UNIT -3

Searching & Sorting

Qpedia.live

Searching

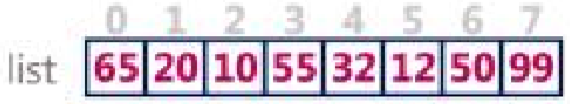
- Searching is an operation or a technique that helps finds the place of a given element or value in the list. Any search is said to be successful or unsuccessful depending upon whether the element that is being searched is found or not. Some of the standard searching technique that is being followed in the data structure is listed below:

- Sequential Search(Linear Search)
- Indexed Sequential Search
- Binary Search

Sequential Search (Linear Search)

- In this, the list or array is traversed sequentially and every element is checked. For example: Linear Search. Linear search is implemented using following steps...
- Step 1 -Read the search element from the user.
- Step 2 -Compare the search element with the first element in the list.
- Step 3 -If both are matched, then display "Given element is found!!!" and terminate the function
- Step 4 -If both are not matched, then compare search element with the next element in the list.
- Step 5 -Repeat steps 3 and 4 until search element is compared with last element in the list.
- Step 6 -If last element in the list also doesn't match, then display "Element is not found!!!" and terminate the function.

Example

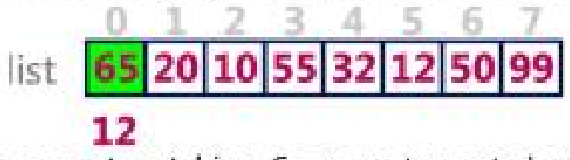


	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99

search element **12**

Step 1:

search element (12) is compared with first element (65)



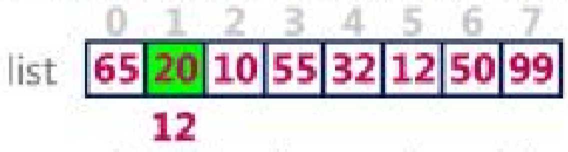
	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99

12

Both are not matching. So move to next element

Step 2:

search element (12) is compared with next element (20)



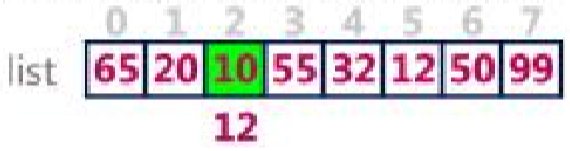
	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99

12

Both are not matching. So move to next element

Step 3:

search element (12) is compared with next element (10)



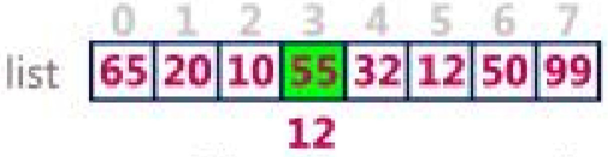
	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99

12

Both are not matching. So move to next element

Step 4:

search element (12) is compared with next element (55)



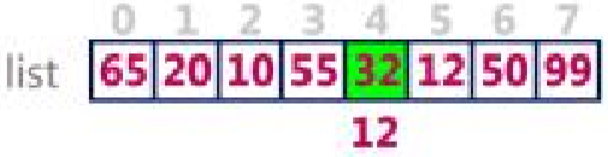
	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99

12

Both are not matching. So move to next element

Step 5:

search element (12) is compared with next element (32)



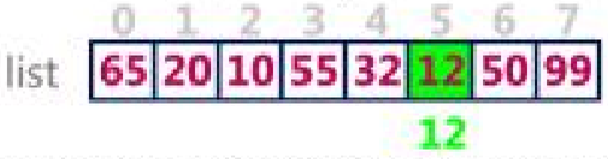
	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99

12

Both are not matching. So move to next element

Step 6:

search element (12) is compared with next element (12)



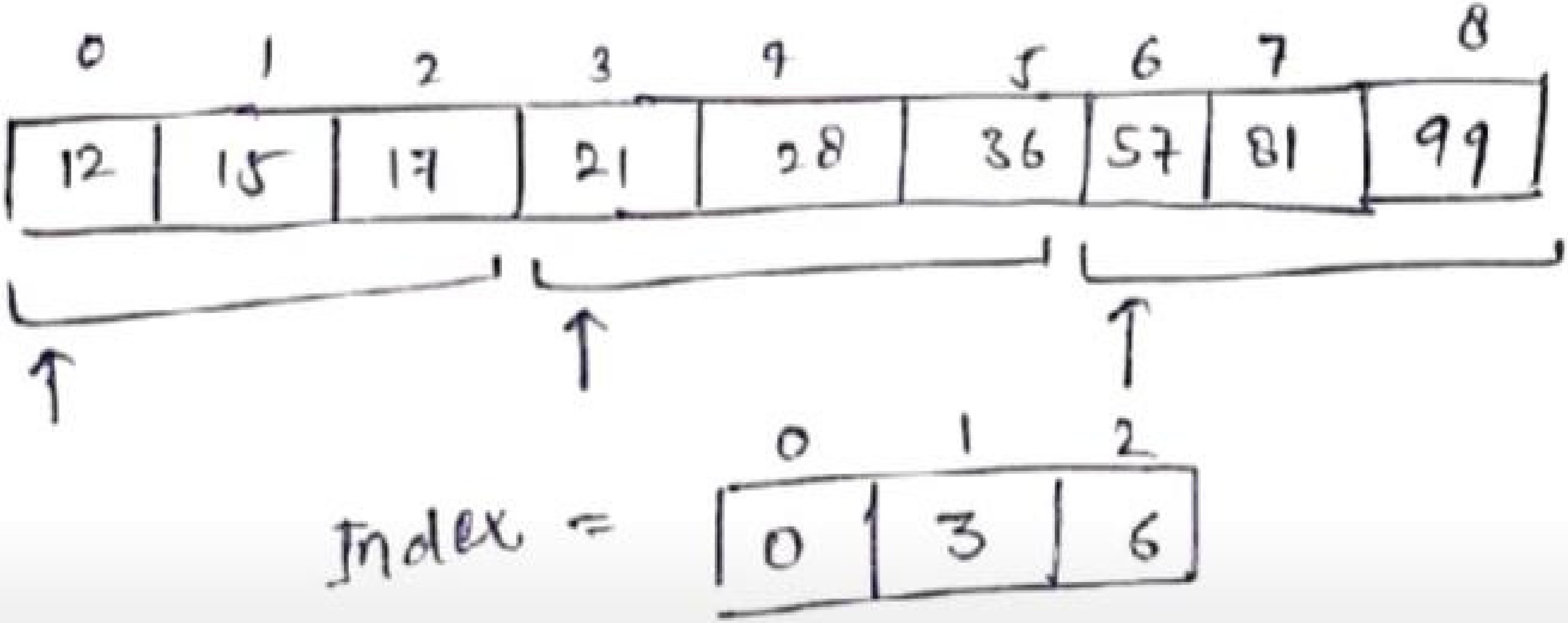
	0	1	2	3	4	5	6	7
list	65	20	10	55	32	12	50	99

12

Both are matching. So we stop comparing and display element found at index 5.

Indexed sequential se arch

- In this searching method, first of all, an index file is created, that contains some specific group or division of required record when the index is obtained, then the partial indexing takes less time cause it is located in a specified group.
- Note: When the user makes a request for specific records it will find that index group first where that specific record is recorded.
- Characteristics of Indexed Sequential Search:
- In Indexed Sequential Search a sorted index is set aside in addition to the array.
- Each element in the index points to a block of elements in the array or another expanded index.
- The index is searched 1st then the array and guides the search in the array. Note: Indexed Sequential Search actually does the indexing multiple times, like creating the index of an index.



0	1	2	3	4	5	6	7	8
12	15	17	21	28	36	57	81	99

Index =

0	1	2
0	3	6

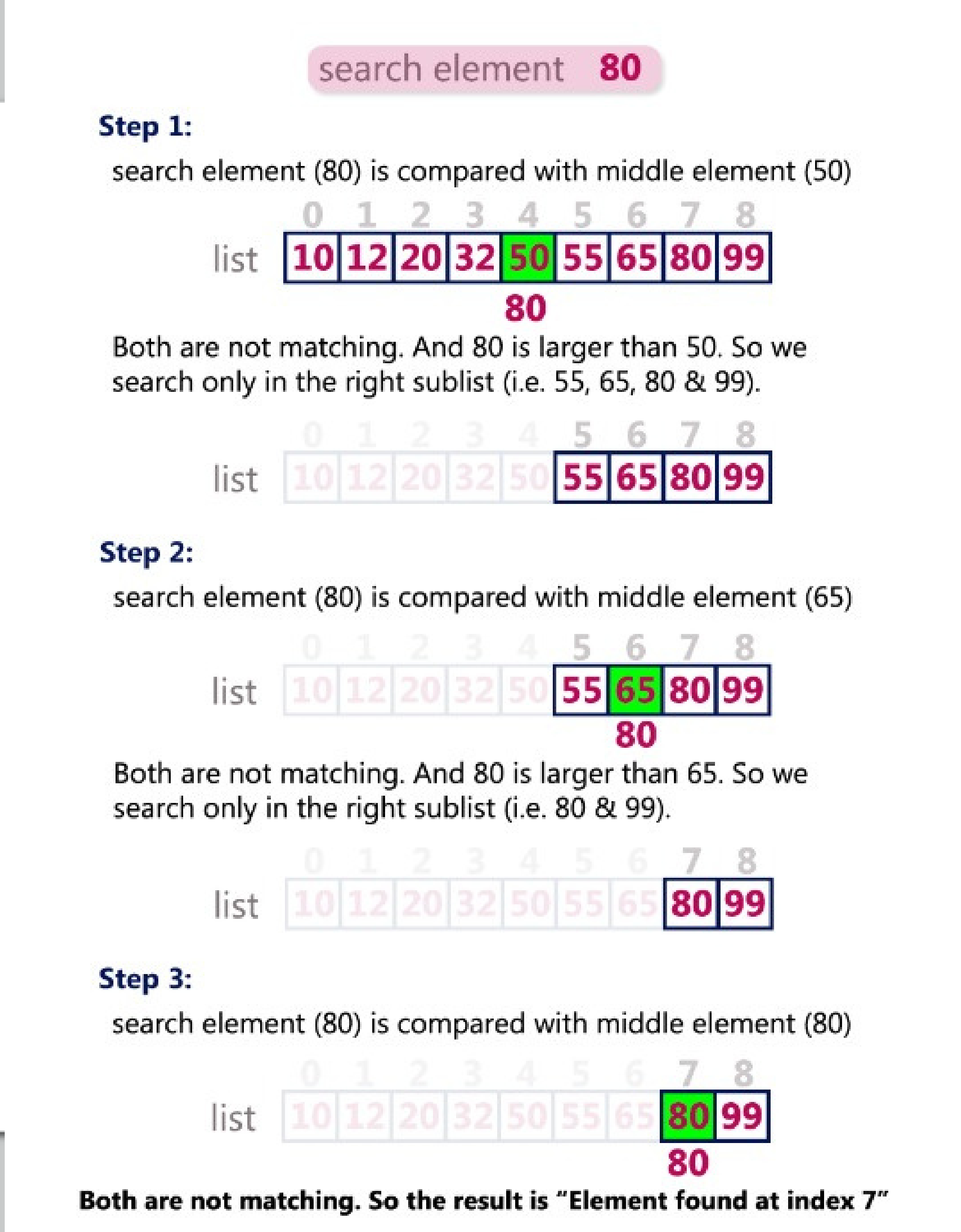
Key $\boxed{28}$

group size = $\boxed{3}$

Binary Search

- Binary search is implemented using following steps...
- Step 1 -Read the search element from the user.
- Step 2 -Find the middle element in the sorted list.
- Step 3 -Compare the search element with the middle element in the sorted list
- Step 4 -If both are matched, then display "Given element is found!!!" and terminate the function.
- Step 5 -If both are not matched, then check whether the search element is smaller or larger than the middle element.
- Step 6 -If the search element is smaller than middle element, repeat steps 2, 3, 4 and 5 for the left sublistof the middle element.
- Step 7 -If the search element is larger than middle element, repeat steps 2, 3, 4 and 5 for the right sublistof the middle element.
- Step 8 -Repeat the same process until we find the search element in the list or until sublistcontains only one element.
- Step 9 -If that element also doesn't match with the search element, then display "Element is not found in the list!!!" and terminate the function.

- **Example**



search element **80**

Step 1:

search element (80) is compared with middle element (50)

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	80	99
					80				

Both are not matching. And 80 is larger than 50. So we search only in the right sublist (i.e. 55, 65, 80 & 99).

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	80	99

Step 2:

search element (80) is compared with middle element (65)

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	80	99
							80		

Both are not matching. And 80 is larger than 65. So we search only in the right sublist (i.e. 80 & 99).

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	80	99

Step 3:

search element (80) is compared with middle element (80)

	0	1	2	3	4	5	6	7	8
list	10	12	20	32	50	55	65	80	99
								80	

Both are not matching. So the result is "Element found at index 7"

Sorting

- Sorting refers to the operation or technique of arranging and rearranging sets of data in some specific order.
- The techniques of sorting can be divided into two categories. These are:
- Internal Sorting
- External Sorting
- Internal Sorting: If all the data that is to be sorted can be adjusted at a time in the main memory, the internal sorting method is being performed.
- External Sorting: When the data that is to be sorted cannot be accommodated in the memory at the same time and some has to be kept in auxiliary memory such as hard disk, floppy disk, magnetic tapes etc, then external sorting methods are performed.

Complexity of sorting algorithm

- The complexity of sorting algorithm calculates the running time of a function in which 'n' number of items are to be sorted. The choice for which sorting method is suitable for a problem depends on several dependency configurations for different problems. The most noteworthy of these considerations are:

- The length of time spent by the programmer in programming a specific sorting program
- Amount of machine time necessary for running the program
- The amount of memory necessary for running the program
- Best case
- Worst case
- Average case

Bubble Sort

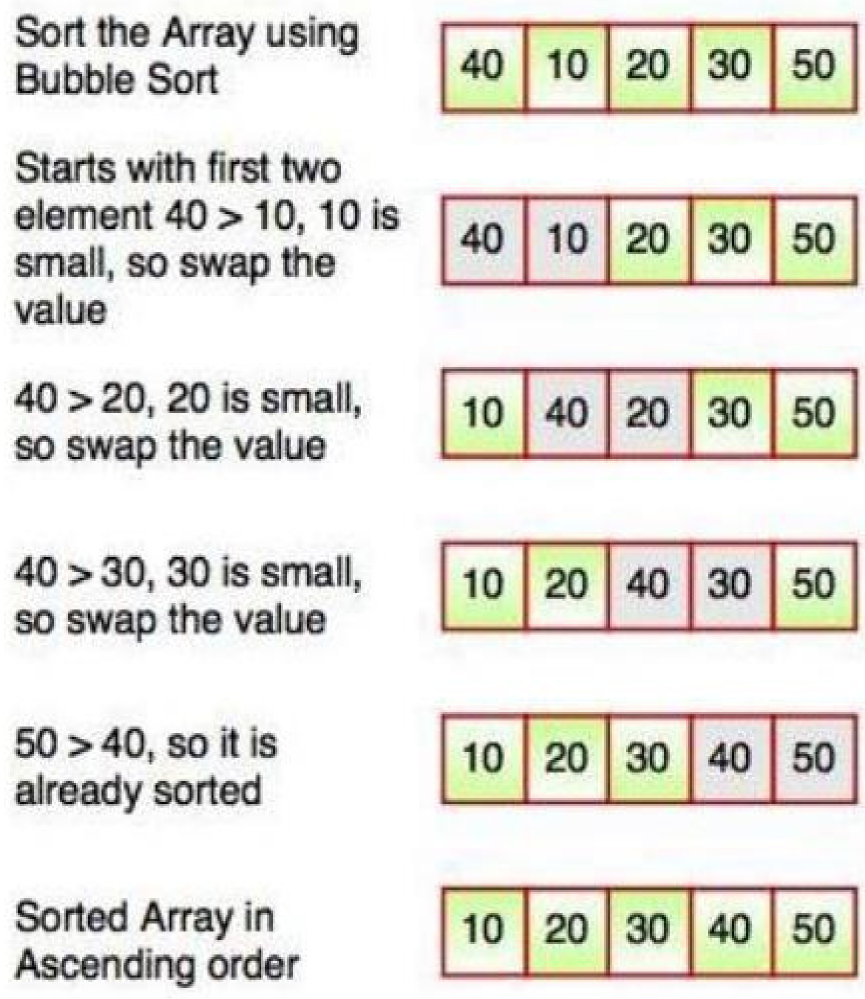
- Bubble Sort is a sorting algorithm, which is commonly used in computer science. Bubble Sort is based on the idea of repeatedly comparing pairs of adjacent elements and then swapping their positions if they exist in the wrong order.

•Steps on how it works:(with 5 elements)

1. In an unsorted array of 5 elements, start with the first two elements and sort them in ascending order. (Compare the element to check which one is greater).

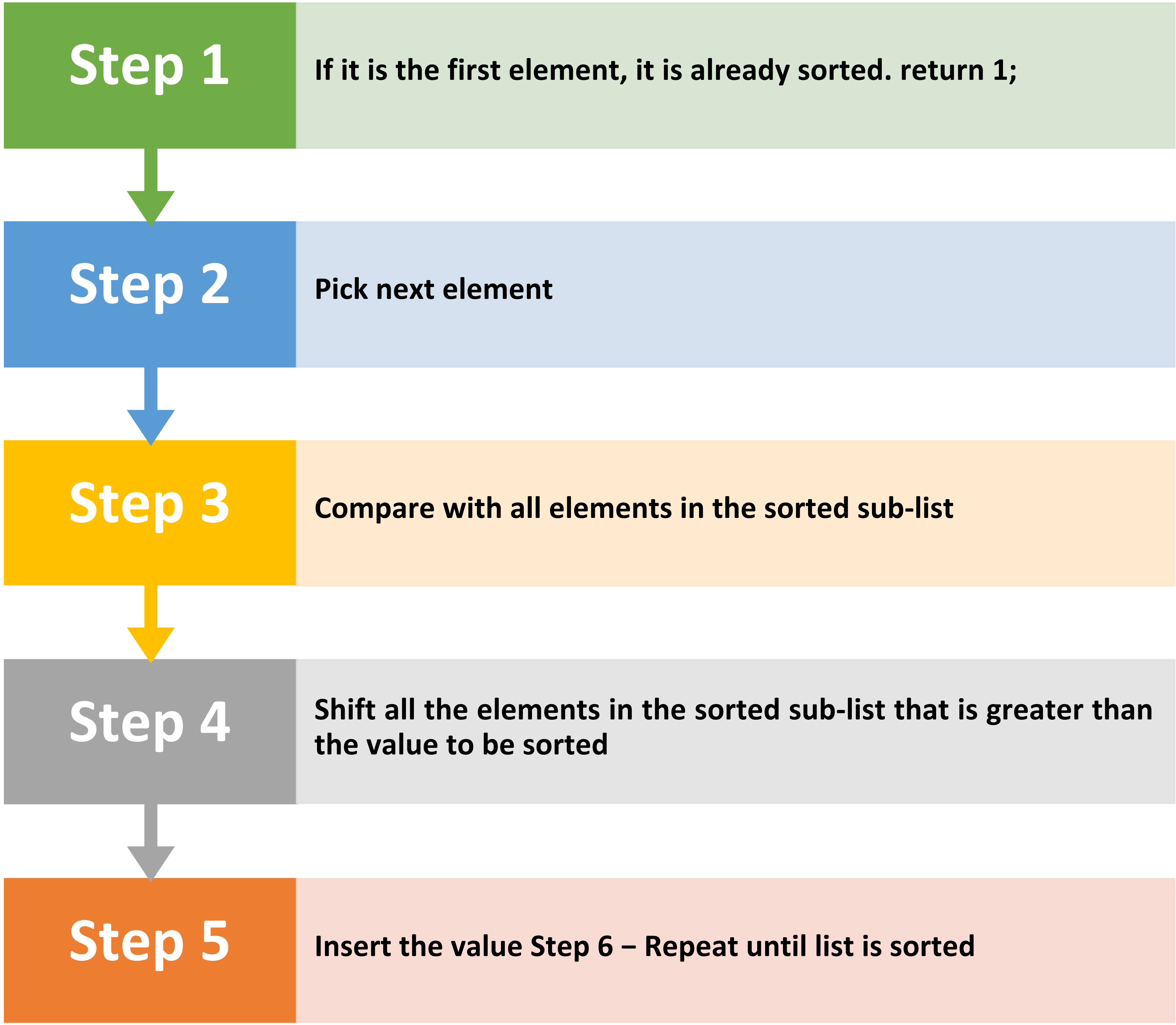
2 Compare the second and third element to check which
one is greater, and sort them in ascending order.
. Compare the third and fourth element to check which one
is greater, and sort them in ascending order.
3 4 Compare the fourth and fifth element to check which one
is greater, and sort them in ascending order.
: Repeat steps 1–5 until no more swaps are required.
5

Below is an image of an array, which needs to be sorted. We will use the Bubble Sort Algorithm, to sort this array:



Sort the Array using Bubble Sort	40	10	20	30	50
Starts with first two element 40 > 10, 10 is small, so swap the value	40	10	20	30	50
40 > 20, 20 is small, so swap the value	10	40	20	30	50
40 > 30, 30 is small, so swap the value	10	20	40	30	50
50 > 40, so it is already sorted	10	20	30	40	50
Sorted Array in Ascending order	10	20	30	40	50

Insertion Sort



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

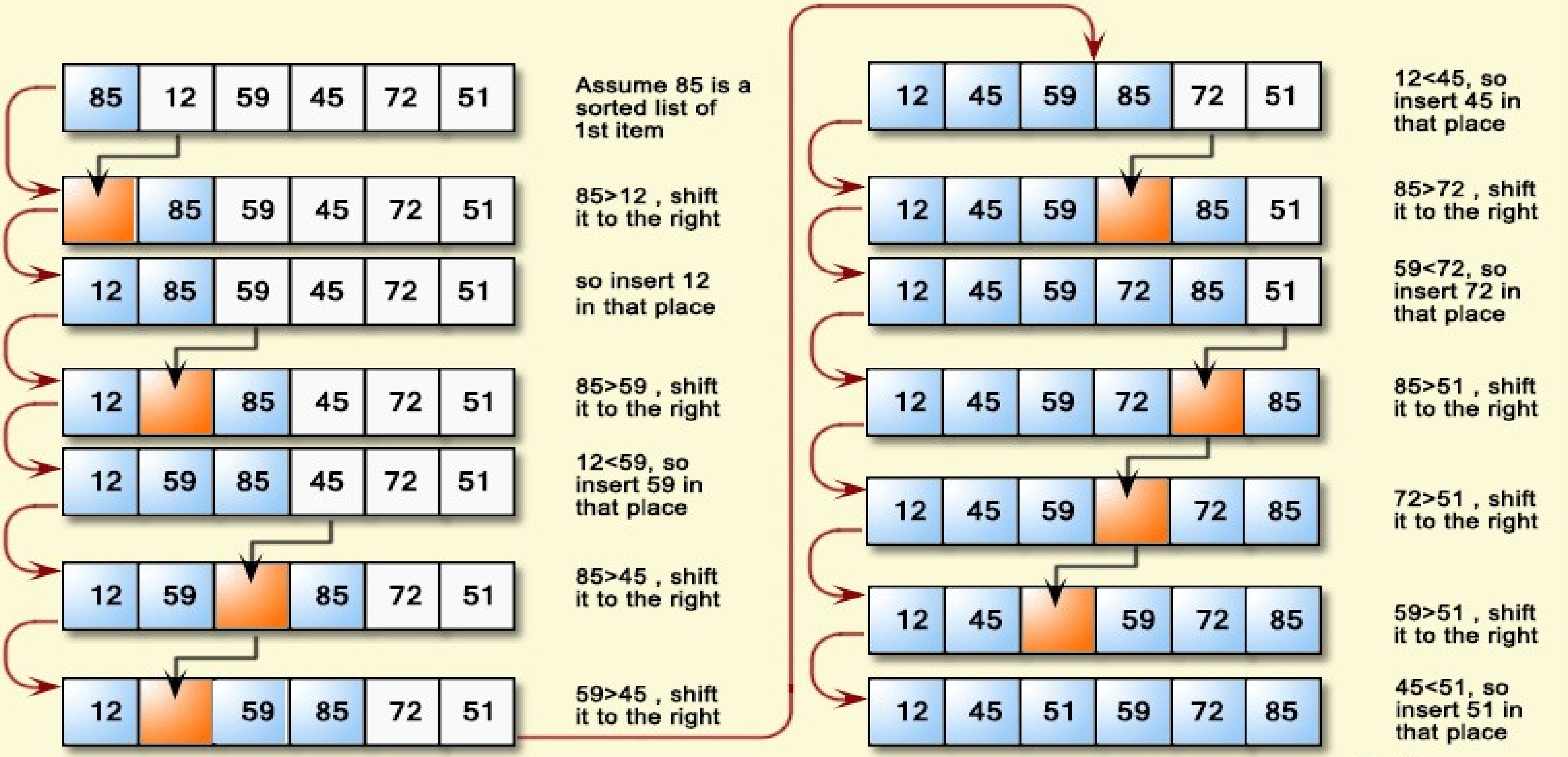
Step 2 Pick next element

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



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

Selection Sort

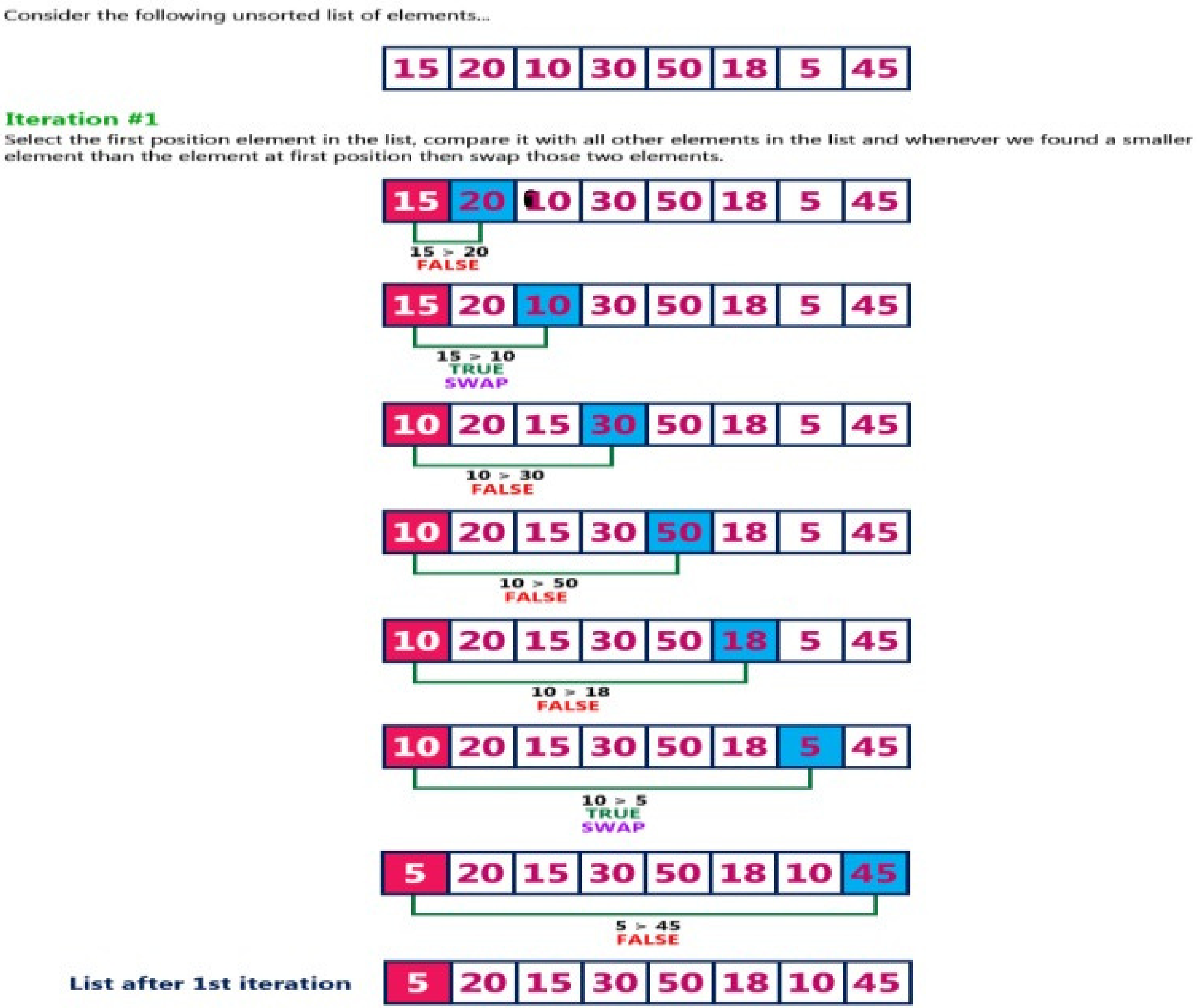
•• Step 1 -Select the first element of the list (i.e., Element at first position in the list).

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

•• Step 3: In every comparison, if any element is found smaller than the selected element (for Ascending order), then both are swapped.

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

Example



Consider the following unsorted list of elements...

15	20	10	30	50	18	5	45

Iteration #1

Select the first position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

15	20	10	30	50	18	5	45

15 > 20
FALSE

15	20	10	30	50	18	5	45

15 > 10
TRUE
SWAP

10	20	15	30	50	18	5	45

10 > 30
FALSE

10	20	15	30	50	18	5	45

10 > 50
FALSE

10	20	15	30	50	18	5	45

10 > 18
FALSE

10	20	15	30	50	18	5	45

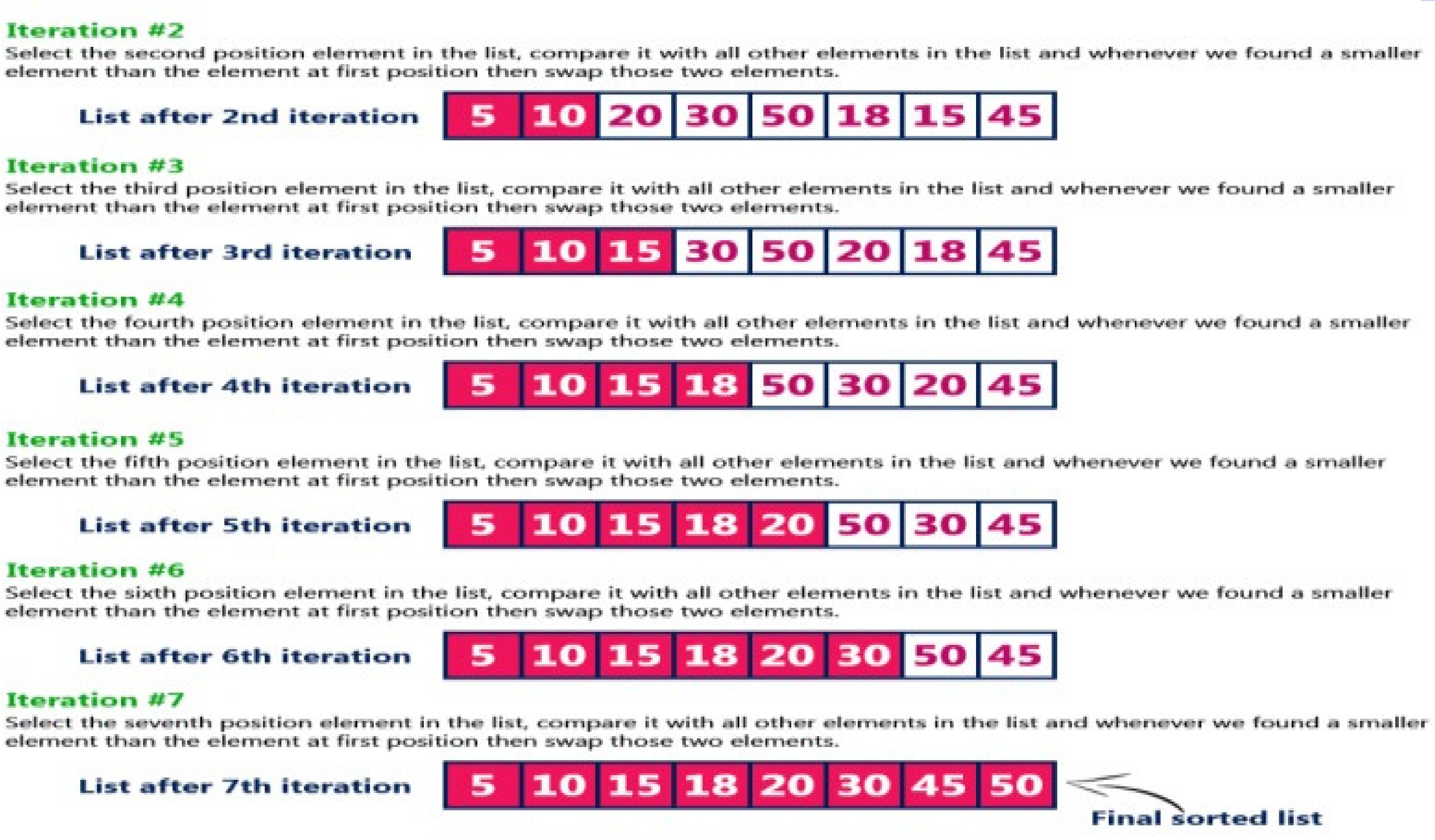
10 > 5
TRUE
SWAP

5	20	15	30	50	18	10	45

5 > 45
FALSE

List after 1st iteration

5	20	15	30	50	18	10	45



Iteration #2

Select the second position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 2nd iteration	5	10	20	30	50	18	15	45

Iteration #3

Select the third position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 3rd iteration	5	10	15	30	50	20	18	45

Iteration #4

Select the fourth position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 4th iteration	5	10	15	18	50	30	20	45

Iteration #5

Select the fifth position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 5th iteration	5	10	15	18	20	50	30	45

Iteration #6

Select the sixth position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 6th iteration	5	10	15	18	20	30	50	45

Iteration #7

Select the seventh position element in the list, compare it with all other elements in the list and whenever we found a smaller element than the element at first position then swap those two elements.

List after 7th iteration	5	10	15	18	20	30	45	50

Final sorted list

Quick Sort

- In Quick sort algorithm, partitioning of the list is performed using following steps...

•Step 1 -Consider the first element of the list as pivot (i.e., Element at first position in the list).

•Step 2 -Define two variables iand j. Set iand j to first and last elements of the list respectively.

- Step 3 -Increment iuntil list[i] > pivot then stop.
- Step 4 -Decrement j until list[j] < pivot then stop.
- Step 5 -If i< j then exchange list[i] and list[j].
- Step 6 -Repeat steps 3,4 & 5 until i> j.
- Step 7 -Exchange the pivot element with list[j] element.

Example

Consider the following unsorted list of elements...

List	5	3	8	1	4	6	2	7

Define pivot, left & right. Set pivot = 0, left = 1 & right = 7. Here '7' indicates 'size-1'.

		left						right
List	5	3	8	1	4	6	2	7
	pivot							

Compare List[left] with List[pivot]. If **List[left]** is greater than **List[pivot]** then stop left otherwise move left to the next.

Compare List[right] with List[pivot]. If List[right] is smaller than List[pivot] then stop right otherwise move right to the previous.

Repeat the same until **left>=right**.

If both left & right are stoped but left<right then swap List[left] with List[right] and countinue the process.

If left>=right then swap List[pivot] with List[right].

		left						right
List	5	3	8	1	4	6	2	7
	pivot							

Compare List[left]<List[pivot] as it is true increment left by one and repeat the same, left will stop at 8.

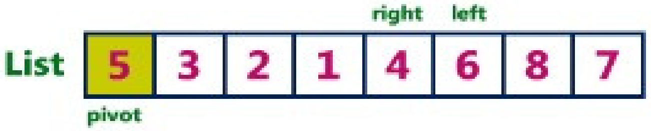
Compare List[right]>List[pivot] as it is true decrement right by one and repeat the same, right will stop at 2.

			left				right	
List	5	3	8	1	4	6	2	7
	pivot							

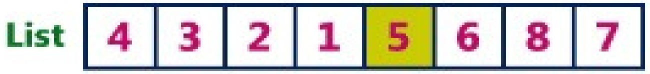
Here left & right both are stoped and left is not greater than right so we need to swap List[left] and List[right]

			left				right	
List	5	3	2	1	4	6	8	7
	pivot							

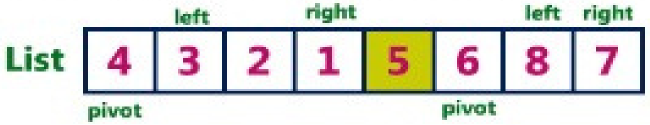
Compare List[left]<List[pivot] as it is true increment left by one and repeat the same, left will stop at 6.
Compare List[right]>List[pivot] as it is true decrement right by one and repeat the same, right will stop at 4.



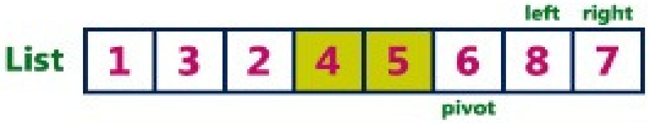
Here left & right both are stoped and left is greater than right so we need to swap List[pivot] and List[right]



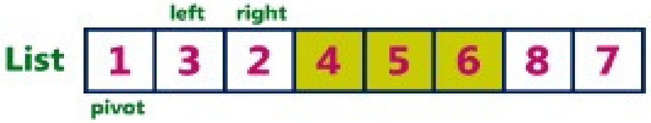
Here we can observe that all the numbers to the left side of 5 are smaller and right side are greater. That means 5 is placed in its correct position.
Repeat the same process on the left sublist and right sublist to the number 5.



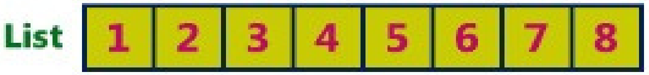
In the left sublist as there are no smaller number than the pivot left will keep on moving to the next and stops at last number. As the List[right] is smaller, right stops at same position. Now left and right both are equal so we swap pivot with right.



In the right sublist left is grester than the pivot, left will stop at same position.
As the List[right] is greater than List[pivot], right moves towords left and stops at pivot number position.
Now left > right so we swap pivot with right. (6 is swap by itself).



Repeat the same recursively on both left and right sublists until all the numbers are sorted.
The final sorted list will be as follows...



Merge Sort

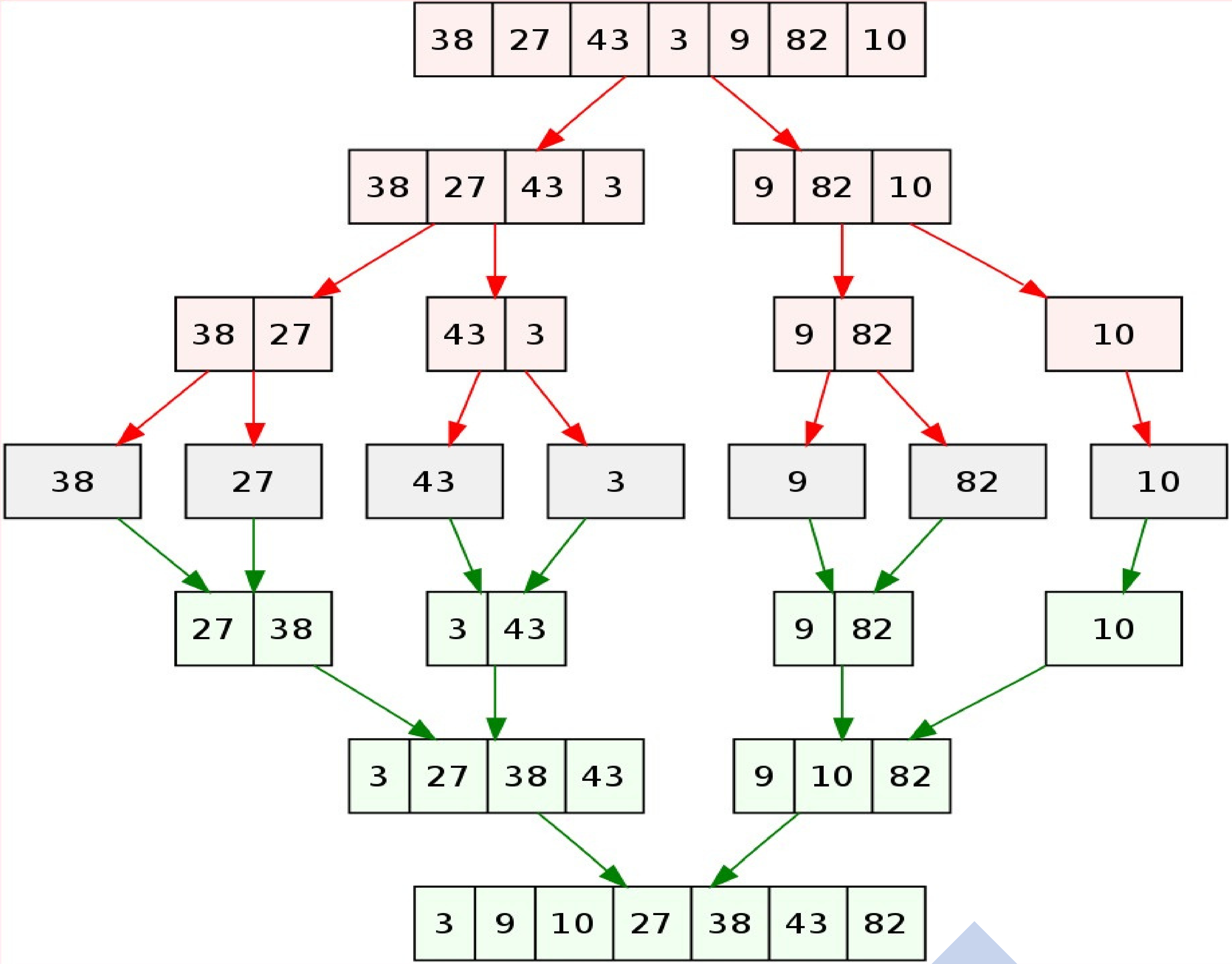
- **Step 1. Find the middle index of the array to divide it in two halves: m = (l+r)/2.**
- **Step 2. Call MergeSortfor first half: mergeSort(array, l, m)**
- **Step 3. Call mergeSortfor second half:**

  mergeSort(array, m+1, r)
- **Step 4. Recursively, merge the two halves in**

  a sorted manner, so that only one sorted array is left:

Example

- The following diagram shows the complete merge sort process for an example array {38, 27, 43, 3, 9, 82, 10}. If we take a closer look at the diagram, we can see that the array is recursively divided in two halves till the size becomes 1. Once the size becomes 1, the merge processes comes into action and starts merging arrays back till the complete array is merged.



38 27 43 3 9 82 10

38 27 43 3 | 9 82 10

38 27 | 43 3 | 9 82 | 10

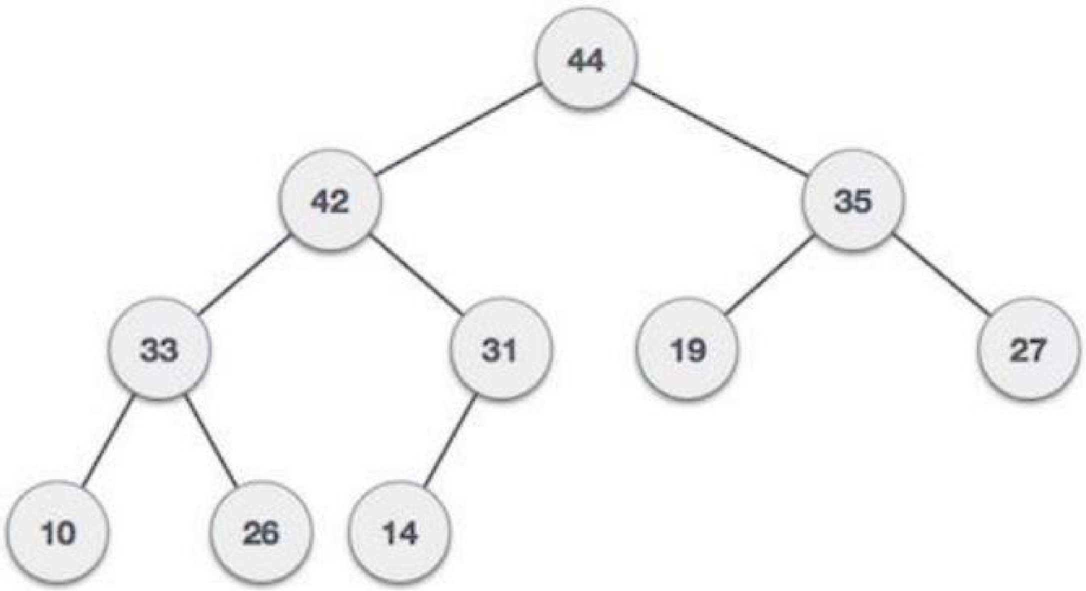
38 | 27 | 43 | 3 | 9 | 82 | 10

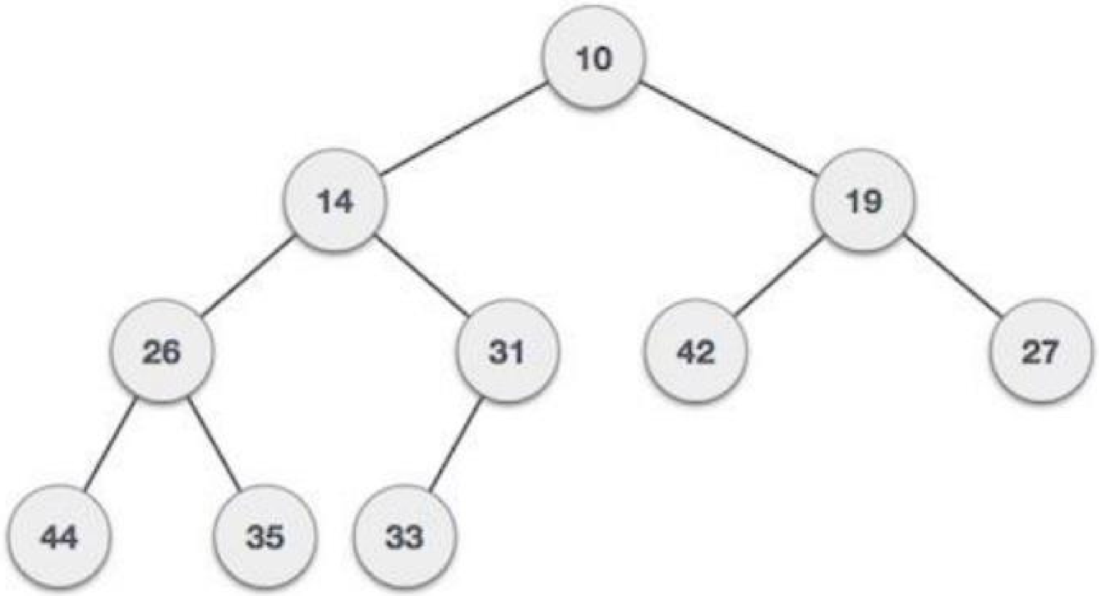
27 38 | 3 43 | 9 82 | 10

3 27 38 43 | 9 10 82

3 9 10 27 38 43 82

MAX & Min -Heap





Heap Sort

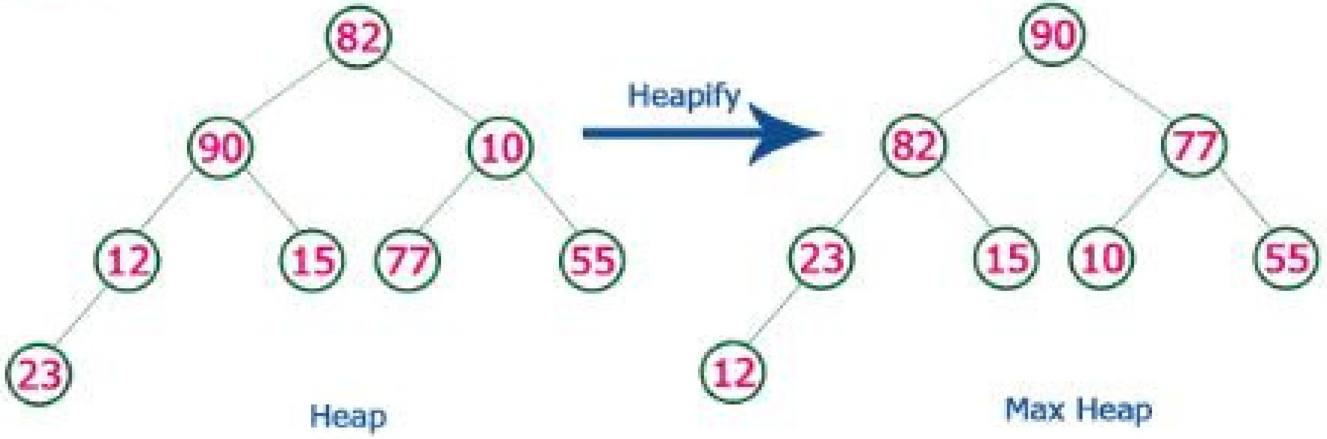
- **The Heap sort algorithm to arrange a list of elements in ascending order is performed using following steps...**
- **Step 1 -Construct a Binary Tree with given list of Elements.**
- **Step 2 -Transform the Binary Tree into Max Heap.**
- **Step 3 -Delete the root element from Max Heap using Heapifymethod.**
- **Step 4 -Repeat the same until Min Heap becomes empty.**
- **Step 5 -Display the sorted list.**

Example

Consider the following list of unsorted numbers which are to be sort using Heap Sort

82, 90, 10, 12, 15, 77, 55, 23

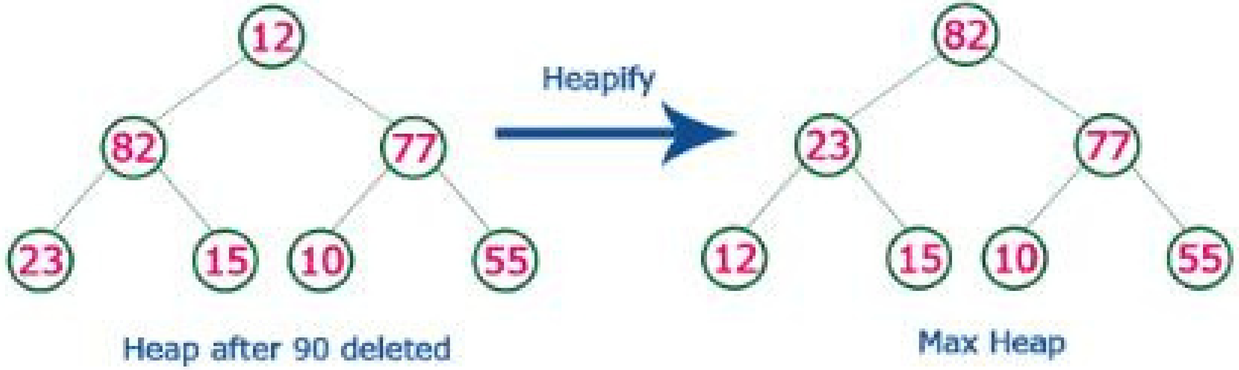
Step 1 - Construct a Heap with given list of unsorted numbers and convert to Max Heap



list of numbers after heap converted to Max Heap

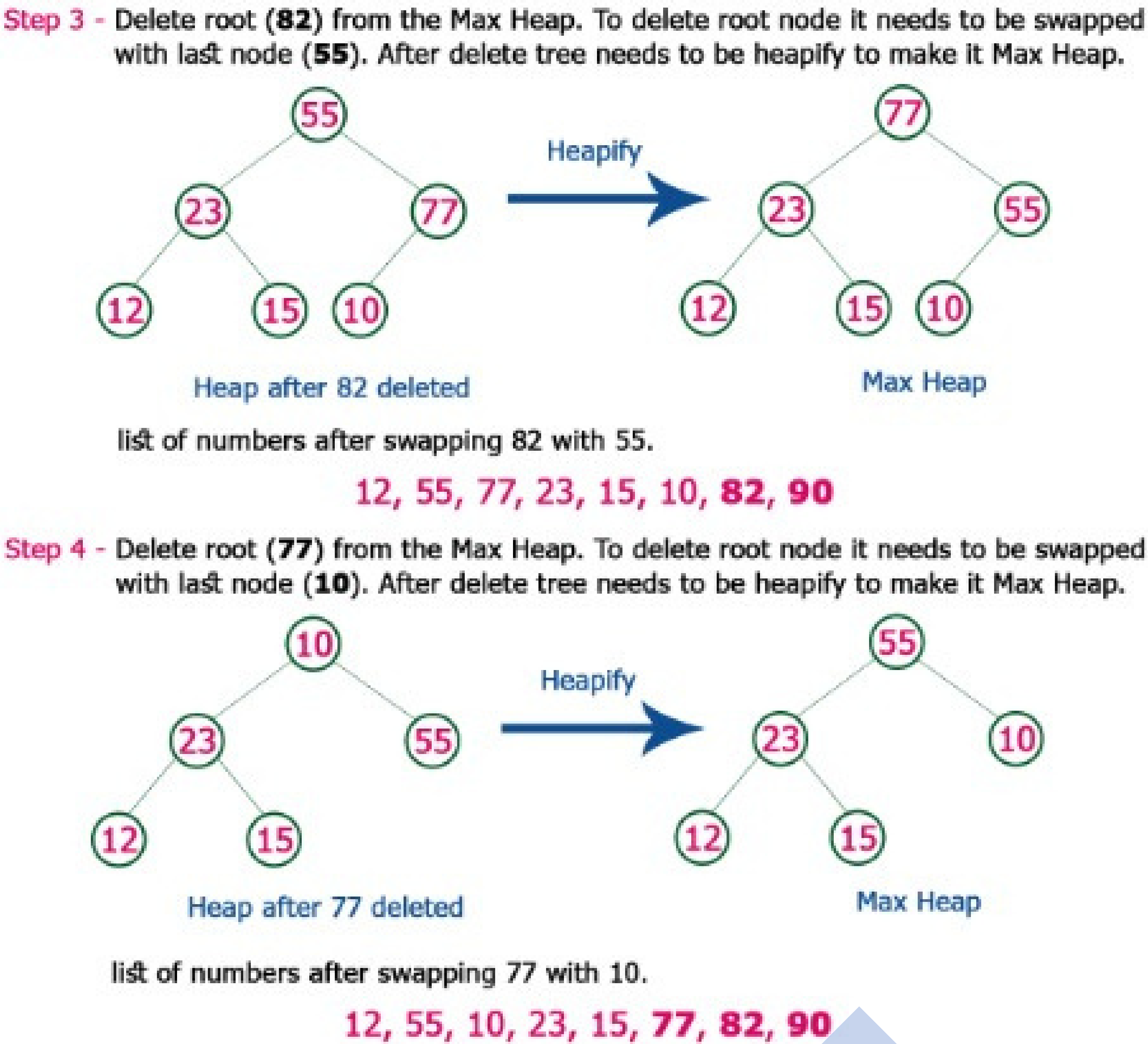
90, 82, 77, 23, 15, 10, 55, 12

Step 2 - Delete root (**90**) from the Max Heap. To delete root node it needs to be swapped with last node (**12**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 90 with 12.

12, 82, 77, 23, 15, 10, 55, **90**



Step 3 - Delete root (**82**) from the Max Heap. To delete root node it needs to be swapped with last node (**55**). After delete tree needs to be heapify to make it Max Heap.

Heap after 82 deleted

Max Heap

list of numbers after swapping 82 with 55.

12, 55, 77, 23, 15, 10, **82**, **90**

Step 4 - Delete root (**77**) from the Max Heap. To delete root node it needs to be swapped with last node (**10**). After delete tree needs to be heapify to make it Max Heap.

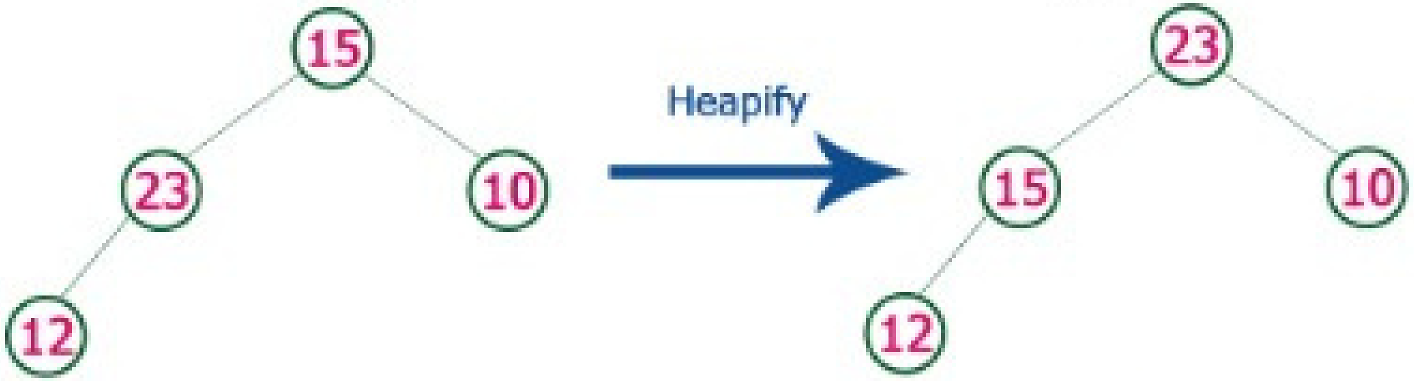
Heap after 77 deleted

Max Heap

list of numbers after swapping 77 with 10.

12, 55, 10, 23, 15, **77**, **82**, **90**

Step 5 - Delete root (**55**) from the Max Heap. To delete root node it needs to be swapped with last node (**15**). After delete tree needs to be heapify to make it Max Heap.



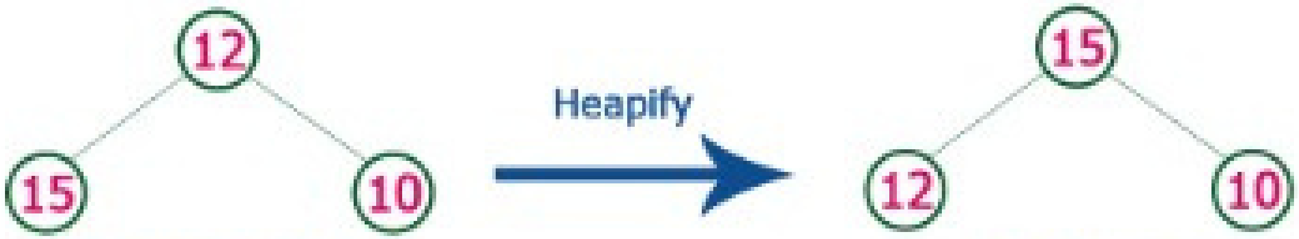
Heap after 55 deleted

Max Heap

list of numbers after swapping 55 with 15.

12, 15, 10, 23, **55, 77, 82, 90**

Step 6 - Delete root (**23**) from the Max Heap. To delete root node it needs to be swapped with last node (**12**). After delete tree needs to be heapify to make it Max Heap.



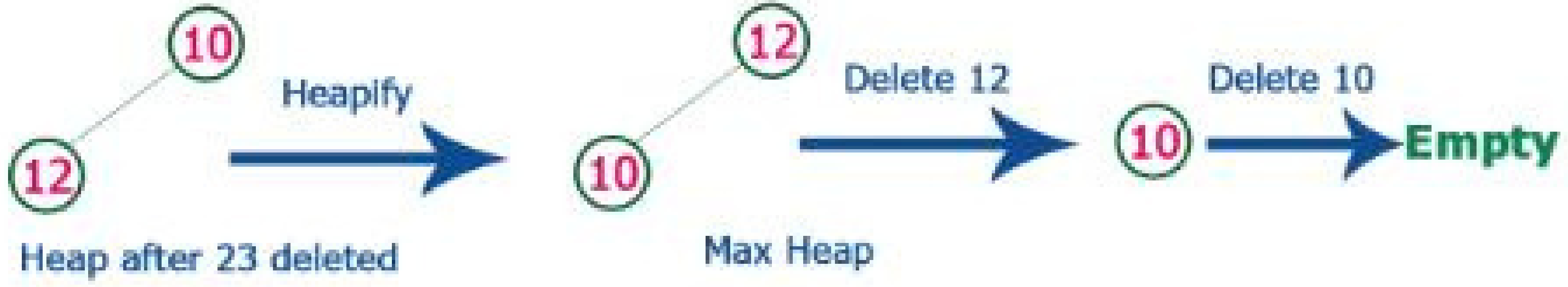
Heap after 23 deleted

Max Heap

list of numbers after swapping 23 with 12.

12, 15, 10, **23, 55, 77, 82, 90**

Step 7 - Delete root (**15**) from the Max Heap. To delete root node it needs to be swapped with last node (**10**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after Deleting 15, 12 & 10 from the Max Heap.

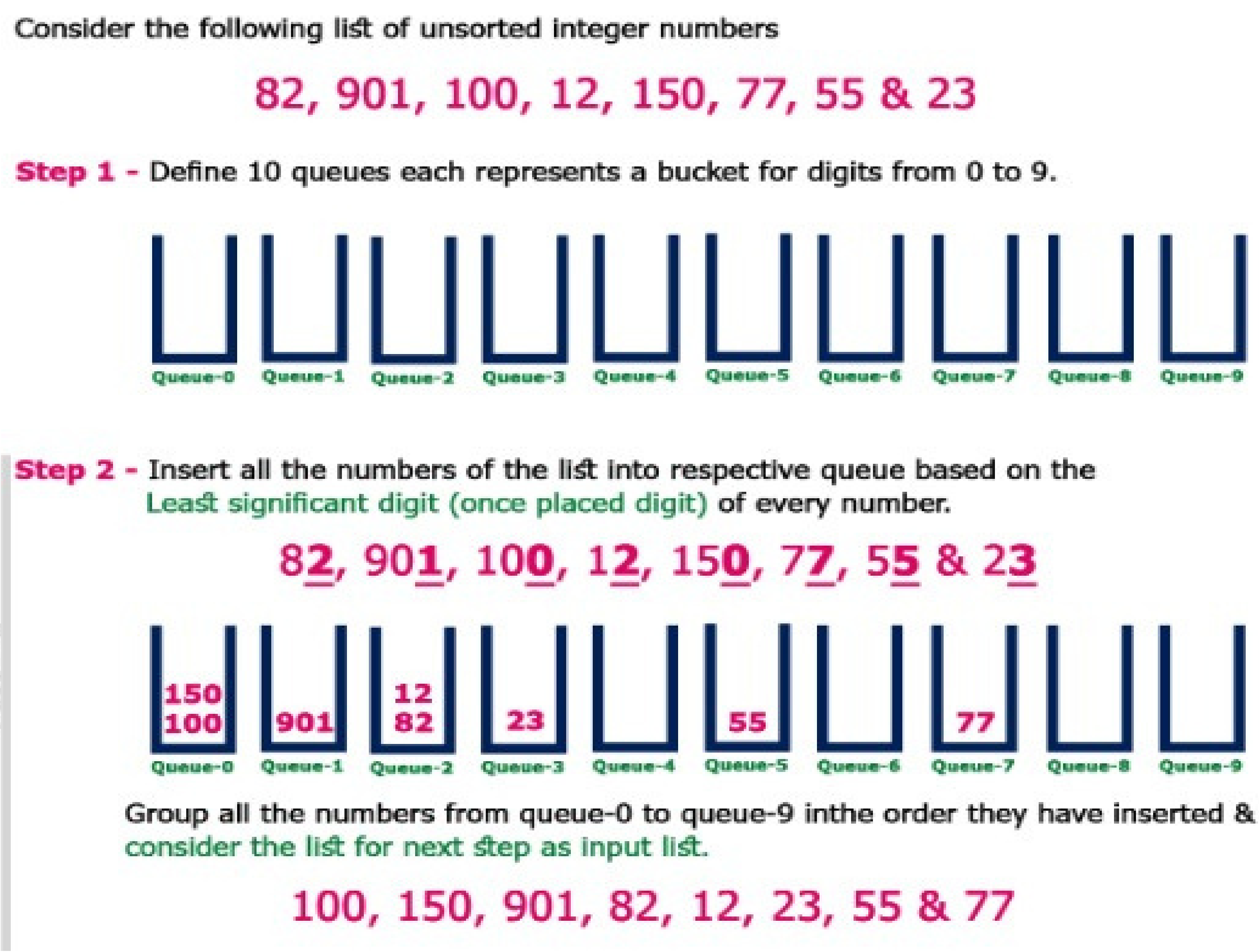
10, 12, 15, 23, 55, 77, 82, 90

Whenever Max Heap becomes Empty, the list get sorted in Ascending order

Radix sort

- The Radix sort algorithm is performed using the following steps...
- Step 1 -Define 10 queues each representing a bucket for each digit from 0 to 9.
- Step 2 -Consider the least significant digit of each number in the list which is to be sorted.
- Step 3 -Insert each number into their respective queue based on the least significant digit.
- Step 4 -Group all the numbers from queue 0 to queue 9 in the order they have inserted into their respective queues.
- Step 5 -Repeat from step 3 based on the next least significant digit.
- Step 6 -Repeat from step 2 until all the numbers are grouped based on the most significant digit.

Example



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.

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

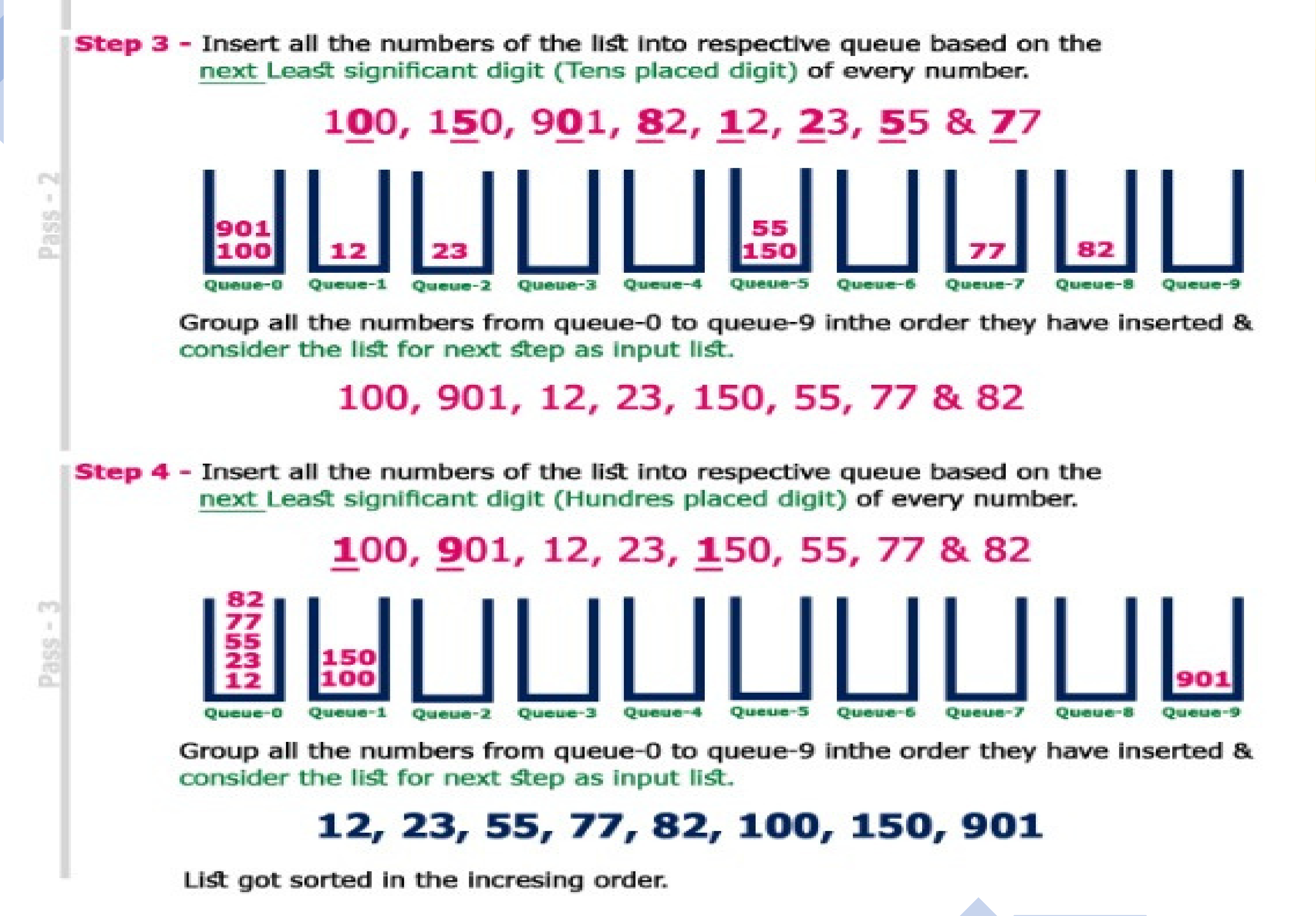
Step 2 - Insert all the numbers of the list into respective queue based on the Least significant digit (once placed digit) of every number.

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

Pass - 1
150
100
Queue-0
901
Queue-1
12
82
Queue-2
23
Queue-3
Queue-4
55
Queue-5
Queue-6
77
Queue-7
Queue-8
Queue-9

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 (Hundres 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 incresing order.

Hashing Me chanism

- **In hashing,**
- **An array data structure called as Hash table is used to store the data items.**
- **Based on the hash key value, data items are inserted into the hash table.**
- **•Hash Key Value-**
- **Hash key value is a special value that serves as an index for a data item.**
- **It indicates where the data item should be stored in the hash table.**
- **Hash key value is generated using a hash function.**

Division method

- **In this the hash function is dependent upon the remainder of a division. For example:-if the record 52,68,99,84 is to be placed in a hash table and let us take the table size is 10.**

Then:

h(key)=record% table size.

2=52%10

8=68%10

9=99%10

4=84%10

Mid square method

- In this method firstly key is squared and then mid part of the result is taken as the index.
- For example:
- consider that if we want to place a record of 3101 and the size of table is 1000.
- So
- 3101*3101=9616201
- i.e.
- h (3101) = 162 (middle 3 digit)

Digit folding method

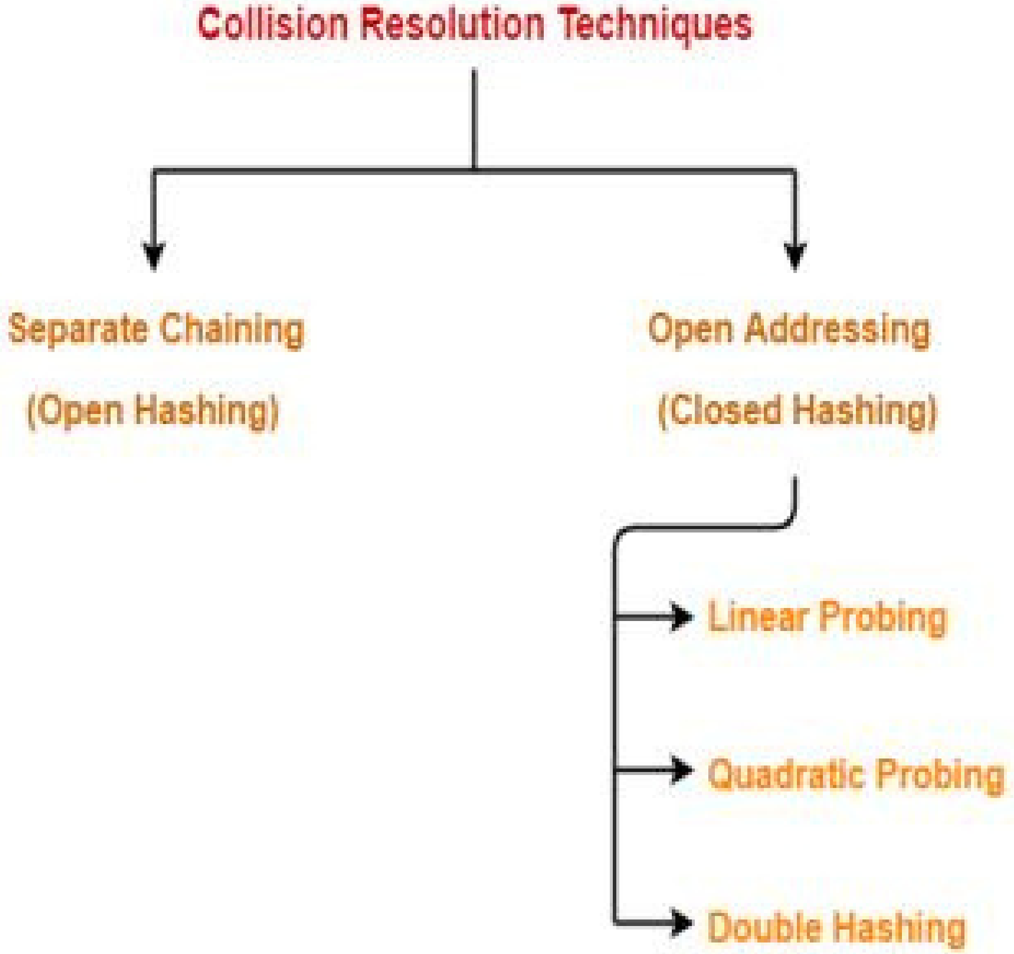
- In this method the key is divided into separate parts and by using some simple operations these parts are combined to produce a hash key.
- For example:
- consider a record of 12465512 then it will be divided into parts
- i.e. 124, 655, 12. After dividing the parts combine these parts by adding it.
- H(key)=124+655+12 =791

Collision in Hashing

- When the hash value of a key maps to an already occupied bucket of the hash table, it is called as a Collision.. In hashing,
  - Hash function is used to compute the hash value for a key.
  - Hash value is then used as an index to store the key in the hash table.
  - Hash function may return the same hash value for two or more keys

Collision Resolution Techniques

are the techniques used for resolving or handling the collision.



Collision Resolution Techniques

Separate Chaining

(Open Hashing)

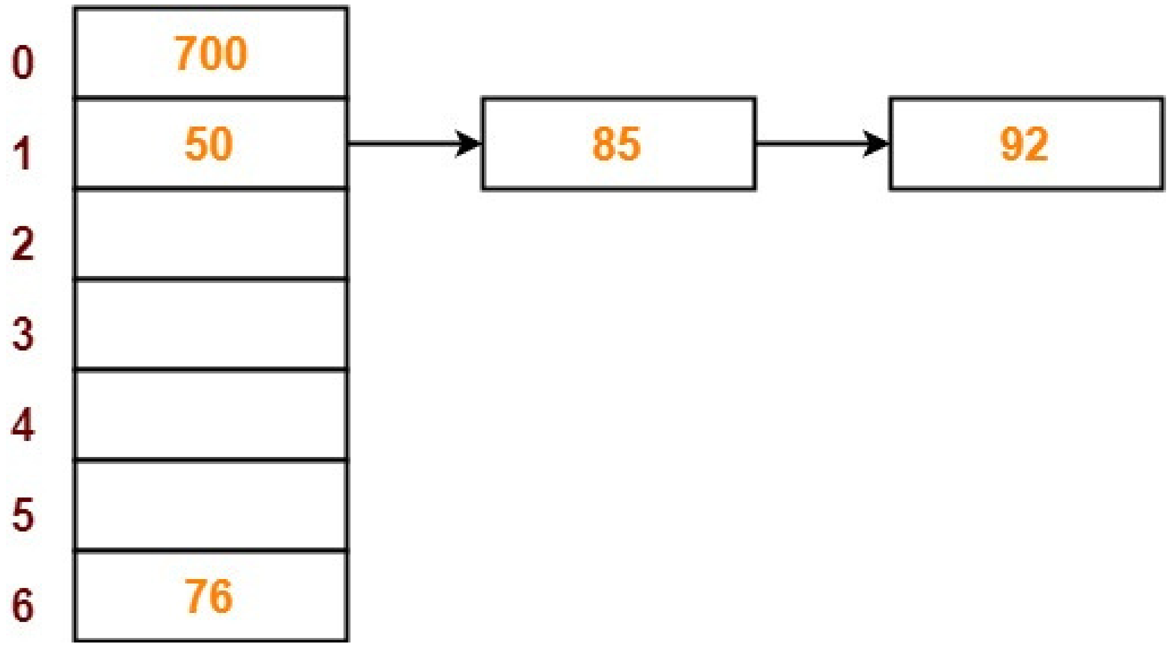
Open Addressing

(Closed Hashing)

Linear Probing

Quadratic Probing

Double Hashing


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