KARNAVATI UNIVERSITY

UIT

Unitedworld Institute Of Technology

शिक्षणतः सिद्धि

B.Tech. Computer Science & Engineering

Semester-3

Data Structures and Algorithms

Course Code: 71203002002

Analyzing Control Statements

For Loop

```
# Input     : int A[n], array of n integers
# Output    : Sum of all numbers in array A


Algorithm: int Sum(int A[], int n)
{
      int s=0;                            n+1
      for (int i=0; i<n; i++)
 1          s = s + A[i];
      return s;
                        n
}
```

Total time taken = n+1+n+2 = 2n+3
Time Complexity f(n) = 2n+3

Running Time of Algorithm

- The time complexity of the algorithm is : $f(n) = 2 \cdot n + 3$
- Estimated running time for different values of n :

$n = 10$	23 steps
$n = 100$	203 steps
$n = 1000$	2,003 steps
$n = 10000$	20,003 steps

- As n grows, the number of steps grow in linear proportion to n for the given algorithm Sum.
- The dominating term in the function of time complexity is n: As n gets large, the $+3$ becomes insignificant.
- **The time is linear in proportion to n.**

Analyzing Control Statements

Example 1:

$sum = a + b;$ $\mathbf{c}$

- Statement is executed once only
- So, The execution time $T(n)$ is some constant $\mathbf{c} \approx \boldsymbol{O}(\mathbf{1})$

Example 2:

$for\ i = 1\ to\ n\ do$ $\boldsymbol{c_1 * (n+1)}$

$sum = a + b;$ $\boldsymbol{c_2 * (n)}$

- Total time is denoted as,

$$\boldsymbol{T(n) = c_1 n + c_1 + c_2 n}$$

$$\boldsymbol{T(n) = n(c_1 + c_2) + c_1 \approx O(n)}$$

Example 3:

$for\ i = 1\ to\ n\ do$ $\boldsymbol{c_1\ (n+1)}$

$for\ j\ =\ 1\ to\ n\ do$ $\boldsymbol{c_2\ n\ (n+1)}$

$sum = a + b;$ $\boldsymbol{c_3 * n * n}$

- Analysis

$$T(n) = c_1(n+1) + c_2 n(n+1) + c_3 n(n)$$

$$T(n) = c_1 n + c_1 + c_2 n^2 + c_2 n + c_3 n^2$$

$$T(n) = n^2(c_2 + c_3) + n(c_1 + c_2) + c_1$$

$$T(n) = an^2 + bn + c$$

$$\boldsymbol{T(n) = O(n^2)}$$

Analyzing Control Statements

Example 4:

$l = 0$

$for\ i = 1\ to\ n\ do$

$\quad for\ j = 1\ to\ i\ do$

$\quad\quad for\ k = j\ to\ n\ do$

$\quad\quad\quad l = l + 1$

$$\boldsymbol{t(n) = \theta(n^3)}$$

Example 5:

$l = 0$

$for\ i = 1\ to\ n\ do$

$\quad for\ j = 1\ to\ n^2\ do$

$\quad\quad for\ k = 1\ to\ n^3\ do$

$\quad\quad\quad l = l + 1$

$$\boldsymbol{t(n) = \theta(n^6)}$$

Example 6:

$for\ j = 1\ to\ n\ do$

$\quad for\ k = 1\ to\ j\ do$

$\quad\quad sum = sum + j * k$

$\boldsymbol{\theta(n^2)}$

$for\ l = 1\ to\ n\ do$

$\quad sum = sum - l + 1$

$\boldsymbol{\theta(n)}$

printf(“sum is now %d”, *sum*)

$\boldsymbol{\theta(1)}$

$$\boldsymbol{t(n) = \theta(n^2) + \theta(n) + \theta(1)}$$
$$\boldsymbol{t(n) = \theta(n^2)}$$

Sorting Algorithms

Bubble Sort, Selection Sort, Insertion Sort

Introduction

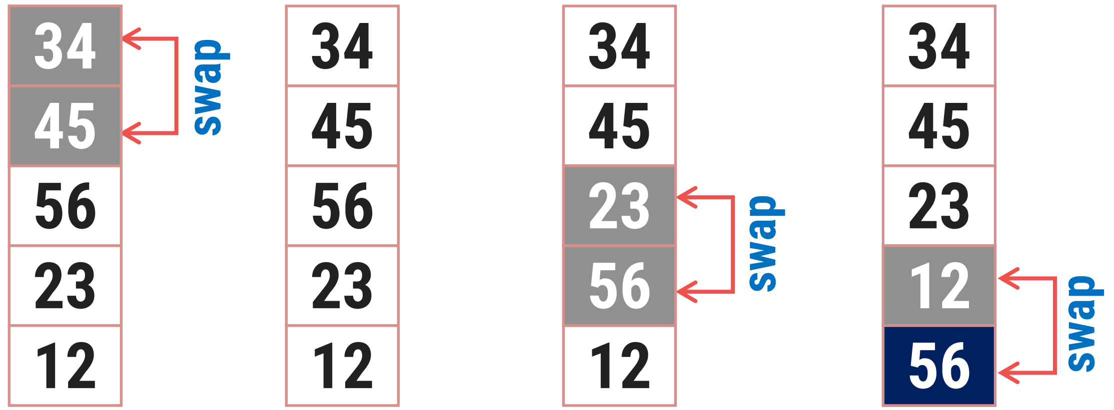
- Sorting is any process of arranging items systematically or arranging items in a sequence ordered by some criterion.
- Applications of Sorting
    1. Phone Bill: the calls made are date wise sorted.
    2. Bank statement or Credit card Bill: transactions made are date wise sorted.
    3. Filling forms online: “select country” drop down box will have the name of countries sorted in Alphabetical order.
    4. Online shopping: the items can be sorted price wise, date wise or relevance wise.
    5. Files or folders on your desktop are sorted date wise.

Bubble Sort – Example

Sort the following array in Ascending order

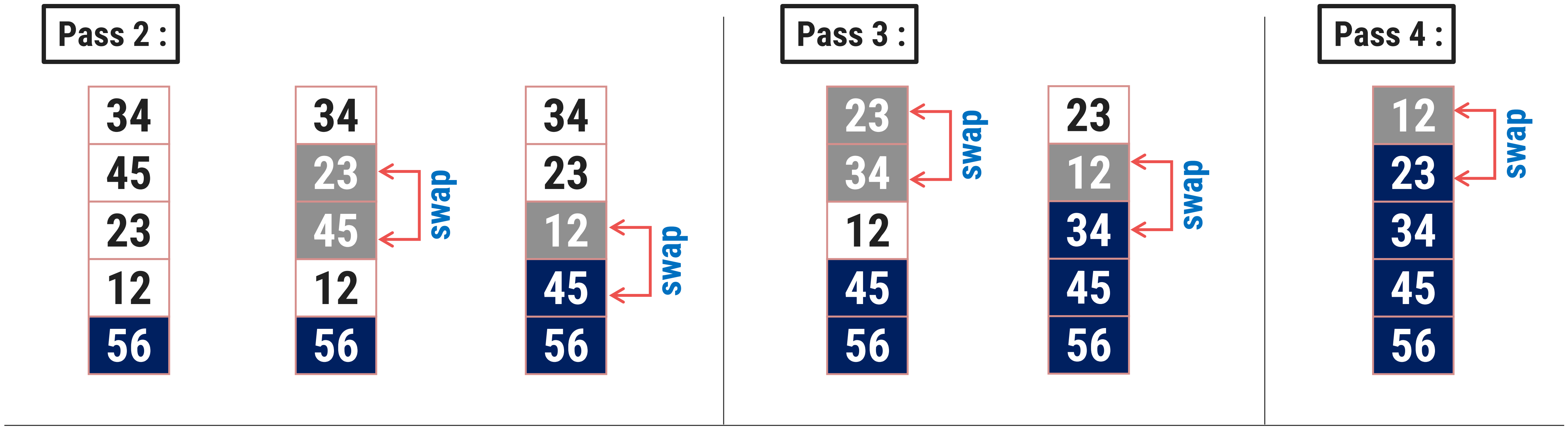
45	34	56	23	12

Pass 1 :



$$if\,(A[j] \;>\; A[j+1])$$
$$swap(A[j], A[j+1])$$

Bubble Sort – Example



$$if\,(A[j]\ >\ A[j+1])$$
$$swap(A[j], A[j+1])$$

Bubble Sort - Algorithm

```
# Input: Array A
# Output: Sorted array A

Algorithm: Bubble_Sort(A)
for i ← 1 to n-1 do
    for j ← 1 to n-i do
      if  A[j] > A[j+1] then
          temp ← A[j]
          A[j] ← A[j+1]
          A[j+1] ← temp
```

$\theta(n)$

$\theta(n^2)$

Bubble Sort

- It is a simple sorting algorithm that works by comparing each pair of adjacent items and swapping them if they are in the wrong order.
- The pass through the list is repeated until no swaps are needed, which indicates that the list is sorted.
- As it only uses comparisons to operate on elements, it is a comparison sort.
- Although the algorithm is simple, it is too slow for practical use.
- The time complexity of bubble sort is $\boldsymbol{\theta(n^2)}$

Bubble Sort Algorithm – Best Case Analysis

```
# Input: Array A
# Output: Sorted array A
Algorithm: Bubble_Sort(A)
int flag=1;
for i ← 1 to n-1 do
      for j ← 1 to n-i do
            if A[j] > A[j+1] then
                  flag = 0;
                  swap(A[j],A[j+1])
      if(flag == 1)
            cout<<"already sorted"<<endl
            break;
```

Condition never becomes true

Pass 1 : i = 1

12
23
34
45
59

j = 1
j = 2
j = 3
j = 4

Best case time complexity = $\theta(n)$

Selection Sort – Example 1

Sort the following elements in Ascending order

5	1	12	-5	16	2	12	14

Step 1 :

Unsorted Array

5	1	12	-5	16	2	12	14
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**

Step 2 :

Unsorted Array (elements 2 to 8)

-5	**1**	**12**	**5**	**16**	**2**	**12**	**14**
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**

Swap

Index = 4, value = -5

- **Minj** denotes the current index and **Minx** is the value stored at current index.
- **So, Minj = 1, Minx = 5**
- Assume that currently **Minx** is the smallest value.
- Now find the smallest value from the remaining entire Unsorted array.

Selection Sort – Example 1

Step 3 :

Unsorted Array (elements 3 to 8)

-5	1	12	5	16	2	12	14
1	2	3	4	5	6	7	8

- **Now Minj = 2, Minx = 1**
- Find min value from remaining unsorted array

Index = 2, value = 1

No Swapping as min value is already at right place

Step 4 :

Unsorted Array (elements 4 to 8)

-5	1	2	5	16	12	12	14
1	2	3	4	5	6	7	8

Swap

- **Minj = 3, Minx = 12**
- Find min value from remaining unsorted array

Index = 6, value = 2

Selection Sort – Example 1

Step 5 :

Unsorted Array
(elements 5 to 8)

-5	1	2	5	16	12	12	14
1	2	3	4	5	6	7	8

- **Now Minj = 4, Minx = 5**
- Find min value from remaining unsorted array

Index = 4, value = 5

No Swapping as min value is already at right place

Step 6 :

Unsorted Array
(elements 6 to 8)

-5	1	2	5	12	16	12	14
1	2	3	4	5	6	7	8

Swap

- **Minj = 5, Minx = 16**
- Find min value from remaining unsorted array

Index = 6, value = 12

Selection Sort – Example 1

Step 7 :

Unsorted Array (elements 7 to 8)

1	2	3	4	5	6	7	8
-5	1	2	5	12	12	16	14

Swap

- **Now Minj = 6, Minx = 16**
- Find min value from remaining unsorted array

Index = 7, value = 12

Step 8 :

Unsorted Array (element 8)

1	2	3	4	5	6	7	8
-5	1	2	5	12	12	14	16

Swap

- **Minj = 7, Minx = 16**
- Find min value from remaining unsorted array

Index = 8, value = 14

The entire array is sorted now.

Selection Sort

- Selection sort divides the array or list into two parts,
  1. The sorted part at the left end
  2. 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.
- Then it finds the second smallest element and exchanges it with the element in the second leftmost position.
- This process continues until the entire array is sorted.
- The time complexity of selection sort is $\boldsymbol{\theta(n^2)}$

Selection Sort - Algorithm

```
# Input: Array A
# Output: Sorted array A

Algorithm: Selection_Sort(A)
for i ← 1 to n-1 do
      minj ← i;
      minx ← A[i];
      for j ← i + 1 to n do
            if A[j] < minx then
                  minj ← j;
                  minx ← A[j];
      A[minj] ← A[i];
      A[i] ← minx;
```

$\theta(n)$

$\theta(n^2)$

Selection Sort – Example 2

Algorithm: Selection_Sort(A)

```
for i ← 1 to n-1 do
    minj ← i; minx ← A[i];
    for j ← i + 1 to n do
        if A[j] < minx then
            minj ← j ; minx ← A[j];
    A[minj] ← A[i];
    A[i] ← minx;
```

Pass 1 :

i = 1

minj ← 2

minx ← 34 No Change

j = 2 3

A[j] = 56

Sort in Ascending order

45	34	56	23	12
1	2	3	4	5

Selection Sort – Example 2

```
Algorithm: Selection_Sort(A)
for i ← 1 to n-1 do
    minj ← i; minx ← A[i];
    for j ← i + 1 to n do
        if A[j] < minx then
            minj ← j ; minx ← A[j];
    A[minj] ← A[i];
    A[i] ← minx;
```

Pass 1 :

i = 1

minj ← 5

minx ← 12

j = 2 3 4 5

A[j] = 12

Sort in Ascending order

45	34	56	23	12
1	2	3	4	5

Swap

Unsorted Array

12	34	56	23	45
1	2	3	4	5

12	23	34	45	56

Insertion Sort – Example

Sort the following elements in Ascending order

5	1	12	-5	16	2	12	14

Step 1 :

Unsorted Array

5	1	12	-5	16	2	12	14
1	2	3	4	5	6	7	8

Step 2 :

j

5 1	1	12	-5	16	2	12	14
1	**2**	**3**	**4**	**5**	**6**	**7**	**8**

Shift down

$i = 2, x = 1$ | $j = i - 1 \; and \; j > 0$

while $x < T[j]$ do
$T[j+1] \leftarrow T[j]$
$j--$

Insertion Sort – Example

Step 3 :

	j						
1	5	12	-5	16	2	12	14
1	2	3	4	5	6	7	8

$i = 3, x = 12$ $\quad$ $j = i - 1$ and $j > 0$

while $x < T[j]$ do
$\quad T[j + 1] \leftarrow T[j]$
$\quad j - -$

No Shift will take place

Step 4 :

j		j					
1 -5	5	12	-5	16	2	12	14
1	2	3	4	5	6	7	8

Shift down
Shift down
Shift down

$i = 4, x = -5$ $\quad$ $j = i - 1$ and $j > 0$

while $x < T[j]$ do
$\quad T[j + 1] \leftarrow T[j]$
$\quad j - -$

Insertion Sort – Example

Step 5 :

j

-5	1	5	12	16	2	12	14
1	2	3	4	5	6	7	8

$i = 5, x = 16$ | $j = i - 1$ and $j > 0$

while $x < T[j]$ do
$T[j+1] \leftarrow T[j]$
$j--$

No Shift will take place

Step 6 :

$i = 6, x = 2$ | $j = i - 1$ and $j > 0$

j j

-5	1	2	12	16	2	12	14
1	2	3	4	5	6	7	8

Shift down Shift down Shift down

while $x < T[j]$ do
$T[j+1] \leftarrow T[j]$
$j--$

Insertion Sort – Example

Step 7 :

					j		
-5	1	2	5	12	12	12	14
1	2	3	4	5	6	7	8

Shift down

$i = 7, x = 12$ | $j = i - 1 \; and \; j > 0$

while $\boxed{x < T[j]}$ do
$T[j+1] \leftarrow T[j]$
$j--$

Step 8 :

$i = 8, x = 14$ | $j = i - 1 \; and \; j > 0$

while $\boxed{x < T[j]}$ do
$T[j+1] \leftarrow T[j]$
$j--$

						j	
-5	1	2	5	12	12	14	14
1	2	3	4	5	6	7	8

Shift down

The entire array is sorted now.

Insertion Sort - Algorithm

```
# Input: Array T
# Output: Sorted array T

Algorithm: Insertion_Sort(T[1,…,n])
for i ← 2 to n do
      x ← T[i];
      j ← i - 1;
      while x < T[j] and j > 0 do
            T[j+1] ← T[j];
            j ← j - 1;
      T[j+1] ← x;
```

$\theta(n)$

$\theta(n^2)$

Insertion Sort Algorithm – Best Case Analysis

```
# Input: Array T
# Output: Sorted array T

Algorithm: Insertion_Sort(T[1,…,n])
for i ← 2 to n do
      x ← T[i];
      j ← i – 1;
      while x < T[j] and j > 0 do
            T[j+1] ← T[j];
            j ← j – 1;
      T[j+1] ← x;
```

$\theta(n)$

Pass 1 :

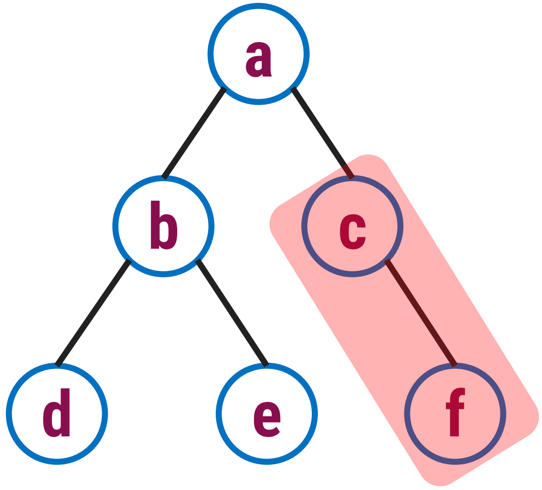
T	i	x	T[j]
12			
23	i=2	x=23	T[j]=12
34	i=3	x=34	T[j]=23
45	i=4	x=45	T[j]=34
59	i=5	x=59	T[j]=45

The best case time complexity of Insertion sort is $\theta(n)$
The average and worst case time complexity of Insertion sort is $\theta(n^2)$

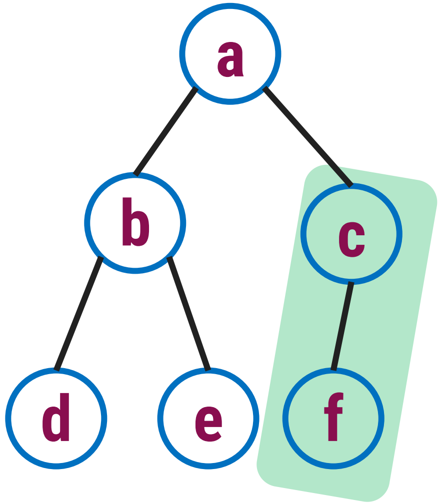
Heap & Heap Sort Algorithm

Introduction

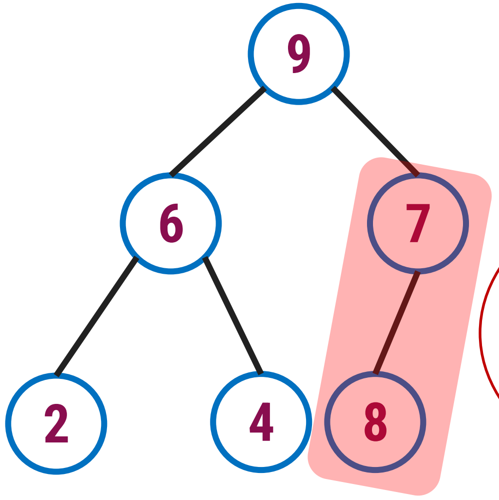
- A heap data structure is a binary tree with the following two properties.
  1. It is a complete binary tree: Each level of the tree is completely filled, except possibly the bottom level. At this level it is filled from left to right.
  2. It satisfies the **heap order** property: the data item stored in each node is **greater than or equal to** the data item stored in its children node.



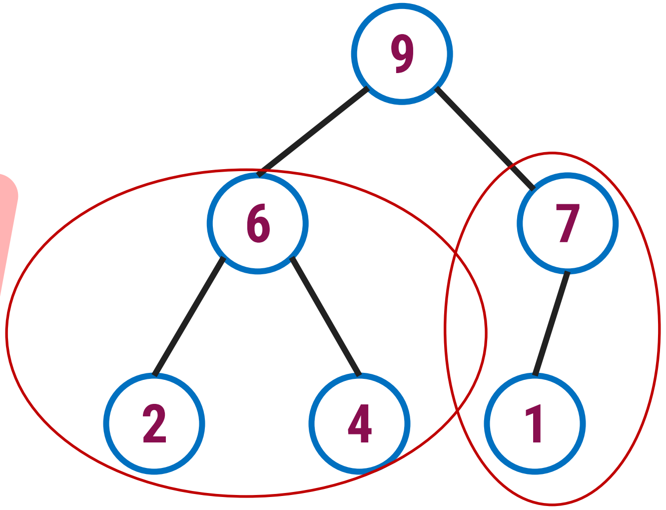
Binary Tree but not a Heap



Complete Binary Tree - Heap



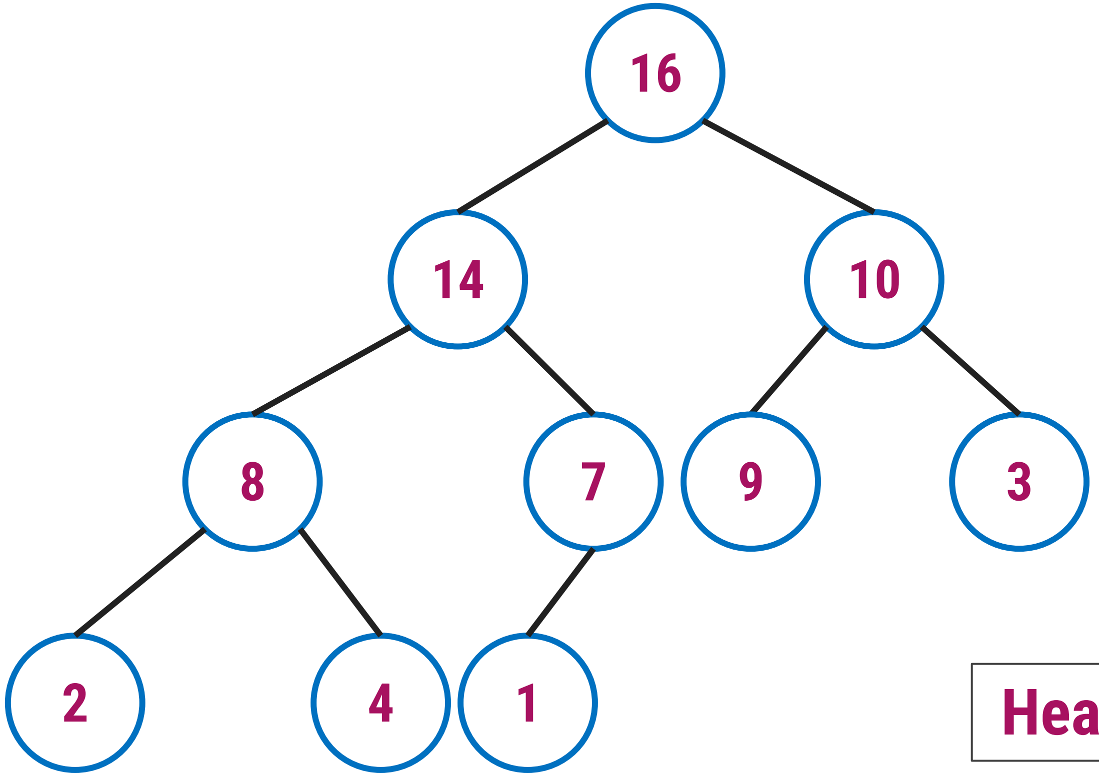
Not a Heap



Heap

Array Representation of Heap

- Heap can be implemented using an Array.
- An array A that represents a heap is an object with two attributes:
  1. $length[A]$, which is the number of elements in the array, and
  2. $heap\text{-}size[A]$, the number of elements in the heap stored within array A



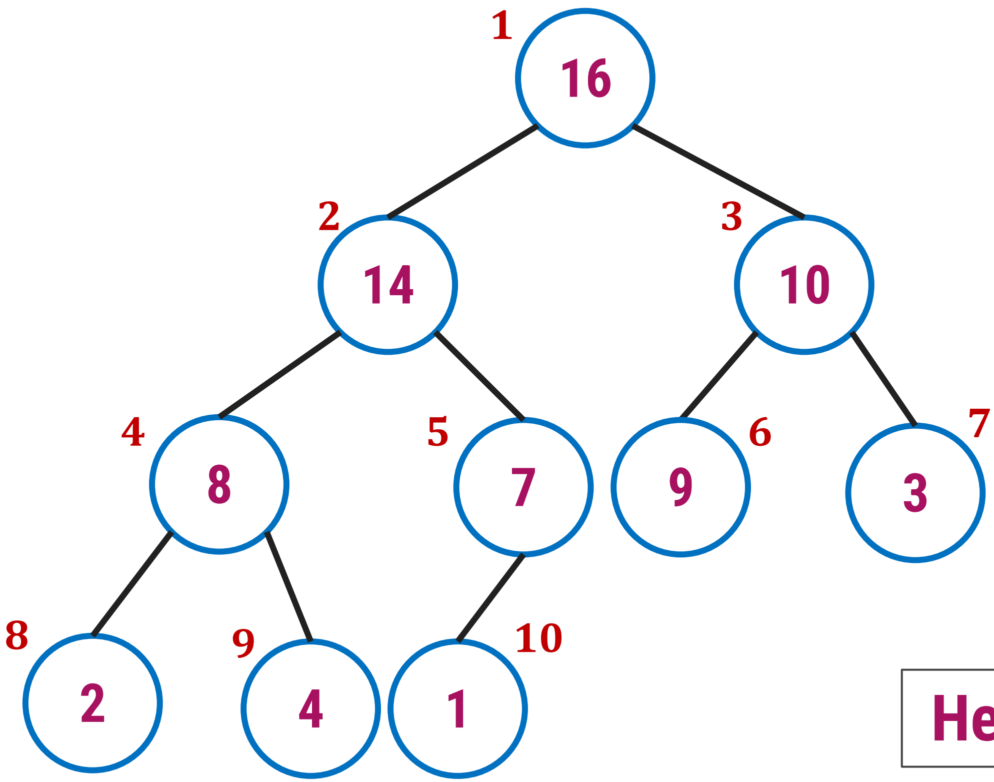
Heap

Array representation of heap

16	14	10	8	7	9	3	2	4	1

Array Representation of Heap

- In the array A, that represents a heap
  1. length[A] = heap-size[A]
  2. For any node $\boldsymbol{i}$ the parent node is $\boldsymbol{i/2}$
  3. For any node $\boldsymbol{j}$, its left child is $\boldsymbol{2j}$ and right child is $\boldsymbol{2j+1}$



Heap

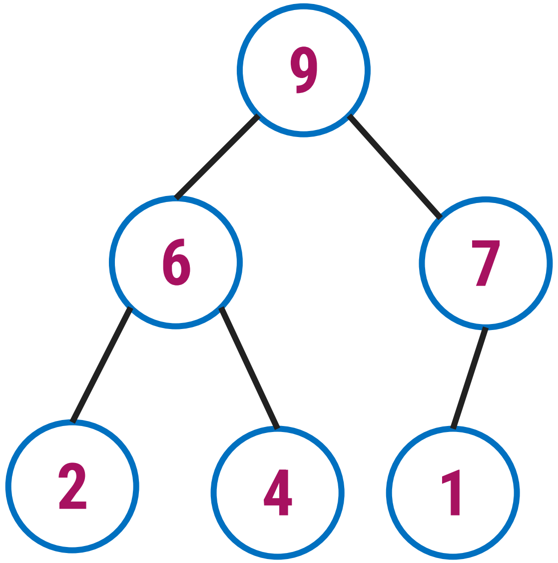
For node $i = 4$, parent node is $4/2 = 2$

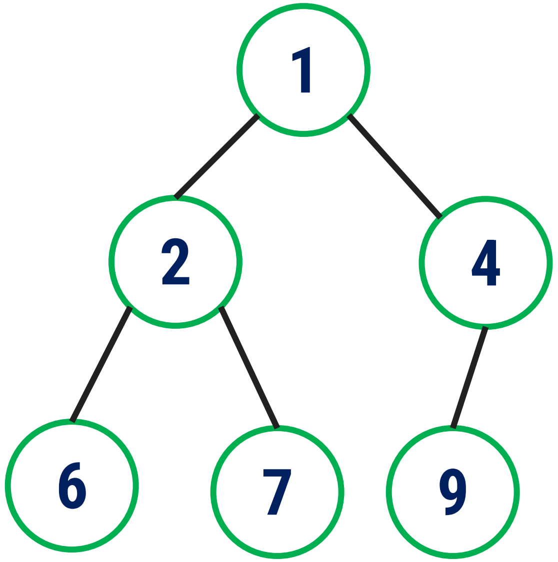
For node $i = 4$,
Left child node is $2 * 4 =$ node 8
Right child is $2 * 4 + 1 =$ node 9

1	2	3	4	5	6	7	8	9	10
16	14	10	8	7	9	3	2	4	1

Types of Heap

1. **Max-Heap** – Where the value of the root node is greater than or equal to either of its children.





2. **Min-Heap** – Where the value of the root node is less than or equal to either of its children.

Introduction to Heap Sort

1. Build the complete binary tree using given elements.
2. Create Max-heap to sort in ascending order.
3. Once the heap is created, swap the last node with the root node and delete the last node from the heap.
4. Repeat step 2 and 3 until the heap is empty.

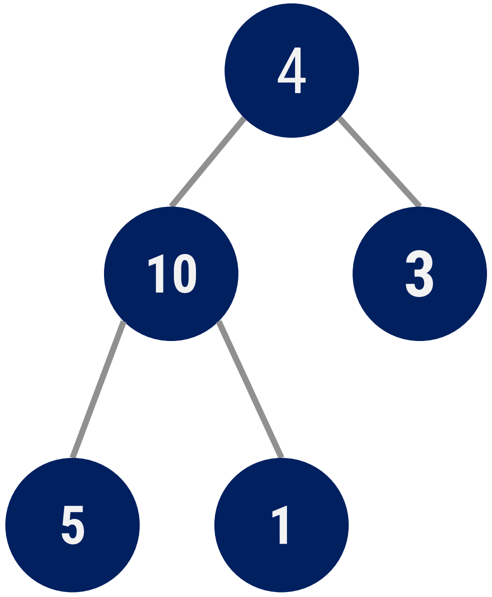
Heap Sort – Example 1

Sort the following elements in Ascending order

4	10	3	5	1

Step 1 : Create Complete Binary Tree

1	2	3	4	5
4	10	3	5	1



Now, a binary tree is created and we have to convert it into a Heap.

Heap Sort – Example 1

Sort the following elements in Ascending order

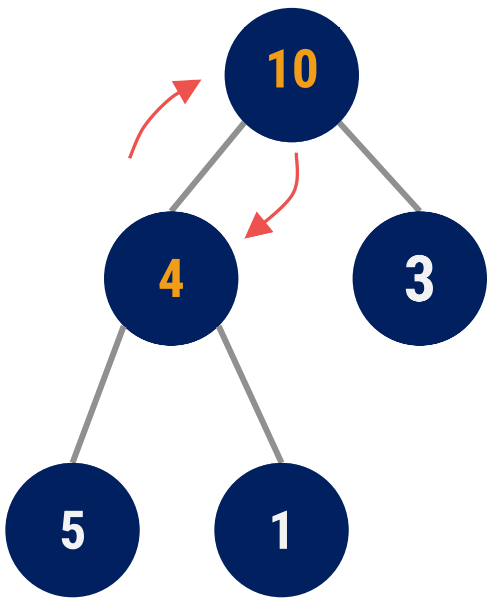
4	10	3	5	1

Step 2 : Create Max Heap

1	2	3	4	5
10	4	3	5	1

Swap

In a Max Heap, parent node is always greater than or equal to the child nodes.



10 is greater than 4
So, swap 10 & 4

Heap Sort – Example 1

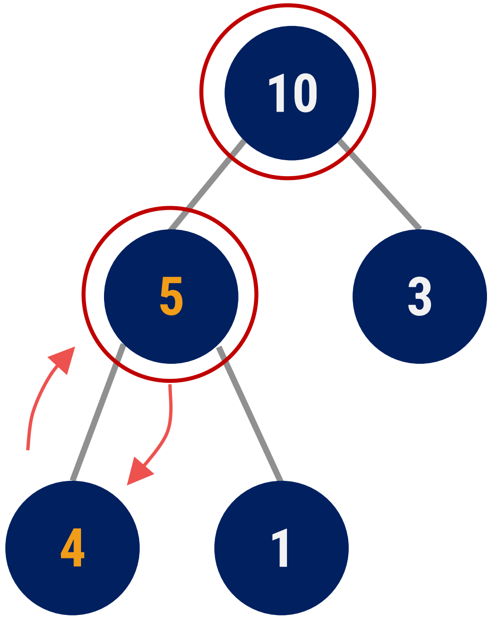
Sort the following elements in Ascending order

4	10	3	5	1

Step 2 : Create Max Heap

1	2	3	4	5
10	5	3	4	1

Swap



5 is greater than 4
So, swap 5 & 4

In a Max Heap, parent node is always greater than or equal to the child nodes.

Max Heap is created

Heap Sort – Example 1

Sort the following elements in Ascending order

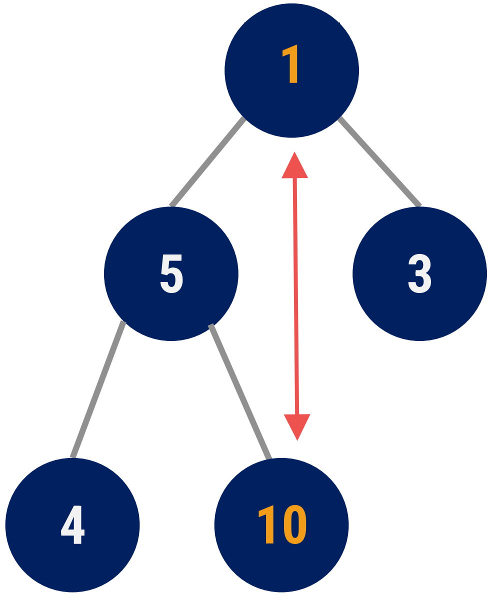
4	10	3	5	1

Step 3 : Apply Heap Sort

1	2	3	4	5
1	5	3	4	10

Swap

1. Swap the first and the last nodes and
2. Delete the last node.



Heap Sort – Example 1

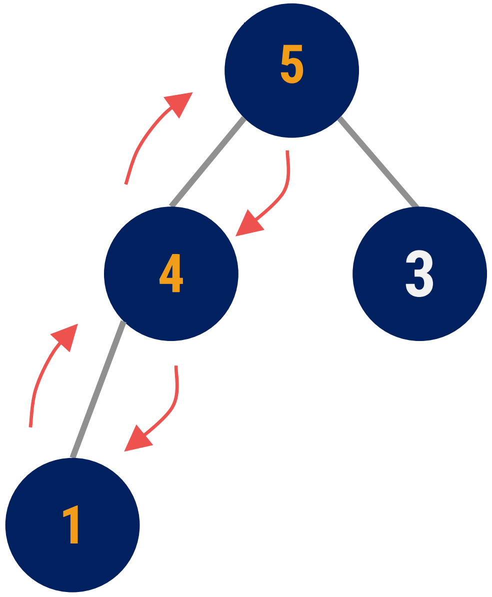
Sort the following elements in Ascending order

4	10	3	5	1

Step 3 : Apply Heap Sort

1	2	3	4	5
5	4	3	1	**10**

Swap



Max Heap Property is violated so, create a Max Heap again.

Heap Sort – Example 1

Sort the following elements in Ascending order

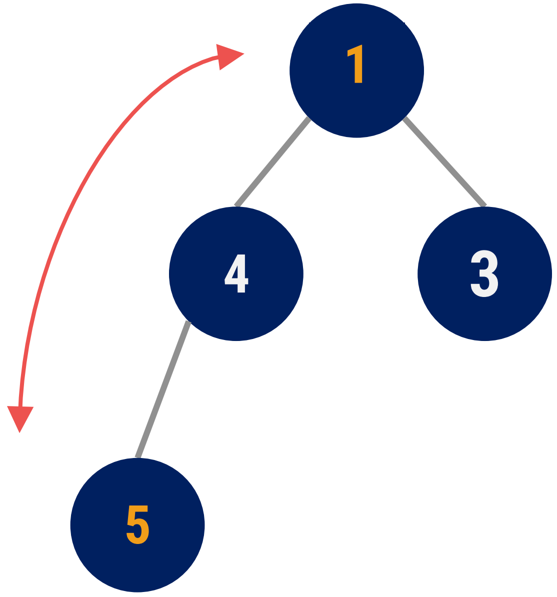
4	10	3	5	1

Step 3 : Apply Heap Sort

1	2	3	4	5
1	4	3	5	**10**

Swap

1. Swap the first and the last nodes and
2. Delete the last node.



Max Heap is created

Heap Sort – Example 1

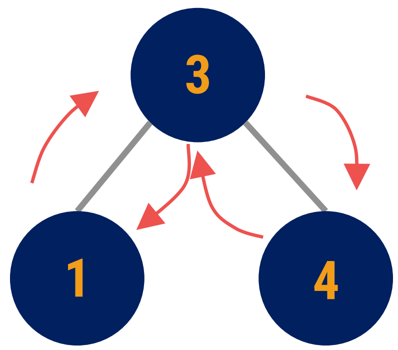
Sort the following elements in Ascending order

4	10	3	5	1

Step 3 : Apply Heap Sort

1	2	3	4	5
3	1	4	**5**	**10**

Swap



Create Max Heap again

Max Heap is created

1. Swap the first and the last nodes and
2. Delete the last node.

Heap Sort – Example 1

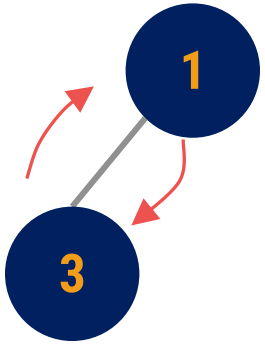
Sort the following elements in Ascending order

4	10	3	5	1

Step 3 : Apply Heap Sort

1	2	3	4	5
1	3	**4**	**5**	**10**

Swap



Already a Max Heap

1. Swap the first and the last nodes and
2. Delete the last node.

Heap Sort – Example 1

Sort the following elements in Ascending order

4	10	3	5	1

Step 3 : Apply Heap Sort

1	2	3	4	5
1	**3**	**4**	**5**	**10**

Already a Max Heap

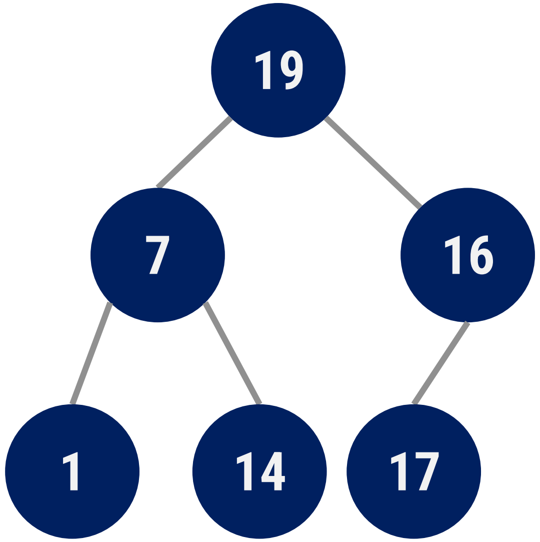
Remove the last element from heap and the sorting is over.

Heap Sort – Example 2

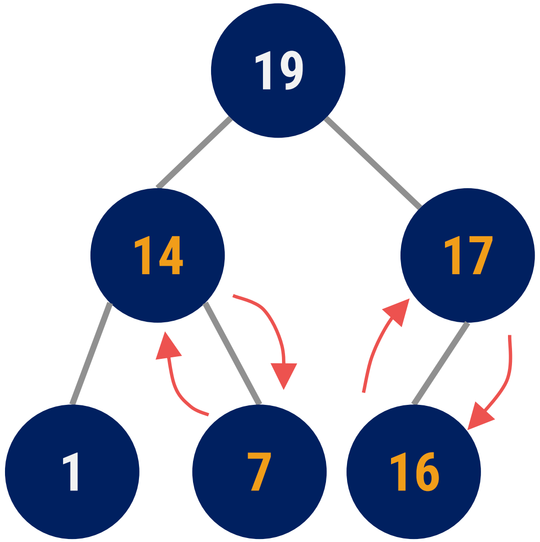
- Sort given element in ascending order using heap sort. 19, 7, 16, 1, 14, 17

19	14	17	1	7	16

Step 1: Create binary tree



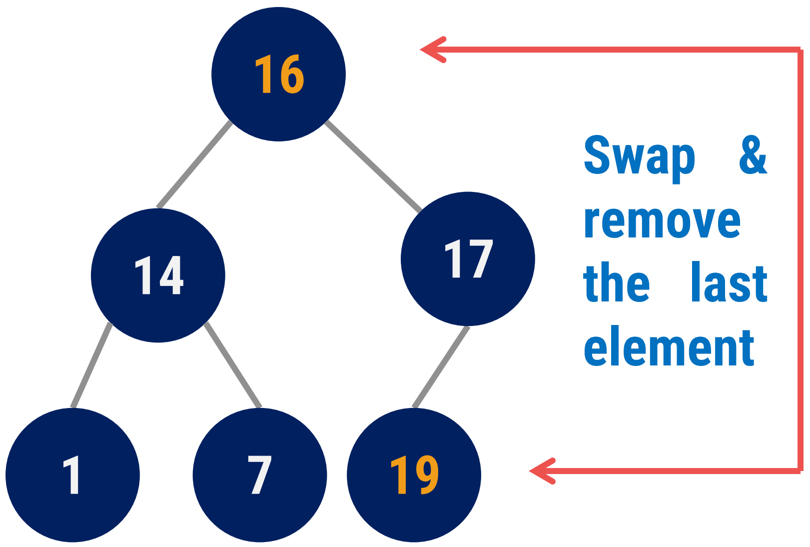
Step 2: Create Max-heap



Heap Sort – Example 2

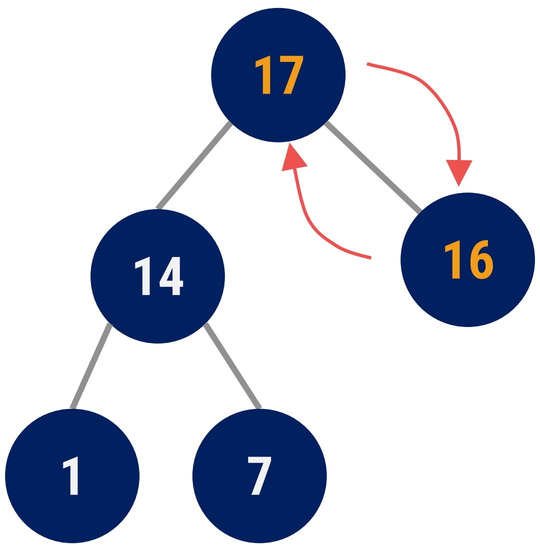
Step 3

16	14	17	1	7	19



Step 4

17	14	16	1	7	19

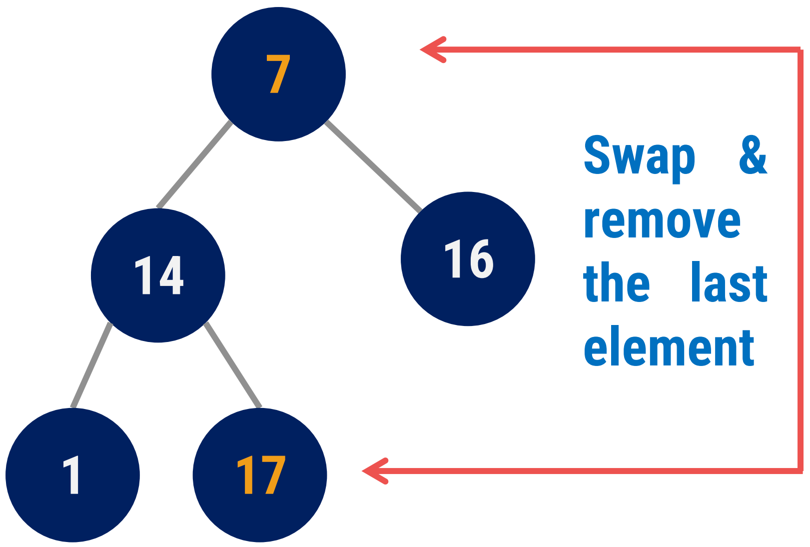


Create Max-heap

Heap Sort – Example 2

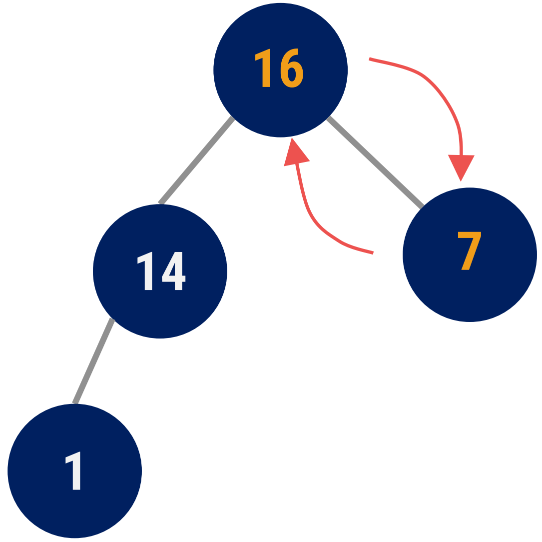
Step 5

7	14	16	1	17	19



Step 6

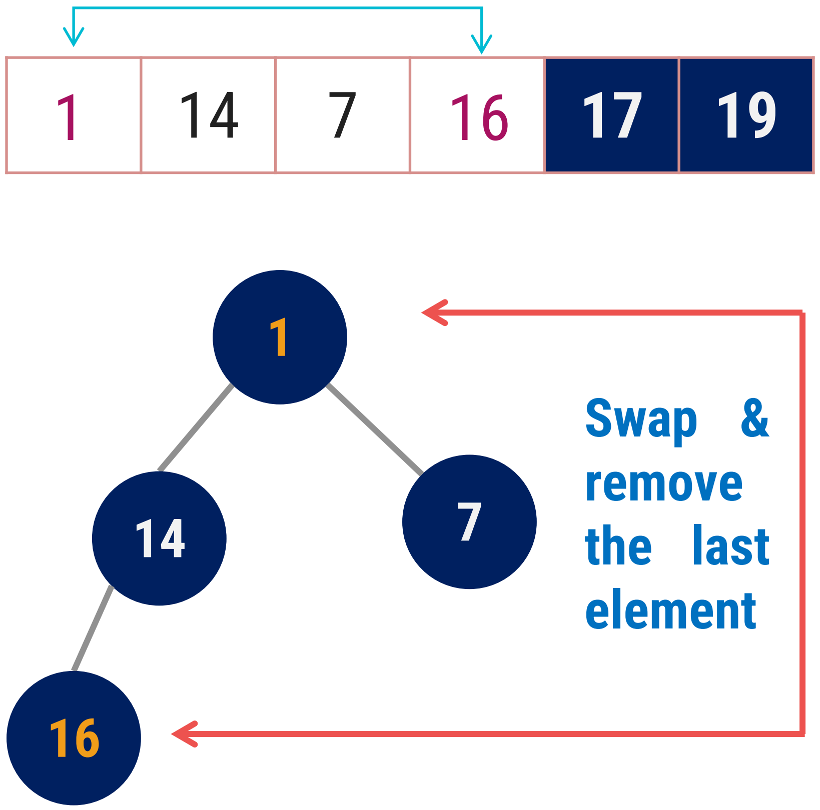
16	14	7	1	17	19



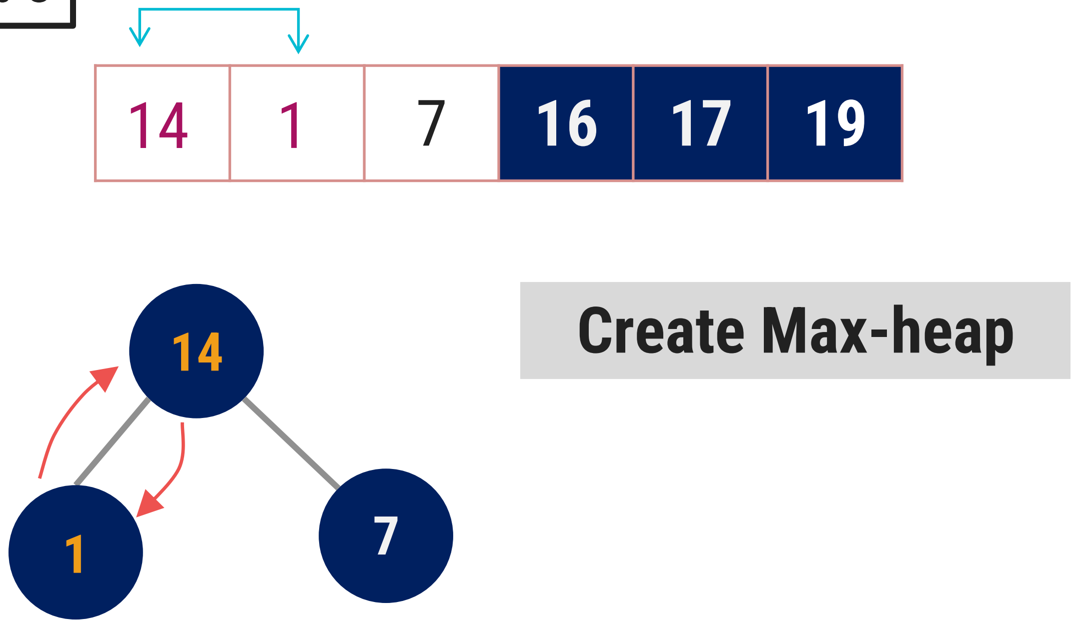
Create Max-heap

Heap Sort – Example 2

Step 7



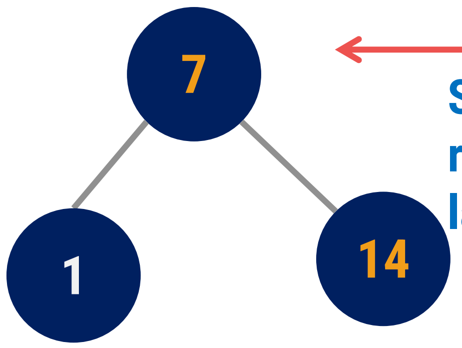
Step 8



Heap Sort – Example 2

Step 9

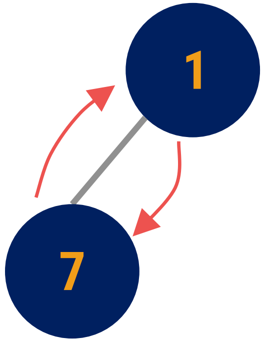
7	1	14	16	17	19



Swap & remove the last element

Step 10

1	7	14	16	17	19



Already a Max-heap

Swap & remove the last element

Step 11

1	7	14	16	17	19

Remove the

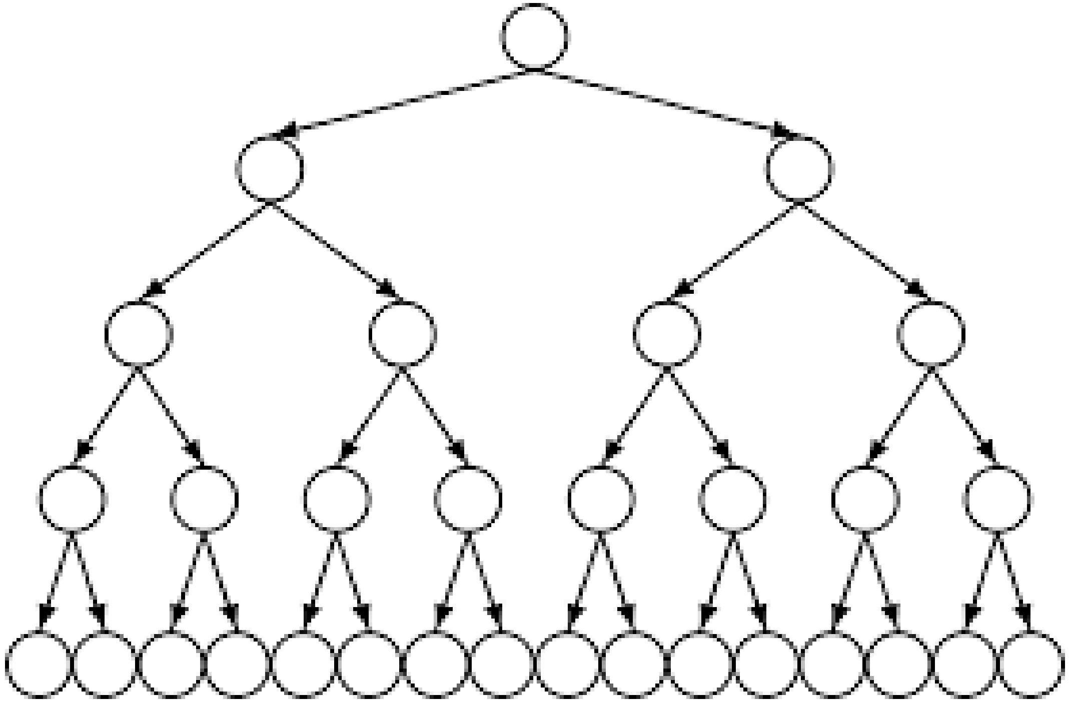
The entire array is sorted now.

Exercises

- Sort the following elements using Heap Sort Method.
  1. 34, 18, 65, 32, 51, 21
  2. 20, 50, 30, 75, 90, 65, 25, 10, 40

- Sort the following elements in Descending order using Hear Sort Algorithm.
  1. 65, 77, 5, 23, 32, 45, 99, 83, 69, 81

Binary Tree Analysis



Heap Sort – Algorithm

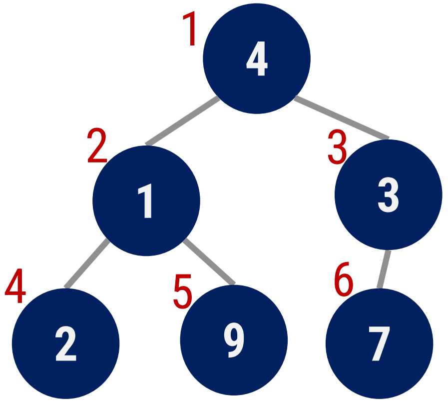
```
# Input: Array A
# Output: Sorted array A

Algorithm: Heap_Sort(A[1,…,n])
     BUILD-MAX-HEAP(A)
     for i ← length[A] downto 2
          do exchange A[1] ↔ A[i]
          heap-size[A] ← heap-size[A] – 1
          MAX-HEAPIFY(A, 1, n)
```

Heap Sort – Algorithm

```
Algorithm: BUILD-MAX-HEAP(A)
heap-size[A] ← length[A]
for i ← ⌊length[A]/2⌋ downto 1
    do MAX-HEAPIFY(A, i)
```

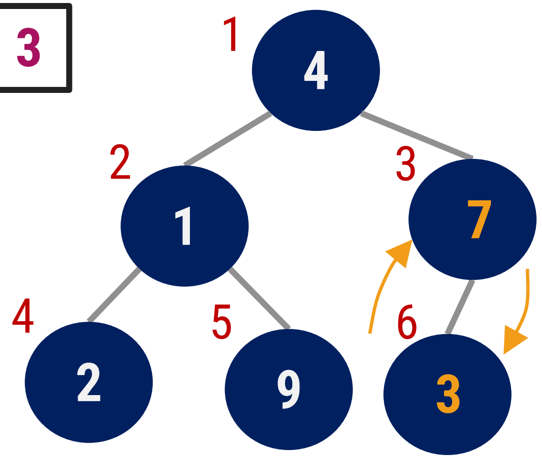
heap-size[A] = 6

4	1	3	2	9	7



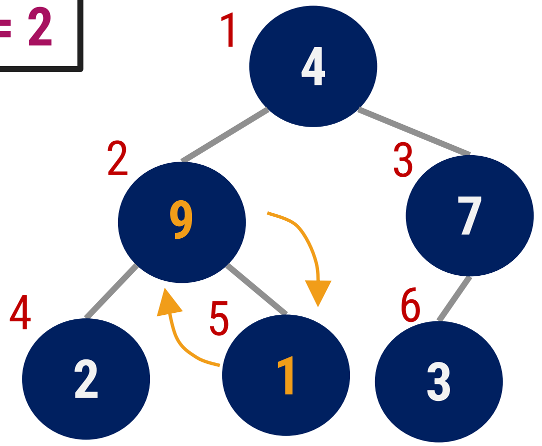
4	1	7	2	9	3

i = 3



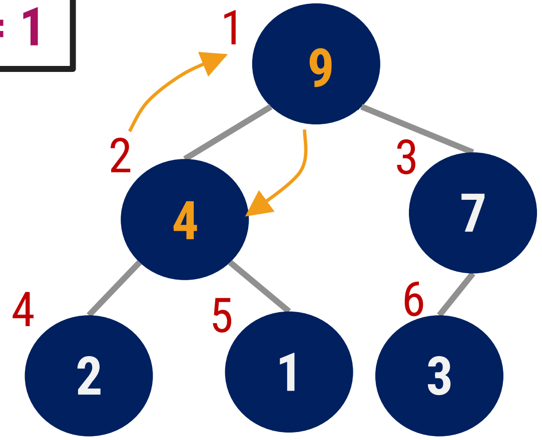
4	9	7	2	1	3

i = 2



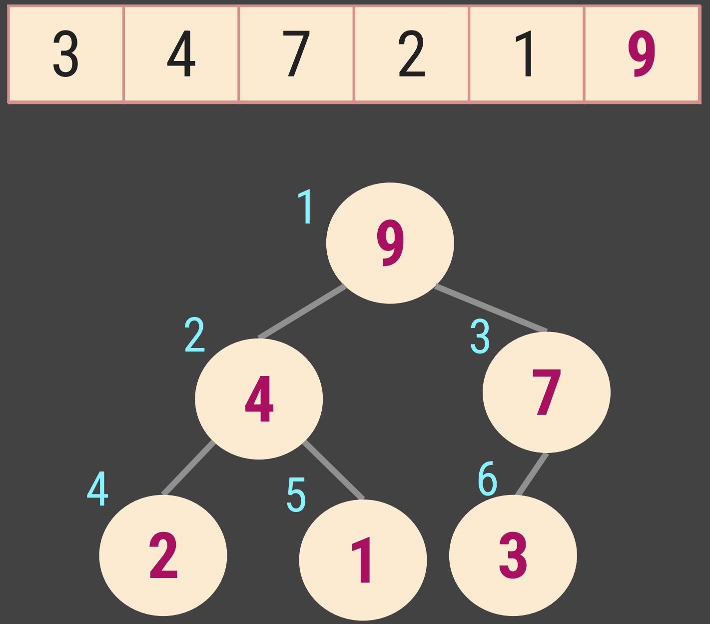
9	4	7	2	1	3

i = 1



Heap Sort – Algorithm

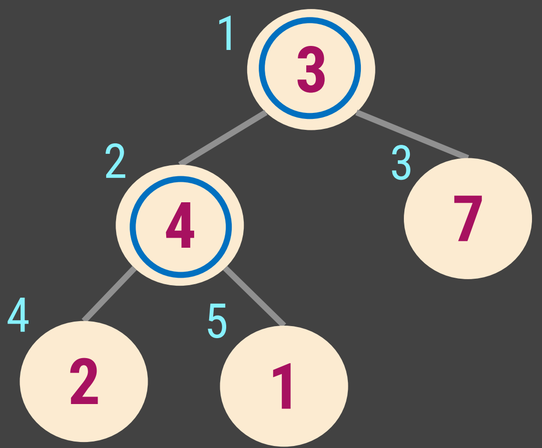
```
# Input: Array A
# Output: Sorted array A

Algorithm: Heap_Sort(A[1,…,n])
      BUILD-MAX-HEAP(A)
      for i ← length[A] downto 2
            do exchange A[1] ↔ A[i]
            heap-size[A] ← heap-size[A] – 1
            MAX-HEAPIFY(A, 1, n)
```



Heap Sort – Algorithm

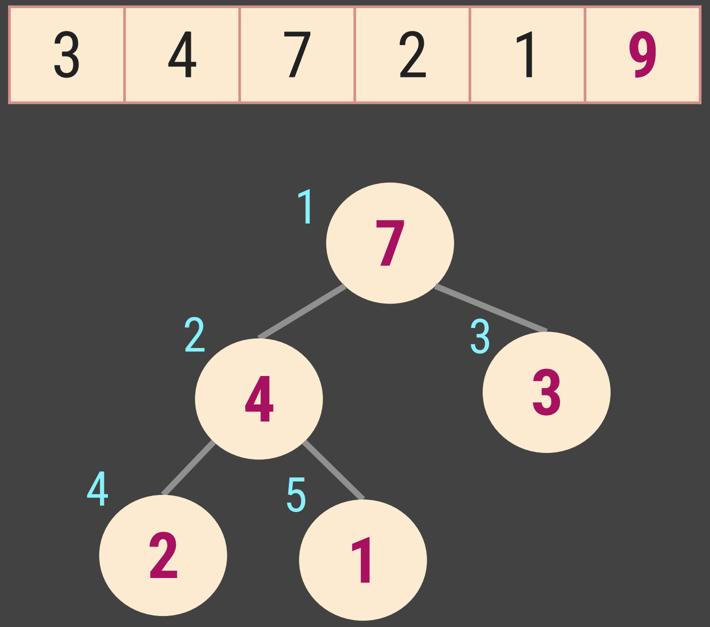
```
Algorithm: Max-heapify(A, i, n)      1
l ← LEFT(i)        l ← 2
r ← RIGHT(i)       r ← 3
if l ≤ n and A[l] > A[i]          Yes
    then largest ← l      largest ← 2
    else largest ← i
if r ≤ n and A[r] > A[largest]    Yes
    then largest ← r      largest ← 3
if largest ≠ i      Yes
    then exchange A[i] ↔ A[largest]
MAX-HEAPIFY(A, largest, n)
```

3	4	7	2	1	9



Heap Sort – Algorithm

```
# Input: Array A
# Output: Sorted array A

Algorithm: Heap_Sort(A[1,…,n])
     BUILD-MAX-HEAP(A)
     for i ← length[A] downto 2
          do exchange A[1] ↔ A[i]
          heap-size[A] ← heap-size[A] – 1
          MAX-HEAPIFY(A, 1, n)
```



Heap Sort Algorithm – Analysis

```
# Input: Array A
# Output: Sorted array A

Algorithm: Heap_Sort(A[1,…,n])
    BUILD-MAX-HEAP(A)
    for i ← length[A] downto 2
        do exchange A[1] ↔ A[i]
        heap-size[A] ← heap-size[A] - 1
        MAX-HEAPIFY(A, 1, n)
```

$O(n\ log\ n)$

```
heap-size[A] ← length[A]
for i ← ⌊length[A]/2⌋ downto 1
        do MAX-HEAPIFY(A, i)
```

$n/2$

$O(\ log\ n)$

$n-1$

$O(n-1)\ (log n)$

Running time of heap sort algorithm is:

$$O(n\ log\ n) + O(\log n)(n-1) + O(n-1) = \boxed{O(n \log n)}$$

Sorting Algorithms

Radix Sort, Bucket Sort, Counting Sort

Radix Sort

- Radix Sort puts the elements in order by comparing the digits of the numbers.
- Each element in the $\boldsymbol{n}$-element array $\boldsymbol{A}$ has $\boldsymbol{d}$ digits, where digit 1 is the lowest-order digit and digit d is the highest order digit.

```
Algorithm: RADIX-SORT(A, d)
 for i ← 1 to d
       do use a stable sort to sort array A on digit i
```

- Sort following elements in Ascending order using radix sort.

  363, 729, 329, 873, 691, 521, 435, 297

Radix Sort - Example

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

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

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

Sort on column 1

Sort on col

The entire array is sorted now.

Bucket Sort – Introduction

- Sort the following elements in Ascending order using bucket sort.

45	96	29	30	27	12	39	61	91

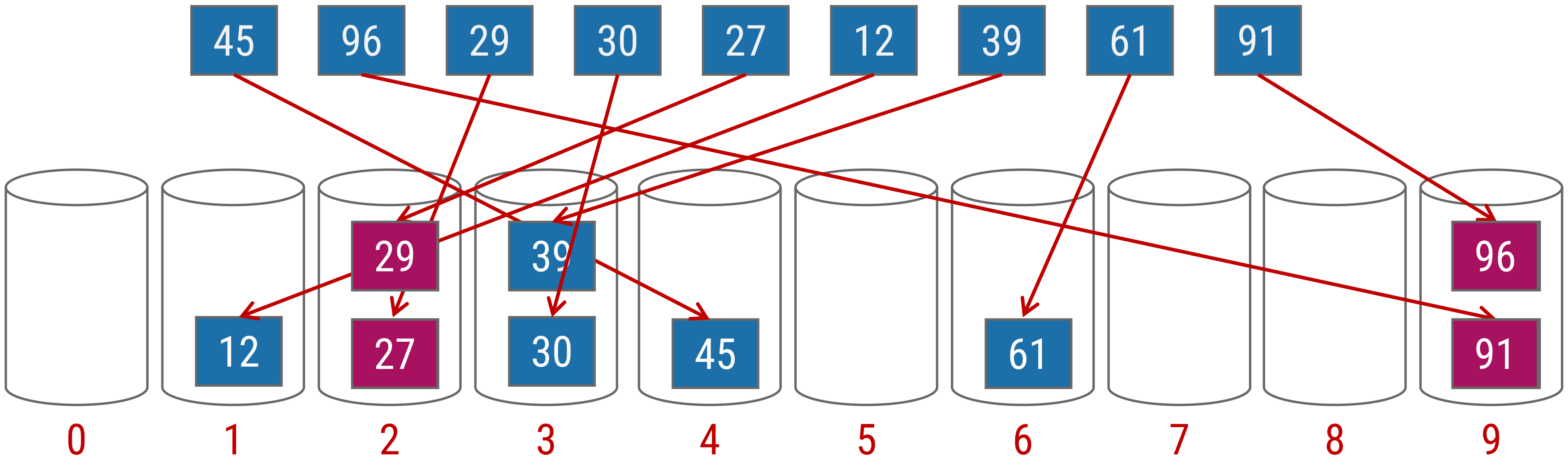
---

1. Create n empty buckets.
2. Add each input element to appropriate bucket as,
   a. Bucket i holds values in the half-open interval,
   $$\boldsymbol{i * 10 \le A[i] < (i+1) * 10}$$
3. Sort each bucket queue with insertion sort.
4. Merge all bucket queues together in order.

- Expected running time is $O(n + N)$, with n = size of original sequence. If N is $O(n)$ then sorting algorithm in $O(n)$.

Bucket Sort – Example

45	96	29	30	27	12	39	61	91



Sort each bucket queue with insertion sort

Merge all bucket queues together in order

Bucket Sort - Algorithm

```
# Input: Array A
# Output: Sorted array A
Algorithm: Bucket-Sort(A[1,…,n])
     n ← length[A]
     for i ← 1 to n do
          insert A[i] into bucket B[⌊A[i]÷n⌋]
     for i ← 0 to n – 1 do
          sort bucket B[i] with insertion sort
     concatenate the buckets B[0], B[1], . . ., B[n - 1] together in
     order.
```

Counting Sort – Example

- Sort the following elements in Ascending order using counting sort.

3	6	4	1	3	4	1	4	2

Step 1 Given elements are stored in an input array $A[1, \ldots, 9]$

Index	1	2	3	4	5	6	7	8	9
Elements	3	6	4	1	3	4	1	4	2

Step 2 Define a temporary array C. The size of an array C is equal to the **maximum element** in array A. Initialize $C[1, \ldots, 6]$ to 0.

Index	1	2	3	4	5	6
Elements	0	0	0	0	0	0

Counting Sort – Example

- Sort the following elements in Ascending order using counting sort.

Input array A									
	3	6	4	1	3	4	1	4	2

Step 3 Update an array C with the occurrences of each value of array A

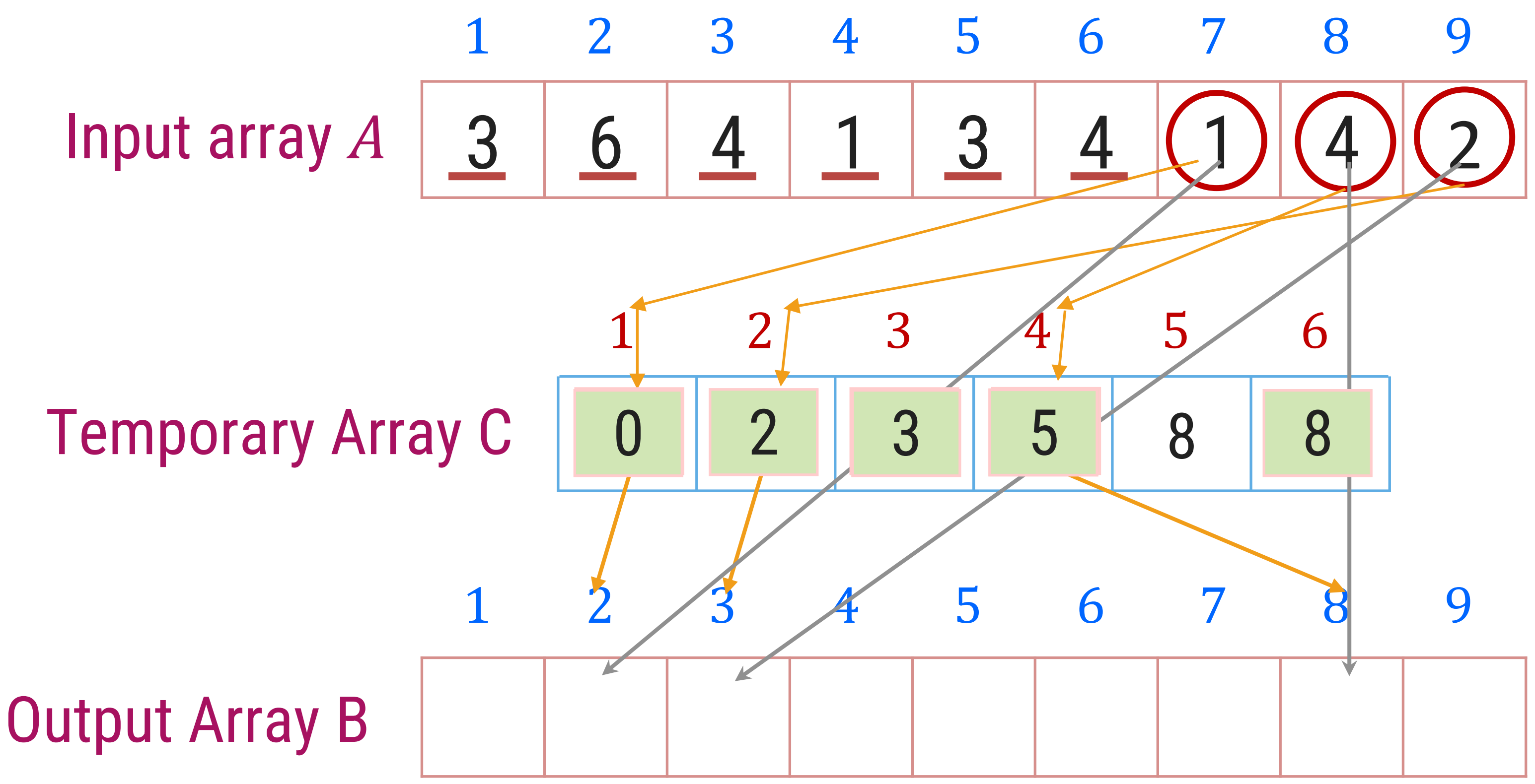
Index	1	2	3	4	5	6
Elements	**2**	**1**	**2**	**3**	**0**	**1**

Step 4 In array C, from index 2 to n add the value with previous element

Index	1	2	3	4	5	6
Elements						

Counting Sort – Example

- Create an output array $B[1 \ldots 9]$. Start positioning elements of Array A to B as shown below.

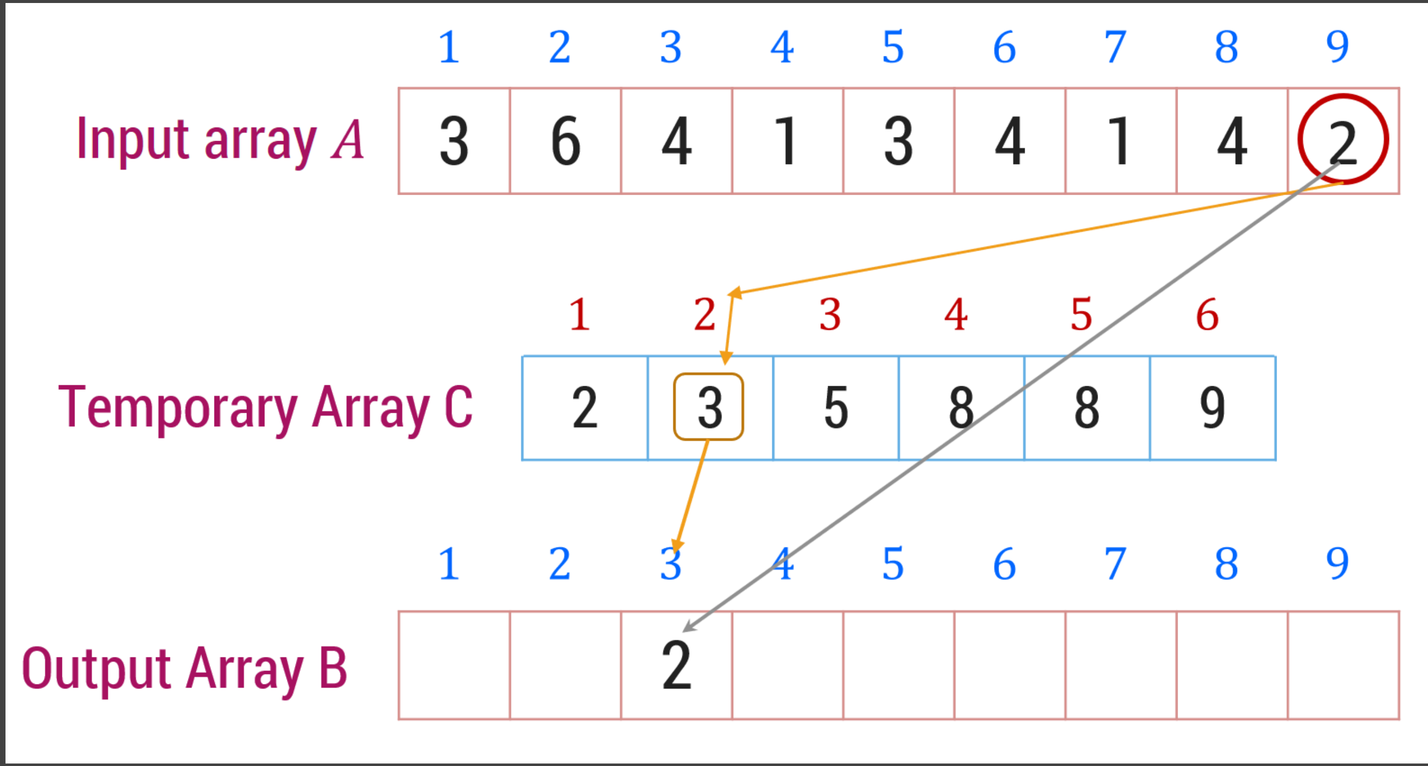


Counting Sort - Procedure

- Counting sort assumes that each of the n input elements is an integer in the range 0 to k, for some integer k.
- When $\boldsymbol{k}$=$\boldsymbol{O}(\boldsymbol{n})$, the counting sort runs in $\boldsymbol{\theta}(\boldsymbol{n})$ time.
- The basic idea of counting sort is to determine, for each input element x, the number of elements less than x.
- This information can be used to place element x directly into its position in the output array.

Counting Sort - Algorithm

```
# Input: Array A
# Output: Sorted array A
Algorithm: Counting-Sort(A[1,…,n], B[1,…,n], k)
for i ← 1 to k do
    c[i] ← 0
for j ← 1 to n do
    c[A[j]] ← c[A[j]] + 1
for i ← 2 to k do
    c[i] ← c[i] + c[i-1]
for j ← n downto 1 do
    B[c[A[j]]] ← A[j]
    c[A[j]] ← c[A[j]] - 1
```



Thank You!