Analysis of Algorithm

Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk WhatsApp# 0346-5100010

(Week 08) Lectures 15 & 16

Objectives: Learning objectives of these lectures are

- What is Count Sort?
- What is Linear Sorting
- Count Sort Algorithm
- What is Bucket Sort?
- Bucket Sort Algorithm
- Bucket Sort Algorithm's Time Complexity
- What is Radix Sort
- Radix Sort Algorithm
- Radix Sort Algorithm's Time Complexity
- Heap Sort
- Heap Sort Complexity Analysis

Text Book & Resources:

1. Introduction to Algorithms by Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest and Clifford Stein, The MIT Press; 3rdEdition (2009). ISBN-10: 0262033844
2. Introduction to the Design and Analysis of Algorithms by Anany Levitin, Addison Wesley; 2ndEdition (2006). ISBN-10: 0321358287
3. Algorithms in C++ by Robert Sedgewick (1999). ASIN: B006UR4BJS
4. Algorithms in Java by Robert Sedgewick, Addison-Wesley Professional; 3rd Edition(2002). ISBN-10: 0201361205

Analysis of Algorithm

Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk WhatsApp# 0346-5100010

In the last lecture (**Week#10**), we discussed "Divide & Conquer" approach with Quick Sort algorithm and its examples. In this week, we will discuss two more sorting techniques which are Count & Bucket Sort.

❖ Count Sort?

Count Sort is a **Linear Sorting** algorithm which sorts elements in O(n) time. No comparison between elements has been done. The other common linear sorts include Bucket and Radix sorts.

What is Linear Sorting Algorithm?

Linear sorting algorithm is a sorting algorithm that does not use any comparison operator (>, <, >= , <= , ==) to determine the sorting order of the elements. The sorting is achieved by acute logic build, and the overall time taken by the algorithm is hence linear.

Actually, if we contrast linear sort to other comparison sorts with respect to time, we will find that comparison sorts can do **nlogn** at their best and **exponential** at worse in terms of time. The linear sort gives linear performance and thus has fine edge in time over these algorithms.

Count Sort Algorithm Description

Counting sort is an algorithm for sorting a collection of objects according to keys that are small integers; that is, it is an integer sorting algorithm. It is a linear time sorting algorithm used to sort items when they belong to a fixed and finite set.

The algorithm proceeds by defining an ordering relation between the items from which the set to be sorted is derived (for a set of integers, this relation is trivial). Let the set to be sorted be called A. Then, an auxiliary array with size equal to the number of items in the superset is defined, say C. For each element in A, say e, the algorithm stores the number of items in A smaller than or equal to e in C(e). If the sorted set is to be stored in an array B, then for each e in A, taken in reverse order, B[C[e]] = e. After each such step, the value of C(e) is decremented.

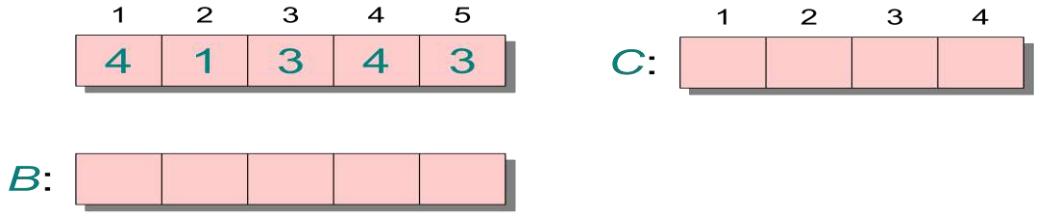
Analysis of Algorithm

Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk WhatsApp# 0346-5100010

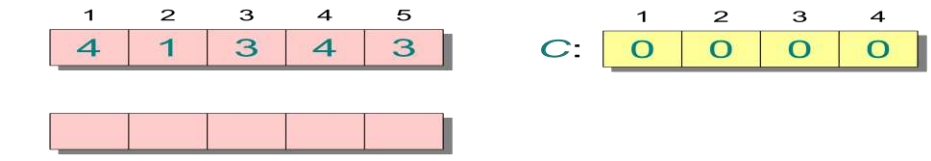
Count Sort Algorithm

```
CountSort(A, B, k)
  {
    for i=1 to k
        C[i]= 0;
    for j=1 to n
        C[A[j]] = C[A[j]] + 1;
    for i=2 to k
        C[i] = C[i] + C[i-1];
    for j=n downto 1
       B[C[A[j]]] = A[j];
       C[A[j]] = C[A[j]] –  1;
  }
```

Example

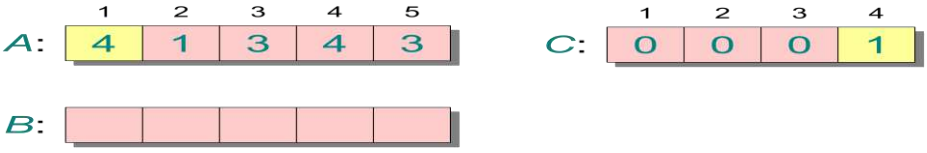


Loop 1

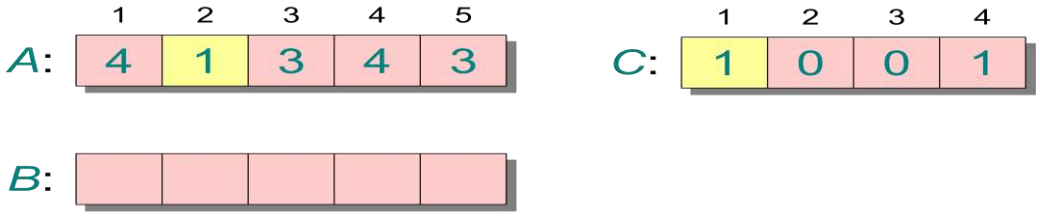


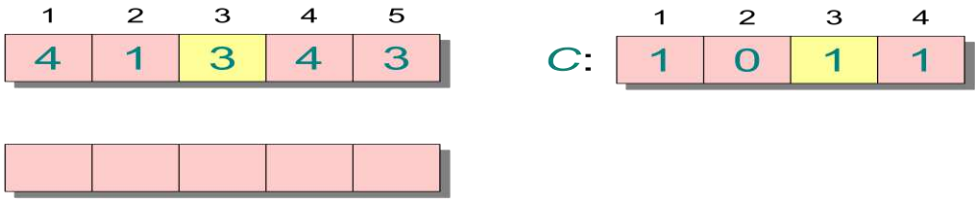

```
for i ← 1 to k
  do C[i] ← 0
```

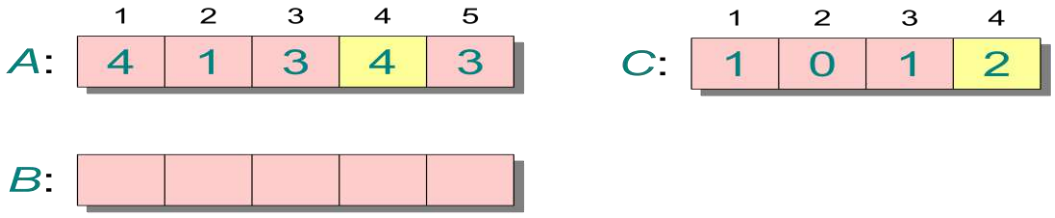
Loop 2

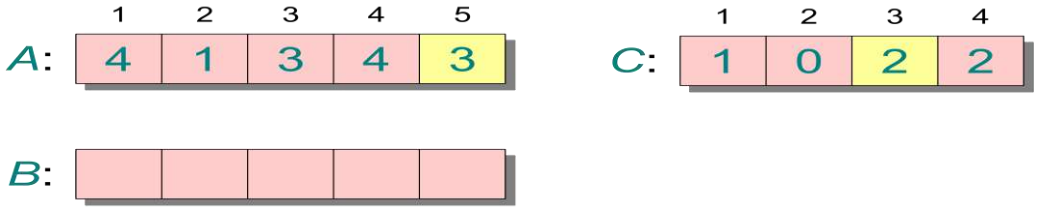



```
for j ← 1 to n
    do C[A[j]] ← C[A[j]] + 1 ▷ C[i] = |{key = i}|
```









Loop 3

	1	2	3	4	5
A:	4	1	3	4	3
B:					

	1	2	3	4
C:	1	0	2	2
C′:	1	1	2	2

for $i \leftarrow 2$ **to** k
do $C[i] \leftarrow C[i] + C[i-1]$ ▷ $C[i] = |\{\text{key} \leq i\}|$

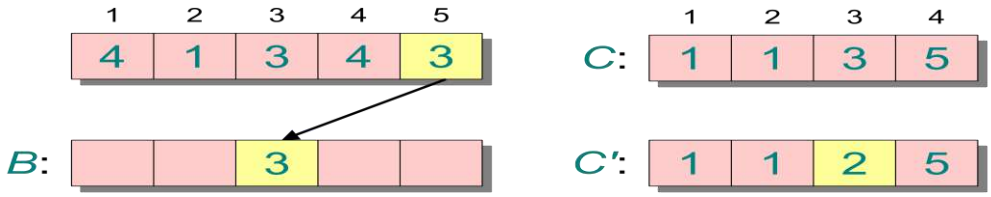
	1	2	3	4	5
A:	4	1	3	4	3
B:					

	1	2	3	4
C:	1	0	2	2
C′:	1	1	3	2

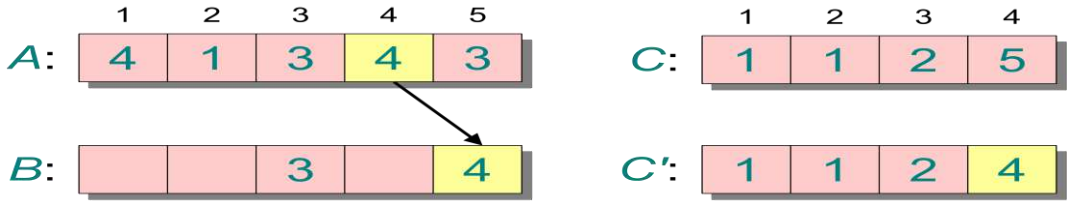
	1	2	3	4	5
	4	1	3	4	3
B:					

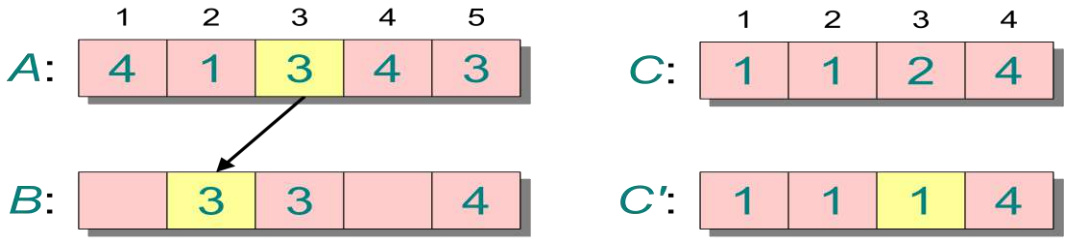
	1	2	3	4
C:	1	0	2	2
C′:	1	1	3	5

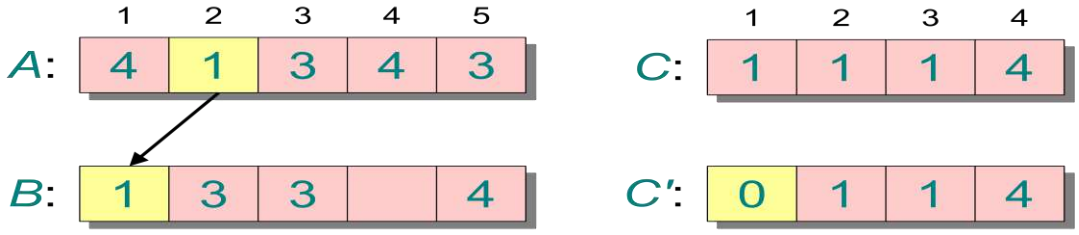
Loop 4



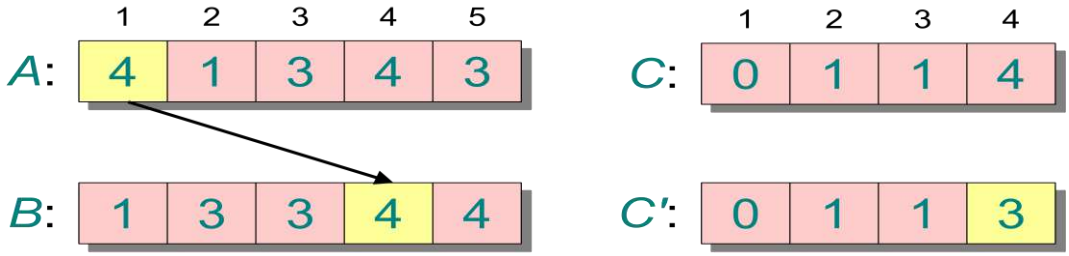

```
for j ← n downto 1
    do    B[C[A[j]]] ← A[j]
      C[A[j]] ← C[A[j]] – 1
```

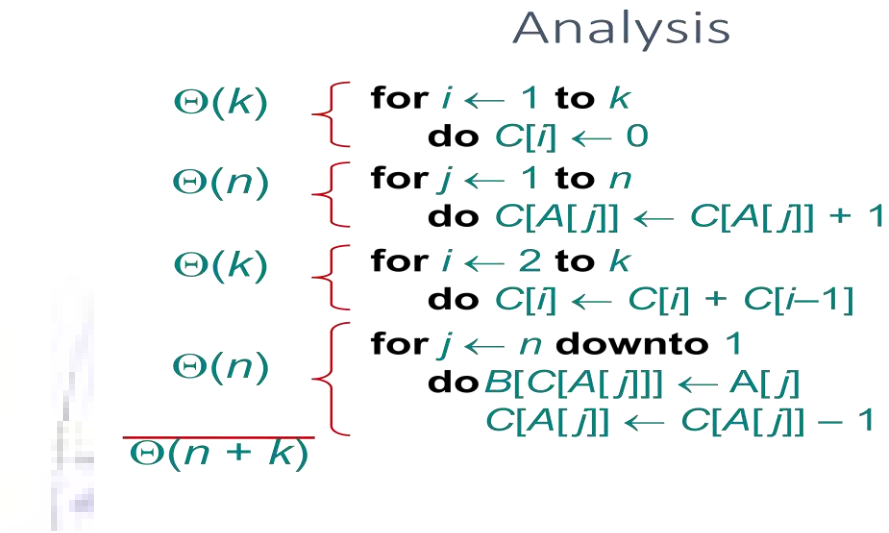






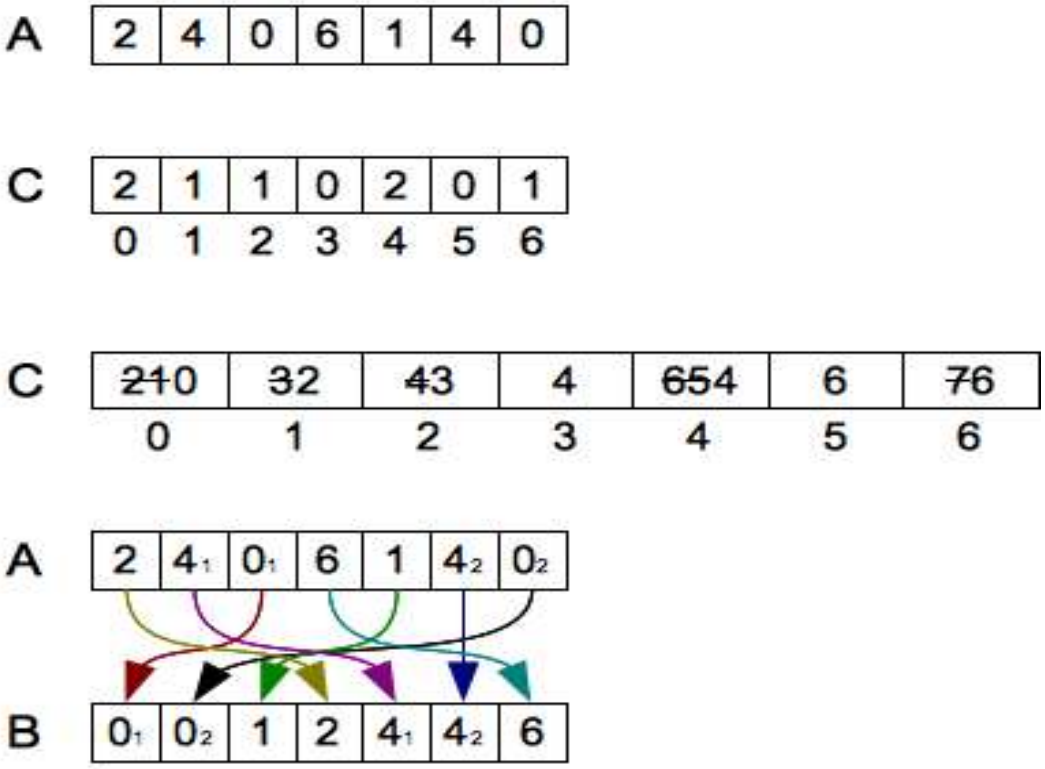
Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk WhatsApp# 0346-5100010





Example 2:

For the given input array *A[]* (note that $k = 6$, i.e. the largest value in the array) the first and second *C[]* arrays are shown along with the reordering (which takes place from the *end* of *A[]* forward with the values subscripted to demonstrate the stable sorting property).



❖ Bucket Sort?

Bucket sort is a comparison sort algorithm that operates on elements by dividing them into different buckets and then sorting these buckets individually. Each bucket is sorted individually using a separate sorting algorithm or by applying the bucket sort algorithm recursively. Bucket sort is mainly useful when the input is uniformly distributed over a range.

For example, consider the following problem. Sort a large set of floating point numbers which are in range from 0.0 to 1.0 and are uniformly distributed across the range. How do we sort the numbers efficiently?

A simple way is to apply a comparison based sorting algorithm. The lower bound for Comparison based sorting algorithm (Merge Sort, Heap Sort, Quick-Sort .. etc) is $\Omega(n \log n)$, i.e., they cannot do better than nLogn.

Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk WhatsApp# 0346-5100010

Can we sort the array in linear time? Counting sort cannot be applied here as we use keys as index in counting sort. Here keys are floating point numbers. The idea is to use bucket sort. Following is bucket algorithm.

Bucket Sort Algorithm

1. Distribute the elements into buckets or bins.
2. Sort each bucket individually.
3. Merge the buckets in order to produce a sorted array as the result.

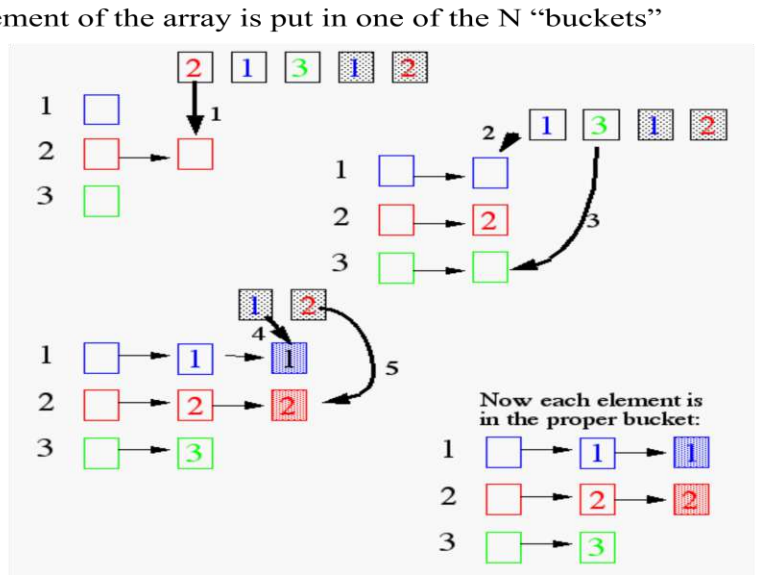
Time Complexity

Expected total time is O(n + N),

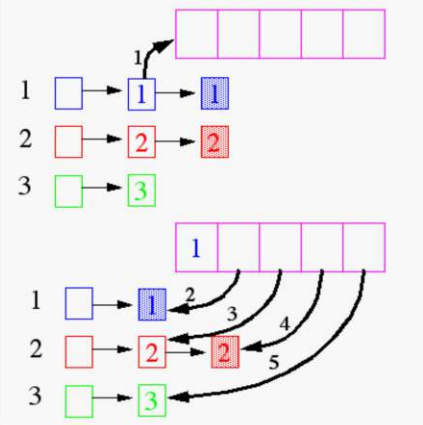
Where n = size of original sequence

if N is O(n) → sorting algorithm in O(n) !

Bucket Sort Algorithm Example



Now, pull the elements from the buckets into the array

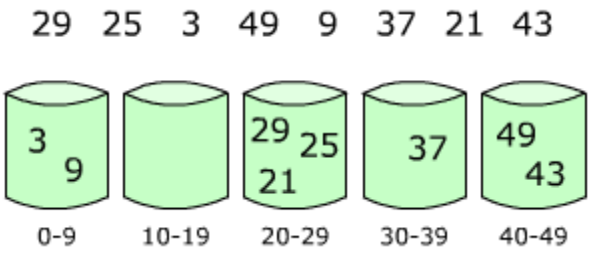


At last, the sorted array (sorted in a stable way):

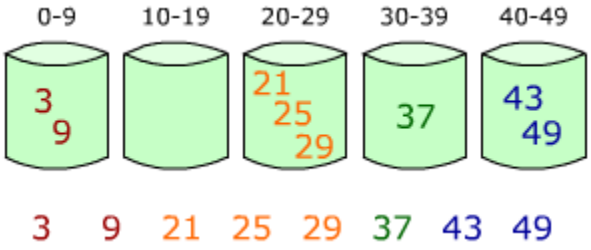
1	1	2	2	3

Example 2

1. Distribute elements in buckets:



2. Sorting inside every bucket and merging:



Example 3
Input

Suppose we have the following list of elements: [2, 56, 4, 77, 26, 98, 55]. Let's use 10 buckets. To determine the capacity of each bucket we need to know the *maximum element value*, in this case 98.

So the buckets are:

- bucket 1: from 0 to 9
- bucket 2: from 10 to 19
- bucket 3: from 20 to 29
- and so on.

Distribution

Now we need to choose a distribution function.

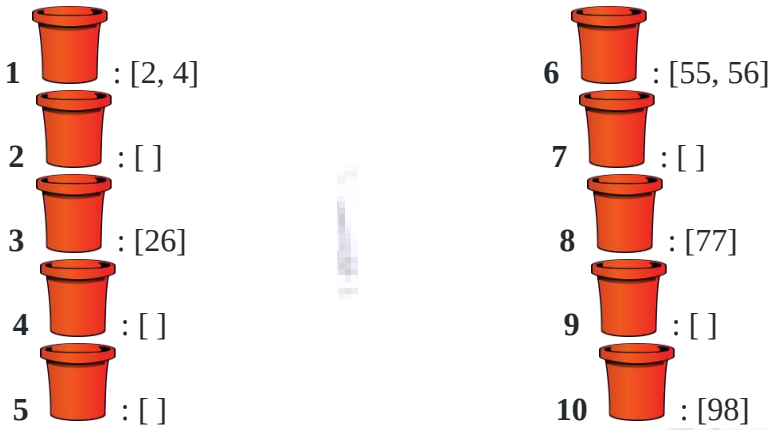
bucketNumber = (elementValue / totalNumberOfBuckets) + 1

Such that by applying that function we distribute all the elements in the buckets.

In our example it is like the following:

1. Apply the distribution function to 2. bucketNumber = (2 / 10) + 1 = 1
2. Apply the distribution function to 56. bucketNumber = (56 / 10) + 1 = 6
3. Apply the distribution function to 4. bucketNumber = (4 / 10) + 1 = 1
4. Apply the distribution function to 77. bucketNumber = (77 / 10) + 1 = 8
5. Apply the distribution function to 26. bucketNumber = (26 / 10) + 1 = 3
6. Apply the distribution function to 98. bucketNumber = (98 / 10) + 1 = 10
7. Apply the distribution function to 55. bucketNumber = (55 / 10) + 1 = 6

Our buckets will be filled now:



We can choose to insert the elements in every bucket in order, or sort every bucket after distributing all the elements.

Put the elements back in the list

Finally we go through all the buckets and put the elements back in the list:

[2, 4, 26, 55, 56, 77, 98]

❖ Radix Sort

The lower bound for Comparison based sorting algorithm (Merge Sort, Heap Sort, Quick-Sort .. etc) is Ω(nLogn), i.e., they cannot do better than nLogn.

Counting sort is a linear time sorting algorithm that sort in O(n+k) time when elements are in range from 1 to k.

What if the elements are in range from 1 to n^2?

We can't use counting sort because counting sort will take O(n^2) which is worse than comparison based sorting algorithms. Can we sort such an array in linear time?

Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk WhatsApp# 0346-5100010

Radix Sort is the answer. The idea of Radix Sort is to do digit by digit sort starting from least significant digit to most significant digit. Radix sort uses counting sort as a subroutine to sort.

The Radix Sort Algorithm

1) Do following for each digit i where i varies from least significant digit to the most significant digit.

a) Sort input array using counting sort (or any stable sort) according to the i'th digit.

Example:

Original, unsorted list:

170, 45, 75, 90, 802, 24, 2, 66

Sorting by least significant digit (1s place) gives: [*Notice that we keep 802 before 2, because 802 occurred before 2 in the original list, and similarly for pairs 170 & 90 and 45 & 75.]

170, 90, 802, 2, 24, 45, 75, 66

Sorting by next digit (10s place) gives: [*Notice that 802 again comes before 2 as 802 comes before 2 in the previous list.]

802, 2, 24, 45, 66, 170, 75, 90

Sorting by most significant digit (100s place) gives:

2, 24, 45, 66, 75, 90, 170, 802

What is the running time of Radix Sort?

Let there be d digits in input integers. Radix Sort takes O(d*(n+b)) time where b is the base for representing numbers, for example, for decimal system, b is 10. What is the value of d? If k is the maximum possible value, then d would be $O(\log_b(k))$. So overall time complexity is $O((n+b) * \log_b(k))$. Which looks more than the time complexity of comparison based sorting algorithms for a large k. Let us first limit k. Let $k <= n^c$ where c is a constant. In that case, the complexity becomes $O(n\log_b(n))$. But it still doesn't beat comparison based sorting algorithms.

What if we make value of b larger?. What should be the value of b to make the time complexity linear? If we set b as n, we get the time complexity as O(n). In other words, we can sort an array

of integers with range from 1 to n^c if the numbers are represented in base n (or every digit takes $\log_2(n)$ bits).

Is Radix Sort preferable to Comparison based sorting algorithms like Quick-Sort?

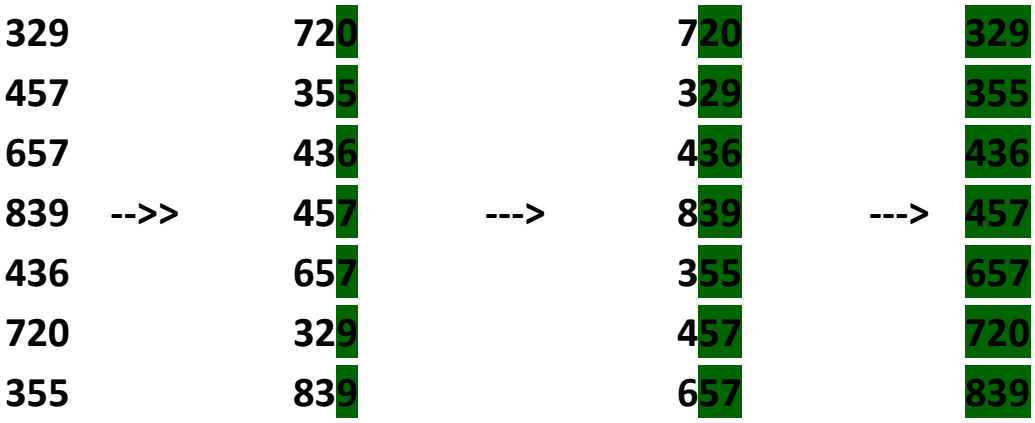
If we have $\log_2 n$ bits for every digit, the running time of Radix appears to be better than Quick Sort for a wide range of input numbers. The constant factors hidden in asymptotic notation are higher for Radix Sort and Quick-Sort uses hardware caches more effectively. Also, Radix sort uses counting sort as a subroutine and counting sort takes extra space to sort numbers.

Algorithm:

For each digit *i* where *i* varies from the least significant digit to the most significant digit of a number. Sort input array using count sort algorithm according to *i*th digit. We used count sort because it is a stable sort.

Examples No# 1

Unsorted List : 329, 457, 657, 839, 436, 720, 355



Sorted List : 329, 355, 436, 457, 657, 720, 839

Examples No# 2

Unsorted List: 326, 453, 608, 835,751, 435, 704, 690

751	-----> 835	----> 435 ---->	690
435	435	751	704
704	326	453	751
690	608	690	835

Sorted List: 326,435, 453,608, 690,704, 751, 835

Time Complexity

- Best, Average and Worst: O(nk)
- where n is the number of elements and k is the number o digits in the largest number (or the number of passes).

Space Complexity

- O(n+k)
- Where n is the number of elements and k is the range of digit values.

Recursive Definations

Base Case

T(n,0)=O(1)

if k=0 (it means there is no digit for process then time complexity is constant)

Recursive Case

T(n,k)=T(n,k-1)+O(n)

Recurrence Relation

T(n,k)=T(n,k-1)+O(n)

T(n,k-1)=T(n,k-2)+O(n)

T(n,k)=T(n,k-2)+O(n)+O(n)

T(n,k)=T(n,k-2)+2O(n)

$T(n,k-2)=T(n,k-3)+O(n)$

$T(n,k)=T(n,k-3)+O(n)+2O(n)$

$T(n,k)=T(n,k-3)+3O(n)$

$T(n,k)=T(n,0)+k.\ O(n)$

$T(n,k)=O(1)+O(k.n)$

$T(n,k)=O(k.n)$

Recursive Algorithm

- Here's an example of a recursive radix sort algorithm:
- Sort the numbers by the most significant digit (MSD). For example, if the numbers are in the hundreds place, sort them into buckets based on the hundreds digit.
- Sort the numbers in each bucket by the next digit. For example, if the numbers in the zero hundreds bucket are sorted by the tens digit, sort them into buckets based on the tens digit.
- Repeat the process until the last significant digit has been sorted.

Explanation

- Radix sort is a non-comparison sorting algorithm that sorts numbers by processing them digit by digit. It can be implemented recursively (MSD radix sort) or non-recursively (LSD radix sort).

Radix Sort Real Life Examples

1. Sorting CNIC Numbers

Scenario: Organizing National Identity Card (CNIC) numbers in a database.

Approach: CNIC numbers are 13 digits long, making them perfect for radix sorting digit by digit.

2. Sorting Roll Numbers for Exam Results

Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk WhatsApp# 0346-5100010

Scenario: Organizing student roll numbers while preparing exam results.

Approach: Roll numbers can be sorted efficiently since they are numeric and structured.

❖ Heap Sort

Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk WhatsApp# 0346-5100010

Heap sort is a comparison based sorting technique based on Binary Heap data structure. It is similar to selection sort where we first find the maximum element and place the maximum element at the end. We repeat the same process for remaining element.

What is Binary Heap?

Let us first define a Complete Binary Tree. A complete binary tree is a binary tree in which every level, except possibly the last, is completely filled, and all nodes are as far left as possible. A Binary Heap is a Complete Binary Tree where items are stored in a special order such that value in a parent node is greater (or smaller) than the values in its two children nodes. The former is called as max heap and the latter is called min heap. The heap can be represented by binary tree or array.

Why array based representation for Binary Heap?

Since a Binary Heap is a Complete Binary Tree, it can be easily represented as array and array based representation is space efficient. If the parent node is stored at index I, the left child can be calculated by 2 * I + 1 and right child by 2 * I + 2 (assuming the indexing starts at 0).

Heap Sort Algorithm for sorting in increasing order:

1. Build a max heap from the input data.

2. At this point, the largest item is stored at the root of the heap. Replace it with the last item of the heap followed by reducing the size of heap by 1. Finally, heapify the root of tree.

3. Repeat above steps while size of heap is greater than 1.

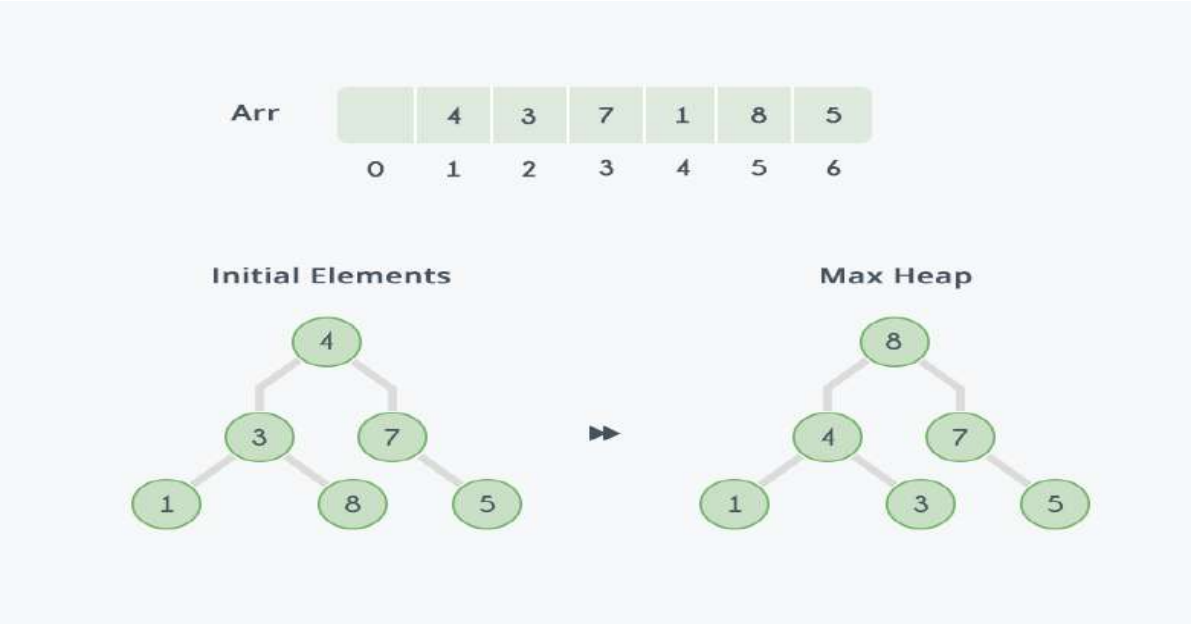
Complexity:

max_heapify has complexity O(logN), build_maxheap has complexity O(N) and we run max_heapify $N-1$ times in heap_sort function, therefore complexity of heap_sort function is O(NlogN)

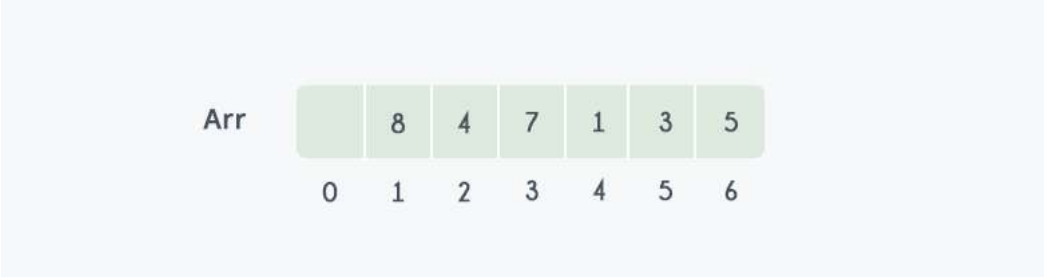
Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk WhatsApp# 0346-5100010

Example:

In the diagram below, initially there is an unsorted array Arr having 6 elements and then max-heap will be built.



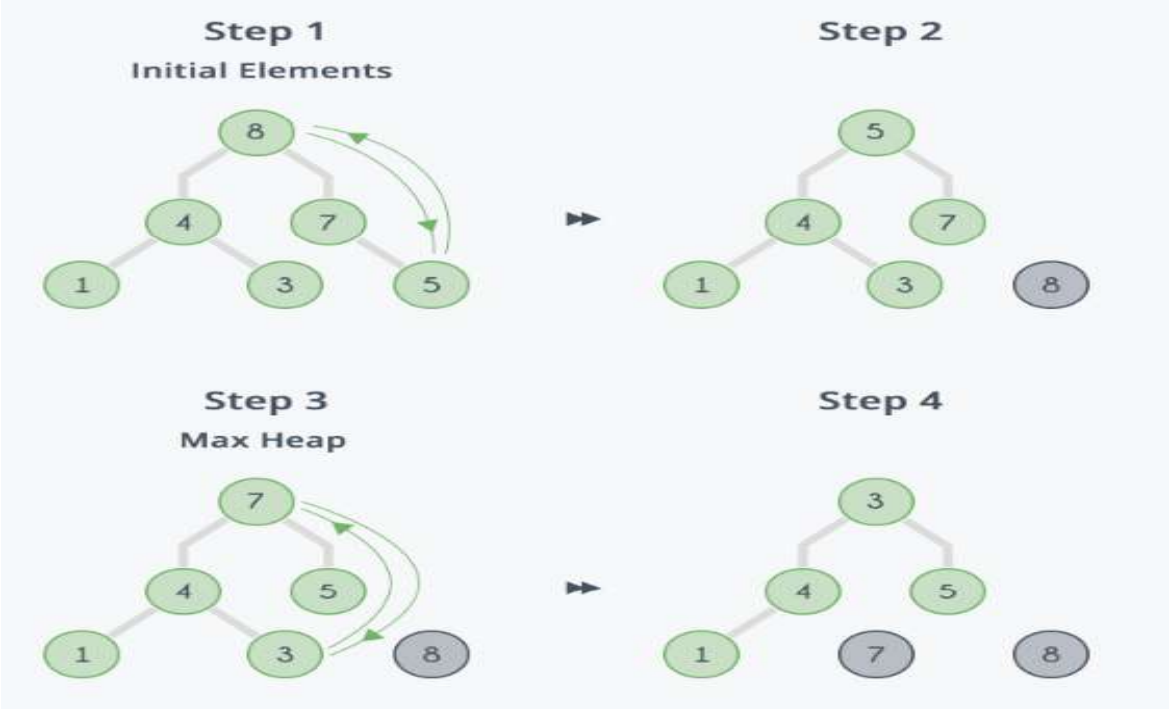
After building max-heap, the elements in the array Arr will be:

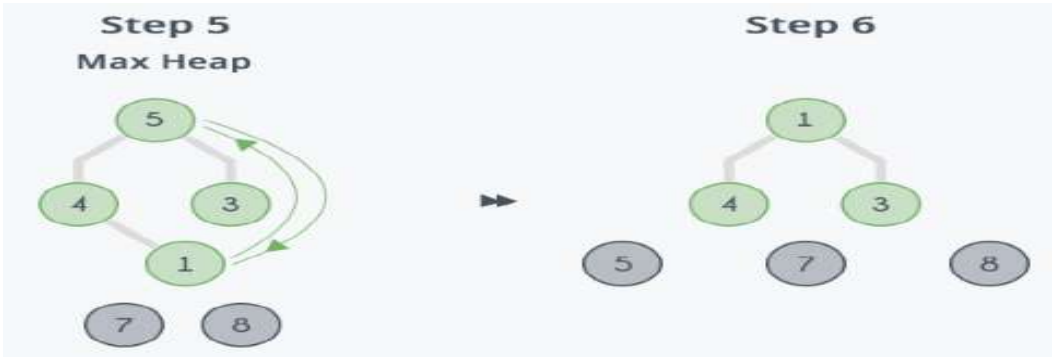


Steps

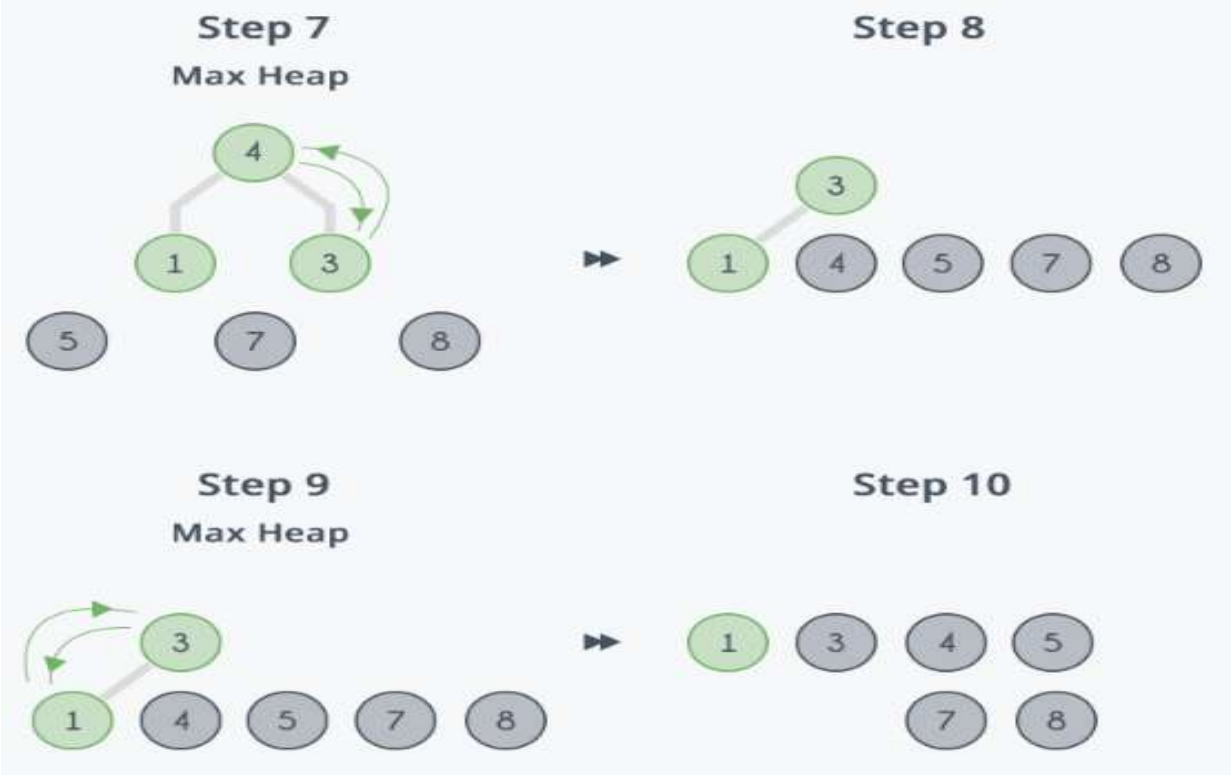
Step 1: 8 is swapped with 5.
Step 2: 8 is disconnected from heap as 8 is in correct position now and.
Step 3: Max-heap is created and 7 is swapped with 3.
Step 4: 7 is disconnected from heap.
Step 5: Max heap is created and 5 is swapped with 1.
Step 6: 5 is disconnected from heap.
Step 7: Max heap is created and 4 is swapped with 3.
Step 8: 4 is disconnected from heap.
Step 9: Max heap is created and 3 is swapped with 1.
Step 10: 3 is disconnected.

Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk WhatsApp# 0346-5100010

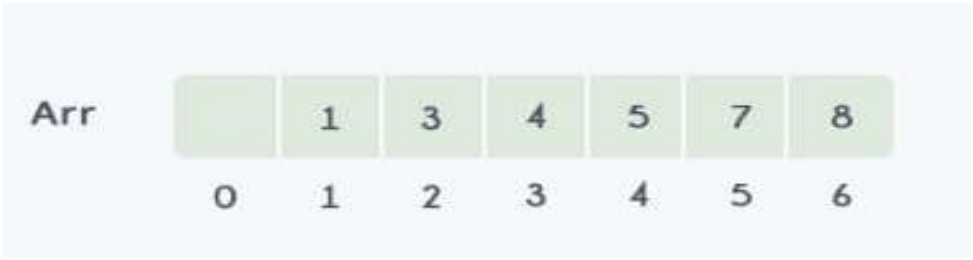




Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk WhatsApp# 0346-5100010



After all the steps, we will get a sorted array.

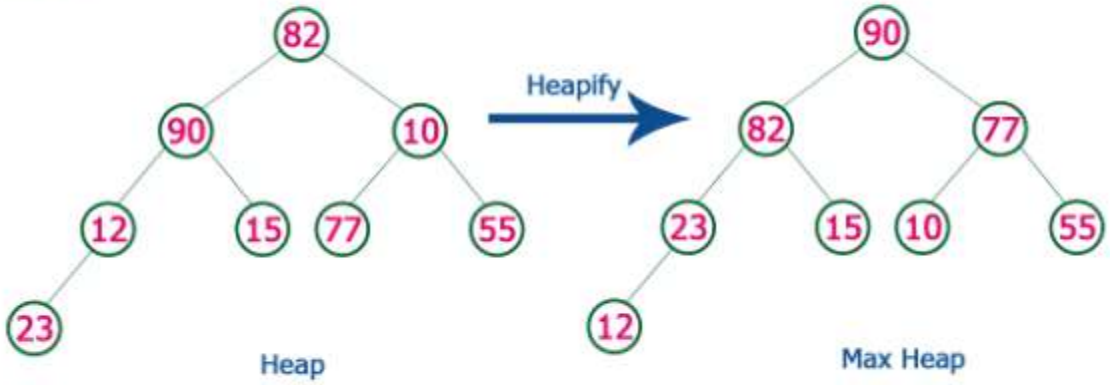


Example

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

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

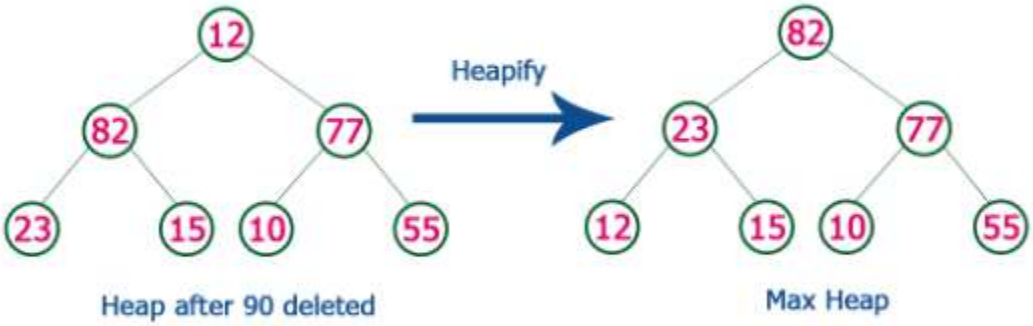
Step 1 - Construct a Heap with given list of unsorted numbers and convert to Max Heap



list of numbers after heap converted to Max Heap

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

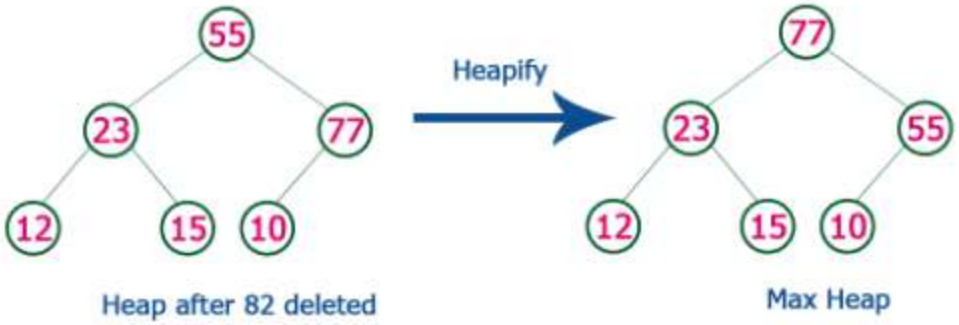
Step 2 - Delete root (**90**) from the Max Heap. To delete root node it needs to be swapped with last node (**12**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 90 with 12.

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

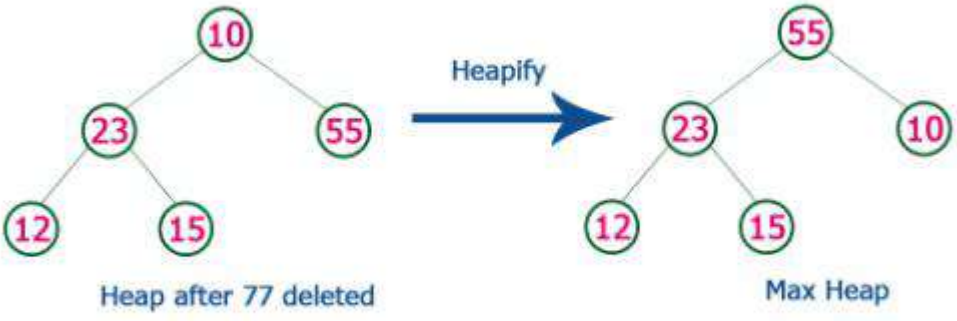
Step 3 - Delete root (**82**) from the Max Heap. To delete root node it needs to be swapped with last node (**55**). After delete tree needs to be heapify to make it Max Heap.



list of numbers after swapping 82 with 55.

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

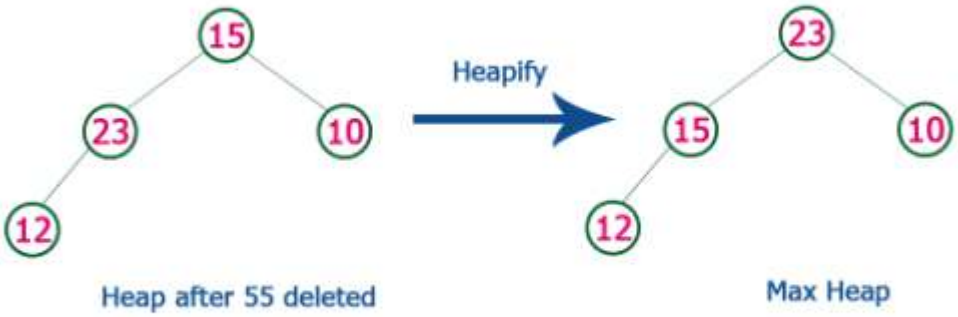
Step 4 - Delete root (**77**) from the Max Heap. To delete root node it needs to be swapped with last node (**10**). After delete tree needs to be heapify to make it Max Heap.



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.



list of numbers after swapping 55 with 15.

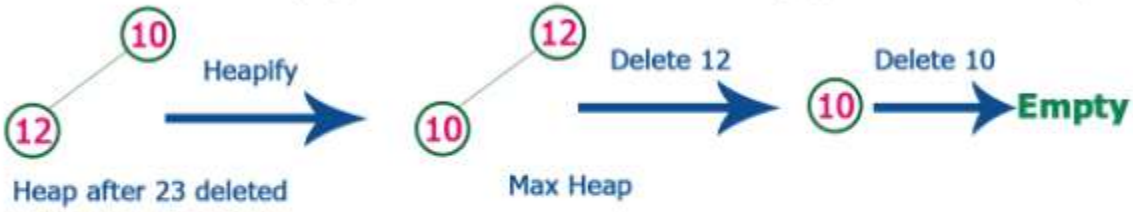
12, 15, 10, 23, 55, 77, 82, 90

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.

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.



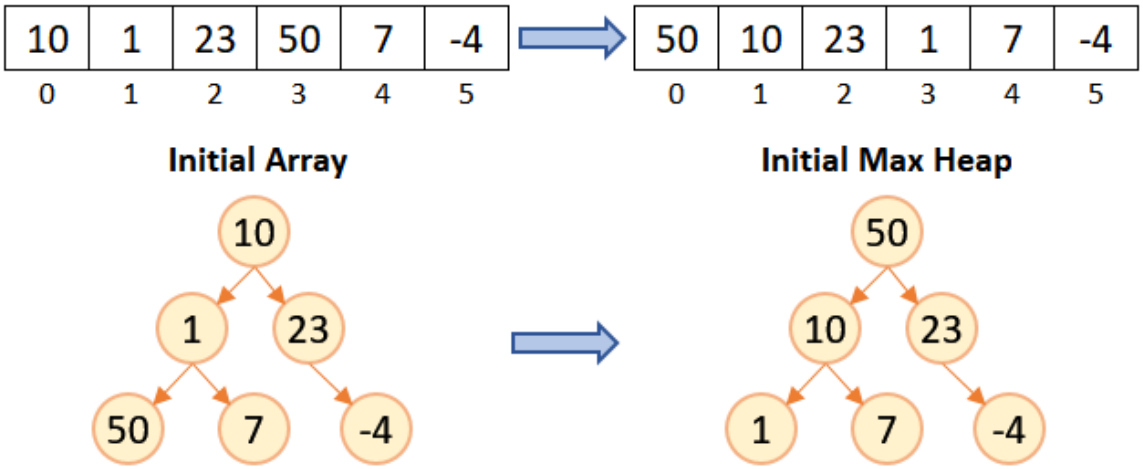
list of numbers after Deleting 15, 12 & 10 from the Max Heap.

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

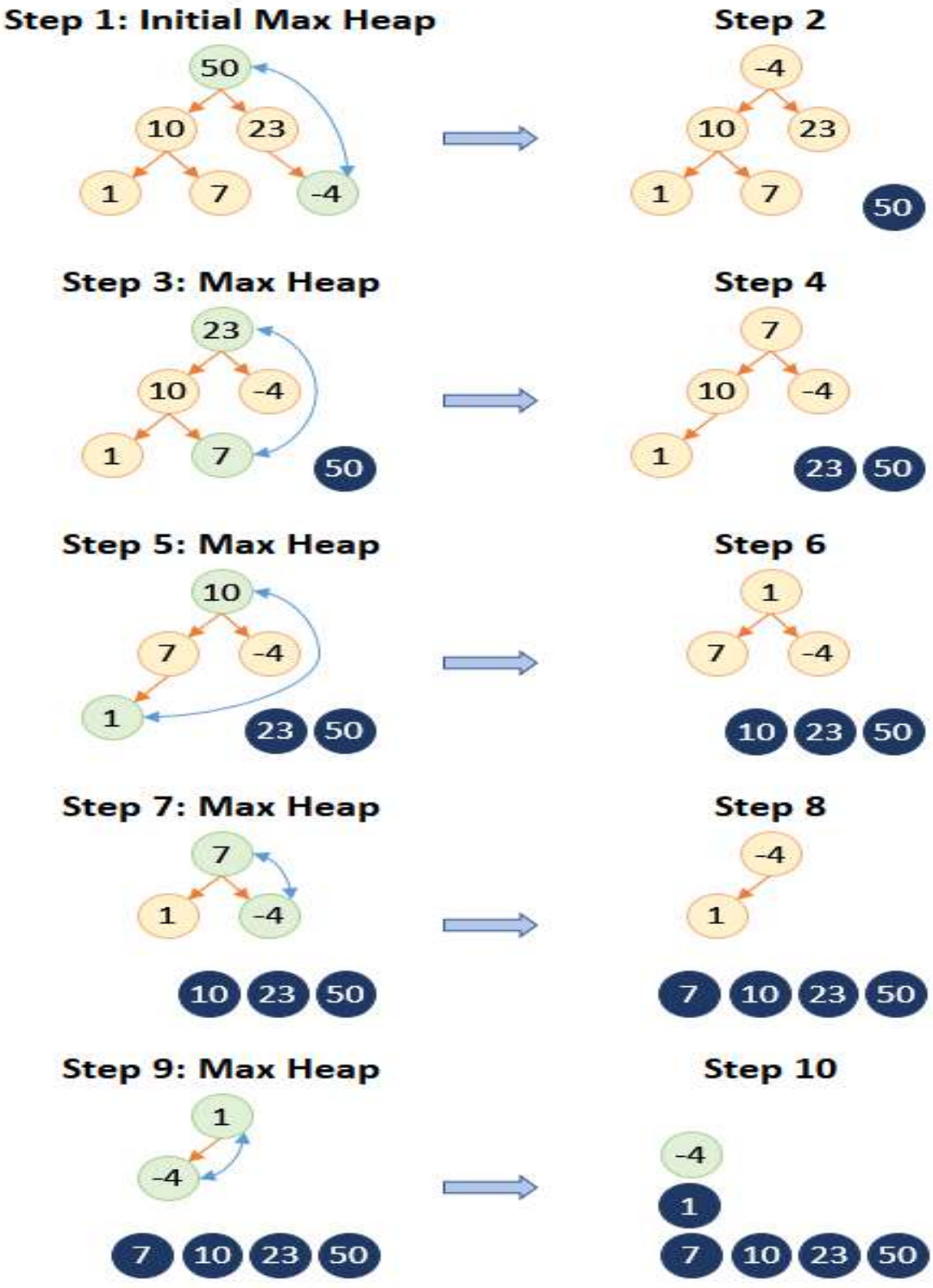
Whenever Max Heap becomes Empty, the list get sorted in Ascending order

Example

Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk WhatsApp# 0346-5100010

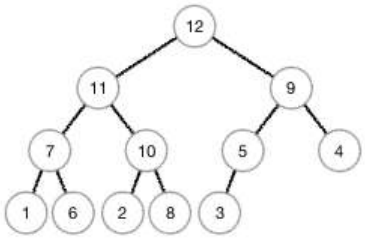


Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk WhatsApp# 0346-5100010

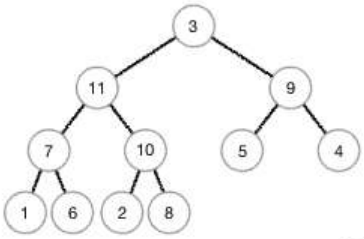


Example

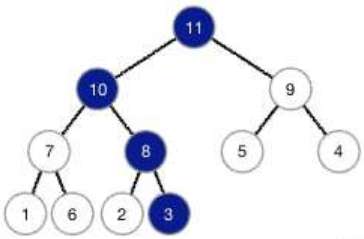
Dr. Naseer Ahmed Sajid email id: naseer@biit.edu.pk WhatsApp# 0346-5100010



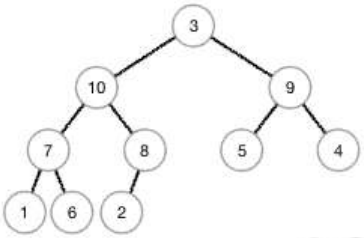
i) Build-Max-Heap



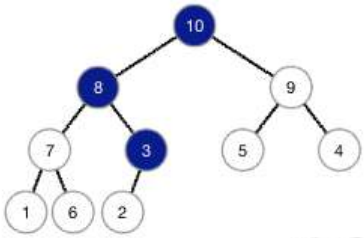
ii) Pick Root 12



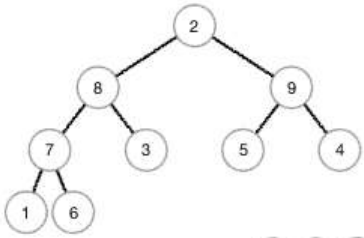
iii) Max-Heapify 12



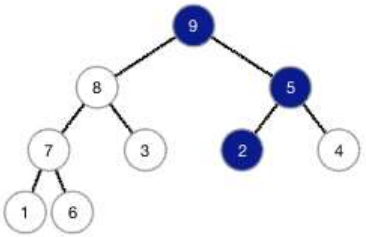
ii) Pick Root 11 12



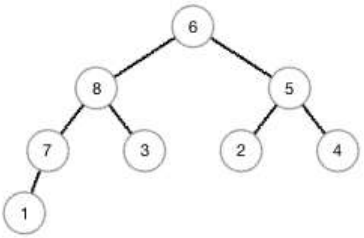
iii) Max-Heapify 11 12



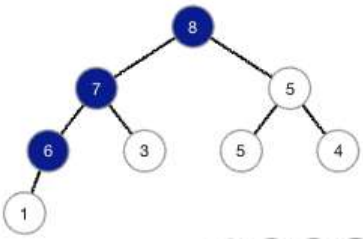
ii) Pick Root 10 11 12



iii) Max-Heapify 10 11 12



ii) Pick Root 9 10 11 12



iii) Max-Heapify 9 10 11 12