Parul® University

Unit 5
Sorting and hashing

Prof. Praveen Kumar, Assistant Professor
Prof. Rajiv Kumar, Assistant Professor
Computer science and Engineering

Topic-1

Sorting

Introduction

❖**Sorting** is a process that organizes a collection of data Into either ascending or descending order.

❖Following are the real-life scenario of sorting

1) Telephone directories
2) Dictionary

Image source : Google

Classification of Sorting Algorithms

- ❖*Stable Vs Unstable Sorting*
- ❖Internal Vs External Sort
- ❖In-Place Vs not-In-Place Sorting
- ❖Comparison based Sorting
- ❖ Counting based Sorting

Introduction

- There are many Well known sorting algorithms, such as:
  - ✓ Selection Sort
  - ✓ Insertion Sort
  - ✓ Bubble Sort
  - ✓ Merge Sort
  - ✓ Quick Sort
  - ✓ Heap Sort

Bubble Sort

❖The smallest element is bubbled from the unsorted list and moved to the sorted sub list.

❖After that, the wall moves one element ahead, increasing the number of sorted elements and decreasing the number of unsorted ones.

❖Each time an element moves from the unsorted part to the sorted part one sort pass is completed.

❖Given a list of n elements, bubble sort requires up to n-1 passes to sort the data

Algorithms of *Bubble Sort*

❖ **Algorithms:**

```
✓Step 1: Repeat Step 2 For i = 0 to N-1
✓Step 2: Repeat For J = i + 1 to N - I
✓Step 3: IF A[J] > A[i]
    Swap A[J] and A[i]
    [End of inner loop]
    [end of outer loop]
✓Step 4: Exit
```

Working of Bubble Sort

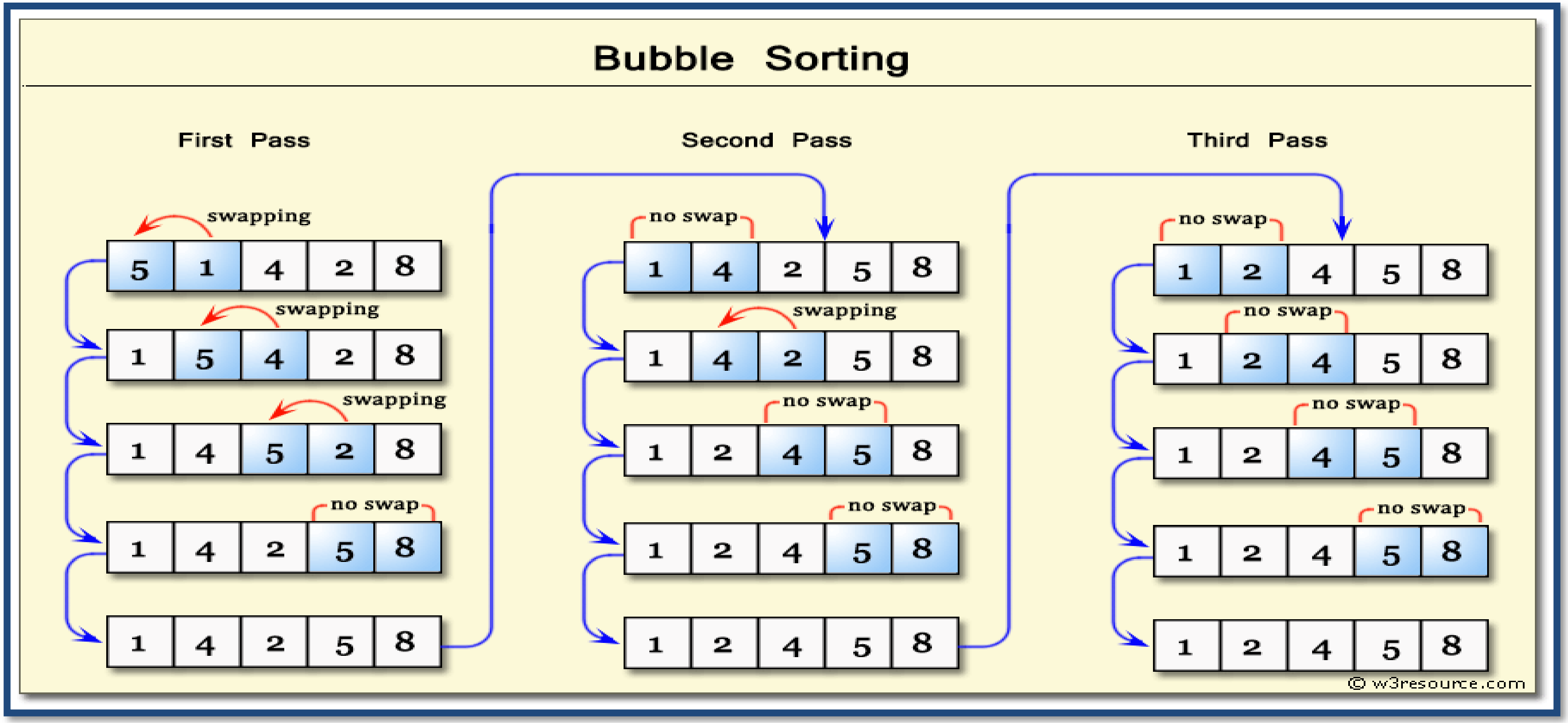


Image source : Google

Analysis of Bubble Sort

❖***1.Best-case:*** **O(n)**

✓ Array is already sorted in ascending order

❖***2.Worst-case:*** **O(n2)**

✓ Worst case occurs when array is reverse sorted.

❖**3.Average-case: O(n2)**

✓ We have to look at all possible initial data organizations

Selection Sort

❖ We find the smallest element from the unsorted sub list and swap it with the element at the beginning of the unsorted data.

❖After each selection and swapping, the imaginary wall between the two sub lists move one element ahead, increasing the number of sorted elements and decreasing the number of unsorted ones

❖Each time we move one element from the unsorted sub list to the sorted sub list, we say that we have completed a sort pass.

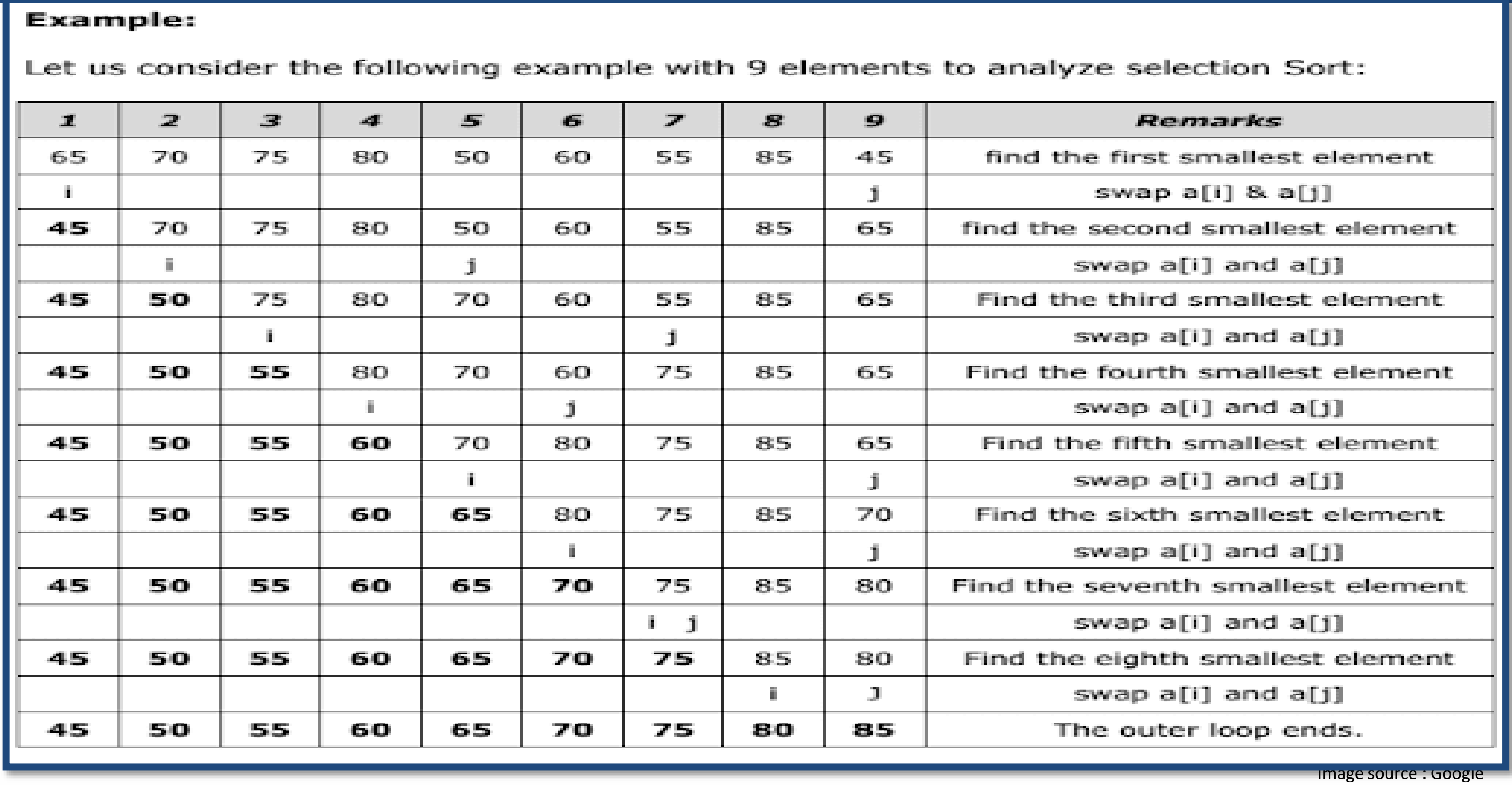
❖ A list of *n* elements requires *n-1* passes to completely rearrange the data.

Algorithms of *Selection Sort*

❖Algorithms

```
SELECTION SORT(A[], N)
✓Step 1: Repeat Steps 2 and 3 for i = 1 to N-1
✓Step 2: Call Smallest (A[], i, N, pos)
✓Step 3:SwapA[i] with A[pos]
[End of Loop]
✓Step 4: Exit
✓Smallest (A[], i, N, POS)
✓Step 1: [Initialize] set Small= A[i]
✓Step 2: [Initialize] Set  pos = i
✓Step 3: Repeat for J = i+1 to N -1
if small > A[J]
Set small = A[J]
Set pos = J
[End of if ]
[End of Loop]
✓Step 4: Return pos
```

Working of Selection Sort



Example:

Let us consider the following example with 9 elements to analyze selection Sort:

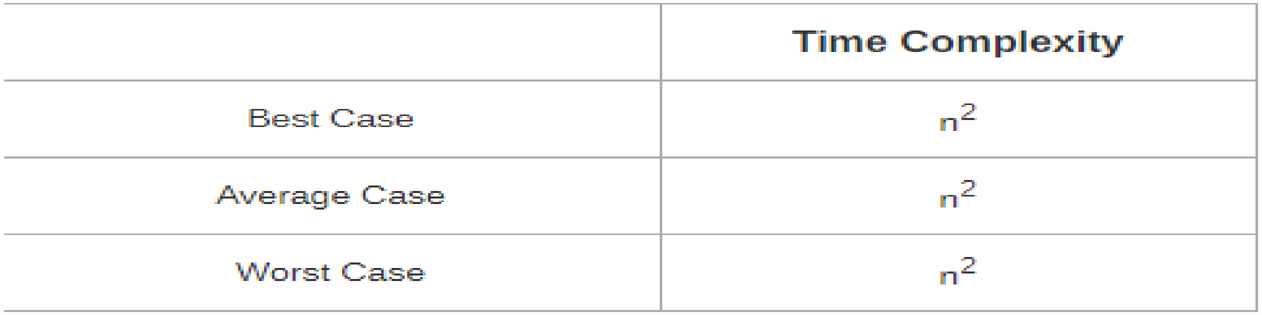
1	2	3	4	5	6	7	8	9	Remarks
65	70	75	80	50	60	55	85	45	find the first smallest element
i								j	swap a[i] & a[j]
45	70	75	80	50	60	55	85	65	find the second smallest element
	i			j					swap a[i] and a[j]
45	50	75	80	70	60	55	85	65	Find the third smallest element
		i				j			swap a[i] and a[j]
45	50	55	80	70	60	75	85	65	Find the fourth smallest element
			i		j				swap a[i] and a[j]
45	50	55	60	70	80	75	85	65	Find the fifth smallest element
				i				j	swap a[i] and a[j]
45	50	55	60	65	80	75	85	70	Find the sixth smallest element
					i			j	swap a[i] and a[j]
45	50	55	60	65	70	75	85	80	Find the seventh smallest element
						i j			swap a[i] and a[j]
45	50	55	60	65	70	75	85	80	Find the eighth smallest element
							i	j	swap a[i] and a[j]
45	50	55	60	65	70	75	80	85	The outer loop ends.

Image source : Google

Analysis of Selection Sort

Time Complexity Analysis-

- ✓ Selection sort algorithm consists of two nested loops.
- ✓ Owing to the two nested loops, it has O(n^2) time complexity.



	Time Complexity
Best Case	n^2
Average Case	n^2
Worst Case	n^2

Image source : Google

Insertion Sort

❖Insertion sort is the simple sorting algorithm which is commonly used in the daily lives while ordering a deck of cards.

❖In each pass, the first element of the unsorted part is picked up, transferred to the sorted sub list, and inserted at the appropriate place.

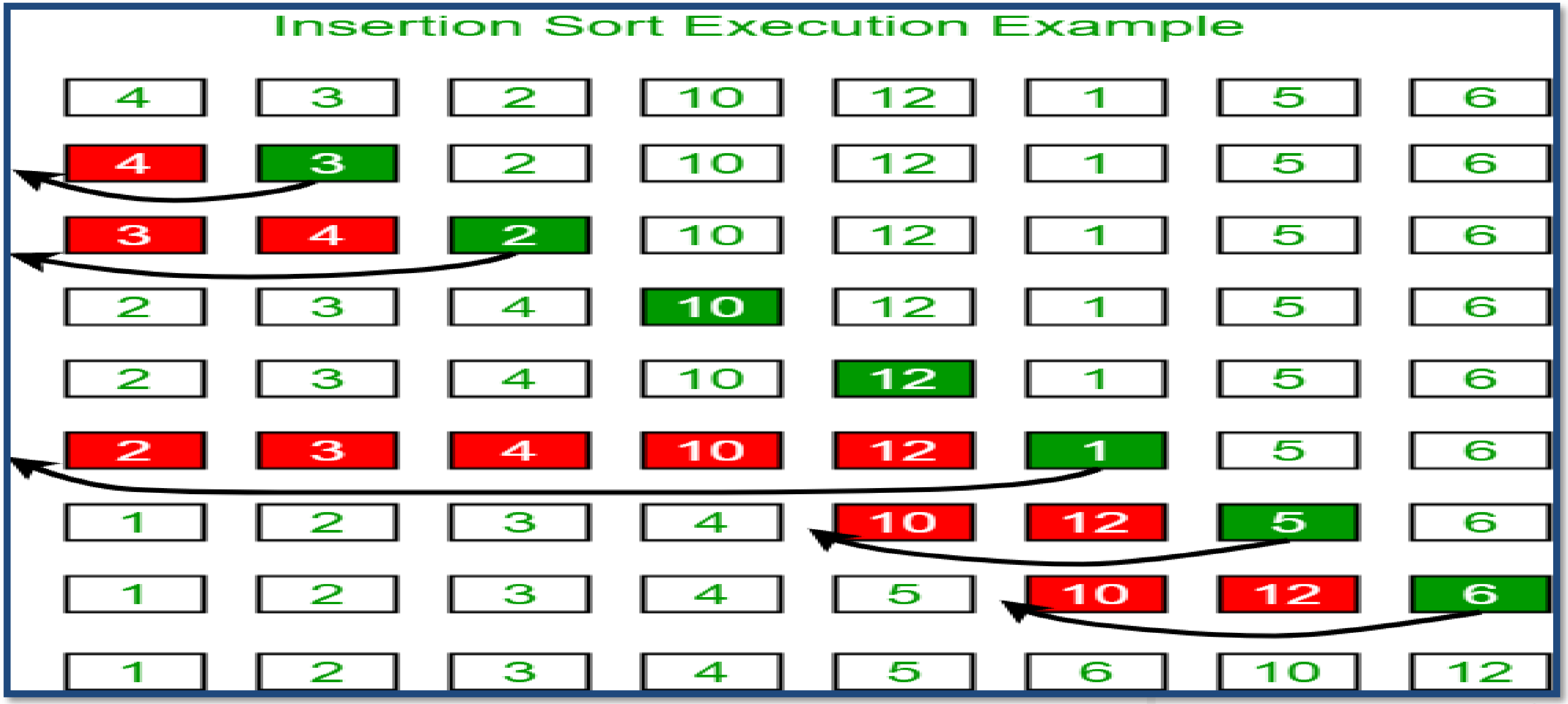
❖A list of *n* elements will take at most *n-1* passes to sort the data.

Algorithms of *Insertion Sort*

❖Algorithms

```
✓Step 1: Repeat Steps 2 to 5 for i = 1 to N-1
✓Step 2: Set Temp = A[i]
✓Step 3: Set J = i - 1
✓Step 4: Repeat while Temp <=A[J]
      Set A[J + 1] = A[J]
      Set J = J - 1
      [End of inner Loop]
✓Step 5: Set A[J + 1] = Temp
      [End of loop]
✓Step 6: Exit
```

Working of Insertion Sort



Insertion Sort Execution Example

4	3	2	10	12	1	5	6
4	3	2	10	12	1	5	6
3	4	2	10	12	1	5	6
2	3	4	10	12	1	5	6
2	3	4	10	12	1	5	6
2	3	4	10	12	1	5	6
1	2	3	4	10	12	5	6
1	2	3	4	5	10	12	6
1	2	3	4	5	6	10	12

Image source : Google

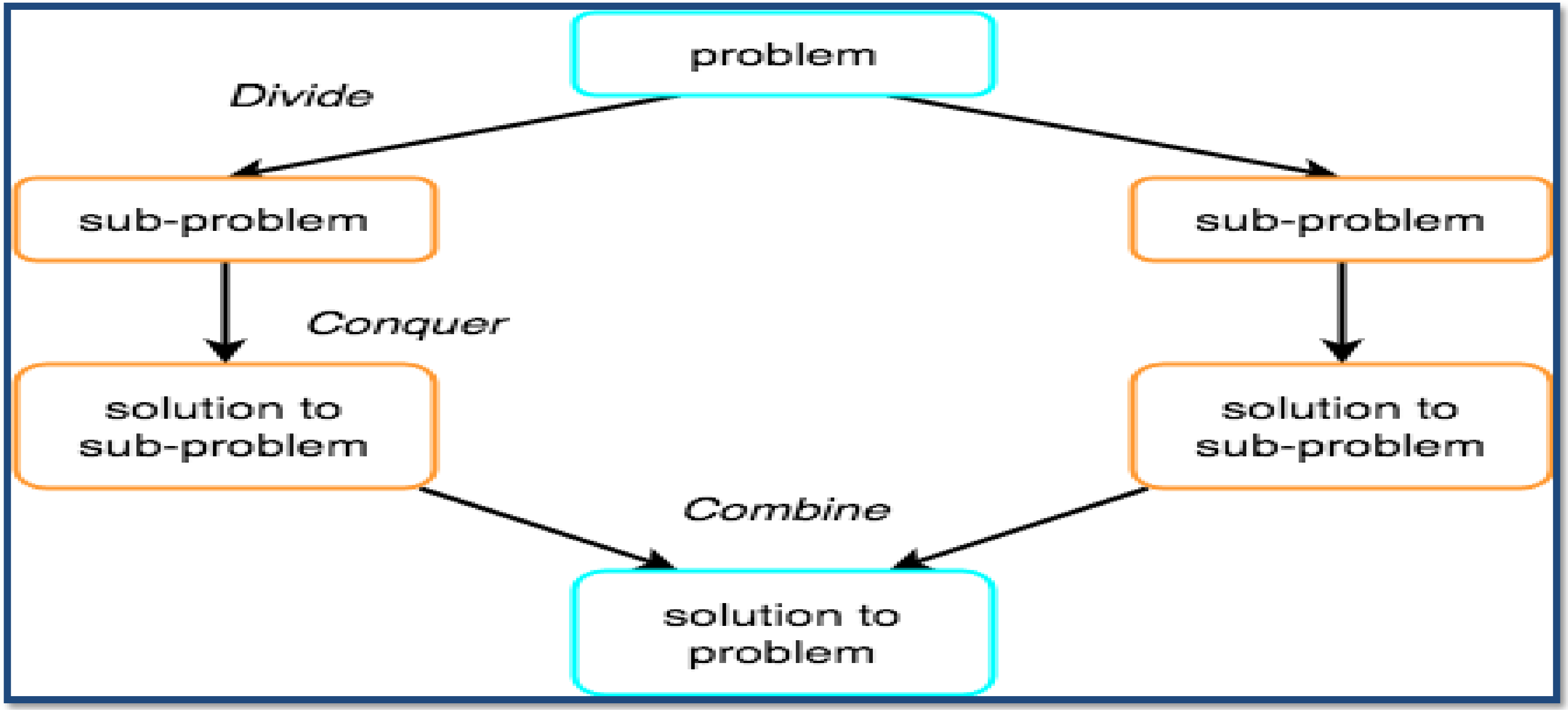
Insertion Sort – Analysis

- ❖ ***Time Complexity***
- ❖ ***Best-case:*** ➔ O(n)
- ❖ ***Worst-case:*** ➔ O(n^2)
- ❖ **Average-case:** ➔ O(n^2)

Merge Sort

- ❖Merge sort is a sorting technique based on divide and conquer technique.
- ❖Before discussing merge sort, we need to understand what is the meaning of splitting and merging
- ❖What do you mean by Divide and Conquer Technique? .

Divide and Conquer Technique


problem

Divide

sub-problem

sub-problem

Conquer

solution to sub-problem

solution to sub-problem

Combine

solution to problem

Image source : Google

Algorithms of *Merge Sort*

❖**Algorithms**

✓**Step 1: [check for recursive call]**

```
if b<e then mid=(b+e)/2
mergesort(a, b, mid)
mergesort(a,mid+1,e)
merge(a,b,mid,e)
end if
```

✓**Step 2: [Finish]**

Merging Procedure Algorithms

❖This algorithm merge two sorted array of size n/2 in to one sorted array a of size n.

✓**Merge Algorithms**

✓**Step 1:[Initialization]**

✓i=b, temp[100], i1=b, e1=mid, i2=mid+1,e2=e;

✓**Step 2:[Merge until all elements of one array got merged]**

✓Repeat step 3 while (i1<= e1 &&i2<=e2)

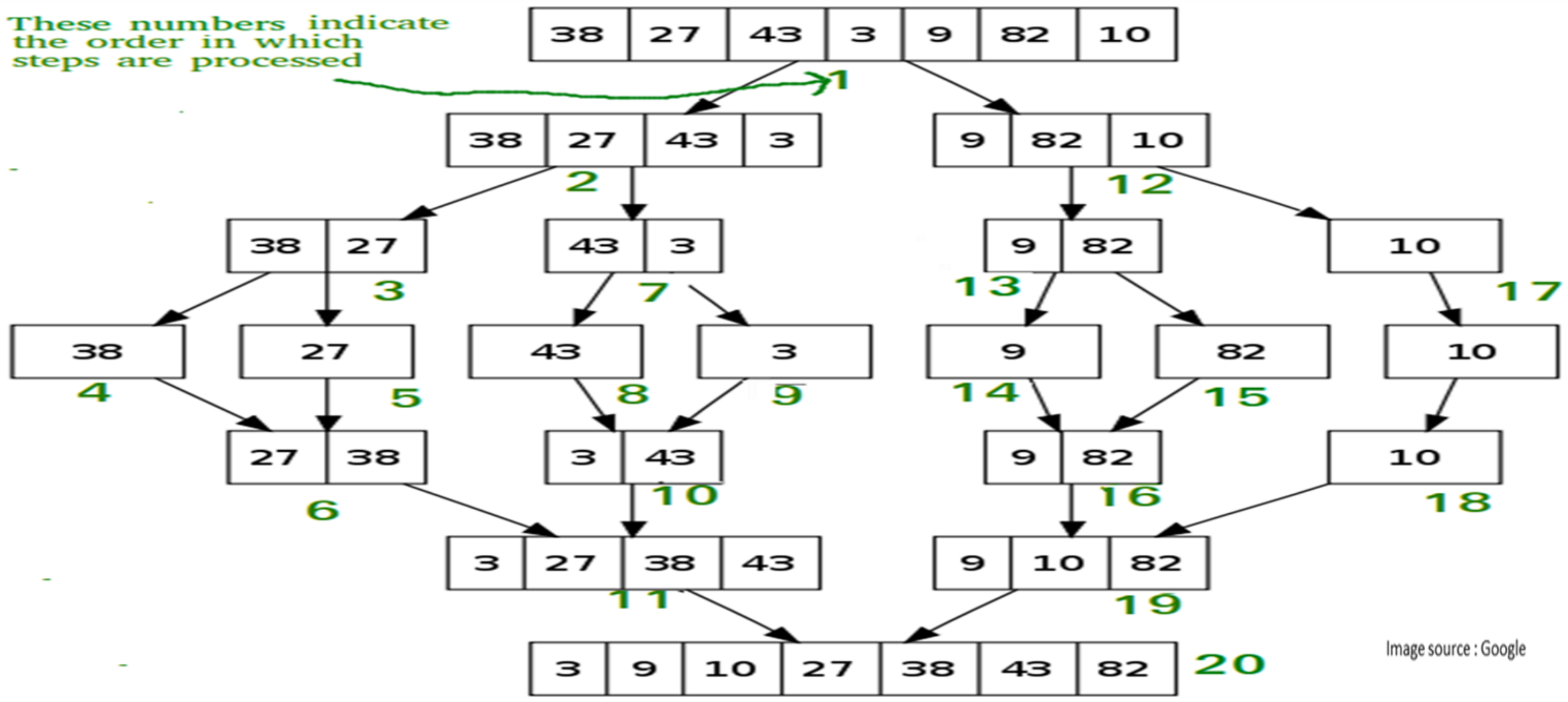
Merging Procedure Algorithms

- **Step 3:[Merging of sub arrays]**
- if a[i1]< a[i2] then temp[i++]=a[i1] i1++
- else
- temp[i++]=a[i2]
- i2++
- end if
- **Step 4:[Copy all elements from first sub array]**
- repeat step 5 while(i1<=e1)
- **step 5: [Copy element from first sub array and prepare for next element]**
- temp[i]=a[i1]
- i++
- i1++

Merging Procedure Algorithms

- ✓ **Step 6: [Copy all elements from second sub array]**
- ✓ repeat step 5 while(i2<=e2)
- ✓ **Step 7: [copy element from second sub array and prepare for next element]**
- ✓ temp[i]=a[i2];
- ✓ i++
- ✓ i1++
- ✓ **Step 8:[Copy temporary temp array to original array a]**
- ✓ for(i=b;i<=e;i++)
- ✓ a[i]=temp[i];
- ✓ **Step 9: [finish]**

Working of Merge sort



These numbers indicate the order in which steps are processed

Image source : Google

Analysis of merge sort

❖ Time complexity of merge sort :

✓ Best Case: **O (n * log2n)**

✓ Average Case: **O (n * log2n)**

✓ Worst Case: **O (n * log2n)**

Quick Sort

❖ Like merge sort, Quicksort is also based on the *divide-and-conquer* paradigm.

❖But it uses this technique in a somewhat opposite manner, as all the hard work is done *before* the recursive calls.

❖The quick-sort algorithm consists of the following three steps

Divide: Partition the list.

Recursion: Recursively sort the sub lists separately.

Conquer: Put the sorted sub lists together

Partition Procedure

❖ Partition means places the pivot in its correct place position within the array.

❖Arranging the array elements around the pivot p generates two smaller sorting problems

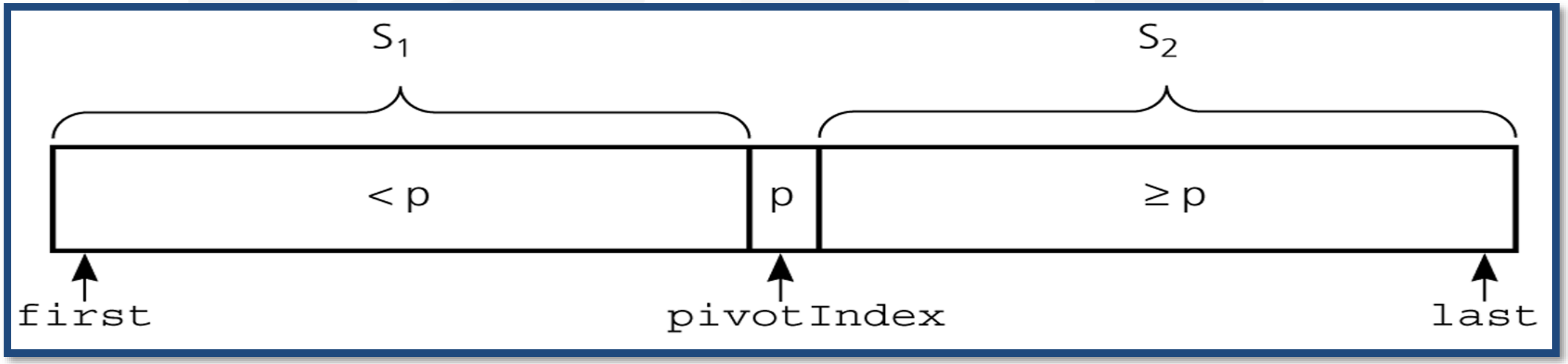


Image source : Google

Partition Algorithms

❖This algorithm returns pivot point.

✓ **Step 1:[Initialization]**

✓ p=b, i=b, J=e,

✓ **Step 2:[Find partition point]**

✓ Repeat steps 3,4and 5 while(i<j)

✓ **Step 3[Check for number greater than pivot point from left]**

✓ while (a[i]<=a[p])

✓   i++;

✓   end While

✓ **Step 4 [check for number less than pivot point from right]**

✓ while (a[J]>a[p])

✓ J--;

✓ end while

Partition Algorithms

- ✓ **Step 5 [Move element in respective sub array]**
- ✓ if(i<j) then
- ✓ swap(a[i], a[j])
- ✓ end if
- ✓ **Step 6[Move pivot at its position]**
- ✓ swap(a[j], a[p])
- ✓ **Step 7 [return partition point]**
- ✓ return j;
- ✓ **Step 8[Finish]**

Quicksort Algorithms

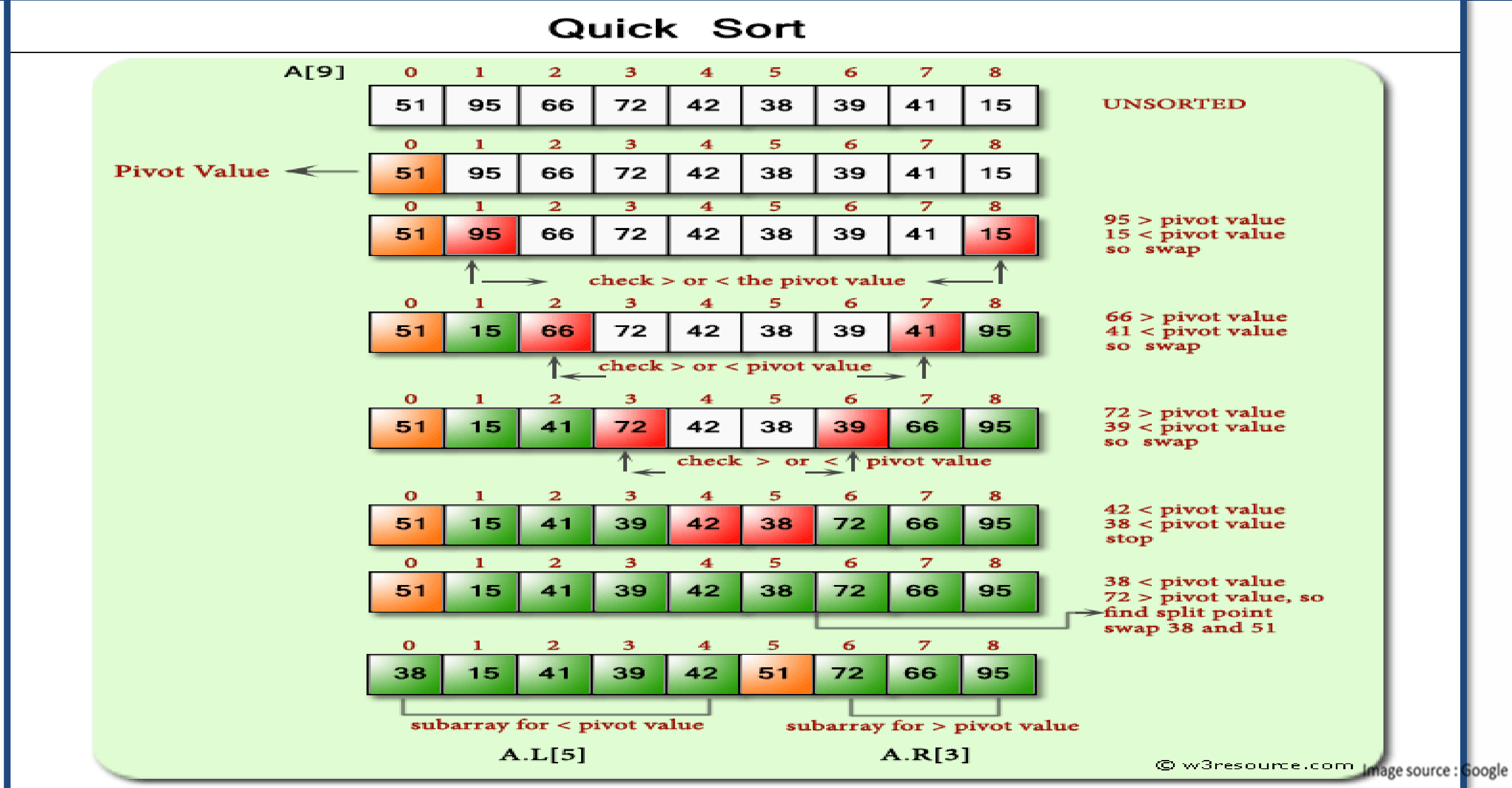
- This algorithm sort unsorted array.

Step 1:[check for recursive call]

```
    If (b<e) then        //b=star index ,e=end index of array
    p=partition (a, b, e)
    Quicksort (a, b, p)
    Quicksort (a, p+1,e)
  End if
```

Step 2: [Finish]

Working of Quick Sort



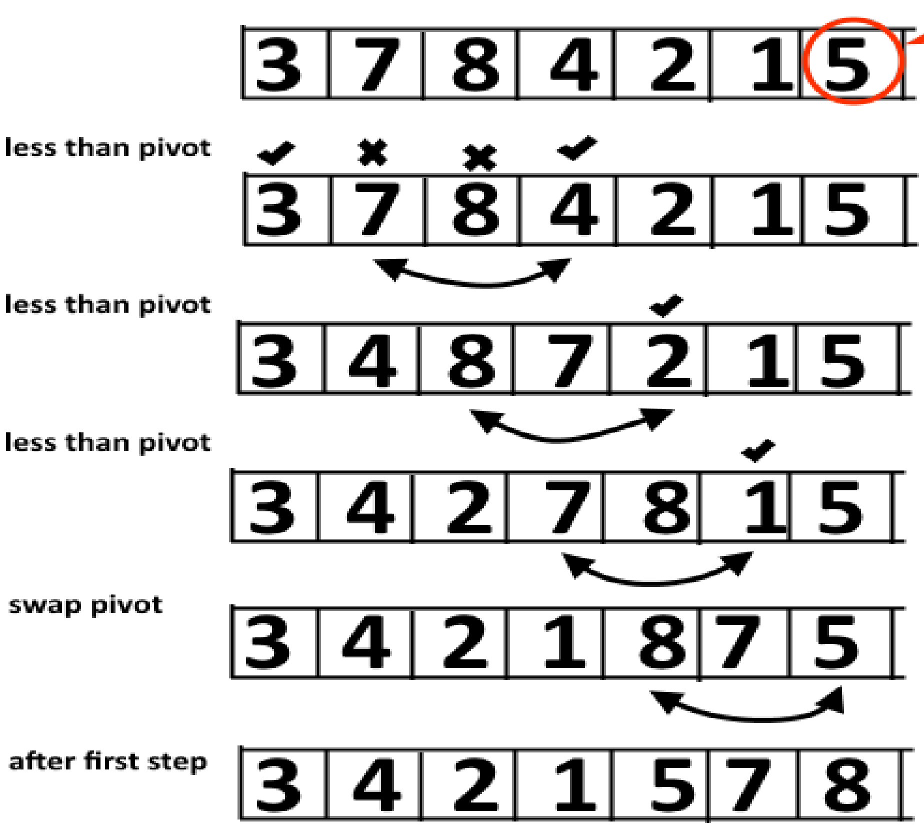
Quick Sort

A[9]	0	1	2	3	4	5	6	7	8	
	51	95	66	72	42	38	39	41	15	UNSORTED
Pivot Value ←	51	95	66	72	42	38	39	41	15	
	51	95	66	72	42	38	39	41	15	95 > pivot value 15 < pivot value so swap
	51	15	66	72	42	38	39	41	95	66 > pivot value 41 < pivot value so swap
	51	15	41	72	42	38	39	66	95	72 > pivot value 39 < pivot value so swap
	51	15	41	39	42	38	72	66	95	42 < pivot value 38 < pivot value stop
	51	15	41	39	42	38	72	66	95	38 < pivot value 72 > pivot value, so find split point swap 38 and 51
	38	15	41	39	42	51	72	66	95	

check > or < the pivot value
check > or < pivot value
check > or < pivot value
subarray for < pivot value
A.L[5]
subarray for > pivot value
A.R[3]
© w3resource.com

Image source : Google

Working of Quick Sort



3 7 8 4 2 1 5

Pivot

Iniitial array

less than pivot

3 7 8 4 2 1 5

3 is smaller than pivot we swap i =0, paritionIndex = 0 Hence stays in the same position but partitionIndex becomes 1. Then i = 1, i =2 is bigger we dont do anything. i = 3 is smaller hence, swap between i =3 with paritionIndex =1.After that partitionIndex =2

less than pivot

3 4 8 7 2 1 5

In i =4 is smalle rthan pivot. we swap i =4 and partitionIndex =2 and increase partitionIndex = 3

less than pivot

3 4 2 7 8 1 5

In I=5 also smaller than pivot. we swap i=5 and partitionIndex = 3. This is the elements before the last one. hence we stop looping.

swap pivot

3 4 2 1 8 7 5

After loping swap the last item with paritionIndex so that the pivot stays in the right position.

after first step

3 4 2 1 5 7 8

Everything before pivot is smaller than the pivot and in the right everything is bigger than pivot

Image source : Google

Quick Sort Analysis

Time Complexity

- ❖Best Case : $\Theta(nlog2n)$
- ❖Average Case: $\Theta(nlog2n)$
- ❖Worst Case : $O(n2)$

Heap Sort

- ❖ Heap is a special tree-based data structure. A binary tree is said to follow a heap data structure if
- ✓ it is a complete binary tree
- ✓ it should follow the properties of either max-heap or Min-heap
- ❖ What is Max or Min Heap ?.

Max Heap and Min Heap

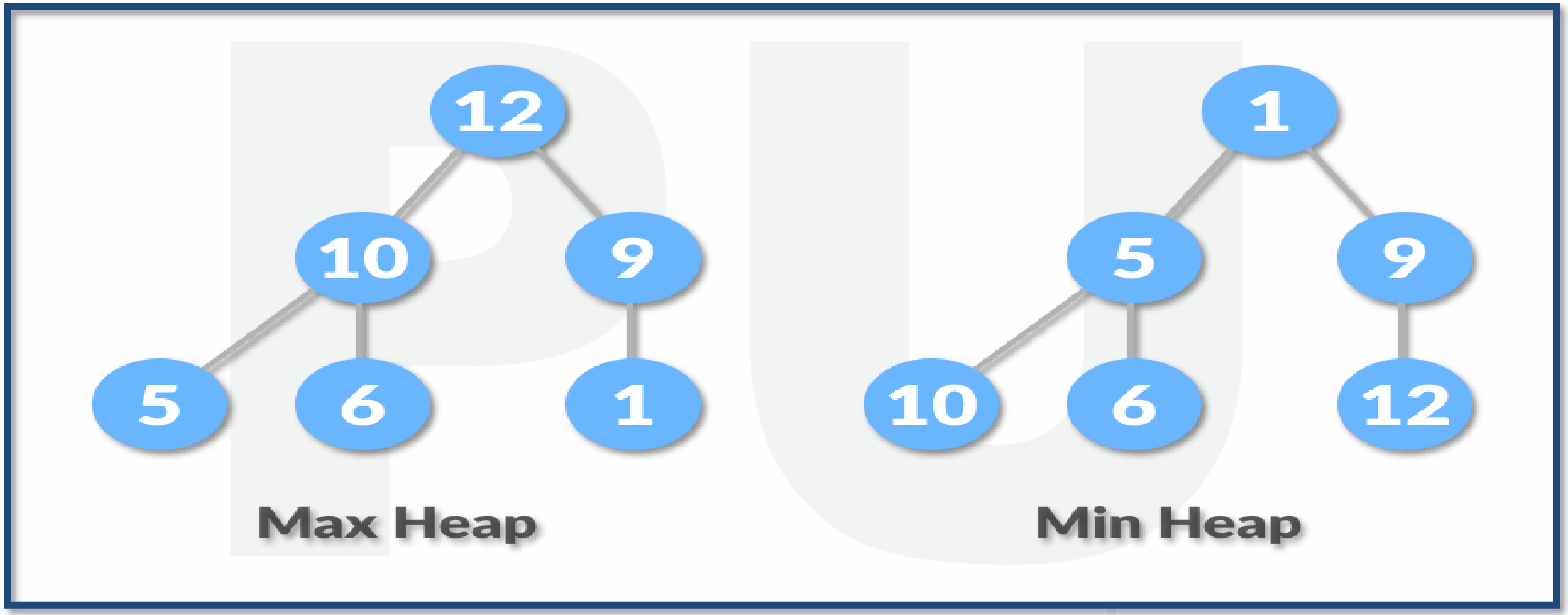


Image source : Google

Heapify Algorithms

- Purpose of heapify algorithms to arrange the complete binary tree in the form of max-heap or Min -heap

How to build Max Heap

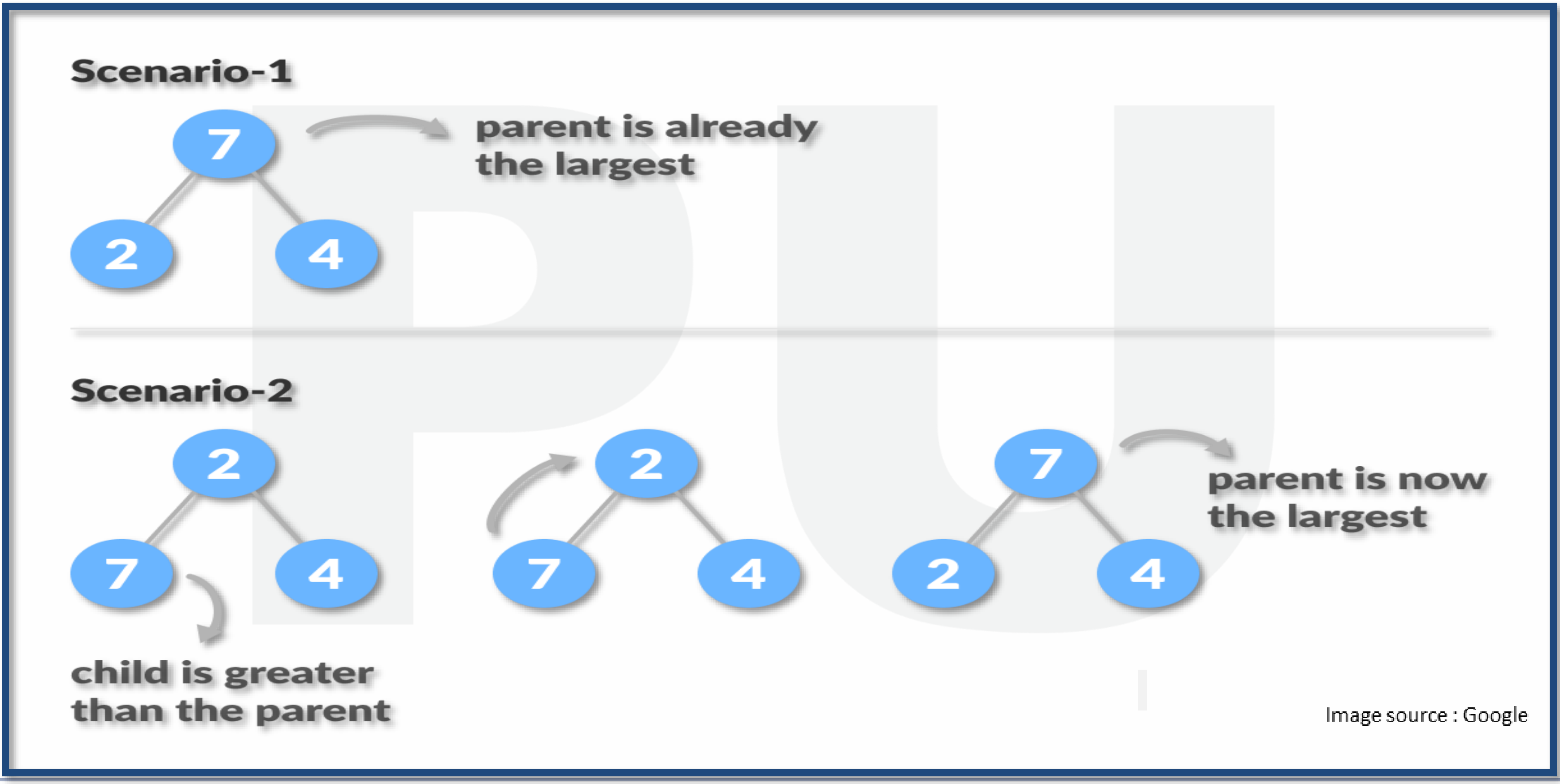

Scenario-1
7
2
4
parent is already the largest
Scenario-2
2
7
4
child is greater than the parent
2
7
4
7
2
4
parent is now the largest

Image source : Google

How to build Max Heap

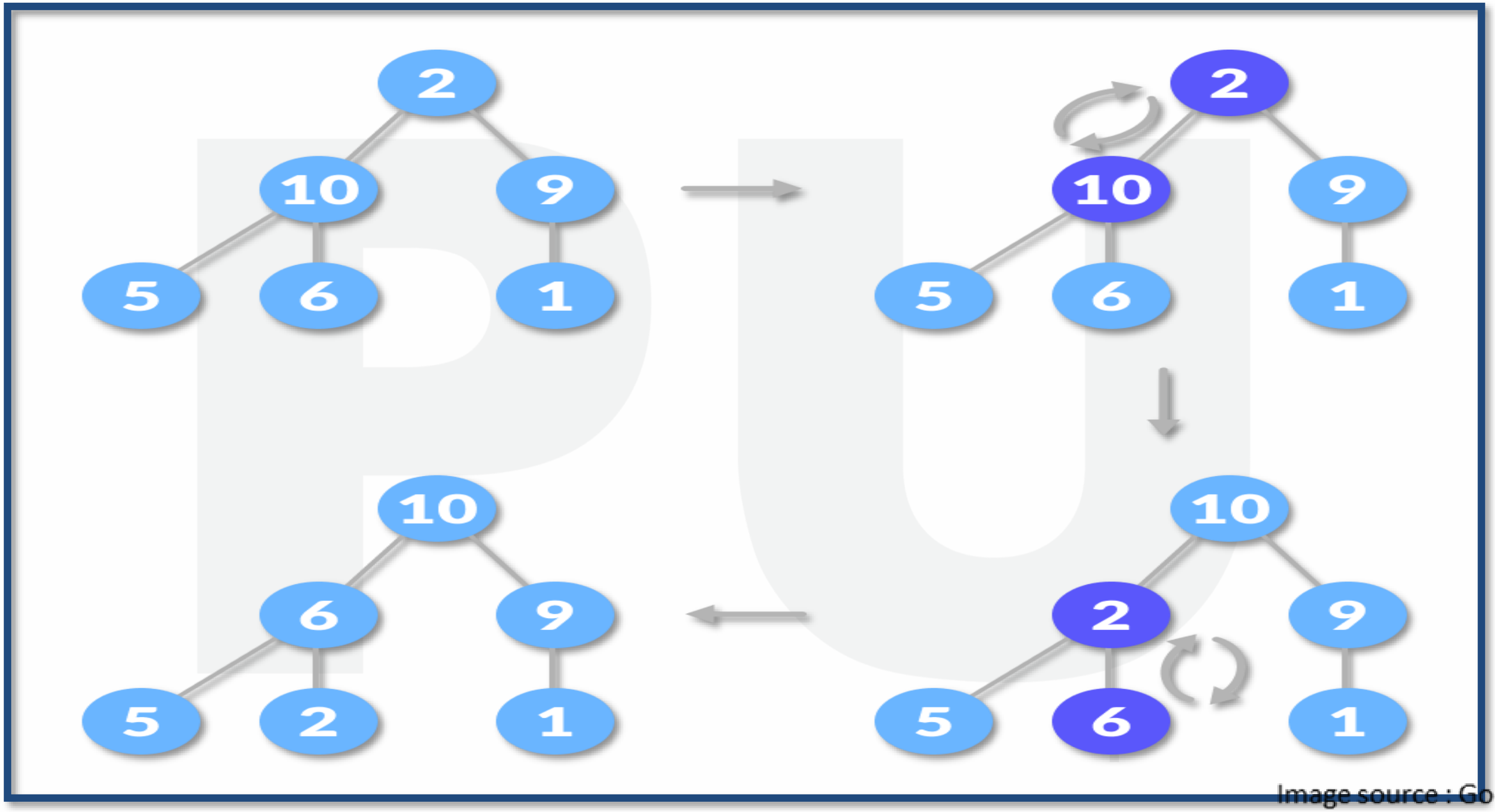


Image source : Google

Heap Sort

How does Heap Sort Works?

✓Since the tree satisfies Max-Heap property, then the largest item is stored at the root node.

✓**Swap:** Remove the root element and put at the end of the array (nth position) Put the last item of the tree (heap) at the vacant place.

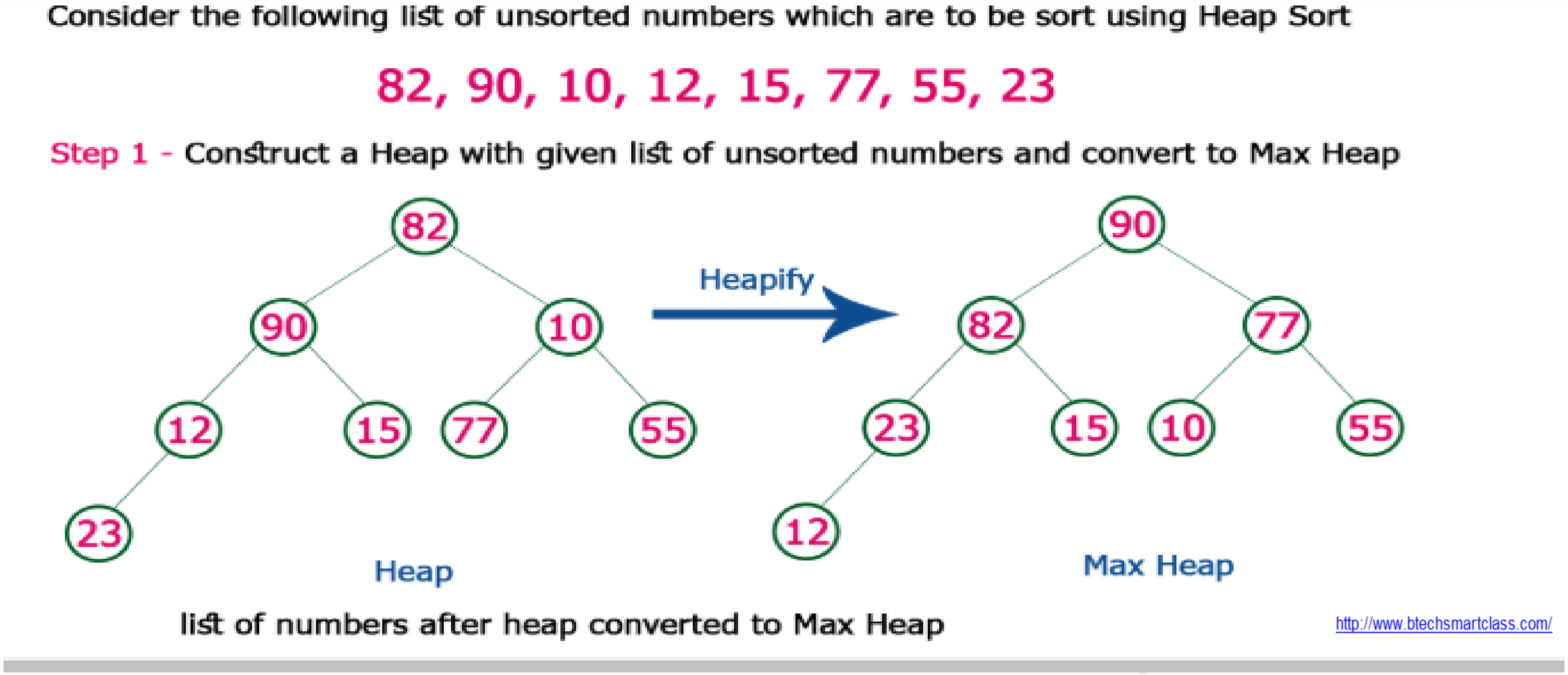
✓**Remove:** Reduce the size of the heap by 1.

✓**Heapify:** Heapify the root element again so that we have the highest element at root.

✓The process is repeated until all the items of the list are sorted.

Image source : Google

Heap Sort Procedure



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

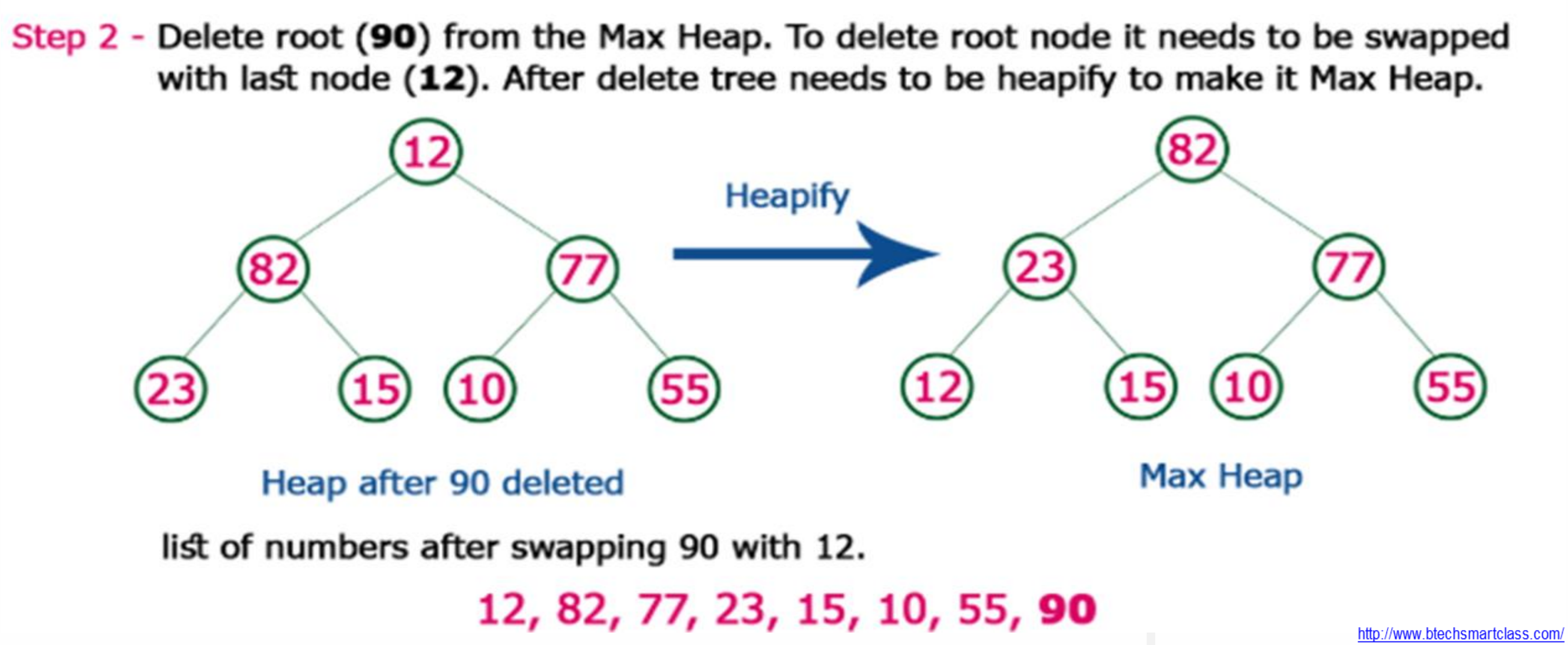
82, 90, 10, 12, 15, 77, 55, 23

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

http://www.btechsmartclass.com/

Heap Sort Procedure



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.

Heap after 90 deleted

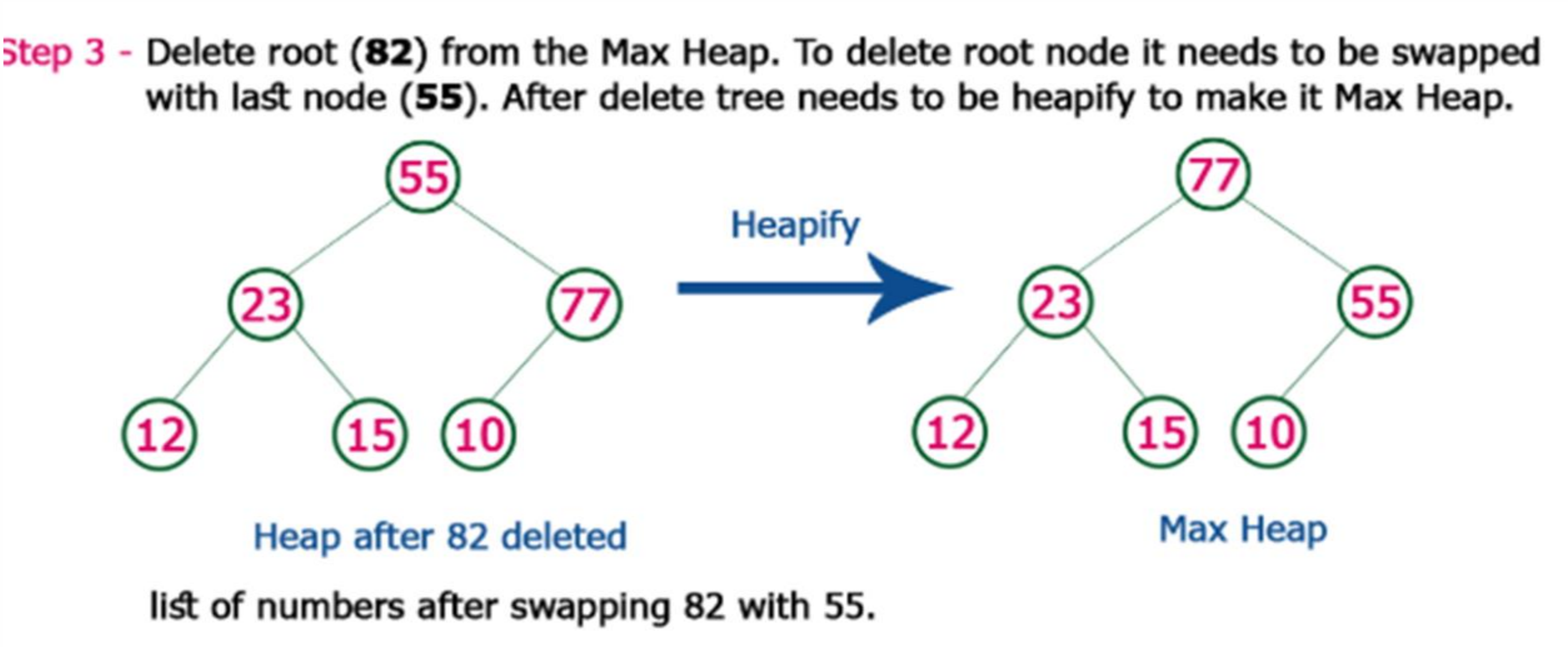
Max Heap

list of numbers after swapping 90 with 12.

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

http://www.btechsmartclass.com/

Heap Sort Procedure



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.

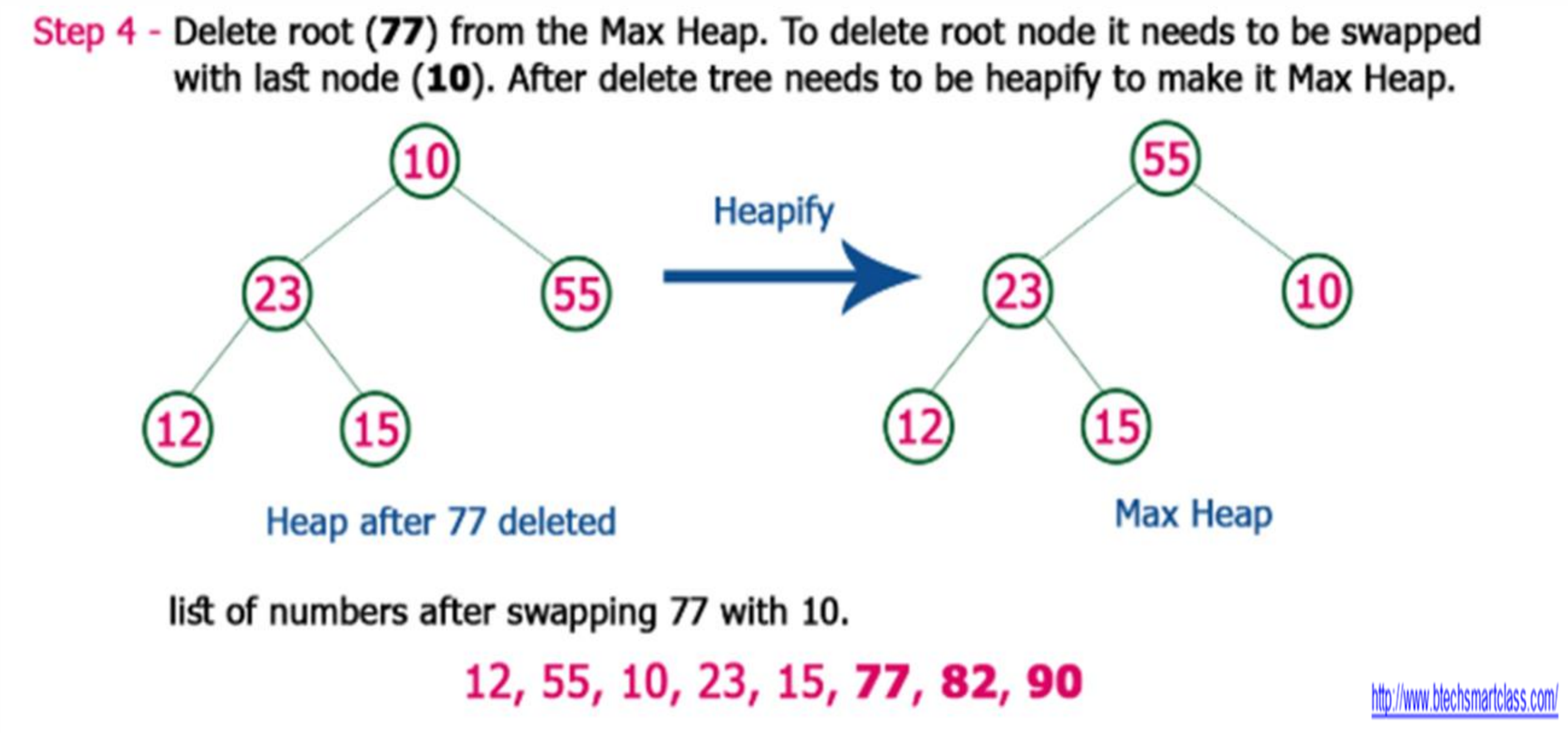
Heap after 82 deleted

Max Heap

list of numbers after swapping 82 with 55.

http://www.btechsmartclass.com/

Heap Sort Procedure



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

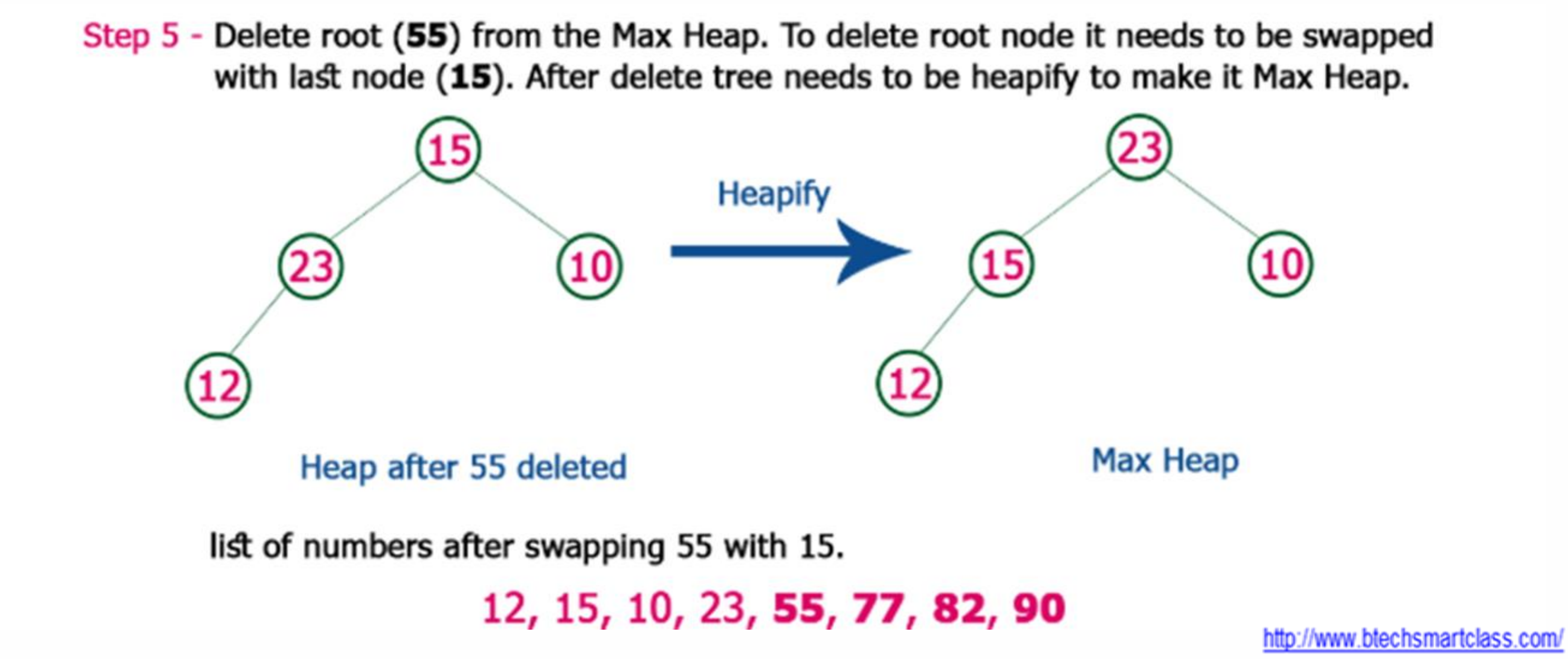
Max Heap

list of numbers after swapping 77 with 10.

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

http://www.btechsmartclass.com/

Heap Sort Procedure



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

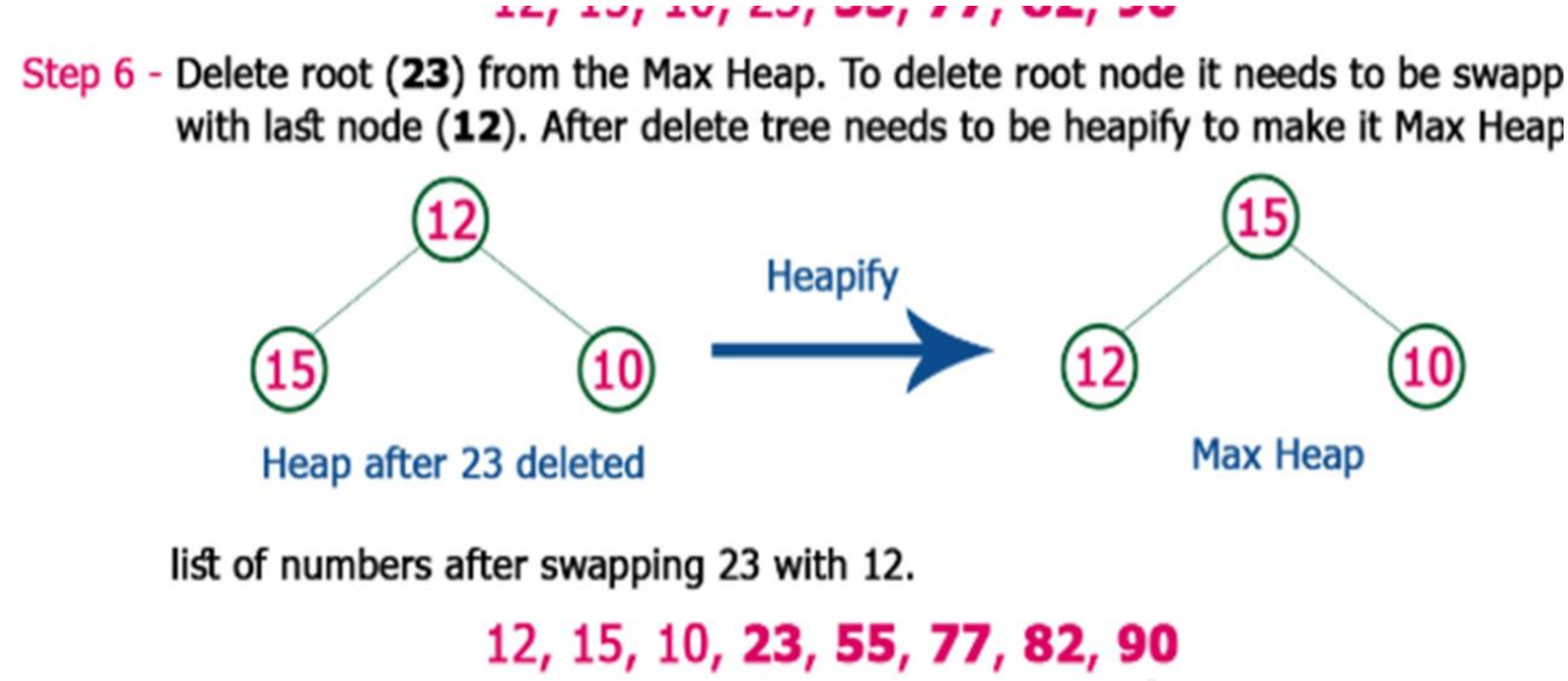
Max Heap

list of numbers after swapping 55 with 15.

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

http://www.btechsmartclass.com/

Heap Sort Procedure



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 swapp with last node (**12**). After delete tree needs to be heapify to make it Max Heap

Heap after 23 deleted

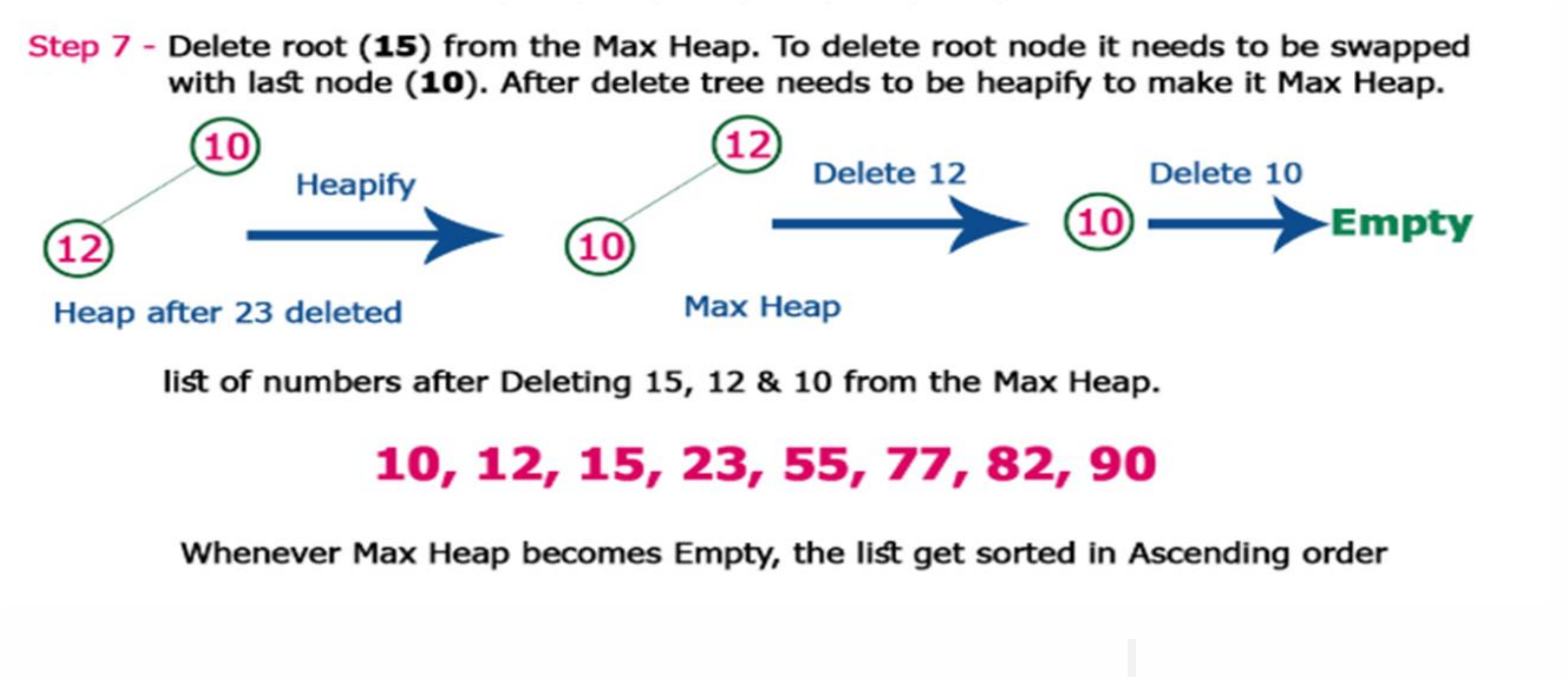
Max Heap

list of numbers after swapping 23 with 12.

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

http://www.btechsmartclass.com/

Heap Sort Procedure



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.

Heap after 23 deleted

Max Heap

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

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

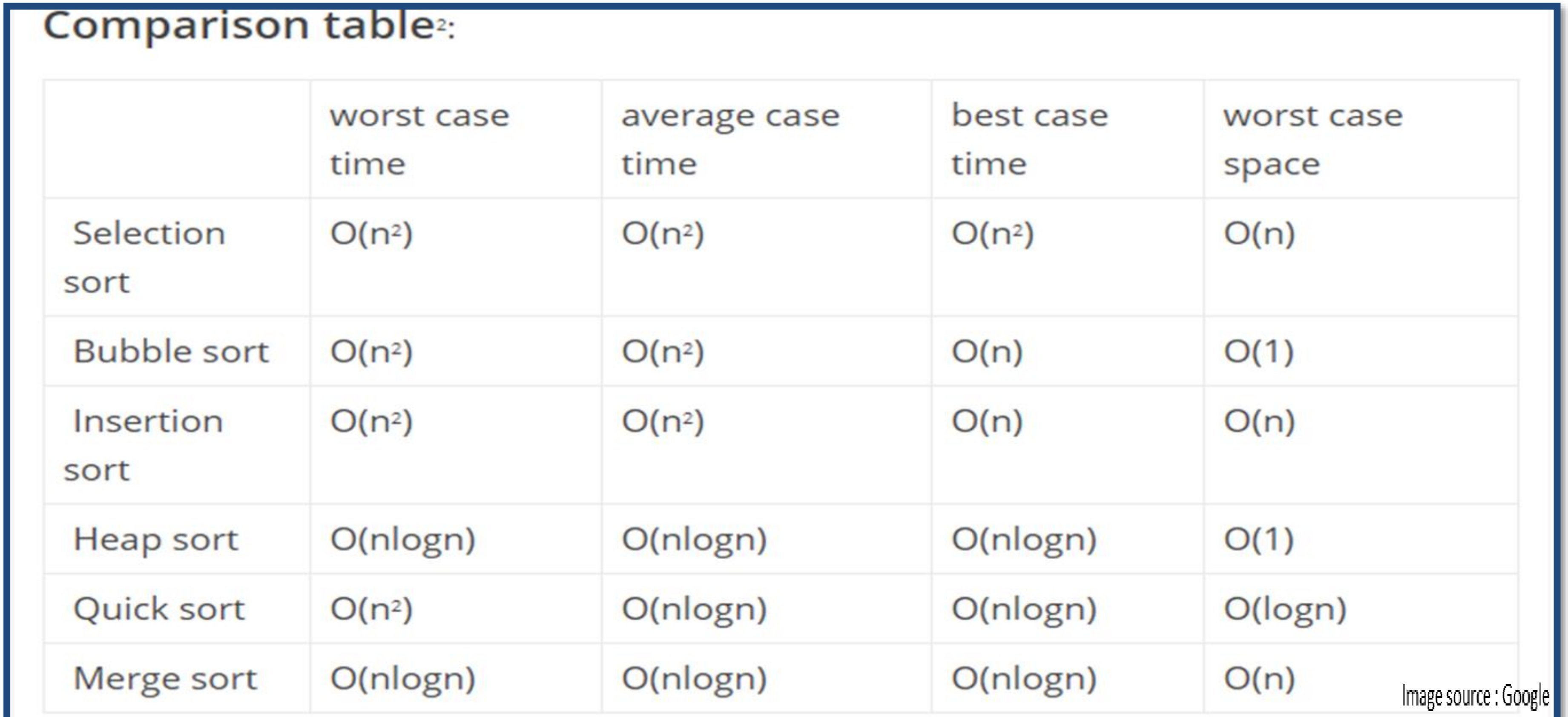
Whenever Max Heap becomes Empty, the list get sorted in Ascending order

http://www.btechsmartclass.com/

Analysis of Heap Sort

- ✓ **Time Complexity:** Time complexity of heapify is O(Logn). Time complexity of createAndBuildHeap() is O(n) and overall time complexity of Heap Sort is O(nLogn).
- ✓ Best Case: O(nLogn)
- ✓ Average Case: O(nLogn)
- ✓ Worst Case: O(nLogn

Performance of Various Sorting Algorithms



Comparison table[2]:

	worst case time	average case time	best case time	worst case space
Selection sort	$O(n^2)$	$O(n^2)$	$O(n^2)$	O(n)
Bubble sort	$O(n^2)$	$O(n^2)$	O(n)	O(1)
Insertion sort	$O(n^2)$	$O(n^2)$	O(n)	O(n)
Heap sort	O(nlogn)	O(nlogn)	O(nlogn)	O(1)
Quick sort	$O(n^2)$	O(nlogn)	O(nlogn)	O(logn)
Merge sort	O(nlogn)	O(nlogn)	O(nlogn)	O(n)

Image source : Google

Topic-2

Hashing

Introduction

- In CS, a **hash table**, or a **hash map**, is a data structure that associates keys (names) with values (attributes).

  – Look-Up Table

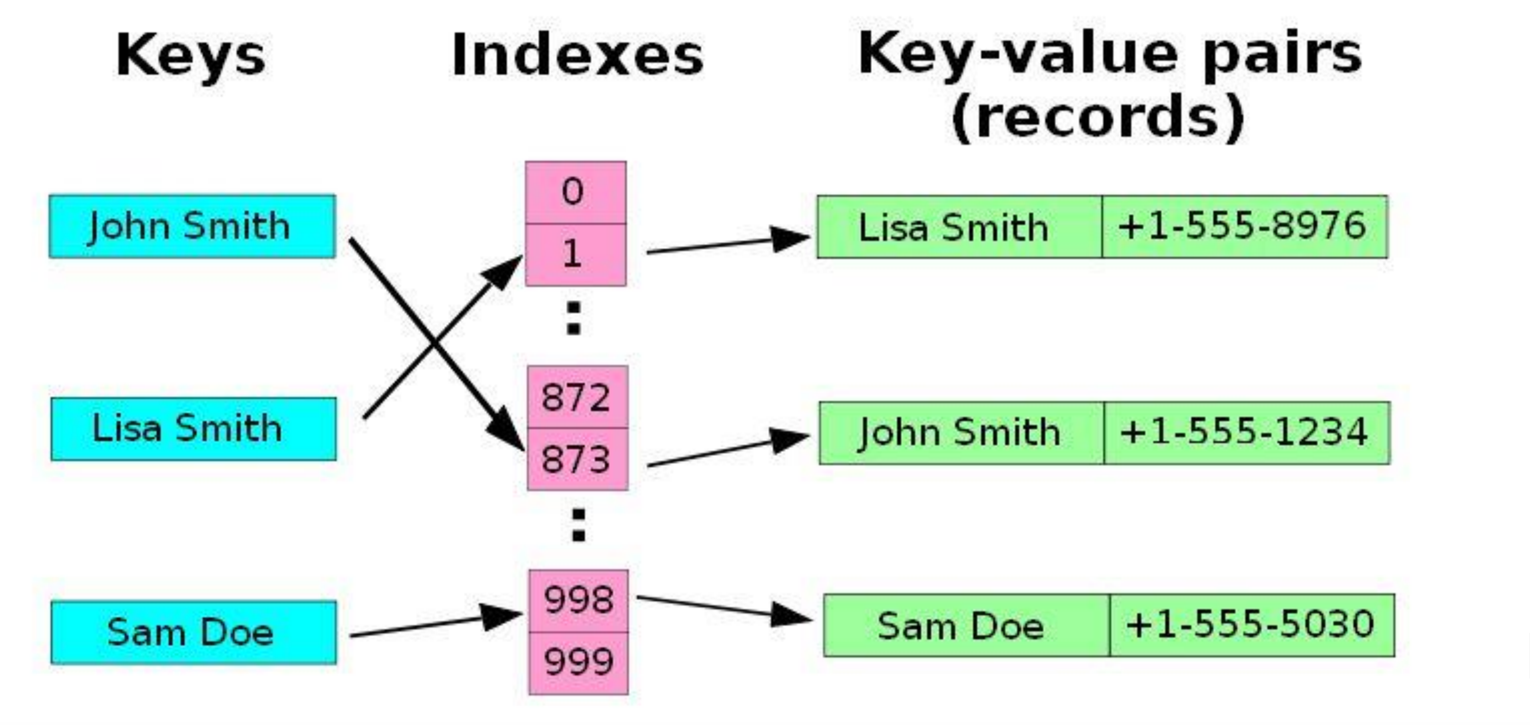
  – Dictionary

  – Cache

  – Extended Array

Introduction

Example: A small phone book as a hash table.



Source: Wikipedia

Hash Function

- It is the mathematical function which takes unique key as an argument and return a unique memory location which is used to store and retrieve the data related to key.

- Ex. m = H(key)
  - Where, **m** is memory location
  - If the key is 3205 and if we are using mid-square method as hash function them m=72. Which means, we are 72th location of memory to store and retrieve data related to key 3205.

Hash Function

- Some of the hash functions are:
  - Mid-Square method
  - Division method
  - Folding method

Hash Function - Mid-Square

- In this method, first we square the given key, then we find out the mid of this squared key by first putting left & then right, moving to mid.

- Ex. Key = 3205
  - Step 1: (key)^2 = (3205)^2 = 10272025
  - Step 2: deleting from left then right, moving towards mid.

    therefore, m = 72

Hash Function - Division

- Using the formula m = key mod D, we find out the value of m to retrieve the required data.
- D is the largest prime no. between lower and upper limit of memory.

- Ex. Key = 3205

  m = key % D,

  suppose lower limit = 0 and upper limit 99 then D will be 97

  Therefore, m = 3205 % 97 = 04

Hash Function - Folding

- We fold the given no. into the digits which will be equal to the digits in upper limit and the we add them.
- Ex 1: key = 3205
  - Step 1: folding -> 32 and 05
  - Step 2: adding -> 37 is the required m
- Ex 2: key = 2209
  - Step 1: folding -> 22 and 09
  - Step 2: adding -> 31 is the required m

Hash Function - Folding : Collision

- Ex 3: key = 3205
  - Step 1: folding -> 32 and 05
  - Step 2: adding -> 37 is the required m

- Ex 4: key = 3106
  - Step 1: folding -> 31 and 06
  - Step 2: adding -> 37 is the required m

- The situation arises in above 2 example is known as collision. Which means for the different keys we are getting same memory address.

Collision Resolution Technique

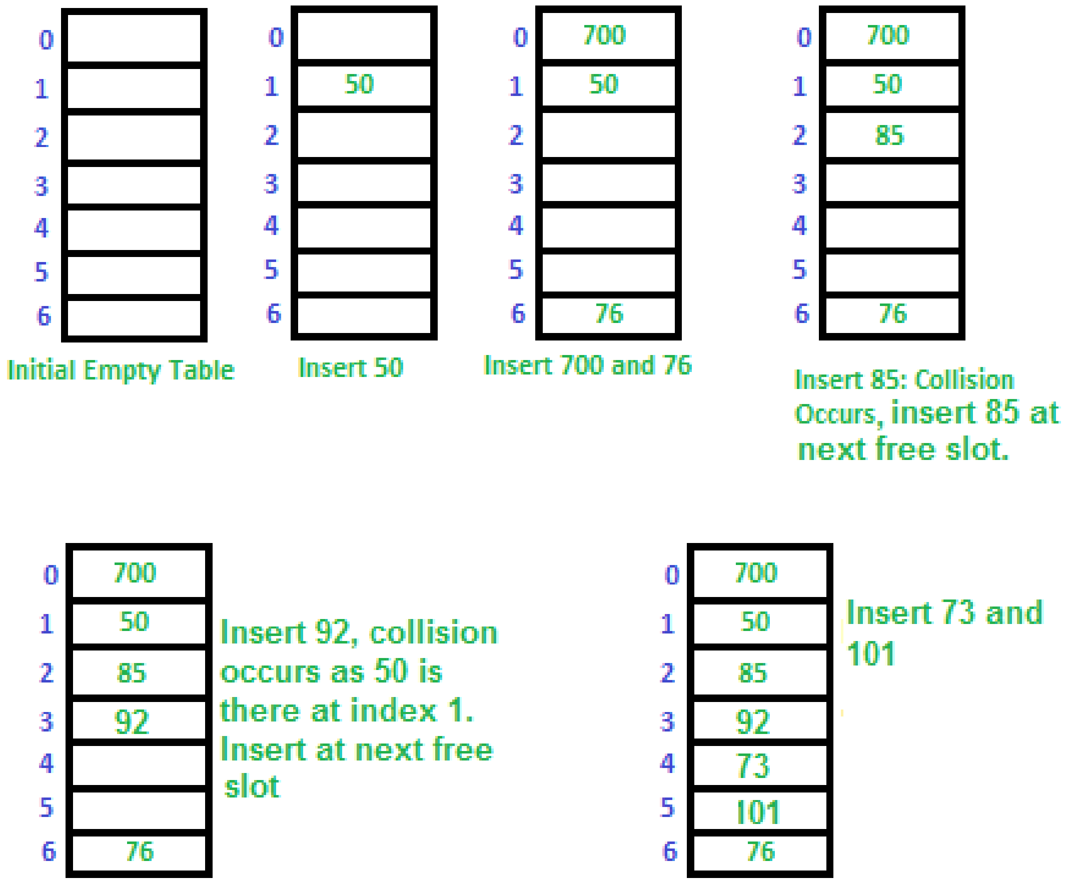
The following techniques are used to deal with the collision problem:

1. Probing
   a) Linear
   b) Quadratic
2. Rehashing
3. Chaining

Probing - Linear

- In linear probing, we linearly probe for next slot.
- We find next empty location as:
  - M = (p + i) % SIZE
  - Ex. Let us consider a simple hash function as “key mod 7” and sequence of keys as 50, 700, 76, 85, 92, 73, 101.

Probing - Linear



	Initial Empty Table	Insert 50	Insert 700 and 76	Insert 85: Collision Occurs, insert 85 at next free slot.	Insert 92, collision occurs as 50 is there at index 1. Insert at next free slot	Insert 73 and 101
0			700	700	700	700
1		50	50	50	50	50
2				85	85	85
3					92	92
4						73
5						101
6			76	76	76	76

Initial Empty Table

Insert 50

Insert 700 and 76

Insert 85: Collision Occurs, insert 85 at next free slot.

Insert 92, collision occurs as 50 is there at index 1. Insert at next free slot

Insert 73 and 101

Image source: google

Probing - Quadratic

- In this method, we find next empty location as:
  - M = (p + i^2) % SIZE
- Rest process is same as linear probing.

Rehashing

- Also known as Double hashing.
- In this method to search the next empty location, we apply different hash functions in a particular order to store and retrieve colliding keys.
- Double hashing : F(i) = i * Hash2(X)

Rehashing

- Ex.
  - let hash(x) be the slot index computed using hash function.
  - If slot hash(x) % S is full, then we try (hash(x) + 1*hash2(x)) % S
  - If (hash(x) + 1*hash2(x)) % S is also full, then we try (hash(x) + 2*hash2(x)) % S
  - If (hash(x) + 2*hash2(x)) % S is also full, then we try (hash(x) + 3*hash2(x)) % S

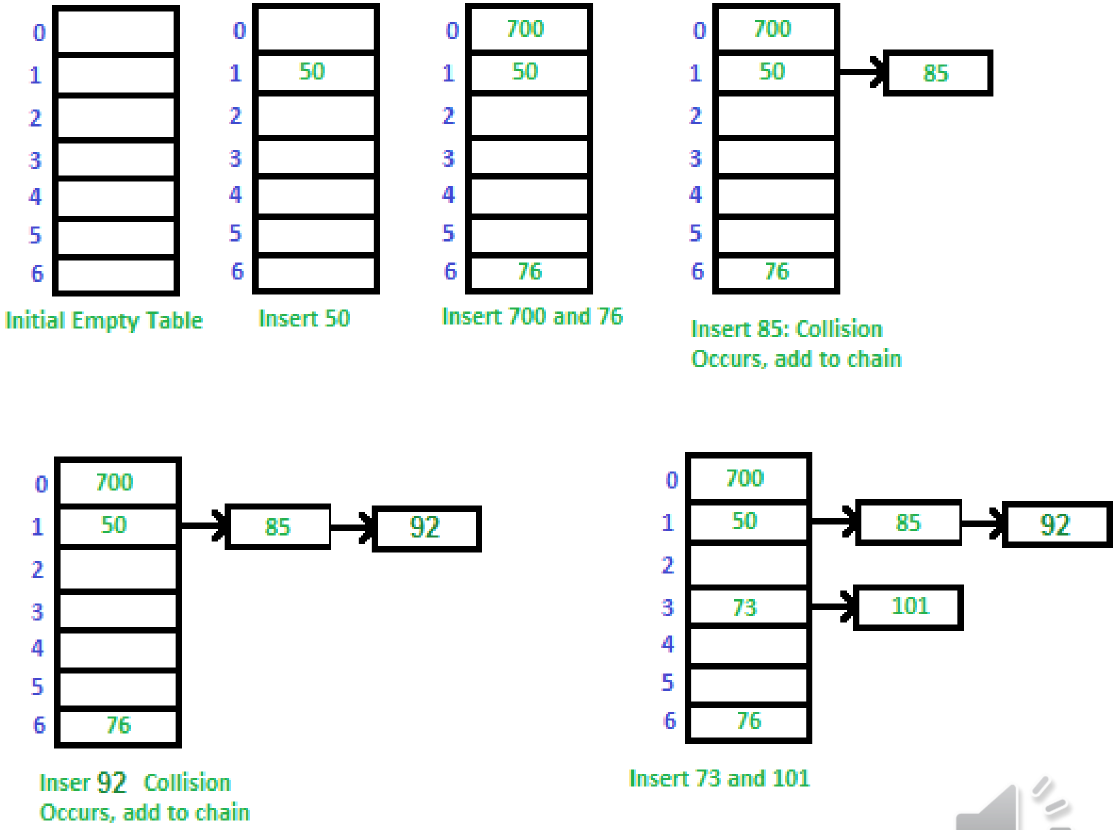
Chaining

- Chaining is also known as Bucketing

- In this we use a separate data structure called as linked list (Chain) of colliding keys. As collision occurs we add related colliding key at the end of link list of colliding frame.

Chaining

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



Initial Empty Table

Insert 50

Insert 700 and 76

Insert 85: Collision Occurs, add to chain

Inser 92 Collision Occurs, add to chain

Insert 73 and 101

Image source: google

DIGITAL LEARNING CONTENT

Parul® University

www.paruluniversity.ac.in