort, we take an unsorted array as the following −

We know that merge sort first divides the whole array iteratively into equal halves unless the atomic values are achieved. We see here that an array of 8 items is divided into two arrays of size 4.

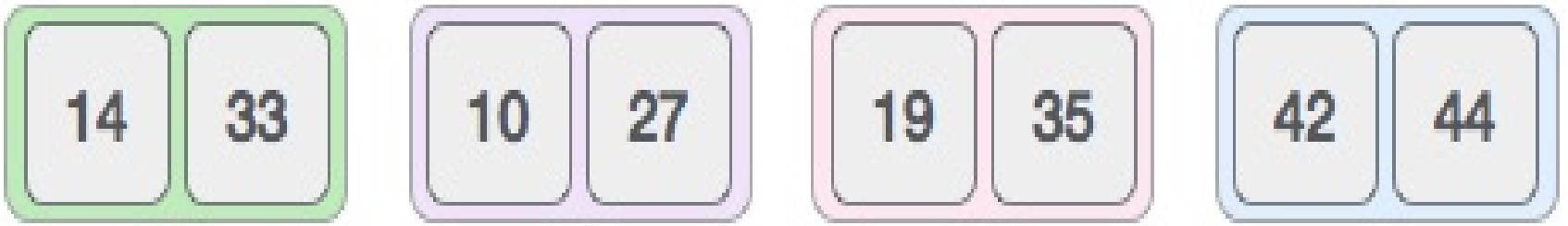
This does not change the sequence of appearance of items in the original. Now we divide these two arrays into halves.



We further divide these arrays and we achieve atomic value which can no more be divided.

Now, we combine them in exactly the same manner as they were broken down. Please note the color codes given to these lists.

We first compare the element for each list and then combine them into another list in a sorted manner. We see that 14 and 33 are in sorted positions. We compare 27 and 10 and in the target list of 2 values we put 10 first, followed by 27. We change the order of 19 and 35 whereas 42 and 44 are placed sequentially.


14
33
10
27
19
35
42
44

In the next iteration of the combining phase, we compare lists of two data values, and merge them into a list of found data values placing all in a sorted order.

After the final merging, the list should look like this −

Algorithm:

Merge sort keeps on dividing the list into equal halves until it can no more be divided. By definition, if it is only one element in the list, it is sorted. Then, merge sort combines the smaller sorted lists keeping the new list sorted too.

Step 1 − if it is only one element in the list it is already sorted, return.

Step 2 − divide the list recursively into two halves until it can no more be divided.

Step 3 − merge the smaller lists into new list in sorted order.

```
procedure mergesort( var a as array )
   if ( n == 1 ) return a

   var l1 as array = a[0] ... a[n/2]
   var l2 as array = a[n/2+1] ... a[n]

   l1 = mergesort( l1 )
   l2 = mergesort( l2 )

   return merge( l1, l2 )
end procedure
```

```
procedure merge( var a as array, var b as array )

   var c as array
   while ( a and b have elements )
      if ( a[0] > b[0] )
         add b[0] to the end of c
         remove b[0] from b
      else
         add a[0] to the end of c
         remove a[0] from a
      end if
   end while
```

```
while ( a has elements )
      add a[0] to the end of c
      remove a[0] from a
   end while

   while ( b has elements )
      add b[0] to the end of c
      remove b[0] from b
   end while

   return c

end procedure
```

Quick Sort Algorithm

Quicksort is the widely used sorting algorithm that makes n log n comparisons in average case for sorting an array of n elements. It is a faster and highly efficient sorting algorithm. This algorithm follows the divide and conquer approach. Divide and conquer is a technique of breaking down the algorithms into subproblems, then solving the subproblems, and combining the results back together to solve the original problem.

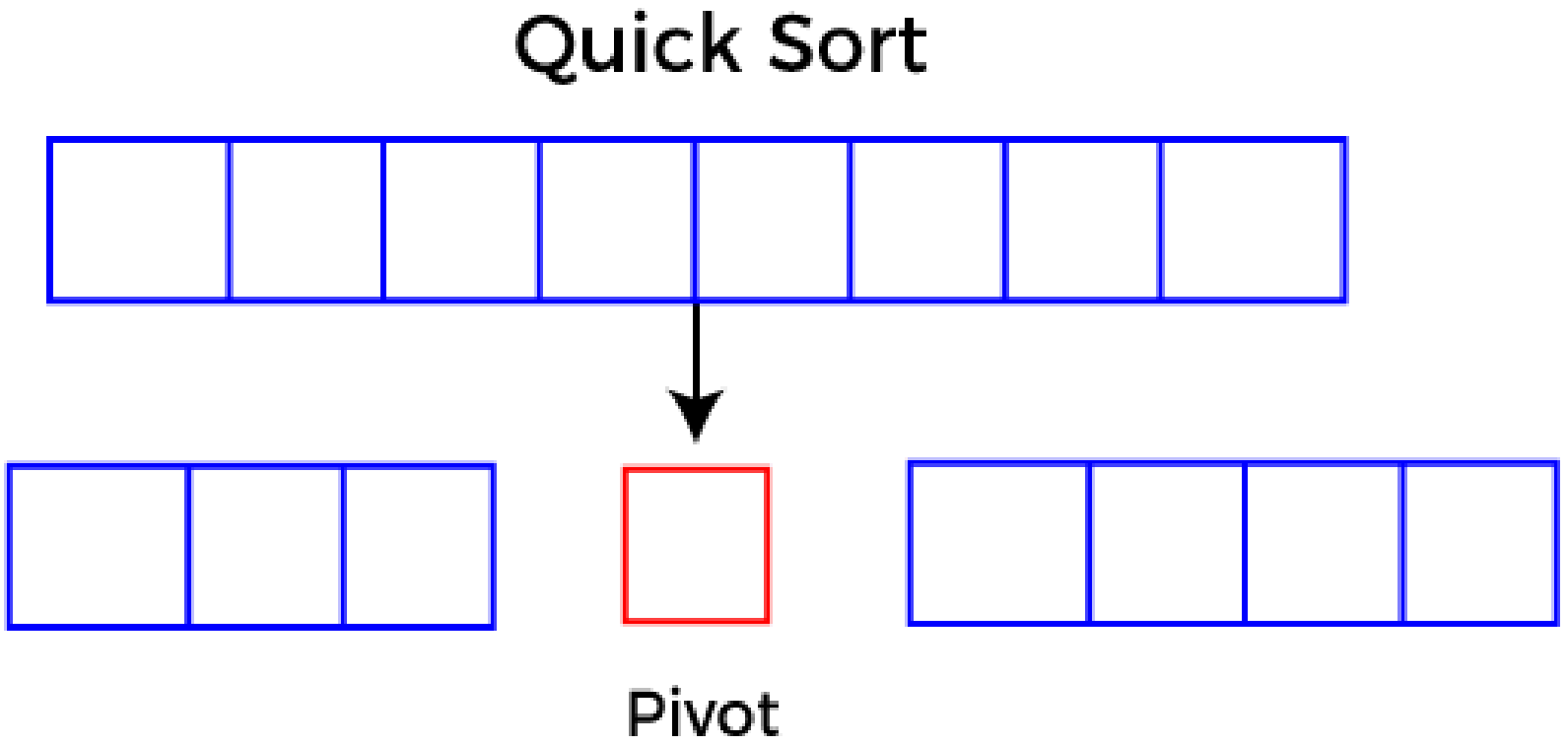
Divide: In Divide, first pick a pivot element. After that, partition or rearrange the array into two sub-arrays such that each element in the left sub-array is less than or equal to the pivot element and each element in the right sub-array is larger than the pivot element.

Conquer: Recursively, sort two subarrays with Quicksort.

Combine: Combine the already sorted array.

Quicksort picks an element as pivot, and then it partitions the given array around the picked pivot element. In quick sort, a large array is divided into two arrays in which one holds values that are smaller than the specified value (Pivot), and another array holds the values that are greater than the pivot.

After that, left and right sub-arrays are also partitioned using the same approach. It will continue until the single element remains in the sub-array.


Quick Sort
Pivot

Choosing the pivot

Picking a good pivot is necessary for the fast implementation of quicksort. However, it is typical to determine a good pivot. Some of the ways of choosing a pivot are as follows -

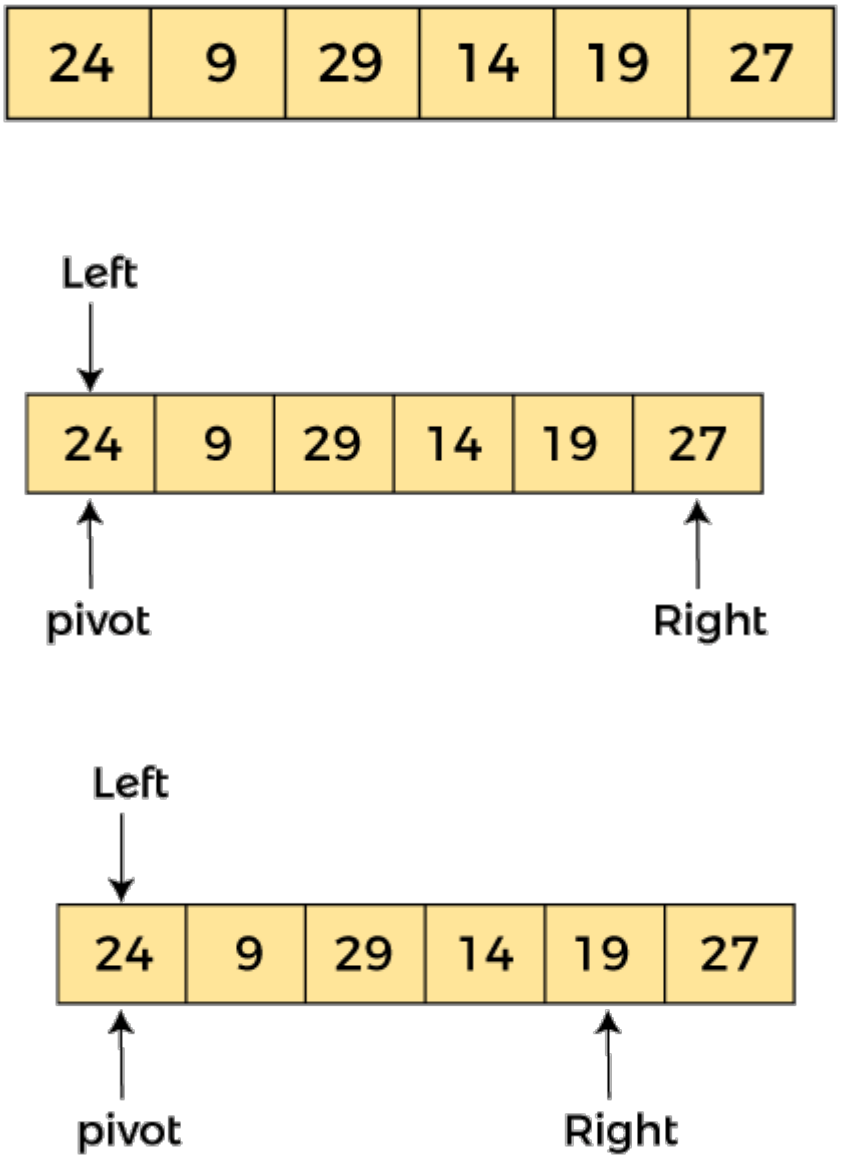
Pivot can be random, i.e. select the random pivot from the given array.

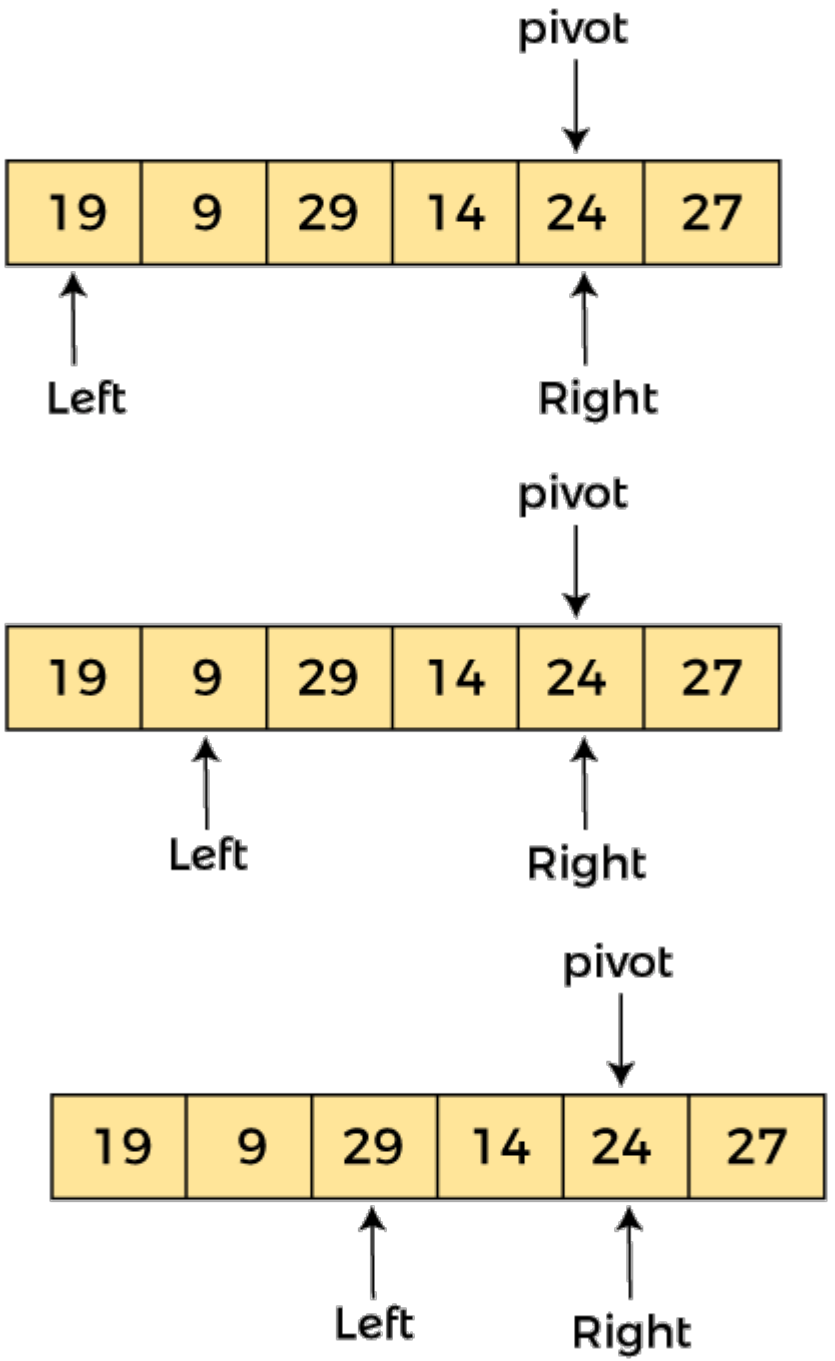
Pivot can either be the rightmost element of the leftmost element of the given array.

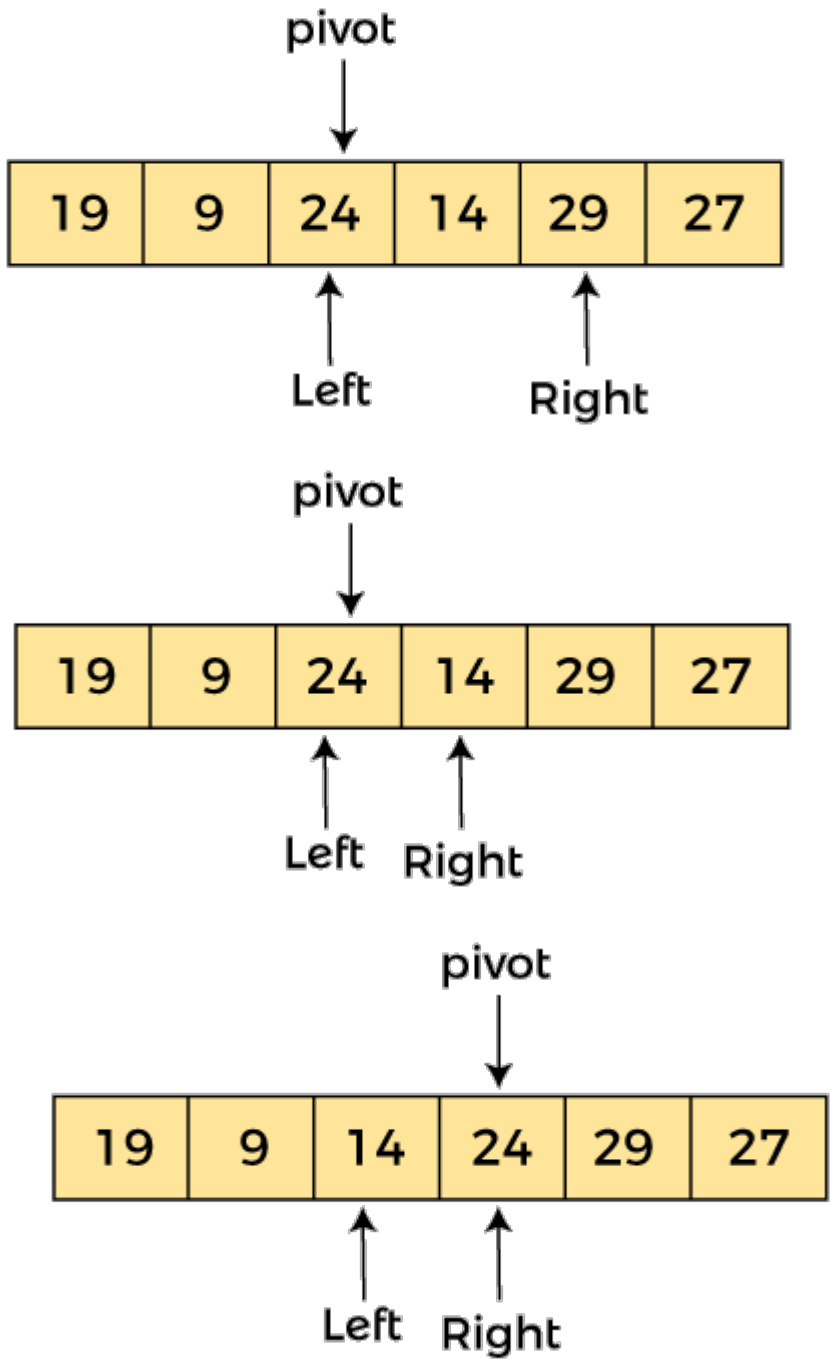
Select median as the pivot element.

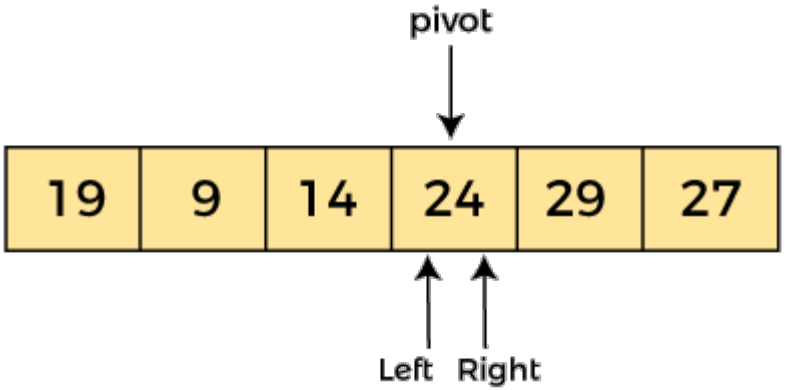
```
QUICKSORT (array A, start, end)
{
 1 if (start < end)
 2 {
3 p = partition(A, start, end)
4 QUICKSORT (A, start, p - 1)
5 QUICKSORT (A, p + 1, end)
6 }
}
```

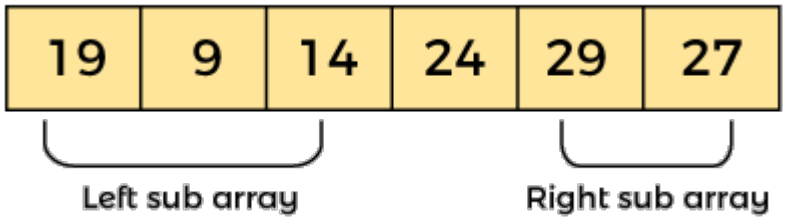
```
PARTITION (array A, start, end)
{
 1 pivot ? A[end]
 2 i ? start-1
 3 for j ? start to end -1 {
 4 do if (A[j] < pivot) {
 5 then i ? i + 1
 6 swap A[i] with A[j]
 7  }}
 8 swap A[i+1] with A[end]
 9 return i+1
}
```

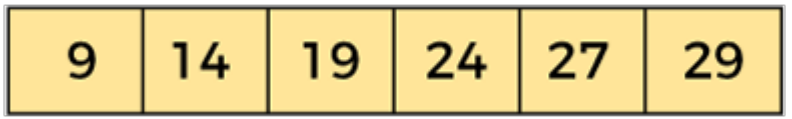

24
9
29
14
19
27
Left
pivot
Right
Left
pivot
Right


pivot
19
9
29
14
24
27
Left
Right
pivot
19
9
29
14
24
27
Left
Right
pivot
19
9
29
14
24
27
Left
Right


pivot
19
9
24
14
29
27
Left
Right
pivot
19
9
24
14
29
27
Left
Right
pivot
19
9
14
24
29
27
Left
Right


pivot
19
9
14
24
29
27
Left
Right


19
9
14
24
29
27
Left sub array
Right sub array


9
14
19
24
27
29

Heap Sort Algorithm

Heap sort processes the elements by creating the min-heap or max-heap using the elements of the given array. Min-heap or max-heap represents the ordering of array in which the root element represents the minimum or maximum element of the array.

Heap sort basically recursively performs two main operations -

Build a heap H, using the elements of array.

Repeatedly delete the root element of the heap formed in 1st phase.

What is a heap?

A heap is a complete binary tree, and the binary tree is a tree in which the node can have the utmost two children. A complete binary tree is a binary tree in which all the levels except the last level, i.e., leaf node, should be completely filled, and all the nodes should be left-justified.

What is heap sort?

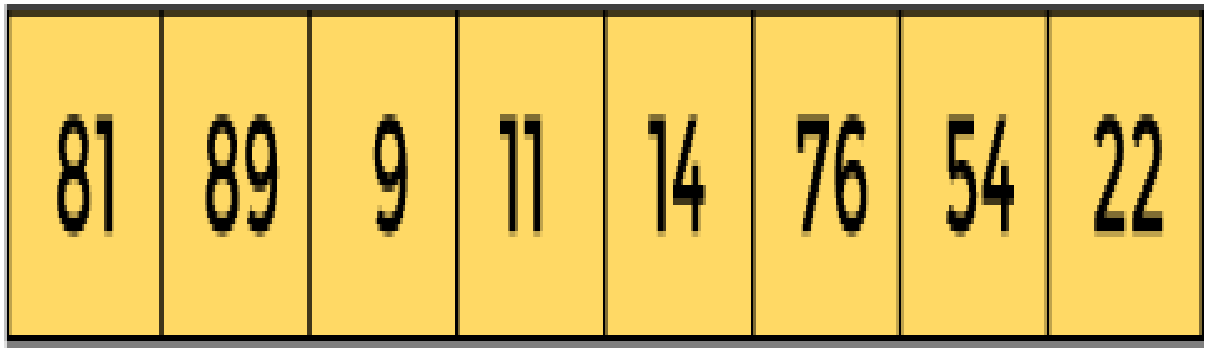
Heapsort is a popular and efficient sorting algorithm. The concept of heap sort is to eliminate the elements one by one from the heap part of the list, and then insert them into the sorted part of the list.

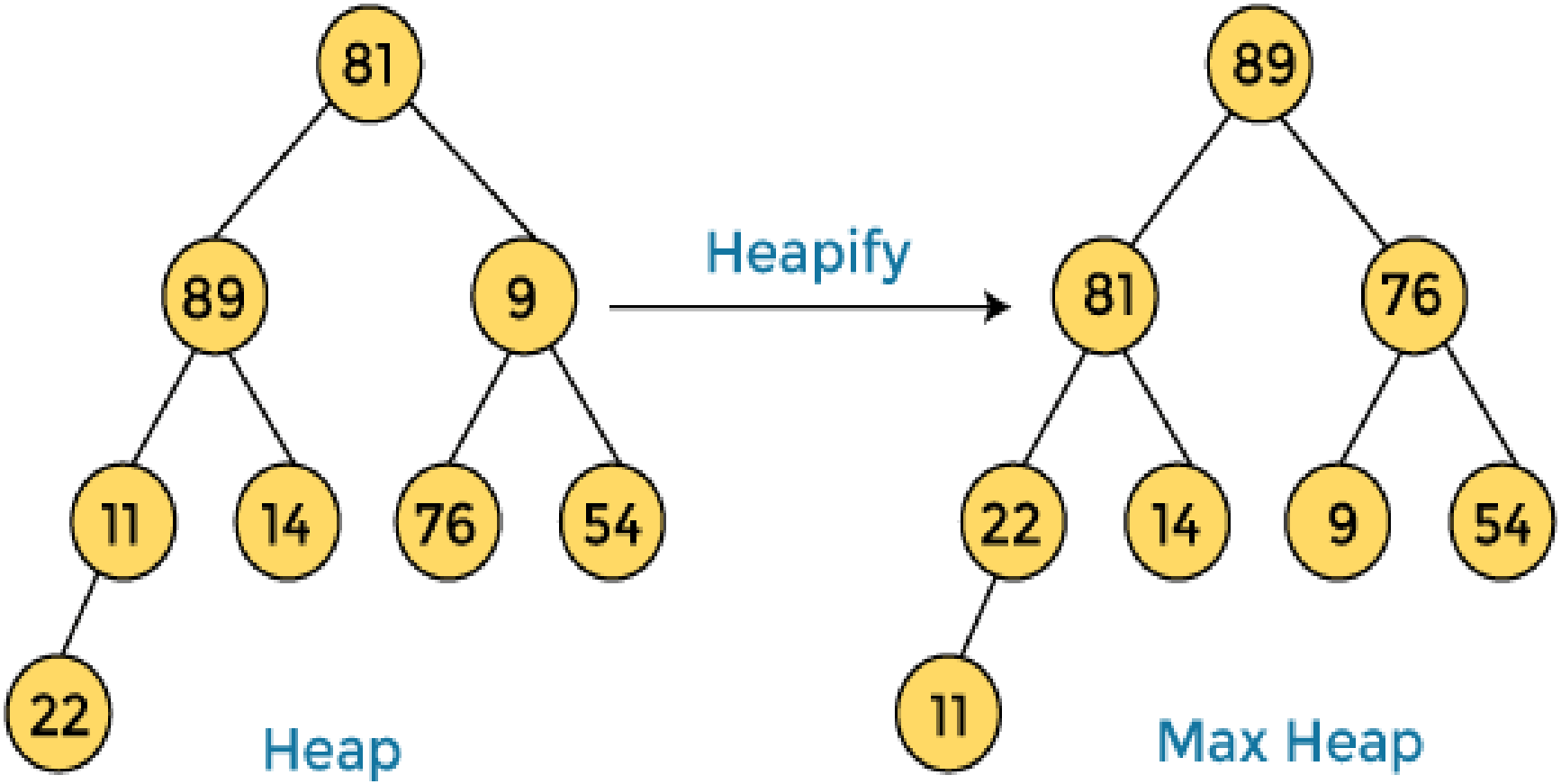
Heapsort is the in-place sorting algorithm.

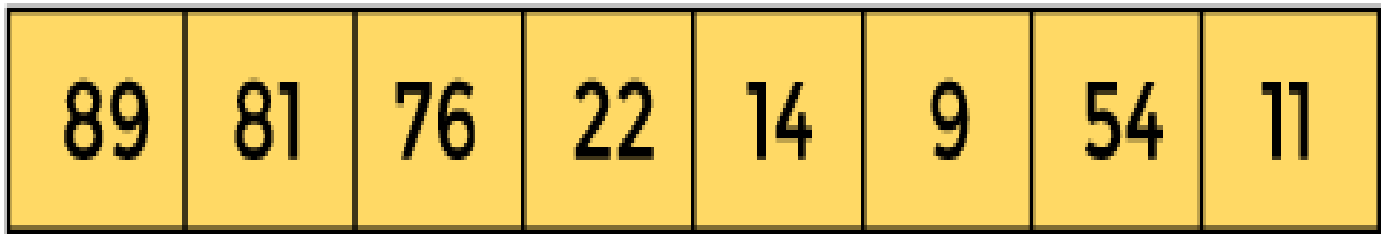
```
HeapSort(arr)
BuildMaxHeap(arr)
for i = length(arr) to 2
    swap arr[1] with arr[i]
        heap_size[arr] = heap_size[arr] ? 1
        MaxHeapify(arr,1)
End
```

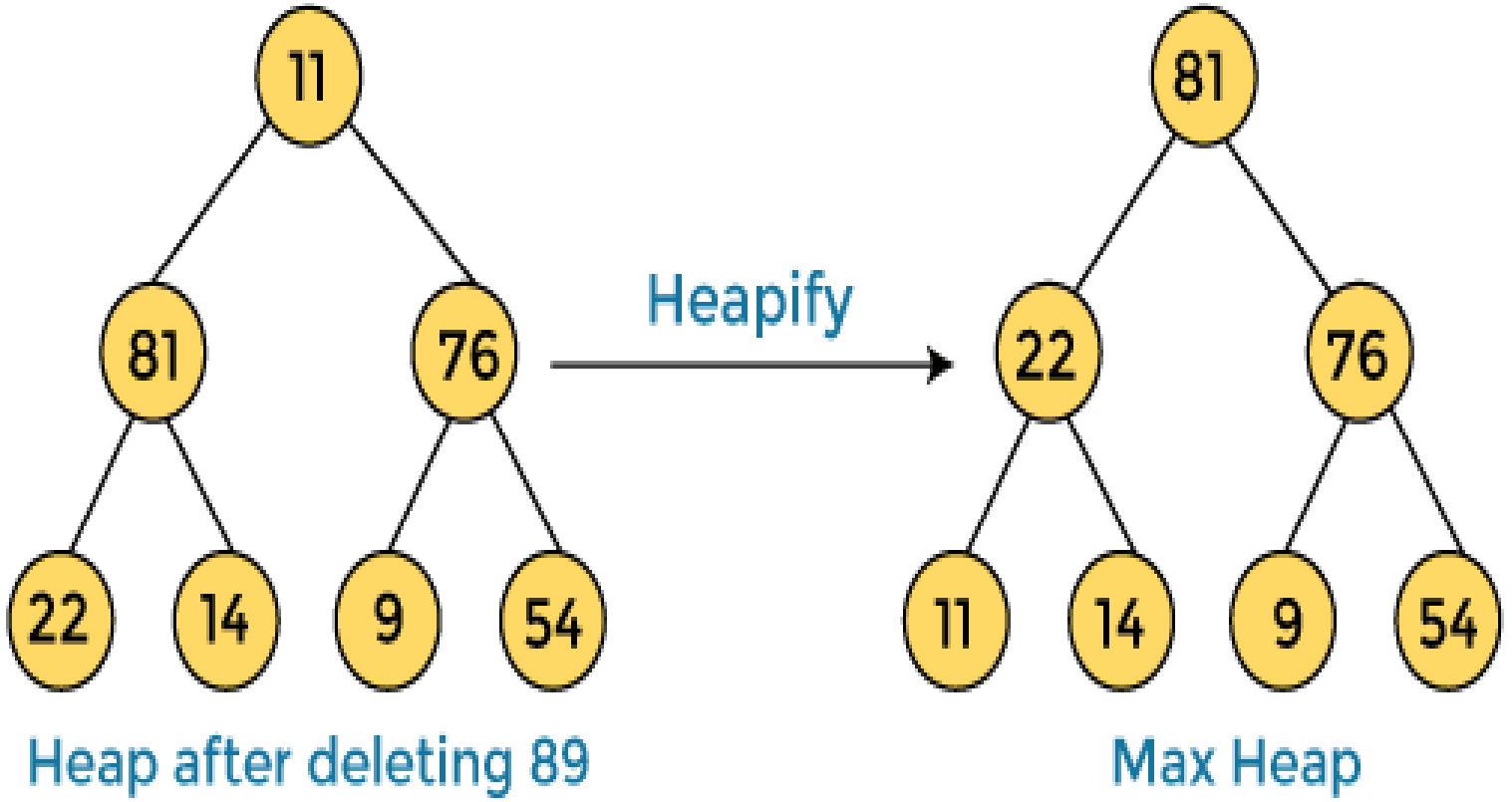
```
BuildMaxHeap(arr)
    heap_size(arr) = length(arr)
    for i = length(arr)/2 to 1
MaxHeapify(arr,i)
End
```

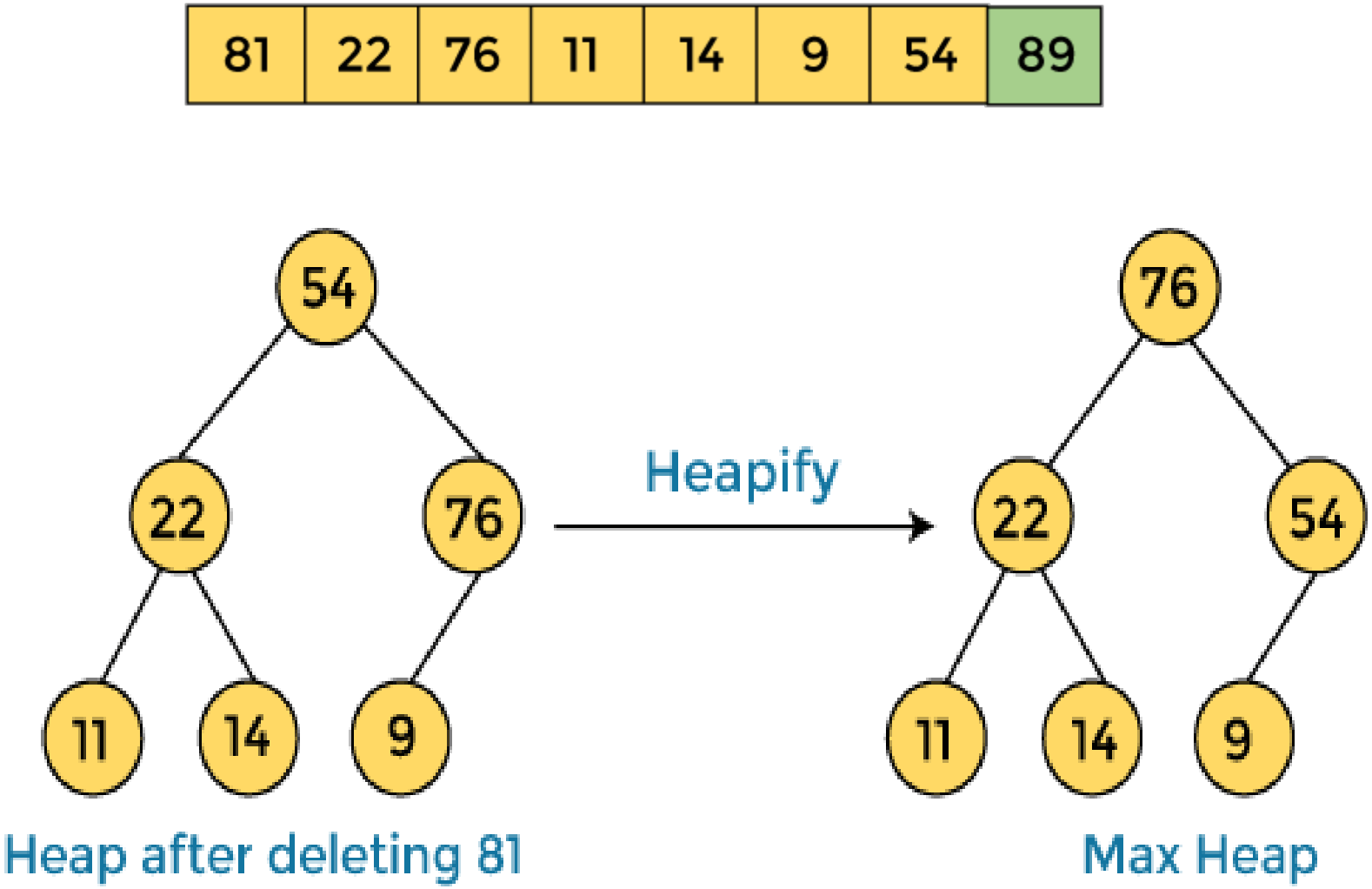
```
MaxHeapify(arr,i)
L = left(i)
R = right(i)
if L ? heap_size[arr] and arr[L] > arr[i]
largest = L
else
largest = i
if R ? heap_size[arr] and arr[R] > arr[largest]
largest = R
if largest != i
swap arr[i] with arr[largest]
MaxHeapify(arr,largest)
End
```

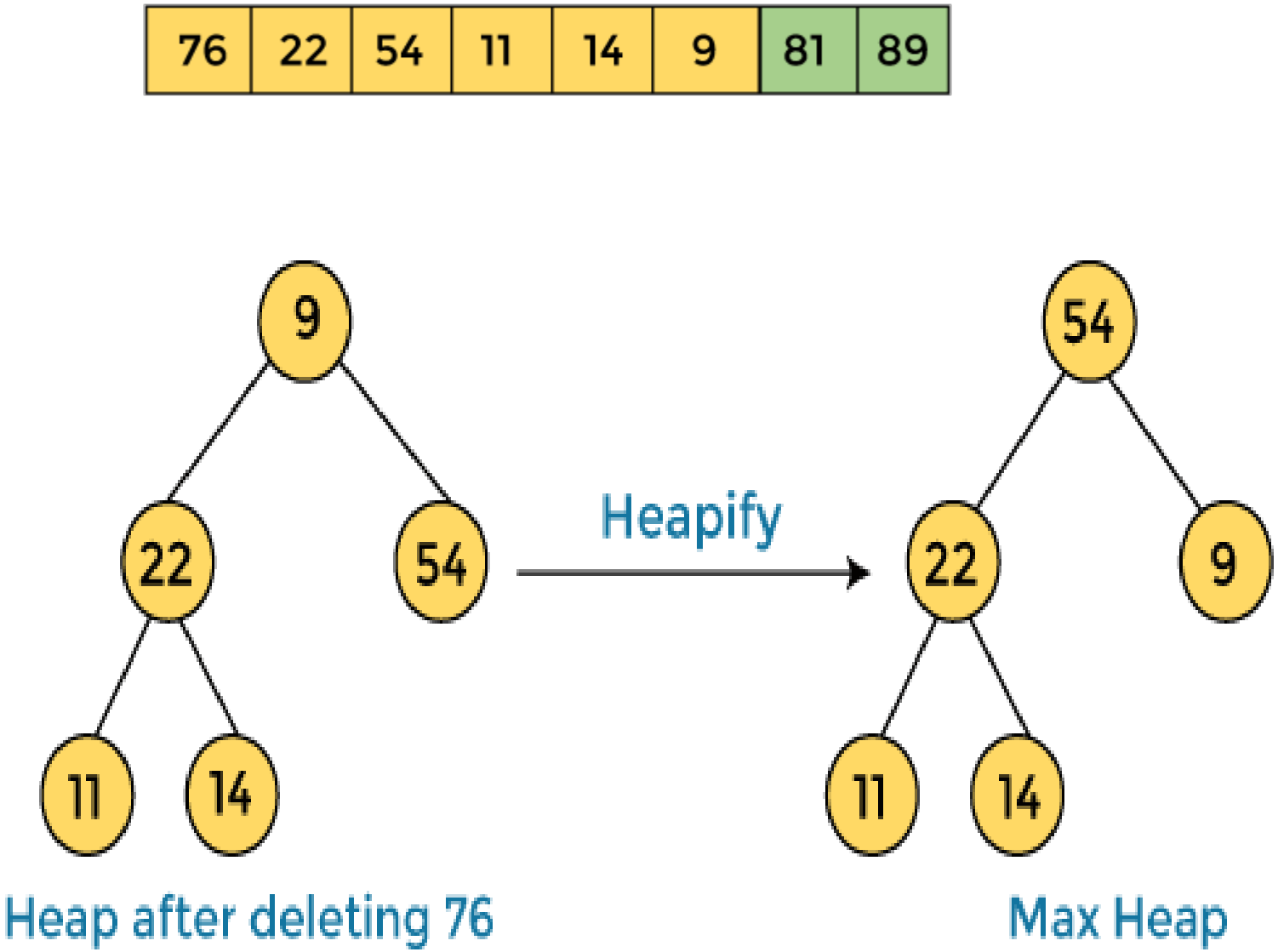

81
89
9
11
14
76
54
22

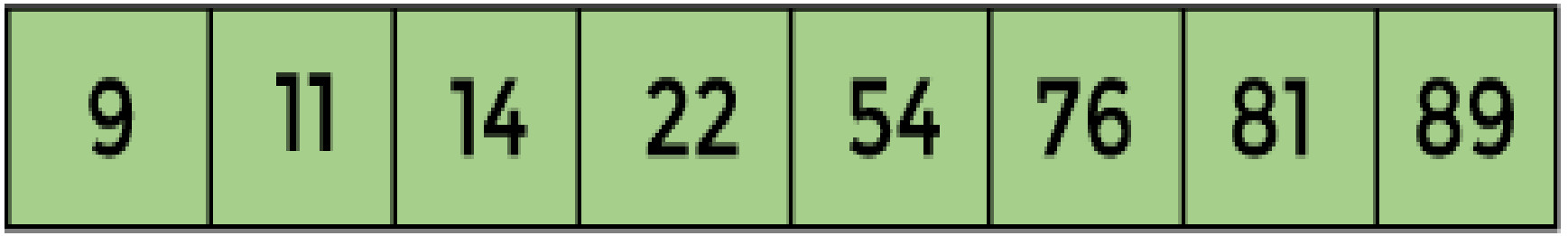

81
89
9
11
14
76
54
22
Heap
Heapify
89
81
76
22
14
9
54
11
Max Heap


89
81
76
22
14
9
54
11


11
81
76
22
14
9
54
Heapify
81
22
76
11
14
9
54
Heap after deleting 89
Max Heap


81
22
76
11
14
9
54
89
54
22
76
11
14
9
Heapify
76
22
54
11
14
9
Heap after deleting 81
Max Heap


76
22
54
11
14
9
81
89
9
22
54
11
14
Heapify
54
22
9
11
14
Heap after deleting 76
Max Heap


9
11
14
22
54
76
81
89

Thank You